Contents

IA32 and x64

Microprocessor Design Trends

- Joy's Law

- MIPS = 2^(year-1984)

Some of the ideas and tecniques used...

- Smaller VSLI feature sizes [1 micron -> 10nm]
- Increased clock rate [1MHz -> 4GHz]
- Reduced *vs* complex instruction sets [RISC *vs* CISC]
- Burst memory access
- Integrated on-chip MMUs [memory management unit], FPU, ...
- Pipelining
- Superscalar [multiple instructions/clock cycle]
- Multi-level on-chip instruction and data caches
- Streamed SIMD [single instruction multiple data] instruction extensions [MMX, SSEx]
  - Allows processor to execute one instruction on lots of data
- Multiprocessor support
  - Connect multiple processors together
- Hyper threading and multi core
- Direct programming of graphics co-processor
- High speed point interconnect [Intel QuickPath, AMD HyperTransport]
- Solid state Dicks

IA32 [Intel Architecture 32 bit]

- First released in 1985 with the 80386 microprocessor
- Still used today by current Intel CPUs
- Modern Intel CPUs have many additions to the original IA32 including MMX, SSE1, SSE2, SSE3, SSE4 and SSE5 [streaming SIMD extensions] *and* even an extended 64 bit instruction set when operating in 64 bit mode
- 32 bit CPU
- 32 bit virtual and physical address space [4GB]
- Each instruction a multiple of bytes in length

Registers

General purpose registers

- eax - accumulator
- ebx

- ecx
- edx

Normally used as memory address registers

- esi
- edi
- ebp - frame pointer
- esp - stack pointer
- eflags - flags [status register]
- eip - instruction pointer [pc]
- "e" in eax = extended = 32bits
- Possible to access 8 and 16 bit parts of eax, ebx, ecx and edx using alternate register names

Instruction Format

- Two address [will use Microsoft assembly language syntac used by VC++, MASM]

```
add eax, ebx    ; eax = eax + ebx
```

- Alternative gnu syntax

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

- Two operands normally [right to left] > register/register > register/immediate > register/memory > memory/register
- memory/memory and memory/immediat **NOT** allowed

Supported Addressing Modes

addressing mode	example	
immediate	mov eax n	eax=n
register	mov eax, ebx	eax=ebx

addressing mode	example	
direct/absolute	mov eax, [a]	eax=[contents of memory at a]
indexed	mov eax, [ebx]	eax=[ebx]
indexed	mov eax, [ebx+n]	eax=[ebx+n]
scaled indexed	mov eax, [ebx*s+n]	eax=[ebx*s+n]
scaled indexed	mov eax, [ebx+ecx]	eax=[ebx+ecx]
scaled indexed	mov eax, [ebx+ecx*s+n]	eax=[ebx+ecx*s+n]

- Address computed as the sum of a register, a scaled register, and a 1, 2 or 4 byte signed constant n; can use most registers
- Scaling constants `s` can be 1 2, 4 and 8

Assembly Language Tips

- Size of the operation can often be determined implicitly by assembler, but when unable to do so, size needs to be specified explicitly

```
mov eax [ebp+8]     ; implicitly 32 bit [as eax is 32 bits]
mov ah, [ebp+8]     ; implicitly 8 bit [as ah is 8 bits]
dec [ebp+8]     ; assembler unable to determine operand size
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
```

- Memory/immedate operations **not** allowed
- Scaled indxed used to index arrays of 1, 2, 4 or 8 byte values [example later in notes]

```
int a[100];     // array of 4 byte values
double b[100];     // array of 8 byte values
```

- lea [load effective address] is useful for performing simple arithmetic

```
lea eax, [ebx+ecx\*4+16]     ; eax = ebx+ecx\*4+16
```

- It's quicker to clear a register by XORing it with itself rather than moving 0 into it, as 0 has to be encoded in the instruction meaning `mov` is a longer instruction.
- Similarly, the fastest way to check if a register is 0 is to `test` it against itself and jump if equal.

Useful Operations

```
mov  ; move
xchg ; exchange
add  ; add
sub  ; subtract
imul ; signed mul
mul  ; unsigned mul
inc  ; increment
dec  ; decrement
neg  ; negate
cmp  ; compare
lea  ; load effective address
test ; AND operands and set flags

and ; and
or  ; or
xor ; exclusive or
not ; not

push ; push onto stack
pop  ; pop from stack

sar ; shift arithmetic right
shl ; shift logical left
shr ; shift logical right

jmp                                   ; unconditional jump
j{e, ne, le, g, ge} signed            ; signed jump
; equal, not equal, less than or equal, greater, greater than or equal
j{b, be, a, ae} unsigned (below, above) ; unsigned jump
; e.g. jle

call ; call subroutine
ret  ; return from subroutine
```

IA32 Calling Conventions

- Several IA32 procedure/function calling conventions
- use Microsoft `_cdecl` calling convention so C/C++ and IA32 assembly language code can be mixxed
  - Function result returned in `eax`
  - `eax`, `ecx` and `edx` considered volatile and are **NOT** preserved across function calls

  - Called removes parameters
- 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 number of parameters) can be handled more easily since the first parameter is always stored at `[ebp+8]` irrespective of how many parameters were pushed

Function Calling

- The steps to call a function are follows:
  1. Pass parameters (push onto stack)
  2. Enter a new function (push return address and jump to first instruction of the function)
  3. Save frame pointer (`push ebp`)
  4. Intialise frame pointer (`ebp=esp`)
  5. Allocate space for local variables (on stack by decrementing esp)
  6. Save non-volatile registers (on stack)
  7. Execute function body
  8. Restore registers (from stack)
  9. De-allocate local variables (increment `esp` or `esp=ebp`)
  10. Restore ebp
  11. Return to calling function (pop return from stack)
  12. Clear parameters from stack (increment `esp`)

Stack Use

- Parameters are pushed right-to-left, so a function call `f(p0, p1, p2)` would have `p2` at the highest address and `p0` at the lowest
- The stack is a full descending stack
- By using `ebp` and `esp`, we can access the parameters pushed onto the stack (relative to `ebp`) and allocate space for the local variables (by moving `esp`)
  - `ebp` is set by the caller
  - `esp` can be changed by the function
- The stack is always 4 byte aligned

pointers	top of stack	
	p2	pushed param
	p1	pushed param

pointers	top of stack	
	p0	pushed param
	ret addr	
ebp ->	saved ebp	
	local var 0	local variable
esp ->	local var 1	local variable

Start of Function

pointers	top of stack	
	p2	pushed param
	p1	pushed param
	p0	pushed param
	ret addr	
ebp ->	saved ebp	
	local var 0	local variable
	local var 1	local variable
	ebx	saved register
esp ->	esi	saved register

End of Function

Accessing Parameters and Local Variables

- `ebp` used as a frame pointer; parameters and local variables accessed at offsets from `ebp`
- Can avoid using a frame pointer (normally for speed) by accessing parameters and locals variables relative to the stack pointer, but more difficult because of the stack pointer can change during execution (BUT easy for a compiler to track)
- Parameters accessed with `+ve` offsets from `ebp` (see stack frame diagram)
    - `p0` at `[ebp+8]`
    - `p1` at `[ebp+12]`

- Local variables accessed with `-ve` offsets from `ebp` (see stack frame diagram)
  - `local variable 0` at `[ebp-4]`
  - `local variable 1` at `[ebp-8]`

Simple Function Example

```c
int f(int p0, int p1, int p2) {
    int x, y; // Local Variables
    x = p0 + p1;
    return x+y; // Result
```

- A call `f(p0, p1, p2)` matches stack frame diagram on previous slide
- 3 parameters `p0`, `p1`, and `p2`
- 2 local variables `x` and `y`

```
; f(1, 2, 3)
push 3 ; push parameters from right to left
push 2
push 1
call f ; call the function
add esp, 12 ; pop the parameters
```

Now we write the function itself

```
; within the function, p0 = [ebp + 8],
;                      p1 = [ebp + 12],
;                      p2 = [ebp + 16]
; x = [ebp - 4], y = [ebp - 8]
mov eax, [ebp + 8]  ; eax = p0
add eax, [ebp + 12] ; eax = p0 + p1
mov [ebp -4], eax   ; x = p0 + p1

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

Reference

See Intel 64 and IA-32 Architectures Software Developer's Manual 2A, 2B, 2C

Function Entry

- Need instructions on function entry to save `ebp` [old frame pointer]
- Initialise `ebp` [new frame pointer]
- Allocate space for local variables on stack
- Push any non volatile registers used by function onto stack

```
push ebp      ; save ebp
mov ebp, esp ; ebp -> new stack frame
sub esp, 8    ; allocate space for locals [x and y]
push ebx     ; save any non volatile registers used by the function

<function body>
```

Function Exit

- Need instrucions to unwind stack frame at function exit

```
...
pop ebx       ; Restored any saved registers
mov esp, ebp ; Restore esp
pop ebp       ; Restore previous ebp
ret 0         ; Return from function
```

- `ret` pops returns address from stack
- Adds integer parameter to `esp`
- If integer parameter not specified, defaults to 0

Simple Array Access

```
int a[100];

main(...) {
    a[1] = a[2]+3;
}
```

```
mov eax, [a+8]
add eax, 3
mov [a+4], eax
```

```
int p() {
    int i = ...;     // Local variable i sored at [ebp-4]
```

```
    int j = ...;     // Local variable j stored at [ebp-8]
    ...
    a[i] = a[j]+3;  // Variable indices
}
```

```
mov eax, [ebp-8]    ; eax = j
mov eax, [a+eax*4] ; eax = a[j]
add eax, 3          ; eax = a[j]+3
mov ecx, [ebp-4]    ; exc = i
```

Putting it Together

- Mixing C/C++ and IA32 Assembly Language
- VC++ main(…) calls an assembly language versions of `fib(n)` to calculate `n^th` Fibonacci number
- Create a VC++ Win32 console application
- Right click on project and select "Build Customications" and tick masm
- Add fib32.h and fib32.asm files to project
- Right click on fib32.asm and check [Genneral][Item Type] == Microsoft Macro Assembler
- Check project [Properties][Debugger][Debugger Type] == Mixed

x64

- Extension of IA32
- Originally designed by AMD
- IA32 registers extended to 64 bits rax … rsp, rflags and rip
- 8 additional registers r8 … r15
- 64, 32, 16 and 8 bit arithmetic
- *Same* instruction set
- 64 bit virtual and physical address spaces
    - $2^{64} = 16$ Exabytes$= 16x10^8$ bytes

Function Calling

- Use Microsoft calling convention
- First 4 parameters passed in `rcx`, `rdx`, `r8`, and `r9` respectively
- Additional parameters are passed on the stack [right to left]
- Stack always aligned on an 8 byte boundary

- Caller *must* allocate *shadow space* on stack [conceptually for the parameters passed in `rcx`, `rdx`, `r8` and `r9`]
- `rax`, `rcx`, `rdx`, `r8`, `r9`, `r10` and `r11` are volatile
- Having so many registers often means
  1. Can use registers for local variables
  2. No need to use the frame pointer
  3. No need to save/restore registers
- It is the responsibility of the caller to allocate 32 (`4 x 8 bytes = 4 x 64 bits`) bytes of shadow space on the stack before calling a function [regardless of the actual number of parameters used] and to deallocate the shadow space afterwards
- Called functions can use its shadow space to spill `rcx`, `rdx`, `r8` and `r9` [spill=save memory]
- Called functions, however, can use its shadow space for any purpose whatsoever and consequently may write to and read from it as it sees fit [which is why it needs to be allocated]
- 32 bytes of shadow space must be made available to all functions, even those with fewer than four parameters
- Callee has 5 parameters, so paramter 5 passed on stack
- Parameters 1 to 4 passed in `rcx`, `rdx`, `r8` and `r9`
- **NB** allocate shadow space
- Old frame pointer saved and new frame pointer initialised [`rbp`]
- Space allocated for local variables on the stack if needed

Simple Function

```
_int64 fib(_int64 n) {
    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;
}
```

- use `_int64` bit integers [Microsoft specific] or `long long`
- Paramters `n` passed to function in `rcx`
- Leaf function [doesn't call any other function]

```
fib_x64:   mov rax, rcx
           cmp rax, 1
           jle fib_x64_1
           xor rdx, rdx    ; fi = 0
           mov rax, 1      ; fj = 1
fib_x64_0: cmp rcx, 1
           jle fib_x64_1
           mov r10, raw    ; t=fj
           add rax, rdx    ; fj+=fi
           mov rdx, r10    ; fi=t
           dec rcx
           jmp fib_x64_0
fib_x64_1: ret
```

- **NB** only use volatile registers

```
_int64 xp2(_int64 a, _int64 b) {
    printf("a=%I64d b=%I64d, a+b=%I64d\n", a, b, a+b);
    return a + b;
}
```

```
fxp2 db   'a=%I64d b=%I64d, a+b=%I64d', 0AH, 00H  ; ASCII format string
                                                  ; (carriage return), (null)
```

```
xp2: push rbx
     sub rsp, 32        ; allocate shadow space
     lea r9, [rcx+rdx]  ; a+b
     mov r8, rdx        ; b
     mov rdx, rcx       ; a
     lea rcx, fxp2      ; printf parameters 1 in rcx
   mov rbx, r9      ; save r9 in rbx so preserved across to call printf
     call printf
     mov rax, rbx       ; return val
     add rsp, 32        ; deallocate shadow space
     pop rbx
     ret
```