

# Computer Organization and Assembly Language CS/EE 320 Spring 2025

**Shahid Masud**

**Lecture 8**

- MIPS Addressing Mode Examples – contd from previous lecture
- Push and Pop operations in Procedure Calls
- Examples and Questions
- Interrupt Processing
- How Does MIPS Architecture Cater to Interrupts
- **QUIZ Next WEEK!!!**

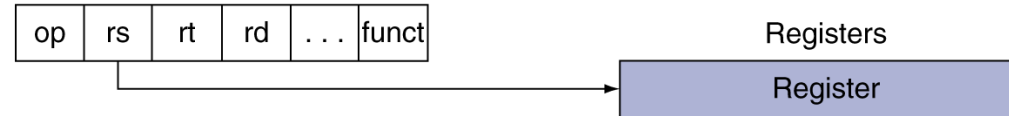
**Quiz 2 next week**

# MIPS Addressing Modes Summary

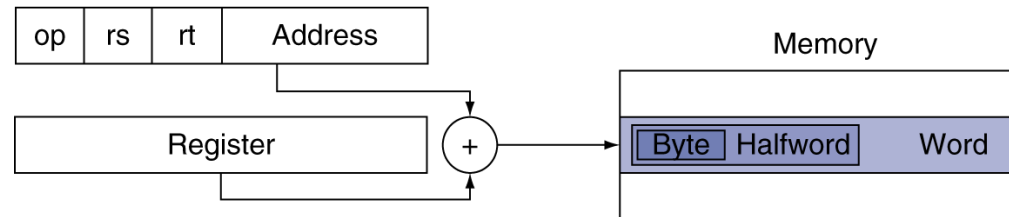
## 1. Immediate addressing



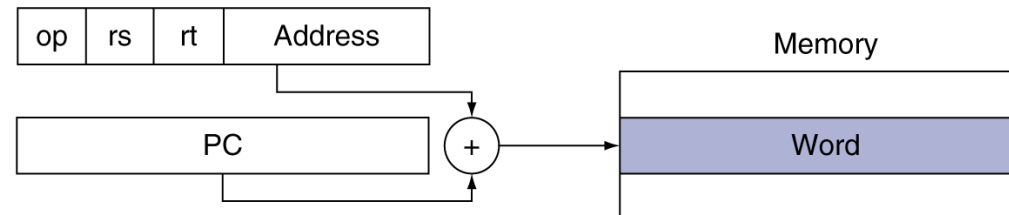
## 2. Register addressing



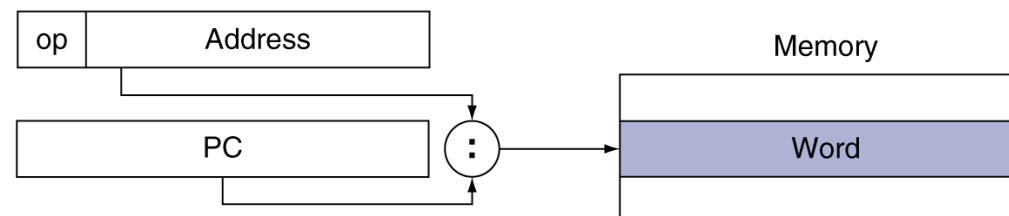
## 3. Base addressing



## 4. PC-relative addressing

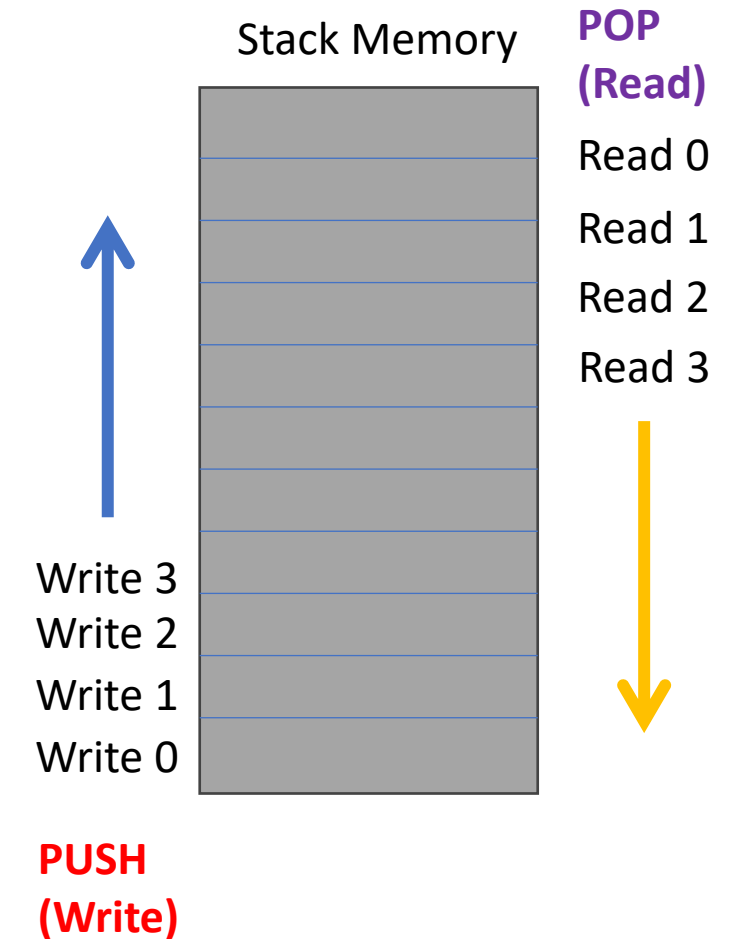
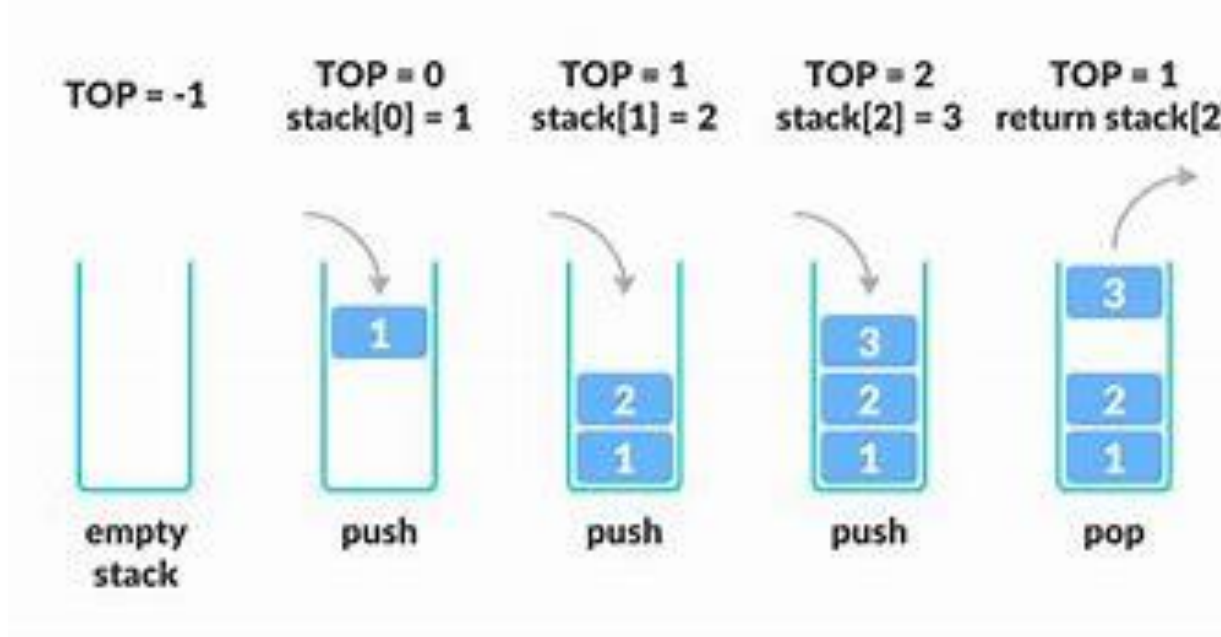


## 5. Pseudodirect addressing



# Stack memory – Push and Pop operations

Status of Registers in CPU is stored in Stack Area of Memory  
Before processing each procedure, function, ISR, Exception



Usually LIFO – Last in First out

Stack Write and Read in opposite direction

# Register usage convention

## RECAP

R0	\$0	Constant 0
R1	\$at	Reserved Temp.
R2	\$v0	Return Values
R3	\$v1	
R4	\$a0	Procedure arguments
R5	\$a1	
R6	\$a2	
R7	\$a3	
R8	\$t0	Caller Save Temporaries: May be overwritten by called procedures
R9	\$t1	
R10	\$t2	
R11	\$t3	
R12	\$t4	
R13	\$t5	
R14	\$t6	
R15	\$t7	

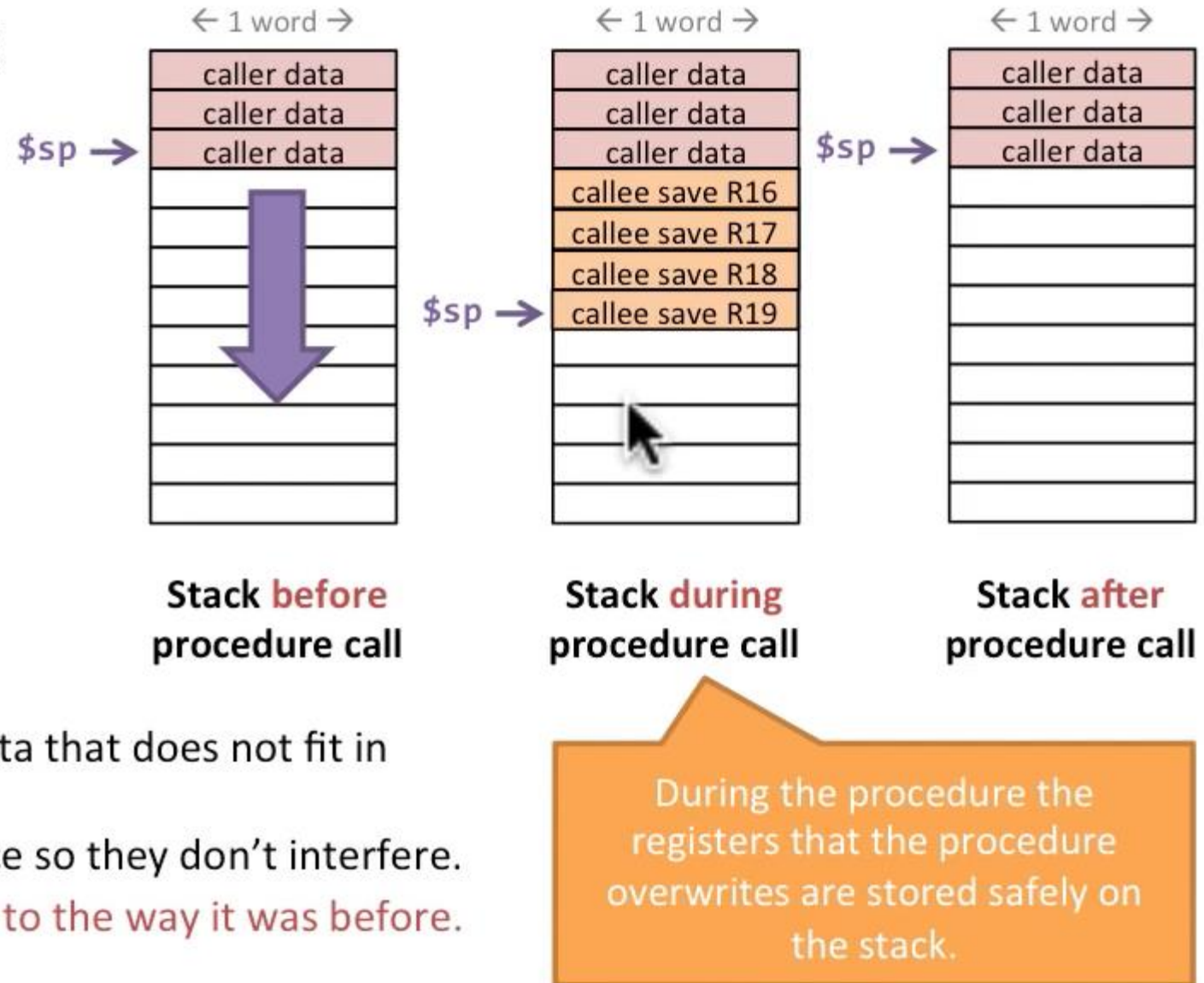
**Caller Save**  
If the caller uses these register, then the caller must save them in case the callee overwrites them.

R16	\$s0	Callee Save Temporaries: May not be overwritten by called pro- cedures
R17	\$s1	
R18	\$s2	
R19	\$s3	
R20	\$s4	
R21	\$s5	
R22	\$s6	
R23	\$s7	
R24	\$t8	Caller Save Temp
R25	\$t9	
R26	\$k0	Reserved for Operating Sys Global Pointer
R27	\$k1	
R28	\$gp	
R29	\$sp	Callee Save Stack Pointer
R30	\$fp	Frame Pointer
R31	\$ra	Return Address

**Callee Save**  
If the callee uses these register, then the callee must save *and* restore them in case the caller uses them.

# Saving registers on the stack

- MIPS has a special part of memory called The **Stack** for saving registers.
- The **Stack Pointer** (kept in `$sp` or R29) keeps the *address* of the end of the stack.  
In MIPS the stack grows **down**.
- Move the stack pointer down** when storing more data on the stack.
- Each procedure **returns the stack** to where it was before it was called.
- Gives procedures a secure place to store data that does not fit in registers. (e.g., saved registers!)
- Each procedure manages its own stack space so they don't interfere.
- Works great as long as you return the stack to the way it was before.





# How to move the stack pointer?

**Q: The stack pointer address is stored in R31 (\$sp). How much do we need to decrement it to make space for a register?**

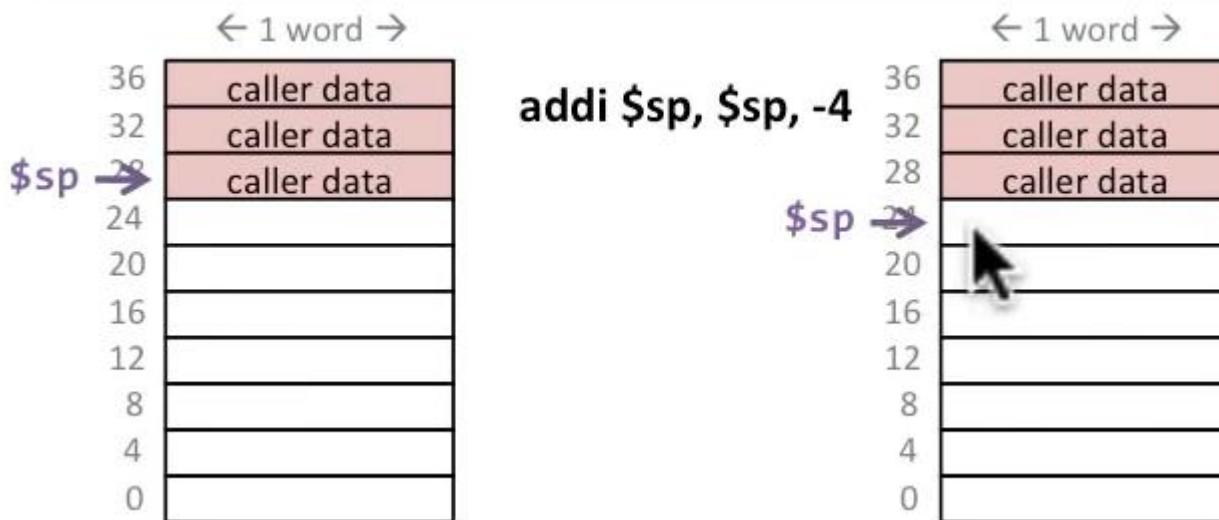
- ☐ 1
- ☐ 4
- ☐ 32
- ☐ Depends on the size of the register

**A: 4**

Each register is 4 bytes and each memory address is 1 byte. The Stack Pointer points to the last byte used on the stack, so to make room for a register (4 bytes) we need to move it down by 4 (4 bytes).

We do this by:

`addi $sp, $sp, -4`



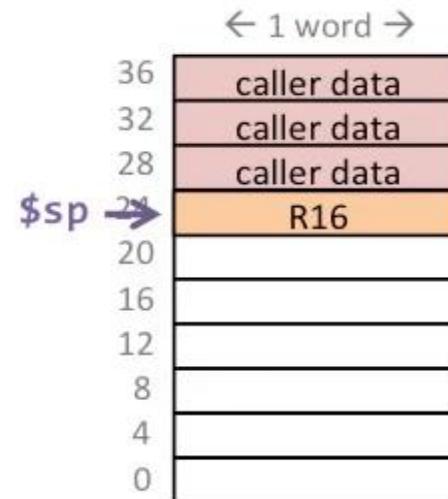
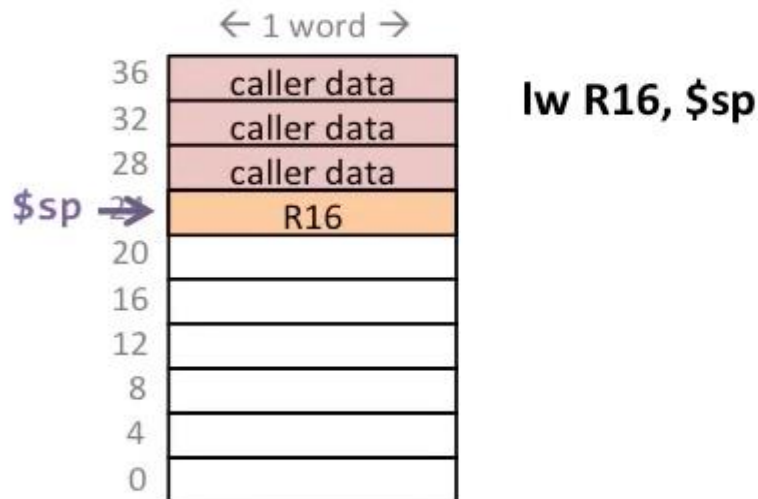
# How do we restore from the stack?

**Q: How do we restore R16 from the stack?**

- lw R16, \$sp
- lw \$sp, R16
- addi \$sp, \$sp, 4  
lw R16, \$sp
- lw R16, \$sp  
addi \$sp, \$sp, 4

**A:** lw R16, \$sp  
addi \$sp, \$sp, 4

First we load the data from where the stack pointer currently points with load word. Then we increment the stack pointer to point to the item before since we have now restored this item.



## Register File

R0:

R1:

...

R16:

R17:

...

R31:





- Put parameters in a place where the procedure can access them
- Transfer control to the procedure
- Acquire the storage resources needed for the procedure
- Perform the desired task
- Put the result in a place where the calling program can access it
- Return control to the point of origin, since a procedure can be called from several points in a program
- MIPS Software Convention to help 'Procedure Calls'
  - \$a0 - \$a3: four argument registers in which to pass parameters
  - \$v0 - \$v1: two value registers in which to return values
  - \$ra: one return address register to return to point of origin
- JAL (jump and link) instruction is especially meant for registers
  - *Jal proc\_addr*
- (book pg 103)

- Useful in procedure calls
- **\$sp** stack pointer is reference, changed by 4 after each read and write
- **sw** Save Word command is used to store s type registers in stack. \$sp is **subtracted by 4** after every step. All s registers are thus pushed to stack before procedure code is executed.
- After executing procedure code, **lw** Load Word command is used to load respective s type registers with stack data. \$sp is **incremented by 4** after every load step.

# Some Examples of MIPS Assembly Instructions

# Memory Operand Example 1

- C code:

```
g = h + A[8];
```

- g in \$s1, h in \$s2, base address of A in \$s3

- Compiled MIPS code:

- Index 8 requires offset of 32
  - 4 bytes per word

```
lw    $t0, 32($s3)    # load word
add   $s1, $s2, $t0
```

offset

base register

# Memory Operand Example 2



- C code:

`A[12] = h + A[8];`

- `h` in `$s2`, base address of `A` in `$s3`

- Compiled MIPS code:

- Index 8 requires offset of 32

```
lw    $t0, 32($s3)    # load word
add   $t0, $s2, $t0
sw    $t0, 48($s3)    # store word
```



- C code:

```
while (save[i] == k) i += 1;
```

- `i` in `$s3`, `k` in `$s5`, address of `save` in `$s6`

- Compiled MIPS code:

```
Loop:  sll    $t1, $s3, 2
       add    $t1, $t1, $s6
       lw     $t0, 0($t1)
       bne    $t0, $s5, Exit
       addi   $s3, $s3, 1
       j      Loop
Exit:  ...
```

# Procedure Example 1



- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Arguments g, ..., j in \$a0, ..., \$a3
- f in \$s0 (hence, need to save \$s0 on stack)
- Result in \$v0

# Leaf Procedure Example



- MIPS code:

leaf\_example:

addi \$sp, \$sp, -4

sw \$s0, 0(\$sp)

add \$t0, \$a0, \$a1

add \$t1, \$a2, \$a3

sub \$s0, \$t0, \$t1

add \$v0, \$s0, \$zero

lw \$s0, 0(\$sp)

addi \$sp, \$sp, 4

jr \$ra

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

#Save \$s0 on stack

#Procedure body

#Result

#Restore \$s0

#Return

# Compiling If Statements

- C code:

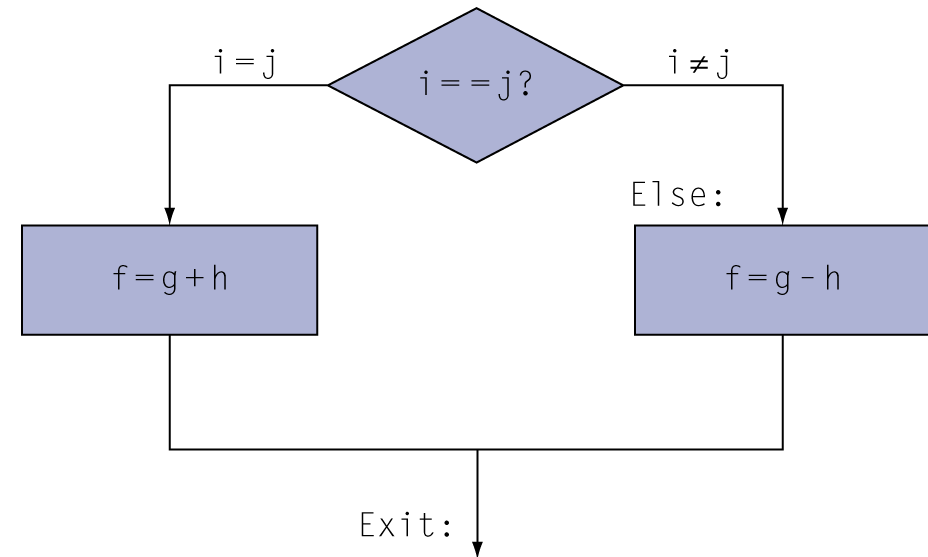
```
if (i==j) f = g+h;
else f = g-h;
```

- `f, g, ...` in `$s0, $s1, ...`

- Compiled MIPS code:

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j   Exit
Else:    sub $s0, $s1, $s2
Exit:    ...
```

Assembler calculates addresses



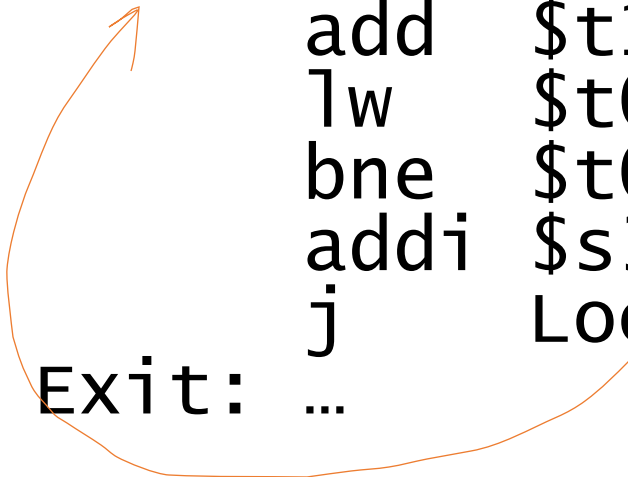
- C code:

```
while (save[i] == k) i += 1;
```

- `i` in `$s3`, `k` in `$s5`, address of `save` in `$s6`

- Compiled MIPS code:

```
Loop:  sll    $t1, $s3, 2  
       add    $t1, $t1, $s6  
       lw     $t0, 0($t1)  
       bne    $t0, $s5, Exit  
       addi   $s3, $s3, 1  
       j      Loop  
Exit:  ...
```





**2.2** [5] <\$2.2> For the following MIPS assembly instructions above, what is a corresponding C statement?

```
add  f, g, h
```

```
add  f, i, f
```

**2.3** [5] <§§2.2, 2.3> For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables *f*, *g*, *h*, *i*, and *j* are assigned to registers *\$s0*, *\$s1*, *\$s2*, *\$s3*, and *\$s4*, respectively. Assume that the base address of the arrays *A* and *B* are in registers *\$s6* and *\$s7*, respectively.

`B[8] = A[i-j];`

**2.29** [5] <§2.7> Translate the following loop into C. Assume that the C-level integer *i* is held in register *\$t1*, *\$s2* holds the C-level integer called *result*, and *\$s0* holds the base address of the integer *MemArray*.

```
        addi $t1, $0, $0
LOOP:   lw    $s1, 0($s0)
        add   $s2, $s2, $s1
        addi  $s0, $s0, 4
        addi  $t1, $t1, 1
        slti  $t2, $t1, 100
        bne   $t2, $s0, LOOP
```

# Interrupts in MIPS

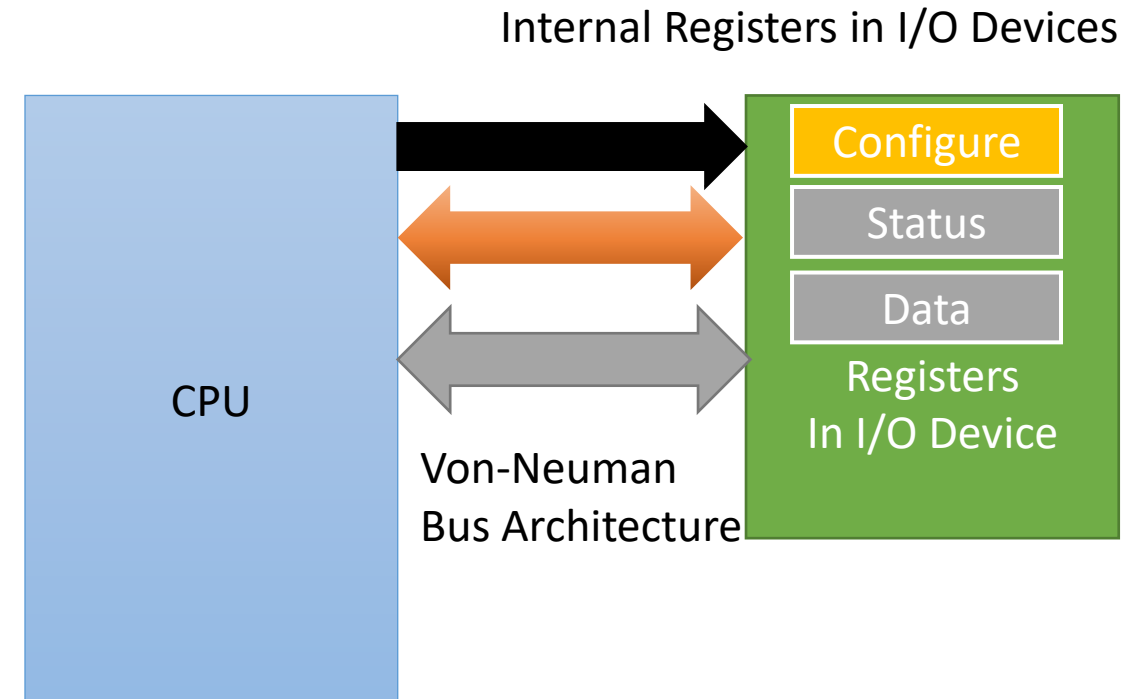
## What is interrupt processing?

An interrupt is an event that alters the sequence in which the processor executes instructions. An interrupt might be planned (specifically requested by the currently running program) or unplanned (caused by an event that might or might not be related to the currently running program).

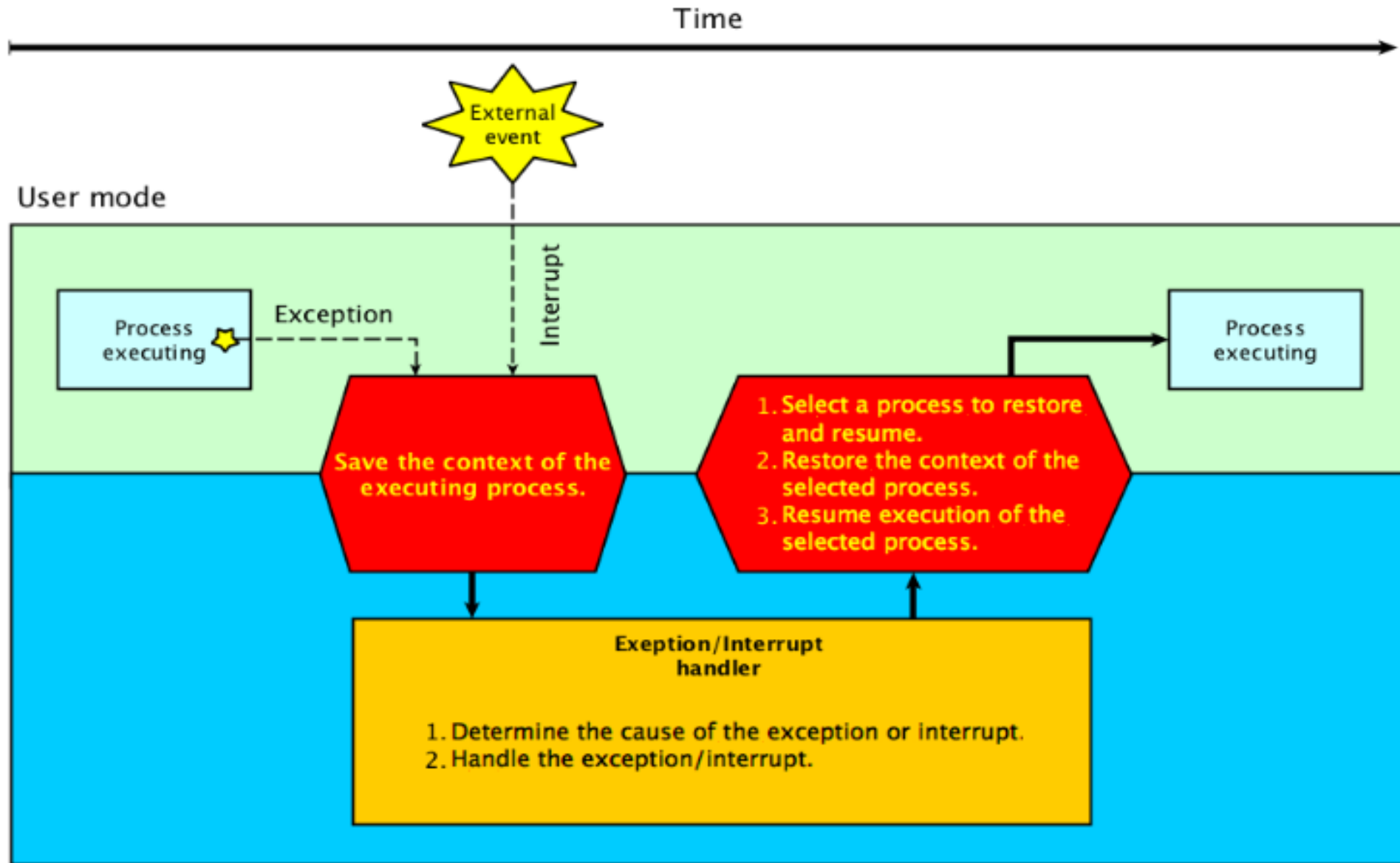
- Supervisor calls or SVC interrupts
- I/O interrupts
- External interrupts
- Restart interrupts
- Program interrupts
- Machine check interrupts



- **Interrupt Driven** vs **Polled Approach**
- **Configuration Register** in I/O Device
- Status Register in I/O Device
- Interrupt **Masking**
- Interrupt **Priority Encoding**



# Exception/ISR sequence



- It is sometimes useful to disable some interrupts. This is called “masking” the interrupt.
- Microcontrollers can be programmed to ignore some interrupt sources. This is achieved by not setting a special bit called the interrupt's enable bit.
- The process of ignoring an interrupt is called masking.
- Interrupts that can be enabled/disabled are called maskable.
- Each maskable interrupt has its own corresponding enable or mask bit
- Typically, an enable/mask bit is set or cleared by writing to a specific I/O control or status register
- When an enable bit is set to 1, the CPU will respond to the interrupt request, when is set to 0, the CPU will ignore the request
- When a mask bit is set to 1, the CPU will ignore the interrupt request, when is set to 0, the CPU will respond to the request.

Usually there are several Interrupt Lines in a CPU. The interrupt lines are assigned a priority so that Most critical interrupts are first processed while others are ignored.

# Interrupts vs Exceptions

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

# Synchronous and Asynchronous ISR/Exceptions



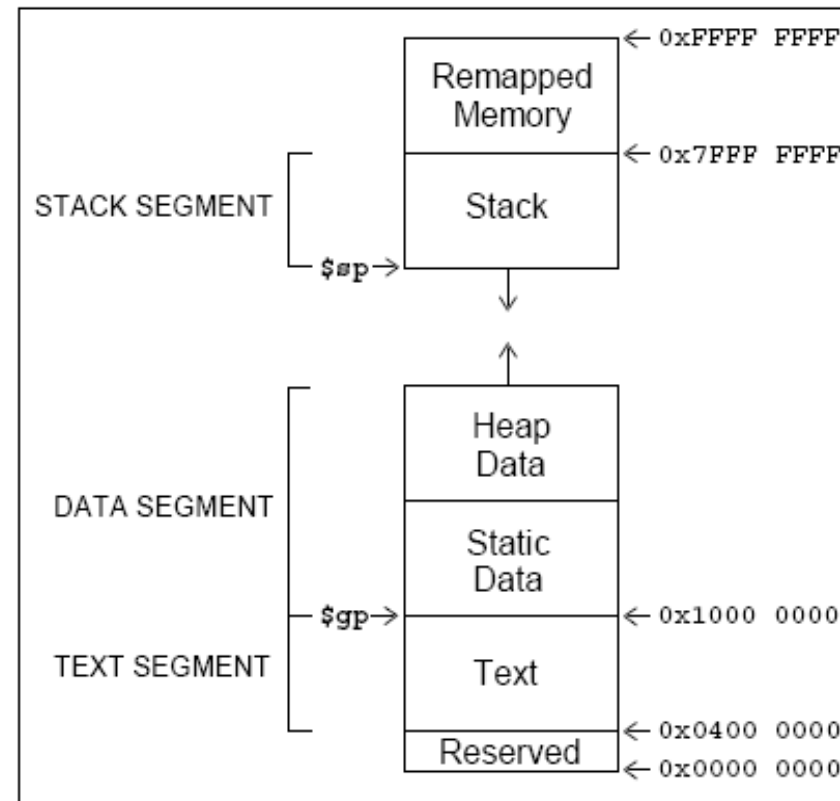
An exception is said to be synchronous if it occurs at the same place every time a program is executed with the same data and the same memory allocation. Arithmetic overflows, undefined instructions, page faults are some examples of synchronous exceptions. Asynchronous exceptions, on the other hand, happen with no temporal relation to the program being executed. I/O requests, memory errors, power supply failure are examples of asynchronous events.

An *interrupt* is an asynchronous exception. Synchronous exceptions, resulting directly from the execution of the program, are called *traps*.



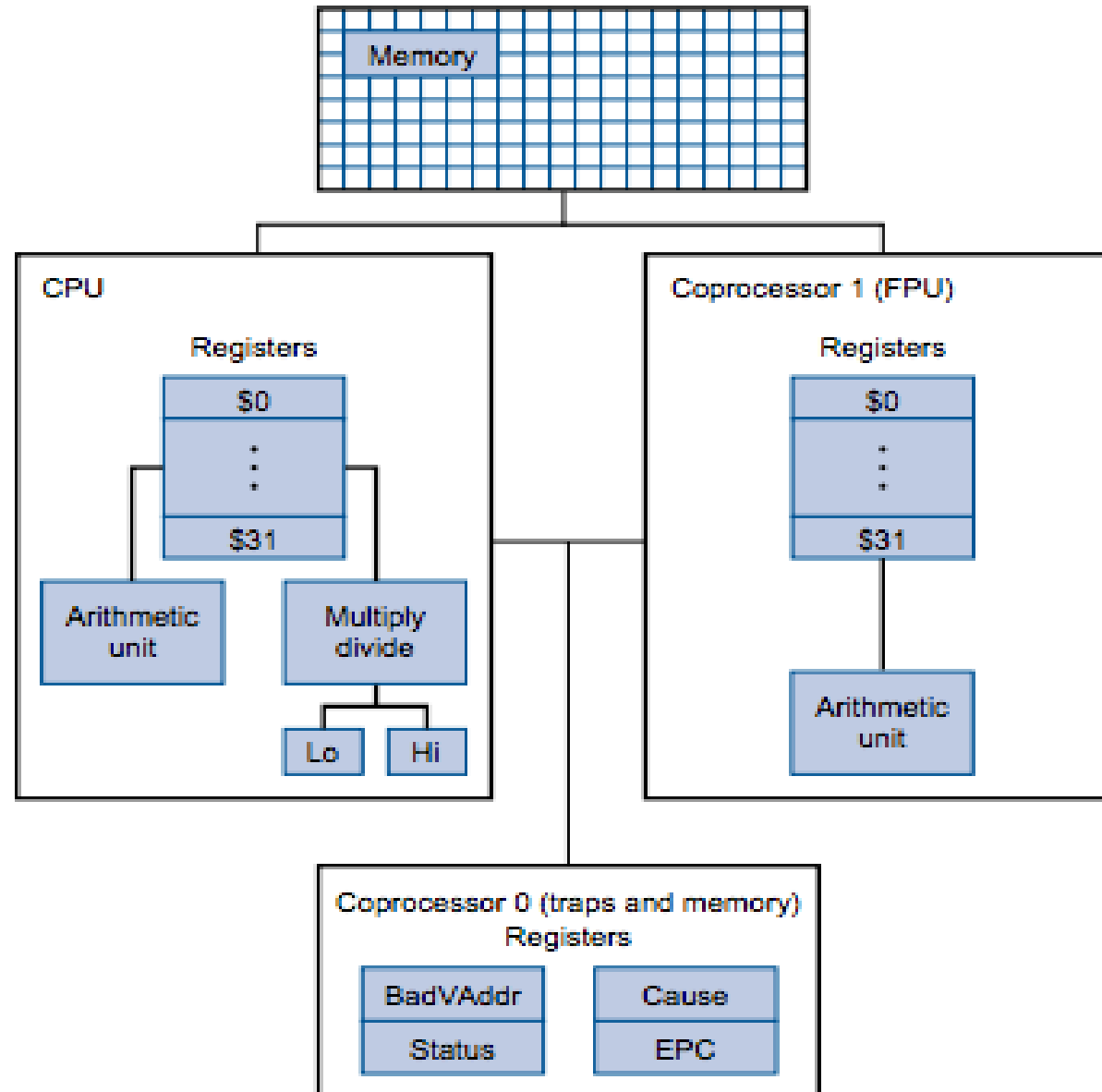
# Stack memory in MIPS memory map

- Actually, everything above 0x7ffffff is used by the system.



- Through **Co-Processor 0** and control through Register Read / Write
- Registers Visible in QtSPIM Assembler:
  - **EPC**
  - **Cause** Register informs reasons for Exception / Interrupts
  - **BVAddr** Bad value address for Exception handling
  - **Status** Register used for Masking / Enabling / Disabling Interrupts
  - **mfc0** and **mtc0** instructions to **read** and **write** to **Co-Proc 0** registers

# MIPS Coprocessor 0 for Exception Processing



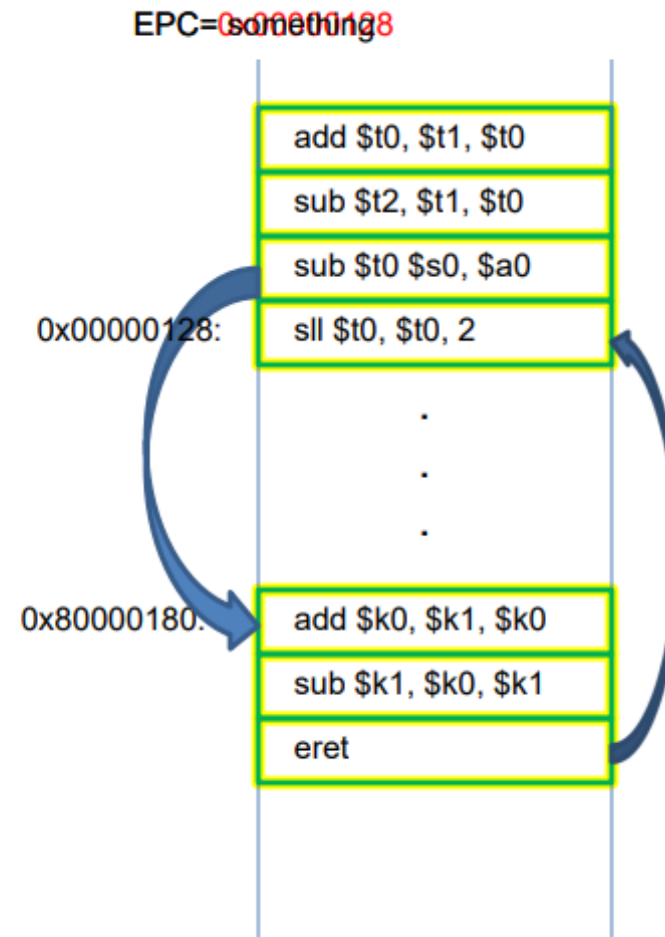
## The MIPS exception mechanism

The exception mechanism is implemented by the coprocessor 0 which is always present (unlike coprocessor 1, the floating point unit, which may or may not be present). The virtual memory system is also implemented in coprocessor 0. Note however that SPIM does not simulate this part of the coprocessor.

The CPU operates in one of the two possible modes, **user** and **kernel**. User programs run in user mode. The CPU enters the kernel mode when an exception happens. Coprocessor 0 can only be used in kernel mode.

The whole upper half of the memory space is reserved for the kernel mode: it can not be accessed in user mode. When running in kernel mode the registers of coprocessor 0 can be accessed using the following instructions:

- With external interrupt, if an event happens that must be processed, the following things will happen:
  - The address of the instruction that is about to be executed is saved into a special register called EPC
  - PC is set to be 0x80000180, the starting address of the interrupt handler
    - which takes the processor to the interrupt handler
  - The last instruction of the interrupt should be “eret” which sets the value of the PC to the value stored in EPC



## Instructions which access the registers of coprocessor 0

Instruction	Comment
<code>mfc0 Rdest, C0src</code>	Move the content of coprocessor's register <code>C0src</code> to <code>Rdest</code>
<code>mtc0 Rsrc, C0dest</code>	Integer register <code>Rsrc</code> is moved to coprocessor's register <code>C0dest</code>
<code>lwc0 C0dest, address</code>	Load word from <code>address</code> in register <code>C0dest</code>
<code>swc0 C0src, address</code>	Store the content of register <code>C0src</code> at <code>address</code> in memory

## Exception handling registers in coprocessor 0

Register Number	Register Name	Usage
8	BadVAddr	Memory address where exception occurred
12	Status	Interrupt mask, enable bits, and status when exception occurred
13	Cause	Type of exception and pending interrupt bits
14	EPC	Address of instruction that caused exception



## The BadVAddr register

This register (its name stands for **Bad Virtual Address**) will contain the memory address where the exception has occurred. An unaligned memory access, for instance, will generate an exception and the address where the access was attempted will be stored in BadVAddr.

## The EPC register

When a procedure is called using `jal`, two things happen:

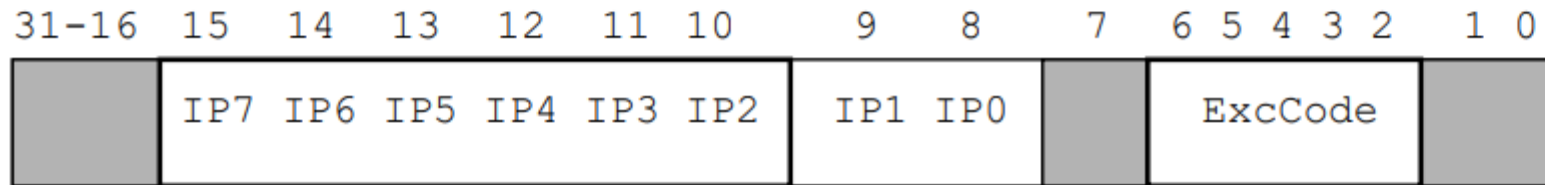
- control is transferred at the address provided by the instruction
- the return address is saved in register **\$ra**

In the case of an exception there is no explicit call. In MIPS the control is transferred at a fixed location, `0x80000080` when an exception occurs. The exception handler must be located at that address.

The return address can not be saved in **\$ra** since it may clobber a return address that has been placed in that register before the exception. The Exception Program Counter (EPC) is used to store the address of the instruction that was executing when the exception was generated.

## The Cause register

The Cause register provides information about what interrupts are pending (IP2 to IP7) and the cause of the exception. The exception code is stored as an unsigned integer using bits 6-2 in the Cause register. The layout of the Cause register is presented below.



Bit  $IP_i$  becomes 1 if an interrupt has occurred at level  $i$  and is pending (has not been serviced yet). The bits  $IP_1$  and  $IP_0$  are used for simulated interrupts that can be generated by software. Note that  $IP_1$  and  $IP_0$  are not visible in SPIM (please refer to the SPIM documentation).

# Exception Codes in QtSPIM



The exception code indicates what caused the exception.

## Exception codes<sup>a</sup> implemented by SPIM

Code	Name	Description
0	INT	Interrupt
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	<code>syscall</code> instruction executed
9	BKPT	<code>break</code> instruction executed
10	RI	Reserved instruction
12	OVF	Arithmetic overflow

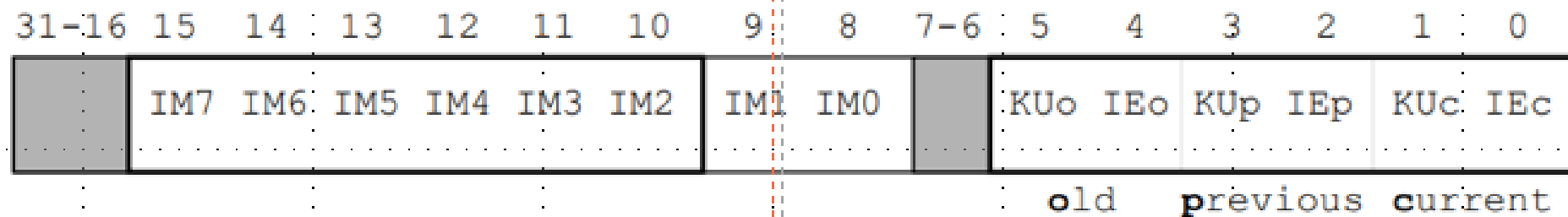
a. Codes from 1 to 3 are reserved for virtual memory, (TLB exceptions), 11 is used to indicate that a particular coprocessor is missing, and codes above 12 are used for floating point exceptions or are reserved.

Code 0 indicates that an interrupt has occurred. By looking at the individual `IPi` bits the processor can learn what specific interrupt happened.

## The Status register

The Status register contains an interrupt mask on bits 15-10 and status information on bits 5-0. The layout of

the Status register is presented below:



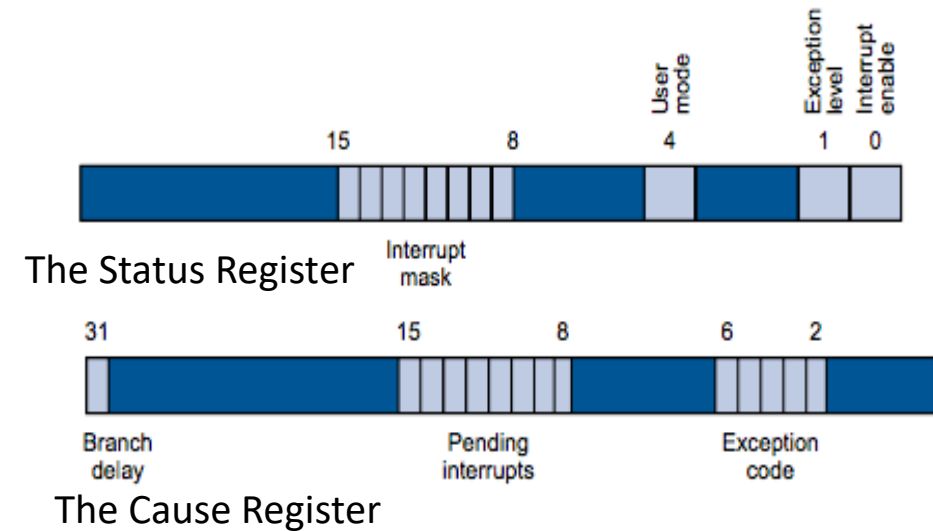
## Returning from exception

The return sequence is standard:

```
#  
# the return sequence from the exception handler for the case of an  
# external exception (interrupt)  
#  
    mfc0 $k0, $14      # get EPC in $k0  
    rfe                # return from exception  
    jr $k0             # replace PC with the return address ■
```

# Exception Status and Cause Registers in MIPS

- Status Register (number 12) is **Read / Write** Register
- Cause Register is a **Read** Register
- **Interrupt Mask** Field contains a bit for each of the six hardware and two software interrupts  
 '1' allows respective interrupt, '0' disables
- When an interrupt arrives, it sets its Interrupt Pending bit in the Cause Register. ISR will proceed further if respective bit is enabled
- User mode bit is '0' if the processor is in kernel mode and '1' in user mode.
- Exception bit is normally '0' but goes to '1' when an exception occurs. Interrupts are disabled in this state.
- Interrupts are allowed when **Interrupt Enable** bit is '1'



Exception Code	Name	Cause of Exception
0	Int	Interrupt (hardware)
4	AdEL	Address Error Exeption (Load or Instruction fetch)
5	AdES	Address Error exception (Store)
6	IBE	Instruction fetch Buss Error
7	DBE	Data load or store Buss Error
8	Sys	Syscall exception
9	Bp	Breakpoint exception
10	RI	Reversed Instruction exception
11	CpU	Coprocessor Unimplemented
12	Ov	Arithmetic Overflow exception
13	Tr	Trap
14	FPE	Floating Point Exception