Computer Organization and Design Processor

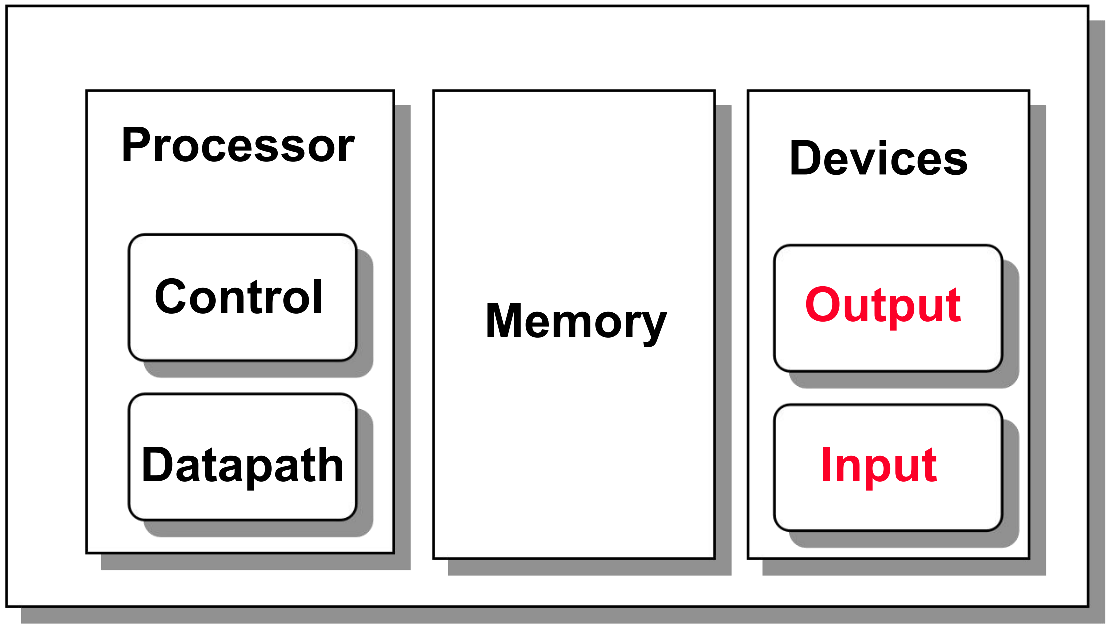
Exceptions and Interrupts

[adapted from Mary Jane Irwin slides]

Reading assignment

- Exceptions and Interrupts
  - PH(3): **5.6** (Exception for multicycle datapath),
    8.1, 8.5 (I/O system)
  - PH(5): Appendix A Assemblers, Linkers, and the SPIM simulator **A.7-A.8- A.9** (Exception and I/O system for SPIM)

Major Components of a Computer



- Important metrics for an I/O system
  - Performance
  - Compatibility
  - Expandability and diversity
  - Dependability
  - Cost, size, weight

Input and Output Devices

- I/O devices are incredibly diverse with respect to
  - Behavior – input, output or storage
  - Partner – human or machine
  - Data rate – the peak rate at which data can be transferred between the I/O device and the main memory or processor

Device	Behavior	Partner	Data rate (Mb/s)
Keyboard	input	human	0.0001
Mouse	input	human	0.0038
Laser printer	output	human	3.2000
Network/LAN	input or output	machine	100.0000-1000.0000
Magnetic disk	storage	machine	240.0000-2560.0000
Graphics display	output	human	800.0000-8000.0000

8 orders of magnitude range

Input/Output in SPIM via System Calls

- SPIM provides a small set of operating-system-like services through the `syscall` instruction
  - Load the system call code into register `$v0` and the arguments into registers `$a0` through `$a3`
  - Return values are put in register `$v0`

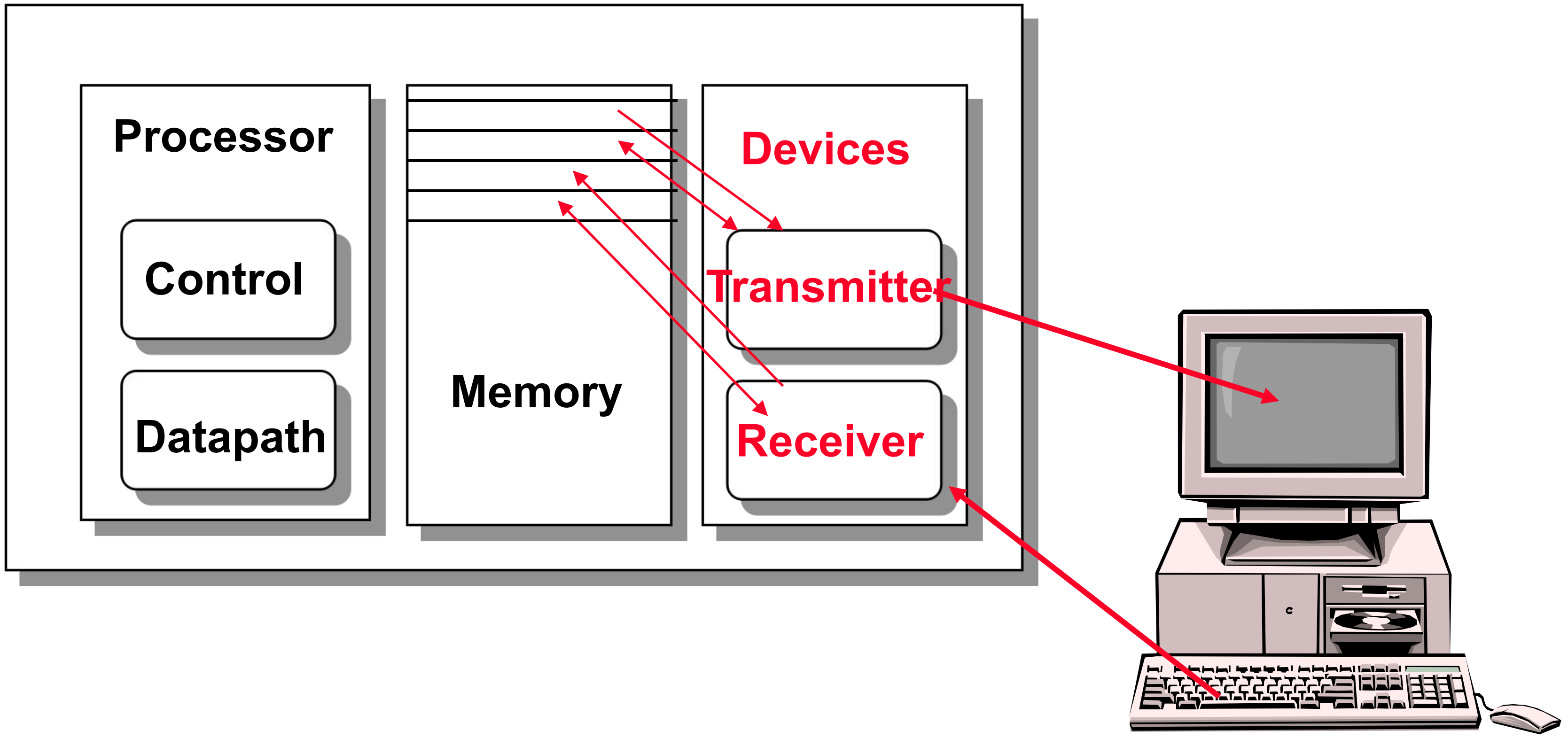
Service	Code	Args	Results
`print_int`	1	`$a0` = integer	
`print_string`	4	`$a0` = string	
`read_int`	5		integer in `$v0`
`read_string`	8	`$a0` = buffer, `$a1` = length	
`print_char`	11	`$a0` = char	
`read_char`	12		char in `$v0`

Communication of I/O Devices and Processor

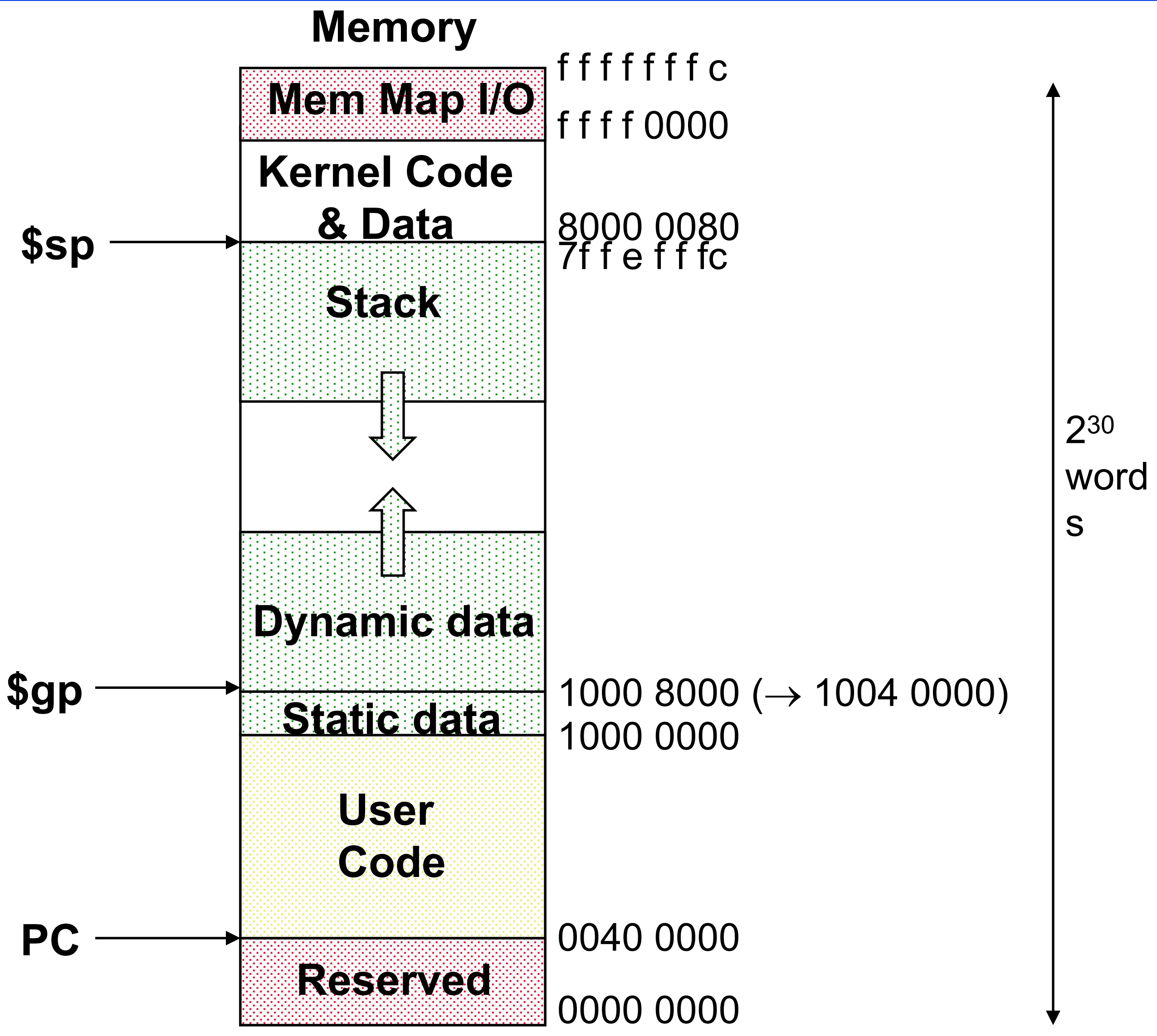
- How the processor directs the I/O devices
  - Special I/O instructions
    - Must specify both the device and the command
  - Memory-mapped I/O
    - Portions of the high-order memory address space are assigned to each I/O device. Read (`lw`) and writes (`sw`) to those memory addresses are interpreted as commands to the I/O devices
    - Load/stores to the I/O address space done only by the OS
- How the I/O device communicates with the processor
  - Polling – the processor periodically checks the status of an I/O device to determine its need for service
    - Processor is totally in control – but does all the work
    - Can waste a lot of processor time due to speed differences
  - Interrupt-driven – the I/O device issues an interrupts to the processor to indicate that it needs attention

“Real” I/O in SPIM

- SPIM supports one memory-mapped I/O device – a terminal with two independent units
  - Transmitter writes characters to the display
  - Receiver reads characters from the keyboard

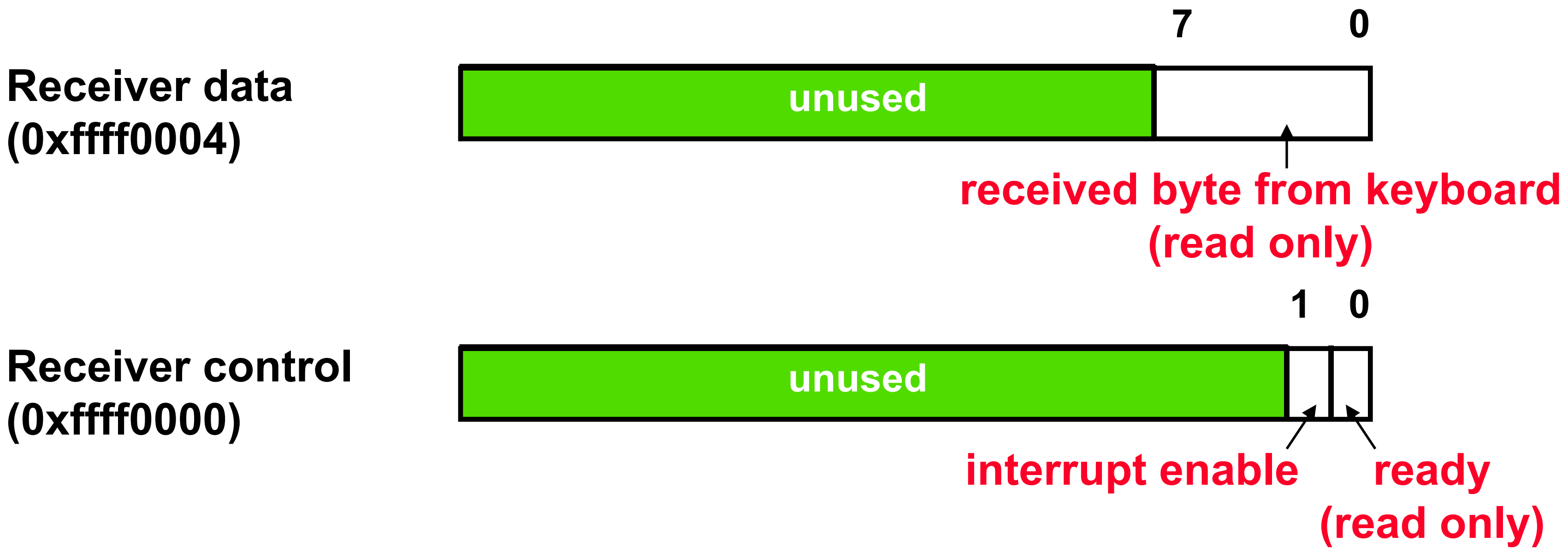


Review: MIPS (spim) Memory Allocation



Terminal Receiver (Input) Control with SPIM

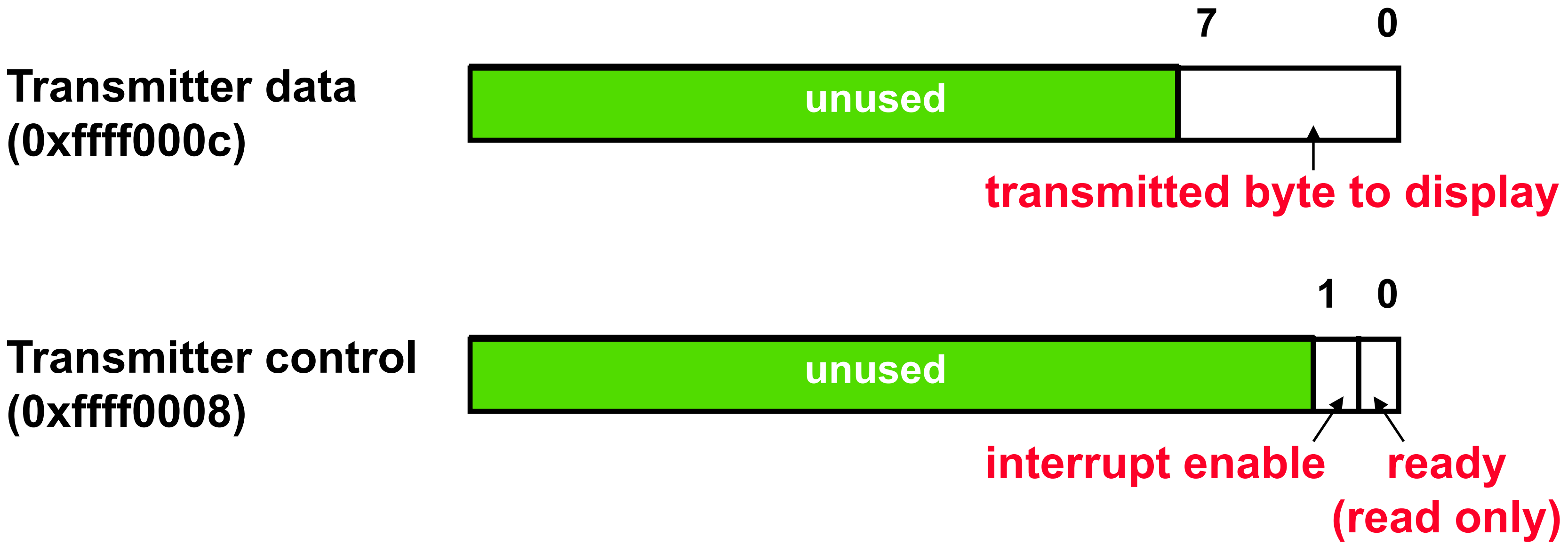
- Input is controlled via two memory-mapped device registers (i.e., each is a special memory location)



- The keyboard inputs into the Receiver data register which sets the ready bit in the Receiver control register (i.e., the keyboard input is ready to be read by the program)
- Reading the input character from the Receiver data register resets the ready bit in the Receiver control register

Terminal Output Control with SPIM

- Output is controlled via two memory-mapped device registers (i.e., each is a special memory location)



  - The display outputs the Transmitter data register character which sets the ready bit in the Transmitter control register (i.e., the display is ready to accept a new output character)
  - Writing the next character to output into the Transmitter data register resets the ready bit in the Transmitter control register

MIPS I/O Instructions

- MIPS has 2 coprocessors: Coprocessor 0 handles exceptions including input and output interrupts, Coprocessor 1 handles floating point
  - Coprocessors have their own register sets so have instructions to move values between these registers and the CPU's registers

Register	#	Use
BadVAddr	8	bad mem addr
Count	9	timer
Compare	11	timer compare
Status	12	intr mask & enable bits
Cause	13	excp type and pending intr's
EPC	14	addr of instr causing excp

```
mfc0  rd, rt  #move from coprocessor 0
```

0x10	0	rt	rd	0	0

```
mtc0  rt, rd  #move to coprocessor 0
```

0x10	4	rt	rd	0	0

Polling in SPIM

- Be sure that memory-mapped I/O is enabled (through the PCSpim “Settings” dialog box)

```
        li      $t0, 0xffff0000     #recv ctrl
        li      $t1, 0xffff0004     #recv data
        li      $t2, 0xffff0008     #trans ctrl
        li      $t3, 0xffff000c     #trans data

        mtc0    $zero, $12          #disable interrupts

I1:     lw      $t4, 0($t0)         #poll recv ready bit
        andi    $t4, $t4, 1
        beq     $t4, $zero, I1      #loop til recv ready
        lw      $t6, 0($t1)         #read input character

I2:     lw      $t4, 0($t2)         #poll trans ready bit
        andi    $t4, $t4, 1
        beq     $t4, zero, I2       #loop til trans ready
        sw      $t6, 0($t3)         #echo (print) character
```

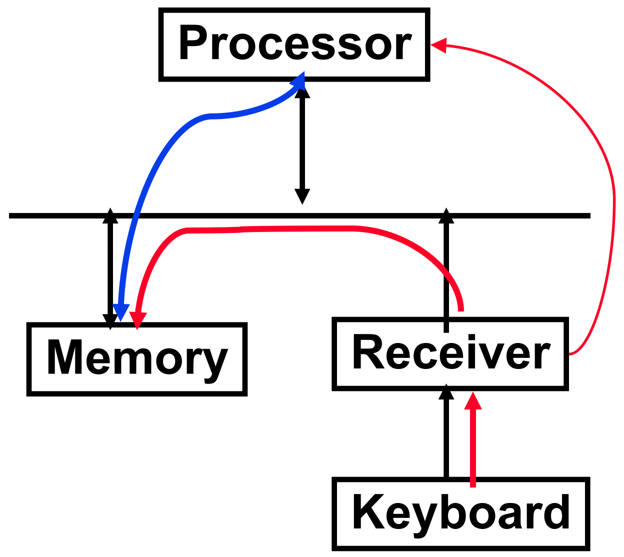
The Downsides of Polling

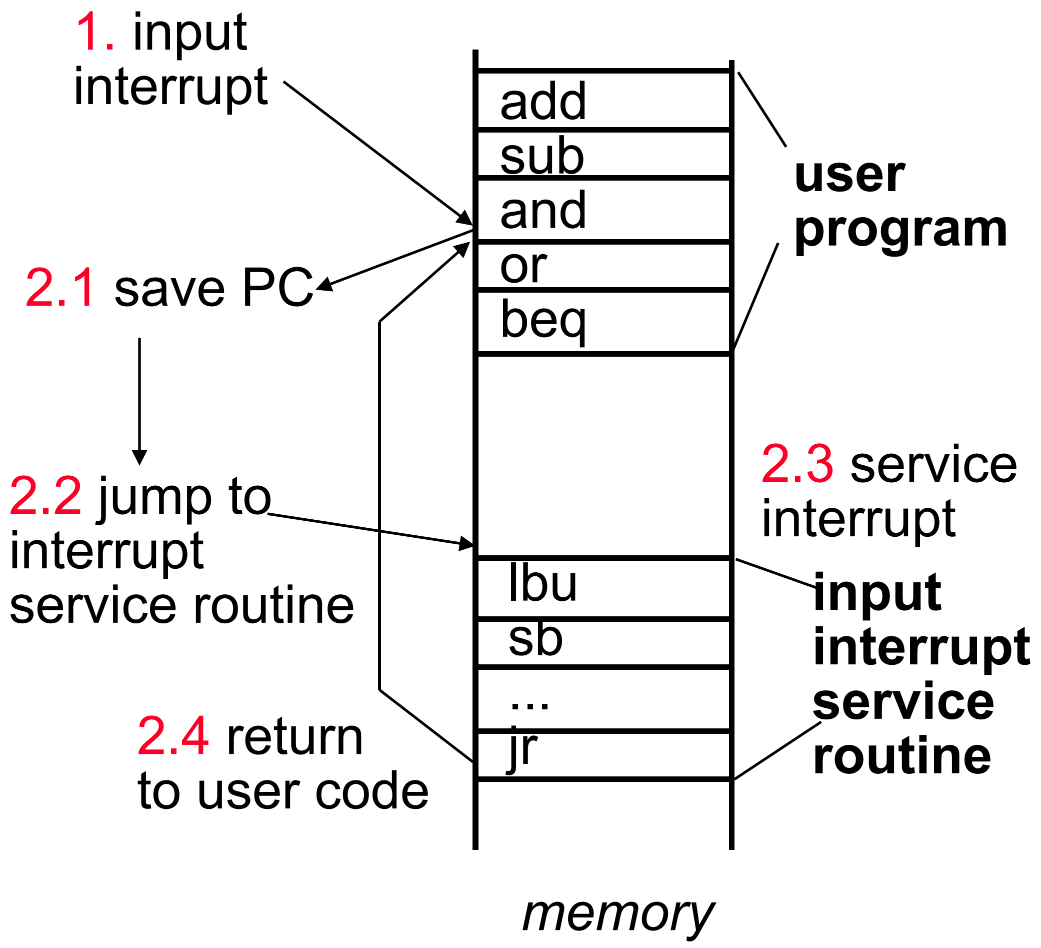
- Input and output devices are *very* slow compared to the processor
  - These time lags are simulated in SPIM which measures time in instructions executed, not in real clock time
  - After the transmitter starts to write a character, the transmitter's ready bit becomes 0. It doesn't become ready again until the processor has executed a (large) fixed number of instructions. (You don't want to single step the simulator!)
- Polling will execute the "loop til ready" code thousands of times. While the input or output is occurring, nothing else can be done – a waste of resources.
- There *is* a better way--interrupt

I/O Interrupts

- An I/O interrupt is used to signal an I/O request for service
  - Can have different urgencies (so may need to be prioritized)
  - Need to identify the device generating the interrupt
- An I/O interrupt is asynchronous wrt instr execution
  - An I/O interrupt is not associated with any instruction and does not prevent any instruction from completion
    - You can pick your own convenient point to take an interrupt
- Advantage
  - User program progress is only halted during the actual transfer of I/O data to/from user memory space
- Disadvantage – special hardware is needed to
  - Cause an interrupt (I/O device)
  - Detect an interrupt and save the proper information to resume after servicing the interrupt (processor)

Interrupt Driven Input

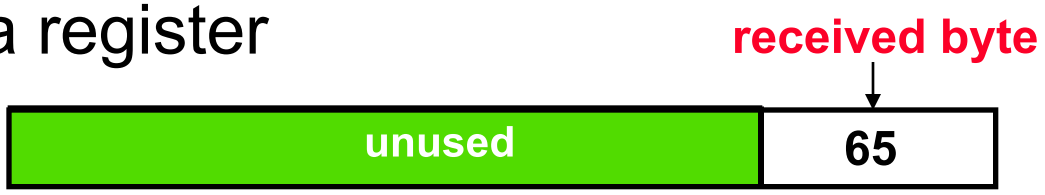




Interrupt Driven Input in SPIM

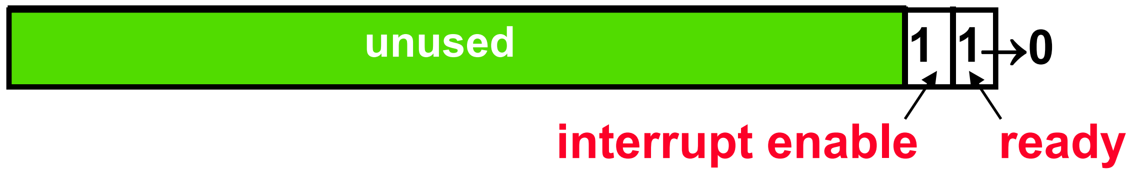
1. the Receiver indicates with an interrupt that it has input a new character from the keyboard into the Receiver data register

Receiver data (0xffff0004)



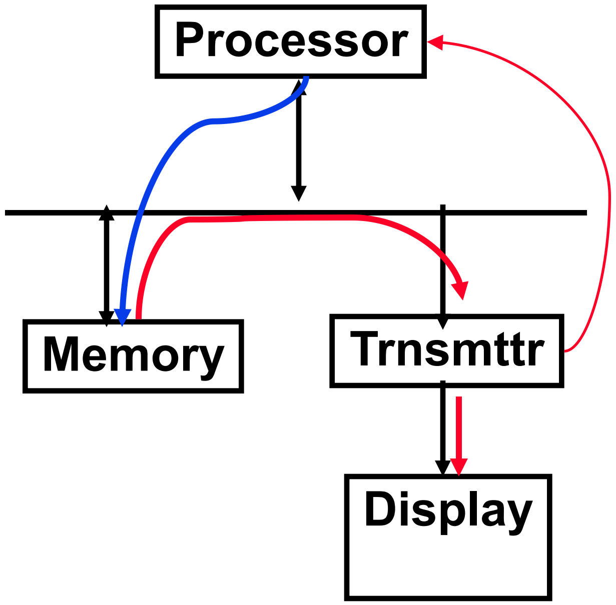
   - writing to the Receiver data register sets the Receiver control register ready bit to 1

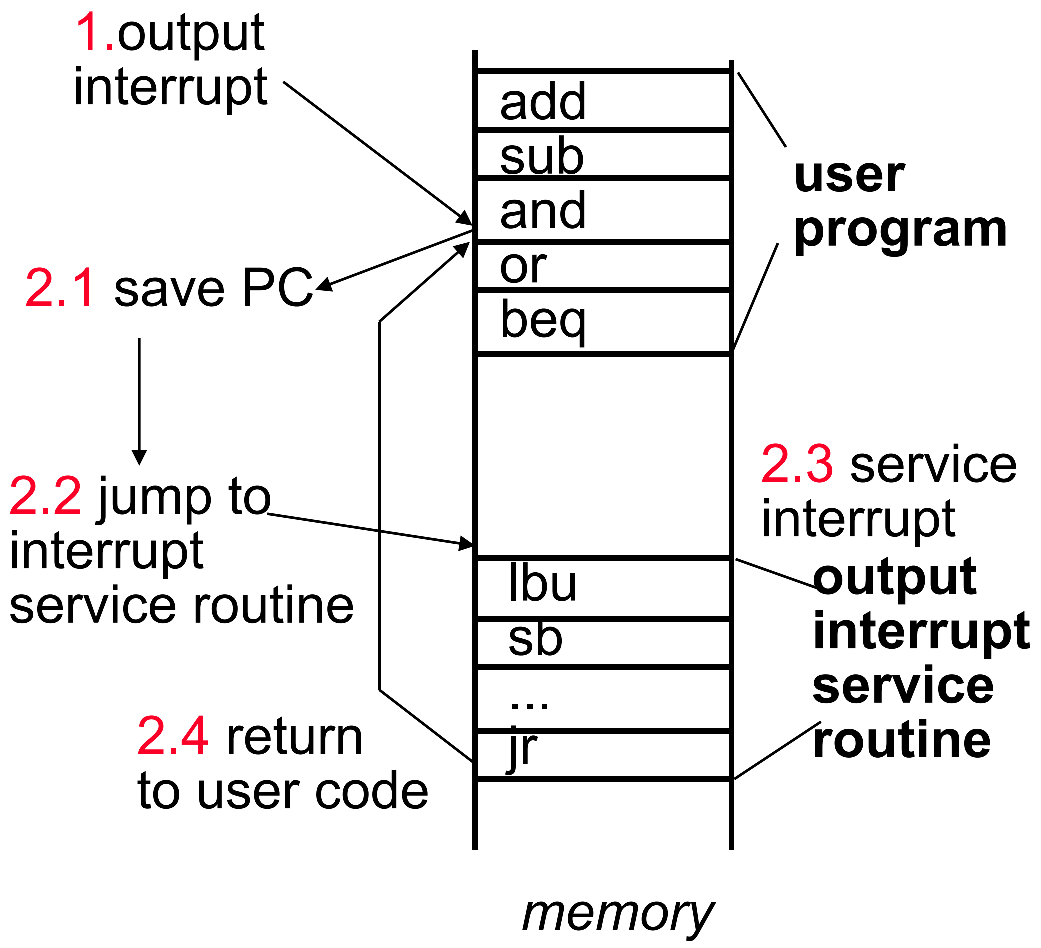
Receiver control (0xffff0000)



2. the user process responds to the interrupt by transferring control to an interrupt service routine that copies the input character into the user memory space
   - reading the Receiver data register resets the Receiver control register ready bit to 0

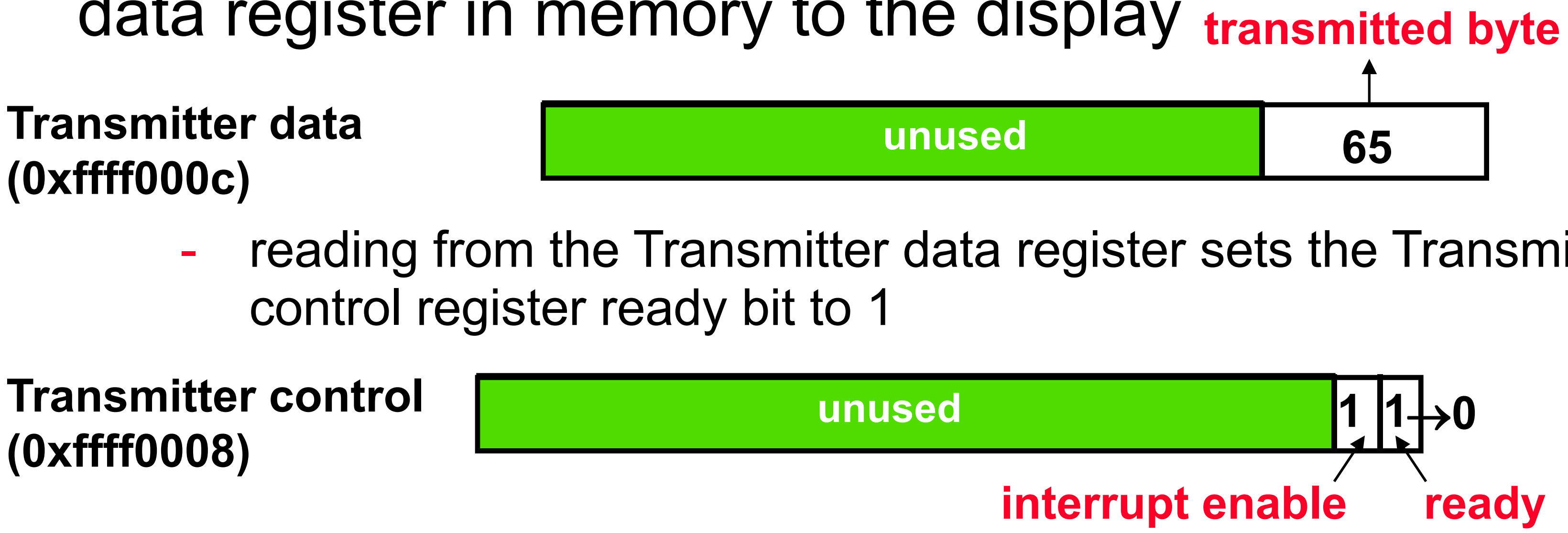
Interrupt Driven Output





Interrupt Driven Output in SPIM

1. the transmitter indicates with an interrupt that it has successfully output the character in the Transmitter data register in memory to the display

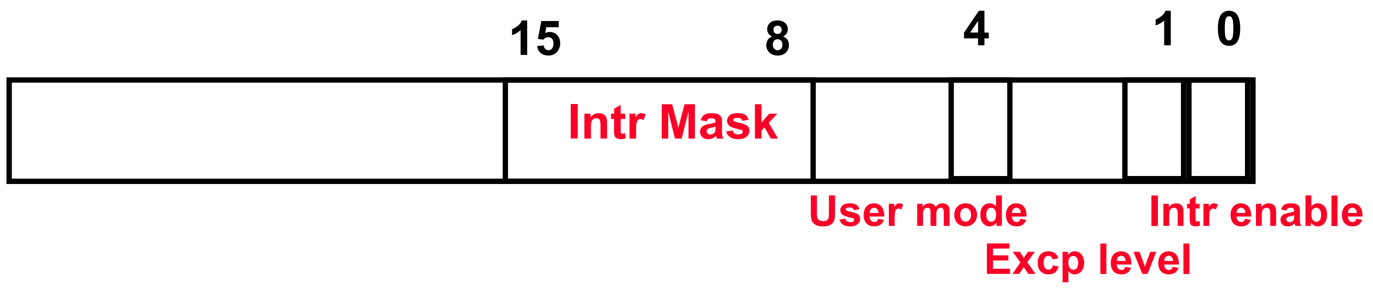


   - reading from the Transmitter data register sets the Transmitter control register ready bit to 1

2. the user process responds to the interrupt by transferring control to an interrupt service routine that writes the next character to output from the user memory space into the Transmitter data register
   - writing to the Transmitter data register resets the Transmitter control register ready bit to 0

Additions to MIPS ISA for I/O

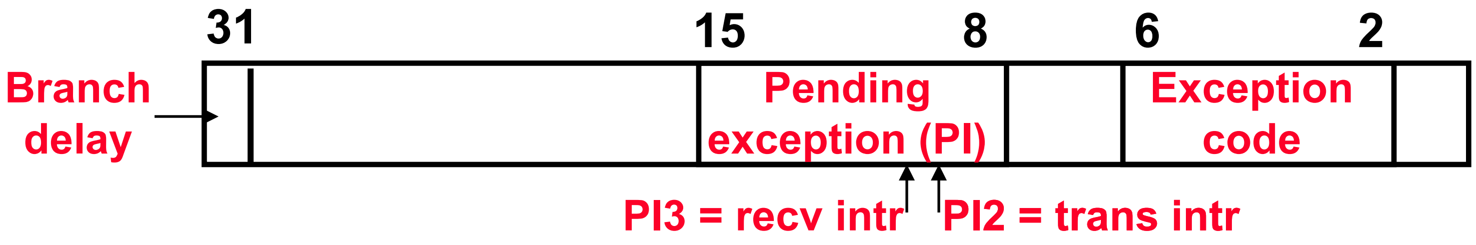
- Coprocessor 0 records the information the software needs to handle exceptions (including interrupts)
  - EPC (register 14) – holds the address+4 of the instruction that was executing when the exception occurred
  - Status (register 12) – exception mask and enable bits



- Intr Mask = 1 bit for each of 6 hardware and 2 software exception levels (1 enables exception at that level, 0 disables them)
- User mode = 0 if running in kernel mode when exception occurred; 1 if running in user mode (fixed at 1 in SPIM)
- Excp level = set to 1 (disable exceptions to avoid nesting calling) when an exception occurs; should be reset by exception handler when done
- Intr enable = 1 if exception are enabled; 0 if disabled

Additions to MIPS ISA, Con't

- Cause (register 13) – exception pending and type bits



- PI: bits set if exception occurs but not yet serviced
  - so can records more than one exception occurring at same time, even when exception are disabled
- Exception code: encodes reasons for exception
  - 0 (INT) → external interrupt (I/O device request)
  - 4 (AdEL) → address error trap (load or instr fetch)
  - 5 (AdES) → address error trap (store)
  - 6 (IBE) → bus error on instruction fetch trap
  - 7 (DBE) → bus error on data load or store trap
  - 8 (Sys) → syscall trap
  - 9 (Bp) → breakpoint trap
  - 10 (RI) → reserved (or undefined) instruction trap
  - 12 (Ov) → arithmetic overflow trap

MIPS Exception Return Instruction

- Exception return – sets the Excp level bit in coprocessor 0's Status register to 0 (reenabling exception) and returns to the instruction pointed to by coprocessor 0's EPC register

```
eret        #return from exception
```

0x10	1	0	0	0	0x18

Example I/O Interrupts in SPIM - Enable

```
    li      $t0, 0xffff0000      #recv ctrl
    li      $t1, 0xffff0004      #recv data
    li      $t2, 0xffff0008      #trans ctrl
    li      $t3, 0xffff000c      #trans data

    mfc0    $t4, $13
    andi    $t4, $t4, 0xffff00ff       #clear Pending
interrupt
    mtc0    $t4, $13             #(PI) bits in Cause reg

    li      $t4, 0x2
    sw      $t4, 0($t0)          #enable recv interrupts
    sw      $t4, 0($t2)          #enable trans interrupts

    mfc0    $t4, $12
    ori     $t4, $t4, 0xff01     #enable intr and mask
    mtc0    $t4, $12             #in Status reg

    #do something useful while I/O is taking place
    #when I/O interrupts occur transfer control to
    #exception handler (at address 0x80000180)
```

Example I/O Interrupts in SPIM - Handler

```
      .ktext 0x80000180
      mfc0   $t4, $13              #get ExcpCode from Cause
      srl    $t5, $t4, 2
      andi   $t5, $t5, 0x1f        #ExcpCode in $t5, if 0
      bne    $t5, $zero, excp      #then I/O intr has occurred

ck_recv:
      andi   $t5, $t4, 0x800       #check for PI3 (input),
      beq    $t5, $zero, ck_trans        #if 0,then trans
  intr

I1:   lw     $t5, 0($t0)           #check recv ready
      andi   $t5, $t5, 1
      beq    $t5, $zero, no_recv_ready
      lw     $t6, 0($t1)           #input character into $t6
      li     $t7  1                #sign of input=1
      andi   $t4, $t4, 0xffff7ff         #clear PI3 bit in
  Cause reg
      mtc0   $t4, $13
```

Example I/O Interrupts in SPIM – Handler, con't

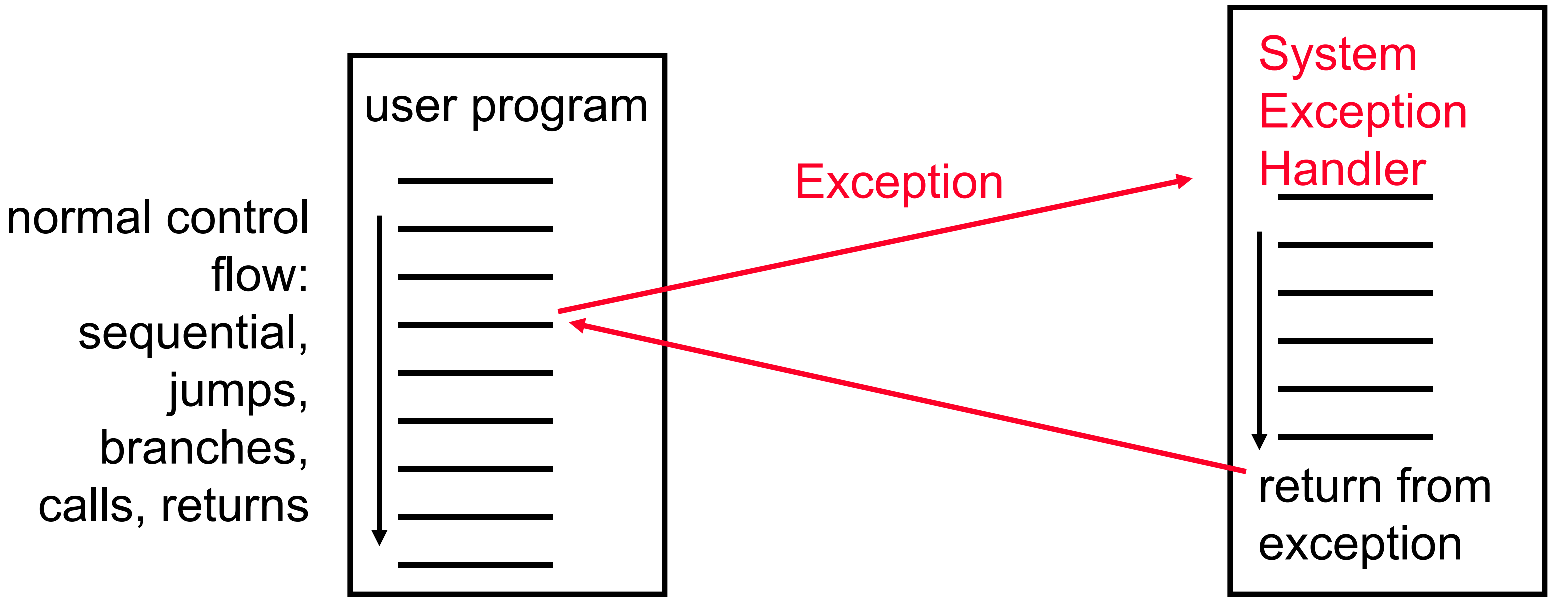
```
ck_trans:
        beq    $t7, $zero, ret_hand          #no character to
  echo yet
        andi   $t5, $t4, 0x400          #check for PI2 (output)
        beq    $t5, $zero, ret_hand          #if 0, then no trans
  intr

I2:     lw     $t5, 0($t2)              #check trans ready
        andi   $t5, $t5, 1
        beq    $t5, $zero, no_trans_ready
        sw     $t6, 0($t3)              #echo character to display
        mfc0   $t4, $13
        andi   $t4, $t4, 0xfffffbff          #clear PI2 bit in
  Cause reg
        mtc0   $t4, $13

ret_hand:
        mfc0   $t4, $12
        ori    $t4, $t4, 0xff01         #enable intr and mask
        mtc0   $t4, $12                 #in Status reg

        eret                            #return from intr
```

Exceptions in General



- Exception = unprogrammed control transfer
  - system takes action to handle the exception
    - must record the address of the offending or next to execute instruction and save (and restore) user state
  - returns control to user after handling the exception

Two Types of Exceptions

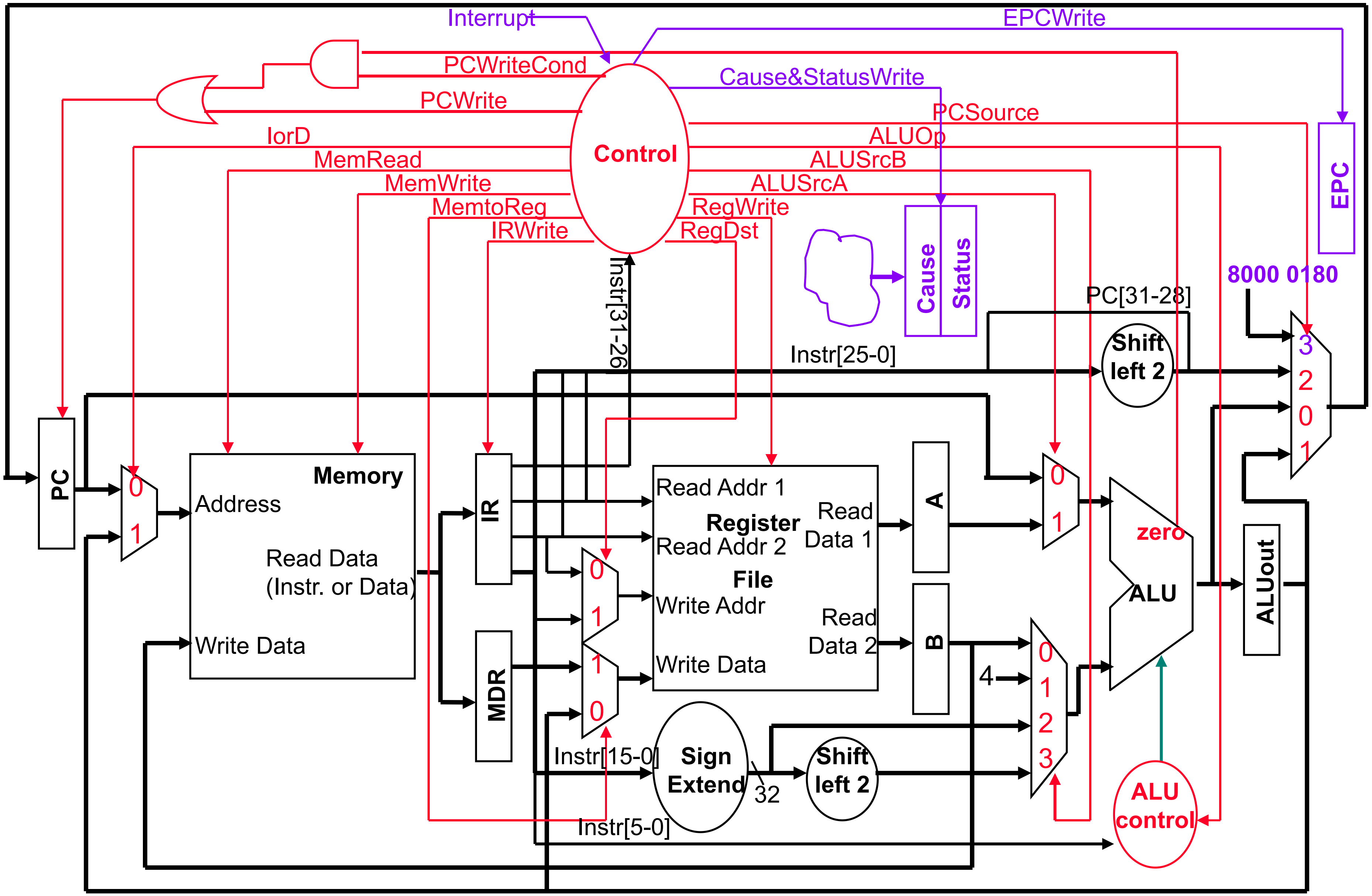
- Interrupts
  - caused by external events (i.e., request from I/O device)
  - asynchronous to program execution
  - may be handled between instructions
  - simply suspend and resume user program

- Traps
  - caused by internal events
    - exceptional conditions (e.g., arithmetic overflow, undefined instr.)
    - errors (e.g., hardware malfunction, memory parity error)
    - faults (e.g., non-resident page – page fault)
  - synchronous to program execution
  - condition must be remedied by the trap handler
  - instruction may be retried (or simulated) and program continued or program may be aborted

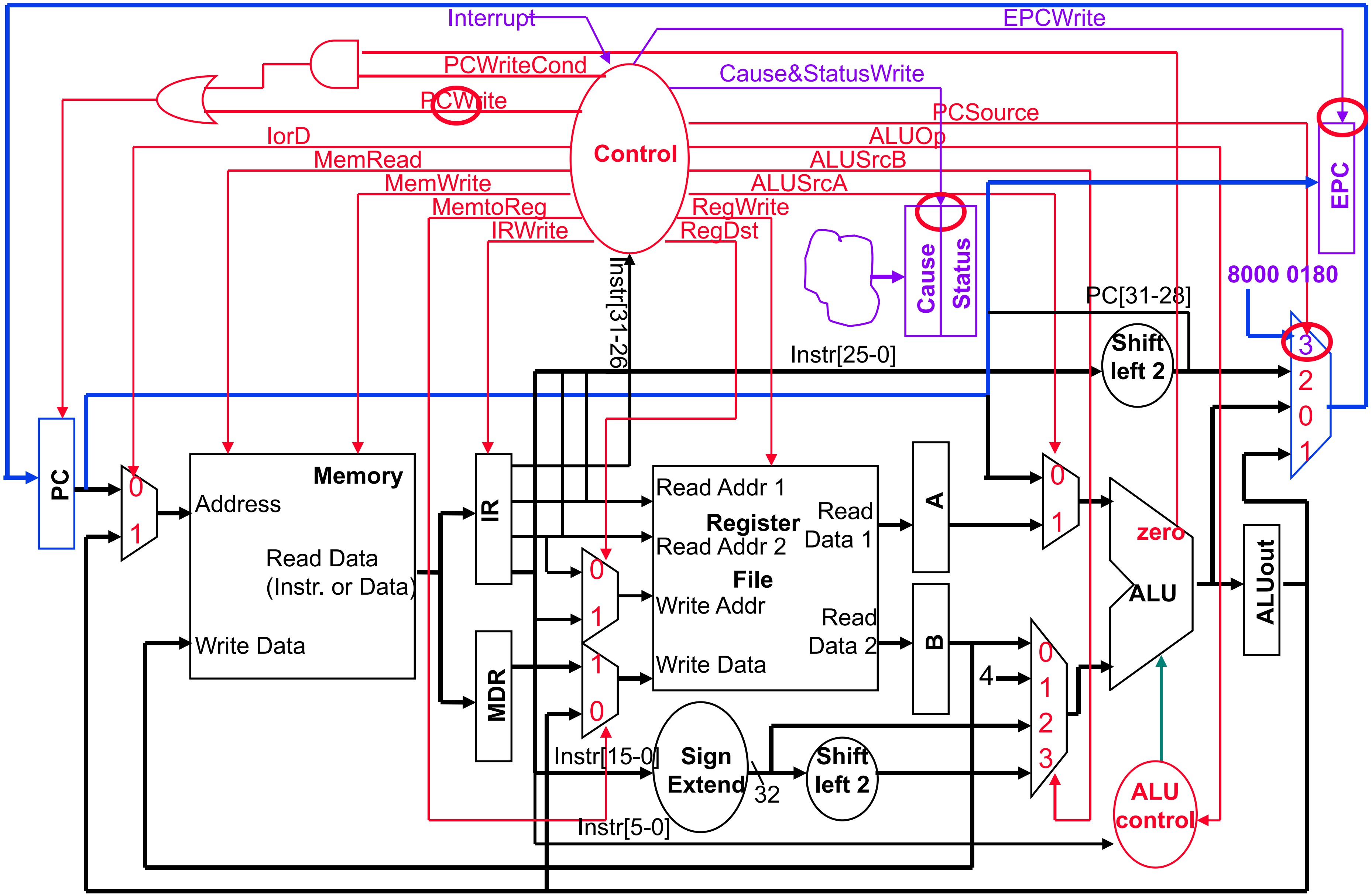
Additions to MIPS ISA for Interrupts

- Control signals to write EPC (EPCWrite), Cause and Status (Cause&StatusWrite)
- Hardware to record the type of interrupt in Cause
- Modify the finite state machine so that
  - the address of interrupt handler ($8000\ 0180_{hex}$) can be loaded into the PC, so must increase the size of PC mux
  - and save the address of the next instr in EPC

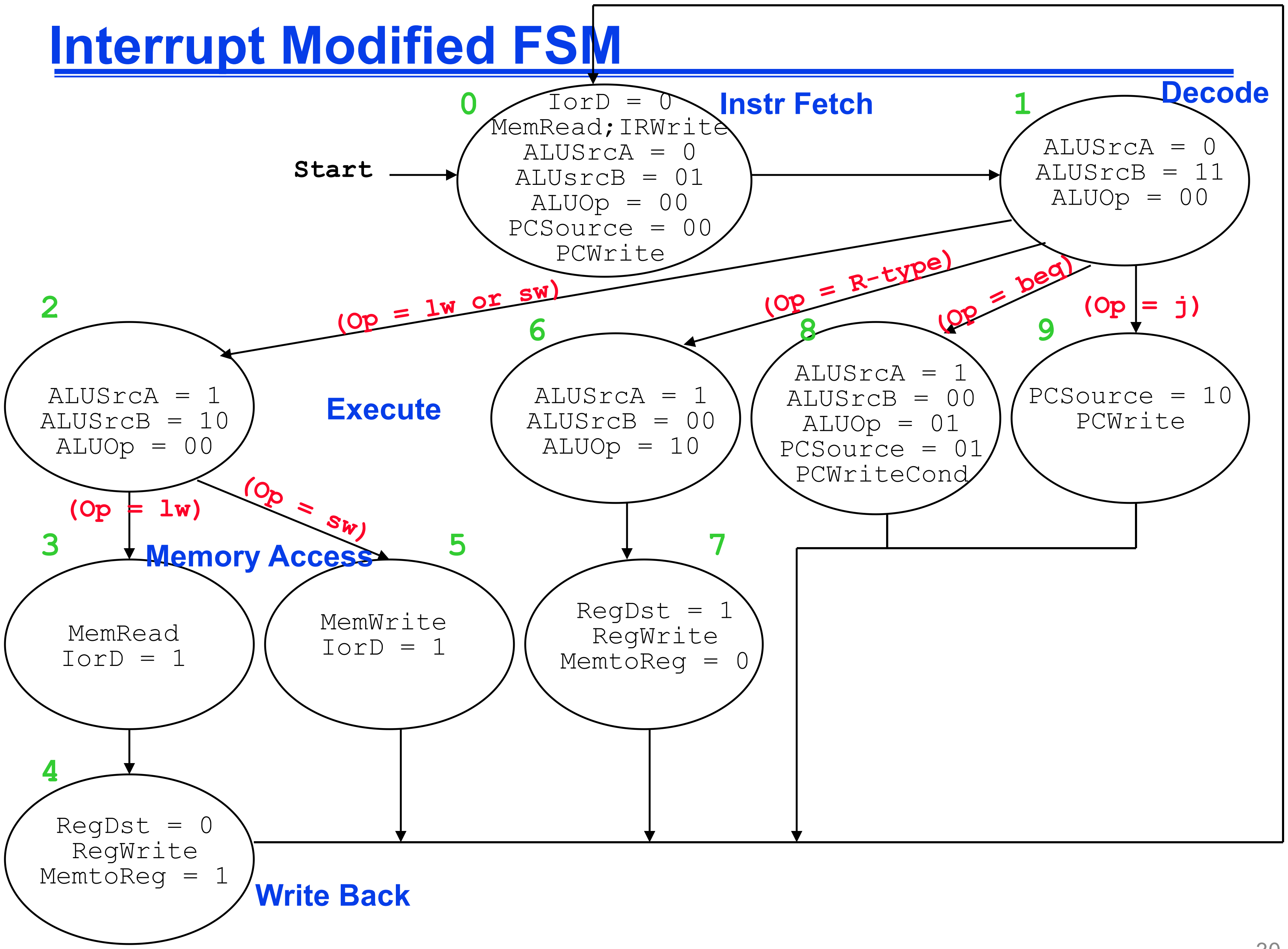
Interrupt Modified Multicycle Datapath



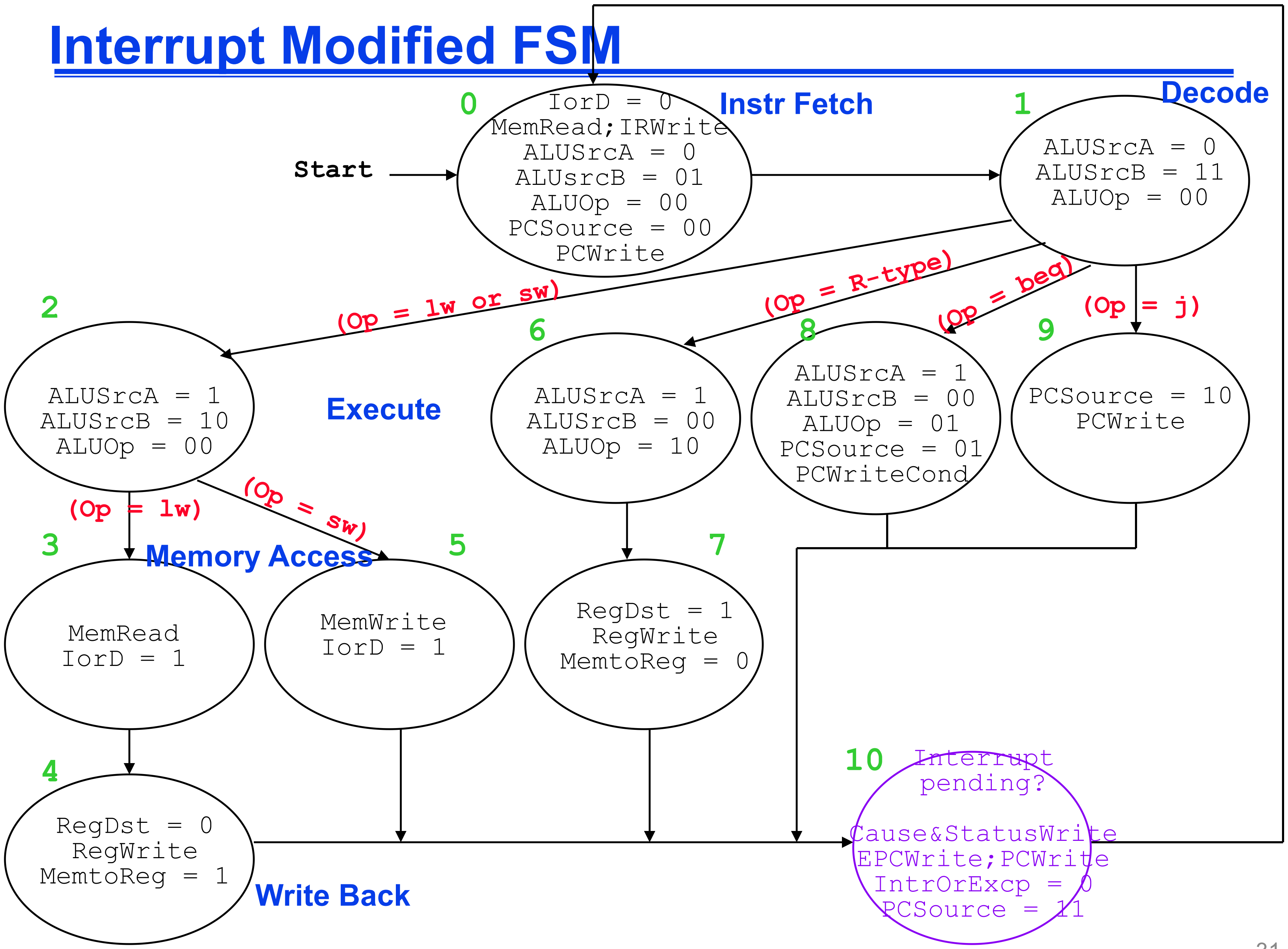
Interrupt Modified Multicycle Datapath



Interrupt Modified FSM



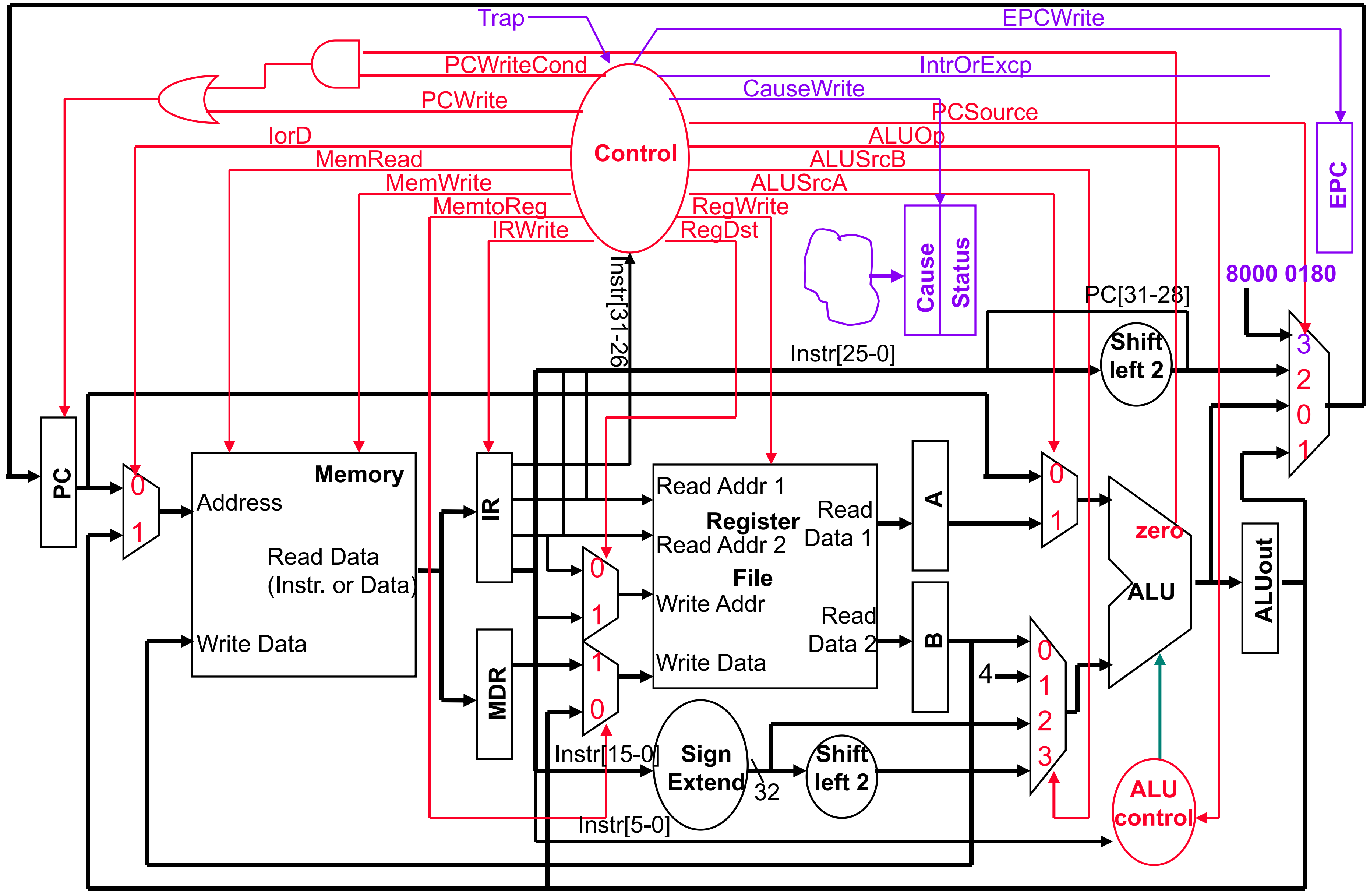
Interrupt Modified FSM



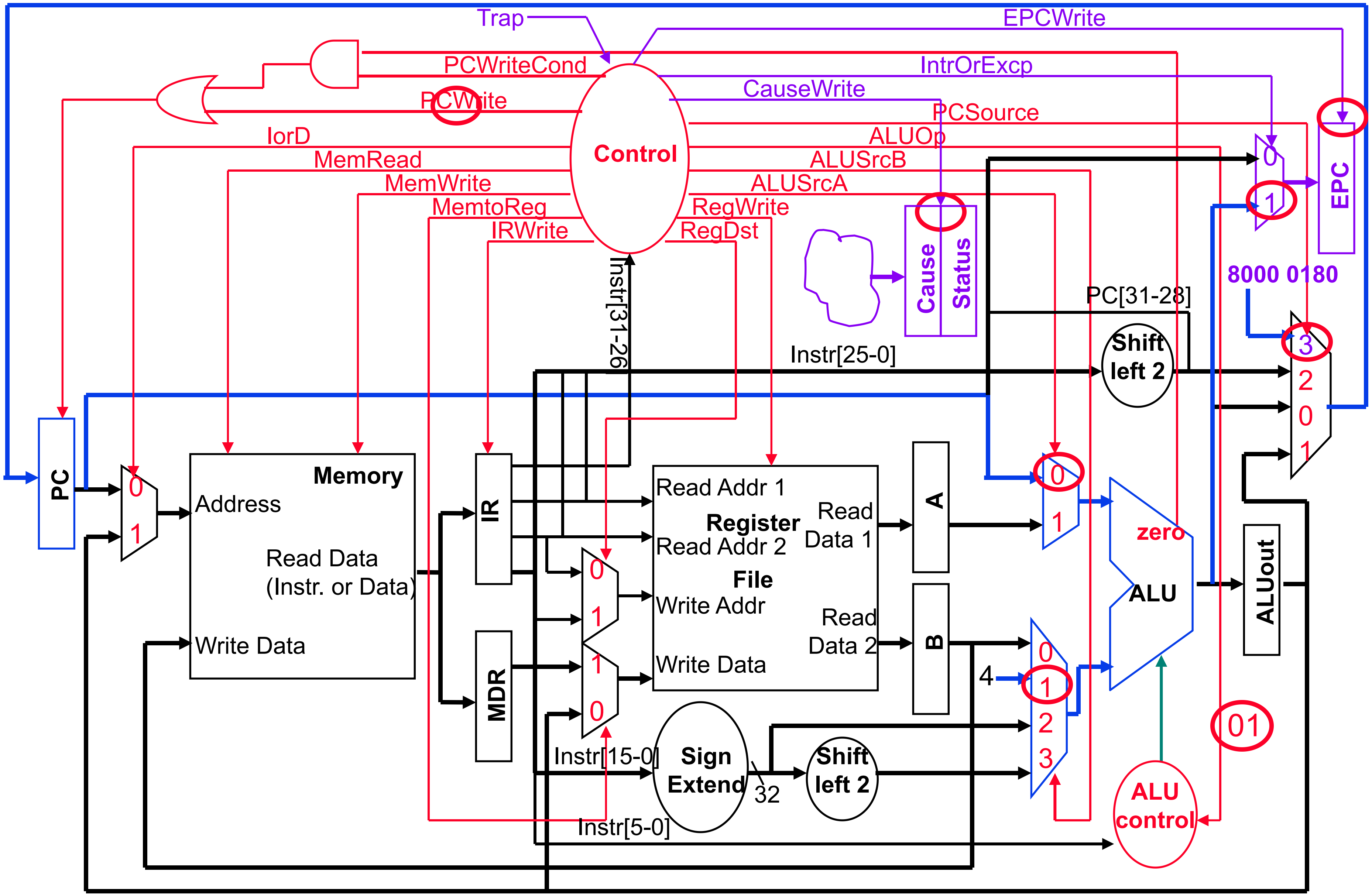
Additions to MIPS ISA for Traps

- Control signals to write EPC (EPCWrite & IntrOrExcp), Cause and Status (Cause&StatusWrite)
- Hardware to record the type of trap in Cause
- Further modify the finite state machine so that
  - for traps, record the address of the current (offending) instruction in the EPC, so must undo the PC = PC + 4 done during fetch

Trap Modified Multicycle Datapath



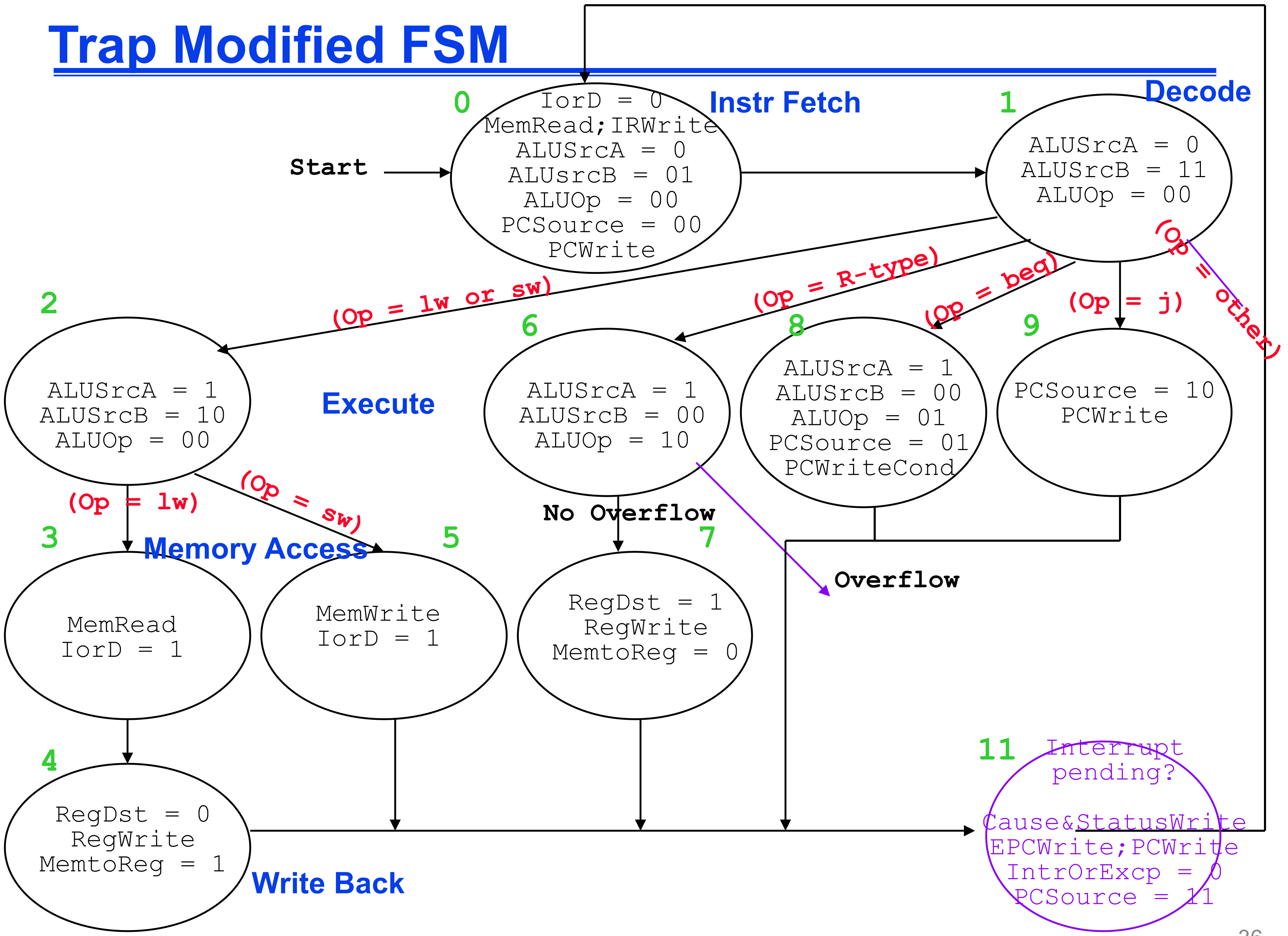
Trap Modified Multicycle Datapath



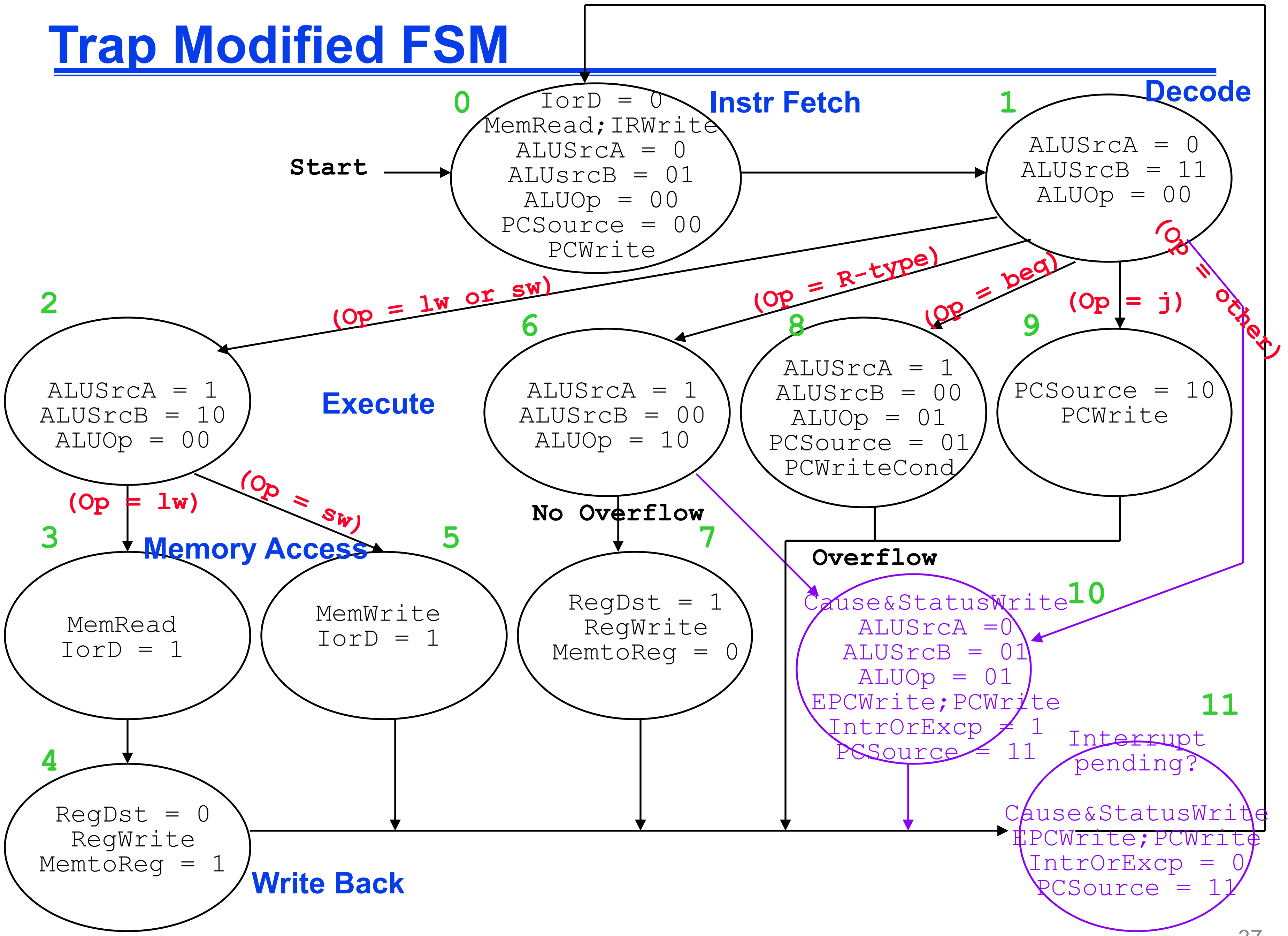
How Control Detects Two Traps

- Undefined instruction (RI) – detected when no next state is defined in state 1 (decode) for the opcode value
  - Define the next state value for all undefined op values as new state 10
- Arithmetic overflow (Ov) – The overflow signal from the ALU is used in state 6 (if don't want to complete RegWrite)
- Need to modify the FSM in a similar fashion for remaining traps
  - Challenge is to handle the interactions between instructions and exception-causing events so that the control logic remains small and fast
    - Complex interactions makes the control unit the most challenging aspect of hardware design, especially in pipelined processors

Trap Modified FSM



Trap Modified FSM



Example Trap Service Routine - OLD

```
    .kdata
s1: .word 0
s2: .word 0
    .ktext   0x80000180
    move     $k1, $at          # Save $at
    sw       $v0, s1           # Not re-entrant and we can't trust $sp
    sw       $a0, s2
    mfc0     $k0, $13          # Save Cause
    sgt      $v0, $k0, 0x00    # ignore interrupt exceptions
    bgtz     $v0, ret

    addu     $0, $0, 0
    li       $v0, 4            # syscall 4 (print_str)
    la       $a0 __m1_
    syscall
    li       $v0, 1            # syscall 1 (print_int)
    srl      $a0, $k0, 2       # shift Cause reg
    syscall
    li       $v0, 4            # syscall 4 (print_str)
    lw       $a0, __excp($k0)
    syscall
    bne      $k0, 0x18, ok_pc # Bad PC requires special checks
    . . .
ok_pc:li     $v0, 4            # syscall 4 (print_str)
    la       $a0, __m2_
    syscall
    mtc0     $0, $13           # Clear Cause register

ret:lw       $v0, s1
    lw       $a0, s2
    move     $at, $k1          # Restore $at
    mfc0     $k0, $14          # read EPC
    addiu    $k0, $k0, 4       # EPC = EPC +4
    mtc0     $k0, $14          # to return to next instruction
    eret                       # Return from exception handler
```