VRIJE UNIVERSITEIT BRUSSEL

Computersystemen

WPO: Extra Session 5

David Blinder
Raees K. Muhamad

Overview

By now, we have covered the basics needed for all projects. Depending on your specific project requirements, you will need to research additional specific features/methods/algorithms.

Some of these features are (briefly) covered in this extra session.

- Random number generation
- Interactive keyboard and mouse
- Reading file data
- Floating-point numbers
- Structs and unions
- Advanced macros usage
- Extra useful instructions
- Jump tables
- Playing sound

VRIJE UNIVERSITEIT BRUSSEL

ASMBox EXAMPLES

ASMBox examples

Whichever project you're planning to make, please look at the code in the EXAMPLES folder. There are very few relevant examples online, so this may give you a reference for new ideas and good coding practices.

The topics covered in the examples are (in order of complexity):

- **RAND** pseudo-random number generation
- **MOUSE** handling of mouse input
- **MYKEYB** handling of interactive keyboard input (games)
- **DANCER** reading and processing files, animations
- **FPU** working with the floating-point registers & instructions
- **BADAPPLE** video decoding algorithms, sound, custom interrupts, multiple file handling, video mode 11h. Resembles a project.

VRIJE UNIVERSITEIT BRUSSEL

Feel free to use the example code in your projects.

Floating-point instruction set

Floating-point numbers

Floating-point numbers are an approximate representation of real numbers for numerical calculations. They are computed using a different set of registers and instructions.

Floating-point numbers are respresented in (binary) scientific notation, supporting a large range of possible values:

$$\underbrace{(-1)^{s}}_{\textbf{sign bit}} \cdot 1.\underbrace{mmmm \ldots mmmm}_{\textbf{mantissa bits}} \cdot 2^{\overbrace{eeee \ldots eeee}^{\textbf{exponent bits (signed integer)}}}$$

Type	Sign	Mantissa	Exponent
Float (32-bit)	1	23	8
Double (64-bit)	1	52	11
FPU register (80-bit)	1	64	15

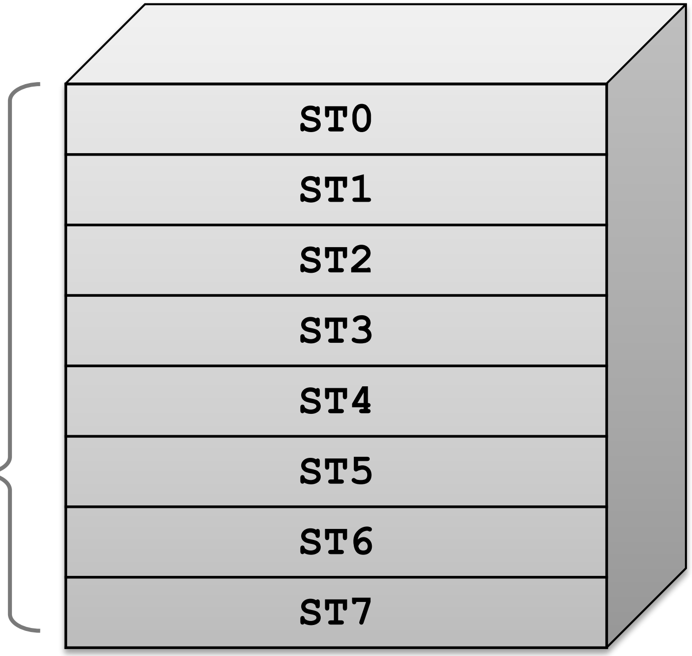
Distribution of #bits

Floating-Point Unit (FPU): registers

Floating-point instructions are computed using **seperate registers** with a **different associated instruction set**. The FPU has i.a. eight 80-bit data registers for performing floating-point calculations (`ST0`, `ST1`, …, `ST7`) and seperate control and status registers.

Refer to them using `st(0)`, `st(1)`, …, `st(7)`. Typically one of both instruction operands always must be the top of stack `ST0`.

Load with `fld`. Pushes down on top of stack. Use `fild` for integers



80-bit registers

Store with `fst` (or `fist`). To pop values, use `fstp`/`fistp`.

Floating-Point Unit (FPU): instruction set

We cannot cover the FPU instruction set in-depth, given time constraints and scope. See the pdf of exercise session 5 for additional resources. Some of the most important instructions are:

```
fabs                        ; take absolute value
fadd/faddp                  ; addition
fchs                        ; change sign
fcom/fcomp/fcompp           ; compare
fdiv/fdivp                  ; divide
finit                       ; initialize FPU
fmul/fmulp                  ; multiply
fld (fild)                  ; load float (or convert int)
fldz, fld1, fldpi           ; load constants (0, 1, pi)
fst/fstp (fist/fistp)       ; store float (or convert int)
fsqrt, fsin, fcos, fsincos  ; square root, sine, cosine
fsub                        ; subtract
```

Structs and Unions

Structs

A `struct` is a composite data type declaration that defines a physically grouped list of variables in a block of memory.

```
STRUC enemy_ship
   x         dw 0         ; x position
   y         dw 0         ; y position
   speed_x   dw 0         ; x speed component
   speed_y   dw 0         ; y speed component
   color     db 1         ; ship hull color
   health    db 100       ; health points (HP)
   sprite    dd 0         ; pointer to used sprite image
ENDS enemy_ship
```

Access fields with the dot operator:

```
mov ebx, [@@pointer]                      ; local pointer to enemy_struct
mov ax, [ebx + enemy_ship.speed_x]          ; speed_X → AX
sub [ebx + enemy_ship.health], 10           ; subtract 10 HP
```

Unions

In a `union` all member variables have the same address in memory. At any given time, only one of the member variables should be used if unless the others be overwritten.

```
UNION color_union
     palette_index  db 0       ; palette index color
     RGBA_tuple     dd 0       ; RGBA 4-byte tuple color
ENDS color_union
```

Nesting structs and unions

You can nest **structs** and **unions:**

```
STRUC vehicle
   passengers  dd 0
   fuel        dd 0
   vehicle_id  db 0  ; e.g.: CAR=0, PLANE=1, TRUCK=2
   UNION vehicle_type
         STRUCT plane_type
               wing_length dd 0
               engine_count dw 0
         ENDS plane_type
         STRUCT truck_type
               cargo_weight dd 0
               container_size dd 0
         ENDS truck_type
   ENDS vehicle_type
ENDS vehicle
```

More on MACROS

More on MACROS

Aside from constants, macros can also be used to define small functions. They will effectively copy and insert the code everywhere you call the macro. This can improve efficiency and source code quality for small functions, but it is not a substitute for all procedures!

```
; macro template with optional input parameters
MACRO macroname optional_param1 optional_param2 ...
  LOCAL ... ; optional local variables
  ...
ENDM macroname
```

VRIJE UNIVERSITEIT BRUSSEL

More on MACROS (example: 486 BSWAP instruction)

```
; flip order of bytes of EAX
MACRO swap_bytes_eax
  xchg ah, al
  ror eax, 16
  xchg ah, al
ENDM swap_bytes_eax

PROC main
  ...
  ; goal: flip bytes so that EAX = 78563412h
  mov eax, 12345678h
  ; will get replaced by macro instructions above
  swap_bytes_eax
  ...
ENDP main
```

REP instructions summary

x86 `REP` instructions: summary

`REP` instructions *(Repeat String Operation Prefix):* Replace `x` by `B/W/D` for working with **byte** (8-bit), **word** (16-bit) or **doubleword** (32-bit) elements.

```
REP INSx          ; Input ECX elements from port DX into [EDI].
REP MOVSx         ; Move ECX elements from [ESI] to [EDI].
REP OUTSx         ; Output ECX bytes from [ESI] to port DX.
(REP LODSx)       ; Load ECX elements from [ESI] to AL/AX/EAX
REP STOSx         ; Fill ECX elements at [EDI] with AL/AX/EAX.
REPE CMPSx        ; Find nonmatching element in [EDI] and [ESI]
REPNE CMPSx       ; Find matching element in [EDI] and [ESI]
REPE SCASx        ; Find non-AL/AX/EAX element starting at [EDI]
REPNE SCASx       ; Find AL/AX/EAX element starting at [EDI]
```

Note: you can use the instructions without “REP” for a single instruction execution round. Especially useful for `LODSx`, `STOSx` and `MOVSx`.

Jump tables

Jump tables

Instead of writing plenty of `CMP`, `JE` pairs, use the equivalent of a switch statement in assembly: a **jump table**. It's more readable & efficient.

```
switch (ebx) {
        case 0:
                ...
                break;
        case 1:
                ...
                break;
        case 2:
                ...
        case 3: //fallthrough
                ...
                break;
        case 4:
                ...
                break;
}
```

```
CODESEG
...
    PROC main
       ...
       jmp [Jump_table + 4*ebx]
      zerolbl: ...
       jmp endlbl
       onelbl:...
       jmp endlbl
      twolbl: ...
      threelbl: ... ; fallthrough
        jmp endlbl
      fourlbl: ...
      endlbl: ...
    ENDP main

DATASEG
  Jump_table dd zerolbl, onelbl,   twolbl,
threelbl, fourlbl
```

Labels are just pointers in the code segment

Good to know: additional tips and instructions

Good to know

Use `UDATASEG` when declaring arrays of uninitialized elements instead of `DATASEG`. This will reduce the size of your executable.

```
…
DATASEG
	codes dw 100, 378, 2345, 10987
	coords dd 12.0, 30.0, -12.0, 46.8 ;32-bit floats
UDATASEG
	palette db 768 dup (?)
```

You can use the specialized instruction

```
	JECXZ (label)
```

to directly jump to `(label)` when `ECX` is zero, without needing any compares, nor modifying any flags.

VRIJE UNIVERSITEIT BRUSSEL

Good to know

Bit test instructions: copy single bit into carry flag (CF) + optional modify
`BT` (bit test), `BTS` (bit test and set to 1), `BTR` (bit test and reset to 0), `BTC` (bit test and complement, negates tested bit).

Find the least/most significant (1) bit position of a register with `BSF`/`BSR`.

Copy flags into a byte register with `SETcc` instructions:
`SETA, SETB, SETC, SETE, SETG, SETGE, SETL, SETLE, SETNE, …`

Other bit-shifting instructions:
`ROL`, `ROR`: bitshift left/right operation, but rotate bits shifted out back around into register from other side.
`RCL`, `RCR`: idem, but include carry bit into the bit shift loop.

`SHLD`, `SHRD`: bitshift left/right # bits from one register into another one. Only modifies the first operand. Example usage: `shld eax, edx, cl`

Extra Exercises

Extra Exercises

Note: these optional exercises and their solutions will not be covered in class.

1. Write a macro computing the absolute value of `EAX`, keeping the result in `EAX`. Try not to use any branching (i.e. no jumps).

2. Use the FPU to write a procedure computing the distance between two points (x_0, y_0) and (x_1, y_1) with input integer coordinates.
   Return the distance as an integer in `EAX`, rounded down.

3. Use the FPU to draw pixels in concentric circles on the screen. Center them in the middle of the screen, draw them for all radii from 30 to 90 in steps of 5, using 64 points per circle.

4. Write a procedure replacing all characters with argument `find` by the character `replace` using `REP SCAS`. For example, if `'A'` and `'D'`, then `ABCDAABCAC` becomes `DBCDDDBCDC`. Give as additional input arguments the pointer to the text data, and a counter for the length of the string.