Understanding Setjmp/Longjmp

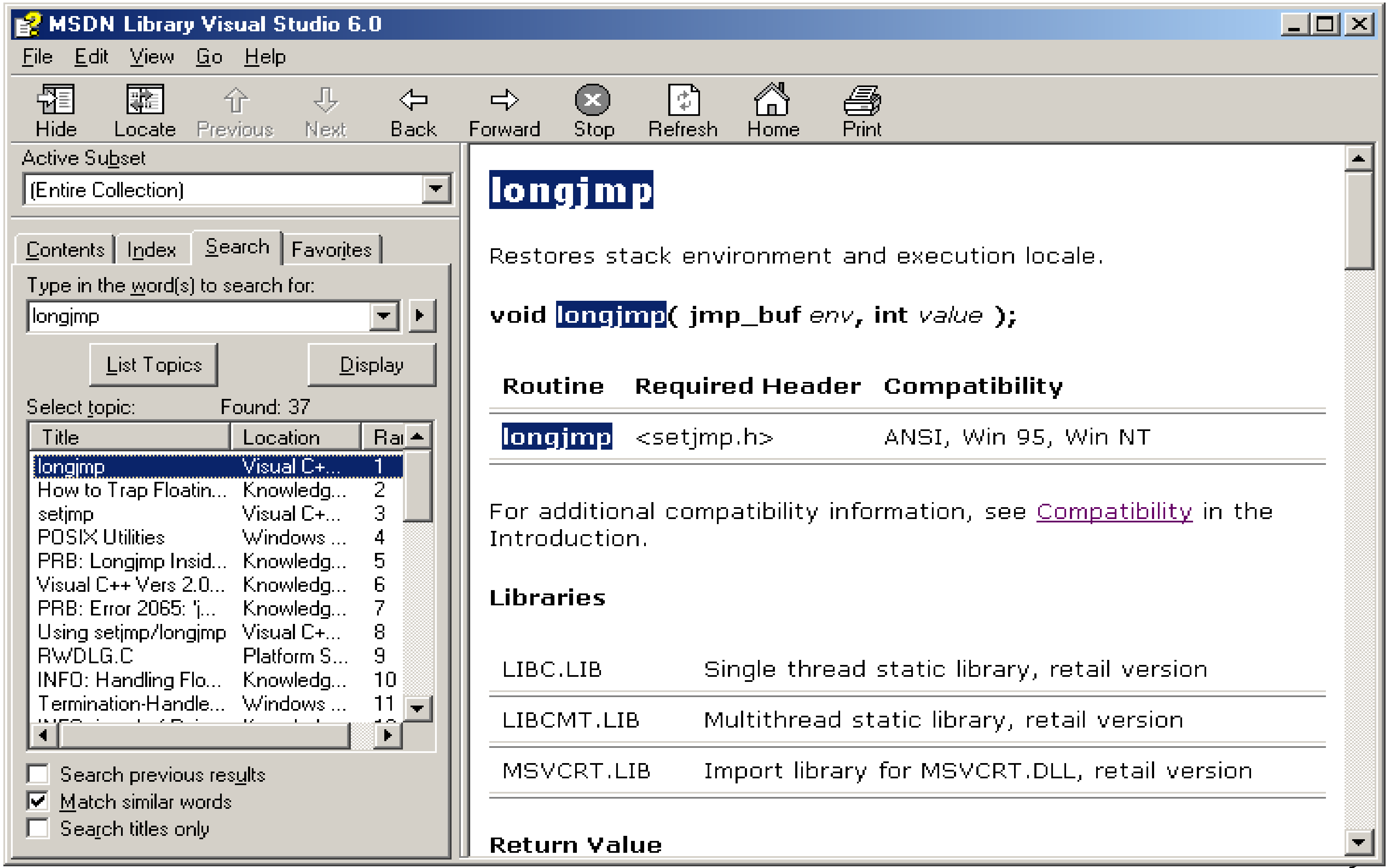
Ken Hopkinson

hopkik@cs.cornell.edu

Lab Overview

- We will be disassembling the C library functions *setjmp* and *longjmp*
- Doing so will give you an understanding of the Intel architecture, C calling conventions, stack operation, and insight into context switching within the Intel architecture

The Visual C++ Help System



Active Subset: (Entire Collection)

Type in the word(s) to search for: longjmp

Select topic: Found: 37

Title	Location	Rank
longjmp	Visual C+...	1
How to Trap Floatin...	Knowledg...	2
setjmp	Visual C+...	3
POSIX Utilities	Windows ...	4
PRB: Longjmp Insid...	Knowledg...	5
Visual C++ Vers 2.0...	Knowledg...	6
PRB: Error 2065: 'j...	Knowledg...	7
Using setjmp/longjmp	Visual C+...	8
RWDLG.C	Platform S...	9
INFO: Handling Flo...	Knowledg...	10
Termination-Handle...	Windows ...	11

□ Search previous results
☑ Match similar words
□ Search titles only

longjmp

Restores stack environment and execution locale.

void longjmp(jmp_buf *env*, int *value*);

Routine	Required Header	Compatibility
longjmp	<setjmp.h>	ANSI, Win 95, Win NT

For additional compatibility information, see Compatibility in the Introduction.

Libraries

LIBC.LIB	Single thread static library, retail version
LIBCMT.LIB	Multithread static library, retail version
MSVCRT.LIB	Import library for MSVCRT.DLL, retail version

Return Value

How to Learn 80x86 Assembly and Intel Conventions

- Links to the Intel Architecture Manuals 1-3 are on the CS 414 web page
- The February and June 1998 Microsoft Systems Journal “Under the Hood” columns by Matt Pietrek will be extremely helpful in understanding and debugging Intel assembly code generated by the Visual C++ compiler (The Microsoft System’s Journal can be found at http://www.microsoft.com/msj/)

Setjmp/Longjmp Overview

- Intel Architecture: General Introduction
- C Calling convention
- Setjmp/Longjmp Basics
- Lab Discussion

Intel Pentium Architecture

- Little endian (least significant byte located at lowest address)
- 32-bit processors
- 16 Integer Unit Registers
  - 8 32-bit General Purpose
  - 6 16-bit Segment
  - 1 32-bit Instruction Pointer
  - 1 32-bit Flag
- The floating point unit has a number of registers, too (see next slide). Our focus is on the Integer Unit.

Floating Point Unit

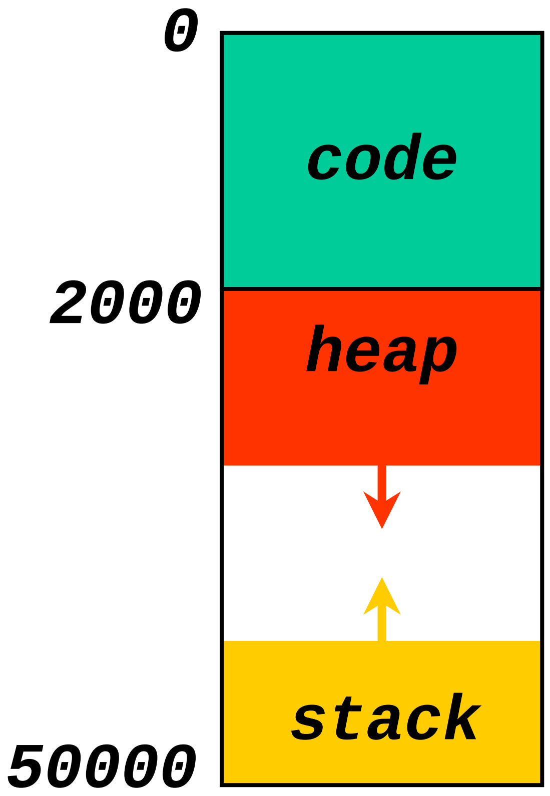
- 14 Floating-point Registers
  - 8 80-bit General Purpose
  - 1 48-bit FPU Instruction Pointer
  - 1 48-bit Operand (Data) Register
  - 1 16-bit Control Register
  - 1 16-bit Status Register
  - 1 16-bit Tag Register
  - 1 11-bit (Last Executed) Opcode Register
- The FPU stack is contained within the 8 General Purpose registers

More on Integer Registers

- General Purpose Registers are eax, ebx, ecx, edx, esi, edi, esp, and ebp
- ebp points to the base of the current stack frame
- esp points to the top of the stack
- If we want to save the current state of a program then we must save the registers it is using including the eip instruction pointer, esp and ebp stack registers, system flags, and all segment registers that might change

NT Processes and their Stacks

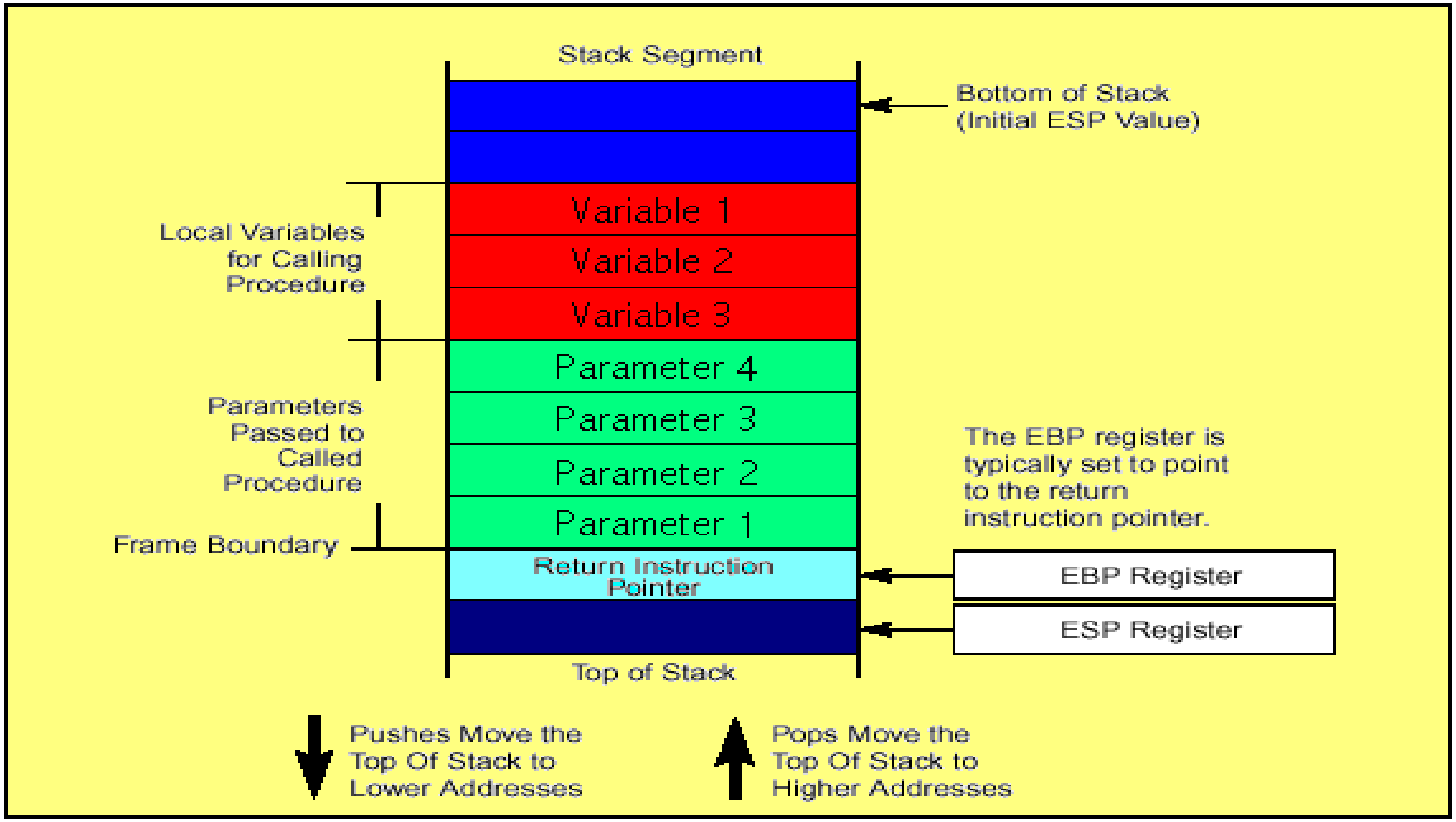
- NT initializes your process with an initial stack, heap, and code segment
- Stacks grow downward
- Dynamic data is allocated from the Heap (grows upward)



Stack Calling Conventions

- ebp points to bottom of stack frame
- Esp points to top of stack
- Function parameters pushed on stack lowest to highest var n, var n - 1, . . . var 1
- Next comes the instruction pointer eip
- One word of padding
- Local Variables local 1, 2 words of padding, local 2, 2 words of padding, . . . Local n
- Finally, the remainder can be pushed/popped to
- (Integer/Pointer) Return values always placed in eax
- Special Note: 196 bytes (49 words) of padding are placed between stack frames by VC++ since we are compiling in debug mode. Some state is also saved.
- Also Note: Visual C++ always sets ebp = esp at the beginning of a function

Sample Stack Portion (Padding Not Shown)



Setjmp/Longjmp Basics

- Setjmp saves the stack pointers (esp, ebp), some general purpose registers, and the instruction pointer into an instance of the jmp_buf data structure.
- Longjmp takes a jmp_buf instance and restores the saved register values. In effect, longjmp allows one to jump up the calling stack to any previous stack frame beginning at the next instruction past the originally called setjmp.
- Saved state in the jmp_buf structure is restored, but everything else remains unchanged. (ie if x, a local variables stored on the stack, was changed from a value of 10 before setjmp was called to a values of 12 after setjmp was called it would still have a value of 12 when longjmp was called.
- Setjmp's return value is always 0. Longjmp jumps to the assembly instruction after setjmp with a non-zero return value.

Project 1 Setjmp.C Source

```
jmp_buf mark;            /* setjmp state data structure */
void main( void ) {
    int v1, v2, v3;
    v1 = 2, v2 = 3, v3 = 4;
    jmpret = setjmp( mark );
    if( jmpret == 0 ) {
      printf("v1 = %d, v2 = %d, v3 = %d\n", v1, v2, v3);
      v1 = v2 = v3 = 222;
      longjmp(mark, -1);
    }
    else {
      printf("v1 = %d, v2 = %d, v3 = %d\n", v1, v2, v3);
    }
    return;
}
```

Setjmp.C Result

- v1 = 2, v2 = 3, v3 = 4 before setjmp is called
- v1 = v2 = v3 = 222 at the second printf statements
- If v1, v2, and/or v3 were a register variable and that register was saved in the jmp_buf structure “mark” then its value would have reverted to 2, 3, and/or 4 respectively
- jmp_ret is 0 when setjmp is called and –1 after the longjmp jump

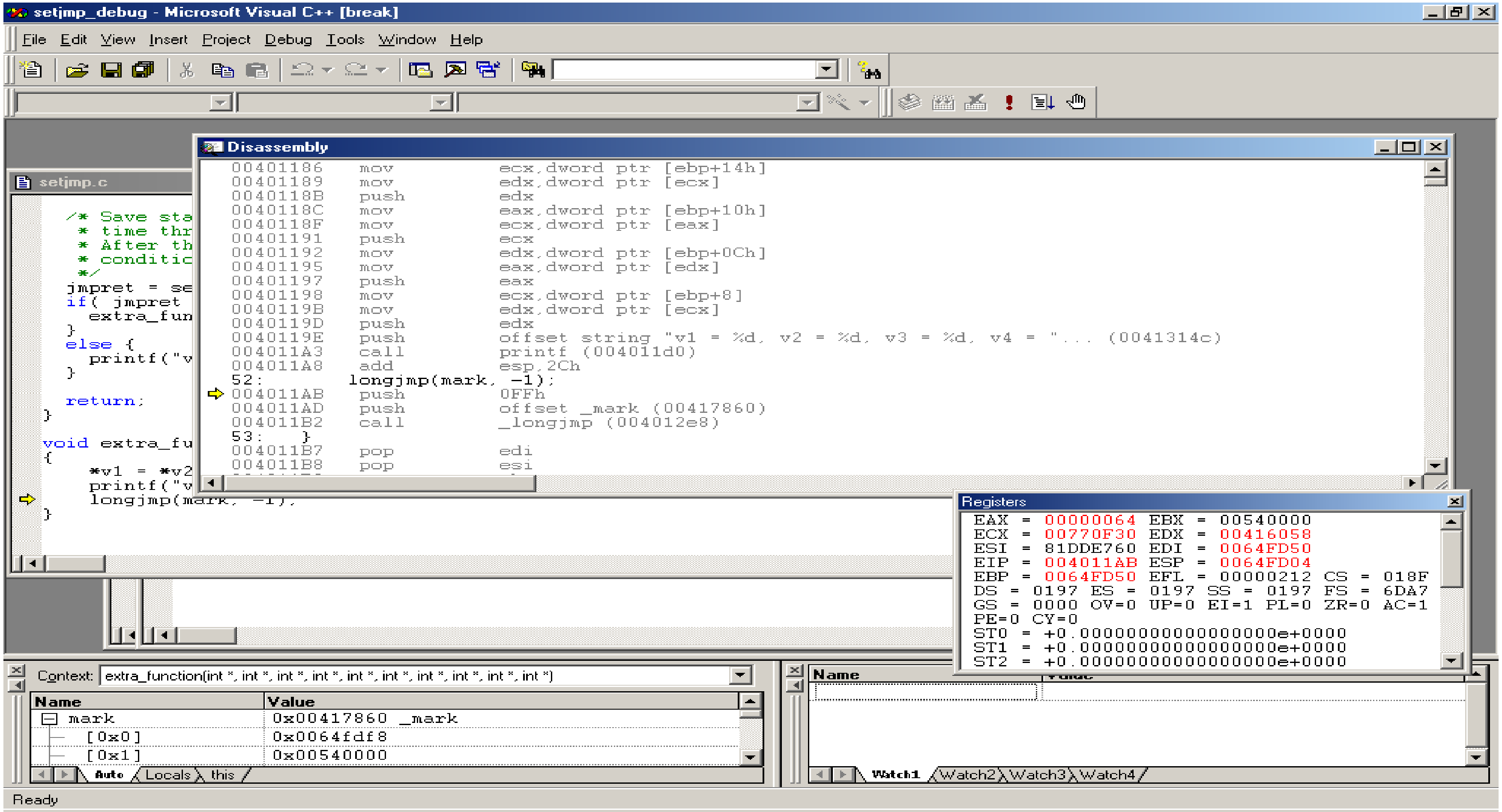
Project 1 Part A

- Download the setjmp.C example file and the project Makefile from the CS 415 web site
- Compile setjmp.exe and load it into the Visual Debugger.
- Experiment with the program (step through the code, change things to see what happens, etc). Be sure to work with the disassembly code view and NOT just the C view.
- When you are comfortable, return to the original setjmp.C code. Get a disassembly view and copy the setjmp and longjmp code (only), into a text file. Since they are macros, you will actually be copying the _setjmp3 and _longjmp functions.
- Begin labeling all assembly code in the text file to show that you understand it.

Part A Notes

- Setjmp/Longjmp have some sanity checks. You should be able to label the assembly instructions, but do not need to understand what is done inside anything called from those two functions. You should label what blocks of instructions are doing if you can determine it.
- There is also a section of setjmp/longjmp in place to work with C++ exception code. You do not need to understand everything that it is doing, but the statements themselves must be labeled.
- FYI: This section accesses the 0'th integer of the fs segment because that is where all status information is kept for the thread of execution in Windows NT.
- Special Note: Because setjmp and longjmp are macros, their disassembly differs a bit from what corresponding function calls would look like.

Part A Disassembly View


setjmp_debug - Microsoft Visual C++ [break]
File Edit View Insert Project Debug Tools Window Help
setjmp.c
Disassembly
Registers
Auto
Locals
this
Watch1
Watch2
Watch3
Watch4
Ready

```
    /* Save sta
     * time thr
     * After th
     * conditic
     */
    jmpret = se
    if( jmpret
      extra_fun
    }
    else {
      printf("v
    }

    return;
}

void extra_fu
{
      *v1 = *v2
      printf("v
      longjmp(mark, -1);
}
```

```
00401186   mov        ecx,dword ptr [ebp+14h]
00401189   mov        edx,dword ptr [ecx]
0040118B   push       edx
0040118C   mov        eax,dword ptr [ebp+10h]
0040118F   mov        ecx,dword ptr [eax]
00401191   push       ecx
00401192   mov        edx,dword ptr [ebp+0Ch]
00401195   mov        eax,dword ptr [edx]
00401197   push       eax
00401198   mov        ecx,dword ptr [ebp+8]
0040119B   mov        edx,dword ptr [ecx]
0040119D   push       edx
0040119E   push       offset string "v1 = %d, v2 = %d, v3 = %d, v4 = "... (0041314c)
004011A3   call       printf (004011d0)
004011A8   add        esp,2Ch
52:       longjmp(mark, -1);
004011AB   push       0FFh
004011AD   push       offset _mark (00417860)
004011B2   call       _longjmp (004012e8)
53:     }
004011B7   pop        edi
004011B8   pop        esi
```

```
EAX = 00000064 EBX = 00540000
ECX = 00770F30 EDX = 00416058
ESI = 81DDE760 EDI = 0064FD50
EIP = 004011AB ESP = 0064FD04
EBP = 0064FD50 EFL = 00000212 CS = 018F
DS = 0197 ES = 0197 SS = 0197 FS = 6DA7
GS = 0000 OV=0 UP=0 EI=1 PL=0 ZR=0 AC=1
PE=0 CY=0
ST0 = +0.00000000000000000e+0000
ST1 = +0.00000000000000000e+0000
ST2 = +0.00000000000000000e+0000
```

Context: extra_function(int *, int *, int *, int *, int *, int *, int *, int *)

Name	Value
⊟ mark	0x00417860 _mark
[0x0]	0x0064fdf8
[0x1]	0x00540000

Name	Value

Part A Comment Example

Call the longjmp procedure with a return value of –1

```
52:       longjmp(mark, -1);
```

Push –1 onto the stack

```
004011AB   push        0FFh
```

Push the offset of the mark structure

```
004011AD   push        offset _mark (00417860)
```

Call longjmp (underscore here since setjmp/longjmp are macros)

```
004011B2   call        _longjmp (004012e8)
```

Part B Assignment

- At the bottom of your handout are a number of setjmp/longjmp questions
- Attempt to answer each of the questions and hand in on a typed sheet of paper
- The first five questions are multiple choice. The sixth is an essay question.

Summary

- Disassemble setjmp/longjmp example
- Comment the disassembly
- Answer the setjmp/longjmp questions