

Computer Architecture and Assembly Language

Joël Porquet

EPITA

2014



License

- ▶ Copyright © 2004-2005, ACU, Benoit Perrot
- ▶ Copyright © 2004-2008, Alexandre Becoulet
- ▶ Copyright © 2009-2013, Nicolas Pouillon
- ▶ Copyright © 2014, Joël Porquet

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just ‘‘Copying this document’’, no Front-Cover Texts, and no Back-Cover Texts.

<https://asm.ssji.net/>

Part I

Introduction

What are we trying to learn?

Computer Architecture

What is in the hardware?

- ▶ A bit of history of computers, current machines
- ▶ Concepts and conventions: processing, memory, communication, optimization

How does a machine run code?

- ▶ Program execution model
- ▶ Memory mapping, OS support

What are we trying to learn?

Assembly Language

How to “talk” with the machine directly?

- ▶ Mechanisms involved
- ▶ Assembly language structure and usage
- ▶ Low-level assembly language features
- ▶ C inline assembly

Course outline

- ▶ Processor architecture
- ▶ Memory
- ▶ Memory mapping
- ▶ Execution flow
- ▶ Object file formats
- ▶ Assembly programming
- ▶ Focus on x86
- ▶ Focus on RISC processors
- ▶ CPU-aware optimizations
- ▶ Multi-/Many-core, heterogeneous systems

Part II

Processor architecture

What a processor is...

A processor must be able to perform the following basic tasks:

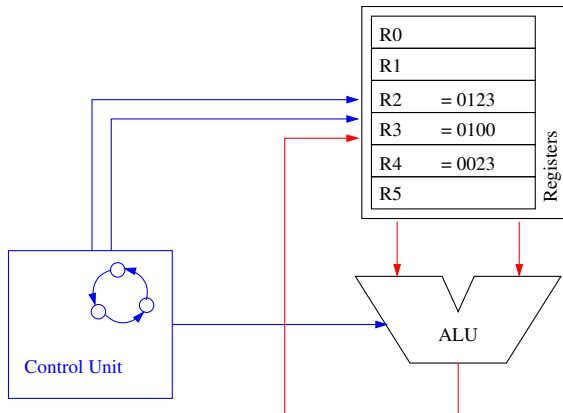
- ▶ Execute instructions
- ▶ Read operands
- ▶ Store results

It needs several basic units to perform those tasks:

- ▶ A control unit
- ▶ An arithmetic and logical unit (*ALU*)
- ▶ A register bank

Let's design it!

Basic architecture



Basic architecture (2)

In this model, the system state is entirely contained in the processor.

- ▶ This might be sufficient for a very basic processor
- ▶ But further features can only be implemented by adding registers or program steps

Unfortunately,

- ▶ Internal memory is expensive and hard to design
- ▶ There is no communication
- ▶ Updating the program may not be easy

We need an access to memory, external devices, etc.

Revised processor model

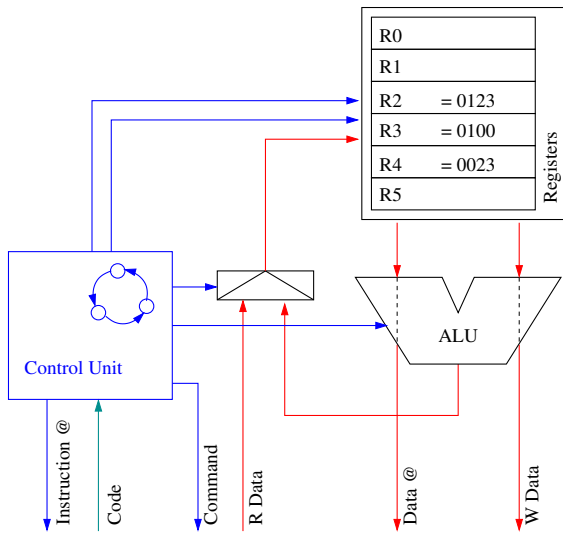
A processor must be able to perform the following basic tasks:

- ▶ **Fetch instructions** from an external entity and understand them (*fetch* and *decode*)
- ▶ Execute instructions
- ▶ Store results **to registers or external memory**

It needs several basic units to perform those tasks:

- ▶ A control unit
- ▶ An arithmetic and logical unit (*ALU*)
- ▶ A register bank
- ▶ **A program memory access**
- ▶ **A data memory access**

Revised processor model (2)



Processor physical layout

A processor is composed of many different units:

- ▶ Caches, MMU
- ▶ Integer unit, Control unit, Floating-point unit

Each unit is:

- ▶ implemented as an hardware component
- ▶ made of switchable parts (transistors)

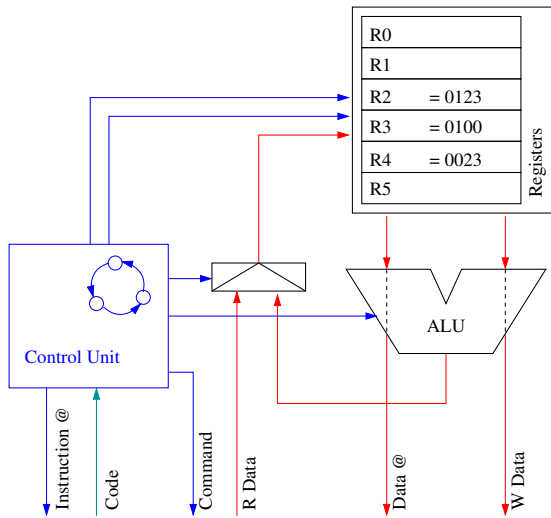
In old processors:

- ▶ Units used to be independent chips
- ▶ Some were even optional “coprocessors”

Today, processors are embedded on a single chip.

Units

- ▶ Control unit fetches and decodes the instruction
- ▶ Registers gives the data
- ▶ ALU implements the operation
- ▶ Some instructions access external data



Registers

May be seen as variables located inside the processor

- ▶ 8, 16, 32, 64, 128, ... -bits large
- ▶ General-purpose registers:
 - ▶ integer (`int`)
 - ▶ floating point (`float`, `double`)
- ▶ Specialized registers:
 - ▶ flags
 - ▶ Zero,
 - ▶ Negative,
 - ▶ Carry,
 - ▶ Overflow,
 - ▶ etc.
 - ▶ system
 - ▶ Mode,
 - ▶ IRQ masking,
 - ▶ etc.

ALU: Arithmetic and Logical Unit

An unit without registers!

- ▶ Logical operations
 - ▶ AND, OR, XOR, NOT, NOR
- ▶ Arithmetic operations
 - ▶ addition, subtraction, (multiplication,) ~~division~~
- ▶ Shifts
- ▶ Compares

Multiplication and division are usually not possible without registers!

Instruction types

There are different instruction types:

- ▶ Arithmetic and logical operations
- ▶ Control instructions
- ▶ Memory access instructions

Instructions classification

Flynn's taxonomy:

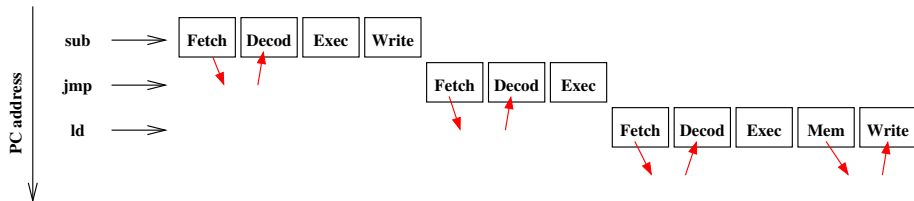
- ▶ SISD: Single Instruction, Single Data
Classical Von Neumann architecture
- ▶ SIMD: Single Instruction, Multiple Data
Vectorial computers
- ▶ MIMD: Multiple Instruction, Multiple Data
Multiprocessor computers

Microprogrammed processor

Instruction flow

Processors may use a *finite state machine* or *micro ops* to process each instruction step (*Fetch*, *Decode*, *Execute*, *Memory access*, *Register write back*, etc.)

A basic processor needs several clock cycles to process all steps of the current instruction before starting the next one.



Microprogrammed processor

Pro/cons

Cons:

- ▶ Slow
- ▶ The more complex the instructions are, the longer they take to get processed
- ▶ Most of the hardware is used only once for each instruction
- ▶ Most of the hardware is unused most of the time

Pros:

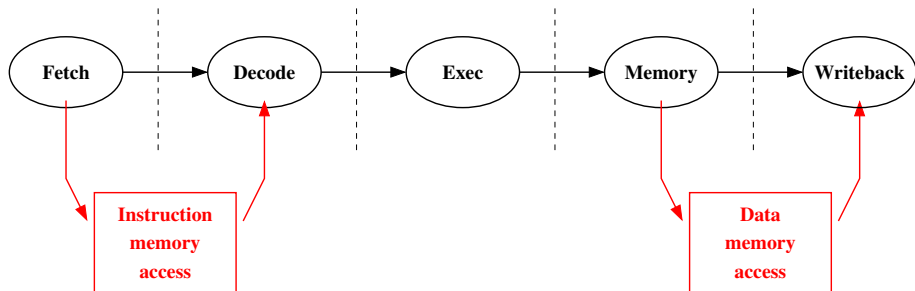
- ▶ Easy to implement
- ▶ Small
- ▶ New instructions can be added just by adding new steps

Pipelined processor

Instruction flow

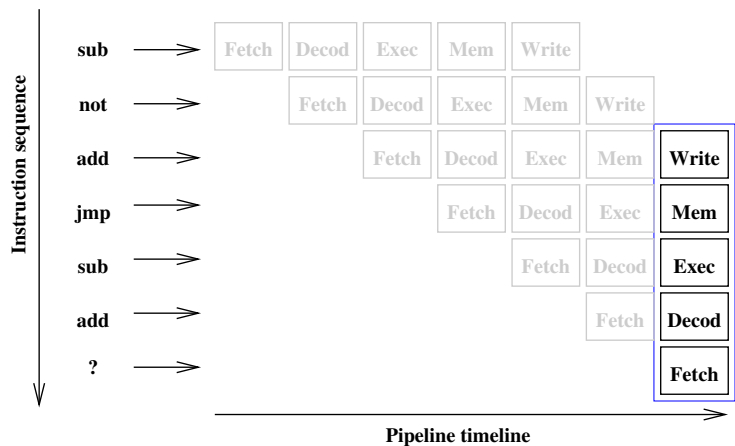
A pipeline architecture enables the parallel execution of several instructions:

- ▶ Split the execution of each instruction in several steps
- ▶ Each step performs an elementary operation
- ▶ Each step is associated to a specific part of the hardware
- ▶ All parts of the hardware work in parallel



Pipeline

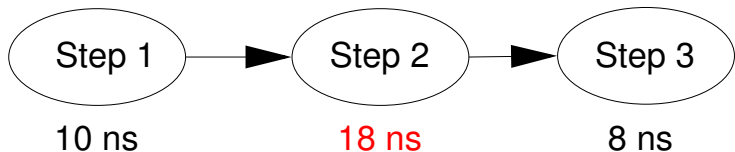
Flow



Speeding up the pipeline

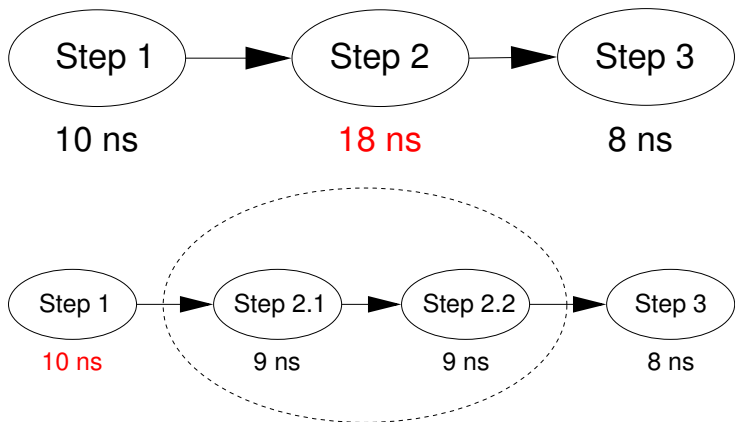
Current processors extensively use the pipeline architecture to accelerate the execution of instructions.

Because a pipeline architecture works in parallel, the slowest step delay determines the pipeline global cycle delay (and working frequency).



Speeding up the pipeline

Splitting operations in shorter steps enables the processor frequency to increase.



Instruction-set classification

Based on internal architecture and instructions formats, processor architectures may be classified in two groups:

- ▶ Complex Instruction Set Computer (*CISC*)
- ▶ Reduced Instruction Set Computer (*RISC*)

Early processor architectures were mostly *CISC*-based: *z80*, *Intel x86*, *Motorola 68000*, etc.

More recent designs are rather *RISC*-based: *MIPS*, *Sparc*, *Alpha*, *PowerPC*, *ARM*, etc.

RISC

Characteristics

Pros:

- ▶ Simple instructions
- ▶ Fixed-length instructions
- ▶ Decoding instructions requires simple hardware

Cons:

- ▶ Programs are longer as they need more instructions
- ▶ Optimization is harder, compilers need to be smarter

Sometimes said as “Reject Important Stuff into Compiler”

RISC

Instruction example

sub %g1, %g2, %g3



0x86	0x20	0x40	0x02
-------------	-------------	-------------	-------------



format	destination	opcode	source	unused	source
10	00011	000100	00001	000000000	00010



%g3



sub



%g1



%g2

CISC

Characteristics

Pros:

- ▶ Lots of instructions and opcodes
- ▶ A single instruction can perform complex operations
- ▶ Assembly programs are easier and shorter to write
- ▶ Code compression ratio is good

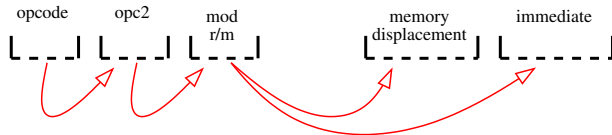
Cons:

- ▶ Binary instruction format is of variable length
- ▶ It requires more complex hardware, and high frequencies are harder to achieve

Modern processors often internally translate the CISC code to RISC microcode

CISC

Instruction example



Part III

Memory

Reasons to access memory

How does the processor interact with memory?

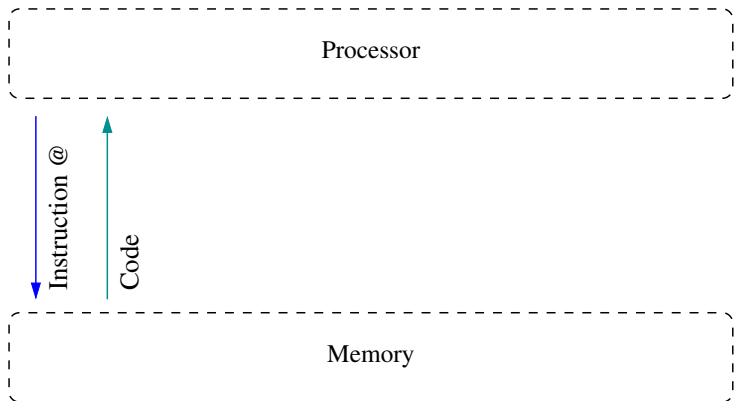
- ▶ Memory is used to fetch instructions,
- ▶ Memory is used to access data.
- ▶ There may be one unique memory, or two:
 - ▶ If there is one memory, instruction / data accesses must be sequentialized
 - ▶ If there are two, code cannot be accessed as data

Traditional memory architectures:

- 1 Von Neumann
- 2 Harvard

Instruction fetch

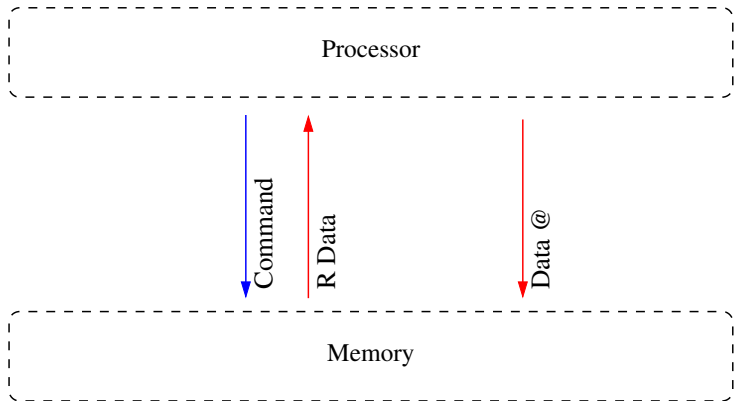
The processor needs an instruction to process:



Data access

Data load

Load the content of the memory cells pointed to by %g1 into %g2 register:

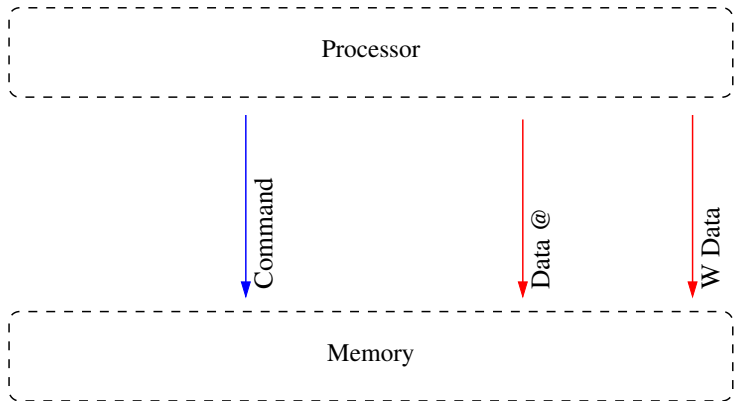


```
ld [%g1], %g2
```

Data access

Data store

Stores the content of %g2 register into the memory cells pointed to by %g1:



```
st %g2, [%g1]
```

Immediate addressing

- ▶ The *value* of data is directly stored in *the instruction*
- ▶ No memory access needed to get the value

In C language:

```
int a, b = ...;
```

```
a = b + 0x831;
```

In assembly language:

```
add %g1, 0x831, %g2
```

Immediate addressing

Sparc instruction details

sub %g1, 0x831, %g2



0x84	0x20	0x68	0x31
-------------	-------------	-------------	-------------



format destination

opcode

source

immediate

10	00010	000100	00001	1	0 1000 0011 0001
-----------	--------------	---------------	--------------	----------	-------------------------



%g2



sub



%g1



0x831

Absolute addressing

- ▶ The *address* of the data is stored in *the instruction*
- ▶ A memory access is needed to get the value

In C language:

```
int a = *(int*)0x830;
```

In assembly language:

```
ld [0x830], %g1
```

Register indirect addressing

- ▶ The *address* of the data is stored in *a register*
- ▶ A memory access is needed to get the value

In C language:

```
int a, *b = ...;
```

```
a = *b;
```

In assembly language:

```
ld [%g2], %g1
```

Complex addressing

- ▶ Register indirect with base register
- ▶ Register indirect with offset
- ▶ And many others...

Assembly example:

```
ld [%g2 + 0x124], %g1  
ld [%g2 + %g3], %g1
```

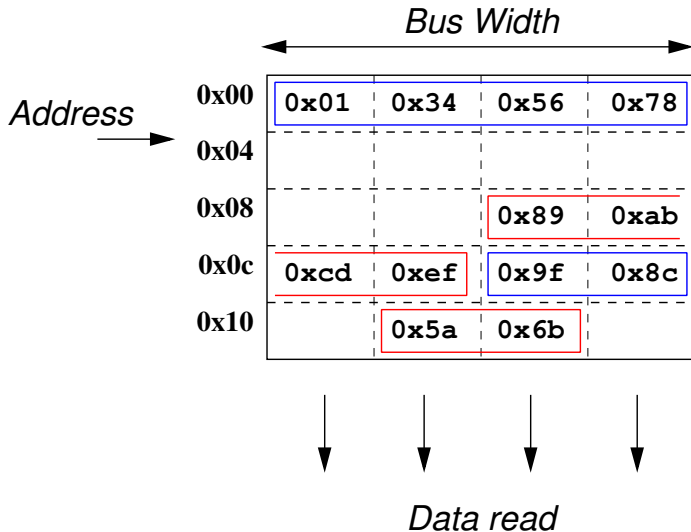
Definition

Data access alignment only depends on the targeted **data type width**.

- ▶ A 32-bit memory access is aligned for addresses multiple of 4
- ▶ A 16-bit memory access is aligned for even addresses
- ▶ A 8-bit (char) memory access is always aligned!

Think about `address % sizeof(type)!`

Access alignment



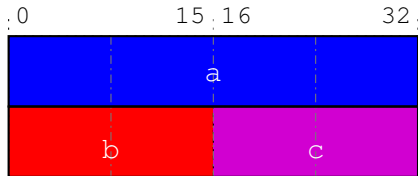
Structure alignment

In a structure:

- ▶ Fields must be in declaration order
- ▶ Fields must all be aligned

Data alignment does not depend on architecture bus width

```
struct bit_packed_s  
{  
    int    a;  
    short  b;  
    short  c;  
};
```

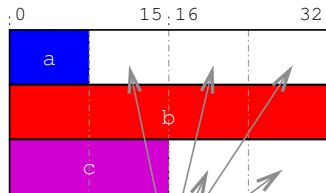


Padding

Sometimes, consecutive fields in declaration cannot be consecutive and aligned in memory

- ▶ Compilers then structure fields on aligned boundaries
- ▶ Alignment may add unused **padding** bytes between fields

```
struct example_aligned_s  
{  
    char  a;  
    int   b;  
    short c;  
};
```

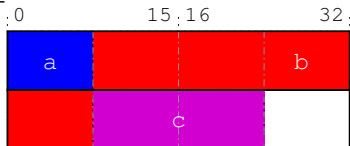


padding bytes

Basic packing

- ▶ Fields alignment can be ignored by compiler, **on request**
- ▶ Few architectures are able to access unaligned fields directly
- ▶ If non-native, unaligned accesses are emulated with multiple memory accesses, shifts, ORs, etc.

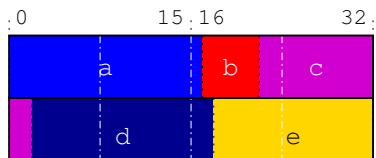
```
struct example_packed_s
{
    char  a;
    int   b;
    short c;
}
__attribute__((packed));
```



Low-level packing

- ▶ Packing can even be done at bit level!
- ▶ Compiler will handle shifts and masks
- ▶ Can be mixed with union
- ▶ Powerful for matching existing protocols

```
struct bit_packed_s
{
    int    a:17;
    int    b:5;
    int    c:12;
    int    d:16;
    int    e:14;
}
__attribute__((packed));
```



Beware of endianness!

Endianness

A data string represented with multiple bytes must be stored in memory. Similarly to written language, these bytes may be written “left-to-right” or “right-to-left”.

- ▶ Big-Endian
- ▶ Little-Endian
- ▶ Other endian modes

Endianness

Mathematical reference

With a base b , a natural N may be decomposed in digits d_k . If we naturally write it:

- ▶ $N = d_n d_{n-1} \dots d_k \dots d_1 d_0$
- ▶ $N = d_n \cdot b^n + d_{n-1} \cdot b^{n-1} + \dots + d_k \cdot b^k + \dots + d_1 \cdot b^1 + d_0 \cdot b^0$

$$N = 48103_{10} = BBE7_{16}$$

- ▶ With $b = 10$: $N = 4 \cdot 10^4 + 8 \cdot 10^3 + 1 \cdot 10^2 + 0 \cdot 10^1 + 3 \cdot 10^0$
- ▶ With $b = 16$: $N = 11 \cdot 16^3 + 11 \cdot 16^2 + 14 \cdot 16^1 + 7 \cdot 16^0$

Logically we tend to count digits from LSB: right to left

Digit no	4	3	2	1	0
Base 10 value	4	8	1	0	3
Base 16 value		b	b	e	7

Memory representation

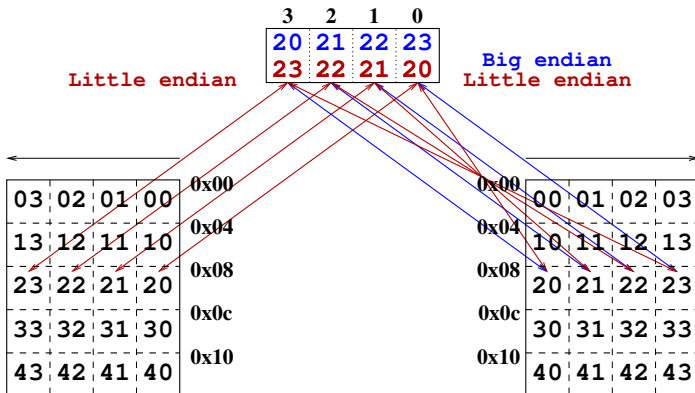
Usually, we like to represent memory in written order, the same way we write words on a paper sheet: left to right

	0	1	2	3
0x00	'A'	' '	's'	'i'
0x04	'm'	'p'	'l'	'e'
0x08	' '	'm'	'e'	's'
0x0c	's'	'a'	'g'	'e'

Integer memory representation

The “endianness” problem is whether to write integers:

- ▶ in text order: the natural way (for western languages)
- ▶ in index order: with digit 0 at address 0



Endianness

Do not mix everything!

- ▶ Data must be stored and fetched using the same convention.
- ▶ Don't worry about byte order in registers, it does not make sense.

Endianness Demo

```
int main(void)
{
    unsigned int a = 0x12345678;
    hexdump(&a);
}
```

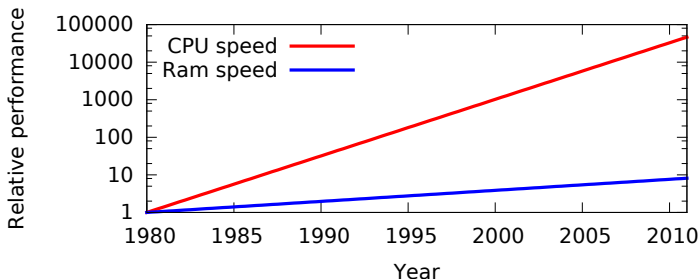
On a typical x86 architecture?

LE 00000000: 78 56 34 12

BE 00000000: 12 34 56 78

Why do we need them?

- ▶ Once upon a time, CPU and memory speeds were the same.
- ▶ Of course, they evolved:
 - ▶ Moore's law: CPU performance roughly doubles every two years,
 - ▶ Memory speed: +7% every year (i.e. doubles every ten years).



Conclusion: need to reduce the average time to access data from the memory!

Definition

A CPU cache is:

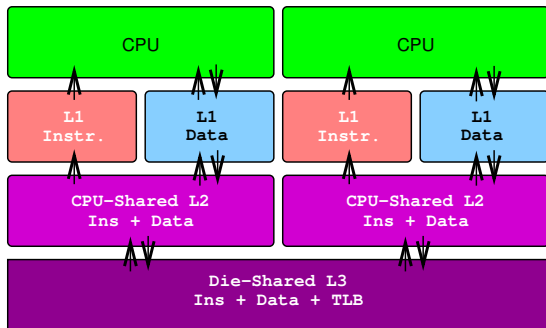
- ▶ a local copy of central memory,
- ▶ transparent and on-demand,
- ▶ volatile (may be flushed anytime),
- ▶ faster, closer to CPU than central memory,
- ▶ expensive!

Cache hierarchy

There are multiple caches “levels”:

- ▶ with different size,
- ▶ with different speed,
- ▶ with different latency.

Caches may be shared between code and data.



Side-effects of CPU caches

Caching memory introduces new issues, especially in multiprocessor systems:

- ▶ Coherence,
- ▶ Consistency.

When programming, beware of the memory operations, at different levels:

Programmer what you handle in C code,

Assembler what the compiler generated,

CPU what the CPU does,

In-cache what the cache does in the system.

Part IV

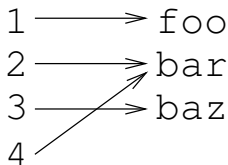
Memory mapping

Address space

Definition

An address space is a set of discrete values unambiguously identifying a set of objects.

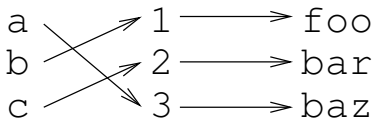
- ▶ Each address points to one object only
- ▶ One given object may be pointed by more than one address



Address space

Translation

An address space translation creates a new address space where each **source** address is mapped to a **destination** address.



Memory address space

Definition

A memory address space:

- ▶ is contiguous
- ▶ can be mapped to another **destination** memory address space (through address space translation)

All memory accesses are done with respect to an address space.

Memory address space

Usual hierarchy

- Virtual** Address space reachable by a program. Generally on 32 or 64 bits.
- CPU** Address space available through a CPU register. Current machines have 32- or 64-bit pointer registers.
- Physical** Address space actually wired between hardware components. Current machines have physical address spaces around 40 to 48 bits.

Physical memory

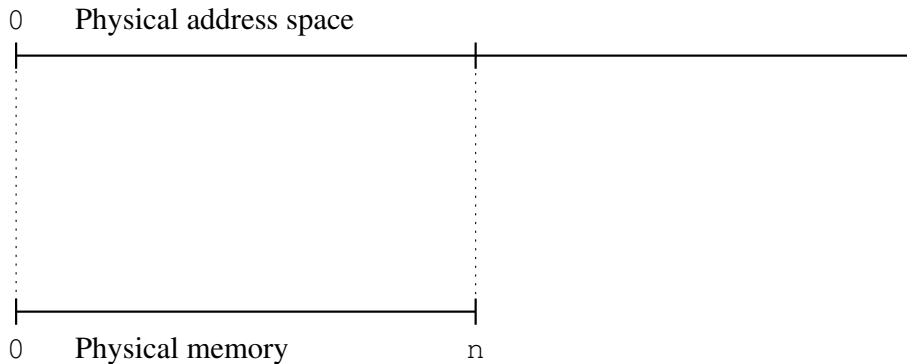
The physical memory address space provides the lowest accessible address space in computer.

The physical RAM banks are usually accessible as a small memory subset of the physical address space.

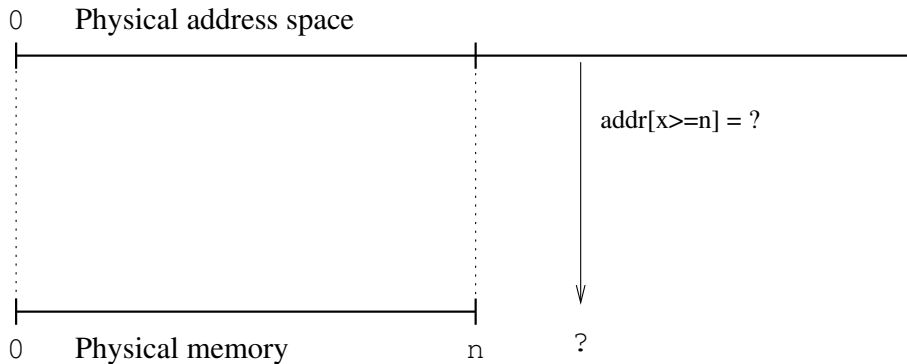
The physical memory can be mapped in different ways depending on the physical address bus implementation:

- ▶ Accessible at a given location,
- ▶ Scattered at multiple locations,
- ▶ Accessible at multiple locations.

Physical address space



Single mapped RAM



Multiple/loop mapped RAM

Interesting kernel trick!

```
static void *detect_magic __initdata = detect_memory_region;

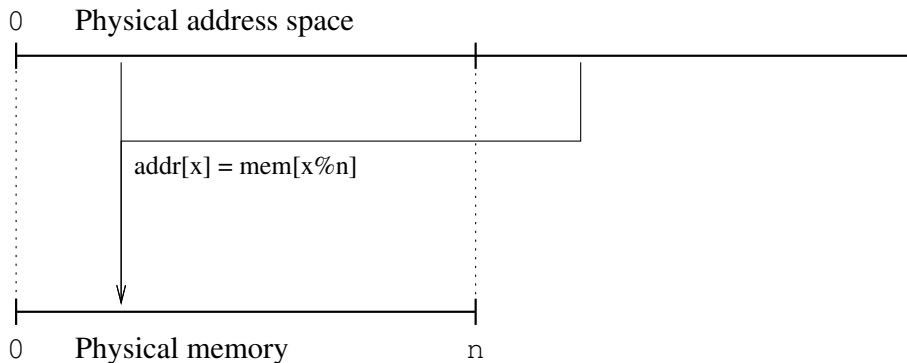
void __init detect_memory_region(phys_t start, phys_t sz_min,
    phys_t sz_max)
{
    void *dm = &detect_magic;
    phys_t size;

    for (size = sz_min; size < sz_max; size <= 1) {
        if (!memcmp(dm, dm + size, sizeof(detect_magic)))
            break;
    }

    add_memory_region(start, size, BOOT_MEM_RAM);
}
```


Multiple/loop mapped RAM

Why does the previous trick work:



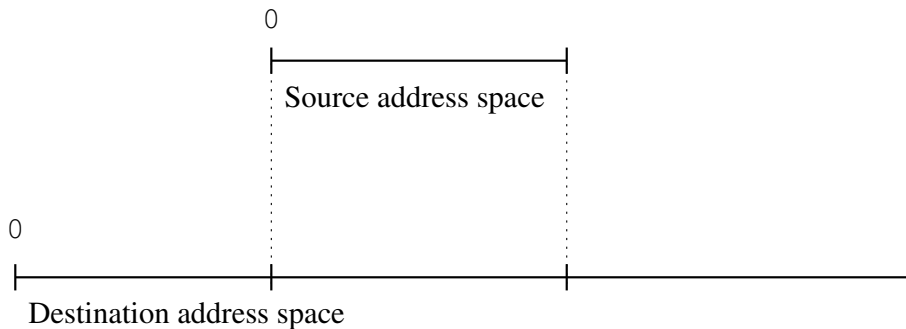
Segmentation

Segmentation defines an address space as a set of sub-regions of another address space. Each sub-region, or segment, is mostly characterized by the following attributes:

- ▶ A base address (in the destination address space),
- ▶ A size,
- ▶ A type and some access rights.

Simple segment

Segmentation keeps memory locations contiguous.

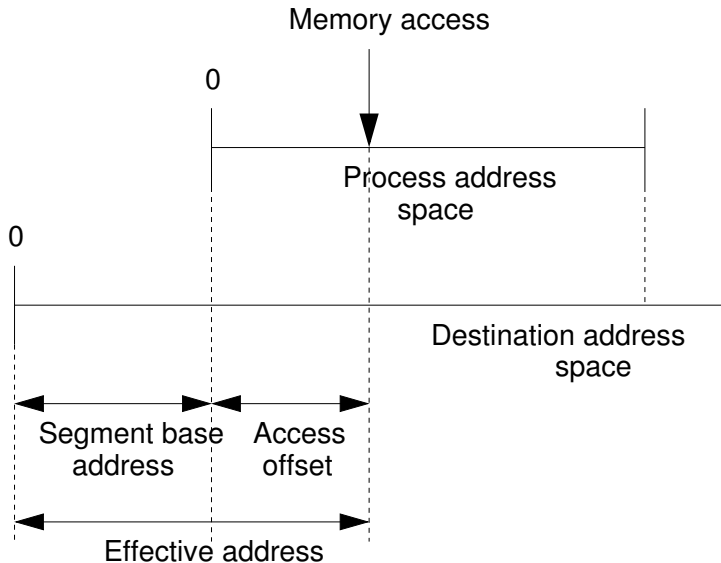


Memory access using segments

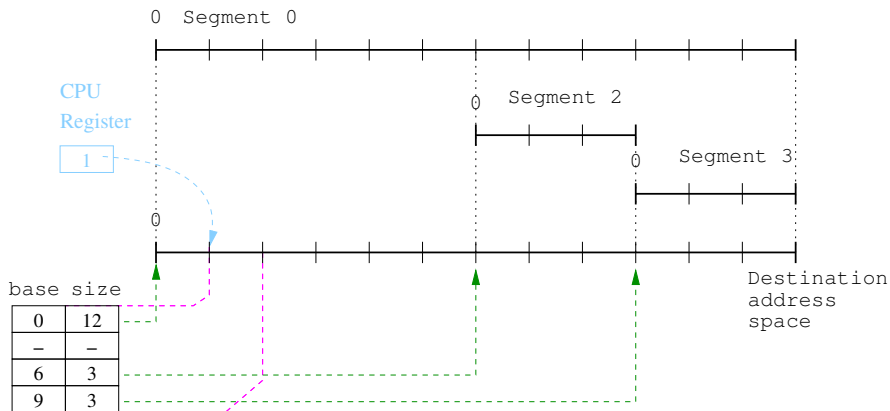
To access memory through a defined segment, the CPU performs the following tasks:

- ▶ Check requested address against the segment size,
- ▶ Add the segment base address to the requested memory address,
- ▶ Check access rights.

Memory access using segments



Segment descriptor table



Limitations of segmentation

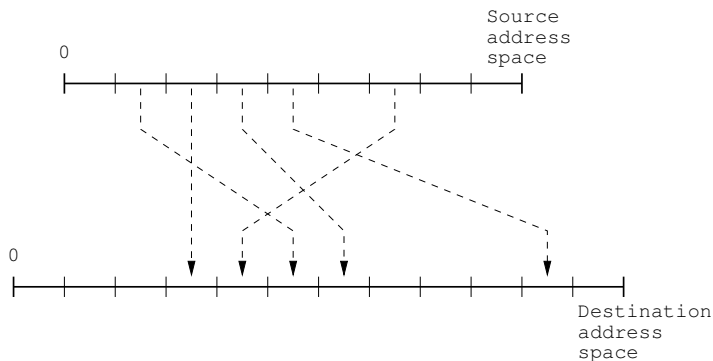
Segmentation is a great concept, but has a few limitations:

- ▶ The destination address space must be mapped in contiguous blocks,
- ▶ It can be difficult to resize segments on-demand,
- ▶ The totality of a segment must be present in the target address space.

Memory pages

Modern operating systems need more than segmentation.

The basic idea is to split the destination address space into **pages**, and logically map each source page to a destination page.

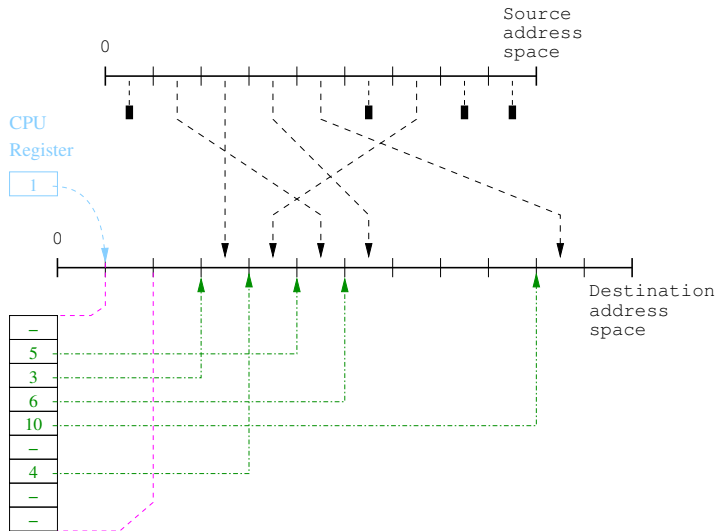


Page mapping

Memory pages need to be mapped using a specific descriptor table recognized by the CPU. Usual attributes for a page are:

- ▶ address in destination address space,
- ▶ type and access rights,
- ▶ page size,
- ▶ other attributes (cacheability, coherence, etc.).

Page descriptor table



Page mapping

Rationale

Splitting memory in pages enables a more powerful memory management:

- ▶ Source address spaces can be mapped to noncontiguous destination pages, making internal memory fragmentation possible,
- ▶ Many interesting operations can be performed on pages (sharing, swapping, copy-on-write, etc.).

Memory protection

Motivations

Why do we need memory protection?

Memory protection

Motivations

In order to securely execute operating systems, the hardware has to provide a memory protection mechanism. Such a mechanism protects:

- ▶ the kernel from the hosted applications,
- ▶ the hosted applications from each other.

It may even protect the kernel from its own components (e.g. in a “micro-kernel” approach).

Memory protection

How?

For each memory access, several checks are performed by the CPU in a transparent way:

- ▶ address bounds validity,
- ▶ privilege level,
- ▶ operation type.

Memory protection

Where?

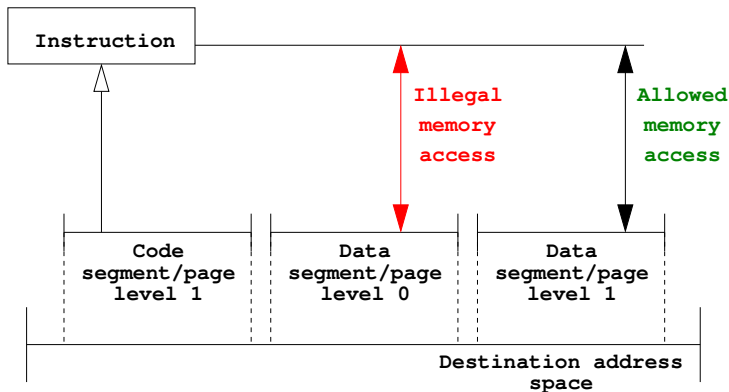
- ▶ In a segmentation-based system, privileges are defined per segment.
- ▶ In a pagination-based system, privileges are defined in the page descriptor table, i.e. at page granularity.

x86 mixes both memory protections, using with segmentation on top of pagination.

Privilege levels

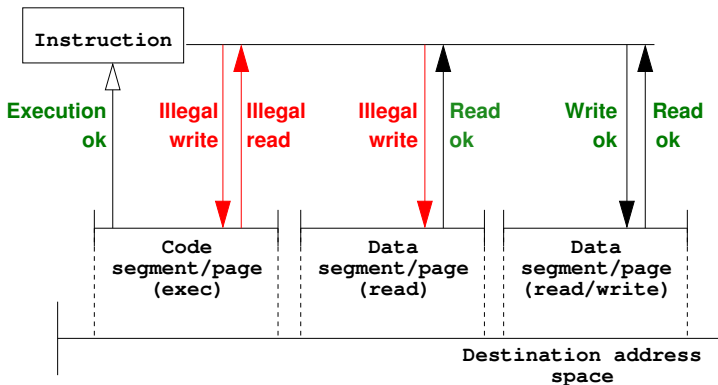
Privilege levels keep memory from being accessed by non-authorized code.

Different CPUs define different privilege levels.



Operation types

Operation types checking keeps code from doing illegal memory operations.

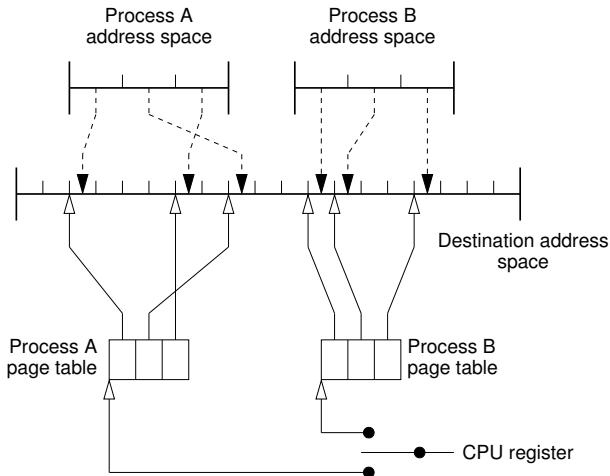


Process switching

Pagination systems are easier to use with a separate memory address space for each process running on a computer:

- ▶ Switching process address space implies only changing the page descriptor table.
- ▶ Each process has its own page descriptor table ready to be used by the CPU.
- ▶ Only the CPU register pointing to this table has to be modified to setup a new address space.

Process switching



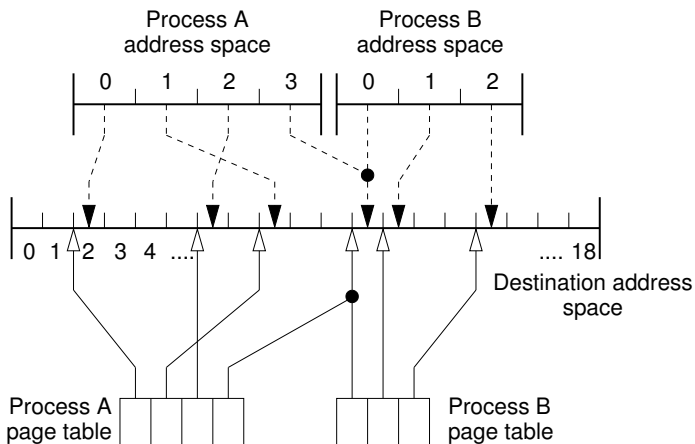
Memory sharing

Memory pagination can be used to easily share pages between processes.

A single page can be mapped in several process address spaces, which enables interesting features such as:

- ▶ Saving physical memory by not duplicating shared code and read-only data memory (e.g. shared libraries),
- ▶ Allowing processes to shared memory for inter-process communication.

Memory sharing



Copy-on-write

Definition

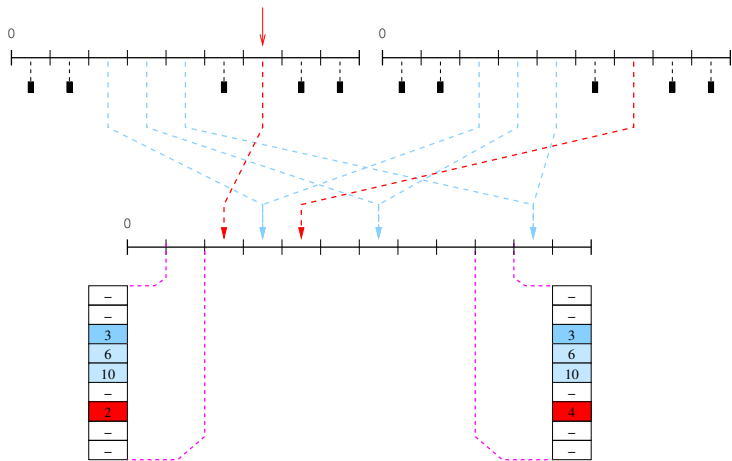
Copy-on-write (COW) is a powerful trick, which is extremely useful in many situations.

For example, when “forking” the whole address space of the original process has to be cloned.

The solution in this case, instead of actually copying the content of the whole address space, is to make the whole memory read-only and to only copy when necessary, as late as possible.

Copy-on-write

Step-by-step



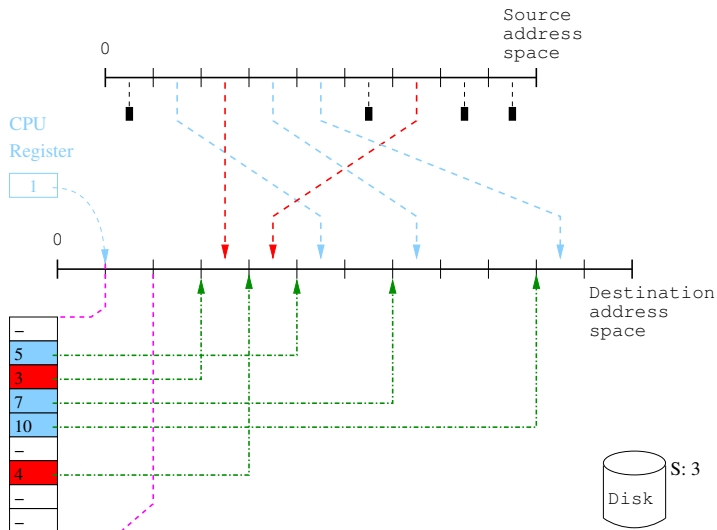
Page swapping

Definition

Page swapping is a mechanism which artificially enlarges the available physical memory by using additional space located on an hard drive.

Page swapping

Step-by-step



mmap()

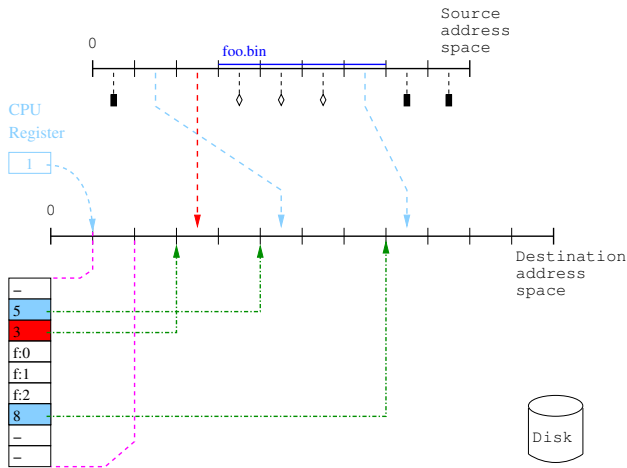
Definition

`mmap()` is a function which makes a part of the memory to match exactly the content of a file. Some of its most interesting features are:

- ▶ Changes are immediately propagated to other processes having the same file open,
- ▶ Different protections can be setup on different parts of the file,
- ▶ Parts of the file can be lazily loaded,
- ▶ Writes can be lazily performed.

mmap()

Step-by-step



Part V

Execution flow

Branch principle

Branching is the action of disrupting the incremental execution flow to go execute code somewhere else.

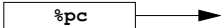
A branch is characterized by:

- ▶ a destination
- ▶ a condition (*optional*)
- ▶ a “link” feature, to save the return address (*optional*)

Branch offset

Instructions are fetched by the CPU from memory, by dereferencing the program counter (%pc register).

- ▶ In a normal execution flow, the %pc is auto-incremented.

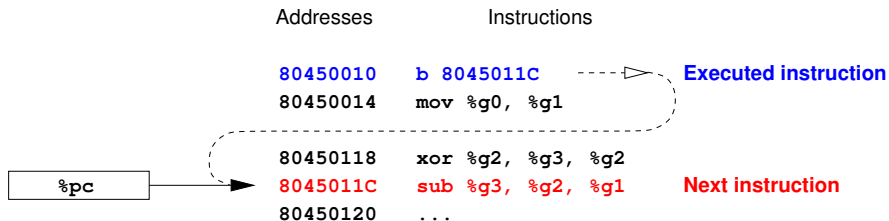
	Addresses	Instructions	
	80450010	add %g1, %g2, %g3	
	80450014	mov %g0, %g1	
	80450018	xor %g2, %g3, %g2	Executed instruction
	8045001C	sub %g3, %g2, %g1	Next instruction
	80450020	...	

- ▶ When a branch occurs, the %pc is affected in two possible ways:
 - relative branch: a constant offset is added to the %pc
 - absolute branch: an absolute address is loaded into the %pc

Unconditional branch

An unconditional branch always modifies the `%pc` register.

- ▶ Explicit jump: `goto`
- ▶ Infinite loop, optimized by the compiler



Unconditional branch

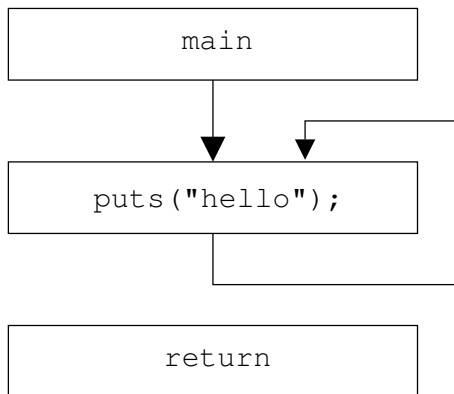
C listing

```
#include <stdio.h>

void main(void)
{
    do {
        puts("hello");
    } while (1);
}
```


Unconditional branch

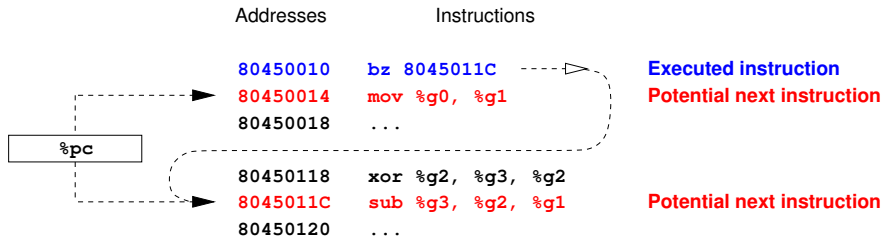
Control flow graph



Conditional branch

The `%pc` register is modified **only if** the condition is verified.

- ▶ if statements
- ▶ loop (for, while) statements



Conditional branch

C listing

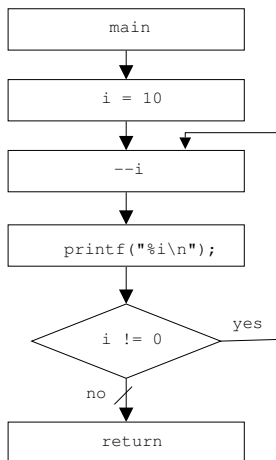
```
#include <stdio.h>

void main(void)
{
    int i = 10;

    do {
        printf("%i\n", --i);
    } while (i != 0);
}
```

Conditional branch

Control flow graph



Conditions

The decision to take a branch is based on the content of registers. A conditional branch occurs if:

- ▶ a specific bit is **set** in a register
- ▶ a specific bit is **clear** in a register
- ▶ a register equals a specific value (often zero)

Complete examples

1. `strlen`
2. `pgcd`

Pipeline considerations

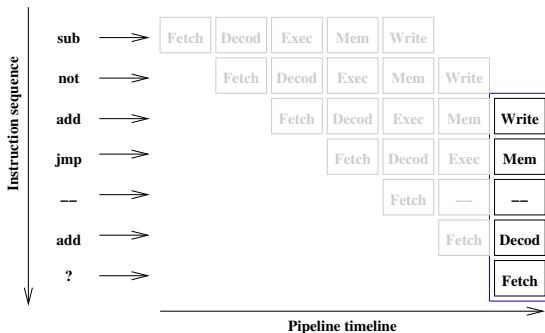
A branch may disrupt the execution flow in the pipeline, thus slowing the execution:

1. when the branch is taken, a bubble is created
2. sometimes, the processor succeeds in predicting the branch target
3. but when a mis-prediction occurs, a bubble is created

Pipeline considerations

Bubble

When a branch occurs, the processor may flush the stages of the pipeline that contain irrelevant instructions:

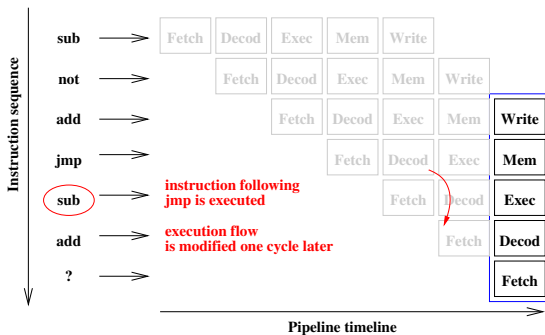


Pipeline considerations

Delay slot

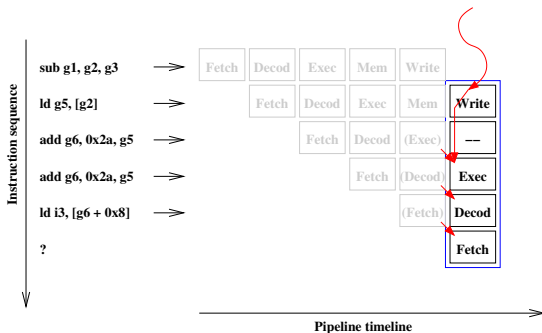
A delay slot is a processor feature. It is a convention defining that the instruction following a branch is always executed.

The branch is then delayed.



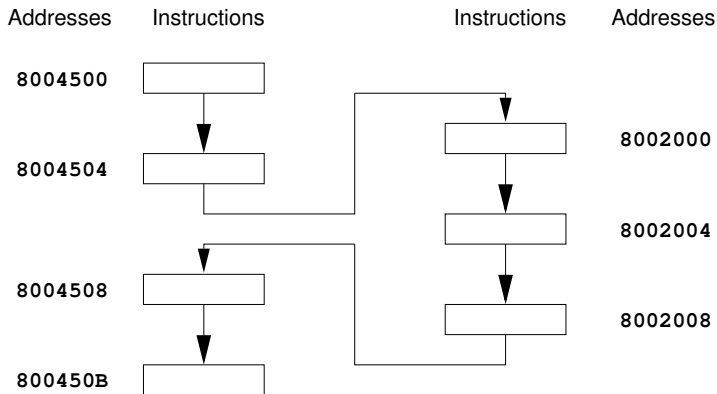
Pipeline considerations

Loads (digression)



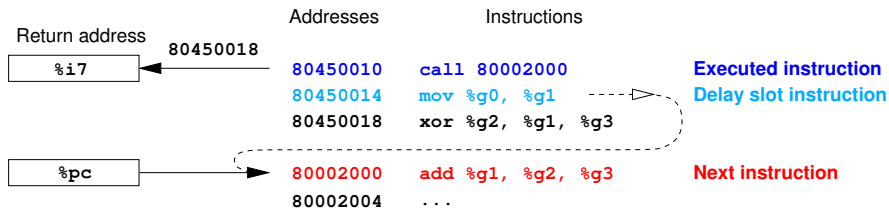
Function call principle

A function is a piece of code that returns a value depending on the parameters the user specifies as inputs, if any.



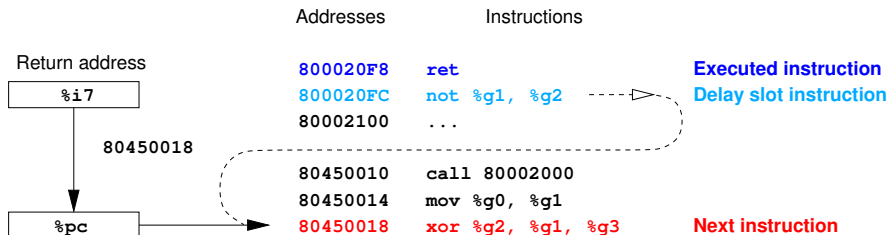
“call” instruction

- ▶ Saves next instruction address (the return path),
- ▶ Jumps to function



“ret” instruction

- Restores the previously-saved PC address



How to pass arguments and return values?

We need a space of shared data between the caller and the callee.

- ▶ From caller to callee, to hold arguments
- ▶ From callee to caller, to hold return values

Some arguments are purely for execution purposes:

- ▶ context pointers (stack, globals, ...)
- ▶ return address

Simplest case

- ▶ Non-nested function call
- ▶ Less arguments than available machine registers

Through registers

On most RISC architectures, there are registers dedicated to argument storage:

- ▶ Each argument is stored and preserved directly in a register
- ▶ Local variables may be held in another set of registers

Example: on the SPARC architecture, the CPU has:

- ▶ 8 registers (%g0, ... %g7) dedicated to global variables
- ▶ 8 registers (%i0, ... %i7) dedicated to input arguments
- ▶ 8 registers (%l0, ... %l7) dedicated to local variables
- ▶ 8 registers (%o0, ... %o7) dedicated to output arguments

A callee's "input" registers are shared with the caller's "output" registers.

Through global memory

Principle:

- ▶ Each argument is stored and preserved directly in memory
- ▶ Each local variable is held in memory

Calling convention

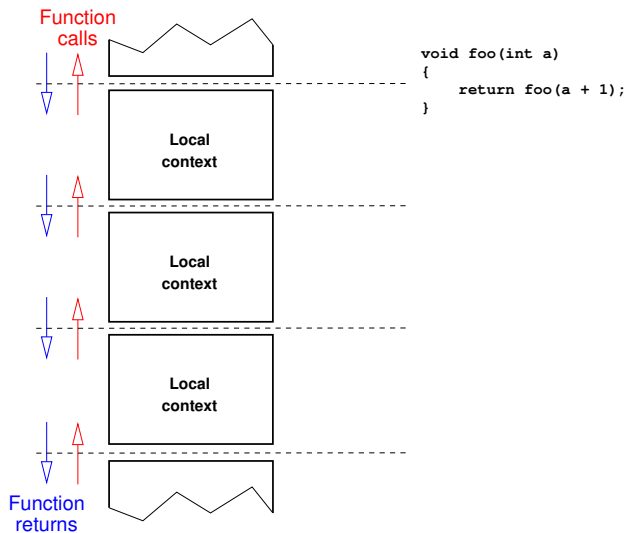
- ▶ How to deal with huge amount of variables and arguments?
- ▶ How to deal with recursive calls and nested function calls?
- ▶ How to deal with variables bigger than registers?
- ▶ How to deal with “...” (like printf)?
- ▶ How to deal with dedicated registers (floats)?

Problems

Consider a recursive function that needs local variables:

- ▶ Each time the function is called, a new context must be allocated to preserve each local variable
- ▶ Each time the function returns, the previous context must be restored
- ▶ The function may call itself an unpredictable number of times
- ▶ These local variables cannot be reserved in a static memory space (as global variables are)

Abstract context stack

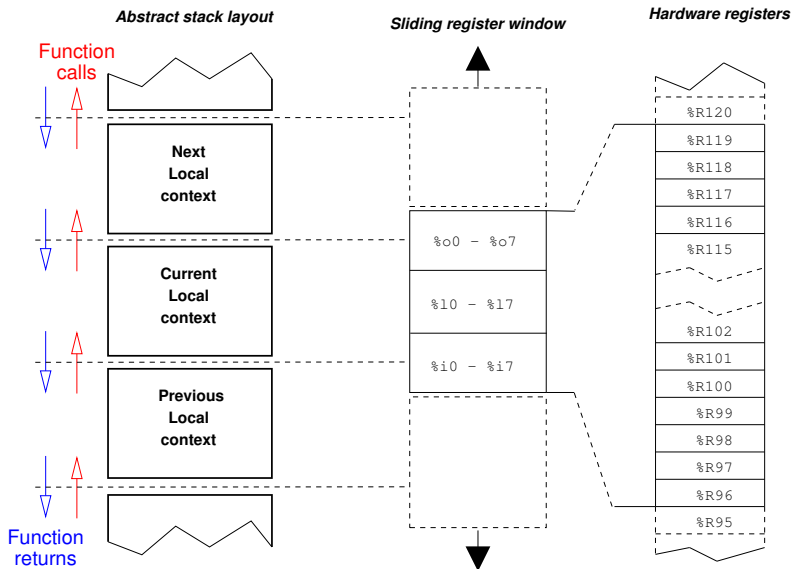


Register window

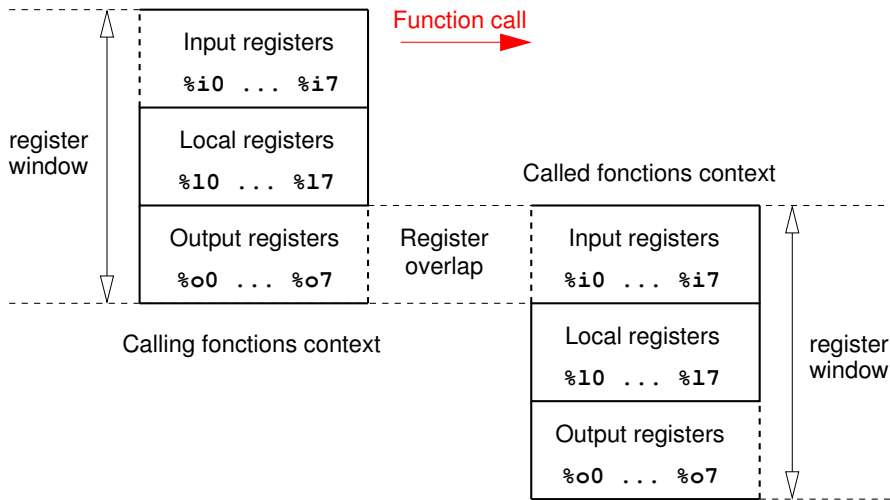
Hardware context stack implementation:

- ▶ Uses a large amount of registers (example: up to 520 on SPARC v8)
- ▶ Uses a logical limitation to define multiple contexts
- ▶ Performs context changes by sliding a window on each function call

Principle



SPARC overlapping register window



Function prologue and epilogue

prologue: slide register window

Example:

```
save %sp, -96, %sp
```

epilogue: slide back register window (i.e. restore previous context)

Example:

```
restore
```


Function prologue and epilogue

```
save r1, constant, r3
```

1. $\text{temp} = \text{r1} + \text{constant}$
2. $\text{CWP} = \text{CWP} - 1$ (slides the window down)
3. $\text{r3} = \text{temp}$

Beware, in "save %sp, -96, %sp", both %sp are different: they belong to different register windows!

```
restore
```

1. $\text{temp} = \text{r1} + \text{constant}$
2. $\text{CWP} = \text{CWP} + 1$ (slides the window up)
3. $\text{r3} = \text{temp}$

restore without any argument means "restore %g0, 0, %g0" which only slides up the window.

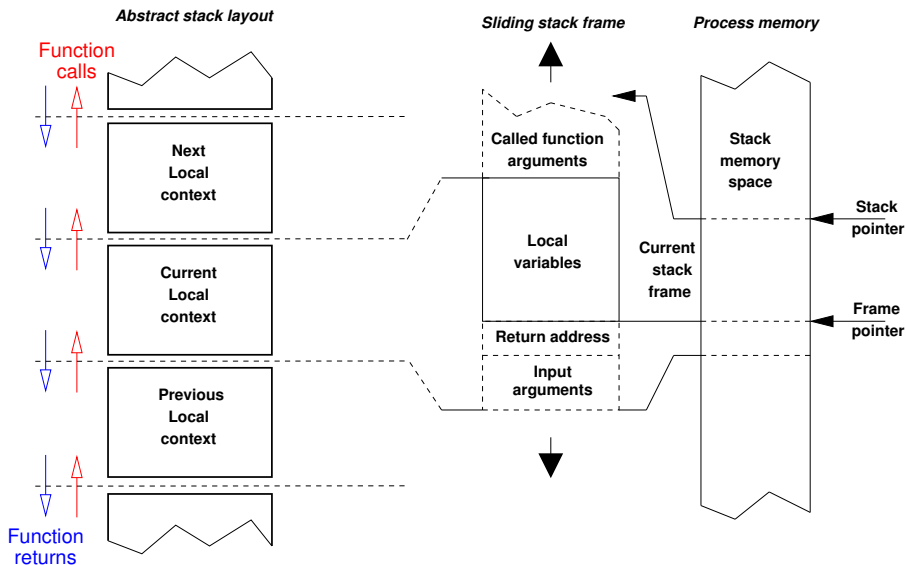
Memory stack

The register window approach has some hard limitations:

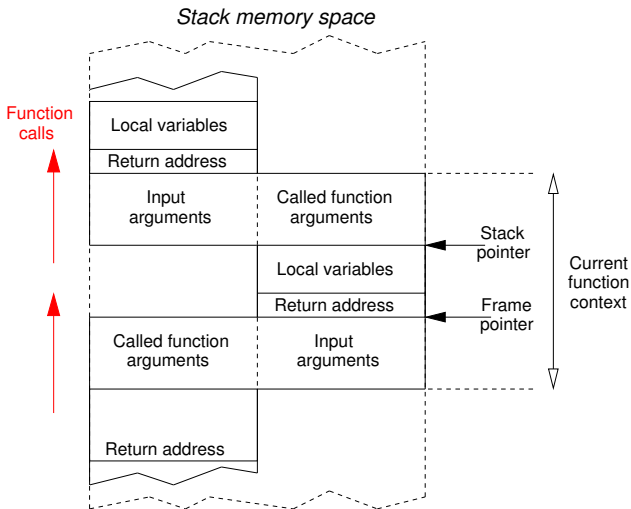
- ▶ The call depth is limited to the total amount of registers
- ▶ The CPU needs a large amount of physical registers \Rightarrow expensive

On most systems, memory is used to implement a cheaper stack.

Memory stack principle



Nested function calls



Function prologue and epilogue

prologue: saves previous frame pointer, set new frame pointer, reserve space on memory stack for local variables

Example: a function that needs three 32-bit local variables (12 bytes) on its stack:

```
[%sp] <- %fp  
%fp <- %sp  
%sp <- %sp - 12
```

epilogue: restores previous frame and stack pointers (i.e. restore previous context)

Example:

```
%sp <- %fp  
%fp <- [%fp]
```

Argument and local variable access

argument: Dereference the address “frame pointer + argument offset”:

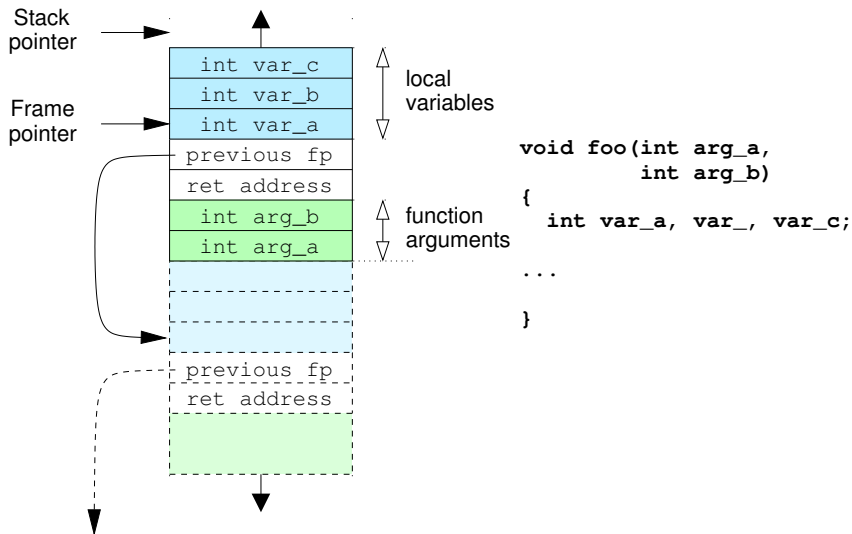
```
ld [%fp + 16], %g1; Access to arg_a
```

local variable: Dereference the address “frame pointer - local variable offset”

```
ld [%fp - 4], %g1; Access to var_b
```

Argument and local variable access

schema



Events

- ▶ What are events?
- ▶ How to handle them?

Event classification

An **interrupt** is caused by an external event.

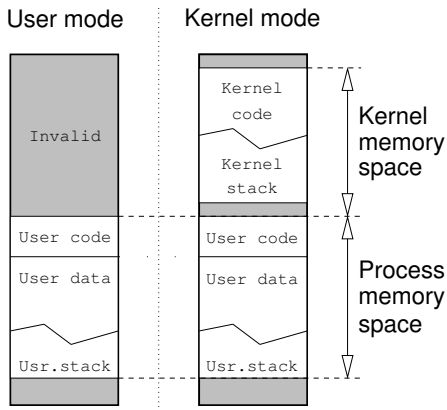
An **exception** is caused by an instruction execution.

	Unplanned	Deliberate
Synchronous	<i>fault</i>	<i>syscall trap, software interrupt</i>
Asynchronous	<i>hardware interruption</i>	

→ An incoming event must be executed with a high priority.

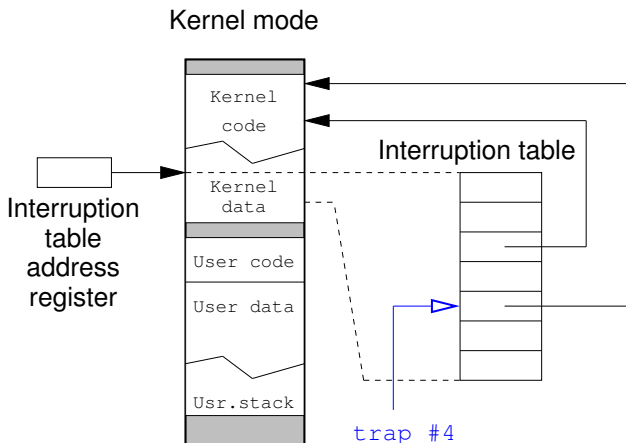
User space / kernel space

A trap is a critical event that must be handled by the kernel in a safe memory space, **the kernel space**.



Trap handler table

The processor jumps to a trap routine defined by the operating system.
The trap routine address is stored in a dedicated descriptor table.



System call

The system call mechanism can be used by a process to request services from the operating system:

- ▶ execute a process, exit
- ▶ read input, write output (`read`, `write...`)
- ▶ perform *restricted actions* such as accessing hardware devices or accessing the memory management unit.
- ▶ etc.

Processor modes

Some hardware features ensure isolation between processes in multi-user/multi-process environment.

Processors usually have two (or more) execution modes:

- user mode:** dedicated to user applications

- supervisor mode:** reserved for the operating system kernel

Execution permissions

For security sake, some operations are restricted to kernel space:

- ▶ peripheral input and output
- ▶ low-level memory management

System calls enable the user space to *switch* to kernel space and perform restricted actions, under certain conditions.

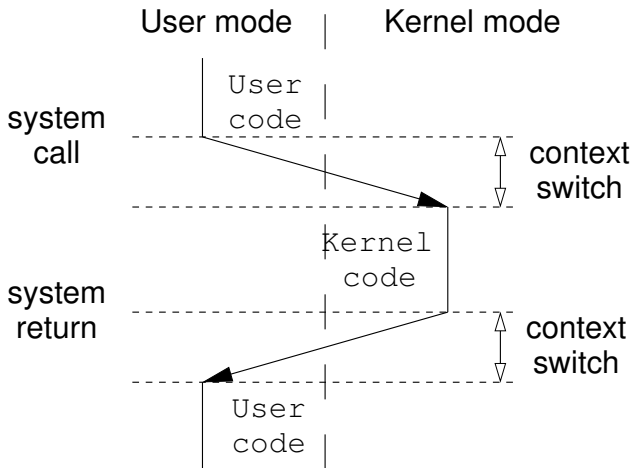
The kernel must check the validity of system call parameters!

System call implementation

1. System calls use a dedicated instruction which causes the processor to change mode to *supervisor* (or *protected*) mode.
2. Each system call is indexed by a single number: the syscall trap handler makes an indirect call through the *system call dispatch table* to the handler for the specific system call.

Execution path

System calls run in kernel mode on the *kernel memory space*.



libc example

Standard C library's read, write, pipe, etc. functions are merely *wrappers* to corresponding system calls:

```
_SYSENTRY(pipe)
mov     %o0, %o2
mov     SYS_pipe, %g1
ta      %xcc, ST_SYSCALL
bcc,a,pt %xcc, 1f
stw     %o0, [%o2]
ERROR()
1:
stw     %o1, [%o2 + 4]
retl
clr     %o0
_SYSEND(pipe)
```

Faults

- ▶ How to handle *divide by zero*?
- ▶ How to handle *overflow*?
- ▶ How to handle ...

Similarities with system calls

Faults are similar to system calls in some respects:

- ▶ Faults occur as a result of a process executing an instruction
- ▶ The kernel exception handler may return to the faulty user context

But faults are different in other respects:

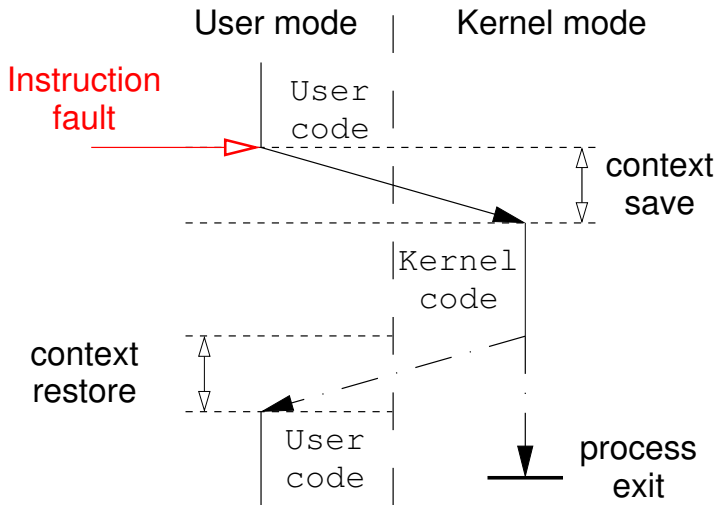
- ▶ Syscalls are deliberate, faults are unexpected
- ▶ Not every execution of the instruction results in a fault

Handling a fault

Different actions may be taken by the operating system in response to faults:

- ▶ kill the user process
- ▶ notify the process that a fault occurred (so it may recover in its own way)
- ▶ solve the problem and resume the process transparently

Execution path



Event interface

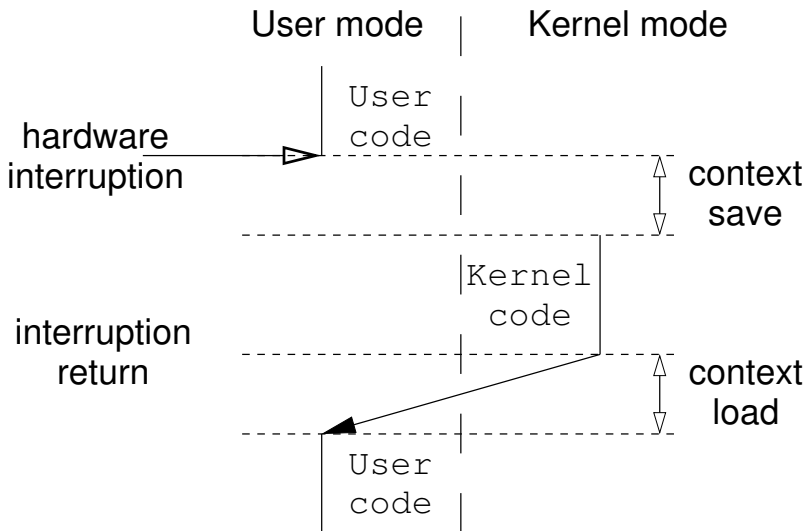
Unix systems can notify a user program of a fault with a *signal*.

Signals are also used for other forms of asynchronous event notifications.

Hardware interruptions

- ▶ How to handle devices interruptions?

Execution path



Multitasking

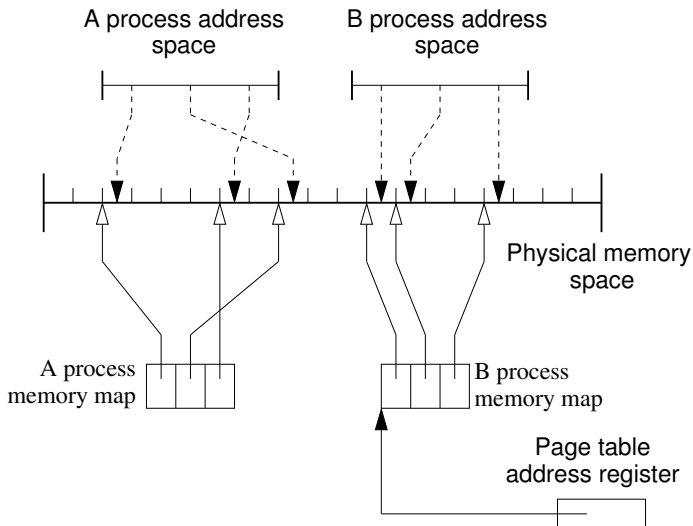
A single process may not use all the system resources at full capacity.

The idea of multitasking is to simulate the concurrent executions of multiple processes, using a single processor.

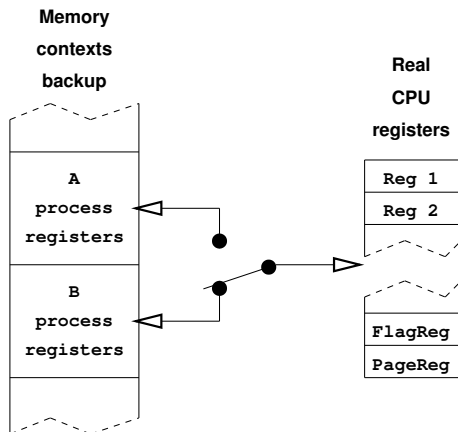
The operating system has to:

- ▶ create and delete processes,
- ▶ organize processes in memory,
- ▶ schedule processes for CPU use.

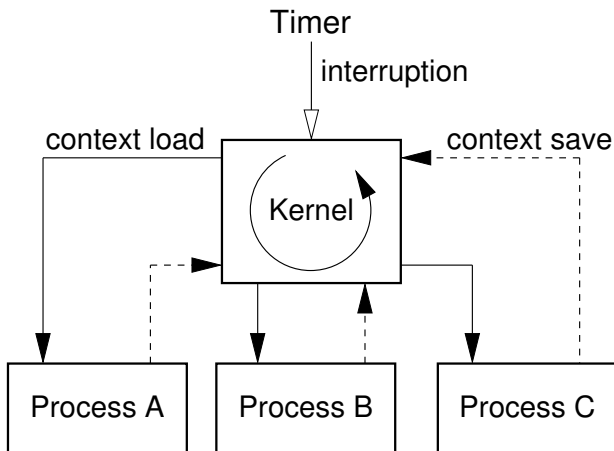
Memory mapping



Context switching



Process life

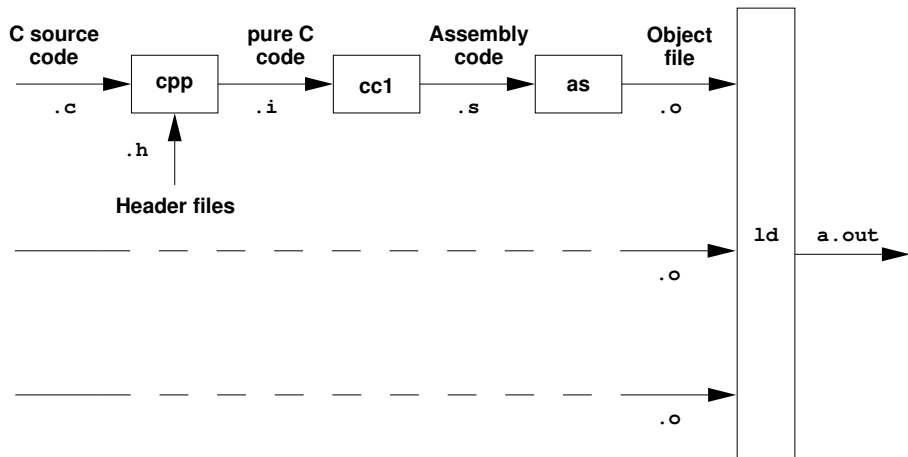


This approach requires a way of handling events.

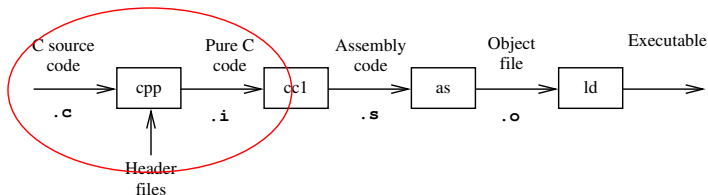
Part VI

Object file formats

The big picture



Preprocessor



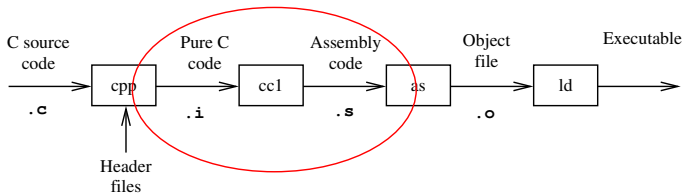
- ▶ Also called macro-processor
- ▶ Transforms a text (source) file into another text (source) file:
 - ▶ merging many files into one (`#include`)
 - ▶ replacing macros by their definitions (`#define`)
 - ▶ removing code considering conditions (`#if*`)

Example: `cpp`

Input: C source file with directives

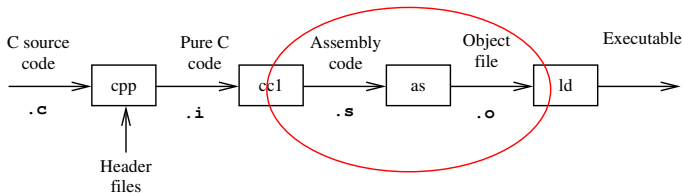
Output: “pure” C source file

C compiler



- ▶ Scans and parses the source file
- ▶ Analyzes the source (type checking)
- ▶ Generates assembly code (another source file) for the target architecture

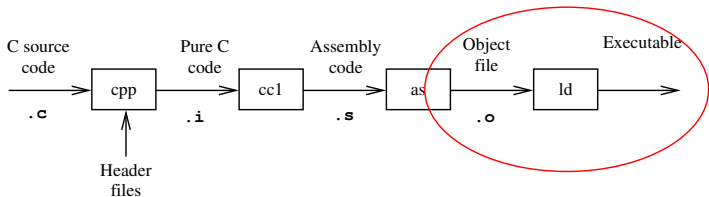
Assembler



- ▶ Translates instructions into a binary opcode sequence for the target architecture
- ▶ Collects symbols and resolves label addresses
- ▶ Writes an object file

The assembler source file depends on the targeted architecture!

Linker



- ▶ Merges (many) object files
- ▶ Resolves external symbols
- ▶ Computes final addresses
- ▶ Writes an executable file

Development tools

- ▶ SPARC assembler: use aasm
 - ▶ Project at <http://savannah.nongnu.org/projects/aasm>
- ▶ Sparc, Mips, PPC, ARM, etc. architecture emulators
 - ▶ QEMU
 - ▶ SoCLib: <https://www.soclib.fr>

objdump

- ▶ Displays object (executable) file tables, on host architecture
- ▶ Disassembles object (executable) file code sections
- ▶ Useful options:
 - ▶ `-h`: display the section headers
 - ▶ `-t`: display the symbol tables
 - ▶ `-d`: disassemble
 - ▶ `-S`: Display source code intermixed with disassembly, if possible (implies `-d`)

readelf

- Displays ELF object (executable) file tables, on any architecture

ELF Header:

```

Magic:    7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
Class:                                ELF64
Data:                                2's complement, little endian
Version:                                1 (current)
OS/ABI:                                UNIX - System V
ABI Version:                            0
Type:                                EXEC (Executable file)
Machine:                                Advanced Micro Devices X86-64
Version:                                0x1
Entry point address:                    0x410009
Start of program headers:                64 (bytes into file)
Start of section headers:                136312 (bytes into file)
Flags:                                0x0
Size of this header:                    64 (bytes)
Size of program headers:                56 (bytes)
Number of program headers:                10
Size of section headers:                64 (bytes)
Number of section headers:                32
Section header string table index: 31

```

Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info Align	
[0]		NULL	0000000000000000	00000000
	0000000000000000	0000000000000000	0 0 0	
[1]	.interp	PROGBITS	0000000000400270	00000270
	000000000000001c	0000000000000000	A 0 0 1	
	Joël Porquet (EPITA)		CAAL	

Simple binary formats

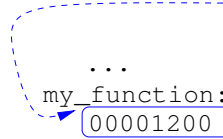
Consider an object file that does not need any other object files in order to build a final binary program. The assembler is then able to:

- ▶ collect code markers (labels)
- ▶ resolve **all** labels and replace **all of them** in the code

Flat program

Labels are immediately replaced and written by the assembler in the binary file:

Address	Opcodes	Source code
00000000	90	nop
00000001	B8 78 56 34 12	mov eax, 0x12345678
00000006	E8 F5 11 00 00	call my_function
...		
my_function:		
00001200	90	nop
00001201	...	



Flat binary format

In raw binary format, the whole program is written directly in file.

Useful at boot time where the processor can only read raw code, and no binary loader is available.

Challenges

- ▶ What happens when a function must be imported?
- ▶ What happens when a function must be exported?

Complex program

When the symbol of a called function is unresolved, the assembler leaves the `call` destination address undefined:

Address	Opcodes	Source code
00000000	90	<code>nop</code>
00000001	B8 78 56 34 12	<code>mov eax, 0x12345678</code>
00000006	E8 F5 11 00 00	<code>call my_function</code>
00000006	E8 00 00 00 00	<code>call printf</code>
...		
<code>my_function:</code>		
00001200	90	<code>nop</code>
00001201	...	

Diagram illustrating the unresolved call destination address:

- The `call my_function` instruction at address 00000006 has an opcode `E8 F5 11 00 00`. The `F5 11 00 00` bytes are highlighted with a blue box.
- The `call printf` instruction at address 00000006 has an opcode `E8 00 00 00 00`. The `00 00 00 00` bytes are highlighted with a red box.
- A dashed blue arrow points from the `my_function:` label to the address 00001200.
- A dashed red arrow points from the `00 00 00 00` bytes of the `call printf` instruction to the text "External reference".

⇒ The binary format must hold information on created “holes”

Needs

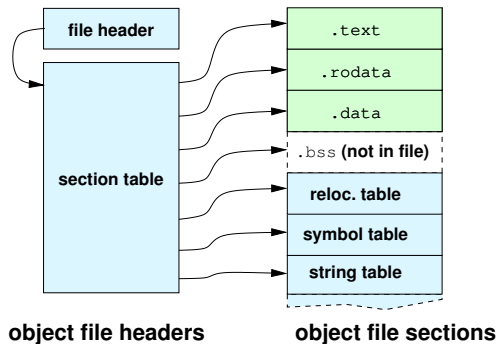
In order to later fill in the holes left by the assembler, the binary file must contain:

- ▶ A symbol table, which stores each label identifier,
- ▶ A relocation table, which shows all the remaining “holes”,
- ▶ Labels associated to each “hole”.

More generally, a binary file must also contain:

- ▶ A header containing general information needed to access various parts of the file,
- ▶ Several sections holding code and data (raw data).

Abstraction of a binary file format

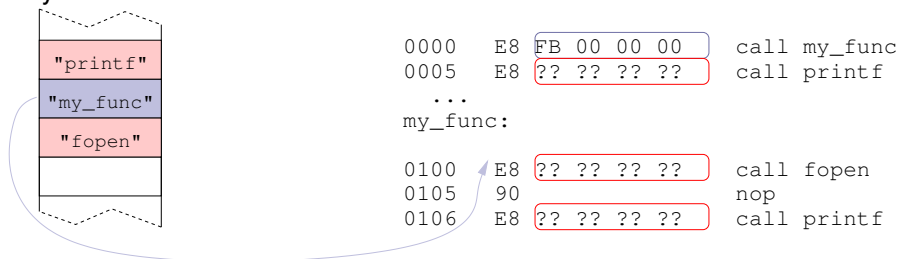


(Information regarding sections, symbol table, etc. are usually identified within the file header)

The symbol table

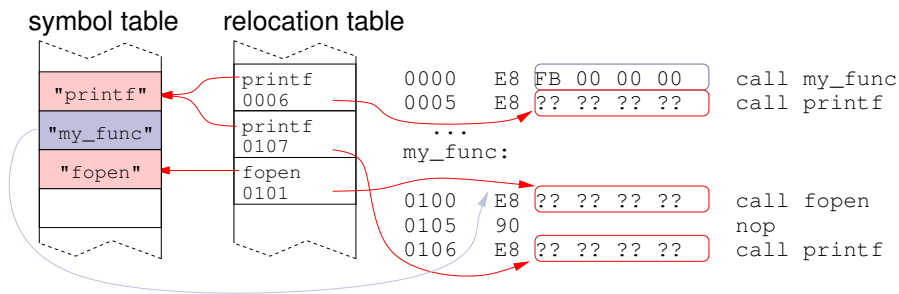
The symbol table associates each symbol identifier to the code (or data) address it represents (when the address has been resolved):

symbol table



The relocation table

The relocation table associates each “hole” address to a label identifier address:



BSS region

Instead of keeping a bunch of empty bytes in the binary file for big static variables (arrays), the header can specify a whole region which must be filled with zero.

Using a BSS region makes the file smaller and faster to load.

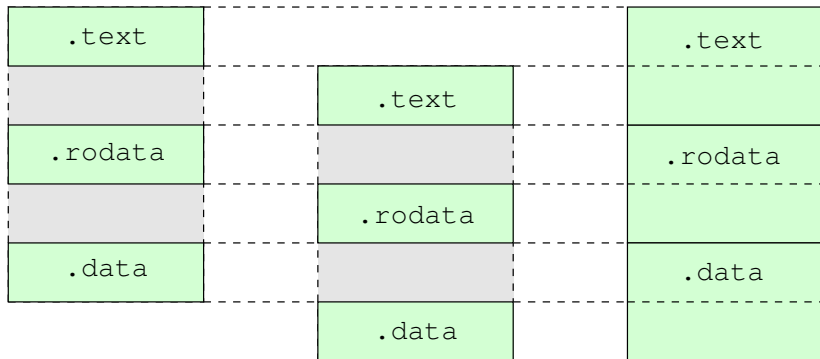
Linking

The operation that combines a list of object programs into a binary program is called linking. The tasks that must be accomplished by the linker are:

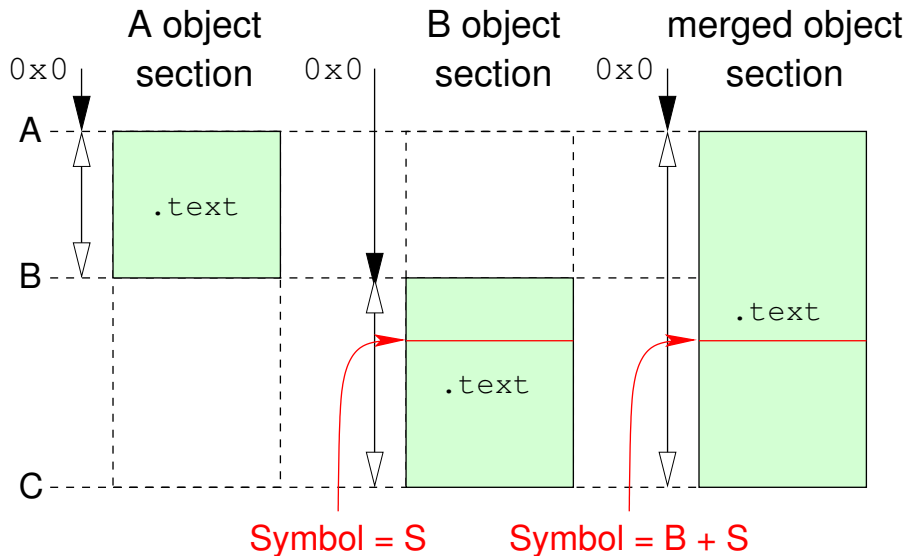
1. Named sections from different object programs must be merged together into one named section.
2. Merged sections must be put together into the sections of the memory model.
3. Each use of a name in an object program's references list must be replaced by an address in the virtual address space.

Merging object files

Combining each `.text` section together, each `.data` section together, etc.:



Zoom on .text section merging

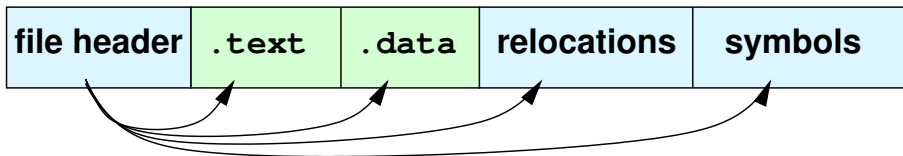


Symbol resolution

raw binary format

`.text / .data`

a.out format



Executable binary files

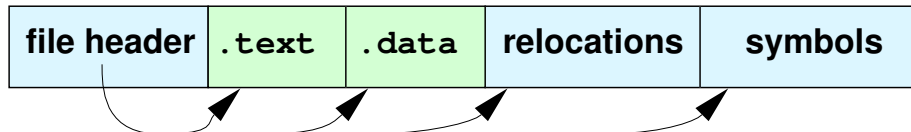
- ▶ When there is neither unresolved symbols nor relocations left, the file is executable.
- ▶ Executable files usually have a fixed constant load address on a given system.
- ▶ Unresolved/unused symbols may exist in an executable file if no relocation uses them.
- ▶ The `strip` command wipes out unused symbols from the symbol table.

a.out: Assembler Output

raw binary format

.text / .data

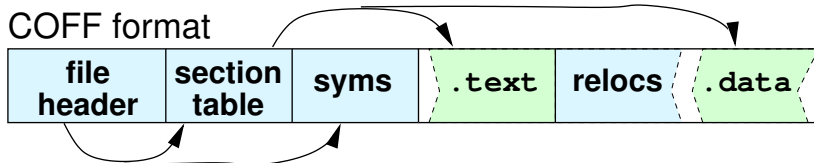
a.out format



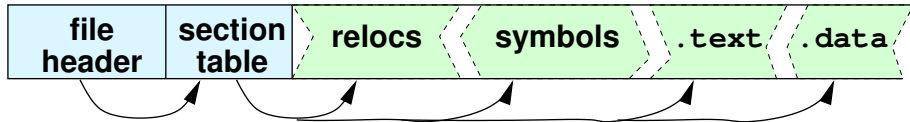
- ▶ Quite dumb
- ▶ ...but pretty fast to load

COFF (Common Object File Format) and ELF (Executable and Linking Format)

COFF format

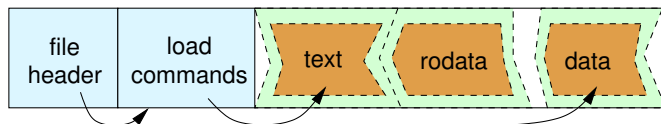


ELF format



- Enables using dynamic (relocatable) libraries

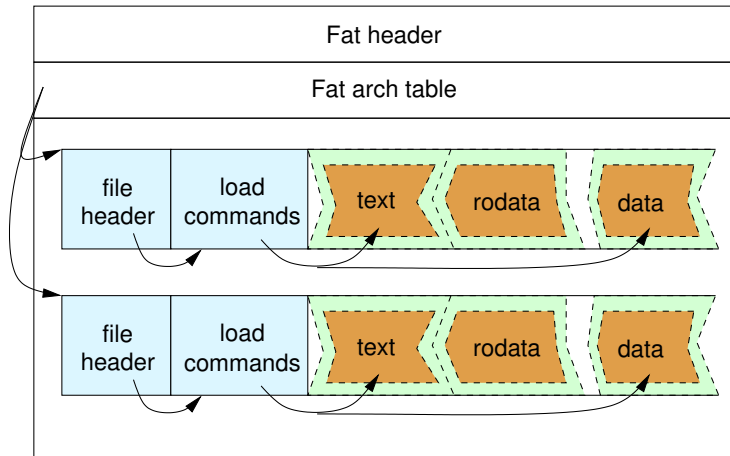
Mach-O (Darwin)



Mach-O (Darwin)

“Fat binaries”

Mach-O fat binaries

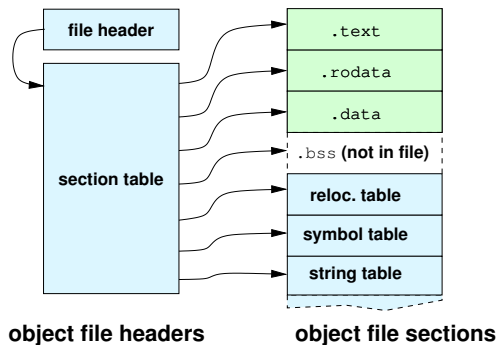


Binary file loading

An executable file format is like an object file format, but with the following restrictions:

- ▶ It has no internal relocations
- ▶ All internal addresses are resolved
- ▶ It is ready to be loaded in memory

Executable format

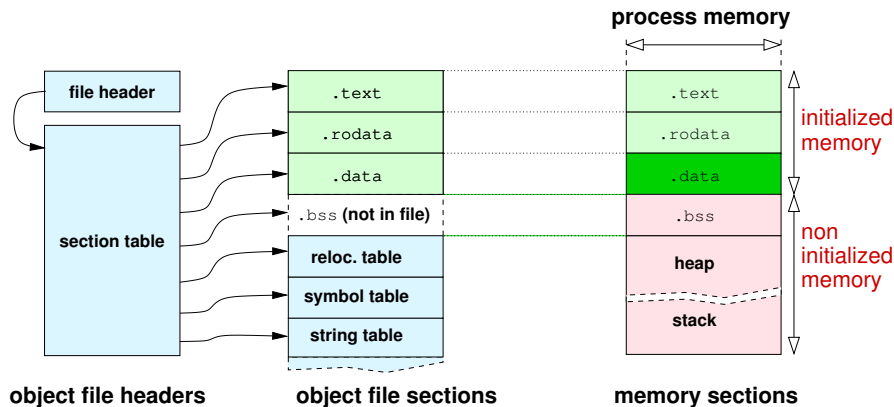


Executable loading

The memory structure of a process corresponds to the internal executable file format:

- ▶ Loadable sections are loaded from file to memory
- ▶ Uninitialised data (`.bss` section) is directly allocated in process memory
- ▶ Internal file sections are ignored
- ▶ Heap and stack are directly allocated

Executable memory mapping

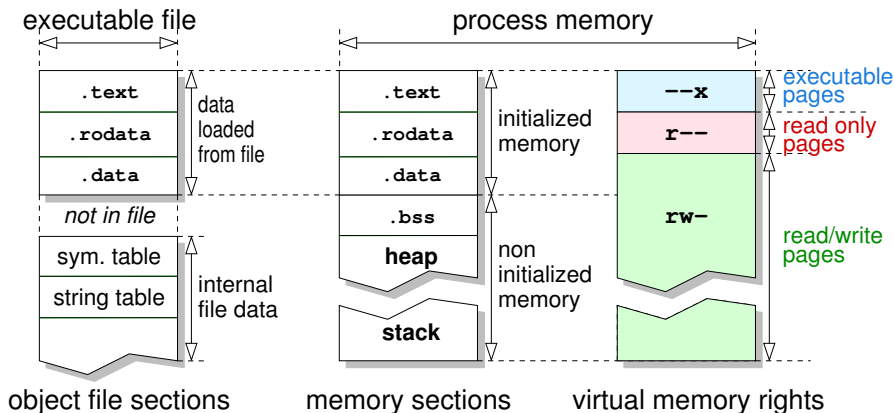


Memory attributes

Memory attributes depend on section and area type:

- ▶ the `.text` section may be loaded in a executable-only memory region (depending on OS and hardware architecture).
- ▶ the region of memory used to store `.rodata` section will be marked as read-only.
- ▶ the region of memory used to store `.data`, `.bss` sections, heap and stack will be marked as read/write.

Memory attributes



Dynamic libraries

Using dynamic libraries implies:

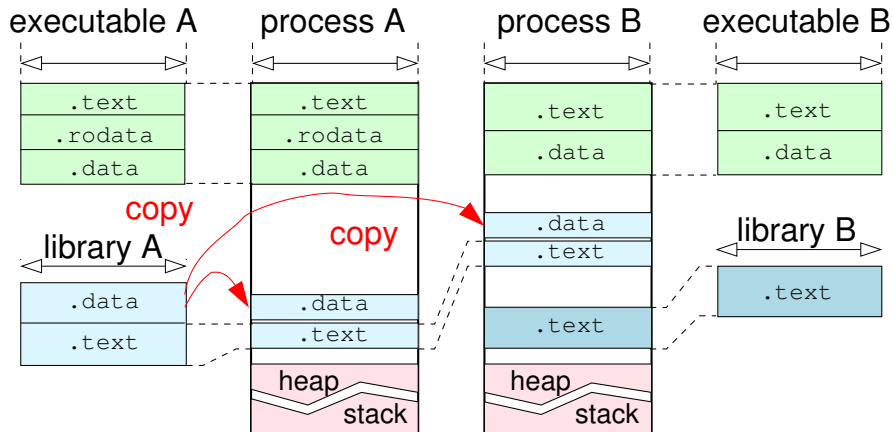
- ▶ a different and more complex memory mapping
- ▶ position independent code (PIC)
- ▶ different ways to handle relocations

Dynamic libs in memory

Libraries have specific memory mapping and organisation:

- ▶ A dynamic library is loaded only once even when several processes use it
- ▶ Library `.text` section is **mapped** in each process memory space
- ▶ Library `.data` section is **copied** in each process memory space

Dynamic process memory layout



Handling code location change

Dynamic libraries may not be mapped at the same virtual address in all processes:

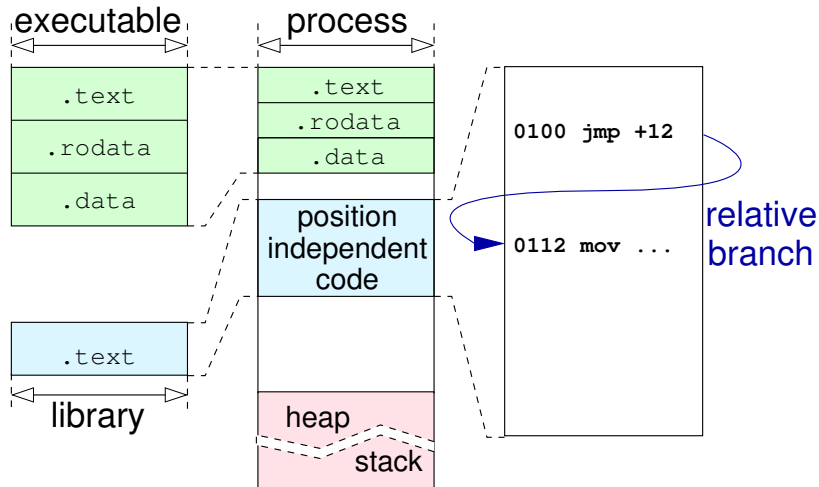
- ▶ The location may already be in use by an other library
- ▶ The same code must be able to run with different base addresses

Position independent code solves these problems without going through a complex relocation process.

Let's try to make everything be relative to each other! Relative jumps and relative data accesses...

Position independent code

Internal function calls



Position dependent code

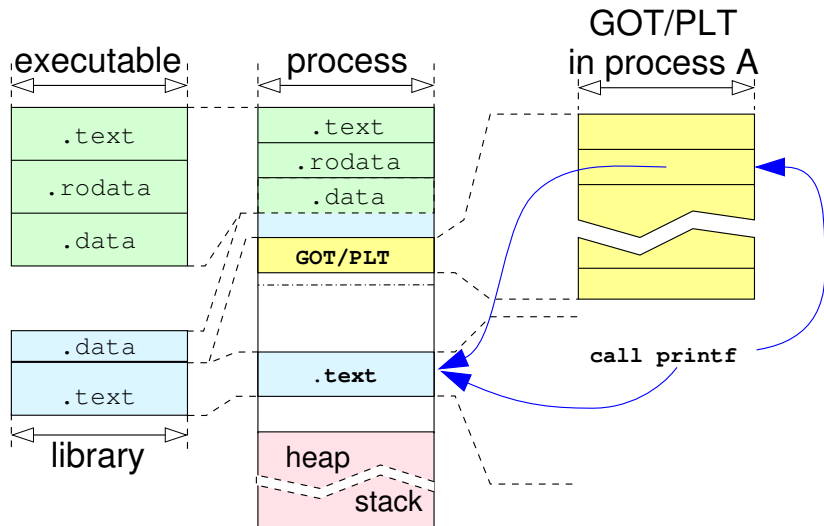
Data and function references

Main idea of PIC: add an additional level of indirection to all global data and function references in the code.

1. offset between text and data sections (known at link-time)
2. make PC-relative offset accesses

At the very beginning of the data section, we find the GOT (*Global Offset Table*) and PLT (*Procedure Linkage Table*). They hold data object and function addresses for each object module.

Use of GOT/PLT



Part VII

Assembly programming

What is assembly?

Assembly **is not**:

- ▶ binary data
- ▶ directly interpreted by the processor
- ▶ do not mix it up with *machine code*

Assembly **is**:

- ▶ a language, with a syntax, keywords, etc.
- ▶ translated in a binary format

Benefits of assembly programming

- ▶ Improves developer's understanding of computer architecture and programming
- ▶ Direct access to hardware resources
- ▶ Access to system features
- ▶ Speed and efficiency of programs
- ▶ Low footprint of programs

Drawbacks of assembly programming

- ▶ Low-level programming forbids abstraction
- ▶ Assembly code is hard to keep clean
- ▶ Slows development down: lots of keywords, unusual behaviors
- ▶ Each processor architecture has its own language: conventions, registers, instructions, privileges, allowed arguments

Assembly file organization

A source file is organized into different sections:

`code`

Contains program instructions

`data`

Contains program global variables

`rodata`

Contains read-only program variables

Assembly language statements

directives

Determine the assembler behavior

instructions

Chosen in the CPU instruction set, translated in binary format, written in output file

operands

Expressions containing register identifiers, immediate operands, symbols

labels

Between instructions, used to mark specific locations in source code

Assembly language statements

Example

```
.mod_load    asm-sparc
.define F00 5

.section code .text
    .mod_asm opcodes v8

    add %g0, F00, %g1
.ends
```

Assembly language statements

Example

```
.mod_load asm-sparc
.mod_load out-elf32

.section code .text
    .mod_asm opcodes v8

    mov 0x12, %g1
    mov 0x34533, %g2

    add %g1, %g2, %g3
.ends
```

Assembly language statements

Example

```
.mod_load asm-sparc
.include sparc/v8.def

.section data .data
lbl:
    .reserve 4
.ends

.section code .text
    .mod_asm opcodes v8

    @set .data:lbl, %g2
    ld [%g2], %g1
.ends
```

Assembly language statements

Example

```
.mod_load asm-sparc
.include sparc/v8.def

.section code .text
    .mod_asm opcodes v8

    .export main
    .proc main
        ret
        restore
    .endp
.ends
```

Assembly language statements

Example

```
.mod_load asm-sparc
.include sparc/v8.def

.section code .text
    .mod_asm opcodes v8

    .extern exit

    .proc my_exit
        call exit
        nop
    .endp
.ends
```


Why?

- ▶ C can't express everything
 - ▶ CPU-specific registers (flags, condition codes, hardware counters)
 - ▶ Features not addressed by language (atomic operations)
 - ▶ System features (Memory handling, interrupts handling, exception handling)
- ▶ Compiler are not always aware of possible optimizations
 - ▶ Use of instruction side-effects
 - ▶ Un-optimizable patterns (or optimization patterns specific to only one algorithm)

Inline Assembly

Compiler's PoV

For the compiler, the inline assembly you give is:

- ▶ a raw string
- ▶ with a printf-like syntax
- ▶ passed directly to the assembler
- ▶ surrounded by optional statements
 - ▶ input values
 - ▶ output values
 - ▶ clobbered values

Inline Assembly

Programmer's PoV

For you, the inline assembly you give is meant to either:

- ▶ compute a value
- ▶ interact with a specific hardware feature
- ▶ have a side-effect

Example

No data

```
static inline
void disable_interrupts(void)
{
    asm volatile("cli");
}
```

Example

Output only

```
typedef uint64_t cpu_cycle_t;

static inline
cpu_cycle_t cpu_cycle_count(void)
{
    uint32_t      low, high;

    asm volatile("rdtsc" : "=a" (low), "=d" (high));

    return (low | ((uint64_t)high << 32));
}
```

Example

Input only

```
static inline
void cpu_io_write_8(uintptr_t addr, uint8_t data)
{
    asm volatile("outb    %0,    %1"
                 :
                 : "a" (data), "d" ((uint16_t)addr)
                 );
}
```

Example

In-out

```
static inline
uint32_t cpu_endian_swap32(uint32_t x)
{
    asm ("bswap    %0"
        : "=r" (x)
        : "0" (x)
        );

    return x;
}
```

Syntax

- ▶ one statement with optional arguments

```
asm [volatile] ("statements    \n\t"  
                "on many lines \n\t"  
                [: [output_variables]  
                [: [input_variables]  
                [: clobbered_registers]])  
    );
```

- ▶ arguments are referenced in-order (%0..%n), whether they are input or output!
- ▶ arguments types abide constraints, enclosed in `""`

Add with carry and overflow

C

```
uint32_t add_cv(uint32_t a, uint32_t b, uint8_t cin,
                uint8_t *cout, uint8_t *vout)
{
    uint64_t sum = (uint64_t)a + (uint64_t)b + (uint64_t)cin;
    *cout = sum >> 32;
    *vout = ( (b ^ ((uint32_t)sum)) & ~(a^b) ) >> 31;
    return sum;
}
```

Add with carry and overflow

ASM

```
uint32_t add_cv(uint32_t a, uint32_t b, uint8_t cin,
                uint8_t *cout, uint8_t *vout)
{
    uint32_t result;

    asm(
        "        btl $0, %k5          \n"
        "        adcl %k3, %k4        \n"
        "        setc %b1              \n"
        "        seto %b2              \n"
        : "=r" (result), "=qm" (*cout), "=qm" (*vout)
        : "r" (a), "0" (b), "r" (cin)
    );

    return result;
}
```

Atomic increment for ARM

```
static inline
bool_t cpu_atomic_inc(volatile atomic_int_t *a)
{
    reg_t tmp = 0, tmp2;

    asm volatile("1:                                \n\t"
                 "ldrex    %0, [%2]                  \n\t"
                 "add      %0, %0, #1                 \n\t"
                 "strex    %1, %0, [%2]              \n\t"
                 "tst      %1, #1                     \n\t"
                 "bne      1b                          \n\t"
                 : "=&r" (tmp), "=&r" (tmp2)
                 : "r" (a)
                 : "m" (*a)
                 );

    return tmp != 0;
}
```

Named values

- ▶ Using numbers for values makes code harder to write and read
- ▶ GCC permits named values in inline assembly

Named values

With numbers

```
static inline
bool_t cpu_atomic_inc(volatile atomic_int_t *a)
{
    reg_t tmp = 0, tmp2;

    asm volatile("1:                                \n\t"
                 "ldrex    %0, [%2]                  \n\t"
                 "add      %0, %0, #1                \n\t"
                 "strex    %1, %0, [%2]              \n\t"
                 "tst      %1, #1                    \n\t"
                 "bne      1b                        \n\t"
                 : "=&r" (tmp), "=&r" (tmp2)
                 : "r" (a)
                 : "m" (*a)
                 );

    return tmp != 0;
}
```

Named values

Named

```
static inline
bool_t cpu_atomic_inc(volatile atomic_int_t *a)
{
    reg_t tmp = 0, tmp2;

    asm volatile("1:                                \n\t"
                 "ldrex    %[tmp], [%[atomic]]        \n\t"
                 "add      %[tmp], %[tmp], #1         \n\t"
                 "strex    %[tmp2], %[tmp], [%[atomic]] \n\t"
                 "tst      %[tmp2], #1                \n\t"
                 "bne      1b                          \n\t"
                 : [tmp] "=&r" (tmp), [tmp2] "=&r" (tmp2)
                 , [clobber] "=m" (*a)
                 : [atomic] "r" (a)
                 );

    return tmp != 0;
}
```

Quizz!

What does this do?

```
asm volatile(""::"memory");
```

Part VIII

Focus on x86

Early events and CPU development

1971 Intel 4004: world's first microprocessor

1978 Intel 8086 & 8088: first x86 CPUs

1981 Intel 80186

1982 Intel 80286: memory management and protection

Successful CPU development

- 1985 Intel 80386: 32 bits, paged MMU
- 1989 Intel 80486: pipeline, on-chip cache, FPU
- 1993 Intel PentiumTM: superscalar, MMX, 64-bit external databus
- 1995 Intel Pentium Pro: ooo, on-package L2 cache
- 1997 Intel Pentium II: slot-based module
- 1999 Intel Pentium III: SSE, CPUID
- 1999 AMD Athlon: ooo, 3dnow

Technology barriers

- 2000 Intel Pentium 4: race to high frequencies
- 2001 Intel Itanium: IA-64
- 2003 AMD Athlon 64: AMD64
- 2003 Intel Pentium M: back to P6 (PPro to PIII)
- 2006 Intel Pentium 4 Prescott 2M: almost the end
- 2006 Intel Core/Core2
- 2009 Intel Core i5/i7

x86 architecture

The x86 architecture is:

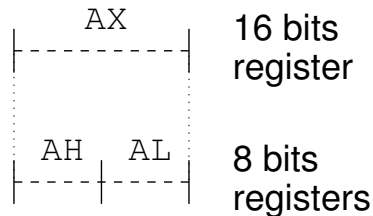
- ▶ based on a CISC instruction set.
- ▶ based on little-endian memory access.
- ▶ based on two operands instructions including memory access.
- ▶ backwards compatible: all new x86 processors are fully compatible with their predecessors.
- ▶ very complex: Pentium IV series had +30-stage pipelines.
- ▶ newer processors have less stages

Available registers

First x86 generation had a few 16-bits registers available:

- ▶ General purpose 16 bits registers:
 - ▶ ax, bx, cx, dx, si, di
- ▶ General purpose 8 bits registers:
 - ▶ al, bl, cl, dl
 - ▶ ah, bh, ch, dh
- ▶ Stack and frame registers:
 - ▶ sp, bp
- ▶ Flag registers, ...

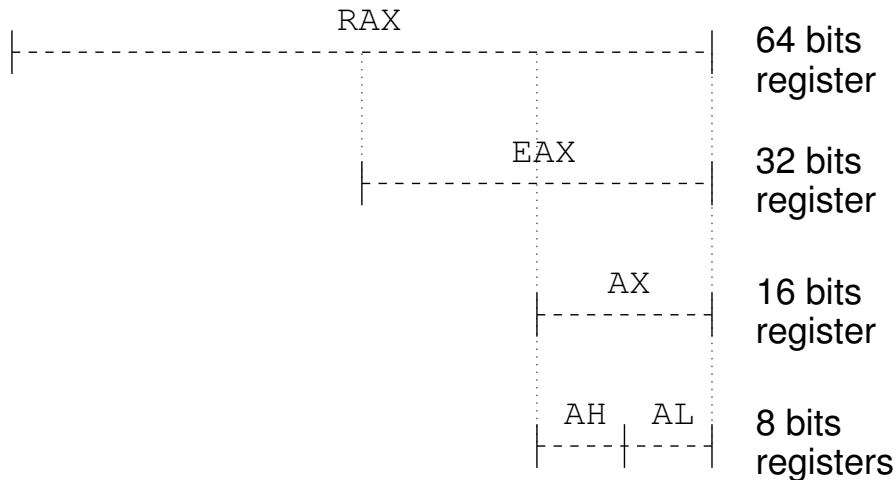
Nested registers



Register extensions

- ▶ Starting with the 386 processor, all registers are now 32-bits wide.
 - ▶ Due to compatibility issue, 16 bits register are still present.
 - ▶ `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `esp`, `ebp` were added.
 - ▶ Even if registers size is increased, registers count remained the same: only 6 registers are available for operations.
- ▶ For amd64, AMD extended the register count to 16, only available with 64 bits mode enabled
 - ▶ `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `rsp`, `rbp`,
 - ▶ `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, `r15`
- ▶ AMD licensed amd64 to Intel after the Itanium flop...

32/64 bits nested registers



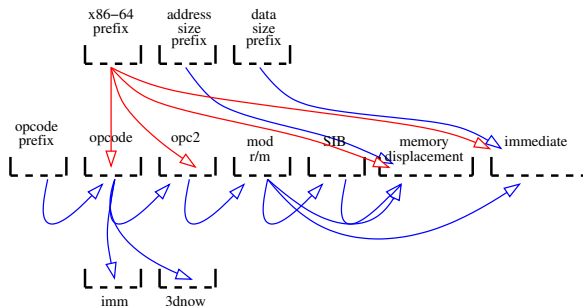
Instruction format

x86 architecture is a CISC based architecture:

- ▶ All instructions are of variable length,
- ▶ The length of an instruction cannot be determined before reading its first bytes,
- ▶ Many instructions with different formats are available.

Instruction format

x86-64



Addressing mode

The x86 architecture supports several complex addressing modes. On 32-bit processors (386 and later), a memory address can contain:

- ▶ A base address register
- ▶ A address displacement
- ▶ An index register
- ▶ An multiply factor on index register

Example: `mov eax, [ebx + 0x12345 + ecx * 8]`

Wired stack management

Specific instructions are available for stack management:

- ▶ `push x` instruction can be used to store data on the top of the stack and decrement the stack pointer.
- ▶ `pop x` instruction removes data from the stack.
- ▶ `call` and `ret` instructions directly push and pop return address on and from the stack.

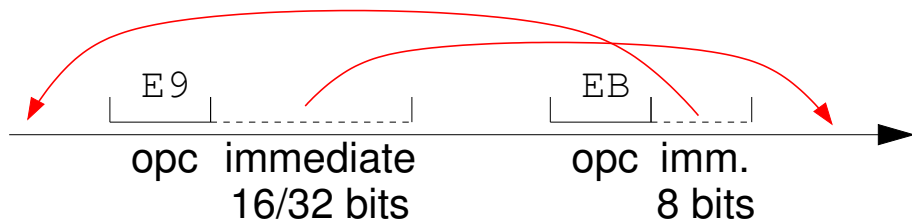
The `(e)sp` register is used as stack pointer.

Branch

- ▶ The x86 architecture uses a flag register to manage conditional branches.
- ▶ Many different instructions with different lengths can be used depending on jump size.
- ▶ No delayed slot are used.

Jump size

long jump
(-32768 to +32767) short jump
(-128 to +127)



Instruction set

x86 instruction set is huge:

- ▶ Over 580 instructions handled by *Pentium 3* CPU
- ▶ Over 850 opcodes handled by *Pentium 3* CPU

A single instruction can be very complex:

- ▶ Access memory,
- ▶ Handle different data widths,
- ▶ Perform complex computation,
- ▶ Etc.

Instruction set extensions

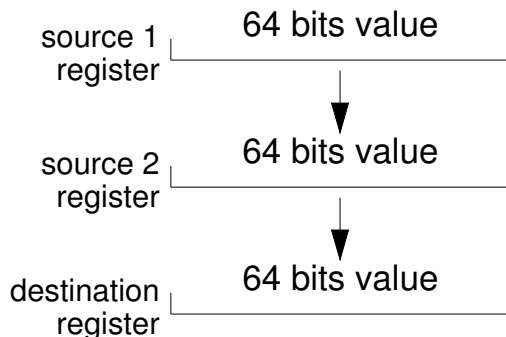
Extensions to the x86 instruction set are common and all instructions are not available on all CPUs. Starting with the pentium processor, SIMD-based instruction sets appeared:

- ▶ MMX
- ▶ 3dNow!
- ▶ MMX2, SSE
- ▶ 3dNow2!, SSE2, SSE3, SSE4s
- ▶ To be continued...

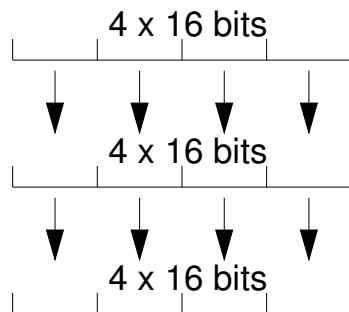
Instruction set

SIMD operations

normal operation



SIMD operation



Instruction set

has-been parts

- ▶ aaa, aad, aam, aas, daa, das
- ▶ xlat

x86-specific coding

Due to its amazing number of available instructions, x86 code is highly tunable for optimizations:

- ▶ High level languages compilers are often not able to use all instructions efficiently,
- ▶ Hand written assembly is often faster,
- ▶ Complex instructions can be used to process unexpected tasks.

x86-specific coding

example: LEA

The LEA instruction is designed to compute memory addresses. But it can be used to add and multiply values.

```
lea eax, [ebx + ecx]
```

```
lea eax, [ebx * 8 + ebx]
```

x86-specific coding

example: tiniest mem*()?

`memcpy` `rep movsb`

`memset` `rep stosb`

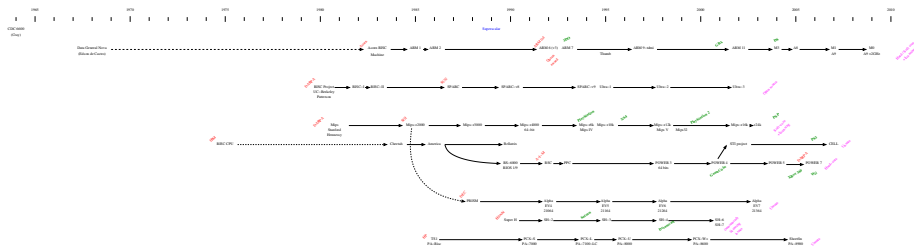
`memcmp` `repe cmpsb`

`strncmp` `repnz cmpsb`

Part IX

Focus on RISC processors

Let's see a timeline...



Mips

Instruction formats and features

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

J/JAL	offset																												
OP	rs	rt	imm																										
Special	rs	rt	rd	...	op																								

- ▶ 32 general-purpose registers
- ▶ hard-wired r0 to 0
- ▶ 32 FPU registers, all sizes aliased
- ▶ delay slot

Mips

Asm code

```
addiu r2, r3, 0x42    ; alu reg / imm
or     r3, r4, r5      ; alu reg / reg

lui    r2, 0x1234      ; load upper immediate
ori    r2, r2, 0x5678  ; r2 = 0x12345678

lw     r4, 0x1234(r9)  ; load word from r9 + 0x1234

slt    r1, r2, r3      ; test if r2 < r3
bnez   r1, 2f          ; jump if true
nop                               ; delay slot

lbu    r2, (r3)         ; load byte, not sign extended

jal    foo              ; pc = foo; r31 = return address
nop
```

SPARC

Instruction formats

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

01	offset																												
----	--------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

00	rd				op2	imm																							
00	a	cond				op2	imm																						

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1x	rd	op3	rs1	0	asi	rs2
1x	rd	op3	rs1	1	imm	
1x	rd	op3	rs1	opf		rs2

SPARC

Instruction features

- ▶ 32 general-purpose registers
- ▶ hard-wired %g0 to 0
- ▶ register window
- ▶ unwindowed aliased FPU registers, 8 * 128 bits, 32 * 32 bits
- ▶ delay slot

SPARC

Asm code

```
add    %g3, 0x42, %g2      ; alu reg / imm
or     %g3, %g4, %g5       ; alu reg / reg

sethi  0x12345, %g2        ; load upper immediate (20 bits)
or     %g2, 0x678, %g2     ; g2 = 0x12345678

ld      [%l2 + 0x1234], %g4 ; load word from l2 + 0x1234
ld      [%l2 + %i3], %g4   ; load word from l2 + i3

tst     %l2, %l3           ; test if l2 < l3
blt     3f                ; jump if true
```

PowerPC

Instruction formats and features

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

op	jump offset										aa	lk	
op	rd	ra	imm										
op	rd	ra	rb	mb	me			rc					
op	rd	ra	rb	op2								rc	

- ▶ 32 general-purpose registers
- ▶ 32 FPR, fixed internal representation
- ▶ additional registers for `lrr`, `ctr`
- ▶ no global flags, but 8 “cause registers” cr_x
- ▶ can merge causes with logical operations

PowerPC

Asm code

```
addi  3, 2, 42      ; alu reg / imm
or.    3, 4, 5       ; alu reg / reg, update cr0

lis    3, 0x1234     ; load upper immediate (16 bits)
ori    3, 3, 0x5678  ; r3 = 0x12345678

lwz    4, 9, 0x1234  ; load word from r9 + 0x1234
lwzxr  4, 9, 3       ; load word from r9 + r3

mtctr  4             ; put r4 in count register
...
bdnzl  2f           ; Decrement CTR, Branch if CTR != 0

rlwinm ...          ; Rotate and mask
```

ARM

Instruction features

- ▶ 16 GPR, one of them is pc (r15)
- ▶ Every instruction is “guarded” by a condition. It may be:
 - ▶ always, >, >=, overflow, negative, carry, ==
 - ▶ their opposites
- ▶ aliased FPR: 16 double, 32 float

You can prefix any instruction with “never”:

- ▶ 1/16th of the instruction set is “nop”...

ARM

Instruction formats

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

cond	00	i	op				s	rn	rd	imm/reg			
cond	01	i	p	u	b	w	l	rn	rd	imm/reg			
cond	100	p		u	b	w	l	rn	reg-list				
cond	101	l	jmp offset										
cond	11	coproc, sys											

ARM

Instruction formats (2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

cond	00	i	op				s	rn	rd	imm/reg				
cond	01	i	p	u	b	w		rn	rd	imm/reg				
		1								rotate	imm			
		0								rs	0	ty	1	rm
		0								amount	ty	0	rm	

ARM

Instruction formats (2)

```
and r3, r7, #42      ; r3 = r7 & 0x2a
add r2, r8, r5, lsl r1 ; r2 = r8 + (r5 << r1)
sub r1, r9, r1, lsl #1 ; r1 = r9 - (r1 << 1)
```

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1110	00	1	and	0	r7	r3	« 0	0x2a							
1110	00	0	add	0	r8	r2	r1	0	lsl	1	r5				
1110	00	0	sub	0	r9	r1	0x1		lsl	0	r1				

ARM

Asm code

```
add    r3, r2, #0x42      ; alu reg / imm
orr     r3, r4, r5         ; alu reg / reg
bic     r3, r4, r5, lsr #4 ; alu reg / reg & shift

cmp     r2, #0             ; test if r2 < 0
rsblt   r2, r2, #0         ; then r2 = 0 - r2

ldr     r2, #0x12345678    ; rewritten as
ldr     r2, [pc, #offset]  ; pc-relative load

streq   r4, [r2, r3, lsl #4] ; store word to r2 + r3 << 4
                                   ; only if last cmp is 'eq'
str     r4, [r2, #8]!      ; store word to r2 + 8
                                   ; and r2 = r2 + 8

; 32-bit immediates zone
offset:
0x12345678
```

ARM

Block transfers

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

cond	100	p	u	b	w		rn	reg-list
------	-----	---	---	---	---	--	----	----------

`stmdb sp!, {r2, r3, r4, r8}`

`push {r2, r3, r4, r8}`

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

cond	100	1	0	0	1	0	r13	0000000100011100
------	-----	---	---	---	---	---	-----	------------------

ARM

Block transfers hacks (1)

The link register is r14, then the following code saves r14 and restores it to PC afterwards...

```
push  {r2, r3, r4, r8, lr}  
...  
pop   {r2, r3, r4, r8, pc}
```

..00		
..04	sp →	r2
..08		r3
..0c		r4
..10		r8
..14		lr
..18	old sp →	xxx
..1c		
..20		

ARM

Block transfers hacks (2)

Do a memcpy with some free registers, for instance, copy 16 bytes from *r0 to *r1, not using r2 nor r5, update r0 and r1 to point on next block...

```
ldmia  r0!, {r3, r4, r6, r7}  
stmia  r1!, {r3, r4, r6, r7}
```

ARM Instruction-space exhaustion

Over time ARM instruction-set continued to grow despite the obvious exhaustion of the “clean” instruction-set space. It led to:

- ▶ the excess of concatenation of isolated bits in the instruction
- ▶ sometimes forbid r15 as an operand, and reuse other fields
- ▶ reuse the “never” condition code

ARM Instruction-space exhaustion

Illustration

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

cond	00	i	op	s	rn	rd	op2				
		1					rotate	imm			
		0					rs	0	ty	1	rm
		0					amount	ty	0	rm	

How to add the multiplication?

cond	00	0	000	a	s	rn	rd	rs	1001	rm
------	----	---	-----	---	---	----	----	----	------	----

Thumb mode

One goal: Code compression.

Concessions to pack a maximum of operations in a minimal space:

- ▶ Access to only 8 registers among 16
- ▶ Implicit stack ops
- ▶ Implicit pc-relative operations
- ▶ No predicates any more

Dirty hacks for making it work:

- ▶ Use lower bit of r15 as a mode indicator
- ▶ Use a new `bx/blx` instruction

Thumb2

Keep up with thumb, this is great.

Ideas:

- ▶ Have an asm-code compatibility with ARM
- ▶ Remap the whole ARM 32-bit instruction set in 16-bit thumb
- ▶ But 16-bit instruction words are not wide enough any more...

Never mind

- ▶ Keep the basic 16-bit thumb encoding
- ▶ When we need more, just make the instruction 32-bits long...
- ▶ Decode mixed 16/32-bits instructions
- ▶ Only 16-bit alignment constraint for 32-bit long thumb-2 ops

Then Thumb-2 is a RISC with a CISC encoding!

Part X

CPU-aware optimizations

Purpose

By knowing the low-level implementation of processors, we can:

- ▶ write code easier for the compiler to translate
- ▶ write code easier for the CPU to run
 - ▶ because nicer to the pipeline
 - ▶ because nicer to the memory subsystem

Nicer with the pipeline

example: `abs()`

The absolute value is a good example of optimized code which is easy to implement in an efficient and smart way.

Nicer to the pipeline

abs() basic code

```
int abs(int x)
{
    if ( x < 0 )
        return -x;
    else
        return x;
}
```

```
int abs(int x)
{
    return (x < 0) ? -x : x;
}
```

Nicer to the pipeline

example: `abs()`

Let's go back to some mathematical principles:

- ▶ Addition: $a + 1 = a - (-1)$
- ▶ Number representation: $-1 = 0xffffffff$
- ▶ Negation: $-a = (\text{not } a) + 1$
- ▶ $\text{not } a = a \text{ xor } 0xffffffff = a \text{ xor } -1$
- ▶ if $(x < 0)$, return $-x((\text{not } x) + 1)((x \wedge 0xffffffff) + 1)$
 $((x \wedge 0xffffffff) - 0xffffffff)$
- ▶ if $(x > 0)$, return $x((x \wedge 0) +/- 0)$

How to produce -1 when $x < 0$?

- ▶ Sign extension: $(x \gg 31)$

Nicer to the pipeline

abs() better code

```
int abs(int x)
{
    int sign_word = x >> 31;
    return (x ^ sign_word) - sign_word;
}
```

```
int abs(int x)
{
    int sign_word = -(1 & (x >> 31));
    return (x ^ sign_word) - sign_word;
}
```


Sign extension at an arbitrary size

Sometimes, we need to sign extend a value from an arbitrary word width (say 13 bits) to a CPU word (say 32 bits).

How to do it?

```

x 00000000000000000000snnnnnnnnnnnnnn
high                                10000000000000
-high 1111111111111111111110000000000000
result sssssssssssssssssssssnnnnnnnnnnnnnn

```

```

int sign_ext(int val, int bits)
{
    int high = 1 << (bits-1);
    return (val & high) ? (val | (-high)) : val;
}

```

Sign extension at an arbitrary size

× 00000000000000000000snnnnnnnnnnnnnn
« snnnnnnnnnnnnnnn00000000000000000000
» **ssssssssssssssssssssssss**snnnnnnnnnnnnnn

```
int sign_ext(int val, int bits)
{
    int shift_bits = 32 - bits;
    return (val << shift_bits) >> shift_bits;
}
```

Sign extension at an arbitrary size

```

x 00000000000000000000snnnnnnnnnnnnnn
sb 00000000000000000000s000000000000
xor 00000000000000000000Snnnnnnnnnnnnnn

```

```

x 000000000000000000001nnnnnnnnnnnnnn
sb 0000000000000000000010000000000000
x ^ sb 000000000000000000000nnnnnnnnnnnnnn
x ^ sb - sb 111111111111111111111nnnnnnnnnnnnnn

```

```

x 000000000000000000000nnnnnnnnnnnnnn
sb 0000000000000000000010000000000000
x ^ sb 000000000000000000001nnnnnnnnnnnnnn
x ^ sb - sb 000000000000000000000nnnnnnnnnnnnnn

```

Sign extension at an arbitrary size

```
int sign_ext(int val, int bits)
{
    int high = 1 << (bits-1);
    return (val ^ high) - high;
}
```

Power of two

Properties

- ▶ Powers of two have only one “1” bit
- ▶ Subtracting 1 from a power of two flips the only “1”

Examples:

- ▶ $11110010 - 1 = 11110001$
- ▶ $0100000 - 1 = 0011111$
- ▶ Lower bits up to the lowest 1 get flipped
- ▶ Other bits stay the same

```
int is_pow2(int n)
{
    return !(n & (n-1));
}
```

Mask merging

7 6 5 4 3 2 1 0

where_0	1	1	1	0	1	0	0	0
where_1	1	0	0	0	1	1	1	0
mask	0	0	1	1	1	1	0	0
result	1	1	0	0	1	1	0	0

```
int mask_merge(int mask, int where_0, int where_1)
{
    return (mask & where_1) | (~mask & where_0);
}
```

Mask merging

Less instructions

	7	6	5	4	3	2	1	0
where_0	1	1	1	0	1	0	0	0
where_1	1	0	0	0	1	1	1	0
$\text{where_0} \wedge \text{where_1}$	0	1	1	0	0	1	1	0
mask	0	0	1	1	1	1	0	0
$(\text{w0} \wedge \text{w1}) \& \text{mask}$	0	0	1	0	0	1	0	0
$((\text{w0} \wedge \text{w1}) \& \text{mask}) \wedge \text{w0}$	1	1	0	0	1	1	0	0

```
int mask_merge(int mask, int where_0, int where_1)
{
    return ((where_0 ^ where_1) & mask) ^ where_0;
}
```

Fast string operation

strlen()

The typical "slow" implementation:

```
size_t strlen(const char *str)
{
    size_t l = 0;
    while (*str++)
        l++;
    return l;
}
```


Fast string operation

strlen()

Faster theoretical implementation:

```
size_t strlen(const char *str)
{
    size_t l = 0;

    /* ... align str on a long word boundary ... */

    /* scan 4 bytes at each iteration on a 32-bit processor */
    const unsigned long *lstr = str;
    while (!has_a_zero_byte(*lstr))
        l += 4;
        lstr++;
}

/* ... find the exact position of the zero byte
   in the matching word ... */

return l;
}
```

Zero byte detection

Detect a zero byte:

$0x00 - 0x01$ $0xFF$

$0x41 - 0x01$ $0x40$

Subtracting 1 to a zero byte will set its high bit.

$0xE9 - 0x01$ $0xE8$

But it also matches bytes greater than $0x80$!

We need to rule out bytes greater than $0x80$, which already have their high bit set:

$\sim 0x00 \ \& \ 0x80$ $0x80$

$\sim 0xE9 \ \& \ 0x80$ $0x00$

Zero byte detection

has_a_zero_byte() implementation

```
#define has_a_zero_byte(x)  
    (((x) - 0x01010101UL) & (~(x) & 0x80808080UL))
```

Example:

x = "ël\0^?"	0x7F006CEB
x - 0x01010101	0x7DFF6BEA
<hr/>	
~x	0x80FF9314
~x & 0x80808080	0x80808000
<hr/>	
result	0x00800000

Code readability

If you ever code with this kind of hack:

- ▶ **always** create functions with explicit name and prototype
- ▶ document the **intended behavior**
- ▶ may use a `static inline`

```
int abs(int x);  
int sign_ext(int val, int bits);  
int is_pow2(int n);  
int mask_merge(int mask, int where_0, int where_1);  
int has_a_zero_byte(unsigned long x);
```

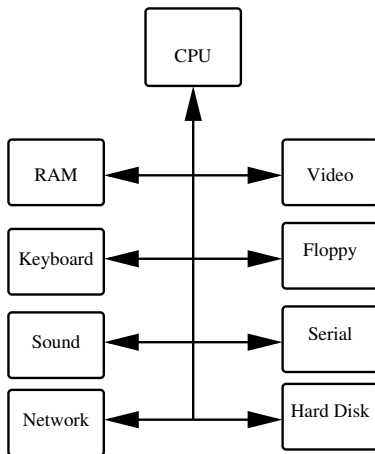
The reference for bit operations:

<http://graphics.stanford.edu/~seander/bithacks.html>

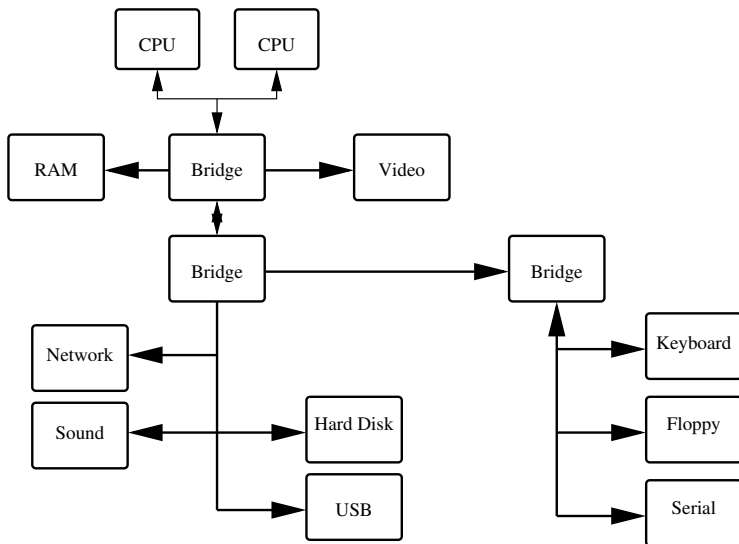
Part XI

Multi-/Many-core and heterogeneous systems

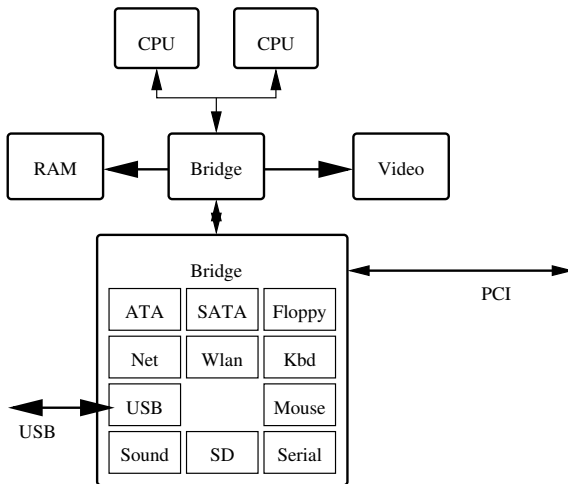
History of hardware topologies



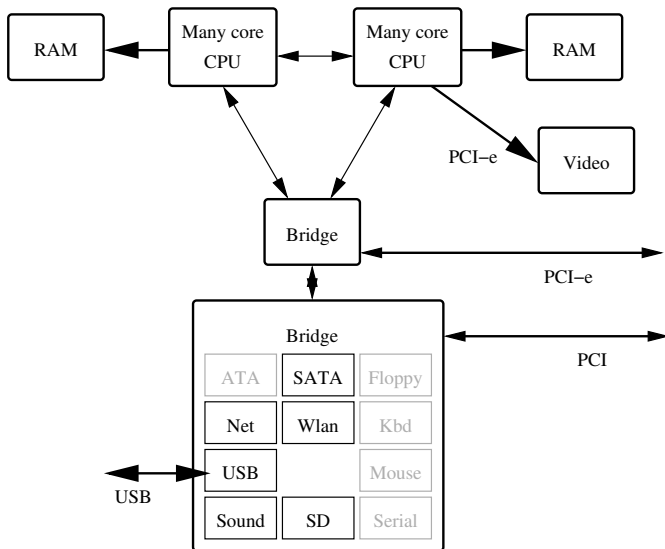
History of hardware topologies



History of hardware topologies



History of hardware topologies



Hardware topologies

Observations

Buses don't scale:

- ▶ Bandwidth is shared among connected peers
- ▶ They need rest cycles between elections

Buses are not used anymore, especially for inter-processor connection:

- ▶ We use networks (QPI, HyperTransport)
- ▶ Unfortunately, such approach forbids snooping
- ▶ Coherence has thus to be done explicitly

NUMA

Observations

- ▶ There are more and more cores
- ▶ Memory gets closer to the cores
- ▶ Memory connections get distributed among cores
- ▶ Systems thus become NUMA (Non-Uniform Memory Access)

Some scalability bottlenecks

Coherent shared-memory systems:

- ▶ are hard to design
- ▶ do not scale well
- ▶ get slower with the load

NUMA systems:

- ▶ are hard to program
- ▶ are not well supported by all OS

Non-coherent shared-memory systems are not ready for prime-time yet.

Why On-Chip Cache Coherence is Here to Stay, by Milo M. K. Martin, Mark D. Hill and Daniel J. Sorin, 2012.