

09-1 Multicycle Pipeline Operations

09-1

Material may be added to this set.

Material Covered

Section 3.7.

Long-Latency Operations (Topics)

Typical long-latency instructions: floating point

Pipelined v. non-pipelined execution units

Initiation interval and latency

Placement in Chapter-3 DLX pipeline

Timing diagrams

09-2 Long-Latency Instructions (Operations)

09-2

Common Long-Latency Instructions

Fastest (shortest—but still long—latency): Floating-Point Add, Subtract, Conversions

DLX: `addf`, `addd`, `cvti2f` (convert integer to float), `ltd` (compare less-than of doubles), etc.

Intermediate Speed: Multiply

DLX: `multd`, `multf`.

Slowest Speed: Divide, Modulo, Square Root

DLX: `divd`, `divf`.

09-3 Implementation of Long-Latency Instructions

09-3

Implementation balances cost and performance.

Low Cost: Unpipelined, Single Functional Unit, Data Recirculates

Whole functional unit occupied by instruction during computation ...
... so it can execute only one instruction at a time.

Intermediate Cost: Multiple Unpipelined Functional Units

Functional units occupied by instruction during computation ...
... each can execute a different instruction.

Cost a multiple of single-unit cost.

Highest Cost: Pipelined Functional Unit

Functional unit pipelined, at best each stage can hold a different instruction.

Cost disadvantage depends on how unpipelined units implemented.

09-4 Hazards With Long-Latency Instructions in Chapter-3 Pipeline

09-4

Structural Hazards

Functional Unit Structural Hazards

Because an instruction can occupy a functional unit (*e.g.*, DIV) more than one cycle ...
... a following instruction needing that unit may be stalled.

(Occurs when initiation interval greater than one.)

Register Write (MEM Stage) Structural Hazards

Because different units have different latencies ...
... instructions that started at different times can finish at the same time ...
... only one can write results (unless extra register file ports added).

Data Hazards

RAW Hazards

As with integer operations, result not ready in time.

With long-latency operations instructions may wait longer.

WAW Hazards

Occurs when two nearby instructions write same register ...
... and second instruction finishes first.

WAR Hazards

Cannot occur in Chapter-3 pipeline because instructions start in order.

Precise Exceptions

A headache because an instruction can be ready to write ...
... long before a preceding instruction raises an exception.

Handling Functional Unit Structural Hazards

Example, 4-cycle latency unpipelined divide.

Unless FU changed, instructions must be stalled to avoid hazard.

divd	f0, f2, f4	IF	ID	DIV	DIV	DIV	DIV	DIV	WB
divd	f6, f8, f10		IF	ID	----->	DIV	DIV	DIV	DIV

Hazard easily handled:

Units provide a *ready-next-cycle* signal to ID stage.

Instruction stalled if ready-next-cycle for needed unit is 0.

Eliminating Hazards

Provide more than one functional unit.

Example, provide two 4-cycle latency divide units, DVa and DVb.

divd	f0, f2, f4	IF	ID	DVa	DVa	DVa	DVa	DVa	WB
divd	f6, f8, f10		IF	ID	DVb	DVb	DVb	DVb	WB

Pipeline functional unit.

Example, use 5-cycle latency, initiation interval 2, pipelined divide ...
... and live with single stall cycle.

divd	f0, f2, f4	IF	ID	DV0	DV0	DV1	DV1	DV2	DV2	WB
divd	f6, f8, f10		IF	ID	-->	DV0	DV0	DV1	DV1	DV2

Handling Register Write Structural Hazards

Example (stall to avoid hazard in cycle 8)

!Cycle		0	1	2	3	4	5	6	7	8	9
multd	f0, f2, f4	IF	ID	M0	M1	M2	M3	M4	M5	WB	
addi	r1, r1, #1		IF	ID	EX	MEM	WB				
addd	f6, f8, f10			IF	ID	-->	A0	A1	A2	A3	WB

Handling Register Write Structural Hazards

Method 1: Delay instruction in ID. (Used above.)

Include a shift register called a *reservation register*.

Each cycle the reservation register is shifted.

A 1 indicates a “reservation” to enter WB.

Bit position indicates time ...

... with the LSB indicating two cycles later ...

... the next bit indicating three cycles later ...

... and so on.

The ID stage controller, based on the opcode of the instruction ...

... knows the number of cycles before WB will be entered.

It checks the corresponding reservation register bit ...

... if it's 1 then IF and ID are stalled ...

... if it's 0 then the bit is set to 1 and the instruction proceeds.

If such a stall occurs the reservation register is still shifted ...

... and so a 0 will eventually move into the bit position.

Method 2: Delay instructions ready to enter WB.

Each functional unit provides a signal ...

... indicating when it has an instruction ready to enter WB.

One of those signals is chosen (using some method) ...

... the corresponding instruction moves to WB ...

... while the others are stalled.

Comparison of Method 1 and 2

Method 1 is easier to implement ...

... since logic remains in one stage.

In contrast, logic for method 2 would span several stages ...

... since stages back to IF might need to be stalled ...

... and so critical paths would be long.

Method 2 is more flexible ...

... since priority could be given to longer-latency instructions.

Handling RAW Hazards

The interlock mechanism for RAW hazards ...

... must keep track of registers with pending writes ...

... and use this information to stall instructions.

Consider, `add f1, f2, f3`.

Check if any uncompleted preceding instructions write `f2` or `f3`.

If so, stall until register(s) written or can be bypassed to adder.

Possible RAW Interlock Implementations.

Brute Force: Check all following stages

As done for integer operations, check following stages ...
... for pending write to register.

Each stage of every pipelined unit must be checked.

Too expensive.

Register file includes *ready bit* for each register.

Ready bit normally 1, indicating no pending writes (so value valid).

When instruction issued, bit set to 0 ...
... when instruction completes and result written, set back to 1.

Instruction stalls if either operand's ready bit is 0 ...
... *and* cannot be bypassed.

WAW Hazards

Example with 3-stage pipelined multiply and one-stage add, no MEM.

mul	f0, f1, f2	IF	ID	M0	M1	M2	WB	
add	f0, f3, f4		IF	ID	A0	WB		! Incorrect execution!!

Handling WAW Hazards

The interlock mechanism for RAW hazards handles WAW hazards in which there is an intervening read.

Example with 3-stage pipelined multiply and one-stage add, no MEM.

mul	f0, f1, f2	IF	ID	M0	M1	M2	WB	
sub	f5, f0, f6		IF	ID	----->	A0	WB	
add	f0, f3, f4			IF	----->	ID	A0	WB ! No problem.

If there is no intervening write the earlier instruction is squashed.

mul	f0, f1, f2	IF	ID	M0x				
add	f0, f3, f4		IF	ID	A0	WB		

WAR Hazards

Possible when register read delayed.

Can't happen in Chapter-3 DLX because instructions

- (1) read registers in ID
- (2) pass through ID in program order
- (3) and produce results only after leaving ID.

Consider:

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11
mult	f0, f1, f2	IF	ID	M0	M1	M2	M3	M4	M5	M6	M7	WB
add	f1, f3, f4		IF	ID	A0	A1	A2	A3	WB			

There *would* be a WAR hazard if **addf** wrote **f1** before **multf** read it.
That can't happen since **multf** would leave ID (with **f1**) as **addf** just enters ID.

CPI and Multicycle Operations

CPI more sensitive to dependencies between instructions.

CPI Loop Example

Consider:

LOOP:												
ld	f0, 0(r1)											
addi	r1, r1, #8											
gtd	f0, f2											
bfpt	LOOP											
addi	r2, r2, #1											
j	LOOP											
xor	r3, r4, r5											

Note dependency between **gtd** and **bfpt**.

What is the CPI during the execution of this loop?

When branch not taken:

LOOP:													
Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12 13
ld f0, 0(r1)	IF	ID	EX	MEM	WB						IF	ID	EX
addi r1, r1, #8		IF	ID	EX	MEM	WB						IF	ID
gtd f0, f2			IF	ID	A0	A1	A2	A3	WB				IF
bfpt LOOP				IF	ID	----->				EX	MEM	WB	
addi r2, r2, #1					IF	----->				ID	EX	MEM	WB
j LOOP										IF	ID	EX	MEM WB
xor r3, r4, r5											IF		

Note: Second iteration will execute exactly as first.

Therefore, can base iteration time on corresponding points in consecutive iterations.

By inspection of diagram, iteration time: 11 cycles. Instructions: 6.

For a large number of iterations. CPI: $\frac{11}{6} = 1.8333$.

When branch taken.

LOOP:													
Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12 13
ld f0, 0(r1)	IF	ID	EX	MEM	WB					IF	ID	EX	MEM WB
addi r1, r1, #8		IF	ID	EX	MEM	WB					IF	ID	EX MEM
gtd f0, f2			IF	ID	A0	A1	A2	A3	WB			IF	ID EX
bfpt LOOP				IF	ID	----->				EX	MEM	WB	IF ID
addi r2, r2, #1					IF	----->				x			IF
j LOOP													
xor r3, r4, r5													

Note: Second iteration will execute exactly as first.

Iteration time: 9 cycles. Instructions: 4.

For a large number of iterations: CPI is $\frac{9}{4} = 2.25$.

Precise Exceptions

Problem is registers written out of order ...
... so some registers must be *unwritten* ...
... so that when handler starts ...
... it must *seem* as though ...
... all instructions before faulting instructions executed ...
... while no instructions after faulting instruction execute.

multf f0, f1, f2	IF	ID	M0	M1	M2	M3	M4	M5	*M6*	WB
addf f1, f3, f4		IF	ID	A0	A1	A2	A3	WB		

To do this either ...
... add lots of stalls so instructions do finish in order ...
... limit those instructions that can raise precise exceptions ...
... or need to *unexecute* instructions.

The first option is fine for debugging, too slow otherwise.

The second option requires lots of hardware.

Stalling to Maintain Precise Exceptions

Method 1: Stall so that instructions complete in order.

multf f0, f1, f2	IF	ID	M0	M1	M2	M3	M4	M5	M6	WB
addf f1, f3, f4		IF	ID	----->				A0	A1	A2 A3 WB

This works, (WB in program order) but reduces performance.

Method 2: Early Detection of Exceptions

FP unit raises exceptions early in computation ...
... if computation passes that point, it will finish without exceptions.

For example, 26-cycle DIV unit may check operands by cycle 3 ...
... if computation reaches cycle 4 there is no possibility of an exception.

Instructions only stall until preceding instruction checked for exceptions.

For example, suppose the FP multiply unit finds exceptions by end of M5.

Then at cycle 8 (below) **addf** can write (no chance of an exception in M6).

Cycle:	0	1	2	3	4	5	6	7	8	9
multf f0,f1,f2	IF	ID	M0	M1	M2	M3	M4	M5	M6	WB
addf f1,f3,f4		IF	ID	->	A0	A1	A2	A3	WB	

Method 3: Have precise and non-precise FP operations.

Let the names of imprecise instructions end in **ip**.

Second **addf** doesn't stall since an exception in **multfip** need not be precise.

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
multf f0,f1,f2	IF	ID	M0	M1	M2	M3	M4	M5	M6	WB					
addf f1,f3,f4		IF	ID	----->	A0	A1	A2	A3	WB						
multfip f5,f6,f7			IF	----->	ID	M0	M1	M2	M3	M4	M5	M6	WB		
addf f6,f8,f9						IF	ID	A0	A1	A2	A3	WB			

Method 4: FP instructions precise when followed by special test instruction.

Call the special instruction **testexc**.

No stalls (and imprecise exceptions) where **testexc** not used.

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
multf	IF	ID	M0	M1	M2	M3	M4	M5	M6	WB								
testexc		IF	ID	----->	EX	MEM	WB											
addf			IF	----->	ID	A0	A1	A2	A3	WB								
multf						IF	ID	M0	M1	M2	M3	M4	M5	M6	WB			
addf							IF	ID	A0	A1	A2	A3	WB					

Unexecuting Instructions

An instruction is unexecuted ...
... by restoring the previous contents of any register it wrote.

Method 1: *History File*

History file holds replaced values.

These are used to undo writes.

Method 2: Writes to register file are buffered.

Register writes (register number and new value) ...
... are first placed in a buffer ...
... possibly out of program order.

Writes from buffer to register file performed in order ...
... waiting for long-latency operations to complete.

Register reads check the buffer first, then the register file.

When an exception occurs ...
... only writes preceding the faulting instruction ...
... are made from the buffer to the register file.

Disadvantage: Checking both buffer and register file is time-consuming.

Method 3: *Future File*

Two register files maintained, *main* and *future*.

Future file written as instruction complete ...
... main file written in program order.

Future file is used for reading registers.

At an exception, ...
... main file updated up to faulting instruction ...
... future file is effectively erased ...
... its contents replaced by main register file before handler starts.

Performance of FP in the Chapter-3 DLX Implementation

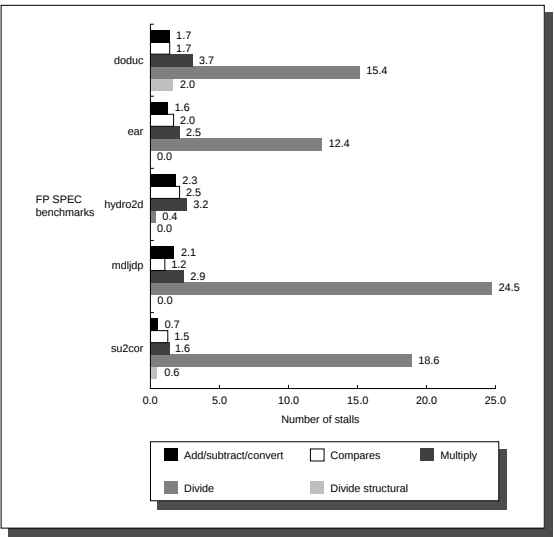
Stalls per FP operation on SPEC 92 FP benchmarks.

Running SPEC 92 benchmarks on DLX compiled using old version of gcc.

Value indicates stall cycles per instruction type.

E.g., running doduc, there are an average of 1.7 stall cycles due to each compare.

Stall cycles are due to RAW hazards except for *divide structural* bars.



Number of stalls determined by:

- latency of functional unit,
- characteristics of program, and
- quality of compiler.

Example:

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
multf	f0,f1,f2	IF	ID	M0	M1	M2	M3	M4	M5	M6	WB				
addf	f3,f0,f4		IF	ID	----->						A0	A1	A2	A3	WB

Here, six stall cycles “charged” to *multf*.

Lower latency (better functional unit) would mean fewer stall cycles.

Example, better scheduling:

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
multf	f0,f1,f2	IF	ID	M0	M1	M2	M3	M4	M5	M6	WB				
gtf	f5,f6,f7		IF	ID	A0	A1	A2	A3	WB						
subd	f8,f10,f12			IF	ID	A0	A1	A2	A3	WB					
addf	f3,f0,f4				IF	ID	----->	A0	A1	A2	A3	WB			

Here **multf** charged with only four cycles because of **gtf** and **subd**.

The existence of such instructions depends on program characteristics.

Discovery and scheduling (arrangement) of such instructions depends on compiler.

Performance of FP in the Chapter-3 DLX Implementation

Running SPEC 92 benchmarks on DLX compiled using old version of gcc.

Value indicates stalls per instruction by cause.

Stalls caused primarily by RAW hazards.

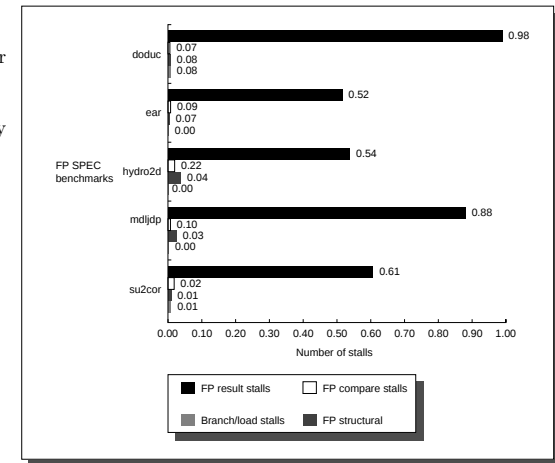


FIGURE 3.49 The stalls occurring for the DLX FP pipeline for the five FP SPEC benchmarks.

Performance of MIPS (now Silicon Graphics) R4000.

Running SPEC 92 benchmarks on R4000.

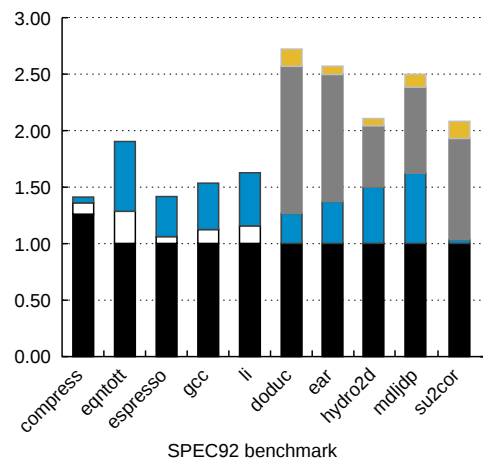
In R4000:

Load latency is two cycles.

Branch delay two cycles.

FP functional units partially pipelined.

Pipeline CPI



Dependencies

Dependency:

A relationship between two instructions ...

... indicating their execution should be in program order.

If there is a dependency between instruction *A* and instruction *B* ...

... and *B* follows *A* in program order ...

... then *B* is said to be *dependent* on *A*.

If *B* is dependent on *A* then *A* should normally execute before *B*.

Dependency Types:

- True, Data, or Flow Dependence (Three different terms used.)
- Name Dependence
- Control Dependence

Data Dependence (a.k.a., *True* and *Flow* Dependence):
 A dependence between two instructions ...
 ... indicating data needed by the second is produced by the first.

Example:

```
add  r1, r2, r3
sub  r4, r1, r5
and  r6, r4, r7
```

The **sub** is dependent on **add** (via **r1**).

The **and** is dependent on **sub** (via **r4**).

The **and** is dependent **add** (via **sub**).

Execution may be incorrect if ...
 ... a program having a data dependence ...
 ... is run on a processor having an uncorrected RAW hazard.

There are two kinds: *antidependence* and *output dependence*.

Antidependence:

A dependence between two instructions ...
 ... indicating a value written by the second ...
 ... that the first instruction reads.

Antidependence Example

```
add  r1, r2, r3
sub  r2, r4, r5
```

sub is antidependent on the **add**.

Execution may be incorrect if ...
 ... a program having an antidependence ...
 ... is run on a processor having an uncorrected WAR hazard.

Output Dependence

A dependence between two instructions ...
 ... indicating that both instructions write the same location ...
 ... (register or memory address).

Output Dependence Example

```
add  r1, r2, r3
sub  r1, r4, r5
```

The **sub** is output dependent on **add**.

Execution may be incorrect if ...
 ... a program having an output dependence ...
 ... is run on a processor having an uncorrected WAW hazard.

Control Dependence

A dependence between a branch instruction and a second instruction ...
 ... indicating that whether the second instruction executes ...
 ... depends on the outcome of the branch.

```
beqz r1, SKIP
add  r2, r3, r4
sub  r5, r6, r7
```

The **add** is control dependent on the **beqz**.

The **sub** is not control dependent on the **beqz**.

Instruction-Level Parallelism
The average number of instructions in a machine-language program ...
... that can be simultaneously started [per cycle] ...
... when execution is only limited by true dependencies.

Note: Text definition is less specific.
Number of instructions started per cycle, IPC, is $\frac{1}{CPI}$.
Provides a bound on performance of an implementation of an ISA.

ILP for SPEC92 programs in MIPS:

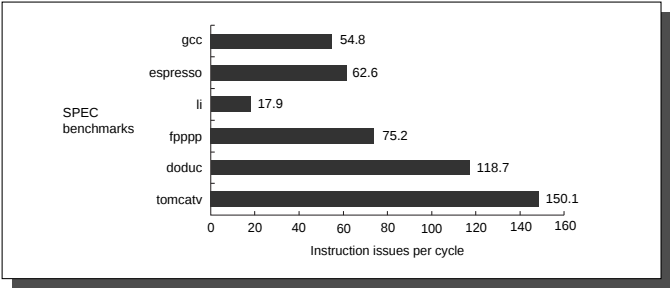


FIGURE 4.38 ILP available in a perfect processor for six of the SPEC benchmarks.

Based on graph it's possible to attain a CPI of $\frac{1}{54.8}$ for gcc ...
... which is much better than 1 for Chapter-3 DLX.
These IPC's are much higher than believed attainable.

Example 1: No control-transfer instructions, no name dependencies.

```
lw  r1, 0(r2)
sub  r4, r1, r5
and  r6, r1, r7
xor  r8, r4, r6
slt  r9, r8, r10
or   r11, r8, r12
addi r13, r8, #1
```

Execution on DLX:

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11
lw r1, 0(r2)	IF	ID	EX	MEM	WB							
sub r4, r1, r5		IF	ID	-->	EX	MEM	WB					
and r6, r1, r7			IF	-->	ID	EX	MEM	WB				
xor r8, r4, r6				IF	ID	EX	MEM	WB				
slt r9, r8, r10					IF	ID	EX	MEM	WB			
or r11, r8, r12						IF	ID	EX	MEM	WB		
addi r13, r8, #1							IF	ID	EX	MEM	WB	

On DLX, execution speed $\frac{7}{8} = 0.875$ inst/cycle.

Execution on ideal machine used for determining ILP:

To find ILP use 0-cycle latencies and true dependencies:

!Cycle:	0	1	2	3	4	5	6	7	8	9	10	11
lw r1, 0(r2)	St											
sub r4, r1, r5		St										
and r6, r1, r7		St										
xor r8, r4, r6			St									
slt r9, r8, r10				St								
or r11, r8, r12				St								
addi r13, r8, #1				St								

ILP is $\frac{7}{4} = 2.75$ inst/cycle, much better.

Note: No stall after load.

Simultaneous execution of instructions at cycle 1 and 3.

Example 1a: No control-transfer instructions, name dependencies.

```
lw  r1, 0(r2)
add r2, r1, r3 ! Data dependence between lw and add
xor r1, r4, r5 ! Anti dependence between add and xor.
add r6, r1, r6 ! Data dependence between xor and add.
```

Execution on the Chapter-3 DLX implementation.

!Cycle:	0	1	2	3	4	5	6	7	8	9	10	11
lw r1, 0(r2)	IF	ID	EX	MEM	WB							
add r2, r1, r3		IF	ID	-->	EX	MEM	WB					
xor r1, r4, r5			IF	-->	ID	EX	MEM	WB				
add r6, r1, r6					IF	ID	EX	MEM	WB			

ILP Analysis.

!Cycle:	0	1
lw r1, 0(r2)	St	
add r2, r1, r3	St	! Wait due to data dependency.
xor r1, r4, r5	St	! No need to wait for name dependency.
add r6, r1, r6	St	

09-43 Example 2: No control-transfer instructions, load/stores.

On Chapter-3 DLX:

!Cycle:	0	1	2	3	4	5	6	7	8	9
add r1, r2, r3	IF	ID	EX	MEM	WB					
sw 0(r10), r1		IF	ID	EX	MEM	WB				
lw r4, 0(r11) ! r10 = r11			IF	ID	EX	MEM	WB			
lw r5, 0(r12) ! r10 != r12				IF	ID	EX	MEM	WB		
sub r6, r4, r7					IF	ID	EX	MEM	WB	
add r8, r5, r9						IF	ID	EX	MEM	WB

ILP Analysis:

!Cycle:	0	1	2	3
add r1, r2, r3	St			
sw 0(r10), r1		St		
lw r4, 0(r11) ! r10 = r11			St	
lw r5, 0(r12) ! r10 != r12	St			
sub r6, r4, r7			St	
add r8, r5, r9		St		

ILP is $\frac{6}{4} = 1.5$.

To achieve this hardware must determine effective-address relationships.

ILP and Basic Blocks

Basic Block:

Consecutive instruction that are always executed consecutively.

Equivalently: consecutive instructions in which ...

... only the first may be a branch target ...

... and only the last may contain a control transfer.

All members of a basic block get executed the same number of times.

```
L1:
  add r1, r2, r3 ! Basic block 1.
L0:
  sub r2, r3, r4 ! Basic block 2.
  and r5, r6, r7 ! Basic block 2.
  bneq r5, TARGET ! Basic block 2.
  xor r6, r7, r8 ! Basic block 3.
TARGET:
  or r9, r10, r11 ! Basic block 4.
L2:
```

Code contains four basic blocks.

Much ILP comes from *ignoring* control dependencies ...
... that is, simultaneously executing instructions in different basic blocks.

Example 3: Control Transfers

On Chapter-3 DLX:

! Cycle	0	1	2	3	4	5	6	7	8	9	10
add r1, r2, r3	IF	ID	EX	MEM	WB						
bneq r4, SKIP		IF	ID	EX	MEM	WB					
add r5, r6, r7			IFx								
SKIP:											
or r8, r9, r10				IF	ID	EX	MEM	WB			
bneq r8 SKIP2					IF	ID	EX	MEM	WB		
addi r1, r1, #5						IFx					
SKIP2:											
xor r11, r11, r12							IF	ID	EX	MEM	WB

ILP Analysis:

! Cycle	0	1		
add r1, r2, r3		St		
bneq r4, SKIP		St		
add r5, r6, r7				
SKIP:				
or r8, r9, r10		St		
bneq r8 SKIP2			St	
addi r1, r1, #5				
SKIP2:				
xor r11, r11, r12		St		

ILP: $\frac{5}{2}$.

Instruction overlap determined by operands only, not branches.

Loop Unrolling:

A code restructuring technique for loops in which ...
... the computations performed by several iterations of the original loop ...
... are performed by one iteration of the unrolled loop.

The unrolled loop performs the same amount of work ...
... but uses fewer instructions and induces fewer stalls.

Loop is said to be unrolled *twice* ...
... if two iterations of original loop performed by one of unrolled loop.

Loop is said to be unrolled *n* times ...
... if *n* iterations of original loop performed by one of unrolled loop.

A loop unrolled once is the same as the original loop.

Suppose loop below runs for 24 iterations.

Execution on DLX:

! Cycle	0	1	2	3	4	5	6	7	8
LOOP:									
lw r1, 0(r2)	IF	ID	EX	MEM	WB			IF	ID
add r3, r3, r1		IF	ID	-->	EX	MEM	WB		IF
addi r2, r2, #4			IF	-->	ID	EX	MEM	WB	
sub r5, r4, r2					IF	ID	EX	MEM	WB
bneq r5, LOOP						IF	ID	EX	MEM
and								IFx	

Execution on DLX. $\frac{7}{5} = 1.5$ CPI ...
... execution time $24 \times 7 = 168$ cycles.

Unrolled twice:

! Cycle	0	1	2	3	4	5	6	7	8	9	10
LOOP:											
lw r1, 0(r2)	IF	ID	EX	MEM	WB				IF	ID	EX
lw r10, 4(r2)		IF	ID	EX	MEM	WB				IF	ID
addi r2, r2, #8			IF	ID	EX	MEM	WB				IF
add r3, r3, r1				IF	ID	EX	MEM	WB			
add r3, r3, r10					IF	ID	EX	MEM	WB		
sub r5, r4, r2						IF	ID	EX	MEM	WB	
bneq r5, LOOP							IF	ID	EX	MEM	WB
and								IFx			

Instruction execution time: $\frac{8}{7} = 1.14$ CPI...
... execution time $12 \times 8 = 96$ cycles.

Double benefit: ...
... faster execution per instruction and ...
... fewer instructions.

ILP Analysis (for comparison): $\frac{1}{5} = 0.2$ CPI.

! Cycle	0	1	2	3	4	5	6	7
LOOP:								
lw r1, 0(r2)	St1	St2						
add r3, r3, r1		St1	St2					
addi r2, r2, #4	St1	St2						
sub r5, r4, r2		St1	St2					
bneq r5, LOOP			St1	St2				
and								

Total execution time: 24 cycles.

Unrolled loop still far from ideal.

Scheduled:

! Cycle	0	1	2	3	4	5	6	7
LOOP:								
lw r1, 0(r2)	IF	ID	EX	MEM	WB		IF	
addi r2, r2, #4		IF	ID	EX	MEM	WB		
add r3, r3, r1			IF	ID	EX	MEM	WB	
sub r5, r4, r2				IF	ID	EX	MEM	WB
bneq r5, LOOP					IF	ID	EX	MEM
and						IFx		

Scheduled: $\frac{6}{5} = 1.2$ CPI ...
... execution time $24 \times 6 = 144$ cycles.

Not as good as unrolled loop, 96 cycles.

How Unrolling Improves Performance

Suppose original loop had 24 iterations and unrolled twice.

Unrolled loop runs for 12 iterations.

Twelve, instead of 24 end-of-loop branches ...
... eliminates 12 branch condition test instructions, 12 branch instructions, ...
... and 12 bubbles inserted after branch.

If indexed addressing allowed (*e.g.*, `lw r10, 4(r2)`) ...
... eliminates 12 address increment instructions.

With more instructions per iterations, its easier to eliminate RAW hazard stalls by scheduling.