

Intel x86 Assembly and Debugging Support

Scott M. Lewandowski

CS295-2: Advanced Topics in Debugging

September 21, 1998

Assembler Syntax

- Everything looks like this:

```
label:    instruction  dest,src  
         instruction  label
```

- Comments:

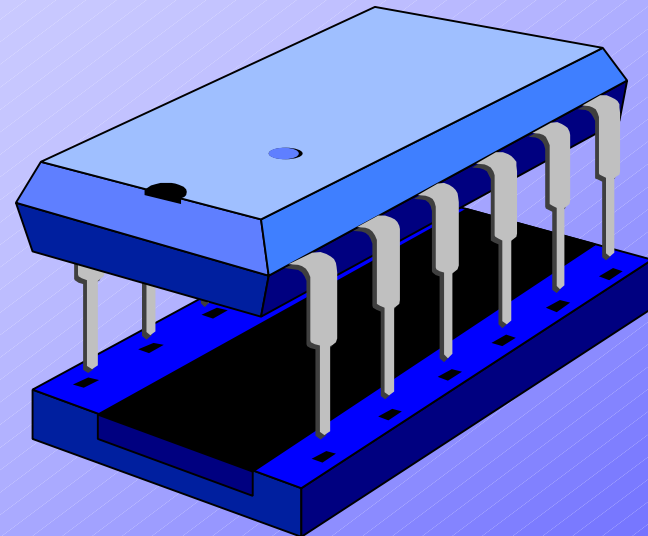
```
comment $  
    This is a comment  
commend $  
; Single line comment
```

- Immediate values:

```
123, 0123, 0x123, 123h, 'm', '\n'
```

Memory Organization

- 4 Gigabytes of addressable memory
- Support for segmentation and paging
- “Flat” memory model is a special case of segmentation



Data Types

- Bytes, words, doublewords, quadwords
- Little-endian
- Basically no alignment restrictions
- Recognized types
 - Integer
 - Ordinal
 - BCD integer, packed BCD integer
 - Near pointer, far pointer
 - Bit field, bit string, byte string
 - Floating point types

Registers

- Sixteen registers for application programmers
 - 8 general (32-bit)
 - ◆ EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI
 - 6 segment (16 bit): determine accessible code
 - 2 status and control: state of processor
 - ◆ All condition codes kept in EFLAGS (32-bit)
 - ◆ DF (of EFLAGS): controls string instructions
- EIP
 - Offset in code segment
 - Not directly accessible to programmer

Non-32-Bit Registers

■ Aliases provided for registers

● EBP, ESI, EDI, and ESP

◆ Can access bits 0-15 as BP, SI, DI, and SP

● EAX, EBX, ECX, and EDX

◆ Can access bits 0-15 as AX, BX, CX, and DX

◆ Can access bits 0-7 as AL, BL, CL, and DL

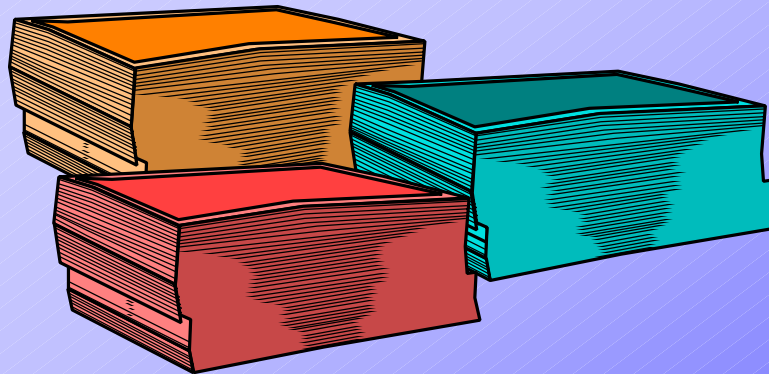
◆ Can access bits 8-15 as AH, BH, CH, DH

■ These are carryovers from 80286, but are useful

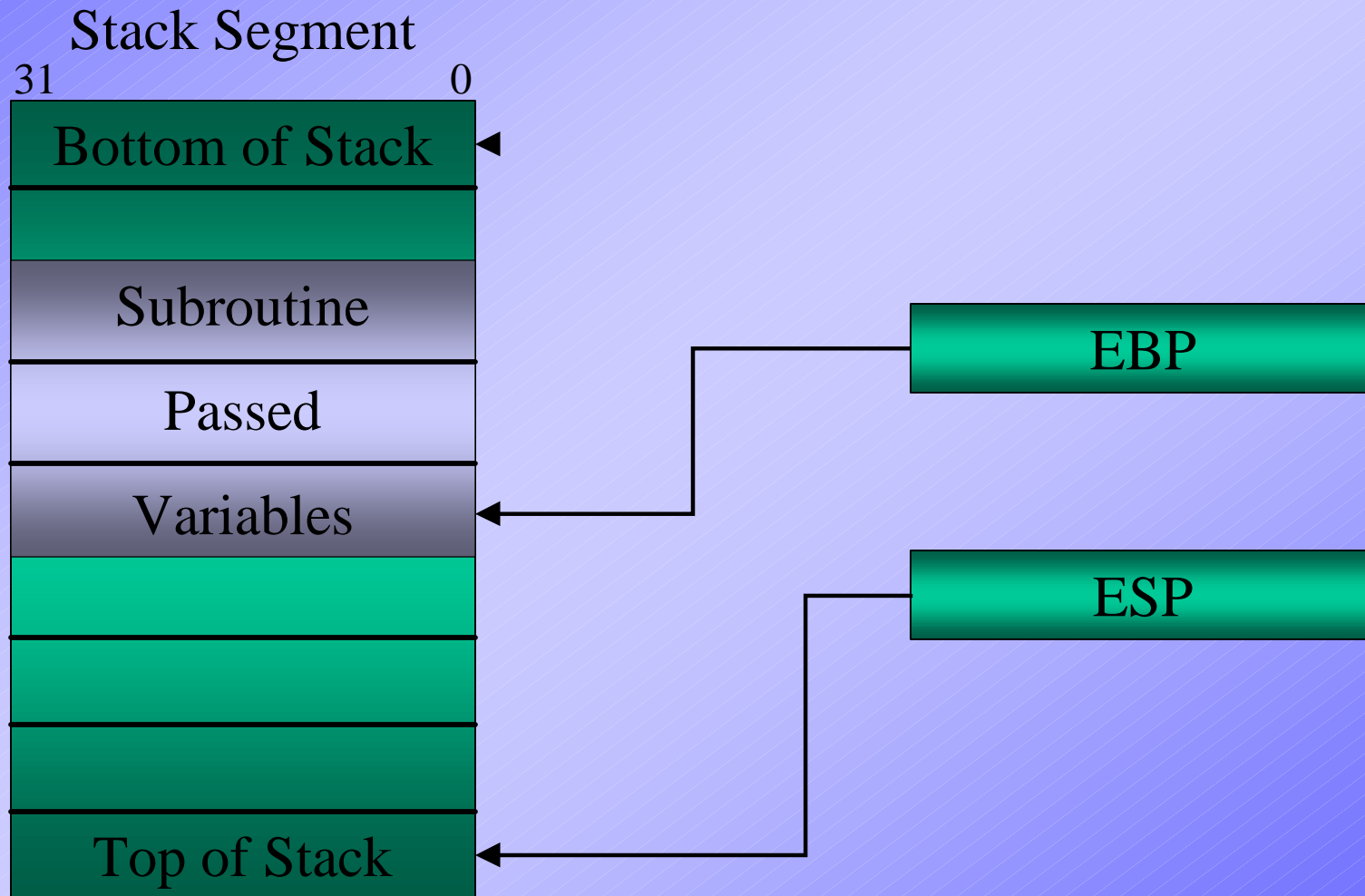


Stack Implementation

- Supported by three registers
 - Stack segment (SS): current stack
 - Stack pointer (ESP): offset to top of stack
 - Stack frame-base pointer (EBP)
 - ◆ Access data structures passed on the stack
 - ◆ Standard to copy ESP to EBP on procedure entry



Stack Layout



Instruction Format

- **Prefixes**
 - Segment override
 - Address and operand size
 - Repeat
 - Lock
- **Opcode and implicit operand(s)**
- **Register specifier**
- **Addressing mode specifier**
- **Scale, index, base (SIB) type**
- **Displacement**
- **Immediate operand (data)**

Instruction Examples

- `XCHG EAX, EBX`

- `ADD [EBP+8][ESI*4], 17`

- `MOV EAX, SS:[42H]`

Data Addressing Modes

- `MOV EAX, 123h` Immediate
- `MOV EAX, NUMBER1` Direct
 - `NUMBER1` is a variable (address)
- `MOV EAX,EBX` Register
- `MOV EAX,[EBX]` Register Indirect
- `MOV EAX,[EBX+8]` Register Relative
 - Operand is at `EBX+8` (can also write `8[EBX]`)
- `MOV [EAX+ESI],EBX` Base Indexed
 - Base and index registers determine the address
- `MOV EAX,[EBX+EDI+2]` Base Index Relative
 - Base and index register plus immediate offset are added

Segmentation: “What” & “Why”

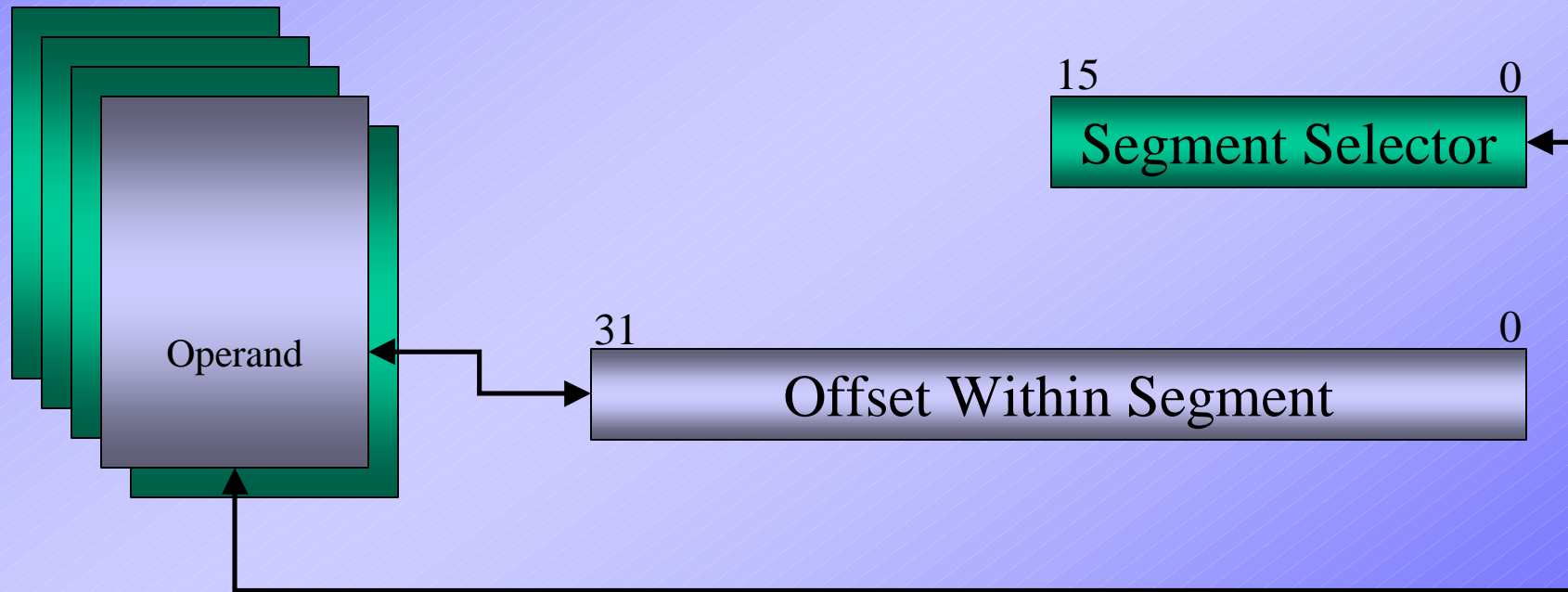
■ What is a segment?

- Provides an independent memory spaces
- Holds code, data, or stack
- Has address space up to 4GB
- Visible to programmer

■ Why segmentation?

- Provide separate memory spaces
- Segments can be individually controlled
- To access large amounts of memory
 - ◆ Up to 16,383 segments of size up to 4GB = 64TB

Segmented Addressing



Segment Registers

- Determine the segments a program can access
- Total of six
 - ◆ CS: points to code segment
 - CALL and JMP instructions
 - ◆ DS: points to main data area
 - Data instructions (MOV, etc.)
 - ◆ SS: stack segment (often same as data segment)
 - PUSH and POP instructions
 - ◆ ES, FS, GS: special information
- Can specify which segment an instruction uses
 - Example: MOV EAX, CS:[0]

Segmentation: Historical Note

- We are discussing 80386 segmentation
- 80286 and earlier limited segment size to 64KB
 - Compatibility with the 8080 (64KB address space)
 - Allowed 16 bit addressing to continue while providing access to more memory

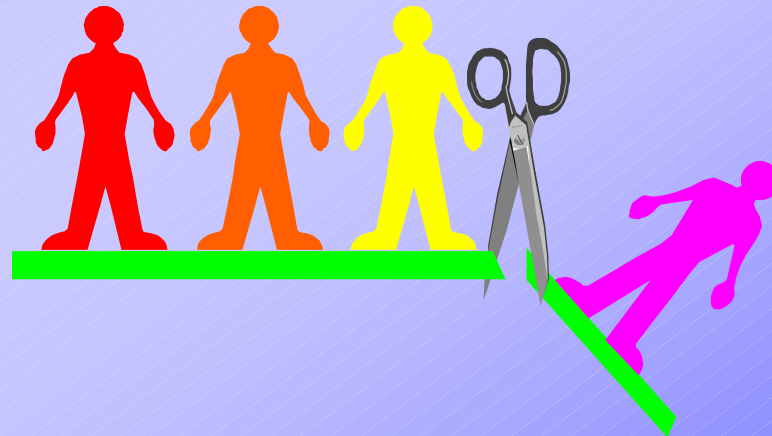


How Segmentation Works

- Segment descriptor: address and size
- Physical address = offset + base address
- Pointer into a segment
 - Segment selector: 16 bits to identify a segment
 - Offset: 32 bit address within a segment
- CR0 register toggles between direct mapping to physical memory and using paging

Do We Still Need Segmenting?

- Not required for memory access purposes
- Can be used to implement memory management
- Most operating systems use a flat-memory model
 - Really just one large segment



Data Movement Instructions

- **MOV dest,src**
 - No segment register to segment register moves
 - Use MOVS for memory to memory moves
- **XCHG reg,reg**
- **Stack manipulation**
 - PUSH val, PUSHA
 - POP val, POPA
 - PUSHA stores ESP, but POPA does not restore it

Arithmetic Instructions

- ADD/ADC dest,src
- SUB/SUBB dest,src
- INC/DEC reg
- CMP dest,src
 - Register unchanged; flags updated
- NEG val

$$\begin{array}{r} 3 \\ +4 \\ \hline 7 \end{array}$$

Arithmetic Instructions (continued)

- MUL/DIV val (AL/AX/EAX implicit)
- IMUL: 1, 2, 3 and 3 operand forms
- IDIV

Operands for Division

Operand Size (Divisor)	Dividend	Quotient	Remainder
Byte	AX	AL	AH
Word	DX and AX	AX	DX
Doubleword	EDX and EAX	EAX	EDX

Logical Instructions

■ Boolean operations

- AND, OR, XOR, NOT
- Update flags and saves value in destination register

■ Bit test and modify

Instruction	Effect on CF Flag	Effect on Selected Bit
BT (Bit Test)	CF Flag \leftarrow Selected Bit	No Effect
BTS (Bit Test and Set)	CF Flag \leftarrow Selected Bit	Selected Bit \leftarrow 1
BTR (Bit Test and Reset)	CF Flag \leftarrow Selected Bit	Selected Bit \leftarrow 0
BTC (Bit Test and Complement)	CF Flag \leftarrow Selected Bit	Selected Bit \leftarrow -(Selected Bit)

Logical Instructions (continued)

- Bit scan instructions
- Shift and rotate instructions
- Byte-set-on-condition instructions
 - SETcc: sets byte to 0 or 1 based on condition code
- TEST instruction
 - Logical AND without alteration to destination operand

Function Calling Conventions

- `__cdecl`
 - Parameters passed on stack, right to left
 - Caller maintains stack
 - Used for variable arguments
- `__stdcall`
 - Parameters passed on stack, right to left
 - Callee maintains stack
- `__fastcall`
 - First two 32-bit values passed in ECX & EDX
 - Remaining params passed on stack, right to left

A Sample Function (`__cdecl`)

```
_x$ = 8
```

```
_y$ = 12
```

```
_z$ = 16
```

```
?sample@@YAHHHH@Z PROC NEAR                                ; sample
; 5      : int sample(int x,int y,int z) {
00000      55                push   ebp
00001      8b ec             mov    ebp, esp
; 6      :      return x+y+z;
00003      8b 45 08             mov    eax, DWORD PTR _x$[ebp]
00006      03 45 0c             add    eax, DWORD PTR _y$[ebp]
00009      03 45 10             add    eax, DWORD PTR _z$[ebp]
; 7      : }
0000c      5d                pop    ebp
0000d      c3                ret    0
```

A Sample Function (`__stdcall`)

```
_x$ = 8
_y$ = 12
_z$ = 16
?sample@@YGHHHH@Z PROC NEAR                                ; sample
; 5      : int sample(int x,int y,int z) {
00000      55                push   ebp
00001      8b ec             mov    ebp, esp
; 6      :      return x+y+z;
00003      8b 45 08             mov    eax, DWORD PTR _x$[ebp]
00006      03 45 0c             add    eax, DWORD PTR _y$[ebp]
00009      03 45 10             add    eax, DWORD PTR _z$[ebp]
; 7      : }
0000c      5d                pop    ebp
0000d      c2 0c 00             ret    12                ; 0000000cH
```

A Sample Function (`__fastcall`)

```
_x$ = -4
_y$ = -8
_z$ = 8
?sample@@YIHhhh@Z PROC NEAR                                ; sample
; 5      : int sample(int x,int y,int z) {
00000      55                push   ebp
00001      8b ec            mov    ebp, esp
00003      83 ec 08          sub    esp, 8
00006      89 55 f8          mov    DWORD PTR _y$[ebp], edx
00009      89 4d fc          mov    DWORD PTR _x$[ebp], ecx
; 6      : return x+y+z;
0000c      8b 45 fc          mov    eax, DWORD PTR _x$[ebp]
0000f      03 45 f8          add    eax, DWORD PTR _y$[ebp]
00012      03 45 08          add    eax, DWORD PTR _z$[ebp]
; 7      : }
00015      8b e5            mov    esp, ebp
00017      5d                pop    ebp
00018      c2 04 00          ret    4
```

Control Transfer Instructions

■ Unconditional transfers

- CALL

- ◆ Pushes EIP onto stack

- RET arg

- ◆ Retrieves return address from stack

- ◆ Optional argument specifies adjustment to ESP

- JMP

■ Conditional jumps

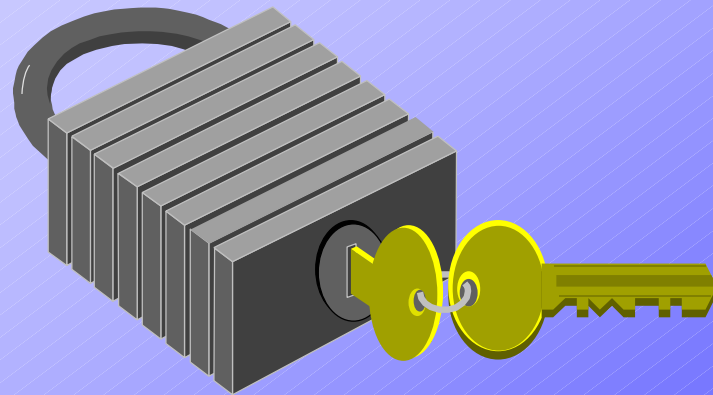
■ Looping

Miscellaneous Instructions

- **LEA reg,memory: Load Effective Address**
 - Puts offset in register
 - Useful for initializing ESI, EDI
- **NOP**
 - 1 byte
 - Advances EIP
- **BSWAP: Byte Swap (Endian conversion)**
 - Bits 7...0 exchanged with 31...24
 - Bits 15...8 exchanged with 23...16
 - Speeds arithmetic

Protected Mode System Instructions

- Not all useful to applications
- Not all protected from application
- Some functions provided
 - Verification of pointer parameters
 - Interrupt control
 - Debugging
 - System control



Types of Exceptions

■ Fault

- CS and EIP point to instruction that generated fault
- Instruction restart (processor registers restored)

■ Trap

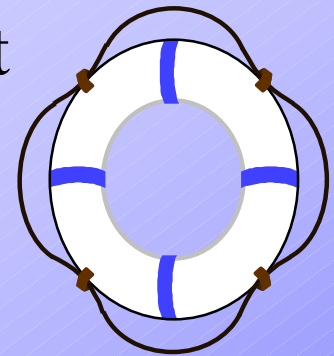
- CS and EIP: instruction after one that generated trap
- Transfers are reflected

■ Abort

- No precise location of instruction causing exception
- Cannot restart program
- Hardware errors, illegal values in system tables

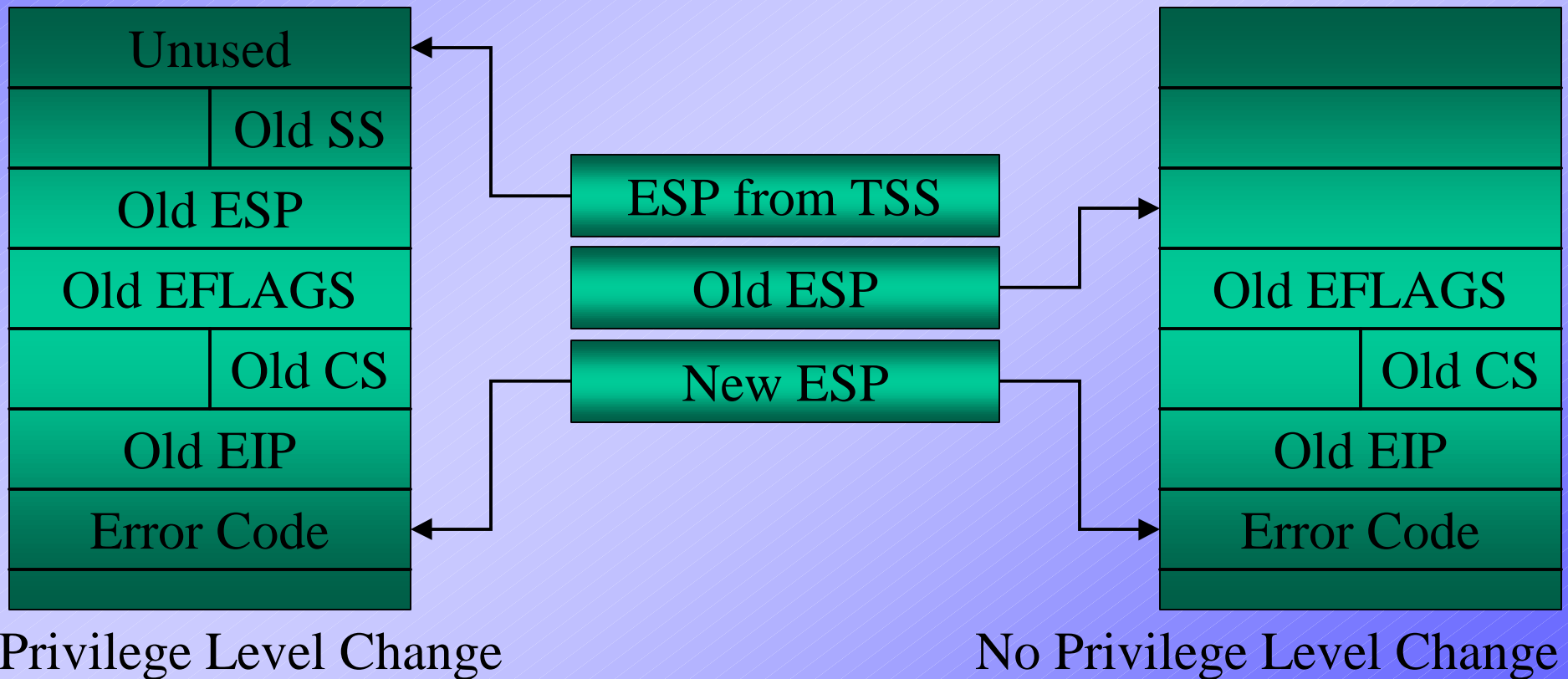
Interrupt Tasks and Procedures

- Use stack to preserve processor state
 - Interrupt pushes EFLAGS onto stack first
 - Error codes can be pushed
- Returning from interrupt procedure
 - Use IRET instead of RET
 - ◆ Increments ESP extra 4 bytes and restores EFLAGS



Stack Frame After an Exception

- Example shown with error code



Exceptions

■ Return address adjustment

- Usually return address is offending instruction
- Debug exception: check DR6

Exception Conditions

Code	Name	Code	Name
0	Division by zero	10	Invalid TSS
1	Debug	11	Segment not present
3	Breakpoint	12	Stack exception
4	Overflow	13	General protection
5	Bounds check	14	Page fault
6	Invalid opcode	16	Floating point error
7	Device not available	17	Alignment check
8	Double fault	18	Machine check

Interesting Exceptions

■ 1: Debug Exception

- Fault or trap?
 - ◆ Instruction address breakpoint: fault
 - ◆ Data address breakpoint: trap
 - ◆ General detect: fault
 - ◆ Single-step: trap
 - ◆ Task-switch breakpoint: trap
- No error code pushed
- Debug registers reveal condition

Interesting Exceptions (continued)

■ 3: Breakpoint

- INT 3: one byte long
- Saved CS and EIP point to byte following breakpoint
 - ◆ Replace INT 3 with real opcode and decrement saved EIP

■ 5: Bounds Check

- Operand exceeds specified limits
- Check signed array index

■ 12: Stack Exception

- Limit violation
- Load SS with segment that is marked not present
- Always restartable

Interesting Exceptions (continued)

■ 13: General Protection

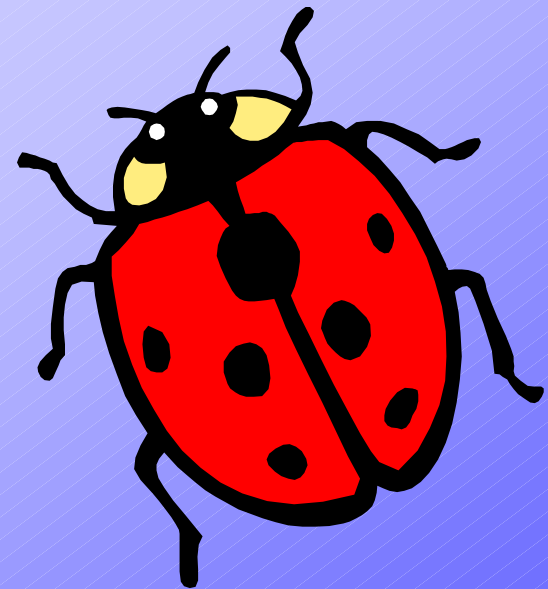
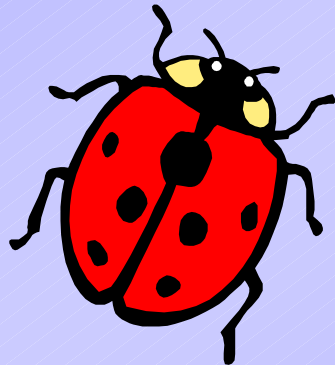
- All other protection violations
- Always a fault
- Pushes error code onto stack
 - ◆ Null is pushed unless loading a descriptor caused exception

■ 17: Alignment Check

- Access to unaligned operands
- Only generated in user-mode

Debugging Support Overview

- Debug support for memory and I/O locations
 - New for the Pentium architecture
 - Main facility is debug registers
 - Instruction and data breakpoints
 - Based on breakpoint fields



Debugging Support

- Debug interrupt vector
- Useful flags
 - Trap bit of TSS (T-bit)
 - ◆ Exception when switch to task with this bit set
 - Resume flag (RF)
 - ◆ No multiple exceptions for same instruction
 - Trap flag (TF)
 - ◆ Exception after each instruction
- Breakpoint instruction
- Interrupt vector for breakpoint exception

Debug Registers

■ Benefits

- No code patching
- Efficiency
- Works on ROM-based software
- Reads and writes of data handled

■ Debug address registers (DR0 - DR3)

- Up to four breakpoints

■ Debug control register (DR7)

- Form of memory or I/O access to trigger breakpoint

■ Debug status register (DR6)

- Conditions in effect at time of exception

Debug Exceptions

- Two interrupt vectors for debug exceptions
- Interrupt 1: Debug Exceptions
 - Handler is debugger
 - Flags in DR6 & DR7 tell what caused the exception
 - Instruction breakpoints: faults
 - Other exceptions: traps
- Interrupt 3: Breakpoint Instruction
 - Execution of INT 3 (replaces first opcode)
 - When exception handler is called return address is first byte of instruction following the INT 3

Interrupt 1

- **Instruction-breakpoint (fault)**
 - Use RF flag to restart execution (may fault again!)
- **Data Memory and I/O Breakpoints (trap)**
 - Original data overwritten before exception generated
 - ◆ Need to store data before setting breakpoint
- **General-detect fault**
 - Emulators get full control to debug registers
- **Single-step trap**
 - First instruction after TF is set does not generate exception
 - Processor clears TF before executing handler
 - INT instructions clear TF flag
- **Task-switch trap**
 - Exception occurs before first instruction in new task

Invoking a Debugger

- Two modes of invocation
 - Separate task
 - Context of current task
- I/O addresses
 - Input of byte or word
 - Output of byte, word, or doubleword
- Memory addresses
 - Read or write of byte, word, or doubleword
 - Write of byte, word, or doubleword



Invoking a Debugger (continued)

■ Execution

- Task switch
- Execution of breakpoint instruction
- Execution of any instruction
- Execution at a memory address
- Attempt to change debug register

Optimization: Alignment

- Pentium: relaxed restrictions on alignment
 - But same performance consequences as 80486
- Code alignment on cache boundary
 - 32-byte for Pentium; 16-byte for 80486
 - No substantial effect on Pentium
 - Large benefit on 80486
- Misaligned access in data cache costs 3 cycles
 - 4 byte: on 4-byte boundary
 - 2 byte: fully contained in aligned 4-byte word
 - 8-byte: 8-byte boundary (double precision reals)

Optimization: Register Usage

- Use EAX when possible
 - Many instructions 1-byte shorter
- Use DS to access data segment
 - One byte shorter than using other segments
- Use ESP to reference stack
- Load displacements into registers

Optimization: Instruction Selection

■ Integer Instruction Selection

- Use TEST to test for 0
 - ◆ ANDs operands (no register writing)
 - ◆ Check zero condition flag
- Put address calculations into loads and stores
 - ◆ Memory reference instructions have four operands
 - ◆ Using base register (not index) saves one clock cycle
- Use XOR reg,reg to clear a register
 - ◆ Watch out: sets condition codes
- Perform task switching in software
 - ◆ Smaller processor state to save/restore
- Minimize segment register loads and far pointers
 - ◆ Segment protection and the loading of segment registers are costly

