Multicycle Pipeline Operations

09-1

Multicycle Pipeline Operation:

An operation (usually arithmetic) that takes more than one or two cycles.

```
mul.d f0, f2, f4   IF  ID  M1  M2  M3  M4  M5  M6  WF
```

Life is Simple with a Five-Stage Pipeline

- Every instruction goes through the same five stages in the same order.
- No problems with writeback structural hazards.
- Registers are written in program order.

Five Stages are Feasible So Far Because

- Instructions need only one or two stages to execute.
- One stage: `add`, `xori`, etc.
- Two stages: `lw`, `sh`, etc.

09-2

The End of Innocence

Unfortunately we must now set aside this simplicity and elegance.

Because floating-point operations . . .
. . . can't feasibly be computed in one or two cycles.

Here are our options:

- A simple pipeline with lots of stages and an expensive bypass network.
- A simple pipeline with lots of stages and large integer instruction latencies.
- A complex pipeline with low latency for integer instructions.

Multicycle Pipeline Operations

09-3

Long-Latency Operations (Topics)

Typical long-latency instructions: floating point

Pipelined v. non-pipelined execution units

Initiation interval and latency

Implementation of long-latency instructions.

Timing diagrams

Long-Latency Instructions (Operations)

09-4

Common Long-Latency Instructions

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

MIPS: `add.d`, `sub.d`, `cvt.s.w` (convert integer to float), etc.

Intermediate Speed: Multiply

MIPS: `mul.d`, `mul.s`.

Slowest Speed: Divide, Modulo, Square Root

MIPS: `div.d`, `sqrt.d`.

Implementation of Long-Latency Instructions

09-5

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.

Floating Point in Chapter-3 MIPS Implementation

09-6

Typical Classroom Example Floating Point Functional Units

- FP Add

  Four stages, fully pipelined: Latency 3, Initiation Interval 1.

  Used for FP Add, FP Subtract, FP Comparisons, etc.

- FP Multiply

  Six stages, fully pipelined: Latency 5, Initiation Interval 1.

  Used for FP Multiply.

- FP Divide

  Twenty five stages, unpipelined: Latency 24, Initiation Interval 25.

Floating-Point Pipeline

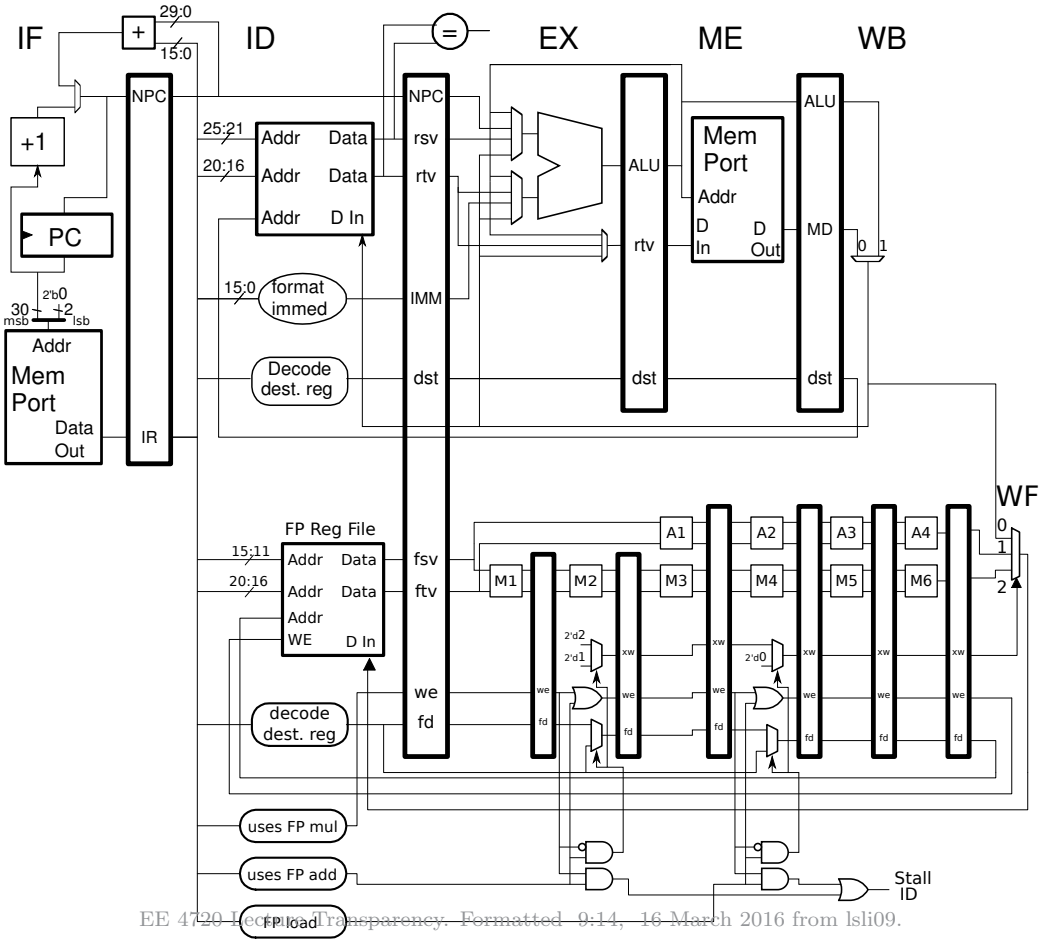
09-7

Example floating unit implementation main features:

Separate register file.

Number of stages vary depending on functional unit.

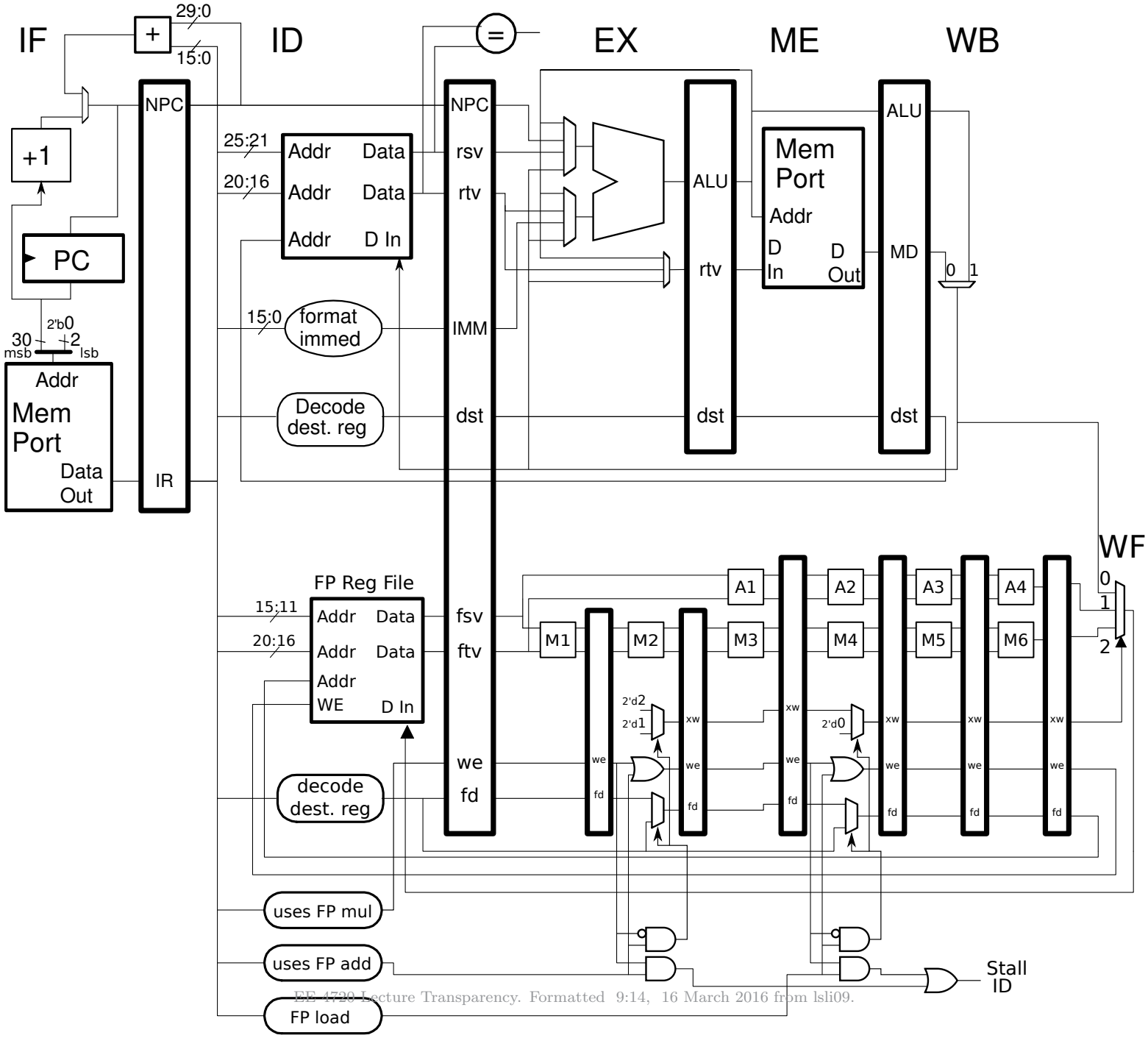
Floating-point writeback separate from integer writeback.



09-7

Floating-Point Pipeline

09-8



09-8

09-8

Floating-Point Pipeline

09-9

Example floating unit implementation notes:

Bypass paths not shown.

Paths to implement FPR → GFP not shown.

Paths for double FP loads and any FP stores (`ldc1`, `sdc1`, etc.) not shown.

Pipeline latches for `we` and `fd` may be part of *reservation register* (covered soon).

Use of register pairs for double operands ignored.

See Spr. 2003 HW 5, Prob. 4, `http://www.ece.lsu.edu/ee4720/2003/hw05sol.pdf`.

The divide functional unit is not shown.

Hazards With Long-Latency Instructions in Chapter-3 Pipeline

09-10

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 (WF-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).

09-11

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.

09-12

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

09-13

Example, 4-cycle latency unpipelined divide.

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

```
div.d f0, f2, f4  IF  ID  DIV DIV DIV DIV DIV WF
div.d f6, f8, f10     IF  ID  --------------> DIV DIV DIV DIV WF
```

Hazard easily handled:

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

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

09-14

Eliminating Hazards

Provide more than one functional unit.

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

```
div.d f0, f2, f4  IF  ID  DVa DVa DVa DVa DVa WF
div.d f6, f8, f10     IF  ID  DVb DVb DVb DVb DVb WF
```

Pipeline functional unit.

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

```
div.d f0, f2, f4  IF  ID  DV0 DV0 DV1 DV1 DV2 DV2 WF
div.d f6, f8, f10     IF  ID  --> DV0 DV0 DV1 DV1 DV2 DV2 WF
```

09-15

Handling Register Write Structural Hazards

Example (stall to avoid hazard in cycle 8)

```
!Cycle              0   1   2   3   4   5   6   7   8   9
mul.d f0, f2, f4    IF  ID  M1  M2  M3  M4  M5  M6  WF
addi  r1, r1, 1         IF  ID  EX  ME  WB
add.d f6, f8, f10           IF  ID  --> A1  A2  A3  A4  WF
```


EE 4720 Lecture Transparency. Formatted 9:14, 16 March 2016 from lsli09.
09-15

Handling Register Write Structural Hazards

09-16

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 WF.

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 WF 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.

09-16

09-17

Method 2: Delay instructions ready to enter WF.

Each functional unit provides a signal ...

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

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

... the corresponding instruction moves to WF ...

... while the others are stalled.

09-18

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.

09-19

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.s 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.

09-20

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.

09-21

WAW Hazards

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

```
mul.s f0, f1, f2    IF  ID  M0  M1  M2  WF
add.s f0, f3, f4        IF  ID  A0  WF      ! Incorrect execution!!
```


09-21

09-22

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 ME.

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

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

```
mul.s f0, f1, f2    IF  ID  M0x
add.s f0, f3, f4        IF  ID  A0  WF
```

09-23

WAR Hazards

Possible when register read delayed.

Can't happen in five-stage MIPS 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
mul.s f0, f1, f2  IF  ID  M0  M1  M2  M3  M4  M5  M6  M7  WF
add.s f1, f3, f4      IF  ID  A0  A1  A2  A3  WF
```

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.

EE 4720 Lecture Transparency. Formatted 9:14, 16 March 2016 from lsli09.
09-23

CPI and Multicycle Operations

09-24

With long-latency ops, dependencies trickier . . .
. . . and structural hazards now present (in our implementations).

Finding CPI for a loop

As before, find a repeating pattern of iterations.

Look out for structural hazards.

Loop Example:

```
LOOP:
        addi $t0, $t0, -1
        mul.s $f2, $f2, $f1 # Note loop-carried dependency through $f2
        bne $t0, $0 LOOP
        lwc1 $f1, 4($t1)
```

Runs on implementation illustrated earlier . . .
. . . with a full set of floating-point bypass paths added.

All bypass paths for integer instructions shown.

What is the CPI during the execution of this loop?

09-25

```
Cycle                  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
LOOP: # First Iteration
 addi $t0, $t0, -1     IF ID EX ME WB
 mul.s $f2, $f2, $f1      IF ID M1 M2 M3 M4 M5 M6 WF
 bne $t0, $0 LOOP            IF ID -> EX ME WB
 lwc1 $f1, 4($t1)               IF -> ID EX ME WF

Cycle                  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
LOOP: # Second Iteration
 addi $t0, $t0, -1                  IF ID EX ME WB
 mul.s $f2, $f2, $f1                   IF ID -> M1 M2 M3 M4 M5 M6 WF
 bne $t0, $0 LOOP                         IF -> ID EX ME WB
 lwc1 $f1, 4($t1)                               IF ID EX ME WF
Cycle                  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

Cycle                  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
LOOP: # Third Iteration
 addi $t0, $t0, -1     IF ID EX ME WB
 mul.s $f2, $f2, $f1      IF ID ----> M1 M2 M3 M4 M5 M6 WF
 bne $t0, $0 LOOP            IF ----> ID EX ME WB
 lwc1 $f1, 4($t1)                  IF ID EX ME WF
Cycle                  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

Note: Each iteration above starts differently.

First, Cycle 0: IF, `addi`; ID, etc. pre-loop instructions.

Second, Cycle 5: IF, `addi`; ID, `lwc1`; EX, `bne`; M3, `mul.s`.

Third, Cycle 10: IF, `addi`; ID, `lwc1`; EX, `bne`; M2, `mul.s`. (Similar, but different.)

09-26

```
Cycle                    10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
LOOP: # Third Iteration
 addi $t0, $t0, -1       IF ID EX ME WB
 mul.s $f2, $f2, $f1        IF ID ----> M1 M2 M3 M4 M5 M6 WF
 bne $t0, $0 LOOP              IF ----> ID EX ME WB
 lwc1 $f1, 4($t1)                       IF ID EX ME WF

Cycle                    10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
LOOP: # Fourth Iteration
 addi $t0, $t0, -1                      IF ID EX ME WB
 mul.s $f2, $f2, $f1                       IF ID ----> M1 M2 M3 M4 M5 M6 WF
 bne $t0, $0 LOOP                             IF ----> ID EX ME WB
 lwc1 $f1, 4($t1)                                      IF ID EX ME WF
Cycle                    10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
```

Third, Cycle 10: IF, `addi`; ID, `lwc1`; EX, `bne`; M2, `mul.s`.

Fourth, Cycle 16: IF, `addi`; ID, `lwc1`; EX, `bne`; M2, `mul.s`.

Since third and fourth start the same way, pattern will repeat.

CPI is $\frac{16-10}{4} = 1.5$.

09-26

09-27

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.

```
mul.s f0, f1, f2  IF  ID  M0  M1  M2  M3  M4  M5 *M6* WF
add.s f1, f3, f4      IF  ID  A0  A1  A2  A3  WF
```

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

09-28

Method 1: Stall so that instructions complete in order.

```
mul.s f0, f1, f2  IF  ID  M0  M1  M2  M3  M4  M5  M6  WF
add.s f1, f3, f4      IF  ID  --------->  A0  A1  A2  A3  WF
```

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

09-29

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
mul.s f0,f1,f2 IF  ID  M0  M1  M2  M3  M4  M5  M6  WF
add.s f1,f3,f4     IF  ID  ->  A0  A1  A2  A3  WF
```


09-29

09-30

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
mul.s   f0,f1,f2    IF  ID  M0  M1  M2  M3  M4  M5  M6  WF
add.s   f1,f3,f4        IF  ID  ---------->  A0  A1  A2  A3  WF
mul.sip f5,f6,f7            IF  ---------->  ID  M0  M1  M2  M3  M4  M5  M6  WF
add.s   f6,f8,f9                             IF  ID  A0  A1  A2  A3  WF
```


09-30

09-31

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
mul.s   IF  ID  M0  M1  M2  M3  M4  M5  M6  WF
testexc     IF  ID  ----------------->  EX  ME  WF
add.s           IF  ----------------->  ID  A0  A1  A2  A3  WF
mul.s                                   IF  ID  M0  M1  M2  M3  M4  M5  M6 WF
add.s                                       IF  ID  A0  A1  A2  A3  WF
```


09-31

09-32

Unexecuting Instructions

An instruction is *unexecuted* . . .

. . . by restoring the previous contents of any register or memory location it wrote.

A system that un-executes can have precise exceptions . . .
. . . for FP operations which execute with few stalls.

Text describes several techniques for unexecution on statically scheduled systems . . .
. . . these not covered in course.

Unexecuting much easier on *dynamically scheduled systems*, covered soon.

Performance of FP in the Chapter-3 DLX Implementation

09-33

Stalls per FP operation on SPEC 92 FP benchmarks.

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

Uses **perfect** cache.

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.

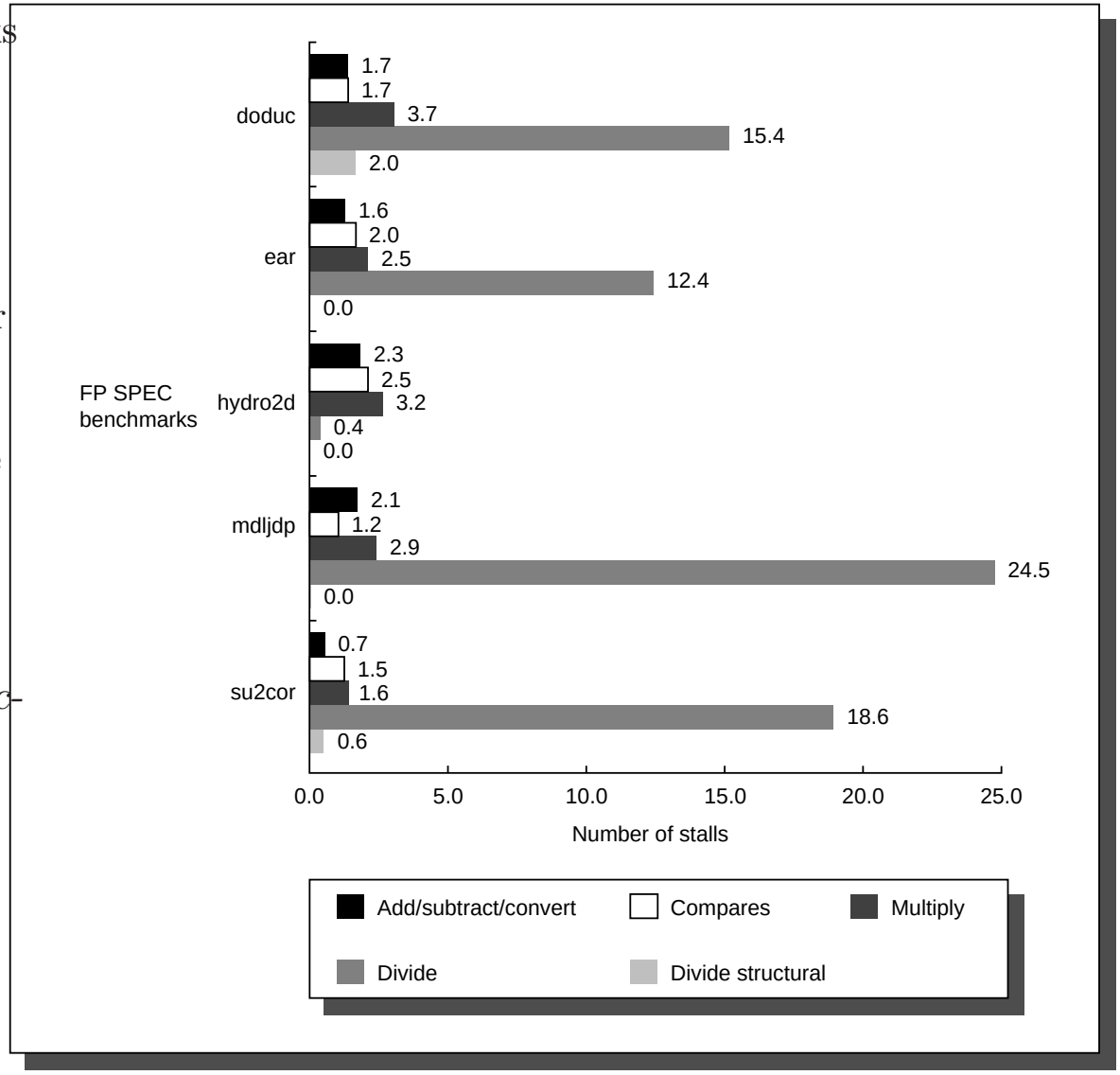


FIGURE 3.48 Stalls per FP operation for each major type of FP operation.

09-33

09-34

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
mul.s   f0,f1,f2    IF  ID  M0  M1  M2  M3  M4  M5  M6  WF
add.s   f3,f0,f4        IF  ID  ---------------------->  A0  A1  A2  A3  WF
```

Here, six stall cycles "charged" to `mul.s`.

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

EE 4720 Lecture Transparency. Formatted 9:14, 16 March 2016 from lsli09.
09-34

09-35

Example, better scheduling:

```
Cycle:              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
mul.s   f0,f1,f2    IF  ID  M0  M1  M2  M3  M4  M5  M6  WF
c.gt.s  f6,f7           IF  ID  A0  A1  A2  A3  WF
sub.d   f8,f10,f12          IF  ID  A0  A1  A2  A3  WF
add.s   f3,f0,f4                IF  ID  ------------>  A0  A1  A2  A3  WF
```

Here `mul.s` 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.

EE 4720 Lecture Transparency. Formatted 9:14, 16 March 2016 from lsli09.
09-35

Performance of FP in the Chapter-3 DLX Implementation

09-36

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

Uses **perfect** cache.

Value indicates stalls per instruction by cause.

Stalls caused primarily by RAW hazards.

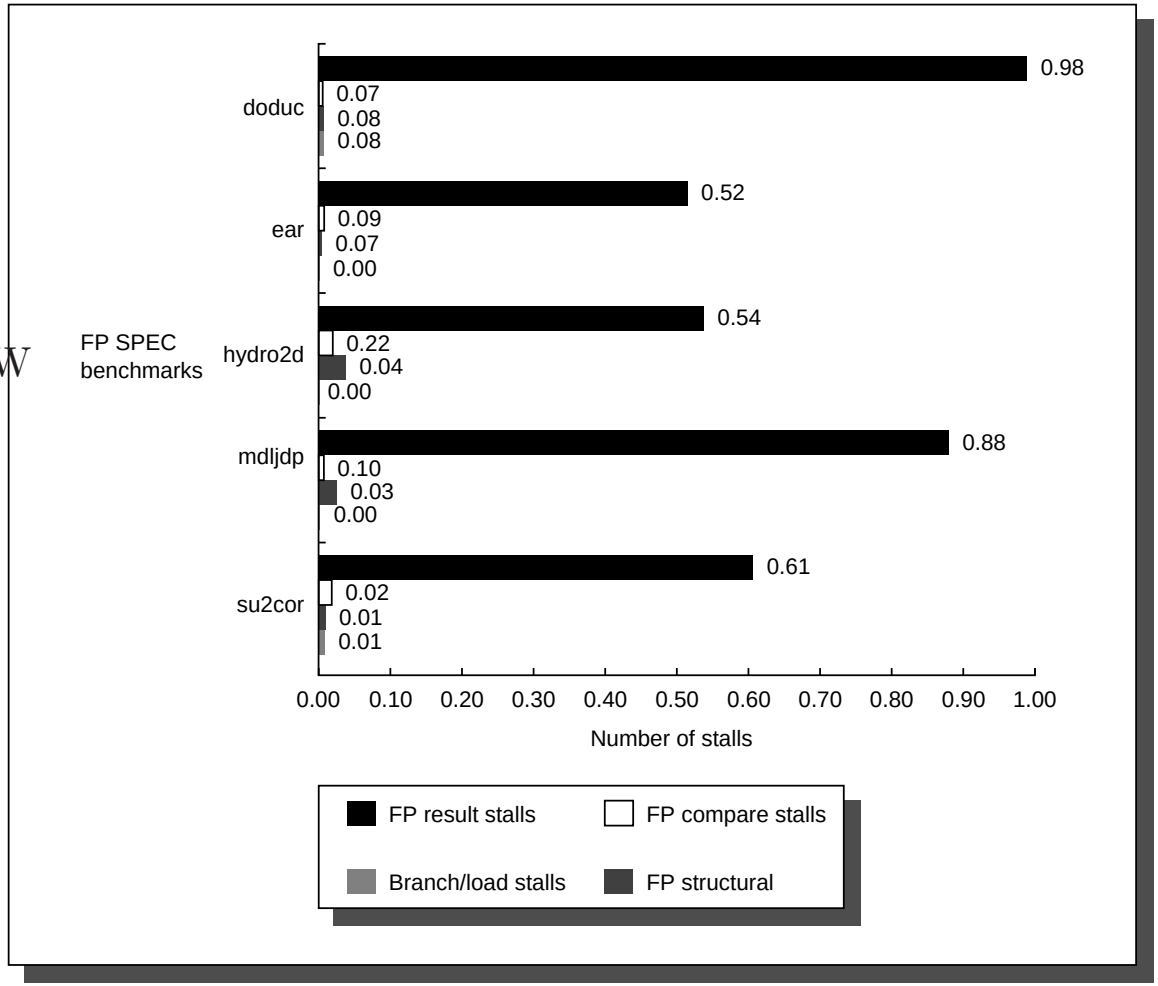


FIGURE 3.49 The stalls occurring for the DLX FP pipeline for the five FP SPEC benchmarks.

09-36

Performance of MIPS R4000.

09-37

Running SPEC 92 benchmarks on R4000.

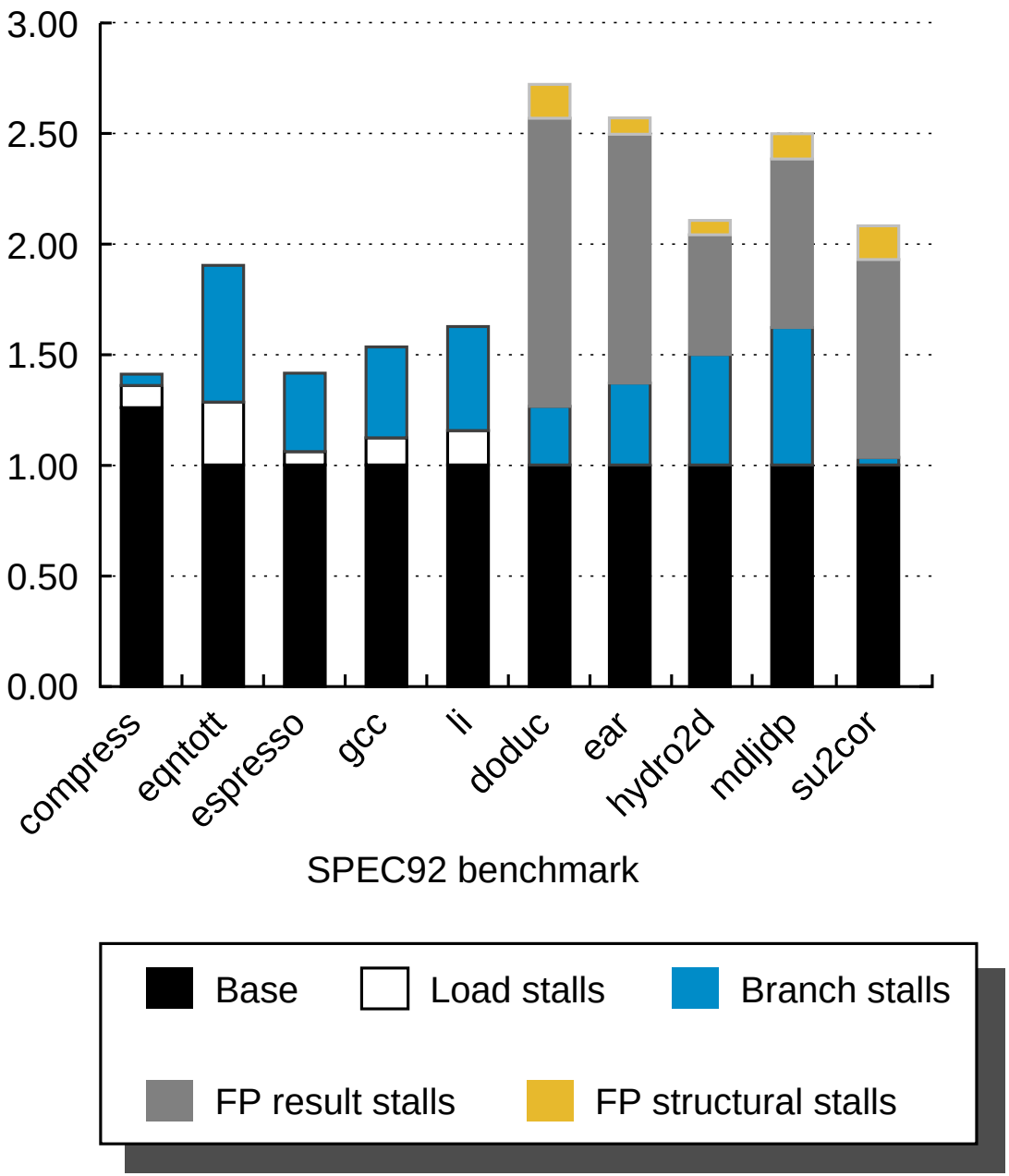
In R4000:

Load latency is two cycles.

Uses **perfect** cache.

Branch penalty two cycles.

FP functional units partially pipelined.



EE 4720 Lecture Transparency. Formatted 9:14, 16 March 2016 from lsli09.

09-37