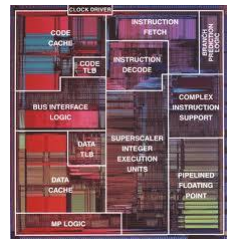
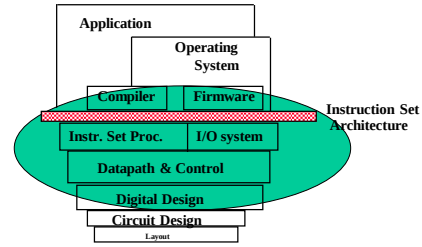
UTD

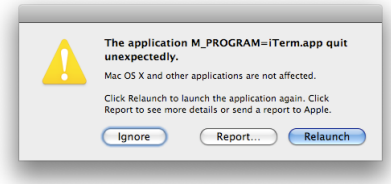
CS/SE 3340
Computer Architecture

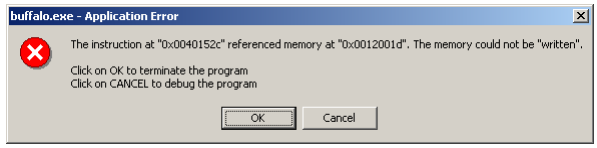


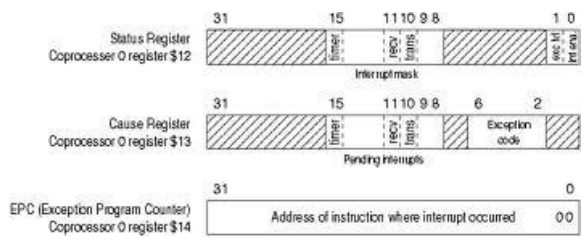


Exception Handling

Adapted from slides by Profs. D. Patterson and J. Hennessey







Exceptions: Traps & Interrupts

- *"Unexpected"* events requiring *change in flow of control*
  - Different ISAs use the terms differently
- *Internal exception* (or *trap*)
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, …
- *External exception (*or *interrupt)*
  - From an external source (I/O controller, timer,…)
- These pose a challenge for *pipelined* processors
  - Dealing with them without sacrificing performance is hard!

3

Example Traps

- Arithmetic overflow trap (exception code 12) is raised when the following user code is executed

```
li   $t0, 0x7FFFFFFF
addi $t0, $t0, 100
```

- A **break** (exception code 9) trap is generated by a user break statement

```
break 0
```

- What trap is generated for following instructions?

```
li   $t0, 0x0010FF00
lw   $s0, 1($t0)
```

4

Interrupt Examples

- *I/O interrupt*
  - Keyboard pressed, mouse moved events
  - Sensor events (e.g. accelerometer in smartphones)
- *Timer interrupt*
  - An event generated when a timer expires
    - Timer use: bounded waiting on some event
  - e.g. timers for process scheduling in OSes, in communication systems

5

Some MIPS Exception Codes

Code	Name	Description
0	INT	Interrupt (external)
4	ADDRL	Load from an illegal address
5	ADDRS	Store to an illegal address
6	IBUS	Bus error on instruction fetch
7	DBUS	Bus error on data reference
8	SYSCALL	**`syscall`** instruction executed
9	BKPT	**`break`** instruction executed
10	RI	Reserved or unimplemented instruction
12	OVF	Arithmetic overflow

6

Handling Exceptions

- When an exception occurs, the processor suspends what it was doing and handles the exception
  - Apply to both types of exceptions: traps and interrupts
- Processor control is transferred to a special program called the *exception handler* to handle the exception
  - Written explicitly to deal with exceptions
  - Typically run in kernel mode (vs. user mode)
- The exception handler is typically stored at a location predefined by hardware
- Control is transferred to the exception handler upon the occurrence of an exception

7

Handling Exceptions – MIPS

- *Hardware*
  - copies PC-4 into EPC ($14) on *coprocessor* 0
  - puts correct exception code into cause registers ($13)
  - sets PC to 0x80000180, executes
  - enters kernel mode
- *Software (the exception handler)*
  - checks cause (bits 6-2 of register $13)
  - handles the exception accordingly to the exception code

8

Handler Actions

- Read cause, and transfer to relevant handler subroutine for each cause
- Determine action required
- If *restartable* (i.e. the program can continue)
  - Take required action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …

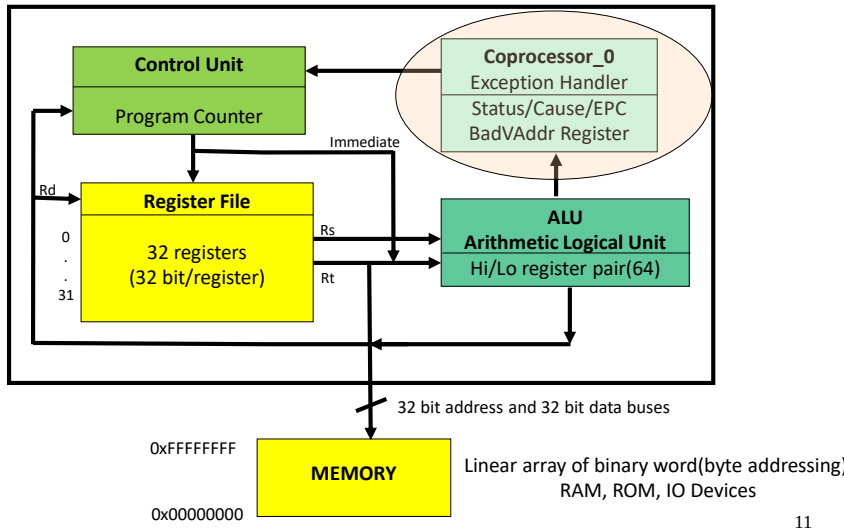
9

Exception Properties

- *Restartable* exceptions
  - Exception handler runs, then returns to the victim instruction
    - Refetched and executed from scratch
    - Corrective action taken by the exception handler
- PC saved in EPC register
  - Identifies exception causing instruction
  - Exception handler needs to compensate when returning from exception handling

10

MIPS Coprocessor_0

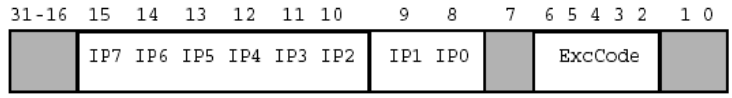


Coprocessor_0 Registers

Number	Name	Usage
8	BadVAddr	Invalid memory address referenced by the instruction.
12	Status	Interrupt mask, enable bits, and status when exception occurred.
13	Cause	Type of exception and pending interrupt bits.
14	EPC	Address of the offending instruction, i.e. where the exception occurs

- Exception handler uses information in these register to handle an exception

Cause Register - $13



- Provides information about the cause of the exception (trap or interrupt) and what external (IP7 to IP2) or internal (IP1 and IP0) interrupts are pending
- The exception code is stored as an unsigned integer using bits 6-2

13

Exception Code

- Exception handler can extract exception code from cause register ($13)

```
mfc0 $a0, $13   # $13 is cause register
srl  $a0, $a0, 2
andi $a0, $a0, 31 # $a0=exception code
```

- How to determine whether the exception is a trap or an interrupt?
  - If all interrupt pending bits (IP0-7) are not set the exception is a trap, otherwise it is an interrupt
  - Or if exception code = 0 the exception is an interrupt!

14

Interrupt Pending Bits – IP0-IP7

- Indicate that an interrupt has occurred
- Bit IPi is set if an interrupt has occurred at level i and is pending (has not been handled yet)
- IP2-7 are for external hardware interrupts, IP0-1 are for internal software (syscall, break) interrupts
- There can be multiple interrupts pending
  - Multiple IP bits are set
  - The exception handler must prioritize their handling

15

Status Register - $12

- Bit 0: interrupt enable (IE) bit
  - Interrupt is disable if this bit is set to 0
- Bits 3-4: indicates the CPU mode
  - 0 = kernel, 1 = supervisor, 2 = user (newer MIPS)
- Bits 8-15: interrupt mask (IM0 to IM7)
  - The CPU can indicate what interrupt it can handle by setting these bits
  - IMi = 1 means CPU can handle interrupt I
  - For example it can mask lower priority interrupts while servicing a pending interrupt

16

Note: the MIPS green sheet swaps description of several fields of $12 and $13, which is *incorrect*.

BadVAddr Register - $8

- Bad Virtual Address register contain the invalid memory address that produces an exception on a memory access violation (error)
- There are various causes for a memory access error
  - Attempt to access an address outside of the program address space. E.g. a user program attempts to access 0x80000000 or higher (reserved for kernel mode)
  - Trying to access address 0
  - Load or store a word (32-bit) from an address that is not word aligned
  - Write to a read-only address

17

MIPS Exception Handler

```
# An Example (incomplete) MIPS exception handler
        .kdata
_k_save_t0:     .word   0       # $t0
_k_save_t1:     .word   0       # $t1
#  . . . more register storage declared here as needed
# Jump Table: each entry is the address of the handler to run when the
# corresponding exception occurs (as dictated by the value in ExcCode)
        .align  2
_k_JumpTable:
        .word   _k_HandleInt    # External Interrupt            [0]
        …
        .word   _k_HandleAdEL   # Addr error exception (load )  [4]
        .word   _k_HandleAdES   # Addr error exception (store)  [5]
        …
        .word   _k_HandleOvf    # Arithmetic overflow exception [12]
# . . .
```

18

MIPS Exception Handler – cont'd

```
        .ktext
# Save all of the registers that are used by the kernel.
        sw      $t0, _k_save_t0
        sw      $t1, _k_save_t1
        mfc0    $k0, $13                # read the Cause register of CP_0
        andi    $k0, $k0, 0x7C          # note the position of Exc Code!
        lw      $k0, _k_JumpTable($k0)  # a clever trick is used here!
        jr      $k0
_k_IncReturn:  # used only for traps, where the PC must still be update
        mfc0    $k1, $14
        addi    $k1, $k1, 4
        mtc0    $k1, $14
_k_Return:   #  Restore all of the registers used by the kernel (except for $1)
        lw      $8, _k_save_t0
        lw      $9, _k_save_t1
# Get EPC, return from exception handling and  got back to user code
        mfc0    $k0, $14
        eret
```

19

MIPS Exception Handler – cont'd

```
# Procedure:    _k_HandleOvf
# Description:  This handler reports integer overflow exceptions
#               and go the  next instruction.
# Returns:      none  (exits kernel by jumping to _k_IncReturn)
        .kdata
_k_HOvf_msg:  .asciiz "Integer Overflow exception!"
        .ktext
_k_HandleOvf:
        la      $a0, _k_HOvf_msg
        jal     _k_Warn
        j       _k_IncReturn
```

20

MIPS Exception Handler – cont'd

```
# Procedure:        _k_HandleInt
# Description:      All interrupts are processed through this routine.
# Returns:          none  (exits kernel by jumping to _k_Return)
            .kdata
_k_HandleInt:
#  Prepare for interrupt processing (e.g. masking further interrupts),
#  determine which interrupt needs to be serviced,
#  call the corresponding interrupt service routine
            ....
            ....
# then exits kernel by jumping to _k_Return
# the EPC contains the PC that points to the interrupted inst.
            j           _k_Return
```

21

Summary

- There are situations where the CPU needs to change control of flow involuntarily due to exceptions
  - Traps (internal) and interrupt (external)
- Exceptions are handle by both hardware and software (exception handler)
  - In MIPS coprocessor_0 is responsible for exception handling
  - It has several special registers for this purpose
- *An exception handler* uses information in these register to handle both traps and interrupts

22