How does an assembler work?

In a two-pass assembler

PASS 1: Symbol table generation

PASS 2: Code generation

Illustration of the two passes

```
        .data
L1:     .word 0x2345          # some arbitrary value
L2:     .word 0x3366          # some arbitrary value
Res:    .space 4
        .text
        .globl main
main:   lw $t0, L1($0)        #load the first value
        lw $t1, L2($0)        # load the second value
        and $t2, $t0, $t1     # compute the bit-by-bit AND
        or $t3, $t0, $t1      # compute the bit-by-bit OR
        sw $t3, Res($0)       # store result at location in memory
        li $v0, 10            # code for program end
        syscall
```

First, a symbol table is formed. It starts from a specified address (say 4000)

Pass 1. Symbol Table generation

Symbol	Address	Comments
L1	4000	Takes 4 bytes
L2	4004	Takes 4 bytes
Res	4008	Takes 4 bytes
main	4012	Program starts from here

Pass 2. Code generation

Address	Content	Comment
4000	Ox2345	Content of L1
4004	Use Ox3366 for L2	Content of L2
4008	????	4 byte reserved
4012	Binary for lw $t0, L1($0)	1st instruction
4016	Binary of lw $t1, L2($0)	2nd instruction
… …	… … … …	… …

This is the object file

In what language will the compiler be written?

Loaders and Linkers

The linker stitches together independently assembled machine language programs by patching both internal and external references. It produces an executable file.

The loader reads the executable file headers, determines the sizes of the text and data segments, makes room in the main memory to accommodate them, and initializes all registers.

Now the program is ready to run.

Other architectures

Not all processors are like MIPS.

Example. Accumulator-based machines

A single register, called the accumulator, stores the operand before the operation, and stores the result after the operation.

```
Load    x    # into accumulator from memory
Add     y    # add y from memory to the acc
Store   z    # store acc to memory as z
```

Can we have an instruction like

```
add  z, x, y    # z:= x + y, (x, y, z in memory) ?
```

For some machines, YES, but not in MIPS! What are the advantages and disadvantages of such an instruction?

Load-store machines

MIPS is a **load-store architecture**. Only load and store instructions access the memory, all other instructions use registers as operands. What is the motivation?

Register access is much faster than memory access, so the program will run faster.

Reduced Instruction Set Computers (RISC)

- The instruction set has only a small number of frequently used instructions. This lowers processor cost, without much impact on performance.
- All instructions have the same length.
- Load-store architecture.

Non-RISC machines are called CISC (Complex Instruction Set Computer). Example: Pentium

Another classification

3-address	add r1, r2, r3	(r1 ← r2 + r3)
2-address	add r1, r2	(r1 ← r1 + r2)
1-address	add r1	(to the accumulator)
0-address or stack machines		(see below)

Example of stack architecture

Consider evaluating f = x * (y + z)

Push x

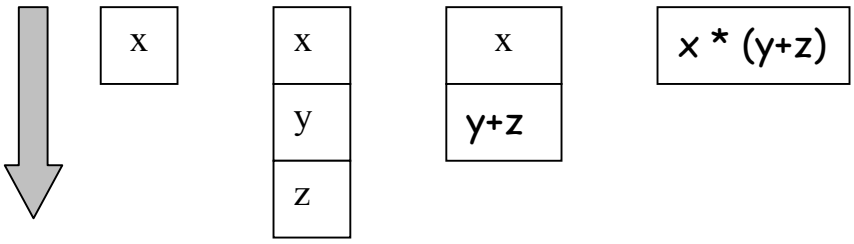
Push y

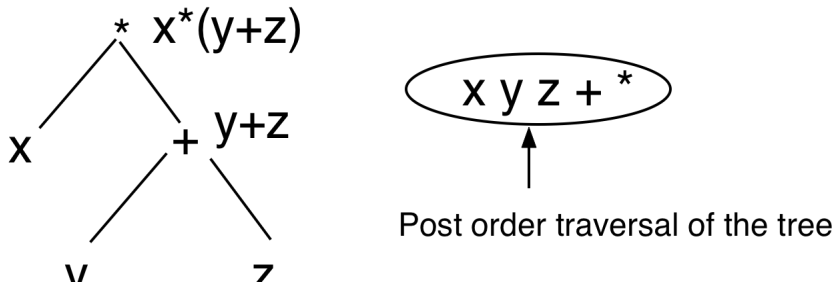
Push z

Add

Multiply

Pop f





Computer Arithmetic: Dealing with overflow

How to represent negative integers? The most widely used convention is 2's complement representation.

+14	=	0, 1 1 1 0
-14	=	1, 0 0 1 0

Largest integer represented using n-bits is $+ (2^{n-1} - 1)$

Smallest integer represented using n-bits is $- 2^{n-1}$

So, using 4-bits (that includes 1 sign bit)

the largest integer is 0,111 (=7), and

the smallest integer is 1,000 (= -8)

Review binary-to decimal and binary-to-hex conversions.

Review BCD (Binary Coded Decimal) and ASCII codes.

How to represent fractions?

Overflow

+12	=	0, 1 1 0 0	+12	=	0, 1 1 0 0
+2	=	0, 0 0 1 0	+7	=	0, 0 1 1 1
add					add
+14	=	0, 1 1 1 0	?	=	1, 0 0 1 1 (WRONG)

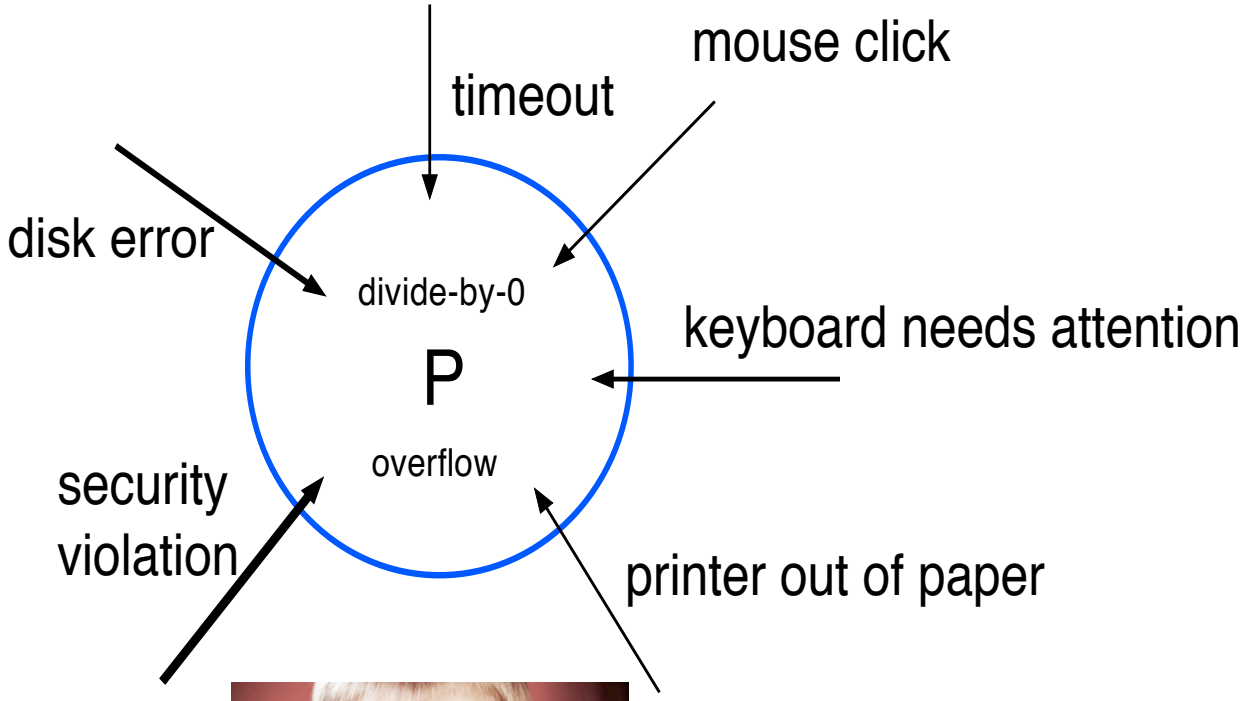
Addition of a positive and a negative number does not lead to overflow. How to detect overflow? Here is a clue.

		0 0			0 1
+12	=	0, 1 1 0 0	+12	=	0, 1 1 0 0
+2	=	0, 0 0 1 0	+7	=	0, 0 1 1 1
add					add
+14	=	0, 1 1 1 0	?	=	1, 0 0 1 1 (WRONG)

The following sequence of MIPS instructions can detect overflow in signed addition of $t1 and $t2:

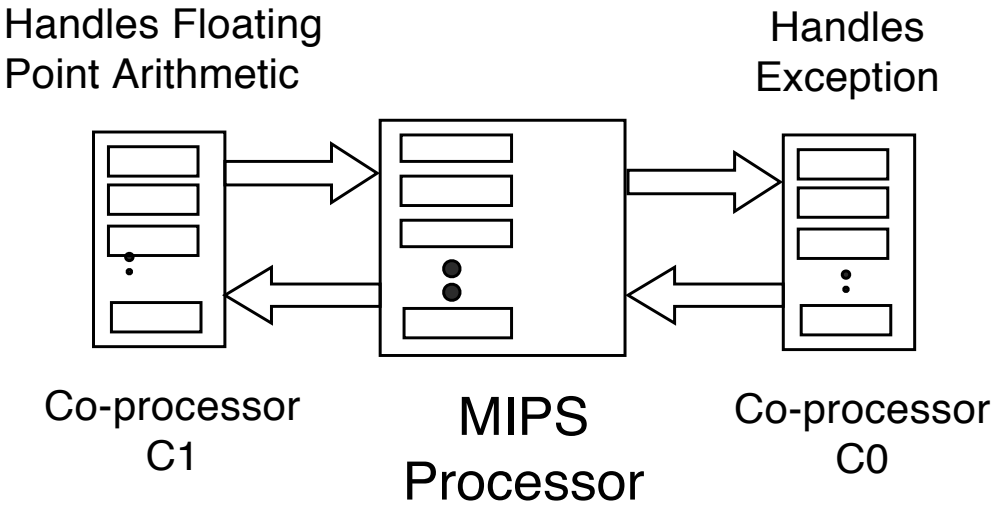
```
addu $t0, $t1, $t2          # add unsigned
xor  $t3, $t1, $t2          # check if signs differ
slt  $t3, $t3, $zero        # $t3=1 if signs differ
bne  $t3, $zero, no_overflow
xor  $t3, $t0, $t1          # sum sign = operand sign?
slt  $t3, $t3, $zero        # if not, then $t3=1
bne  $t3, $zero, overflow
no_overflow:
. . .
. . .
overflow:
<Do something to handle overflow>
```

Exception Handling



There are different levels of interrupt

The Basics of Exception Handling



Interrupts

Initiated outside the instruction stream

Arrive asynchronously (at no specific time),

Examples:

- I/O device status change
- I/O device error condition

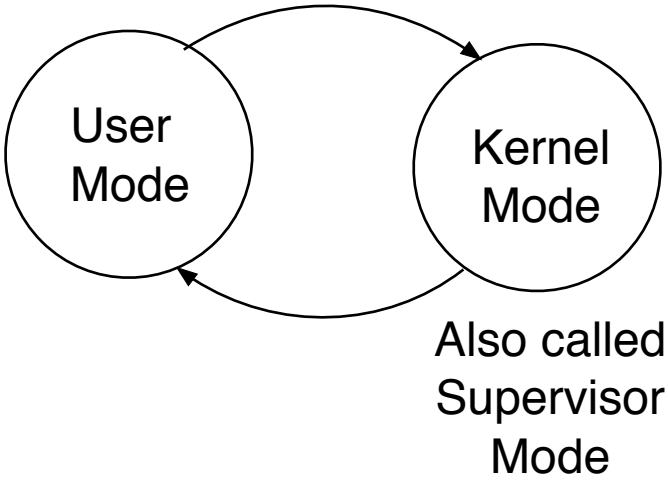
Traps occur due to something in instruction stream.

Examples:

- Unaligned address error or undefined instruction
- Arithmetic overflow
- System call

Different machines classify them in different ways.

When we use the machine to run a program, we operate in the user mode.



An exception takes away control from the user and transfers it to the supervisor (i.e. the operating system). The processor is now in the kernel mode. The supervisor determines how the exception should be handled.

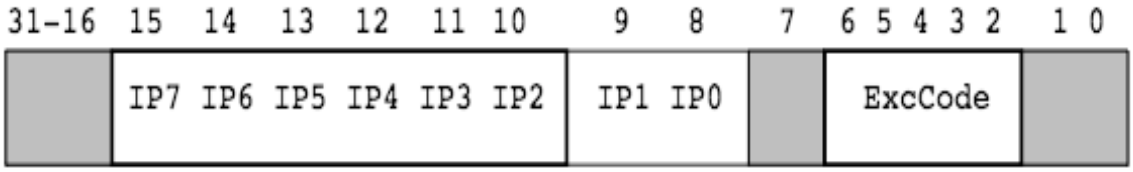
(Think of the various reasons an exception can occur).

An exception triggers an **unscheduled procedure call**.

Inside Coprocessor C0

Coprocessor C0 has a cause register (Register 13) that contains a 5-bit code to identify the cause of exception

(Bits 15-10: Pending Interrupts (not yet serviced)



CAUSE REGISTER

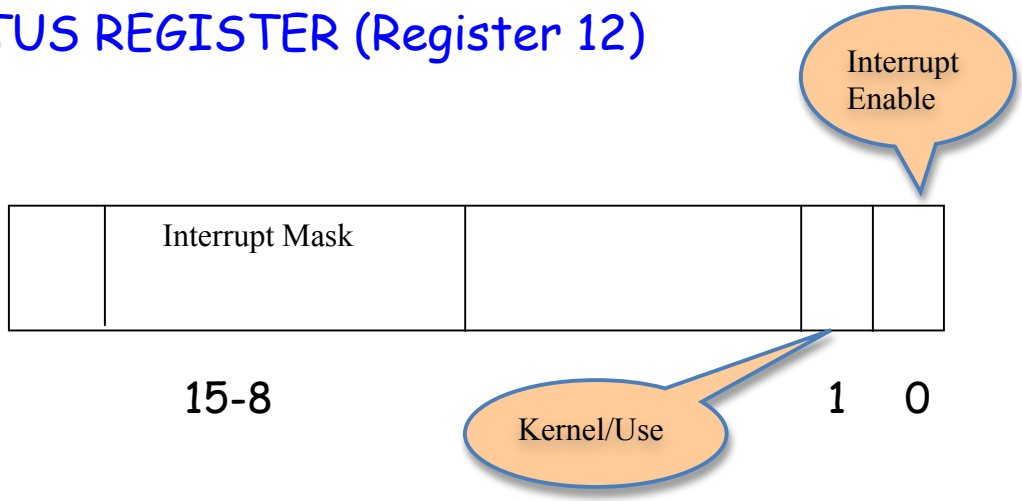
[Exception Code determines what caused the exception

like 0 means I/O interrupt

12 means arithmetic overflow etc]

The MIPS instruction that causes an ***exception*** sets the exception code. It then switches to the kernel mode (designated by a bit in the status register of C0, register 12) and transfers control to a predefined address to invoke a routine (exception handler, which starts from 0x80000080 for MIPS) for handling the exception.

STATUS REGISTER (Register 12)



Interrupts can be disabled if you don't want the program to be disturbed.

Bit 0 = 0 means interrupt is disabled, and

Bit 0 = 1 means interrupts are enabled.

Bits 15-8 mask interrupts at specific levels (what is this?)

Exception Program Counter (EPC) saves the return address of the main program when an interrupt occurs (you don't have the luxury to plan ahead and use a JAL instruction, since interrupts can happen at any time – it is not a planned event. So you can't use ra or r31 for this.

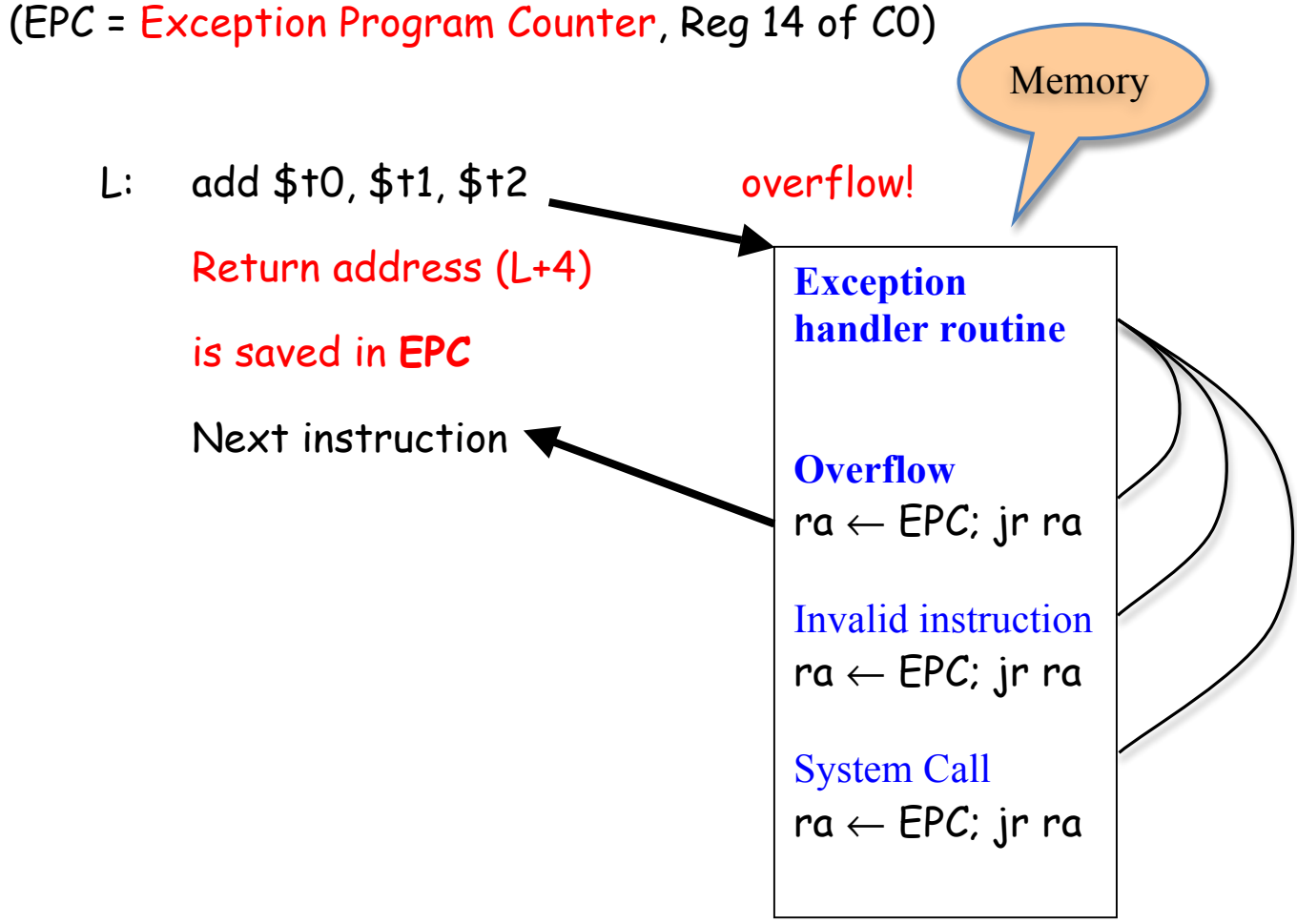
Special instructions mfc0 (move from C0) and mtc0 (move to C0) are used to transfer data from and to a coprocessor register. Thus,

mfc0 $10, $13 implies r10 ← C0 register 13 (Cause Reg)

mtc0 $8, $13 implies r8 → C0 register 13

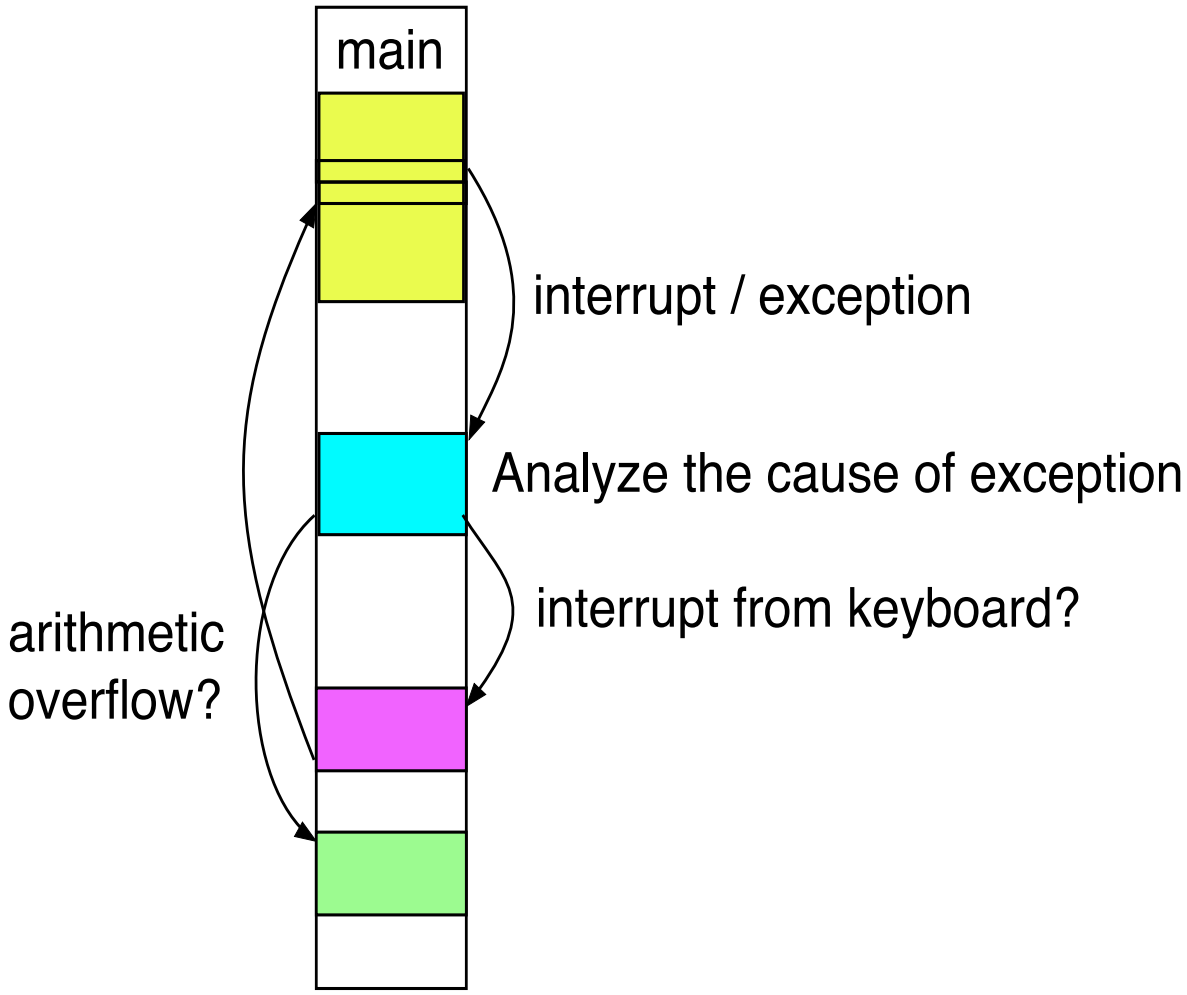
mfc0 $r4, $14 implies r4 ← C)0 register 14 (EPC)

(EPC = Exception Program Counter, Reg 14 of C0)



The Exception Handler determines the cause of the exception by looking at the exception code bits. Then it jumps to the appropriate exception handling routine. Finally, it returns to the main program.

Visualizing Exception Handling



Exceptions cause mostly unscheduled procedure calls.

Example 1: Read one input from a Keyboard

Consider reading a value from the keyboard. Assume that the interrupt enable bit is set to 1. The first line, ".text 0x80000080" places the code explicitly at the memory location where the *interrupt service routine* is called.

```
.text       0x80000080
            mfc0 $k0, $13          # $k0 = $Cause;
            mfc0 $k1, $14          # $k1 = $EPC;
            andi $k0, $k0, 0x003c # $k0 &= 0x003c (hex);
                                   # Filter the Exception Code;
            bne $k0, $zero, NotIO # if ($k0 ≠ 0) go to NotIO
                                   # Exception Code 0 => I/O instr.
            sw $ra, save0($0)      # save0 = $ra;
            jal ReadByte           # ReadByte(); (Get the byte).
            lw $ra, save0($0)      # $ra = save0;
            jr $k1                 # return;
NotIO:      Other routines here
```

Note that procedure ReadByte must save all registers that it plans to use, and restore them later.

Example 2

Simulate a trap because the trap condition evaluates true, control jumps to the exception handler, the exception handler returns control to the instruction following the one that triggered the exception, then the program terminates normally.

```
.text main:
teqi $t0,0      # trap when $t0 contains 0
li   $v0, 10    # After return syscall
                # terminate normally

# Trap handler in the standard MIPS32 kernel
text segment

.ktext 0x80000180

move $k0,$v0    # Save $v0 value
move $k1,$a0    # Save $a0 value
la   $a0, msg   # address of string to print
li   $v0, 4     # Print String syscall

move $v0,$k0    # Restore $v0
move $a0,$k1    # Restore $a0
mfc0 $k0,$14    # Coprocessor 0 register $14
                # has address of the trapping
                # instruction
addi $k0,$k0,4 # point to next instruction
mtc0 $k0,$14    # Store new addr back to $14
eret            # Error return; PC ← $14

.kdata
msg:        .asciiz "Trap generated"
```