

Assembly & System programming

Nicolas Pouillon



2012

License

- ▶ Copyright © 2004-2005, ACU, Benoit Perrot
- ▶ Copyright © 2004-2008, Alexandre Becoulet
- ▶ Copyright © 2009-2012, Nicolas Pouillon

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.

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

How to “talk” with the machine directly ?

- ▶ Involved mechanisms
- ▶ Assembly language structure and usage
- ▶ Low-level assembly language features
- ▶ C inline assembly

Course outline

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

Part II

Assembly programming

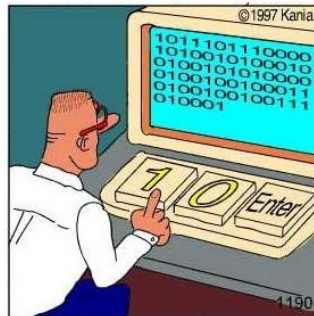
What is assembly ?

Assembly *is not*

- ▶ binary data
- ▶ directly interpreted by the processor
- ▶ don't mix it up with *machine code*

Assembly *is*

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



Real programmers code in binary.

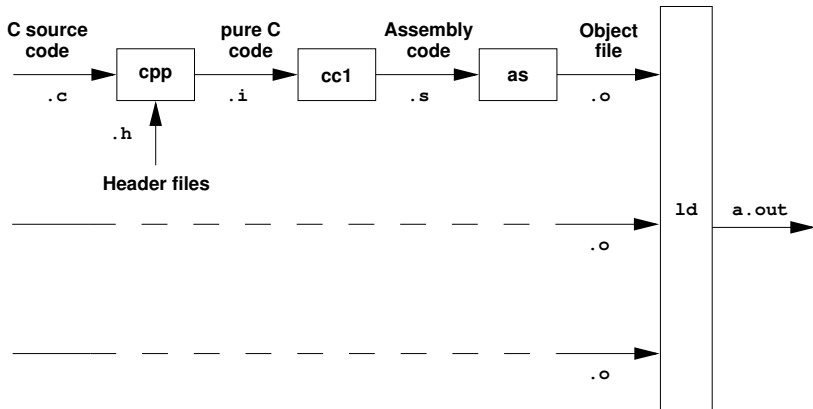
Benefits of assembly programming

- ▶ Improve 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

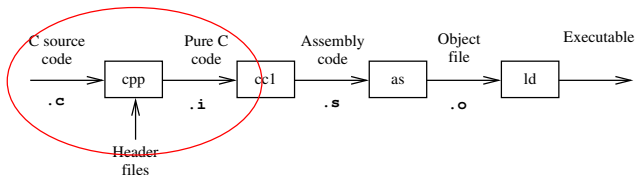
Drawbacks of assembly programming

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

The big picture



Preprocessor

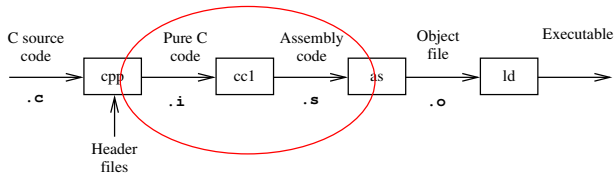


- ▶ Also called macro-processor
- ▶ Transforms a text (source) file into another text (source) file:
 - ▶ merges many files into one (`#include`)
 - ▶ replaces macros by their definitions (`#define`)
 - ▶ removes code considering conditions (`#if*`)

Example: `cpp`

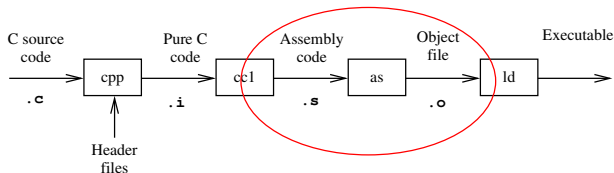
- ▶ Input: C source file with directives
- ▶ Output: “pure” C source file

C compiler



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

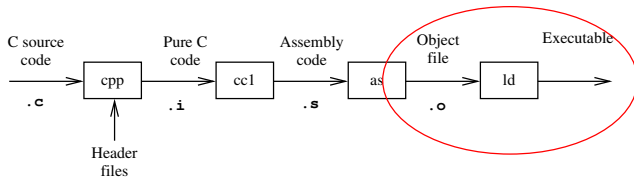
Assembler



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

The assembler source file will be different depending on architecture !

Linker



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

Assembly file organization

Source file is organized in different sections:

code

Contains program binary opcodes

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

```
1  .mod_load    asm-sparc
2  .define F00 5
3
4  .section code .text
5      .mod_asm opcodes v8
6
7      add %g0, F00, %g1
8  .ends
```

Assembly language statements

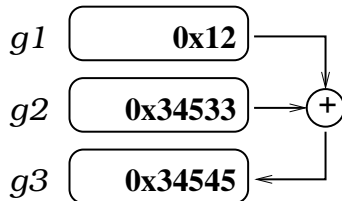
Example

```
.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

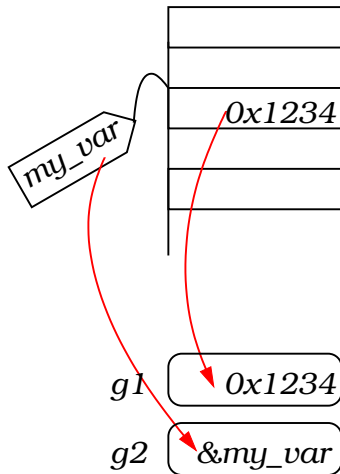
Example

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

.section data .data
my_var:
    .reserve 4
.ends

.section code .text
    .mod_asm opcodes v8

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



Assembly language statements

Example

```
1  .mod_load asm-sparc
2  .include sparc/v8.def
3
4  .section code .text
5      .mod_asm opcodes v8
6
7      .export main
8      .proc main
9          save %sp, -96, %sp
10         ; ...
11         ret
12         restore
13     .endp
14 .ends
```

Assembly language statements

Example

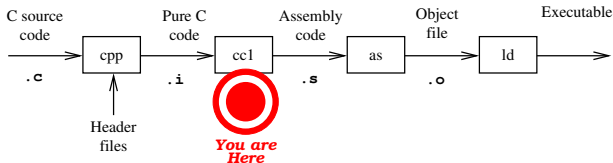
```
1  .mod_load asm-sparc
2  .include sparc/v8.def
3
4  .section code .text
5      .mod_asm opcodes v8
6
7      .extern exit
8
9      .proc my_exit
10         call exit
11         nop
12
13         retl
14         nop
15     .endp
16 .ends
```

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 assembly you pass in 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 assembly you pass is meant to either

- ▶ compute a value
- ▶ change some hardware feature
- ▶ have a side-effect

Example

No data

```
1 void disable_interrupts(void)
2 {
3     asm volatile("cli");
4 }
```

Example

Raw string passed to assembler

```
1 void lorem(void)
2 {
3     asm volatile("lorem ipsum dolor sit amet");
4 }
```

Example

Output only

```
1  static inline
2  uint64_t cpu_cycle_count(void)
3  {
4      uint32_t      low, high;
5
6      asm("rdtsc" : "=a" (low), "=d" (high));
7
8      return (low | ((uint64_t)high << 32));
9  }
```

Example

Input only

```
1  static inline
2  void cpu_io_write_8(uintptr_t addr, uint8_t data)
3  {
4      asm volatile("outb      %0,      %1"
5                  :
6                  : "a" (data), "d" ((uint16_t)addr)
7                  );
8  }
```

Example

In-out

```
1  static inline
2  uint32_t cpu_endian_swap32(uint32_t x)
3  {
4      asm ("bswap    %0"
5          : "=r" (x)
6          : "0" (x)
7          );
8
9      return x;
10 }
```

Syntax

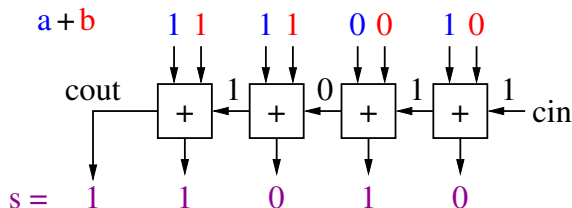
- ▶ one statement with optional arguments

```
1  asm [volatile] ("statements    \n\t"  
2                      "on many lines \n\t"  
3                      [: [output_variables]  
4                      [: [input_variables]  
5                      [: clobbered_registers]])  
6      );
```

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

Add with carry and overflow

Concept



$$s = a + b + cin$$

As ...	a	b	cin	s	cout
unlim unsigned	13	12	1	26	—
unlim signed	-3	3	~ 0	-6	—
4-bit unsigned	13	12	1	10	1
4-bit signed	-3	3	~ 0	-6	1

$$-b = \sim b + 1$$

Add with carry and overflow

ASM

```
1  uint32_t add_cv(uint32_t a, uint32_t b, uint8_t cin,
2                  uint8_t *cout, uint8_t *vout)
3  {
4      uint32_t result;
5
6      asm("        btl $0, %k5          \n"
7          "        adcl %k3, %k4        \n"
8          "        setc %b1             \n"
9          "        seto %b2             \n"
10         : "=r" (result), "=qm" (*cout), "=qm" (*vout)
11         : "r" (a), "0" (b), "r" (cin)
12         );
13
14     return result;
15 }
```

Add with carry and overflow

C

```
1  uint32_t add_cv(uint32_t a, uint32_t b, uint8_t cin,
2                  uint8_t *cout, uint8_t *vout)
3  {
4      uint64_t sum = (uint64_t)a + (uint64_t)b + (uint64_t)cin;
5      *cout = sum >> 32;
6      *vout = ( (b ^ ((uint32_t)sum)) & ~(a^b) ) >> 31;
7      return sum;
8  }
```

Named values

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

Named values

With numbers

```
1  static inline
2  bool_t cpu_atomic_inc(volatile atomic_int_t *a)
3  {
4      reg_t tmp = 0, tmp2;
5
6      asm volatile("1:                                \n\t"
7                  "ldrex    %0, [%2]                  \n\t"
8                  "add      %0, %0, #1                 \n\t"
9                  "strex    %1, %0, [%2]              \n\t"
10                 "tst      %1, #1                     \n\t"
11                 "bne      1b                          \n\t"
12                 : "=&r" (tmp), "=&r" (tmp2)
13                 : "r" (a)
14                 : "m" (*a)
15                 );
16
17     return tmp != 0;
18 }
```

Named values

Named

```
1  static inline
2  bool_t cpu_atomic_inc(volatile atomic_int_t *a)
3  {
4      reg_t tmp = 0, tmp2;
5
6      asm volatile("1:                                \n\t"
7                  "ldrex    %[tmp], [%[atomic]]         \n\t"
8                  "add      %[tmp], %[tmp], #1          \n\t"
9                  "strex    %[tmp2], %[tmp], [%[atomic]] \n\t"
10                 "tst      %[tmp2], #1                  \n\t"
11                 "bne      1b                            \n\t"
12                 : [tmp] "=&r" (tmp), [tmp2] "=&r" (tmp2)
13                 : [atomic] "r" (a)
14                 : "m" (*a)
15                 );
16
17     return tmp != 0;
18 }
```

Quizz !

What does this do ?

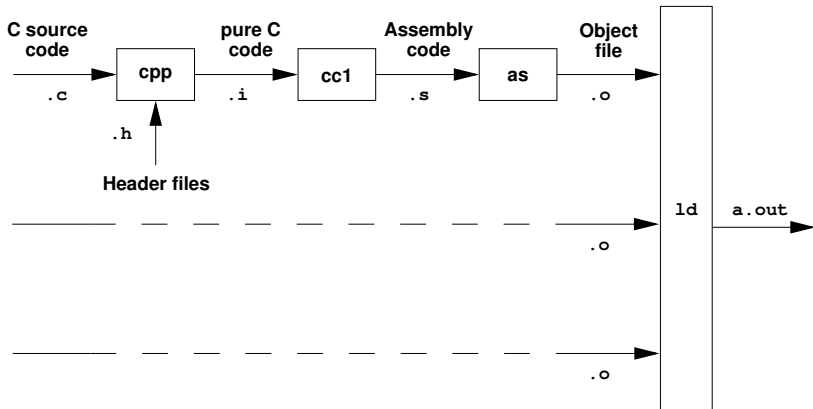
```
1  asm volatile(""::"memory");
```

Part III

Object file formats

The big picture

again



Development tools

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

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
 - ▶ -dx: disassemble

readelf

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

Simple binary formats

Consider an object program that does not need other object programs to build a binary program:

- ▶ the assembler collects code markers (labels)
- ▶ the assembler resolves *all* labels and replaces *all of them* in the code

Flat program

The 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 00 12 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: the processor can only read raw code and there is no binary loader.

Challenges

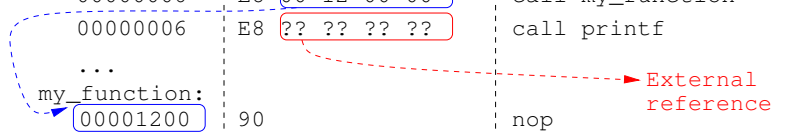
What happens when a function must be imported ?

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 00 12 00 00	<code>call my_function</code>
00000006	E8 ?? ?? ?? ??	<code>call printf</code>
...		
<code>my_function:</code>		
00001200	90	<code>nop</code>
00001201	...	



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

Needs

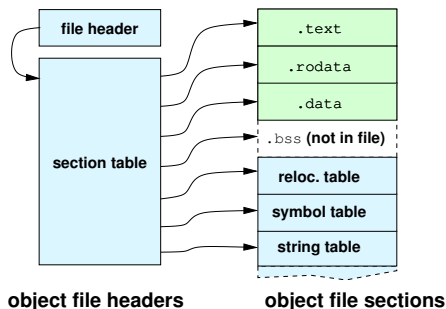
To fill the holes left by the assembler later on, the binary file must hold:

- ▶ 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 format must also hold:

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

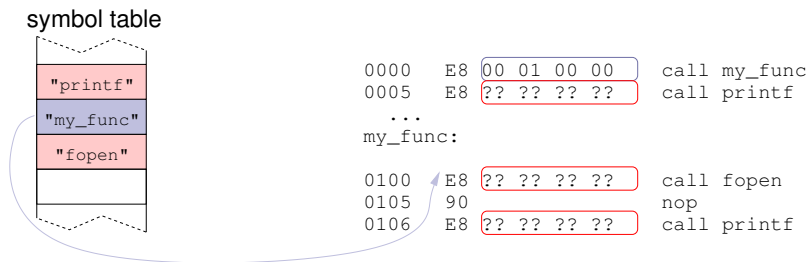
Abstraction of a binary file format



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

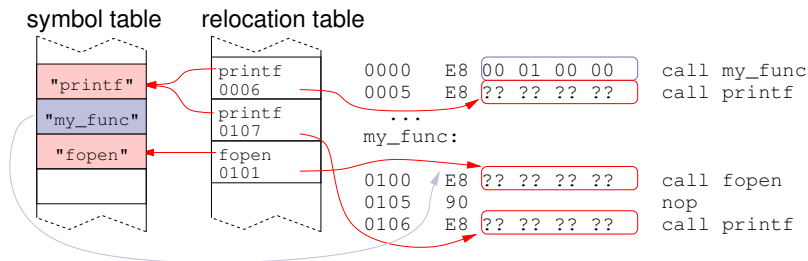
The symbol table

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



The relocation table

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



BSS regions

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.

BSS regions makes the file smaller and faster to load.

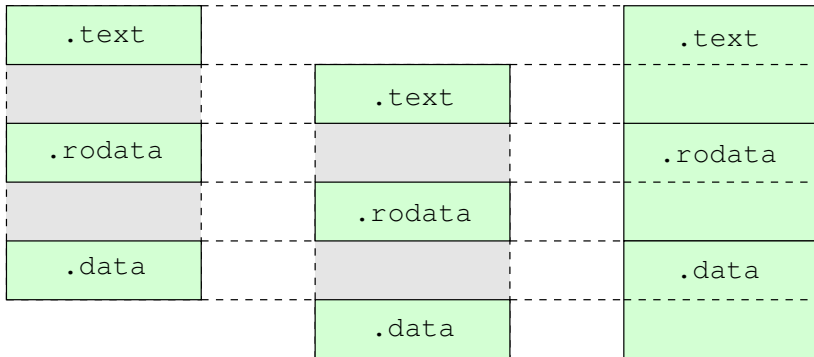
Linking

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:

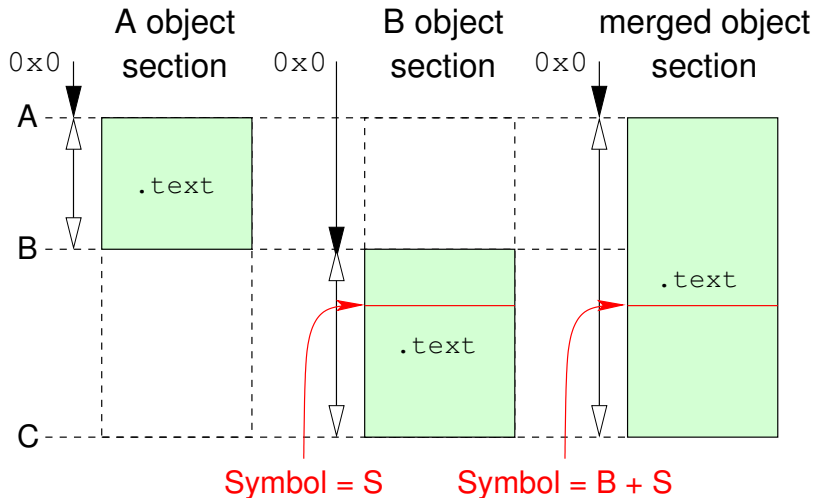
1. Named sections from different object programs must be merged together into one named section.
2. Merged sections must be put together into the sections of the memory model.
3. Each use of a name in an object program's references list must be replaced by an address in the virtual address space.

Merging object files

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



Zoom on .text section merging

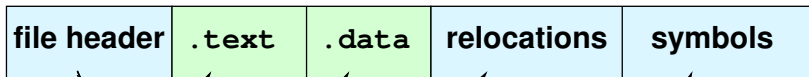


Symbol resolution

raw binary format

`.text / .data`

a.out format



Executable binary files

When there is neither unresolved symbols nor relocations anymore, the file is executable.

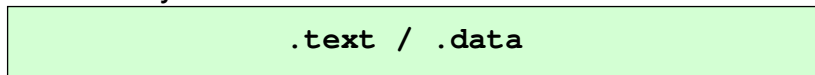
Executable files usually have a fixed constant load address on a given system.

Unresolved/unused symbols may exist in an executable file if no relocation uses it.

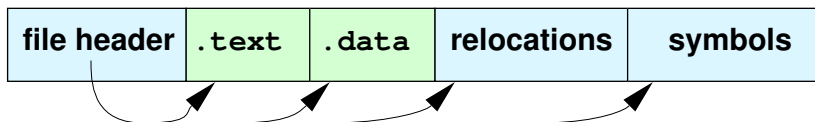
The `strip` command wipes out unused symbols from the symbol table.

a.out: Assembler Output

raw binary format



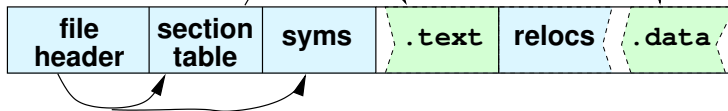
a.out format



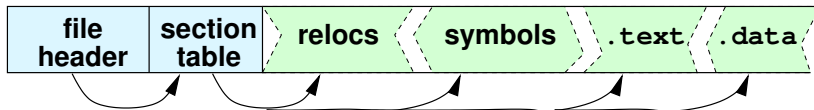
- ▶ Very simple
- ▶ .. but pretty fast to load

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

COFF format

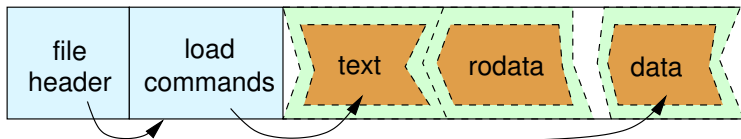


ELF format



- Permits use of dynamic libraries

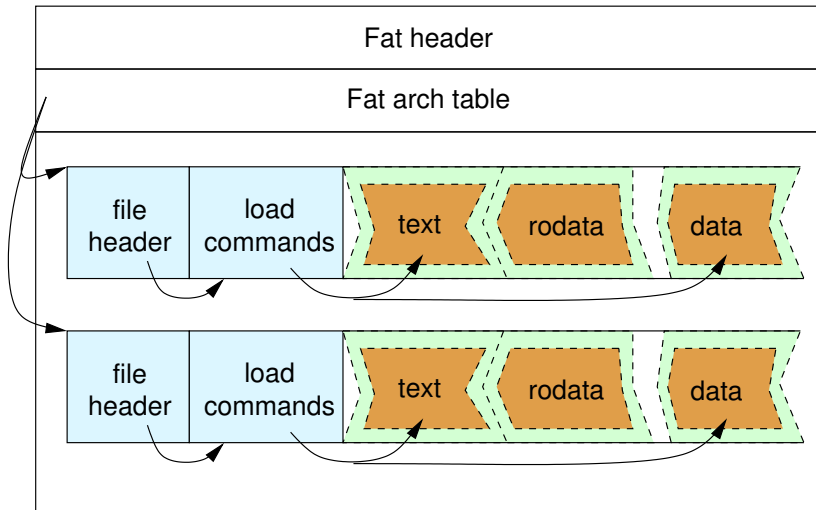
Mach-O (Darwin)



Mach-O (Darwin)

"Fat binaries"

Mach-O fat binaries

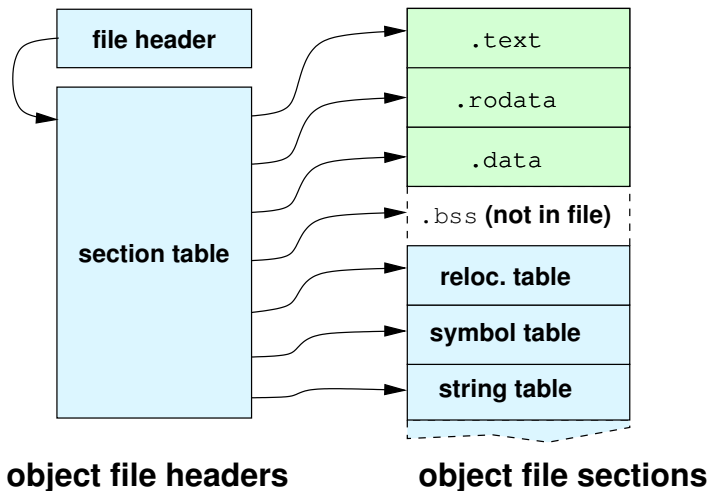


Binary file loading

Executable file format is like object file format, with the following restrictions:

- ▶ It has no relocations,
- ▶ All addresses are resolved
- ▶ It is ready to load in memory

Executable format

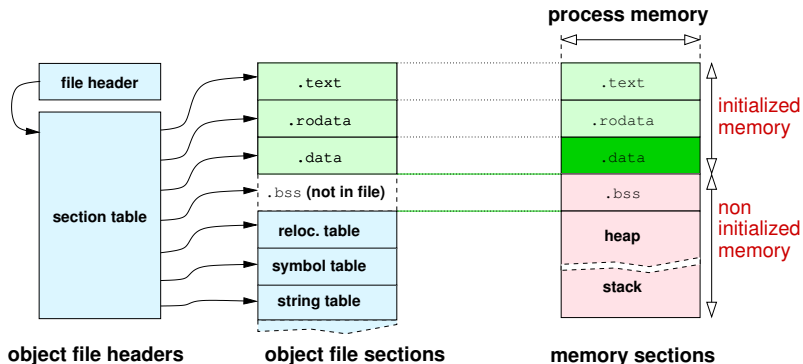


Executable loading

Process memory structure is mapped to internal executable file format:

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

Executable memory mapping

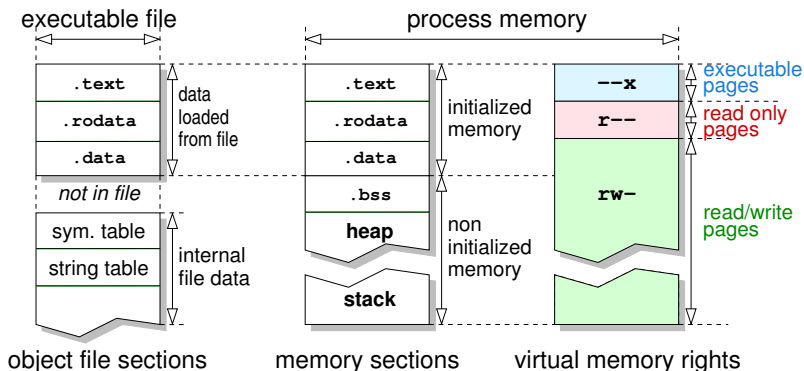


Memory attributes

Memory attributes depends on section and area type:

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

Memory attributes



Dynamic libraries

Use of dynamic libraries implies:

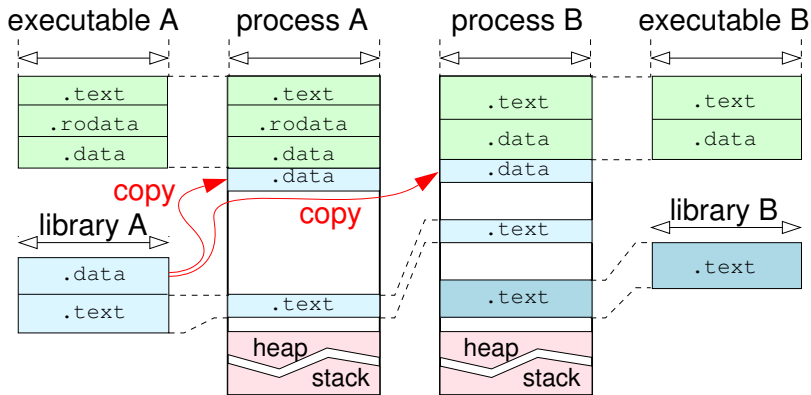
- ▶ different and more complex memory mapping
- ▶ position independent code
- ▶ different way to handle relocations
- ▶ plenty of cool complicated stuff

Dynamic libs in memory

Libraries have specific memory mapping and organisation:

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

Dynamic process memory layout



Handling code location change

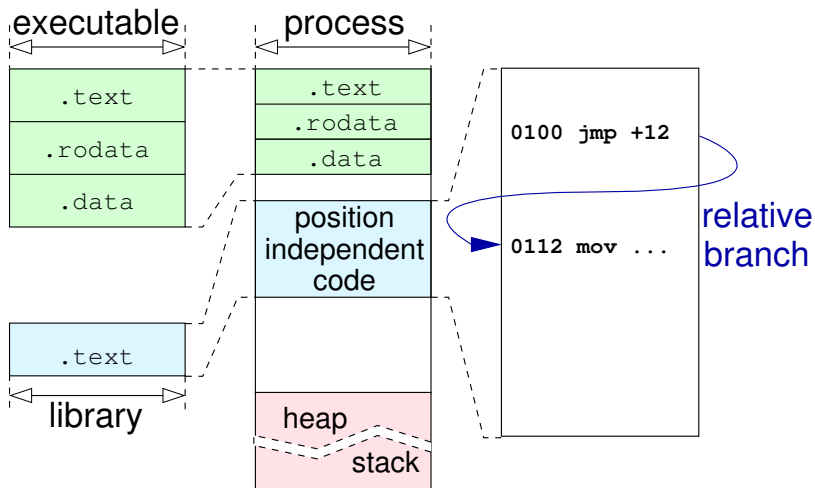
Dynamic libraries may not be mapped with the same virtual address in all processes:

- ▶ Address 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 the complex relocation process.

Relative jumps and relative data memory accesses need to be handled by the CPU.

Position independent code



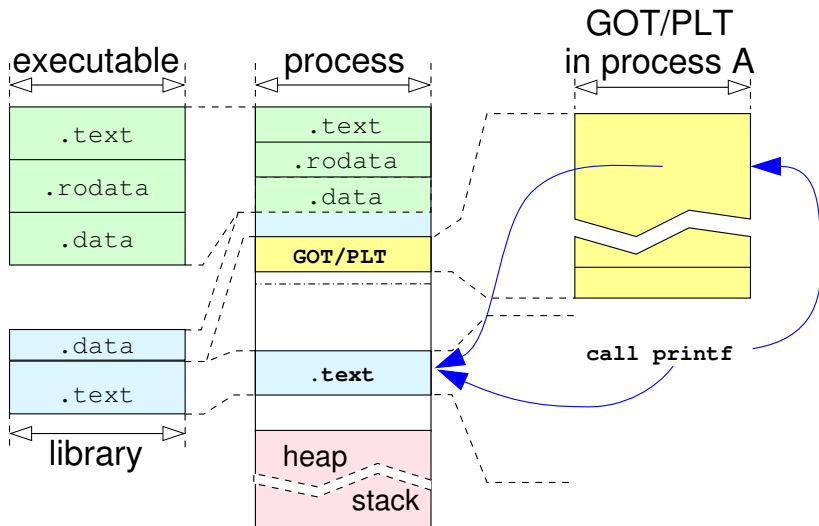
Position dependent code

Some location change are more complicated with dynamic libraries:

- ▶ `.data` can't be referred with relative addressing from `.text` section if no relative data access is available.
- ▶ `.data` section won't have the same address in all process memory maps.
- ▶ `.text` section is common to all processes and can't reflect `.data` location differences.

The solution is to use indirect memory access to reach data objects from code. One GOT (*Global Offset Table*) and PLT (*Procedure Linkage Table*) will hold data object addresses for each process. Some dynamic relocations and data copy will also help to solve this problem.

Use of GOT/PLT



Part IV

Processor architecture

What a processor is...

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

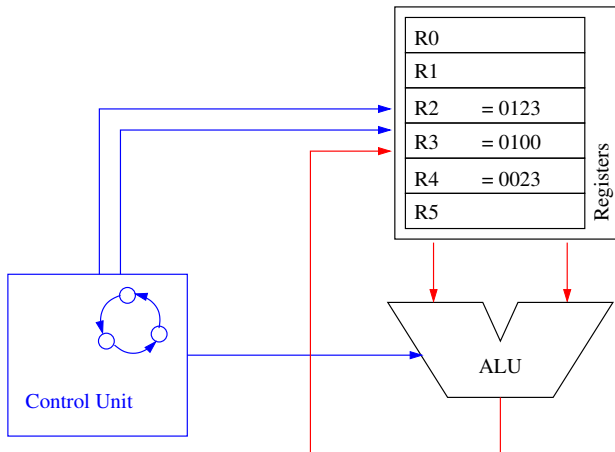
- ▶ Execute instructions
- ▶ Store results (i.e. have some memory)

It needs several basic units to perform those tasks:

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

Lets design it

Basic architecture



Basic architecture (2)

In this model, all the system state is in the processor.

- ▶ This is sufficient for a basic processor
- ▶ More needs may be leveraged by adding registers or program steps

Unfortunately

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

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

Revised processor model

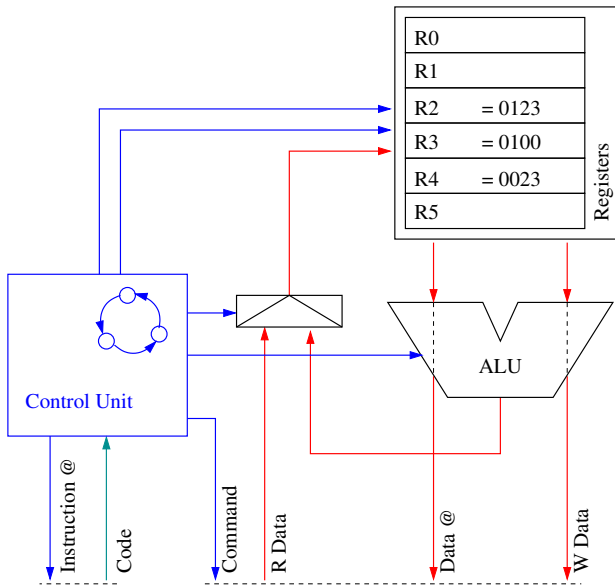
A processor must be able to perform the following basic tasks:

- ▶ 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

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

Each unit

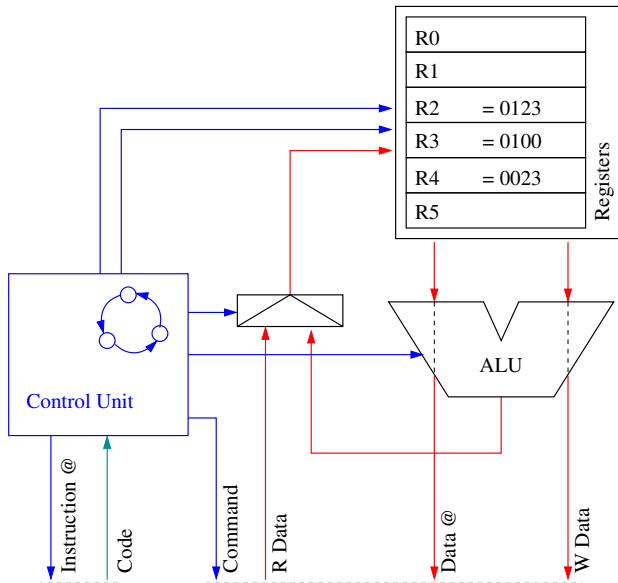
- ▶ is implemented as an hardware component
- ▶ is made of switchable parts (transistors)

In actual processors

- ▶ Units used to be independant chips
- ▶ Some were 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...

ALU: Arithmetic and Logical Unit

An unit without registers !

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

Division is not possible without registers !

Instruction types

There are different instruction types:

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

Instructions classification

The Flynn classification:

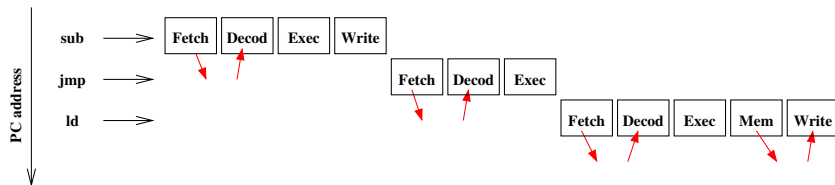
- ▶ SISD: Single Instruction, Single Data
Classical Von Neumann architecture
- ▶ SIMD: Single Instruction, Multiple Data
Vectorial computers
- ▶ MIMD: Multiple Instruction, Multiple Data
Multiprocessor computers

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 ...*).

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 process
- ▶ Most hardware is used only once for each instruction
- ▶ Most hardware is unused most of the time

Pros:

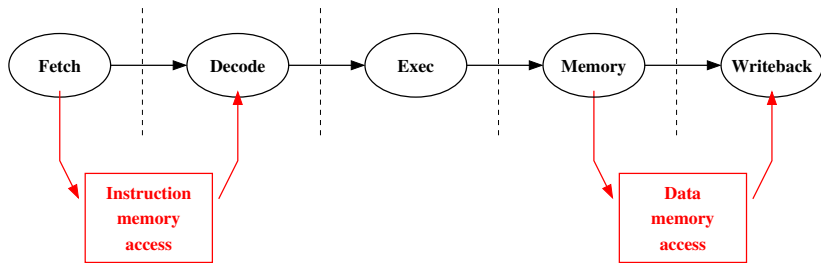
- ▶ Easy to implement
- ▶ Small
- ▶ New instructions can be added just adding new steps

Pipelined processor

Instruction flow

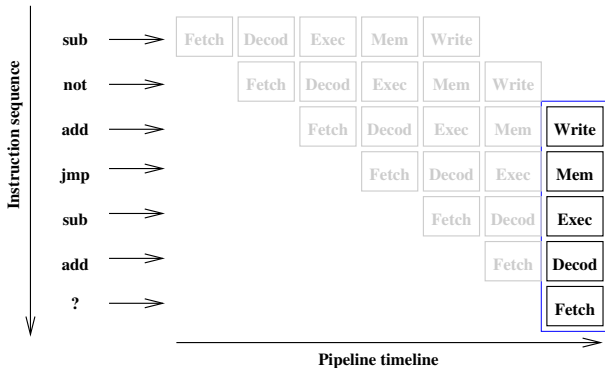
A pipeline architecture allows parallel execution of instructions:

- ▶ Split execution in many 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

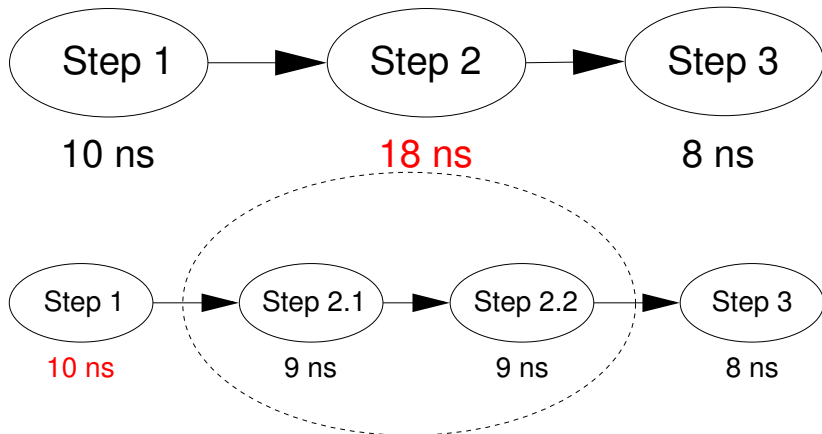
Current processors extensively use the pipeline architecture to speed processing up.

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



Speeding up the pipeline

Splitting operations in shorter steps allows 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 *CISC* based : *z80*, *Intel x86*, *Motorola 68000* ...

Recent designs are *RISC* based : *MIPS*, *Sparc*, *Alpha*, *PowerPC*, *ARM* ...

RISC

Characteristics

Pros:

- ▶ Simple instructions
- ▶ Fixed length instructions
- ▶ Simple hardware is needed to decode instructions

Cons:

- ▶ Programs are longer as they need simpler 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

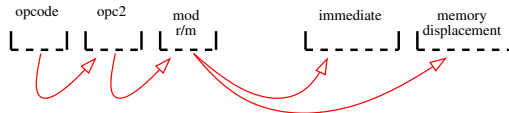
Cons:

- ▶ Binary instruction format has variable length
- ▶ Complex hardware is needed in the processor and high frequencies are harder to achieve

Modern processors internally translate the CISC code to RISC microcode

CISC

Instruction example



Part V

Memory

Reasons to access memory

How does the memory work with the processor ?

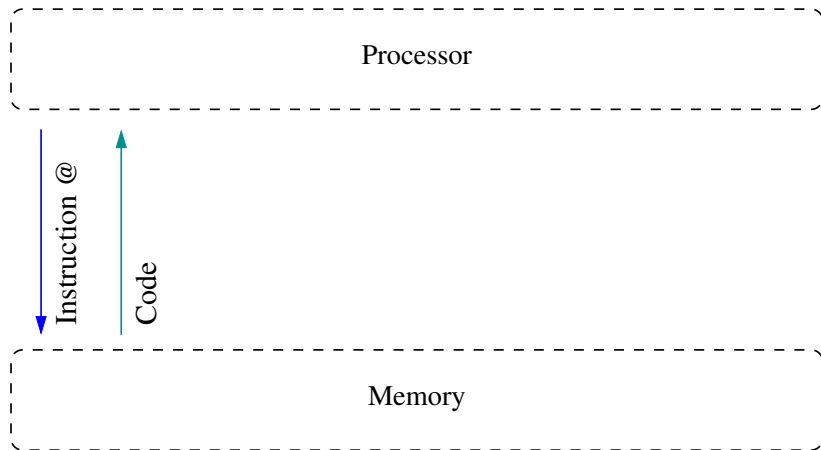
- ▶ Memory is used to fetch instructions,
- ▶ Memory is used to access data.
- ▶ There may be one unique memory, or two.
 - ▶ If there is one memory, instruction / data accesses must be sequentialized
 - ▶ If there are two, code cannot be accessed as data

Conventional names:

- 1 Von Neuman architecture
- 2 Harvard architecture

Instruction fetch

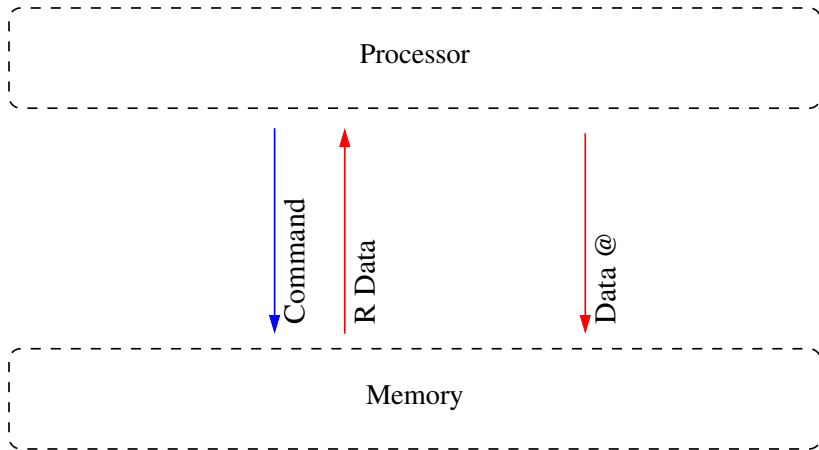
The processor needs an instruction to process



Data access

load

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



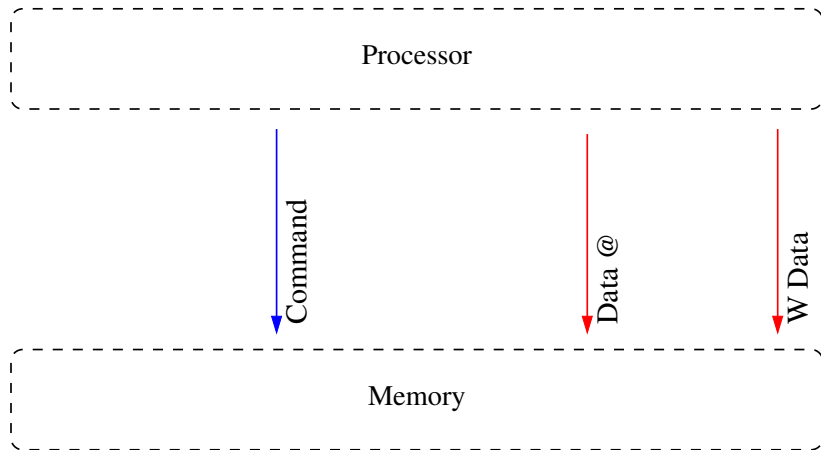
1

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

Data access

store

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



1

`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:

```
1      int a, b = ...;  
2  
3      a = b + 0x831;
```

In assembly language:

```
1      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:

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

In assembly language:

```
1      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:

```
1      int a, *b = ...;  
2  
3      a = *b;
```

In assembly language:

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

Complex addressing

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

Assembly example:

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

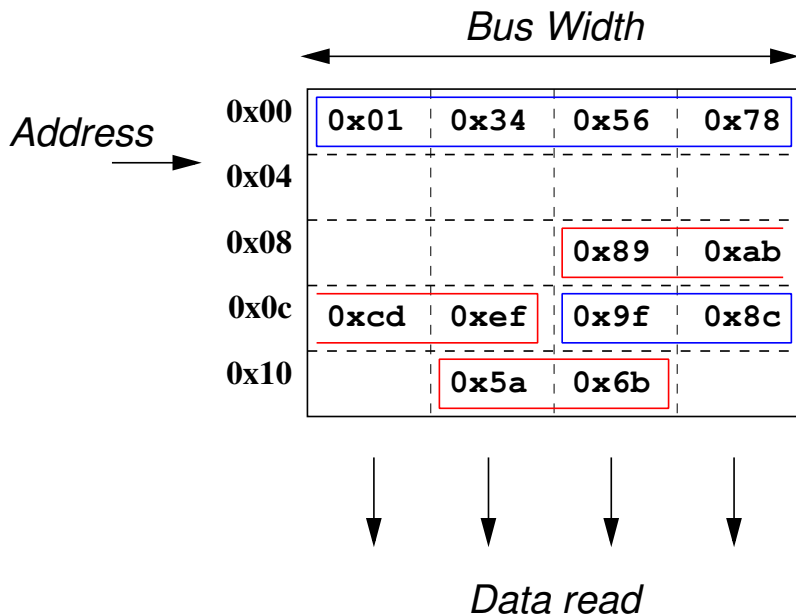
Definition

Data access alignment is solely about considered **data type width**.

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

Think about `address % sizeof(type)`

Access alignment



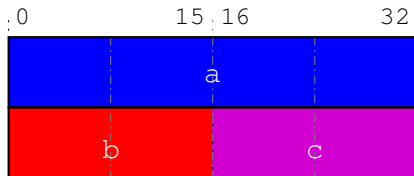
Structure alignment

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;  
};
```

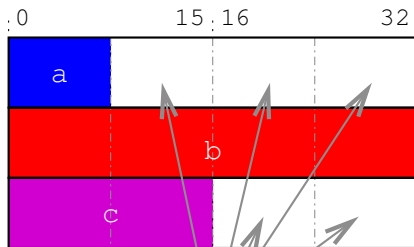


Padding

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

- ▶ Compilers put structure fields at aligned offsets
- ▶ Alignment may add unused **padding** bytes between fields

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

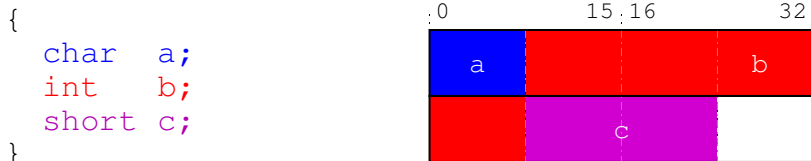


padding bytes

Basic packing

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

```
struct example_packed_s
```



```
__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;
```

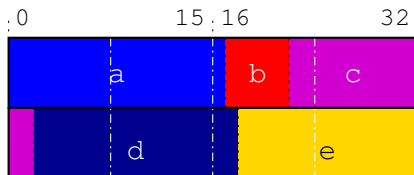
```
    int    c:12;
```

```
    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 $b = 16$: $N = 11 \cdot 16^3 + 11 \cdot 16^2 + 14 \cdot 16^1 + 7 \cdot 16^0$

So 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

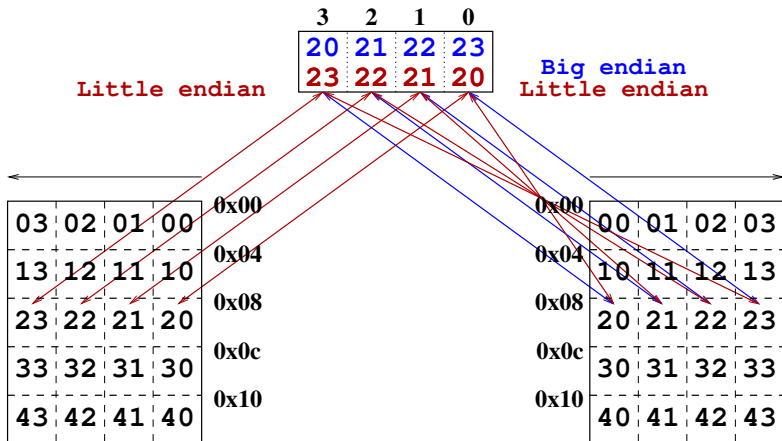
Usually, we like to represent memory in written order, the same way we write words on a paper sheet: left to right, top to bottom.

	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!

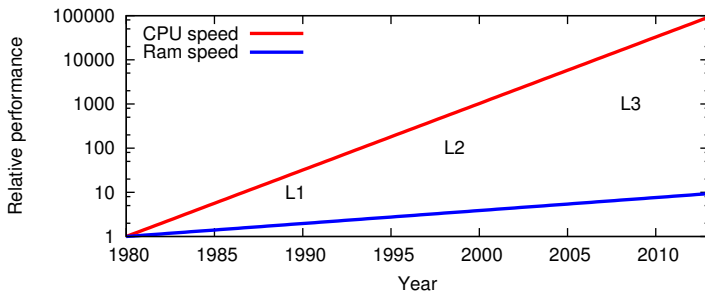
- ▶ Data must be stored and fetched using the same convention.
- ▶ Don't worry about byte order in registers, everyone agrees on this.

Endianness Demo

```
1  int main(void)
2  {
3      unsigned int a = 0x12345678;
4      hexdump(&a);
5  }
```


Cache memory: reason

- ▶ Once upon a time, CPU and memory speed were the same.
- ▶ Of course, they evolved:
 - ▶ Moore's law: CPU power doubles every 2 years,
 - ▶ Memory speed: +7% every year.



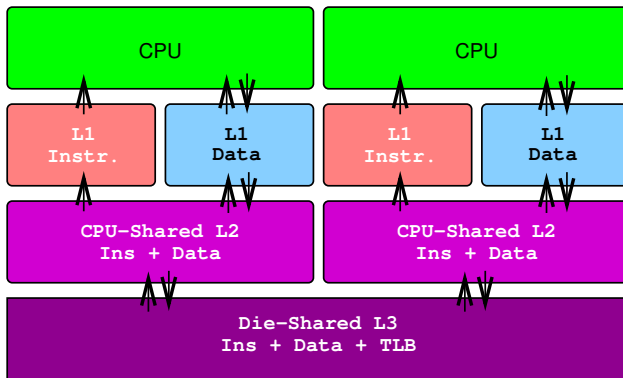
Cache memory: definition

Cache memory is:

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

Cache memory

Hierarchy



- ▶ There are multiple caches “levels”
 - ▶ with different size,
 - ▶ with different speed,
 - ▶ with different latency.
- ▶ Caches may be shared between code and data.

Cache memories side-effects

Caches may also involve strangeness:

- ▶ having copies of memory introduce a coherence problem,
- ▶ an access to a given memory location may vary,

There are various memory operators:

Programmer that you handle in C code,

Assembler that the compiler generated,

CPU that the CPU does,

In-cache that the cache does in the system.

Part VI

Memory mapping

Address space

Definition

An address space is a set of discrete values targetting a set of objects.

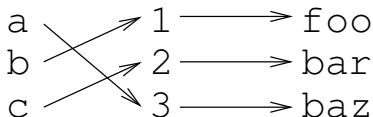
- ▶ Each address points to one object
- ▶ One given object may be pointed by more than one address

1 —————> foo
2 —————> bar
3 —————> baz

Address space translation

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

A given object can then be targetted by either addresses.



Memory address space

Definition

A memory address space

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

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

Computer address spaces

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 bits.

Virtual Address space reachable by a process. Generally 32 or 64 bits.

Physical memory

Physical memory provides the lowest accessible address space in computer. Physical RAM is usually accessible as a small memory subset of the physical address space.

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

- ▶ Accessible at a given location,
- ▶ Scattered at multiple locations,
- ▶ Accessible (many times) at multiple locations.

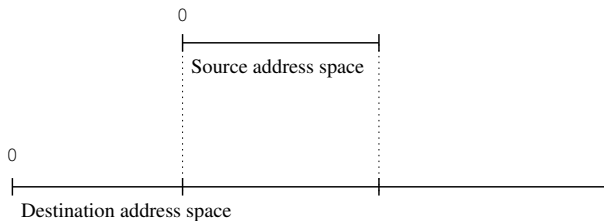
Memory segments

Memory segments define an address space as a sub-region of another address space. It is mostly defined by the following attributes

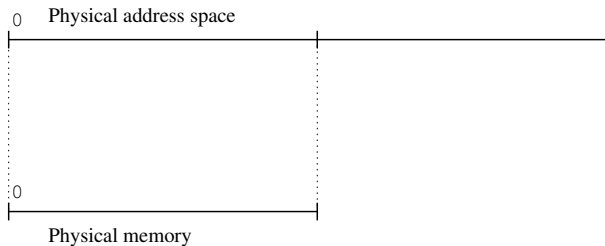
- ▶ Segment base address in target address space,
- ▶ Segment size,
- ▶ Segment type and access rights.

Simple segment

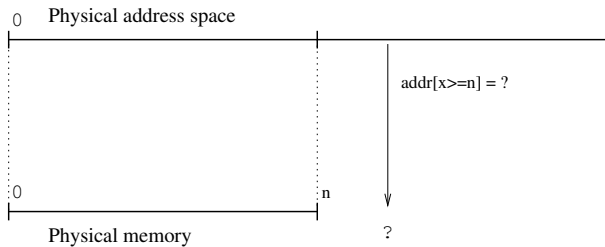
Segmentation keeps memory locations contiguous.



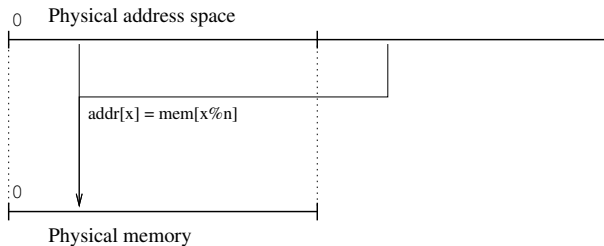
Physical address space



Single mapped RAM



Multiple/loop mapped RAM

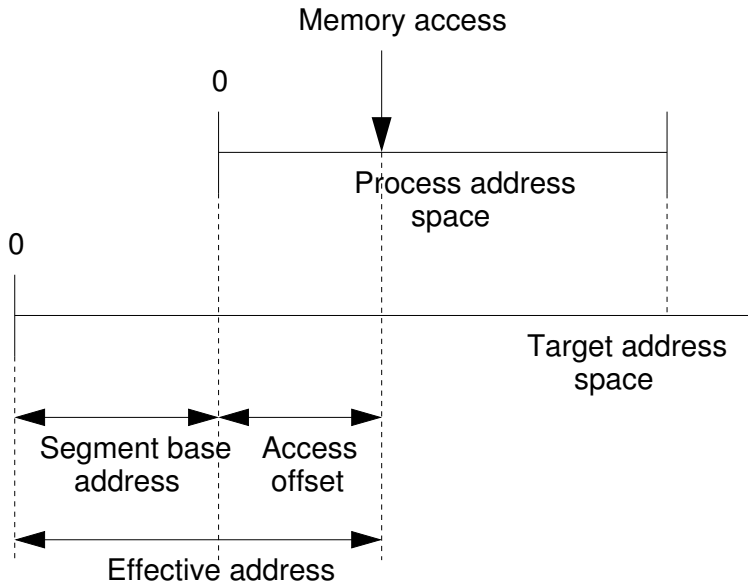


Memory access using segments

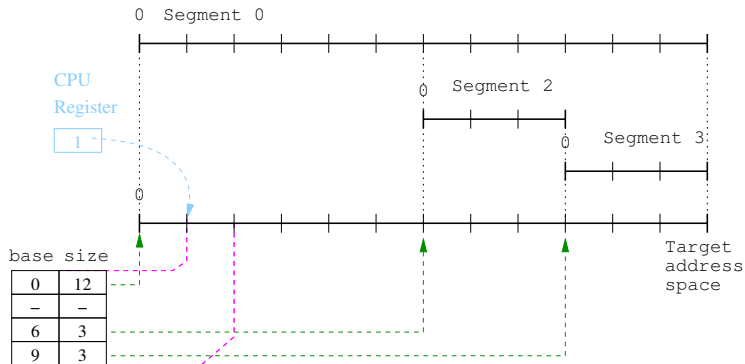
To access memory through a defined segment, the CPU performs the following tasks:

- ▶ 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



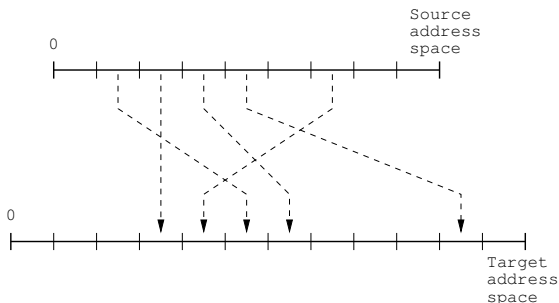
Limitations of segmentation

Segmentation is a great thing, but it has a few limitations:

- ▶ Address-space must be mapped in contiguous blocks
- ▶ Thus segments are difficult to grow on-demand
- ▶ A whole segment must be present in the target address space

Memory pages

Modern operating systems need more than segmentation.
Basic idea is to split the source address space in **pages**, and map each source page to a target page.

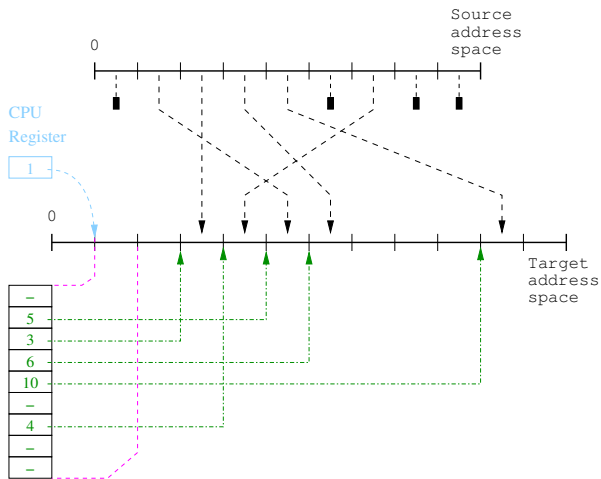


Pages mapping

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

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

Pages descriptor table



Pages mapping

Rationale

Splitting memory in pages allows more powerful memory management:

- ▶ Address spaces may be mapped to unctiguous target pages, allowing memory fragmentation.
- ▶ Many interesting operations can be performed on pages (sharing, swapping, ...)

Memory protection

Motivations

Why do we need memory protection ?

Memory protection

In order to execute secure operating system, hardware has to provide memory protection systems. This protects

- ▶ the system from the hosted processes
- ▶ hosted processes from each other

It may even protect the system from its own components in a Micro-Kernel approach.

Memory protection

How ?

Several memory access checks are performed by the CPU, for each access, transparently:

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

Memory protection

Where ?

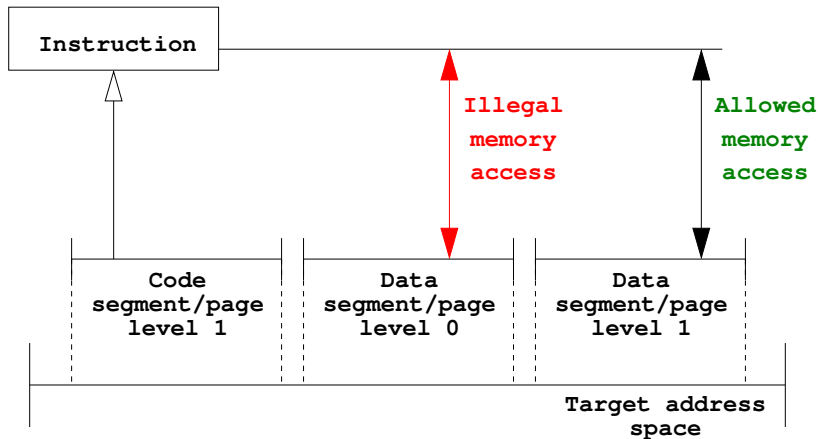
- ▶ In a segmentation-based system, privileges are per segment.
- ▶ In a pagination-based system, privileges are in page descriptor table, with a page granularity.

x86 mixes both with segmentation on top of pagination.

Privilege levels

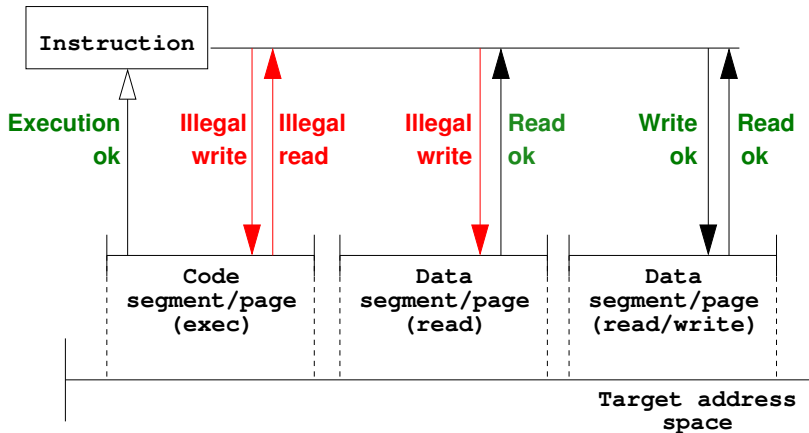
Privilege level keeps memory from being accessed by non-authorized code.

Different CPUs define different privilege levels.



Operation type

Operation types checking keeps code from doing unwanted memory operations.

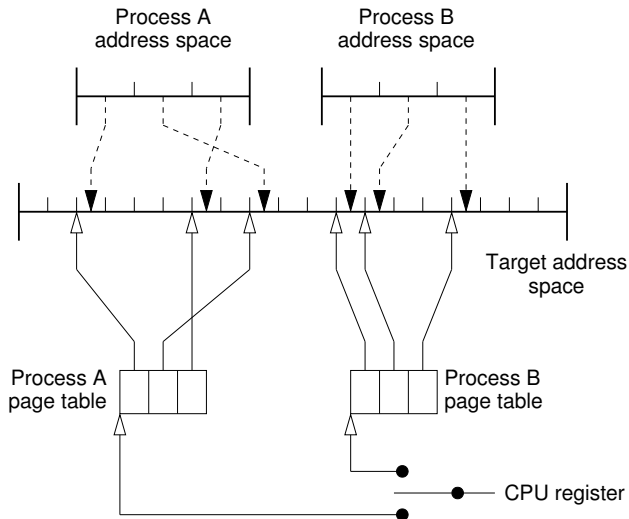


Process switching

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

- ▶ Switching process space implies 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 changed to setup a new address space.

Process switching



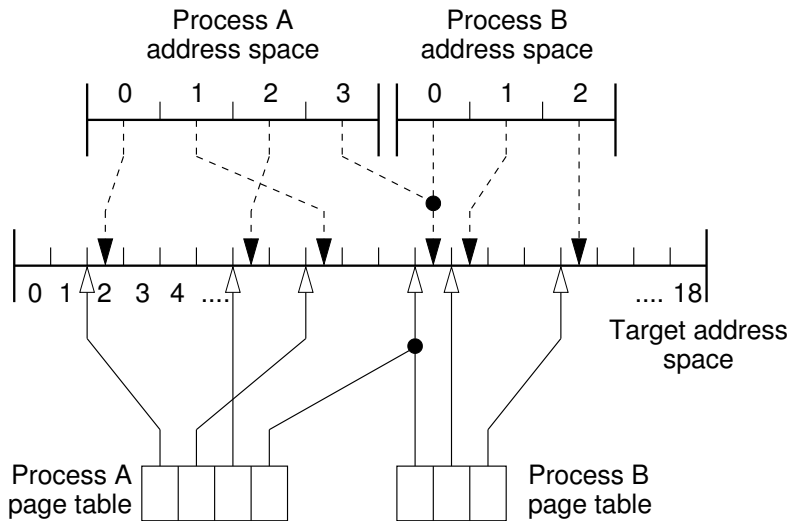
Memory sharing

The memory pagination can be used to share pages between processes.

A single page can be mapped in several process address spaces to permit different behaviors :

- ▶ Save physical memory by not duplicating shared code and read-only data memory (used for shared libraries),
- ▶ Use shared memory for inter-process communication,

Memory sharing



Copy On Write

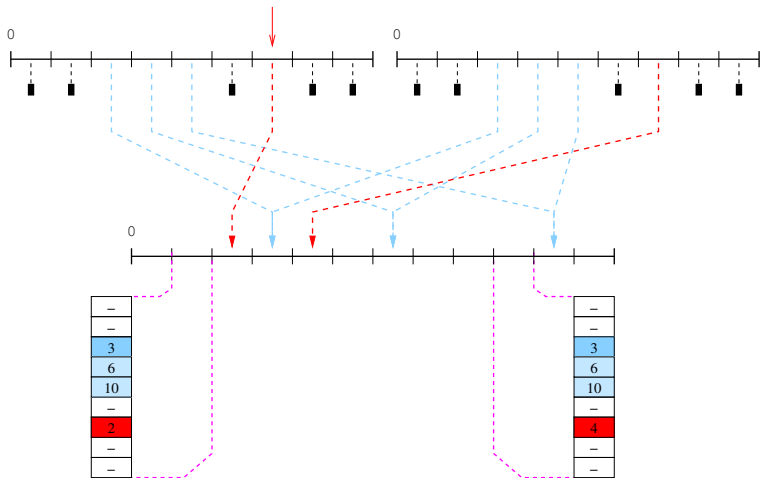
Copy On Write (COW) is a powerful trick used in many situations.

One of the most usual ones is `fork()` where the whole address space of a process has to be cloned.

Basic idea is to make the whole memory read-only and actually copy only when necessary, as late as possible.

Copy On Write

Step by step

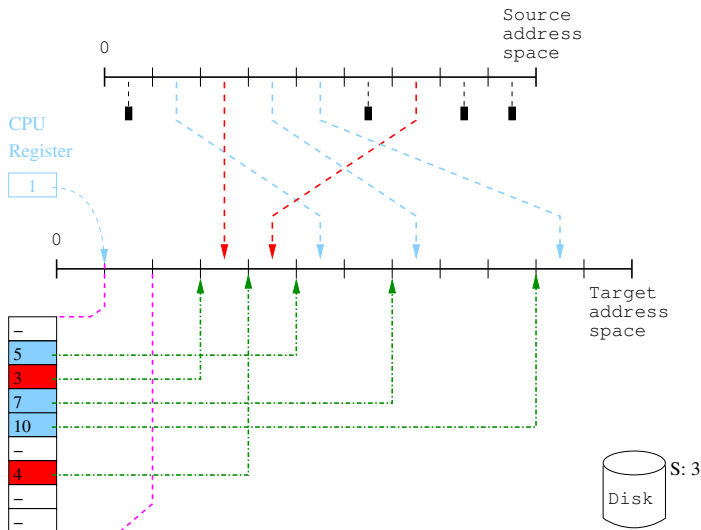


Page swapping

Page swapping is a mechanism artificially enlarging Physical memory with a part of the hard-disk.

Page swapping

Step by step



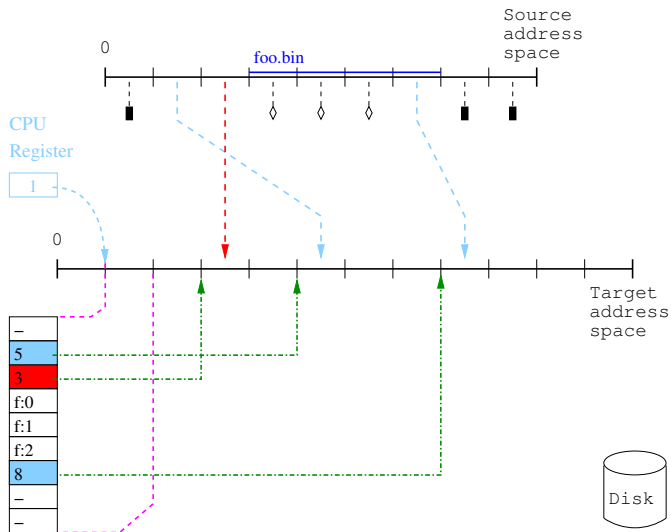
mmap()

Make part of the memory exactly match the contents of a file.

- ▶ Reflect changes immediatly between processes having file opened
- ▶ Permit different protections on different parts of the file
- ▶ Lazily load parts of file
- ▶ Lazily write parts back

mmap()

Step by step



Part VII

Execution flow

Branch principle

Branching is breaking the normal incremental execution flow to go execute code somewhere else.


A branch has

- ▶ a destination,
- ▶ optionally a condition,
- ▶ optionally a “link” feature, saving return point.

Branch offset

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

- ▶ in 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:

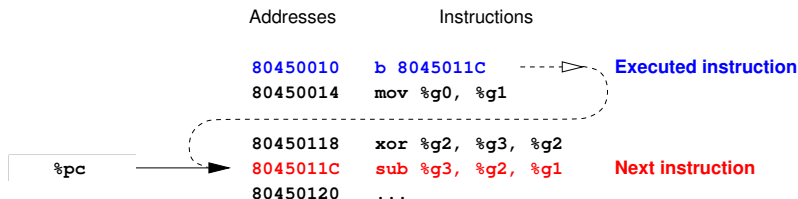
relative branch: a constant offset is added to the %pc

absolute branch: an absolute address is loaded into the %pc

Unconditional branch

The `%pc` register is always modified.

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



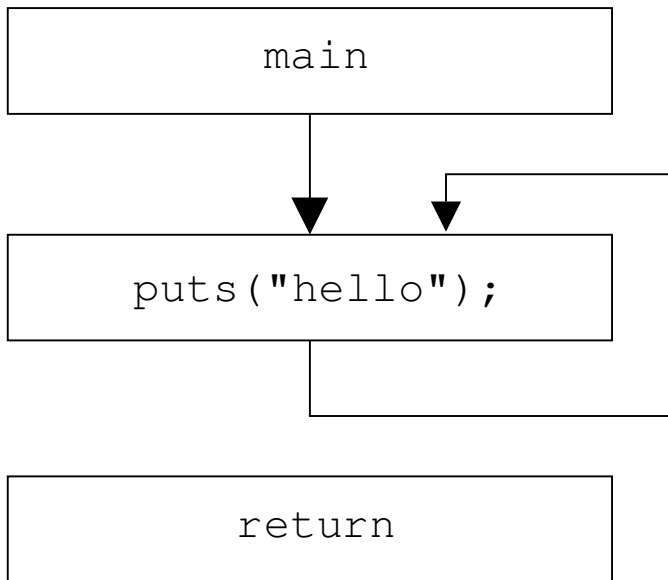
Unconditional branch

C listing

```
1  #include <stdio.h>
2
3  void main(void)
4  {
5      do {
6          puts("hello");
7      } while (1);
8  }
```

Unconditional branch

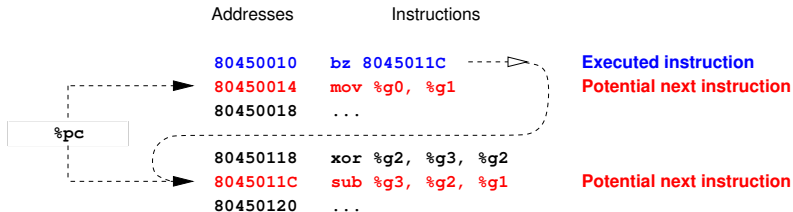
Control flow graph



Conditional branch

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

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



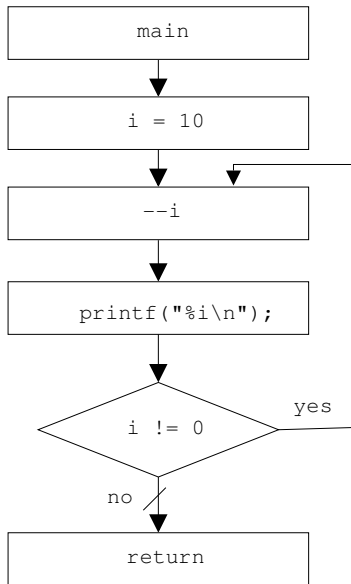
Conditional branch

C listing

```
1  #include <stdio.h>
2
3  void main(void)
4  {
5      int i = 10;
6
7      do {
8          printf("%i\n", --i);
9      } while (i != 0);
10 }
```

Conditional branch

Control flow graph



Conditions

The decision to take the branch is based on register contents:

- ▶ conditional branch may occur if a specific bit is **set** in a register
- ▶ conditional branch may occur if a specific bit is **clear** in a register
- ▶ conditional branch may occur if a register equals a specific value (usually zero)

Complete examples

1. `strlen`
2. `pgcd`

Pipeline considerations

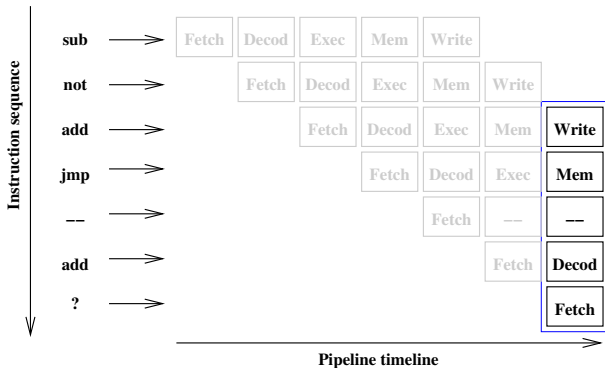
A branch may break the pipeline, slowing the execution

1. when things goes wrong, a bubble is created
2. sometimes, the processor succeeds in predicting the branch target
3. when a misprediction occurs, a bubble is created

Pipeline considerations

Bubble

When a branch occurs, the processor may flush the stages of the pipeline that contains useless instructions:

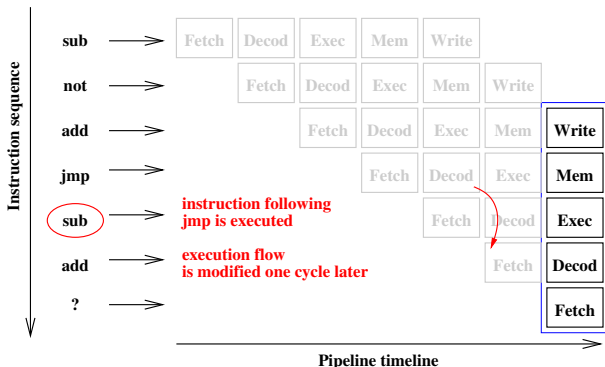


Pipeline considerations

Delay slot

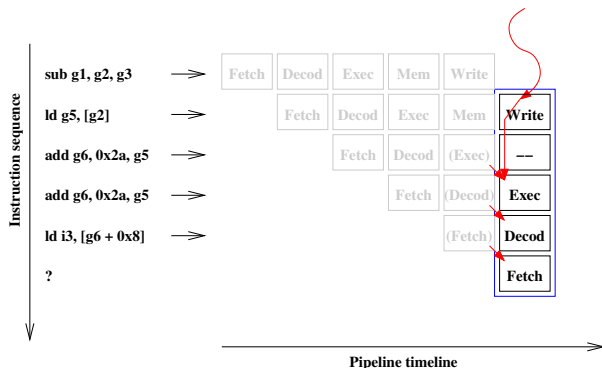
A delay slot is a processor feature. It's a convention saying the instruction following branches is always executed.

Branch is delayed.



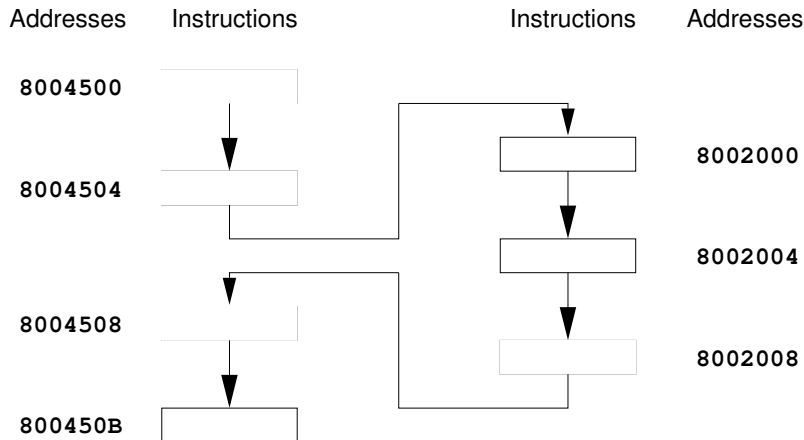
Pipeline considerations

Loads (digression)



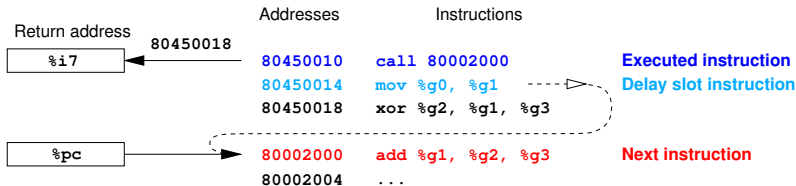
Function calls principle

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



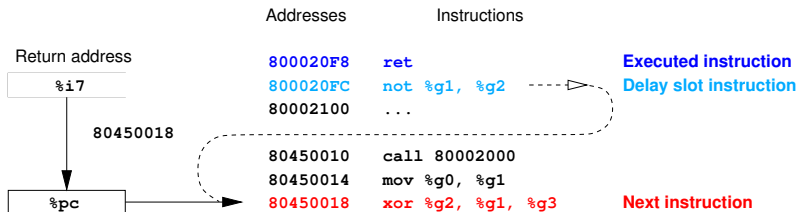
“call” instruction

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



“ret” instruction

- ▶ Stores the result in a dedicated register
- ▶ 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, for arguments
- ▶ From callee to caller, for 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 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 caller's "output" registers.

Through global memory

Principle:

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

Call conventions

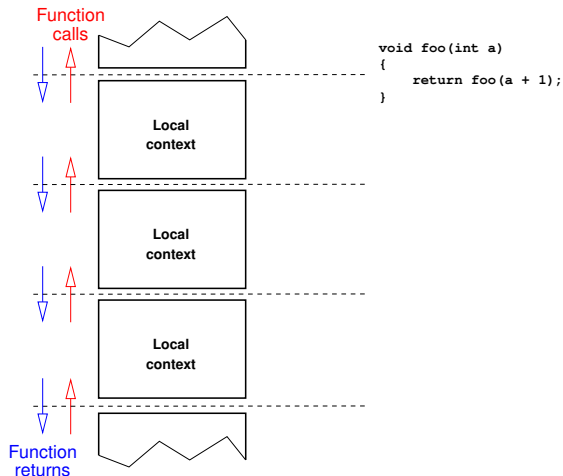
- ▶ How to deal with huge amount of variables and arguments ?
- ▶ How to deal with recursive calls and nested function calls ?
- ▶ How to deal with variables bigger than registers ?
- ▶ How to deal with “...” (like printf) ?
- ▶ How to deal with dedicated registers (floats) ?

Problem

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

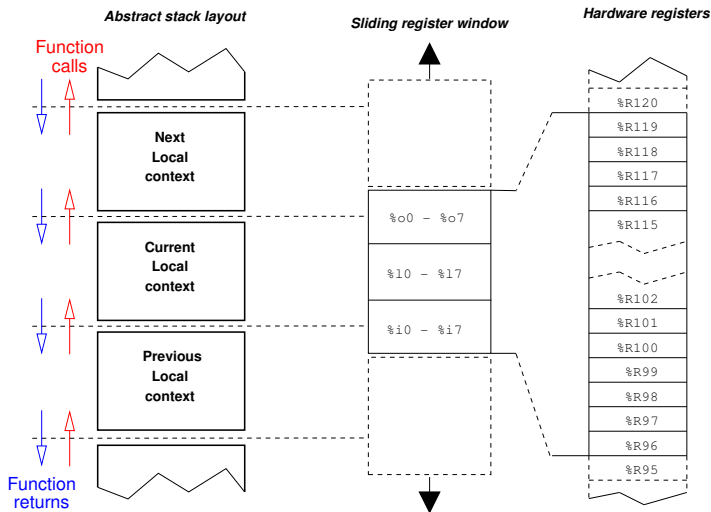


Register window

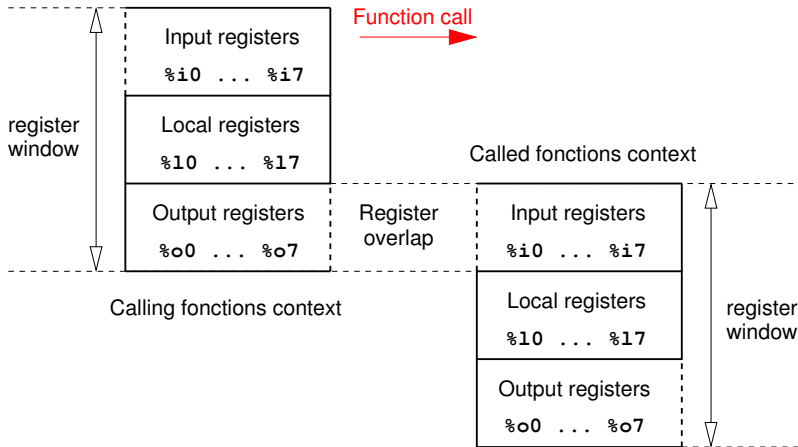
Hardware context stack implementation:

- ▶ Uses a large amount of registers (example: 512 on Sparc)
- ▶ Uses a logical limitation to define multiple contexts
- ▶ Does context change by sliding a window on each function call

Principle



Sparc overlapping register window



Function prologue and epilogue

prologue: slide register window

Example:

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

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

Example:

```
1          restore
```

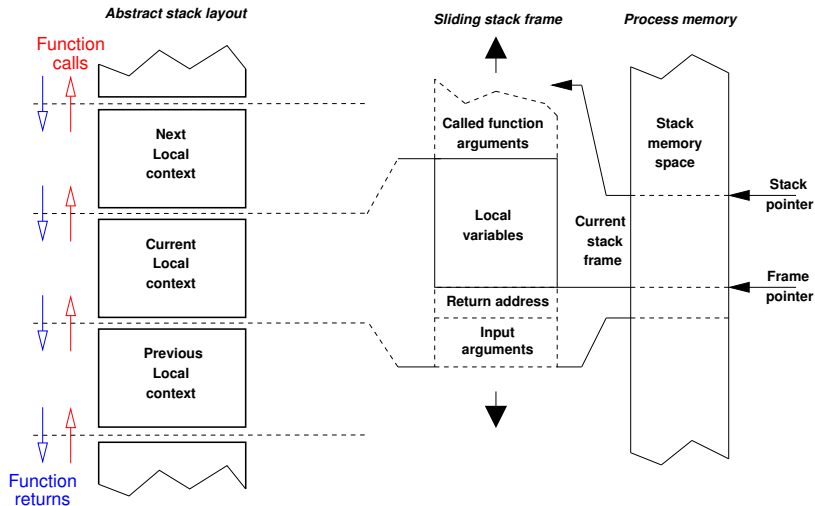
Memory stack

Register window has hard limitations:

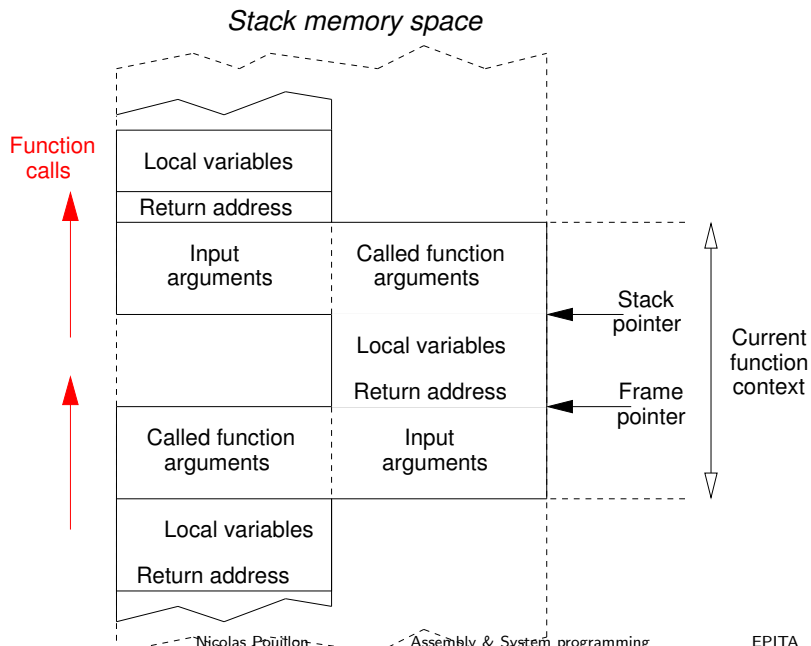
- ▶ The call depth is limited to the total amount of registers
- ▶ The CPU needs a large amount of physical registers \Rightarrow expensive

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

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 bits local variables (12 bytes) on its stack:

```
1      [%sp] <- %fp
2      %fp <- %sp
3      %sp <- %sp - 12
```

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

Example:

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


Argument and local variable access

argument:

Dereference the address “frame pointer + argument offset”:

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

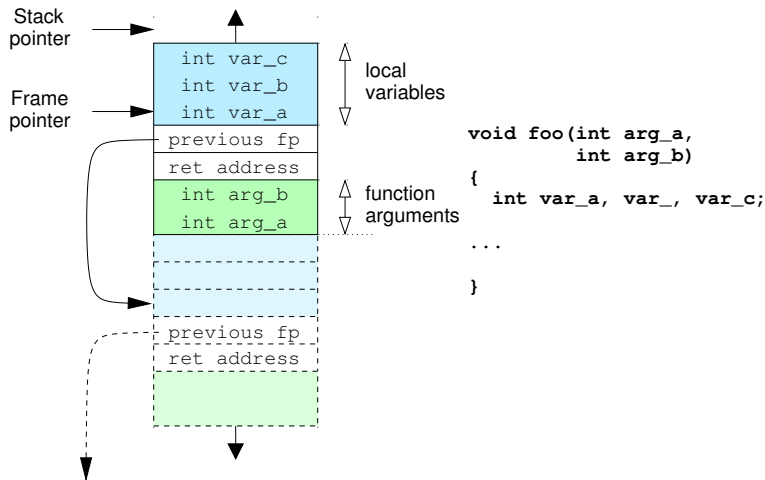
local variable:

Dereference the address “frame pointer - local variable offset”

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

Argument and local variable access

schema



Events

What are events ?

How to handle them ?

Categorizing events

An *interrupt* is caused by an external event.

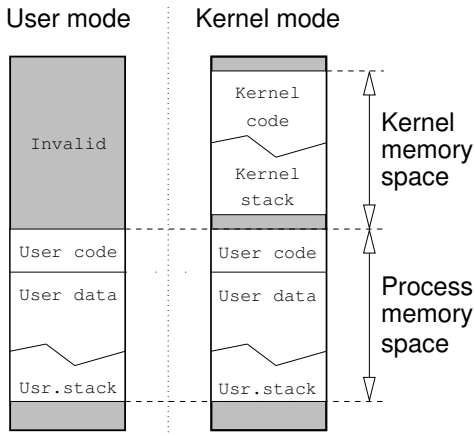
An *exception* is caused by instruction execution.

	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.

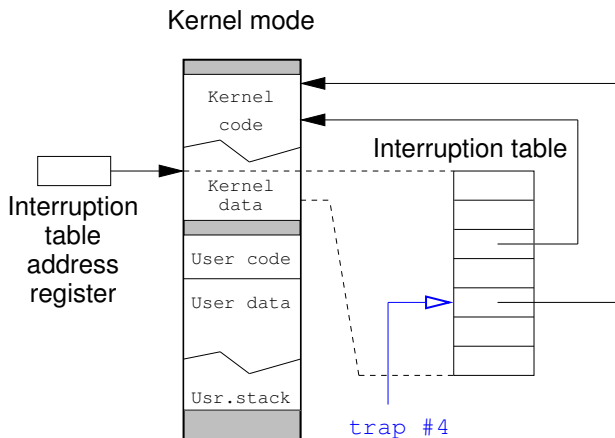
User space / kernel space

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 calls

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

- ▶ executing a process, exiting
- ▶ reading input, writing output (read, write...)
- ▶ performing *restricted actions* such as accessing hardware devices or accessing the memory management unit.
- ▶ etc, ...

Processor modes

Hardware facilities have been implemented to ensure process isolation in multi-user/multi-processor environment.

Today, processors have two (or more) execution modes:

- user mode

 - dedicated to user applications

- supervisor mode

 - reserved for 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 allow user to *switch* to kernel space and perform restricted actions, under certain conditions.

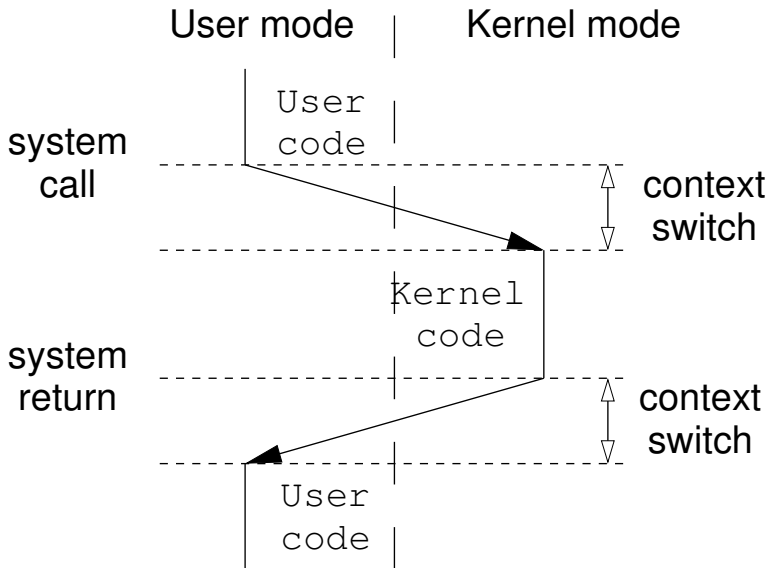
The kernel must check system call parameters validity.

System call implementation

1. System calls use a dedicated instruction which causes the processor to change mode to *superuser* (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

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

```
1  _SYSENTRY(pipe)
2  mov     %o0, %o2
3  mov     SYS_pipe, %g1
4  ta      %xcc, ST_SYSCALL
5  bcc,a,pt %xcc, 1f
6  stw     %o0, [%o2]
7  ERROR()
8  1:      stw     %o1, [%o2 + 4]
9  retl
10  clr     %o0
11  _SEND(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:

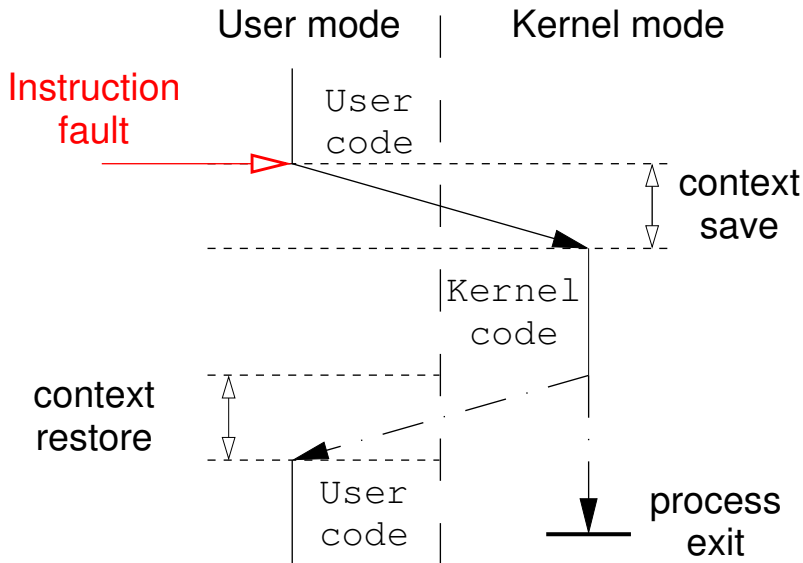
- ▶ Syscalls are deliberate, faults are unexpected
- ▶ Not every execution of the instruction results in a fault

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



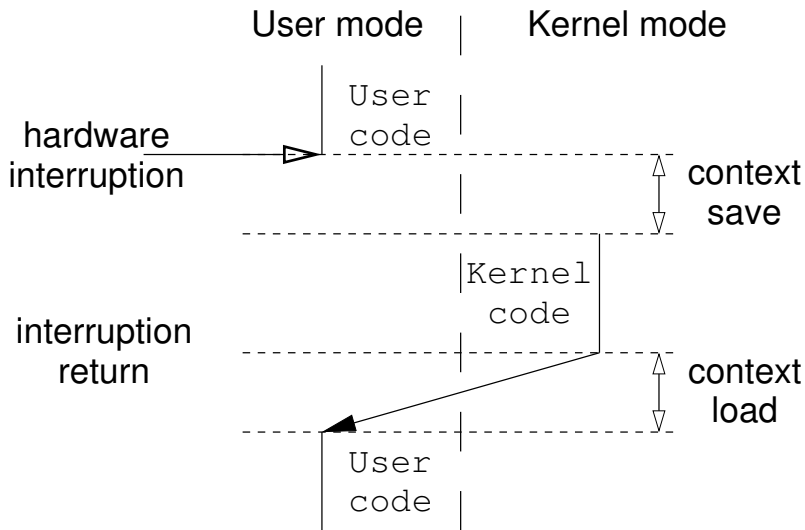
Events interface

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



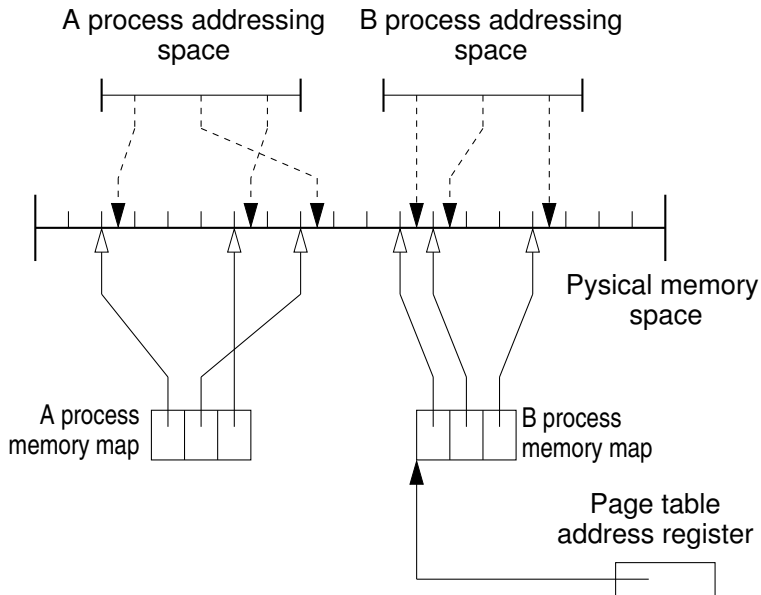
Multitasking

A single process may not use system resources at full capacity.
The idea of multitasking is to simulate the execution of concurrent execution of many processes, using a single processor.

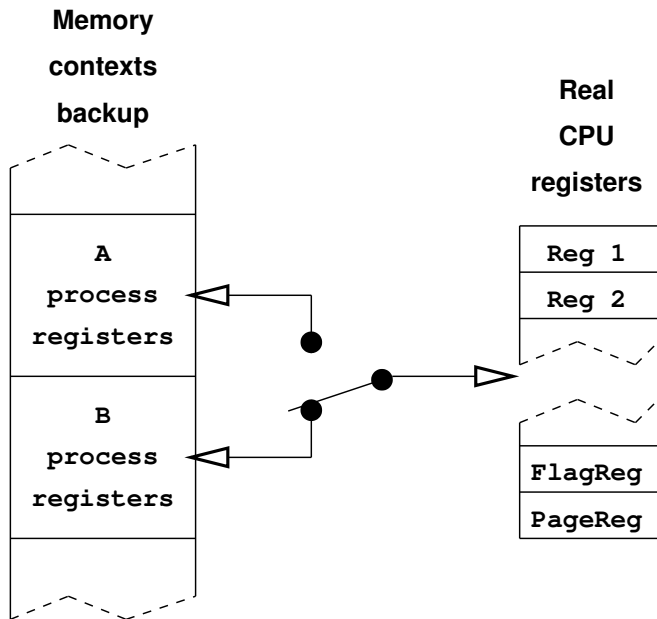
The operating system has to:

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

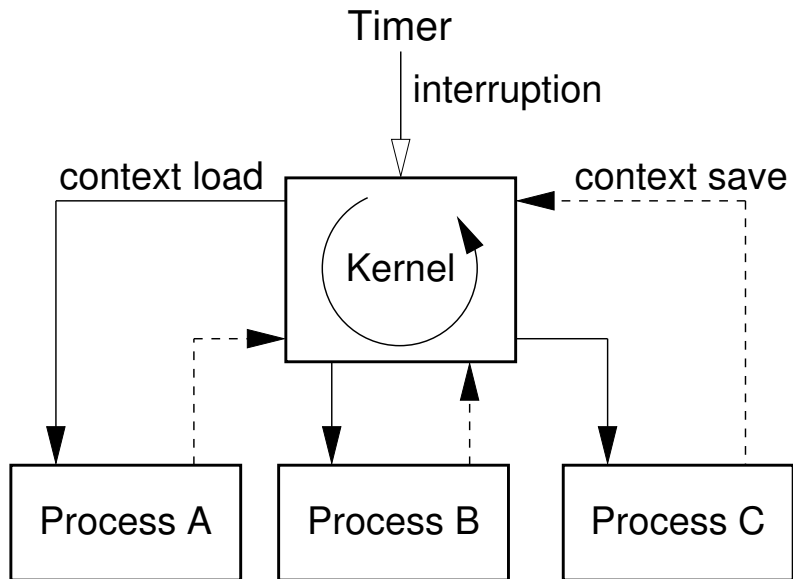
Memory mapping



Context switching



Process life



Part VIII

Focus on x86

Early events and CPU development

1971 Intel 4004

1978 Intel 8086 & 8088, first x86 CPUs

1981 Intel 80186

1982 Intel 80286: 16 bits, 24 address bits, protection

Successful CPU development

- 1985 Intel 80386: 32 bits, MMU
- 1989 Intel 80486: on-chip cache, pipeline
- 1993 Intel PentiumTM: superscalar, mmx, 64 address bits
- 1995 Intel Pentium Pro: ooo
- 1997 Intel Pentium II: internal L2
- 1999 Intel Pentium III: cpuid !
- 1999 AMD Athlon: ooo, 3dnow

Technology barriers

2000 Intel Pentium 4: HT

2001 Intel Itanium

2003 AMD Athlon 64

2003 Intel Pentium M: P6 (PPro to PIII)

2006 Intel Pentium 4 Prescott 2M: 64 bits

2006 Intel Core/Core2: fusion

2009 Intel Core i5/i7: ...

x86 architecture

The x86 architecture is:

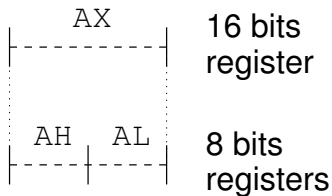
- ▶ based on a CISC instruction set.
- ▶ based on little-endian memory access.
- ▶ based on two operands instructions including memory access.
- ▶ backwards compatible: all new x86 processors are fully compatible with their predecessors.
- ▶ very complex: Pentium IV has a 20 stages pipeline.
- ▶ newer processors have less pipeline 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, ...

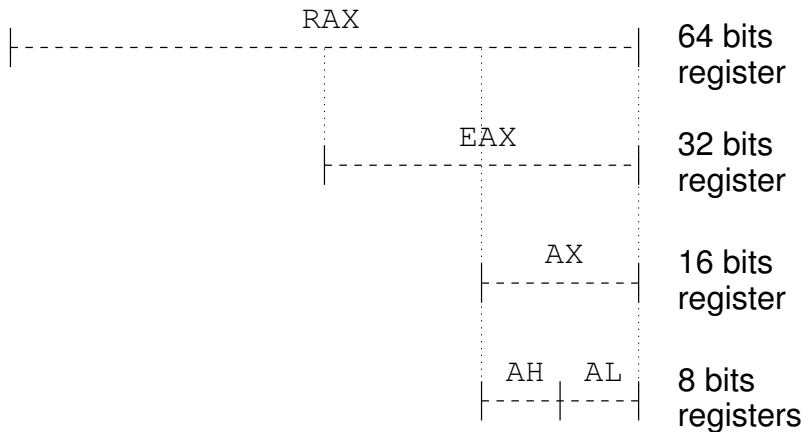
Nested registers



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 the amd64 to Intel after the Itanium flop...

32/64 bits nested registers



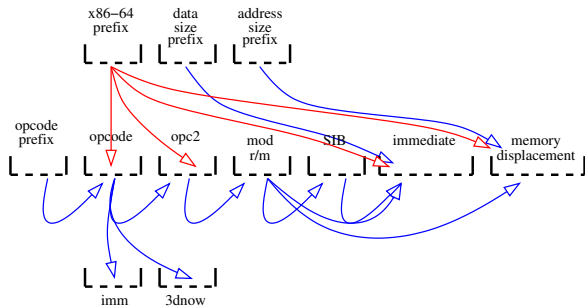
Instruction format

x86 architecture is a CISC based architecture:

- ▶ All instructions have a variable length,
- ▶ Instructions length can't be determined before reading 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.

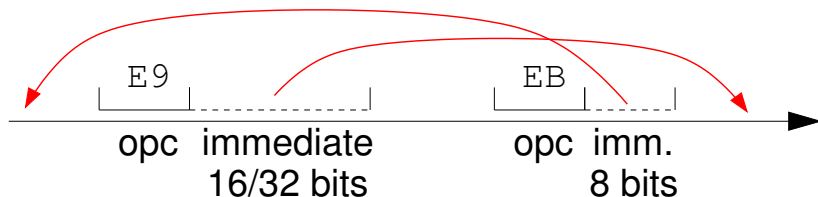
Branch

- ▶ 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:

- ▶ Memory access capable
- ▶ Handle different data widths
- ▶ Perform complex computation

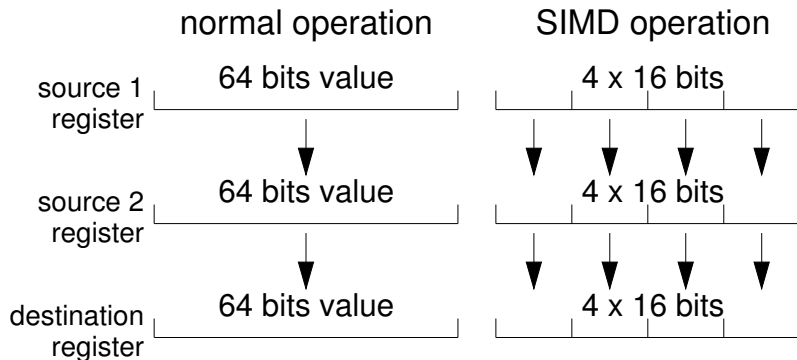
Instruction set extensions

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



x86-specific coding

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

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

Instruction set

has-been parts

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

x86-specific coding

example: LEA

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

```
1      lea eax, [ebx + ecx]
2
3      lea eax, [ebx * 8 + ebx]
```

x86-specific coding

example: tiniest mem*() ?

`memcpy` `rep movbs`

`memset` `rep stosd`

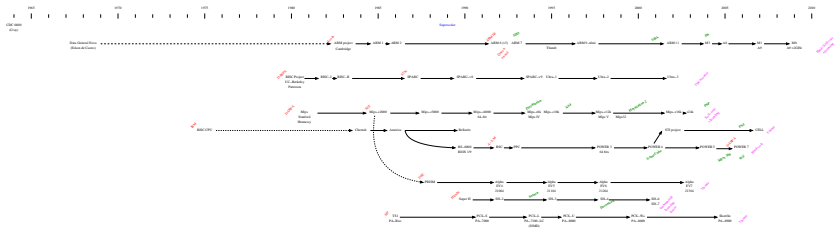
`memcmp` `repe cmpsb`

`strncmp` `repnz cmpsb`

Part IX

Focus on RISC processors

Let's see a timeline...



Mips

Instruction format

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

Mips

Asm code

```
1      addiu r2, r3, 0x42    ; alu reg / imm
2      or     r3, r4, r5     ; alu reg / reg
3
4      lui    r2, 0x1234     ; load upper immediate
5      ori    r2, r2, 0x5678 ; r2 = 0x12345678
6
7      lw     r4, 0x1234(r9) ; load word from r9 + 0x1234
8
9      slt    r1, r2, r3     ; test if r2 < r3
10     bnez   r1, 2f         ; jump if true
11     nop                               ; delay slot
12
13     lbu    r2, (r3)        ; load byte, not sign extended
14
15     jal    foo             ; pc = foo; r31 = return address
16     nop
```

SPARC

Instruction format

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

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

SPARC

Asm code

```
1      add    %g3, 0x42, %g2      ; alu reg / imm
2      or     %g3, %g4, %g5      ; alu reg / reg
3
4      sethi  0x12345, %g2      ; load upper immediate (20 bits)
5      or     %g2, 0x678, %g2    ; g2 = 0x12345678
6
7      ld     [%l2 + 0x1234], %g4 ; load word from l2 + 0x1234
8      ld     [%l2 + %i3], %g4   ; load word from l2 + i3
9
10     tst    %l2, %l3          ; test if l2 < l3
11     blt    3f                ; jump if true
```

PowerPC

Instruction format

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 GPR
- ▶ 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

```
1      addi    3, 2, 42          ; alu reg / imm
2      or.     3, 4, 5           ; alu reg / reg, update cr0
3
4      lis     3, 0x1234         ; load upper immediate (16 bits)
5      ori     3, 3, 0x5678      ; r3 = 0x12345678
6
7      lwz     4, 9, 0x1234      ; load word from r9 + 0x1234
8      lwzx    4, 9, 3           ; load word from r9 + r3
9
10     mtctr   4                 ; put r4 in count register
11     ...
12     bdnz    2f                ; Decrement CTR, Branch if CTR != 0
13
14     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, >, >=, “higher”, 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 format

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 format (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	l	rn	rd	imm/reg				
		1								rotate	imm			
		0								rs	0	ty	1	rm
		0								amount	ty	0	rm	

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

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

```
1      add    r3, r2, #0x42          ; alu reg / imm
2      orr    r3, r4, r5             ; alu reg / reg
3      bic    r3, r4, r5, lsr #4     ; alu reg / reg & shift
4
5      cmp    r2, #0                 ; test if r2 < 0
6      rsblt   r2, r2, #0             ; then r2 = 0 - r2
7
8      ldr     r2, #0x12345678        ; rewritten as
9      ldr     r2, [pc, #offset]      ; pc-relative load
10
11     streq   r4, [r2, r3, lsl #4]    ; store word to r2 + r3 << 4
12                                           ; only if last cmp is 'eq'
13     str     r4, [r2, #8]!           ; store word to r2 + 8
14                                           ; and r2 = r2 + 8
15
16     ; 32-bit immediates zone
17     offset:
18     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	l	rn	reg-list
------	-----	---	---	---	---	---	----	----------

- 1 `stmdb sp!, {r2, r3, r4, r8}`
- 2 `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...

```
1  push  {r2, r3, r4, r8, lr}
2  ...
3  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...

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

ARM Instruction-space exhaustion

Eventually, ARM instruction-set continued to grow despite the obvious exhaustion of the “clean” instruction-set space.

- ▶ abuse of concatenation of seldom bits around the instruction word
- ▶ 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
- ▶ 16 bit instruction words are not 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

Knowing the low-level implementation of the processors, we can:

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

abs() basic code

```
1  int abs(int x)
2  {
3      if ( x < 0 )
4          return -x;
5      else
6          return x;
7  }
```

```
1  int abs(int x)
2  {
3      return (x < 0) ? -x : x;
4  }
```

Nicer with 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 $-x((\text{not } x) + 1)((x \wedge 0xffffffff) - 0xffffffff)$
- ▶ if $(x > 0)$, return $x((x \wedge 0) + 0)$

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

- ▶ Sign extension: $(x \gg 31)$

Nicer with the pipeline

abs() better code

```
1  int abs(int x)
2  {
3      int sign_word = x >> 31;
4      return (x ^ sign_word) - sign_word;
5  }
```

```
1  int abs(int x)
2  {
3      int sign_word = -(1 & (x >> 31));
4      return (x ^ sign_word) - sign_word;
5  }
```

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 00000000000000000000snnnnnnnnnnnnnnn
-high 1111111111111111111110000000000000
result sssssssssssssssssssssnnnnnnnnnnnnnnn
```

```
1  int sign_ext(int val, int bits)
2  {
3      int high = 1 << (bits-1);
4      return (val & high) ? (val | (-high)) : val;
5  }
```

Sign extension at an arbitrary size

```
x 00000000000000000000snnnnnnnnnnnnnnn
<< snnnnnnnnnnnnnnn00000000000000000000
>> sssssssssssssssssssssssnnnnnnnnnnnnnnn
```

```
1 int sign_ext(int val, int bits)
2 {
3     int shift_bits = 32 - bits;
4     return (val << shift_bits) >> shift_bits;
5 }
```


Sign extension at an arbitrary size

```
x 00000000000000000000snnnnnnnnnnnnnnn
sb 00000000000000000000s00000000000000
xor 00000000000000000000Snnnnnnnnnnnnnnn
```

```
x 000000000000000000001nnnnnnnnnnnnnnn
sb 0000000000000000000010000000000000
x ^ sb 000000000000000000000nnnnnnnnnnnnnnn
.. - sb 111111111111111111111nnnnnnnnnnnnnnn
```

```
x 000000000000000000000nnnnnnnnnnnnnnn
sb 0000000000000000000010000000000000
x ^ sb 000000000000000000001nnnnnnnnnnnnnnn
.. - sb 000000000000000000000nnnnnnnnnnnnnnn
```

```
1 int sign_ext(int val, int bits)
2 {
3     int high = 1 << (bits-1);
4     return (val ^ high) - high;
5 }
```

Power of two

Properties

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

Examples:

- ▶ $1110010 - 1 = 1110001$
- ▶ $0100000 - 1 = 0011111$
- ▶ Lower bits up to the lowest 1 get flipped
- ▶ Other bits stay the same

```
1  int is_pow2(int n)
2  {
3      return !(n & (n-1));
4  }
```

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

```
1  int mask_merge(int mask, int where_0, int where_1)
2  {
3      return (mask & where_1) | (~mask & where_0);
4  }
```

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
where_0 ^ where_1	0	1	1	0	0	1	1	0
mask	0	0	1	1	1	1	0	0
(w0 ^ w1) & mask	0	0	1	0	0	1	0	0
((w0 ^ w1) & mask) ^ w0	1	1	0	0	1	1	0	0

```
1 int mask_merge(int mask, int where_0, int where_1)
2 {
3     return ((where_0 ^ where_1) & mask) ^ where_0;
4 }
```

Code readability

If you ever code with this kind of hack

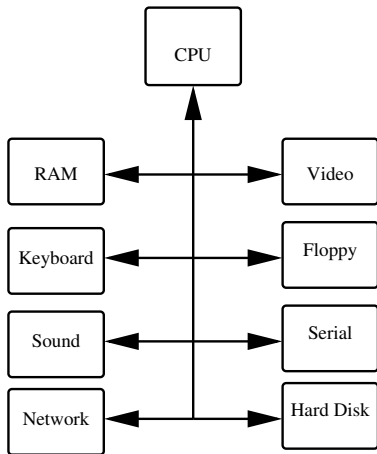
- ▶ **always** create functions with explicit name and prototype
- ▶ eventually document the **intended behavior**
- ▶ may use a static inline

```
1 int abs(int x);  
2 int sign_ext(int val, int bits);  
3 int is_pow2(int n);  
4 int mask_merge(int mask, int where_0, int where_1);
```

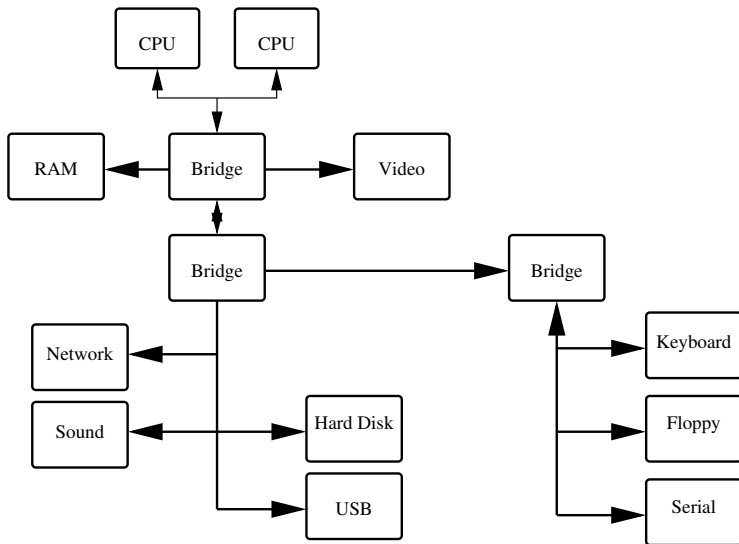
Part XI

Multi/Many core, heterogeneous systems

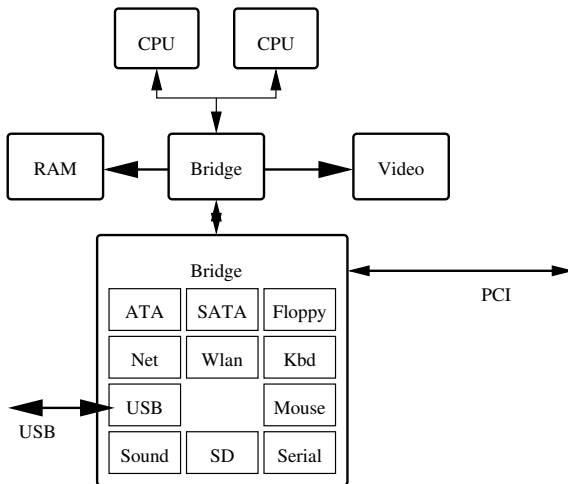
History of hardware topologies



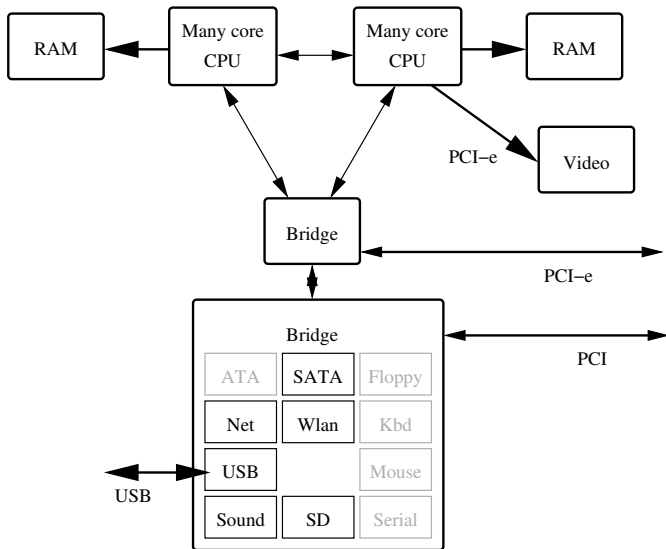
History of hardware topologies



History of hardware topologies



History of hardware topologies



Hardware topologies

Observations

Busses don't scale

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

We don't have busses any more

- ▶ We use networks (QPI, HyperTransport)
- ▶ This forbids snooping
- ▶ Coherence has 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 become NUMA

Some scalability bottlenecks

Coherent shared-memory systems

- ▶ are hard to design
- ▶ don't scale well
- ▶ get slower with the load

NUMA systems

- ▶ are hard to program for
- ▶ are not well supported in all OSes

Uncoherent shared-memory systems are not ready for prime-time yet