



Chapter 3: Instruction-Level Parallelism

1

Instruction-Level Parallelism

- Straightforward pipelining overlaps the execution of multiple independent instructions (processing overlapped across pipeline stages).
- Instruction-level parallelism: Exploiting parallelism in instructions to improve performance.
- Generally invisible to the programmer (but not the compiler!)

2

Improving Performance

Pipeline CPI

$$\text{Effective CPI} = \text{Ideal CPI} + \text{Structural Stalls} + \text{RAW Stalls} + \text{WAR Stalls} + \text{Control Stalls}$$

To improve performance, we can tackle any one of these terms.

Control stalls	Loop unrolling, dynamic branch prediction, speculation
RAW stalls	Scheduling, scoreboarding, memory disambiguation
WAR stalls	Scheduling with register renaming
Data stalls	Dependence analysis, software pipelining, speculation
Ideal CPI	Multiple issue, dependence analysis, software pipelining

3

Basic Unit of Instruction-Level Parallelism

- Sequence at instruction level is a *basic block*.
- Basic block (BB)
 - Straight-line code with no branches into the sequence except at the top and no branches out except at the bottom.
- Procedure represented as a control flow graph on nodes that are basic blocks

Code Sequence

```
if (x < y)
    A
else
    B
C
```

4

Basic Unit of Instruction-Level Parallelism

- Sequence at instruction level is a *basic block*.
- Basic block (BB)
 - Straight-line code with no branches into the sequence except at the top and no branches out except at the bottom.
- Procedure represented as a control flow graph on nodes that are basic blocks

Code Sequence

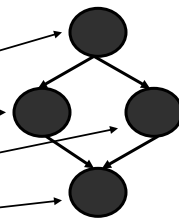
```
if (x < y)
```

```
  A
```

```
else
```

```
  B
```

```
C
```



5

Basic Blocks

```
L0:  ADD r1,r2,r3
      SUB r4,r1,r5
      BEQZ r4,L0
      ADD r3,r2,r1
      BNEZ r3,L1
      OR r8,r10,r11
L1:  AND r5,r6,r7
      OR r3,r2,r1
      BEQZ r5,L2
```

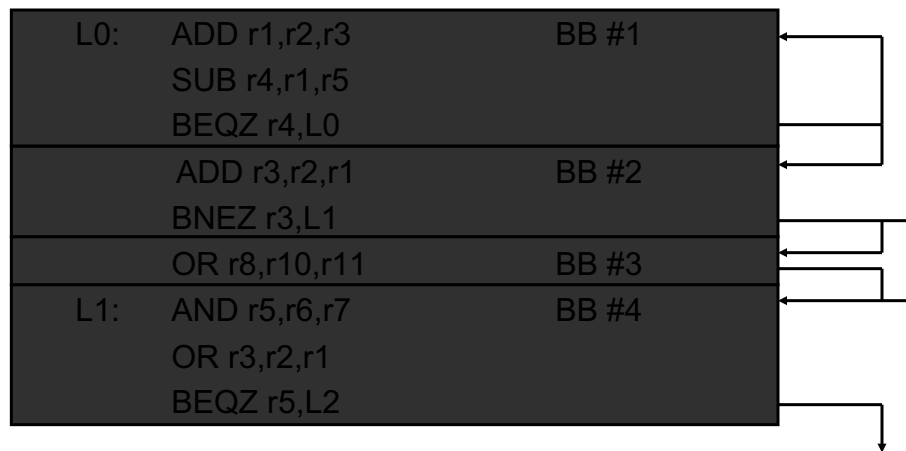
6

Basic Blocks

L0:	ADD r1,r2,r3 SUB r4,r1,r5 BEQZ r4,L0	BB #1
	ADD r3,r2,r1 BNEZ r3,L1	BB #2
	OR r8,r10,r11	BB #3
L1:	AND r5,r6,r7 OR r3,r2,r1 BEQZ r5,L2	BB #4

7

Control Flow Graph



8

Parallelism in Basic Blocks

- Amount of parallelism within a BB is relatively small.
 - Average BB size is 4-6 instructions
 - Parallelism less b/c instructions are typically interdependent
- Consider branches 20% of instruction mix. How big is the typical basic block??

20% = 1/5 instructions is a branch → 5

9

Exploiting Parallelism

- BB is too small and interdependent instructions.
- Exploit parallelism *across* basic blocks
- ILP frequently found in loops, across iterations of the loop. E.g.:

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

- Every iteration is independent and can be fully overlapped with every other one

10

Loop Unrolling

- Keep pipeline full with independent instructions that can execute simultaneously
 - Operation latency
 - Finding independent instructions
- Expose parallelism across loop iterations
 - Compiler pipeline scheduling
 - Hardware dynamic scheduling

11

Loop Unrolling Example

- DLX pipeline with latencies

FP ALU op → FP ALU op	3 cycles
FP ALU op → Store double	2 cycles
Load double → FP ALU op	1 cycle
Load double → Store double	0 cycle

- Simple loop (array X and scalar S are doubles)

```
for (I = 1; I < 1000; I++)  
    x[I] = x[I] + s;
```

12

Straightforward DLX for Example





```

Loop:   LD      F0,0(R1)      ; F0=array elem
        ADDD   F4,F0,F2      ; add scalar S
        SD     0(R1),F4      ; store X[I]
        SUBI   R1,R1,8       ; decrement ptr
                                   ; double=8bytes
        BNEZ  R1,Loop        ; branch R1!=0
    
```

- R1 address of array (highest element), F2 contains the scalar S, and &X[1]=8.
- How fast does this code run on DLX without scheduling?

13

Latency of Straightforward Example

Loop:		LD	F0,0(R1)	; F0=array elem	1
				stall	2
		ADDD	F4,F0,F2	; add scalar S	3
				stall	4
				stall	5
		SD	0(R1),F4	; store X[I]	6
		SUBI	R1,R1,8	; decrement ptr	7
				stall	8
		BNEZ	R1,Loop	; branch R1!=0	9
				stall	10

- Total of 5 stall cycles

14

Latency of Scheduled Example

```
Loop:  ↻ LD    F0,0(R1)    ; F0=array elem    1
      stall                2
      SUBI   R1,R1,8      ; decrement ptr    3
      ADDD  F4,F0,F2      ; add scalar S     4
      BNEZ  R1,Loop       ; branch R1!=0     5
      SD    8(R1),F4      ; store X[I]       6
```

- Interchanged SUBI and SD - compiler has to notice that offset is affected by swapping SUBI and SD
- SUBI and BNEZ in delay of ADDD, and SD in delay slot of branch

15

Scheduled Example

- Computation: 3 cycles (LD, ADDD, SD)
- Branch overhead: 3 cycles (SUBI, BNEZ)
- 50% of cycles for each iteration spent in branching
- Unroll loop to lower branch overhead - replicate loop body several times and remove the extraneous branches
- May also expose more scheduling opportunities - chances to overlap independent instructions.

16

Unrolled Example

```
Loop:  LD    F0, 0(R1)
      ADDD  F4, F0, F2
      SD    0(R1), F4
      LD    F6, -8(R1)
      ADDD  F8, F6, F2
      SD    -8(R1), F8
      LD    F10, -16(R1)
      ADDD  F12, F10, F2
      SD    -16(R1), F12
      LD    F14, -24(R1)
      ADDD  F16, F14, F2
      SD    -24(R1), F16
      SUBI  R1, R1, 32
      BNEZ  R1, Loop
```

17

Unrolled Example

```
Loop:  { LD    F0, 0(R1)
      { ADDD  F4, F0, F2
      { SD    0(R1), F4
      { LD    F6, -8(R1)
      { ADDD  F8, F6, F2
      { SD    -8(R1), F8
      { LD    F10, -16(R1)
      { ADDD  F12, F10, F2
      { SD    -16(R1), F12
      { LD    F14, -24(R1)
      { ADDD  F16, F14, F2
      { SD    -24(R1), F16
      SUBI  R1, R1, 32
      BNEZ  R1, Loop
```




Unroll factor is 4
assuming that the
trip count is a
multiple of 32.

Register names
changed to avoid
dependences.

Offsets adjusted for
the change in the
iteration count.

18

Unrolled Example

Loop:	LD	F0, 0(R1)	<p>Loop evaluation eliminated for 3 iterations (removes 6 instructions)</p>
	ADDD	F4, F0, F2	
	SD	0(R1), F4	
	LD	F6, -8(R1)	
	ADDD	F8, F6, F2	
	SD	-8(R1), F8	
	LD	F10, -16(R1)	
	ADDD	F12, F10, F2	
	SD	-16(R1), F12	
	LD	F14, -24(R1)	
	ADDD	F16, F14, F2	
	SD	-24(R1), F16	
	SUBI	R1, R1, 32	
	BNEZ	R1, Loop	

19

Unrolled Example

Loop:	LD	F0, 0(R1)	<p>The loop takes 28 cycles per iteration, or 7 cycles (28/4=7) per original loop body.</p> <p>Branches removed, so multiple bodies can be scheduled together</p>
2	ADDD	F4, F0, F2	
4	SD	0(R1), F4	
2	LD	F6, -8(R1)	
4	ADDD	F8, F6, F2	
4	SD	-8(R1), F8	
2	LD	F10, -16(R1)	
4	ADDD	F12, F10, F2	
4	SD	-16(R1), F12	
2	LD	F14, -24(R1)	
4	ADDD	F16, F14, F2	
4	SD	-24(R1), F16	
2	SUBI	R1, R1, 32	
2	BNEZ	R1, Loop	

20

Scheduled Unrolled Example

<p>Loop:</p> <pre>LD F0, 0(R1) LD F6, -8(R1) LD F10, -16(R1) LD F14, -24(R1) ADDD F4, F0, F2 ADDD F8, F6, F2 ADDD F12, F10, F2 ADDD F16, F14, F2 SD 0(R1), F4 SD -8(R1), F8 SUBI R1, R1, 32</pre> <p><u>Offset adjustments</u> $-32+16 = -16$ $-32+8 = -24$</p>	<pre>LD F6, -8(R1) LD F10, -16(R1) LD F14, -24(R1) ADDD F4, F0, F2 ADDD F8, F6, F2 ADDD F12, F10, F2 ADDD F16, F14, F2 SD 0(R1), F4 SD -8(R1), F8 SUBI R1, R1, 32 SD 16(R1), F12 BNEZ R1, Loop SD 8(R1), F16</pre>	<p>Independent iterations used to mask latency.</p> <p>The loop takes 14 cycles per iteration, or 3.5 cycles per original loop body ($14/4=3.5$).</p> <p>SUBI moved above the last two SDs, so the offset is adjusted</p>
--	---	--

21

Loop Unrolling

- When upper bound isn't known, we can precondition the loop iteration count.
 - Preconditioning loop: Replicate original loop with a trip count of $(n \text{ mod } F)$, where n is iteration variable and F is unroll factor.
 - Unrolled loop: Unroll original loop by F with a trip count of n/F .
- Code size growth can be significant with unrolling. What about the I-cache????

22

Summary of Unrolling Example

- 1) Determine we could move SD
- 2) Determine loop bodies are independent and there is a benefit to unrolling
- 3) Change register names to avoid name conflicts
- 4) Eliminate extra branches, adjust trip count
- 5) Determine loads and stores can be interchanged in the unrolled loop
- 6) Schedule the code, preserving dependences

23

Dependences

- The key to the unrolling example was finding *independent instructions* that can execute in parallel.
- ***Dependences constrain ILP***
- Dependence types
 - Data (dependence on values)
 - Name (conflicts on names)
 - Control (dynamic data flow dependent on branches)

24

Data Dependences

- Instruction j dependent on i if:
 $i \rightarrow j$ or $i \rightarrow k, k \rightarrow j$
 (“ \rightarrow ” is the producer-consumer relation)
- This is a “true dependence”

```
Loop:   LD      F0, 0(R1)
        ADDD   F4, F0, F2
        SD     0(R1), F4
        ...
        SUBI   R1, R1, 8
        ...
        BNEZ   R1, Loop
```

25

Data Dependences

- Instruction j dependent on i if:
 $i \rightarrow j$ or $i \rightarrow k, k \rightarrow j$
 (“ \rightarrow ” is the producer-consumer relation)
- This is a “true dependence”

```
Loop:   LD      F0, 0(R1)
        ADDD   F4, F0, F2
        SD     0(R1), F4
        ...
        SUBI   R1, R1, 8
        ...
        BNEZ   R1, Loop
```

26

Data Dependent Instructions

- Can't fully overlap data dependent instrs. due to producer-consumer relationship (RAW hazards).
- Between registers or between memory locations

```
LD A
C=A+B
ST C
...
LD C
D=C*F
ST D
...
```



Data dependence through
memory location C

27

Data Dependent Instructions

- Data dependences: Property of programs!
- Detecting and handling hazards: Property of a pipeline!
 - E.g., Data dependence means hazard possible, actual behavior (length, detection) of hazard is pipeline related.

28

Overcoming Data Dependences

- Maintain the dependence but remove (or reduce the impact) of the hazard
- Eliminate dependences by compiler scheduling

```
Loop:    ...
         SUBI   R1, R1, 8
         LD     F6, 0(R1)
         ADDD  F8, F6, F2
         SD    0(R1), F8
         SUBI   R1, R1, 8
         ...
```

29

Overcoming Data Dependences

- Maintain the dependence but remove (or reduce the impact) of the hazard
- Eliminate dependences by compiler scheduling

```
Loop:    ...
         SUBI   R1, R1, 8
         LD     F6, 0(R1)
         ADDD  F8, F6, F2
         SD    0(R1), F8
         SUBI   R1, R1, 8
         ...
```

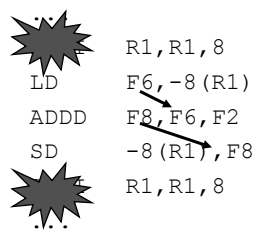
In this case, the data dependences cause execution to be serialized

30

Overcoming Data Dependences

- Maintain the dependence but remove (or reduce the impact) of the hazard
- Eliminate dependences by compiler scheduling

Loop:



Compiler folds computation of offsets in LD and SD and the iteration count decrement into one SUBI.

Offsets adjustment by 8

31

Overcoming Data Dependences

- Eliminating data dependences - requires knowledge of program structure (data flow analysis)
- Hence, the compiler applies code transformations to eliminate the dependences
- Hardware (and the compiler) can reduce the impact of data dependences by mitigating effect of the hazard

32

Name Dependences

- Antidependence - j writes same location as i and j finishes before i (WAR hazard)
- Output dependence - i and j write the same location (WAW hazard)
- Not “true dependences” because they represent conflicts and not actual data flow.

33

Overcoming Name Dependences

- Name dependences - represent conflicts for the same name
- Eliminate by changing names!
- For registers - “register renaming” - done by compiler or hardware

34

Renaming in the Unroll Example

```

Loop:   LD      F0, 0(R1)
        ADDD   F4, F0, F2
        SD     0(R1), F4
        LD     F0, -8(R1)
        ADDD   F4, F0, F2
        SD     -8(R1), F4
        LD     F0, -16(R1)
        ADDD   F4, F0, F2
        SD     -16(R1), F4
        LD     F0, -24(R1)
        ADDD   F4, F0, F2
        SD     -24(R1), F4
        SUBI   R1, R1, 32
        BNEZ   R1, Loop
    
```

Where are the name dependences in this loop?

35

Renaming in the Unroll Example

```

Loop:   LD      F0, 0(R1)
        ADDD   F4, F0, F2
        SD     0(R1), F4
        LD     F0, -8(R1)
        ADDD   F4, F0, F2
        SD     -8(R1), F4
        LD     F0, -16(R1)
        ADDD   F4, F0, F2
        SD     -16(R1), F4
        LD     F0, -24(R1)
        ADDD   F4, F0, F2
        SD     -24(R1), F4
        SUBI   R1, R1, 32
        BNEZ   R1, Loop
    
```

The name dependences force the loop to be mostly scheduled in a serial fashion.

Eliminate the name dependences by register renaming

Output dependence
 Antidependence

36

Renaming in the Unroll Example

```

Loop:  { LD    F0, 0(R1)
        { ADDD  F4, F0, F2
        { SD    0(R1), F4
        { LD    F6, -8(R1)
        { ADDD  F8, F6, F2
        { SD    -8(R1), F8
        { LD    F10, -16(R1)
        { ADDD  F12, F10, F2
        { SD    -16(R1), F12
        { LD    F14, -24(R1)
        { ADDD  F16, F14, F2
        { SD    -24(R1), F16
        SUBI  R1, R1, 32
        BNEZ  R1, Loop
    
```

With registers renamed,
the true dependences
now constrain the
scheduling of the loop.

Each loop body can
be overlapped with the
others.

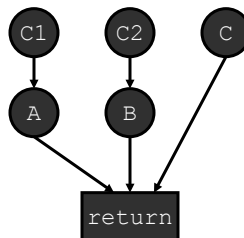
37

Control Dependences

- Determines ordering of instructions with respect to branches
- All instructions are dependent on some branch

```

if (cond)
    A;
if (cond2)
    B;
C;
return;
    
```



A,B,C are completely
independent - C could
depend on A or B and
would need an edge
(A,C) or (B,C)

38

Control Constraints

- If i dependent on branch B , then i can't be moved above B (so its exception isn't controlled by B)
- If i not dependent on branch B , then i can't be moved after B (so its execution dependent on B)

```
Loop:    ...
         BEQZ   R1,exit
         LD     F6,0(R1)
         ADD    F8,F6,F2
         SD     0(R1),F8
         SUBI   R1,R1,8
         BEQZ   R1,exit
         ...
```

39

Control Constraints

- If i dependent on branch B , then i can't be moved above B
- If i not dependent on branch B , then i can't be moved after B .

```
Loop:    ...
         BEQZ   R1,exit
         LD     F6,0(R1)
         ADD    F8,F6,F2
         SD     0(R1),F8
         SUBI   R1,R1,8
         BEQZ   R1,exit
```

Presence of BEQZ prevents moving instructions because of the control dependences

40

Control Constraints

- If i dependent on branch B , then i can't be moved above B .
- If i not dependent on branch B , then i can't be moved after B .

Loop:

```
..*  
LD      F6, 0(R1)  
ADD     F8, F6, F2  
SD      0(R1), F8  
SUBI    R1, R1, 8  
..*  
R1, exit
```

Because trip count is a multiple of 32, the compiler can determine that the intervening branches won't be taken. The control dependences are reduced.

41

Overcoming Control Constraints

- DLX pipeline - control dependences preserved by
 - Executing in order
 - Not executing before branch outcome known
- To improve performance - may violate control dependences
 - As long as program correctness maintained!
 - Preserve exception and data flow behavior
 - Exceptions: "No new exceptions" caused by code motion above a branch.

42

Overcoming Control Constraints

- Exceptions

```
      BEQZ    R2, L1
      LW      R1, 0(R2)
L1:    ...
```


43

Overcoming Control Constraints

- Exceptions

```
      BEQZ    R2, L1
      LW      R1, 0(R2)
L1:    ...
```

Hoist LW above the branch



44

Overcoming Control Constraints

- Exceptions

```
      BEQZ  R2, L1
      LW    R1, 0(R2)
L1:    ...
```



Hoist LW above the branch -
a new exception may result
since BEQZ guards R2

45

Overcoming Control Constraints

- Exceptions

```
      BEQZ  R2, L1
      LW    R1, 0(R2)
L1:    ...
```

- Data flow

```
      ADD   R1, R2, R3
      BEQZ  R4, L0
      SUB   R1, R5, R6
L0:    OR    R7, R1, R8
```

46


Overcoming Control Constraints

- Exceptions

```
        BEQZ   R2, L1
        LW     R1, 0(R2)
L1:     ...
```

- Data flow

```
        ADD    R1, R2, R3
        BEQZ   R4, L0
        SUB    R1, R5, R6
L0:     OR     R7, R1, R8
```



If branch taken, R1 comes from the ADD

47


Overcoming Control Constraints

- Exceptions

```
        BEQZ   R2, L1
        LW     R1, 0(R2)
L1:     ...
```

- Data flow

```
        ADD    R1, R2, R3
        BEQZ   R4, L0
        SUB    R1, R5, R6
L0:     OR     R7, R1, R8
```



If branch not taken, R1 comes from the SUB

48

Overcoming Control Constraints

- Exceptions

```
      BEQZ    R2, L1
      LW     R1, 0(R2)
L1:   ...
```

- Data flow

```
      ADD    R1, R2, R3
      BEQZ   R4, L0
      SUB    R1, R5, R6
L0:   OR     R7, R1, R8
```



Have to preserve the data flow although the dependences on OR are preserved.

- OR data dependent on ADD, SUB but control dependent on BEQZ

49

Overcoming Control Constraints

- Sometimes it's OK

```
      ADD    R1, R2, R3
      BEQZ   R12, L0
      SUB    R4, R5, R6
      ADD    R5, R4, R9
L0:   OR     R7, R8, R9
```

Suppose R4 is dead at L0



R4 is dead here

50

Overcoming Control Constraints

- Sometimes it's OK

```
ADD    R1, R2, R3
BEQZ   R12, L0
SUB    R4, R5, R6
ADD    R5, R4, R9
L0:    OR     R7, R8, R9
```

Suppose R4 is dead after L0 - then we can move the SUB above the branch

51

Overcoming Control Constraints

- Sometimes it's OK

```
ADD    R1, R2, R3
BEQZ   R12, L0
SUB    R4, R5, R6
ADD    R5, R4, R9
L0:    OR     R7, R8, R9
```

Suppose R4 is dead after L0 - then we can move the SUB above the branch

- R4 dead, so data flow not affected and SUB won't cause an exception.
- It may also be OK if we can "fix up" the data flow if we went the "wrong way" - path sensitive optimizations (speculation) try to do this.

52

Loop Carried Dependences

- A parallelizable loop - no cyclic dependences across successive iterations of the loop
- Loop carried dependence - exists across more than one iteration of the loop

```
for (i = 0; i < 100; i++)  
    A[i+1] = A[i] + C[i] + S;
```

- A[i+1] depends on the value of A[i] from the previous loop iteration.

53

Loop Carried Dependences

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

- Can we parallelize this loop?

54

Loop Carried Dependences

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



- Loop carried dependence involving B[*I*].
 - S1 dependent on S2 from previous iteration.
 - S2 not dependent on S1
- Hence, we can parallelize this loop (no cycles)

55

Parallelized Loop

```
A[I] = A[1] + B[1];  
for (I = 1; I <= 99; 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 carried dependence removed - moved to within an iteration rather than between iterations.
- Statements within the iteration must obey the true dependence on B[*I*+1].

56