Programming Intel i386 Assembly with NASM

Yorick Hardy

International School for Scientific Computing

NASM

NASM can be used in combinations of the following

- With C/C++ to define functions that can be used by C/C++
- To construct a program in assembly language only
- To construct a program in assembly language which calls C functions

The mechanisms to do this depend on the compiler.

We will consider the GNU Compiler Collection (GCC).

NASM: C calling conventions

Parameters are passed on the stack. Functions are called using the `call` instruction which pushes the return address on the stack before jumping to the function pointer.

Functions return to the stored address by popping the address off the stack and jumping to the return address using the `ret` instruction.

The return value is usually assumed to be in `eax`.

```
...
call zero_eax     ; push the return address on the stack
...


zero_eax:
 xor eax, eax
 ret              ; the return address is in [esp]
```

NASM: C calling conventions

When using GCC we consider `cdecl`, `stdcall` and other calling conventions.

- `cdecl`: Parameters are pushed onto the stack in right to left order before the function call, and removed in left to right order after the function call. It is the **caller's** responsibility to remove the parameters from the stack after a function call.

- `stdcall`: Parameters are pushed onto the stack in right to left order before the function call, and removed in left to right order before returning from function call *(unless the function takes a variable number of arguments)*. It is the **callee's** responsibility to remove the parameters from the stack after a function call.

- other: A combination of stack, registers and memory addresses may be used to define the calling convention. Usually, these functions cannot be called from C/C++.

NASM: `cdecl` calling convention

```
int add(int a, int b) { return a+b; }

 ...
 push b
 push a
 call add
 add  esp, 8
 ;or pop  ebx   ; remove parameter
 ;   pop  ebx   ; remove parameter
 ...

add:
 ; [esp] is the return address,
 ; [esp+4] the first parameter, etc.
 mov  eax, [esp+4]
 add  eax, [esp+8]
 ret
```

NASM: stdcall calling convention

```
int add(int a, int b) { return a+b; }

 ...
 push b
 push a
 call add
 ...

add:
 ; [esp] is the return address,
 ; [esp+4] the first parameter, etc.
 mov  eax, [esp+4]
 add  eax, [esp+8]
 push ebx            ; save ebx
 mov  ebx, [esp+4]   ; return address (after ebx)
 sub  esp, 16        ; ebx, ret addr, 1st param, 2nd param
 push ebx            ; restore return address
 mov  ebx, [esp-12]  ; 16-4 for return address
 ret
```

NASM: `cdecl` calling convention

We will consider the `cdecl` calling convention.

To avoid the pointer arithmetic we usually follow the convention

1. On entry to the function the return address is in `[esp]`, the first parameter in `[esp+4]`, etc.

2. Save `ebp`:
   `push ebp`

3. Save the current stack pointer in `ebp`:
   `mov ebp, esp`

4. Now the saved `ebp` is in `[ebp]`, the return address is in `[ebp+4]`, the first parameter in `[esp+8]`, etc.

5. Before `ret`, remember to restore `ebp`:
   `pop ebp`

NASM: calling assembler from C

```
/* fact.c */

#include <stdio.h>

extern int factorial(int);

int main(void)
{
 printf("%d\n", factorial(5));
 return 0;
}
```

NASM: calling assembler from C++

```
// fact.cpp
#include <iostream>

using namespace std;

extern "C" int factorial(int);

int main(void)
{
 cout << factorial(5) << endl;
 return 0;
}
```

NASM: calling assembler from C/C++

```
; fact.asm
SECTION .text      ; program

global factorial   ; linux
global _factorial  ; windows

 factorial:
_factorial:
    push ebp            ; save base pointer
    mov  ebp, esp       ; store stack pointer
    push ecx            ; save ecx
    ;;; start function
    mov ecx, [ebp+8]    ; ecx = first argument
    mov eax, 1          ; eax = 1
  mainloop:
    cmp  ecx, 0         ; if(ecx == 0)
    jz   done           ;  goto done
    mul  ecx            ; else eax = eax * ecx
    dec  ecx            ;      ecx = ecx - 1
    jmp  mainloop
  done:
    pop  ecx            ; restore ecx
    pop  ebp            ; restore ebp
    ret                 ; return from function
```

NASM: calling assembler from C/C++

Compiling `fact.asm` on LINUX:

```
nasm -f elf -o factasm.o fact.asm
gcc -o fact fact.c factasm.o
g++ -o fact fact.c factasm.o
```

Compiling `fact.asm` on WINDOWS:

```
nasm -f win32 -o factasm.o fact.asm
gcc -o fact.exe fact.c factasm.o
g++ -o fact.exe fact.c factasm.o
```

NASM: calling C from assembler

```c
/* useprintf.c */
#include <stdio.h>

extern void use_printf();

int main(void)
{
 use_printf();
 return 0;
}
```

NASM: calling C from assembler

```
; printf.asm
SECTION .text

extern printf        ; linux
extern _printf       ; windows
global use_printf    ; linux
global _use_printf   ; windows

message:
    db "output some text", 10, 0   ; newline, null terminator

 use_printf:
_use_printf:
    push message
    call printf
    pop  eax     ; first parameter
    ret
```

NASM: calling C from assembler

Compiling `printf.asm` on LINUX:

```
nasm -f elf -o printf.o printf.asm
gcc -o useprintf useprintf.c printf.o
```

Compiling `printf.asm` on WINDOWS:

```
nasm -f win32 -o printf.o printf.asm
gcc -o useprintf.exe useprintf.c printf.o
```

NASM: main program in assembler

```
#include <stdio.h>

int main(int argc, char *argv[])
{
 int integer1, integer2;
 printf("Enter the first number: ");
 scanf("%d", &integer1);
 printf("Enter the second number: ");
 scanf("%d", &integer2);
 printf("%d\n", integer1+integer2);
 return 0;
}
```

NASM: main program in assembler

```
; add1.asm

SECTION .data

message1: db "Enter the first number: ", 0
message2: db "Enter the second number: ", 0
formatin: db "%d", 0
formatout: db "%d", 10, 0 ; newline, nul terminator

integer1: times 4 db 0   ; 32-bits integer = 4 bytes
integer2: times 4 db 0   ;

SECTION .text

global main      ; linux
global _main     ; windows

extern scanf     ; linux
extern printf    ; linux
extern _scanf    ; windows
extern _printf   ; windows
```

NASM: main program in assembler (cont.)

```
; add1.asm
_main:
 main:
    push ebx            ; save registers
    push ecx

    push message1
    call printf
    add  esp, 4         ; remove parameters

    push integer1       ; address of integer1 (second parameter)
    push formatin       ; arguments are right to left (first parameter)
    call scanf
    add  esp, 8         ; remove parameters

    push message2
    call printf
    add  esp, 4         ; remove parameters

    push integer2       ; address of integer2
    push formatin       ; arguments are right to left
    call scanf
    add  esp, 8         ; remove parameters
```

NASM: main program in assembler (cont.)

```
; add1.asm (main)

    mov  ebx, dword [integer1]
    mov  ecx, dword [integer2]
    add  ebx, ecx      ; add the values

    push ebx
    push formatout
    call printf        ; display the sum
    add  esp, 8        ; remove parameters

    pop  ecx
    pop  ebx           ; restore registers in reverse order
    mov  eax, 0        ; no error
    ret
```

NASM: main program in assembler (cont.)

Compiling `add1.asm` on LINUX:

```
nasm -f elf -o add1.o add1.asm
gcc -o add1 add1.o
```

Compiling `add1.asm` on WINDOWS:

```
nasm -f win32 -o add1.o add1.asm
gcc -o add1.exe add1.o
```

NASM: command line arguments

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
 int integer1, integer2;
 integer1 = atoi(argv[1]);
 integer2 = atoi(argv[2]);
 printf("%d\n", integer1+integer2);
 return 0;
}
```

Below we provide our own `atoi` function with a custom calling convention.

NASM: command line arguments

```
; add2.asm

SECTION .text

global main       ; linux
global _main      ; windows

extern printf     ; linux
extern _printf    ; windows
```

NASM: command line arguments (cont.)

```
; add2.asm

; we assume the argument is in eax
atoi:
    push ebx
    push ecx
    mov  ebx, eax
    mov  eax, 0
  atoi_repeat:
    cmp  byte [ebx], 0
    je   atoi_done
    mov  ecx, 10
    mul  ecx                    ; eax *= 10, make place for the next digit
    mov  cl, byte [ebx]
    sub  cl, '0'
    add  eax, ecx
    inc  ebx
    jmp  atoi_repeat
  atoi_done:
    pop  ecx
    pop  ebx
    ret
```

NASM: command line arguments (cont.)

```
; add2.asm

_main:
 main:
    push ebp
    mov  ebp, esp
    push ebx          ; save registers
    push ecx

    cmp  dword [ebp+8], 3   ; argc
    jne  main_end

    mov  ebx, [ebp+12]      ; argv

    mov  eax, [ebx+4]       ; argv[1]
    call atoi
    mov  ecx, eax           ; store the result

    mov  eax, [ebx+8]       ; argv[2]
    call atoi

    add  eax, ecx           ; add the values
```

NASM: command line arguments (cont.)

```
; add2.asm (main)

    push eax
    push formatout
    call printf        ; display the sum
    add  esp, 8        ; remove parameters

  main_end:
    pop  ecx
    pop  ebx           ; restore registers in reverse order
    pop  ebp
    mov  eax, 0        ; no error
    ret

SECTION .data

formatout: db "%d", 10, 0 ; newline, nul terminator
```

NASM: command line arguments (cont.)

Compiling `add2.asm` on LINUX:

```
nasm -f elf -o add2.o add2.asm
gcc -o add2 add2.o
```

Compiling `add2.asm` on WINDOWS:

```
nasm -f win32 -o add2.o add2.asm
gcc -o add2.exe add2.o
```

NASM: DOS interrupts

Operating systems provide different mechanisms for using the operating system facilities. The DOS operating system uses interrupts, specifically interrupt 21h. Interrupts are similar to functions using CALL and RET, however parameters are usually passed via registers and we use INT and IRET. Hardware events also trigger interrupts which is used when writing device drivers.

A list of interrupt calls can be found at:

`http://www.cs.cmu.edu/~ralf/files.html`
`http://www.ctyme.com/rbrown.htm`

NASM: DOS interrupts

DOS provides functionality via interrupt 21h. The requested function must be specified in `AH`. In the example program we use the following functions.

`AH`	Function
02h	Write the ASCII character in `DL` to stdout
09h	Write the `$` terminated string at `DS:DX` to stdout
0Bh	Check stdin (`AL` $= 0 \rightarrow$ no character available)
2Ch	Get time (hour: `CH`, minute: `CL`, second: `DH`)
4Ch	Terminate and return to DOS with return code `AL`

NASM: DOS clock

The following program displays a clock. Whenever the time changes, the displayed time must be updated.

The `ORG` directive is used to tell NASM that the programm starts at byte 100h, which is the format for a DOS `.COM` file. In this format, all segment registers point to the same segment, so it is not neccessary to change `DS` when printing a string.

```
ORG 100h        ; DOS COM file

main:
 MOV  AH, 0Bh  ; check stdin
 INT  21h      ; call DOS
 CMP  AL, 0    ; no character
 JNE  return   ; user pressed a key - exit
```

NASM: DOS clock

```
MOV  AH, 2Ch  ; get time
INT  21h      ; call DOS

CMP  DH, BH
JE   main     ; seconds did not change
MOV  BH, DH

PUSH DX
MOV AH, 09h   ; write string
MOV DX, clear_line
INT 21h
POP DX
```

NASM: DOS clock

```
XOR  AX, AX
MOV  AL, CH         ; hour
CALL show_integer

MOV  AH, 02h
MOV  DL, ':'
INT  21h

XOR  AX, AX
MOV  AL, CL         ; minute
CALL show_integer

MOV  AH, 02h
MOV  DL, ':'
INT  21h

XOR  AX, AX
MOV  AL, DH         ; second
CALL show_integer

JMP main
```

NASM: DOS clock

```
return:
 MOV AH, 09h  ; write string
 MOV DX, new_line
 INT 21h

 MOV AH, 4Ch  ; terminate program
 MOV AL, 0    ; no error
 INT 21h      ; call dos

clear_line: db "                ", 13, '$'
new_line:   db 10, 13, '$'
buffer:     times 100 db 0
            db '$'
```

NASM: DOS clock

```
; show ascii representation of AX
show_integer:
 PUSH BX
 PUSH CX
 PUSH DX
 MOV  BX, buffer
 ADD  BX, 100
 MOV  CX, 10
show_integer_loop:
 CMP  AX, 0
 JE   show_integer_done
 MOV  DX, 0
 DIV  CX
 ADD  DL, '0'
 DEC  BX                     ; write number backwards
 MOV  BYTE [BX], DL
 JMP  show_integer_loop
```

NASM: DOS clock

```
show_integer_done:
 CMP  BYTE [BX], '$'  ; empty string
 JNE  not_zero
 DEC  BX
 MOV  BYTE [BX], '0'
not_zero:
 MOV AH, 09h          ; write string
 MOV DX, BX
 INT 21h
 POP  DX
 POP  CX
 POP  BX
 RET
```