Embedded Intelligent System and Novel Computer Architecture

Lecture 01 – Essential of Computer System

Pengju Ren
Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University

http://gr.xjtu.edu.cn/web/pengjuren

Pengju Ren@XJTU 2022

Course Administration

Instructor: Pengju Ren

Lectures: Two 100-minutes lectures a week

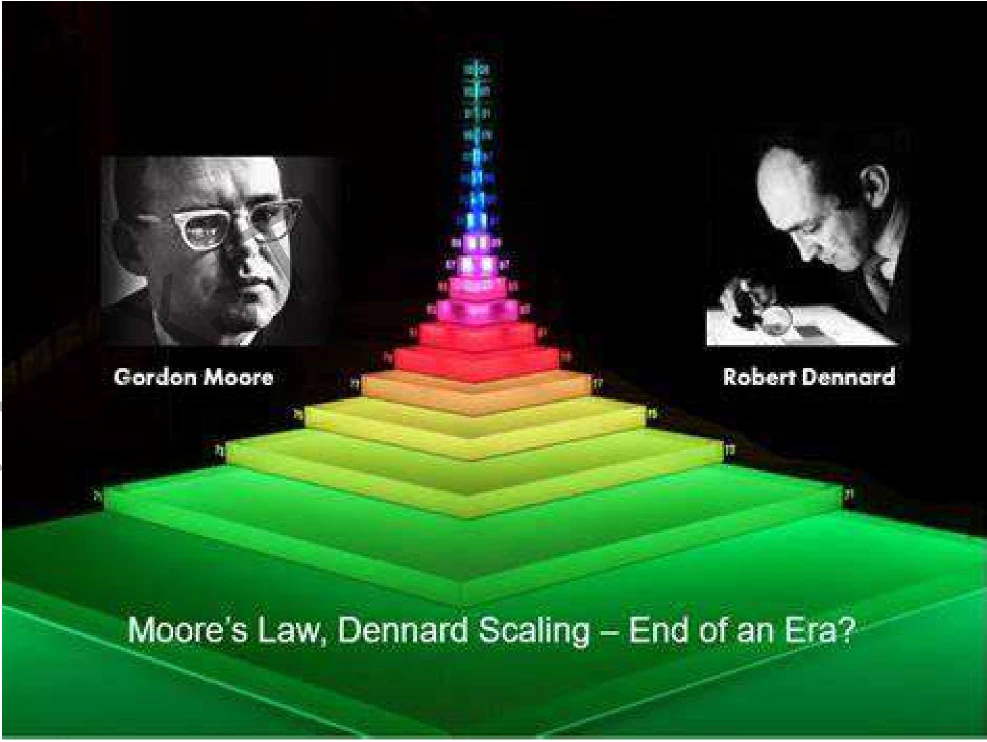
Ref Textbook: Computer System: A Programmer's Perspective (Third's Edition, CS:APP)《深入理解计算机系统》**(第三版)**

Prerequisite: Digital System Structure, Computer Architecture

Pengju Ren@ XJTU 2022

Highly Revolved Semiconductor Industry

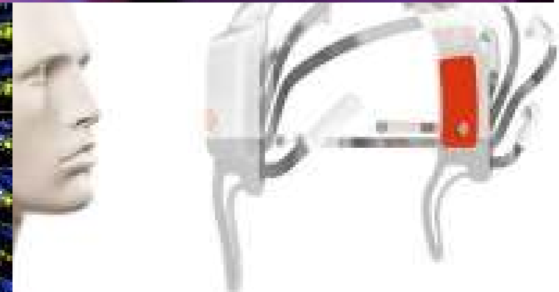
1957 v.s 2021



10^9 Transistors @1mm@5nm

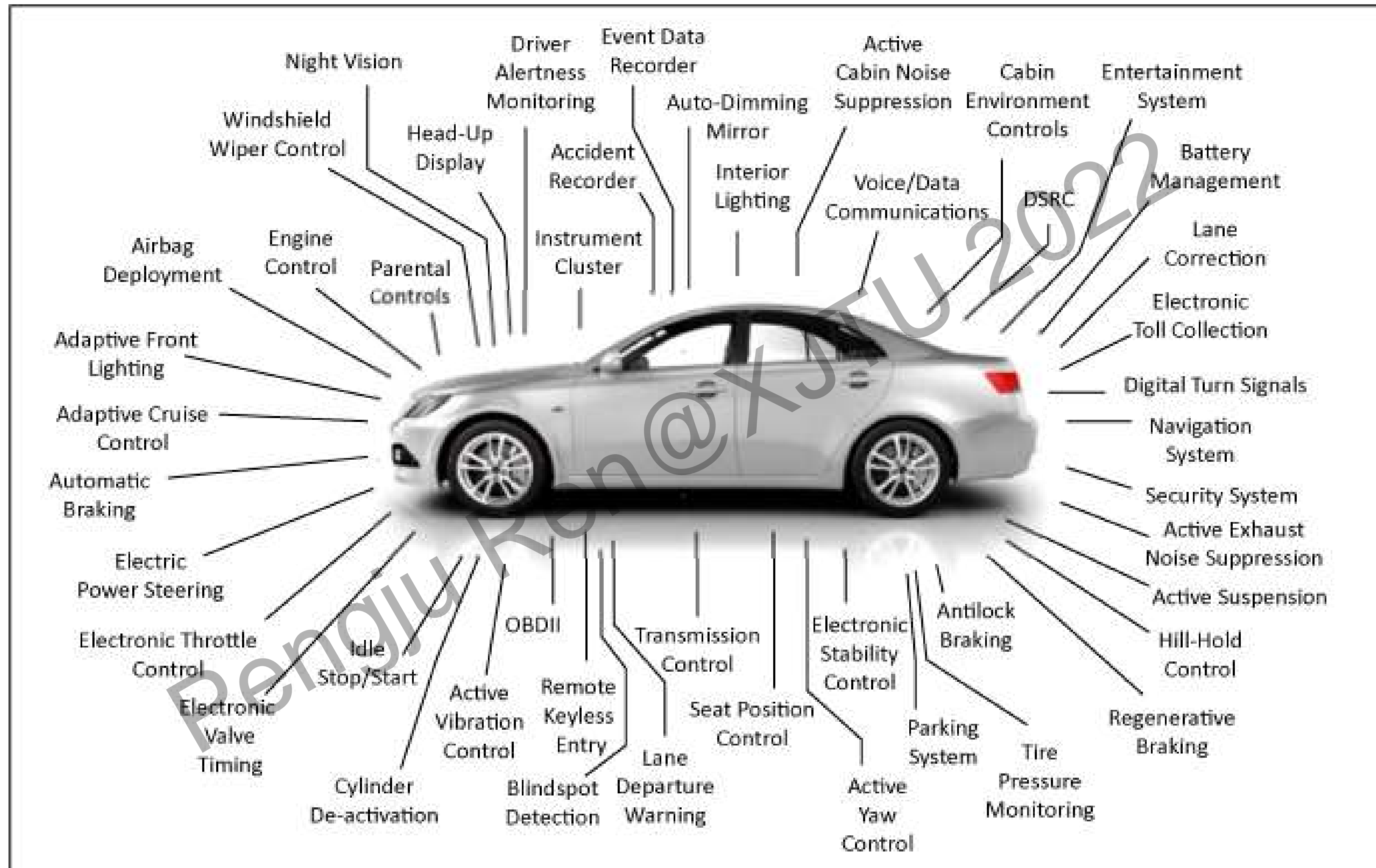
Pengju Ren

Different Platforms, Different Goals



Modern computing is as much about enhancing capabilities as data processing!

How many Chips in one Car?



Pengju Ren@XJTU 2022

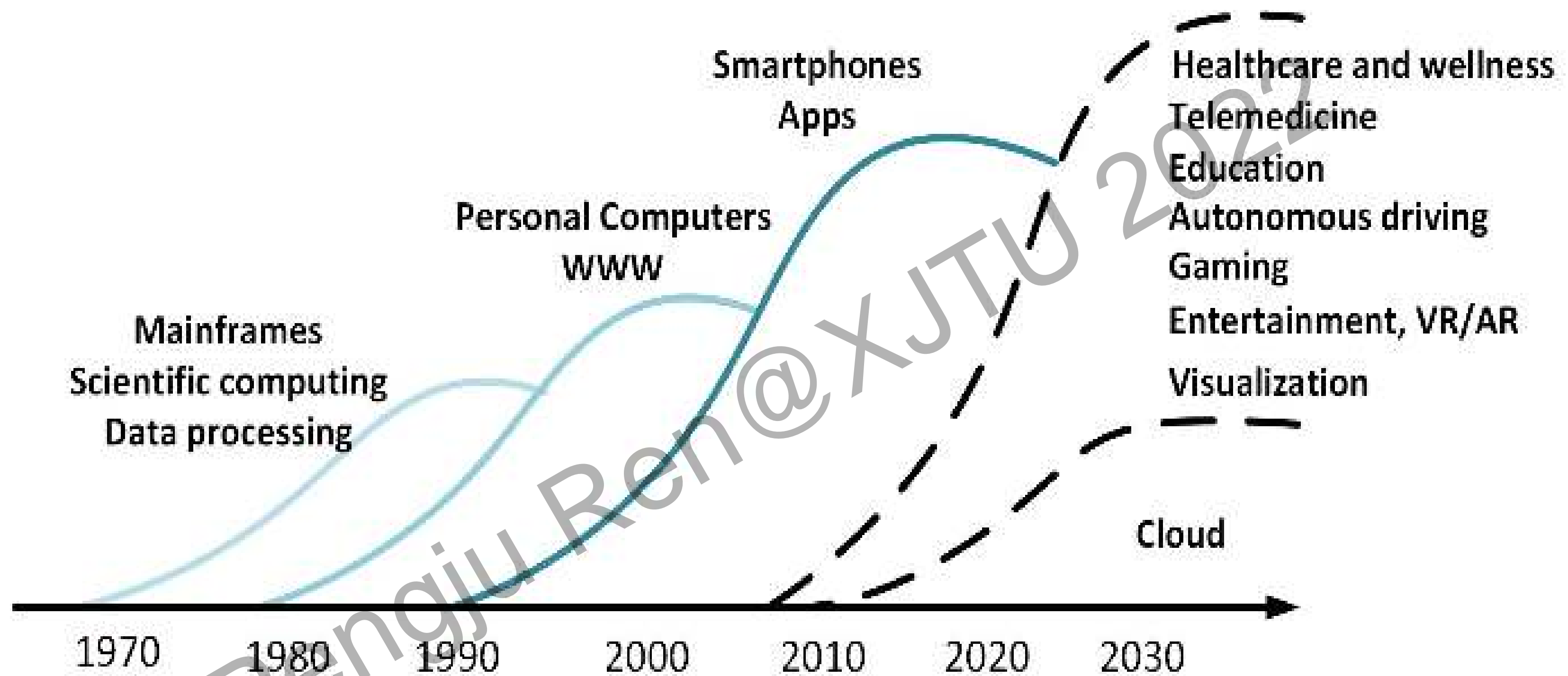
- 500~600/car for Traditional One
- >1000/car for new energy vehicles

How many Chips in a smart Phone?

Pengju Ren@XJTU 2022

114 Chips @ Redmi note10 Pro

The trends of applications

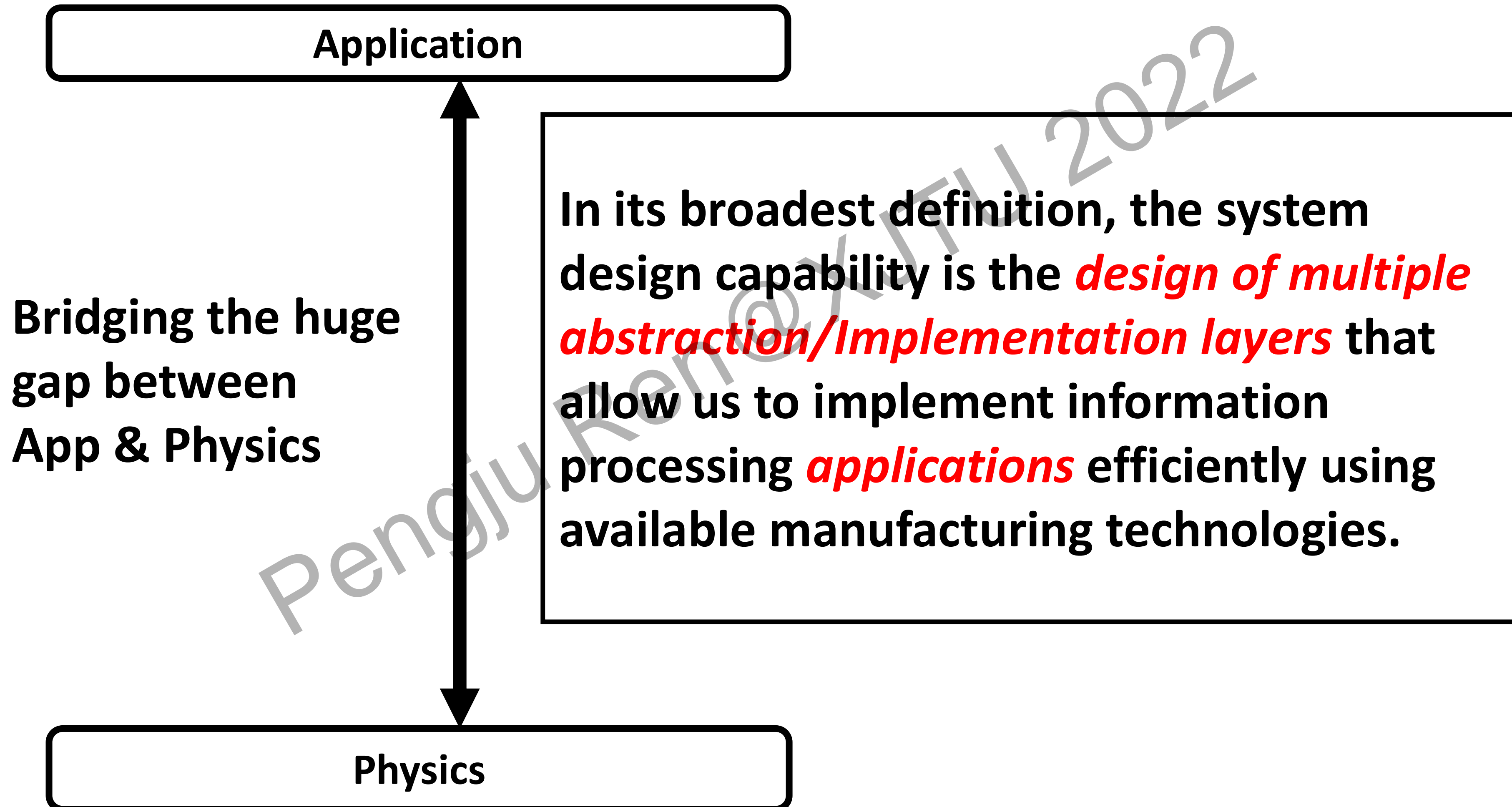


Pengju Ren@XJTU 2022

Number of deployed devices continues growing, but which is the killer app in the future (Robotics ? Healthcare ? Meta Universe?)

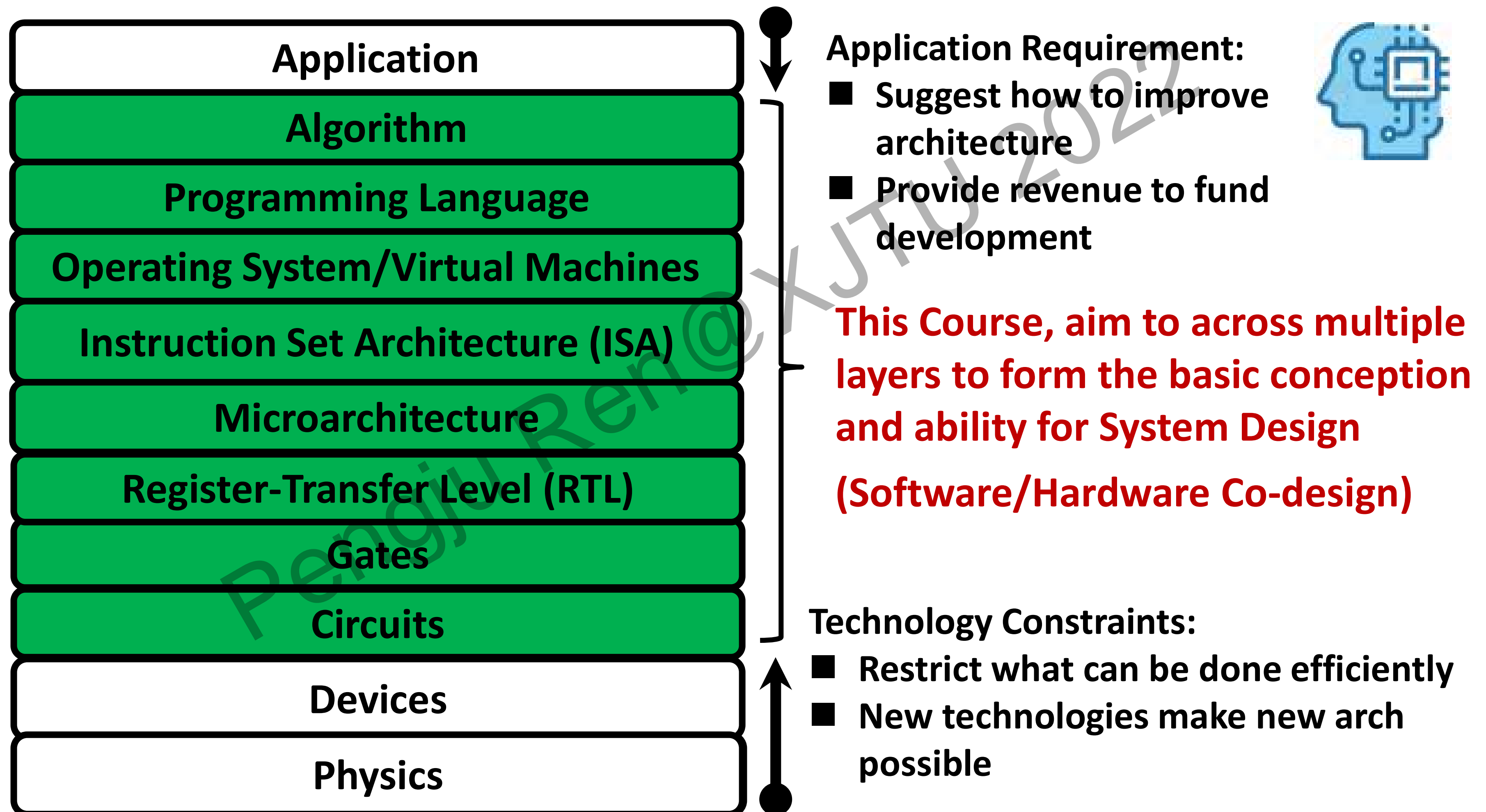
- **Diversification of needs, therefore various architectures**

The perspective of this Course (A system view)



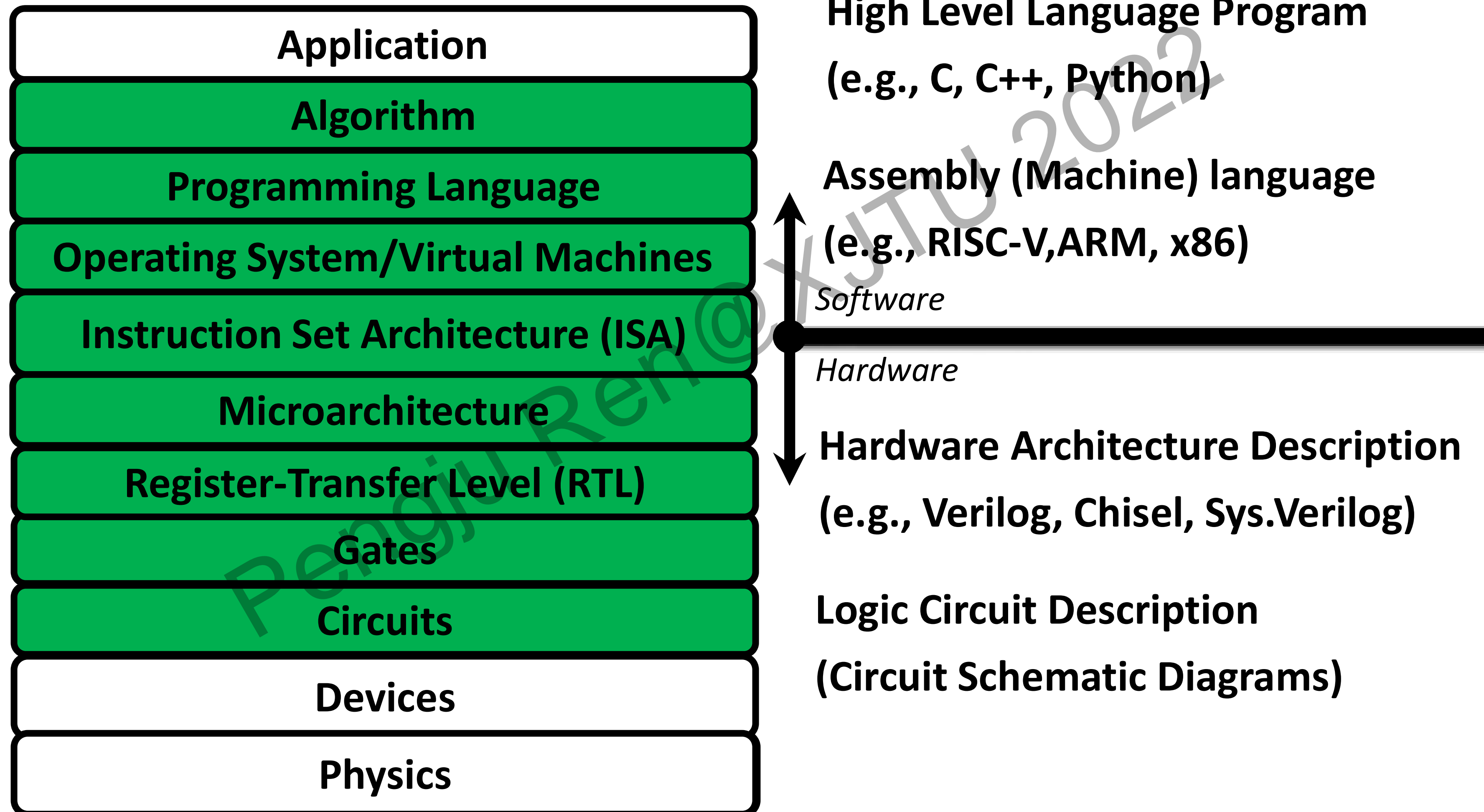
Pengju Ren@XJTU 2022

The perspective of this Course (A system view)



Pengju Ren@XJTU 2022

The perspective of this Course (A system view)



Pengju Ren@XJTU 2022

Three key goals of this Course

- To understanding how a processor works underneath the software layer and <u>how decisions made in hardware affect software/programmer and vice versa</u>

"*Somehow a program ends up executing as digital logic*"

- To enable you to be comfortable in making design and optimization decisions that <u>cross the boundaries of different layers and system components</u>
- Allows you to be a better programmer or hardware architecture designer or <u>master of both</u>

Pengju Ren@XJTU 2022

What you will learn from this Course

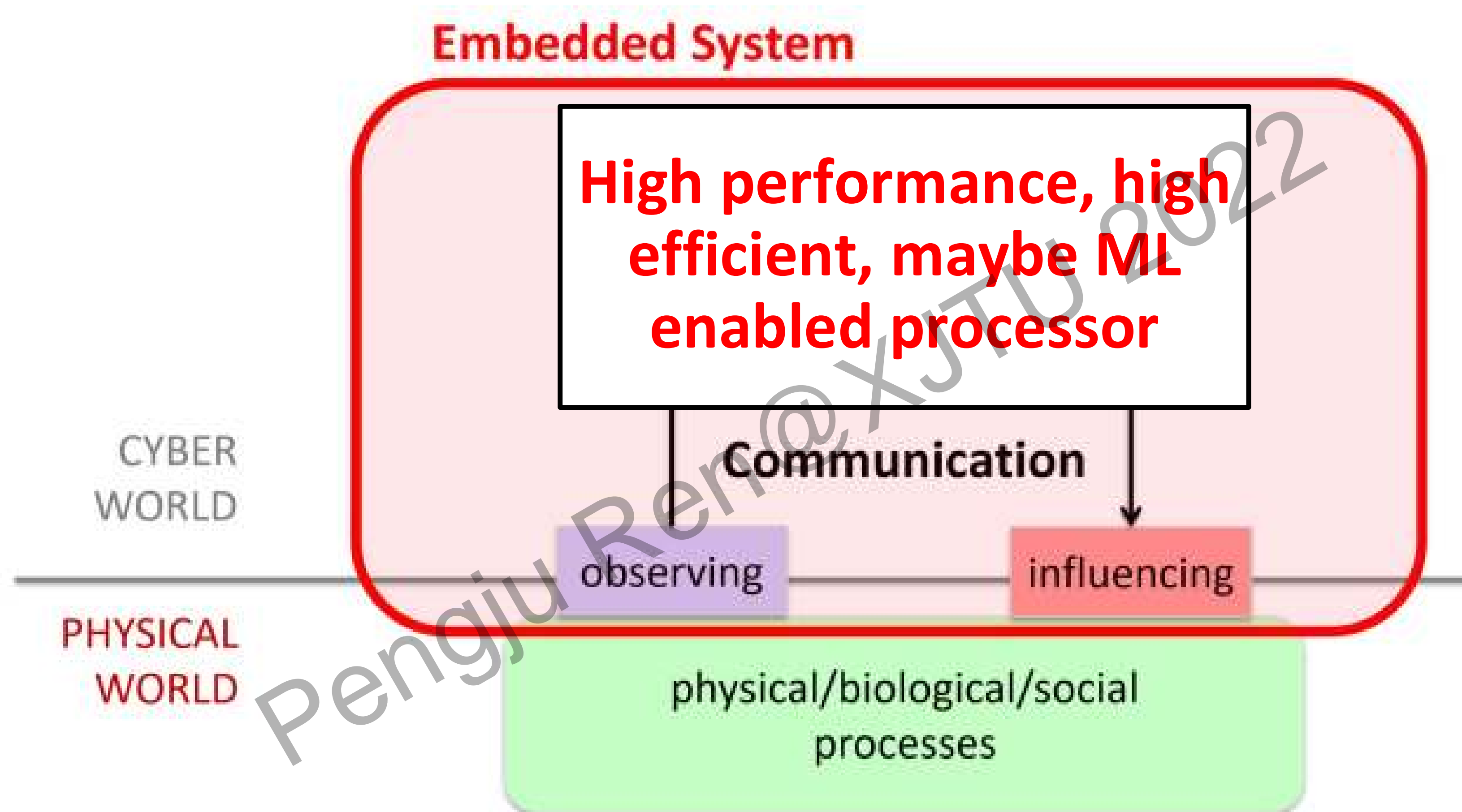
- Fundamental rules of Computer System Design (Amdahl's law, Roofline Model, Dark Silicon …)
- Memory (Regs, Cache, Scratchpad, Virtual Memory, DRAM)
- Instruction Level Parallel (Pipeline, SuperScalar & VLIW, OoO)
- Data Level Parallel SIMD
- Thread Level Parallel (Multicore/SMT/GPGPU)
- Mixed Arch (Manycore Architecture & NoC & (CC and SC))
- Data-stream Arch: (Systolic Array, for DFT, CONV, Pattern Matching, Vector Matrix Multiplier, Matrix-Matrix Multiplier, …)
- Coarse Grained Reconfigurable Architecture (CGRA)
- Field Program Gate Array (FPGA)
- DNN Processor (TPU, Eyeriss)
- Neuromorphic Computing (TureNorth, Loihi and Tianjic)

Pengju Ren@XJTU 2022

Let’s consider the most widely used computer —— Embedded Intelligent systems

Pengju Ren@XJTU 2022

Embedded Intelligent Systems



Pengju Ren@XJTU 2022

Use feedback to influence the dynamics of the Physical world by taking smart decisions in the cyber

Embedded Intelligent Systems

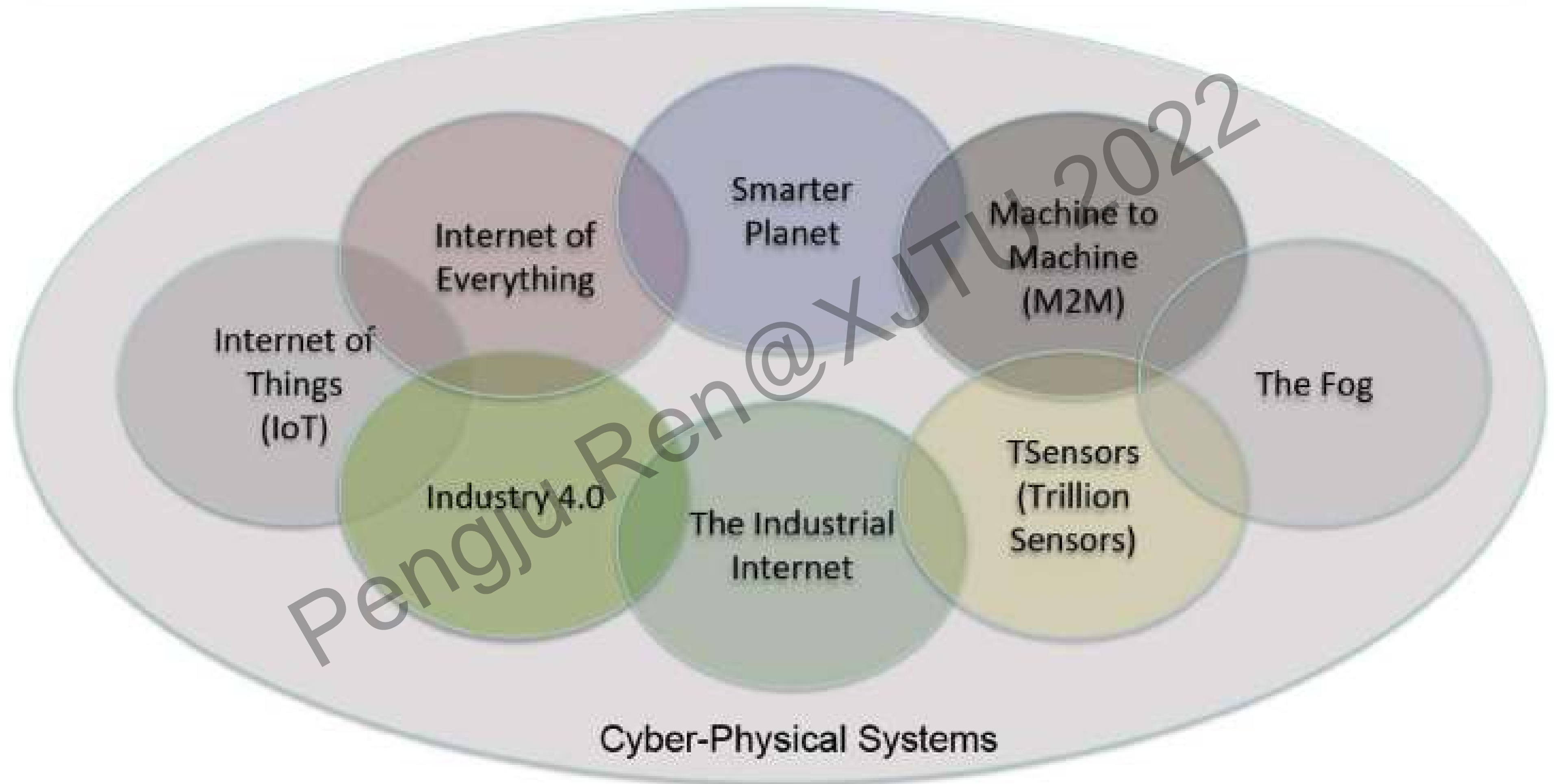
Embedded intelligent system: a machine that has an embedded, Internet-connected computer which can gather and analyze data and communicate with other systems.

Examples:

Pengju Ren@XJTU 2022

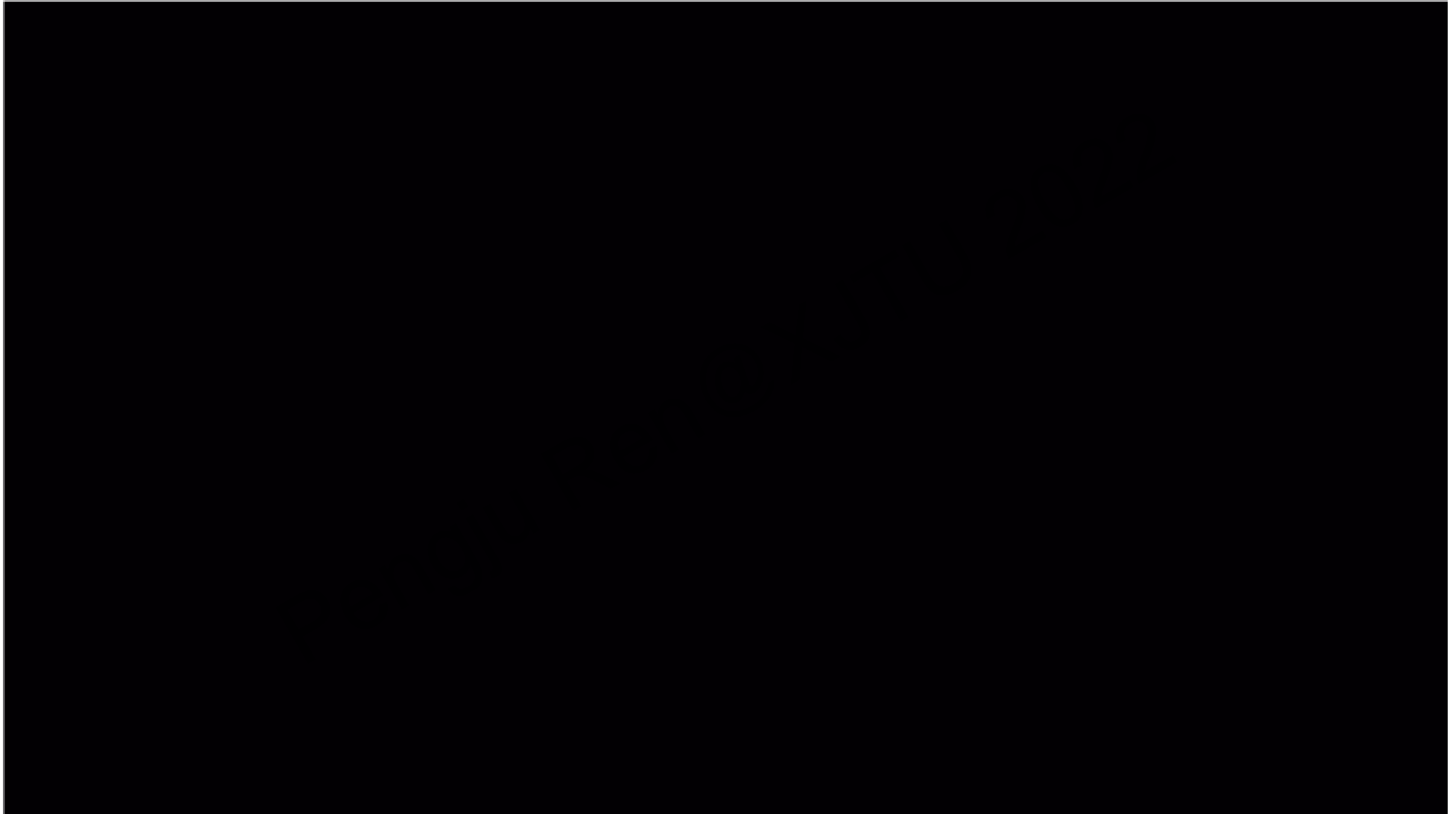
These systems have the ability to learn from experience, security, connectivity, the capacity for remote monitoring and management, and can adapt themselves according to current data.

Many Names – Similar Meanings



Pengju Ren@XJTU 2022

Creating Better Possibilities



ARM power a very wide range of applications
95% of domestic SoCs are based on ARM technology

arm

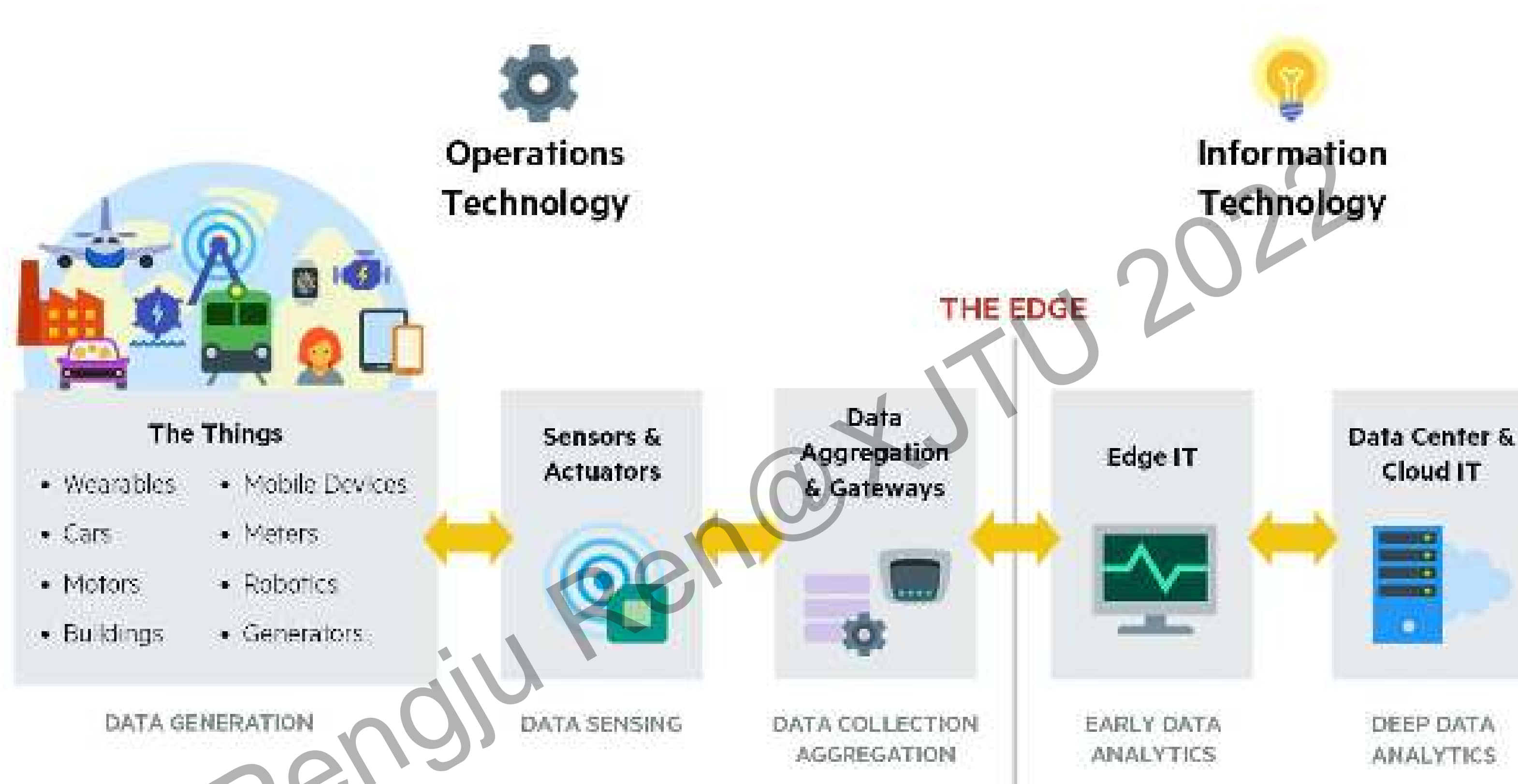
"That next big thing is what I look out for constantly and I expect it's the same for you reading this article: **AI-enhanced intelligent machines** anchored in stronger security, delivering greater connectivity and specialized processing power that will unlock new levels of possibilities. The compute power that will transform businesses and societies. Our purpose is to Spark the World's potential; to put the building blocks, systems, and design tools in your hands, and to support your creativity. I can't wait to see what you come up with next."

——CEO of ARM Ltd

毫无疑问，Arm是芯片产业最具影响力的全球标杆。

Rengju Ren@XJTU 2022

From Edge to Cloud (ARM series)



Pengju Ren@XJTU 2022

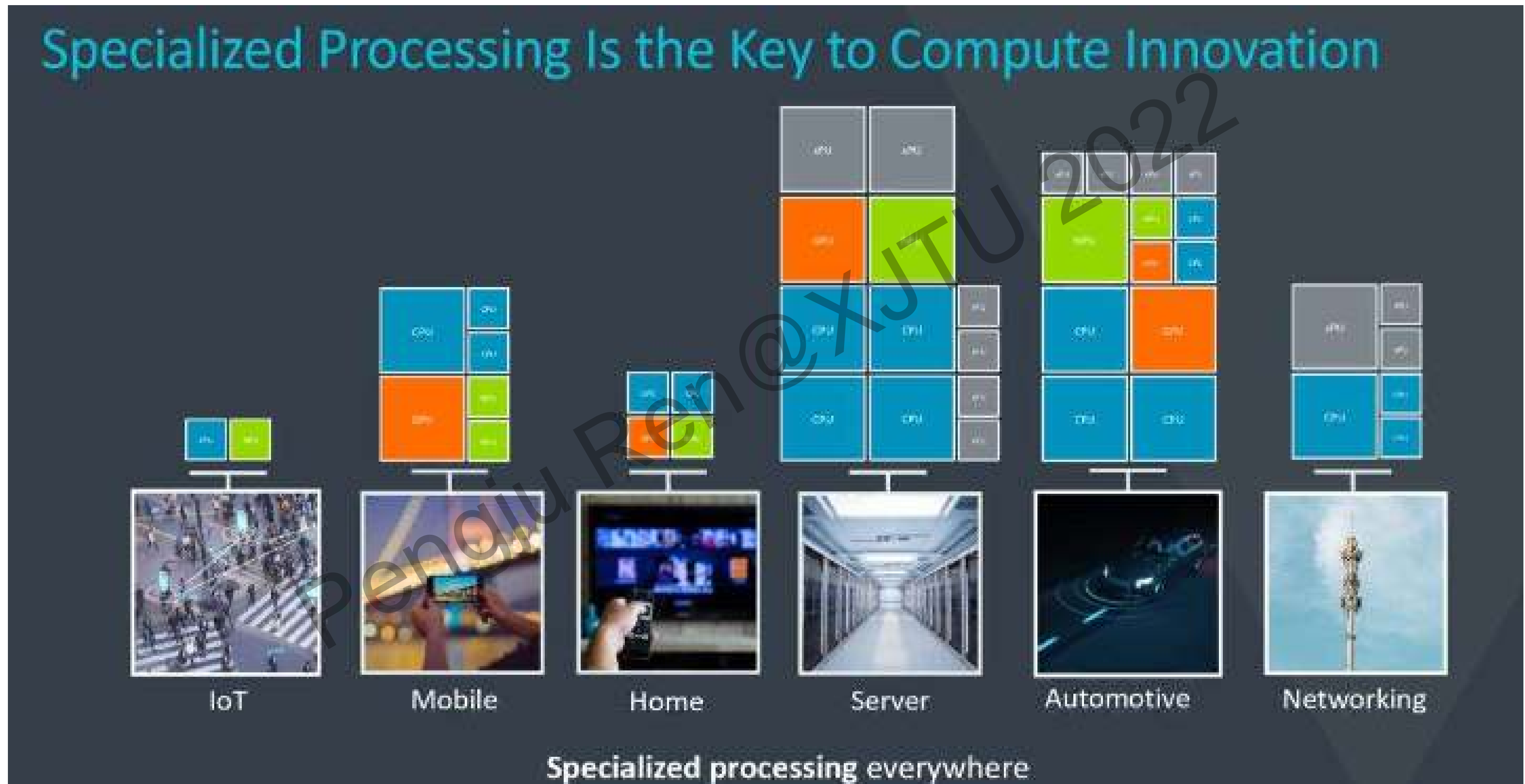
ARM Cortex (Mobile CPU)

- **X-Series: (Performance-first)**
- **A-Series: (Area and power efficiency-first)**
- **R-Series: (Real-time)**
- **M-Series: (Microcontroller)**

ARM Neoverse (From Infrastructure Servers to Edge)

- **V-Series: Maximum (Wider/Deeper)**
- **N-Series: Scale out (Balanced)**
- **E-Series: Efficient (Efficient)**

Heterogeneous Manycore Arch



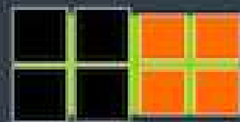
Pengju Ren@XJTU 2022

from Sensors to Smartphones to Servers

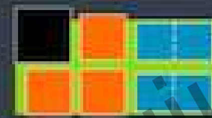
Big.Little for CPU Cluster

The DSU-110 is the Backbone of the Armv9 CPU Cluster

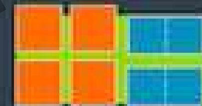
Uncompromised performance and efficiency



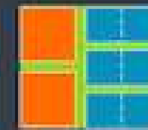
4X+4b



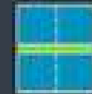
1X+3b+4L



4b+4L

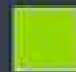


2b+6L



4L

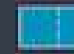
arm DynamIQ Scalability across Client segments



DSU-110

Cortex-X2

Cortex-A710



Cortex-A510

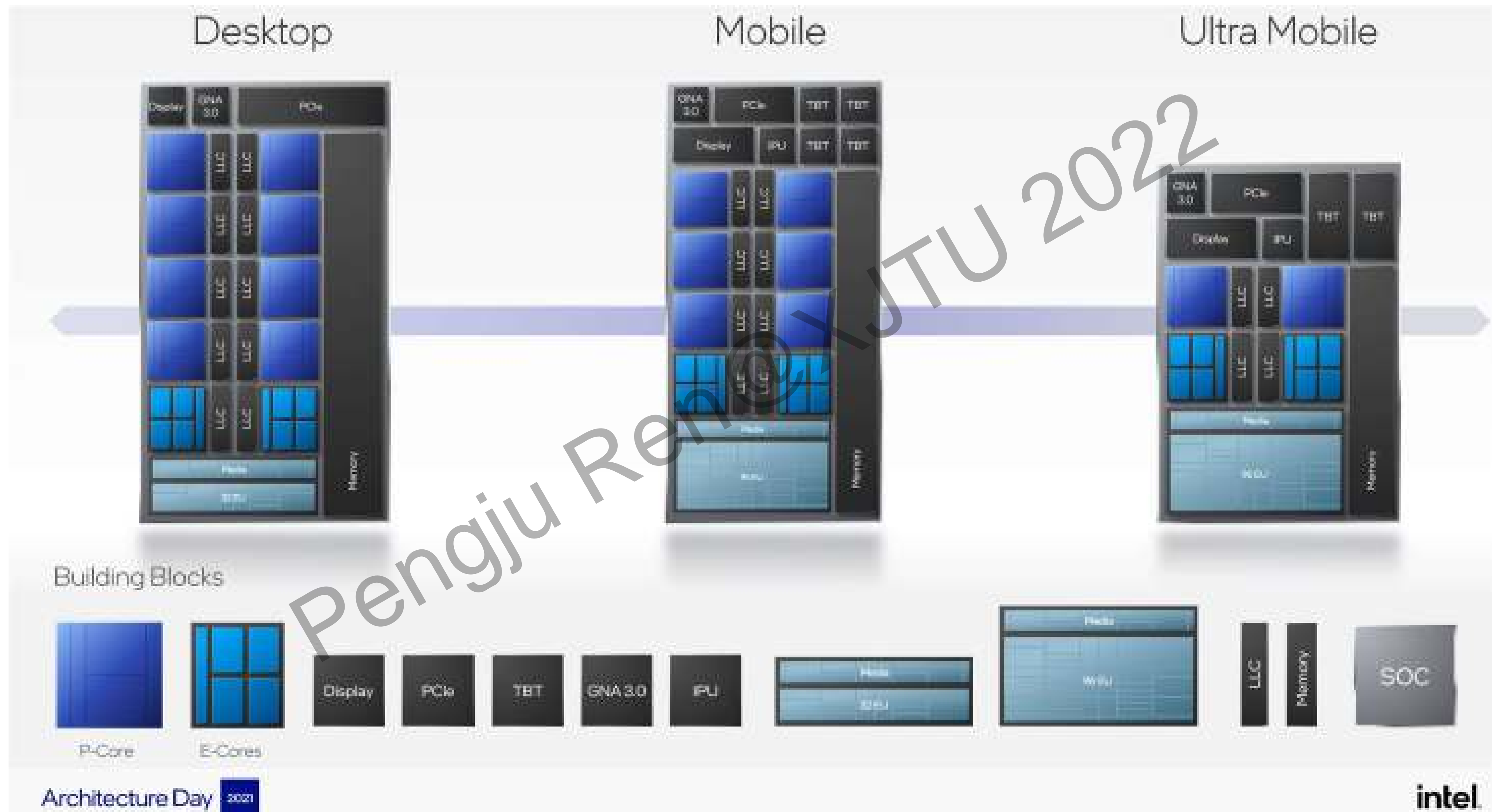
Examples only; other configurations possible

© 2021 Arm Limited

arm

Pengju Ren@XJTU 2022

Intel follows the same Paradigm (Heterogeneity and Big.Little)



Pengju Ren@XJTU 2022

Heterogeneity (Asymmetry) → Specialization

- **Idea:** Instead of having multiple instances of a "resource" to be the same (i.e., homogeneous or symmetric), design some instances to be different (i.e., heterogeneous or asymmetric)
- Heterogeneity and asymmetry have the same meaning
  - Contrast with homogeneity and symmetry
- Heterogeneity is a very general system design concept (and *life* concept, as well)

- Different instances can be optimized to be more efficient in executing *different types of workloads* or satisfying *different requirements/goals*
  - Heterogeneity enables specialization/customization

Pengju Ren@XJTU 2022

Asymmetry Enables Customization

C	C	C	C
C	C	C	C
C	C	C	C
C	C	C	C

Symmetric

C1 C2 C3
C4 C4 C4 C4
C5 C5 C5 C5

Asymmetric

Pengju Ren@XJTU 2022

- **Symmetric: One size fits all**
  - **Energy and performance suboptimal for different "workload" behaviors**
- **Asymmetric: Enables customization and adaptation**
  - **Processing requirements vary across workloads (applications and phases)**
  - **Execute code on best-fit resources (minimal energy, adequate perf.)**

Why Asymmetry in Design? (I)

- **Different workloads executing in a system can have different behavior**
  - **Different applications can have different behavior**
  - **Different execution phases of an application can have different behavior**
  - **The same application executing at different times can have different behavior (due to input set changes and dynamic events)**
  - **E.g., *locality, predictability of branches, instruction-level parallelism, data dependencies, serial fraction in a parallel program, bottlenecks in parallel portion of a program, interference characteristics, …***

- **Systems are designed to satisfy different metrics at the same time**
  - **There is almost never a single goal in design, depending on design point**
  - **E.g., *Performance, energy efficiency, fairness, predictability, reliability, availability, cost, memory capacity, latency, bandwidth, …***

Pengju Ren@XJTU 2022

Why Asymmetry in Design? (II)

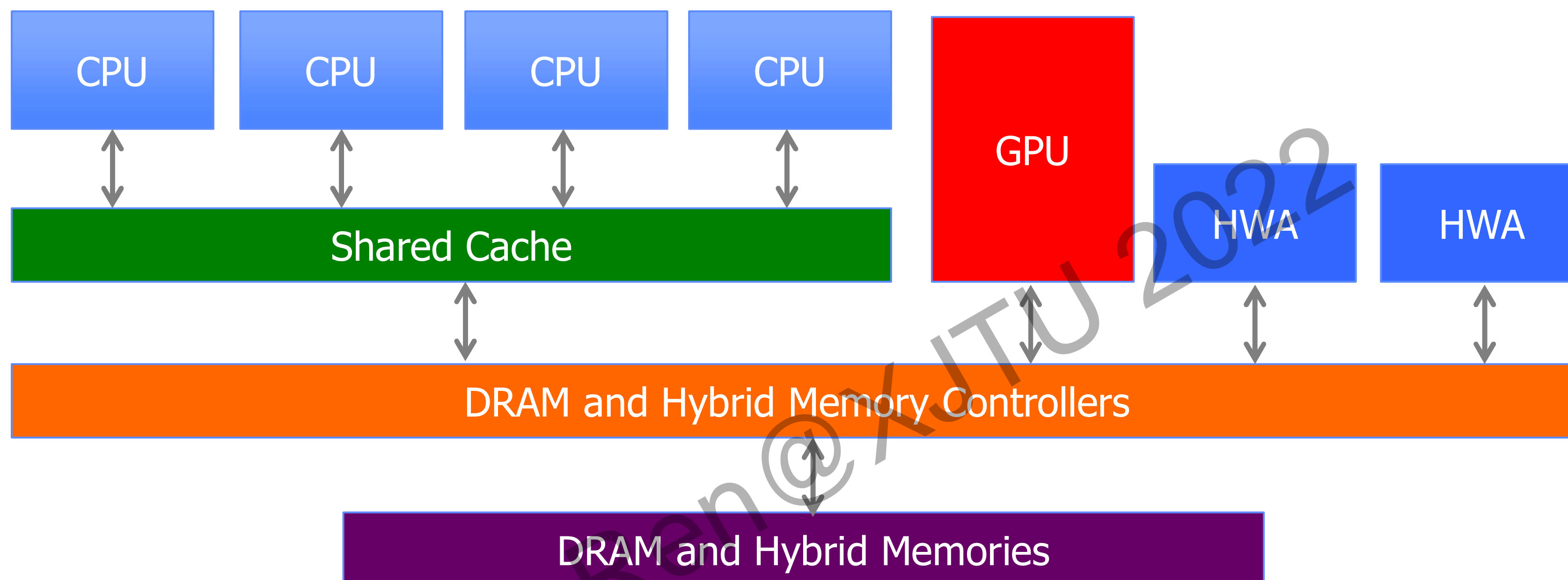
- Problem: Symmetric design is one-size-fits-all
- It tries to fit a single-size design to all workloads and metrics

- It is very difficult to come up with a single design
  - that satisfies all workloads even for a single metric
  - that satisfies all design metrics at the same time

- This holds true for different system components, or resources
  - Cores, caches, memory, controllers, interconnect, disks, servers, …
  - Algorithms, policies, …

Pengju Ren@XJTU 2022

Increasing Asymmetry in Modern Systems



Pengju Ren@XJTU 2022

- **Heterogeneous agents:**

  CPUs, GPUs, HWA (hardware Accelerators, e.g. DNN), DSP, Encryption …

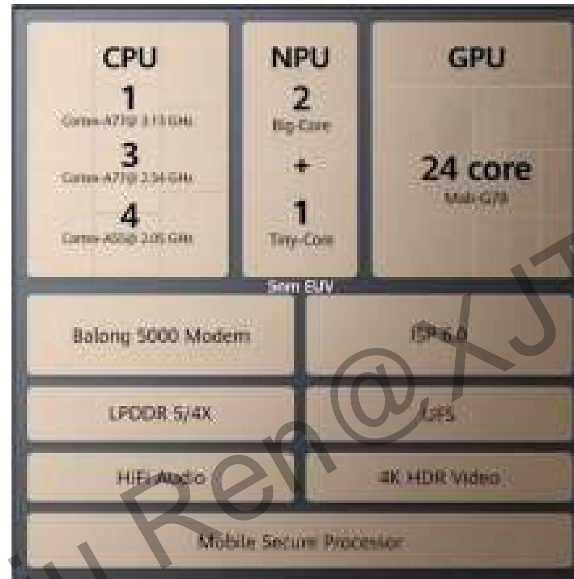
- **Heterogeneous memories:**

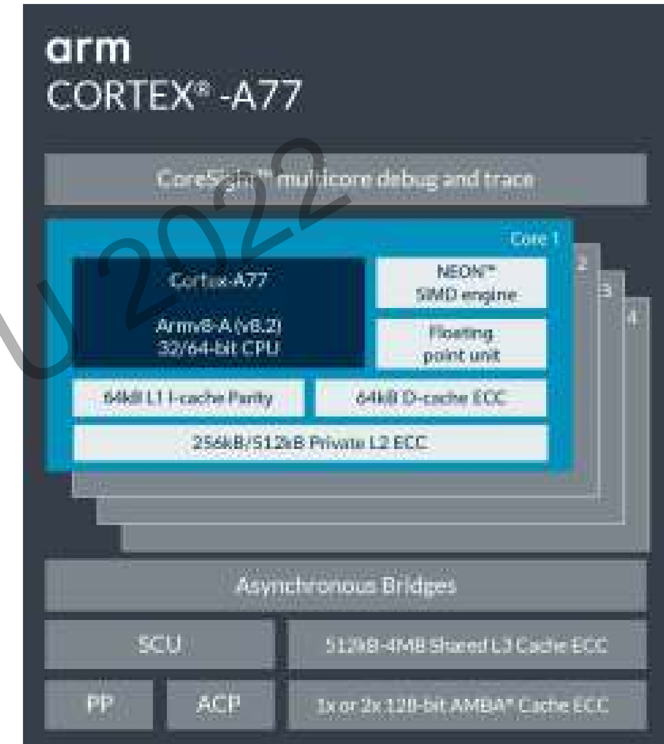
  Fast vs. Slow DRAM, e.g. HBM, DDR5

- **Heterogeneous interconnects:**

  Control, Data, Synchronization, e.g. Nvlink, NoC, BUSs, Ethernet, etc

ARM Cortex-A77





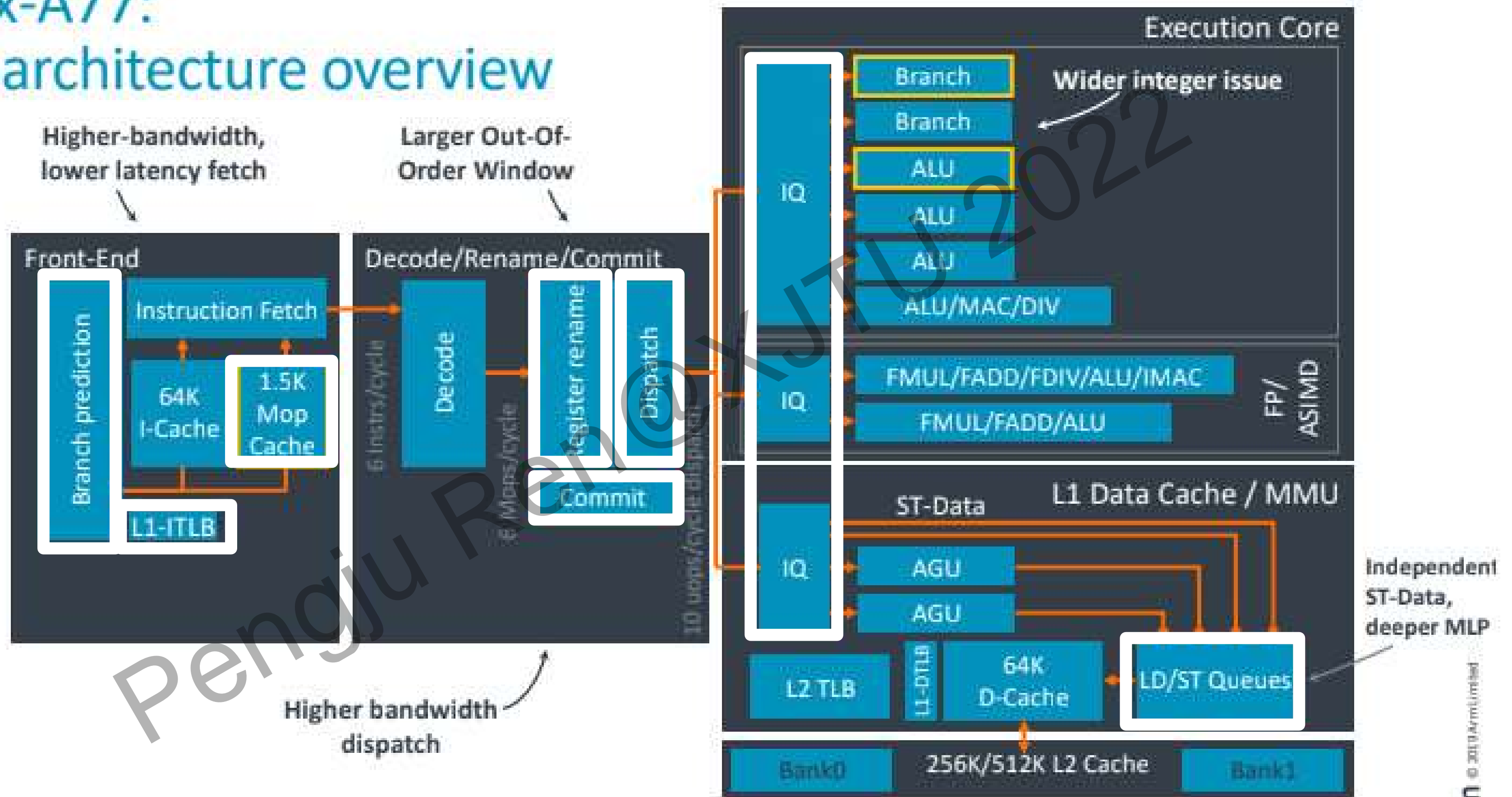
Pengju Ren@XJTU 2022

Huawei Mate40 Pro+
AP: Kirin 9000@5nm (Big.LITTLE)
4x Cortex-A77, 4x Cortex-A55
GPU: Mali-G78 (24core)
NPU: 2(Big)+1(Small)

Cortex-A77(ARM V8.2 ISA, 7nm):

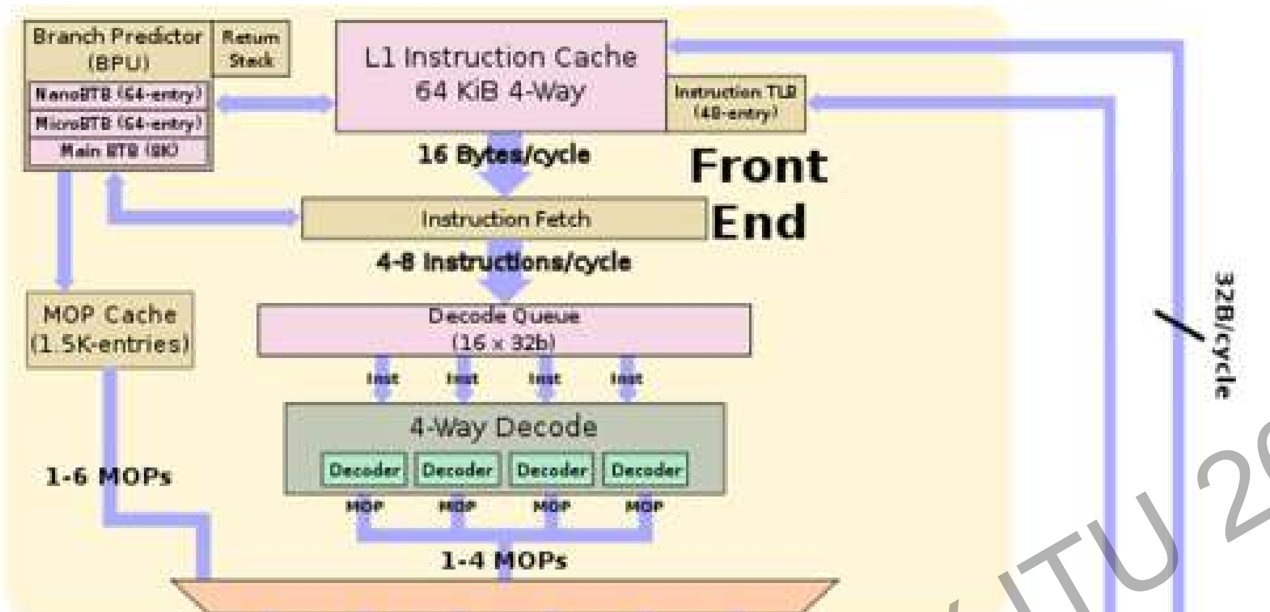
- 6-way superscalar out-of-order processor with a 10-issue back end.
- The pipeline is 13 stages with an 10-cycle branch misprediction penalty best-case.
- It has a 64 KiB L1 I-cache and a 64 KiB L1 D-cache along with a private L2 cache

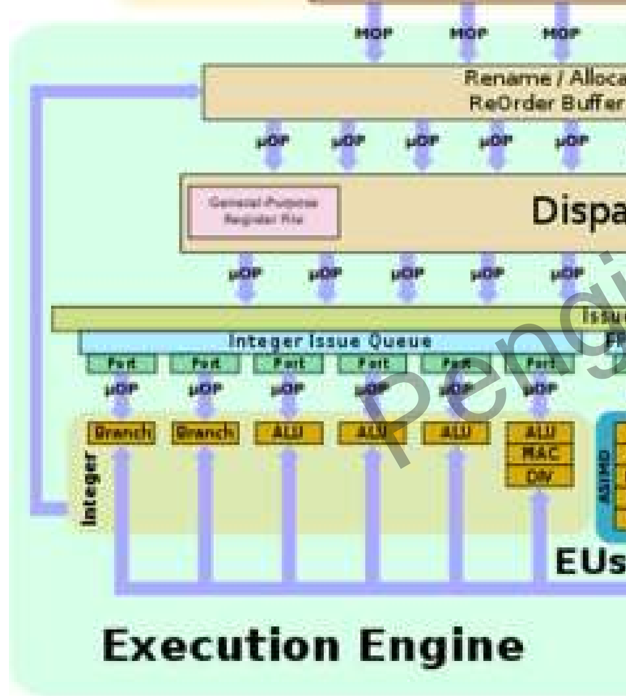
ARM Cortex-A77



Pengju Ren@XJTU 2022

Do you familiar with these components ?
Why do we need them ? What are the functions ?





Front-end

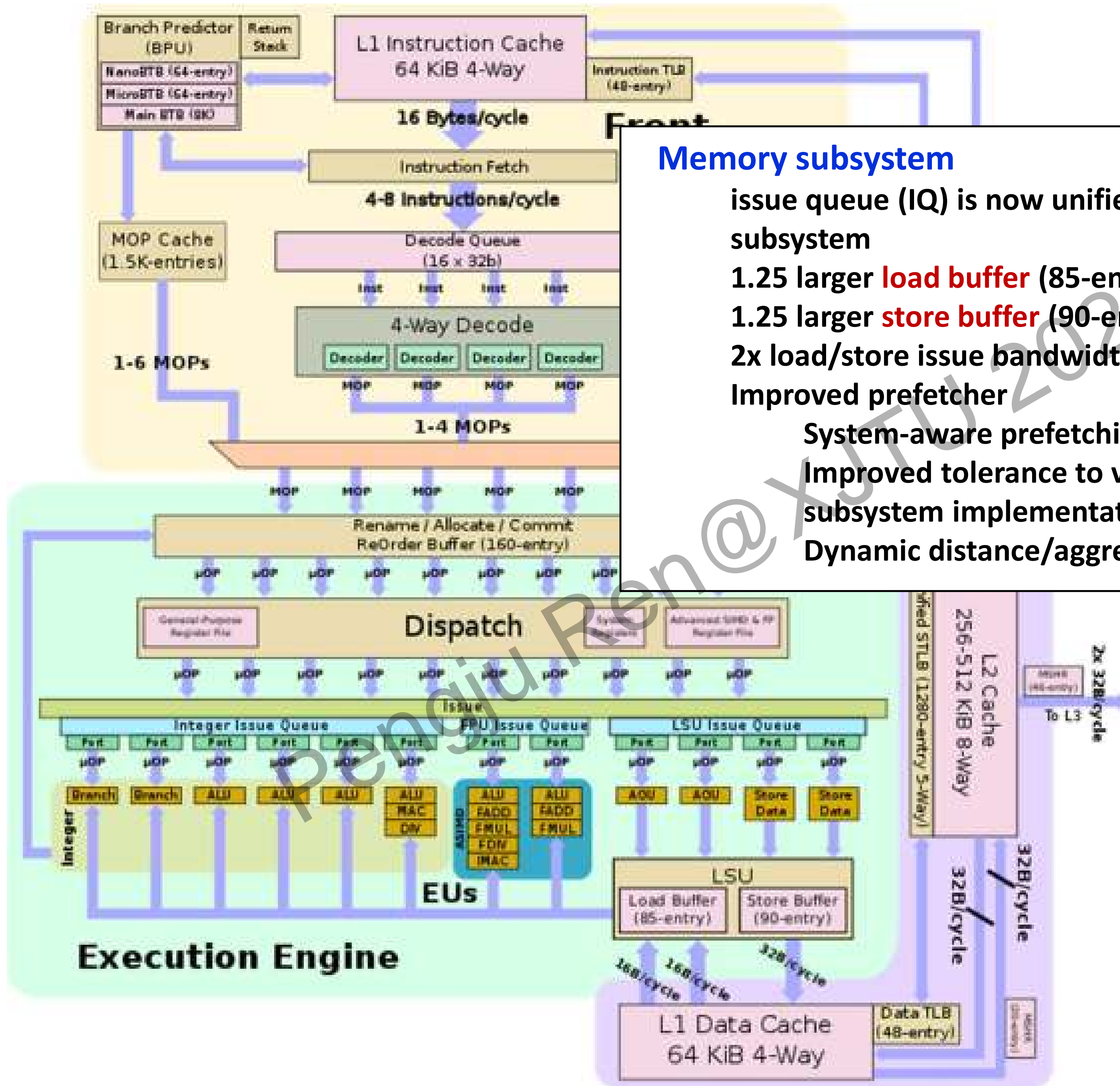
- Branch-prediction
  - Improved accuracy
  - 2x larger runahead window (64B, up from 32B)
  - 1.33x larger BTB capacity (8K-entry, up from 6K)
  - 4x larger L1 BRB capacity (64-entry, up from 16)
- Improved prefetcher
- New L0 MOP cache
- 1.5x wider instruction fetch (6 instrs/cycle, up from 4)

Pengju Ren@XJTU 2022

Execution engine

- 1.5x wider instruction fetch (6 instrs/cycle, up from 4)
- 1.0x wider decode (4-way)
- 1.5x wider rename/commit (6-way, up from 4-way)
  - lower latency recovery from branch mispredict flushes
- 1.25x larger ROB (160-entry, up from 128)
- 1.25x wider dispatch (10-way, up from 8-way)
- 1.5x wider issue (12-way, up from 8-way)
  - Execution units
    - issue queue (IQ) is now unified for integers
    - issue queue (IQ) is now unified for floating points
    - New integer ALU unit and port
    - New branch unit and port
    - New dedicated store data ports
    - New AES unit added

Pengju Ren@XJTU 2022



Memory subsystem

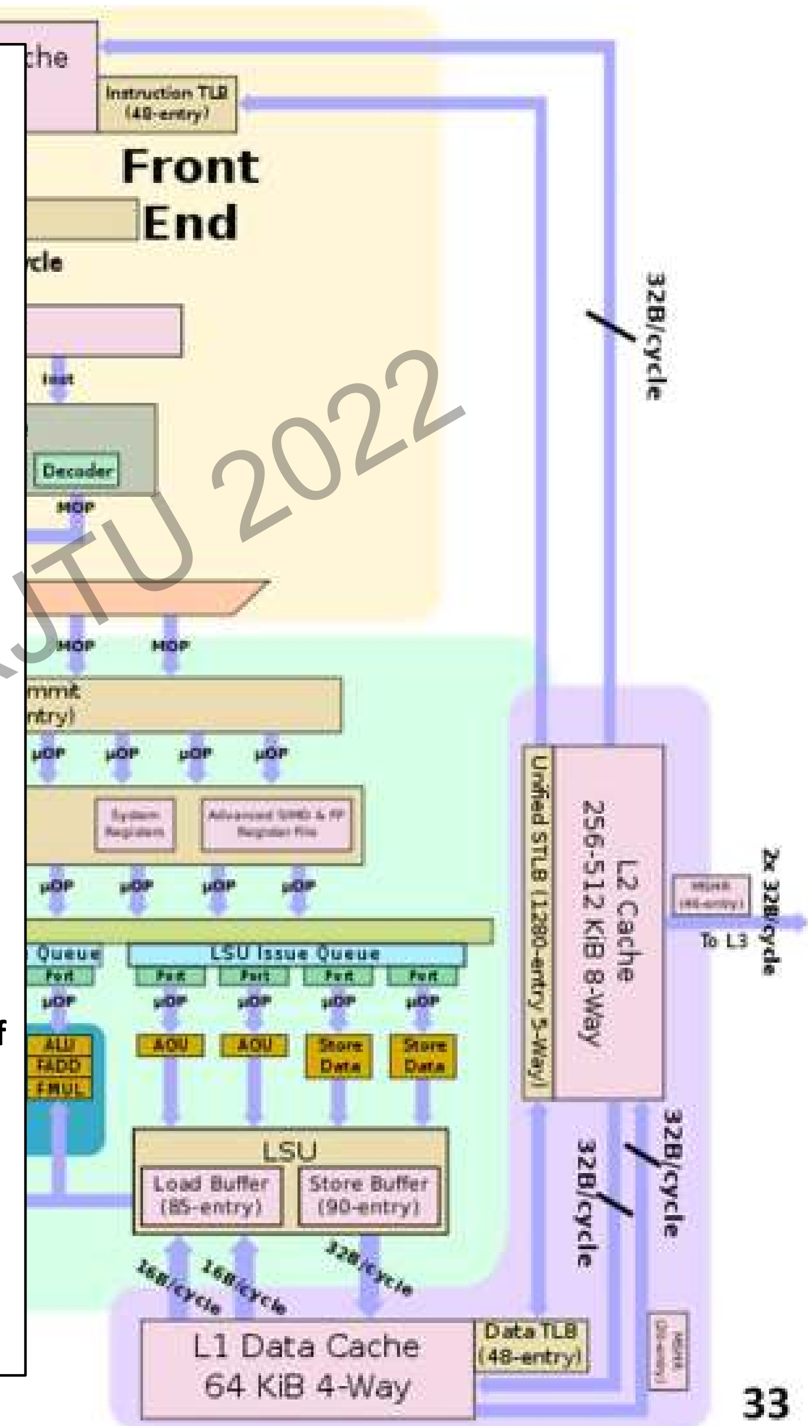
- issue queue (IQ) is now unified for the memory subsystem
- 1.25 larger load buffer (85-entry, up from 68)
- 1.25 larger store buffer (90-entry, up from 72)
- 2x load/store issue bandwidth
- Improved prefetcher
  - System-aware prefetching
  - Improved tolerance to varying memory subsystem implementations
  - Dynamic distance/aggressiveness

Pengju Ren@XJTU 2022

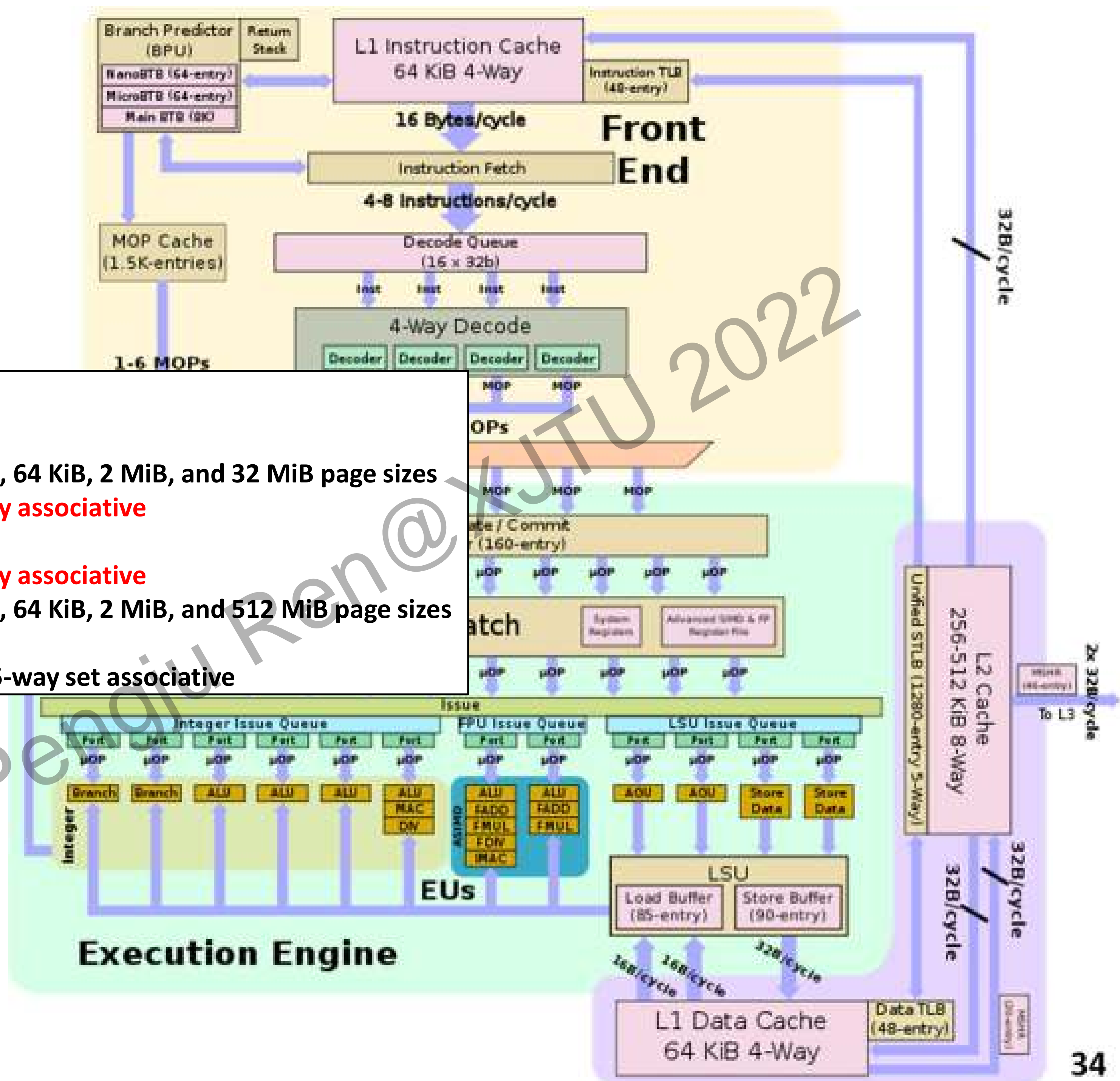
Memory Hierarchy

Cache

- **L0 MOP Cache**
  - **1536-entry**
- **L1I Cache (private)**
  - **64 KiB, 4-way set associative**
  - **64-byte cache lines**
  - **Optional parity protection**
  - **Write-back**
- **L1D Cache (private)**
  - **64 KiB, 4-way set associative**
  - **64-byte cache lines**
  - **4-cycle fastest load-to-use latency**
  - **Optional ECC protection per 32 bits**
  - **Write-back**
- **L2 Cache (private)**
  - **256 KiB OR 512 KiB (2 banks)**
  - **8-way set associative**
  - **9-cycle fastest load-to-use latency**
  - **optional ECC protection per 64 bits**
  - **Modified Exclusive Shared Invalid (MESI) coherency**
  - **Strictly inclusive of the L1 data cache & non-inclusive of the L1 instruction cache**
  - **Write-back**
- **L3 Cache (Shared)**
  - **2 MiB to 4 MiB, 16-way set associative**
  - **26-31 cycles load-to-use**
  - **Shared by all the cores in the cluster**
    - **located in the DynamIQ Shared Unit (DSU)**



Pengju Ren@XJTU 2022



TLBs (Virtual Mem)

- **ITLB**
  - **4 KiB, 16 KiB, 64 KiB, 2 MiB, and 32 MiB page sizes**
  - **48-entry fully associative**
- **DTLB**
  - **48-entry fully associative**
  - **4 KiB, 16 KiB, 64 KiB, 2 MiB, and 512 MiB page sizes**
- **STLB**
  - **1280-entry 5-way set associative**

Pengju Ren@XJTU 2022

Recall: What is a computer system ? (von Neumann Arch.)

Pengju Ren@XJTU 2022

Review: What is a computer program ?

$$value = \sum_{j=0}^{terms} coef[j]x^j$$

```
int poly(int *coef,
         int terms, int x) {
  int power = 1;
  int value = 0;
  for (int j = 0; j < terms; j++) {
    value += coef[j] * power;
    power *= x;
  }
  return value;
}
```

Pengju Ren@XJTU 2022

What is a computer program ? (from a processor's perspective)

$$value = \sum_{j=0}^{terms} coef[j]x^j$$

```
  int value = 0;
  for (int j = 0; j < terms; j++) {
    value += coef[j] * power;
    power *= x;
  }
  return value;
}
```

```
poly:
   cmp      r1, #0
   ble      .L4
   push     {r4, r5}
   mov      r3, r0
.L3:
   ldr      r5, [r3], #4
   cmp      r1, r3
   mla      r0, r4, r5, r0
   mul      r4, r2, r4
   bne      .L3
   pop      {r4, r5}
   bx       lr
.L4:
   movs     r0, #0
   bx       lr
```

A program is just a list of processor instructions!

- **Compilers for languages like C/C++ and Fortran:**
  - **Check that the program is legal**
  - **Translate into assembly code**
  - **Optimizes the generated code**

Pengju Ren@XJTU 2022

Compiler Optimizes Code

- **The compiler manages operations (load/store) that are important to performance but we don't "see" in code.**

- **Compiler performs a number of optimizations:**
  - **Unrolls loops (because control isn't free)**
  - **Fuses loo... two together)**
  - –
  - –
  - –
  - – ... 2, into
  - –

```
for (i=0; i<4; i++)
        a[i] = b[i] * c[i];
```

```
for (i=0; i<2; i++) {
        a[i*2] = b[i*2] * c[i*2];
        a[i*2+1] = b[i*2+1] * c[i*2+1];
}
```

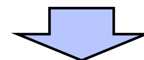
Pengju Ren@XJTU 2022

Compiler Optimizes Code

- **The compiler manages operations (load/store) that are important to performance but we don't "see" in code.**

- **Compiler performs a number of optimizations:**
  - Unrolls loops (because control isn't free)
  - Fuses loops (m... two together)
  - Interc...
  - Elimin...
  - Reord...
  - Streng... cheap...
  - Share...

```
for (i=0; i<N; i++) a[i] = b[i] * 5;
for (j=0; j<N; j++) w[j] = c[j] * d[j];
```



```
for (i=0; i<N; i++) {
        a[i] = b[i] * 5;
        w[i] = c[i] * d[i];
      }
```

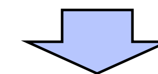
PengjuRen@XJTU 2022

Compiler Optimizes Code

- **The compiler manages operations (load/store) that are important to performance but we don't "see" in code.**

- **Compiler performs a number c**
  - Unrolls loops (because control is
  - Fuses loops (merge two toget
  - Interchanges loops (reorde
  - Eliminates dead code (the branc
  - Reorders instructions to improve
  - Strength reduction (turns expens cheaper one, shift left)
  - Share Common Subexpressions

```
for(j=0; j < N; j++) {
  for(i=0; i < M; i++) {
     x[i][j] = 2 * x[i][j];
  }
}
```



```
for(i=0; i < M; i++) {
  for(j=0; j < N; j++) {
     x[i][j] = 2 * x[i][j];
  }
}
```

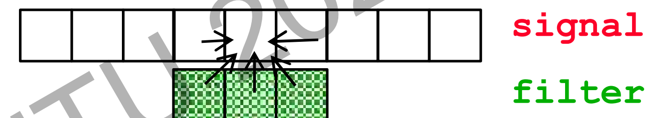
Penglu Ren@XJTU 2022

Compiler Optimizes Code

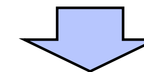
- **The compiler manages operations (load/store) that are important to performance but we don't "see" in code.**

- **Compiler performs a number o**
  - **Unrolls loops (because control is**
  - **Fuses loops (merge two togeth**
  - **Interchanges loops (reorder)**
  - **Eliminates dead code (the br**
  - **Reorders instructions to improve**
  - **Strength reduction (turns expens cheaper one, shift left)**
  - **Share Common Subexpressions**

Example is a convolution




```
for (i=0, i<N; i++) {
  y[i]=x[i-1]*filter[0]
       +x[i]*filter[1]
       +x[i-2]*filter[2]
}
```



```
f0=filter[0];
f1=filter[1];
F2=filter[2];
for (i=0, i<N; i++) {
  y[i]=x[i-1]*f0
       +x[i]*f1
       +x[i-2]*f2
}
```

Penglu Ren@XJTU 2022

Compiler Optimizes Code

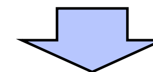
- **The compiler manages operations (load/store) that are important to performance but we don't "see" in code.**

- **Compiler performs a number**
  - **Unrolls loops (because control i**
  - **Fuses loops (merge two togeth**
  - **Interchanges loops (reorder)**
  - **Eliminates dead code (the b**
  - **Reorders instructions to improv**
  - **Strength reduction (turns expen... cheaper one, shift left)**
  - **Share Common Subexpressions**

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

```
16*x    -->    x << 4
```

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```



```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

Pengju Ren@XJTU 2022

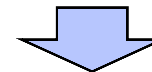
Compiler Optimizes Code

- **The compiler manages operations (load/store) that are important to performance but we don't "see" in code.**

- **Compiler performs a number**
  - Unrolls loops (because control i
  - Fuses loops (merge two togeth
  - Interchanges loops (reorder)
  - Eliminates dead code (the b
  - Reorders instructions to improv
  - Strength reduction (turns expen cheaper one, shift left)
  - Share Common Subexpressions

▪ Reuse portions of expressions

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```



```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

Penglu Ren@XJTU 2022

Compiler Optimizes Code

- The compiler manages operations (load/store) that are important to performance but we don't "see" in code.
- Compiler performs a number of optimizations:
  - Unrolls loops (because control isn't free)
  - Fuses loops (merge two together)
  - Interchanges loops (reorder)
  - Eliminates dead code (the branch never taken)
  - Reorders instructions to improve register reuse and more
  - Strength reduction (turns expensive instruction, e.g., multiply by 2, into cheaper one, shift left)
  - Share Common Subexpressions
- Why is this your problem?
  - Most analysis is based only on static information
  - When in doubt, the compiler must be conservative

Pengju Ren@XJTU 2022

Example : Why Couldn't Compiler Optimize (1)

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
         b[i] = 0;
         for (j = 0; j < n; j++)
             b[i] += a[i*n + j];
    }
}
```

Code updates b[i] (need write to memory) on every iteration.

Why couldn't compiler optimize this away?

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

```
double A[9] =
  { 0,   1,   2,
    3,  22, 224},
   32,  64, 128};
```

```
init:   [4, 8, 16]
i = 0:  [3, 8, 16]
i = 1:  [3, 22, 16]
i = 2:  [3, 22, 224]
```

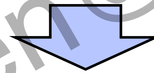
Value of B

Memory Aliasing (Two different memory references specify single location) : Must consider possibility that these updates will affect program behavior

Pengju Ren@XJTU 2022

Example : Why Couldn't Compiler Optimize (2)

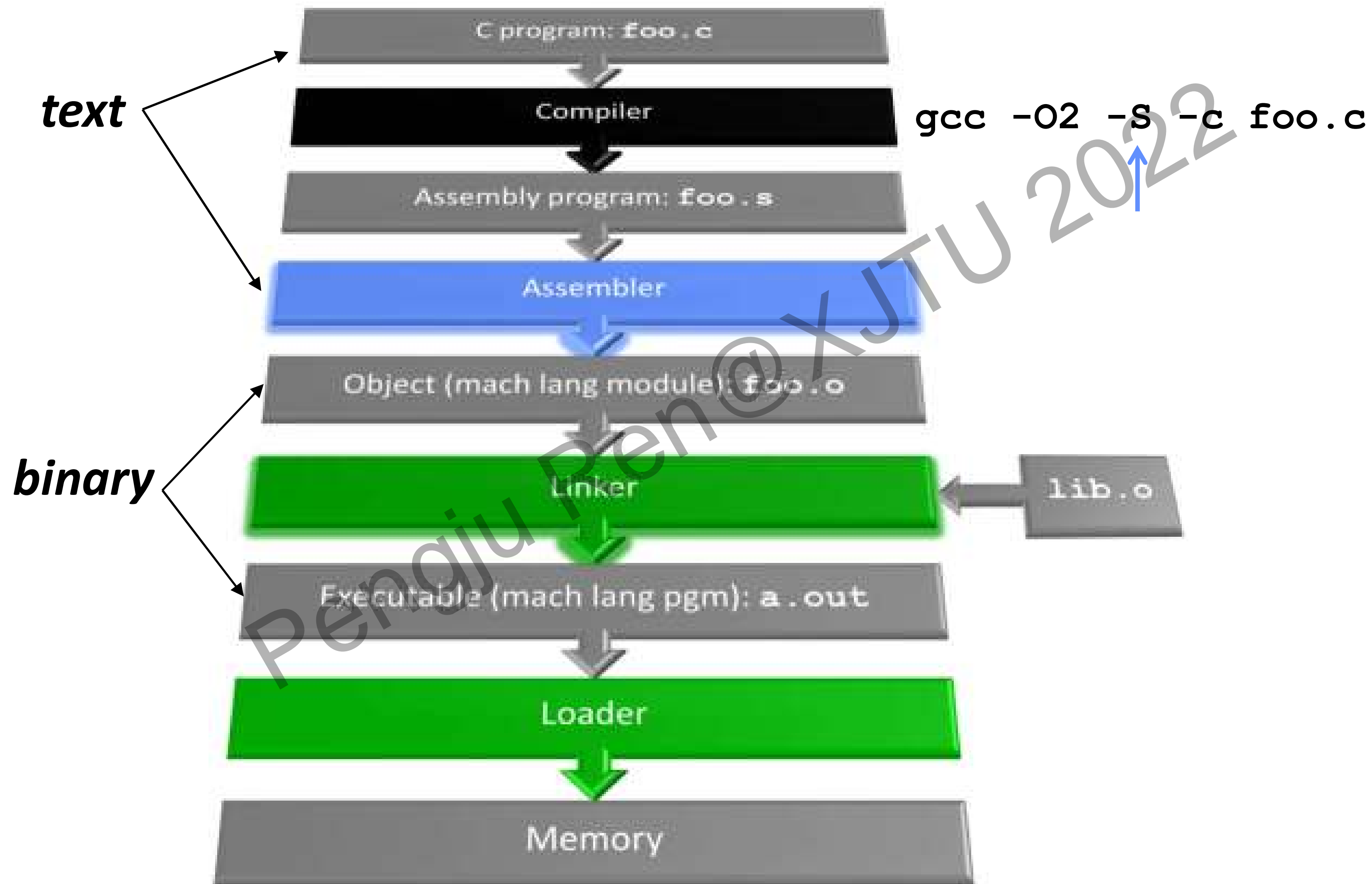
```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```



```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

Pengju Ren@XJTU 2022

Steps in Compiling and Running a C Program



gcc -O2 -S -c foo.c

Pengju Ren@XJTU 2022

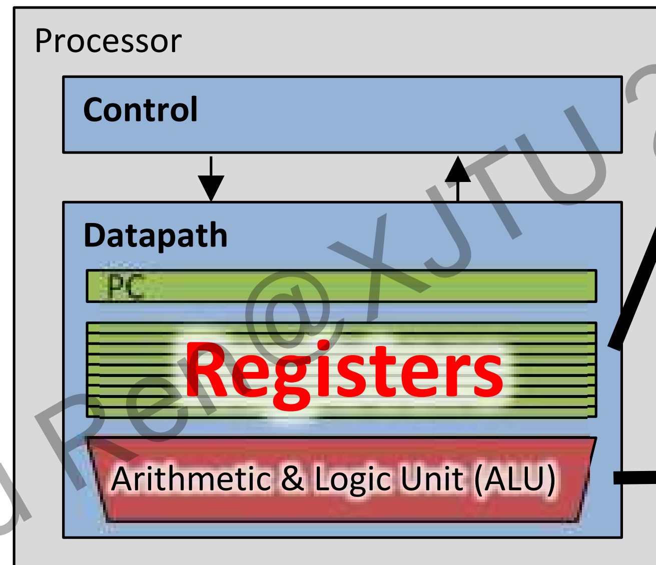
Review: What is a Processor?

Pengju Ren@XJTU 2022

Processor for Desktop/Laptop

HiSilicon Processor on Mobile Phone

What does a Processor do?



Registers: maintain program state: store value of variables used as inputs and outputs to operations

Execution unit: performs the operation required by an instruction, which may modify values in registers or memory

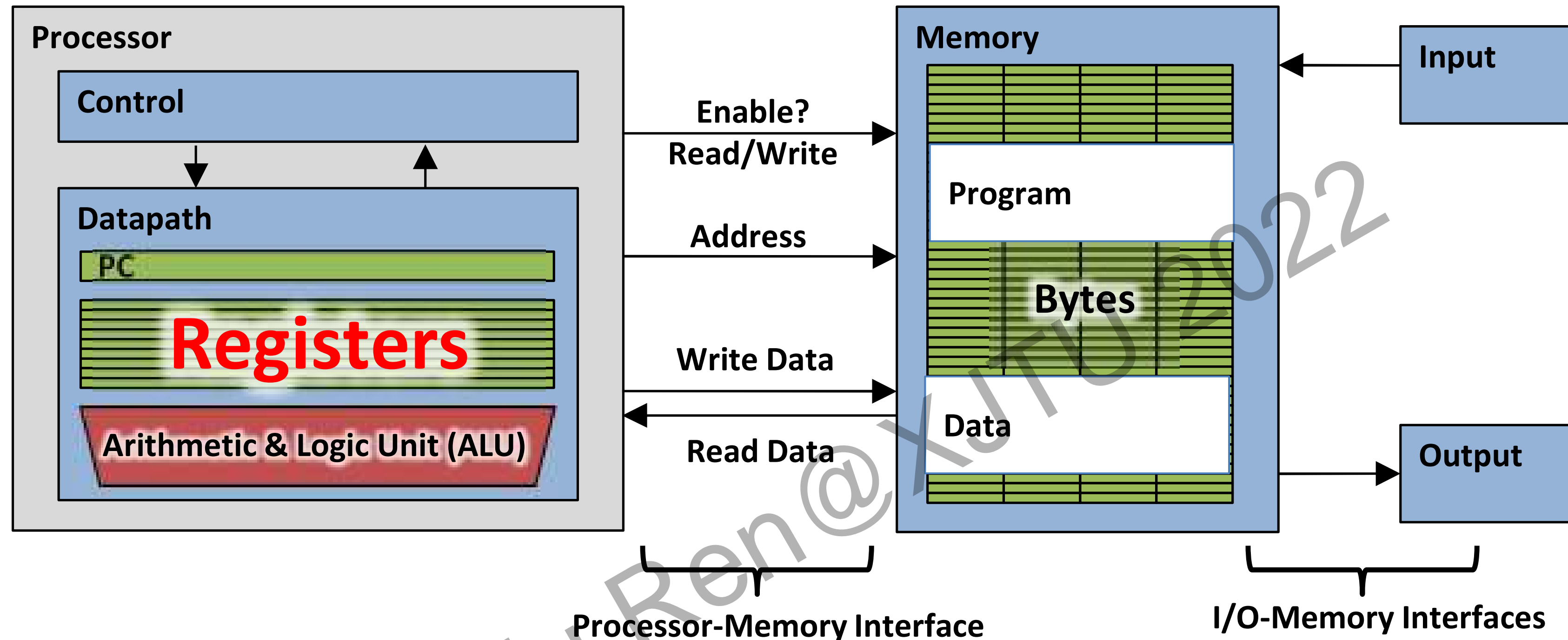
A processor executes instructions

Executing an instruction modifies the computer's "state"

Where do instructions/data come and go ?

Pengju Ren@XJTU 2022

What does a Processor do?



Pengju Ren@XJTU 2022

Programmer-Visible State

- PC: Program counter
  - » Address of next instruction
  - » Called "RIP" (x86-64)
- Register file
  - » Heavily used program data
- Condition codes
  - » Store status information about most recent arithmetic or logical operation
  - » Used for conditional branching (x86, ARM)

Memory

- Byte addressable array
- Code and user data
- Stack to support procedures
- Heap for dynamic usage

I/O (Memory mapped)

- I/O Devices respond to their assigned address ranges

What does a Processor do?

Processor

Memory

- In C (and most High Level Languages) variables declared first and given a type. E.g.,

```
int fahr, celsius;
char a, b, c, d, e;
```

- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match `int` and `char` variables).
- In Assembly Language, the registers have no type
  - Operation determines how register contents are treated
  - Transfer data between memory and register
    - Load data from memory into register
    - Store register data into memory
  - Perform arithmetic function on register or memory data
  - Transfer control
    - Unconditional jumps to/from procedures
    - Conditional branches
    - Indirect branches

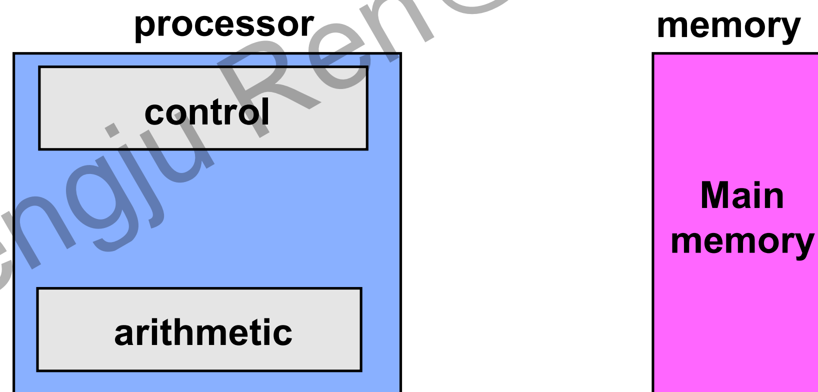
recent arithmetic or logical operation

» Used for conditional branching

Pengju Ren@XJTU 2022

Idealized Uniprocessor Model

- Processor names **variables**:
  - Integers, floats, doubles, pointers, arrays, structures, etc.
- Processor performs **operations** on those variables:
  - Arithmetic, logical operations, etc.
- Processor **controls** the order, as specified by program
  - Branches (if), loops, function calls, etc.

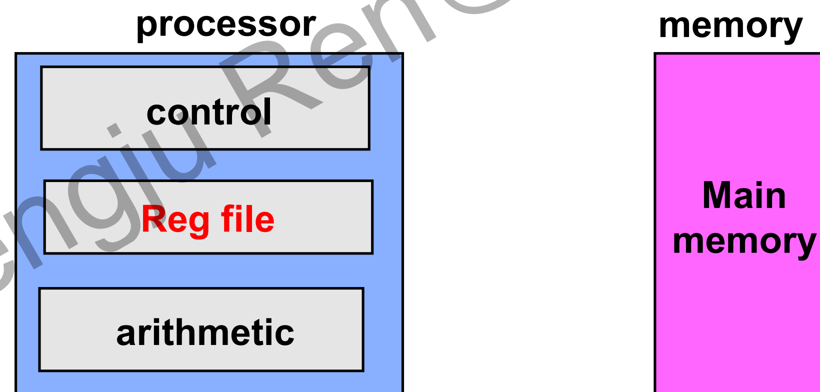


- ***Idealized Cost***
  - Each operation (+,*, &, etc.) has roughly the same cost (on “free” registers)

Pengju Ren@ XJTU 2022

Realistic Uniprocessor Model

- Processor names variables:
  - Integers, floats, doubles, pointers, arrays, structures, etc.
  - These are really words, e.g., 64-bit doubles, 32-bit ints, bytes, etc.
- Processor performs operations on those variables:
  - Arithmetic, logical operations, etc.
  - Only performs these operations on values in registers (need LD/ST Reg↔Mem)
- Processor controls the order, as specified by program
  - Branches (if), loops, function calls, etc.

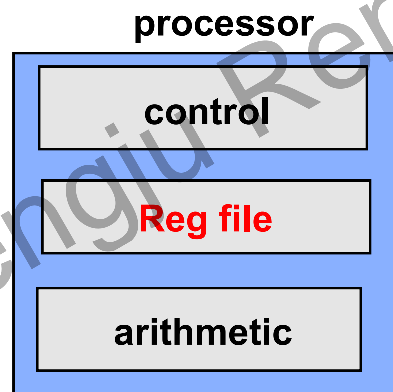


Pengju Ren@XJTU 2022

- *Idealized Cost*
  - Each operation (+,*, &, etc.) has roughly the same cost (on “free” registers)
  - Load/store is 100x the cost of +,*,&, etc.

Realistic Uniprocessor Model

- Processor names **variables**:
  - Integers, floats, doubles, pointers, ar
  - These are really words, e.g., 64-bit do
- Processor performs **operations** on th
  - Arithmetic, logical operations, etc.
  - Only performs these operations o
- Processor **controls** the order, as
  - Branches (if), loops, function calls, et



- *Idealized Cost*
  - Each operation (+,*, &, etc.) has roug
  - Load/store is 100x the cost of +,*,&, etc.

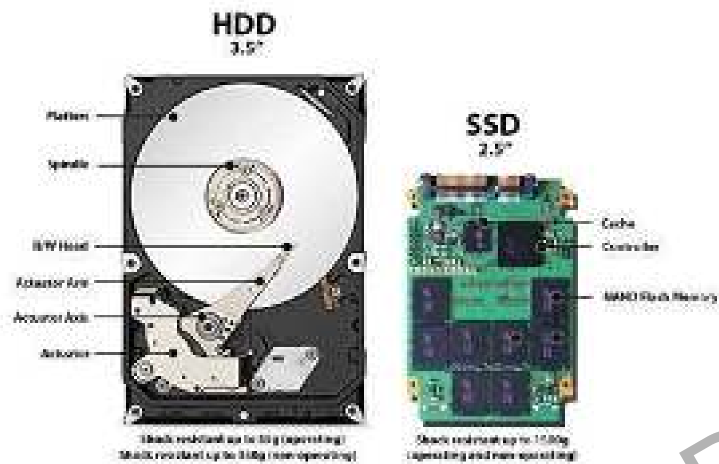
Hardware executes instructions in order specified by compiler

Six Fundamental Steps in Calling a Function

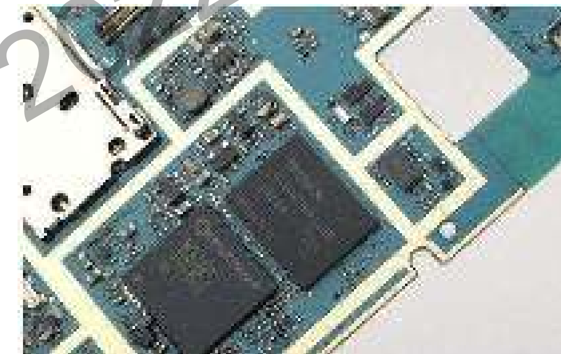
① Put parameters in a place where function can access them
② Transfer control to function
③ Acquire (local) storage resources needed for function
④ Perform desired task of the function
⑤ Put result value in a place where calling code can access it and restore any registers you used; release local storage
⑥ Return control to point of origin, since a function can be called from several points in a program

Pengju Ren@XJTU 2022

Review: What is a Memory（Storage）?



Hard Disk (nonvolatile memory)



Memory on PC and Mobile Phone

Pengju Ren 2022

What is a Memory（Storage）?

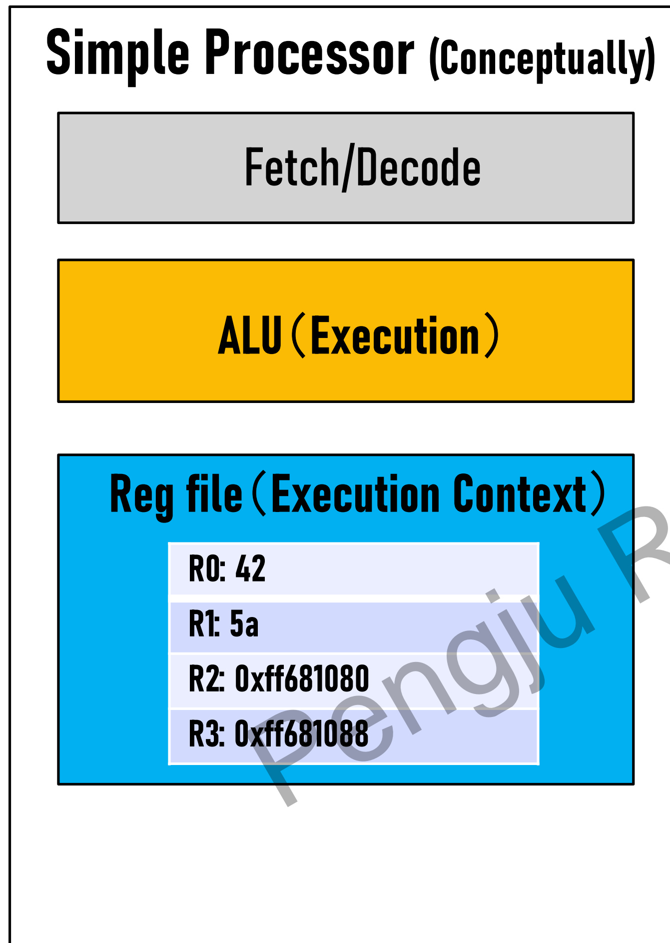
Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
⋮	⋮
0x1F	0

A computer's memory is organized as a array of bytes, each byte is identified by its "address" in memory (its position in this array)

- Instruction Memory
- Data Memory

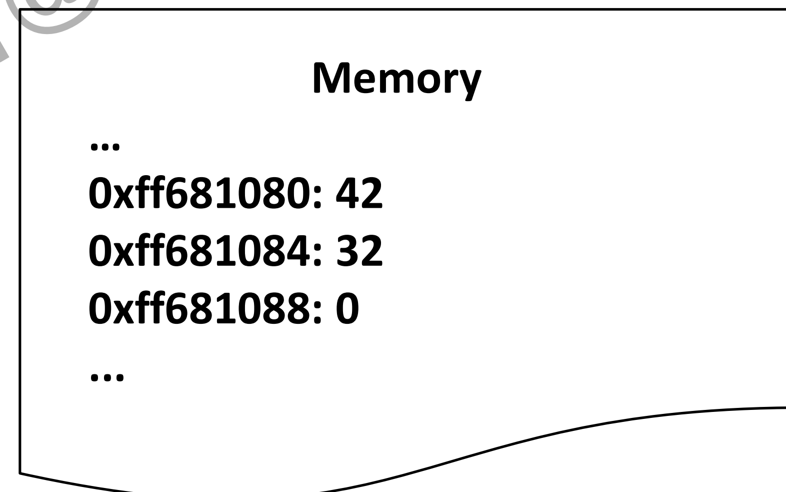
Pengju Ren@XJTU 2022

Load (inst): an instruction/data for accessing the contents of memory



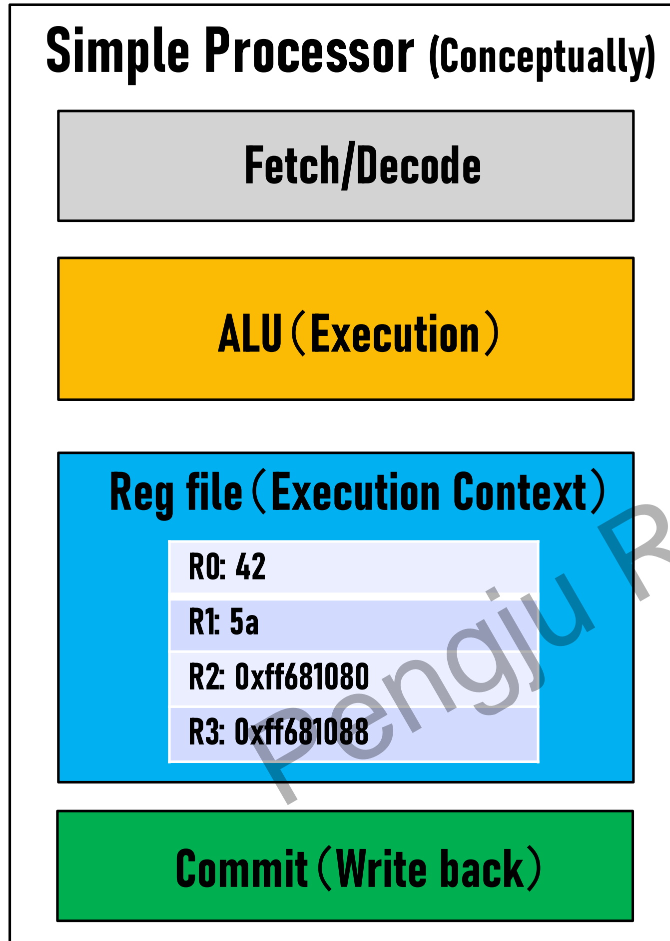
Ld R0 <- mem [R2]

"load the four-byte value in memory starting from the address stored by register R2 and put this value into register R0"



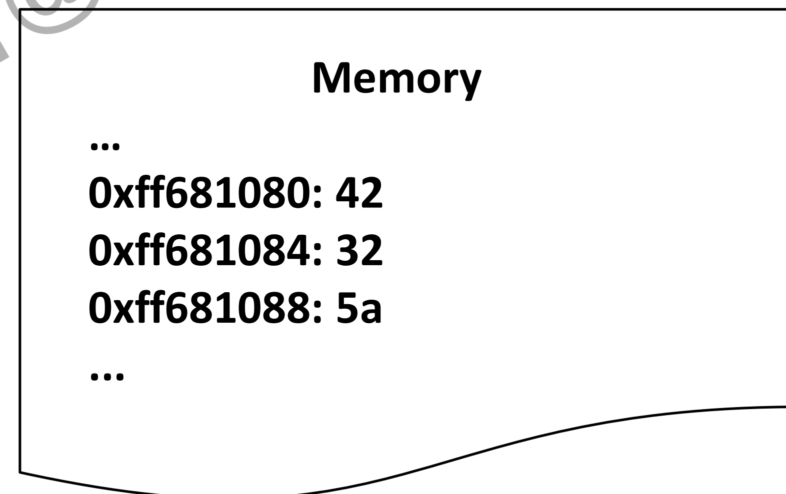
Pengju Ren@XJTU 2022

Store(inst): write a data to memory



ST R1 -> mem [R3]

"Store the four-byte value from register R1 and put this value into memory starting from the address stored by register R3"



Pengju Ren@XJTU 2022

Review of how computers work (von Neumann) …

- **What is a computer program? (from a processor's perspective)**

  It is a list of instructions to execute!

- **What is an instruction?**

  It describes an operation for a processor to perform. Executing an instruction typically modifies the computer's state.

- **What is memory ?**

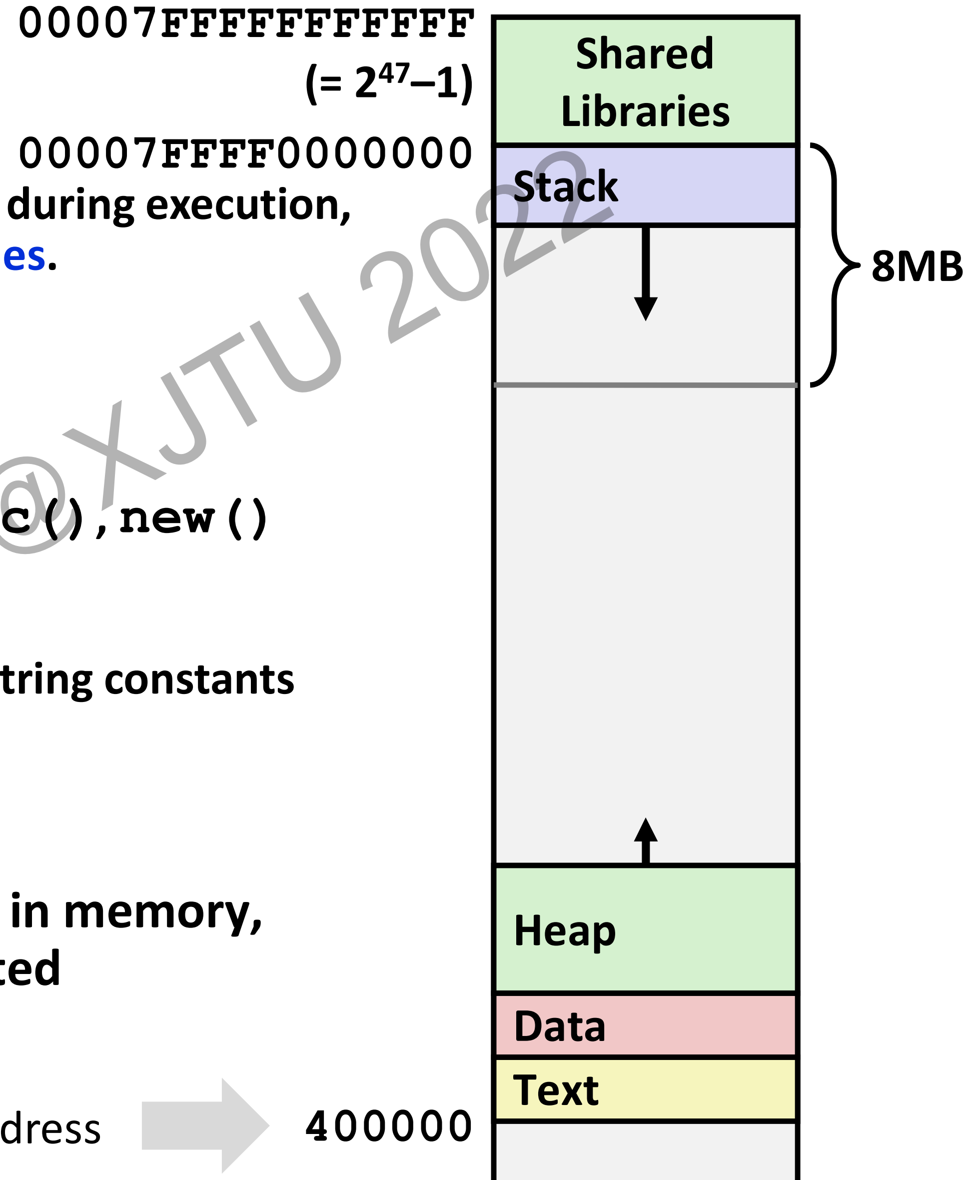
  A computer's memory is organized as a array of bytes, each byte is identified by its "address" in memory (its position in this array). It can be categorized as instruction and data memory accordingly.

How can Computer like this handle real Applications(Algorithms)?

Pengju Ren@XJTU 2022

C Memory Management

- C has 4 pools of memory
  - Stack: Space to be used by procedure during execution, this is where we can save register values.
    - Runtime stack (8MB limit)
    - E. g., local variables
  - Heap
    - Dynamically allocated as needed
    - When call `malloc()`, `calloc()`, `new()`
  - Data
    - Statically allocated data
    - E.g., global vars, `static` vars, string constants
  - Text / Shared Libraries
    - Executable machine instructions
    - Read-only
- C requires knowing where objects are in memory, otherwise things don't work as expected



Pengju Ren@XJTU 2022

Memory Allocation Example

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *phuge1, *psmall2, *phuge3, *psmall4;
    int local = 0;
    phuge1 = malloc(1L << 28);  /* 256 MB */
    psmall2 = malloc(1L << 8);  /* 256  B */
    phuge3 = malloc(1L << 32);  /*   4 GB */
    psmall4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```

Where does everything go?

Pengju Ren@XJTU 2022

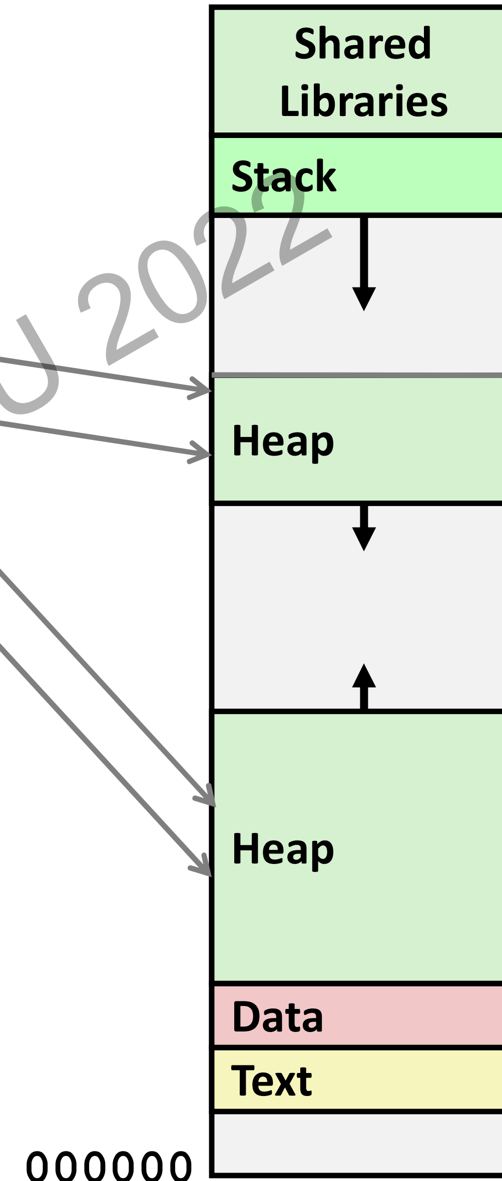
x86-64 Example Addresses

address range ~2^{47}

not drawn to scale

`local`	`0x00007ffe4d3be87c`
`phuge1`	`0x00007f7262a1e010`
`phuge3`	`0x00007f7162a1d010`
`psmall4`	`0x000000008359d120`
`psmall2`	`0x000000008359d010`
`big_array`	`0x0000000080601060`
`huge_array`	`0x0000000000601060`
`main()`	`0x000000000040060c`
`useless()`	`0x0000000000400590`

(Exact values can vary)



Pengju Ren@XJTU 2022

Mechanisms in Procedures/Programs

- **Passing control**
  - **To beginning of procedure code**
  - **Back to return point**
- **Passing data**
  - **Procedure arguments**
  - **Return value**
- **Memory management**
  - **Allocate during procedure execution**
  - **Deallocate upon return**
- **Mechanisms all implemented with machine instructions**
- **Most processor implementation of a procedure uses only those mechanisms required**

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

Pengju Ren@XJTU 2022

Mechanisms in Procedures/Programs

- **Passing control**
  - **To beginning of procedure code** ***(jump to label)***
  - **Back to return point** ***(jump to addr poped from stack)***
- **Passing data**
  - **Procedure arguments**
  - **Return value**
- **Memory management**
  - **Allocate during procedure execution**
  - **Deallocate upon return**
- **Mechanisms all implemented with machine instructions**
- **Most processor implementation of a procedure uses only those mechanisms required**

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

Pengju Ren@XJTU 2022

Mechanisms in Procedures/Programs

- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments *(specific regs)*
  - Return value *(specific regs)*
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **Most processor implementation of a procedure uses only those mechanisms required**

Only allocate stack space when needed

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

Pengju Ren@XJTU 2022

Mechanisms in Procedures/Programs

- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments (Caller to Callee)
  - Return value (Callee to Caller)
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **Most processor implementation of a procedure uses only those mechanisms required**

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

Pengju Ren@XJTU 2022

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: `call label`
  - Push return address on stack
  - Jump to *label*
- Return address:
  - Address of the next instruction right after call
  - Example from disassembly
- Procedure return: `ret`
  - Pop address from stack
  - Jump to address
- Machine instructions implement the mechanisms, but the choices are determined by designers. These choices make up the Application Binary Interface (ABI).

Pengju Ren@XJTU 2022

Symbolic Register Names (RISC-V & x86)

Reg# hardware understands

Human-friendly symbolic names in assembly code

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Table 18.2: RISC-V calling convention register usage.

RISC-V Register Usage

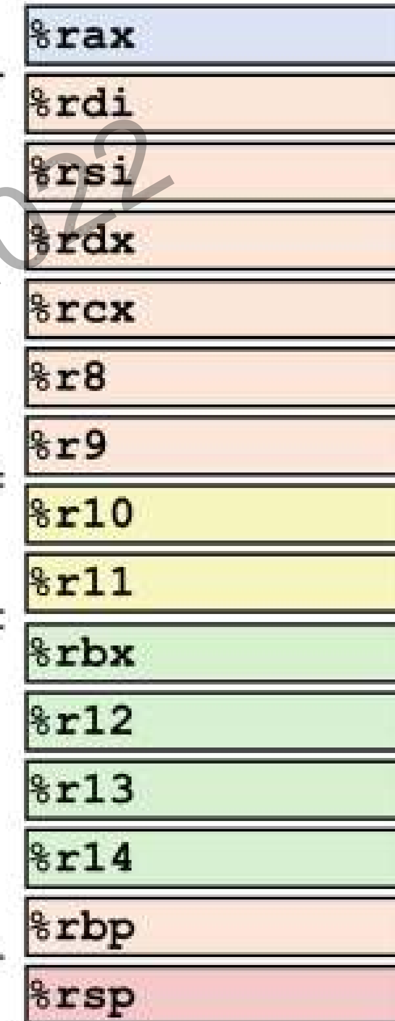
Caller-saved Return value

Caller-saved Arguments

Caller-saved Temporaries

Callee-saved Temporaries

Special



x86-64 Linux Register Usage

Pengju Ren@XJTU 2022

Call Chain Example

```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

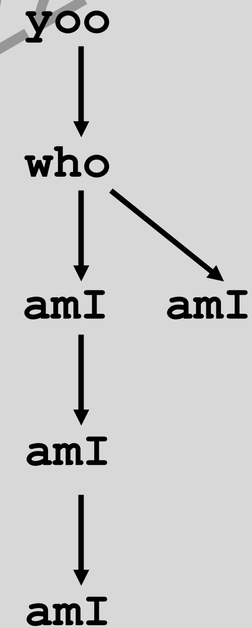
```
who(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```

```
amI(…)
{
  •
  •
  amI();
  •
  •
}
```

Procedure `amI()` is recursive

Example

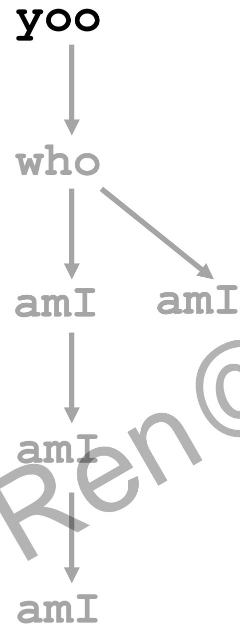
Call Chain

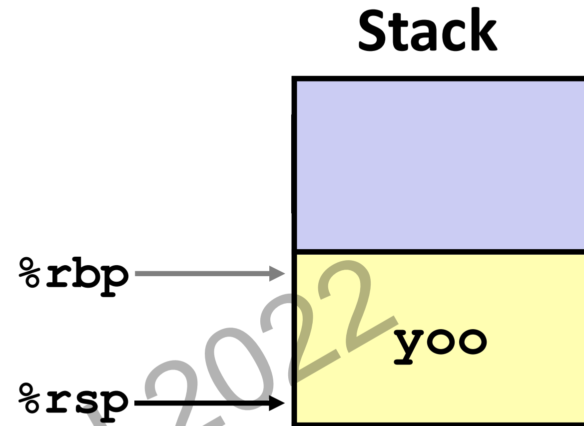


Pengju Ren@XJTU 2022

Example

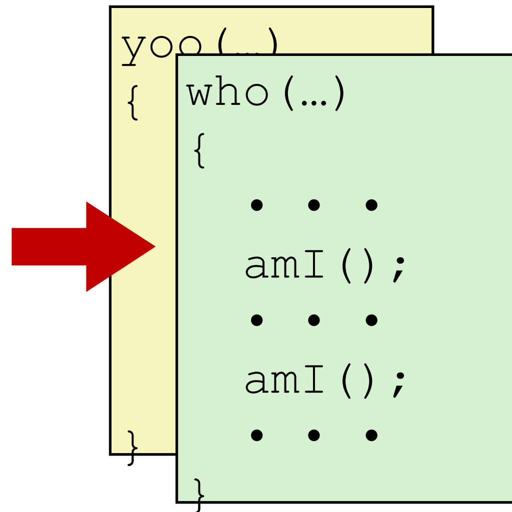
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

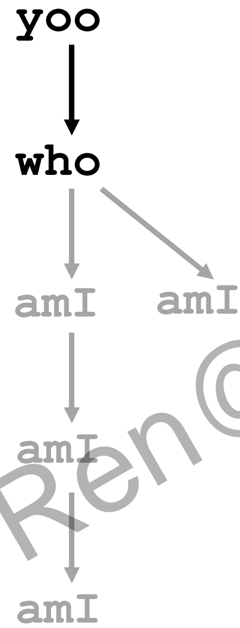


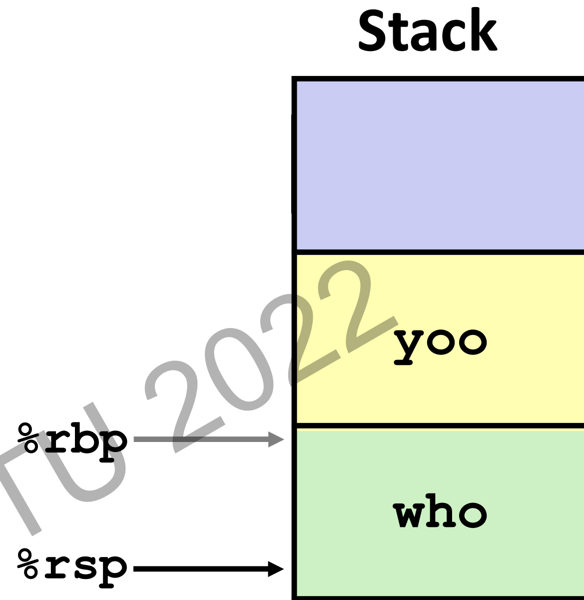


Pengju Ren@XJTU 2022

Example

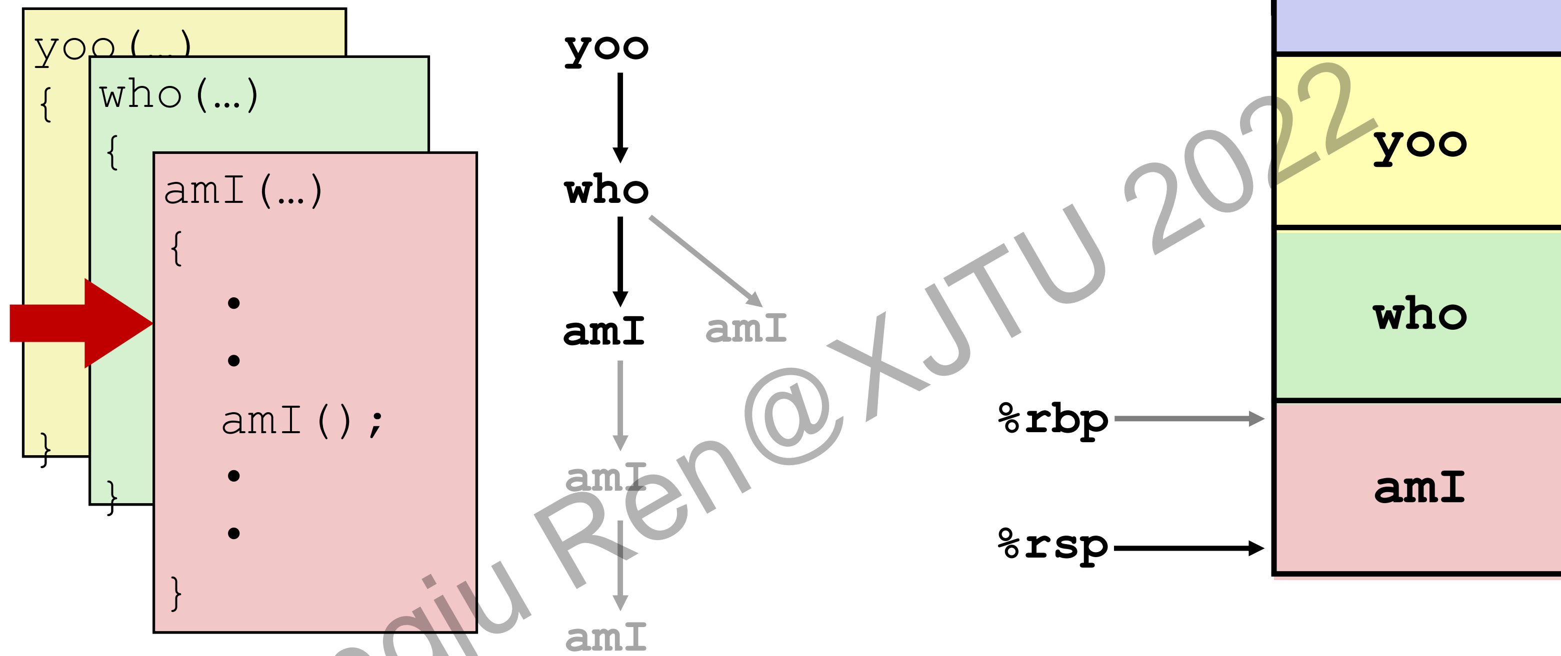






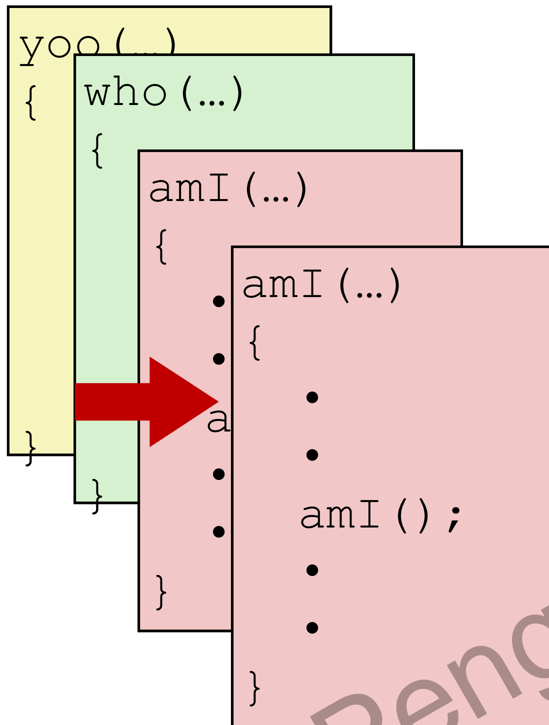
Pengju Ren@XJTU 2022

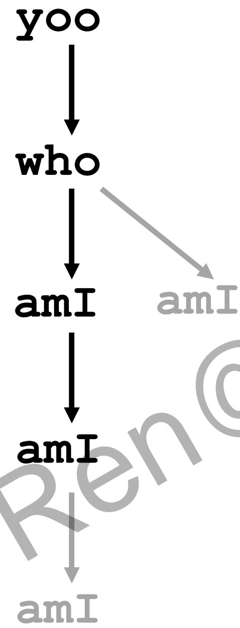
Example

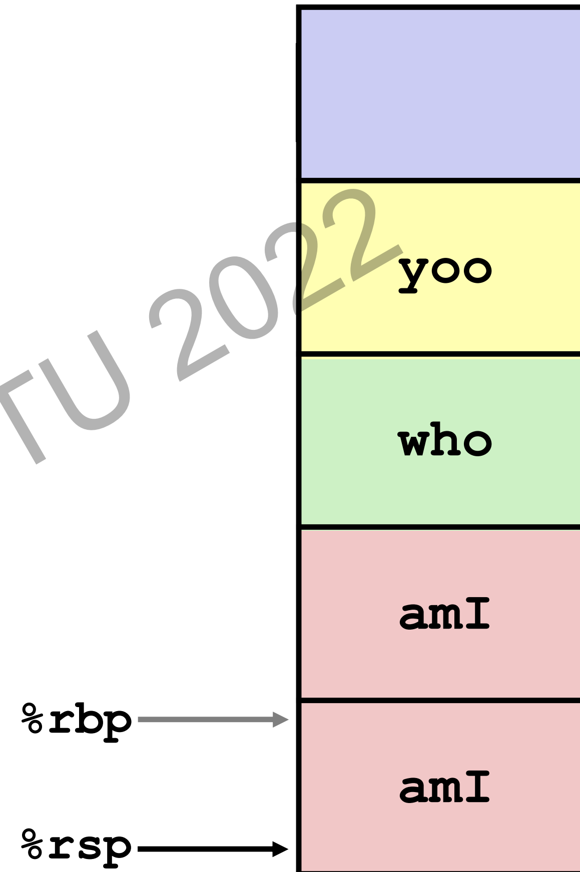


Pengju Ren@XJTU 2022

Example

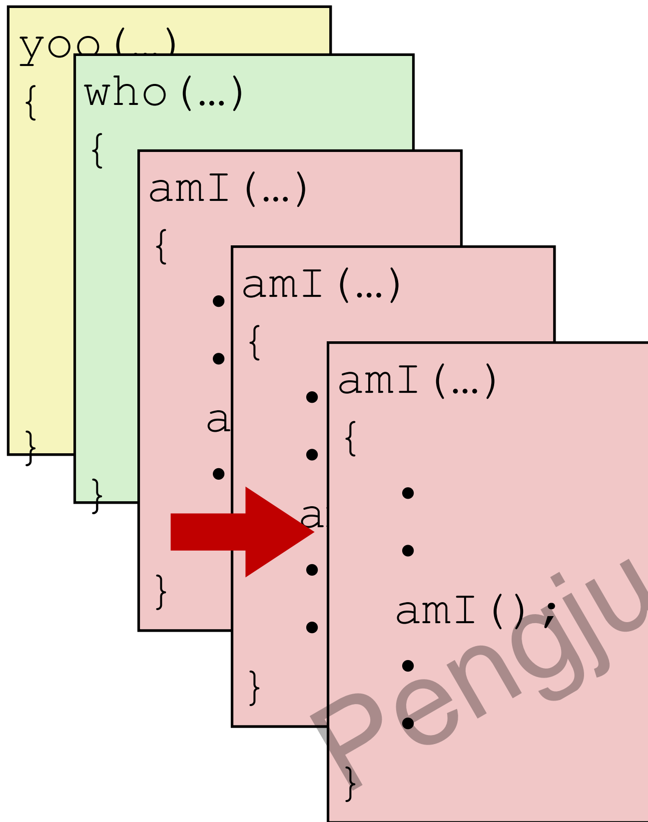


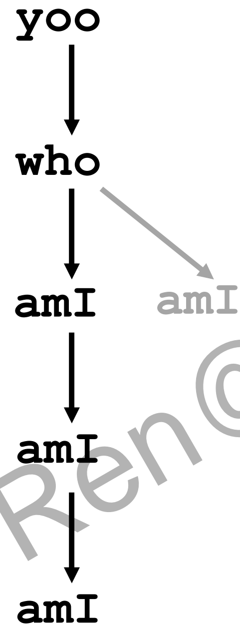


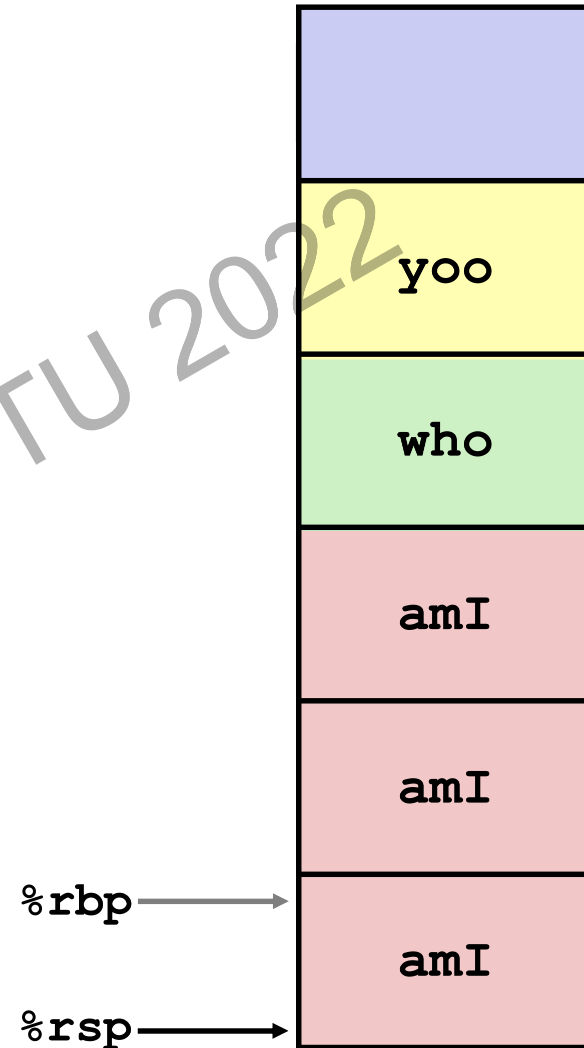


Pengju Ren@XJTU 2022

Example

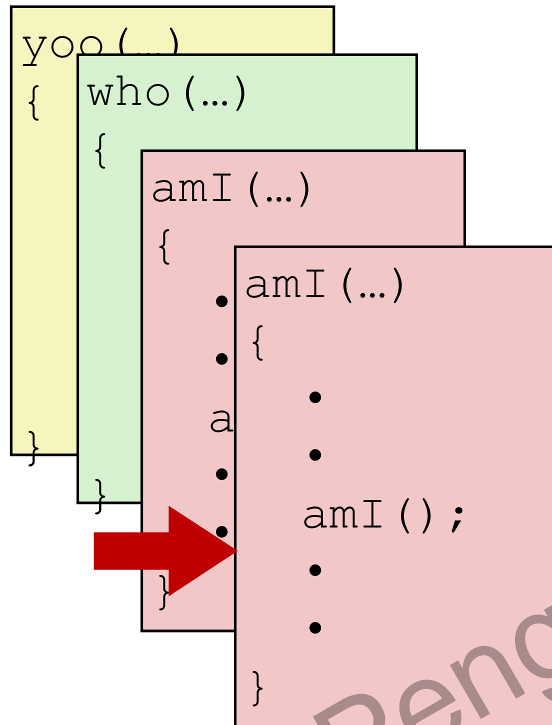


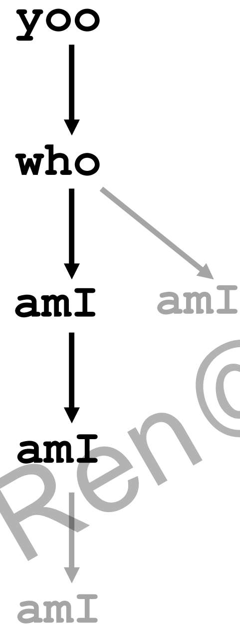


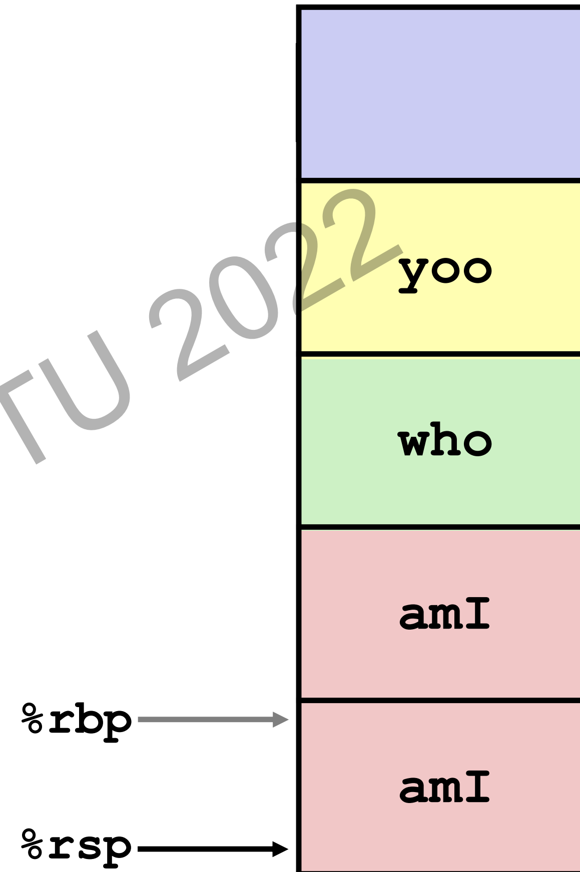


Pengju Ren@XJTU 2022

Example

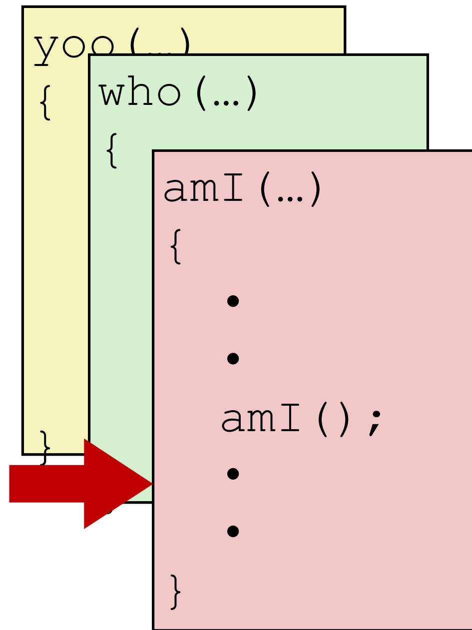


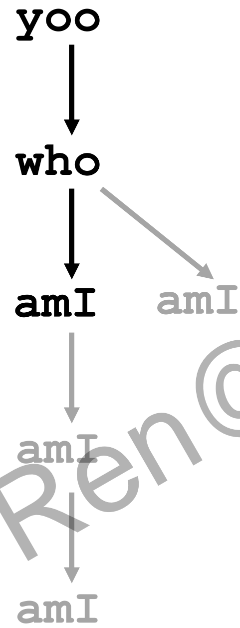


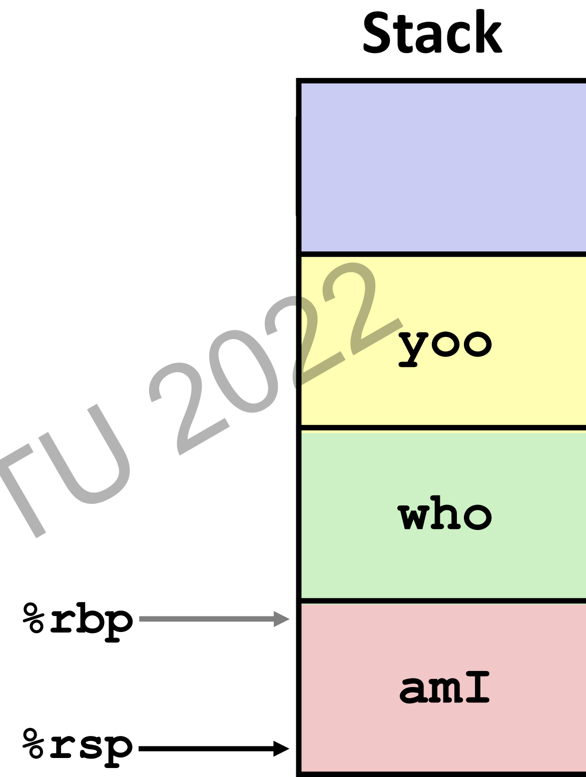


Pengju Ren@XJTU 2022

Example

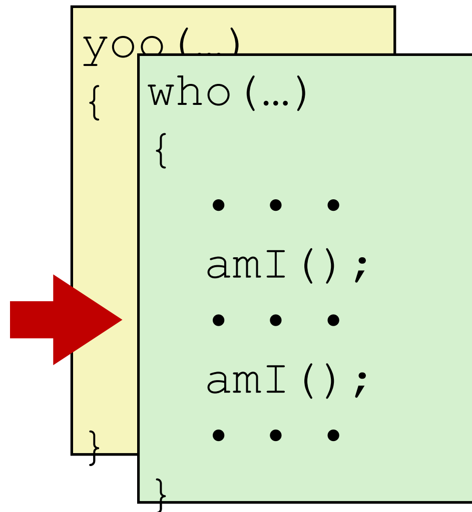


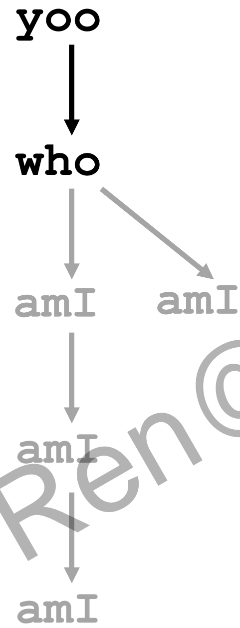


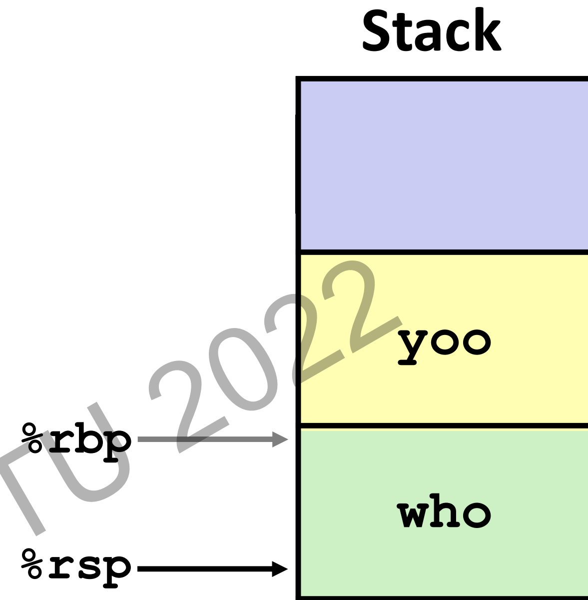


Pengju Ren@XJTU 2022

Example

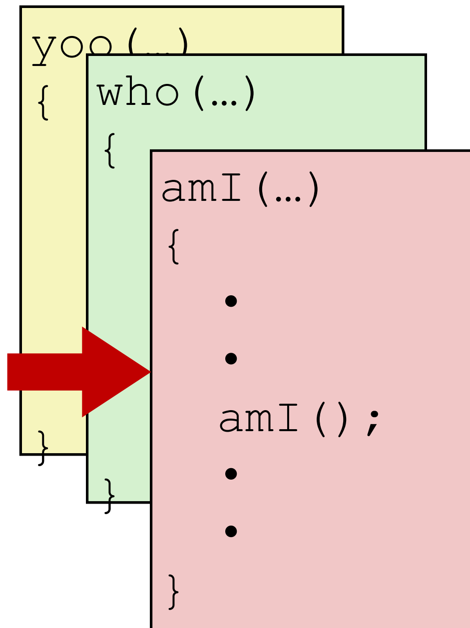


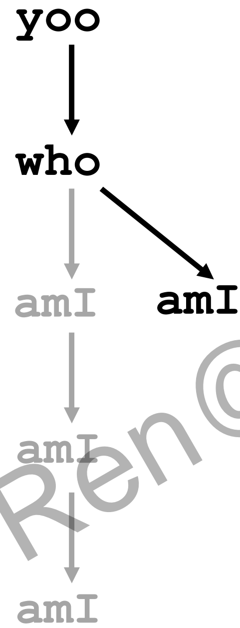


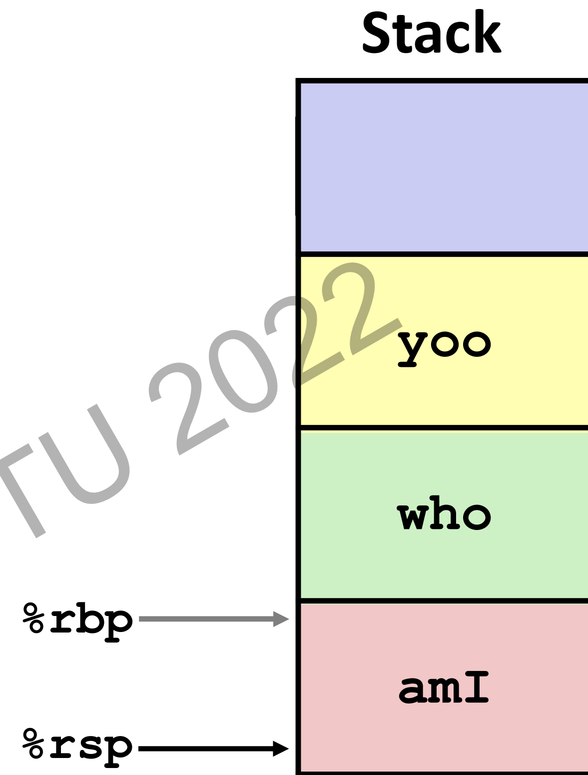


Pengju Ren@XJTU 2022

Example

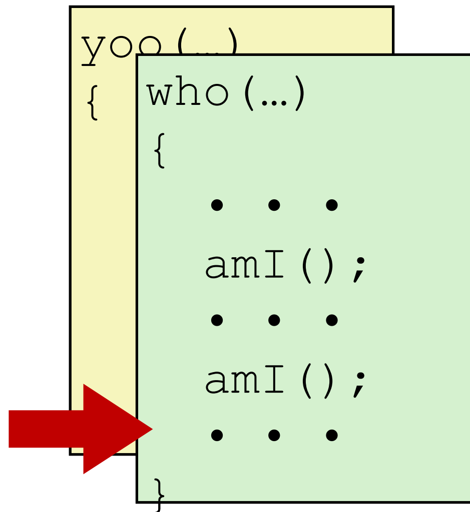


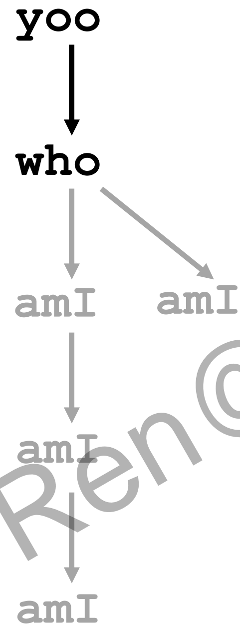


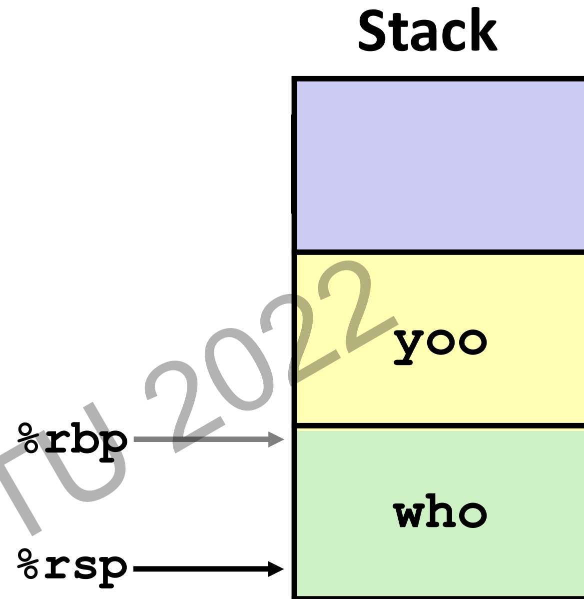


Pengju Ren@XJTU 2022

Example

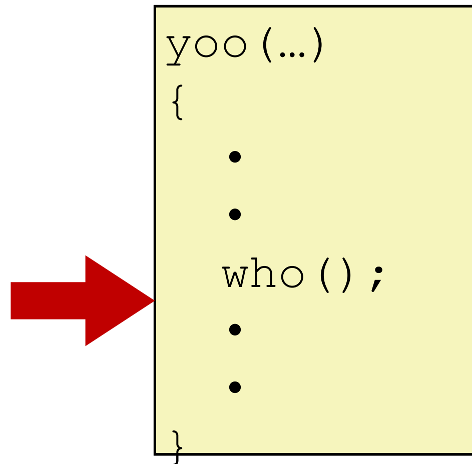


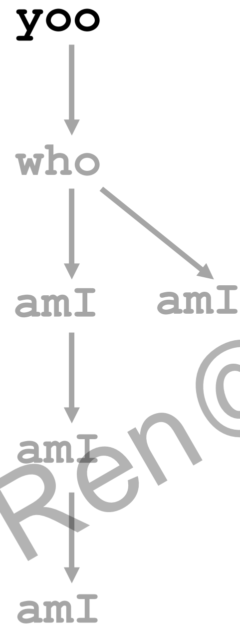


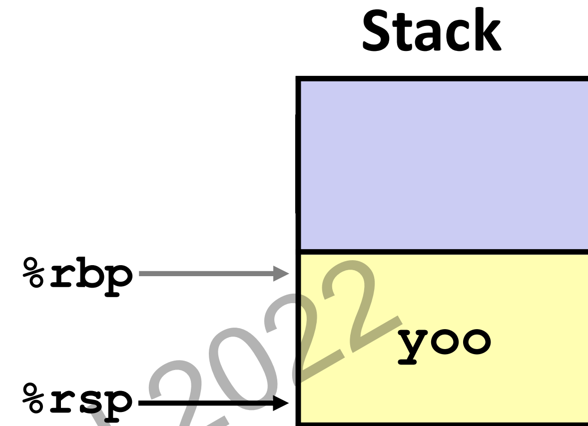


Pengju Ren@XJTU 2022

Example







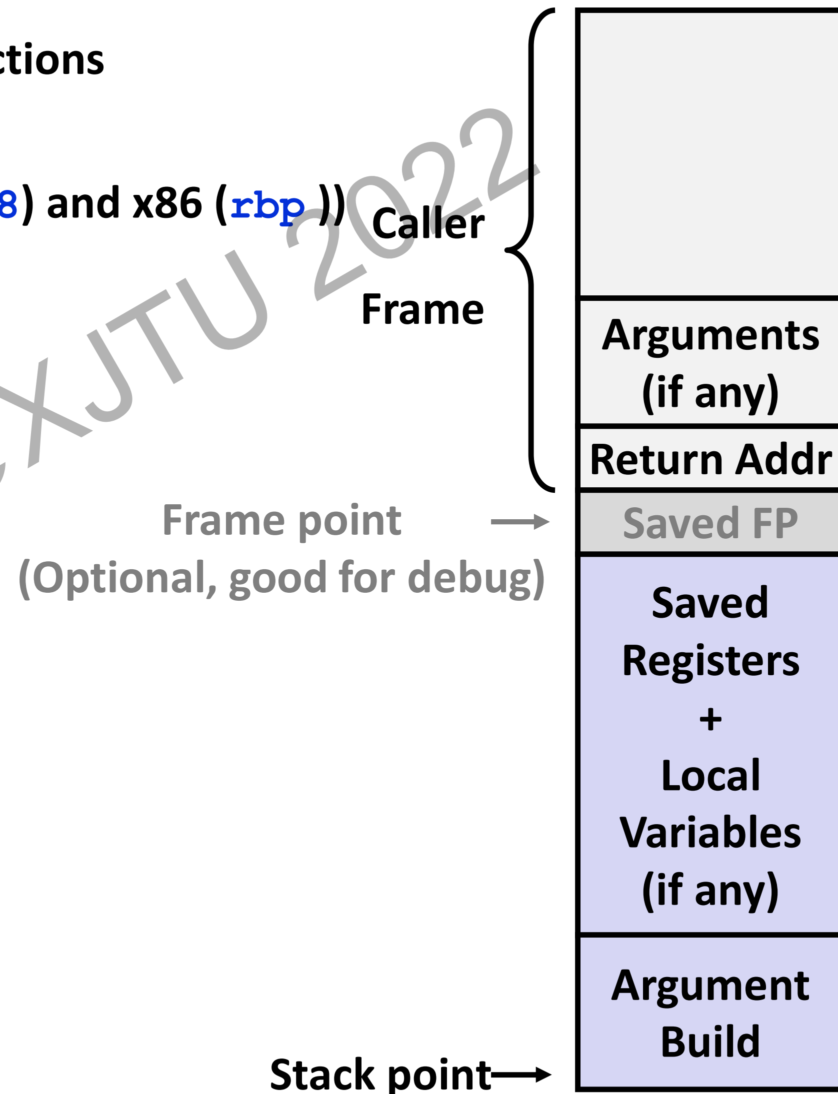
Pengju Ren@XJTU 2022

Understanding Function Calls by Stack

Stack divided into frames
– Each frame stores locals and args to called functions

Stack pointer points to the top of the stack
Frame pointer points to caller's frame (RISC-V (`x8`) and x86 (`rbp`))

- Calling a function
  - Caller
    - Pass arguments
    - Call and save return address
  - Callee
    - Save old frame pointer
    - Set frame pointer = stack pointer
    - Allocate stack space for local storage
- When returning
  - Callee
    - Pop local storage
    - Set stack pointer = frame pointer
    - Pop frame pointer
    - Pop return address and return
  - Caller
    - Pop arguments



Pengju Ren@XJTU 2022

Function Call Example

Callee

```
int Leaf
(int g, int h, int i, int j)
{
int f;
f = (g + h) - (i + j);
return f;
}
```

Parameter variables g, h, i, and j in argument registers a0, a1, a2, and a3.

Assume f in s0 and need one temporary register s1

- Need a place to save old values (s0 and s1) before call function, restore them when return, and then delete
- Ideal is a stack frame (grow stack down from high to low addresses) :
  - *Push*: placing data onto stack (decrements sp)
  - *Pop*: removing data from stack (increments sp)
- Stack in memory, so need register to point to it (sp is the stack pointer in RISC-V (x2) and in x86(rsp))

Pengju Ren@XJTU 2022

RISC-V Code for Leaf()

```
int Leaf
(int g, int h, int i, int j)
{
int f;
f = (g + h) – (i + j);
return f;
}
```

Prologue

```
Leaf:  addi sp,sp,-8 # adjust stack for 2 items
       sw s1, 4(sp) # save s1 for use afterwards
       sw s0, 0(sp) # save s0 for use afterwards
```

Body

```
       add s0,a0,a1 # f = g + h
       add s1,a2,a3 # s1 = i + j
       sub a0,s0,s1 # return value (g + h) – (i + j)
```

```
       lw s0, 0(sp) # restore register s0 for caller
       lw s1, 4(sp) # restore register s1 for caller
       addi sp,sp,8 # adjust stack to delete 2 items
       jr ra # jump back to calling routine
```

Epilogue

if the callee call another function, need to store the return address

```
sw ra,8(sp) # save ret on the top of the stack and the stack should contain more spaces.(addi sp, sp, -12)
```

```
lw ra,8(sp) # get ret addr
```

Pengju Ren@XJTU 2022

Register Conventions Summary

- Register conventions preserves values of registers between function calls
- For RISC-V
  - **Saved registers**: These registers are expected to be the same before and after a function Call. (*Don't touch* or `push(pop)` before(after) function call)
    - `s0-s11`(saved registers), `sp`, `ra`.
  - **Volatile registers**: These registers can be freely changed by the Callee.
    - `t0-t6`(temporary registers), `a0-a7`(return address and arguments)
- **Caller** must save any **volatile registers** it is using onto the stack before making a procedure call
- **Callee** must save any **saved registers** it intends to use by putting them on the stack before overwriting their values.

Pengju Ren@XJTU 2022

NOTEs:

- ONLY the appropriate registers need saved (not ALL!)
- Don't forget to restore the values later

Data in Memory

- **Arrays: a collection of similar data types of elements**
  - **One-dimensional**
  - **Multi-dimensional (nested)**
  - **Multi-level**
- **Structures: suit for dynamic and Non-linear Data structure**
  - **Allocation**
  - **Access**
  - **Alignment**
- **Floating Point**
- **Buffer Overflow**

Pengju Ren@XJTU 2022

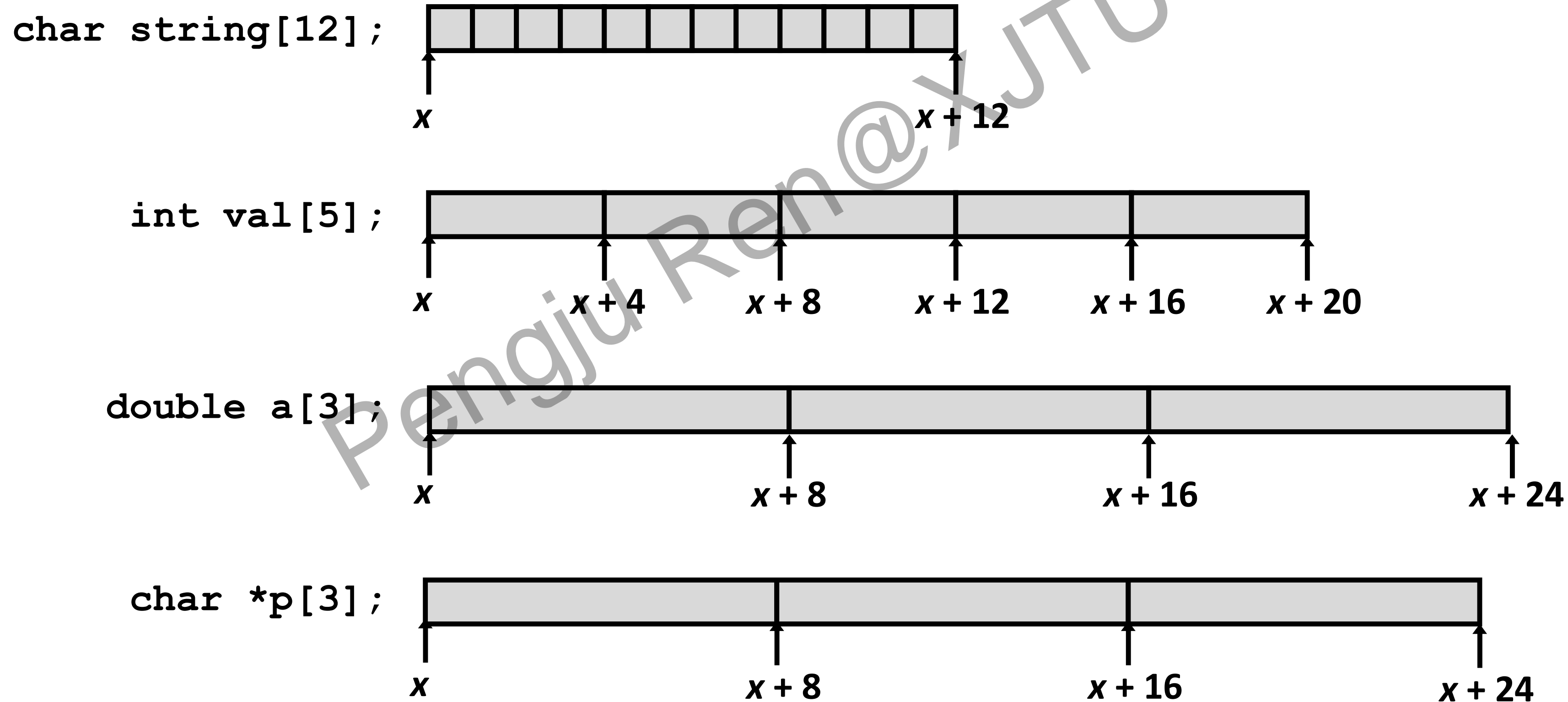
Array Allocation

- **Basic Principle**

  `T A[L];`

  – Array of data type *T* and length *L*

  – Contiguously allocated region of *L* * `sizeof(`*T*`)` bytes in memory



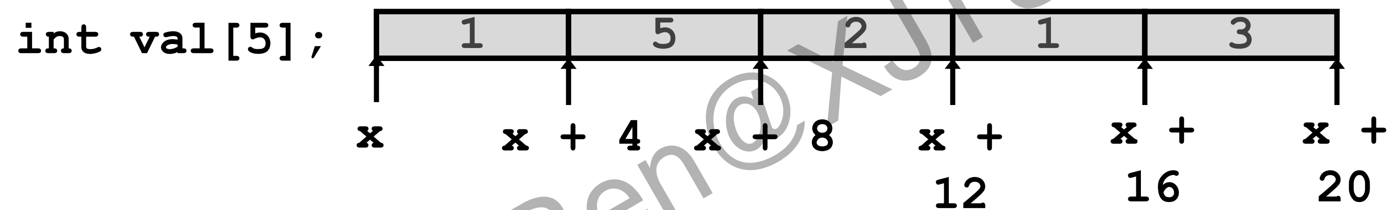
Pengju Ren@XJTU 2022

Array Access

- **Basic Principle**

  T **A**[*L*];

  - Array of data type *T* and length *L*
  - Identifier **A** can be used as a pointer to array element 0: Type *T**



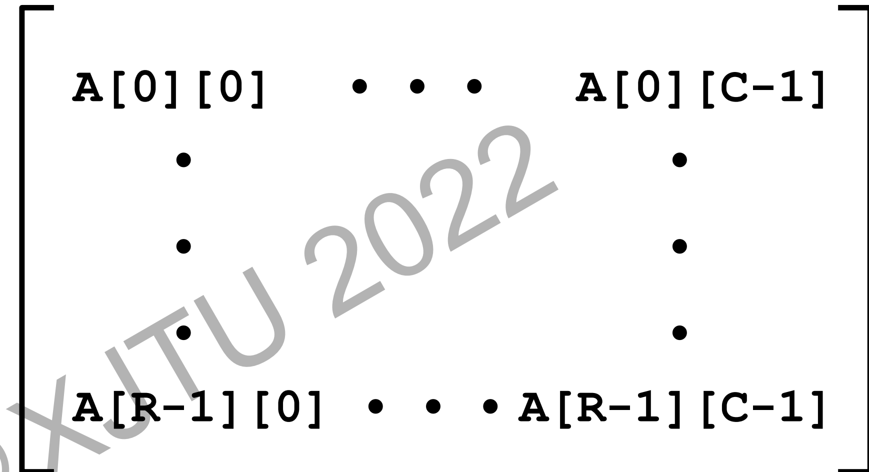
- **Reference Type Value**

Reference	Type	Value
`val[4]`	`int`	`3`
`val`	`int *`	`x`
`val+1`	`int *`	`x + 4`
`&val[2]`	`int *`	`x + 8`
`val[5]`	`int`	`??`
`*(val+1)`	`int`	`5 //val[1]`
`val + i`	`int *`	`x + 4 * i //&val[i]`

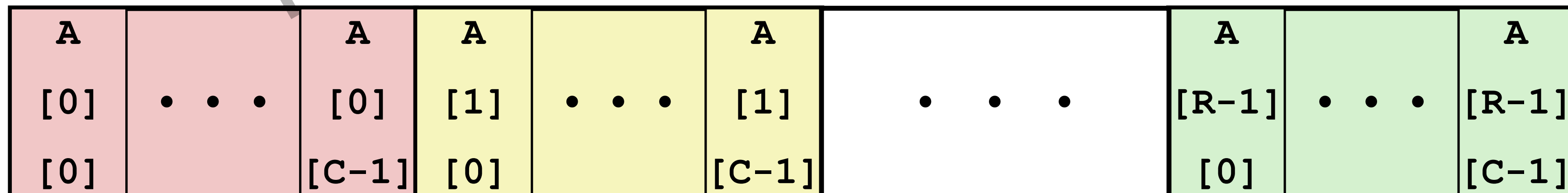
Pengju Ren@XJTU 2022

Multidimensional (Nested) Arrays

- **Declaration**
  - *T* `A`[*R*][*C*];
  - 2D array of data type *T*
  - *R* rows, *C* columns
- **Array Size**
  - *R* * *C* * `sizeof`(*T*) bytes
- **Arrangement**
  - Row-Major Ordering




```
int A[R][C];
```



Pengju Ren@XJTU 2022

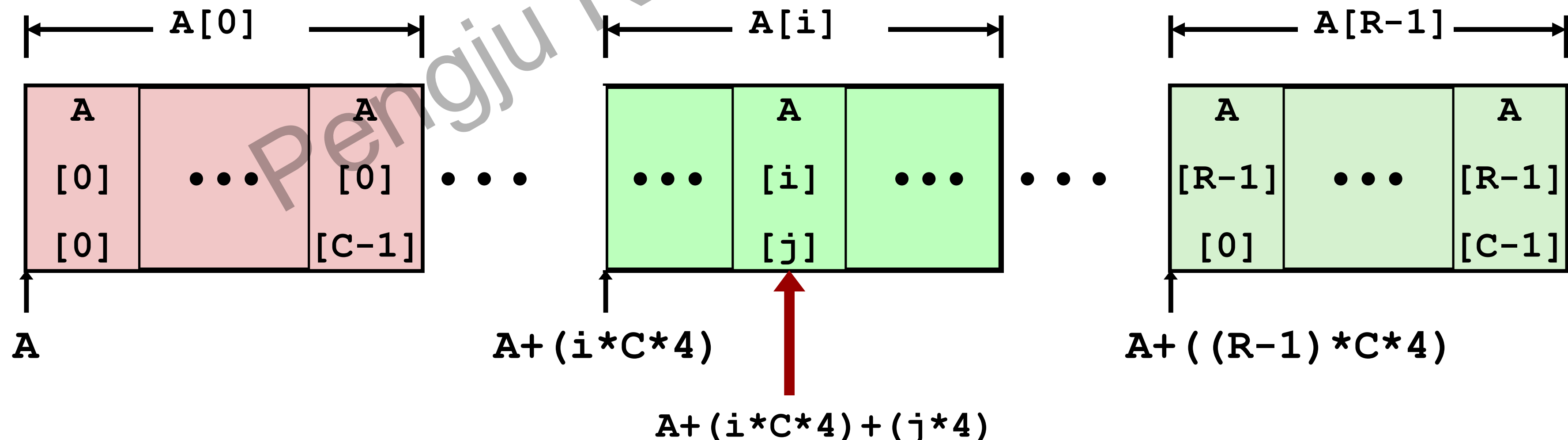
Nested Array Element Access

- **Array Elements (RxC)**
  - `A[i][j]` is element of type *T*, which requires *K* bytes
  - `A` is the base address in Reg X1
  - Element address `A + i * (C`
    `= A + (i * C`
  - `LD/ST X2 X1 #offset`

What is A[i][j], if j > C ?

e.g. A is 5x5 (R=C=5) A[1][7]=?

```
int A[R][C];
```



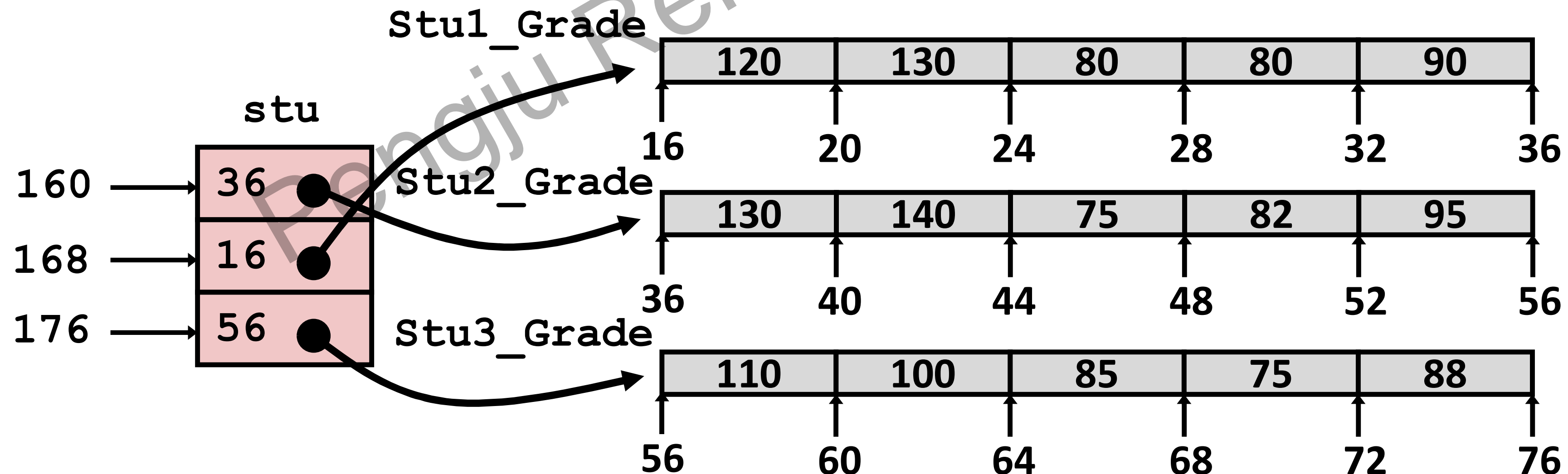
Pengju Ren@XJTU 2022

Multi-Level Array Example

```
int stu1_Grade ={120,130,80,80,90};
int stu2_Grade ={130,140,75,82,95};
int stu3_Grade ={110,100,85,75,88};
```

```
#define UCOUNT 3
int *stu[UCOUNT] = {stu1_Grade,
                    stu2_Grade,
                    stu3_Grade};
```

- Variable `stu` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes
- Each pointer points to array of `int`'s



Pengju Ren@XJTU 2022

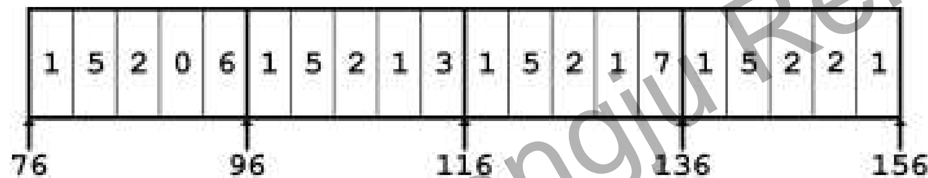
Element Access (Nest Array v.s Multi-level Array)

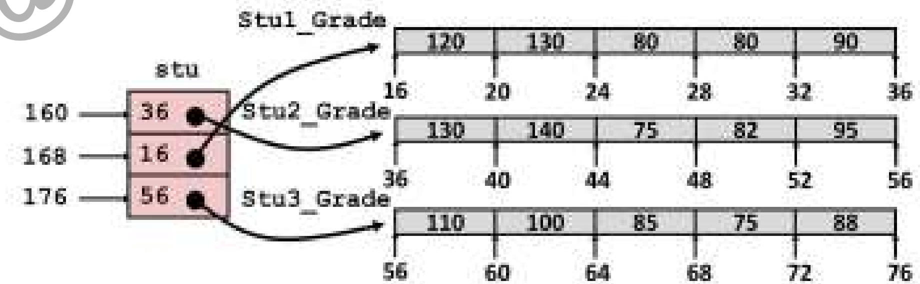
Nested array

```
int get_A_digit
  (size_t i, size_t j)
{
  return A[i][j];
}
```

Multi-level array

```
int get_stu_grade
  (size_t i, size_t j)
{
  return stu[i][j];
}
```





Accesses looks similar in C, but address computations very different:

`Mem[A+20*i+4*j]`

`Mem[Mem[stu+4(or8)*i]+4*j]`

Must do two memory reads, First get pointer to row array, then access element within array

Pengju Ren@XJTU 2022

Data in Memory

- **Arrays: a collection of similar data types of elements**

Applications of Array Data Structure:

- data structures like a stack, queue, etc.
- Matrices (Image, Computer Graph—in the form of an adjacency matrix, etc), Tensor (Video) and other mathematical implementations
- Lookup tables in computers
- Binary search trees and balanced binary trees are used in data structures such as a heap, map, and set, which can be built using arrays

Advantages of array data structure:

- Accessing an element is very easy by using the index, and allows random access
- The search process can be applied to an array easily. (e.g. Search in W/O ordered array)

Disadvantages of array data structure:

- Insertion and deletion operations are costly (use linked list to overcome)
- If the size of the declared array is more than required, it can lead to memory wastage (use dynamic memory allocation to overcome)

Pengju Ren@XJTU 2022

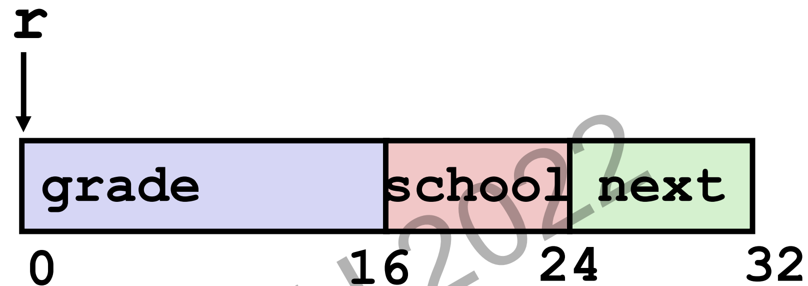
Data in Memory

- **Arrays:** a collection of similar data types of elements
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures:** suit for dynamic and Non-linear Data structure
  - Allocation
  - Access
  - Alignment
- Floating Point
- Buffer Overflow

Pengju Ren@XJTU 2022

Structure Representation

```
struct stu {
    int grade[4];
    char School[8];
    struct stu *next;
};
```

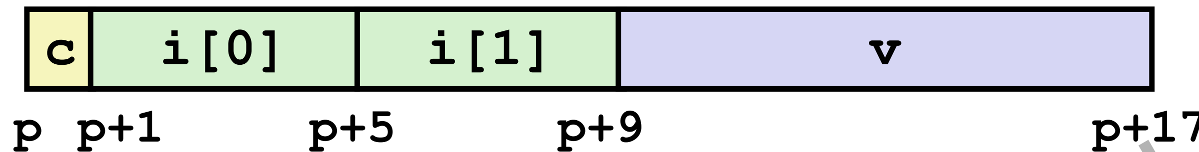


Pengiu Ren@XJTU 2022

- **Structure represented as block of memory**
  - Big enough to hold all of the fields
- **Fields ordered according to declaration**
  - Even if another ordering could yield a more compact representation (*due to alignment rules*—coming soon)
- **Compiler determines overall size + positions of fields**
  - Machine-level program has no understanding of the structures in the source code

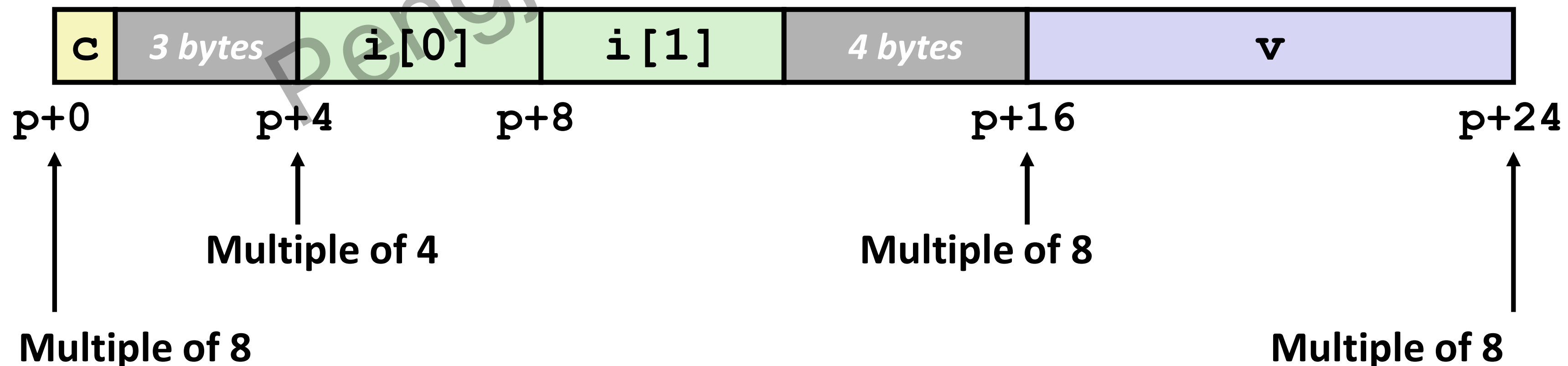
Structures & Alignment

- **Unaligned Data**




```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- **Aligned Data**
  - Primitive data type requires ***B*** bytes implies (in S1, the primitive data type is ***double***), therefore address **must be multiple of *B***



Pengju Ren@XJTU 2022

Alignment Principles

- **Aligned Data**
  - Primitive data type requires ***B*** bytes
  - Address must be multiple of ***B***
  - Required on some machines (advised on x86-64)
- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - » Inefficient to load or store datum that spans cache lines (64 bytes). Intel states should avoid crossing 16 byte boundaries.
    - » Virtual memory trickier when datum spans 2 pages (4 KB pages)
- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields

Pengju Ren@XJTU 2022

Specific Cases of Alignment (x86-64)

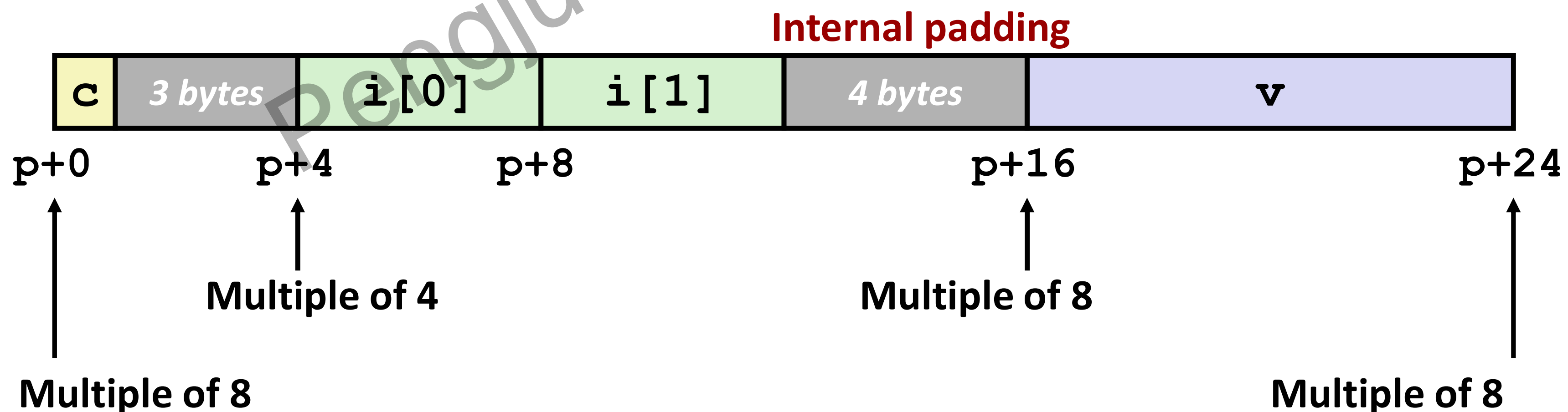
- **1 byte: `char`, …**
  - no restrictions on address
- **2 bytes: `short`, …**
  - lowest 1 bit of address must be 0_2
- **4 bytes: `int`, `float`, …**
  - lowest 2 bits of address must be 00_2
- **8 bytes: `double`, `long`, `char *`, …**
  - lowest 3 bits of address must be 000_2

Pengju Ren@XJTU 2022

Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy each element's alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement K
    - » K = Largest alignment of any element
  - Initial address & structure length must be multiples of K
- **Example:**
  - K = 8, due to `double` element

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

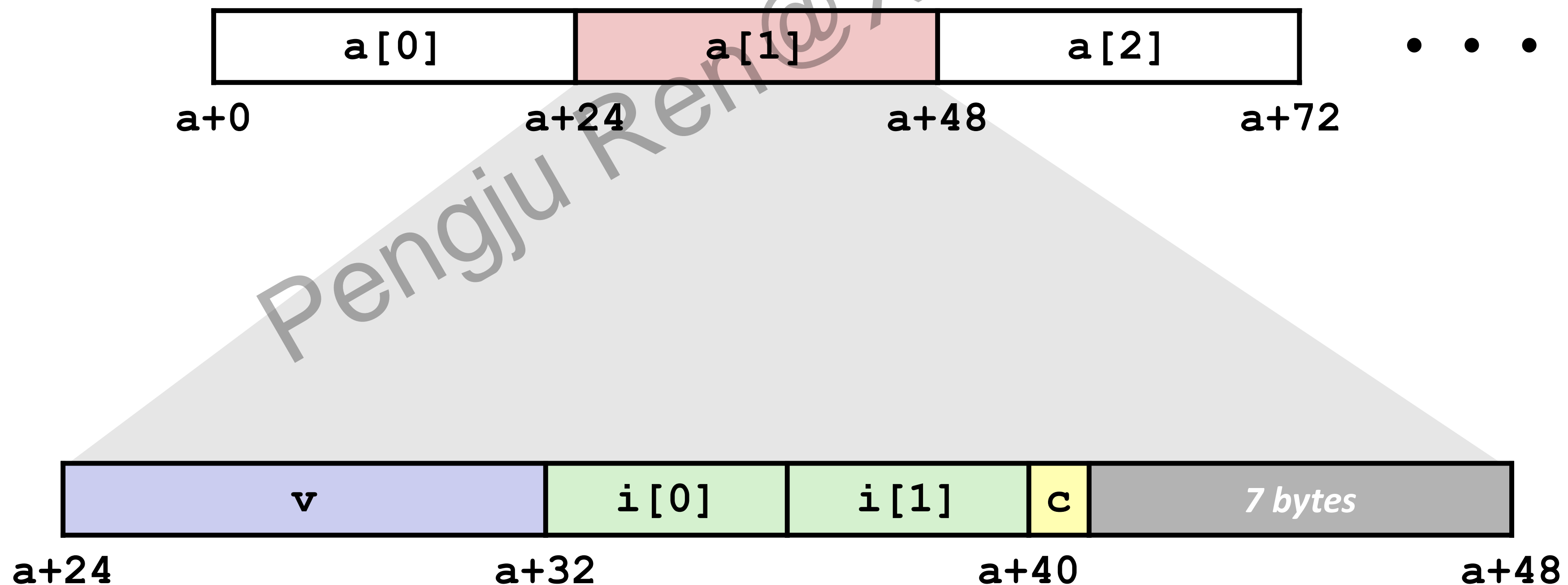


Pengju Ren@XJTU 2022

Array of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Pengju Ren@XJTU 2022

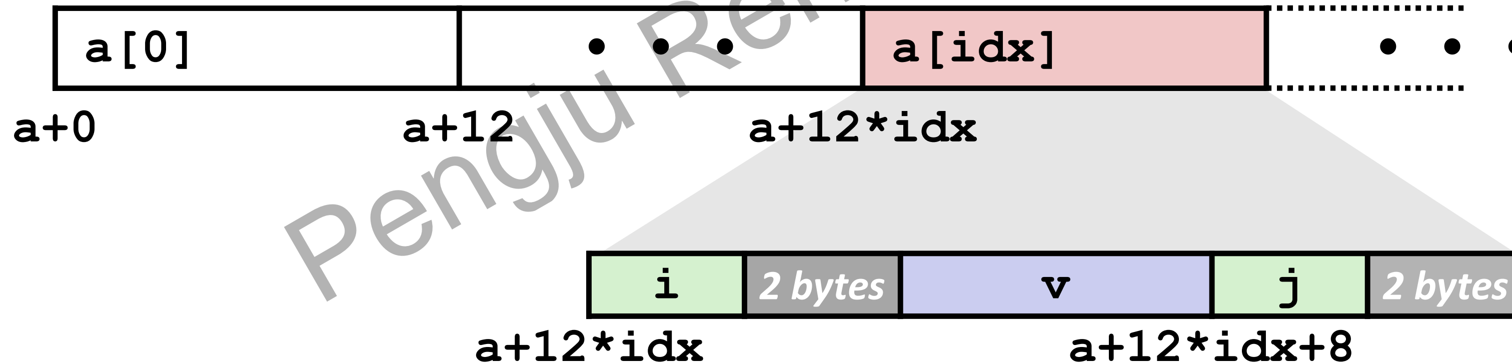
Accessing Array Elements

- **Compute a**
  - `sizeof(`

Can we save the size of a[i]?

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```

- **Element `j` is at offset 8 within structure**
- **Assembler gives offset a+8**
  - Resolved during linking

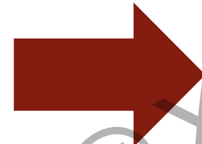


Pengju Ren@XJTU 2022

Saving Space

- **Put same data types together, then decreasing (or increasing) data types**

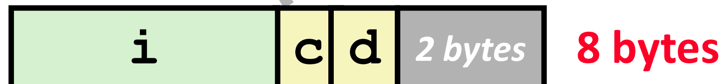
```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```



```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

12 bytes

- **Effect (largest alignment requirement K=4)**



8 bytes

- **Saving Space is good for Memory Hierarchy (Cache .etc)**

Rengju Ren@XJTU 2022

Example Struct Exam Question

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct {
  char a;
  long b;
  float c;
  char d[3];
  int *e;
  short *f;
} foo;
```

1. Show how `foo` would be allocated in memory on an x86-64 Linux system. Label the bytes with the names of the various fields and **clearly mark the end of the struct.** Use an X to denote space that is allocated in the struct as padding.

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Pengju Ren@XJTU 2022

Example Struct Exam Question

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct {
  char a;
  long b;
  float c;
  char d[3];
  int *e;
  short *f;
} foo;
```

1. Show how `foo` would be allocated in memory on an x86-64 Linux system. Label the bytes with the names of the various fields and **clearly mark the end of the struct.** Use an X to denote space that is allocated in the struct as padding.

Can we do better？ How？

Pengju Ren@XJTU 2022

Example Struct Exam Question

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct {
  char a;
  long b;
  float c;
  char d[3];
  int *e;
  short *f;
} foo;
```

2. Rearrange the elements of `foo` to conserve the most space in memory. Label the bytes with the names of the various fields and **clearly mark the end of the struct.** Use an X to denote space that is allocated in the struct as padding.

Pengju Ren@XJTU 2022

Data in Memory

- **Arrays:** a collection of similar data types of elements
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures:** suit for dynamic and Non-linear Data structure
  - Allocation
  - Access
  - Alignment
- **Floating Point**
- **Buffer Overflow**

Pengju Ren@XJTU 2022

How do we encode the following?

Real numbers (e.g. π: 3.1415926…)

Very large numbers (e.g. NA: 6.02×10^{23})

Very small numbers (e.g. h: 6.626×10^{-34})

Special numbers (e.g. ∞, NaN)

$$(-1)^S M 2^E$$

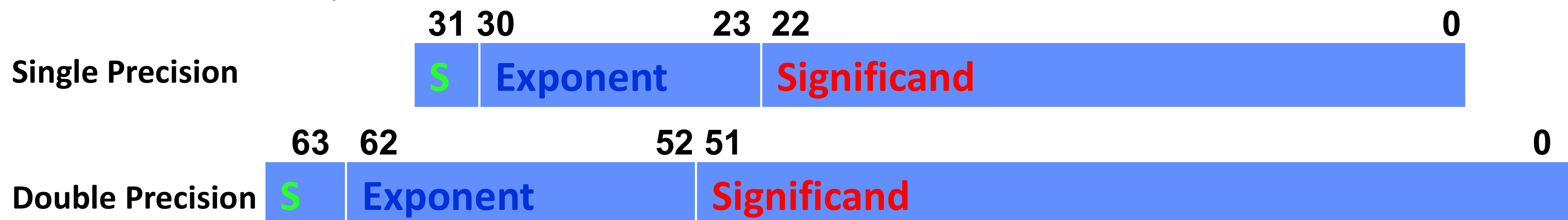
Split into 3 fields:

— S represents Sign (1 is negative, 0 positive)

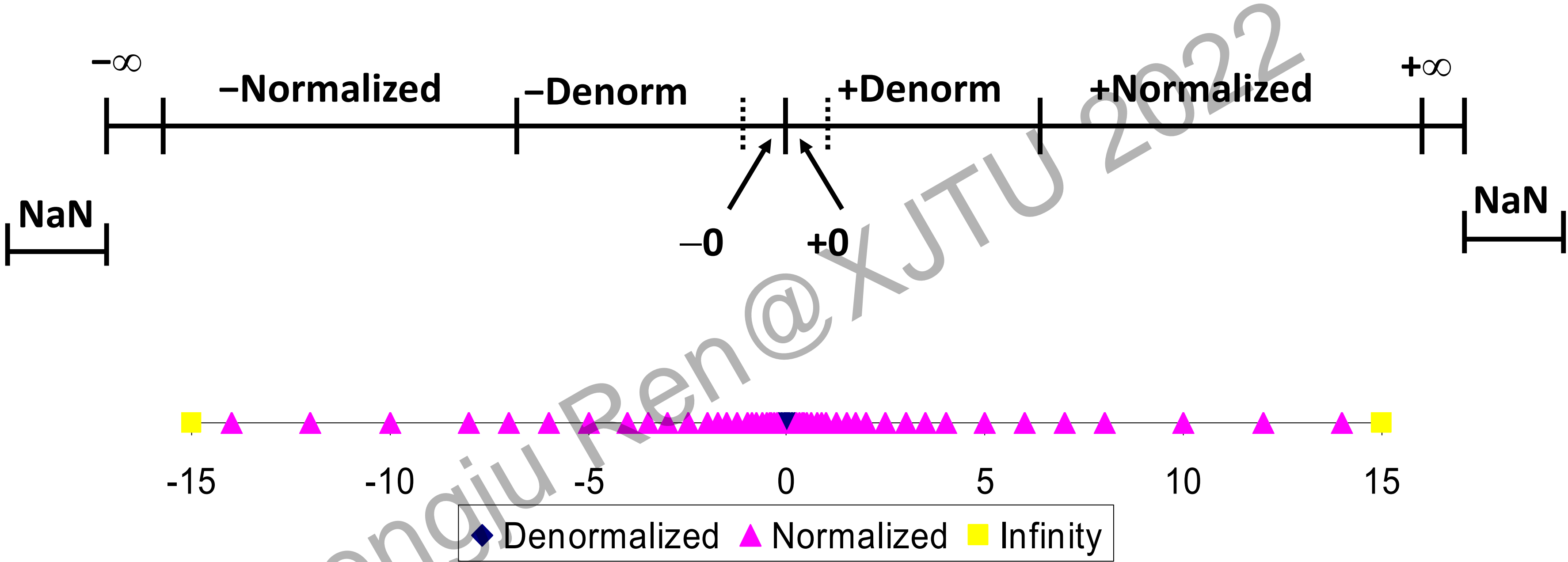
— Exponent E（weights value by power of 2)

— Significand M (normally a fractional value in range 0~1)

Pengju Ren@XJTU 2022



Visualization Floating Point



Pengju Ren@XJTU 2022

The Exponent Field

exp ≠ 0 and exp ≠ 11…11

00…00

11…11

Normalized Values

Use biased notation : *E=exp-Bias*

Bias = $2^{k-1} - 1$, k is the number of exponent bits

Single precision: 127 (exp: 1…254, E: -126…127)

Double precision: 1023 (exp: 1…2046, E: -1022…1023)

Significand coded with implied leading 1: $M = 1.xx \ldots x_2$

Example: *float F = 15213.0;*

$15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13}$

S = 0

E = 13+bias = 13+127=140= 10001100_2

M = 1.1101101101101_2 frac= $11011011011010000000000_2$

Rengju Ren@XJTU 2022

The Exponent Field

exp ≠ 0 and exp ≠ 11…11

00…00

11…11

Denormalized Values

Use biased notation : *E= 1 - Bias (Bias* = $2^{k-1} - 1$*)*

Significand **coded with implied leading** ***0****:* $M = 0.xx \dots x_2$

Cases

- `exp` = 000…0, `frac` = 000…0
  - Represents **zero value**
- `exp` = 000…0, `frac` ≠ 000…0
  - Numbers closest to 0.0
  - Equi-spaced

Pengju Ren@XJTU 2022

The Exponent Field

S	Exponent	Significand

exp ≠ 0 and exp ≠ 11…11

00…00

11…11

Special Cases

- `exp = 111…1, frac = 000…0`
  - Represents value ∞ (infinity)
  - Both positive and negative
  - E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞
- `exp = 111…1, frac ≠ 000…0`
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., ∞ − ∞, ∞ × 0

Pengju Ren@XJTU 2022

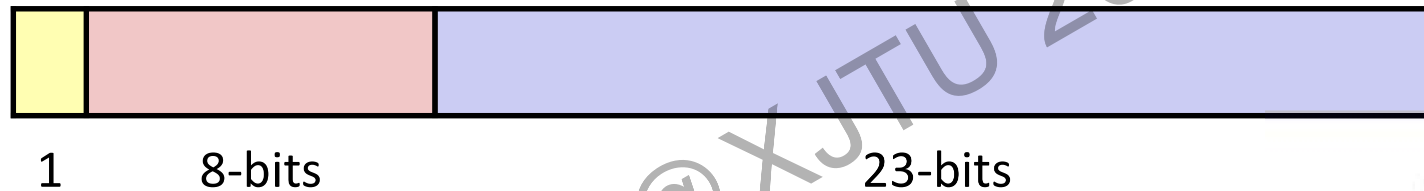
C float Decoding Example

$v = (-1)^s M\, 2^E$
$E = \texttt{exp} - Bias$

Bias = $2^{k-1} - 1 = 127$

float: `0xC0A00000`

binary: ___ ___ ___ ___ ___ ___ ___ ___



1 8-bits 23-bits

E =

S =

M =

v = $(-1)^s$ M 2^E =

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

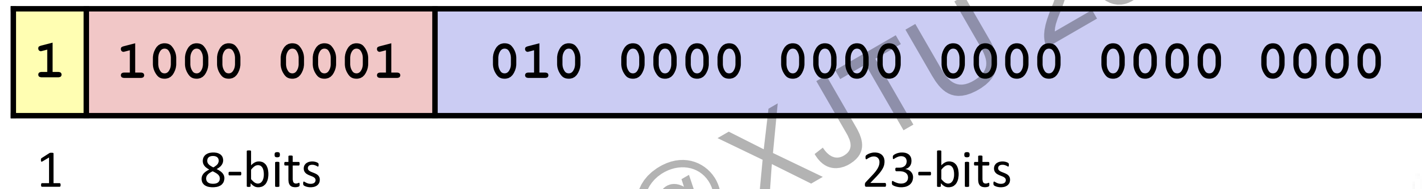
Pengju Ren@XJTU 2022

C float Decoding Example #1

$$v = (-1)^s \, M \, 2^E$$
$$E = \texttt{exp} - Bias$$

float: `0xC0A00000`

binary: 1100 0000 1010 0000 0000 0000 0000 0000



1 8-bits 23-bits

E =

S =

M = 1 .

$v = (-1)^s \, M \, 2^E =$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Pengju Ren@XJTU 2022

C float Decoding Example #1

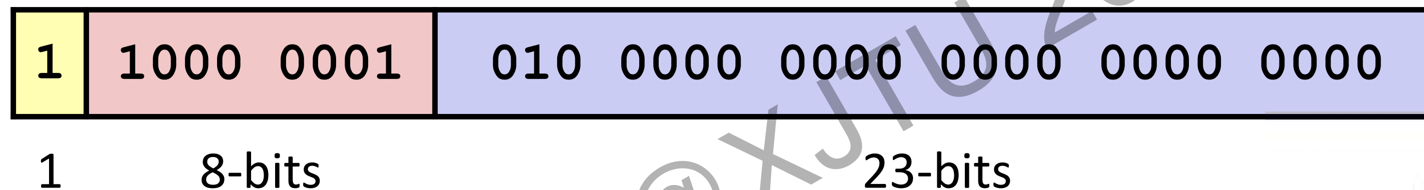
$v = (-1)^s M 2^E$

$E = \texttt{exp} - Bias$

$Bias = 2^{k-1} - 1 = 127$

float: 0xC0A00000

binary: 1100 0000 1010 0000 0000 0000 0000 0000



E = exp – Bias = 129 – 127 = 2 (decimal)

S = 1 -> negative number

M = 1.010 0000 0000 0000 0000 0000

= 1 + 1/4 = 1.25

$v = (-1)^s M 2^E = (-1)^1 * 1.25 * 2^2 = -5$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Pengju Ren@XJTU 2022

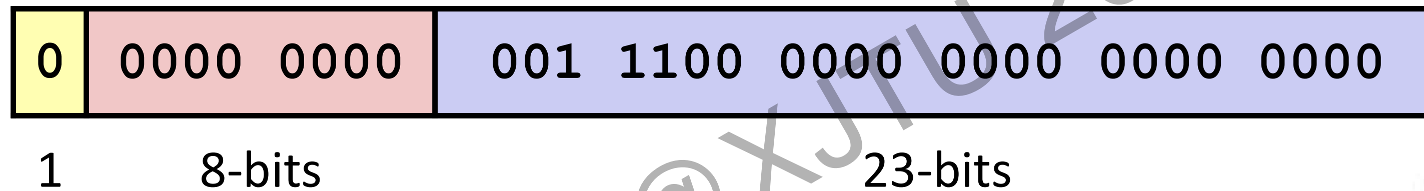
C float Decoding Example #2

$v = (-1)^s M\, 2^E$
$E = 1 - Bias$

float: 0x001C0000

Bias = $1\text{-}2^{k-1} - 1 = -126$

binary: 0000 0000 0001 1100 0000 0000 0000 0000



E =

S =

M = 0 .

v = $(-1)^s M\, 2^E$ =

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Pengju Ren@XJTU 2022

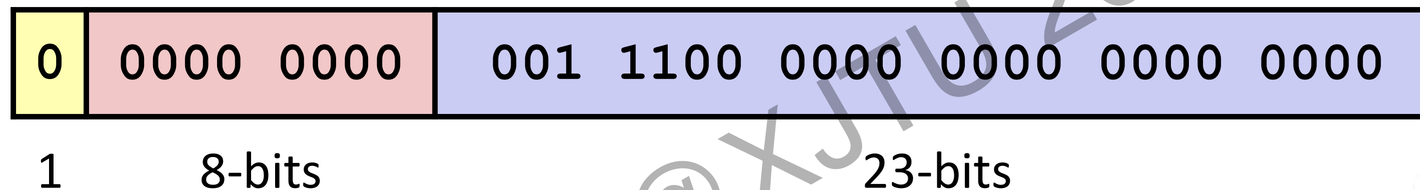
C float Decoding Example #2

$v = (-1)^s M 2^E$
$E = 1 - Bias$

$Bias = 1\text{-}2^{k-1} - 1 = -126$

float: 0x001C0000

binary: 0000 0000 0001 1100 0000 0000 0000 0000



E = 1 – Bias = 1 – 127 = –126 (decimal)

S = 0 -> positive number

M = 0.001 1100 0000 0000 0000 0000

= 1/8 + 1/16 + 1/32 = 7/32 = $7*2^{-5}$

$v = (-1)^s M 2^E = (-1)^0 * 7*2^{-5} * 2^{-126} = 7*2^{-131}$

$\approx 2.571393892 \times 10^{-39}$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Pengju Ren@XJTU 2022

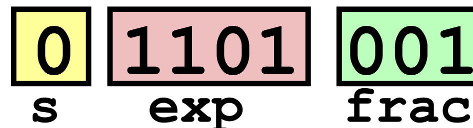
FP Multiplication

- $(-1)^{s1}\ \boldsymbol{M1}\ 2^{E1} \times (-1)^{s2}\ \boldsymbol{M2}\ 2^{E2}$
- Exact Result: $(-1)^{s}\ \boldsymbol{M}\ 2^{E}$
  - Sign *s*: *s1 ^ s2*
  - Significand *M*: *M1* x *M2*
  - Exponent *E*: *E1* + *E2*
- Fixing
  - If *M* ≥ 2, shift *M* right, increment *E*
  - If *E* out of range, overflow
  - Round *M* to fit `frac` precision

FP8 FORMATS FOR DEEP LEARNING (NV, ARM, Intel)

FP8: E4M3 & E5M2

Bias = $2^{k-1} - 1$ = 7/15



4 bit significand: $1.010*2^2 \times 1.110*2^3 = 10.0011*2^5$

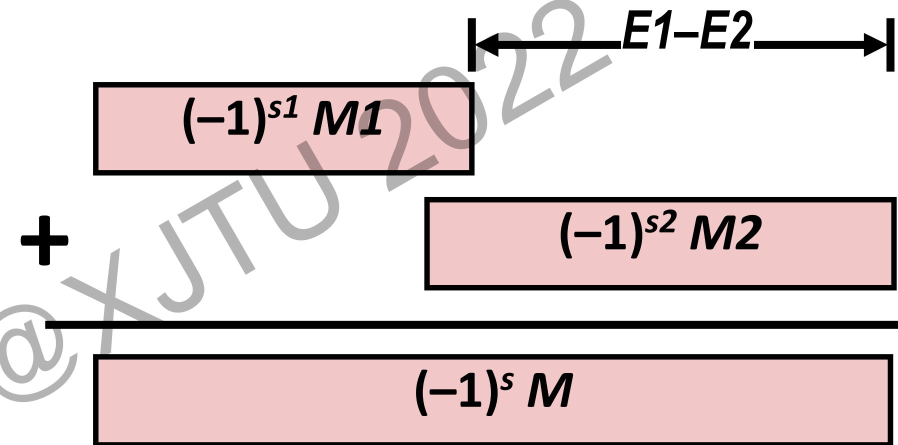
$= 1.00011*2^6 = 1.001*2^6$

Pengju Ren@XJTU 2022

Floating Point Addition

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
  - –Assume $E1 > E2$
- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - –Sign s, significand M:
    - » Result of signed align & add
  - –Exponent E: $E1$
- **Fixing**
  - –If $M \geq 2$, shift M right, increment E
  - –if $M < 1$, shift M left k positions, decrement E by k
  - –Overflow if E out of range
  - –Round M to fit `frac` precision

Get binary points lined up



FP8: E4M3

$1.010*2^2 + 1.110*2^3 = (0.1010 + 1.1100)*2^3$
$= 10.0110 * 2^3 = 1.00110 * 2^4 = 1.010 * 2^4$

Pengju Ren@XJTU 2022

Precision and Accuracy

Precision is a count of the number of bits in a computer word used to represent a value

Accuracy is a measure of the difference between the actual value of a number and its computer representation

- **High precision permits high accuracy but doesn't guarantee it. It is possible to have high precision but low accuracy.**
- **Example: float π= 3.14**

π will be represented using all 24 bits of the significand (highly precise), but is only an approximation (not accurate)

Pengju Ren@XJTU 2022

Floating Point Limitations

- **FP addition and multiplication are NOT associative!**

(x+y)+z = x+(y+z)

– **You can find Big and Small numbers such that (FP32):**

Big + Small + Small ≠ + Big + (Small + Small)

$$3.14 + 1 \times 10^{10} - 1 \times 10^{10} = ?$$
$$3.14 + (1 \times 10^{10} - 1 \times 10^{10}) = ?$$
$$1 \times 10^{20} \times 1 \times 10^{20} \times 1 \times 10^{-20} = ?$$
$$1 \times 10^{20} \times (1 \times 10^{20} \times 1 \times 10^{-20}) = ?$$

– **This is due to rounding errors: FP approximates results**

Pengju Ren@XJTU 2022

Data in Memory

- **Arrays:** a collection of similar data types of elements
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures:** suit for dynamic and Non-linear Data structure
  - Allocation
  - Access
  - Alignment
- **Floating Point**
- **Buffer Overflow**

Pengju Ren@XJTU 2022

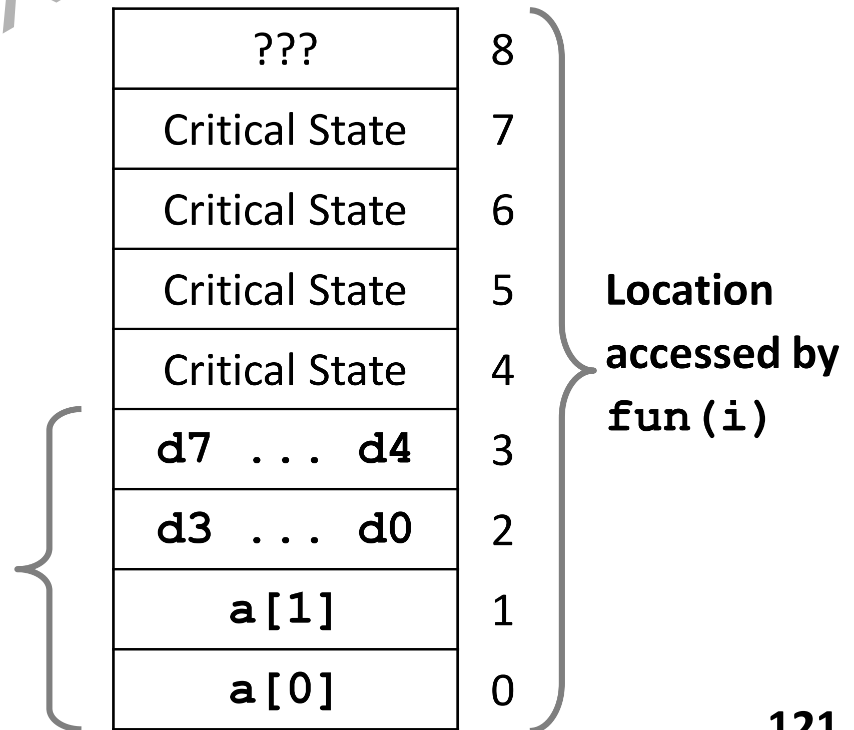
Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

```
double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0)  -> 3.1400000000
fun(1)  -> 3.1400000000
fun(2)  -> 3.1399998665
fun(3)  -> 2.0000006104
fun(4)  -> Segmentation fault
fun(8)  -> ???
```

Explanation:



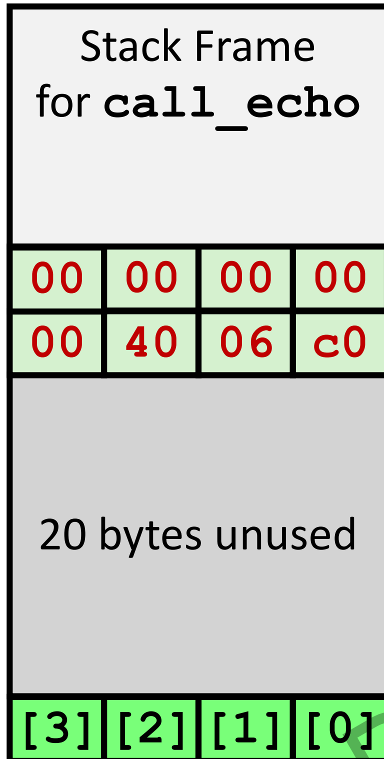
Pengju Ren@XJTU 2022

Such problems are a BIG deal

- **Generally called a "buffer overflow"**
  - when exceeding the memory size allocated for an array
- **Why a big deal?**
  - It's the #1 technical cause of security vulnerabilities
  - What is #1 overall cause?
    - » social engineering / user ignorance
- **Most common form**
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - » sometimes referred to as stack smashing

Pengju Ren@XJTU 2022

Buffer Overflow Stack Example




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .}
```

```
echo:
  400690: addi sp, sp, -24
  400694: sw   ra, 20(sp)
  400698: sw   s1, 16(sp)
  40069c: jal  gets
  . . .
```

```
void call_echo() {
    echo();
}
```

```
  . . .
  4006bc: jal  echo
  4006c0: addi x1, x3
  . . .
```

Pengju Ren@XJTU 2022

Buffer Overflow Stack Example

Before call to gets

Stack Frame for `call_echo`			
00	00	00	00
00	40	06	c0
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %sp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .}
```

```
echo:
  400690: addi sp, sp, -24
  400694: sw   ra, 20(sp)
  400698: sw   s1, 16(sp)
  40069c: jal  gets
  . . .
```

```
void call_echo() {
    echo();
}
```

```
  . . .
  4006bc: jal  echo
  4006c0: addi x1, x3
  . . .
```

```
unix>./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

"01234567890123456789012**3**\0"

Overflowed buffer, but did not corrupt state

Pengju Ren@XJTU 2022

Buffer Overflow Stack Example

Before call to gets

Stack Frame for `call_echo`			
00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %sp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .}
```

```
echo:
 400690: addi sp, sp, -24
 400694: sw   ra, 20(sp)
 400698: sw   s1, 16(sp)
 40069c: jal  gets
   . . .
```

```
void call_echo() {
    echo();
}
```

```
  . . .
  4006bc: jal  echo
  4006c0: addi x1, x3
  . . .
```

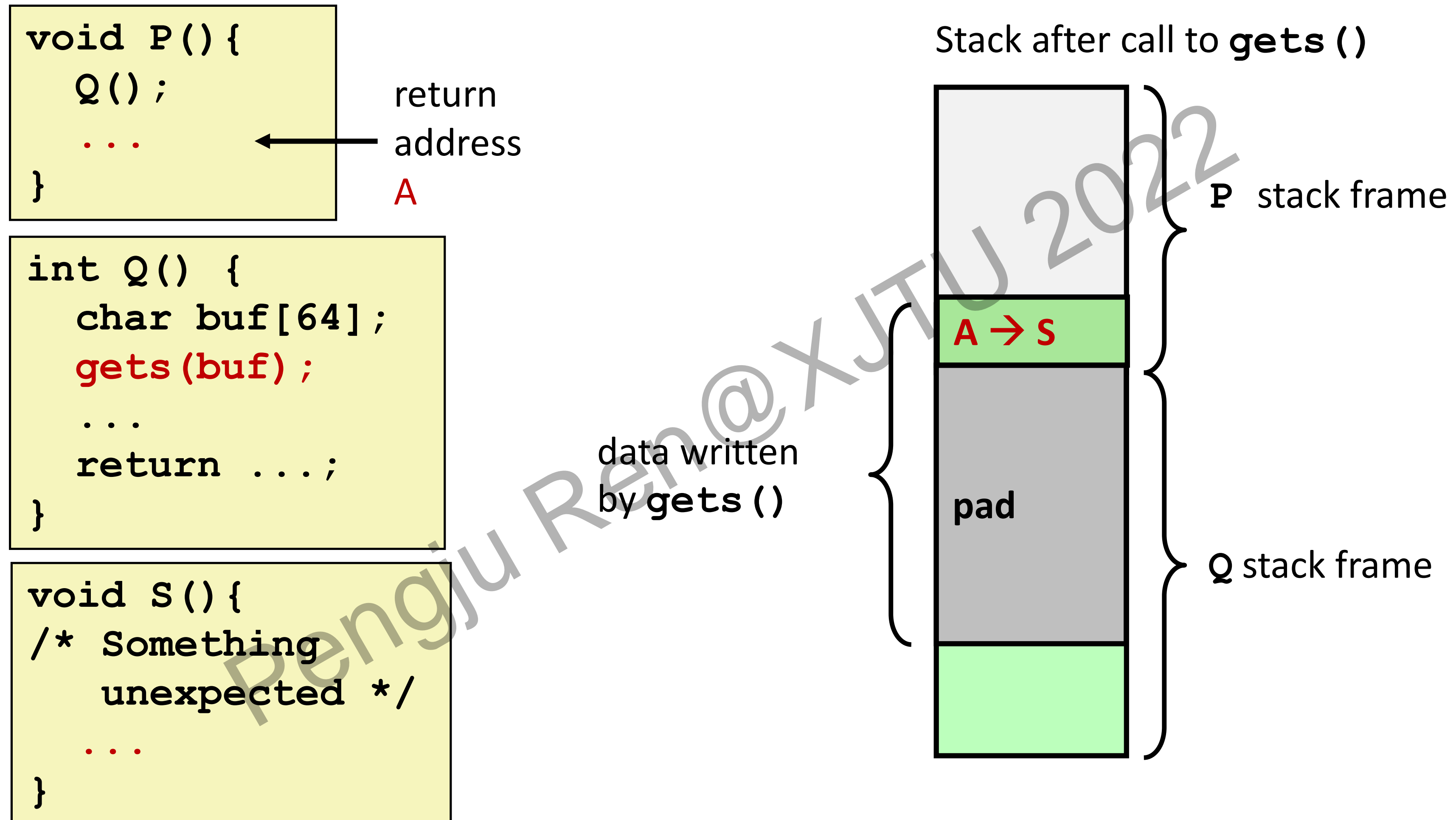
```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
Segmentation fault
```

"012345678901234567890 12\0"

Program "returned" to 0x0400600, and then crashed.

Pengju Ren@XJTU 2022

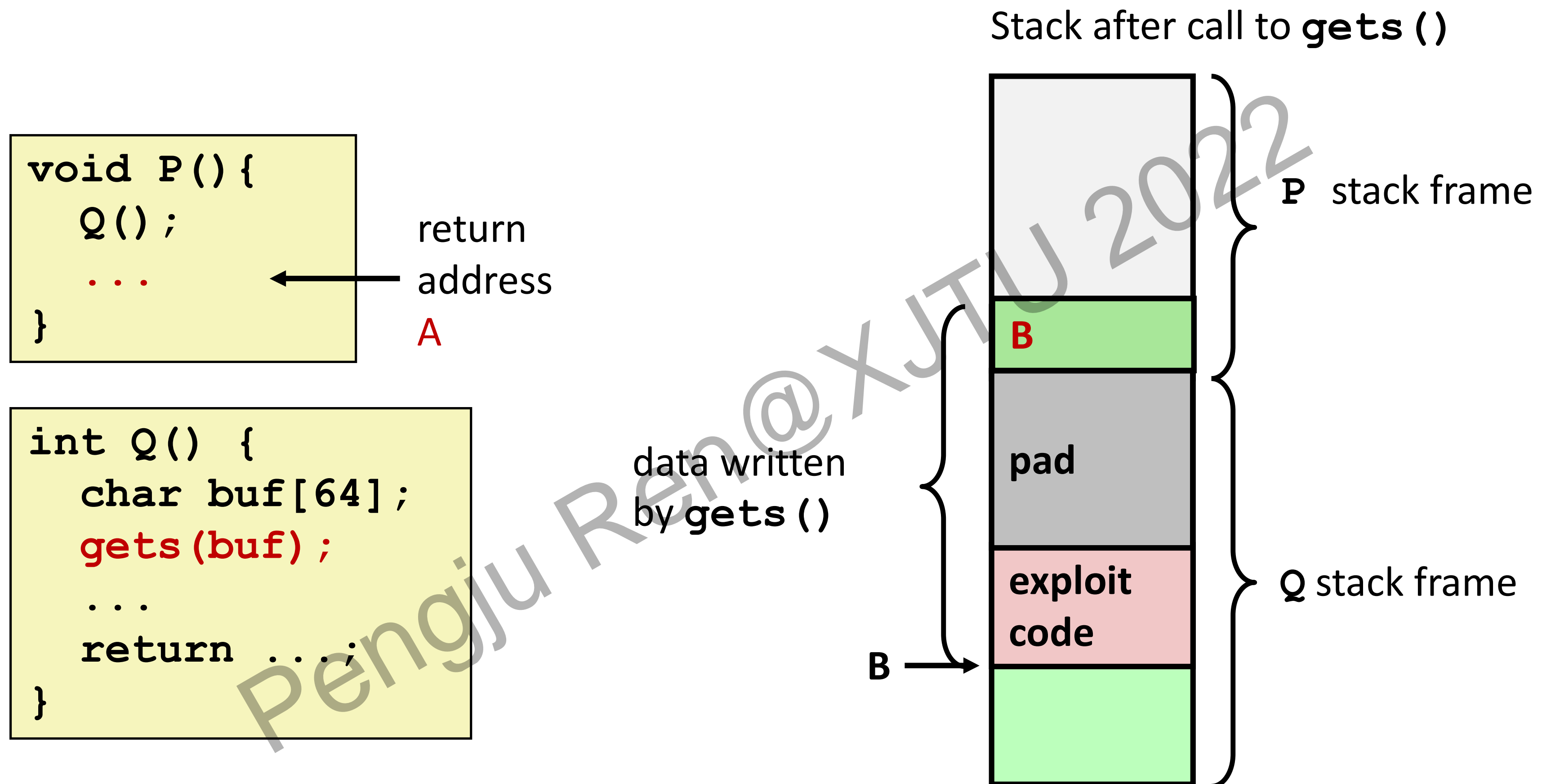
Stack Smashing Attacks



Pengju Ren@XJTU 2022

- Overwrite normal return address A with address of some other code S
- When `Q` executes `ret`, will jump to other code

Code Injection Attacks



- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When `Q` executes `ret`, will jump to exploit code

Pengju Ren@XJTU 2022

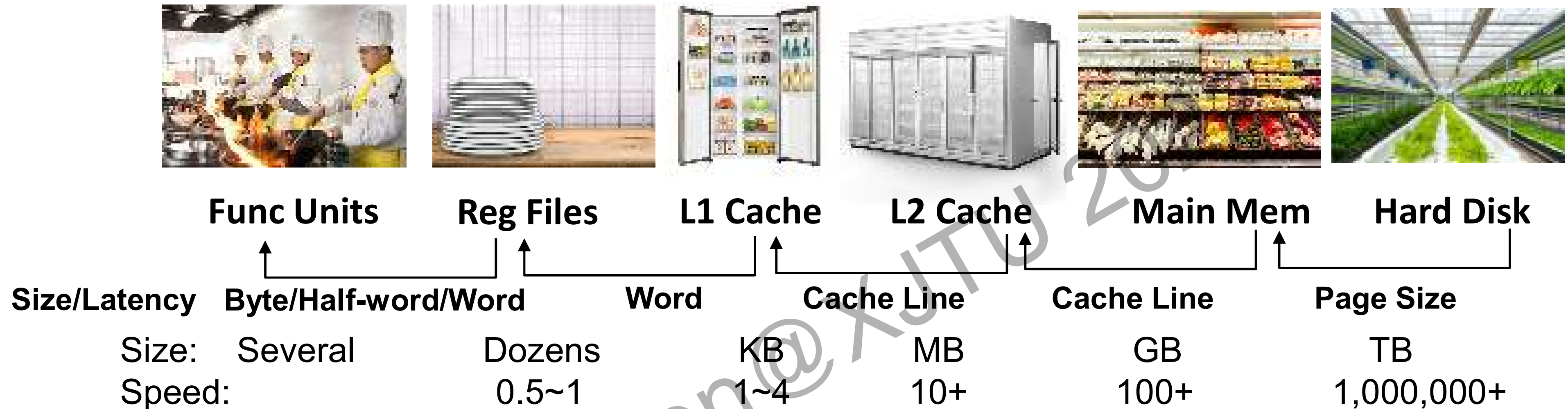
How Does The Attack Code Execute?

```
void P(){
  Q();
  ...
}
```

```
int Q() {
  char buf[64];
  gets(buf); // A->B
  ...
  return ...;
}
```


Pengju Ren@XJTU 2022

Analogy : Computing v.s Cooking



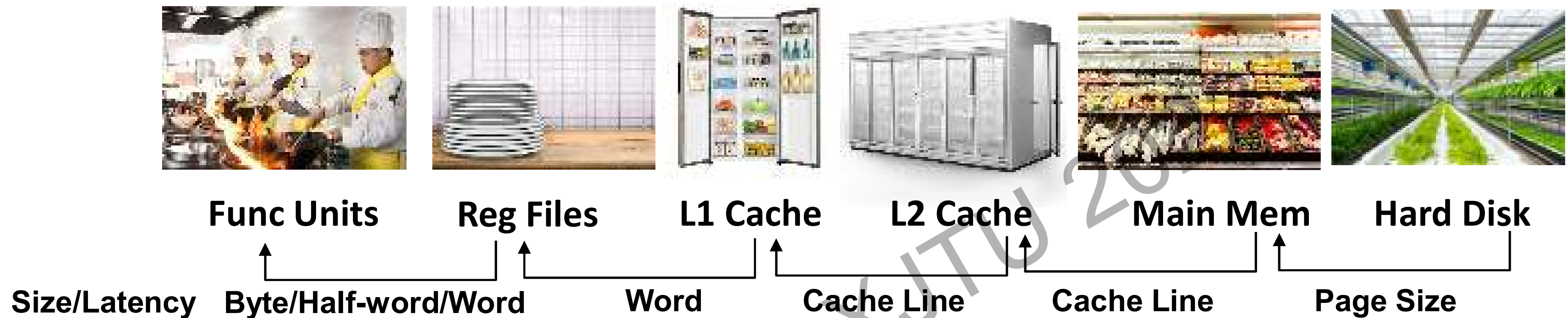
Pengju Ren@XJTU 20

Processor Design:

- **Pipeline ?**
- **Multi Function Units?**
- **SuperScalar (#-way-#issue)**
- **Distributed Issue Queue? Centralized Queue ?**
- **Data Hazard ? Control Hazard ? Structure Hazard ?**
- **Instruction-level Parallel ? Data-level Parallel ? Thread-level Parallel ?**
- **Manycore, Heterogeneous ? Big.Little Arch?**

…

Analogy : Computing v.s Cooking



Memory:

- Mem hierarchy ? Mem replacement Policy ?
- Cache Locality (Temporal, Spacial)
- Write Back/ Write Through
- Cache Coherence ? (Snoopy based, Directory based)

...

Others:

- Throughput? Latency?
- Dishes menu
- Interrupt?
- Task mapping, Scheduling, workload balance

...

Pengju Ren@XJTU 20

Appendix A: Avoid Buffer Overflow Attack

Pengju Ren@XJTU 2022

What To Do About Buffer Overflow Attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”

Pengju Ren@XJTU 2022

1. Avoid Overflow Vulnerabilities in Code (!)

```
/* Echo Line */
void echo()
{
    char buf[4];
    fgets(buf, 4, stdin);
    puts(buf);
}
```

Pengju Ren@XJTU 2022

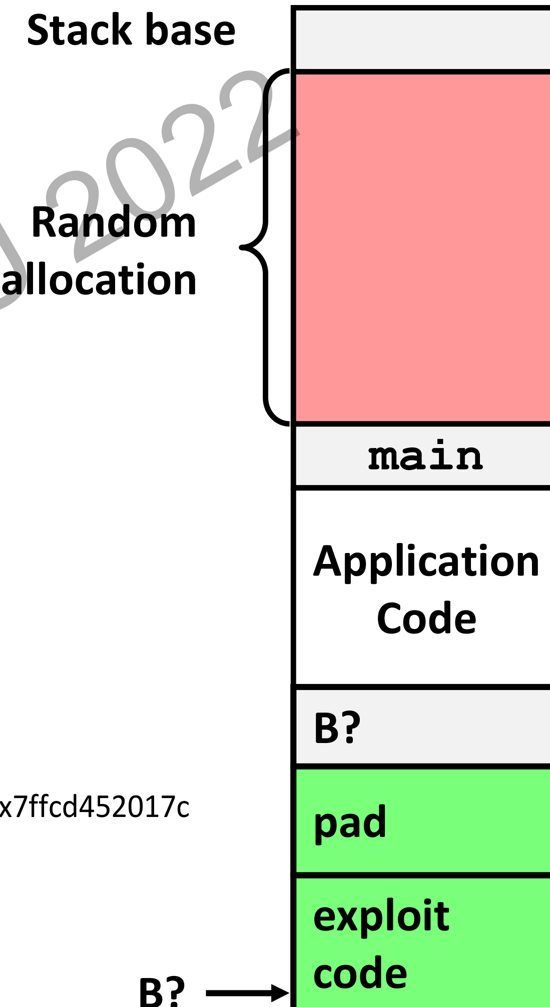
- For example, use library routines that **limit string lengths**
  - **fgets** instead of **gets**
  - **strncpy** instead of **strcpy**
  - Don't use **scanf** with **%s** conversion specification
    - » Use **fgets** to read the string
    - » Or use **%ns** where **n** is a suitable integer

2. System-Level Protections can help

- Randomized stack offsets
  - At start of program, **allocate random amount of space on stack**
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code
  - E.g.: 5 executions of memory allocation code

local	0x7ffe4d3be87c	0x7fff75a4f9fc	0x7ffeadb7c80c	0x7ffeaea2fdac	0x7ffcd452017c

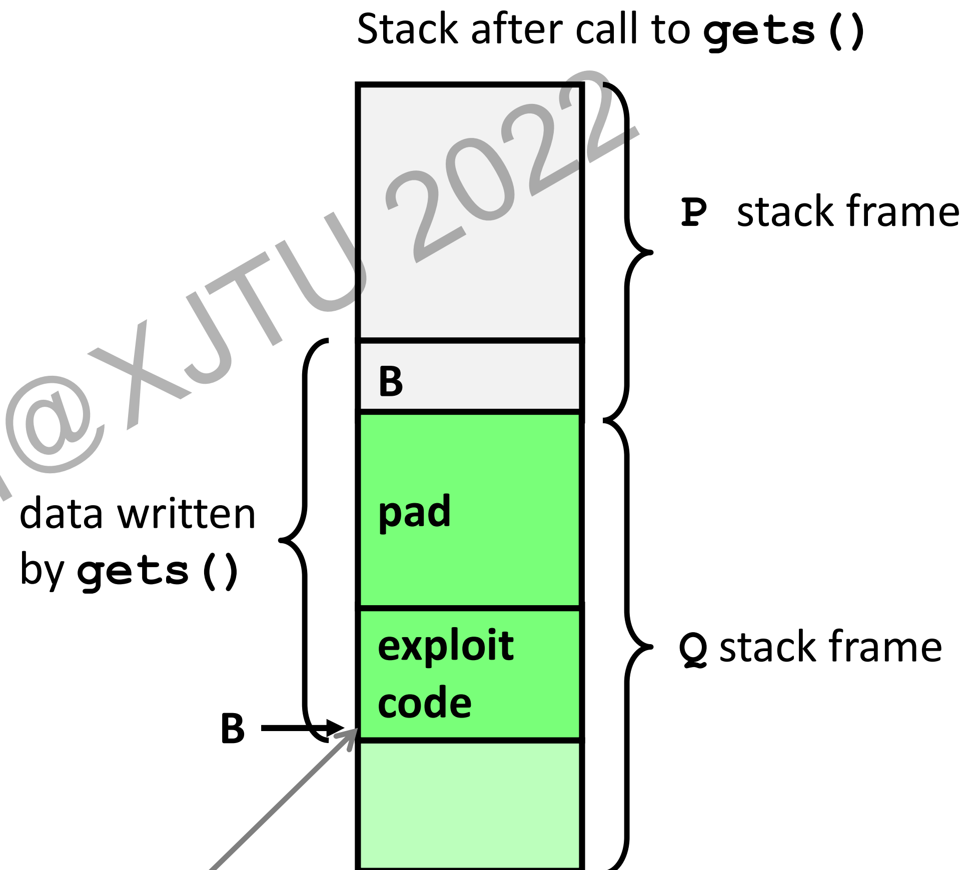
    - » Stack repositioned each time program executes



Pengju Ren@XJTU 2022

2. System-Level Protections can help

- Nonexecutable code segments
  - In traditional x86, can **mark region of memory as either "read-only" or "writeable"**
    - » Can execute anything readable
  - x86-64 added explicit "execute" permission
  - Stack marked as **non-executable**



Any attempt to execute this code will fail

Pengju Ren@XJTU 2022

3. Stack Canaries can help

- Idea
  - **Place special value ("canary")** on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - **`-fstack-protector`**
  - Now the default (disabled earlier)

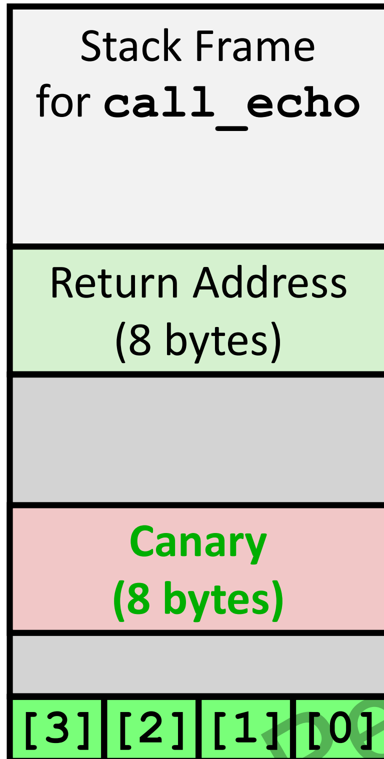
```
unix>./bufdemo-sp
Type a string:0123456
0123456
```

```
unix>./bufdemo-sp
Type a string:012345678
*** stack smashing detected ***
```

Pengju Ren@XJTU 2022

Setting Up Canary

Before call to gets



buf ⟵ %sp

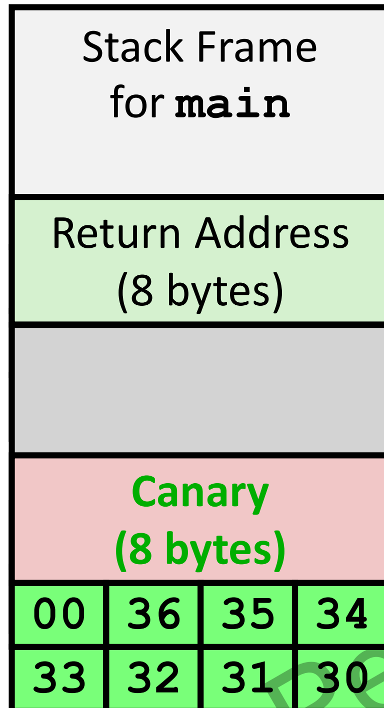
```
/* Echo Line */
void echo()
{
    char buf[4];   /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    mov     %fs:0x28, %rax   # Get canary
    mov     %rax, 0x8(%rsp) # Place on stack
    xor     %eax, %eax       # Erase register
    . . .
```

Pengju Ren@XJTU 2022

Checking Canary

After call to gets



buf ⟵ %rsp

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

Pengju Ren@XJTU 2022

Input: ***0123456***

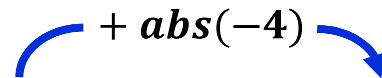
Some systems:
LSB of canary is 0x00
Allows input 01234567

```
echo:
    . . .
    mov     0x8(%rsp),%rax     # Retrieve from stack
    xor     %fs:0x28,%rax      # Compare to canary
    je      .L6                # If same, OK
    call    __stack_chk_fail   # FAIL
```

Appendix B: 3x3 Matrix Transpose

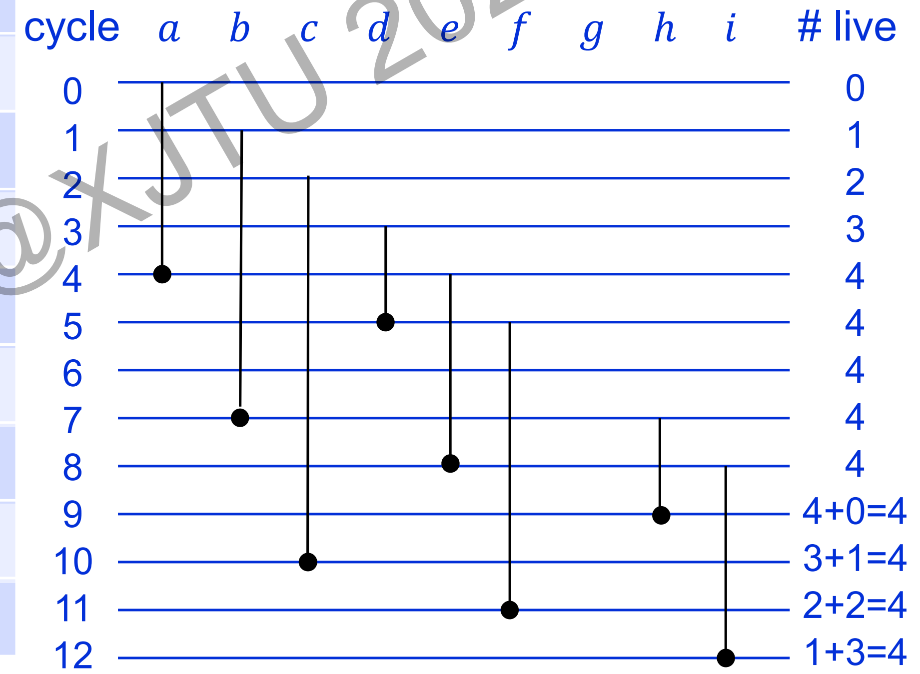
Pengju Ren@XJTU 2022

Example: Register Minimization Technique



Sample	T_{in}	T^*_{zlout}	T_{diff}	T_{out}	Life
a	0	0	0	4	$0 \to 4$
b	1	3	2	7	$1 \to 7$
c	2	6	4	10	$2 \to 10$
d	3	1	-2	5	$3 \to 5$
e	4	4	0	8	$4 \to 8$
f	5	7	2	11	$5 \to 11$
g	6	2	-4	6	$6 \to 6$
h	7	5	-2	9	$7 \to 9$
i	8	8	0	12	$8 \to 12$

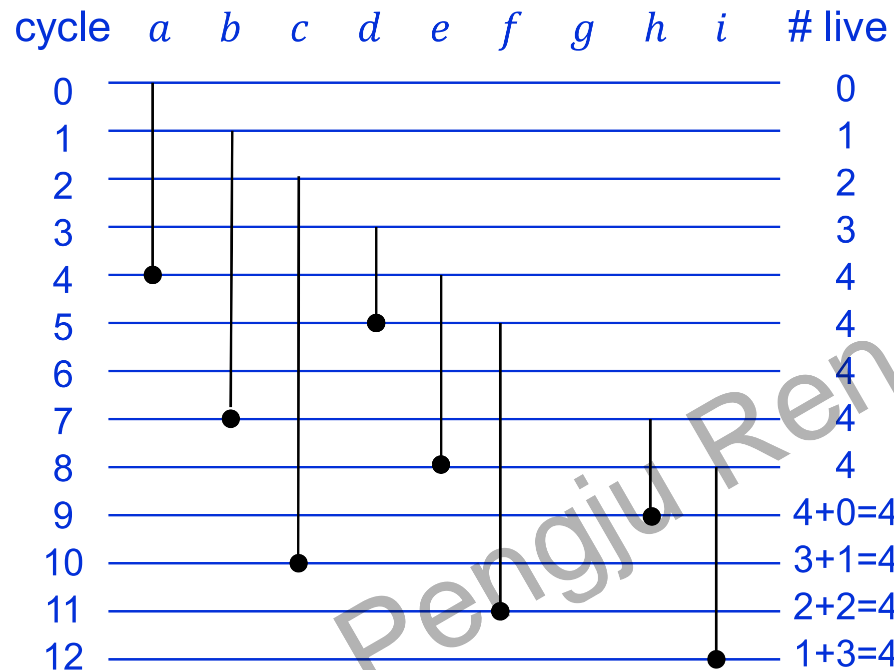
T_{zlout}: zero-latency output time



To make the system causal a latency of 4 is added to the difference so that T_{out} is the actual output time.

Pengju Ren@XJTU 2022

Example : Forward backward Register Allocation(1)



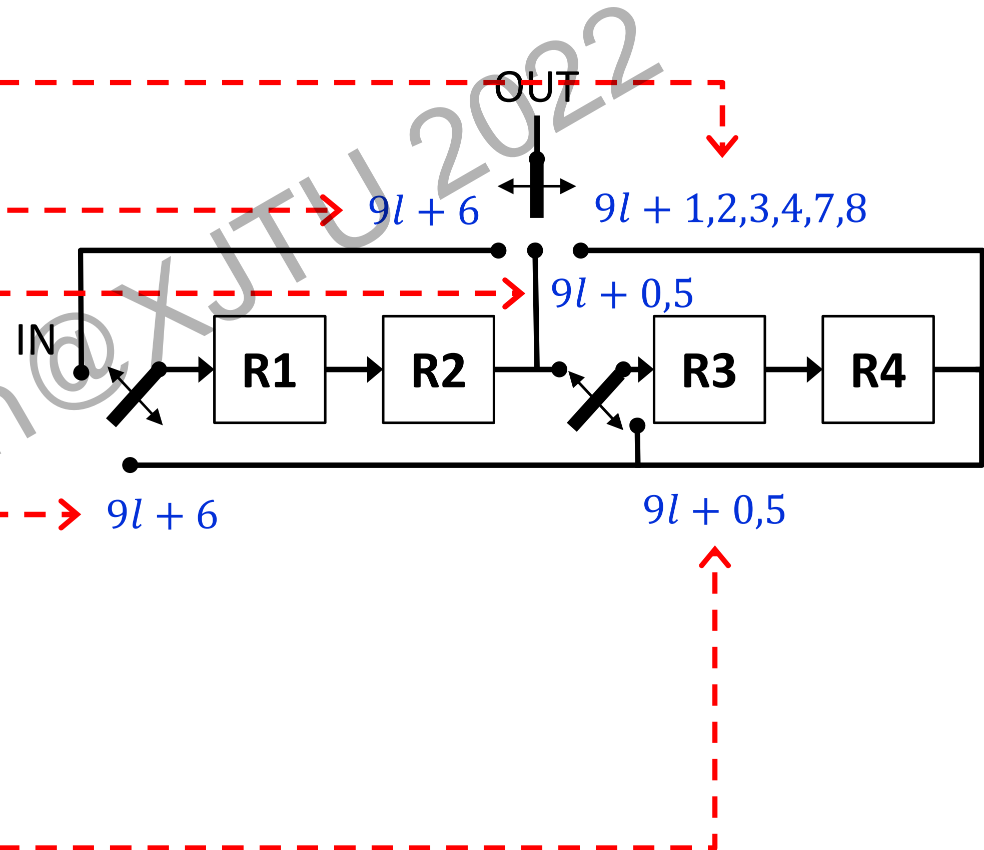
Cycle	Input	R1	R2	R3	R4	output
0	a					
1	b	a				
2	c	b	a			
3	d	c	b	a		
4	e	d	c	b	a	a
5	f	e	d	c	b	d
6	g	f	e	b	c	g
7	h	c	f	e	b	b
8	i	h	c	f	e	e
9		i	h	c	f	h
10			i	f	c	c
11				i	f	f
12					i	i

Pengju Ren@XJTU 2022

Note : Hashing is done to avoid conflict during backward allocation.

Example : Forward backward Register Allocation(2)

Cycle	Input	R1	R2	R3	R4	output
0	a					
1	b	a				
2	c	b	a			
3	d	c	b	a		
4	e	d	c	b	a	a
5	f	e	d	c	b	d
6	g	f	e	b	c	g
7	h	c	f	e	b	b
8	i	h	c	f	e	e
9		i	h	c	f	h
10			i	f	c	c
11				i	f	f
12					i	i



Pengju Ren@XJTU 2022

Home work (占期末成绩比重20%)

1. 实现4x4矩阵转置运算的ASIC电路设计（画出电路图）
2. 运行 nxm 矩阵转置运算（ASIC和CPU两种实现方式）在不同条件下，应该如何选择？（给出分析报告）

注意事项：

作业提交邮箱：zhiwanghuo@stu.xjtu.edu.cn

作业名称：学号-姓名-HW1 例如：霍志旺-4121155033-HW1

截至时间：2022-12-1 24：00前

Pengju Ren@XJTU 2022

Next Lecture :

A first Glance of Parallel Computing

(Key themes)

Pengju Ren@XJTU 2022