Digital System Design

Advanced RISCV Topic

M Extension

Speaker：王景平

Advisor：Prof. An-Yeu Wu

Date：2025/05/15

RISC-V M Standard Extension

- RISC-V M extension
  - An optional extension to the RISC-V base integer ISA
  - Specifically adds instructions for integer multiplication and division
- RV32IM = RV32I + M standard extension

Instruction		Format	Meaning
`mul`	`rd, rs1, rs2`	R	Multiply and return lower bits
`mulh`	`rd, rs1, rs2`	R	Multiply signed and return upper bits
`mulhu`	`rd, rs1, rs2`	R	Multiply unsigned and return upper bits
`mulhsu`	`rd, rs1, rs2`	R	Multiply signed-unsigned and return upper bits
`div`	`rd, rs1, rs2`	R	Signed division
`divu`	`rd, rs1, rs2`	R	Unsigned division
`rem`	`rd, rs1, rs2`	R	Signed remainder
`remu`	`rd, rs1, rs2`	R	Unsigned remainder
`mulw`	`rd, rs1, rs2`	R	Multiply, 32-bit
`divw`	`rd, rs1, rs2`	R	Signed division, 32-bit
`divuw`	`rd, rs1, rs2`	R	Unsigned division, 32-bit
`remw`	`rd, rs1, rs2`	R	Signed remainder, 32-bit
`remuw`	`rd, rs1, rs2`	R	Unsigned remainder, 32-bit

Multiplication Operations

*RV32 : XLEN = 32bit
RV64 : XLEN = 64bit

- MUL *mul rd, rs1, rs2*
  - Multiplication instructions operates on registers rs1 and rs2. The operation produces a 2 x XLEN-bit result. The **lowest** XLEN bits of this result are written into the destination register rd.
- MULH/MULHU/MULHSU *mulh/mulhu/mulhsu rd, rs1, rs2*
  - Multiplication instructions operates on registers rs1 and rs2.The operation produces a 2 x XLEN-bit result. The **highest** XLEN bits of this result are written into the destination register rd.
  - MULH : signed x signed / MULHU : unsigned x unsigned / MULHSU : signed x unsigned
- MULW *mulw rd, rs1, rs2*
  - Is only available in RV64
  - Multiplies the **lowest 32 bits of register rs1** with the **lowest 32 bits of register rs2**. The lowest 32 bits of the result are sign-extended to 64 bits and written into the destination register rd.

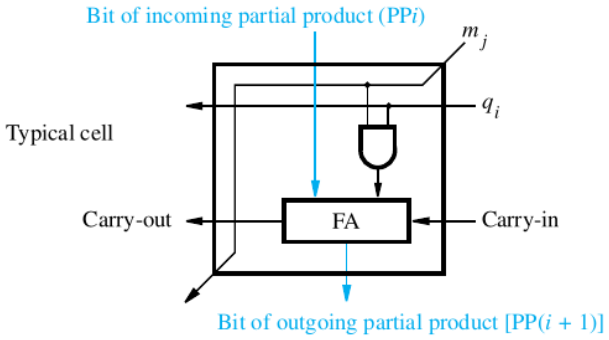
Design of Multiplier

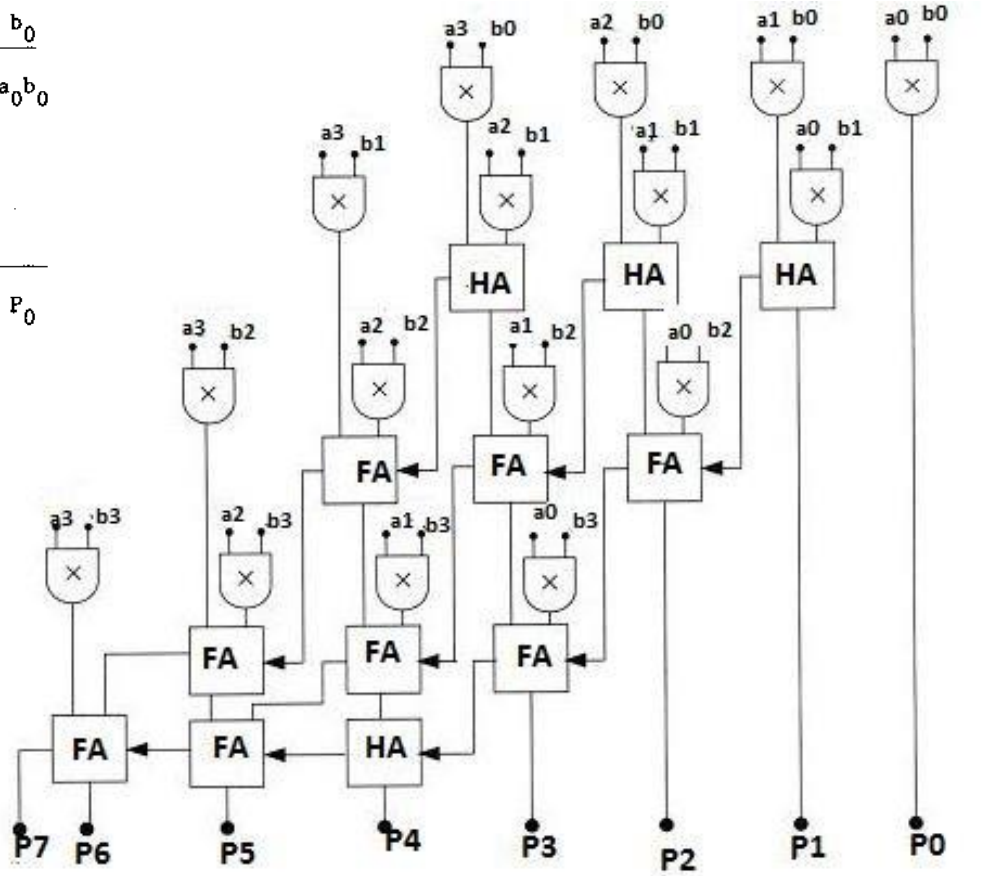
- The CPU performs multiplication operations using a combination of **hardware** and **algorithms**

- Multiplication Algorithms:
  - **Shift and Add Method**: This is a basic algorithm where one number is shifted (multiplied by 2) and conditionally added based on the bits of the other number.
  - **Wallace Tree Multiplication**: This is a method of structuring the addition of multiple binary numbers (partial products) in a way that minimizes the time taken to add them all together
  - **Booth's Algorithm**: Used for multiplying binary numbers. It reduces the number of addition operations by handling strings of 0s and 1s more efficiently.

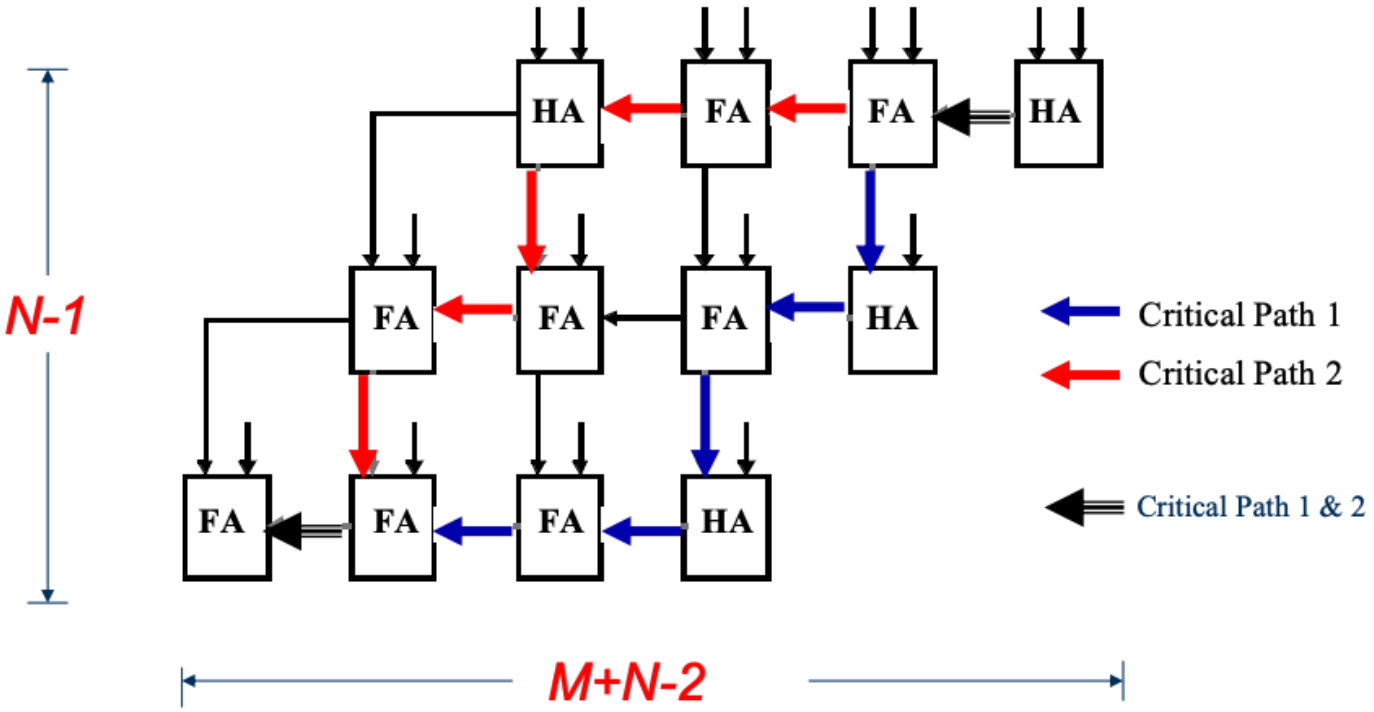
Method 0: Array Multiplier

				(a_3)	a_2	a_1	a_0
				(b_3)	b_2	b_1	b_0
				(a_3b_0)	a_2b_0	a_1b_0	a_0b_0
			(a_3b_1)	a_2b_1	a_1b_1	a_0b_1	
		(a_3b_2)	a_2b_2	a_1b_2	a_0b_2		
	a_3b_3	(a_2b_3)	(a_1b_3)	(a_0b_3)			
(P_7)	P_6	P_5	P_4	P_3	P_2	P_1	P_0





Method 0: Array Multiplier (Cont'd)



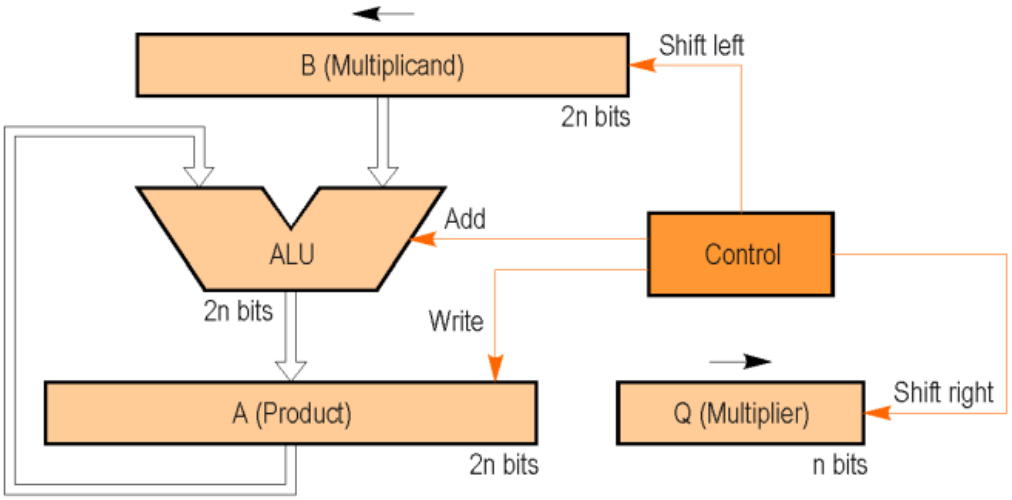
Time: $[(M-1)+(N-1)]t_{carry}+(N-1)t_{sum}$

Area: $M \times N$ full adders

Method 1: Shift and Add Multiplication

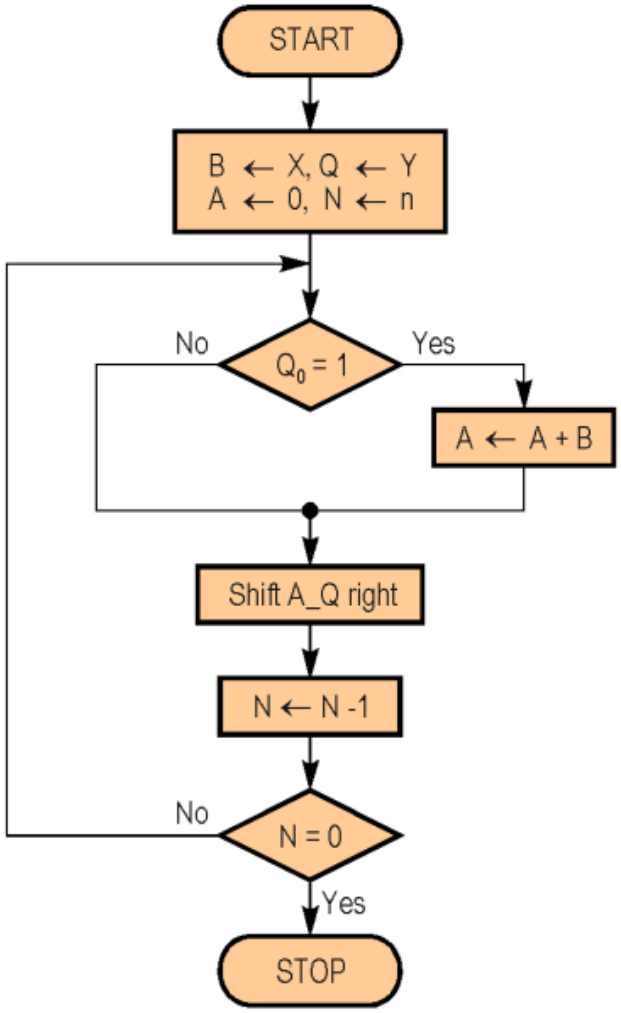
- Binary Representation:
  - Both numbers involved in the multiplication are represented in binary
- Iterative Process:
  - The process is repeated for each bit in the second number, shifting the first number and adding it to the total as required.

```
Multiplicand      1000 ×
Multiplier        1001
                  1000
                 0000
                0000
               1000
Product        1001000
```

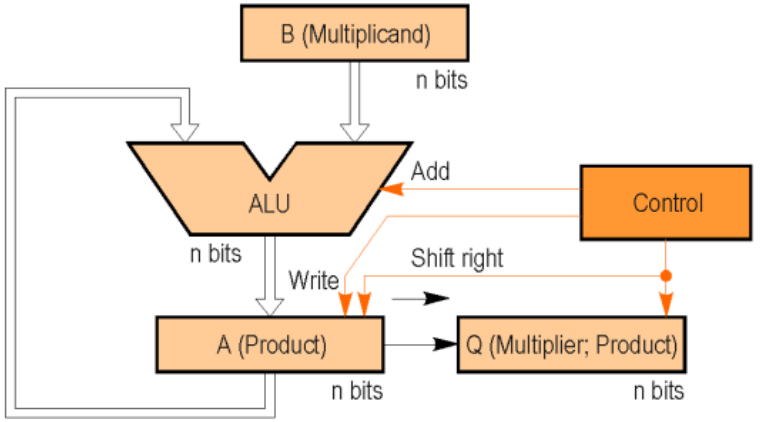


[ref] https://users.utcluj.ro/~Baruch/book_ssce/SSCE-Shift-Mult.pdf

Method 1: Improvement in Area



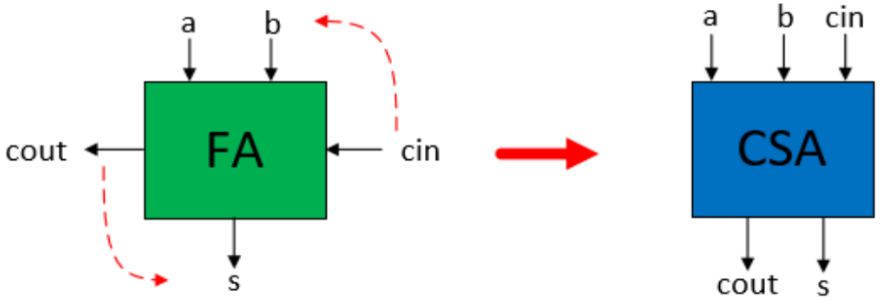
Step	A	Q	B	Operation
0	0000	1100	1001	Initialization
1	0000	0110	1001	Shift right A_Q
2	0000	0011	1001	Shift right A_Q
3	1001	0011	1001	Add B to A
	0100	1001	1001	Shift right A_Q
4	1101	1001	1001	Add B to A
	0110	1100	1001	Shift right A_Q

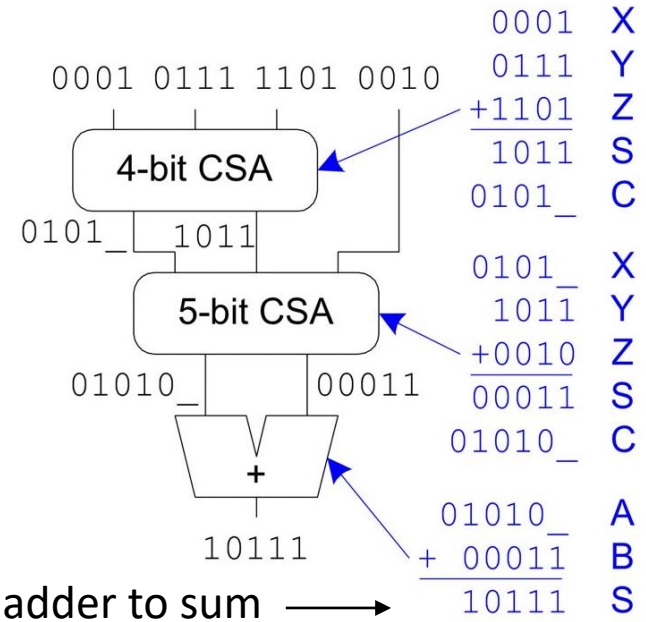


The A register is only n bits wide

Method 2: Carry Save Addition

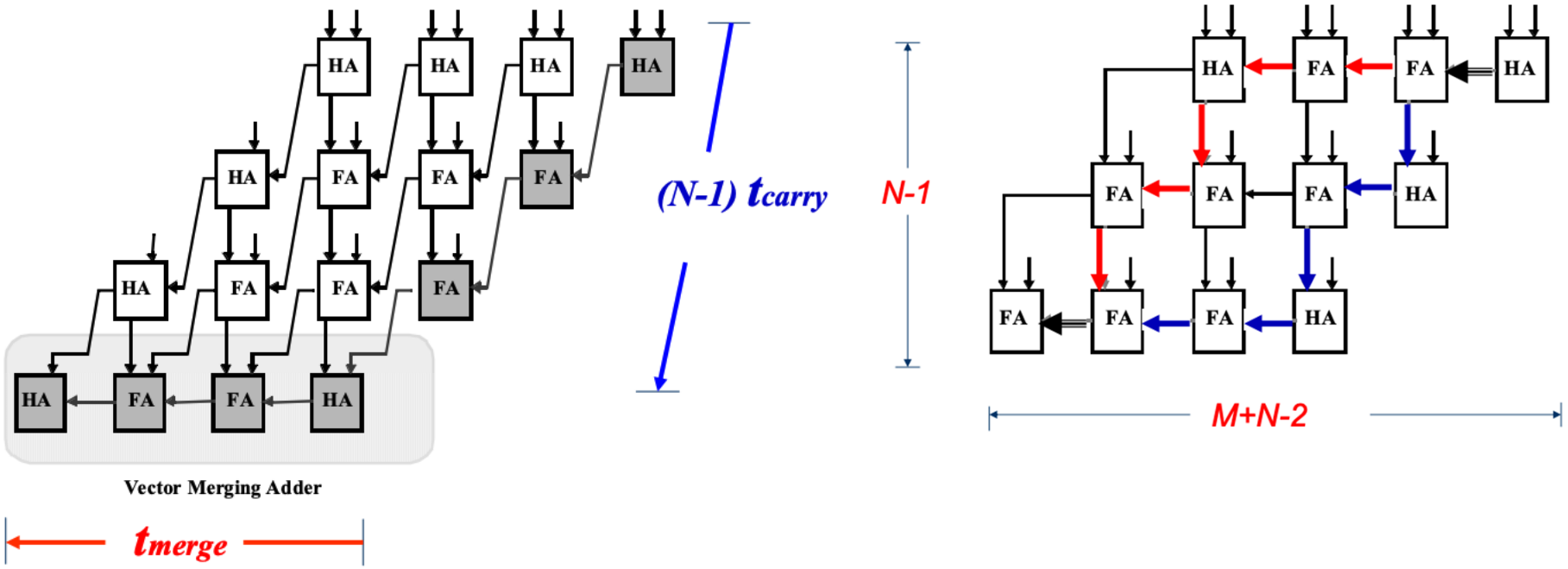
- CSA is designed to add three or more binary numbers quickly without having to propagate carries immediately.
- A full adder sums 3 inputs and produces 2 outputs
  - Carry output has twice weight of sum output
- N full adders in parallel are called carry save adder
  - Produce N sums and N carry outs





Finally, Use carry propagate adder to sum →

Method 2: CSA Multiplier

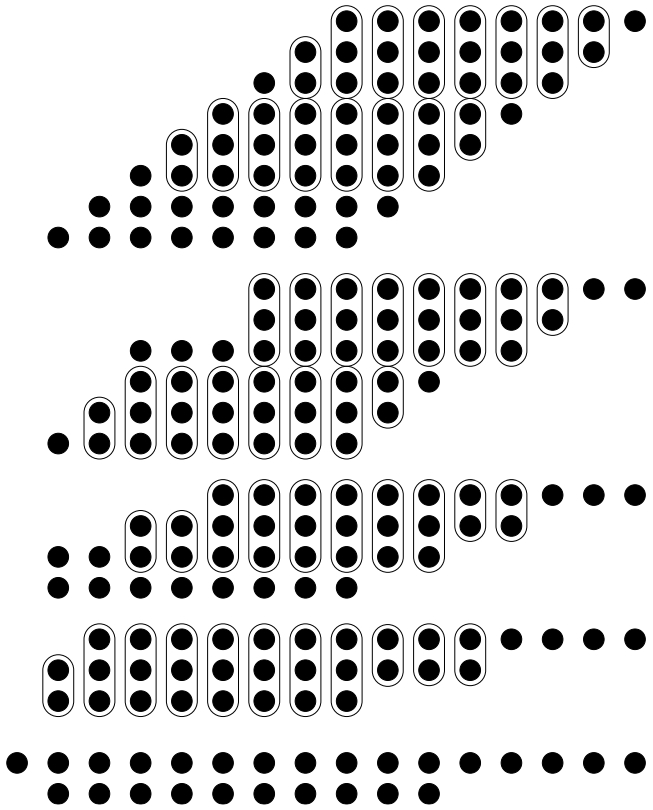


Time: $(N-1)t_{carry} + t_{merge}$ $\Longleftrightarrow$ $(M-1)t_{carry} + (N-1)t_{sum}$

Area: $M \times N$ full adders + M-bit merge

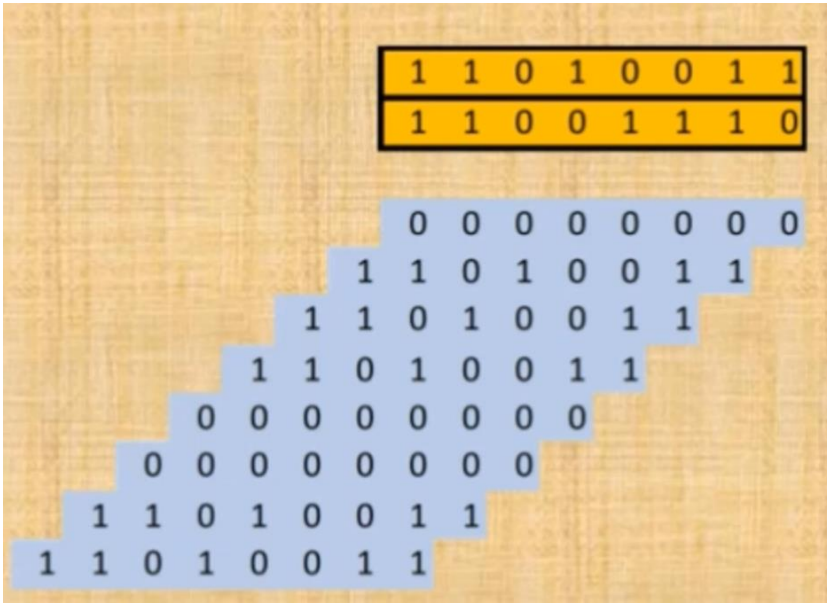
Method 2: Wallace Tree Multiplication

- A fast way to multiply two binary integers
  - Flatten the CSA multiplier for a shorter delay
  - However, it requires a lot of hardware
- Three stages
  - Stage 1 : Partial Products
    - Time Delay : 1 AND gate
  - Stage 2 : Partial Products Addition
    - Time Delay : several full adder delay
  - Stage 3 : Final Addition
    - Time Delay : (M+N)-bit adder



Stage 1 : Partial Products

- Products are the result of a simple AND gate
- All products are done simultaneously

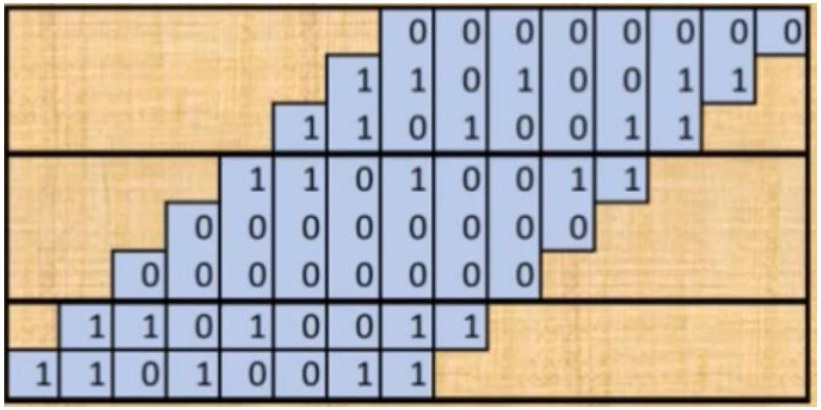


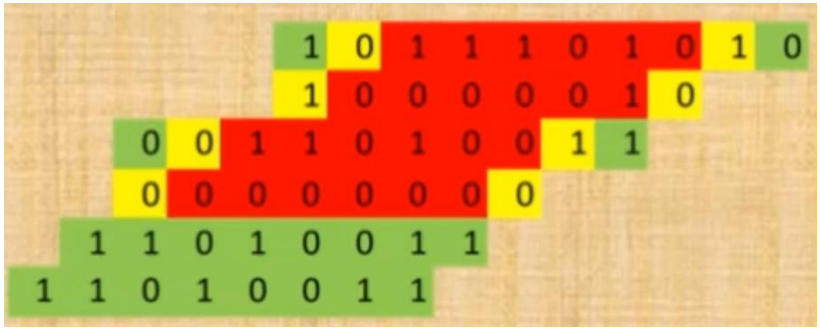
Example : 8 * 8 bit multiplication

[ref] https://www.youtube.com/watch?v=4-l_PGPog9o

Stage 2 : Partial Products Addition (1/3)

- To add up columns, add up three rows at a time
- The result for each set of three rows is a set of two rows
- Each resulting set of two rows has a row for the sum and a row for the carry-out
- Odd rows are left alone
- Red : full adder output
- Yellow : half adder output
- Green: left alone





Stage 2 : Partial Products Addition (2/3)

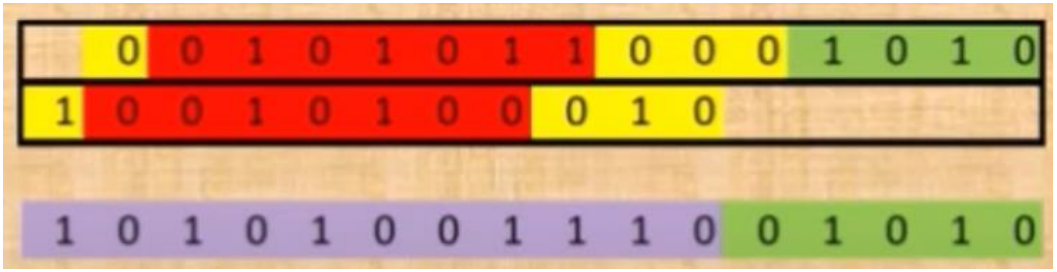
- Repeat the process
- This time, there are two sets of three rows
- Gray boxes indicate the summation bits have been moved down to the carry-out row

Stage 2 : Partial Products Addition (3/3)

- Stop if two bits are available in the column an also in preceding column.

Stage 3 : Final Addition

- Final result is calculated by adding the final two rows
- Result is that Wallace Tree multiplication takes about the same amount of times as a 2N-bit ripple-carry adder



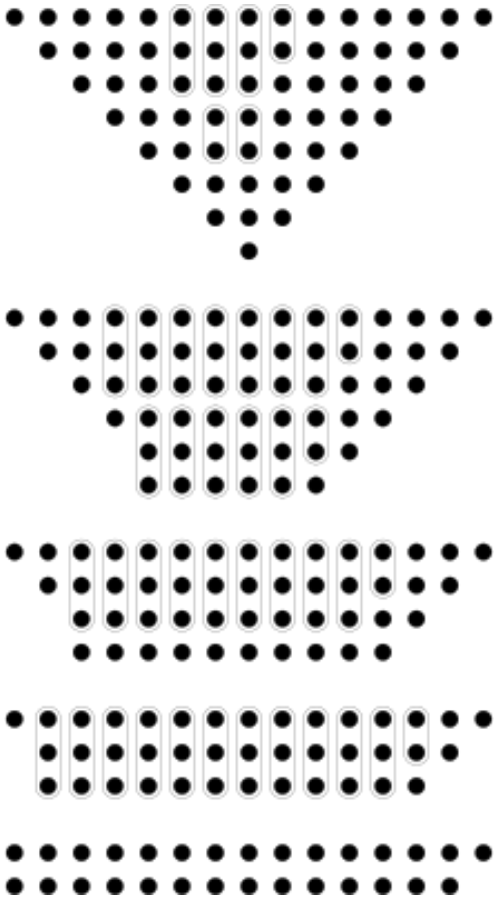
Method 2: Dadda Multiplier

- Dadda multiplier is a type of binary multiplier that is similar to the Wallace tree multiplier
  - Fine-grain manipulate on CSA
  - Pros: Faster than the Wallace tree multiplier
  - Cons: More complex rules

- The progression of the reduction is controlled by a maximum-height sequence d_j, defined by:

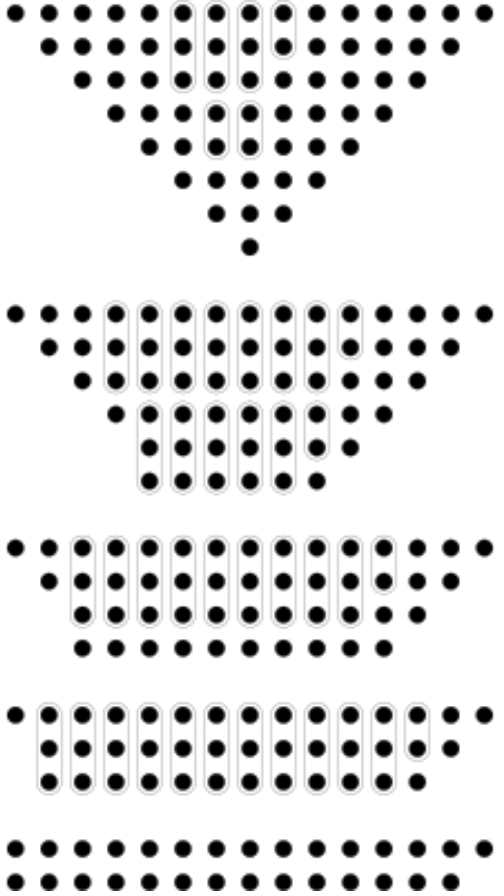
$$d_1 = 2, d_{j+1} = floor(1.5 d_j)$$

  - This yields a sequence like so: 2, 3, 4, 6, 9, etc.



Method 2: Dadda multiplier (Cont'd)

- Rule for reduction
  1. If the position has bits less than d, move on
  2. If the position has bits equal to d + 1, use an HA to compress two bits, pushing one result to the next position. The current position would be reduced to d, allowing us to move to the next bit position
  3. If the position has any higher number of bits, use an FA to compress, pushing bits and repeating again from the first step. In other words, this step would keep us looping on the same position until the height in the position reduces to d.



Method 3: Booth Encoding

- Multiplies two **signed binary numbers** in **two's complement** notation

- Algorithm principle :

$$M \times ''0\ 0\ 1\ 1\ 1\ 1\ 1\ 0'' = M \times (2^5 + 2^4 + 2^3 + 2^2 + 2^1) = M \times 62$$

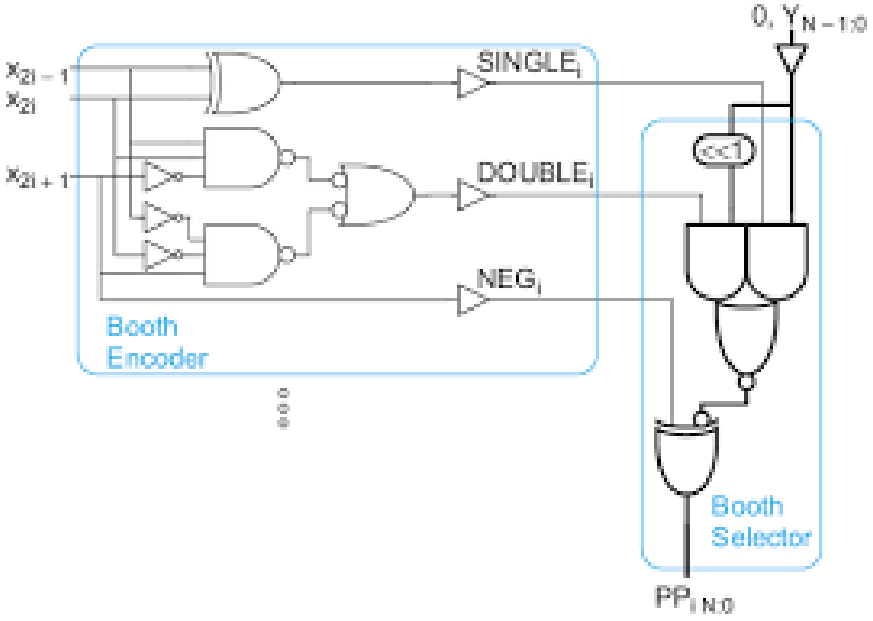
$$M \times ''0\ 1\ 0\ 0\ 0\ 0\text{-}1\ 0'' = M \times (2^6 - 2^1) = M \times 62$$

- Generalization :

$$(\ldots 0\overbrace{1\ldots 1}^{n}0\ldots)_2 \equiv (\ldots 1\overbrace{0\ldots 0}^{n}0\ldots)_2 - (\ldots 0\overbrace{0\ldots 1}^{n}0\ldots)_2 \circ$$

Method 3: Booth Encoding (Cont'd)

- Booth encoder generates control lines for each PP
- Booth selectors choose PP lines



Multiplier Bits Block			Recoded 1-bit pair		2 bit booth	
i+1	i	i-1	i+1	i	Multiplier Value	Partial Product
0	0	0	0	0	0	Mx0
0	0	1	0	1	1	Mx1
0	1	0	1	-1	1	Mx1
0	1	0	1	0	2	Mx2
1	0	0	-1	0	-2	Mx-2
1	0	1	-1	1	-1	Mx-1
1	1	0	0	-1	-1	Mx-1
1	1	0	0	0	0	Mx0

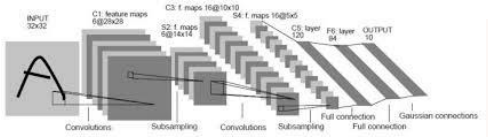
Digital System Design

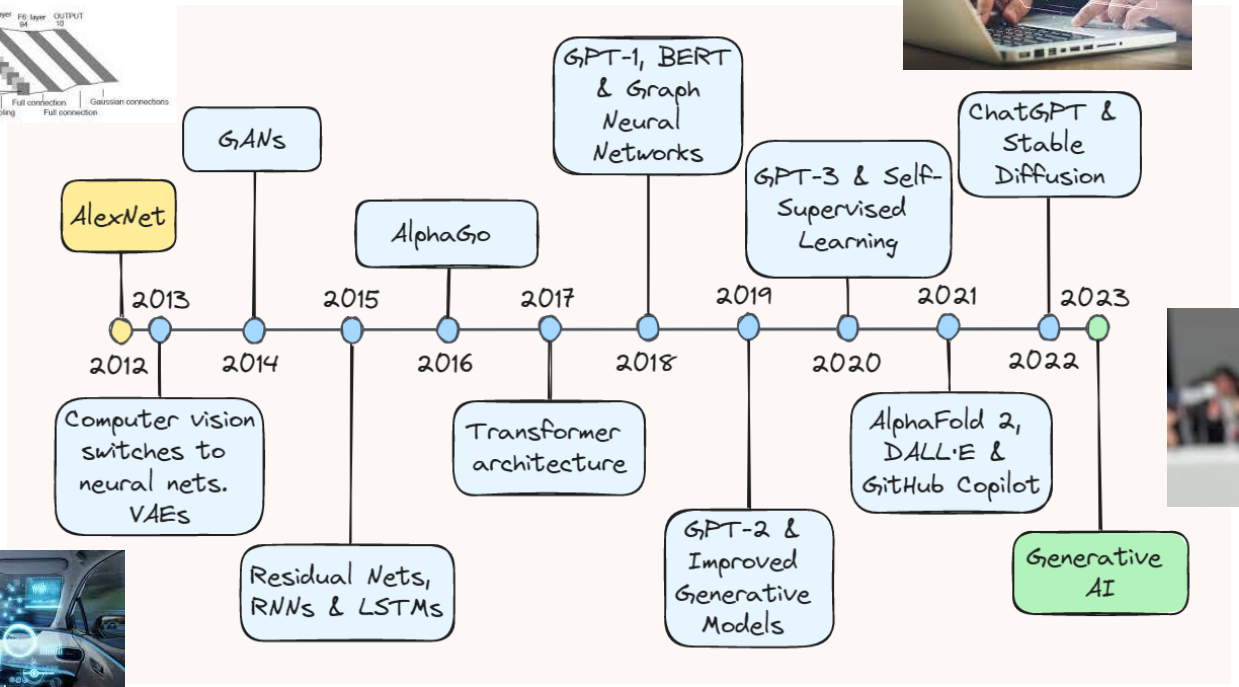
Advanced RISCV Topic

Advanced Applications

Neural Network (NN) Applications

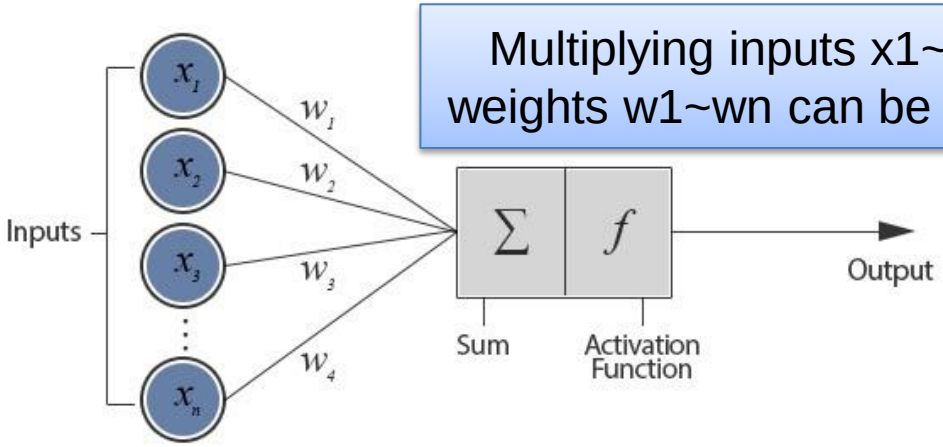
- Neural Network algorithms are increasingly used in many ML tasks
  - Computer Vision (CV)
  - Natural Language Processing (NLP)





(1950) The Perceptron

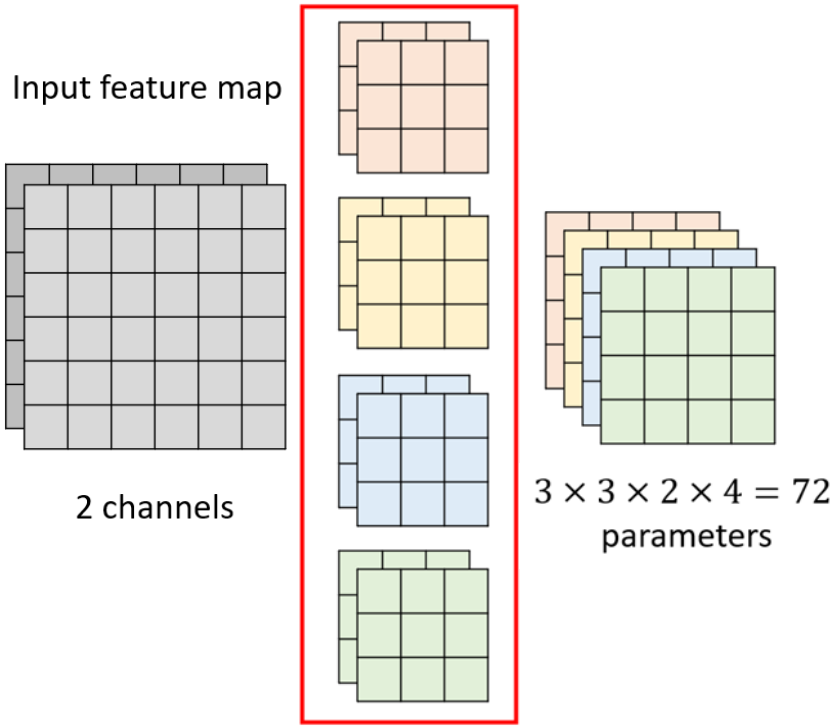
- **Perceptrons**: conceptualized in the 1950s-60s by Frank Rosenblatt
  - Contemporary neural networks use diverse neuron models but remain rooted in the foundational principles set by the perceptron



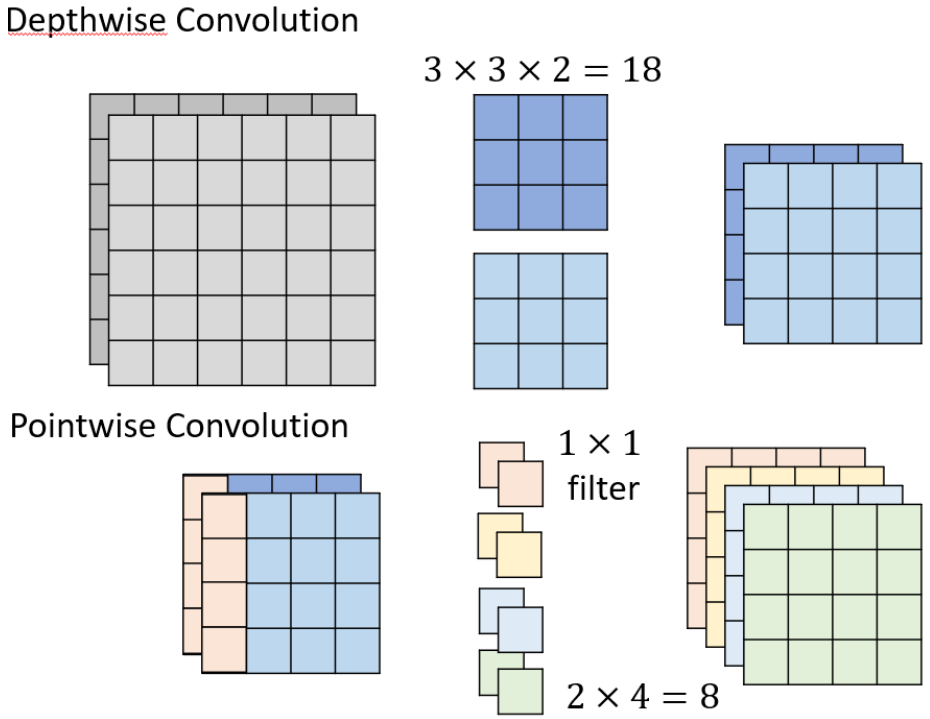
- As you can see, the network of nodes sends signals in one direction
  - This is called **a feed-forward network**
- The figure depicts a neuron connected with *n* other neurons and thus receives *n* inputs (x_1, x_2, ….. x_n)
  - This configuration is called a **Perceptron**

(2017) MobileNet: Depthwise & Pointwise Separable Convolution Neural Network (CNN)

- Standard CNN



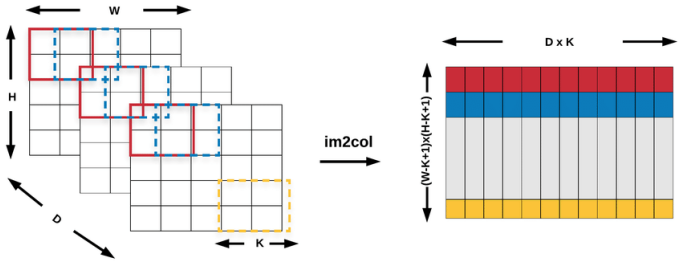
- Depthwise & Pointwise CNN



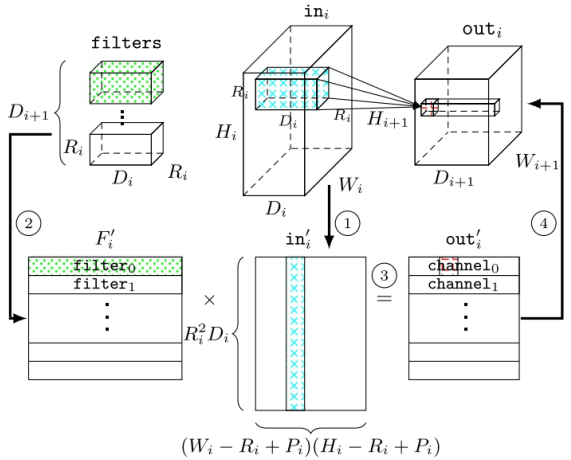
Performing consecutive **matrix multiplications** and managing **data movement** efficiently are crucial !

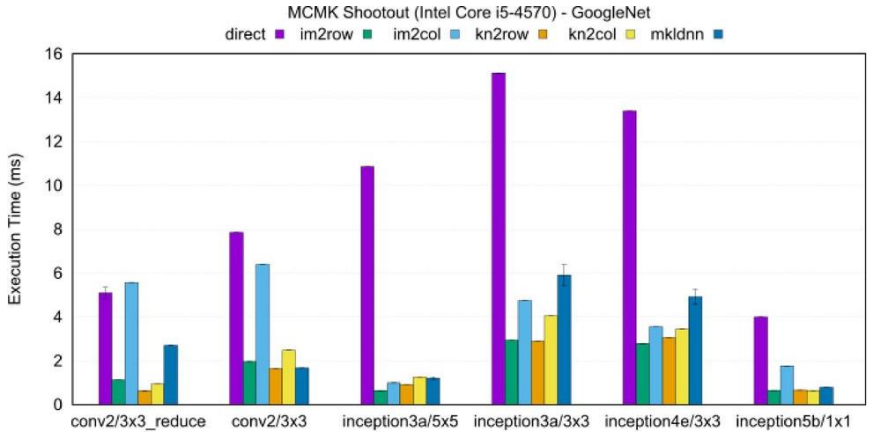
Expressing a Convolutional Layer as a Single Matrix Multiplication

- In a standard convolutional operation
  - A kernel slides over the input image
  - Dot products are calculated at each position



- The **im2col** technique reshapes the input image patches and kernel filters into matrices
  - Utilizing efficient linear algebra libraries
  - Hardware optimizations that can accelerate matrix multiplications

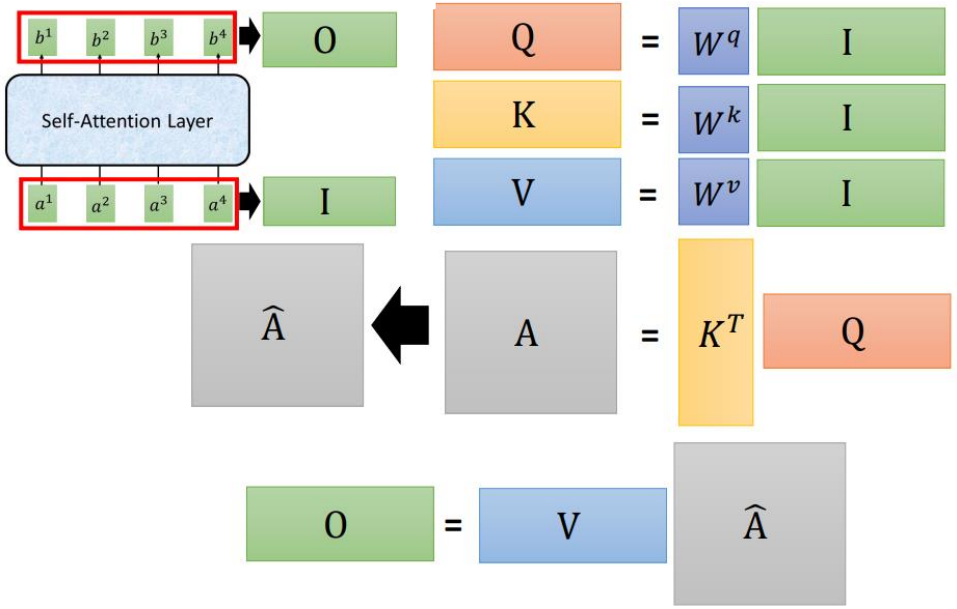


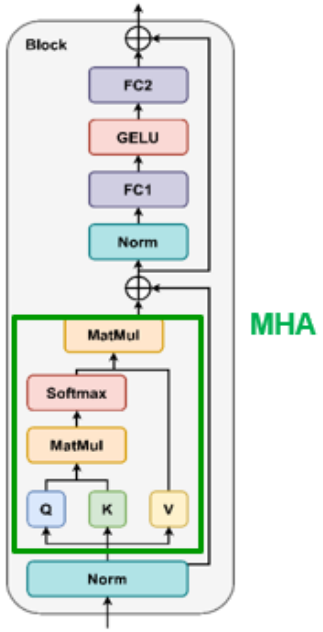


(2019) Vision Transformer (ViT): Self-attention

❖ Multi-head attention (MHA)

- ❖ Self-attention: $\boldsymbol{Attention(Q,K,V) = softmax\left(\frac{QK_T}{\sqrt{d_k}}\right)V}$
- ❖ MHA: $\boldsymbol{MultiHead(Q,K,V) = Concat(head_1, \ldots, head_h)W^O}$

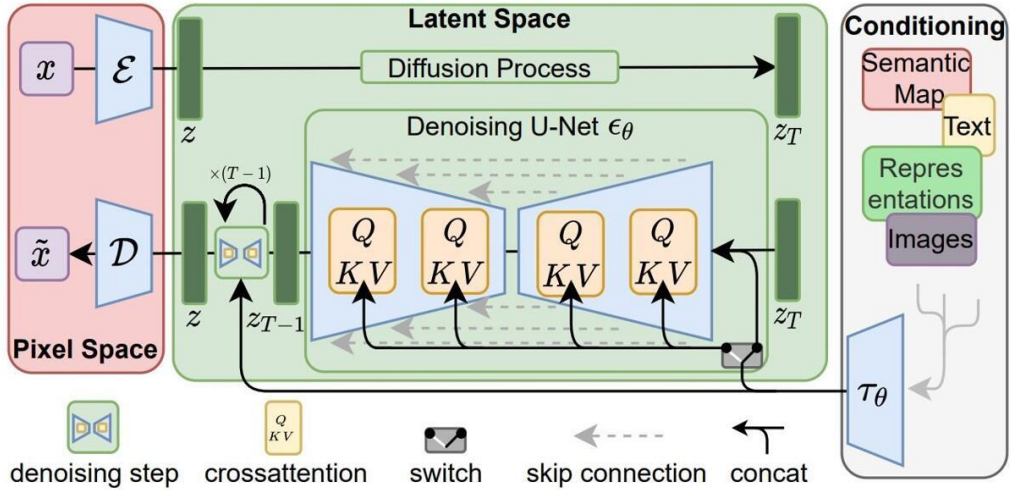


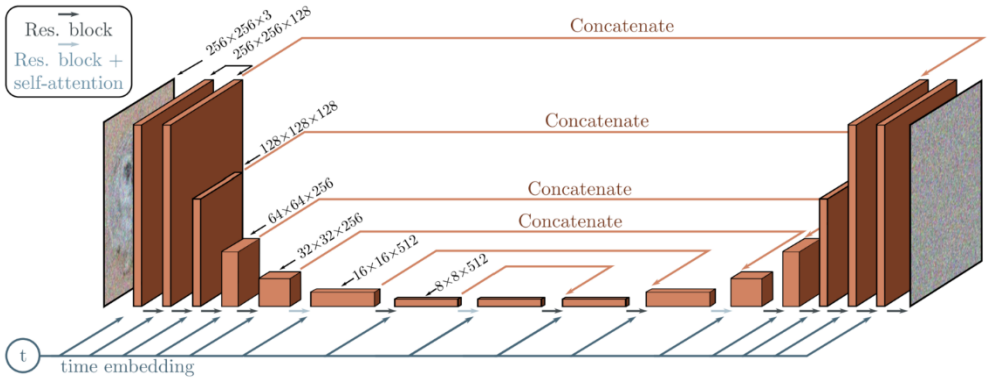


ViTs offer more possibilities for matrix parallelism compared to CNNs !

(2022) Stable Diffusion: U-Net

- Stable Diffusion
  - Variational autoencoder (VAE)
  - **U-Net**
  - Test Encoder
- U-Net
  - ResNet backbone
  - Cross-attention mechanism





Diffusion model is composed of CNNs and VITs !

NNs Computational Characteristics (1/2)

- **Parallelism**: NNs often involve matrix multiplications and other parallelizable operations
  - ISAs that support parallel processing, such as SIMD or MIMD can significantly enhance the performance
- **Floating-point Operations**: NNs require high precision floating-point operations for training and inference
  - ISAs that efficiently support floating-point operations are crucial
- **Memory Access**: NN computations involve frequent memory access
  - ISAs with features like wide SIMD registers or optimized memory access patterns can mitigate bottlenecks associated with memory bandwidth

NNs Computational Characteristics (2/2)

- **Energy Efficiency**: Given the resource-intensive nature of NN computations
  - ISAs that prioritize energy efficiency can be advantageous, especially in edge computing or mobile devices

- **Scalability**: As NNs continue to grow in size and complexity
  - Scalable ISAs that accommodate larger models and datasets can be beneficial

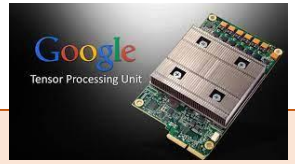
Specialized hardware or customized ISA extensions are often explored to optimize neural network computations !

Computation Platform for NN Applications

- **CPU**
  - Suitable for various applications
  - Strong in single-threaded tasks
  - Limited parallelism
- **GPU**
  - Well-suited for parallel tasks
  - Efficient for large-scale matrix computations
  - Not general
- **TPU**
  - Optimized for deep learning tasks involving tensors
  - Designed for high-throughput, energy-efficient processing

CPU
- Small models
- Small datasets
- Useful for design space exploration

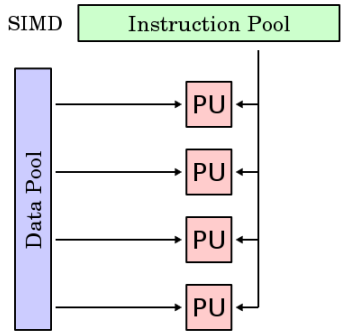
GPU
- Medium-to-large models, datasets
- Image, video processing
- Application on CUDA or OpenCL



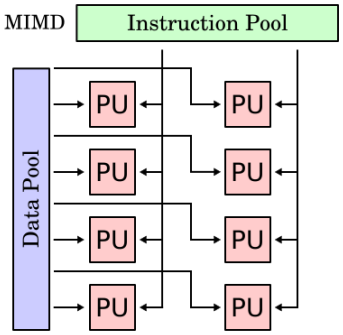
TPU
- Matrix computations
- Dense vector processing
- No custom TensorFlow operations

ISAs Optimized for NN Computations

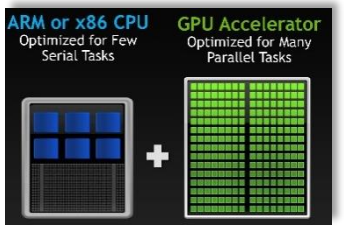
- SIMD (Single Instruction, Multiple Data):
  - Intel SSE: Extension to x86 architecture with SIMD instructions
  - ARM NEON: SIMD extension for ARM processors



- MIMD (Multiple Instruction, Multiple Data):
  - x86_64: Supports MIMD through multiple cores and threads
  - ARM multicore processors: Provide MIMD capabilities

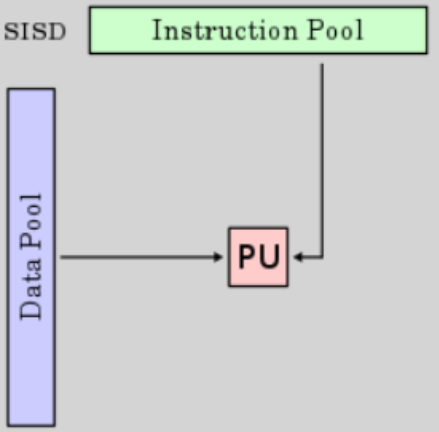
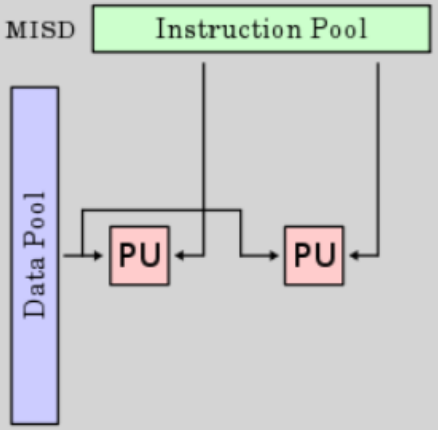
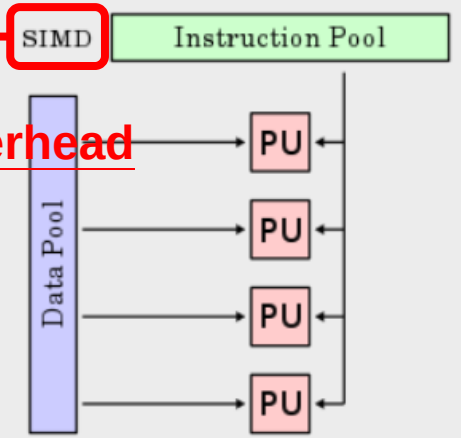
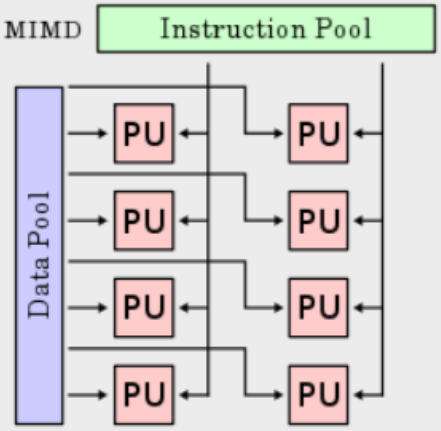


- GPU Architectures:
  - NVIDIA CUDA: Enables parallel processing on NVIDIA GPUs
  - AMD ROCm: AMD's open-source GPU computing platform



- Google TPU Architecture:
  - Custom TPU hardware designed for parallel NN inference and training

Flynn's taxonomy

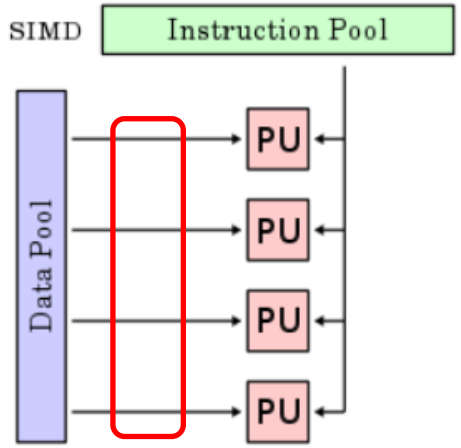
	Single instruction	Multiple instruction
Single data		
Multiple data		

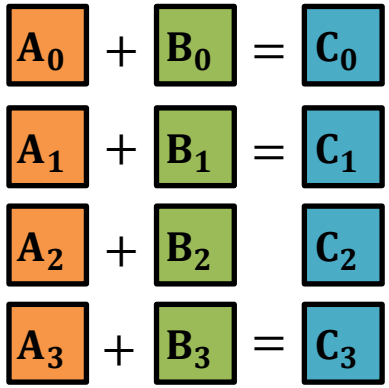
Reduce Instruction Overhead

[9]

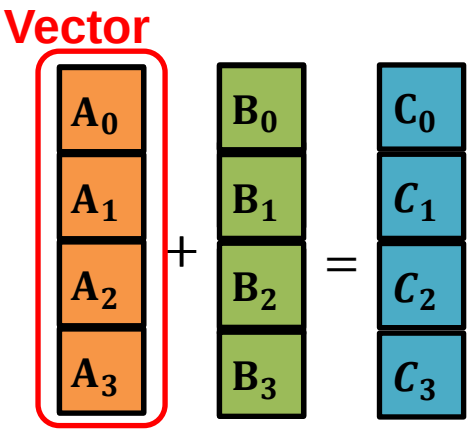
SIMD Processing

- Single instruction operates on multiple data elements
- SIMD architectures can exploit significant data-level parallelism for:
  - Matrix-oriented scientific computing
  - Media-oriented image and sound processors
- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
- The vector architecture is SIMD





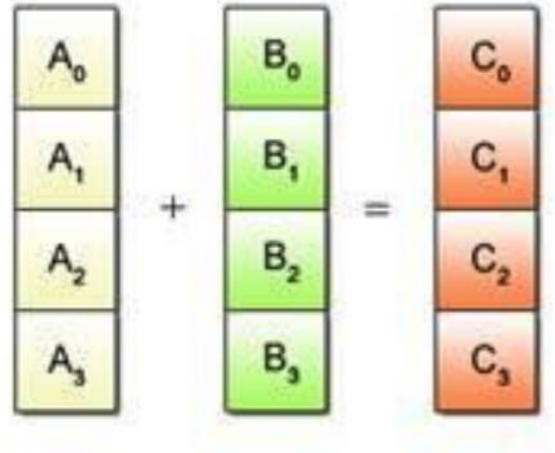
Scalar Operation



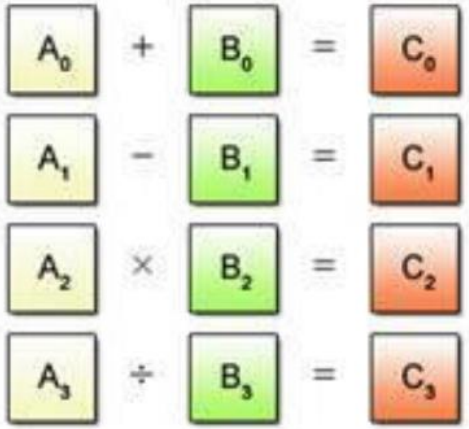
SIMD Operation

Restrictions on SIMD Operations

- Despite the advantage of being able to process multiple data per instruction, SIMD operations can only be applied to certain predefined processing patterns.



SIMD Processable Pattern



SIMD Unprocessable Pattern

Data Used in SIMD Programming

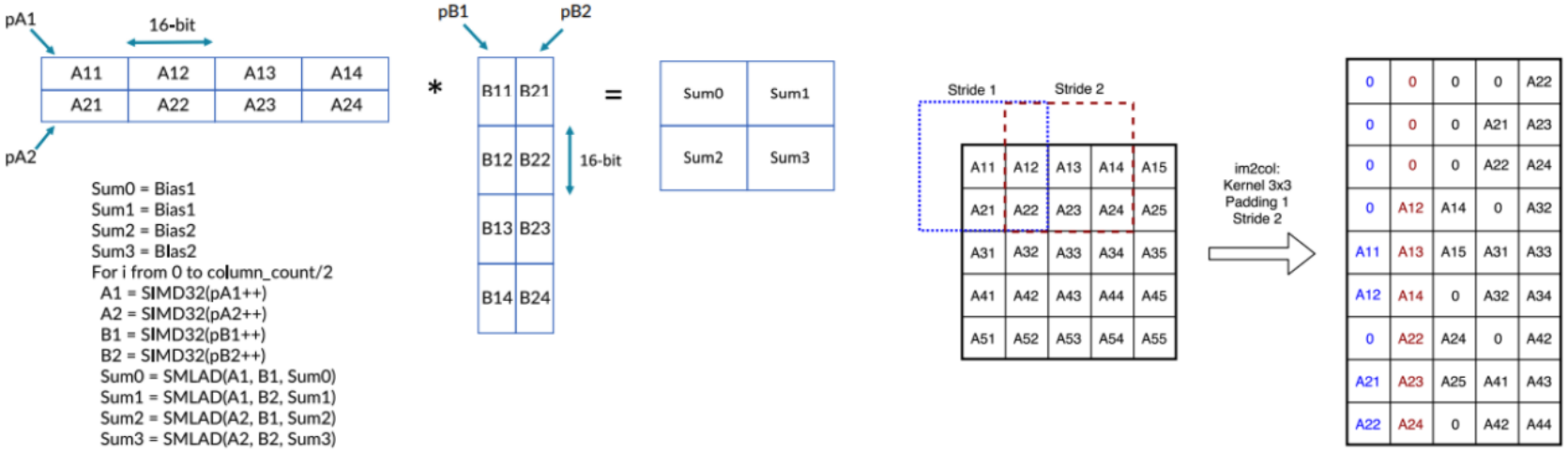
- Data types used for SIMD operations are called vector types. Each vector type has its corresponding scalar type:

Data type	Description
__m128	128 bits, with 4 float
__m128d	128 bits, with 2 double
__m128i	128 bits, with several integer
__m256	256 bits, with 8 float
__m256d	256 bits, with 4 double
__m256i	256 bits, with several integer

Table: List of vector type

SIMD in Machine Learning

- SIMD is widely used in Neural Network [Matrix-oriented computing]
  - Matrix Multiplication
  - Convolution
  - Activation Function
  - Pooling



RISC-V Vector Extension (RVV)

RISC-V: The Free and Open RISC Instruction Set Architecture

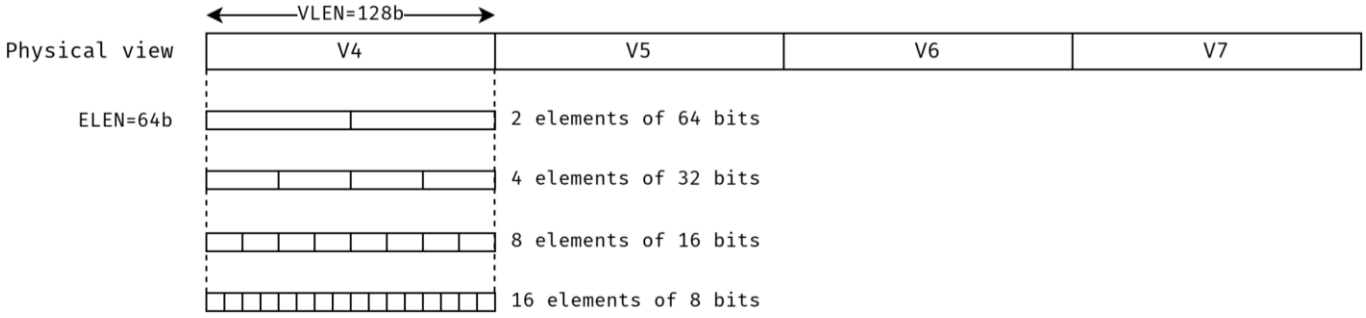
- An open processor architecture started by UC Berkeley
  - Compact, modular, extensible
- The RISC-V Vector Extension (RVV) aims at providing vector computation capabilities to the RISC-V architecture
- RVV wants to have wide applicability, so the design is very flexible
- The flexible design poses some challenges in different aspects when using the ISA
  - Compiler and Developers
- RISC-V Vector Extension (RVV):
  - Defined in a RISC-V International Task Group
  - RVV extension supports vector instruction set with scalable vector registers
  - 2x and 4x data expansion arithmetic
  - Over 300+ vector instructions, covering load/store, integer, fixed-point/ floating-point operations

RVV Design

- Vector ISAs are typically large for several reasons
  - **Vector Equivalents**: RVV include equivalents for most scalar arithmetic instructions, facilitating seamless integration of vectorized processing
  - **Memory Access and Vector Element Manipulation**: Specific instructions within RVV are dedicated to memory access and manipulation of vector elements, optimizing data handling
  - **Predication (Masking) Support**:
    - Operations to form predicates (e.g., comparisons)
    - Additional operands in the instructions
- This impacts the design of RVV
  - Vector operations cannot be encoded in a 32-bit instruction
  - CPU state is used instead

RVV Parameters and Basic State

- RVV defines 32 architectural vector registers of size VLEN bits
  - v0 to v31
  - VLEN is a constant parameter chosen by the implementor and must be a power of 2
  - 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



RVV Operational State (1/2)

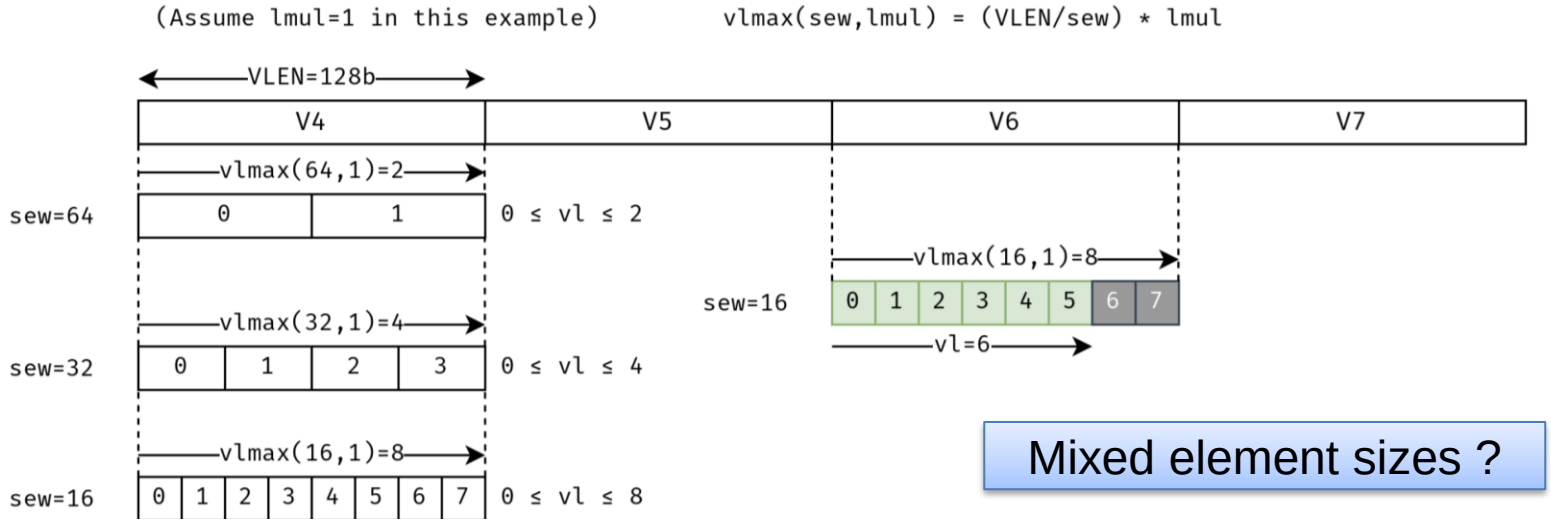
- There are two unprivileged registers used when operating vectors in RVV
  - ***vtype***: vector type
  - ***vl***: vector length (not to be confused with VLEN!)

- ***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
    - $8 \le sew \le ELEN$
  - lmul (length multiplier): Allows grouping registers
    - $lmul = 2^k$ where $-3 \le k \le 3$ (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
  - $0 \le vl \le vlmax(sew, lmul)$
  - $vlmax(sew, lmul) = (VLEN / sew) \times lmul$

RVV Operational State (2/2)

- 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



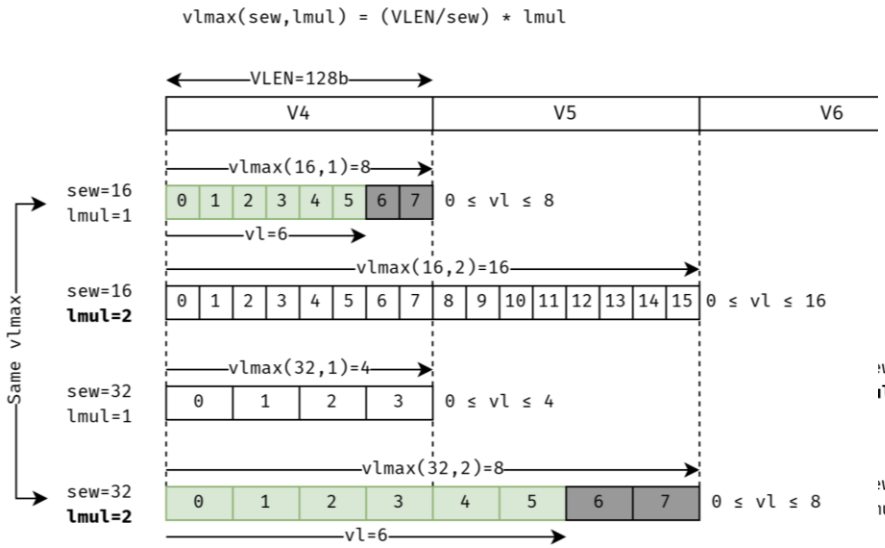
- When operating with vectors whose elements are of different size, we have different number of elements
  - This causes problems to algorithms, which want to operate with the same number of elements

Length Multiplier (1/2)

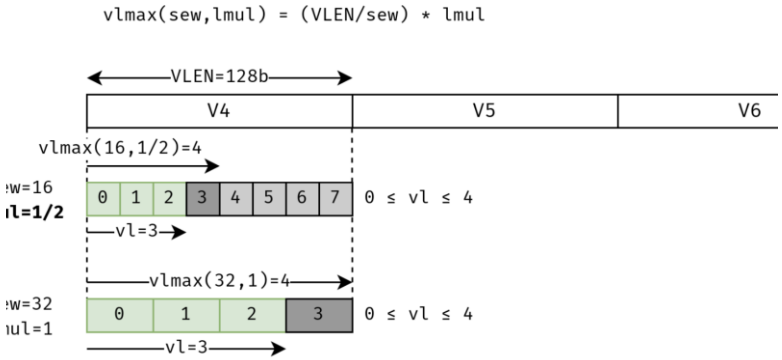
- RVV allows the two scenarios via the length multiplier
- When lmul = 1 we can operate up to all the elements of a vector register
- When lmul < 1 we can operate up to a fraction of all the elements of a vector register lmul ∈ {1/2, 1/4, 1/8}
- When lmul > 1 the operation uses a vector group of lmul vector registers
  - A vector group “gangs” several vector registers
  - The 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

Length Multiplier (2/2)

- We can “harmonise” the number of elements
  - Not using the whole vector register for the small element sizes
  - Use more than one vector register for the large element sizes



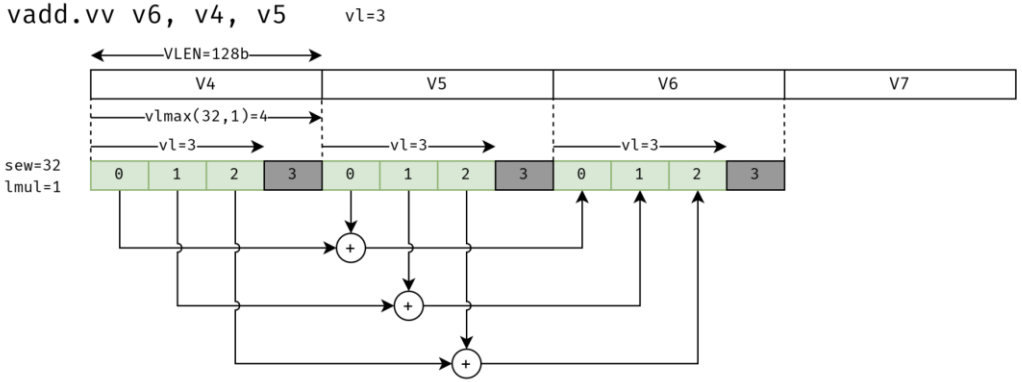
(1) lmul > 1



(2) lmul < 1

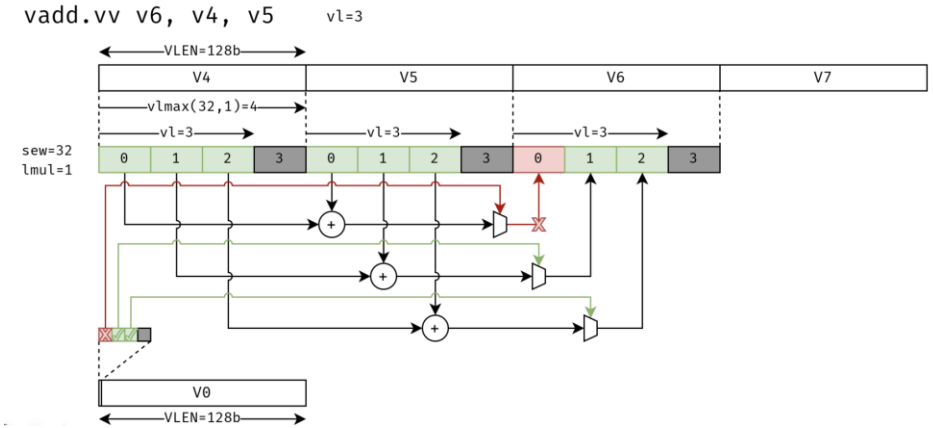
Vector Operation

- 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

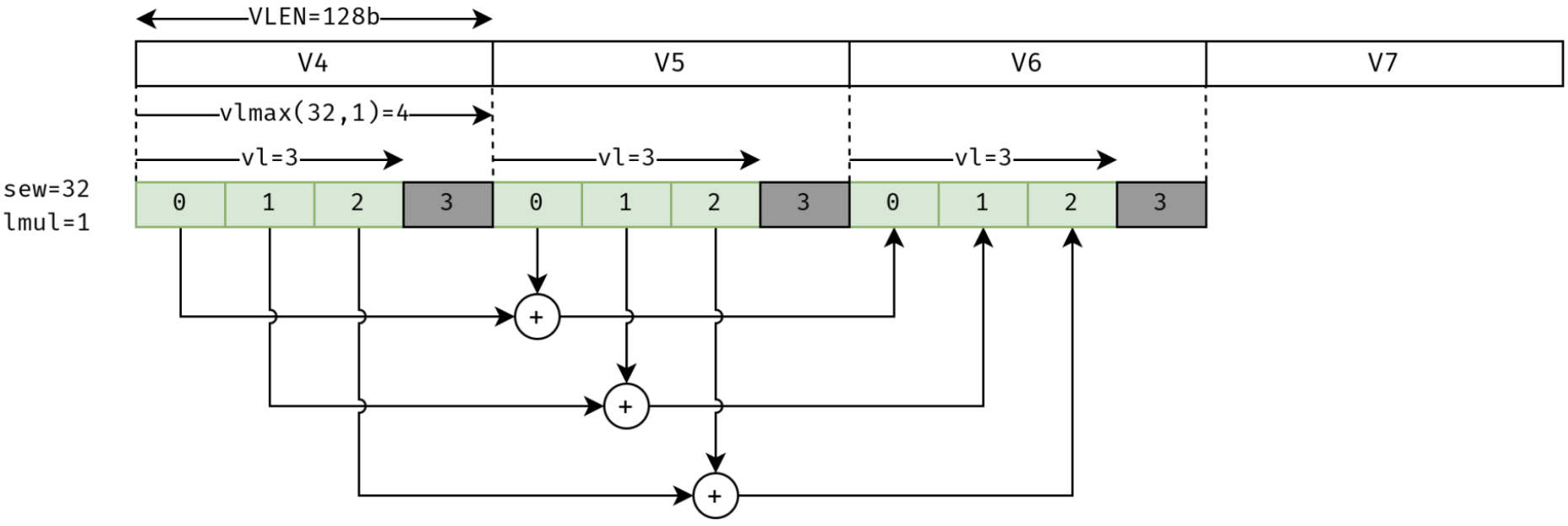


Masking (Predication)

- Control flow may be problematic when using vector instructions
- A mask vector is a vector whose elements are single bits
  - There are no distinguished vector registers for mask vectors (v0~v31)
  - RVV defines a specific layout for mask vectors
    - Bits are packed contiguously in the vector register, starting from the LSB bit as the 0th element of the mask
- Instructions can be masked using the **v0 register**
  - While it is possible to operate mask vectors in all the other registers, only v0 can be used as a mask operand when masking a vector instruction



```
vsetivli x10, 3, e32,m1,ta,ma   # vl ← 3, sew ← 32, lmul ← 1
vadd.vv v6, v4, v5
```



- vsetvli rd, rs, eN,mX,tP,mP (updates rd with the vector length computed)
  - rs is an input register operand that contains the application vector length (AVL) which represents the vector length the program wants to use
  - vsetivli replaces this operand with a small immediate from 0 to 31
  - N in eN is the sew (8, 16, 32, 64, …)
  - X in mX is the lmul (spelled as fY for 1/Y cases)
  - P is the policy for tail (t) and mask (m): u for undisturbed, a for agnostic

Vector-vector Add Example

```
4         # vector-vector add routine of 32-bit integers
5         # void vvaddint32(size_t n, const int*x, const int*y, int*z)
6         # { for (size_t i=0; i<n; i++) { z[i]=x[i]+y[i]; } }
7         #
8         # a0 = n, a1 = x, a2 = y, a3 = z
9         # Non-vector instructions are indented
10    vvaddint32:          vsetvli rd, rs1, vtypei   # rd = new vl, rs1 = AVL, vtypei = new vtype setting
11        vsetvli t0, a0, e32, ta, ma  # Set vector length based on 32-bit vectors
12        vle32.v v0, (a1)         # Get first vector
13          sub a0, a0, t0         # Decrement number done
14          slli t0, t0, 2         # Multiply number done by 4 bytes
15          add a1, a1, t0         # Bump pointer
16        vle32.v v1, (a2)         # Get second vector
17          add a2, a2, t0         # Bump pointer
18        vadd.vv v2, v0, v1       # Sum vectors
19        vse32.v v2, (a3)         # Store result
20          add a3, a3, t0         # Bump pointer
21          bnez a0, vvaddint32    # Loop back
22          ret                    # Finished
```

Many More Details We Cannot Cover ….

- We cannot cover many more details of the ISA of the V extension
- We recommend you look at the full specification here:
  - https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf

20. Vector Instruction Listing

Integer					Integer				FP			
funct3					funct3				funct3			
OPIVV	V				OPMVV	V			OPFVV	V		
OPIVX		X			OPMVX		X		OPFVF		F	
OPIVI			I									

funct6					funct6				funct6			
000000	V	X	I	vadd	000000	V		vredsum	000000	V	F	vfadd
000001					000001	V		vredand	000001	V		vfredsum
000010	V	X		vsub	000010	V		vredor	000010	V	F	vfsub
000011		X	I	vrsub	000011	V		vredxor	000011	V		vfredosum
000100	V	X		vminu	000100	V		vredminu	000100	V	F	vfmin
000101	V	X		vmin	000101	V		vredmin	000101	V		vfredmin
000110	V	X		vmaxu	000110	V		vredmaxu	000110	V	F	vfmax
000111	V	X		vmax	000111	V		vredmax	000111	V		vfredmax
001000					001000	V	X	vaaddu	001000	V	F	vfsgnj
001001	V	X	I	vand	001001	V	X	vaadd	001001	V	F	vfsgnjn
001010	V	X	I	vor	001010	V	X	vasubu	001010	V	F	vfsgnjx
001011	V	X	I	vxor	001011	V	X	vasub	001011			

001100	V	X	I	vrgather	001100				001100			
001101					001101				001101			
001110		X	I	vslideup	001110		X	vslide1up	001110		F	vfslide1up
001111		X	I	vslidedown	001111		X	vslide1down	001111		F	vfslide1down

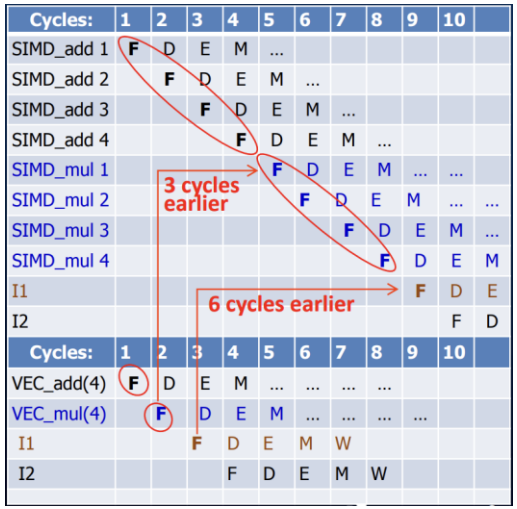
funct6					funct6				funct6			
010000	V	X	I	vadc	010000	V		VWXUNARY0	010000	V		VWFUNARY0
					010000		X	VRXUNARY0	010000		F	VRFUNARY0
010001	V	X	I	vmadc	010001				010001			
010010	V	X		vsbc	010010				010010			
010011	V	X		vmsbc	010011				010011			
010100					010100	V		VMUNARY0	010100			
010101					010101				010101			
010110					010110				010110			
010111	V	X	I	vmerge/vmv	010111	V		vcompress	010111		F	vfmerge.vf/vfmv
011000	V	X	I	vmseq	011000	V		vmandnot	011000	V	F	vmfeq
011001	V	X	I	vmsne	011001	V		vmand	011001	V	F	vmfle
011010	V	X		vmsltu	011010	V		vmor	011010			
011011	V	X		vmslt	011011	V		vmxor	011011	V	F	vmflt
011100	V	X	I	vmsleu	011100	V		vmornot	011100	V	F	vmfne
011101	V	X	I	vmsle	011101	V		vmnand	011101		F	vmfgt

RVV vs. GPU

- RVV offers several advantages, making it a favorable choice in certain contexts compared to GPUs
  - **Energy Efficiency**: RVV is well-suited for power-sensitive applications like mobile devices and embedded systems
  - **Versatility**: RVV is a general-purpose vector extension suitable for diverse applications, including scientific computing, image / signal processing
  - **Low Latency**: In specific scenarios, RVV may offer lower operation latency, crucial for real-time applications such as control systems
  - **Hardware Simplicity**: RVV may feature a simpler hardware implementation
  - **Open Standard**: RISC-V is an open standard not bound by patents
  - **Software Support**: The evolving RISC-V ecosystem has extensive support with open-source compilers, libraries, and tools for developers optimizing software for RVV

RVV vs. SIMD

- Assume the same computation width in hardware
  - SIMD_add: do 4 “8+8” in 1 cycle
  - SIMD_mul: do 4 “8*8” in 1 cycle
  - VEC_add(N): do N SIMD_add, 1 per cycle
  - VEC_mul(N): do N SIMD_mul, 1 per cycle



Feature	Intel AVX-512	Arm SVE	NEC SX-Aurora TSUBASA	RISC-V Vector
Is the vector register size defined by the architecture?	**Yes**. 512 bit VLE extension allows using 128-bit (SSE) and 256-bit (AVX-2) registers.	**No**. From 128 bit to 208 bits (in multiples of 128 bits)	**Yes**. Current generation is 16,384 bits.	**No**. Powers of two, from 64/128 up to 65,536 bits.
Predication/Masking	**Yes**. 8 mask registers `k0-k7` (`k0` hardcoded to all ones)	**Yes**. 16 vector predicate registers `p0-p15`. (`p0-p7` masking, `p8-p15` loops)	**Yes**. 16 vector mask registers.	**Yes**. Only `v0` as an implicit operand if the instruction is masked.
Set vector length	**No**	**No** (Privileged, compatibility-only)	**Yes**	**Yes**

Applications of Vector Processors

- Computation Kernels (Matrix Multiply, FFT, Sort)
- Cryptography (RSA, DES/IDEA, SHA/MD5)
- Machine/Deep Learning, AR/VR
- Multimedia Processing (Graphics, Image, Video)
- Networking (memcpy, memset)
- Scientific Computing (Modeling and Analysis)

The Vector Processor integrates features from both the CPU and GPU, creating a unified processing unit !
=> This enhances the flexibility of CPU instruction processing for data and enables dynamic adjustment of parallelism
=> However, the trade-off involves a more complex, larger hardware design, and the need for the reintegration of software and hardware interfaces

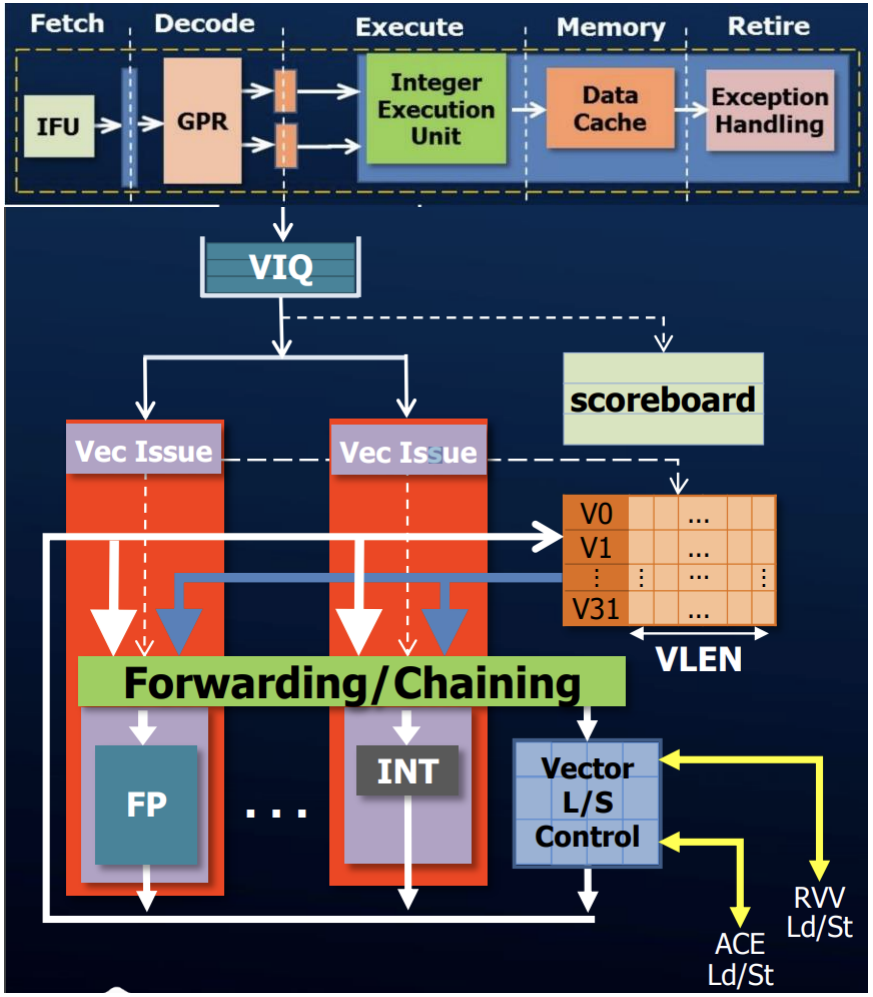
CRAY-1

- The CRAY-1, introduced in 1976 by the American supercomputer company Cray Research, was a pioneering supercomputer

- CRAY-1 Key Features:
  - Vector Processing: Featuring a vector processing architecture
  - Distinctive Design: Known for its iconic cylindrical shape
  - High Performance: Achieved remarkable speeds through parallel vector processing
  - 64-Bit Processor

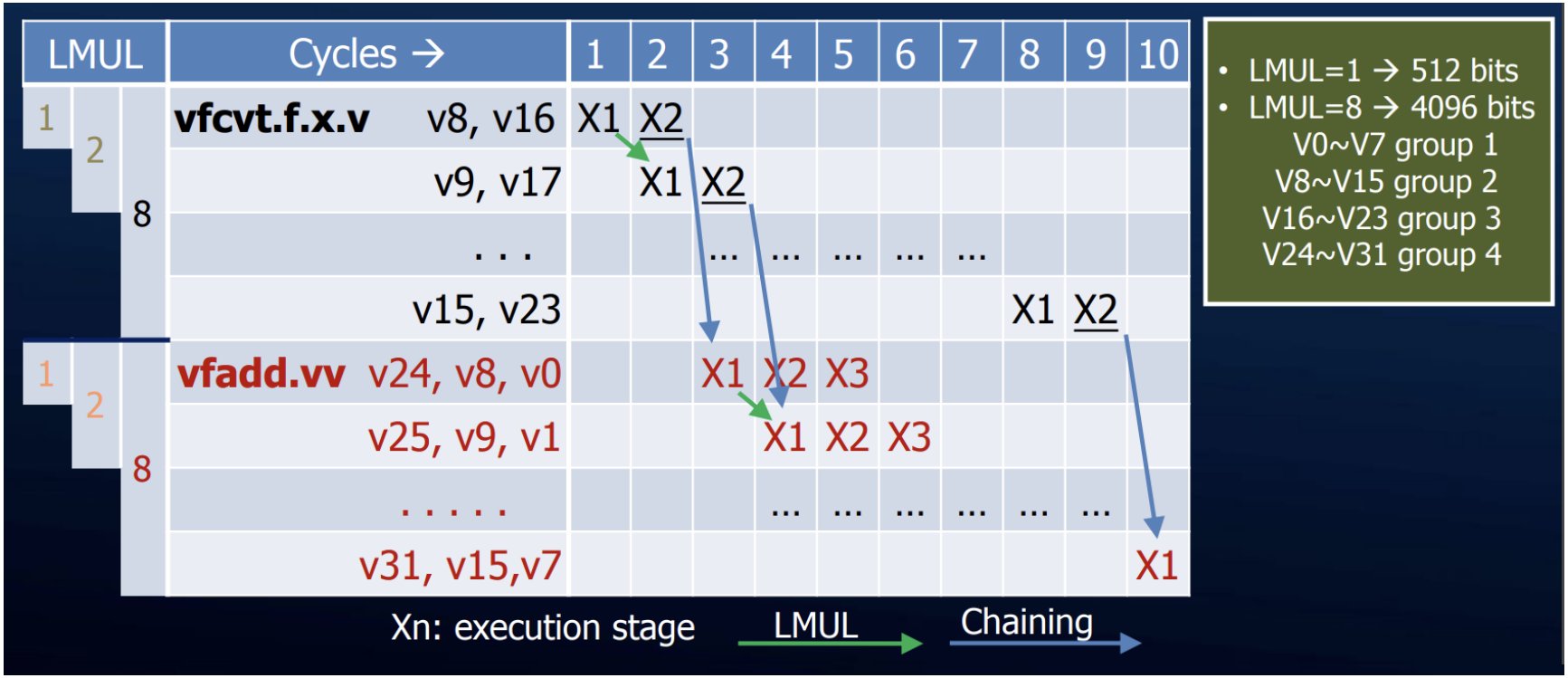
Andes VPU Microarchitecture [1]

- Supporting the latest RVV spec
- Data formats:
  - Standard: int8~int64, fp16~fp64
  - Andes-extended: bfloat16 and int4
- VLEN & SIMD width: 128, 256, 512
- Vector compute instructions:
  - Start execution after retired
  - Chainable, and most fully pipelined
  - Multiple Functional Units operating independently



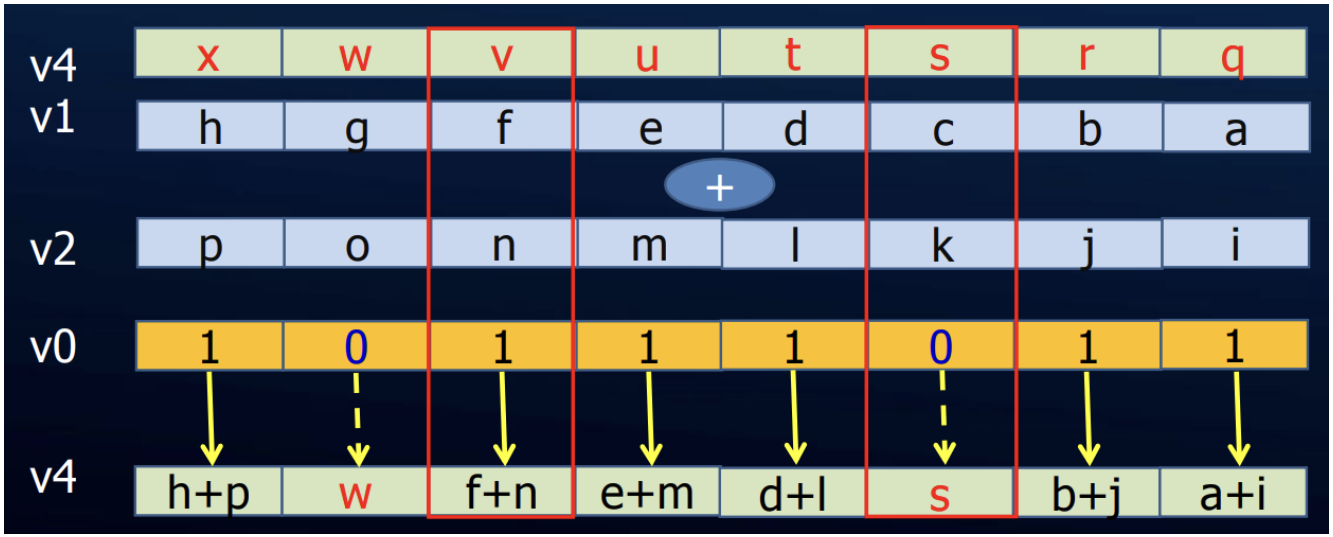
A Chaining Example with LMUL=8

- LMUL can be dynamically adjusted to change the parallelism



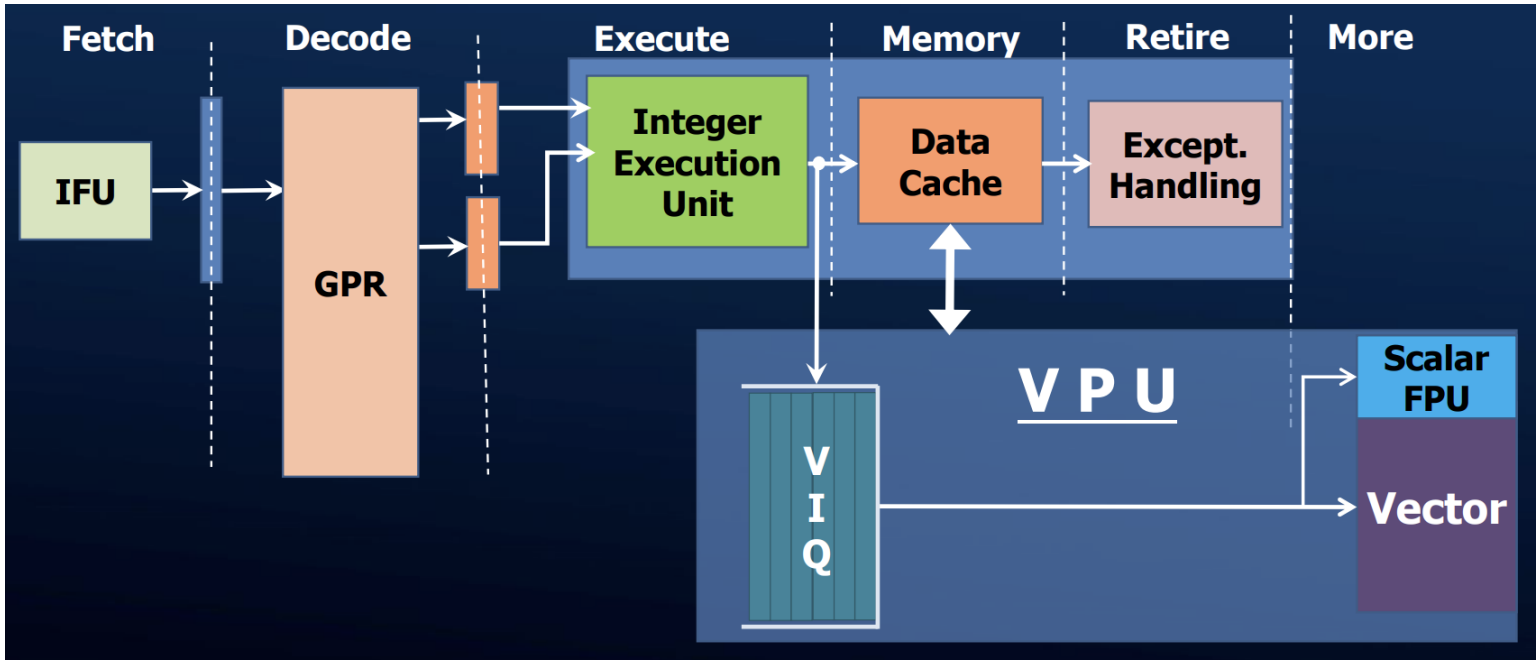
Vector Mask Example

- Only the least-significant bit of each element of the mask vector v0 is used to control execution
  - vadd v4, v2, v1, v0.t
  - v0.t : enable the update if v0[i].LSB=1



NX27V Pipeline

- It enables the simultaneous processing of multiple data elements using a single instruction
- NX27V FPGA is designed with a 5-stage pipeline
  - Providing a streamlined flow for instruction execution, and it includes a Vector Processing Unit (VPU) to handle vectorized data efficiently



NX27V Speedups

- Compared to pure C scalar code compiled with high optimization
  - Both vector and scalar code ran on the NX27V FPGA with 512-bit VLEN, 256-bit bus

Functions	Speedup[1]
F32 basic mathematical functions	19X
RGB CNN functions	18X
Depthwise CNN functions	18X
Pointwise CNN functions	21X
F32 filtering functions	19X
Q7 filtering functions	39X
F32 32x32x32 matrix multiplication	57X

- NX27V is suitable for a variety of applications, including those requiring parallelism and vectorized computations

Conclusion

- Performing consecutive matrix multiplications and managing data movement efficiently are crucial for NN applications!
- Specialized hardware or customized ISA extensions are often explored to optimize neural network computations !
- RISC-V Vector Extension (RVV) aims at providing scalable, configurable vector computation capabilities to the RISC-V architecture
- The Vector Processor integrates features from both the CPU and GPU, creating a unified processing unit !
- The Andes VPU microarchitecture can provide 18 to 57 times speedups