# Systems Programming & Computer Architecture
# HS2023 Essentials

Simon Sure
info@simonsure.com / simon.sure@inf.ethz.ch

February 3, 2024

This is an unofficial script containing information from the Systems Programming & Computer Architecture course as taught in HS of 2023. Although I am fairly confident that this contains everything mentioned in the lecture videos, ... - there is no guarantee for correctness of completeness!

If you have remarks or questions, or find any mistakes, don't hesitate to send me a mail to info@simonsure.com.

Also, feel free to share this script via https://simonsure.com/ethz which will always contain a link to the newest version. Use the date of this document to identify the newest version.

# Contents

# 1 Introduction

Systems Programming describes programming on and above the hw/sw boundary. With systems, one always has to balance between theory and engineering (practical use). Specifically, we consider programs in the context of the real-world. While much of CS assumes many abstractions (data types, algorithms as mathematical objects, performance as asymptotic analysis, ...) systems recognizes the limitations of this as those abstractions do not really match reality, which can lead to bugs etc. when not having the proper knowledge. Systems Programmer understand the underlying reality to enable optimizations by recognizing that programs are more than just the algorithmic ideas.

Among the specialties which we will discuss in the following are:

- Computers don't really deal with numbers. Integers have an upper and lower bound. And floating point numbers are discrete.

- Understanding and writing (limited) assembly is important to fully grasp performance behavior and enable special functionality not accessible through a programming language.

- Memory is not a nice array that just stores data. It is finite. It must be managed (allocated & freed). There are caches, prefetching, virtual memory, ... Those all are deviations from the ideal simplified understanding. Performance depends on those factors. Adapting a program to the given environment can yield great improvements.

- Performance is about much more than asymptotic complexity. We must consider different latencies of different instructions, vector instructions, loop orders, ... Despite having the same asymptotic complexity, such simple changes may increase program performance by 160x.

- Computers don't just execute programs. They are a very complex and intricate piece of machinery. There is the Operating System, external devices (I/O), concurrent operations, different platforms, ... The generic programmer only knows a small fraction of the whole system. But knowing the entire systems structure enables a great systems programmer to unlock additional performance.

- Programs are not semantic specifications. Much behavior is implementation defined and we rely on the compiler to map operations in the program to the most natural operations in hardware.

Another reason for the relevance of Systems Programming is that Moore's Law does not continue to imply an increase in single-core performance as it has done in the past. To combat this decline, more specialized compute units (GPUs, TPUs, NPUs, ...) are developed. Programming those is a systems software problem.

# 2 Introduction to C

This course discusses C on a practical level. There won't be a formal introduction or formal definitions. Instead, we see the different properties of C introduced through example and applications.

## 2.1 History

C is an old language. It was developed quickly by Dennis Ritchie with the purpose to reimplement the Unix kernel in C instead of Assembly as it had been done before. At that time, C was considered a high-level language. Many characteristics of (even modern) C can be traced back to the properties of architecture used at the time of the inception of C.

The C language was influenced by other earlier languages, which preceded C: CPL (Combined Programming Language), BCPL (Basic Combined Programming Language), and B. CPL was so complex that nobody ever managed to implement a compiler for it.

By its design, C was portable across many architectures. Accordingly, there were numerous debates on the C standard. As C was and is used on many different platforms, it is often difficult to agree on a shared definition for a standard. Hence, many features of C are stated to be implementation-defined. Chronologically, those are the relevant C standards:

- K&R C (the standard was not defined in plain English, instead the compiler was the standard)
- ANSI C
- C99 (from 1999, will be used in this course)
- C11
- C17
- C23 (?) to be released in 2024

## 2.2 Characteristics of C

Compared to other languages, C is very fast. The reason that C is so fast is that (a) it is close to the hardware and writing C code is hence very optimized for the used hardware, and (b) C compilers are very, very sophisticated. It is almost impossible to write as good assembly code manually.

For the above reasons (speed and hardware closeness) C is the language of choice for OS developers, embedded systems, speed-critical applications, security exploits, ... despite its age. Moreover, C is a very simple language by itself. Simple not meaning it is simple to write good C code, but that the count of language constructs is rather small. There are no objects, classes, traits, features, methods, interfaces, ... We only really have functions, variables, and pointers. There are also no fancy build-in types. We mostly deal with what the hardware provides. Boolean are just integers with an easier to read name, for example. Exceptions do also not exist. Nevertheless, objects, polymorphism etc. can still be used in C because those concepts can of course be implemented.

Because C is so close to the hardware, writing C programs is about caring about data organization. C is about directly building and manipulation structures in main memory. In C, one must manually manage the memory. That unlocks great potential for optimization. We mostly store things on the stack. When we want to use the heap, we need to manually allocate memory and free it after we have used it as there is no garbage collection. Pointers (weakly types) enable us to directly access memory addresses.

While those concepts are different from Java, JavaScript, C++, and other languages, the syntax is fairly similar. Comments, identifiers, blocks,. . . , they all work basically the same. Still, as C has less language constructs, the list of reserved words is different. Also, there is a powerful macro pre-processor, which is used for string and file substitution, conditional compilation, etc. Such pre-processors are not present in other relevant languages.

## 2.3 C Program Structure

Listing 1: Simple C Program

```c
int main() {
  return 0;
}
```

Without relying on default type definitions etc., one will struggle to write a shorter C program. But for illustratory purposes we will consider a slightly more elaborate program.

Listing 2: Basic C Program

```c
#include <stdio.h>

int main(int arc, char *argv[])
{
  printf("hello,␣world\n");
  return 0;
}
```

- The `#include<...>` can be understood as an import. The `#` indicates a pre-processor instruction. What is actually done is that the contents of the specified file, here the stdio.h header file, is pasted to replace the include during compilation.
- The `main()` function is the entry point of the program. Its arguments are the command line arguments. `argc` specifies the number of arguments and `*argv[]` is an array of strings.
- `printf()` is a generic function for printing formatted strings. The `\n` is required to add a line break. This function can only be used when importing `stdio.h`.

In C, the program always returns an integer. By convention, 0 tells the operating system that the program terminated successfully. Anything else indicates some sort of error. One can use different integers for different errors. But notice that also returning `void` will likely compile correctly due to compiler defaults.

## 2.4 Toolchain/Workflow

If the reader is familiar with languages such as `Python`, one probably knows that it is an interpreted language. During execution the program is interpreted and executed at runtime. C (and other languages such as C++) are different. Those are compiled languages. Before execution, C is compiled into assembly, which can then directly be executed. This removes a runtime performance overhead.

Compilation is done in different steps. In the early times of C, the linking as part of loading did not exist. But today, the assembly being executed is different from the binary assembly on disk. Dynamic linking (linking during load) is discussed in the section on linking.

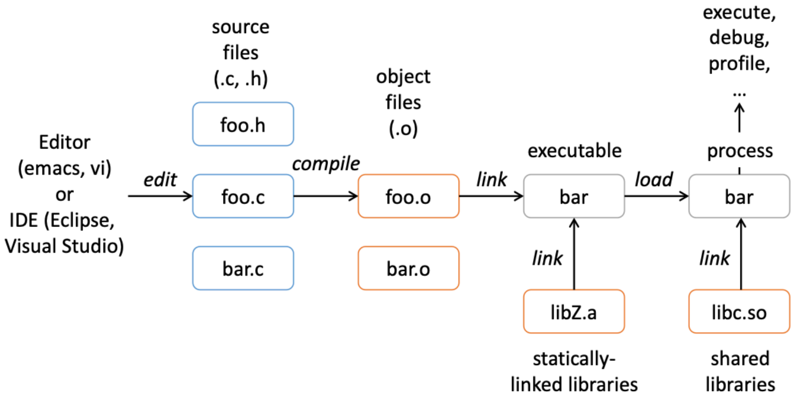


Figure 1: C Workflow

1. Editing the raw .c files.
2. Pre-processing as part of compilation with `cpp` generates new .c files.
3. The pre-processed .c files are compiled into assembly using `cc1`. The assembly is still stored in human-readable form in .s files. This finishes the compilation step.
4. The .s files are then assembled into .o object files. This now corresponds to assembly as defiend by the ISA and is not human friendly to read. This is done with the `as` tool.
5. Various object files and statically linked libraries are then linked to create an executable using `ld`. Executables often have the ending .a.
6. During execution, the loader loads an executable and finishes to link dynamically linked libraries.

The different steps of this pipeline could be done manually. But in practice, we use `gcc`, which combines all the above steps into a single tool. Gcc stands for GNU Compiler Collection. With different flags, one can instruct `gcc` to only perform some of the steps outlined above.

- `gcc -S`: Up to including step 3. Results in human-readable assembly.
- `gcc -c`: Up to including step 4. Results in non-human-readable machine code.
- `gcc`: Up to including step 5. This creates the executable. With `-o` once can specify the name of the output file. On Unix/Linux it is `a.out` by default.

## 2.5 Control Flow

`if`, `switch`, `for`, `while`, `do-while`, `return`, ... all have basically the same semantics and syntax as in Java or C++. Hence, we will not elaborate them here. Notice that for `switch` we need a `break;` in each case if we would rather not fall through and continue executing the next case. We also have `break` and `continue`. But there are no labels, so we can always only affect the innermost loop.

Also, we have `goto`, which we can use to jump to a specific label, i.e., point in the code. This is a remnant from early programming. It should only be used in a few specific cases. Using `goto` almost never increases program performance and it is very confusing to understand programs using C.

## 2.6 Functions

Functions work mostly as in Java/C++. We won't explain them explicit.y

Listing 3: Functions in C

```
returnType fucntionName(parameterType parameterName, /* ... */ )
{
  // ...
  return returnType;
}
```

## 2.7 Basic I/O

Using I/O to interact with devices is complicated. It will be discussed in the last section. But more basic I/O is printing to the console and reading input from the console. To print to the console we use `printf();` which may only be used when importing `stdio.h`. It is a variadic function, meaning it accepts a variable number of arguments.

The first argument is the format string. The remaining arguments are arbitrary but must fit the what we indicated to insert into the format string. Inside the format string, we use `%d` to insert a decimal, `%s` to insert a string, ... The following arguments then specify those values in the corresponding order.

## 2.8 Basic Types

Variable declaration works as known: `type name;`. Together with initialization: `type name = value;`. Furthermore, we may add additional keywords such as `static` or `volatile`. The latter is discussed as part of multiprocessing.

| | **inside a block/function** | **outside a block/function** |
|---|---|---|
| **static** | value persists between calls | scope limited to compilation unit |
| **non-static** | scope is just the block | scope is the entire program |

Figure 2

The compilation unit is what is being compiled after the pre-processing. I.e., the unit that is actually being compiled and not all files that are separately compiled but only combined during linking.

Besides basic types as `int`, `float`, `double`, `char`, ..., we also have `void`. It has no value and is used for untyped pointers (`void *`) or declaring functions with no return value. `char` can be signed or unsigned. It is basically a byte of memory. `float` is always signed. Integer types can be signed or unsigned.

We usually distinguish between integers and floating point types. Chars and booleans are also just integers under the hood. Thus, we can all discuss them as being the same. They only correspond to different sizes/amounts of bits in memory. Integers are stored in 2s complement format. Floating point types will be discussed in a separate section.

Booleans are integers as they are an integer-type. In basic C there even isn't a bool type. We just use small ints. A 0 is interpreted as false and anything else means true. One can use `!` to convert true/false to false/true. When one wants to use a type bool, one has to import `stdbool.h`.

### 2.8.1 Type Width

The width (i.e., size) of types is not defined in the C standard. It is implementation-defined. However, it is defined that integers are signed by default.

| C Data Type | typical 32-bit | ia32 (intel x86) | intel x86-64 |
|---|---|---|---|
| char | 1 | 1 | 1 |
| short | 2 | 2 | 2 |
| int | 4 | 4 | 4 |
| long | 4 | 4 | 8 |
| long long | 8 | 8 | 8 |
| float | 4 | 4 | 4 |
| double | 8 | 8 | 8 |
| long double | 8 | 10/12 | 10/16 |

Figure 3: Default Type Widths in Bytes

”Native” types are accessible by just writing `int` or similar. However, one may import `stdint.h`, which provides additional type definitions. This is also helpful, because `int` may have different sizes on different platforms. But with the definitions from `stdint.h` one can specify specific sizes etc. We may use types such as `uint_32, uint_8, int_64, ....`

### 2.8.2 Conversion/Casting

Between different integer-type types we have ’implicit’ conversion in a sense that we just reinterpret the bits without changing them. If we change the size of the integer, more interesting things happen. Reducing the size just corresponds to truncation, i.e., only the unaltered fitting lower bits are kept. (For a signed original int, this technically is implementation defined.) Increasing the size differs depending on whether the original value is signed or unsigned. If the original value is signed, the new value will be sign extended. If the original value is unsigned, the new value will be zero extended.

Between integers and floating-point types are also is implicit conversion. But that conversion actually changes the bit representation of a number so that it is best approximated in the other number type. In the floating point section the details of this conversion will be discussed in detail. Similarly, one can also implicitly convert between floating point types.

## 2.9 Statements as Expressions

In C, any statement is also an expression. Assignments etc. all evaluate to a value. Assignments specifically, evaluate to the new/assigned value.

## 2.10 Operators

As with other properties too, operators in C are similar in other languages such as C++ or Java as those are heavily inspired by C. For all operators, we shall remember their associativity, i.e., whether it is left-to-right or right-to-left associative. If there are multiple occurrences of the same operator, right-to-left means that we start evaluating on the right, while left-to-right means that we start evaluating on the left.

The following list shows operators in C in order of their presedence.

- `(), [], ->` are left-to-right

- unary operators: `!,~,++,--,+,-,*,&,(type),sizeof` are right-to-left
- binary operators: `*,/,%` are left-to-right
- binary operators: `+,-` are left-to-right
- `<<,>>` are left-to-right
- `<,<=,>,>=` are left-to-right
- `==,!=` are left-to-right
- `&` is left-to-right
- `^` is left-to-right
- `|` is left-to-right
- `&&` is left-to-right
- `||` is left-to-right
- `?` (ternary operator) is right-to-left
- `=,+=,*=,/=,%=,&=,^=,|=,<<=,>>=` are right-to-left
- `,` is left-to-right

Almost all of those operators should be known. Only `,` may be unfamiliar as it is not common in other languages. It can chain two instructions together but discard the value of the left expression as it returns the value of the right expression.

Notice the difference between pre- and post-increment and -decrement.

## 2.11 Arrays

Arrays are finite vectors of variables of the same type. An N-element array is indexed from 0 to N-1. In contrast to high-level languages, the C compiler does not check for array bounds.

In memory, an array just is a contiguous block of memory. With $N$ elements, we have `N * sizeof(type)` contiguous types, where the array elements are just lined up. Indexing the array corresponds to specifying an offset from the first element. If we give some bogus offset (too large, some negative number, ...) we just offset to some memory location which has nothing to do with the array. This may lead to program failure.

Listing 4: C Array Example

```c
#include <stdio.h>

//creating array
float data[5];

int main()
{
  data[0] = 34.0;
  data[1] = 27.0;
  data[2] = 45.0;
  data[3] = 82.0;
  data[4] = 22.0;
  total = data[0] + data[1] + data[2] + data[3] + data[4];
  return (0);
}
```

We can also create multi-dimensional arrays.

C is row major. So access pattern should look like `arr[0][0],arr[0][1],...,arr[1][0],arr[1],[1],...`. Always remember row-majority when iterating over arrays!

Listing 5: Multi-Dimensional Arrays in C

```c
int array[3][3];
for(int i = 0; i<3; i++) {
  for(int j = 0; j<3; j++)
    array[i][j] = i+j;
}
```

For multi-dimensional arrays in C there also is another option. Instead of storing all elements contiguously, one may also store a pointer to the row for each row. So we basically have an 'outer' array `void* array[length]` and length many inner arrays `type inner[length2]` with length2 being the size in the second dimension. This has fundamentally different memory (latency) behavior. What's most confusing about this in practice is that both types of arrays are indexed in the same way.

Finally, arrays can be initialized while being created:

Listing 6: Array Initialization during Declaration in C

```c
int a[3] = {3, 7, 9};
int b[3][3] = {
  { 1, 2, 3 },
  { 4, 5, 6 },
  { 7, 8, 9 },
};
```

## 2.12 Strings

In C there isn't a special type for strings. Instead, strings are just arrays or characters terminated with a null byte `0x0` or `\0` (this is different from 0 itself!). So strings are `char string[length];`. Thus, in a string, i.e., a char array, not an arbitrarily long text can be stored. It can be at most length-1 long as we still need to add the null byte. Forgetting the null byte is fatal because the null byte stops functions to continue processing the string. If it is missing, they will likely run beyond the bounds of the char array.

To then use strings as one would use strings in other languages (copying, concatenation, getting the length, ...) one has to import `string.h`.

Listing 7: C String Example

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char name1[12], name2[12];
  char mixed[25], title[20];

  //strcpy(name1, "Rosalinda");
  strncpy(name1, "Rosalinda", sizeof(name1)-1);
  //strcpy(name2, "Zeke");
  strncpy(name2, "Zeke", sizeof(name2)-1);
  //strcpy(title, "This is the title.");
  strncpy(title, "This␣is␣the␣title.", sizeof(title)-1);

  printf("%s\n\n", title);
  printf("Name␣1␣is␣%s\n", name1);
  printf("Name␣2␣is␣%s\n", name2);

  /* returns 1 if name1 > name2 */
  if (strcmp(name1, name2) > 0) {
    //strcpy(mixed, name1);
    strncpy(mixed, name1, sizeof(mixed)-1);
  } else {
    //strcpy(mixed, name2);
    strncpy(mixed, name2, sizeof(mixed)-1);
  }
  printf("The␣biggest␣name␣alphabetically␣is␣%s\n", mixed);

  //strcpy(mixed, name1);
  strncpy(mixed, name1, sizeof(mixed)-1);
  //strcat(mixed, " ");
  strncat(mixed, "␣", sizeof(mixed)-strlen(mixed)-1);
  //strcat(mixed, name2);
  strncat(mixed, name2, sizeof(mixed)-strlen(mixed)-1);
  printf("Both␣names␣are␣%s\n", mixed);
  return 0;
}
```

We use `strncpy` instead of `strcpy` to avoid going over the bounds of the target buffer by specifying the maximum string size allowed.

- `strlen(char[])`: returns the length of a string
- `strncpy(char[], char[], size_t)`: copies the second array to the first array upu to a given length
- `strncat(char[], char[], size_t)`: concatenates the second string to the first string, adds at most the specified number of chars
- `strcmp(char[], char[])`: compares the two strings by comparing their chars one by one

  returns 0 if identical

  returns value $>$ 0 if: with first different char, the first string's char is larger in ASCII OR the first string is longer if no different up to then

  returns value $<$ 0 if: with first different char, the second string's char is larger in ASCII OR the second string is longer if no difference up to then

# 3 Representing Integers in C

## 3.1 Representing Numbers

In modern machines, integers are usually stored as several bytes. With little-endian systems, we store the least significant byte first/at the low address. With big-endian systems, we store the most significant byte first/at the low address.

Integers are just a sequence of bits. Their decimal value interpretation depends on whether we consider signed or unsigned integers. For unsigned integers, we consider standard binary representation:

$$\text{binary2unsigned}(x) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

For signed integers, we use two's complement:

$$\text{binary2signed}(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

| | 8 bit decimal | 16 bit decimal | 16 bit hexadecimal | 16 bit binary | 32 bit decimal | 64 bit decimal |
|---|---|---|---|---|---|---|
| UMax | 255 | 65,535 | FF FF | 11111111 11111111 | 4,294,967,295 | 18,446,744,073,709,551,615 |
| TMax | 127 | 32,767 | 7F FF | 01111111 11111111 | 2,147,483,657 | 9,223,372,036,854,775,807 |
| TMin | -128 | -32,768 | 80 00 | 10000000 00000000 | -2,147,483,658 | -9,223,372,036,854,775,808 |
| -1 | | -1 | FF FF | 11111111 11111111 | | |
| 0 | | 0 | 00 00 | 00000000 00000000 | | |

Figure 4: Integer Ranges

In C we can use `#include <limits.h>` to get declared constants with those values specific for our platform. Those constants would then be `ULONG_MAX, LONG_MAX, LONG_MIN , ...`

Integers are signed by default. If we want an unsigned value we need to specify it by appending a U as in: `0U`. We already discussed how casting (not) changes the bit representation. But another question is when casting occurs. One case is explicit casing. But when mixing different integer types, we also have implicit casting. Then, signed values are interpreted as unsigned values.

## 3.2 Number Operations

### 3.2.1 Bit-wise Operations

Notice that numbers in C are not numbers as in mathematics. They are `int`s, i.e., they are a sequence of bits. On those bits, we can perform bit-level operations. Such operations can be used on integer-type data types, i.e., everything not a float. We have `&` (AND), `|` (OR), `~` (NOT), and `^` (XOR).

### 3.2.2 Logical Operations

C has the logical operators `&&` (logic AND), `||` (logic OR), and `!` (logic NOT). They consider 0 to be false and anything non-zero as true and always evaluate to 0 or 1. Evaluation uses short-circuit evaluation.

Listing 8: Logical Operation in C

```
!0x41 -> 0x00
!0x00 -> 0x01
!!0x41 -> 0x01
0x60 && 0x55 -> 0x01
0x69 || 0x55 -> 0x01
```

### 3.2.3 Shift Operations

A left-shift moves all bits one to the left and adds 0s to the right. Bits pushed out on the left are discarded.

Right shifts can be logical or arithmetic. Independent of the variant, the bits are shifted to the right by the given number of positions. Bits pushed out on the right are discarded. With a logical shift, we fill the void placed on the left with 0s. With an arithmetic shift, we fill the void places with the most significant bit of the original int.

In C, we do logical shifts on unsigned `int`s and arithmetic shifts on signed `int`s.

Listing 9: Shifting in C

```
01100010U << 3 -> 00010000
01100010 << 3 -> 00010000
01100010U >> 2 -> 00011000
01100010 >> 2 -> 00011000

10100010U << 3 -> 00010000
10100010 << 3 -> 00010000
10100010U >> 2 -> 00101000
10100010 >> 2 -> 11101000
```

Shifting by $< 0$ or $\geq$ word size is undefined behavior.

## 3.3 Integer Addition & Subtraction

The one's complement of an integer `i` is defined to be `~i`, i.e., all bits flipped. Then see that `i+~i=0xFF...=-1` so that `-x = ~x + 1` is the two's complement of an integer and corresponds to its negation.

### 3.3.1 Unsigned Addition

The true sum of two unsigned integers may require one additional bit to store. However, we discard of that bit on the left to get an int of the same size. So, the standard addition function ignores the carry output. Instead, the C standard defines that the number 'wraps around', i.e., is just the result without the carry. Mathematically:

$$\text{unsignedAdd}_w(u, v) = u + v \mod 2^w$$

$$\text{unsignedAdd}_w(u, v) = \begin{cases} u + v, & u + v < 2^w \\ u + v - 2^w, & u + v \geq 2^w \end{cases}$$

This has several mathematical properties so that integer addition in C/computer arithmetics is an Abelian group.

- closed under addition: $0 \leq UAdd_w(u, v) \leq 2^w - 1$

- commutative: $UAdd_w(u,v) = UAdd_w(v,u)$
- associative: $UAdd_w(t, UAdd_w(u,v)) = UAdd_w(UAdd_w(t,u),v)$
- 0 is the additive identity: $UAdd_w(u,0) = u$
- Every element has an additive inverse:

  - $UComp_w(u) = 2^w - u$

  - $UAdd_w(u, UComp_w(u)) = 0$

### 3.3.2 Signed Addition

On a bit level, this works just as unsigned addition: We do standard addition and discard of the carry bit. This this is possible is even the reason why signed integers use two's complement format. In contrast to unsigned addition, overflow is not defined by the standard. But all sensible implementations do this just as unsigned addition in hardware, so the behavior is usually the same.

$$\text{signedAdd}_w(u,v) = \begin{cases} u+v+2^w, & u+v < TMin_w \\ u+v, & TMin_w \le u+v \le TMax_w \\ u+v-2^w, & TMax_w < u+v \end{cases}$$

Mathematically, this is isomorphic to unsigned addition with unsignedAdd. 2's complement under signedAdd forms a group just as we have for unsigned addition.

An interesting difference is the inverse, which is $TComp_w(u) = \begin{cases} -u, & u \neq TMin_w \\ TMin_w, & u = TMin_w \end{cases}$.

## 3.4 Integer Multiplication

One can quickly see that adding two arbitrary integers may yield a new integer which is outside the range of representable integers.

$$\text{range for unsigned ints: } 0 \le x \cdot y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$$

$$\begin{gathered} \text{range for signed ints:} \\ -2^{2w-2} + 2^{w-1} =\ge (-2^{w-1}) \cdot (2^{w-1} - 1) \le x \cdot y \\ x \cdot y \le (-2^{w-1})^2 = 2^{2w-2} \end{gathered}$$

To get a precise result, we would (to cover the worst case) have to double the word size with each multiplication. If one wants such precision, it is done using 'arbitrary' precision libraries. But in practice we want to the result to be of the same word size as the inputs.

### 3.4.1 Unsigned Multiplication

As with addition, we just discard any additional bits and just keep the lower bits that fit our word size.

$$UMult_w(u,v) = u \cdot v \mod 2^w$$

The mathematical properties of this correspond to a commutative ring. We don't have a field because the inverse generally doesn't exist.

- closed under multiplication
- commutative
- associative
- 1 is the multiplicative identity
- multiplication distributes over addition

### 3.4.2 Signed Multiplication

This is again just as for unsigned multiplication: We simply ignore/discard of the higher order $w$ bits and retain the word size. Accordingly, this is also isomorphic to unsigned multiplication and addition.

## 3.5 Ordering Properties of Numbers

This is just a brief remark that common ordering properties of integers don't translate to `int`s. The following are NOT ebeyed by machine arithmetic:

- $u > 0 \Rightarrow u + v > v$
- $u > 0, v > 0 \Rightarrow u \cdot v > 0$

## 3.6 Integer Multiplication and Division using Shifts

Integer addition and subtraction typically work in one cycle. Meanwhile, multiplication takes significantly longer. Historically, about 20 cycles. Modern machines may manage in about 4 cycles. Division is even worse. But even if it can be done in a few cycles, the power consumption and required chip area is still noticeably higher. Therefore, programmers have sought alternative approaches to multiply and divide for special cases. One such special case is multiplication and division with powers of two.

### 3.6.1 Left Shift / Multiplication by Powers of Two

With left-shifts, we can multiply by powers of 2. We just discard of bits which are shifted out on the left and add zeros on the right. Accordingly, we have modular arithmetic for unsigned integers. For signed integers, the behavior may be wired.

This can be used to do certain operations faster. For instance, `u * 8` can be written as `u << 3`, or `u * 24` as `(u<<5)-(u<<3)` (may even be done in 2 cycles on superscalar machines). As machines generally execute shift and add faster, these alternatives are chosen. Even the compiler can detect such simplifications and generate assembly, which implements such optimizations.

### 3.6.2 Right Shift / Division by Powers of Two

**Unsigned Integers** Here, with logical shift (fill with zeros on the left), `u >> k` and $\lfloor$`u/2^k`$\rfloor$ are equivalent expressions. We discard of all bits which are shifted out on the right side. This has the same semantics as $u/2^k$ if we write it directly in C.

**Signed Integers** If we do right-shifts with signed integers, we do an arithmetic shift (fill with the sign bit on the left) and `u >> k` and ⌊`s/2^k`⌋ are equivalent. But this does not correspond to what we usually want. If we divide $-15213$ by two, a mathematical division results in $-7606.5$. Normally, we round toward $0$, but with shifting we get $-7607$ instead, i.e., we round toward $-\infty$. Actually, we want to compute ⌈`s/2^k`⌉ for negative numbers. We can do so by computing $\lfloor (s + 2^k - 1)/2^k \rfloor$. This can be done in C as we take the floor with `(s+(1<<k)-1)>>k`.

When one writes a regular division in C and the dividend is known, C compiles the division to an assembly expression of the above form.

## 3.7 Puzzle

# 4 Pointers

## 4.1 Memory Basics

The OS provides each program with its own virtual address space. This address space is private to a program. In modern byte-addressable systems, each address corresponds to a byte. On a 32 bit host, we have $2^{32}$ bytes of virtual memory. On a 64 bits host, we have $2^{64}$ bytes of virtual memory. This memory is not actually available, as we do not have sufficiently large storage/memory units. But this is the potentially usable address space.

When a program is loaded by the OS, it lazily copies regions of the executable file into the right place in the address space while performing final linking, relocation, and other preparations.

Some top addresses are reserved by the OS, some bottom addresses contain the program itself, global variables, ..., and somewhere in the middle we can find dynamically linked libraries.

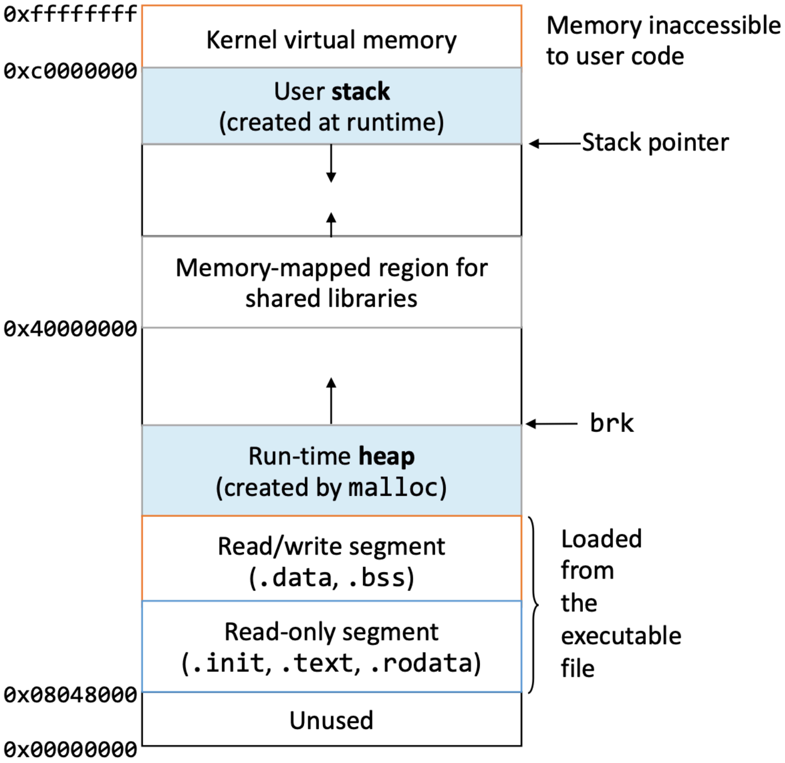

0xffffffff
Kernel virtual memory
Memory inaccessible to user code
0xc0000000
User stack (created at runtime)
Stack pointer
Memory-mapped region for shared libraries
0x40000000
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Read-only segment (.init, .text, .rodata)
Loaded from the executable file
0x08048000
Unused
0x00000000

Figure 5: Program Memory Regions overview

In that order from low to high addresses:

- `.text`: executable code, i.e., the program
- `.rodata`: read-only data like string literals and constant values.
- `.data`: initialized global and static variables.
- `.bss`: uninitialized global and static variables (zeroed at startup)
- `.init`: initialization code for the program

Furthermore, we have the heap and stack. Those are common concepts used in many languages. In C, the stack is automatically managed (throught the compiler & OS), while we manually need to put and remove elements on/from the heap. In assembly, the stack pointer is just increased to allocate on the stack and the OS then must ensure that those virtual addresses are properly mapped. On the other hand, the heap requires more programmer attention, because memory needs to be allocated and freed as disucssed later.

**Stack** Many languages such as C, C#, Java, Python, Rust, ... use a stack. It is used to store the state of each instantiation of functions/procedures (i.e., whatever is 'running'). This is used to support 'reentrant' code.

Memory on the stack is allocated by the program in frames. Each procedure/function uses a stack frame. They allocate memory but only need it for a limited amount of time: From when being called to returning. Stack discipline dictates how the stack is to be used by functions. This includes:

- where to put function parameters not fitting in registers
- alignment of the stack (usually 16 bytes ?)
- where the return address is stored (return instruction pointer)

The stack grows downwards. The lower end of the stack is called the top and identified by the stack pointer. The frame pointer points to the end of the current frame (i.e., the highest address of the current frame).

The stack pointer is initialized by the OS and then used by the program to grow the stack (by incrementing it). The OS randomly initializes the stack pointer for a program. This is a security feature to prevent unauthorized access and prevent overflow attacks by being able to predict where things will reside in memory. (We will look at those attacks later.) A consequence of this is: When printing memory locations of variables/pointers, those will vary between executions of the program, even if it did the same thing. Just because the base address of the stack pointer has been varied by the OS. Unfortunately, this also makes debugging more difficult. Such random initialization/address assignments also happen for shared library locations.

## 4.2 Pointers in C

A pointer is a type of variable in C, which stores the memory address of another variable. Given a variable, we can use the `&` operator to get the (virtual) address of that variable, i.e., the memory location of where the data of that variable is stored. In `printf()`, a pointer can be printed with `%p`. We can use `&` not only on variables, but also on functions and get the location of the function code in memory.

Listing 10: Pointers in C

```
int foo() {}
int x;
int a[2];
printf("x␣is␣at␣%p\n", &x);
printf("a[0]␣is␣at␣%p\n", &a[0]);
printf("foo␣is␣at␣%p\n", &foo);
```

Pointers are stored in special variables. We add an `*` to the type to indicate that we want to store a pointer to such a type. We may declare a pointer with `type *name;` and can also initialize it at the same time with `type *name = address;`.

The pointer type specifies what kind of data is stored at the memory location the pointer references. This is important for pointer arithmetic. If we write `void` for the type, we store a pointer to an element of unknown type. But generally, we should always use typed pointers to have as much support from the compiler as possible to detect mistakes and have pointer arithmetic function properly. As the type, we may also user pointer types so that we specify pointers to pointers of values. We may nest arbitrarily.

Listing 11: Tricky C Pointer Declarations

```
int arr[3] = {2, 3, 4};
int *p = &arr[1];
int **dp = &p;

int *p // p is pointer to int
int *p[13] // p is array[13] of pointers to int
int *(p[13]) // p is array[13] of pointers to int
int **p // p is pointer to pointer to int
int (*p)[13] // p is pointer to array[13] of int
int *f() // f is function returning a pointer to int
int (*f)() // f is pointer to function returning int
int (*(*f())[13])() // f is function returning pointer to array
    [13] of pointers to functions returning int
int (*(*x[3])())[5] // x is array[3] of pointers to functions
    returning pointers to array[5] of ints
```

One may use (not in the exam, of course) https://cdecl.org to get a translation between English and C gibberish. The general scheme of understanding a declaration is to start at the name/innermost part and then always alternate between considering the next specification on the right and left.

When having a pointer (variable), we often need to get the value, which is located at the memory address, which is stored in the pointer. Doing so is called dereferencing and is done by putting a * in front of the variable name. Dereferencing a double/triple/... pointer works by putting multiple *.

Listing 12: Dereferencing Pointers in C

```
v = *pointer; // dereference a pointer
*pointer = value; // dereference and assign a pointer
```

Technically, a pointer just stores an 8 byte address (on a 64 bit system). So we can assign anything we want and try to dereference the contents of any variable (once casted to a pointer). Notice that when we try to access a memory location (by dereferencing a pointer with some random value/address), we usually get a (page) segmentation fault as the program may only access memory, which has been allocated. There are no checks where a pointer points do. We have to make sure it is a valid address or accept that our program will crash.

In many cases it is helpful to have a pointer value which is definitely invalid to indicate that some pointer is not set/known or similar. For that we may use the macro `NULL`. During preprocessing it is substituted with all zeros and any attempt to dereference `NULL` will result in a page segmentation fault. `typeof(NULL)` is `void **`.

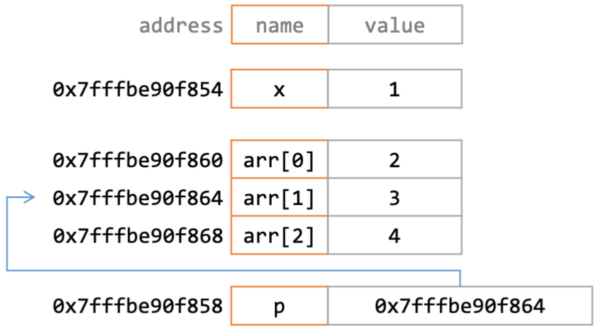


Figure 6: Box and Arrow Diagram to Visualize a Memory Layout

### 4.3 Pointer Arithmetic

Pointer arithmetic refers to doing addition etc. on pointers. When we do `*p += 1;`, we work on the dereference pointer, i.e., the value stored at the memory address contained in the pointer. But if we do `p += 1;`, we work on the pointer itself.

We already know that pointers are typed. When doing pointer arithmetic, the basic unit corresponds to the size of the type the pointer references. So, if we do `p += 1;` for `int *p`, we increase the address by 4 bytes as an integer is stored in 4 bytes. But if `p` were to reference a `char`, we would only increase by one byte as a char is stored in one byte.

So, pointer arithmetic internally works knowing the (bit-/byte-)size of different types. We can also manually get this size in C using `sizeof(...)`. As a parameter, we pass a type or a value/variable. This evaluates at compile time to the size of the parameter in bytes.

### 4.4 Arrays & Pointers

An array is a collection of homogeneous data elements stored at continuous memory addresses. Generally, an array is NOT a pointer - but we will see that it is mostly treated as one.

In expressions, an array is usually treated as a pointer to the first element of the array. But there are some exceptions: `sizeof(array)` provides the length of the array and `&array` returns the address of the array.

In fact, `A[i]` is always interpreted as `*(A+i)` by the compiler. The squared brackets `[]` don't do anything besides addition and dereferencing. So, instead of `a[5]` we may write `5[a]`. But because they are not truly pointers, assignments such as `array1 = array2` do not work, because `array1` is not an pointer and thus can't be assigned a pointer (what the `array2` in the expression is considered to be).

A special case is if we take an array as a function parameter. Then it is just special notation for taking a pointer. If ‘array‘ is the parameter to a function, we can write ‘array = array2;‘. Getting ‘array‘ as a parameter can be done in various ways.

Listing 13: Array as Parameter in C

```c
int arrfun(int *array) {}
int arrfun(int array[]) {}
int arrfun(int array[43]) {}
```

But notice that `int arrfun(int (*array)[]){}` is different. Here we consider a pointer to an array. Thus, one must pass the array's pointer and do some more things adequately. Only then one can also assign. In the end, this is still a pointer and not a 'true' array.

Finally, pay attention to initialization. Depending on how an array is initialized (as an array or pointer) the content may be stored on the stack or in the `.data` section of memory. This is particularly relevant for strings.

Listing 14: String Initialization on Stack vs. in .data Section

```
char a[] = "Hello!"; // allocated on the stack
char *b = "Hello!"; // pointer allocated, points to .data
    segment (read-only)
```

## 4.5 Pass-by-Value/-Reference

In general, function arguments are passed by value in C, which means that the callee gets a copy of the argument. If the callee edits the parameters, it only edits values in its stack frame. But this does not have an impact on the caller's stack frame. Therefore, the callers variables remain unchanged.

When using pointers, we can enable pass-by-reference. The callee still receives a copy of the argument/variable. But now that copy is a pointer/memory address. And when dereferencing that address/pointer, one works on the original value in memory. Then, the callee can modify a variable in the scope of the caller (or even another scope or dynamic memory).

Listing 15: Pass-by-Value vs. Pass-by-Reference in C

```
// Pass-by-Value - this doesn't work as intended
void swapValue(int a, int b) { // we need to pass values when
    calling
  int tmp = a;
  a = b;
  b = tmp;
}
// Pass-by-Reference - this works as intended
void swapReference(int *a, int *b) { // we need to pass pointers
     when calling
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

A real-world example of pass by reference is `strcpy`:

Listing 16: `strcpy` in C

```
char *strcpy(char *dest, char *src)
{
  char *r = dest;
  while(*dest++ = *src++);
  return r;
}
```

# 5 Dynamic Memory

When dealing with memory so far, we have considered statically and automatically allocated memory. The former are global variables, and static function member variables. They are allocated when the program is loaded in the .data section, and deallocated when the program exists. The latter are variables inside functions, which only exist in the scope of that function. They are allocated when the function is called and deallocated when the function returns. But this does not cover all use cases:

- We may want some variable/memory location to outlive the lifetime of a function but not live for the entire program duration (i.e., be global).
- Some complex data structure may be too large to fit on a stack. The stack is pre-allocated and its size of often determined at program startup.
- A function might want to return something of which the size is unknown during compilation. So it can't simply return the value.

The solution is to explicitly request memory locations, which can then be used to store large data, can be used until not longer needed/freed accross multiple functions, threads, etc., and can be passed around by means of pointers.

The program explicity requests a new block of memory, which is then allocated by the language runtime (with support by the OS). Dynamically allocated memory persists until it is explicitly deallocated by the program in C. In other languages a garbage collector might automate this task, but in C we have much more manual control (and need to exercise it).

The difficult thing is to allocate and mange (i.e. deallocate correctly) memory explicitly and manually. One needs to allocate the right amount of memory and then fill it with structured types. This involves tracking the allocated addresses so that one doesn't fill up memory with allocated blocks that the program forgot exist. More complex data structures can be and need to be built manually using pointers to allocated memory.

## 5.1 The C Memory API

The C memory API is declared in `stdlib.h`, which needs to be imported to be able to use the dynamic memory system explained in the following. In that header file, we also have the type definition of `size_t`. We will use `size_t` whenever referring to pointer indices or similar. It is an unsigned integer of some size, commonly an `unsigned long`. It is large enough to hold the size of the largest possible array in memory. We also have `ptrdiff_t`, which is less common, and used to store signed integers. It is common to store the result of subtracting two pointers in a `ptrdiff_t`.

### 5.1.1 malloc

`void *malloc(size_t sz)` is the function signature of the function to be used to allocate a block of memory on the heap of the given size, where `sz` refers to the number of bytes, i.e., the number of addresses to be allocated.

The function returns a pointer to the first byte of that memory, i.e., the lowest address of the allocated memory locations. If allocation failed, we get `NULL`. Before using a pointer from `malloc`, we should always make sure that it is not `NULL` and then cast it to the proper pointer type we intend to use the memory for. Also, there is no guarantee regarding the content of the allocated memory. We should assume there are garbage values.

When allocation memory, we need to specify how much memory we require. As already stated, the parameter specifies the number of bytes. A useful function to compute the number of required bytes is `sizeof(...)`, which is evaluated during compilation to the implementation defined byte size of passed variable/type.

Listing 17: Malloc Allocation

```
long *arr = (long *)malloc(10*sizeof(long));
if (arr == NULL) {
  return ERRCODE;
}
arr[0] = 5L;
// ...
```

### 5.1.2 calloc

`void *calloc(size_t m, size_t sz)` is the function signature of the function to be used to allocate blocks of memory of size (nm × sz). The return behavior etc. is just as with malloc with one important difference: The memory is filled with zeros. Hence, this is also slightly slower than `malloc`.

Listing 18: Calloc Example

```
long *arr = (long *)calloc(10, sizeof(long));
if (arr == NULL) {
  return ERRCODE;
}
arr[0] = 5L;
// ...
```

### 5.1.3 free

Once we finished using memory we allocated with `malloc` or `calloc`, we should release it so that it can be used again. Otherwise, the OS and language runtime assume that it is still required and may fill the entire memory.

To free memory locations that have been allocated, we need a pointer to the first byte of the malloced/calloced memory, and then call `free(pointer).` Some future malloc/calloc then might reuse that memory address. It is good practice to `NULL` the pointer afterwards so that we don't accidentally use the freed memory again.

Listing 19: Free Example

```
long *arr = (long *)calloc(10, sizeof(long));
if (arr == NULL)
{
  return ERRORCODE;
}
// Do something ...
free(arr);
arr = NULL;
```

### 5.1.4 realloc

Above, we allocated memory of a fixed size. But in some cases, we might want to extend our memory region (i.e., if we want to grow an array, for instance). For this purpose, we can use `void *realloc(void *ptr, size_t size)`. We must pass a pointer to the first byte in our memory block and our newly desired memory region size as the number of bytes.

Either, we get a pointer to some memory location, which has our newly requested size (and the content of the old location). Then, the original pointer is invalid. Or we get ‘NULL‘ and couldn’t get a larger memory region. Then, our original pointer is still valid.

Listing 20: Realloc in C

```c
long *arr;
if (!(arr = (long *)malloc(10*sizeof(long)))) {
  return ERRCODE;
}
// Do something ...
if (!(arr = (long *)realloc(arr, 20*sizeof(long)))) {
  return ERRCODE;
}
```

### 5.1.5 Example

Listing 21: Dynamic Memory Usage Example in C

```c
#include <stdio.h>
#include <stdlib.h>
#define INIT_SIZE 8

int main(int argc, char *argv[]) {
  int num; // Num. of integers
  int *arr; // Array of ints.
  size_t sz; // Array size
  int m, in; // Index & input num

  // Allocate the initial space.
  sz = INIT_SIZE;
  arr = (int *)calloc(sz, sizeof(int));
  if (arr == NULL) {
    fprintf(stderr, "calloc␣failed.\n");
    exit(1);
  }

  // Read in the numbers.
  num = 0;
  while(scanf("%d", &in) == 1) {
    // See if there's room.
    if(num >= sz) {
      // There's not.  Get more.
      sz *= 2;
      arr = (int *)realloc(arr, sz*sizeof(int));
      if(arr == NULL) {
        fprintf(stderr, "realloc␣failed.\n");
        exit(1);
      }
    }
    // Store the number.
    arr[num++] = in;
  }
  // Print out the numbers
  for(m = 0; m < num; ++m) {
    printf("%d\n", arr[m]);
  }
  free(arr);
  return 0;
}
```

## 5.2 Managing the Heap

The heap, a large pool of memory/area of the virtual memory space, is where all the memory addresses acquired through dynamic memory allocation reside.

`malloc()` allocates chunks of memory in the heap, `free()` returns them. `malloc()` (must) maintain bookkeeping data in the heap to track allocated blocks so that (a) addresses are not allocated multiple times, (b) the size of an allocated region is known, ... We will consider the details of the implementation and bookkeeping data in the section on implementing dynamic memory allocation and management.

But already on the level using the specified memory API provided by C, we need to know what we are doing. Otherwise we may corrupt memory and cause our program to crash.

### 5.2.1 Memory Corruption

- assign past the end of an integer
- mess up pointer arithmetic by assigning unallocated pointers
- work with `void *` points (cast to a specific type whenever possible)
- assume `malloc` puts zeroes in memory
- pass pointer to `free` that wasn't malloced
- free memory address twice
- use a pointer on which we called `free`

Listing 22: How to Corrupt Memory in C

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
  int a[2];
  int *b = malloc(2*sizeof(int)), *c;
  a[2] = 5; // assign past the end of an array
  b[0] += 2; // assume malloc zeroes out memory
  c = b+3; // mess up your pointer arithmetic
  free(&(a[0])); // pass pointer to free() that wasn't malloc'ed
  free(b);
  free(b); // double-free the same block
  b[0] = 5; // use a free()'d pointer
  return 0;
}
```

### 5.2.2 Memory Leaks

Memory leaks happen when our code doesn't deallocate allocated memory that is not used anymore. This does not impact correctness, but performance. The program's memory footprint will keep growing. This might:

- slow down the program over time due to virtual memory trashing (discussed later)
- use up all available memory and crash (need to allocate A LOT)
- might starve other programs of memory

## 5.3 Structures and Unions

### 5.3.1 struct

In C, a `struct` is a type that contains a set of fields. `struct`s don't have member functions but only contain fields, which are put contiguously in memory. Instances may be allocated on the stack or heap - depending on where they are used.

Listing 23: Structs in C

```
struct structNAme {
  type name;
  type2 name2;
  type3 anem3;
  ...
}
struct Point {
  int x;
  int y;
}
```

To access a field of a struct we use `.` after the variable name followed by the field name. When working with a pointer to a struct, we would write `(*pointer).fieldName`. A shorter notation for that is `pointer->fieldName`.

Listing 24: Accessing Structs

```
int main(int argc, char **argv) {
  struct Point origin = { 0, 0 };
  struct Point *origin_ptr = &origin;
  origin.x = 1;
  origin_ptr->y = 2;
  return 0;
}
```

When assigning one struct to another struct, we assign by copying every field from one to the other struct. When passing as parameters, structs are also passed by value (as everything in C). When we want to pass a reference, we need to pass a pointer to the struct.

A struct is aligned in memory by the size of its largest element. And inside the struct, each element is aligned according to its size. Because the elements are in memory in the specified order, the compiler may add padding in between values and at the end.

### 5.3.2 union

A union is similar to a struct by defining a set of fields. But other than a struct, it only holds one of those values at a time. The memory allocated (and memory alignment) for a union corresponds to its largest field and we can access any of a union's fields without checks for what is currently stored in the memory location.

Access to the fields of a union works just as for a struct.

Listing 25: Unions in C

```
union unionName {
  type name;
  type2 name2;
  type3 name3;
  ...
}
union u {
  int ival;
  float fval;
  char *sval;
}
union u my_uval;
```

## 5.4 Type Definitions

With typedef we can define a new type. Specifically, we create a new name for a type (which may be a combination of various other types, pointers, arrays, functions). After declaring a new type, we can use the new name interchangeably with the original expression. We use typedef like this: typedef actualType newNameForThatType.

This is particularly often used with structs and unions.

Listing 26: typedef in C

```
typedef unsigned uint32_t;
uint32_t ui;
typedef int **myptr;
int *p;
myptr mp = &p;
typedef struct skbuf skbuf_t;
skbuf_t *sptr;

// instead of:
int (*(*x[3])())[5];
// we may do:
typedef int fiveints[5];
typedef fiveints* p5i;
typedef p5i (*f_of_p5is)();
f_of_p5is x[3];
```

Listing 27: typedef with structs and unions

```
// option 1
struct list_el {
  unsigned long val;
  struct list_el *next;
};
typedef struct list_el el_t;
struct list_el my_list;
el_t my_other_list;

// option 2
typedef struct list_el {
  unsigned long val;
  struct list_el *next;
} el_t;
struct list_el my_list;
el_t my_other_list;
```

## 5.5 Dynamic Data Structures

Using pointers and dynamically allocated memory, we can manually build data structures, which are often provided in other languages. We will consider a singly linked list as an example here.

### 5.5.1 Fixed Data Types

Listing 28: int Linked List in C - only pushable

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
struct node {
  int element;
  struct node *next;
};
struct node *Push(struct node *head, int e) {
  struct node *n = (struct node *)malloc(sizeof(struct node));
  assert(n != NULL);   // crashes if false
  n->element = e;
  n->next = head;
  return n;
}
int basicMain(int argc, char **argv) {
  struct node n1, n2;
  n1.element = 1;
  n1.next = &n2;
  n2.element = 2;
  n2.next = NULL;
  return 0;
}
int pushMain(int argc, char **argv) {
  struct node *list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return 0;
}
```

### 5.5.2 Generics

Listing 29: Generic Linked List in C - only pushable

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
struct node {
  void *element;
  struct node *next;
};
struct node *Push(struct node *head, void *e) {
  struct node *n = (struct node *)malloc(sizeof(struct node));
  assert(n != NULL); // crashes if false
  n->element = e;
  n->next = head;
  return n;
}
int main(int argc, char **argv) {
  char *hello = "Hi␣there!";
  char *goodbye = "Bye␣bye.";
  struct node *list = NULL;
  list = Push(list, (void *) hello);
  list = Push(list, (void *) goodbye);
  return 0;
}
```

# 6 Implementing Dynamic Memory Allocation

We have used `malloc`, `calloc`, and `free` to allocate and free memory. But how do we end up to get a pointer to a memory address? What processes happen to assign memory, make sure that it isn't used otherwise, we free exactly the right amount of memory when calling `free`, ...?

Memory allocation is a process that happens jointly between the virtual memory hardware and kernel software. On the abstraction level of the OS, one deals with pages, which can be multiple KB large. But application objects (structs, arrays, ...) are typically smaller. Hence, one needs to manage to allocate pages and then assign and distribute memory blocks in pages upon calls to `malloc`. (A block is here understood as a continuous range of bytes in memory.)

The program/application requests some number of bytes of memory. The dynamic memory allocator manages and allocates pages if necessary. In other words, this makes the abstraction between the actual heap memory and us calling 'malloc' etc.

In C, memory management means that we allocate memory from some page explicitly when `malloc` is called and only free that memory when `free` is called. In other languages that is different. If languages have garbage collection, the garbage collector can free memory as soon as some memory location/object becomes unreachable. And in languages such as Rust, the compiler generates the free automatically by analyzing the program and requiring the programmer to specify some additional information for variables/use specific programming paradigms.

## 6.1 The Challenge

Consider the heap, from which `malloc` etc. take addresses, as just one large chunk of memory. We have several properties that implementations of the Memory API should meet for calls in arbitrary order to `malloc`, `calloc`, `realloc`, and `free` (only when freeable):

- must respond immediately to `malloc` requests
- can't control the number or size of allocated blocks
- must allocate memory blocks from free memory
- must align blocks so that they satisfy all alignment requirements (commonly, 8 byte alignment – may require to add padding)
- an only manipulate and modify free memory
- can't move allocated blocks

Besides those properties that are relevant to the user, there are also some apparent challenges that must be solved as problems internal to the implementation:

- How does the dynamic memory allocator know how much memory to free when given only a pointer to the first byte of the block?
- How do we keep track of free blocks to distribute them to `malloc` requests?
- How do we handle space that is not needed, i.e., when allocating a block that does not cover the entire free block is placed in?
- How to we pick a free block for allocation? There may be multiple that fit?
- How to we technically free a block?

**The Goal** Any implementation (that must adhere to the before mentioned properties) aspires to provide high performance and peak memory utilization. Lets consider a sequence of $n$ memory requests $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$. Performance can clearly be understood as the number of completed memory requests per unit time. Peak memory utilization is more difficult to understand and we must introduce some definitions for it. When calling `malloc(p)`, a block with a payload of $p$ bytes is allocated. the *aggregate payload* $P_k$ is the sum of currently allocated payloads after request $R_k$ has been completed. And as stated earlier, all those blocks are put on the heap. The heap grows by the implementation requesting additional pages from the OS with `sbrk()`. The *current heap size* $H_k$ is the current and nondecreasing size of the heap as the size of all pages requested from the OS. Then: Peak memory utilization after $k$ requests is $U_k = \frac{\max_{i \leq k} P_i}{H_k}$.

Ideally, we would have $U_k$ but that is not possible in practice, because we need to add meta data and have fragmentation.

Meta-data is necessary to store where the blocks are located, which memory addresses are currently allocated and which are free, what is the size of the allocated blocks, ... The meta-data (including the length of the allocated word) is usually put in memory right before the data. That is called the header/header file. It takes up memory but does not count towards the payload data. Furthermore, we often also store similar information after the payload in a footer. We will see later why that is done.

Fragmentation describes that some memory also remains unused. For instance, we might have to add padding to guarantee alignment. Or we have free regions of memory but the to be allocated block is too large to fit and we must request a new page despite having free regions.

### 6.1.1 Internal Fragmentation

The request `malloc(p)` actually allocates more than $p$ bytes of memory. It allocates some memory for a header, some memory for a footer, and in-between there are the $p$‘ bytes for the payload. This has already been stated before. The header/footer are necessary (and useful) to maintain the heap data structure.

However, additionally, we might add padding if the head, footer, and payload do not cover the entire addresses, which are connected by the alignment requirements. So, internal fragmentation is caused by the dynamic memory allocator - specifically, when it allocates more memory than requested.

We say that the internal fragmentation only depends on the pattern of previous requests. Thus it is comparatively easy to measure.

### 6.1.2 External Fragmentation

External fragmentation occurs when there is enough aggregated heap memory but no single free block is large enough to cover our `malloc` request. Then, we (may) have to request a new page and leave memory free in the existing pages. Because we can’t modify pointers we returned earlier, we can’t move the allocated blocks to a different location to reduce external fragmentation.

## 6.2 Keeping Track of Free Blocks

There are 4 common methods to keep track of free blocks. We will consider those below. But first, we need to discuss a technique used in multiple of those methods to reduce external fragmentation. It iss called coalescing.

**Coalescing** It may be helpful to read the part on implicit free list until coalescing is mentioned to best understand how this is relevant.

The idea of coalescing is that if we free a block and there is/are free blocks before/after, that we join (coalesce) them instead of introducing false fragmentation.

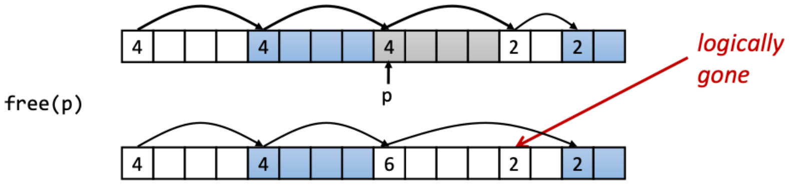


Figure 7: Coalescing

Listing 30: Coalesce with the next Block

```
void free_block_after(ptr p) {
  *p = *p & -2; // clear allocated flag
  next = p + *p; // find next block
  if ((*next & 1) == 0) {
    *p = *p + *next; // add to this block if not allocated
  }
}
```

To enable bidirectional coalescing we utilize the footer, which we already mentioned earlier. In case of the implicit free list it also stores the size and is-allocated flag and comes at the end of the memory block. This allows us to traverse the list backwards but also requires extra space.

Coalescing takes constant time. This becomes apparent when considering the four possible cases:

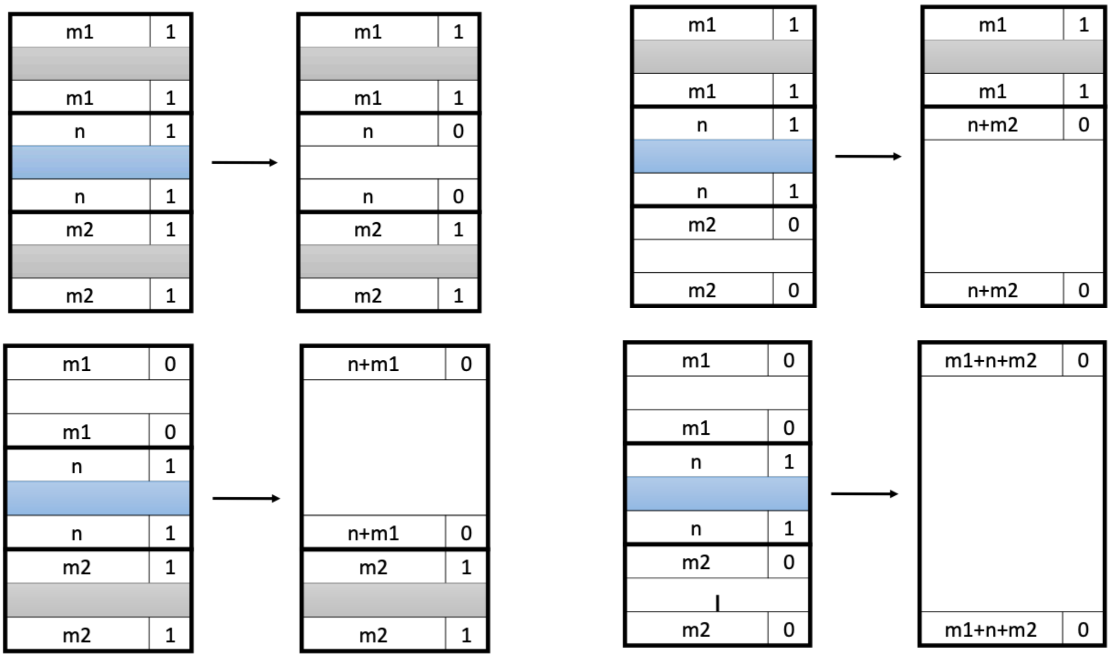


Figure 8: Four Cases of Coalescing

Another interesting question is to decide when to coalesce. This is referred to as the coalescing policy. Immediate coalescing: Coalesces each time ‘free‘ is called. Deferred coalescing: This improves the performance of ‘free‘ by deferring coalescing until needed (possibly when ‘malloc‘ is called, or when external fragmentation reaches some threshold).

### 6.2.1 Implicit Free List

For each block we store its length (including the header, footer, etc.) and whether it is currently allocated or not. Usually, memory blocks are aligned, so some lower-order address bits are always 0. Hence, we can just use the least significant bit to store the is-allocated flag.

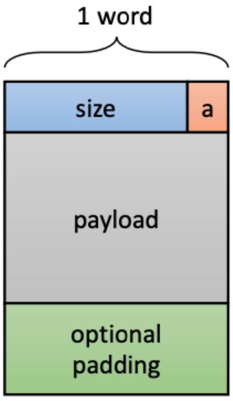


Figure 9: Implicit Free List Memory Block (Header only)

When the payload is required to adhere to some alignment, we must make sure that the header is put before the alignment requirement so that the payload and not the entire used memory region starts at the aligned address.

Summary:

- simple implementation
- linear time worst case allocate coast
- constant time worst case free cost (event with coalescing)
- memory usage depends on the placement policy

Because of the linear time complexity, this is only used in special purpose applications.

**Finding a Free Block / Placement Policies** There are multiple approaches to finding a free block in such a list. Fundamentally, we must iterate over the list and can access subsequent blocks by jumping size bits to the next block. We must always mask out the is-allocated flag bit.

With first fit, we search the list from the beginning and choose the first free block that fits. This can take linear time in the total number of blocks (allocated and free). The following code snippet does not take care of requesting a new page from the OS if space is not sufficient.

Listing 31: Implicit Free List - Frist Fit

```
p = start;
while ((p < end) && \\ not passed end
  ((*p & 1) ||\\ already allocated
  (*p <= len))) { \\ too small
    p = p + (*p & -2);
    \\ goto next block (word addressed)
}
```

With next fit, we start where the previous search finished. This should be faster than first fit, because we avoid re-scanning unhelpful blocks. But research suggests that fragmentation is worse (as we tend to allocate new pages quicker).

With best fit, we iterate over the entire list and choose he best free block, i.e., the block that fits with the fewest bytes left over. This reduces fragmentation but runs slower as we iterate the entire list instead of likely stopping early.

**Allocating in a Free Block / Splitting** When having found a suitable block, it might very well be that the block is larger than the actually required and to be allocated space. In such a case, we may want to split the block so that the unused parts can be allocated by another `malloc` call. The decision of how much internal fragmentation we intend to allow as the result of splitting is referred to as the splitting policy.

Listing 32: Example Splitting / Adding Implementation

```c
void addblock(ptr p, int len) {
  int newsize=((len+1)>>1)<<1; // round up to even - assumes len covers payload and header
  int oldsize = *p & -2; // mask out low bit
  *p = newsize | 1; // set new length
  if (newsize < oldsize) {
    *(p+newsize) = oldsize - newsize; // set length in remaining part of block
  }
}
```

**Freeing a Block** In the simplest implementation we only need to clear the is-allocated flag with `void free_block(ptr p){ *p = *p & -2 }`. But this can lead to what we call false fragmentation. Instead, we do coalescing as explained at the beginning of this subsection.

### 6.2.2 Explicit Free List

Here, we introduce an explicit list of only all free blocks and not all free and allocated blocks. Because the next free block could be anywhere in memory, we have to store forward/backward pointer for a doubly-linked list. We still need the boundary tags (header and footer) for coalescing. So the question rises where we store the next and prev pointers. Remember that the block is free, so there is no payload. But there are bytes available for a payload due to alignment.

Summary:

- Allocation is in linear time in the number of free blocks (in comparison to all blocks a before). This is much faster when the memory is (almost) full.
- More complicated allocate and free (need to edit the list).
- Additional memory/space needed for the links (does not increase fragmentation if alignment already coarse enough).

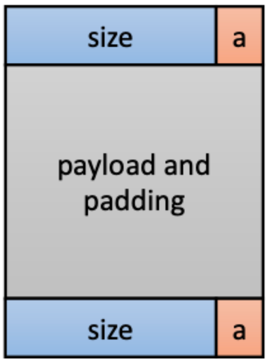


Figure 10: Explicit Free List - Allocated

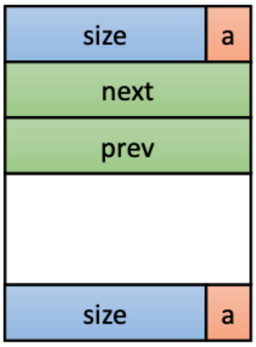


Figure 11: Explicit Free List - Free

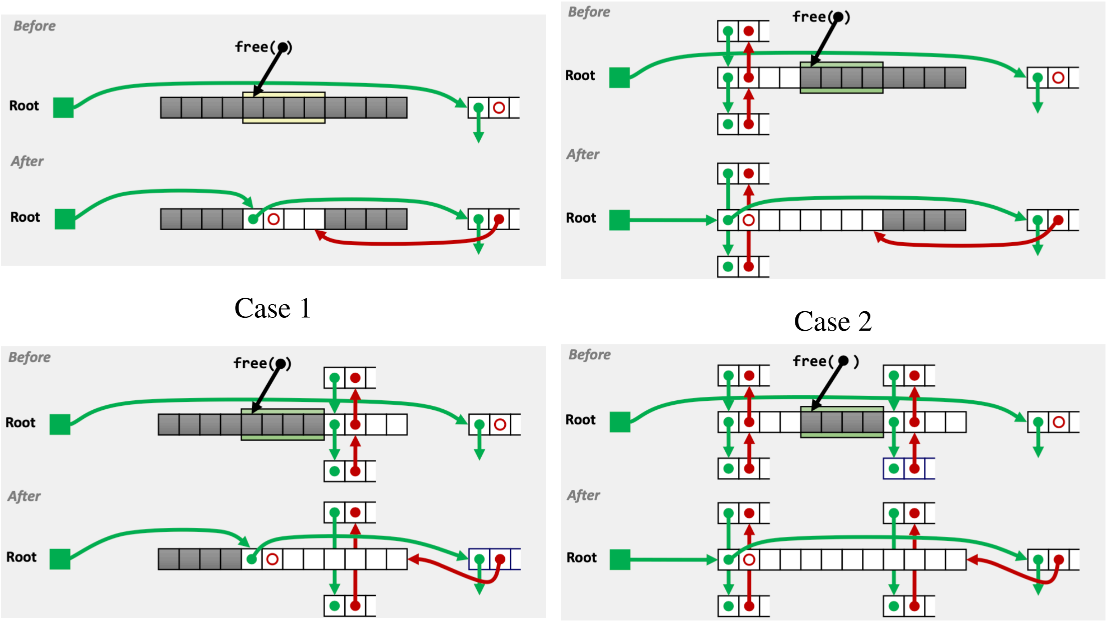
**Allocating Memory & Placement Policy** Allocating memory includes finding a large enough-free block through the explicit free list and then redirecting the list pointers. This means either removing one element or redirecting the pointers to some late part of the current block if we do splitting.

**Insertion Policy** An insertion policy defines where we insert a free block into the explicit free list.

With the last-in-first-out (LIFO) policy we insert a freed block at the beginning of the free list. This is simple to implement and works in constant time. But studies suggests that this has worse external fragmentation.

With the address-ordered policy we insert freed blocks so that the free list's blocks are always in address order. Studies suggests that this has lower fragmentation. But it comes at the cost of having to search for the corresponding place in the free list.

The following demonstrates how the LIFO policy looks with coalescing.



Case 1

Case 2

Case 3

Case 4

Figure 12: Explicit Free List coalescing

### 6.2.3 Segregated Free List

This is an extension of the explicit free list. Instead of maintaining one list, we now maintain separate lists for different sizes of blocks. Thus, we can spend less time searching for a block. The asymptotic runtime is still unchanged, but the real-world constants are much lower as we can skip all small free blocks caused by excessive external fragmentation.

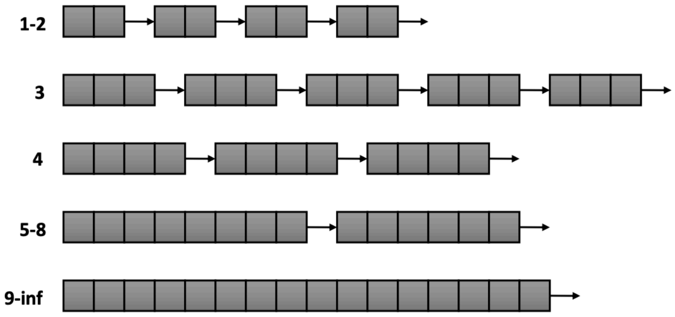


Figure 13: Segregated Free Lists

The advantage of seglists is higher throughput and better memory utilization. It can be understood as a more efficient approximation of best-fit placement.

**Segregated Free List Allocator** To allocate a block of size $n$, we search the appropriate (i.e. first one with available blocks) free list for block of size $m \geq n$. If no block is available across all lists, one has to request additional heap memory from the OS using the system call `sbrk()`.

**Freeing a Block** To free a block one can coalesce and then integrate the new free block into the suitable list.

## 6.3 Garbage Collection

So far we considered explicit memory collection. Garbage collection is available in many modern object oriented programming languages such as Java, Perl, Mathematica etc. In C and C++ only some conservative garbage collectors exist. But they are not part of the standard language and we don't use them. With garbage collection, instead of manually calling `free(...)`, the system should automatically detect that the memory is unreferenced and free it.

Garbage Collectors need to know which memory can be freed. But in general what is used or not depends on conditionals, which we don't know yet. Still, an implicit memory management system can be sure that some memory location isn't needed anymore as soon as there is not pointer referencing it, because the program can't work with that location afterwards.

For this to work we need to make some assumptions regarding pointers. Those are not guaranteed in C, hence, garbage collection is not part of C.

- memory manager can distinguish pointers from non-pointers (only possible in C when sticking to good-practices)
- cannot hide pointers (can store pointers in ‘int‘/‘long‘ in C, for example)
- all pointers point to the start of a block (not-guaranteed in C)

Classical GC algorithms are: mark-and-sweep collection (1960), reference counting (1960), copying collection (1963), and generational collectors (1983).

### 6.3.1 Garbage Collection Algorithms

### 6.3.2 Memory as a Graph

Most GC algorithms consider memory (locations) as a graph. Blocks in memory are nodes. We have a pool of nodes on the stack (root nodes). Directed edges correspond to references to some location. All unreachable nodes, which are not on the stack (or .data segment) can be freed.

### 6.3.3 Mark and Sweep Collection

This always allocates new memory when 'malloc' is called. Only when the program runs out of memory, the program is halted and the GC runs. The GC uses an extra mark bit in the head of each block. It starts at each root node in the graph and sets the mark bit on it and each reachable block. Afterwards, the GC sweeps all blocks and frees all those which are not marked.

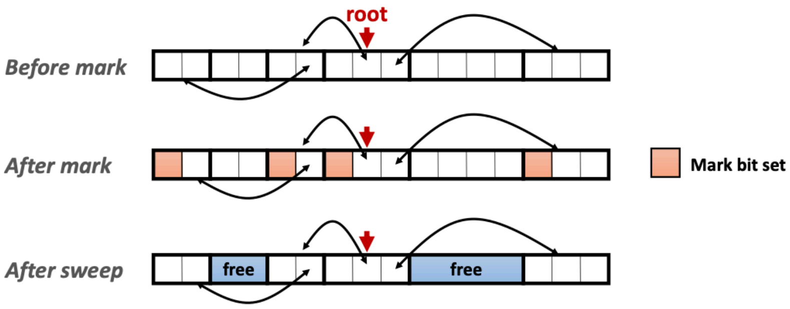


Figure 14: Mark and Sweep Collection

Listing 33: Mark and Sweep Code Idea

```
ptr mark(ptr p) {
  if (!is_ptr(p)) return; // do nothing if not pointer
  if (markBitSet(p)) return; // check if already marked
   setMarkBit(p); // set the mark bit
  for (i=0; i < length(p); i++) { // call mark on all words
    mark(p[i]); //   in the block
  }
  return;
}
ptr sweep(ptr p, ptr end) {
  while (p < end) {
    if markBitSet(p) {
      clearMarkBit();
    } else if (allocateBitSet(p)) {
      free(p);
    }
    p += length(p);
  }
}
```

Here it is specifically important that the assumptions regarding pointers stated earlier are made, including that pointers always point to the beginning of a block. This is not guaranteed in C. So instead of 'conservative garbage collection' we can maintain a balanced-tree of pointers, ordered by their size. Then, we can easily find the beginning of the allocated block for some pointer.

## 6.4 Memory Pitfalls

This repeats some common pitfalls with memory. We already outlined some/many of those in the preceding sections.

**Dereferencing bad Pointers** Now we understand why this is troublesome. We just don't know if the location has been allocated at all. If it hasn't been we likely get a page segmentation fault as we don't have access to that address. And if it has been allocated by chance the next pitfall applies.

**Reading uninitialized Memory** In many high-level languages any data will be initialized to zero for us. Not in C. This has the consequence that we would ready whatever has been in that location last. And that likely seems as complete nonsense.

**Overwriting Memory** If we allocate memory we must make sure that we allocate enough memory. Otherwise it may happen that we exceed the allocated memory with an access or write. That could result in corrupting the heap data structure and other data. This pitfall commonly occurs when...

- allocating memory for a single element and then treating the pointer as an array.
- working with off-by one errors
- writing to string pointers and exceeding their allocated memory
- forgetting that pointer arithmetics behaves differently depending on the pointer type

**Referencing nonexistent Variables** This commonly occurs when returning a pointer to a variable which has been allocated on the stack. This memory location won't be available later and and accessing it may be an error, corrupt other data, etc.

**Freeing Blocks multiple Times** The `free` function does the procedure considered above (and may include coalescing). This is troublesome if the meaning of the data in the location of the specified pointer does not correspond to what the `free` function excepts. Its behavior is unpredictable then.

**Referencing Freed Blocks** Referencing freed block corresponds basically to Dereferencing bad pointers and/or Reading uninitialized memory.

**Failing to free Blocks** Well, this results in memory leaks. In the long run those will render our program and system overwhelmed and irresponsive. In any case, we will encounter virtual memory trashing and very bad performance.

## 6.5 Finding Memory Bugs

Because the memory systems is so intricate and there are little to no guarantees/help from the OS/compiler to avoid mistakes, even minor mistakes can be destructive. Finding those small mistakes is surprisingly hard. But there are some tools that can help us.

- **gdb**: This is the conventional C debugger. It can be used to find bad pointer dereferences but helps little with finding other memory bug.
- **UToronto CSRI malloc**: This is a wrapper for 'malloc'. It detects memory bugs at 'malloc' and 'free' boundaries such as heap structure corrupting writes, memory writes, and maybe freeing blocks multiple times. But it cannot detect overwriting data in the middle of allocated blocks.
- **valgrind**: This is a powerful debugging and analysis technique. It works by rewriting the text section (i.e., the actual code) of executable object files. By doing so it can help to detect:
  - all errors already mentioned with before mentioned techniques
  - bad pointers (at runtime)
  - overwriting (at runtime)
  - referencing outside of allocated blocks (at runtime)

  Valgrind manages a separate memory area to keep track of our memory usage by adding code to our program. Thus, it enables some sort of runtime environment to track errors and problems.

# 7 Some More C

## 7.1 Function Pointers

As their name suggests, function pointers are pointer who point to the location of where some function resides in memory. Hence, one can call a function pointer to change the program counter to that address to execute said function. A function pointer is a pointer just as any pointer and, hence, needs to have a type: `returnType (*name)(param1Type, param2Type);`. We can use this with `typedef` just as any other type.

When adding function pointers as variables to a `struct`, we can achieve something similar to the concept of member variables in OO programming languages. However, we have no protection, visibility, or other convenience features. Also, we still need to define the function somewhere.

## 7.2 The C Preprocessor

Now, we talk about a specific part of the C toolchain again: The preprocessor. Its executable is called `cpp`. We can execute the preprocessor either with that executable or by adding the `-E` flag to `gcc`: `gcc -E foo.c`.

The C processor goes over a C file before compilation and does basic text modifications/substitutions to it. A command for the preprocessor starts with `#`.

### 7.2.1 #include

An include statements can have two forms:

- `#include <file1.h>`: This includes system headers. The preprocessor looks in predefined system folders.
- `#include "file1.h"`: This includes custom/one's own headers. The preprocessor looks in the current folder.

What happens is that the preprocessor takes the content of the to be included file and replaces the `#include ...` with the contents of that file. So one may also include the same file twice via nested includes, which will most likely cause a compilation error. Avoiding double includes can be done with macros.

### 7.2.2 Macros

This enables token-based text substitution. We define a macro with `#define NAME SUBSTITUTION`. After a `#define ...`, every occurrence of `NAME` will be replaced with `SUBSTITUTION`. With `#undef NAME` we can stop the effect.

Also, we can do more complex and parameterized replacements: `#define BAR(x) (x+3)`. When writing `BAR(y)` somewhere after the `define`, it will substituted with `(y+3)`. Notice that NO evaluation happens. This is simply text substitution.

When wanting to insert a full statement, consider the use of `;`. Macros are no statements, so they don't need a `;`. If one adds one, there might end up being two. To be able to write a `;` after a macro one has to omit the `;` in the substitution. Furthermore, macros can be written across multiple lines for convenience and readability. To do so, one has to add a backslash \ at the end a line to continue with the next line. By using those features, one can build large macros.

Listing 34: Macros Examples

```
// without ;
```

```
#define SKIP_SPACES(p, limit)    \
  { char *lim = (limit);         \
    while (p < lim) {            \
      if (*p++ != '␣') {         \
      p--; break; }}}
SKIP_SPACES(p, limit)
// with ;
#define SKIP_SPACES(p, limit)     \
  do { char *lim = (limit);      \
    while (p < lim) {            \
      if (*p++ != '␣') {         \
      p--; break; }}}    \
  while (0)
SKIP_SPACES(p, limit);
```

**Stringizing, Concatenation** When considering macros with parameters, we may either want to paste the parameters directly, be attached in other expressions without spaces or be represented as strings. To paste them directly, we just use them separated from other text and get substitution. But if they are not separated, i.e., the parameter `c` appears in `c_command` in the substitution, the `c_command` will not be modified. Similarly, the `c` in `"c"` won't be substituted.

We use `#c` to get the string of the text of `c`, i.e., `"c"` and `##` to concatenate after substitution, i.e., remove the spaces on both sides of the `##`.

Listing 35: Stringizing and Concatenation with Macros

```
#define COMMAND(c) { #c, c ## _command }
COMMAND(quit) // is equivalent to:
{ "quit", quit_command }
#undef COMMAND
#define COMMAND(c) { "c", c_command }
COMMAND(quit) // is equivalent to:
{ "c", c_command }
```

**Predefined Macros** The preprocessor has many predefined macros. Among those are:

- `__FILE__`: the name of the file being processed
- `__LINE__`: the line number of this usage of the macro
- `__DATE__`: the date when the preprocessor is running
- `__TIME__`: the time at which the preprocessor is being run
- `__STDC__`: defined if this is an ANSI standard C compiler
- `__STDC_VERSION__`: the version of the standard C being compiled
- ...

### 7.2.3 Conditionals

We can also use #if, #else, #endif, #ifdef, #ifndef, ... to conditionally (not) remove sections from our file. The conditions need to use only things defined for the preprocessor with # define ... as this happens even before compile-time and we don't know about program values yet. #ifdef FOO und #ifndef BAR are shorthand notations for #if defined(FOO) and #if !defined(BAR).

Listing 36: C Preprocessor Conditionals

```
#if expression
... (text1)
#else
... (text2)
#endif

#ifdef FOO
... (text1)
#else
... (text2)
#endif

#ifndef BAR
... (text1)
#else
... (text2)
#endif
```

## 7.3 Assertions

A statement assert( <scalar expression> ); evaluates the expression at run time. If it is true the statement does nothing. If it is false, it prints the file, line, and expression, which failed and aborts execution. We need to include #include <assert.h> to use assert(...);.

assert statements are basically macros, because they are turned into code which actually achieves the just mentioned behavior at compile-time. But they are still evaluated at run-time. We can also add the compiler flag -DNDEBUG to remove all asserts during compilation. We would do so because they have no benefit for the user but slightly decrease runtime. The primary purpose is to support debugging for the developer.

Listing 37: Assertions Example

```
void array_copy(int a[], int b[], size_t count) {
  int i;
    assert(a != NULL);
    assert(b != NULL);
    for(i=0; i<count; i++) {
    a[i] = b[i];
  }
}
```

## 7.4 Modularity

In C, we achieve modularity with header `.h` files. But first, we need to distinguish declaration and definition. We already know about forward declaration. Thus, we basically understood what a declaration is: It states that something exists somewhere, and declares how it looks, i.e., specifies its signature. But only the definition says what it actually is/does.

Listing 38: Declaration vs. Definition in C

```c
// declaration
char *strncpy(char *dest, const char *src, size_t n);
// definition
char *strncpy(char *dest, const char *src, size_t n)
{
  size_t i;
  for (i = 0; i < n && src[i] != '\0'; i++)
  {
    dest[i] = src[i];
    }
  for ( ; i < n; i++)
  {
    dest[i] = '\0';
  }
    return dest;
}
```

During compilation, C deals with compilation units. A compilation unit is a `.c` file and everything it includes, i.e., the file after preprocessing. For declarations we have two annotation options:

- `extern`: the definition is either in this compilation unit or somewhere else (default)
- `static`: the definition is in this compilation unit and can't be seen outside

So, how does this help us with modularity in C? A module is a self-contained piece of a larger program. It consists of...

- externally visible stuff: functions, typedefs, global variables, cpp macros, ... (the client should/can use this)
- internal stuff: functions, types, variables, ... (the client should not look at this)

The externally visible stuff is called the module's interface. In C, we specify such interfaces with header files.

Some module `foo` has its interface defined in `foo.h` via declarations for the externally visible stuff. Clients in foo then `#include "foo.h"` to know about what `foo` can do. But `foo.h` does not provide and definitions. The definitions and only internally visible declarations are defined in the `foo.c` file.

Listing 39: Avoiding Including `.h` Files Multiple Times

```c
#ifndef __FILE_H_ // __FILE_H_ if chosen uniquely for each header file
#define __FILE_H_
...
#endif
```

## 7.5 goto

The goto statement was dominant in very early programming. However, today it is almost never a good idea to use it. On the one hand, there is little/no performance gain to other programming constructs. On the other hand, it makes code very confusing and difficult to understand. Still, there certain cases where goto is helpful.

### 7.5.1 Early Termination of Nested Loops

When having nested loops, other languages offer labels to break some outer loop. We don't have this option in C. Instead, we would have to set some flag variable and then also break the outer loop if that flag is set. But this is cumbersome. Instead we can jump out directly with a goto statement.

```
// cumbersome version without goto
int a[MAXROW][MAXCOL]
int i,j;
bool found=false;
for(i=0; i< MAXROW; i++) {
  for(j=0; j < MAXCOL; j++) {
    if (a[i][j] == 0) {
      found=true;
      break;
    }
  }
  if (found)
    break;
}
if (found) {
  <do processing on i and j>
} else {
  <do processing when not found>
}

// improved with goto
int a[MAXROW][MAXCOL]
int i,j;
for(i=0; i< MAXROW; i++) {
  for(j=0; j < MAXCOL; j++) {
    if (a[i][j] == 0) {
      goto found;
    }
  }
}
<do some processing when not found>
found:
<do some processing on i and j>
```

### 7.5.2 Nested Cleanup Code

Consider the case where we have a sequence of operations. And if any one of them fails, we need to undo al the previous steps. This might happen if some operation on a data structure fails and we don't want to corrupt it. Or if we allocate memory in multiple steps and want to free what we allocated in earlier steps if one allocation fails.

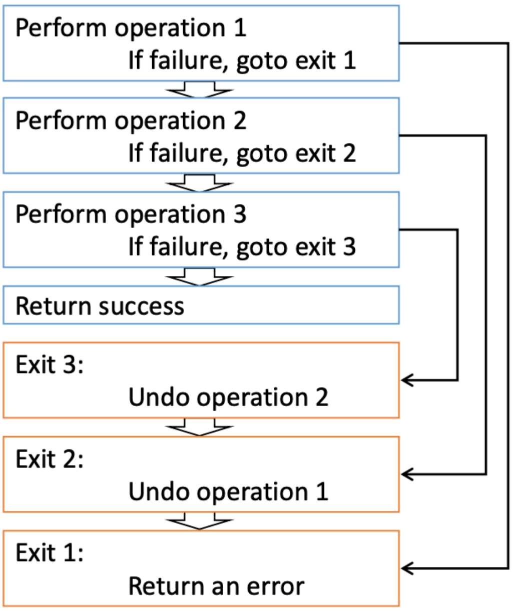


Figure 15: Nested Cleanup with goto

# 8 Basic x86 Architecture

## 8.1 Instruction Set Architectures (ISAs)

When talking about architecture, we mostly refer to the Instruction Set Architecture (ISA). It is the part of the processor design that one needs to understand to write assembly code for the processor. It is the 'contract' between the processor manufacturer and the programmer/compiler. ISAs specify the set of instructions, available register, addressing modes, data formats for processing units, etc. Some ISAs are: x86, RISC-V, MIPS, ia64, VAX, Alpha, ARM, ...

A microarchitecture, on the other hand, is a specific implementation of the ISA. The uarch then defines some aspects, which the developer/compiler doesn't need to know to create correct programs. Such as the cache size, core frequency, etc.

There have been various ISAs in the past. IBM S/360 has been the first to adapt the just described separation between ISA and uarch. When writing C programs, knowing the ISA is sufficient to write correct code. However, considering the uarch unlocks additional opportunity for performance optimizations.

ISA is a form of abstraction for the programmer. It specifies the instructions and assembly language to create programs for uarchs that implement this ISA. How the computer works looks different, depending on the abstraction layer.

- Above the ISA: This is about how to program the machine.

  We see a processor state as registers, memory, ... We can use instructions (`addl`, `movq`, `leal`, ...), which are presented in bytes to the processor to modify the state. Instructions appear to execute in-order.

- Below the ISA: This consider what needs to be built to execute programs.

  Doesn't need to conform to sequential instruction execution or any other requirement of the ISA as long as it appears it does. Code may be executed out-of-order. There may be more/different registers than the ISA specifies, which are, however, not modifiable by the program.

### 8.1.1 Complex Instruction Set Computer (CISC)

This was dominant throughout the mid-80s. The basic philosophy was: Add instructions to perform 'typical' programming tasks. So if some complex operation was common, one would usually introduce an instruction to cover this complex task with one instruction. Some characteristics:

- stack-oriented instruction set (pass arguments and return pointer via stack, explicit push/pop instructions)
- arithmetic instructions can access memory (complex address calculation)
- condition codes ('memorize' side effects of arithmetic and logical instructions)

### 8.1.2 Reduced Instruction Set computer (RISC)

This was popularized at Stanford and Berkeley as folks started to challenge the idea of adding instructions for everything and having very complex systems. Instead, they opted for fewer, simpler instructions. One usually needs more instructions to get things done, but instructions can also be executed faster and with simpler hardware. Some characteristics:

- register-oriented instruction set (more registers, used for arguments/return pointers/temporaries/...)
- load-store architecture (memory access separated with specific instructions)
- no condition codes
- conditional instructions (less branches, higher code density)
- explicit delay slots (e.g. MIPS) - quickly abandoned

### 8.1.3 CISC vs. RISC

RISCs usually have highly uniform operations (all instructions have same length, take similar time to execute, operations between registers, operate on 64 or 32 bit quantities, ...). Note that the variable instruction length of CISCs has little impact on execution performance today.

CISC proponents say they are easy for the compiler and provide smaller code sizes. RISC proponents say they are better for compiler optimizations and run fast with simple chip design. But today the choice of not a technical issue as basically anything can run fast. Still, for embedded processors RISC still makes sense for size, cost, and power constraints.

## 8.2 x86 History

The x86 architecture was born with the 8086 processor (1978), which was the first 16-bit processor. (x86-16 architecutre) Later came the 80386 (1985), which extended the capabilities to 32 bits. This was capable of running Unix. Modern linux for 32-bit uses no instructions introduced after this. (x86-32/ia32 architecture) The Pentium 4F (2005) introduced 64-bit processors for Intel. (x86-64/em64t architecture) Throughout those advancements, clock speeds increased noticeably just as the transistor count did.

Some features which were added to the x86 ISA over time are:

- instructions for multimedia operations (parallel operations on 1, 2, 4-byte data)
- more efficient conditional operations
- virtualization extensions
- multiprocessor synchronization

## 8.3 Basics on Machine Code

When looking as x86 assembly, we will consider the AT&T syntax, which is most commonly used for Unix. The Intel syntax is more present when using windows and just switches the order of the operands. A processor is very complicated but for writing assembly a simplified understanding is sufficient. We have the CPU, which consists of:

- The Program Counter (PC) that contains the address of the next instruction. This ist the `rip` register on x86-64.
- The register file contains the registers accessible and heavily used by programs.
- condition codes store status information/side effects from arithmetic and logic operations for conditional branching.
- Then we have the memory which provides the instructions to the processor. Also, we can provide addresses to get the data stored at the specified location. Through memory we have access to code, user data, (some) OS data, etc. The stack and heap are locate in memory.

### 8.3.1 Asembling

After having written a program, it is first compiled into assembly. To compile a `.c` file into assembly (an `.s` file), we can use the `-S` flag: `gcc -S code.c`. When also adding `-O0` we turn of optimizations and get a raw translation of our program. What we will see after this can be understood after reading further about instructions/operations, ... below.

Afterwards, the assembly is translated into an `.o` object file in which every instruction is binary encoded. This is a nearly-complete image of the executable code.

Notice that these steps can also (somewhat) be reserved. Disassembling turns an object file intro the corresponding instructions. However, some part of the assembly `.s` file gets removed during conversion to an `.o` file. Thus, the disassembled `.s` file usually misses some stuff from the original `.s` file. We can disassemble with `objdump -d p`. This works on objects file but also executables, which are created after linking.

One can get a similar result by using the gdb debugger. We do so by writing `disassemble name` in a gdb session to disassemble the function `name`. With gdb we can also examine bytes at some memory location. We can write `x/30bx sum` to examine (`x/`) 30 bytes (`30b`) in hex (`x`) at sum (`sum`).

Finally, the linker resolves references between `.o` files and combines the `.o` file with static run-time libraries. Some libraries are also dynamically linked when the program begins execution.

### 8.3.2 Assembly Data Types

In x86 assembly we can work with integer data of 1, 2, 4 or 8 bytes. This stores whatever we just want to represent as bits in memory. Then we also have floating point data of 4, 8, or 10 bytes. This is for floating point numbers and will be considered later. We do not have aggregated types (arrays, ...). Those are all build on top of the basic integer types etc. from the compiler.

### 8.3.3 Assembly Code Operations/Machine Instructions

With instructions we perform operations to alter the state of the system. Using those, we can:

- perform arithmetic/logic functions on registers or memory data
- transfer data between memory and register
- transfer control (unconditional jumps to/from procedures, conditional branches)

With x86 assembly, instructions have the format `op operand1, operand2`, where not all operands may be used with all instructions. Registers are identified by a unique name. With AT&T syntax, the result of an operaiton on `operand1` and `operand2` is put in `operand2`.

## 8.4 x86 Architecture

### 8.4.1 Registers

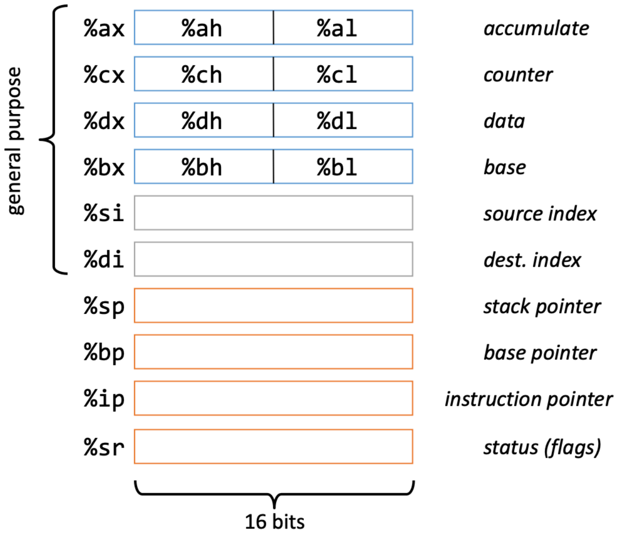


Figure 16: Registers of Intel 8086 / x86-16 Architecture

With `%ah`, `%al`, etc. we can access the 8 high or low bits of a 16 bit unit. Here we have 6 general purpose registers. The six registers had some special meaning in earlier versions. That's the reason for the strange naming. They are acronyms for their original meaning.

All other register cannot be freely used. The `%ip`, `%bp`, `%sp`, and `%sr` are special-purpose and must be used according to their definition/meaning.

The `x` in addresses stands for extended, because originally, we couldn't access the the four top 16 bit registers on the preceding 8-bit machine.

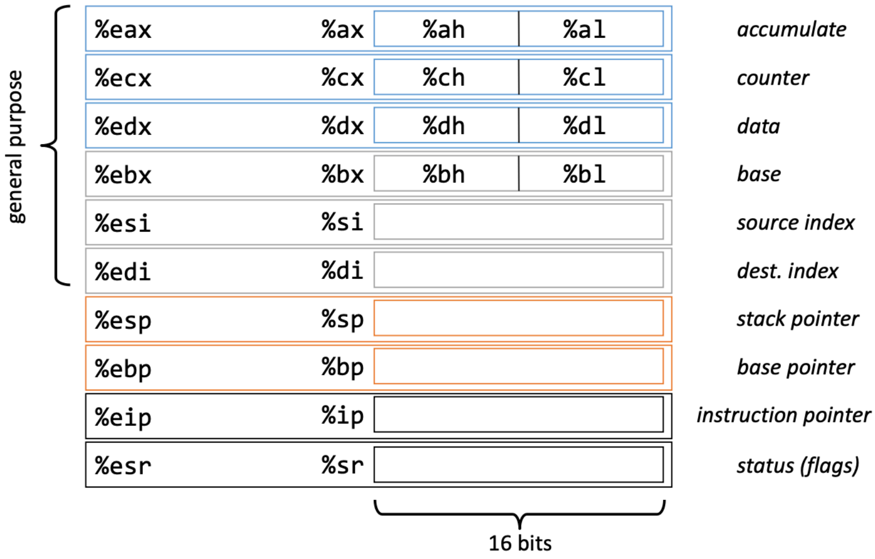


Figure 17: Registers of Intel 80386 / x86-32 Architecture

With the step to 32 bits with the 80386, the register size was increased. An `e` is preceding when wanting to access the 32 bit version and not just work on the lower 16 bits with the same names as before. The `e` stands again for extended.

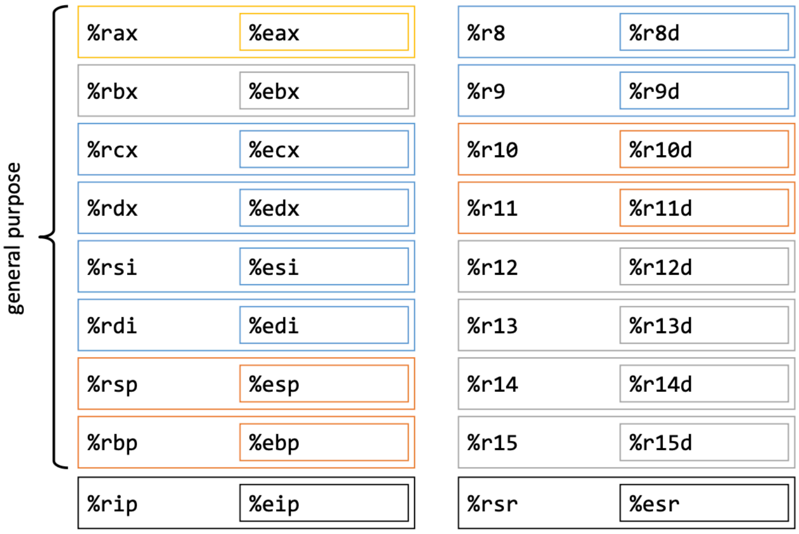


Figure 18: Registers of x86-64 Architecture

With the step to x86-64 the registers were extended again by adding an `r` in front of each naming. Another change was that the ordering of the first four registers now is alphabetically. Furthermore, the `%rsp` and `%rbp` (stack pointer and base pointer) have become general-purpose registers too. Anyway, they are still used for their original purposes almost always. Finally, an additional 8 general purpose registers have been added. Conveniently those are numbered in chronological order.

If we set the lower 8 or 16 bits, the upper bits remain unchanged.
But if we set the lower 32 bits, the upper 32 bits will be zeroed out.

### 8.4.2 Move Instructions

We use the `movx source, dest` instruction to move data (where `x` is replaced with an element from $\{b, w, l, q\}$).

- `b` is 1-byte.
- `w` is a word / 2-bytes.
- `l` is a long-word / 4-bytes.
- `q` is a quad-word / 8-bytes.

We also have `movABx`, where `x` is as above. `A` is either s for sign extending the remaining-bits of the register (if only writing to lower bits) or `z` for zeroing out. The `B` identifeis what we choose as input, i.e., what size our source has.

We have different types of data we can choose as operands:

- immediate: constant integer data

example: `$0x400, $-533` (we have to write the prefix `$`).

Those values are encoded with 1, 2, 4, or 8 bytes

- register values (we can choose one of the 16 general purpose registers)

  example: `%eax, %r14d`

- values from memory (the number of bytes from memory depends on the instruction type)

  example: `(%rax)`

We can combine those data types mostly arbitrarily. But we can't specify an immediate as a destination or have to memory accesses in the same operation.

Memory addressing is more capable than just staed. It follows the general scheme of `D(Rb ,Ri,S)`, where `D` is a constant displacement (offset), `Rb` is the base register address (any of the general purpose integer registers), `Ri` is the index register, and `S` specifies a scale as one of the values 1, 2, 4, or 8. The memory access corresponds to this: `Mem[Reg[Rb] + S*Reg[Ri] + D]`. We may leave out some of those 'parameters'.

In addition to memory accesses, we also have the `lea operand1, operand2` operation. Ir computes the address according to the same scheme but instead of accessing that location, the address is saved to the second operand. So this corresponds to `Reg[Rb] + S*Reg[Ri] + D`.

## 8.5 x86 Integer Arithmetic

Integer arithmetics in x86 assembly generally uses two-operand instructions. As with move instructions, the suffix determines the size we operate on. I.e., if there is an `l` in the end, we operate on long-words (32 bits) etc.

| **instruction** | **format** | **computation** |
|---|---|---|
| `addl` | `src, dest` | `dest <- dest + src` |
| `subl` | `src, dest` | `dest <- dest - src` |
| `imull` | `src, dest` | `dest <- dest * src` |
| `sall` | `src, dest` | `dest <- dest << src` |
| `sarl` (arithmetic shift) | `src, dest` | `dest <- dest >> src` |
| `shrl` (logical shift) | `src, dest` | `dest <- dest >> src` |
| `xorl` | `src, dest` | `dest <- dest ^ src` |
| `andl` | `src, dest` | `dest <- dest & src` |
| `orl` | `src, dest` | `dest <- dest \| src` |

| **instruction** | **format** | **computation** |
|---|---|---|
| `incl` | `dest` | `dest <- dest + 1` |
| `decl` | `dest` | `dest <- dest - 1` |
| `negl` | `dest` | `dest <- -dest` |
| `notl` | `dest` | `dest <- ~dest` |

We can also use `leal` for (basic) arithmetic instructions. Because `leal` is usually very optimized in hardware because it uses the same frequently used unit as for memory accesses, this is rather fast.

The compiler is also quite smart. So when writing C code it might rearrange and change stuff so that the observed behavior is identical while the specific computation is different.

Listing 40: Compiler Evaluation

```
int logical(int x, int y) {
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
// is assembled into
logical:
  movl %edi, %eax
  xorl %esi, %eax
  sarl $17, %eax
  andl $8185, %eax
  ret
```

## 8.6 Condition Codes

The x86 architecture specifies some additional special-purpose one-bit registers, which can't be set explicitly. They are set implicitly/as a side effect by arithmetic operations performed in the ALU. So 'leal' doesn't set those bits. But a regular add or multiply does. They are always set to indicate whether their respective condition holds on the produced output. The condition codes are then used for branching etc.

- CF: Carry Flag (overflow for unsigned)

  set if carry out from the most significant bit (unsigned overflow)

- ZF: Zero Flag

  set if $t == 0$ ($t$ is the operation result)

- SF: Sign Flag (for signed)

  set if $t < 0$ ($t$ is the operation result)

- OF: Overflow Flag (for signed)

  set if 2's complement (signed) overflow, `(a>0 && b>0 && t<0)|| (a<0 && b<0 && t>=0)` ($t$ is the operation result)

There are also instructions, which compute some value, i.e., do some arithmetic operation (at least as long as necessary to compute the condition codes) without actually saving the result.

- `cmpl src2, src1` computes `src1-src2` and sets the condition codes without storing the result.

- `testl src2, src1` computes `src1 & src2` and sets the condition codes without storing the result

Conditions codes are mostly used for conditional branching. But we can also extract the bit values and put them into general purpose registers with special instructions.

| **instruction** | **condition** | **description** |
|---|---|---|
| `sete` | `ZF` | equal / zero |
| `setne` | `~ZE` | no equal / not zero |
| `sets` | `SF` | negative |
| `setns` | `~SF` | non-negative |
| `setg` | `~(SF^OF)&~ZF` | greater (signed) |
| `setge` | `~(SF^OF)` | greater or equal (signed) |
| `setl` | `(SF^OF)` | less (signed) |
| `setle` | `(SF^OF)\|ZF` | less or equal (signed) |
| `seta` | `~CF&~ZF` | above (unsigned) |
| `setb` | `CF` | below (unsigned) |

We use the same syntax for doing conditional jumps.

| **instruction** | **condition** | **description** |
|---|---|---|
| `jmp` | 1 | unconditional |
| `je` | `ZF` | equal / zero |
| `jne` | `~ZE` | no equal / not zero |
| `js` | `SF` | negative |
| `jns` | `~SF` | non-negative |
| `jg` | `~(SF^OF)&~ZF` | greater (signed) |
| `jge` | `~(SF^OF)` | greater or equal (signed) |
| `jl` | `(SF^OF)` | less (signed) |
| `jle` | `(SF^OF)\|ZF` | less or equal (signed) |
| `ja` | `~CF&~ZF` | above (unsigned) |
| `jb` | `CF` | below (unsigned) |

# 9 Compiling C Control Flow

In this section we consider how control flow structures from the C language are translated to (x86) assembly.

## 9.1 if-then-else statements

If conditions work by testing the opposite conditions and if the opposite is true, we do a conditional jump to the section after the loop content. If we have an else condition, then the assembly for that comes after the if code and when the if-code finishes, it jumps after the else section. The ternary operator is compiled to an if statement as just described.

Listing 41: If Statement in C

```c
#include <stdio.h>
int putmax(int x, int y) {
  int result;
  if (x> y) {
    result = printf("%d\n", x
       );
  } else {
    result = printf("%d\n", y
       );
  }
  return result;
}
```

Listing 42: Compiled If Statement

```
putmax:
  subq $8, %rsp
  cmpl %esi, %edi
  jle .L2
  movl %edi, %edx
  movl $.LCO, %esi
  movl $1, %edi
  movl $0, %eax
  call __printf_chk
  jmp .L3
.L2:
  movl %esi, %edx
  movl .LC0, %esi
  movl $1, %edi
  movl $0, %eax
  call __printf_chk
.L3:
  addq $8, %rsp
  ret
```

However, in certain cases the assembly can be optimized. Specifically if the code blocks of the if/else block don't call functions or have other side effects. Then it is often faster to compute both cases and then use a conditional move to keep the right result in the end. This is used for many very short if/else conditionals. This is more efficient because we don't have the overhead from branching.

Listing 43: If Statement in C Optimized

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

Listing 44: Compiled If Statement Optimized

```
absdiff:
  movl %edi, %eax
  subl %esi, %eax
  movl %esi, %edx
  subl %edi, %edx
  cmpl %esi, %edi
  cmovle %edx, %eax
  ret
```

## 9.2 do-while loops

Here we use a backward branch to continue looping. I.e., we execute our code once and if the condition then holds, we again jump to the beginning of the loop content.

Listing 45: Do-While Loop in C

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

Listing 46: Compiled Do-While Loop

```
fact_do:
  movl $1, %eax
.L3:
  imull %edi, %eax
  subl $1, %edi
  cmpl $1, %edi
  cmpl $1, %edi
  jg .L3
  rep ret
```

## 9.3 while loops

while loops can be implemented in various ways (just as other control flow structures can be too). gcc's approach is to test the condition initially. If it doesn't hold we jump to the section after the loop. If it does, a do while basically follows.

Listing 47: While Loop in C

```c
int fact_while(int x) {
  int result = 1;
  while (x>1) {
    result *= x;
    x = x - 1;
  };
  return result;
}
```

Listing 48: First Compiled While Loop

```c
int fact_while_goto1(int x) {
  int result = 1;
  if (!(x>1))
    goto done;
Loop:
  result *= x;
  x = x - 1;
  if (x>1$
    goto Loop;
done:
  return result;
}
```

A newer style for while loops (jump-to-middle while) is to not have separate code for the initial condition test. Instead we do an unconditional jump to the beginning of the test of the do-while. So, we have basically a do while with one additional jump. This approach is chosen today, because unconditional jumps/‘goto‘s don't incur any noticeable performance penalty these days.

Listing 49: Second Compiled While Loop

```c
int fact_while_goto2(int x) {
  int result = 1;
  goto middle;
Loop:
  result *= x;
  x = x - 1$;
middle:
  if (x>1)
    goto Loop;
  return result;
}
```

## 9.4 for loops

A for loop is basically a while loop with some initial variable setup and some variable update and test at the end of each iteration to check whether to execute the next iteration.

Listing 50: $p$-th power of $x$

```
for (result = 1; p != 0; p = p>>1) {
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

Listing 51: Compiled For Loop

```
  result = 1;
  if (p == 0)
    goto done;
loop:
  if (p & 0x1)
    result *= x;
  x = x*x;
  p = p >> 1;
  if (p != 0)
    goto loop;
done:
```

Listing 52: Compiled For Loop - Modern

```
  result = 1;
  goto middle;
loop:
  if (p & 0x1)
    result *= x;
  x = x*x;
  p = p >> 1;
middle:
  if (p != 0)
    goto loop;
dome:
```

## 9.5 switch

### 9.5.1 jump tables

This is efficient if we only have a small range of values as cases to check. We then add a static jump table with target addresses for each case and index this table with the variable on which to do the switch after doing a sanity check that we don't definitely have the default case in which we jump to it directly.

Listing 53: Jump Table Setup

```
  .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
```

Listing 54: Switch Statement Compiled

```
switch_eg:
  movq %rdx, %rcx
  cmpq $6, %rdi
  ja .L8
  jmp *.L4(,%rdi,8)
```

### 9.5.2 sparse switch statements via binary search tree

But if we have many cases or large values for our cases the previous approach is undesirable, because it consumes a lot of memory and we might run into cache issues. The following example would generate a 1000 entry jump table, for instance.

Listing 55: Switch Statement

```
int div111(int x) {
  switch(x) {
    case 0: return 0;
    case 111: return 1;
    case 222: return 2;
    case 333: return 3;
    case 444: return 4;
    case 555: return 5;
    case 666: return 6;
    case 777: return 7;
    case 888: return 8;
    case 999: return 9;
    default: return -1;
  }
}
```

Listing 56: Compiled Switch Statement as Binary Search Tree

```
div111:
  cmpl $444, %edi
  je .L3
  jle .L28
  cmpl $777, %edi
  je .L10
  jg .L11
  cmpl $555, %edi
  movl $5, %eax
  je .L5
  cmpl $666, %edi
  movb $6, %al
  jne .L2
.L5: // done
  rep ret
.L2: // default case
.L3: // case 444
.L28: // cases 0-333
.L10: // case 777
.L11: // cases 888-999
// ...
```

## 9.6 procedure calls and returns

We already discussed the stack as the area of memory where local variables etc. are managed. We say that it is managed with stack discipline, i.e., by respecting conventions on how to use the stack. Historically, the stack pointer had to be located in %rsp (or their lower-bit equivalents) because the concept of a stack is baked in to x86. Today we could use a different register but still stick to the convention.

There exist some basic instructions to interact with the stack through the stack pointer. In practice we only use those instructions in a limited setting. If we want to allocate a larger region on the stack, we usually work on the stack pointer directly. However, for single values this still is used.

- pushl src: Fetches the operand at src, decrements %rsp by 4 and then write the operand to that address. Equivalently, we could use pushw, etc.

- popl dest: Reads the operand at address %rsp and increments %rsp by 4, and write the operand to dest.

The stack is used to support procedure calls and returns. When calling a function, we write call addr, where addr is the location of the function in memory to which the instruction pointer is changed. We can also use a label in assembly language. That adds the address of the next instruction to the stack and jumps to the target address. So, the return address for a function is stored on the stack. Returning can be done with `ret`, which reads the return address, removes it from the stack and jumps to that address.

This is a variation from the original wheeler jump. It stored the return address in a special register. On other ISAs we have that with 'jump and link' instructions. Then, this register is callee saved, or at least it must make sure to return to that address. x86 is different, because we don't store the return address in a special register (i.e., link to it for returning later) but simply put it on the stack instead.

Furthermore, when passing arguments to a function and their size/number exceeds what we can store in the designated registers by the calling conventions, we put them on the stack before the return address.

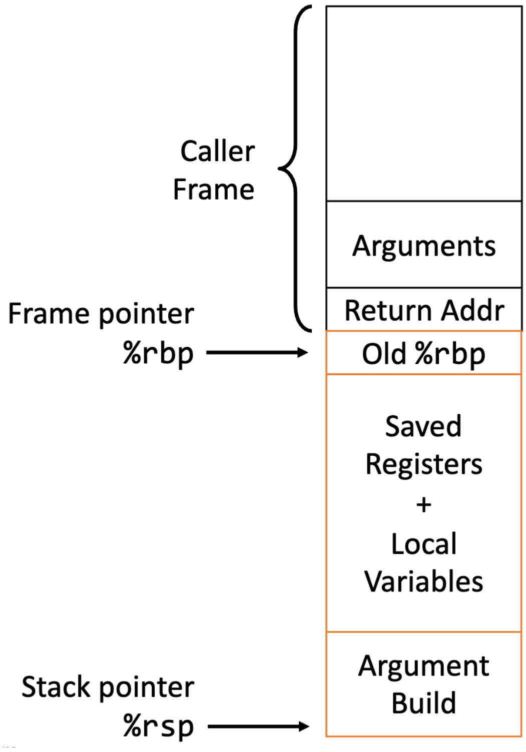


Figure 19: x86-64 Full Stack Frame

Usually, the stack is aligned. Mostly to 16 bytes. Thus, a function can assume that this holds when it is called and must make sure that it holds again once it calls a function itself.

## 9.7 calling conventions

In x86 and especially other ISAs we have plenty registers and the ISA does not define how to use them. But the calling conventions specify a common schema to pass arguments to a method, provide a return value etc. This is helpful to be able to link binaries/assembly from different compilers etc. The calling conventions specify a common interface between the caller and the callee.

The calling conventions boil down to the special meaning of some registers and rules on who (caller or callee) saves which registers. The procedure executing ‘call‘ is the caller and the procedure which we jump to is the callee. Caller saved registers are those registers saved and restored by the caller (by using the stack) before calling another procedure. Callee saved registers are those registers saved and restored by the callee (by using the stack) before using them itself.

| name | meaning | who saves? |
|---|---|---|
| `%rax` | return value, #args | caller |
| `%rbx` | base pointer | callee |
| `%rcx` | 4th argument | caller |
| `%rdx` | 3rd argument | caller |
| `%rsi` | 2nd argument, 2nd return value | caller |
| `%rdi` | 1st argument | caller |
| `%rsp` | stack pointer | callee |
| `%rbp` | frame ptr | callee |
| `%r8` | 5th argument | caller |
| `%r9` | 6th argument | caller |
| `%r10` | static chain ptr | caller |
| `%r11` | used for linking | caller |
| `%r12` | | callee |
| `%r13` | | callee |
| `%r14` | | callee |
| `%r15` | | callee |

This table specifies six registers for arguments. If we need to pass more than 6 parameters, the rest is passed on the stack as specified in the previous subsection. These registers (if not used for parameters) can be used as caller saved registers. All references to the stack frame are done via the stack pointer `%rsp`. Thus, we don't need to update the `%rbp` (base pointer).

If we need to (re)store some registers or otherwise need to handle more data than fits in the available registers, we need to work on the stack. If we are a leaf procedure, we can access a few bytes beyond the stack pointer in the red zone without risking a page fault or stack overflow. The red

However, if we call another procedure ourselves or want to be sure, we should allocate a stack frame. This is done by decrementing the stack pointer once according to our memory requirements & stack alignment requirements. Once we want to return from the function, we can deallocate memory from the stack by increasing the stack pointer by the same amount we decrement it earlier.

### 9.7.1 Examples

**Factorial** Here, the register `%rax` is used within the function without worrying about existing data. The register `%rbx` is used too, but we save the contents before and put it back afterwards.

Listing 57: Factorial in C

```c
int rfact(int x) {
  int rval;
  if (x<=1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

Listing 58: Factorial Compiled

```
rfact:
  pushq %rbx
  movl %edi, %ebx
  movl $1, %eax
  cmpl $1, %edi
  jle .L2
  leal -1(%rdi), %edi
  call rfact
  imull %ebx, %eax
.L2:
  popq %rbx
  ret
```

**Locals in the Red Zone** .

Listing 59: Swap Using Local Array

```c
void swap_a(long *xp, long *
    yp) {
  volatile long loc[2];
  loc[0] = *xp;
  loc[1] = *yp;
  *xp = loc[1];
  *yp = loc[0];
}
```

Listing 60: Swap Compiled

```
swap_a:
  movl (%rdi), $rax
  movq %rax, -24(%rsp)
  movq (%rsi), %rax
  movq %rax, -16(%rsp)
  movq -16(%rsp), %rax
  movq %rax, (%rdi)
  movq -24(%rsp), %rax
  movq %rax, (%rsi)
  ret
```

**Non-Leaf Function Without Stack Frame** .

Listing 61: Swap Without Stack Frame

```c
long scount = 0;

void swap_ele_se(long a[],
    int i) {
  swap(&a[i], &a[i+1]);
  scount++;
}
```

Listing 62: Swap Compiled

```
swap_ele_se:
  movslq %esi, %rsi
  leaq (%rdi, %rsi, 8), %rdi
  leaq 8(%rdi), %rsi
  call swap
  incq scount(%rip)
```

**Call Using a Jump** This optimizes a function call if it does not require a stack frame itself and does not need to result execution after calling another function. Then that function can just return the original caller.

Listing 63: Swap Using Local Array

```c
long scount = 0;

void swap_ele(long a[], int i ) {
  swap(&a[i], &a[i+1]);
}
```

Listing 64: Swap Compiled

```
swap_ele:
  movslq %esi, %rsi
  leaq (%rdi, %rsi, 8), %rdi
  leaq 8(%rdi), %rsi
  jmp swap
```

## 9.8 variadic functions

We have used `printf(...)` at various occasions before. Now that we have seen the calling conventions and how to pass arbitrarily many arguments, we can understand how this function processes its parameters.

In C, `printf` is declared like this: `int printf(const char *format, ...);`, which is C's syntax for variadic functions. Let's consider `void print_integers(unsigned num_ints, char *msg, ...);` to make it easier to understand what's going one. We may invoke the following calls, which returns the expected values.

- `print_integers(0, "Hello");`
- `print_integers(1, "The␣answer...", 42);`
- `print_integers(2, "A␣couple", 2, 4);`
- `print_integers(10, "Quite␣a␣few", 10, 8, 7, 6, 5, 4, 3, 2, 1);`

Listing 65: Implementation of `print_integers`

```c
#include <stdio.h>
#include <stdarg.h> // library to use work with variable amounts of arguments

void print_integers(unsigned num_ints, char *msg, ...)
{
  printf("%s\n", msg);
  printf("About␣to␣print␣%u␣ints:\n", num_ints);
  va_list ap; // variable which tracks the currently considered argument (similar to an iterator)

  va_start(ap, msg); // initialization of ap based on the last fixed argument
  for(int i = 0; i<num_ints; i++)
  {
    int j = va_arg(ap, int); // return the next argument cast to an int
    printf("int␣%d␣=␣%d\n", i, j);
  }
  va_end(ap); // free up the iterator
}
```

va_arg clearly can't be a function as it accepts a type as a parameter. It may be a macro. On modern machines, gcc turns this into compiler intrinsic. So it is replaced with code provided by the compiler. We can see this when looking at the definitions in stdarg: #define va_arg(v,l)__buildin_va_arg(v,l).

# 10 Compiling C Data Structures

## 10.1 One-Dimensional Arrays

If we allocate for `T A[L]` (type `T`, length `L`), we allocate a continuous region of `L * sizeof(T)` in memory.

Listing 66: Array Access in C

```c
int get_digit(zip_dig z, int dig) {
  return z[dig];
}
```

Listing 67: Array Access Compiled

```
get_digit:
  movsql %esi, %rsi
  movl (%rdi,%rsi,4), %eax
  ret
```

Figure 20: Looping over an Array in C

```c
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

```c
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while (z <= zend);
  return zi;
}
```

```
zd2int:
.LFB0:
  endbr64
  leaq 20(%rdi), %rcx
  xorl %eax, %eax
.L2:
  leal (%rax,%rax,4), %edx
  movl (%rdi), %eaxaddq $4, %rdi
  leal (%rax,%rdx,2), %eax
  cmpq %rcx, %rdi
  jne .L2
  ret
```

## 10.2 Nested Arrays

If we have `T A[R][C]`, we have a 2D array of data type `T` with `R` rows and `C` columns. If type `T` requires `K` bytes, this has size `R*C*K` bytes in memory. It is arranged in row-major order.

```
typedef int[][] zip_dig;
zip_dig pgh[4] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

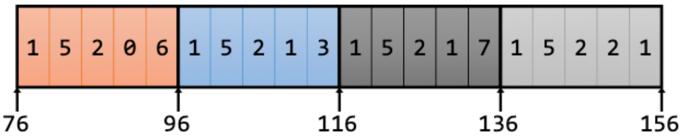


Figure 21: 2D Array in Memory

To access some element in a row, we add `j*K` for the *j*-th element in that row to the base address for that row. In total, to access `A[i][j]` (of elements of type `T` with `K` bytes each), we get `A + i*(C*K)+ j*K = A + (i*C+j)*K`.

Listing 68: 2D Array Access in C

```
int get_pgh_digit(int index, int dig)
{
  return pgh[index][dig];
}
```

Listing 69: 2D Array Access in Assembly

```
get_pgh_digit:
  endbr64
  movslq %edi, %rdi
  movslq %esi, %rax
  leaq (%rdi,%rdi,4), %rsi # 5 * index
  addq %rax, %rsi # dig + 5 * index
  leaq pgh(%rip), %rax
  movl (%rax,%rsi,4), %eax # *(pgh + 4*(dig + 5*index))
  ret
```

When accessing arrays and interpreting `a[i][j]` as just described, it would be reasonable that in our `pgh` example `pgh[0][19]` and `pgh[4][-1]` correspond to the same element. In practice this holds for most compilers but is not guaranteed by the C standard. The C standard (C99) only guarantees the expected behavior for access at most 1 beyond the bound of the array. I.e., `a[2][5]` is guaranteed while `a[2][6]` is not in our example.

## 10.3 Multi-Level Arrays

Variable multi-level arrays can contain the same data as nested arrays. However, multi-level arrays require more memory. With this cost, we ’buy’ additional flexibility.

Listing 70: Multi-Level Array in C

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
int *univ[3] = {mit, cmu, ucb};
```

This has the benefit of being more flexible. The subarrays may have different length, for instance. But it also requires more memory as we now need to store the pointers to the subarrays in the main arrays. Also, we require multiple memory accesses to get to the actual value so that the access latency is increased.

Despite this being different than nested arrays, we can still use the same indexing syntax in C. But it will be compiled to different assembly. An access `array[a][b]` to an array like `int array[20][30]` would be compiled to something implementing `Mem[Mem[array+8*a]+4*b]`.

Listing 71: Multi-Level Array Indexing in C

```c
int get_univ_digit(
  int index,
  int dig
) {
  return univ[index][dig];
}
```

Listing 72: Compiled Listing 71

```
get_univ_digit:
  endbr64
  movslq %edi, %rdi
  leaq univ(%rip), %rax
  movslq %esi, %rsi
  movq (%rax,%rdi,8), %rax
    # Mem[univ+8*index]
  movl (%rax,%rsi,4), %eax
    # Mem[... +4*dig]
  ret
```

Notice that with nested arrays, we were able to go beyond the borders of one array to access the last/first element of another array. But this does not hold here, because we are not guaranteed that the subarrays are put continuously in memory. The pointers may lead to arbitrary locations.

## 10.4 Dynamic Arrays

Above we have considered array with static sizes. But though explicit memory management, we can also work with dynamic array sizes. But we then must do the index computation explicitly. Also accessing elements is costly, because we must do a multiplication.

Listing 73: Dynamic Array Allocation

```c
int * new_var_matrix(int n) {
  return (int *)
    calloc(sizeof(int), n*n);
}
```

Listing 74: Dynamic Array Indexing

```c
int var_ele(int *a, int i,
  int j, int n) {
  return a[i*n+j];
}
```

Notice that the function `int var_prod_ele(int *a, int *b, int i, int k, int n);`, which computes an entry of the matrix-matrix produce, may perform very different, depending on its implementation.

Listing 75: Inefficient Multiply

```c
  // per iteration:
  // - 3 multiplications
  // - 4 additions
{
  int j;
  int result = 0;
  for (j = 0; j < n; j++)
    result += a[i*n+j] * b[j*n+k
       ];
  return result;
}
```

Listing 76: Efficient Multiply

```c
  // per iteration:
  // - 4 multiplications
  // - 1 addition
{
  int j;
  int result = 0;
  int iTn = i*n;
  int jTnPk = k;
  for (j = 0; j < n; j++) {
    result += a[iTn+j] * b[jTnPk
       ];
    jTnPk += n;
  }
  return result;
}
```

## 10.5 Structures

For structures we allocate a contiguous region of memory for the members (of different types).

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

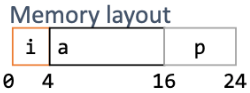


Figure 22: struct layout in memory

The memory layout of structures must validate alignment. Consider that section to see why this memory layout it valid and how other structures would be put in memory.

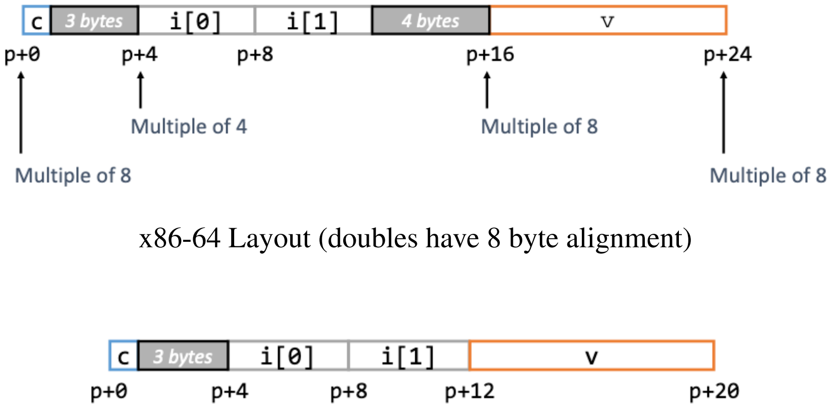
## 10.6 Alignment

Alignment describes the memory locations at which some data may reside in memory. In general it describes that a data type which requires K bytes must have an address, which is a multiple of K i.e., be K aligned. On early ARM, Alpha, ... this is required. On x86 this is treated depending on the platform but generally advised.

The concept of alignment helps performance, because memory is accessed by aligned chunks of 4 or 8 bytes. Alignment then simplifies providing data to the ALU and moving it within the system. The compiler usually inserts gaps in structures to ensure correct alignment of fields.

For structures, each element's alignment requirement must be satisfied. And the overall structure has alignment requirement K, where K is the largest alignment of any element. To guarantee this for the fields to be in memory in the specified order, padding is added in memory.

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



x86-64 Layout (doubles have 8 byte alignment)

ia32 Layout (doubles have 4 byte alignment)

Figure 23: struct Alignment

A good C programmer knows about this and writes the structs in such an order that padding is minimized. With structs, one may also specify bitfields. In those, one provides the size of each field in bits. But do not use these for explicit memory layout! The compiler can pack multiple fields into a single char/short/int/... We have no guarantee on the memory format! Padding, byte-ordering, event-bit ordering, ... may be unexpected.

Listing 77: Bitfields in C

```c
struct BitFieldExample {
    unsigned int field1 : 3; // 3 bits
    unsigned int field2 : 4; // 4 bits
};
```

## 10.7 Unions

We have already discussed unions and know that we can only use one field at a time. Thus, it is intuitive to understand that memory is allocated only according to the largest element.

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

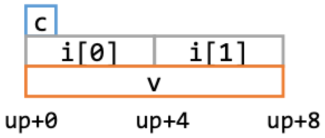


Figure 24: union memory layout

A common use case for unions is to access the bit pattern of some types. When casting floats/doubles to ints, the C compiler/machine tries approximate conversion by rounding. But if we want to access the bit pattern, we can store the data in a union and then read the int values.

Listing 78: Accessing Bit Pattern of Types through unions

```c
union bit_float {
  float f;
  unsigned u;
};

float bit2float(unsigned u) {
  union bit_float arg;
  arg.u = u;
  return arg.f;
}

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

# 11 Unorthodox Control Flow

In high-level languages we are familiar with many common programming techniques/structures. We work with OOP, threads, synchronization techniques, etc. While those are helpful in many situations, C unlocks many more possibilities and techniques. However, we have to implement them mostly manually. Here we will consider unhinging normal control flow to unlock possibilities not feasible in higher-level languages. Learning about such optimizations and learning such low-level concepts is essential to reach even better optimizations.

## 11.1 setjmp() and longjmp()

`setjmp` and `longjmp` allow us to leave regular control flow. With `setjmp` we can set a position in code and, later, go to that specific position with `setjmp` in a way that resets all registers etc. to the state that has been once we have been at that location in code the first time.

The way we save the information with `setjmp` about the location we jump to later mandates, that we can only jump there in a way that the stack has not yet been deallocated. I.e., we can only jump there from the function itself or subroutines.

### 11.1.1 setjmp

Calling `setjmp(jmp_buf env)` saves the current stack state (i.e., the position of the stack) and the environment (register values) in the buffer `env` and returns 0.

Listing 79: `setjmp` in C

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

### 11.1.2 longjmp

`setjmp(val)` changes to control flow to jump to the point in code saved earlier with `setjmp(..)`.. However, once we resume execution at the `setjump`, it does not return 0 but instead it returns `val` (if `val` is 0, it is turned into 1 - so one can't return 0). For each `setjmp(..)`. we can call `longjmp(..)`. only once.

As already mentioned in the introduction: By how this works under the hood, it is invalid to call `longjmp(..)`. if the function that called `setjmp(..)`. already returned!

Listing 80: `setjmp` and `longjmp` Toy Example

```c
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void second(void) {
  printf("second\n");
  longjmp(buf,1);
}

void first(void) {
  second();
  printf("first\n");        // does not print
}

int main() {
  if ( ! setjmp(buf) ) {
    first();              // when executed, setjmp returns 0
  } else {                  // when longjmp jumps back, setjmp returns 1
    printf("main\n");     // prints
  }
  return 0;
}
```

## 11.2 implementing setjmp() and longjmp()

### 11.2.1 setjmp

When calling `setjmp(env)`, we store (a) all callee saved registers, (b) the current state of the stack (i.e., the stack pointer), and (c) the return address in the `env` buffer.

Listing 81: `setjmp` in Assembly

```
/* Copyright 2011-2012 Nicholas J. Kain, licensed under standard MIT license */
setjmp:
  mov %rbx,(%rdi).  /* rdi is jmp_buf, move registers onto it */
  mov %rbp,8(%rdi)
  mov %r12,16(%rdi)
  mov %r13,24(%rdi)
  mov %r14,32(%rdi)
  mov %r15,40(%rdi)
  lea 8(%rsp),%rdx  /* this is our rsp WITHOUT current ret addr */
  mov %rdx,48(%rdi)
  mov (%rsp),%rdx   /* save return addr ptr for new rip */
  mov %rdx,56(%rdi)
  xor %eax,%eax   /* always return 0 */
  ret
```

### 11.2.2 longjmp

When `longjmp` is called, (a) all the previously stored registers are restored and (b) the stack pointer is set to its value as it was in the function calling `setjmp` and, thus, deleting all stack frames allocated after `setjmp` (assuming it did not return). Afterwards, `longjmp` returns to the return address of the `setjmp` call, i.e., the next instruction after `setjmp`. Before that is done, however, the return value is set according to the parameter to `longjmp` - assuming it is not 0.

Listing 82: `longjmp` in Assembly

```
/* Copyright 2011-2012 Nicholas J. Kain, licensed under standard MIT license */
longjmp:
  xor %eax,%eax
  cmp $1,%esi          /* CF = val ? 0 : 1 */
  adc %esi,%eax        /* eax = val + !val */
  mov (%rdi),%rbx      /* rdi is the jmp_buf, restore regs from it */
  mov 8(%rdi),%rbp
  mov 16(%rdi),%r12
  mov 24(%rdi),%r13
  mov 32(%rdi),%r14
  mov 40(%rdi),%r15
  mov 48(%rdi),%rsp
  jmp *56(%rdi)        /* goto saved address without altering rsp */
```

The `adc` instructions sets the output to the sum of the destination operand, the source operand, and the carry flat (set by the previous `cmp` instruction if `%esi` is 0).

## 11.3 Why Coroutines?

Now we begin to elaborate on the use of all of this. For what reason do we consider coroutines? We will discuss this based on an example. But coroutines are the foundation of almost all concurrent programming. We intend to read a file, which as been compressed and then run a parser/lexer on the decoded text to tokenize it.

The first stage to processing the compressed file is the **decompressor**, which is rather simple. If a character is repeated in the text, the compressed version stores the number of repetitions and the character.

Listing 83: Decompressor Code

```
while (1) {
  c = getchar();
  if (c == EOF)
    break;
  if (c == 0xFF) {
    len = getchar();
    c = getchar();
    while (len--)
      emit(c);
  } else
    emit(c);
}
emit(EOF);
```

This reads a character from the file through `getchar()` and then produces output for this using `emit(...)`.

The **lexer** distinguishes between alphabetic characters and everything else. It outputs words as tokens and each other symbol as a separate punctuation token. It reads characters from the decoded text using `getchar()` and then outputs tokens using `got_token(...)` after having constructed them using `add_to_token(...)`.

Listing 84: Parser/Lexer Code

```c
while (1) {
  c = getchar();
  if (c == EOF)
    break;
  if (isalpha(c)) {
    do {
      add_to_token(c);
      c = getchar();
    } while (isalpha(c));
    got_token(WORD);
  }
  add_to_token(c);
  got_token(PUNCT);
}
```

Notice that we can't simply stick those simple two methods together. The decoder calls a method to provide a new symbol. Similarly, the lexer calls a method to get a new token. We see two immediate solutions:

- We first decode the entire file and then run the lexer on the complete text. This is undesirable, because we want to process the file in real time. But this approach would only yield tokenized output once the entire file had been processed by the decoder (which might take a while).
- We make one of the two methods callable. This involves rewriting at least one of the two methods - work we don't want to do. Such implementations would use `static` variables. But that limits us to processing one file unless we have multiple instanciations.

Listing 85: Rewritten Compressor

```c
int decompressor(void) {
  static int repchar;
  static int replen;
  if (replen > 0) {
    replen--;
    return repchar;
  }
  c = getchar();
  if (c == EOF)
    return EOF;
  if (c == 0xFF) {
    replen = getchar();
    repchar = getchar();
    replen--;
    return repchar;
  } else
    return c;
}
```

Listing 86: Rewritten Lexer

```c
void lexer(int c) {
  static enum {
    START, IN_WORD
  } state;
  switch (state) {
    case IN_WORD:
    if (isalpha(c)) {
      add_to_token(c);
      return;
    }
    got_token(WORD);
    state = START;
    /* fall through */
    case START:
    add_to_token(c);
    if (isalpha(c))
      state = IN_WORD;
    else
      got_token(PUNCT);
    break;
  }
}
```

What we would like instead is that the parser calls the lexer when it has a new character and then the lexer hands execution back to the parser when it requires new characters. This concept is implemented by continuations.

The decompressor runs until it has a character to emit. It then saves its state (stack, variables, etc.) and calls into the parser. The parser continues where it previously left off. It processes the new character and runs until it needs a new character. Once that is the case, it calls back to the decompressor. This then goes back and forward until the entire file is processed properly.

## 11.4 Implementing Coroutines

This is a very minimal implementation for illustratory purposes. It has to be used like this:

1. Call `co_init();`
2. Create the coroutines, which should switch between each other with `co_new(...);`
3. Switch to the first coroutine with `co_switchto(...);`
4. Once one of the coroutines terminates, we will continue execution here.

```c
typedef void (co_start_fn)(void *);
static struct coroutine *cur_co;
static struct coroutine *main_co;

struct coroutine
{
  void *stack;          // call stack of coroutine
  jmp_buf env;          // the saved context
  co_start_fn *start;   // function to call
  void *arg;            // argument to the function
};

struct coroutine *co_new(co_start_fn *start, void *ctxt)
{
  struct coroutine *co = (struct coroutine *)calloc(1, sizeof(struct coroutine));
  co->stack = calloc(1, CORO_STACK_SIZE + 16);
  co->start = start;
  co->arg = ctxt;

  setjmp(co->env);
  // Those two lines are a machine-depenent hack to set the %rsp and %rip.
  // We do this so that we jump to the desired location and not back here.
  co->env[0].__jmpbuf[6] = ((uint64_t)(co->stack) + CORO_STACK_SIZE);
  co->env[0].__jmpbuf[7] = ((uint64_t)(start_cl));

  return co;
}
```

```c

static void start_cl(void)
{
  (cur_co->start)(cur_co->arg);
  co_switchto(main_co);
  printf("Error:␣returned␣from␣coroutine␣start␣closure.\n");
  exit(-1);
}

extern void co_free(struct coroutine *self);

// switching coroutines
void co_switchto(struct coroutine *next)
{
  if (setjmp(cur_co->env) == 0) {
    cur_co = next;
    longjmp(cur_co->env, 1);
  }
}

void co_init()
{
  main_co = (struct coroutine *)calloc(1, sizeof(struct
      coroutine));
  cur_co = main_co;
  co_switchto(main_co);
}
```

### 11.4.1 Coroutines in Decoder, Lexer Example

```c
static char param;
struct coroutine *lx_co;
struct coroutine *dc_co;

#define DEC_PUTCHAR(c) do { param = c; co_switchto(lx_co); }
    while(0)
void decompress(void *ctxt)
{
  int c;
  int len;
  printf("Started␣decompressor\n");
  while ((c=getchar()) != EOF) {
    if (c == ESCAPE) {
      len = getchar();
      c = getchar();
      while (len--) {
        DEC_PUTCHAR(c);
      }
    } else {
      DEC_PUTCHAR(c);
    }
  }
}

#define LEX_GETCHAR() ( co_switchto(dc_co), param )
void lexer(void *ctxt)
{
  int c;
  while ( (c = LEX_GETCHAR() ) != EOF) {
    if (isalpha(c)) {
      do {
        add_to_token(c);
        c = LEX_GETCHAR();
      } while (isalpha(c));
      got_token(WORD);
    } else {
      add_to_token(c);
      got_token(PUNCT);
    }
  }
}

int main(int argc, char *argv[])
{
  co_init();

  // Create the decompressor and lexer coroutines
  dc_co = co_new( decompress, NULL );
  lx_co = co_new( lexer, NULL );

  // Start by switching to the lexer
  co_switchto( lx_co );

  // When one of them returns or switches back to main, we're done.
  return 0;
}
```

To avoid passing the char with a global variable, one could modify co_switchto() to enable argument passing between coroutines:

```
void *co_switchto(struct coroutine *next, void *arg) {
  if (setjmp(cur_co->env) == 0) {
    cur_co = next;
    cur_co->arg = arg;
    longjmp(cur_co->env, 1);
  }
  return cur_co->arg;
}
```

## 11.5 Adding a Scheduler: ”Not Quite Threads”

Coroutines enable a generalized control flow passing. It is known under many different names: Lightweight threads, Protothreads, Fibers, Cooperative multitasking, Goroutines, ...

So far, however, we are missing:

- true concurrency or parallelism as everything is executed on a single thread
- this is a directed context switch, so we have no scheduling
- no blocking/sleep+wakeup functionality
- no preemption (cannot pause a coroutines unless it hands of execution itself)

To create some approximation of threads, we now add a scheduler. This is used like:

1. Implement your threads as functions `void *threadName(void *arg)`.
2. Call `nathread_init()`.
3. Call `nathread_create(&nathreads[i], threadName, (void *)param)` for each thread with a unique $i$.
4. Call `nathread_schedule()`.

```c
// Maximum number of nathreads we can create
#define MAX_NATHREADS 20

// State of each nathread - extend this for blocking/asleep/etc.
enum status { NAT_FREE = 0, NAT_RUNNABLE, NAT_EXITED };

// The nathread table itself
struct nat {
  struct coroutine *co;
  void *(*start)(void *);
  void *arg;
  enum status state;
} nathreads[MAX_NATHREADS] = {};

// What's currently running?
static nathread_t cur_nathread = NULL;

typedef struct nat *nathread_t;

// create a not-a-thread
int nathread_create(nathread_t *nt, void *(*start)(void *), void
    *arg) {
  // Look for a free nathread in the array
  for(int i=0; i < MAX_NATHREADS; i++) {
    *nt = &nathreads[i];
    if ((*nt)->state == NAT_FREE) {
      // Initialize the nathread
      (*nt)->start = start;
      (*nt)->arg = arg;
      (*nt)->co = co_new(wrapper, NULL);
      (*nt)->state = NAT_RUNNABLE;
      return 0;
    }
  }
  return EAGAIN;
}

static void wrapper(void *arg) {
  nathread_t nt = cur_nathread;
  (nt->start)(nt->arg);
  nathread_exit(nt->co->arg);
}

// terminate this thread with a return value
void nathread_exit(void *retval) {
  cur_nathread->state = NAT_EXITED;
  nathread_yield();
}

// switch to another nathread (not preemtion!)
void nathread_yield(void) {
  co_switchto( main_co, NULL );
}

// initialize the package
void nathread_init(void) {
  co_init();
}

// scheduler, run until nothing is runnable any more
void nathread_schedule(void) {
  int t = 0;
  do {
    // Look for a runnable nathread
    cur_nathread = NULL;
    for(int i=0; i < MAX_NATHREADS; i++) {
      t = (t + 1) % MAX_NATHREADS;
      if ( nathreads[t].state == NAT_RUNNABLE) {
        // Switch to this runnable nathread.
        cur_nathread = &nathreads[t];
        co_switchto( cur_nathread->co, NULL );
        break;
      }
    }
  } while (cur_nathread != NULL);
}
```

```
#include <stdio.h>
#include "nt.h"

/*
* Very simple not-a-thread demo: just count to 3!
*/

static void *my_thread(void *arg)
{
  long index = (long)arg;
  for(int count=0; count < 3; count++) {
    printf("Not-a-thread␣%ld:␣count␣%d\n", index, count);
    nathread_yield();
  }
  return NULL;
}

#define NUM_NATHREADS 3

int main(int argc, char *argv[])
{
  nathread_t nathreads[ NUM_NATHREADS ];

  // Create our not-a-threads
  nathread_init();

  for( int i=0; i<NUM_NATHREADS; i++) {
    nathread_create( &nathreads[i], my_thread, (void *)(long)i )
        ;
  }

  // Enter the scheduler
  nathread_schedule();

  // Back here when nothing left to run
  return 0;
}
```

### 11.5.1 What is still Missing for Threads?

Although this is a pretty complete setup, it is still missing a few key aspects to be considered to implement the concept of threads.

- Join: Waiting for a `nathread` to terminate and get its return value.
- Blocking/sleep and wakeup: We would need to add a new sate `NAT_WAITING`. A signal would then handle to change the state back to `NAT_RUNNABLE`.
- Locks, condition variables: This is a trivial extension to sleep/wakeup.

- Parallelism (multiple cores): This is a bit tricky. We would need to synchronize the scheduler queue. For instance with a spinlock or atomics? Also, we would have to make changes to sleep/wakeup and store the state of thread per-core. For each core we would need to know the current thread.
- Preemption: So far, each coroutines/not-a-thread has to call `yield(...)`, which is called cooperative multitasking. Instead, we want a mechanism to asynchronously switch coroutines. That may be done with processor exception, which we will consider later.

All in all, the details of everything discussed in this section of unorthodox control flow proably isn' very relevant. Well, `setjmp` and `longjmp` should be understand. But the rest probably only is to illustrate what those (rather simple but still complex) functions enable.

# 12 Linking

We have already learned about the C toolchain/workflow. When we typically just say compiling, in reality multiple steps happen: preprocessing, compilation, assembling, and linking. We have already discussed preprocessing (# instructions/macros) and compilation of C functions and data structures. Assembling simply corresponds to converting the assembly into its binary equivalent. After assembling, we have an .o object file, but that is not executable yet. We will look at this .o file in depth here and what we have to do to make it executable.

At its core, (static) linking combines different compiled compilation units into one executable. This has to be done, because the different compilation units may depend on each other. For instance, one specifies an interface, which the other library knows exists (as it includes the header files), but the actual implementation is somewhere else.

Consider this example: We have two .c files, where `main.c` references a function in `swap.c` . Pre-processing, compilation, and assembling happens independently for each .c file/compilation unit. Thus, the assembled .o file of `main.c` can not contain a proper reference for the call to `swap()` as the compiler did not have any information about its location in memory.

Thus, we call those .o files relocatable object files. Only by using a linker, we can resolve those references and create an executable object file, which contains all necessary code and references.

Dynamic linking is still different. But the intuition for that is better built after static linking has been fully understood.

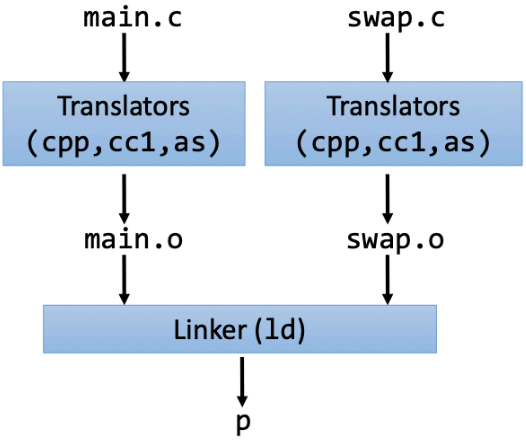


Figure 25: Static Linking Conceptually

Linking has many practical advantages:

- Without linking, we would be required to have everything in one compilation unit from the beginning.
- Separating compilation like this enables us to build libraries of common functionality which must not be built every time but only included through linking. But even without libraries, this increases efficiency, because when changing the content of one file, we only need to recompile that file and relink instead of recompiling the entire source code.
- Linking saves a lot of space. This will become apparent when we understand dynamic linking later on. We only include code for the functions which are actually unique to our program.

Notice that each .c file is considered separately for compilation. This means that we don't have any cross-file compilation optimizations. We accept this because we can rarely increase the performance by compiling across multiple files.

However, one thing that is sometimes done is called inlining. Inlining a function means to insert its code directly into the caller to avoid the calling overhead. But there are engineering concerns: Changes to the source but not modify the inlined function. This is prevented by including the inlined code in a header file which we include so that recompilation automatically happens. Generally, inlining is only helpful in certain cases (usually often called from one position such as loops), where the inlined function is small.

## 12.1 Linking in 2 Steps

### 12.1.1 Symbol Resolution

Every program defines and references symbols, which correspond to variables and functions.

Examples:

- `void swap(){...}` defines the symbol `swap`
- `swap();` references the symbol `swap`
- `int *xp = &x;` defines the symbol `xp` and references the symbol `x`

The compiler worked with the user-defined names for variables and functions but the assembler then works out the different symbols and puts all symbols into a symbol table. There is one entry for each symbol that is declared and/or referenced. The symbol table is an array of structs. Each entry/struct contains:

- symbol name
- symbol type
- symbol size
- symbol location (if known)
- (meta-information whether the symbol is static/its visibility etc.)

So this symbol table is created during assembling, i.e., even before the linker is involved. During its first step, the linker then looks at the symbol tables of multiple relocatable object files and resolves dependencies. It associates each symbol (reference) with exactly one symbol definition.

### 12.1.2 Relocation

After having figured out one definition for each referenced symbol, i.e., one large symbol table for each definition, relocation corresponds to merging the separate code and data sections of the different relocatable object file into a single executable file.

This also involves using the definitions figured out during symbol resolution to relocate symbols from their relative locations in the relocatable .o file to their final absolute memory locations in the executables.

Finally, all references to any symbols are updated according to their now defined position in the executable file. The memory layout chosen by the linker for this executable file has an impact on performance. Different layouts may result in different cache performance/misses.

## 12.2 Object Files

There are three types of object files.

- relocatable object file (.o file)

  Each .o file is created from exactly one .c file/compilation unit through compilation and assembling. It contains code and data in a form that can be combined with other relocatable object files.

- executable object file

  This contains code and data in a form that can be copied directly into memory and then executed. This requires that linking has been performed on the .o/relocatable object file. Also, executing an executable object file may required dynamic linking.

- sharable object file (.so file, DDLs on Windows)

  This is a special type of relocatable object files that can be loaded into memory and linked dynamically, at either load time or run-time. It is not an executable itself but a compiled library instead.

All those different types have had different formats depending on the specific machine. Today, we use ELF (Executable and Linkable Format) as the standard binary format for object files. This format was originally proposed by AT&T System V Unix and later adopted by basically everyone else.

| |
|---|
| ELF header |
| Segment header table<br>(required for executables) |
| . text section |
| . rodata section |
| . data section |
| .bss section |
| . symtab section |
| .rel.txt section |
| .rel.data section |
| . debug section |
| Section header table |

Figure 26: Basic Layout of ELF Binary

- ELF header

  word size, byte ordering (big/little endianess), file type (‘.o‘, ‘.so‘, ‘exec‘), machine type, ...

- segment header table

  page size, virtual address memory segments (sections), segment sizes

- ‘.text‘ sectoin

  code

- ‘.rodata‘ section

  read only data: jump tables, literals, ...

- ‘.data‘ section

  initialized global variables

- ‘.bss‘ section

  uninitialized global variables

  this has a section header but occupies not space in the object file

  still, memory addresses are already designated to the entries of this section

- ‘.symtab‘ section

  this is the symbol table

  it stores the names, locations, etc. of procedures and (static) variables

- ‘.rel.text‘ section

  relocation information for the ‘.text‘ section

  specifically:

  - addresses of instructions that will need to be modified in the executable

  - instructions for modifying those instructions

- ‘.rel.data‘ section

  relocation information for the ‘.data‘ section

  specifically:

  addresses of pointer data that will need to be modified in merged executable

- ‘.debug‘ section

  info for symbolic debugging

  enables getting the line number of the original ‘.c‘ file when sth. goes wrong for example

- section header table

  offsets and sizes of each section

  this comes last, because it was historically easies to compute this table in the end and append it then

## 12.3 Linker Symbols

The linker distinguishes between three types of linker symbols.

- global symbols: symbols defined by module `m` that can be referenced by other modules (e.g., ‘non-static‘ C functions and ‘non-static‘ global variables)

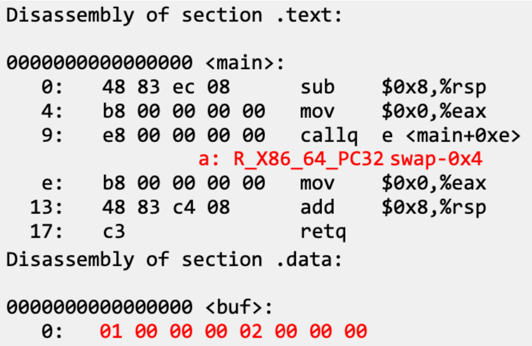
- external symbols: global symbols that are referenced by module `m` but defined by some other module

- local symbols: symbols that are defined and referenced exclusively by module `m` such as `static` functions and attributes. (local linker symbols are NOT local program variables!)

## 12.4 Example

Listing 87: main.c.

```c
int buf[2] = {1,2};

int main() {
  swap();
  return 0;
}
```

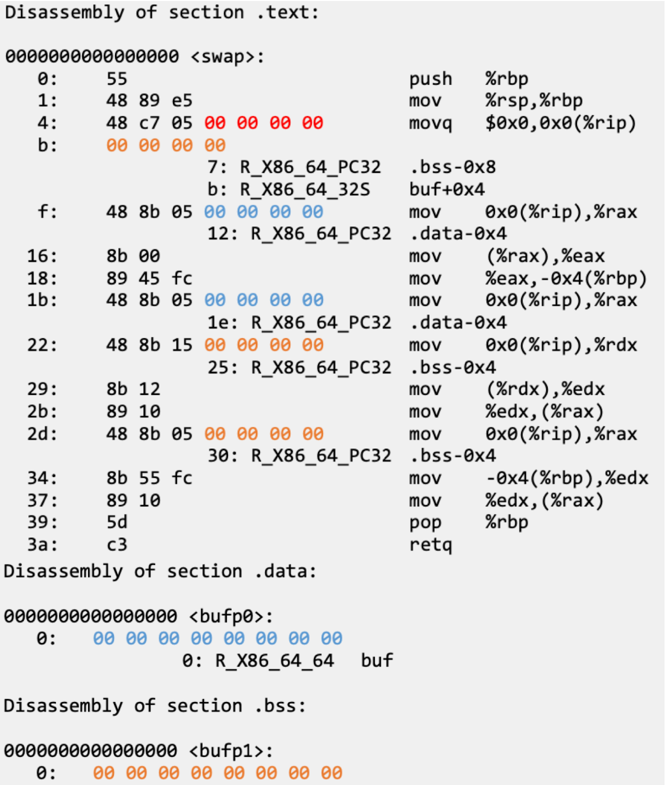


```
Disassembly of section .text:

0000000000000000 <main>:
   0:   48 83 ec 08       sub    $0x8,%rsp
   4:   b8 00 00 00 00    mov    $0x0,%eax
   9:   e8 00 00 00 00    callq  e <main+0xe>
                  a: R_X86_64_PC32 swap-0x4
   e:   b8 00 00 00 00    mov    $0x0,%eax
  13:   48 83 c4 08       add    $0x8,%rsp
  17:   c3                retq
Disassembly of section .data:

0000000000000000 <buf>:
   0:   01 00 00 00 02 00 00 00
```

Figure 27: Disassembly of main.c

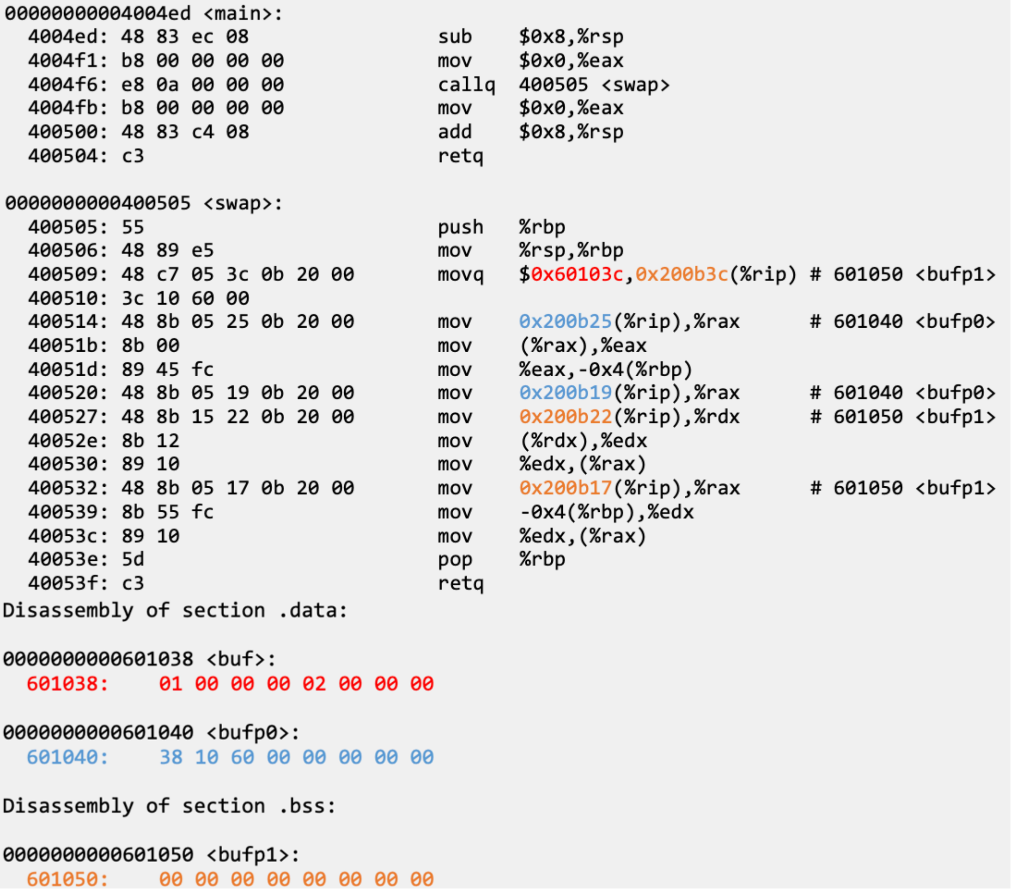
Listing 88: swap.c

```c
extern int buf[];

static int *bufp0 = &buf[0];
static int *bufp1;

void swap() {
  int temp;
  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```



```
Disassembly of section .text:

0000000000000000 <swap>:
   0:    55                            push   %rbp
   1:    48 89 e5                      mov    %rsp,%rbp
   4:    48 c7 05 00 00 00 00          movq   $0x0,0x0(%rip)
   b:    00 00 00 00
                  7: R_X86_64_PC32    .bss-0x8
                  b: R_X86_64_32S     buf+0x4
   f:    48 8b 05 00 00 00 00          mov    0x0(%rip),%rax
                  12: R_X86_64_PC32  .data-0x4
  16:    8b 00                         mov    (%rax),%eax
  18:    89 45 fc                      mov    %eax,-0x4(%rbp)
  1b:    48 8b 05 00 00 00 00          mov    0x0(%rip),%rax
                  1e: R_X86_64_PC32  .data-0x4
  22:    48 8b 15 00 00 00 00          mov    0x0(%rip),%rdx
                  25: R_X86_64_PC32  .bss-0x4
  29:    8b 12                         mov    (%rdx),%edx
  2b:    89 10                         mov    %edx,(%rax)
  2d:    48 8b 05 00 00 00 00          mov    0x0(%rip),%rax
                  30: R_X86_64_PC32  .bss-0x4
  34:    8b 55 fc                      mov    -0x4(%rbp),%edx
  37:    89 10                         mov    %edx,(%rax)
  39:    5d                            pop    %rbp
  3a:    c3                            retq
Disassembly of section .data:

0000000000000000 <bufp0>:
   0:   00 00 00 00 00 00 00 00
                0: R_X86_64_64   buf

Disassembly of section .bss:

0000000000000000 <bufp1>:
   0:   00 00 00 00 00 00 00 00
```

Figure 28: Disassembly of swap.c



```
00000000004004ed <main>:
  4004ed: 48 83 ec 08                    sub    $0x8,%rsp
  4004f1: b8 00 00 00 00                 mov    $0x0,%eax
  4004f6: e8 0a 00 00 00                 callq  400505 <swap>
  4004fb: b8 00 00 00 00                 mov    $0x0,%eax
  400500: 48 83 c4 08                    add    $0x8,%rsp
  400504: c3                             retq

0000000000400505 <swap>:
  400505: 55                             push   %rbp
  400506: 48 89 e5                       mov    %rsp,%rbp
  400509: 48 c7 05 3c 0b 20 00           movq   $0x60103c,0x200b3c(%rip) # 601050 <bufp1>
  400510: 3c 10 60 00
  400514: 48 8b 05 25 0b 20 00           mov    0x200b25(%rip),%rax      # 601040 <bufp0>
  40051b: 8b 00                          mov    (%rax),%eax
  40051d: 89 45 fc                       mov    %eax,-0x4(%rbp)
  400520: 48 8b 05 19 0b 20 00           mov    0x200b19(%rip),%rax      # 601040 <bufp0>
  400527: 48 8b 15 22 0b 20 00           mov    0x200b22(%rip),%rdx      # 601050 <bufp1>
  40052e: 8b 12                          mov    (%rdx),%edx
  400530: 89 10                          mov    %edx,(%rax)
  400532: 48 8b 05 17 0b 20 00           mov    0x200b17(%rip),%rax      # 601050 <bufp1>
  400539: 8b 55 fc                       mov    -0x4(%rbp),%edx
  40053c: 89 10                          mov    %edx,(%rax)
  40053e: 5d                             pop    %rbp
  40053f: c3                             retq
Disassembly of section .data:

0000000000601038 <buf>:
  601038:     01 00 00 00 02 00 00 00

0000000000601040 <bufp0>:
  601040:     38 10 60 00 00 00 00 00

Disassembly of section .bss:

0000000000601050 <bufp1>:
  601050:     00 00 00 00 00 00 00 00
```

Figure 29: Disassembly of Relocated File

## 12.5 Strong and Weak Symbols

When considering symbols, the linker behaves differently depending of whether symbols are considered strong or weak. By default, strong symbols are procedures/functions and initialized globals. Weak symbols are uninitialized globals if we set `-fcommon` and strong otherwise.

`-fcommon` has been the default even a few years ago. But today, the default is `-fno-common`, which means that also uninitialized globals are strong symbols.

Then linking, the linker proceeds as follows with those two symbol types:

1. Multiple strong symbols are NOT allowed (each item can be defined only once).
2. Given a strong symbol and multiple weak symbols, the strong symbol is chosen as the definition and references to the weak symbols resolve to the strong symbol.
3. If there are (only) multiple weak symbols, the linker picks an arbitrary one.

One challenge is that the symbols might not even refer to the same type so that writing to the same symbol may lead to writing to 64 bytes (if considered double) even if defined as 32 bit int. This leads to unintended behavior without a guaranteed compiler/linker warning.

Variables/symboles declared with `extern` always rely on a definition in another file. If such a definition is not provided, we get an error. Usually, `extern` symbols have a strong version somewhere.

We rarely make a symbol explicitly weak. By by default everything is strong. However, we can use a pragma to declare a symbol as weak: `__attribute__((weak));`.

Avoid ambiguities and difficulties by never letting such cases arise. Use `static` if possible and either initialize a global variable or use `extern`.

## 12.6 Static Libraries

Now we consider how to package and include functions commonly used by programmers such as math, I/O, memory management (malloc, ...), string manipulation, ... With the system so far, we have two options: Either put all functions into a single source file which is space and time inefficient (big objects file linked of which only a fraction is needed) or put each function in a separate source file which is efficient but burdensome on the programmer (must explicitly link/include all files).

The solution to this are static libraries. Those are .a archive files, which concatenate related relocatable object files into a single file with an index (called an archive). An enhanced linker now tries to resolve all references by considering the provided files. But if there are still unresolved references in the end, the linker looks for those in specified (or default) libraries. If an archive member file resolves a reference, it links that into the file.

An archive can be created from .o files using an archiver (`ar` on linux). Notice that each function from such as `printf` etc. must still be given through its own .c file so that it gets its own .o file. The archiver allows incremental updates and recompiles functions that changed and replaces their .o file in the archive.

Commonly-used libraries are `libc.a` (the C standard library), which as over 900 object file in 8 MB memory for I/O, memory allocation, signal handling, string allocation, data and time, random numbers, integer math, ... Also, `libm.a` (the C math library) is usually a default library. It has 226 object files and is 1 MB in size. It provides floating point math functionality such as sin, cos, tan, log, exp, sqrt, ...

When resolving dependencies, the linker proceeds like this:

1. Scan .o files and .a files in the command line order.

2. During the scan, keep a list of the current unresolved references.

3. As each new .o or .a file, is encountered, try to resolve each unresolved reference in the list against the symbols defined in that file.

4. If any entries in the unresolved list at end of scan, then error.

This means that the command line order matters. The default libraries should always come later if we want to overwrite them with custom implementations. In practice this means that libraries should be put at the end of the command line. In modern linkers, we may also specify batches for the processing order.

## 12.7 Shared Libraries

Static libraries have some disadvantages:

- duplication in the stored executables

- duplication in the running executables

- (minor) bug fixes of system libraries require each application to explicitly relink

Shared libraries address those issues. We consider executable object files, which have not all symbols resolved. Instead, at load-/run-time, specific object files (.so on linux, dynamic link libraries (DLLs) in windows) are linked into an application dynamically.

Dynamic linking of shared libraries can happen in two ways:

- load-time linking

  This is the common case for Linux and handled automatically by the dynamic linker (`ld-linux.so`). The standard C library (`libc.so`) is dynamically linked, for instance.

- run-time linking

  In Unix, this is done by calls to the `dlopen()` interface

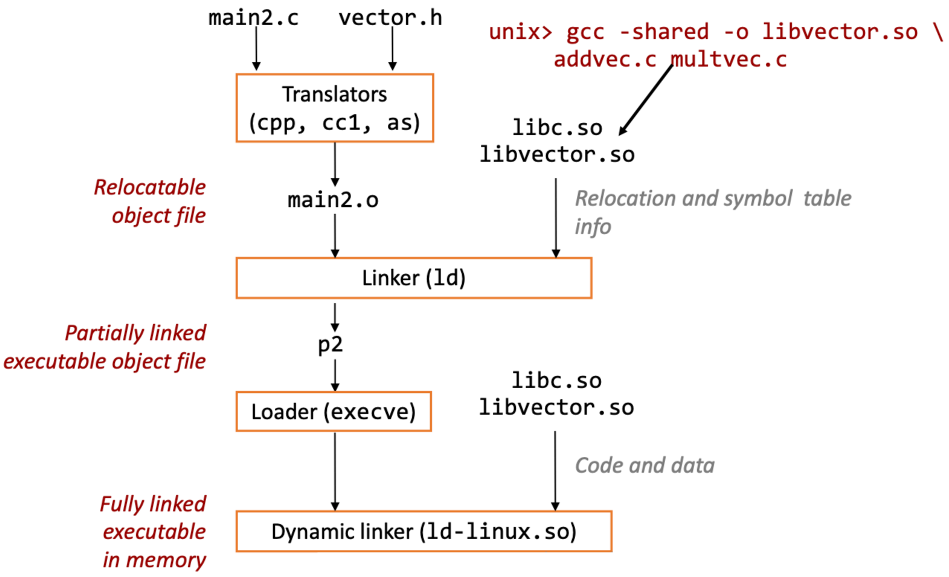


Figure 30: Dynamic Linking at Load Time

Listing 89: Dynamic Linkin gat Run Time

```c
#include <stdio.h>
#include <dlfcn.h>
int x[2] = {1,2};
int y[2] = {3,4};
int z[2];

int main(int argc, char *argv[]) {
  void *handle;
  void (*addvec)(int *, int *, int *, int);
  char *error;

  /* dynamically load the shared lib that contains addvec() */
  handle = dlopen("./libvector.so", RTLD_LAZY);
  if (!handle) {
    fprintf(stderr, "%s\n", dlerror());
    exit(1);
  }
  /* get a pointer to the addvec() function we just loaded */
  addvec = dlsym(handle, "addvec");
  if ((error = dlerror()) != NULL ){
    fprintf(stderr, "%s\n", error);
    exit(1);
  }

  /* Now we can call addvec() it just like any other function */
  addvec(x,y,z,2);
  printf("z␣=␣[%d␣%d]␣\n", z[0], z[1]);

  /* unload the shared library */
  if (dlclose(handle) < 0 ) {
    fprintf(stderr, "%s\n", dlerror());
    exit(1);
  }
  return 0;
}
```

## 12.8 Library Inter-Positioning

One technique is called library inter-positioning. It allows programmers to intercept calls to arbitrary functions by providing that function themselves and then linking the library after some processing is done beforehand. Inter-positioning can occur at compile-time, link-time or load-/run-time.

We use inter-positioning for several reasons:

- security: confinement (sandboxing) and behind the scenes encryption
- monitoring and profiling: count number of calls to functions, characterize call sizes and arguments to functions, malloc tracing

Listing 90: Intercepting `malloc` to Count Calls

```
void *malloc(size_t size){
  static void *(*fp)(size_t) = 0;
  void *mp;
  char *errorstr;

  /* Get a pointer to the real malloc() */
  if (!fp) {
    fp = dlsym(RTLD_NEXT, "malloc");
    if ((errorstr = dlerror()) != NULL) {
      fprintf(stderr, "%s():␣%s\n", fname, errorstr);
      exit(1);
    }
  }

  /* Call the real malloc function */
  mp = fp(size);

  mem_used += size;

  return mp;
}
```

# 13 Code Vulnerabilities

## 13.1 Worms and Viruses

First, we can distinguish between worms and viruses. A worm is a program than can run by itself and can propagate a fully working version fo itself to other computers. A Virus is a program that adds itself to other programs and cannot run independently. Both types can cause wide destruction of data and computer networks.

The term worm was coined by early cyberpunk/sci-fi novels. And initial worms weren't necessarily malicious. They were used to gather status data or meta-information.

Probably the first internet worm was designed by Robert Morris in 1988. While convicted, he is now a distinguished professor at MIT and had no harmful intentions. His goal was to count the number of hosts connected to the internet. The following subsections are just some of the vulnerabilities his worm exploited.

## 13.2 Stack Overflow Bugs

Stack Overflow Bugs as considered here are a special type of overflow bugs. Those must not always occur on the stack. If they occur on the heap, we call them heap overflow bugs, for instance.

The `gets()` function is a vulnerable function.

Listing 91: Vulnerable `gets()` Function

```c
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
    *p++ = c;
    c = getchar();
    }
  *p = '\0';
  return dest;
}
```

This does not limit the number of characters read and, hence, may write read characters beyond the allocated memory section provided with the pointer ‘char *dest‘. If this pointer is located on the stack, we can overwrite other's stack frames and also the return address if we are a function ourselves and, thus, redirect to malicious code. Other Unix functions such as `strcpy`, `scanf`, `fscanf`, and `sscanf` (when using `%s`) are analogously vulnerable.

Listing 92: Example Misuse of `gets`

```c
int main() {
  printf("Type␣a␣string:");
  echo();
  return 0;
}

void echo() {
    char buf[4];   /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Listing 93: Compiled Version

```
00000000004005bd <echo>:
  sub    $0x18,%rsp
  mov    %rsp,%rdi
  callq  4004c0 <gets@plt>
  mov    %rsp,%rdi
  callq  400480 <puts@plt>
  add    $0x18,%rsp
  retq

00000000004005d6 <main>:
  sub    $0x8,%rsp
  mov    $0x400684,%edi
  mov    $0x0,%eax
  callq  400490 <printf@plt>
  mov    $0x0,%eax
  callq  4005bd <echo>
  mov    $0x0,%eax
  add    $0x8,%rsp
  retq
```

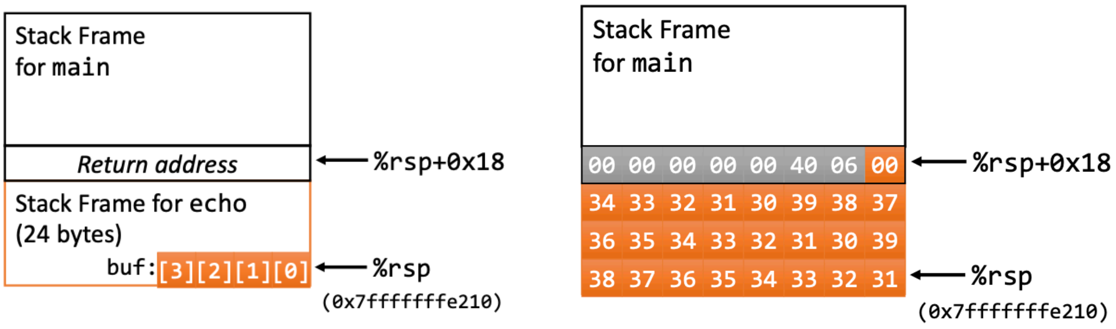


Figure 31: Buffer Overflow Stack & Memory Layout

With `123` as input everything is fine, as with `1234567` as input, although we already write beyond the bounds of the buffer `buf`. That is as the compiler inserted some padding in memory. When we input `12345678901234567801234`, we get a segmentation fault.

Nevertheless, the compiler wants to help us and may detect some of those vulnerabilities and print warning or may refuse to compile. We use the `gcc` flags as in this command `gcc -D_FORTIFY_SOURCE=0 -fno-stack-protector -O -S bufdemo` to turn of those protections.

A more sophisticated approach may first write some exploit code, then some padding, and use the return address to jump to the just written exploit code. Notice that this only works if security mechanism, which prevent from executing instructions from the stack, are not active.

Furthermore, one usually wants to know the source code to plan the memory layout, the endianness of the target machine, as well as its architecture. Otherwise, this attack is a shot in the dark and one has to do much try and error.

Generally, such buffer overflow bugs allow remote machines to execute arbitrary code on victim machines!

But because we only discussed stack overflow bugs, this doesn't mean that heap overflow bugs aren't fatal! They can mess with (segregated) linked list used by malloc to redirect the execution.

## 13.3 Buffer Overflow Bugs

Here, we consider the `SUN XDR` library as an example to see that overflows on the heap and not only on the stack are destructive. While we don't store return addresses on the heap, we can mess with the dynamic memory allocators meta-information and change the (segregated) linked-list used for memory management.

More specifically: There are function pointers on the heap. We set free bits and prev/next pointers in a memory chunk to hijack control flow, when the memory allocator coalesces 'free' blocks

Listing 94: Vulnerable Function of the XDR Library

```c
void* copy_elements(void *ele_src[], int ele_cnt, size_t
    ele_size) {
  void *result = malloc(ele_cnt * ele_size);
  if (result == NULL)
    /* malloc failed */
    return NULL;
  void *next = result;
  int i;
  for (i = 0; i < ele_cnt; i++) {
    /* Copy object i to destination */
    memcpy(next, ele_src[i], ele_size);
    /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
```

The issue arises from the computation `ele_cnt * ele_size` in `malloc`. On a 32-bit machine, we can set $\mathit{ele\_cnt} = 2^{20} + 1$ and $\mathit{ele\_size} = 4096 = 2^{12}$ so that the multiplication is just `ele_size`. Thus, way to little memory is allocated and we can write beyond the allocated memory location when copying the data continuously.

## 13.4 Stopping Overrung Bugs

### 13.4.1 Avoiding Overflow Vulnerability

Instead of `gets()` or similar functions, we use library routines that limit the string lengths. Use `fgets` instead of `gets` Use `strncpy` instead of `strcpy` Use `%ns`, where $n$ is a suitable integer in `scanf` or similar.

### 13.4.2 System-Level Protections

- The compiler can insert checks on functions.

  It may add some arbitrary values in code and check during runtime whether they have been modified, i.e., whether a buffer overflow may have occurred.

- Randomized stack offsets.

  During execution, a random offset can be added to the stack. Then, it is more difficult for attackers to predict memory locations of different segments and successfully redirect control flow. This is done by default in moern machines.

- Nonexecutable code segments

  Besides flags declaring sections as ’read-only’ or ’writeable’, we may also change the permission ’execute’ so that the CPU just won’t execute code, which is located on the stack, so that no malicious code can be injected.

## 13.5 Attack Method: Return Oriented Programming

This is an approach of attackers to deal with limitations on executing dynamically inserted code on the stack. Instead, the addresses are overwritten so that the control flow jumps to and executes cherry-picked machine instruction sequences, which are already present in the application process memory. Those ’useful’ instruction sequences are called ’gadgets’.

# 14 Floating Point Numbers

## 14.1 Representing Floating-Point Numbers

### 14.1.1 Fractional Binary Numbers

We start by considering fractional binary numbers in general. They work very similarly to fractional decimal numbers, just using 2 instead of 10 as the base. Bits to the right of the binary point represent fractional powers of 2. Thus: Division by 2 corresponds to shifting right. Multiplication with 2 corresponds to shifting left. And there are fractions, which cannot be exactly represented in this notation. Only numbers of the form $\frac{x}{2^k}$ can be exactly represented. Numbers which can not precisely expressed are for example:

- $\frac{1}{3}$ is $0.0101010101[01]\ldots_2$
- $\frac{1}{5}$ is $0.001100110011[0011]\ldots_2$
- $\frac{1}{10}$ is $0.0001100110011[0011]\ldots_2$

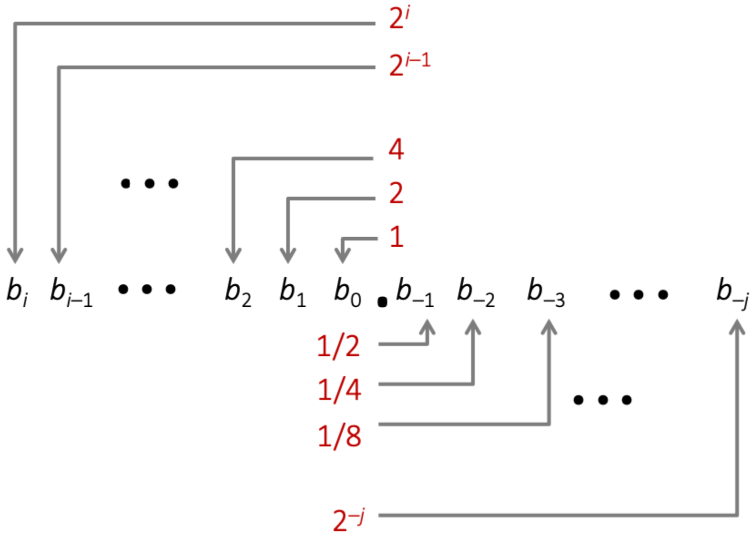

$2^i$
$2^{i-1}$
4
2
1
$b_i$ $b_{i-1}$ • • • $b_2$ $b_1$ $b_0$ . $b_{-1}$ $b_{-2}$ $b_{-3}$ • • • $b_{-j}$
1/2
1/4
1/8
$2^{-j}$

Figure 32: Fractional Binary Numbers

$$\sum_{k=-j}^{i} b_k \cdot 2^k$$

### 14.1.2 IEEE Floating Point

This is a standard that was established in 1985 to have uniform floating point arithmetics. Before, many custom formats existed. Luckily, all major manufacturers were part of the commission so that the standard is adopted by all major CPUs. The definition was driven by numerical concerns. Numerical scientists wanted a nice standard for rounding, overflow, underflow, ... That came at the tradeoff of being rather difficult to implement in hardware.

The most general form of floating points looks numerically like

$$(-1)^s.M.2^E$$

where $s$, $M$, and $E$ are the parameters to define a number.

- $s$: sign
- $M$ (significant): fractional value, normally in the range $[1.0, 2.0)$ (normalized but in special case also $[0.0, 1.0)$).
- $E$: exponent, weights value by a power of two

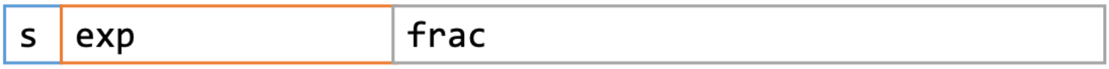


Figure 33: IEEE Floating Point Encoding

- the MSB is the sign bit $s$
- `exp` encodes $E$ but is not $E$
- `frac` encodes $M$ but is not $M$

The details of the encoding scheme will be part of the next subsection.

## 14.2 Types of IEEE Floating-Point Numbers

If we designate different amounts of bits to ‘exp‘ and ‘frac‘, we get different precision for the representable numbers. Thus, the standard defines two original precisions: IEEE 754 Single Precision (32 bits) and IEEE 754 Double Precision (64 bits).

| **precision** | **significant bits** | **exponent bits** | **total** |
|---|---|---|---|
| half | 11 | 5 | 16 |
| single | 24 | 8 | 32 |
| double | 53 | 11 | 64 |
| quadruple | 113 | 15 | 128 |
| octuple | 237 | 19 | 256 |
| Google bfloat16 | 7 | 8 | 16 |
| Nvidia TensorFloat | 10 | 8 | 19 |
| AMD fp24 | 17 | 7 | 24 |

Figure 34: Floating Point Precisions (sign bit part of significant bits)

In C99, single precision is guaranteed with `float` and double precision is guaranteed with `double`. `long double` etc. usually correspond to what we expect such as quadruple precision but it is implementation defined and not guaranteed by the standard.

Furthermore, we have seen conversion of ‘int‘, ‘char‘, ... as just reinterpreting the bit representation. Here, we instead actually change the bit representation when casting/converting.

- `double`/`float`→`int`

  This truncates the fractional part (like rounding toward zero). When the result is out of range or NaN, this is implementation defined but generally `TMin`.

- `int`→`double`

  This is an exact conversion as an `int` has $\leq 53$ bits

- `int`→`float`

  This will round according to rounding mode. We will see (default) rounding later.

### 14.2.1 Normalized Values

This is the case if `exp`≠`000...0` and `exp`≠`111...1`.

- exponent

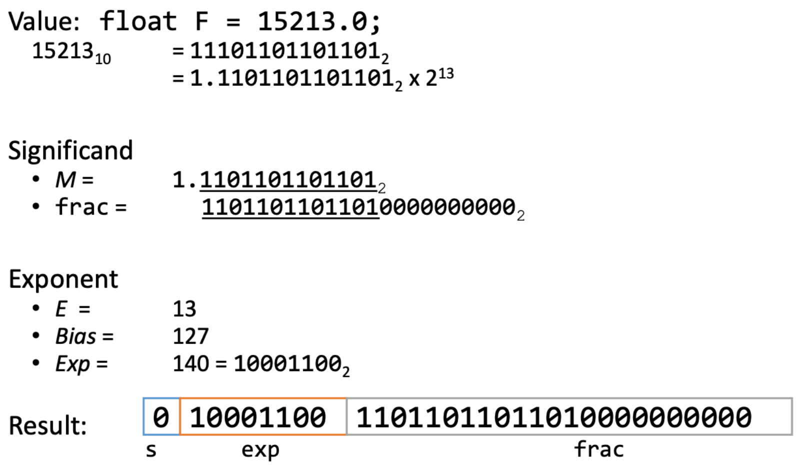
  This is encoded as a biased value: `E = exp - bias`, where `exp` is the unsigned value in hardware. `bias`= $2^{e-1} - 1$, where $e$ is the number of exponent bits. This leads to roughly half of the exponents being above and half below zero.

  single precision: 127 (-126...127)

  double precision: 1023 (-1022...1023)

- significant

  Because values of normalized values are in $[1.0, 2.0)$, we have `1.xxx...x`. Thus, the leading 1 is implied and mustn't be stored.



Value: float F = 15213.0;

$15213_{10}$ = $11101101101101_2$

= $1.1101101101101_2 \times 2^{13}$

Significand

- *M* = $1.\underline{1101101101101}_2$
- frac = $\underline{1101101101101}0000000000_2$

Exponent

- *E* = 13
- *Bias* = 127
- *Exp* = 140 = $10001100_2$

Result: 0 | 10001100 | 11011011011010000000000

Figure 35: Normalized Encoding Example

### 14.2.2 Denormalized Values

This is the case if `exp`=`000...0`.

- exponent

  The `bias` is computed as for normalized values. Then, the exponent is `E = 1-bias`. So we add 1 to get the same exponent as in the case where the exponent is 1.

- significant

  With denormalized values, we have values in `[0.0,1.0)`. So, we implicitly encode with a leading `0`.

Thus, with `frac = 000...0` we encode zero. (One, for each sign.)

### 14.2.3 Special Values

This is the case if `exp = 111...1`.

- `frac = 000...0`

  This represents $\infty$ and is assigned of an operation overflows. Notice that we still have the sign bit and, thus, have positive and negative infinity.

- `frac`$\neq$`000...0`

  This corresponds to 'NaN' (Not-a-Number). This is assigned whenever a numeric value can't be assigned, such as `sqrt(-1)`, $\infty - \infty$, $\infty * 0$.

### 14.2.4 Interesting Numbers & Properties

| description | exp | frac | numerical value |
|---|---|---|---|
| Zero | $00\ldots00$ | $00\ldots00$ | $0.0$ |
| Smallest pos. denorm.<br>- Single $\approx 1.4 \times 10^{-45}$<br>- Double $\approx 4.9 \times 10^{-324}$ | $00\ldots00$ | $00\ldots01$ | $2^{-\{23,52\}} \times 2^{-\{126,1022\}}$ |
| Largest denormalized<br>- Single $\approx 1.18 \times 10^{-38}$<br>- Double $\approx 2.2 \times 10^{-308}$ | $00\ldots00$ | $11\ldots11$ | $(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$ |
| Smallest pos. normalized<br>- Just larger than largest denormalized | $00\ldots01$ | $00\ldots00$ | $2^{-\{126,1022\}}$ |
| One | $01\ldots11$ | $00\ldots00$ | $1.0$ |
| Largest normalized<br>- Single $\approx 3.4 \times 10^{38}$<br>- Double $\approx 1.8 \times 10^{308}$ | $11\ldots10$ | 11...11 | $(2.0 - \varepsilon) \times 2^{\{127,1023\}}$ |

Figure 36: Interesting Floating Point Numbers

We can almost use unsigned integer comparison to compare floating point numbers. We must only consider to first compare the sign bit, the special values NaN and $\infty$ and $-\infty$ as well as that $-0 = 0$.

## 14.3 Floating-Point Spacing

Spacing decreases with distance for normliazed values so that the relative errors stays roughly the same. But there are always blocks according to the size of `frac` which have the same spacing. The denormalized values have the same spacing as the smallest normalized values to fill the entire gap to 0.

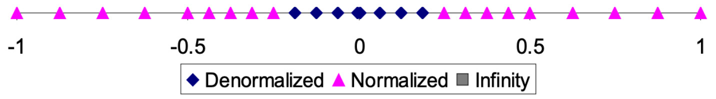


Figure 37: Floating Point Spacing

## 14.4 Floating-Point Rounding

The basic idea is to define floating point operations is that we would first compute the exact result and then make it fit into the desired precision. If the result is too large, we may get an overflow. And we likely have to round to fit the `frac` into the given number of bits.

So that we can deal with operations, let's understand rounding first. We have four rounding modes of which nearest even is the default one.

| | CHF 1.40 | CHF 1.60 | CHF 1.50 | CHF 2.50 | CHF -1.50 |
|---|---|---|---|---|---|
| Towards zero | 1 | 1 | 1 | 2 | -1 |
| Round down $(-\infty)$ | 1 | 1 | 1 | 2 | -2 |
| Round up $(+\infty)$ | 2 | 2 | 2 | 3 | -1 |
| Nearest Even (default) | 1 | 2 | 2 | 2 | -2 |

Figure 38: Rounding Modes Illustration

Nearest Even/"round-to-even" is the default rounding mode for IEEE FP. It means we do 'normal' rounding to the closest number unless we're EXACTLY in between two. Then, we round to the even one. Compared to the other modes, this is the only which is statistically unbiased. All others consistently over-/under-estimate.

| Value | Binary | Rounded | Action | Result |
|---|---|---|---|---|
| $23/32$ | $10.00011_2$ | $10.00_2$ | $< 1/2$ : down | 2 |
| $23/16$ | $10.00110_2$ | $10.01_2$ | $> 1/2$ : up | $21/4$ |
| $27/8$ | $10.11100_2$ | $11.00_2$ | $= 1/2$ : up | 3 |
| $25/8$ | $10.10100_2$ | $10.10_2$ | $= 1/2$ : down | $21/2$ |

Figure 39: Rounding Examples

To consider rounding further, we assume that we are given a real number and want to create a floating pointer from it. We assume that `exp` has 4 bits and that `frac` has 3 bits.

### 14.4.1 Normalize to Have Leading 1

This will introduce the exponent. Specifically, we start with exponent 0 and decrement the exponent as we shift left/increment as we shift right.

| Value | Binary | Fraction | Exponent |
|---|---|---|---|
| 128 | 10000000 | 1.0000000 | 7 |
| 13 | 00001101 | 1.1010000 | 3 |
| 17 | 00010001 | 1.0001000 | 4 |
| 19 | 00010011 | 1.0011000 | 4 |
| 138 | 10001010 | 1.0001010 | 7 |
| 63 | 00111111 | 1.1111100 | 5 |

Figure 40: Normalization Examples

### 14.4.2 Round to Fit Within Fraction

Second, we have to round the fraction so that it fits within the designated bits. We denote the bits at the edge of the length of our `frac` with GRS as in the next figure.

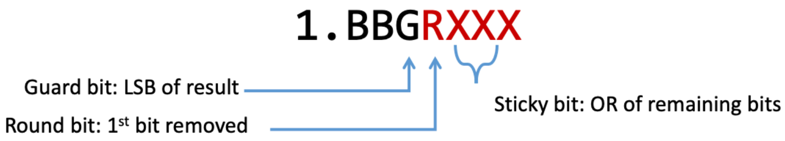
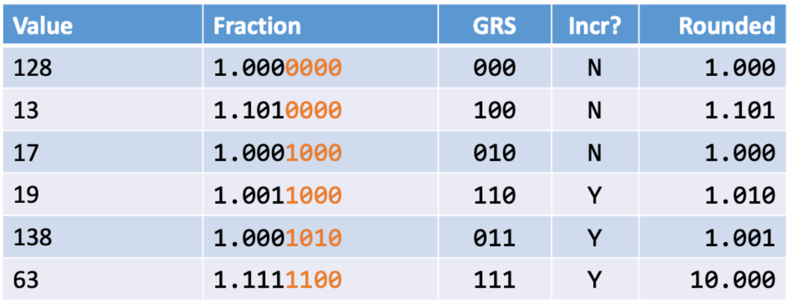


Figure 41: Creating Special Rounding Bits

- round $= 0 \Rightarrow$ round down/truncate
- round = 1, sticky = 1 $\Rightarrow$ round up
- round = 1, sticky = 0 $\Rightarrow$ round to even
  - ground $= 1 \Rightarrow$ round up
  - ground $= 0 \Rightarrow$ round down/truncate

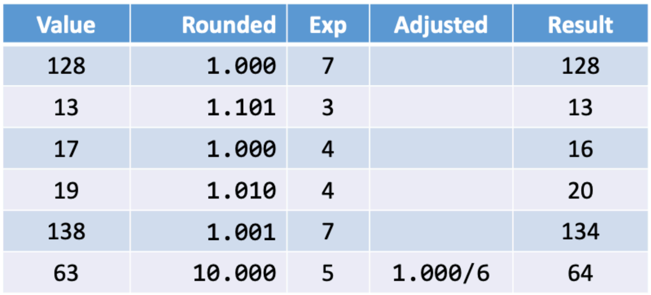


| Value | Fraction | GRS | Incr? | Rounded |
|---|---|---|---|---|
| 128 | 1.0000000 | 000 | N | 1.000 |
| 13 | 1.1010000 | 100 | N | 1.101 |
| 17 | 1.0001000 | 010 | N | 1.000 |
| 19 | 1.0011000 | 110 | Y | 1.010 |
| 138 | 1.0001010 | 011 | Y | 1.001 |
| 63 | 1.1111100 | 111 | Y | 10.000 |

Figure 42: Rounding Fraction Examples

### 14.4.3 Postnormalize to Deal with Effects of Rounding

Finally, rounding up might have led to an overflow. In that case, we have to shift right and increment the exponent to postnormalize.



| Value | Rounded | Exp | Adjusted | Result |
|---|---|---|---|---|
| 128 | 1.000 | 7 | | 128 |
| 13 | 1.101 | 3 | | 13 |
| 17 | 1.000 | 4 | | 16 |
| 19 | 1.010 | 4 | | 20 |
| 138 | 1.001 | 7 | | 134 |
| 63 | 10.000 | 5 | 1.000/6 | 64 |

Figure 43: Postnormalization Example

## 14.5 Floating-Point Addition and Multiplication

### 14.5.1 Floating-Point Multiplication

$$(-1)^s M 2^E = (-1)^{s1} M_1 2^{E1} \times (-1)^{s2} M_2 2^{E2}$$

- new sign $s = s_1 \hat{\ } s_2$

- new significand $M = M_1 \cdot M_2$
- new exponent $E = E_1 + E_2$

But this theoretical result probably doesn't fit into the floating point representation. If $M \geq 2$, $M$ needs to be shifted right and the exponent must be incremented. If $E$ is out of range, we have an overflow and get $\infty$. Finally, $M$ must be rounded to fit into `frac` precision.

The most difficult past is multiplying the significands, but that can be done by the already existing integer multiplier.

| **Mathematical Properties of Comm. Ring** | **FP Multiplication ($\neq$ Real Number Mult.)** |
|---|---|
| closed under multiplication | yes (but may generate infinity or NaN) |
| commutativity | yes |
| associativity | no (due to overflow and inexactness of rounding) |
| 1 as multiplicative identity | yes |
| multiplication distributes over addition | no |

Figure 44: Mathematical Properties of Floating Point Multiplication

Monotonicity holds except for infinities and NaNs.

### 14.5.2 Floating-Point Addition

$$(-1)^s M 2^E = (-1)^{s1} M_1 2^{E1} + (-1)^{s2} M_2 2^{E2}$$

where $E_1 > E_2$ without loss of generality.

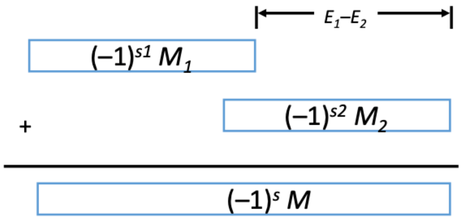


Figure 45: Signed Align and Add

The result of addition is simply signed align and add. Thus, the exponent of the result is just $E_1$ if we don't have to modify it due to shifting and rounding in the end. Specifically, because the precise result likely won't fit in our floating point representation, we have to modify the result:

- If $M \geq 2$, shift $M$ right and increment $E$.
- If $M < 1$, shift $M$ left $k$ positions, decrement $E$ by $k$
- overflow if $E$ out of range
- round $M$ to fit `frac`

| **Mathematical Properties of Abelian Group** | **FP Addition ($\neq$ Real Number Add.)** |
|---|---|
| closed under addition | yes (but may generate infinity or NaN) |
| commutativity | yes |
| associativity | no (due to overflow and inexactness of rounding) |
| $O$ as additive identity | yes |
| every element has additive inverse | almost (except infinities and NaNs) |

Figure 46: Mathematical Properties of Floating Point Addition

Additionally, monotonicity holds except for infinities and NaNs.

It follows that the order of adding FP numbers matters. We should always add from small to large numbers after sorting.

### 14.6 Floating Poin Puzzles

`x == (int)(float)x`

- This does not always hold, because a float may not store all integers.
- Floats have 24 significant bits, so they don't have the precision of 31 bits of a signed int.
- #significantBits ¿ #numberBitsInt

`x == (int)(double)x`

- This always holds, because the significant bits of a double have a enough precision for an int.

`f == (float)(double)f`

- This always holds, because a double may store a float without precision loss.

`d == (float)d`

- This does not always hold, because when comparing, we implicitly cast so that no precision is lost ( $f$ loat $\rightarrow$ double ). Hence, we may loose precision of $d$ when having it as a float in an intermediate step. This is as with integers, which are also promoted so that accuracy is preserved.

`f == -(-f)`

- This always holds, because floating points numbers are fully symmetric. We have a sign bit.

`2/3 == 2/30.`

- This does not always hold.
- `2/3` becomes 0 in integer arithmetic. After promoted to a floating point number, it does not equal the fraction of `2/30`. (being floating point division as 2 is promoted to a floating point number).

**d<0.0⇒((d*2)<0.0)**

- This always holds. Going to infinity would still preserve this equality.

**d>f⇒-f>-d**

- This always holds.
- Negation is just flipping the sign bit.

**d*d >= 0.0**

- This only breaks if d is NaN.

**(d+f)-d == f**

- This does not always hold, because d+f may become infinity, which remains infinity even with -d. And $\infty$ may $\neq$f.

## 14.7 SSE Floating Point

This refers to 'Streaming SIMD Extensions', which is an extension to the x86 architecture to support floating point arithmetics. We refer to them as SIMD, because they operate as vector instructions, which allows parallel operation on small vectors of integers or floats.

They are available with Intel's SSE family starting with Pentium 3, which supported 4-way single precision instructions. With SSE2 from Pentium 4, we also had 2-way double precision instructions. An all x86-64 processors have SSE3, which is a superset of SSE2 and SSE. Originally with the 8086 processor, one had to connect a separate unit (8087) to enable IEEE floating point arithmetic.

The floating point SSE instructions work on separate SSE3 registers. Those are 16 128 bit/16 byte register, which each can store 2 double precision or 4 single precision floating point numbers. They are named %xmm0 to %xmm15. All of those are caller saved and we pass arguments in % xmm0 to %xmm7. Further arguments are passed on the stack. The return value is stored in %xmm0.

In those XMM registers, we can store different data types and have different associated instructions accordingly.

- integer vectors: 16-way bytes, 8-way 2 bytes, 4-way 4 bytes/int

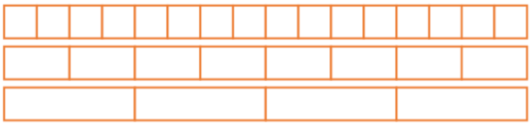


Figure 47: Integer Vectors in XMM Registers

- floating point vectors: 4-way single precision, 2-way double precision

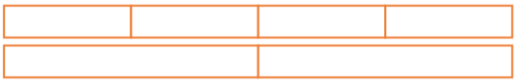


Figure 48: Floating Point Vectors in XMM Registers

- floating point scalars: single precision, double precision

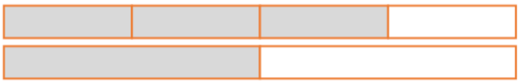


Figure 49: Floating Point Scalars in XMM Registers

As one can store integer vectors in those register, there are also integer operations. But we focus on floating point instructions, which look like `operation + single slot/packed + single precision/double precision`.

- `addps %xmm0 %xmm1`

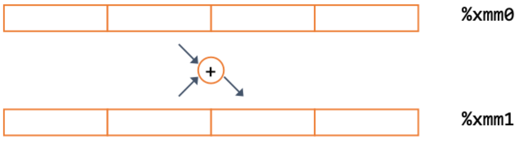


Figure 50: Integer Vectors in XMM Registers

- `addss %xmm0 %xmm1`

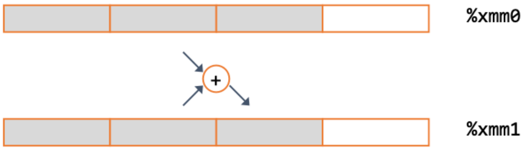


Figure 51: Floating Point Vectors in XMM Registers

Some more instructions:

- `movss S D: D <- S`
- `addss S D: D <- D + S`
- `subss S D: D <- D - S`
- `mulss S D: D <- D * S`
- `divss S D: D <- D / S`
- `maxss S D: D <- max(D,S)`
- `minss S D: D <- min(D,S)`
- `sqrtss S D: D <- sqrt(S)`

Furthermore, there are instructions to convert between single and double precision as well as integers and and floating point numbers:

- `cvtss2sd: single -> double`
- `cvtsd2ss: double -> single`

- `cvtsi2ss`: `int -> single` (rounding)
- `cvtsi2sd`: `int -> double` (rounding)
- `cvtsi2ssq`: `quad int -> single` (rounding)
- `cvtsi2sdq`: `quad int -> double` (rounding)
- `cvttss2si`: `single -> int` (truncation)
- `cvttsd2si`: `double -> int` (truncation)
- `cvttss2siq`: `single -> quad int` (truncation)
- `cvttsd2siq`: `double -> quad int` (truncation)

When the `int` can't contain the `double` (being `NaN`, $\infty$, ...) the `int` gets `T_MIN` in most cases. But this is undefined behavior technically. And when converting with packed and between differently sized types (single and double precision), we convert only as many entries as possible with the larger type and always consider the lowest blocks in the register if we can not fill the entire register.

When using floating point constants in code, the assembly looks unexpected. That is mostly, because assembly doesn't support floating point numbers in a sense, that we can only express bits and show them as integers. Thus, double precision floating point numbers will be stored as two consecutive `.long` (two 32 bit numbers). In the assembly, we see the respective int representation of those 32 bits listed separately.

When writing C code with floating pint numbers, one can instruct the compiler to vectorize the instructions instead of only operating on one floating point value at a time. With `gcc` one can use the flag `-ftree-vectorize` for example. But a speedup is not guaranteed and the optimizations are rather limited. `icc` (Intel compiler) is much better at autovectorizing. To get really high performance one has to do the vectorization oneself. To do so, one uses intrinsics, the C interface to vector instructions.

Intel AVX512 is a more recent extension and support 8-way double and 16-way single precision vectorization.

To get the highest performance, one often resorts to GPUs. Those support very high parallelism with several thousands of threads even on older cards. But programming GPUs is very hard, because one can't use generic C. Programming involves using frameworks/languages such as CUDA or OpenCL. There is also little support for higher-level languages, because compilers struggle to vectorize (complicated) code without a lot of code annotation. Restrictions for vectorization include: data must be aligned, data cannot overlap, loop sizes must be multiple of vector width, ...

# 15 Optimizations

Here, we consider how the compiler can optimize our code during compilation to increase performance. In the workflow/pipeline required to get an executable from a .c file, this is part of the actual compilation done by `cc1`. We will see what things the compiler can do to help and what may stop the compiler from doing so. Also, we learn about optimizations we can do manually but the compiler can't do by itself for various reasons.

We often consider matrix multiplication as an example of the different concepts here. As already seen in the introduction to the course, we can speed up matrix multiplication noticeably. The main speedup (20x) comes from: blocking/tiling, loop unrolling, array scalarization, instruction scheduling, ... This increases instruction level parallelism (ILP), leads to better register use, and reduces cache misses. That is all possible although all codes use the same fundamental approach of three nested loops with asymptotic complexity $\mathcal{O}(n^3)$.

We already know about asymptotic complexity from other classes. While constants are neglected there, they matter in the real world. Those aspects (among others) impact runtime performance:

- constants
- coding style (unnecessary procedure calls, unrolling, reordering, ...)
- algorithm structure (locality, ILP, ...)
- data representation (complicated structs or simple arrays)

To get very good performance, one must optimize at multiple levels: algorithms, data representation, procedures, loops, ... We try to understand how the compiler optimizes with consideration of execution and memory units to write the best code. This also includes measuring program performance and identifying bottlenecks. We want to do all this while still having maintainable (modular and general) code.

## 15.1 Optimizing Compilers

For `gcc`, we can use the `-O` flag to specify different optimization levels: `-O0`, `-O1`, `-O2`, `-O3` . But there are also other flags such as `-march=xxx`, with which one can specify the target architecture and `-m64` to state that this will only run on 64 bit machines. One should try different flags, because `-O3` will not always get better performance than `-O2`, for example. Also, other compilers such as `icc` may produce faster binaries than `gcc`.

Compilers are good at mapping a program to a machine. This includes optimizing: register allocation, code selection and ordering, dead code elimination, and eliminating minor (common) inefficiencies. But compilers usually do not increase asymptotic performance. It is up to the programmer to find the best algorithmic solution for a problem. For large inputs, algorithmic improvements may be more important than system-level optimizations.

Also, there are certain 'optimization blocker' (discussed later), which prevent the compiler to optimize for correctness reasons, because of implicit assumptions made by the programmer but not known to the compiler.

Compiler optimizations are conservative to guarantee correctness. Program behavior must not be changed under any circumstances. Most analysis is performed only within procedures as whole-program analysis is too expensive in most cases. Furthermore, analysis is based on static information. The compiler usually does not have information on runtime characteristics such as input values.

As the potential from just working with compiler flags, we may consider the following matrix multiplication. Without optimizations it runs in 820 cycles, with -O1 in 200 cycles, with -O2 in 132 cycles, with -O3 in 32 cycles, and with -O3 -m64 -march=haswell -fno-tree-vectorize in 8 cycles.

Listing 95: Matrix Multiplication for Benchmark

```
double a[4][4];
double b[4][4];
double c[4][4];  // set to zero
/* Multiply 4 x 4 matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
  int i, j, k;
  for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
      for (k = 0; k < 4; k++) c[i * 4 + j] += a[i * 4 + k] * b[k
          * 4 + j];
}
```

## 15.2 Code Motion and Precomputation

Code motion refers to moving some computation so that it is not unnecessarily repeated. This leads to a frequency reduction of said computation. This is also called precomputation, because we precompute a value once to reuse it.

But notice that compilers are not always good at exploiting arithmetic properties.

Listing 96: Unoptimized Code

```
void set_row(double *a, double
    *b, long i, long n) {
  long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

Listing 97: Optimized with Code Motion

```
void set_row(double *a, double
    *b, long i, long n) {
  long j;
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni+j] = b[j];
}
```

The compiler will hesitate to do some optimizations. For instance, because floating point operations usually are not associative and, thus, refrain from reordering. We would have to do this manually. So it is worth looking at the assembly to see what's going on.

Listing 98: Unoptimized Code

```
// C
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;

// ASSEMBLY
leaq   1(%rsi), %rax   # i+1
leaq   -1(%rsi), %r8   # i-1
imulq  %rcx, %rsi   # i*n
imulq  %rcx, %rax   # (i+1)*n
imulq  %rcx, %r8    # (i-1)*n
addq   %rdx, %rsi   # i*n+j
addq   %rdx, %rax   # (i+1)*n+j
addq   %rdx, %r8    # (i-1)*n+j
```

Listing 99: Optimized by Extracting Common Subexpression

```
// C
int inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;

// ASSEMBLY
imulq %rcx, %rsi # i*n
addq %rdx, %rsi # i*n+j
movq %rsi, %rax # i*n+j
subq %rcx, %rax # i*n+j-n
leaq (%rsi,%rcx), %rcx # i*n+j+n
```

## 15.3 Strength Reduction

This means replacing a costly operation with a simpler one. This is usually a specialized/"weaker" instruction. For instance, we may replace `*16` with `<<4`. The usefulness of such replacements depends on the target machine as the relative machine latency of the instructions determines the performance gain. Replacing multiplications with shifting is usually worth it. This may also be done automatically by the compiler if optimizations are enabled.

Another common strength reduction is to replace multiplications with additions.

Listing 100: Unoptimized Code

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    a[n * i + j] = b[j];
  }
}
```

Listing 101: Optimized with Strength Reduction

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    a[ni + j] = b[j];
  }
  ni += n;
}
```

## 15.4 Optimization Blockers

### 15.4.1 Procedure Calls

Listing 102: Unoptimized Code

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

Listing 103: Optimized with Code Motion

```c
void lower2(char *s) {
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++) {
    if (s[i] >= 'A' && s[i] <= 'Z') {
      s[i] -= ('A' - 'a');
    }
  }
}
```

It may seem surprising, but Listing 102 is actually $\mathcal{O}(n^2)$. This is, because `strlen(s)` is computed during each iteration to check the condition. And being implemented via iteration of the string, that function takes $\mathcal{O}(n)$, leading to $\mathcal{O}(n^2)$ totally. To improve performance, we can do manual code motion/precomputation and, thus, make the assumption that the string length is constant explicit.

The compiler can't do this optimization on its own, because it does not know about potential side effects of `strlen(...)`. And if calls to that function might return different values for the call, it cannot be guaranteed that the behavior is identical if we precompute the value. The compiler does not look inside functions that are called. It treats each as a black box.

Thus, either we have to do the code motion manually as above. Or we have to inline the function so that the compiler optimizes.

### 15.4.2 Memory Aliasing

In C one can easily have two memory references/pointers refer to a single location. That is a problem, because the compiler generally must assume that this is the case to guarantee correctness in any case. This assumption, however, prohibits the compiler from doing many optimizations. Mostly, because it must assume that when working with two pointers that those might reference to the same/overlapping memory. Hence, simple looking operations cannot be simplified, because they might not be that simple in certain edge cases.

Listing 104: Row Sum Comptutation

```c
void sum_rows1(double *a, double *b, long n) {
  long i, j;
  for (i = 0; i < n; i++) {
    b[i] = 0;
    for (j = 0; j < n; j++)
      b[i] += a[i*n + j];
  }
}
```

Listing 105: Assembly of Listing 104 (inner-most loop only)

```asm
.L53:
  addsd (%rcx), %xmm0
  addq $8, %rcx
  decq %rax
  movsd %xmm0, (%rsi,%r8,8)
  jne .L53
```

We can not guarantee that `b` and `a` don't overlap in the addresses they reference to. If they don't overlap, this works as our first intuition tells us. But if they overlap, the result will be quite different. Thus, the compiler must make the update to `b[i]` in every iteration of the loop to correspond to the program's semantics. Doing it just once after accumulating all values would be faster but not behave identically in every case.

One way to combat aliasing is scalar replacement. We use a temporary variable for arrays elements that are reused. This only works if there is no memory aliasing (at least with the same behavior) - in many cases we make this assumption.

Listing 106: Scalar Replacement Example

```
void sum_rows2(double *a,
    double *b, long n) {
  long i, j;
  for (i = 0; i < n; i++) {
    double val = 0;
    for (j = 0; j < n; j++)
      val += a[i*n + j];
    b[i] = val;
  }
}
```

Listing 107: use of `restrict` keyword

```
void sum_rows2(double *restrict
    a, double *restrict b, long
    n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Another option is to use the `restict` keyword, which as introduced with C99 for pointer declarations. This tells the compiler that during the lifetime of the annotated pointer, no other pointer will access the same data structure, which allows the compiler to ignore aliasing for that pointer. If the user violates this guarantee, the behavior is undefined.

## 15.5 Blocking and Unrolling

This introduces a technique that is relevant when working with matrices or data structured in matrices. Often many or even all array elements are reused but it is a bit tricky to take advantage of that. Blocking and loop unrolling provide techniques to utilize this reuse.

Listing 108: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);
/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
  int i, j, k;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
        c[i*n+j] += a[i*n + k]*b[k*n + j];
}
```

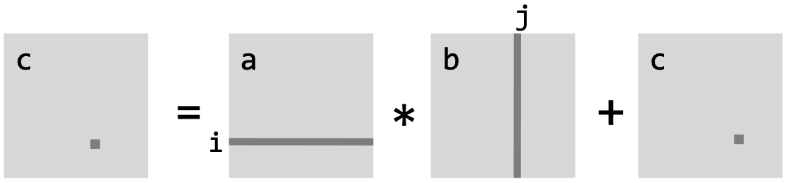


Figure 52: Matrix Multiplication Schematic

The performance of this and why blocking and unrolling works can be attributed cache optimization. When reading the section on caches, this will become more apparent. This example will also be discussed again in that section. Here, it is important to understand which matrix entries are in cache when the computation is done to understand when and how we want to reuse the elements while they are still in the cache.

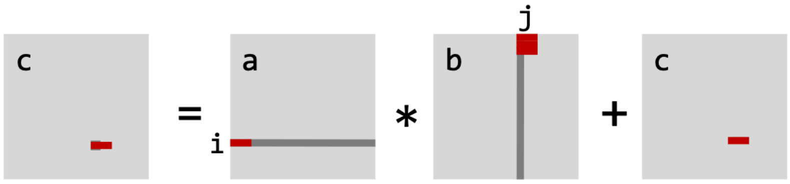


Figure 53: Entries In Cache - Beginning of c Entry Computation

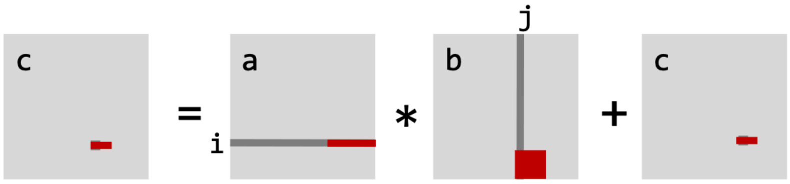


Figure 54: Entries In Cache - End of c Entry Computation

### 15.5.1 Blocking

Blocking means to do computations in blocks, which we saw to fit into the cache. By doing so, we compute on a set of elements, which can be kept in the cache so that we minimize cache misses and reduce wait time for memory.

Listing 109: Blocking in C

```
c = (double *)calloc(sizeof(double), n *n);
/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i += 2) // iterating over blocks
        for (j = 0; j < n; j += 2)
            for (k = 0; k < n; k += 2)
                for (i1 = i; i1 < i + 2; i1++) // in block
                    for (j1 = j; j1 < j + 2; j1++)
                        for (k1 = k; k1 < k + 2; k1++)
                            c[i1 * n + j1] +=
                              a[i1 * n + k1] * b[k1 * n + j1];
}
```

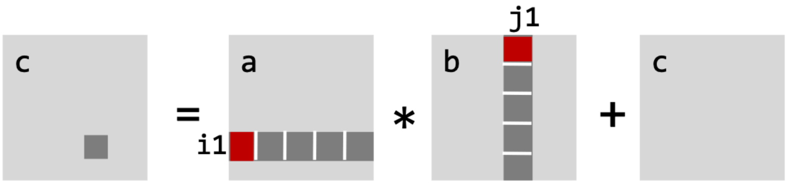


Figure 55: Blocking

### 15.5.2 Loop Unrolling

Unrolling loops means that we concatenate the different loop iterations directly and thus can also do cross-loop-iteration optimizations. Those optimizations can be very intricate. But in any case we save some jumping at run time.

In this case, we can unroll the computation of the four values in our 2 × 2 grid. This can be beneficial, because we use each element of our input matrices twice. If we unroll the four computations, we can do scalar replacement for the input values.

Listing 110: Unrolling Matrix Multiplication

```c
c = (double *) calloc(sizeof(double), n*n);
/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
  int i, j, k;
  for (i = 0; i < n; i+=2)
    for (j = 0; j < n; j+=2)
      for (k = 0; k < n; k+=2)
        <body>
}

<body>
c[i*n + j] =
a[i*n + k]*b[k*n + j] + a[i*n + k+1]*b[(k+1)*n + j] + c[i*n + j]

c[(i+1)*n + j] =
a[(i+1)*n + k]*b[k*n + j] + a[(i+1)*n + k+1]*b[(k+1)*n + j] + c
    [(i+1)*n + j]

c[i*n + (j+1)] =
a[i*n + k]*b[k*n + (j+1)] + a[i*n + k+1]*b[(k+1)*n + (j+1)] + c[
    i*n + (j+1)]

c[(i+1)*n + (j+1)] =
a[(i+1)*n + k]*b[k*n + (j+1)]
+ a[(i+1)*n + k+1]*b[(k+1)*n + (j+1)] + c[(i+1)*n + (j+1)]
```

# 16 Architecture and Optimizations

In this chapter, we consider how to optimize our programs beyond what the compiler can do for us by considering the specific characteristics of a processor. To do so, we need some general understanding of modern processor designs and their relation to performance. Much of the architectural foundations here have already been discussed in in the DDCA class and won't be elaborated on much.

Optimizations in general are about exploiting ILP (instruction level parallelism), where the performance is limited by data dependencies. Simple transformations which change the dependencies of subsequent instructions/tasks can yield dramatic performance improvements. Those optimizations often require the use of associativity and distributivity in computations, even if floating point arithmetics technically doesn't guarantee those. We then trade accuracy (floating point numbers are an approximation anyway) for performance.

To analyze and compare the performance of different setups, we need a program, which we optimize to see how those optimizations impact performance. Such a program whose performance we are interested in is called a benchmark. In addition, we need performance metrics. Those are quantitative measurements of the performance of our benchmark. We can look at the latency and throughput of requests.

## 16.1 Modern Processor Design

Most abstractly, we can compute the program execution time (”performance”) from three metrics: IC (instruction count of the program), CPI (cycles per instruction), and CCT (clock cycle time). We have program execution time $= IC \cdot CPI \cdot CCT$. We can reduce the $IC$ by optimizing our program and reducing its complexity in regards to the instruction count. The other two factors will be considered below.

The most simple model of a processor considers a sequential processor design. Those are not used in practice but provide fundamental understanding. Modern processors are pipelined, superscalar, out-of-order, have pipeline registers, pipeline control logic, data/control hazards, stalls, data forwarding, branch prediction, ... (all this was discussed in DDCA). All those enable faster systems by utilizing parallelism.

### 16.1.1 Sequential Processor Stages

In a sequential processor, each instruction passes through some stages: Fetch, Decode, Execute, Memory, Write Back, and PC Update. In one cycle, the signal must propagate through instruction memory, the register file, the ALU, data memory, ... Thus, the clock needs to run very slowly and h/w units are only active for a fraction of a cycle. This is wasteful in regards to h/w and time. Sequential processing is too slow to be practical.

### 16.1.2 Pipelined Hardware

With pipelining, we separate the stages and allow executing among those in a pipelined fashion. However, then we must consider data hazards (handled by data forwarding and stalling), control hazards (handled by canceling upon a misprediction and stalling), and control combinations.

By increasing the pipeline depth, we can reduct the $CCT$. But notice that reducing the $CCT$ by increasing the pipeline depth naturally also slightly increases the latency for each instruction to go through the pipeline. But the throughput on average still get's higher, which also keeps the $CPI$ low. But there are limits to increasing the pipeline depth, because pipeline registers add delay, we can't split the steps completely equally, we have clock skew, ...

Furthermore, we can't pipeline optimally/keep the pipeline full at the entire time. Thus, the $CPI$ does not only increase slightly due to latencies of pipeline registers. But also because of stalls and cancelled instructions when a branch is mispredicted. We write: $CPI = CPI_{base}+CPI_{stalls}$. Stalls may be caused by data hazards (read-after-write, write-after-read, write-after-write), control hazards (resolving jumps, branches), and memory latency.

### 16.1.3 Superscalar Processors

Modern processors are not only pipelined but also superscalar. This means that they can issue and execute multiple instructions in one cycle. Superscalar is understood as out-of-order execution engines here. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically. The benefit is clearly that superscalar processors can take advantage of instruction level parallelism that most programs have without any programming effort.

Superscalar CPUs for x86 are a bit more complicated, because x86 is a CISC (complex instruction set computer). On the one hand, instructions have different sizes when encoded. So the PreDecode separates those instructions, which requries more work that in a RISC machine. On the other hand, the complex instructions of the x86 ISA are split up in hardware into micro instructions, which is done to keep the hardware comparatively simple.

**Superscalar Processor Performance** .

| Instruction | Latency | Cycles/issue |
|---|---|---|
| Load/store | 4 | 1 |
| Int Add | 1 | 1 |
| Int Multiply | 3 | 1 |
| Int/Long Divide | $3 - 30$ | $3 - 30$ |
| Sngl/Dbl FP Multiply | 5 | 1 |
| Sngl/Dbl FP Add | 3 | 1 |
| Sngl/Dbl FP Divide | $3 - 15$ | $3 - 15$ |

Figure 56: Intel Superscalar Haswell CPU Latencies

Notice that an execution of several instructions may be latency bound or throughput bound. If we want to do five integer multiplications and those five multiplications are independent, we can just issue them one after another and are done in 7 cycles. We are throughput bound. But if they depend on each other, we must wait for one to complete for the next to start. This takes 15 cycles. We are latency bound.

So the major aspect to reach high performance is to keep pipelines filled and not let any h/w run idle.

### 16.1.4 Register Renaming

Remember that we discussed data hazards before.

- read after write (RAW): true dependence
- write after write (WAW): output dependence
- write after read (WAR): anti dependence

Notice that only RAW requires us to really wait for the previous computation to complete. The other dependencies only exist due to a limited number of registers. For instance, with write after write we only need to ensure that subsequent instructions read the second write. But nothing is stopping us from computing the second value to be written first. And we can actually do so with register renaming.

This bases on having more registers in hardware than the ISA makes available to the programmer. We map architectural registers to a larger pool of physical registers and give each new value produced its own physical register. That avoids WAW and WAR hazards.

### 16.1.5 Dataflow Execution

Through what we have now learned about modern processors, we see that they internally function somewhat like a dataflow processor. The traditional imperative execution view considers registers as fixed storage locations and instruction must be executed in the specified sequence to guarantee proper program behavior. But through pipelining, out-of-order execution, register renaming, etc. modern processors can be better understood with a functional view. Each write is understood to create a new instance of a value and operations can be performed as soon as all operands are available. There is no need to execute in the original sequence. Only the retirement/write-back to memory must be done in-order.

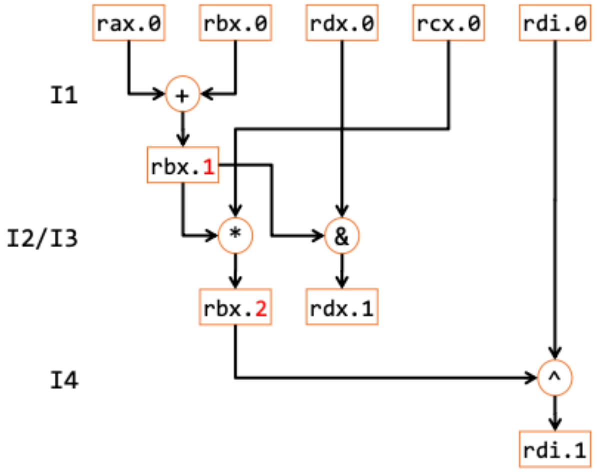


Figure 57: Dataflow Function View

## 16.2 Architecture-Based Optimization

We will consider this simple benchmark here, where `data_t` may be some data type such as `int`, `long`, `float` or `double`. `OP` and `IDENT` will be replaced by the operation and associated initial value: `+/0`, `*/1`.

Listing 111: Benchmark Program

```
struct vec {
  size_t len;
  data_t *data;
}

int get_vec_element(struct vec* v, size_t idx, data_t *val) {
  if (idx >= v->len)
```

```
    return 0;
  *val = v->data[idx];
  return 1;
}

void combine1(struct vec *v, data_t *dest) {
  long int i;
  *dest = IDENT;
  for (i = 0; i<vec_length(v); i++)
  {
    data_t val;
    get_vec_element(v,i,&val);
    *dest = *dest OP val;
  }
}
```

For this we will consider the Intel Haswell Architecture. Our CPU has 2 load units, 2 store units, 4 integer addition units, 1 integer multiplication unit, 2 floating point multiplication units, 1 floating point addition unit, and 1 floating point division unit. The latencies are as before an as in this figure.

| **Instruction** | **Latency** | **Cycles/issue** |
|---|---|---|
| Load/store | 4 | 1 |
| Int Add | 1 | 1 |
| Int Multiply | 3 | 1 |
| Int/Long Divide | 3 – 30 | 3 – 30 |
| Sngl/Dbl FP Multiply | 5 | 1 |
| Sngl/Dbl FP Add | 3 | 1 |
| Sngl/Dbl FP Divide | 3 – 15 | 3 – 15 |

Figure 58: Intel Superscalar Haswell CPU Latencies

One important metric is Cycles per Element (CPE). It measures how well the procedure would scale with big input vectors. When looking at the total cycles taken for sone amount of elements, it is comprised of an overhead and some amount of CPE. So the CPI is the slope of the execution time graph.

If we take our program as in `combine1` we get a different performance depending on the chosen optimization flags.

| **Method** | **Integer** | | **Double FP** | |
|---|---|---|---|---|
| Operation | **Add** | **Mult** | **Add** | **Mult** |
| Combine1 unoptimized | 22.68 | 20.02 | 19.98 | 20.18 |
| Combine1 -01 | 10.12 | 10.12 | 10.17 | 11.14 |

Figure 59

### 16.2.1 Basic Optimizations

- First, there are some optimizations we have already seen that compilers can not do. For instance, the compiler doesn't know that `vec_length(...)` doesn't have side effects and reevaluates it at each iteration for an overhead.

- Additionally, because of potential memory aliasing between dest and v, the compiler must mandate a write to dest in each iteration, although we semantically care about the accumulation and exclude memory aliasing in general.
- The element access with get_vec_element(...) also is quite inefficient, because it's an unnecessary procedure call with an overhead. And we perform some redundant bound checks in that procedure. By inlining it, we can increase performance.

Listing 112: Optimized Benchmark - combine4

```
void combine4(struct vec *v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i<length; i++)
    t = t OP d[i];
  *dest = t;
}
```

| **Method** | **Integer** | | **Double FP** | |
|---|---|---|---|---|
| Operation | **Add** | **Mult** | **Add** | **Mult** |
| Combine1 unoptimized | 22.68 | 20.02 | 19.98 | 20.18 |
| Combine1 -01 | 10.12 | 10.12 | 10.17 | 11.14 |
| Combine4 | 1.27 | 3.01 | 3.01 | 5.01 |

### 16.2.2 Latency Bound (Superscalar CPU)

When we look at our program, performance after the latest optimizations, we can see that our execution is currently latency bound. This is easy to understand, because we execute the operations one after another and the previous one must complete before the next one can start. The dependencies are sequential and our performance is dictated by the latency of the OP.

Also, we notice that the integer is even worse than latency bound. This comes, because we of course have some overhead with our for loop. But with the longer multiply and FP operations, those instructions can happen in parallel. But the 'quick' int add cannot swallow this overhead.

| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | **Add** | **Mult** | **Add** | **Mult** |
| Combine1 unoptimized | 22.68 | 20.02 | 19.98 | 20.18 |
| Combine1 -01 | 10.12 | 10.12 | 10.17 | 11.14 |
| Combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| Latency bound | 1.00 | 3.00 | 3.00 | 5.00 |

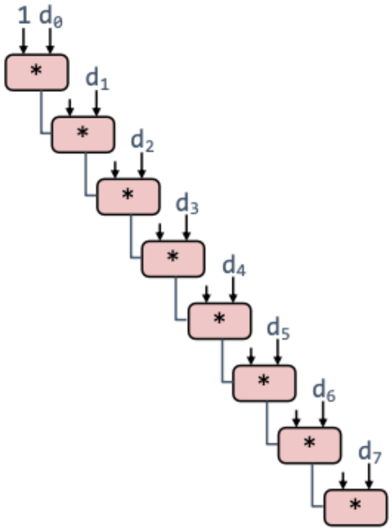


Figure 60: Reason for Latency Boundness

### 16.2.3 Loop Unrolling

Through loop unrolling, we perform 2x more useful work per iteration. This helps to improve performance for int add, because we now have additional time to execute the loop jumping in parallel through the additional computation.

Listing 113: Optimized Benchmark - `unroll2a_combine`

```
void unroll2a_combine(struct vec *v, data_t *dest) {
  long length = vec_length(v);
  long limit = length-1;
  data_t *d = get_vec_start(v);
  data_t x = IDENT;
  long i;
  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i+=2) {
    x = (x OP d[i]) OP d[i+1];
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    x = x OP d[i];
  }
  *dest = x;
}
```

| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | **Add** | **Mult** | **Add** | **Mult** |
| Combine1 unoptimized | 22.68 | 20.02 | 19.98 | 20.18 |
| Combine1 -01 | 10.12 | 10.12 | 10.17 | 11.14 |
| Combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| Unroll 2x1 | 1.01 | 3.01 | 3.01 | 5.01 |
| Latency bound | 1.00 | 3.00 | 3.00 | 5.00 |

### 16.2.4 Throughput Bound (Superscalar CPU)

Now, we can compare our performance the the theoretical throughput bound. Because we have 4 functional units for integers addition and 2 functional units for load, the integer add is bottlenecked by the 2 loads per iteration. For int multiply and FP add we only have one unit and are limited to one cycle per element. For fp multiply, we have two units and thus can also improve the throughput bound.

| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | **Add** | **Mult** | **Add** | **Mult** |
| Combine1 unoptimized | 22.68 | 20.02 | 19.98 | 20.18 |
| Combine1 -01 | 10.12 | 10.12 | 10.17 | 11.14 |
| Combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| Unroll 2x1 | 1.01 | 3.01 | 3.01 | 5.01 |
| Latency bound | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput bound | 0.50 | 1.00 | 1.00 | 0.50 |

### 16.2.5 Reassociation

Now we slightly change our program. This is called reassociation, because we change the order in which the operations are performed on the elements in one loop iteration. Notice that for FPs this may change the result of the program, because fp operations are not associative.

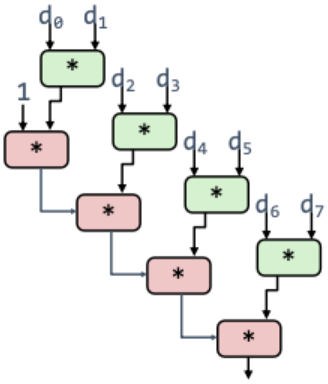
With this approach we break sequential dependence. Instead of having to wait for the `x` write of the previous iteration to finish, we can already start performing an operation. We thus need to stall/waste less time to wait for the `x` from the previous iteration to become available.

Listing 114: Optimized Benchmark - `unroll2aa_combine`

```c
void unroll2aa_combine(struct vec *v, data_t *dest) {
  long length = vec_length(v);
  long limit = length-1;
  data_t *d = get_vec_start(v);
  data_t x = IDENT;
  long i;
  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i+=2) {
    x = x OP (d[i] OP d[i+1]);
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    x = x OP d[i];
  }
  *dest = x;
}
```

As we shorted the linear dependencies by about half, this reduces the required cycles by 50 % generally. Notice that this didn't help for integer add, because integer addition is here limited by the loads as they take comparatively longer - 4 cycles instead of a 1 cycle add.

| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | **Add** | **Mult** | **Add** | **Mult** |
| Combine1 unoptimized | 22.68 | 20.02 | 19.98 | 20.18 |
| Combine1 -01 | 10.12 | 10.12 | 10.17 | 11.14 |
| Combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| Unroll 2x1 | 1.01 | 3.01 | 3.01 | 5.01 |
| Unroll 2x1a | 1.01 | 1.51 | 1.51 | 2.51 |
| Latency bound | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput bound | 0.50 | 1.00 | 1.00 | 0.50 |



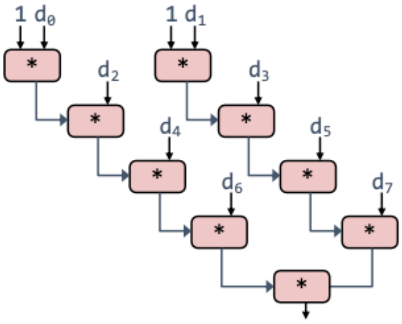
### 16.2.6 Loop Unrolling with Separate Accumulators

To fix the issue for the integer addition, where the load dominates the actual operation, we can use separate accumulators. This is a similar to reassociation as we still change the order of evaluation for improved performance. But now we don't have to wait for both loads to finish to continue but can wait on the loads and accumulate each independently.

Listing 115: Optimized Benchmark - `unroll2a_combine`

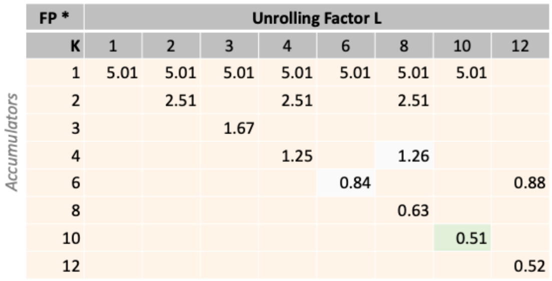
```c
void unroll2a_combine(struct vec *v, data_t *dest) {
  long length = vec_length(v);
  long limit = length-1;
  data_t *d = get_vec_start(v);
  data_t x0 = IDENT;
  data_t x1 = IDENT;
  long i;
  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i+=2) {
    x0 = x0 OP d[i];
    x1 = x1 OP d[i+1];
    }
  /* Finish any remaining elements */ for (; i < length; i++) {
    x0 = x0 OP d[i];
  }
  *dest = x0 OP x1;
}
```

| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | **Add** | **Mult** | **Add** | **Mult** |
| Combine1 unoptimized | 22.68 | 20.02 | 19.98 | 20.18 |
| Combine1 -01 | 10.12 | 10.12 | 10.17 | 11.14 |
| Combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| Unroll 2x1 | 1.01 | 3.01 | 3.01 | 5.01 |
| Unroll 2x1a | 1.01 | 1.51 | 1.51 | 2.51 |
| Unroll 2x2 | 0.81 | 1.51 | 1.51 | 2.51 |
| Latency bound | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput bound | 0.50 | 1.00 | 1.00 | 0.50 |



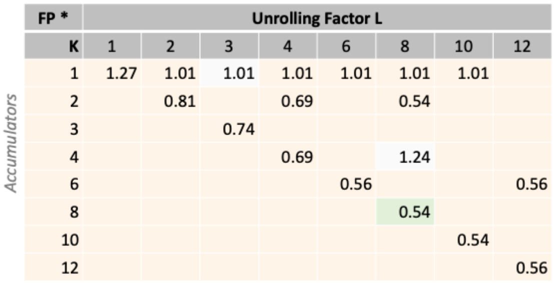
### 16.2.7 Combining Multiple Accumulators and Unrolling

Finally, we can unroll further. Specifically, we can unroll to any degree $L$ and accumulate $K$ results in parallel. But $L$ must be a multiple of $K$ for this to make sense. Otherwise, the accumulation would be not evenly distributed and performance doesn't profit. For different operations and data types, the best unrolling factor is different. Through a practical measurement one can best determine how far to unroll and accumulate.



| | FP * | Unrolling Factor L | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | K | 1 | 2 | 3 | 4 | 6 | 8 | 10 | 12 |
| Accumulators | 1 | 5.01 | 5.01 | 5.01 | 5.01 | 5.01 | 5.01 | 5.01 | |
| | 2 | | 2.51 | | 2.51 | | 2.51 | | |
| | 3 | | | 1.67 | | | | | |
| | 4 | | | | 1.25 | | 1.26 | | |
| | 6 | | | | | 0.84 | | | 0.88 |
| | 8 | | | | | | 0.63 | | |
| | 10 | | | | | | | 0.51 | |
| | 12 | | | | | | | | 0.52 |

Figure 61: Experiment for Double Mult.



| | FP * | Unrolling Factor L | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | K | 1 | 2 | 3 | 4 | 6 | 8 | 10 | 12 |
| Accumulators | 1 | 1.27 | 1.01 | 1.01 | 1.01 | 1.01 | 1.01 | 1.01 | |
| | 2 | | 0.81 | | 0.69 | | 0.54 | | |
| | 3 | | | 0.74 | | | | | |
| | 4 | | | | 0.69 | | 1.24 | | |
| | 6 | | | | | 0.56 | | | 0.56 |
| | 8 | | | | | | 0.54 | | |
| | 10 | | | | | | | 0.54 | |
| | 12 | | | | | | | | 0.56 |

Figure 62: Experiment for Integer Add.

| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | **Add** | **Mult** | **Add** | **Mult** |
| Combine1 unoptimized | 22.68 | 20.02 | 19.98 | 20.18 |
| Combine1 -01 | 10.12 | 10.12 | 10.17 | 11.14 |
| Combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| Unroll 2x1 | 1.01 | 3.01 | 3.01 | 5.01 |
| Unroll 2x1a | 1.01 | 1.51 | 1.51 | 2.51 |
| Unroll 2x2 | 0.81 | 1.51 | 1.51 | 2.51 |
| Best | 0.54 | 1.01 | 1.01 | 0.52 |
| Latency bound | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput bound | 0.50 | 1.00 | 1.00 | 0.50 |

### 16.2.8 SIMD Operations

But there is one trick to get even higher performance. Specifically, to use vector instructions/SIMD instructions. Then, one can truly parallelize the operations on multiple values. One unrolled loop iteration can then be done in one instruction instead of multiple instructions for each accumulator separately.

| **Method** | **Integer** | | **Double FP** | |
|---|---|---|---|---|
| Operation | **Add** | **Mult** | **Add** | **Mult** |
| Combine1 unoptimized | 22.68 | 20.02 | 19.98 | 20.18 |
| Combine1 -01 | 10.12 | 10.12 | 10.17 | 11.14 |
| Combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| Unroll 2x1 | 1.01 | 3.01 | 3.01 | 5.01 |
| Unroll 2x1a | 1.01 | 1.51 | 1.51 | 2.51 |
| Unroll 2x2 | 0.81 | 1.51 | 1.51 | 2.51 |
| Scalar Best | 0.54 | 1.01 | 1.01 | 0.52 |
| Vector Best | 0.06 | 0.24 | 0.25 | 0.16 |
| Latency bound | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput bound | 0.50 | 1.00 | 1.00 | 0.50 |
| Vector throughput bound | 0.06 | 0.12 | 0.25 | 0.12 |

# 17 Caches

The main performance bottleneck for modern processors is memory. Processors speed evolved according to Moore's law (doubling about every 18 months). But the memory infrastructure doesn't scale as well.

Consider the Intel Haswell architecture. Such processor can process at least 512 bytes/cycle. But the memory bandwidth is limited at 10 Bytes/cycles with a latency of 100 cycles. Thus, when having to work with lots of memory, it will reduce our performance noticeably.

The solution to the memory problem evolved to be mainly caches. Caches store a subset of the content of main memory for faster access for the processor. When we request some content from an address from the processor, the request goes through multiple levels of caches. When content is not in the cache, the request is forwarded to the next cache level or main memory. We talk of a miss when content is not in a cache, otherwise we have a hit. When we want to add content to a cache, we need to discuss two policies:

- placement policy: where the data goes in the cache
- replacement policy: determine which block gets evicted if there is not sufficient space

We manage the content of a cache in the form of cache blocks, which are blocks of consecutive memory addresses.

## 17.1 Cache Performance

To properly assess and discuss cache, we need to measure their performance. To do so, we define multiple metrics:

- miss rate
  - fraction of the memory addresses not found in the cache
  - $\frac{\text{misses}}{\text{accesses}} = 1 - \text{hit rate}$
  - For L1 caches, the miss rate is usually $3 - 10\%$
  - For L2, ... caches, the miss rate can be very low - $\leq 1\%$
- hit time
  - time to deliver a line/block in the cache to the processor (this includes the time to check whether that block is even in the cache)
  - For L1 caches, the hit time usually is 1-2 cycles.
  - For L2 caches, the hit time usually is 5-20 cycles.
- miss penalty
  - additional time required because of a miss
  - When we miss and not only need to look in another cache but have to go to main memory, the latency is typically 50-200 cycles.

When we consider performance across two caches, we must consider the miss rate for each level and then the latency of processing the request at the next level.

- average memory access time $= \text{hitTimeL1} + \text{missRateL1} * \text{missPenaltyL1}$
- $\text{missPenaltyL1} = \text{hitTimeL2} + \text{missRateL2} * \text{missPenaltyL2}$

- missPenaltyL2 = DRAMaccessTime + (blockSize/bandwidth)

We distinguish between different types of cache misses:

- cold (compulsory) miss: occurs on first access to block
- conflict miss: most caches limit placement to a small subset of available slots; the cache may be large enough to store this but this block got evicted because all possible slots are in use
- capacity miss: set of active cache blocks (working set) is larger than the entire cache
- coherency miss: this occurs on multiprocessors where a different core changed so content so that the current content is now invalid

## 17.2 Cache Organization

- Caches may have (basically always) multiple sets. Each cache block is assigned to one set.
- Associativity of a cache says how many lines are available per set. Meaning, how many cache blocks assigned to one set can be stored at the same time in the cache.

  If associativity is 1, we say that we have a direct mapped cache. If associativity is $x \geq 2$, we say that we have a $x$-way set-associative cache.
- Each cache block contains of course the bytes according to the size of a cache blot. But additionally, it contains a valid bit (which is used for fast eviction) and a tag, which clarifies to which address the cache block in that set belongs, as the set only specifies part of the address.

There really is only one replacement and placement policy with direct mapped caches as if our cache block is not present we only have one choice to evict and place our cache block. If we have associativity $\geq$ 2 we need proper replacement and placement policies to choose a line for evication/replacment. Such policies are: random, least recently used (LRU), ...

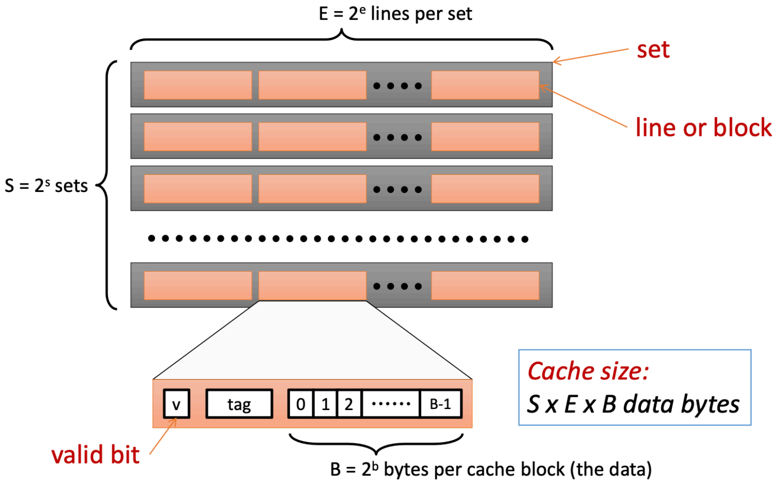


Figure 63: Cache Organization

## 17.3 Cache Reads

When we want to read from a cache, we have the memory address that is requested and need to look at all possible positions, where that cache block may be in the cache.

1. We use the index bits to identify the corresponding set.
2. We check all lines in that set for matching tag bits. Once we found a match, we must validate that the valid bit is set or otherwise continue to look at the remaining lines.
3. If we find the cache block in that set, we use the offset bits to identify the corresponding byte(s) in the cache block content. If the cache block is not in the set, we must forward the request.

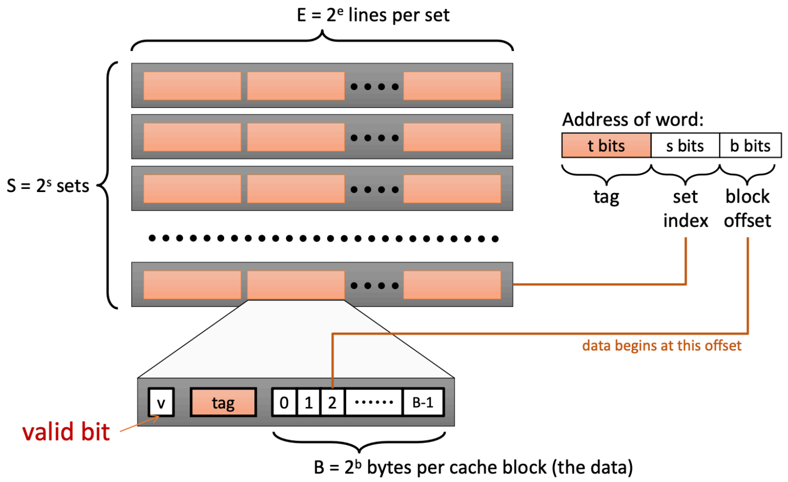


Figure 64: Cache Read

## 17.4 The Memory Hierarchy

Caching happens at various levels of systems. We have already read about L1 and L2 caches as two cache levels between the processor and main memory. Today, we often also have an L3 cache. But caching happens at higher levels too, although we don't really consider them here.

| Cache type | What is cached? | Where is it cached? | Latency (cycles) | Managed by |
|---|---|---|---|---|
| Registers | 4/8-byte words | CPU core | 0 | Compiler |
| TLB | Address translations | On-chip TLB | 0 | Hardware |
| L1 cache | 64-byte blocks | On-chip L1 | 1 | Hardware |
| L2 cache | 64-byte blocks | On-chip L2 | 10 | Hardware |
| Virtual memory | 4kB page | Main memory (RAM) | 100 | Hardware + OS |
| Buffer cache | 4kB sectors | Main memory | 100 | OS |
| Network buffer cache | Parts of files | Local disk, SSD | 1, 000, 000 | SMB/NFS client |
| Browser cache | Web pages | Local disk | 10, 000, 000 | Web browser |
| Web cache | Web pages | Remote server disks | 1, 000, 000, 000 | Web proxy server |

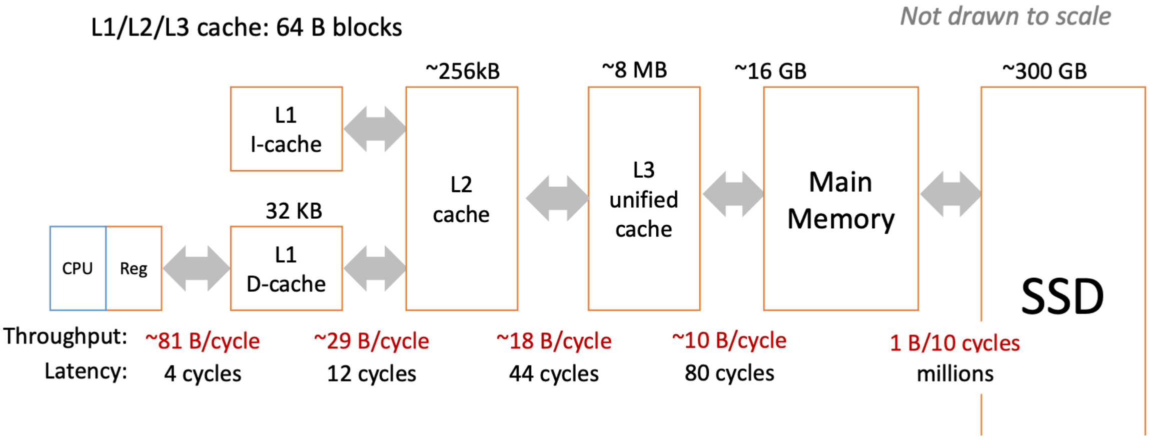


Figure 65: Example Memory Hierarchy - Core i7

## 17.5 Cache Writes

So far we considered the case that some cache blocks are already in the cache and we only want to read from them. But of course we also want to write to memory. There are two possibilities: We want to write to a cache block that is in the cache or that is not in the cache. For each case there are two common policies:

- write on hit
  - write-through: We write immediately to memory so that memory is always consistent with the cache copy. But this is inefficient if we do multiple writes to the same line.
  - write-back: We defer the write to memory until replacement of the line. To do so we need a dirty bit for each block to note whether it was edited or not. This generally has higher performance but is more complex to implement.
- write on miss
  - write-allocate: This loads a cache block into the cache if we want to write to it and it is not yet present. This is common with write-back caches.
  - no-write-allocate: This writes directly to memory. This is common with write-through caches.

## 17.6 Other Cache Features

- unified: serves both instruction and data fetches
- private: only one core uses this cache
- shared: multiple cores share the cache
- inclusive: anything in this cache is also in every lower-level cache
- exclusive: anything in this cache is not in any lower-level cache

Another special type of caches are software caches. L1 and L2 caches considered so far are (usually) implemented in hardware. Software caches are commonly used in higher-level caches. For instance, virtual memory has a software component, because the operation system contributes to the cache management. Software caches commonly have some special properties:

- fully associative (index structures like hash tables are common)
- complex replacement policies (as misses are way more expensive at higher levels, we can spend much more work on the best possible placement)
- not constrained to single ”block” transfer

## 17.7 Cache Optimization

Caches work/increase performance because of locality. Locality means that programs tend to use data and instructions with addresses near or equal to those they have used recently.

- temporal locality: recently referenced items are likely to be referenced again in the near future
- spatial locality: items with nearby addresses tend to be referenced close together in time

So, to optimize code for the memory hierarchy, we have to write code that has locality. For spatial locality we want to access data contiguously. And for temporal locality, we want to do as much of the necessary work on some data as close as possible. This can be done by choosing a proper algorithm and transforming loops. We can work towards cache and register locality.

### 17.7.1 Example 1

This example exhibits temporal locality on `sum` and spatial locality on `a[]`.

```
sum = 0;
for (i = 0; i < n; i++)
  sum += a[i];
return sum;
```

### 17.7.2 Example 2

This utilizes locality through row-majority in C. Switching the loops would fail to exploit locality.

```
int sum_array_rows(int a[M][N])
{
  int i, j, sum = 0;
  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      sum += a[i][j];
  return sum;
}
```

### 17.7.3 Example 3

We look at matrix multiplication. First we analyze the cache performance of a basic implementation. Afterwards we will do some basic optimizations and repeat the cache analysis.

**Basic Implementation** .

Listing 116: Matrix Multiplication

```
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      for (k = 0; k < n; k++) {
        c[i*n+j] += a[i*n + k]*b[k*n + j];
      }
    }
  }
}
```

We assume that the matrix elements are doubles, that a cache block has a size of 8 doubles, and that the cache size $C$ is noticeably smaller than the matrix size $n$, i.e., $C \ll n$, while being a multiple of $n$.

In the first iteration of the outer loop we will incur $\frac{n}{8} + n = \frac{9n}{8}$ cache misses. There are $n^2$ such iterations of which all will incur the same cache misses. So that total misses will amount to $\frac{9n}{8} \cdot n^2 = \frac{9}{8}n^3$.

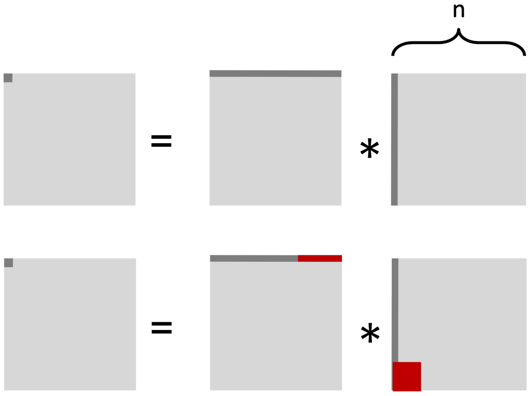


Figure 66: Unoptimized Matrix Multiplication Cache Utilization

**Optimized Implementation** .

Listing 117: Matrix Multiplication with Blocking

```
c = (double *) calloc(sizeof(double), n*n);

void mmm(double *a, double *b, double *c, int n) {
  int i, j, k;
  for (i = 0; i < n; i+=B)
    for (j = 0; j < n; j+=B)
      for (k = 0; k < n; k+=B)
        /* B x B mini matrix multiplications */
          for (i1 = i; i1 < i+B; i1++)
            for (j1 = j; j1 < j+B; j1++)
              for (k1 = k; k1 < k+B; k1++)
                c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

Again, we make some assumptions for our analysis:

- cache block size is 8 doubles
- cache size $C$ is much smaller than the matrix size $n$, i.e., $C \ll n$
- $B$ is a multiple of a cache block size
- $n$ is a multiple of $B$

During the first iteration for some block, there will be $\frac{B^2}{8}$ misses for each block. As there are $\frac{2n}{B}$ blocks to be considered for one target blocks, this leads to $\frac{2n}{B}\frac{B^2}{8} = \frac{nB}{4}$ misses. As all target blocks incur the same misses this leads to a total number of misses of $(\frac{n}{B})^2 \cdot \frac{nB}{4} = \frac{n^3}{4B}$.

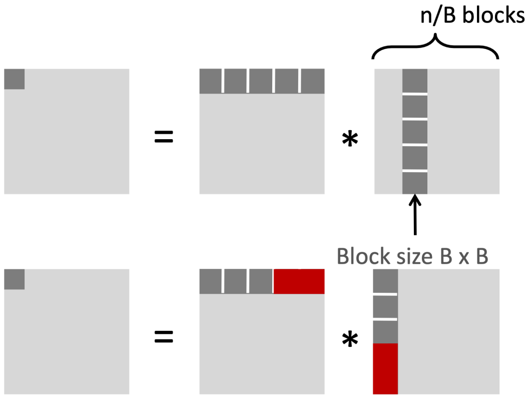


Figure 67: Simple Optimized Analysis

If we now assume that $B \leq$ cache block size but still assume alignment of the blocks with caches blocks, the computation changes.

$$\underbrace{\left(\frac{n}{B}\right)^2}_{\text{number of iteration}} \cdot \left( \underbrace{\frac{Bn}{8}}_{\text{misses in matrix a per iteration}} + \underbrace{n}_{\text{misses in matrix b per iteration}} \right)$$

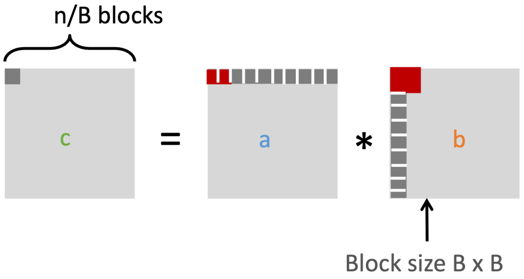


Figure 68: Optimized Analysis

# 18 Exception

Processors only execute instructions from startup to shutdown. They get a sequence of instructions and execute them. The sequence of those instructions is called the control flow. So far we have seen jumps, branches, call, and return to change the control flow. Those are caused by our program.

We now consider another way to alter control flow based on changes in the system state instead of program state. For instance, data might arrive via a network, keys are pressed on the keyboard, a system timer expires, ... This corresponds to the concept of exceptions: When certain events occur, we alter the control flow.

Exceptions as familiar from high-level languages such as Java are different. They are enabled by software constructs. But the exceptions we will discuss here rely on hardware support for their functioning.

Fundamentally, an exception is a transfer of control to the OS in response to some event. After handling the event, the OS may abort the program which ran, run the last instruction from the program again, or proceed to the next instruction from the program. The code/part of the OS that handles the exception/the event that caused the exception is called the exception handler. We say that it processes the exception.

We distinguish between two kinds of exceptions. Synchronous exceptions are the result of a program executing an instruction. Asynchronous exceptions are the result of events that are external to the processor/current process.

Beyond that, we may distinguish exceptions further.

| **Type of Exception** | **Cause** | **Async/Sync** |
|---|---|---|
| Interrupt | Signal from I/O device, ... | async |
| Trap | intentional exception | sync |
| Fault | potentially recoverable error | sync |
| Abort | nonrecoverable error | sync |

Figure 69: Exception Types

## 18.1 Exception Vectors & Kernel Mode

At boot time, the OS allocates and initializes the exception table. The table's address is stored in the Exception Table Base Register. Each type of event has a unique exception number $k$, which is used to index the table. The table stores addresses, which correspond to locations with code to handle that specific exception. So, the handler for some exception $k$ is called each time the exception $k$ occurs.

When an exception occurs and the control flow is changed to the exception handler, this also causes a switch to kernel mode (supervisor mode/privileged mode/ring 0). This gives the exception handler additional possibilities. Even for the same ISA/architecture, the behavior when kernel mode is activated may vary in details.

- access to system state
- some other exceptions are disabled
- additional instructions and registers
- MMU behavior may change

The OS (kernel) always comprises a set of trap handling functions (system calls for programs) and code to illustrate a user-space for programs (for instance regarding virtual memory).

| | |
|---|---|
| 0 | Divide error |
| 1 | Debug exception |
| 2 | Non-maskable interrupt (NMI) |
| 3 | Breakpoint |
| 4 | Overflow |
| 5 | Bounds check |
| 6 | Invalid opcode |
| 7 | Coprocessor not available |
| 8 | Double fault |
| 9 | Coprocessor segment overrun (386 or earlier) |
| A | Invalid task state segment |
| B | Segment not present |
| C | Stack fault |
| D | General protection fault |
| E | Page Fault |
| F | Reserved |
| 10 | Math fault |
| 11 | Alignment check |
| 12 | Machine check |
| 13 | SIMD floating point exception |
| 14 - 1F | Reserved to Intel |
| 20 - FF | Available for external interrupts |

Figure 70: x86 Exception Vectors

## 18.2 Synchronous Exceptions

### 18.2.1 Traps

Traps are invoked intentionally and return the control flow to the next instruction of the program after completion. Those may be system calls, breakpoints for debugging, special instructions, ...

One common system call/trap is to open a file.

1. the user calls `open(filename, options)`
2. this executes a system call (`int` opcode)
3. the OS exception handler then finds or creates the file and prepares it for reading/writing (which includes loading it into memory)
4. returns an integer file descriptor and control flow back to the program

### 18.2.2 Faults

Faults are unintentional exceptions, where the handler may still be able to recover from the exception. If recover is successful, it returns control flow to the program and executes the faulting instruction again. If recover fails, it aborts the program. Examples are: page faults (recoverable), protection faults (unrecoverable), floating points exception, ...

Example: Page Fault

1. user write to memory location
2. the corresponding page is currently not in memory but on disk
3. page handler must load page into physical memory
4. returns to faulting instruction (memory write is successful on the second try)

Example Page Fault & Invalid Memory Reference

1. user writes to memory location
2. the corresponding page is currently not in memory but on disk
3. page handler must load page into physical memory but detects an invalid memory address
4. sends SISEGV signal to user process
5. user process exits with ”segmentation fault”

### 18.2.3 Abort

Those are unintentional and unrecoverable exceptions. They always abort execution of the program. Examples are: parity error (disk corruption), machine check (errors), ...

## 18.3 Asynchronous Exceptions

Those are exceptions, which are caused by events external to the processor. After the exception has been processed, the control flow is set to continue at the 'next' instruction of the program. Some interrupts are:

- I/O interrupts: network package arriving, data arriving from disk, hitting keyboard shortcuts
- hard reset interrupt: hitting the reset ('power') button
- soft reset interrupt: hitting ctrl-alt-delete on a PC

Interrupts are registered through the CPU interrupt-request line (edge- or level-triggered). The interrupts are forwarded to an interrupt handler. Some interrupts are 'maskable', which means that they may be delayed or ignored. But some interrupt are also nonmaskable. Then, based on the priority of the interrupt, the interrupt is dispatched to the exception handler corresponding to the interrupt vector. Masking and prioritization is done with a PIC (programmable interrupt controller) on modern machines. PICs are discussed below.

Neglecting the PIC for now, x86 has two interrupt pins: INTR (interrupt request) and NMI (non-maskable interrupt). The PIC is 'before' the INTR pin. The PIC then also makes sure that the CPU gets one interrupt at a time. The CPU uses the INTA pin to acknowledge the interrupt. Afterwards, the pins D0 to D7, which are part of the data bus, are used by the PIC to specify the interrupt vector to the CPU.

### 18.3.1 x86 NMI

When an interrupt is triggered through this pin, the CPU completes the current instruction and then issues the exception handler with the interrupt vector 2. No masking may occur! This cannot be disabled by the processor! Non-maskable interrupts are reserved for the most important events such as major hardware failures and the watchdog timer (which is used to monitor general system health).

Notice that there may be multiple devices connected to this pin. Thus, it cannot be said certainly which caused the interrupt. Hence, the CPU must poll potential sources. This is a slow process, because the to be polled devices can have quite high latencies. Then, the CPU might even get a second interrupt during that time.

### 18.3.2 x86 INTR (IRQ)

Notice that such interrupts can be enabled and disabled by the program by setting the `IE` system flag. The `CLI/STI` instructions are used to set/clear the interrupt flag. If interrupts are enabled, the CPU completes the current instruction and then proceeds like this:

1. the CPU acknowledges the interrupt using the INTA pin
2. the interrupt vector is acquired via the data bus
3. the control flow is redirected to the exception handler corresponding to the interrupt vector

Notice that the x86 interrupt vectors leave space for additional external interrupts, which may be configured by device drivers etc. (device drivers are discussed in the final section).

## 18.4 Interrupt Controllers

If a processor would just expose the pins INTR, NMI, and D0 to D7 to get informed about interrupts there would be some difficulties:

- identifying who caused an interrupt
- getting the correct interrupt vector (multiple devices may write simultaneously)
- what happens when multiple interrupts happen at once? (should not miss one!)

Those problems were especially dramatic in early days, when CPUs only had one pin for IRQ and NMI.

The solution to this problem are programmable interrupt controllers (PIC).

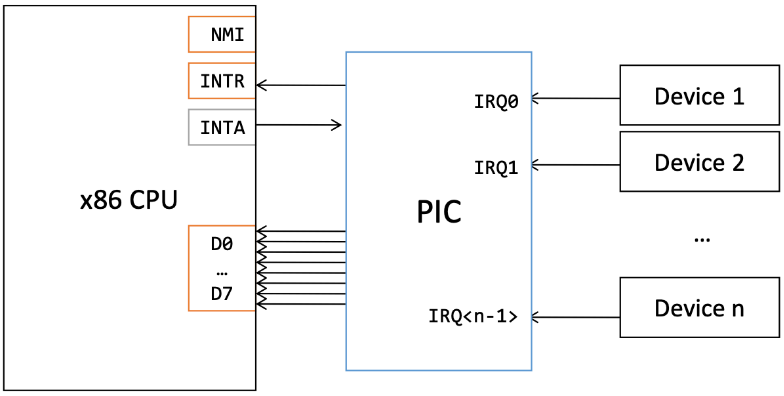


Figure 71: Programmable Interrupt Controller

Each device connects separately to the PIC, which maps the pins for the individual devices to interrupt vectors. The mapping is picked by the OS when programs register interrupt handlers and the PIC is configured. A PIC has various capabilities:

- mapping device to interrupt vector
- buffer simultaneous interrupts: deliver interrupts separately and don't loose any interrupts
- prioritize interrupts: prioritization of devices selected by the OS
- selectively mask individual device's interrupts: useful for high-interrupt rate devices

All modern processors have a 'local APIC' (local advanced programmable interrupt controller), which is integrated onto the same die as the core itself. Those have even more capabilities:

- inter-processor interrupts
- programmable timers
- sophisticated interrupt scheduling

Generally, it is the OS's job to discover and configure interrupt controllers present in the system.

# 19 Virtual Memory

So far we jus looked at memory by the addresses supplied to instructions in a programs machine code. All those addresses are virtual memory addresses. We can specify addresses with 64 bits so can reference $2^{64}$ different bits. This even exceeds the memory capacity of high-performance machines by far. Those virtual addresses are only an illusion created for each process and do not correspond to the addresses of physical, i.e., 'actual' memory. The indirection to a physical memory location is implemented with a hierarchy of different mappings.

All allocations we do from the program (`malloc` etc.) happen inside the program's virtual memory. The allocation then tells the system to map that virtual memory to some physical memory so that we can actually work with those addresses.

## 19.1 Why Virtual Memory?

There are some fundamental reasons for virtual memory:

- Efficient use of limited main memory (RAM): We can only use it to store limited data and consider it a cache for the disk to store parts of the virtual address spaces. Some non-cached parts are stored on disk and unallocated parts are stored nowhere. We only keep active areas in the cache/memory and transfer data back and forth as needed.
- Simplification of memory management for programmers: Each process gets the same large address space and must not worry about which addresses are used by other programs.
- Isolated address space (security): Programs can't interfere with another's memory and, thus, are prevented from accessing privileged information.

### 19.1.1 Problem 1: Limited Physical Memory Capacity

The $2^{64}$ bytes of the virtual memory space exceed the few Gigabytes of memory of modern machines by magnitudes. We use VM as a 'cache'. Virtual memory corresponds to $N = 2^n$ contiguous bytes. We can think of that data being stored on disk. The physical main memory (DRAM) is then used as a cache for allocated virtual memory. Thus, we can create the illusion of the entire $64$ bit address space while only having a subset of this data on-hand.

When discussing caching, we looked at cache blocks. Here we call the units, which are moved between disk and main memory, pages. Those usually are about 4-8 KB in size (even up to 4 MB in some cases).

Although we understand main memory as a cache for disk, this 'main memory cache' is still very slow compared to registers, L1, and L2 caches. To combat this, main memory is designed with some approprite properties:

- Large page size: usually 4-8 KB (up to 4 MB)
- The cache is fully associative to not occur unnecessary misses. This requires a complicated mapping function, which is different from caches.
- Due to the large penalty for a miss, we can afford and it makes sense to use highly sophisticated (and expensive) replacement policies.
- write-back instead of write-through

This is very efficient in practice because of locality. At any point, programs tend to access a set of virtual pages called the working set. If the size of the working set is smaller than main memory, we have good performance after compulsory misses. But if the size of the working set is larger than main memory we occur trashing. This leads to a performance meltdown as pages are constantly switched between main memory and disk.

### 19.1.2 Problem 2: Memory Management

All virtual pages from different processes are mapped to physical memory. The allocated physical pages are scattered in memory. Through a well-chosen virtual to physical mapping, the OS can simplify memory allocation and management. The OS can move physical pages and just update the indirection so that the program does not notice it. Code and data can be shared among programs because respective virtual addresses can map to the same physical address. This is useful, for example, for shared library code.

### 19.1.3 Problem 3: Protection

There is some mechanism to translate/map a virtual address to a physical address. Besides the actual mapping, we also enclose some meta-information in this translation. The meta-data includes permission bits with fields for supervisor, read, write, ... The page fault handler checks those permissions before providing the remapped address's content. If the permissions would be violated, it sends the process a SIGSEGV (segmentation fault) signal.

## 19.2 Address Translation

First, we need to learn to distinguish beteween different address spaces:

- linear address space: ordered set of contiguous natural numbers addresses $\{0, 1, 2, ...\}$
- virtual address space: set of $N = 2^n$ virtual addresses $\{0, 1, 2, ..., N - 1\}$
- physical address space: set of $M = 2^m$ physical addresses $\{0, 1, 2, ..., M - 1\}$

A virtual address from the address space of a program is mapped to a physical address for one program. Thus, we separate the address from the stored data. And multiple virtual addresses from different programs may reference the same physical address.

Physical addressing is only still used on some embedded systems such as cars, elevators, toasters, ... Virtual addressing is used on all sophisticated modern systems such as laptops, desktops, servers, workstations, ...

The necessary translation from virtual to physical addresses is done by the MMU (memory management unit), which is configured and maintained by the OS. Thus, virtual memory is a collaboration of hardware and software.

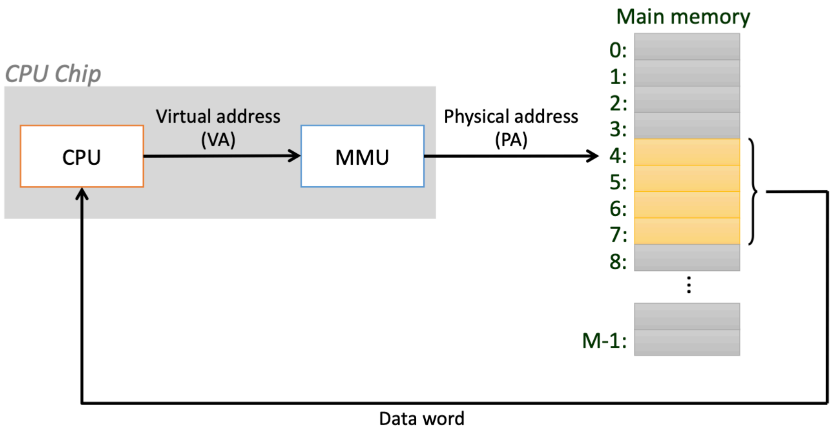


Figure 72: Simplified Address Translation Process

The MMU must convert a VA (virtual address) to a PA (physical address). Each VA is split into a VPN (virtual page number) and VPO (virtual page offset). The VPN is used to index into the page table. A page table maintains the PPN (physical page number) for each VPN. Hence, for each process a separate page table must exist. The page table not only stores the PPN but also whether this is a valid page, i.e., currently in memory. And if not in memory but has been used, this also provides information on how to locate the data on disk.

The page table is stored in memory itself and its location is given by the PTBR (page table base register). The VPO is identical for the PPO. Notice that the PPN is usually smaller (has less bits) than the VPN. But for x86 this does already not hold. Mostly, because virtual addresses are less than 64 bits in x86 under the hood, and x86 wants to support virtualization/running multiple operating systems at the same time. Hence, we need a multiple of everything - including the availability of physical addresses.

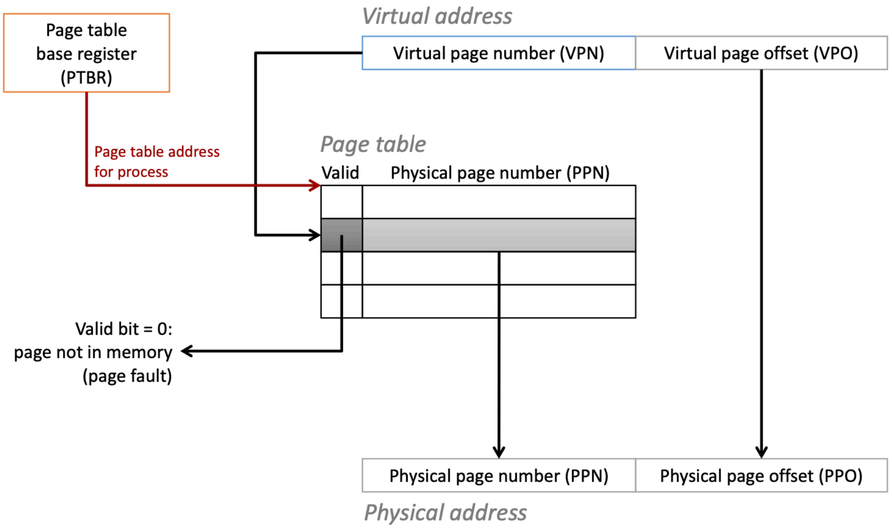


Figure 73: Address Translation

Now we will consider the individual steps of translating an address. Here we consider physically indexed and physically tagged caches. Just remember this for now as it will be explained later what it means.

**Page Hit**

1. processor sends virtual address to MMU
2. MMU requests PTE from page table in memory
3. MMU receives PTE from memory
4. MMU sends physical address to cache/memory
5. cache/memory sends data word to processor

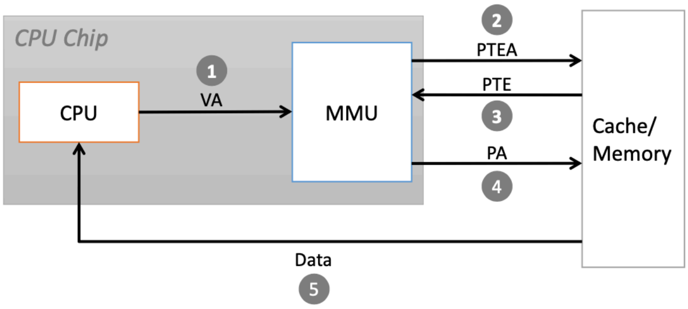


Figure 74: Page Hit Address Translation Process

**Page Fault**

1. processor sends virtual address to MMU
2. MMU requests PTE from page table in memory
3. MMU receives PTE from memory
4. valid bit is zero, so MMU triggers page fault exception
5. handler identifies victim page to evict (if necessary) and pages it out of disk if the dirty bit is set
6. handler pages in the new page and update the PTE in memory
7. handler returns to original process, restarting faulting isntruction

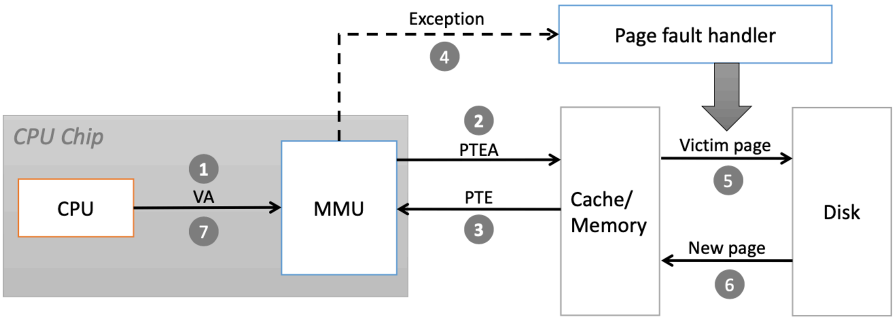


Figure 75: Page Fault Address Translation Process

## 19.3 Translation Lookaside Buffers (TLB)

For one memory request from the processor we now have to serve two actual requests to memory. That noticeably degrades performance. To reduce the cost of memory requests we cache page table entires (PTEs).

Instead of using the existing caching infrastructure, we use Translation Lookaside Buffers as a cache for PTEs. This is a small hardware cache in the MMU that maps VPNs to PPNs. For a small subset of the virtual address space, this contains the complete page table entries.

**TLB Hit** If we hit in the TLB we save an entire memory request.

1. processor sends virtual address to MMU
2. MMU indexes TLB with VPN
3. MMU gets PTE and validates that the valid bit is set (TLB hit) and that the entry contains a PPN (and not only a disk reference).
4. MMU sends the constructed physical address (PA) to main memory/cache
5. cache/memory sends data word to processor

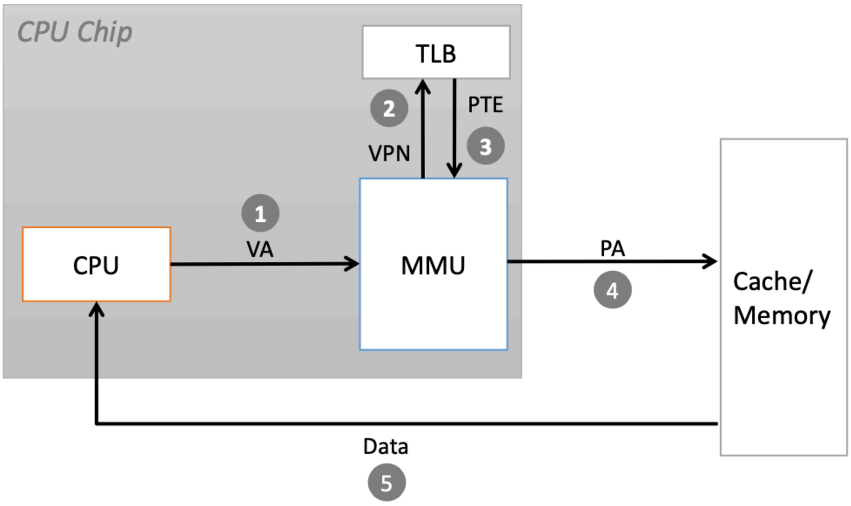


Figure 76: TLB Hit Address Translation Process

**TLB Miss** Notice that a TLB miss leads to an increase in latency compared to not having a TLB. But, fortunately, TLB misses are rare so that this is acceptable.

1. processor sends virtual address to MMU
2. MMU indexes TLB with VPN
3. MMU gets PTE and detects that valid bit not set → MMU requests PTE from main memory (or cache if physically indexed and tagged)
4. memory/cache delivers the PTE to the MMU and the TLB
5. MMU constructs the PA and sends it to the cache/memory

6. memory/cache delivers the word to the CPU

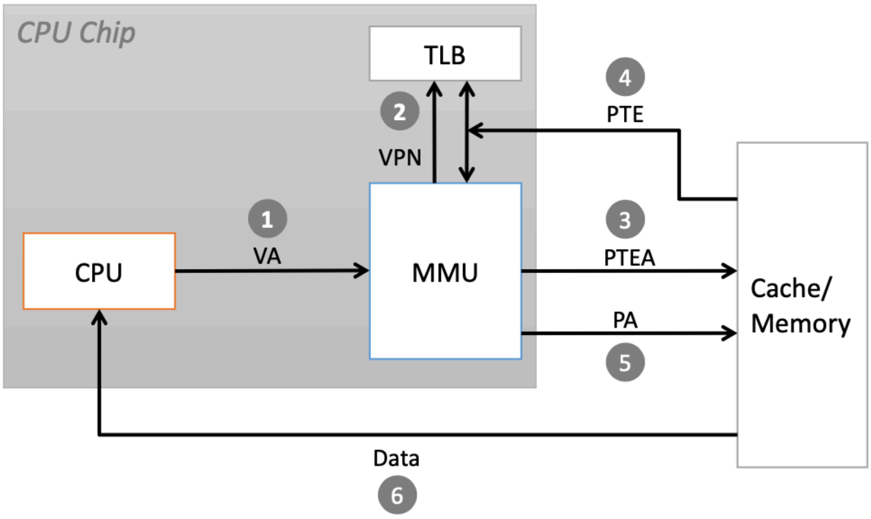


Figure 77: TLB Miss Address Translation Process

## 19.4 Example: Simple Memory System

To not only learn the theoretical underpinning of virtual memory but also really understand it by heart, we will look at how those different parts of the address translation look in an example system.

We consider a memory system with:

- 14-bit virtual addresses
- 12-bit physical addresses
- page size of 64 bytes (corresponding to 6 bits in the address)

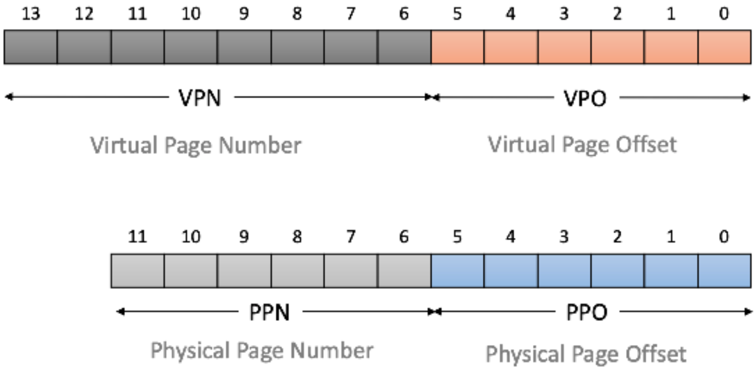


Figure 78: Simple Memory System Addresses

**Page Table** A page table would be indexed by the VPN and contain a PPN as well as metadata such as the valid bit.

| VPN | PPN | Valid |
|---|---|---|
| 00 | 28 | 1 |
| 01 | - | 0 |
| 02 | 33 | 1 |
| 03 | 02 | 1 |
| 04 | - | 0 |
| 05 | 16 | 1 |
| 06 | - | 0 |
| 07 | - | 0 |

| VPN | PPN | Valid |
|---|---|---|
| 08 | 13 | 1 |
| 09 | 17 | 1 |
| OA | 09 | 1 |
| OB | - | 0 |
| OC | - | 0 |
| OD | 2D | 1 |
| OE | 11 | 1 |
| OF | OD | 1 |

Figure 79: Example Page Table

**TLB** The TLB translates a VPN to a PPN. It is a cache indexed by the (lower) index bits of the VPN and tagged by the remaining upper bits of the VPN. Below is an example of a 4-way associative TLB cache with 4 sets.

| Set | Tag | PPN | Valid | Tag | PPN | Valid | Tag | PPN | Valid | Tag | PPN | Valid |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 03 | - | 0 | 09 | 0D | 1 | 00 | - | 0 | 07 | 02 | 1 |
| 1 | 03 | 2D | 1 | 02 | - | 0 | 04 | - | 0 | 0A | - | 0 |
| 2 | 02 | - | 0 | 08 | - | 0 | 06 | - | 0 | 03 | - | 0 |
| 3 | 07 | - | 0 | 03 | 0D | 1 | 0A | 34 | 1 | 02 | - | 0 |

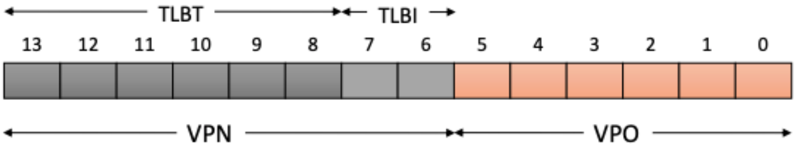


Figure 80: TLB Indexing

**Cache** If our cache is physically tagged, we consider the physical address and index with the corresponding cache index bits and then compare the tags from the physical page number to find a map. The below cache is physically tagged, has 16 lines/set and is direct mapped. Each block has a size of 4 bytes.

| Idx | Tag | Valid | B0 | B1 | B2 | B3 |
|---|---|---|---|---|---|---|
| 0 | 19 | 1 | 99 | 11 | 23 | 11 |
| 1 | 15 | 0 | - | - | - | - |
| 2 | $1B$ | 1 | 00 | 02 | 04 | 08 |
| 3 | 36 | 0 | - | - | - | - |
| 4 | 32 | 1 | 43 | 6D | 8 F | 09 |
| 5 | 0D | 1 | 36 | 72 | F0 | 1D |
| 6 | 31 | 0 | - | - | - | - |
| 7 | 16 | 1 | 11 | C2 | DF | 03 |

| Idx | Tag | Valid | B0 | B1 | B2 | B3 |
|---|---|---|---|---|---|---|
| 8 | 24 | 1 | $3A$ | 00 | 51 | 89 |
| 9 | $2D$ | 0 | - | - | - | - |
| A | 2D | 1 | 93 | 15 | DA | $3B$ |
| B | OB | 0 | - | - | - | - |
| C | 12 | 0 | - | - | - | - |
| D | 16 | 1 | 04 | 96 | 34 | 15 |
| E | 13 | 1 | 83 | 77 | $1B$ | D3 |
| F | 14 | 0 | - | $51-$ | - | - |

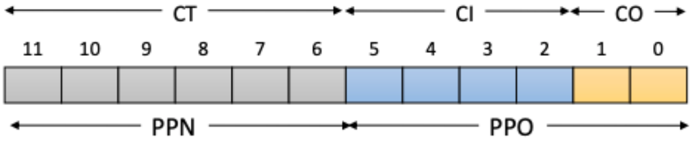


Figure 81: Cache Indexing

## 19.5 Multi-Level Paging

Consider this scenario resembling x86 to compute the size of the page table in bytes. Each page has size $4KB = 2^{12}$bytes. Our virtual addresses comprise 48bits. And each page table entry is of size 8bytes. We see that our VPN will comprise $48 - 12 = 36$bits. For each VPN, we must store one translation in the page table. So we have $2^{36} \cdot 8$bytes $= 2^{39}$bytes as the size for the page table. That corresponds to roughly 550GB. But no common modern consumer computer has that much RAM. The solution to not needing to have that much RAM is multi-level paging, which we will discuss now.

The idea of multi-level paging is to split the VPN into several parts and use those to index into different tables hierarchically. But not all of these tables exist/are allocated. With only some bits, we index into a first-level page table. Because we only have a fraction of the index bits, this page table is manageable in size. The entries of this page table then contain pointers to next-level page tables, which we then index with the next bits of the VPN. We can have $k$ such nested page tables.

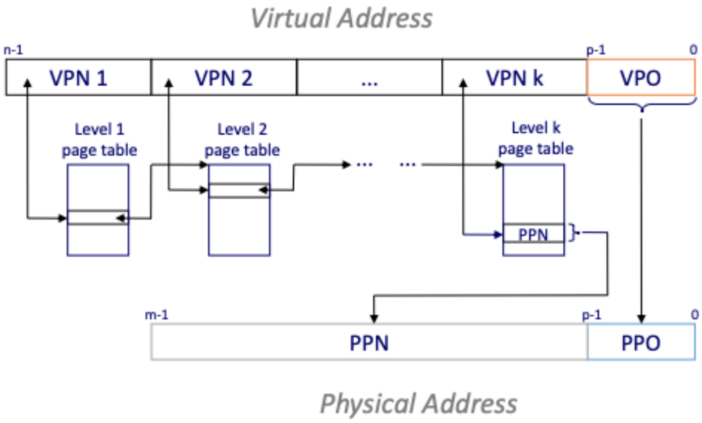


Figure 82: $k$-Level Paging Illustration

The key benefit comes from that not all page tables are actually allocated. Starting on the first level page table, for many addresses we only have invalid pointers, because that region of virtual memory has never been used/allocated. Thus we always have an entire first level page table in memory but all subsequent tables are only in memory after allocation, i.e., when actually needed. This noticeably saves space, while slightly increasing the latency due to multiple indexing.

But notice that if we would use this multi-level page table for each CPU memory request, our latency would increase significantly. For each page table indexing we would have to perform one request to physical memory. But as we use the TLB, we can bypass most of those requests so that we observe a performance gain in the end.

## 19.6 Case Study: Core i7 Virtual Memory System

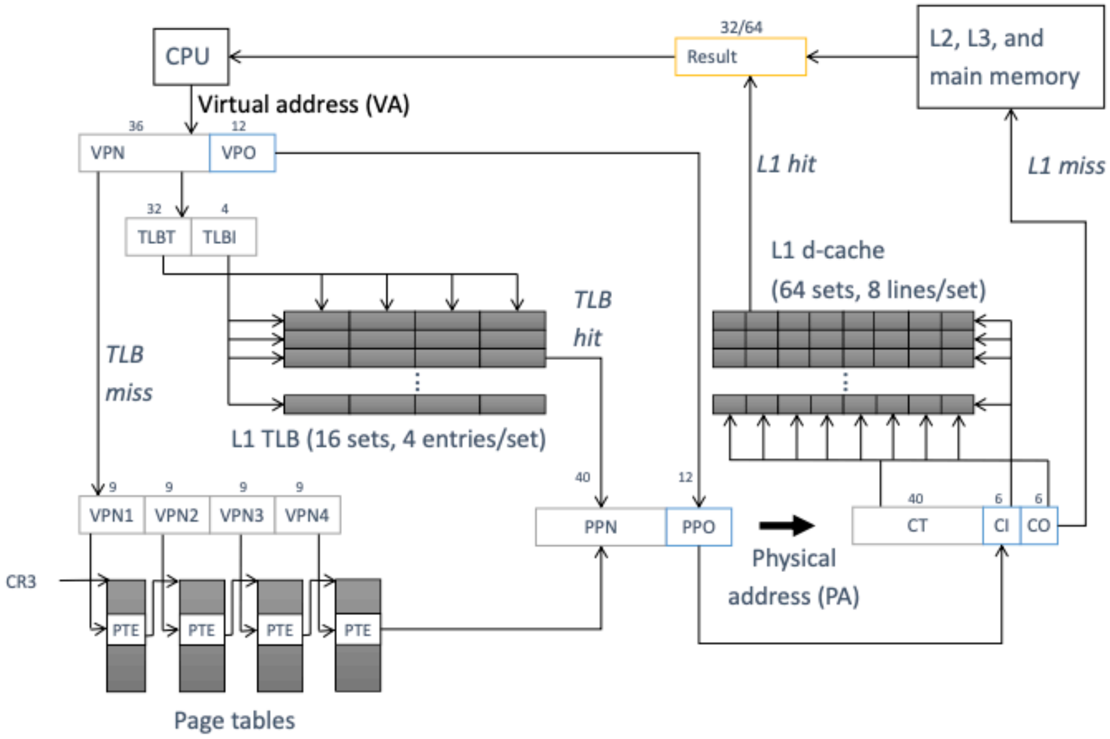


Figure 83: Simplified Core i7 Memory System

- CR3 is the x86 first level page table base register.
- 48 bit virtual addresses: Despite it being a 64-bit architecture, we have 48-bit canonical addresses that only allow 'free' usage of 48 bits. This already provides a huge 256 TB of addressable space. We do this, because our systems are not yet ready for full 64 bits as the mapping tables would be huge and we can't build that much DRAM yet anyway.
- 52 bit physical addresses of which 40 bits are used for the PPN.
- This increase from the virtual to the physical address size is useful if we want to run multiple OSs for virtualization on a server, for example
- Pages are 4KB in size. And one page tables is exactly one page, i.e., 4KB, in size. This means that one page table can store references to $2^{12}/2^{3} = 2^{9} = 512$ pages.

Entries of the TLB contain the information from the page table. Among it:

- V: valid bit

- TLBTag: cache tag
- PPN: translation of the address indicated by index & tag
- G: ”global” page according to PDE/PTE entry, indicates to not evict upon context switch between processes
- S: ”supervisor-only” according to PDE, PTE
- W: page is writable according to PDE, PTE
- D: PTE has already ben marked ”dirty”

The translation process happens like this:

1. partition VPN into TLBT and TLBI
2. check, whether PTE vor the VPN is in the cache set of TLBI
3. yes: check permission and build physical address
4. no: read PTE from memory and build physical address

Notice, that the TLB not only stores PTEs but also PDEs. This enables the memory system to skip some table indexing steps while resolving the address with the multi-level page table.

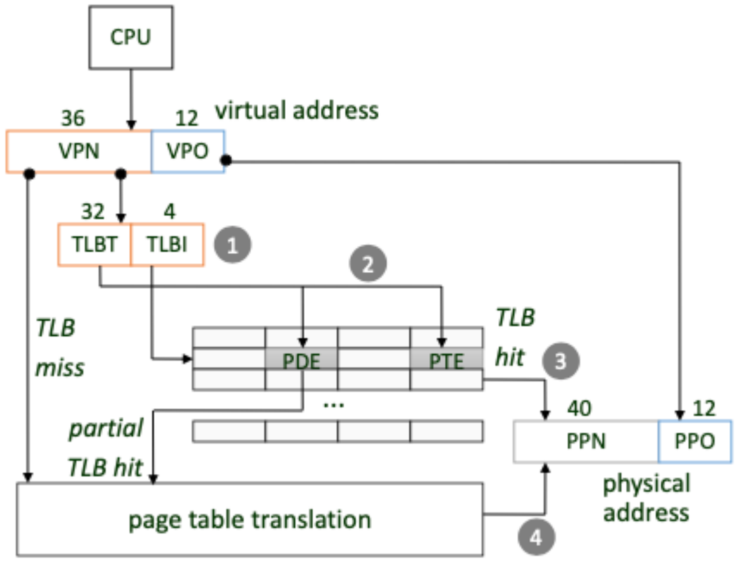


Figure 84: Core i7 TLB Translation

### 19.6.1 x86-64 Page Translation

Now, having understood the general setup of the x86 address translation setup, we look at how the different levels of page tables actually look in memory. We have 4 different levels of page tables with distinctive names: Page Map Table, Page Directory Pointer Table, Page Directory Table, and Page Table. Starting with the Page Directory Pointer Table, we may not only store a reference to a subsequent level page table but also a reference to an accordingly large page itself. This will be discussed in Large Pages in more detail.

Furthermore, each level page table specifies permission bits. The permission bits apply to all entries reachable from this page table entry. Thus, high-level restrictions apply even if on a lower level we don't specify them. The restrictions are valid for all covered virtual addresses.

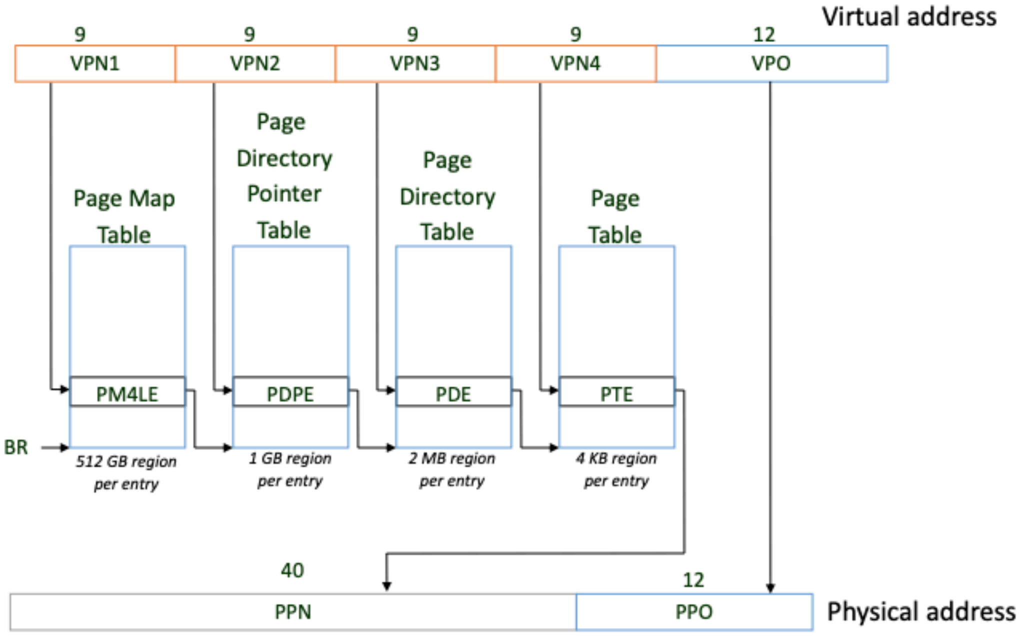
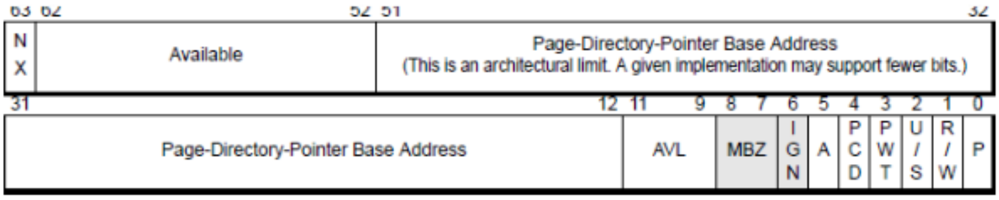


Figure 85: x86 Page Table Hierarchy

The entries of the different page table hierarchies will only be discussed in referencing other page tables up to 'Page Table'. Large pages are discussed later.

**Page Map Table** .



63 62
52 51
32
N X
Available
Page-Directory-Pointer Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31
12 11
9 8 7 6 5 4 3 2 1 0
Page-Directory-Pointer Base Address
AVL
MBZ
IGN
A
PCD
PWT
U/S
R/W
P

Figure 86: Page Map Table

- Page-Directory-Pointer Base Address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)
- Avail: Bits available to system programmers
- A: accessed (set by MMU, cleared by software)
- PCD: cache disabled (1) or enabled (0)
- PWT: write-through or write-back cache policy for this page table
- U/S: user or supervisor mode access

- R/W: read-only or read-write access
- P: page table is present in memory (1) or not (0)

**Page Directory Pointer Table** .

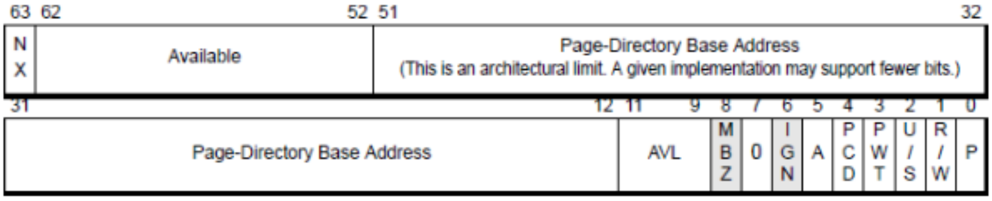

63
62
52
51
32
NX
Available
Page-Directory Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31
12
11
9
8
7
6
5
4
3
2
1
0
Page-Directory Base Address
AVL
MBZ
0
IGN
A
PCD
PWT
U/S
R/W
P

Figure 87: Page Map Table

- Page-Directory Base Address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)
- Avail: Bits available to system programmers
- A: accessed (set by MMU, cleared by software)
- PCD: cache disabled (1) or enabled (0)
- PWT: write-through or write-back cache policy for this page table
- U/S: user or supervisor mode access
- R/W: read-only or read-write access
- P: page table is present in memory (1) or not (0)

**Page Directory Table** .

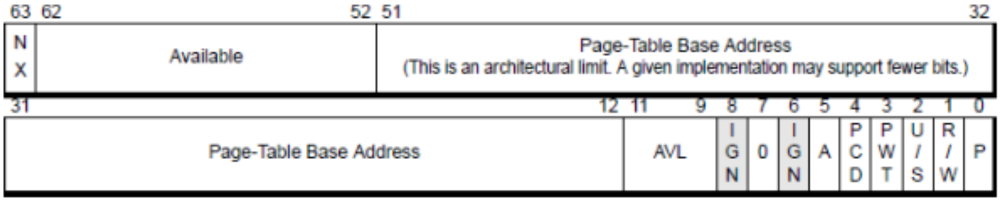


Figure 88: Page Map Table

- Page table physical Base Address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)
- Avail: Bits available to system programmers
- A: accessed (set by MMU, cleared by software)
- PCD: cache disabled (1) or enabled (0)
- PWT: write-through or write-back cache policy for this page table
- U/S: user or supervisor mode access

- R/W: read-only or read-write access
- P: page table is present in memory (1) or not (0)

**Page Table** .

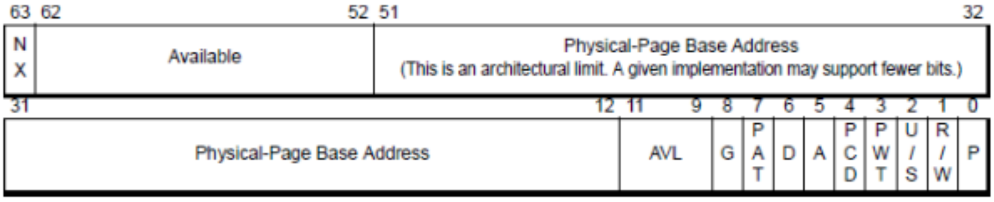

63
62
52
51
32
N X
Available
Physical-Page Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31
12
11
9
8
7
6
5
4
3
2
1
0
Physical-Page Base Address
AVL
G
P A T
D
A
P C D
P W T
U / S
R / W
P

Figure 89: Page Map Table

- Physical page base Address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)
- Avail: Bits available to system programmers
- G: global page (don't evict from TLB on task switch)
- PAT: page-attribute table (allow OS to control how individual pages are cached)
- D: dirty (set by MMU on writes)
- A: accessed (set by MMU, cleared by software)
- PCD: cache disabled (1) or enabled (0)
- PWT: write-through or write-back cache policy for this page table
- U/S: user or supervisor mode access
- R/W: read-only or read-write access
- P: page table is present in memory (1) or not (0)

### 19.6.2 x86 Paging Flow

**Page Tables in Memory, Page in Memory** .

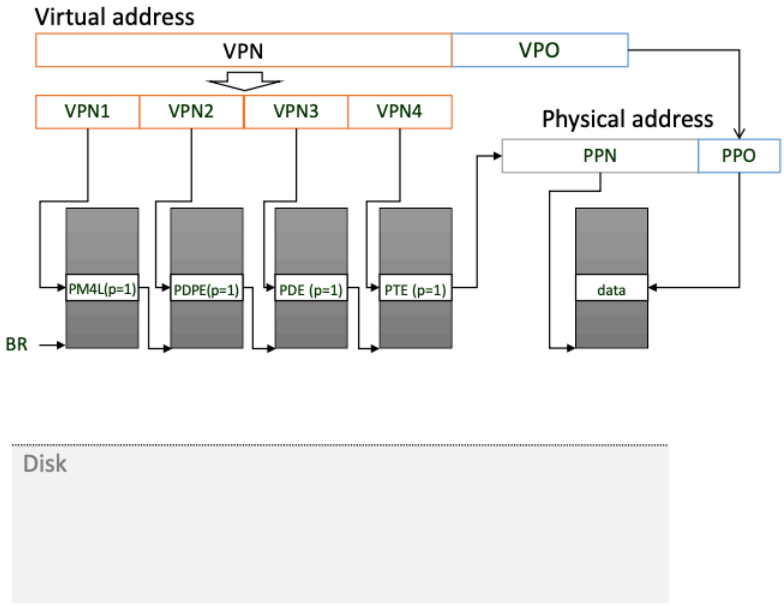


Figure 90: Flow with Page Tables in Memory and Page in Memory

MMU action:

- build physical address
- fetch data word

OS action: NONE

**Page Tables in Memory, Page not in Memory** .

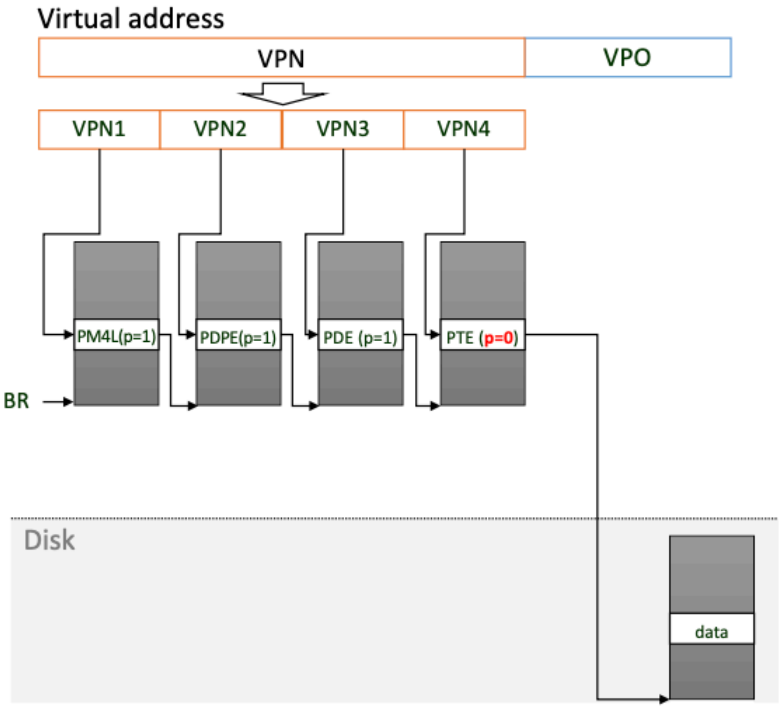


Figure 91: Flow with Page Tables in Memory and Page not in Memory

MMU action:

- page fault exception
- handler receives args: `%rip` of fault, VA of fault, fault type (non-present page, page-level protection)

OS action:

- check for legal VA
- read PTE through PT
- find free physical page
- read VP from disk into physical page
- adjust PTE to point to PP, set ‘p=1‘
- restart faulting instruction

**Page Tables not in Memory, Page not in Memory** .

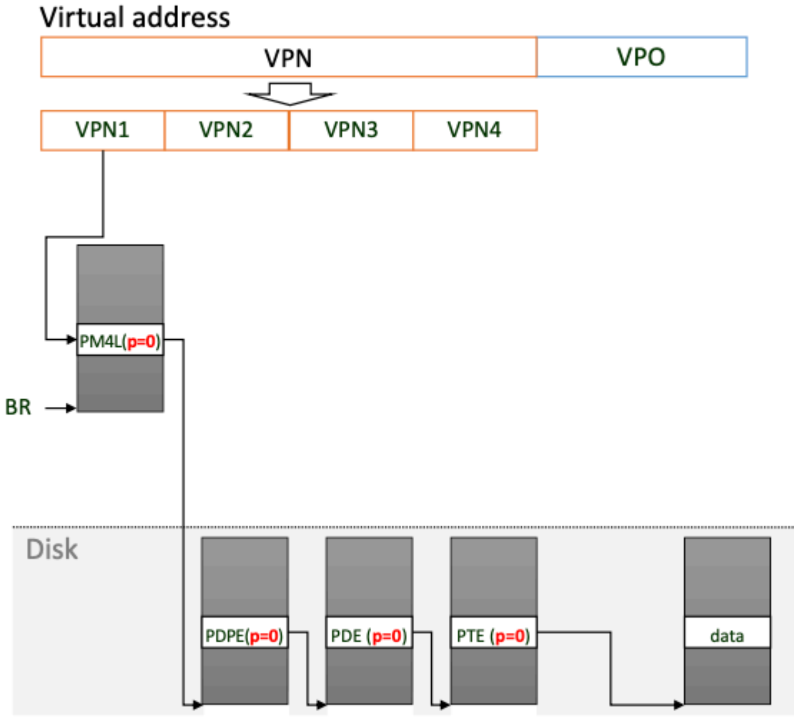


Figure 92: Flow with Page Tables not in Memory and Page not in Memory

MMU action:

- page fault

OS action:

- swap in page table
- restart faulting instruction by returning from handler

goto previous case → multiple disk reads

## 19.7 Caches as part of Virtual Memory

We have seen L1, L2, ... caches. But one important question is how to access those. We already discussed the general principle of dividing an address into offset bits, index bits, and tag bits. Here we discuss which bits we take in the context of virtual and physical addresses.

There are different options with different tradeoffs. For x86 we use the virtually indexed, physically tagged scheme as introduced below. That works, because the physical/virtual page offset is 12 bits in size. And of those we use 6 bits as offset and 6 bits as index. But because of this setup this is equivalent to physically indexed, physically tagged. We don't say so to make explicit that this enables us to already index into the cache before the address translation for the entire physical address is complete.

This means that caches are bound by the virtual/physical page offset. Specifically, the index and offset bits cannot exceed 12 bits. This prevents us from increasing associativity. But generally we don't want to do that anyway because it requires much more hardware.

### 19.7.1 Virtually Indexed, Virtually Tagged (VV)

This is also referred to as virtual addressing. It only uses the virtual address to index into the cache. The problem with this is that it may be that multiple processes use the same virtual address for different data (homonyms). To prevent errors resulting from different processes using the same cached data for different things, we have different possibilities. All of those are undesirable because of their overhead.

- flash the cache on context switches
- force non-overlapping address-space layouts
- tag VAs with address-space ID (ASID)

Even when fixing homonyms, we still have synonyms, i.e., different VAs referring to the same PA. Furthermore, write-back are also an issue. Because we don't know how that data is relevant for other processes with different VAs. The easy solution would be to make the cache write-through, a burden we don't want to put on the memory system.

Still, the clear benefit of VV is speed. Among all schemes, this is the fastest because we don't need to wait for any translation to complete. Today, it is rarely used on data caches. But it is still in use for instruction caches.

### 19.7.2 Virtually Indexed, Physically Tagged (VP)

Here, we index with bits from the VPO/PPO so that we only need to select the correct line once address translation through the MMU completed. Technically, we may also index with more bits from the virtual address than covered by the offset - but then we loose the benefits we wish to achieve with physical tagging.

The benefits are that we avoid issues from homonyms and synonyms, because we identify data by its physical (unique!) address. Still, the virtual indexing with the offset bits allows us to already start working with the cache before address translation is complete to combat the cost of the translation latency.

This is common for L1 data caches.

### 19.7.3 Physically Indexed, Virtually Tagged (PV)

This is very complex and rare. How it works can be deduced from the other instances, but it won't be discussed.

### 19.7.4 Physically Indexed, Physically Tagged (PP)

This only uses the physical addresses. Thus, we must always wait for the MMU to finish address translation before we can access the cache - which makes this the slowest addressing scheme. This is only used off-core for performance reasons. The main benefit is that we don't need to adhere to alignment for picking the index, offset, and tag bits.

PP caches are frequently used for L2 or L3 unified caches. It can be used with bus snooping to receive/supply DMA (direct memory access) data. Meaning, it observes main memory requests and intervenes when it has the data in cache or otherwise copies it after retrieval. This is part of some multiprocessing coherence protocols.

## 19.8 Write Buffers

Because memory is slow, store operations take a long time to complete. To avoid stalling the CPU, it usually contains write buffers. When the CPU wants to write, it queues the write in the buffer. And when wanting to read, it checks the buffer first, before going to memory. Thus, it does not have to wait for all memory accesses to complete.

## 19.9 Large Pages

x86 (and other architectures) also support more options for address translation. Specifically, there are large pages. With normal-sized x86 pages, we use the lower 12 bits as the VPO/PPO. With large pages, each page is 2 MB in size. With huge pages, it is 1 GB in size.

Technically, this is enabled by referencing to such a large/huge page from the page directory pointer table/page directory table instead of referencing the next level page table. As a programmer we can instruct the OS to use large/huge pages. For that, we may use `libhugetlbfs` as part of the `hugetlbfs` package - something not detailed in class. This changes some instructions in the link to the OS and replaces the implementations of malloc, calloc, and realloc.

### 19.9.1 x86 Large Pages

With large pages, we eliminate one level of the multi-level TLB and increase the page size. Instead of four we only have three levels of address translation. Thus, we need to be able to store different kind of entries as PDEs. The entry format we considered before contained one flag to indicate whether it is a page base address of a page-table base address. The flag has been 0 so far. If we now set it to one, we skip the last translation step.

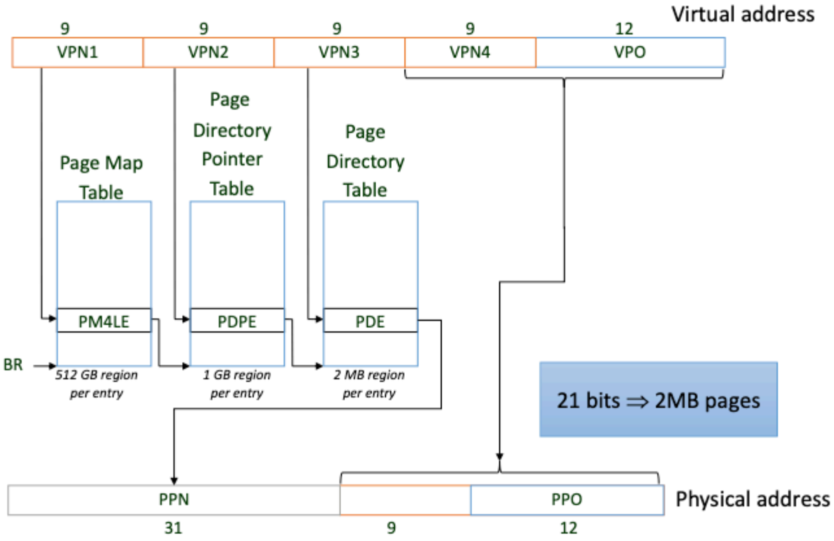


Figure 93: Large Pages Translation Flow

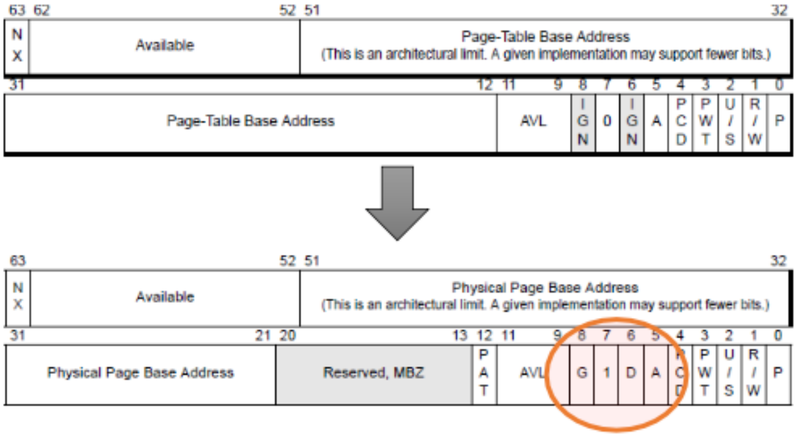

63 62
52 51
32
NX
Available
Page-Table Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31
12 11 9 8 7 6 5 4 3 2 1 0
Page-Table Base Address
AVL
IGN
0
IGN
A
PCD
PWT
U/S
R/W
P
63
52 51
32
NX
Available
Physical Page Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31
21 20
13 12 11 9 8 7 6 5 4 3 2 1 0
Physical Page Base Address
Reserved, MBZ
PAT
AVL
G
1
D
A
PCD
PWT
U/S
R/W
P

Figure 94: Large Pages Page Directory Entry

### 19.9.2 x86 Huge Pages

With huge pages, we eliminate another level of the TLB hierarchy. As with PDEs, we also have a flag for PDPE. If we set that to 1, we directly store a pointer to the page table.

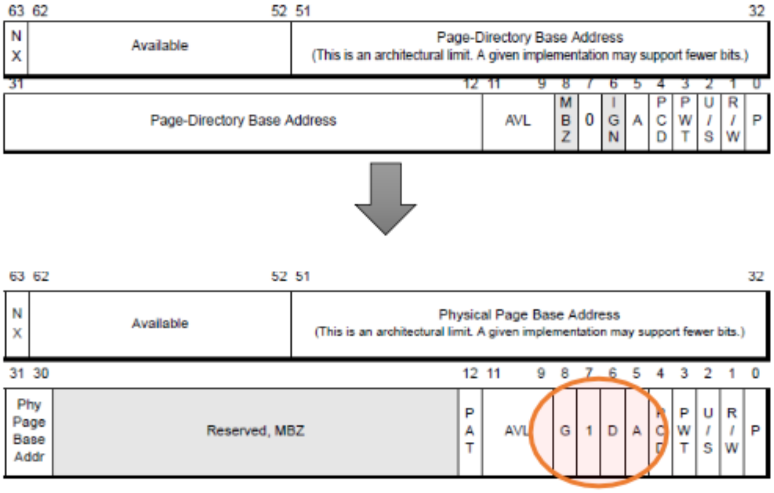

63 62
52 51
32
NX
Available
Page-Directory Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31
12 11 9 8 7 6 5 4 3 2 1 0
Page-Directory Base Address
AVL
MBZ
0
IGN
A
PCD
PWT
U/S
R/W
P
63 62
52 51
32
NX
Available
Physical Page Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31 30
12 11 9 8 7 6 5 4 3 2 1 0
Phy Page Base Addr
Reserved, MBZ
PAT
AVL
G
1
D
A
PCD
PWT
U/S
R/W
P

Figure 95: Huge Pages Page Directory Pointer Entry

## 19.10 Optimizing for the TLB

Earlier we discussed how to optimize code for cache performance. We can do the same to optimize our code for TLB performance, i.e., maximize TLB hits and prevent trashing in the TLB. The TLB is a cache after all.

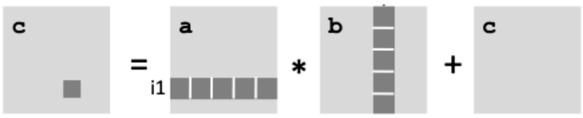


Figure 96: Matrix Multiplication Setup

If we say our L2 cache has size 16 MB $= 2^{24} = C = 2^{21}$ doubles. We need $3B^2 < C$ so that we fit in the cache. Thus, $B \approx 800$.

Now, additionally, we consider the TLB with 128 entries. Each page has size 4 KB and we assume that the dimension of the matrix $n$ is $> 4$ KB $= 512$ doubles. Thus, each row in the block is on a different page. If we have more rows than TLB entries, we incur TLB thrashing. So we want $3B \leq 128$, which leas to $B \approx 42$. And of course we also need $8B \leq 4096 = 4$ KB to ensure that one row of a block fits into one page. But this doesn't change our $B \approx 42$ estimate here.

An alternative approach is to copy the content of the four $B \times B$ blocks of our three matrices to a contiguous memory region. Because we have $\mathcal{O}(B^3)$ in the end, this $\mathcal{O}(B^2)$ operation doesn't affect asymptotic runtime. But it allows for larger $B$, because now we only need $3B^2 \frac{8}{4096} = 128$, i.e., the pages we copy to must fit in the TLB. This would result in $B \approx 148$.

# 20 Multiprocessing & Multicore

There are various reasons, why people started developing multiprocessors and multicore systems, including:

- one core/processor just didn't meet performance requirements
- some jobs can be well parallelized
- multiple processes can run at the same time

Since about 2003, multiprocessors turned from research prototypes and high-performance supercomputers to consumer-grade products. This was caused by what some say is the failing of Moore's law. I.e., hardware supported speedups weren't sufficient anymore to keep increasing single core performance. Despite a continuously increasing transistor density, we can't clock the CPUs higher due to power dissipation/head management problems.

**Power Wall** This illustrates the issue of further scaling individual cores due to heat dissipation issues. The power dissipation is

$$= P_{dyn} + P_{leak} + P_{short}$$

- $P_{dyn}$: dynamic power for switching the transistors
- $P_{leak}$: transistor leakage (increasing with smaller transistors)
- $P_{short}$: short circuit while switching

$P_{dyn} = CV^2 f$ ($C$: capacitance, $V$: supply voltage, $f$: clock frequency) can be kept stable with increasing frequencies by decreasing the voltage. But then still power leakage increases, leading to such an increase in power dissipation, which systems cannot handle.

- increase frequencies ⇒ more power dissipated ⇒ more cooling required
- decrease voltage ⇒ reduce dynamic power & increase static power leakage

This is a problem, because we've reached the practical power limit for cooling.

**Memory Wall** This describes the difference between the performance increase of compute/CPU and memory. CPU speed increases exponentially faster than memory speed. This means that memory has quickly become a bottleneck in computing.

**ILP Wall** This describes that CPUs are internally designed to execute instructions in parallel. But there are limits to how much one can optimize. At some point, programs just don't offer any more instruction level parallelism to data dependencies. But also the hardware to support larger context windows to consider more instructions to possibly execute in parallel is very expensive in hardware.

## 20.1 Symmetric Multiprocessing

Multiprocessing addresses the power, memory, ILP, ... walls. Because we want to manage power handling while the demands on system increases (higher I/O and network speeds) and we have to face the walls, we turn to multiprocessing to further increase performance.

Multiprocessing is what is done on all modern consumer-grade chips. Only some embedded systems and special-purpose applications still work work with single-core systems. Most systems are shared memory multiprocessors, which means a single physical address space is shared by all processors and the communication between processors happens through shared variables in the memory. Cache coherence (discussed later) is typically required for this to function properly.

One idea to implement multiprocessing is to attach multiple processors/cores to the same system bus (of course with necessary modifications). The functionality and practicality of this approach is driven by caches. Otherwise, the memory requests from the different CPUs could cause congestion at the memory interface. But with (shared) caches, this can be mediated.

## 20.2 Consistency and Coherence

**Coherence** is the concept that the values in different caches match each other. I.e., all processors see a coherent view of the memory. Memory coherence defines requirements for the observed behavior of reads and writes to the same memory location. This guarantees for instance, that if two writes are issued after each other by two processors that the second write persists.

**Consistency** is the concept that considers the order of changes to memory as seen by different processors. Memory consistency defines the behavior of reads and writes to different locations (as observed by other processors). Consistency determines whether and how memory operations can be reordered by the processor. To reason about program correctness, it is important to define consistency. There are many different memory consistency models.

Most modern machines are cache coherent. This means that the memory, although being split caches, behaves as a single memory array. The big advantage of cache coherency is the ease of programming. Some multiprocessing programming paradigms rely on cache coherence. The disadvantage is mostly on the side of the hardware designer as cache coherence is complex to implement and prone to bugs. Besides, such more complex systems are typically also slower than simpler designs.

To make working with consistency and coherence simpler, we introduce two terms:

- program order: This corresponds to the order in which a program on a processor appears to issue reads and writes. It is only concerned with the reads/writes local to one processor. This may not be the same order as in which the CPU issues the instruction to hardware/memory due to instruction reordering as part of out-of-order execution, write buffers, write-back caches, ...

- visibility order: This corresponds to the order in which all reads and writes (i.e., of all processors) are seen by some processor. We may have to distinguish between different visibility orders between different processors, depending on the memory model. Each processor always reads the value written by the last write in its visibility order.

## 20.3 Sequential Consistency

This is a very fundamental consistency model that makes things very easy for the programmer. But to guarantee this in hardware is very expensive. There are two core requirements:

1. Operations from a processor appear in program order as part of the visibility order.

2. Every processor's visibility order is the same (interleaving of all the program orders).

This works if processors issue memory operations in program order and do not perform optimizations through reordering and the memory system the totally orders all operations. One can understand this as memory operations taking effect globally atomically. With sequential consistency, all four memory operation orderings must be guaranteed to hold: Read after write, read after read, write after read, and write after write. Those cannot be changed.

The advantages are that its easy to understand and program for the programmer and easy to analyze automatically. However, on the other hand, it is hard to build a fast implementation, no component (compiler, processor, ...) may reorder reads/writes to optimize execution, and one cannot use write buffers but write-through instead.

When implementing sequential consistency one must make sure to prevent processors to read outdated data from caches etc. When a write is performed, this data must be propagated throughout the entire system or at least outdated copies must be invalidated.

Notice that while consistency and coherence are distinct, sequential consistency requires cache coherence. However, we may have cache coherence without sequential consistency. More general consistency models usually don't require (cache) coherence.

## 20.4 Relaxing Sequential Consistency

Sequential consistency is very hard to implement in hardware. Thus, we relax the conditions of SC to get a sensible hardware implementation while still offering a middle ground to programmability.

First, it is not generally guaranteed that all processors see the same visibility order/interleaving of program orders of the other processors. We have to assume that those orders are different.

With out-of-order execution, write buffers, compiler optimizations, ... may reorder program's memory requests if they don't show data dependencies within one thread/one processor instruction stream. More formally, there are many different reordering options:

- write-to-read/read-after-write reordering: `write; read;` → `read; write;`
- write-to-write/write-after-write reordering: `write1; write2;`→`write2; write1;`
- read-to-read/read-after-read reordering: `read1; read2;` → `read2; read1;`
- read-to-write/write-after-read reordering: `read; write;` → `write; read;`
- weak ordering: no implicit order guarantees at all, i.e., all memory reorderings allowed
- break write atomicity (no single visibility order across processes)

| | **Alpha** | **PA-RISC** | **ARM** | **Power** | **X86_32** | **X86_64** | **ia64** | **zSeries** |
|---|---|---|---|---|---|---|---|---|
| Reads after reads | ✓ | ✓ | ✓ | ✓ | | | ✓ | |
| Reads after writes | ✓ | ✓ | ✓ | ✓ | | | ✓ | |
| Writes after reads | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Writes after writes | ✓ | ✓ | ✓ | ✓ | | | ✓ | |
| Dependent reads | ✓ | | | | | | | |
| Instr. Fetch after write | ✓ | | ✓ | ✓ | ✓ | | ✓ | ✓ |

Figure 97: Allowed Reorderings for different ISAs

If we don't want to allow the by default possible reorderings, we can use the synchronization instructions provided by an ISA. Then we have more guarantees. That is necessary for some algorithms but comes at a runtime/performance cost. In x86 we use fences to prevent the reordering of instructions beyond their bound: `lfence` (load fence), `sfence` (store fence), and `mfence` (memory fence).

**Processor Consistency (PC)** is the standard consistency model for x86-64 machines and sometimes also called TSO (total store order). Earlier 32-bit x86 implementations only had weaker guarantees. This guarantees that all processors see the writes from one processor in the order they were issued. But different processors can see different interleavings of writes from different processors.

## 20.5 Barriers and Fences

Generally, hardware only has very weak guarantees because it makes software run much faster in case where the synchronization is not necessary. This means that consistency requirements must be explicitly stated when required as they are essential for the functioning of some algorithms. To get such stronger guarantees we use barriers (also called fences at times). Compiler barriers prevent the compiler from reordering instructions, i.e., can be used by the programmer to enforce an ordering. Then, memory barriers are introduced as instructions into the assembly (by the compiler) and prevent the CPU from reordering instructions across the bound of the barrier.

### 20.5.1 Compiler Barriers

The compiler is prevented from reordering visible loads and stores. But register accessed may still be reordered (as the are private). Compiler barriers are specific for different compilers. Common synchronization libraries use such barriers under the hood.

- gcc: `__asm__ __volatile__ ("":: "memory");`
- intel eec: `__memory_barrier()`
- microsoft visual c & c++: `__ReadWriteBarrier()`

### 20.5.2 Memory Barriers

Memory barriers are just specific instructions that tell the processor to not reorder beyond their bound. In x86 we have: `MFENCE` for memory barriers, `SFENCE` to prevent store reorderings, and `LFENCE` to prevent load reorderings.

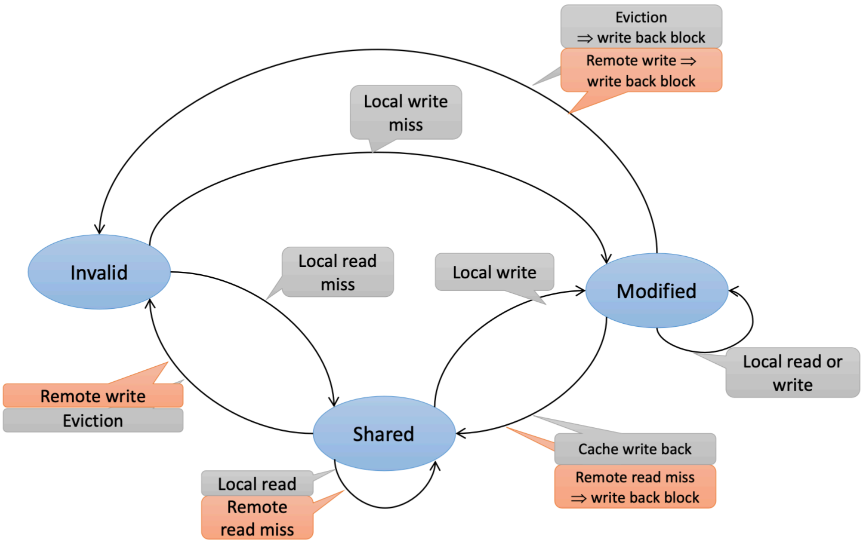
## 20.6 Cache Coherence with Snooping & MSI

When working with multiple processors, we can use snoopy caches to enable cache coherency. Snooping means that processors 'snoop' (i.e., listen in) on reads/writes from other processors. When a processor then sees that some other processor writes to a block, it invalidates the lines which cache this block in its own block.

The general idea of snooping is that each cache listens to the writes of the other caches and invalidates its own blocks if they are written to elsewhere. The simplest implementation then uses write-through caches to guarantee that any cache may then always access the newest data.

The MSI (Modified, Shared, Invalid) protocol allows for cache coherency without requiring caches to be write-through, which is very costly. It is the simplest protocol and works by maintaining states for each line in each cache. The three states are: Modified, Shared, and Invalid. Some line can only be dirty/modified in one cache at most. Upon events such as reads, writes, ... the cache logic must update the states and possibly evict/flash/write-back the cache's content. To do so it deals with its own processor reads/writes but also utilizes the remote bus reads and writes to respond to the actions of the other processors.

Important: This state is maintained for each cache block in each cache!



grey: processor initiated - orange: snoop-initated

Figure 98: MSI State Transitions

| | M | S | I |
|---|---|---|---|
| **M** | | | ✓ |
| **S** | | ✓ | ✓ |
| **I** | ✓ | ✓ | ✓ |

Figure 99: MSI State Invariants

But MSI clearly isn't perfect. This protocol requires sending many messages over the bus. Even many unnecessary messages. For instance, when we are the only processor to have something in the cache and then want to write. If we would know that nobody else has this in memory, we could proceed to writing directly.

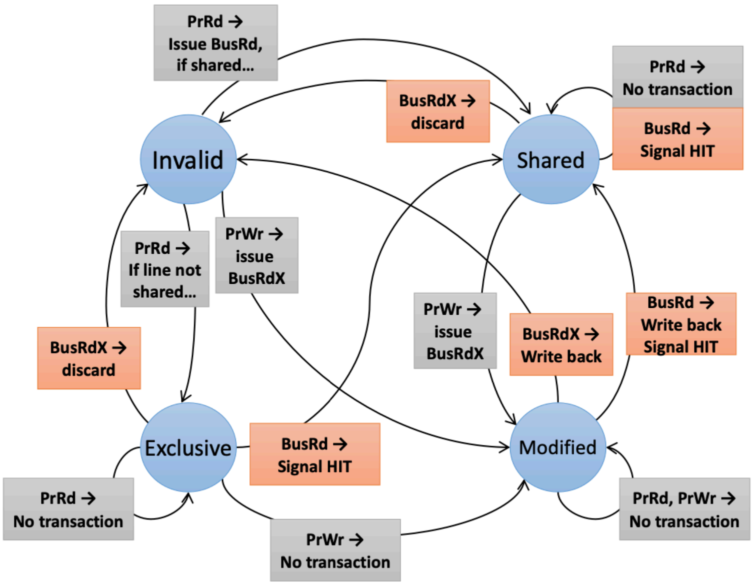
## 20.7 The MESI Cache Coherence Protocol

The MESI protocol is an extension of the MSI protocol by an additional state to optimize performance. It first appeared int the x86 Pentium processor. In addition to the existing states, we now have the state exclusive:

- modified: This is the only copy, it's dirty.
- exclusive: This is the only copy, it's clean.

- shared: This might be one of several copies, all clean.
- invalid

The cache functionality is enabled by a more complex state machine, HIT signals, and a bus. A HIT signal indicates to a remote processor that some block is in its cache. This is necessary so that caches know whether they can load a block as exclusive or not. The bus is used to tell other processors about reads and exclusive/eviction requirements.



PrRd: processor read (local) - PrWr: processor write (local)
BusRd: bus read (remote) - BusRdX: bus read exclusive (remote)

Figure 100: MESI State Transitions

| | M | E | S | I |
|---|---|---|---|---|
| **M** | | | | ✓ |
| **E** | | | | ✓ |
| **S** | | | ✓ | ✓ |
| **I** | ✓ | ✓ | ✓ | ✓ |

Figure 101: MESI State Invariants

But this still isn't optimal. Dirty data is always written through memory. We say that this is an 'invalidation-based' protocol without any cache-to-cache transfers. This only is good if the latency of the memory is noticeably smaller than the latency of a remote cache access. If that is not the case, we consider modifications of MESI: MOESI and MESIF. Those are quite similar but not the same. They mostly differ in MOESI being used by AMD and MESIF being used by Intel.

### 20.7.1 MOESI (AMD)

This introduces an additional state owned.

- modified: No other cached copies exist, local copy dirty

- owned: Multiple dirty copies exist (all consistent). This copy has sole right to modify line.
- exclusive: No other cached copies exist, local copy clean
- shared: Other cached copies exist (all consistent, but might be dirty or clean). One other copy might be able to write (owned state).
- invalid: Not in cache.

This allows to quickly performs reads for dirty cache lines without requiring another cache to write back to memory. Instead, the owned of that cache line provides the most up to date version of the data or tells the other caches to invalidate (in certain situations) when it modifies the line.

| | **M** | **O** | **E** | **S** | **I** |
|---|---|---|---|---|---|
| **M** | | | | | ✓ |
| **O** | | | | ✓ | ✓ |
| **E** | | | | | ✓ |
| **S** | | ✓ | | ✓ | ✓ |
| **I** | ✓ | ✓ | ✓ | ✓ | ✓ |

Figure 102: MOESI State Invariants

This approach will work well if the latency of remote caches is smaller than the latency of main memory.

### 20.7.2 MESIF (Intel)

Instead of an owned state, we now have the forward state. However, we don't allow multiple dirty copies as with MOESI. But we allow one cache to be in forward state so that other caches not necessarily have to go to memory to load some line.

- modified: No other cached copies exist, local copy dirty.
- exclusive: No other cached copies exist, local copy clean.
- shared: Other cached copies exist, all copies are clean. At most one other (clean) copy might be in Forward state.
- forward: As shared, but this is the designated responder for requests. Always the most recent cache to request line.
- invalid: Not in cache.

| | **M** | **E** | **S** | **I** | **F** |
|---|---|---|---|---|---|
| **M** | | | | ✓ | |
| **E** | | | | ✓ | |
| **S** | | | ✓ | ✓ | ✓ |
| **I** | ✓ | ✓ | ✓ | ✓ | ✓ |
| **F** | | | ✓ | ✓ | |

Figure 103: MOESI State Invariants

## 20.8 Multicore Synchronization

When writing software for multi-core setups, we don't manually include instructions for synchronization as discussed in Barriers and Fences. Instead, we use higher-level constructs. We now look at atomic operations mostly. There are instructions for such operations provided by the ISA while also being accessible to the programmer. Even higher level concepts such as locks are then implemented using atomics, fences, and memory bus locking.

- Atomic operations (on shared memory)

  e.g.: test-and-set, compare-and-swap (characterized by appearing atomically despite consisting of multiple steps)

  ordering constraints (specified in the memory model) - at least in java, we can't reorder across atomic operations

- Interprocessor interrupts (IPIs)

  This works by invoking an interrupt handler on a remote CPU. As this process is very slow it is rarely used (only by non-critical OS operations usually).

Notice that synchronization operations are generally very expensive. Atomic operations usually block the entire memory system so that no other memory access can proceed. Thus, we want to use those operations only if really necessary.

### 20.8.1 Test and Set

This is one of the simplest non-trivial atomic operations. It consists of two steps. While performing those two steps, the memory bus must be 'locked' so that nothing can interrupt this step or read the memory location in between.

1. Read a memory's location's value into a register.

2. Store "1" into the location.

TAS can be used to implement spinlocks. But due to the high cost of TAS and spinning, this is very intensive on resources and inefficient. Instead we can to test and test-and-set. That first does a cheaper plausibility test by doing a regular request to the cache before doing the cost TAS. Unless we have contention, the modified check in the outer loop is likely to succeed.

Listing 118: Spinlock with TAS

```
void acquire(int *lock) {
  while (TAS(lock) == 1);
}
void release(int *lock) {
  *lock = 0;
}
```

Listing 119: Spinlock Acquire with TTAS

```
void acquire( int *lock) {
  do {
    while (*lock == 1);
  } while (TAS(lock) == 1);
}
```

### 20.8.2 Compare and Swap

Listing 120: CAS Semantics

```
CAS(location, old, new) atomically {
  1. Load location into value
  2. If value == "old" then store "new" to location
  3. Return value
}
```

Implementing CAS requires bus locking or similar in the memory system. With CAS we can implement almost all wait-free data structures. Theoretically, it is more powerful than TAS as discussed in parallel programming.

CAS enables concurrent updates without locking. The general pattern to implement (wait-free) data structures adheres to ”read-copy-update”. Readers all read the same data structure. Writers create a copy, modify the copy, and then write back the copy. The old version is deleted as soon as nobody requires it anymore. Usually this is done by always referencing a pointer to the data structure, which enables simple replacement of large structures.

As part of the data structure we maintain a reference counter that stores how many references currently exist to a data structure. That is to ensure that we can safely remove old instances of the data structure after modifications when there are not references to such an old instance anymore.

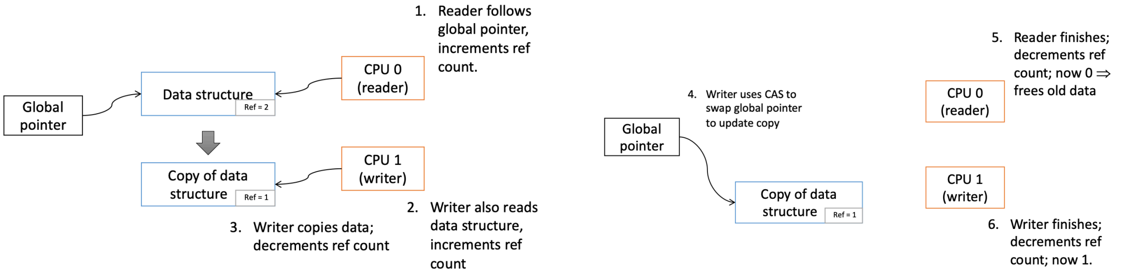

1. Reader follows global pointer, increments ref count.
Global pointer
Data structure
Ref = 2
CPU 0 (reader)
Copy of data structure
Ref = 1
CPU 1 (writer)
2. Writer also reads data structure, increments ref count
3. Writer copies data; decrements ref count
4. Writer uses CAS to swap global pointer to update copy
Global pointer
Copy of data structure
Ref = 1
5. Reader finishes; decrements ref count; now 0 ⇒ frees old data
CPU 0 (reader)
CPU 1 (writer)
6. Writer finishes; decrements ref count; now 1.

Figure 104: Wait-Free Data Structure with CAS

**Double CAS** `DCAS(loc1, old1, new1, loc2, old2, new2)` is done atomically. Meaning:

1. Compares two memory locations with different values.
2. If both match, each is updated with a different new value.
3. If not, existing values in the locations are loaded.

This is rarely implemented by hardware, because it is very prone to bugs and costly. It became apparent in past years that one can just as well use CAS cleverly. Specifically, we use CAS on a larger register. x86 supports CAS on up to 128 bits, for instance. Then we just pack the two data values in 128 bits an use normal CAS.

**ABA Problem** CAS detects when the location content differs. But when the content is the same, we perform the replacement. An issue arises if we change the data and then change back. With CAS we cannot detect that the system state has changed. It seems that everything is as it has been before. In Parallel Programming we discussed problems that may arise in a stack in great detail.

One ”solution” (that makes the problem very very unlikely) is to split the value into two parts: The original value and a with each operation monotonically increasing counter. With CAS we operate on both parts to modify our data and increase the pointer. If we change our data back to what it has been originally, it is still very very likely that the pointer is different and did not wrap around the exact amount to be what is was before.

Just as double CAS this is usually implemented with CAS on large registers.

### 20.8.3 x86 Synchronization

x86 has plenty synchronization operations/instructions. Among those:

- XCHG: atomic exchange of register and memory location (corresponds to TAS)
- LOCK prefix: executes the prefixed instruction with the (memory) bus locked

  LOCK ADD, LOCK BTS, ...
- LOCK XADD: atomic fetch-and-add
- CMPXCHG: 32-bit compare and swap
- CMPXCHG8B: 64-bit compare and swap
- CMPXCHG16B: 128-bit compare and swap (x86_64 only!)

CMPXCHG16B is not the same as DCAS. It only works on one memory location. Still, it is used to solve the ABA problem by using parts of the location for the counter and the other part for the data.

## 20.9 Simultaneous Multithreading

Despite our effort with caches, memory is still the main bottleneck for computers. All memory accesses still stall the processor. If reads need to be serviced from an off-core cache, this may also lead to increased latencies. Furthermore, this approach isn't scalable. With many processors, main memory and the caches may be constantly contended, leading to bad performance.

The core problem is that processors need to stall, while memory accesses are being served. With simultaneous multithreading, we try to utilize the processor's resources as much as possible by using them for something else while some process stalls. One idea pursued by out-of-order executing, superscalar processors, etc. is to use all possible instruction level parallelism (ILP). But there are limits to independence of instructions in a context window. The idea is to execute instructions form other threads. Those are independent instruction streams and thus can be freely executed. This is possible because we have many resources in a processor, which some thread doesn't need when waiting for some memory access to be served.

We start labeling instructions with an id for the thread, so that we can execute multiple threads in an interleaved fashion in one processor. Multithreaded CPUs often appear as multiple CPUs to the operation system.

- Coarse-grained multithreading: Whenever there is a long-latency memory stall, we switch between threads. (Only issuing instructions for one thread each cycle.)
- Fine-grained multithreading: On each cycle, we switch between threads. (Only issuing instructions for one thread each cycle.)
- Simultaneous Multithreading (SMT): On a superscalar machine, we also issue instructions from multiple threads in one cycle to maximize resource utilization.

For individual threads, this may very well lead to worse performance. Maybe they don't execute again as soon as the memory access is served. And other threads may evict cache blocks relevant for them from caches. Anyway, we typically get a 10 to 20 % performance increase. And we do this, because compared to other approaches it is very cheap to implement because not much additional hardware is required.

For scientific computing, we often disable such functionality, because it leads to less predictable performance. But memory-intensive applications such as web servers exhibit major performance improvements through the use of multithreading.

## 20.10 Non-Uniform Memory Access (NUMA)

Another issue with the current setup is memory bus contention. Despite having more processors, we might not be able to scale the amount of cache at the same rate due to hardware constraints (manufacturing & indexing/index bits limited). Thus, much more traffic happens on the memory bus. When this get to high, performance drops.

As a solution, we use distributed memory architectures. Instead of having one shared memory and I/O bus, modern systems use distributed memory architectures. We divide main memory into chunks and assign some subset of CPUs some part of main memory. This subset of processors has exclusive access to this region through a separate memory management unit. Those subsets of processors are then connected through ’networking’ interconnects instead of a single bus. This means that not all processor groups must be connected to all other processor groups. If a processor now needs to access memory that is managed by a different processor group, message passing is used to get the data. Compared to the earlier version, this approach is very scalable and can support hundrets and thousands of CPUs.

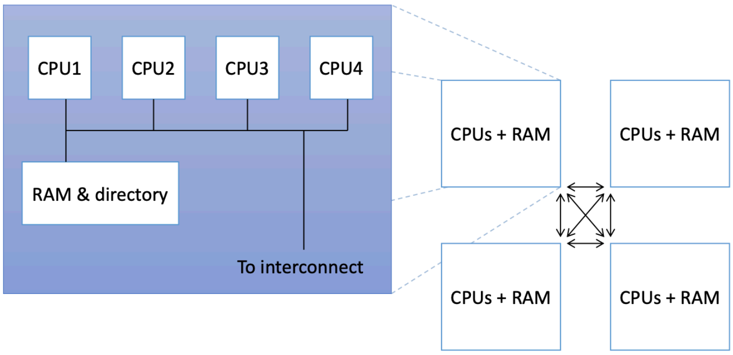


Figure 105: NUMA System Layout

This approach removes the bottleneck, because the individual processor groups now have separate access paths to memory. They each manage their own bank. The ’networking’ interconnect is not a bus! Instead of providing signals to all processor groups, messages are exchanged explicitly between two processor groups. Usually, one such processor group corresponds to a physical CPU socket in a machine. The messages sent indicate read requests, write requests, responses about cache content, cache invalidation instructions, ...

Not all processors need to be connected to all other processors. There may be various intermediate processors, which need to forward communication requests. Still, all memory is globally accessible. But the memory managed by the own processor group can of course be accessed faster. One doesn’t have to go over an interconnect and exchange messages.

### 20.10.1 NUMA Cache Coherence

Maintaining cache coherency is more complicated with NUMA. A cache can’t simply snoop the memory bus to get all updates, because we don’t have a bus anymore. One thing we can do is to emulate a bus using the interconnect (**bus emulation**). Each node always sends a message to all other nodes (processor groups), which must acknowledge the receipt. Implementing this in hardware is very complicated.

An alternative approach is to use a **Cache Directory**. Such a directory extends what is stored for each memory location in a node for the part of memory managed by that node. In addition to the data, now also bits indicating presence of this line in some other node's cache in addition to an owner entry that indicates which node currently has the latest modified version of this line.

This data is usually maintained in DRAM but could also be stored in the last-level cache. This shows a major disadvantage: We loose a noticeably amount of memory capacity as it is used to maintain the additional meta-data. (Some specific system loose up to 33% of their L3 cache capacity.)

This is useful if lines are not widely shared. Because if they are, we don't gain much from this additional metadata, because we quite often just need to send a message to all other nodes anyway to instruct them to flush. And the more nodes we have, the higher the benefit is too. Because then the network bandwidth savings are proportionally higher compared to the number of bits. This is because the distance to other nodes increases.

### 20.10.2 Performance Implications

From this distributed memory structure follows that access latencies are non-uniform (as the name NUMA implies). With increasing distance the latency increases.

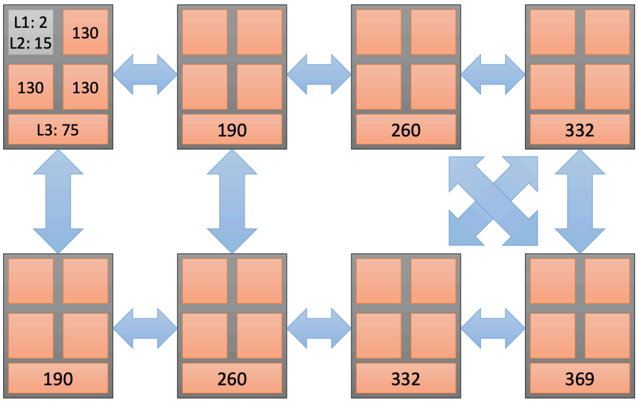


Figure 106: Example NUMA Cache Access Latencies

One thing that has a big negative impact on performance is false sharing. This refers to two threads on different nodes writing to different parts of one cache line. On a single CPU this is clearly no problem. On different nodes, the cache line needs to be transferred between the two nodes on every write - although the change of the one thread is irrelevant to the other.

## 20.11 MCS Lock

Very simple lock implementations use spin locks. For a few processors this works well. But as the system scales, contention increases and the approach leads to a lot of wasted cycles spent spinning. MCS locks avoid spinning on remote data (which is bad for memory performance due to long-latency read/write processes). Instead one only spins on local data, which does not impact overall system performance, and wakes up once notified that we may access the external data.

To reiterate: The issue with locks by means of atomic operations is that the cache line containing the lock is a hot spot. It is continuously invalidated as every processor tries to acquire it. This leads to an overhead and much much interconnect traffic. To reduce the interconnect traffic, we introduce a queue so that processors don't compete for the cache line but are notified when they can access the lock. Meanwhile, they only need to check a local flag.

Listing 121: MCS Lock Pseudocode

```
struct qnode {
  struct qnode *next;
  int locked;
};
typedef struct qnode lock_t;

void acquire(lock_t *lock, struct qnode *local) {
  local->next = NULL;
  struct qnode *prev = XCHG(lock, local);
  if (prev) { // queue was non-empty
    local->locked = 1;
    prev->next = local;
    while (local->locked)
      ; // spin
  }
}

void release(lock_t *lock, struct qnode *local) {
  if (local->next == NULL) {
    if ( CAS(lock, local, NULL) ) {
      return;
    }
    while (local->next == NULL)
      ; // spin
  }
  local->next->locked = 0;
}
```

As soon as systems scale beyond a few cores, the MCS lock typically outperforms a simple spinlock.

# 21 Devices & I/O

Devices are pieces hardware connected to our computer and visible to the software. This is especially relevant for OS programmers and the operating system. Below we will discuss how devices integrated in a computer, its OS, how they communicate, ...

## 21.1 Device Registers

A processor/computer interacts with devices through device registers. Those are characterized by having memory addresses. We write to and read from them. This corresponds to literally writing to and reading from (main) memory. The device then also has access to those locations. Through those memory locations, the device and processor communicate.

When the CPU loads from device registers, it obtain status information or reads input data. When it stores something to device registers it sets the device state and configuration, writes output data, or resets the device.

For the specifics of how those memory addresses work and how we access them, there are two options: Memory mapped addresses and I/O instructions.

- When using memory mapped addresses, the registers appear as regular memory location in main memory. We can interact with those addresses/the device through `movb`, `movw`, ...
- With I/O instructions, there is an entirely different address space (16 bit on x86). Today, this is typically only used for older I/O devices. To access this separate address space (with separate memory), we must use special instructions: `inb`, `outb`, ...

Additionally, we often have (separate) interrupt lines with which the device can 'notify' the processor that some event took place, i.e., that some transfer is complete (considering Direct Memory Access discussed below). This lets us avoid polling in driver software, which of course frees up processor resources for other tasks.

When working with a device, we typically have a base address for the memory-mapped I/O connection (provided by h/w specifications of the processor or OS or ...) and then access the relevant parts of that memory region through offsets. What is located where in this region is specified by device manual/technical specifications.

| Addr. | Name | Description | Notes |
|---|---|---|---|
| 0 | RBR | Receive Buffer Register (read only) | DLAB=0 |
| 0 | THR | Transmit Holding Register (write only) | DLAB=0 |
| 1 | IER | Interrupt Enable Register | DLAB=0 |
| 2 | IIR | Interrupt Identification Register (read only) | |
| 2 | FCR | FIFO Control Register (write only) | |
| 3 | LCR | Line Control Register | |
| 4 | MCR | MODEM Control Register | |
| 5 | LSR | Line Status Register | |
| 6 | MSR | MODEM Status Register | |
| 7 | SCR | Scratch Register | DLAB=1 |
| 0 | DLL | Divisor Latch (LSB) | DLAB=1 |
| 1 | DLM | Divisor Latch (MSB) | |

DLAB is the 7th bit of the LCR Register

Figure 107: Example - UART Memory Mapped Region

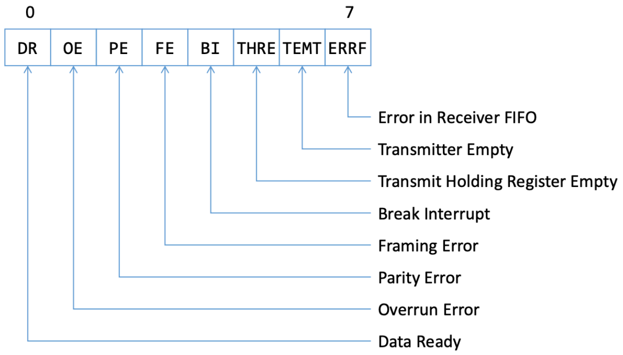


Figure 108: Example - UART Line Status Register content

One major challenge with memory-mapped I/O is: Where do the memory addresses come from? If devices just write to arbitrary locations, this could screw up data. Also the processor needs to know where in memory to communicate with the device. For complex devices, discoverable busses (such as PCI) solve this challenge. They manage physical address allocation (with the OS) for the devices, ... For simple devices, for example UART, we often have some specific registers designated from the chip manufacturer to correspond to the region for the UART pins.

As already mentioned, a device an OS communicate through memory. But there are different events with a specific interpretation in regards to changes in the memory content and interrupts:

- OS writing a device register ⇒ device can read that register
- OS reading a device register ⇒ OS gets device data, may trigger device
- device writing a device register ⇒ no direct action, OS may poll
- device requesting interrupt ⇒ CPU notified
- shared memory (DMA), later

  ⇒ CPU writes, DMA reads

  ⇒ DMA writes, CPU reads

Notice: Although we interact through memory, those locations don't behave as memory would usually. The content of registers may change without writes from the CPU, which causes issues with caching (discussed in the next section). Similarly, writes to registers don't simply store data. Those writes can trigger actions on the device, ...

Another remark: When we write software for devices, we often end up in a situation, where it appears to the compiler that we read from memory locations which nobody writes to. It then may remove entire sections of code. But this is of course not what we want, because the memory-mapped registers are volatile - their content could be changed by the device. Thus, we have to declare used variables in C as `volatile`, which tells the compiler, that the registers are volatile and the code can not be removed.

## 21.2 Dealing with Caches

When the device writes to the registers, the cache becomes invalid. So we can't serve memory reads from the cache. Using write-back caches causes problems, because we can't predict when some line will be written to main memory and then read by the device. Read and writes may not be combined in one cache line. If the processor writes to a line and then write back, it may overwrite what the device has put in some other registers in main memory. We loose data.

It follows that device register accesses must bypass the cache. This is done by setting the ”no cache” flat on the page table entries.

## 21.3 Device Drivers

The basic purpose of device drivers is clear. The bits sent by the device are, well, just bits. By itself, we don't learn much from the control registers. The device driver implements the specification of what the different control bits, status bits, ... mean. Then, they allow higher-level calls to the device, which they then translate into bits to put in main memory to instruct the device to do something or to return data from the device.

To implement a driver, it becomes clear that the driver always needs to know the state of the device to know how to interpret the currently set bits and know what the device can be instructed to do next. One can understand the device and the device driver to contain state machines, which must be in synchronized. Events then signal both state machines to switch to a new state. So that we can work with a newly connected device, the first thing a device driver should do is to reset the device to a known state.

## 21.4 Direct Memory Access (DMA)

So far we discussed how the OS/computer and some device interact through some memory locations. But in addition to such bits used for control and status information we have DMA in modern systems. The reason is that for many peripherals (such as GPUs) we need to transfer a lot of information. And having the CPU copy data from the control/status registers to some other location takes up a lot of valuable compute time.

Instead, with DMA, we enable the device to put the data somewhere in main memory itself so that the processor can work on other things. Once the transfer is complete, the device typically issues only one interrupt in the end (compared to potently one interrupt for each set of bits) to let the processor know. DMA can not only be used to transfer large amounts of data. It can also be used to transfer control instruction from the OS/user application to the device.

DMA benefits:

- save CPU clock cycles
- prevent CPU cache pollution
- higher-performance: CPU and device work in parallel
- save memory bandwidth

To let the device bypass the CPU, we need a DMA controller. The DMA controller makes sure to put the data at the intended location in main memory as instructed by some program. It then provides a bridge between the connected device and main memory and issues a transfer complete interrupt to the CPU once the transfer is complete.

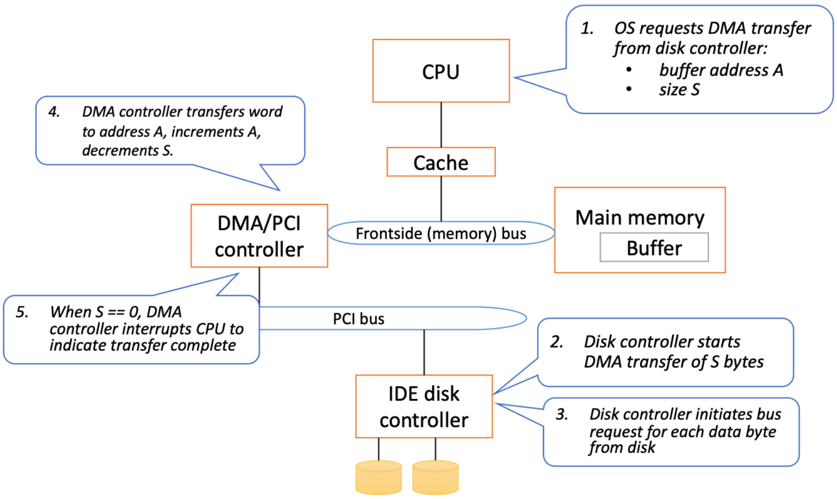

CPU
1. OS requests DMA transfer from disk controller:
• buffer address A
• size S
4. DMA controller transfers word to address A, increments A, decrements S.
Cache
DMA/PCI controller
Frontside (memory) bus
Main memory
Buffer
5. When S == 0, DMA controller interrupts CPU to indicate transfer complete
PCI bus
2. Disk controller starts DMA transfer of S bytes
IDE disk controller
3. Disk controller initiates bus request for each data byte from disk

Figure 109: Legacy DMA Controller Setup from 1986

With a DMA data transfer, the CPU caches become inconsistent with main memory. To solve this, we have three main possibilities:

- Affected addresses can be set as non-cachable by the DMA controller. But this is a large performance hit as the CPU likely wants to process the data afterwards.
- Caches can 'snoop' the DMAC bus transactions. Unfortunately, this does not scale well to large systems.
- But what is done in practice mostly: The OS can explicitly flush/invalidate cached regions. Thus, we see that cache management is an important part of device drivers!

On x86 PC hardware, PCI-based DMA transfers (PCI is introduced later and used for basically all 'modern' devices) are cache coherent with CPU caches. But because this causes lots of overhead, this functionality is not part of most other ISAs. On other systems we must:

- DMA reads (device reads from memory)
  - before: CPU should flush the cache for that data so that main memory is up to date
  - after: CPU should invalidate the cache for that address so that caches don't hold values of the memory content
- DMA writes (device writes to memory)
  - before: CPU should flush or invalidate so that the cache does not contain to be outdated values
  - after: CPU should invalidate so that the newly transmitted data is fetched next

As with the memory-mapped I/O before, the device works directly on physical memory. But the user and OS mostly work with virtual addresses. Thus, the OS and device drivers must convert virtual addresses to physical addresses for correct mappings. Furthermore, it may be that a process requests a device to transfer a lot of data into its virtual address space. But those addresses then may span multiple pages, which are not guaranteed to be contiguously in memory. For such cases, scatter-gather DMA controllers exist. Those take a list of addresses to put the transferred data in. In newer systems this is often done with an IOMMU. It works similar to the MMU as it translates virtual addresses to physical addresses for writing to continuous blocks of virtual memory. The IOMMU must be programmed and watched by the OS to match the MMU state!

Remember: DMA may be used to trasmit data and control information!

## 21.5 Buffer Rings and Description Rings

When we use Direct Memory Access, there needs to be a way for the device and the program/driver to exchange information in a controlled manner. The data needs to be saved in a structured format in memory. This is where buffer/descriptor rings come in. They are used as a common scheme to store data. But notice that there are also other ways. For instance, a linked list could be used - but we would call it a descriptor ring in chain mode (later more).

In any case, the device gets assigned some memory region. This assignment happens through PCI (more below). Furthermore, we usually have one of the below explained rings for transmission (OS → device) and one ring for receiving (device → OS).

Buffer/descriptor rings are circular buffers. The ring's elements are contiguous in memory. If we have a buffer ring, the ring's element contain the data of interest. If we have a descriptor ring, the ring's elements contain pointers to the actual data and meta-data. Today, we usually have descriptor rings.

The OS stores the tail pointer and the device stores the head pointer. Whatever comes after the tail pointer is said to be owned by the OS and what whatever comes after the head pointer is said to be owned by the device. For each entry it is stored in a meta-data flag whether it is currently OS- or device-owned. Through this flag, the OS and device can also detect when they have run up to the other pointer.

- Underrun: The receiver is faster than the sender and must wait for additional packages to arrive. If the receiver and has underrun, it may signal the sender to interrupt it once there is data again, to avoid polling.

- Overrun: The sender is faster than the receiver and there isn't space for more messages. If the device is the sender, it then usually starts to discard packages. In a status register, it can signal the processor that it lost some packages. The CPU is a sender, it usually waits to continue writing data into the buffer.

## 21.6 Example - Network Interface Card (NIC)

We won't provide details here. This is more to enable a better understanding how DMA is used in real-world applications. Specifically, this shows some selected information on the DEC "Tulip" - an ethernet adaptor.

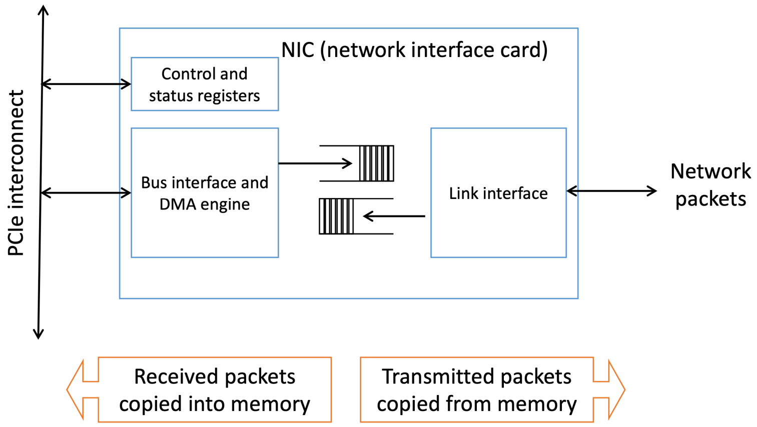

PCIe interconnect
NIC (network interface card)
Control and status registers
Bus interface and DMA engine
Link interface
Network packets
Received packets copied into memory
Transmitted packets copied from memory

Figure 110: NIC and DMA

| Register | Description | Address offset |
|---|---|---|
| CSRO | Bus mode | 0x00 |
| CSR1 | Transmit poll demand | 0x08 |
| CSR2 | Receive poll demand | 0x10 |
| CSR3 | Receive list base address | 0x18 |
| CSR4 | Transmit list base address | 0x28 |
| CSR5 | Status | 0x28 |
| CSR6 | Operation mode | 0x30 |
| CSR7 | Interrupt enable | 0x38 |
| CSR8 | Missed frames and overflow counter | 0x40 |
| CSR9 | Boot ROM, serial ROM, and MII management | 0x48 |
| CSR10 | Boot ROM programming address | 0x50 |
| CSR11 | General-purpose timer | 0x58 |
| CSR12 | General-purpose port | 0x60 |
| CSR15 | Watchdog timer | 0x78 |

Figure 111: Memory Mapped Control and Status Registers

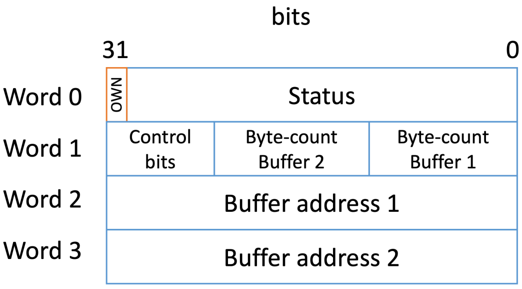


Figure 112: Descriptors of Descriptor Ring

## 21.7 Device Initialization

We already mentioned that the device and its driver can be interpreted to contain state machines, which must be kept in sync. This is done by putting them in a known state initially. This is called resetting. The OS resets the device once connected. The device's manual will describe a way to do so.

The OS/driver must:

- wait for the hw to become responsive
- stop the drive doing anything, i.e., let it run idle
- create all necessary shared data structures such as ring buffers, ...
- write the control registers of the device to have it start working and inform it about the memory regions to use (allocated in last step) if it uses DMA

## 21.8 I/O State Machines

### 21.8.1 hardware

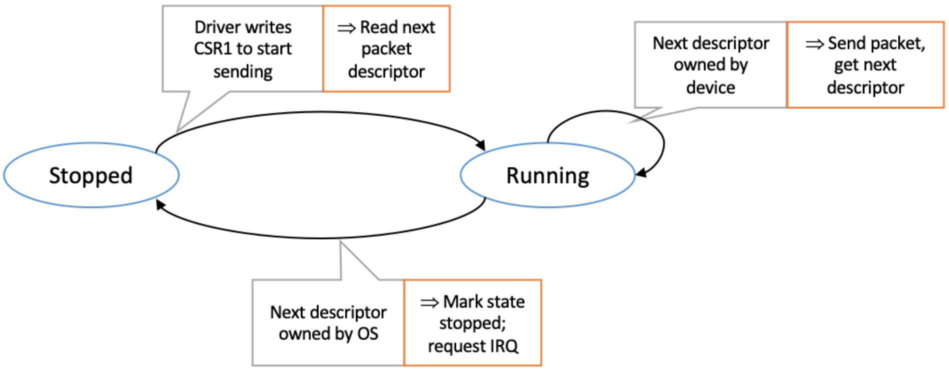

Driver writes CSR1 to start sending
⇒ Read next packet descriptor
Next descriptor owned by device
⇒ Send packet, get next descriptor
Stopped
Running
Next descriptor owned by OS
⇒ Mark state stopped; request IRQ

Figure 113: HW - Sending Packets

### 21.8.2 software

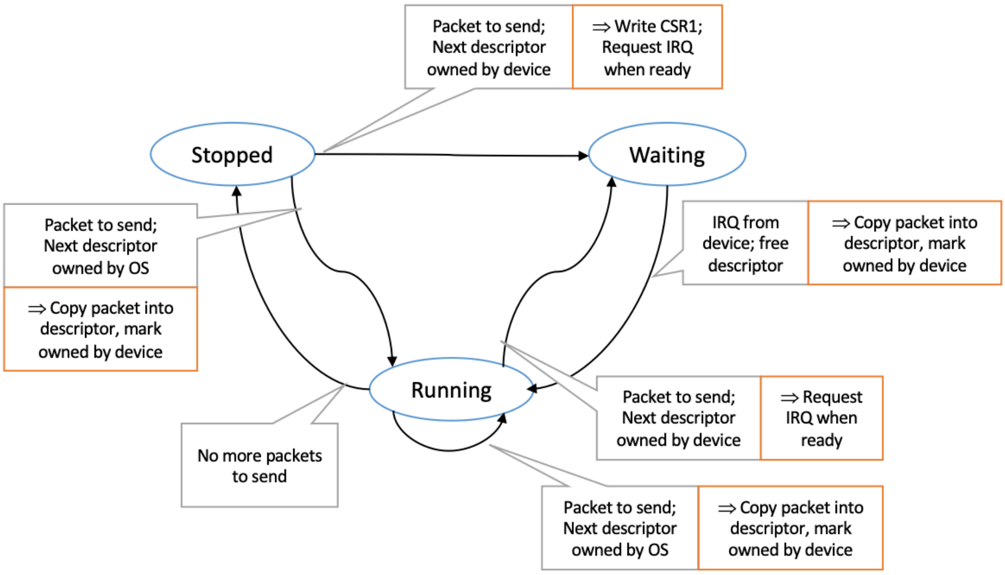

Packet to send; Next descriptor owned by device
⇒ Write CSR1; Request IRQ when ready
Stopped
Waiting
Packet to send; Next descriptor owned by OS
⇒ Copy packet into descriptor, mark owned by device
IRQ from device; free descriptor
⇒ Copy packet into descriptor, mark owned by device
Running
Packet to send; Next descriptor owned by device
⇒ Request IRQ when ready
No more packets to send
Packet to send; Next descriptor owned by OS
⇒ Copy packet into descriptor, mark owned by device

Figure 114: SW - Sending Packets

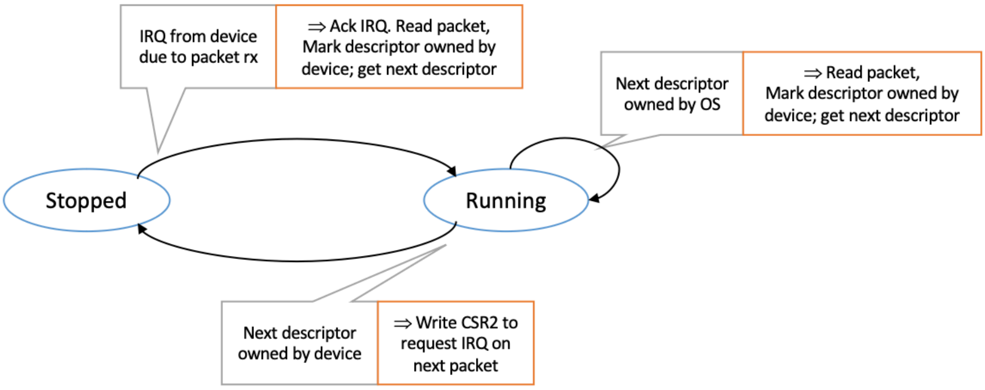

IRQ from device due to packet rx
⇒ Ack IRQ. Read packet, Mark descriptor owned by device; get next descriptor
Next descriptor owned by OS
⇒ Read packet, Mark descriptor owned by device; get next descriptor
Stopped
Running
Next descriptor owned by device
⇒ Write CSR2 to request IRQ on next packet

Figure 115: SW - Receiving Packets

## 21.9 Discoverable Busses - PCI

Some questions remain unanswered: How does the OS know about connected devices? Where do devices have their registers in physical memory? Which interrupt vectors are used for which device? Those problems are solved by the design of the I/O bus, such as the Peripheral Component Interconnect (PCI).

PCI is a standard for connecting devices. Several extension such as PCI-X, PCMCIA, PCI-Express, ... exist. It defines a set of bus protocols for communication between devices and a software-visible interface to I/O hardware. Today, PCIe is used instead. It is an extension of the software-visible interface defined by PCI.

PCI solves many problems:

- device discovery (finding out which devices are connected)
- address allocation (for the device's registers)
- which exception vectors to call on a certain device interrupt signal
- intelligent DMA ('bus mastering' devices no longer need a DMA controller)

PCI is set up as a tree. On the one side, Tthe PCI root bridge connects to the CPU, main memory, ... On the other side it connects to peripherals and PCI-PCI bridges. Those allow subtrees of PCI peripherals to be connected at shared bandwidth.

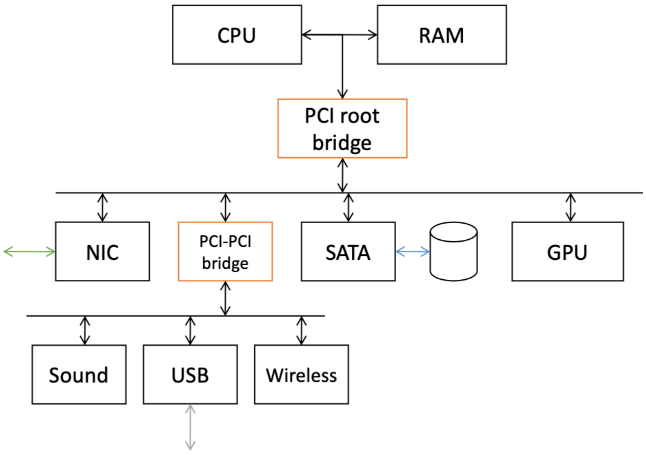


Figure 116: PCI Tree

PCI has several neat properties. Among them being that the PCI address space is flat. This means that each device asks for a set of address ranges (physical space or I/O space). The PCI bridges in the PCI tree then remap addresses from main memory or I/O memory to the requested addresses. This allows each device to appear as a contiguous address range in memory.

Furthermore, PCI devices are self-describing. A configuration header is initially accessed through the parent bridge. This header contains:

- manufacturer ID (16 bits) - assigned by a consortium
- model ID (16 bits)
- class code, what kind of device is this (24 bits)
- version identifier (8 bits)
- allocated/required address ranges
- interrupts
- electrical information
- ...

When working with PCI there are several setup steps to follow for the PCI system:

- Finding all devices
  - We find all 'root' PCI bridges of which there may be multiple in a modern system.
  - Read the configuration file for all attached devices, and do so recursively if one encounters another bridge. Add all found devices to a list of devices and functions. Also record the address space requirements for each device.
- Allocating Addresse
  - Find address ranges for each device and bridge. Each bridge has a range which includes all its children's ranges. This is recursive till the end of the PCI tree. Each device has a size of address range it needs. All ranges must be aligned to a power of 2.
  - Program each PCI bridge with translation information. Each device is, thus, assigned a base-address/range pair (BAR).

Once setup, there are two main ways for PCI devices to interact with the system:

- PCI Interrupts
  - PCI uses four interrupt lines: INTA, INTB, INTC, INTD
  - Bridges allow arbitrary wiring of the four device device lines each to bridge lines. The root bridge translates the signals to system interrupts.
  - PCI Express introduced Message-signaled interrupts (MSI). A device writes an encoding of an interrupt to a specific memory address. This is translated by the root bridge to a system interrupt. Such interrupts can target specific cores or APICs (Advanced Programmable Interrupt Controller).
- DMA over PCI
  - PCI allows Bus Mastering. This means that devices can issue read/write transactions to anywhere in memory by themselves. External DMA controllers, thus, become irrelevant. This provides much more flexibility and intelligent behavior from the device side.

## 21.10 Beyond

Today, devices are very complex. The descriptors generates by DMA can be thought of as a program instructing some user program to perform specific tasks. It's not like devices supply raw data. Instead, the with increasing frequency can be considered processors themselves.

With bus mastering, we also enable intelligent devices, which can freely work with main memory, access other devices, ... This allows for complex interactions between the CPU and devices. High-performance is enabled through extensive in-RAM buffering.

Besides, devices today don't always mean bulky hardware to connect to a system. Advanced GPUs etc. are of course still large and require to be manually connected. But many devices are already build-in today: APICs, MMUs, power controllers, GPUs, DSP, USB controllers, ... Systems that already integrate many of their devices are called Systems on a Chip (SoCs).