University of Southern California

Viterbi School of Engineering

EE352

Computer Organization and Architecture

Exceptions

References:

1) Textbook
2) Mark Redekopp's slide series

Shahin Nazarian

Spring 2010

What are Exceptions?

- Any event that causes a break in normal execution
  - Error Conditions
    - Invalid address, Arithmetic/FP overflow/error
  - Hardware Interrupts / Events
    - Handling a keyboard press, mouse moving, USB data transfer, etc.
  - System Calls / Traps
    - User applications calling OS code

Error Condition Exceptions

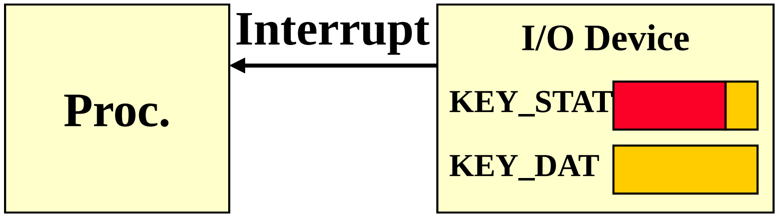
- Bus Error
  - No memory or I/O device responds to a read or write
- Address Error
  - Unaligned address used for half or word accesses
  - Access to Kernel memory space when in User mode
- Floating Point Exceptions
  - Many possible exceptions encompassed by this
- Integer Overflow
  - An instruction that causes 2's complement overflow
- TRAP Instructions
  - SW definable error conditions
- Reserved Instruction
  - User code attempting to perform kernel mode operations
- Coprocessor Unusable
  - Use of a coprocessor instruction when the coprocessor is not implemented

I/O Notification

- Two methods for I/O devices to indicate they need the processor's attention
  - Polling "busy" loop (responsibility on proc.)
    - Processor has responsibility of checking each I/O device
    - Many I/O events happen infrequently (1-10 ms) with respect to the processors ability to execute instructions (1-10 ns)
  - Interrupts (responsibility on I/O device)
    - I/O device notifies processor only when it needs attention

```
Getkey: la      $t0,KEY_STAT ; get status
        lw      $t1,0($t0)
        andi    $t1,$t1,0x0001
        beq     $t1,$zero,getkey
        la      $t2,KEY_DATA ; get key
        lw      $t3,0($t2)
```

Polling Loop



Interrupt

System Calls / TRAP Exceptions

- Provide a controlled method for user mode applications to call kernel mode (OS) code
- Syscalls and traps are very similar to subroutine calls but they switch into kernel mode when called
  - Kernel mode is a special mode of the processor for executing trusted (OS) code

Exception Processing

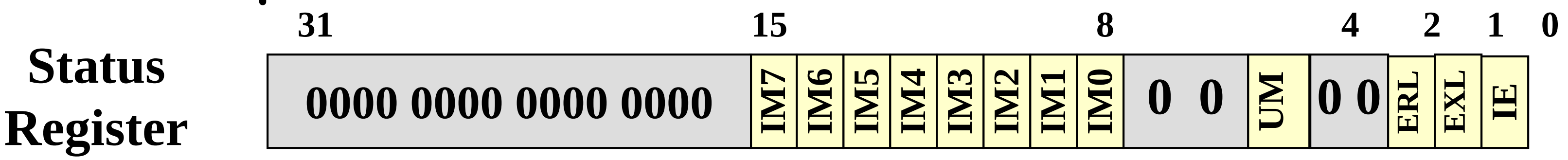
- Save necessary state to be able to restart the process
  - Save PC of offending instruction
- Call an appropriate handler routine to deal with the error / interrupt / syscall
  - Handler identifies the cause of exception and handles it
  - May need to save more state
- Restore state and return to offending application (or kill it if recovery is impossible)

Coprocessor 0 Registers

- Status Register
  - Enables and disables the handling of exceptions/interrupts
  - Controls user/kernel processor modes
    - Kernel mode allows access to certain regions of the address space and execution of certain instructions
- Cause Register: Indicates which exception/interrupt occurred
- Exception PC (EPC) Register
  - Indicates the address of the instruction causing the exception
  - This is also the instruction we should return to after handling the exception
- Coprocessor registers can be accessed via the 'mtc0' and 'mfc0' instructions
  - `mfc0   $gpr,$c0_reg        # R[gpr]   = C0[c0_reg]`
  - `mtc0   $gpr,$c0_reg        # C0[c0_reg] = R[gpr]`

Status Register

- Register 12 in coprocessor 0
- Bit definitions
  - IM[7:0] – Interrupt Mask
    - 1 = Ignore interrupt level/ 0 = Allow interrupt level
  - UM – User Mode
    - 1 = User mode / 0 = Kernel mode
  - ERL/EXL = Exception/Error Level
    - 1 = Already handling exception or error / 0 = Normal exec.
    - If either bit is '1' processor is also said to be in kernel mode
  - IE = Interrupt Enable
    - 1 = Allow unmasked interrupts / 0 = Ignore all interrupts despite the IM bits



Status Register

31 15 8 4 2 1 0

0000 0000 0000 0000 | IM7 | IM6 | IM5 | IM4 | IM3 | IM2 | IM1 | IM0 | 0 0 | UM | 0 0 | ERL | EXL | IE

Cause Register

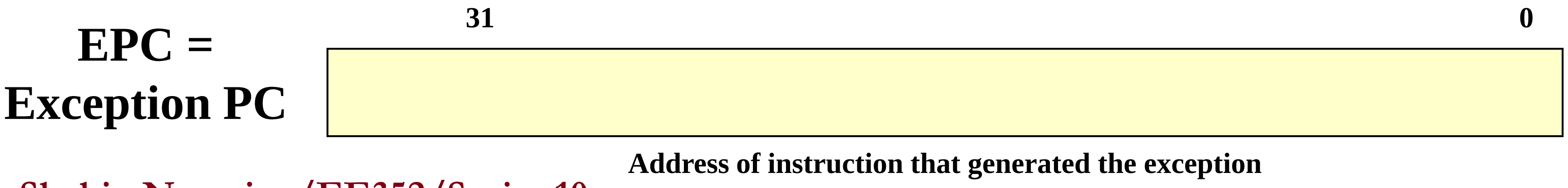
- Register 13 in coprocessor 0
- Bit definitions
  - BD – Branch Delay
    - The offending instruction was in the branch delay slot
    - EPC points at the branch but it was EPC+4 that caused the exception
  - PI[7:0] – Pending Interrupt
    - 1 = Interrupt Requested / 0 = No interrupt requested
  - Exception Code – Indicates cause of exception (see table)

Code	Cause
0	Interrupt (HW)
4, 5	Load (4), Store (5) Address Error
6, 7	Instruc. (6), Data (7) Bus Error
8	Syscall
9	Breakpoint
10	Reserved Instruc.
11	CoProc. Unusable
12	Arith. Overflow
13	Trap
15	Floating Point

	31		15							8	7	6 2	1 0
Cause Register	BD	000 0000 0000 0000	PI7	PI6	PI5	PI4	PI3	PI2	PI1	PI0	0	Exception Code	0 0

EPC Register

- Exception PC holds the address of the offending instruction (unless BD bit is set)
  - Can be used along with 'Cause' register to find and correct some error conditions
- 'eret' instruction used to return from exception handler, back to execution point in original code (unless handling the error means having the OS kill the process)
  - 'eret' Operation: PC = EPC

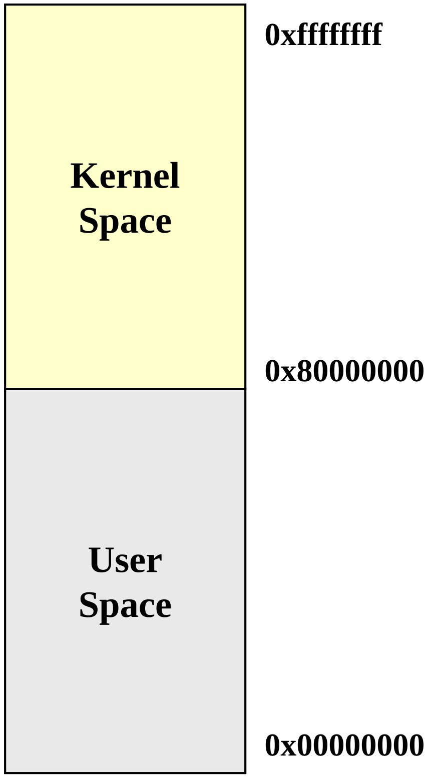


Kernel and User Mode

- User applications are designed to run in user mode
- OS and other system software should run in kernel mode
- Certain features/privileges are only allowed to code running in kernel mode
- Processor is in kernel mode if {UM = 0 OR ERL = 1 OR EXL = 1}

Kernel Mode Privileges

- Privileged instructions
  - 'eret', 'di', 'ei', etc.
  - User apps. shouldn't be allowed to disable/enable interrupts, change memory mappings, etc.
- Privileged Memory or I/O access
  - Processor supports special areas of memory or I/O space that can only be accessed from kernel mode
- Separate stacks and register sets
  - MIPS processors can use "shadow" register sets (alternate GPRs when in kernel mode)



Address Space

Syscalls

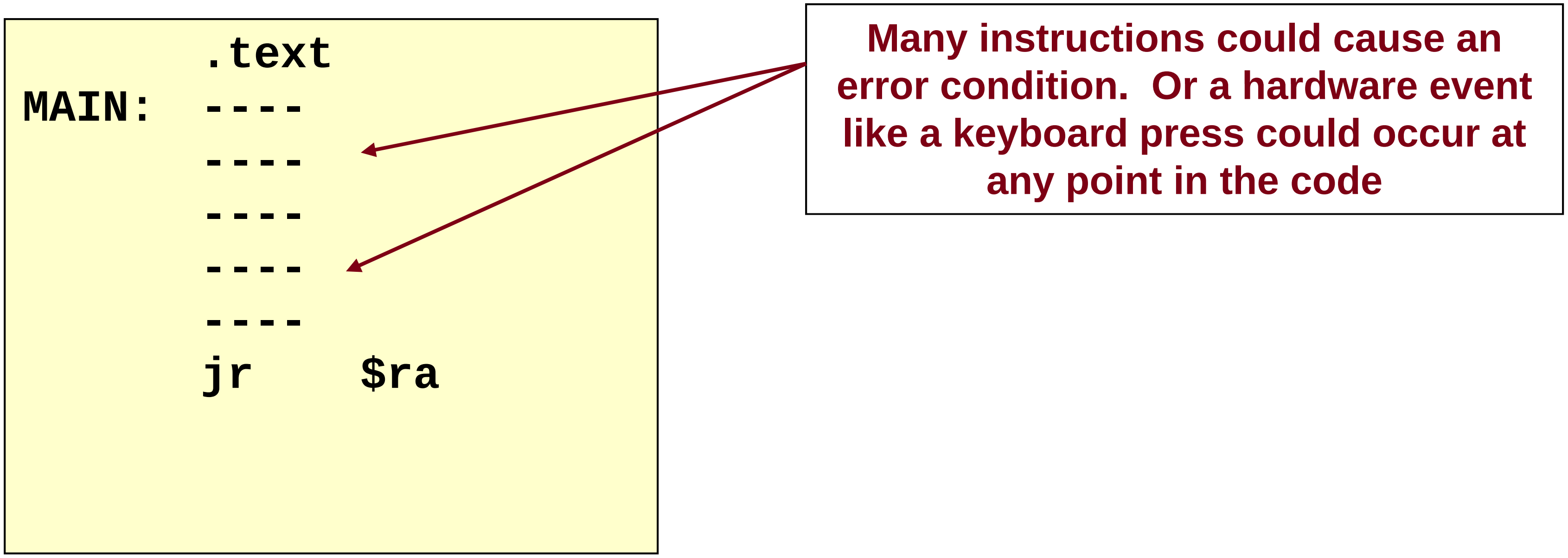
- Provides a structured entry point to the OS
  - Really just a subroutine call that also switches into kernel mode
  - Often used to allow user apps. to request I/O or other services from the OS
- Syntax: syscall
  - Necessary arguments are defined by the OS and expected to be placed in certain registers

Calling Exception Handlers

- Processor will automatically call a routine known as an exception handler to handle the exception
- Problems with calling exception handlers
  - How does the HW 'automatically' call handler routine since we don't know where we will be in our code when an exception occurs
  - Calling the exception handler can cause changes in state (registers) when we return to the running application

Problem of Calling a Handler

- We can't use explicit 'jal' instructions to call exception handlers since we don't when they will occur



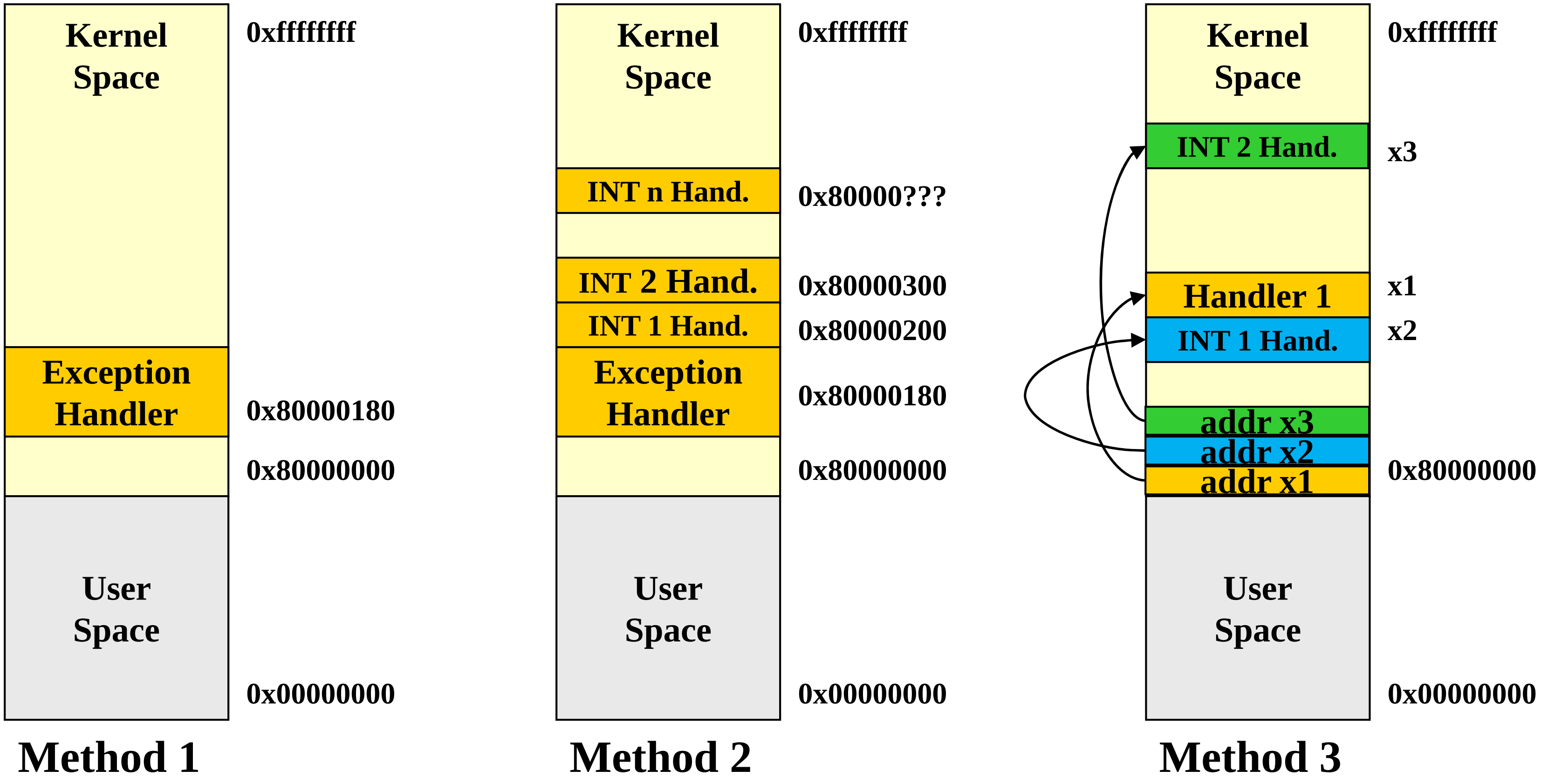
```
         .text
MAIN:    ----
         ----
         ----
         ----
         ----
         jr    $ra
```

Many instructions could cause an error condition. Or a hardware event like a keyboard press could occur at any point in the code

Solution for Calling a Handler

- Since we don't know when an exception will occur there must be a preset location where an exception handler should be defined or some way of telling the processor in advance where our exception handlers will be located
- Method 1: Single hardwired address for master handler
  - Early MIPS architecture defines that the exception handler should be located at 0x8000_0180. Code there should then examine CAUSE register and then call appropriate handler routine (Mars uses this method)
- Method 2: Vectored locations (usually for interrupts)
  - Each interrupt handler at a different address based on interrupt number (a.k.a. vector) (INT1 @ 0x80000200, INT2 @ 0x80000300)
- Method 3: Vector tables
  - Table in memory holding start address of exception handlers (i.e. overflow exception handler pointer at 0x0004, FP exception handler pointer at 0x0008, etc.)

Handler Calling Methods

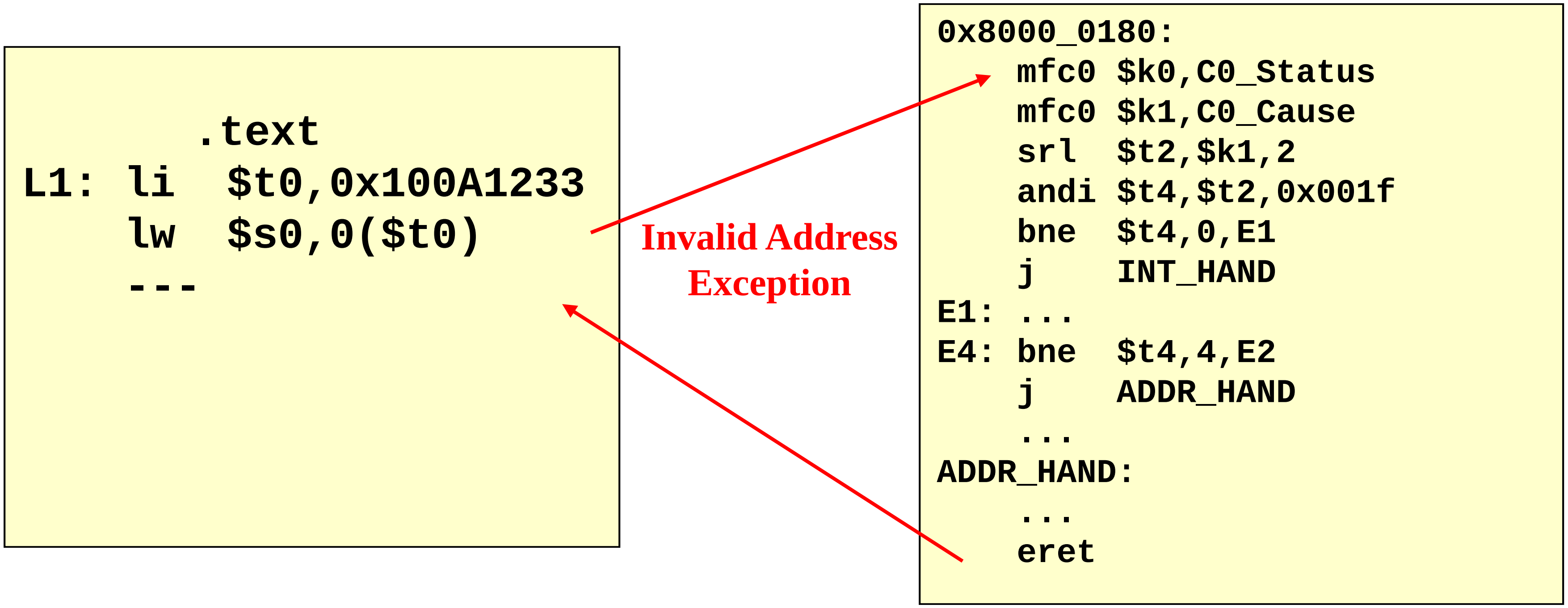


Exception Processing Steps

- Save PC (address of offending instruction) [HW]
  - So we know where to return once we're done handling the exception
- Record the cause of the exception [HW]
  - HW will identify source of HW errors or interrupts
- Switch to kernel mode [HW]
- Calculate and jump to exception handler [HW]
  - Using appropriate method for locating handler
- Execute Handler [SW]
  - Likely need to save/restore state (i.e. GPRs)
- Use 'eret' to return to address in EPC [SW]

Sample Exception Handler

- Main handler needs to examine cause register and call a more specific handler
- Handlers must end with 'eret' rather than 'jr $ra'



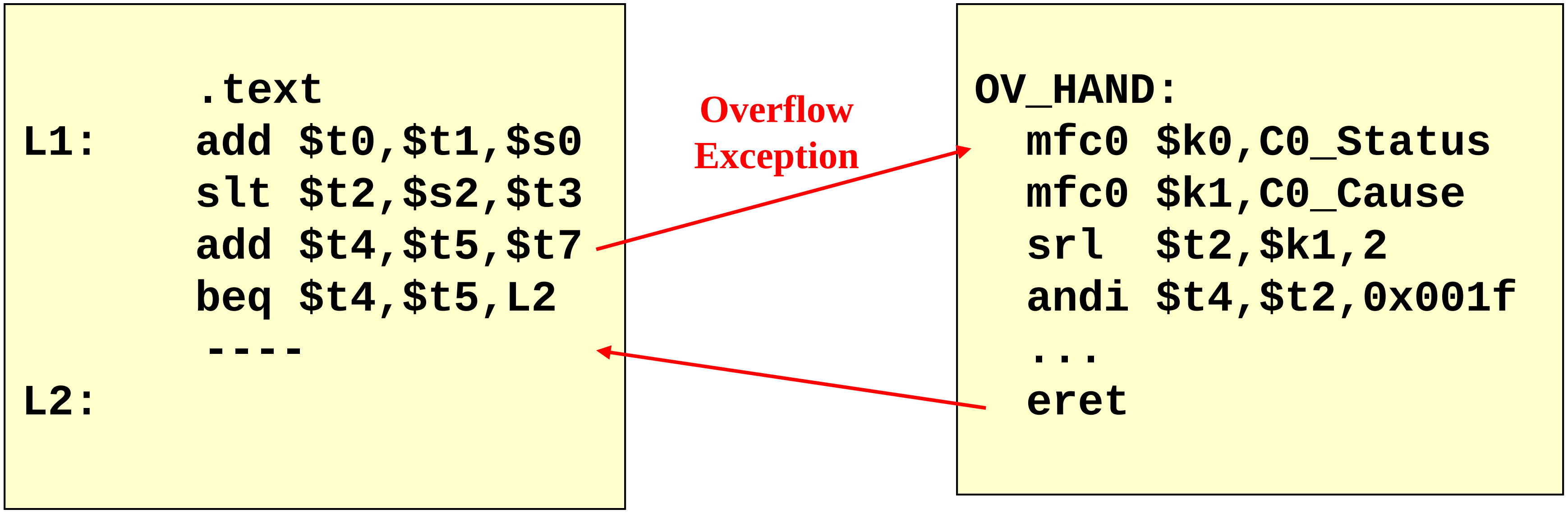
```
        .text
L1: li  $t0,0x100A1233
    lw  $s0,0($t0)
    ---
```

```
0x8000_0180:
    mfc0 $k0,C0_Status
    mfc0 $k1,C0_Cause
    srl  $t2,$k1,2
    andi $t4,$t2,0x001f
    bne  $t4,0,E1
    j    INT_HAND
E1: ...
E4: bne  $t4,4,E2
    j    ADDR_HAND
    ...
ADDR_HAND:
    ...
    eret
```

Main handler can determine cause and

Problem of Changed State

- Since exceptions can occur at any time, the handler **must save all registers it uses** (except for kernel registers $k0-$k1 which should be unused by the user app)



```
        .text
L1:     add $t0,$t1,$s0
        slt $t2,$s2,$t3
        add $t4,$t5,$t7
        beq $t4,$t5,L2
         ----
L2:
```

Overflow Exception

```
OV_HAND:
  mfc0 $k0,C0_Status
  mfc0 $k1,C0_Cause
  srl  $t2,$k1,2
  andi $t4,$t2,0x001f
  ...
  eret
```

Handlers need to save/restore values to stack to avoid overwriting needed register values (e.g. $t2, $t4)

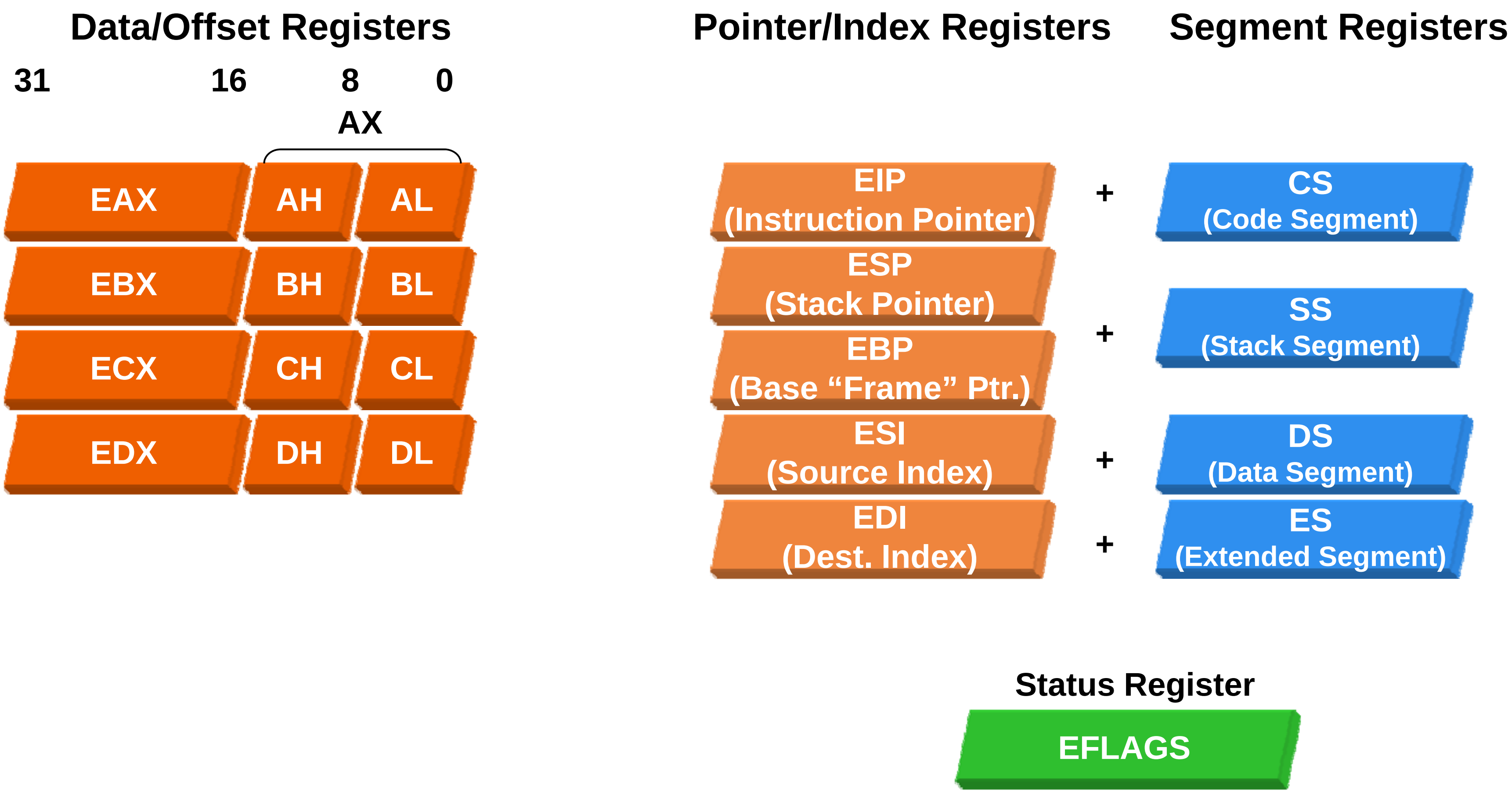
Bonus Material

Intel Architectures

Intel Architectures

Processor	Year	Address Size	Data Size
8086	1978	20	16
80286	1982	24	16
80386/486	'85/'89	32	32
Pentium	1993	32	32
Pentium 4	2000	32	32
Core 2 Duo	2006	64	64

Intel (IA-32) Architectures



Data/Offset Registers

31 16 8 0

AX

EAX AH AL

EBX BH BL

ECX CH CL

EDX DH DL

Pointer/Index Registers

Segment Registers

EIP (Instruction Pointer) + CS (Code Segment)

ESP (Stack Pointer)

EBP (Base “Frame” Ptr.)

+ SS (Stack Segment)

ESI (Source Index) + DS (Data Segment)

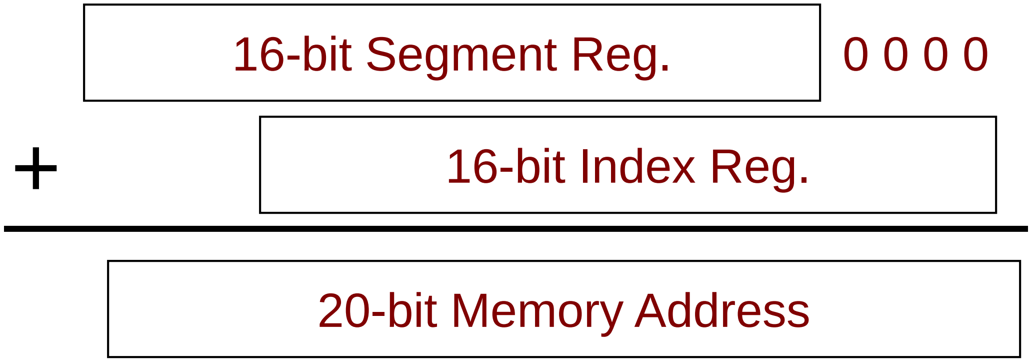
EDI (Dest. Index) + ES (Extended Segment)

Status Register

EFLAGS

Real Mode Addressing

- How to make 20-bit address w/ 16-bit registers?
  - Use 2 16-bit registers
  - (Segment register * 16) + Index Reg.
- Format:
  - Seg Reg:Index Reg (e.g. CS:IP)



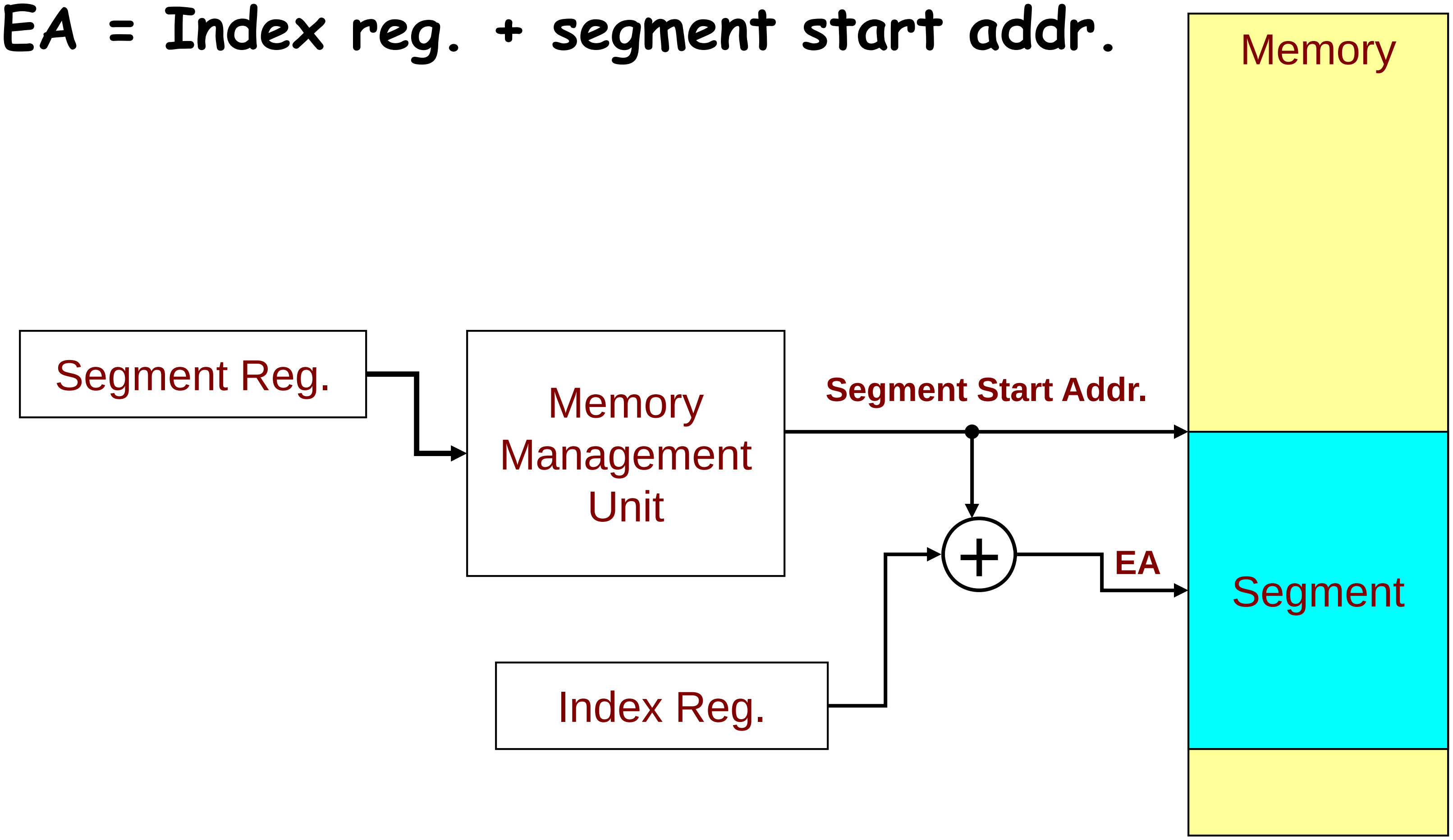
Examples:

$1A5C:$405E → $1A5C0 + $405E = $1E61E

$74AB:$E892 → $74AB0 + $E892 = $83345

Protected Mode Addressing

- Segment Register value selects a segment (area of memory)
- EA = Index reg. + segment start addr.



IA-32 Addressing Modes

Name	Example	Effective Address
Immediate	MOV EAX,5	Operand = Value
Direct	MOV EAX,[100]	EA = Addr
Register	MOV EAX,EDX	EA = Reg
Register indirect	MOV EAX,[EBX]	EA = (Reg)
Base w/ Disp.	MOV EAX,[EBP+60]	EA = (Reg) + Disp
Index w/ Disp.	MOV EAX,[ESI*4+10]	EA = (Reg)*S + Disp.
Base w/ Index	MOV EAX,[EBP + ESI*4]	EA = (Reg1)+(Reg2)*S
Base w/ Index & Disp.	MOV EAX,[EBP+ESI*4+100]	EA = (Reg1)+(Reg2)*S + Disp.

IA-32 Instructions

- Stack aware instructions
  - ‘call’ / ‘ret’ automatically push/pop return address onto stack
  - ‘push’ / ‘pop’ instructions for performing these operations
- Memory / Register architecture
  - One operand can be located in memory
    - add eax, [ebp] # adds reg. EAX to value pointed at by EBP
- Specialized Instructions
  - xchg src1, src2 # exchanges/swaps operands
  - string copy and compare instructions