Multicycle Pipeline Operations

Material in This Set

Typical long-latency instructions: mostly floating point

Pipelined v. non-pipelined execution units

FP hardware for the 5-stage MIPS pipeline.

Long-latency implications for hazards, dependencies, and exceptions.

Pipeline diagrams and computation iteration time and CPI.

Practice Problems

The problems below draw on material covered in this set.

Easier Problems

2021 FE P2 (a). PED for a loop. Find insn thpt. Show mux select signals.

2019 FE P2: Simple PED on pipeline including compare unit.

2016 FE P2b: Simple PED. Hardware for swc1 (one wire)

2015 FE P2b: Simple PED.

2012 FE P2: Show execution of code. Add bypass paths.

Medium Difficulty

2021 FE P2 (b): Re-write (schedule) FP code to avoid stalls.

2020 FE P2 (a,b): Show FP execution on not-fully-pipelined adders.

2020 HW 5 P2: Design control logic for an initiation-interval 2 adder.

2018 FE P3: Integrate comparison unit and FCC reg into FP pipeline.

2014 MT P2: Change pipe so instructions stall in ME to avoid a WF hazard.

2012 FE P1: Use FP multiply stages for integer multiply instructions.

Multicycle Operations and Why they are Different

Multicycle Pipeline Operation:

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

Examples, floating-point multiply and add.

```
# Cycle            0  1  2  3  4  5  6  7  8
mul.d f0, f2, f4   IF ID M1 M2 M3 M4 M5 M6 WF
#                        -----------------
#                           Six Cycles


# Cycle            0  1  2  3  4  5  6  7  8
add.d f6, f8, f10  IF ID A1 A2 A3 A4 WF
#                        -----------
#                        Four Cycles
```

Note: The number of cycles needed depends on . . .

. . . the operation . . .

. . . and of course the implementation.

Life is Simple with a Five-Stage Pipeline! :-)

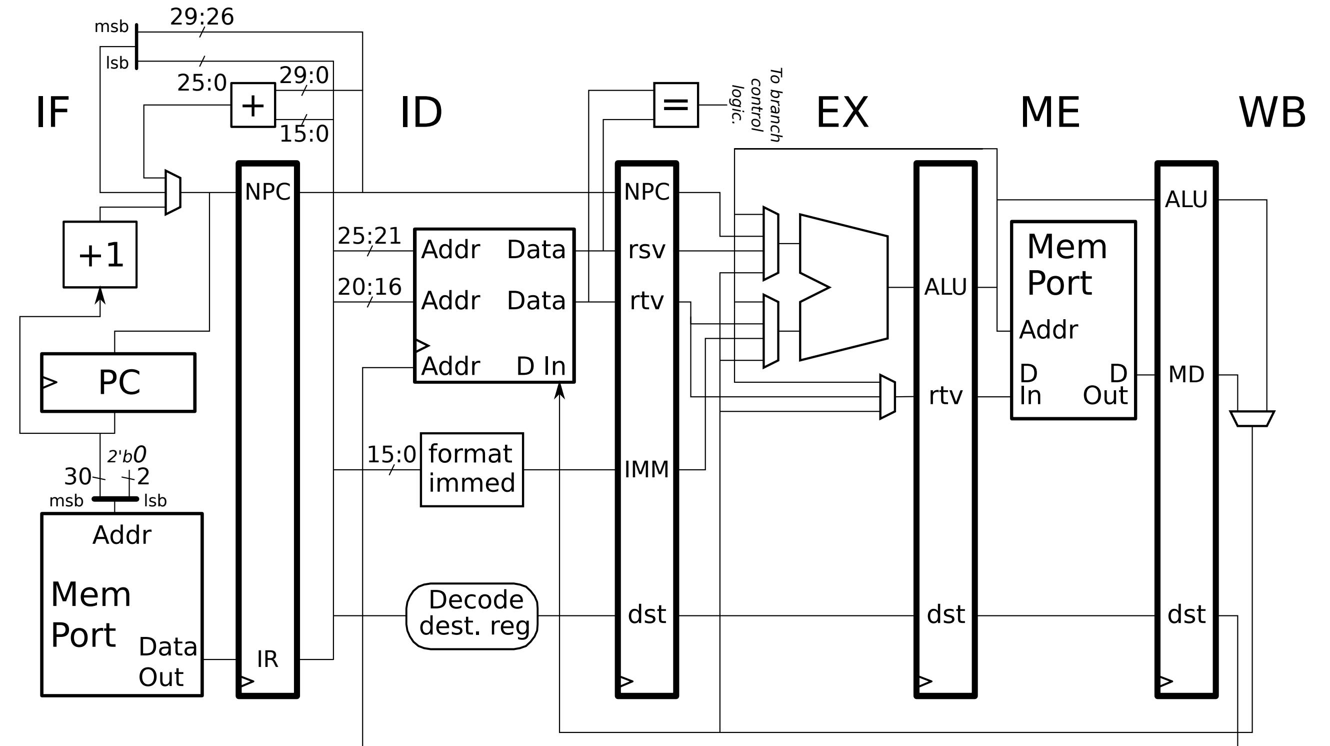
Every instruction goes through. . .

. . . the same five stages. . .

. . . in the same order.

There are no writeback structural hazards.

Registers are written in program order.



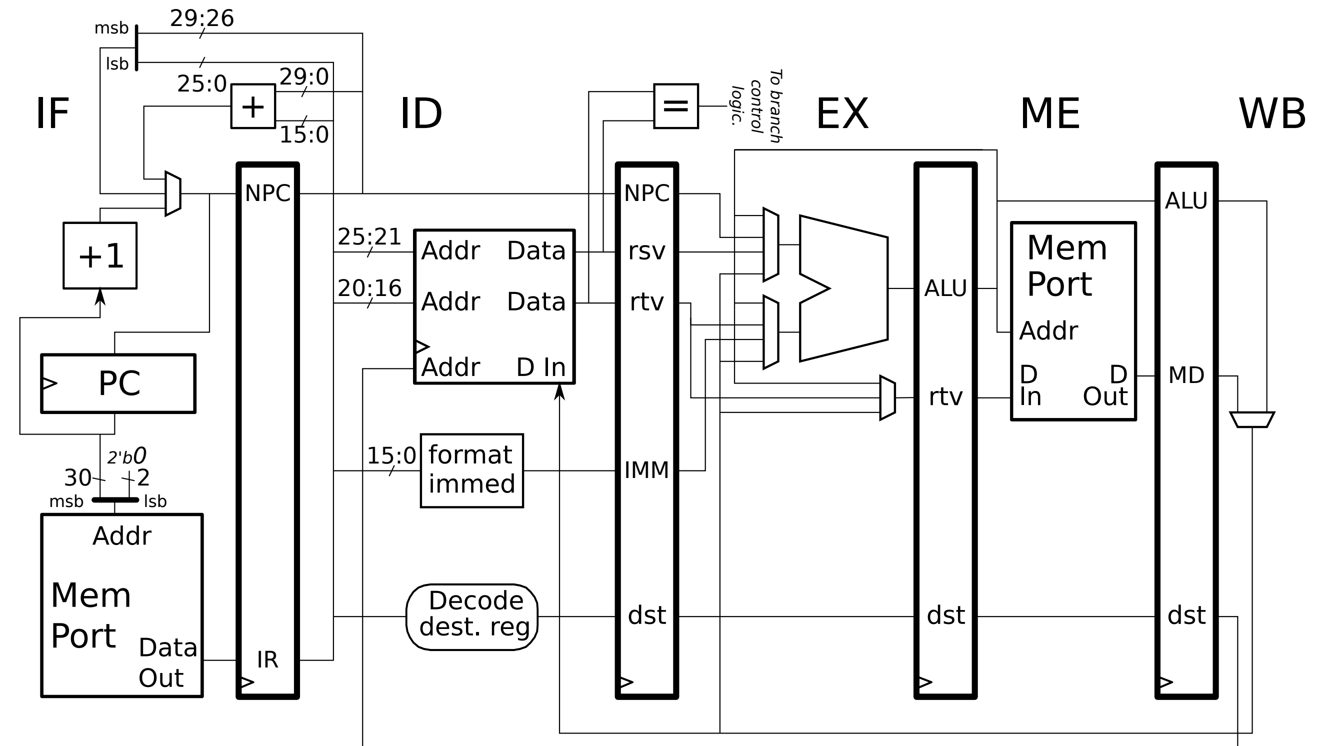

```
# Cycle                 0  1  2  3  4  5  6  7
add r1, r2, r3          IF ID EX ME WB
sub r4, r1, r5             IF ID EX ME WB
xori r6, r4, 0baa             IF ID EX ME WB
lw r8, 8(r9)                     IF ID EX ME WB
# Cycle                 0  1  2  3  4  5  6  7

# Register being written:        r1 r4 r6 r8
# These are 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.

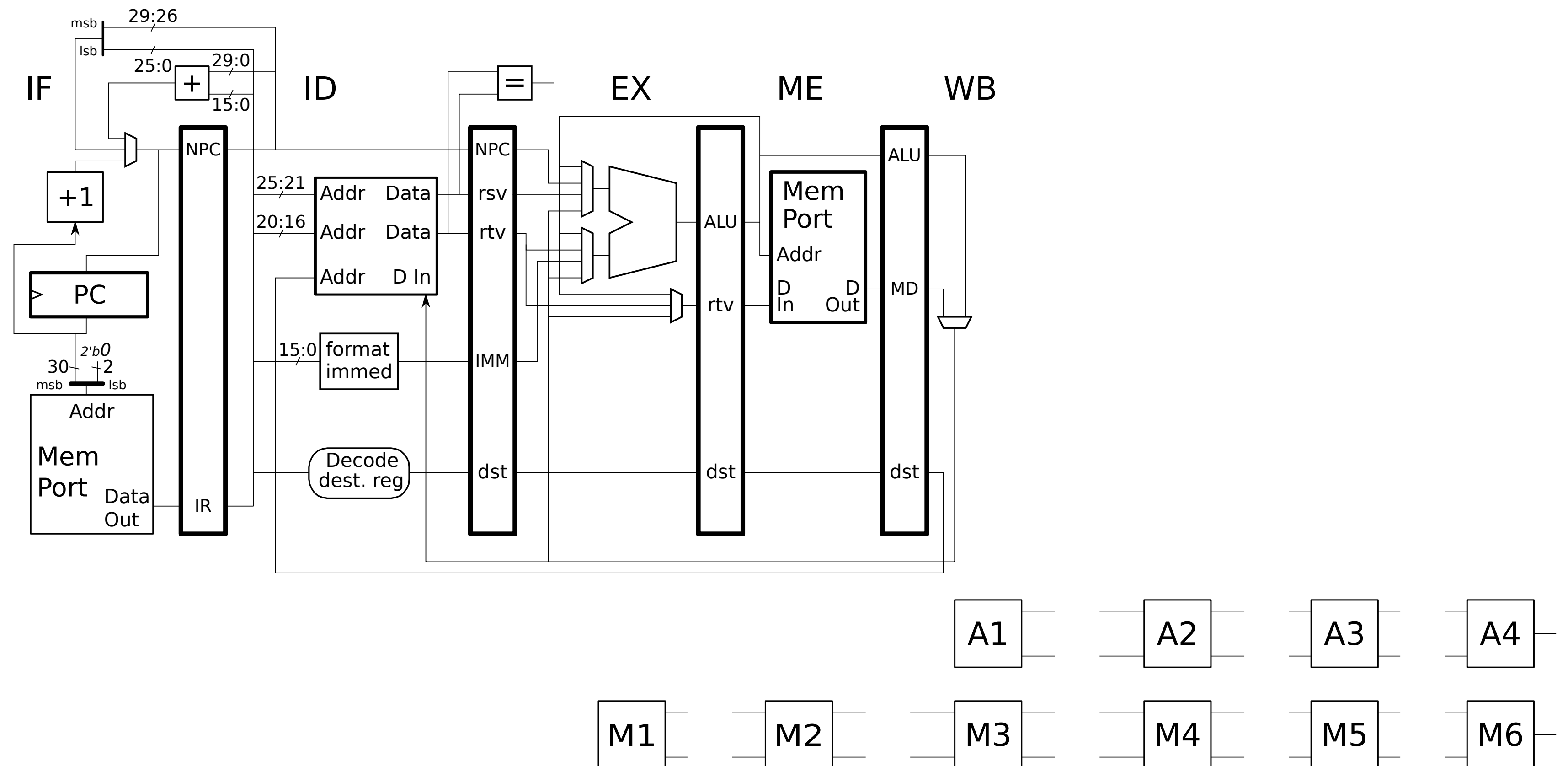


The End of Innocence

Unfortunately we must now set aside this simplicity and elegance.

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

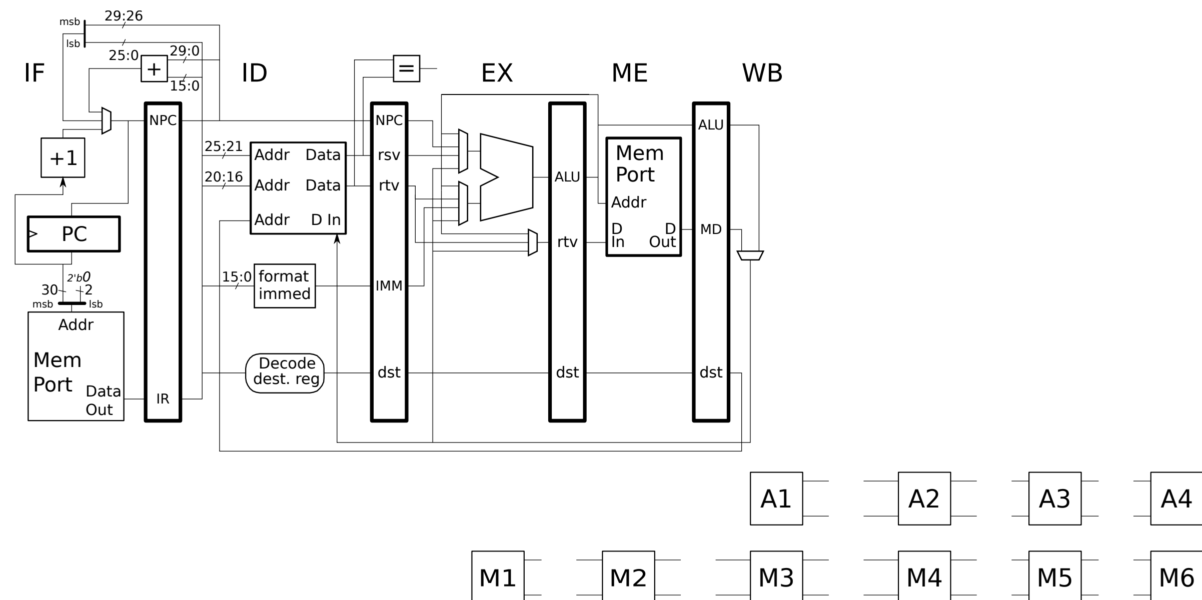
The challenge is putting these units in our pipeline.



Possible implementation strategies:

- 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.

  This is the approach used in most classroom examples.



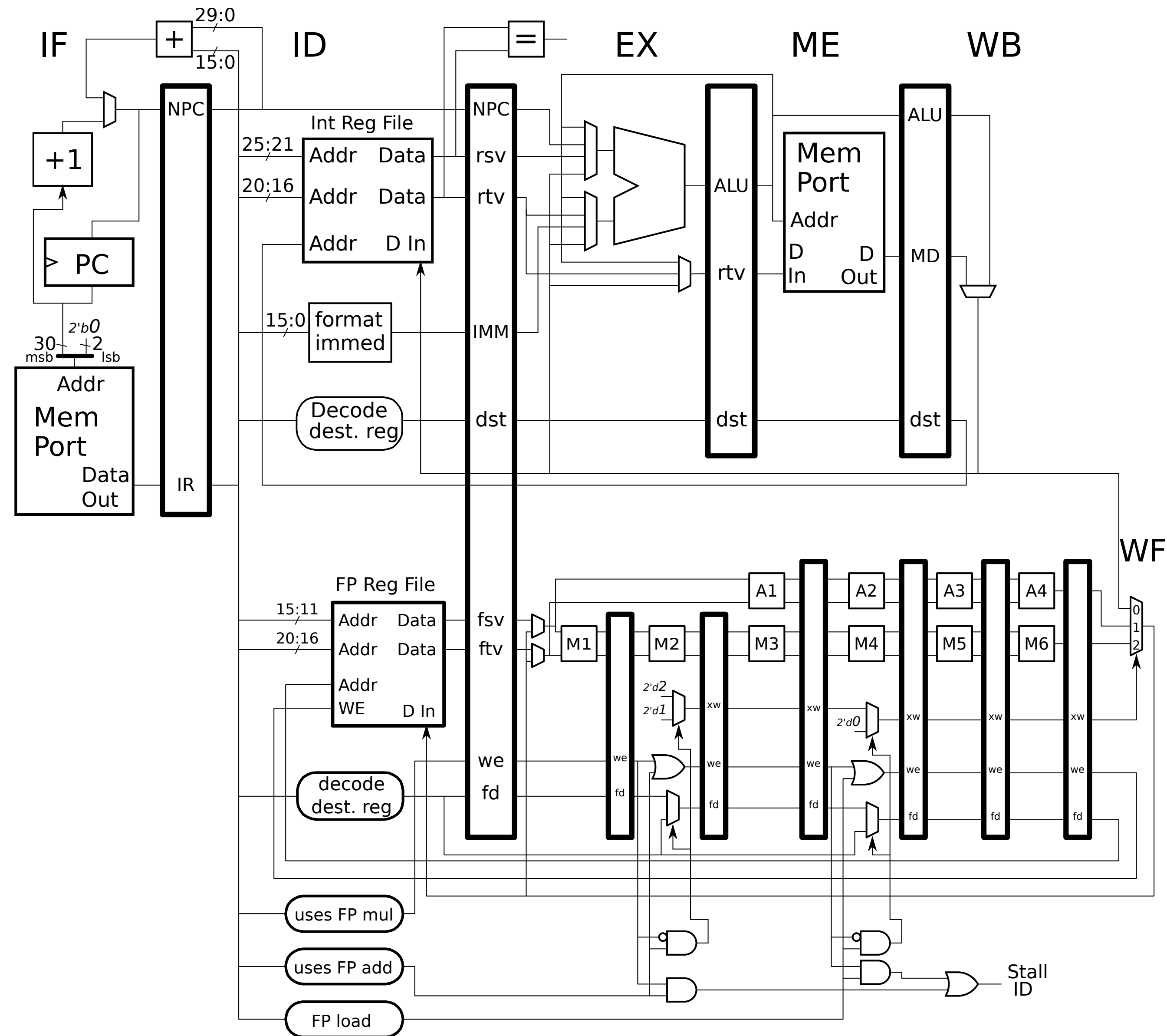
-8

Default (for course) FP Implementation

Separate register file for FP values (as defined by MIPS).

Split pipeline at ID.

Instructions requiring fewer stages, such as add.s, skip stages.



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`.

Functional Units and Pipelining

Implementations often balance cost and performance.

Consider an add operation that takes about 4 ns ...
... and that can be split into four stages, `A1`, `A2`, `A3`, `A4`.

Definitions

Initiation Interval:
The number of cycles that a functional unit stage will need to perform an operation.

The initiation interval for all stages in the MIPS integer pipeline is 1 ...
... and the initiation interval frequently used FP operations is usually 1.

Operation Latency:
The number of cycles needed to complete an operation.

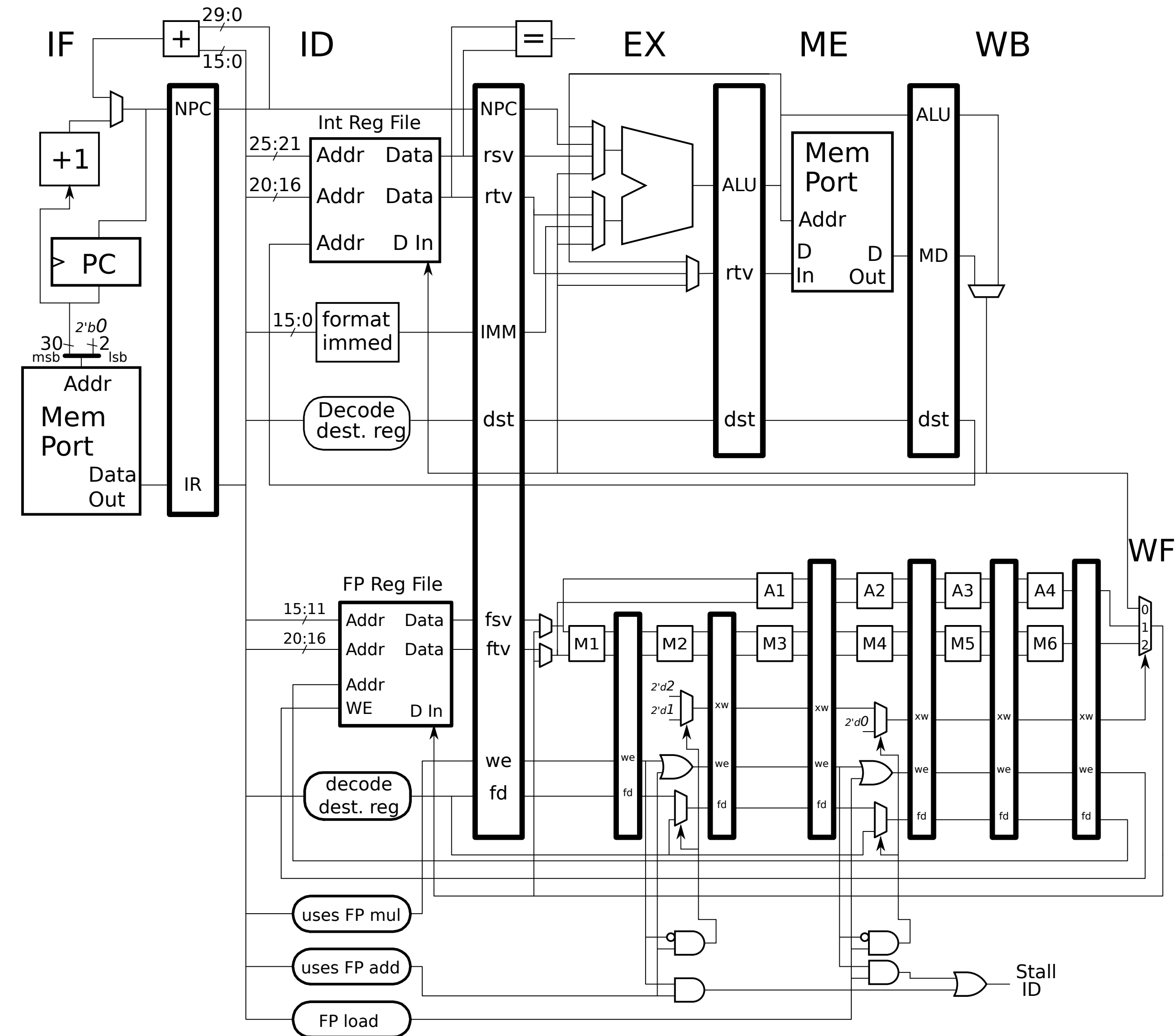
That's 1 for ALU operations, 4 for the addition above.

Highest Cost: *Fully Pipelined Functional Unit*

In a fully pipelined unit all stages have an initiation interval of 1.

Example, fully pipelined FP add:

```
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12
add.d f0, f2, f4     IF ID A1 A2 A3 A4 WF
add.d f6, f8, f10       IF ID A1 A2 A3 A4 WF
add.d F12, f14, f16        IF ID A1 A2 A3 A4 WF
add.d f18, F12, f20           IF ID -------> A1 A2 A3 A4 WF
```



The `add.d f18` stalls due to a dependence.

The initiation interval is 1. (True for all fully pipelined units.)

The add operation latency is 4. (True for this FP adder unit.)

Typically addition and multiplication units are fully pipelined.

Low Cost: *Unpipelined*

In an unpipelined unit the initiation interval equals the operation latency.

Example, unpipelined FP add

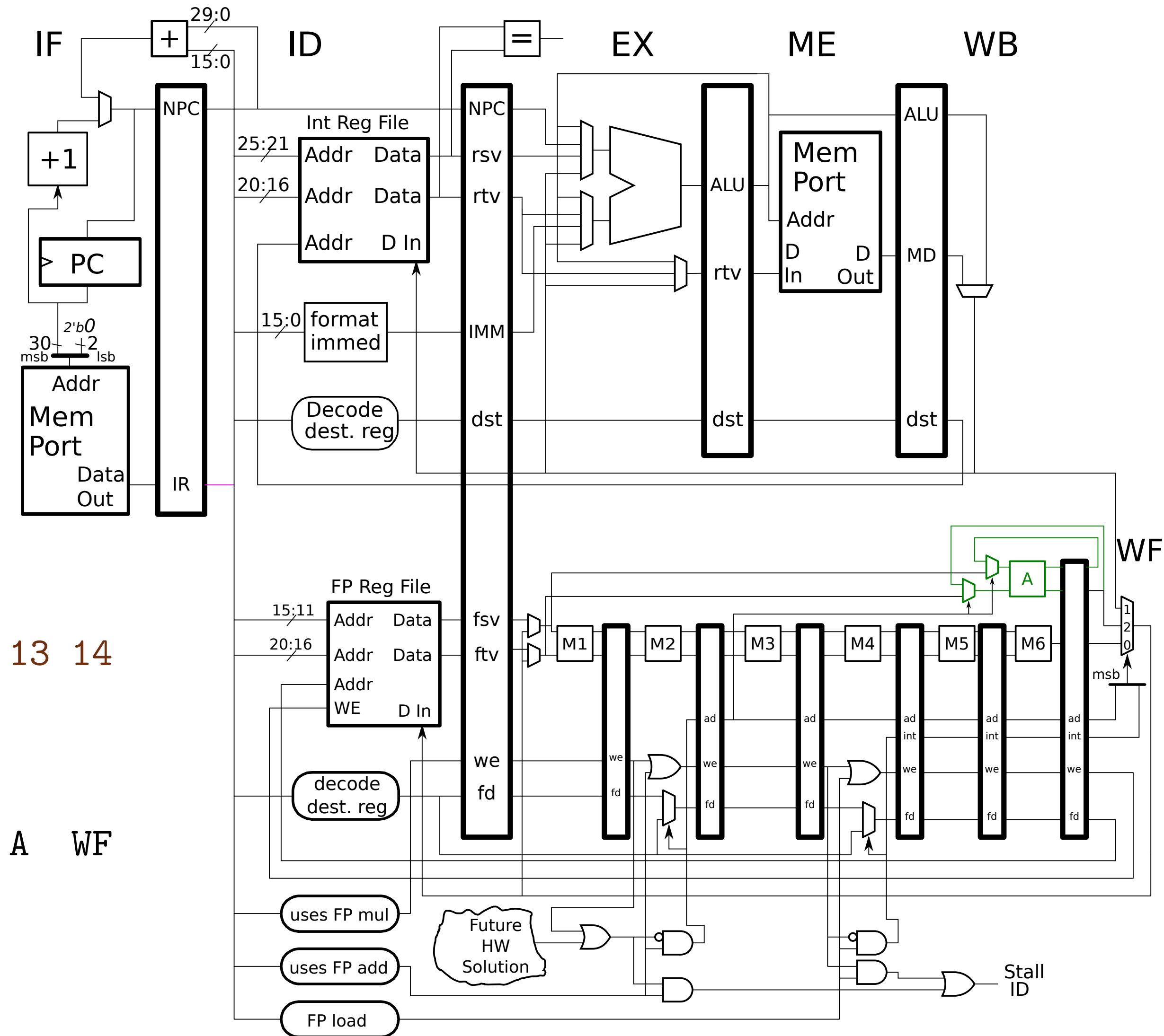
Suppose A1 through A4 are very similar . . .

. . . and so one piece of hardware, A,. . .

. . . can perform each stage's operation.

Then the entire computation could be done by using A four times:

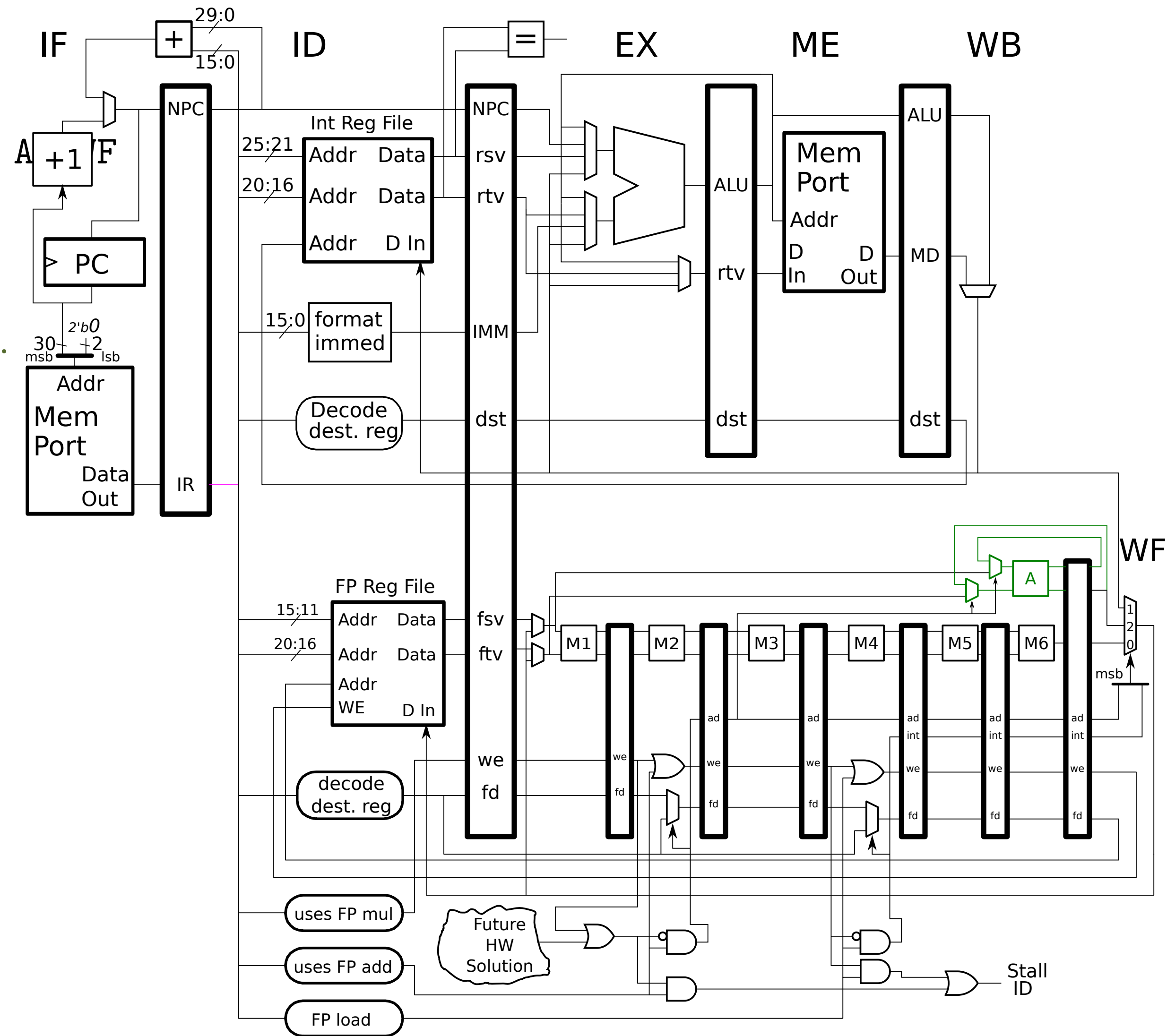
```
# Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
add.d f0, f2, f4    IF ID A  A  A  A  WF
add.d F6, f8, f10      IF ID -------> A  A  A  A  WF
addi r1, r1, 8            IF -------> ID EX ME WB
add.d f12, F6, f14                    IF ID ----> A  A  A  A  WF
```



-13

Continued: Example, unpipelined FP add.

```
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
add.d f0, f2, f4     IF ID A  A  A  A  WF
add.d F6, f8, f10       IF ID -------> A  A  A  A  WF
addi r1, r1, 8             IF -------> ID EX ME WB
add.d f12, F6, f14                     IF ID ----> A  A  A  A  WF
```



The add.d F6 stalls to use the A functional unit.

The add.d f12 stalls to dependence and use of the A functional unit.

The initiation interval of A is 4, the operation latency is 4.

Cost as low as $\frac{1}{4}$ of fully pipelined version.

But even non-dependent consecutive FP add instructions stall.

Division and trigonometric operations are usually unpipelined.

Intermediate Cost: Multiple Unpipelined Functional Units

Two unpipelined FP add units, A and B.

```
# Cycle            0  1  2  3  4  5  6  7  8  9  10
add.d f0, f2, f4   IF ID A  A  A  A  WF
add.d f6, f8, f10     IF ID B  B  B  B  WF
add.d f12, f14, f16      IF ID ----> A  A  A  A  WF
```

Initiation interval of both A and B are 4.

Intermediate Cost: *Partially Pipelined* Functional Units

A unit is partially pipelined if its stages have an initiation interval strictly greater than 1, and strictly less than the operation latency.

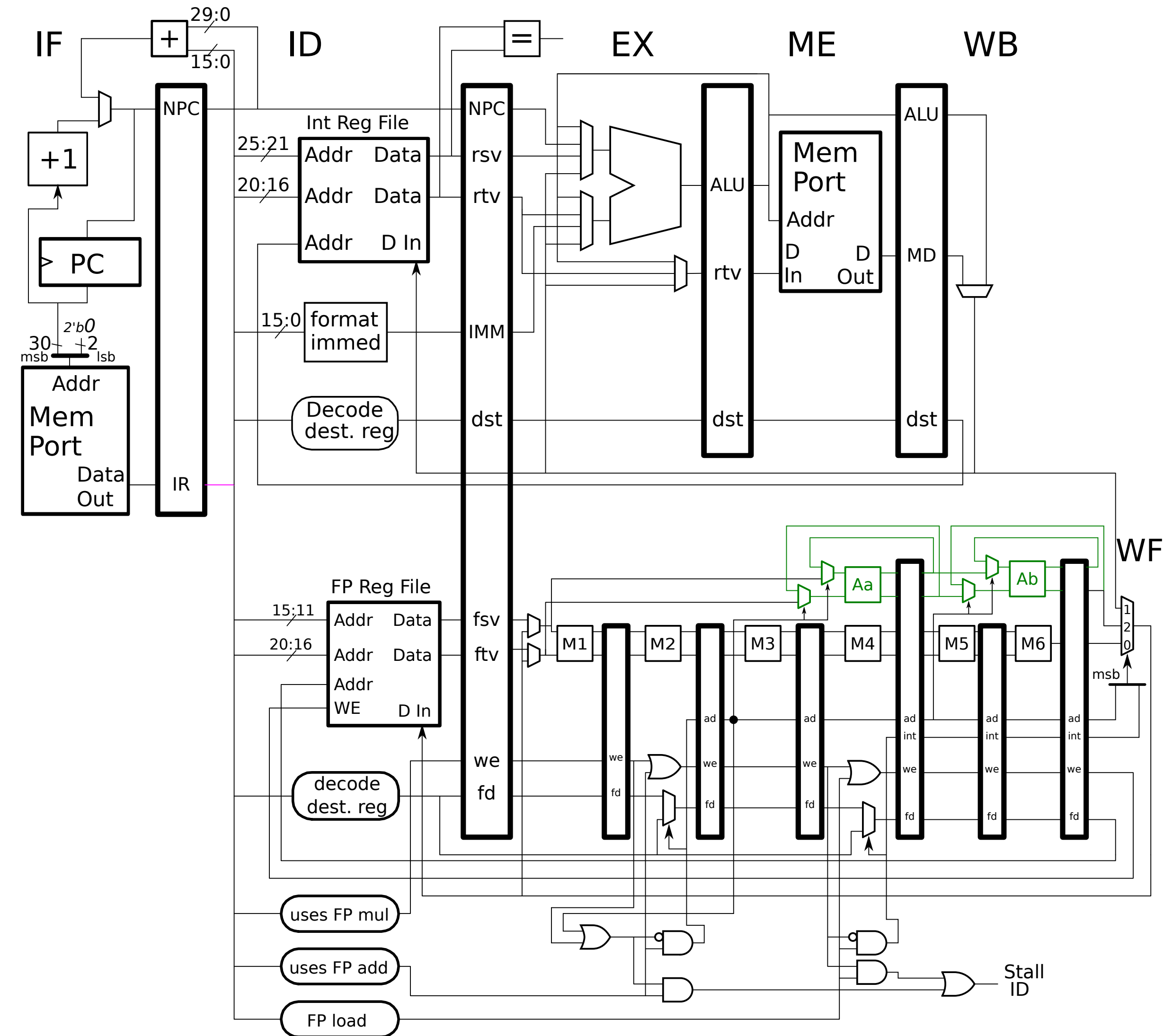
Example, partially pipelined add.

Consider an adder in which `Aa` performs either `A1` or `A2` . . .

. . . and `Ab` performs either `A3` or `A4`.

```
# Cycle              0  1  2  3  4  5  6  7  8  9  10
add.d f0, f2, f4     IF ID Aa Aa Ab Ab WF
add.d f6, f8, f10       IF ID -> Aa Aa Ab Ab WF
add.d f12, f14, f16        IF -> ID -> Aa Aa Ab Ab WF
```

Stages `Aa` and `Ab` each of an initiation interval of 2.



-16

Floating Point in Chapter-3 MIPS Implementation

Typical Classroom Example Floating Point Functional Units

- FP Add

  Four stages, fully pipelined: Operation Latency 4, Initiation Interval 1.

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

- FP Multiply

  Six stages, fully pipelined: Operation Latency 6, Initiation Interval 1.

  Used for FP Multiply.

- FP Divide

  Twenty five cycles, unpipelined: Operation Latency 25, Initiation Interval 25.

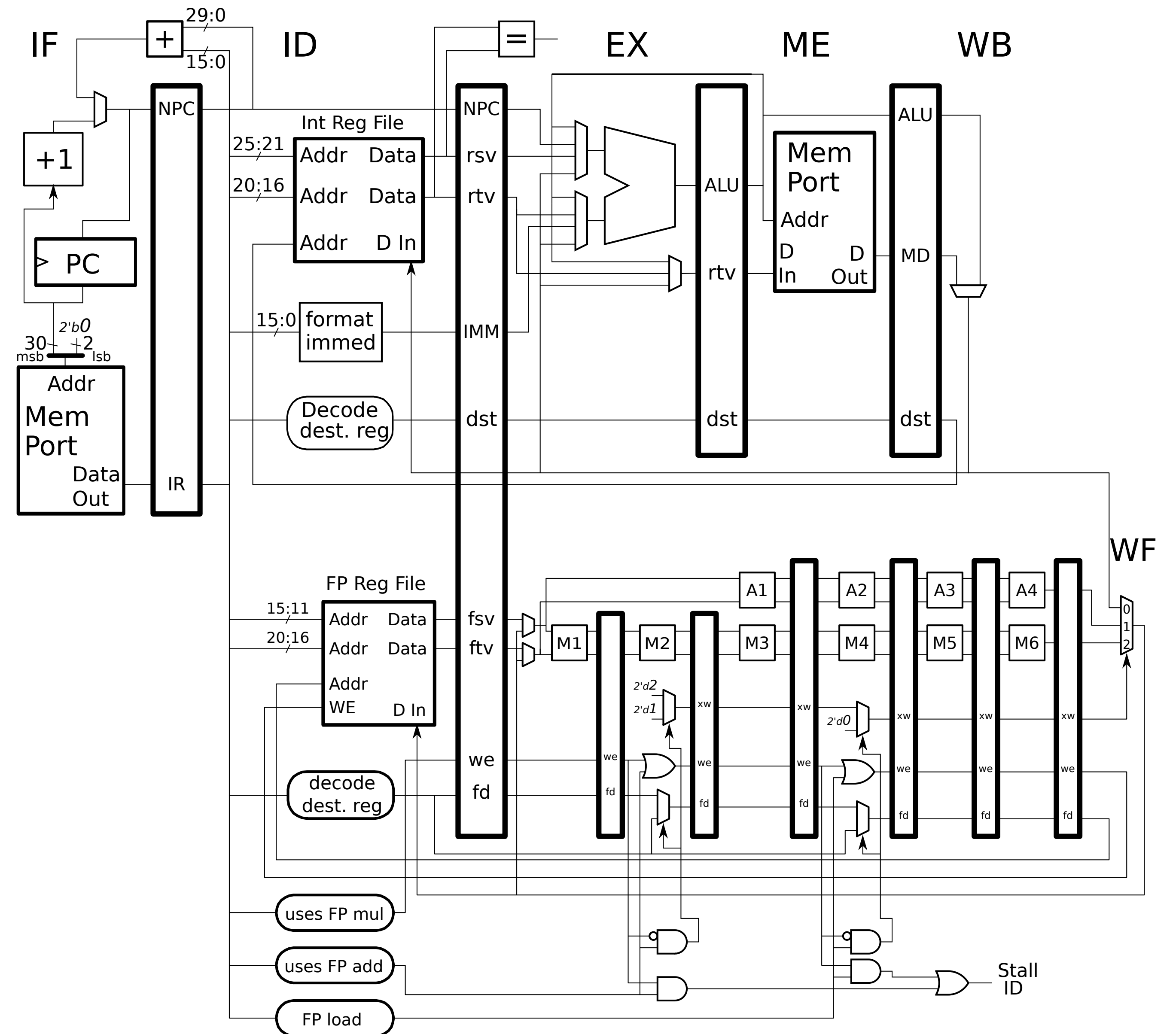
Classroom Floating-Point MIPS Pipeline

Example floating unit implementation main features:

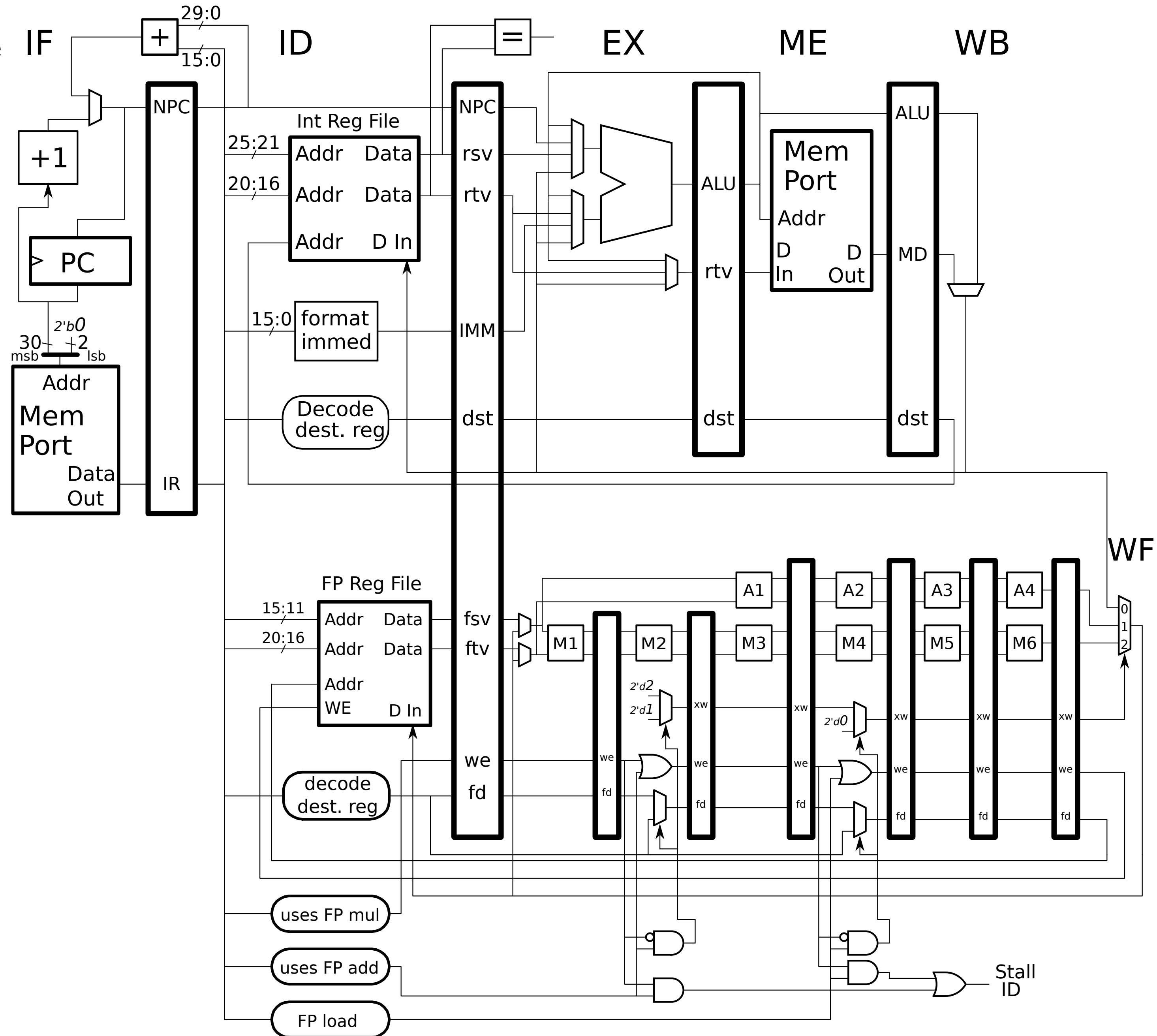
Separate register file.

Number of stages vary depending on functional unit.

Floating-point writeback separate from integer writeback.



Floating-Point Pipeline



-19

Floating-Point Pipeline

Example floating unit implementation notes:

Paths to implement FPR $\rightarrow$ GFP not shown.

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

Use of register pairs for double operands ignored.

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

The divide functional unit is not shown.

Hazards With Long-Latency Instructions in Classroom Pipeline

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).

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.

-23

Handling Functional Unit Structural Hazards

Example, 6-cycle unpipelined divide.

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

```
# Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
div.d f0, f2, f4  IF  ID  DIV DIV DIV DIV DIV DIV WF
div.d f6, f8, f10     IF  ID  ------------------> DIV DIV 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.

Eliminating Hazards

Provide more than one functional unit.

Example, provide two 6-cycle divide units, `DVa` and `DVb`.

```
# Cycle           0   1   2   3   4   5   6   7   8   9
div.d f0, f2, f4  IF  ID  DVa DVa DVa DVa DVa DVa WF
div.d f6, f8, f10     IF  ID  DVb DVb DVb DVb DVb DVb WF
```

Pipeline functional unit.

Example, use 6-cycle, initiation interval 2, pipelined divide . . .
. . . and live with single stall cycle.

```
# Cycle           0   1   2   3   4   5   6   7   8   9   10
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
```

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
```

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.

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.

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.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.

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 ME.

```
# Cycle                 0   1   2   3   4   5
mul.s f0, f1, f2        IF  ID  M1  M2  M3 WF
add.s f0, f3, f4            IF  ID  A1  WF       # 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 ME.

```
# Cycle                0   1   2   3   4   5   6   7
mul.s f0, f1, f2       IF  ID   M1  M2  M3 WF
sub.s f5, f0, f6           IF  ID  ----->  A1  WF
add.s f0, f3, f4               IF  ----->  ID  A1  WF # No problem.
```

Handling WAW Hazards

If there is no intervening write. . .

. . . and ISA insists that *all* faulting instructions raise exceptions. . .

. . . the WF can be suppressed by changing `we` to `0`.

```
# Cycle               0    1    2    3    4    5
mul.s f0, f1, f2      IF   ID   M1   M2   M3  WFx
add.s f0, f3, f4           IF   ID   A1   WF
```

If `mul.s` faulted the handler would be called in execution above.

For a less strict ISA instruction could be squashed earlier.

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  M1  M2  M3  M4  M5  M6  M7  M8  WF
add.s f1, f3, f4      IF  ID  A1  A2  A3  A4  WF
```

There *would* be a WAR hazard if `add.s` wrote `f1` before `mul.s` read it.

That can't happen since `mul.s` would leave ID (with `f1`) as `add.s` just enters ID.

Instruction Throughput (IPC) and Multicycle Operations

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

Finding Instruction Throughput (IPC) For a Loop

As before, find a repeating pattern of iterations.

Look out for structural hazards.

Determining Instruction Throughput (IPC)

Instruction Throughput:
The number of instructions executed per unit time of some code fragment on some hardware.

IPC:
Instructions Per Cycle, a unit of instruction throughput. Also written as insn/cycle.

CPI:
Cycles Per Instruction, the reciprocal of IPC. This measure of instruction execution efficiency can be misleading so it is not used much in this course. Do not confuse CPI with the number of cycles needed to execute an individual instruction.

Code running on the five-stage MIPS pipeline . . .

. . . the runs without stalls (do to good scheduling) . . .

. . . has an instruction throughput of 1 insn/cycle.

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 instruction throughput during the execution of this loop?

```
# 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 16 17 18 19 20 21
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                   0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23
LOOP: #                   Fourth Iteration
 addi $t0, $t0, -1                                                        IF ID EX ME WB
```

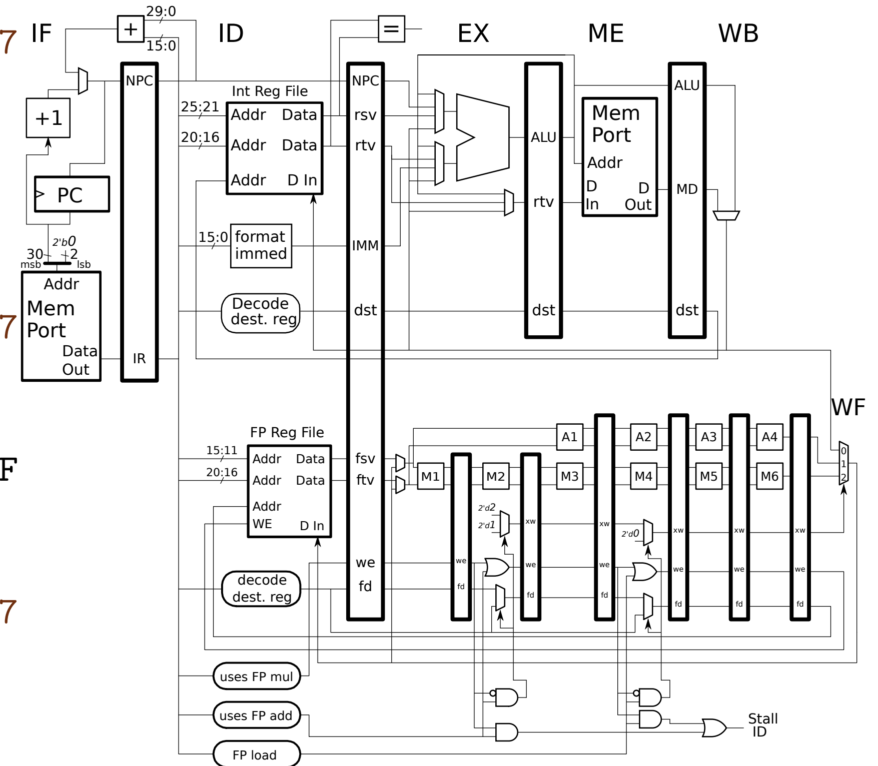
Note: Each iteration above starts differently.

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.)

```
# 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.

Throughput is $\frac{4\,\text{insn}}{16\,\text{cyc} - 10\,\text{cyc}} = \frac{2}{3}\,\text{insn/cycle}$.

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.

```
# Cycle           0   1   2   3   4   5   6   7   8   9
mul.s f0, f1, f2  IF  ID  M1  M2  M3  M4  M5  M6 *M7* WF
add.s f1, f3, f4      IF  ID  A1  A2  A3  A4  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.

Method 1: Stall so that instructions complete in order.

```
# Cycle           0   1   2   3   4   5   6   7   8   9   10
mul.s f0, f1, f2  IF  ID  M1  M2  M3  M4  M5  M6  M7  WF
add.s f1, f3, f4      IF  ID  ---------> A1  A2  A3  A4  WF
```

This works, (WF 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 M6.

Then at cycle 8 (below) `add.s` can write (no chance of an exception in M7).

```
# Cycle:       0   1   2   3   4   5   6   7   8   9
mul.s f0,f1,f2 IF  ID  M1  M2  M3  M4  M5  M6  M7  WF
add.s f1,f3,f4     IF  ID  ->  A1  A2  A3  A4  WF
```

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

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

Second `add.s` doesn't stall since an exception in `mul.sip` 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  M1  M2  M3  M4  M5  M6  M7  WF
add.s   f1,f3,f4        IF  ID  --------->  A1  A2  A3  A4  WF
mul.sip f5,f6,f7            IF  --------->  ID  M1  M2  M3  M4  M5  M6  M7  WF
add.s   f6,f8,f9                            IF  ID  A1  A2  A3  A4  WF
```

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  M1  M2  M3  M4  M5  M6  M7  WF
testexc     IF  ID  ------------------> EX  ME  WF
add.s           IF  ------------------> ID  A1  A2  A3  A4  WF
mul.s                                   IF  ID  M1  M2  M3  M4  M5  M6  M7  WF
add.s                                       IF  ID  A1  A2  A3  A4  WF
```