

Architecture and program structure

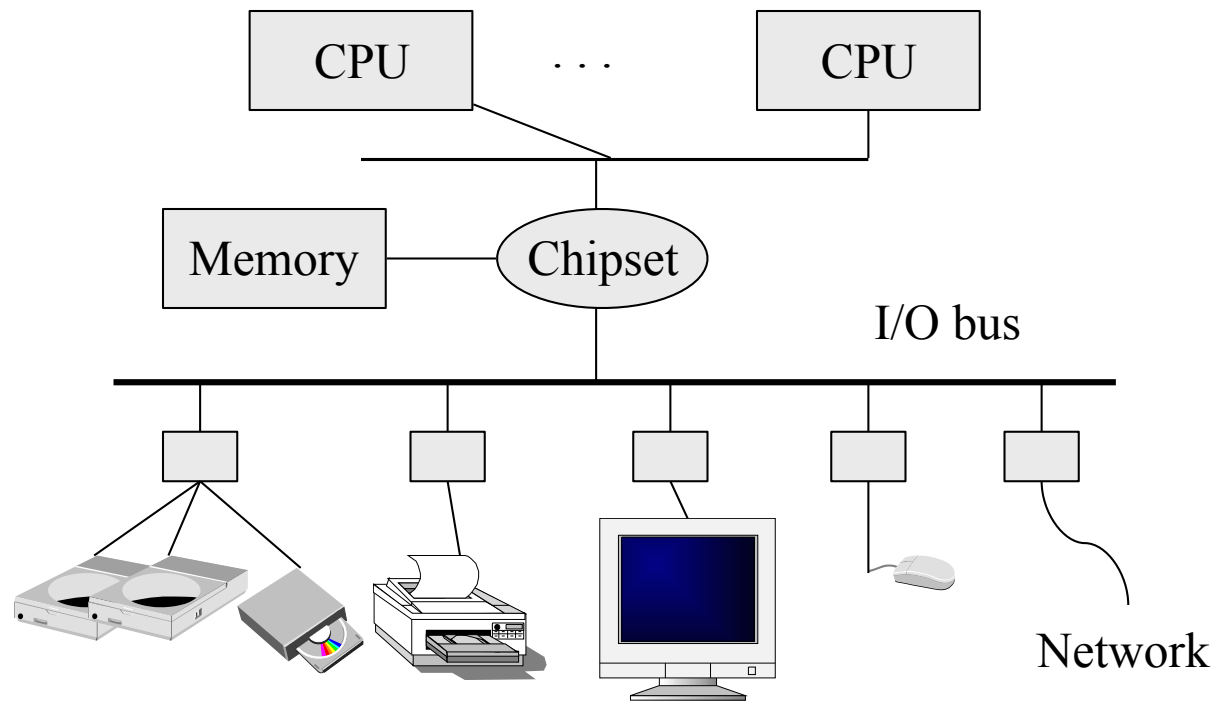
CS503: Operating systems, Spring 2019

Pedro Fonseca
Department of Computer Science
Purdue University

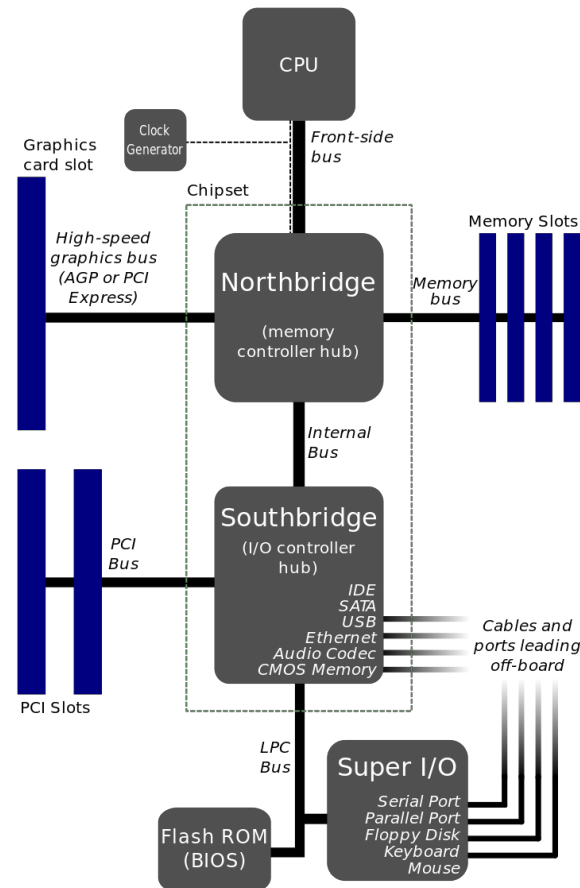
Previous lecture

- The definition of an OS
- The three roles of an OS
- A brief history of OSs
- The challenges uncovered by OSs and principles that are applicable to other systems

Typical computer

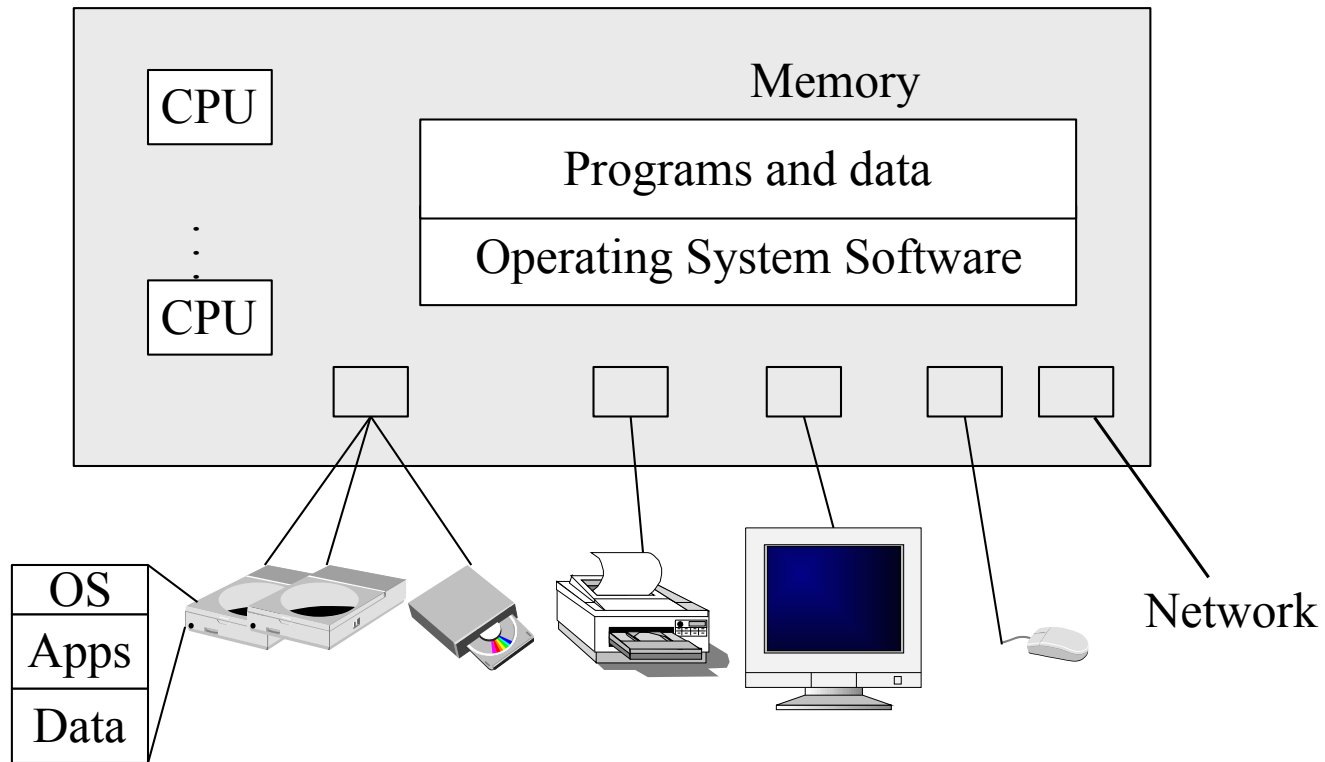


Typical PC architecture



(Modern Intel Core processors have the Northbridge integrated on the CPU die)

Typical computer



Processor

Processors

- Each CPU supports a specific set of instructions
- All CPUs contain
 - General registers inside to hold key variables and temporary results
 - Special registers visible to the programmer
 - Program counter contains the memory address of the next instruction to be fetched
 - Stack pointer points to the top of the current stack in memory
 - Control registers (e.g., CR0-CR4 in x86)
 - PSW (Program Status Word) contains the condition code bits which are set by comparison instructions, the CPU priority, the mode (user or kernel) and various other control bits.

How processors work?

- CPUs are always executing instructions
- Execute instructions
 - CPU cycles
 - Fetch (from mem) -> decode -> execute
 - Program counter (PC)
 - When is PC changed?
 - Pipeline: fetch $n+2$ while decode $n+1$ while execute n
 - Two modes of CPU (why?)
 - User mode (a subset of instructions)
 - Privileged mode (all instruction)
 - Trap (special instruction)

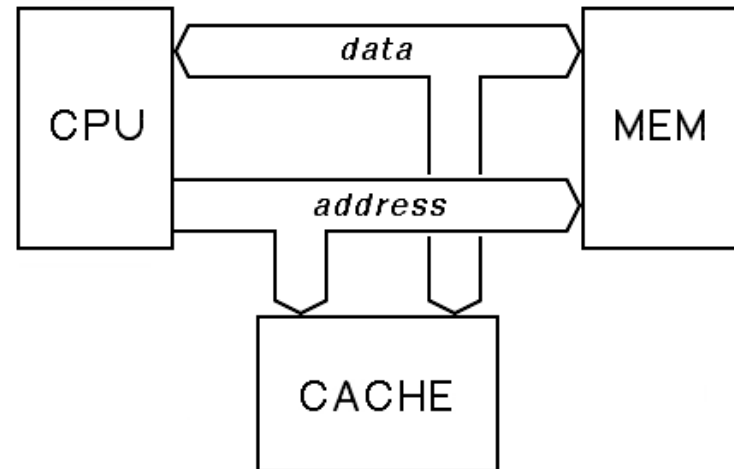
Memory and I/O

Memory access

- Memory read:
 - Assert address on address lines
 - Wait till data appear on data line
 - Much slower than CPU!
- How many mem access for one instruction?
 - Fetch instruction
 - Fetch operand (0, 1 or 2)
 - Write results (0 or 1)
- How to speed up instruction execution?

CPU cache

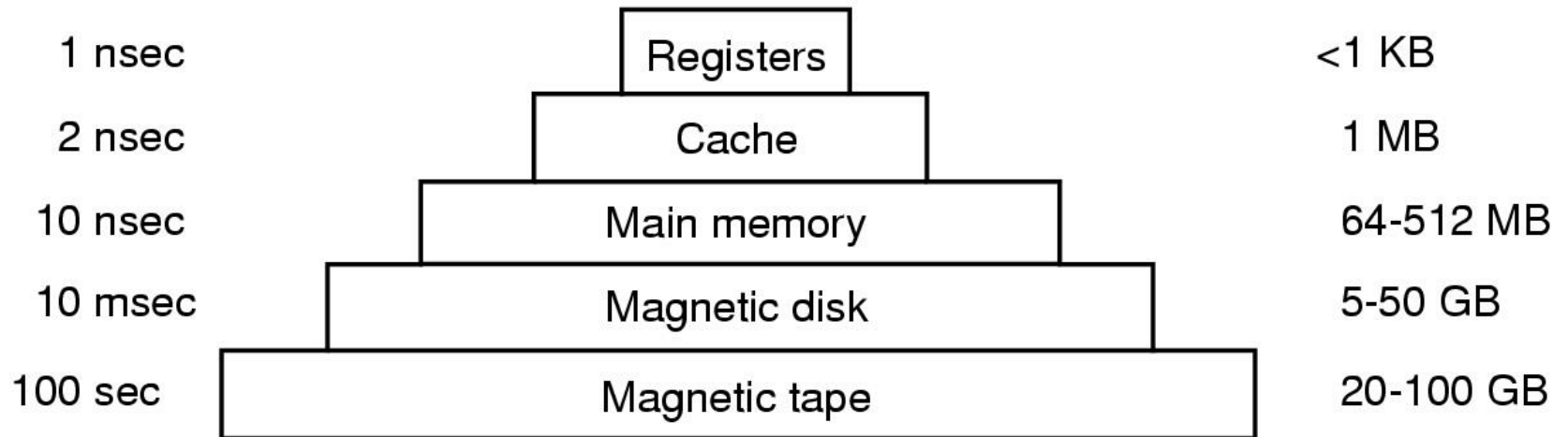
- Cache hit:
 - No need to access memory
- Cache miss:
 - Data obtained from memory, possibly updating the cache



Memory-storage hierarchy*

Typical access time

Typical capacity



What are the two other metrics missing?

***outdated numbers :(**

Memory

- Registers internal to CPU (as fast as CPU)
 - Storage 32x32 bits on a 32-bit CPU, 64x64 on 64 bit CPU (less than 1KB in both cases)
- Cache memory controlled by hardware
 - Cache hit and miss
- RAM (Random Access Memory)
- Disk (magnetic disk), CD-ROM, DVD,...
 - Cylinder, track, ...

Memory

- Non-volatile Memory
 - ROM (Read Only Memory)
 - Programmed at the factory and can't be changed
 - EEPROM (Electrically Erasable ROM)
 - Flash RAM
 - Can be erased and re-written
- Volatile Memory
 - CMOS holds current time and date

More details

Memory management

- How to protect programs from each other?
- How to handle relocation?
- Base register, limit register
- Check and Mapping of Addresses
 - Virtual Address - Physical Address
 - Memory Management Unit (MMU – located on CPU chip or close to it)
- Performance effects on memory system
 - Cache
 - Context switch

I/O devices

- Controller
 - Example: Disk Controller
 - Controllers are complex converting OS request into device parameters
 - Controllers often contain small embedded computers
- Device
 - Fairly simple interfaces and standardized
 - IDE (Integrated Drive Electronics) – standard disk type on Pentiums and other computers

I/O devices

- Device Driver
 - Needed since each type of controller may be different.
 - Software that talks to a controller, giving it commands and accepting responses
 - Each controller manufacturer supplies a driver for each OS it supports (e.g., drivers for Windows XP, Longhorn, UNIX)

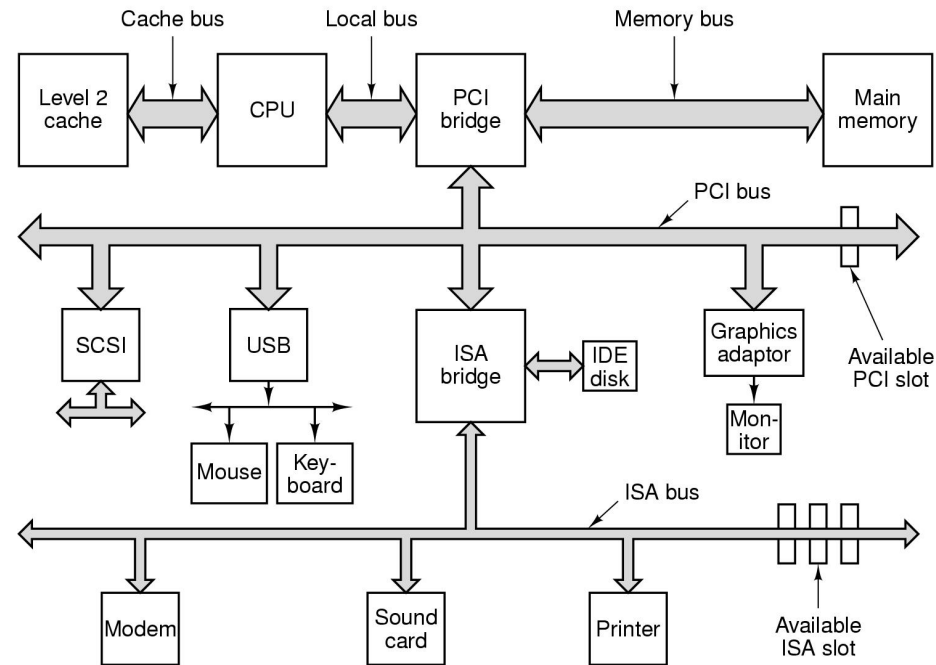
I/O mechanisms

- How device driver talks to controller?
 - Busy wait
 - Interrupt
 - Direct memory access (DMA)

Bus

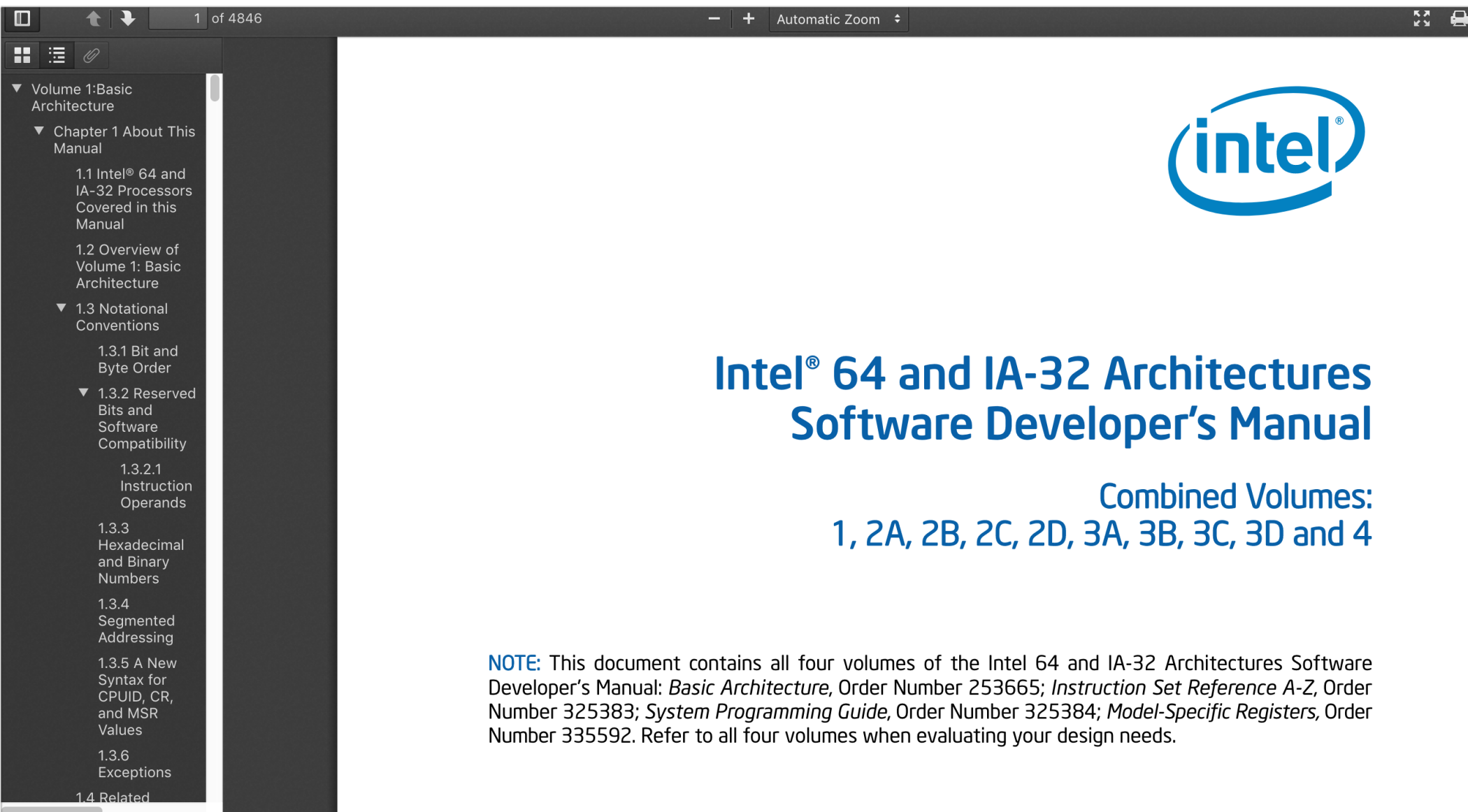
- Traditional Pentium systems had eight buses
- Cache, local, memory, PCI, SCSI, USB, IDE, ISA
 - PCI (Peripheral Component Interconnect)
 - ISA (Industry Standard Architecture) bus, 16.67 MB/sec
- Specialized buses:
 - SCSI (Small Computer System Interface)
 - USB (Universal Serial Bus)
 - IEEE 1394 – FireWire (Apple) bus, 50MB/sec
 - Thunderbolt 3, up to 40 Gb/s
 - IDE (Integrated Drive Electronics) bus (disk, CD-ROM. etc.)
- Internal buses tend to be **higher bandwidth** and **faster** (latency) than external buses

Structure of an Intel Pentium system



**How developers learn the CPU
interface?**

How to learn about the CPU interface?

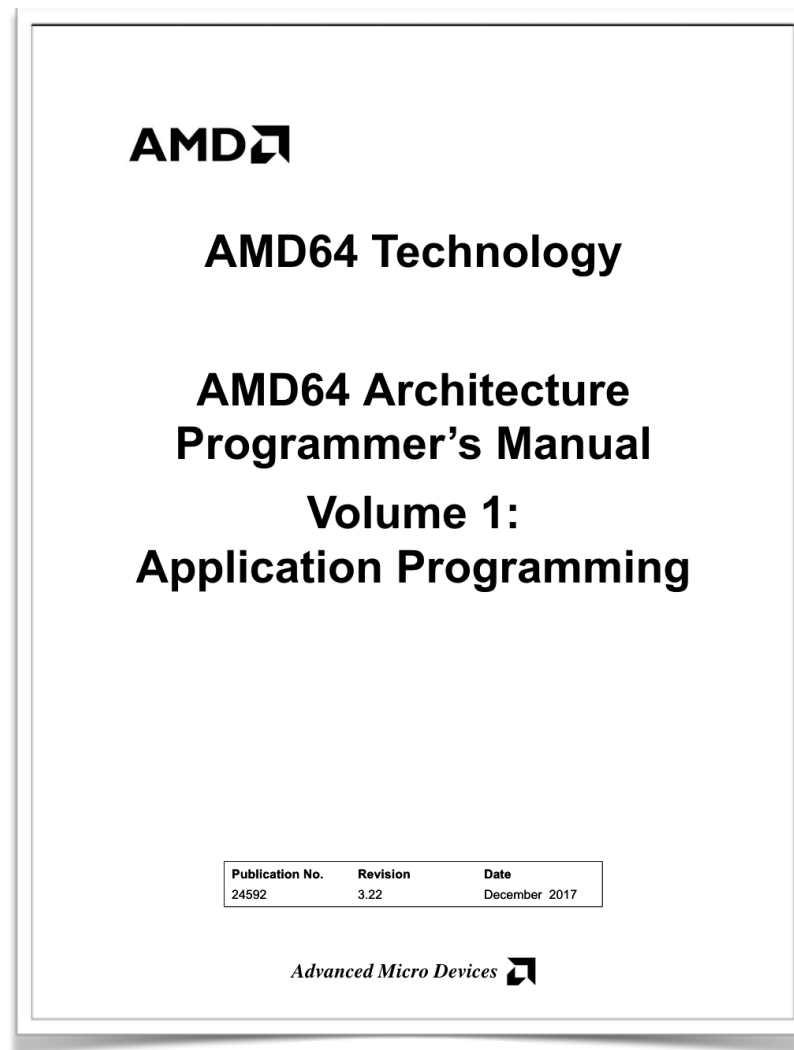


The image shows a PDF viewer window displaying the Intel 64 and IA-32 Architectures Software Developer's Manual. The window has a dark sidebar on the left with a table of contents, a top toolbar with navigation and zoom controls, and a main content area on the right. The sidebar lists the following sections:

- Volume 1: Basic Architecture
 - Chapter 1 About This Manual
 - 1.1 Intel® 64 and IA-32 Processors Covered in this Manual
 - 1.2 Overview of Volume 1: Basic Architecture
 - 1.3 Notational Conventions
 - 1.3.1 Bit and Byte Order
 - 1.3.2 Reserved Bits and Software Compatibility
 - 1.3.2.1 Instruction Operands
 - 1.3.3 Hexadecimal and Binary Numbers
 - 1.3.4 Segmented Addressing
 - 1.3.5 A New Syntax for CUID, CR, and MSR Values
 - 1.3.6 Exceptions
 - 1.4 Related

The main content area features the Intel logo at the top right, followed by the title "Intel® 64 and IA-32 Architectures Software Developer's Manual" in a large blue font. Below the title, it lists "Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4". At the bottom, a "NOTE" states: "This document contains all four volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-Z*, Order Number 325383; *System Programming Guide*, Order Number 325384; *Model-Specific Registers*, Order Number 335592. Refer to all four volumes when evaluating your design needs."

How to learn about the CPU interface?



AMD and Intel CPU's have differences but

INTEL 80386

PROGRAMMER'S REFERENCE MANUAL 1986

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and may only be used to identify Intel Products:

Above, BITBUS, COMPUTER, CREDIT, Data Pipeline, FASTPATH, Genius, i, I, ICE, iCEN, iCS, iDSP, iDIS, iFIRE, iLSE, iM, iMDX, iMMX, iBOARD, iAudio, iData, iData, iIntel, iIntel803, iIntel Certified, iIntelization, iIntelligent Identifier, iIntelligent Programming, iIntellic, iIntellink, iISP, iSPD, iSFC, iSMG, iSNO, iSRC, iSRX, iSEM, iSDM, iKERN, iLibrary Manager, iMAPIET, iMC, iMegachassis, iMICRONALFRAME, iMULTIBUS, iMULTICHANNEL, iMULTIMODULE, iMULTISERVER, iNCE, iONASSET, iOP, iC KERNEL, iPlug-A-Busible, iPROMPT, iProware, iQUEST, iQuik, iQuick-Pulse Programming, iRippledice, iRM/RS, iRTV, iSeamless, iSLD, iSuperCube, iSupportNET, iUPI, and iVLSI/CIL, and the combination of iCK, iCS, iLNX, iSRC, iSRX, iSEM, iSDM, iMC, or iUPI and a numerical suffix, 4-5TTE.

MDS is an ordering code only and is not used as a product name or trademark. MDS(R) is a registered trademark of Rohak Data Sciences Corporation.

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Distribution
Mail Stop SCV-59
3065 Bowers Avenue
Santa Clara, CA 95051

©INTEL CORPORATION 1987 CG-5/26/87 Edited 2001-02-01 by G.N.

Page 1 of 421

How to learn about the HW interface?

How to learn about the HW interface?



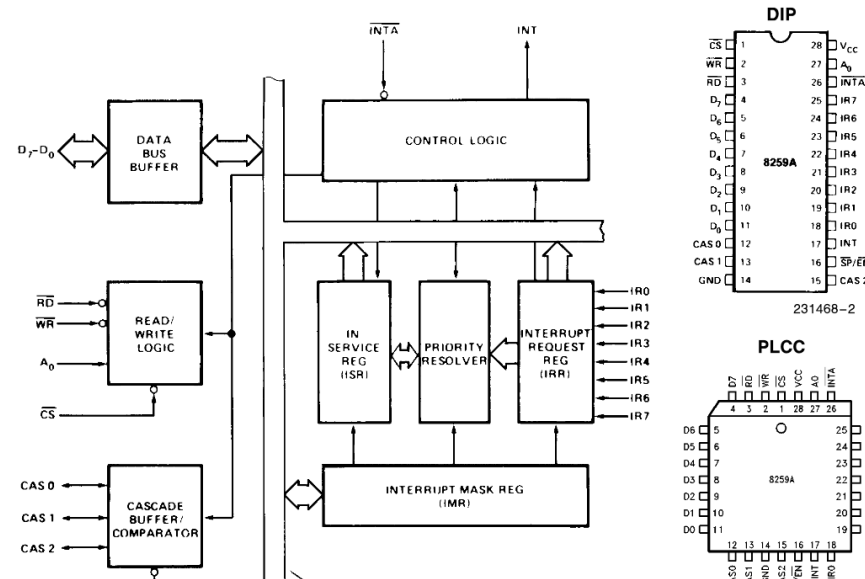
8259A PROGRAMMABLE INTERRUPT CONTROLLER (8259A/8259A-2)

- 8086, 8088 Compatible
- MCS-80, MCS-85 Compatible
- Eight-Level Priority Controller
- Expandable to 64 Levels
- Programmable Interrupt Modes
- Individual Request Mask Capability
- Single +5V Supply (No Clocks)
- Available in 28-Pin DIP and 28-Lead PLCC Package
(See Packaging Spec., Order #231369)
- Available in EXPRESS
 - Standard Temperature Range
 - Extended Temperature Range

The Intel 8259A Programmable Interrupt Controller handles up to eight vectored priority interrupts for the CPU. It is cascadable for up to 64 vectored priority interrupts without additional circuitry. It is packaged in a 28-pin DIP, uses NMOS technology and requires a single +5V supply. Circuitry is static, requiring no clock input.

The 8259A is designed to minimize the software and real time overhead in handling multi-level priority interrupts. It has several modes, permitting optimization for a variety of system requirements.

The 8259A is fully upward compatible with the Intel 8259. Software originally written for the 8259 will operate the 8259A in all 8259 equivalent modes (MCS-80/85, Non-Buffered, Edge Triggered).



How to learn the HW interface?

- Read vendor manuals / datasheets
- Read the source code of another driver
- Reverse engineer an existing driver:
 - Binary analysis
 - HW probes
- Try to guess (e.g., assume it's similar to other devices)

CPU instructions are implemented with code too!

Reverse Engineering x86 Processor Microcode

Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison,
Robert Gawlik, Christof Paar, and Thorsten Holz

Ruhr-Universität Bochum

Abstract

Microcode is an abstraction layer on top of the physical components of a CPU and present in most general-purpose CPUs today. In addition to facilitate complex and vast instruction sets, it also provides an update mechanism that allows CPUs to be patched in-place without requiring any special hardware. While it is well-known that CPUs are regularly updated with this mechanism, very little is known about its inner workings given that microcode and the update mechanism are proprietary and have not been thoroughly analyzed yet.

In this paper, we reverse engineer the microcode semantics and inner workings of its update mechanism of conventional COTS CPUs on the example of AMD's K8 and K10 microarchitectures. Furthermore, we demonstrate how to develop custom microcode updates. We describe the microcode semantics and additionally present a set of

hardware modifications [48]. Dedicated hardware units to counter bugs are imperfect [36, 49] and involve non-negligible hardware costs [8]. The infamous *Pentium fdiv* bug [62] illustrated a clear economic need for field updates after deployment in order to turn off defective parts and patch erroneous behavior. Note that the implementation of a modern processor involves millions of lines of HDL code [55] and verification of functional correctness for such processors is still an unsolved problem [4, 29].

Since the 1970s, x86 processor manufacturers have used microcode to decode complex instructions into series of simplified microinstructions for reasons of efficiency and diagnostics [43]. From a high-level perspective, microcode is an interpreter between the user-visible Complex Instruction Set Computer (CISC) Instruction Set Architecture (ISA) and internal hardware based on Reduced Instruction Set Computer (RISC) paradigms [54]. Al-

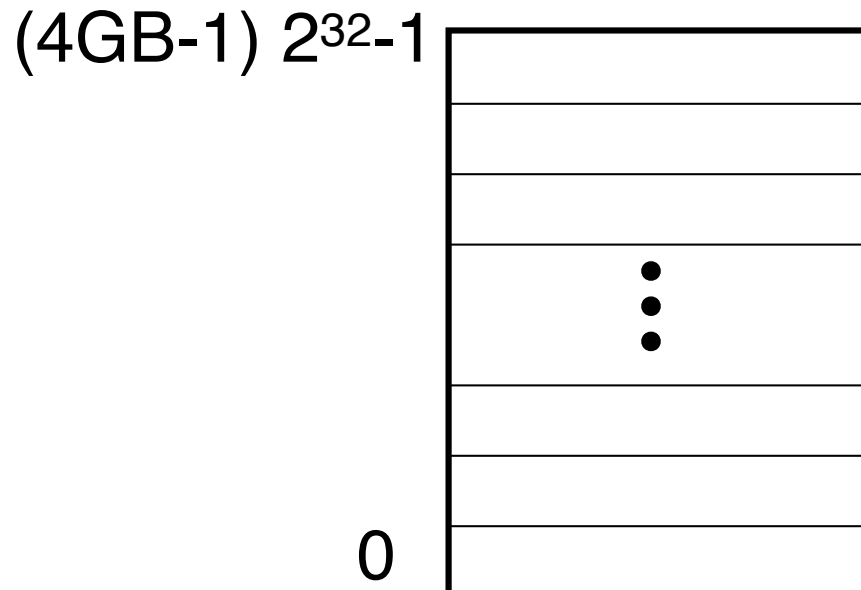
USENIX Security'17

...and some have found ways to modify them

Program structure

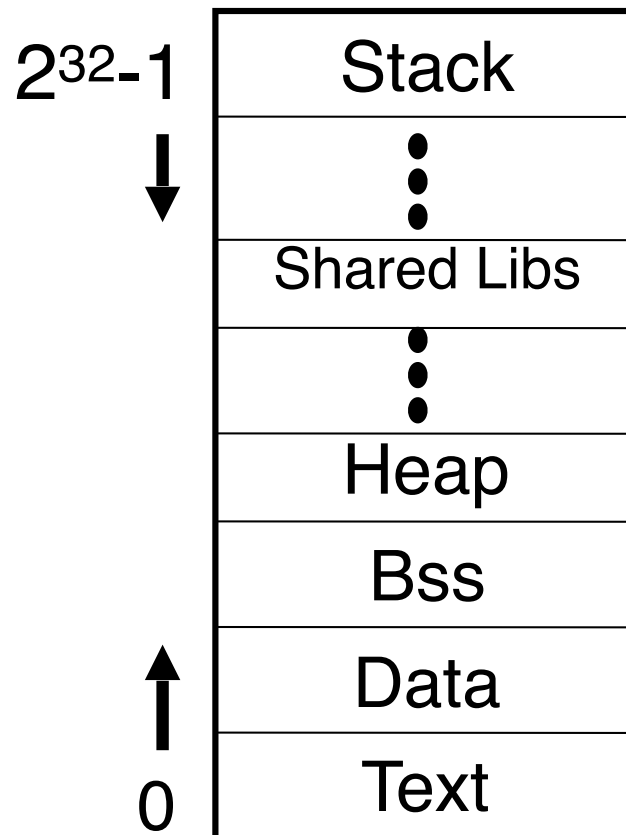
Virtual address space of a program

- A program sees (virtual) memory as an array of bytes
- Array goes from address 0 to $2^{32}-1$ (0 to 4GB-1)
 - (assuming a 32-bit architecture)



Memory sections

- The virtual address space is organized into sections:



Memory sections

- Each section has different permissions: read/write/execute or a combination of them
- **Text:** Instructions that the program runs
- **Data:** Initialized global variable.
- **BSS:** Uninitialized global variables. They are initialized to zeroes (saving object file size)
- **Heap:** Memory returned when calling malloc/new. It grows upwards (from low to high address)
- **Stack:** It stores local variables and return addresses. It grows downwards (from high to low address)

Memory sections

- Dynamic libraries – They are libraries shared with other processes
 - Each dynamic library has its own text, data, and bss
- Each run of the program (a.k.a. **process**) has its own view of the memory that is independent of each other
- This view is called the “Virtual Address Space” of the process
- If a **process** modifies a byte in its own address space, it will not modify the address space of another process

Example

Program hello.c

```
int a = 5;    // Stored in ? section
int b[20];    // Stored in ? section
int main() {  // Stored in ? section
    int x;     // Stored in ? section
    int *p = (int*)
        malloc(sizeof(int)); //In ? section
}
```

Example

Program hello.c

```
int a = 5;    // Stored in data section
int b[20];    // Stored in bss
int main() {  // Stored in text
    int x;     // Stored in stack
    int *p = (int*)
        malloc(sizeof(int)); //In heap
}
```

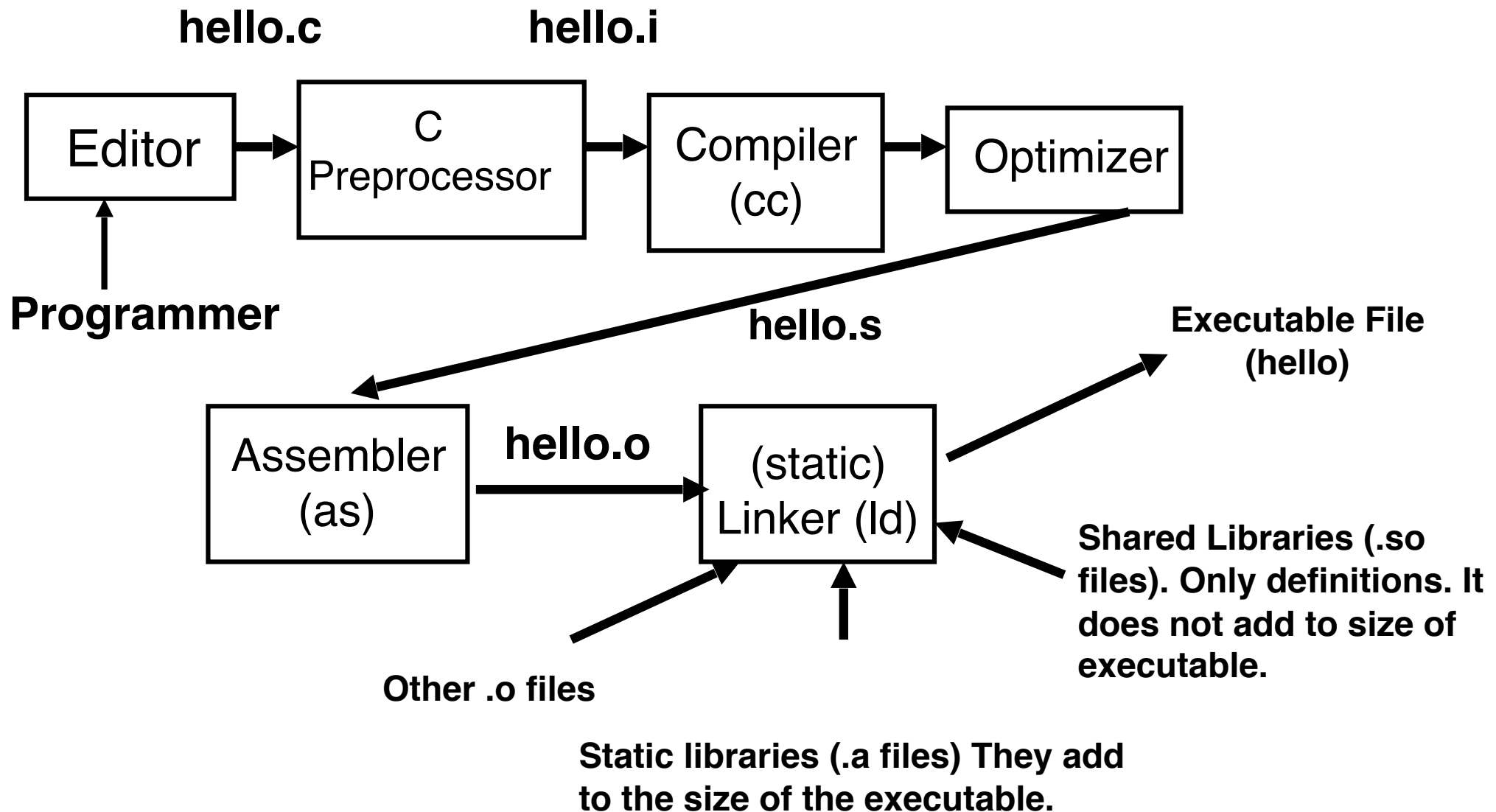
Building a program

- The programmer writes a program `hello.c`
- The preprocessor expands `#define`, `#include`, `#ifdef` etc preprocessor statements and generates a `hello.i` file.
- The compiler compiles `hello.i`, optimizes it and generates an assembly instruction listing `hello.s`
- The assembler (`as`) assembles `hello.s` and generates an object file `hello.o`
- The compiler (`cc` or `gcc`) by default hides all these intermediate steps. You can use compiler options to run each step independently.

Building a program

- The ***linker*** puts together all object files as well as the object files in static libraries
- The linker also takes the definitions in shared libraries and verifies that the symbols (functions and variables) needed by the program are completely satisfied
- If there is symbol that is not defined in either the executable or shared libraries, the linker will give an error
- Static libraries (.a files) are added to the executable.
shared libraries (.so files) are not added to the executable file

Building a program



Original file hello.c

```
#include <stdio.h>

int a;
int b = 1;

main()
{
    printf("Hello\n");
}
```

After preprocessor

```
gcc -E hello.c > hello.i
```

(-E stops compiler after running preprocessor)

```
hello.i:
```

```
    /* Expanded /usr/include/stdio.h */
typedef void *__va_list;
typedef struct __FILE __FILE;
typedef int      ssize_t;
struct FILE {...};
extern int fprintf(FILE *, const char *, ...);
extern int fscanf(FILE *, const char *, ...);
extern int printf(const char *, ...);
/* and more */
main()
{
    printf("Hello\n");
}
```


After compilation

gcc -S hello.c

(-S stops compiler after compilation)

hello.s:

```
        .align 8
.LLC0:  .asciz  "Hello\n"
.section      ".text"
        .align 4
        .global main
        .type   main,#function
        .proc   04
main:    save   %sp, -112, %sp
        sethi   %hi(.LLC0), %o1
        or      %o1, %lo(.LLC0), %o0
        call    printf, 0
        nop
.LL2:    ret
        restore
.
```

Output of assembler: Object file

- “gcc -c hello.c” generates hello.o
- hello.o has undefined symbols
 - that we don’t know where they will be located
 - e.g., the **printf** function call
- The **main** function, **a** and **b** already have values relative to the object file hello.o

```
$ nm -vS hello.o
```

```
                                U printf
00000000000000000000000000000004 D b
00000000000000000000000000000033 T main
00000000000000000000000000000004 C a
```

Objdump can also
analyze object files

Object file components

- Object file header
 - Size and position of other sections
- Text segment
- Data segment
- Relocation information
- Symbol table
 - Labels not defined such as external references
- Debugging information

After linking

- “gcc -o hello hello.c” generates the hello executable
- **printf** does not have a value yet until the program is loaded

```
$ nm hello
```

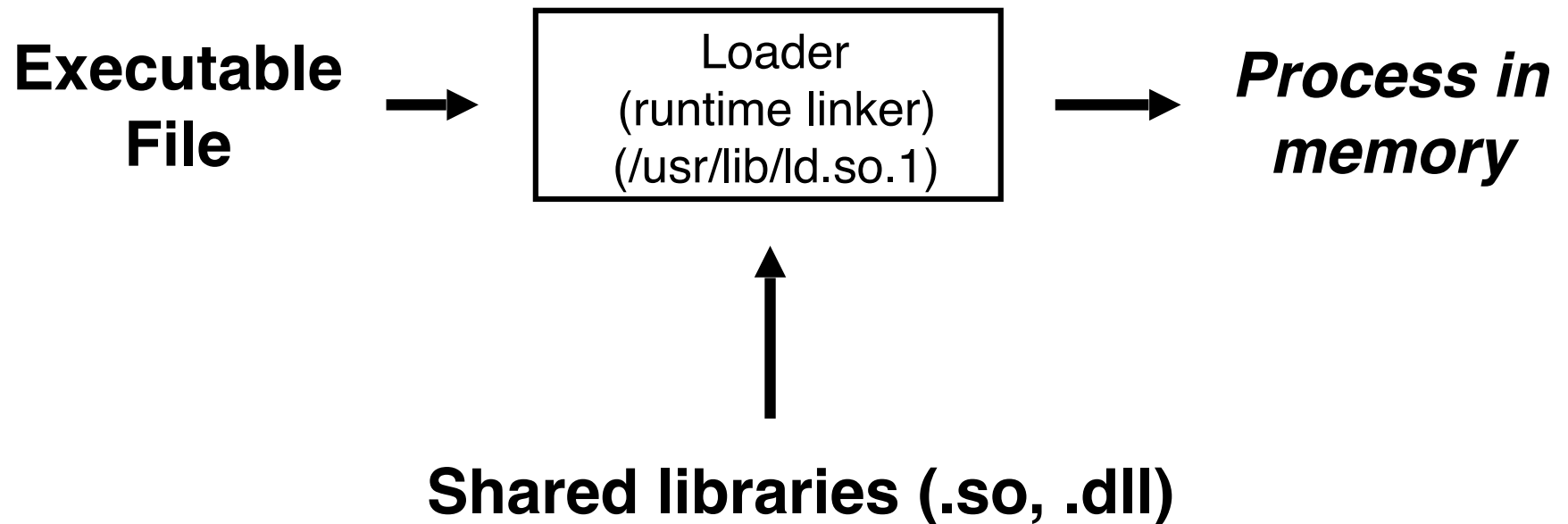
```
                U printf@@GLIBC_2.2.5
00000000004003c8 T _init
0000000000400430 000000000000002a T _start
0000000000400460 t deregister_tm_clones
00000000004004a0 t register_tm_clones
00000000004004e0 t __do_global_dtors_aux
0000000000400500 t frame_dummy
0000000000400526 0000000000000033 T main
0000000000400560 0000000000000065 T __libc_csu_init
00000000004005d0 0000000000000002 T __libc_csu_fini
00000000004005d4 T _fini
00000000004005e0 0000000000000004 R _IO_stdin_used

..
0000000000601028 D __data_start
0000000000601028 W data_start
0000000000601030 D __dso_handle
0000000000601038 0000000000000004 D b
000000000060103c B __bss_start
0000000000601040 0000000000000004 B a
```

During run-time

Loading a program

(... during runtime)



Loading a program

- The loader is part of the OS to prepare a process for an executable program
- Reads the executable file header to determine size of text and data segments
- Creates an address space large enough for text and data
- Copies instructions and data from executable file into memory

Loading a program

- Copies parameters to the main program (**argc**, **argv**) onto the stack
- Initialize registers and set stack pointer
- Jumps to a startup routine **_start** that in turn calls the **main()** routine. When the main routine returns, the startup routine terminates the program by making system call (executed by the OS) **exit()**
 - Will see how this is done in Xinu

Static and shared libraries

- Shared libraries are shared across different processes
- There is only one instance of each shared library for the entire system
- Static libraries are not shared
- There is an instance of a static library for each process

```
#include <stdlib.h>
```

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

```
char a;
```

```
int b = 3;
```

```
int main(){
```

```
    int s;
```

```
    printf("a = %p\n", &a);
```

```
    printf("b = %p\n", &b);
```

```
    printf("s = %p\n", &c);
```

```
    printf("h = %p\n", malloc(sizeof(int)));
```

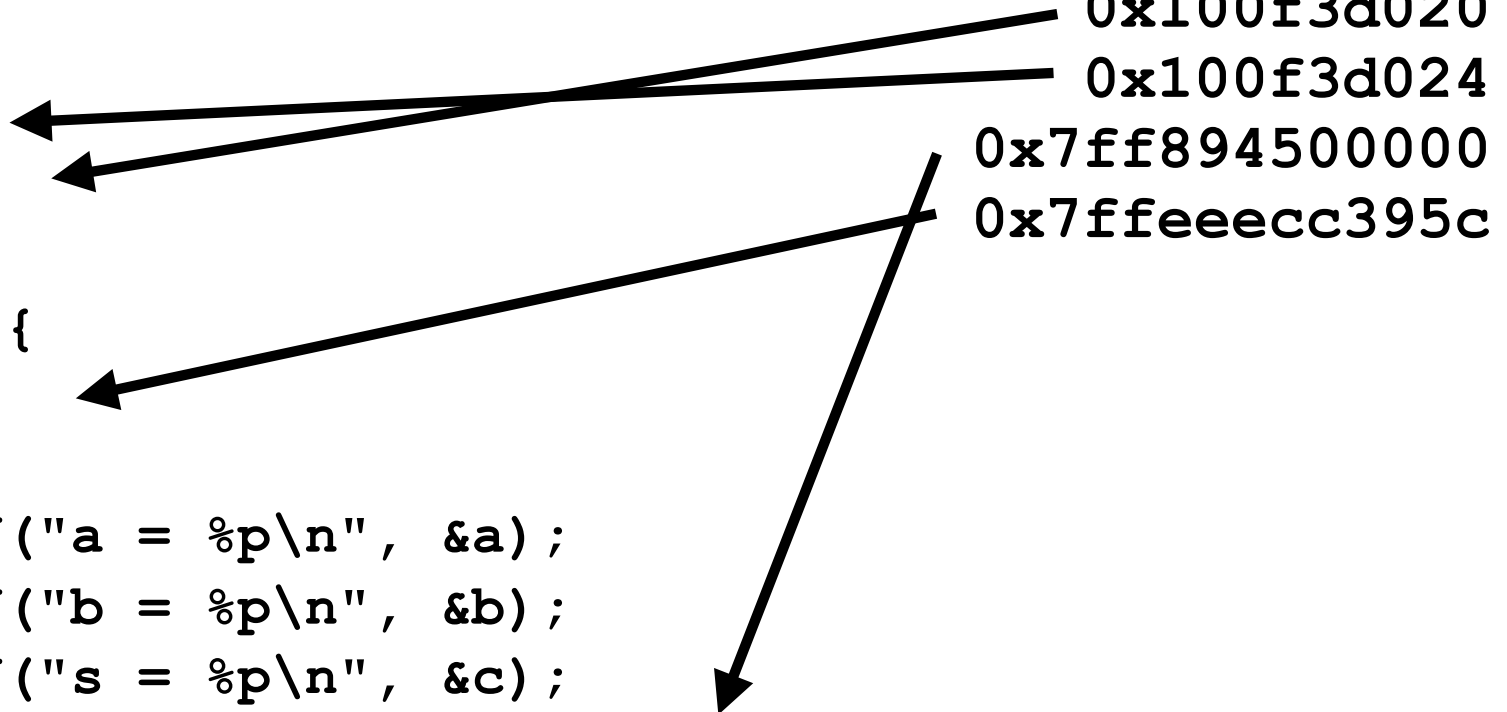
```
}
```

0x100f3d020

0x100f3d024

0x7ff894500000

0x7ffeeec395c



Run 1:

```
./hello  
a = 0x10d9ac024  
b = 0x10d9ac020  
s = 0x7ffee225495c  
h = 0x7fdd02402a80
```

Run 2:

```
./hello  
a = 0x100f3d024  
b = 0x100f3d020  
s = 0x7ffeeecc395c  
h = 0x7ff894500000
```

Why are values of the two runs different?

- Security feature: address space randomization (ASLR)
- Constant / predictable memory addresses makes certain types of attacks (when apps are buggy) easier or more dangerous

Summary

- Background on architecture and program structure
- Architecture:
 - CPU, memory hierarchy
 - IO devices
- Program structure
 - Compilation, linking
 - Run-time: Loading
 - Shared vs. static variables