

ECE 486/586

Computer Architecture

Prof. Mark G. Faust

Maseeh College of Engineering
and Computer Science



Instruction Level Parallelism and Its Static Exploitation

- Reading:
 - Hennessy & Patterson: Chapter 2.2, Appendix G.1 – G.3
- Homework:

Exploiting ILP

- Hardware Approaches
 - Dynamic Scheduling
 - Speculative Execution
- Software Approaches
 - Static Scheduling
 - Rely on Compiler Technology
 - Some techniques have been known to programmers for years as means of optimizing code
 - Relies on
 - Inherent ILP
 - Cleverness of compiler
 - Knowledge of latencies in microprocessor pipeline

Static Scheduling

- Latency information required for scheduling
 - Need to know when FU available
 - Need to know when results from FU available

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Figure 4.1 Latencies of FP operations used in this chapter. The first column shows the originating instruction type. The second column is the type of the consuming instruction. The last column is the number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is zero, since the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0.

Static Scheduling


- C code fragment for adding a scalar s to a vector x

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

- MIPS equivalent (as a smart compiler might generate) assuming at least one iteration and pre-computing R2 so that 8(R2) is last element

```
Loop:   L.D      F0,0(R1)      ; F0=array element  
        ADD.D    F4,F0,F2      ; add scalar in F2  
        S.D      F4,0(R1)      ; store result  
        DADDIU   R1,R1,#-8     ; 8 bytes/double  
        BNE      R1,R2,Loop    ; branch R1 != R2
```

Pointer Implementation of Code



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


```
Ptr = &X[1000];
```

```
EndPtr = &X[0];
```

```
do
```

```
    *Ptr-- = *Ptr + s
```

```
while (Ptr != EndPtr);
```



```
Loop: L.D      F0,0(R1)      ; F0=array element  
      ADD.D    F4,F0,F2      ; add scalar in F2  
      S.D      F4,0(R1)      ; store result  
      DADDIU   R1,R1,#-8     ; 8 bytes/double  
      BNE      R1,R2,Loop    ; branch R1 != R2
```

Static Scheduling

- Execution of loop without scheduling

		<u>clock cycle issued</u>
Loop:	L.D F0,0(R1)	1
	<*stall*>	2
	ADD.D F4,F0,F2	3
	<*stall*>	4
	<*stall*>	5
	S.D F4,0(R1)	6
	DADDIU R1,R1,#-8	7
	<*stall*>	8
	BNE R1,R2,Loop	9
	<*stall*>	10

Potential WAR {

Static Scheduling

Execution of loop with scheduling

		<u>clock cycle issued</u>
Loop:	L.D F0,0(R1)	1
	DADDIU R1,R1,#-8	2
	ADD.D F4,F0,F2	3
	<*stall*>	4
	BNE R1,R2,Loop	5
	S.D F4,8(R1)	6

Delayed branch slot

Altered 0(R1) to 8(R1) because subtraction move earlier
Compiler swapped order of **DADDUI** and **S.D** by changing
address to which **S.D** stored from **0(R1)** to **8(R1)**

Compare...

```
Loop:  L.D    F0,0(R1)
        <*stall*>
        ADD.D  F4,F0,F2
        <*stall*>
        <*stall*>
        S.D    F4,0(R1)
        DADDIU R1,R1,#-8
        <*stall*>
        BNE    R1,R2,Loop
        <*stall*>
```

```
        L.D    F0,0(R1)
        DADDIU R1,R1,#-8
        ADD.D  F4,F0,F2
        <*stall*>
        BNE    R1,R2,Loop
        S.D    F4,8(R1)
```

Static Scheduling

6 cycles

3 for actual loop body

2 for loop overhead (**DADDIU** and **BNE** instructions)

1 stall

		<u>clock cycle issued</u>
Loop:	L.D F0,0(R1)	1
	DADDIU R1,R1,#-8	2
	ADD.D F4,F0,F2	3
	<*stall*>	4
	BNE R1,R2,Loop	5
	S.D F4,8(R1)	6

How do we get more instructions in parallel if loop only 3-6 cycles long?

Loop Unrolling

- Well known software optimization technique
 - Originally intended to improve execution speed
 - Reduces loop overhead
 - Independent of architecture/pipeline



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

n iterations

```
for (i=1000; i>0; i=i-4) {  
    x[i]    = x[i]    + s;  
    x[i-1]  = x[i-1]  + s;  
    x[i-2]  = x[i-2]  + s;  
    x[i-3]  = x[i-3]  + s;  
}
```


k loop copies

n/k iterations

Loop Unrolling

- What if n/k not an integer?
 - Perform n/k iterations of k copies
 - Perform $n \bmod k$ iterations of original loop body

```
for (i=1002; i>0; i--)  
    x[i] = x[i] + s;
```



```
for (i=1002; i>1000; i=i-1)  
    x[i] = x[i] + s;
```


$n \bmod k$ iterations
original loop body

```
for (i=1000; i>0; i=i-4) {  
    x[i]      = x[i]      + s;  
    x[i-1]    = x[i-1]    + s;  
    x[i-2]    = x[i-2]    + s;  
    x[i-3]    = x[i-3]    + s;  
}
```

n/k iterations
 k loop copies

Loop Unrolling

- Why not completely unroll loops?
 - Code size!
 - Register pressure (need temporary registers!)



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

```
for (i=1000; i>0; i=i-4) {  
    x[i]    = x[i]    + s;  
    x[i-1]  = x[i-1]  + s;  
    x[i-2]  = x[i-2]  + s;  
    x[i-3]  = x[i-3]  + s;  
}
```

Loop Unrolling

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

Needed to change displacement values since 32 (4 x 8 bytes) subtracted
Needed additional registers

Loop Unrolling

Loop:	→	L.D	F0,0(R1)	
1	→	ADD.D	F4,F0,F2	
2	→	S.D	F4,0(R1)	;drop DADDUI & BNE
1	→	L.D	F6,-8(R1)	
2	→	ADD.D	F8,F6,F2	
2	→	S.D	F8,-8(R1)	;drop DADDUI & BNE
1	→	L.D	F10,-16(R1)	
2	→	ADD.D	F12,F10,F2	
2	→	S.D	F12,-16(R1)	;drop DADDUI & BNE
1	→	L.D	F14,-24(R1)	
2	→	ADD.D	F16,F14,F2	
		S.D	F16,-24(R1)	
1	→	DADDUI	R1,R1,#-32	
		BNE	R1,R2,Loop	
1	→			

Without scheduling, every operation is followed by a dependent operation and will thus cause stalls – resulting in 28 (14 operations + 14 stalls) clock cycles – slower than 4 iterations of the original (but scheduled) loop

Scheduled Unrolled Loop

```
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)
        BNE     R1,R2,Loop
        S.D      F16,8(R1);8-32 = -24
```

Note: because **DADDUI** occurs before 2nd two **S.D** instructions we need to change the displacement values to correct for the prior subtraction of 32.

Scheduled Unrolled Loop

```
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)
        BNE      R1,R2,Loop
        S.D      F16,8(R1);8-32 = -24
```

Scheduling results in no stalls! 14 clock cycles required (3.5 clock cycles per loop iteration). Overhead of 2 clock cycles now spread over 4 iterations → 0.5 clock cycles

Compiler Had To...

- Determine that it was legal to move the **S.D** after the **DADDIU** and **BNE** and adjust the **S.D** offset appropriately
- Determine that loop unrolling would be useful by determining that the loop iterations were independent
- Use different registers to hold results to avoid data hazards
- Eliminate test and branch instructions and adjust loop termination and iteration code
- Determine that the loads and stores in unrolled loop could be interchanged by observing loads/stores from different iterations of the loop are independent
- Schedule the code taking into consideration the pipeline latency information and preserving dependencies

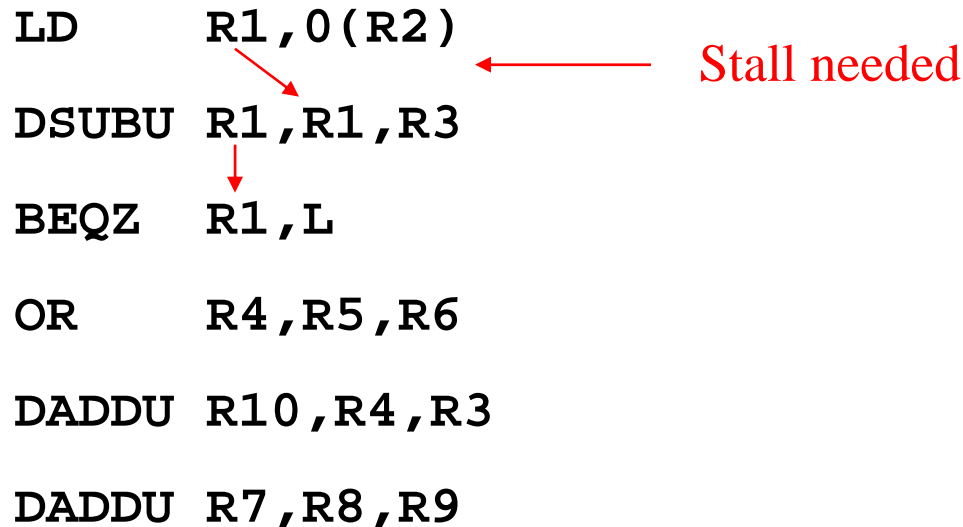
Static Multiple Issue

	Integer instruction		FP instruction	Clock cycle
Loop:	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	F8,-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,Loop		11
	S.D	F20,8(R1)		12

Loop unrolled 5 times to avoid stalls! 12 clock cycles required (2.5 clock cycles per loop iteration). Savings from: loop overhead reduction (fewer loop maintenance instructions), elimination of dependencies (no stalls), scheduling (multiple issue). Parallelism: multiple issue, pipeline without data dependence-induced stalls

Static Branch Prediction

LD R1, 0(R2) ← Stall needed
DSUBU R1, R1, R3
BEQZ R1, L
OR R4, R5, R6
DADDU R10, R4, R3
L: DADDU R7, R8, R9



- What if we knew:
 - Branch almost always taken
 - R7 is not used in the fall through path
- Could move the **DADDU R7, R8, R9** to position after **LD**, eliminating stall
- Observations
 - Correctness preserved if prediction wrong (**R7** not used in fall through path)
 - Improved performance if prediction correct (stall eliminated, **DADDU** needed)
 - Performance no worse if prediction wrong (**DADDU** not needed, but stall eliminated)

Static Branch Prediction

LD R1, 0(R2) ← Stall needed
 ↙
DSUBU R1, R1, R3
 ↓
BEQZ R1, L

OR R4, R5, R6

DADDU R10, R4, R3

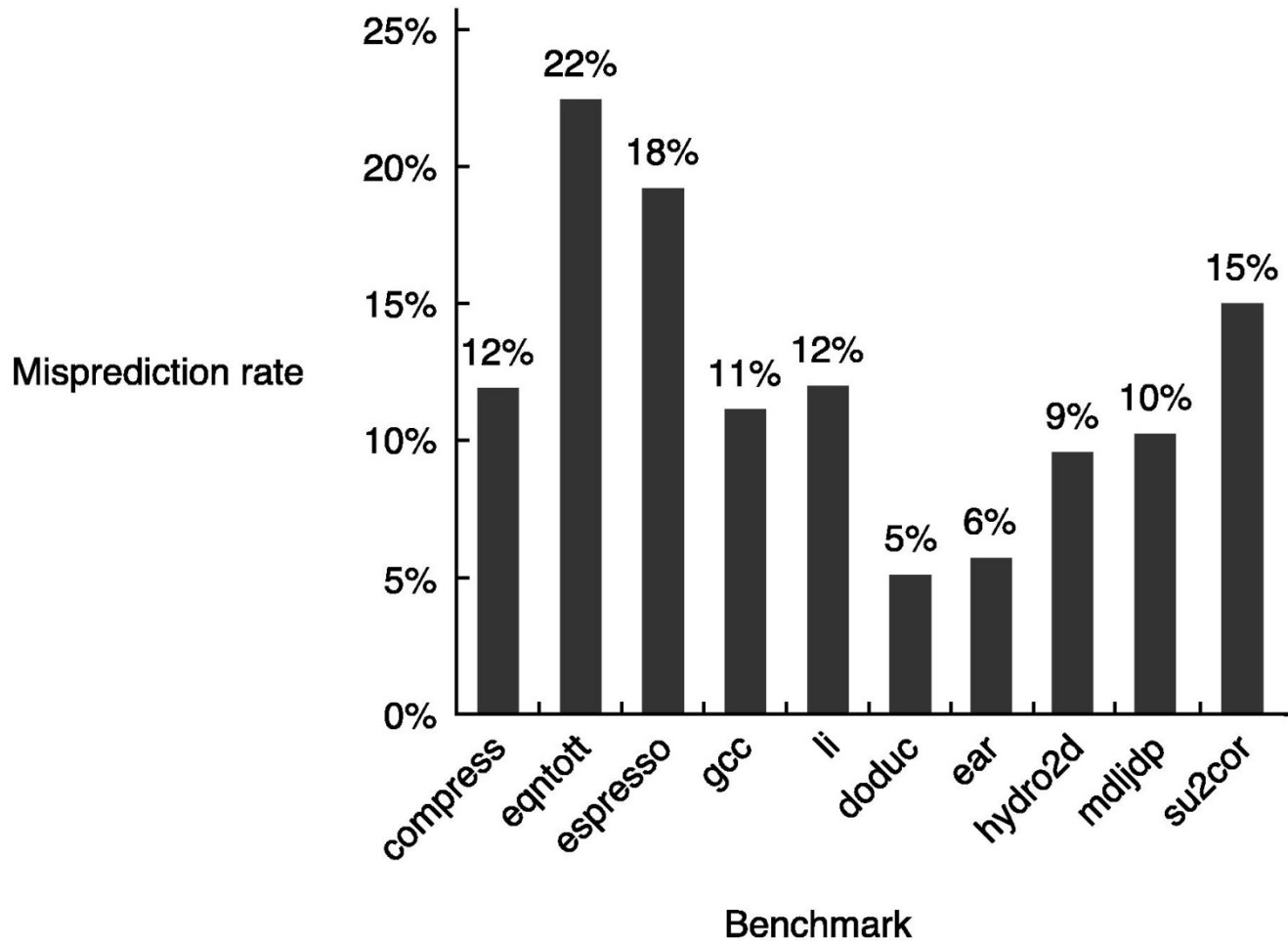
L: DADDU R7, R8, R9

- What if we knew:
 - Branch almost never taken
 - R4 is not used in the taken path
- Could move the **OR R4, R5, R6** to position after **LD**, eliminating stall
- Observations
 - Correctness preserved if prediction wrong (**R4** not used in taken path)
 - Improved performance if prediction correct (stall eliminated, **OR** needed)
 - Performance no worse if prediction wrong (**OR** not needed, but stall eliminated)

Static Branch Prediction Strategies

- Predict Taken
 - Misprediction rate equal to untaken branch frequency
 - SPEC programs: 34% (accurate 64% of time!)
 - But wide variation (59% to 9%) untaken frequency
- Use Branch Direction
 - Predict forward-going branches as not taken
 - Predict backward-going branches as taken
 - Again, sensitive to benchmark
- Use Profiling from Previous Runs
 - Bimodal distribution (individual branches highly biased)

Misprediction Rate of Profile-Based Predictor on SPEC92

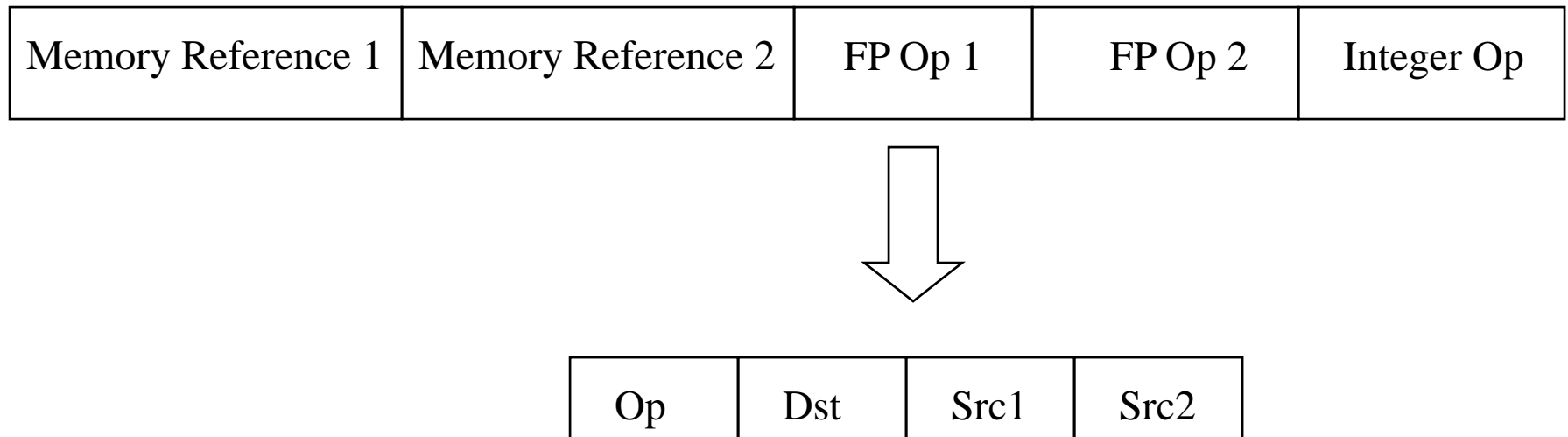


Static Multiple Issue: VLIW Approach

- Superscalar processors determine number of instructions to issue on-the-fly
 - Scheduling involves
 - Checking dependences between instructions in the issue packet
 - Checking for dependences between each instruction in packet and instructions already in pipeline
 - Dynamically Scheduled
 - H/W intensive, little or no S/W support
 - Statically Scheduled
 - Good support from smart compiler
- Alternative
 - Use compiler technology to schedule, reducing stalls
 - Use compiler technology to ensure no dependences in issue packet
 - Eliminate checks in H/W
 - Earliest use of this technique in VLIW (Very Long Instruction Words)

VLIW

- Explicit parallelism
- Instruction word contains parallel instructions for multiple independent functional units
- Very long instruction words (80-180 bits common)
 - Parallel instructions
 - Wider operand fields (more registers needed because of parallel ops)



VLIW: An Example

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch	
L.D F0,0(R1)	L.D F6,-8(R1)				
L.D F10,-16(R1)	L.D F14,-24(R1)				
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2		
		ADD.D F20,F18,F2	ADD.D F24,F22,F2		
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		DADDUI R1,R1,#-56	
S.D F12,-16(R1)	S.D F16,-24(R1)				
S.D F20,24(R1)	S.D F24,16(R1)				
S.D F28,8(R1)				BNE R1,R2,Loop	

Loop unrolled 7 times to avoid stalls! 9 clock cycles required (1.29 clock cycles per loop iteration). Nearly 2X faster than two-issue superscalar example shown earlier.

Issues: code size (more iterations unrolled, more registers (larger field sizes)), un-used functional units.

VLIW

- Issues
 - Code size
 - More “unrolling”
 - Long word sizes result not just from parallel operations
 - Longer field sizes for additional registers
 - No-Ops consume space if FU not utilized
 - Control
 - Lock step execution
 - Stall on FU (e.g. cache miss) stall all FUs
 - Binary Compatibility
 - Impact of adding FUs
 - Explicit → instruction format changes
 - Contrast with impact on superscalar → invisible to ISA

Advanced Compiler Support for Exposing and Exploiting ILP

- Loop-Level Parallelism
 - Loop-Carried Dependence

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

Dependence on two uses of $x[i]$ in loop but no loop-carried dependence – could execute entire loop in parallel if sufficient resources available

Dependence on i between successive iterations but its an induction variable and easily eliminated (as we saw with pointer implementation)

Advanced Compiler Support for Exposing and Exploiting ILP

- Loop-Carried Dependence

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

Both S1 and S2 contain loop carried dependences: A[i+1] computation depends on result A[i] computed in an earlier iteration of the loop (likewise for B[i+1] dependence on B[i]). Forces successive iterations to execute in series.

S2 uses value A[i+1] computed by S1 in same iteration – doesn't prevent parallel execution of successive loop iterations

Assumes A,B,C are distinct, non-overlapping arrays (which may not be known – arrays may be passed as parameters to procedure containing the loop)

Advanced Compiler Support for Exposing and Exploiting ILP

- Loop-Carried Dependence

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

S1 uses value assigned by S2 in previous iteration so there is a loop-carried dependence between S2 and S1.


But not circular:

neither statement depends upon a prior iteration of itself

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

Therefore the loop can be made parallel

Advanced Compiler Support for Exposing and Exploiting ILP



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

Can swap order of
S1 and S2

```
A[1] = A[1] + B[1];  
for (i=1; i<=99; i++) {  
    B[i+1] = C[i] + D[i];    /* S2 */  
    A[i+1] = A[i+1] + B[i+1]; /* S1 */  
}  
B[101] = C[100] + D[100];
```

On first iteration of
loop S1 depends on
pre-existing value
of B[1]

Transformation eliminates loop-carried dependence allowing parallel
execution of successive loop iterations

Loop-Carried Dependence

- Analysis needs to find all loop-carried dependences
 - Operate on source level where intent is easier to determine
 - Inexact – analysis can tell us that a dependence *may* exist

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

- Smart compiler will determine that second reference to A[i] doesn't require a load instruction but could instead use the register into which A[i] was computed in the previous source statement.
 - Both statements always to same location
 - No intervening access to same location

Recurrences

- Recurrences are common form of loop-carried dependence. (Fibonacci...)

```
for (i=2; i<=100; i++) {  
    Y[i] = Y[i-1] + Y[i];  
}
```

- Dependence distance indicates how “far apart” the dependency is in terms of loop iterations and therefore how far we can unroll the loop to achieve parallelism

```
for (i=6; i<=100; i++) {  
    Y[i] = Y[i-5] + Y[i];  
}
```

Dependence distance = 5

Finding Dependences

- Critical part of three tasks
 - Good scheduling of code
 - Determining which loops might contain parallelism
 - Eliminating name dependences
- Task is made more difficult because of arrays and pointers (C, C++), pass-by-reference semantics of FORTRAN
- Assumption that array indices are affine
 - One dimensional array index is affine if it can be written as $a \times i + b$ where i is the loop index
 - Non affine indices arise in sparse arrays: $X[Y[i]]$
- Determining whether there is a dependence between two references to the same array in a loop is equivalent to determining whether two affine functions can have the same value for different indices between the bounds of the loop

Finding Dependences

- Example: assume we've stored to an array element with index value $a \times i + b$ and loaded from the same array with index value $c \times i + d$ where i is the loop index variable. A dependence exists if two conditions hold:
 - There are two iteration indices, j and k , both within the limits of the loop. That is $m \leq j \leq n$, $m \leq k \leq n$.
 - The loop stores into an array element indexed by $a \times j + b$ and later fetches from that same array element when it is indexed by $c \times k + d$. That is, $a \times j + b = c \times k + d$.
- In general, cannot determine dependence at compile time (values of a, b, c, d may not be known – contained in other arrays)
- But most loops are simple and reasonable compile time tests exist

Finding Dependences: GCD Test

- A test for the absence of a dependence is the greatest common divisor (GCD) test
- If a loop-carried dependence exists, $\text{GCD}(c,a)$ must divide $(d-b)$.
- Example:

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

- Then $a = 2$, $b = 3$, $c = 2$, $d = 0$. $\text{GCD}(a,c) = 2$. $(d - b) = -3$. 2 does not divide -3 , therefore no loop dependence is possible.
- Sufficient to guarantee no dependence exists, but possible for GCD test to succeed and no dependence exists.
- General case is NP-complete

An Example

- Identify all true dependences, output dependences, and anti-dependences in the following loop. Eliminate output dependences and anti-dependences by renaming.

```
for (i=1; i<=100; i++) {  
    Y[i] = X[i] / c;    /* S1 */  
    X[i] = X[i] + c;    /* S2 */  
    Z[i] = Y[i] + c;    /* S3 */  
    Y[i] = c - Y[i];    /* S4 */  
}
```

An Example

```
for (i=1; i<=100; i++) {  
    Y[i] = X[i] / c;    /* S1 */  
    X[i] = X[i] + c;    /* S2 */  
    Z[i] = Y[i] + c;    /* S3 */  
    Y[i] = c - Y[i];    /* S4 */  
}
```

- True dependences from S1 to S3 and from S1 to S4 because of Y[i]. Not loop-carried so don't prevent loop from being considered parallel and unrolled.
- Anti-dependence from S1 to S2 based on X[i]
- Anti-dependence from S3 to S4 based on Y[i]
- Output dependence from S1 to S4 based on Y[i]

An Example

```
for (i=1; i<=100; i++) {  
    Y[i] = X[i] / c;    /* S1 */  
    X[i] = X[i] + c;    /* S2 */  
    Z[i] = Y[i] + c;    /* S3 */  
    Y[i] = c - Y[i];    /* S4 */  
}
```

Y renamed to T: removes output dependence from S1 to S4

Y renamed to T: removes anti-dependence from S3 to S4

X renamed to X1: removes anti-dependence from S1 to S2

```
for (i=1; i<=100; i++) {  
    T[i] = X[i] / c;    /* S1 */  
    X1[i] = X[i] + c;   /* S2 */  
    Z[i] = T[i] + c;    /* S3 */  
    Y[i] = c - T[i];    /* S4 */  
}
```

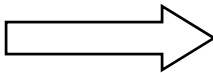
Because of renaming, subsequent references to X should be made to X1

Techniques to Reduce Dependent Operations

Sequences often arise as result of loop unrolling

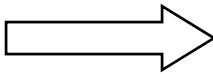
- Copy Propagation

- Simplify sequences (saw this with displacements)

DADDIU	R1,R2,#4		DADDIU	R1,R2,#8
DADDUI	R1,R1,#4			

- Tree Height Reduction

- Increase parallelism (possibly increasing number of operations)

ADD	R1,R2,R3		ADD	R1,R2,R3
ADD	R4,R1,R6		ADD	R4,R6,R7
ADD	R8,R4,R7		ADD	R8,R1,R4

Requires 3 cycles because of dependences

Requires 2 cycles

Other Techniques

Sequences often arise as result of loop unrolling

- Recurrences

- Assume statement appears in loop

```
sum = sum + x[i];
```

- After unrolling 5 times

```
sum = sum + x[1] + x[2] + x[3] + x[4] + x[5];
```

- Un-optimized, this leads to 5 dependent operations
- But can be rewritten to require only 3 dependent operations

```
sum = ((sum + x[1]) + (x[2] + x[3]) + (x[4] + x[5]));
```

Software Pipelining: Symbolic Loop Unrolling

- Reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop
- Best understood by reviewing scheduled code for superscalar MIPS (Figure 4.2)
- Separates dependent instructions by entire loop body, decreasing stalls

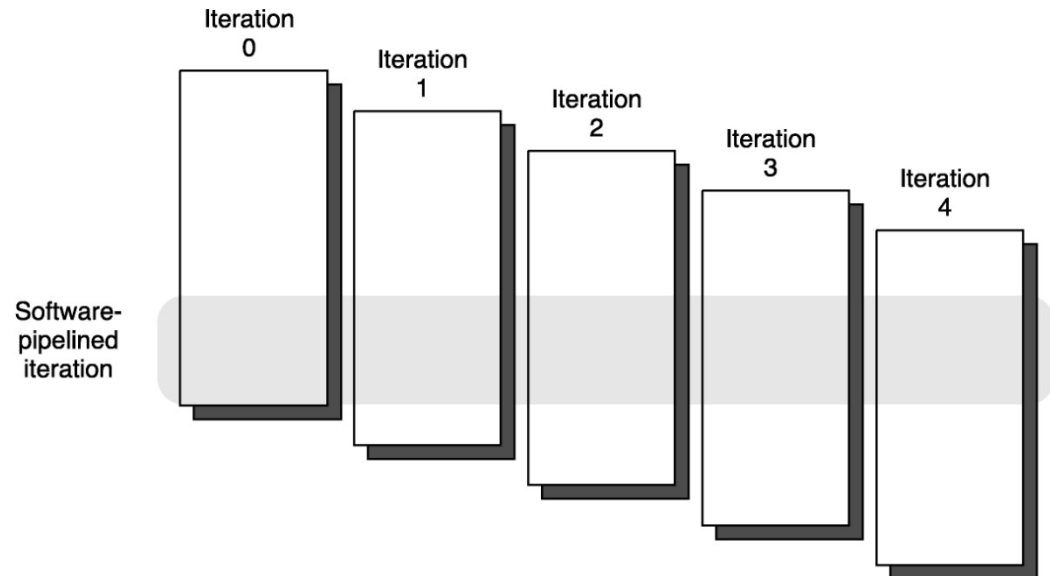
	Integer instruction		FP instruction	Clock cycle
Loop:	L.D	F0,0(R1)		1
	L.D	F6,-8(R1)		2
	L.D	F10,-16(R1)	ADD.D F4,F0,F2	3
	L.D	F14,-24(R1)	ADD.D F8,F6,F2	4
	L.D	F18,-32(R1)	ADD.D F12,F10,F2	5
	S.D	F4,0(R1)	ADD.D F16,F14,F2	6
	S.D	F8,-8(R1)	ADD.D F20,F18,F2	7
	S.D	F12,-16(R1)		8
	DADDUI	R1,R1,#-40		9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

Software Pipelining: Symbolic Loop Unrolling

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

No parallelism without loop unrolling because of dependences between instructions – requires 10 cycles (5 stalls) [see earlier slide]

Increment every array element by F2
Array begins at R1



Software Pipelining: Symbolic Loop Unrolling

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

Symbolically unroll the loop (ignoring loop overhead instructions)...

```
Iteration i:      L.D      F0,0(R1)
                  ADD.D    F4,F0,F2
                  S.D      F4,0(R1)
Iteration i+1:    L.D      F0,0(R1)
                  ADD.D    F4,F0,F2
                  S.D      F4,0(R1)
Iteration i+2:    L.D      F0,0(R1)
                  ADD.D    F4,F0,F2
                  S.D      F4,0(R1)
```

Software Pipelining: Symbolic Loop Unrolling

```
Iteration i:    L.D    F0,0(R1)
                ADD.D  F4,F0,F2
                S.D    F4,0(R1)
Iteration i+1:  L.D    F0,0(R1)
                ADD.D  F4,F0,F2
                S.D    F4,0(R1)
Iteration i+2:  L.D    F0,0(R1)
                ADD.D  F4,F0,F2
                S.D    F4,0(R1)
```

Use instructions from different iterations to create new loop...

```
Loop:    S.D    F4,16(R1)    ; stores in M[i]
        ADD.D  F4,F0,F2      ; add to M[i-1]
        L.D    F0,0(R1)      ; loads from M[i-2]
        DADDUI R1,R1,#-8
        BNE    R1,R2,Loop
```

Software Pipelining: Symbolic Loop Unrolling

```
Loop:      S.D      F4,16(R1)    ; stores in M[i]
           ADD.D    F4,F0,F2     ; add to M[i-1]
           L.D      F0,0(R1)     ; loads to M[i-2]
           DADDIU   R1,R1,#-8
           BNE      R1,R2,Loop
```

Can be scheduled by moving **DADDIU** earlier, **L.D** to branch delay slot (adjusting displacement)

```
Loop:      S.D      F4,16(R1)    ; stores in M[i]
           DADDIU   R1,R1,#-8
           ADD.D    F4,F0,F2     ; add to M[i-1]
           BNE      R1,R2,Loop
           L.D      F0,8(R1)     ; loads to M[i-2]
```

Requiring only 5 cycles per result (vs 10)

Software Pipelining: Symbolic Loop Unrolling

Need to add start-up and clean-up code before and after loop

```
Iteration i:    L.D    F0,0(R1)    }  
                ADD.D   F4,F0,F2    }  
                S.D     F4,0(R1)    }  
Iteration i+1:  L.D    F0,0(R1)    }  
                ADD.D   F4,F0,F2    }  
                S.D     F4,0(R1)    }  
Iteration i+2:  L.D    F0,0(R1)    }  
                ADD.D   F4,F0,F2    }  
                S.D     F4,0(R1)    }
```

```
Loop:    S.D    F4,16(R1)    ; stores in M[i]  
        ADD.D   F4,F0,F2    ; add to M[i-1]  
        L.D     F0,0(R1)    ; loads to M[i-2]  
        DADDUI  R1,R1,#-8  
        BNE     R1,R2,Loop
```