Embedded Intelligent System and Novel Computer Architecture

Lecture 03(b) –Modern Processor：Data-Level Parallel (DLP) and Thread-Level Parallel (TLP)

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

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

Contents and Goals

- **Four key concepts about how modern computers work**
  - **Three concern parallel execution (Data-level, Thread-level, Explicit ILP（VLIW))**
  - **Two concern challenges of accessing memory (Latency and Throughput)**
- **Understanding these architecture basics will help you**
  - **Understand and optimize the performance of your parallel programs**
  - **Gain intuition about what workloads might benefit from fast parallel machines**

Example Program

Compute sin(x) using Taylor Expansion: sin(x)=x-x^3/3!+x^5/5!-x^7/7!+…

For each element of an array of N floating-point numbers

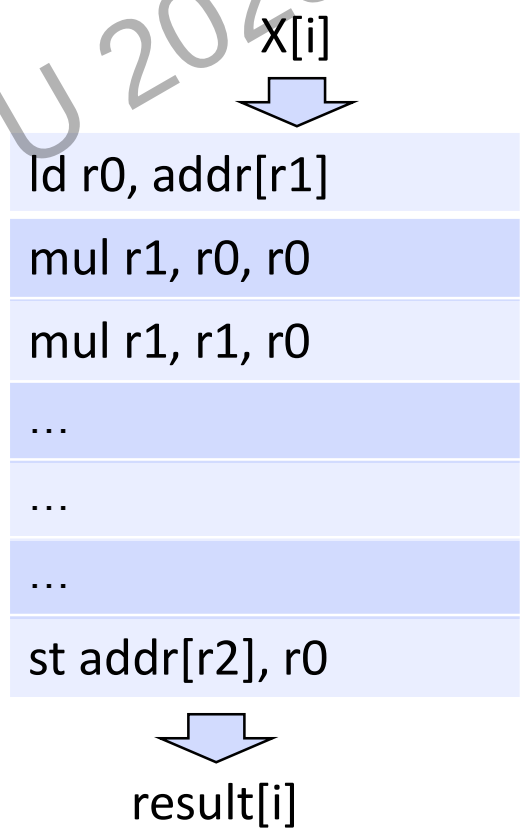
```
void sinx(int N, int terms, float * x,
                  float *result) {
    for (int i=0; i<N; i++) {
        float value = x[i];
        float numer = x[i]*x[i]*x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

Compile Program

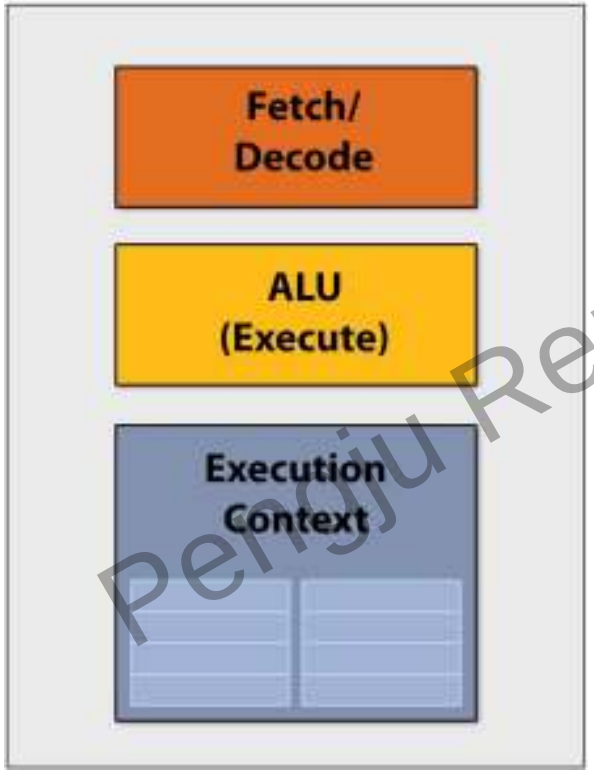
Compute sin(x) using Taylor Expansion: sin(x)=x-x^3/3!+x^5/5!-x^7/7!+…

For each element of an array of N floating-point numbers

```
void sinx(int N, int terms, float * x,
                  float *result) {
    for (int i=0; i<N; i++) {
        float value = x[i];
        float numer = x[i]*x[i]*x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```



Execute Program



X[i]

ld r0, addr[r1]

mul r1, r0, r0

mul r1, r1, r0

…

…

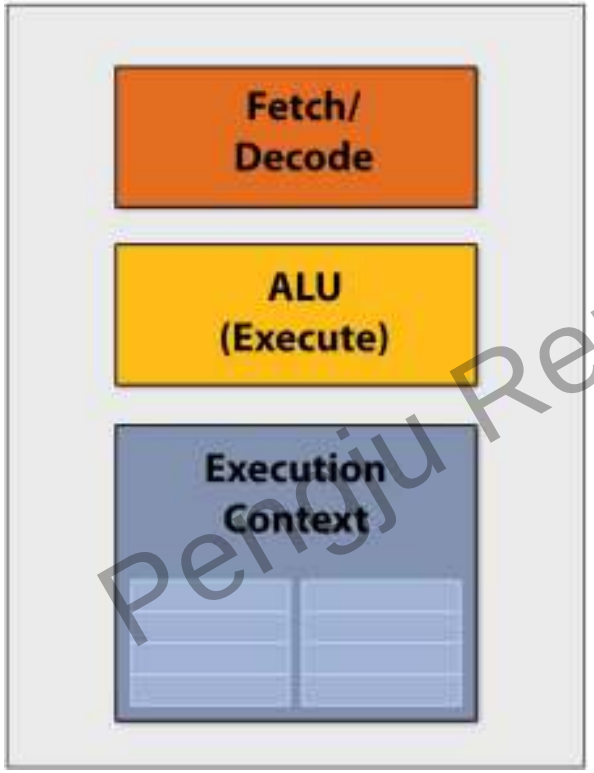
…

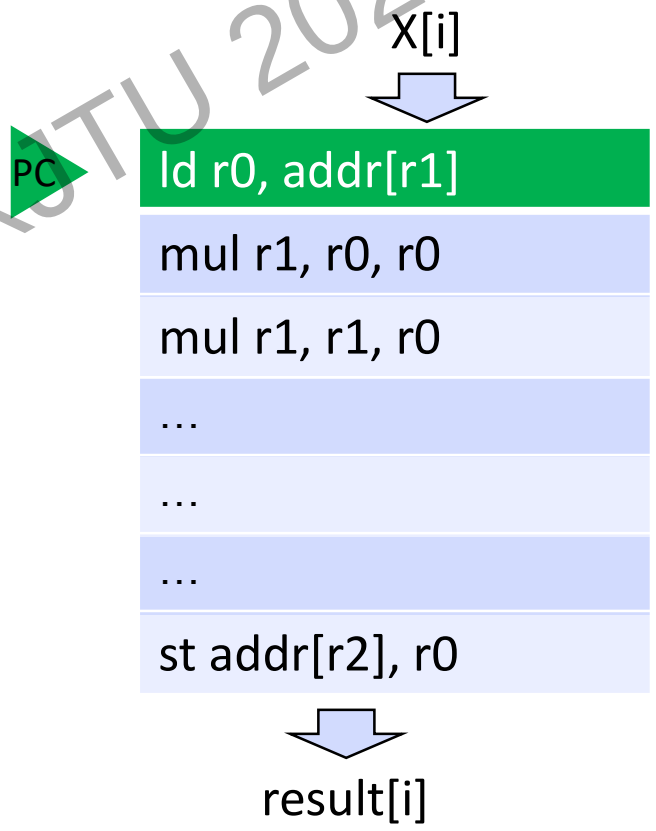
st addr[r2], r0

result[i]

Execute Program

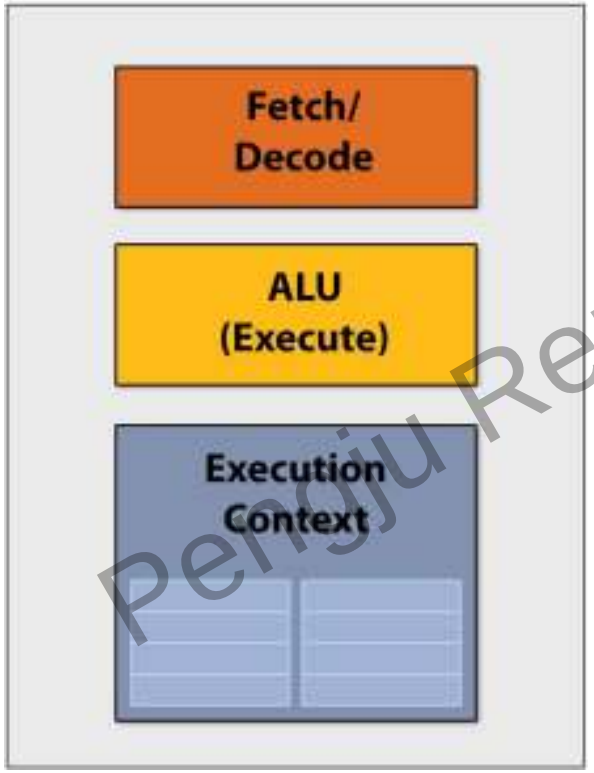
A simple processor: executes one instruction per clock

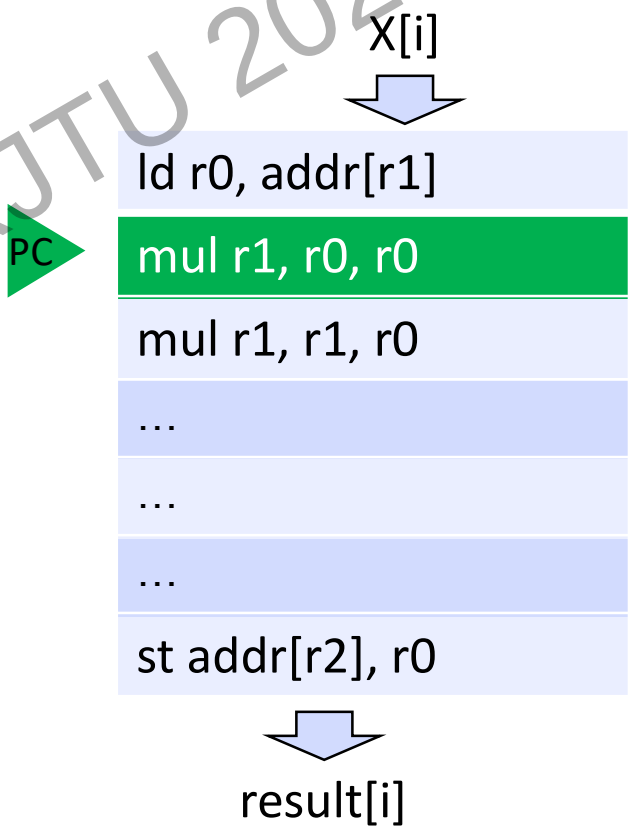




Execute Program

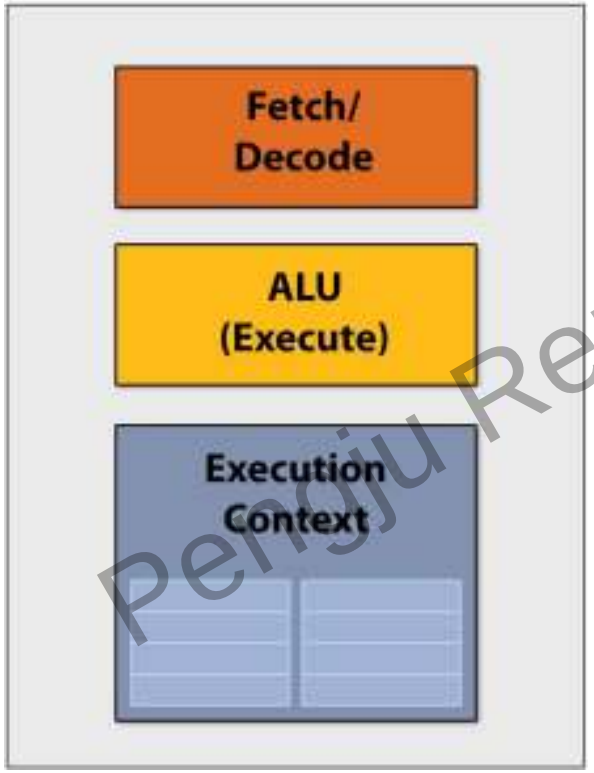
A simple processor: executes one instruction per clock

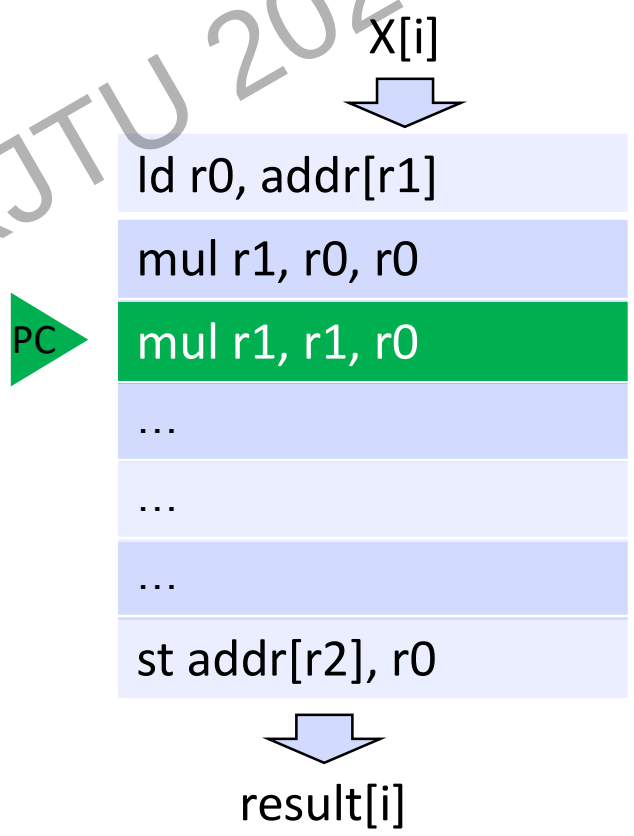




Execute Program

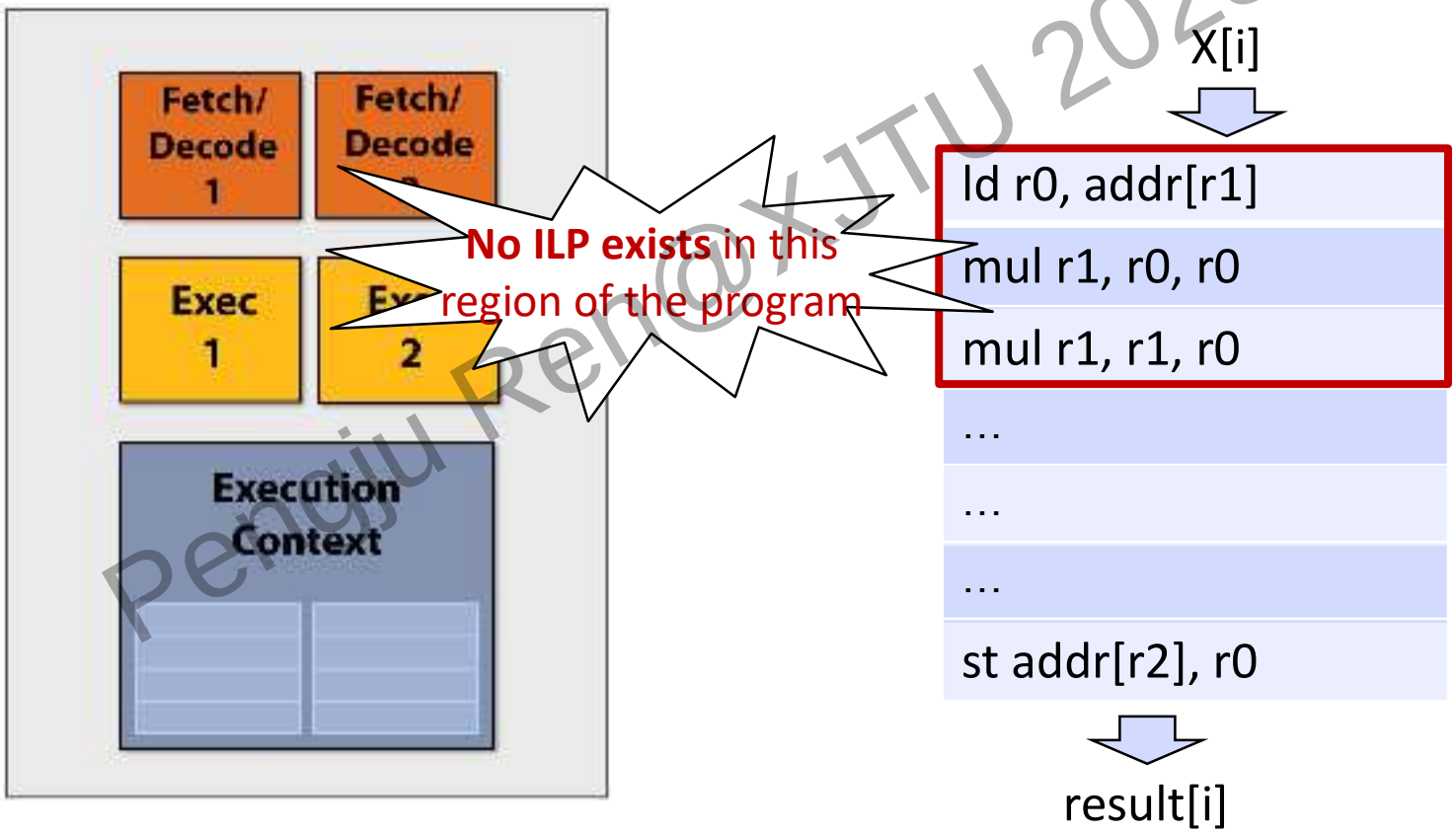
A simple processor: executes one instruction per clock





Superscalar (in order) Processor

Recall from last class: instruction level Parallelism (ILP)
Decode and execute two instructions per clock (if possible)



Uniprocessors in the Real World

- Real processors have
  - pipelining
    - a form of parallelism, like an assembly line in a factory (Data, Control, Structure hazards)
  - Superscalar and OoO
    - multiple "functional units" that can run in parallel （Multi-way & multi-issue）
    - different orders, instruction mixes have different costs （OoO）
  - registers and caches
    - small amounts of fast memory
    - store values of recently used or nearby data
    - different memory ops can have very different costs

Recap: Pipeline and Superscalar

- Higher levels of device integration have made available a large number of transistors. Current processors use these resources in multiple functional units and execute multiple instructions in the same cycle.
- Pipelining overlaps various stages of instruction execution to achieve performance, however, it has several limitations.
  - The speed of a pipeline is eventually limited by the slowest stage.
  - In typical program traces, every 5-6th instruction is a conditional jump! This requires very accurate branch prediction. The penalty of a misprediction grows with the depth of the pipeline, since a larger number of instructions will have to be flushed.

Recap: Superscalar Execution

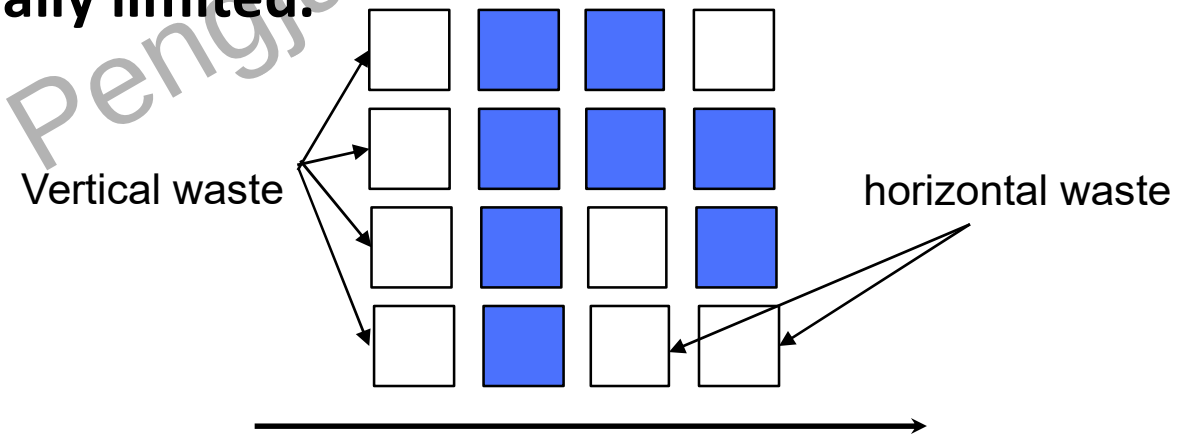
- Scheduling of instructions is determined by a number of factors:
  - True Data Dependency: The result of one operation is an input to the next. (Read after Write, RAW)
  - Resource Dependency: Two operations require the same resource.
  - Branch Dependency: Scheduling instructions across conditional branch statements cannot be done deterministically a-priori.
  - The scheduler, a piece of hardware looks at a large number of instructions in an instruction queue and selects appropriate number of instructions to execute concurrently based on these factors.
  - The complexity of this hardware is an important constraint on superscalar processors.

Recap: Superscalar (In-order and OoO)

- In the simpler model, instructions can be issued only in the order in which they are encountered. That is, if the second instruction cannot be issued because it has a data dependency with the first, only one instruction is issued in the cycle. This is called *in-order* issue.
- In a more aggressive model, instructions can be issued out of order. In this case, if the second instruction has data dependencies with the first, but the third instruction does not, the first and third instructions can be co-scheduled. This is also called dynamic issue.
- Performance of in-order issue is generally limited. However, OoO is expensive, with the cost of: *Issue Queue, Scoreboard, Register Renaming, Reorder Buffer, Store Buffer, etc.*

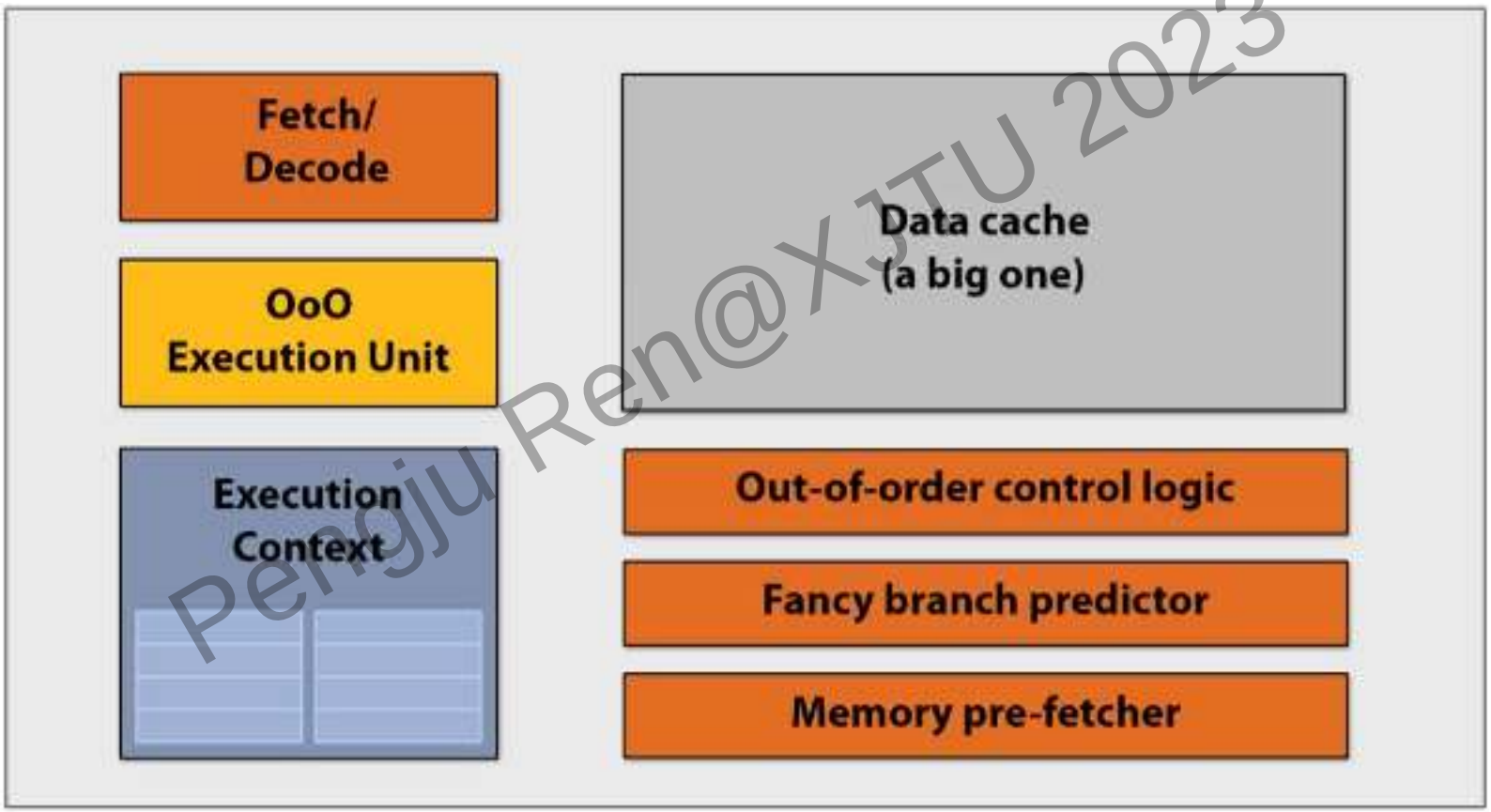
Superscalar Execution: Efficiency Considerations

- Not all functional units can be kept busy at all times.
- If during a cycle, no functional units are utilized, this is referred to as vertical waste.
- If during a cycle, only some of the functional units are utilized, this is referred to as horizontal waste.
- Due to limited parallelism in typical instruction traces, dependencies, or the inability of the scheduler to extract parallelism, the performance of superscalar processors is eventually limited.



Processor: pre Multi-core era

Majority of chip transistors (area and energy) used to perform operations that help a single instruction stream run fast（instruction level parallel, ILP)



More transistors = larger cache, smarter out-of-order logic, smarter branch predictor, etc. (Also: more transistors → smaller transistors → higher frequencies)

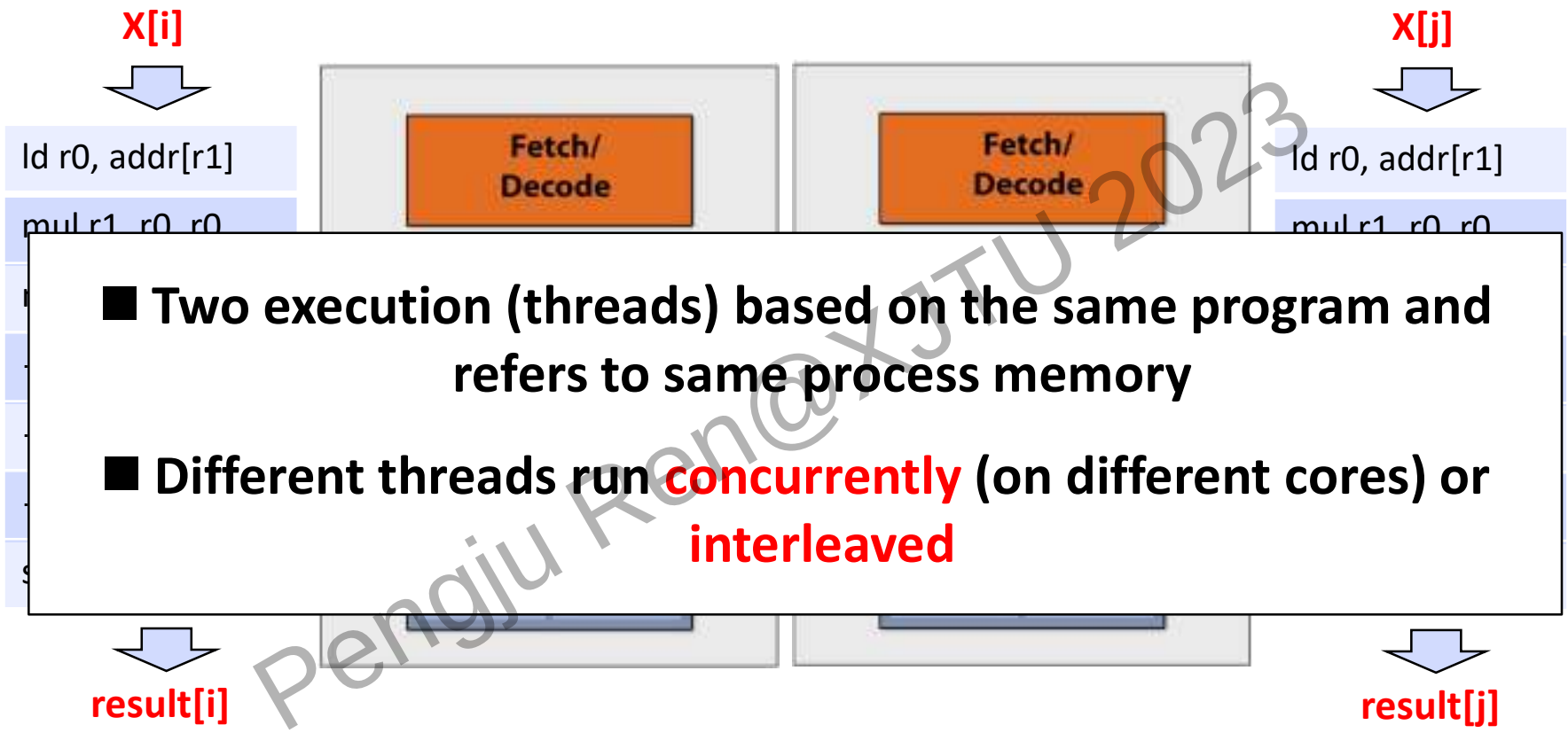
Processor: Multi-core era

Idea #1:

Use increasing transistor count to add more cores to the processor

Rather than use transistors to increase sophistication of processor logic that accelerates a single instruction stream (e.g., out-of-order and speculative operations)

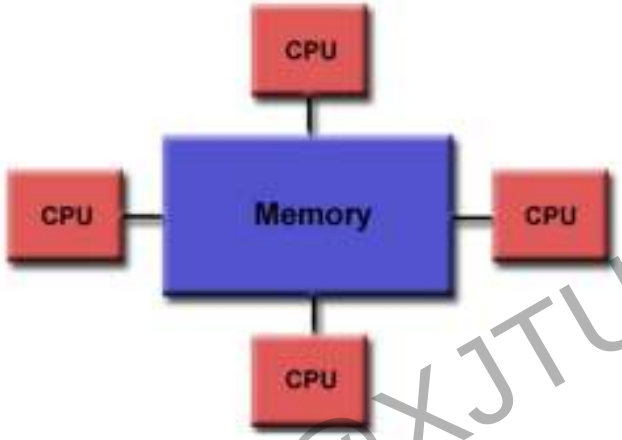
Two cores: compute two elements in parallel



- Two execution (threads) based on the same program and refers to same process memory
- Different threads run concurrently (on different cores) or interleaved

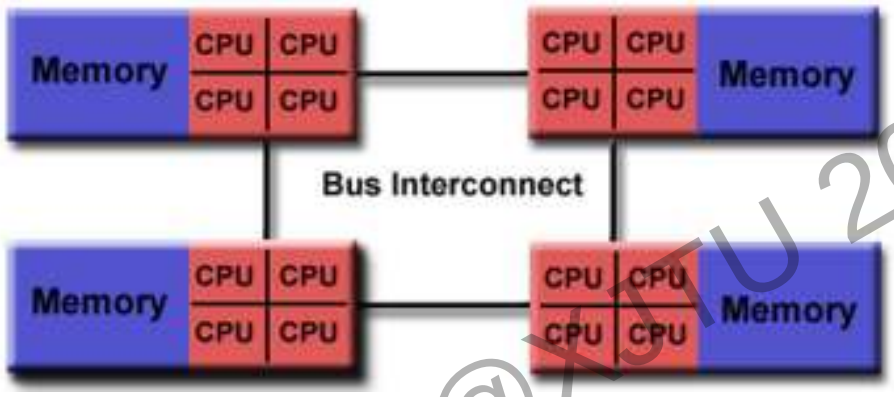
Simpler cores: each core is slower at running a single instruction stream than our original “fancy” core (e.g., 25% slower)
But there are now two cores（physically or Logically）: 2 × 0.75 = 1.5
(potential for speedup!)

MIMD Shared Memory：Symmetric Multiprocessors（SMPs）



- Symmetric：
  - identical processors connected to common memory
  - all processors have equal access to system (mem & I/O)
  - OS can schedule any process on any processor
- Uniform Memory Access (UMA)
  - processor/memory connected by bus or crossbar
  - all processors have equal memory access performance to all memory locations
  - caches need to stay coherent

MIMD Shared Memory：Symmetric Multiprocessors（SMPs）

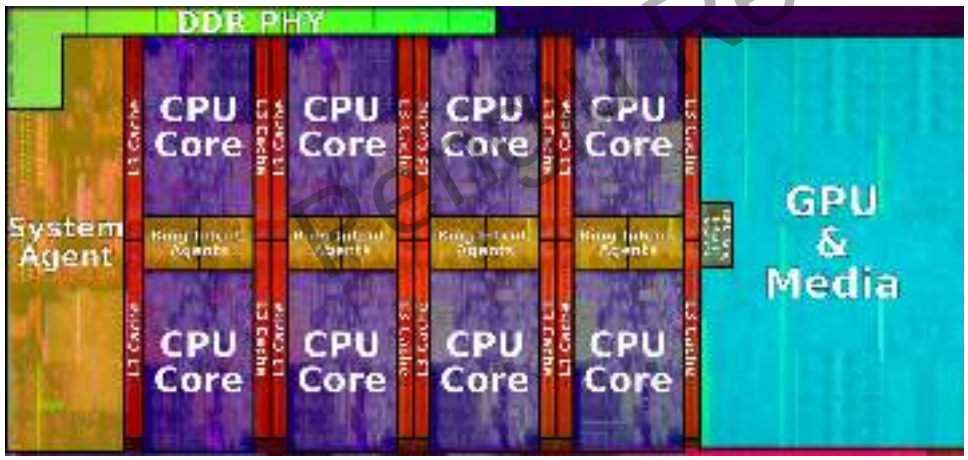


- UMA hard to scale due to concentration of BW
- Large scale SMPs have distributed memory with non-uniform memory (NUMA)
  - – “local” memory pages (faster to access)
  - – “remote” memory pages (slower to access)
  - – cache-coherence still possible but complicated

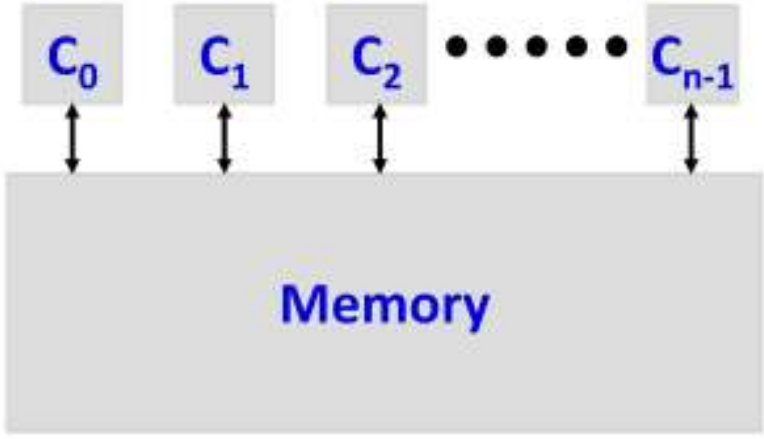
Shared-Memory Multicores

Today's general-purpose multicore processors are MIMD, symmetric, shared memory

- **individual cores follow classic von Neuman**
- **common access to physical address space and mem**
- **processes/threads on different cores communicate by writing and reading agreed-upon mem locations**



Intel i7 Gen8th/9th



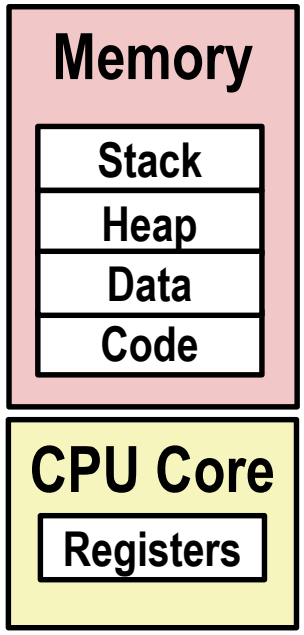
Terminology (Program & Process)

- **Program**: An executable task
  - Example: Calculating the *sum of 1,000,000* number
- **Process**: An instance of a running program or portion of program
  - Example a running instance of *sum of 1,000,000*

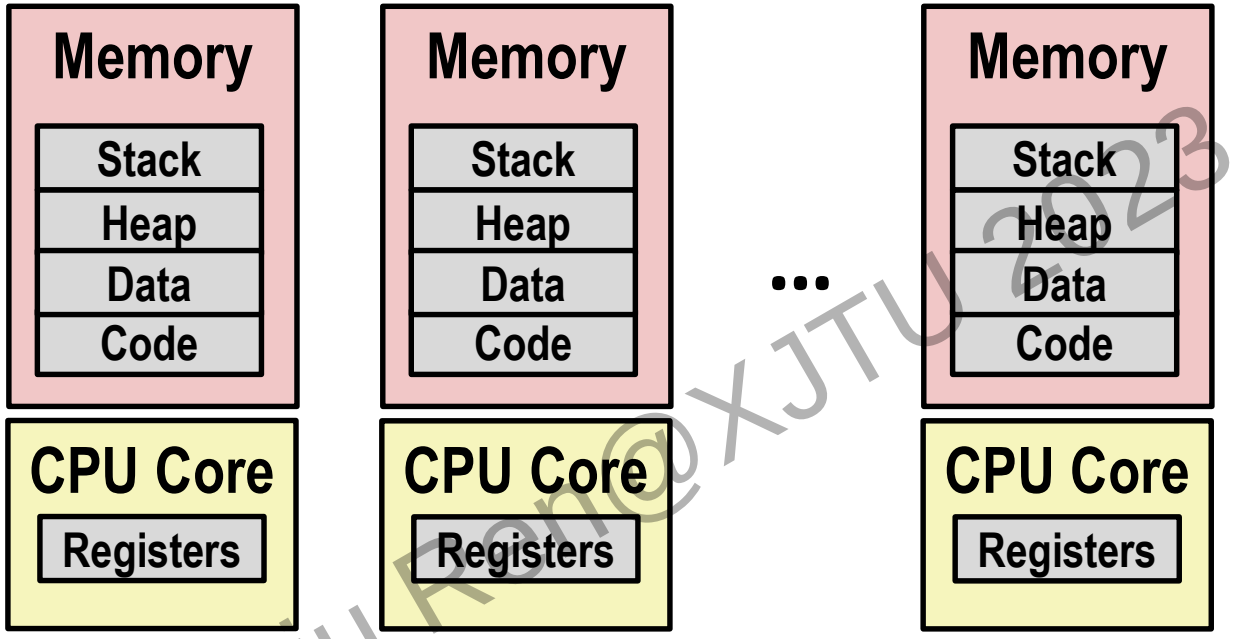
- Process provides each program with two key abstractions:

  Logical control flow： Each program seems to have exclusive use of the CPU

  Private address space(Virtual Mem)： Each program seems to have exclusive use of main memory.

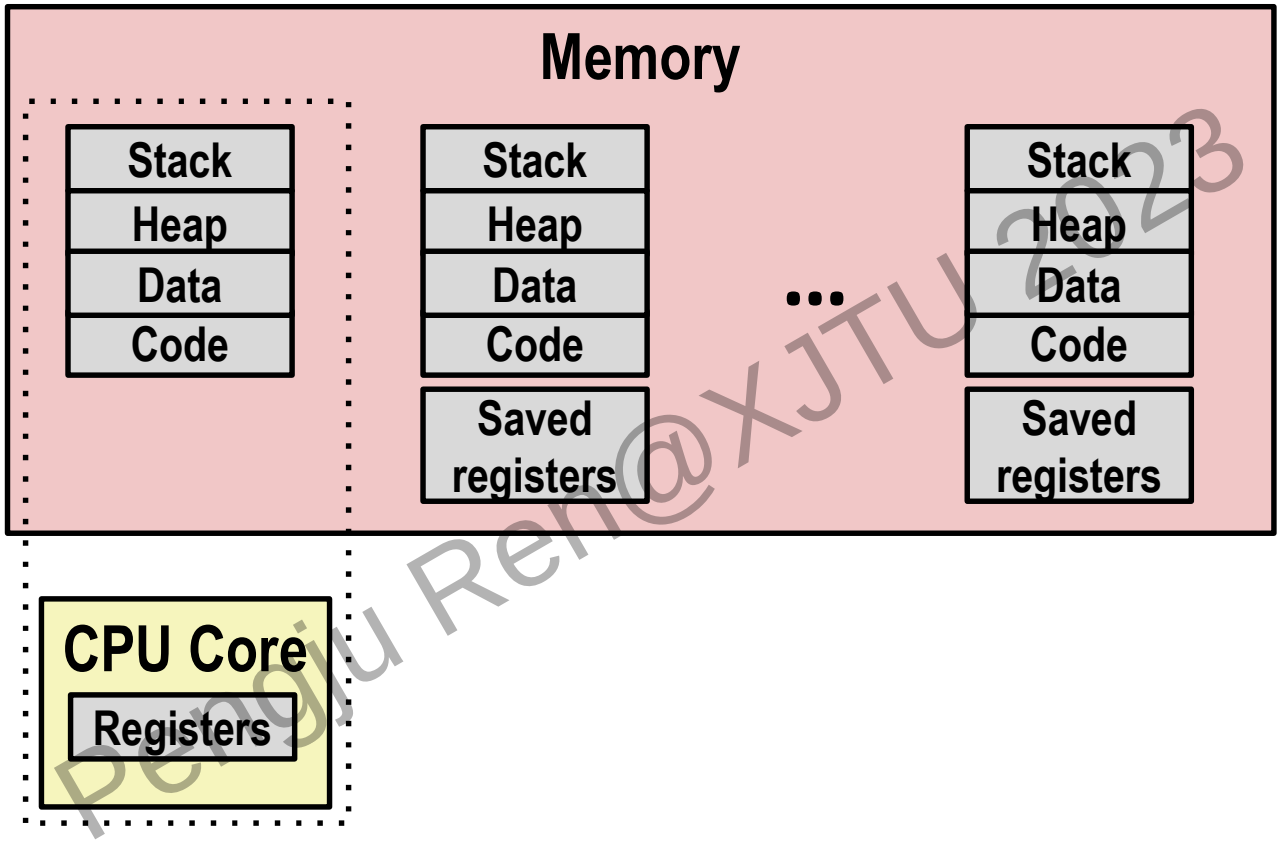


Processes Execution



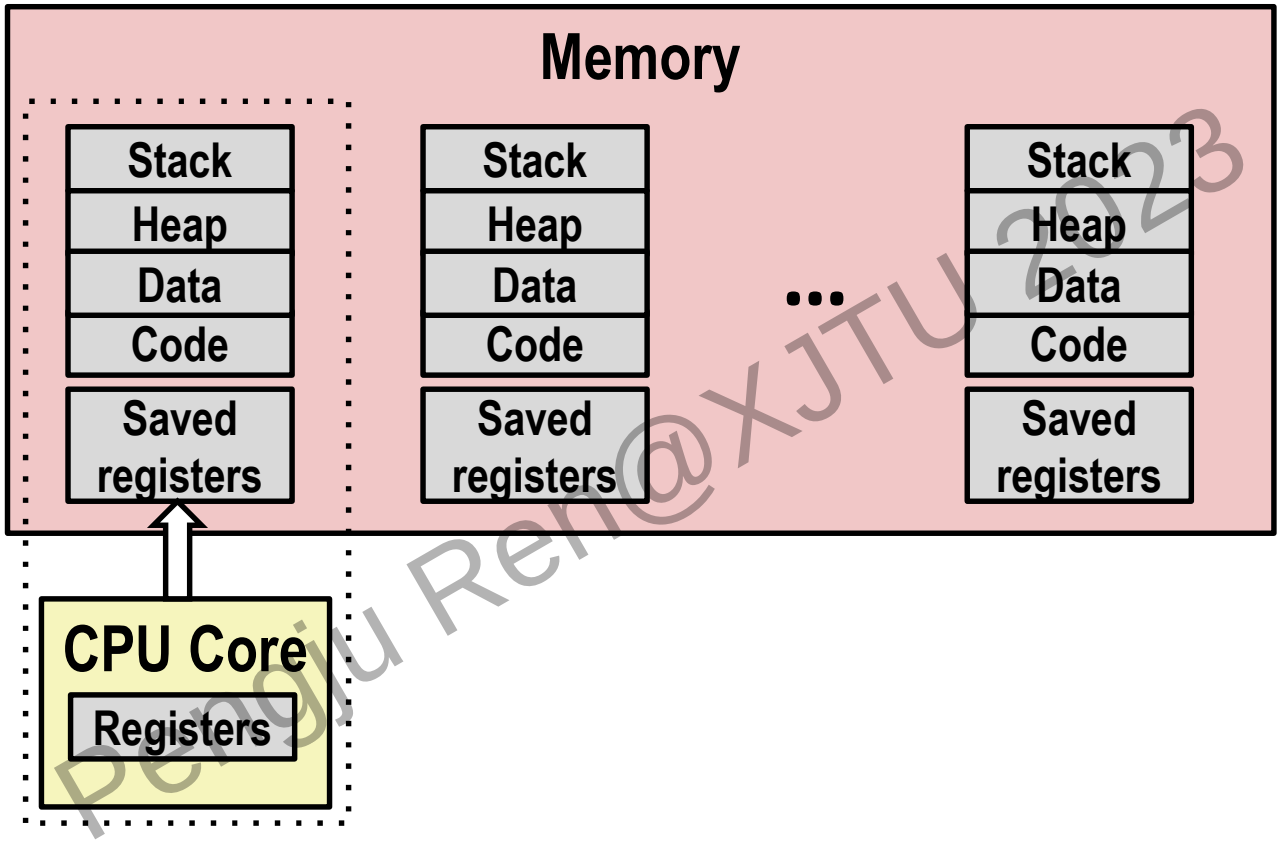
- **Processor runs many processes simultaneously**
  - **Applications for one or more users**
    - **Web browsers, email clients, editors, …**
  - **Background tasks**
    - **Monitoring network & I/O devices**

Process on Single core Processor



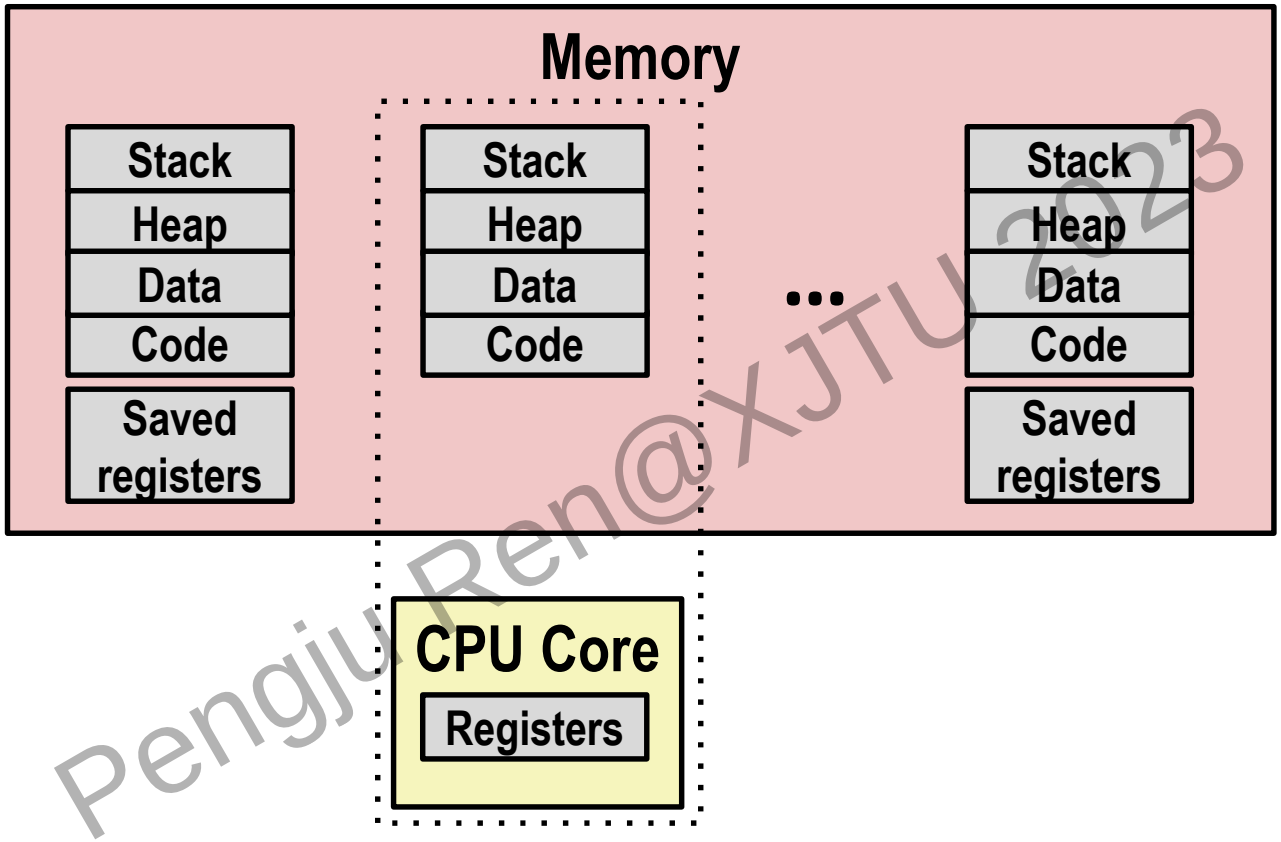
- Single core executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system
  - Register values for nonexecuting processes saved in memory

Process on Single core Processor



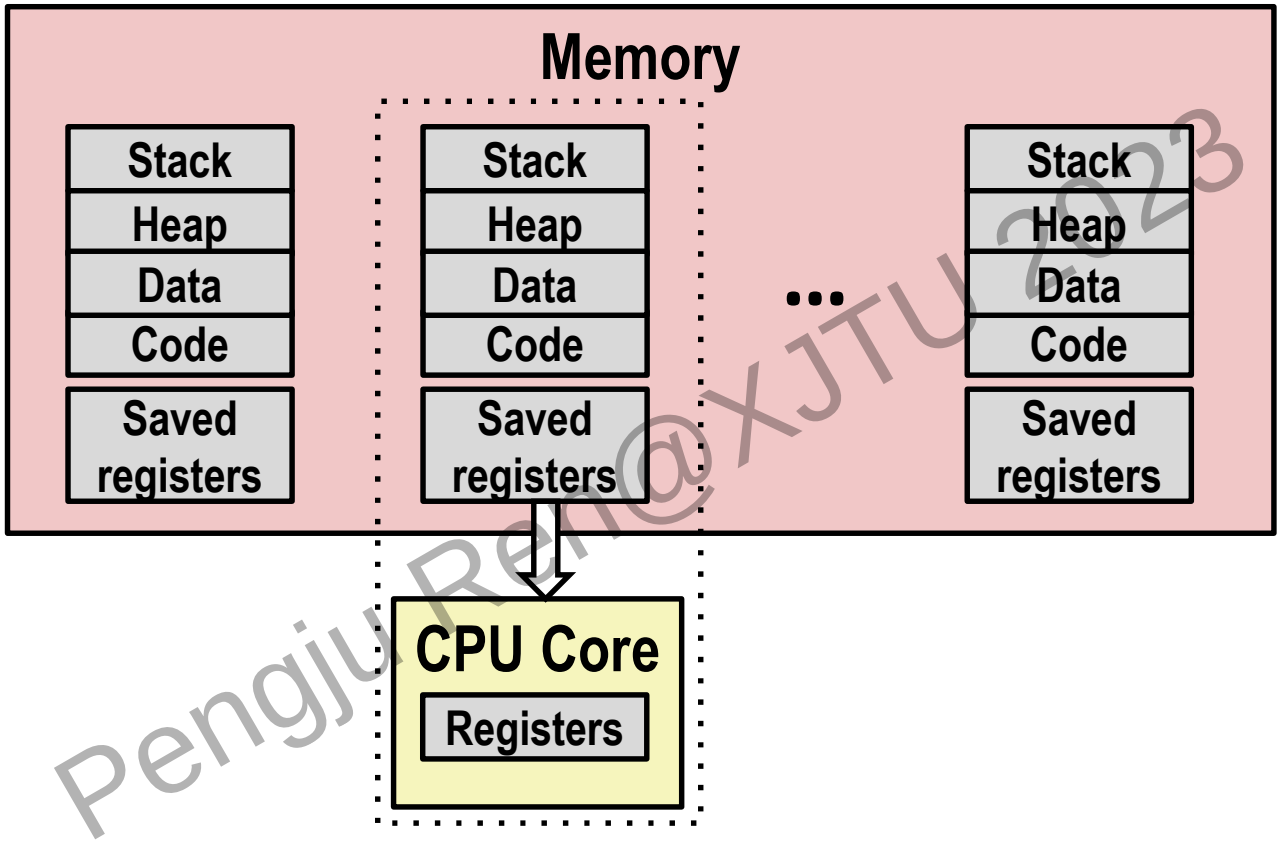
- Save current registers in memory

Process on Single core Processor



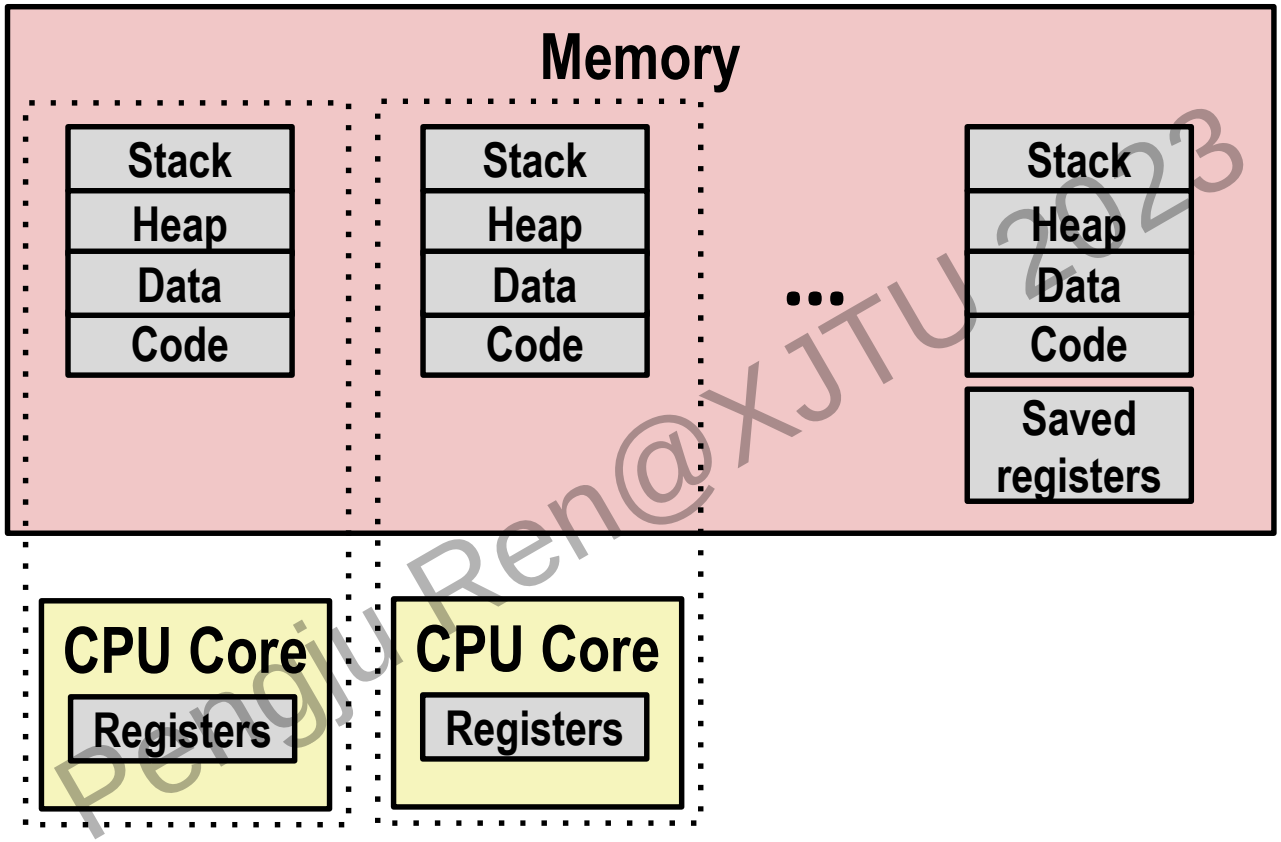
- Schedule next process for execution

Process on Single core Processor



- Load saved registers and switch address space (*context switch*)

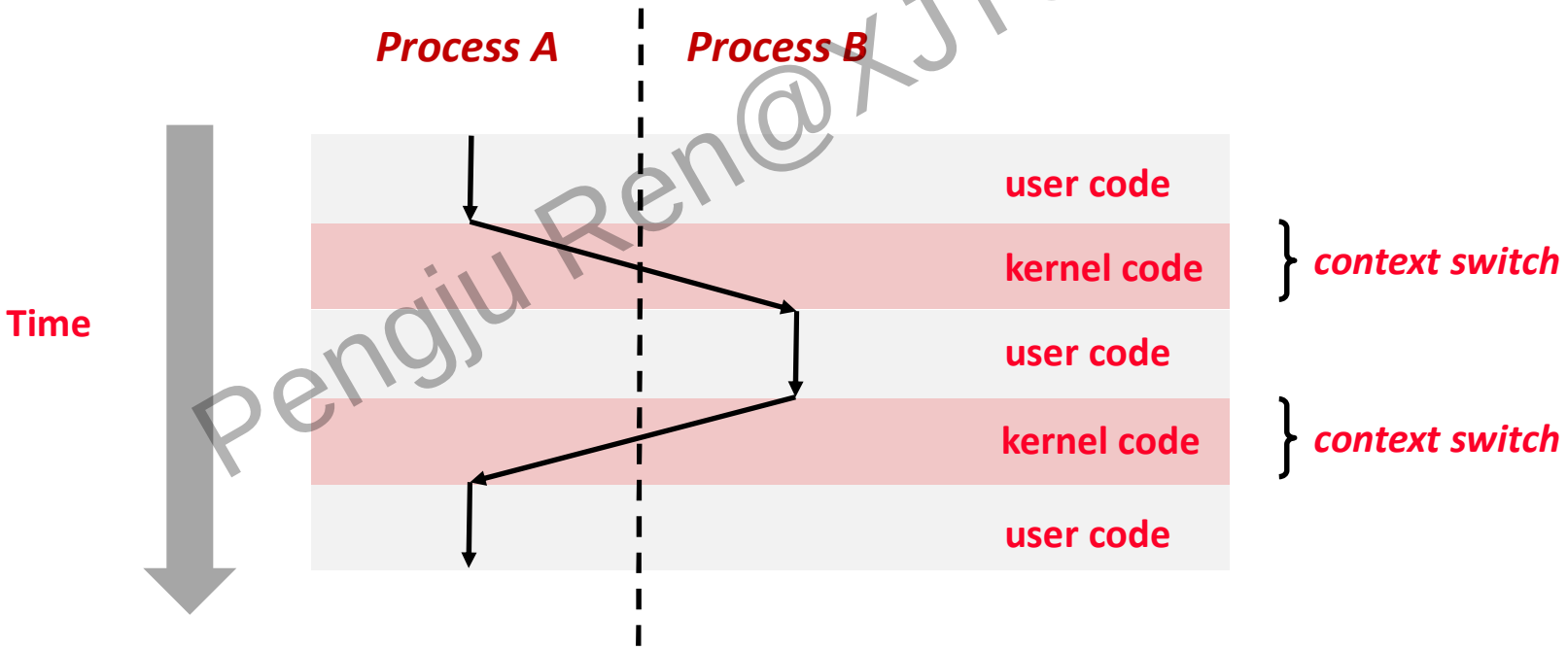
Process on Multicore Processor



- **Multicore processors**
  - Multiple CPUs on single chip
  - Share main memory (and some lower level caches)
  - Each can execute a separate process
    - Scheduling of process onto CPU cores is done by OS

Context Switching

Control flow passes from one process to another via a *context switch,* that conducted by *Operating System (OS)*



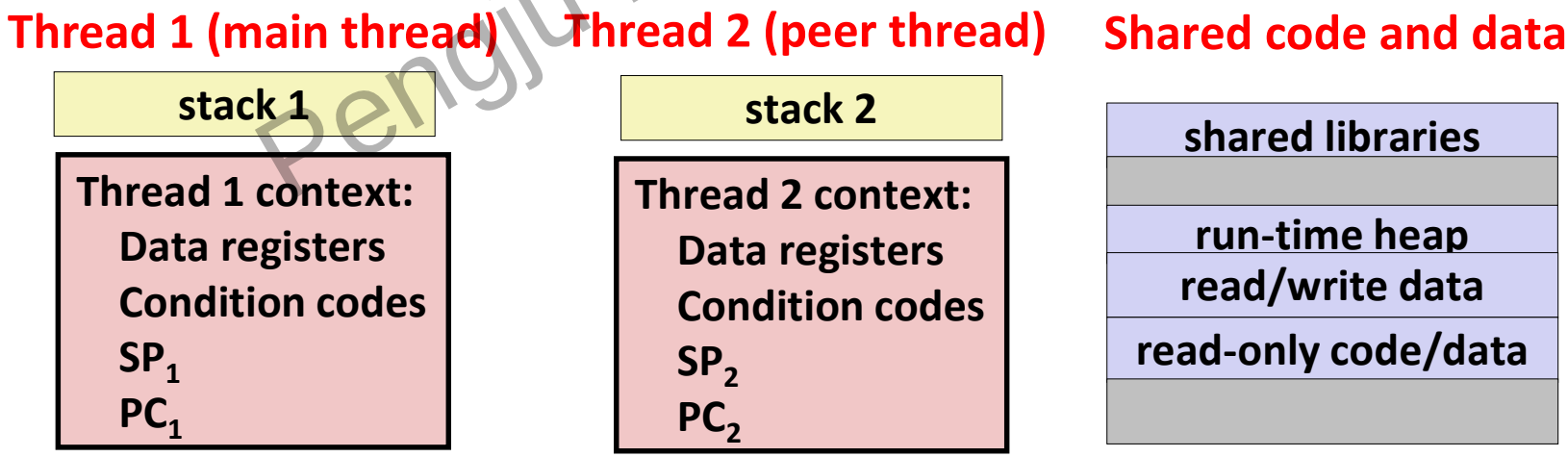
Recap: Terminology

- **Program: An executable task**
  - **Example: Calculating the *sum of 1,000,000* number, could partition a single problem into multiple related tasks (threads) through *parallel programming***
- **Process: An instance of a running program or portion of program**
  - **Example a running instance of *sum of 1,000,000***
- **Software Thread (Programmer view point): it is an abstraction to the hardware to make multi-processing possible, the *smallest unit of processing* assigned and scheduled by OS**
  - **Example: using 100 threads to conduct the *sum of 1,000,000***
- **Hardware Thread (Architecture view point) : Can be thought of as the physical/logical CPU or cores. hardware thread can run many software threads by time-slicing by the OS.**
  - **Example: Your Laptop i7 CPU with 4 core/8 thread; Lab Server Xeon CPU with 24core/48 thread**

Thread models : two very different implementations ***POSIX Threads*** and ***OpenMP***.

A Process With Multiple Threads

- **Multiple threads can be associated with a process, each thread has:**
  - ***Dedicated PC (program counter)***
  - ***Separate registers***
  - ***Private variables (local stack variables)***
  - ***Accesses the shared memory (static variables, global heap)***
- **Threads communicate and coordinate** implicitly by **writing/reading shared variables (will coming soon!)**



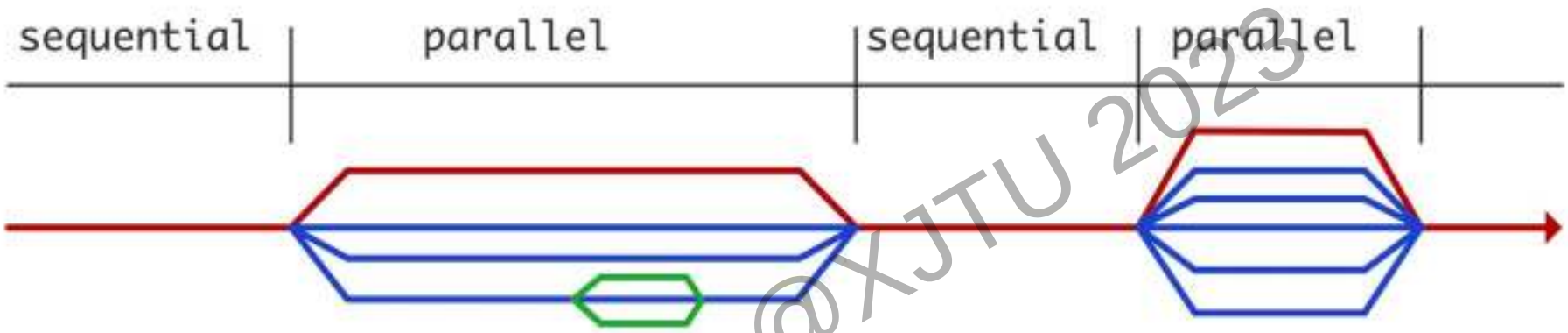
Threads vs. Processes

- How threads and processes are similar
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched
- How threads and processes are different
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread

Thread-Level Parallelism（TLP）

- Many workloads can make use of thread-level parallelism (TLP)
  - TLP from multiprogramming *(run independent sequential jobs)*
  - TLP from multithreaded applications *(run one job faster using parallel threads)*
- Multithreading: uses TLP to improve utilization of a single processor
- Multi-/Many-core: Duplicate Processors, it plays a major role from the low end to the high end
- Modern CPU do both
  - Multiple or tens of cores with multiples threads per core

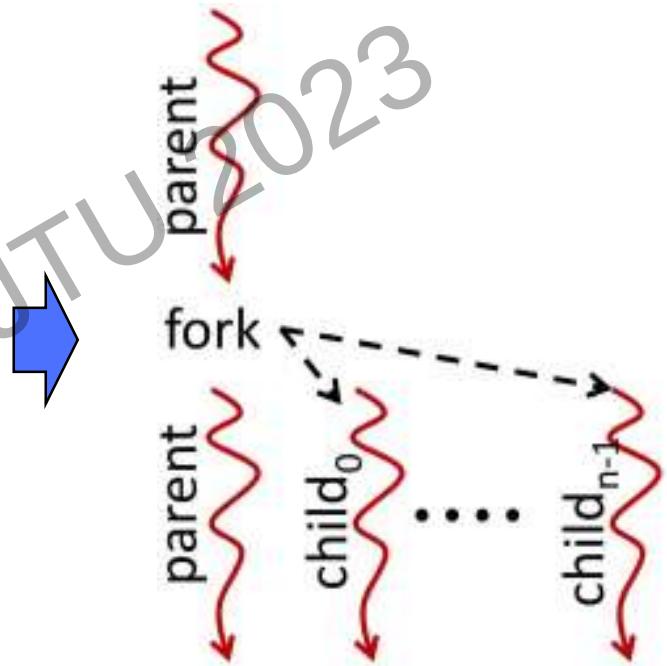
Expressing parallelism using pthreads



Threads are dynamic, that they can be created during program execution. When a program starts, there is one thread active: the *master thread*. Other threads are created by *thread spawning*, and the master thread can wait for their completion. This is know as the *fork-join model*.

Expressing parallelism using pthreads

```
void sinx(int N, int terms, float * x,
                    float *result) {
    for (int i=0; i<N; i++) {
        float value = x[i];
        float numer = x[i]*x[i]*x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```



This program, compiled with gcc will run as one thread on one of the processor cores. Requires explicit programming (e.g. POSIX thread and OpenMP)

https://hpc-turotials.llnl.gov/posix/

Expressing parallelism using Pthreads

```
typedef struct {
    int N; int terms; float* x; float* result;
                    } my_args;

void parallel_sinx(int N, int terms, float* x, float* result)
     { pthread_t thread_id;
       my_args args;

       args.N = N/2;
       args.terms = terms;
       args.x = x;
       args.result = result;

      pthread_create(&thread_id, NULL, my_thread_start,
&args); // launch thread
      sinx(N - args.N, terms, x + args.N, result + args.N); //
do work
       pthread_join(thread_id, NULL); }

void my_thread_start(void* thread_arg)
   {  my_args* thread_args = (my_args*) thread_arg;
      sinx(args->N, args->terms, args->x, args->result); //
do work
    }
```

```
void sinx(int N, int terms, float * x,
                                float *result) {
    for (int i=0; i<N; i++) {
        float value = x[i];
        float numer = x[i]*x[i]*x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```


https://hpc-turotials.llnl.gov/posix/

Pthread routines (1)

pthread_create (thread_id, attr, start_routine, arg)

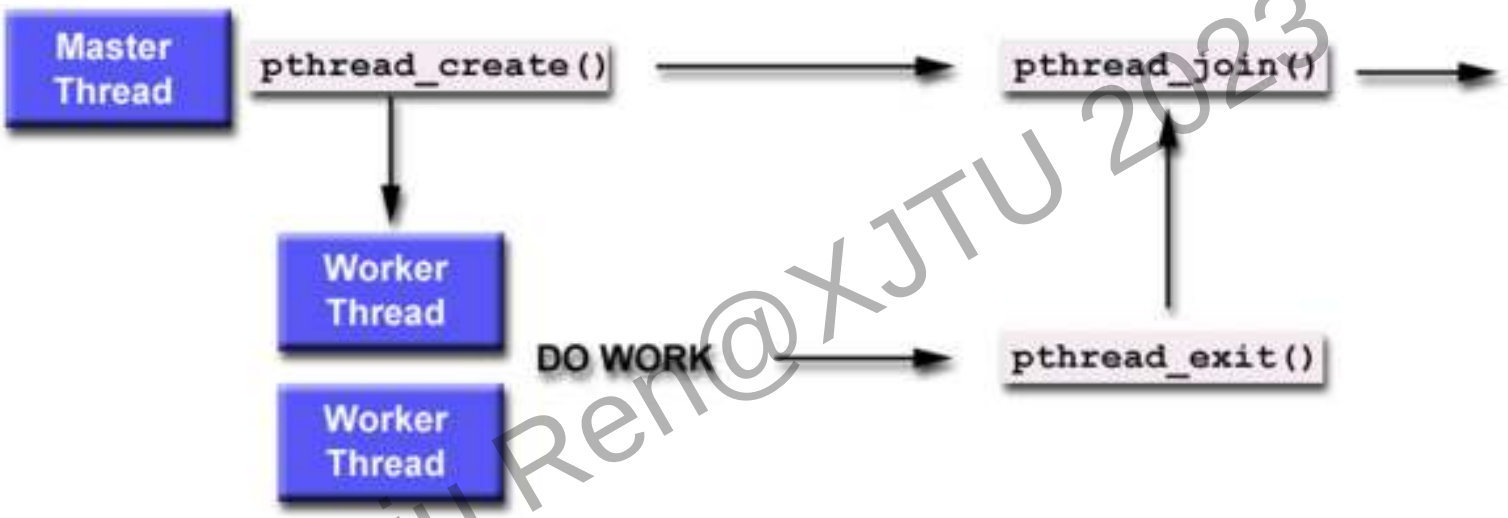
pthread_create **creates a new thread and makes it executable**

- ***thread_id*****: An opaque, unique identifier for the new thread returned by the subroutine.**
- ***attr:*** **An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.**
- ***start_routine:*** **the C routine that the thread will execute once it is created.**
- ***arg:*** **A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.**

The **pthread_create()** routine permits the programmer to pass one argument to the thread start routine. For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the **pthread_create()** routine.

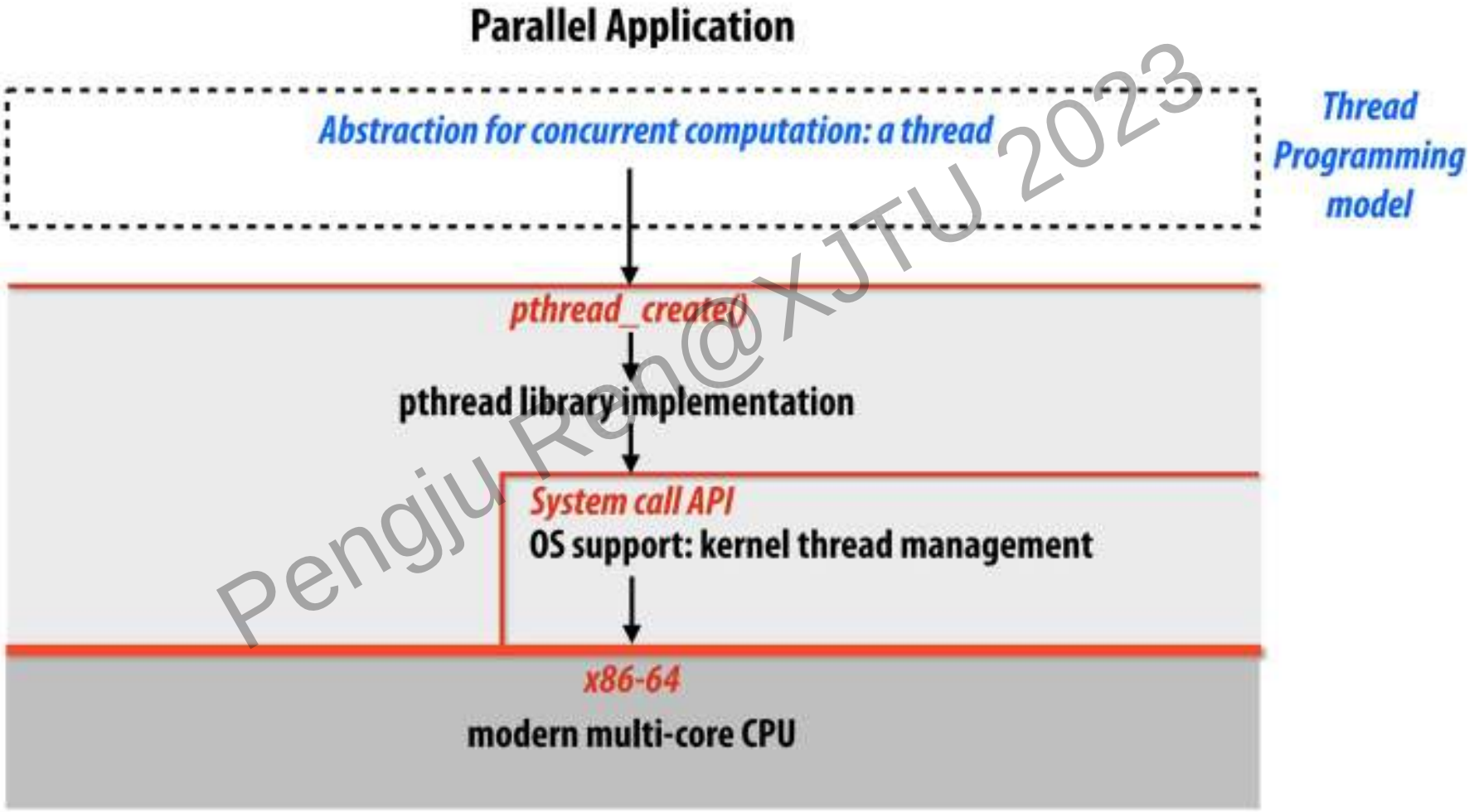
Pthread routines (2)

pthread_join (threadid,status)



- "Joining" is one way to accomplish synchronization between threads
- The pthread_join() subroutine blocks the calling thread until the specified threadid thread terminates
- A joining thread can match one pthread_join() call. It is a logical error to attempt multiple joins on the same thread.

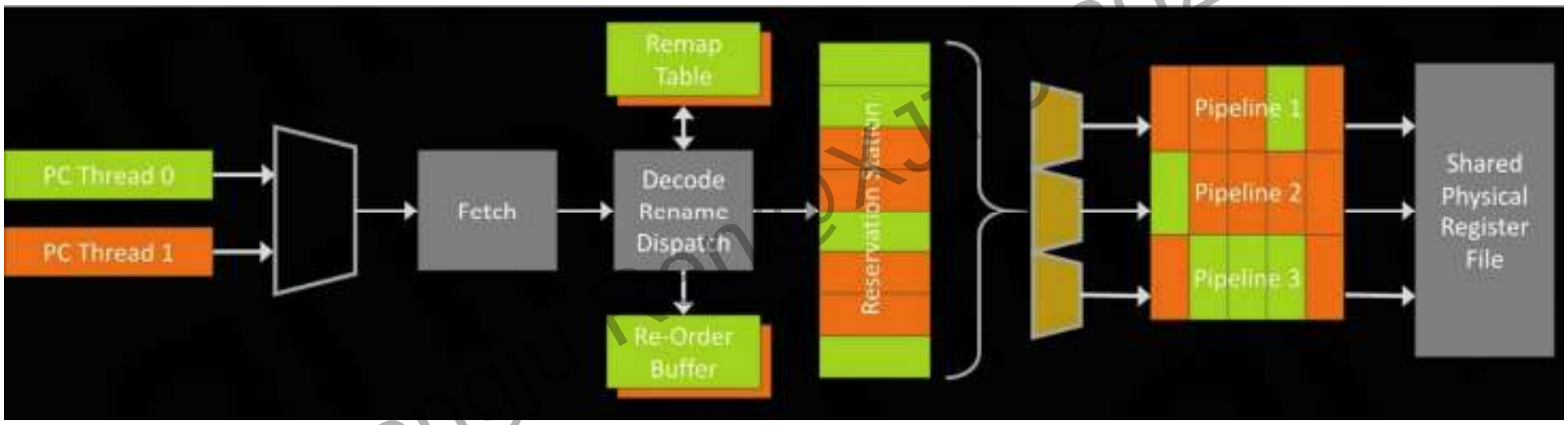
Expressing Parallelism with Pthreads



Hardware-supported multi-threading

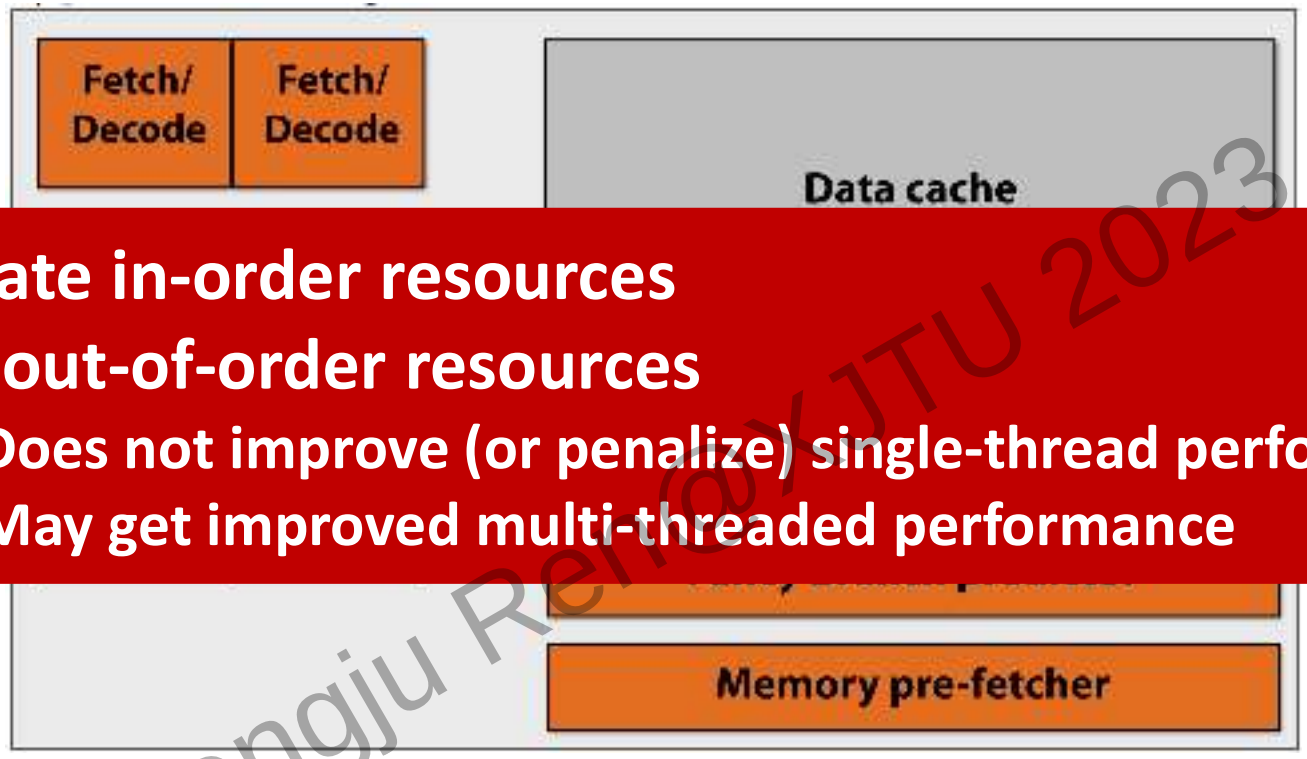
- Core manages execution contexts for multiple threads
  - Runs instructions from runnable threads (processor makes decision about which thread to run each clock, not the operating system)
  - Core still has the same number of ALU resources: multi-threading only helps use them more efficiently in the face of high-latency operations like memory access
- Interleaved multi-threading (a.k.a. temporal multi-threading)
  - As described on the previous slides: each clock, the core chooses a thread, and runs an instruction from the thread on the ALUs
- Simultaneous multi-threading (SMT)
  - Extension of out-of-order CPU design
  - Example: Intel Hyper-threading (2 threads per core)

Simple Multithreaded Pipeline（SMT）



- Have to carry thread select down pipeline to ensure correct state bits read/written at each pipe stage
- Appears to software (including OS) as multiple, albeit slower, CPUs

Simultaneous Multi-Threading(a.k.a Hyperthreading)



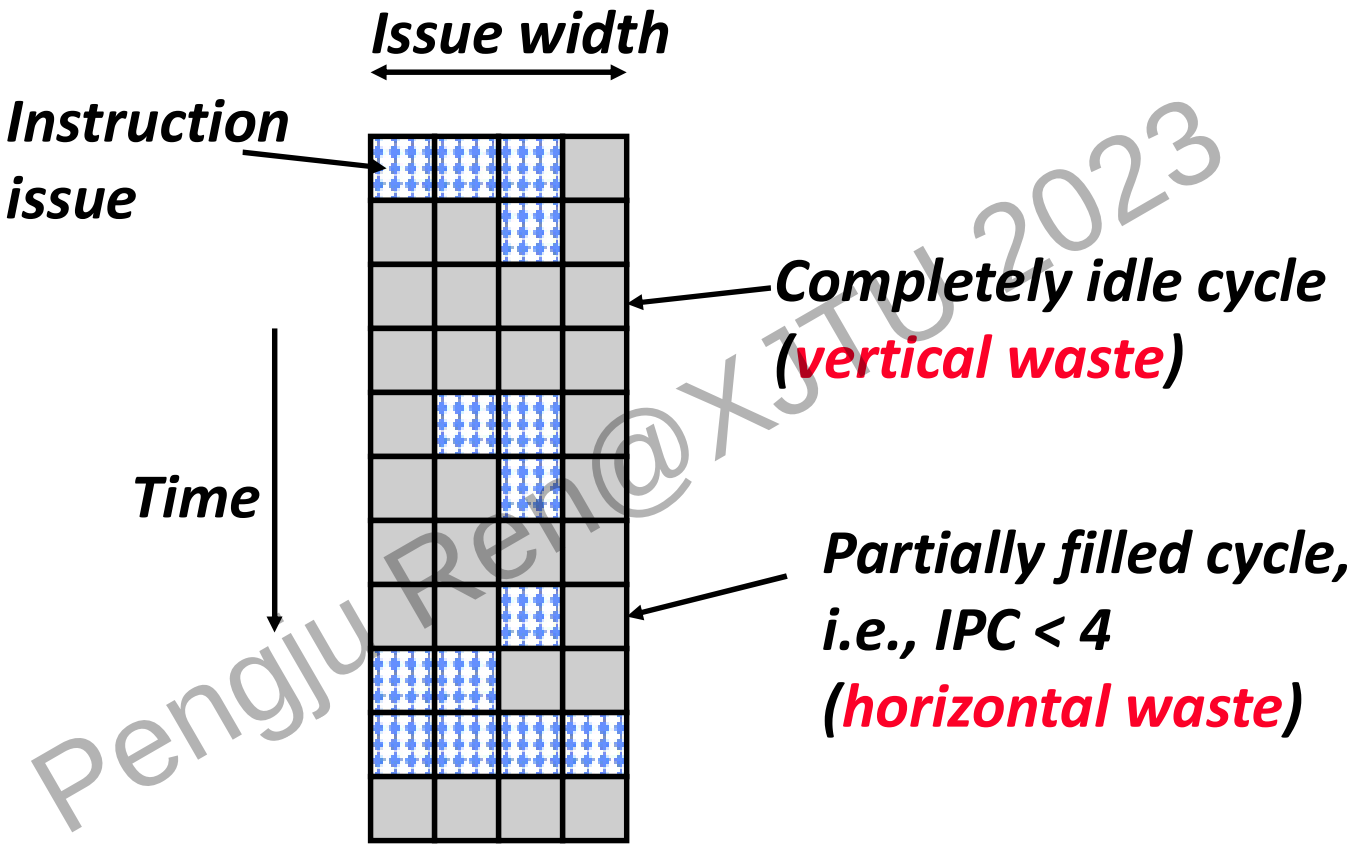
Replicate in-order resources

Share out-of-order resources

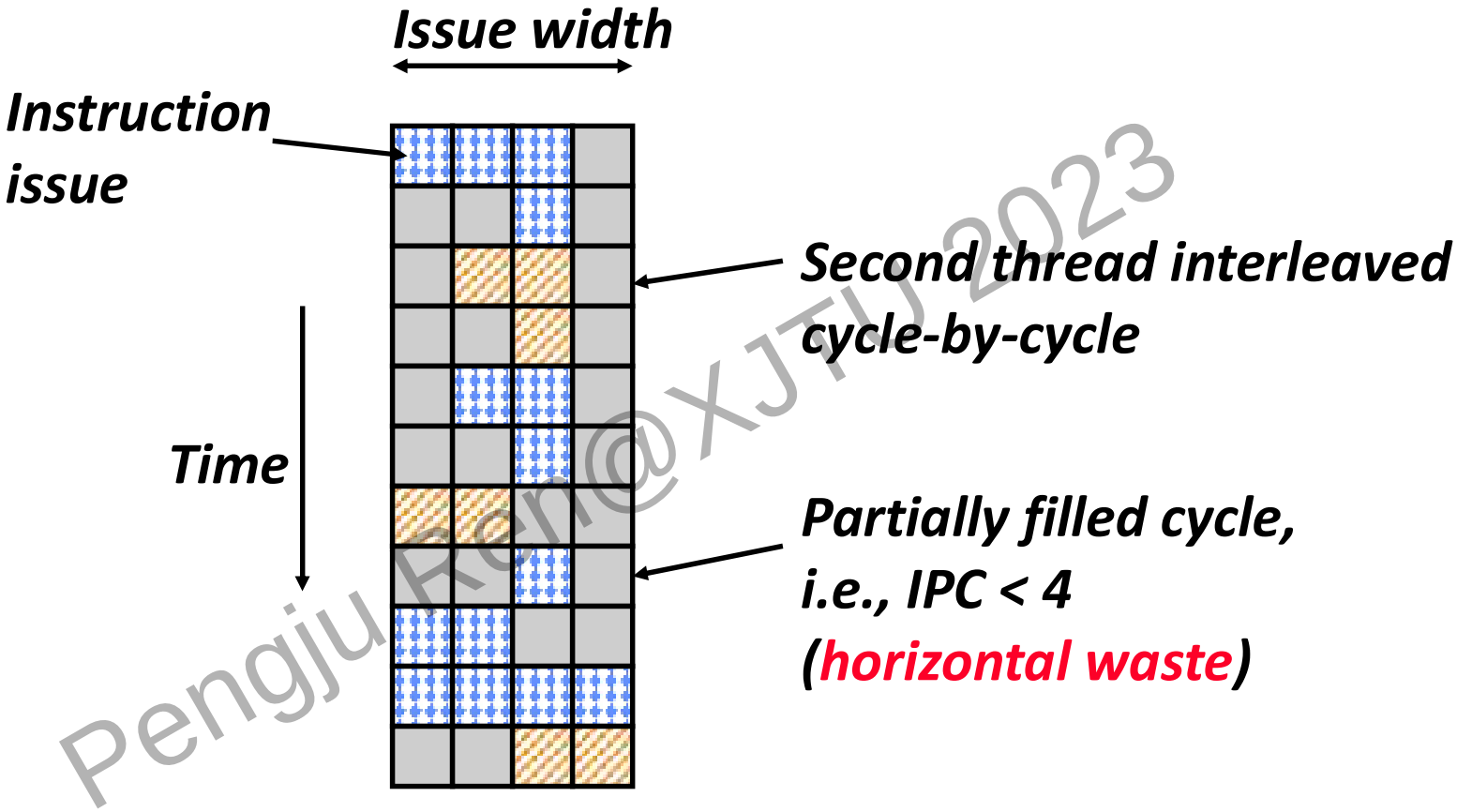
- **Does not improve (or penalize) single-thread performance**
- **May get improved multi-threaded performance**

- **Single core (physically) does the work of multiple cores (logically)**
  - **Fetch/decode independent instruction streams from different threads**
  - **Map onto shared out-of-order execution unit**
  - **Make fuller use of execution unit when ILP is low**
  - **Reduce impact of stalls / mispredictions**

Superscalar Machine Efficiency



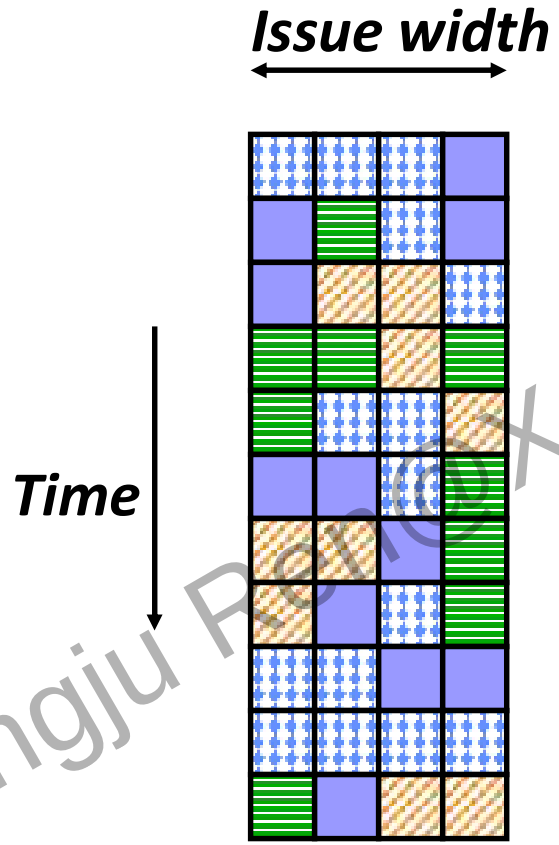
Vertical Multithreading (temporal multi-threading)



- Cycle-by-cycle interleaving removes vertical waste, but leaves some horizontal waste

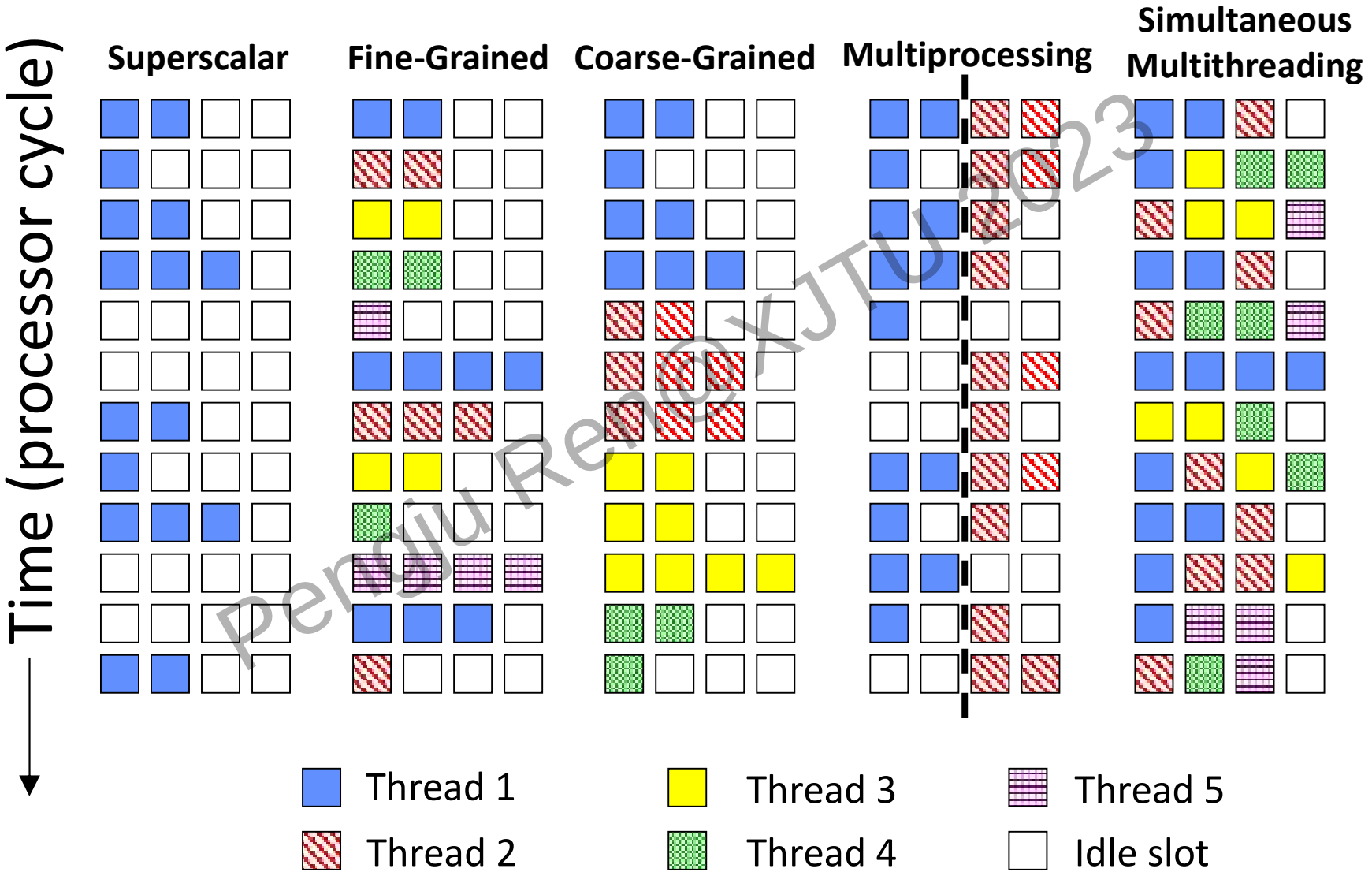
Ideal Superscalar Multithreading

[Tullsen, Eggers, Levy, UW, 1995]



- Interleave multiple threads to multiple issue slots with no restrictions (no data dependence between threads, hence no stall or forwarding among threads)

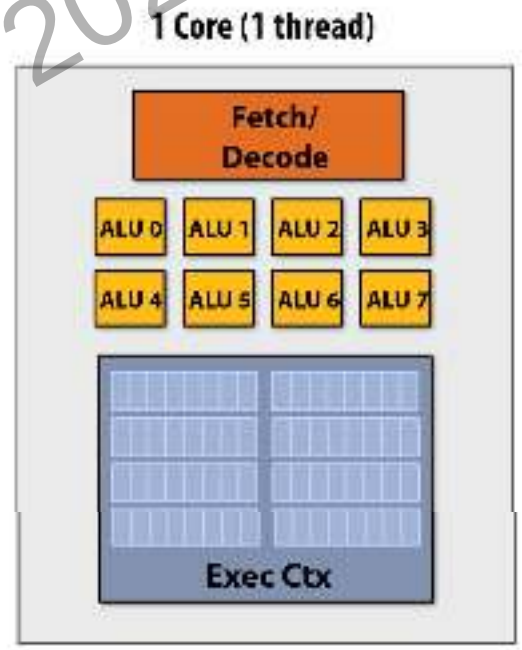
Summary: Multithreaded Categories



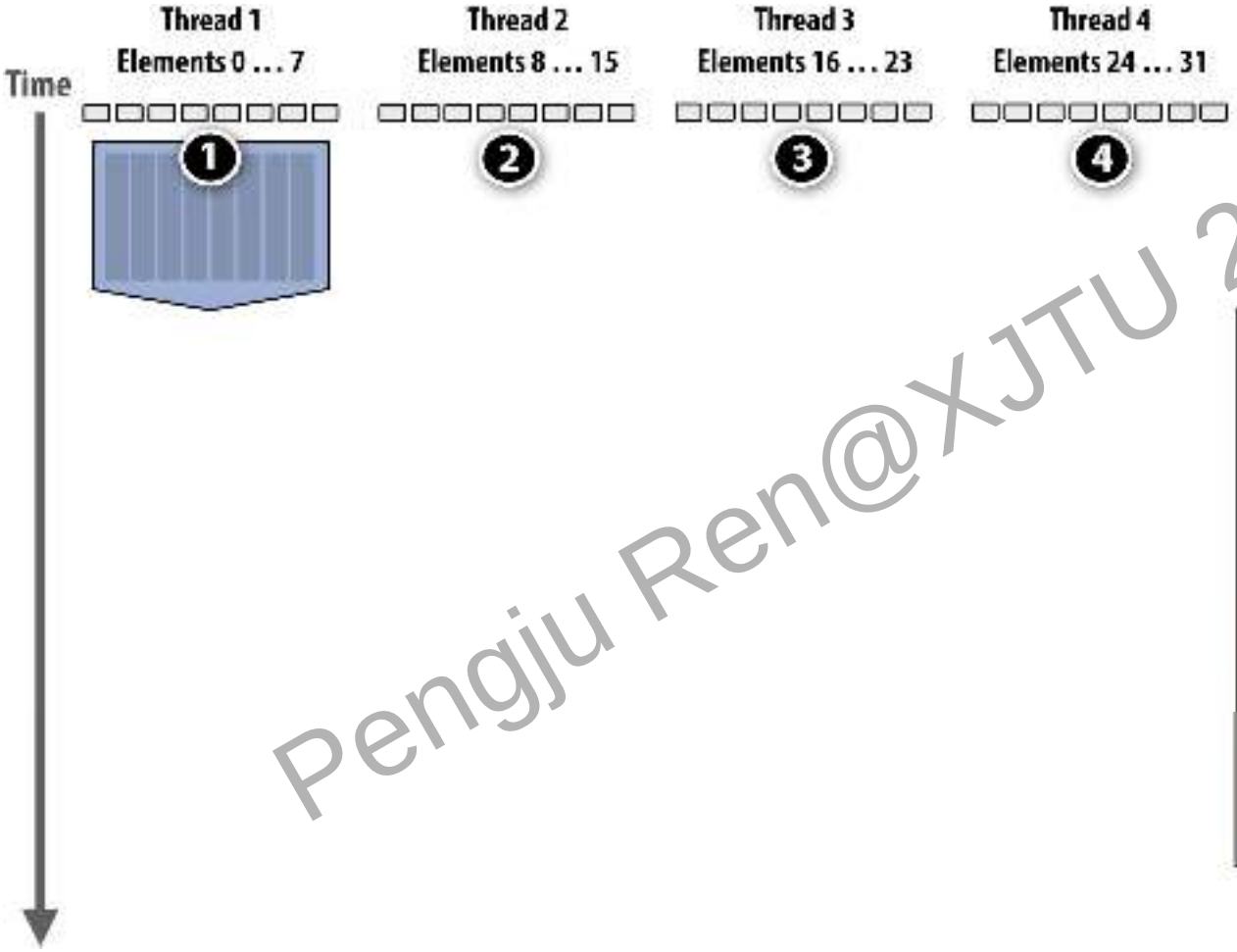
Multi-threading reduces stalls

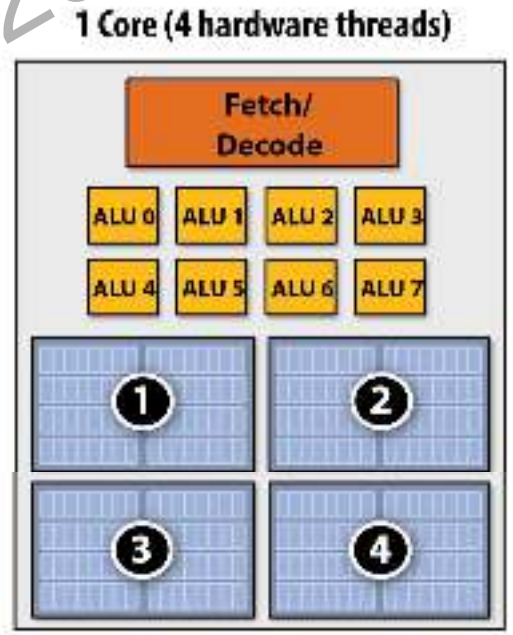
- Idea: interleave processing of multiple threads on the same core to hide stalls
- Helps even with in-order processing of instructions
- Like prefetching, multi-threading is a latency hiding, not a latency reducing technique
- We will consider several forms of multi-threading
- Context switch is costly

Hiding stalls with Multi-Threading

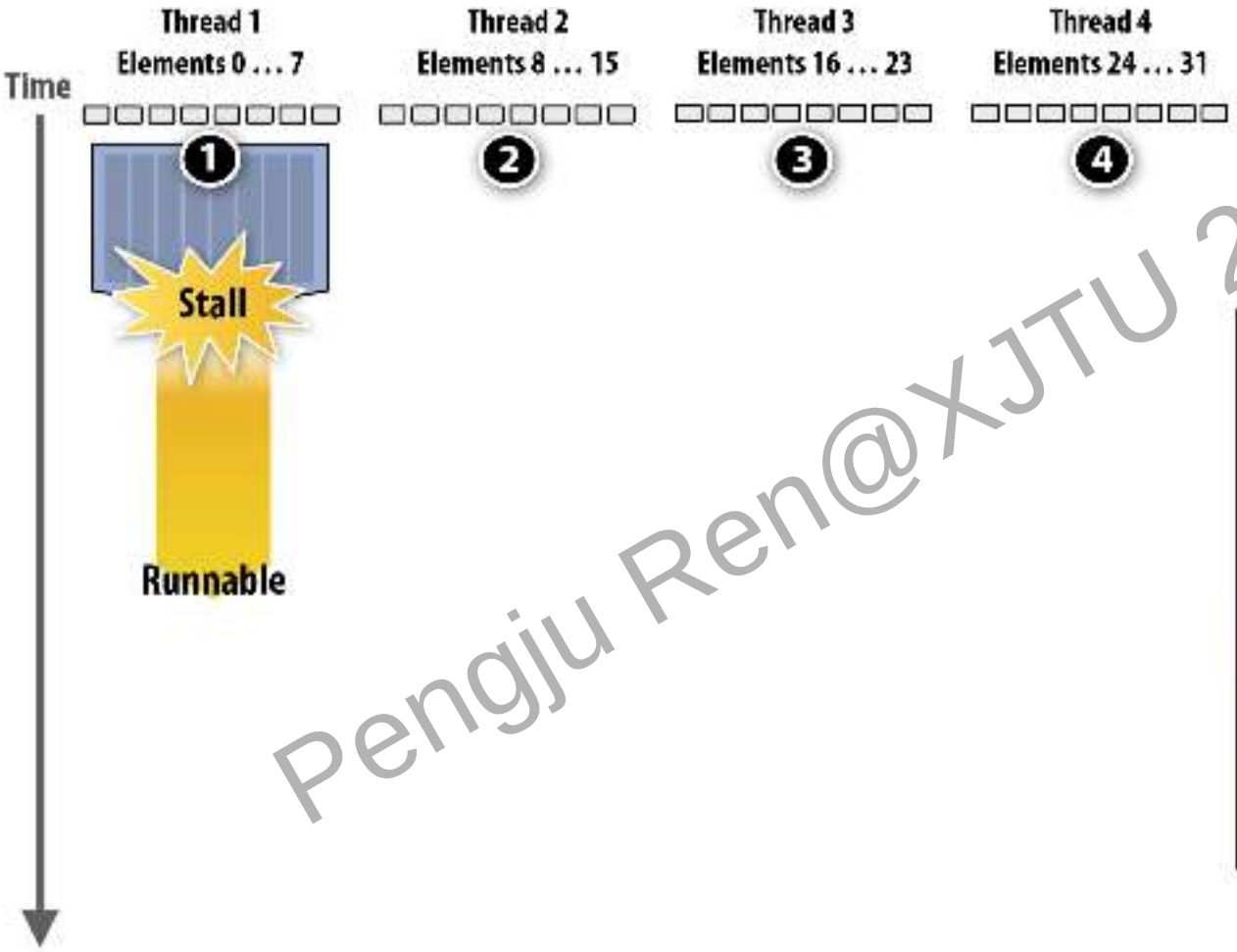


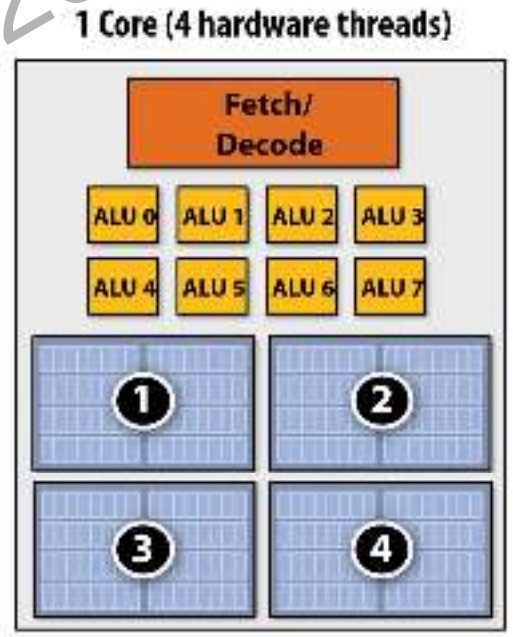
Hiding stalls with Multi-Threading



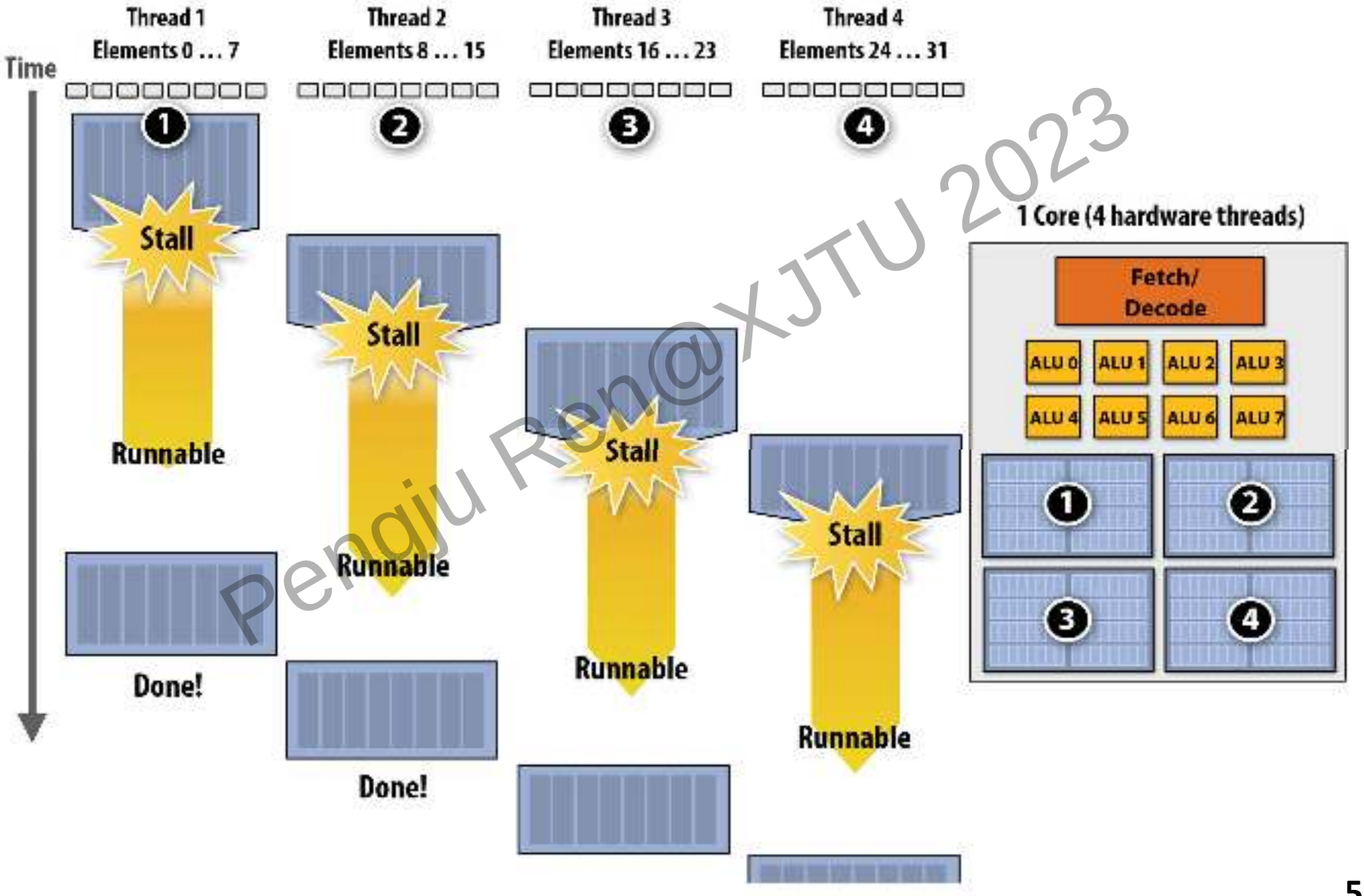


Hiding stalls with Multi-Threading

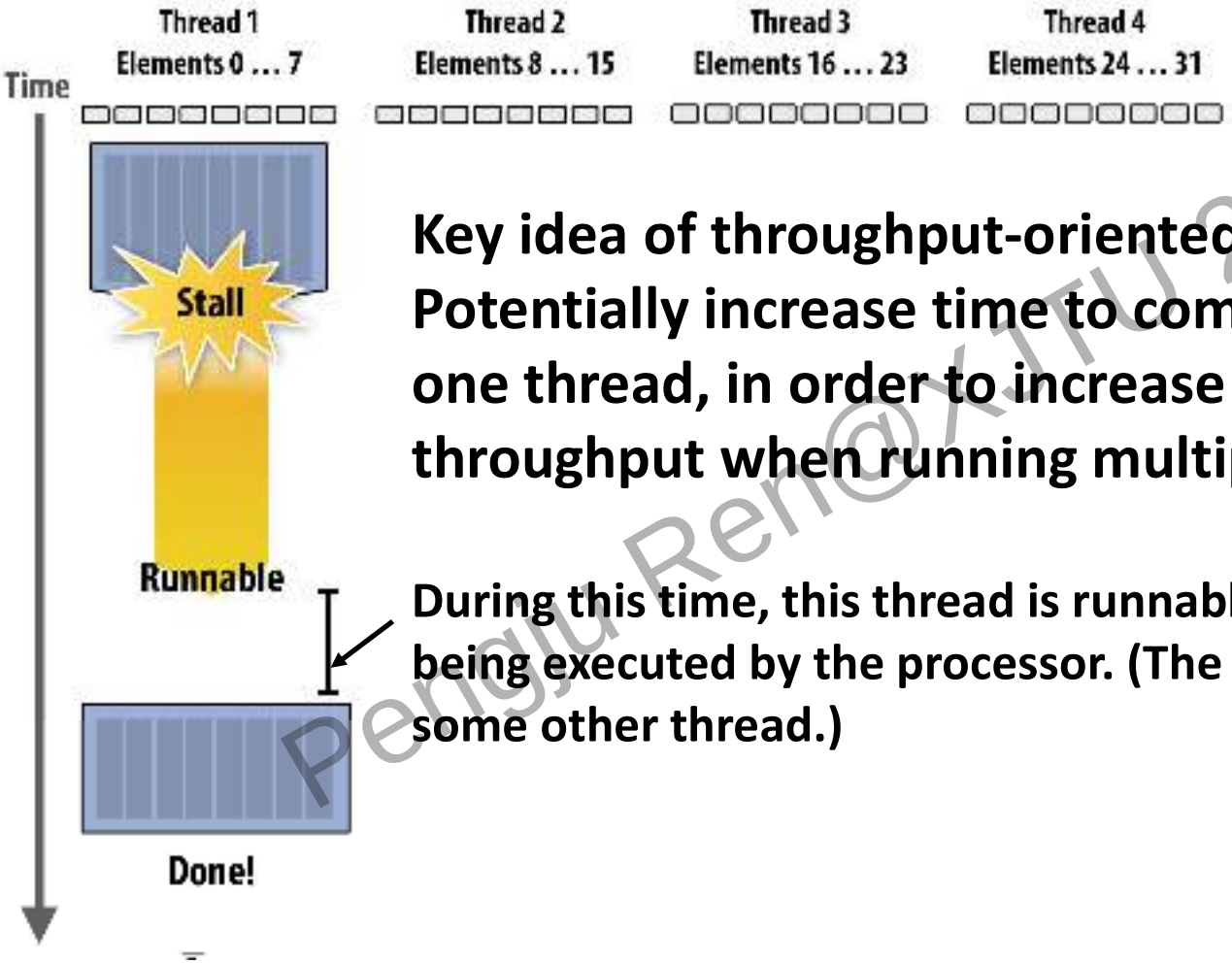




Hiding stalls with Multi-Threading



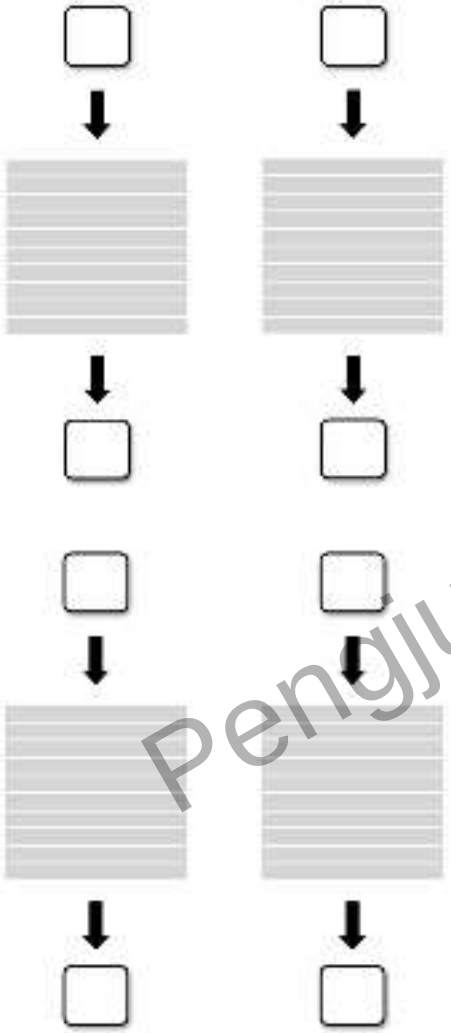
Throughput computing trade-off

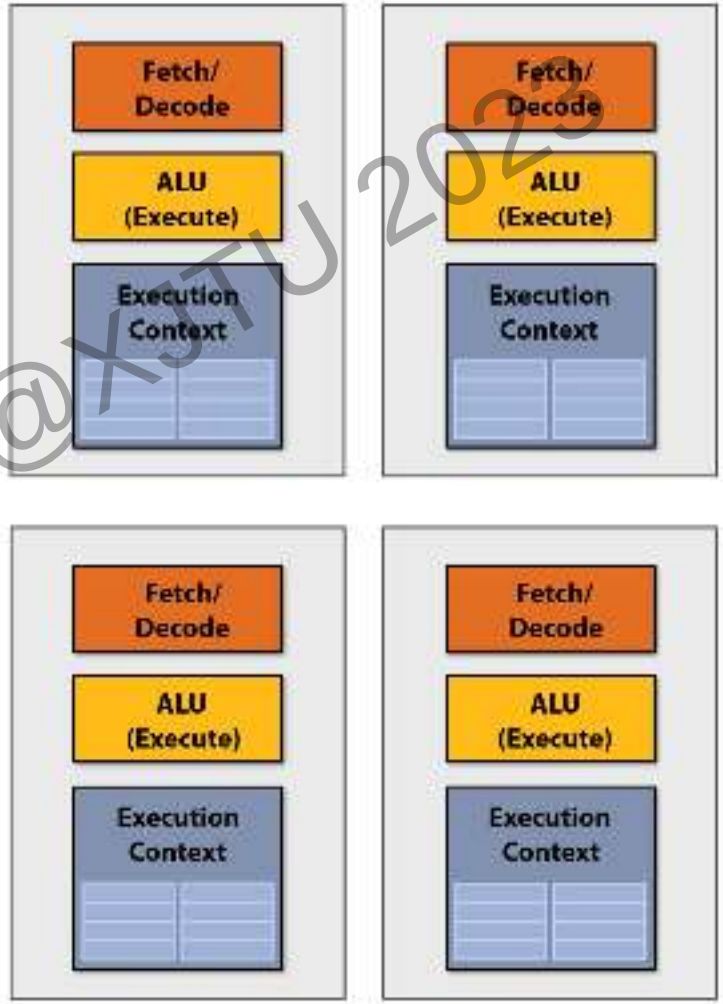


Key idea of throughput-oriented systems: Potentially increase time to complete work by any one thread, in order to increase overall system throughput when running multiple threads.

During this time, this thread is runnable, but it is not being executed by the processor. (The core is running some other thread.)

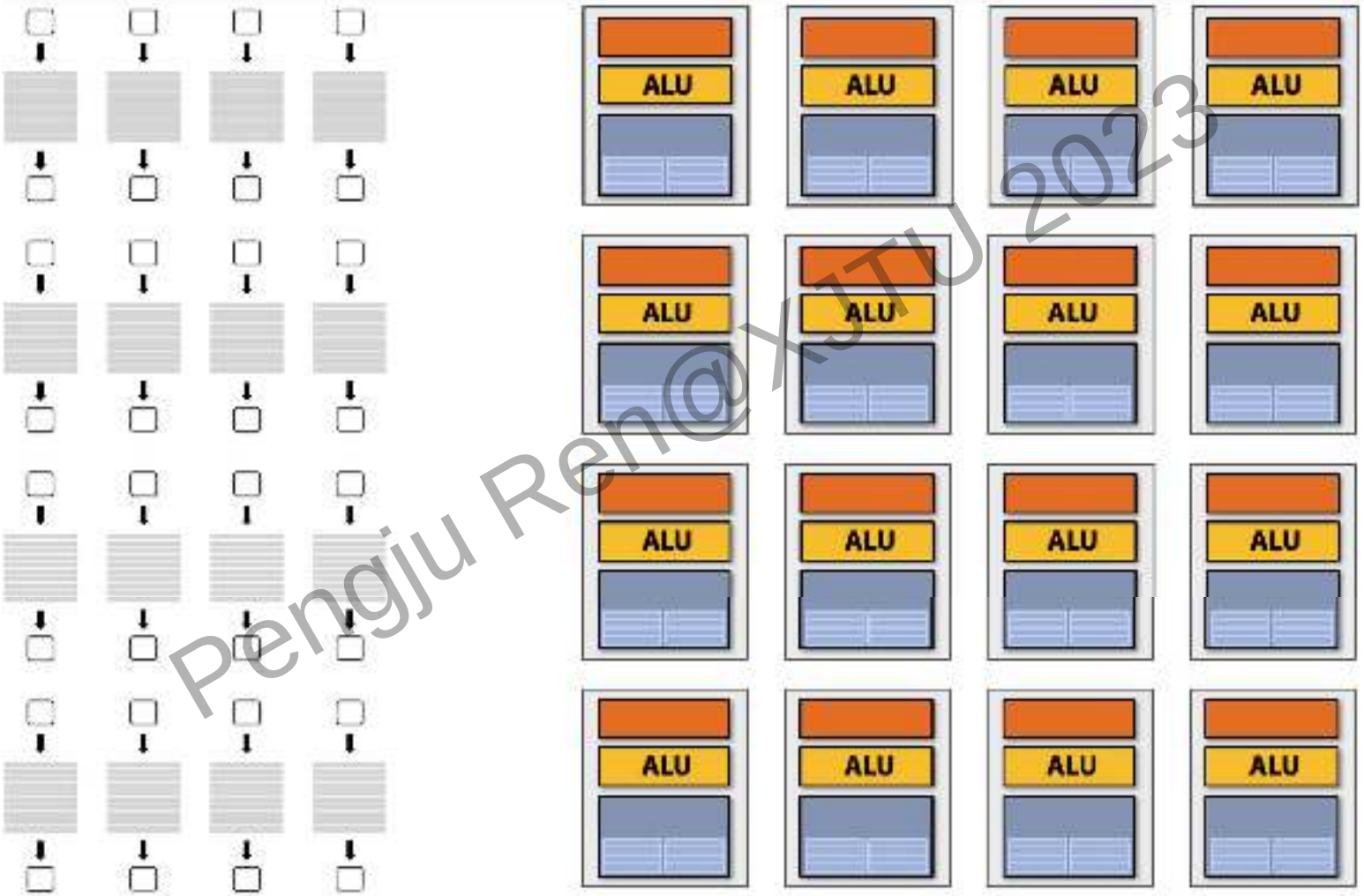
Four cores: compute four elements in parallel





Four cores, four simultaneous instruction streams

Sixteen cores: compute sixteen elements in parallel



Sixteen cores, sixteen simultaneous instruction streams

Logical Core and Physical Core

Physical cores are physical units on a CPU, whereas, Logical core are a software abstraction

Logical processors are also related to threads. They are enabled by the Simultaneous Multi-Threading (short for SMT, hyper-threading technology for Intel Processors.)

Logical cores are the number of Physical cores times the number of threads that each core can run.

CPU Specifications

Total Cores	6
Total Threads	12
Max Turbo Frequency	4.10 GHz
Intel® Turbo Boost Technology 2.0 Frequency‡	4.10 GHz
Processor Base Frequency	2.20 GHz

e.g. if you have a 6 core processors like the Intel Core i7-8750H processor with hyper threading enabled (2 thread/core), you essentially get 12 threads(logical cores) running simultaneously.

Multi-threading summary

- Benefit: use a core's execution resources more efficiently
  - Hide memory latency
  - Fill multiple functional units of superscalar architecture (when one thread has insufficient ILP)
- Costs
  - Requires additional storage for thread contexts
  - Increases run time of any single thread (often not a problem, we usually care about throughput in parallel apps)
  - Requires additional independent work in a program (more independent work than ALUs!)
  - Relies heavily on memory bandwidth
    - More threads → larger working set → less cache space per thread
    - May go to memory more often, but can hide the latency

Embarrassingly Parallel Processing

Summing 10,000 elements from array `A[ ]`

In Sequential Algorithm:

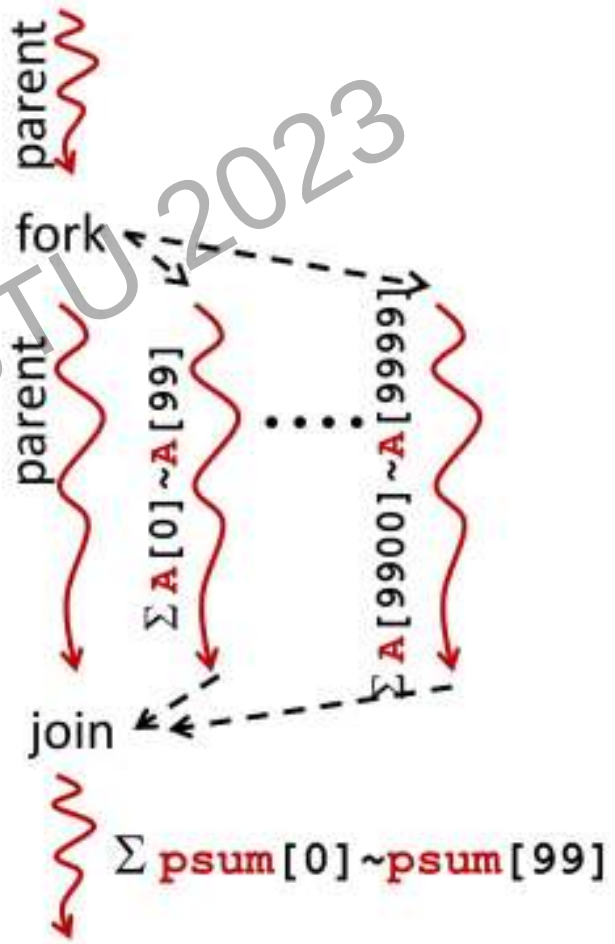
```
For (i=0; i<10000; i=i+1)
    sum = sum + A[i]
```

Assuming add"+" is 1 clock cycle, with perfect mem access

T = 10,000

Shared-Memory Pthreads Strategy 1

- Fork p=100 threads on a p-way shared memory multiprocessor (`A[10000]` and `Psum[i]` are in shared Mem)
- Child thread-i uses `psum[i]` to compute its portion of the partial sum
- When all threads finish, parent sums `psum[0]~psum[99]`



Thread Code of Strategy 1

```
double A[ARRAY_SIZE];
double psum[p];

void *sumParallel(void *_id) {
  long id=(long) _id;
  long i;

  psum[id]=0;

  for(i=0;i<(ARRAY_SIZE/p);i++)
    psum[id]+=A[id*(ARRAY_SIZE/p)+i];

  return NULL;
}
```

Children Thread Code

```
double A[ARRAY_SIZE];
double psum[p];
double sum=0;

int main(){
  //... skipped pthreads
boilerplate ...
  for(i=0; i<p; i++ )
    pthread_create( &tid[i],
                    NULL,
                    sumParallel,
                    (void*)i);
  for (i=0; i<p; i++ ) {
    pthread_join(tid[i],&retval);
    sum+=psum[i];
  }
}
```

Parent Thread Code

Performance Analysis (Strategy 1)

Summing 10,000 on 100 cores

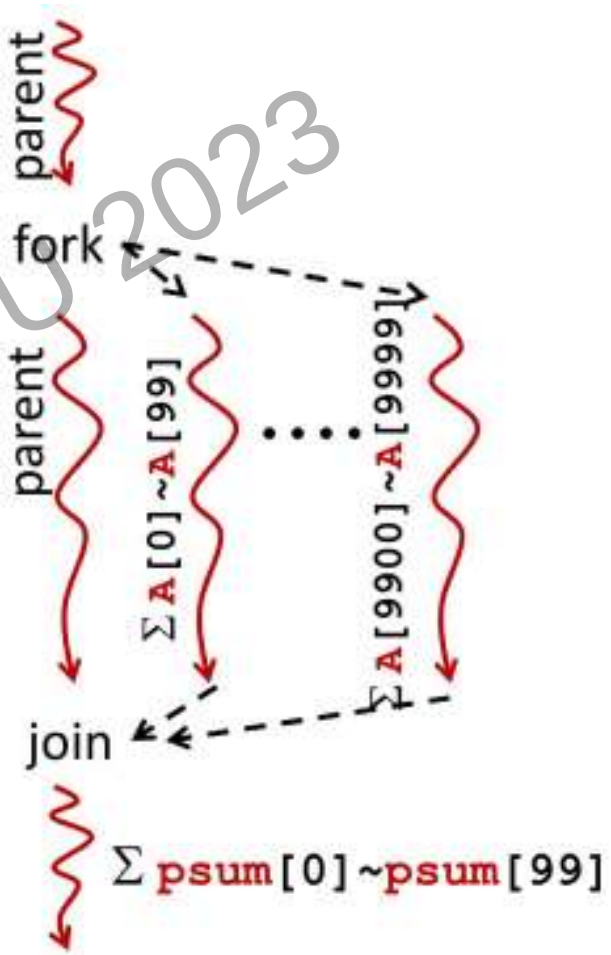
- 100 threads performs 100 +'s each in parallel
- parent thread performs 100 +'s sequentially
- T_{100}= 100 + 100
- $Speedup_{100}$= 50

If summing 100,000 on 100 cores

- T_{100}= 1000 + 100
- $Speedup_{100}$= 90.9

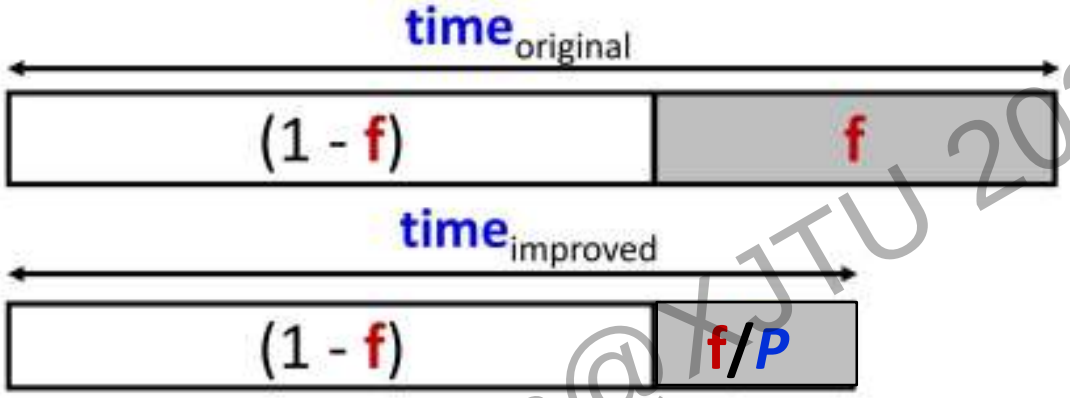
If summing 10,000 on 10 cores

- T_{10}= 1000 + 10
- $Speedup_{10}$= 9.9



Recap : Amdal's Law on Speedup

If only a fraction *f* (parallel fraction of a program) is speedup by *P*



$$\text{Speedup} = \frac{1}{(1-f)+f/P}$$

if f is small, *P* doesn't matter

even f is large, diminishing return on *P*, eventually "1-f" dominates

Parallel portion is usually not perfectly parallel

- Synchronization overhead (e.g., updates to shared data, data races)
- Load imbalance overhead (imperfect parallelization)
- Resource sharing overhead (contention among N processors)

Thread Code of Strategy 2

```
double A[ARRAY_SIZE];
double psum[p];

void *sumParallel(void *_id) {
  long id=(long) _id;
  long i;

  psum[id]=0;

  for(i=0;i<(ARRAY_SIZE/p);i++)
    psum[id]+=A[id*(ARRAY_SIZE/p)+i];

  sum=sum+psum[id];

  return NULL;
}
```

Each children thread do a bit more of the reduction, but will cause "data race"

Children Thread Code

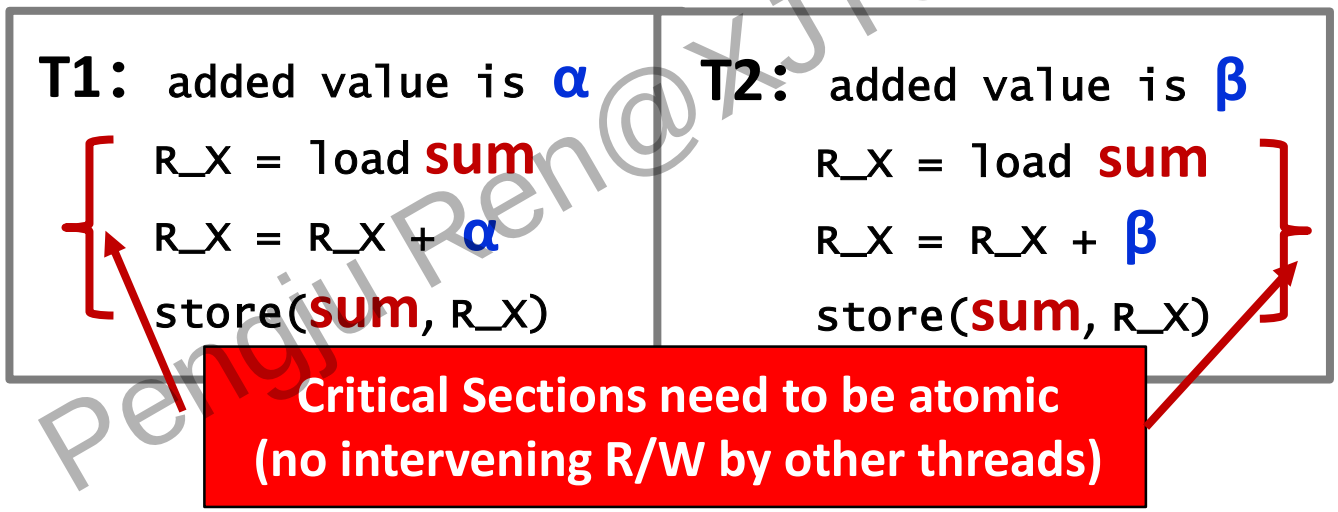
```
double A[ARRAY_SIZE];
double psum[p];
double sum=0;

int main(){
  //... skipped pthreads
boilerplate ...
  for(i=0; i<p; i++ )
    pthread_create( &tid[i],
                    NULL,
                    sumParallel,
                    (void*)i);
  for (i=0; i<p; i++ ) {
    pthread_join(tid[i],&retval);
    sum+=psum[i];
  }
}
```

Parent Thread Code

Data Races

For Strategy 2, sum is read(load) and updated(store) by all threads at around the same time

For simplicity, assuming there are two threads T1 and T2, and sum is initially 0



What are the possible final values of sum ?

α、β and α+β depending on the interleaving of the load/update/store sequence in T1 and T2

Critical Sections

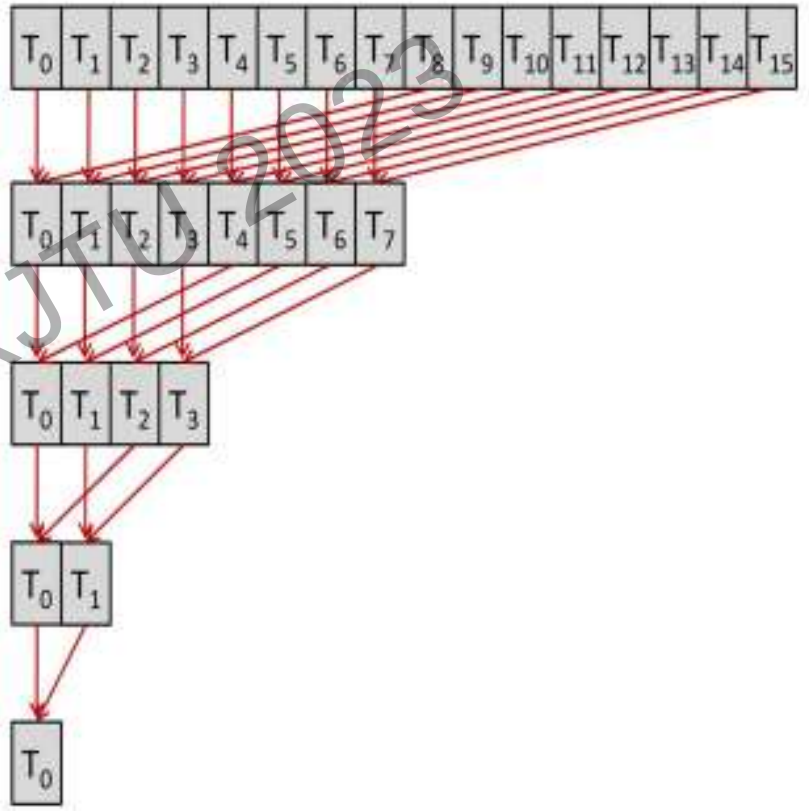
Special "lock" variables and lock/unlock operators to demarcate a "critical section" that only one thread can enter at a time, e.g.,

```
pthread_mutex_lock(&lockvar)
sum = sum + psum[id];
pthread_mutex_unlock(&lockvar)
```


- `lock()` **blocks until lockvar is free or freed (released by previous owner)**
- **On `unlock()`, if multiple `lock()` pending, only 1 should succeed; the rest keep waiting**
- **Strategy 2 is now correct but actually slower**

Thread Code of Strategy 3

```
double A[ARRAY_SIZE];
double psum[p];
void *sumParallel(void *_id) {
  long id=(long) _id;
  long i;
  psum[id]=0;
  for(i=0;i<(ARRAY_SIZE/p);i++)
    psum[id]+=A[id*(ARRAY_SIZE/p)+i];
  remain=p;
  do {
    pthread_barrier_wait(&barrier);
    half=(remain+1)/2;
    if (id<(remain/2))
      psum[id]=psum[id]+psum[id+half];
      remain=half;
    } while (remain>1);
  return NULL;
}
```



Children Thread Code

Performance Analysis (Strategy 3)

Summing 10,000 on 100 cores

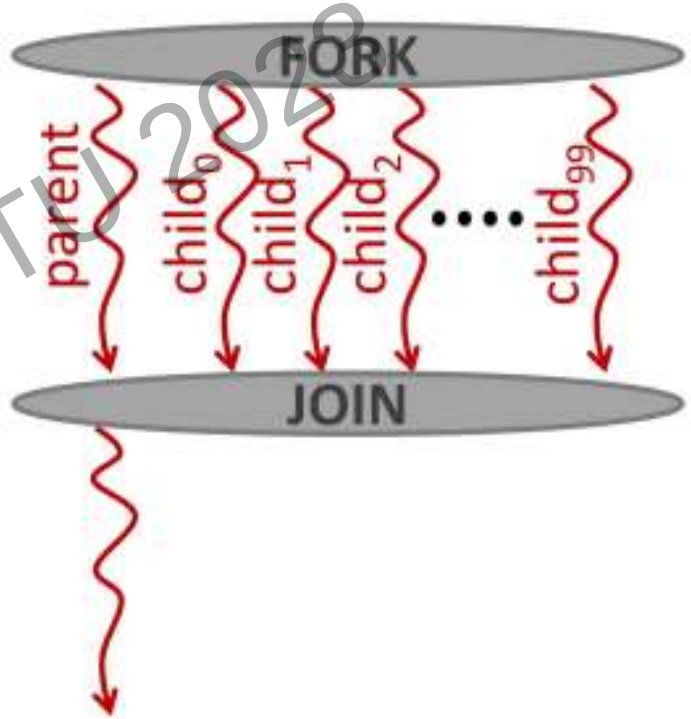
- 100 threads performs 100 +'s each in parallel
- –children thread performs parallel reduction
- T_{100}= 100 + 7
- $Speedup_{100}$= 93.5

If summing 100,000 on 100 cores

- T_{100}= 1000 + 7
- $Speedup_{100}$= 99.3

If summing 10,000 on 10 cores

- T_{10}= 1000 + 4
- $Speedup_{10}$= 10.0



Concerns of Memory Consistency

Memory consistency model says for each read which write bound the value to be returned

Intuitively: *a read should return value of "most recent" write to the same address*

In a shared-memory multicore, threads(cores) T1/T2/T3 perform following streams of reads and writes

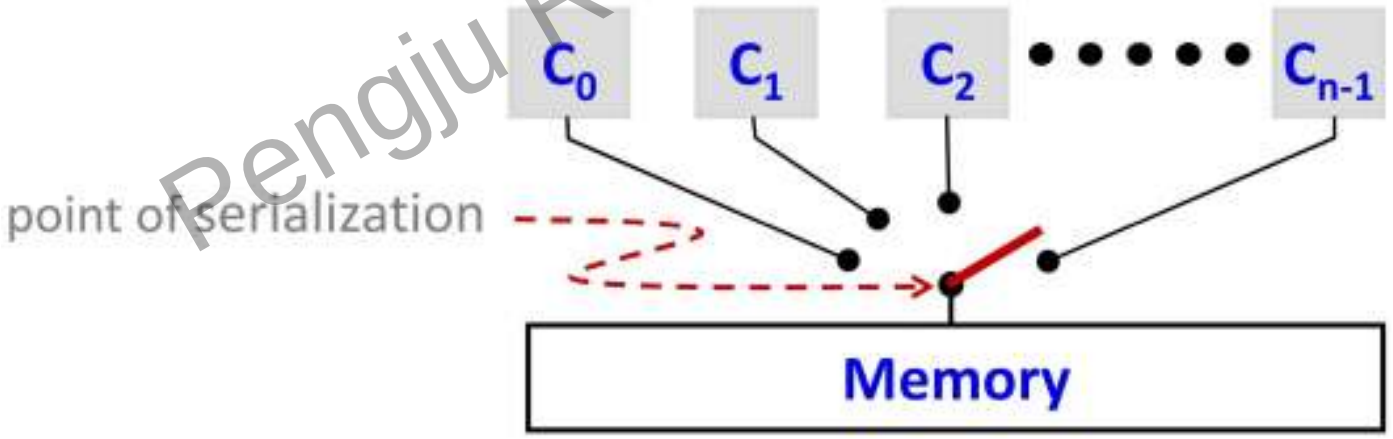
```
T1:   … ST(X) …
T2:   … ST(X), ST(X), ST(Y), LD(X), LD(Y)…
T3:   … ST(X), ST(Y), ST(X) …
```

Which is the last write to x before `LD(x)` by T2?

Sequential Consistency (SC)

A thread perceives its own memory ops in program order (of course)

- **Memory ops from threads in program order can be interleaved arbitrarily; different interleaving allowed on different runs, i.e., nondeterminism**
- **Across different executions, different global orders can be observed (Debugging is difficult as order changes across runs)**



(Memory is a switch that services one load or store at a time from any processor)

Issues with Sequential Consistency

- Performance enhancement techniques that could make SC implementation difficult
- Out-of-order execution
  - Loads happen out-of-order with respect to each other and with respect to independent stores → makes it difficult for all processors to see the same global order of all memory operations
- Caching
  - A memory location is now present in multiple places
  - Prevents the effect of a store to be seen by other processors → makes it difficult for all processors to see the same global order of all memory operations

SC Example: what can and cannot be

Threads T1 and T2 and shared locations X and Y

(initially X = 0, Y = 0)

T1:	T2:
. . . .	. . . .
store(X, 1);	vy = load(Y);
store(Y, 1);	vx = load(X);
. . . .	. . . .

SC says:

– vy and vx may get different values from run to run

e.g., (vy=0, vx=0), (vy=0, vx=1), or (vy=1, vx=1)

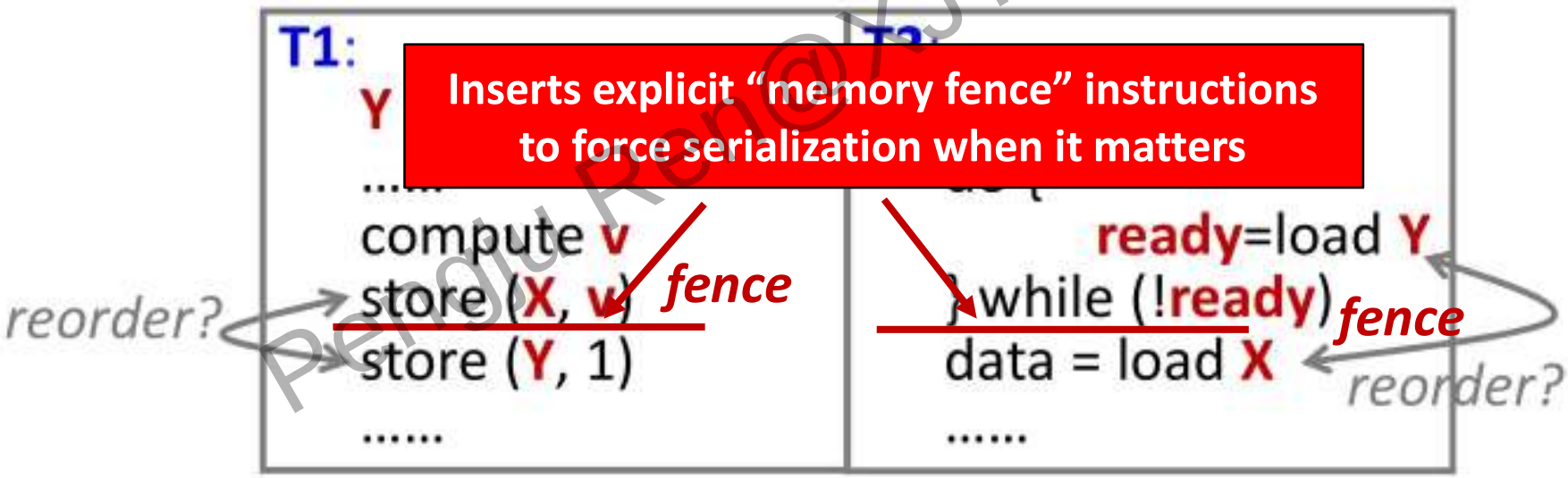
– but if vy is 1 then vx cannot be 0

An Useful Example: Producer & Consumer

Threads T1 and T2 communicate via shared memory locations X and Y

– T1 produces result in X to be consumed by T2

– T1 signals readiness to T2 by setting Y



This works because SC says T1 and T2 must see the stores to X and Y in the same order

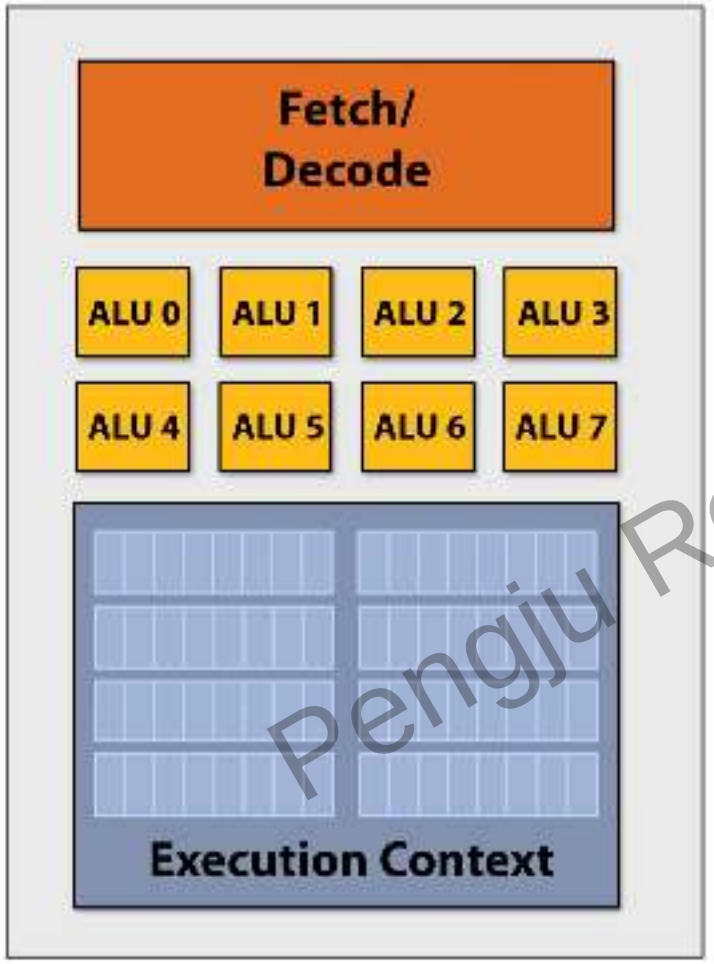
Weaker Memory Consistency

- The ordering of operations is important when the order affects operations on shared data → i.e., when processors need to synchronize to execute a "program region"
- Weak consistency
  - Idea: Programmer specifies regions in which memory operations do not need to be ordered
  - "*Memory fence*" instructions delineate those regions
    - » All memory operations before a *fence* must complete before fence is executed
    - » All memory operations after the *fence* must wait for the fence to complete
    - » *fences* complete in program order
  - All synchronization operations act like a *fence*

Main Design Issues in Tightly-Coupled MP

- **Shared memory synchronization**
  - How to handle locks, atomic operations, barrier synch
- **Memory consistency:** Ordering of all memory operations
  - What should the programmer expect the hardware to provide? Communication is necessary when tasks need values from each other
- **Cache coherence**
  - How to ensure correct operation in the presence of private caches keeping the same memory address cached
- **Shared resource management**
- **Communication: Interconnects**

Add ALUs to increase compute capability



Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

Single instruction, multiple data

Same instruction broadcast to all ALUs
Executed in parallel on all ALUs

Data Parallel

- SIMD exploits operation-level parallelism on different data
  - Same operation concurrently applied to different pieces of data
  - A form of ILP where instruction happens to be the same across data
- Basic requirements
  - Need to load/store vectors → vector registers (contain vectors)
  - Need to operate on vectors of different lengths → vector length register (VLEN)
  - Elements of a vector might be stored apart from each other in memory → vector stride register (VSTR)
    - » Stride: distance in memory between two elements of a vector

Scalar v.s Vector Program (using AVX intrinsics)

```
void sinx(int N, int terms, float * x,
                          float *result) {
    for (int i=0; i<N; i++) {
        float value = x[i];
        float numer = x[i]*x[i]*x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

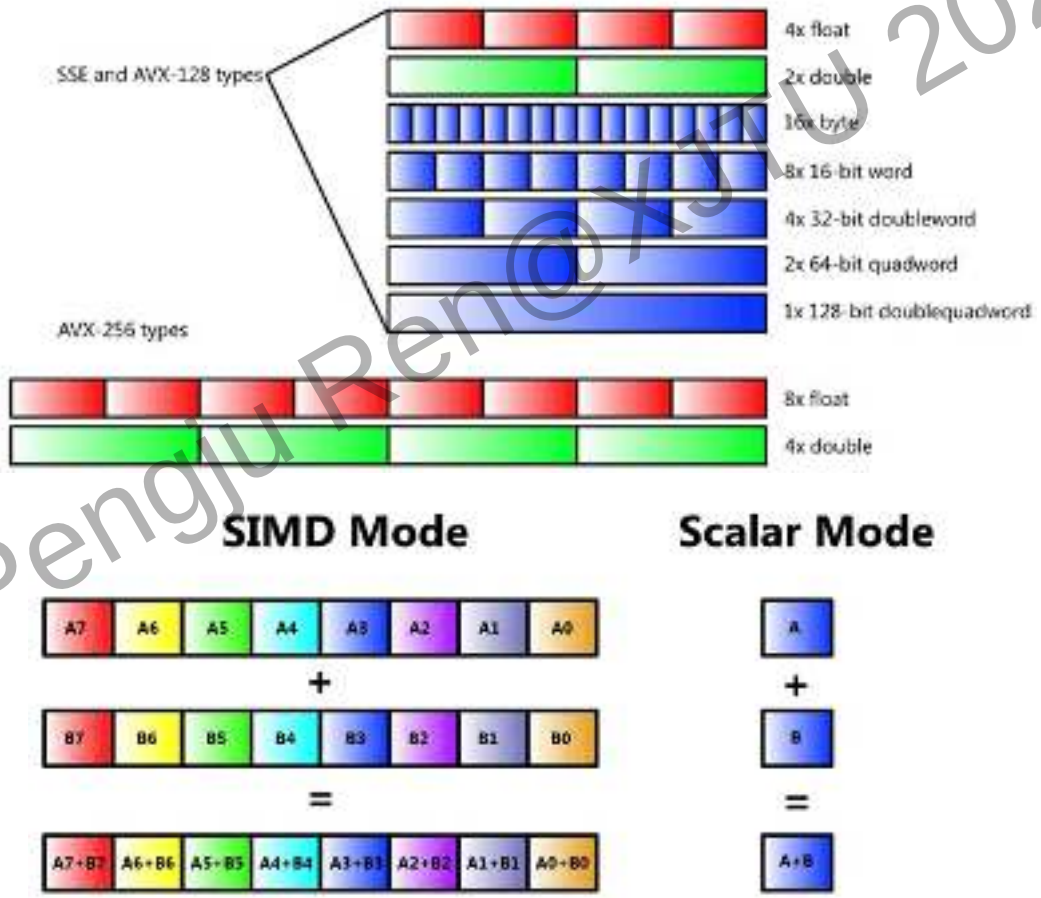
```
#include <immintrin.h> //get AVX intrinsics
void sinx(int N, int terms, float* x, float* result) {
   float three_fact = 6;  // 3!
  for (int i=0; i<N; i+=8)    {
       __m256 origx = _mm256_load_ps(&x[i]);
       __m256 value = origx;
       __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
       __m256 denom = _mm256_set1_ps(three_fact);
       float sign = -1;

       for (int j=1; j<=terms; j++)     {
         // value += sign * numer / denom
         __m256 tmp = _mm256_div_ps(_mm256_mul_ps(_mm256_set1ps(sign),
                                    numer), denom);
       value = _mm256_add_ps(value, tmp);
       numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
       denom = _mm256_mul_ps(denom, _mm256_set1ps((2*j+2) * (2*j+3)));
       sign *= -1;       }
       _mm256_store_ps(&result[i], value);    } //end for i
     }
```


Processes one array element using scalar instructions on scalar registers (e.g., 32-bit floats)

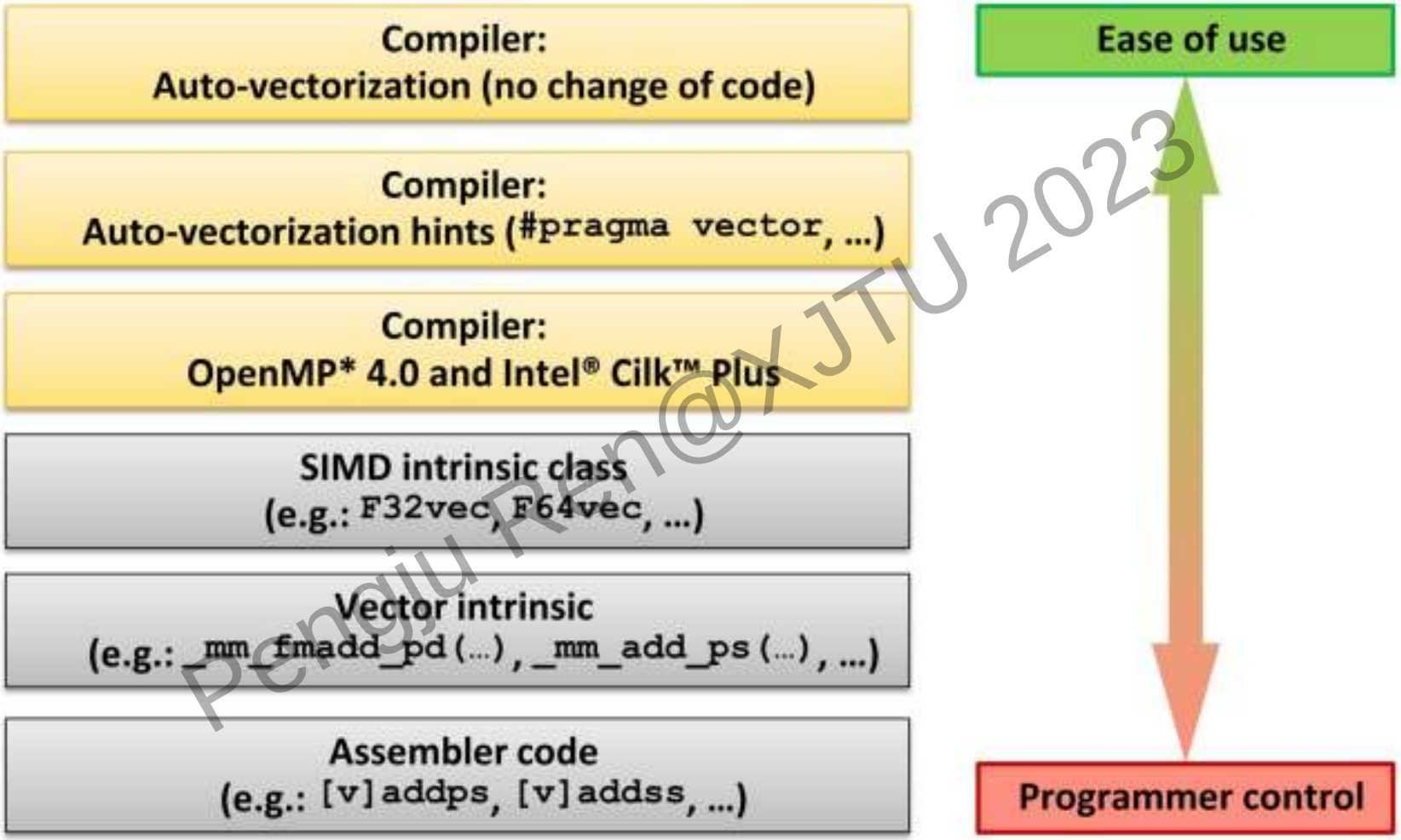
Processes 8 array elements simultaneously using vector instructions on 256-bit vector registers (e.g., 256=32-bit floats x 8)

A short intro about Intel AVX

Intel® Advanced Vector Extensions (Intel® AVX) is a set of instructions for doing Single Instruction Multiple Data (SIMD) operations on Intel® architecture CPUs.



Many ways to Vectorize



Intel compilers use heuristics to vectorize, also there are extensions that allow expression of vectorization explicitly

Example using AVX intrinsics

```
#include <immintrin.h>

double A[40], B[40], C[40];
for (int i = 0; i < 40; i += 4) {
        __m256d a = _mm256_load_pd(&A[i]);
        __m256d b = _mm256_load_pd(&B[i]);
        __m256d c = _mm256_add_pd(a, b);
        _mm256_store_pd(&C[i], c);
}
```

```
#include <immintrin.h>

double A[40], B[40], C[40];
for (int i = 0; i < 40; i += 8) {
        __m512d a = _mm512_load_pd(&A[i]);
        __m512d b = _mm512_load_pd(&B[i]);
        __m512d c = _mm512_add_pd(a, b);
        _mm512_store_pd(&C[i], c);
}
```

Intel AVX intrinsic (1)

_mm<bit_width>_<name>_<data_type>(data_type param1, data_type param2, data_type param3)

- <bit_width> identifies the size of the vector returned by the function. For 128-bit vectors, this is empty. For 256-bit vectors, this is set to 256.
- <name> describes the operation performed by the intrinsic
- <data_type> identifies the data type of the function's primary arguments
  - ps - vectors contain floats (ps stands for packed single-precision)
  - pd - vectors contain doubles (pd stands for packed double-precision)
  - epi8/epi16/epi32/epi64 - vectors contain 8-bit/16-bit/32-bit/64-bit signed integers
  - epu8/epu16/epu32/epu64 - vectors contain 8-bit/16-bit/32-bit/64-bit unsigned integers

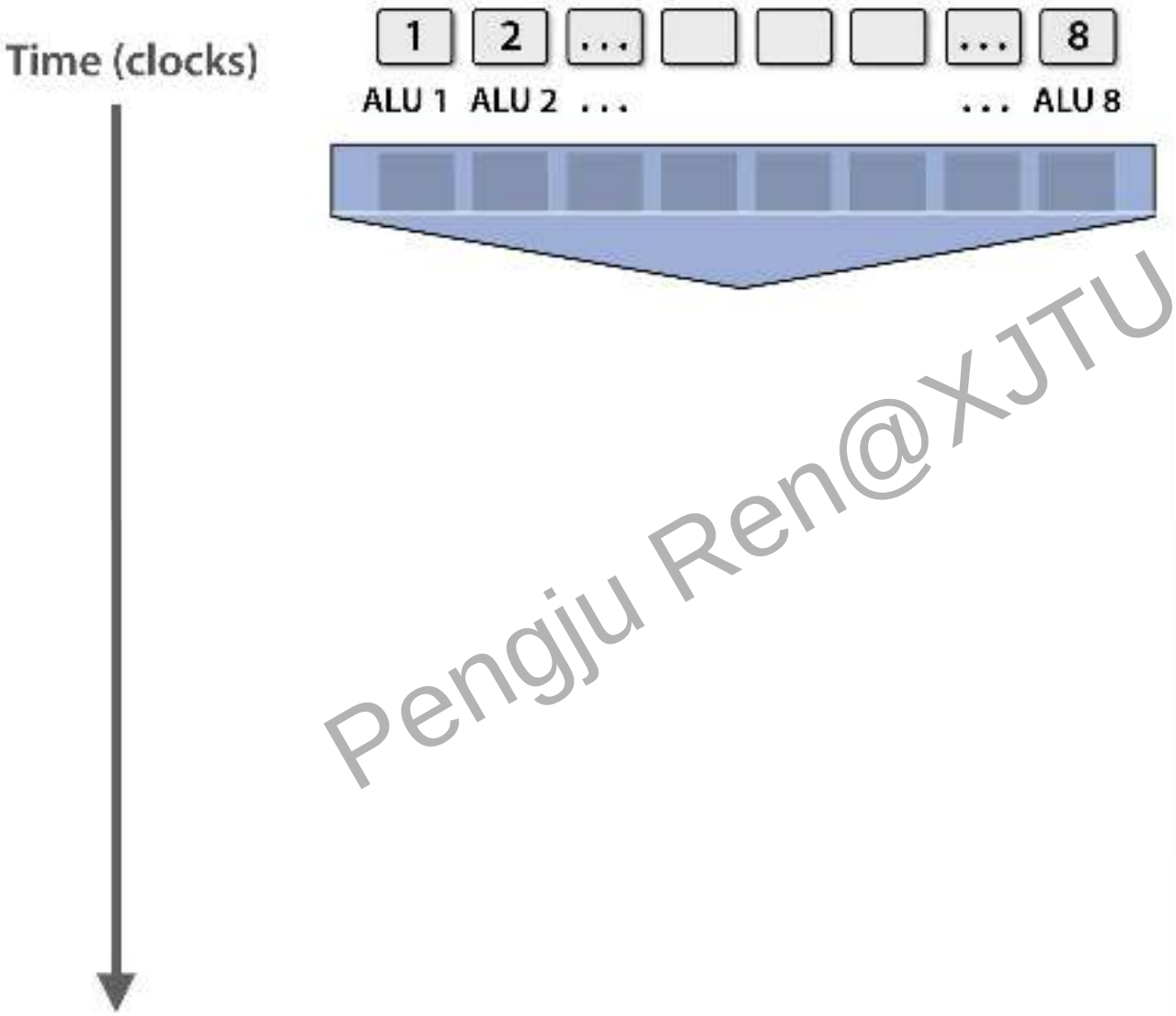
https://software.intel.com/en-us/intel-isa-extensions

Intel AVX intrinsic (2)

Function	Description
_mm256_load_ps/pd	Loads a floating-point vector from an aligned memory address
_mm256_mul_ps/pd	Multiply two floating-point vectors
_mm256_set1_ps/pd	Fill a vector with a floating-point value
_mm256_div_ps/pd	Divide two floating-point vectors
_mm256_add_ps/pd	Add two floating-point vectors
_mm256_store_ps/pd	Moves packed floating point values from a float32 vector to an aligned memory location

https://software.intel.com/sites/landingpage/IntrinsicsGuide/

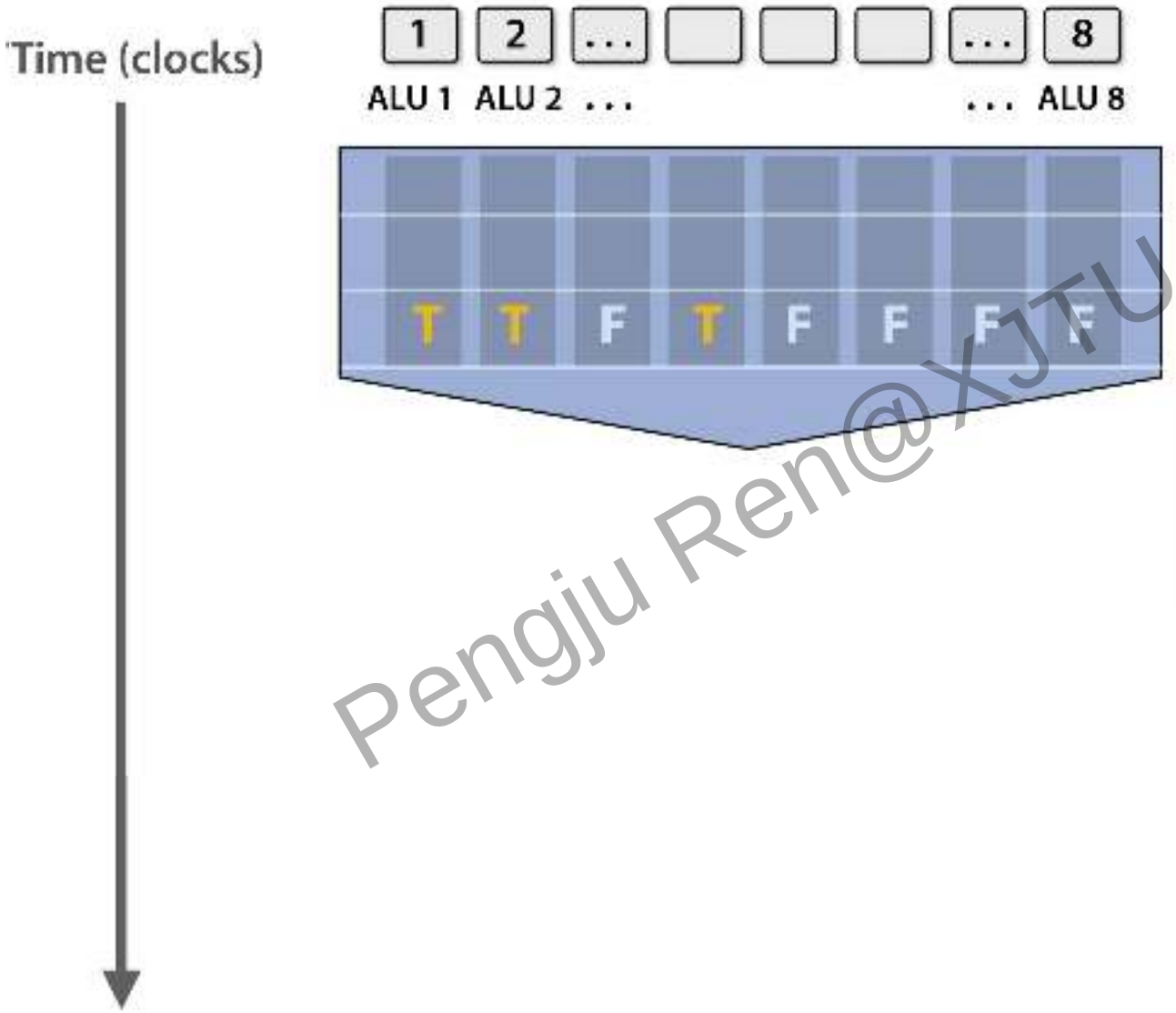
What about conditional execution?



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

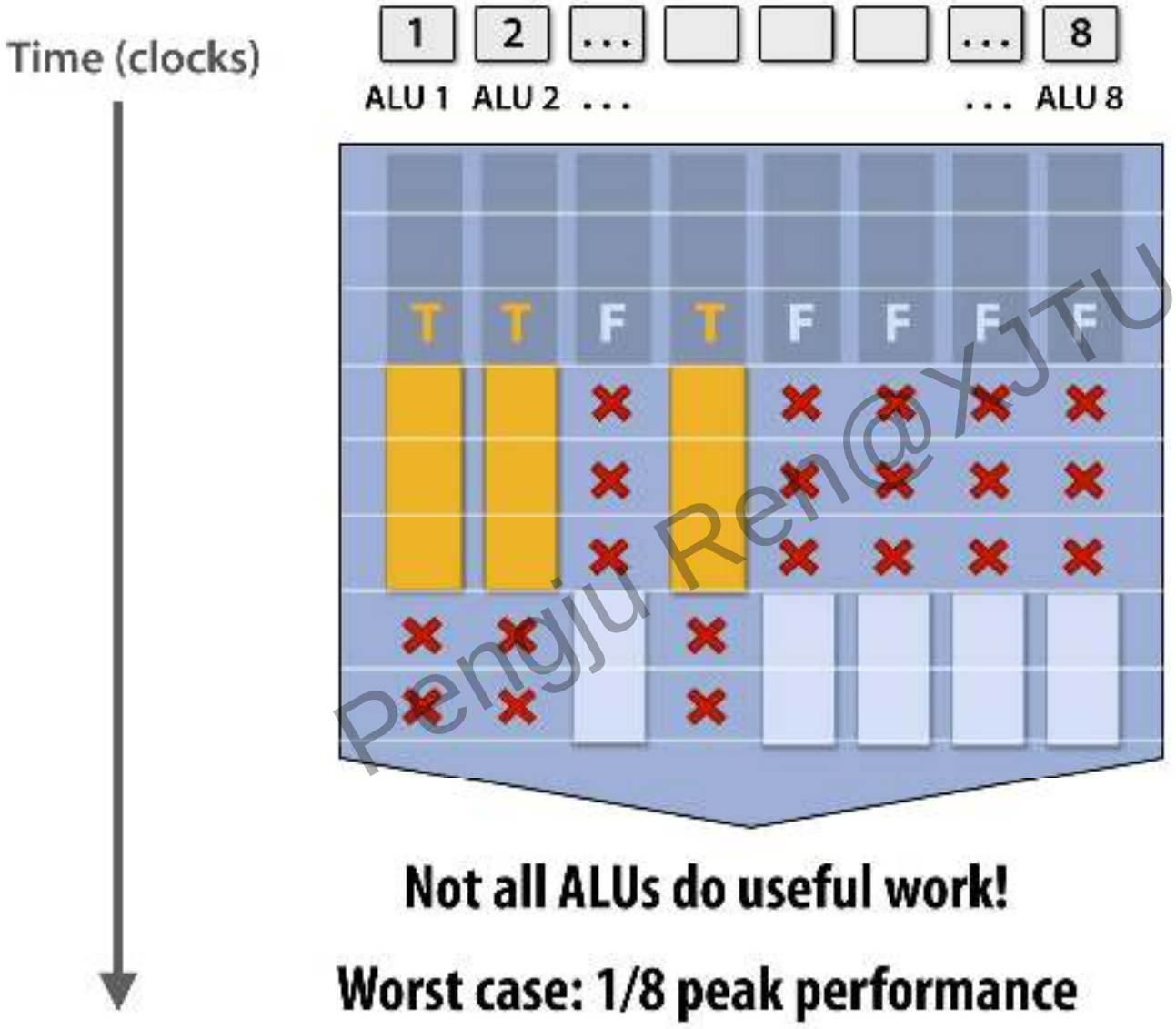
What about conditional execution?



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>


result[i] = x;
```

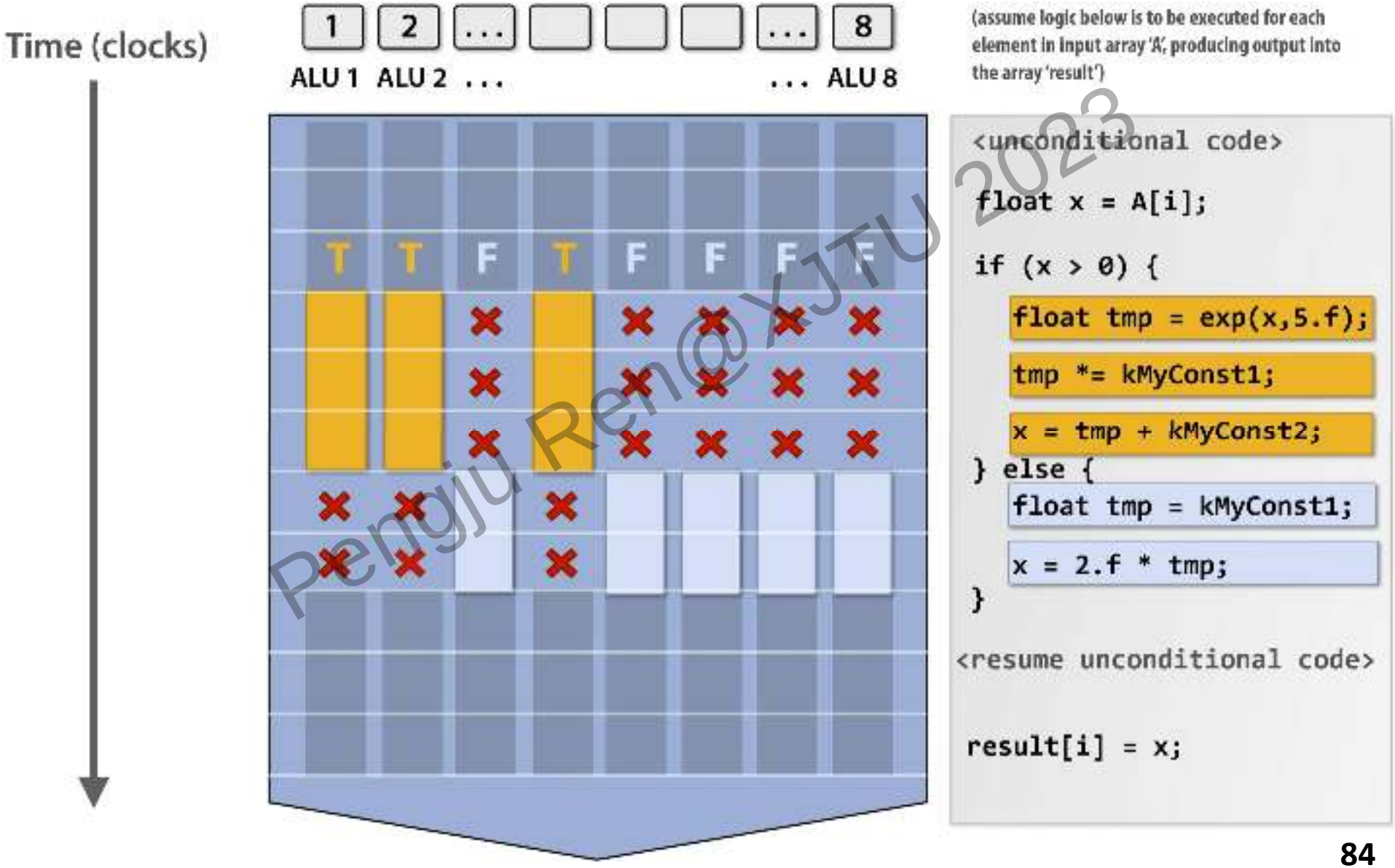
Mask (discard) output of ALU



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

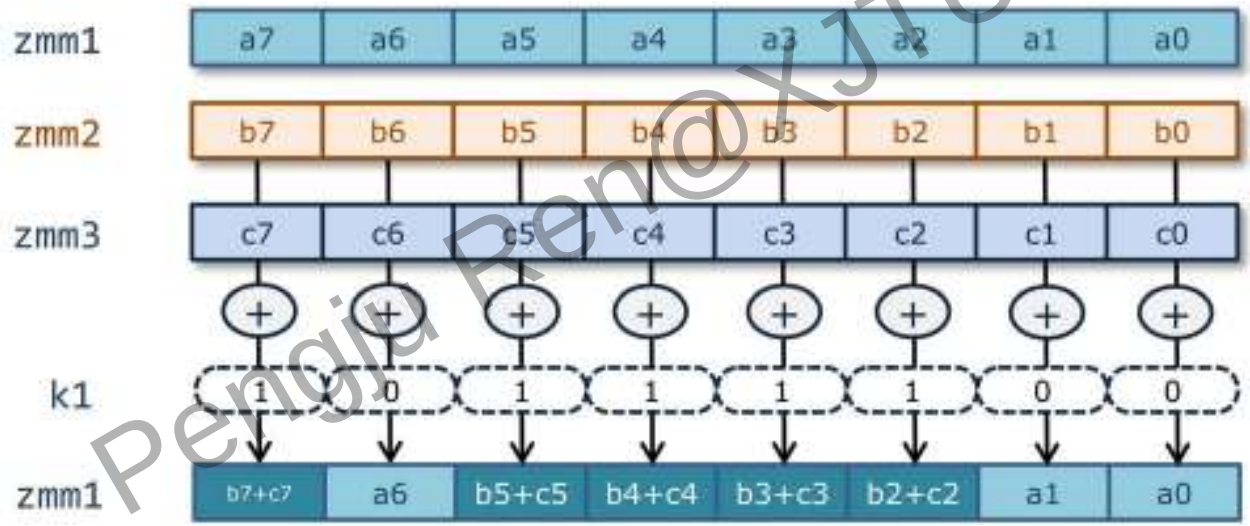
```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

After branch: continue at full performance



Bit Masking of SIMD

- Masking possible to omit operations on selected elements
- Masking supported via dedicated registers (K0-7)
  - Operation + masking in one instruction
  - Does not require additional instruction to mask out elements in a vector



Unmasked elements remain unchanged:

VADDPD zmm1{k1}, zmm2, zmm3

Or zeroed:

VADDPD zmm1{k1}{z}, zmm2, zmm3

Terminology (SIMD)

- Instruction stream coherence ("coherent execution")
  - Same instruction sequence applies to all elements operated upon simultaneously
  - Coherent execution is necessary for efficient use of SIMD processing resources
  - Coherent execution IS NOT necessary for efficient parallelization across cores, since each core has the capability to fetch/decode a different instruction stream
- "Divergent" execution
  - A lack of instruction stream coherence

SIMD execution on modern CPUs

- SSE instructions: 128-bit operations: 4x32 bits or 2x64 bits (4-wide float vectors)
- AVX instructions: 256-bit operations: 8x32 bits or 4x64 bits (8-wide float vectors)
- AVX512 instructions: 512-bit operations: 16x32 bits or 8x64 bits (16-wide float vectors)
- Instructions are generated by the compiler
  - Parallelism explicitly requested by programmer using intrinsics
  - Parallelism conveyed using parallel language semantics (e.g., forall example)
  - Parallelism inferred by dependency analysis of loops (hard problem, even best compilers are not great on arbitrary C/C++ code)
- Terminology: "explicit SIMD": SIMD parallelization is performed at compile time
  - Can inspect program binary and see instructions (vstoreps, vmulps, etc.)

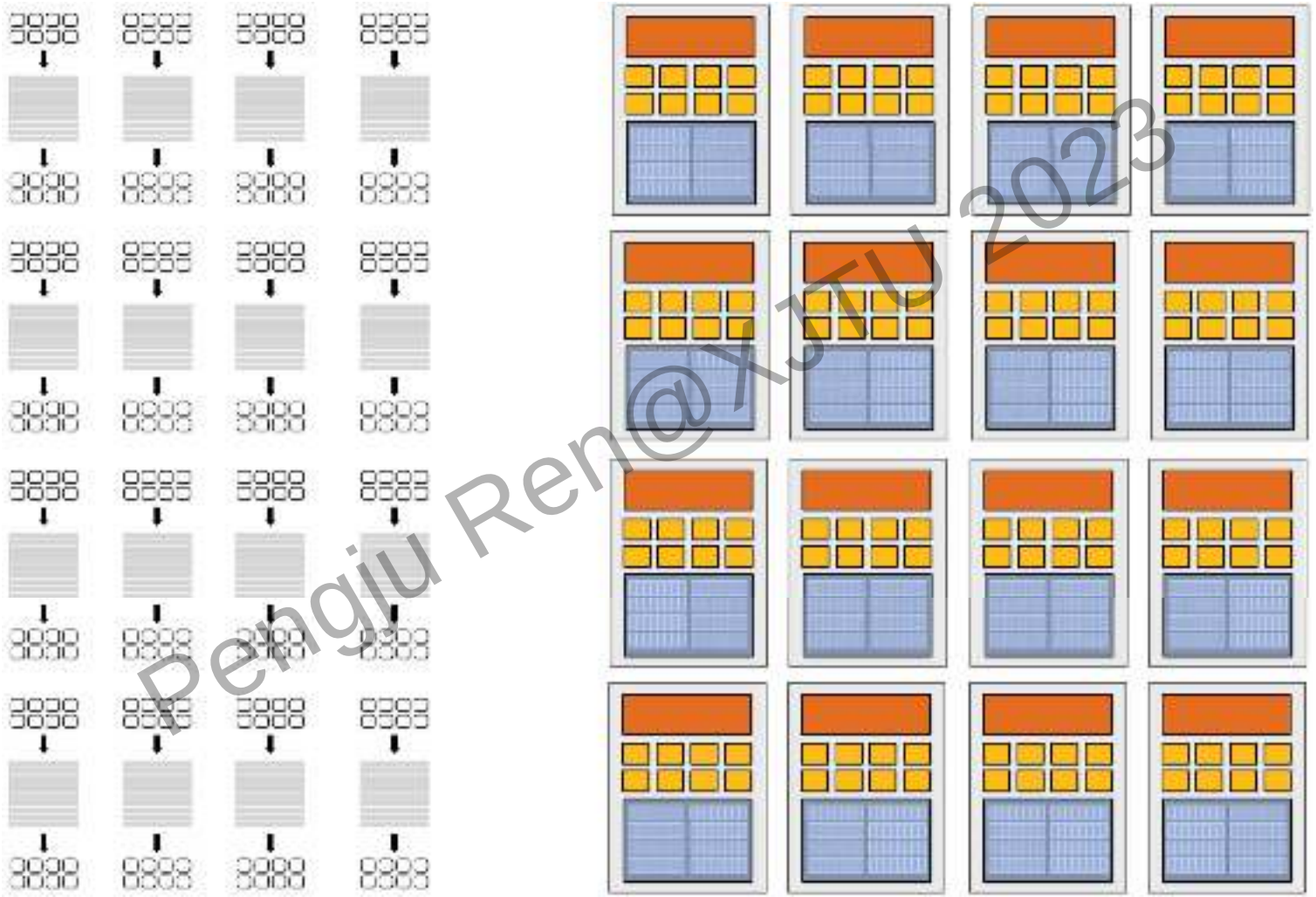
SIMD execution on modern GPUs

- "Implicit SIMD"
  - Compiler generates a scalar binary (scalar instructions)
  - But N instances of the program are *always run* together on the processor

    execute(my_function, N) // execute my_function N times
  - In other words, the interface to the hardware itself is data-parallel
  - Hardware (not compiler) is responsible for simultaneously executing the same instruction from multiple instances on different data on SIMD ALUs
- SIMD width of most modern GPUs ranges from 8 to 32
  - Divergence can be a big issue

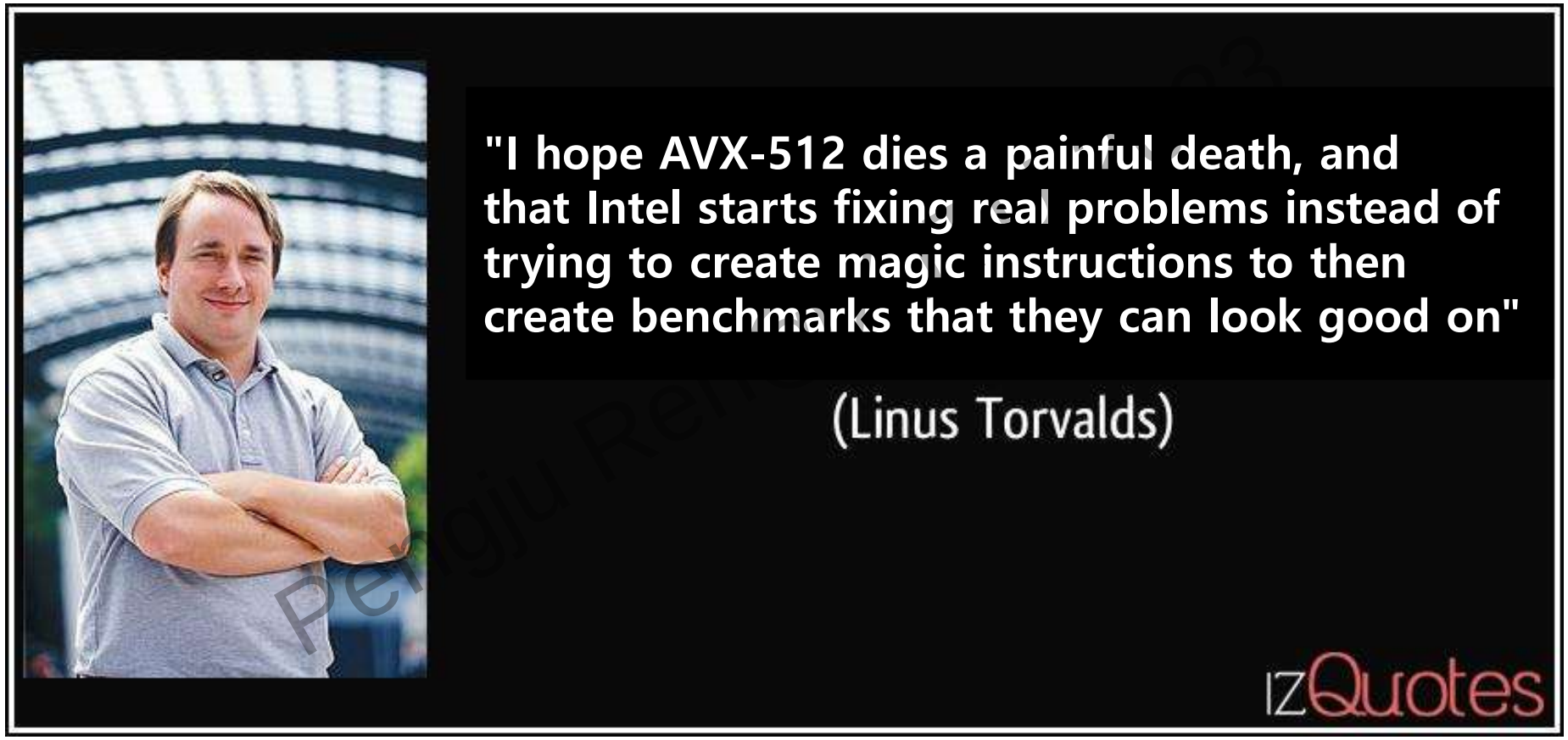
    (poorly written code might execute at 1/8~1/32 the peak capability of the machine!)

SIMD + Multi-core



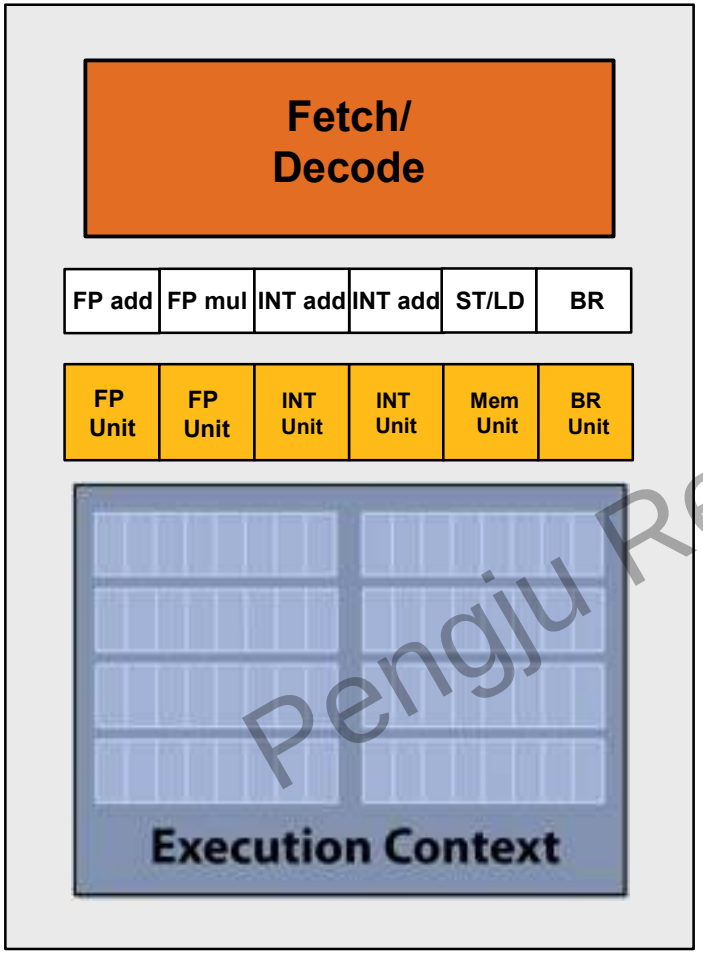
16 cores, 128 ALUs, 16 simultaneous instruction streams

“I hope AVX-512 dies a painful death”



The creator and the principal developer of the Linux kernel

Add more Ops in Single Inst



Idea #3:

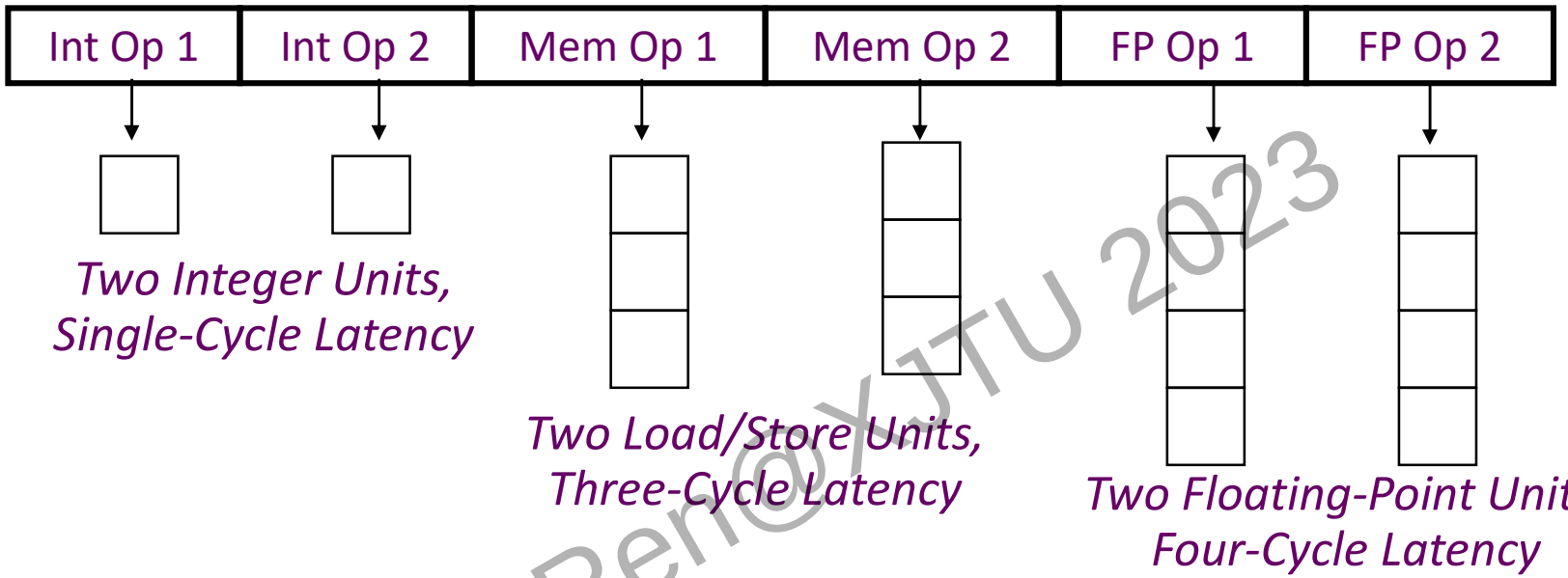
A technique for instruction-level parallelism by executing instructions without dependencies (known at compile-time) in parallel

VILW processing

Multiple operations packed into one instruction (Very long instruction word)

Each operation slot is for a fixed function

VLIW: Very Long Instruction Word



- **Multiple operations packed into one instruction**
- **Each operation slot is for a fixed function**
- **Constant operation latencies are specified**
- **Architecture requires guarantee of:**
  - **Parallelism within an instruction => no cross-operation RAW check**
  - **No data use before data ready => no data interlocks**

VLIW Compiler Responsibilities

- Schedule (reorder) operations to maximize parallel execution
- Guarantees intra-instruction parallelism
- Schedule to avoid data hazards (no interlocks)
  - Typically separates operations with explicit NOPs

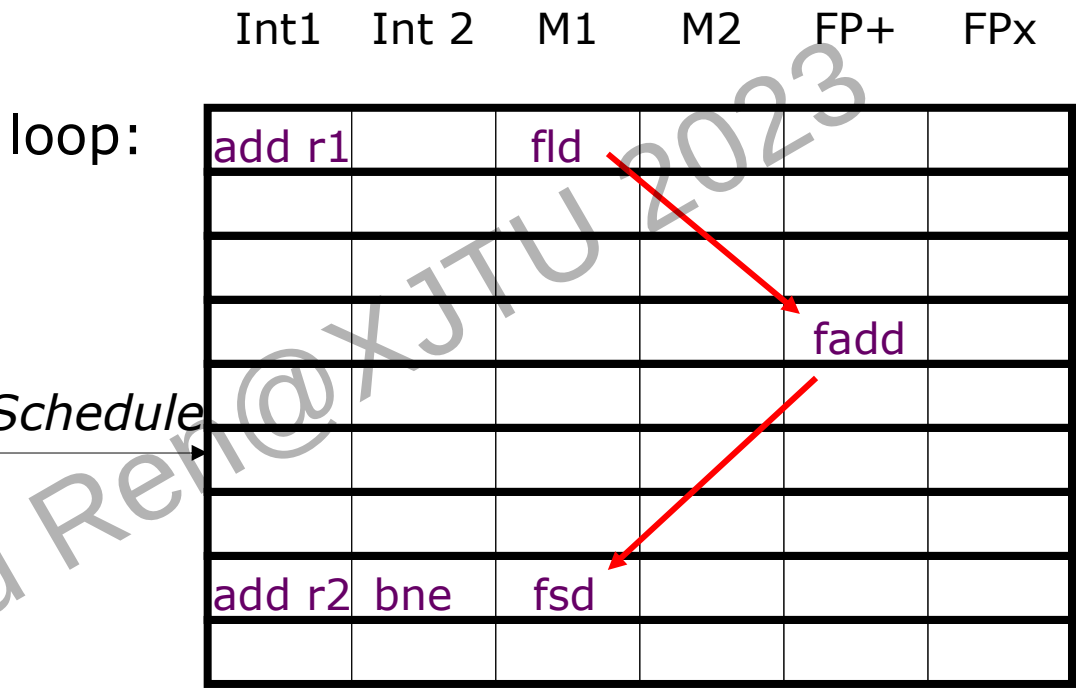
Loop Execution

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

Compile

```
loop:   fld f1, 0(r1)
        add r1, 8
        fadd f2, f0, f1
        fsd f2, 0(r2)
        add r2, 8
        bne r1, r3
loop
```

Schedule



Double Floating Point
r1 A[i], r2 B[i], f2 C, r3 N

How many FP ops/cycle?

1 fadd / 8 cycles = 0.125

Loop Unrolling

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```


Unroll inner loop to perform 4 iterations at once

```
for (i=0; i<N; i+=4)
{
    B[i]   = A[i] + C;
    B[i+1] = A[i+1] + C;
    B[i+2] = A[i+2] + C;
    B[i+3] = A[i+3] + C;
}
```

Is this code correct ?

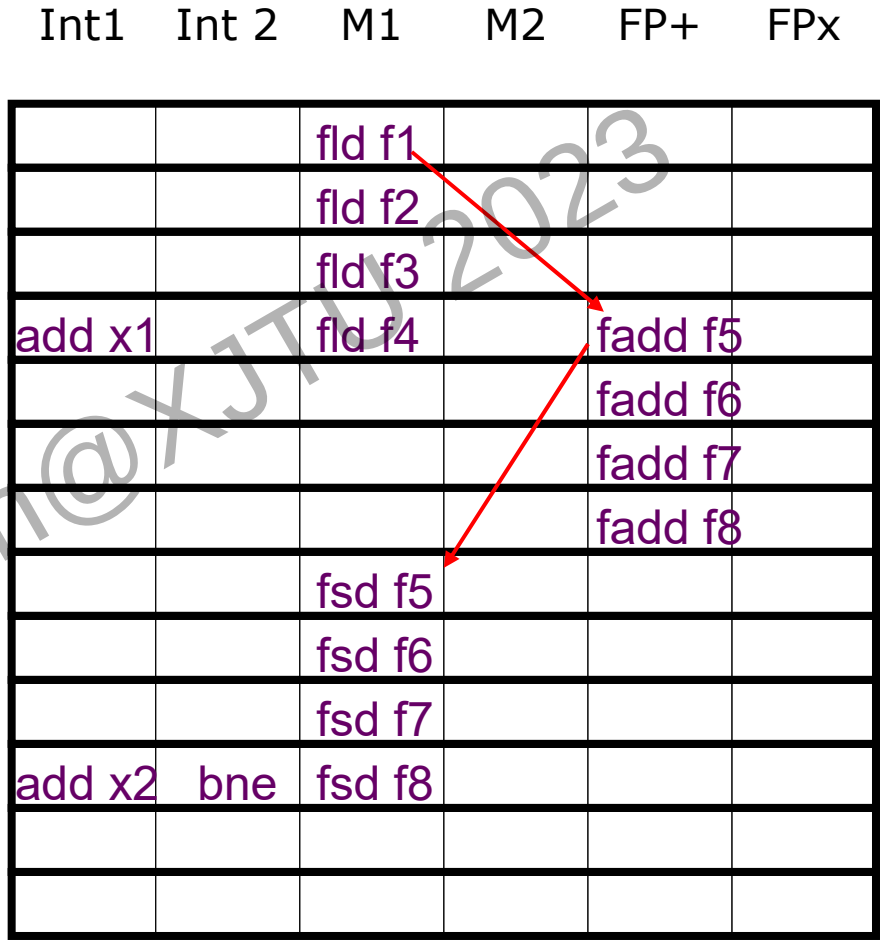
Need to handle values of N that are not multiples of unrolling factor with final cleanup loop

Scheduling Loop Unrolled Code

Unroll 4 ways

```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       fsd f8, 24(x2)
       add x2, 32
       bne x1, x3, loop
```

Schedule



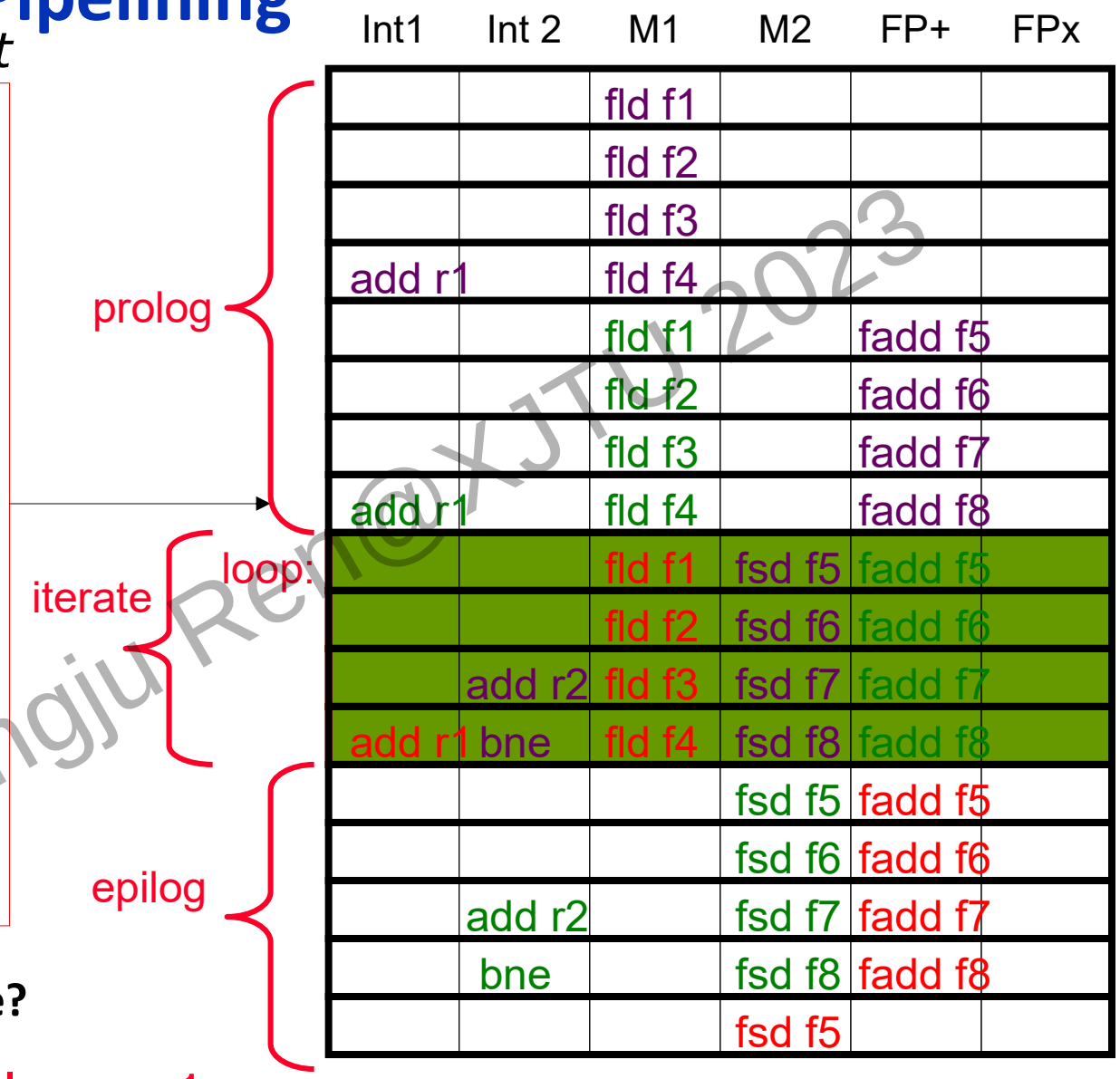
How many FLOPS/cycle?

4 fadds / 11 cycles = 0.36

Software Pipelining

Unroll 4 ways first

```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       add x2, 32
       fsd f8, -8(x2)
       bne x1, x3, loop
```



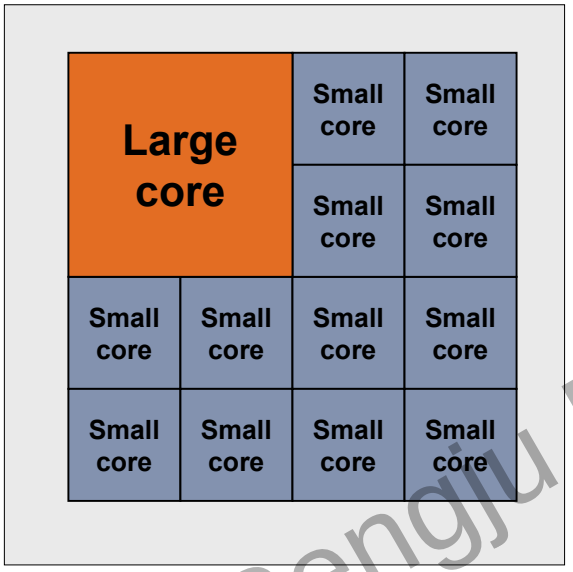
How many FLOPS/cycle?

4 fadds / 4 cycles = 1

Limits of Static Scheduling of VLIW

- Unpredictable branches
- Variable memory latency (unpredictable cache misses)
- Code size explosion
- Compiler complexity
- Despite several attempts, VLIW has failed in general-purpose computing arena (so far).
  - More complex VLIW architectures are close to in-order superscalar in complexity, no real advantage on large complex apps.
- Successful in embedded DSP market
  - Simpler VLIWs with more constrained environment, friendlier code.

Heterogeneity



Idea #4:

Powerful "large" core to speed serialized code (or workload) and wimpy "small" cores to speed parallel code (or workload)

Asymmetric Chip Multiprocessor

Provide small number of large cores and many small cores

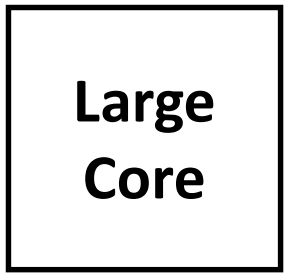
The Problem: Serialized Code Sections

- Many parallel programs cannot be parallelized completely
- Causes of serialized code sections
  - Sequential portions (Amdahl's "serial part")
  - Critical sections
  - Barriers
  - Limiter stages in pipelined programs
- Serialized code sections
  - Reduce performance
  - Limit scalability
  - Waste energy

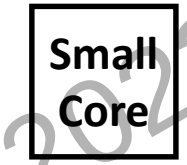
Demands in Different Code Sections

- What we want:
  - In a serialized code section → one powerful “large” core
  - In a parallel code section → many wimpy “small” cores
- These two conflict with each other:
  - If you have a single powerful core, you cannot have many cores
  - A small core is much more energy and area efficient than a large core

“Large” vs. “Small” Cores



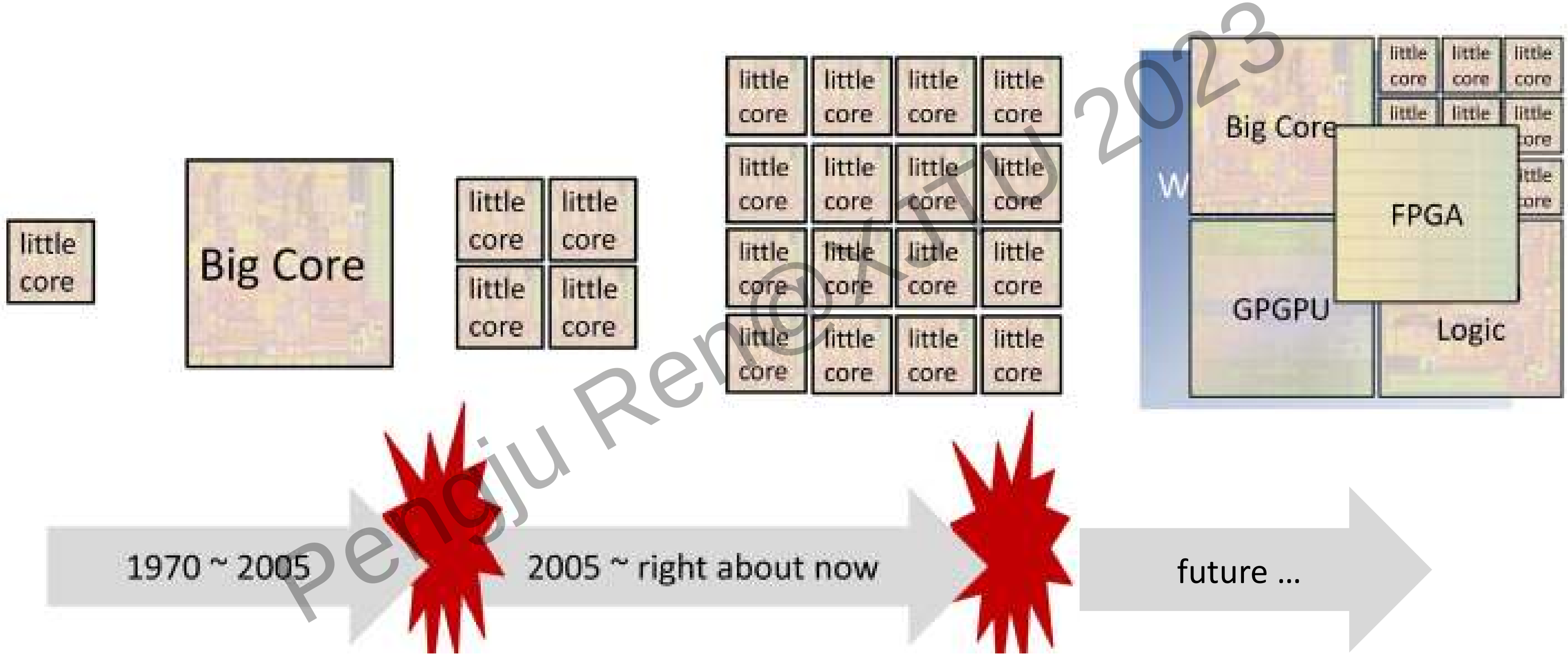
- *Out-of-order*
- *Wide fetch e.g. 4-wide*
- *Deeper pipeline*
- *Aggressive branch predictor (e.g. hybrid)*
- *Multiple functional units*
- *Trace cache*
- *Memory dependence speculation*



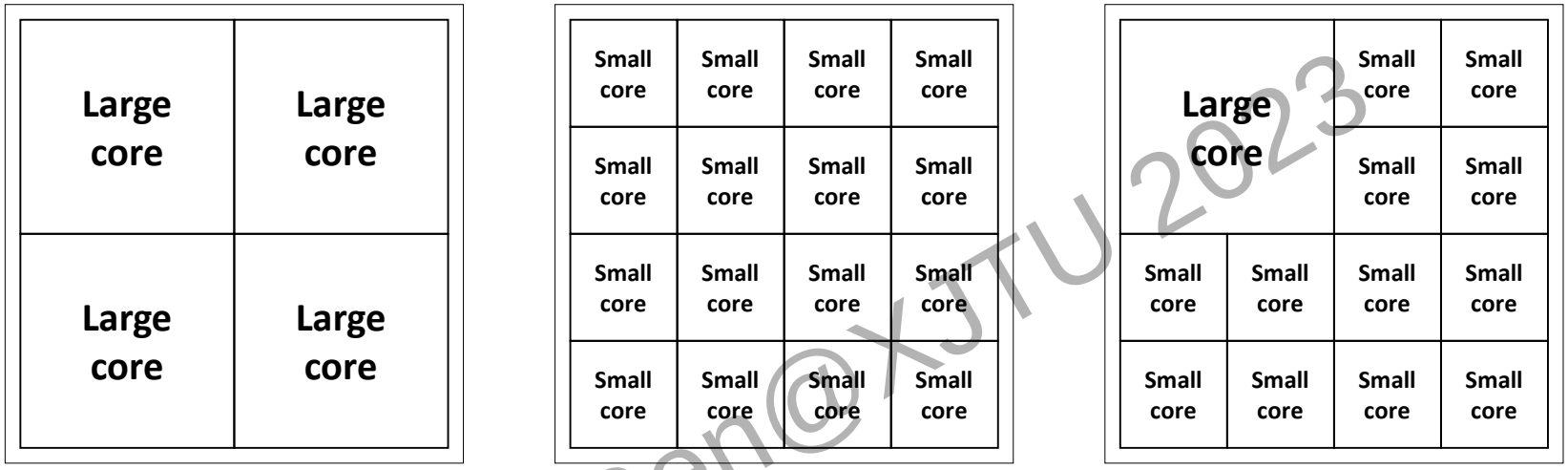
- *In-order*
- *Narrow fetch e.g. 2-wide*
- *Shallow pipeline*
- *Simple branch predictor (e.g. Gshare)*
- *Few functional units*

Large Cores are power inefficient:
e.g., ~2x performance for ~4x area (power)

Moore's Law Scaling with Cores



Asymmetric Chip Multiprocessor (ACMP)



"Tile-Large"（4 units） "Tile-Small"（16 units） ACMP（13 units）

- **Provide one large core and many small cores**
  - + Accelerate serial part using the large core (1 units)
  - + Execute parallel part on small cores and large core for high throughput (12+1 units)

Amdahl's Law Modified

- **Simplified Amdahl's Law for an Asymmetric Multiprocessor**
- **Assumptions:**
  - **Serial portion executed on the large core**
  - **Parallel portion executed on both small cores and large cores**
  - **f: Parallelizable fraction of a program**
  - **L: Number of large processors (L+S=N)**
  - **S: Number of small processors**
  - **X: *Speedup of a large processor over a small one***

$$\text{Speedup} = \frac{1}{\frac{1 - f}{X} + \frac{f}{S + X*L}}$$

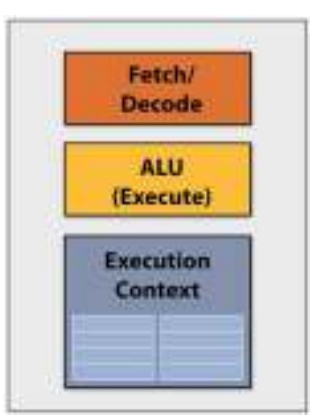
Asymmetric vs. Symmetric Cores

- **Advantages of Asymmetric**
  - + Can provide better performance when thread parallelism is limited
  - + Can be more energy efficient
  - + Schedule computation to the core type that can best execute it
- **Disadvantages**
  - - Need to design more than one type of core.
  - - Scheduling becomes more complicated
    - - What computation should be scheduled on the large core?
    - - Who should decide? HW vs. SW?
  - - Managing locality and load balancing can become difficult if threads move between cores (transparently to software)
  - - Cores have different demands from shared resources

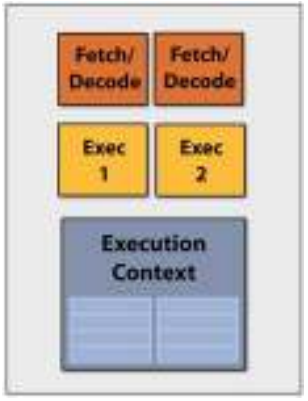
Asymmetric vs. Symmetric Cores

- Priority tasks scheduled on Performance-Cores
- Background tasks scheduled on Efficient-Cores
- Spin loop wait moved from P to E-core

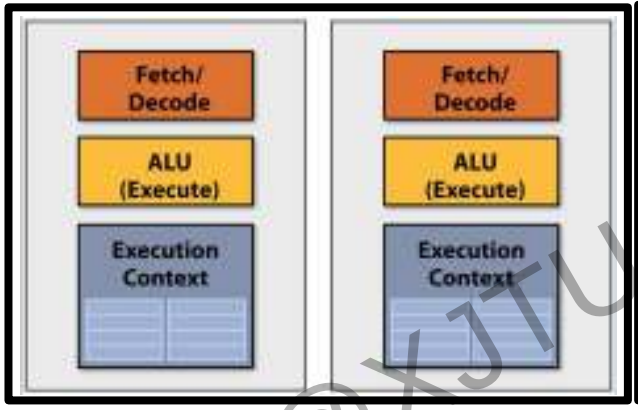
The Revolution of Processor (Conceptionally)



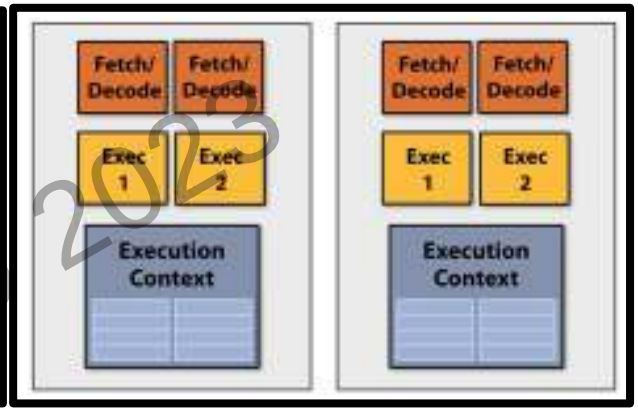
Simple Processor
one instruction per cycle



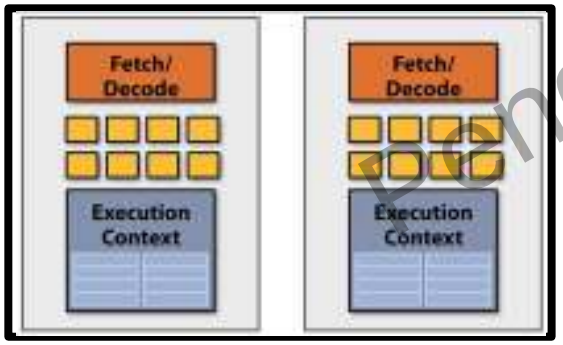
SuperScalar Processor
Two instructions per cycle from a single instruction stream



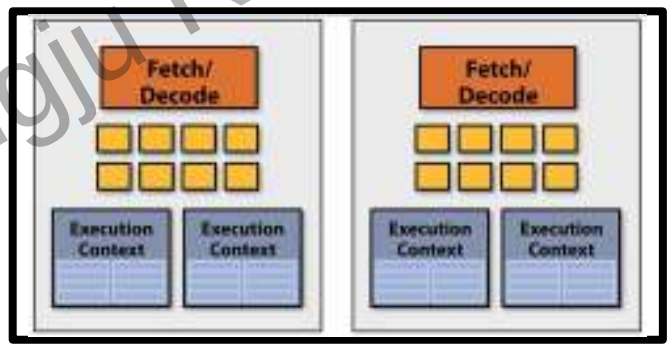
Dual-core Processor
one instruction per cycle from an instruction stream on each core



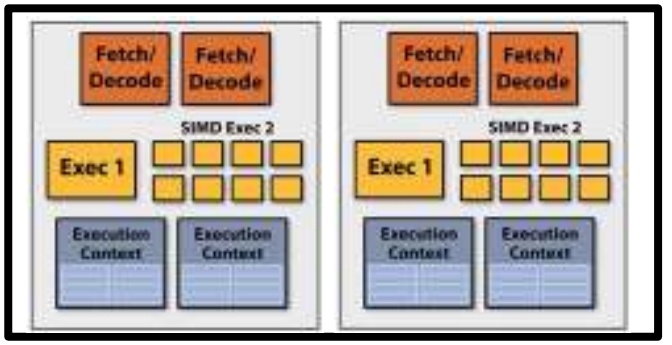
SuperScalar Dual-core Processor
Two instructions per cycle from an instruction stream on each core



SIMD Dual-core Processor
One 8-wide SIMD instruction per cycle from an instruction stream on each core

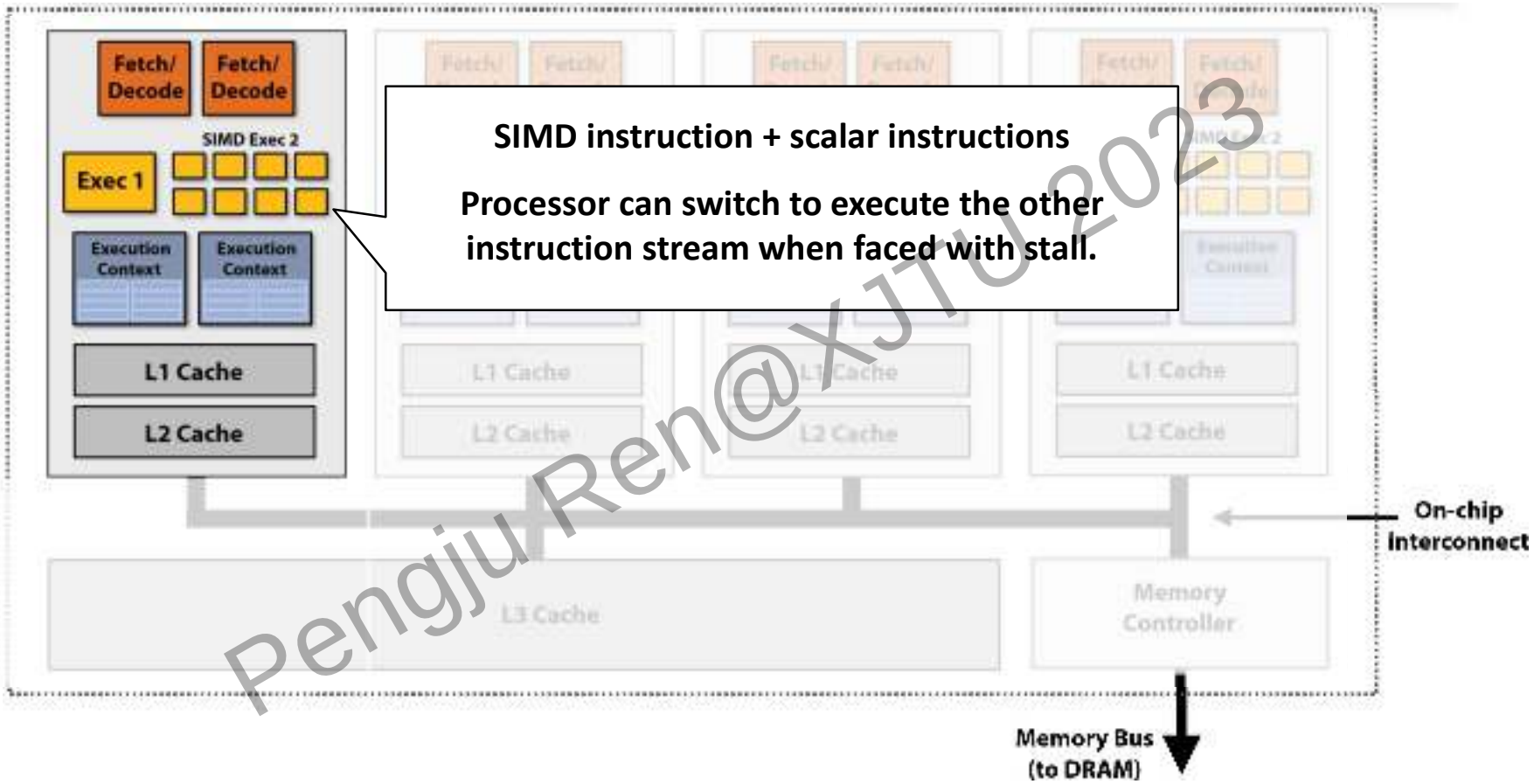


Multi-threaded, SIMD Dual-core Processor
One 8-wide SIMD instruction per cycle from one instruction stream on each core. But can context switch when face long stall



Multi-threaded, SuperScalar, SIMD Dual-core Processor
Two instructions (1 SIMD+1 Scalar) per cycle from one instruction stream on each core. But can context switch when face long stall

Advanced CPU: 4 SuperScalar+SIMD+Multi-threaded cores (Intel Core i7 (10th Gen)



Quad-core Processor: Four cores, two-way multi-threading per core (max eight threads active on chip at once), up to two instructions per clock per core (one of those instructions is 8-wide SIMD)

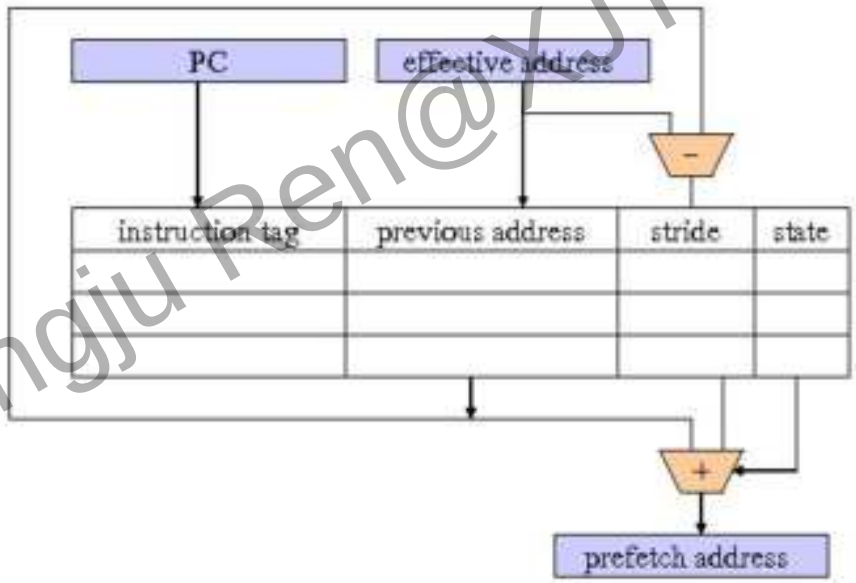
Summary

- Four major ideas that all modern processors employ to varying degrees
  - Employ multiple processing cores (Multicore or Manycore)
    - ➢Simpler cores (embrace thread-level parallelism over instruction-level parallelism)
    - ➢Software decides when to create threads (e.g., via pthreads API)
    - ➢Use multi-threading to make more efficient use of processing resources (hide latencies, fill all available resources, SMT)
  - Amortize instruction stream processing over many ALUs (SIMD)
    - ➢Increase compute capability with little extra cost
    - ➢Vectorization can be done by compiler (explicit SIMD) or at runtime by hardware
  - Accelerate serial part using the large cores, while execute parallel part on small cores (Asymmetric Chip Multiprocessor, big.LITTLE)
  - Superscalar: exploit ILP within an instruction stream. Process different instructions from the same instruction stream in parallel (within a core)
    - ➢Parallelism automatically and dynamically discovered by the hardware during execution (not programmer visible)
- VLIW(DSP): Multiple operations packed into one instruction (Very long instruction word)
  - ➢Parallelism statically scheduled by compiler to maximize parallel execution, while guarantees intra-instruction dependence (poor binary compatibility)
- Due to high arithmetic capability on modern chips, many parallel applications are bandwidth bound

Next Lecture: Understanding Modern Processor: GPGPU Arch & Massively Parallel Programming

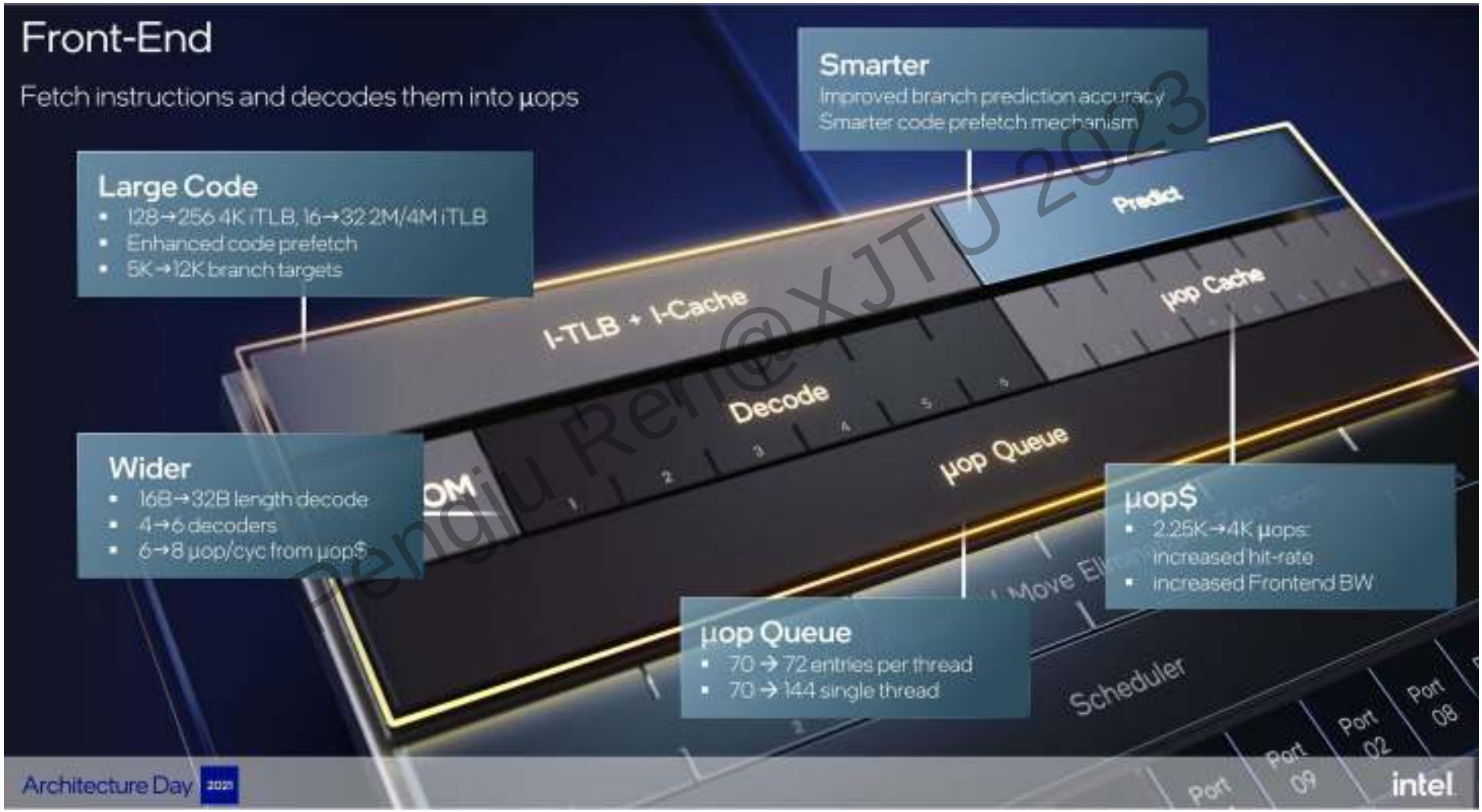
Appendix A: Prefetch instructions

- **Reference Prediction Table (RPT)**
- **Hold information for the most recently used memory instructions to predict their access pattern**



Intel's New Performance x86 Core

Intel's New Performance x86 Core



Intel's New Performance x86 Core

Intel's New Performance x86 Core

Intel's New Performance x86 Core

Best Processors (benchmarks @2022)

Intel i9-12950HX（Laptop)

- Intel 7nm
- 8 P-core/8 E-core
- Total Threads 24
- 14MB L2 Cache/30MB L3 Cache

AMD Ryzen Pro3995WX

- 64 cores /128 threads
- 32MB L2 Cache
- 256 MB L3 Cache

Apple A16 Binoic@4nm

2 P-Cores/4 E-Cores

5 GPU/16 core Neural Engine