Computer Architecture II
CSU34021

Syed Asad Alam

School of Computer Science and Statistics

September 26, 2021
Moore’s Law

- “The number of transistors incorporated in a chip will approximately double every 24 months.”\(^1\)
- “The complexity for minimum component costs has increased at a rate of roughly a factor of two per year”\(^2\)

\(^1\)Intel
\(^2\)SemiWiki
Joy’s Law

- Bill Joy → Co-founder and chief scientist at Sun Microsystems

\[ \text{MIPS} = 2^{Y-1984} \]

where MIPS = Millions of instructions per second

---

**Microprocessor Design Trends**  
IA32 – Intel’s 32-bit Architecture  
Procedures  
\( \times 64 \) – Intel’s 64-bit Architecture

**How is this Enabled?**

- Smaller VLSI feature sizes [1 micron (\( \mu \)) \( \ldots \) 7nm] and 3-D transistors
- Increased clock rate [1MHz \( \ldots \) 4GHz]
- Reduced vs complex instruction sets [RISC vs CISC]
- Faster memory access modes (eg burst accesses)
- Integrated on-chip MMUs, FPsUs, ...
- Pipelining
- Superscalar [multiple instructions/clock cycle]
- Multi-level on-chip instruction and data caches
- Streaming SIMD [single instruction multiple data] instruction extensions [MMX, SSEx]
- Hyper threading, multi-core and multiprocessor support
- Direct programming of graphics co-processor
- High speed point to point interconnect [Intel QuickPath, AMD HyperTransport]
- Solid state disks ...
What Defines an Architecture

- Modes of operation
- Address space and memory management
- Programmer visible registers
- Instruction set architecture
  - Size of instructions
  - Addressing modes
  - Instruction formats
  - Types of instructions and operations
  - Number and type of operands
  - Source of operands

Architecture Differentiation on ISA

- Architecture can be differentiated on the type of instructions a processor executes
- Two major types
  - Complex instruction set computers (CISCs)
  - Reduced instruction set computers (RISCs)
- Commercial processors?
  - Intel → CISC
  - ARM → RISC
RISC Characteristics

- Simple and a small number of instructions
- Load/store architectures → Explicitly load data from memory and store results
- The add instruction will look like:
  
  ```
  load   mem(B), reg(1)
  load   mem(C), reg(1)
  add    reg(1), reg(2), reg(3)
  store  reg(3), mem(A)
  ```

- Large code size
- Large number of generation purpose registers

CISC Characteristics

- Complex and a large number of instructions that access memory directly
- Small number of general purpose registers
- Complex instruction decoding hardware
- Load/Stores embedded in complex instructions
- Example: add mem(B), mem(C), mem(A)
- Internally CISC instructions are broken down into small micro-operations which are executed in a single cycle, sometimes even in parallel
- Intel has a RISC core to execute simple instructions in one clock cycle
IA32 – Intel’s 32-bit Architecture

Internal Microprocessor Architecture – 32-bit, x86 Processors

- Intel’s 80386 → First 32-bit processor
- Past processors
  - 80286 (16-bit, 16 MB of RAM)
  - 8086/8088 (16-bit, 1 MB of RAM)
  - 8085 (8-bit, 64 KB of RAM)
- IA32 still used today by current Intel CPUs Modern Intel CPUs have many additions to the original IA32 including MMX, SSE1, SSE2, SSE3, SSE4, SSE5, AVX, AVX2 and AVX512 [Streaming SIMD Extensions] and an extended 64 bit instruction set when operating in 64 bit mode [named IA-32e or IA-32e or x64]
- 32 bit CPU [performs 8, 16 and 32 bit integer + 32 and 64 bit floating point arithmetic]
Address Space

- Amount of memory space available to the processor
- Linear address of 4 GB
- P6 and later → 36-bit address bus, 64 GB of memory through extended physical addressing (only 4 GB of linear space available to a program)

Processor Registers

32-Bit General-Purpose Registers

<table>
<thead>
<tr>
<th>EAX</th>
<th>EBP</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBX</td>
<td>ESP</td>
</tr>
<tr>
<td>ECX</td>
<td>ESI</td>
</tr>
<tr>
<td>EDX</td>
<td>EDI</td>
</tr>
</tbody>
</table>

16-Bit Segment Registers

<table>
<thead>
<tr>
<th>EFLAGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
</tr>
<tr>
<td>SS</td>
</tr>
<tr>
<td>DS</td>
</tr>
<tr>
<td>ES</td>
</tr>
<tr>
<td>FS</td>
</tr>
<tr>
<td>GS</td>
</tr>
</tbody>
</table>
Registers – Flexibility of Usage

<table>
<thead>
<tr>
<th>32-Bit</th>
<th>16-Bit</th>
<th>8-Bit (High)</th>
<th>8-Bit (Low)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>AX</td>
<td>AH</td>
<td>AL</td>
</tr>
<tr>
<td>EBX</td>
<td>BX</td>
<td>BH</td>
<td>BL</td>
</tr>
<tr>
<td>ECX</td>
<td>CX</td>
<td>CH</td>
<td>CL</td>
</tr>
<tr>
<td>EDX</td>
<td>DX</td>
<td>DH</td>
<td>DL</td>
</tr>
</tbody>
</table>

Registers – Specialized Usage

- **EAX** (extended accumulator register) → Automatically used by multiplication and division instructions
- **ECX** → Loop counter
- **ESP** → Extended stack pointer register
- **ESI** and **DSI** → Extended source and destination index registers
- **EBP** → Extended base pointer or extended frame pointer register
**Instruction Pointer and EFLAGS Register**

- **EIP**
  - Extended instruction pointer or program counter (PC)
  - Contains the address of the next instruction to be executed
  - Needs to be manipulated for loop instructions

- **EFLAGS Register**
  - A register where individual bits have a different meaning
  - Either shows the status of the processor after a user operation or system level operation or is used to control the processor
  - Examples
    - Carry flag, parity flag, zero flag, sign flag → Status
    - Direction and interrupt enable → Control

**Instruction Size**

- Variable instruction size
- Requires multi-step fetch and decode
- High code density
Number of Operands and Operand Types

The general structure of an instruction

[label:] mnemonic [operands] [; comments]

- Number of operands → zero, one, two, three or four
- Three basic type of operands
  - Immediate
  - Register
  - Memory

Operand Types

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg8</td>
<td>8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL</td>
</tr>
<tr>
<td>reg16</td>
<td>16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP</td>
</tr>
<tr>
<td>reg32</td>
<td>32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP</td>
</tr>
<tr>
<td>reg</td>
<td>Any general-purpose register</td>
</tr>
<tr>
<td>sreg</td>
<td>16-bit segment register: CS, DS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>imm</td>
<td>8-, 16-, or 32-bit immediate value</td>
</tr>
<tr>
<td>imm8</td>
<td>8-bit immediate byte value</td>
</tr>
<tr>
<td>imm16</td>
<td>16-bit immediate word value</td>
</tr>
<tr>
<td>imm32</td>
<td>32-bit immediate doubleword value</td>
</tr>
<tr>
<td>reg/mem8</td>
<td>8-bit operand, which can be an 8-bit general register or memory byte</td>
</tr>
<tr>
<td>reg/mem16</td>
<td>16-bit operand, which can be a 16-bit general register or memory word</td>
</tr>
<tr>
<td>reg/mem32</td>
<td>32-bit operand, which can be a 32-bit general register or memory doubleword</td>
</tr>
<tr>
<td>mem</td>
<td>An 8-, 16-, or 32-bit memory operand</td>
</tr>
</tbody>
</table>
Instruction Format

- Two address (MASM)
  
  ```
  add eax, ebx → eax = eax + ebx
  ```

- Alternative GNU syntax

  ```
  addl %ebx, %eax → eax = eax + ebx
  ```

- Most common: two operands

  - register/register
  - register/immediate
  - register/memory
  - memory/register

- Memory/memory and memory/immediate are NOT allowed

Addressing Modes

- How to interpret the addresses/operands/data-in-operand-field

- A number of types

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>mov eax, n</td>
<td>eax = n</td>
</tr>
<tr>
<td>Register</td>
<td>mov eax, ebx</td>
<td>eax = ebx</td>
</tr>
<tr>
<td>Direct</td>
<td>mov eax, a</td>
<td>eax = a</td>
</tr>
<tr>
<td>Reg. indirect</td>
<td>mov eax, [ebx]</td>
<td>eax = [ebx]</td>
</tr>
<tr>
<td>Reg. relative</td>
<td>mov eax, [ebx+n]</td>
<td>eax = [ebx + n]</td>
</tr>
<tr>
<td>Base plus index</td>
<td>mov eax, [ebx+ecx]</td>
<td>eax = [ebx + ecx]</td>
</tr>
<tr>
<td>Base rel.+index</td>
<td>mov eax, [ebx+ecx+n]</td>
<td>eax = [ebx+ecx+n]</td>
</tr>
<tr>
<td>Scaled indexed</td>
<td>mov eax, [ebx*s+n]</td>
<td>eax = [ebx*s + n]</td>
</tr>
<tr>
<td>Scaled indexed</td>
<td>mov eax, [ebx+ecx*s+n]</td>
<td>eax = [ebx + ecx*s + n]</td>
</tr>
</tbody>
</table>

![Diagram of address calculation]
Addressing Modes – More Examples

.data
array byte 84h,32h,2dh,8bh,8Fh

.code
mov bx,0
mov ecx,2
mov al,array
mov al,[array+1]
mov al,[array+bx+2]
mov dl,[array+ecx+1]

.data
array byte 84h,32h,2dh,8bh,8Fh

.code
lea rsi,array
mov rcx,2
mov dh,[rsi]
mov al,[rsi+1]
mov ah,[rsi+rcx+2]
### Basic Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>move</td>
</tr>
<tr>
<td>xchg</td>
<td>exchange</td>
</tr>
<tr>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>sub</td>
<td>subtract</td>
</tr>
<tr>
<td>cdq</td>
<td>convert double to quad word</td>
</tr>
<tr>
<td>idiv</td>
<td>unsigned divide</td>
</tr>
<tr>
<td>imul</td>
<td>signed multiply</td>
</tr>
<tr>
<td>inc</td>
<td>increment by 1</td>
</tr>
<tr>
<td>dec</td>
<td>decrement by 1</td>
</tr>
<tr>
<td>neg</td>
<td>negate</td>
</tr>
<tr>
<td>cmp</td>
<td>compare</td>
</tr>
<tr>
<td>lea</td>
<td>load effective address</td>
</tr>
<tr>
<td>test</td>
<td>AND operands and set flags</td>
</tr>
<tr>
<td>and</td>
<td>and</td>
</tr>
<tr>
<td>or</td>
<td>or</td>
</tr>
<tr>
<td>xor</td>
<td>exclusive or</td>
</tr>
<tr>
<td>not</td>
<td>invert</td>
</tr>
<tr>
<td>push</td>
<td>push onto stack</td>
</tr>
<tr>
<td>pop</td>
<td>pop from stack</td>
</tr>
<tr>
<td>sar</td>
<td>shift arithmetic right</td>
</tr>
<tr>
<td>shr</td>
<td>shift logical left</td>
</tr>
<tr>
<td>jmp</td>
<td>shift logical right</td>
</tr>
<tr>
<td>j {e, ne, l, le, g, ge}</td>
<td>unconditional jump</td>
</tr>
<tr>
<td>j {b, be, a, ae}</td>
<td>unsigned jump</td>
</tr>
</tbody>
</table>

- Intel®64 and IA-32 Architectures Software Developer’s Manuals for details

### Assembly Language Instructions – Examples

```assembly
mov ax,00FFh
add ax,1
sub ax,1

; implicit 32 bit move from memory [eax is 32 bits]
mov eax, [ebp+8]

; implicit 8 bit move from memory [ah is 8 bits]
mov ah, [ebp+8]

dec DWORD PTR [ebp+8]; make explicitly 32 bit
dec WORD PTR [ebp+8]; make explicitly 16 bit
dec BYTE PTR [ebp+8]; make explicitly 8 bit
```
Assembly Language Instructions – Examples

```assembly
.data
num1 qword 0A1B2C3D4E5F6A7Bh
.code
label: mov rax, num1
inc rax
jmp label
end

mov [ebp+8], 123 ; not allowed

; Use two instructions with implicit 32-bit operands
mov eax, 123
mov [ebp+8], eax
```

- lea [load effective address] is a useful instruction for loading memory address

```assembly
.data
array byte 84 h, 32 h, 2 dh, 8 bh, 8 Fh
.code
lea eax, [ebx+ecx*4+16]; eax = ebx+ecx*4+16
lea rsi, array
end
```
Assembly Language Instructions – Examples

- Quickest way to clear a register?
  
  \[
  \text{mov eax, } 0 ; \text{ instruction requires more bytes} \\
  \text{xor eax, eax} ; \text{ exclusive OR with itself}
  \]

- Quickest way to test if a register is zero:
  
  \[
  \text{test eax, eax} ; \text{ AND eax with itself, set flags and} \\
  \text{je ...} ; \text{ jump if zero}
  \]

http://mark.masmcode.com/
Procedure Calling Conventions

- **Caller** → The function that calls another function
  - Passes argument in a certain way
  - Will have data in some of registers which are important (need to preserve)
  - Remove parameters

- **Callee** → The function that is called from another function
  - Receive arguments in a certain way
  - Use internal registers for own computations (need to save them before usage)
  - Use the stack for some local variables
  - Restore saved registers
  - De-allocate space for local variables
  - Return

- Important is the agreement between the caller and callee → Calling convention
Calling Convention

- Several IA32 procedure/function calling conventions
- Use Microsoft _cdecl calling convention so C/C++ and IA32 assembly language code can mixed

Registers
- Caller must save EAX, ECX and EDX, if it wants its contents to be preserved (they are volatile)
- Callee must save EBX, EDI and ESI, if it needs to use them
- Callee returns result in EAX

Parameters
- Caller pushes arguments (right-to-left) onto the stack
- Caller pops (removes) arguments from the stack
- Why are parameters pushed right-to-left?
  - C/C++ pushes parameters right-to-left so functions like printf(char *formats, ...) [which can accept an arbitrary numbers of parameters] can be handled more easily

Stack Frames

- Stack plays an important role in procedure calling
- For each procedure call, a stack frame or runtime stack is established
- ESP and EBP maintain the stack frame
- It consists of:
  - Passed arguments, if any, are pushed on the stack.
  - The subroutine is called, causing the subroutine return address to be pushed on the stack.
  - As the subroutine begins to execute, EBP is pushed on the stack.
  - EBP is set equal to ESP. From this point on, EBP acts as a base reference (frame pointer) for all of the sub-routine parameters.
  - If there are local variables, ESP is decremented to reserve space for the variables on the stack.
  - If any registers need to be saved, they are pushed on the stack.
  - Stack grows downwards
Stack Frames

IA32 Function Stack Frame

- stack frame after call to f(p0, p1, p2)
- stack grows down in memory [from highest address to lowest]
- parameters pushed right to left
- NB: stack always aligned on a 4 byte boundary [it's not possible to push a single byte]
- ebp used as a frame pointer: parameters and locals accessed relative to ebp [p0 @ ebp+8]

An Example

```c
int f (int p0, int p1, int p2) { // parameters
    int x, y; //local variables
    x = p0 + p1;
    ...
    return x + y; //result
}
```
An Example

`; f(1,2,3) 
`; passing arguments right to left
push 3
push 2
push 1

`; calling the function
call f

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

`; Establishing the stack frame
f: push ebp ; save ebp
mov ebp, esp ; ebp -> new stack frame
sub esp, 8 ; space allocation for local variables (x,y)
push ebx ; save non-volatile registers used by function

`; Function body
`; x = p0+p1
mov eax, [ebp+8] ; access first parameter
add eax, [ebp+12] ; eax = p0+p1
mov [ebp-4], eax ; x = p0+p1

`; return x+y
mov eax,[ebp-4] ; eax = x
add eax, [ebp-8] ; eax = x+y

`; Returning
pop ebx ; restore saved registers
mov esp, ebp ; restore esp
pop ebp ; restore ebp
ret 0 ; return from function
Next video → Using Visual Studio with an example project containing a mix of C/C++ and Assembly
Introduction

- Extension of IA32
- Originally developed by AMD
- IA32 registers extended to 64 bits
- Eight additional registers (r8 – r15)
- 64, 32, 16 and 8 bit arithmetic
- Same instruction set as IA32
- 64 bit virtual and physical address space (theoretically anyway, 48-bit actual address bus)

\[ 2^{64} = 16 \text{ Exabytes} = 16 \times 10^{18} \text{ bytes} \]
Registers

- 16, 64-bit general purpose registers
- 8, 80-bit floating point registers
- A 64-bit status flag register named RFLAGS (only the lower 32-bits are used) and 64-bit instruction pointer named RIP
- 8, 64-bit MMX registers
- 16, 128-bit XMM registers

Function Calling and Conventions

- Various conventions
  - Calling convention for Windows
    - First 4 parameters $\rightarrow$ RCX, RDX, R8, R9
    - Rest pushed onto the stack in reverse order
    - Caller preserved registers $\rightarrow$ RAX, RCX, RDX, R8, R9, R10, R11
    - Callee preserved registers $\rightarrow$ RBX, RBP, RDI, RSI, R12, R14, R14, R15
    - Stack aligned on the 16-byte boundary
    - Caller must allocate 32-bytes of shadow space
    - No need to create a stack frame with RBP (may be created if more than 32-byte space is needed)
    - Subtract 8 from RSP to align to 16 bytes, if a non-leaf function
  - Calling convention for GCC
    - First six parameters $\rightarrow$ RDI, RSI, RDX, RCX, R8, R9
    - Rest pushed onto the stack in reverse order
    - Caller preserved registers $\rightarrow$ RAX, RCX, RDX, RDI, R8, R9, R10, R11
    - Callee preserved registers $\rightarrow$ RBX, RBP, R12, R14, R14, R15
    - Stack aligned on the 16-byte boundary
    - No shadow space
    - Stack frame may be created if stack needed for local storage
    - Subtract 8 from RSP to align to 16 bytes, if a non-leaf function
Calling a Leaf Function

```c
_int64 fib(_int64 n)
// _int64 is Microsoft specific
// may use long long
{
    // here INT64 can be defined as:
    // #define INT64 long long
    INT64 fi, fj, t;

    if(n<=1)
        return n

    fi = 0;
    fj = 1;

    while (n>1){
        t = fj;
        fj = fi+fj;
        fi = t;
        n--;
    }
    return fj;
}
```
### Calling a Leaf Function

```
fib_x64:  mov rax, rcx ; rax = n
cmp rax, 1 ; if(n<=1)
jle fib_x64_1 ; return n
xor rdx, rdx ; fi = 0
mov rax, 1 ; fj = 1

fib_x64_0: cmp rcx, 1 ; while (n > 1)
jle fib_x64_1 ;
mov r10, rax ; t = fj
add rax, rdx ; fj = fi + fj
mov rdx, r10 ; fi = t
dec rcx ; n--
jmp fib_x64_0 ;

fib_x64_1: ret ; return
```

---

### Calling a Non-Leaf Function with Arbitrary Arguments

- A function that calls another function with arbitrary/variable arguments
- Allocate stack space according to: $\max(32, 8 \times n)$ where $n =$ number of arguments
- Use the same stack space for multiple calls to such a function
- Makes it easier for compiler to determine how much stack space is required
Calling a Non-Leaf Function with Arbitrary Arguments

```c
function f(...) {
    ...
    printf(5 parameters)
    ...
    printf(6 parameters)
    ...
    printf(4 parameters)
    ...
}
```

```c
_int64 print_nums(_int64 a, _int64 b) {
    printf("a = %I64d b = %I64d a+b = %I64d\n", a, b, a + b);
    return a + b;
}
```

- Uses `%I64d` to format a 64 bit integer
- Parameters `a` and `b` passed to `print_nums` in `RCX` and `RDX` respectively
- Need to call external `printf(...)` function with 4 parameters
  - `rcx` [address of format string]
  - `rdx` [a]
  - `r8` [b]
  - `r9` [a+b]

48 bytes of stack space allocated at start
32 bytes used as the shadow space
### Calling a Non-Leaf Function

```
fxp2 byte 'a = %I64d b = %I64d a+b = %I64d', 0AH, 00H ; ASCII format string

xp2: sub rsp, 8 ; align the stack to 16-byte boundary
    push rbx ; save rbx (rbx used to remember a+b across call to printf)
    sub rsp, 32 ; allocate shadow space
    lea r9, [rcx+rdx] ; printf parameter 4 in r9 [a+b] - a passed in rcx, b in rdx
    mov r8, rdx ; printf parameter 3 in r8 [b]
    mov rdx, rcx ; printf parameter 2 in rdx [a]
    lea rcx, fxp2 ; printf parameter 1 in rcx [&fxp2]
    mov rbx, r9 ; save r9 [a+b] in rbx so preserved across call to printf
    call printf ; call printf
    mov rax, [rsp+64] ; function result in rax = rbx {a+b}
    add rsp, 32 ; deallocate shadow space
    pop rbx ; restore rbx
    add rsp, 8 ; restore the stack ptr to ret addr
    ret ; return
```
You are now able to:
- Write simple IA32 assembly language functions
- Write simple x64 assembly language functions
- Call IA32/x64 assembly language functions from C/C++