Pipelined Processor Implementation Wrap-Up
April 2, 1998

Exceptions

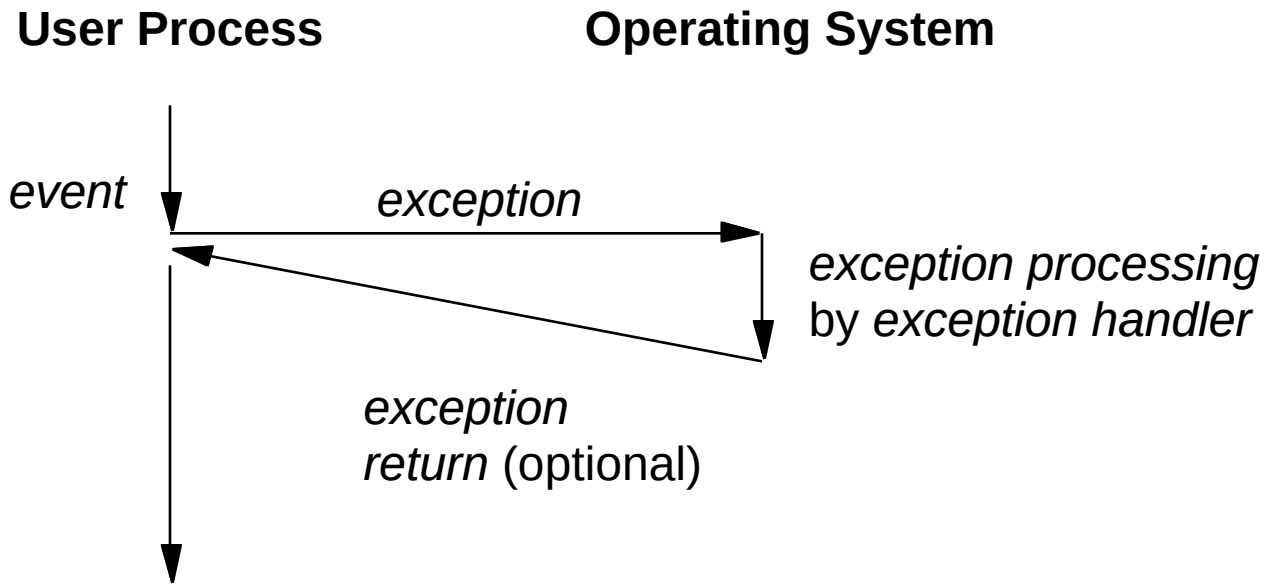
- basic issues
- simple implementation

Multicycle Instructions

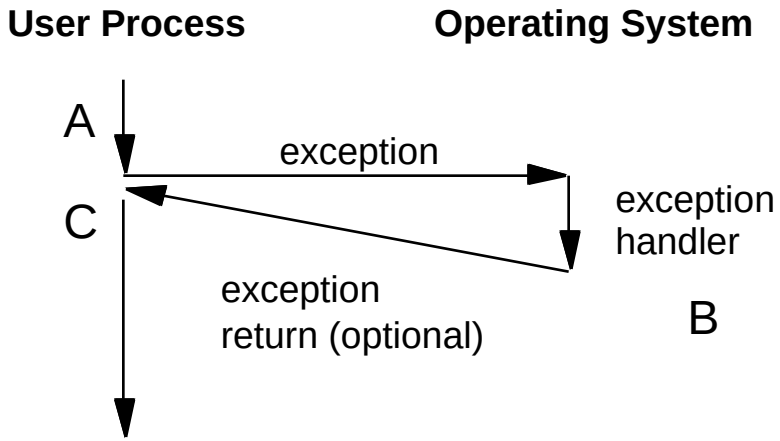
- integer multiplication
- floating point

Exceptions

An *exception* is a transfer of control to the OS in response to some *event* (i.e. change in processor state)



Issues with Exceptions



A1: What kinds of events can cause an exception?

A2: When does the exception occur?

B1: How does the handler determine the location and cause of the exception?

B2: Are exceptions allowed within exception handlers?

C1: Can the user process restart?

C2: If so, where?

Internal (CPU) Exceptions

Internal exceptions occur as a result of events generated by executing instructions.

Execution of a CALL_PAL instruction.

- **allows a program to transfer control to the OS**

Errors during instruction execution

- **arithmetic overflow, address error, parity error, undefined instruction**

Events that require OS intervention

- **virtual memory page fault**

External (I/O) exceptions

External exceptions occur as a result of events generated by devices external to the processor.

I/O interrupts

- hitting ^C at the keyboard
- arrival of a packet
- arrival of a disk sector

Hard reset interrupt

- hitting the reset button

Soft reset interrupt

- hitting ctl-alt-delete on a PC

Exception handling (hardware tasks)

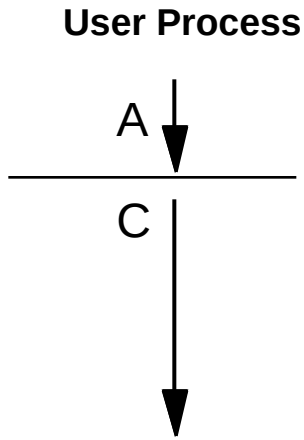
Recognize event(s)

Associate one event with one instruction.

- **external event: pick any instruction**
- **multiple internal events: typically choose the earliest instruction.**
- **multiple external events: prioritize**
- **multiple internal and external events: prioritize**

Create Clean Break in Instruction Stream

- **Complete all instructions before excepting instruction**
- **Abort excepting and all following instructions**
  - this clean break is called a *"precise exception"*



Exception handling (hardware tasks)

Set status registers

- **Exception Address: the EXC_ADDR register**
  - external exception: address of instruction about to be executed
  - internal exception: address of instruction causing the exception
    - » except for arithmetic exceptions, where it is the following instruction
- **Cause of the Exception: the EXC_SUM and FPCR registers**
  - was the exception due to division by zero, integer overflow, etc.
- **Others**
  - which ones get set depends on CPU and exception type

Disable interrupts and switch to kernel mode

Jump to common exception handler location

Exception handling (software tasks)

Deal with event

(Optionally) resume execution

- **using special `REI` (return from exception or interrupt) instruction**
- **similar to a procedure return, but restores processor to user mode as a side effect.**

Where to resume execution?

- **usually re-execute the instruction causing exception**

Precise vs. Imprecise Exceptions

In the Alpha architecture:

- **arithmetic exceptions may be *imprecise* (similar to the CRAY-1)**
  - motivation: simplifies pipeline design, helping to increase performance
- **all other exceptions are precise**

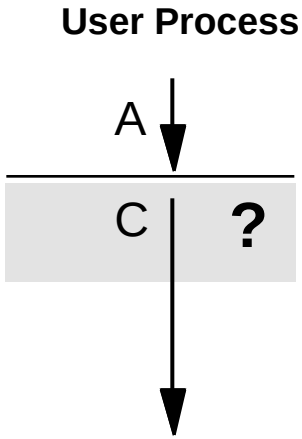
Imprecise exceptions:

- **all instructions before the excepting instruction complete**
- **the excepting instruction and instructions after it may or may not complete**

What if precise exceptions are needed?

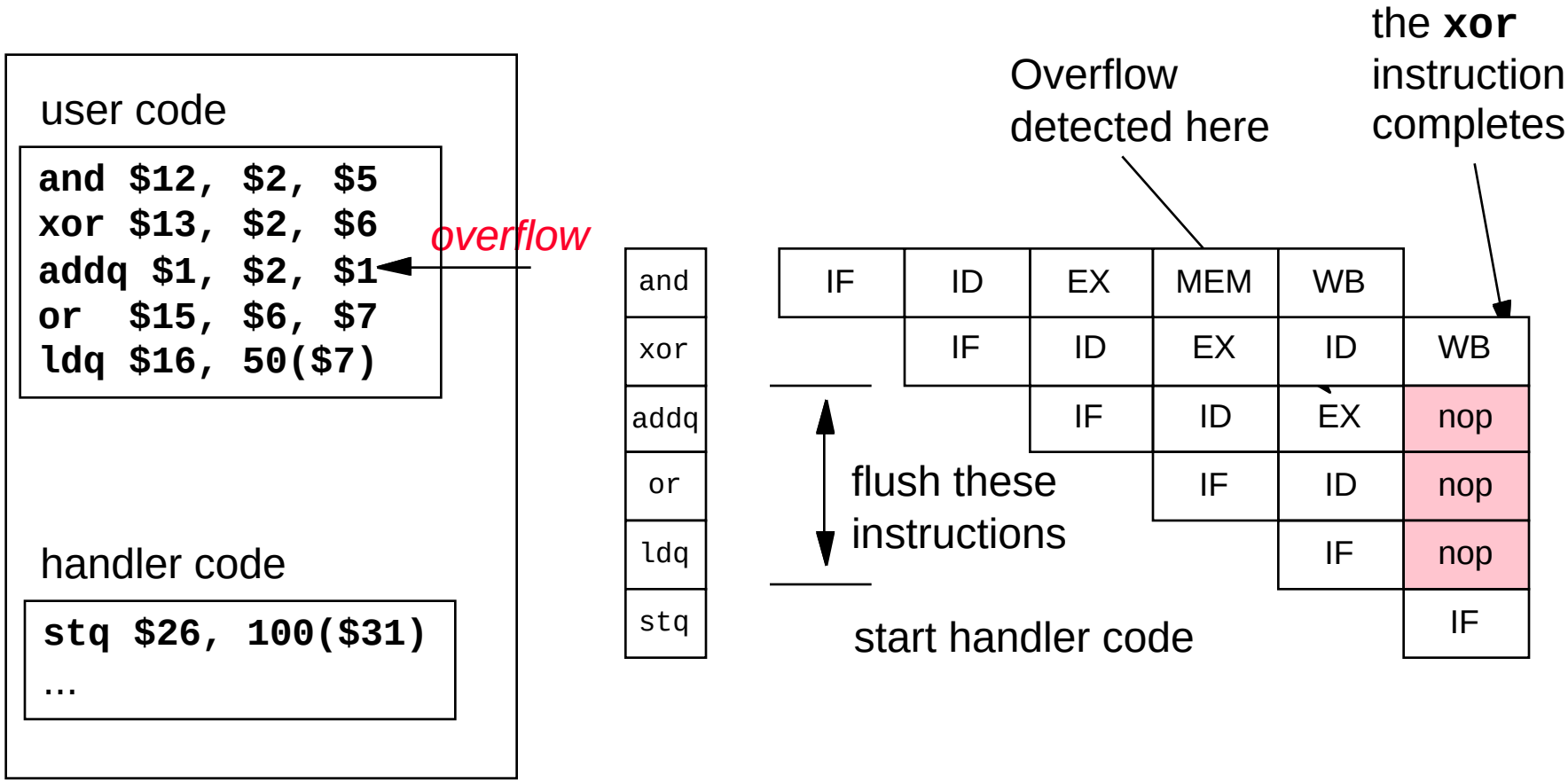
- **insert a TRAPB (trap barrier) instruction immediately after**
  - stalls until certain that no earlier insts take exceptions

In the remainder of our discussion, assume for the sake of simplicity that all Alpha exceptions are precise.



Example: Integer Overflow

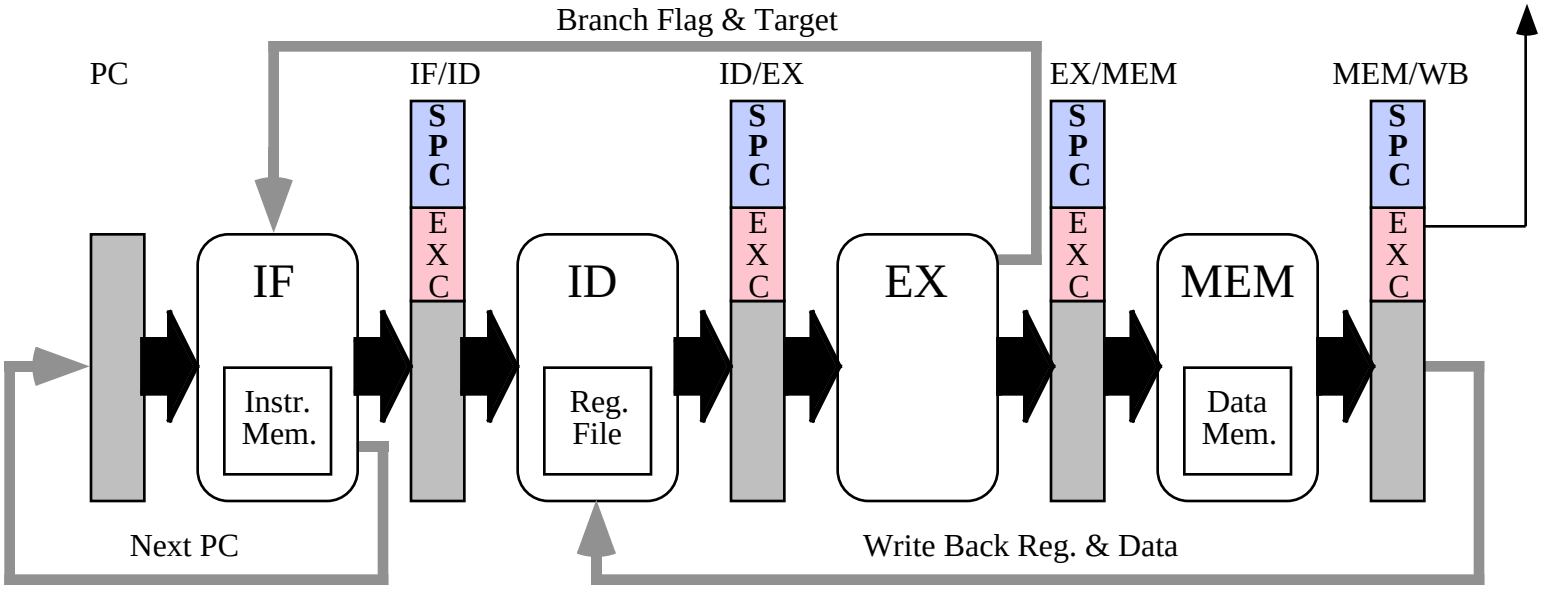
(This example illustrates a *precise* version of the exception.)



Exception Handling in pAlpha Simulator

Relevant Pipeline State

- **Address of instruction in pipe stage (SPC)**
- **Exception condition (EXC)**
  - Set in stage when problem encountered
    - » IF for fetch problems, EX for instr. problems, MEM for data probs.
  - Triggers special action once hits WB



Alpha Exception Examples

- **In directory *HOME347*/public/sim/demos**

Illegal Instruction (exc01.O)

```
0x0: sll   r3, 0x8, r5  # unimplemented
0x4: addq r31, 0x4, r2  # should cancel
```

Illegal Instruction followed by store (exc02.O)

```
0x0:  addq r31, 0xf, r2
0x4:  sll  r3, 0x8, r5  # unimplemented
0x8:  stq  r2, 8(r31)   # should cancel
```

More Examples: Multiple Exceptions

EX exception follows MEM exception (exc03.O)

```
0x0: addq r31, 0x3, r3
0x4: stq   r3, -4(r31)  # bad address
0x8: sll   r3, 0x8, r5  # unimplemented
0xc: addq r31, 0xf, r2
```

MEM exception follows EX exception (exc04.O)

```
0x0: addq r31, 0x3, r3
0x4: sll   r3, 0x8, r5  # unimplemented
0x8: stq   r3, -4(r31)  # bad address
0xc: addq r31, 0xf, r2
```

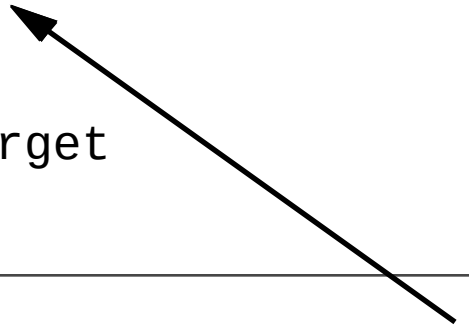
These exceptions are detected *simultaneously* in the pipeline!

Which is the excepting instruction?

Final Alpha Exception Example

Avoiding false alarms (exc05.O)

```
  0x0:  beq   r31, 0xc       # taken
  0x4:  sll    r3, 0x8, r5   # should cancel
  0x8:  bis   r31, r31, r31
  0xc:  addq r31, 0x1, r2   # target
  0x10: call_pal halt
```



Exception detected in the pipeline, but should not really occur.

Implementation Features

Correct

- **Detects excepting instruction**
  - Furthest one down pipeline = Earliest one in program order
  - (e.g., **exc03.O** vs. **exc04.O**)
- **Completes all preceding instructions**
- **Usually aborts excepting instruction & beyond**
- **Prioritizes exception conditions**
  - Earliest stage where instruction ran into problems
- **Avoids false alarms (exc05.O)**
  - Problematic instructions that get canceled anyhow

Shortcomings

- **Store following excepting instruction (exc02.O)**

Requirements for Full Implementation

Exception Detection

- **Detect external interrupts at IF**
  - Complete all fetched instructions

Instruction Shutdown

- **Suspend if unusual condition in MEM or WB**
- **Save proper value of EXC_ADDR**
  - Not always same as SPC
- **Rest of control state**

Handler Startup

- **Begin fetching handler code**

Multicycle instructions

Alpha 21264 Execution Times:

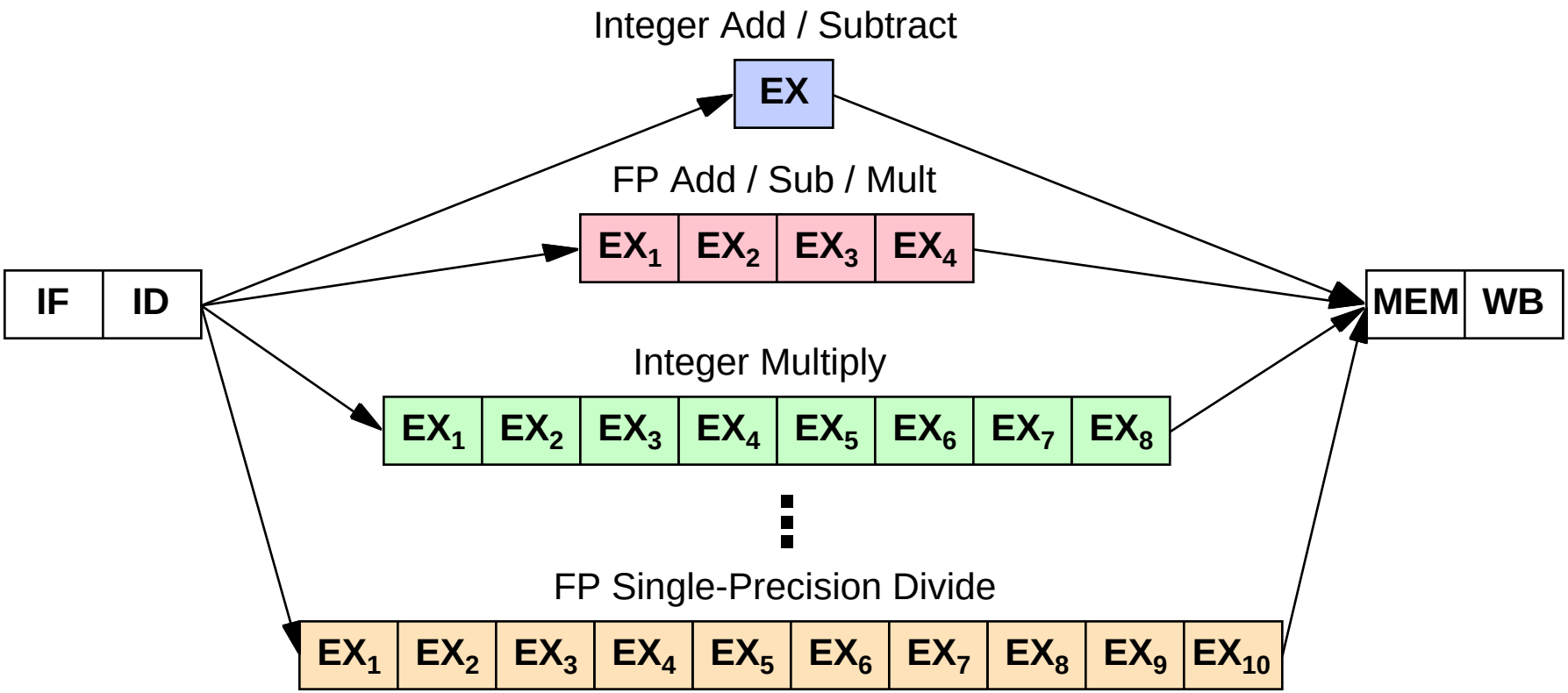
- Measured in clock cycles

Operation	Integer	FP-Single	FP-Double
add / sub	1	4	4
multiply	8-16	4	4
divide	N / A	10	23

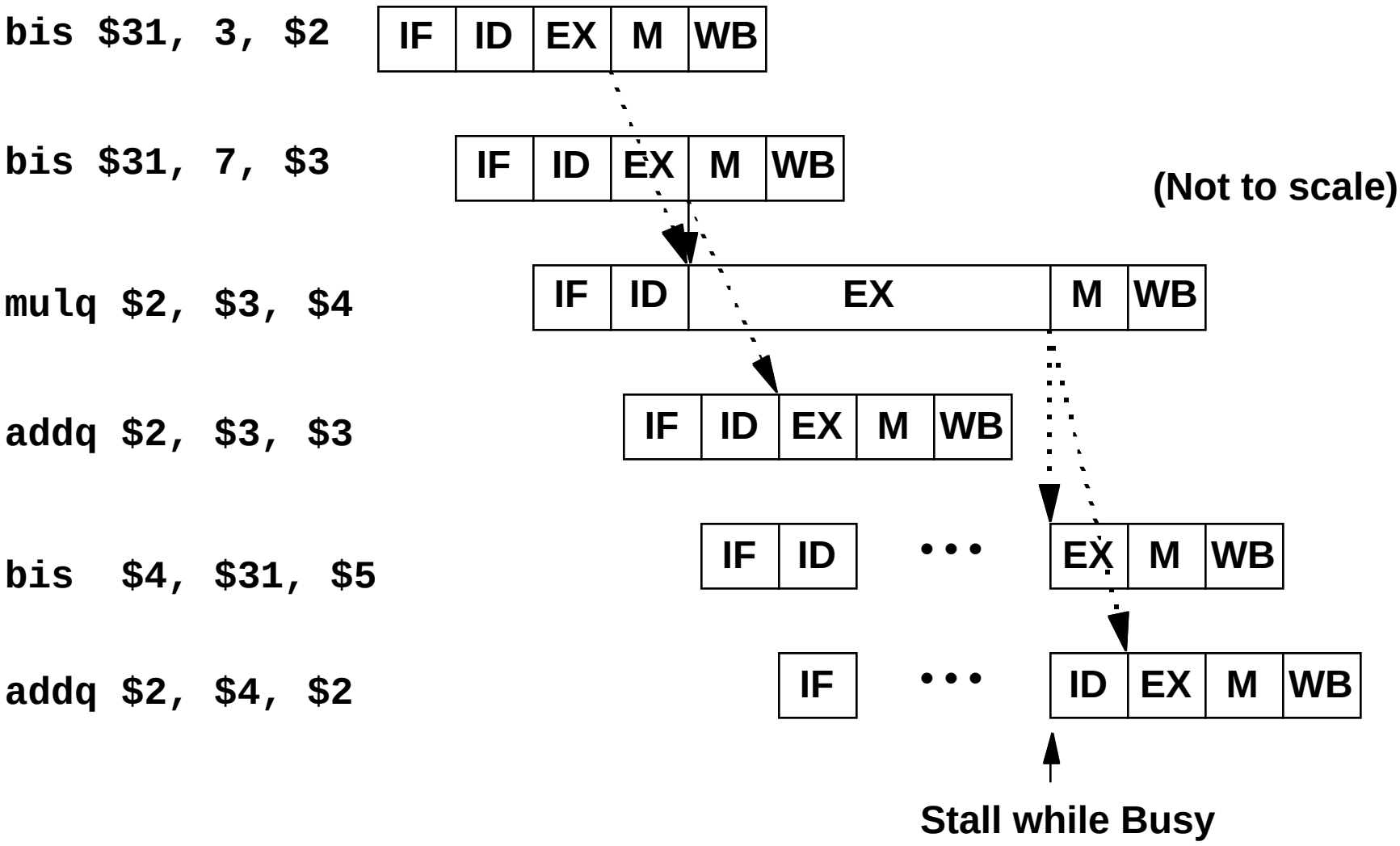
H&P Dynamic Instruction Counts:

Operation	Integer Benchmarks	FP Benchmarks Integer	FP Benchmarks FP
add / sub	14%	11%	14%
multiply	< 0.1%	< 0.1%	13%
divide	< 0.1%	< 0.1%	1%

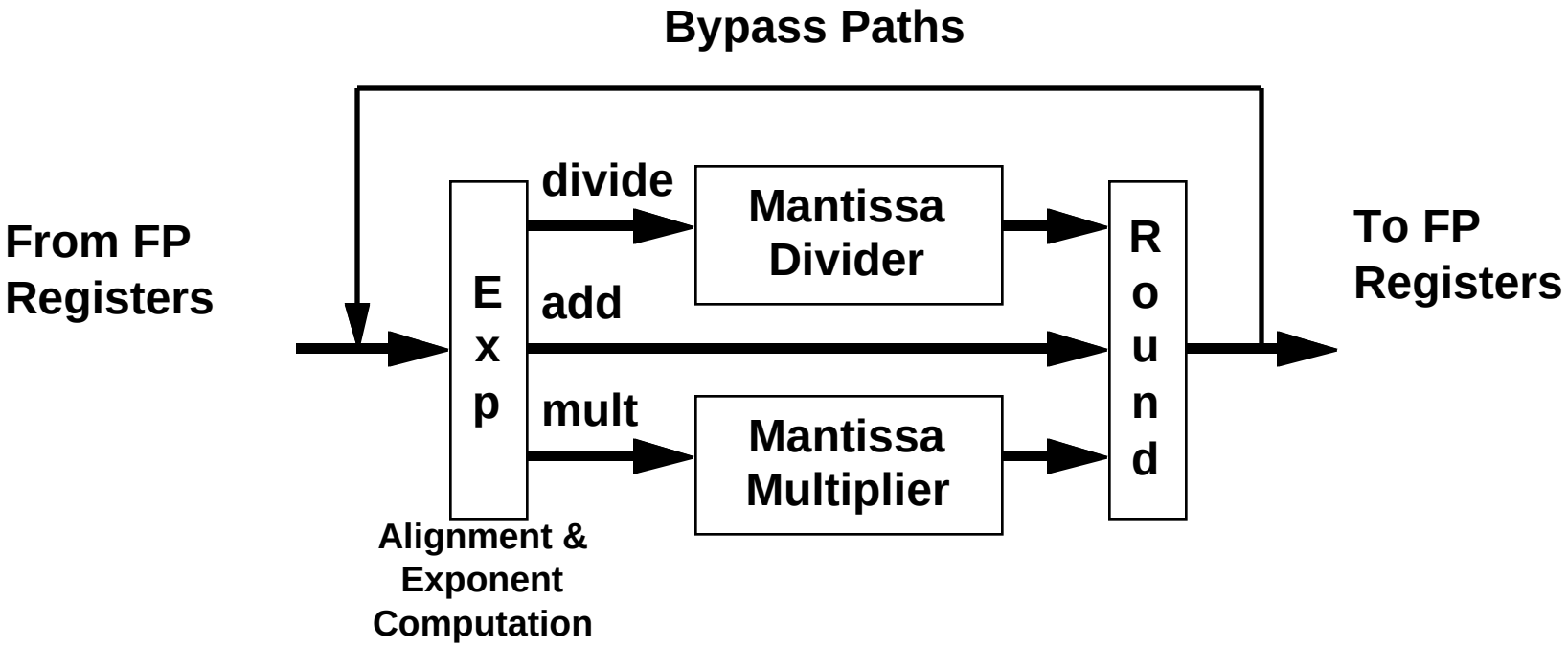
Pipeline Revisited



Multiply Timing Example



Floating Point Hardware (from MIPS)



Independent Hardware Units

- Can concurrently execute add, divide, multiply
- Except that all share exponent and rounding units
- Independent of integer operations

Control Logic

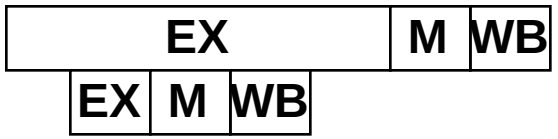
Busy Flags

- **One per hardware unit**
- **One per FP register**
  - Destination of currently executing operation

Stall Instruction in ID if:

- **Needs unit that is not available**
- **Source register busy**
  - Avoids RAW (Read-After-Write) hazard
- **Destination register busy**
  - Avoids WAW hazard

```
divt $f1, $f2, $f4
addt $f1, $f2, $f4
```



Bypass paths

- **Similar to those in integer pipeline**

FP Timing Example

```
addt $f2, $f4, $f1
divt $f1, $f2, $f6
```

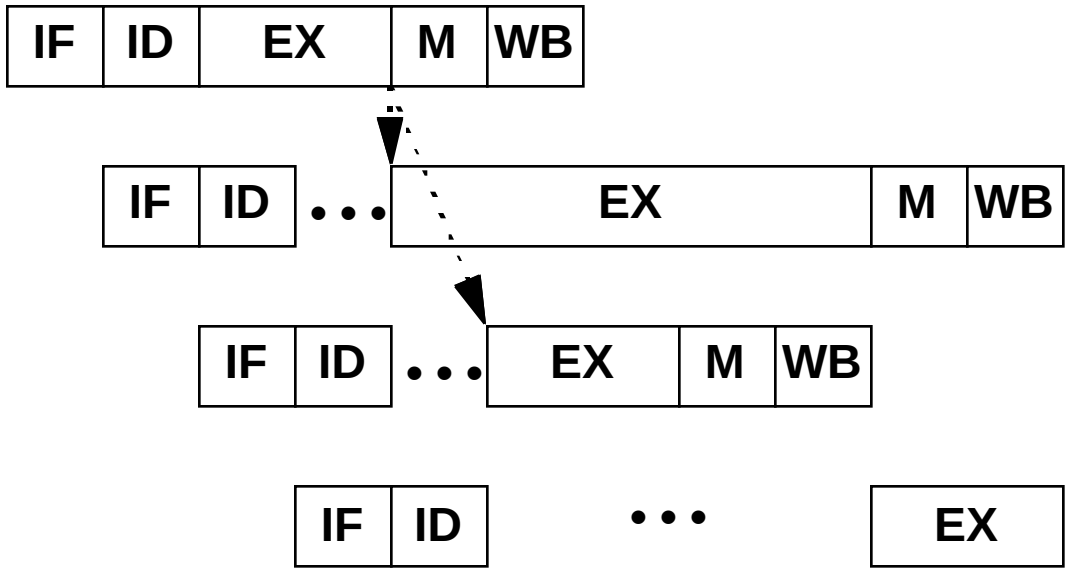
RAW Stall

```
addt $f1, $f2, $f8
```

Structural Stall

```
addt $f1, $f2, $f6
```

WAW Stall



Conclusion

Pipeline Characteristics for Multi-cycle Instructions

- **In-order issue**
  - Instructions fetched and decoded in program order
- **Out-of-order completion**
  - Slow instructions may complete after ones that are later in program order

Performance Opportunities

- **Transformations such as loop unrolling & software pipelining to expose potential parallelism**
- **Schedule code to use multiple functional units**
  - Must understand idiosyncracies of pipeline structure