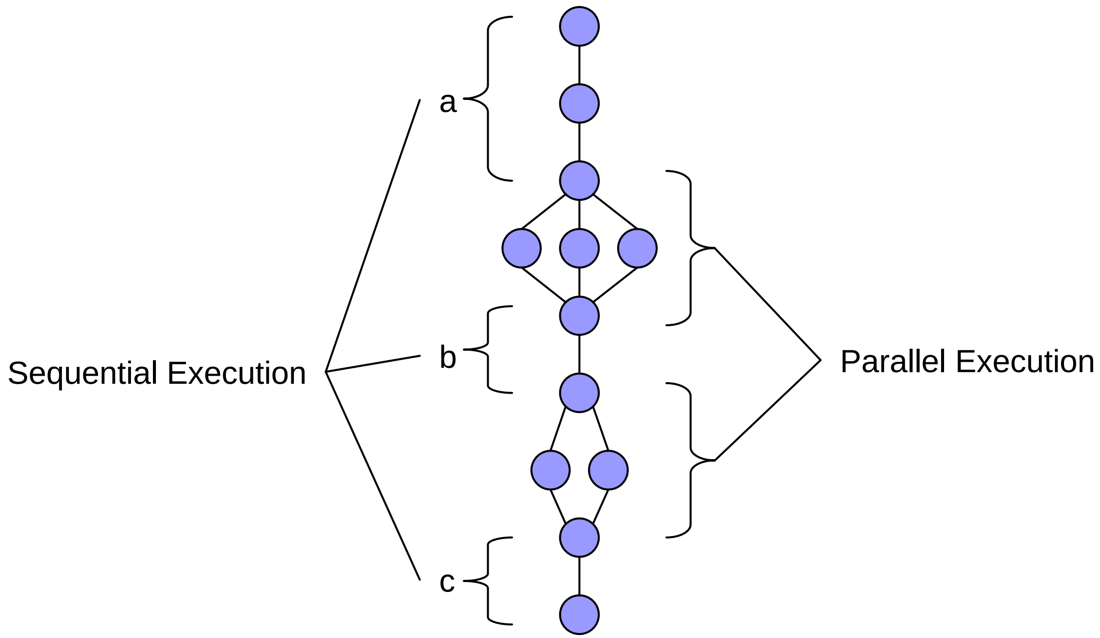
CDA 4150 Lecture 4

Vector Processing

CRAY like machines

Amdahl's Law



$$\alpha = a + b + c$$

$$T_S = \text{Time Spent in Sequential Processing}$$

$$T_P = \text{Time Spent in Parallel Processing}$$

$$S_P = \text{Speedup}$$

$$P = \text{Number of Processors}$$

Amdahl's Law (cont.)

$$S_p = \frac{T_s}{T_p}$$

$$T_p = \alpha T_s + \frac{(1-\alpha)T_s}{P}$$

$$S_p = \frac{T_s}{\alpha T_s + \frac{(1-\alpha)T_s}{P}}$$

$$S_p = \frac{1}{\alpha + \frac{(1-\alpha)}{P}}$$

$$S_p = \frac{P}{P\alpha + (1-\alpha)}$$

$$S_p = \frac{1}{\frac{1}{P} + \left(1 - \frac{1}{P}\right)\alpha}$$

$$\lim_{P\to\infty} S_p = \lim_{P\to\infty} \frac{1}{\frac{1}{P} + \left(1 - \frac{1}{P}\right)\alpha}$$

$$\lim_{P\to\infty} S_p = \frac{1}{\alpha}$$

Amdahl's Law (revisited)

$$Speedup = \frac{1}{\frac{1}{p} + \left(1 - \frac{1}{p}\right)\alpha} \Rightarrow \lim_{p \to \infty} Sp = \frac{1}{\alpha}$$

- Using α as a function of n, where $\alpha(n) = \frac{1}{n}$, then

$$Speedup = \frac{p}{1+(p-1)\alpha(n)} = \lim_{n \to \infty} \frac{p}{1+(p-1)\frac{1}{n}} = p$$

An extension of Amdahl's Law in terms of a matrix multiplication equation (AX = Y).

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

$$y_1 = a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4$$

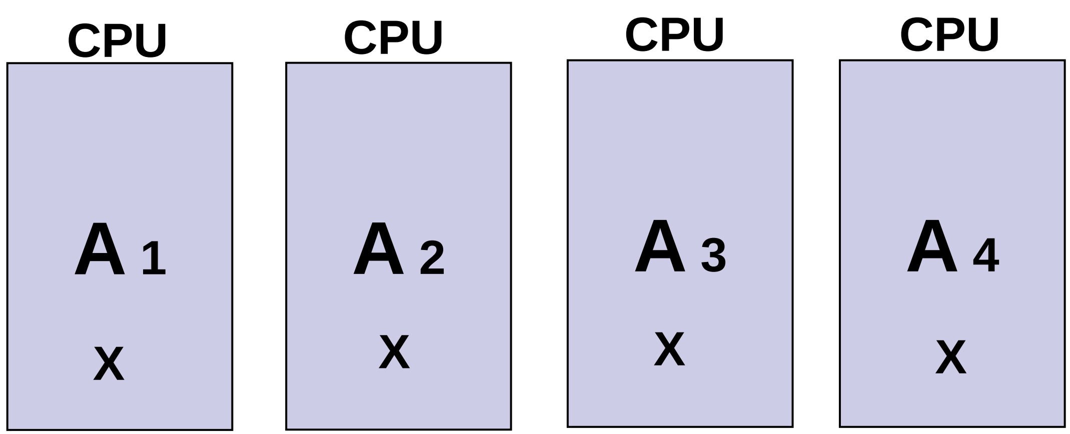
$$y_2 = a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4$$

$$y_3 = a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4$$

$$y_4 = a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4$$

$$\begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

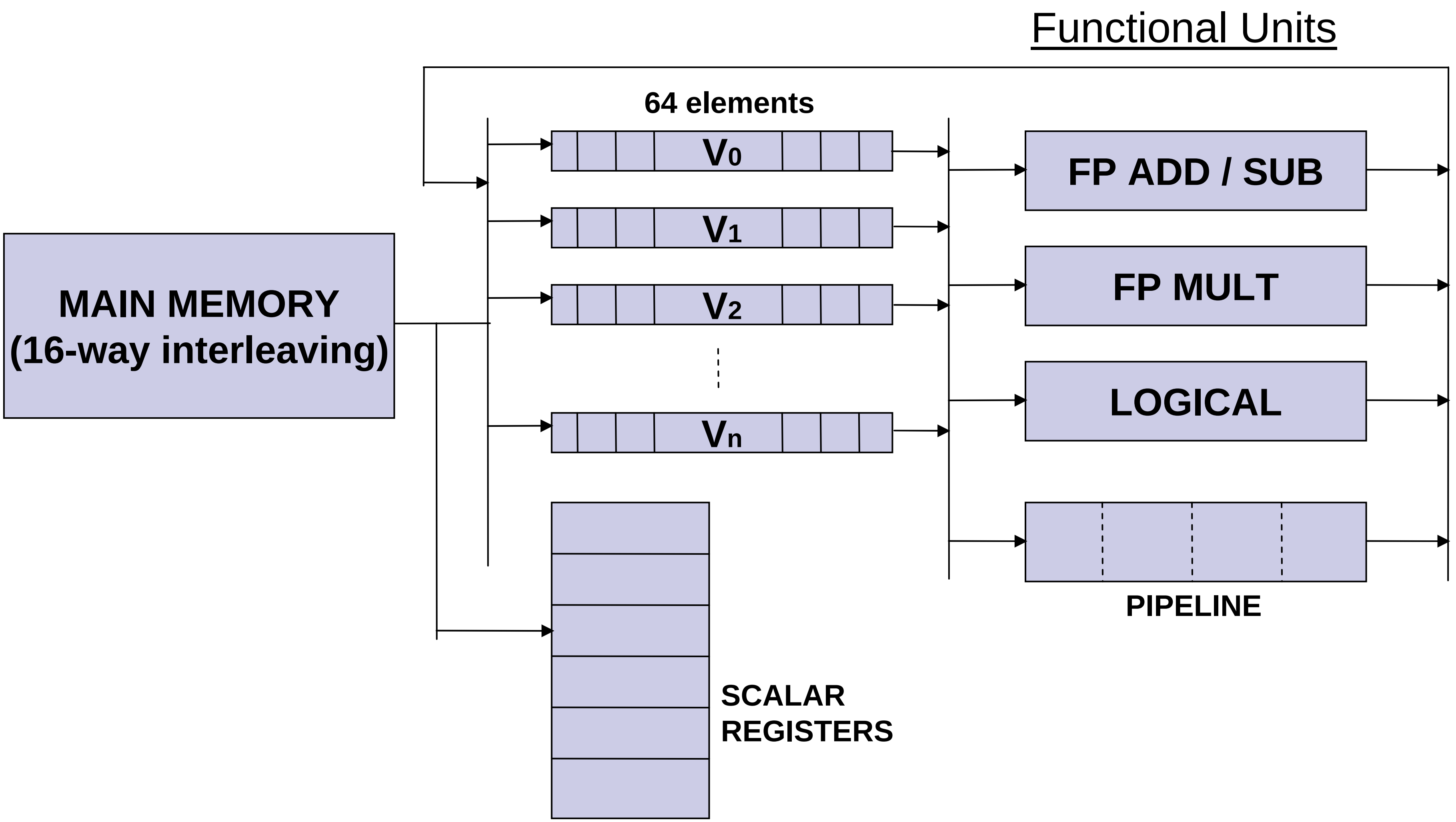
Compute each vector element in parallel by partitioning.



R. M. Russell, “The CRAY-1 Computer System”, CACM, vol. 21, pp. 63-72, 1978.

- Introduces CRAY-1 as a vector processing Architecture

CRAY -1


Functional Units
64 elements
V_0
V_1
V_2
V_n
MAIN MEMORY
(16-way interleaving)
FP ADD / SUB
FP MULT
LOGICAL
PIPELINE
SCALAR
REGISTERS

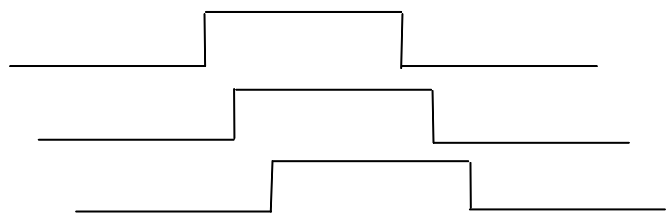
Instruction	Operation	Function
■ADDV	■V_1, V_2, V_3	■$V_1 \leftarrow V_2 + V_3$
■ADDSV (add scalar vector)	■V_1, F_0* , V_2	■$V_1 \leftarrow V_2 + F_0$
■MULTV	■V_1, V_2, V_3	■$V_1 \leftarrow V_2 + V_3$
■LV (load vector)	■V_1, R_1	■Load V_1 with memory address location starting at address [R_1]
■SV (store vector)	■R_1, V_1	■Store V_1 into memory starting at location [R_1]

* F_0 – a floating point number

NOTE: Each vector register (R_n) holds floating point numbers.

Timing

- A pipeline machine can initiate several instructions within 1 clock tick, which are then being executed in parallel.



- Related Concepts:

  - Convoys
  - Chimes

Convoy

- The set of vector instructions that could potentially begin execution together in one clock period.

- Example:

LV	V_1, R_X	→	Load vector X
MULTSV	V_2, F_0, V_1	→	Vector scalar multiplication
LV	V_3, R_Y	→	Load vector X
ADDV	V_4, V_2, V_3	→	Add
SV	R_Y, V_4	→	Storing results

Convoy

Note: MULTSV V_2, F_0, V_1 || LV V_3, R_Y is an example of a convoy, where 2 independent instructions are initiated within same chime.

LV	V_1, R_X	Load vector X
MULTSV	V_2, F_0, V_1	Vector scalar multiplication
LV	V_3, R_Y	Load vector X
ADDV	V_4, V_2, V_3	Add
SV	R_Y, V_4	Storing results

Chime

- Not a specific amount of time, but rather a timing concept representing the number of clock periods required to complete a vector operation.

- CRAY-1 chime is 64 clock periods.

  - Note: CRAY-1 clock cycle takes 12.5 ns.

  5 chimes would take : 5 * 64 * 12.5 = 4000 ns

Chime – Example #1

- How many chimes will the vector sequence take?

LV	V_1, R_X	→	Load vector X
MULTSV	V_2, F_0, V_1	→	Vector scalar multiplication
LV	V_3, R_Y	→	Load vector Y
ADDV	V_4, V_2, V_3	→	Add
SV	R_Y, V_4	→	Store result

Chime - Example #1

- ANSWER: 4 chimes

1st chime : LV V_1, R_X
2nd chime : MULTSV V_2, F_0, V_1 || LV V_3, R_Y
3rd chime : ADDV V_4, V_2, V_3
4th chime : SV R_Y, V_4

Note: MULTSV V_2, F_0, V_1 || LV V_3, R_Y
is an example of a convoy, where 2 independent instructions are initiated within same chime.

Chime - Example #2

- CRAY-1

  For I $\leftarrow$ 1 to 64

  A[I] = 3.0 * A[I] + (2.0 + B[I]) * C[I]

- To execute this:

  1st chime : $V_0 \leftarrow A$

  2nd chime : $V_1 \leftarrow B$

  $V_3 \leftarrow 2.0 + V_1$

  $V_4 \leftarrow 3.0 * V_0$

  3rd chime : $V_5 \leftarrow C$

  $V_6 \leftarrow V_3 * V_5$

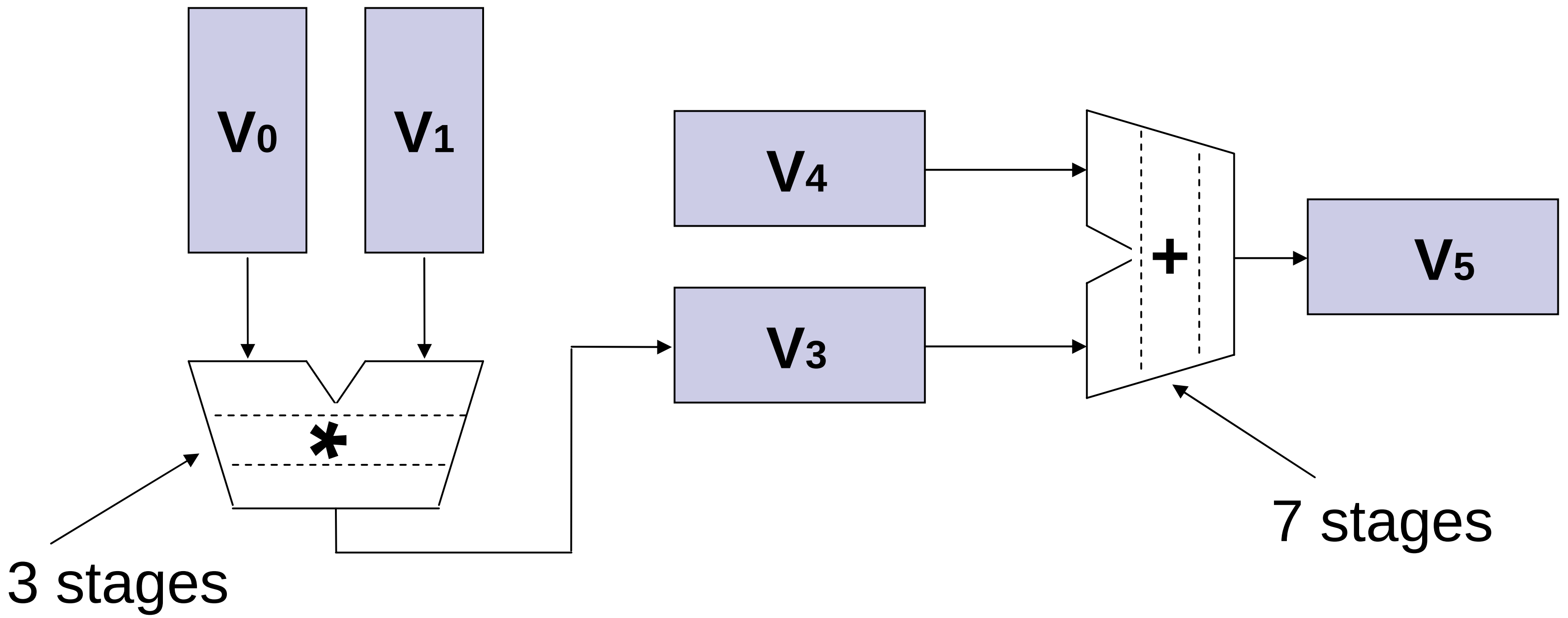
  $V_7 \leftarrow V_4 + V_6$

  4th chime : $A \leftarrow V_7$

Can initiate operations to use array values immediately after they have been loaded into vector registers.

Chaining

- Building dynamically a larger pipeline by increasing number of stages.



Chaining – Example #1

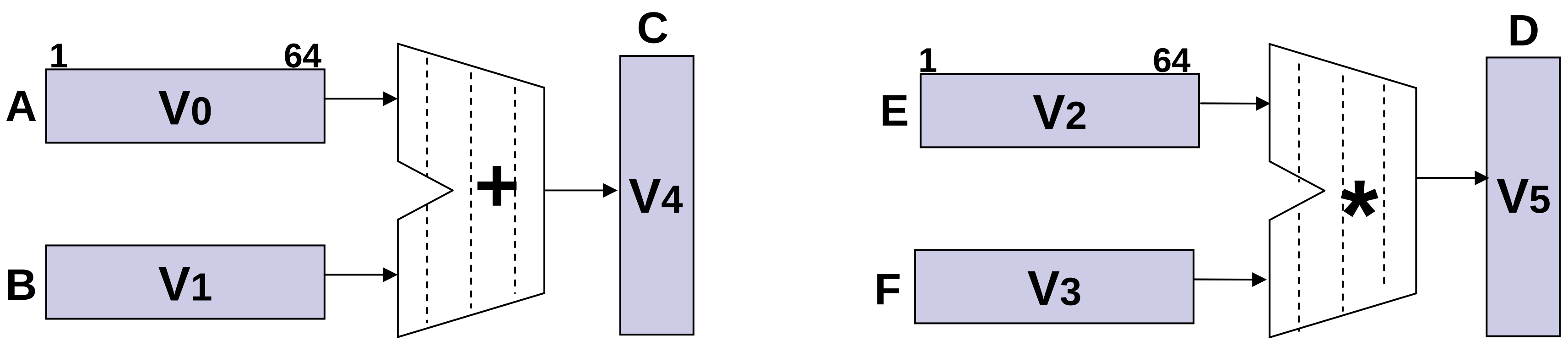
- For J ← 1 to 64

  C[J] ← A[J] + B[J]

  D[J] ← F[J] * E[J]

  END

* No chaining - these are independent!!



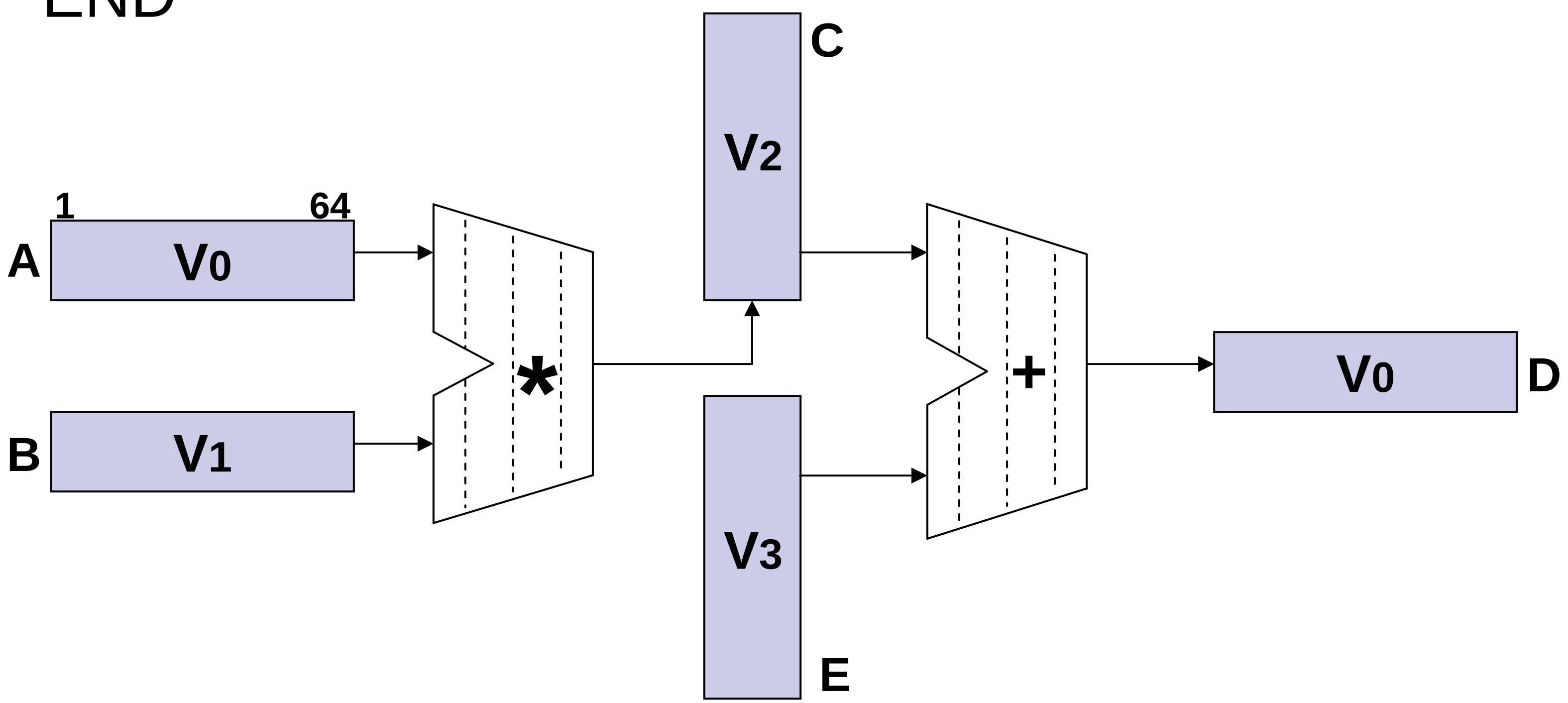
Chaining – Example #2

- For J ⟵ 1 to 64

  C[J] ⟵ A[J] * B[J]

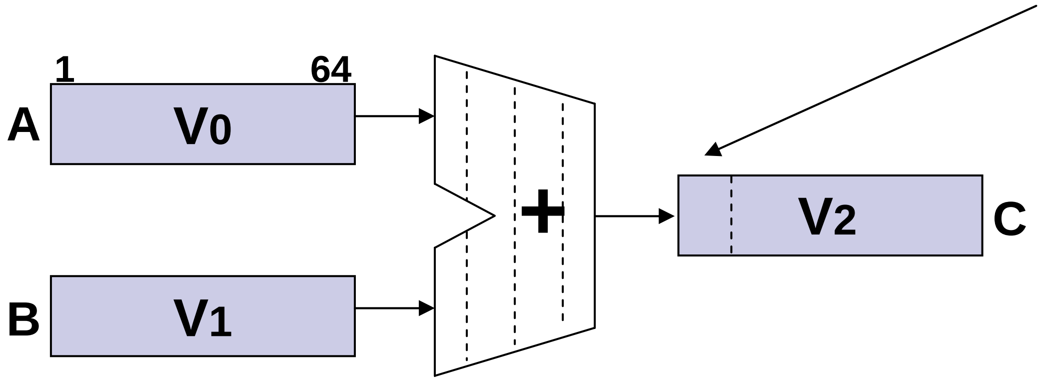
  D[J] ⟵ C[J] * E[J]

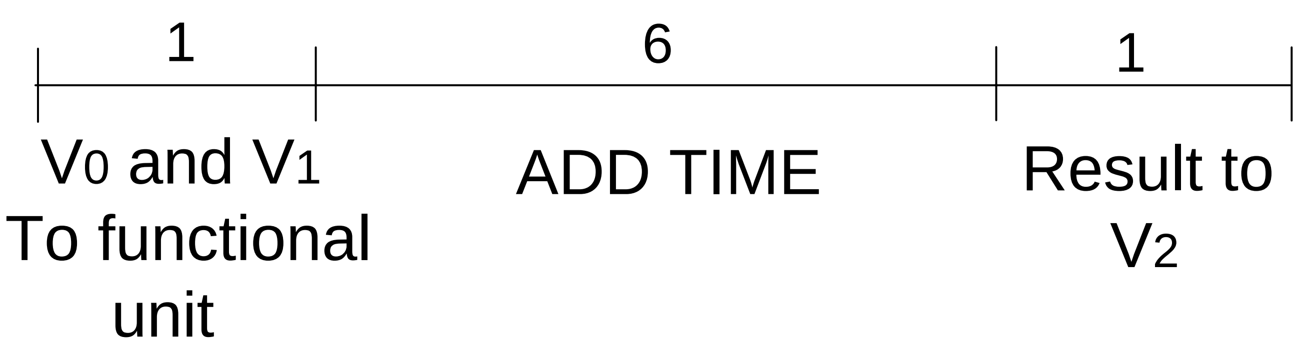
  END



Latency

It takes 8 units to get the result to here

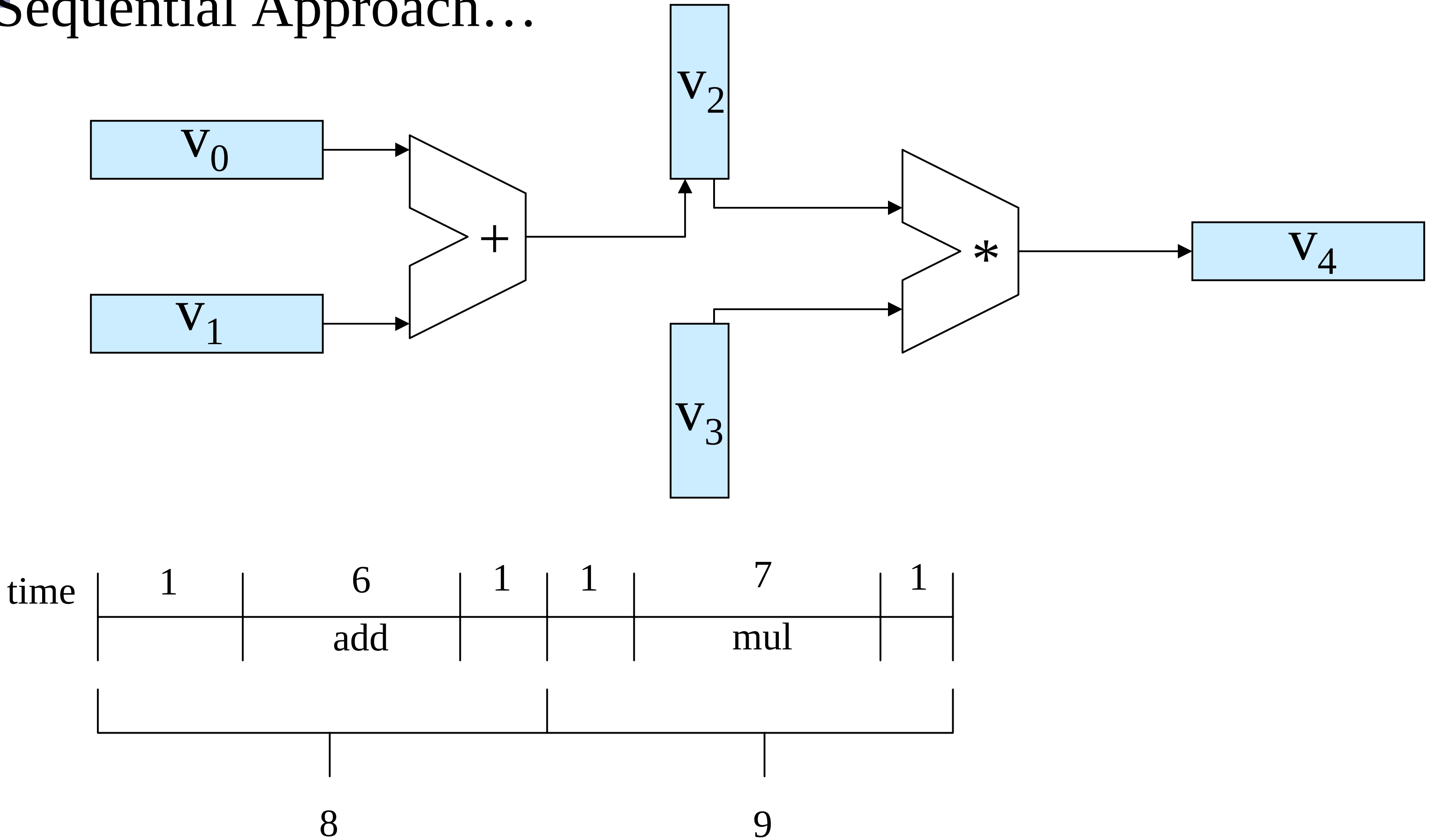




More Chaining and Storing Matrices

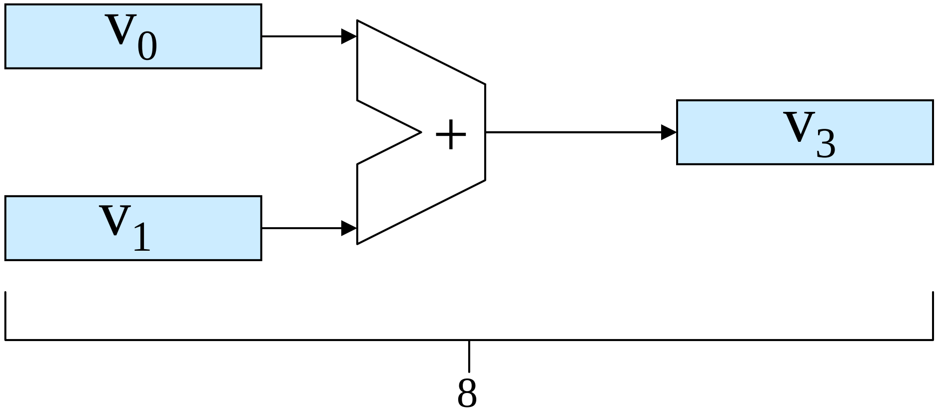
Thanks to Dusty Price

Sequential Approach…



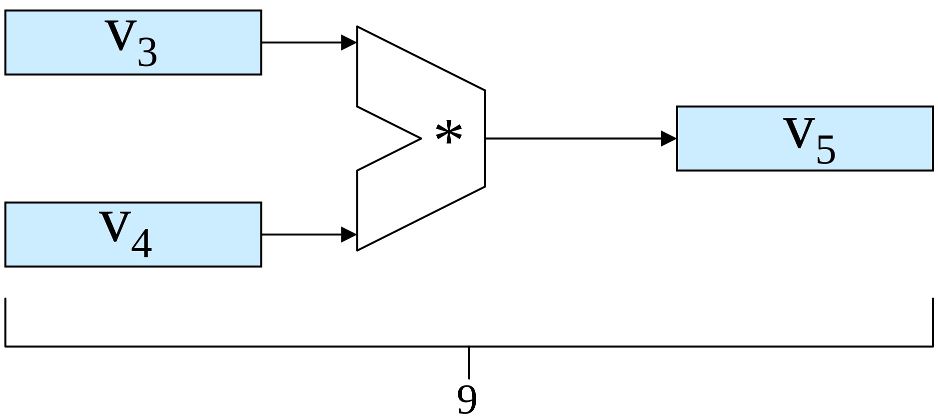
64 Elements in sequence: $T_s = 64 * (8 + 9) = 1088$

Using Pipeline Approach…



Using pipelining it takes 8 units of time to fill pipeline and produce first result, each unit of time after that produces another result

$T_{p+} = 8 + 63$

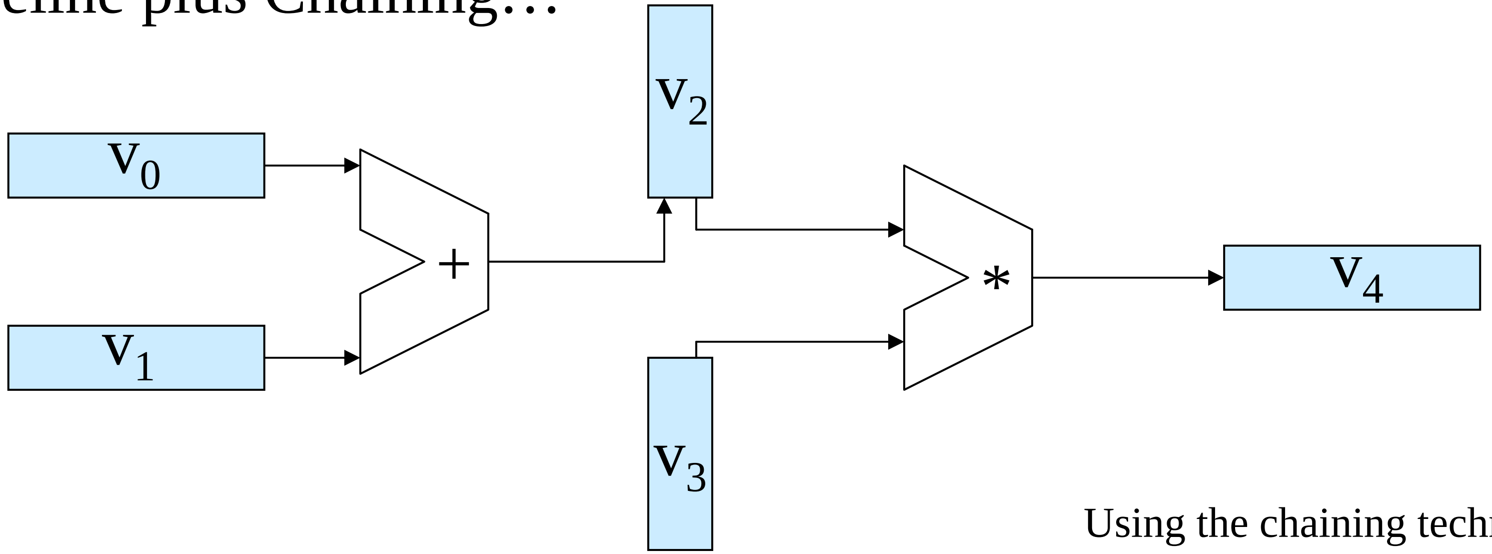


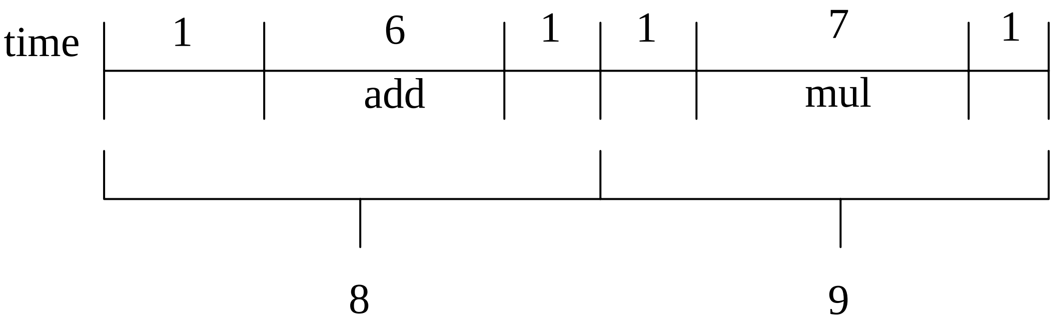
The multiplication pipeline takes 9 units of time to fill, and produces another result after each additional unit of time

$T_{p*} = 9 + 63$

The combination of the two $T_p = T_{p+} + T_{p*} = 8 + 63 + 9 + 63 = 143$

Pipeline plus Chaining…





Using the chaining technique, we now have one pipeline. This new pipeline takes 17 units of time to fill, and produces another result after each unit of time.

$T_c = 17 + 63 = 80$

Operation using Chaining $T_c = 17 + 63 = 80$

Review of time differences in the three approaches…

Sequential: $T_s = 17 * 64 = \boxed{1088}$

Pipelining: $T_p = 8 + 63 + 9 + 63 = \boxed{143}$

Chaining: $T_c = 17 + 63 = \boxed{80}$

Storing Matrixes for Parallel Access (Memory Interleaving)

Matrix

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{44}

4 Memory Modules

M_1	M_2	M_3	M_4
A_{11}	A_{21}	A_{31}	A_{41}
A_{12}	A_{22}	A_{32}	A_{42}
A_{13}	A_{23}	A_{33}	A_{43}
A_{14}	A_{24}	A_{34}	A_{44}

One column of the matrix can be accessed in parallel.

Storing the Matrix by Column…

Matrix

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{44}

4 Memory Modules

M_1	M_2	M_3	M_4
A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{44}

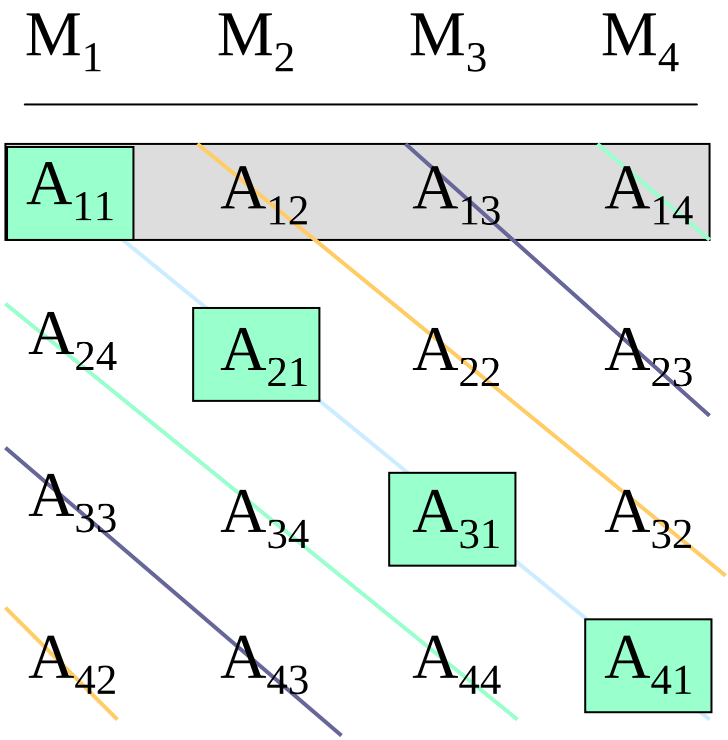
One Row can be accessed in parallel with this storage technique.

Sometimes we need to access both rows and columns fast…

Matrix

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{44}

4 Memory Modules



By using a skewed matrix representation, we can now access each row and each column in parallel.

Sometimes we need access to the main diagonal as well as rows and columns…

Matrix

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{44}

5 Memory Modules

M_1	M_2	M_3	M_4	M_5
A_{11}	A_{12}	A_{13}	A_{14}	
	A_{21}	A_{22}	A_{23}	A_{24}
A_{34}		A_{31}	A_{32}	A_{33}
A_{43}	A_{44}		A_{41}	A_{42}

At the cost of adding another memory module and wasted space, we can now access the matrix in parallel by row, column, and main diagonal.

Program Transformation

```
FOR I ← 1 TO n do
    X ← A[I] + B[I]
    .
    .
    Y[I] ← 2 * X
    .
    .
    X ← C[I] / D[I]      ← data dependency
    .
    .
    P ← X + 2
ENDFOR
```

⇒ removes data dependency

```
FOR I ← 1 TO n do
    X ← A[I] + B[I]
    .
    .
    Y[I] ← 2 * X
    .
    .
    XX ← C[I] / D[I]
    .
    .
    P ← XX + 2
ENDFOR
```

Scalar Expansion

```
FOR I ← 1 TO n do
    X ← A[I] + B[I]
    .
    .
    Y[I] ← 2 * X
    .
    .
ENDFOR
```

data dependency

⇒

```
FOR I ← 1 TO n do
    X ← A[I] + B[I]
    .
    .
    Y[I] ← 2 * X[I]
    .
    .
ENDFOR
```

removes data dependency

Loop Unrolling

```
FOR I ← 1 TO n do
    X[I] ← A[I] * B[I]
ENDFOR
```

⇒

```
X[1] ← A[1] * B[1]
X[2] ← A[2] * B[2]
.
.
X[n] ← A[n] * B[n]
```

Loop Fusion or Jamming

```
FOR I ← 1 TO n do
   X[I] ← Y[I] * Z[I]
ENDFOR
FOR I ← 1 TO n do
   M[I] ← P[I] + X[I]
ENDFOR
```

⇒

```
a)  FOR I ← 1 TO n do
       X[I] ← Y[I] * Z[I]
       M[I] ← P[I] + X[I]
    ENDFOR
b)  FOR I ← 1 TO n do
       M[I] ← P[I] + Y[I] * Z[I]
    ENDFOR
```