# Computer Systems II

Li Lu
Room 605, CaoGuangbiao Building
li.lu@zju.edu.cn
https://person.zju.edu.cn/lynnluli

# Exceptions and Interrupts

# Exceptions and Interrupts

- “Unexpected” events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within CPU
    - e.g., undefined opcode, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

# Handling Exceptions

- Save PC of offending (or interrupted) instruction
  - In RISC-V: Supervisor Exception Program Counter (SEPC)
- Save indication of the problem
  - In RISC-V: Supervisor Exception Cause Register (SCAUSE)
  - 64 bits, but most bits unused
    - Exception code field: 2 for undefined opcode, 12 for hardware malfunction, …
- Jump to handler
  - Assume at 0000 0000 1C09 0000hex

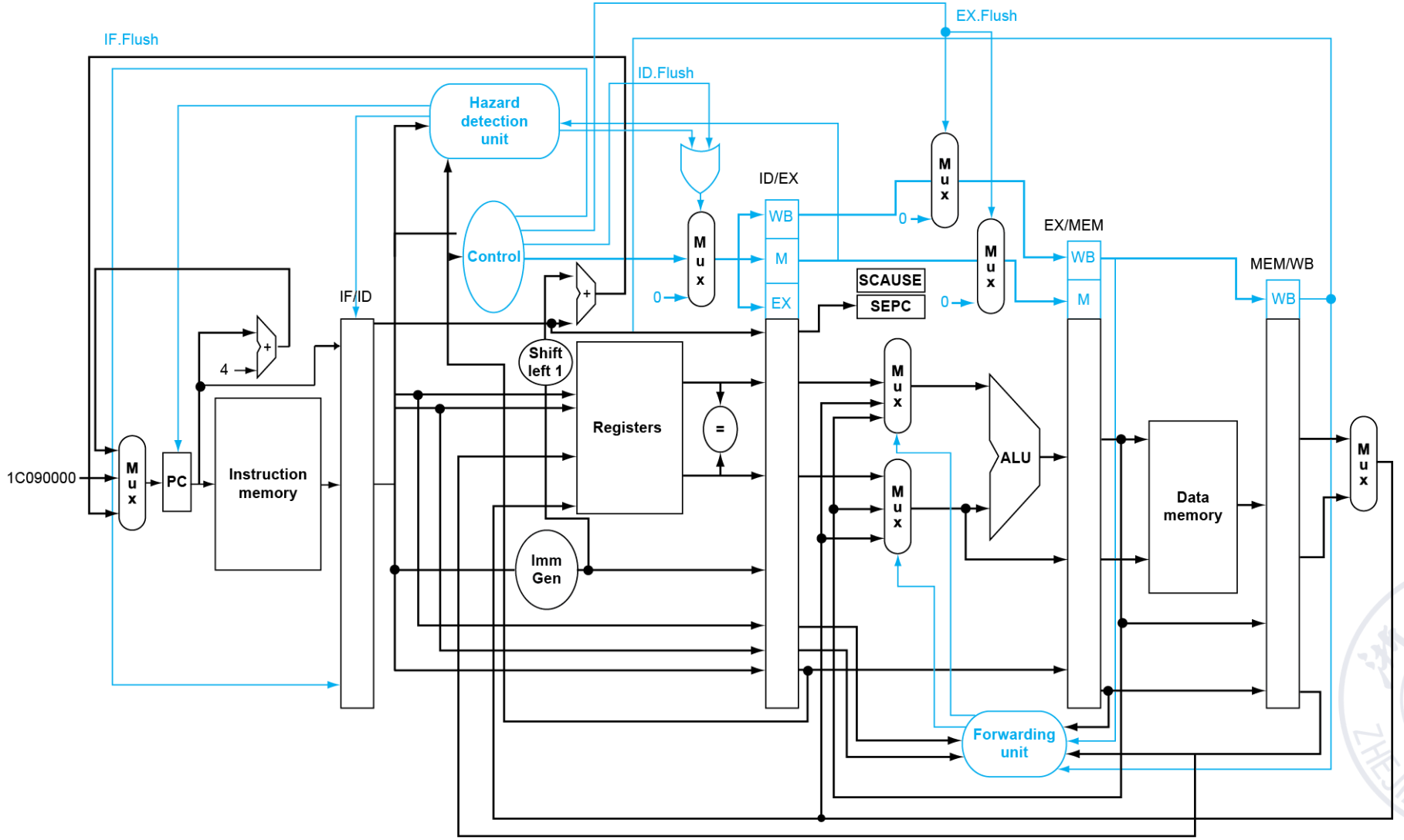
# Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use SEPC to return to program
- Otherwise
  - Terminate program
  - Report error using SEPC, SCAUSE, …

# Exceptions in a Pipeline

- Another form of control hazard
- Consider malfunction on add in EX stage add x1, x2, x1
  - Prevent x1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set SEPC and SCAUSE register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

# Pipeline with Exceptions



# Exception Properties

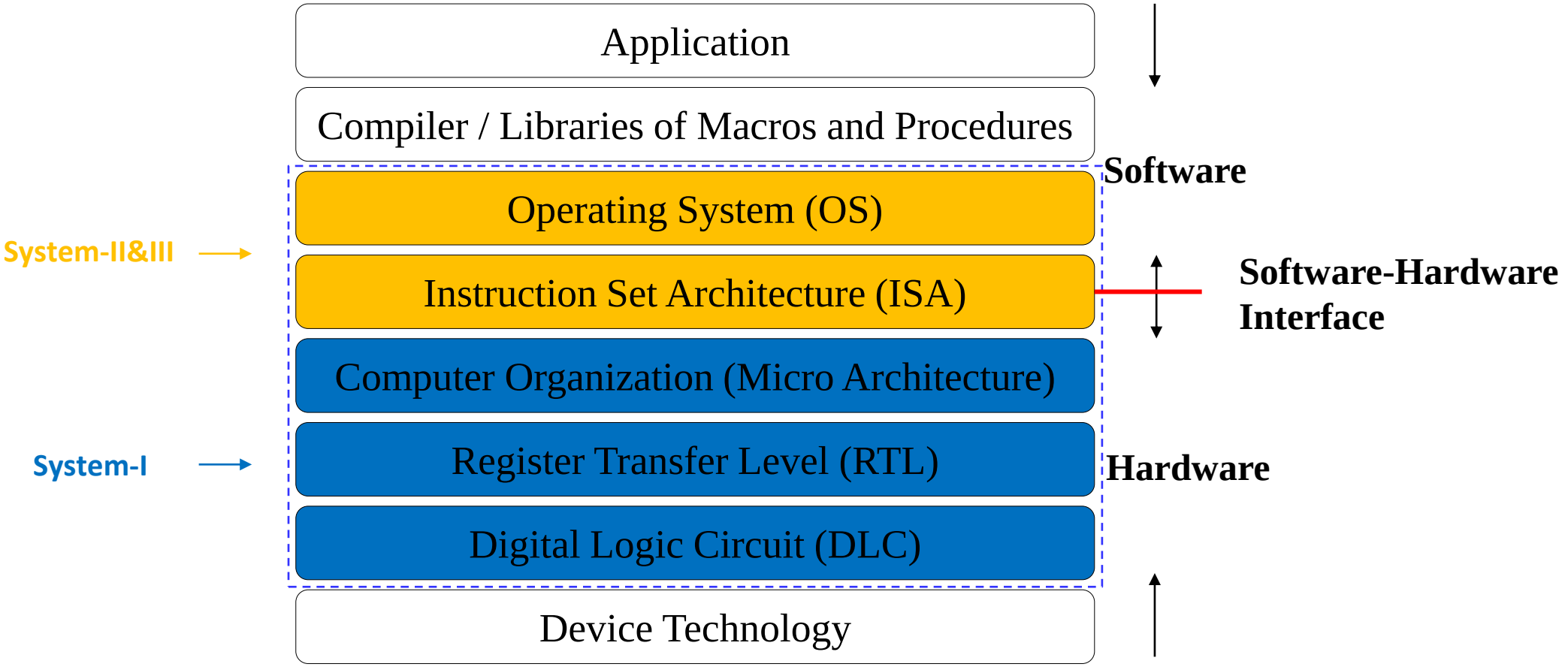
- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in SEPC register
  - Identifies causing instruction

# Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

# Software-Hardware Interface

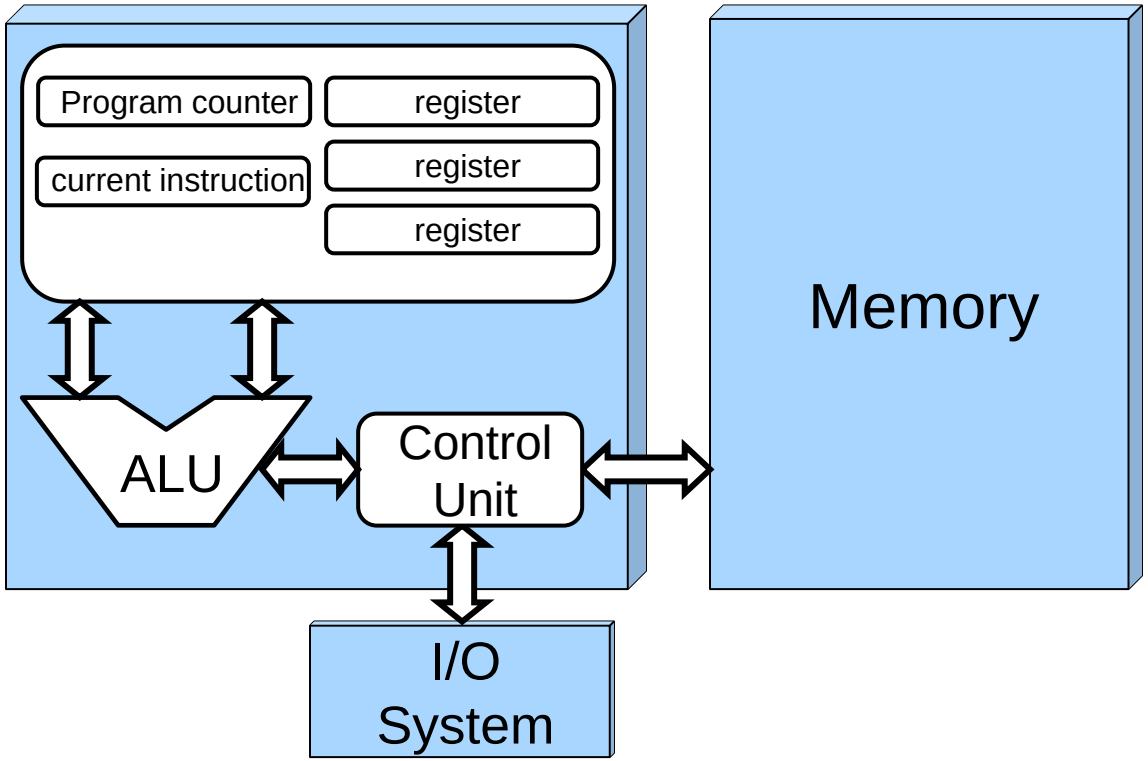
# Revisit of Computer Systems




How to understand software-hardware interface?

# Let’s Revisit What We Have Learned!

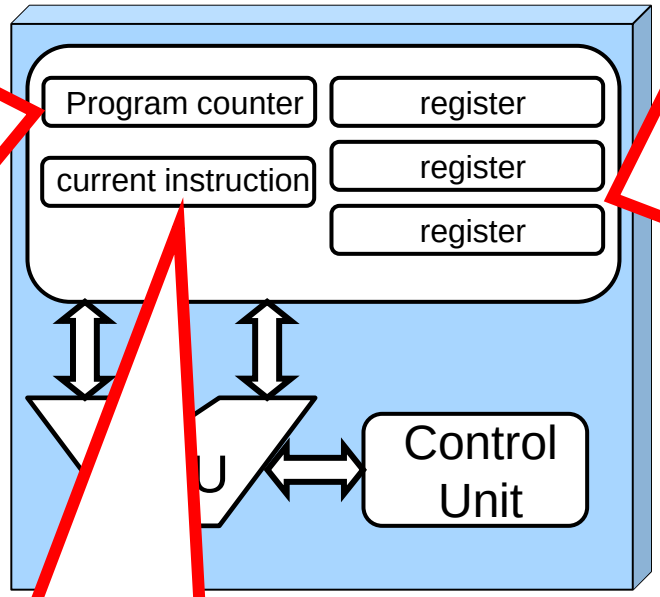
# What is in the CPU?



# What is in the CPU?

**Program Counter:** Points to the next instruction

Special register that contains the address in memory of the next instruction that should be executed (gets incremented after each instruction, or can be set to whatever value whenever there is a change of control flow)

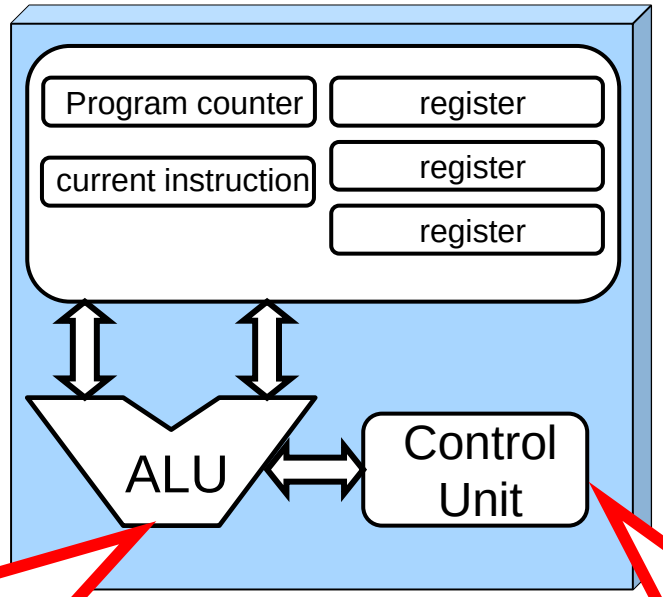



**Registers:** the "variables" that hardware instructions work with

Data can be loaded from memory into a register
Data can be stored from a register back into memory
Operands and results of computations are in registers
Accessing a register is really fast
There is a limited number of registers

**Current Instruction:** Holds the instruction that's currently being executed

# What is in the CPU?




**Arithmetic and Logic Unit:** what you do computation with

Used to compute a value based on current register values and store the result back into a register

+, *, /, -, OR, AND, XOR, etc.

**Control Unit:** Decodes instructions and make them happen

Logic hardware that decodes instructions (i.e., based on their bits) and sends the appropriate (electrical) signals to hardware components in the CPU

# Software-Hardware Collaboration in CPU

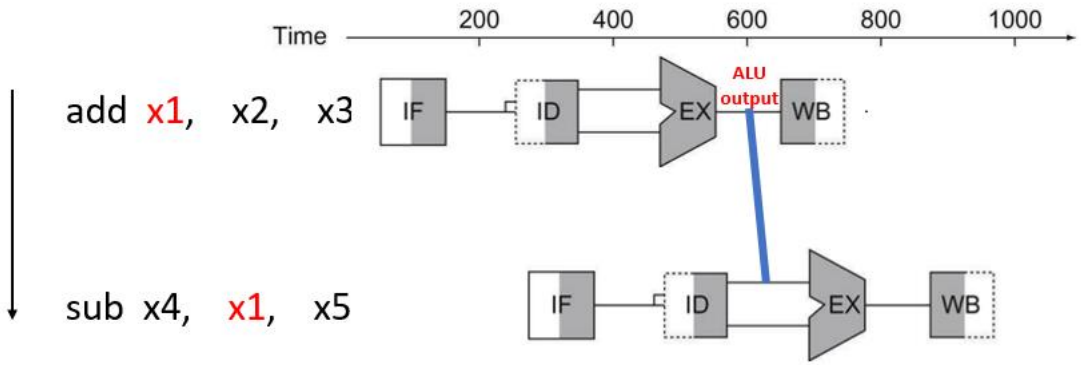
- Can we find some examples of software-hardware interface in CPU design?
- Yes! We have learned some collaborations between software and hardware in CPU design

Solutions for Hazards!

# Software-Hardware Collaboration in CPU

- Data hazards

## Forwarding




## Code reordering

**Consider the following code segment in C:**

a = b + e;
c = b + f;

- Assuming all variables are in memory
- and are addressable as offsets from x31

**The generated RISC-V code:**

```
ld  x1, 0(x31) // Load b
ld  x2, 8(x31) // Load e
add  x3, x1, x2 // b + e
sd  x3, 24(x31) // Store a
ld  x4, 16(x31) // Load f
add  x5, x1, x4 // b + f
sd  x5, 32(x31) // Store c
```

eliminates both hazards

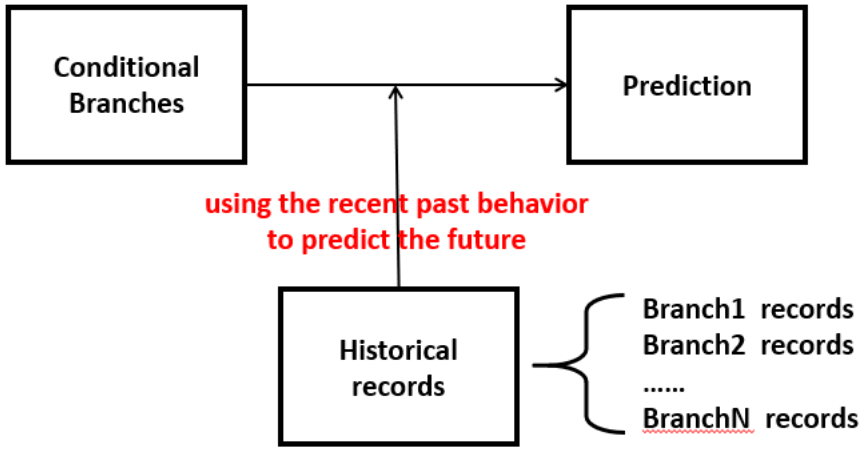
```
ld  x1, 0(x31)
ld  x2, 8(x31)
ld  x4, 16(x31)
add  x3, x1, x2
sd  x3, 24(x31)
add  x5, x1, x4
sd  x5, 32(x31)
```

Hardware ⟷ Collaboration ⟷ Software

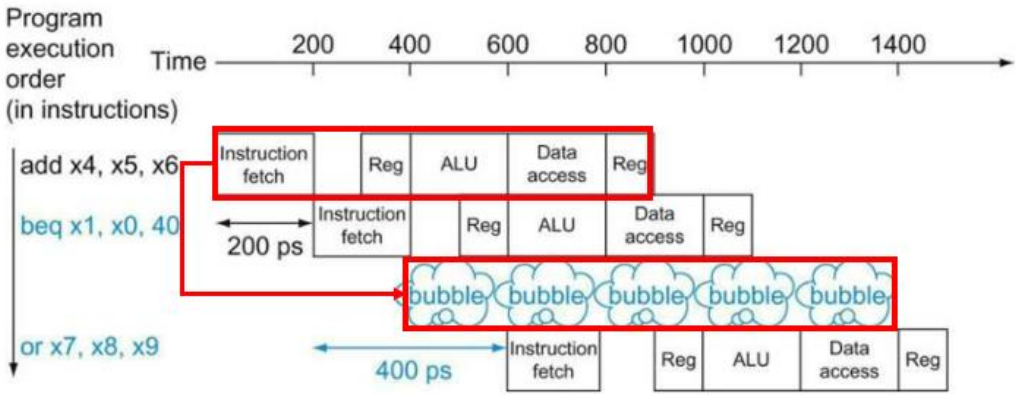
# Software-Hardware Collaboration in CPU

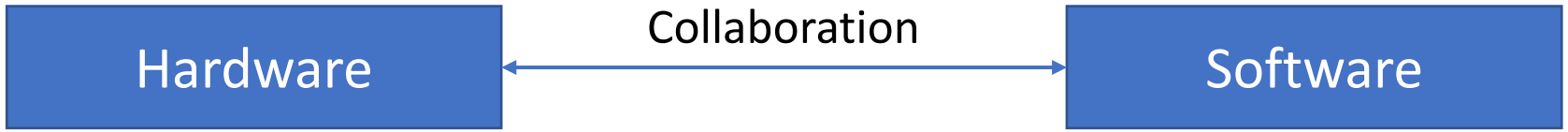
- Control hazards

Prediction




Code Scheduling



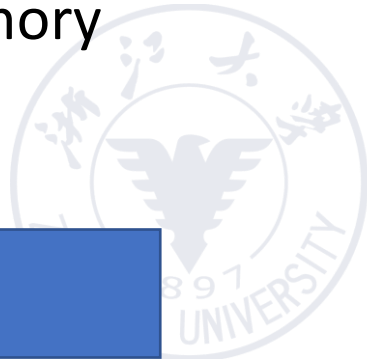
# Any other collaboration?

Let’s continue to revisit CPU working principles

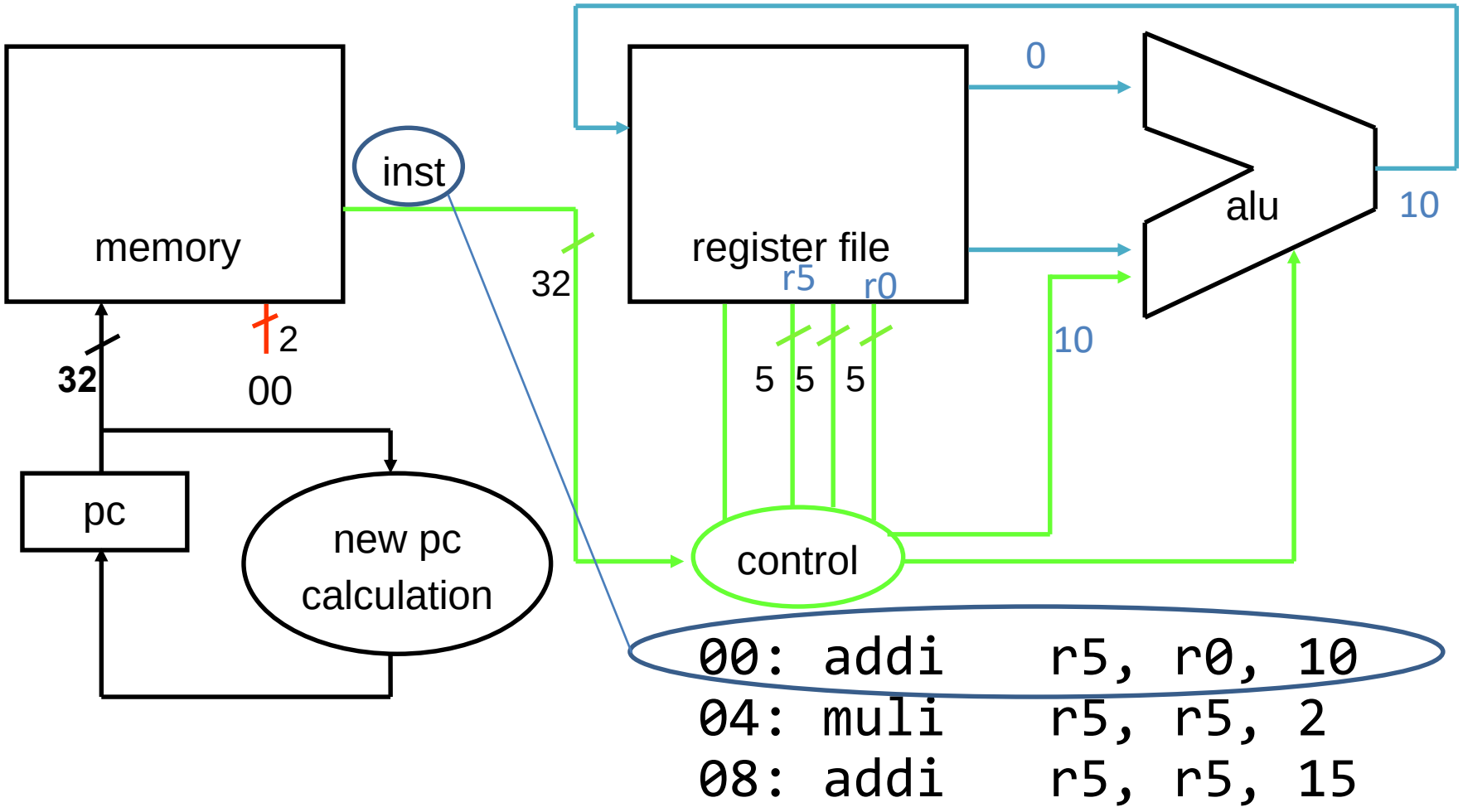
# What we do using a CPU?

- We have some bytes stored at precise location in a memory
  - e.g., integers, ASCII codes of characters, floating points numbers, RGB values
  - e.g., instructions that specify what to do with the data; when you buy a processor, the vendor defines the instruction set (e.g., instruction “0010 1101” means “increment some useful counter”)
- Obviously, these bytes should be processed to realize our purposes
- The CPU is the piece of hardware that modifies these bytes
  - In fact, one can really think of the CPU as a device that moves one memory state (i.e., all the stored content) to another memory state (some new, desired stored content)

What does a CPU really do in a computer system?

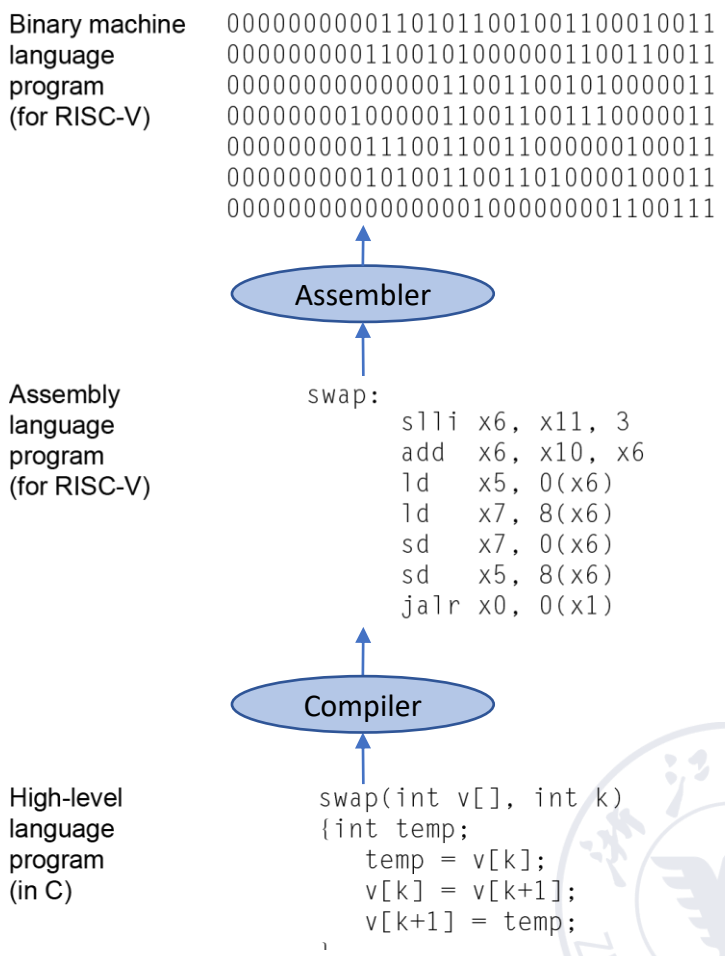


# Hardware Behaviours

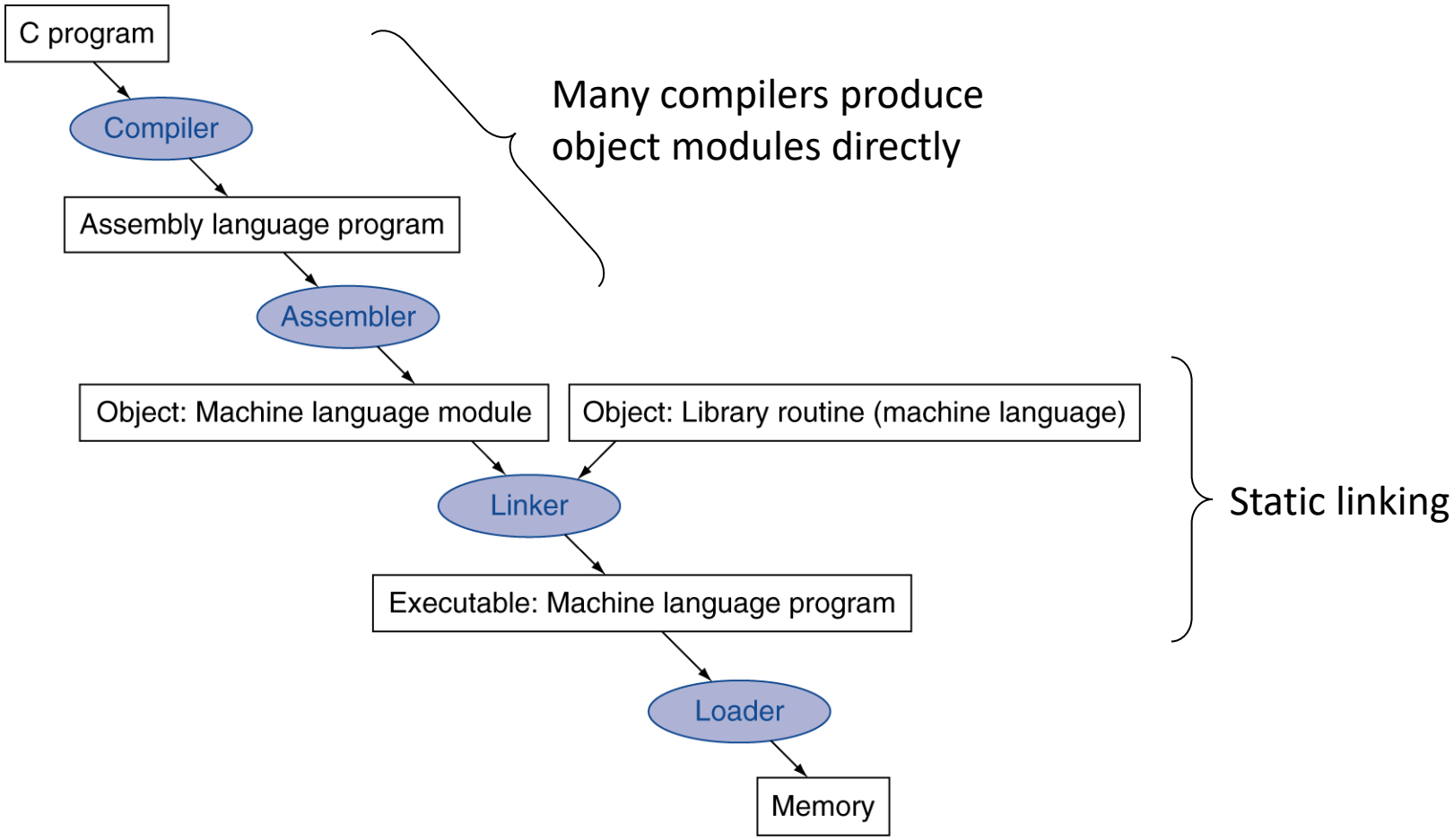


# Levels of Program Code

- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data
- Assembly language
  - Textual representation of instructions
- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability



# Closer Look at Program Translation



# Reflect the first Hello World!

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        printf("Hello World!\n");

        return 0;
}
```

```
➜  example1 gcc helloworld.c
➜  example1 ./a.out
Hello World!
```

- **About a.out**
  - **Let's take a look!**

# Reflect the first Hello World!

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        printf("Hello World!\n");

        return 0;
}
```

```
➜  example1 gcc helloworld.c
➜  example1 ./a.out
Hello World!
```

- **How does it work?**
  - Why we need to compile the program?
  - What is in an executable file?
  - How to execute a program?
  - What is Operating System (OS)? What does OS do?
  - How does the underlying hardware support the software?

# Source Code -> Executable File

C

```
int x = 10;
x = 2 * x + 15;
```

compiler

RISC-V assembly language

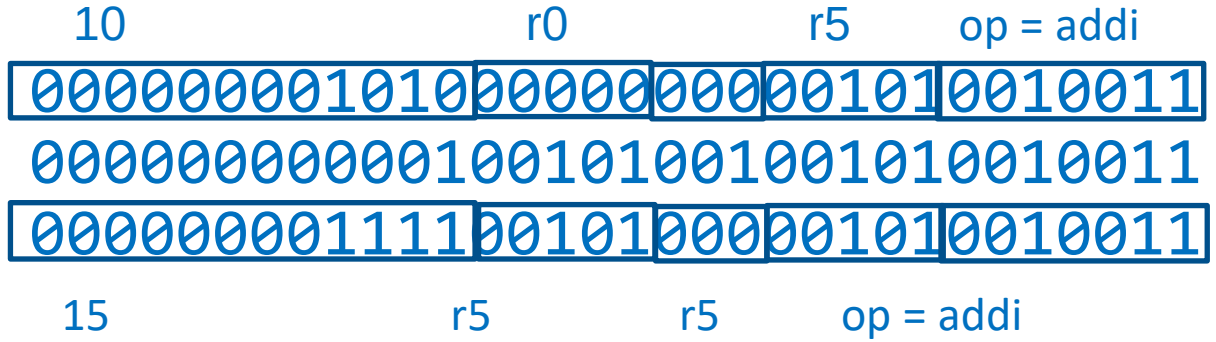
```
addi   r5, r0, 10
muli   r5, r5, 2
addi   r5, r5, 15
```

```
r0 = 0
r5 = r0 + 10
r5 = r5 * 2
r5 = r5 + 15
```

***Everything is a number!***

assembler

RISC-V machine language



# Assembler Pseudo-instructions

- Most assembler instructions represent machine instructions one-to-one

- Pseudo-instructions: figments of the assembler's imagination

```
mov x5, x6          →   add x5, x0, x6

blt x5, x6, L       →   slt  x1, x5, x6
                        bne x1, x0, L
```

  - x1 (register 1): assembler temporary

# Producing an Object Module

- Assembler (or compiler) translates program into machine instructions

- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

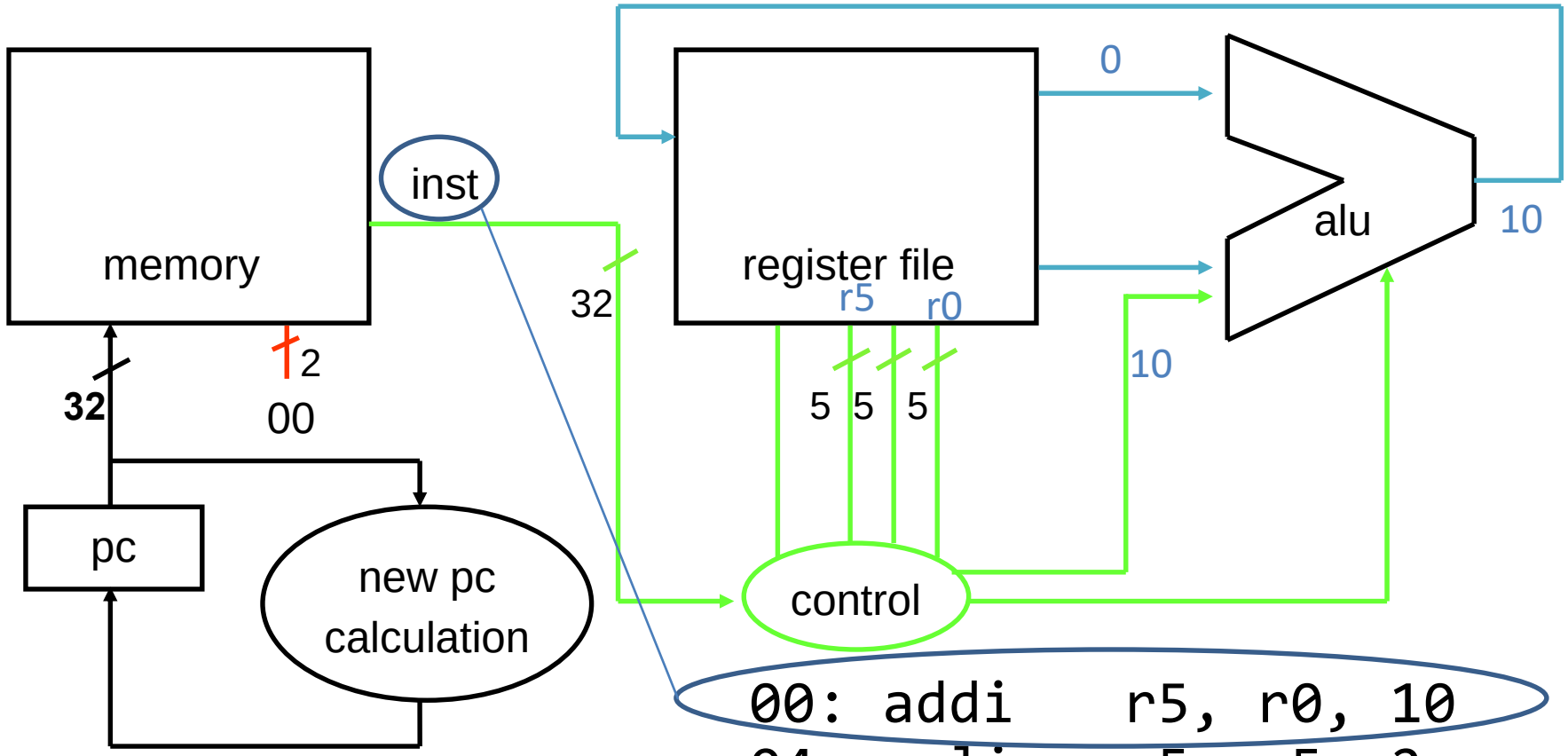
# Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space
- Dynamic Linking: only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

# Loading a Program

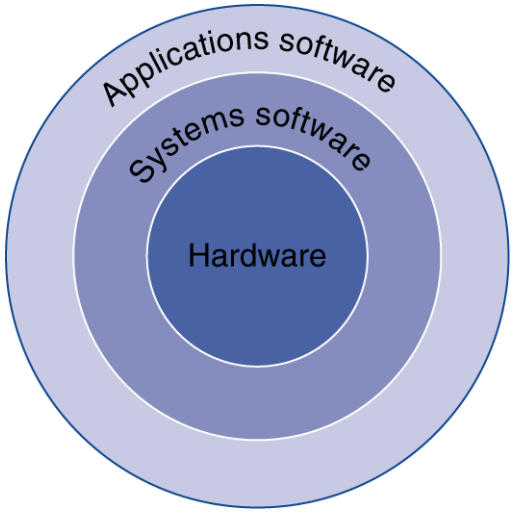
- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including x2, x3)
  6. Jump to startup routine
     - Copies arguments to x10, … and calls main
     - When main returns, do exit syscall

# Executing Program




Seems separated, where is the interface?

# Below Your Program




- Application software
  - Written in high-level language
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
- Hardware
  - Processor, memory, I/O controllers

"Hello World" (Software)

Software-Hardware Interface!

CPU (Hardware)

Here comes OS in the following courses!

# Brief Introduction of OS

# What is an Operating System (OS)?

- A program that acts as an intermediary between a user of a computer and the computer hardware
- Operating system goals:
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner

# What Operating Systems Do

- Depends on the point of view
- Users want convenience, **ease of use** and **good performance**
  - Don’t care about **resource utilization**
- But shared computer such as **mainframe** or **minicomputer** must keep all users happy
- Users of dedicate systems such as **workstations** have dedicated resources but frequently use shared resources from **servers**
- Handheld computers are resource poor,  optimized for usability and battery life
- Some computers have little or no user interface, such as embedded computers in devices and automobiles

# Operating System Definition

- OS is a **resource allocator**
  - Manages all resources
  - Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
  - Controls execution of programs to prevent errors and improper use of the computer

# Operating System Definition (Cont.)

- No universally accepted definition
- “Everything a vendor ships when you order an operating system” is a good approximation
  - But varies wildly
- “The one program running at all times on the computer” is the **kernel**
- Everything else is either
  - a system program (ships with the operating system) , or
  - an application program

# Operating System Structure

- **Multiprogramming** (**Batch system**) needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via **job scheduling**
  - When it has to wait (for I/O for example), OS switches to another job
- **Timesharing** (**multitasking**) is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory → **process**
  - If several jobs ready to run at the same time → **CPU scheduling**
  - If processes don't fit in memory, **swapping** moves them in and out to run
  - **Virtual memory** allows execution of processes not completely in memory

# Operating-System Operations

- **Interrupt driven** (hardware and software)
  - Hardware interrupt by one of the devices
  - Software interrupt (**exception** or **trap**):
    - Software error (e.g., division by zero)
    - Request for operating system service
    - Other process problems include infinite loop, processes modifying each other or the operating system

# Operating-System Tasks

- Process Management
- Memory Management
- Storage Management
- I/O Subsystem
- Protection and Security

# Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a **passive entity**, process is an **active entity**
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one **program counter** specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads

# Memory Management

- To execute a program all (or part) of the instructions must be in memory
- All  (or part) of the data that is needed by the program must be in memory.
- Memory management determines what is in memory and when
  - Optimizing CPU utilization and computer response to users
- Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed

# Storage Management

- OS provides uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit - **file**
  - Each medium is controlled by device (i.e., disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
  - Files usually organized into directories
  - Access control on most systems to determine who can access what
  - OS activities include
    - Creating and deleting files and directories
    - Primitives to manipulate files and directories
    - Mapping files onto secondary storage
    - Backup files onto stable (non-volatile) storage media

# I/O Subsystem

- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
  - Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
  - General device-driver interface
  - Drivers for specific hardware devices

# Protection and Security

- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** – defense of the system against internal and external attacks
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
  - User identities (**user IDs**, security IDs) include name and associated number, one per user
  - User ID then associated with all files, processes of that user to determine access control
  - Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
  - **Privilege escalation** allows user to change to effective ID with more rights