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 improvements made over the last 40 years ...

- smaller VLSI feature sizes [1 micron (μ) ... 7nm]
- increased clock rate [1MHz ... 4GHz]
- reduced vs complex instruction sets [RISC vs CISC]
- faster memory access modes (eg burst accesses)
- integrated on-chip MMUs, FPUs, ...
- 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
- ...

IA32 AND X64
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, 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]
- 32 bit virtual and physical address space $2^{32}$ bytes [4GB]
- each instruction a multiple of bytes in length [from 1 to 17+]
Registers [far fewer than a typical RISC]

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

- 32

- Accumulator
- Frame pointer
- Stack pointer

- General purpose registers
  - Normally used as memory address registers

- Flags [status register]
- Instruction pointer [pc]

NB: Floating point and SSE registers, ... not shown
Registers...

- "e" in eax = extended = 32 bits

- possible to access 8 and 16 bit parts of eax, ebx, ecx and edx using alternate register names ah, al and ax
Instruction Format

- two address [will use Microsoft assembly language syntax used by VC++, MASM]
  
  ```
  add     eax, ebx ; eax = eax + ebx [right to left]
  ```

- alternative gnu syntax
  
  ```
  addl   %ebx, %eax ; eax = eax + ebx [left to right]
  ```

- two operands normally
  
  - register/register
  - register/immediate
  - register/memory
  - memory/register

- memory/memory and memory/immediate are NOT allowed
## Supported Addressing Modes

<table>
<thead>
<tr>
<th>addressing mode</th>
<th>example</th>
<th>[a] = contents of memory address [a]</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
### IA32 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 quadword</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>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

IA32 Assembly Language examples

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

```assembly
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] ; decrement memory location [ebp+8] by 1
                 ; assembler unable to determine operand size
                 ; is it an 8, 16 or 32 bit value??
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
```

NB: unusual assembly language syntax
IA32 Assembly Language examples ...

- memory/immediate operations NOT allowed

```
mov [ebp+8], 123 ; NOT allowed and operation size ALSO unknown
mov eax, 123 ; use 2 instructions instead...
mov [ebp+8], eax ; implicitly 32 bits
```

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

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

- does the effective address calculation, but doesn’t access memory
IA32 Assembly Language examples ...

• 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?
• NB: mov instruction doesn’t update the condition code flags

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

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

- pass parameters [IA32: evaluate and push on stack]
- enter new function [IA32: push return address and jump to first instruction of function]
- allocate space for local variables [IA32: on stack by decrementing esp]
- save registers [IA32: on stack]

<function body>

- restore saved registers [IA32: from stack]
- de-allocate space for local variables [IA32: increment esp]
- return to calling function [IA32: pop return address from stack]
- remove parameters [IA32: increment esp]
IA32 AND X64

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]
IA32 Calling Conventions

• several IA32 procedure/function calling conventions

• will 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, others registers need to be saved and restored if used
  – 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 @ [ebp+8]
  p1 @ [ebp+12]
  ...

- local variables accessed with –ve offsets from ebp [see stack frame diagram]

  local variable 0 @ [ebp-4]
  local variable 1 @ [ebp-8]
  ...
Consider the IA32 Code for a Simple Function

```c
int f (int p0, int p1, int p2) { // parameters
    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` and 2 local variables `x` and `y`

- need to generate code for
  - calling function `f`
  - function `f` entry
  - function `f` body
  - function `f` exit
IA32 Code to Call Function $f$

- parameters $p0$, $p1$ and $p2$ pushed onto stack by caller right to left

```assembly
f(1, 2, 3)
push 3 ; push immediate values...
push 2 ; right to left
push 1 ;
call f ; call f
add esp, 12 ; add 12 to esp to remove parameters from stack
```

Push return address and jump to $f$
Function Entry

- need instructions to save ebp [old frame pointer] and ...
- initialize ebp [new frame pointer] and ...
- allocate space for local variables on stack and ...
- push non volatile registers used by function onto stack

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

<function body>                    ; function body

<function exit>                    ; function exit
```

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

• parameters pushed on stack and ...
• space already allocated for local variables

parameters $p0 @ [ebp+8]$ and $p1 @ [ebp+12]$
lords $x @ [ebp-4]$ and $y @ [ebp-8]$

• $x = p0 + p1$

    mov eax, [ebp+8] ; eax = p0
    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

NB: result returned in eax
Function Exit

• need instructions to unwind stack frame at function exit

```assembly
... pop ebx ; restore 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

• since using _cdecl convention caller will remove parameters from stack

• make sure you know why a stack frame needs to be created for each function call
IA32 Code for Accessing an Array

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

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

- int is 4 bytes
- assume array `a` is stored at absolute address `a` (eg. `a = 0x10000`)
- `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] assembler computes a + 8
add    eax, 3      // eax = a[2] + 3
mov    [a+4], eax   // a[1] = a[2] + 3
```
IA32 Code for Accessing an Array ...

```c
int *a = (int*) malloc(100*sizeof(int));  // array allocated on heap

int p() {
    int i = ...;  // local variable i @ [ebp-4]
    int j = ...;  // local variable j @ [ebp-8]
    ...
    a[i] = a[j] + 3;  // variable indices
}
```

- assume global variable a contains the address of the array allocated on heap

```asm
mov  edx, [a]  // edx -> a
mov  eax, [ebp-8]  // eax = j
mov  eax, [edx+eax*4]  // eax = a[j]
add  eax, 3  // eax = a[j]+3
mov  ecx, [ebp-4]  // ecx = i
mov  [edx+ecx*4], eax  // a[i] = a[j]+3
```
Tutorial 1 (next lecture)

- mixing C/C++ and IA32 Assembly Language

- example using Visual Studio, VC++ and MASM

- you have to write IA32 assembly language functions \( \text{min}(\text{int}, \text{int}, \text{int}), \text{p}(\text{int}, \text{int}, \text{int}, \text{int}) \)
  and \( \text{gcd}(\text{int}, \text{int}) \) [create files t1.h and t1.asm]

- you are given a “main” program t1Test.cpp which will call and test the functions you have written

- you are also given files fib32.h and fib32.asm as an example of how to write an IA32 assembly language function:
  (1) fib32.h contains the function definition (signature) and
  (2) fib32.asm contains the assembly language for the function fib(int) which calculates the \( n^{\text{th}} \) Fibonacci number

- t1Test.cpp also contains a C/C++ versions of fib(int)
Tutorial 1 ...

• create a VC++ Win32 Console Application [call it t1Test and specify that it creates a “main” file called t1Test.cpp]

• select project name (t1Test), click on Project menu, select "Build Customizations..." and tick masm

• paste the contents of the file t1Test.cpp from the web into the project file t1Test.cpp

• copy the files fib32.h and fib32.asm into the project directory and add them to the project [Project][Add Existing Item...]

• you can create your t1.h and t1.asm externally and include them into the project using [Project][Add Existing Item...]

• right click on the .asm files to make sure [Properties][General][Item Type] is set to Microsoft Macro Assembler
Tutorial 1 ...

fib32.h

- declares 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 _cdecl fib_IA32a(int); // external function
```

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

fib32.asm

- fib_IA32a(int) – simulating mechanical Debug mode code generation
- fib_IA32b(int) – simulating optimized Release mode code generation
- MASM specific directives at start of file
- .data and .code sections
- public
Tutorial 1 ...

- t1Test.cpp [_tmain]
- `#include fib32.h` and `t1.h`
- calls `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

- calls ALL versions of fib(n) for n = 1 to 20
- Visual Studio automatically compiles t1Test.cpp, assembles fib32.asm and t1.cpp and links them to produce an executable which is then run

- **WARNING**: Visual Studio on SCSS machines (eg. ICT Huts) has problems when source files are stored on a Network drive
Tutorial 1...

- make sure the configuration is x86 [you can delete the default x64 configuration as it is NOT applicable in this case]

- how to see the code generated by the VC++ compiler??
  - right click on C/C++ file name [Properties] [C/C++] [Output Files] [Assembler Output] and select Assembly, Machine Code and Source [listing has a .cod extension]
  - code generated in Debug and Release mode is different
Tutorial 1...

- you will need to define an the external global variable gin t.h
  
  ```
  extern "C" int g;  // external global variable g
  ```

- and you will also need to declare g in t1.asm
  
  ```
  .data ; start of a data section
  public g ; export variable g
  g DWORD 4 ; declare global variable g initialised to 4
  .code ; start of a code section
  ```

- setting breakpoints in .asm file
  
  - setting breakpoints in an assembly source file hasn't worked properly since VS2013
  - to debug min (for example), set breakpoint in .cpp file on the call to min
  - when breakpoint reached, select [Debug][Windows][Disassembly]
  - THEN single step using F11
  - hover mouse over register names to see their values etc.
x64 Basics

- extension of IA32
- originally developed 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 [*theoretically anyway*]
- $2^{64} = 16$ Exabytes = $16 \times 10^{18}$ bytes
x64 Function Calling

• use Microsoft Windows calling convention
• first 4 parameters passed in rcx, rdx, r8 and r9 respectively
• additional parameters passed on stack [right to left]
• stack always aligned on an 8 byte boundary
• caller must allocate 32 bytes of shadow space on stack
• rax, rcx, rdx, r8, r9, r10 and r11 volatile
• having so many registers often means:
  1. can use registers for local variables
  2. no need to use a frame pointer
  3. no need to save/restore registers
x64 Function Calling - Microsoft Windows Convention ...

- caller must allocate 32 bytes (4 x 8bytes) 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 in memory]

- called functions may use the *shadow space* for any purpose whatsoever and consequently may read and write to 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

- what are the advantages of having shadow space?
x64 Function Calling Unix/Linux

- brief description
- first six parameters passed in registers RDI, RSI, RDX, RCX, R8, R9 respectively
- additional arguments are passed on the stack [right to left]
- use of frame pointer [rbp], allocation of locals on stack and saving of registers as per Microsoft convention
- result returned in rax
- registers ebp, rbx, r12, r13, r14 and r15 non volatile
- no shadow space as per Microsoft convention
x64 Function Calling - Microsoft Windows Convention ...

- a more complex x64 stack frame
- callee has 5 parameters, so parameter 5 passed on stack
- parameters 1 to 4 passed in rcx, rdx, r8 and r9
- shadow space allocated
- old frame pointer saved and new frame pointer initialised [rbp]
- space allocated for local variables on stack [if needed]
- non-volatile registers saved on stack
x64 Function Calling - Microsoft Windows Convention ...

```c
_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 to declare 64 bit integers [Microsoft specific]
- alternatively declare 64 bit integers using `long long`

```c
#define INT64 long long
```

- parameter n passed to function in rcx
- leaf function [as fib doesn't call any other functions]
- usually easier to code with x64 assembly language rather than IA32 because a simpler stack frame is used and more registers are available
x64 Function Calling - Microsoft Windows Convention ...

```assembly
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
```

- code ONLY uses volatile registers
- **leaf function** so no need to allocate shadow space
x64 Function Calling - Microsoft Windows Convention ...

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

- uses %I64d to format a 64 bit integer
- parameters `a` and `b` passed to `xp2` 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]
x64 Function Calling (Microsoft Convention) ...

Fxp2 db 'a = %I64d b = %I64d a+b = %I64d', 0AH, 00H ; ASCII format string

Xp2: 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, rbx ; function result in rax = rbx {a+b}
add rsp, 32 ; deallocate shadow space
pop rbx ; restore rbx
ret ; return
x64 Function Calling (Microsoft Convention) ...

- instead of using rbx to preserve r9 across the call to printf, an alternate approach is to use a location its shadow space [eg. rsp+64]

xp2: sub rsp, 32 ; allocate shadow space
    lea r9, [rcx+rdx] ; printf parameter 4 in r9 [a+b]
    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
    mov [rsp+64], r9 ; save r9 in shadow space so...
    call printf     ; preserved across call to printf
    mov rax, [rsp+64]; result in rax = saved r9  [a+b]
    add rsp, 32     ; deallocate shadow space
    ret              ; return
x64 Function Calling (Microsoft Convention) ...

typical code generation strategy for an **non-leaf function**

- allocate enough stack space to accommodate calls to the function with the most parameters [*NB: must allocate a minimum 32 bytes for the shadow space + enough stack space to accommodate the maximum number of additional parameters passed on the stack*] 

- allocating stack space (which includes the shadow space) ONCE at start of function 

- use the same stack space [*and registers*] to pass parameters to ALL the functions it calls 

- straightforward for compiler to determine how much stack space is required
Typical code generation strategy...

function f(...)

...  
printf(5 parameters);
...  
printf(6 parameters);
...  
printf(2 parameters);
..

}  

• maximum number of parameters is 6
• need to allocate $6 \times 8 = 48$ bytes on stack at start
• in general, allocate $\max(32, n \times 8)$ bytes where $n$ is the maximum number of parameters (minimum 32 bytes of stack space for shadow space allocated)
• parameter 5 is moved directly to stack (NOT pushed) eg $\text{mov [rsp+32], eax}$
• reuse allocated stack space for all 3 calls to printf
• deallocate stack space on exit
Using Visual Studio

• fib64.h, fib64.asm and t2Test.cpp available on CS3021/CS3421 website

• need to create a console application and use the Configuration Manager to select a x64 solution platform

• one way to link with printf is to include the following at the head of t2.asm

```
#include <stdio.h>

extrn printf:near
```

• no x64 inline assembler, can use intrinsics defined in instrin.h instead
Summary

• 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++
  ▪ program the two most widely used CPUs in assembly language [IA32/x64 and ARM]