ECE/CS 757: Advanced Computer Architecture II
SIMD

Instructor:Mikko H Lipasti

Spring 2017
University of Wisconsin-Madison

Lecture notes based on slides created by John Shen, Mark Hill, David Wood, Guri Sohi, Jim Smith, Natalie Enright Jerger, Michel Dubois, Murali Annavaram, Per Stenström and probably others

SIMD & MPP Readings

Read: [20] C. Hughes, “Single-Instruction Multiple-Data Execution,” Synthesis Lectures on Computer Architecture, http://www.morganclaypool.com/doi/abs/10.2200/S00647ED1V01Y201505CAC032

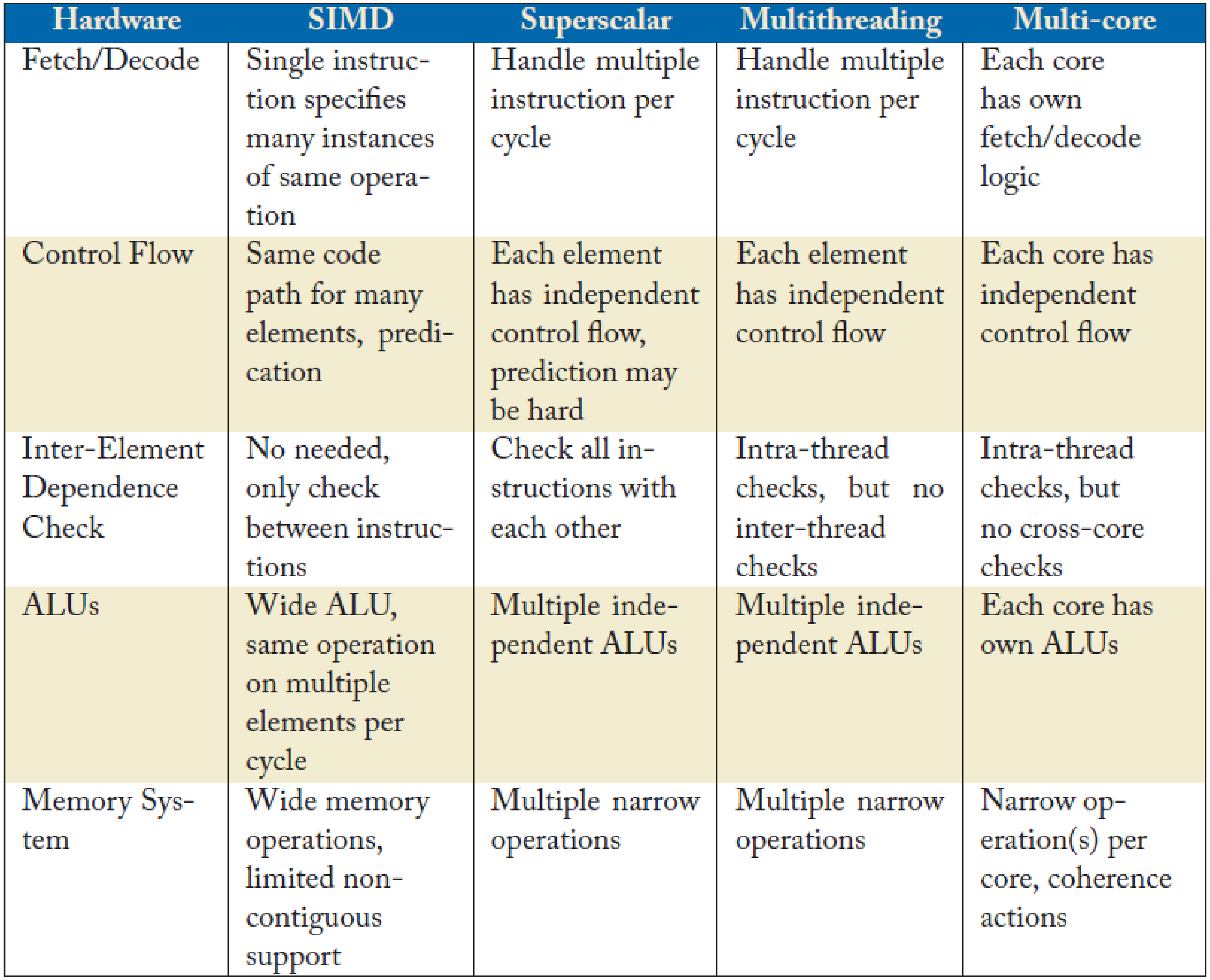
Review: [21] Steven L. Scott, Synchronization and Communication in the T3E Multiprocessor, Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems, pages 26-36, October 1996.

Lecture Outline

- SIMD introduction
- Automatic Parallelization for SIMD machines
- Vector Architectures
  - Cray-1 case study

SIMD vs. Alternatives

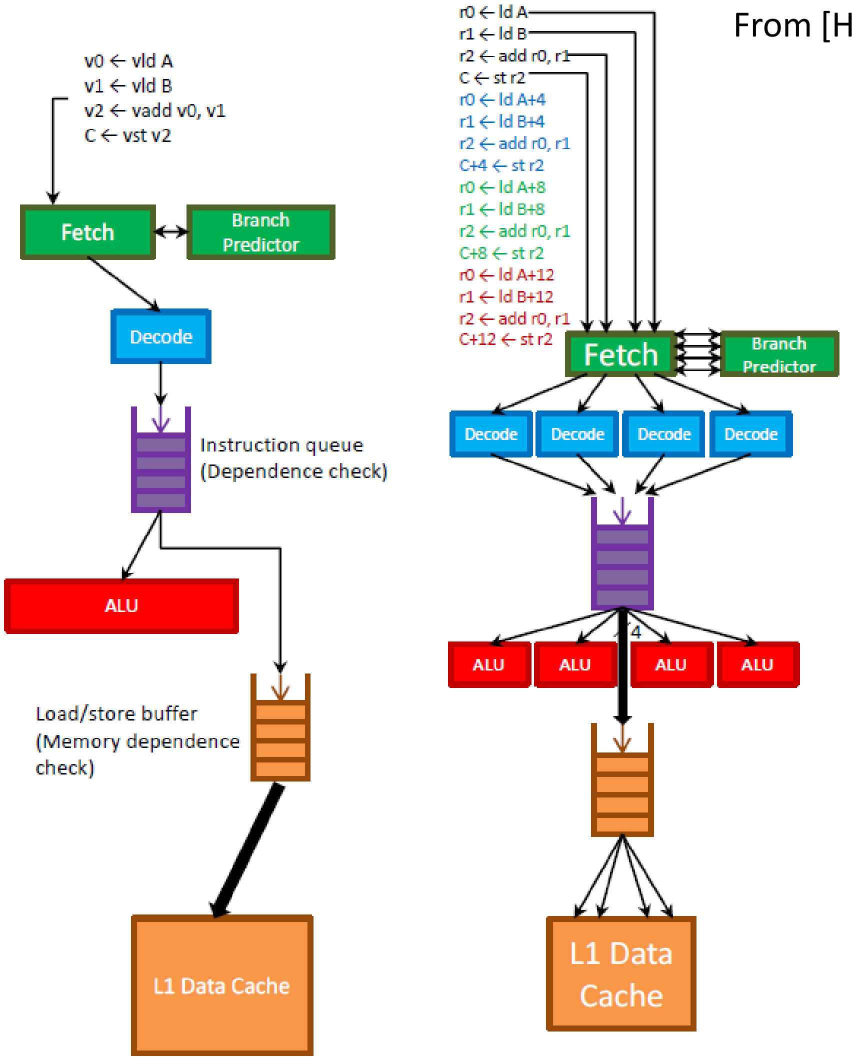
From [Hughes, SIMD Synthesis Lecture]



Hardware	SIMD	Superscalar	Multithreading	Multi-core
Fetch/Decode	Single instruction specifies many instances of same operation	Handle multiple instruction per cycle	Handle multiple instruction per cycle	Each core has own fetch/decode logic
Control Flow	Same code path for many elements, predication	Each element has independent control flow, prediction may be hard	Each element has independent control flow	Each core has independent control flow
Inter-Element Dependence Check	No needed, only check between instructions	Check all instructions with each other	Intra-thread checks, but no inter-thread checks	Intra-thread checks, but no cross-core checks
ALUs	Wide ALU, same operation on multiple elements per cycle	Multiple independent ALUs	Multiple independent ALUs	Each core has own ALUs
Memory System	Wide memory operations, limited non-contiguous support	Multiple narrow operations	Multiple narrow operations	Narrow operation(s) per core, coherence actions

SIMD vs. Superscalar

From [Hughes, SIMD Synthesis Lecture]

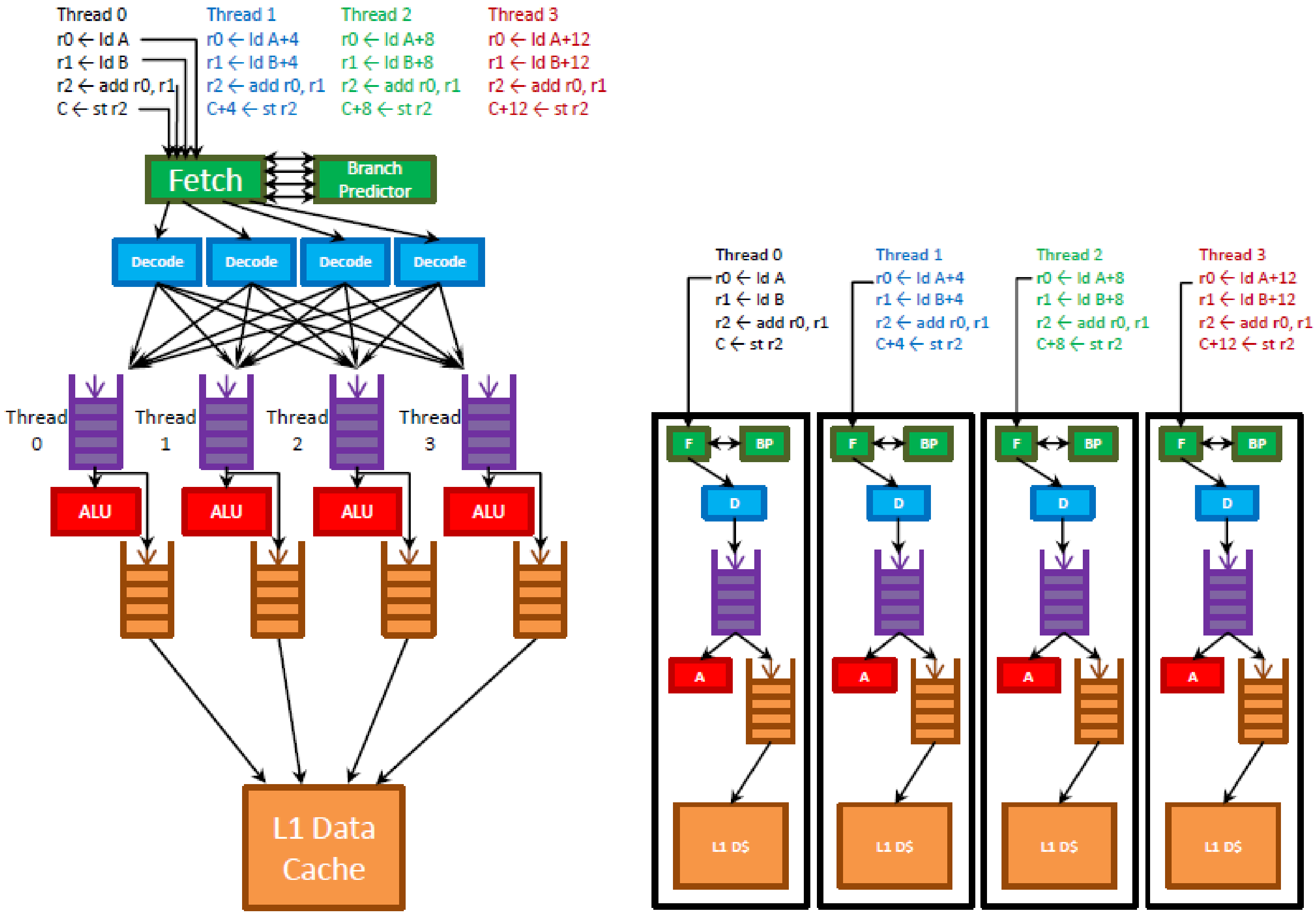


(a) SIMD

(b) Superscalar

Multithreaded vs. Multicore

From [Hughes, SIMD Synthesis Lecture]



```
Thread 0
r0 ← ld A
r1 ← ld B
r2 ← add r0, r1
C ← st r2

Thread 1
r0 ← ld A+4
r1 ← ld B+4
r2 ← add r0, r1
C+4 ← st r2

Thread 2
r0 ← ld A+8
r1 ← ld B+8
r2 ← add r0, r1
C+8 ← st r2

Thread 3
r0 ← ld A+12
r1 ← ld B+12
r2 ← add r0, r1
C+12 ← st r2
```

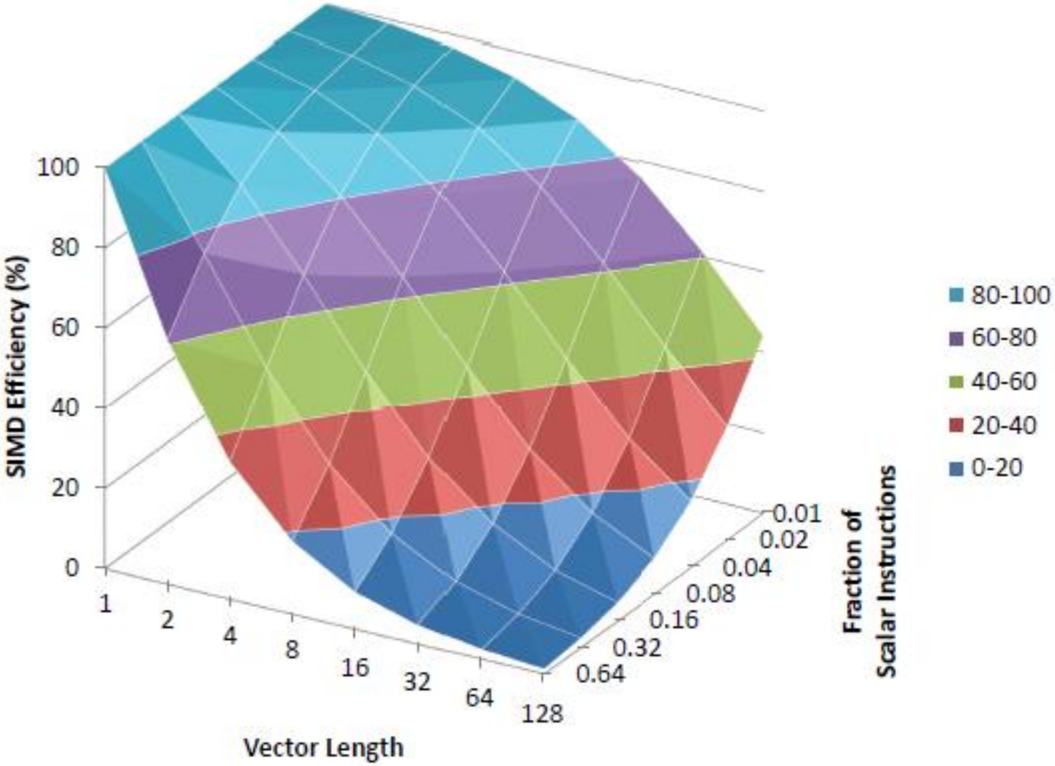
(a) Multithreaded

```
Thread 0
r0 ← ld A
r1 ← ld B
r2 ← add r0, r1
C ← st r2

Thread 1
r0 ← ld A+4
r1 ← ld B+4
r2 ← add r0, r1
C+4 ← st r2

Thread 2
r0 ← ld A+8
r1 ← ld B+8
r2 ← add r0, r1
C+8 ← st r2

Thread 3
r0 ← ld A+12
r1 ← ld B+12
r2 ← add r0, r1
C+12 ← st r2
```

(b) Multi-core

SIMD Efficiency

From [Hughes, SIMD Synthesis Lecture]



$$SIMD\ efficiency = \frac{\frac{instructions_{scalar}}{instructions_{SIMD}}}{vector\ \ length}$$

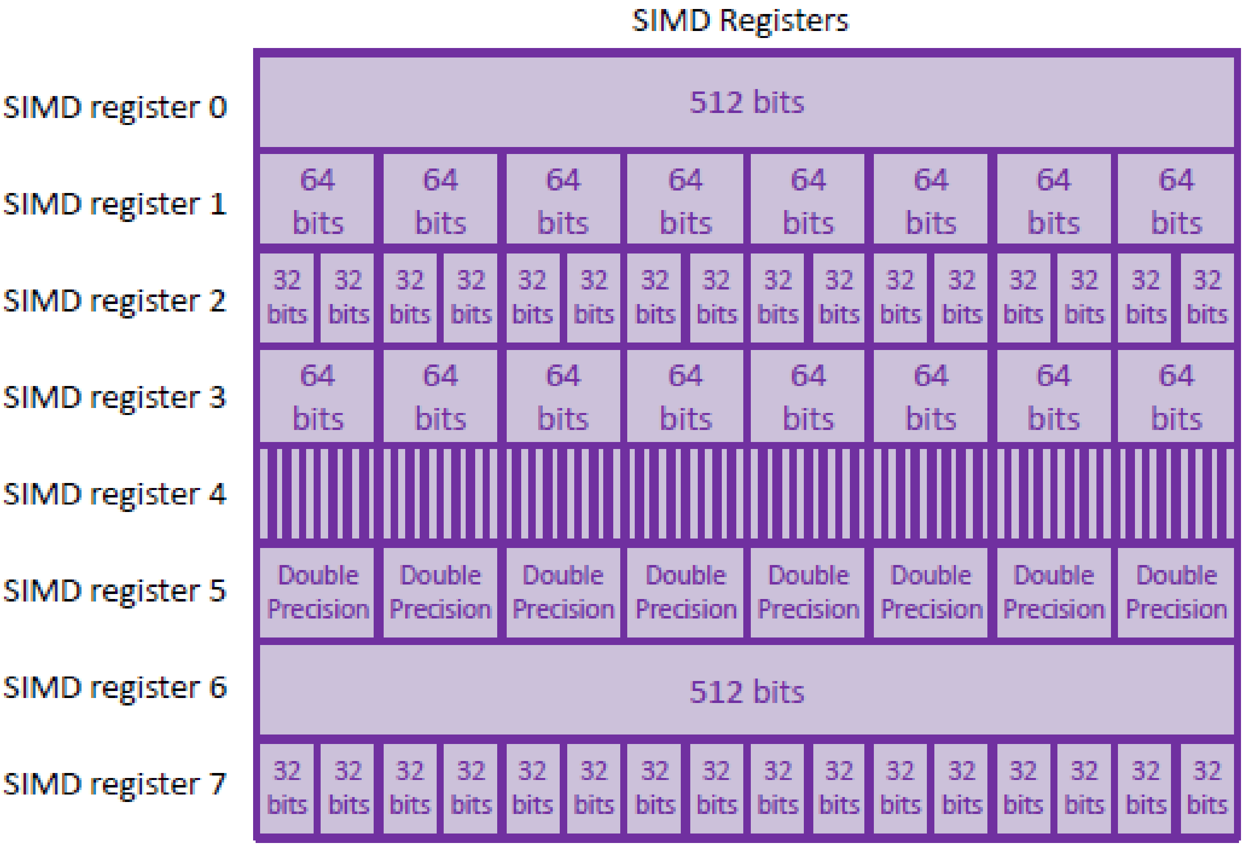
- Amdahl’s Law…

SIMD History

- Vector machines, supercomputing
  - Illiac IV, CDC Star-100, TI ASC,
  - Cray-1: *properly* architected (by Cray-2 gen)
- Incremental adoption in microprocessors
  - Intel Pentium MMX: vectors of bytes
  - Subsequently: SSEx/AVX-y, now AVX-512
  - Also SPARC, PowerPC, ARM, …
  - *Improperly* architected…
  - Also GPUs from AMD/ATI and Nvidia (later)

Register Overlays

From [Hughes, SIMD Synthesis Lecture]



SIMD Registers

SIMD register 0: 512 bits

SIMD register 1: 64 bits | 64 bits | 64 bits | 64 bits | 64 bits | 64 bits | 64 bits | 64 bits

SIMD register 2: 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits

SIMD register 3: 64 bits | 64 bits | 64 bits | 64 bits | 64 bits | 64 bits | 64 bits | 64 bits

SIMD register 4

SIMD register 5: Double Precision | Double Precision | Double Precision | Double Precision | Double Precision | Double Precision | Double Precision | Double Precision

SIMD register 6: 512 bits

SIMD register 7: 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits | 32 bits

SIMD Challenges

- Remainders
  - Fixed vector length, software has to fix up
  - Properly architected: VL is supported in HW
- Control flow deviation
  - Conditional behavior in loop body
  - Properly architected: vector masks
- Memory access
  - Alignment restrictions
  - Virtual memory, page faults (completion masks)
  - Irregular accesses: properly architected gather/scatter
- Dependence analysis (next)

Lecture Outline

- SIMD introduction
- Automatic Parallelization for SIMD machines
- Vector Architectures
  - Cray-1 case study

Automatic Parallelization

- Start with sequential programming model
- Let the compiler attempt to find parallelism
  - It can be done…
  - We will look at one of the success stories
- Commonly used for SIMD computing – *vectorization*
  - Useful for MIMD systems, also -- *concurrentization*
- Often done with FORTRAN
  - But, some success can be achieved with C

    (Compiler address disambiguation is more difficult with C)

Automatic Parallelization

- Consider operations on arrays of data

```
do I=1,N
    • A(I,J) = B(I,J) + C(I,J)
end do
```

  - Operations along one dimension involve *vectors*
- Loop level parallelism
  - *Do all* – all loop iterations are independent
    - Completely parallel
  - *Do across* – some dependence across loop iterations
    - Partly parallel

```
A(I,J) = A(I-1,J) + C(I,J) * B(I,J)
```

Data Dependence

- Independence $\Rightarrow$ Parallelism

  OR, dependence inhibits parallelism

```
S1:   A=B+C
S2:   D=A+2
S3:   A=E+F
```

- True Dependence (RAW):

  S1 δ S2
- Antidependence (WAR):

  S2 δ^{-} S3
- Output Dependence (WAW):

  S1 δ^{o} S3

Data Dependence Applied to Loops

- Similar relationships for loops
  - But consider iterations

```
     do   I=1,2
S1:           A(I)=B(I)+C(I)
S2:           D(I)=A(I)
     end do
```

- S1 $\delta_=$ S2
  - Dependence involving A, but on same loop iteration

Data Dependence Applied to Loops

- $S1\ \delta_{<}\ S2$

```
   do  I=1,2
S1:  A(I)=B(I)+C(I)
S2:  D(I)=A(I-1)
   end do
```

  - Dependence involving A, but read occurs on next loop iteration
  - *Loop carried dependence*

- $S2\ \delta^{-}_{<}\ S1$
  - Antidependence involving A, write occurs on next loop iteration

```
   do  I=1,2
S1:  A(I)=B(I)+C(I)
S2:  D(I)=A(I+1)
   end do
```

Loop Carried Dependence

❑ **Definition**

```
    • do    I = 1, N
S1:         X(f(i)) = F(...)
S2:         A = X(g(i)) ...
    end do
```

S1 δ S2 : is loop-carried

- **if there exist i_1, i_2 where**

$1 \leq i_1 < i_2 \leq N$ and $f(i_1) = g(i_2)$

❑ **If f and g can be arbitrary functions, the problem is essentially unsolvable.**

❑ **However, if (for example)**

```
f(i) = c*I + j and g(i) = d*I + k
```

there are methods for detecting dependence.

Loop Carried Dependences

- GCD test

```
     do    I = 1, N
S1:          X(c*I + j ) = F(...)
S2:          A = X(d*I + k) ...
     end do
```

f(x) = g(y) if `c*I + j = d*I + k`

This has a solution iff gcd(c, d) | k- j

- Example

```
A(2*I) =
      = A(2*I +1)
```

GCD(2,2) does not divide 1 - 0

- The GCD test is of limited use because it is very conservative

often gcd(c,d) = 1

```
  X(4i+1) = F(X(5i+2))
```

- Other, more complex tests have been developed

  e.g. Banerjee's Inequality, polyhedral analysis

Vector Code Generation

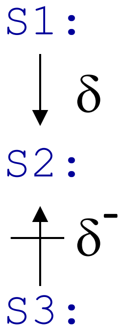
- In a vector architecture, a vector instruction performs identical operations on vectors of data
- Generally, the vector operations are *independent*
  - **A common exception is reductions (*horizontal* ops)**
- In general, to vectorize:
  - **There should be no cycles in the dependence graph**
  - **Dependence flows should be downward**

    ⇒ some rearranging of code may be needed.

Vector Code Generation: Example

```
   do I = 1, N
S1:     A(I) = B(I)
S2:     C(I) = A(I) + B(I)
S3:     E(I) = C(I+1)
   end do
```

- Construct dependence graph


S1:

δ

S2:

δ^-

S3:

Vectorizes (after re-ordering S2: and S3: due to antidependence)

```
S1:     A(I:N) = B(I:N)
S3:     E(I:N) = C(2:N+1)
S2:     C(I:N) = A(I:N) + B(I:N)
```

Multiple Processors (Concurrentization)

- Often used on outer loops
- Example

```
   do  I = 1, N
       do  J = 2, N
S1:        A(I,J) = B(I,J) + C(I,J)
S2:        C(I,J) = D(I,J)/2
S3:        E(I,J) = A(I,J-1)**2 + E(I,J-1)
       end do
   end do
```

- Data Dependences & Directions

  S1 $\delta_{=,<}$ S3

  S1 $\delta_{=,=}$ S2

  S3 $\delta_{=,<}$ S3

- Observations
  - All dependence directions for I loop are =

    ⇒ Iterations of the I loop can be scheduled in parallel

Scheduling

- Data Parallel Programming Model
  - SPMD (single program, multiple data)
- Compiler can pre-schedule:
  - Processor 1 executes 1st N/P iterations,
  - Processor 2 executes next N/P iterations
  - Processor P executes last N/P iterations
  - Pre-scheduling is effective if execution time is nearly identical for each iteration
- Self-scheduling is often used:
  - If each iteration is large
  - Time varies from iteration to iteration
    - iterations are placed in a "work queue”
    - a processor that is idle, or becomes idle takes the next block of work from the queue (critical section)

Code Generation with Dependences

```
    do  I = 2, N
S1:  A(I) = B(I) + C(I)
S2:  C(I) = D(I) * 2
S3:  E(I) = C(I) + A(I-1)
   end do
```

- Data Dependences & Directions

  S1 $\delta^{-}_{=}$ S2

  S1 $\delta_{<}$ S3

  S2 $\delta_{=}$ S3

- Parallel Code on N-1 Processors

```
S1:  A(I) = B(I) + C(I)
      signal(I)
S2:  C(I) = D(I) * 2
      if (I > 2) wait(I-1)
S3:  E(I) = C(I) + A(I-1)
```

- Observation
  - Weak data-dependence tests may add unnecessary synchronization.
    ⇒Good dependence testing crucial for high performance

Reducing Synchronization

```
     do  I = 1, N
S1:      A(I) = B(I) + C(I)
S2:      D(I) = A(I) * 2
S3:      SUM  = SUM + A(I)
     end do
```

- Parallel Code: Version 1

```
     do  I = p, N, P
S1:      A(I) = B(I) + C(I)
S2:      D(I) = A(I) * 2
          if (I > 1) wait(I-1)
S3:      SUM  = SUM + A(I)
          signal(I)
     end do
```

Reducing Synchronization, contd.

- Parallel Code: Version 2

```
    SUMX(p)  = 0
    do  I = p, N, P
S1:       A(I) = B(I) + C(I)
S2:       D(I) = A(I) * 2
S3:       SUMX(p) = SUMX(p) + A(I)
    end do
    barrier synchronize
    add partial sums
```

- Not always safe (bit-equivalent): why?

Vectorization vs Concurrentization

- When a system is a vector MP, when should vector/concurrent code be generated?

```
      do   J = 1,N
             do   I = 1,N
S1:                 A(I,J+1) = B(I,J) + C(I,J)
S2:                 D(I,J) = A(I,J) * 2
             end do
      end do
```

- Parallel & Vector Code: Version 1

```
      doacross J = 1,N
S1:          A(1:N,J+1) = B(1:N,J)+C(1:N,J)
             signal(J)
             if (J > 1) wait (J-1)
S2:          D(1:N,J)  = A(1:N,J) * 2
      end do
```

Vectorization vs Concurrentization

- Parallel & Vector Code: Version 2

  Vectorize on J, but non-unit stride memory access (assuming Fortran Column Major storage order)

```
      doall I = 1,N
S1:    A(I,2:N+1) = B(I,1:N) + C(I,1:N)
S2:    D(I,1:N) = A(I,1:N) * 2
      end do
```

- Need support for gather/scatter

Summary

- Vectorizing compilers have been a success
- Dependence analysis is critical to any auto-parallelizing scheme
  - Software (static) disambiguation
  - C pointers are especially difficult
- Can also be used for improving performance of sequential programs
  - Loop interchange
  - Fusion
  - Etc.

Aside: Thread-Level Speculation

- Add hardware to resolve difficult concurrentization problems
- Memory dependences
  - Speculate independence
  - Track references (cache versions, r/w bits, similar to TM)
  - Roll back on violations
- Thread/task generation
  - Dynamic task generation/spawn (Multiscalar)
- References
  - Gurindar S. Sohi , Scott E. Breach , T. N. Vijaykumar, Multiscalar processors, Proceedings of the 22nd annual international symposium on Computer architecture, p.414-425, June 22-24, 1995
  - J. Steffan , T Mowry, The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization, Proceedings of the 4th International Symposium on High-Performance Computer Architecture, p.2, January 31-February 04, 1998

Cray-1 Architecture

- Circa 1976
- 80 MHz clock
  - When high performance mainframes were 20 MHz
- Scalar instruction set
  - 16/32 bit instruction sizes
    - Otherwise conventional RISC
  - 8 S register (64-bits)
  - 8 A registers (24-bits)
- In-order pipeline
  - Issue in order
  - Can complete out of order (no precise traps)

Cray-1 Vector ISA

- 8 vector registers
  - 64 elements
  - 64 bits per element (word length)
  - Vector length (VL) register
- RISC format
  - Vi ← Vj OP Vk
  - Vi ← mem(Aj, disp)
- Conditionals via vector mask (VM) register
  - VM ← Vi pred Vj
  - Vi ← V2 conditional on VM

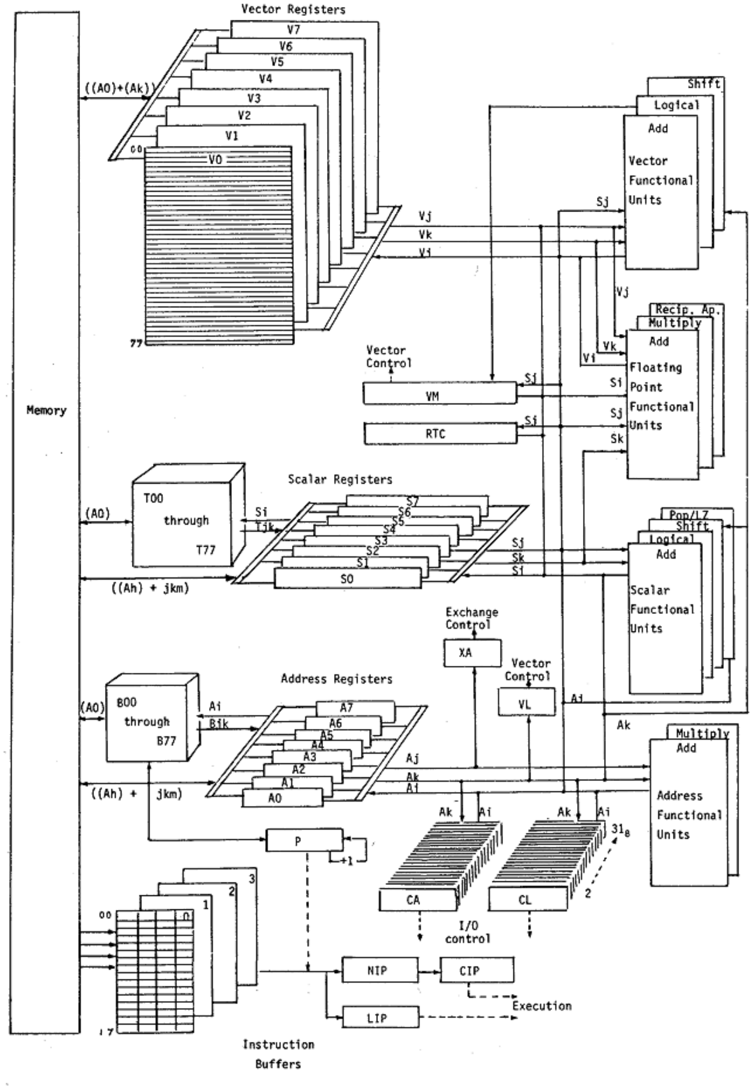


Figure 3-1. Computation section

Vector Example

```
      Do 10 i=1,looplength
         a(i) = b(i) * x + c(i)
   10  continue
```

```
            A1    ←     looplength          .initial values:
            A2    ←     address(a)          .for the arrays
            A3    ←     address(b)          .
            A4    ←     address(c)          .
            A5    ←     0                   .index value
            A6    ←     64                  .max hardware VL
            S1    ←     x                   .scalar x in register S1
            VL    ←     A1                  .set VL – performs mod function
              .
            BrC         done, A1<=0         .branch if nothing to do

more:       V3    ←     A4,A5               .load c indexed by A5 – addr mode not in Cray-1
            V1    ←     A3,A5               .load b indexed by A5
            V2    ←     V1 * S1             .vector times scalar
            V4    ←     V2 + V3             .add in c
            A2,A5 ← V4                      .store to a indexed by A5
            A7    ←     VL                  .read actual VL
            A1    ←     A1 – A7             .remaining iteration count
            A5    ←     A5 + A7             .increment index value
            VL    ←     A6                  . set VL for next iteration
            BrC         more, A1>0          .branch if more work
done:
```

Compare with Scalar

```
   Do 10 i=1,looplength
     a(i) = b(i) * x + c(i)
10  continue
```

2 loads
1 store
2 FP
1 branch
1 index increment (at least)
1 loop count increment

total -- 8 instructions per iteration

4-wide superscalar => up to 1 FP op per cycle
vector, with chaining => up to 2 FP ops per cycle (assuming mem b/w)

Also, in a CMOS microprocessor would save a lot of energy

Vector Conditional Loop

```
     do 80 i = 1,looplen
          if (a(i).eq.b(i)) then
               c(i) = a(i) + e(i)
          endif
80   continue
```

```
V1      ←      A1              .load a(i)
V2      ←      A2              .load b(i)
VM      ←      V1 == V2        .compare a and b; result to VM
V3      ←      A3; VM          .load e(i) under mask
V4      ←      V1 + V3; VM     .add under mask
A4      ←      V4; VM          .store to c(i) under mask
```

Vector Conditional Loop

Gather/Scatter Method (used in later Cray machines)

```
   do 80 i = 1,looplen
          if (a(i).eq.b(i)) then
              c(i) = a(i) + e(i)
          endif
80   continue
```

```
V1  ←       A1                      .load a(i)
V2  ←       A2                      .load b(i)
VM ←        V1 == V2                .compare a and b; result to VM
V5  ←       IOTA(VM)                .form index set
VL  ←       pop(VM)                 .find new VL (population count)
V6  ←       A1, V5                  .gather a(i) values
V3  ←       A3, V5                  .gather e(i) values
V4  ←       V6 + V3                 .add a and e
A4,V11 ←    V4                      .scatter sum into c(i)
```

Lecture Summary

- SIMD introduction
- Automatic Parallelization
- Vector Architectures
  - Cray-1 case study