Digital microsystems design

Marius Marcu

2020

Objectives

- Specific objectives
  - Instruction set architecture of x86 processors and assembly language

Outline

- Instruction set architecture (ISA, IA-16)
- Assembly language

Instruction set

- High level constructions provided by modern programming languages are translated by compiler tools into microprocessors' instructions
- Every assignment and arithmetic operation is implemented by an instruction or a sequence of instructions
  - Reading or writing a variable – data transfer instructions
  - Addition or subtraction – arithmetic instructions
  - Multiplication and division – arithmetic instructions
  - Logic operations – logic instructions
  - Ifs and loops constructs – branch/jump/loop instructions
  - Function calls – subroutine call and return instructions
  - System function calls – software interrupts

Instruction set

- Instruction format

  mnemonic list_of_operands

- Name of the instruction
  - Mnemonic
- List of operands
  - Destination – the location of the result
  - Source – input operands

Instruction set

- Instruction encoding
  - Binary encoding/representation of the instruction
  - Processor specific
  - Structure
    - Opcode – operation/instruction code
    - Operands encoding

Instruction

Opcode	Operands encoding

Instruction set

- Disassembly

```
    if (buff != NULL)
00DAE47C  cmp          dword ptr [buff],0
00DAE480  je           $LN13+2Bh (0DAE576h)
    {
        ptr = buff;
00DAE486  mov          eax,dword ptr [buff]
00DAE489  mov          dword ptr [ptr],eax
        while (len > 0)
00DAE48C  cmp          dword ptr [len],0
00DAE490  jbe          $LN13+2Bh (0DAE576h)
        {
            printf("  Processor mask: %lx \n", ptr->ProcessorMask);
00DAE496  mov          eax,dword ptr [ptr]
00DAE499  mov          ecx,dword ptr [eax]
00DAE49B  push         ecx
00DAE49C  push         offset string "  Processor mask: %lx \n" (0E3CE80h)
00DAE4A1  call         _printf (0DAAE86h)
00DAE4A6  add          esp,8
```

Instruction operands

- Data transfer instructions

  MOV destination, source

  - Destination and source operands can be specified in multiple ways
    - Register
    - Memory locations
    - Constants

Instruction operands

- Addressing modes
  - Specify the different ways in which instructions can access data
  - Addressing modes are provided to specify the operands of instructions

Instruction operands

- Addressing modes
  – Register addressing mode
    - Source/destination operands are values stored by processor's registers
    - Registers identities are encoded in the instruction code

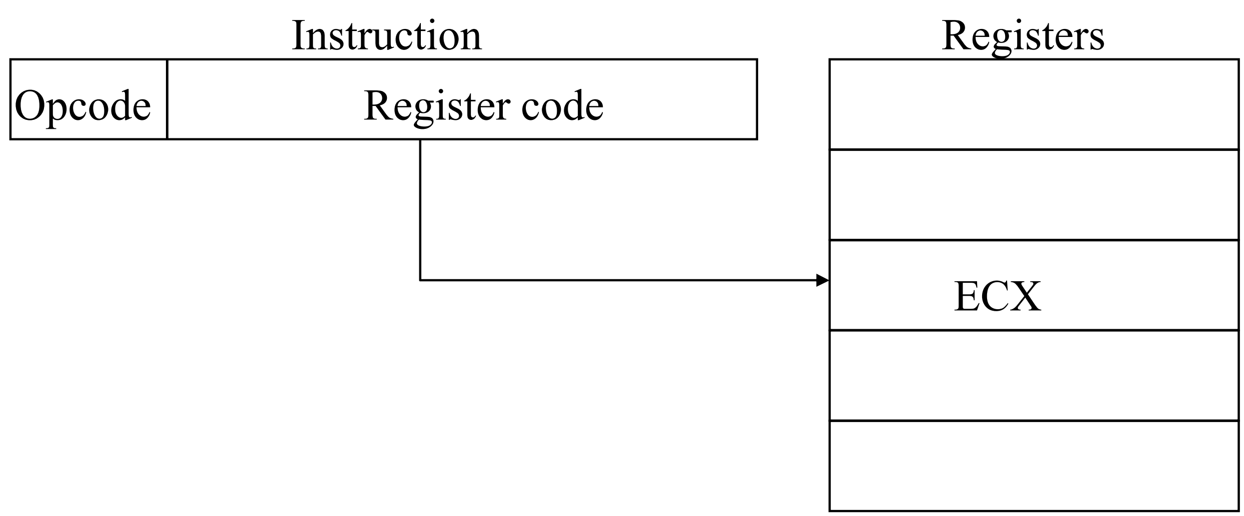
```
MOV RAX, RBX    ; 64 bits transfer
MOV EAX, EBX    ; 32 bits transfer
MOV AX, BX      ; 16 bits transfer
MOV AL, BL      ; 8 bits transfer
```

    - Used to store temporary variables (register variables)
    - Fastest access to operands

    - C/C++ register variables

Instruction operands

- Addressing modes
  - Register addressing mode
    - Opcode – machine code of the instruction
    - Register code – part of the opcode encoding the register



Instruction operands

- Addressing modes
  - Immediate addressing mode
    - The operand is specified as part of the instruction code

MOV EAX, 12345678h

MOV AX, 0AAAAH

MOV AL, 055H

- Constant values syntax

Instruction operands

- Addressing modes
  - Constants

Instruction

Opcode	Constant value

Instruction operands

- Addressing modes
  - Immediate addressing mode
    - Constant values syntax:
      - Specific to the assembler
      - Binary, decimal, octal, hexadecimal numbers
      - Distinction between identifiers and constant values
    - Examples

      0AAH – 8 bits hexadecimal number

      00101101b – 8 bits binary number

      5638 – decimal number

      2345q – octal number

      ‘a’ – ASCII character

Instruction operands

- Addressing modes
  - Memory addressing mode
    - One of the operands is located in memory
    - Specified by name of the variable or by []
    - Multiple ways to compose the logical address but all of them will use:
      - Segment
      - Offset

Instruction operands

- Addressing modes
  - Memory addressing mode - Segments
    - Implicit
      - DS for data transfers (MOV)

        MOV [BX], AX equivalent with MOV DS:[BX], AX

      - SS for stack operations (PUSH, POP, CALL, RET)
      - CS for branches (JMP, JNE, JE, etc.)
    - Explicit

      MOV ES:[BX], AX

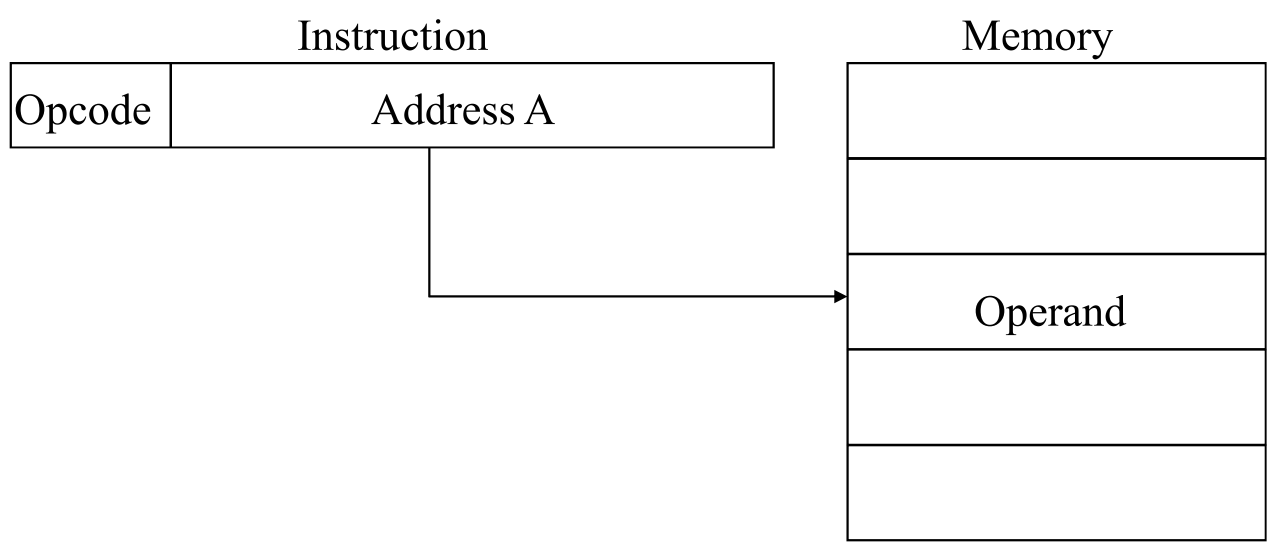
      MOV DS:[EBX], AH

Instruction operands

- Addressing modes
  - Memory addressing mode – offsets
    - Direct memory addressing
      - Effective memory address of the operand is specified directly in the instruction code
    - Indirect memory addressing
      - Effective memory address of the operand is located in an internal register or in a memory location
      - The indirect register or memory location storing the effective address is encoded in the instruction code

Instruction operands

- Addressing modes
  - Direct memory addressing



Instruction operands

- Addressing modes
  - Direct memory addressing

    MOV AX, [1000H]

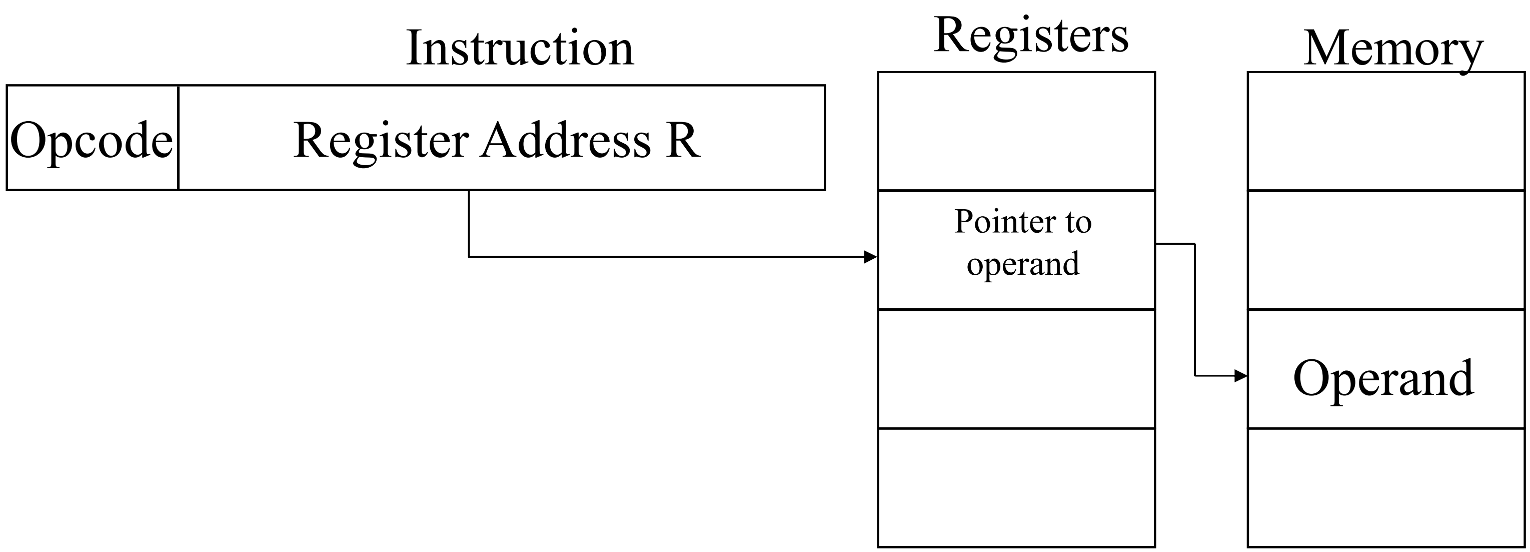
  - variables

```
cnt     db      100     ; in data segment
MOV     CL, cnt         ; in code segment
```

    - The assembler associates an address to the name of the variable

Instruction operands

- Addressing modes
  - Register indirect memory addressing



Instruction operands

- Addressing modes
  - Register indirect memory addressing
    - The effective address is present in one of the internal registers of the processor

```
MOV BX, 1000H   ; the address of the operand
MOV AX, [BX]    ; the indirect addressing using BX
```

- The content of memory location DS:[1000H] is read and stored in AX

Instruction operands

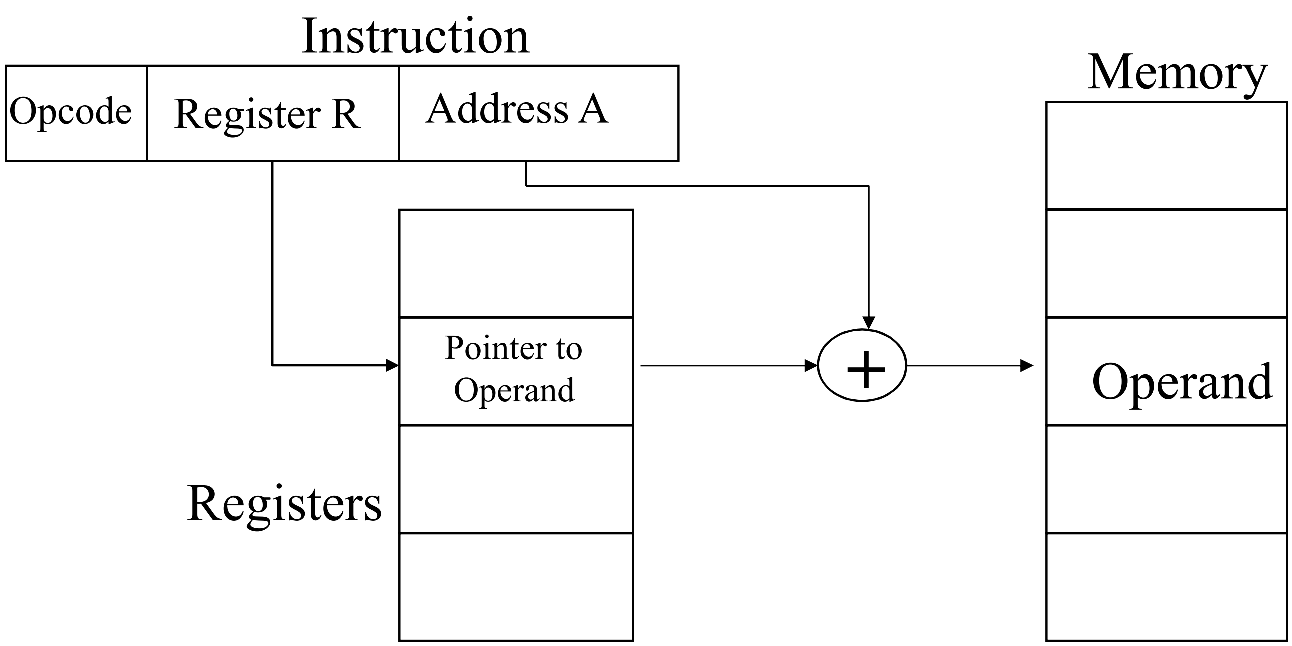
- Addressing modes
  - Register indirect memory addressing
    - Based addressing
      - Effective address is sum between content of one of the base registers BX or BP, specified in the instruction and a 16-bit offset given in the instruction

MOV AX, [BX+4]

MOV CX, 4[BX]

Instruction operands

- Addressing modes
  - Register indirect memory addressing



Instruction operands

- Addressing modes
  - Register indirect memory addressing
    - Indexed addressing
      - Effective address is sum between content of one of the index registers SI or DI, specified in the instruction and a 16-bit offset given in the instruction

```
MOV AX, [SI+4]
MOV CX, 4[SI]
```

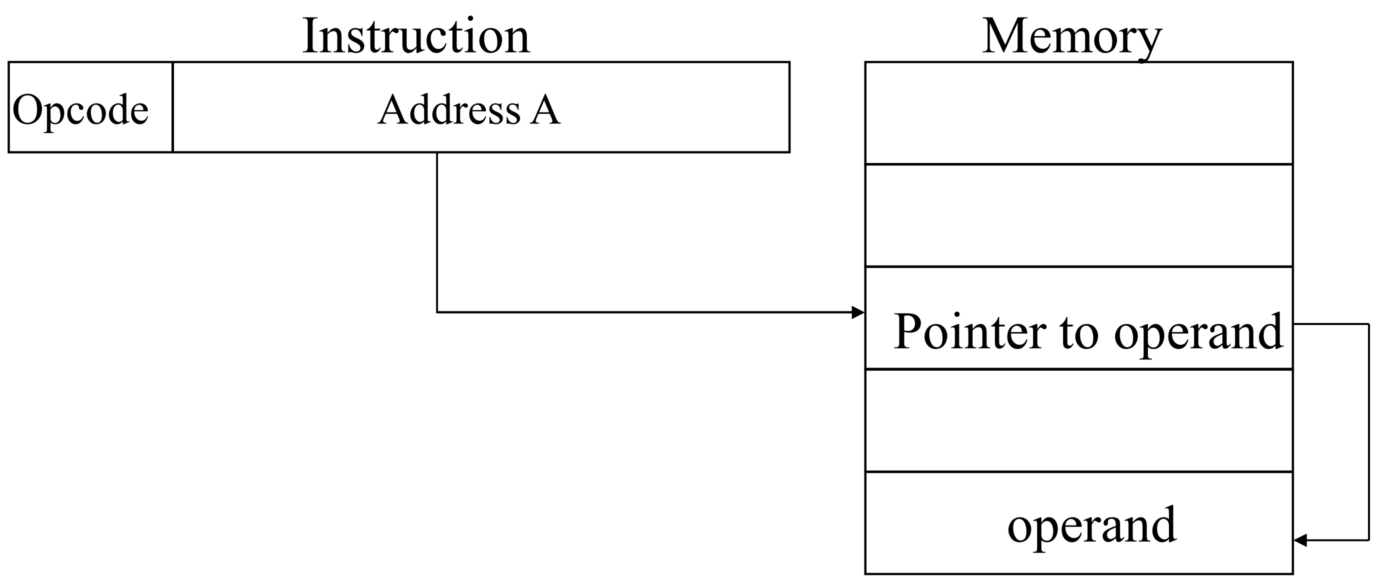
Instruction operands

- Addressing modes
  - Register indirect memory addressing
    - Based indexed addressing
      - Effective address is sum between content of one of the base registers BX or BP, one of the index registers SI or DI and a 16-bit offset given in the instruction

```
MOV AX, [BX+SI+4]
MOV CX, 4[BX][SI]
```

Instruction operands

- Addressing modes
  - Memory indirect memory addressing
    - Not used by x86 instructions



Instruction operands

- Addressing modes
  - Examples

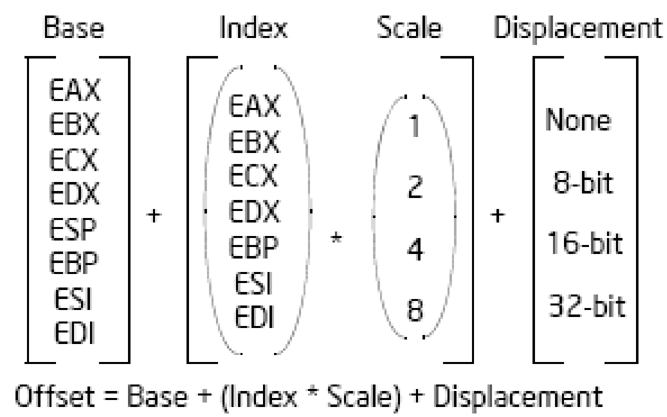
```
array_int16  dw 10 dup (?) ; array definition

xor   eax, eax
mov   esi, 2
mov   ax, array_int16
mov   ebx, offset array_int16
mov   ax, [ebx+2]
mov   ax, array_int16[esi]
mov   ax, array_int16[ebx][esi]
```

Instruction operands

C operands	ASM addressing modes	Description
constants	immediate	The operand is specified as part of the instruction code
scalar variables	direct memory addressing	The operand is located in memory, the effective memory address of the operand is specified directly in the instruction code
array variables	based indexed addressing	The array is located in memory as a sequence of bytes, the effective address is sum between content of one of the base registers
pointers	indirect memory addressing	The location of the address of the operand is encoded as part of the instruction code
register	register	The operand is located in a register, this register is encoded in the instruction cod

Summary

- Memory addressing modes
  - Based
  - Indexed
  - Displacement
  - Scale



Instruction set

- Data movement
- Arithmetic and logic
- String/array
- Control flow

Instruction set

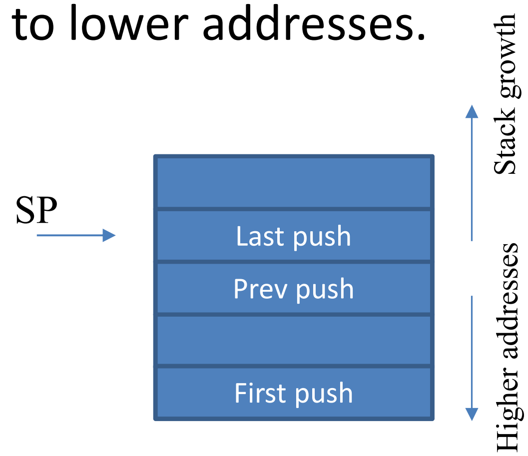
- Data movement instructions
  - mov (Move)
    - Copies the data referred to by its second operand into the location referred to by its first operand
    - Data width: 8, 16, 32, 64
    - *Syntax*

```
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
mov <reg>,<const>
mov <mem>,<const>
```

Instruction set

- Data movement instructions
  - push (Push into stack)
    - Places its operand onto the top of the hardware supported stack in memory.
      - Decrements stack pointer by 2 (16 bits) or 4 (32 bits)
      - Stores the operand at address [SP] or [ESP]
    - The stack grows from higher addresses to lower addresses.
    - *Syntax*

```
push <reg16/32>
push <mem16/32>
push <const16/32>
```



Instruction set

- Data movement instructions
  - pop (Pop from stack)
    - removes the data element from the top of the stack and stores it into the specified operand
      - Moves the 2/4 bytes located at memory location [SP] (ESP) into the specified register or memory location
      - Increments (E)SP by 2 (16 bits) or 4 (32 bits)
    - *Syntax*

```
pop <reg16/32>
pop <mem>
```

Instruction set

- Data movement instructions
  - xchg (Exchange)
    - Exchange the content of the two operands
    - *Syntax*

```
xchg <reg>,<reg>
xchg <reg>,<mem>
xchg <mem>,<reg>
```

Instruction set

- Data movement instructions
  - lea (Load effective address)
    - Places the address (offset) specified by its second operand into the register specified by its first operand
    - This is useful for obtaining a pointer into a memory region.
    - *Syntax*

```
lea <reg16/32>,<mem>
```

    - *Examples*

```
lea di, [bx+si]       ; the value BX+SI
                      ; is placed in DI.

mov eax, var          ; loads the value in var
lea ebx, var          ; loads the address of var
```

Instruction set

- Data movement instructions
  - Size of data transfers
    - Specified by source or destination register size
    - When no register is used what is the size of data to be moved?

```
mov [ebx], 2
```

Instruction set

- Data movement instructions
  - Size of data transfers
    - Specified by source or destination register size
    - When no register is used data size has to be specified explicitly (by data size specifiers)

```
mov BYTE PTR [ebx], 2     ; Move 2 into the single byte at
                          ; the address stored in EBX
mov WORD PTR [ebx], 2     ; Move the 16-bit integer 2 into
                          ; the 2 bytes at the address in EBX
mov DWORD PTR [ebx], 2    ; Move the 32-bit integer 2 into the
                          ; 4 bytes starting at the address in EBX
```

Instruction operands

- Transfer size
  - Examples

```
array_int16   dw 10 dup (?) ; array definition

mov    esi, 2
mov    ax, array_int16
mov    eax, array_int16     ; type mismatch
mov    ebx, offset array_int16
mov    ax, [ebx+2]
mov    eax, [ebx+2]
mov    ax, array_int16[esi]
mov    ax, array_int16[ebx][esi]
mov    [ebx], 12            ; undefined data size
                            ; specifier must be used
```

Instruction set

- Arithmetic and logic instructions
  - Perform arithmetical and logical operations on integer numbers

Instruction set

- Arithmetic and logic instructions
  - add (Integer Addition) / sub (Integer Subtraction)
    - Adds together its two operands, storing the result in its first operand
    - *Syntax*

```
add <reg>,<reg>      sub <reg>,<reg>
add <reg>,<mem>      sub <reg>,<mem>
add <mem>,<reg>      sub <mem>,<reg>
add <reg>,<con>      sub <reg>,<con>
add <mem>,<con>      sub <mem>,<con>
```

Instruction set

- Arithmetic and logic instructions
  - adc (Integer Addition with Carry) / sbb (Integer Subtraction with Borrow)
    - Adds together its two operands and the carry flag (CF), storing the result in its first operand
    - *Syntax*

```
adc <reg>,<reg>      sbb <reg>,<reg>
adc <reg>,<mem>      sbb <reg>,<mem>
adc <mem>,<reg>      sbb <mem>,<reg>
adc <reg>,<con>      sbb <reg>,<con>
adc <mem>,<con>      sbb <mem>,<con>
```

Instruction set

- Arithmetic and logic instructions
  - inc / dec Increment, Decrement
    - Increments/ decrements the contents of its operand by on
    - *Syntax*

```
inc <reg>
inc <mem>
dec <reg>
dec <mem>
```

Instruction set

- Arithmetic and logic instructions
  - imul Integer multiplication

    DX:AX = AX * <reg16>/<mem16>

    EDX:EAX = EAX * <reg32>/<mem32>

    - *Syntax*

      ```
      imul <reg>/<mem>
      ```
  - idiv Integer division

    AX = DX:AX / <reg16>/<mem16>

    DX = DX:AX % <reg16>/<mem16>

    EAX = EDX:EAX / <reg32>/<mem32>

    EDX = EDX:EAX % <reg32>/<mem32>

    - *Syntax*

      ```
      idiv <reg>/<mem>
      ```

Instruction set

- Arithmetic and logic instructions
  - and, or, xor Bitwise logical and, or and exclusive or
    - perform the specified logical operation (logical bitwise and, or, and exclusive or, respectively) on their operands, placing the result in the first operand location.
    - *Syntax*

```
and <reg>,<reg>   (or, xor)
and <reg>,<mem>
and <mem>,<reg>
and <reg>,<con>
and <mem>,<con>
```

Instruction set

- Arithmetic and logic instructions
  - not Bitwise Logical Not
    - Logically negates the operand contents
    - *Syntax*

```
not <reg>
not <mem>
```

Instruction set

- Arithmetic and logic instructions
  - neg Negate
    - Performs the two's complement negation of the operand contents.
    - *Syntax*

```
neg <reg>
neg <mem>
```

Instruction set

- Arithmetic and logic instructions
  - shl, shr — Shift Logically Left, Shift Logically Right
    - shift the bits in their first operand's contents left and right, padding the resulting empty bit positions with zeros.
    - Syntax

```
shl <reg>,<con8>
shl <mem>,<con8>
shl <reg>,<cl>
shl <mem>,<cl>

shr <reg>,<con8>
shr <mem>,<con8>
shr <reg>,<cl>
shr <mem>,<cl>
```

Instruction set

- Arithmetic and logic instructions
  - sal, sar — Shift Arithmetically Left, Shift Arithmetically Right
    - shift the bits in their first operand's contents left and right, padding the resulting empty bit positions with the sign bit (right) or zero (left).
    - Syntax

```
sal <reg>,<con8>
sal <mem>,<con8>
sal <reg>,<cl>
sal <mem>,<cl>

sar <reg>,<con8>
sar <mem>,<con8>
sar <reg>,<cl>
sar <mem>,<cl>
```

Instruction set

- Arithmetic and logic instructions
  - rol, ror — Rotate Left, Rotate Right
    - rotate the bits in their first operand's contents left and right, padding the resulting empty bit positions with the bits leaving the register.
    - Syntax

```
rol <reg>,<con8>
rol <mem>,<con8>
rol <reg>,<cl>
rol <mem>,<cl>

ror <reg>,<con8>
ror <mem>,<con8>
ror <reg>,<cl>
ror <mem>,<cl>
```

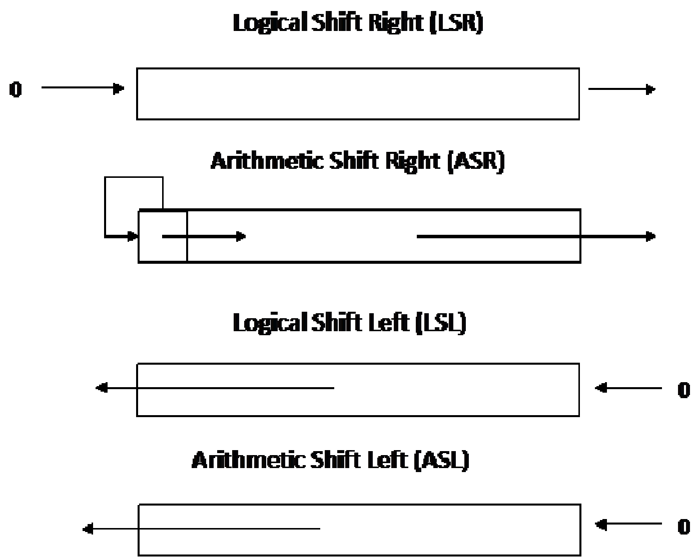
Instruction set

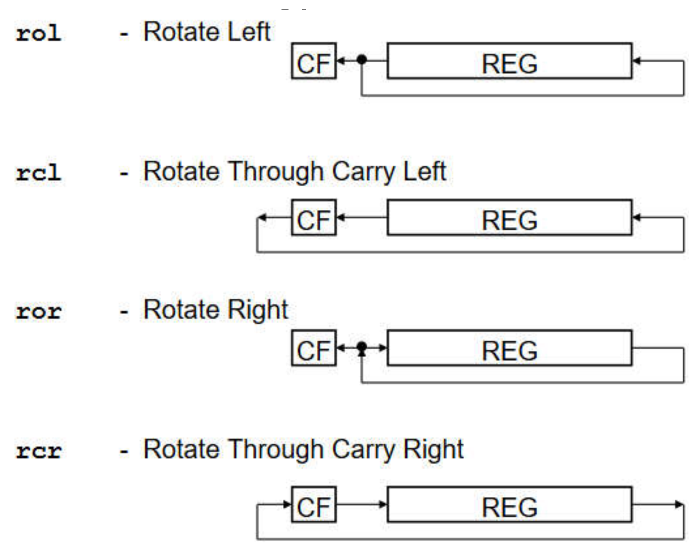
- Arithmetic and logic instructions
  - rcl, rcr — Rotate Left, Rotate Right with carry
    - rotate the bits in their first operand's contents left and right, padding the resulting empty bit positions with the bits leaving the register, including the CF in the loop
    - Syntax

```
rcl <reg>,<con8>
rcl <mem>,<con8>
rcl <reg>,<cl>
rcl <mem>,<cl>

rcr <reg>,<con8>
rcr <mem>,<con8>
rcr <reg>,<cl>
rcr <mem>,<cl>
```

Instruction set

- Arithmetic and logic instructions





Instruction set

- String/Array instructions
  - Move one byte/word/double from string to string
  - Similar to memcpy()

```
MOVSB
MOVSW
MOVSD

(byte/word/dword ptr)ES:[EDI++] = DS:[ESI++]
```

  - Move direction: DF

Instruction set

- String/Array instructions
  - Move multiple byte/word/double from string to string
  - Similar to memcpy()

```
REP MOVSB
REP MOVSW
REP MOVSD
```

```
while ( ECX > 0 )
    if ( DF == 0 )
        (byte/word/dword ptr)ES:[EDI++] = DS:[ESI++]
    else
        (byte/word/dword ptr)ES:[EDI--] = DS:[ESI--]
```

Instruction set

- String/Array instructions
  - Store byte/word/double to string
  - Similar to memset ()

```
REP STOSB
REP STOSW
REP STOSD
```

```
while ( ECX > 0 )
    if ( DF == 0 )
        (byte/word/dword ptr)ES:[EDI++] = AL/AX/EAX
    else
        (byte/word/dword ptr)ES:[EDI--] = AL/AX/EAX
```

Instruction set

- String/Array instructions
  - Compare byte/word/double in memory
  - Similar to memcmp()

```
REPE/REPNE CMPSB
REPE/REPNE CMPSW
REPE/REPNE CMPSD
```

```
ZF=1
while ( ECX > 0 )
    if ( DF == 0 )
        if( (byte/word/dword ptr)ES:[EDI++] != DS:[ESI++] ) ZF=0,
        break;
    else
        if( (byte/word/dword ptr)ES:[EDI--] != DS:[ESI--] ) ZF=0,
        break;
```

Instruction set

- Control flow instructions
  - Data transfer and arithmetical and logic instructions are executed sequentially one after another
  - Exception from linear execution are branches and loops
    - They are implemented by jump instructions
  - IP (Instruction Pointer) register keeps the address of the following instruction that has to be fetched
    - IP increments after the current instruction is fetched by the processor

Instruction set

- Control flow instructions
  - The IP register cannot be manipulated directly, it is updated instead by the mean of control flow instructions
  - Jump instructions change the linear execution of the program
    - Set a label to the destination instruction the processor has to execute next
    - Jump instructions load IP register with the address specified by the label

Instruction set

- Control flow instructions
  - Labels can be inserted anywhere in assembly code by entering a label name followed by a colon.

```
          mov esi, [ebp+8]
begin:    xor ecx, ecx
          mov eax, [esi]
```

  - Control flow instructions refers to a label where from the execution flow will continue

Instruction set

- Control flow instructions
  - jmp Jump
    - Transfers program control flow to the instruction at the memory location indicated by the operand.
    - Unconditional jump
  - *Syntax*
    jmp <label>
  - *Example*

```
jmp begin ; Jump to the instruction labeled begin.
```

Instruction set

- Control flow instructions
  - j condition – Conditional jump
    - Change the linear control flow if the condition is true
  - *Syntax*

```
je <label> (jump when equal)
jne <label> (jump when not equal)
jz <label> (jump when last result was zero)
jnz <label> (jump when last result was not zero)
jg <label> (jump when greater than)
jge <label> (jump when greater than or equal to)
jl <label> (jump when less than)
jle <label> (jump when less than or equal to)
```

Instruction set

- Control flow instructions
  - call Subroutine call
    - Actions:
      - pushes the current code address onto the hardware supported stack in memory
      - performs an unconditional jump to the code location indicated by the label operand.
    - Unlike the simple jump instructions, the call instruction saves the location to return to when the subroutine completes.
  - *Syntax*

```
call <label>
```

Instruction set

- Control flow instructions
  - ret Return from subroutine
    - This instruction
      - first pops a code location off the hardware supported in-memory stack.
      - It then performs an unconditional jump to the retrieved code location.
  - *Syntax*

    ```
    ret
    ```

Instruction set

- Status flags register control
  - Flags register cannot be changed directly
  - Flags will be set based on previous arithmetic or logic instruction

Instruction set

- Flags control instructions
  - Clear/Set Carry/Direction/Interrupt flags

```
CLC
CLD
CLI

STC
STD
STI
```

Instruction set

- Status flags register control
  - cmp Compare
    - Compare the values of the two specified operands
    - Sets the condition flags in the machine status word according to the subtraction of the two values
  - *Syntax*

```
cmp <reg>,<reg>
cmp <reg>,<mem>
cmp <mem>,<reg>
cmp <reg>,<con>
```

  - *Example*

```
cmp DWORD PTR [var], 10
jeq loop
```

Instruction set

- Status flags register control
  - test Test of the values of the operands
    - Logic and of the values
    - Sets the condition flags in the machine status word according to the and of the two values
  - *Syntax*

```
test <reg>,<reg>
test <reg>,<mem>
test <mem>,<reg>
test <reg>,<con>
```

Assembly language

- Subroutine calling convention
  - Conventions adopted by assembly language programmers and compiler designers to share the same implementation patterns
  - The calling convention is a protocol about how to call and return from routines.
    - a programmer does not need to examine the implementation of a subroutine to determine how parameters should be passed to that subroutine
    - high-level language compilers can be made to follow the rules
    - allowing hand-coded assembly language routines and high-level language routines to call one another.

Assembly language

- Subroutine calling convention
  - many calling conventions are possible
    - C language calling convention
  - Specifies how parameters and return values are passed between the caller and the callee
    - Subroutine parameters are passed on the stack

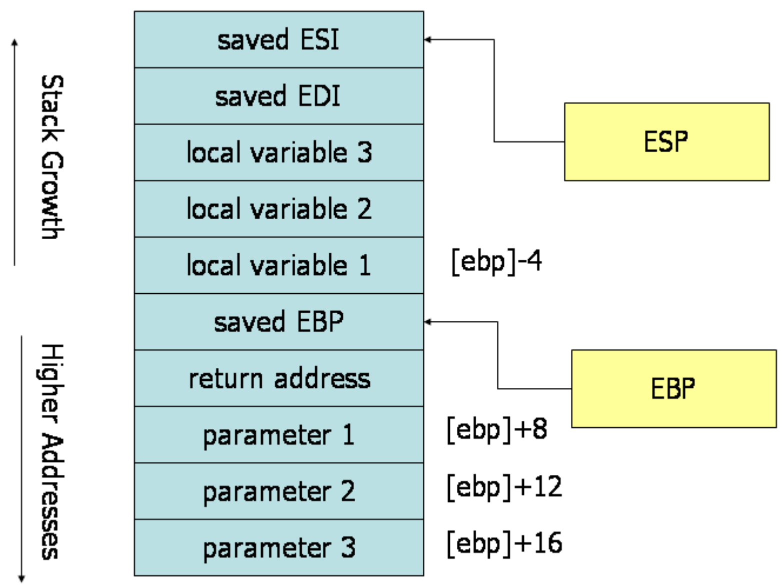
Assembly language

- Example
  - Do you remember the order of C function call parameters evaluation?

```
_myFunc(a+b, ++a, b++);
```

Assembly language

- Subroutine calling convention
  - Stack content during subroutine execution



Assembly language

- Caller rules
  - Before calling a subroutine, the caller should save the contents of certain registers that are designated *caller-saved*
  - To pass parameters to the subroutine, push them onto the stack before the call.
    - Reverse order (last parameter first)
  - To call the subroutine, use the call instruction.
    - This instruction places the return address on top of the parameters on the stack, and branches to the subroutine code.

Assembly language

- Example
  - Function declaration

```
int _myFunc (int a, int b, int c);
```

  - Function call

```
_myFunc(eax, 216, var);
```

  - Function definition

```
int _myFunc(int a, int b, int c) {
        int x, y, z;
        …
        return …;
}
```

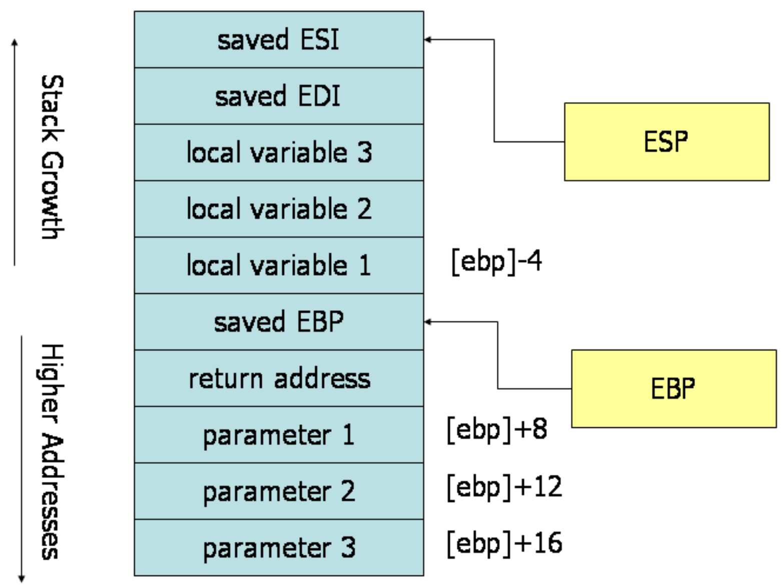
Assembly language

- Example

```
push [var] ; Push last parameter first
push 216   ; Push the second parameter
push eax   ; Push first parameter last

call _myFunc ; Call the function
              ;(assume C naming)

add esp, 12  ; remove parameters from stack
```

Assembly language

- Subroutine calling convention
  - Stack content during subroutine execution - Example



Assembly language

- Callee rules (start of the subroutine)
  - Save and initialize EBP – it is used by convention as a point of reference for finding parameters and local variables on the stack.
  - Allocate local variables by making space on the stack.
  - Save the values of the *callee-saved* registers that will be used by the function must be saved.

Assembly language

- Callee rules (return from subroutine)
  - Leave the return value in EAX.
  - Restore the old values of any callee-saved registers that were modified
  - Deallocate local variables.
  - Restore the caller's base pointer value by popping EBP off the stack
  - Return to the caller by executing a ret instruction

Assembly language

- Example

```
_myFunc PROC
; Subroutine Prologue
  ; Save the old base pointer value.
  push ebp
  ; Set the new base pointer value.
  mov ebp, esp
  ; Make room for a 4-byte local var.
  sub esp, 4
  ; Save the values of registers
  ;that the function will modify
  push edi
  ; This function uses EDI and ESI.
  push esi
```

```
; Subroutine Body
…

; Subroutine Epilogue
  pop esi      ; Recover register values
  pop edi
  ; Deallocate local variables
  mov esp, ebp
  ; Restore the caller's base pointer
  pop ebp
  ret
_myFunc ENDP
```

Assembly language

- Example

```
; Subroutine Body
  ; Move value of parameter 1 into EAX
  mov eax, [ebp+8]
  ; Move value of parameter 2 into ESI
  mov esi, [ebp+12]
  ; Move value of parameter 3 into EDI
  mov edi, [ebp+16]
  ; Move EDI into the local variable
  mov [ebp-4], edi
  ; Add ESI into the local variable
  add [ebp-4], esi
  ; Add the result into EAX
  add eax, [ebp-4]
```

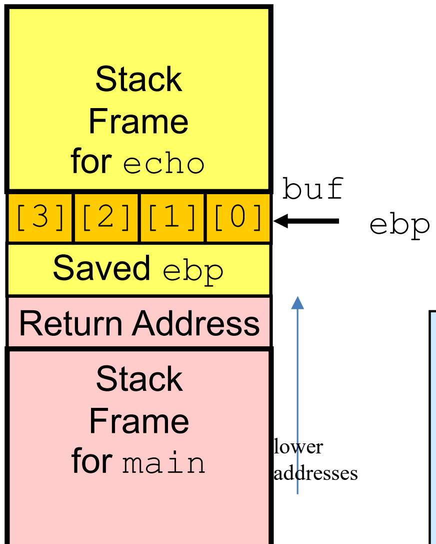
Assembly language

- Disassembly

```
    if (buff != NULL)
00DAE47C  cmp         dword ptr [buff],0
00DAE480  je          $LN13+2Bh (0DAE576h)
    {
        ptr = buff;
00DAE486  mov         eax,dword ptr [buff]
00DAE489  mov         dword ptr [ptr],eax
        while (len > 0)
00DAE48C  cmp         dword ptr [len],0
00DAE490  jbe         $LN13+2Bh (0DAE576h)
        {
            printf("  Processor mask: %lx \n", ptr->ProcessorMask);
00DAE496  mov         eax,dword ptr [ptr]
00DAE499  mov         ecx,dword ptr [eax]
00DAE49B  push        ecx
00DAE49C  push        offset string "  Processor mask: %lx \n" (0E3CE80h)
00DAE4A1  call        _printf (0DAAE86h)
00DAE4A6  add         esp,8
```

Assembly language

- Code security

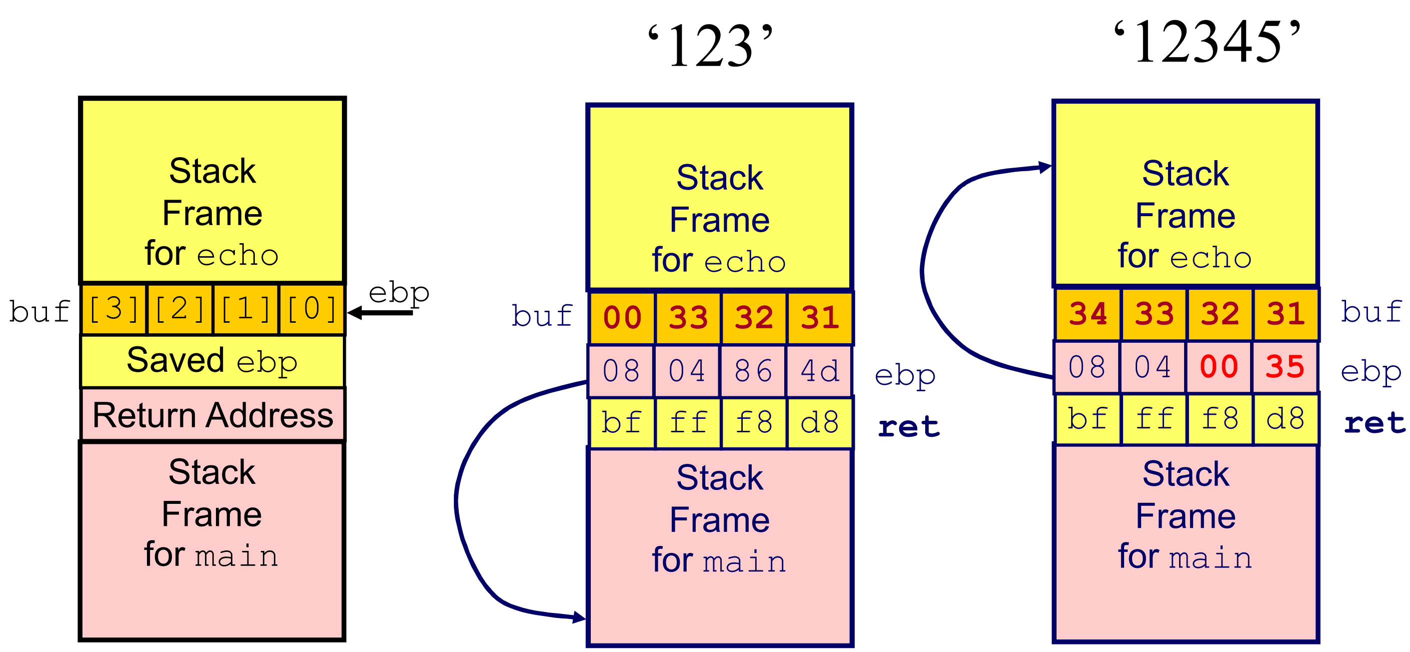



```
/* Echo Line */
void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}
```

```
echo:
    push ebp              # Save ebp on stack
    mov ebp,esp
    sub esp,4             # Allocate space on stack
    push ebx              # Save ebx
    lea ebx,-4(ebp)       # Compute buf as ebp-4
    push ebx              # Push buf on stack
    call gets             # Call gets
    . . .
    pop ebp
    ret
```

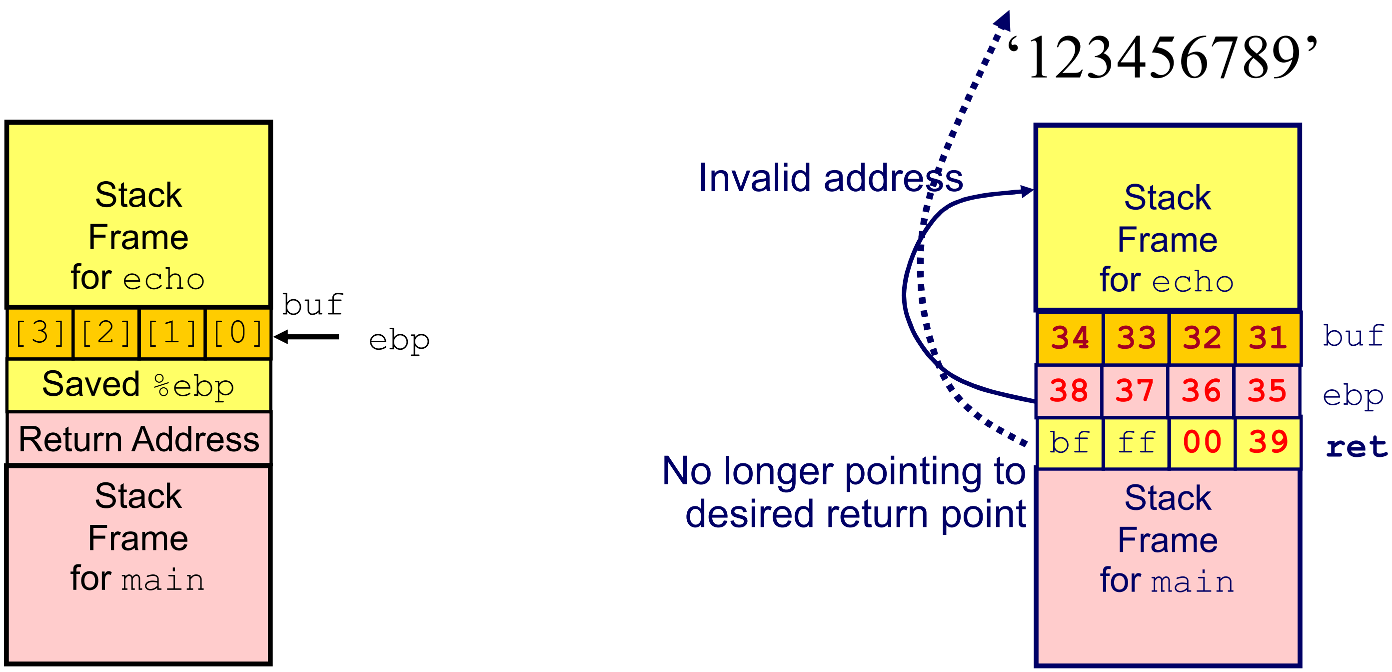
Assembly language

- Code security



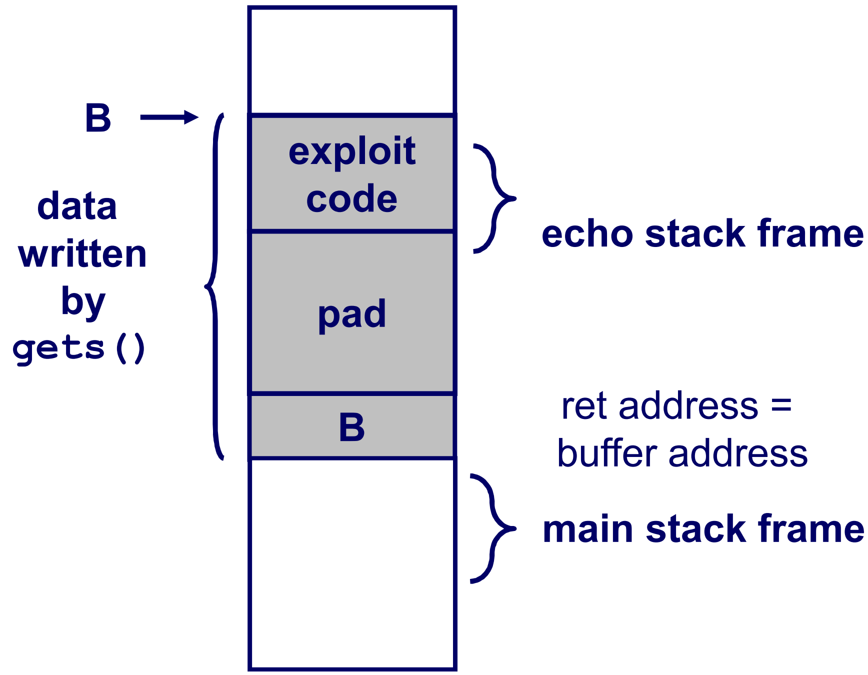
Assembly language

- Code security



Assembly language

- Code security



Assembly language

- Code security issues
  - Local memory buffers are filled with the exploit machine code
  - Return address is overridden by buffer address
  - On return from subroutine:
    - the malicious code will be executed in the context of the user application
    - or the application will be executed incorrectly

Assembly language

- Code security issues
  - Memory buffer overflow can allow remote code to be executed using the local permissions of the current application context
  - Example: Internet Worm

Assembly language

```
if cond then
    secv_instr_1
else
    secv_instr_2
secv_instr
```

```
    evaluate_cond
    jmp cond, label_if
    secv_instr_2
    jmp label_endif
label_if:
    secv_instr_1
label_endif:
    secv_instr
```

```
    evaluate_cond
    jmp !cond, label_else
    secv_instr_1
    jmp label_endif
label_else:
    secv_instr_2
label_endif:
    secv_instr
```

Assembly language

```c
int main()
{
   char * buff = "ABC";
   if (buff != NULL)
   {
      printf("Not null");
   }
   else
   {
      printf("Null");
   }
   return 0;
}
```

```
int main()
{
…
; char * buff = "ABC";
00C717AE  mov         dword ptr [buff],offset string "ABC" (0C76B30h)
; if (buff != NULL)
00C717B5  cmp         dword ptr [buff],0
00C717B9  je          main+3Ah (0C717CAh)
; {
; printf("Not null");
00C717BB  push        offset string "Not null" (0C76B34h)
00C717C0  call        _printf (0C71316h)
00C717C5  add         esp,4
; }
; else
00C717C8  jmp         main+47h (0C717D7h)
; {
; printf("Null");
00C717CA  push        offset string "Null" (0C76B40h)
00C717CF  call        _printf (0C71316h)
00C717D4  add         esp,4
; }
;    return 0;
…
; }
```

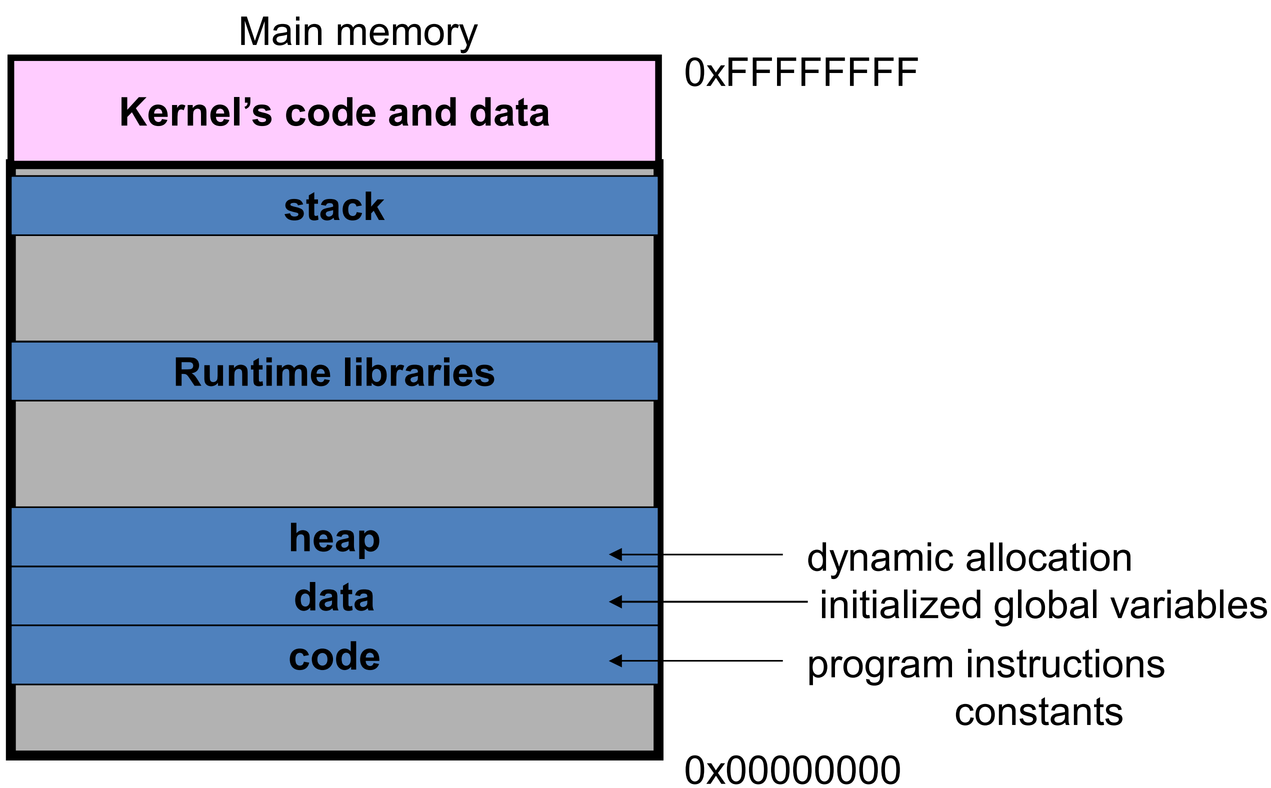
Assembly language

```
while( cond )
    secv_instr_1
secv_instr_2

etich_while:
    eval_cond
    jmp !cond, etich_end
    secv_instr_1
    jmp etich_while
etich_end:
    secv_instr_2
```

Assembly language

```
repeat
    secv_instr_1
until cond
secv_instr_2

for i = 1 to n
    secv_instr_1
secv_instr_2
```

Summary


Main memory
Kernel's code and data
stack
Runtime libraries
heap
data
code
0xFFFFFFFF
0x00000000
dynamic allocation
initialized global variables
program instructions
constants