Sample 64-bit nasm programs

Specifically: for Intel X86-64

Specifically: for use with gcc with its libraries and gdb

Specifically: simple nasm syntax using "C" literals

Specifically: showing an equivalent "C" program

Generally, for Linux and possibly other Unix on Intel

Generally, not using 8-bit or 16-bit or 32-bit for anything

Contents

hello_64.asm first sample program

```
  The nasm source code is hello_64.asm
  The result of the assembly is hello_64.lst
  Running the program produces output hello_64.out

  This program demonstrates basic text output to a file and screen.
  Call is made to C printf

; hello_64.asm    print a string using printf
; Assemble:       nasm -f elf64 -l hello_64.lst  hello_64.asm
; Link:           gcc -m64 -o hello_64  hello_64.o
; Run:            ./hello_64 > hello_64.out
; Output:         cat hello_64.out

; Equivalent C code
; // hello.c
; #include <stdio.h>
; int main()
; {
;   char msg[] = "Hello world\n";
;   printf("%s\n",msg);
;   return 0;
; }

; Declare needed C  functions
        extern  printf          ; the C function, to be called

        section .data           ; Data section, initialized variables
msg:    db "Hello world", 0     ; C string needs 0
```

```
fmt:    db "%s", 10, 0          ; The printf format, "\n",'0'

        section .text           ; Code section.

        global main             ; the standard gcc entry point
main:                           ; the program label for the entry point
        push    rbp             ; set up stack frame, must be alligned

        mov     rdi,fmt
        mov     rsi,msg
        mov     rax,0           ; or can be  xor  rax,rax
        call    printf          ; Call C function

        pop     rbp             ; restore stack

        mov     rax,0           ; normal, no error, return value
        ret                     ; return
```

printf1_64.asm basic calling printf

The nasm source code is printf1_64.asm
The result of the assembly is printf1_64.lst
The equivalent "C" program is printf1_64.c
Running the program produces output printf1_64.out

This program demonstrates basic use of "C" library function printf.
The equivalent "C" code is shown as comments in the assembly language.

```
; printf1_64.asm   print an integer from storage and from a register
; Assemble:     nasm -f elf64 -l printf1_64.lst  printf1_64.asm
; Link:         gcc -o printf1_64  printf1_64.o
; Run:          ./printf1_64
; Output:       a=5, rax=7

; Equivalent C code
; /* printf1.c  print a long int, 64-bit, and an expression */
; #include <stdio.h>
; int main()
; {
;   long int a=5;
;   printf("a=%ld, rax=%ld\n", a, a+2);
;   return 0;
; }

; Declare external function
        extern  printf          ; the C function, to be called

        SECTION .data           ; Data section, initialized variables

        a:      dq      5       ; long int a=5;
fmt:    db "a=%ld, rax=%ld", 10, 0      ; The printf format, "\n",'0'


        SECTION .text           ; Code section.

        global main             ; the standard gcc entry point
main:                           ; the program label for the entry point
        push    rbp             ; set up stack frame

        mov     rax,[a]         ; put "a" from store into register
        add     rax,2           ; a+2  add constant 2
        mov     rdi,fmt         ; format for printf
        mov     rsi,[a]         ; first parameter for printf
        mov     rdx,rax         ; second parameter for printf
        mov     rax,0           ; no xmm registers
        call    printf          ; Call C function

        pop     rbp             ; restore stack

        mov     rax,0           ; normal, no error, return value
        ret                     ; return
```

printf2_64.asm more types with printf

```
  The nasm source code is printf2_64.asm
  The result of the assembly is printf2_64.lst
  The equivalent "C" program is printf2_64.c
  Running the program produces output printf2_64.out

  This program demonstrates general use of "C" library function  printf.
  The equivalent "C" code is shown as comments in the assembly language.
```

```
; printf2_64.asm  use "C" printf on char, string, int, long int, float, double
;
; Assemble:     nasm -f elf64 -l printf2_64.lst  printf2_64.asm
; Link:         gcc -m64 -o printf2_64  printf2_64.o
; Run:          ./printf2_64 > printf2_64.out
; Output:       cat printf2_64.out
;
; A similar "C" program   printf2_64.c
; #include <stdio.h>
; int main()
; {
;   char      char1='a';            /* sample character */
;   char      str1[]="mystring";    /* sample string */
;   int       len=9;                /* sample string */
;   int       inta1=12345678;       /* sample integer 32-bit */
;   long int  inta2=12345678900;    /* sample long integer 64-bit */
;   long int  hex1=0x123456789ABCD; /* sample hexadecimal 64-bit*/
;   float     flt1=5.327e-30;       /* sample float 32-bit */
;   double    flt2=-123.4e300;      /* sample double 64-bit*/
;
;   printf("printf2_64: flt2=%e\n", flt2);
;   printf("char1=%c, srt1=%s, len=%d\n", char1, str1, len);
;   printf("char1=%c, srt1=%s, len=%d, inta1=%d, inta2=%ld\n",
;          char1, str1, len, inta1, inta2);
;   printf("hex1=%lX, flt1=%e, flt2=%e\n", hex1, flt1, flt2);
;   return 0;
; }
        extern printf                   ; the C function to be called

        SECTION .data                   ; Data section

                                        ; format strings for printf
fmt2:   db "printf2: flt2=%e", 10, 0
fmt3:   db "char1=%c, str1=%s, len=%d", 10, 0
fmt4:   db "char1=%c, str1=%s, len=%d, inta1=%d, inta2=%ld", 10, 0
fmt5:   db "hex1=%lX, flt1=%e, flt2=%e", 10, 0

char1:  db      'a'                     ; a character
str1:   db      "mystring",0            ; a C string, "string" needs 0
len:    equ     $-str1                  ; len has value, not an address
inta1:  dd      12345678                ; integer 12345678, note dd
inta2:  dq      12345678900             ; long integer 12345678900, note dq
hex1:   dq      0x123456789ABCD         ; long hex constant, note dq
flt1:   dd      5.327e-30               ; 32-bit floating point, note dd
flt2:   dq      -123.456789e300         ; 64-bit floating point, note dq

        SECTION .bss

flttmp: resq 1                          ; 64-bit temporary for printing flt1

        SECTION .text                   ; Code section.

        global  main                    ; "C" main program
main:                                   ; label, start of main program
        push    rbp                     ; set up stack frame
        fld     dword [flt1]            ; need to convert 32-bit to 64-bit
        fstp    qword [flttmp]          ; floating load makes 80-bit,
                                        ; store as 64-bit
        mov     rdi,fmt2
        movq    xmm0, qword [flt2]
        mov     rax, 1                  ; 1 xmm register
        call    printf
```

```
        mov     rdi, fmt3               ; first arg, format
        mov     rsi, [char1]            ; second arg, char
        mov     rdx, str1               ; third arg, string
        mov     rcx, len                ; fourth arg, int
        mov     rax, 0                  ; no xmm used
        call    printf

        mov     rdi, fmt4               ; first arg, format
        mov     rsi, [char1]            ; second arg, char
        mov     rdx, str1               ; third arg, string
        mov     rcx, len                ; fourth arg, int
        mov     r8, [inta1]             ; fifth arg, inta1 32->64
        mov     r9, [inta2]             ; sixth arg, inta2
        mov     rax, 0                  ; no xmm used
        call    printf

        mov     rdi, fmt5               ; first arg, format
        mov     rsi, [hex1]             ; second arg, char
        movq    xmm0, qword [flttmp]    ; first double
        movq    xmm1, qword [flt2]      ; second double
        mov     rax, 2                  ; 2 xmm used
        call    printf

        pop     rbp                     ; restore stack
        mov     rax, 0                  ; exit code, 0=normal
        ret                             ; main returns to operating system
```

intarith_64.asm simple 64-bit integer arithmetic

The nasm source code is intarith_64.asm
The result of the assembly is intarith_64.lst
The equivalent "C" program is intarith_64.c
Running the program produces output intarith_64.out

This program demonstrates basic integer arithmetic add, subtract,
multiply and divide.
The equivalent "C" code is shown as comments in the assembly language.

```
; intarith_64.asm    show some simple C code and corresponding nasm code
;                    the nasm code is one sample, not unique
;
; compile:      nasm -f elf64 -l intarith_64.lst  intarith_64.asm
; link:         gcc -m64 -o intarith_64  intarith_64.o
; run:          ./intarith_64 > intarith_64.out
;
; the output from running intarith_64.asm and intarith.c is:
; c=5  , a=3, b=4, c=5
; c=a+b, a=3, b=4, c=7
; c=a-b, a=3, b=4, c=-1
; c=a*b, a=3, b=4, c=12
; c=c/a, a=3, b=4, c=4
;
;The file  intarith.c  is:
;  /* intarith.c */
;  #include <stdio.h>
;  int main()
;  {
;    long int a=3, b=4, c;
;    c=5;
;    printf("%s, a=%ld, b=%ld, c=%ld\n","c=5  ", a, b, c);
;    c=a+b;
;    printf("%s, a=%ld, b=%ld, c=%ld\n","c=a+b", a, b, c);
;    c=a-b;
;    printf("%s, a=%ld, b=%ld, c=%ld\n","c=a-b", a, b, c);
;    c=a*b;
;    printf("%s, a=%ld, b=%ld, c=%ld\n","c=a*b", a, b, c);
;    c=c/a;
;    printf("%s, a=%ld, b=%ld, c=%ld\n","c=c/a", a, b, c);
;    return 0;
; }
        extern printf           ; the C function to be called
```

```
%macro  pabc 1                  ; a "simple" print macro
        section .data
.str    db      %1,0            ; %1 is first actual in macro call
        section .text
        mov     rdi, fmt4       ; first arg, format
        mov     rsi, .str       ; second arg
        mov     rdx, [a]        ; third arg
        mov     rcx, [b]        ; fourth arg
        mov     r8, [c]         ; fifth arg
        mov     rax, 0          ; no xmm used
        call    printf          ; Call C function
%endmacro

        section .data           ; preset constants, writable
a:      dq      3               ; 64-bit variable a initialized to 3
b:      dq      4               ; 64-bit variable b initializes to 4
fmt4:   db "%s, a=%ld, b=%ld, c=%ld",10,0       ; format string for printf

        section .bss            ; uninitialized space
c:      resq    1               ; reserve a 64-bit word

        section .text           ; instructions, code segment
        global   main           ; for gcc standard linking
main:                           ; label
        push    rbp             ; set up stack
lit5:                           ; c=5;
        mov     rax,5           ; 5 is a literal constant
        mov     [c],rax         ; store into c
        pabc    "c=5  "         ; invoke the print macro

addb:                           ; c=a+b;
        mov     rax,[a]         ; load a
        add     rax,[b]         ; add b
        mov     [c],rax         ; store into c
        pabc    "c=a+b"         ; invoke the print macro

subb:                           ; c=a-b;
        mov     rax,[a]         ; load a
        sub     rax,[b]         ; subtract b
        mov     [c],rax         ; store into c
        pabc    "c=a-b"         ; invoke the print macro

mulb:                           ; c=a*b;
        mov     rax,[a]         ; load a (must be rax for multiply)
        imul    qword [b]       ; signed integer multiply by b
        mov     [c],rax         ; store bottom half of product into c
        pabc    "c=a*b"         ; invoke the print macro

diva:                           ; c=c/a;
        mov     rax,[c]         ; load c
        mov     rdx,0           ; load upper half of dividend with zero
        idiv    qword [a]       ; divide double register edx rax by a
        mov     [c],rax         ; store quotient into c
        pabc    "c=c/a"         ; invoke the print macro

        pop     rbp             ; pop stack
        mov     rax,0           ; exit code, 0=normal
        ret                     ; main returns to operating system
```

fltarith_64.asm simple floating point arithmetic

The nasm source code is fltarith_64.asm
The result of the assembly is fltarith_64.lst
The equivalent "C" program is fltarith_64.c
Running the program produces output fltarith_64.out

This program demonstrates basic floating point add, subtract,
multiply and divide.
The equivalent "C" code is shown as comments in the assembly language.

```
; fltarith_64.asm   show some simple C code and corresponding nasm code
```

```
;                    the nasm code is one sample, not unique
;
; compile  nasm -f elf64 -l fltarith_64.lst  fltarith_64.asm
; link     gcc -m64 -o fltarith_64  fltarith_64.o
; run      ./fltarith_64 > fltarith_64.out
;
; the output from running fltarith and fltarithc is:
; c=5.0, a=3.000000e+00, b=4.000000e+00, c=5.000000e+00
; c=a+b, a=3.000000e+00, b=4.000000e+00, c=7.000000e+00
; c=a-b, a=3.000000e+00, b=4.000000e+00, c=-1.000000e+00
; c=a*b, a=3.000000e+00, b=4.000000e+00, c=1.200000e+01
; c=c/a, a=3.000000e+00, b=4.000000e+00, c=4.000000e+00
; a=i  , a=8.000000e+00, b=1.600000e+01, c=1.600000e+01
; a<=b , a=8.000000e+00, b=1.600000e+01, c=1.600000e+01
; b==c , a=8.000000e+00, b=1.600000e+01, c=1.600000e+01
;The file  fltarith.c  is:
;  #include <stdio.h>
;  int main()
;  {
;    double a=3.0, b=4.0, c;
;    long int i=8;
;
;    c=5.0;
;    printf("%s, a=%e, b=%e, c=%e\n","c=5.0", a, b, c);
;    c=a+b;
;    printf("%s, a=%e, b=%e, c=%e\n","c=a+b", a, b, c);
;    c=a-b;
;    printf("%s, a=%e, b=%e, c=%e\n","c=a-b", a, b, c);
;    c=a*b;
;    printf("%s, a=%e, b=%e, c=%e\n","c=a*b", a, b, c);
;    c=c/a;
;    printf("%s, a=%e, b=%e, c=%e\n","c=c/a", a, b, c);
;    a=i;
;    b=a+i;
;    i=b;
;    c=i;
;    printf("%s, a=%e, b=%e, c=%e\n","c=c/a", a, b, c);
;    if(ab  ", a, b, c);
;    if(b==c)printf("%s, a=%e, b=%e, c=%e\n","b==c ", a, b, c);
;    else    printf("%s, a=%e, b=%e, c=%e\n","b!=c ", a, b, c);
;    return 0;
; }

        extern printf           ; the C function to be called

%macro  pabc 1                  ; a "simple" print macro
        section .data
.str    db      %1,0            ; %1 is macro call first actual parameter
        section .text
                                ; push onto stack backwards
        mov     rdi, fmt        ; address of format string
        mov     rsi, .str       ; string passed to macro
        movq    xmm0, qword [a] ; first floating point in fmt
        movq    xmm1, qword [b] ; second floating point
        movq    xmm2, qword [c] ; third floating point
        mov     rax, 3          ; 3 floating point arguments to printf
        call    printf          ; Call C function
%endmacro

        section .data           ; preset constants, writable
a:      dq      3.0             ; 64-bit variable a initialized to 3.0
b:      dq      4.0             ; 64-bit variable b initializes to 4.0
i:      dq      8               ; a 64 bit integer
five:   dq      5.0             ; constant 5.0
fmt:    db "%s, a=%e, b=%e, c=%e",10,0  ; format string for printf

        section .bss            ; uninitialized space
c:      resq    1               ; reserve a 64-bit word

        section .text           ; instructions, code segment
        global  main            ; for gcc standard linking
main:                           ; label

        push    rbp             ; set up stack
lit5:                           ; c=5.0;
        fld     qword [five]    ; 5.0 constant
```

```
        fstp    qword [c]       ; store into c
        pabc    "c=5.0"         ; invoke the print macro

addb:                           ; c=a+b;
        fld     qword [a]       ; load a (pushed on flt pt stack, st0)
        fadd    qword [b]       ; floating add b (to st0)
        fstp    qword [c]       ; store into c (pop flt pt stack)
        pabc    "c=a+b"         ; invoke the print macro

subb:                           ; c=a-b;
        fld     qword [a]       ; load a (pushed on flt pt stack, st0)
        fsub    qword [b]       ; floating subtract b (to st0)
        fstp    qword [c]       ; store into c (pop flt pt stack)
        pabc    "c=a-b"         ; invoke the print macro

mulb:                           ; c=a*b;
        fld     qword [a]       ; load a (pushed on flt pt stack, st0)
        fmul    qword [b]       ; floating multiply by b (to st0)
        fstp    qword [c]       ; store product into c (pop flt pt stack)
        pabc    "c=a*b"         ; invoke the print macro

diva:                           ; c=c/a;
        fld     qword [c]       ; load c (pushed on flt pt stack, st0)
        fdiv    qword [a]       ; floating divide by a (to st0)
        fstp    qword [c]       ; store quotient into c (pop flt pt stack)
        pabc    "c=c/a"         ; invoke the print macro

intflt:                         ; a=i;
        fild    dword [i]       ; load integer as floating point
        fst     qword [a]       ; store the floating point (no pop)
        fadd    st0             ; b=a+i; 'a' as 'i'  already on flt stack
        fst     qword [b]       ; store sum (no pop) 'b' still on stack
        fistp   dword [i]       ; i=b; store floating point as integer
        fild    dword [i]       ; c=i; load again from ram (redundant)
        fstp    qword [c]
        pabc    "a=i  "         ; invoke the print macro

cmpflt: fld     dword [b]       ; into st0, then pushed to st1
        fld     dword [a]       ; in st0
        fcomip  st0,st1         ; a compare b, pop a
        jg      cmpfl2
        pabc    "a<=b "
        jmp     cmpfl3
cmpfl2:
        pabc    "a>b  "
cmpfl3:
        fld     dword [c]       ; should equal [b]
        fcomip  st0,st1
        jne     cmpfl4
        pabc    "b==c "
        jmp     cmpfl5
cmpfl4:
        pabc    "b!=c "
cmpfl5:

        pop     rbp             ; pop stack
        mov     rax,0           ; exit code, 0=normal
        ret                     ; main returns to operating system
```

fib_64l.asm print 64-bit fib numbers

```
The nasm source code is fib_64l.asm
The result of the assembly is fib_64l.lst
The equivalent "C" program is fib.c
Running the program produces output fib_64l.out
The nasm source code, like C, is fib_64m.asm
The result of the assembly is fib_64m.lst
Running the program produces output fib_64m.out

Note: output may go negative when size of numbers
exceed 63-bits without sign. Wrong results with overflow.
```

This program demonstrates a loop, saving state between calls.
First, the 64-bit C program:

```
// fib.c  same as computation as fib_64m.asm similar fib_64l.asm
#include <stdio.h>
int main(int argc, char *argv[])
{
  long int c = 95;  // loop counter
  long int a = 1;   // current number, becomes next
  long int b = 2;   // next number, becomes sum a+b
  long int d;       // temp

  for(c=c; c!=0; c--)
  {
    printf("%21ld\n",a);
    d = a;
    a = b;
    b = d+b;
  }
  return 0;
}
```

Now, the first 64-bit assembly language implementation

```
; fib_64l.asm  using 64 bit registers to implement fib.c
        global main
        extern printf

        section .data
format: db '%15ld', 10, 0
title:  db 'fibinachi numbers', 10, 0

        section .text
main:
        push rbp                ; set up stack
        mov rdi, title          ; arg 1 is a pointer
        mov rax, 0              ; no vector registers in use
        call printf

        mov rcx, 95             ; rcx will countdown from 52 to 0
        mov rax, 1              ; rax will hold the current number
        mov rbx, 2              ; rbx will hold the next number
print:
        ;  We need to call printf, but we are using rax, rbx, and rcx.
        ;  printf may destroy rax and rcx so we will save these before
        ;  the call and restore them afterwards.
        push rax                ; 32-bit stack operands are not encodable
        push rcx                ; in 64-bit mode, so we use the "r" names
        mov rdi, format         ; arg 1 is a pointer
        mov rsi, rax            ; arg 2 is the current number
        mov eax, 0              ; no vector registers in use
        call printf
        pop rcx
        pop rax
        mov rdx, rax            ; save the current number
        mov rax, rbx            ; next number is now current
        add rbx, rdx            ; get the new next number
        dec rcx                 ; count down
        jnz print               ; if not done counting, do some more

        pop rbp                 ; restore stack
        mov rax, 0              ; normal exit
        ret
```

Now an implementation closer to C, storing variables

```
; fib_64m.asm  using 64 bit memory more like C code
; // fib.c  same as computation as fib_64m.asm
; #include
; int main(int argc, char *argv[])
```

```
; {
;   long int c = 95;  // loop counter
;   long int a = 1;   // current number, becomes next
;   long int b = 2;   // next number, becomes sum a+b
;   long int d;       // temp
;   printf("fibinachi numbers\n");
;   for(c=c; c!=0; c--)
;   {
;     printf("%21ld\n",a);
;     d = a;
;     a = b;
;     b = d+b;
;   }
; }
        global main
        extern printf

        section .bss
d:      resq    1               ; temp  unused, kept in register rdx

        section .data
c:      dq      95              ; loop counter
a:      dq      1               ; current number, becomes next
b:      dq      2               ; next number, becomes sum a+b


format: db '%15ld', 10, 0
title:  db 'fibinachi numbers', 10, 0

        section .text
main:
        push rbp                ; set up stack
        mov rdi, title          ; arg 1 is a pointer
        mov rax, 0              ; no vector registers in use
        call printf

print:
        ;  We need to call printf, but we are using rax, rbx, and rcx.
        mov rdi, format         ; arg 1 is a pointer
        mov rsi,[a]             ; arg 2 is the current number
        mov rax, 0              ; no vector registers in use
        call printf

        mov rdx,[a]             ; save the current number, in register
        mov rbx,[b]             ;
        mov [a],rbx             ; next number is now current, in ram
        add rbx, rdx            ; get the new next number
        mov [b],rbx             ; store in ram
        mov rcx,[c]             ; get loop count
        dec rcx                 ; count down
        mov [c],rcx             ; save in ram
        jnz print               ; if not done counting, do some more

        pop rbp                 ; restore stack
        mov rax, 0              ; normal exit
        ret                     ; return to operating system
```

loopint_64.asm simple loop

The nasm source code is loopint_64.asm
The result of the assembly is loopint_64.lst
The equivalent "C" program is loopint_64.c
Running the program produces output loopint_64.out

This program demonstrates basic loop assembly language

```
; loopint_64.asm  code loopint.c for nasm
; /* loopint_64.c a very simple loop that will be coded for nasm */
; #include <stdio.h>
; int main()
; {
;   long int dd1[100]; // 100 could be 3 gigabytes
;   long int i;        // must be long for more than 2 gigabytes
;   dd1[0]=5; /* be sure loop stays 1..98 */
```

```
;   dd1[99]=9;
;   for(i=1; i<99; i++) dd1[i]=7;
;   printf("dd1[0]=%ld, dd1[1]=%ld, dd1[98]=%ld, dd1[99]=%ld\n",
;           dd1[0], dd1[1], dd1[98],dd1[99]);
;   return 0;
;}
; execution output is dd1[0]=5, dd1[1]=7, dd1[98]=7, dd1[99]=9

        section .bss
dd1:    resq    100                     ; reserve 100 long int
i:      resq    1                       ; actually unused, kept in register

        section .data                   ; Data section, initialized variables
fmt:    db "dd1[0]=%ld, dd1[1]=%ld, dd1[98]=%ld, dd1[99]=%ld",10,0

        extern  printf                  ; the C function, to be called

        section .text
        global  main
main:   push    rbp                     ; set up stack

        mov     qword [dd1],5           ; dd1[0]=5;  memory to memory
        mov     qword [dd1+99*8],9      ; dd1[99]=9; indexed 99 qword

        mov     rdi, 1*8                ; i=1; index, will move by 8 bytes
loop1:  mov     qword [dd1+rdi],7       ; dd1[i]=7;
        add     rdi, 8                  ; i++;  8 bytes
        cmp     rdi, 8*99               ; i<99
        jne     loop1                   ; loop until incremented i=99

        mov     rdi, fmt                ; pass address of format
        mov     rsi, qword [dd1]        ; dd1[0]   first list parameter
        mov     rdx, qword [dd1+1*8]    ; dd1[1]   second list parameter
        mov     rcx, qword [dd1+98*8]   ; dd1[98]  third list parameter
        mov     r8,  qword [dd1+99*8]   ; dd1[99]  fourth list parameter
        mov     rax, 0                  ; no xmm used
        call    printf                  ; Call C function

        pop     rbp                     ; restore stack
        mov     rax,0                   ; normal, no error, return value
        ret                             ; return 0;
```

testreg_64.asm use rax, eax, ax, ah, al

The nasm source code is testreg_64.asm
The result of the assembly is testreg_64.lst

This program demonstrates basic use of registers in assembly language

```
; testreg_64.asm   test what register names can be used
;
; compile:      nasm -f elf64 -l testreg_64.lst  testasm_64.asm
; link:         gcc -o testreg_64  testreg_64.o
; run:          ./testreg  # may get segfault or other error
;
        section .data           ; preset constants, writable
aa8:    db      8               ; 8-bit
aa16:   dw      16              ; 16-bit
aa32:   dd      32              ; 32-bit
aa64:   dq      64              ; 64-bit

        section .bss
bb16:   resw    16

        section .rodata
cc16:   db      8

        section .text           ; instructions, code segment
        global   main           ; for gcc standard linking
main:                           ; label
        push    rbp             ; set up stack
```

```
        mov     rax,[aa64]      ; five registers in RAX
        mov     eax,[aa32]      ; four registers in EAX
        mov     ax,[aa16]
        mov     ah,[aa8]
        mov     al,[aa8]

        mov     RAX,[aa64]      ; upper case register names
        mov     EAX,[aa32]
        mov     AX,[aa16]
        mov     AH,[aa8]
        mov     AL,[aa8]

        mov     rbx,[aa64]      ; five registers in RBX
        mov     ebx,[aa32]      ; four registers in EBX
        mov     bx,[aa16]
        mov     bh,[aa8]
        mov     bl,[aa8]

        mov     rcx,[aa64]      ; five registers in RCX
        mov     ecx,[aa32]      ; four registers in ECX
        mov     cx,[aa16]
        mov     ch,[aa8]
        mov     cl,[aa8]

        mov     rdx,[aa64]      ; five registers in RDX
        mov     edx,[aa32]      ; four registers in EDX
        mov     dx,[aa16]
        mov     dh,[aa8]
        mov     dl,[aa8]

        mov     rsi,[aa64]      ; three registers in RSI
        mov     esi,[aa32]      ; two registers in ESI
        mov     si,[aa16]

        mov     rdi,[aa64]      ; three registers in RDI
        mov     edi,[aa32]      ; two registers in EDI
        mov     di,[aa16]

        mov     rbp,[aa64]      ; three registers in RBP
        mov     ebp,[aa32]      ; two registers in EBP
        mov     bp,[aa16]

        mov     r8,[aa64]       ; just 64-bit r8 .. r15

        movq    xmm0, qword [aa64] ; xmm registers special

        fld     qword [aa64]    ; floating point special

;       POPF                    ; no "mov" on EFLAGS register
;       PUSHF                   ; 32 bits on 386 and above

;       mov     rsp,[aa64]      ; three registers in RSP
;       mov     esp,[aa32]      ; two registers in ESP
;       mov     sp,[aa16]       ; don't mess with stack

        pop     rbp
        mov     rax,0           ; exit code, 0=normal
        ret                     ; main returns to operating system

; end testreg_64.asm
```

shift_64.asm shifting

```
  The nasm source code is shift_64.asm
  The result of the assembly is shift_64.lst
  Running the program produces output shift_64.out

  This program demonstrates basic shifting in assembly language

; shift_64.asm    the nasm code is one sample, not unique
;
; compile:      nasm -f elf64 -l shift_64.lst  shift_64.asm
```

```
; link:         gcc -o shift_64  shift_64.o
; run:          ./shift_64 > shift_64.out
;
; the output from running shift.asm (zero filled) is:
; shl rax,4, old rax=ABCDEF0987654321, new rax=BCDEF09876543210,
; shl rax,8, old rax=ABCDEF0987654321, new rax=CDEF098765432100,
; shr rax,4, old rax=ABCDEF0987654321, new rax= ABCDEF098765432,
; sal rax,8, old rax=ABCDEF0987654321, new rax=CDEF098765432100,
; sar rax,4, old rax=ABCDEF0987654321, new rax=FABCDEF098765432,
; rol rax,4, old rax=ABCDEF0987654321, new rax=BCDEF0987654321A,
; ror rax,4, old rax=ABCDEF0987654321, new rax=1ABCDEF098765432,
; shld rdx,rax,8, old rdx:rax=0,ABCDEF0987654321,
;                 new rax=ABCDEF0987654321 rdx=              AB,
; shl rax,8     , old rdx:rax=0,ABCDEF0987654321,
;                 new rax=CDEF098765432100 rdx=              AB,
; shrd rdx,rax,8, old rdx:rax=0,ABCDEF0987654321,
;                 new rax=ABCDEF0987654321 rdx=2100000000000000,
; shr rax,8     , old rdx:rax=0,ABCDEF0987654321,
;                 new rax=  ABCDEF09876543 rdx=2100000000000000,

        extern printf           ; the C function to be called

%macro  prt     1               ; old and new rax
        section .data
.str    db      %1,0            ; %1 is which shift string
        section .text
        mov     rdi, fmt        ; address of format string
        mov     rsi, .str       ; callers string
        mov     rdx,rax         ; new value
        mov     rax, 0          ; no floating point
        call    printf          ; Call C function
%endmacro

%macro  prt2    1               ; old and new rax,rdx
        section .data
.str    db      %1,0            ; %1 is which shift
        section .text
        mov     rdi, fmt2       ; address of format string
        mov     rsi, .str       ; callers string
        mov     rcx, rdx        ; new rdx befor next because used
        mov     rdx, rax        ; new rax
        mov     rax, 0          ; no floating point
        call    printf          ; Call C function
%endmacro

         section .bss
raxsave: resq   1               ; save rax while calling a function
rdxsave: resq   1               ; save rdx while calling a function

        section .data           ; preset constants, writable
b64:    dq      0xABCDEF0987654321      ; data to shift
fmt:    db "%s, old rax=ABCDEF0987654321, new rax=%16lX, ",10,0 ; format string
fmt2:   db "%s, old rdx:rax=0,ABCDEF0987654321,",10,"                new rax=%16lX rdx=%16lX, ",10,0

        section .text           ; instructions, code segment
        global   main           ; for gcc standard linking
main:   push    rbp             ; set up stack

shl1:   mov     rax, [b64]      ; data to shift
        shl     rax, 4          ; shift rax 4 bits, one hex position left
        prt     "shl rax,4 "    ; invoke the print macro

shl4:   mov     rax, [b64]      ; data to shift
        shl     rax,8           ; shift rax 8 bits. two hex positions left
        prt     "shl rax,8 "    ; invoke the print macro

shr4:   mov     rax, [b64]      ; data to shift
        shr     rax,4           ; shift
        prt     "shr rax,4 "    ; invoke the print macro

sal4:   mov     rax, [b64]      ; data to shift
        sal     rax,8           ; shift
        prt     "sal rax,8 "    ; invoke the print macro

sar4:   mov     rax, [b64]      ; data to shift
        sar     rax,4           ; shift
```

```
        prt     "sar rax,4 "    ; invoke the print macro

rol4:   mov     rax, [b64]      ; data to shift
        rol     rax,4           ; shift
        prt     "rol rax,4 "    ; invoke the print macro

ror4:   mov     rax, [b64]      ; data to shift
        ror     rax,4           ; shift
        prt     "ror rax,4 "    ; invoke the print macro

shld4:  mov     rax, [b64]      ; data to shift
        mov     rdx,0           ; register receiving bits
        shld    rdx,rax,8       ; shift
        mov     [raxsave],rax   ; save, destroyed by function
        mov     [rdxsave],rdx   ; save, destroyed by function
        prt2    "shld rdx,rax,8"; invoke the print macro

shla:   mov     rax,[raxsave]   ; restore, destroyed by function
        mov     rdx,[rdxsave]   ; restore, destroyed by function
        shl     rax,8           ; finish double shift, both registers
        prt2    "shl rax,8     "; invoke the print macro

shrd4:  mov     rax, [b64]      ; data to shift
        mov     rdx,0           ; register receiving bits
        shrd    rdx,rax,8       ; shift
        mov     [raxsave],rax   ; save, destroyed by function
        mov     [rdxsave],rdx   ; save, destroyed by function
        prt2    "shrd rdx,rax,8"; invoke the print macro

shra:   mov     rax,[raxsave]   ; restore, destroyed by function
        mov     rdx,[rdxsave]   ; restore, destroyed by function
        shr     rax,8           ; finish double shift, both registers
        prt2    "shr rax,8     "; invoke the print macro

        pop     rbp             ; restore stack
        mov     rax,0           ; exit code, 0=normal
        ret                     ; main returns to operating system
```

ifint_64.asm if then else

```
  The nasm source code is ifint_64.asm
  The result of the assembly is ifint_64.lst
  The equivalent "C" program is ifint_64.c
  Running the program produces output ifint_64.out

  This program demonstrates basic if then else in assembly language
```

```
; ifint_64.asm  code ifint_64.c for nasm
; /* ifint_64.c an 'if' statement that will be coded for nasm */
; #include <stdio.h>
; int main()
; {
;   long int a=1;
;   long int b=2;
;   long int c=3;
;   if(a<b)
;     printf("true a < b \n");
;   else
;     printf("wrong on a < b \n");
;   if(b>c)
;     printf("wrong on b > c \n");
;   else
;     printf("false b > c \n");
;   return 0;
;}
; result of executing both "C" and assembly is:
; true a < b
; false b > c

        global  main            ; define for linker
        extern  printf          ; tell linker we need this C function
        section .data           ; Data section, initialized variables
```

```
a:      dq 1
b:      dq 2
c:      dq 3
fmt1:   db "true a < b ",10,0
fmt2:   db "wrong on a < b ",10,0
fmt3:   db "wrong on b > c ",10,0
fmt4:   db "false b > c ",10,0

        section .text
main:   push    rbp             ; set up stack
        mov     rax,[a]         ; a
        cmp     rax,[b]         ; compare a to b
        jge     false1          ; choose jump to false part
        ; a < b sign is set
        mov     rdi, fmt1       ; printf("true a < b \n");
        call    printf
        jmp     exit1           ; jump over false part
false1: ;  a < b is false
        mov     rdi, fmt2       ; printf("wrong on a < b \n");
        call    printf
exit1:                          ; finished 'if' statement

        mov     rax,[b]         ; b
        cmp     rax,[c]         ; compare b to c
        jle     false2          ; choose jump to false part
        ; b > c sign is not set
        mov     rdi, fmt3       ; printf("wrong on b > c \n");
        call    printf
        jmp     exit2           ; jump over false part
false2: ;  b > c is false
        mov     rdi, fmt4       ; printf("false b > c \n");
        call    printf
exit2:                          ; finished 'if' statement

        pop     rbp             ; restore stack
        mov     rax,0           ; normal, no error, return value
        ret                     ; return 0;
```

intlogic_64.asm bit logic, and, or

```
  The nasm source code is intlogic_64.asm
  The result of the assembly is intlogic_64.lst
  The equivalent "C" program is intlogic_64.c
  Running the program produces output intlogic_64.out

  This program demonstrates basic and, or, xor, not in assembly language
```

```
; intlogic_64.asm    show some simple C code and corresponding nasm code
;                    the nasm code is one sample, not unique
;
; compile:      nasm -f elf64 -l intlogic_64.lst  intlogic_64.asm
; link:         gcc -m64 -o intlogic_64  intlogic_64.o
; run:          ./intlogic_64 > intlogic_64.out
;
; the output from running intlogic_64.asm and intlogic.c is
; c=5  , a=3, b=5, c=15
; c=a&b, a=3, b=5, c=1
; c=a|b, a=3, b=5, c=7
; c=a^b, a=3, b=5, c=6
; c=~a , a=3, b=5, c=-4
;
;The file  intlogic_64.c  is:
;  #include <stdio.h>
;  int main()
;  {
;    long int a=3, b=5, c;
;
;    c=15;
;    printf("%s, a=%ld, b=%ld, c=%ld\n","c=5  ", a, b, c);
;    c=a&b; /* and */
;    printf("%s, a=%ld, b=%ld, c=%ld\n","c=a&b", a, b, c);
;    c=a|b; /* or */
;    printf("%s, a=%ld, b=%ld, c=%ld\n","c=a|b", a, b, c);
;    c=a^b; /* xor */
```

```
;    printf("%s, a=%ld, b=%ld, c=%ld\n","c=a^b", a, b, c);
;    c=~a;  /* not */
;    printf("%s, a=%ld, b=%ld, c=%d\n","c=~a", a, b, c);
;    return 0;
; }

        extern printf           ; the C function to be called

%macro  pabc 1                  ; a "simple" print macro
        section .data
.str    db      %1,0            ; %1 is first actual in macro call
        section .text
        mov     rdi, fmt        ; address of format string
        mov     rsi, .str       ; users string
        mov     rdx, [a]        ; long int a
        mov     rcx, [b]        ; long int b
        mov     r8, [c]         ; long int c
        mov     rax, 0          ; no xmm used
        call    printf          ; Call C function
%endmacro

        section .data           ; preset constants, writable
a:      dq      3               ; 64-bit variable a initialized to 3
b:      dq      5               ; 64-bit variable b initializes to 4
fmt:    db "%s, a=%ld, b=%ld, c=%ld",10,0 ; format string for printf

        section .bss            ; unitialized space
c:      resq    1               ; reserve a 64-bit word

        section .text           ; instructions, code segment
        global   main           ; for gcc standard linking
main:                           ; label
        push    rbp             ; set up stack

lit5:                           ; c=5;
        mov     rax,15          ; 5 is a literal constant
        mov     [c],rax         ; store into c
        pabc    "c=5  "         ; invoke the print macro

andb:                           ; c=a&b;
        mov     rax,[a]         ; load a
        and     rax,[b]         ; and with b
        mov     [c],rax         ; store into c
        pabc    "c=a&b"         ; invoke the print macro

orw:                            ; c=a-b;
        mov     rax,[a]         ; load a
        or      rax,[b]         ; logical or with b
        mov     [c],rax         ; store into c
        pabc    "c=a|b"         ; invoke the print macro

xorw:                           ; c=a^b;
        mov     rax,[a]         ; load a
        xor     rax,[b]         ; exclusive or with b
        mov     [c],rax         ; store result in c
        pabc    "c=a^b"         ; invoke the print macro

notw:                           ; c=~a;
        mov     rax,[a]         ; load c
        not     rax             ; not, complement
        mov     [c],rax         ; store result into c
        pabc    "c=~a "         ; invoke the print macro

        pop     rbp             ; restore stack
        mov     rax,0           ; exit code, 0=normal
        ret                     ; main returns to operating system
```

horner_64.asm Horner polynomial evaluation

```
The nasm source code is horner_64.asm
The result of the assembly is horner_64.lst
The equivalent "C" program is horner_64.c
Running the program produces output horner_64.out
```

This program demonstrates Horner method of evaluating polynomials,
using both integer and floating point and indexing an array.

```
; horner_64.asm  Horners method of evaluating polynomials
;
; given a polynomial  Y = a_n X^n + a_n-1 X^n-1 + ... a_1 X + a_0
; a_n is the coefficient 'a' with subscript n. X^n is X to nth power
; compute y_1 = a_n * X + a_n-1
; compute y_2 = y_1 * X + a_n-2
; compute y_i = y_i-1 * X + a_n-i   i=3..n
; thus    y_n = Y = value of polynomial
;
; in assembly language:
;   load some register with a_n, multiply by X
;   add a_n-1, multiply by X, add a_n-2, multiply by X, ...
;   finishing with the add  a_0
;
; output from execution:
; a  6319
; aa 6319
; af 6.319000e+03

        extern  printf
        section .data
        global  main

        section .data
fmta:   db      "a  %ld",10,0
fmtaa:  db      "aa %ld",10,0
fmtflt: db      "af %e",10,0

        section .text
main:   push    rbp             ; set up stack

; evaluate an integer polynomial, X=7, using a count

        section .data
a:      dq      2,5,-7,22,-9    ; coefficients of polynomial, a_n first
X:      dq      7               ; X = 7
                                ; n=4, 8 bytes per coefficient
        section .text
        mov     rax,[a]         ; accumulate value here, get coefficient a_n
        mov     rdi,1           ; subscript initialization
        mov     rcx,4           ; loop iteration count initialization, n
h3loop: imul    rax,[X]         ; * X     (ignore edx)
        add     rax,[a+8*rdi]   ; + a_n-i
        inc     rdi             ; increment subscript
        loop    h3loop          ; decrement rcx, jump on non zero

        mov     rsi, rax        ; print rax
        mov     rdi, fmta       ; format
        mov     rax, 0          ; no float
        call    printf


; evaluate an integer polynomial, X=7, using a count as index
; optimal organization of data allows a three instruction loop

        section .data
aa:     dq      -9,22,-7,5,2    ; coefficients of polynomial, a_0 first
n:      dq      4               ; n=4, 8 bytes per coefficient
        section .text
        mov     rax,[aa+4*8]    ; accumulate value here, get coefficient a_n
        mov     rcx,[n]         ; loop iteration count initialization, n
h4loop: imul    rax,[X]         ; * X     (ignore edx)
        add     rax,[aa+8*rcx-8]; + aa_n-i
        loop    h4loop          ; decrement rcx, jump on non zero

        mov     rsi, rax        ; print rax
        mov     rdi, fmtaa      ; format
        mov     rax, 0          ; no float
        call    printf

; evaluate a double floating polynomial, X=7.0, using a count as index
; optimal organization of data allows a three instruction loop
```

```
        section .data
af:     dq      -9.0,22.0,-7.0,5.0,2.0  ; coefficients of polynomial, a_0 first
XF:     dq      7.0
Y:      dq      0.0
N:      dd      4

        section .text
        mov     rcx,[N]         ; loop iteration count initialization, n
        fld     qword [af+8*rcx]; accumulate value here, get coefficient a_n
h5loop: fmul    qword [XF]      ; * XF
        fadd    qword [af+8*rcx-8] ; + aa_n-i
        loop    h5loop          ; decrement rcx, jump on non zero

        fstp    qword [Y]       ; store Y in order to print Y
        movq    xmm0, qword [Y] ; well, may just mov reg
        mov     rdi, fmtflt     ; format
        mov     rax, 1          ; one float
        call    printf

        pop     rbp             ; restore stack
        mov     rax,0           ; normal return
        ret                     ; return
```

call1_64.asm change callers array

```
  The nasm source code is call1_64.asm
  The main "C" program is test_call1_64.c
  Be safe, header file is call1_64.h
  The equivalent "C" program is call1_64.c
  Running the program produces output test_call1_64.out

  This program demonstrates passing an array to assembly language
  and the assembly language updating the array.
```

```
; call1_64.asm  a basic structure for a subroutine to be called from "C"
;
; Parameter:   long int *L
; Result: L[0]=L[0]+3  L[1]=L[1]+4

        global call1_64         ; linker must know name of subroutine

        extern  printf          ; the C function, to be called for demo

        SECTION .data           ; Data section, initialized variables
fmt1:    db "rdi=%ld, L[0]=%ld", 10, 0  ; The printf format, "\n",'0'
fmt2:    db "rdi=%ld, L[1]=%ld", 10, 0  ; The printf format, "\n",'0'

        SECTION .bss
a:      resq    1               ; temp for printing

        SECTION .text           ; Code section.

call1_64:                       ; name must appear as a nasm label
        push    rbp             ; save rbp
        mov     rbp, rsp        ; rbp is callers stack
        push    rdx             ; save registers
        push    rdi
        push    rsi

        mov     rax,rdi         ; first, only, in parameter
        mov     [a],rdi         ; save for later use

        mov     rdi,fmt1        ; format for printf debug, demo
        mov     rsi,rax         ; first parameter for printf
        mov     rdx,[rax]       ; second parameter for printf
        mov     rax,0           ; no xmm registers
        call    printf          ; Call C function

        mov     rax,[a]         ; first, only, in parameter, demo
        mov     rdi,fmt2        ; format for printf
        mov     rsi,rax         ; first parameter for printf
        mov     rdx,[rax+8]     ; second parameter for printf
        mov     rax,0           ; no xmm registers
        call    printf          ; Call C function
```

```
        mov     rax,[a]         ; add 3 to L[0]
        mov     rdx,[rax]       ; get L[0]
        add     rdx,3           ; add
        mov     [rax],rdx       ; store sum for caller

        mov     rdx,[rax+8]     ; get L[1]
        add     rdx,4           ; add
        mov     [rax+8],rdx     ; store sum for caller

        pop     rsi             ; restore registers
        pop     rdi             ; in reverse order
        pop     rdx
        mov     rsp,rbp         ; restore callers stack frame
        pop     rbp
        ret                     ; return
```

Go to top

Last updated 1/27/2015