Chapter 4a: Basic Pipelining

Principles of Pipelining

The DLX Pipeline

Structural Hazards

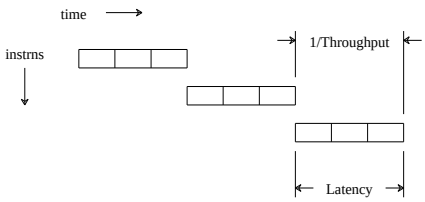
Data Hazards

Control Hazards

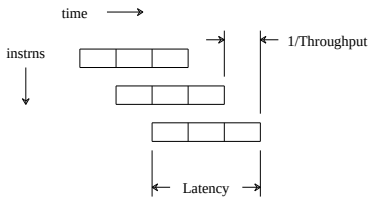
Interrupts

Key Idea: Pipelining

Unpipelined:



Pipelined:



Ideally,

$$Time_{pipeline} = \frac{Time_{sequential}}{Pipeline\ Depth}$$

Principles of Pipelining: An Ideal Example

Let T be the time to execute an instruction.

The instruction execution requires n stages $s_1, s_2, ..., s_n$, taking time $t_1, t_2, ..., t_n$.

Let $T = \Sigma t_i$

Without pipelining

$$\text{Throughput} = \frac{1}{T} = \frac{1}{\Sigma t_i}$$

$$\text{Latency} = T = \frac{1}{\text{Throughput}}$$

With an n-stage pipeline

$$\text{Throughput} = \frac{1}{\max t_i} \leq \frac{n}{T}$$

$$\text{Latency} = n \times \max t_i \geq T = \frac{n}{\text{Throughput}}$$

$$\text{Speedup} \leq \frac{\Sigma t_i}{\max t_i} \leq n$$

Principles of Pipelining, cont.

Let $\Delta > 0$ be extra delay per stage

Latches

Clock/data skew [Kunkel Smith 1986]

Δ limits the useful depth of a pipeline.

With an n-stage pipeline

$$\text{Throughput} = \frac{1}{\Delta + \max t_i} < \frac{n}{T}$$

$$\text{Latency} = n \times (\Delta + \max t_i) \geq n\Delta + T$$

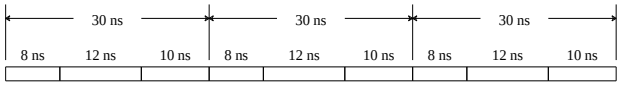
$$\text{Speedup} = \frac{\Sigma t_i}{\Delta + \max t_i} < n$$

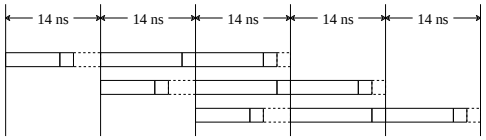
Pipelined machines sometimes have faster clocks.

Real Pipelines do not achieve this Ideal.

Pipelining Example

Let $t_{1,2,3}$ = 8, 12, 10 ns and $\Delta = 2$ ns





$$\text{Throughput} = \frac{1}{2\text{ ns} + 12\text{ ns}} = \frac{1}{14\text{ ns}}$$

$$\text{Latency} = 3 \times (2\text{ ns} + 12\text{ ns}) = 42\text{ ns}$$

$$\text{Speedup} = \frac{30\text{ ns}}{2\text{ ns} + 12\text{ ns}} = 2.14 < 3$$

Principles of Pipelining, cont.

Speedup: Cycle Time and CPI

$$\text{Pipeline Speedup} = \frac{\textbf{\textit{Time}}_{\textbf{\textit{sequential}}}}{\textbf{\textit{Time}}_{\textbf{\textit{pipeline}}}}$$

$$= \frac{\textbf{\textit{CPI}}_{\textbf{\textit{sequential}}}}{\textbf{\textit{CPI}}_{\textbf{\textit{pipeline}}}} \times \frac{\textbf{\textit{CycleTime}}_{\textbf{\textit{sequential}}}}{\textbf{\textit{CycleTime}}_{\textbf{\textit{pipeline}}}}$$

If we ignore CycleTime differences:

$$\textbf{\textit{CPI}}_{\textbf{\textit{ideal}}} = \frac{\textbf{\textit{CPI}}_{\textbf{\textit{sequential}}}}{\text{Pipeline } \textbf{\textit{Depth}}}$$

$$\text{Pipeline Speedup} = \frac{\textbf{\textit{CPI}}_{\textbf{\textit{ideal}}} \times \text{Pipeline Depth}}{\textbf{\textit{CPI}}_{\textbf{\textit{ideal}}} + \text{Pipeline stall cycles}}$$

The DLX Pipeline

DLX Execution Stages

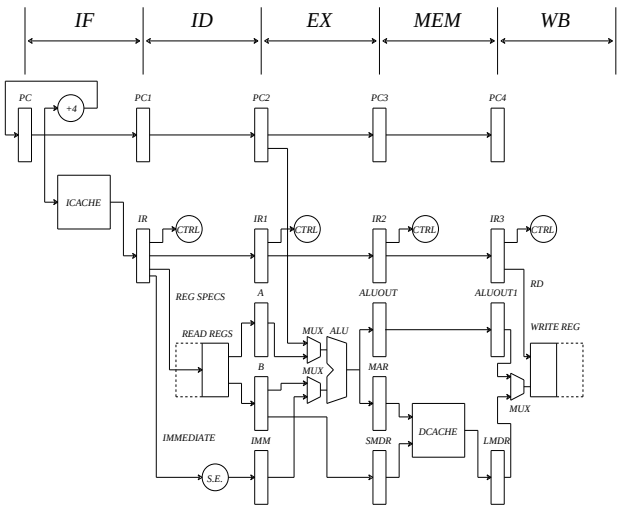
- IF -- Instruction Fetch
- ID -- Instruction Decode
- EX -- Execution and Effective Address
- MEM -- Memory Access
- WB -- Writeback

A 5-stage Pipeline

```
      1   2   3   4   5   6   7   8   9
i    IF  ID  EX  MEM WB
i+1      IF  ID  EX  MEM WB
i+2          IF  ID  EX  MEM WB
i+3              IF  ID  EX  MEM WB
i+4                  IF  ID  EX  MEM WB
```

Pipelining really isn't this simple.

A Naive DLX Pipeline Implementation



Ignores most *hazards*

- Structural (no I/D misses)
- Data (no register bypass)
- Control (no PC-changing instructions)

Hazards

Hazards

Conditions that prevent the next instruction from executing.

Structural Hazards

When two *different* instructions want to use the *same* hardware resource in the *same* cycle (Resource conflict).

Data Hazards

When two *different* instructions use the *same* location, it must *appear* as if instructions execute one at a time.

Control Hazards

When an instruction affects *which* instructions are executed *next* -- branches, jumps, calls

Handling Hazards

(Pipeline) Interlock Logic

Detects hazard and takes appropriate action

Simplest Solution: Stall

Increases CPI

Decreases Performance

Not that simple

Some instructions stall

Others must proceed

Partial stalls can cause problems...

Especially with interrupts.

Other solutions are harder, but have better performance.

Structural Hazards

When two *different* instructions want to use the *same* hardware resource in the *same* cycle

Stall (cause bubble)

+ Low cost, simple

- Increases CPI

Use for rare events

In DLX: double cache misses

Duplicate Resource

+ Good performance

- Increases cost (and maybe cycle time for interconnect)

Use for cheap resources

In DLX: ALU and PC adder

Structural Hazards, cont.

Pipeline Resource

+ Good performance

- Often complex to do

Use when simple to do

In DLX: write & read registers every cycle

Structural hazards are avoided if each instruction uses a resource

At most once

Always in the same pipeline stage

For one cycle

DLX designed with these rules in mind

Pipeline depends upon instruction set

DLX pipeline would not work for the VAX

Structural Hazard Example

Load and stores (MEM) use the same memory port as instruction fetches (IF)

30% of all instructions are loads and stores

Assume CPI_{old} is 1.5

```
      1   2   3   4   5   6   7   8   9
i    IF  ID  EX  MEM WB  <-- a load
i+1      IF  ID  EX  MEM WB
i+2          IF  ID  EX  MEM WB
i+3              **  IF  ID  EX  MEM WB
i+4                      IF  ID  EX  MEM
```

How much faster could a new machine with two memory ports be?

$$CPI_{new} = 1.5 - 1 \times 30\% = 1.2$$

$$Speedup = \frac{CPI_{old}}{CPI_{new}} = \frac{1.5}{1.2} = 1.25$$

Resource Conflicts

Make sure overlapping instructions do not use the same resources at the same time.

Need a separate incrementer for PC, since ALU is used every cycle by the EX stage.

Resource charts

	Resource		
Stage	PC Unit	Memory	Data path
IF	PC ← PC+4	IR ← Mem[PC]	
ID	PC1 ← PC+4	IR1 ← IR	A ← Rs1 B ← Rs2
EX			
MEM			
WB			

Data Hazards

When two *different* instructions use the *same* location, it must *appear* as if instructions execute one at a time.

```
i     ADD r1,r2,--
i+1   SUB r2,--,r1
i+2   OR  r1,--,--
```

Read-After-Write (RAW, data-dependence)

A true dependence

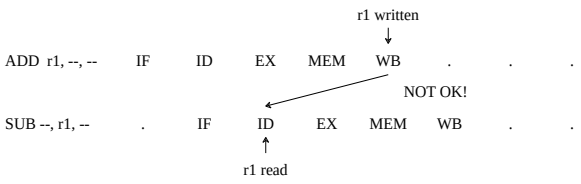
MOST IMPORTANT

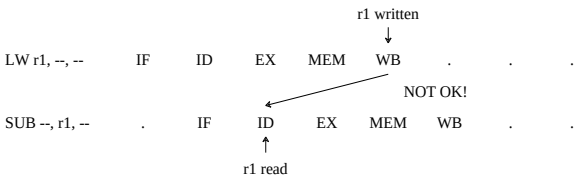
Write-After-Read (WAR, anti-dependence)

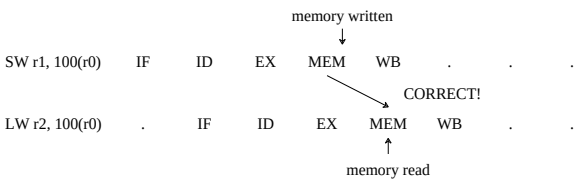
Write-After-Write (WAW, output-dependence)

NOT: Read-After-Read (RAR)

Example RAW (Read-After-Write) Hazards





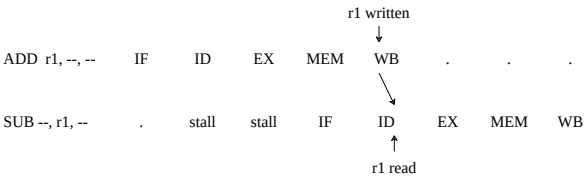


(Unless LW instrn is at address 100(r0))

RAW Solutions

Solutions must first detect RAW, and then ...

Stall



(Assumes registers written then read each cycle)

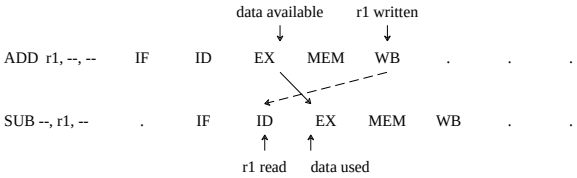
+ Low cost, simple

- Increases CPI (plus 2 per stall in DLX)

Use for rare events

RAW Solutions

Bypass/Forward/Short-Circuit



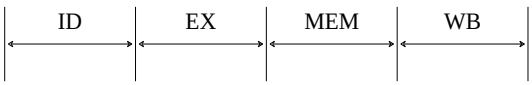
Use data before it is in register

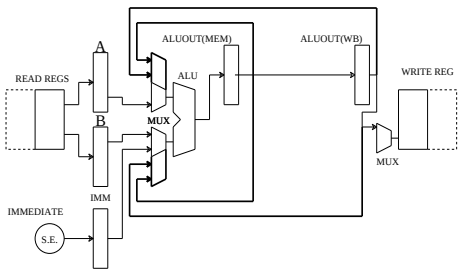
+ Reduces (avoids) stalls

- More complex

Critical for common RAW hazards

Bypass, cont.





Additional hardware

Muxes supply correct result to ALU

Additional control

Interlock logic must control muxes

(What about loads?)

Bypass, cont.

Interlock logic

Detect hazard

Bypass correct result to ALU

Hazard detection requires extra hardware

Instruction latches for each stage

Comparators to detect the hazards

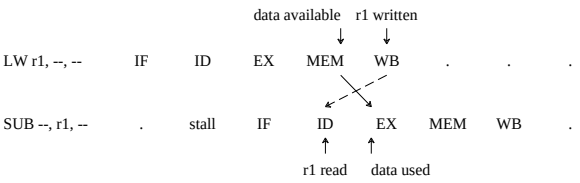
MUX Control (B side):

If instrn(EX) uses immediate then select IMM
Else if rs2(EX)==rd(MEM) then ALUOUT(MEM)
Else if rs2(EX)==rd(WB) then select ALUOUT(WB)
Else select (B)

(Assumes ALU used in two previous cycles)

RAW Solutions, cont.

Hybrid solution sometimes required:



Thus, DLX has one-cycle bubble if result of a load is used by the next instruction

Pipeline scheduling at compile-time

- Moves instructions to eliminate stalls
- Can eliminate over half the stalls

Pipeline Scheduling Example

Before:

```
a = b + c;      LW  Rb, b
                LW  Rc, c
                              <-- stall
                ADD Ra, Rb, Rc
                SW  a, RA

d = e - f;      LW  Re, e
                LW  Rf, f
                              <-- stall
                SUB Rd, Re, Rf
                SW  d, Rd
```

After:

```
a = b + c;      LW  Rb, b
                LW  Rc, c
                LW  Re, e
                ADD Ra, Rb, Rc

d = e - f;      LW  Rf, f
                SW  a, Ra
                SUB Rd, Re, Rf
                SW  d, Rd
```


Delayed Load

Disallow use of load value by next instruction

- Compiler inserts NOP in *delay slot*

Pipeline scheduler tries to reduce the number of NOPs.

Unscheduled	Scheduled
LW Rb, b	LW Rb, b
NOP	LW Rc, c
LW Rc, c	LW Re, e
NOP	ADD Ra, Rb, Rc
ADD Ra, Rb, Rc	LW Rf, f
SW a, RA	SW a, Ra
LW Re, e	SUB Rd, Re, Rf
NOP	SW d, Rd
LW Rf, f	
NOP	
SUB Rd, Re, Rf	
SW d, Rd	
42-65% NOPs	15-30% NOPs

Other Data Hazards

```
i     ADD r1,r2,--
i+1   SUB r2,--,r1
i+2   OR  r1,--,--
```

Write-After-Read (WAR, anti-dependence)

- Not in DLX: read early / write late
- Consider late read then early write:

```
i    MULT --, (r2), r1   /* RX mult */
i+1  LW   --, (r1)+  /* autoincrement */
```

Write-After-Write (WAW, output-dependence)

- Not in DLX: write are in order
- Consider: slow then fast operation

```
i    DIVF  fr1, --, --    /* slow */
i+1  BEQZ  --, --
i+2  ADDF  fr1, --, --    /* fast */
```

Control Hazards

When an instruction affects *which* instructions are executed *next* -- branches, jumps, calls

```
i    BEQZ r1, #8
i+1  SUB  --,--,--
     ...
i+8  OR   --,--,--
i+9  ADD  --,--,--


     1   2   3   4   5   6   7   8   9
i   IF  ID  EX  MEM WB
i+1     **  **  **
i+8                 IF  ID  EX  MEM WB
i+9                     IF  ID  EX  MEM
```

But don't know its a branch instruction until ID.

```
     1   2   3   4   5   6   7   8   9
i   IF  ID  EX  MEM WB
i+1     IF  (aborted)
i+8         **  **  IF  ID  EX  MEM WB
i+9                     IF  ID  EX  MEM
```


Control Hazards

Handling control hazards is very important

VAX Example:

- Emer and Clark report 39% of instructions change the PC.
- Naive solution adds approximately 5 cycles (8800 pipeline).
- Therefore, control hazards add almost 2 cycles to the VAX CPI.
- Approximately 20% increase in CPI.

DLX example:

- H+P report 13% branch instructions.
- Naive solution cost 3 cycles per branch
- Implies 0.39 added to CPI
- Approximately 30% increase.

Control hazards are most important in machines with small CPIs.

Handling Control Hazards

Move up ''control point''

- Find out whether branch is taken earlier
- Compute target address faster

For example, in ID stage

- BTA <-- PC + immediate
- if (Rs1 op 0) PC <-- BTA

```
     1   2   3   4   5   6   7   8
i   IF  ID  EX  MEM WB
i+1     IF  (aborted)
i+8         IF  ID  EX  MEM WB
i+9             IF  ID  EX  MEM WB
```

Implies only one-cycle bubble, BUT:

- Special PC adder required
- *IS CYCLE TIME LONGER?*

ISA and Control Hazards

Comparisons in ID stage

- Must be FAST!
- Can't afford subtract.
- Compares with 0 are simple.
  - GT, LT test sign-bit
  - EQ, NE must 'OR' all 32 bits

More general comparisons need ALU

- DLX uses conditional sets

Handling Control Hazards, cont.

Branch Prediction

- Guess the direction of the branch
- Minimize penalty when right
- May increase penalty when wrong

Techniques

- Static -- At compile time
- Dynamic -- At run time

Static Techniques

- Predict Not-Taken
- Predict Taken
- Delayed Branches

Dynamic Techniques

- Later ...

Handling Control Hazards, cont.

Predict NOT-TAKEN Always

- Not-Taken:

```
     1   2   3   4   5   6   7   8
i   IF  ID  EX  MEM WB
i+1     IF  ID  EX  MEM WB
i+2         IF  ID  EX  MEM WB
i+3             IF  ID  EX  MEM WB
```

- Taken:

```
     1   2   3   4   5   6   7   8
i   IF  ID  EX  MEM WB
i+1     IF  (aborted)
i+8         IF  ID  EX  MEM WB
i+9             IF  ID  EX  MEM WB
```

Don't change machine state until branch outcome is known.

- DLX: State always changes late (WB)
- VAX: Auto-increment addressing???

Handling Control Hazards, cont.

Predict TAKEN Always

```
      1   2   3   4   5   6   7   8
i   IF  ID  EX  MEM WB
i+8   'IF' ID  EX  MEM WB
i+9       IF  ID  EX  MEM WB
i+10          IF  ID  EX  MEM WB
```

For DLX, must know what address to fetch at BEFORE branch is decoded

- Easier for some other pipelines
  - e.g., VAX
- Special hardware for fast decode
  - Not practical for DLX

Predict Both

- Hardware & memory bandwidth expensive
- What if branch again?

Handling Control Hazards, cont.

Delayed Branch -- long used in μcode

- Execute next instruction regardless (of whether branch is taken)

```
i    BEQZ r1, #8
i+1  SUB  --,--,--
     ...
i+8  OR   --,--,--
i+9  ADD  --,--,--
```

```
     1   2   3   4   5   6   7   8
i   IF  ID  EX  MEM WB
i+1     IF  ID  EX  MEM WB
i+8         IF  ID  EX  MEM WB
i+9             IF  ID  EX  MEM WB
```

What do we execute in the DELAY SLOT?

Delay Slots

Fill from before branch

- When: Branch independent of instruction
- Helps: Always

Fill from target

- When: OK to execute target
  - May have to duplicate code
- Helps: On taken branch
  - May increase code size

Fill from fall through

- When: OK to execute instruction
- Helps: when not taken

Effectiveness:

- 57% filled; 48% usefully filled

Comparison of Branch Schemes

Branch Statistics (DLX)

- 14% PC-Changing instructions
- 65% of all PC-changing instructions

$$\textit{CPI-penalty} = \%\ \textit{branches} \times (\%\ \textit{taken} \times \textit{taken-penalty} + \%\ \textit{not-taken} \times \textit{not-taken-penalty})$$

Branch Scheme	Taken Penalty	Not-Taken Penalty	CPI Penalty
Naive	3	3	.42
Fast Branch	1	1	.14
Not-Taken	1	0	.09
Taken0	0	1	.05
Taken1	1	1	.14
Delayed Branch	.5	.5	.07

More Delayed Branches

What happens on interrupt while in the delay slot?

- Next instruction is not sequential.

Solution #1: Save multiple PCs.

- Save both current and next PCs.
- Special return sequence
- More complex hardware

Solution #2: Single PC plus

- Branch Delay Bit
- PC points to branch instruction
- Requires S/W Restriction

Interrupts

Some examples

- Power failing
- Arithmetic overflow
- I/O device request
- O.S. call
- Page fault

Interrupts (a.k.a., faults, exceptions, traps) often require

- Surprise jump (to vectored address)
- Linking of return address
- Saving of PSW (including CCs)
- State change (e.g., to kernel mode)

Classifying Interrupts, Part 1 of 2

1a. Synchronous
function of program and memory state
(e.g., arithmetic overflow, page fault)

1b. Asynchronous
external device or hardware malfunction
(printer ready, bus error)

2a. User request
from user program (O.S. call)

2b. Coerced
from operating system or hardware
(page fault, protection violation)

3a. User maskable
can be (temporarily) ignored
(arithmetic overflow, breakpoint)

3b. Nonmaskable
must be handled (page fault, power failing)

Classifying Interrupts Part 2 of 2

4a. Within an instruction
must be dealt with to complete an instruction (page fault)

4b. Between instructions
not part of an instruction
(I/O device request, O.S. call)

5a. Resume
must transparently return to user process
(page fault, I/O device request)

5b. Terminate
give up and die (protection violation, power failing)

Handling Interrupts

Precise Interrupts (Sequential Semantics)

Complete instrns before offending one

Squash (effects of) instrns after

Save PC (& next-PC w/ delayed branches)

Force trap instrn into IF

Must handle simultaneous interrupts

IF -- memory problems (page-fault, misaligned reference, protection violation)
ID -- illegal or privileged instrn
EX -- arithmetic exception
MA -- memory problems
WB -- none

Which interrupt should be handled first?

Interrupts, cont.

Example: Data Page Fault

```
     1   2   3   4   5   6   7   8
i   IF  ID  EX  MEM WB
i+1     IF  ID  EX  MEM WB <-- page fault (MEM)
i+2         IF  ID  EX  MEM WB  <-- squash
i+3             IF  ID  EX  MEM WB  <-- squash
i+4                 IF  ID  EX  MEM WB <-- squash
i+5             trap --> IF  ID  EX  MEM WB
i+6         trap handler --> IF  ID  EX  MEM WB
```

Preceding instruction already complete.

Squash succeeding instructions

Prevent from modifying state

'Trap' instruction jumps to trap handler

Hardware saves PC in IAR

Trap handler must save IAR

Interrupts, cont.

Example: Arithmetic Exception

```
     1   2   3   4   5   6   7   8
i   IF  ID  EX  MEM WB
i+1     IF  ID  EX  MEM WB
i+2         IF  ID  EX  MEM WB  <-- Exception (EX)
i+3             IF  ID  EX  MEM WB  <-- squash
i+4                 IF  ID  EX  MEM WB <-- squash
i+5             trap --> IF  ID  EX  MEM WB
i+6         trap handler --> IF  ID  EX  MEM WB
```

Let preceding instructions complete.

Squash succeeding instructions

Interrupts, cont.

Example: Illegal Opcode

```
     1   2   3   4   5   6   7   8
i   IF  ID  EX  MEM WB
i+1     IF  ID  EX  MEM WB
i+2         IF  ID  EX  MEM WB
i+3             IF  ID  EX  MEM WB  <-- ill. op (ID)
i+4                 IF  ID  EX  MEM WB <-- squash
i+5             trap --> IF  ID  EX  MEM WB
i+6         trap handler --> IF  ID  EX  MEM WB
```

Let preceding instructions complete.

Squash succeeding instruction

Interrupts, cont.

Out-of-order Interrupts

Which page fault should we take?

```
     1   2   3   4   5   6   7   8
i   IF  ID  EX  MEM WB  <-- page fault (MEM)
i+1     IF  ID  EX  MEM WB  <-- page fault (IF)
i+2         IF  ID  EX  MEM WB
i+3             IF  ID  EX  MEM WB
```

Post Interrupts

Check interrupt bit on entering WB

Precise interrupts

Longer latency

Handle Immediately

Not fully precise

Interrupts may occur in different order than on a sequential machine.

May cause implementation headaches.

Interrupts, cont.

Why can’t we allow an interrupt in WB?

```
     1   2   3   4   5   6   7   8
i   IF  ID  EX  MEM WB
i+1     IF  ID  EX  MEM WB
```

Other complications

Odd bits of state (e.g., CCs)

Early-writes (e.g., auto-increment)

Instruction buffers and prefetching logic

Dynamic scheduling

Out-of-order execution

Interrupts come at random times

The frequent case isn’t everything.

The rare case MUST work correctly.