

CHAPTER 6

Parallel Processors from Client to Cloud

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University

國立成功大學
National Cheng Kung University

Outline

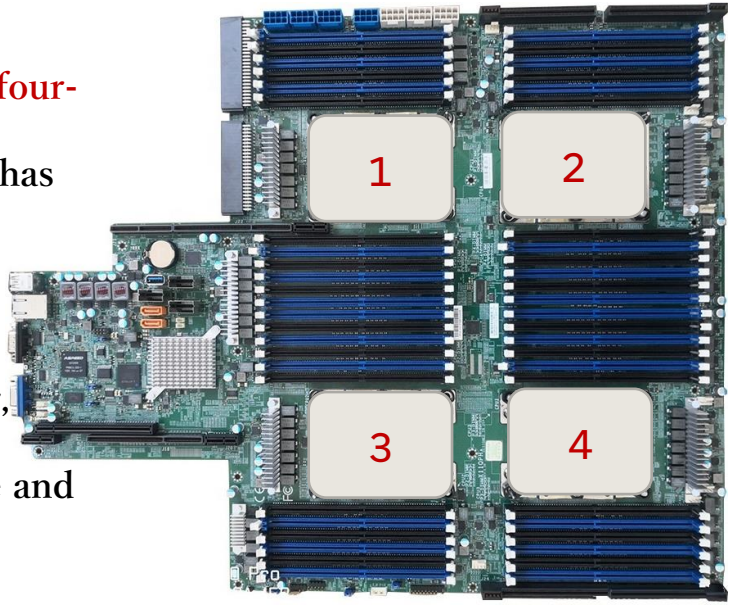
× means not covered in this file

Chapter Goal

- Provide basic knowledge of parallel computing
  - ranging from small devices to large datacenters
- These information is provided from architecture perspective

Introduction

- Connecting multiple computers to get higher performance
  - → Multiprocessors
  - ➢ A computer system with at least two processors (e.g., four-way CPU server in img)
  - ➢ This computer is in contrast to a uniprocessor, which has one, and is increasingly hard to find today
- Power efficiency
  - ➢ Replace large inefficient processors with many smaller, efficient processors
  - ➢ Deliver better performance per Joule both in the large and in the small, if software can efficiently use them
- Availability
  - ➢ If a single processor fails in a multiprocessor with n processors, these systems would continue to provide service with n – 1 processors
- Scalability
  - ➢ Ideally, customers order as many processors as they can afford and receive a commensurate amount of performance
  - ➢ Thus, multiprocessor software must be designed to work with a variable number of processors



Server Motherboard For SuperMicro X11QPH **Four-way CPU Server** LGA3647 Will

Tasks Run on Computers

- High performance can mean greater throughput for *independent tasks*
  - called task-level parallelism or process-level parallelism
  - These tasks are independent single-threaded applications, and
  - they are an important and popular use of multiple processors
  - This approach contrasts with running a single job on multiple processors

- Parallel processing program is used to refer to
  - a single program that runs on multiple processors simultaneously

Parallel Computers

- Scientific problems have needed much faster computers

- These problems can be handled simply today, using a cluster composed of microprocessors housed in many independent servers
  - Clusters can also serve equally demanding applications, e.g., search engines, Web servers, email servers, and databases

- Multiprocessors have been shoved into the spotlight
  - Multicore microprocessors
    - A microprocessor containing multiple processors ("cores") in a single integrated circuit
    - Virtually all microprocessors today in desktops and servers are multicore
  - Multicores are often Shared Memory Processors (SMPs) as they usually share a single physical address space

Terminology

- Hardware part
  - Serial (single-core)
    - E.g., Pentium 4
    - Handle one task at a time
  - Parallel (multicore)
    - E.g., quad-core Xeon e5345
    - Can handle multiple tasks at a time

		Software	
		Sequential	Concurrent
Hardware	Serial	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel	Matrix Multiply written in MATLAB running on an Intel Core i7	Windows Vista Operating System running on an Intel Core i7

FIGURE 6.1 Hardware/software categorization and examples of application perspective on concurrency versus hardware perspective on parallelism

- Software part
  - Sequential: e.g., matrix multiplication
  - Concurrent: e.g., operating system
  - For example, compiler engineers think of them as sequential programs: the steps include parsing, code generation, optimization
  - In contrast, OS developers normally think of them as concurrent programs: cooperating processes handling I/O events due to independent jobs running on a computer
- Sequential and concurrent software can run on serial and parallel hardware
- Challenge: making effective use of parallel hardware
- Use parallel processing program or parallel software to mean either sequential or concurrent software running on *parallel hardware*

Related Concepts in Our Textbook

- § 2.11: Parallelism and Instructions
  - ➢Synchronization
- § 3.6: Parallelism and Computer Arithmetic
  - ➢Subword Parallelism
- § 4.11: Parallelism via Instructions
- § 5.10: Parallelism and Memory Hierarchies
  - ➢Cache Coherence

Difficulty of Creating Parallel Programs

- Parallel software is the problem
  - Challenge: Too few important application programs have been rewritten to complete tasks sooner on multiprocessors
  - It is difficult to write software that uses multiple processors to complete one task faster, and
  - the problem gets worse as the number of processors increases
- Need to get significant performance/energy improvement
  - Otherwise, just use a faster uniprocessor, since it's easier!

- Difficulties
  - Partitioning (dividing a task into sub-tasks for parallel execution)
  - Coordination (orchestrating the execution of sub-tasks)
  - Communications overhead (sending data among sub-tasks)

Example: Speedup Challenge

- Suppose you want to achieve a speed-up of 90 times faster with 100 processors against the sequential version
- What percentage of the original computation can be sequential?

→ Sequential part can limit speedup

- According to Amdahl's Law in Ch. 1

$$\text{Speedup} = \frac{1}{(1 - F_{\text{parallelizable}}) + F_{\text{parallelizable}}/100} = 90$$

➔ Solving: $F_{\text{parallelizable}} = 0.999$
Need sequential part to be 0.1% of original time

Example: Scaling Problem

- Suppose you want to perform a program with two sums (workload):
  1. one is a sum of 10 scalar variables, and
  2. one is a matrix sum of a pair of two-dimensional arrays, with dimensions 10 by 10
- What speed-up do you get with 10 versus 40 processors?

- Assume an addition operation takes time t
  - 10 sequential addition for scalar = 10t
  - 100 additions for matrix summation = 100t
  - It takes 110t on a single processor

- Time on 10 processors
  - 10t+100t/10 = 20t
  - Speedup = 110t/20t = 5.5x
- Time on 40 processors
  - 10t+100t/40 = 12.5t
  - Speedup = 110t/12.5t = 8.8x

- You can try to do the exercise when the problem size of the matrix multiplication grows from 10x10 to 100x100
- Are the speedups getting better when a larger problem size is involved?

- To know more about the impact of the problem sizes on the performance
  - ✓ Please refer to the example in p. 524~525 in our textbook

Performance Scaling

- Comparison: two types of the scaling problem
- Strong scaling
  - Get speed-up on a multiprocessor while keeping the problem size fixed
  - This is harder than the weak scaling
  - As shown in the previous example
- Weak scaling
  - Get good speed-up by increasing the size of the problem
  - This is easier than the strong scaling

  Example:
  - 10 processors, 10 × 10 matrix
  - Time = 10t + (100t/10) = 20t
  - 100 processors, 32 × 32 matrix
  - Time = 10t + (1024/100)t = 20.24t

Categorization of Parallel Hardware

- The categorization in Fig. 6.2 was proposed in 1960s
  - Still useful today
  - Known as Flynn's Taxonomy
- SISD: Single Instruction stream, Single Data stream
  - A conventional uniprocessor has a single instruction stream and single data stream
- MIMD: Multiple Instruction streams, Multiple Data streams
  - A conventional multiprocessor has multiple instruction streams and multiple data streams
- SPMD: Single Program, Multiple Data streams
  - The conventional MIMD *programming* model, where a single program runs across all processors
  - Programmers can write a single program that runs on all processors of a MIMD computer,
  - relying on conditional statements when different processors should execute distinct sections of code

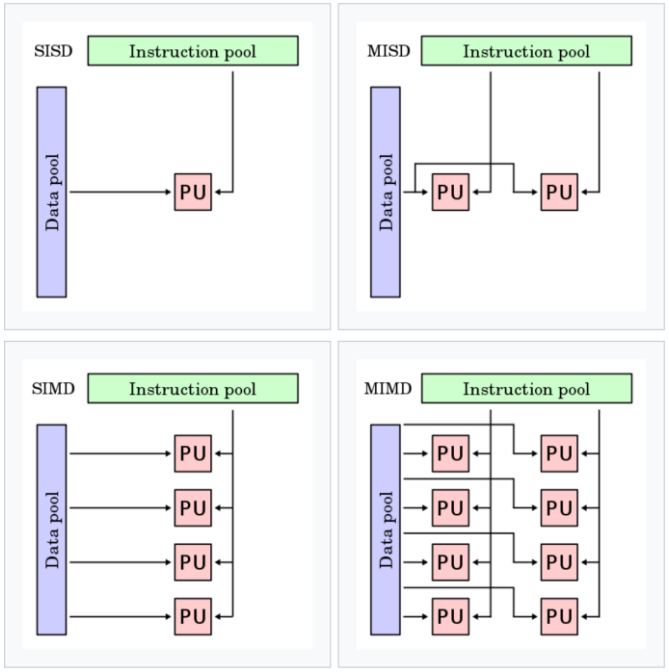
		Data Streams	
		Single	***Multiple***
Inst. Streams	***Single***	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

Important today

FIGURE 6.2 Hardware categorization and examples based on number of instruction streams and data streams: SISD, SIMD, MISD, and MIMD

Categorization of Parallel Hardware (Cont'd)

- SIMD: Single Instruction stream, Multiple Data streams
  - The same instruction is applied to many data streams, as in a vector processor
  - SIMD computers operating on vectors of data are popular today
  - A single SIMD instruction might add 64 numbers by sending 64 data streams to 64 ALUs to form 64 sums within a single clock cycle
  - E.g., SSE/AVX instructions in x86 CPUs
- MISD: multiple instruction streams, single data stream
  - No examples today
  - E.g., a "stream processor" could fit this category
  - The processor would perform a series of computations on a single data stream in a pipelined fashion: parse the input from the network, decrypt the data, decompress it, search for match, and so on



The Flynn's taxonomy from Wikipedia

data-level parallelism: Parallelism achieved by performing the same operation on independent data

		Data Streams	
		Single	***Multiple***
Inst. Streams	***Single***	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

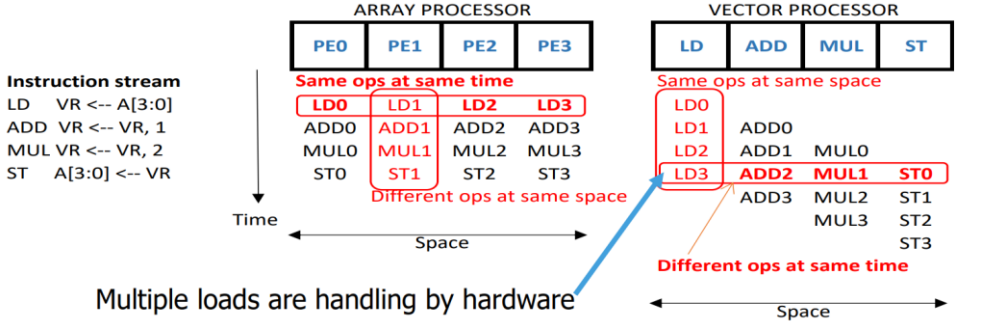
FIGURE 6.2 Hardware categorization and examples based on number of instruction streams and data streams: SISD, SIMD, MISD, and MIMD

Vector Processor (SIMD)

- More elegant interpretation of SIMD is called a vector architecture
  - which has been closely identified with computers designed by Seymour Cray starting in the 1970s
  - A great match to problems with lots of data-level parallelism

- Two variants of SIMD architecture: array processor and vector processor

1. An array processor has 64 ALUs perform 64 additions simultaneously
2. Vector architectures pipelined the ALU to get good performance at lower cost
   - Data collected from memory into registers
   - Stream data from/to vector registers to units
   - Operate data sequentially in registers using pipelined execution units
   - Results stored from registers to memory
   - Key component of the architecture: vector registers
   - E.g., a vector architecture might have 32 vector registers, each with 64 64-bit elements



Courtesy of RISC-V Vector Extension Webinar I, July 13, 2021, Thang Tran, Andes Technology

RISV-V Support for Vectors

- RISC-V offers the vector extension V with vector instructions and vector registers
- Vector instructions
  - vsetvli x0, x0, e64: the vector elements are 64 bits long
  - vfadd.vv: adds two double-precision floating-point vectors after the execution of the vsetvli inst above
- Operations could be a vector–vector operation or a vector–scalar operation
  - vfmul.vf: a vector–scalar floating-point multiply
  - vle.v, vse.v: vector load, vector store (data length is determined by the vsetvli inst.

Example

- RISC-V Vector (RVV) Extension
  - The same names as RISC-V operations but with the prefix "V" appended
  - v0 to v31: 32 × 64-element registers, (64-bit elements)
- Significantly reduces instruction-fetch bandwidth

Example: DAXPY (Y = a × X + Y)

- The conventional RISC-V code vs. the vector RISC-V code
  for so called DAXPY: Y = a × X + Y
  where X and Y are vectors of 64 double-precision floating-point numbers, initially resident in memory, and a is a scalar double precision variable
- Starting addresses of X and Y are in x19 and x20

NOTE: This example is the so-called DAXPY loop that forms the inner loop of the Linpack benchmark; DAXPY stands for double precision a × X plus Y

- Total instruction counts:
  - ✓ 500 for conventional version vs. 7 for vector version
- Fewer instructions for vector version
  - ✓ The vector operations work on 64 elements at a time
  - ✓ The overhead instructions that constitute nearly half the loop on RISC-V are not present in the vector code
- Three pipeline hazards (three arrows on the left code)
  - ✓ fadd.d waits for fmul.d
  - ✓ fsd waits for fadd.d
  - ✓ fadd.d and fmul.d wait for fld
- On the vector processor, each vector instruction will only stall for the first element in each vector
  - ✓ then subsequent elements will flow smoothly down the pipeline

- Conventional RISC-V code:

```
      fld     f0,a(x3)      // load scalar a
      addi    x5,x19,512    // end of array X
loop: fld     f1,0(x19)     // load x[i]
      fmul.d  f1,f1,f0      // a * x[i]
      fld     f2,0(x20)     // load y[i]
      fadd.d  f2,f2,f1      // a * x[i] + y[i]
      fsd     f2,0(x20)     // store y[i]
      addi    x19,x19,8     // increment index to x
      addi    x20,x20,8     // increment index to y
      bltu    x19,x5,loop   // repeat if not done
```

- Vector RISC-V code:

```
fld f0, a(x3)          // load scalar a
vsetvli x0, x0, e64    // 64-bit-wide elements
vle.v v0, 0(x19)       // load vector x
vfmul.vf v0, v0, f0    // vector-scalar multiply
vle.v v1, 0(x20)       // load vector y
vfadd.vv v1, v1, v0    // vector-vector add
vse.v v1, 0(x20)       // store vector y
```

Vector and Scalar Architectures

- A single vector instruction specifies a great deal of work
  - It is equivalent to executing an entire loop
  - It dramatically reduces the instruction fetch/decode bandwidth
- Vector operations enjoys data-level parallelism
  - Simplify the hardware design without checking for data hazards
  - The computation of each result in the vector is independent of the computation of other results in the same vector
- For data-parallel applications, it is easier to write vector programs for vector architectures
  - Harder when programming for MIMD multiprocessors
- If the data handled by vector units are adjacent
  - the cost of the memory access latency is seen only once for the entire vector, rather than once for each word of the vector
- No control hazards arise from the loop branches
  - as the complete loop is replaced by vector instructions
- Vector architectures
  - save instruction bandwidth and hazard checking
  - use memory bandwidth efficiently
  - has energy efficiency against scalar architectures

Vector Lanes

- Vector lane
  - One or more vector functional units and a portion of the vector register file
  - Inspired by lanes on highways that increase traffic speed, multiple lanes execute vector operations simultaneously

- Improve vector performance by using *parallel pipelines* to execute a vector instruction
  - Fig. 6.3(a) executes 1 addition/cycle
  - Fig. 6.3(b) runs 4 additions/cycle
- The structure of a four-lane vector architecture is in Fig. 6.4
  - Each lane has an FP add, an FP multiply and a load-store unit
  - Each lane operates as a vector processor

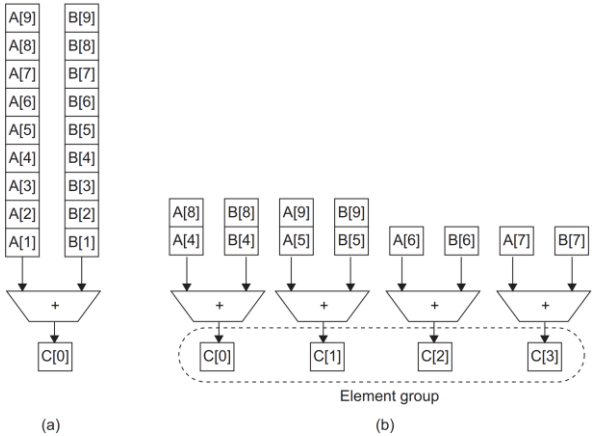


FIGURE 6.3 Using multiple functional units to improve the performance of a single vector add instruction, C = A + B. The vector processor (a) on the left has a single add pipeline and can complete one addition per cycle. The vector processor (b) on the right has four add pipelines or lanes and can complete four additions per cycle. The elements within a single vector add instruction are interleaved across the four lanes

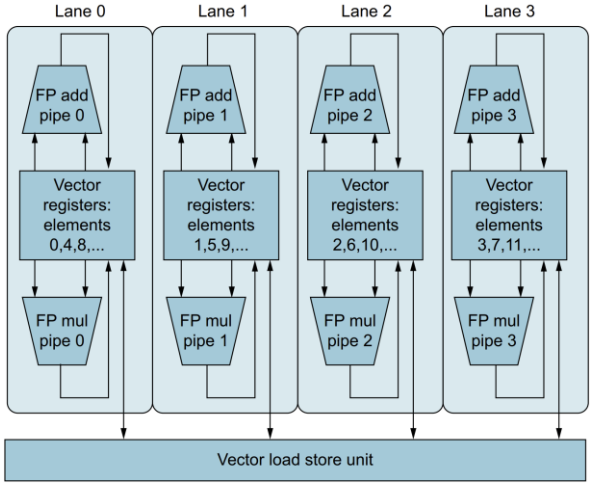


FIGURE 6.4 Structure of a vector unit containing four lanes. The vector-register storage is divided across the lanes, with each lane holding every fourth element of each vector register. The figure shows three vector functional units: an FP add, an FP multiply, and a load-store unit. Each of the vector arithmetic units contains four execution pipelines, one per lane, which acts in concert to complete a single vector instruction. Note how each section of the vector-register file only needs to provide enough read and write ports (see Chapter 4) for functional units local to its lane

Hardware Multithreading

- Hardware multithreading is a related concept to MIMD
  - ➢especially from the programmer's perspective
- MIMD relies on multiple processes or threads to try to keep many processors busy
- Hardware multithreading allows multiple threads to share a single processor

  - ➢That is, multiple threads share the functional units of a single processor in an overlapping fashion to try to utilize the hardware resources efficiently
- To permit this sharing, the processor must duplicate the independent state of each thread
  - ➢For example, each thread would have a separate copy of the register file and the program counter
  - ➢Besides, the hardware must support the ability to change to a different thread relatively quickly

Image source

Approaches for Hardware Multithreading

- Fine-grained multithreading
  - switches between threads on each instruction,
  - resulting in interleaved execution of multiple threads
  - Interleaving done in a round-robin fashion, skipping any threads that are stalled at that clock cycle

- Advantage
  - it can hide the throughput losses that arise from both short and long stalls,
  - since instructions from other threads can be executed when one thread stalls
- Disadvantage
  - it slows down the execution of the individual threads,
  - since a thread that is ready to execute without stalls will be delayed by instructions from other threads

Approaches for Hardware Multithreading II

- Coarse-grain multithreading
  - ➢was invented as an alternative to fine-grained multithreading
  - ➢switches threads only on expensive stalls, such as last-level cache misses

- Advantage (compared with the fine-grain)
  - ➢It relieves the need to have thread switching be extremely fast and
  - ➢is unlikely to slow down the execution of an individual thread,
  - ➢since instructions from other threads will only be issued when a thread encounters a costly stall
- Disadvantage
  - ➢it is limited in its ability to overcome throughput losses, especially from shorter stalls
  - ➢The limitation arises from the pipeline startup overhead
  - ➢i.e., need to flush the entire pipeline when a stall occurs
  - ➢Good for reducing the cost of high-cost stalls

Approaches for Hardware Multithreading III

- Simultaneous multithreading (SMT)
  - ➢is a variation on hardware multithreading
  - ➢that uses the resources of a multiple-issue, dynamically scheduled pipelined processor to exploit thread-level parallelism
  - ➢at the same time, it exploits instruction-level parallelism
- Insight of SMT
  - ➢Multiple-issue processors often have more functional unit parallelism available than most single threads can effectively use
- With register renaming and dynamic scheduling (Ch.4),
  - ➢multiple instructions from independent threads can be issued without regard to the dependences among them
  - ➢since the resolution of the dependences can be handled by the dynamic scheduling capability
  - →SMT is *always* executing instructions from multiple threads
  - →Does not switch resources every cycle

Variants of Hardware Multithreading

- Top portion
  - Four threads would execute independently on a superscalar with no multithreading support
  - Each runs alone on the processor

- The bottom portion
  - Four threads could be run together on the processor more efficiently using three multithreading variants
    1. Coarse-grain scheme
    2. Fine-grain scheme
    3. SMT

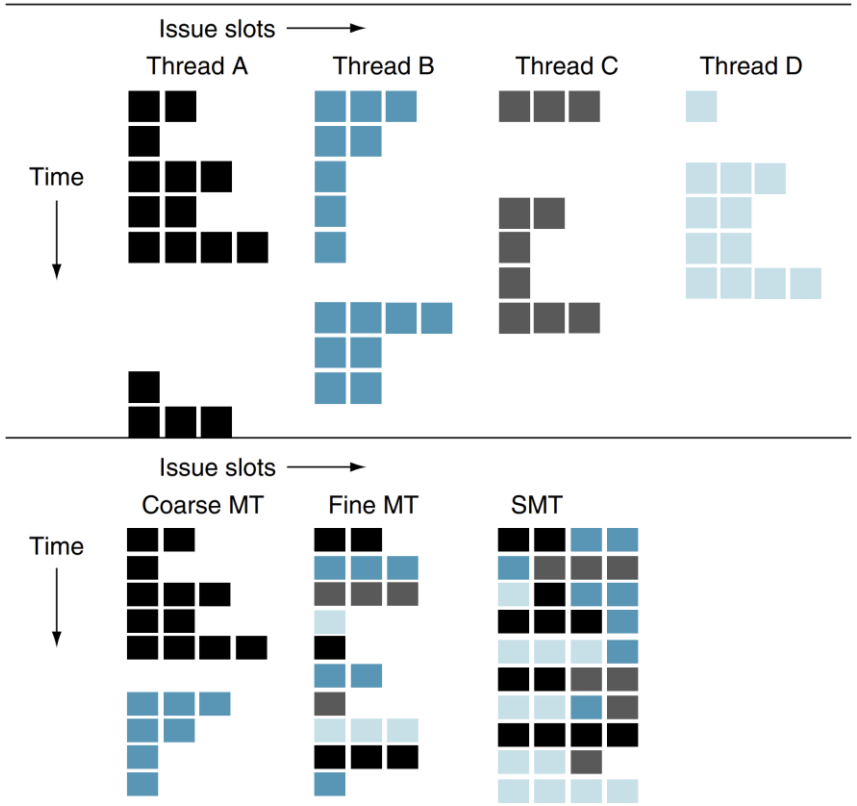


FIGURE 6.5 How four threads use the issue slots of a superscalar processor in different approaches. The four threads at the top show how each would execute running alone on a standard superscalar processor without multithreading support. The three examples at the bottom show how they would execute running together in three multithreading options.
The horizontal dimension represents the instruction issue capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding issue slot is unused in that clock cycle. The shades of gray and color correspond to four different threads in the multithreading processors. The additional pipeline start-up effects for coarse multithreading, which are not illustrated in this figure, would lead to further loss in throughput for coarse multithreading

Comparison of Hardware Multithreading

- Without hardware multithreading
  - The use of issue slots is limited by a lack of instruction-level parallelism (ILP)
  - In addition, a major stall, such as an instruction cache miss, can leave the entire processor idle
- Coarse-grain scheme
  - The long stalls are partially hidden by switching to another thread that uses the resources of the processor
  - Although this reduces the number of completely idle clock cycles, the pipeline start-up overhead still leads to idle cycles, and
  - limitations to ILP mean all issue slots will not be used
- Fine-grain scheme
  - The interleaving of threads mostly eliminates idle clock cycles
  - Because only a single thread issues instructions in a given clock cycle,
  - however, limitations in ILP still lead to idle slots within some clock cycles
- SMT
  - Thread-level parallelism and ILP are both exploited, with multiple threads using the issue slots in a single clock cycle
  - Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads

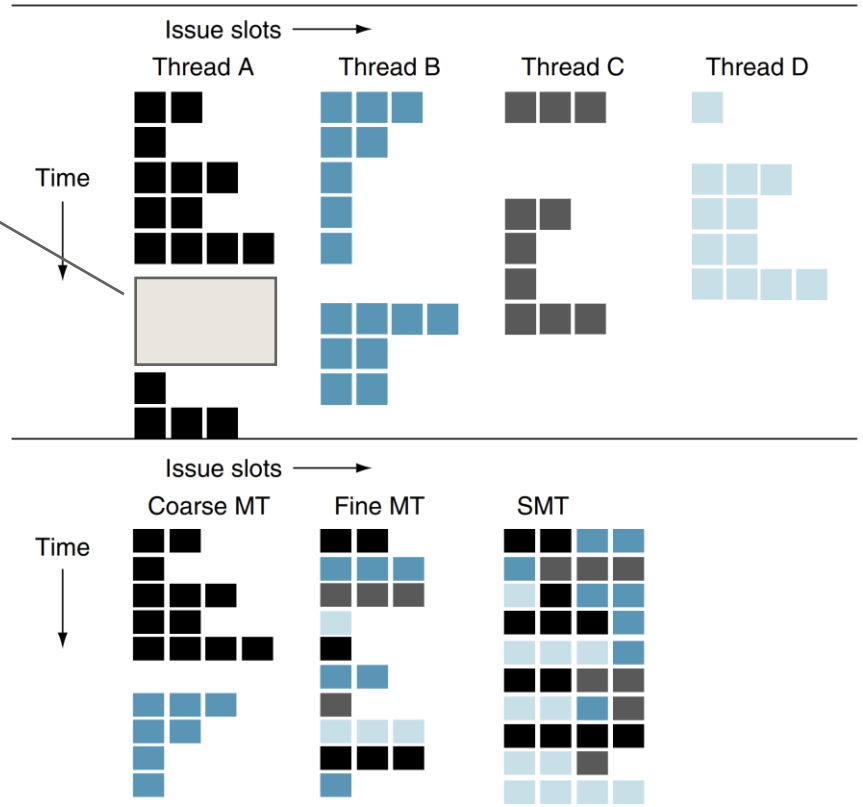


FIGURE 6.5 How four threads use the issue slots of a superscalar processor in different approaches. The four threads at the top show how each would execute running alone on a standard superscalar processor without multithreading support. The three examples at the bottom show how they would execute running together in three multithreading options.
The horizontal dimension represents the instruction issue capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding issue slot is unused in that clock cycle. The shades of gray and color correspond to four different threads in the multithreading processors. The additional pipeline start-up effects for coarse multithreading, which are not illustrated in this figure, would lead to further loss in throughput for coarse multithreading

Shared Memory Architecture

- Challenge:
  - How to efficiently programming the increasing number of processors per chip, given the difficulty of rewriting old programs to run well on parallel hardware?

- One answer was to provide a single physical address space that all processors can share,
  - so that programs need not concern themselves with where their data are, merely that programs may be executed in parallel
  - In this approach, all variables of a program can be made available at any time to any processor

- The alternative is to have a separate address space per processor that requires that sharing must be explicit
  - ✓ For detail information, please refer to Sec. 6.8

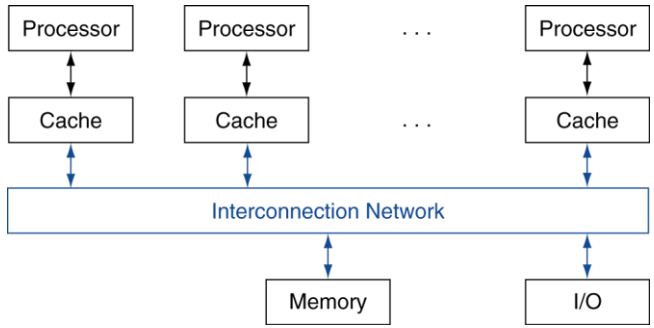


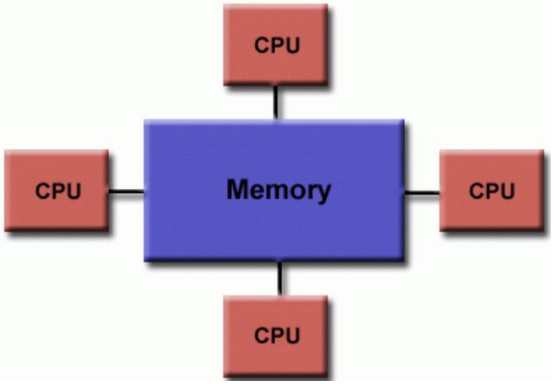
FIGURE 6.7 Classic organization of a shared memory multiprocessor.

Shared Memory Multiprocessors

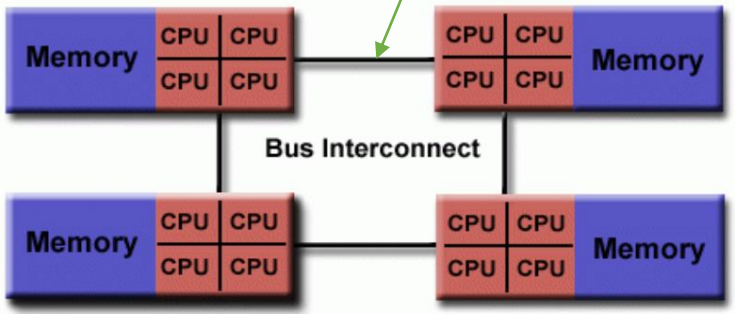
- Shared memory multiprocessor (SMP)
  - A multiprocessor offers the programmer a single physical address space across all processors (see Fig. 6.7)
  - Processors communicate through shared variables in memory, with all processors capable of accessing any memory location via loads and stores
- Multicore chips nearly always implement the shared memory architecture
  - Also called shared-address multiprocessor

Two SMP Styles

- UMA
  - Identical processors
  - Equal access and access times to memory
  - Sometimes called CC-UMA - Cache Coherent UMA
    - Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update
    - Cache coherency is accomplished at the hardware level



- NUMA
  - Often made by physically linking two or more SMPs
  - One SMP can directly access memory of another SMP
  - Not all processors have equal access time to all memories
    - Memory access across link is slower
  - If cache coherency is maintained, then may also be called CC-NUMA (Cache Coherent NUMA)



Synchronization

- Coordination is necessary while multiple processors (processes) run in parallel accessing shared data
  - otherwise, one processor could start working on data before another is finished with it
- This coordination is called synchronization (Ch.2)
  - When sharing is supported with a single address space, there must be a separate mechanism for synchronization
  - It is efficient to have hardware assistance for synchronization

- Lock is one of the approaches when accessing to a shared variable
  - Only one processor at a time can acquire the lock, and
  - other processors interested in shared data must wait until the original processor unlocks the variable
  - Refer to Sec. 2.11 for the instructions for locking in RISC-V

Example: Parallel Sum on SMP (1/2)

- To get summation of 64,000 numbers on 64 CPUs w/ UMA
  - Each processor has processor identifier: $0 \leq Pn \leq 63$
  - Partition 1000 numbers per processor to have a balanced load
  - Allocating the subset data to different memory space is unnecessary thanks to the shared memory architecture

- Initial summation on processor Pn

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i += 1)
  sum[Pn] += A[i];
```

- Reduction: divide and conquer
- Need to add these partial sums obtained by 64 CPUs
  - Half the processors add pairs, then quarter the processors, etc.
  - Need to synchronize between reduction steps

Example: Parallel Sum on SMP (2/2)

- The hierarchical steps for the reduction sum in Fig. 6.8
  - ➢Three steps to compute the summation of eight numbers
- In "half=2" step
  - ➢P0 = P0+ P2 should wait for the updates of P0 and P2 in "half=4" step
  - ➢Otherwise, the old values of P0 and P2 will be used
  - →Synchronization
    synch(); in the code below is a barrier to wait for all processors

- The parallel reduction code of the code in previous page

```
half = 64;
do
 synch();
 if (half%2 != 0 && Pn == 0)
  sum[0] += sum[half-1];
  /* Conditional sum needed when half is odd;
     Processor0 gets missing element */
 half = half/2; /* dividing line on who sums */
 if (Pn < half) sum[Pn] += sum[Pn+half];
while (half > 1);
```

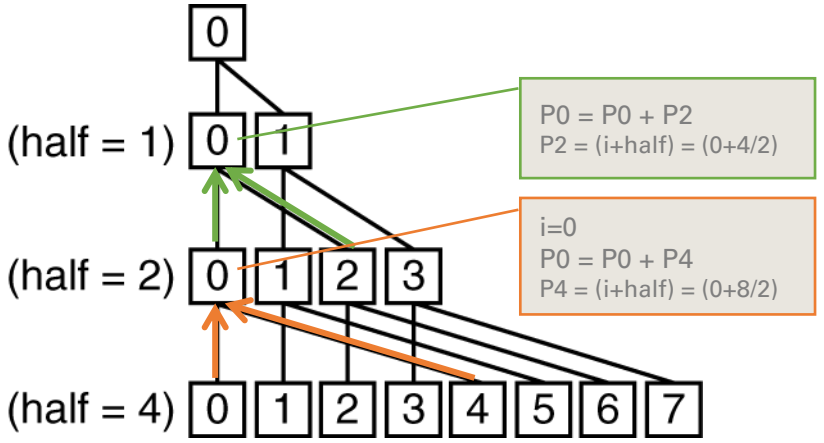


FIGURE 6.8 The last four levels of a reduction that sums results from each processor, from bottom to top. For all processors whose number i is less than half, add the sum produced by processor number (i + half) to its sum

Share Nothing Infrastructure

- Each processor has its own private physical address
  - The alternative approach to sharing an address space
- Fig. 6.14 shows the classic organization of a multiprocessor with multiple private address spaces
- Multiprocessors must communicate via explicit message passing (e.g., MPI)
  - which traditionally is the name of such style of computers
- Provided the system has routines to send and receive messages (provided by programming facilities)
  - Coordination is built in with message passing,
  - since one processor knows when a message is sent, &
  - the receiving processor knows when a message arrives

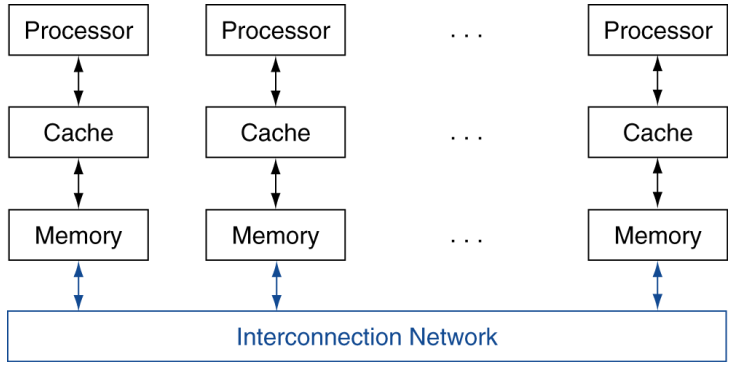


FIGURE 6.14 Classic organization of a multiprocessor with multiple private address spaces, traditionally called a message-passing multiprocessor. Note that unlike the SMP in Figure 6.7, the interconnection network is not between the caches and memory but is instead between processor-memory nodes

Multiprocessor Network Topologies

- Clusters require local area networks to connect servers together
  - Multicore chips require on-chip networks to connect cores together, and
- Network costs include
  - the number of switches, the number of links on a switch to connect to the network,
  - the width (number of bits) per link, and
  - length of the links when the network is mapped into silicon

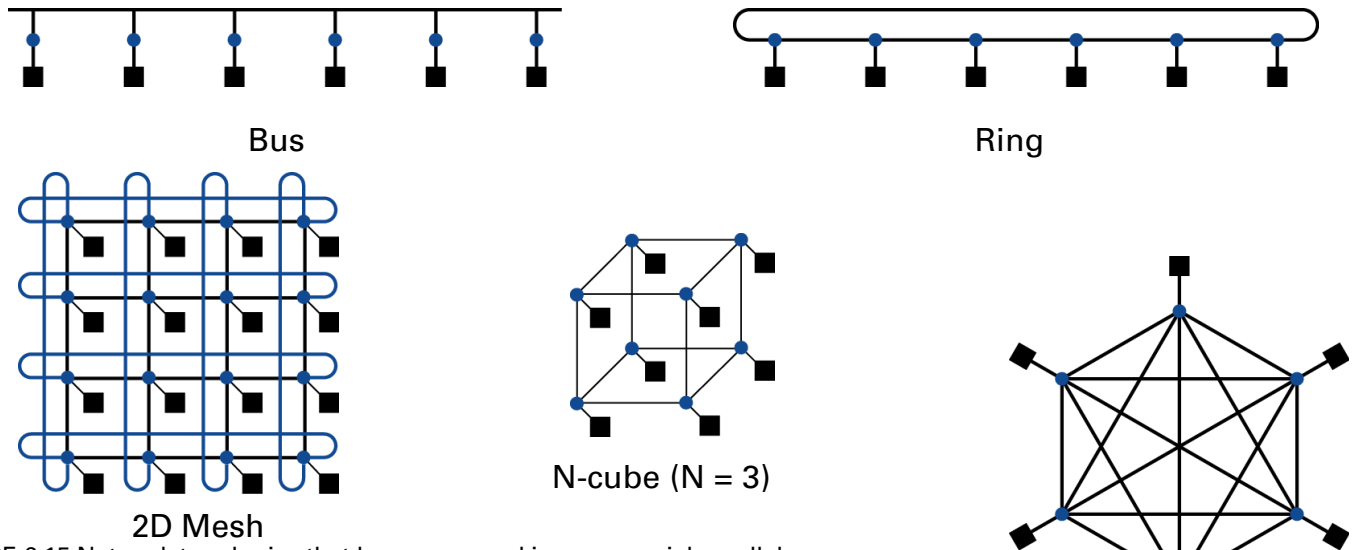


FIGURE 6.15 Network topologies that have appeared in commercial parallel processors. The colored circles represent switches and the black squares represent processor-memory nodes. Even though a switch has many links, generally only one goes to the processor. The Boolean n-cube topology is an n-dimensional interconnect with 2n nodes, requiring n links per switch (plus one for the processor) and thus n nearest-neighbor nodes. Frequently, these basic topologies have been supplemented with extra arcs to improve performance and reliability

Multiprocessor Network Topologies II

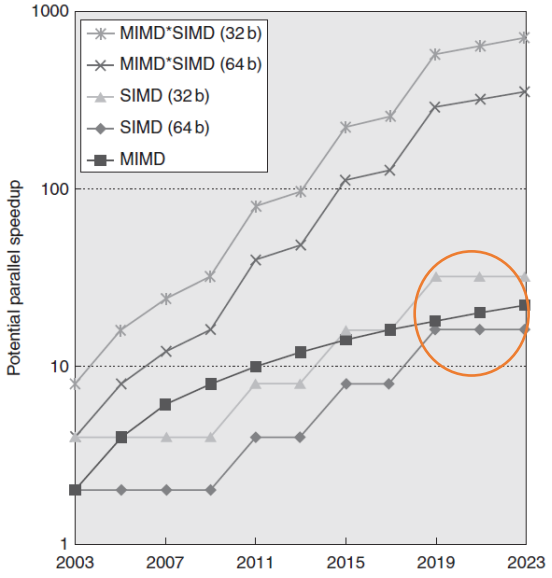
- Network performance includes
  - the latency on an unloaded network to send and receive a message,
  - the throughput in terms of the maximum number of messages that can be transmitted in a given time period,
    - Network bandwidth: the peak transfer rate of a network; can refer to the speed of a single link or the collective transfer rate of all links in the network
  - delays caused by contention for a portion of the network, and
  - variable performance depending on the pattern of communication
- Another obligation of the network may be fault tolerance,
  - since systems may be required to operate in the presence of broken components
- Finally, in this era of energy-limited systems,
  - the energy efficiency of different organizations may trump other concerns

Concluding Remarks

- Information technology industry has now tied its future to parallel computing

- Goal: higher performance by using multiple processors
- Difficulties
  - ➢Developing parallel software
  - ➢Devising appropriate architectures
- Performance evaluation for both mobile and WSC
  - ➢Performance per dollar and
  - ➢performance per Joule

- SIMD and vector operations are a good match to multimedia applications
- They share the advantage of being easier for the programmer than classic parallel MIMD programming and being more energy-efficient than MIMD
  - ✓ The data suggest that SIMD can achieve similar performance as MIMD does

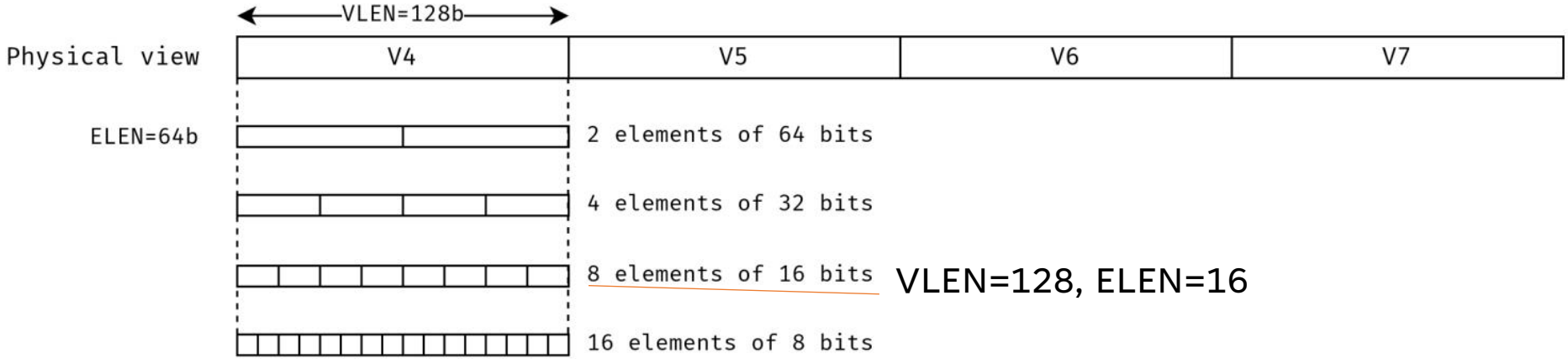


Questions?

RVV Parameters and Basic State

- RVV defines 32 **vector registers** of size VLEN bits
  - `v0` to `v31`
  - VLEN is a **constant parameter** chosen by the implementor and must be a power of two
    - Zv* standard extensions constraint VLEN to be at least 64 or 128
  - E.g., VLEN=512 would be equivalent in size to Intel AVX-512
  - VLEN is not a great name so read it as "vector register size (in bits)"
- Vectors in RVV are divided in **elements**.
  - The size of elements in bits is at least 8 bits up to ELEN bits
  - ELEN is a **constant parameter** chosen by the implementor
  - Must be a power of two and 8 ≤ ELEN ≤ VLEN
    - Zv* standard extensions constrain ELEN to be at least 32 or 64

Example (1)

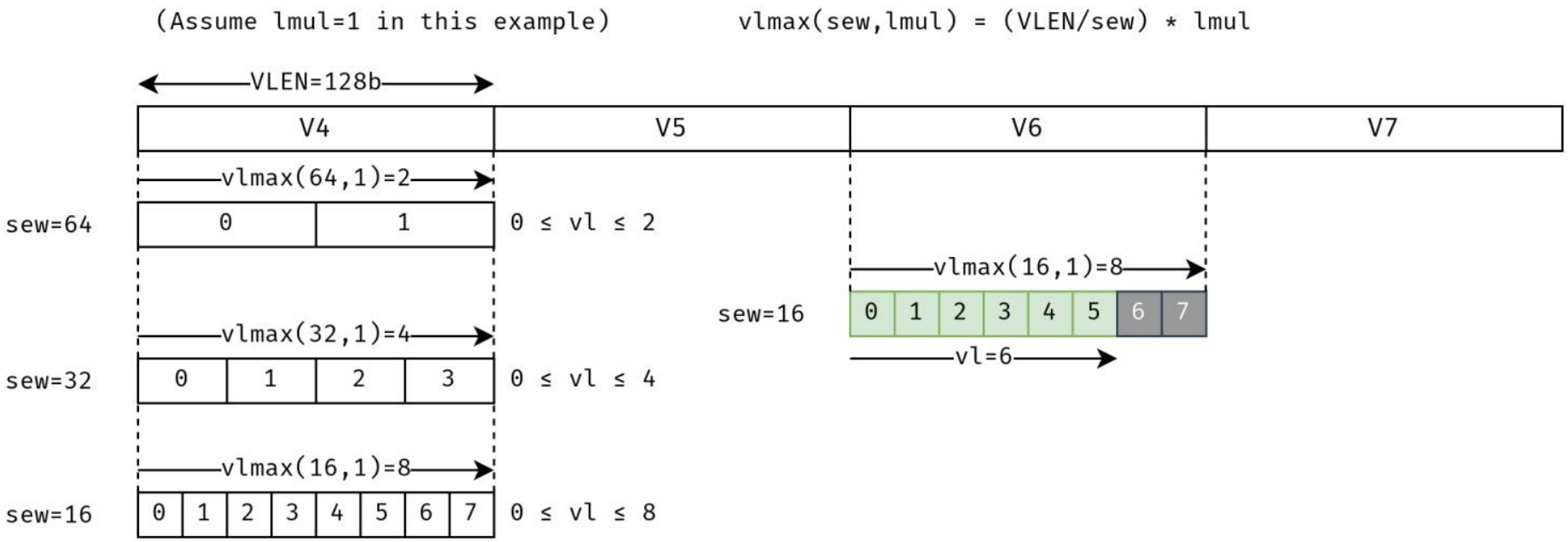


VLEN=128, ELEN=16

RVV Operational State

- There are two **registers** used when operating vectors in RVV
  - **`vtype`**: Vector Type.
  - **`vl`**: Vector Length (not to be confused with VLEN!)
- `vtype` describes **the type of vector we are going to operate** and includes
  - **`sew`**: Standard Element Width. Size in bits of the elements being operated
    - $8 \leq sew \leq ELEN$
  - **`lmul`**: Length Multiplier. Allows grouping registers (more on this later!)
    - $lmul = 2^k$ where $-3 \leq k \leq 3$ (i.e., $lmul \in \{1/8, 1/4, 1/2, 1, 2, 4, 8\}$)
- `vl` describes **how many elements of the vector** (starting from the element zero) **we are going to operate**
  - $0 \leq vl \leq vlmax(sew, lmul)$
  - $vlmax(sew, lmul) = (VLEN / sew) \times lmul$

Example (2)


(Assume lmul=1 in this example)
vlmax(sew,lmul) = (VLEN/sew) * lmul
VLEN=128b
V4
V5
V6
V7
vlmax(64,1)=2
sew=64
0
1
0 ≤ vl ≤ 2
vlmax(32,1)=4
sew=32
0
1
2
3
0 ≤ vl ≤ 4
vlmax(16,1)=8
sew=16
0
1
2
3
4
5
6
7
0 ≤ vl ≤ 8
vlmax(16,1)=8
sew=16
0
1
2
3
4
5
6
7
vl=6

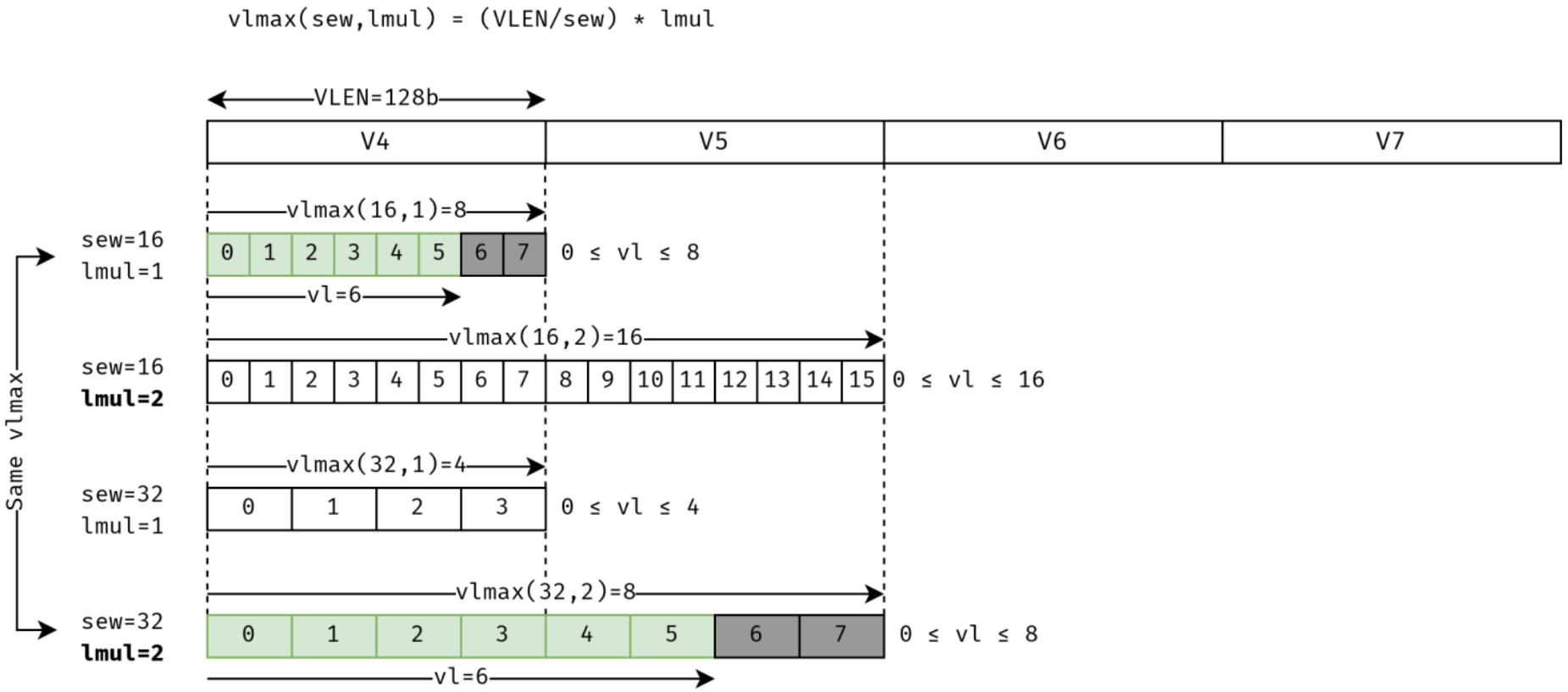
Mixed element sizes

- Vectors with a smaller element size can fit a larger **number of elements**
  - And the opposite: a vector with elements of size ELEN bits can fit the smallest number of elements
- When operating with vectors whose elements are of different size, we have different number of elements
  - This causes problems to algorithms, which want to operate with the same number of elements
- We can "harmonise" the number of elements when the element size is different by either
  - Not using the whole vector register for the small element sizes
  - Use more than one vector register for the large element sizes

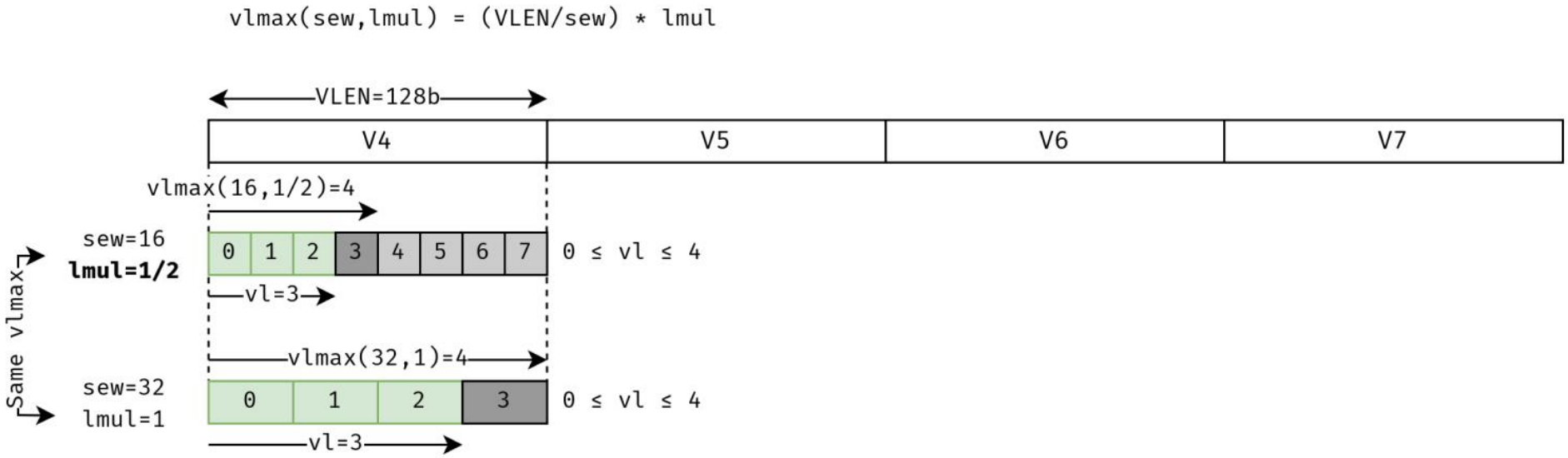
Length multiplier

- RVV allows the two scenarios via the length multiplier
- When `lmul = 1` we can operate up to all the elements of a vector register
- When `lmul < 1` we can operate up to a fraction of all the elements of a vector register `lmul` ∈ {1/2, 1/4, 1/8}
- When `lmul > 1` the operation uses a **vector group** of `lmul` vector registers
  - A vector group "gangs" several vector registers. The vector group is identified by the smallest numbered vector register in the group.
  - 16 vector groups of `lmul = 2`
    - `v0, v2, v4, v6, v8, v10, v12, v14, v16, ..., v28, v30`
  - 8 vector groups of `lmul = 4`
    - `v0, v4, v8, v12, v16, v20, v24, v28`
  - 4 vector groups of `lmul = 8`
    - `v0, v4, v8, v16`

Example (3)


vlmax(sew,lmul) = (VLEN/sew) * lmul
VLEN=128b
V4
V5
V6
V7
Same vlmax
sew=16
lmul=1
vlmax(16,1)=8
0 1 2 3 4 5 6 7
0 ≤ vl ≤ 8
vl=6
sew=16
lmul=2
vlmax(16,2)=16
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 ≤ vl ≤ 16
sew=32
lmul=1
vlmax(32,1)=4
0 1 2 3
0 ≤ vl ≤ 4
sew=32
lmul=2
vlmax(32,2)=8
0 1 2 3 4 5 6 7
0 ≤ vl ≤ 8
vl=6

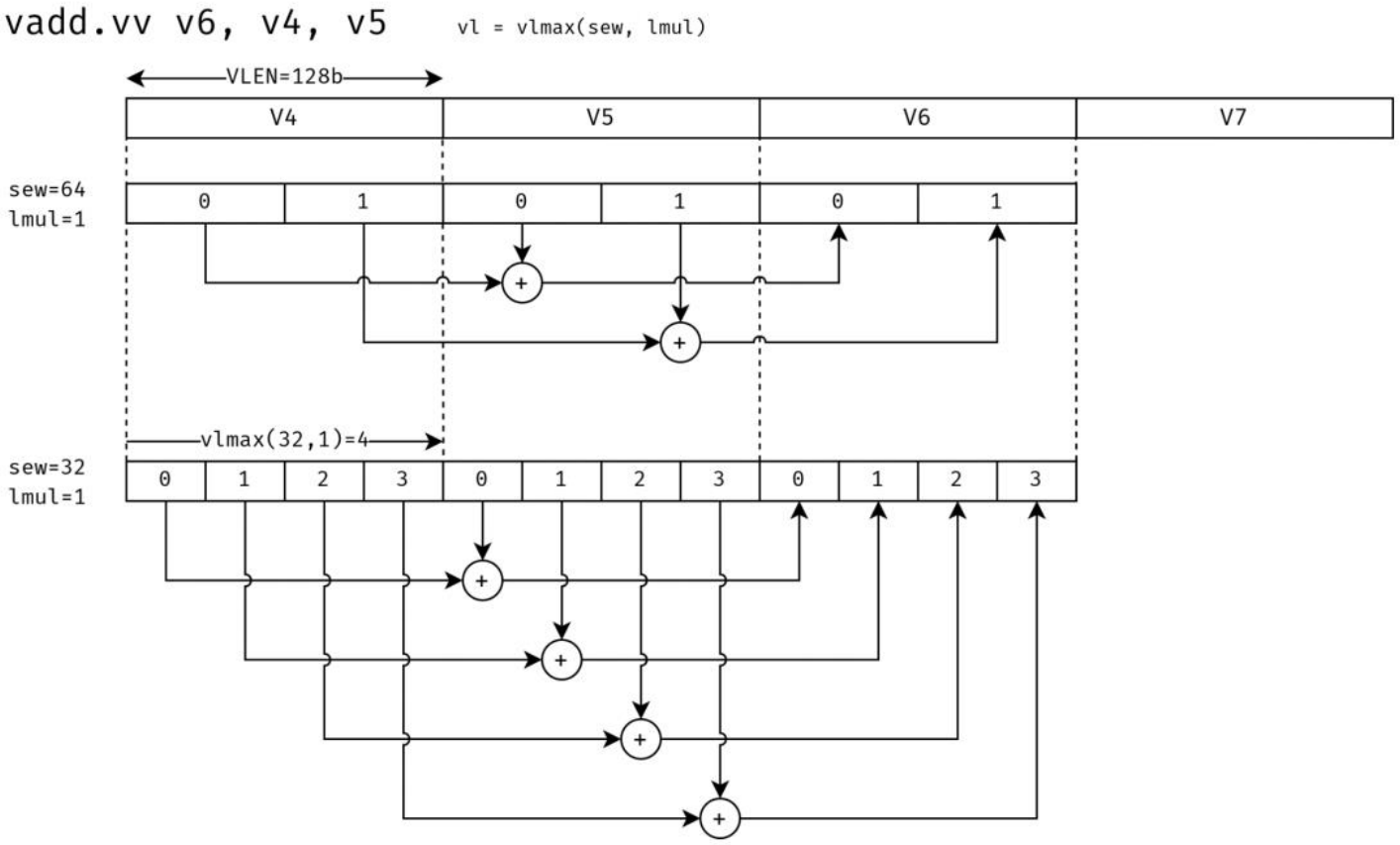
Example (4)


vlmax(sew,lmul) = (VLEN/sew) * lmul
VLEN=128b
V4
V5
V6
V7
vlmax(16,1/2)=4
sew=16
lmul=1/2
0 1 2 3 4 5 6 7
0 ≤ vl ≤ 4
vl=3
Same vlmax
vlmax(32,1)=4
sew=32
lmul=1
0 1 2 3
0 ≤ vl ≤ 4
vl=3

Vector operation

- Vector instructions fully determine the vector operation we are going to execute by using the values of vl and vtype
  - `vl` and `vtype` act as implicit operands of the vector instructions
- When `vl < vlmax` then we have elements that are not operated
  - Those elements are called the **tail elements**
- RVV offers two policies here
  - tail undisturbed. Tail elements in the destination register are left unmodified.
  - tail agnostic. Can behave like tail undisturbed or, alternatively, all the bits of the tail elements of the destination register are set to 1

Example (5)


vadd.vv v6, v4, v5
vl = vlmax(sew, lmul)
VLEN=128b
V4
V5
V6
V7
sew=64
lmul=1
0
1
0
1
0
1
+
+
vlmax(32,1)=4
sew=32
lmul=1
0
1
2
3
0
1
2
3
0
1
2
3
+
+
+
+

Example (6)

vadd.vv v6, v4, v5 vl=3