Chapter 8:

Interrupts and Exceptions

Topics:

Definitions
Trap handler behavior and traces
Hardware support for interrupts/exceptions
Supporting a multitasking operating system

Reading: P&H Appendix B B-33 to B-38, 4.9

Definitions and basic concepts

Interrupts and exceptions:
* "unusual" events encountered by CPU

When CPU detects an interrupt or exception:
* interrupt execution of current program
* service the interrupt/exception

Common terminology:

interrupts: events external to CPU

examples: ex.: user input, input/output devices

exceptions: events internal to CPU

examples: error conditions, ex. array out of bounds, segmentation fault (memory faults), divide by zero

Both also called *traps* or *signals*.

How does the CPU service the trap?

When CPU detects trap, it jumps to a special system routine called the *trap handler*.
(alternate names: `exception handler`
`interrupt service routine, interrupt handler`

Like an automatically triggered function call!

What does the trap handler do?
Depends on what type of interrupt/exception occurred, and how hardware is set.

Minor events: print a message, resume execution of program (default for SPIM)

Major errors: print error message, core dump

Example: arithmetic overflow
(in http://unixlab.sfsu.edu/~whsu/csc256/PROGS/8.1)

```
        .data
str:    .asciiz "...Now we're past
exception...\n"

        .text

main:
        li      $s0,0x7fffffff
        add     $s1,$s0,1
        li      $v0,4
        la      $a0,str
        syscall
        li      $v0,10
        syscall
```

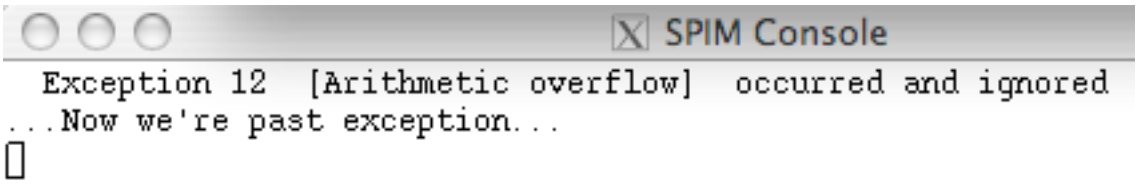
Sample run (`-quiet` turns off warning messages):

```
libra% spim -quiet
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /afs/sfsu.edu/f1/whsu/spim-8.0/CPU/exceptions.s
(spim) lo "8.1"
(spim) run
  Exception 12  [Arithmetic overflow] occurred and
ignored
...Now we're past exception...
(spim)
```

exception handler

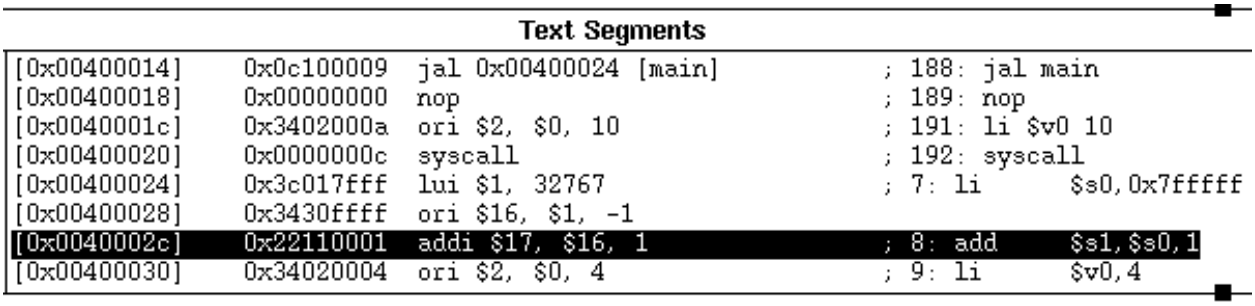
user program (8.1)

Or in xspim:



Note: user code does not check for overflow! When overflow occurs, CPU detects arithmetic overflow exception.

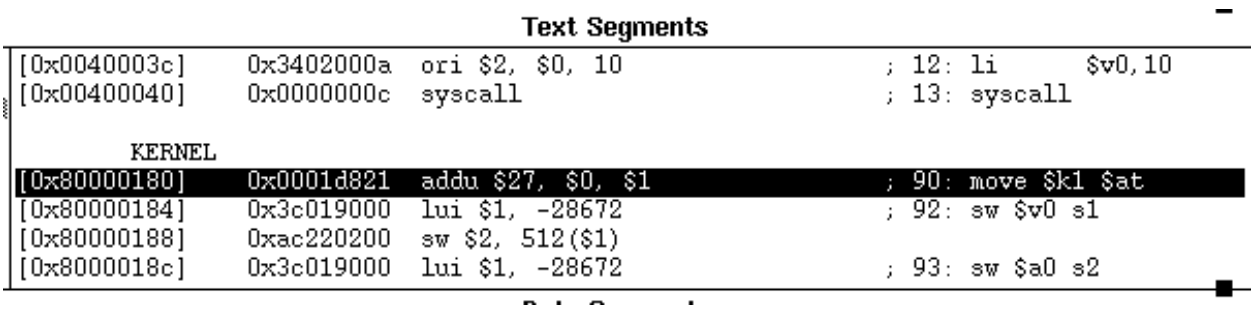
If we step through the program instead, we see that the trap handler is automatically called.
Before the add with overflow:



Text Segments

```
[0x00400014]    0x0c100009  jal 0x00400024 [main]          ; 188: jal main
[0x00400018]    0x00000000  nop                            ; 189: nop
[0x0040001c]    0x3402000a  ori $2, $0, 10                 ; 191: li $v0 10
[0x00400020]    0x0000000c  syscall                        ; 192: syscall
[0x00400024]    0x3c017fff  lui $1, 32767                  ; 7: li      $s0,0x7fffff
[0x00400028]    0x3430ffff  ori $16, $1, -1
[0x0040002c]    0x22110001  addi $17, $16, 1               ; 8: add     $s1,$s0,1
[0x00400030]    0x34020004  ori $2, $0, 4                  ; 9: li      $v0,4
```

After the add with overflow:



Text Segments

```
[0x0040003c]    0x3402000a  ori $2, $0, 10                 ; 12: li      $v0,10
[0x00400040]    0x0000000c  syscall                        ; 13: syscall

        KERNEL
[0x80000180]    0x0001d821  addu $27, $0, $1               ; 90: move $k1 $at
[0x80000184]    0x3c019000  lui $1, -28672                 ; 92: sw $v0 s1
[0x80000188]    0xac220200  sw $2, 512($1)
[0x8000018c]    0x3c019000  lui $1, -28672                 ; 93: sw $a0 s2
```

SPIM's trap handler only does one thing:

```
prints message about exception
```

Then return to user program (print "...past exception..." message).
(code for arithmetic overflow: 12)

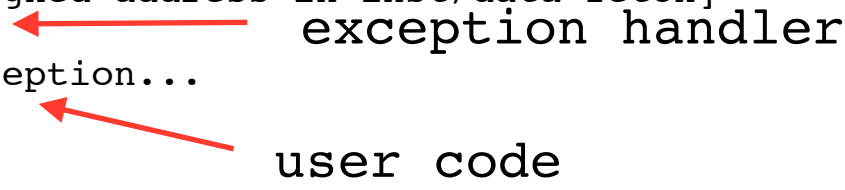
Another example: unaligned load
(in http://unixlab.sfsu.edu/~whsu/csc256/PROGS/8.2)

```
        .data
x:      .word   4
str:    .asciiz "...Now we're past
exception...\n"

        .text
main:
      la      $s0,x
      lw      $s1,1($s0)
      li      $v0,4
      la      $a0,str
      syscall
      li      $v0,10
      syscall
```

Sample run:

```
libra% spim -quiet
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /afs/sfsu.edu/f1/whsu/spim-8.0/CPU/exceptions.s
(spim) lo "8.2"
(spim) run
  Exception 4  [Unaligned address in inst/data fetch]
occurred and ignored
...Now we're past exception...
(spim)
```



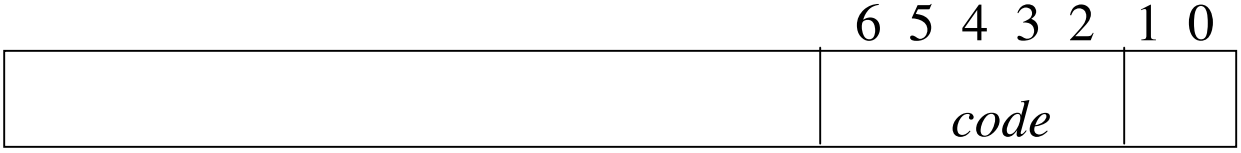
When unaligned load address is detected, CPU automatically calls trap handler.

(code for bad address in load or instruction fetch: 4)

When an exception is detected, MIPS hardware automatically places exception code in a special register called the *Cause* register.

Trap handler checks Cause register to determine type of event that caused exception.

Cause format (Figure B.7.2):
exception code in bits 6-2 (5 bits)



MIPS Cause register is coprocessor 0's $13
(different from MIPS general register $13!)

[Coprocessor 0 is part of MIPS CPU for handing interrupts/exceptions.
What is Coprocessor 1?]

```
floating point co-processor
```

To move Cause register into any MIPS reg R:
mfc0 R,$13 `move from coprocessor 0`
To move any MIPS reg R into Cause register:
mtc0 $13, R `move to coprocessor 0`

More MIPS exception codes: Appendix B, B-35

Trap handler also needs to know where the faulting instruction is.

New register: *exception program counter (EPC)*

EPC is $14 in coprocessor 0.

List of coprocessor 0 registers: Appendix B, B-33

In SPIM, the file spim-8.0/CPU/exception.s is automatically loaded into memory when SPIM is invoked:

```
libra% spim -quiet
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /afs/sfsu.edu/f1/whsu/spim-8.0/CPU/exceptions.s
(spim)
```

exceptions.s contains:
1) default trap handler at 0x80000180

2) `MIPS kernel`

starts at __start (0x400000)

Main operations:
1) `sets up runtime environment`

2) `calls main!`

Scroll down and find __start; program starts running here:

```
            .text
            .globl __start
__start:
```

1) set up environment and parameters.
2) call main.

```
        lw $a0 0($sp)              # argc
        addiu $a1 $sp 4            # argv
        addiu $a2 $a1 4            # envp
        sll $v0 $a0 2
        addu $a2 $a2 $v0
        jal main
```

Back to top of exceptions.s: list of error messages.

Then:
.ktext 0x80000180: start of trap handler

Trap handler operations:

1) save $a0, $v0 (for printing messages)
2) save Cause, Exception PC

```
.set noat
move $k1 $at    # Save $at
.set at
sw $v0 s1       # Not re-entrant and
                # we can't trust $sp
sw $a0 s2       # But we need to use
                # these registers

mfc0 $k0 $13    # Cause register
srl $a0 $k0 2   # Extract ExcCode Field
andi $a0 $a0 0x1f

# Print information about exception.
#

li $v0 4        # syscall 4 (print_str)
la $a0 __m1_
syscall

# etc etc…
```

The system trap handler exceptions.s is not much different from all the MIPS code you've seen before.
The main differences:
- It occupies specific memory locations
- It is called automatically on an exception

In spim/xspim, it's possible to customize the trap handler to do somewhat different things.
Usual procedure:
1) make a copy of exceptions.s
2) make changes/customizations, save as
   newEx.s
3) start spim without loading exceptions.s:
   `spim –notrap`
4) load newEx.s
5) load user program
6) run!

By changing the trap handler, we can change system behavior when we have exceptions.
[See lab exercise for changing exception.s]

C++ and Java allow users to have some control over exception handling with try/catch blocks.

Example:

```
try {

    Statement 1
    Statement 2
    Statement 3

}

catch (exception type a) {

    user code for handling exception type a

}
catch (exception type b) {

    user code for handling exception type b

}
```

In Unix, interrupt and exception behavior can be changed using the signals library (signal.h).

Default behavior:
(http://unixlab.sfsu.edu/~whsu/csc256/PROGS/int0.c)

```
/*****************************************/
/* int0.c: found in ~whsu/310/PROGS/int0.c */
/*                                       */
/* default case:                         */
/*    program goes into infinite loop    */
/*    ctrl-c interrupts and aborts program */
/*                                       */
/*****************************************/


#include <stdio.h>

int main()
{

  printf("Entering infinite loop...\n");

  while (1) {
    printf("In while loop\n");
    sleep(1);
  }
  printf("End of program\n"); /* this line never
reached */

}

libra% gcc int0.c -o int0
libra% int0
Entering infinite loop...
In while loop
In while loop
```

```
In while loop
In while loop
^Clibra% ^D
```

Rewrite to ignore Control-C.

(http://unixlab.sfsu.edu/~whsu/csc256/PROGS/int1.c)

```
/*******************************************/
/* int1.c: found in ~whsu/310/PROGS/int1.c */
/*                                         */
/* interrupt is masked:                    */
/*    signal(SIGINT,SIG_IGN) causes        */
/*         to be ignored                   */
/*    program goes into infinite loop      */
/*    ctrl-c does not affect program       */
/*    (stop program with ctrl-z and kill)  */
/*                                         */
/*******************************************/

#include <stdio.h>
#include <signal.h>

int main()
{
  signal(SIGINT,SIG_IGN); /* mask or ignore ctr-c
interrupt/signal */

  printf("Entering infinite loop...\n");

  while (1) {
    printf("In while loop\n");
    sleep(1);
  }
  printf("End of program\n"); /* this line never
reached */

}
```

Control-C is ignored. Trace:

```
libra% gcc int1.c -o int1
libra% int1
Entering infinite loop...
In while loop
In while loop
In while loop
In while loop
^CIn while loop
In while loop
In while loop
In while loop
^CIn while loop
In while loop
In while loop
In while loop
^Z
Stopped (user)
libra% kill %1
libra%
[1]    Terminated              int1
```

User-defined signal handler.

(http://unixlab.sfsu.edu/~whsu/csc256/PROGS/int2.c)

```
/*****************************************/
/* int2.c: found in ~whsu/310/PROGS/int2.c */
/*                                       */
/* interrupt is detected                 */
/* user-defined interrupt handler triggered*/
/*    signal(SIGINT,intHandler) installs   */
/*         user-defined interrupt handler  */
/*    program goes into infinite loop      */
/*    ctrl-c causes intHandler() to be     */
/*      called, instead of system routine  */
/*                                       */
/*****************************************/

#include <stdio.h>
#include <signal.h>

void intHandler();

int main()
{

  signal(SIGINT,intHandler); /* install interrupt
handler */

  printf("Entering infinite loop...\n");

  while (1) {
    printf("In while loop\n");
    sleep(1);
  }
  printf("End of program\n"); /* this line never
reached */

}
```

```
void intHandler()
{

  printf("Interrupt received and ignored\n");
  signal(SIGINT,intHandler); /* reinstall interrupt
handler */
}
```

User-defined signal handler. Trace:

```
libra% gcc int2.c -o int2
libra% int2
Entering infinite loop...
In while loop
In while loop
In while loop
^CInterrupt received; in user routine
In while loop
In while loop
In while loop
^CInterrupt received; in user routine
In while loop
In while loop
In while loop
In while loop
^Z
Stopped (user)
libra% kill %1
libra%
```

Hardware support for exceptions/interrupts

Goal: Extend MIPS hardware implementation discussed earlier to support interrupts and exceptions.

Add two new registers:

Cause

Exception PC (EPC)

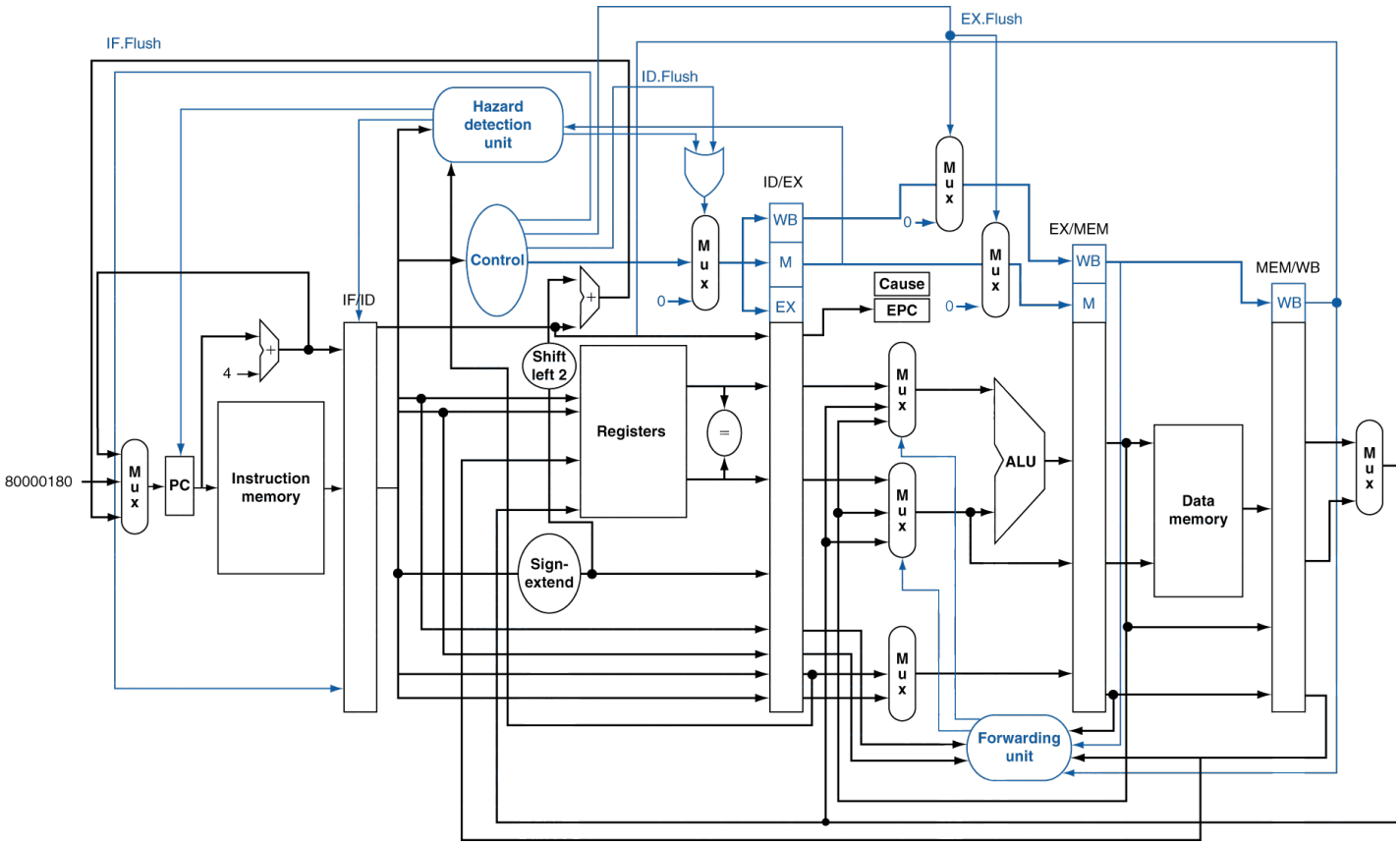
Actions taken by hardware:

1) `Cause = exception code`

2) `EPC = address of faulting instr`

3) `PC = address of trap handler`

Datapath changes (basic idea only)



Supporting a multitasking operating system

Multi-tasking is implemented using timer interrupts.

Suppose two processes are running simultaneously on a system with one CPU. How does this work?

Process 0:

[code not shown]

```
add $12, $14, $12
bne $12, $18,…
```

Process 1:

[code not shown]

```
lw $12, 4($29)
sub $20, $20, $12
…
```

CPU is *timeshared* between processes.

A *timer interrupt* is triggered by hardware periodically (can be set by system administrator).

In MIPS, two coprocessor 0 registers

count (coprocessor 0 $9):
incremented periodically by hardware

compare (coprocessor 0 $11):
when count == compare, timer interrupt occurs

When timer interrupt occurs, jump to system trap handler.

Typical scenario:

Process 0 executes for some time
Timer interrupt occurs
Trap handler saves info for Process 0
Trap handler restores info for Process 1

Process 1 executes for some time
Timer interrupt occurs
Trap handler saves info for Process 1
Trap handler restores info for Process 0

Process 0 executes for some time etc