Computer Architecture and System Programming Laboratory

TA Session 4

MUL, DIV

Dot Product code example (self-study)

ld Assembler (_start, exit)

gdb debugger

Advanced Instructions - multiplication

MUL r/m - unsigned integer multiplication

IMUL r/m - signed integer multiplication

Multiplicand	Multiplier	Product
AL	*r/m8*	AX
AX	*r/m16*	DX:AX
EAX	*r/m32*	EDX:EAX

MUL / IMUL - multiply unsigned / singed numbers
Why are two different instructions needed?

Example:
MUL: 0xFF x 0xFF = 255 x 255 = 65025 (0xFE01)
IMUL: 0xFF x 0xFF = (−1) x (−1) = 1 (0x0001)

MUL r/m8

```
mov al, 9 ; multiplicand
mov bl, 5 ; multiplier
mul bl    ; = 0x2D
```

AX

0x00	0x2D

MUL r/m16

```
mov ax, 0x2000
mov bx, 0x8000
mul bx    ; = 0x10000000
```

DX

0x10	0x00

AX

0x00	0x00

MUL r/m32

```
mov eax, 0x20002000
mov ebx, 0x80008000
mul ebx   ; = 0x10002000 10000000
```

EDX

0x10	0x00	0x20	0x00

EAX

0x10	0x00	0x00	0x00

Advanced Instructions – division

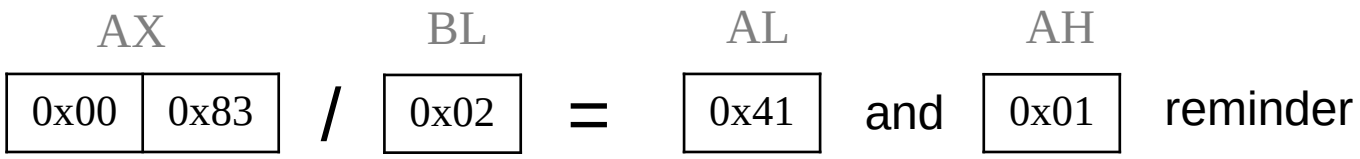
DIV r/m - unsigned integer division

IDIV r/m - signed integer division

Dividend	Divisor	Quotient	Remainder
AX	*r/m8*	AL	AH
DX:AX	*r/m16*	AX	DX
EDX:EAX	*r/m32*	EAX	EDX

DIV r/m8

```
mov   ax, 0x83     ; dividend
mov   bl, 0x02     ; divisor
DIV   bl           ; al = 0x41 quotient, ah = 0x01 remainder
```



Advanced Instructions – division

DIV r/m - unsigned integer division

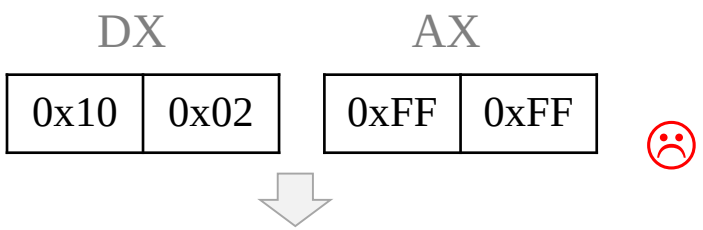
IDIV r/m - signed integer division

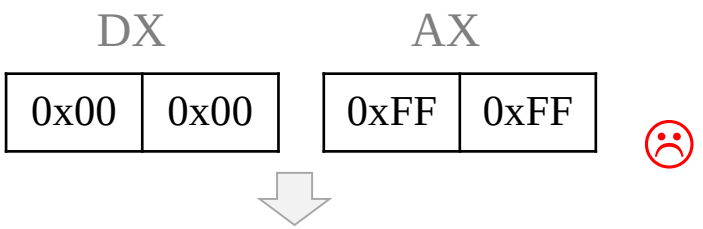
Dividend	Divisor	Quotient	Remainder
AX	*r/m8*	AL	AH
DX:AX	*r/m16*	AX	DX
EDX:EAX	*r/m32*	EAX	EDX

IDIV r/m16

```
mov    ax, 0xFFFF   ; dividend, low-part register
mov    cx, 0x100    ; divisor
IDIV   cx           ; ax = 0x0080 quotient,  dx = 0x0003 remainder
```

required: 0xFFFF/ 0x100 = -1/0x100





Advanced Instructions – division

DIV r/m - unsigned integer division

IDIV r/m - signed integer division

Dividend	Divisor	Quotient	Remainder
AX	*r/m8*	AL	AH
DX:AX	*r/m16*	AX	DX
EDX:EAX	*r/m32*	EAX	EDX

for r/m32 use '**cdq**' to convert EAX doubleword to EDX:EAX quadword

IDIV r/m16

```
mov    ax, 0xFFFF   ; dividend, low-part register
mov    cx, 0x100    ; divisor
cwd                 ; convert AX word to DX:AX double word by copying MSB of AX to all the bits of DX
IDIV   cx           ; ax = 0x0080 quotient,  dx = 0x0003 remainder
```

required: 0xFFFF/ 0x100 = -1/0x100

DX		AX	
0xFF	0xFF	0xFF	0xFF

☺

executed: 0xFFFFFFFF / 0x100

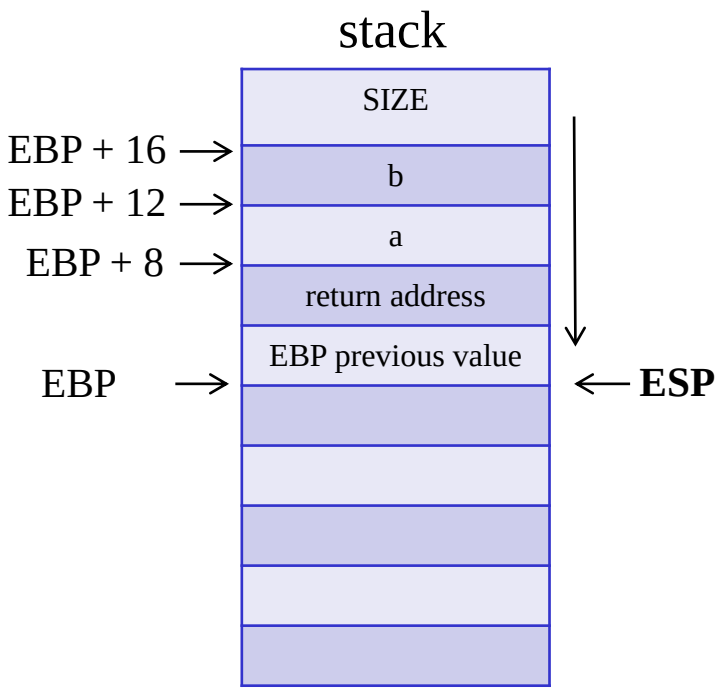
Multiplication – Code Example

self reading

The dot product of two vectors **a** = [a_1, a_2, ..., a_n] and **b** = [b_1, b_2, ..., b_n] is defined as:

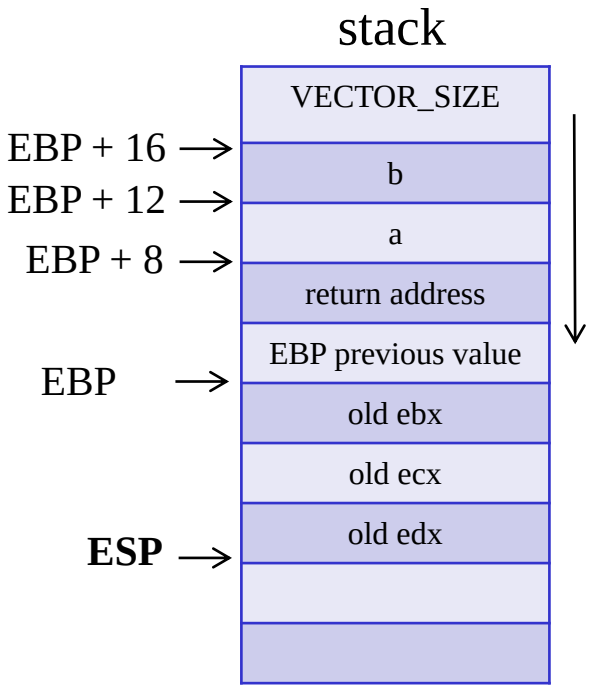
$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

```
#include <stdio.h>
#define SIZE 5
extern long long * DotProduct (int a[SIZE], int b[SIZE], int size);

void main () {
  int a[SIZE] = {1,0,1,0,2};
  int b[SIZE] = {1,0,1,0,-2};

 long long * result = DotProduct(a, b, SIZE);
  printf ("%#llx\n ", result);
}
```



The dot product of two vectors **a** = [a_1, a_2, ..., a_n] and **b** = [b_1, b_2, ..., b_n] is defined as:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$



next element of vector a

next element of vector b

self reading

```
section     .data
result:  dd  0,0
section     .text
         global  DotProduct
DotProduct:
         push    ebp
         mov     ebp, esp
         push    ebx
         pushad
         mov     ecx, 0
DotProduct_start:
         mov     edx, 0
         cmp     ecx, dword [ebp+16]
         je      DotProduct_end
         mov     ebx, dword [ebp+8]
         mov     eax, dword [ebx + (4*ecx)]
         mov     ebx, dword [ebp+12]
         imul    dword [ebx + (4*ecx)]
         add     dword [result], eax
         adc     dword [result+4], edx
         inc     ecx
         jmp     DotProduct_start
DotProduct_end:
         mov     eax, result  ; return value
         pop     edx
         pop     ecx
         pop     ebx
         mov     esp, ebp
         pop     ebp
         ret
```

Assembly program with no C file usage

```
section .data
        …
section .bss
        …
section .rodata
        …
section .text
        global  _start   ; entry point
_start:

        …

        mov ebx,0        ; exit system call
        mov eax,1
        int 0x80
```

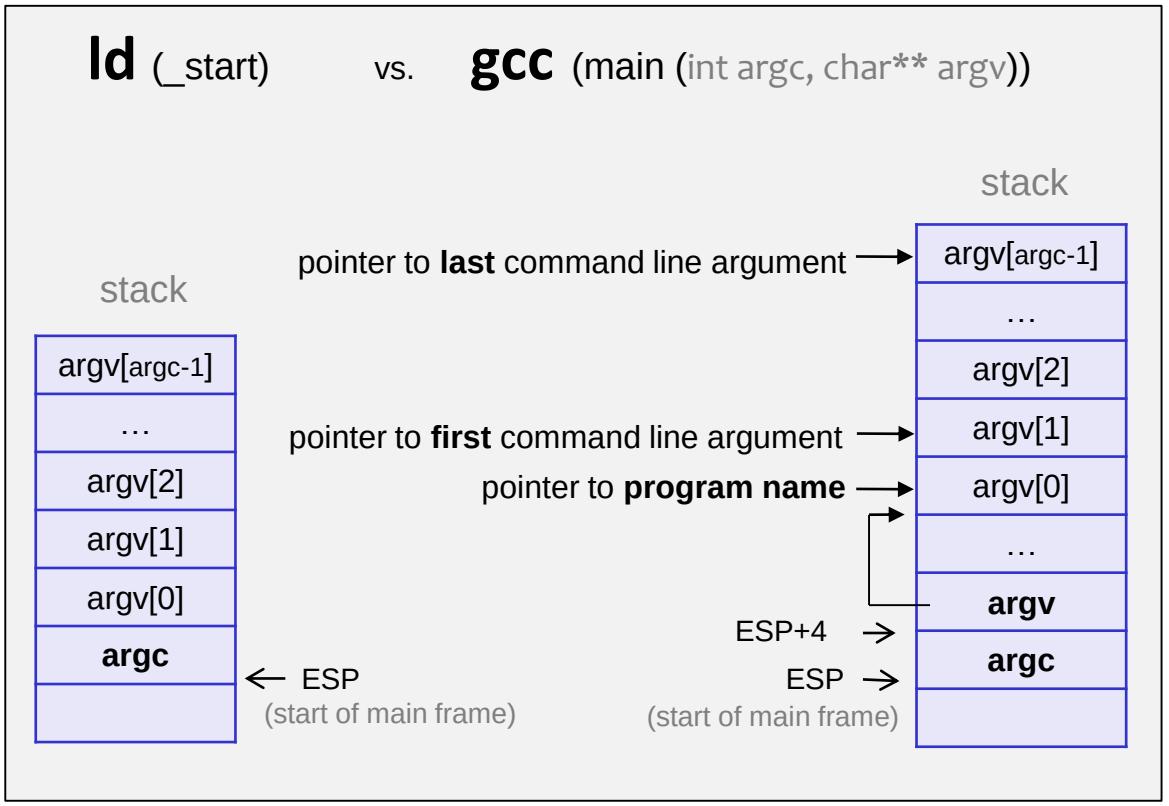
use 'main' entry point label in your pure assembly code to compile it with gcc

GNU Linker

ld links assembly object file(s)

```
> nasm –f elf32 asm.s –o asm.o
> ld -m elf_i386 asm.o –o asm
> ./asm
```

Command-line arguments

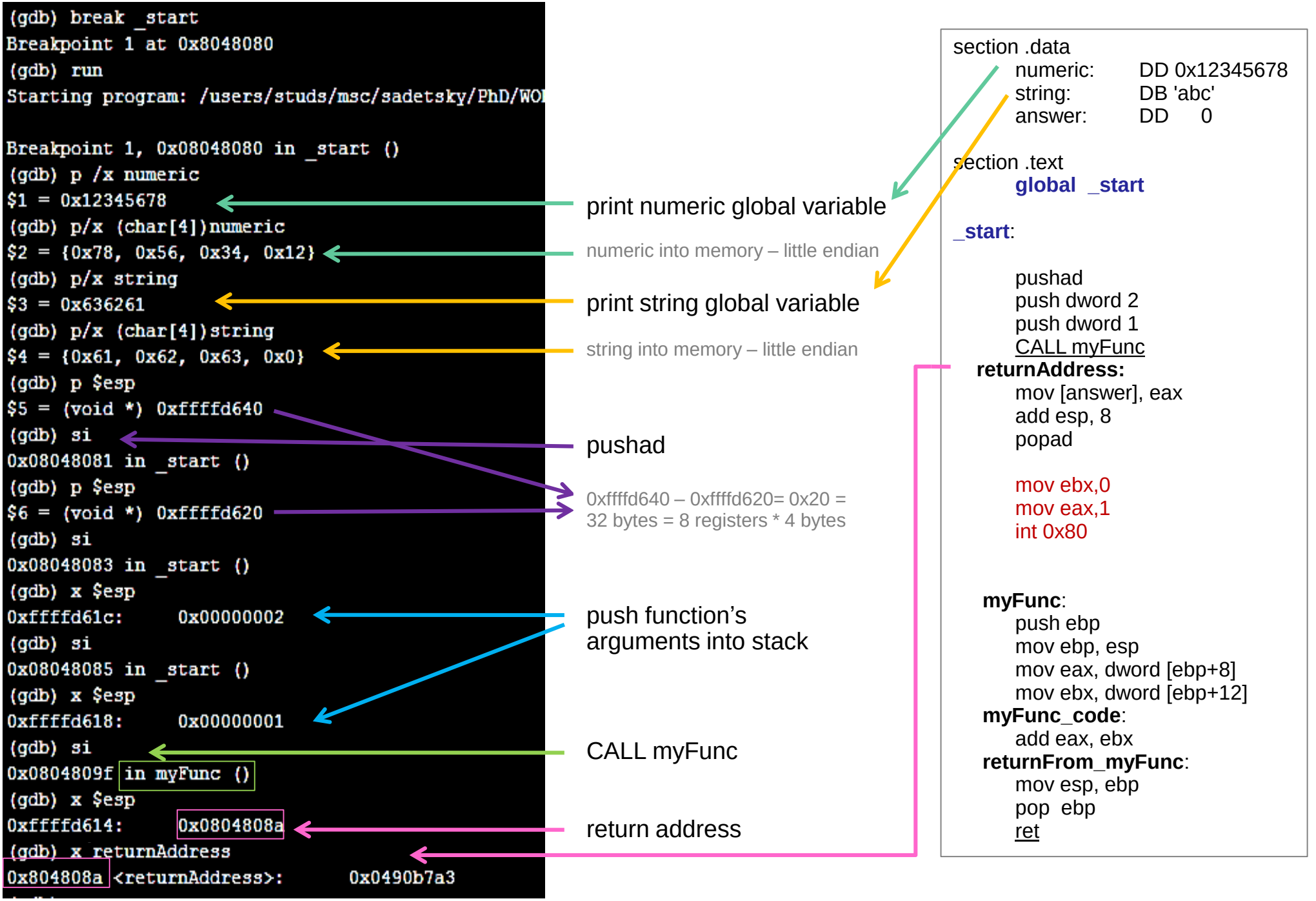


gdb-GNU Debugger – very basic usage

- run Gdb from the console by typing:
  gdb executableFileName
- add breaking points by typing:
  break label
- start debugging by typing:
  run (command line arguments)

(gdb) set disassembly-flavor intel — change presentation of assembly-language instructions from the default Motorola conventions, that are used by gdb, to the Intel conventions that are used by nasm, that is, from 'opcode source, dest' to 'opcode dest, src'
(gdb) layout asm – display assembly language
(gdb) layout regs – display registers

- **s/si** – one step forward
- **c** – continue to run the code until the next break point
- **q** – quit gdb
- **p/x $eax** – prints the value in eax
- **x $esp**– prints esp value (address) and value (dword) that is stored in this address. It is possible to use label instead of esp.
  Type **x** again will print the next dword in memory.


(gdb) break _start
Breakpoint 1 at 0x8048080
(gdb) run
Starting program: /users/studs/msc/sadetsky/PhD/WO
Breakpoint 1, 0x08048080 in _start ()
(gdb) p /x numeric
$1 = 0x12345678
(gdb) p/x (char[4])numeric
$2 = {0x78, 0x56, 0x34, 0x12}
(gdb) p/x string
$3 = 0x636261
(gdb) p/x (char[4])string
$4 = {0x61, 0x62, 0x63, 0x0}
(gdb) p $esp
$5 = (void *) 0xffffd640
(gdb) si
0x08048081 in _start ()
(gdb) p $esp
$6 = (void *) 0xffffd620
(gdb) si
0x08048083 in _start ()
(gdb) x $esp
0xffffd61c: 0x00000002
(gdb) si
0x08048085 in _start ()
(gdb) x $esp
0xffffd618: 0x00000001
(gdb) si
0x0804809f in myFunc ()
(gdb) x $esp
0xffffd614: 0x0804808a
(gdb) x returnAddress
0x804808a <returnAddress>: 0x0490b7a3
print numeric global variable
numeric into memory – little endian
print string global variable
string into memory – little endian
pushad
0xffffd640 – 0xffffd620= 0x20 =
32 bytes = 8 registers * 4 bytes
push function's
arguments into stack
CALL myFunc
return address
section .data
numeric: DD 0x12345678
string: DB 'abc'
answer: DD 0
section .text
global _start
_start:
pushad
push dword 2
push dword 1
CALL myFunc
returnAddress:
mov [answer], eax
add esp, 8
popad
mov ebx,0
mov eax,1
int 0x80
myFunc:
push ebp
mov ebp, esp
mov eax, dword [ebp+8]
mov ebx, dword [ebp+12]
myFunc_code:
add eax, ebx
returnFrom_myFunc:
mov esp, ebp
pop ebp
ret