Interrupt Types and Overview

Terminology in Text and in Class

The book uses "exception" as a general term for all interrupts ...

... in these notes interrupt is used as the general term ...

... and a narrower definition is used for exception.

The definitions of trap, interrupt, and exception given here ...

... are not explicitly provided in the text ...

... but are widely used.

Interrupt, Handler, Modes

Interrupt:
Event that requires OS attention.

OS takes control of computer, . . .

. . . attends to whatever caused the interrupt, . . .

. . . and (most of the time) resumes interrupted program.

Handler:
The code that takes control in response to interrupt. Typically that code is part of the OS.

Privileged Mode:
A state in which the CPU controller and memory system . . .

. . . do not restrict which instructions can be executed . . .

. . . or the memory addresses that can be accessed.

Processor switches into privileged mode in response to interrupt . . .

. . . and out of privileged mode when resuming the program.

Three Types of Interrupts.

- *Trap:*
  Something like a subroutine call to the operating system, initiated by executing a type of instruction called a trap.

- *Exception:*
  Something went wrong (not necessarily an error) in the execution of an instruction.

  Exception has both a general and this specific meaning.

- *Hardware Interrupt:*
  Something outside the CPU is trying to get the computer's attention.

  Interrupt has both a general and this specific meaning.

Traps

Definitions

Trap:
(1) An instruction intended for user programs that transfers control to the operating system (privileged code).
(2) The execution of such an instruction.

Sort of a subroutine call to OS.

Trap causes branch to OS code and a switch to *privileged mode*.

Privileged Mode: a.k.a. *System Mode* and *Supervisor Mode*
A processor mode in which there are fewer restrictions on instruction execution.

Some instructions can only be executed in privileged mode.

When in privileged mode a *trap handler* is executed to service request.

Trap Handler:

A program, running in privileged mode that responds to a trap.

Traps typically used for I/O, memory allocation, etc.

Examples

Example, SPARC V8 trap instruction:

`ta` ⟨**rs1**⟩,⟨**imm**⟩.

ISA has a *trap base register (TBR)* that is used to construct the trap address.

Trap handler starts in *trap table*, each entry holds first four instructions of trap handler.

Trap Address Construction:

OS initializes TBR with upper 20 bits of trap table base.

When, say, `ta r1,3` executed, bits 4-10 set to low seven bits of `r1+3`.

Low four bits of TBR always zero.

Example: Using trap for read on Solaris (Sun OS)

```
System Calls                                                  read(2)


NAME
     read, readv, pread - read from file


SYNOPSIS
     #include <unistd.h>
     ssize_t read(int fildes, void *buf, size_t nbyte);


DESCRIPTION
     The read() function attempts to read nbyte  bytes  from  the
     file  associated with the open file descriptor, fildes, into
     the buffer pointed to by buf.
```

```
! Read System Call.
! Parameters placed in %o0, %o1, %o2.
!  %o0:  File descriptor (fildes)
!  %o1:  Buffer pointer (buf, address where read data copied to).
!  %o2:  Number of bytes to read. (nbyte)

mov  SYS_read, %g1    ! Argument for trap.
ta   %g0, ST_SYSCALL ! Call trap.
```

Some Solaris Trap Codes

```
/* Copyright (c) 1989 by Sun Microsystems, Inc. */
/* Software traps (ticc instructions). */
/* In sys/trap.h */
#define ST_OSYSCALL             0x00
#define ST_BREAKPOINT           0x01
#define ST_SYSCALL              0x08
#define ST_GETCC                0x20  // Move condition code to reg.
#define ST_SETCC                0x21  // Move condition code from reg.


/* In sys/syscall.h */
#define SYS_syscall     0
#define SYS_exit        1
#define SYS_fork        2
#define SYS_read        3
#define SYS_write       4
#define SYS_open        5
#define SYS_close       6
```

```
mov  SYS_read, %g1   ! Argument for trap.
ta   %g0, ST_SYSCALL ! Call trap.
```

Exceptions

Exception:

An interruption in normal execution triggered by an instruction that could not complete execution.

An exception occurs when an instruction cannot fully execute.

Faulting Instruction:

Instruction that caused an exception.

Some Exception Causes

○ Unrecognized opcode.

○ Division by zero.

○ Insufficient privilege level for instruction.

○ Insufficient privilege level for memory address.

○ Access to misaligned memory address. (Bus Error)

○ Access to illegal memory address. (Segmentation Fault)

○ Memory system needs updating. (TLB miss, page fault, etc.)

In response to an exception, handler may:

◇ Fix the problem and re-try the faulting instruction.

◇ Fix the problem and skip the faulting instruction.

◇ Call a program-supplied cleanup routine and then terminate the program.

Exception Example

Exception in ME stage of `lw`, indicated with an asterisk.

```
# Cycle:                    0  1  2  3  4  5 ...              99 100 ...
        add  r1, r2, r3   IF ID EX ME WB
        lw   r6, 0(r1)       IF ID EX ME*x                    IF ID
        sub  r5, r6, r7         IF ID EXx                        IF
        xor  r10, r11, r12         IF IDx
        and  r20, r21, r22            IFx
...
Handler:
        sw ...                                IF ...
        ...
        eret (exception return)                    IF ID EX ME WB
```

Cycle 4: `lw` raises a page-fault exception, ...

... `lw` and following instructions are squashed, ...

... and address of handler routine computed and sent to `PC`.

Cycle 5: handler fetched.

When handler returns at 99, execution resumes with `lw` (its second try).

Another Exception Example

Two faults, ID stage of ant and ME stage of `lw`, indicated asterisks.

```
# Cycle:                     0  1  2  3  4  5 ...          99 100 ...
        add   r1, r2, r3   IF ID EX ME WB
        lw    r6, 0(r1)       IF ID EX ME*x              IF ID EX ME WB
        ant   r5, r6, r7         IF ID*EXx                  IF ID*EX MEx
        xor   r10, r11, r12         IF IDx                     IF ID EXx
        and   r20, r21, r22            IFx                        IF IDx
...
Handler:
        sw ...                                IF ...                      IF ...
        ...
        eret  (exception return)              IF ID EX ME WB
```
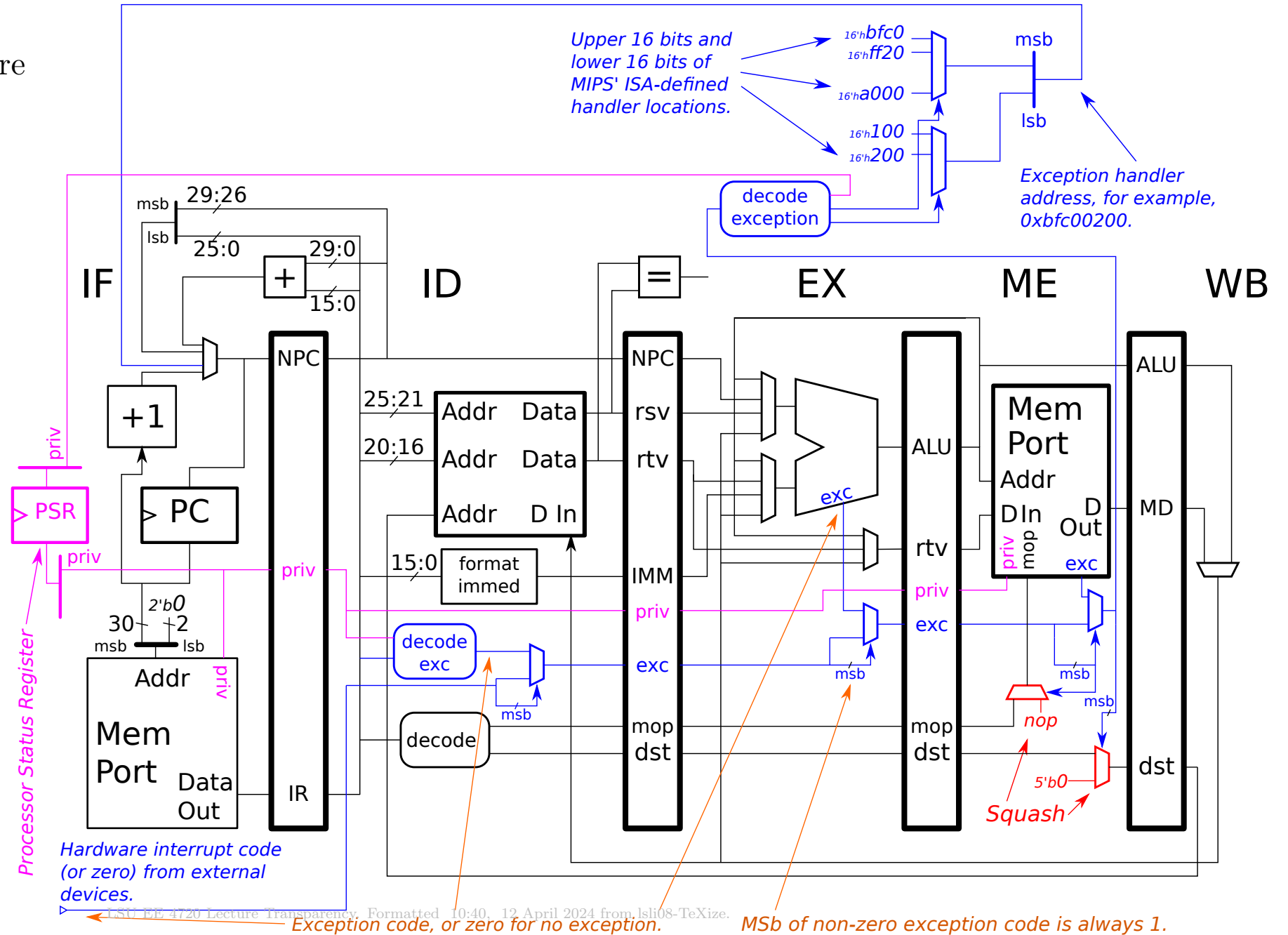
Cycle 3: `ant` raises an illegal instruction exception.

Cycle 4: `lw` raises a page-fault exception.

Hardware checks for exceptions only in ME to preserve program order and so `lw` is handled first.

When handler returns at 99, execution resumes with `lw` (its second try).

Possible MIPS Interrupt Hardware

IF   ID   EX   ME   WB

Upper 16 bits and lower 16 bits of MIPS' ISA-defined handler locations.

16'h bfc0
16'h ff20
16'h a000
16'h 100
16'h 200

Exception handler address, for example, 0xbfc00200.

Processor Status Register

Hardware interrupt code (or zero) from external devices.

Exception code, or zero for no exception.

MSb of non-zero exception code is always 1.

Squash

Hardware Interrupt

Caused by an external event . . .
. . . which typically needs routine attention.

For example,

- Disk drive has data that was requested 20 ms ago.

- User pressed a key on the keyboard.

- User sneezed, causing mouse to move.

- Timer (used by the OS as an alarm clock) expired.

Hardware Interrupt Example

```
add  r1, r2, r3   IF  ID  EX  ME  WB
sub  r4, r5, r6       IF  ID  EX  ME  WB
xor  r7, r8, r9           IF  ID  EX  ME  WB
```

As execution reaches code above, *achoooo* (user sneezes) . . .

. . . moving mouse, triggering an interrupt.

Based on time of sneeze, hardware completes `add` and `sub` . . .

. . . but squashes `xor` (for now).

The handler starts . . .

. . . the screen pointer (the little arrow) is moved . . .

. . . the handler finishes . . .

. . . and execution resumes with `xor`.

Interrupt Hardware Design

Goals and Difficulties

Goals

Exceptions: Handle in program order.

All Interrupts: Ability to resume execution as though nothing happened.

Precise Exception: Must be able to re-start or skip faulting instruction. . .
. . . and so last instruction must immediately precede faulting instruction.

Difficulties

Precise exceptions are easy to implement for integer pipeline . . .
. . . because exceptions can be detected before later instructions write.

Much harder with floating point operations (covered in next set).

Actions Initiated by Interrupt

Hardware executes up to and including last instruction . . .
. . . and squashes all following instructions.

• Type and timing of interrupt determine a *last* instruction.

• The last and all preceding instructions allowed to complete . . .
. . . instructions following last are squashed (nullified).

• Address of handler is determined . . .
. . . based on the type of interrupt and exception.

• Processor jumps to handler (OS code) and switches to privileged mode.

• Handler attends to interrupt.

• If appropriate, state is restored and program resumes . . .
. . . with the instruction following last.

The Last Instruction

The last and all preceding instructions must execute completely ...

... while instructions following the last must have no effect at all ...

... even though they may have already started when the interrupt occurred.

These *squashed* instructions will be executed after the handler completes.

Choice of Last Instruction

Choice depends on type of interrupt.

• For traps, the trap instruction itself is last.

  No problem.

• For hardware interrupts, a convenient last instruction can be chosen.

  No problem again.

• Precise exception, the instruction preceding faulting instruction.

  Problem: exception can occur in any of several stages.

  Problem: more than one instruction can raise exception.

  Problem: an instruction can raise exception before its *predecessor*.

  Problem: despite problems, some exceptions must be precise.

Squashing Instructions

When an interrupt occurs instructions may be squashed.

When the handler finishes and the program resumes ...
it must be as though the squashed instructions never even started.

So ...
... they cannot write registers ...
... they cannot write memory ...
... or set any kind of condition codes ...
... *unless* ...
... the state change can be un-done.

Squashing Instructions in MIPS

In the 5-stage MIPS implementation it's easy to squash integer instructions . . .

. . . because they only change state in the last two stages (ME & WB).

To squash an instruction in the IF, ID, or EX stages . . .

. . . the opcode is replaced with a NOP . . .

. . . or any control bits that initiate a memory or register write . . .

. . . are set to perform no action.

An instruction in WB cannot easily be squashed . . .

. . . because the following instruction, in ME, . . .

. . . would already be changing state.

Fortunately, there's never a need to squash an instruction in WB . . .

. . . although an already-squashed instruction can enter WB.

An instruction in ME cannot be squashed . . .

. . . *unless* the memory operation fails . . .

. . . which, luckily, is the only reason to squash the instruction.

Implementing Exceptions in MIPS

Each stage has hardware to detect exceptions that occur in that stage.

Each exception has a code, 0 used for no exception.

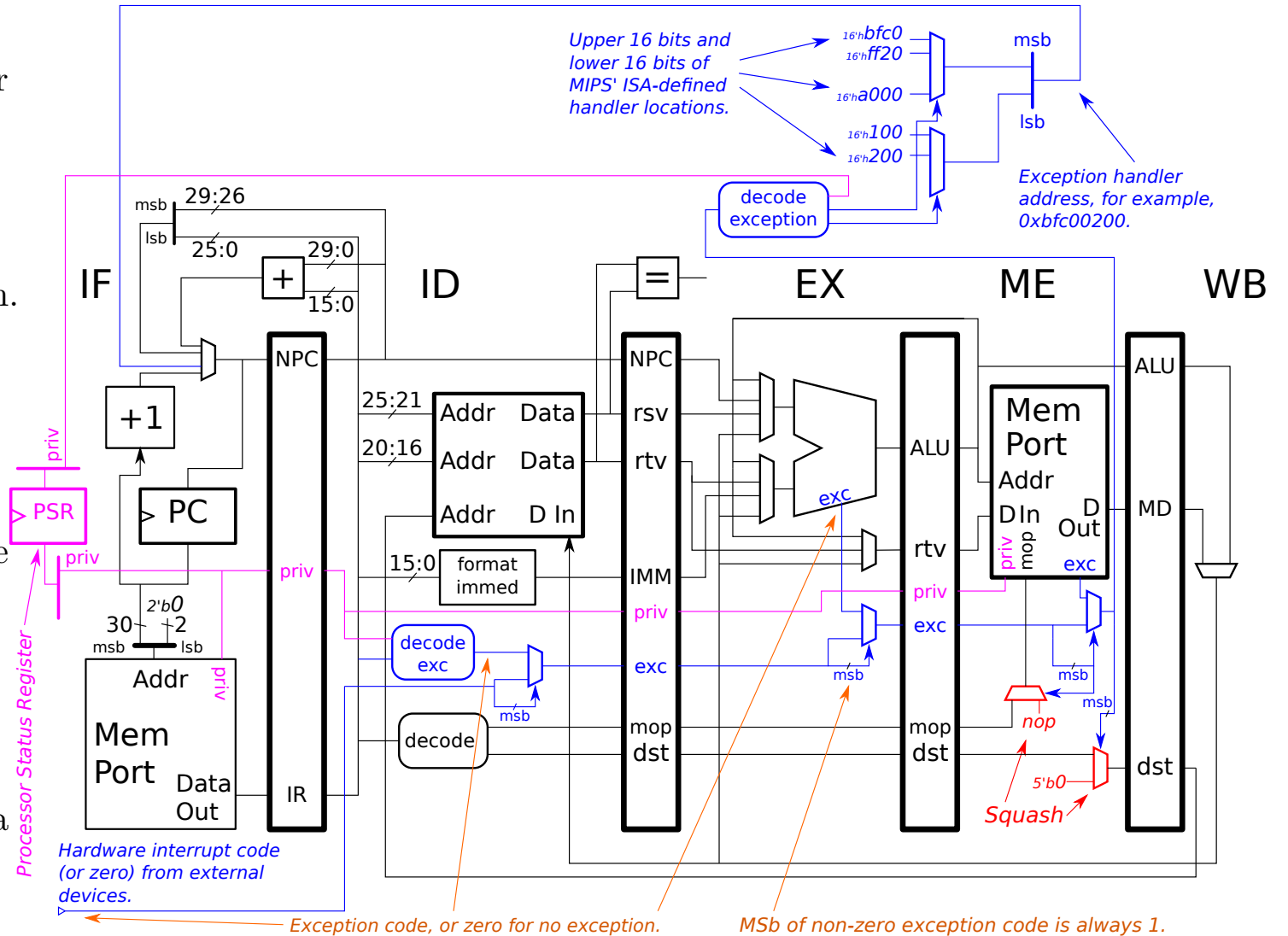Code is carried with instruction, using `exc` pipeline latch.

Codes arriving from earlier stages take priority.

In the `ME` stage the code is examined, and if non-zero:

A handler address is constructed and placed in the `PC`.

The faulting instruction is squashed (shown), …
… in `IF`, `ID`, `EX` squashed (not shown).

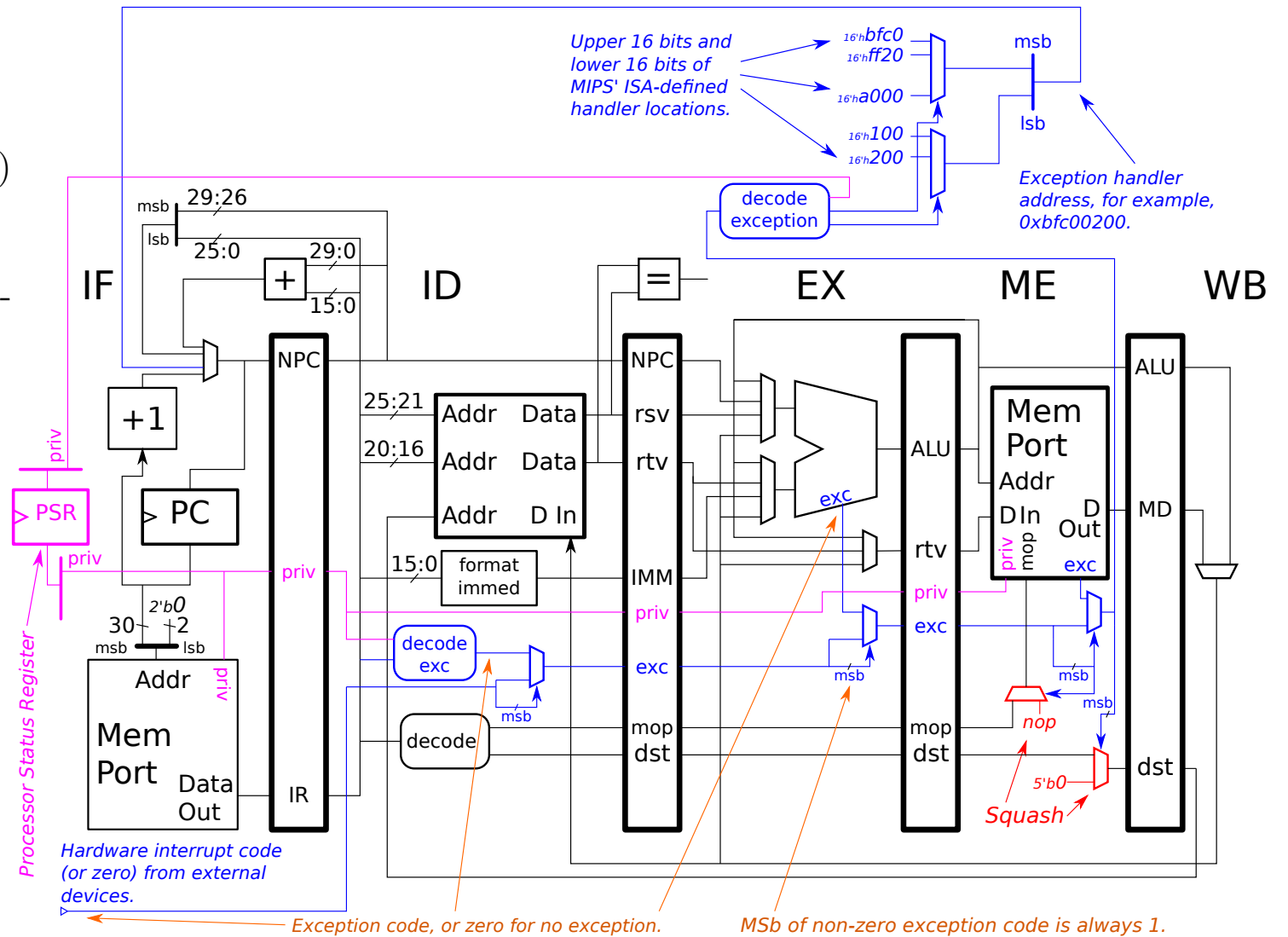Processor changed to privileged mode by setting a bit in the `PSR`.

Actions not shown in diagram

Exception code (`exc`) written to *Cause* register.

Faulting `PC` and `NPC` written to *EPC* (exception PC) register.

Memory address used for Mem Port to *BadVAddr* register.

Precise Exceptions

Future Me: There is a detailed solution in the 2020 Final Exam Problem 4d.

Precise Exception:
An exception for which when the handler is called all of the instructions before the faulting instruction finish and none of the instructions after finish.

Deferred Exception:
An exception for which when the handler is called all of the instructions before the faulting instruction finish and a few consecutive instructions after the faulting instruction finish.

Resuming of Interrupted Program

After handler finishes it may **resume** or **terminate** program.

With **precise exceptions** handler can resume **at** or **after** faulting instruction.

With deferred exceptions handler can only resume long after faulting instruction.

Resuming of Program for Precise Exceptions

If an exception is **precise**, handler **can return** to faulting instruction (thereby giving it a second chance).

Handler also **has option to** return to instruction **after** faulting instruction . . .
. . . this is done to emulate instructions that are not implemented in hardware.

Resuming of Program for Deferred Exceptions

Cannot return to faulting instruction if it raises a deferred exception. . .
. . . because instructions after faulting instruction would be executed twice!

Need for Precise Exceptions

Precise exceptions are necessary for some instructions ...
... and expensive for others.

They are **necessary** for instructions ...
... such as memory **loads and stores**.

For other instructions they are a **convenience** ...
... for example **FP instructions** ...
... that can write error values instead of numbers ...
... if they don't complete.

In many systems precise exceptions are optional for floating point ...
... but always provided for other instructions.