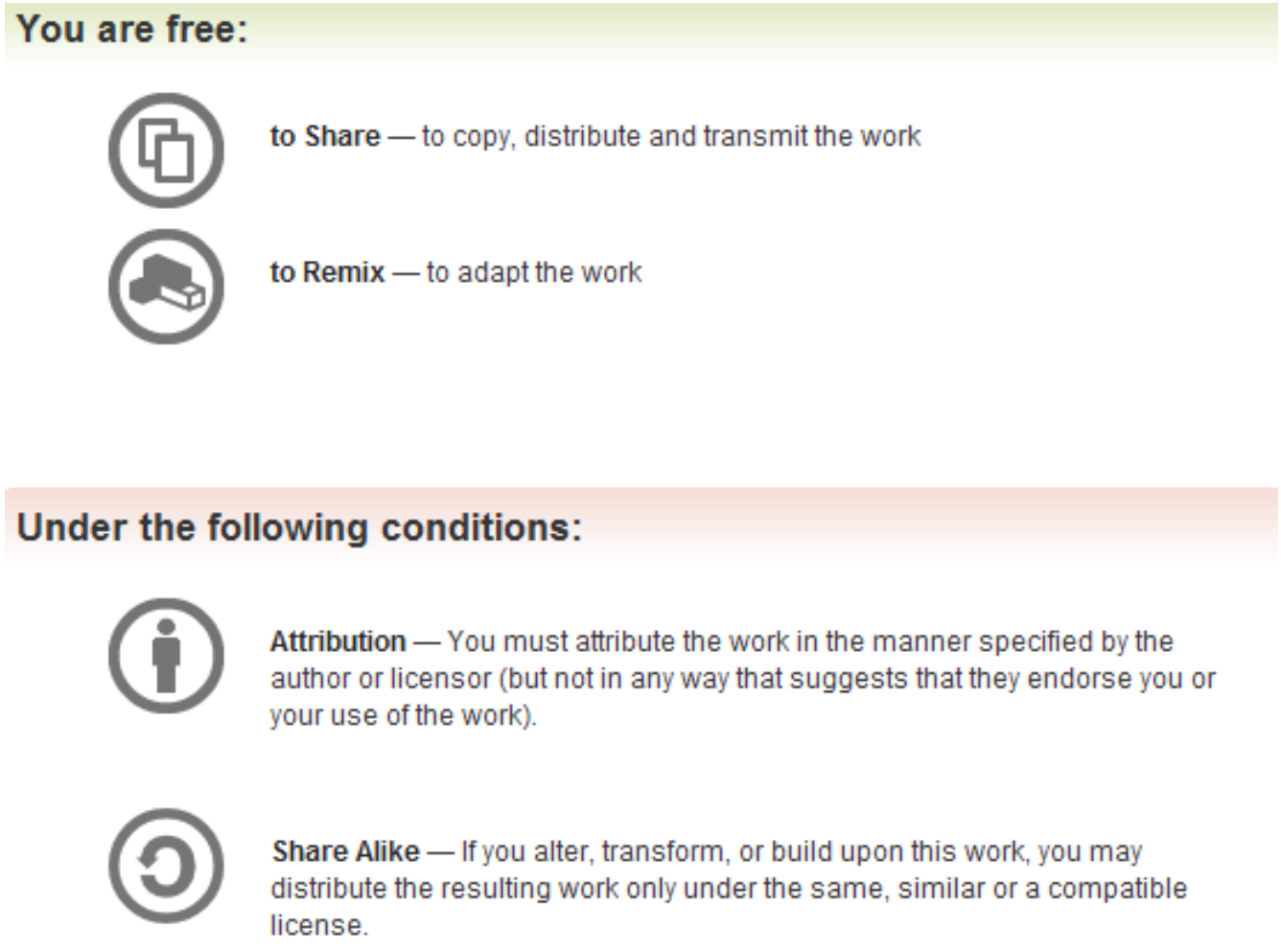
Intro x86 Part 2: More Examples and Analysis

Xeno Kovah – 2009/2010

xkovah at gmail

All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/



Control Flow

- Two forms of control flow
  - Conditional - go somewhere if a condition is met. Think “if”s, switches, loops
  - Unconditional - go somewhere no matter what. Procedure calls, goto, exceptions, interrupts.
- We’ve already seen procedure calls manifest themselves as push/call/ret, let’s see how goto manifests itself in asm.

Example2.999repeating.c:

(I missed this when I reordered slides and then didn't want to change everything else again. Also, VS orders projects alphabetically, otherwise I would have just called it GotoExample.c. Say 'lah vee' :P)

```
//Goto example
#include <stdio.h>
int main(){
        goto mylabel;
        printf("skipped\n");
mylabel:
        printf("goto ftw!\n");
        return 0xf00d;
}
```

```
00401010  push      ebp
00401011  mov       ebp,esp
00401013  jmp       00401023
00401015  push      405000h
0040101A  call      dword ptr ds:[00406230h]
00401020  add       esp,4
mylabel:
00401023  push      40500Ch
00401028  call      dword ptr ds:[00406230h]
0040102E  add       esp,4
00401031  mov       eax,0F00Dh
00401036  pop       ebp
00401037  ret
```

JMP - Jump

- Change eip to the given address
- Main forms of the address
  - Short relative (1 byte displacement from end of the instruction)
    - "jmp 00401023" doesn't have the number 00401023 anywhere in it, it's really "jmp 0x0E bytes forward"
    - Some disassemblers will indicate this with a mnemonic by writing it as "jmp short"
  - Near relative (4 byte displacement from current eip)
  - Absolute (hardcoded address in instruction)
  - Absolute Indirect (address calculated with r/m32)
- jmp -2 == infinite loop for short relative jmp :)

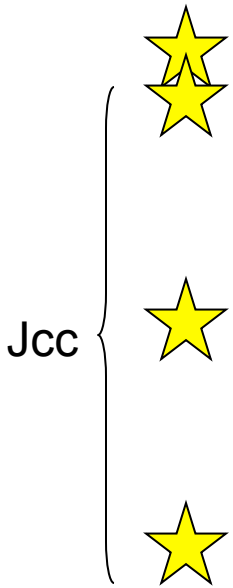
Book p. 129

Example3.c

(Remain calm)

```
int main(){
        int a=1, b=2;
        if(a == b){
                return 1;
        }
        if(a > b){
                return 2;
        }
        if(a < b){
                return 3;
        }
        return 0xdefea7;
}
```

```
main:
00401010  push    ebp
00401011  mov     ebp,esp
00401013  sub     esp,8
00401016  mov     dword ptr [ebp-4],1
0040101D  mov     dword ptr [ebp-8],2
00401024  mov     eax,dword ptr [ebp-4]
00401027  cmp     eax,dword ptr [ebp-8]
0040102A  jne     00401033
0040102C  mov     eax,1
00401031  jmp     00401056
00401033  mov     ecx,dword ptr [ebp-4]
00401036  cmp     ecx,dword ptr [ebp-8]
00401039  jle     00401042
0040103B  mov     eax,2
00401040  jmp     00401056
00401042  mov     edx,dword ptr [ebp-4]
00401045  cmp     edx,dword ptr [ebp-8]
00401048  jge     00401051
0040104A  mov     eax,3
0040104F  jmp     00401056
00401051  mov     eax,0DEFEA7h
00401056  mov     esp,ebp
00401058  pop     ebp
00401059  ret
```

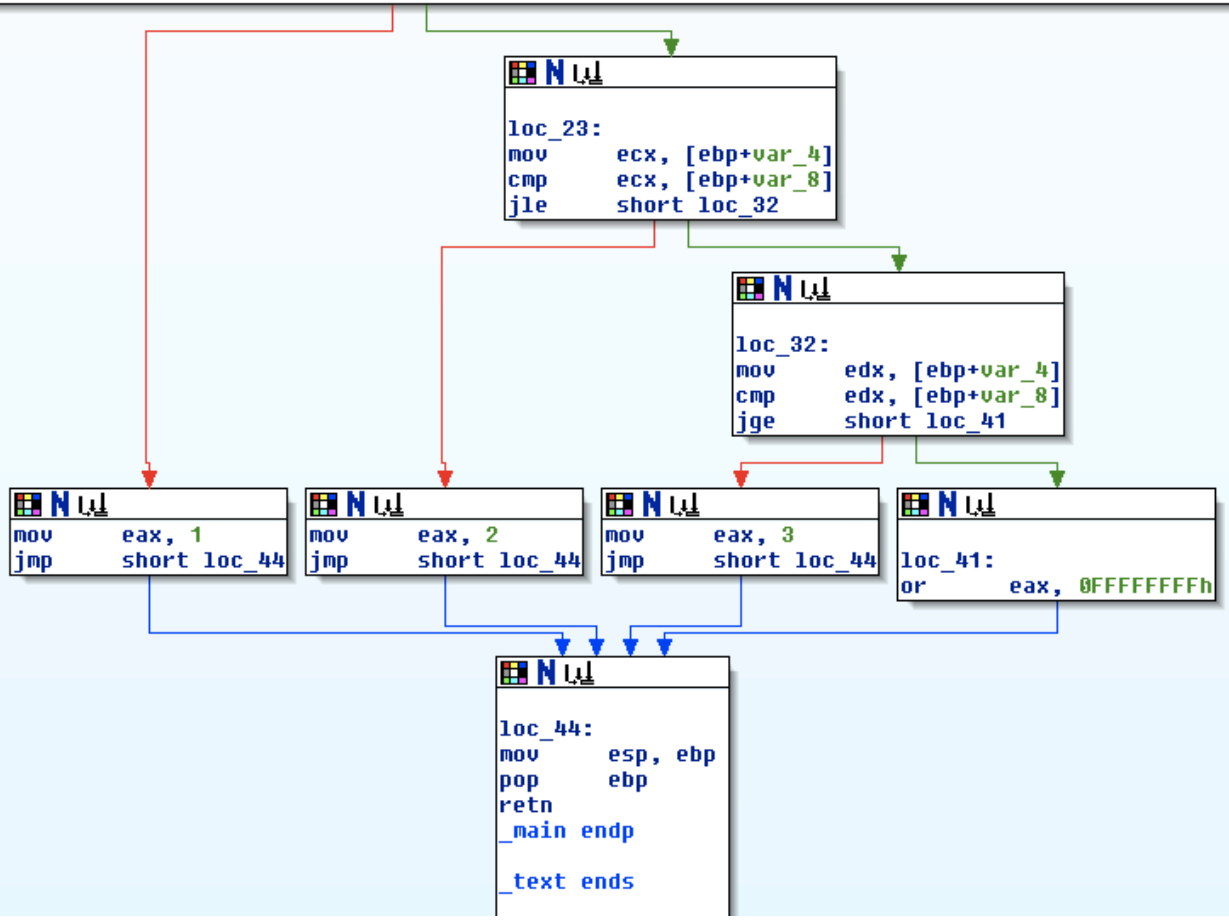


```
public _main
_main proc near

var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 8
mov     [ebp+var_4], 1
mov     [ebp+var_8], 2
mov     eax, [ebp+var_4]
cmp     eax, [ebp+var_8]
jnz     short loc_23
```

Ghost of Xmas Future:
Tools you won't get to use today
generate a Control Flow Graph (CFG)
which looks much nicer.
Not that that helps you. Just sayin' :)



Jcc - Jump If Condition Is Met

- There are more than 4 pages of conditional jump types! Luckily a bunch of them are synonyms for each other.
- JNE == JNZ (Jump if not equal, Jump if not zero, both check if the Zero Flag (ZF) == 0)

Book p. 137

Some Notable Jcc Instructions

- JZ/JE: if ZF == 1
- JNZ/JNE: if ZF == 0
- JLE/JNG : if ZF == 1 or SF != OF
- JGE/JNL : if SF == OF
- JBE: if CF == 1 OR ZF == 1
- JB: if CF == 1
- Note: Don't get hung up on memorizing which flags are set for what. More often than not, you will be running code in a debugger, not just reading it. In the debugger you can just look at eflags and/or watch whether it takes a jump.

Flag setting

- Before you can do a conditional jump, you need something to set the condition flags for you.
- Typically done with CMP, TEST, or whatever instructions are already inline and happen to have flag-setting side-effects

CMP - Compare Two Operands

- "The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction."
- What's the difference from just doing SUB? Difference is that with SUB the result has to be stored somewhere. With CMP the result is computed, the flags are set, but the result is discarded. Thus this only sets flags and doesn't mess up any of your registers.
- Modifies CF, OF, SF, ZF, AF, and PF
- (implies that SUB modifies all those too)

Book p. 138

TEST - Logical Compare

- “Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result.”
- Like CMP - sets flags, and throws away the result

Book p. 232

Example4.c

```
#define MASK 0x100

int main(){
        int a=0x1301;
        if(a & MASK){
                return 1;
        }
        else{
                return 2;
        }
}
```

```
main:
00401010  push    ebp
00401011  mov     ebp,esp
00401013  push    ecx
00401014  mov     dword ptr [ebp-4],1301h
0040101B  mov     eax,dword ptr [ebp-4]
0040101E  and     eax,100h
00401023  je      0040102E
00401025  mov     eax,1
0040102A  jmp     00401033
0040102C  jmp     00401033
0040102E  mov     eax,2
00401033  mov     esp,ebp
00401035  pop     ebp
00401036  ret
```

jcc

I actually expected a TEST, because the result isn't stored

Eventually found out why there are 2 jmps!
(no optimization, so simple compiler rules)

Refresher - Boolean (”bitwise”) logic

AND “&”

0	0	0
0	1	0
1	0	0
1	1	1

Operands **Result**

OR “|”

0	0	0
0	1	1
1	0	1
1	1	1

XOR “^”

0	0	0
0	1	1
1	0	1
1	1	0

NOT “~”

0	1
1	0

AND - Logical AND

- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate (No source *and* destination as r/m32s)

and al, bl

	00110011b (al - 0x33)
AND	01010101b (bl - 0x55)
result	00010001b (al - 0x11)

and al, 0x42

	00110011b (al - 0x33)
AND	01000010b (imm - 0x42)
result	00000010b (al - 0x02)

Book p. 231

OR - Logical Inclusive OR

- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate (No source *and* destination as r/m32s)

or al, bl

	00110011b (al - 0x33)
OR	01010101b (bl - 0x55)
result	01110111b (al - 0x77)

or al, 0x42

	00110011b (al - 0x33)
OR	01000010b (imm - 0x42)
result	01110011b (al - 0x73)

Book p. 231

XOR - Logical Exclusive OR

- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate (No source *and* destination as r/m32s)

xor al, al

	00110011b (al - 0x33)
XOR	00110011b (al - 0x33)
result	00000000b (al - 0x00)

XOR is commonly used to zero a register, by XORing it with itself, because it's faster than a MOV

xor al, 0x42

	00110011b (al - 0x33)
OR	01000010b (imm - 0x42)
result	01110001b (al - 0x71)

Book p. 231

NOT - One's Complement Negation

- Single source/destination operand can be r/m32

not al

NOT	00110011b (al - 0x33)
result	11001100b (al - 0xCC)

Xeno trying to be clever on a boring example, and failing…

not [al+bl]

al	0x10000000
bl	0x00001234
al+bl	0x10001234
[al+bl]	0 (assumed memory at 0x10001234)
NOT	00000000b
result	11111111b

Book p. 231

Example5.c - simple for loop

```
#include <stdio.h>

int main(){
   int i;
   for(i = 0; i < 10; i++){
      printf("i = %d\n“, i);
   }
}
```

What does this add say about the calling convention of printf()?

Interesting note: Defaults to returning 0

```
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  push      ecx
00401014  mov       dword ptr [ebp-4],0
0040101B  jmp       00401026
0040101D  mov       eax,dword ptr [ebp-4]
00401020  add       eax,1
00401023  mov       dword ptr [ebp-4],eax
00401026  cmp       dword ptr [ebp-4],0Ah
0040102A  jge       00401040
0040102C  mov       ecx,dword ptr [ebp-4]
0040102F  push      ecx
00401030  push      405000h
00401035  call      dword ptr ds:[00406230h]
0040103B  add       esp,8
0040103E  jmp       0040101D
00401040  xor       eax,eax
00401042  mov       esp,ebp
00401044  pop       ebp
00401045  ret
```

Instructions we now know(17)

- NOP
- PUSH/POP
- CALL/RET
- MOV/LEA
- ADD/SUB
- JMP/Jcc
- CMP/TEST
- AND/OR/XOR/NOT

Example6.c

```
//Multiply and divide transformations
//New instructions:
//shl - Shift Left, shr - Shift Right

int main(){
        unsigned int a, b, c;
        a = 0x40;
        b = a * 8;
        c = b / 16;
        return c;
}
```

```
main:
  push    ebp
  mov     ebp,esp
  sub     esp,0Ch
  mov     dword ptr [ebp-4],40h
  mov     eax,dword ptr [ebp-4]
  shl     eax,3
  mov     dword ptr [ebp-8],eax
  mov     ecx,dword ptr [ebp-8]
  shr     ecx,4
  mov     dword ptr [ebp-0Ch],ecx
  mov     eax,dword ptr [ebp-0Ch]
  mov     esp,ebp
  pop     ebp
  ret
```

SHL - Shift Logical Left

- Can be explicitly used with the C “<<” operator
- First operand (source and destination) operand is an r/m32
- Second operand is either cl (lowest byte of ecx), or a 1 byte immediate. The 2nd operand is the number of places to shift.
- It **multiplies** the register by 2 for each place the value is shifted. More efficient than a multiply instruction.
- Bits shifted off the left hand side are “shifted into” (set) the carry flag (CF)
- For purposes of determining if the CF is set at the end, think of it as n independent 1 bit shifts.

shl cl, 2

	00110011b (cl - 0x33)
result	11001100b (cl - 0xCC) CF = 0

shl cl, 3

	00110011b (cl - 0x33)
result	10011000b (cl - 0x98) CF = 1

SHR - Shift Logical Right

- Can be explicitly used with the C “>>” operator
- First operand (source and destination) operand is an r/m32
- Second operand is either cl (lowest byte of ecx), or a 1 byte immediate. The 2nd operand is the number of places to shift.
- It **divides** the register by 2 for each place the value is shifted. More efficient than a multiply instruction.
- Bits shifted off the right hand side are “shifted into” (set) the carry flag (CF)
- For purposes of determining if the CF is set at the end, think of it as n independent 1 bit shifts.

shr cl, 2

	00110011b (cl - 0x33)
result	00001100b (cl - 0x0C) CF = 1

shr cl, 3

	00110011b (cl - 0x33)
result	00000110b (cl - 0x06) CF = 0

Example7.c

```
//Multiply and divide operations
//when the operand is not a
//power of two
//New instructions: imul, div

int main(){
        unsigned int a = 1;
        a = a * 6;
        a = a / 3;
        return 0x2bad;
}
```

```
main:
  push      ebp
  mov       ebp,esp
  push      ecx
  mov       dword ptr [ebp-4],1
  mov       eax,dword ptr [ebp-4]
  imul      eax,eax,6
  mov       dword ptr [ebp-4],eax
  mov       eax,dword ptr [ebp-4]
  xor       edx,edx
  mov       ecx,3
  div       eax,ecx
  mov       dword ptr [ebp-4],eax
  mov       eax,2BADh
  mov       esp,ebp
  pop       ebp
  ret
```

IMUL - Signed Multiply

- Wait…what? Weren't the operands unsigned?
  - Visual Studio seems to have a predilection for imul over mul (unsigned multiply). I haven't been able to get it to generate the latter for simple examples.
- Three forms. One, two, or three operands
  - imul r/m32 — edx:eax = eax * r/m32
  - imul reg, r/m32 — reg = reg * r/m32
  - imul reg, r/m32, immediate — reg = r/m32 * immediate
- **Three** operands? Only one of it's kind?(see link in notes)

initial

edx	eax	r/m32(ecx)
0x0	0x44000000	0x4

eax	r/m32(ecx)
0x20	0x4

eax	r/m32(ecx)
0x20	0x4

operation

imul ecx

imul eax, ecx

imul eax, ecx, 0x6

result

edx	eax	r/m32(ecx)
0x1	0x10000000	0x4

eax	r/m32(ecx)
0x80	0x4

eax	r/m32(ecx)
0x18	0x4

DIV - Unsigned Divide

- Two forms
  - Unsigned divide ax by r/m8, al = quotient, ah = remainder
  - Unsigned divide edx:eax by r/m32, eax = quotient, edx = remainder
- If dividend is 32bits, edx will just be set to 0 before the instruction (as occurred in the Example7.c code)
- If the divisor is 0, a divide by zero exception is raised.

initial

operation

result

ax	r/m8(cx)
0x8	0x3

div ax, cx

ah	al
0x2	0x2

edx	eax	r/m32(ecx)
0x0	0x8	0x3

div eax, ecx

edx	eax	r/m32(ecx)
0x1	0x2	0x3

Example8.c

```
//VisualStudio runtime check
//buffer initialization
//auto-generated code
//New instruction: rep stos

int main(){
        char buf[40];
        buf[39] = 42;
        return 0xb100d;
}
```

Example8.c

```
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  sub       esp,30h
00401016  push      edi
00401017  lea       edi,[ebp-30h]
0040101A  mov       ecx,0Ch
0040101F  mov       eax,0CCCCCCCCh
00401024  rep stos  dword ptr es:[edi]
00401026  mov       byte ptr [ebp-5],2Ah
0040102A  mov       eax,0B100Dh
0040102F  push      edx
00401030  mov       ecx,ebp
00401032  push      eax
00401033  lea       edx,[ (401048h)]
00401039  call      _RTC_CheckStackVars (4010B0h)
0040103E  pop       eax
0040103F  pop       edx
00401040  pop       edi
00401041  mov       esp,ebp
00401043  pop       ebp
00401044  ret
```

REP STOS - Repeat Store String

- One of a family of "rep" operations, which repeat a single instruction multiple times. (i.e. "stos" is also a standalone instruction)
  - Rep isn't technically it's own instruction, it's an instruction prefix
- All rep operations use ecx register as a "counter" to determine how many times to loop through the instruction. Each time it executes, it decrements ecx. Once ecx == 0, it continues to the next instruction.
- Either moves one byte at a time or one dword at a time.
- Either fill byte at [edi] with al or fill dword at [edi] with eax.
- Moves the edi register forward one byte or one dword at a time, so that the repeated store operation is storing into consecutive locations.
- So there are 3 pieces which must happen before the actual rep stos occurs: set edi to the start destination, eax/al to the value to store, and ecx to the number of times to store

Book p. 284

rep stos setup

004113AC lea edi,[ebp-0F0h]
Set edi - the destination

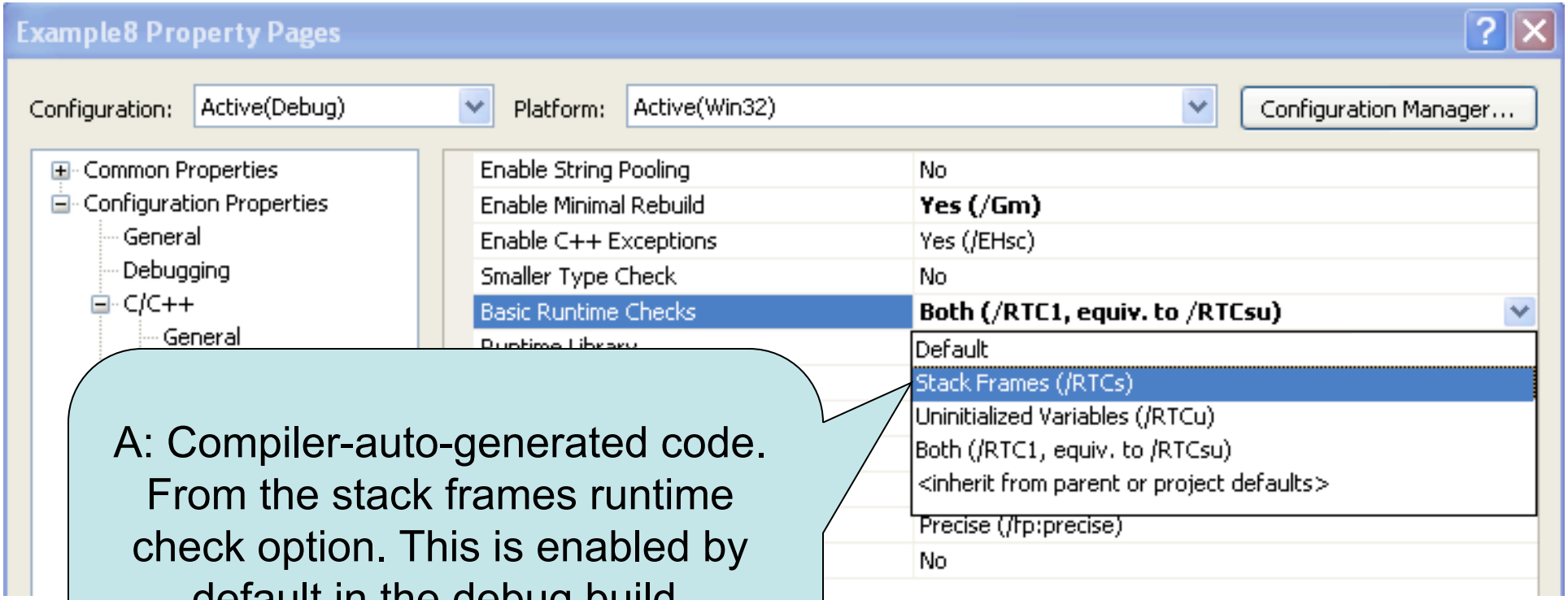
004113B2 mov ecx,3Ch
Set ecx - the count

004113B7 mov eax,0CCCCCCCCh
Set eax - the value

004113BC rep stos dword ptr es:[edi]
Start the repeated store

- So what's this going to do? Store 0x3C copies of the dword 0xCCCCCCCC starting at ebp-0xF0
- And that just happens to be 0xF0 bytes of 0xCC!

Q: Where does the rep stos come from in this example?



A: Compiler-auto-generated code. From the stack frames runtime check option. This is enabled by default in the debug build. Disabling this option removes the compiler-generated code.

More straightforward without the runtime check

```
main:
00401010  push        ebp
00401011  mov         ebp,esp
00401013  sub         esp,28h
00401016  mov         byte ptr [ebp-1],2Ah
0040101A  mov         eax,0B100Dh
0040101F  mov         esp,ebp
00401021  pop         ebp
00401022  ret
```

Example9.c
Journey to the center of memcpy()

```
//Journey to the center of memcpy
#include <stdio.h>

typedef struct mystruct{
     int var1;
    char var2[4];
} mystruct_t;

int main(){
    mystruct_t a, b;
    a.var1 = 0xFF;
    memcpy(&b, &a, sizeof(mystruct_t));
    return 0xAce0Ba5e;
}
```

```
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  sub       esp,10h
00401016  mov       dword ptr [a],0FFh
0040101D  push      8
0040101F  lea       eax,[a]
00401022  push      eax
00401023  lea       ecx,[b]
00401026  push      ecx
00401027  call      memcpy (401042h)
0040102C  add       esp,0Ch
0040102F  mov       eax,0ACE0BA5Eh
00401034  mov       esp,ebp
00401036  pop       ebp
00401037  ret
```

It begins…

```
memcpy:
   push        ebp
  mov         ebp,esp
  push        edi          ;callee save
  push        esi          ;callee save
  mov         esi,dword ptr [ebp+0Ch] ;2nd param - source ptr
  mov         ecx,dword ptr [ebp+10h] ;3rd param - copy size
  mov         edi,dword ptr [ebp+8]  ;1st param - destination ptr
  mov         eax,ecx  ;copy length to eax
  mov         edx,ecx  ;another copy of length for later use
  add         eax,esi    ;eax now points to last byte of src copy
  cmp         edi,esi    ;edi (dst) – esi (src) and set flags
  jbe         1026ED30 ;jump if ZF = 1 or CF = 1
```

;It will execute different code if the dst == src or if the destination is below (unsigned less than) the source (so jbe is an unsigned edi <= esi check)

```
1026ED30  cmp       ecx,100h    ;ecx - 0x100 and set flags
1026ED36  jb        1026ED57    ;jump if CF == 1
```
;Hmmm…since ecx is the length, it appears to do something different based on whether the length is below 0x100 or not. We could investigate the alternative path later if we wanted.

```
1026ED57  test      edi,3       ;edi AND 0x3 and set flags
1026ED5D  jne       1026ED74    ;jump if ZF == 0
```
;It is checking if either of the lower 2 bits of the destination address are set. That is, if the address ends in 1, 2, or 3. If both bits are 0, then the address can be said to be 4-byte-aligned. so it's going to do something different based on whether the destination is 4-byte-aligned or not.

```
1026ED5F  shr       ecx,2  ;divide len by 4
1026ED62  and       edx,3  ;edx still contains a copy of ecx
1026ED65  cmp       ecx,8  ;ecx - 8 and set flags
1026ED68  jb        1026ED94     ;jump if CF == 1
```

;But we currently don't get to the next instruction 1026ED6A, instead we jump to 1026ED94… :(

```
1026ED6A  rep movs    dword ptr es:[edi],dword ptr [esi]
1026ED6C  jmp         dword ptr [edx*4+1026EE84h]
```

The rep movs is the target of this expedition.

Q: But how can we reach the rep mov?

A: Need to make it so that (length to copy) / 4 >= 8, so we don't take the jump below

REP MOVS - Repeat Move Data String to String

- One of a family of “rep” operations, which repeat a single instruction multiple times. (i.e. “movs” is also a standalone instruction)
- All rep operations use ecx register as a “counter” to determine how many times to loop through the instruction. Each time it executes, it decrements ecx. Once ecx == 0, it continues to the next instruction.
- Either moves one byte at a time or one dword at a time.
- Either move byte at [esi] to byte at [edi] or move dword at [esi] to dword at [edi].
- Moves the esi and edi registers forward one byte or one dword at a time, so that the repeated store operation is storing into consecutive locations.
- So there are 3 pieces which must happen before the actual rep movs occurs: set esi to the start source, set edi to the start destination, and set ecx to the number of times to move

Book p. 274 & 278

LEAVE - High Level Procedure Exit

```
1026EE94  mov       eax,dword ptr [ebp+8]
1026EE97  pop       esi
1026EE98  pop       edi
1026EE99  leave
1026EE9A  ret
```

- "Set ESP to EBP, then pop EBP"
- That's all :)
- Then why haven't we seen it elsewhere already?
- Depends on compiler and options

Book p. 309

Some high level pseudo-code approximation

```
memcpy(void * dst, void * src, unsigned int len){
    if(dst <= src){
            //Path we didn't take, @ 1026ED28
    }
    if(dst & 3 != 0){
            //Other path we didn't take, @ 1026ED74
    }
    if((len / 4) >= 8){
            ecx = len / 4;
            rep movs dword dst, src;
    }
    else{
            //sequence of individual mov instructions
            //as appropriate for the size to be copied
    }
…
}
```


Instructions we now know(24)

- NOP
- PUSH/POP
- CALL/RET
- MOV/LEA
- ADD/SUB
- JMP/Jcc
- CMP/TEST
- AND/OR/XOR/NOT
- SHR/SHL
- IMUL/DIV
- REP STOS, REP MOV
- LEAVE

Homework

- Write a program to find an instruction we haven't covered, and report the instruction tomorrow.
- Instructions to be covered later which don't count: SAL/SAR
- Variations on jumps or the MUL/IDIV variants of IMUL/DIV also don't count
- Additional off-limits instructions: anything floating point (since we're not covering those in this class.)