# CPU2.0

CPU2.0 is a set of low-level routines libraries intended for beginners of x64 assembly language programming for the command-line or console interface. It caters to both Win and Linux 64-bit platforms. CPU2.0 is a full ABI-compliant library with strong emphasis on instructions and CPU interfacing.

CPU2.0 is a bare-metal library that doesn't hide anything from the users and makes no attempt to become an abstraction layer. CPU2.0 does not take away anything from a user's initial intention to learn the low-level details of x86 programming. The *non-inclusive*, *non-restrictive* nature of CPU2.0 should take away concerns from users that the use of this library might prevent one from dealing with or to learn the low-level details of assembly programming.

## Advantages

Debugger-like features and output to help learners build awareness around the instructions and machine state at any point of code development at all times. This can prevent time-consuming mistakes and bugs from accumulating.

Realtime visual feedback can greatly enhance instructional productivity and learning efficiency where proof-of-concept codes can be quickly built without going back and forth the debugger. Code behavior and effects can be quickly confirmed from the insertions of relevant CPU2.0 routines at any point of code.

Callee-saved registers will greatly reduce beginners common mistake when dealing with the registers. A user needs only to worry about the CPU state (registers, stack etc) at the caller's environment of which he / she is responsible of programming.

CPU2.0 offers a rich set of floating-point routines to enable the users to learn floating point instructions at the early phase of learning instead of integer-only assembly programming.

CPU2.0 can be used as an effective probing and debugging tool. It has a very small memory footprint and comes with a very thin abstraction layer.

CPU2.0 is a non-restrictive library where its use does not impose any constraints to access other libraries such as C and the OS APIs. Full compliancy making most of the routines callable from high-level languages such as C/C++.

## Limitations

CPU2.0 is a low-level library that does not have the luxury often offered by a HLL compiler such as verbose error messages, exception handling, bound-checking, garbage collector and similar things.

For portability reasons and due to its supportive nature, CPU2.0 have never been designed with intensive optimization in mind.

CPU2.0 routines are designed to achieve their basic functionalities only. Not suitable for industrial and scientific use that require extended features and should not be used as an inclusive library.

## Bug Report / Contact

soffianabdulrasad @ gmail . com (no spaces)
@SoffianAbdRasad (twitter)

## Download

Our main site:
http://www.cpu2.net

(BASELIB/CPULIB)
https://sourceforge.net/projects/baselibs/files/
Note: BASELIB/CPULIB are no longer maintained / updated

## The Library

CPU2.0 offers 4 type of binaries;

1) **.obj** - Win64 .obj format for static linking
2) **.dll** - Win64 DLL shared library
3) **.o** - Linux64 .o format (ELF) for static linking
4) **.so** - Linux64 .so shared library

To save space and effort, I have included in the respective folders the sample source codes with ample descriptions on how to access CPU2.0 from that source and steps in linking it to CPU2.0 and other co-library such as C.

## How To Use CPU2.0 Routines

Header references to each and every routines provided by CPU2.0 can be found in **cpu2api** documentation that comes with this distribution.

To use a particular routine, you issue a CALL instruction followed by the routine name you wish to use.

```
call   fpu_stack
call   dumpreg
```

**Parameter** Parameters are in the registers while a few routines use the stack. All parameters default to the CPU's architectural size. CPU2.0 supports up to 5 arguments and each argument is expected to be in the correct parameter number.

| | Parameters | | |
|---|---|---|---|
| | Win64 | Linux64 | Floats |
| Argument 5 | Shadow space+32 | R8 | |
| Argument 4 | R9 | RCX | |
| Argument 3 | R8 | RDX | |
| Argument 2 | RDX | RSI | XMM1 |
| Argument 1 | RCX | RDI | XMM0 |

For Win64, the integer and float arguments are sequenced by their positions, if both exists. For example if `Argument1` is a floating point and `Argument 2` is an integer, then the parameters are arranged as follow;

```
mov    rdx,[Argument2]     ;not in RCX
movq   xmm0,[Argument1]
```

Or in a reverse situation where `Argument1` is an integer, and `Argument2` is a floating point type;

```
movq   xmm1,[Argument2]    ;not in XMM0
mov    rcx,[Argument1]
```

For Linux64, the integer and float arguments stick to their default arrangements. For example, if `Argument1` is a floating point type, and `Argument2` is an integer, it becomes;

```
mov    rdi,[Argument2]
mov    xmm0,[Argument1]
```

Or in a reverse situation where `Argument1` is an integer, and `Argument2` is a floating point type;

```
movq   xmm0,[Argument2]
mov    rdi,[Argument1]
```

While maintaining compliancy with the AMD64 calling convention, CPU2.0 Linux64 version does not observe floating argument count in AL register. RAX is a callee-saved register.

Despite that, a few routines use different parameter than the rest of its containing library. This is to reflect the nature of such routines. For example, routine `prnchrs` and `flags` in 64-bit version uses the stack, via `push` and `pushfq` instructions respectively. Refer to the documentation for details. Observe the calling convention requirement, such as stack alignments.

**Arguments** Arguments must be supplied (prior to making the CALL) to routines that require arguments. Argument supplied must be in correct;

- number
- size
- argument format

Not all routines need arguments. The brackets found at each routine's header descriptions indicate the number of parameters / arguments needed by that routine.

**Size of Arguments** In most cases, arguments are assumed to be in their default architecture size. For example in 64-bit settings, the the default argument size is 8-bytes. However, due to the nature of particular routines, argument size may differ.

For example, displaying a char requires only a byte argument and not the entire 8 bytes register. In other case, a `prndbl` routine requires a full 8-byte argument size in floating point format while `prnflt` takes only half of that (using MOVD instead of MOVQ). Supplying a full 8-byte (DQ or REAL8) argument to a `prnflt` will result in semantic error even if it is syntactically correct. Below is an example;

```
mov    rbx,1829.127               ;FASM format.
mov    rbx,__float64__(1829.127)  ;NASM format.
movq   xmm0,rbx                   ;Argument in XMM0
call   prndbl                     ;displays 1829.127
```

Meanwhile (case: size is bigger than the required parameter size)

```
myFloat: dq 1829.127
...
movq   xmm0,[myFloat]      ;should use DD and MOVD.
call   prnflt              ;displays 1.554056E-31
```

Or in a reverse situation (size is smaller than the required parameter size)

```
myFloat: dd 1829.127
...
movd   xmm0,[myFloat]      ;should use DQ and MOVQ.
call   prndbl              ;displays 1.554056E-31
```

**Types of Arguments** Arguments supplied to each parameter may be either a value or a memory (both are just numbers anyway). A value could come from memory, an immediate, a variable, a constant or from another register. A value must be in valid format. Consult your assemblers on how to deal with various immediate formatting.

For example, presenting a hex immediate might be slightly different in NASM and MASM syntax. Similarly getting a value from memory might be different in FASM and MASM syntax. Floating point immediate formatting is definitely different for NASM and FASM.

Example below demonstrates how to copy a value from memory (variable) as the parameter to function `prnint`.

```
mov    rcx,[myNumber]      ;FASM/NASM syntax. RDI in Linux64
mov    rcx,myNumber        ;MASM syntax
call   prnint
...
myNumber dq 0xCAFEBABE     ;FASM/NASM syntax. Invalid in MASM
myNumber dq 0CAFEBABEh     ;MASM syntax. Valid for others as well.
```

Arguments can also come in form of address for parameters that require such argument especially when addressing larger data such as string or an array. Again, the format and method they are passed to the parameters might be different in different assemblers. For Win64 example;

```
mov    rcx,Hello           ;FASM/NASM syntax. Address as argument
mov    rcx,offset Hello    ;MASM syntax
call   prnstrz
...
Hello  db 'Hello CPU2.0!',0ah,0
```

For Linux64 example;

```
mov    rdi,Hello           ;FASM/NASM syntax. Address as argument
call   prnstrz
...
Hello  db 'Hello CPU2.0!',0ah,0
```

**Arguments Number & Sequence** You can supply the arguments in no particular order but the correct argument and number of arguments must be supplied in the correct registers according to the routines requirements. Routines however may require no arguments.

**Floating Point Arguments** All floating point arguments and return values are in XMM0 and XMM1 respectively. To transfer an argument of REAL8 or double data type, use MOVQ. To transfer an argument of REAL4 or single-precision type, use MOVD. REAL4 is of a *dword* magnitde (DD) and REAL8 is of *qword* in magnitude (DQ). REAL10 or extended precision are passed via pointers and therefore not bound to any of the SIMD registers. REAL10 is of *tword* / *tbyte* in magnitude (DT).

For example, given this declaration;

```
x dq 34.12          ;a REAL8
y dd -8.093         ;a REAL4
z dt 565490.33456   ;a REAL10
```

Using NASM / FASM syntax to make transfers

```
movq   xmm0,[x]      ;qword / real8 transfer
movd   xmm1,[y]      ;dword / real4 transfer

movq   xmm0,x        ;MASM
movd   xmm1,y
```

For floating-point immediates transfers

```
mov    ecx,__float32__(45.643)    ;NASM style
mov    ecx,45.643                 ;FASM
movd   xmm4,ecx                   ;dword / real4 transfer
```

To transfer a REAL10 argument instead, use a pointer to a data of DT type

```
mov    rcx,z         ;FASM/NASM. RDI for Linux64
mov    rcx,offset z  ;MASM/ML64. RDI for Linux64
call   prndblx
```

**Return Values** A number of CPU2.0 routines return a value in EAX/RAX (integer) and XMM0(floating point) . A few routines also return to EDX/RDX and XMM1 for the second return value, in case of multiple return values. The caller may use the value directly or temporarily store it at some other place for later use. This type of register returns are *direct return* values and denoted by the ***/x*** following the function name where x is the number of return values. Examples;

```
mem_load(1)/2        ;takes one argument and return 2 values
readint/1            ;takes no argument and return 1 value
```

However in some cases, return values are not explicitly stated in the header definition due to the nature of such routine. In this case, the routine uses *indirect/silent return* value. Example;

```
dblx2str(2)          ;convert a REAL10 to string
```

This routine does indeed return the filled buffer back to the caller, although not explicitly stated in the header description. Most of these indirect returns involve pass-by-reference string routines. An example for Linux64 version;

```
mov    rsi,theBuffer ;the address of buffer
mov    rdi,theValue  ;address of the value
call   dblx2str      ;call and return with a filled buffer

mov    rdi,theBuffer
call   prnstrz         ;display  the  print  buffer  returned  by
dblx2str     ...
theBuffer rb 30      ;theBuffer. Use “resb 30” in NASM .bss
theValue dt -19.117  ;a REAL10
```

## Cautions

**Data Alignment** When using SIMD instructions (SSE, AVX), various instructions need the data to be aligned to 16-byte boundary. That simply means that your data first byte's address or offset must land exactly at address ending with 0 such as 403080h. Do observe these alignment by using the *align* directive according to your assembler's syntax when using SSE-based routines such as *dumpxmm, dumpymm, prnymm and prnxmm*. One example;

```
section '.data' data readable writeable

mystring db 'Hello',0            ;next item's address no longer aligned
16
align 16                         ;so align it for next data below
mySSE dd 13.12,-9.12,80.14,11.04
gender db 'M'                    ;this single byte breaks alignment.
So...
align 16                         ;re-align for next SSE data
myXMM0  dq 78292.1823,9982.11289
```

For section(s) in object sources, use **align 32** or **align=32** (in NASM) to set the alignment at 32-byte granularity for AVX data. Examples;

```
section '.data' ... align 32 (FASM COFF's format. Not for PE or PE64)
section .data align=32       (NASM format)
```

**String Direction** All string operations of CPU2.0 routines are forward. If you are using backward direction in your own code, make sure to save the flags before calling any one of the CPU2.0 routines that alters the string direction. It is advisable to use CLD instruction once in a while to make sure that the string direction policy is always observed, particularly prior to making a CPU2.0 routines

**Bound-Checking and Scope** This library is a simple library that assumes the users are fully responsibile for their codes behaviour. This reflects the free-flowing nature of the CPU that knows no bound and scope. For example supplying a negative value to a function that expects a positive size would probably end up in erratic function behaviour. This library does not always provide such checking, such as strings and arrays.

**Personal and Educational Purposes Only** CPU2.0 is designed for light purposes only. Use of CPU2.0 in risky businesses and operations is not recommended.

## Examples

**Examples: Vector Applications**
One strong feature of CPU2.0 is the ability to deal with vector SIMD (MMX,SSE,AVX) instructions up to byte-granularity operations. That means one get to see how such extended registers behave and react to various packed instructions in multiple magnitudes of packed data;

1.Bytes (signed and unsigned)
2.Hex bytes (unsigned)
3.Words (Signed and unsigned)
4.Double words (unsigned and signed)
5.Quadwords (unsigned and signed)
6.Singles
7.Doubles

With this feature, a user can deal with almost any kind of packed data instructions and get the results immediately. As an added feature, each of the options have forward and reverse versions, which are useful when doing matrix operations and transpositions. Even more advance feature is the ability to view all those packed data in formatted and unformatted views. This is the only low-level library that allows one to actively view packed SIMD data and registers in multiple partitions and orientations.

For instance, a user get to see how saturated arithmetics is implemented using MMX saturated instructions **paddusb** using **dumpmmxf** routine (and its variants). And example in Win64 using FASM(assembler) GoLink(linker), and cpu2.dll from CPU2.0,

```
;Compile >> fasm this.asm
;Link >> golink /console this.obj cpu2.dll
format MS64 COFF
public start

extrn dumpmmxf
extrn prnline
extrn exitx

section '.data' data readable writeable
x db 254,0,254,127,254,254,253,250
y db 3,255,2,129,8,-2,4,6

section '.text' code readable executable
start:
        sub     rsp,40

        movq    mm0,qword[x]  ;populate MM0
        movq    mm7,qword[y]  ;populate MM7

        ;Initial view
        mov     rcx,0         ;view as unsigned bytes
        call    dumpmmxf      ;view formatted (aligned layout)
        call    prnline

        ;saturated add against packed unsigned bytes
        paddusb mm0,mm7

        ;after unsigned saturated arithmetics
        mov     rcx,0
        call    dumpmmxf

        call    exitx
```

Should yield this output below where bytes in MM0 retain their maximum values after being added with some random integers from MM7. The output;

```
mm0: 250|253|254|254|127|254|  0|254|   ;Initial MMX registers
mm1:   0|  0|  0|  0|  0|  0|  0|  0|
mm2:   0|  0|  0|  0|  0|  0|  0|  0|
mm3:   0|  0|  0|  0|  0|  0|  0|  0|
mm4:   0|  0|  0|  0|  0|  0|  0|  0|
mm5:   0|  0|  0|  0|  0|  0|  0|  0|
mm6:   0|  0|  0|  0|  0|  0|  0|  0|
mm7:   6|  4|254|  8|129|  2|255|  3|

mm0: 255|255|255|255|255|255|255|255|   ;After saturated add, mm0=mm0+mm7
mm1:   0|  0|  0|  0|  0|  0|  0|  0|
mm2:   0|  0|  0|  0|  0|  0|  0|  0|
mm3:   0|  0|  0|  0|  0|  0|  0|  0|
mm4:   0|  0|  0|  0|  0|  0|  0|  0|
mm5:   0|  0|  0|  0|  0|  0|  0|  0|
mm6:   0|  0|  0|  0|  0|  0|  0|  0|
mm7:   6|  4|254|  8|129|  2|255|  3|
```

Or maybe MMX instructions are too outdated for you? Perhaps you want to see how square root operations can be carried out simultaneously against four single-precisions at once via SSE **sqrtps** instruction. Using FASM, ld linker and cpu2.o (Linux64 version) as the main library;

```
;------------------------------
; fasm this.asm
; ld this.o cpu2.o -o this
; ./this
;------------------------------
format ELF64
public _start

extrn dumpxmmrf
extrn prnline
extrn exitx

section '.data' writeable align 16
align 16
x dd 45.45,81.0,34.23,42.01

section '.text' executable
_start:
        sub     rsp,8               ;alignment to 16

        movdqu  xmm0,dqword[x]      ;populate XMM0

        ;Initial view
        mov     rdi,8               ;view as packed singles
        call    dumpxmmrf           ;view XMMs in reversed, formatted
        call    prnline

        ;SQRTPS at work
        sqrtps  xmm1,xmm0

        ;See the result, in XMM1
        mov     rdi,8
        call    dumpxmmrf

        call    exitx
```

Output (reversed, formatted)

```
     xmm0:        +45.45|        +81.0|       +34.23|       +42.01|
     xmm1:          +0.0|         +0.0|         +0.0|         +0.0|
     xmm2:          +0.0|         +0.0|         +0.0|         +0.0|
     xmm3:          +0.0|         +0.0|         +0.0|         +0.0|
     xmm4:          +0.0|         +0.0|         +0.0|         +0.0|
     xmm5:          +0.0|         +0.0|         +0.0|         +0.0|
    --snip--

     xmm0:        +45.45|        +81.0|       +34.23|       +42.01|
     xmm1:     +6.741661|         +9.0|    +5.850641|    +6.481512|
     xmm2:          +0.0|         +0.0|         +0.0|         +0.0|
     xmm3:          +0.0|         +0.0|         +0.0|         +0.0|
     xmm4:          +0.0|         +0.0|         +0.0|         +0.0|
     xmm5:          +0.0|         +0.0|         +0.0|         +0.0|
    --snip--
```

Executing the same code setting in Win64, using ld and cpu.dll, and this time with forward view of XMMs instead of reversed view, yields;

```
;------------------------------
; fasm this.asm
; ld this.obj cpu2.dll -o this
;------------------------------
format MS64 COFF
public _start

extrn dumpxmmf
extrn prnline
extrn exitx

section '.data' readable writeable align 16
align 16
x dd 45.45,81.0,34.23,42.01

section '.text' readable executable
_start:
        sub     rsp,40

        movdqa  xmm0,dqword[x]      ;populate XMM0

        ;Initial view
        mov     rcx,8               ;view as packed singles
        call    dumpxmmf            ;view formatted
        call    prnline

        ;SQRTPS at work
        sqrtps  xmm1,xmm0

        ;See the result, in XMM1
        mov     rcx,8
        call    dumpxmmf
        call    exitx
```

Output (forward and formatted)

```
 xmm0:        +42.01|       +34.23|        +81.0|       +45.45|
 xmm1:          +0.0|         +0.0|         +0.0|         +0.0|
 xmm2:          +0.0|         +0.0|         +0.0|         +0.0|
 xmm3:          +0.0|         +0.0|         +0.0|         +0.0|
 xmm4:          +0.0|         +0.0|         +0.0|         +0.0|
 xmm5:          +0.0|         +0.0|         +0.0|         +0.0|
 --snip--

 xmm0:        +42.01|       +34.23|        +81.0|       +45.45|
 xmm1:     +6.481512|    +5.850641|         +9.0|    +6.741661|
 xmm2:          +0.0|         +0.0|         +0.0|         +0.0|
 xmm3:          +0.0|         +0.0|         +0.0|         +0.0|
 xmm4:          +0.0|         +0.0|         +0.0|         +0.0|
 xmm5:          +0.0|         +0.0|         +0.0|         +0.0|
--snip-
```

Or maybe, you want to know whether your PC does support AVX instruction sets, by trying to shuffle a few packed doubles? Now using ML64, GCC64 and cpu2.obj

```
;------------------------------
; ml64 /c this.asm
; gcc -m64 this.obj cpu2.obj -s -o this
;------------------------------
option casemap:none

externdef dumpymm:proc          ;These are from cpu2.obj
externdef prnline:proc
externdef ExitProcess:proc      ;This is from kernel32.dll

dseg segment page 'DATA'
align 32
x dq 204.561,891.211,-23.892,102.389
y dq 150.167,170.188,290.313,310.324
dseg ends

cseg segment para 'CODE'

main proc
        sub     rsp,40
        vmovdqu ymm0,ymmword ptr x  ;populate YMM0
        vmovdqu ymm1,ymmword ptr y  ;populate YMM1

        ;Initial view
        mov     rcx,9                   ;view as packed doubles
        call    dumpymm                 ;view unformatted,forward
        call    prnline

        ;Packed doubles shuffling
        vshufpd ymm5,ymm1,ymm0,2

        ;See the shuffled result, in YMM5
        mov     rcx,9
        call    dumpymm

        xor     ecx,ecx
        call    ExitProcess
main endp

cseg ends
end
```

Output (unformatted)

```
 ymm0: 102.389|-23.892|891.211|204.561|
 ymm1: 310.324|290.313|170.188|150.167|
 ymm2: 0.0|0.0|0.0|0.0|
 ymm3: 0.0|0.0|0.0|0.0|
 ymm4: 0.0|0.0|0.0|0.0|
 ymm5: 0.0|0.0|0.0|0.0|
 ymm6: 0.0|0.0|0.0|0.0|
--snip--

 ymm0: 102.389|-23.892|891.211|204.561|
 ymm1: 310.324|290.313|170.188|150.167|
 ymm2: 0.0|0.0|0.0|0.0|
 ymm3: 0.0|0.0|0.0|0.0|
 ymm4: 0.0|0.0|0.0|0.0|
 ymm5: -23.892|290.313|891.211|150.167|
 ymm6: 0.0|0.0|0.0|0.0|
--snip--
```

The four programs above demonstrated how CPU2.0 offers such a great flexibility in viewing any MMX, SSE or AVX registers in any direction and layout, in multiple magnitudes, and in formatted / unformatted views. They also demonstrated how CPU2.0 can run seamlessly on both platforms with very thin layer of abstractions - no complicated linking, no third party library dependency (except with the kernel services) and no confusing include files. CPU2.0 is essentially baremetal.

**Examples: Floating-Point and FPU**
While previous examples deal with floating-point from application standpoint, CPU2.0 also offers routines with floating-point educational materials and thus providing a full set of FPU helper routines.

This is implemented via many routines such as **fpdinfo**, **fpfinfo**, **fpu_stack**, **fpbind**, **fpbin**, **fpu_sflag**, **fpu_tag**, **fpu_cflag** and the likes. In the following examples, one can see how the FPU, which is an IEEE-754 compliant hardware, maintains both double-precision and single-precision data in their three-part floating point binary format:

- One sign bit
- Exponent bits
- Mantissa/Significand bits

In addition, there's also a hidden bit implicitly embedded in a floating point quantity of the CPU/FPU.

All these information are revealed by both **fpdinfo** (for double-precision) and **fpfinfo** (for single-precision). Example below is for Win64, via NASM, using ld as the linker and using our library cpu2.dll.

```
;------------------------
; nasm -f win64 this.asm
; ld this.obj cpu2.dll -o this
;------------------------
        global _start

        ;import these from cpu2.dll
        extern fpdinfo
        extern fpfinfo
        extern prnstrz
        extern prnline
        extern exitx
```

```
        section .data
msg1:   db 'IEEE 754 float64 format for: ',0ah,0
msg2:   db 'IEEE 754 float32 format for: ',0ah,0
myDbl:  dq -786.561         ;a REAL8
myFlt:  dd 0.009239123      ;a REAL4

        section .text
_start:
        sub     rsp,40

        mov     rcx,msg1        ;Display message
        call    prnstrz
        movq    xmm0,[myDbl]    ;View real8 FP format
        call    fpdinfo

        call    prnline

        mov     rcx,msg2
        call    prnstrz
        movd    xmm0,[myFlt]    ;View real4 FP format
        call    fpfinfo

        call    exitx
```

Output:

```
IEEE 754 float64 format for:
-786.561 = C088947CED916873
1.10000001000.1000100101000111110011101101100100010110100001110011
S.EXPONT   +1.MANT
SIGN: 1
EXP : +9 (1032 - 1023)
MANT: 1.536251953124999

IEEE 754 float32 format for:
0.009239 = 3C175FB1
0.01111000.00101110101111110110001
S.EXPONT 1.MANT
SIGN: 0
EXP : -7 (+120 - 127)
MANT: 1.182608
```

All these technical information revealed by those two routines are important in floating-point binary study. For example, the nett exponent tells how many bits off the **Mantissa** part that are used to identify the integral part of your floating-point quantity, plus the hidden bit.

From the example above, the double value -786.561 has the nett exponent of +9. That means the highest 9 bits of Mantissa, plus 1 hidden bit should sum up to integer 786 while the rest of the bits are used to represent the fraction part.

```
0.100010010....Higher 9 bits of mantissa
1              +1 hidden bit
1 100010010b   Sums up to 786
```

Now that the secret is out, how about learning some more about the FPU instructions and non-comformant floating point such as a REAL10 / Extended precision? Example below demonstrates how one can learn multiple FPU operations with the help of **fpu_stack** routine. Using Linux64, FASM and cpu2.o as the library;

```
;-----------------------
; fasm this.asm
; ld this.o cpu2.o -o this
;-----------------------
        format ELF64
        public _start

        ;import these from cpu2.o
        extrn fpu_stack
        extrn fpu_sflag
        extrn prnline
        extrn exitx

        section '.data' writeable
myExt   dt 56.154           ;a REAL10 value

        section '.text' executable
_start:
        sub     rsp,8       ;align stack

        finit               ;clear/init FPU
        fld     [myExt]     ;Load a REAL10
        fldpi               ;Load an FPU PI constant
        fldz                ;Load an FPU zero
        call    fpu_stack   ;View FPU stack after loading some values
        call    prnline

        ;FDIV operation to produce an infinity (Div-by-Zero)
        fdiv    st2,st0
        call    fpu_stack   ;See what happens in ST2
        call    prnline

        ;View the FPU status flag
        call    fpu_sflag   ;Watch the 'Z' (div-by-zero) flag is up

        call    exitx
```

Output:

```
    st0: +0.000000000000000000
    st1: +3.141592653589793238
    st2: +56.154000000000000000
    st3: ...
    st4: ...
    st5: ...
    st6: ...
    st7: ...

    st0: +0.000000000000000000
    st1: +3.141592653589793238
    st2: -Inf-
    st3: ...
    st4: ...
    st5: ...
    st6: ...
    st7: ...

    B C3 TP TP TP C2 C1 C0 IR SF  P  U  O  Z  D  I
    0  0  1  0  1  0  0  0  0  0  0  0  0  1  0  0 =2804
```

This way, the users of CPU2.0 do not have to run back and forth the debugger to explore deep into the dynamics of FPU instructions and technicalities. One can learn, modify, program and see all of them on-the-fly, as the codes progress. This is the most effective way of learning and even teaching where everything can be built up and modified quickly without going through the time-consuming process of debugging.

**Examples: Memory Viewing and Manipulation**
CPU2.0 comes with 5 memory viewing routines plus some other memory-related routines. They are **memviewn**, **memview**, **memviewc**, **memviewb** and finally **stackview**.

All these offer positive and negative **offsets** so to enable one to navigate through the selected memory space up and down. The listed offsets are accurate so that one can use them in building valid addressing modes to access (read or write) any portion of memory area allowed for manipulation. **memviewn** and **memviewb** offers Little-Endian views so that you get to see how the CPU actually lays out information in memory. An example on Win64, via NASM and GCC64:

```
;---------------------------------
;Win64 Compile >> nasm -f win64 this.asm
;Win64 Link    >> gcc -m64 this.obj cpu2.dll -s -o this.exe
;---------------------------------
global main
extern memviewc             ;import these from cpu2.dll
extern prnstrz
extern prnline

section .data
msg: db 'Hello CPA',0ah,0

section .code
main:
    sub rsp,40           ;Shadow space and alignment

    mov rdx,100          ;view 100 bytes
    mov rcx,msg          ;View this memory area.
    call memviewc        ;view as string bytes. Reversed Little-Endian

    ;memory manipulation. Change 'A' in 'CPA' to 'U'
    mov byte[rcx+8],'U'  ;From the output, 'A' is at offset 8

    call prnline

    ;Now watch it again.
    mov rdx,100
    mov rcx,msg
    call memviewc        ;See 'A' is changed to 'U'

    ;For proof, print the msg
    mov rcx,msg
    call prnstrz

    add rsp,40
    ret
```

The output:

```
    0000000000404010| H e l l o _ C P A _ _ _ _ _ _ _ |+F|+15
    0000000000404020| p - @ _ _ _ _ _ _ _ _ _ _ _ _ _ |+1F|+31
    0000000000404030| _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ |+2F|+47
    0000000000404040| _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ |+3F|+63
    0000000000404050| _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ |+4F|+79
    0000000000404060| _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ |+5F|+95
    0000000000404070| _ + @ _                         |+63|+99

    0000000000404010| H e l l o _ C P U _ _ _ _ _ _ _ |+F|+15
    0000000000404020| p - @ _ _ _ _ _ _ _ _ _ _ _ _ _ |+1F|+31
    0000000000404030| _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ |+2F|+47
    0000000000404040| _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ |+3F|+63
    0000000000404050| _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ |+4F|+79
    0000000000404060| _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ |+5F|+95
    0000000000404070| _ + @ _                         |+63|+99
    Hello CPU
```

Or perhaps you want to see on-the-fly how your stack and TOS is changing after pushing two quadword items?

```
;------------------------
; ml64 /c this.asm
;------------------------
; gcc -m64 this.obj cpu2.obj -o this.exe
;------------------------
option casemap:none
externdef stackview:proc    ;import these from cpu2.obj
externdef prnline:proc

.data
x dq 45h
y dq 77h

.code
main proc

    ;alignment and shadow space
    sub rsp,40

    ;Original stack before adding two items
    mov rcx,5     ;see 5 items of stack
    call stackview
    call prnline

    ;push two items onto the stack
    push x
    push y

    ;Stack after two "push"
    mov rcx,5
    call stackview

    add rsp,56      ;Restore all stack
    ret

main endp
end
```

Output:

```
0000000000000000 |000000000061FE78  ;First stackview
000000000040FA20 |000000000061FE70
0000000000000005 |000000000061FE68
0000000000000000 |000000000061FE60
00000000004013E8 |000000000061FE58

0000000000000005 |000000000061FE68  ;Second stackview after two PUSHes
0000000000000000 |000000000061FE60
00000000004013E8 |000000000061FE58  ;Old TOS
0000000000000045 |000000000061FE50
0000000000000077 |000000000061FE48  ;New TOS (RSP=0x61FE48, TOS=77h)
```

Observe that as per convention, CPU2.0 shows the stack growth downwards and shrinks upwards the memory space in order to preserve the semantics of **push** and **pop** stack operations of the x86 architecture.

Or just maybe you're on Linux64 right now and want to see how a decimal number 1234567 is maintained by the CPU in memory. Using NASM, on Linux64 and cpu2.o binary

```
;------------------------
; nasm -f elf64 this.asm
; ld this.o cpu2.o -o this
;------------------------
        global _start
        extern memviewb    ;from cpu2.o
        extern prnline

        section .data
myVal:  dq 1234567         ;an 8-byte value

        section .text
_start:
        mov     rdx,1      ;Option: view in decimal format
        mov     rsi,10     ;view ten bytes anyway
        mov     rdi,myVal  ;view at this address
        call    memviewb   ;View as flat Little-Endian

        call    prnline

        xor     edi,edi    ;Exit to terminal
        mov     eax,60
        syscall
```

Output (in 10 flat bytes):

```
000 |00000000006000DD
000 |00000000006000DC
000 |00000000006000DB       ;High address
000 |00000000006000DA
000 |00000000006000D9
000 |00000000006000D8       ;a sequence of 8 bytes (quadword,DQ)
000 |00000000006000D7
018 |00000000006000D6
214 |00000000006000D5
135 |00000000006000D4       ;Low address
```

You must be surprised to see that a decimal 1234567 does not even look like 1234567 in flat eight bytes sequence of memory, even when represented in decimal format.

**mem_load** routine lets you load a content of a file to memory for whatever reasons. You can then use other supporting routines such **memview**, **memviewc** or **memviewn** (to display the memory content) or **prnstrz** (if the loaded file is a text file) or use **mem_tofile** to save the content to a new file for future references.

Using ML64, GCC64 and cpu2.dll you can extend your creativity via **mem_load** routine and supporting others. In this example, you can load and view the content of our own cpu2.dll. (Warning: Long output)

```
;---------------------------------
;Win64 Compile >> ml64 /c this.asm
;Win64 Link    >> gcc -m64 this.obj cpu2.dll
;---------------------------------
option casemap:none

externdef mem_load:proc       ;import these from cpu2.dll
externdef mem_free:proc
externdef memview:proc
externdef ExitProcess:proc    ;import this from kernel32.dll

.data
myFile db 'cpu2.dll',0        ;load this to mem, and see content
```

```
.code
main proc
        sub rsp,40

        mov rcx,offset myFile
        call mem_load   ;Return size in RDX. Pointer in RAX

        mov r8,rax      ;Duplicate pointer

                        ;Size is already in RDX (2nd arg)
        mov rcx,rax     ;use pointer
        call memview    ;View content of entire "cpu2.dll"

        mov rcx,r8
        call mem_free   ;Free memory from "mem_load"

        xor ecx,ecx
        call ExitProcess
main endp
end
```

Sample output (somewhere in the middle of such long output):

```
...
000000000002A590| 65 72 73 65 00 73 74 72 |+A597|+42391
000000000002A598| 5F 74 72 69 6D 00 73 74 |+A59F|+42399
000000000002A5A0| 72 5F 77 6F 72 64 63 6E |+A5A7|+42407
000000000002A5A8| 74 00 73 74 72 5F 74 6F |+A5AF|+42415
000000000002A5B0| 6B 65 6E 00 73 74 72 5F |+A5B7|+42423
000000000002A5B8| 66 69 6E 64 00 73 74 72 |+A5BF|+42431
000000000002A5C0| 5F 66 69 6E 64 7A 00 73 |+A5C7|+42439
000000000002A5C8| 74 72 5F 61 70 70 65 6E |+A5CF|+42447
000000000002A5D0| 64 7A 00 73 74 72 5F 61 |+A5D7|+42455
...
```

The two right-most colums are the offsets parts, both in hex and decimal respectively to let you reference or pinpoint to any particular byte you're interested in viewing in your addressing modes build up. The offsets are precise.

**Examples: Converters**
CPU2.0 is very rich in string conversion routines, both ways. You can practically convert from any base to another base, up to base-36. All of the number display routines (**prnxxx**) are basically some of them. On the other hand, each of them have their (**xxx2str**) counterparts so that you can choose your own string display functions from other sources much later instead of using the defaults provided by CPU2.0.

For example, to display a double-precision, CPU2.0 uses **prndbl** and **prndble** functions. In addition, you can use their xxx2str (to string) counterparts, namely, **dbl2str** and **dble2str** to do similar jobs then display them much later using any string display services you know, such as C's **printf**. (Please note that CPU2.0 does not use any C library)

Example below demonstrates how a user input is converted to Base-19 and Base-25 strings using **int2str** and **bconv** converter routines respectively. To make it more fun, we use an interactive program via FASM, linker ld, cpu2.o and on Linux64. Observe the calling convention used;

```
;------------------------
; fasm this.asm
; ld this.o cpu2.o -o this
;------------------------
        format ELF64
        public _start

        ;import these from cpu2.o
        extrn prnstrz
        extrn readint
        extrn int2str
        extrn bconv
        extrn prnline

        section '.data' writeable
prompt1 db 'Enter an integer to convert: ',0
msg19   db 'Your integer in Base-19 is: ',0
msg25   db 'Your integer in Base-25 is: ',0
val1    dq 0
buff    rb 80

        section '.text' executable
_start:
        sub     rsp,8

        mov     rdi,prompt1     ;Display prompt
        call    prnstrz
        call    readint         ;Get user input
        mov     [val1],rax      ;copy to val1

        mov     rcx,0           ;Option. Unsigned
        mov     rdx,19          ;Use Base-19
        mov     rsi,buff        ;Buffer to keep the string
        mov     rdi,[val1]      ;The value to convert
        call    int2str

        mov     rdi,msg19       ;Display msg19
        call    prnstrz
        mov     rdi,buff        ;Display the returned string
        call    prnstrz

        call    prnline

        mov     rdi,msg25       ;Display msg25
        call    prnstrz
```

```
        mov     rsi,25          ;Use Base-25
        mov     rdi,[val1]      ;The value to convert
        call    bconv

done:   call    prnline
        xor     edi,edi
        mov     eax,60
        syscall
```

Output (A user input a Hex value)

```
Enter an integer to convert: C334EFh
Your integer in Base-19 is: 5332GA
Your integer in Base-25 is: 17IIML
```

On the other hand, one can also convert a null-terminated string to its integer or float counterpart. This can be done via **str2int**, **str2dbl**, **str2flt** functions and then display the converted value any way necessary, either as signed, unsigned or to any other number format. Using NASM, on Win64, cpu2.obj and GCC64 as the linker;

```
;------------------------
; nasm -f win64 this.asm
; gcc -m64 this.obj cpu2.obj -s -o this.exe
;------------------------
        global main                     ;GCC entry point

        ;import these from cpu2.obj
        extern str2int
        extern str2flt
        extern prnhexu
        extern prnoctu
        extern prnflt
        extern prnline

        section .data
iVal:   db '234566o',0              ;An unsigned octal integer
xVal:   db '-234566o',0             ;A signed octal
fVal:   db '0.0000000000000031265',0 ;a very small float

        section '.text' executable
main:
        sub     rsp,40

        ;Convert string to integer
        mov     rcx,iVal  ;An octal string
        call    str2int   ;return integer in RAX
        mov     rcx,rax
        add     rcx,1     ;Proof that returned value is an integer
        call    prnoctu   ;Display as unsigned octal
        call    prnline

        ;Convert string to integer
        mov     rcx,xVal  ;Another octal string
        call    str2int   ;return integer in RAX
        mov     rcx,rax
        call    prnhexu   ;Display/convert to/as unsigned Hex
        call    prnline

        ;Convert string to Float
        mov     rcx,fVal
        call    str2flt   ;Return single-precision in XMM0
        call    prnflt    ;Display a float in XMM0

done:   add     rsp,40
        ret
```

Output:

```
234567
FFFFFFFFFFEC68A
3.126499E-15
```

**Examples: Registers Dump**
The key to mastering assembly programming is the constant awareness around operands and their behavior. To keep track of their behaviors, CPU2.0 offers various register dumping routines, designed to return the most current state of the registers, such as;

- **dumpreg**- View all 64-bit registers in Unsigned Hexadecimal
- **dumpregd**- View all 64-bit registers in Signed integer format
- **dumpregdu**- View all 64-bit registers in Unsigned integer format
- **dumprego**- View all 64-bit registers in Signed octal format
- **dumpregou**- View all 64-bit registers in Unsigned octal format
- **dumpregb**- View all 64-bit registers in Unsigned binary format
- **flags**- View the CPU's RFLAG status

These registers are very useful for users that deal with various numeral systems, such as octal, to perform complex integer arithmetic and bitwise operations. CPU2.0 allows the users to call any of these routines at any point where they are needed and can be called repetitively without limitations.

For example, to see how the bitwise **AND** instruction works against RAX register in order to find out the next lower modulo 16 value of an operand (Win64, GoLink, ML64,cpu2.dll);

```
;------------------------
; ml64 /c this.asm
; golink /console this.obj cpu2.dll -o this
;------------------------
option casemap:none

;import these from cpu2.dll
externdef dumpreg:proc
externdef exitx:proc

.code
start proc
      sub     rsp,40              ;align stack
      mov     rax,34567h
      and     rax,-16             ;even down to modulo 16
      call    dumpreg             ;See change in RAX, as hex.
      call    exitx
start endp
end
```

Output:

```
RAX|0000000000034560 RCX|0000000000222000 RDX|0000000000401000
RBX|0000000000000000 R8 |0000000000222000 R9 |0000000000401000
RDI|0000000000000000 RSI|0000000000000000 R10|0000000000000000
R11|0000000000000000 R12|0000000000000000 R13|0000000000000000
R14|0000000000000000 R15|0000000000000000 RBP|0000000000000000
RSP|000000000014FF50 RIP|000000000040100F [UHEX]
```

The symbol **[UHEX]** is an added feature serving as a constant reminder that you're currently viewing the registers in Hexadecimal format, and not in any other format. This is important because numbers of different format may appear the same in certain cases.

The output in RAX shows that your value is now even down to the nearest modulo 16 (divisible by 16) via the **and rax,-16** instruction. This is a familiar technique mostly found in string length, fast memory copying algorithms and even stack/memory alignment.

Similarly, a beginner can quickly learn and build confidence dealing with signed and unsigned binaries by using **dumpregb**. For example, to see how the CPU stores negative numbers in **Two's complement format**, two of the most common techniques are using **NEG** and **NOT** instructions. Using NASM, GCC64 linker, cpu2.dll, one can quickly build up two test-cases and see the results instantly without involving any unproductive guesswork.

```
;------------------------
; nasm -f win64 this.asm
; gcc -m64 this.obj cpu2.obj -s -o this.exe
;------------------------
        global main

        ;import these from cpu2.dll/cpu2.obj
        extern dumpregb
        extern prnline

        section .text
main:
        sub     rsp,40

        mov     rax,45      ;a positive integer
        mov     rbx,45        ;The same value
        call    dumpregb    ;View RAX and RBX in unsigned binaries

     ;Using NEG against RAX
        neg     rax         ;Negate RAX. Two's complement form of +45.
        call    prnline

     ;Using NOT against RBX
        not     rbx         ;Invert RBX. One's Complement form
        add     rbx,1       ;+1. Two's complement (negative form) of +45
        call    dumpregb    ;View as unsigned bits. Watch RAX and RBX

        add     rsp,40
        ret
```

From the output below, one can immediately see that both RBX and RAX are having the same values, that is in Two's Complement form to represent the negative value of 45

```
RAX|00000000 00000000 00000000 00000000 00000000 00000000 00000000 00101101
RCX|00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
RDX|00000000 00000000 00000000 00000000 00000000 00011110 00010011 01010000
R8 |00000000 00000000 00000000 00000000 00000000 00011110 01000011 10110000
R9 |00000000 00000000 00000000 00000000 00000000 00011110 00010011 01010000
RBX|00000000 00000000 00000000 00000000 00000000 00000000 00000000 00101101
--snip--

RAX|11111111 11111111 11111111 11111111 11111111 11111111 11111111 11010011
RCX|00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
RDX|00000000 00000000 00000000 00000000 00000000 00011110 00010011 01010000
R8 |00000000 00000000 00000000 00000000 00000000 00011110 01000011 10110000
R9 |00000000 00000000 00000000 00000000 00000000 00011110 00010011 01010000
RBX|11111111 11111111 11111111 11111111 11111111 11111111 11111111 11010011
--snip–
```

On the same note, if you're dealing with lots of signed and unsigned integer operations, you can use dumpregd and dumpregdu respectively.

**Examples: System Probe**

Beside being educational, CPU2.0 is also strong in system works and very handy when used in reverse engineering, via expert use of various CPU2.0 routines. For example, the **memviewc** is also capable of viewing other memory area using negative offsets, such as to see the **MZ** header of your own executable. An example on Win64, using FASM and GoLink, and cpu2.dll

```
;----------------------------------
;Win64 Compile >> fasm this.asm
;Win64 Link    >> golink /console this.obj cpu2.dll kernel32.dll
;----------------------------------
format MS64 COFF
public start

extrn memviewc          ;import this from cpu2.dll
extrn ExitProcess       ;import this from kernel32.dll

section '.data' data readable writeable
msg: db 'Hello CPA',0ah,0

section '.text' code readable executable
start:

        sub rsp,40          ;Shadow space and alignment

        mov rdx,-2000h      ;view -8192 negative offsets. Yours may vary
        mov rcx,msg         ;View this starting area but further back
        call memviewc       ;See your "MZ" header and stub

        xor ecx,ecx
        call ExitProcess
```

Output (long output)

```
    ...
    000000000040007F| _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ |-1F90|-8080
    000000000040006F| _ _ _ _ \ _ _ f _ _ _ d _ _ E P |-1FA0|-8096
    000000000040005F| _ _ _ _ _ _ _ _ m o c . l o o T |-1FB0|-8112
    000000000040004F| v e D o G . w w w _ k n i L o G |-1FC0|-8128
    000000000040003F| _ _ _ ` ! _ L _ ! _ _ _ _ _ _ $ |-1FD0|-8144
    000000000040002F| _ _ ! m a r g o r P _ 4 6 n i W |-1FE0|-8160
    000000000040001F| _ _ _ _ _ _ _ @ _ _ _ _ _ _ _ _ |-1FF0|-8176
    000000000040000F| _ _ _ _ _ _ _ _ _ _ _ _ _ l Z M |-2000|-8192
```

If you need to reveal more technical information about your executable or DLL header, you can use **memviewn** or **memview** routines instead, in combination with **dumpreg** where the revealed information is in numbers instead of texts.

```
;----------------------------------
;Win64 Compile >> fasm this.asm
;Win64 Link    >> golink /console this.obj cpu2.dll kernel32.dll
;----------------------------------
format MS64 COFF
public start

extrn memview           ;import these from cpu2.dll
extrn prnline
extrn dumpreg
extrn ExitProcess       ;import this from kernel32.dll

section '.data' data readable writeable
msg: db 'Hello CPA',0ah,0

section '.text' code readable executable
start:

        sub rsp,40        ;Shadow space and alignment
```

```
    mov rax,msg
    call dumpreg      ;Verify position of .data section

    call prnline      ;Separate the two output

    mov rdx,200       ;view 200 bytes
    mov rcx,400000h   ;View from BASE address. Around RAX
    call memview      ;'MZ' magic number, COFF stub and others

    xor ecx,ecx
    call ExitProces
```

Output:

```
RAX|0000000000402000 RCX|000000000026F000 RDX|0000000000401000
RBX|0000000000000000 R8 |000000000026F000 R9 |0000000000401000
RDI|0000000000000000 RSI|0000000000000000 R10|0000000000000000
R11|0000000000000000 R12|0000000000000000 R13|0000000000000000
R14|0000000000000000 R15|0000000000000000 RBP|0000000000000000
RSP|000000000014FF30 RIP|000000000040100E [UHEX]

0000000000400000| 4D 5A 6C 00 01 00 00 00 |+7|+7     ;'MZ' magic
0000000000400008| 02 00 00 00 FF FF 00 00 |+F|+15
0000000000400010| 00 00 00 00 11 00 00 00 |+17|+23
0000000000400018| 40 00 00 00 00 00 00 00 |+1F|+31
0000000000400020| 57 69 6E 36 34 20 50 72 |+27|+39
0000000000400028| 6F 67 72 61 6D 21 0D 0A |+2F|+47
0000000000400030| 24 B4 09 BA 00 01 CD 21 |+37|+55
0000000000400038| B4 4C CD 21 60 00 00 00 |+3F|+63
0000000000400040| 47 6F 4C 69 6E 6B 20 77 |+47|+71
0000000000400048| 77 77 2E 47 6F 44 65 76 |+4F|+79
0000000000400050| 54 6F 6F 6C 2E 63 6F 6D |+57|+87
0000000000400058| 00 00 00 00 00 00 00 00 |+5F|+95
0000000000400060| 50 45 00 00 64 86 03 00 |+67|+103  ;'PE' magic
0000000000400068| 7F CE 19 5C 00 00 00 00 |+6F|+111
0000000000400070| 00 00 00 00 F0 00 03 00 |+77|+119
0000000000400078| 0B 02 01 00 00 02 00 00 |+7F|+127
0000000000400080| 00 04 00 00 00 00 00 00 |+87|+135
0000000000400088| 00 10 00 00 00 10 00 00 |+8F|+143
0000000000400090| 00 00 40 00 00 00 00 00 |+97|+151
0000000000400098| 00 10 00 00 00 02 00 00 |+9F|+159
00000000004000A0| 05 00 02 00 00 00 00 00 |+A7|+167
00000000004000A8| 05 00 02 00 00 00 00 00 |+AF|+175
00000000004000B0| 00 40 00 00 00 02 00 00 |+B7|+183
00000000004000B8| 6D DD 00 00 03 00 00 00 |+BF|+191
00000000004000C0| 00 00 10 00 00 00 00 00 |+C7|+199
```

Another example is in dealing with instruction encoding. Using CPU2.0's **opcode** and **opsize** routines, one can see the encoding produced by particular assemblers. Example below shows what's the encoding for instruction **add eax,1** produced by NASM with the default optimization option. On Win64, using NASM and GCC64;

```
;------------------------
; nasm -f win64 this.asm
; gcc -m64 this.obj cpu2.obj -s -o this.exe
;------------------------
        global main

        ;import this from cpu2.dll/cpu2.obj
        extern opcode


a:
        add eax,1
b:

        section .text
main:
        sub     rsp,40

        mov     rdx,b
        mov     rcx,a
        call    opcode

        add     rsp,40
        ret
```

Output

```
01 C0 83
```

From the output one can see that NASM produces 3-byte opcodes for such instruction. Now what if you compiled with optimization off? Lets see the encoding produced by NASM with optimization off (**-O0**) switch for the same instruction

```
;------------------------
; nasm -f win64 this.asm -O0
; gcc -m64 this.obj cpu2.obj -s -o this.exe
;------------------------
        global main

        ;import this from cpu2.dll/cpu2.obj
        extern opcode


a:
        add eax,1
b:

        section .text
main:
        sub     rsp,40

        mov     rdx,b
        mov     rcx,a
        call    opcode

        add     rsp,40
        ret
```

Output

```
00 00 00 01 05
```

Now you see NASM produces 5-byte instructions instead of 3. This is one way how one can see, on-the-fly, how an optimizing assembler such as NASM implements **multi-pass assembling** as one of its core features in order to reduce code size.

**Examples: C/C++ access**

Another feature of CPU2.0 is accessibility from high-level languages such as C/C++. This is due to ABI-compliancy of CPU2.0. When accessed this way, CPU2.0 routines will reveal vital information of C/C++ internal structures and the CPU state at points of insertions. One will just have to supply the correct argument types to the designated parameters and capture the correct return values. Note that not all CPU2.0 are suitable for high-level access due to their nature.

Example below demonstrates the power of CPU2.0 when accessed via C on Win64

```
// Example: Accessing CPU2.0 Library from C
// Binaries (cpu2.o,cpu2.obj) should reside in working folder
// or in searchable PATH of your own choosing

//Compile>> gcc -m64 this.c cpu2.obj -s -o this.exe (For Win64)
//Compile>> gcc -m64 this.c cpu2.o -s -o this (For Linux64)

#include <stdio.h>
#include <math.h>

//Import these from CPU2.0
extern void dumpreg();
extern void dumpxmmf(unsigned long long);
extern void stackview(unsigned long long);
extern void fpdinfo(double);
extern void fpfinfo(float);

int main()
{

    float x=56.456;
    double y=0.454,w=1.234;
    double z=0.0;

    //Display 5 items of stack of main();
    stackview(5);
    putchar('\n');

    //Display register state of main();
    dumpreg();
    putchar('\n');

    //Display IEEE-754 Floating point Double format
    fpdinfo(y);
    putchar('\n');

    //Display IEEE-754 Floating point Single format
    fpfinfo(x);
    putchar('\n');

    //See what is returned by C's POW(), via XMM registers
    z=pow(w,y);
    dumpxmmf(9);  //View as (formatted) packed doubles

    return 0;
}
```

Producing this power output

```
0000000000000001 |000000000061FE20
00000000004017B5 |000000000061FE18
FFFFFFFFFFFFFFFF |000000000061FE10
0000000000000001 |000000000061FE08
00000000001B1330 |000000000061FE00

RAX|000000000000000A RCX|00000000FFFFFFFF RDX|0000000000000001
RBX|0000000000000001 R8 |00007FFBD6985920 R9 |000000000061E2A0
RDI|00000000001B1330 RSI|0000000000000006 R10|0000000000000000
R11|0000000000000246 R12|0000000000000001 R13|0000000000000008
R14|0000000000000000 R15|0000000000000000 RBP|000000000061FE50
RSP|000000000061FE00 RIP|0000000000401601 [UHEX]

0.454 = 3FDD0E5604189375
0.01111111101.1101000011100101011000000100000110001001001101110101
S.EXPONT   +1.MANT
SIGN: 0
EXP : -2 (1021 - 1023)
MANT: 1.8159999999999999

56.456 = 4261D2F2
0.10000100.11000011101001011110010
S.EXPONT 1.MANT
SIGN: 0
EXP : +5 (+132 - 127)
MANT: 1.76425

 xmm0:                    +0.0|     +1.100163120495035|
 xmm1:                    +0.0|                +0.454|
 xmm2:                    +0.0|                  +0.0|
 xmm3:                    +0.0|                  +0.0|
 xmm4:                    +0.0|                  +0.0|
 xmm5:                    +0.0|                  +0.0|
 xmm6:                    +0.0|                  +0.0|
 xmm7:                    +0.0|                  +0.0|
 xmm8:                    +0.0|                  +0.0|
 xmm9:                    +0.0|                  +0.0|
xmm10:                    +0.0|                  +0.0|
xmm11:                    +0.0|                  +0.0|
xmm12:                    +0.0|                  +0.0|
xmm13:                    +0.0|                  +0.0|
xmm14:                    +0.0|                  +0.0|
xmm15:                    +0.0|                  +0.0|
```

The same effect can be achiveved via C++, using similar code setting as shown below

```
// Example: Accessing CPU2.0 Library from C++
// Binaries (cpu2.o,cpu2.obj) should reside in working folder
// or in searchable PATH of your own choosing

//Compile>> g++ -m64 this.cpp cpu2.obj -s -o this.exe (For Win64)
//Compile>> g++ -m64 this.cpp cpu2.o -s -o this (For Linux64)

#include <iostream>
#include <cmath>

//Import these from CPU2.0
extern "C"
{
    void dumpreg();
    void dumpxmmf(unsigned long long);
    void stackview(unsigned long long);
    void fpdinfo(double);
    void fpfinfo(float);
}
```

```
int main()
{
    float x=56.456;
    double y=0.454,w=1.234;
    double z=0.0;

    //Display 5 items of stack of main();
    stackview(5);
    putchar('\n');

    //Display register state of main();
    dumpreg();
    putchar('\n');

    //Display IEEE-754 Floating point Double format
    fpdinfo(y);
    putchar('\n');

    //Display IEEE-754 Floating point Single format
    fpfinfo(x);
    putchar('\n');

    //See what is returned by C's POW(), via XMM registers
    z=pow(w,y);
    dumpxmmf(9);  //View as (formatted) packed doubles

    return 0;
}
```

This will become handy if you want to know what's happening under the hood of your C/C++ programs, particularly when you deal with lots of **intrinsics** and floating-point data. CPU2.0 is providing you with low-level view of your high-level programs.

# CPU86

CPU86 is the 32-bit version of CPU2.0. This library is to be used in 32-bit assembly programming on both Win32 and Linux32 platforms in either protected mode or compatiblity mode. This library offers almost the same services / functionalities as its 64-bit counterpart. Just like CPU2.0, CPU86 works in console environment only.

To guarantee cross-platform functionality, CPU86 observes **C calling convention** or **CDECL**. That means this library behaves the same on both platforms, thus, you do not need to use different calling conventions and can use the same documentation for reference on either platform. Note that while CPU86 uses CDECL, it does not use any C library and not necessarily linked with GCC. CDECL is chosen solely for the purpose of the CPU86 API.

## The Library

CPU86 offers 4 type of binaries;

5) **.obj** - Win32 .obj format for static linking
6) **.dll** - Win32 DLL shared library
7) **.o** - Linux32 .o format (ELF) for static linking
8) **.so** - Linux32 .so shared library

The documentation of the API references and routine listing is included in **cpu2api**.

## How to Use CPU86

To use any of the CPU86 routines, use call instruction followed by the name of the routine you wished to use. Supply arguments via push instruction for routines that take arguments.

CDECL uses the stack as the placeholders or parameters for arguments. The default parameter size is four bytes. Arguments must be supplied prior to calling CPU86 routines that require arguments. All arguments are pushed on the stack, except for **prnstreg** routine (which uses EAX register).

### Caller Cleanup

The caller is responsible in unwinding the stack by the number of arguments pushed or used in function calls. For example, from the cpu2api, “prnflt” routine takes one Real4 argument. Upon exit, the caller must restore the stack exactly like so;

```
push    __float__(34.312)       ;NASM syntax. A Real4 value
call    prnflt
add     esp,4                   ;<-- stack unwinding
```

The “4” is added to ESP because the size of a single “push” is four bytes. For calling “prndbl” routine which requires exactly two arguments;

```
myValue dq -189.102934          ;A double or Real8 value
...
...
push    dword[myValue+4]        ;Upper DWORD
push    dword[myValue]          ;Lower DWORD
call    prndbl
add     esp,4*2                 ;Restore 8 bytes due to two pushes
```

or it can be simple restored using “add esp,8” instruction which gives similar unwinding effect as “4*2”. For three arguments, use “add esp,12” and so on.

**Arguments**
All arguments are pushed on the stack. Arguments supplied must be in correct:

i) Type & Format
ii) Size
iii) Sequence
iv) Numbers / Count

Refer to the **cpu2api** for references on parameters and arguments.

**Argument Types**
Argument type can be integer, floating-point or pointers. Integer and floating-point can come from immediate values or memory (variables). For example, using a dword immediate as argument

```
push    3456    ;pushing an immediate decimal
call    prnintu ;Display unsigned integer
add     esp,4   ;Stack unwinding
```

Or it can come from a variable

```
myVal dd 3456
...
push    dword[myVal]
call    prnintu
add     esp,4
```

Or from a register

```
mov     eax,3456
...
push    eax
call    prnintu
add     esp,4
```

Another examples;

```
push    'C'
call    prnchr
add     esp,4
```

Arguments can also be some floating-point arguments;

```
push    -56.567         ;FASM syntax. A Real4 immediate
call    prnflt          ;Display a float
add     esp,4
```

Except for float-specific (real4) routines, all other routines taking floating-point arguments default to double-precision.

**Argument Size**
CPU86 routines expect the callers to supply the correct argument size, failing of which may cause undefined behavior, bugs or logical errors. Due to the default parameter size (4 bytes), all arguments are confined to dword-size operands. Please note that parameter size is not necessarily the same as the argument size.

While the parameter size defaults to 4 bytes, argument size can be smaller, depending on the routines requirements. For example;

```
push    'C'     ;Parameter size=4 bytes, argument size=1 byte
call    prnchr  ;Expects a byte argument
add     esp,4   ;Restore 4 byte of parameter.
```

Use valid quantifier or size-override for smaller arguments. The example below uses “dword” quantifier against a byte-sized argument.

```
myChar db 'A'                   ;A byte character
...
push    dword[myChar]           ;Observe ”dword” keyword
call    chr_tolower             ;Convert to lower-case. Returns in AL
add     esp,4
```

Another example;

```
mov     eax,'D'         ;Copy a byte immediate to eax
...
push    eax             ;cpu86 will take care of the byte in EAX
;push   al              ;Wrong way.
call    prnchr
add     esp,4
```

Arguments can be larger than the size of the parameter. For quadword (8 bytes) arguments, they must be separated and pushed into two consecutive DWORDS. The first argument pushes the lower dword, and the next argument pushes the upper dword. This holds true for both double-precision and quadword arguments.

```
myQuad dq 19282333334       ;a 64-bit variable
myDouble dq 78.56           ;a double-precision
...
push    [myQuad+4]          ;Upper DWORD
push    [myQuad]            ;Lower DWORD
call    prnintqu            ;Display 64-bit unsigned integer
add     esp,8               ;Stack unwinding
...
push    dword ptr myDouble+4  ;MASM syntax
push    dword ptr myDouble
call    rad2deg             ;Convert 78.56 radian to degree
add     esp,8
```

Arguments larger than a quadword is passed by reference or pointer. This is particularly true for string and array arguments that are more than four bytes in size.

```
myVal dt 7820334.220123     ;A real10 (10-byte variable)
msg db “hello CPU!”,0ah,0   ;a C-string
...
push msg            ;passed a pointer / address of argument instead
;push offset msg    ;MASM-syntax
```

```
call prnstrz            ;Display 0-ended string
add esp,4

push myVal              ;Passed the address of the variable.
call prndblx            ;Display a Real10
add esp,4
```

**Argument Sequence**
The caller must observe the correct argument sequence or else a routine get the wrong data in doing its task. **cpu2api** uses *push1* for first argument and *push-n* for the next subsequent arguments. For example, routine “int2str” converts an integer to any base specified by the third argument among other arguments. From the API documentation;

```
int2str(4)
Convert integer to 0-ended string
push4| Signness. 0=unsigned. 1=signed
push3| Target base (2 to 36 only)
push2| Buffer's address
push1| Value
Ret  | String in the sent buffer
Note | Arg2 size must reflect # of digits
```

For example if you want to convert a decimal integer to Base-17, then the correct way to do it is to supply the arguments exactly as specified from the header, in correct sequence, amount, types and sizes;

```
myBuffer db 100 dup(?)      ;A buffer of 100 bytes
...
push    0                   ;Unsigned
push    17                  ;Convert to Base-17
push    myBuffer            ;Target buffer
push    4532                ;The dword value to convert
call    int2str             ;call the routine
add     esp,4*4             ;Stack unwinding
```

The example below shows how argument 1 and 2 are carelessly supplied out of the sequence.

```
myBuffer db 100 dup(?)      ;A buffer of 100 bytes
...
push    0                   ;Unsigned
push    17                  ;Convert to Base-17
push    4532                ;The dword value to convert
push    myBuffer            ;Target buffer
call    int2str             ;call the routine
add     esp,4*4             ;Stack unwinding
```

In this particular case, CPU86's “int2str” routine will likely to crash due to hostile address (4532) being used as string buffer.

**Argument Counts**
Arguments must be supplied in the correct count as specified. This applies to routines that take arguments. Supplying more or less arguments than specified can result in undefined behaviors. Supplying arguments to functions that do not take arguments is just as fatal.

## Return Values

A number of routines have return values for the callers to use.

### Integer Return Values

As per CDECL specification, integer arguments are returned in EAX. For example;

```
push    '1'
call    chr_isalpha     ;is argument an alphabet. Return status in EAX
add     esp,4
call    dumpregd        ;See status EAX register
```

You can use the return values immediately for subsequent processing or save it for later use. Another example;

```
fact dd 0               ;For saving return value
...
push    5
call    factorial       ;What's the factorial of 5! Answer in EAX
add     esp,4
sub     eax,5           ;Use it immediately, or,
mov     ebx,eax         ;save / copy to EBX, or
push    eax             ;Save to stack, or
mov     [fact],eax      ;save to a variable
```

Larger integer return values (quadwords) are passed in EAX (lower dword) and EDX (upper dword). For example, “dbl2int” routine takes a double-precision argument and returns a 64-bit (quadword) integer in EAX:EDX pair.

```
myVal: dq -12873043.23          ;a double
...
push    dword[myVal+4]
push    dword[myVal]
call    dblt2int                ;Truncate a double to 64-bit integer
add     esp,8

;returns a quadword in EAX (low), EDX,(high)
;Lets print its value

push    edx     ;High DWORD
push    eax     ;Low DWORD
call    prnintq ;Display a 64-bit signed integer. Should print -12873043
add     esp,8
```

Or you could save the return values for later use in another similar size variable like so;

```
myVal: dq -12873043.23          ;a double
temp: dq 0                      ;For saving the return value
...
push    dword[myVal+4]
push    dword[myVal]
call    dblt2int                ;Truncate a double to 64-bit integer
add     esp,8

mov     dword[temp+4],edx       ;High dword in high address
mov     dword[temp],eax         ;Low dword in low address
```

**Float Return Value**
All floating-point return values are placed in ST0. Other FPU registers will be cleared. Impliedly, the FPU registers will be clobbered by calling such functions. If you need to save the FPU state, do so before calling such routines. This can be done via FXSAVE/FSAVE to save them and FXRSTOR/FRSTOR instructions to restore them back to original state.

CPU86 makes no explicit assumptions on the size of the float return values. This is because FPU registers like ST0 defaults to Real10. It's up to the subsequent routines and programmers decision whether to treat them as a real4, real8 or real10 values.

```
myDeg  dq 3.4                ;Convert this double to radian
v1     dd 0.0
v2     dq 0.0
...
push   dword[myDeg+4]        ;deg2rad takes double-precision arg
push   dword[myDeg]
call   deg2rad               ;Return radian in ST0
add    esp,8

fst    dword[v1]             ;Return value in ST0. Store as float
fst    qword[v2]             ;Return value in ST0. Store as double

push   dword[v1]
call   prnflt                ;display radian as float
add    esp,4

call   prnline

push   dword[v2+4]
push   dword[v2]
call   prndbl                ;display radian as double
add    esp,8
```

## Cautions

CPU86 is a stack-sensitive library due to its CDECL nature. Therefore it is extremely important to use the stack in a controlled, calculated and balanced manner. Always observe the stack-unwinding activity in your code to avoid unwanted behavior. On the same note, always observe exact count of the arguments supplied to a routine.

While CPU86 share many similarites in names to its 64-bit siblings, some routines may behave completely different. Do not assume. Refer to the supplied documentation for each particular library.

## Copyrights and Licence



## Disclaimer



(End of Documentation)