Intro x86 Part 3: More Instructions

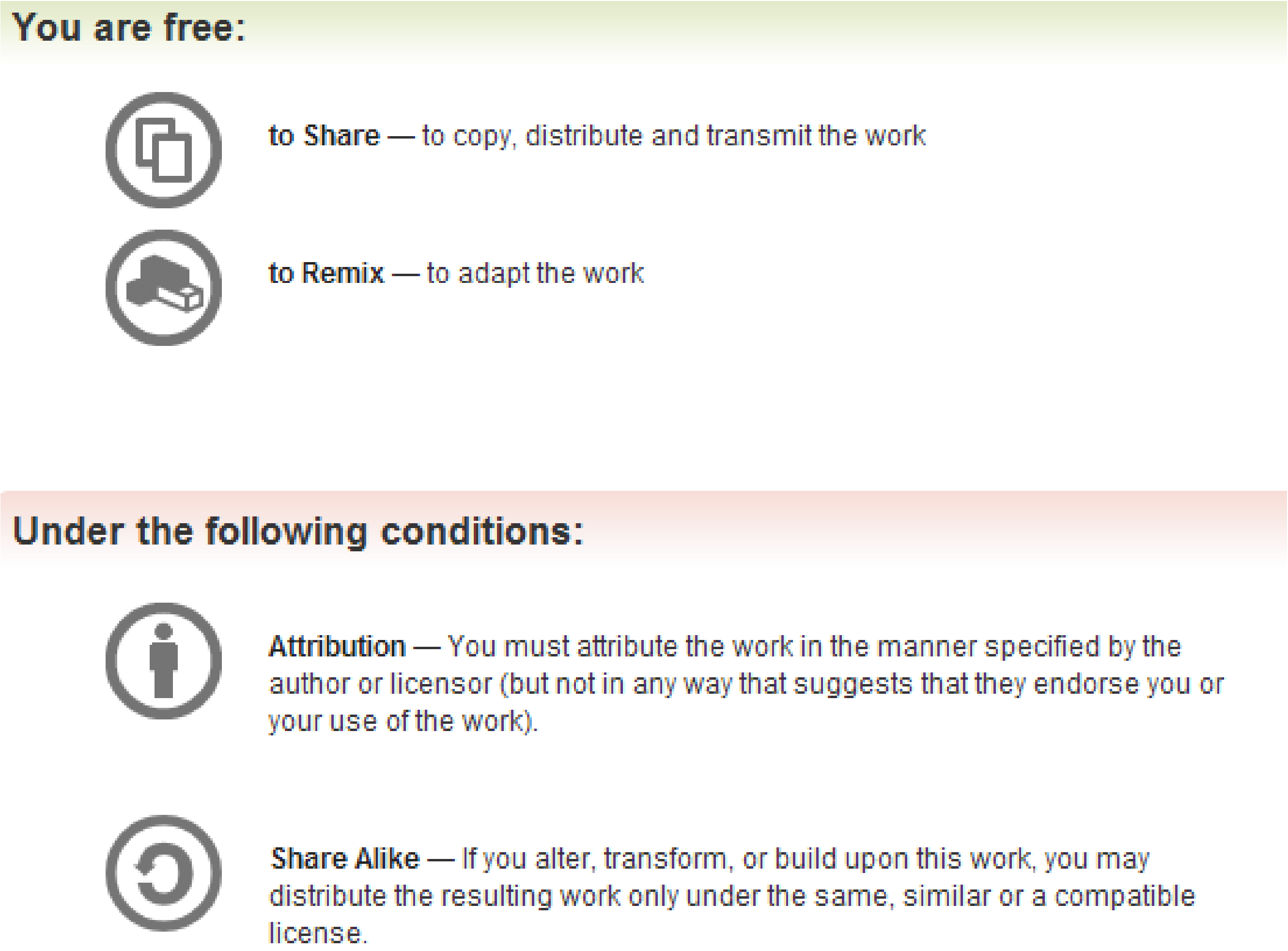
Updated slides 2021/2022

Slides derived from content made by 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/



You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

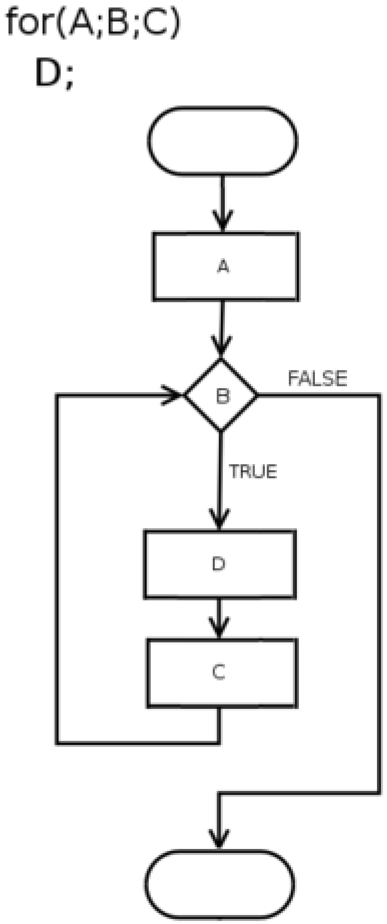
Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

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.



Resources: https://en.m.wikipedia.org/wiki/Control_flow

Example2.999repeating.c:

```
//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
```

10

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 :)

Example3.c

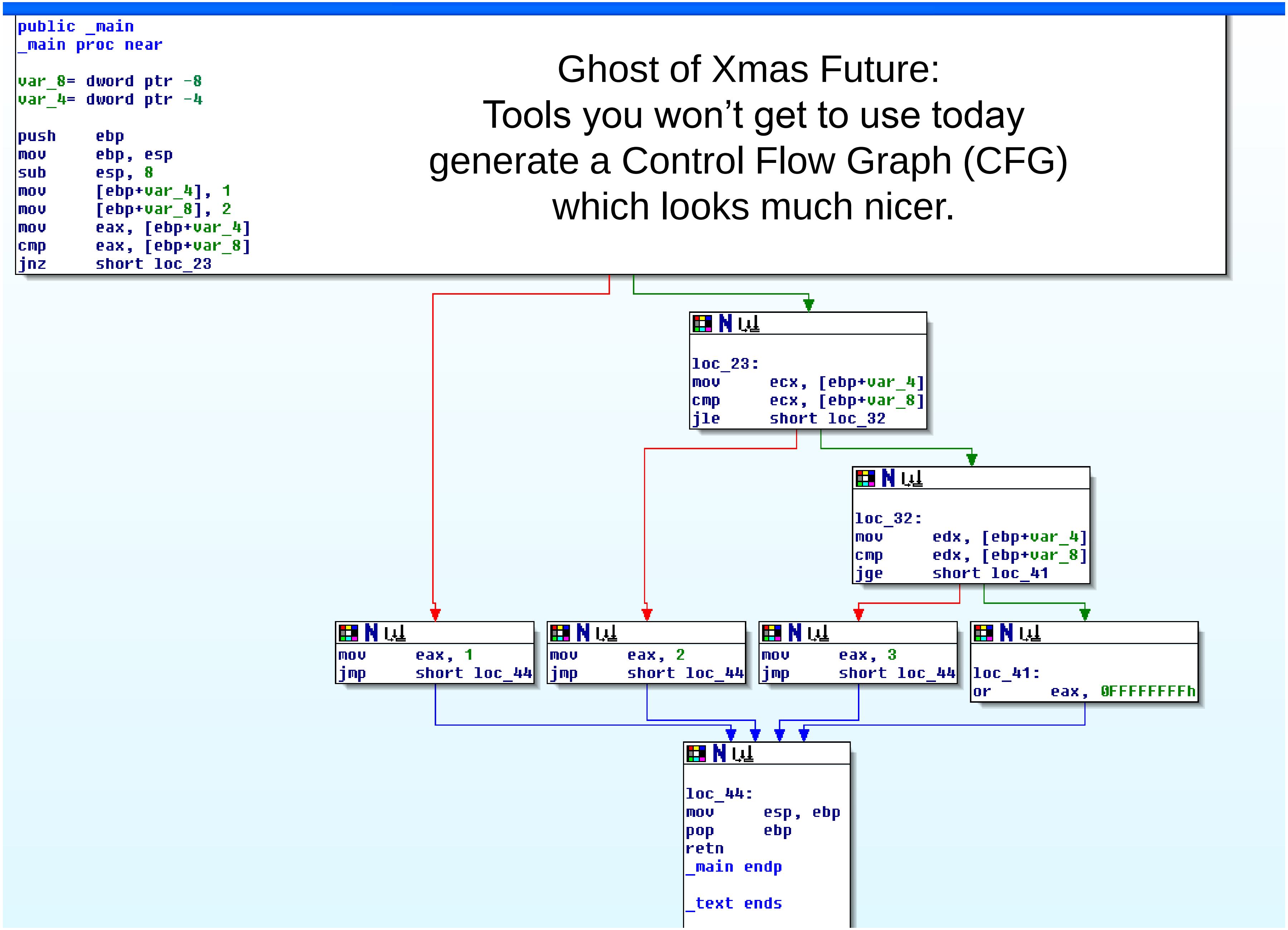
(Remain calm)

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

Jcc

```
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
```

Ghost of Xmas Future:
Tools you won’t get to use today generate a Control Flow Graph (CFG) which looks much nicer.



```
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
```

```
loc_23:
mov     ecx, [ebp+var_4]
cmp     ecx, [ebp+var_8]
jle     short loc_32
```

```
loc_32:
mov     edx, [ebp+var_4]
cmp     edx, [ebp+var_8]
jge     short loc_41
```

```
mov     eax, 1
jmp     short loc_44
```

```
mov     eax, 2
jmp     short loc_44
```

```
mov     eax, 3
jmp     short loc_44
```

```
loc_41:
or      eax, 0FFFFFFFFh
```

```
loc_44:
mov     esp, ebp
pop     ebp
retn
_main endp

_text ends
```

11

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)

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.
- Note 2: Don’t listen to that guy ^ for what were gonna be doing

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

12

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)

13

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

Example4.c

```c
#define MASK 0x100

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

jcc

```
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
```

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

14

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)

15

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)

16

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 al, 0x42

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

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

17

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

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
```

18

Book p. 224

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

19

Book p. 225

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
```

20

Book p. 218

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

21

Book p. 221

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

ax	r/m8(cx)
0x8	0x3

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

operation

div ax, cx

div eax, ecx

result

ah	al
0x2	0x2

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

24

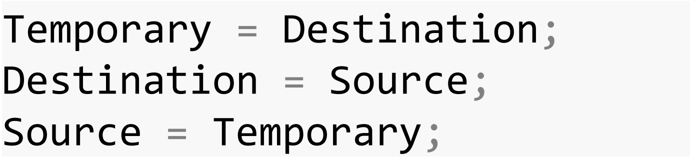
LEAVE - High Level Procedure Exit

```
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

24

XCHG



```
Temporary = Destination;
Destination = Source;
Source = Temporary;
```

- Exchanges the contents of the destination (first) and source (second) operands
- Both operands need to be a register in format
  - XCHG reg, reg
  - XCHG reg, mem
  - XCHG mem, reg
- xchg does not accept immediates, otherwise same rules apply as for mov instruction
  - NO mem to mem
- XCHG eax, eax same as a nop!
- Examples
  - XCHG ax, bx
  - XCHG

Instructions we now know!

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

Cool website to learn more about x86_64!!
(There’s also a paperback copy too)
https://www.xorpd.net/pages/xchg_rax/snip_00.html

HW: Let’s Modify our Hello World

- Lab2: Try modifying your helloworld to create a jump to the exit syscall using the jmp instruction!

Homework Cont: Lab 4

- Modify your assembly from past homeworks to implement the following C program in x86 and compile.
- Change the value in test to see if you can get it to print out “Goodbye World” instead
- Use GDB to troubleshoot any problems you’re having!

```
int main(){
        int test = 1;
        int size_hello = 12;
        int size_goodbye = 15;
        char* hello = “Hello World!”;
        char* goodbye = “Goodbye World!”;
        if(test){
                write(1, hello, size_hello);
        }else{
                write(1, goodbye, size_goodbye);
        }
        exit(0);
}
```

Homework Cont: Lab 5

- Edit your assembly from the previous homework to implement the following C program
- Try to use a mix of call and jump instructions!
- Remember you can save register values that are important to you on the stack!

```
int main(){
        int loop_count = 5;
        int size_hello = 12;
        int size_goodbye = 15;
        char* hello = “Hello World!”;
        char* goodbye = “Goodbye World!”;
        while(loop_count){
                write(1, hello, size_hello);
                loop_count--;
        }
        write(1, goodbye, size_goodbye);
        exit(0);
}
```

Homework Cont: Lab 6

- Write the following C program in x86 and compile.

```
int main(){
           int loop_count = 5;
           int size_hello = 12;
           int size_goodbye = 15;
           char* hello = “Hello World!”;
           char* goodbye = “Goodbye World!”;
           int i = 0;
           do{
                      write(1, hello, size_hello);
                      i++;
           }while(i != loop_count);
           write(1, goodbye, size_goodbye);
           exit(0);
}
```

Homework Cont: Lab 7

- Write the following C program in x86 and compile.

```
int main(){
        int loop_count = 5;
        int size_hello = 12;
        int size_goodbye = 15;
        char* hello = “Hello World!”;
        char* goodbye = “Goodbye World!”;
        for(int i = 0; i != loop_count; i++){
                write(1, hello, size_hello);
        }
        write(1, goodbye, size_goodbye);
        exit(0);
}
```

An Interesting resource which indicates the difference between different loop types and their disassembly/decompiled versions: https://en.wikibooks.org/wiki/X86_Disassembly/Loops
Check out that link for why for loops and do-while loops are sometimes indistinguishable!

Homework BONUS

- Find an interesting instruction we haven’t covered yet, and report the instruction the next time we meet!