Static Compiler Optimization Techniques

- We examined the following static ISA/compiler techniques aimed at improving pipelined CPU performance:
  - Static pipeline scheduling.
  - Loop unrolling.
  - Static branch prediction.
  - Static multiple instruction issue: VLIW.
  - Conditional or predicted instructions/predication.
  - Static speculation

e.g. IA-64 (EPIC)

- Here we examine two additional static compiler-based techniques:
  - 1 – <u>Loop-Level Parallelism (LLP) analysis:</u> + relationship to Data Parallelism
    - Detecting and enhancing loop iteration parallelism
      - Greatest Common Divisor (GCD) test.
  - 2 – <u>Software pipelining (Symbolic loop unrolling).</u>
- In addition a brief introduction to vector processing (Appendix G) is included to emphasize the importance/origin of LLP analysis. } FYI

4th Edition: Appendix G.1-G.3, vector processing: Appendix F
(3rd Edition: Chapter 4.4, vector processing: Appendix G)

Data Parallelism & Loop Level Parallelism (LLP)

- **Data Parallelism:** Similar independent/parallel computations on different elements of arrays that usually result in independent (or parallel) loop iterations when such computations are implemented as sequential programs.
- A common way to increase parallelism among instructions is to exploit data parallelism among independent iterations of a loop (e.g exploit Loop Level Parallelism, LLP).
  - One method covered earlier to accomplish this is by unrolling the loop either statically by the compiler, or dynamically by hardware, which increases the size of the basic block present. This resulting larger basic block provides more instructions that can be scheduled or re-ordered by the compiler/hardware to eliminate more stall cycles.

Usually: Data Parallelism → LLP

- The following loop has parallel loop iterations since computations in each iterations are data parallel and are performed on different elements of the arrays.

Example

```
for (i=1; i<=1000; i=i+1;)
        x[i] = x[i] + y[i];
```

4 vector instructions:

LV	Load Vector X
LV	Load Vector Y
ADDV	Add Vector X, X, Y
SV	Store Vector X

- In supercomputing applications, data parallelism/LLP has been traditionally exploited by vector ISAs/processors, utilizing vector instructions
  - Vector instructions operate on a number of data items (vectors) producing a vector of elements not just a single result value. The above loop might require just four such instructions.

Loop Unrolling Example

From Lecture #3 (slide # 11) ←

When scheduled for pipeline

Note:
Independent Loop Iterations
Resulting from data parallel
operations on elements of array X

for (i=1000; i>0; i=i-1)
x[i] = x[i] + s;

Usually: Data Parallelism → LLP

```
Loop:  L.D      F0, 0(R1)
       L.D      F6,-8 (R1)
       L.D      F10, -16(R1)
       L.D      F14, -24(R1)
       ADD.D    F4, F0, F2
       ADD.D    F8, F6, F2
       ADD.D    F12, F10, F2
       ADD.D    F16, F14, F2
       S.D      F4, 0(R1)
       S.D      F8, -8(R1)
       DADDUI   R1, R1,# -32
       S.D      F12, 16(R1),F12
       BNE      R1,R2, Loop
       S.D      F16, 8(R1), F16    ;8-32 = -24
```

The execution time of the loop has dropped to 14 cycles, or 14/4 = 3.5 clock cycles per element

compared to 7 before scheduling and 6 when scheduled but unrolled.

Speedup = 6/3.5 = 1.7

Unrolling the loop exposed more computations that can be scheduled to minimize stalls by increasing the size of the basic block from 5 instructions in the original loop to 14 instructions in the unrolled loop.

i.e more ILP exposed

Loop unrolling exploits data parallelism among independent iterations of a loop

Larger Basic Block ⟶ More ILP Exposed

Loop unrolled four times and scheduled

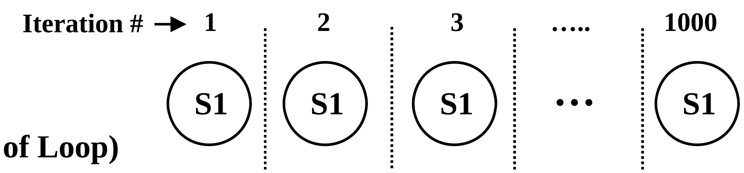
Loop-Level Parallelism (LLP) Analysis

- Loop-Level Parallelism (LLP) analysis focuses on whether data accesses in later iterations of a loop are data dependent on data values produced in earlier iterations and possibly making loop iterations independent (parallel).

e.g. in **for (i=1; i<=1000; i++)**

x[i] = x[i] + s; ← S1 (Body of Loop)

Usually: Data Parallelism → LLP



the computation in each iteration is independent of the previous iterations and the loop is thus parallel. The use of X[i] twice is within a single iteration.

⇒ Thus loop iterations are parallel (or independent from each other).

Classification of Date Dependencies in Loops:

Between iterations or inter-iteration

1. Loop-carried Data Dependence: A data dependence between different loop iterations (data produced in an earlier iteration used in a later one).
2. Not Loop-carried Data Dependence: Data dependence within the same loop iteration.

Within an iteration or intra-iteration

- LLP analysis is important in software optimizations such as loop unrolling since it usually requires loop iterations to be independent (and in vector processing).
- LLP analysis is normally done at the source code level or close to it since assembly language and target machine code generation introduces loop-carried name dependence in the registers used in the loop.
  - Instruction level parallelism (ILP) analysis, on the other hand, is usually done when instructions are generated by the compiler.

4th Edition: Appendix G.1-G.2 (3rd Edition: Chapter 4.4)

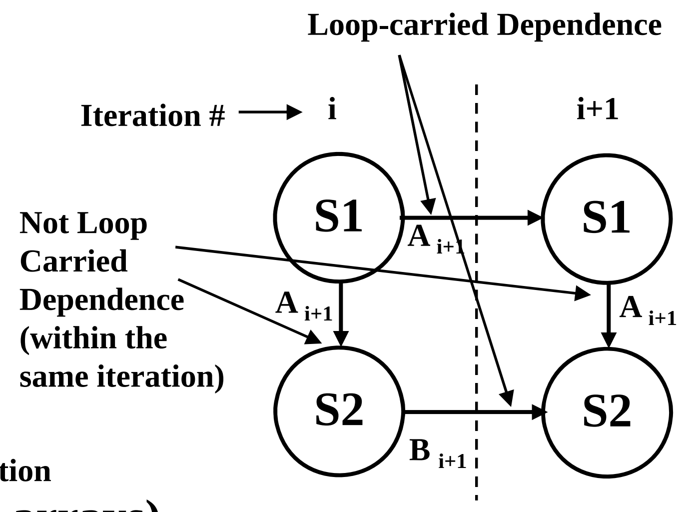
LLP Analysis Example 1

- **In the loop:**

```
for (i=1; i<=100; i=i+1)  {
        A[i+1] = A[i] + C[i];  /*  S1 */
        B[i+1] = B[i] + A[i+1];}  /* S2 */
}
```

Produced in previous iteration — Produced in same iteration

(Where A, B, C are distinct non-overlapping arrays)



Dependency Graph

- S2 uses the value A[i+1], computed by S1 in the same iteration. This data dependence is within the same iteration <u>(not a loop-carried data dependence).</u>

i.e.	S1 → S2	on	A[i+1]	Not loop-carried data dependence

  ⇒ does not prevent loop iteration parallelism.

- S1 uses a value computed by S1 in the earlier iteration, since iteration i computes A[i+1] read in iteration i+1 <u>(loop-carried dependence, prevents parallelism).</u> The same applies for S2 for B[i] and B[i+1]

  Loop-level

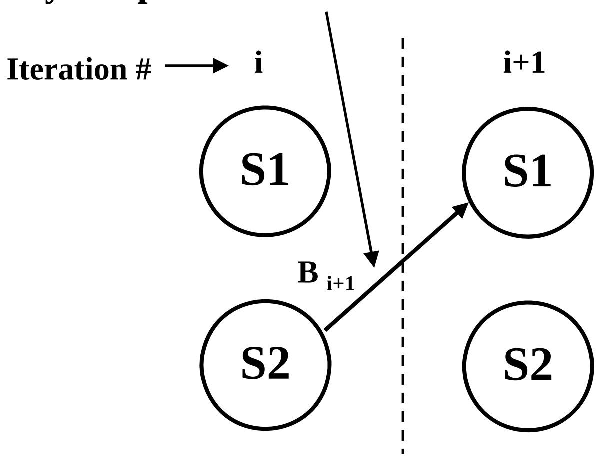
i.e.	S1 → S1	on	A[i]	Loop-carried data dependence
	S2 → S2	on	B[i]	Loop-carried data dependence

  ⇒These two data dependencies are loop-carried spanning more than one iteration (two iterations) preventing loop parallelism.

In this example the loop carried dependencies form two dependency chains starting from the very first iteration and ending at the last iteration

LLP Analysis Example 2

Dependency Graph



- In the loop:

```
for (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i];      /* S1 */
    B[i+1] = C[i] + D[i];    /* S2 */
}
```

  – S1 uses the value B[i] computed by S2 in the previous iteration (loop-carried dependence)

  i.e. S2 → S1 on B[i] Loop-carried data dependence

  – This dependence is not circular:

  And does not form a data dependence chain

    • S1 depends on S2 but S2 does not depend on S1.

  i.e. loop

  – Can be made parallel by replacing the code with the following:

```
A[1] = A[1] + B[1];              Very First S1 — Loop Start-up code
for (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];        S2
    A[i+1] = A[i+1] + B[i+1];    S1 (From Next Iteration)
}
B[101] = C[100] + D[100];        Very Last S2 — Loop Completion code
```

Parallel loop iterations (data parallelism in computation exposed in loop code)

→ Illustrated Next Slide

4th Edition: Appendix G.2 (3rd Edition: Chapter 4.4)

LLP Analysis Example 2

Original Loop:

```
for (i=1; i<=100; i=i+1) {
        A[i] = A[i] + B[i];        /* S1 */
        B[i+1] = C[i] + D[i];      /* S2 */
    }
```

	Iteration 1	Iteration 2	. . . . . .	Iteration 99	Iteration 100
S1	A[1] = A[1] + B[1];	A[2] = A[2] + B[2];	. . . . . .	A[99] = A[99] + B[99];	A[100] = A[100] + B[100];
S2	B[2] = C[1] + D[1];	B[3] = C[2] + D[2];	Loop-carried Dependence	B[100] = C[99] + D[99];	B[101] = C[100] + D[100];

Modified Parallel Loop:

(one less iteration)

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1)  {
       B[i+1] = C[i] + D[i];
       A[i+1] = A[i+1] + B[i+1];
 }
 B[101] = C[100] + D[100];
```

	Loop Start-up code	Iteration 1	. . . .	Iteration 98	Iteration 99
S1	A[1] = A[1] + B[1];	A[2] = A[2] + B[2];		A[99] = A[99] + B[99];	A[100] = A[100] + B[100];
S2	B[2] = C[1] + D[1];	B[3] = C[2] + D[2];	Not Loop Carried Dependence	B[100] = C[99] + D[99];	B[101] = C[100] + D[100];
					Loop Completion code

ILP Compiler Support: Loop-Carried Dependence Detection

For access to elements of an array

- To detect loop-carried dependence in a loop, **the Greatest Common Divisor (GCD) test** can be used by the compiler, which is based on the following:
- If an array element with index: $a \times i + b$ is **stored** (i.e written to) and element: $c \times i + d$ of the same array is **loaded** later where index runs from m to n, **a dependence exists** if the following **two conditions hold**:

1. There are two iteration indices, j and k, $m \leq j, k \leq n$ (i.e. within iteration limits m, n)
2. The loop **stores** into an array element indexed by:

$a \times j + b$ — Produce or write (store) element with this Index

and later **loads** from the same array the element indexed by:

$c \times k + d$ — **Later** read (load) element with this index

Thus:

$$a \times j + b = c \times k + d \qquad j < k$$

(i.e later iteration)

Index of element written (stored) earlier — Index of element read(loaded) later

Here a, b, c, d are constants

The Greatest Common Divisor (GCD) Test

- **If a loop carried dependence exists, then :**

For access to elements of an array in a loop

GCD(c, a) must divide (d-b)

The GCD test is <u>sufficient to guarantee no loop carried dependence</u>
However there are cases where GCD test succeeds but no dependence exits because GCD test <u>does not take loop bounds into account</u>

In an earlier iteration

Index of element stored:
a x i + b
Index of element loaded:
c x i + d

In a later iteration

Example:

```
for(i=1; i<=100; i=i+1) {
        x[2*i+3] = x[2*i] * 5.0;
}
```

+ 0

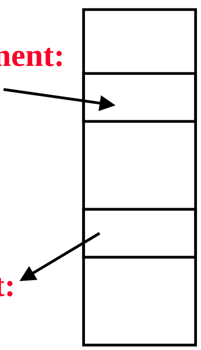
a = 2 b = 3 c = 2 d = 0

GCD(a, c) = 2

d - b = -3

2 does not divide -3 ⇒ No loop carried dependence possible.

Index of written element:
a x i + b = 2i + 3



Index of read element:
c x i + d = 2i

CMPE550 - Shaaban

4th Edition: Appendix G.2 (3rd Edition: Chapter 4.4)

Showing Last Example Loop Iterations to Be Independent

```
for(i=1; i<=100; i=i+1) {
        x[2*i+3] = x[2*i] * 5.0;
        }
```

Index of element stored:
a x i + b
Index of element loaded:
c x i + d

c x i + d **= 2 x i + 0**

a x i + b **= 2 x i + 3**

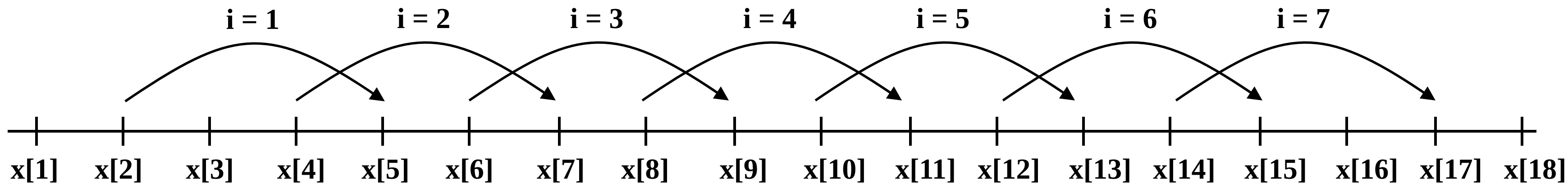
Iteration i	Index of x loaded	Index of x stored
1	2	5
2	4	7
3	6	9
4	8	11
5	10	13
6	12	15
7	14	17

a = 2   b = 3   c = 2   d = 0

GCD(a, c) = 2
d - b = -3
2 does not divide -3

⇒ No dependence possible.

What if GCD (a, c) divided d - b ?



For example from last slide

ILP Compiler Support:

Software Pipelining (Symbolic Loop Unrolling)

- A compiler technique where loops are reorganized:
  - Each new iteration is made from <u>instructions selected</u> from <u>a number of independent iterations</u> of the original loop.

i.e parallel iterations

Why?

- The instructions are selected to **separate dependent** instructions within the original loop iteration.

By one or more new iterations

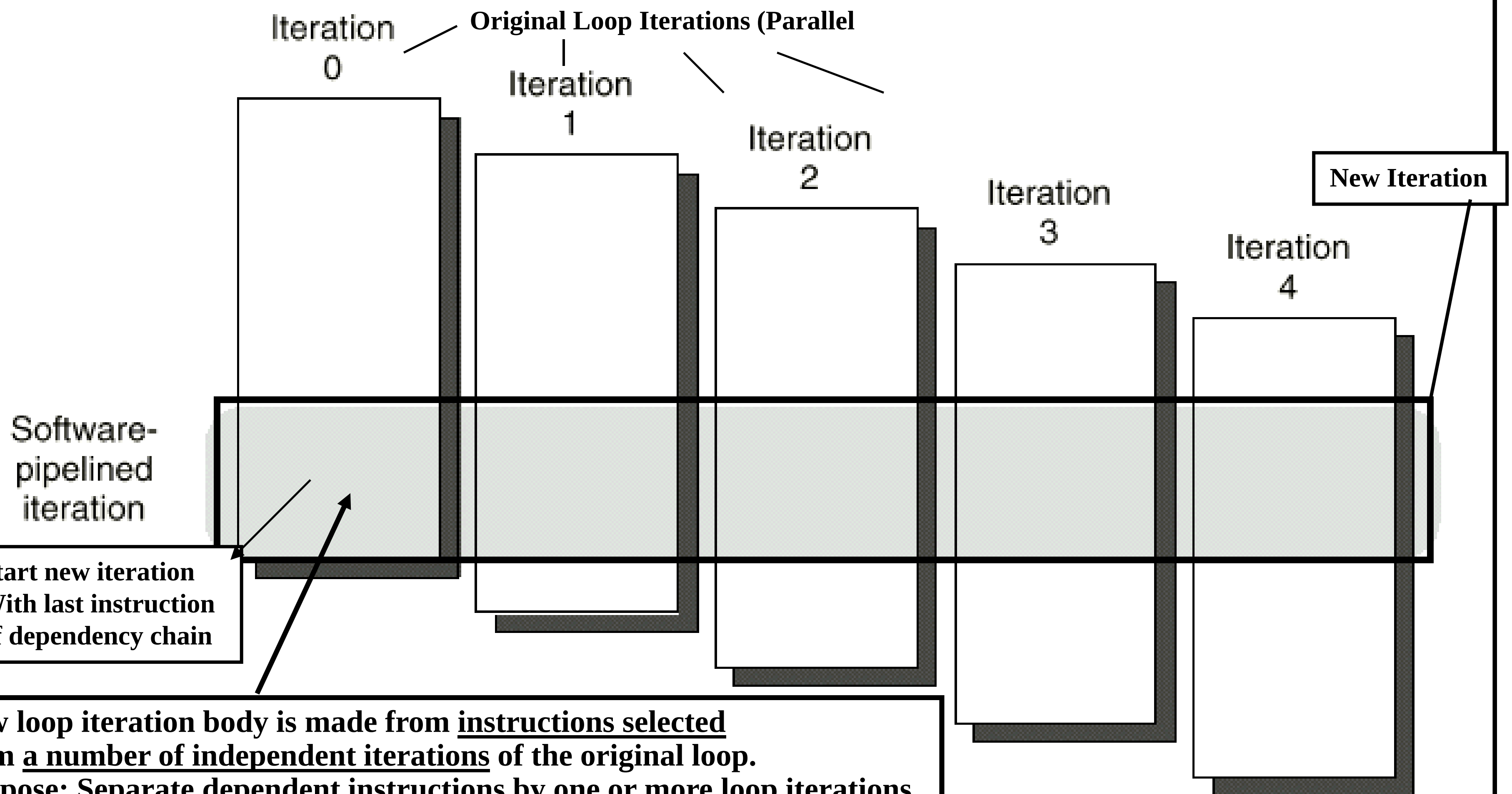
- No actual loop-unrolling is performed.
  - A software equivalent to the Tomasulo approach?
- <u>Requires:</u>

This static optimization is done at machine code level

  - Additional <u>start-up code</u> to execute code left out from the first original loop iterations.
  - Additional <u>finish code</u> to execute instructions left out from the last original loop iterations.

4th Edition: Appendix G.3 (3rd Edition: Chapter 4.4)

Software Pipelining (Symbolic Loop Unrolling)



New loop iteration body is made from <u>instructions selected</u> from <u>a number of independent iterations</u> of the original loop.
<u>Purpose:</u> Separate dependent instructions by one or more loop iterations.

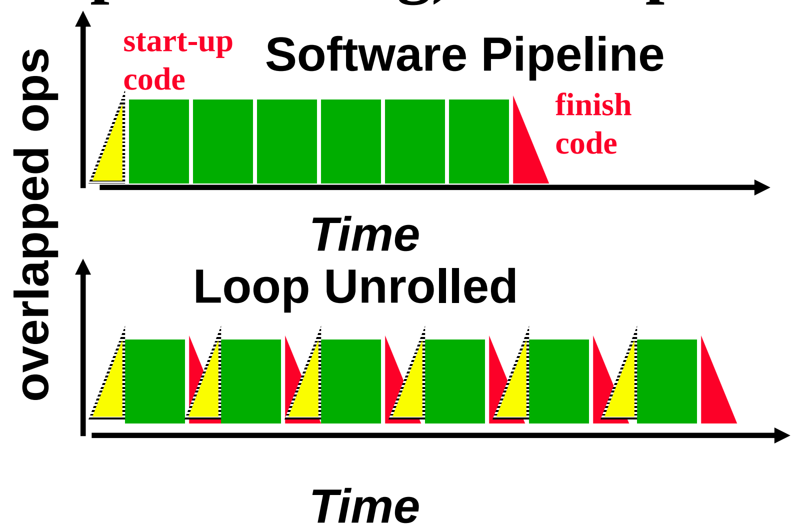
A software-pipelined loop chooses instructions from different loop iterations, thus separating the dependent instructions within one iteration of the original loop.

4th Edition: Appendix G.3 (3rd Edition: Chapter 4.4)

Software Pipelining (Symbolic Loop Unrolling) Example

Show a software-pipelined version of the code:

```
Loop:   L.D      F0,0(R1)
        ADD.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDUI   R1,R1,#-8
        BNE      R1,R2,LOOP
```



3 times because <u>chain</u> of dependence of <u>length 3 instructions</u> exist in body of original loop   i.e. L.D → ADD.D → S.D

Before: Unrolled 3 times

Iteration			
1	1	L.D	F0,0(R1)
1	2	ADD.D	F4,F0,F2
1	3	S.D	F4,0(R1)
2	4	L.D	F0,-8(R1)
2	5	ADD.D	F4,F0,F2
2	6	S.D	F4,-8(R1)
3	7	L.D	F0,-16(R1)
3	8	ADD.D	F4,F0,F2
3	9	S.D	F4,-16(R1)
	10	DADDUI	R1,R1,#-24
	11	BNE	R1,R2,LOOP

After: Software Pipelined Version

```
          L.D      F0,0(R1)      }
          ADD.D    F4,F0,F2      }  start-up code
          L.D      F0,-8(R1)     }
LOOP:  1  S.D      F4,0(R1)    ;Stores M[i]          ]
       2  ADD.D    F4,F0,F2    ;Adds to M[i-1]       ]
       3  L.D      F0,-16(R1);Loads M[i-2]           ]  New Loop
       4  DADDUI   R1,R1,#-8                         ]
       5  BNE      R1,R2,LOOP                        ]
          S.D      F4, 0(R1)     }
          ADDD     F4,F0,F2      }  finish code
          S.D      F4,-8(R1)     }
```

2 fewer loop iterations

No Branch delay slot in this example

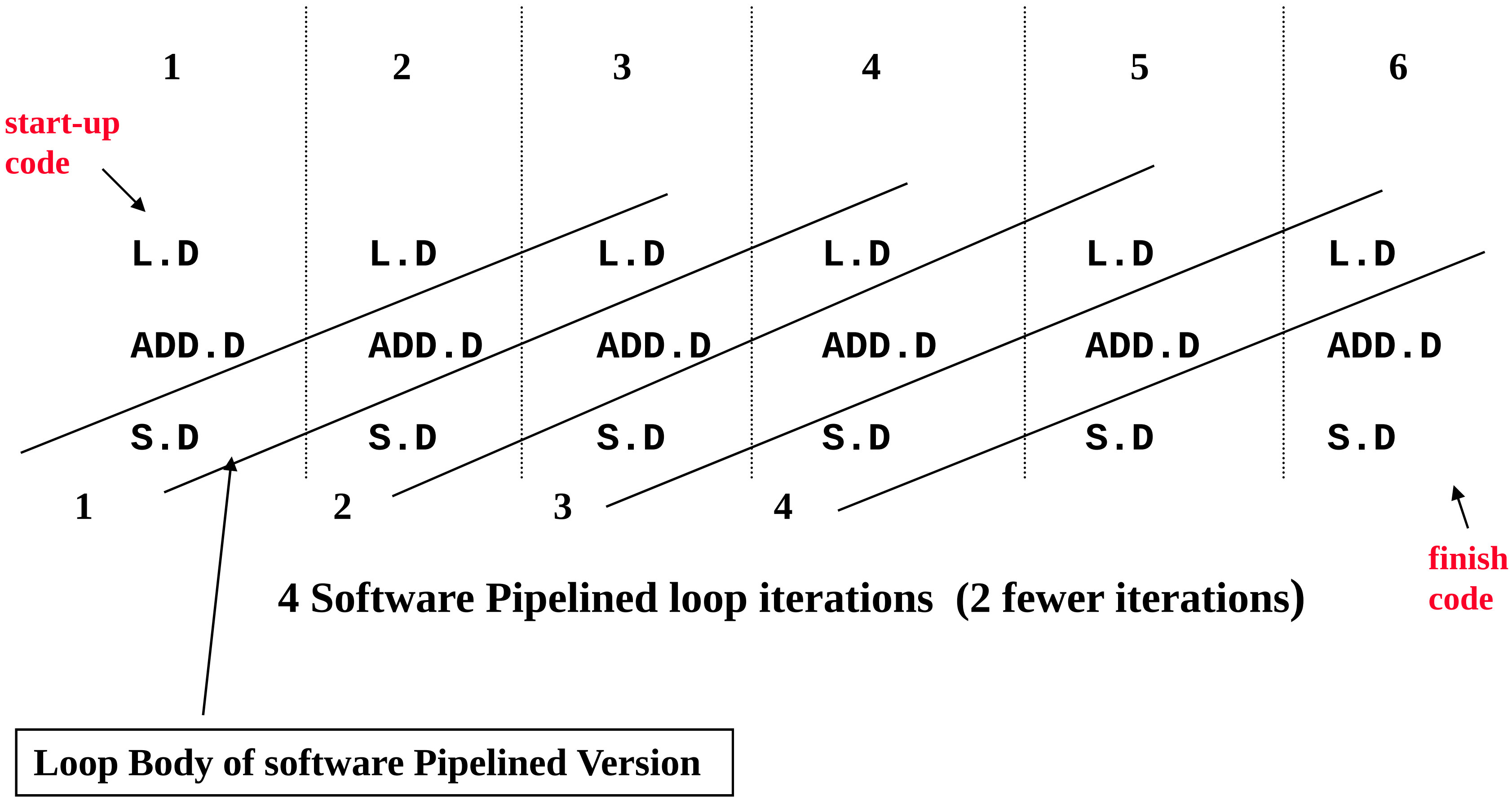
No actual loop unrolling is done (do not rename registers)

Software Pipelining Example Illustrated

Assuming 6 original iterations (for illustration purposes):

```
L.D      F0,0(R1)
ADD.D    F4,F0,F2
S.D      F4,0(R1)
```

Body of original loop



Problems with Superscalar approach

- **Limits to conventional exploitation of ILP:**

1) ***Pipelined clock rate*: Increasing clock rate requires deeper pipelines with longer pipeline latency which increases the CPI increase (longer branch penalty , other hazards).**
2) ***Instruction Issue Rate*: Limited instruction level parallelism (ILP) reduces actual instruction issue/completion rate. (vertical & horizontal waste)**
3) ***Cache hit rate*: Data-intensive scientific programs have very large data sets accessed with poor locality; others have continuous data streams (multimedia) and hence poor locality. (poor memory latency hiding).**
4) **Data Parallelism: Poor exploitation of data parallelism present in many scientific and multimedia applications, where similar independent computations are performed on large arrays of data (Limited ISA, hardware support).**

- **As a result, actual achieved performance is much less than peak potential performance and low computational energy efficiency (computations/watt)**

Flynn's 1972 Classification of Computer Architecture

SISD

- **Single Instruction stream over a Single Data stream (SISD): Conventional sequential machines (e.g single-threaded processors: Superscalar, VLIW ..).**

SIMD

- **Single Instruction stream over Multiple Data streams (SIMD): Vector computers, array of synchronized processing elements. (exploit data parallelism)**

AKA Data Parallel Systems

MISD

- **Multiple Instruction streams and a Single Data stream (MISD): Systolic arrays for pipelined execution.**

MIMD

- **Multiple Instruction streams over Multiple Data streams (MIMD): Parallel computers:**

  Parallel Processor Systems: Exploit Thread Level Parallelism (TLP)

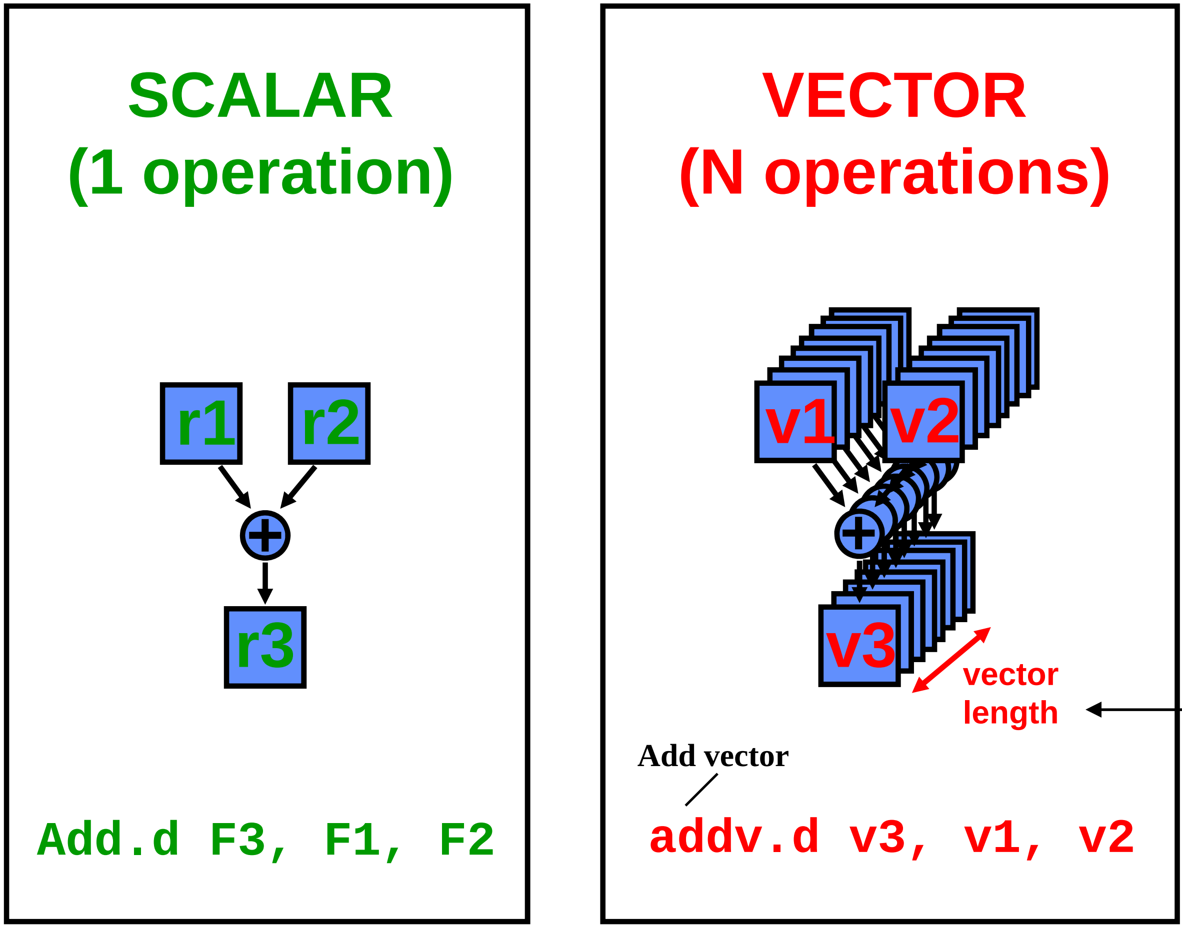
  - **Shared memory multiprocessors (e.g. SMP, CMP, NUMA, SMT)**
  - **Multicomputers: Unshared distributed memory, message-passing used instead (e.g Computer Clusters)**

From Multiple Processor Systems EECC756 Lecture 1

Vector Processing

- Vector processing exploits data parallelism by performing the same computation on linear arrays of numbers "vectors" using one instruction.
- The maximum number of elements in a vector supported by a vector ISA is referred to as the Maximum Vector Length (MVL).

Scalar ISA (RISC or CISC)



Vector ISA

Up to Maximum Vector Length (MVL)

Typical MVL = 64 (Cray)

Properties of Vector Processors/ISAs

- **Each result in a vector operation is independent of previous results (Data Parallelism, LLP exploited)**
  => Multiple pipelined Functional units (lanes) usually used, vector compiler ensures no dependencies between computations on elements of a single vector instruction
  => higher clock rate (less complexity)
- **Vector instructions access memory with known patterns**
  => Highly interleaved memory with multiple banks used to provide the high bandwidth needed and hide memory latency.
  => Amortize memory latency of over many vector elements
  => No (data) caches usually used. (Do use instruction cache)
- **A single vector instruction implies a large number of computations (replacing loops or reducing number of iterations needed)** **By a factor of MVL**
  => Fewer instructions fetched/executed.
  => Reduces branches and branch problems (control hazards) in pipelines.

As if loop-unrolling by default MVL times?

Changes to Scalar Processor to Run Vector Instructions

1

- A vector processor typically consists of an ordinary <u>pipelined scalar unit</u> plus
2 <u>a vector unit</u>.
- The scalar unit is basically not different than advanced pipelined CPUs, commercial vector machines have included both out-of-order scalar units (NEC SX/5) and VLIW scalar units (Fujitsu VPP5000).
- Computations that don't run in vector mode don't have high ILP, so can make scalar CPU simple (e.g in-order).
- The vector unit supports a vector ISA including decoding of vector instructions which includes:
  1 – Vector functional units.
  2 – ISA vector register bank, vector control registers (vector length, mask)
  3 – Vector memory Load-Store Units (LSUs).
  4 – Multi-banked main memory (to support the high data bandwidth needed, data cache not usually used)
- Send scalar registers to vector unit (for vector-scalar ops).
- Synchronization for results back from vector register, including exceptions.

Basic Types of Vector Architecture
(ISAs)

- Types of architecture for vector ISAs/processors:
  - ***Memory-memory vector ISAs/processors*:**

    All vector operations are memory to memory
  - ***Vector-register ISAs/processors*:**

    All vector operations between vector registers (except load and store)
    - Vector equivalent of load-store architectures (ISAs)
    - Includes all vector machines since the late 1980 Cray, Convex, Fujitsu, Hitachi, NEC

Appendix F (4th) Appendix G (3rd)

Basic Structure of Vector Register Architecture (Vector MIPS)

4 **Multi-Banked memory for bandwidth and latency-hiding**

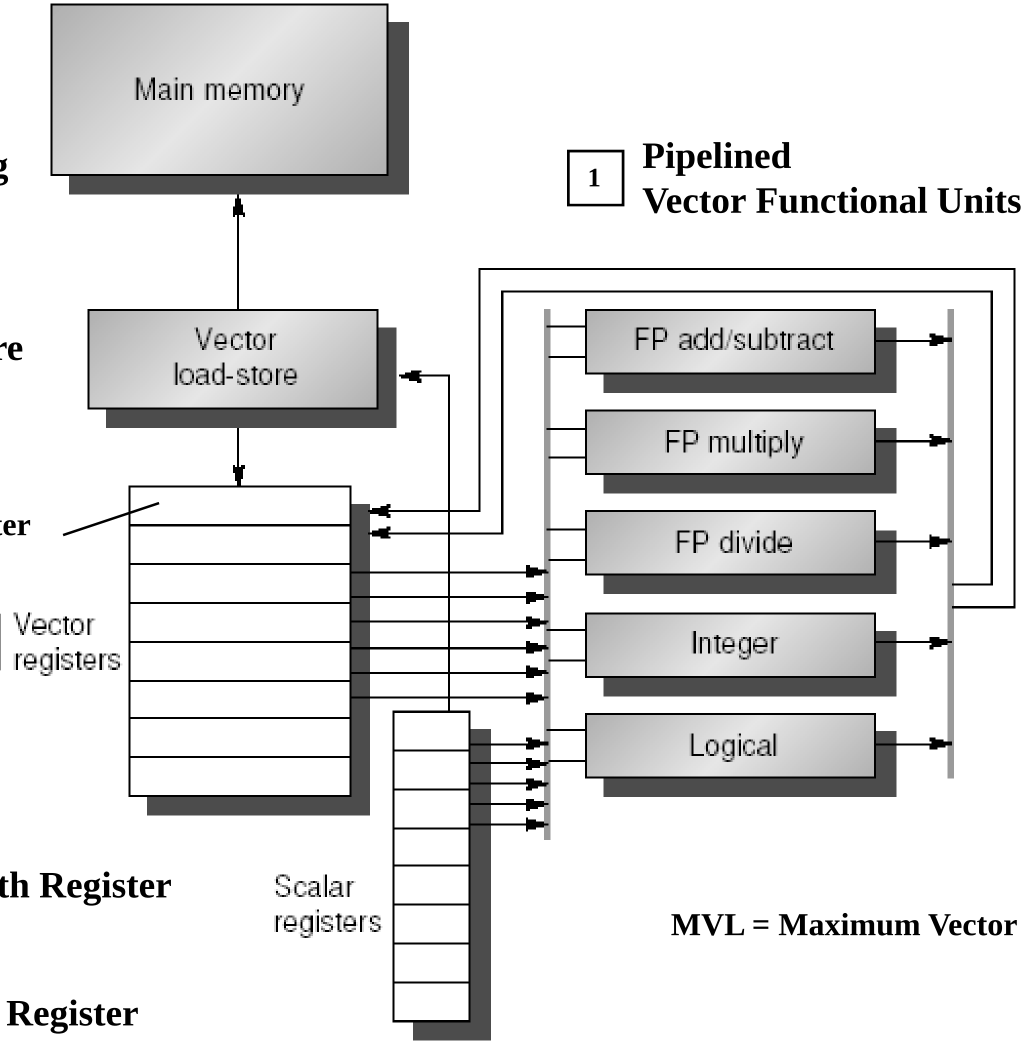
1 **Pipelined Vector Functional Units**

3 **Vector Load-Store Units (LSUs)**

Each Vector Register has MVL elements (each 64 bits)

8-Bytes Each

2



MVL = Maximum Vector Length

Vector Control Registers

VLR
Vector Length Register

VM
Vector Mask Register

Typical MVL = 64 (Cray)
MVL range 64-4096 (4K)

Example Vector-Register Architectures

First Vector Supercomputer Cray 1 (1976) 133 MFLOPS/s (Peak)

Processor (year)	Clock rate (MHz)	Vector registers	Elements per register (64-bit elements)	Vector arithmetic units	Vector load-store units	Lanes
Cray-1 (1976)	80	8	64	6: FP add, FP multiply, FP reciprocal, integer add, logical, shift	1	1
Cray X-MP (1983) Cray Y-MP (1988)	118 166	8	64	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store	1
Cray-2 (1985)	244	8	64	5: FP add, FP multiply, FP reciprocal/sqrt, integer add/shift/population count, logical	1	1
Fujitsu VP100/VP200 (1982)	133	8–256	32–1024	3: FP or integer add/logical, multiply, divide	2	1 (VP100) 2 (VP200)
Hitachi S810/S820 (1983)	71	32	256	4: FP multiply-add, FP multiply/divide-add unit, 2 integer add/logical	3 loads 1 store	1 (S810) 2 (S820)
Convex C-1 (1985)	10	8	128	2: FP or integer multiply/divide, add/logical	1	1 (64 bit) 2 (32 bit)
NEC SX/2 (1985)	167	8 + 32	256	4: FP multiply/divide, FP add, integer add/logical, shift	1	4
Cray C90 (1991) Cray T90 (1995)	240 460	8	128	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store	2
NEC SX/5 (1998)	312	8 + 64	512	4: FP or integer add/shift, multiply, divide, logical	1	16
Fujitsu VPP5000 (1999)	300	8–256	128–4096	3: FP or integer multiply, add/logical, divide	1 load 1 store	16
Cray SV1 (1998) SV1ex (2001)	300 500	8	64	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	1 load-store 1 load	2 8 (MSP)
VMIPS (2001)	500	8	64	5: FP multiply, FP divide, FP add, integer add/shift, logical	1 load-store	1

VMIPS = Vector MIPS

Appendix F (4th) Appendix G (3rd)

The VMIPS Vector FP Instructions

8 Vector Registers
V0-V7
MVL = 64
(Similar to Cray)

VMIPS = Vector MIPS

	Instruction	Operands	Function	
Vector FP	ADDV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.	
	ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.	
	SUBV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.	
	SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.	
	SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.	
	MULV.D	V1,V2,V3	Multiply elements of V2 and V3, then put each result in V1.	
	MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.	
	DIVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.	
	DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.	
	DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.	
Vector Memory	LV	V1,R1	Load vector register V1 from memory starting at address R1.	1- Unit Stride Access
	SV	R1,V1	Store vector register V1 into memory starting at address R1.	
	LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2, i.e., R1+i × R2.	2- Constant Stride Access
	SVWS	(R1,R2),V1	Store V1 from address at R1 with stride in R2, i.e., R1+i × R2.	
	LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1+V2(i), i.e., V2 is an index.	3- Variable Stride Access (indexed)
	SVI	(R1+V2),V1	Store V1 to vector whose elements are at R1+V2(i), i.e., V2 is an index.	
Vector Index	CVI	V1,R1	Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1.	
Vector Mask	S--V.D	V1,V2	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.	
	S--VS.D	V1,F0		
	POP	R1,VM	Count the 1s in the vector-mask register and store count in R1.	
	CVM		Set the vector-mask register to all 1s.	
Vector Length	MTC1	VLR,R1	Move contents of R1 to the vector-length register.	
	MFC1	R1,VLR	Move the contents of the vector-length register to R1.	
	MVTM	VM,F0	Move contents of F0 to the vector-mask register.	
	MVFM	F0,VM	Move contents of vector-mask register to F0.	

Appendix F (4th)
Appendix G (3rd)

Vector Control Registers: VM = Vector Mask
VLR = Vector Length Register

Scalar Vs. Vector Code Example

DAXPY (Y = a * X + Y)

Does it have good data Parallelism? Indication?

Assuming vectors X, Y are length 64 =MVL

Scalar vs. Vector

VLR = 64
VM = (1,1,1,1 ..1)

```
L.D      F0,a        ;load scalar a
LV       V1,Rx       ;load vector X
MULVS.D  V2,V1,F0    ;vector-scalar mult.
LV       V3,Ry       ;load vector Y
ADDV.D   V4,V2,V3    ;add
SV       Ry,V4       ;store the result
```

```
       L.D      F0,a
       DADDIU   R4,Rx,#512   ;last address to load
loop:  L.D      F2, 0(Rx)    ;load X(i)
       MUL.D    F2,F0,F2     ;a*X(i)
       L.D      F4, 0(Ry)    ;load Y(i)
       ADD.D    F4,F2, F4    ;a*X(i) + Y(i)
       S.D      F4 ,0(Ry)    ;store into Y(i)
       DADDIU   Rx,Rx,#8     ;increment index to X
       DADDIU   Ry,Ry,#8     ;increment index to Y
       DSUBU    R20,R4,Rx    ;compute bound
       BNEZ     R20,loop     ;check if done
```

As if the scalar loop code was unrolled MVL = 64 times: Every vector instruction replaces 64 scalar instructions.

Scalar Vs. Vector Code

578 (2+9*64) vs.
321 (1+5*64) ops (1.8X)

578 (2+9*64) vs.
6 instructions (96X)

64 operation vectors + no loop overhead

also 64X fewer pipeline hazards

Unroll? What does loop unrolling accomplish?

Vector Control Registers:
VM = Vector Mask
VLR = Vector Length Register

Vector/SIMD/Multimedia Scalar ISA Extensions

- **Vector or Multimedia ISA Extensions: Limited vector instructions added to scalar RISC/CISC ISAs with MVL = 2-8**

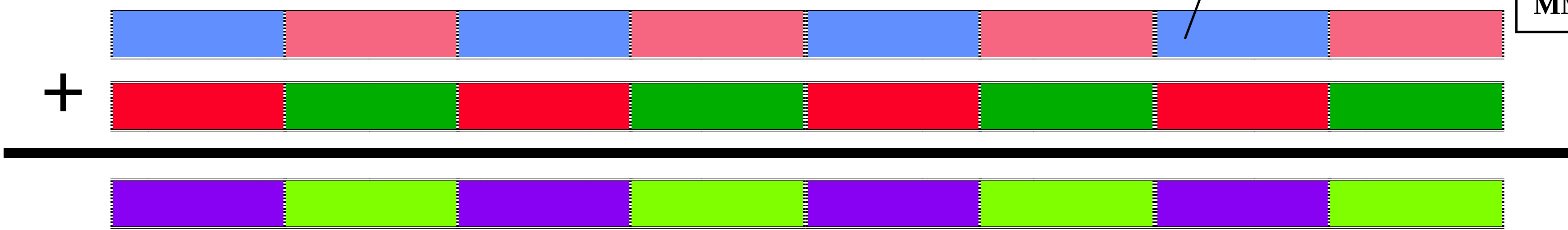
Why? **Improved exploitation of data parallelism in scalar ISAs/processors**

- **Example: Intel MMX: 57 new x86 instructions (1st since 386)**
  - **similar to Intel 860, Mot. 88110, HP PA-71000LC, UltraSPARC ...**
  - **3 integer vector element types: 8 8-bit (MVL =8), 4 16-bit (MVL =4) , 2 32-bit (MVL =2) in packed in 64 bit registers**
    - **reuse 8 FP registers (FP and MMX cannot mix)**

short vector: load, add, store 8, 8-bit operands

MVL = 8
for byte elements

MMX



  - **Claim: overall speedup 1.5 to 2X for multimedia applications (2D/3D graphics, audio, video, speech …)**

- **Intel SSE (Streaming SIMD Extensions) adds support for FP with MVL =2 to MMX**
- **Intel SSE2 Adds support of FP with MVL = 4 (4 single FP in 128 bit registers), 2 double FP MVL = 2, to SSE**

Major Issue: **Efficiently meeting the increased data memory bandwidth requirements of such instructions**