AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Introduction to Computer Science
NASM Assembler

Version: 2024

Marek Wilkus, Ph.D. **http://home.agh.edu.pl/~mwilkus**
Faculty of Metallurgy and Industrial Computer Science
AGH UST Kraków

History introduction

- First programmable computers were programmed by **adapting the hardware memory to the program**. Usually by plugs or switches.

- When the program memory became a part of the random-access memory, it was possible to **program the computer by inserting commands right to the specific memory addresses.**

- The programmer had to know what bit alignment does what, write it in own words, then translate it to machine code and enter the commands on the "console".

Introduction

- With growing complexity of software, it was needed to **encode the program** the way that it would be easier to modify or adapt.

- The machine-specific commands have been described using handy abbreviations called **mnemonics.**

```
;*****************************************************************
;
RDBOOT: CALL  SETSD     ;set single density mode
        LD    C,0       ;select drive 0
        LD    DE,0      ;set track 0, logsec 0
        LD    HL,LDBUF  ;set bootstrap allocation address
        CALL  RDSEC     ;read selected setor
        JR    NZ,TRYDD  ;Jump if any error occur
        RET             ;exit OK
TRYDD:  CALL  SETDD     ;set double density mode
        CALL  RDSEC     ;read selected sector
        RET
;
;*****************************************************************
```

- Usually, the mnemonics had some arguments, like value to put into the register. These mnemonics were expanded with them to form one-line assembly commands.

The objective of this lecture

- Introduce with general knowledge of Intel x86 assembler principles,
- Present some of problems during assembly programming,
- Show general structure of assembly programs, and how to make one.

- Yes, Intel is not a good assembler to begin with, but the platform is widely used.

Intel CPUs

- In the 70s, Intel created 8088 and 8086 processors.
  - 16-bit registers, 1MB of address space.
- 1982: Intel 80286:
  - 16MB of address space, more instructions.
- 1985: Intel 80386:
  - 4GB of address space, 32-bit registers, protected mode, more instructions.
- 1989: Intel 486:
  - More address space, more commands, 32-bit registers, FPU built-in.
- 1992 – Pentium:
  - Incremental changes, power management.
- 1995, 1997, 1999 – MMX, 3DNow!, SSE instruction enhancement for CPUs
- 2003 – 64-bit x86:
  - 64-bit registers.

Intel architecture

- Lots of backward compatibility.
- Quite troublesome because it was never redesigned from scratch – it extends previous versions.

- The initial architecture had the following registers:
  - AX, BX, CX, DX – General-purpose 16-bit registers (each can be used as two 8-bit: AH, AL, BH, BL, etc.)
  - SI and DI general purpose 16-bit registers, for intention to be used as indexes or pointers.
  - BP and SP pointer registers (Base Pointer, Stack Pointer),
  - CS, DS, ES, SS segment registers (we will not use them),
  - IP Instruction Pointer,
  - Flags register.

1MB of address space vs 16-bit register?

- With 16 bits, you can address 2^{16}=64kB of memory!
- How they addressed 1MB with its 2^{20} bits?

- Selector + Offset:
  - First group of bits defines a **segment**.
  - Second – **offset** in that segment.
  - This way, both of these numbers count from 0 upwards.
  - Scrolling through the memory, the selector stays the same for a longer time, and the offset changes much faster, again and again as selector slowly increases.
  - For 20-bit addresses, we need 16 bits for offset and 4 bits for segment selector.

    (2^4=16, 2^{16}=65536, 65536*16=1048576=1MB)

Summing up the Intel’s trick

- It’s good that we will use a 64-bit assembler in this course.

- If you program something under 64kB, you don’t neet to think about segments.
- If it is larger, you constantly need to make sure you have chosen the correct segment. If not, switch back and forth which is troublesome.

The Assembler

- Because we are working directly with CPU’s commands, this is a **really fast** language.

- On the other side, it is **more difficult** to align a complex program, it is **platform-specific,**
  - The aspect of being platform-specific is important when looking for information about solving a specific problem!

- There are **many assemblers** for the same architecture. They differ by used mnemonics or their order, or some macro-mnemonics (mnemonics which are substituted by specific blocks).

The assembly process

- First, the mnemonics are detected and translated to the machine code "object file".
  - Note that if you make a mistake, but mnemonics are correct and their arguments are possible (could be a total nonsense), there **will be no error shown**.

- Then, object files are joined properly by the **linker**, proper headers and designations are added and the **executable** file you can run is generated.

Netwide Assembler (NASM)

- An assembler/disassembler for x86-64 architecture.
- Operates under Windows, DOS, Linux and a few other OS,
- Outputs **object files** which have to be then linked into the executable.

- Assembler code files traditionally have an **.asm** extension.
- Object files -**.o**,
- Executables in Unix have no extension or have a default name **a.out** (even if they are not accorfing to a.out executable standard, but modern ELF standard).

Build the program.asm in Linux:

- `nasm -felf64 program.asm`
- `ld program.o`
- `./a.out`

- Or in an one-liner:

```
nasm -felf64 program.asm; ld program.o; ./a.out
```

Assembler program template:

```
bits 64

; The program writes a pre-defined text to console
          global    _start

; This is a section with program's code
          section   .text
_start:   mov       rax, 1                  ; system call number 1: Write to handle
          mov       rdi, 1                  ; Handle number 1: The console
          mov       rsi, message            ; The address to output
          mov       rdx, 13                 ; Number of bytes to write
          syscall                           ; Call the system routine

          mov       rax, 60                 ; System call number 60: Exit
          xor       rdi, rdi                ; Zero the rdi - exit code 0
          syscall                           ; Call the system routine


; This is a section with program's data and constants.

          section   .data
message:  db        "Hello, World", 10      ; note the newline at the end
```

Labels Instructions Operands Comments

Mnemonics

- **mov** x y – moves y to x. Y can be a constant, register or memory location. Both operands must be the same size.

- **xor** x y – xor-s x with y, writing the output to the x. Like **and**, **or**, ...
- ...but also **add** or **sub –** add or subtract.

- **syscall** – a macro! Calls the operating system's routine.
  - The **routine's number** is stored in rax.
  - The arguments may be stored in other registers.

- **db** – declare bytes – put bytes in the memory. The label works then as bytes' address.

The program

- **Directives –** these inform about general conditions of assembling the program.
- **Labels** – work as a "chekcpoints" in program’s memory. Program can use their address (if we label some data) or jump to them (if we label program’s part).
- **Sections** – contain instructions. There should be a code section and a data section, as in the example.
- We usually **ask the operating system to end the program** by executing a specific syscall at the end.

The registers

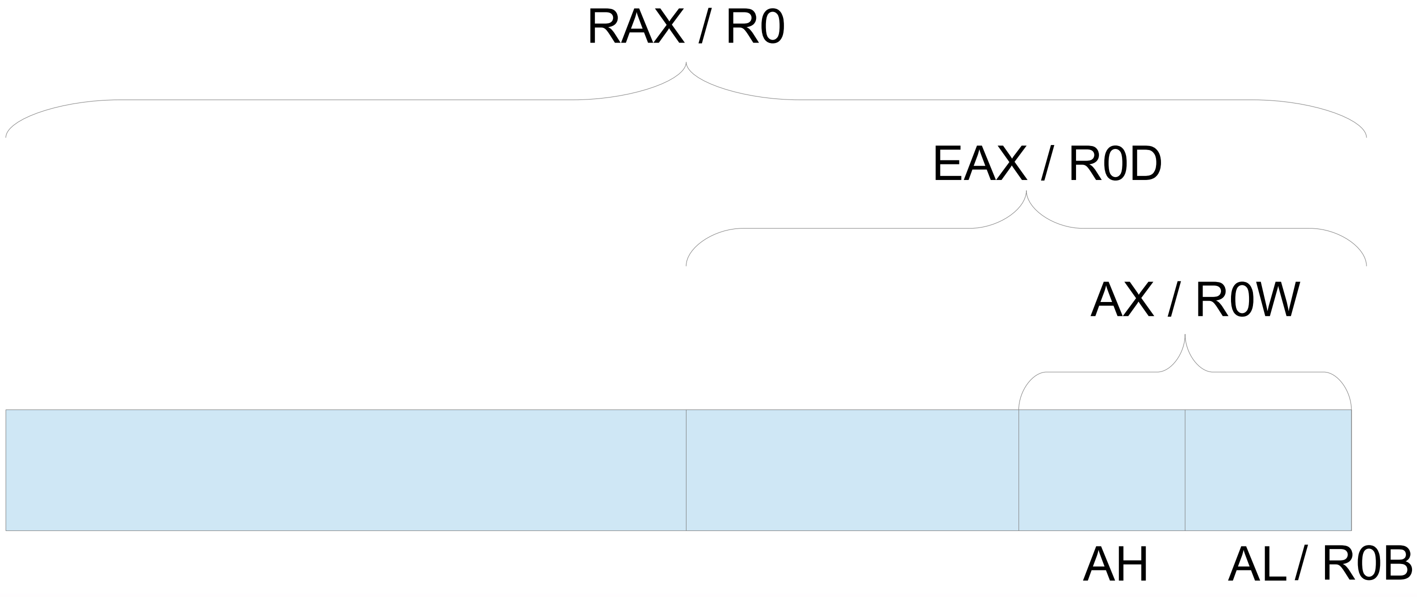
- We used the following registers: rax, rdi, rsi, rdx. There are also rcx, rbx, rsp and rbp. So R0, R1, R2 .. R7 are not used, RAX, RBX, RCX, ... RDI are.
  – If you pass %use altreg into the code after bits 64, it will be possible to use R0, R1 ... R7 names.

- There are **16** basic integer registers available, and the 8 remaining registers are called just r8, r9, r10, … r14, r15.

R0 or RAX	R1 or RCX	R2 or RDX	R3 or RBX	R4 or RSP	R5 or RBP	R6 or RSI	R7 or RDI	R8	R9	R10	R11	R12	R13	R14	R15

Intel's backward compatibility

- Because of compatibility…
  - The lowest 32 bits of each of these first 8 64-bit registers can be considered as eax, edi, esi …
  - The lowest 16 bits of these are also available as ax, di, si etc.
  - and the lowest 8-bits of them can be used too – al, dl, sil etc.
  - ...And then, the highest bits are: ah, ch, dh, bh.



Memory operands

- As some command's operand, we can use data from the memory. Then, registers hold the **address** and we instruct the assembler to obtain data from the specific memory location.
- It is very rare for a command to allow two memory operands.

- The following operands can be used:
  - [x] – arbitrary memory address, x is a number.
  - [reg] – the memory address stored in a register.
  - [reg+x] – base register + displacement (offset).
  - [reg+reg*s] – where s=1, 2, 4 or 8 – useful when navigating data structures.
  - [reg+reg*s+x] – offset in the structure

Let’s use a memory now

```
bits 64
; The program writes a set of asterisks to the console
          global    _start
          section   .text
_start:
          mov       rdx, output             ; Move the address of dataSize to rdx
          mov       r8, 1                   ; initially 1 asterisk
          mov       r9, 0                   ; written so far - 0
line:
          mov       byte [rdx], '*'         ; move a '*' to the byte under [rdx] address
          inc       rdx                     ; advance the memory by 1
          inc       r9                      ; increase number of asterisks by 1
          cmp       r9, r8                  ; compare number of asterisks with target number
          jne       line                    ; if not equal, jump to 'line' again
lineDone:
          mov       byte [rdx], 10          ; append newline character (10)
          inc       rdx                     ; advance the memory by 1
          inc       r8                      ; add 1 to line length - will be 1 asterisk longer
          mov       r9, 0                   ; reset asterisks counter
          cmp       r8, maxlines            ; compare the current line with max lines
          jng       line                    ; not greater -> jump to the line again
done:
          mov       rax, 1                  ; system call for write
          mov       rdi, 1                  ; handle 1 is a console
          mov       rsi, output             ; address of 'output'
          mov       rdx, dataSize           ; number of bytes
          syscall                           ; system call to write our info
          mov       rax, 60                 ; system call for exit
          xor       rdi, rdi                ; exit code 0
          syscall                           ; system call to exit

          section   .data
maxlines  equ       8                       ; maximum number of lines - constant
dataSize  equ       44                      ; dataSize constant = 44

          section   .bss
output:   resb      dataSize                ; allocate 44 bytes for memory
                                            ; because 1+2+3+4+5+6+7+8=36 +8 newlines = 44
```

The result:

```
mcbx@mwilkus:~/Publiczny/0Dydaktyka/ICS/nasm$ nasm -felf64 tree.asm; ld tree.o; ./a.out
*
**
***
****
*****
******
*******
********
```

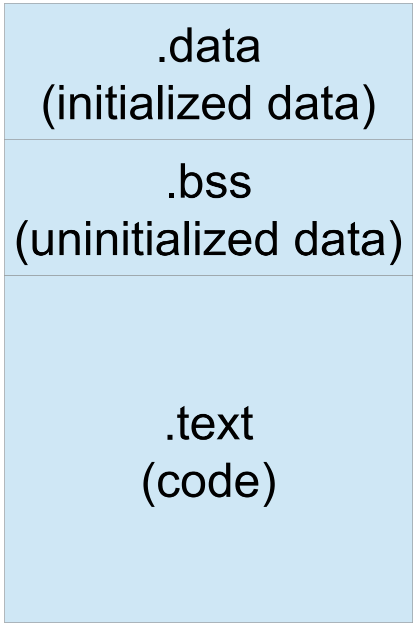
Commands used in this example

- **inc** – increment the register.
- **cmp** – compare two operands. The result can be checked by…
- **jne** – jump if not equal – jumps to label if comparison gave "not equal" result.
- **jng** – jump if not greater than.

- **resb** – reserve one (or more – here 44) bytes.
- **equ** – defines a constant.

.bss and .data segments

- Notice **we used .bss** segment in the second example, and **.data** in the first one.
- Generally, the **data segment** is used for **initialized memory**, while **bss** is used for uninitialized variables we will overwrite during program's execution.



- Resb vs db:
  - resb reserves uninitialized area.
  - db – defines the memory initializing it with value.

db vs equ

- db - define byte - this byte is in the memory, it has a value and can be used by address.
  - Like C's e.g. **int dataSize = 44;**

- equ - all calls to this symbol are replaced by value.
  - Like's C's e.g. **#define dataSize 44**

- Notice we don't address the equ by memory access (with brackets ~~[dataSize]~~) but by name which will get replaced.

Data „types”

- B – byte – 1 byte,
- W – word – 2 bytes,
- D – double word – 4 bytes,
- Q – quad word – 8 bytes.

- So we can **dd, dq**, **resq**, etc.

- Now for the future:
  Multi-byte value in registers is described as Little endian, while the memory uses Big Endian!

32	4A	00	00

In registers

00	00	4A	32

In memory

Mov-ing arbitrary values to the memory

- We write:

```
mov [rdx], 10
```

Will end with error.

- The assembler does not know what is this 10. Byte? Word? Double? Quad?
- We must show it what size we want:

```
mov byte [rdx], 10
```

- There are 5 size specifiers: byte, word, dword, qword and tword (10 bytes).

Mov-ing registers and the memory

- However, we can assume the size by the register size:

```
mov eax, [rdx]
```

- We know that eax is 4-bytes in width. So we take 4-bytes from location pointed in rdx, and copy them to eax register.
- And now we should remember this endianness problem.

Memory limitations

- Remember that in the assembly **there are no safeguards against overwriting** one part of memory by another.
- If we declare two 2-byte values and write 4-byte value in the first one, the leftover 4 bytes **will overwrite** the next variable without any warning.

- That’s why programming in assembler requires careful planning.

Assembly is dangerous

- While this overwriting can be used for some purposes, Intel’s Assembler becomes dangerous as it allows to do this:

- Now the memory structure we made lost all sense.

- Some architectures just will not let the programmer access an X-byte variable for an address which is not a multiple of X.

Using C libraries

```
bits 64
; Program counts 0..9

         global    main
         extern    printf
         section   .text
main:
         ; code goes here
         mov r15, 0             ; counter
         mov r14, 10            ; maximum
loop:
         mov rdi,format         ; printf with format...
         mov rsi,r15            ; print the r15
         mov rax,0              ; zero flag
         call printf            ; call printf

         inc r15                ; r15++
         cmp r15, r14           ; is r15==r14?
         jne loop               ; no - loop again

         ; exit routine
         mov       rax, 60                  ; system call for exit
         xor       rdi, rdi                 ; exit code 0
         syscall                            ; system call to exit

         section   .data
format:  db "v=%ld",10
         ; define constants here

         section   .bss
         ; define uninitialized variables here
```

```
mcbx@mwilkus:~/$ ./count
v=0
v=1
v=2
v=3
v=4
v=5
v=6
v=7
v=8
v=9
```

Using C libraries

- To build and run:

```
nasm -felf64 -l count.lst count.asm && gcc -no-pie -o count count.o && ./count
```

- Notice a few changes:

```
        global    main
        extern    printf
        section   .text
main:
        ; code goes here
        mov r15, 0
```

main – to get gcc know where the program starts

extern to be able to call external (gcc's) functions

-no-pie means that the executable is not position-independent. This way it is possible to jump almost into an entire executable scope, including this linked printf.

Now a small change ;-) …

```
bits 64
; Program counts 0..9

          global    main
          extern    printf
          section   .text
main:
          ; code goes here
          mov r8, 0
          mov r9, 10
loop:
          mov rdi,format
          mov rsi,r8
          mov rax,0
          call printf

          inc r8
          cmp r8, r9              ;
          jne loop

          ; exit routine
          mov       rax, 60
          xor       rdi, rdi
```

- The result:

```
v=0
v=2
v=2
v=2
v=2
v=2
v=2
v=2
v=2
v=2
v=2
v=2
v=2
v=2
v=2
v=2
v=2
v=2
v=2
v=2
```

What happened?

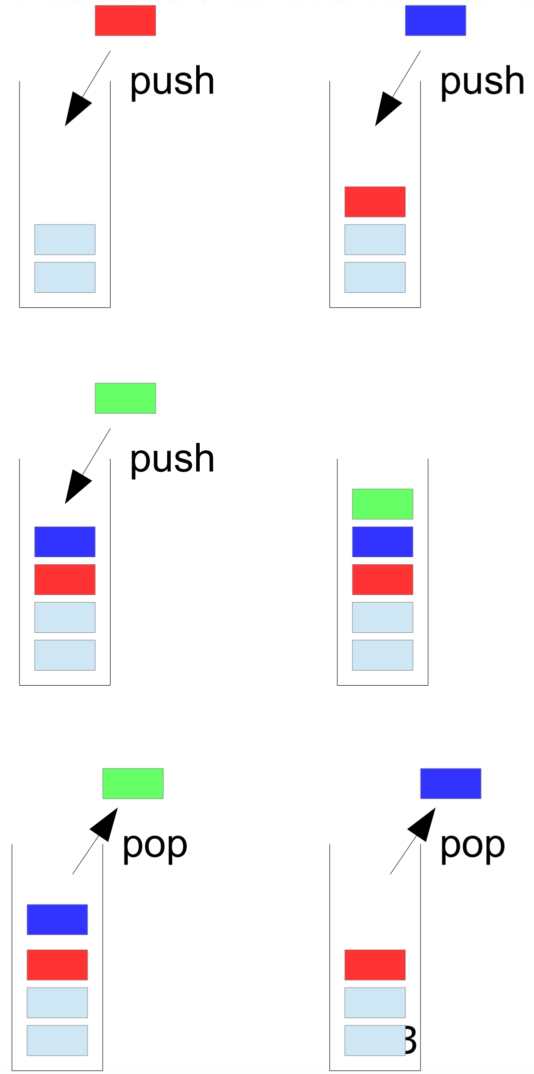
- We only changed the general purpose registers. Nothing else.
  - But these registers are not for our program only!
  - The function we call must use some registers too.

- The external function **used** our registers we were using for something and **overwritten** its values.
- Do we need to use memory?

- There is a space for temporarily storing such data and it is called a **stack**.

Stack as the data structure

- There are two operations: **push** to the stack and **pop** from the stack.
- We can **push** and **pop** values and registers.
- Initially, the stack contains the program name, argument count and arguments addresses.
- As in the stack of objects, the last thing gets in, it goes out first.



Let’s hold the registers on the stack

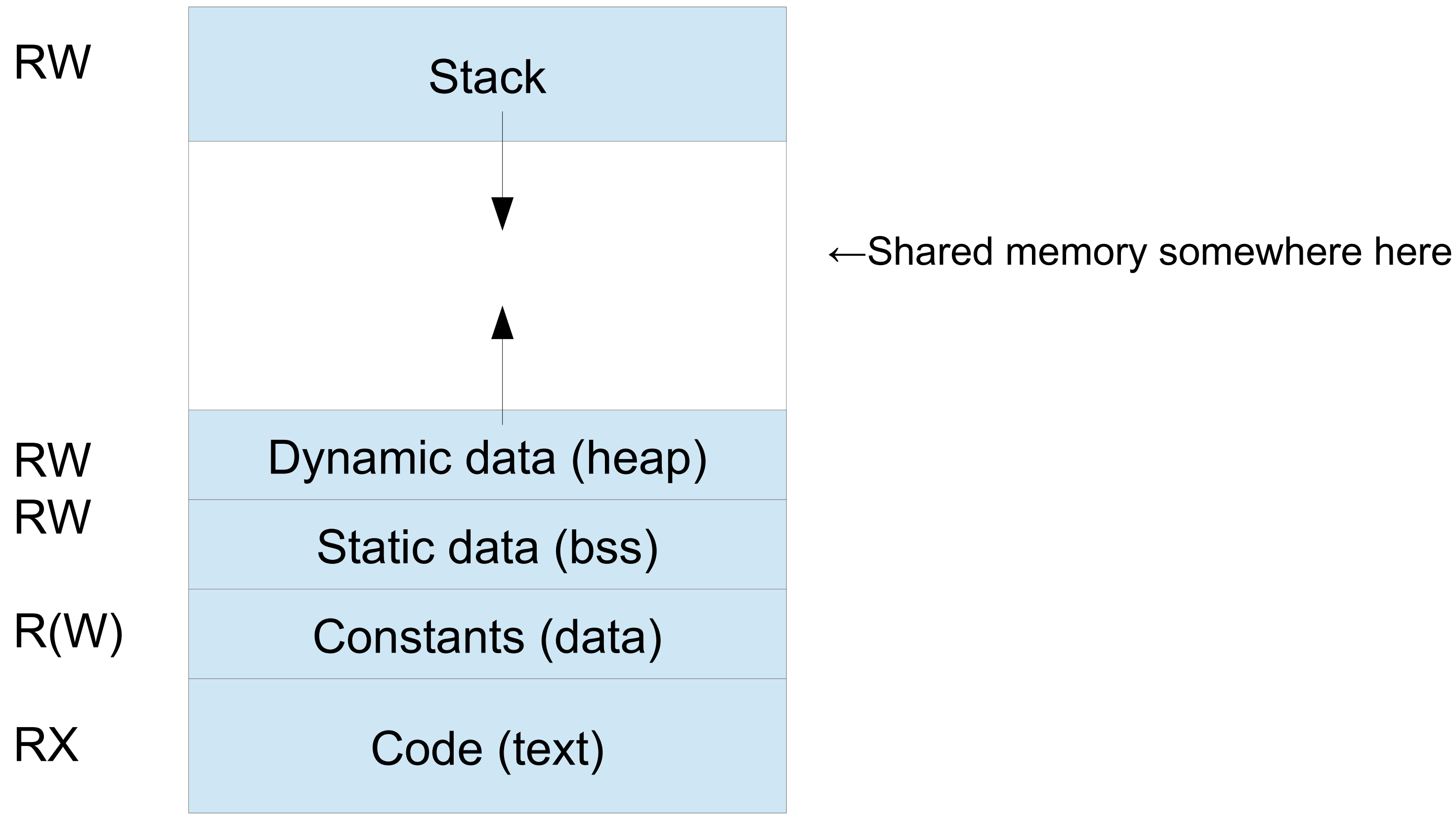
```
main:
        ; code goes here
        mov r8, 0               ; counter
        mov r9, 10              ; maximum
loop:
        push r8
        push r9

        mov rdi,format          ; printf with format...
        mov rsi,r8             ; print the r15
        mov rax,0               ; zero flag
        call printf             ; call printf

        pop r9
        pop r8

        inc r8                 ; r15++
        cmp r8, r9            ; is r15==r14?
        jne loop                ; no - loop again

        ; exit routine
        mov       rax, 60                   ; system call fo
        xor       rdi, rdi                  ; exit code 0
```

Notice the order we push and pop these registers!

Stack pointer

- In most architectures, the stack grows „upwards” - more items on stack → the higher value of the pointer to the top.
- In Intel, start of stack is pre-declared and it **grows backwards**, means, pushing a 64-bit register into it results in stack’s top being 8 bytes **lower.**
- Then the stack pointer (rsp register) **decrases**.
- The **base pointer** (rbp register) points to the start of the stack.

Program in the memory



Stack requirements

- When we call a function, the stack pointer must be **aligned** to the 16-bit boundary.
- The stack is aligned before making a call to the function? Great, but **calling a function makes it out of alignment** because it pushes the 8-bit return address to the stack.
- We have to **prepare** the stack before using when functions are called, or we will get the...

Preparing the stack

- So to use the stack reliably we have to:
  - Store the information where the stack begins - somewhere (the beginning of the stack is a good point, it is always there!)
  - ...so put a new base pointer to the new beginning of the stack.

```
        extern    printf
            section   .text
main:

    push rbp
    mov rbp, rsp  ; prepare the stack
```

- Most external functions work properly only if the stack is made this way - otherwise it may not be possible to return from called functions!

Preparing the stack

- After the stack is prepared, it nevertheless would be wise to make sure there are no solitary **push …** - as it will shift the stack pointer 8 bytes lower, where we want 16.

- A quick hack is to just push and pop something else. There is usually something we may want to save from messing up by function call.

- In the next example, I balanced this problem by making **three** stack operations before function call, aligning the misaligned (by 8 bytes) stack with **8+8+8** bytes.

Fibonacci sequence

$F_1=0$

$F_2=1$

$F_n=F_{n-2}+F_{n-1}$ **(when n>2 of course)**

- It can be calculated iteratively or recursively.
- Every next element grows very fast, so it will overflow a register quickly.
- Parts of this sequence appears surprisingly frequently in mathematics, physics, modelling.

Fibonacci sequence

```
        global    main
        extern    printf
        section   .text
main:
        ; code goes here
        mov r10, 10           ; iteration count
        mov r8, 1             ; current element
        mov r9, 1             ; previous element
loop:
        push r10
        push r8
        push r9               ; we may damage these - store them in stack

        mov rdi,format        ; printf with format...
        mov rsi,r8           ; print the r15
        mov rax,0             ; zero flag
        call printf           ; call printf

        pop r9                ; restore from stack
        pop r8
        pop r10

        mov r11, r8           ; temporarily store current
        add r8, r9            ; current = current + former
        mov r9, r11           ; former = ex-current

        dec r10               ; decrement counter
        jnz loop              ; nonzero - loop again

        ; exit routine
        mov       rax, 60                 ; system call for exit
        xor       rdi, rdi                ; exit code 0
        syscall                           ; system call to exit

        section   .data
format:  db "v=%ld",10
```

Stack store-restore of registers

Notice we can jump depending on result of operation **other than cmp!**

```
mcbx@m4800:~/Publiczny/0Dydaktyka/ICS/nasm$ ./fibonacci
v=1
v=2
v=3
v=5
v=8
v=13
v=21
v=34
v=55
v=89
mcbx@m4800:~/Publiczny/0Dydaktyka/ICS/nasm$
```

Floating point operations

Operation of an FPU

- Base x86 assembly has no FPU operations at all.
- All FPU operations have to be performed using a specific strategy:
  - If the operands are in the registers, store them somewhere else, e.g. in the memory.
  - Load the numbers from the program’s constants, data or memory into the FPU stack.
  - Perform the needed operation/operations.
  - Pop the results back from the FPU stack. Again, not to registers!
  - FPU stack has capacity of 8 operands.

FPU internal registers

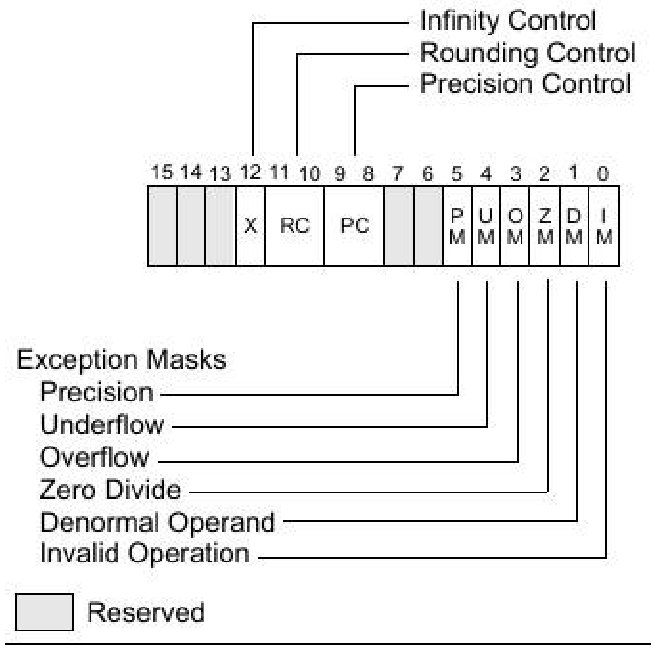
- There are 8 of them, describes as st(0)..st(7).
- From the CPU's point of view, they are accessible as a special stack. St(0) is current top of this stack.

- You can force an FPU to increase/decrease its "stack top" pointer (FINCSTP/FDECSTP), and it now cycles in 0..7 - like in some cyclic queue.
  - It means that you can "rotate" these registers having result on top, trading one memory access by one rotation instruction, but it doesn't mean too much with modern hardware.

- We finally ended with a structure, which, depending on how we look at it, looks like a stack, a cyclic queue or random-access registers.

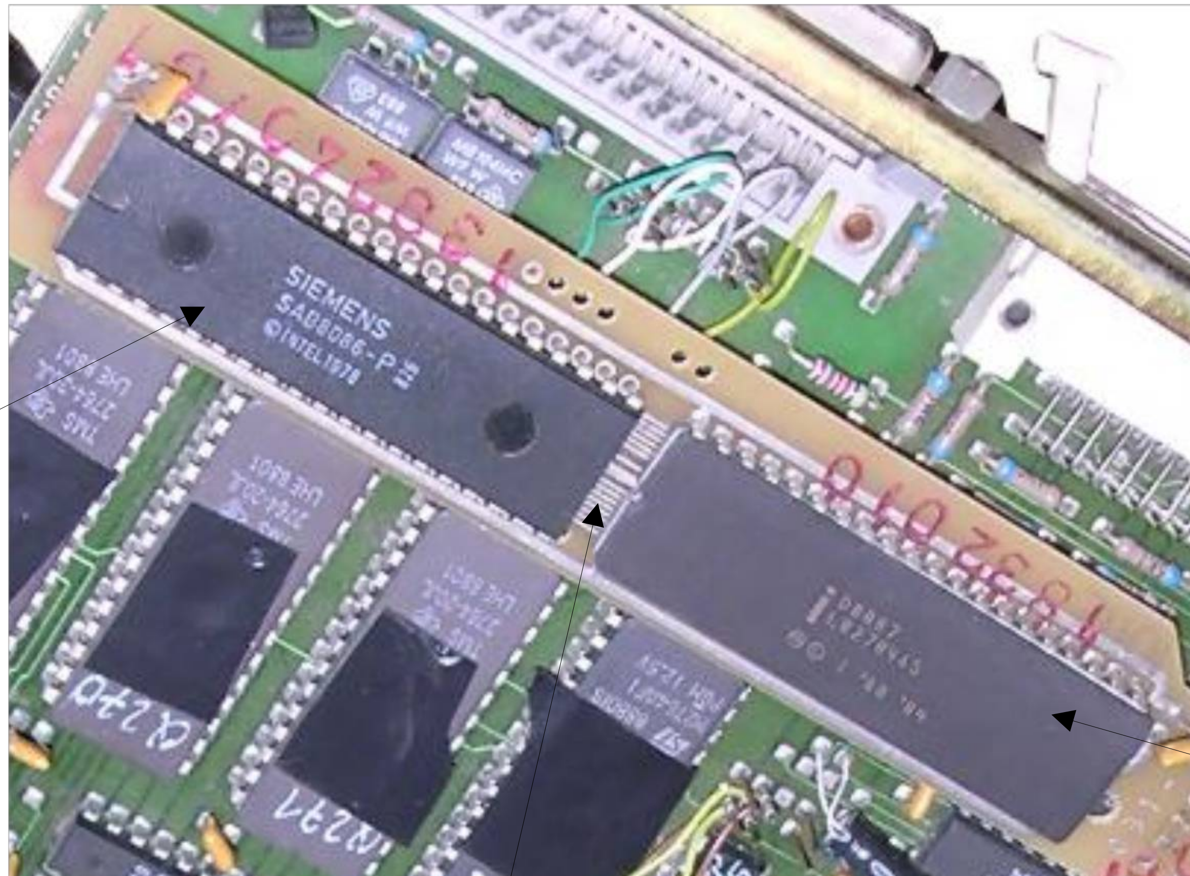
FPU control word

- Set to default on initialization.
- Exception masks: These bits, when set, make the FPU throw interrupt when something happens.
- Precision control:
  - 24bit (00),
  - Not used (some FPUs hang) (01),
  - 53bits (10),
  - 64bit (11) ← default.
- Rounding control:
  - To the closest value (to even if equal) (00), ←default
  - To lower (01)
  - To higher (10)
  - Truncating (11)
- Infinity control (unused), before 287 it was used to determine should there be just an ∞, or +∞ and -∞.



But… why?

CPU



FPU

Too slow for register transfers!

FPU caveats

- Depending on architecture, 32 or 64 bits.
- Internally, 80-bits. This allows to truncate precision errors.
- Some rare architectures allow to get wider registers.
- There are multiple floating-point representations for FPU, CPU and print-like functions and CPU has instructions to convert between them.
- Sometimes the floating-point number has to be put in a specific register to make function operate on it like in a floating-point.

Floating point computations end with errors.

- ...the objective is to minimize their influence on the result!
- Typical errors:
  - Some numbers are **not represented properly** in the system (you cannot put an entire *π* into the system!).
  - You may **run out of precision** or numbers are misrepresented.
  - If the computation program runs in multiple passes, and each pass adds more detail to the result, the computation may be **prematurely stopped** because of time or roundoff constraints.
  - Poor mathematical assumptions (especially in simulations!) - like „let the friction be zero”.
  - Human errors in algorithms.

The most important commands

- Now, when we loaded the data like in the stack, **FPU commands address the same data as registers.**

- finit - initialize the FPU.
- fld … - push (load) the number into FPU stack.
- fstp - pop the number from the FPU stack storing real number in the memory (fst will skip popping).
- Many arithmetic operands have „p” suffix which means „perform the operation **and pop the result from the FPU stack**”.

Arithmetic

- fsqrt - square roots the ST0 FPU register.
- fmul - multiplication (fdiv - division)
  - One operand - multiply ST0 by the operand and store it in ST0 (operand can be a constant or memory variable ([...]).
  - Two operands - multiply numbers by each other, store in the first one. But **one of the operands must be ST0.**
  - fmulp - pops the stack after multiplication.
- fsin, fcos - operate on ST0, write to ST0
- fadd, fsub - like fdiv, fmul.

Example:

```
          section   .text
main:
    push rbp
    mov rbp, rsp  ; prepare the stack
    finit ;initialize
    mov r8, 28  ; loop init
loop:
      push r8
      push r10
      movsd xmm0, qword [number]  ;load the flt1 into xmm0 register
      mov rdi,format        ; printf with format...
      mov al, 1
      call printf           ; call printf
      pop r10
      pop r8
                        ;Perform the square root operation:
      fld qword [number]            ;push the FLT1 into the fpu stack
      fsqrt                       ;perform the x=sqrt(x) on fpu stack
      fstp qword [result]         ;pop the result to ram
      movsd xmm0, qword [result]  ;load the result into xmm0 register
      movsd qword [number], xmm0    ;save the xmm0 register to the flt1

      dec r8
      jnz loop

      mov       rax, 60                 ; system call for exit
      xor       rdi, rdi                ; exit code 0
      syscall                           ; system call to exit

          section   .data
number:    dq  123.45 ; 1 qword for argument
format:  db "v=%f",10,0

          section   .bss
result: resq 1 ; 1 qword for result
```

Result:

- Squareroot the number 28 times (and we ran of precision so got 1.000000)

```
mcbx@m4800:~/Publiczny/0Dydaktyka/ICS/nasm$ ./FPU
v=123.450000
v=11.110806
v=3.333287
v=1.825729
v=1.351196
v=1.162409
v=1.078151
v=1.038340
v=1.018990
v=1.009450
v=1.004714
v=1.002354
v=1.001176
v=1.000588
v=1.000294
v=1.000147
v=1.000073
v=1.000037
v=1.000018
v=1.000009
v=1.000005
v=1.000002
v=1.000001
v=1.000001
v=1.000000
v=1.000000
v=1.000000
v=1.000000
```

A few important parts:

- Initialize the stack to point at proper boundary:

```
main:
    push rbp
    mov rbp, rsp  ; prepare the stack
```

- Because we're dealing with floating point numbers, now printf expects the data in the xmm0 wide (128bit) register:

```
movsd xmm0, qword [number]  ;load the flt1 into xmm0 register
mov rdi,format         ; printf with format...
mov al, 1
call printf            ; call printf
```

A few important parts…

- In the main calculation, we convert everything from/to qword to get rid of FPU's precision errors:

```
fld qword [number]              ;push the FLT1 into the fpu stack
fsqrt                         ;perform the x=sqrt(x) on fpu stack
fstp qword [result]           ;pop the result to ram
movsd xmm0, qword [result]    ;load the result into xmm0 register
movsd qword [number], xmm0      ;save the xmm0 register to the flt1
```

- Constants and data for FPU operation:

```
           section   .data
number:     dq  123.45 ; 1 qword for argument
format:   db "v=%f",10,0

           section   .bss
result: resq 1 ; 1 qword for result
```

Stack or register?

- The FPU is externally filled/emptied **as a stack**.
- The numbers can be internally processed **as a set of registers**.
- However it is implemented **as a set of shift registers holding 80-bit numbers at once.**
- It means that if we load (fld) two numbers, always the last one is the ST0.
- Now: while it is possible to „shift left” the stack to the previously pushed value, pushing any value next would result in the value trying to be written over the ST0.
- **This will shift the stack properly, but destroy the ST0 contents!**

So one more time

- FLD - Load into the ST0 - previous ST0 becomes ST, ST1 becomes ST2 etc.
- FILD - Load to ST0 as integer.
  - FLDPI - load Pi to ST0.

- FST - Store the ST0 into the operand (memory address or ST register)
- FSTP - As above, but pop the ST0 from stack.
- FIST - FST, but converts the number to integer.
- FISTP - As above, but pops the value.

Other useful instructions

- FABS - Absolute value of ST0
- FCHS - Change sign of ST0
- FRNDINT - Round ST0 to integer
- FINIT used after another FINIT - resets the FPU totally, including clearing the stack.

- FYL2X - 2-base logarithm: $ST1=ST1*\log_2(ST0)$

- FCOM - Compare 2 operands, at least one must be ST.

FCOM considerations

- The comparison is in the FPU, and the code is executed by the CPU.
- It is needed to **transfer** the result of comparison from FPU status register to CPU's status register:

```
fcom       ; compare
fstsw ax   ; store FPU's status register to AX
sahf       ; store AH register to CPU flags
```

SSE Extension

SSE

- Streaming SIMD Extensions.
- Introduced in 1999 with Pentium III processor.
- Allows to perform opreations on 4 floats at once (packed in a 128-bit special XMM registers).
  - Or 2 doubles, or 2 floats stored as doubles.

- Applications:
  - Multimedia (en/decoding),
  - Signal processing (SSE2 has DSP instructions)
  - 3D graphics,
  - Scientific computation,

XMM registers

- 128-bit wide,
- Initially 8, in 64-bit architecture 16 of them,
- Can keep 4 32-bit floats,
- In SSE2, it is also possible to keep and process two 64-bit doubles, two 64-bit integers or four 32-bit integers.
- More rarely, it is possible to keep 8 16-bit integers or 16 8-bit integers.

SSE instructions

- There are two kinds of instructions:
  - Packed - perform the same operation on each of the number in packed register (example: MULPS):

1	*	9	=	9
2	*	8	=	16
3	*	7	=	21
4	*	6	=	24

  - Scalar - only the first number is processed (MULSS):

1	*	9	=	9
2		8		2
3		7		3
4		6		4

SSE: Example

```
main:
    push rbp
    mov rbp, rsp  ; prepare the stack

        movapd xmm0, [vec1]
        movapd xmm1, [vec2]     ; load the numbers

        mulpd xmm0, xmm1        ; perform the operation

        movapd xmm1, xmm0       ; copy xmm0 to xmm1
        unpckhpd xmm1, xmm0     ; get higher double from xmm0 to xmm1

        mov rdi,format        ; printf with format...
        mov al, 2             ; one number (printf expects numbers in bottom of xmm0, xmm1 etc)
        call printf           ; call printf

          ; exit routine
          mov       rax, 60                 ; system call for exit
          xor       rdi, rdi                ; exit code 0
          syscall                           ; system call to exit

          section   .data
format:  db "v=%f %f",10
align 16
vec1: dq 1.2, 2.5
align 16
vec2: dq 5.0, 6.0
```

SSE: Result:

```
ncbx@m4800:/tmp$ nasm -felf64 -l sseasm.lst sseasm.asm; gcc -no-pie -o ss
/=6.000000 15.000000
```

Two floating point numbers get multiplied with a single command.

```
mulpd xmm0, xmm1          ; perform the operation
```

SSE: Important things

- It is needed to pack and unpack values before/after executing SSE instructions.

- The types must be maintained all time.
  - However, you can use e.g. double-based calculus for floats if you align them properly.
  - There are instructions for aligned and unaligned data (like movaps/movups for aligned/unaligned singles). This way it is possible to align singles the way that they are considered as doubles.

SSE: Arithmetic operations

- MULPS, MULPD - packed multiplication.
- MULSS MULSD - scalar multiplication.
- ADD[P/S][S/D] - addition.
- SUB[P/S][S/D] - subtraction.

- SQRT[P/S][S/D] - Square root.
  – WARNING: SQRT..S is guaranteed to work all time. The double operations are available in newer CPUs (>=Pentium 4).

SSE: Packing/unpacking

- MOVUPS/MOVUPD - move unaligned data as floats/doubles.
- MOVAPS/MOVAPS - move aligned data as floats/doubles.

- UNPCKHPD - Unpack higher double
- UNPCKHPS - Unpack higher float
- UNPCKLPD/UNPCKLPS - a similar one.

Thank you for attention