

System Calls



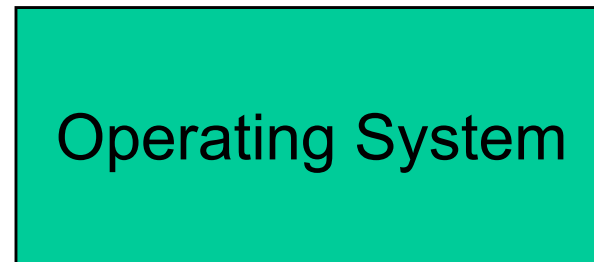
Learning Outcomes

- A high-level understanding of System Calls
 - Mostly from the user's perspective
 - From textbook (section 1.6)
- Exposure architectural details of the MIPS R3000
 - Detailed understanding of the of exception handling mechanism
 - From “Hardware Guide” on class web site
- Understanding of the existence of compiler function calling conventions
 - Including details of the MIPS ‘C’ compiler calling convention
- Understanding of how the application kernel boundary is crossed with system calls in general
 - Including an appreciation of the relationship between a case study (OS/161 system call handling) and the general case.



Operating System System Calls

Kernel Level



Requests
(System Calls)



User Level



System Calls

- Can be viewed as special procedure calls
 - Provides for a controlled entry into the kernel
 - While in kernel, they perform a privileged operation
 - Returns to original caller with the result
- The system call interface represents the abstract machine provided by the operating system.



A Brief Overview of Classes System Calls

- From the user's perspective
 - Process Management
 - File I/O
 - Directories management
 - Some other selected Calls
 - There are many more
 - On Linux, see `man syscalls` for a list



Some System Calls For Process Management

Process management

| Call | Description |
|--|--|
| <code>pid = fork()</code> | Create a child process identical to the parent |
| <code>pid = waitpid(pid, &statloc, options)</code> | Wait for a child to terminate |
| <code>s = execve(name, argv, environp)</code> | Replace a process' core image |
| <code>exit(status)</code> | Terminate process execution and return status |



Some System Calls For File Management

File management

| Call | Description |
|---|--|
| <code>fd = open(file, how, ...)</code> | Open a file for reading, writing or both |
| <code>s = close(fd)</code> | Close an open file |
| <code>n = read(fd, buffer, nbytes)</code> | Read data from a file into a buffer |
| <code>n = write(fd, buffer, nbytes)</code> | Write data from a buffer into a file |
| <code>position = lseek(fd, offset, whence)</code> | Move the file pointer |
| <code>s = stat(name, &buf)</code> | Get a file's status information |



Some System Calls For Directory Management

Directory and file system management

| Call | Description |
|--------------------------------|--|
| s = mkdir(name, mode) | Create a new directory |
| s = rmdir(name) | Remove an empty directory |
| s = link(name1, name2) | Create a new entry, name2, pointing to name1 |
| s = unlink(name) | Remove a directory entry |
| s = mount(special, name, flag) | Mount a file system |
| s = umount(special) | Unmount a file system |



Some System Calls For Miscellaneous Tasks

Miscellaneous

| Call | Description |
|--------------------------|---|
| s = chdir(dirname) | Change the working directory |
| s = chmod(name, mode) | Change a file's protection bits |
| s = kill(pid, signal) | Send a signal to a process |
| seconds = time(&seconds) | Get the elapsed time since Jan. 1, 1970 |



System Calls

- A stripped down shell:

```
while (TRUE) {                                     /* repeat forever */
    type_prompt( );                                /* display prompt */
    read_command (command, parameters)           /* input from terminal */

    if (fork() != 0) {                             /* fork off child process */
        /* Parent code */
        waitpid( -1, &status, 0);                /* wait for child to exit */
    } else {
        /* Child code */
        execve (command, parameters, 0);         /* execute command */
    }
}
```



System Calls

| UNIX | Win32 | Description |
|---------|---------------------|--|
| fork | CreateProcess | Create a new process |
| waitpid | WaitForSingleObject | Can wait for a process to exit |
| execve | (none) | CreateProcess = fork + execve |
| exit | ExitProcess | Terminate execution |
| open | CreateFile | Create a file or open an existing file |
| close | CloseHandle | Close a file |
| read | ReadFile | Read data from a file |
| write | WriteFile | Write data to a file |
| lseek | SetFilePointer | Move the file pointer |
| stat | GetFileAttributesEx | Get various file attributes |
| mkdir | CreateDirectory | Create a new directory |
| rmdir | RemoveDirectory | Remove an empty directory |
| link | (none) | Win32 does not support links |
| unlink | DeleteFile | Destroy an existing file |
| mount | (none) | Win32 does not support mount |
| umount | (none) | Win32 does not support mount |
| chdir | SetCurrentDirectory | Change the current working directory |
| chmod | (none) | Win32 does not support security (although NT does) |
| kill | (none) | Win32 does not support signals |
| time | GetLocalTime | Get the current time |



The MIPS R2000/R3000

- Before looking at system call mechanics in some detail, we need a basic understanding of the MIPS R3000



MIPS R3000

- Load/store architecture
 - No instructions that operate on memory except load and store
 - Simple load/stores to/from memory from/to registers
 - Store word: `sw r4, (r5)`
 - Store contents of r4 in memory using address contained in register r5
 - Load word: `lw r3, (r7)`
 - Load contents of memory into r3 using address contained in r7
 - Delay of one instruction after load before data available in destination register
 - » Must always be an instruction between a load from memory and the subsequent use of the register.
 - `lw, sw, lb, sb, lh, sh,`



MIPS R3000

- Arithmetic and logical operations are register to register operations
 - E.g., `add r3, r2, r1`
 - No arithmetic operations on memory
- Example
 - `add r3, r2, r1` $\Rightarrow r3 = r2 + r1$
- Some other instructions
 - `add, sub, and, or, xor, sll, srl`



MIPS R3000

- All instructions are encoded in 32-bit
- Some instructions have *immediate* operands
 - Immediate values are constants encoded in the instruction itself
 - Only 16-bit value
 - Examples
 - Add Immediate: `addi r2, r1, 2048`
⇒ $r2 = r1 + 2048$
 - Load Immediate : `li r2, 1234`
⇒ $r2 = 1234$

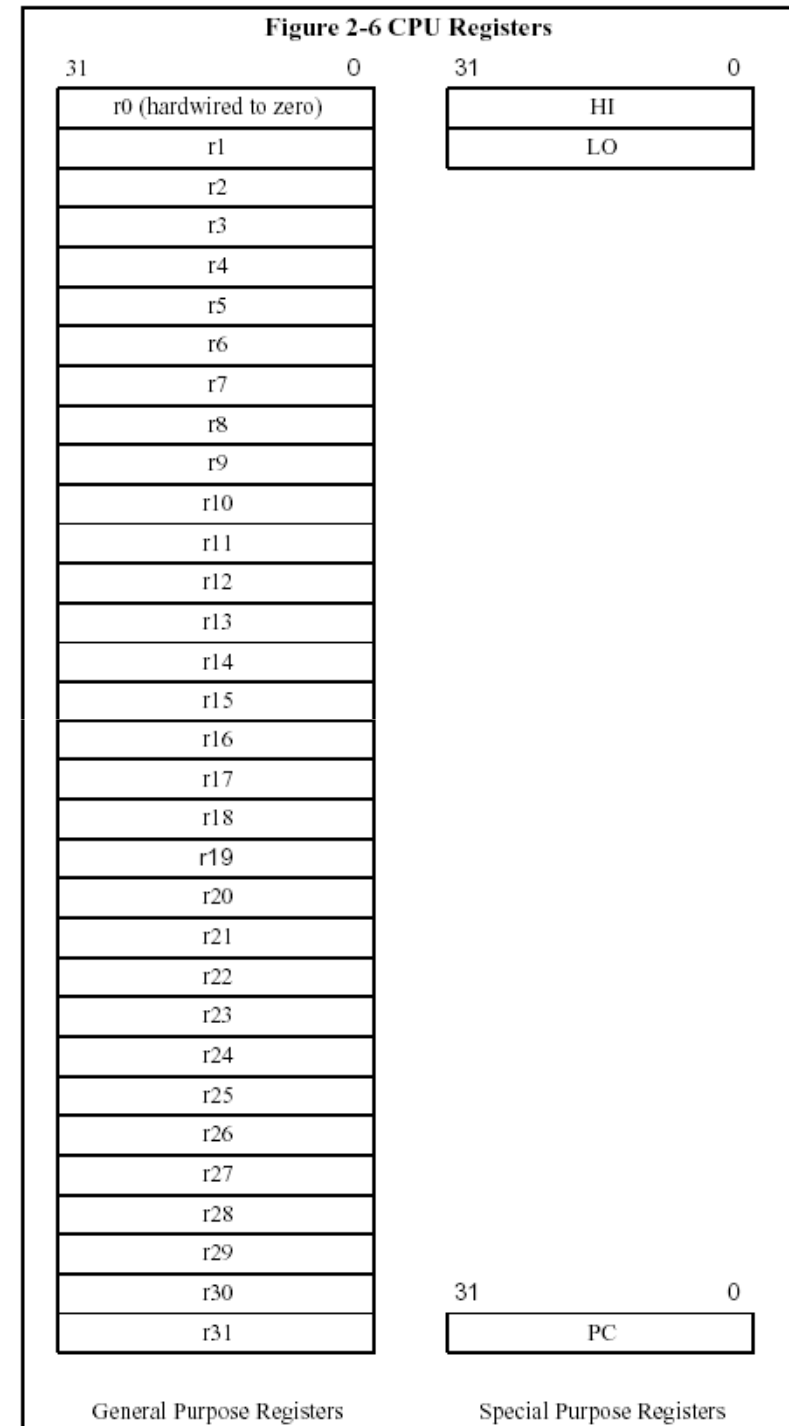


MIPS Registers

- User-mode accessible registers
 - 32 general purpose registers
 - r0 hardwired to zero
 - r31 the *link* register for jump-and-link (JAL) instruction
 - HI/LO
 - 2 * 32-bits for multiply and divide
 - PC
 - Not directly visible
 - Modified implicitly by jump and branch instructions



THE UNIVERSITY OF
NEW SOUTH WALES



Branching and Jumping

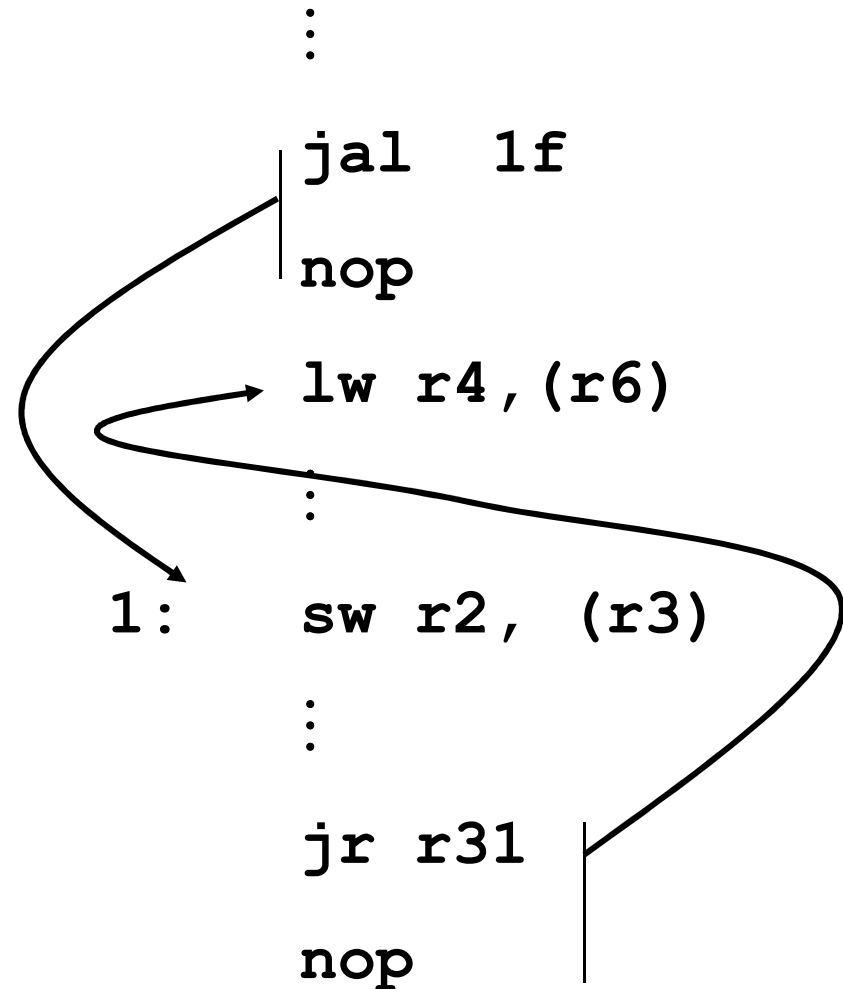
- Branching and jumping have a *branch delay slot*
 - The instruction following a branch or jump is always executed prior to destination

```
li    r2, 1
sw   r0, (r3)
j     1f
li    r2, 2
li    r2, 3
1:   sw   r2, (r3)
```



Jump and Link Instruction

- JAL is used to implement function calls
 - $r31 = PC+8$
- Return Address register (RA) is used to return from function call



MIPS R3000

- RISC architecture – 5 stage pipeline

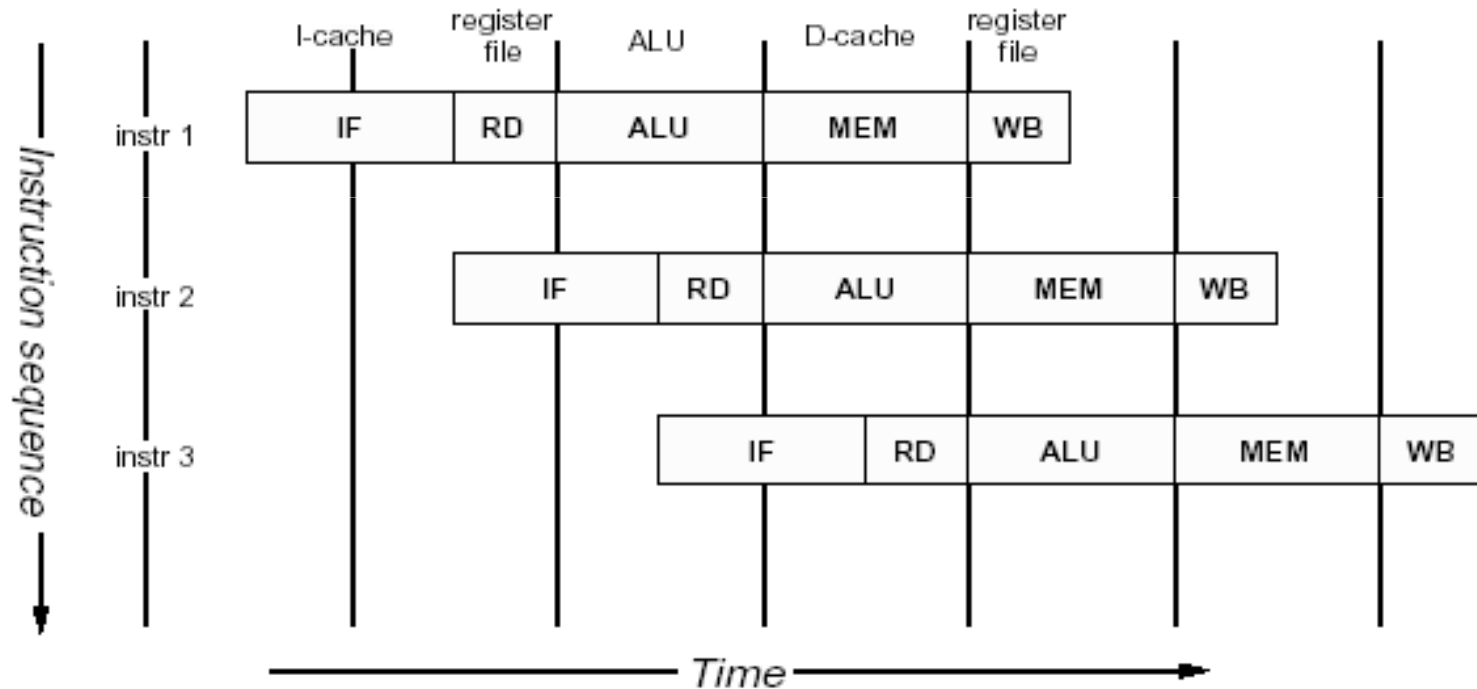


Figure 1.1. MIPS 5-stage pipeline



Coprocessor 0

- The processor control registers are located in CP0
 - Exception management registers
 - Translation management registers
- CP0 is manipulated using mtc0 (move to) and mfc0 (move from) instructions
 - mtc0/mfc0 are only accessible in kernel mode.



CP0 Registers

- Exception Management
 - c0_cause
 - Cause of the recent exception
 - c0_status
 - Current status of the CPU
 - c0_epc
 - Address of the instruction that caused the exception
 - c0_badvaddr
 - Address accessed that caused the exception
- Miscellaneous
 - c0_prid
 - Processor Identifier
- Memory Management
 - c0_index
 - c0_random
 - c0_entryhi
 - c0_entrylo
 - c0_context
 - More about these later in course



c0_status

| | | | | | | | | | | | | | | | |
|-----|-----|-----|-----|----|----|----|-----|----|-----|-----|-----|-----|-----|-----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| CU3 | CU2 | CU1 | CU0 | 0 | RE | 0 | BEV | TS | PE | CM | PZ | SwC | IsC | | |
| 15 | | | | | | | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | | | | | IM | | | 0 | KUo | IEo | KUp | IEp | KUc | IEc | |

Figure 3.2. Fields in status register (SR)

- For practical purposes, you can ignore most bits
 - Green background is the focus



c0_status

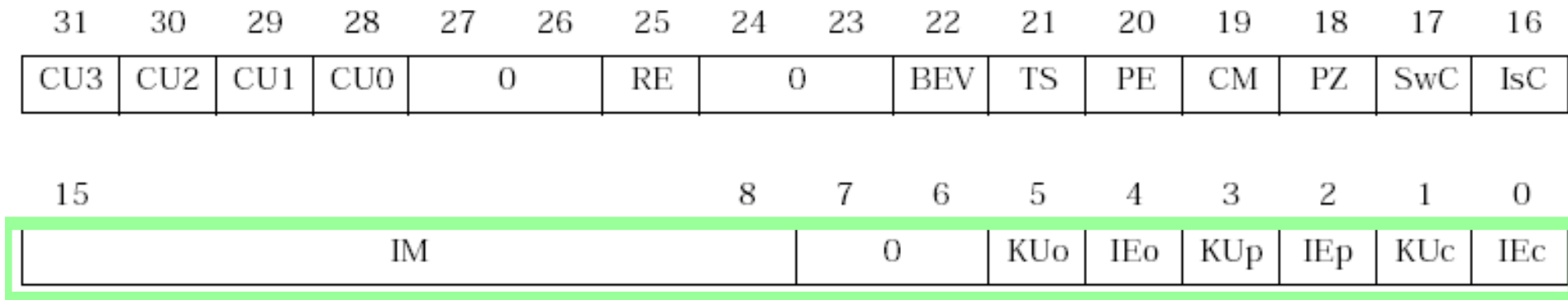


Figure 3.2. Fields in status register (SR)

- **IM**
 - Individual interrupt mask bits
 - 6 external
 - 2 software
- **KU**
 - 0 = kernel
 - 1 = user mode
- **IE**
 - 0 = all interrupts masked
 - 1 = interrupts enable
 - Mask determined via IM bits
- c, p, o = current, previous, old



c0_cause

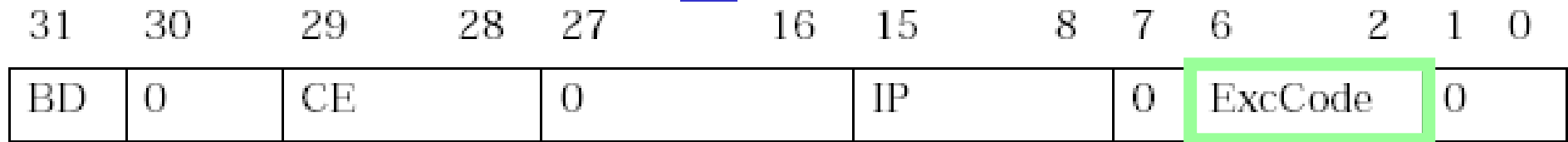


Figure 3.3. Fields in the Cause register

- IP
 - Interrupts pending
 - 8 bits indicating current state of interrupt lines
- CE
 - Coprocessor error
 - Attempt to access disabled Copro.
- BD
 - If set, the instruction that caused the exception was in a branch delay slot
- ExcCode
 - The code number of the exception taken



Exception Codes

| ExcCode Value | Mnemonic | Description |
|---------------|----------|---|
| 0 | Int | Interrupt |
| 1 | Mod | "TLB modification" |
| 2 | TLBL | "TLB load/TLB store" |
| 3 | TLBS | |
| 4 | AdEL | Address error (on load/I-fetch or store respectively). Either an attempt to access outside kuseg when in user mode, or an attempt to read a word or half-word at a misaligned address. |
| 5 | AdES | |

Table 3.2. ExcCode values: different kinds of exceptions



Exception Codes

| ExcCode Value | Mnemonic | Description |
|---------------|----------|---|
| 6 | IBE | Bus error (instruction fetch or data load, respectively). External hardware has signalled an error of some kind; proper exception handling is system-dependent. The R30xx family CPUs can't take a bus error on a store; the write buffer would make such an exception "imprecise". |
| 7 | DBE | |
| 8 | Syscall | Generated unconditionally by a <i>syscall</i> instruction. |
| 9 | Bp | Breakpoint - a <i>break</i> instruction. |
| 10 | RI | "reserved instruction" |
| 11 | CpU | "Co-Processor unusable" |
| 12 | Ov | "arithmetic overflow". Note that "unsigned" versions of instructions (e.g. <i>addu</i>) never cause this exception. |
| 13-31 | - | reserved. Some are already defined for MIPS CPUs such as the R6000 and R4xxx |

Table 3.2. ExcCode values: different kinds of exceptions



c0_epc

- The Exception Program Counter

- Points to address of where to restart execution after handling the exception or interrupt
- Example
 - Assume `sw r3, (r4)` causes a fault exception

`nop`

`sw r3 (r4)`

`nop`

`c0_epc`

Aside: We ignore BD-bit in `c0_cause` which is also used in reality on rare occasions.



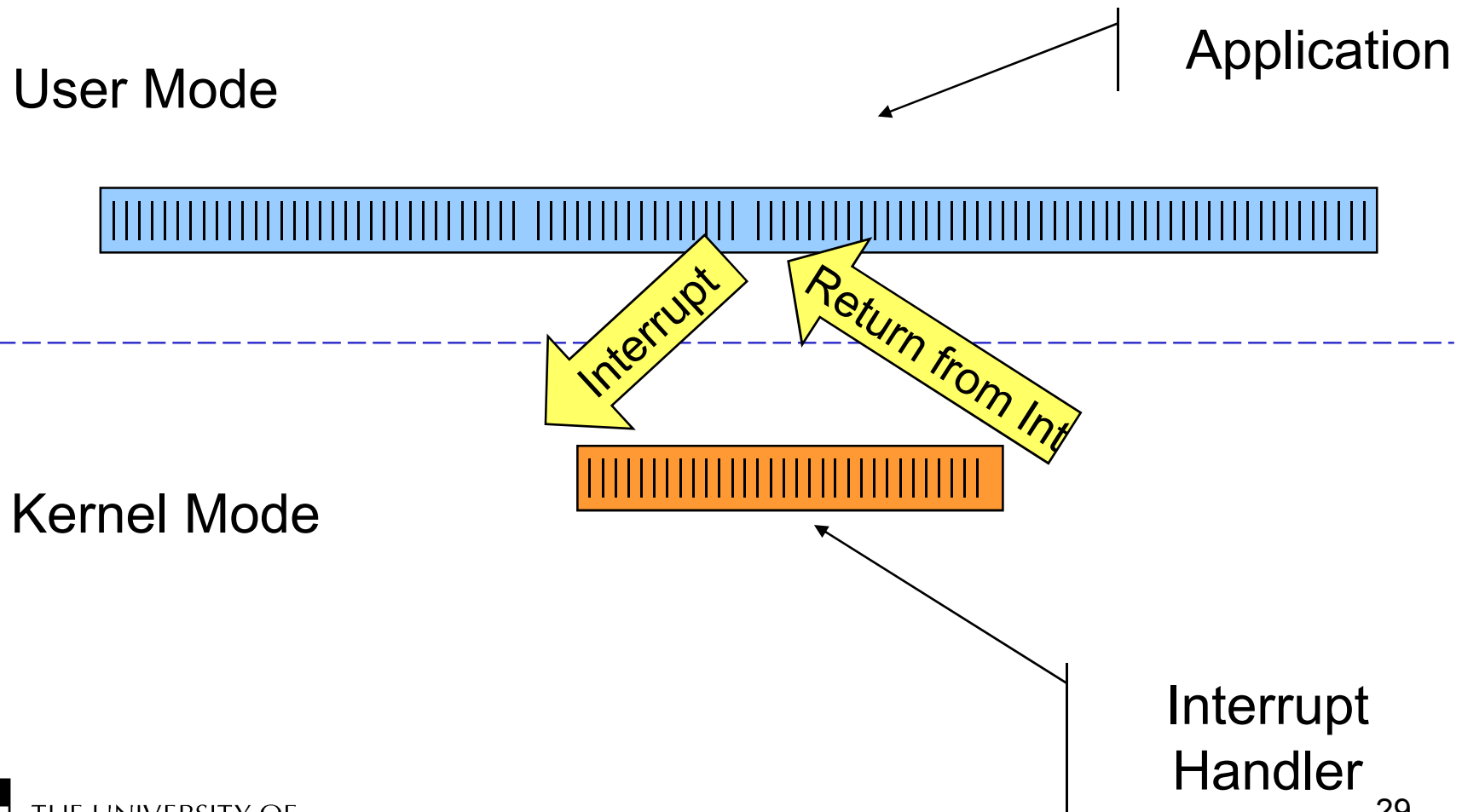
Exception Vectors

| Program address | “segment” | Physical Address | Description |
|-----------------|-----------|------------------|--|
| 0x8000 0000 | kseg0 | 0x0000 0000 | TLB miss on <i>kuseg</i> reference only. |
| 0x8000 0080 | kseg0 | 0x0000 0080 | All other exceptions. |
| 0xbf00 0100 | kseg1 | 0x1fc0 0100 | Uncached alternative <i>kuseg</i> TLB miss entry point (used if <i>SR</i> bit <i>BEV</i> set). |
| 0xbf00 0180 | kseg1 | 0x1fc0 0180 | Uncached alternative for all other exceptions, used if <i>SR</i> bit <i>BEV</i> set). |
| 0xbf00 0000 | kseg1 | 0x1fc0 0000 | The “reset exception”. |

Table 4.1. Reset and exception entry points (vectors) for R30xx family



Simple Interrupt Walk-through



Hardware exception handling

PC

0x12345678

EPC

?

- Let's now walk through an exception
 - Assume an interrupt occurred as the previous instruction completed
 - Note: We are in user mode with interrupts enabled

Cause

?

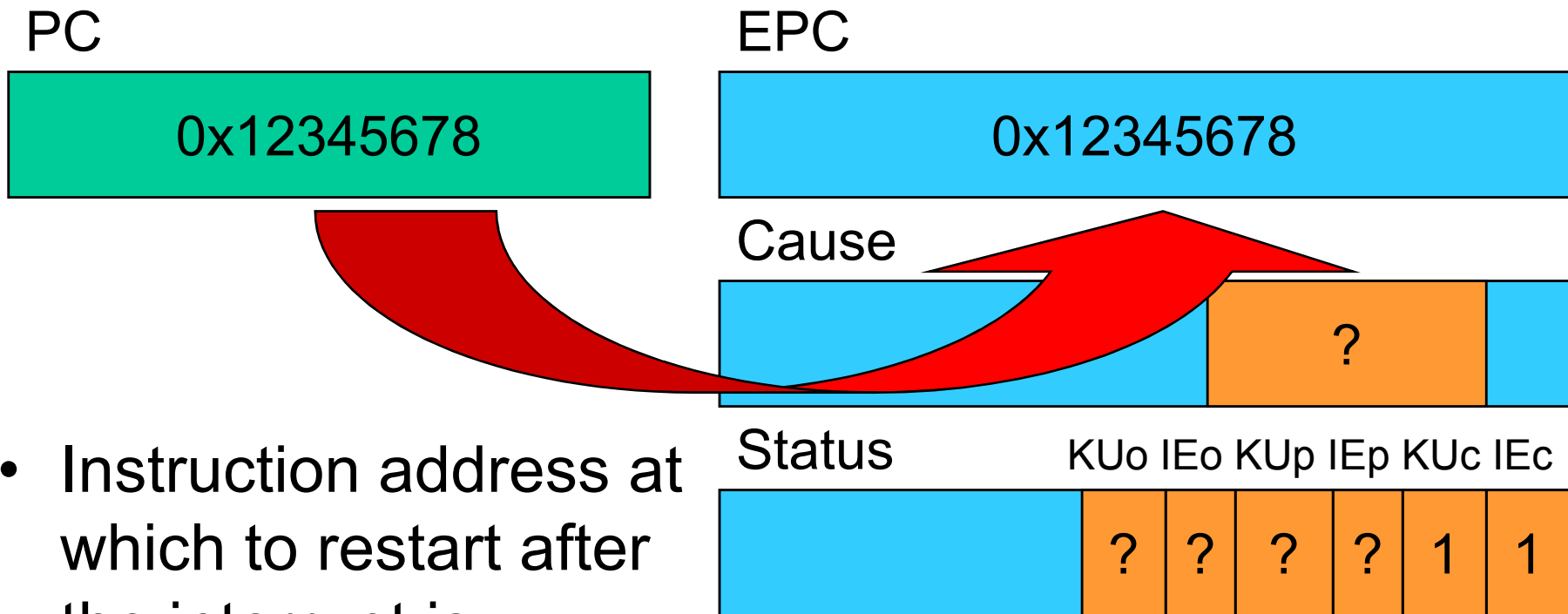
Status

KUo IEo KUp IEp KUc IEc

? ? ? ? 1 1



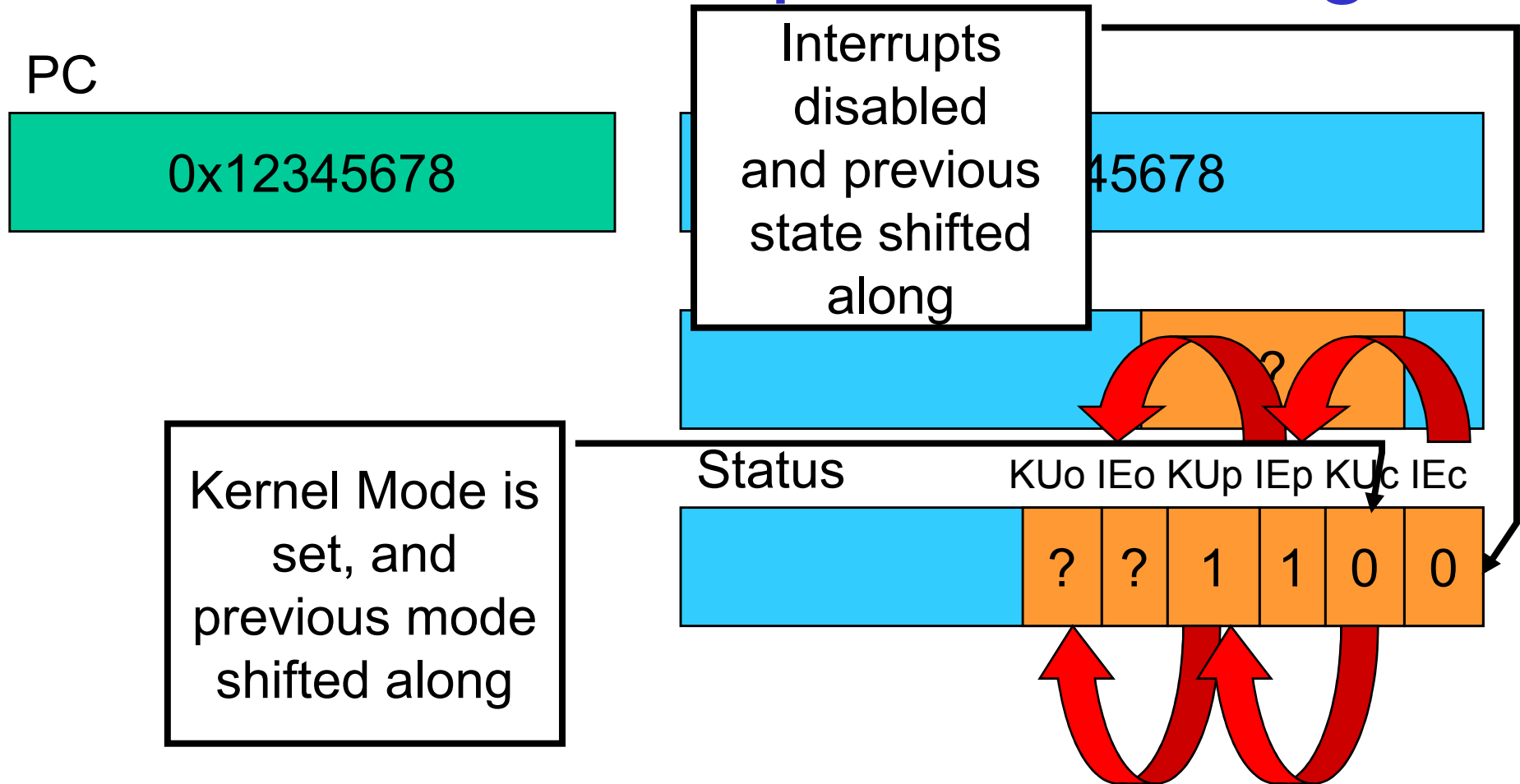
Hardware exception handling



- Instruction address at which to restart after the interrupt is transferred to EPC



Hardware exception handling



Hardware exception handling

PC

0x12345678

EPC

0x12345678

Cause

0

Status

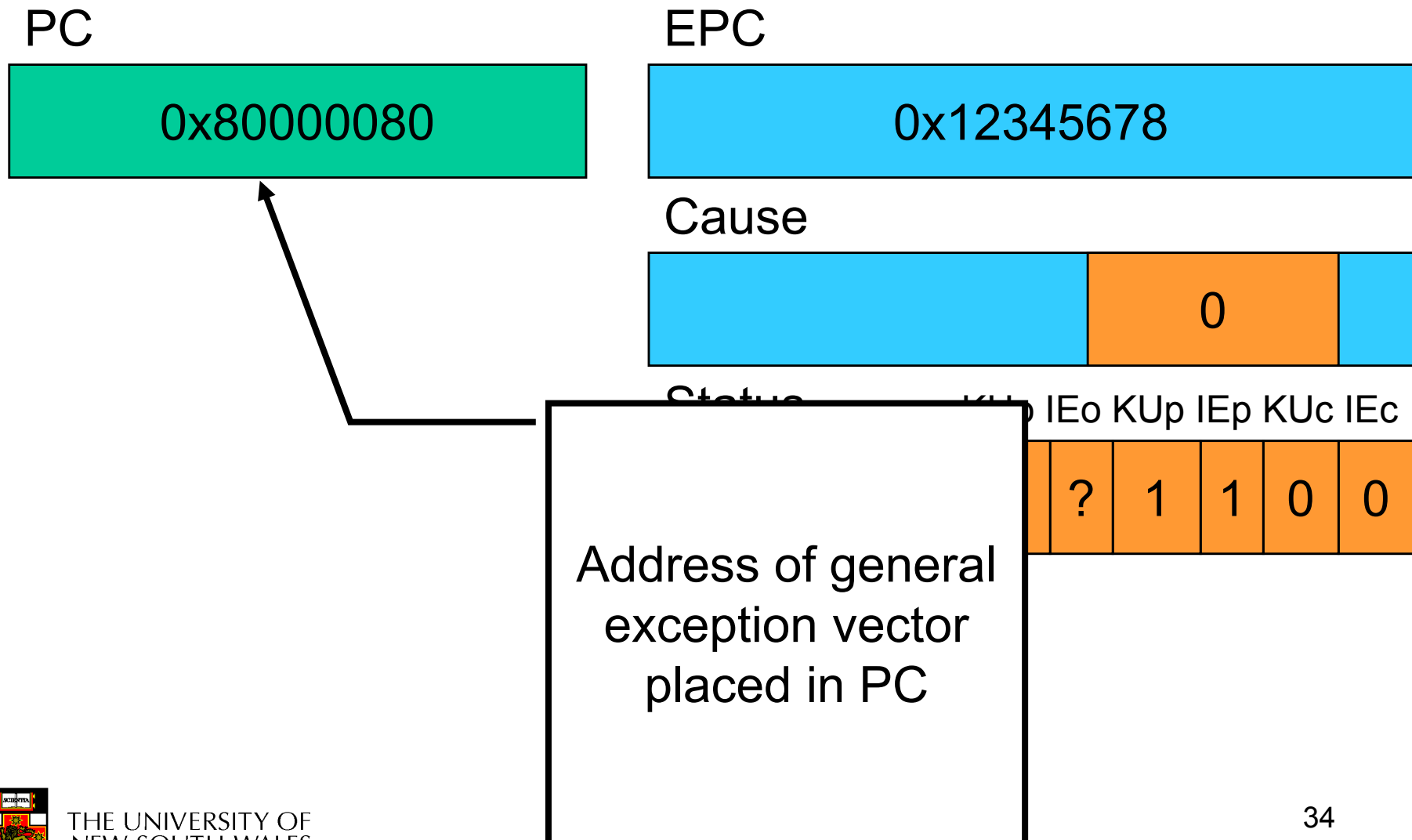
KUo IEo KUp IEp KUc IEc

? ? 1 1 0 0

Code for the exception placed in Cause. Note Interrupt code = 0



Hardware exception handling



Hardware exception handling

PC

0x80000080

- CPU is now running in kernel mode at 0x80000080, with interrupts disabled
- All information required to
 - Find out what caused the exception
 - Restart after exception handling

is in coprocessor registers

EPC

0x12345678

Cause

0

Status

KUo IEo KUp IEp KUc IEc

? ? 1 1 0 0

Badvaddr



Returning from an exception

- For now, lets ignore
 - how the exception is actually handled
 - how user-level registers are preserved
- Let's simply look at how we return from the exception



Returning from an exception

PC

0x80001234

EPC

0x12345678

- This code to return is

Cause

0

```
lw    r27, saved_epc
```

Status

KUo IEo KUp IEp KUc IEc

```
nop
```

```
jr    r27
```

```
rfe
```

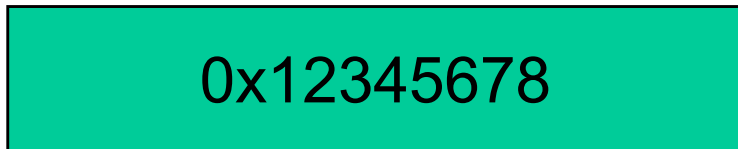
0

Load the contents of EPC which is usually moved earlier to somewhere in memory by the exception handler

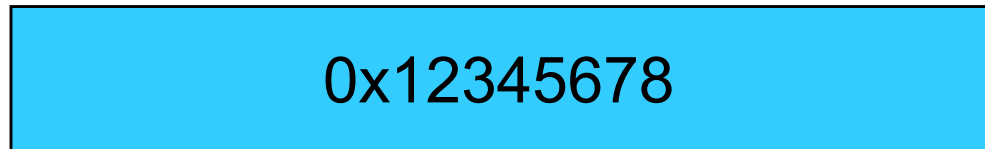


Returning from an exception

PC



EPC



- This code to return is

Cause



```
lw    r27, saved_epc
```

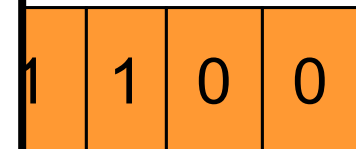
```
nop
```

```
jr    r27
```

```
rfe
```

Status

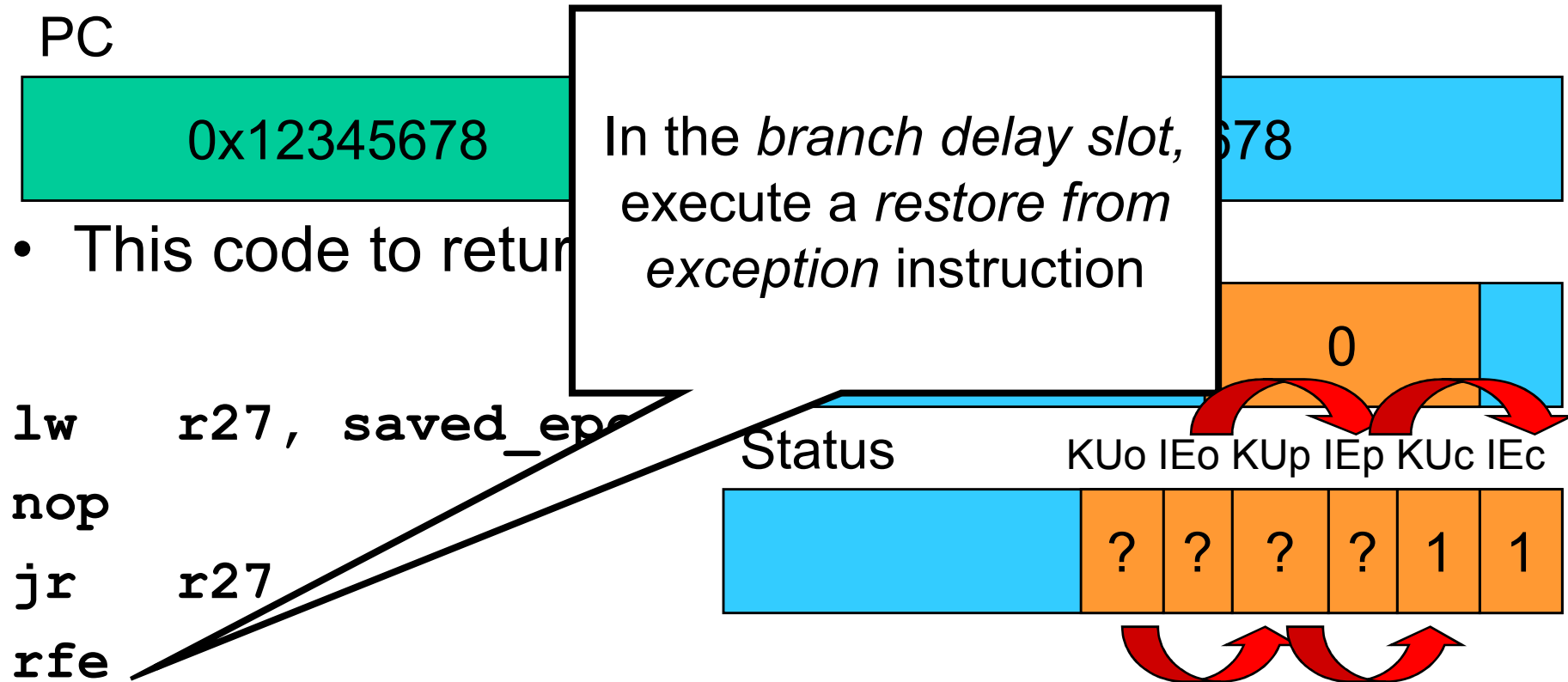
KUp IEo KUp IEp KUc IEc



Store the EPC back in the PC



Returning from an exception



Returning from an exception

PC

0x12345678

- We are now back in the same state we were in when the exception happened

EPC

0x12345678

Cause

0

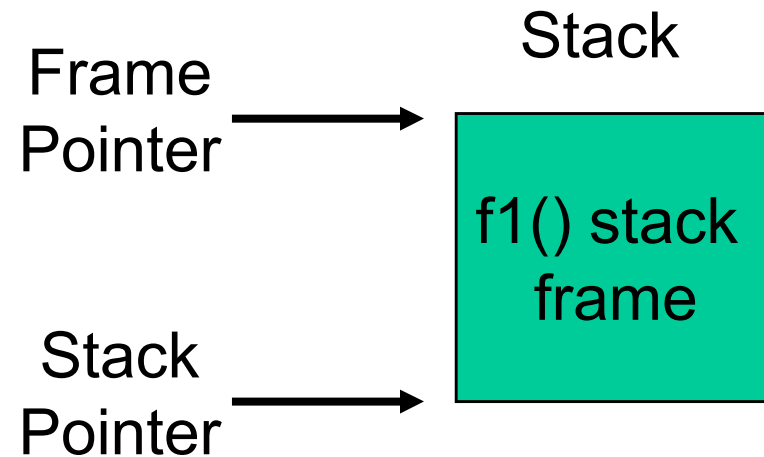
Status

| | KUo | IEo | KUp | IEp | KUc | IEc |
|--|-----|-----|-----|-----|-----|-----|
| | ? | ? | ? | ? | 1 | 1 |



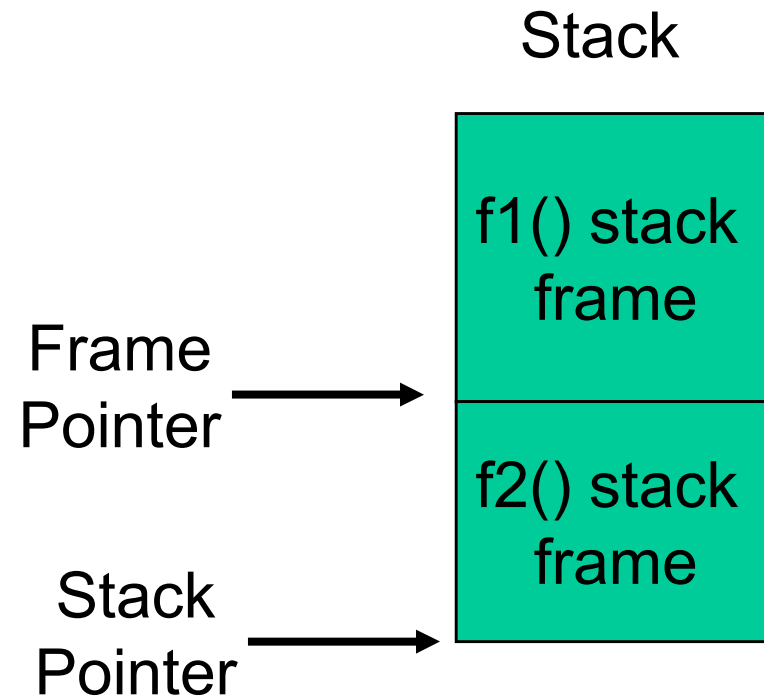
Function Stack Frames

- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
- Example: assume f1() calls f2(), which calls f3().



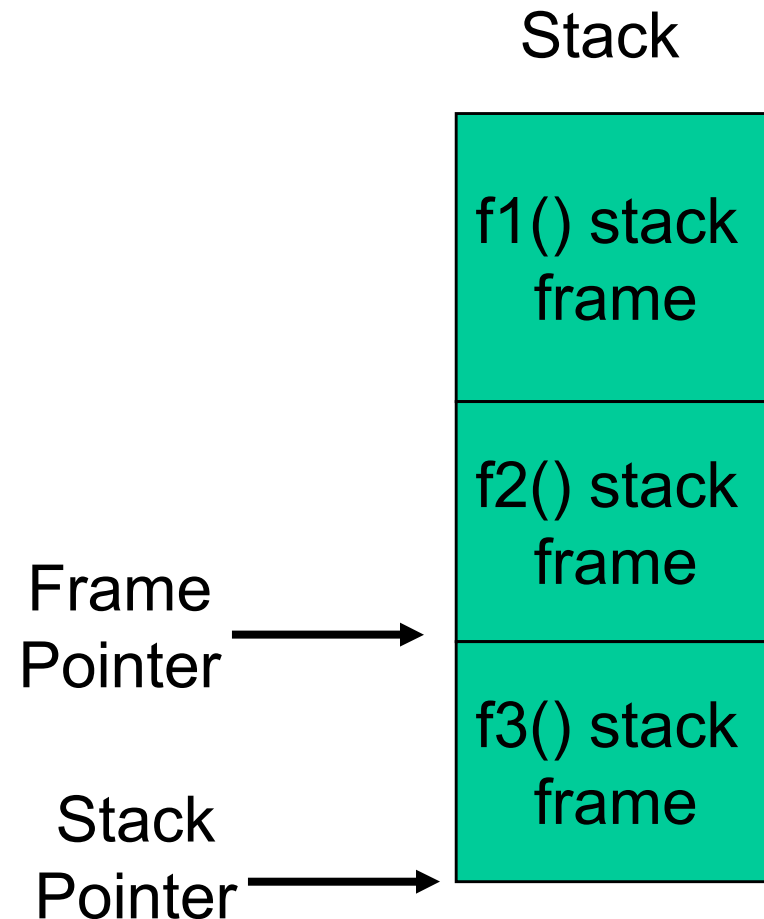
Function Stack Frames

- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
- Example: assume f1() calls f2(), which calls f3().



Function Stack Frames

- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
- Example: assume f1() calls f2(), which calls f3().



Software Register Conventions

- Given 32 registers, which registers are used for
 - Local variables?
 - Argument passing?
 - Function call results?
 - Stack Pointer?



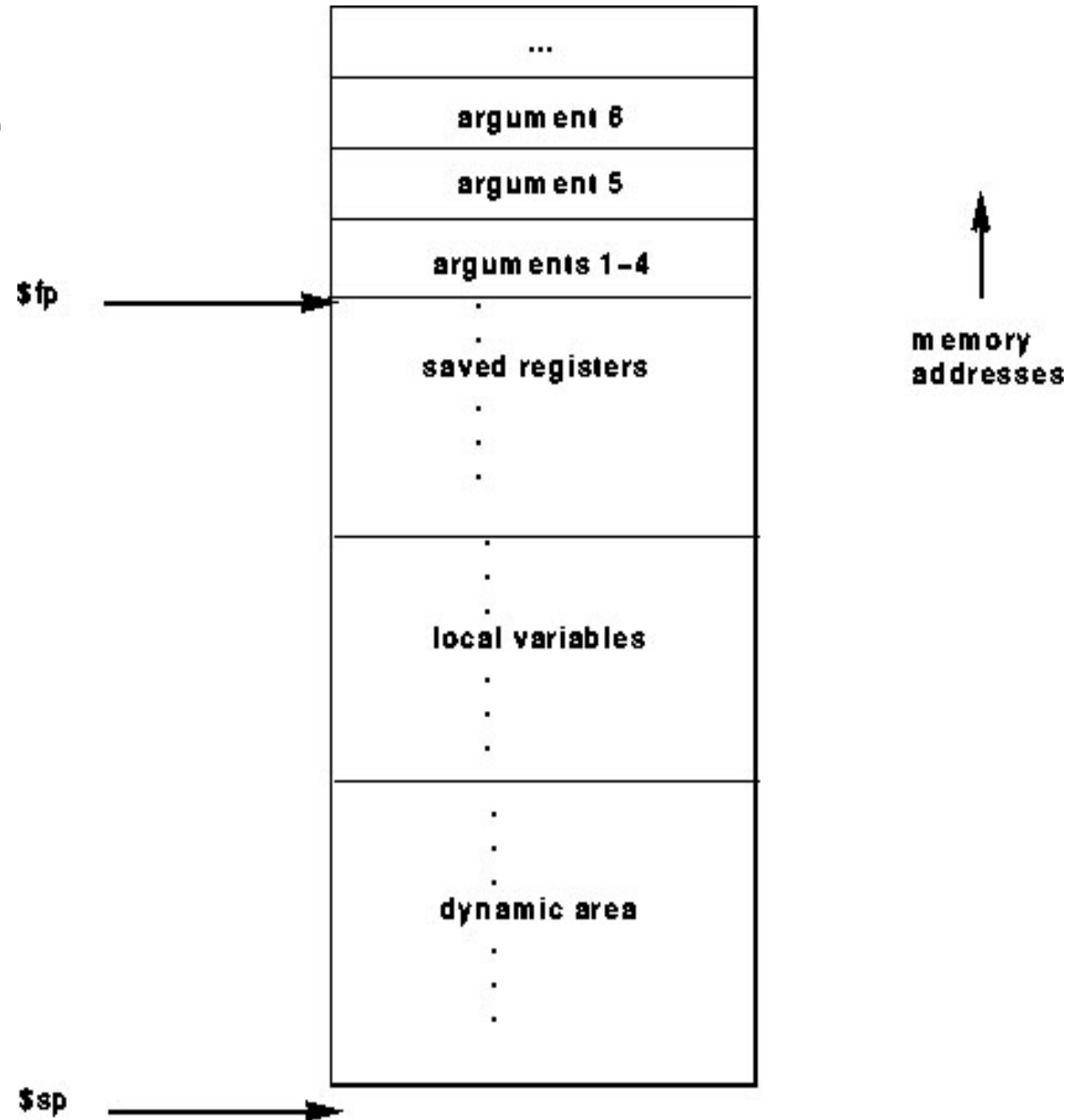
Software Register Conventions

| Reg No | Name | Used for |
|--------|-------|--|
| 0 | zero | Always returns 0 |
| 1 | at | (assembler temporary) Reserved for use by assembler |
| 2-3 | v0-v1 | Value (except FP) returned by subroutine |
| 4-7 | a0-a3 | (arguments) First four parameters for a subroutine |
| 8-15 | t0-t7 | (temporaries) subroutines may use without saving |
| 24-25 | t8-t9 | |
| 16-23 | s0-s7 | Subroutine "register variables"; a subroutine which will write one of these must save the old value and restore it before it exits, so the <i>calling</i> routine sees their values preserved. |
| 26-27 | k0-k1 | Reserved for use by interrupt/trap handler - may change under your feet |
| 28 | gp | global pointer - some runtime systems maintain this to give easy access to (some) "static" or "extern" variables. |
| 29 | sp | stack pointer |
| 30 | s8/fp | 9th register variable. Subroutines which need one can use this as a "frame pointer". |
| 31 | ra | Return address for subroutine |



Stack Frame

- MIPS calling convention for gcc
 - Args 1-4 have space reserved for them



Example Code

```
main ()
{
    int i;

    i =
    sixargs (1,2,3,4,5,6);
}
```

```
int sixargs(int a, int b,
            int c, int d, int e,
            int f)
{
    return a + b + c + d
           + e + f;
}
```



0040011c <main>:

```
40011c:      27bdffd8      addiu   sp,sp,-40
400120:      afbf0024      sw     ra,36(sp)
400124:      afbe0020      sw     s8,32(sp)
400128:      03a0f021      move   s8,sp
40012c:      24020005      li     v0,5
400130:      afa20010      sw     v0,16(sp)
400134:      24020006      li     v0,6
400138:      afa20014      sw     v0,20(sp)
40013c:      24040001      li     a0,1
400140:      24050002      li     a1,2
400144:      24060003      li     a2,3
400148:      0c10002c      jal    4000b0 <sixargs>
40014c:      24070004      li     a3,4
400150:      afc20018      sw     v0,24(s8)
400154:      03c0e821      move   sp,s8
400158:      8fbf0024      lw     ra,36(sp)
40015c:      8fbe0020      lw     s8,32(sp)
400160:      03e00008      jr     ra
400164:      27bd0028      addiu  sp,sp,40
```

...



004000b0 <sixargs>:

| | | | |
|---------|-----------|-------|-------------|
| 4000b0: | 27bdf8ff8 | addiu | sp, sp, -8 |
| 4000b4: | afbe0000 | sw | s8, 0 (sp) |
| 4000b8: | 03a0f021 | move | s8, sp |
| 4000bc: | afc40008 | sw | a0, 8 (s8) |
| 4000c0: | afc5000c | sw | a1, 12 (s8) |
| 4000c4: | afc60010 | sw | a2, 16 (s8) |
| 4000c8: | afc70014 | sw | a3, 20 (s8) |
| 4000cc: | 8fc30008 | lw | v1, 8 (s8) |
| 4000d0: | 8fc2000c | lw | v0, 12 (s8) |
| 4000d4: | 00000000 | nop | |
| 4000d8: | 00621021 | addu | v0, v1, v0 |
| 4000dc: | 8fc30010 | lw | v1, 16 (s8) |
| 4000e0: | 00000000 | nop | |
| 4000e4: | 00431021 | addu | v0, v0, v1 |
| 4000e8: | 8fc30014 | lw | v1, 20 (s8) |
| 4000ec: | 00000000 | nop | |
| 4000f0: | 00431021 | addu | v0, v0, v1 |
| 4000f4: | 8fc30018 | lw | v1, 24 (s8) |
| 4000f8: | 00000000 | nop | |



```
4000fc:    00431021    addu    v0,v0,v1
400100:    8fc3001c    lw      v1,28(s8)
400104:    00000000    nop
400108:    00431021    addu    v0,v0,v1
40010c:    03c0e821    move    sp,s8
400110:    8fbe0000    lw      s8,0(sp)
400114:    03e00008    jr      ra
400118:    27bd0008    addiu   sp,sp,8
```



System Calls

Continued



User and Kernel Execution

- Simplistically, execution state consists of
 - Registers, processor mode, PC, SP
- User applications and the kernel have their own execution state.
- System call mechanism safely transfers from user execution to kernel execution and back.



System Call Mechanism in Principle

- Processor mode
 - Switched from user-mode to kernel-mode
 - Switched back when returning to user mode
- SP
 - User-level SP is saved and a kernel SP is initialised
 - User-level SP restored when returning to user-mode
- PC
 - User-level PC is saved and PC set to kernel entry point
 - User-level PC restored when returning to user-level
 - Kernel entry via the designated entry point must be strictly enforced



System Call Mechanism in Principle

- Registers
 - Set at user-level to indicate system call type and its arguments
 - A convention between applications and the kernel
 - Some registers are preserved at user-level or kernel-level in order to restart user-level execution
 - Depends on language calling convention etc.
 - Result of system call placed in registers when returning to user-level
 - Another convention

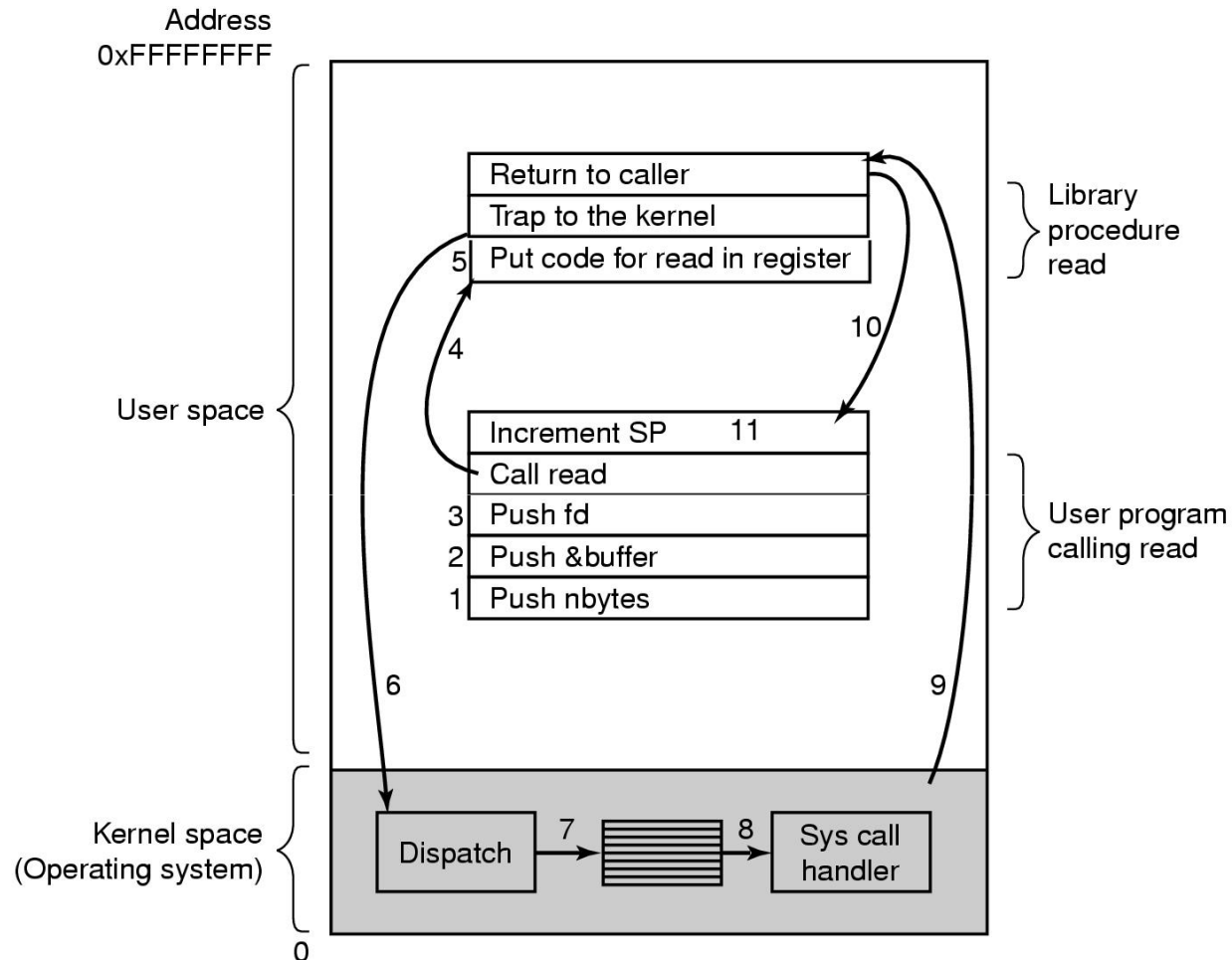


Why do we need system calls?

- Why not simply jump into the kernel via a function call????
 - Function calls do not
 - Change from user to kernel mode
 - and eventually back again
 - Restrict possible entry points to secure locations



Steps in Making a System Call



There are 11 steps in making the system call read (fd, buffer, nbytes)



MIPS System Calls

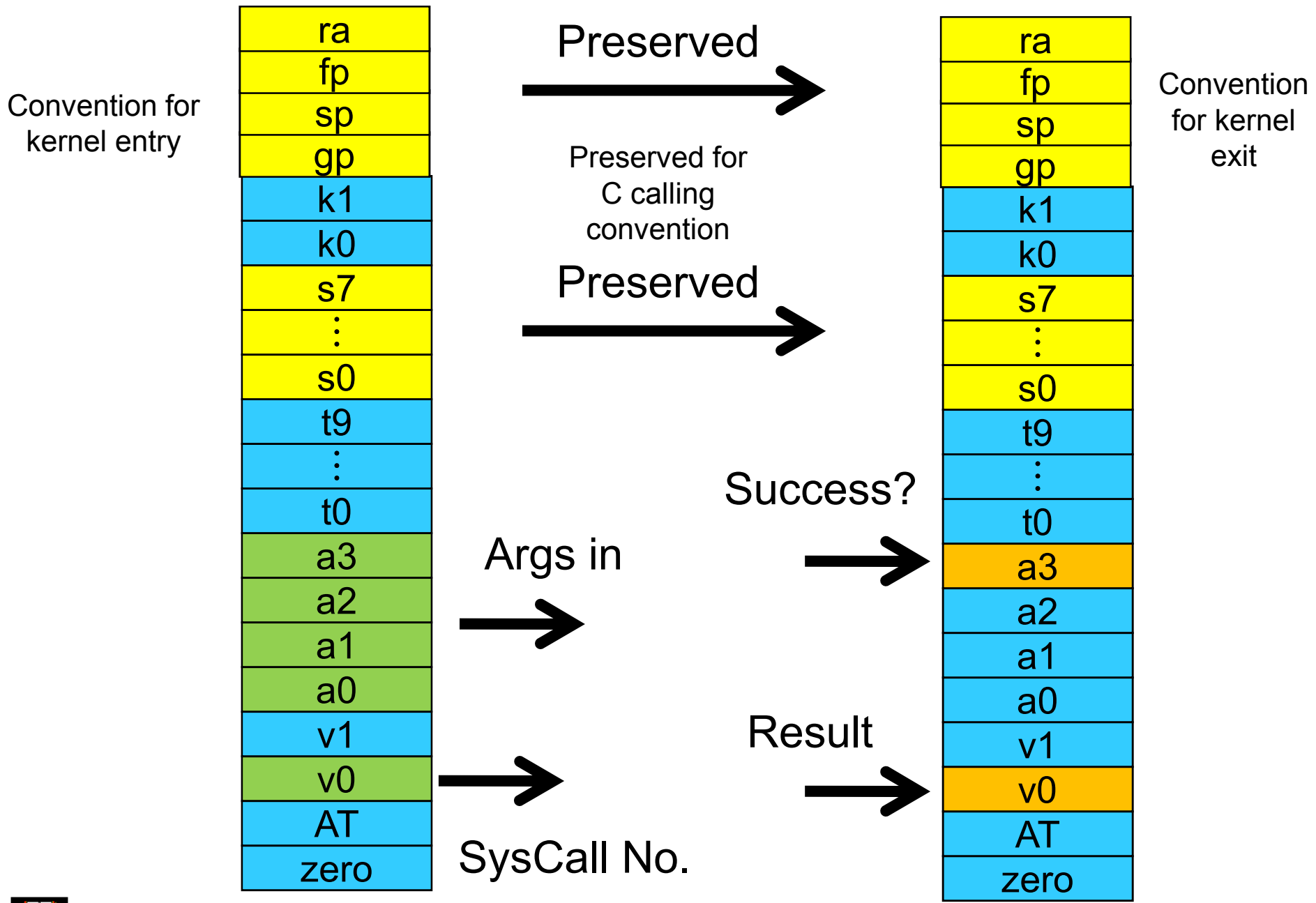
- System calls are invoked via a *syscall* instruction.
 - The *syscall* instruction causes an exception and transfers control to the general exception handler
 - A convention (an agreement between the kernel and applications) is required as to how user-level software indicates
 - Which system call is required
 - Where its arguments are
 - Where the result should go



OS/161 Systems Calls

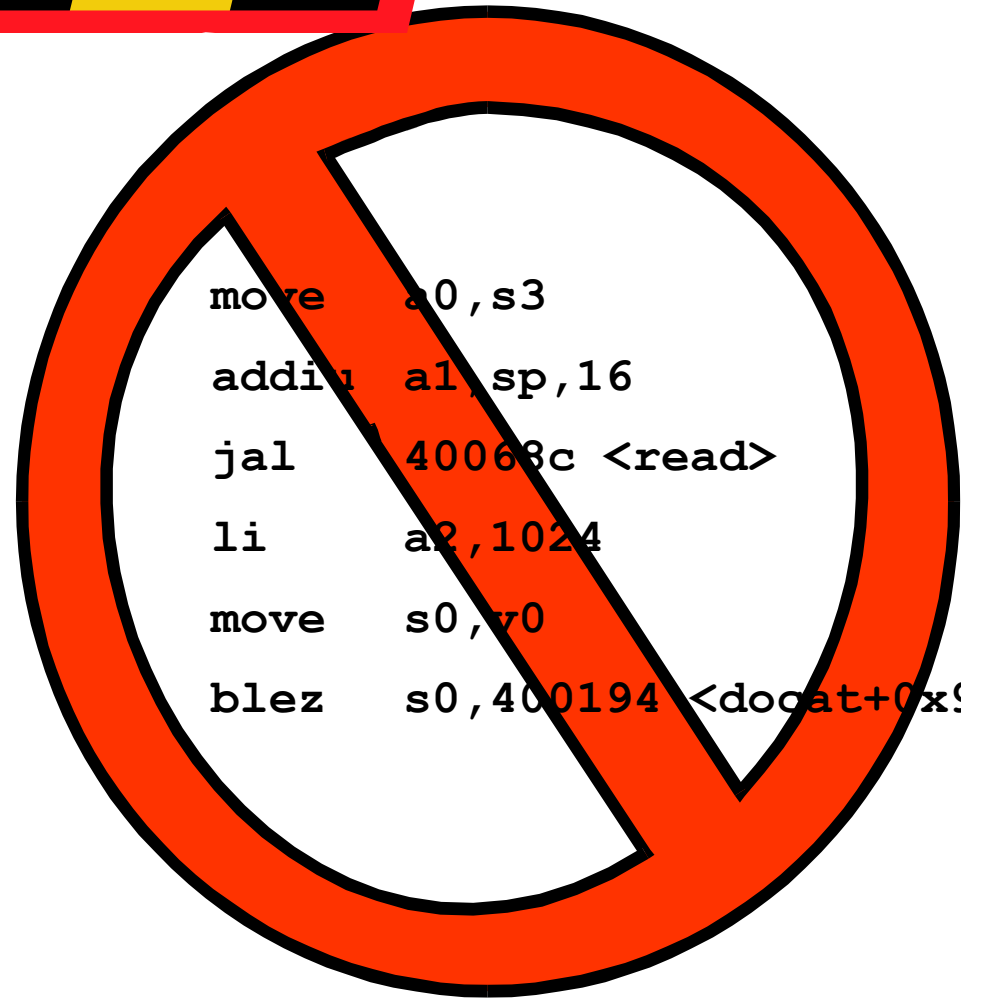
- OS/161 uses the following conventions
 - Arguments are passed and returned via the normal C function calling convention
 - Additionally
 - Reg v0 contains the system call number
 - On return, reg a3 contains
 - 0: if success, v0 contains successful result
 - not 0: if failure, v0 has the errno.
 - » v0 stored in errno
 - » -1 returned in v0





CAUTION

- Seriously low-level code follows
- This code is not for the faint hearted



User-Level System Call Walk Through

```
int read(int filehandle, void *buffer, size_t size)
```

- Three arguments, one return value
- Code fragment calling the read function

```
400124: 02602021    move    a0,s3
400128: 27a50010    addiu   a1,sp,16
40012c: 0c1001a3    jal    40068c <read>
400130: 24060400    li     a2,1024
400134: 00408021    move   s0,v0
400138: 1a000016    blez   s0,400194 <docat+0x94>
```

- Args are loaded, return value is tested



The read() syscall function part 1

```
0040068c <read>:
```

```
40068c: 08100190  j      400640 <__syscall>  
400690: 24020005  li     v0,5
```

- Appropriate registers are preserved
 - Arguments (a0-a3), return address (ra), etc.
- The syscall number (5) is loaded into v0
- Jump (not jump and link) to the common syscall routine



The read() syscall function part 2

Generate a syscall
exception

00400640 <__syscall>:

| | | |
|---------|----------|---------------------------------|
| 400640: | 0000000c | syscall |
| 400644: | 10e00005 | beqz a3,40065c <__syscall+0x1c> |
| 400648: | 00000000 | nop |
| 40064c: | 3c011000 | lui at,0x1000 |
| 400650: | ac220000 | sw v0,0(at) |
| 400654: | 2403ffff | li v1,-1 |
| 400658: | 2402ffff | li v0,-1 |
| 40065c: | 03e00008 | jr ra |
| 400660: | 00000000 | nop |



The read() syscall function part 2

Test success, if yes,
branch to return
from function

```
00400640 <__syscall>:
 400640: 0000000c      syscall
 400644: 10e00005      beqz    a3,40065c <__syscall+0x1c>
 400648: 00000000      nop
 40064c: 3c011000      lui    at,0x1000
 400650: ac220000      sw     v0,0(at)
 400654: 2403ffff      li     v1,-1
 400658: 2402ffff      li     v0,-1
 40065c: 03e00008      jr     ra
 400660: 00000000      nop
```



The read() syscall function part 2

```
00400640 <__syscall>:
 400640: 0000000c syscall
 400644: 10e00005 beqz   a3,40065c
 400648: 00000000 nop
 40064c: 3c011000 lui    at,0x1000
 400650: ac220000 sw     v0,0(at)
 400654: 2403ffff li     v1,-1
 400658: 2402ffff li     v0,-1
 40065c: 03e00008 jr     ra
 400660: 00000000 nop
```

If failure, store code
in *errno*



The read() syscall function part 2

```
00400640 <__syscall>:
 400640: 0000000c syscall
 400644: 10e00005 beqz   a3,40065c
 400648: 00000000 nop
 40064c: 3c011000 lui    at,0x1000
 400650: ac220000 sw     v0,0(at)
 400654: 2403ffff li     v1,-1
 400658: 2402ffff li     v0,-1
 40065c: 03e00008 jr     ra
 400660: 00000000 nop
```

Set read() result to
-1



The read() syscall function part 2

```
00400640 <__syscall>:
 400640: 0000000c      syscall
 400644: 10e00005      beqz  a3,40065c
 400648: 00000000      nop
 40064c: 3c011000      lui   at,0x1000
 400650: ac220000      sw    v0,0(at)
 400654: 2403ffff      li    v1,-1
 400658: 2402ffff      li    v0,-1
 40065c: 03e00008      jr    ra
 400660: 00000000      nop
```

Return to location
after where read()
was called



Summary

- From the caller's perspective, the read() system call behaves like a normal function call
 - It preserves the calling convention of the language
- However, the actual function implements its own convention by agreement with the kernel
 - Our OS/161 example assumes the kernel preserves appropriate registers(s0-s8, sp, gp, ra).
- Most languages have similar *support libraries* that interface with the operating system.



System Calls - Kernel Side

- Things left to do
 - Change to kernel stack
 - Preserve registers by saving to memory (the stack)
 - Leave saved registers somewhere accessible to
 - Read arguments
 - Store return values
 - Do the “read()”
 - Restore registers
 - Switch back to user stack
 - Return to application



exception:

```
move k1, sp /* Save previous stack pointer in k1 */
mfc0 k0, c0_status /* Get status register */
andi k0, k0, CST_ /* Check the we-were-in-user-mode bit */
beq k0, $0, 1f /* If clear, from kernel, already have stack */
nop /* delay slot */
```

```
/* Coming from user mode ... into sp */
la k0, curkstack /* curkstack" */
lw sp, 0(k0) /* value */
nop /* e load */
```

1:

```
mfc0 k0, c0_cause /* No ... cause. */
j common_exception /* de */
nop
```

Note k0, k1
registers
available for
kernel use



exception:

```
    move k1, sp          /* Save previous stack pointer in k1 */
    mfc0 k0, c0_status   /* Get status register */
    andi k0, k0, CST_Kup /* Check the we-were-in-user-mode bit */
    beq k0, $0, 1f /* If clear, from kernel, already have stack */
    nop                  /* delay slot */

/* Coming from user mode - load kernel stack into sp */
    la k0, curkstack     /* get address of "curkstack" */
    lw sp, 0(k0)         /* get its value */
    nop                  /* delay slot for the load */
```

1:

```
    mfc0 k0, c0_cause    /* Now, load the exception cause. */
    j common_exception   /* Skip to common code */
    nop                  /* delay slot */
```



common_exception:

```
/*
 * At this point:
 *     Interrupts are off. (The processor did this for us.)
 *     k0 contains the exception cause value.
 *     k1 contains the old stack pointer.
 *     sp points into the kernel stack.
 *     All other registers are untouched.
 */

/*
 * Allocate stack space for 37 words to hold the trap frame,
 * plus four more words for a minimal argument block.
 */
addi sp, sp, -164
```




```
/* The order here must match mips/include/trapframe.h. */
```

```
sw ra, 160(sp)    /* dummy for gdb */  
sw s8, 156(sp)   /* save s8 */  
sw sp, 152(sp)   /* dummy for gdb */  
sw gp, 148(sp)   /* save gp */  
sw k1, 144(sp)   /* dummy for gdb */  
sw k0, 140(sp)   /* dummy for gdb */  
  
sw k1, 152(sp)   /* real saved sp */  
nop              /* delay slot for store */  
  
mfc0 k1, c0_epc /* Copr.0 reg 13 == PC for  
sw k1, 160(sp)  /* real saved PC */
```

These six stores are
a “hack” to avoid
confusing GDB
You can ignore the
details of why and
how



```
/* The order here must match mips/include/trapframe.h. */
```

```
sw ra, 160(sp)    /* dummy for gdb */  
sw s8, 156(sp)   /* save s8 */  
sw sp, 152(sp)   /* dummy for gdb */  
sw gp, 148(sp)   /* save gp */  
sw k1, 144(sp)   /* dummy for gdb */  
sw k0, 140(sp)   /* dummy for gdb */  
  
sw k1, 152(sp)   /* real saved sp */  
nop              /* delay slot for store */  
  
mfc0 k1, c0_epc /* Copr.0 reg 13 == PC for exception */  
sw k1, 160(sp)  /* real saved PC */
```

The real work starts
here



```
sw t9, 136(sp)
sw t8, 132(sp)
sw s7, 128(sp)
sw s6, 124(sp)
sw s5, 120(sp)
sw s4, 116(sp)
sw s3, 112(sp)
sw s2, 108(sp)
sw s1, 104(sp)
sw s0, 100(sp)
sw t7, 96(sp)
sw t6, 92(sp)
sw t5, 88(sp)
sw t4, 84(sp)
sw t3, 80(sp)
sw t2, 76(sp)
sw t1, 72(sp)
sw t0, 68(sp)
sw a3, 64(sp)
sw a2, 60(sp)
sw a1, 56(sp)
sw a0, 52(sp)
sw v1, 48(sp)
sw v0, 44(sp)
sw AT, 40(sp)
sw ra, 36(sp)
```

Save all the registers
on the kernel stack



```

/*
 * Save special registers.
 */
mfhi t0
mflo t1
sw t0, 32(sp)
sw t1, 28(sp)

```

We can now use the other registers (t0, t1) that we have preserved on the stack

```

/*
 * Save remaining exception context information.
 */

sw k0, 24(sp) /* k0 was loaded with cause earlier */
mfc0 t1, c0_status /* Copr.0 reg 11 == status */
sw t1, 20(sp)
mfc0 t2, c0_vaddr /* Copr.0 reg 8 == faulting vaddr */
sw t2, 16(sp)

/*
 * Pretend to save $0 for gdb's benefit.
 */
sw $0, 12(sp)

```



```
/*
 * Prepare to call mips_trap(struct trapframe *)
 */

addiu a0, sp, 16          /* set argument */
jal mips_trap            /* call it */
nop                      /* delay slot */
```

Create a pointer to the base of the saved registers and state in the first argument register



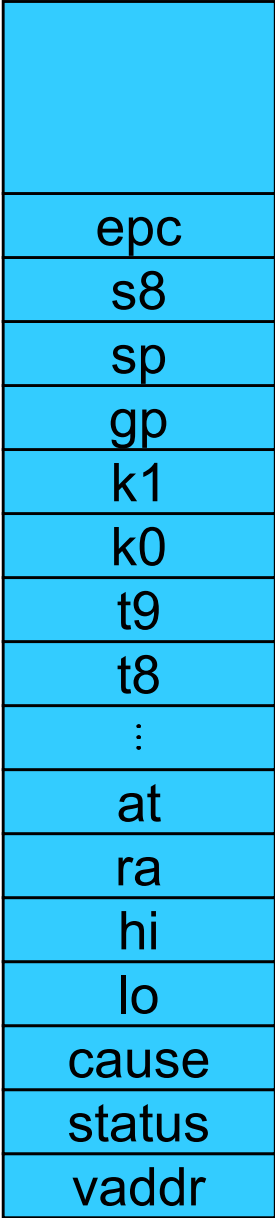
```

struct trapframe {
    u_int32_t tf_vaddr;          /* vaddr register */
    u_int32_t tf_status;        /* status register */
    u_int32_t tf_cause;         /* cause register */
    u_int32_t tf_lo;
    u_int32_t tf_hi;
    u_int32_t tf_ra; /* Saved register 31 */
    u_int32_t tf_at; /* Saved register 1 (AT) */
    u_int32_t tf_v0; /* Saved register 2 (v0) */
    u_int32_t tf_v1; /* etc. */
    u_int32_t tf_a0;
    u_int32_t tf_a1;
    u_int32_t tf_a2;
    u_int32_t tf_a3;
    u_int32_t tf_t0;
    :
    u_int32_t tf_t7;
    u_int32_t tf_s0;
    :
    u_int32_t tf_s7;
    u_int32_t tf_t8;
    u_int32_t tf_t9;
    u_int32_t tf_k0; /* dummy (see exception.S comment) */
    u_int32_t tf_k1; /* dummy */
    u_int32_t tf_gp;
    u_int32_t tf_sp;
    u_int32_t tf_s8;
    u_int32_t tf_epc;          /* coprocessor 0 epc register

```

By creating a pointer to here of type struct trapframe *, we can access the user's saved registers as normal variables within 'C'

Kernel Stack



Now we arrive in the 'C' kernel

```
/*  
 * General trap (exception) handling function for mips.  
 * This is called by the assembly-language exception handler once  
 * the trapframe has been set up.  
 */  
void  
mips_trap(struct trapframe *tf)  
{  
    u_int32_t code, isutlb, iskern;  
    int savespl;  
  
    /* The trap frame is supposed to be 37 registers long. */  
    assert(sizeof(struct trapframe)==(37*4));  
  
    /* Save the value of curspl, which belongs to the old context. */  
    savespl = curspl;  
  
    /* Right now, interrupts should be off. */  
    curspl = SPL_HIGH;
```



What happens next?

- The kernel deals with whatever caused the exception
 - Syscall
 - Interrupt
 - Page fault
 - It potentially modifies the *trapframe*, etc
 - E.g., Store return code in v0, zero in a3
- ‘mips_trap’ eventually returns



exception_return:

```
/*      16(sp)          no need to restore tf_vaddr */
lw t0, 20(sp)          /* load status register value into t0 */
nop                    /* load delay slot */
mtc0 t0, c0_status     /* store it back to coprocessor 0 */
/*      24(sp)          no need to restore tf_cause */

/* restore special registers */
lw t1, 28(sp)
lw t0, 32(sp)
mtlo t1
mthi t0

/* load the general registers */
lw ra, 36(sp)

lw AT, 40(sp)
lw v0, 44(sp)
lw v1, 48(sp)
lw a0, 52(sp)
lw a1, 56(sp)
lw a2, 60(sp)
lw a3, 64(sp)
```



```
lw t0, 68(sp)
lw t1, 72(sp)
lw t2, 76(sp)
lw t3, 80(sp)
lw t4, 84(sp)
lw t5, 88(sp)
lw t6, 92(sp)
lw t7, 96(sp)
lw s0, 100(sp)
lw s1, 104(sp)
lw s2, 108(sp)
lw s3, 112(sp)
lw s4, 116(sp)
lw s5, 120(sp)
lw s6, 124(sp)
lw s7, 128(sp)
lw t8, 132(sp)
lw t9, 136(sp)

/*      140(sp)      "saved" k0 was dummy garbage anyway */
/*      144(sp)      "saved" k1 was dummy garbage anyway */
```



```

lw gp, 148(sp)      /* restore gp */
/*      152(sp)      stack pointer - below */
lw s8, 156(sp)     /* restore s8 */
lw k0, 160(sp)     /* fetch exception return PC into k0 */

lw sp, 152(sp)     /* fetch saved sp (must be last) */

/* done */
jr k0              /* jump back */
rfe               /* in delay slot */
.end common_exception

```

Note again that only
k0, k1 have been
trashed

