Summary: System Programming and Computer Architectures

Jean-Claude Graf

December 22, 2020

Contents

0.1 Lecture 1: Introduction SysProg and C . . . 1
0.2 Lecture 2: Continue Introduction C . . . 2
0.3 Lecture 3: Continue Introduction to C and Integers . . . 5
0.4 Lecture 4: Continue Integers . . . 11
0.5 Lecture 5: C Pointers . . . 16
0.6 Lecture 6: Dynamic Memory Allocation . . . 18
0.7 Lecture 07: Wrapping up C . . . 21
0.8 Lecture 08: Implementing Dynamic Memory Allocation . . . 25
0.9 Lecture 09: Basic x86 Architecture . . . 29
0.10 Lecture 10: Basic x86 Architecture and C Control Flow . . . 33
0.11 Lecture 11: Continue C Control Flow . . . 38
0.12 Lecture 12: Compiling C Data Structure . . . 41
0.13 Lecture 13: Linking . . . 45
0.14 Lecture 14: Code Vulnerabilities . . . 56
0.15 Lecture 15: Floating Point . . . 62
0.16 Lecture 16: Optimizing Compilers . . . 68
0.17 Lecture 17: Architecture and Optimisation . . . 70
0.18 Lecture 18: Cache Optimisation . . . 82
0.19 Lecture 19: Exception Handling in Processors . . . 90
0.20 Lecture 20: Virtual Memory . . . 97
0.21 Lecture 21: Continue: Virtual Memory . . . 105
0.22 Lecture 22: Large Pages . . . 115
0.23 Lecture 23: Continue: Coherency Protocols . . . 124
0.24 Lecture 24: Non-Uniform Memory Access(NUMA) . . . 133
0.25 Lecture 25: Direct Memory Access . . . 140

0.1 Lecture 1: Introduction SysProg and C

Week: 1

System Definition

System encompasses operating systems, database systems networking protocols and routing, compiler design and implementation, distributes systems, cloud computing and online services, big data and machine learning frameworks. System programming is on and above the HW/SW boundary.

Introduction to C

There are many C standards and C-like variants. In this course we use the C99 standard. Many languages have borrowed the syntax of C. C is very fast and because it is close to the metal one knows what the code is doing on the hardware.

Therefore, C is the language of choice for OS devs, embedded systems, tasks where speed is important etc.

In contrast to e.g. Java, C does not have objects, classes, methods, interfaces, built-in types, exceptions etc. We get what the hardware gives us. Furthermore, the programmer is responsible for the memory management, garbage collection etc. In C one is also able to access the memory directly via pointers.

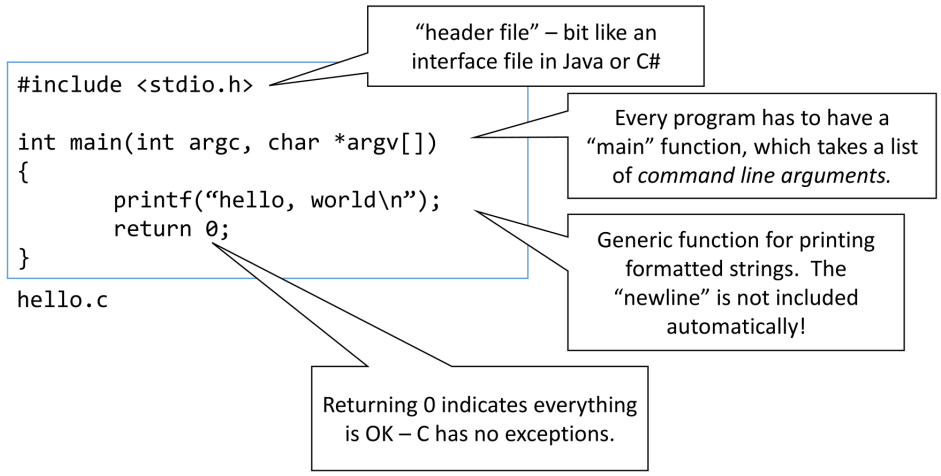
- **Comments**: `/*...*/` or `//`
- **Identifiers**: Same as in Java
- **Block** structure using `{...}`

0.2 Lecture 2: Continue Introduction C

Week: 1

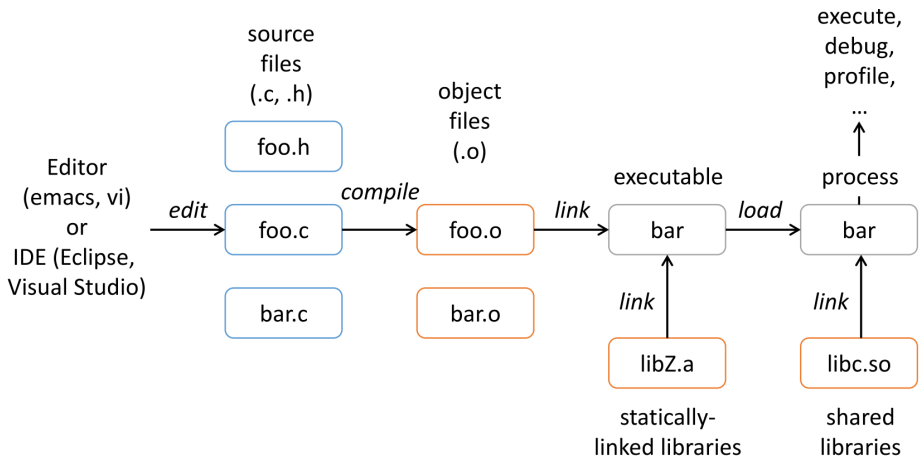
Example Code

Header files are external linked files. Ever program has to have `main` function which takes several command line options. The number of arguments if given by `argc` (argument count) and the arguments themselves via `argv` (argument vector)



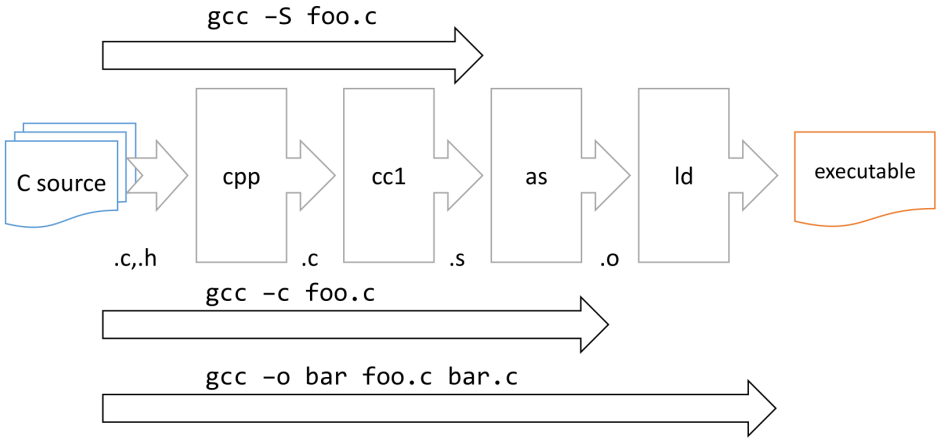
Workflow

Programmers write source code `.c` files and header files `.h`. At first, macros get substituted. Each of these files is compiled into assembly `.s` and then into object files `.o`. Object files are linked with their external libraries `.a` and we get an executable file. There are also libraries (shared libraries) `.so` which are dynamically linked while running the code.



`gcc` is the *compiler* (it does also all other parts of the workflow) we use in all exercises in this lecture.

We can stop the compilation at each stage:



Control Flow

Conditionals

They are written pretty much are we are used to.

```
if (Expression) {
    Statement_when_true
} else {
    Statement_when_false
}
```

The brackets can be omitted when the statements are just one line.

```
if (Expression) Statement_when_true
else Statement_when_false
```

Switch If we not not break on a successful condition, multiple conditions may be executed.

```
switch (Expression) {
        case Constant_1: Statement; break;
        case Constant_2: Statement; break;
        ...
        default: Statement; break;
}
```

Loops

The syntax of the three loop types available in C are as we are used to from other languages.

If the statement is just one line, they may we written as:

- `for (Initial; Test; Increment) Statement`
- `while (Expression) Statement`

- `do Statement while (Expression)`

Other Statements

Break Is used to break out of the current loop/switch. Unlike in Java, we cannot give a label.

Continue Stops current iteration of loop. Unlike in Java, we cannot give a label.

Goto `goto Label` jumps straight to `label`. It is controversial, but occasionally very useful. It makes code hard to read.

Functions

Functions are very similar to Java. They have a name, return type, argument types and a body.

`type name(paraType paraName) body`

The entry point to each C program is the `int main(int argc, char *argv[]) body; return 0` function. `argc` is the number of parameter and `*argv` a list of string parameter.

`return (Expression)` is used to return a value (may be computed by an expression) of a valid type (must conform to the defined return type).

I/O

`printf()` is used for outputs. The general format is that the first argument is the format string, containing special *replacement delimiters*, followed by the values to be inserted into the string. The number as well as the type of the variables must match the one in the format string.

`man 3 printf` for docs.

Types

Declarations Similarly as in Java, before a variable can be used, it must be declared `type name;`. Declaration and assignment can be combined into a single statement `type name = value;`.

Scope Variables declared inside a block have a scope of only this block. Variables which are not defined inside a block have a scope of the entire program.

Static `static` in C is very different than in Java. Its meaning depends on its scope (/if it is inside/outside a block). If inside a block, the value of this variable persists between function calls. If static is used in a variable declared outside any block, the static makes that this variables is only accessible to this file (compilation unit) instead of the entire program.

Types and Sizes

C data type	Intel x86-64
char	1
short	2
int	4
long	8
float	4
double	8
long double	10/16

Numbers

Integers are signed by default. To make it more explicit, `signed` or `unsigned` can be prepended.

In standard, `int` have a size of 4 bytes. Sometimes we want larger or smaller ints. In such cases we can use `<stdint.h>`, which provides extended integer types:

- Signed: `int8_t`, `int16_t`, `int32_t`, `int64_t`
- Unsigned: `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`

Conversion between different types is complicated and determined by the hardware. Between different integer types as well as between different floating types there is implizite conversation. Between anything else, we can use explizite conversation.

Booleans

Booleans are integers. Zero means false, anything else means true. Negation turns zero into non-zero and vice-versa. However, C99 supports *real* boolean types via `<stdbool.h>`.

Statements as Expressions

Any C statement is also an expression. This means that any statements returns a integer ($\rightarrow$ boolean) value. This also counts for loops, assignments etc. which do not explicitly return any value.

For example assignment statements return the value they are assigned. We could use the following code to check for errors in a function call.

```
int rc;
if (rc = myFunc()) {
    # error
} else {
    // everything good
}
```

void

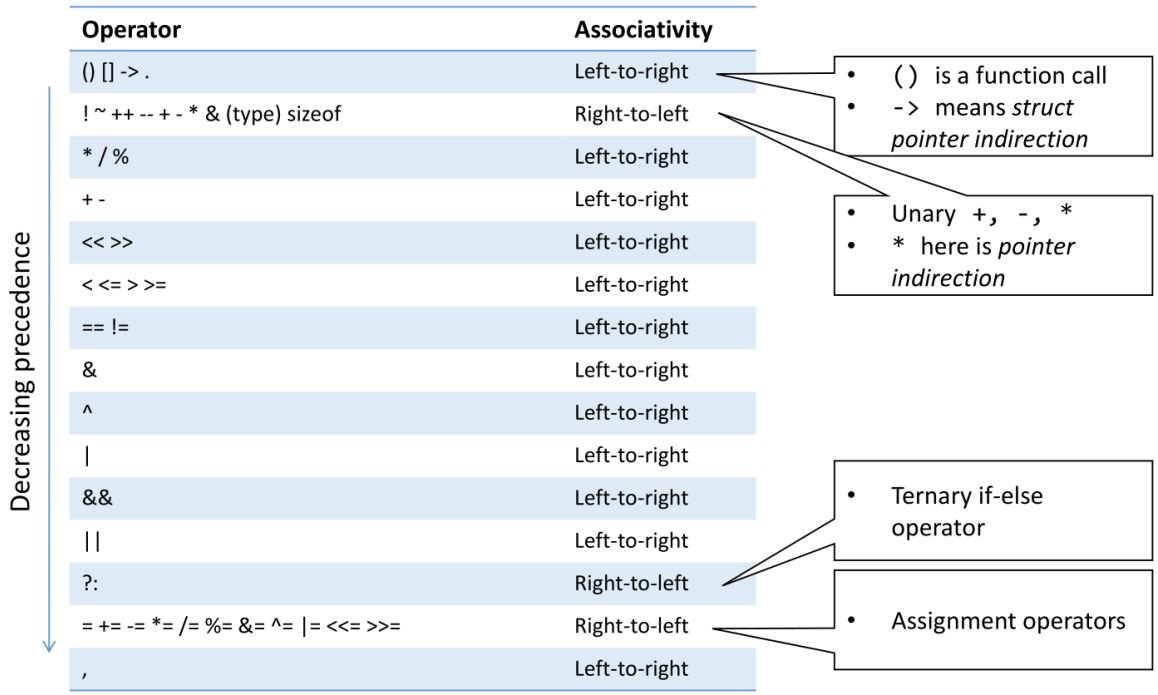
`void` is a type which has no value. Pointers have typically a type assigned. `void` is used for untyped pointers (make them point to raw memory). We also use it for declaring functions without return value.

0.3 Lecture 3: Continue Introduction to C and Integers

Week: 2

Operators

Operators and their precedence.



Assignment

Beside the ternary operator, assignment operators are the only operators which are right-to-left associative while all other operators are left-to-right associative.

As said before, assignments are expressions to (unlike in most other languages).

Post-/Pre-Increment

They work as we are used to from Java. We can also apply them to pointers. In that case the pointer value is not incremented by one but by the size of the type of the pointer.

Casting

Casting is a prefix operator `(newType) value;` which takes a value of a certain type and casts it to another designated type. The bit-representation does usually not change (but sometimes it does). E.g. when casing a small singend integer to a larger singed integer, the leading most significant bits is copied over into the extended region.

Arrays

An array is a finite vector of values of the same type. They are arranged contiguously in memory. (Most) C compilers do not check array bounds. The indexed as usual, starting at 0 up to $N-1$ (aka. C++ indexing :P).

Even though arrays are defined with a certain length, given a array we cannot get its length.

Basics

Definition The are defined using `type name[length];`. E.g. a float array with 5 elements is defined with `float data[5];`.

After the definition, no values are assigned to the array, meaning we do not know what elements the array holds. So we cannot assume that it is all zero or something. In order to initialize each value to zero, we can use `type name[length] = {};`.

We can also initialize by assigning a set of values like `type name[length] = {val1, val2, ... }`.

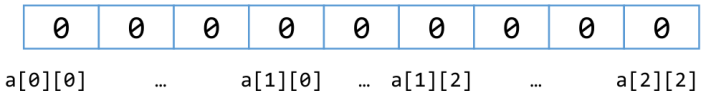
Assignment Assignment is done one-by-one using `var[i] = val`.

Access Arrays are accessed using `var[i]`.

Multidimensional Arrays

Multidimensional arrays are stored in row-major in a continuous array. Meaning they are stored as $a[0][0], \ldots, a[1][0], \ldots, a[2][0], \ldots, a[2][2]$.

```
int a[3][3];
```



When viewed as a matrix, the first brackets defines the number of columns and the second bracket the number of rows.

Strings

There is no string type in C. Instead strings are an array of chars. The end of the array is marked by the null character
0. We could assign it directly as an array of chars `char arr[3] = {'a', 'b', '0'}`. This way we must add the null character explicitly. It is often easier to assign the sting directly as `char arr[3]= "ab"`.

Very important is that when setting the length of the array, we keep in mind that the null character requires some space too.

Given that strings have a special termination character, unlike for general arrays, for char arrays we can determine the length of the array given the array.

String Manipulation There are no built-in methods for manipulating strings, but there are many libraries which support this. For example `<string.h>` is such a library and their manipulation methods have the format `strxxx()`.

`strcpy` copies a given string, `strcat` concatenates two given strings, `strcmp` compares two strings of equality etc.

Integers

Represent Integers

Integers are stored as a sequence of bytes. x86 uses little-endian, meaning that the least significant byte comes first.

Operations

Bit-Level Manipulate single bits of any *integral* data type (long, int, short, char, unsigned). The arguments can be viewed as bit vectors.

&	and
|	or
~	not
^	xor

We can also use the to represent and manipulate sets. We encode bit vectors in a certain way.

Logical Viewing data as true and false where 0 means false and everything else true. Either of the two states is always returned. They apply to the vector as a whole.

&&	and
||	or
!	not

Shift

- Left shift: $x << y$ shifts bit-vector x left by y positions
  - Fill right with 0s.
- Right shift: $x >> y$ shift bit-vector x right y positions
  - Logical shift: fill with 0s (always applied if int is signed)
  - Arithmetic shift: replicate most significant bit (always applied if int is unsigned)

The behaviour of negative numbers and numbers larger then the vectors is undefined.

Encoding

Unsigned integers are encoded in such a way, that with increasing position (LSB to MSB), the factor of two increases: $\sum_{i=0}^{w-1} x_i \cdot 2^i$.

With two's complements encoding we have to consider that the MSB is the sign bit. If is set, 2^{w-1} is subtracted of the number represented by the last $w-2$ bits: $-w_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$.

Ranges

- UMin $= 0$
- UMax $= 2^w - 1$
- TMin $= -2^{w-1}$
- TMax $= 2^{w-1} - 1$

Depending on the word size of the number, we can represent different ranges:

	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

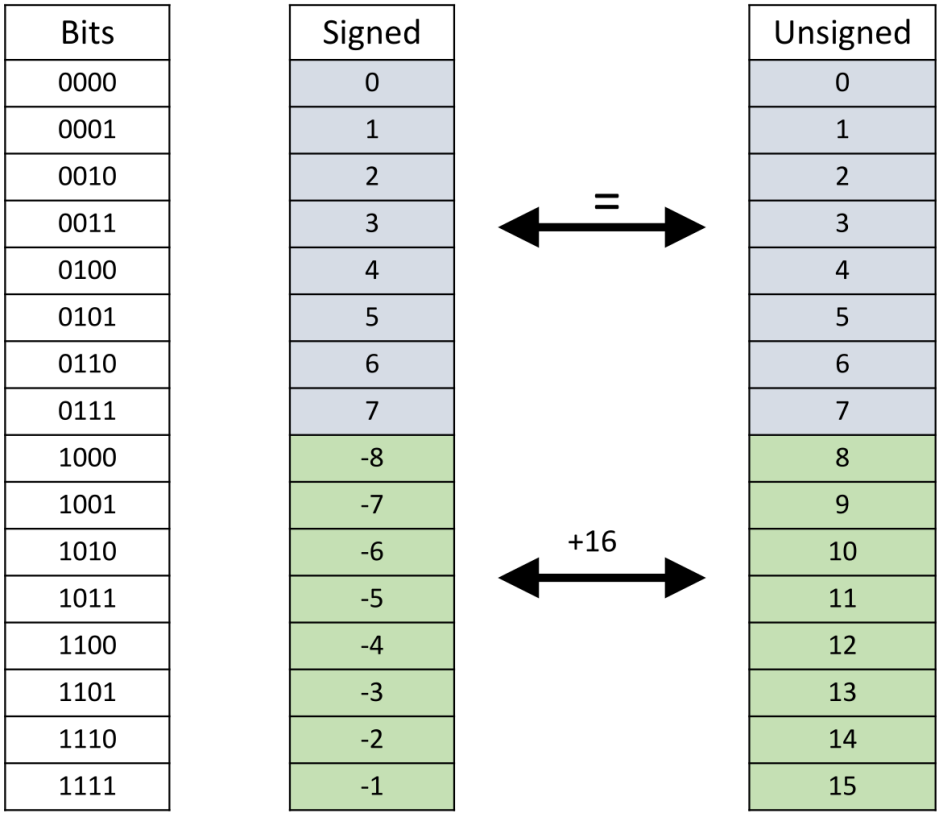
From this table we can notice two interesting things:

1. $|\text{TMIN}| = |\text{TMAX} + 1$. I.e. we can represent one negative number more than positive (asymmetric range)
2. $\text{UMAX} = 2 \cdot \text{TMAX} + 1$

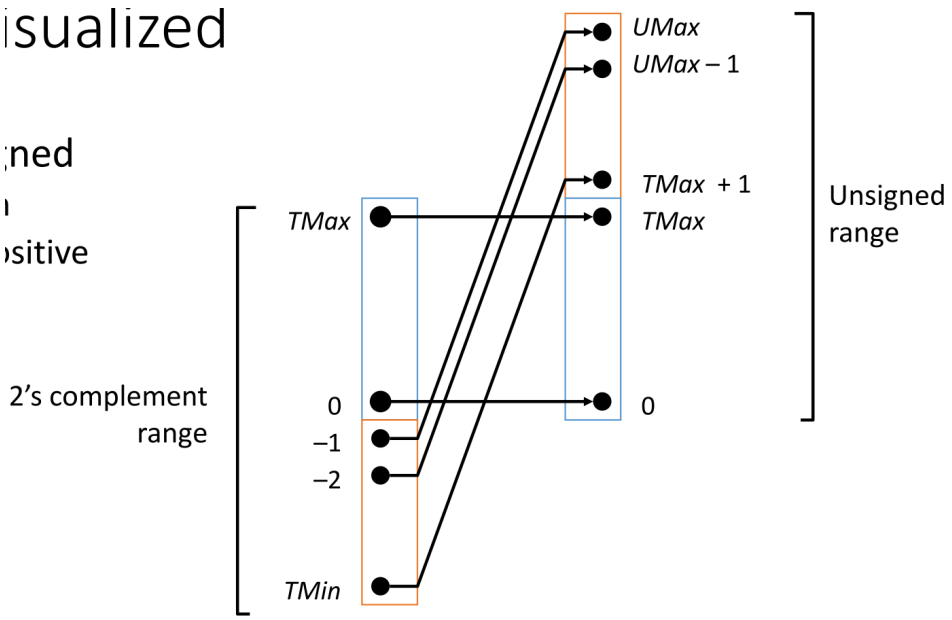
In practice, we can use `<limits.h>` which provides `ULONG_MAX`, `LONG_MAX`, `LONG_MIN`... to get these bounds.

Visualisation

When comparing the interpreted signed/unsigned number of a certain bit patters, we see that for the lower halve of numbers, the signed and unsigned numbers are equivalent. Fur the other halve of numbers, i.e. all which have MSB of 1, we have to add $2^{\text{word size}}$ to get from signed to unsigned.



The difference between signed and unsigned can also be seen as a shift of the ranges.



Signed vs Unsigned

By default numbers are considered as signed. In order to get a unsigned number, we have to suffix the number with a `U`.

In order to get from one to the other, we have to cast the number. During a cast the bit pattern of the number does not change, it is just interpreted differently.

There are two different ways to cast a number:

Explicit Given signed number `int a;` we can cast it to unsigned number `unsigned b` using `b = (unsigned) a;`. The other ways around, casting from unsigned `unsigned b` to signed number `int a` is done using `a = (int) b;`.

Implicit This kind of conversion happens when we assign a signed/unsigned number to a unsigned/signed. I.e. given `int a; unsigned b;` and we assign `a = b;` or `b = a;`.

Beside assignments, implicit conversion also happens when we mix signed and unsigned numbers in an expression. It that case the signed number is casted implicitly to a unsigned.

Constant 1	Constant 2	Relation	Evaluation
0	0U	==	Unsigned
-1	0	<	Signed
-1	0U	>	Unsigned
2147483647	-2147483647-1	>	Signed
2147483647U	-2147483647-1	<	Unsigned
-1	-2	>	Signed
(unsigned)-1	-2	>	Unsigned
2147483647	2147483648U	<	Unsigned
2147483647	(int) 2147483648U	>	signed

If not done by purpose, this can have undesired side effects including security concerns.

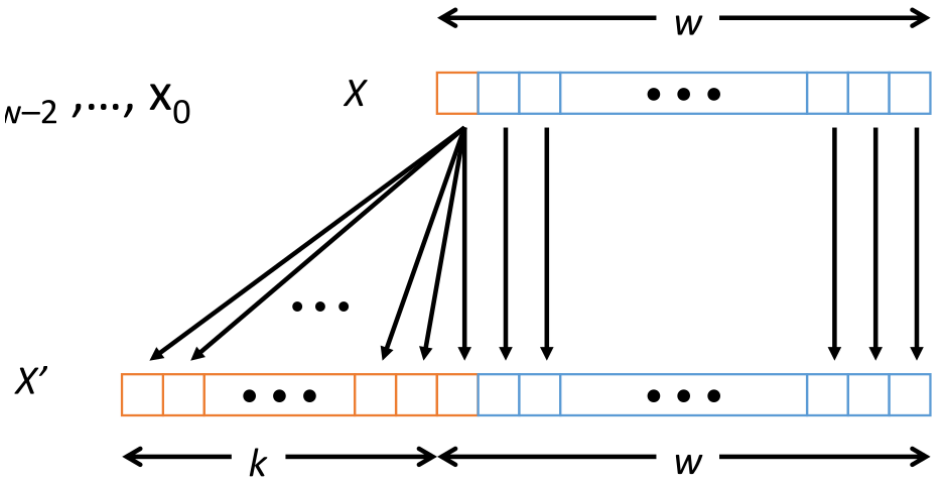
0.4 Lecture 4: Continue Integers

Week: 2

Sign extension

Given a w bit signed integer x, convert it to a $w+k$ bit signed integer with the same value.

We do that by coping the most significant bit of x and copy it to all k new bits



For unsigned numbers, the number is padded with zeros at the beginning.

C does this sign extend automatically for us.

Truncation

Throws away the left bits. For unsigned numbers it represents the modulo operator. For signed integers it is also like modulo, but it does it in a special way (negative number may get positive and vice versa).

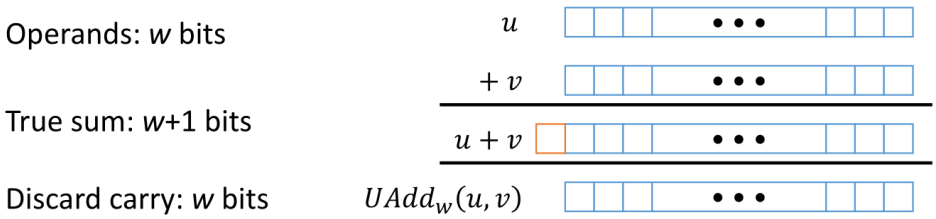
Also, only small numbers remain unchanged.

Addition and Subtraction in C

Negation Number x can be negated using $\tilde{x} + 1 == -x$.

It follows that $\tilde{x} + x == 111\ldots111 == -1$

Unsigned addition When adding two unsigned number u and v of length w, we get a number of length $w + 1$.

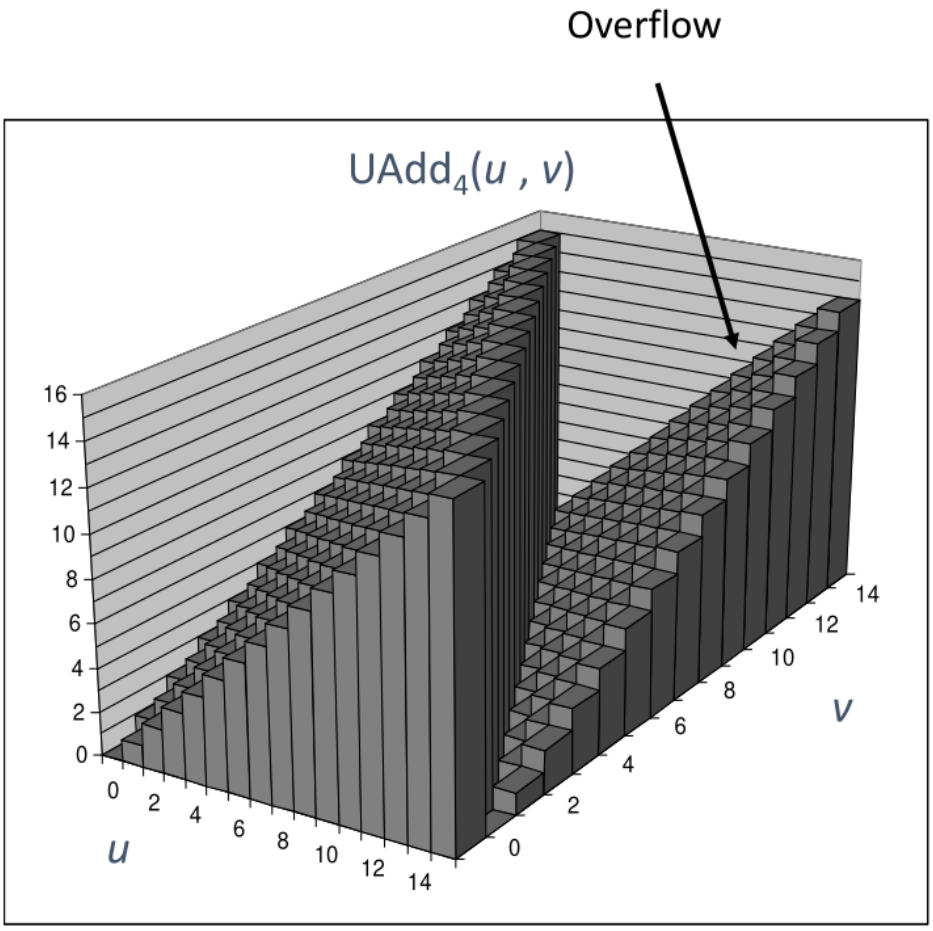


The function $UAdd_w(u, v)$ ignores the extra bit which would be required if the sum overflows the w bits. This function can also be seen as:

$$s = UAdd_w(u, v) = u + v \mod 2^w$$

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

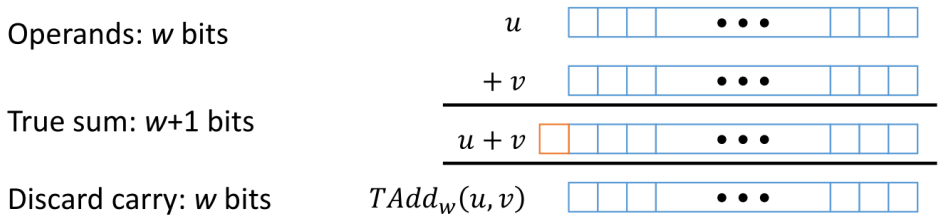
Then the sum overflows, it wraps around.



The $Uadd_w$ function is the classical unsigned addition operator in C. There are certain compilers which provide some possibility to retrieve a possible overflow.

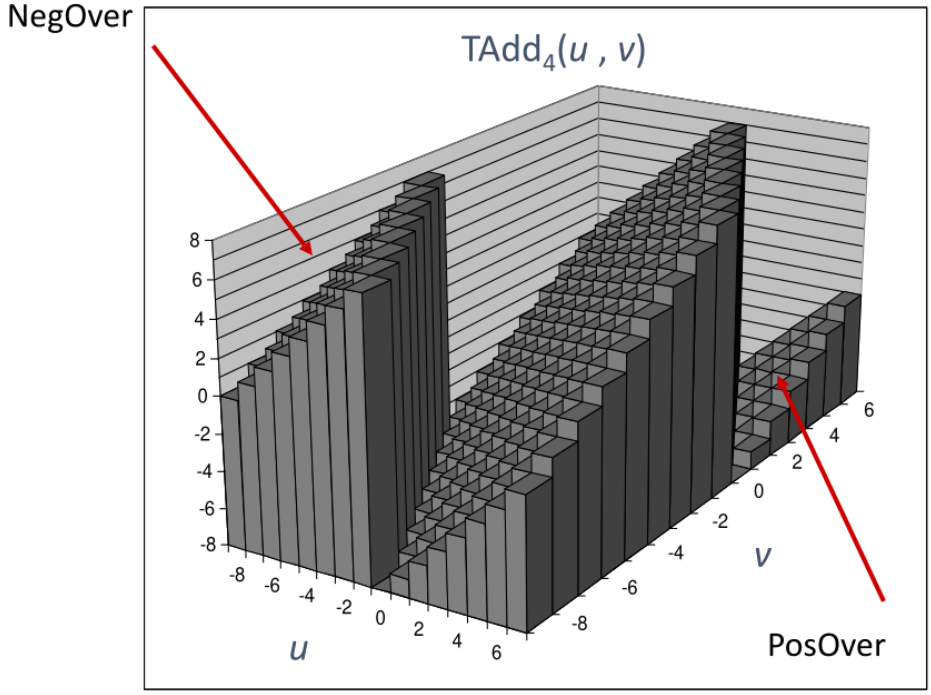
$UAdd_w(u, v)$ is an Abelian group (Closed, Commutative, Associative, Identity, Inverse)

Two's complement addition When adding two signed number u and v of length w, we get a number of length $w + 1$.



The function $TAdd_w(u, v)$ ignores the extra bit which would be required if the sum overflows the w bits.

$TAdd_w(u, v)$ behaves identical to $UAdd$ in the bit level interpretation. But the overflow behaves differently (are interpreted differently). Positive overflow gives a negative number, a negative overflow gives a positive number.



The function can be seen as:

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

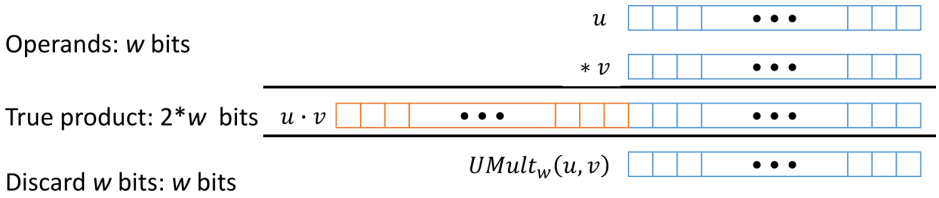
(Neg. overflow)

(Pos. overflow)

$TAdd$ is isomorphic to $UAdd$ and $TAdd$ forms a group (closed, commutative, associative, identity, inverse).

Integer multiplication in C

Unsigned Multiplication When multiplying two unsigned number u and v of length w, we get a number of length $2w$.



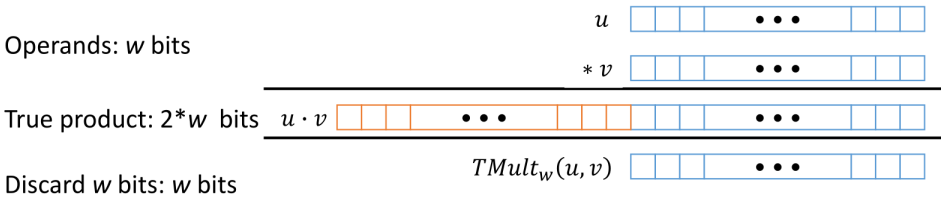
The operator $UMult_w(u,v)$ truncates the overflowing bits. The formula can be seen as:

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

In order that the product is exact x and y must satisfy $0 \le x \cdot y \le (2^w - 1)^2$

$UMULt_w$ forms a commutative ring with $UAdd$.

Signed Multiplication When multiplying two signed number u and v of length w, we get a number of length $2w$.



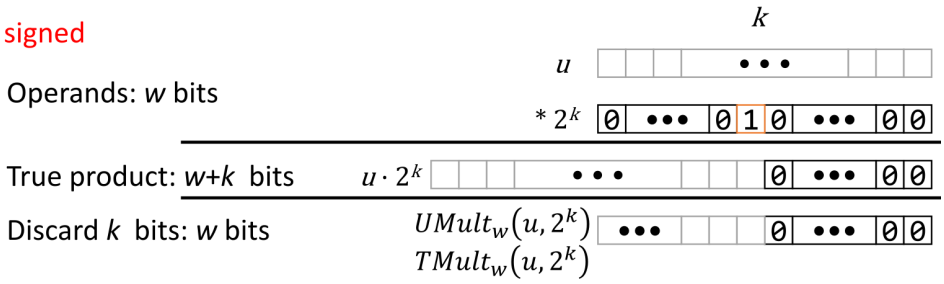
The operator $TMult_w(u, v)$ ignores the overflowing bits. Some hardware puts overflowing bits into a special register, but they are normally not accessibly in C.

The minimal product which can be represented accurately is $x \cdot y \geq (-2^{w-1}) \cdot (2^{w-1} - 1)$. And the greatest number is $x \cdot y \leq (-2^{w-1})^2$.

Multiplication and Division Using Shifts

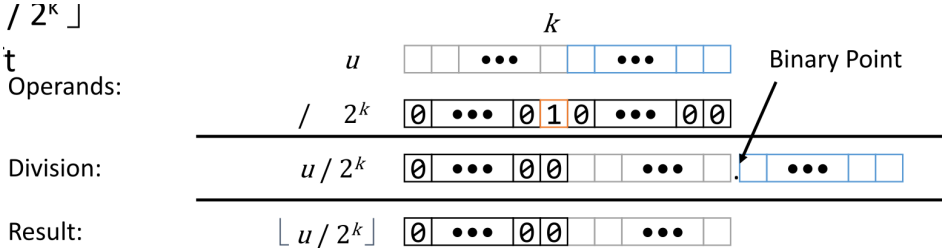
Multiplication As we have seen, the shift operator shifts the binary representation of a number. Regarding the interpreted number, $u << k$ results in $u \cdot 2^k$. This is valid for signed and unsigned numbers.

When shifting a signed/unsigned number of length w by k, the k over flowing bits are truncated.

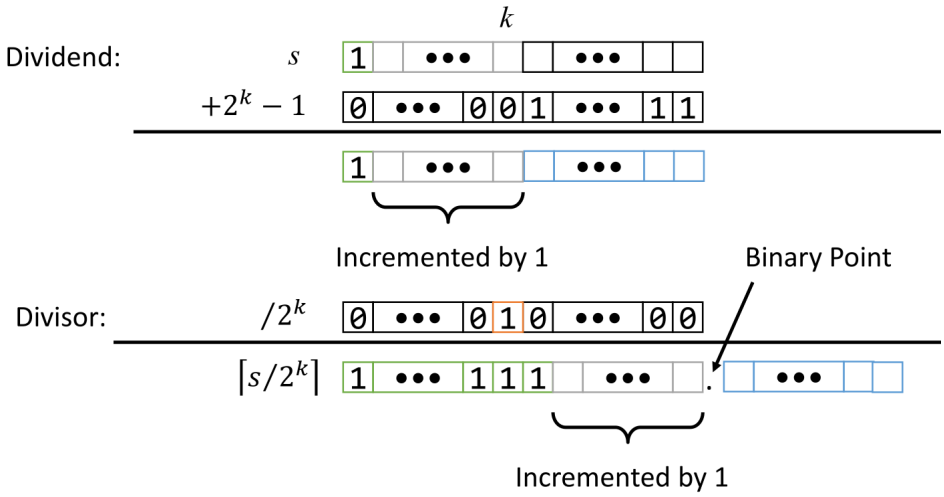


Multiplication is much more complex than simple shifts. Therefore, to calculate $u \cdot 24$ we may write $(u << 5) - (u << 3)$. But modern compiler do such thinks (and even better optimisations) for us, therefore we should not do it.

Division Equally as with multiplication, shifts can be used for division of power of twos. $u >> k$ gives $\lfloor \frac{u}{2^k} \rfloor$. The underflowing k bits get truncated.



Even though this works for unsigned and signed number, for signed numbers the results may be wrong for negative numbers. This is because we round down, but actually we would like to round towards 0. We can fix that by computing $\lfloor \frac{(s+2^k-1)}{2^k} \rfloor$. In C this is equivalent to $(s + (1 << k) - 1) >> k$. This formula works for positive and negative numbers.



Again, the compiler can translate divisions into shifts. For unsigned numbers, it applies the first formula, for signed numbers the second one.

C Integer puzzles

In the following are some integer related puzzles to solve.

- Assume 32-bit word size, two's complement integers
- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Give an example where it is not true

Initialization

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

- `x < 0` ⇒ `((x*2) < 0)`
- `ux >= 0`
- `x & 7 == 7` ⇒ `(x<<30) < 0`
- `ux > -1`
- `x > y` ⇒ `-x < -y`
- `x * x >= 0`
- `x>0 && y>0` ⇒ `x + y > 0`
- `x >= 0` ⇒ `-x <= 0`
- `x <= 0` ⇒ `-x >= 0`
- `(x|-x)>>31` `== -1`
- `ux >> 3` `== ux/8`
- `x >> 3` `== x/8`
- `x & (x-1)` `!= 0`

0.5 Lecture 5: C Pointers

Week: 3

Address Space

The OS gives each process an address space, which contains the virtual memory. The virtual memory is visible only to the process. Each byte is addresses and there are 2^{64} bytes of memory on a 64 bit machine. The OS maps the virtual memory to the physical memory. When the OS loads a program, the following steps are executed:

Loading a program

- create an address space
- inspect the executable file
- copy regions of the file into memory space
- do final linking, reallocation or other preparations

Stack

The stack is allocated in frames. The current procedure's frame always lies on the top of the stack. The frame store local variables, return types etc. On recursive calls, for each recursion a new frame is pushed to the stack

C Pointers

Pointer are variables which store memory addresses. `&x` gives the virtual address where the value of variable `x` is stored. With `%p` its value can be printed. Pointers are declared using the format `type *name;`. Dereferencing a pointer means accessing the memory referred by the pointer. This way we can access the value at the address or write some value to the address. Double pointers are pointers which point to a pointer. They are declared using `type **name;`. Double pointers are mostly used for multidimensional arrays.

The address space layout randomization is responsible that the stack bases and shared library location addresses changes for each execution. This is a security feature which makes debugging tougher. But it could be disabled.

`NULL` is a *guaranteed-to-be-invalid* memory location and has type `void *`. Dereferencing `NULL` causes a segmentation fault. `NULL` is useful to mark end points and invalid addresses.

A `*` is used to indicate that a function returns a pointer as `int *getPtr()`.

Pointer Arithmetic

When incrementing a pointer by 1, the value of the pointer (address) is incremented by the size of the type of the pointer. For integer this is e.g. 4 bytes, for double 8 bytes, for char 1 byte etc... These values depend on the machine and compiler. The size can be evaluated using `sizeof(type)` or `sizeof(value)`.

Arrays and Pointers

An array is a collection of homogeneous data elements stored at contiguous memory adresses. A Pointer is a variable that stores a memory address. So pointer and array are different, but related things. The array name is an expression and treated as a pointer to the first element of the array `a == &(a[0])`. We can access the array by the *pointer* + some offset. The compiler rewrites `A[i]` to `*(A+i)`, hence we could theoretically access arrays like `i[A]`.
However, the array name is different from the pointer when:

1. The array is an opperad of `sizeof()`

   ```
   int a[10]:
   assert(sizeof(a) == 10 * sizeof(int));
   assert(sizeof(&a[0] == siizeof(int *));
   ```

2. The array's address is taken with `&`

   ```
   int a[10];
   assert(&a == a);
   ```

3. The array is sting literal initializer

   ```
   chat a[] = "Hello!";
   chat *b = "Hello!"; // undef. behaviour when modified
   ```

In C parameters are passed as value (since there is nothing like a reference), with the only exception of arrays and functions. They are passed by pointers. Therefore, the following three formal arguments `int *arr`, `int arr[]` and `int arr[10]` are equivalent.
Arrays cannot be renamed, but we change the pointer.

Passed by Reference

Since C is passed by value, we can pass a reference by passing a pointer. However, this pointer is again passed by value and therefore can be modified in the scope of the functions.

Examples

Here some pointer declaration examples:

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

0.6 Lecture 6: Dynamic Memory Allocation

Week 3

Global variables are allocated statically, meaning that they are allocated when the program is loaded and deallocated when the program exits. Function variables and function parameters are automatically allocated when the function is called and deallocated when the function returns.
Often, we want some data to persist across multiple function calls but not for the whole lifetime of a program. Sometimes, the data is also to big to fit on the stack or a function may return a value whose size us unknown to the caller. All these problems an be solved with explizite memory allocation where the program explicitly requests new blocks of memory.

C Memory API

Allocation `void *malloc(unsigned long sz)` allocates a lock of size `sz` and returns a pointer to the first byte of that memory. Actually, it returns a void pointer, which we then need to convert to our desired type. When null is returns, the memory could not be allocated. Therefore, one should always check if the procedure was successful.

Typical usage may look as follows:

```
long *arr = (long *)malloc(10 * sizeof(long));
if (arr == NULL) {
    printf("Error");
    return ERRORCODE;
}
arr[0] = 5L; // etc.
```

Unlike `malloc`, where the initialized mameory contains garbage, `void *calloc(unsigned long nm, unsigned lon` initialized a memory block of size $nm \cdot sz$ containing zeros. It is slightly slower than `malloc` but less error-prone and more readable.

Deallocation When memory is no longer used, one deallocated it using `void free(void *);`, where the argument is the pointer to the first byte of the memory. It is also good practice to null the memory pointer after freeing.

```
long *arr = (long *)malloc(10 * sizeof(long));
if (arr == NULL) {
    printf("Error");
    return ERRORCODE;
}
arr[0] = 5L; // etc.

free(arr);
arr = NULL;
```

Resize Memoryblock With `void *realloc(void *ptr, unsigned long size);` a given block in memory, designated by the pointer to the first memory block, can be resized to the desired size. In most cases the whole memory block is copied to a new location and hence, one should alway use the newly returned address.

```
long *arr = (long *)malloc(10 * sizeof(long));
if (arr == NULL) {
    printf("Error");
    return ERRORCODE;
}

*arr = (long *)realloc(arr, 20 * sizeof(long));
if (arr == NULL) {
    printf("Error");
    return ERRORCODE;
}
```

size_t Is a unsigned int, e.g. `unsigned long` is often used to store memory pointers. They are large enough to hold the size of the largest possible array in memory and are therefore often used to store pointers to memory addresses.

ptrdiff_t Is a unsigned integer of some size and the result of subtracting two pointers. They are used for array loops, size calculations etc.

Managing the Heap

The heap is a large pool of unused memory, used for dynamic allocated data structures. `malloc/calloc` allocate chunks of memory, `free` empties and returns them. In order to know which parts of the

heap are used, `malloc` maintains bookkeeping data to track allocations.

Memory Corruption Memory can corrupt in many different ways:

- Assign values past the end of the allocated memory array
- Assume `malloc` zeroes out memory
- Wrong pointer arithmetic
- Free unallocated pointer
- Double from same block
- Use freed pointer

Memory Leaks Happen when code fails to deallocated memory that is no longer used. For short-lived programs this is ok, but for long-lived program this is bad.
`Valgrind` is a tool which helps to detect memory leaks. In the report, *directly loss* means that locations which are directly pointed to are not freed. *Indirect loss* means that the object we point directly to, points to other objects, which are not freed.

Stuctured Data A `struct` is a C type that contains a set of fields. The tag of the structure is like the name of the structure. To referee to fields of a struct, we use the `tag.field` notation, to referee to fields from a pointer to the struct, we can use `p_tag->field` which is the shorthand notation for `(*p_tag).field`. Stucts can be initialized by either providing all values at the same time, or by assigning value by value.

```
struct Point {
    int x, y;
};

int main() {
    struct Point p1 = {0, 2};
    struct Point p2;
    p2.x = 4;
    px.y = 6;

    struct Point *p_p1 = &p1;
    p_p1->x = 5;

    return 0;
}
```

When we assign struct types `p1 = p2`, the entire content of `p2` is copied to `p1`. As everything else, a struct is passed by value end hence, the whole stuct is copied. Structs can be the return type of a function.
When a struct is mixed of fields of different type, the size of the struct may not be equal to the size of the summe of the sizes of the fiels, but the stuct may get padded. We can print the size of the struct with `printf("%s", sizeof(mystruct));`.

Unions They are very similar to `struct`, but with the difference that only most recently assigned value is saved. All other fields cannot be accessed. There is no safety measurements for checking which field is the valid one.

Type Definitions

typedef They introduce a new type definition and can be used to give a new name to a type. They are kind of an *alias* for the original value. They allow for writing cleaner code and they are very useful for `struct`. For structs we have to add the `struct` keyword everytime we want to reference them. Typedef a struct allows to drop this additional keyword and reference it only by the typedef. Typedef can also be useful to build up complex declarations.

Namespaces A namespace is a set of names of objects in a system. They allow the disambiguate different objects with same name. C has different namespaces for:

1. Label names for `goto`
2. Tags of `struct`, `union`, `enum`
3. Member names, for each `struct`, `union`, `enum`
4. Everything else, including `typedef`

Dynamic Data Structures

In C we can implement dynamic data structures, like linked-lists, as we are used to in other languages.

Generic Data Structures

When we want to let the user choose the type stored in the data structure, we can store in each node a generic pointer `void *element` instead of a fixed type. Clients can convert their specific type to `void *` before pushing it to the data structure.

0.7 Lecture 07: Wrapping up C

Week 4

C Preprocessor The C preprocessor (cpp) is not part of the language buts sits on top of it.

Include `#include` is used to include header files inline in the source code. They provide a basic mechanism for for defining APIs.
Header files are either system header files, which are included using `<systemHeader>`, or own header files, included using `"ownHeader"`. We do not need to provide a path to the file, but the preprocessor will look at certain dedicated paths for the appropriate files.
It is important to **not** include a fie twice. Also one should never include `.c` files.

Macros Macros provide token-based text substitution. A macro is defined using `#define foo bar`. Any appearance of `foo` in the code is replaced with `baz`. Since macros are purely textual, we can use them for *expressions* too. E.g. `#define inc(x) (x + 1)` will replace `inc(1)` with `(1 + 1)`. An undefined, or a defined macro which was removed using `#undef foo` will evaluate to 1 which evaluates to true in an expression.
Macros can span multiple lines and contains *complex* structures by adding a backslash to the end of the line. The problem we may encounter with macros which represent a expression is that adding a semicolon after the term which gets replaced by the macro, results in two semicolons (if there is one in our macro too). When using the trick of *Semicolon Swallowing*, we put our macro block inside a `do ourMacro while(0)` block. Adding a semicolon after the macro will no longer result in an error. The compiler removes the loop anyways.
An advantage of macros is to prevent overhead by inlineing. But the compiler can do that by itself nowadays.

Conditionals Conditionals are useful to take different action at the build time. Preprocessors support conditionals of the form:

```
\#if expression1
    (text1)
\#elif expression2
    (text2)
\#else
    (text3)
\#endif
```

Important is that expressions must be macros and literals only!

The conditional `#ifdef foo` and `#ifndef bar` evaluate to true or false depending whether the macro is defined or not. This is a shorthand for `#if defined(foo)` and `#if !defined(bar)`.

Modularity

A module is a self-contained piece of a larger program. It consists of an externally visible part, containing functions, typedefs, global variables, cpp macros (they define the module's interface) and of internal functions, types, variables which are hidden from the client.

Declaration vs Definition C differentiates between declaration and definition. A declaration is like the definition but without the body. Structs are declarations and cannot be defined.
A *compilation unit* is a C source file and everything it includes. The compilation unit defines the visibility of of definitions.
When we want to access a definition from another compilation unit (or the same) we prepend the declaration with `extern`. When prepending the declaration by `static`, this definition is only visible inside this compilation unit and cannot be accessed from the outside using `extern`.

Global Variables Global variables are also declared and the visibility can be controlled using `extern` as described above.

Header Files Header files specify interfaces and should not contain declarations (structs are definitions and not declarations). The module `foo` has the interface specified in `foo.h` and all clients of `foo` are required to include `#include "foo.h"`. The implementation of `foo` is typically in the `foo.c`, which also must include `foo`. This source file ought not contain external declarations but only definitions and internal declarations.
The naming of `foo.c` and `foo.h` is just a convention which the compiler does not care about. For the execution simple the name of the definition matter.

CPP Convention One should never include the same header file twice or redeclare macros. Therefore, it is convenient to have a mechanism which prevents this. The following boilerplate ensures that:

```
\#ifndef __FILE_H_
\#define __FILE_H_
(macros, headers)
\#endif
```

`__FILE_H_` is the name if the program.

Function Pointers

Function pointers are pointers for functions. For example in `int (*funct)(int *, char);` `func` is a pointer to a function which takes two arguments, a pointer to int and a char. The return value of the function is int. Function pointers are rare in OOP languages, but they are sometimes referenced as *delegates* or *agents*.
Function pointers are very fundamental for a lot of techniques in system code. They allow for example to write *object oriented like code* by having a struct of variables and functionpointers (sometimes called *vtable*). This method does not provide any protection or hiding but does provide the benefit of reusability.

Assertions

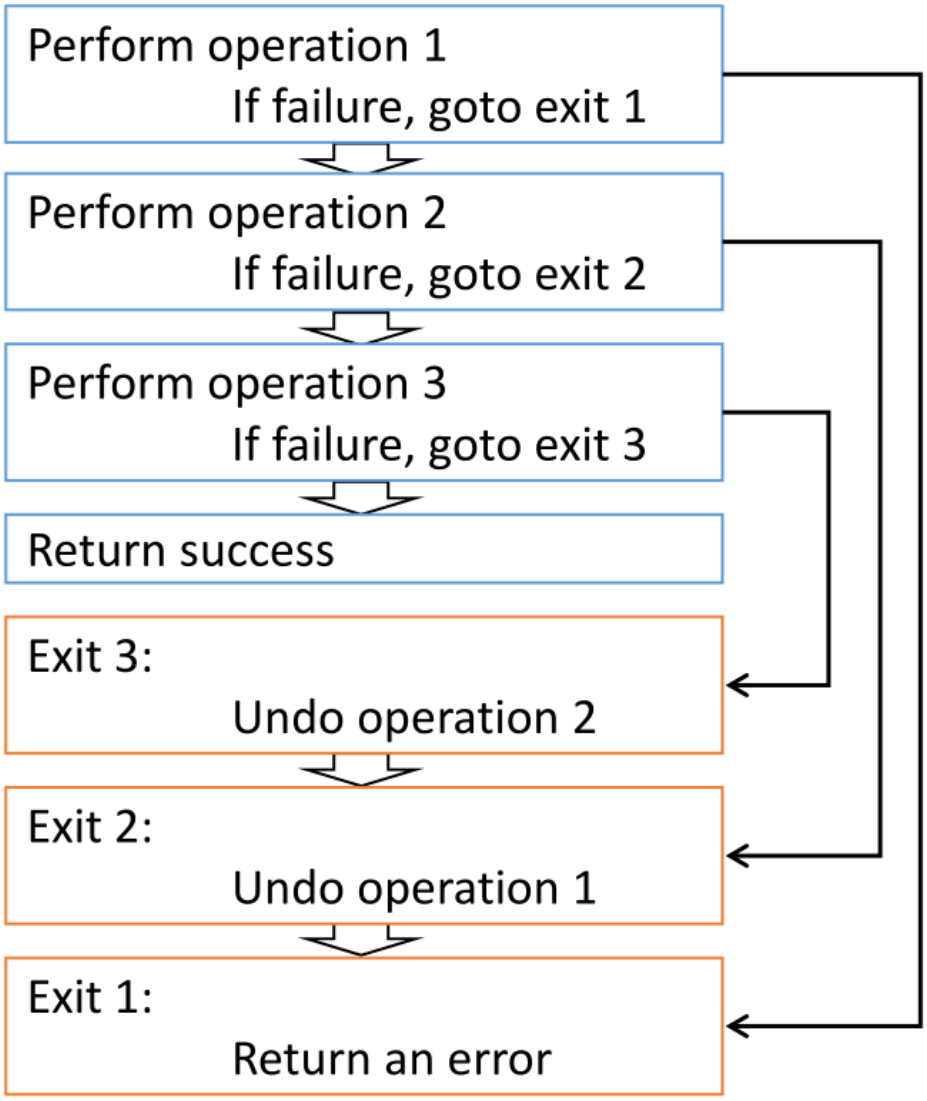
Assertions evaluate an expression at runtime and crash the program if evaluated to false. In that case they print `"<filename>:<linenumber> <functionname>: Assertion <assertionexpression> failed."` and then abort.
Assertions are macros and using the compile flag `-DNDEBUG` they can be removed.

goto

Goto should be be used, except for:

- Early termination of nested loops. But we may also put this in a dedicated function and return.
- Cleanup code. Like transactional memory, when one step fails, undo all previous steps. Depending on which stage failed, we need to undo a different number of steps.

The second usecase is visualized here:



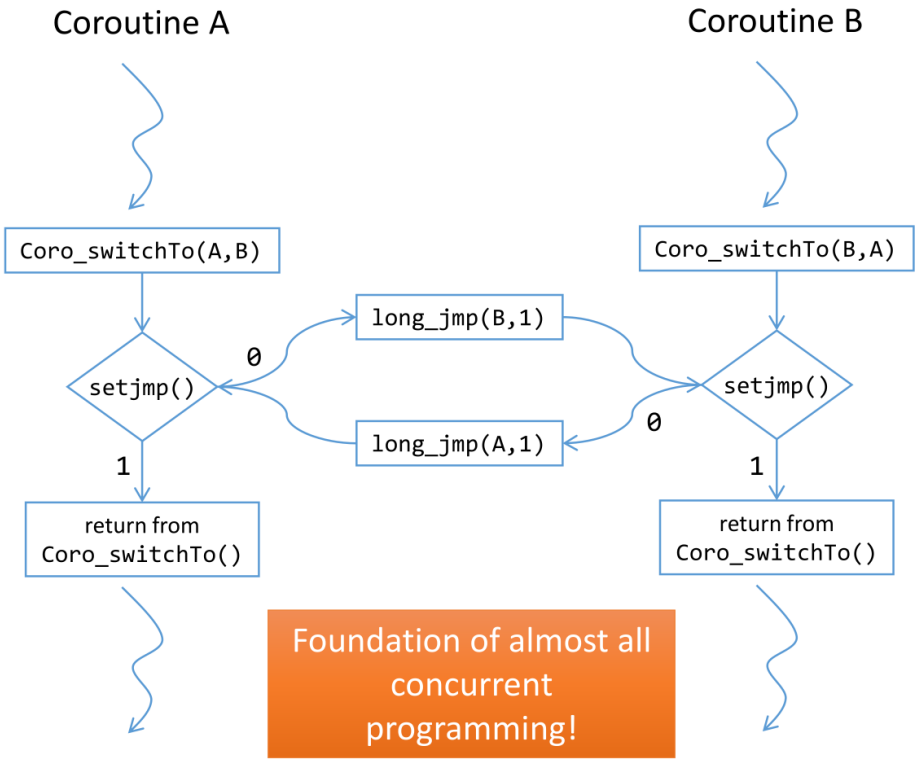
setjmp() and longjmp()

The function `int setjmp`$jmp_buf env$, which must be imported with `<setjmp.h>`, saves the current stack state and environment to `env`. Then it returns 0.

The function `void longjmp`$jmp_buf env, int val$ takes a buffer of `setjmp` and causes the `setjmp` to return again the value of `val` or 1 if $val = 0$. For each `setjmp`, this can only be done once and also only as long as the function containing `setjmp` has not returned.
This construct is very unique to C.

Coroutine

Coroutines are methods which are both at the top of the stack at the same times. Both routines should be able to call eachother to transfer e.g. data. They look similar to threads but they are not. In essence they allow dynamically switching between functions.



Coroutines are completely based on `setjmp` and `longjmp`.

Coroutines are sometimes called *Lightweight threads*, *Protothreads*, *Cooperative mtultitasking*... Thread based programming is based on this schema and we can make coroutines look like threads by dynamically switching into different functions. But there is no concurrently and no parallelism.

The comma is an operator which takes two statements. Both are evaluated, but only the second one is returned.

0.8 Lecture 08: Implementing Dynamic Memory Allocation

Week 4

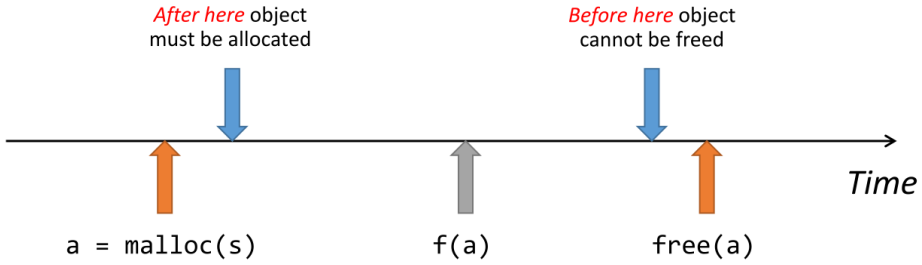
Sidenote: A Runtime allows a programming language to use C libraries. C does not have a runtime, therefore it is well suites for e.g. OS programming.

The OS allocates memory to the C program in pages. But we often only need a few bytes. A memory allocator is a interface which sits between the application and the memory and gives out memory blocks to applications. A block is a continuous range of bytes of any size.
`malloc` is such a dynamic memory allocator

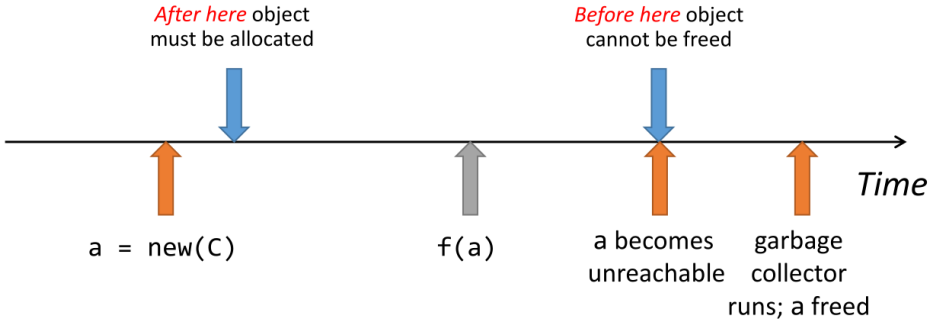
Address Space The address space is split into multiple parts. The NULL pointer is at address $0x00000000$ and contains nothing. On top of it, at $0x00000008$ is the *Text*. This section contains the machine code of our functions. On top of it is the *Data* section, where global variables and parameters (?) are stored. On top of all that is the heap, which grows upwards. At the top at $0xFFFFFFFF$ is the stack, which grown downwards. Functions of libraries, like `malloc` are dynamically linked and the address is determined at runtime. Addresses in the heap are not allocated linearly, but in some different fashion.

Explicit Memory Allocation Allocation and free is done manually by the user. All operations appear in the program source. After allocation, there is a point of time, from when on the object

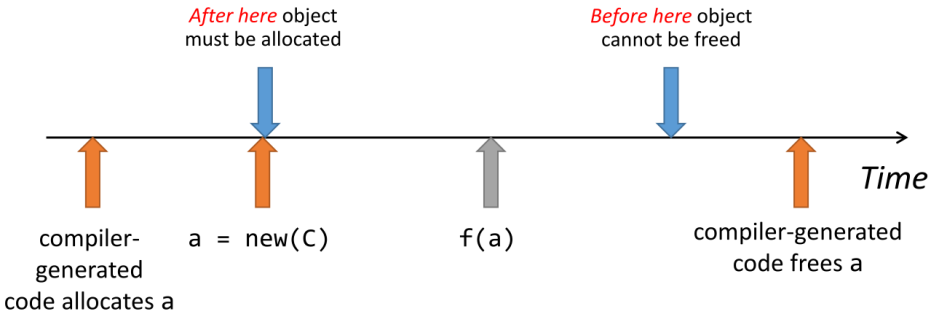
must be accessible. There is another point in time, before which the data object cannot be freed.



Implicit Memory Allocation The user allocates memory by creating new objects. But they are implicitly deallocated by a *Garbage Collector*. After a new object is initialized, there is some point in time, where the allocation must be finished and the object must be available. There is again a point in time before which the object cannot be freed. Once the object becomes unreachable, meaning we do not habe a reference to the abject anymore (this does only work because such languages do not support pointers), the garbage collector frees the memory.



Compiler Supported Memory Allocation The compiler proves from which point on a object must exist, and from which point on it is no longer required. This way the compiler decides when to allocate and deallocate occur. This makes a language extremely fast.



All memory allocations use explicit allocation under the hood, therefore it is crucial to have a good understanding about it.

The Problem

In this lecture we assume that memory is word addressed. This has the advantage that each word can hold a pointer.

Constraints One of the difficulties is that in a program we can issue arbitrary sequences of allocate and deallocate. Memory allocators cannot control the number or size of allocated block and obviously the y can only allocate free parts of the memory. Since they must respond immediately, they also have

to time to reorder or buffer requests. Blocks must also be aligned to satisfy external requirements (8 byte alignment for GNU `malloc`). This means that the payload (/block) must always start at an alignment line (for better CPU performance). One a block is allocated, this block cannot be moved or changes anymore.

Performance Goal Given a sequence of allocations and deallocations, we want to maximize the *throughput* as well as the *peak memory utilization*. These goals are conflicting by nature.

Throughput The throughput is the number of requests per unit time. The goal is to perform allocations and deallocation operations in $O(1)$.

Peak Memory Utilization The peak memory utilization defines how efficiently we use the available memory.

To measure the peak memory utilization we define for some sequence of allocations/deallocation $R_0, R_1 \dots R_k, \dots R_{n-1}$

Aggregate playload P_k: The sum of currently allocated payloads (the difference between all `malloc(p_i)` and all `free(p_j)`). Where a payload of p bytes is generated by `malloc(p)`.

Current heap size H_k: Assume H_k is monotonically increasing.

Peak memory utilisation after k requests U_k: $(\max_{i<k} P_i)/H_k$

Fragmentation Fragmentation may cause poor memory utilization. The two types of fragmentation are:

Interlan Fragmentation: For a given block where the payload is smaller than the block size the payload is padded by unused unused space. Another cause is bookkeeping information, like start and end tags, are added.

External Fragmentation: When there is enough free space on the heap, but not as a single free block.

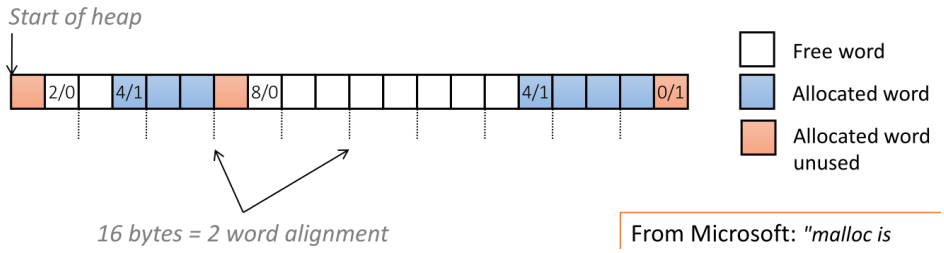
While internal fragmentation is predictable and easy to measure, external fragmentation depends on the pattern of requests und therefore is difficult to predict and measure.

Size of a Memory Block In order that `free` knows how large the block of the memory to free is, `malloc` keeps track of the size of a memory block, by storing the size in the first word of the newly created memory block. The pointer it returns actually points on the second word.

Keeping Track of Free Block

The key policies for memory allocators are Placement policy (wo do we find the block to use), Splitting policy (how do we split blocks which are too large) and Coalescing policy (how do we merge free blocks).

Implicit Free List For each block we need to know its size, as well as whether it is allocated or not. We store all these information in the first word of block. The allocated flag is actually the least signification bit of the length. This introduces some internal fragmentation (because the length up rounded up), but it can be neglected. Especial also since due to the alignment of block, some of the lower-order address bits are often anyways always 0.



Due to alignment restrictions, the first word of the heap is never allocated.

Finding a free block In order to find a free block, we iterate all blocks, starting from the first black, till we find the first free block which confirms to our space requirements. This approach takes linear time in the number of total blocks. In practice, we end up with many small blocks at the beginning of the list.

There are slight variations from this method:

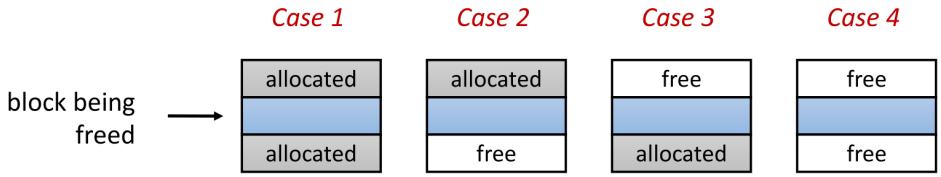
First Fit: As described, takes the first suitable block from the beginning of the list

Next Fit: As first fit, but does not start iteration from the beginning, but where it found a suitable free block the last time. This should be faster but may introduce more fragmentation.

Best Fit: Iterates the whole list and takes the best fit (the one with the fewest bits left over). This keeps fragmentation small but is slower.

Often, we have to split an free block to not waste too much space.

Freeing a block and coalescing When a block is freed, we do not want that a large free block is actually split in multiple small ones. Therefore, sequential free blocks should be merged into one large free block. This procedure is refereed to as *coalescing*.
We can easily skip following free blocks by simple increasing the length of the freed block. This does logically ignore the length entry of the following free block.
In order to coalesce with previous block, we actually need to add an end tag/footer to each block, wich contains, equally as the header tag, the length of the block and a empty flag. This way we can coalesce previous blocks too by looking at the last word of the previous head.
Coalescing can be done in constant time for all possible allocated/unallocated scenarios.



The additional end tag leads to internal fragmentation.

Explicit free lists As the implicit list, it has an start tag and an end tag for coalescing. But in order to find free blocks, a pointer to the previous free block is stored in the first word of the payload area and in the second word, a pointer to the previous free block.

Finding a free block We iterate our list and look for a valid block. In order to allocate a choose block we simple change the pointer pointing to our block.

Freeing a block and coalescing In order to free a block simply add it to our list. There are multiple insertion policies which determine were in the list the new block is inserted:

LIFO policy: We insert the new block at the beginning of the list. This is simple to implement and has constant time. However, fragmentation may be worse.

Address-ordered policy: We insert the blocks so that the address of the blocks are ordered. This requires search to find the desired location, but in turn, fragmentation should be better.

This method is faster than implicit free list, since the required time is linear in the number of free blocks.

Segregated free list Similar to the explicit free list. But in contrast to that, we keep separate lists blocks of different size. Most of the time, small blocks are allocated, hence this gives very good performance.

Finding a free block To allocate a block of size n we search the list of blocks of size $m > n$. On success, we split the block (if required) and place the fragment on the appropriate list. We to to higher-level classes, till we have found a suitable block. If there is no such block, we request a new memory block from the OS.

Freeing a block and coalescing We coalesce the block and put in on the appropriate list.

The advantage of this method is the high throughput as well as the better memory utilisation.

Most allocated blocks are rather small. Segregated list give very good performance. Large blocks are only occasionally assigned. Many systems use this method.

0.9 Lecture 09: Basic x86 Architecture

Week 5

Instruction Set Architecture

The term *Architecture* or *Instruction Set Architecture (ISA)* is very vaguely defended in computer science. It kind of is a building plan for a CPU and it is what one should understand in order to write assembly code. The *Microarchitecture* is in contrast, the implementation of the architecture. A good knowledge of the microarchitecture, on top of the ISA, allows to write very fast, but hardware specific code.
The first real architecture was IBM 360.

CISC: Complex Instruction Set Their philosophy is to add hardware based instruction for often used functionality. So for example, a CISC CPU may have a sorting instruction. This was the dominant style through mid-80's. It was a stack oriented instruction set, meaning a set was used to pass arguments, save PC. Instructions were rather complex and for example one could access memory directly from arithmetic instructions. Conditional codes were used.

RISC: Reduced Instruction Set This is a ISA which which tried to keep instructions as simple as possible, and hence, was a direct contrast to the CISC. Many simple instructions may be required to replace a single CISC. But in turn, they can be executed on small and fast hardware. It is a register-oriented instruction set. Memory access was limited to dedicated load/store operations and it has no conditional code.
RISC is quite different from x86, they tried to strip away everything and make the hardware very simple.

MIPS MIPS is a RISC ISA. It has very standardized instructions which all have the same bite length and take approximately the same amount of time to execute. All operations act on registers except load and store. This makes the circuit simple and faster.
Most numbers in computers are zero. Therefore, it is also handy to have a zero register, as MIPS has.

CISC vs. RISC This is a very old debate and both philosophies have their advantages and disadvantages. Code for CISC is easier to compile and has a smaller code size. RISC was better optimized for compilers and runs faster with simpler chip design.

CISC vs. RISC Today This debate is over and there is no winner. Complex hardware got cheaper to do, so RISC packed more advanced features on their CPU which CISC decoded their complex instruction into simpler microinstructions which were faster to execute. Today, there is almost no difference between CISC and RISC.
Performance is not really an issue today. Today a successfully ISA is one which is compatible with existing code, licensing, security etc.

x86 History

Intel x86 A very successful chip was the 8086, which was a 16 bit processor and which was introduced in 1978. It has many revisions and was also adapted to a 32 version, with the 80386 in 1985. This ISA was named IA32 and recent code can still be run on this CPUs. They tried to radically shift from ia32 to ia64, however they failed because their Itanium architecture were very slow executing existing code, but only fast when writing new, code.
8 months after AMD released their x86-64, Intel released a compatible version ISA called EM64T (later renamed to Intel 64), wich was almost identical to x86-64. Their first 64 bit CPU was the Pentium 4F in 2005.
From 2012 on, when the i7 was released, this Architecture was kept the same and many features were added.

AMD x86 AMD is a rival of Intel and normally, they produces a bit slower but cheaper CPUs with less features. However, they have hired many top engineers and compete with Intel. Thanks to this, they have developed the Opteron, the first 64bit CPU to x86-64 (now called AMD64) based.

Basics of Machine Code

There are two common ways to write x86 assemly. The first one is the AT&T syntax which is common to Unix and which we will use in this course, and there is the Intel syntax, which Windows uses.

Compiling C to Assembly In order to compile to so assembly, we can use `gcc -O -S code.c`, where `-S` is the flag to get assembly code. This produces a file called `code.s`.

Assembly Data Types Assembly has very little concerning semantics. The integers, containing data values and untyped pointer addresses have either 1, 2, 4 or 8 bytes length. Floating point data has 4, 8 or 10. Besides that, there is nothing, no arrays, no structs...

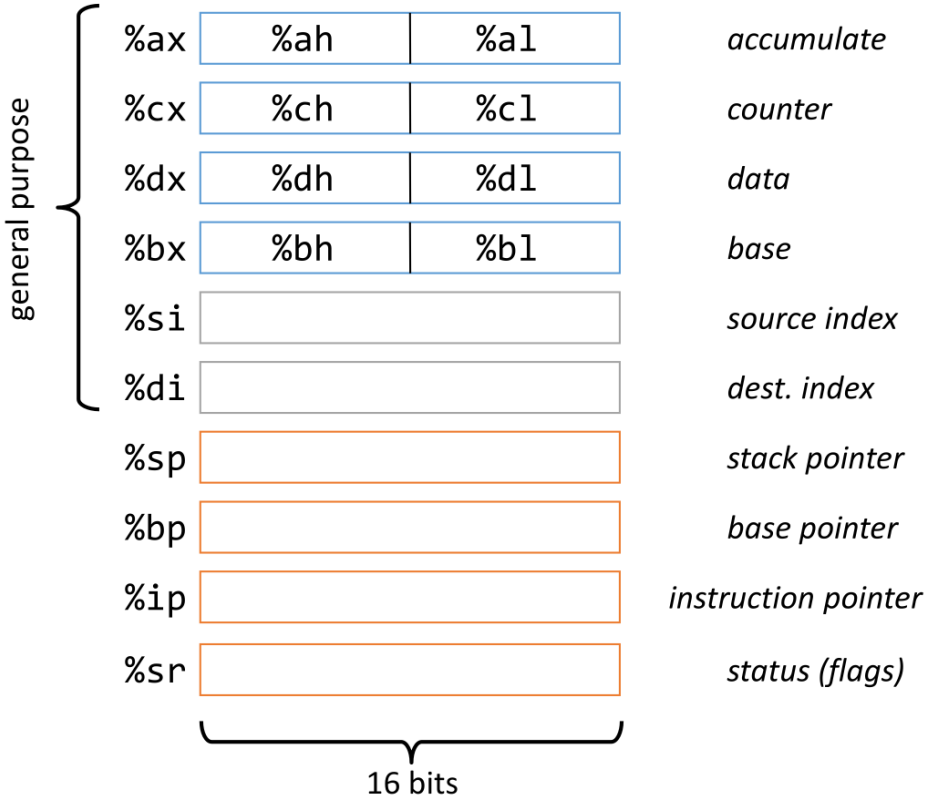
Assembly Code Operations We can perform arithmetic operations on registers or memory data. We can move data between registers and memory and there are conditional and unconditional branches.

Object Code In object code `.o`, the assembly code `.s` ins binary encoded. This is done by the assembler. In x86, instructions have variable length (unlike in MIPS).j

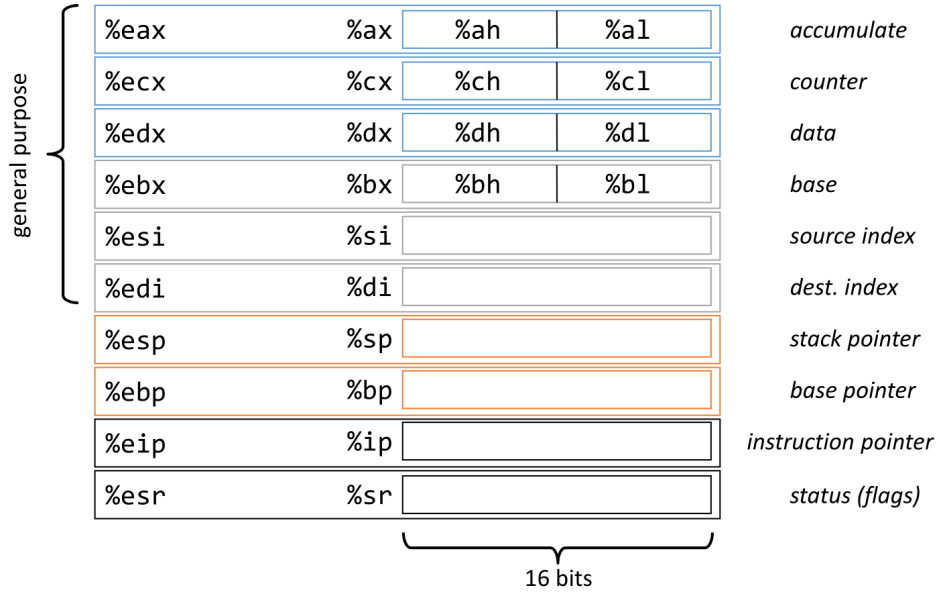
Disassembling Object Code This is the process of generating assembly code `.s` from object code `.o`. Disassembled code is not identical to the original assembly code, since object code does not have all program information, since assembly has some *high level* usercode, like jump target names etc. A disassembler works by examining bytes and reconstructing the assembly instructions.
A disassembler is `objdump -d <file>` and it works for `a.out` as well as `.o` files. Alternatively, one can use the `gdb` debugger for disassembling. This tool has a little cumbersome syntax.

x86 Architecture

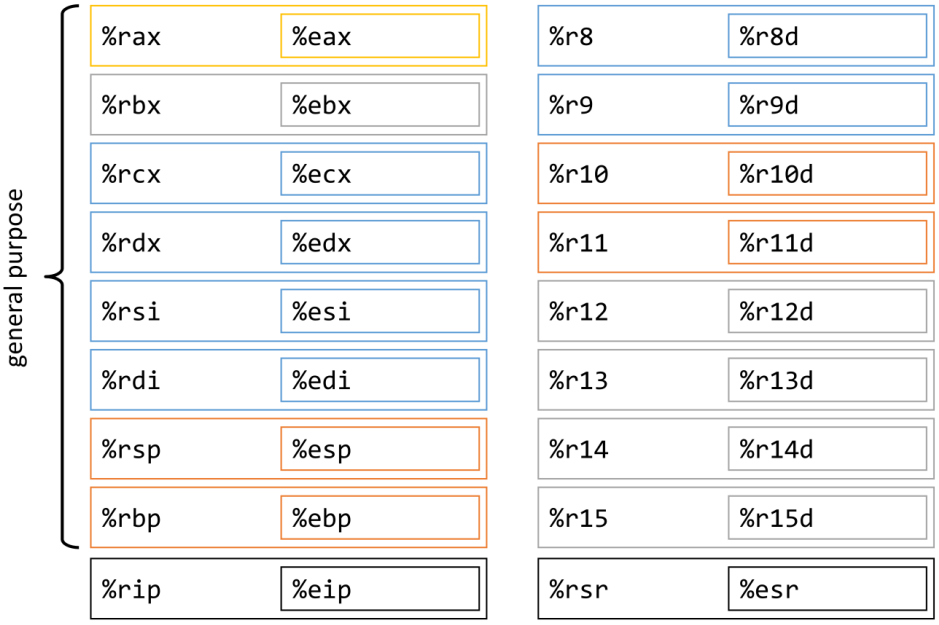
8086 This 16 bit CPU had 10 registers where 6 of them were for general purpose. Although, even the general purpose registers wer designed for a specific purpose. This processor is the predecessor of the 8008, which was a 8 bit processor wich has 4 registers. The 8086 is actually an extension of that and therefore 4 of the general purpose registers could access in high/low (access top 4 respectively bottom 4 bits separately).



80386 (ia32) This ISA is again an extension of the 8086. It extends all registers to 32 bits, but still allows access to the lower 16 bits and the high/low bits. The name for the full 32 bit register are simple the original register names with an appended e.



x86-64 In this ISA they have extended the 32 bit registers to 64 and doubled the number of general purpose registers compared to the ia32 (have a total of 16). Again, they allow to access the lower 32 (but not 16) bits and the register name of ia32 was extended by prepending an r. Even though some registers still have some special name, we can actually use them for whatever we want.



Moving Data There are four flavours of the move instruction `movx Source, Dest`.

movq: move 8-byte *quad word*

movl: move 4-byte *long word*

movw: move 2-byte *word*

movb: move 1-byte *byte*

There are three operand types.

Immediate: Constant integer in decimal or hexadecimal prepended by a $ sign. The size can be 1, 2, 4 or 8 bytes.

Register: Registers are referred by their name. There are general and special purpose registers.

Memory: The general access mode is `D(Rb, Ri, S)` which is leads to a access of `Mem[Reg[Rb] + S * Reg[Ri] + D`

D: Constant displacement of size 1, 2 or 4 bytes

Rb: Base register; any of the 16 registers

Ri: Index register; any of the registers except `%rsp`

S: Scale; any of 1, 2, 4 or 8. It is used for array access for arrays of certain type.

Not all combinations of source and destination are available. So it is not possible to transfer data from memory to memory in one instruction.

Address Computation Instruction The command `lea Src, Dest` does not move any data, but rather calculates the address given by the address mode passed by `Src` and writes the address to `Dest`. This is useful when we need to compute an address without memory memory access, e.g. when creating a pointer. But it can also be used for general arithmetic.

The reason being is that the address calculation requires lot of hardware. Therefore, it does also make sense to use it for arithmetic.

0.10 Lecture 10: Basic x86 Architecture and C Control Flow

Week 5

x86 Integer Arithmetic

Some Arithmetic Operations Important two-operand instructions:

`addl`	$Dest = Dest + Src$
`subl`	$Dest = Dest - Src$
`imull`	$Dest = Dest * Src$
`sall`	$Dest = Dest << Src$
`shll`	$Dest = Dest >> Src$
`shrl`	$Dest = Dest >> Src$
`xorl`	$Dest = Dest \hat{} Src$
`andl`	$Dest = Dest \& Src$
`orl`	$Dest = Dest \mid Src$

There is no difference between arithmetic and logic shift left, and hence, the instructions `sall` and `shll` are equivalent.

Two-operand instructions to store the result of the calculation in one of the operands. There is not separate register. This is a limitation which stems from the early days where registers were rar but nowadays, where registers are actually stored in a register file with +200 resists which is is no longer an issue. (The name of a regster is just a temporary reference to an entry of the register file)

All operations act on the binary numbers and there is no notion of signed or unsigned integers. That is because it does not make any difference for the computation!

Important one operand instructions:

`incl`	$Dest = Dest + 1$
`decl`	$Dest = Dest - 1$
`negl`	$Dest = -Dest$
`notl`	$Dest = \tilde{}Dest$

`leal` for Arithmetic Expressions The compiler clever and makes us of `leal` for arithmetic instructions. This is especially powerful when replacing a multiplication by `leal` and shifts.

The C compiler is better in assembly than we humans :)

Condition Codes

Condition codes are extra bits, like one-bit registers. But they are packed into the status register. They provide us extra information about executed operations.

They have names and they are set (implicitly) during arithmetic instructions (but **not** by `lea`):

CF: Carry Flag (for unsigned)

- Set if carry out of most significant bit
- i.e unsigned overflow

ZF: Zero Flag

- Set if result is equal to 0

SF: Sign Flag (for signed)

- Set equal the most significant bit
- i.e. if interpreted as signed, the result is negative

OF: Overflow Flag (for signed)

- Set to `( a > 0 && b > 0 && t < 0) || (a < 0 && b < 0 && t >= 0)`
- i.e. if interpreted as signed, the results overflows

Condition codes can also be set explicitly using the `cmpl/cmpq Src2, Src1` instruction. This instruction computes i.e. `Src1 - Src2` and sets the conditional flags (but does not store the result of the computation).

The second instruction to set the condition codes explicitly is `testl/testq Src2, Src1`. It computes `Src1 & Src2`, sets the condition codes but again, not the result. This instruction is very useful when one instruction is a mask.

Read Condition Codes The `SetX Dest` instruction computes an expression based on the current flags and writes that to the low-byte of a register. There are the following flavours:

`sete`	ZF	Equal/Zero
`setne`	$\tilde{}ZF$	Not Equal/Not Zero
`sets`	SF	Negative
`setns`	$\tilde{}SF$	Non-Negative
`setg`	$\tilde{}(SF \hat{} OF) \& \tilde{}ZF$	Greater (Signed)
`setge`	$\tilde{}(SF \hat{} OF)$	Greater or Equal (Signed)
`setl`	$(SF \hat{} OF)$	Less (Signed)
`setle`	$(SF \hat{} OF) \mid ZF$	Less or Equal (Signed)
`seta`	$\tilde{}CF \& \tilde{}ZF$	Above (Unsigned)
`setb`	CF	Below (Unsigned)

32 bit instructions set the higher-order bit, of e.g. `%rax` to the same value when writing to `%eax`. The `setX` operations are not 32 bit instructions and therefore do not modify the higher-order bytes but only sets the least significant byte! `xor` is a cheap way to set a register to 0. So with the combination of `xor` and `setX` we can move a certain flag wo any register we want.

`movzbl` moves a byte to a longword and fills with leading zeros. `movsbl` is equivalent, but in contrast to `movzbl`, it sign-extends the value. Using one of these instructions, we do not have to explicitly zero the register.

Jumping The `jX Tag` instructions jump depending on some conditional codes. There are the following flavours:

`jmp`	1	Unconditional
`je`	ZF	Equal/Zero
`jne`	$\tilde{}ZF$	Not Equal/Not Zero
`js`	SF	Negative
`jns`	$\tilde{}SF$	Non-Negative
`jg`	$\tilde{}(SF \hat{} OF) \& \tilde{}ZF$	Greater (Signed)
`jge`	$\tilde{}(SF \hat{} OF)$	Greater or Equal (Signed)
`jl`	$(SF \hat{} OF)$	Less (Signed)
`jle`	$(SF \hat{} OF) \mid ZF$	Less or Equal (Signed)
`ja`	$\tilde{}CF \& \tilde{}ZF$	Above (Unsigned)
`jb`	CF	Below (Unsigned)

There is also conditional move `cmovX src, dest` which moved from source to destination depending on the condition. But is is not that useful. Not even RISC had this feature. This is most of the x86 assembly we will cover.

C Control Flow

If-then-else

`If then else` statements can be compiled as conditional branches. In fact, any conditional can we rewritten as a conditional with if clause only (no else) with a goto in its clause. One should not write code like that, but let the compiler to that work.

C code

```
val = Test ? Then-Expr : Else-Expr;
```

```
val = x>y ? p(x) : p(y);
```

Goto version

```
   nt = !Test;
   if (nt) goto Else;
   val = Then-Expr;
   . . .
  goto Done;
Else:
  val = Else-Expr;
Done:
   return
```

or

```
   nt = !Test;
   if (nt) goto Else;
   val = Then-Expr;
   . . .
Done:
   return
Else:
  val = Else-Expr;
  goto Done;
```

Both orderings are very similar. It depends on the hardware which one is better.

Sometimes the compiler uses conditional move `cmovX src, dest` instead of jumps. Generally, this is more efficient event though we may have to do more work since all instructions get executed. But if the additionally work dominates or if the execution of both branches have side effects, one should not use conditional moves.

C code

```
val = Test ? Then-Expr : Else-Expr;
```

Conditional move version

```
val1  = Then-Expr;
val2  = Else-Expr;
val1  = val2 if !Test;
```

`do-while` loops

`do while` are fancy conditionals with a jump. A conditional backward branch is used to continue looping as long as the condition is true.

C code

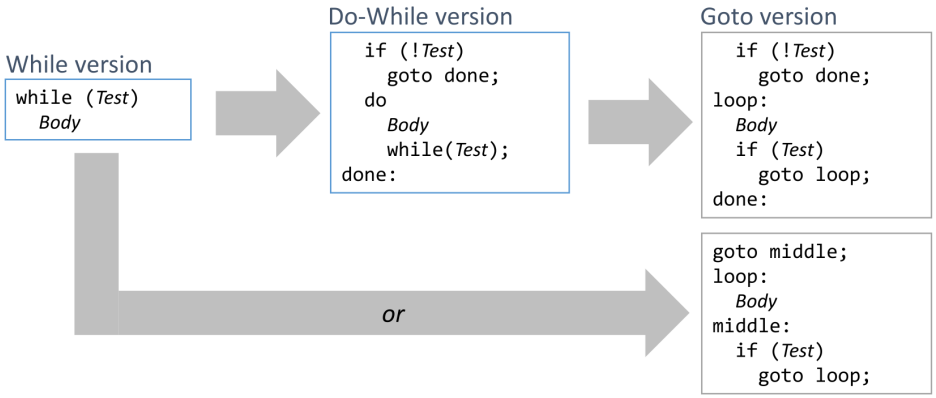
```
do
   Body
   while (Test);
```

Goto version

```
Loop:
   Body
   if (Test)
     goto Loop
```

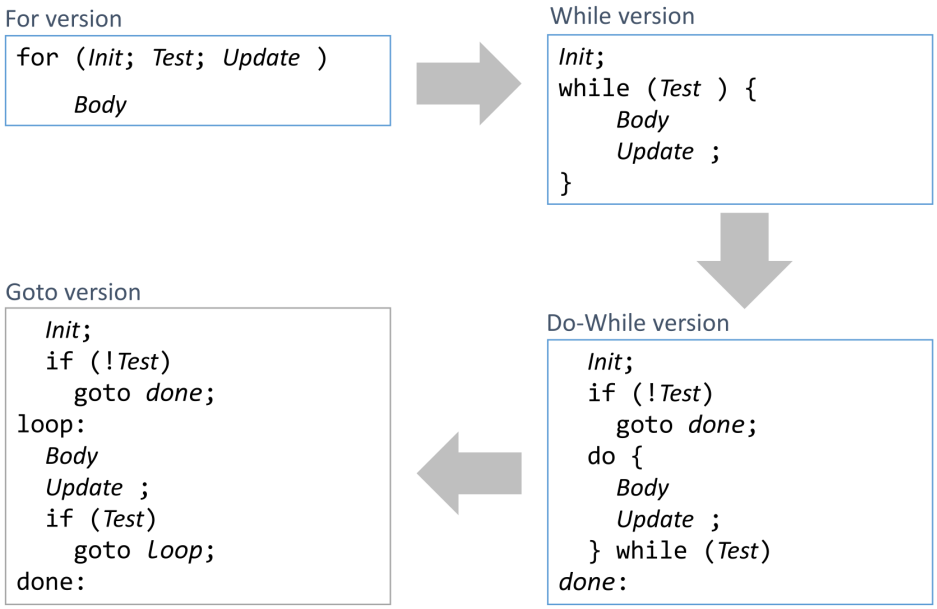
`while` loops

GCC used to convert `while` to `do-whiles` with an additional conditional jump at the beginning. However, nowadays, there are separate blocks for the conditional check and the loop. At the beginning we jump to the conditional, called middle, and jump to the loop, if the condition is meat. This method is called `jump to middle`.



`for` loops

Are `while` loops with an initialisation.



`switch` statements

The obvious way is `if then else`, but nobody does it. There are two kinds of statements, one where the values are compact, and switch when the values are not compact.

Additional Statements:

Anything starting with a dot in assembler is stuff not in the machine code. This is related to linking and information about what is in this file. This is handy for debugging.

`.text` says that code is following

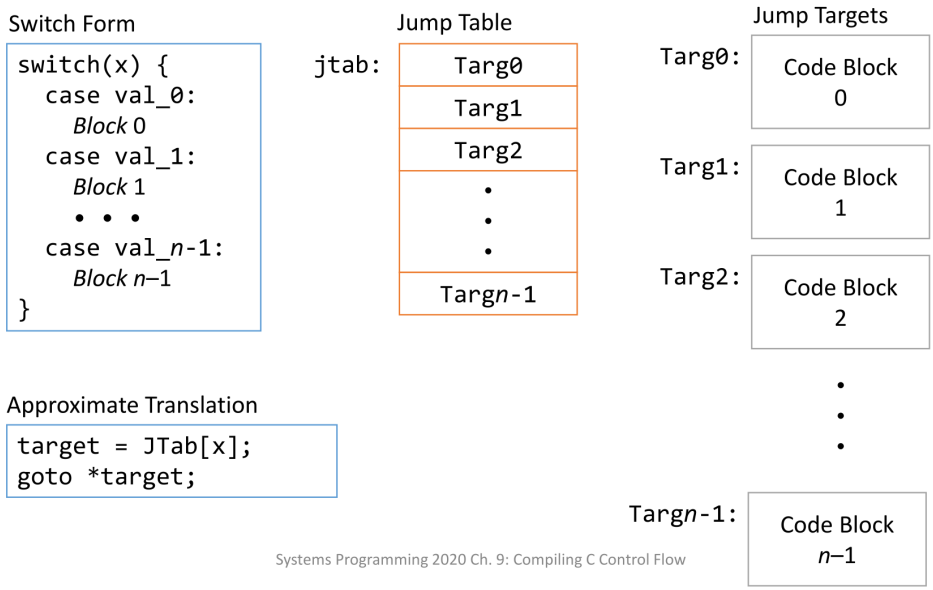
`.global` means that the scope of the following name is global. .type tells the type of a variable/-function.

0.11 Lecture 11: Continue C Control Flow

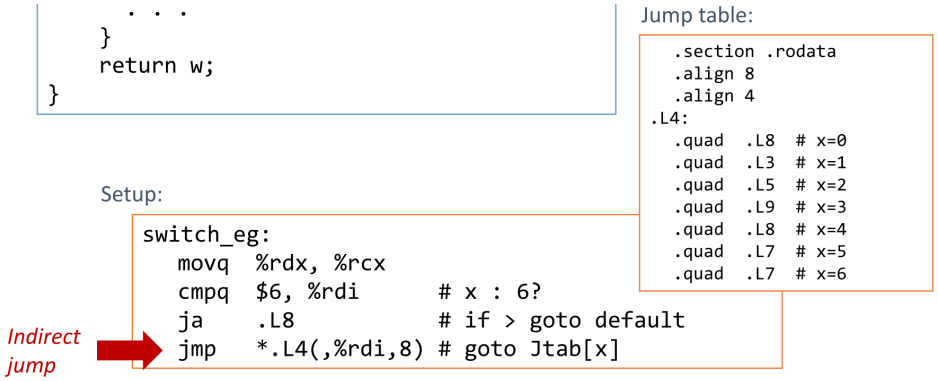
`switch` statement

We could implement a switch as a bunch of `if then else` and `goto` but that gets complicated with fall through and there would be very many comparisons. So compilers do not to that. Firstly, they differentiate between dense and spares `swtich` statements.

Dense `switch` In a dense `switch` the values of `x` lie close together. For each possible case block the compiler creates an entry in the jump table. This includes cases, which land at the default. We can regard the jump table as an array which gives us for a given `x` the address to the jump target, which we want to execute in this case.



The usage of the jump table makes the actual switch statement extremely compact. It actually only consists of a comparison, a conditional jump to the default block, and a indirect jump, via the jump table, to the designated jump target. This indirect jump could be implemented as `jmp *.L4(,%rdi, 8)`, where `.L4` is the base address and `%rdi` holds our `x`. We scale by 8 since labels are 64 bit on x86-64.

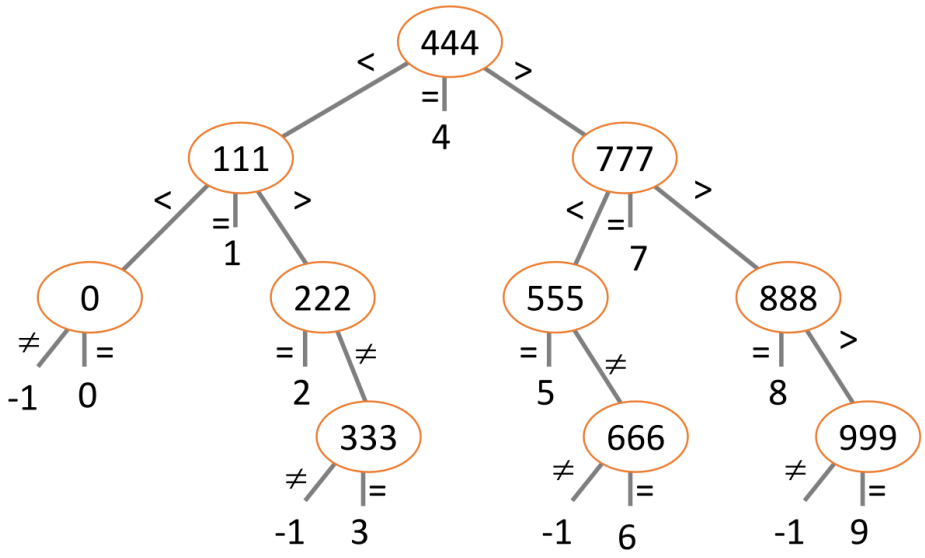


The jump table does not seem to be in an order, that is because the compiler orders in in a way that it is most efficient.

Jump table is just data, so it does not show up in assembly or disassembled code. Nevertheless, we

can inspect it using `gdb`.

Spare `switch` When there are many cases which should result in the default action, it is often too expensive to create a jump table for all these default entries. In these cases, compiler do create an binary three with three actions per node. A less than, equal and greater than. This yields logarithmic performance. In assembly this is achieved by a series of conditional jumps.



The compile are very smart about when to use a jump table and then to rely on a search tree. We should trust the compiler in choosing the most suitable type. They may even create combinations of both types.

Procedure Calls and Return

Stack The stack is in our memory space. It starts at the top and grows downwards to lower adressess. `%rsp` is the stack pointer which always points to the top (lowest address) of the stack. The stack provides the following typical CISC instructions:

`pushl Src`**:** Fetches the operand at `Src`, decrements the stack pointer by 4 and writes the operand at to `%rsp`.

`popl Dest`**:** Reads the operand at the address pointed to by `%rsp`, increments the stack pointer by 4 and writes the operand to `Dest`.

These two operations are not that often used on x86-64. Even though they are very slim instructions, they are often replaced by other read and writes to relative addresses of the stack pointer.

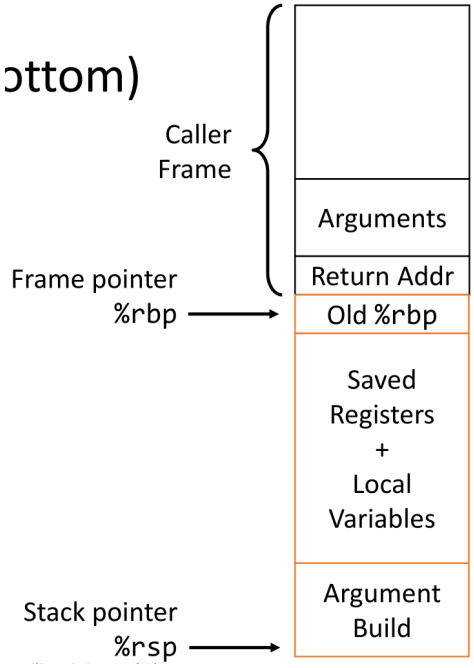
Procedure Control Flow The stack is vital for procedure calls and returns. It is used to hold arguments, return values as well as return addresses.

In order to call a procedure, we call `call label`. This instruction pushed the return address on the stack and then jumps to `label`. The return address it vital in order to get back to the right place after the procedure has finished. In order to prevent the procedure to be called again after returning, the return address has to actually point on the address following the call instruction.

The instruction `ret` is called from a procedure. It pops the return address from the stack and jumps to this address.

There is no need to return after a call. This is in essence simple jump. RISC systems save the return address in a register and do not use a stack. This is called a *wheeler jump*.

Stack Frame The stack frame is the current status of the stack. Generally speaking, it has the following structure.



The *current stack* is the part between stack pointer and the *frame pointer* `%rbp`. The frame pointer points to the start of the current frame (frame contains everything related to the current procedure). At the top there are the *Argument build*. It is actually not that often used nowadays. There we prepare the arguments for the function we are going to call. On top of that, there are the *local variables*, which is data which we cannot keep in a register as well as the saved register content. And at the top of the current stack frame, appointed by the frame pointer, there is the `%rbp` stored, of the previous frame.

Above, we find the caller's stack frame. It contains the return address and arguments passed to the call of the current procedure.

Calling Convection

There are certain conventions to which one should stick when calling procedures, and working with registers and the stack.

How do you construction procedure calls using these utilities.

Register Saving Conventions When one procedure calls another, then the one which initiates the calls ins the *caller* and the one being called is the *callee*. It has to be defined, which register who may change one which should stay untouched. There are the following two register types:

Caller Saved: These registers may be overwritten by the callee and therefore, it is the caller's responsibility to save them to their stack in case it needs them after the procedure returns.

Callee Saved: These registers are long-lived and should not be overwritten by the callee. Hence, when using them they have to restore then before returning.

%rax	Return value, # varargs	%r8	Argument #5
%rbx	Callee saved; base ptr	%r9	Argument #6
%rcx	Argument #4	%r10	Static chain ptr
%rdx	Argument #3 (& 2nd return)	%r11	Used for linking
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved; frame ptr	%r15	Callee saved

Arguments are passed to the callee via registers. If more than 6 arguments are passed, the rest is passed via the stack. These registers are also used used as caller-saved and hence, can be altered by the callee.

There are two return registers. This way structs of size less than 128 bits can be returned. If they exceed this size, they are put on the stack.

All in all, genreal purpose registers are not actually not really general purpose.

Working with the Stack and Stack Pointer All references to the stack are via the stack pointer. When the compiler adds several values to the stack (e.g. to restore callee-saved registers) it often adds then relative to the stack pointer, and decrements the stack pointer only after adding all elements. This is done for optimisation reason since the stack pointer is not decremented several times. The newly added values, which are not yet actually on the stack (stack pointer not yet updated) are in the so called *red zone*.

Optimisation actions as such may break some conventions however, since it does not influence other parts, it is actually acceptable.

Other tricks are to not use the stack, to not use any callee-saved registers. In a *tail call* we no not even push the current address to the stack when calling. This is useful when nested functions return (i.e. there is no need to return the inner function, just jump to the outer function which then returns). This prevents stack overflow for recursive function calls. All in all, often the stack frame is not very heavily used.

0.12 Lecture 12: Compiling C Data Structure

Week 6

This chapter is important because it tells us what happens in memory.

Recap: Basic Data Types

The fundamental data types on C are integers and floating-point values. They are both stored and operated on in dedicated integer respectively floating point registers. For integers, we differentiate between signed and unsigned and the interpretation depends on the used instruction.

The following table summarizes their size:

Intel	GAS	Bytes	C
byte	n	1	signed/unsigned char
word	w	2	signed/unsigned short
double word	l	4	signed/unsigned int
quad word	q	8	signed/unsigned long int (x86-64)
single	s	4	float
double	l	8	double
extended	t	10/12/16	long double

GAS stands for Gnu Assembly Format.

Pointers used to be 4 bytes long on ia32, on x86-64 they are 8 bytes long.

Arrays

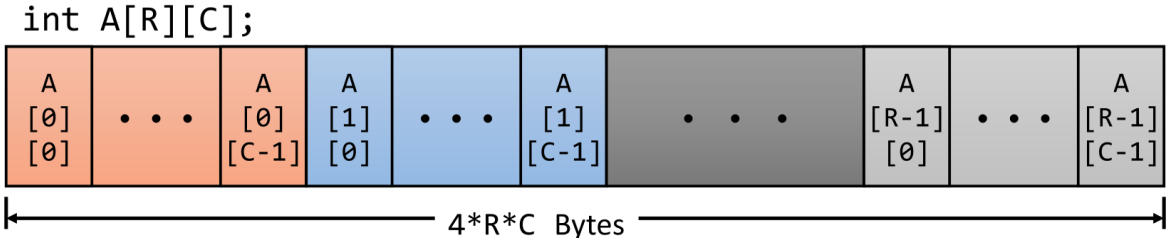
One-Dimensional Arrays One dimension arrays are blocks of homogeneous data structured contiguously in memory. The type of data it contains, determines the size of a single element. The length of the array is the size of each single block times the length of the array.

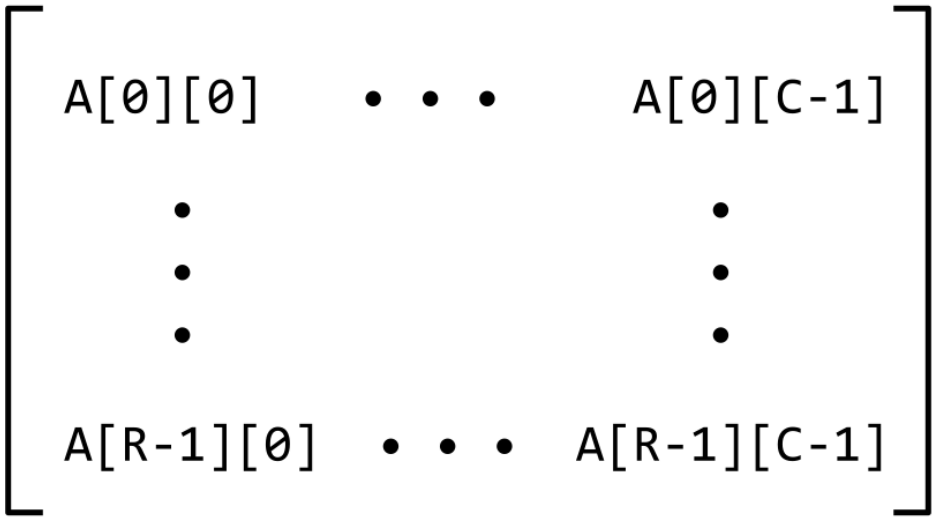
Since there is no out-of-bounds check for array, we may access random data when indexing an element which is not part of the array. When we define multiple arrays, they may be placed contiguously in memory and this way we may access the second array via the pointer of the first array. However, this is not guaranteed to works and hence, should be omitted. Compiler may arrange data in memory however they like. Compiler may also notice out-of-bound accesses and remove such instructions.

When iterating over an array and accessing its elements, compiles may eliminate loop variables and instead use pointers.

Nested Arrays Nested arrays are multidimensional arrays. Since C is row-major, row elements are stored contiguously and all the single rows itself again.

They are declared as `T A[R][C];` and in this example, the array would be 2D and contains R rows and C columns. Hence, the array would be of size `R * C * sizeof(T)`.

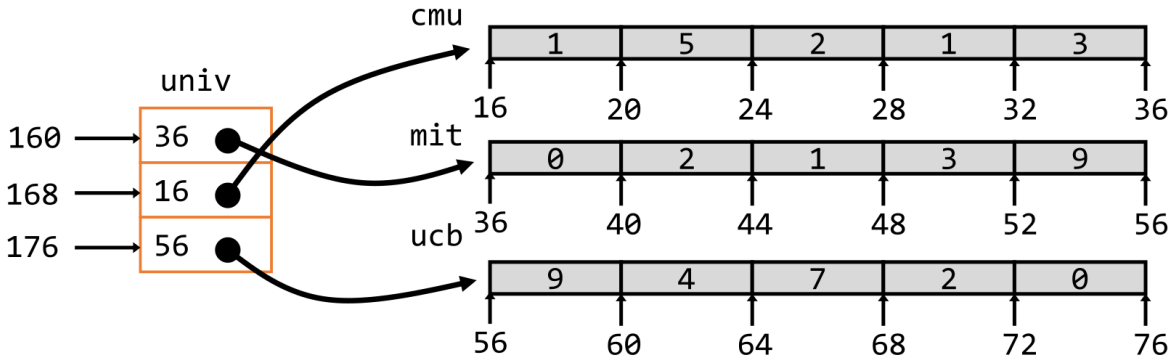




Since all rows are arranged contiguously, it is guaranteed that we can index into other rows. The access of any element, e.g. `<row>, <column>` is possible with only one memory access with the formula $base\_addr + row * column\_len * sizeof(T) + column * sizeof(T) = base\_addr + (row * column\_len + column) * sizeof(T)$.

However, there can be *weird* cases where this is not guaranteed (not exam relevant...).

Multi Level Array Multi level arrays are constructed of several separate arrays. It is actually rather an array of pointers to array and it is also declared as `int *A[R]`. So, the rows of the array are not guaranteed to be placed contiguously in memory and therefore, one cannot access any element of the array by adding an offset to the base pointer. But instead, we have to first access the *pointer array*, and then access the desired element. This requires two memory accesses compared to only one for nested arrays. For arrays of higher dimension, even more memory accesses are required.



Arrays for Matrix Multiplication Compilers are good at doubly subscripted array accesses. However, in order to implement represent an matrix with nested arrays, its size needs to be known at compile time. When we want it more dynamically, we may implemented the *nested array structure ourself* by heaving a block of memory and do all index computation by ourself. Since we need to do many multiplications and access of single elements are therefore rather costly, this is not very performant. Nevertheless, compiler are actually quite good ad improving this solution since there are many regular patters for e.g. a matrix multiplication.

Structures

Stucts are a contiguously-allocated region of memory. The values are ordered contiguously in the same order as their are defined. Since they are arrange contiguously, we can theoretically access a subsequent element via the address of a previous element.

Alignment For performance reasons, data is always aligned in memory. If it ways not, maybe multiple memory accesses may be required to read or write instead of just one, because memory is accessed in junks of 4 or 8 bytes (system dependant).

For x86-64 we have to following alignment cases:

1 byte: char

2 bytes: short

- lowest 1 bit of address must be 0_2

4 bytes: int, float

- lowest 2 bits of address must be 00_2

8 bytes: double, char, *

- lowest 3 bits of address must be 000_2

16 bytes: long double

- lowest 3 bits of address must be 000_2 (treated equivalently to 8 byte aligned types)

In short this means that if a primitive data type requires K bytes, it is always aligned on a multiple of K.

In order that the data fields in the strut are aligned, the compiler may add padding between fields. Therefore, it is also not guaranteed that we can access a subsequent field by simple adding some padding. The placement of the struct itself is determined by the sizes of its fields. If K is the alignment of the largest field, then the struct is aligned at a multiple of K.

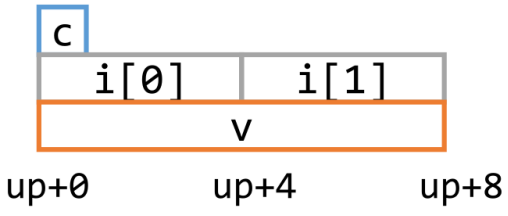
It may be optimal to define files of larger types first, but since the whole struct must be aligned in a multiple of the size of the largest fields, in practice there is not much gain.

Arrays of Structures Stucts in arrays have to follow the alignment requirements too. This may lead to lots of unused padding space.

Union

The size of a union is simply the size of the largest field. The space for the fields is *overlayed*.

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



0.13 Lecture 13: Linking

When a project in split into multiple files `.c`, they are compiled independently into object files `.o`. The linker is then responsible for combining all object files into the final executable. The Unix linker is called `ld` but since gcc is actually a wrapper of many programs, it does also the linking part.

When we compile a file using ggc, the `-g` flag makes sure that debugging information remains in the executable.

Why Linkers

Modularity: Linkers allow to break a project into many small pieces on which one can work independently. Furthermore, linkers allow to build libraries.

Efficientcy: Linkers allow to write efficient code in two ways:

- Compiling takes time. Then only one file changes, it would be wasteful to recompile also the unchanged files.
- We can put function which may be reused in a dedicated file. This allow for simple reuse as well as lower memory usage when loading this function, since the file only contains code we actually need.

Modularity

What does a Linker do?

Linking is a two-step process:

1. **Symbol Resolution:** Programs define and reference symbols. Symbols are function names and variables. The symbols definitions get stored into the symbol table by the compiler. This is an array of stucts where each entry includes symbol name, type, size and location. The linker then associates each symbol reference with exactly one symbol definition.
2. **Reallocation:** The linker merges separate code and data section into a single section. This requires the reallocation of the symbols from their relative location in the `.o` files to their final absolute memory location in the executable.

Object Files

There exist three different kinds of object files (modules):

Relocatable Object Files They have the extension `.o` and each single file is the product of a compiled source file. They contain code and data which is in a format, such that it can be combined with other relocatable object files to form an executable.

Executable Object Files They contain code and data which can be directly copied to memory and executed.

Shared Object File They have the `.so` extension. These are special object files which have the property, that they can be loaded and linked dynamically at runtime, into memory.

Executable and Linkable Format (ELF) All mentioned object files types share the same format. The ELF is the standard finery format for object files and the generic name for such object files is ELF binary.

A section referees to bits of a file where segments referees to bits in memory.

The general format is as follows:

Elf Header: Tells that this file is in the ELF format as well as provides information about endianess, word size, machine type, file type etc.

Segment Header Table: Contains information about the page size, virtual address memory segments (where is stack, heap etc), segment size.

.text section: Machine code given by the compiler.

.rodata section: Read-only data like the jump-table.

.data section: Initialized global variables. The last segment tells where the .bss section starts.

.bss section: Uninitialized global variables. Since they have no value, it is normally smaller than the data segment. Called *Block Started by Symbol* or *Better Save Space*.

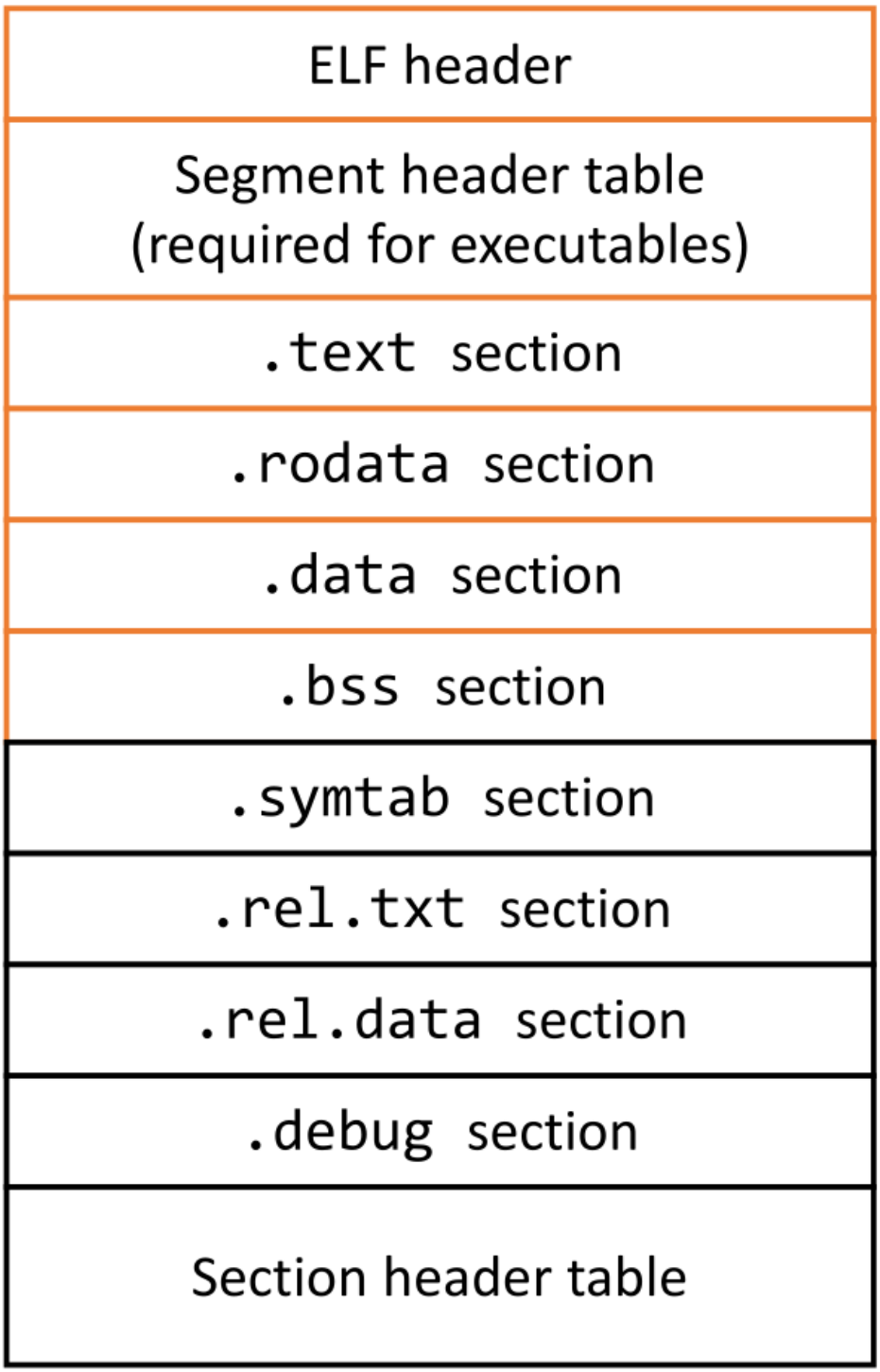
.symtab section: Symbol table and the names of procedures and static variables.

.rel.text section: Reallocation information for .text section (addresses of instruction which need to be modified as well as instruction for modifying).

.rel.data section: Reallocation information for .data section (addresses of pointer data which need to be modified).

.debug section: Information fo symbolic debugging with `gcc -g`.

Section Header Table: Offset and sized of each section.



Linker Symbols

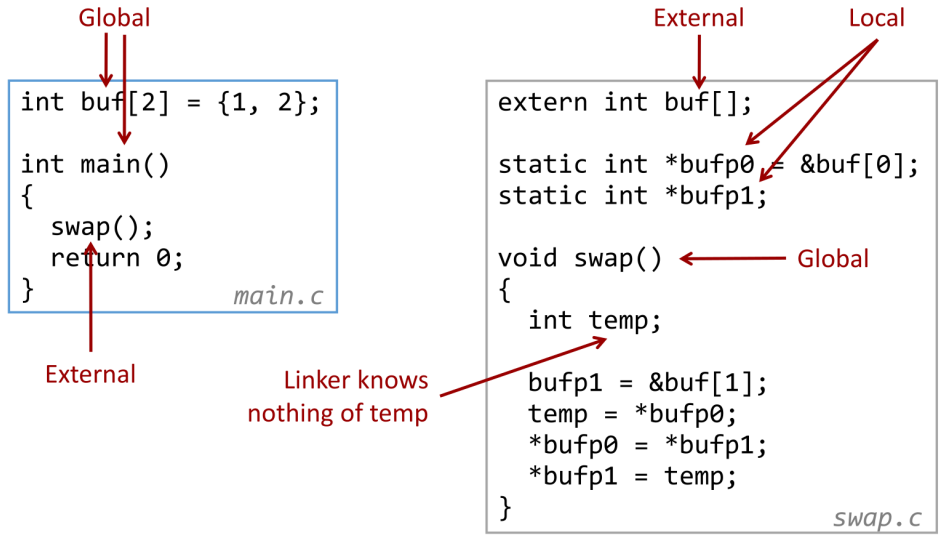
There are three types of symbols:

Global Symbols Symbols which are defined as globals in a certain module.

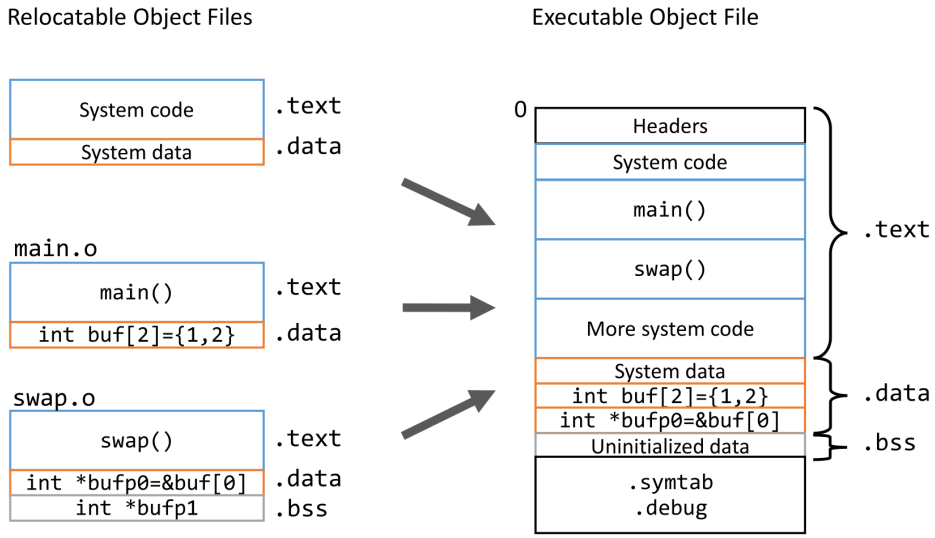
External Symbols Symbols which are referenced in a certain module, but defined in another module. One can prepend external symbols by `extern`. This is not required, nevertheless, one should always do it for good practice.

Local Symbols Symbols which are defined and referenced only in a certain module. However, they are **not** local program variables. Linkers are not aware of any local program variables.

Example Linker Given the following code, we can find the symbols and their type:



The linker combines them as follows into an executable. The system code and data is stuff the compiler sees but we do not.



In the disassembled .tex section of main.o we see, that the compiler does not know where swap is. Instead, it inserted an entry into the symbol table (the string in read). String tells that at address a there are 4 bytes which need to be filled with the address of the function called `swap`. The `R_X86_64_PC32` means it is a reallocation entry for a x86-32 processor which used pc relative 32 addressing. This information allows the linker to fill in the right address.

```
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
```

The disassembled .data section we can see the global two integer.

```
Disassembly of section .data:

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

The first movq instruction has again two 4 byte long addresses of zero. The first address is again a PC relative 32 offset reallocation instruction. It is the unallocated `bufp1` symbol and hence, it is referred to as `.bss-0x8`. The second address is again a reallocation entry but it is a 32 bit scalar value, which is referred too as `buf+0x4`.

```
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
```

In the data section we find the `buf`. In the bss section we find the `bufp1` symbol. it has 8 bytes, but they are actually not in the file, it is just recorded that they should be there.

```
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
```

The linker takes the previously discussed files, and puts them into a single executable. The .text section looks as follows. The file looks very familiar, but the missing addresses were filled in (the linker reallocated the addresses).

```
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
```

And the .data and .bss. We notice that all addresses are only 8 bytes long. The reason is in order to save space. The compiler uses relative addresses and the linker figures out what this address is.

```
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
```

Strong and Weak Symbols

Strong: Procedures and initialized globals

Weak: Uninitialized globals

Linker's Symbol Rules Then multiple symbols with the same name are given the linker proceeds as follows:

1. Multiple string symbols are not allowed. This leads to a linker error.
2. When given a strong and multiple weak symbols, the liner always chooses string symbol. So, references to the weak symbol resolve to the strong one.
3. When given multiple weak symbols, the linker picks one arbitrary. This can lead to weird problems. E.g. when there are two structs with the same name and same size, but different fields (one has two ints, the other one double), the linker may select one at random and when writing to e.g. the address of the first int, we may be dealing with a double of the size of both ints. This problem may also arise when we combine code which was compiled by different compiler which have different alignment rules.

Some Puzzles Some Linking Puzzles:

`int x;` `int p1() {}`	`int p1() {}`	Link time error: two strong symbols (p1)
`int x;` `int p1() {}`	`int x;` `int p2() {}`	References to x will refer to the same uninitialized int. Is this what you really want?

`int x;`
`int y;`
`int p1() {}`
	`double x;`
	`int p2() {}`
		Writes to x in p2 might overwrite y!
		Evil!

`int x=7;`
`int y=5;`
`int p1() {}`
	`double x;`
	`int p2() {}`
		Writes to x in p2 will overwrite y!
		Nasty!

`int x=7;` `int p1() {}`	`int x;` `int p2() {}`	References to x will refer to the same initialized variable.

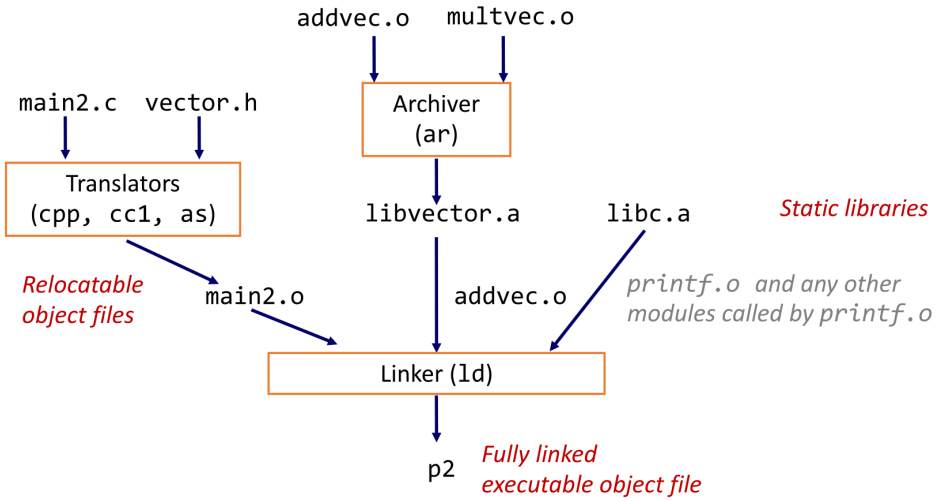
Global Variables Globals can cause many issues. Besides the already disused ones, it may also quickly happen that different programs declare globals with the same name. Good practice is to avoid them when possible. If not, one should at least initialized them. An when using an external global, one should always prepend it by `extern`.

Static Libraries

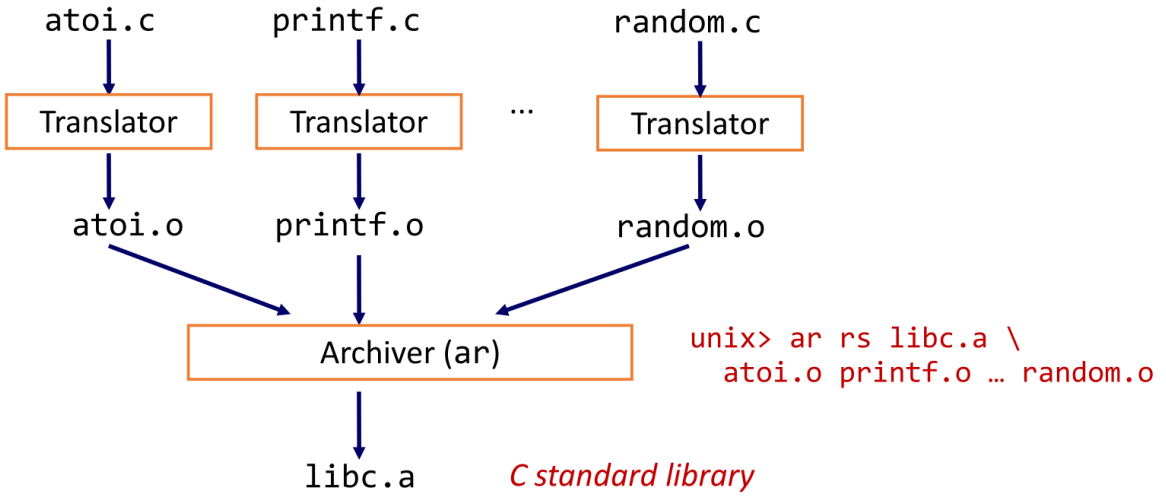
We may want to package commonly used functions into a library. Trivial ways to accomplish that is by putting all function into a single source file or by putting each function into a separate file. Both methods have some downsides. While the first methods is very space and time inefficient, the second methods puts a burden on the programmer to link the appropriate binaries into their programs.

Static libraries solve this problem. They have the extension `.a` and the concatenate related relocatable object files into a single archive file. The linker tries to resolve unresolved external references by looking for that symbol in one or multiple archived. On success, the linker links the object file into the executable.

The following diagram shows the usage of static libraries:



Build Archives The archiver `ar` is used to create such archives:



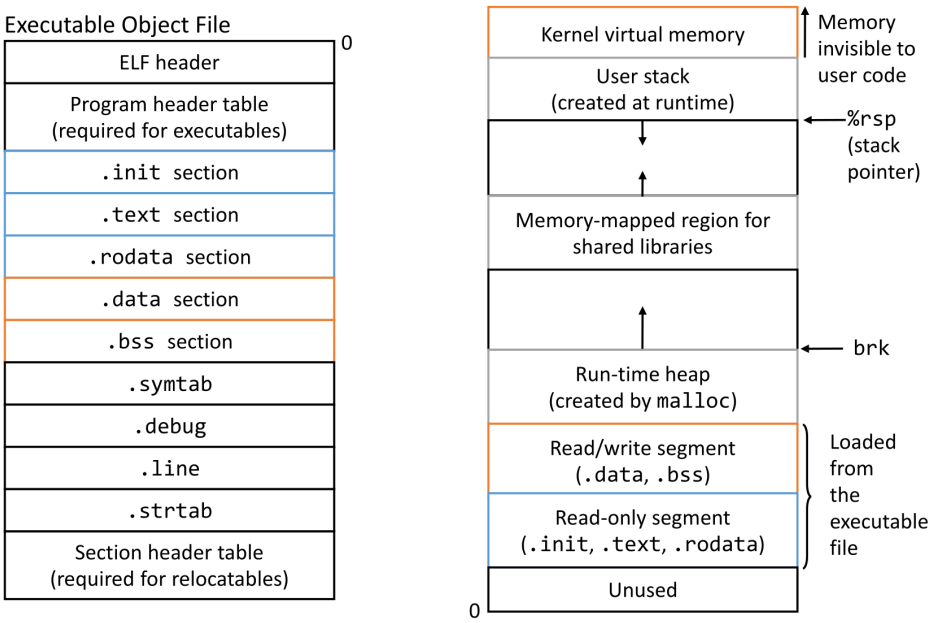
The archiver allows incremental updates. This means that when a function changes, only the source of that file is recompilation and the .o file is replaced in the archive.

How does the linker know where these archives are? There are certain defaults directories. To include certain libraries, it is also possible to load it with `gcc -lm` where `m` is the name of the library (in this case libm (the math library)).

How Linker Resolve External References The linker scans the .o files and .a archives in the ordre they were provided to the command. During the scan, it weeks a list of currently unresolved reference. Always when a new archive .a or object .o file is encountered, it tries resolve unresolved reference and updates the unresolved references list. If at the end of this process there are unresolved reference, this leads to an error.

The main problem is, that the linker does only one pass of this algorithm. Since files are scanned in the order they are provided, one should put libraries at the end of the command. E.g. `gcc -L. libtest.o -lmine` instead of `gcc -L. -lmine libtest.o`.

Loading an Executable Object File When loading an executable, different sections are put into corresponding segments in memory. It used to be the way, the executables would be put directly into memory in reversed order.



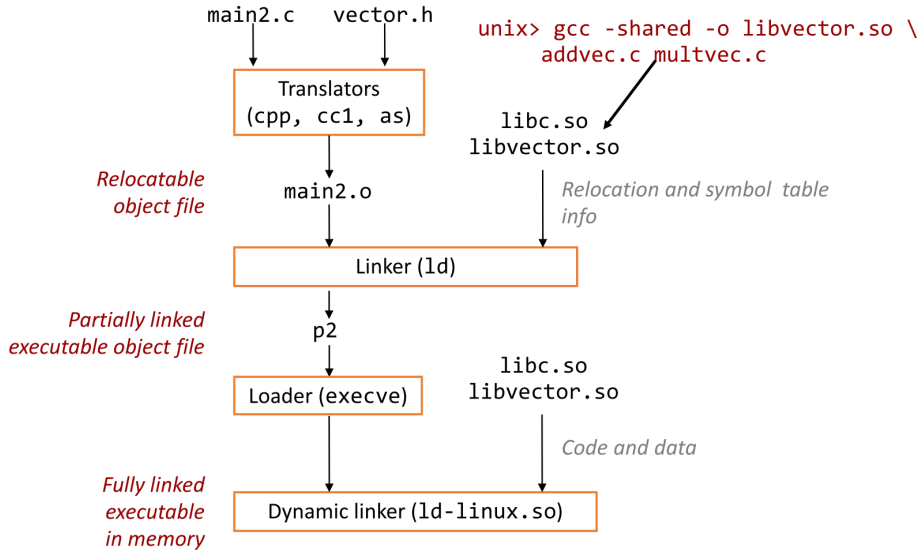
Shared Libraries

Since when using static libraries, all code is copied into the executable, one has to recompile the program, when the library is updated. Further, the code is always duplicated. Therefore, static libraries are only actually used during the building process and shared libraries in production.

Shared libraries provide object files containing code and data which get linked into a program dynamically during load-time or run-time.

Shared libraries are also called dynamic link libraries (DLLs) and they can be shared between multiple processes.

Load-Time Linking is automatically handled by the dynamic linker `ld-linux.so`. The following graphic depicts the procedure:



Run-Time Linking is done by calls to `dlopen()` interface.

```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);
    }
    ...
```

```c
    ...

    /* 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;
}
```

main+addr is used then the location of the function is unknown. The linker is responsible to fill in the missing address at address a. The red string is stored in the symbol table.

.bss-0x8 stands for the rip we do not know. The buf+0x4 is a scalar value and is the

compiling p2 alone, we get something in the .bss called foo compiling p1 alone, we get something in the .data called foo

if the two files are linked together, we only get one foo, and this is the p1 because the linkers sees assumes that the foos are equivalent

0.14 Lecture 14: Code Vulnerabilities

Worms And Viruses

Both viruses and worms are designed to spread among computers, network and drives. Though they are fundamentally different:

Worm: is a program than runs by itself and which is capable to propagate a fully working version of itself to other computers.

Virus: is code which adds itself to other programs ant it cannot run without its *host* program.

In the early days of computers, worms were actually used in distributes systems for e.g. update purpose. The first *malicious* worms was written in 1988 by Robert Morris and was designed to count the number of hosts on a network.

Stack Overflow Bugs

This is one of the simplest code vulnerability bugs one can exploit.

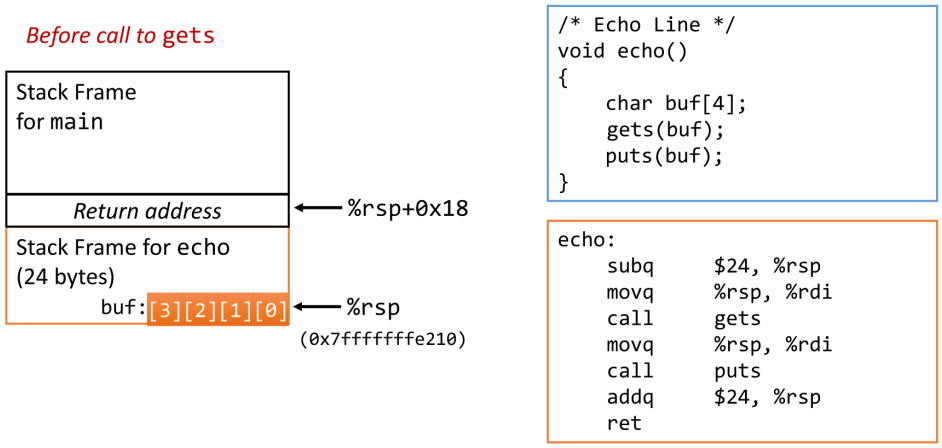
`gets` is a C functions which ready input till it receives an *end-of-file* character and stores them in an array designated by a pointer. The problem with this early implementation was, that there is no way to specify the maximal number of characters to read.

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

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

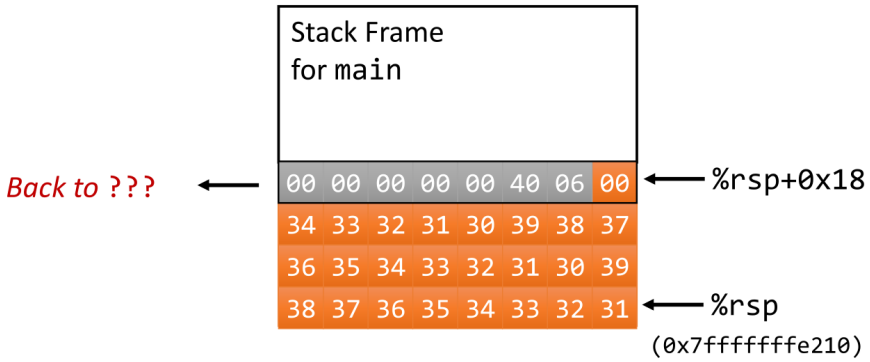
The previous code works perfectly for inputs of length 3 (the 4-th byte is required for the end character) or maybe even for a few more bytes, but when we input a large amount of characters we get a segmentation fault.

A stack frame of a certain size gets initialized and the buf variable is stored within in.

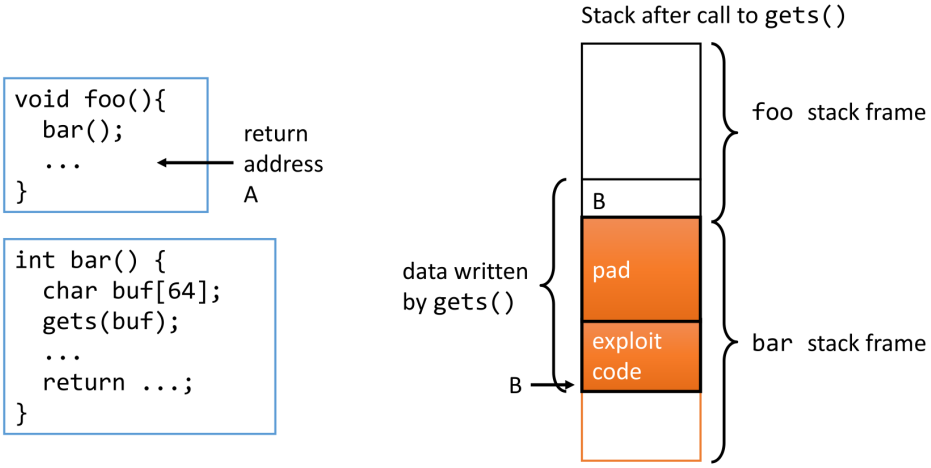


As long as we only overwrite data in the stack frame, this may not cause direct problems, when when the return address gets overwritten, we will receive a segmentation fault.

Input: `1234567901234567901234`



Instead of simply crashing the program, we may make malicious use of this vulnerability. My overwriting the return address of another stack frame and make it point to some exploit code which we have pushed to the stack via gets.



There used to be a server called `fingered` which used `gets`. Instead of calling it like `finger username@addres`,

one could execute something like `finger "exploit code padding new-return-address"` which then gets executed on the root shell.

Prevent Stack Overflow Issues There are several ways to prevent these issues. One should for example always use the more modern and robust functions. I.e. use `fgets` instead of `gets`, `strncpy` instead of `strcpy` and `fgets` instead of `scanf` for reading strings (or use `%ns` instead if `%s`).

Further, one should also always take care when defining these limits. It is actually really hard to get them right. It can easily happen that the end character is forgotten.

Compilers do have a notion about libraries (which is not only something positive...) and may detect such issues. For experimentation purpose we can disable this feature using the flags `-D_FORTIFY_SOURCE=0` and `-fno-stack-protector`.

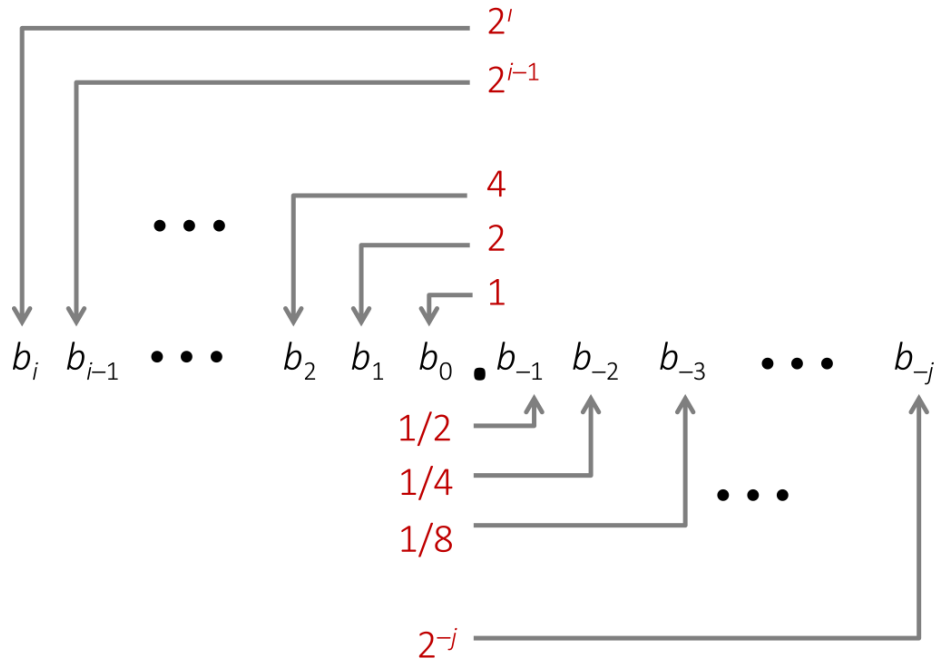
In order to make it more difficult for a attacker to guess the right address, stack offsets are initialized at random. Furthermore, some CPUs can make certain parts of the memory read- or write only. Or they can make the stack non-executable, meaning that no code on the stack can be executed. However, we can still use e.g. the heap for exploit code and therefore, this is not really a fix.

All in all, even though this problem is mostly fixed, it still pops up from time to time.

Floaging Point

Representing Floating-Point Numbers

Fractional binary numbers can be very intuitively represented as $\sum_{k=-j}^{i} b_k \cdot 2^k$. All bits to the right of some *binary point* represent fractions of powers of 2.



As we are used to from integers, a right shift represents a division of 2 and a left shift a multiplication by 2.

The problem with this representation is that, we can only exactly represent numbers which are of the form $\frac{x}{2^k}$. In order to represent any number exactly we would need infinitely many decimal points.

IEEE Floating Point In the early days, there was no standard on how to represent binary numbers, and all CPU manufactures implemented their own format. In 1985 the first standard for floating point arithmetic was introduced. It was supported by all major CPUs.

Floating point numbers are represented in the form $(-1)^S \cdot M \cdot 2^E$.

S: sign bit determines whether the number is positive or negative.

M: is the significant and it is a fractional value in the range $)1.0, 2.0)$.

E: exponent weight.

The M and E are not directly stored as twos complement, but encoded in a specific was. This allows the format to represent a much wider range. The encoding is as follows:

s: directly represents the S and is the MSB of the encoding.

exp: is the encoding of E

frac: is the encoding of M

The length of the exp and frac section is determined by the used Standard.

Precision	Significand 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*

There used to be the two single and double precision standards. Over time the total length got bigger. But only recently a few other, much shorter non-standards, but widely used ones, were introduced. They are used mainly for machine learning. One found out that it is worth trading precision for efficiently.

Floating Point in C In C99, float (single precision) and double (double precision) are guaranteed to have a certain size independent of the machine. Other types vary in length.

When casting one type to another, the bit representation changes. This is very different from the signed/unsigned integer representation, where only the interpretation changed.

double/float → int: truncates fractional parts which is equivalent to a rounding towards zero. The behaviour is undefined when out of range.

int → double: As long as the int has less that ≤ 53 bit word size, the conversation is exact.

int → float: Will round according to the rounding mode.

Floating point numbers are einher normalized, or denormalized:

Normalized Values

- This are numbers, where $exp \neq 000\ldots0$ and exp $\neq 111\ldots1$.
- E is encoded as $E = Exp - Bias$

  Exp: is the unsigned value of exp

  Bias: is equal $2^{e-1} - 1$, where e is the number of exponent bits.

  Singe Percision: $Bias = 127, Exp \in [1, 254], E \in [-126, 127]$

  Double Perc.: $Bias = 1023, Exp \in [1, 2046], E \in [-1022, 1023]$
- M is equal to $1.xxx\ldots x_2$ where $xxx\ldots x$ are the bits of frac.
  - Minimum when $000\ldots0 \implies M = 1.0$
  - Maximum when $111\ldots1 \implies M = 2.0 - \epsilon$

Example Normalized encoding of 15213.0:

- Value: `float F = 15213.0;`
  15213_{10} = 11101101101101_2
  = $1.1101101101101_2 \times 2^{13}$
- Significand
  - *M* = 1.1101101101101_2
  - frac = $1101101101101\underline{0000000000}_2$
- Exponent
  - *E* = 13
  - *Bias* = 127
  - *Exp* = 140 = 10001100_2
- Result:

s	exp	frac
0	10001100	11011011011010000000000

Denormalized Values

- These are numbers, where $exp = 000\ldots0$.
- E is encoded as $E = -Bias + 1$

  Bias: is equal $2^{e-1} - 1$, where e is the number of exponent bits.

  Singe Percision: $Bias = 127, Exp \in [1, 254], E \in [-126, 127]$

  Double Perc.: $Bias = 1023, Exp \in [1, 2046], E \in [-1022, 1023]$
- M is equal to $0.xxx\ldots x_2$ where $xxx\ldots x$ are the bits of frac.

When $exp = 000\ldots0$ and one if $frac = 000\ldots0$, the number is zero. Depending on s, we have actually -0 and $+0$. This is actually useful to distinguish between convergence from below and from above.

When $exp = 000\ldots0$ and $frac \neq 000\ldots0$ we represent numbers very close to 0.0. Such numbers are *equispaced*. Also we notice, that we lose precision the smaller the number gets.

Interesting Numbers This table show some interesting floating point numbers.

`{single,double}`

Description	exp	frac	Numerical value
Zero	00…00	00…00	0.0
Smallest pos. denorm. • Single ≈ 1.4×10^{-45} • Double ≈ 4.9×10^{-324}	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
Largest denormalized • Single ≈ 1.18×10^{-38} • Double ≈ 2.2×10^{-308}	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
Smallest pos. normalized • Just larger than largest denormalized	00…01	00…00	$2^{-\{126,1022\}}$
One	01…11	00…00	1.0
Largest normalized • Single ≈ 3.4×10^{38} • Double ≈ 1.8×10^{308}	11…10	11…11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$

Special Values The description above does not handle the case when $exp = 111\ldots1$. This value is used as follows:

exp $= 111\ldots1$, **frac** $= 000\ldots0$**:** Represents $+\infty$ and $-\infty$ as well as operation overflow.

exp $= 111\ldots1$, **frac** $\neq 000\ldots0$**:** Represents NaN, i.e. cases where no numerical value can be determined.

Spacial Properties

- The floating point +0 is zero for all bits. This is equivalent for the integer zero.
- The allow comparison with unsigned integer with the cravats:
  - Compare sign bits
  - Consider $-0 = 0$
  - Nan is problematic since what should the comparison yield?

Floating-Point Ranges

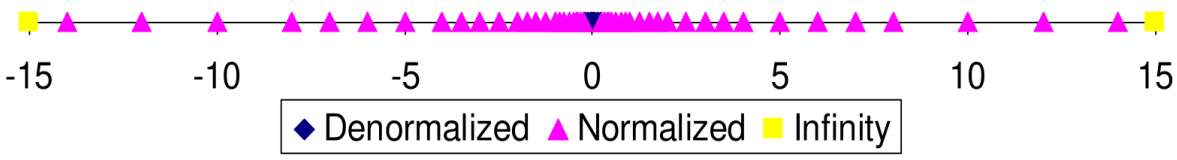
For a 8-bit floating point representation with 1 bit for s, 4 bits for exp and 3 bits for frac

this table shows the positive range of our 8-bit number

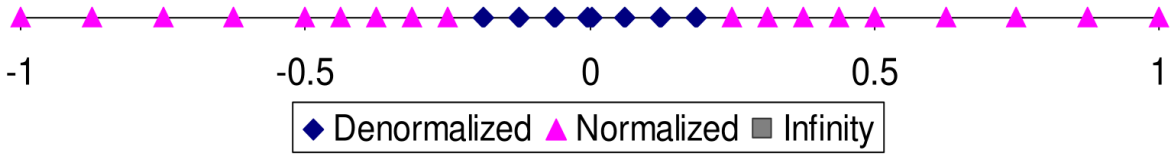
	s exp frac	E	Value	
Denormalized numbers	0 0000 000	-6	0	
	0 0000 001	-6	1/8*1/64 = 1/512	closest to zero
	0 0000 010	-6	2/8*1/64 = 2/512	
	…			
	0 0000 110	-6	6/8*1/64 = 6/512	
	0 0000 111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0 0001 000	-6	8/8*1/64 = 8/512	smallest norm
	0 0001 001	-6	9/8*1/64 = 9/512	
	…			
	0 0110 110	-1	14/8*1/2 = 14/16	
	0 0110 111	-1	15/8*1/2 = 15/16	closest to 1 below
	0 0111 000	0	8/8*1 = 1	
	0 0111 001	0	9/8*1 = 9/8	closest to 1 above
	0 0111 010	0	10/8*1 = 10/8	
	…			
	0 1110 110	7	14/8*128 = 224	
	0 1110 111	7	15/8*128 = 240	largest norm
	0 1111 000	n/a	inf	

For a 6-bit floating point representation with 1 bit for s, 3 bits for exp and 2 bits for frac

We can see that the distribution gets denser towards zero



When zooming in, we can see the equispaced denormalized values



0.15 Lecture 15: Floating Point

Week 8

Floating-Point Rounding

The basic idea of rounding is the first compute the exact result, then round it to make it fit into the desired precision.

There are four different rounding modes. All are acceptable IEEE standards, but in C we cannot change the mode without getting into Assembly.

- Towards zero
- Round down ($-\infty$)
- Round up ($+\infty$)
- Nearest Even (default)

All modes, beside round-to-even are biased towards some systematic error. Round-to-even is statistically the most accurate one. It rounds the number in such a ways, that when we are half-way between two numbers, it rounds towards the even number. In all other cases, it rounds towards the closer number.

The following example show rounding to the nearest hundredth

Value	Result	Description
1.2349999	1.23	(less than half way)
1.2350001	1.24	(greater than half way)
1.2350000	1.24	(half-way – round up)
1.2450000	1.24	(half way – round down)

When considering binary numbers we first notice that:

- Number is even if LSB is 0
- Number is half way between to numbers when the bits to the right of the rounding position are $100\ldots_2$

In the following example we round towards the nearest 1/4 (2 bit positions after the binary point)

Value	Binary	Rounded	Action	Result
$2\,{}^{3}/_{32}$	10.00011_2	10.00_2	< ½ : down	2
$2\,{}^{3}/_{16}$	10.00110_2	10.01_2	> ½ : up	$2\,{}^{1}/_{4}$
$2\,{}^{7}/_{8}$	10.11100_2	11.00_2	= ½ : up	3
$2\,{}^{5}/_{8}$	10.10100_2	10.10_2	= ½ : down	2 ½

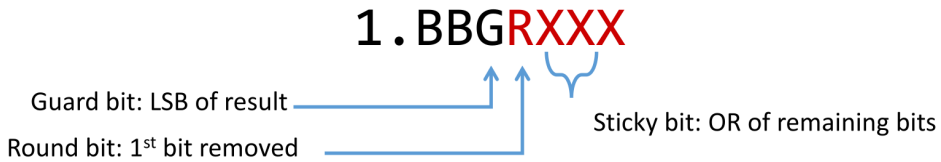
Building circuits for rounding is rather easy.

Creating floating point number Before considering rounding in arithmetic in floating point notation, we have a look how to convert a given unsigned binary number into a floating point number. This comprises of the following steps:

Normalize In the first step we make sure that the binary number is of the form $1.xxx\ldots$. We drop all leading 0 and shift the number to the left, while incrementing the exponent, till we have our desired format.

Value	Binary	Fraction	Exponent
128	10000000	1.0000000	7
15	00001101	1.1010000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

Rounding Assume the frac bit of the floating point number has length 3. Therefore, we have to round the fraction to three digits after the binary point. For that purpose we name the LSB which remains after rounding, the *guard bit*, the MSB which gets remove the *round bit*, and the *sticky bit* is the OR of all other truncated bits.



The condition for the rounding are the following:

- Round $= 1$, Sticky $= 1 \implies > 0.5$ (round up)
- Guard $= 1$, Round $= 1$, Sticky $= 0 \implies$ Round-to-even

Value	Fraction	GRS	Incr?	Rounded
128	1.0000000	000	N	1.000
15	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

Postnormalize Rounding may cause overflow. We fix that by right-shifting once and increasing the exponent.

Value	Rounded	Exp	Adjusted	Result
128	1.000	7		128
15	1.101	3		15
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	1.000/6	64

Floating-Point Addition and Multiplication

Given two number is FP notation $(-1)^{s1}M_1 2^{E1}$ and $(-1)^{s2}M_2 2^{E2}$:

Multiplication The result of the multiplication of the two numbers of $(-1)^s M 2^E$, where:

- $\mathbf{s} = s_1 \wedge s_2$
- $\mathbf{M} = M_1 \cdot M_2$
- $\mathbf{E} = E_1 + E_2$

After the calculation, we have to consider a few cases:

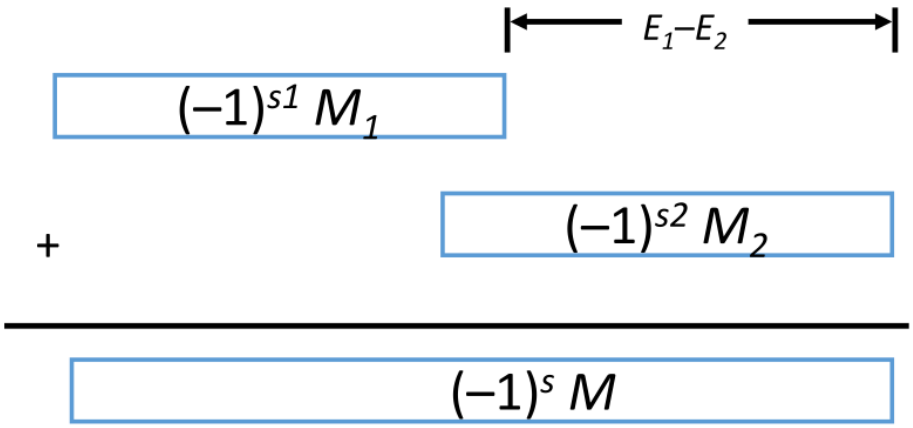
- If $M \geq 2$ (i.e. not of the form $1.xx\ldots_2$), we shift M right and increment E
- If E out of range we overflow
- Round M to find frac precision (as we did when converting unsigned int to FP notation)

Properties

- Closed under multiplication: **Yes**
  - May generate infinity or NaN
- Commutative: **Yes**
- Associative: **No**
  - Overflow and inexactness of rounding
- 1 is multiplicative identity: **Yes**
- Multiplicant distributes over addition: **No**
  - Possibility of overflow and inexactness of rounding
- Monotonicity: **Almost**
  - Except for infinity and Nan

Addition Addition is a little more complex than multiplication because the result can easily overflow. We assume that $E_1 > E_2$ and the result of the addition is $(-1)^s M 2^E$, where:

- $\mathbf{s}$ and M are the result of signed align and addition
- $\mathbf{E} = E_1$



After the calculation, we have to consider a few cases:

- If $M \geq 2$ (i.e. not of the form $1.xx\ldots_2$), we shift M right and increment E
- If $M < 1$ (i.e. not of the form $1.xx\ldots_2$), we shift M left and decrement E
- If E out of range we overflow
- Round M to find frac precision (as we did when converting unsigned int to FP notation)

Properties

- Closed under addition: **Yes**
  - May generate infinity or NaN
- Commutative: **Yes**
- Associative: **No**
  - Overflow and inexactness of rounding
- 0 is additive identity: **Yes**
- Every element has additive inverse: **Almost**
  - Except for infinity and Nan
- Monoticity: **Almost**
  - Except for infinity and Nan

Add smallest numbers first to get the most accurate result.

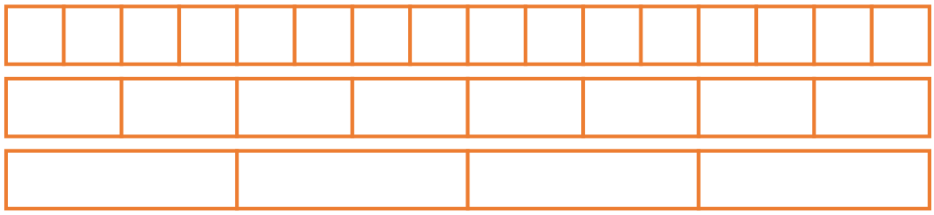
2/3 is float 2/3.0 is double (is is it the other way?)

SSE Floating Point

CPUs have designated modules for floating point arithmetic. In old architectures, the floating point unit was even placed on a different chip(co-processor). All x86-64 have SSE3 (streaming SIMD extension), which is a superset of SSE, and which supports single and double precision. This allows CPUs to conduct vector instructions, meaning, parallel operation on small (length $2-8$ byte) vectors of integers of floats. n-way means that a vector holds n elements.

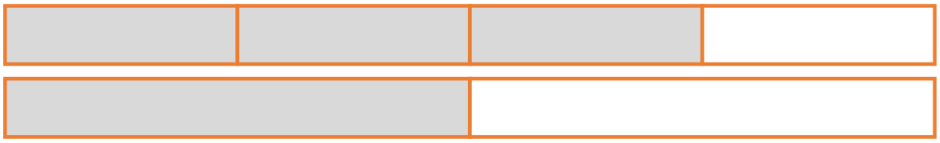
SSE3 Registers SSE3 provides an additional set of 16 registers. They have each size of 128 bits and can be used to store:

- Integer Vector:
  - 16-way byte
  - 8-way 2 bytes
  - 4-way 4 bytes



- Floating Point Vector:
  - 4-way single
  - 2-way double

- Floating Point Scalars (not whole vector is used):
  - single
  - double



They are named `%xmm0` through `%xmm15` and they are all caller saved.

SSE3 Instructions Basic instruction names are composed of three parts:

- The name of the instruction: `add, sub, mov` etc
- The format of the data: `p` for packed (vectors) and `s` for scalar
- The data type of the data: `s` for single precision and `d` for double precision

Moves `movss`, `movsd` are used to move data from reg to reg, reg to mem or mem to reg.

Single	Double	Effect
movss	movsd	D ← S

Arithmetic There exist the following arithmetic instructions:

Single	Double	Effect
addss	addsd	D ← D + S
subss	subsd	D ← D − S
mulss	mulsd	D ← D x S
divss	divsd	D ← D / S
maxss	maxsd	D ← max(D,S)
minss	minsd	D ← min(D,S)
sqrtss	sqrtsd	D ← sqrt(S)

Concessions Are used to convert the type of data. Their format is a little more complicated.

Instruction	Description
cvtss2sd	single → double
cvtsd2ss	double → single
cvtsi2ss	int → single
cvtsi2sd	int → double
cvtsi2ssq	quad int → single
cvtsi2sdq	quad int → double
cvttss2si	single → int (truncation)
cvttsd2si	double → int (truncation)
cvttss2siq	single → quad int (truncation)
cvttsd2siq	double → quad int (truncation)

GCC and SSE3 Since recently, gcc can try to to autovectorize given code. This functionality is very limited and does not guarantee any speed-up. C itself is poorly suited for writing vectorized code, i.e. code for GPUs. But there are dedicated languages for this purpose. Writing vectorized code is very hard but the key to performance.

0.16 Lecture 16: Optimizing Compilers

Week 8

In order to make best of the compiler, it is mandatory to have a good knowledge about what the compiler does, how we can help it it optimize code, and what we should omit to not stop the compiler from optimising.

In practice, not only the asymptotic runtime determines the performance of our code. There are many other very important factors which we have to consider like constants, coding style, algorithm structure, data representation etc.

Optimizing Compilers The compiler can try to optimize our code in four levels, which are set via flags. The compiler does no optimisation when setting `-O0` and does full optimisation for `-O3`. The default is no optimisation, since this provided the highest compilation speed. `-O3` does not necessarily provide the best optimisation in all cases. These flags do in essence nothing than setting a bunch of underlaying compiler flags.

Also, different compiler optimize in different ways. Using the `-march=xxx` flag, we can tell the compiler to optimize code for a very specific architecture.

Compilers Strengths Compilers are good at mapping programs to machines. They know the hardware well and know how code is best adopted to certain hardware. Further they are good at:

- Register allocation
- Code selection and ordering (scheduling)
- Dead code elimination (code which is not reachable)
- Eliminating minor inefficiencies

Compiler Weaknesses Compilers cannot improve asymptotic efficiently. It is up to the programmer to select the best overall algorithm. Furthermore, there are certain things, which block the compiler from optimizing.

Compiler Optimisation Constraints Compilers are conservative if in doubt. They operate under fundamental constraints and must not change program behaviour under any possible condition. This may also lead to cases where the compiler does not do optimisations which are obvious for the programmer and which we expect the compiler to do.

Compiler Optimisation Tricks

In the following we will discuss certain compiler optimisations.

Code Motion Compiler may move code to reduce the frequency or cost of certain computations. The is particularly useful in loops which do a computation which does not change during the execution of the loop. The compiler may move this computation outside the for loop.

This is sometimes also called precomputation.

Strength Reduction Compiler may replace costly operation with simpler ones. These optimisations are very hardware specific. An example if for example the replacement of a multiplication by a shift.

Common Subexpressions Compiler may precompute some common subexpression of a larger computation and prevent the re-computation of the same expression.Tough, we must remember that certain operations are not recursive, and especially for floating-point number this technique may easily change a result. Therefore, compilers are not that good at exploiting this property.

Optimisation Blockers

There a certain things which prevent the compiler from making optimisations.

Procedure Calls Procedure calls are treated as black boxes which the compiler does not touch. Even though the input does not change, it does not mean that the return value does not change. Procedure call may also have other side effects about which the compiler has not understanding. Therefore, the compiler cannot move procedure calls.

When we inline to procedure, the compiler would be able to do more optimisation.

Memory Aliasing In C it is not uncommon to have multiple referenced to the same memory location. Compilers cannot assume that two different points point to different locations and therefore they may interfere each other during access. If we know that pointers do point to different location, we should use local variables, and write these to memory instead of many single accesses.

Clocking and Unrolling This is very relevant for e.g matrix multiplication. All elements are accessed multiple times and we want to make use of the memory locality. A possible solution is blocking (tiling); i.e. compute blocks (/submatrices) at a time. This provides better cache locality. But we need to know the machine well in order to determine the best block size. Another solution is unrolling; this is the reuse of precomputed expressions.

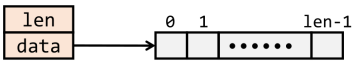
These optimisation we have to do ourself, the compiler cannot do this for us.

0.17 Lecture 17: Architecture and Optimisation

Week 9

In this lecture we explore how code can be optimised based on the hardware on which it is run. These techniques are introduced based on the following benchmark code. It is composed of a `struct vec`, which represents a vector, `int get_vec_element()` which returns a certain element from the vector and the heavy-work method `combine()` which does multiply of add all elements of a given vector.

- Data types: different declarations for `data_t`:
  - `int`
  - `long`
  - `float`
  - `double`



- Data types (`data_t`):
  - `int`
  - `long`
  - `float`
  - `double`
- Operations: definitions of OP and IDENT
  - `+` / `0`
  - `*` / `1`

```
/* data structure for vectors */
struct vec {
    size_t len;
    data_t *data;
};
```

```
/* retrieve vector element
   and store at val */
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;
    }
}
```

Compute sum or product of vector elements

In order to compare different optimisations, we we need a comparable unit. The most obvious one is execution time, which is calculated as $= \text{CPE} \cdot n + \text{overhead}$. Where *Cycles per Element (CPE)* is a way to express performance of a program which operates on vectors or lists.

Example: Base Performance We measure the following when running our program directly and with the `-O1` optimizer:

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14

All in all, we can observe that the optimizer shortened the execution time by a factor of two.

Some obvious optimisations we can to to our benchmark loop is:

- Do `vec_length` check outside of the loop.
- Access vector directly instead of using wrapper function `get_vec_element`.
- Accumulate results and write to memory after the loop.

```
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;
}
```

This already gives great improvements:

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

A bit about modern processor design

Sequential Processor Stages The fetch, decode, execute, memory, write back and pc stages are run in sequential. Since signals must propagate through instruction memory, register file, ALU and data memory in one cycle, the clock is very slow. Further hardware units are only active for a fraction of a cycle.

Pipelined Hardware The fetch, decode, execute and write-back stages are pipelined. While this allows for better utilisation of the hardware, it yields numerous problems. Most notably are data and control hazards.

Performance The program execution time is =)$\text{IC} \cdot \text{CPI} \cdot CCT$

IC: Instruction count

- Increased by optimising program

CPI: Cycles per instruction (= 1/IPC)

CCT: Clock cycle time (= 1/Frequency)

- Increased by increasing pipeline depth

While increasing the pipeline depth leads to higher CCT, the limits are:

- Delay of pipeline registers
- Inequalities in work per stage since we cannot break work into stages at arbitrary points
- Clock skew

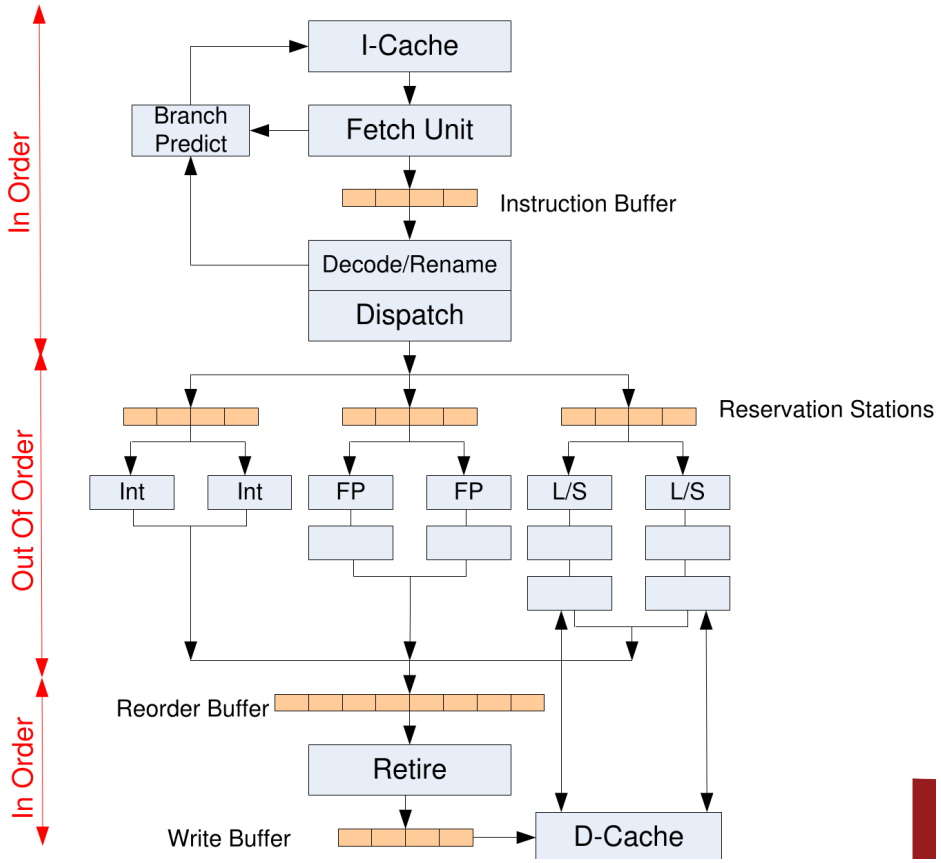
The Cycles per instruction (CPI) is calculated as = $\text{CPI}_{\text{base}} + \text{CPI}_{\text{stalls}}$, where the stalls CPI is determined by the stalls due to data and control hazards, as well as due to memory latency of larger memories.

Superscalar Processor A superscalar processor can issue and execute multiple instructions in one cycle. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

So a clear benefit is that is happens without programming effort. The processors can take advantage of the instruction level parallelism that most programs have.

In todays CPUs this is something very common.

The following graphics depicts the key design points



Superscala Processor Performance

We analyze the performance of a super scalar processor an hand of the Intel Haswell CPU

- 8 functional units
- Multiple instruction can execute in parallel
  - 2 load + address computation
  - 1 store + address computation
  - 4 integer
  - 2 FP multiply
  - 1 FP add
  - 1 FP divide

The following table depicts the latency (number of cycles required) and cycles/issue (how many cycles we have to wait before issueing the next instruction in this functional unit)

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

Latency vs Throughput For example FP multiplication has a latency of 5 and cycles/issue of 1. The execution of 5 independent multiplications takes thus 7 cycles to complete. In comparison, the execution of 5 dependant instructions takes 25 cyces to complete.

Data hazards There are three types of data hazards

Read after Write (RAW): Only true dependency, we actually need to wait

Write afte rWrite (WAW): Can be fixed with register renaming

Write after Read (WAR): Can be fixed with register renaming

In register renaming we map architectural registers to a larger pool of physical registers.

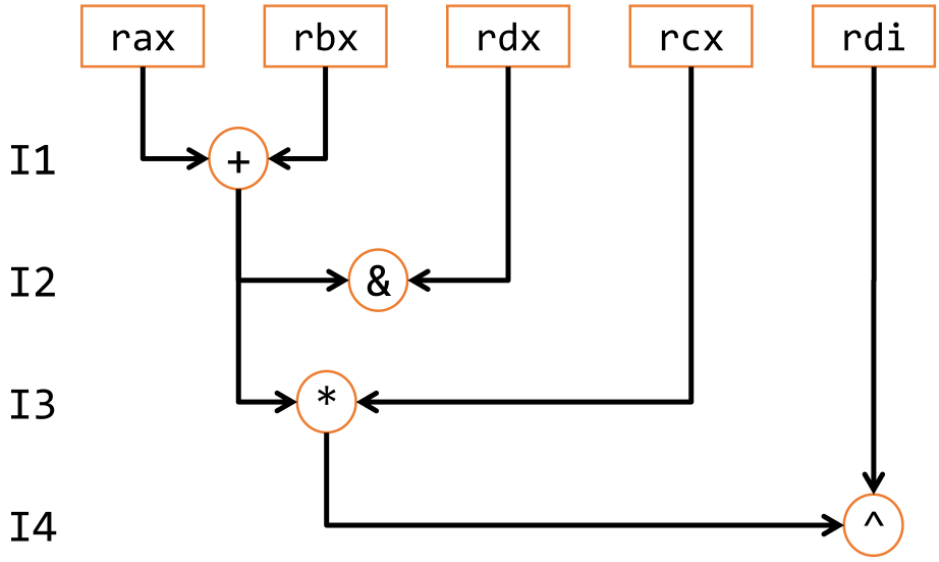
View of Instruction Execution

Imperative View Imperatively, registers are viewed as fixed storage locations which are read and written to by individual instructions. Instructions are executed in a specific order to guarantee proper behaviour.

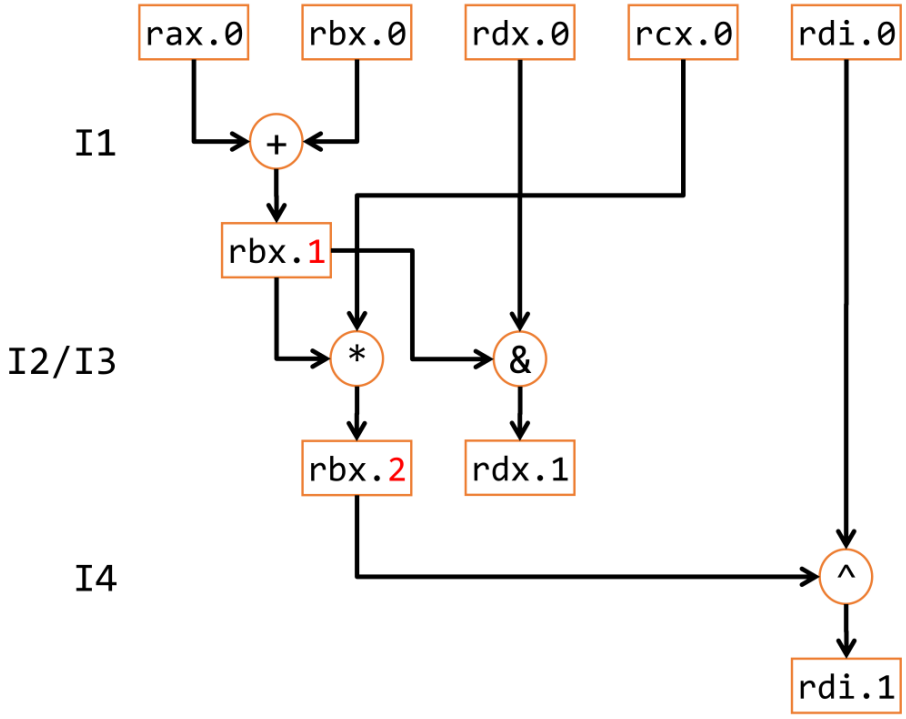
Given the following code:

```
addq %rax, %rbx  # I1
andq %rbx, %rdx  # I2
mulq %rcx, %rbx  # I3
xorq %rbx, %rdi  # I4
```

The imperative view looks als follows:



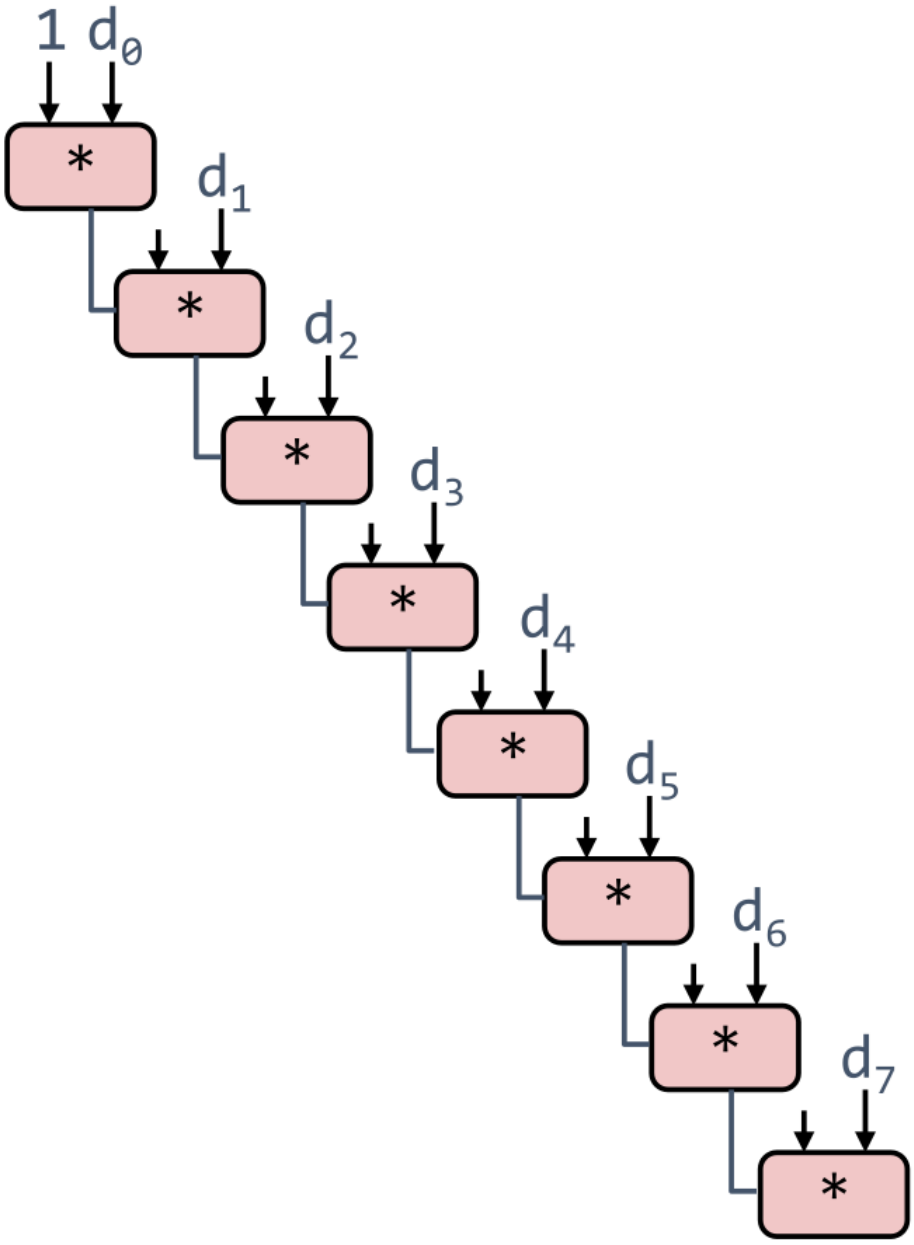
Functional View From a functional perspective we can view the execution of instruction as a dependency graph. Each write as creating new insurances of a value. So, operations can perform as soon as all operands are available. The sequence of execution may differ from the original instruction sequence.



Performance Bounds There are two types of bounds.

Latency Bound The latency bound limits the performance, when operations must be executed sequentially.

The performance is determined by the latency of the operation. So, the latency bound of our benchmark problem is simply the latency of the designated operation. For a vector of length 8 and multiplication operation, we get following *dependency graph*:



For the Intel Haswell example, we get

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Combine4	1.27	3.01	3.01	5.01
Latency bound	**1.00**	**3.00**	**3.00**	**5.00**

Loop Unrolling ($2x1$) Loop unrolling in the action of repeat a loop fewer times, but do more computation in one execution. For example a $2x1$ unrolling does $\frac{1}{2}$ as many iteration but $2x$ as much work per iteration.

Our benchmark code

```
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;
}
```

We get a performance of

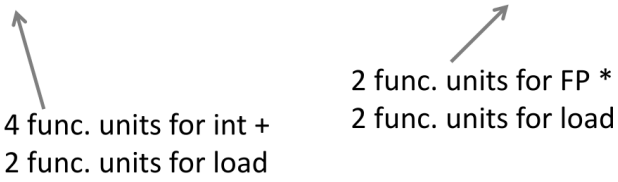
Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	**1.00**	**3.00**	**3.00**	**5.00**

We can see that only the performance of the integer addition has improved. All other operation stay the same, because they are still sequentially dependant.

Throughput Bound The throughput bound is the bound of the performance when instructions can execute in parallel.

For our benchmark we get

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
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**

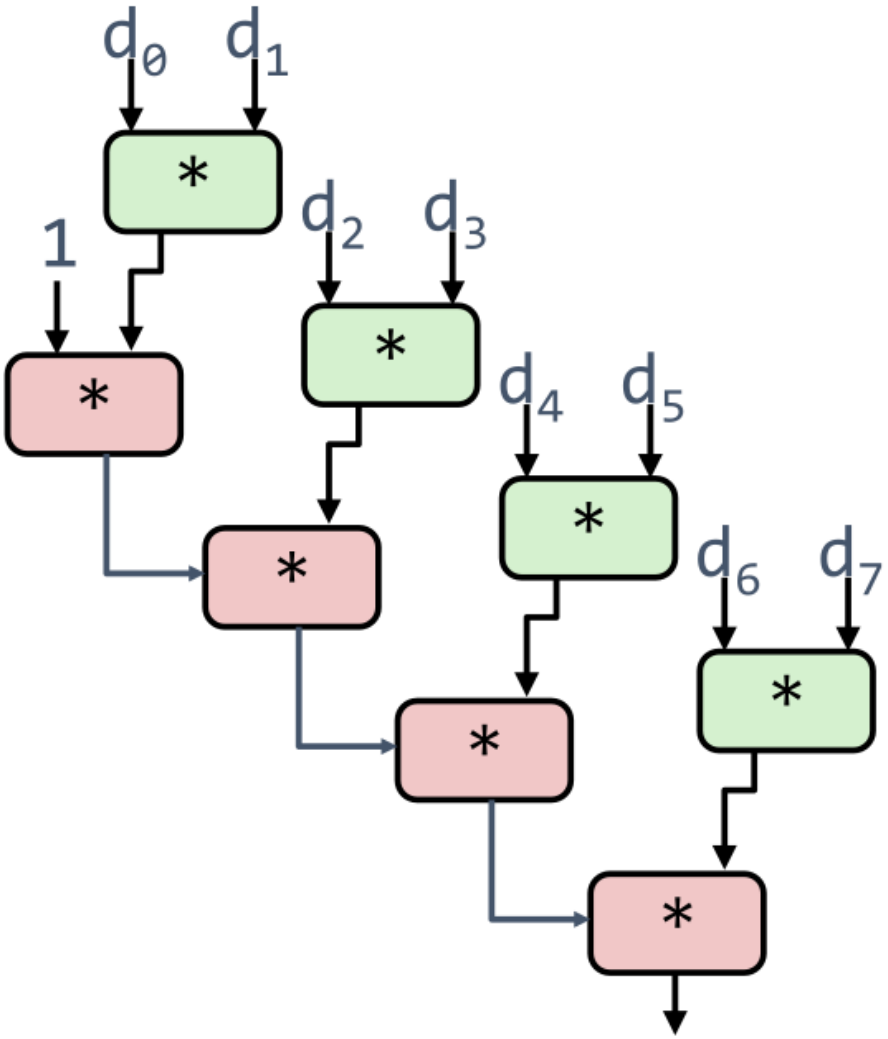


Reassociation

The idea of reassociation is to change the order of the brackets in order to create an independant operation. So it can be scheduled in parallel. In the previous loop unrolling the changed to computation to $x = (x \text{ OP } d[i]) \text{ OP } d[i+1]$. Now the change it to $x = x \text{ OP } (d[i] \text{ OP } d[i+1]$.

```
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;
}
```

The graph for the multiplication of 8 elements looks as



But one has to notice, that for FP this may change the results, since FP operations are not associative.

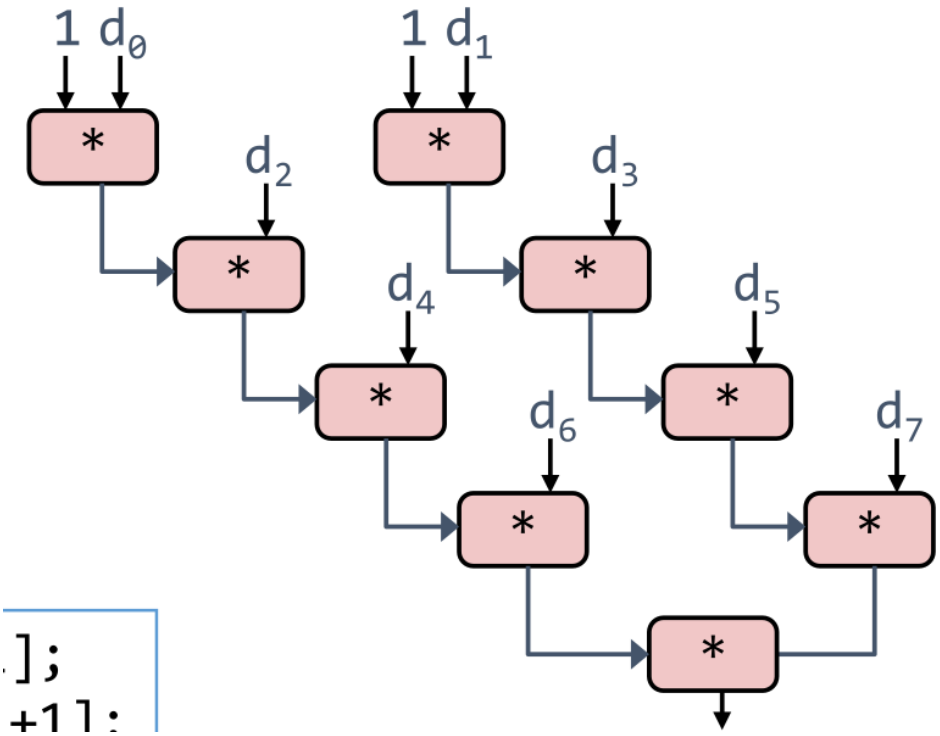
For our benchmark we get a nearly $2x$ speedup for FP addition and FP and integer multiplication. This is due to the sequential dependency.

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
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**

Ĺoop Unrolling with Separate Accumulators ($2x2$) Another idea is to increase the number of accumulators. This yields a different for of reassociation.

```
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;
}
```

This way we get two independant *streams* of operations



Concerning our benchmark performance, integer addition can make use of two load units and for

integer and FP multiplication and FP addition we get a $2x$ speedup compared to the $2x1$ unrolling

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
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**

Combining Multiple Accumulators and Unrolling

The idea is to unroll the loop to a degree of L and accumulate K results in parallel (L must be a multiple of K). The limitations for this are diminishing returns since we cannot go beyond throughput limitations, and we also get large overheads for short lengths.

The best values can only be evaluated through tests. Compiler cannot do these kind of improvements.

Double Multiplication For double multiplication on Intel Haswell we get

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
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

Accumulators

Integer Addition For integer addition on Intel Haswell we get

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
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

Accumulators

Summary Using the described methods the best performance we could achieve are

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
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**

It is limited only be the throughput of the functional units.

SIMD The only way to do even better is using SIMD instructions. Using the AVX2 SIMD operations on 256-bit vectors yields

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
Latency Bound	**0.50**	**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**

0.18 Lecture 18: Cache Optimisation

Week 9

Knowing the cache hierarchy is extremely important to write optimized code.

Problem CPU performance increased much faster than memory. This has lead to a gap which got greater over time. The solution to this problem are caches.

General Concept When we need some data, e.g. block b in the CPU, it it first checked, if the block is in cache. If it is not, the block is fetched from memory and placed in cache according to some placement policies. The next time the CPU needs block b, it can get it directly from cache.

Performance Metrics The three fundamental metrics are:

Miss Rate: Fraction of misses and total memory reference: misses/accesses $= 1 -$ hit rate.

- It tells how often to be have to go down the hierarchy to memory.
- Typically, it is $3 - 10\%$ for L1 and $< 1\%$ L2.

Hit Time: The time it takes to deliver a hit from the cache to the CPU, including the time required to determine of the data is in the cache.

- It is the time he have to stall in ordre to wait for data.
- Typically, it is $1 - 2$ clock cycles for L1 and $5 - 20$ clock cycles for L2.

Miss Penality: Additional time required on a miss to get the data from memory.

- Typically, $50 - 200$ clock cycles.

The more we increase clock speed, the higher the time penalties get in cycles. This is the general trend.

Often, we use the miss rate when talking about cache access instead of hit rate. This is because the hit rate is sometimes a bit misleading. For a cache with hit time of 1 cycle and miss penalty of 100 cycles, we get an average hit time when assuming hit rates of 99% and 97% off:

- 99% hits: 1 cycle $+ 0.01 \cdot 100$ cycles $= 2$ cycles
- 97% hits: 1 cycle $+ 0.03 \cdot 100$ cycles $= 4$ cycles

As we can see, a hit rate of 99% is twice as good as a hit rate of 97%.

Calculate Average Memory Access Time The following principle can be extended to n level of caches:

- **Average Memory Access Time** = HitTimeL1 + MissRateL1 * MissPenaltyL1
- MissPenaltyL1 = HitTimeL2 + MissRateL2 * MissPenaltyL2
- MissPenaltyL2 = DRAMaccessTime + (BlockSize/Bandwidth)

Types of Cache Miss There are three main types of cache misses:

Cold: Occurs when the cache is empty and we access the first block

Conflict: When at the designated location in the cache is a block, but not the one we are looking for

Capacity: When the set of active cache blocks is larger than than the cache

Additionally, for muliprocessor systems, there is a fourth type, the **Coherency**. We will learn more about that later.

Cache Organisation

There are three parameters to describe the organisation of the cache.

S $= 2^s$ sets

E $= 2^e$ lines per set

B $= 2^b$ bytes per cache block (for data)

Set: Different part of the memory address map to different parts of the cache. A Set is one such part of the memory.

Entry/Way/Line: Are the block in a set. When a cache is n-way, then n different blocks can be held in one set.

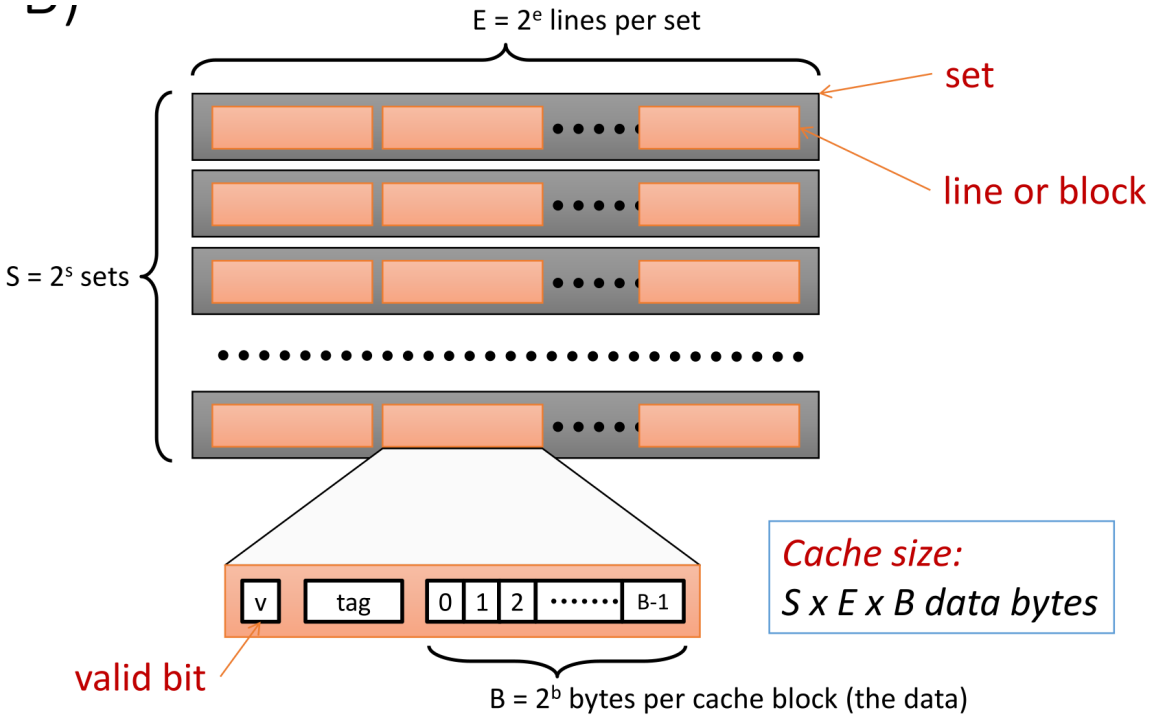
Each block has:

Valid Bit: If set, the entry is actual a valid block.

Tag: Helps to match the entry and determine of we have a hit.

Data: The data of the block.

The cache has size $S \cdot E \cdot B$ bytes.



Cache Reads

Given a memory address: In order to check if the desired block is already in memory, the address is split into three parts

Block Offset: Last b bits.

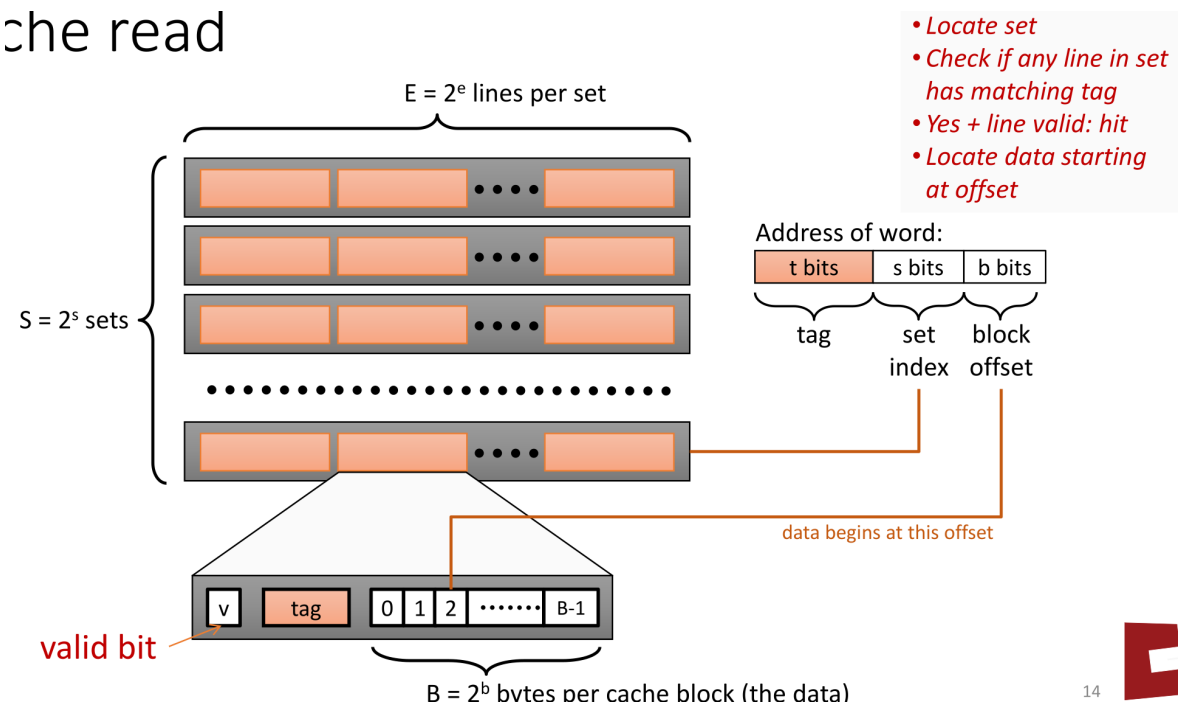
Set Index: The middle ss bits.

Tag: The most important t bits.

In order to check if the block is in cache, we proceed as follows:

1. Locate the set using the set idex
2. Check if any line/entry has a matching tag
3. If yes and entry valid, we have a hit and the data starts at the offset

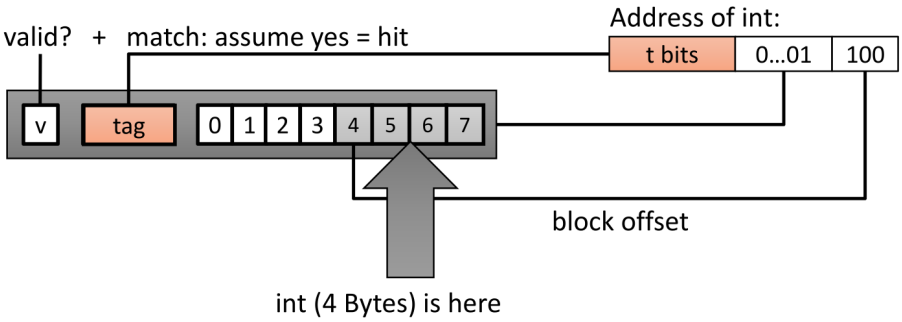
Unlike memory, a cache block is not byte accessible, therefore we use a block offset in order to access the desired data.



Types of Cache Caches different in their way. The simplest one is the 1-way cache, called direct mapped cache.

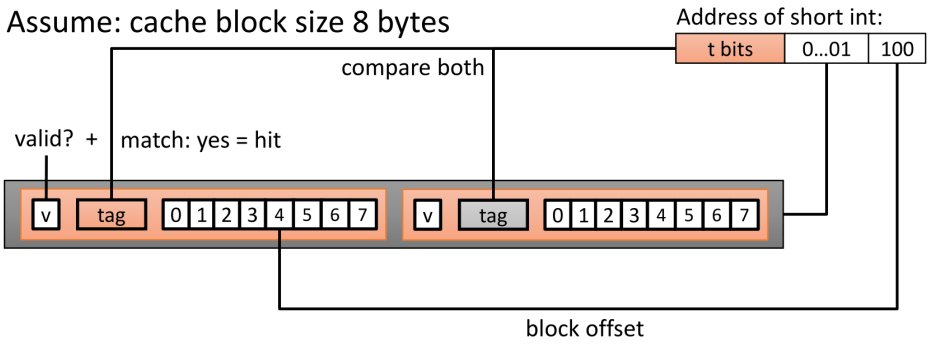
Direct Mapped Cache ($E = 1$) This is the simplest type and each set contains only one entry.

The following image shows an example with a cache block of size 8.



If there is no match, the block is replaced by the newly fetched block.

2-way Set-Associative Cache ($E = 2$) Each set contains two lines. Again, we assume a cache block size of 8 in the following example



Since we know that this is the address of a short int, we would access the bytes 4 and 5.

If there is no match, one line in the set gets replaced according to some replacement policies.

While a higher associative cache has , it comes with advantage of being more flexible in what to store, it comes with the cost of more tag comparisons.

The Memory Hierarchy

As the following table shows, there are many different caches responsible for different task

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

Different architectures have different caches. Most CPUs have 3 different caches and the L1 cache is often split into a instruction and a data cache. This is due to the locality of instruction (if no branch, we access the next instruction) and data (we often need the next data block).

Cache Writes

Write Hit There are policies on what to do if we write to a block which is also in cache:

Write-Through: Write changes down the hierarchy to memory

- Advantage: Memory is always consistent
- Disadvantage: Very slow then we write the same location multiple times

Write-Back: Change only value in cache and defer write to memory

- We need a *dirty* bit which indicates that an entry is different from memory.
- Advantage: Higher Performance
- Disadvantage: More complex hardware

Write Miss These policies describe what do to when we write to a location which is not in memory

Write-allocate: Load block into cache and update the entry only in the cache

- Advantage: Good if there are more writes to this location
- Disadvantage: More complex hardware and this may evict existing blocks from cache
- Common with write-back caches

No-Write-Allocate: Write change to memory

- Advantage: Simpler to implement
- Disadvantage: Slower if there are multiple writes to the same location
- Common with write-through caches

Other Features There are other demission to be made when designing a cache:

Unified: Instructions and data is in the same cache

- Often, L1 is not unified and L2 and L3 caches are unified

Private vs Shared: Is the cache exposed to other cores or only a single one

- Often, L1 and L2 are private while L3 is shared

Inclusive vs Exclusive:

- If everything which is is this cache is **also in all** lower-level caches, it is incisive.
  - Is easier for eviction policy
- If anything which is in this cache is **not in any** lower-level cache, it is exclusive.
- Often, only L3 is inclusive and L1 and L2 are exclusive

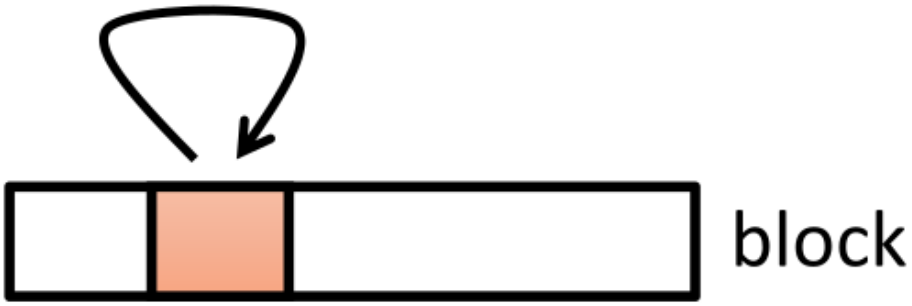
Software Caches Software caches allow for more flexible and complex policies.

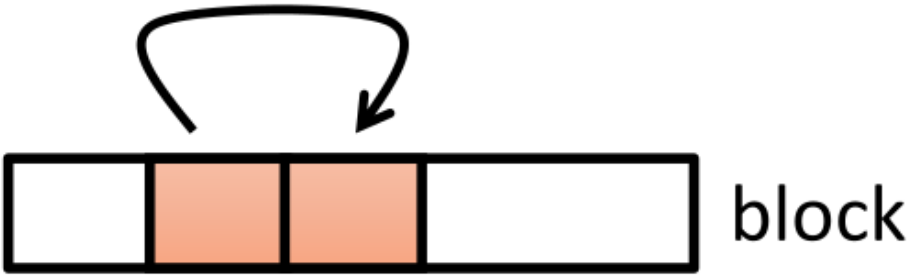
Cache Optimisations

Principle of Locality The reason why caches work, is due to locality. Programms tend to use data and instruction with addresses near or equal to recently used ones.

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.





Optimisation for the Memory Hierarchy As a programmer, it is important to consider the locality of code. E.g. a linked list is not contiguously in memory. Furthermore, once data is fetched, we should do as much work with if, was we can.

- Write code that has *locality*
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time
- How to achieve this?
  - Proper choice of algorithm
  - Loop transformations

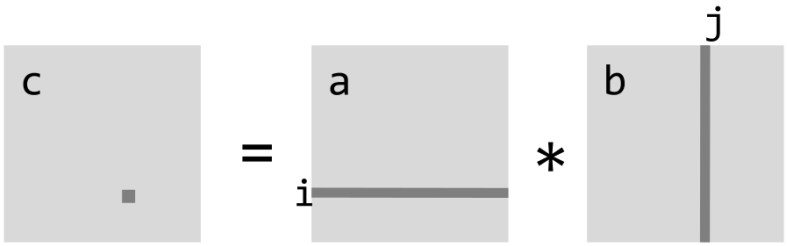
- Cache- versus register-level optimization:
  - In both cases locality desirable
  - Register space much smaller $\Rightarrow$ requires scalar replacement to exploit temporal locality
  - Register level optimizations include exhibiting instruction level parallelism (conflicts with locality)

Example: Matrix Multiplication Matrix multiplication is a great algorithm do demonstrate usage of locality. And how the performance can be drastically improved when considering locality.

The most trivial implementation is probably:

```
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];
}
```



In order to analyse the cache miss, we assume:

- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size $C << n$ (I.e. we cannot load the whole matrix into cache)

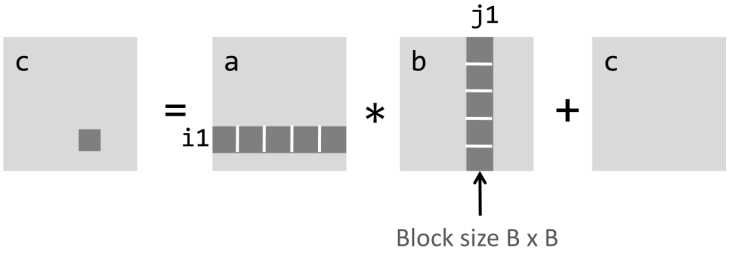
For the first iteration of the inner loop we have $\frac{n}{8} + n = \frac{9n}{8}$ misses. The first operand are the misses for matrix A, the n misses for operand B.

For the following n^2 repetitions we get the same number of misses, thus the total number of misses is $\frac{9n}{8} \cdot n^2 = \frac{9}{8} \cdot n^3$.

When doing block matrix multiplication, we try to make better use of locality by changing the order in which we access elements.

```
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+=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];
}
```



Additionally, to the previous assumptions, we assume that three blocks fit into cache ($3B^2 < C$). So B is a free parameter, but restricted by this assumption.

For one blocks, we have a total of $\frac{B^2}{8}$ misses. The first block iteration we have $\frac{2n}{B} \cdot \frac{b^2}{8} = \frac{nB}{4}$ many misses.

The number of misses for each iteration is equal. Thus, we get a total number of misses of $\frac{nB}{4} \cdot (\frac{n}{B})^2 = \frac{n^3}{4B}$.

The reason of the drastic difference between non-blocking ($\frac{9}{8} \cdot n^3$ and blocking ($\frac{1}{4B} \cdot n^3$ is because matrix multiplication has inherent temporal locality.

0.19 Lecture 19: Exception Handling in Processors

Week 10

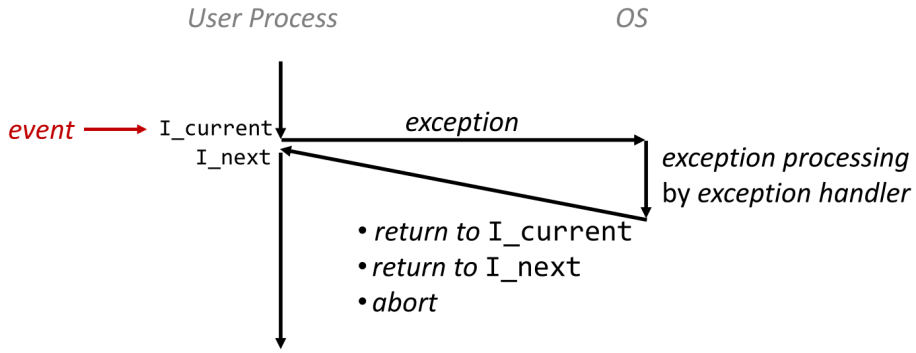
(Exceptional) Control Flow While processors do in essence nothing other than processing a stream of instruction, they also have to react to special events like listen for keyboard inputs.

Jumps and branches, as well as calls and returns are two mechanism which change the control flow based on the program state. But there are also changes to the system state, on which the processor must react. For example data arrival from disk or network, division by zero, user terminates a program or system timers responsible for scheduling. So the system needs to handle *exceptional control flow*.

Exceptional control flow exists on all levels of a computer system:

- Low Level mechanisms
  - Hardware exception
  - Combination of HW and OS SW, i.e. malloc
- High Level Mechanisms
  - Process content switch
  - Signals, e.g. terminate a program, segmentation fault...
  - Nonlocal Jumps used for coroutines
  - Programming language exceptions

Exception An exception is a transfer of control to the OS in response to some event. It is in essence a context switch. After the OS has handled the exception, the processor switches back to the user process.



Types of Exceptions There are two main types of exceptions:

Synchronous Exception: Result of executing an instruction.

Trap: Internal exception

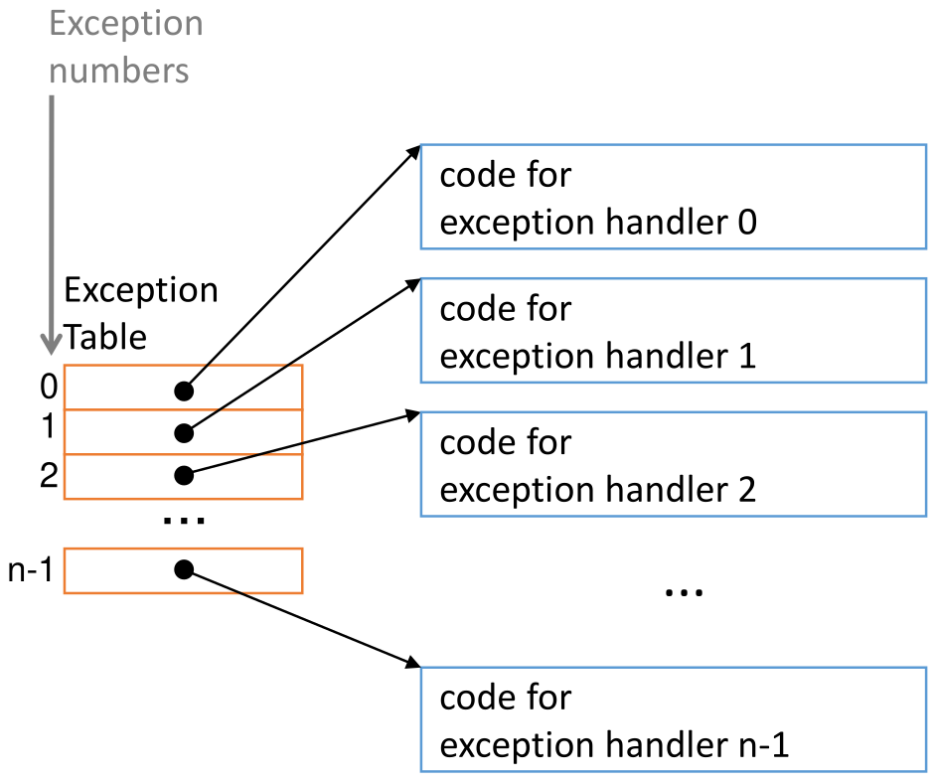
Fault: Potentially recoverable error

Abort: Nonrecoverable error

Asynchronous Execption: Result of an event that are external to the processor.

Interrupt: Signal from i/O device

Exception Vector At boot time, the OS allocated and initializes the *exception table*. Its base address is stored in the *Exception Table Base Register*. Each type of exception has a unique number k, called interrupt vector, which is used to index into the exception table. When exception k occurs, handler k is called.



The exception vectors for x86 are the following:

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

Kernel Most OS container a kerne. The kernel is the part of the OS which runs in kernel mode. This means it has some privileges compared the user mode, where programs are run. But on a exception, to content is switched to kernel mode (aka supervisor mode, privileged mode, ring 0 etc.). In kernel mode a program has access to system states (virtual memory etc.), there are some new instructions and registers available, some exceptions are disabled etc. I.e. the kernel is always:

- a set of trap handling functions
- code to create the illusion of user-space processes

and sometimes:

- a set of threads in a special address space

Synchronous Exceptions

They are caused by executing instructions.

Traps:
- Intentional
- Returns control to the "next" instruction
- Example: system calls, breakpoint traps (while debugging), special instructions

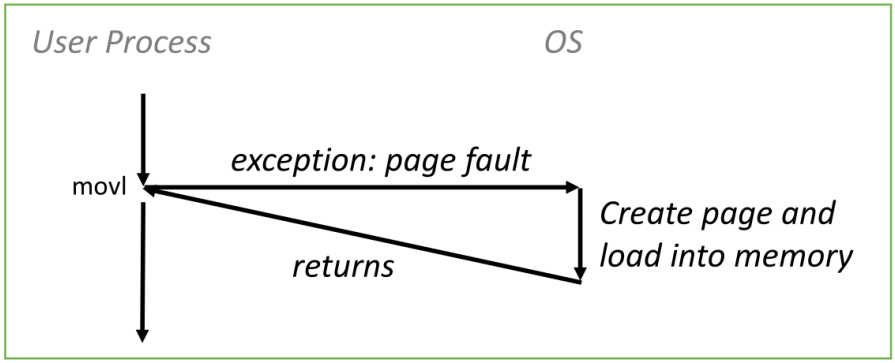
Faults:
- Unintentional
- Sometimes recoverable

- Re-execute faulty ("current") instruction or abort
- Example: page fault (recoverable), protection fault (unrecoverable), floating point exception

Aborts:
- Unintentional
- Unrecoverable
- Aborts current program
- Example: parity error, machine check

Fault Example: Page Fault Code tries to access a location of its address space which is currently not in main memory but on disk. So the page handler must load that page into physical memory. After doing this, it returns to faulting instruction and successfully execute it on the second try.

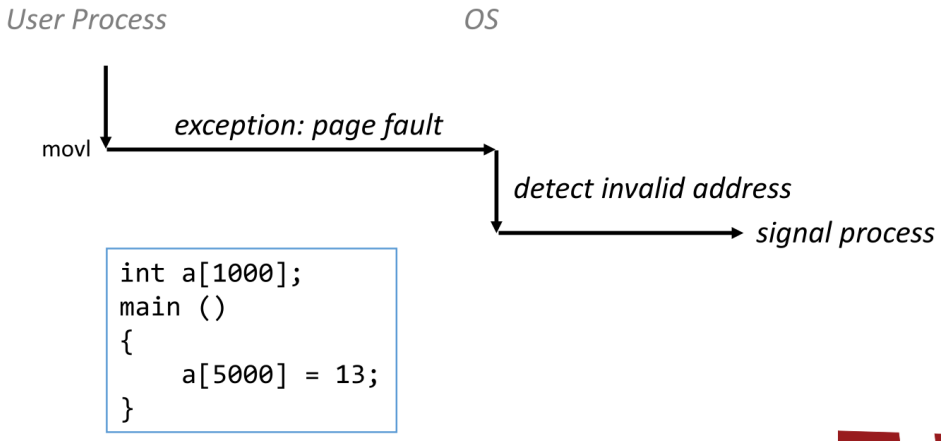
```
80483b7: c7 05 10 9d 04 08 0d        movl   $0xd,0x8049d10
```




```
int a[1000];
main ()
{
    a[500] = 13;
}
```

Fault Example: Invalid Memory Reference Code tries to access a location outside of its address space (i.e. an page which we do not have permissions for). The page handler detects the invalid address and sends a **SIGSEGB** signal to the user process. The user process exits with *segmentation fault*.

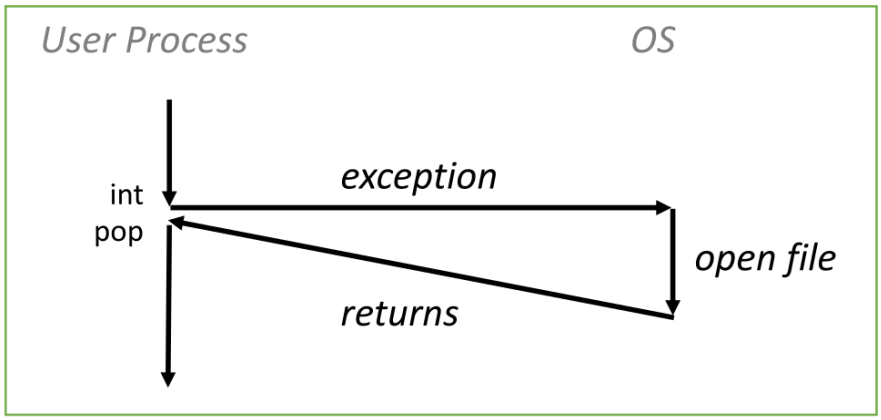
```
80483b7: c7 05 60 e3 04 08 0d        movl   $0xd,0x804e360
```




```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

Trap Example: Opening a File User opens a file by calling `open(filename, options)` which executes a systems call (e.g. *int* instruction). The OS must find or create the file and get it ready for reading or writing. Then it returns an integer integer which uniquely defines this file (required for reading the next time the same file).

```
0804d070 <__libc_open>:
 . . .
 804d082: cd 80                      int    $0x80
 804d084: 5b                         pop    %ebx
 . . .
```



Asynchronous Exceptions

They are caused by events external to the processor and indicated to the processor by setting the processor's interrupt pin. The processor finished the current instruction and saves the state before switching to the interrupt.

Examples:

- I/O interrupts (keyboard, network packages etc.)

- Hard reset interrupts (click power button)
- Soft reset interrupts

These interrupts pins of the processor are connected to interrupt-request lines. They are either edge- or level-triggered. The **exception handler** is the code, which we get by indexing into the interrupt exception table using the interrupt table.

Interrupt are either **maskable** or **non-maskable**. It they are, then the interrupt can be ignored or delayed. This allows to prioritize certain interrupts over others.

The mechanism for handling interrupts is equivalent to the one of exceptions (index into exception table using exception vector etc...).

x86 interrupts

These processors have to interrupt pins:

Non-Maskable Interrupt (NMI) Non-maskable means that these interrupts cannot be disabled. So they are used for interrupts of high importance like major hardware faults (e.g. memory parity error), watchdog timer etc.

When asserted, the processor completes the current instruction and then issues the exception #2 of the exception interrupt table (this is hardcoded).

Since multiple devices are connected to this pin, the exception handler has to figure out which device has cause the interrupt. The OS iterates all possible devices and checks which one has a state which indicates that the device has caused an interrupt. This check is called *pulling* and is something quite inefficient and low. However, since we are very likely to crash anyways, it is not something that happens very often.

NMI does not support nesting. This means that when processing a first interrupt the processor does not listen for any new interrupts which may occurs in the meantime.

Interrupt Request (INTR) These requests can be disabled by setting certain bit (flags). If asserted, the processor completes the current instruction and saved the state, then it acknowledges the receive by setting the INTA pin. The device detects that and in consequence sends the interrupt vector to the processor via the data bus. The processor issues the designated exception handler.

Interrupt Controllers

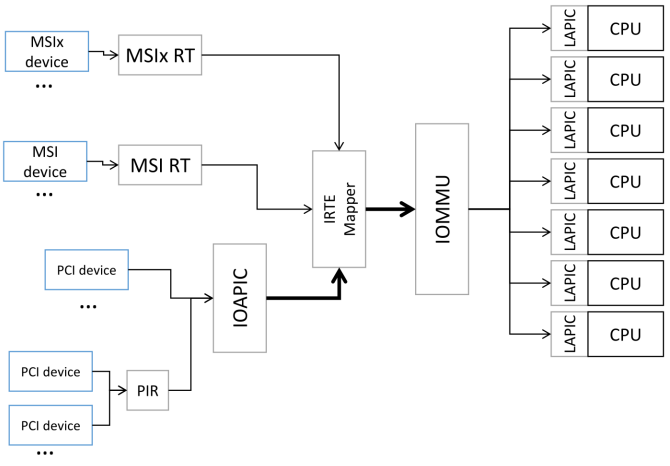
Using the introduces setup, we have to problem that a device does not know what vector to give. Also, it may cause troubles when multiple devices cause an interrupt at the same time.

Programmable Interrupt Controller (PIC) The PIC solves these problem. It is a device which is connected to the INTR pin and the data bus, and the INTA pin of the processor is connected to the PIC. Further, all devices are no longer connected to the processor directly, but to the PIC via IRQ-lines. This way, the OS can flexibly configure what vector to send on what interrupt of which device. I.e. the PIC maps interrupt pins from devices to interrupt vectors.

The PIC can also buffer simultaneous interrupts and deliver each interrupt vector separately. It also allows to prioritize devices over others as well as selectively mask any individual device's interrupts.

Over time the PIC was improved and integrated into the processor itself. Also, they support multiple processors systems as used in datacenters. They also have many more advanced features.

This gets more complex....



0.20 Lecture 20: Virtual Memory

Week: 11

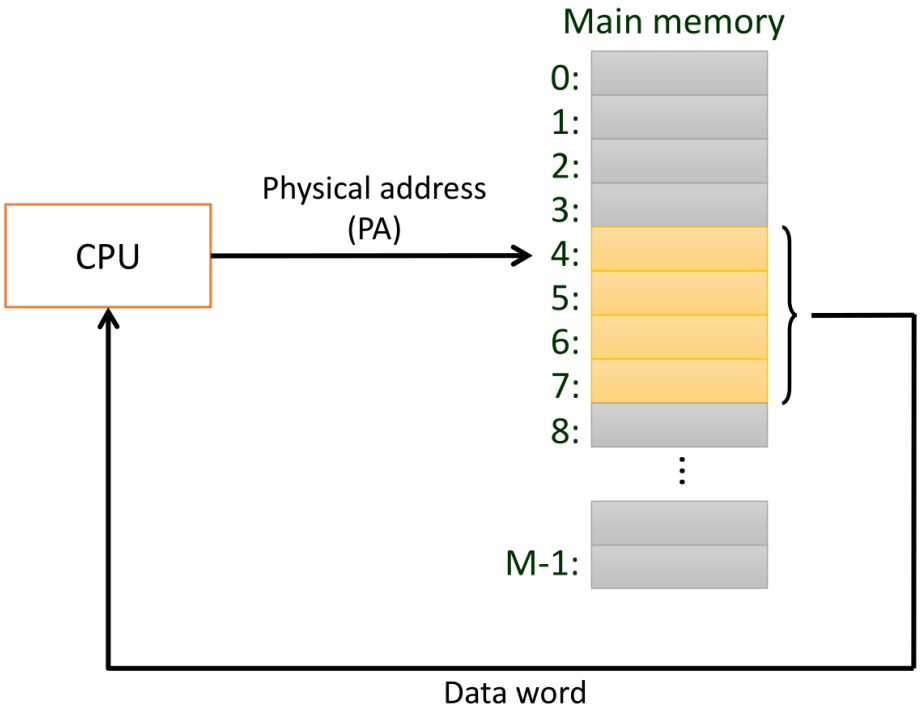
Recap: Address Translation

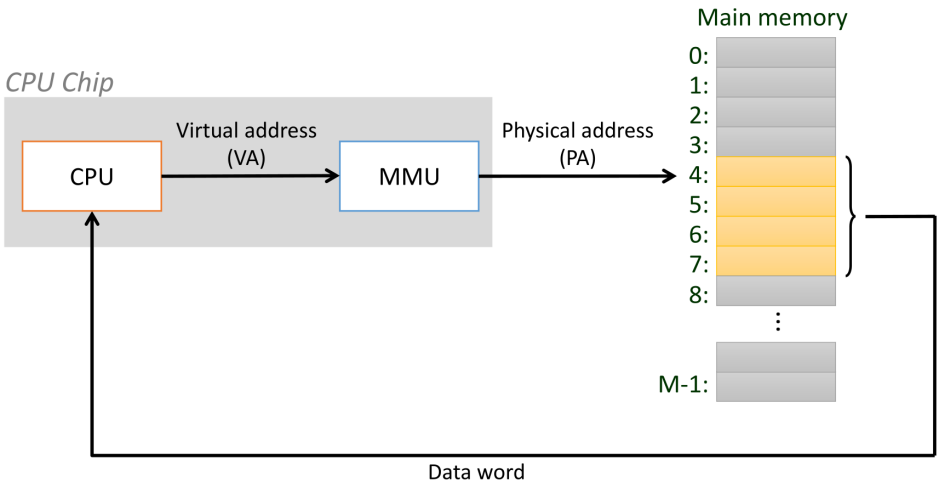
On most modern systems programs do not directly interact with the physical memory, but with its own virtual memory space. All these private virtual memory spaces are mapped on he physical memory.

Address Space The virtual/physical address space is a ordere set of contagious non-negative integer address from 0 up to some $N-1$. Where N is determined by the system (bits) for virtual memory, and for physical memory by the size of the available memory.

Each virtual address must map to a physical address. A single physical address may be mapped to by multiple virtual addresses. This is useful for e.g. shared libraries.

Virtual memory is used in all modern computers. Only in simple systems like embedded mikrocontrollers physical addressing is still used.

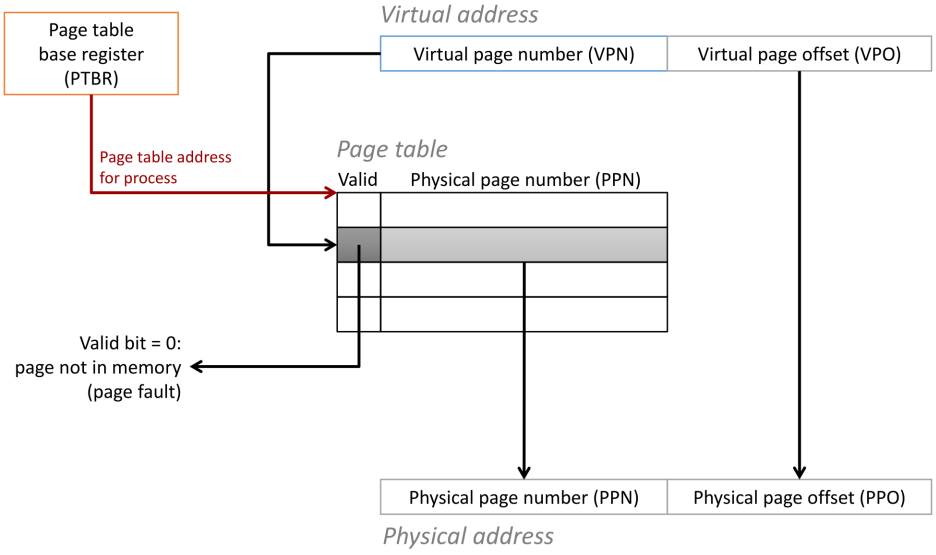




Virtual memory is handles by OS and hardware. We need hardware for a efficient translation. This is done by the memory management units (MMU). The translation table is managed by the OS.

Address Translation The OS operates on memory in blocks called pages. This is similar to caches and cache blocks. The default page size is 4KB. This size determines the granularity of the translation.

The translation is done using the page table. The table itself is stored in XXX and the start address is stored in the page table base register (PTBR). The start address is different for each process. The translation is done as as follows. The virtual address contains two parts, the *Virtual Page Numer (VPN)* and the *Virtual Page Offset (VPO)*. The VPN is taken and used to index into the page table. The entry gives uf the *Physical Page Number (PPN)*. The PPN is concatenated with the unchanged VPO to get the physical address. An additional valid bit tells if the given VPN is valid or if it is not in memory (page fault).



Uses of Virtual Memory

Why Virtual Memory The use of VM has three main advantages:

Efficient use of limited RAM Using 64 bit addressing, we can address 16 Exabytes of data. However, a commons system has only a few GB of RAM. So, VM provides a way to cache data by

only storing parts of the virtual address space in disk and only keep active arias of virtual address space on memory.

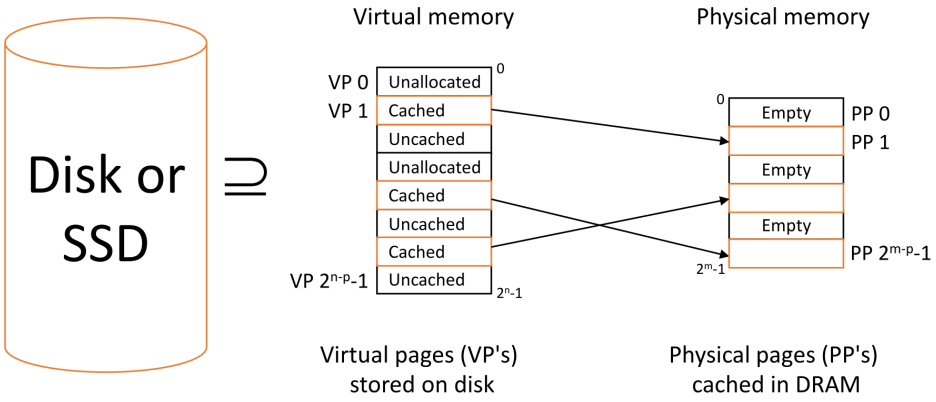
Memory pages can be in three different stages:

Unallocated: Page which was not yet used.

Cached: Pages found on disk and on main memory.

Uncached: Pages found on disk but not main memory.

Cached and uncached pages are found on disk. Additionally, cached pages are also found on physical memory.



Since disk access is incredibly expensive, pages are larger then cache blocks to prevent going to cache too often. The process of bringing data (pages) from disk into memory is called paging or swapping. Replacement algorithms are highly sophisticated and therefore implemented in software. To further reduce the number of required paging, write-back policy is used instead of write-through.

This works thanks to locality. Meaning, the programs tend to access a set of active virtual pages, called the *working set*. As long as the size of the working set is smaller than the main memory size, we can expect good performance. But as soon as it exceeds the memory size we get a performance meltdown, caused by copying and moving of pages.

Further, the smaller the page size, the more mappings we have. Since the *Translation Lookaside Buffer (TLB)* is very small, only few mappings can be cached there.

Simplified memory management for programmers Each process gets a virtual address space of equal layout (i.e. heap, stack etc are located at the same addresses). The virtual address space is kind of a abstraction of the full memory mode. The linear virtual memory as actually scattered across the physical memory. But it is all managed by the OS.

Protection and Sharing Virtual address spaces are isolated from each other different programs cannot interfere which each others memory. Besides that, PTE has some extra bits to tell if a physical memory location is readable, writable, executale or only accessible by superusers. The page fault hander checks these bits before remapping and causes a SIGSEGV exception in case they are violated.

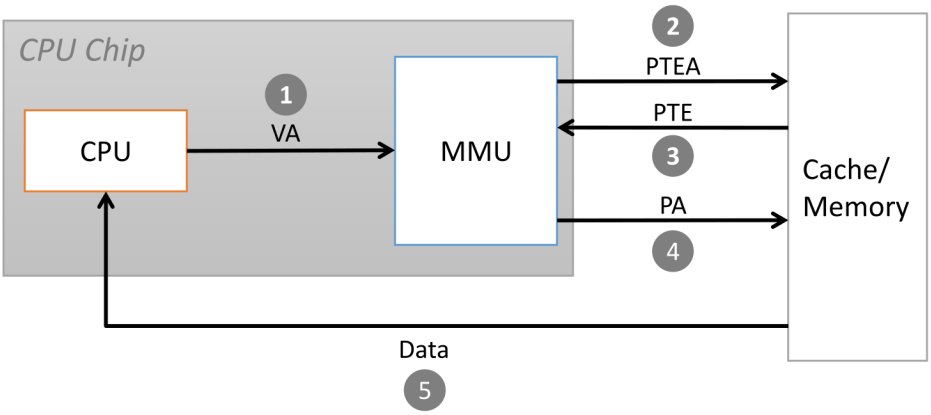
Simplifies linking and loading Since address spaces share the same layout. Linker precisely know at what address which section starts. This simplifies the implementation of linkers.

Pages also allow for on dempand loading, where .data and .text are copied on demand. This improved loading efficiently greatly.

Address Translation Process

Page Hit The steps in case of a page hit are the following:

1. The CPU sends the VM address to the MMU
2. The MMU does a PTE access in the lookup table in the main memory
3. The MMU receives the PTE from memory
4. The MMU accesses the physical memory using the translated address
5. The data is received and send to the CPU



These steps are all handled by HW.

Page Fault If the page is not cached, but uncached we have a miss (get a page fault) and have to fetch the data from disk.

1. The CPU sends the VM address to the MMU
2. The MMU does a PTE access in the lookup table in the main memory
3. The MMU receives the PTE from memory
4. The valid bit of the PTE is zero and so, the MMU trigger a page fault exception.
5. The exception handler gets active and decides on a page to evict (to make space for the new page). Before evicting the page, it must check the dirty bit and depending on if it is set, write the page first back to disk.
6. The handler pages (load page from memory) the new page and updates the PTE in memory.
7. The handler returns to the original process and restarts the faulty instruction.

The page fault hander is implemented in OS since it is more sophisticated.

Translation Lookaside Buffers (TLB)

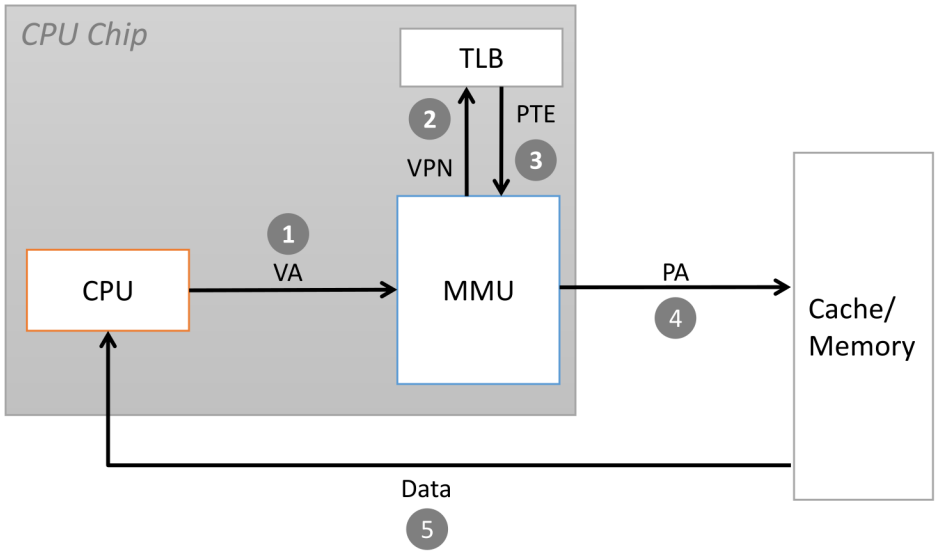
Since the page table is stored in memory as all other data, PTEs gets also cached in the cache hierarchy. However, PTEs may get evicted by other data reference. Also, while L1 cache is quite fast, it still takes a few cycles.

The *Translation Lookaside Buffer (TLB)* solves this problem. It is a small hardware cache in the MMU and caches mappings of virtual page number ot physical page numbers. For small page tables, it may even contain the complete table.

TLB are very common.

TLB Hit In case the translation mapping in in the TLB, the following steps are executed.

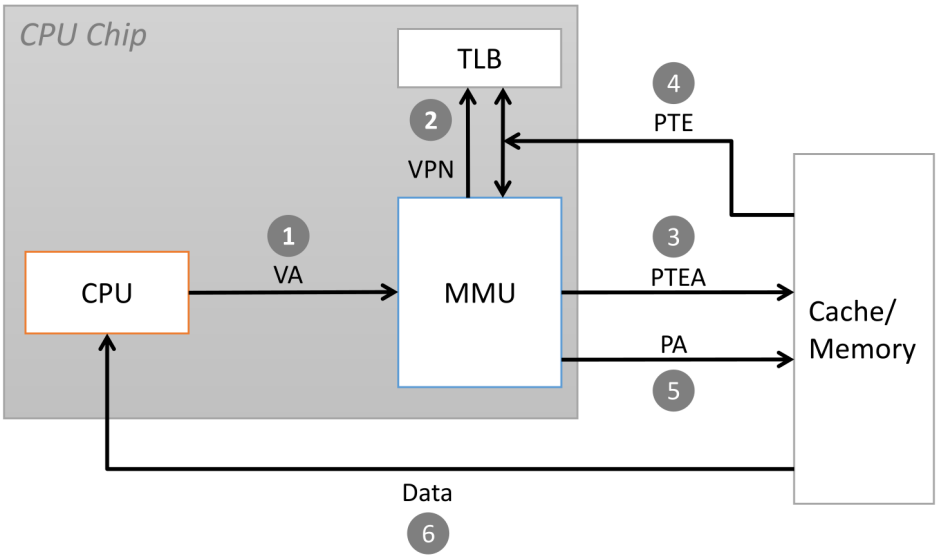
1. The CPU sends the VM address to the MMU
2. The MMU indexes the TLB using the VPN bits
3. From the TLV it receives the PTE bits
4. The MMU concatenates the PTE and address offset and accesses and accesses the memory.
5. The data is received and send to the CPU.



A hit eliminates an additional memory access to get the PTE.

TLB Miss If the VPN is not in TLB, me steps are:

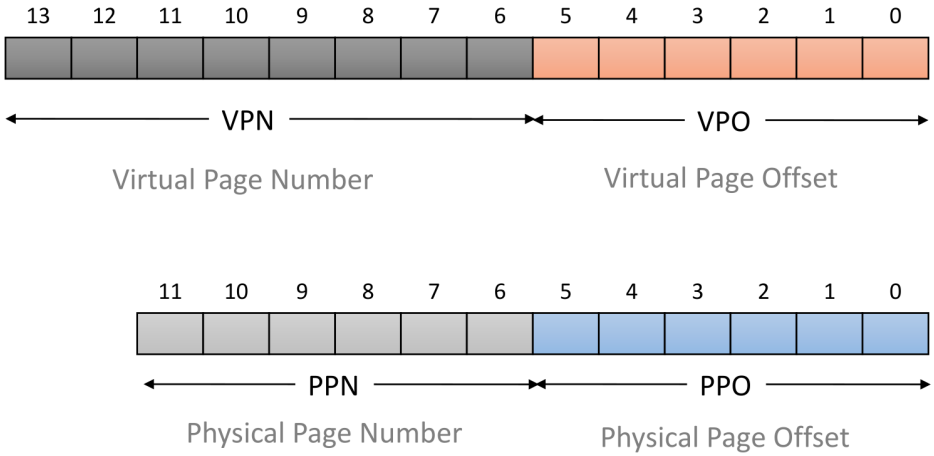
1. The CPU sends the VM address to the MMU
2. The MMU indexedthe LTB using the VPN bits. But VPN is not in the TLB and therefore we do not get anything back.
3. The MMU does a PTE access in the lookup table in the main memory
4. The MMU receives the PTE from memory and puts it into the TLB (for further accesses to this page). receives the PTE from memory and puts it into the TLB (for further accesses to this page).
5. The MMzu accesses the physical memory using the translated address
6. The data is receive and send to the CPU



TLB misses are actually rather rare.

Simple Memory System Example

Addressing Let's consider a memory system with 14-bit virtual addresses and 12-bit physical addresses and a page size of 64 bytes. The size of the VPN, VPO, PPN and PPO is visible in the following graphic:

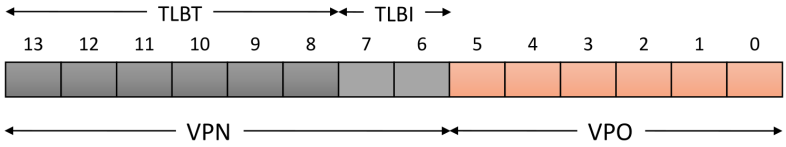


Page Table The first 16 of total 256 entries of the page table are:

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
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

TLB The TLB contains 16 entries and is 4-way associative. The VPN is used to index into the TLB. The 2 LSB of the VPN, the TLB index (TLBI), are used to determine the set of the TLB to use, and the remaining bits of the VPN, the TLB tyg (TLBT), are used to verify the tag of the TLB entry.



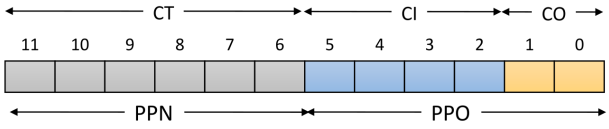
Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Cache The cache contains 16 lines of each 4-byte blocks size. The cache is addressed using physical addresses and it is directly mapped.

The physical address is split into three parts. The most significant 4 bits of the PPO are used for the cache index (CI) and the remaining two for the cache offset (CO). The CI tells which index of the cache we want, and the CO determines in wich blocks we are actually interested. The full PPN is used as the cache tag (CT) to determine if we have a match or not.

ally addressed

mapped

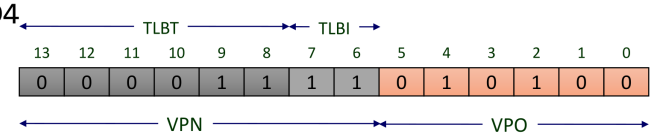


Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

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	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

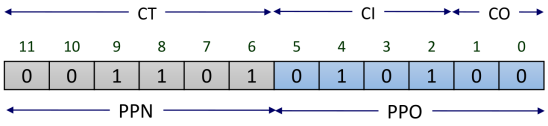
Example 1: For the virtual address $0x03D4$ we have:

Virtual Address: 0x03D4



VPN 0x0F TLBI 3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

Physical Address

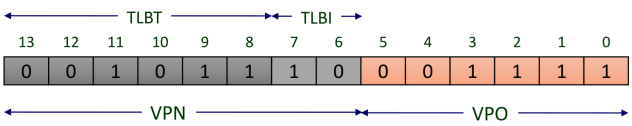


CO 0 CI 0x5 CT 0x0D Hit? Y Byte: 0x36

The PPN is in the TLB and can be retrieved from there. The physical address can be easily constructed and used to access the cache.

Example 2: For the virtual address $0x0B8F$ we have:

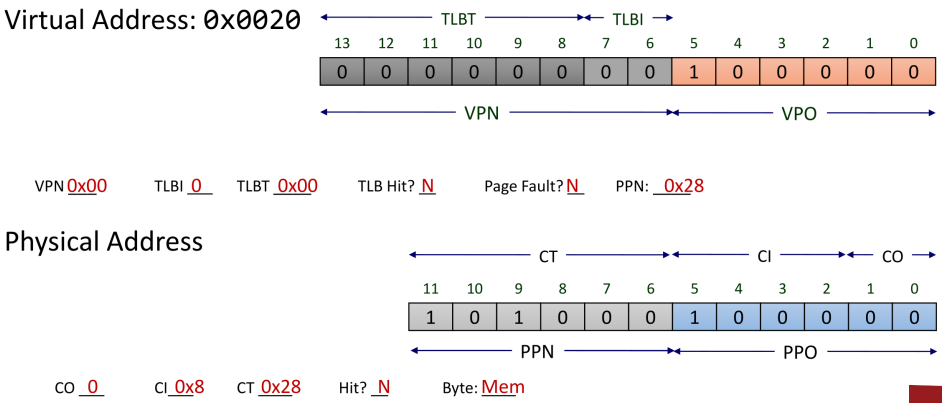
Virtual Address: 0x0B8F



VPN 0x2E TLBI 2 TLBT 0x0B TLB Hit? N Page Fault? Y PPN: TBD

The PPN is not in the TLB. In this example, the PTE is also not in memory and hence, we get a page fault and is has to be fetched from memory.

Example 3: For the virtual address $0x0020$ we have:



While the tag is actually in the TLB, it is invalid and hence we do not get a PPN. Therefore, we check in the page table in memory, which is the case for this example.

RECHECK: The images of the last two examples are probably wrong

0.21 Lecture 21: Continue: Virtual Memory

Week: 11

Terminology *Virtual Page* may refer to a page-aligned region of virtual memory space or to the content of this region. The *Physical Page* reverse to a page-aligned region in physical memory. The *Physical Frame* is the container of the page.

Multi-Level Page Tables

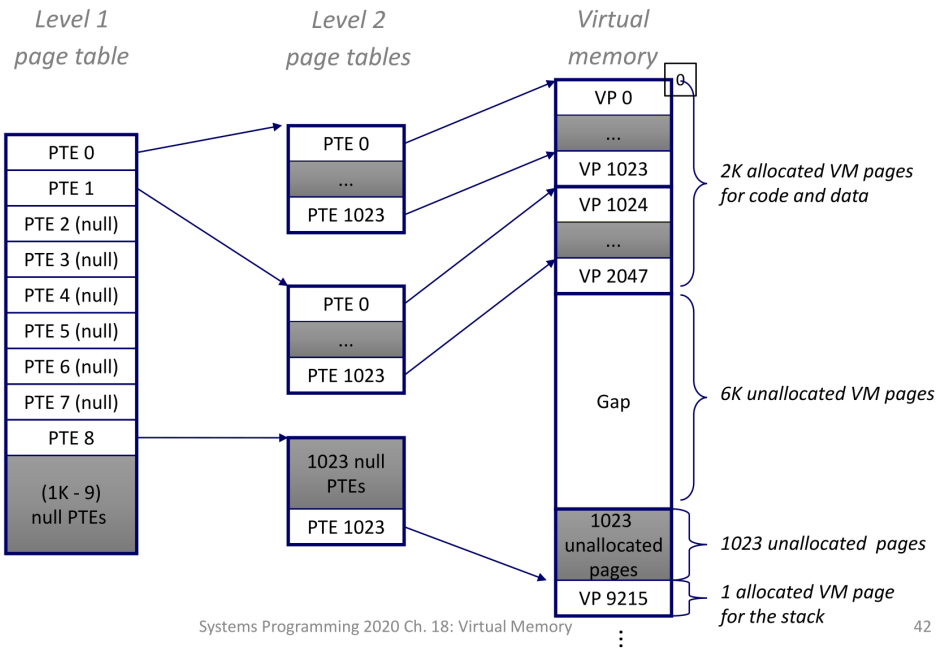
Motivation: Linear Page Table Size Given:

- 4 KB page size
- 48-bit address space
- 8-byte PTE

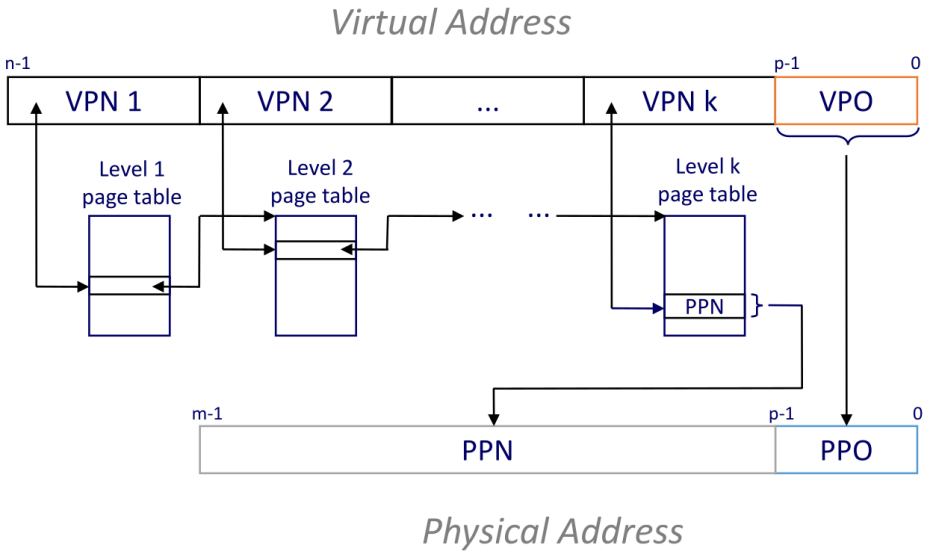
The resulting page table has size: #PTE·PTE size and the number of PTE is address space size/page size. This gives $2^{48}/2^{12} \cdot 2^3 = 2^{39}$bytes $= 512$GB.

This page table would never fit into memory. To reduce this size, we use additional levels of indirections.

Two Level Page Table Hierarchy In this setup, we have two page tables. Lable one contains the full range of virtual addresses. In contrast to that, the second level tables are only created when needed. When we access not previously accessed page table, an entry in the PT1 is created, and its entry points to the base address of the table of the second page table. The second page table entry does then point to the actual PTE from where we can get the PPN.



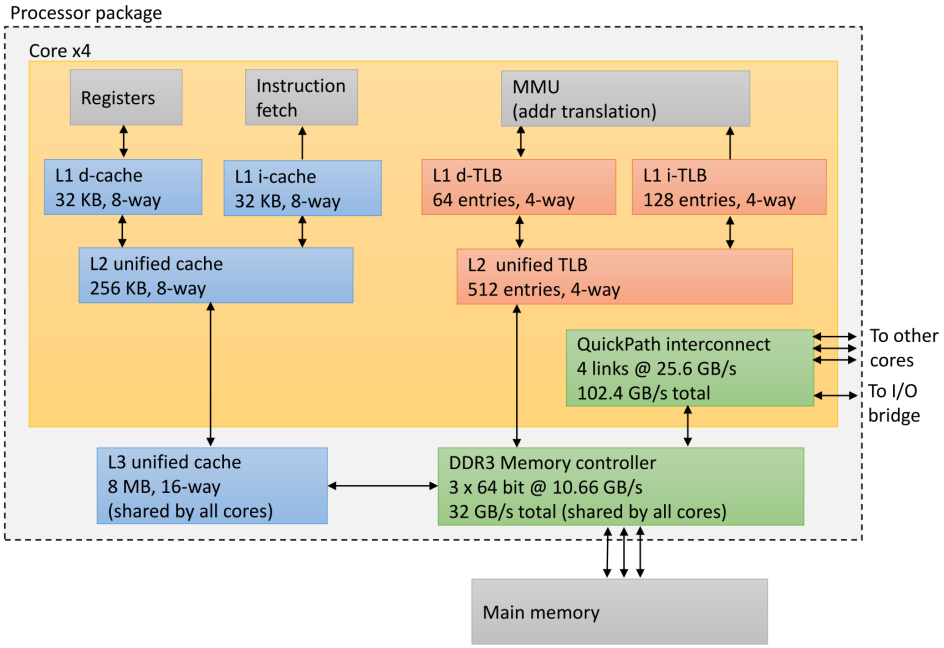
The page tables of the different levels are induced by different parts of the VPN. This concept can easily be extended to k levels. In x86-64 4 layers of indirection.



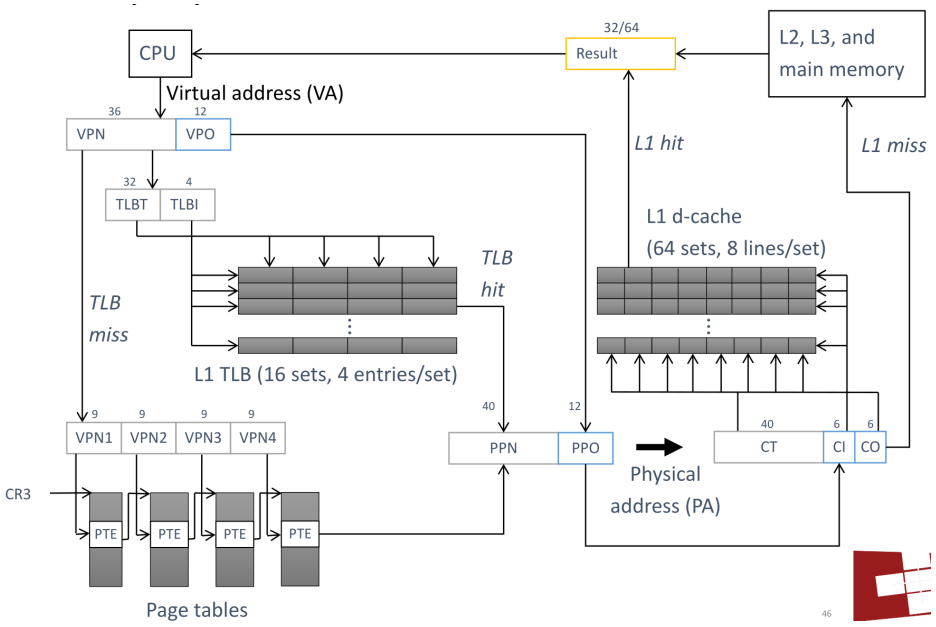
As long as we get a hit in page table $i-1$ we can index into page table i.

Case Study: i7

A typical i7 processor has the following CPU schema:



And the following memory system:



The `CR3` is the register which hold the process specific base address of the page table.

The VA has typically 48 bit size and the PA is 52 bits. The PPN has to be 40 bits long and hence, a table entry must be 64 bits.

TLB This part of the CPU is poorly documented and includes speculations.

A TLB entry is 77 bits long and contains beside the PPN and the TLBT, also a few flags. It is important that the PTE contains also the permissions bits.

V: valid bit determines if this entry contains valid data.

TLBT: is used to match and check if this is the right entry.

PPN: the address we are looking for.

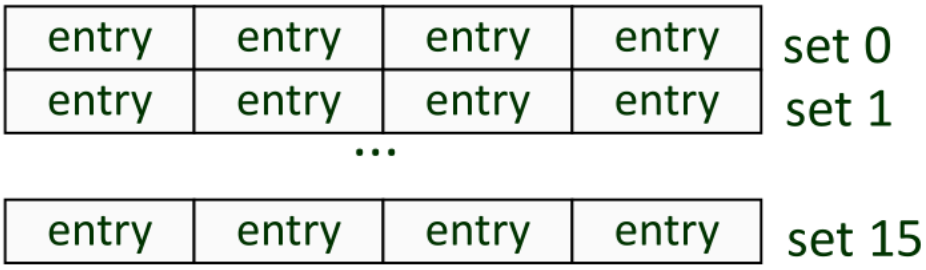
G: global bit tells if page is global or not (Global tables are not evicted on content switch).

S: supervisor flag tells if page is accessible by superuser only.

W: writ flag.

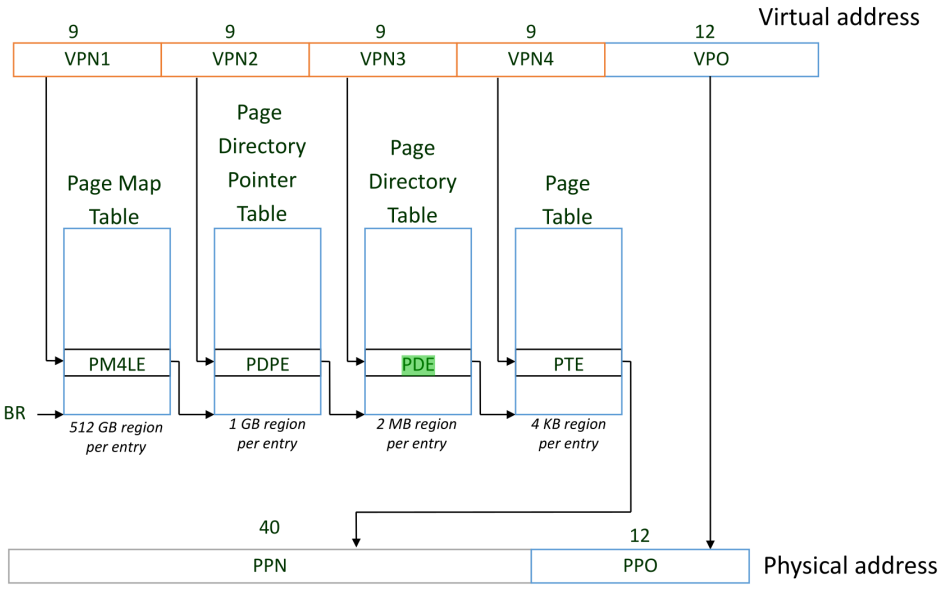
D: dirty bit tells if data of this page was modified and requires writeback to memory.

The TLB is structured as 16 sets with 4 entries. The 4 entries are used to cache the entries of the 4 layers.

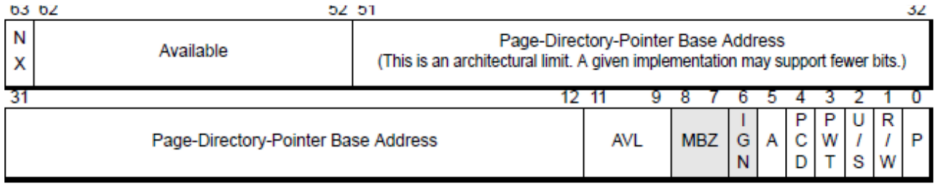


Paging *Paging* or *page walk* is the process of going though the different levels of the tage tables. In i7 the 4 tables have different names and are numbered in decreasing order starting with the *first* table.

All 4 tables have a entry size of 64 bits.



Level 4: Page Map Table This is the *first* table in the hierarchy.



Page Directory Pointer Base Address: 40 MSB are the pointer to the base address of level 3

Avail: Bits available to programmers for testing

A: access bit is set by the MMU on a read or write. Periodically, it is reset by a OS process. This way we can determine less frequently used entries.

PCD: is used to tell if a page should be cached or not.

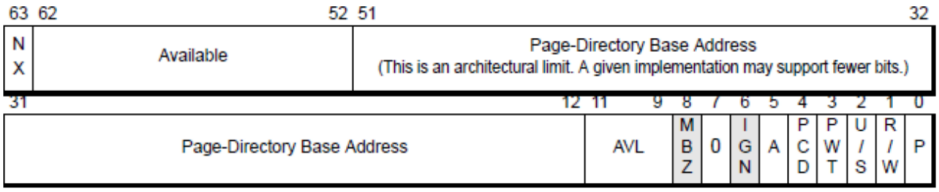
PWT: determines if we use write-through or write-back cache policy for this table.

U/S: determine is table belongs to user or supervisor mode.

R/W: determine read-only or read-write access

P: flag which tells if page table is present in memory or not.

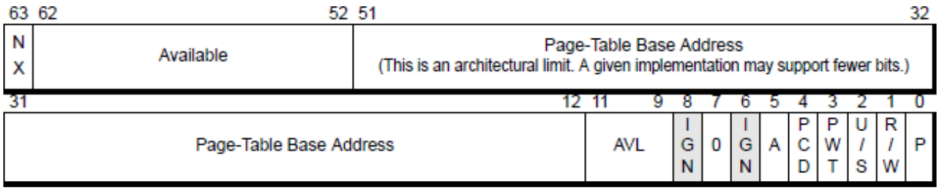
Level 3: Page Directory Pointer Entry Is appointed by the Page Map Table.



Page Directory Base Address: 40 MSB are the pointer to the base address of level 2

All other parts of the entry are equivalent to level 4.

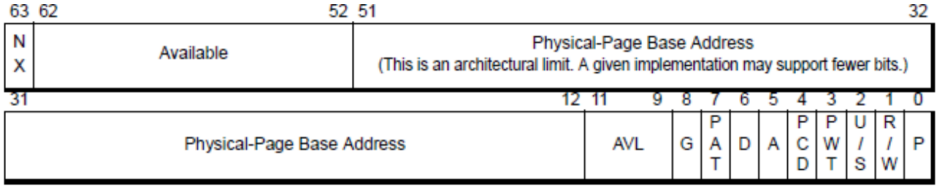
Level 2: Page Directory Entry Is appointed by the page directory pointer entry.



Page Table Phsyical Base Address: 40 MSB are the pointer to the base address of level 1

All other parts of the entry are equivalent to the previous levels.

Level 1: Page Table Entry Is appointed by the page directory entry.



Phsyical Page Base Address: 40 MSB are the PPN

Avail: Bits available to programmers for testing

G: global bit

PAT: page-attribute-table ???

D: dirty bit

A: access bit

PCD: cache enable/disable bit

PWT: write policy

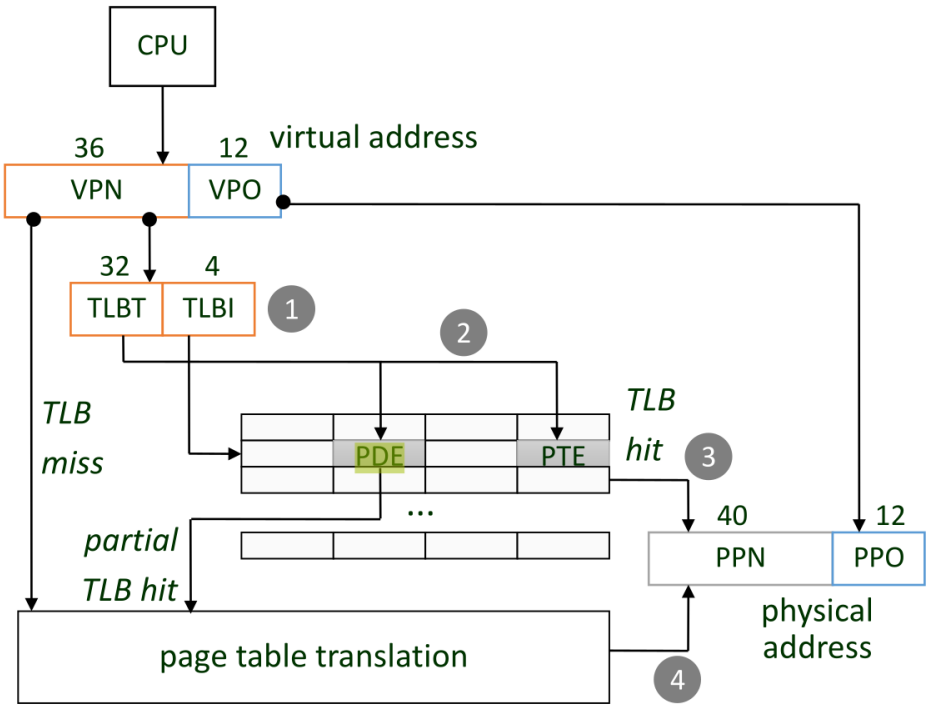
U/S: user/supervisor

R/W: read/read-write

P: present in memory flag

Translation After the MMU receives the submitted VA, it splits it into the VPN and VPO parts. Firstly, the TLB is checked for a hit.

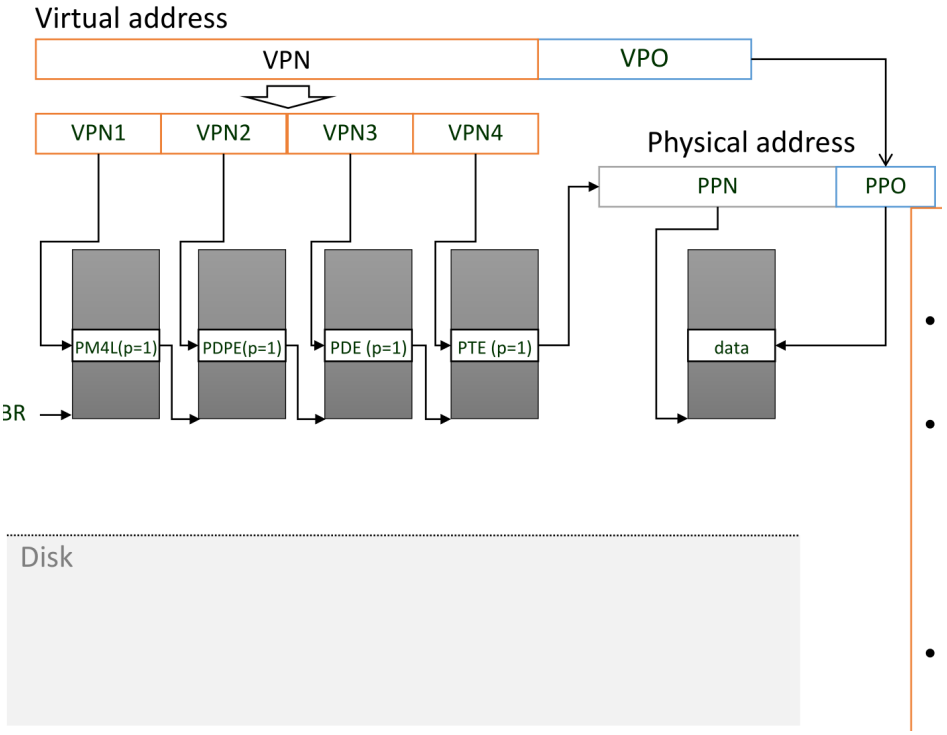
TLB Hit This is the ideal case for a translation.



1. The VPN is split again into TLBT and TLBI.
2. The TLBI is used to index the table and simultaneously, the TLBT is used to check for matching layer entries.
3. On a hit, the permissions are checked and if valid, the PPN is returned.
4. If no TLBT matches (complete miss) or of not all tags of the 4 entries matches (partial TLB miss), we have to get the missing data from the PT.

Page Translation (TLB miss) If we have a complete miss (none of the entries of the layers could be fetched) or a partial miss only some of the layer entries could be fetched), we have to page walk to get the PPN.

In the first example, all page tables as well as the page itself are present in memory. The MMU builds the physical address and can fetch the real data. This does not require any OS action.



In case that the page tables are present in memory but the page itself is in disk, de MMU triggers a page fault exception. The handler (OS) checks if the VA is legal and reads the PTE through the PT. It find free physical space and reads the VP from disk into the physical page. Then the handler adjusts the PTE to point to the PPN and sets the P flag to true. Then the faulty instruction is restarted.

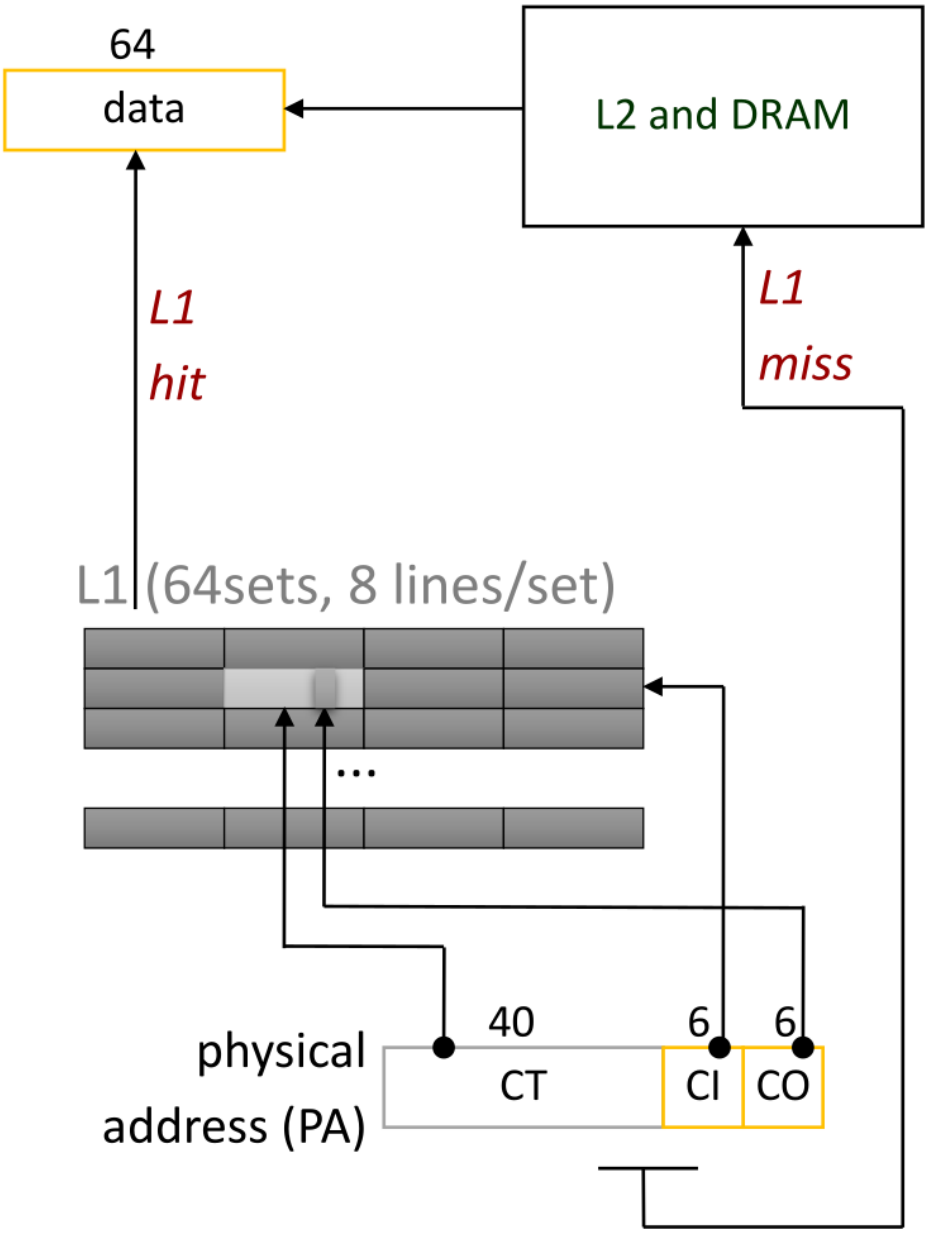
The page fault exception handler may not only be triggered on *non-present pages*, but also by *page-level protections*. The so called *Fault type*, is also given to the handler such that it knows the problem and can take appropriate action.

When one or multiple page tables as well as the page are missing, the MMU causes an page fault. As in the previous case, the OS brings the required data into memory, adjusts all flags and restarts the faulty instruction. Since this requires multiple consecutive disk reads, this is very expensive.

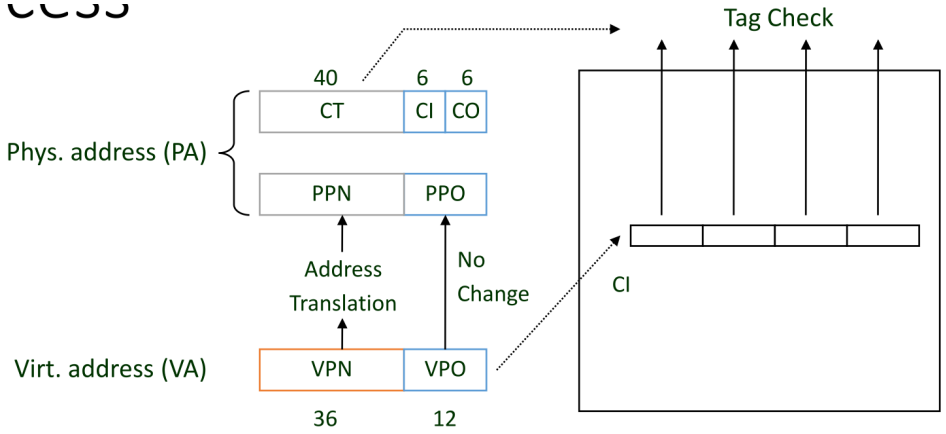
It may also happen that one ore multiple page tables are missing, but the page itself is present. This may introduce inconsistency issues and e.g. Linux disallows this.

Caches After the MMU translated the PA, we can check if the desired data is in cache.

First, the PA is split into CT, CI and CO. Using CI we index into the cached and check if we have a match using the CT. If we no not have a match the same procedure is repeated for L2, L3.... If we have a match, the word at offset CO is extracted.



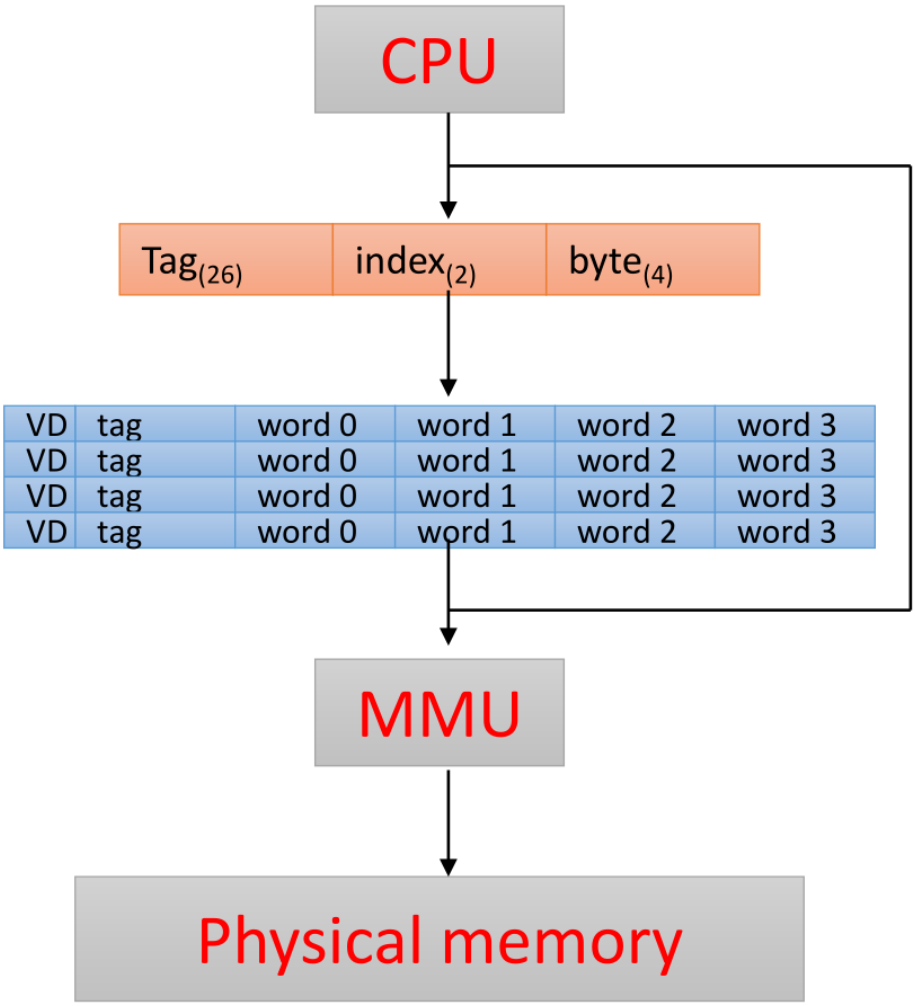
Even though L1 is rather fast, we can speed up the process even more. Since the CI is part of the PPO and hence does not change during the translation, we can start indexing the cache while address translation takes place. Since TLB hits most of the time, the PPN and with that also the CT bits get available rather quickly to validate cache matches.



Cache Addressing

Cache can be addressed using any combination of virtually/physical index and tag. The addressing also often differs for L1, L2 and L3.

Virtuall Indexed, Virtually Taged (VV) VV is also called virtually addressed cache and it can operate concurrently with the MMU. This makes it the fastest of all schemas for retrieving data.

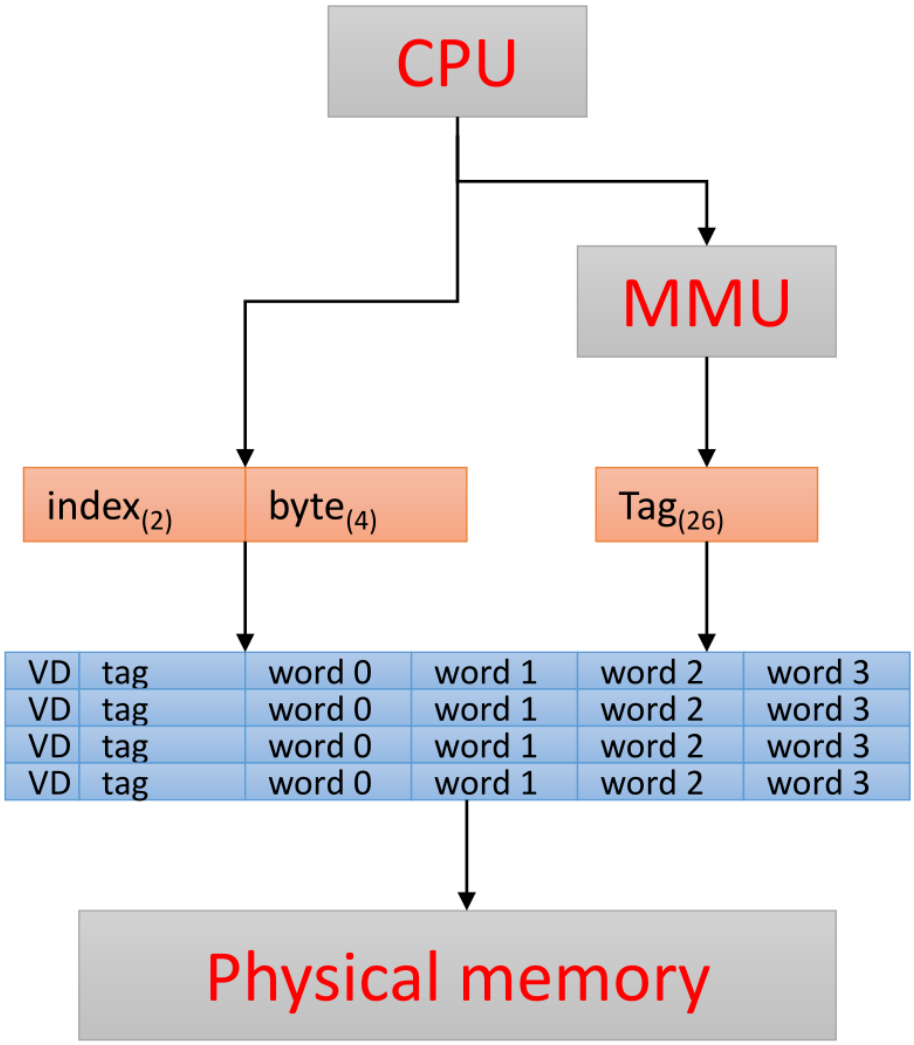


The main problem with this schema is *homonyms*. The same VA refers different PAs for different processes. So, the tag does not uniquely identify data which leads to problems when trying to get data from the cache. Two simple fixes would be to flush the cache on context switch or to force non-overlapping address space layouts. A more reasonable solution is the tag the VA with a *address space ID (ASID)*. On each access, the ASID of the cache is compared with the one of the process.

This resolved homonyms but leads to *synonyms*. Different VAs map to the same PA. Therefore, we may have the same page in cache multiple times.

Another problem is caused by the write-back, then his schema requires a TLB lookup on writebacks in order to get the address to write to.

Virtually Indexed, Physically Tagged (VP) In this schema the VA gives the CI and the PA the CT. A smart implementation using alignment is required to prevent homonyms. This schema is commonly used for L1 caches.



Physically Indexed, Virtually Tagged (PV) This schema is a bit pointless. If the PA has to be computed anyways we may easily also use the PA to get the CT.

Physically Indexed, Physically Tagged (PP) Since this schema only uses the PA, the PA has to be fully translated before we can start accessing the cache. This makes this type the slowest of all. But is is very simple to implement since there are no homonyms nor synonyms. This is often used for L2 or L3 caches.

Write Buffers Write operation take long time to complete. To avid stalling the CPU we can buffer writes on a FIFO *Write Buffer* queue. It also allows for reading data from the queue to prevent stalling the CPU for reading data not yet written to disk. However, this makes the memory content temporarily stale. This gets especially tricky in multiprocessor systems.

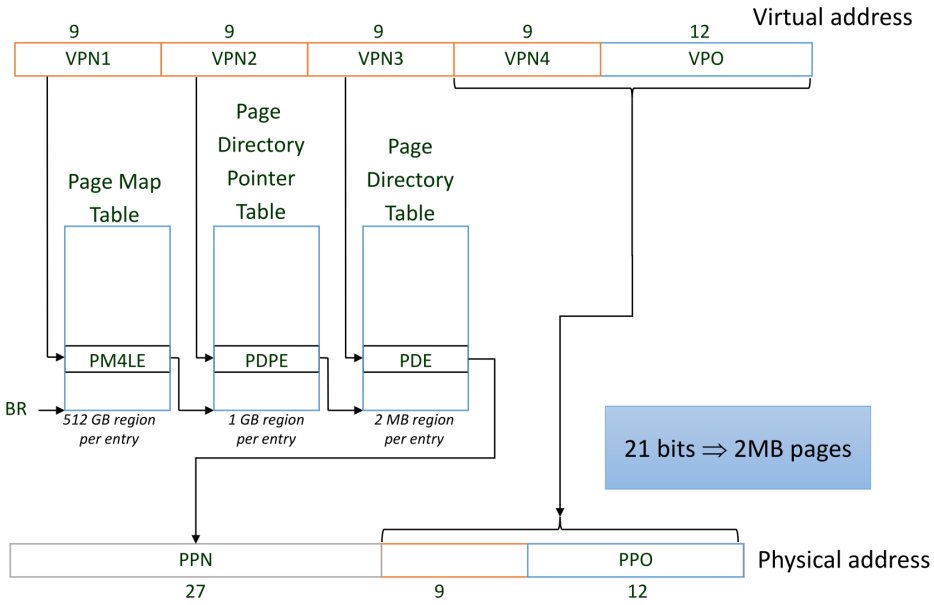
0.22 Lecture 22: Large Pages

Week: 12

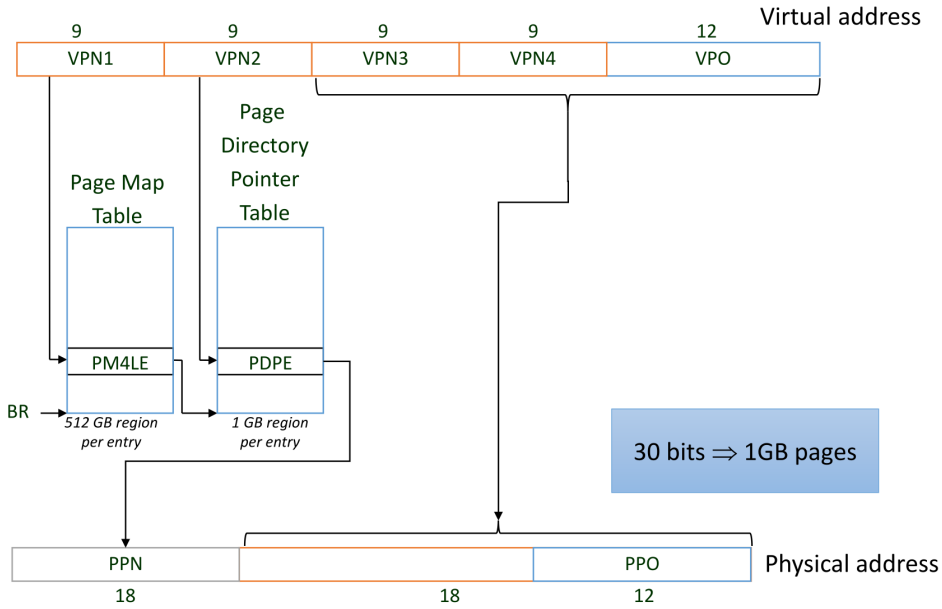
Large Pages

Sometimes, we want pages which are larger than 4KB. In that case we want pages of 2 MB size, we use large pages or if we want pages of 1 GB size we can use huge pages. All the hardware for larger pages is already there. We simple stop the page walk earlier.

For large pages, it skip the page table table 1 and go do memory using the address obtained from the page directory table.

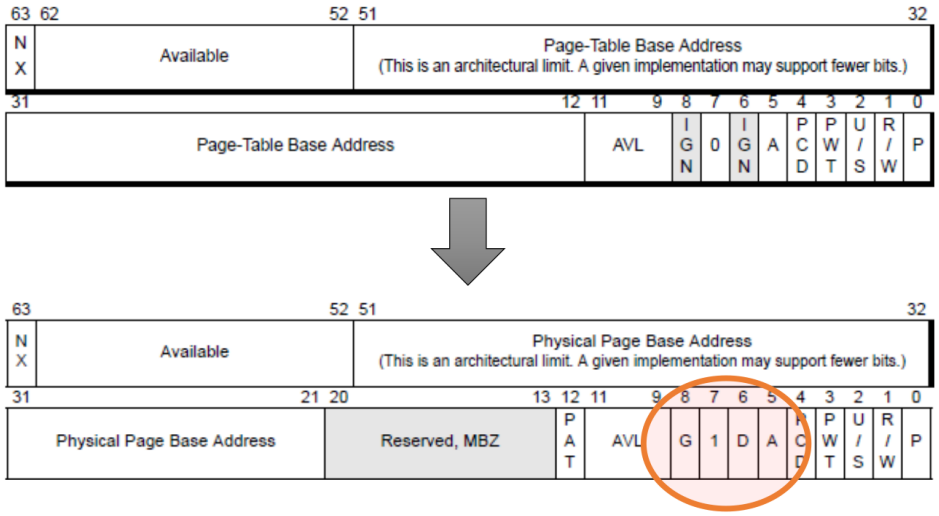


For huge pages, we skip additionally the page directory table.

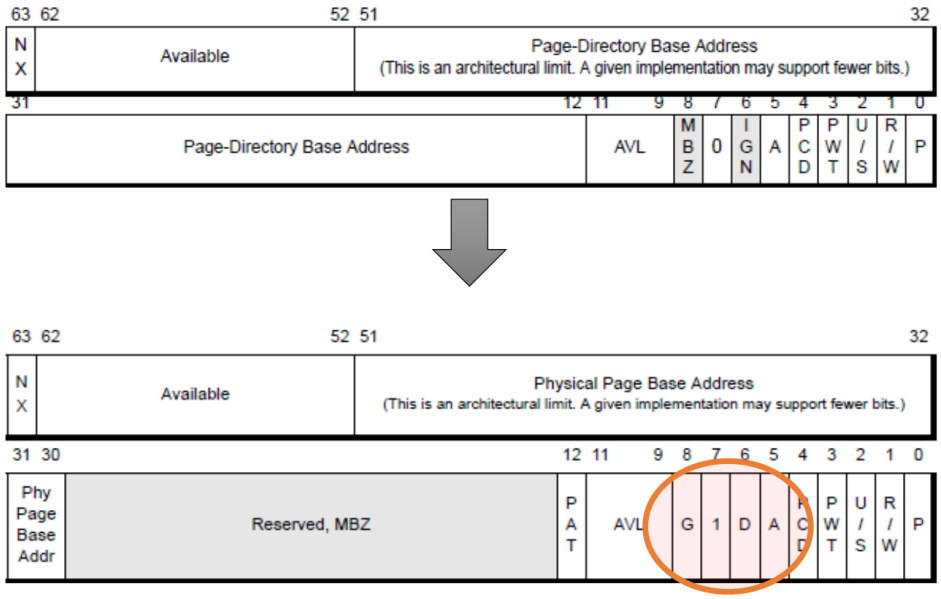


Some of the bits of the last table (page directory table for large pages, page directory pointer table for huge pages) need to be changes to represent the dirty bit as well as the global flap.

For large pages:



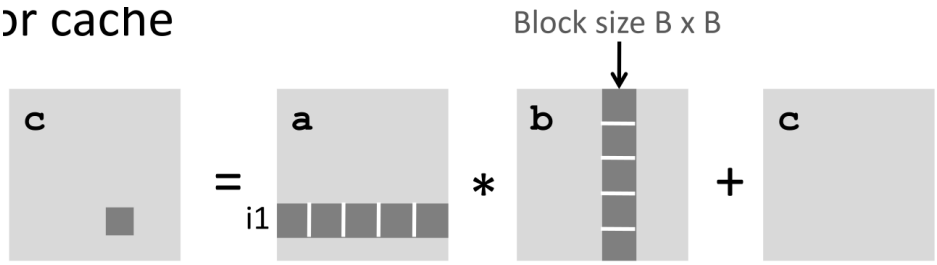
and for huge pages:



All in all, this simplifies and speeds up the address translation. This is very useful for programs with very large, contiguous working sets. Since the TLB is very small, reducing the number of pages increases the hit rate of the TLB. CPUs can do deal with different page sizes in parallel. In Linux, `hugetlbfs` is uses for support of larger tables and they get mounted like a regular file system.

Optimizing for the TLB

Beside caches, we also need to consider the TLB when optimizing code. Lets assume we want to do matrix multiplication followed by a matrix addition.



Let's say we optimize for the L2, which is 16MB $= 2^{24} = C$. It can store 2^{21} doubles. So to optimize for cache and we want to cache the $B \times B$ for matrix a, b, c. So we have $3B^2 < C$ which gives us $B \approx 800$.

But if a block has 800 rows of size 8 bits each (i.e. 6400 bits) is larger than a page size. Therefore, each row is on two pages and then we have this for 800 rows. This again, we have for the three matrices. The number of pages of a working set is $800 \cdot 2 \cdot 3 = 4800$ pages. That is a lot more than will fit the TLP.

Just looking at L2 blocking at this size looks ideal, but it is not for the TLB. It will cause TLB thrashing. To prevent this, it makes since the shrink the size of a block and copy the contents of each blocks into a contiguous region of memory to make them fit into a small enough number of page such that they can be hold in the TLB.

We assume a TLB size of 128 size. If $B = 150$ and the copy the content into a contiguous region of a memory we have a contiguous region of memory of size $150 \cdot 150 \cdot 8 \cdot 3$ bytes. Divided by 4 KB gives about 128 tage, which will fit into the TLB. Even though the copying cost $O(B^2)$ we do $O(B^3)$ operations on them. We also need extra space.

Multiprocessing

Introduction

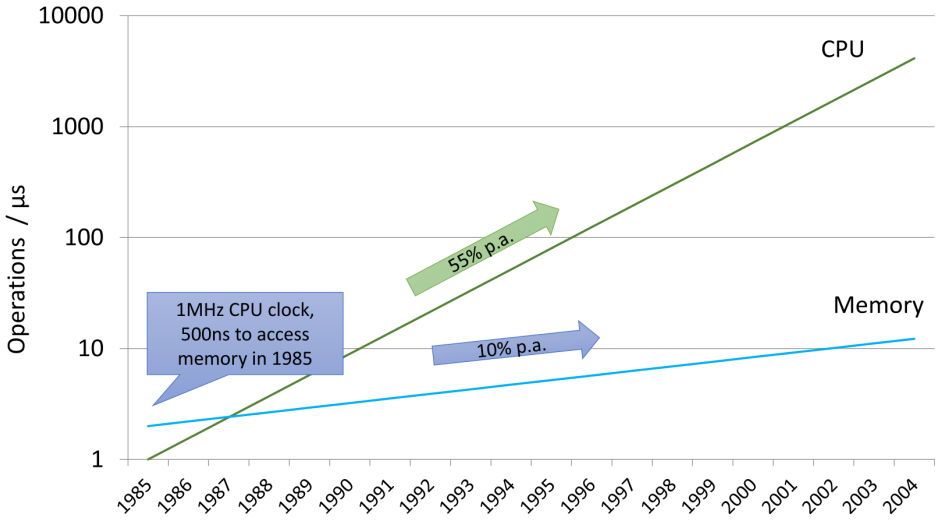
We are all aware of Moore's law; the number of transistors in a circuit doubles every two year, which is thanks to smaller components, which leads to higher clock speeds. But this has changes and therefore we need more cores.

Power Wall The power dissipation is $P_{\text{Dynamic}} + P_{\text{Leakage}} + P_{\text{Short}}$

Dynamic Power Used to be the main component. Now it is about 50%. It is causes by switching. It is equal CV^2f, where C is the capacitance V the supply voltage and f the processor frequency. According to Moore's law, C and V went down, so we could increase f for free. But today, C does not decrease. Decreasing V increases P_{Leakage} and increasing f increases the power dissipation.

Cooling is an issue too, we have pretty much reached the limit for cooling.

Memory Wall The memory call says that the CPU speed increases much faster than the memory bandwidth. This causes memory assesses to get more and more expensive.



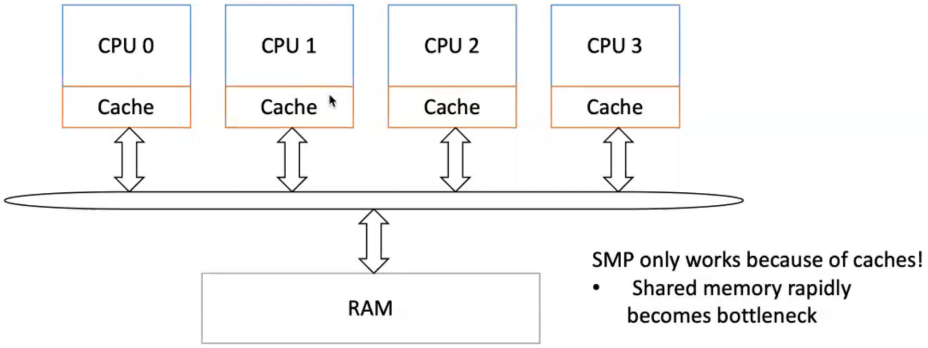
Instruction Level Parallelism (ILP) Wall We can do super scalar execution, branch prediction, specular execution to increase ILP. But as we add more transistors, the returns we get are diminishing. A reason for that is that we cannot extract more parallelism from a program than there exists.

Wall Summary

- Power wall: cannot clock processors any faster
- Memory wall: often, memory access dominates the performance
- ILP wall: cannot keep functional units busy while waiting for memory

Multicore Processors Multicore CPUs have multiple processor cores on a single CPU chip. Most such CPUs have a single physical address space, which is shared among all processors. The cores communicate though shared variables in memory. This is called *shared memory multiprocessors*.

It is also refered to as symmetric multiprocessing (SMP).



One major difficulty is how we ensure that all processors have the same view on memory even though hall have their own caches. This is called *cache coherence*.

Coherency and Consistency

Coherency Cache coherency is about the view on memory by all processors. It means that even though all processors have the same cache, they need a coherent view of memory. I.e. all processors must see the same data on a certain memory location. Most modern CPUs are actually cache coherent.

The main advantage of this s the ease of programming. But is it more complex to implement.

Consistency Memory consistency is the order in which changes to memory are soon by different processors. It is impotent to to have a clear definition.

Program Order Program order is the order in which a program on a processor appear to issue read and writes. It only referees to local read and writes, since it is what is seen by other processes. But the CPU my rearrange other instructions (out-of-order execution, write buffers etc.).

Visibility Order Visibility order it the order in which all reads and writes are seen by other processors. It referees to all operations in the machine. Different processors may have a different visibility order.This means that each processor reads the value written by the last write in visibility order.

Sequential Consistency

It is conceptionally the simplest consistency model. Operation from a processor appear to all other processors in program order. And all processors see (visibility order) the same interleaving of the program orders of the different processors. I.e. the visibility order of each processor is the same.

Requirements This requires:

- each processors issues memory ops in program order (i.e. each processor obeys their program order)
- RAM orders all operations is receives in the order it receives them (it has a total order)
- memory operations are atomic

Example

Ex1 Given two CPUs, A and B, where A writes two variables and B reads these variables. Initially, all variables are 0.

CPU A		CPU B	
a_1:	*p = 1;	b_1:	u = *q;
a_2:	*q = 1;	b_2:	v = *p;

A valid outcome in SC is `u=1, v=1`, received when A has program order `a1, a2` and B has `b1, b2` and the interleaving is `a1, a2, b1, b2`.

Could we get `u=1, v=0` in SC? We see that this is not possible since `a2 > b1 > b2 > a1` which does not respect program order of the processors.

Ex2 Given two CPUs, A and B, where both processors first write a variable and then read a variable. Initially, all variables are 0.

CPU A		CPU B	
a_1:	*p = 1;	b_1:	*q = 1;
a_2:	u = *q;	b_2:	v = *p;

A valid outcome in SC is `u=1, v=1`, received when A has program order `a1, a2` and B has `b1, b2` and the interleaving is `a1, b1, a2, b2`.

Could we get `u=1, v=0` in SC? We see that this is not possible since `a2 > b1 > b2 > a1` which does not respect program order of the processors.

Advantage / Disadvantage

- Advantage:
  - Easy to understand for the programmer
  - Easy to write correct to to
  - Easy to analyse automatically
- disadvantage:
  - Hard to build fast implementation
  - Cannot reorder reads/writes (even in the compiler or single processors)
  - Cannot combine writes to same cache line (write buffer)
  - Serializing operations at memory controller is too restrictive

Cache Coherence With Snooping

How do we ensure a coherent view of memory. Snooping is a possible solution. *Snooping* is the act on listening on the bus for reads/writes of other processors. If we have a local copy of a cache line and a other processors writes the same line, we invalidate our local cache line. But for this to work,

caches must be write-through (else memory is not directly updated on a write and hence, we do not get notified). Obviously, multiple processors can have the same cache line in cache.

Write-Back Caches For the snooping to work out of the box, we need write-through caches. Though, often systems have write-back caches two and now cache liens can be dirty. For this to work, we need a *cache coherency protocol*.

MSI

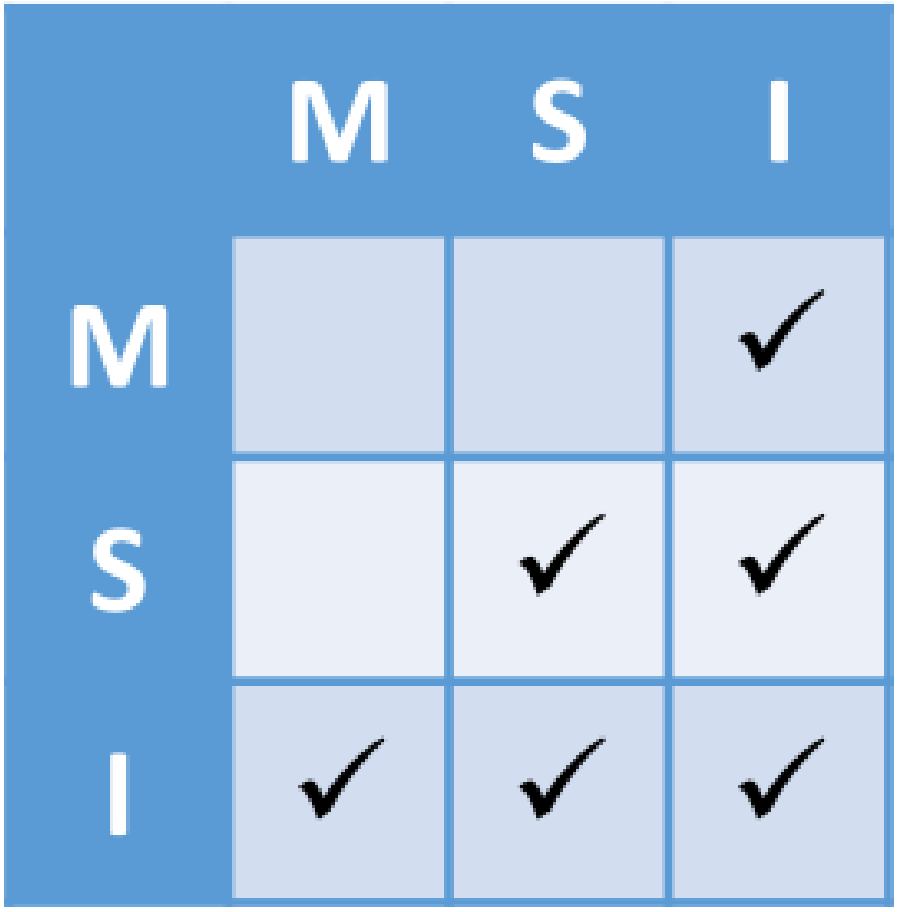
A cache line can be in one of three states:

- *Modified*: we have changed the cache block (it is dirty)
- *Shared*: we read it and there are possible other readers
- *Invalid*: we have not cached this line

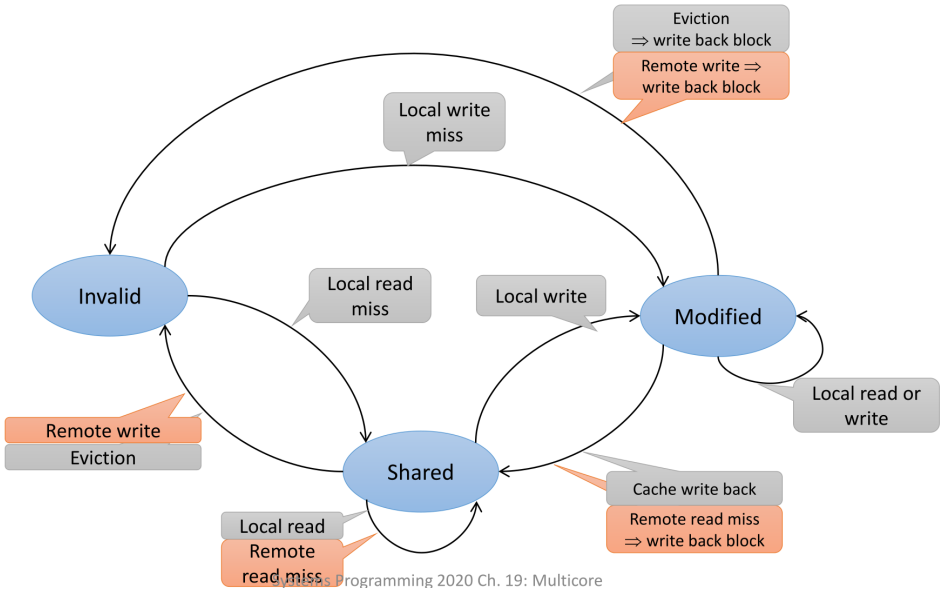
The state we are in depends on

- the action we take
- what we hear on the bus (i.e. actions of other processors on the same cache line).

Invariants The MSI invariants are:



Transitions We can have the following transitions. Grey are local operations by us, orange remote operations by other processors.



Issues It causes unnecessary bus messages. E.g. if we are in shared and want to write, we first have to check if there are other readers. If we are the only reader, this check is redundant.

Also, the protocol assumes that we are able to distinguish remote processors read and write misses. But in practice this is often not the case.

MESI

Compared to the MSI, it has an additional *Exclusive* state.

The meaning of the states is the following.

When in this state, we know that this is the only copy of this address and the address is clean.

- *Modified*: we have the only copy of this cache block and we have changed the it (it is dirty)
- *Exclusive*: we have the only copy of this cache block and it is clean
- *Shared*: there may be several copies of this cache block and all are clean
- *Invalid*: (same meaning)

This protocol also introduces the new bus signal *RdX (Read exclusive)*. If we want to read a cache line and others are reading this block already, we are notified that we read it as *shared*. If there are no other readers, we are notified that we are reading it as *exclusive*. We get this response through the *hit* statement.

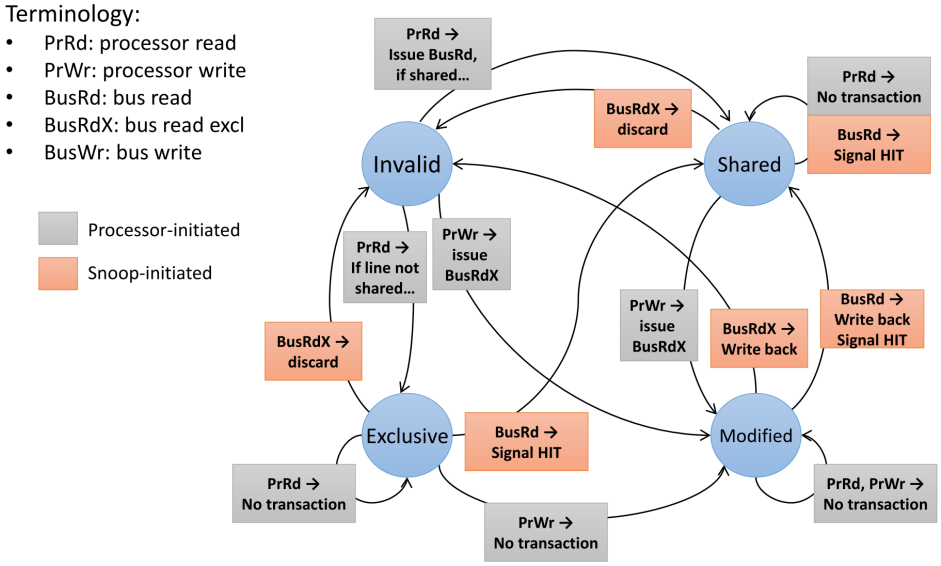
If we hear the RdX signal, it means that another bus is trying to write this block and therefore, we have to invalidate it.

Invariants The MESI invariants are:

	M	E	S	I
M				✓
E				✓
S			✓	✓
I	✓	✓	✓	✓

The main advantage over MSI is that when we are in exclusive and we want to modify it, we can straight do it without having to notify anyone else.

Transitions We can have the following transitions. Grey are local operations by us, orange remote operations by other processors.



Observations Data is always first written back to memory, before another process can read it. If data is dirty, we need to write it back to memory. If data is clean, other processes can directly fetch it from main memory. There is no cache-to-cache transfer. So, MESI is good if going to memory is

faster than going to remote cache (but often this is not the case).

0.23 Lecture 23: Continue: Coherency Protocols

MOESI

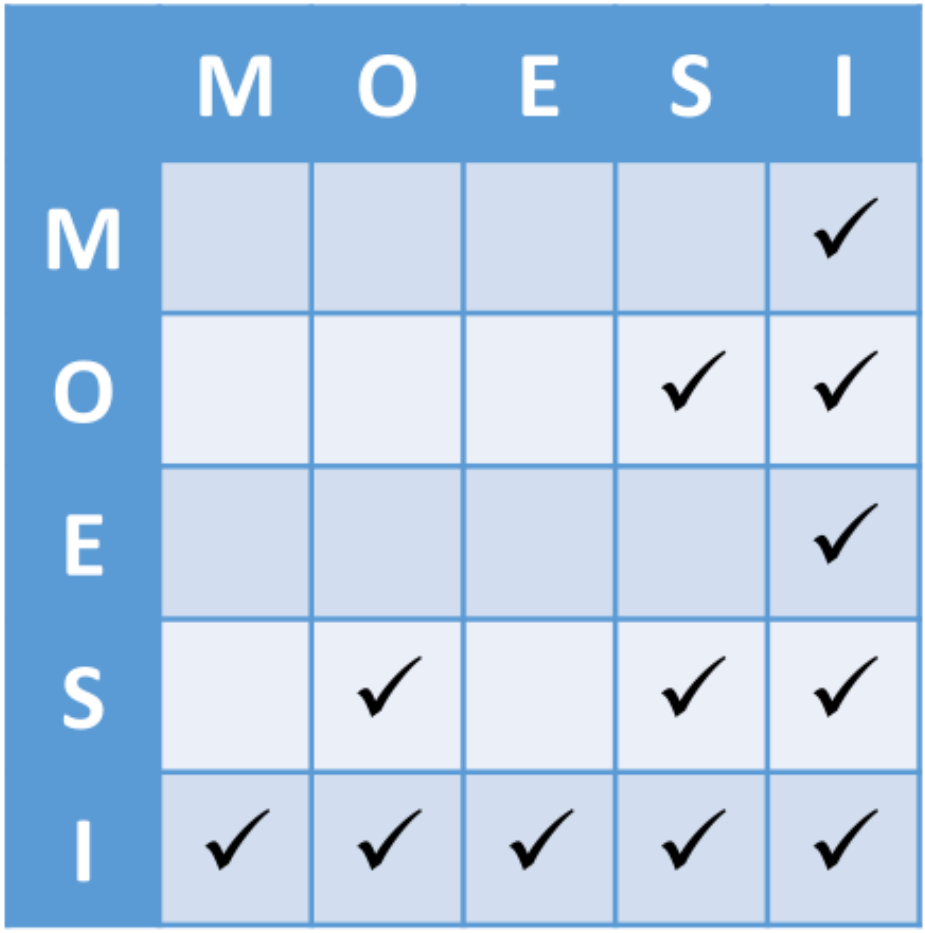
Compared to the MESI, MOESI added an *Owner* state.

- *Modified*: (same meaning)
- *Owner*: there may be several copies of this cache block. We are the only ones who are allowed to modify it but we need to let the others know about the new value.
- *Exclusive*: (same meaning)
- *Shared*: there may be several copies of this cache block. Only one other process has the right to modify it.
- *Invalid*: (same meaning)

The owner state allows to share dirty data, additionally to clean cache lines as it it in MESI. So it allows for quickly satisfy read requests for dirty cache blocks without writeback to memory. Accesses to clean data is still done via memory.

This protocol is good if going to remote cache is cheaper than going into main memory.

Invariants The MOESI invariants are:



MESIF

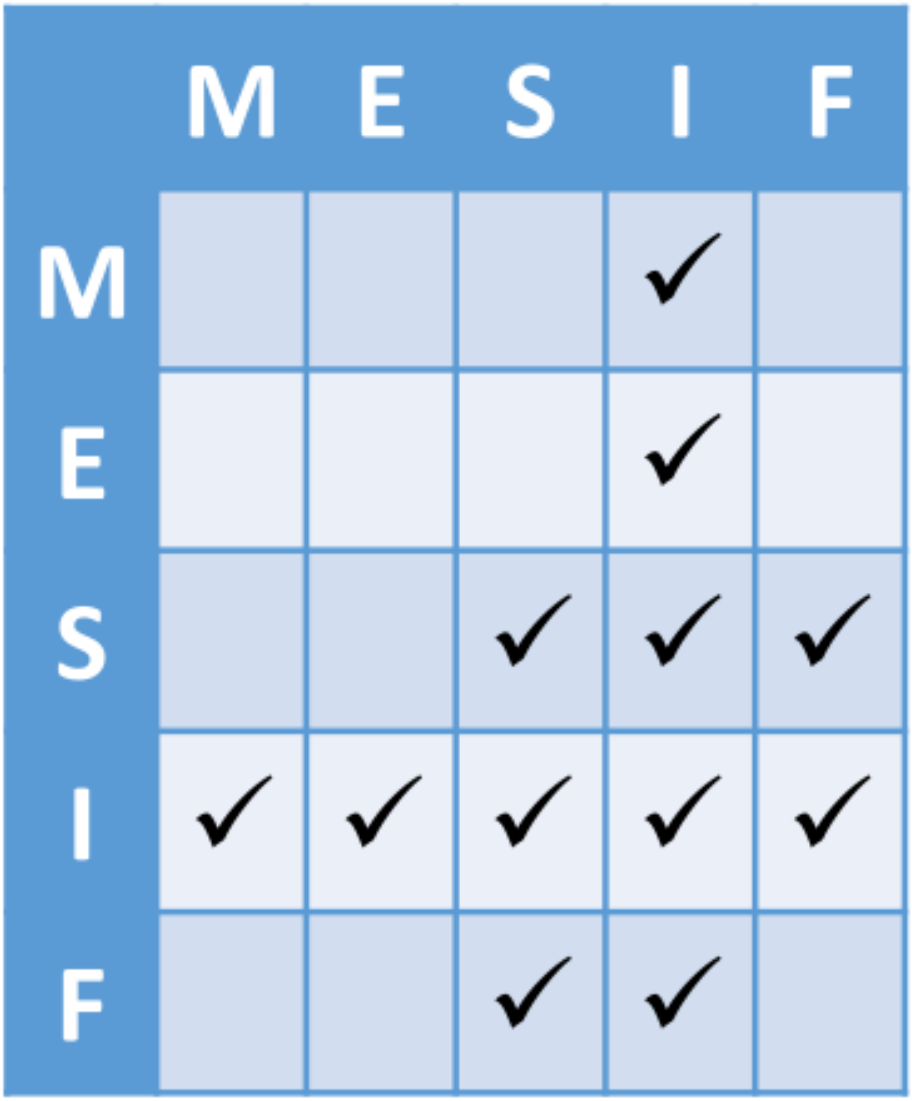
Compared to MESI, MESIF adds an *Forward* state.

- *Modified*: (same meaning)
- *Exclusive*: (same meaning)
- *Shared*: there may be several copies of this cache block. There is at most one forwarder, whose job is to provided the new line in case it is changed to all others.
- *Invalid*: (same meaning)
- *Forward*: as shared, but this one is the designated responder for requests.

This new forward state allows for cache-to-cache forwards. The new cache block is transferred without memory access.

If there is no forwarder, data is still serverd by main memory. This is even the case if the cache line is shared.

Invariants The MESIF invariants are:



Relaxing Sequential Consistency

SC required to obtain the program order of processors. But for good performance it is required that we are able to reorder instruction. I.g. Out-of-order execution might reorder instruction, write buffers reorder write instruction, given $a1 < a2$, a1 may be a cache miss while a2 is a cache hit. Also, compilers may reorder or remove reads and writes for performance optimisation reason.

Relaxing Sequential Consistency When we relay the requirements for SC, we can make use of the described efficiency gains. There are many different ways to achieve this:

Write-to-Read: later reads can bypass earlier writes

Write-to-Write: later writes can bypass earlier writes

Break write atomicity: no single visibility order (different processors can see instructions in different orderings)

Weak ordering: no implizite order guarantees at all

The more we relaxe the constrains, the less and less guarantees about the model we have and the more and more difficult it gets to reason about modes. But the cheaper and faster implementations we get.

To notify when we need again synchronisation, CPUs provide explicit synchronisation instructions (load fence, memory barrier). I.e. by default CPUs can reorder, but at a certain point (e.g. when entering a critical section) we an request synchronisation.

Processor Consistency This is a certain relaxation of SC and it is the default for x86-64. It is also referred to as *Total Store Ordering (TOS)*.

It provides the write-to-read relaxation.This means that all processors see writes from one processor in the order they were issued by that processors. But processors can see many different interleaving of writes from different processors.

This provides a noticeable performance improvement.

Example Recall our earlier example:

CPU A		CPU B	
a_1:	*p = 1;	b_1:	*q = 1;
a_2:	u = *q	b_2:	v = *p;

Using SC it is not possible to get `u = 0, v = 0` but using PC we can get this results. The read instruction a2 can bypass the a1 write resulting in reading 0 before writing 1.

Other Consistency Models There are also many other relaxations and different architectures support different ones:

	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	✓							
Ifetch after write	✓		✓	✓	✓		✓	✓

Icache is incoherent: requires explicit Icache flushes for self-modifying code

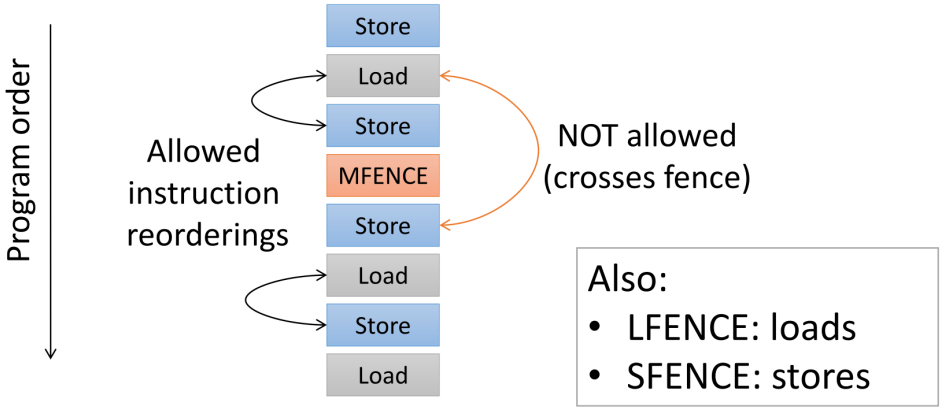
Read of value can be seen before read of address of value!

Barriers and Fences

Barriers/fences provide a way for synchronisation in a weak consistency model. There are two main types:

Compiler Barriers Compiler barriers prevent the compiler from reordering memory loads and stores, but they may still reorder register accesses (since they are private to the compiler). In gcc we would use `__asm__ __volatile__("" ::: "memory");` to tell the gcc compiler to use such a barrier. This code is called compiler intrinsic.

Memory Barriers Memory barriers prevent the CPU from reordering load/store instructions before a barrier with load/store instructions after a barrier. They are like a *wall* for instructions. For this the `MFENCE` instruction is used.



There are also fences for only stores `LFENCE` and store `SFENCE`.

Multicore Synchronisation: Test-and-Set

There are two different ways so synchronize across processors:

- *Atomic operations*: Allow doing a series of operations without interruption by other processes
  - e.g. test-and-set, compare-and-swap
- *Interprocessor Interrups*: when processors interrupt each other. This is e.g. used in OSs (not really part of this course)

Test-And-Set (TAS) TAS is one of the simplest non-trivial atomic registers and is is implemented in HW. Firstly, we read value stored at a certain memory location into a register. Then we store 1 into that location. This is all done atomically. So the memory bus must be blocked in the meantime.

Spinlock

We can use TAS to implement a spinlock. To acquire a mutex we use

```
void acquire( int *lock) {
    while ( TAS(lock) == 1)
        ;
}
```

and to release it, we simple set it back to 0

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

`TAS(someAddr)` reads the value which is at `someAddr` and then writes 1 into the lock. After, the read value is returned.

In the case when 1 is returned, we know that the lock is already locked and hence we retry till we get a 0.

This is only efficient if we know that there are not many processes waiting for the mutex and that a lock is only keeps for a short amount of time, because

- memory must be locked while lock is locked
- atomically read and write memory over and over
- busy waiting requires CPU usage

Test And Test-And-Set This provides great improvements over the normal TAS. The inner while loop is not done atomically and it does also not involve any writes. Therefore, it is rather cheap to busy loop here. The expensive TAS is only done when the inner while loop detects that there is a chance of being successful.

```
void acquire( int *lock) {
    do {
      while (*lock == 1);
    } while ( TAS(lock) == 1);
}
```

In general, we should not use this in practice. Compilers and processors may reorder the instructions and the memory consistency may break the code's semantics too. While TAS is implemented in HW, Test and TAS is not.

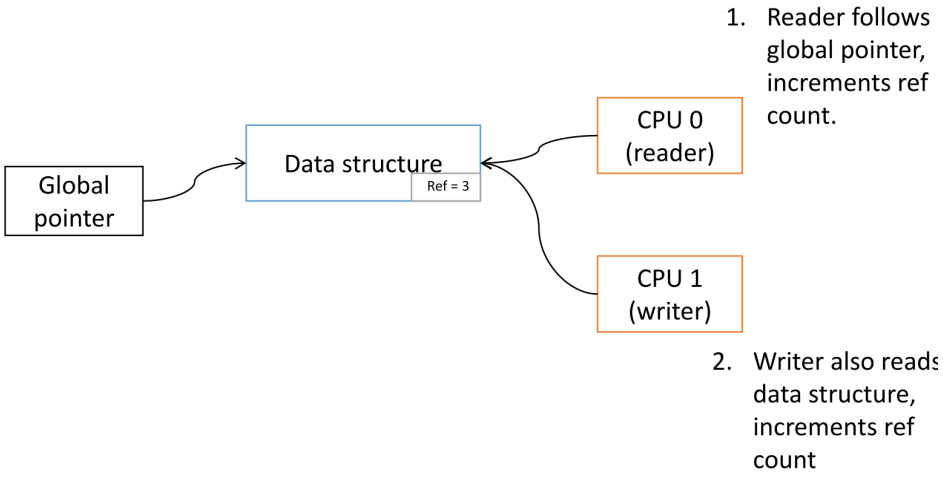
Compare and Swap

This is another atomic operations. It is more complex than TAW but also more powerful. It is commonly used for wait-free data structures. It takes three parameters, a address and two values:

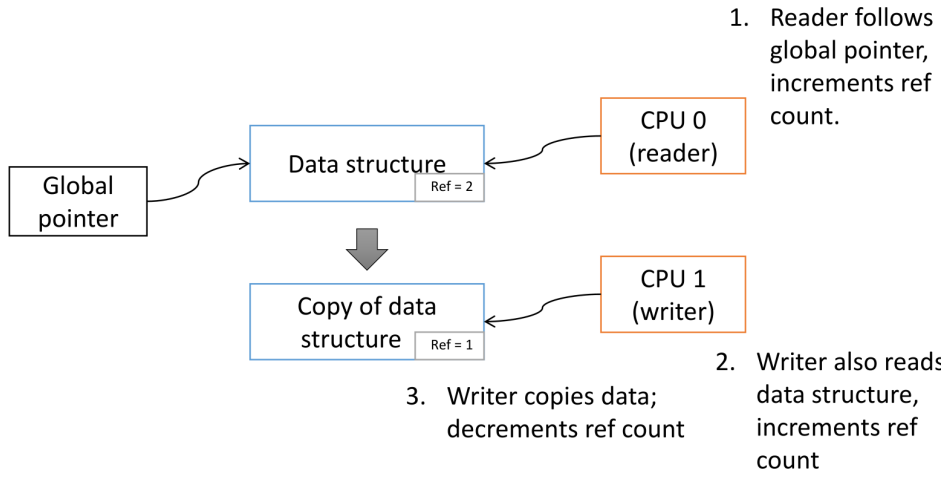
CAS(location, old, new) atomically
{
1. Load location into value
2. If value == “old” then store “new” to location
3. Return value
}

Often, the location is a pointer and we need to update where the pointer points to.

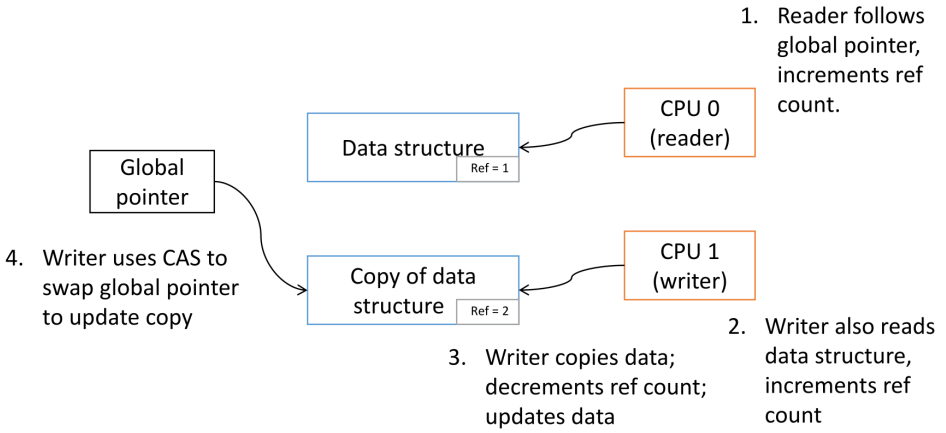
CAS Lock-Free Update In general we do a copy of the DS which we modify. When none has changed the original one, we make our copy the new original. The CAS is used on the pointer.



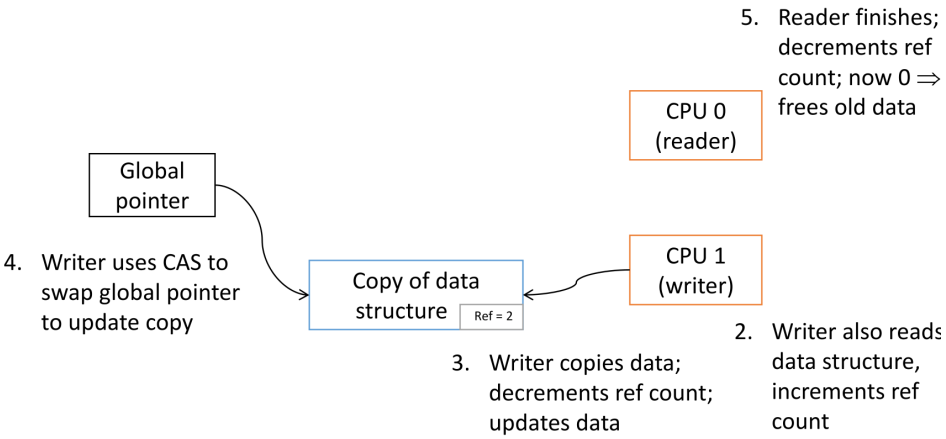
Given a DS, a reader and a write. `Ref` is the reference count to the DS. The reader increments the ref and starts reading. The writer increments the ref, copies the DS, decrement the ref and then modifies it. When done, it wants to make the copied DS the new original one.



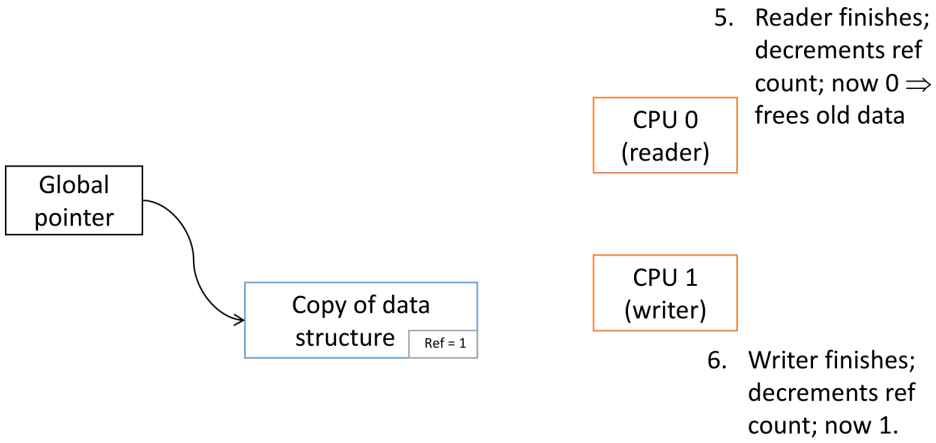
Using CAS, we adjust the global pointer to point to our new copy, as long as the global pointer did not change between making our copy and storing our copy.



After global pointer is updated and there are no reader to the old original anymore, the old original id discarded.



When the write finishes, we decrement the ref



ABA-Problem The problem with CAS is, that when there is a change to the value, and the another change, which changes it back to the initial value, we cannot detect that there was actually a change in the meantime:

- A reads value as x
- B write y to value
- B writes x as value
- A reads value as x and wrongly assumes it is untouched

Sometimes this is not really an issue, but sometimes it can break everything.

I.e. the value did not change while the data changed. To solve this, we must ensure that the value always changes. This can be done e.g. by splitting the value into the original value and an increasing counter.

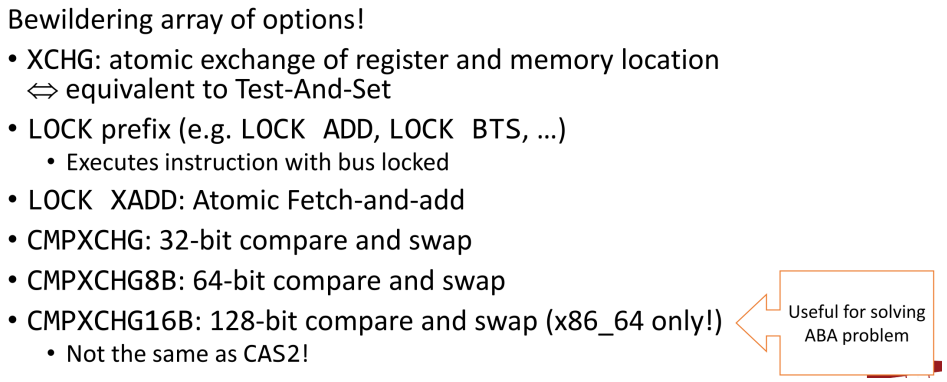
Double Compare-And-Swap Double CAS, aka CAS2 or DCAS takes not only 3 arguments, but 6. It takes two memory locations, two times the old data and two times the new data. Then it does (atomically):

1. Compare the memory locations
2. If both match, each is updated with a different new value
3. If not, the existing values in the locations are loaded

This is is claimed to be more powerful than CAS, it is rarely implemented tue to its complexity.

x86

On this architecture we have:



Atomiv exchange of registers is pretty much TAS.

Simultaneous Multithreading

This is called hyperthreading too (by intel).

SMP Performance Limits Memory is the bottleneck for cache-coherent SMPs. All accesses to main memory stall the processor. While MOESI allows to cache-to-cahce transfer, cache themselves can be slow too. Stalls caused by memory asses will most likely halt the complete processor (unless we can extract enough ILP).

Simultaneous Multithreading Many parts of the CPU are idle at a given time and many instructions do not required the memory units. But ILP is very limited. It there a way to make use of the processor while it is waiting for memory? The question arises if we cannot use the superscalar idea to make use of idle parts of a stalled core of a multicore CPU.

Types of MT There are three types of multithreading.

A color represents works of a certain process, white boxes are ones which could not be filled due to low ILP

Coarse-gain MT: We switch the thread as soon as we hit a time consuming instruction, e.g. a cache miss.

Fine-Grain MT: We switch the thread much more often, e.g. after each instruction.

SMT: In the same cycle we execute instructions of different threads.

In Practice A modern CPU has often two hyperthreads per code and they are exposed to the OS as separate threads. Therefore, the OS tells us that we have twice the number of threads than we actually have.

Advantages/Disadvantages It is not always a win, but it provided 10 to 20 percent performance boost and it is rather cheap since the hardware is already there. Tough in practice, two physical cores are more performant than two hyperthreads (running on the same core) (which is not very surprising). Hyperthreading can be disabled.

0.24 Lecture 24: Non-Uniform Memory Access(NUMA)

Week: 11

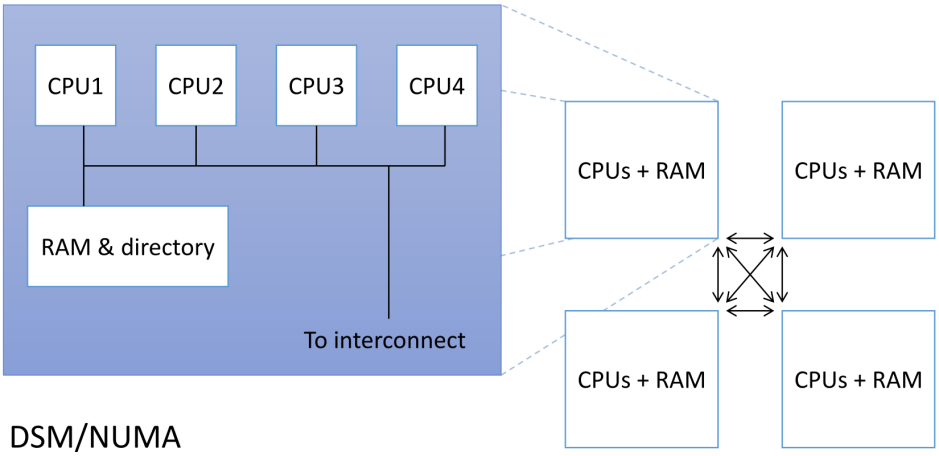
Non-Uniform Memory Access (NUMA)

SMP Limitations SMP is not well suited for scalability. While we may add more cache to compensate for adding more cores, the main bottleneck is the memory bandwidth. Using a memory bus,

only one CPU can communicate via the bus at the time. NUMB arises due to theses limitations. In this architecture, the memory bus is replaced by a interconnection network.

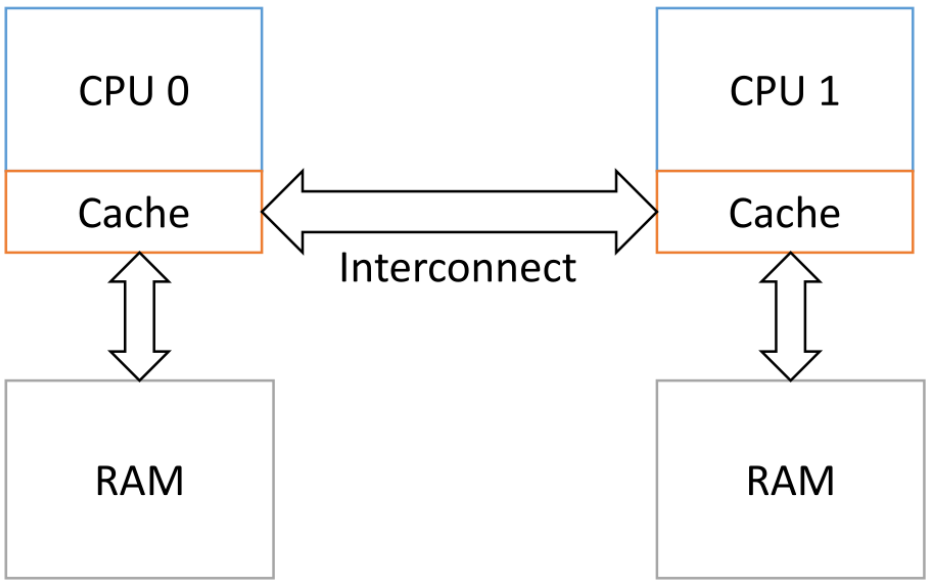
NUMA is pretty common in modern systems (mainly servers).

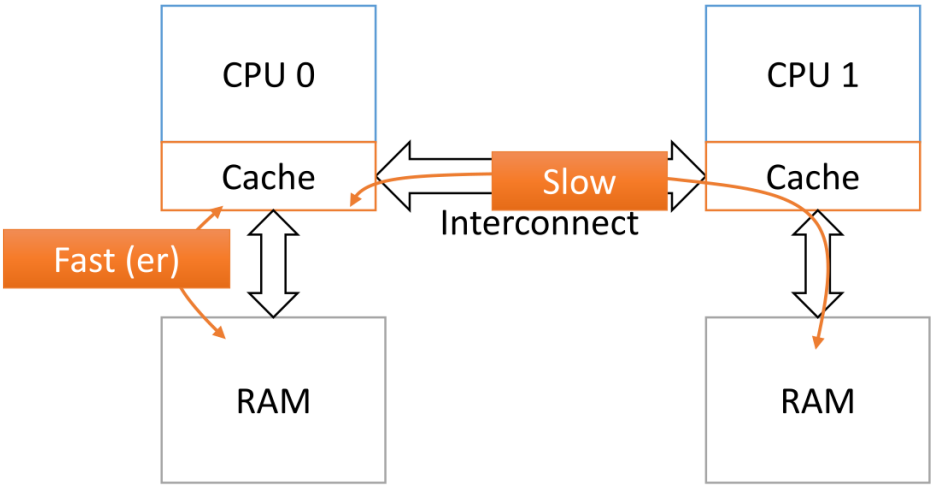
Distributed Memory Architecture There are multiple slices, each containing one or multiple CPU cores, own ram and directory and a interconnection part to other slices. Adding more CPUs (more slices) brings more of most other things too (RAM etc.). This allows for scalability to 1000s of cores. The interconnects are not buses but it is network like. It carries messages between nodes like read/write requests/responses, cache invalidates etc.



Non-Uniform Memory Access Since the RAM each slice has is smaller than the main memory in SMP, access times to local memory faster. Access to RAM of other slices is possible to due to the shared address space (there is one single address space), but typically is much slower. The problem is how to figure out where certain data is and depending on its location, access takes non-uniform amount of time.

Another difficulty which arises from splitting cache is how to keep caches coherent.





Cache Coherence

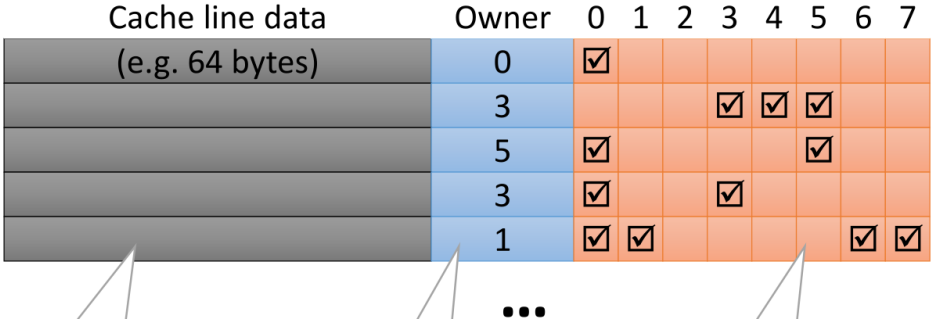
CPUs cannot snoop anymore since there is no public bus. Possible solutions to overcome the resulting cache coherence problem are (we still want to be able to used the introduced cacher coherency protocols):

Bus emulation A shared bus is emulated, which allows CPUs to snoop again. In practice, this requires that each nodes sends a message to all other nodes and waits for a response from them before proceeding. This implementation is rather complex.

Simulate a bus, everyone sends messages to everyone else.

Cache Directory For each cacheline we have some extra information. Together it is called cache directory. It stored the data itself, as well as things like who owns this data and which CPU node has this line currently. This way we do not need to broadcast messages related to this memory block to everyone but only the nodes which have a copy of this.

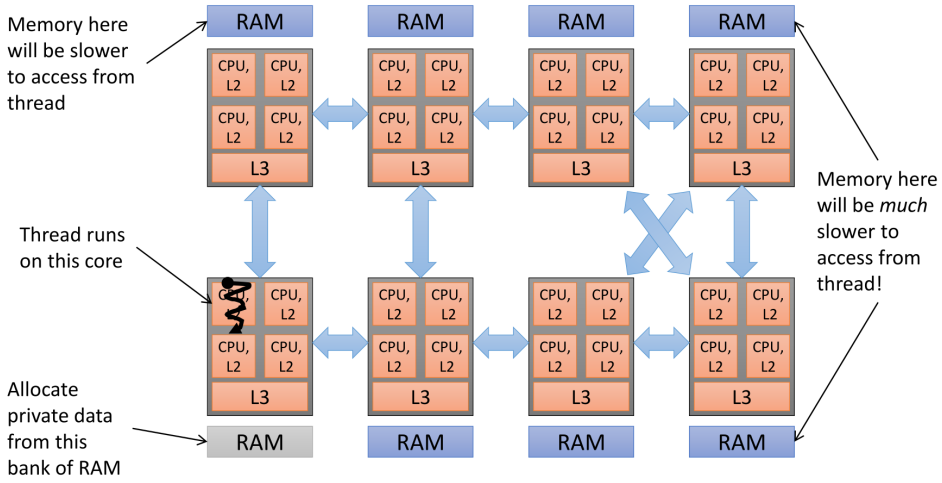
This method is useful when data is not widely shared among nodes and the when there are many NMUMA nodes.



This is used in large multiprocessors.

Performance Implications of Multicore

Example: 8-Socket 32 Cores AMD Barcelona This architecture has a core local L1 and L2 cache, and a node shared L3 cache. Access of node local RAM or L3 is rather fast while access to other node's RAM or L3 gets slower the more hops we have to do.



This typology is slightly unusual. It was split into two boards and the connection between the two boards is more expensive. Messages are not routed in the obvious way to make sure that not too many packages are sent at the time (or something like that).

The left side has an additional PCI connection and therefore there is no crosslink.

L3 is unified for the socket. All misses of the non-shared L2 goes to the L3. A line may be not in our L2 and not in our L3 but in someone else L2. That is a little weird but not unusual.

False Sharing For a single CPU it is optimal when we have good locality, i.e. do many operations to the same cache line. But when there are multiple threads of different CPUs on different sockets which do that, we get a ping-pong effect between the caches on every write. This is very bad.

Optimisation Example: MCS Locks

There are many data structures which rely on locks. For NUMA architectures, cache lines containing a lock are a hot spot since many different thread on multiple sockets try to read and write it. This gets very expensive due to ping-pong of the cache line. The idea if MCS (acronym for Mellor-Crummey and Scott) is that threads spin on local data and only one processor wakes up and acquires the free lock.

This is possibly the best multiprocessors locking system which we have.

Implementation The idea is that we have a linked list. A thread which wants to acquire the lock adds itself to the linked list. Once the lock is released, the next CPU in the list is notified which then quires the lock.

```
struct qnode {
        struct qnode *next;
        int locked;
};
typedef struct qnode *lock_t;
```

Acquire

1. Add ourself to the end of the queue using `XCHG`.
2. If the queue was empty we acquire the lock.
3. If the queue was not empty, point previous tail to us and spin on local cache line.

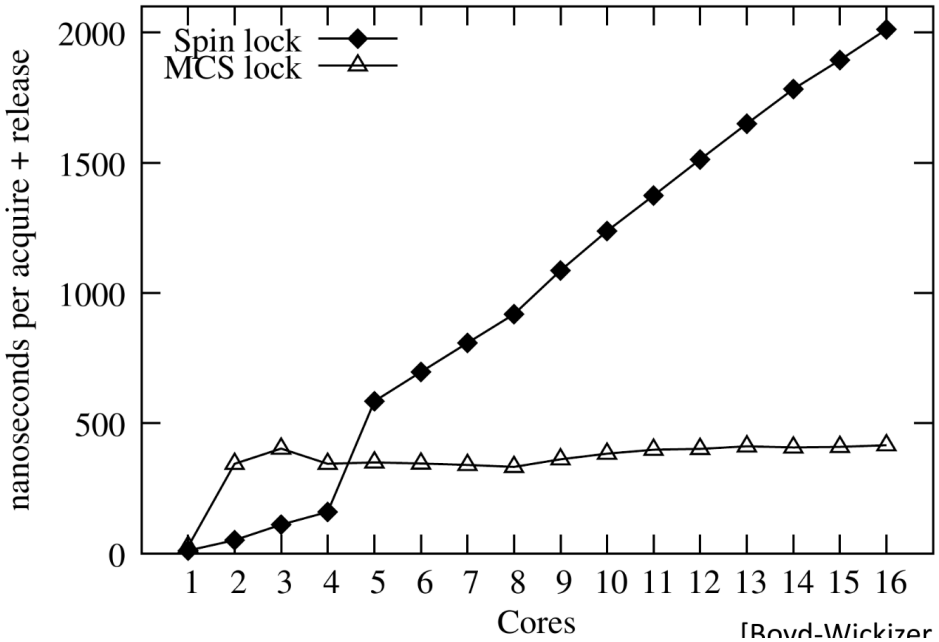
```
void acquire( lock_t *lock, struct qnode *local) {
        local->next = NULL;
        struct qnode *prev = XCHG(lock, local);
        if (prev) {  // queue was non-empty
                local->locked = 1;
                prev->next = local;
                while (local->locked)
                    ;  // spin
        }
}
```

The qnode, on which we spin is thread local.

Release

1. We have the lock. Check is someone is waiting for the lock.
2. If yes, we notify them and they will acquire the lock.
3. If no, set the lock to `NULL` (unless someone appears in the meantime).
4. If someone has appeared in the meantime, wait for them to enqueue and notify them.

Spin Lock Comparison Spin lock to no scale well due to congestion, false sharing etc.



Devices

Devices are hardware which interact with the OS through some kind of software. They are connected via a bus (e.g. PCI) and we communicate with them via a set of shared registers (CPU and device can read/write). Devices can cause interrupts and data can be transfered to/from them using *Direct Memory Access* without work from the CPU.

Device Registers

CPUs can read device registers for:

- Obtaining status information
- Read input data

CPUs can write to device registers for:

- Set device into a certain state and reset the state
- Configure the device in a certain way
- Write output data

Device registers are conceptually different from CPU registers. They can be addressed in two ways. Both are supported by x86.

Memory Mapped When the device registers are memory mapped, they appear to the system as a memory location and hence, common operations like loads and stores can be used in them. This is a great advantage and allows for much more flexibility compared to the second approach.

Device drivers are responsible for correctly initializing the device.

This is the most commonly used approach today.

I/O Instructions There are separated, dedicated instructions to communicate with the device. There is also a separate address space, the I/O address space. This space is typically smaller, e.g. 16 bits.

Summary Both of the introduced methods are used today.

Device registers behave differently from regular RAM. But device registers are actually volatile (change without warning). They are not meant to save data but used for communication. RAM only changes when the CPU changes it but device registers are also manipulated by the device. When writing to a device register we also expect to trigger a certain action. Cs volatile instruction is useful to use in conjunction with device registers to mark to the CPU do not e.g. optimise away such instructions.

Example: National Semiconductor ns16550 UART UART used to be a very common and was used for serial communication. It is connected to the south bridge.

South bridge is for IO while the north bridge is for CPU.

This device is pretty much the standard even though it is not heavily used anymore.

Datasheets about the device are used to figure out how to interact with a device. Often they are rather poorly written and not very accurate.

UART has for example the following registers, each of then is 8 bit long.

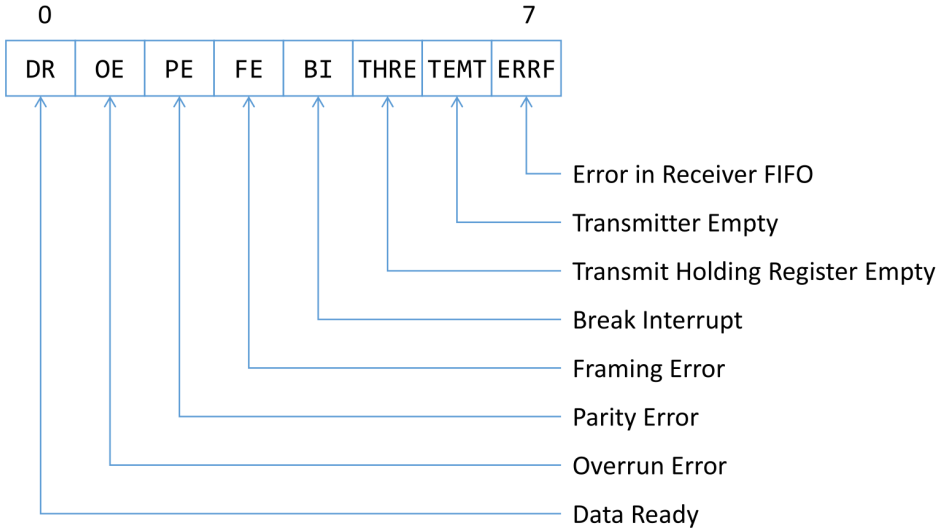
Addr.	Name	Description	Notes
0	RBR	Receive Buffer Register (read only)	DLAB=0
0	THR	Transmit Holding Register (write only)	DLAB=0
1	IER	Interrupt Enable Register	DLAB=0
2	IIR	Interrupt Identification Register (read only)	
2	FCR	FIFO Control Register (write only)	
3	LCR	Line Control Register	
4	MCR	MODEM Control Register	
5	LSR	**Line Status Register**	
6	MSR	MODEM Status Register	
7	SCR	Scratch Register	
0	DLL	Divisor Latch (LSB)	DLAB=1
1	DLM	Divisor Latch (MSB)	DLAB=1

DLAB = bit 7 of the LCR register

The DLAB bit is devicor latch access bit and is part of the LSR. Depending on this bit, the purpose of certain registers differ.

The address starts at 0 but that is just the offset from the memory mapped IO base address.

In the Line Status Registers, different bits have different meaning.



Simple UART Driver The following is a very simplified UART driver, containing no initialisation, no error handling etc.

At first we define the base register of the device register. This information is taken from the datasheet.

For sending a character, we first have to make sure that we can actually write a new char to the device by bitmasking the right register. We spin loop fill we are successful and when this is the case, we write our character to a designated device register. To read, we do again spin loop and check the ready bit if there is data to be read. When there is, we read the data from the register.

```
#define UART_BASE 0x3f8
#define UART_THR  (UART_BASE + 0)
#define UART_RBR  (UART_BASE + 0)
#define UART_LSR  (UART_BASE + 5)

void serial_putc(char c)
{
  // Wait until FIFO can hold more chars
  while( (inb(UART_LSR) & 0x20)== 0);
  // Write character to FIFO
  outb(UART_THR, c);
}

char serial_getc()
{
  // Wait until there is a char to read
  while( (inb(UART_LSR) & 0x01) == 0);
  // Read from the receive FIFO
  return inb(UART_RBR);
}
```

Summary The simple driver uses *Programmed I/O (PIO)* (which is in contrast to *Direct Memory Accessed*). This means that the CPU does explicitly read/write values from registers and hence, all data passes through the CPU.

Also, the simple drives uses polling to wait for a certain condition. An alternative would be to use interrupts, where we are notified, when a certain condition is reached. In polling 100% CPU is used just for waiting. But its advantage is that that it can take action as soon as a condition is reached (there is not need to wait for a notification). Interrupts are superior when there is very infrequent arrival of information.

Drivers are part of the OS. Device drivers are a significant portion of the code (about 70%).

Dealing with Caches

We cannot use caching for device registers since these registers are volatile (the device can change the registers too). This would lead to inconsistent caches.

Write-back caches and write buffers are also problematic because changes are not visible to the device immediately.

Also, read and writes cannot be combined into cache lines (not sure that they meant by that).

As a consequence, device registers must bypass cache. In the page table there was a flag to indicate that a certain page should not be cached. They are used for this purpose.

0.25 Lecture 25: Direct Memory Access

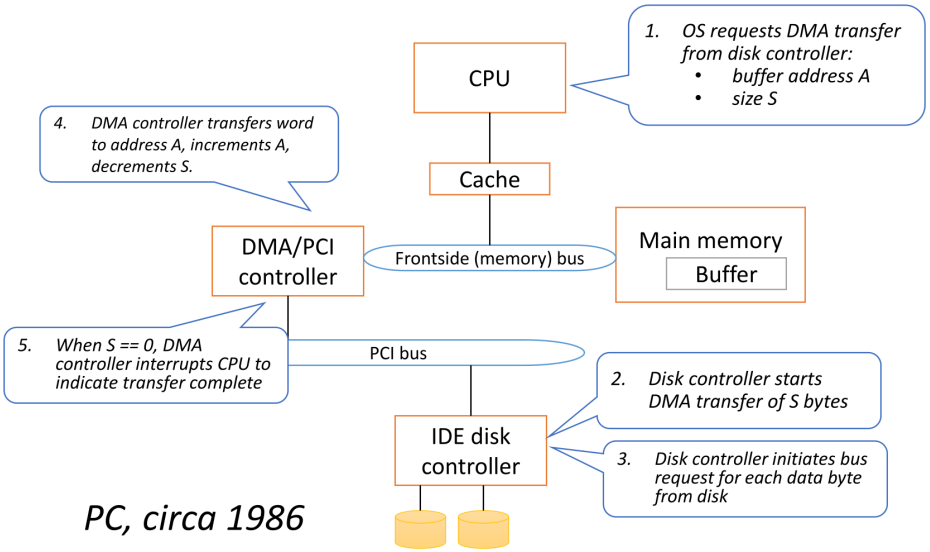
Week: 13

For large data transfers we should not use programmed I/O but direct memory access. This way the data transfer is not routed via the CPU but over a dedicated DMA controller. This way the CPU can

do different things in the meantime. The CPU is still responsible for initiating the transfern but once this is done the DMA controller takes over. Then done, the DMA controller interrupts the CPU.

This is build into the south bridge of processors.

DMA Transfer The DMA controller sits on the PCI bus, to which the disk controller is connected. This represents the schematics of a rather old PC.



In the following the steps for a CPU initiated DMA transfer are depicted. The OS is trying create a file which gets memory mapped from the disk to main memory.

1. The file should get the content of a particular buffer address A of size S. We signal the disk controller that we want to read buffer address A of size S.
2. The disk controller starts to transfer S bytes. Instead of communicating with the CPU it sends the data to the DMA controller.
3. For each byte the disk controller wants to send, it does a bus request.
4. The DNS controller listens on the bus and transfers the data words it receives to address A (in the main memory). The it increments A and decrements S.
5. When `S == 0` the DMA controller interrupts the CPU to indicate that the transfer is complete.

The DMA controller is not a very complex hardware component and can be seen as a finite state machine.

Advantaged/Disadvantages The DNA controller decouples data transfer from processing. So, the CPU does not have to deal with copying data from/to devices. This way the cache is also not polluted. Further, the OS can decide when to schedule disk reads/writes and work in parallel to these disk actions.

Possible disadvantages are that there is a possibility of bus contentions during transfer. For small transfers the setup overhead may dominate. But in general this is not a problem and it is widely used.

DMA and Caches DMA may make memory inconsistent when it change memory blocks which are cached by the CPU. Possible solutions to this issue are:

- We can mark pages as non-cacheable (similarly as we did with regions of memory which correspond to device drivers). This is not a great data since once the transfer is complete we want to be able to cache this data.
- Since de DMA controller sits on the main memory bus it can snoop and interact with the cache coherency protocol. Obviously, this does only work for SMP systems.
- The OS can explicitly flush and invalidate regions of memory. This mean that the OS writes addresses, which are about to get written to disk by the DMA controller back to memory. And it marks regions of memory as invalid when they are going to get overwritten by the DMA.

DMA and Virtual Memory DMA uses physical addresses while the user and OS deals with virtual addresses. Therefore, the OS and device drivers must manually translate between virtual and physical addresses. This is more complicated than it seems at first glance because contiguous virtual addresses may not be contiguous in physical addresses. Therefore, SMA controllers support scatter and gather. This means that the controller is not given a base address and a size but rather a list of addresses. This way we can still access a large portion without it must be contiguous.

Most modern systems have a IOMMU (IO memory management unit) which is, as the regular MMU, responsible for address translation but for IO devices (DMA writes from devices). This IOMMU is programmed by the OS because it must coincide with the address space and the MMU.

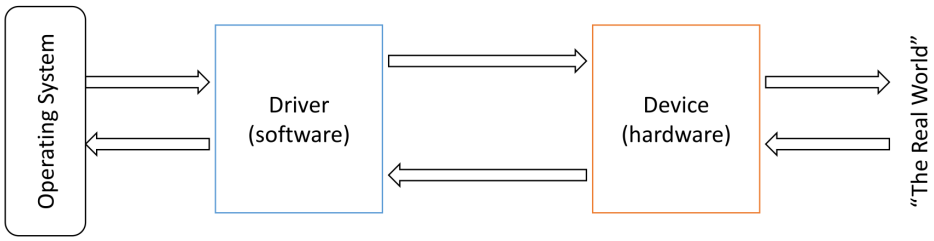
Besides the discussed purpose, IOMMU has many other purposes.

Evolution of IO Devices

1. Drivers relied on programmed IO and polling was used to get device states.
2. Polling is too slow so interrupts are used to notify the CPU for certain actions.
3. CPU spends too much time copying, therefore DMA controllers were used to offload this task from CPUs and allow them to happen in parallel.
4. Too many interrupts slow down the controller and CPU and resynchronisation is required (do not know that this means).
5. Devices become more complex and are almost a processor themselves.

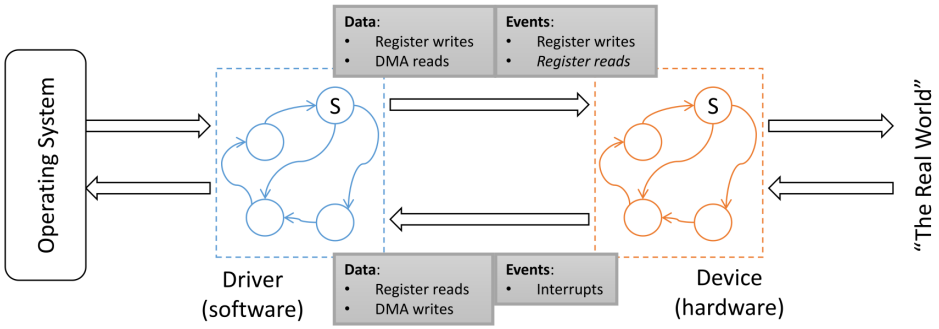
Device Drivers

A basic model of how drivers interact with the OS are depicted below.



Drivers are often considered to be part of the OS.

The driver as well as the device can be considered as a state machine which take action depending on their state as well as the state of the device. Date must be transfered between the two and events are signal state transitions.

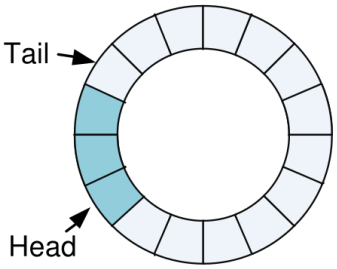


Device - CPU Communication As discussed, there are multiple ways how a device can communicate with the CPU:

1. CPU -> Device; The CPU can write device registers. This communication is synchronous.
2. CPU <-> Device; The CPU can read device registers. This communication is synchronous.
3. Device -> CPU; The device requests an interrupts. This communication is synchronous.
4. Shared memory is used for actually transferring data. It is asynchronous, meaning that not both actors need to be available on the same time.

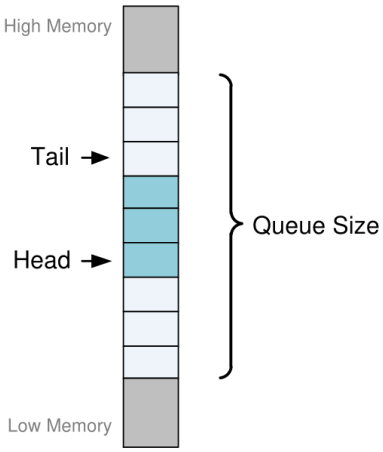
Buffer Rings and Descriptor Rings

Buffer / descriptor rings are visualized as a ring with two pointer to it. The *Head* pointer is the pointer of the device and the *Tail* pointer is controlled by the CPU. Both pointers can move independently as soon as data is added or read.



Logical View

mory



Physical View in Memory

The buffer is of limited size and therefore it is crucial to check if the is space. As a data structure, it is implemented as a queue with two pointers and a fixed size. The entries are contiguous in memory. Often, the values of the queue are pointer to other places in memory to allow to store entries of different size.

Each descriptor is either owned by the CPU or the device. The one who own a descriptor is the one who can read it.

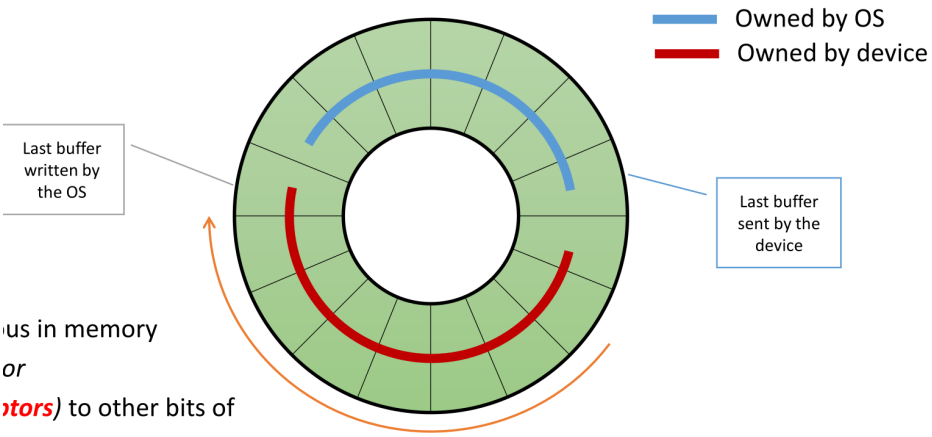
Once everything is initialized by the CPU, there is no need for further communication and synchronisation between the device and the CPU (expect for underruns and overruns). When writing or

reading, the only thing the two agents have to make is that they read/write a entry which belongs to them (i.e. that they are not causing an overrun or underrun).

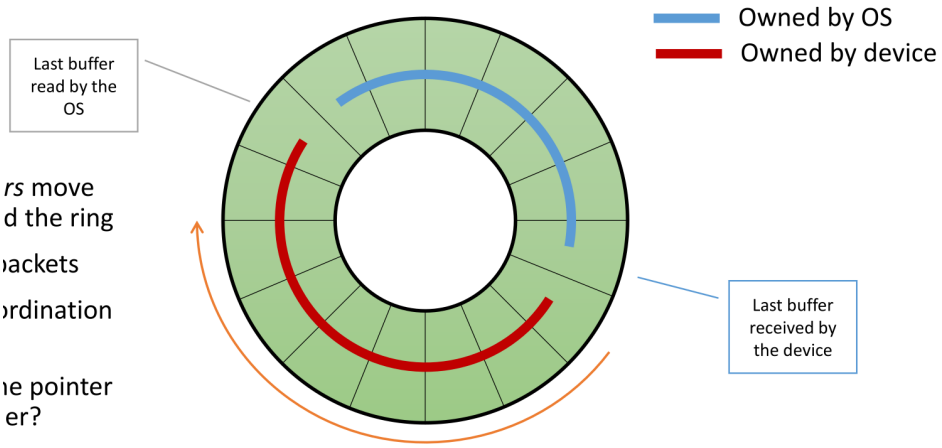
In essence, these buffers are producer-consumers queues which are not based on mutexeses, locks etc. for synchronisation but relay on messages based on reading/writing registers or interrupting each other.

Often, there are two buffer rings in a system, one for reading/receiving (CPU reads from device) and one for writing/transmitting (CPU writes to device).

Transfer The following image shows a buffer ring for transmitting (writing) data.



Receive The following image shows a buffer ring for receiving (reading) data.



Advantages Most modern systems use such buffer descriptors (instead of reading and writing directly to memory). This pointers provide a level of indirection. In addition to the raw data, also some metadata like error messages are transmitted. So we have to ability to write them to a separate buffer. This way data and metadata is not mixed. Further advantages are that we are much more flexible where data is stored since as we said, the entries of the ring can be pointers them selves. Buffers can be of different size and they can even resize dynamically.

Overruns and Underruns while Receiving While the CPU is receiving data, if the buffer gets full and it does not have space for received packages the device starts to discard new packages which are coming in (it does not overwrite old packages). There may be some other buffers which kind of

act like a backup buffer once the "main" buffer is full. In general this is not that bad when packages are lost since there are protocol that detect that and can request resending of this package. The CPU must signal the device must wait (tell the device our status).

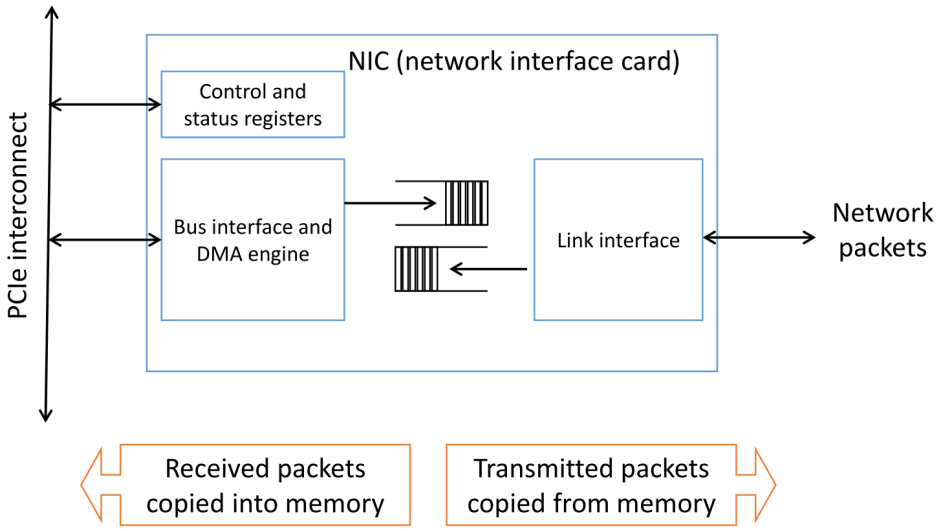
When the CPU has read all data of the buffer, it must wait. One solution to deal with that is to spin poll, i.e. spin loop and check if there is new data to process. But this keeps to CPU busy and it cannot do anything else in the meantime. Another solution is to signal to the device that is should interrupt the CPU once there is new data is the buffer. This way the CPU can deal with other things in the meantime.

Overruns and Underruns while Transmitting When the CPU is writing to the buffer and the buffer is full, it has to wait for the device to make space on the buffer for further writes. Again, it can spin poll or tell the device that it should interrupt it once there is again space in the buffer.

If there are no more packages for the device to receive it must wait. The device could continue to poll the buffer till there is new data to read or it could signal the CPU to wake it up where there is more data in the buffer and go to sleep.

More Complex Devices

Network Adapter/Network Interface Card (NIC) NICs are often connected to the PCIe bus. It has a integrated DMA engine and has a link interface where it received network packages.



The NIC transmits data from memory to the network and moves packages received from the network to memory.

Example: DEC "Tulip" Fast Ethernet Adaptor This device is very old but very well documented. It illustrated all basic principles of more complex devices.

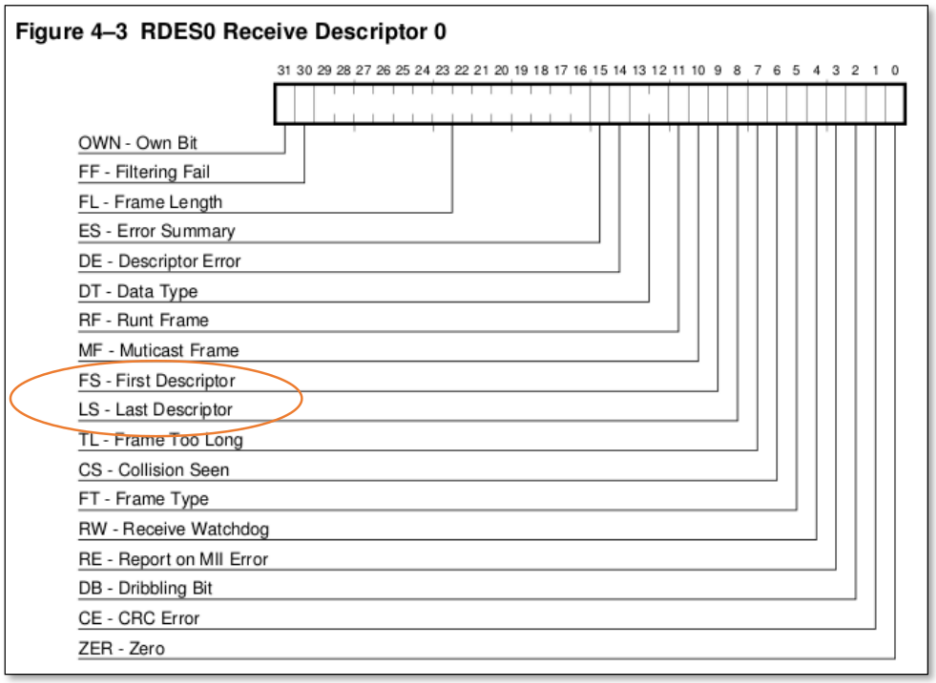
As with the UART, we have again different registers to communicate with the device. For example receive/transmit poll demand are flags which indicate that there is data to read/write. The Receive/-Transmit list base addresses are the base address of the buffers. The interrupt enable is a flag to enable/disable interrupts.

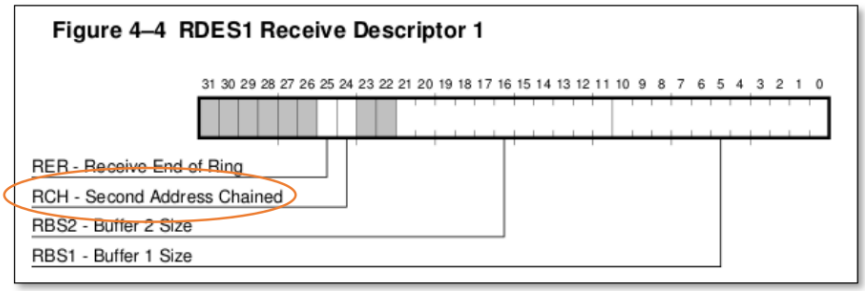
CSR13 and CSR14 are not listed below. That is because according to the data sheet they are not meant to be accessed by the CPU and changing them can have unexpected results.

Register	Description	Address offset
CSR0	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

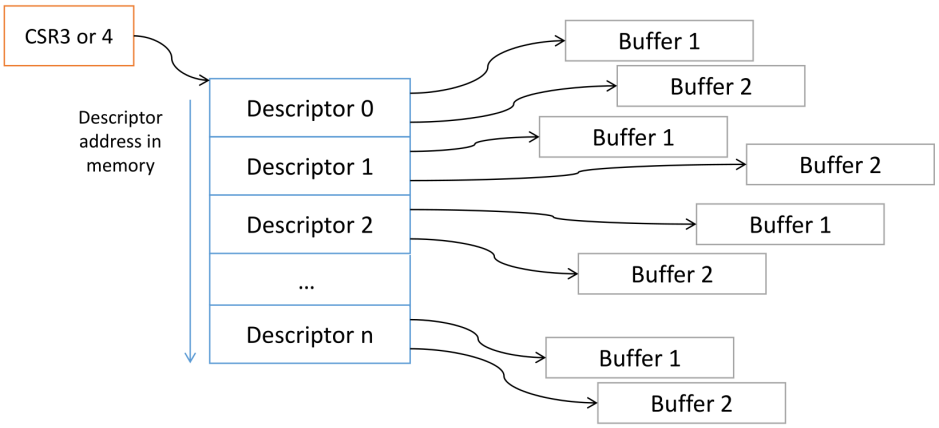
The descriptors of this device look as follows. They have space for two buffer addresses. There are status bits and own bits which indicate if the buffer are device or hardware owned (do not know what this means). The two size count bits tell the size of the two buffers. The control bits are used

The second address buffer can either act as a buffer or a liked list. But in general the two are used for transmitting and one for receiving. This is controlled by the RSH bit. The FS and LS flags help to figure out the size of the queue.

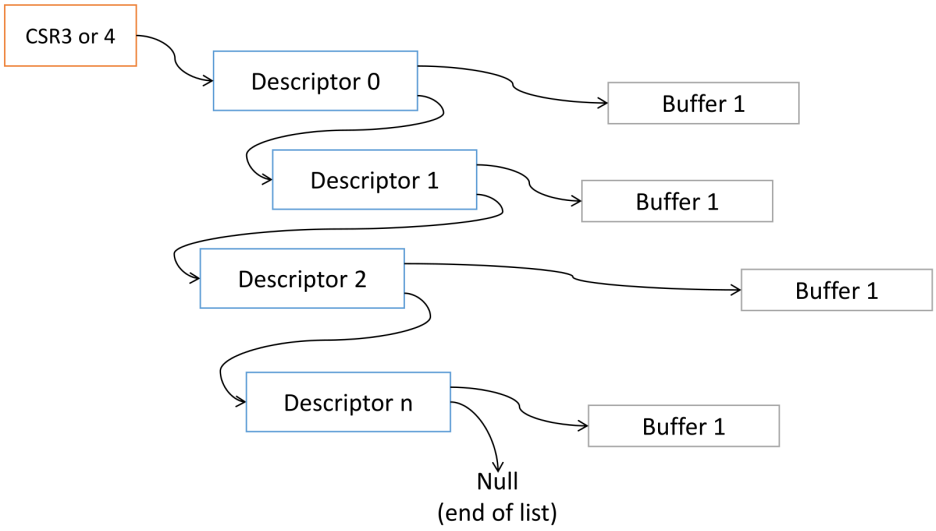




Descriptor Ring - Normal Mode The base address points to the first descriptor. Then we have the whole chain of descriptors, where each holds a pointer to some other region in memory.



Descriptor Ring - Chain Mode In this mode the base address points to the first descriptor. This node then hold a pointer to some data as well as a pointer to the next descriptor. In this case the descriptors are not contiguous in memory.



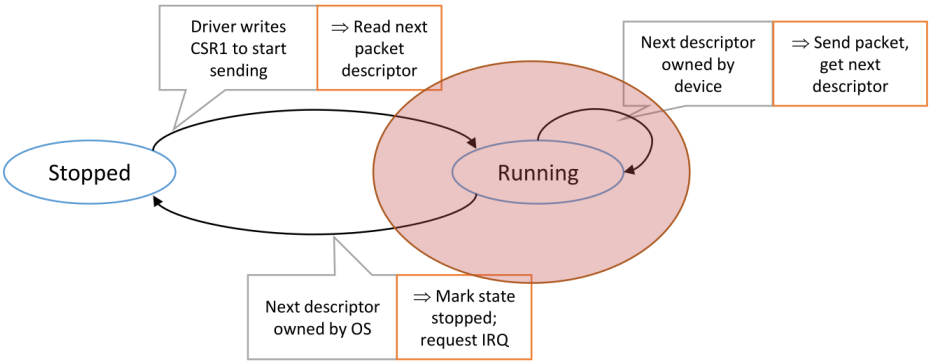
Device Initialisation

Before a device can be used, the driver must initialize it. We said that the driver as well as the device are some kind of statemachine. When plugged in, we do not know what state a device is in. So we have to put it into a known state. In general, this requires the following steps:

1. Wait for the hardware to settle down.
2. Stop the device from doing anything (sending interrupts, DMA, sending packages etc.).
3. Create the shared data structure and tell the device where it is by writing the address into some register.
4. Start the device by writing some registers.

IO State Machines; Transfer (hardware side) We are looking at the statemachine of the device and assume that we send packages to the device.

The device is either in Stopped or Running state. To start the device, the driver must write a certain register. In response, the device looks at the buffer and DMA reads the next descriptor. If the next descriptor owned bit indicates that the descriptor is owned by the device, it DMA reads the buffer and sends the package. Then the device DMA writes the owned flag to make it belong to the OS again. Afterwards, it calculates the address of the next descroptor (depends if we use a buffer or a linked list). The device repeats this procedure till the next descriptor is owned by the OS. In that event, it goes to the stopped state and notifies the CPU that it waits for an interrupt.

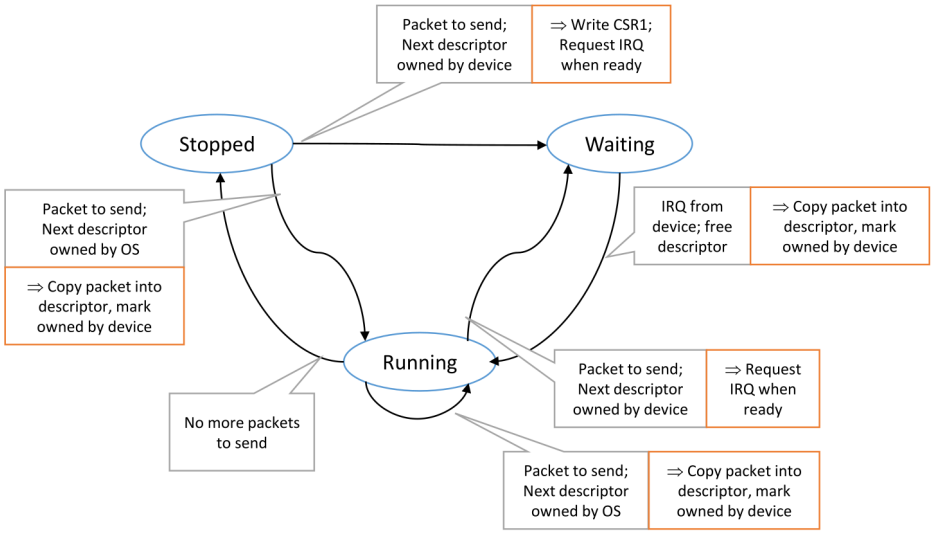


IO State Machines; Transfer (software side) We are looking at the statemachine of the driver and assume that we send packages to the device.

The driver has three different states; stopped, waiting and running. We are in the stopped state and we have a package to send. So we read the descriptor and check the owned bit in order to make sure that there is space for us to write. We copy the data to the buffer and mark it as owned by the device. This brings us to the running state.

Back in the stopped state, if we want to send a package and hence, read the owned bit of the next descriptor in order to make sure it is ours. Now it case that it is not our (i.e. the buffer is full), we get into the waiting state and indicate to the device that we want to get interrupted by setting a register. When the device interrupts us, we copy the data into the descriptor and mark it as owned by the device.

If we are in running state, read a next descriptor and it is owned by us, we copy the data into the descriptor and go to the next descriptor. We stay in the running state as long as we can read next descriptors. From the running state, we go into waiting if we want to send data but the buffer is full, or we go into stopped once we do not want to send any more data.

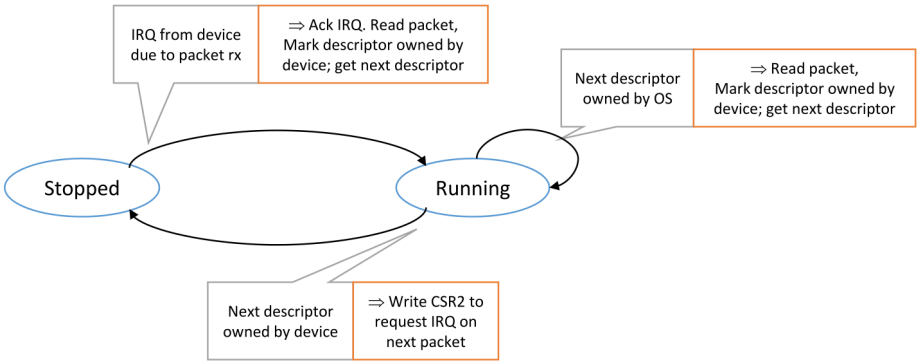


Avoid Cache Problems On x86 hardware, PCI-based DMA listens to the memory bus and ensures this way that the cache are consistent. On all other hardware we have to:

- On DMA read a certain address. Before the DMA reads, we have to make sure that the lastest changes of that address are in memory (DMA reads memory). We do this by flushing (write dirty data back to memory) the addresses. After the DMA read, the CPU invalidates (discard all lines from the cache) the cache for the address. This is done to prevent using outdated descriptors (on a DMA read, the device also writes to memory).
- On DMA write a certain address. Before the DMA write, we flush or invalidate the whole (I guess) cache to make sure that memory has the latest changes. After the write we invalidate to cached whole cache in order to fetch the new data from memory.

IO State Machines; Receive (software side) We have two states, stopped and running. When in stopped, we get an interrupt from the device that new packages have arrived. We acknowledge the interrupt and read the package. We change the owned bit to indicate that is owned by the device. This brings us into the running state. When in this state, we keep reading descriptors, as long as they are owned by us. If the next descriptor is owned by the device, we write to a certain register to

indicate that we are stopped are request a interrupt when there is new work. This brings us to the stopped state.



Discoverable Buses: PCI

How does the OS know what devices are plugged in, how does it keep track of the addresses of each device (where are the registers of these devices)... There are many open question regarding the connection of devices.

We look at PCI as a particular solution to this problem.

PCI stands for *Peripheral Component Interconnect*. It is a standard for connecting devices to a computer and also standards the physical connectors (the ones sitting on a motherboard). It also defines the bus protocol and how messages are interpreted by the CPU und devices. It is kind of the software-visible interface to IO hardware.

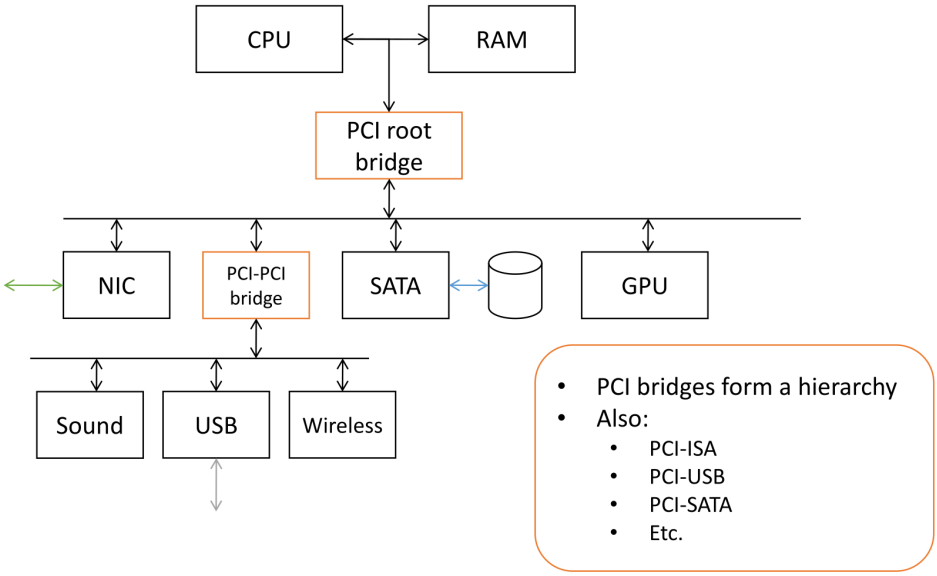
PCIe (PCI-Express) is the successor of PCI, but we still consider PCI in this course since the general principle is the same.

PCIe is a high speed serial connector ($O(1GB/s)$) while PCI is a parallel connector ($O(100MB/s)$).

PCI Solves PCI allow to discover devices and keep track of addresses. It also helps to map interrupts to exception vectors. It also provides a DMA. This allows devices to initiate DNA requests (this used to be initiated by the CPU).

Physical Connections PCI can be seen as a tree.

The PCI root bridge (PCI root complex) connects the PCI to the main memory bus. Through other bridges, PCI is connected to other things like USB, SATA, NIC etc.



Address Space The PCI address space is flat. This means that it is one big address space for all devices. Each device asks for a set of addresses in this range. Physical address space is usually 32 or 64-bits and the I/O address space is usually 16-bit.

The PCI bridges remap addresses to the device. This is required since the PCI addresses may not be contiguous while the device required contiguous addresses

PCI Devices PCI devices are self describing. This means that they have all information required to identify them. This included vendor, model, version etc.

Finding All The Devices The first thing we do is finding the root bridge. In large systems there may be multiple such root bridges. Then we read the configuration of each device. While iterating the we add them to a list and record the size of the required address space. If we find a bridge, we do recursively discover it.

This results in a list of all devices and there address space requirements.

Allocating Addresses We find a suitable address range for each device following the requirements:

- Required address space size of device must match
- All devices of a bridge must fit into the address space range of the bridge of that bus.
- The range is aligned as a power of 2

Then we must program the address translation information. For each device we must save the base address of that device. This is stored in the *base-address/range (BAR)* register.

PCI Interrupts There are four interrupt lines (INTA, INTB, INTC, INTD). When a PCI bus detects a interrupts, it forwards it to the root bridge which then interrupts the CPU.

PCIe introduces *message-signaled interrupts (MSIs)*. Instead of settings a physical pin of the CPU, we wan set a certain memory mapped address. This will signal the right interrupt to the CPU. So interrupts are sent to specific addresses and depending on the address a different interrupt is sent. This reduces the need of pins on the CPU.