Microprocessor Design Trends

• Joy's Law [Bill Joy of BSD4.x and Sun fame]

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

• Millions of instructions per second [MIPS] executed by a single chip microprocessor

• More realistic rate is a doubling of MIPS every 18 months [or a quadrupling every 3 years]

• What ideas and techniques in new microprocessor designs have contributed to this continued rate of improvement?
Some of the ideas and techniques used...

- smaller VLSI feature sizes \([1 \text{ micron (μ)} \rightarrow 10\text{nm}]\)
- increased clock rate \([1\text{MHz} \rightarrow 4\text{GHz}]\)
- reduced vs complex instruction sets \([\text{RISC vs CISC}]\)
- burst memory accesses
- integrated on-chip MMUs, FPUs, ...
- pipelining
- superscalar \([\text{multiple instructions/clock cycle}]\)
- multi-level on-chip instruction and data caches
- streaming SIMD \([\text{single instruction multiple data}]\) instruction extensions \([\text{MMX, SSEx}]\)
- multiprocessor support
- hyper threading and multi core
- direct programming of graphics co-processor
- high speed point to point interconnect \([\text{Intel QuickPath, AMD HyperTransport}]\)
- solid state disks
IA32 [Intel Architecture 32 bit]

- IA32 first released in 1985 with the 80386 microprocessor
- IA32 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 [named IA-32e or IA-32e or x64]
- 32 bit CPU [performs 8, 16 and 32 bit arithmetic]
- 32 bit virtual and physical address space $2^{32}$ bytes [4GB]
- Each instruction a multiple of bytes in length [1 to 17+]
Registers [not as many as a typical RISC]

- eax: accumulator
- ebx
- ecx
- edx
- esi
- edi
- ebp
- esp

- eflags: flags [status register]
- eip: instruction pointer [pc]

NB: floating point and SSE registers etc. not shown
Registers...

- "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 syntax used by VC++, MASM]

  \[\text{add} \quad \text{eax}, \text{ebx} \quad ; \quad \text{eax} = \text{eax} + \text{ebx} \quad [\text{right to left}]\]

- alternative gnu syntax

  \[\text{addl} \quad \%\text{ebx}, \%\text{eax} \quad ; \quad \text{eax} = \text{eax} + \text{ebx} \quad [\text{left to right}]\]

- two operands normally

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

- memory/memory and memory/immediate NOT allowed
**Supported Addressing Modes**

<table>
<thead>
<tr>
<th>addressing mode</th>
<th>example</th>
<th>result</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/absolute</td>
<td>mov eax, ([a])</td>
<td>( eax = [a] )</td>
</tr>
<tr>
<td>indexed</td>
<td>mov eax, ([ebx]]</td>
<td>( eax = [ebx] )</td>
</tr>
<tr>
<td>indexed</td>
<td>mov eax, ([ebx+n])</td>
<td>( eax = [ebx + 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])</td>
<td>( eax = [ebx + ecx] )</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>

- 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
- scaled indexed addressing used to index into arrays
- scaling constant \( s \) can be 1, 2, 4 or 8;
Assembly Language Tips

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

\[
\begin{align*}
\text{mov} & \quad \text{eax}, \ [\text{ebp}+8] \quad \text{; implicitly 32 bit [as eax is 32 bits]} \\
\text{mov} & \quad \text{ah}, \ [\text{ebp}+8] \quad \text{; implicitly 8 bit [as ah is 8 bits]} \\
\text{dec} & \quad \text{[ebp}+8\text{]} \quad \text{; decrement memory location [ebp+8] by 1} \\
\text{dec} & \quad \text{DWORD PTR [ebp}+8\text{]} \quad \text{; make explicitly 32 bit} \\
\text{dec} & \quad \text{WORD PTR [ebp}+8\text{]} \quad \text{; make explicitly 16 bit} \\
\text{dec} & \quad \text{BYTE PTR [ebp}+8\text{]} \quad \text{; make explicitly 8 bit}
\end{align*}
\]

NB: unusual assembly language syntax
Assembly Language Tips...

- memory/immediate operations NOT allowed

\[ \text{mov } [\text{ebp}+8], 123 \] ; NOT allowed and operation size ALSO unknown
\[ \text{mov eax, 123} \] ; use 2 instructions instead...
\[ \text{mov } [\text{ebp}+8], \text{eax} \] ; explicitly 32 bits

- \text{lea [load effective address]} is useful for performing simple arithmetic

\[ \text{lea eax, } [\text{ebx}+\text{ecx} \times 4+16] \] ; \text{eax = ebx+ecx} \times 4+16
## 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>imul</td>
<td>signed multiply</td>
</tr>
<tr>
<td>mul</td>
<td>unsigned 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>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>shl</td>
<td>shift logical left</td>
</tr>
<tr>
<td>shr</td>
<td>shift logical right</td>
</tr>
<tr>
<td>jmp</td>
<td>unconditional jump</td>
</tr>
<tr>
<td>j {e, ne, l, le, g, ge}</td>
<td>signed jump</td>
</tr>
<tr>
<td>j {b, be, a, ae}</td>
<td>unsigned jump</td>
</tr>
<tr>
<td>call</td>
<td>call subroutine</td>
</tr>
<tr>
<td>ret</td>
<td>return from subroutine</td>
</tr>
</tbody>
</table>

- should be enough instructions to complete tutorials
- Google *Intel® 64 and IA-32 Architectures Software Developer’s Manual 2A, 2B, 2C* for details
Assembly Language Tips...

- quickest way to clear a register?
  
xor eax, eax ; exclusive OR with itself

mov eax, 0 ; instruction occupies more bytes and…
            ; probably takes longer to execute

- quickest way to test if a register is zero?

  test eax, eax ; AND eax with itself, set flags and…
  je ... ; jump if zero
Function Calling

reminder of the steps normally carried out during a function/procedure call and return

• pass parameters [evaluate and push on stack]
• enter new function [push return address and jump to first instruction of function]
• allocate space for local variables [on stack by decrementing esp]
• save registers [on stack]

<function body>

• restore saved registers [from stack]
• de-allocate space for local variables [increment esp]
• return to calling function [pop return address from stack]
• remove parameters [increment esp]
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 [eg \( p0 @ ebp+8 \)]
IA32 Calling Conventions

• several IA32 procedure/function calling conventions

• use Microsoft _cdecl calling convention [as per previous diagram] so C/C++ and IA32 assembly language code can mixed

  function result returned in eax

  eax, ecx and edx considered volatile and are NOT preserved across function calls

  caller 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 numbers of parameters] can be handled more easily since the first parameter is always stored at [ebp+8] irrespective of how many parameters are pushed
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 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]
  ...

Consider the IA32 Code for a Simple Function

```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`
IA32 Code to Call Simple Function...

• parameters $p_0$, $p_1$ and $p_2$ pushed onto stack by caller right to left

$f(1, 2, 3)$

push 3 ; push immediate values...
push 2 ; right to left
push 1 ;
call f ; call subroutine f
add esp, 12 ; on return, add 12 to esp to remove parameters

• inside function parameters accessed relative to ebp [see stack frame diagram]

$p_0$ stored at $[ebp+8]$
$p_1$ stored at $[ebp+12]$
$p_2$ stored at $[ebp+16]$
Accessing Local Variables of Simple Function...

- space allocated on stack for local variables x and y (see diagram)

  \[x \text{ stored at } [ebp-4]\]
  \[y \text{ stored at } [ebp-8]\]

- \(x = p0 + p1\)
  
  \[
  \begin{align*}
  &\text{mov eax, [ebp+8]} \quad ; \text{eax = p0} \\
  &\text{add eax, [ebp+12]} \quad ; \text{eax = p0 + p1} \\
  &\text{mov [ebp-4], eax} \quad ; x = p0 + p1 \\
  \end{align*}
  \]

- return \(x + y\);
  
  \[
  \begin{align*}
  &\text{mov eax, [ebp-4]} \quad ; \text{eax = x} \\
  &\text{add eax, [ebp-8]} \quad ; \text{eax = x + y} \\
  \end{align*}
  \]

NB: result returned in eax
Function Entry

- need instructions on function entry to save ebp [old frame pointer]
- initialize 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 function

<function body>      ; function code
```

NB: _cdecl convention means there is NO need to save eax, ecx and edx
Function Exit

- need instructions to unwind stack frame at function exit

```asm
...  
  pop  ebx          ; restore any saved registers
  mov  esp, ebp     ; restore esp
  pop  ebp          ; restore previous ebp
  ret  0            ; return from function
```

- ret pops return address from stack and...
- adds integer parameter to esp [used to remove parameters from stack]
- if integer parameter not specified, defaults to 0

- NB: _cdecl convention - caller removes parameters from stack

- NB: make sure you know why a stack frame needs to be created for each function call
IA32 Code for Simple Array Accesses

```c
int a[100]; // global array of int

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

NB: int is 4 bytes

NB: a[0] store at address a, a[1] at a+4, a[2] at a+8, a[n] at a+n*4

```assembly
mov    eax, [a+8]; // eax  = a[2]
add    eax, 3      // eax  = a[2] + 3
mov    [a+4], eax  // a[1] = a[2] + 3
```
int p() {
    int i = ...;       // local variable i stored at 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]  // ecx = i
    mov   [a+ecx*4], eax // a[i] = a[j]+3
Putting it Together - Mixing C/C++ and IA32 Assembly Language

• Example using Visual Studio 2010/2013/2015, VC++ and MASM

• VC++ main(...) calls an assembly language versions of fib(n) to calculate the n^{th} Fibonacci number

• create a VC++ Win32 console application
• right click on project and select "Build Customizations..." and tick masm
• add fib32.h and fib32.asm files to project [file can be edited within Visual Studio once included, but there doesn’t appears to be a way to create an .asm file from within Visual Studio]
• right click on fib32.asm and check [General][Item Type] == Microsoft Macro Assembler
• check project [Properties][Debugger][Debugger Type] == Mixed

• how to look at code generated by VC++ compiler??
right click on file name [Properties] [C/C++] [Output Files] [Assembler Output] and select Assembly, Machine Code and Source [generates listing file with .cod extension]
• NB: code generated in Debug mode will be different from Release mode
Putting it together...

fib32.h

- declare fib_IA32a(int) and fib_IA32b(int) as external C functions so they can be called from a C/C++ program

```c
extern "C" int g;         // external global int
extern "C" int _cdecl fib_IA32a(int); // external function
```

- extern "C" because C++ function names have extra characters which encode their result and parameter types

fib32.asm

- fib_IA32a(int) – *mechanical* code generation simulating Debug mode
- fib_IA32b(int) – *optimized* code generation simulating Release mode
- NB: MASM specific directives at start of file
- NB: .data and .code sections
- NB: public
Mixing C/C++ and IA32 Assembly Language...

IA32codegen.cpp [main]

- #include fib32.h
- call fib_IA32a(n) and fib_IA32b(n) like any other C/C++ function
- file also contains
  1) a C++ version of fib(n) and...
  2) a version of fib(n) that mixes C/C++ and IA32 assembly language using the IA32 inline assembler supported by the VC++ compiler
- call ALL versions of fib(n) for n = 1 to 20
- Visual Studio automatically compiles IA32codegen.cpp, assembles fib32.asm and links them to produce an executable which can then be run
- **Warning:** Visual Studio on SCSS machines (eg ICT Huts) has problems with source files stored on a Network drive