

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?

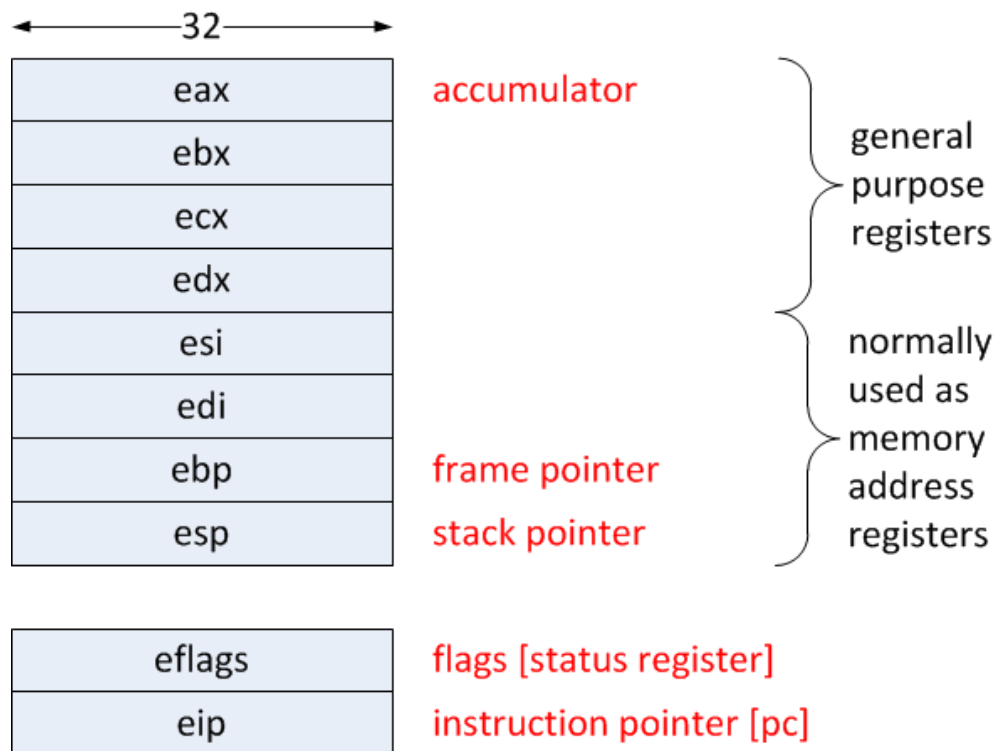
Some of the ideas and techniques used...

- smaller VLSI feature sizes [**1 micron (μ) -> 14nm**]
- increased clock rate [**8MHz -> 4GHz**]
- reduced vs complex instruction sets [**RISC vs CISC**]
- burst memory accesses
- integrated on-chip MMUs, FPU, ...
- pipelining
- superscalar [**multiple instructions/clock cycle**]
- multi-level on-chip instruction and data caches
- streaming SIMD [**single instruction multiple data**] instruction extensions [**MMX, SSE**]
- multiprocessor support
- hyper threading and multi core
- direct programming of graphics co-processor
- high speed point to point interconnect [**Intel QuickPath, AMD HyperTransport**]

IA32 [Intel Architecture 32 bit]

- IA32 first released in 1985 with the 80386 microprocessor
- IA32 still used today in 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+]

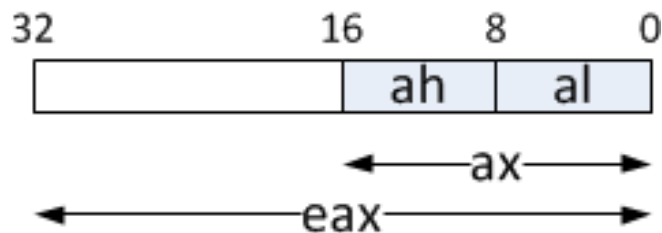
Registers [not as many as a typical RISC]



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**]

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 **NOT** allowed

Supported Addressing Modes

$[a]$ = contents of
memory address a

immediate	<code>mov eax, n</code>	$\text{eax} = n$
register	<code>mov eax, ebx</code>	$\text{eax} = \text{ebx}$
direct/absolute	<code>mov eax, $[a]$</code>	$\text{eax} = [a]$
indexed	<code>mov eax, [ebx]</code>	$\text{eax} = [\text{ebx}]$
indexed	<code>mov eax, [ebx + n]</code>	$\text{eax} = [\text{ebx} + n]$
scaled indexed	<code>mov eax, [ebx * s + n]</code>	$\text{eax} = [\text{ebx} * s + n]$
scaled indexed	<code>mov eax, [ebx + ecx]</code>	$\text{eax} = [\text{ebx} + \text{ecx}]$
scaled indexed	<code>mov eax, [ebx + ecx * s + n]</code>	$\text{eax} = [\text{ebx} + \text{ecx} * s + n]$

- address computed as the sum of a register, a scaled register and a 1, 2 or 4 byte signed constant n
- 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

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 decrement??

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

Assembly Language Tips...

- 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 ; explicitly 32 bits

- scaled indexed used to index into arrays of 1, 2, 4 or 8 byte values

int a[100] is an array of 4 byte values

double b[100] is an array of 8 byte values

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

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

Basic Instruction Set

mov	move
xchg	exchange
add	add
sub	subtract
imul	signed multiply
mul	unsigned multiply
inc	increment by 1
dec	decrement by 1
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, l, le, g, ge}	signed jump
j {b, be, a, ae}	unsigned jump
call	call subroutine
ret	return from subroutine

- 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
```

```
je     ...
```

; AND eax with itself, set flags and...
; jump if zero

Function Calling

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

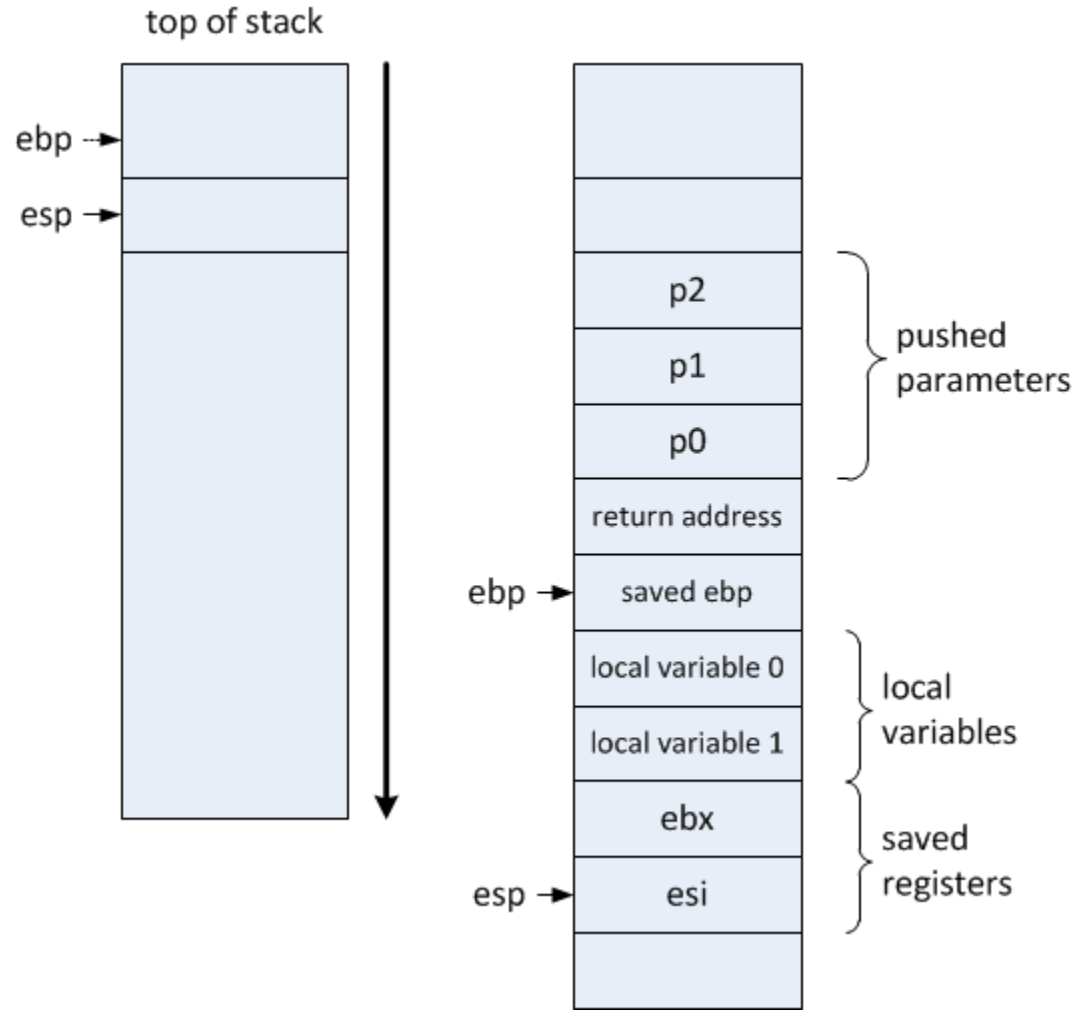
- pass parameters [**evaluate and push on stack**]
- enter new function [**push return address and jump to first instruction of function**]
- allocate space for local variables [**on stack by decrementing esp**]
- save registers [**on stack**]

<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 so parameters and local variables can be accessed easily
- 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

```
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*
- 2 local variables *x* and *y*

IA32 Code to Call Simple Function...

- parameters $p0$, $p1$ and $p2$ 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]

$p0$ stored at $[ebp+8]$

$p1$ stored at $[ebp+12]$

$p2$ stored at $[ebp+16]$

Accessing Local Variables of Simple Function...

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

x stored at $[ebp-4]$

y stored at $[ebp-8]$

- $x = p0 + p1$

```
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 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

...

```
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 new stack frame needs to be created for each function call

IA32 Code for Simple Array Accesses

```
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

```
mov    eax, [a+8];         // eax = a[2]  
add    eax, 3              // eax = a[2] + 3  
mov    [a+4], eax          // a[1] = a[2] + 3
```

IA32 Code for Simple Array Accesses...

```
int p()
{
    int i = ...;           // local variable i stored 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

```
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...

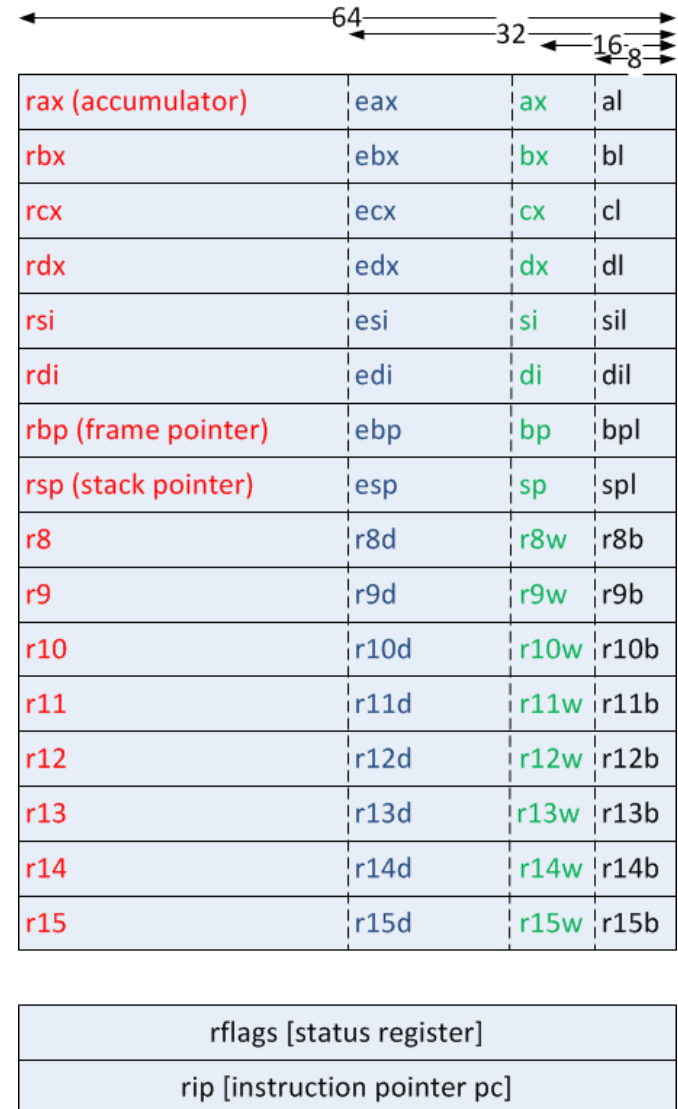
IA32codegen.cpp [**main**]

- #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

IA32 AND x64

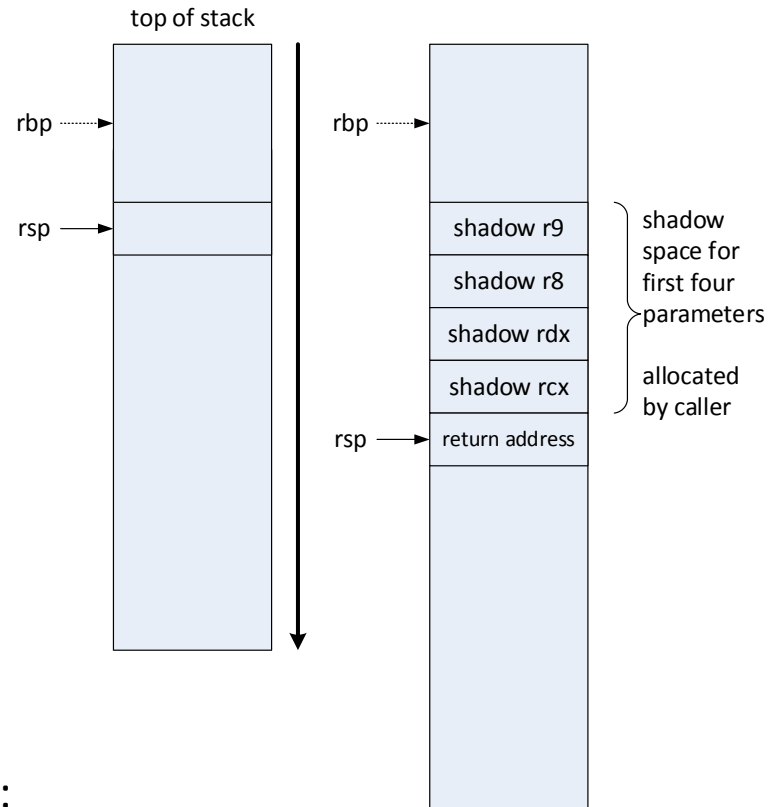
x64 Basics

- 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* instructions set
- 64 bit virtual and physical address spaces
[theoretically anyway]
- $2^{64} = 16 \text{ Exabytes} = 16 \times 10^{18} \text{ bytes}$



x64 Function Calling

- use Microsoft 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 *shadow space* on stack for parameters passed in rcx, rdx, r9 and r10
- rax, rcx, rdx, r8, r9, r10 and r11 volatile
- having so many registers available often means:
 1. can use registers for local variables
 2. no need to use a frame pointer
 3. no need to save/restore registers



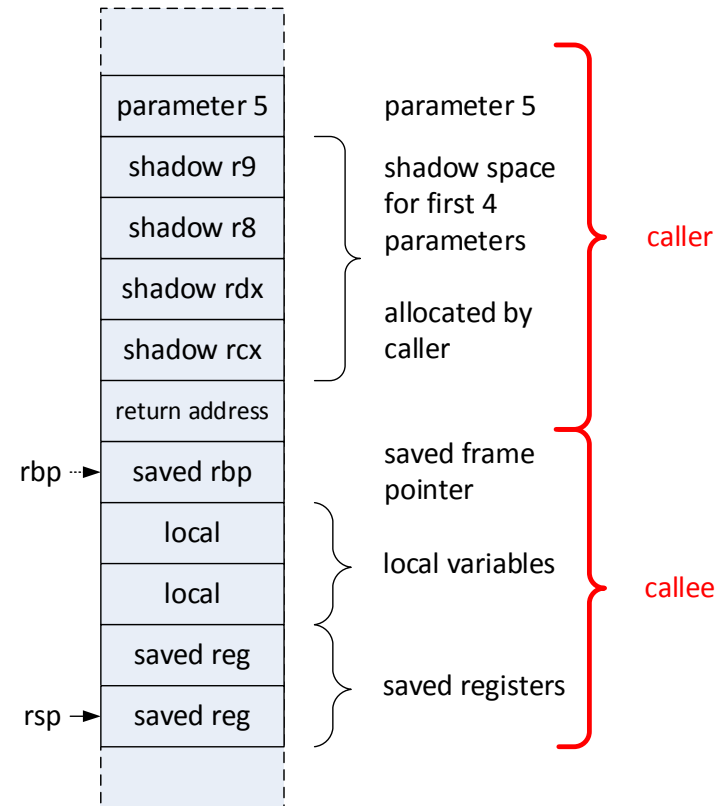
simple case
no frame pointer
no locals on stack (in registers)
no saved registers
leaf function

x64 Function Calling...

- in the Microsoft x64 calling convention, it is the responsibility of the caller to allocate 32 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 the *shadow space* to spill rcx, rdx, r8, and r9 [**spill = save in memory**]
- called functions may write data to and read data from its *shadow space* [**which is why it needs to be allocated**]
- the *shadow space* must be made available to all functions, even those with fewer than four parameters

x64 Function Calling...

- 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
- must allocate shadow space
- old frame pointer saved and new frame pointer initialised [**rbp**]
- space allocated for local variables on stack [**if needed**]
- registers saved on stack [**if needed**]



x64 Code for a 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` to declare 64 bit integers [**Microsoft specific**]

- alternatively

declare 64 bit integers using `long long`

`#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 Code for a Simple 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
```

NB: code ONLY uses volatile registers

x64 Code for a more Complex Function

```
_int64 xp2(_int64 a, _int64 b)
{
    printf("a = %l64d b = %l64d a+b = %l64d\n", a, b, a + b);
    return a + b;    // NB
}
```

- uses %l64d to format a 64 bit integer
- parameters **a** and **b** passed into 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 Code for a more Complex Function...

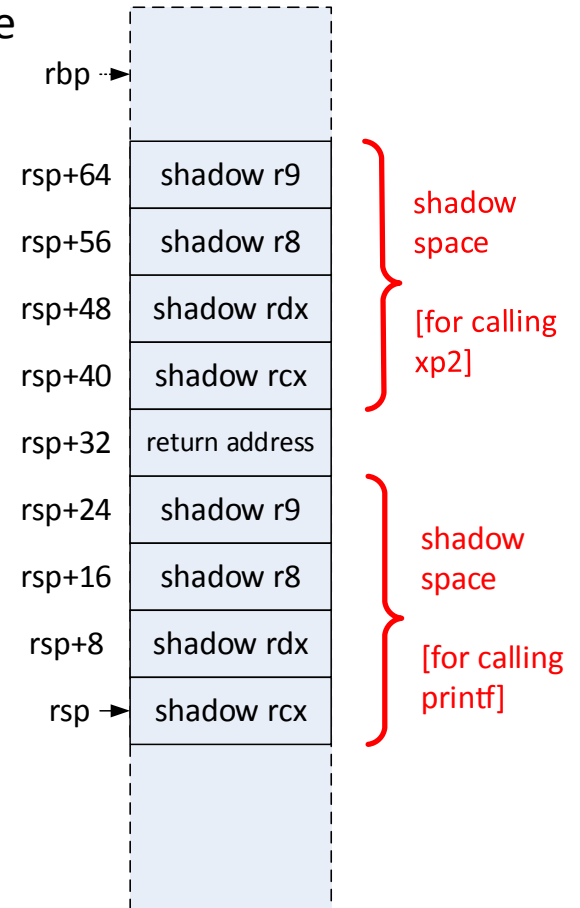
```
fxp2 db      'a = %l64d b = %l64d a+b = %l64d', 0AH, 00H    ; ASCII format string
```

```
xp2: push    rbx                ; save rbx
      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 {&fxp2}
      mov     rbx, r9            ; save r9 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 Code for a more Complex Function...

- 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
```



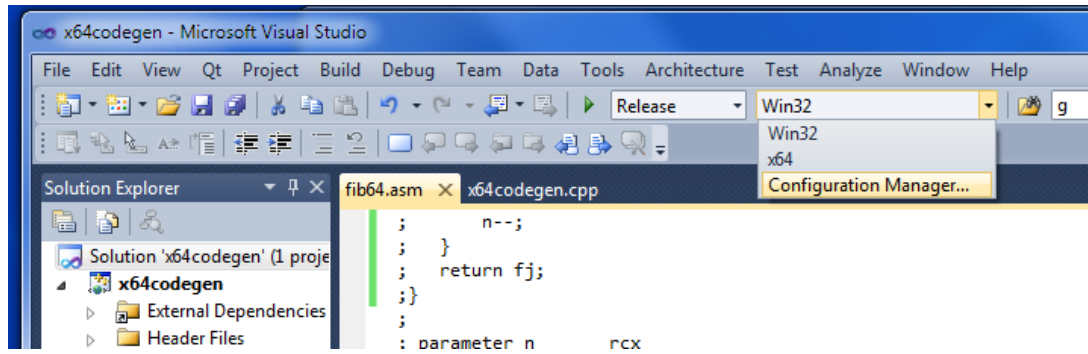
x64 Code for a more Complex Function...

Typical code generation strategy

- shadow space allocated ONCE at start of function
- allocate enough shadow space to accommodate calls to the function with the most parameters [**NB: must allocate a minimum 32 bytes**]
- use the same shadow space [**and registers**] to pass parameters to ALL the functions that a function calls
- straightforward for compiler to determine how much shadow space is required

x64 Code for a Simple Function...

- fib64.h, fib64.asm and x64codegen.cpp on CS3021/3421 website
- need to create a console application and use the Configuration Manager to select a x64 solution platform



- fib64.asm

extern printf:near ; allows external printf() function to be called

- ALL OK with VS2013, but printf linker problems with VS2015 [solve temporarily by configuring VS2015 to use VS2013 toolset for project]
- no x64 inline assembler, can use intrinsics defined in [intrin.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]

