

Static Instruction Level Parallelism and VLIW

COE 501

Computer Architecture

Prof. Muhamed Mudawar

Computer Engineering Department

King Fahd University of Petroleum and Minerals

Presentation Outline

❖ **Loop Level Parallelism**

❖ Loop Unrolling

❖ Software Pipelining

❖ Predicated Instructions

❖ VLIW Approach

Instruction Level Parallelism (ILP)

- ❖ Overlap instruction execution to improve performance
- ❖ Approaches to increase ILP
 1. Rely on compiler technology to extract parallelism
 - ✧ Static ILP: Instruction-Level Parallelism detected at compile time
 - ✧ Multiple-issue in-order execution pipeline: ARM Cortex-A8
 - ✧ VLIW and EPIC approach: Transmeta Crusoe, Intel Itanium
 2. Rely on hardware to discover parallelism at runtime
 - ✧ Dynamically scheduled, out-of-order execution processor
 - ✧ Examples: AMD Opteron, IBM Power, Intel Core

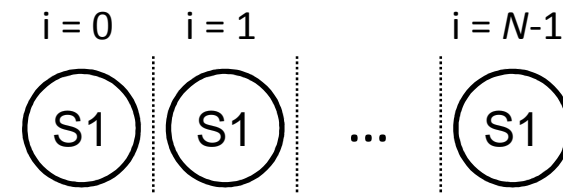
Loop-Level Parallelism

- ❖ Compiler does the analysis of loops in the source code
- ❖ Detects dependences across the loop iterations
- ❖ Loop-carried dependences
 - ✧ Whether later iterations are data dependent on earlier iterations
- ❖ NO loop-carried dependences → Parallel Loop
 - ✧ Loop iterations can run in parallel

❖ Example of a Parallel Loop

```
for (i=0; i<N; i++)
```

```
    A[i] = A[i] + B[i]; // S1
```



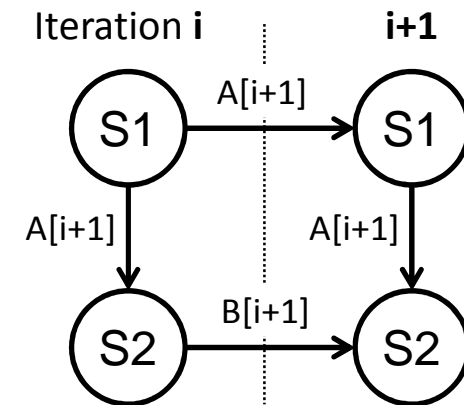
Dependency Graph

- ✧ Array references are independent across loop iterations

Loop-Carried Dependences

❖ Consider the following loop:

```
for (i=0; i < N; i++) {  
    A[i+1] = A[i] + C[i];    // S1  
    B[i+1] = B[i] + A[i+1]; // S2  
}
```



Dependency Graph

❖ Two loop-carried dependences

- ✧ S1 reads $A[i]$ computed as $A[i+1]$ in **previous iteration**
- ✧ S2 reads $B[i]$ computed as $B[i+1]$ in **previous iteration**
- ✧ Successive iterations should be executed in series

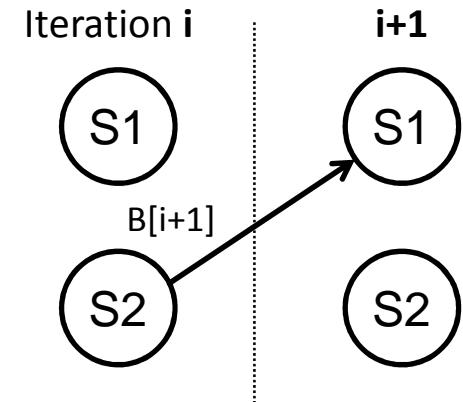
❖ There is one data dependence, which is not loop-carried

- ✧ S2 reads $A[i+1]$ computed by S1 in the **same iteration**

Non-Circular Loop-Carried Dependence

- ❖ Consider a loop like this one:

```
for (i=0; i < N; i++) {  
    A[i] = A[i] + B[i];    // S1  
    B[i+1] = C[i] + D[i]; // S2  
}
```



Dependency Graph

- ❖ There is one loop-carried dependence
 - ✧ S1 reads $B[i]$ computed as $B[i+1]$ by S2 in **previous iteration**
- ❖ Loop-carried dependence is **not circular**
 - ✧ No loop-carried dependence between S1 and S1
 - ✧ No loop-carried dependence between S2 and S2
 - ✧ S1 depends on S2, but S2 does not depend on S1

Loop Transformation

- ❖ Consider loop with no circular loop-carried dependence:

```
for (i=0; i < N; i++) {  
    A[i] = A[i] + B[i];           // S1  
    B[i+1] = C[i] + D[i];       // S2  
}
```

- ❖ This loop can be transformed and made parallel:

```
A[0] = A[0] + B[0];  
for (i=0; i < N-1; i++) {  
    B[i+1] = C[i] + D[i];       // S2  
    A[i+1] = A[i+1] + B[i+1];   // S1  
}  
B[N] = C[N-1] + D[N-1];
```

Dependence Distance

- ❖ Often a loop-carried dependence is a recurrence

```
for (i=1; i<N; i++) A[i] = A[i-1] + B[i];
```

- ❖ Dependence distance can be larger than one

```
for (i=4; i<N; i++) A[i] = A[i-4] + B[i];
```

- ❖ Loop can be unrolled to increase parallelism within iteration

```
for (i=4; i<N; i=i+4) {  
    A[i]      = A[i-4] + B[i];  
    A[i+1]    = A[i-3] + B[i+1];  
    A[i+2]    = A[i-2] + B[i+2];  
    A[i+3]    = A[i-1] + B[i+3];  
}
```


Next ...

- ❖ Instruction Level Parallelism
- ❖ **Loop Unrolling**
- ❖ Software Pipelining
- ❖ Predicated Instructions
- ❖ VLIW Approach

Example of a Parallel Loop

- ❖ Consider the following loop: adds a scalar to a vector

```
for (i=0; i < N; i++)
```

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

- ❖ Assume the following latencies (stall cycles before use)

Instruction Producing Result	Instruction Using Result	Latency
FP operation	Another FP operation	3 stall cycles
FP operation	Store	2 stall cycles
Load	FP operation	1 stall cycle
Load	Store	0
Integer ALU operation	Integer ALU operation	0
Integer ALU operation	Branch instruction	0

Translate the Loop into MIPS Code

- ❖ Initially, R1 contains the address of $x[0]$
- ❖ Register R1 is used as a pointer to $x[i]$
- ❖ Register R2 contains address of $x[N]$
- ❖ Register F2 contains scalar value s

; Initialize R1, R2, F2 before entering loop


```
Loop: L.D      F0, 0(R1)      ; load F0 = x[i]
      ADD.D    F4, F0, F2     ; F4 = x[i] + s
      S.D      F4, 0(R1)     ; store x[i] = F4
      DADDUI   R1, R1, 8      ; point to next x[i]
      BNE      R1, R2, Loop  ; branch if (R1 != R2)
```

- ❖ Loop block has five instructions (including branch)

Showing Stall Cycles

Loop:

```
1  L.D      F0, 0(R1)      ; load F0 = x[i]
2  stall
3  ADD.D    F4, F0, F2      ; F4 = x[i] + s
4  stall
5  stall
6  S.D      F4, 0(R1)      ; store x[i] = F4
7  DADDUI   R1, R1, 8       ; point to next x[i]
8  BNE      R1, R2, Loop    ; branch if (R1 != R2)
```




- ❖ Loop branch is predicted to be always taken (zero delay)
- ❖ Clock cycles per iteration = 8
- ❖ Stall cycles per iteration = 3, CPI = $8/5 = 1.6$

Reducing the Stall Cycles

❖ We can schedule the loop to reduce the stall cycles

Loop:

```
1    L.D      F0, 0(R1)      ; load F0 = x[i]
2    DADDUI   R1, R1, 8      ; point to next x[i]
3    ADD.D    F4, F0, F2     ; F4 = x[i] + s
4    stall
5    stall
6    S.D      F4, -8(R1)     ; store x[i] = F4
7    BNE      R1, R2, Loop   ; branch if (R1 != R2)
```



❖ Number of cycles is reduced to 7 per iteration

❖ $CPI = 7 / 5 = 1.4$ (how to make it run faster?)

Basic Block

- ❖ Basic Block: straight-line code sequence
 - ✧ No branches-in except at the entry of the basic block
 - ✧ No branches-out except at the exit of the basic block
- ❖ Basic block can be a loop block with a backward branch
- ❖ Basic block is quite small
 - ✧ Average dynamic branch frequency = 15% to 25%
 - ✧ Four to seven instructions execute between a pair of branches
- ❖ Instructions in basic block likely to depend on each other
- ❖ Must exploit ILP across multiple basic blocks
 - ✧ To obtain substantial performance enhancements

Unroll Loop Four Times

Loop:

1	L.D	F0, 0(R1)	; load F0 = x[i]
3	ADD.D	F4, F0, F2	; F4 = x[i] + s
6	S.D	F4, 0(R1)	; store F4 at x[i]
7	L.D	F6, 8(R1)	; load F6 = x[i+1]
9	ADD.D	F8, F6, F2	; F8 = x[i+1] + s
12	S.D	F8, 8(R1)	; store F8 at x[i+1]
13	L.D	F10, 16(R1)	; load F10 = x[i+2]
15	ADD.D	F12, F10, F2	; F12 = x[i+2] + s
18	S.D	F12, 16(R1)	; store F12 at x[i+2]
19	L.D	F14, 24(R1)	; load F14 = x[i+3]
21	ADD.D	F16, F14, F2	; F16 = x[i+3] + s
24	S.D	F16, 24(R1)	; store F16 at x[i+3]
25	DADDUI	R1, R1, 32	; point to x[i+4]
26	BNE	R1, R2, Loop	; branch if (R1 != R2)

❖ 26 clock cycles for 4 iterations = 6.5 cycles per iteration

Unrolled Loop with Scheduling

Loop:			; Zero Stall Cycles
1	L.D	F0, 0(R1)	; load F0 = x[i]
2	L.D	F6, 8(R1)	; load F6 = x[i+1]
3	L.D	F10, 16(R1)	; load F10 = x[i+2]
4	L.D	F14, 24(R1)	; load F14 = x[i+3]
5	ADD.D	F4, F0, F2	; F4 = x[i] + s
6	ADD.D	F8, F6, F2	; F8 = x[i+1] + s
7	ADD.D	F12, F10, F2	; F12 = x[i+2] + s
8	ADD.D	F16, F14, F2	; F16 = x[i+3] + s
9	S.D	F4, 0(R1)	; store F4 at x[i]
10	S.D	F8, 8(R1)	; store F8 at x[i+1]
11	S.D	F12, 16(R1)	; store F12 at x[i+2]
12	S.D	F16, 24(R1)	; store F16 at x[i+3]
13	DADDUI	R1, R1, 32	; point to x[i+4]
14	BNE	R1, R2, Loop	; branch if (R1 != R2)

❖ 14 clock cycles for 4 iterations = 3.5 cycles per iteration

Unrolled Loop Details

- ❖ Unroll loop k times to make k copies of the loop body
- ❖ If N is not multiple of k then generate a pair of loops
 - ✧ First loop is the unrolled body that iterates (N / k) times
 - ✧ Second loop executes $(N \% k)$ times (identical to original loop)
- ❖ Another choice is to increase N to be multiple of k
 - ✧ Small additional overhead in array size and computation
- ❖ Reduce loop overhead

DADDUI R1, R1, 8
DADDUI R1, R1, 8
DADDUI R1, R1, 8
DADDUI R1, R1, 8

Merged into:

DADDUI R1, R1, 32 (Copy Propagation)

Adjust memory addresses in L.D and S.D

Advantages of Loop Unrolling

- ❖ Increases the size of the basic block
 - ✧ From 5 to 14 instructions in the previous example
 - ✧ Reduce branch frequency: one BNE per 14 instructions
- ❖ Reduces the loop overhead
 - ✧ Only 3 instructions to compute $x[i]$: L.D, ADD.D, and S.D
 - ✧ Loop overhead: DADDUI to advance pointer and BNE
 - ✧ Loop overhead is reduced from 2 to 0.5 cycles per iteration
- ❖ Increases Instruction Level Parallelism (ILP)
 - ✧ More independent instructions to execute in parallel
 - ✧ Hide the latency of load instructions
 - ✧ Hide the latency of floating-point instructions

Limits to Loop Unrolling

❖ Growth in Code Size

- ✧ Original loop was only 5 instructions (in previous example)
- ✧ Unrolled loop is 14 instructions after unrolling 4 iterations
- ✧ Occupies I-Cache space (larger loop body with more unrolling)

❖ Register Pressure

- ✧ Unrolled loop allocates more registers to increase ILP
- ✧ Additional registers are allocated to eliminate name dependences
- ✧ Can easily run out of registers when unrolling many iterations

❖ Benefit of unrolling decreases with additional unrolling

- ✧ Cycles per iteration = 3.5 (four unrolled iterations)
- ✧ Cycles per iteration = 3.25 (eight unrolled iterations)

Summary of Loop Unrolling

- ❖ Determine whether loop unrolling is useful
 - ✧ Find whether loop iterations are independent
- ❖ Use different registers to avoid name dependences
- ❖ Adjust the loop termination and iteration code
 - ✧ Reduce the loop overhead and loop branching
- ❖ Determine that loads and stores are independent
 - ✧ Independent loads and stores can be interchanged
 - ✧ Requires analyzing memory addresses
- ❖ Schedule the code preserving data dependences
 - ✧ Unrolled loop should compute the same result as original one

Next ...

- ❖ Instruction Level Parallelism
- ❖ Loop Unrolling
- ❖ **Software Pipelining**
- ❖ Predicated Instructions
- ❖ VLIW Approach

Software Pipelining


- ❖ Called also: **Symbolic Loop Unrolling**
- ❖ Compiler technique that restructures loops
- ❖ Choose instructions from multiple iterations of original loop
 - ✧ Select instructions from independent iterations
 - ✧ Separate the dependent instructions within an iteration
 - ✧ The idea is to make software pipelined loop run without stalls
- ❖ **Startup Code**
 - ✧ Execute instructions left out from the first original loop iterations
- ❖ **Finish Code**
 - ✧ Execute instructions left out from the last original loop iterations

Original Loop Example

```
for (i=0; i < N; i++) x[i] = x[i] + s;
```

Loop:

```
1  L.D      F0, 0(R1)      ; load F0 = x[i]
2  stall
3  ADD.D    F4, F0, F2      ; F4 = x[i] + s
4  stall
5  stall
6  S.D      F4, 0(R1)      ; store x[i] = F4
7  DADDUI   R1, R1, 8       ; point to next x[i]
8  BNE      R1, R2, Loop    ; branch if (R1 != R2)
```



❖ Loop branch is predicted to be always taken (zero delay)

❖ Clock cycles per iteration = 8

Example of Software Pipelining

Unroll 3 Iterations

To separate 3 instructions

L.D → ADD.D → S.D

Iteration i:

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

Iteration i+1:

```
L.D    F0, 8(R1)
ADD.D  F4, F0, F2
S.D    F4, 8(R1)
```

Iteration i+2:

```
L.D    F0, 16(R1)
ADD.D  F4, F0, F2
S.D    F4, 16(R1)
```

Software Pipelined Loop

Two fewer loop iterations

Initialize R2 = address of x[N-2]

```
L.D    F0, 0(R1)
ADD.D  F4, F0, F2
L.D    F0, 8(R1)
```

} Startup Code

Loop:

```
S.D    F4, 0(R1)
ADD.D  F4, F0, F2
L.D    F0, 16(R1)
```

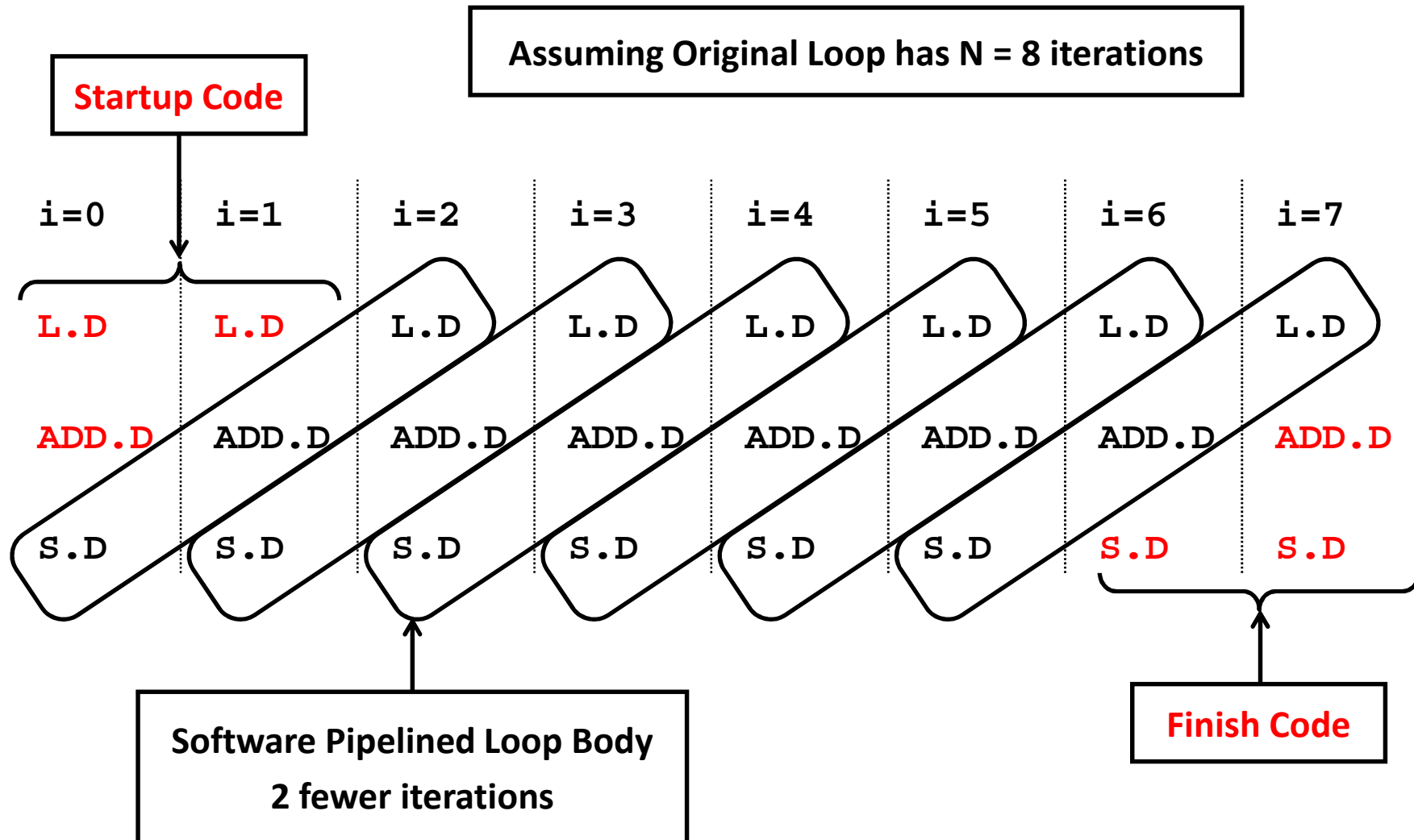
} Reuse of F4, F0

```
DADDUI R1, R1, 8
BNE    R1, R2, Loop
```

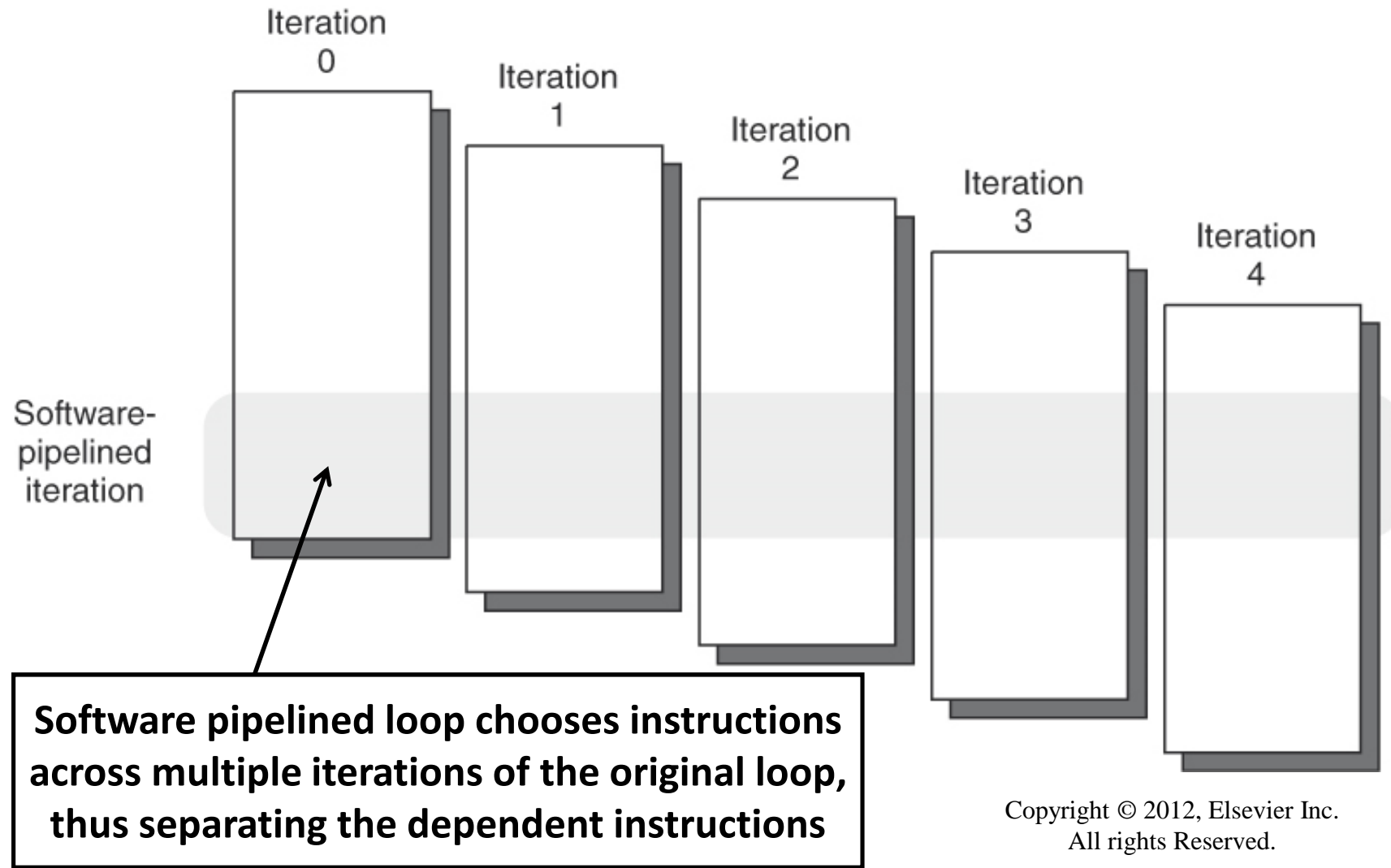
```
S.D    F4, 0(R1)
ADD.D  F4, F0, F2
S.D    F4, 8(R1)
```

} Finish Code

Illustrating Software Pipelining



Software Pipelining Summary



Next ...

- ❖ Instruction Level Parallelism
- ❖ Loop Unrolling
- ❖ Software Pipelining
- ❖ **Predicated Instructions**
- ❖ VLIW Approach

Predication

- ❖ Problem: Branches are sometimes difficult to predict
 - ✧ When the behavior is not well known
 - ✧ Mispredicted branches reduce performance
 - ✧ Control dependences may severely limit ILP
 - ✧ Compiler techniques may not work with some branches
- ❖ Solution: Eliminate branches with predication
- ❖ Extend the instruction set to include
 - ✧ Conditional move instructions, or
 - ✧ Fully predicated instructions
- ❖ Convert control dependences into data dependences

Example of Nested If Statement

```
for (i=0; i < N; i++) {  
    if (a[i] < b[i]) a[i] = b[i];  
}
```

```
loop: LW      R2, 0(R4)      ; load R2 = a[i]  
      LW      R3, 0(R5)      ; load R3 = b[i]  
      SLT     R1, R2, R3     ; R1 = (a[i] < b[i])  
      BEQ     R1, R0, L1     ; Branch if false  
      SW      R3, 0(R4)      ; store a[i] = R3  
L1:    ADDU    R4, R4, 4      ; R4 = address a[i+1]  
      ADDU    R5, R5, 4      ; R5 = address b[i+1]  
      BNE     R4, R8, loop   ; R8 = address a[N]
```

❖ **BEQ** is hard to predict for each iteration

Example of Conditional Move

- ❖ Conditional move (MOVZ) eliminates BEQ instruction
- ❖ SW was **control-dependent** on **BEQ** (previous slide)
- ❖ SW is now **data-dependent** on **MOVZ**

```
loop:  LW      R2, 0(R4)      ; load R2 = a[i]
        LW      R3, 0(R5)      ; load R3 = b[i]
        SLT     R1, R2, R3      ; R1 = (a[i] < b[i])
        MOVZ    R3, R2, R1      ; if (R1==0) R3 ← R2
        SW      R3, 0(R4)      ; store a[i] = R3
L1:    ADDU     R4, R4, 4        ; R4 = address a[i+1]
        ADDU     R5, R5, 4        ; R5 = address b[i+1]
        BNE     R4, R8, loop ; R8 = address a[N]
```

MIPS Conditional Move Instructions

Most basic form of predication

MOVZ	Rd, Rs, Rt ; if (Rt == 0) Rd ← Rs
MOVN	Rd, Rs, Rt ; if (Rt != 0) Rd ← Rs
MOVT	Rd, Rs, cc ; if (cc == 1) Rd ← Rs
MOVF	Rd, Rs, cc ; if (cc == 0) Rd ← Rs
MOVZ.S	Fd, Fs, Rt ; if (Rt == 0) Fd ← Fs
MOVZ.D	Fd, Fs, Rt ; if (Rt == 0) Fd ← Fs
MOVN.S	Fd, Fs, Rt ; if (Rt != 0) Fd ← Fs
MOVN.D	Fd, Fs, Rt ; if (Rt != 0) Fd ← Fs
MOVT.S	Fd, Fs, cc ; if (cc == 1) Fd ← Fs
MOVT.D	Fd, Fs, cc ; if (cc == 1) Fd ← Fs
MOVF.S	Fd, Fs, cc ; if (cc == 0) Fd ← Fs
MOVF.D	Fd, Fs, cc ; if (cc == 0) Fd ← Fs

Full Predication

❖ Conditional instructions

- ✧ Associate a qualifying predicate (qp) with each instruction
`(qp) instruction ; if (qp) execute instruction`
- ✧ Qualifying predicate is checked as part of instruction execution
- ✧ If (qp) is true → instruction executes normally
- ✧ If (qp) is false → instruction becomes a no-op

❖ Full predication can eliminate most non-loop branches

- ✧ Simplifies instruction scheduling

❖ Only few architectures support full predication

- ✧ Intel Itanium uses qualifying predicates for conditional execution
- ✧ ARM uses condition codes for conditional execution

Example of Predicated Instruction

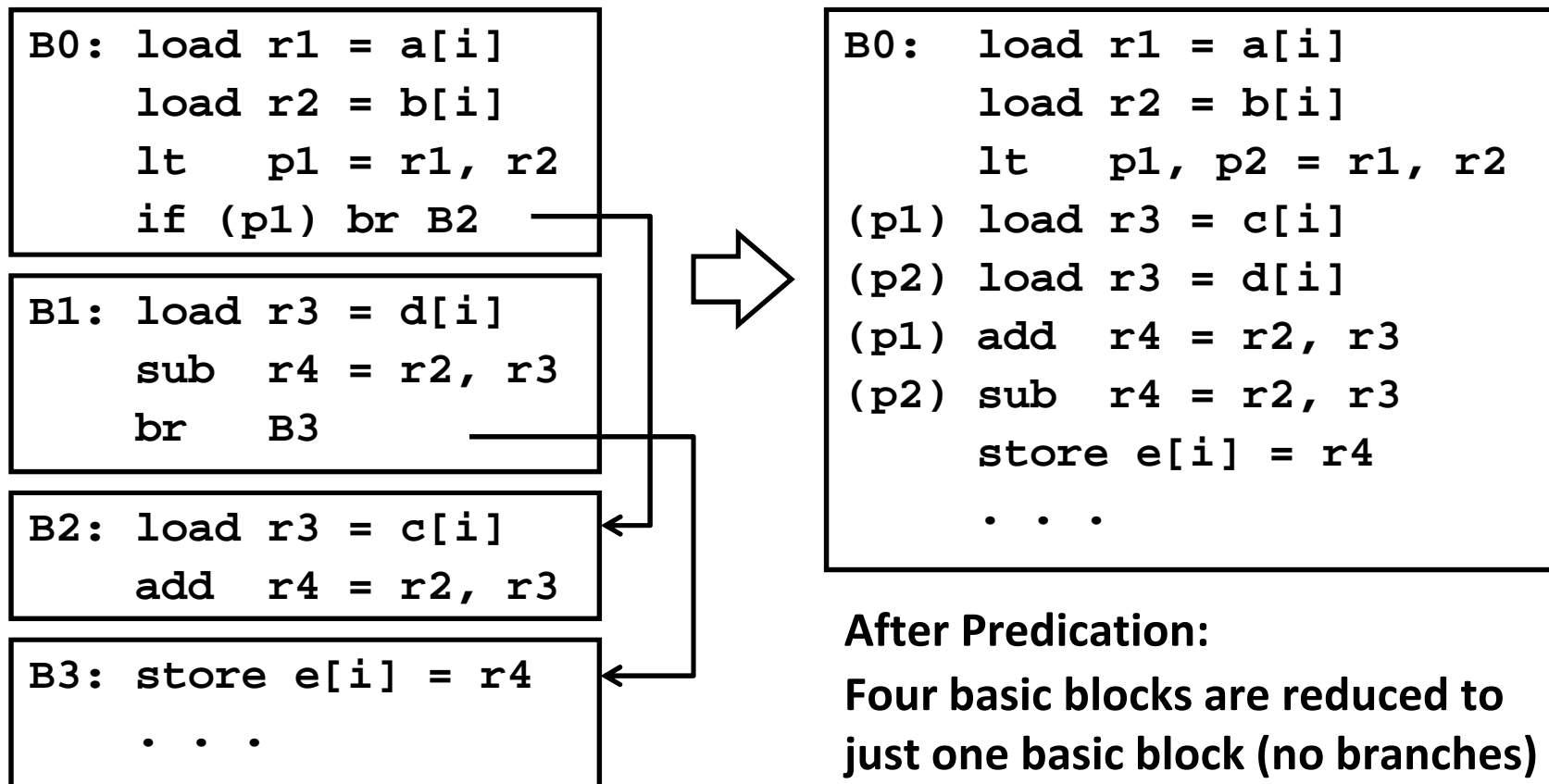
```
for (i=0; i < N; i++) {  
    if (a[i] < b[i]) a[i] = b[i];  
}
```

```
loop: LW      R2, 0(R4)      ; load R2 = a[i]  
      LW      R3, 0(R5)      ; load R3 = b[i]  
      SLT     p1, R2, R3      ; p1 = (a[i] < b[i])  
(p1) SW      R3, 0(R4)      ; if (p1) a[i] = R3  
L1:   ADDU     R4, R4, 4      ; R4 = address a[i+1]  
      ADDU     R5, R5, 4      ; R5 = address b[i+1]  
      BNE     R4, R8, loop    ; R8 = address a[N]
```

- ❖ Conditional SW instruction eliminates BEQ and MOVZ
- ❖ MIPS does not support predication (just for illustration)

Example of If-Else Conversion

```
if (a[i] < b[i]) e[i] = b[i] + c[i];  
else e[i] = b[i] - d[i];
```



Predication Advantages/Complications

❖ Predication Advantages

- ✧ Eliminates hard-to-predict branches
- ✧ Converts control into data dependences (if-else conversions)
- ✧ Increases basic block size and instruction level parallelism
- ✧ Does not generate an exception when predicate is false
- ✧ Full predication is more useful than conditional move

❖ Complications

- ✧ Deciding when to terminate a predicated instruction
 1. Early in the pipeline when issued (if predicate value is known)
 2. Later in the pipeline before writing result (consumes resources)
- ✧ Forwarding predicate values complicates the implementation

Next ...

- ❖ Instruction Level Parallelism
- ❖ Loop Unrolling
- ❖ Software Pipelining
- ❖ Predicated Instructions
- ❖ **VLIW Approach**

Getting CPI below 1

- ❖ $CPI \geq 1$ if only 1 instruction is issued every clock cycle
- ❖ Getting CPI below 1 requires multiple-issue
- ❖ Multiple-issue processors come in 3 flavors
 1. Statically-scheduled in-order execution superscalar processors
 2. Dynamically-scheduled out-of-order execution processors
 3. VLIW (very long instruction word) processors
- ❖ VLIW processors
 - ✧ Issue a fixed number of instructions each cycle
 - ✧ Formatted as a fixed instruction group
 - ✧ Parallelism among instructions explicitly indicated

VLIW: Very Long Instruction Word

- ❖ Each long instruction has multiple operations
 - ✧ Intel Itanium calls it an instruction group
 - ✧ No register data dependencies
 - ✧ All the instructions in a group could be executed in parallel
 - ✧ Compiler must explicitly indicate boundary between groups
 - ✧ Transmeta: grouping is called a molecule (atoms are operations)
- ❖ All operations in the long instruction are parallel
 - ✧ The long instruction word has slots for many operations
 - ✧ Slots for memory, floating-point, integer, and branch units
 - ✧ To simplify the decoding and issuing of operations
 - ✧ Unused slots are filled with no-ops

Recall: Loop Unrolling

```
for (i=0; i < N; i++) x[i] = x[i] + s;
```

Loop:		; Unrolled four times
1	L.D	F0, 0(R1) ; load F0 = x[i]
2	L.D	F6, 8(R1) ; load F6 = x[i+1]
3	L.D	F10, 16(R1) ; load F10 = x[i+2]
4	L.D	F14, 24(R1) ; load F14 = x[i+3]
5	ADD.D	F4, F0, F2 ; F4 = x[i] + s
6	ADD.D	F8, F6, F2 ; F8 = x[i+1] + s
7	ADD.D	F12, F10, F2 ; F12 = x[i+2] + s
8	ADD.D	F16, F14, F2 ; F16 = x[i+3] + s
9	S.D	F4, 0(R1) ; store F4 at x[i]
10	S.D	F8, 8(R1) ; store F8 at x[i+1]
11	S.D	F12, 16(R1) ; store F12 at x[i+2]
12	S.D	F16, 24(R1) ; store F16 at x[i+3]
13	DADDUI	R1, R1, 32 ; point to x[i+4]
14	BNE	R1, R2, Loop ; branch if (R1 != R2)

Loop Unrolling in VLIW (3 Slots)

- ❖ Unroll loop 4 times and schedule the code on 3 slots
- ❖ Cycles per iteration = $10/4 = 2.5$ (2 stall cycles: cc5, cc8)
- ❖ Fills $14 / 24 = 58\%$ of slots

Slot 0 = Memory	Slot 1 = FPU	Slot 2 = ALU Branch	Cycle
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
S.D F4,0(R1)	ADD.D F12,F10,F2		6
S.D F8,8(R1)	ADD.D F16,F14,F2		7
S.D F12,16(R1)		DADDUI R1,R1,32	9
S.D F16,24(R1)		BNE R1,R2,Loop	10

Scheduling Code to Minimize Cycles

- ❖ Cycles per iteration = $9 / 4 = 2.25$ (zero stall cycles)
- ❖ Fills $14 / 27 = 52\%$ of slots

Slot 0 = Memory	Slot 1 = FPU	Slot 2 = ALU Branch	Cycle
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
	ADD.D F12,F10,F2		5
S.D F4,0(R1)	ADD.D F16,F14,F2		6
S.D F8,8(R1)			7
S.D F12,16(R1)		DADDUI R1,R1,32	8
S.D F16,24(R1)		BNE R1,R2,Loop	9

Problems with VLIW

❖ Increase in code size

- ✧ Ambitiously unrolling loop to increase ILP
- ✧ No-ops are wasted bits in the instruction encoding
- ✧ Complex encoding and grouping of instructions

❖ Increased compiler complexity

- ✧ Compiler must unroll loops and schedule code
- ✧ Compiler closely tied to processor implementation
- ✧ Binary code compatibility can be a problem in VLIW processors

❖ Limited static instruction level parallelism

❖ Hardware must still detect hazards

- ✧ VLIW does not fundamentally solve instruction scheduling