

Architectural Features in VLIW Processors

- VLIW processors rely on the compiler to identify a **packet** of instructions that can be issued in the same cycle
 - Compiler takes responsibility for scheduling instructions so that their dependences are satisfied

<code>r1 = L r4</code>	<code>r2 = Add r1,M</code>	<code>f1 = Mul f1,f2</code>	<code>r5 = Add r5,4</code>
------------------------	----------------------------	-----------------------------	----------------------------

- Optimizations such as **loop unrolling**, and **software pipelining** expose more ILP, allowing the compiler to build issue packets
- Architectural support helps compiler expose/exploit more ILP

Basic Compiler Techniques (S1): Loop Unrolling

(Recap)

- Consider the example from last week:

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

```
L1: L.D      F0, 0(R1)  
    ADD.D   F4, F0, F2  
    S.D     F4, 0(R1)  
    DADDUI  R1, R1, #-8  
    BNE     R1, R2, L1
```

	Instruction	Issue Cycle
L1	L.D F0, 0(R1)	1
	stall	2
	ADD.D F4, F0, F2	3
	stall	4
	stall	5
	S.D F4, 0(R1)	6
	DADDUI R1, R1, #-8	7
	stall	8
	BNE R1, R2, L1	9
	stall	10

3 cycles

Basic Compiler Techniques: Loop Unrolling (cont'd)

- **Loop unrolling** optimization: Replicate loop body multiple times, adjusting the loop termination code

```

L1: L.D      F0, 0(R1)
     ADD.D   F4, F0, F2
     S.D     F4, 0(R1)
     L.D     F6, -8(R1)
     ADD.D   F8, F6, F2
     S.D     F8, -8(R1)
     L.D     F10, -16(R1)
     ADD.D   F12, F10, F2
     S.D     F12, -16(R1)
     L.D     F14, -24(R1)
     ADD.D   F16, F14, F2
     S.D     F16, -24(R1)
     DADDUI  R1, R1, #-32
     BNE    R1, R2, L1
  
```

	Instruction	Issue Cycle
L1	L.D F0, 0(R1)	1
	L.D F6, -8(R1)	2
	L.D F10, -16(R1)	3
	L.D F14, -24(R1)	4
	ADD.D F4, F0, F2	5
	ADD.D F8, F6, F2	6
	ADD.D F12, F10, F2	7
	ADD.D F16, F14, F2	8
	S.D F4, 0(R1)	9
	S.D F4, -8(R1)	10
	DADDUI R1, R1, #-32	11
	S.D F12, 16(R1)	12
	BNE R1, R2, L1	13
	S.D F16, 8(R1)	14

Basic Compiler Techniques: Loop Unrolling(cont'd)

- Unroll loop 5 times

```

L1: L.D      F0, 0(R1)
    ADD.D   F4, F0, F2
    S.D     F4, 0(R1)
    L.D     F6, -8(R1)
    ADD.D   F8, F6, F2
    S.D     F8, -8(R1)
    L.D     F10, -16(R1)
    ADD.D   F12, F10, F2
    S.D     F12, -16(R1)
    L.D     F14, -24(R1)
    ADD.D   F16, F14, F2
    S.D     F16, -24(R1)
    L.D     F18, -32(R1)
    ADD.D   F20, F18, F2
    S.D     F20, -32(R1)
    DADDUI  R1, R1, #-40
    BNE     R1, R2, L1
  
```

Provide instructions for VLIW

	Integer Instruction	FP Instruction	
L1	L.D F0, 0(R1)		1
	L.D F6, -8(R1)		2
	L.D F10, -16(R1)	ADD.D F4, F0, F2	3
	L.D F14, -24(R1)	ADD.D F8, F6, F2	4
	L.D F18, -32(R1)	ADD.D F12, F10, F2	5
	S.D F4, 0(R1)	ADD.D F16, F14, F2	6
	S.D F4, -8(R1)	ADD.D F20, F18, F2	7
	S.D F12, -16(R1)		8
	DADDUI R1, R1, #-40		9
	S.D F16, 16(R1)		10
	BNE R1, R2, L1		11
	S.D F20, 8(R1)		12

Hardware Support for VLIW

- To expose more parallelism at compile time
 - **Conditional** or **predicated** instructions
 - Predication registers in IA64
 - Allow the compiler to group instructions across branches
- To allow compiler to speculate, while ensuring program correctness
 - Result of speculated instruction will not be used in final computation if mispredicted
 - Speculative movement of instructions (before branches, reordering of loads/stores) must not cause exceptions
 - HW allows exceptions from speculative instructions to be ignored
 - **Poison bits** and **Reorder Buffers**
 - HW tracks memory dependences between loads and stores
 - LDS (speculative load) and LDV (load verify) instructions
 - Check for intervening store
 - Variant: LDV instruction can point to fix-up code

HW Support for Speculative Operations (H1)

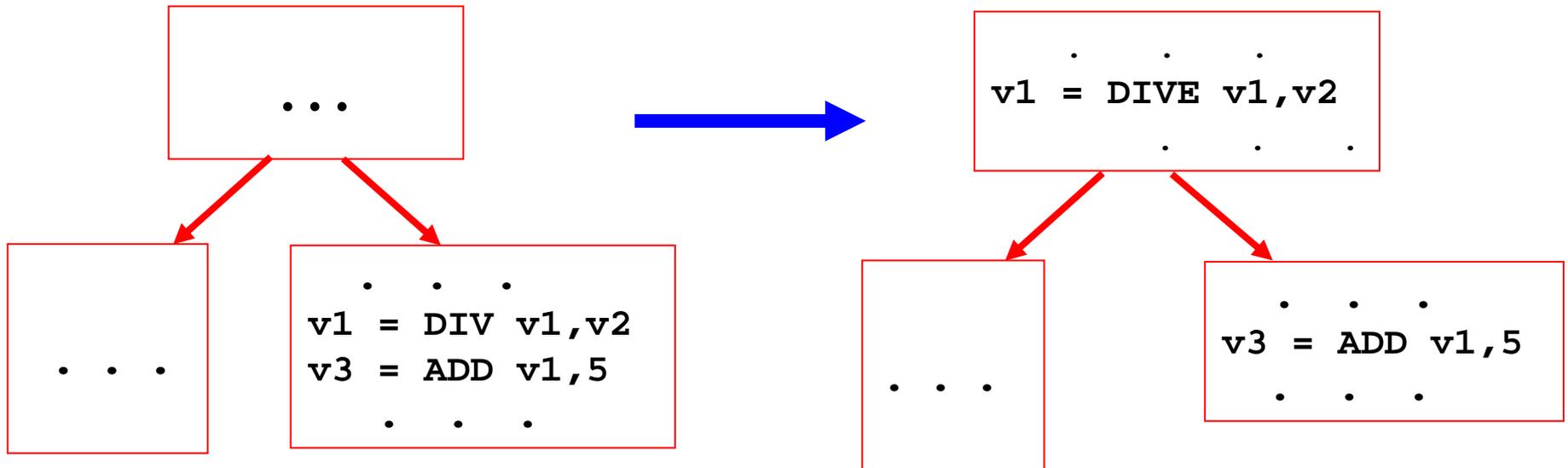
- Speculative operations in HPL-PD architecture from HP Labs written identically to their non-speculative counterparts, but with an “E” appended to the operation name.
 - E.g., **DIVE**, **ADDE**, **PBRRE**

Poison bits: If an exceptional condition occurs during a speculative operation, the exception is not raised

- A bit is set in the result register to indicate that such a condition occurred
- Speculative bits are simply propagated by speculative instructions
- When a non-speculative operation encounters a register with the speculative bit set, an exception is raised

(H1) Compiler Use of Speculative Operations

- Here is an optimization that uses speculative instructions:



- Also the effect of the DIV latency is reduced
- If a divide-by-zero occurs, an exception will be raised by ADD

HW Support for Predication (H2)

- **Conditional** or **predicated** instructions
 - Instruction is “conditionally” executed, else no-op
 - Originally: a separate set of (simple) instructions
 - Now: more general support
- In HPL-PD, most operations can be predicated
 - they can have an extra operand that is a one-bit predicate register.
r2 = ADD r1,r3 if p2
 - If the predicate register contains 0, the operation is not performed
 - The values of predicate registers are typically set by “compare-to-predicate” operations
p1 = CMPP<= r4,r5

Compiler Uses of Predication

- if-conversion
- To aid code motion by instruction scheduler
 - e.g. hyperblocks

Uses of Predication: If-conversion

- If-conversion replaces conditional branches with predicated operations
- For example, the code generated for:

```
if (a < b)
  c = a;
else
  c = b;
if (d < e)
  f = d;
else
  f = e;
```

might be the two VLIW instructions:

P1 = CMPP.< a,b	P2 = CMPP.>= a,b	P3 = CMPP.< d,e	P4 = CMPP.>= d,e
c = a if p1	c = b if p2	f = d if p3	f = e if p4

Compare-to-predicate instructions

- In previous slide, there were two pairs of almost identical instructions
 - just computing complement of each other
- HPL-PD provides two-output CMPP instructions


p1,p2 = CMPP.W.<.UN.UC r1,r2

(H2) If-conversion, revisited

- Using two-output CMPP instructions, the code generated for:

```
if (a < b)
  c = a;
else
  c = b;
if (d < e)
  f = d;
else
  f = e;
```

might instead be:

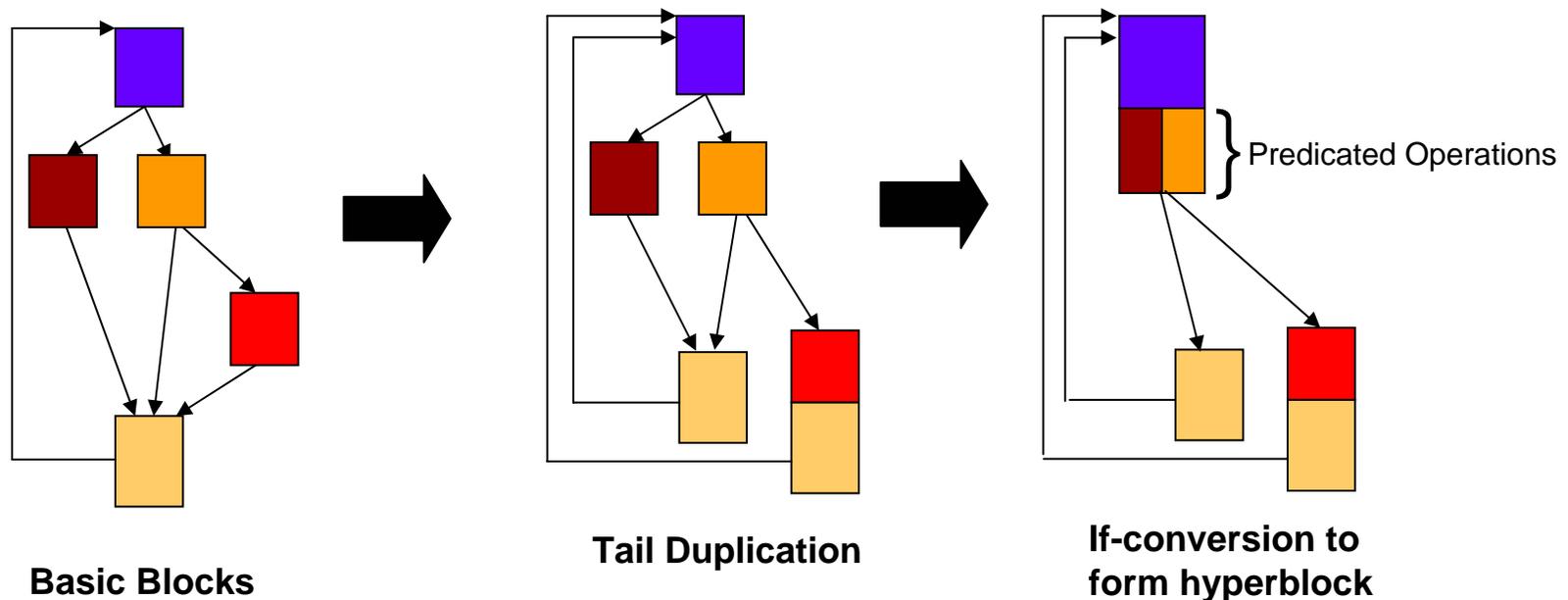
Only two CMPP operations,
occupying less of the VLIW
instruction.

`p1,p2 = CMPP.W.<.UN.UC a,b` `p3,p4 = CMPP.W.<.UN.UC d,e`

`c = a if p1` `c = b if p2` `f = d if p3` `f = e if p4`

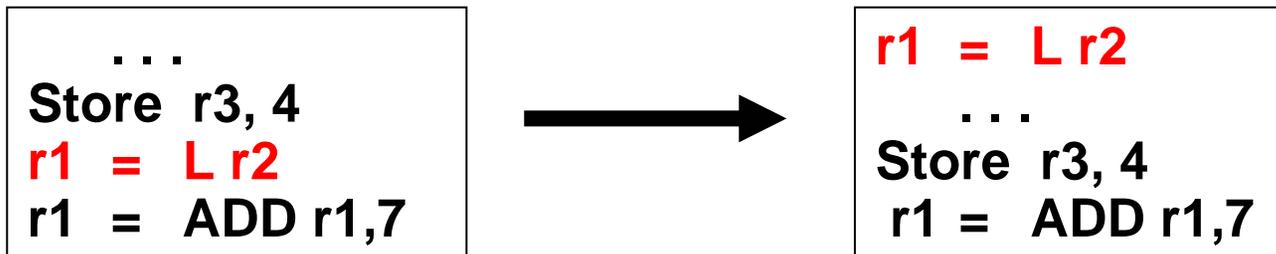
Uses of Predication: Hyperblock Formation

- In hyperblock formation, if-conversion is used to form larger blocks of operations than the usual basic blocks
 - tail duplication used to remove some incoming edges in middle of block
 - if-conversion applied after tail duplication
 - larger blocks greater opportunity for code motion to increase ILP



HW Support for Memory Disambiguation (H3)

- Here's a desirable optimization (due to long load latencies):



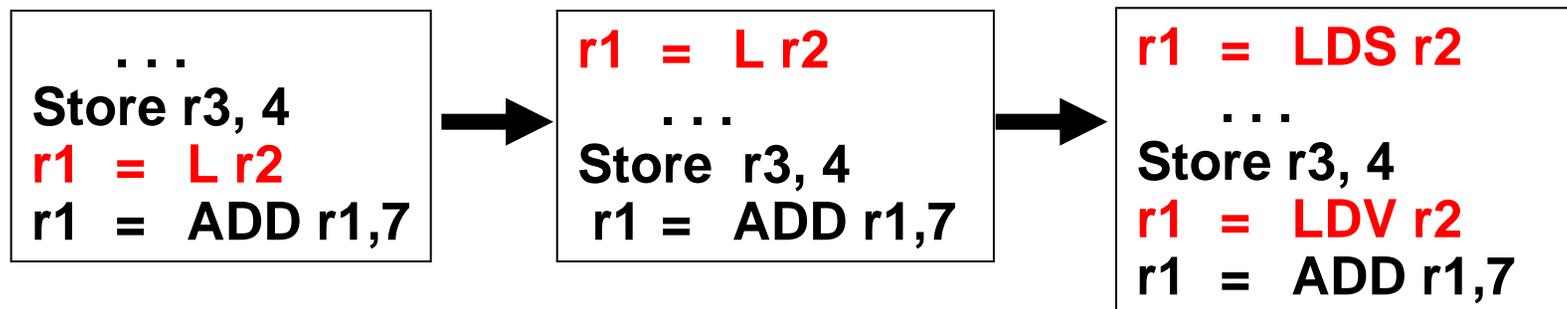
- However, this optimization is not valid if the load and store reference the same location
 - i.e., if **r2** and **r3** contain the same address
 - **this cannot be determined at compile time**
- HPL-PD solves this by providing run-time memory disambiguation

(H3) HW Support for Memory Disambiguation (cont'd)

HPL-PD provides two special instructions to replace a load instruction:

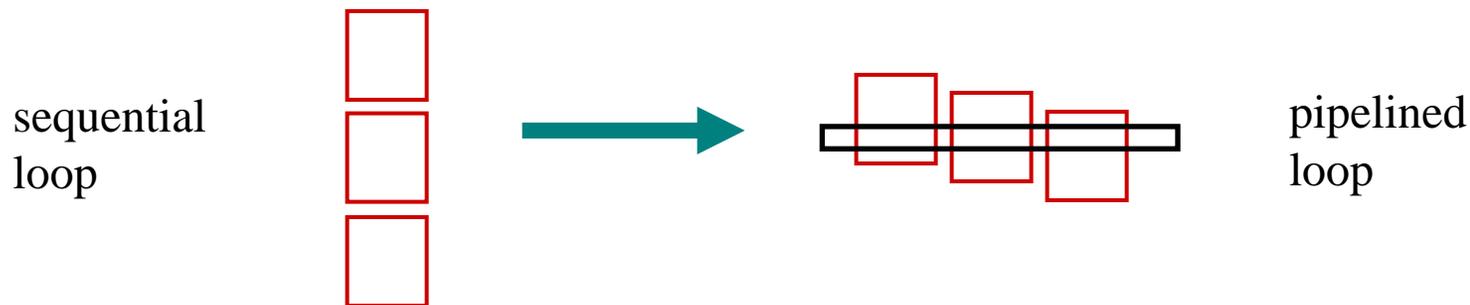
- **r1 = LDS r2** ; **speculative load**
 - Initiates a load like a normal load instruction
 - A log entry can be made in a table to store the memory location
- **r1 = LDV r2** ; **load verify**
 - Checks to see if store to memory location has occurred since the LDS
 - If so, the new load is issued and the pipeline stalls. Otherwise, it's a no-op

The previous optimization becomes



More Sophisticated Compiler Optimizations: Software Pipelining (S2)

- Software Pipelining is the technique of scheduling instructions across several iterations of a loop
 - reduces pipeline stalls on sequential pipelined machines
 - exploits instruction level parallelism on superscalar and VLIW machines
 - intuitively, iterations are overlaid so that an iteration starts before the previous iteration have completed



(S2) Software Pipelining Example

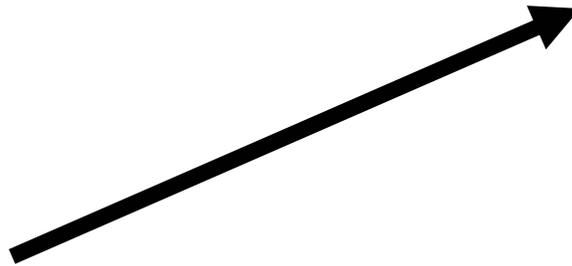
- Source code:

```
for(i=0;i<n;i++) sum += a[i]
```

- Loop body in assembly:

```
r1 = L r0  
--- ;stall  
r2 = Add r2,r1  
r0 = add r0,4
```

- Unroll loop and
allocate registers



```
r1 = L r0  
--- ;stall  
r2 = Add r2,r1  
r0 = Add r0,12  
r4 = L r3  
--- ;stall  
r2 = Add r2,r4  
r3 = add r3,12  
r7 = L r6  
--- ;stall  
r2 = Add r2,r7  
r6 = add r6,12  
r10 = L r9  
--- ;stall  
r2 = Add r2,r10  
r9 = add r9,12
```

(S2) Software Pipelining Example (cont'd)

- Schedule unrolled Instructions, exploiting VLIW (or not)

```

r1 = L r0
r4 = L r3
r2 = Add r2,r1 r7 = L r6
r0 = Add r0,12 r2 = Add r2,r4 r10 = L r9
r3 = add r3,12 r2 = Add r2,r7 r1 = L r0
r6 = add r6,12 r2 = Add r2,r10 r4 = L r3
r9 = add r9,12 r2 = Add r2,r1 r7 = L r6
r0 = Add r0,12 r2 = Add r2,r4 r10 = L r9
r3 = add r3,12 r2 = Add r2,r7 r1 = L r0
r6 = add r6,12 r2 = Add r2,r10 r4 = L r3
r9 = add r9,12 r2 = Add r2,r1 r7 = L r6
. . .
r0 = Add r0,12 r2 = Add r2,r4 r10 = L r9
r3 = add r3,12 r2 = Add r2,r7
r6 = add r6,12 Add r2,r10
r9 = add r9,12

```

Identify repeating pattern (kernel)

(S2) Software Pipelining Example (cont)

Loop becomes:

```
r1 = L r0  
r4 = L r3  
r2 = Add r2,r1 r7 = L r6
```

```
r0 = Add r0,12 r2 = Add r2,r4 r10 = L r9  
r3 = Add r3,12 r2 = Add r2,r7 r1 = L r0  
r6 = Add r6,12 r2 = Add r2,r10 r4 = L r3  
r9 = Add r9,12 r2 = Add r2,r1 r7 = L r6
```

```
r0 = Add r0,12 r2 = Add r2,r4 r10 = L r9  
r3 = Add r3,12 r2 = Add r2,r7  
r6 = Add r6,12 Add r2,r10  
r9 = Add r9,12
```

← prolog

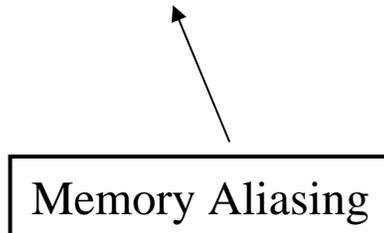
← kernel

← epilog

Constraints on Software Pipelining

The instruction-level parallelism in a software pipeline is limited by

- Resource Constraints
 - VLIW instruction width, functional units, bus conflicts, etc.
- Dependence Constraints
 - particularly loop carried dependences between iterations
 - arise when
 - the same register is used across several iterations
 - the same memory location is used across several iterations



(S2) Aliasing-based Loop Dependences

Source code:

```
for(i=3; i<n;i++)
    a[i] = a[i-3] + c;
```



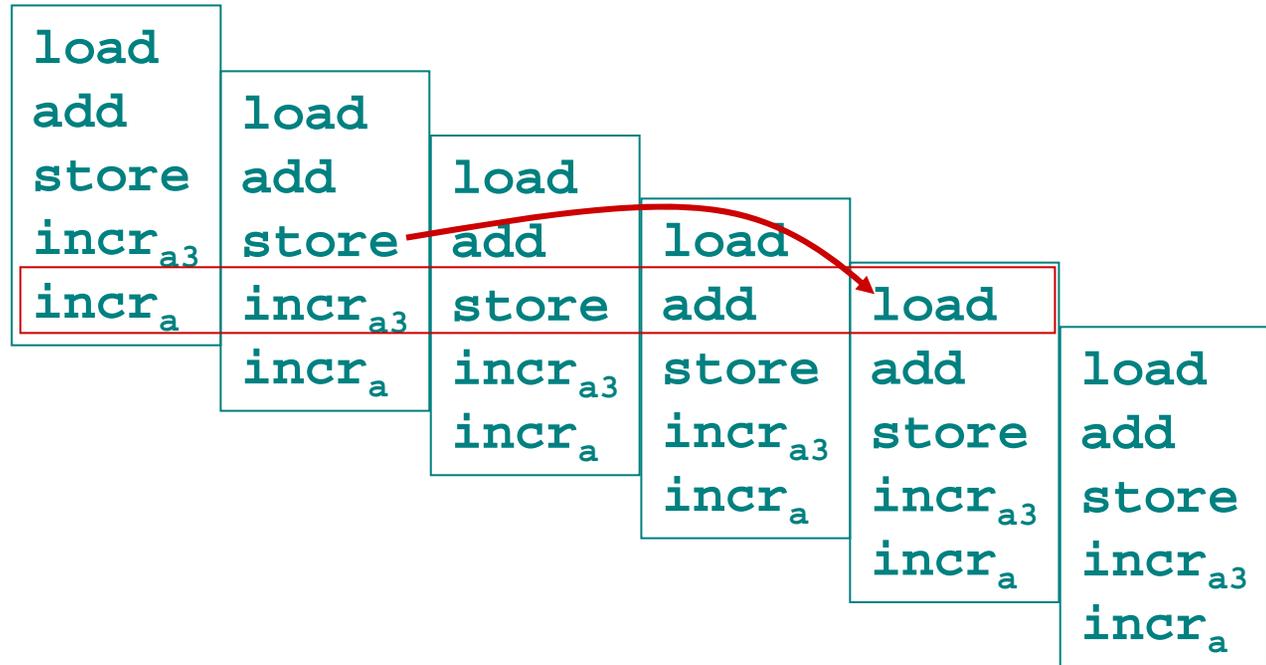
dependence spans three iterations
 “distance = 3”

Assembly:

```
load
add
store
incra3
incra
```

Pipeline:

kernel
 1 cycle



Dynamic Memory Aliasing

- What if the code were:

```
for(i=A;i<n;i++)  
    a[i] = a[i-k] + c;
```

where k is unknown at compile time?

- The dependence distance is the value of k (“dynamic” aliasing)
 - $k = 0$ (no dependence), $k > 0$ (true dependence with distance k),
 $k < 0$ (anti-dependence with distance $|k|$)
 - The worst case is $k = 1$
-
- What can the compiler do?
 - Assume the worst, and generate the most pessimistic pipelined schedule
 - Generate different versions of the software pipeline for different distances
 - branch to the appropriate version at run-time
 - possible code explosion, cost of branch

Summary: VLIW Processors

- Architectural features enable aggressive compiler optimizations
 - To pack multiple instructions per VLIW packet
 - Loop unrolling and software pipelining
- Hardware support
 - Speculative instructions
 - Conditional/Predicated instructions
 - Run-time memory disambiguation
 - Hardware support for preserving exception behavior
 - Poison bits, reorder buffer
- Limiting factors
 - Increased code size: requires aggressive unrolling; not full instructions
 - VLIW lock step => 1 hazard and all instructions stall
 - Binary code compatibility is practical weakness