Assembly Coding 2024

introduzione al vostro prossimo hobby preferito

Sandro "guly" Zaccarini, Diego "freshness" Barzon

21/09/2024 - EndSummerCamp

who we are

Sandro "guly" Zaccarini

fix if broken, break if works

(wine|food|security|martial) artist

https://github.com/theguly

Diego "freshness" Barzon

somewhat miscellaneous useful computer related task for making the ends meet

useless passion-leaded computer stuff to fill the free time holes

https://github.com/freshness79

1-minute intro to assembly

- not a language, but a set of instructions
- such instructions set is CPU dependent (eg: arm != x86)
- closest to hardware, if you don't want to code with binary numbers
- no (or very little) abstraction

1-minute intro to assembly - data

- registers
  - accumulator, counter, general, instruction pointer, ...
  - flags
  - speed/size
- memory:
  - code: not read-only ;)
  - data
  - stack versus heap

C source - linear disassembly

```c
// gcc -m32 -g hello.c -o hello
#include <stdio.h>
int main () {
  printf("Hello World!\n");
  return 0;
}
```

```
00401146  int32_t main(int32_t argc, char** argv, char** envp)

00401146  55                push    rbp {__saved_rbp}
00401147  4889e5            mov     rbp, rsp {__saved_rbp}
0040114a  bf04204000        mov     edi, 0x402004{"Hello World!"}
0040114f  b800000000        mov     eax, 0x0
00401154  e8d7feffff        call    printf
00401159  b800000000        mov     eax, 0x0
0040115e  5d                pop     rbp {__saved_rbp}
0040115f  c3                retn     {__return_addr}
```

use cases

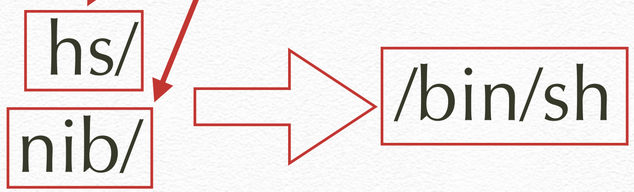
- pwn/exploit
- reverse engineering
- high performances
- embedded/very low resources
- very specific tasks

use case > pwn

- when you wear a black hat you usually have very low resources
- custom assembly routines are often the only way

use case > pwn > shellcode

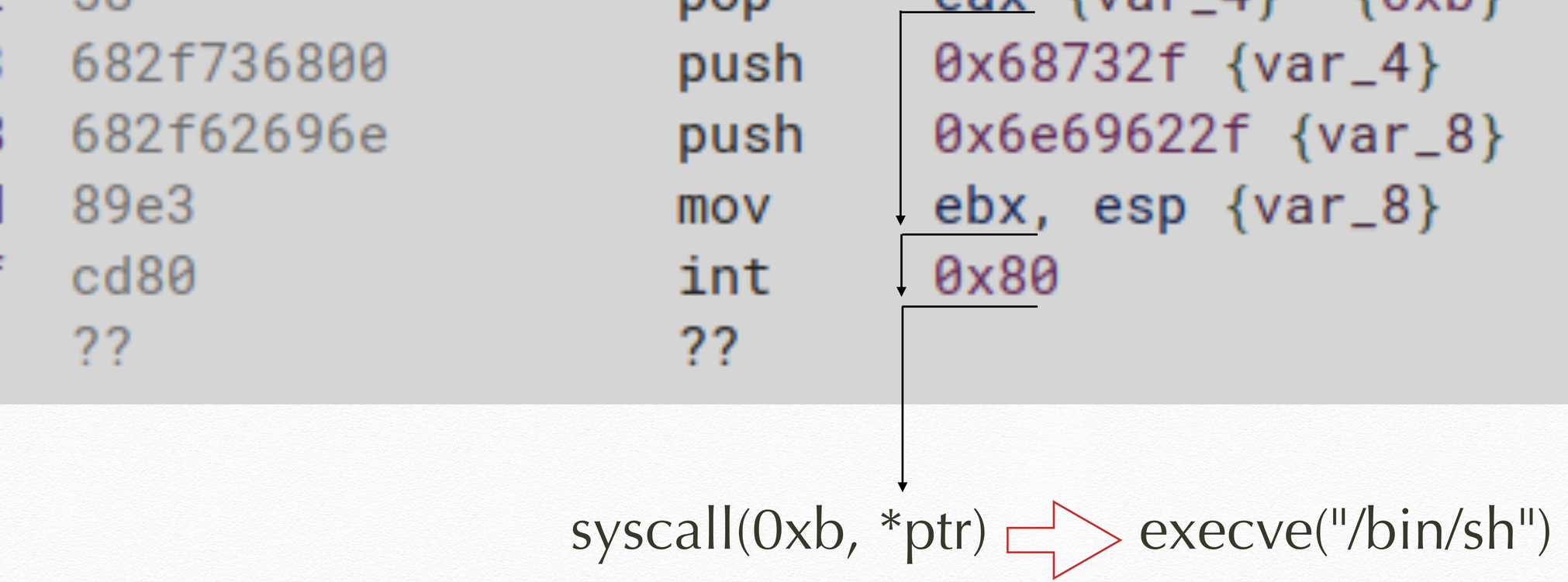
```
08049000  6a0b          push    0xb {var_4}
08049002  58            pop     eax {var_4}  {0xb}
08049003  682f736800    push    0x68732f {var_4}
08049008  682f62696e    push    0x6e69622f {var_8}
0804900d  89e3          mov     ebx, esp {var_8}
0804900f  cd80          int     0x80
08049011  ??            ??
```



source: https://www.exploit-db.com/shellcodes/49768

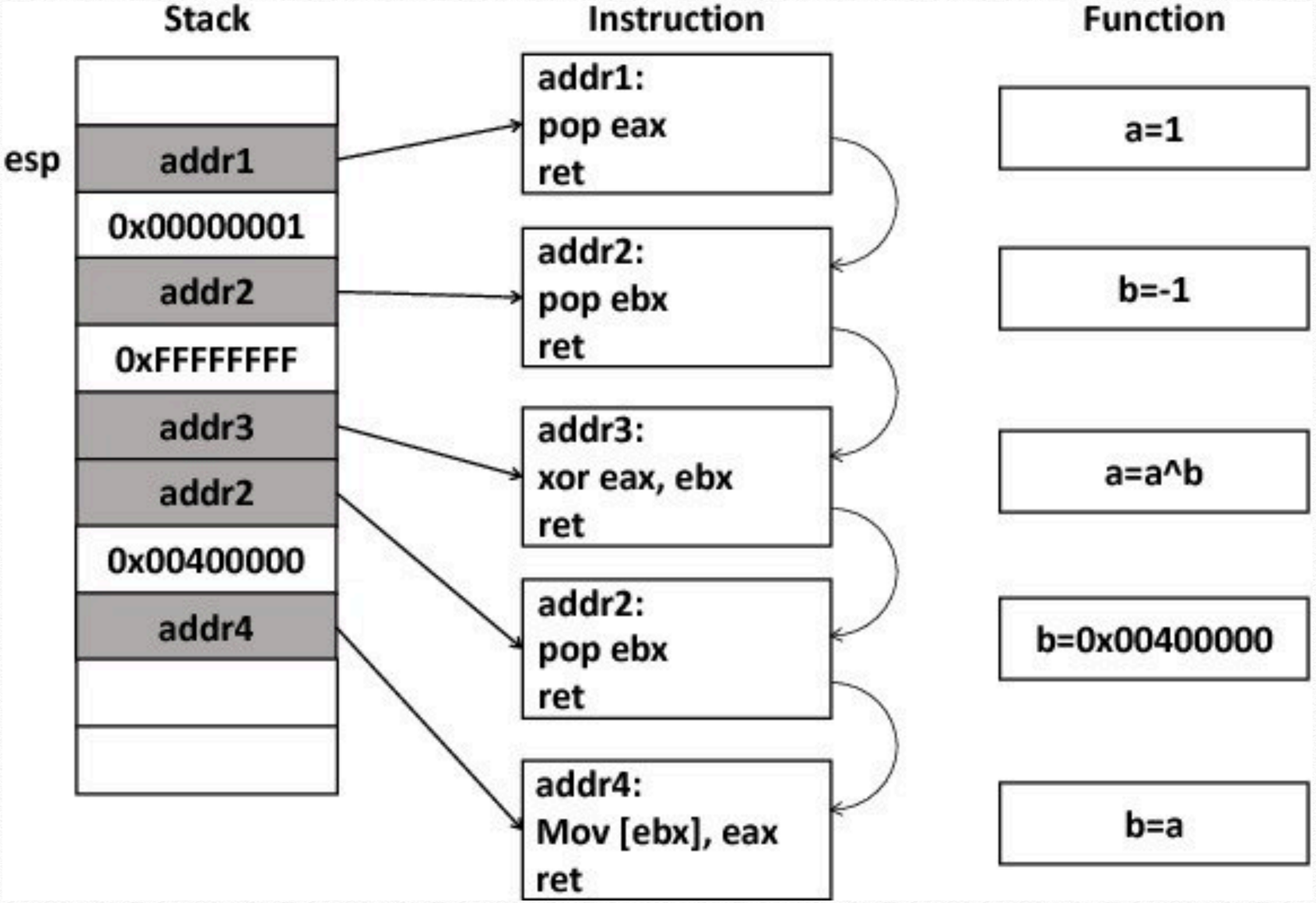
use case > pwn > shellcode

```
08049000  6a0b          push    0xb {var_4}
08049002  58            pop     eax {var_4}  {0xb}
08049003  682f736800    push    0x68732f {var_4}
08049008  682f62696e    push    0x6e69622f {var_8}
0804900d  89e3          mov     ebx, esp {var_8}
0804900f  cd80          int     0x80
08049011  ??            ??
```



source: https://www.exploit-db.com/shellcodes/49768

use case > pwn > rop



use case > pwn > rop

```
$ ropper -f target --nocolor
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
0x080484eb: pop ebp; ret;
0x080484e8: pop ebx; pop esi; pop edi; pop ebp; ret;
0x080482ed: pop ebx; ret;
0x080484ea: pop edi; pop ebp; ret;
0x080484e9: pop esi; pop edi; pop ebp; ret;
0x08048484: popal; cld; ret;
```

use case > pwn > rop

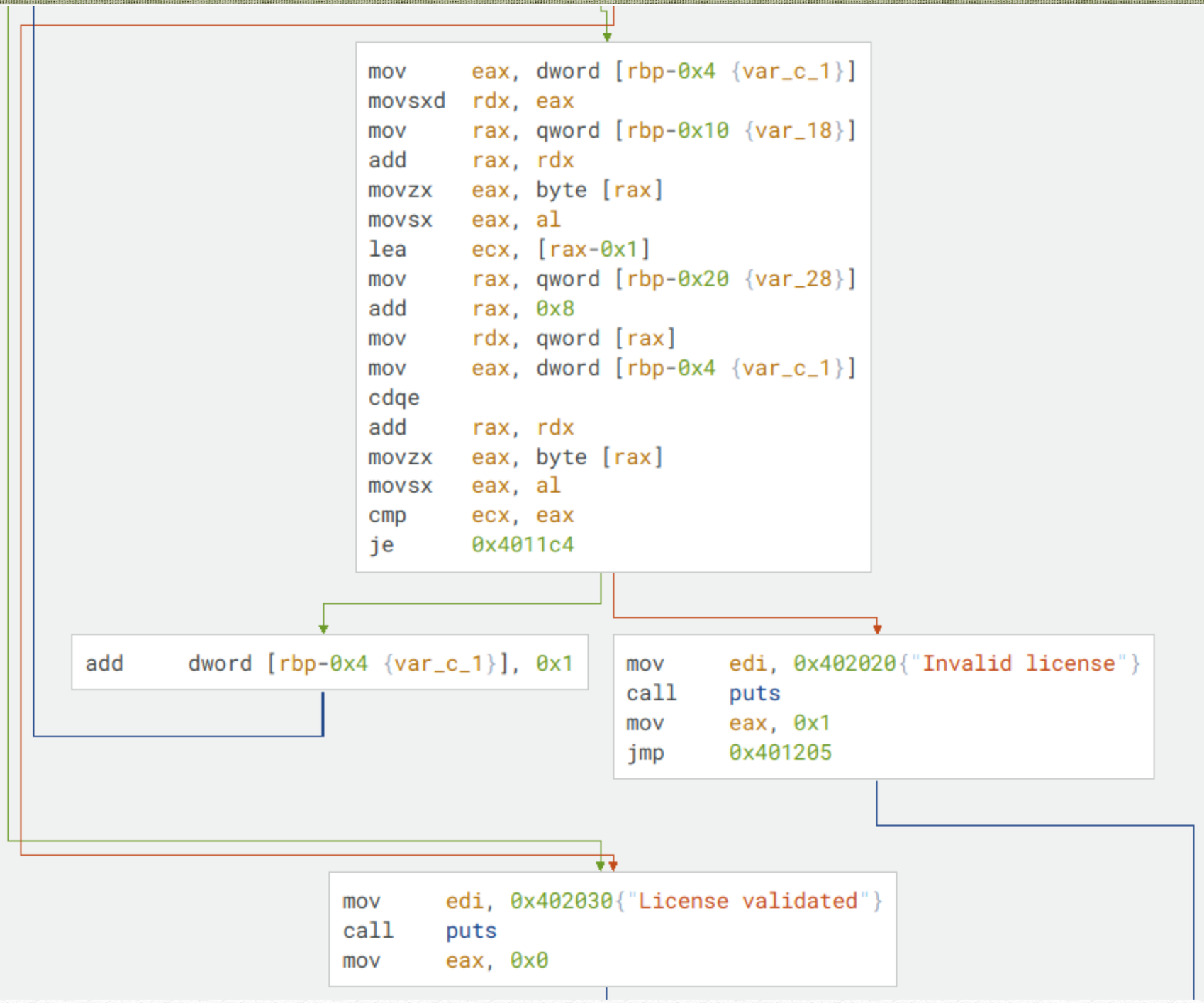
```
# size 0x21000 in ecx
buf = p32(0x080484bd)                  # pop ecx; pop edx; ret
buf+= p32(0x01031101)
buf+= p32(0x01010101)
buf+= p32(0x08048332)                  # sub ecx, edx; not eax; and eax, ecx; ret;

# 7 in edx
buf+= p32(0x0804831d)                  # xor eax, eax; ret;
buf+= p32(0x08048330)                  # xchg eax, edx; ret;
buf+= p32(0x080482a4 ) * 7             # inc edx; ret;

# stack in ebx
buf+= p32(0x080482ed) + stack_addr     # pop ebx; ret;
buf+= p32(0x080482b6)                  # dec ebx; ret;

buf+= mprotect_addr                    # mprotect(addr, len, permissions)
```

use case > reversing

- patching/fix
- malware/ransomware decrypt0r
- underdocumented/undocumented features
- decompiling

use case > reversing > patching

- recompile is not an always option...
- ...especially if source code is not available
- by mastering assembly, you *could* replace small pieces of a binary
- "everything is open source if you can reverse engineer"


mov eax, dword [rbp-0x4 {var_c_1}]
movsxd rdx, eax
mov rax, qword [rbp-0x10 {var_18}]
add rax, rdx
movzx eax, byte [rax]
movsx eax, al
lea ecx, [rax-0x1]
mov rax, qword [rbp-0x20 {var_28}]
add rax, 0x8
mov rdx, qword [rax]
mov eax, dword [rbp-0x4 {var_c_1}]
cdqe
add rax, rdx
movzx eax, byte [rax]
movsx eax, al
cmp ecx, eax
je 0x4011c4
add dword [rbp-0x4 {var_c_1}], 0x1
mov edi, 0x402020{"Invalid license"}
call puts
mov eax, 0x1
jmp 0x401205
mov edi, 0x402030{"License validated"}
call puts
mov eax, 0x0

use case > reversing > ransomware

- https://decoded.avast.io/threatresearch/decrypted-akira-ransomware/
- https://github.com/srlabs/black-basta-buster

use case > reversing > ReactOS

- "Imagine running your favorite Windows applications and drivers in an open-source environment you can trust"
- a bunch of people disassembly Windows DLL/Drivers and documents what they understand

source: https://reactos.org/

use case > reversing > OpenDUNE

- take an abandonware
- write a decompiler
- decompile runtime
- conquer arrakis

source: https://github.com/OpenDUNE/OpenDUNE/issues/228

use case > reversing > OpenDUNE

- take an abandonware
- write a decompiler
- decompile runtime
- conquer arrakis

```
 7  /**
 8   * Decompiled function f__0070_0040_0005_1DBE()
 9   *
10   * @name f__0070_0040_0005_1DBE
11   * @implements 0070:0040:0005:1DBE ()
12   *
13   * Called From: 0000:0000:0000:0000
14   * Called From: 2756:0623:0006:2DB2
15   */
16  void f__0070_0040_0005_1DBE()
17  {
18      emu_syscall(0x8);
19
20      /* Return from this function */
21      emu_pop(&emu_ip);
22      emu_pop(&emu_cs);
23      emu_popf();
24      return;
25  }
```

source: https://github.com/OpenDUNE/OpenDUNE/issues/228

use case > high performances

- especially when hardware upgrade is not an option, asm snippets can still be faster or smaller than compilers made code
- gaming console are a good example: on a specific platform the hardware can't be improved

use case > high performances > microcontroller

- when resources are limited, compiler may not be clever enough to fit all needed code
- microcontrollers are literally everywhere, and they are often slow and small in terms of memory

use case > high performances > AoE

Think Assembly code is useless to learn?
Age of Empires I+II used ~13,000 lines of x86 32-bit assembly code.
"The use of assembly in the drawing core resulting in a ~10x sprite"

source: https://twitter.com/lauriewired/status/1744063043716399290
orginal: https://www.reddit.com/r/aoe2/comments/18ysttu/

use case > high performances > RCT

Sawyer wrote 99% of the code for RollerCoaster Tycoon in assembly code for the Microsoft Macro Assembler, with the remaining one percent written in C.

source: https://en.wikipedia.org/wiki/RollerCoaster_Tycoon_(video_game)

use case > high performances > KolibriOS

"KolibriOS is a tiny yet incredibly powerful and fast operating system. This power requires only a few megabyte disk space and 8MB of RAM to run. Kolibri features a rich set of applications that include word processor, image viewer, graphical editor, web browser and well over 30 exciting games. Full FAT12/16/32 support is implemented, as well as read-only support for NTFS, ISO9660 and Ext2/3/4. Drivers are written for popular sound, network and graphics cards. [CUT] This speed is achieved since the core parts of KolibriOS (kernel and drivers) are written entirely in FASM assembly language!"

source: https://www.kolibrios.org/

use case > specific tasks

- specific routines
- hardware requirements

use case > specific tasks > C vs ASM

```c
// Test C function
printf("Starting C optimized function:\n");
  for(int i=0;i<64000;i++)
      test2[i] = 0;
c_start = clock();
for(long i=0;i<50000;i++) {
  unsigned int data;
  ((unsigned char *)&data)[0] = 0;
  ((unsigned char *)&data)[1] = 64;
  ((unsigned char *)&data)[2] = 128;
  ((unsigned char *)&data)[3] = 192;
  for(int i=0;i<16000;i++) {
      ((char *)&data)[0] += 1;
      ((char *)&data)[1] += 1;
      ((char *)&data)[2] += 1;
      ((char *)&data)[3] += 1;
      ((unsigned int *)test2)[i] = data;
    }
}
c_end = clock();
  total = (float)((c_end - c_start)/50000);
printf("Done - Elapsed: %5.1f\n",total);
```

```asm
BITS 64
; RDI has "test" 64 bit buffer pointer (standard arg passing)
; We need to save only RBX, which must not be clobbered
; across function calls (RBX, RBP, R12-R15)

  push rbx
  mov dx, 0xc080
  mov ebx, 0x4000
  mov ecx, 16000
mainloop:
  add dl,1
  add dh,1
  mov eax,edx
  shl eax,16
  add bl,1
  add bh,1
  add eax,ebx
  stosd
  dec ecx
  jne mainloop
  pop rbx
  ret
```

46seconds vs 31seconds => ASM is 32% faster

use case > specific tasks > CPU

- all modern OS running on Intel CPUs executes the so called "Protected Mode Jump"
- other platforms require similar operation when booting
- those ops are mostly carried out by small pieces of assembly code

source: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/boot/pmjump.S?h=v4.14-rc8

use case > specific tasks > emulator

- Z80 emulator written in ARM assembly for Nintendo DS:

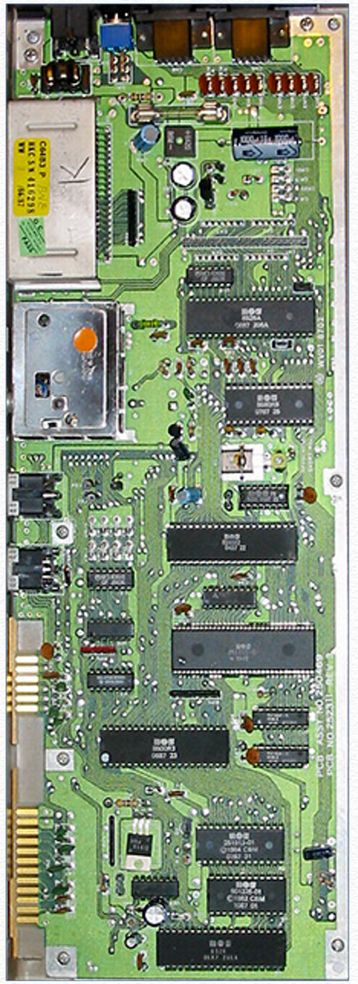
https://github.com/FluBBaOfWard/ARMZ80/blob/main/ARMZ80.s

demo time

- demo1: abuse instruction behavior to overcome hardware limits
- demo2: playable arkanoid in 506bytes

demo > platform

- CPU: 6510 (6502)
  - ~1Mhz
  - 8bit
  - 56(151) instructions
- RAM 64Kb (a bit more...)
- Video: 320x200, 16 colors

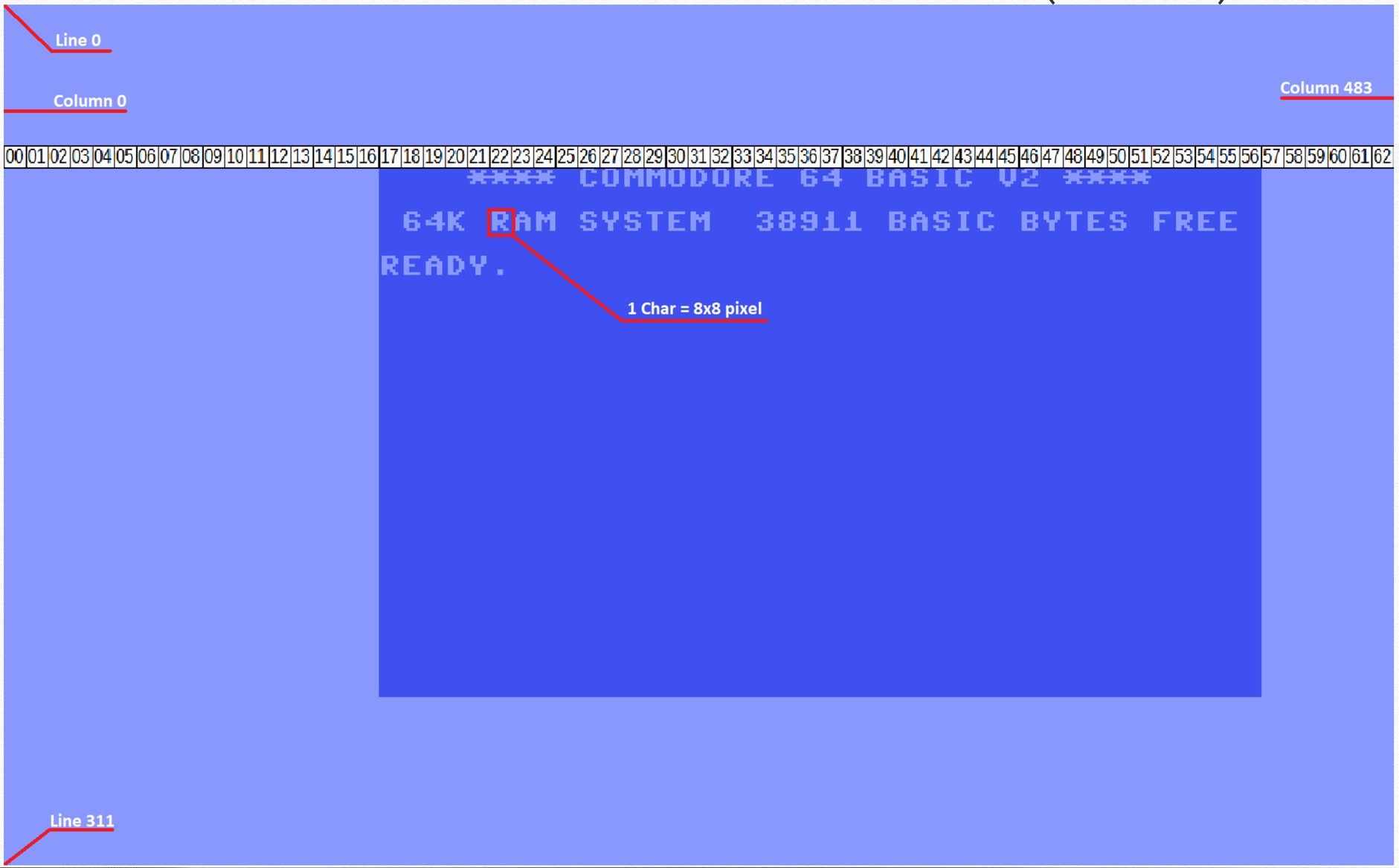


demo > instruction timings

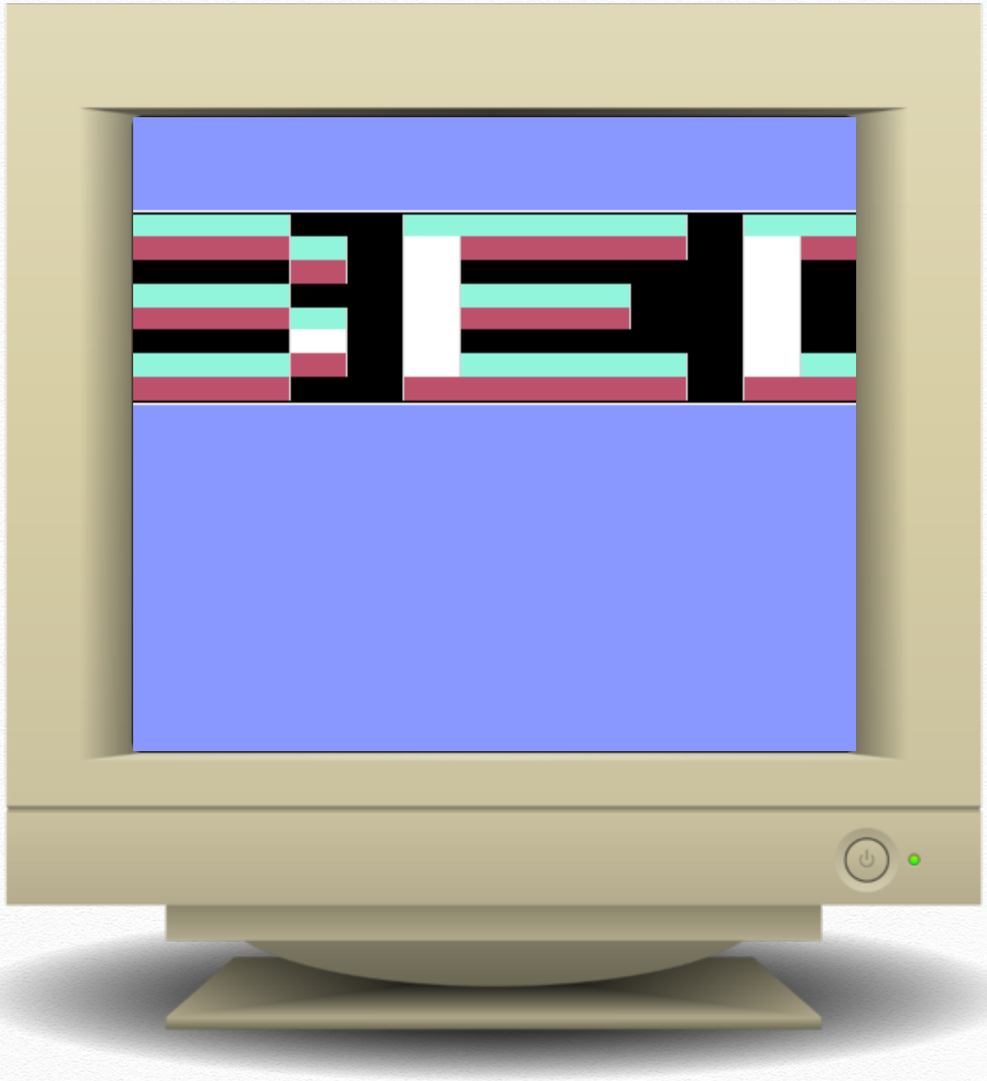
Instruction cycles: 2 - 8

2. NOP, LDA #imm, CLC, BEQ (*)
3. LDA zp, BIT zp, BEQ (*)
4. LDA abs, STA abs, LDA abs,x, BEQ (*)
5. INC zp, STA abs,x, LDA abs,x (**)
6. INC abs, DEC abs
7. INC abs,x, DEC abs,x
8. ISC (zp),y, DCP (zp),y

demo > video details (PAL)



demo > scroller



demo > synchronization loop

Always cycle 3 to 10

62 cycles, a bit less than a line

```
42  // Synchronize timer to raster (countdown 6 => 0)
43  synctimer:
44      lda #$00
45      sta $dc0e           // Stop timer
46      lda #$08
47      sta $dc04
48      lda #$00            // Set timer to 9 cycles
49      sta $dc05
50      ldx #$ff            // Wait for lower part of the screen
51  waitlowerscreen:
52      cpx $d012           // 4
53      bne waitlowerscreen // 3/2 -> Max jitter 7 cycles
54
55      // Sync loop: 62 cycles
56  waitline:
57      ldy #10             // 2
58  waitcycles:
59      dey                 // 2    Loop: (5*10)-1  = 49
60      bne waitcycles      // 3/2
61      lda #$11            // 2
62      inx                 // 2
63      cpx $d012           // 4
64      beq waitline        // 3/2
65      // Next instruction on cycle 3
66
67      nop         // 2
68      nop         // 2
69      nop         // 2
70      sta $dc0e   // 4 Start timer
71      rts
```

demo > synced loop

```
75   // -> Explicit 63 cycles block with different branches
76   .align $0100
77   syncedblock:
78       ldx #$00
79
80   openloop:
81       cpx upperline       // 3
82
83       bne !+              // 2/3 - 17
84       nop                 // 2  - If upper line reached
85       nop                 // 2
86       nop                 // 2
87       lda #$06            // 2  - Use color 06 (Blue)
88       bne bgsetcolor      // 3
89   !:
90       cpx lowerline       // 3
91       bne !+              // 2/3
92       lda #$0e            // 2  - If lower line reached
93       bne bgsetcolor      // 3  - Use color 0e (Light blue)
94   !:
95       bit $ff             // 3
96       clc                 // 2
97       bcc nosetcolor      // 3  - No color changes on all other lines
98   bgsetcolor:
99       sta $d021           // 4
100
```

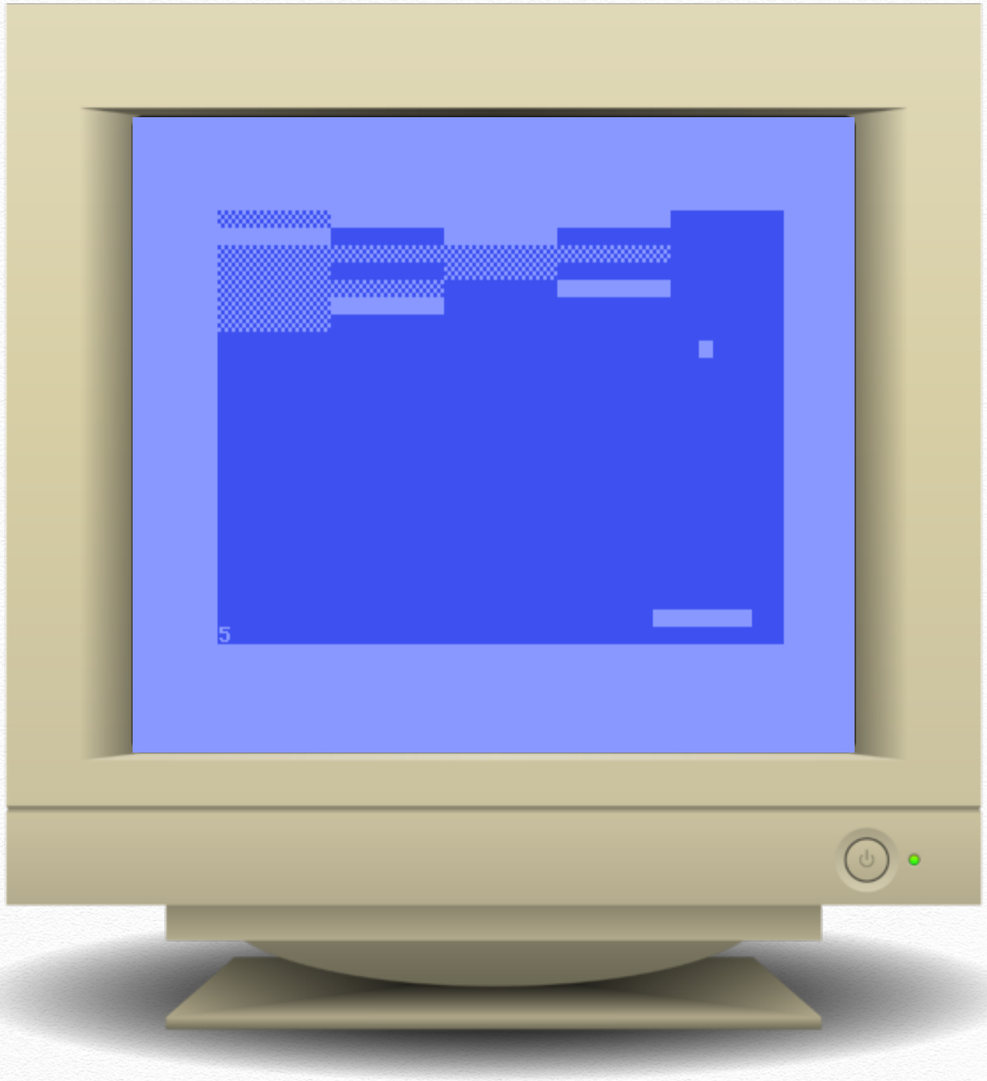
```
101  nosetcolor:
102      cpx #$03            // 2  - 18
103      bcs !+              // 2/3
104      nop                 // 2  - Enable 24 rows screen mode
105      bit $ff             // 3
106      lda #$00            // 2
107      beq openscreen      // 3
108  !:
109      cpx #$c5            // 2
110      bcs !+              // 3/2
111      lda stophs:#$08     // 2
112      bpl openscreen      // 3
113  !:
114      lda #$00            // 2
115      nop                 // 2 - Enable 25 rows screen mode
116  openscreen:
117      sta $d011           // 4
118
119      .fill 9,NOP         // 18  - 25 - Waste some cycles
120      inx                 // 2
121      cpx #205            // 2
122      bne openloop        // 3 - Loop for the whole screen
123
124
125
126
```

demo > synced loop

```
309    // Speed code will be created this way:
310    // sta $d020,x  // 5 - 63 (5 + 13*4 + 6)
311    // stx $d020    // 4
312    // sty $d020    // 4
313    // sax $d020    // 4
314    // sta $d020    // 4
315    // sta $d020    // 4
316    // stx $d020    // 4
317    // sta $d020    // 4
318    // sty $d020    // 4
319    // sta $d020    // 4
320    // sax $d020    // 4
321    // sty $d020    // 4
322    // stx $d020    // 4
323    // sax $d020    // 4
324    // rts          // 6
```

```
sta4cycles:
.byte BIT_ABS, STX_ABS, STY_ABS, SAX_ABS
.byte STA_ABS, BIT_ABS, STY_ABS, SAX_ABS
.byte STA_ABS, STX_ABS, BIT_ABS, SAX_ABS
.byte STA_ABS, STX_ABS, STY_ABS, BIT_ABS
sta5cycles:
.byte STA_ABSX, SHX_ABSY, SHY_ABSX, SHA_ABSY
```

demo > PETSCIInoid



demo > multiplication

- start conditions:
  - ballCoordHiTemp_2: current Y coordinate
  - A register: ballCoordHiTemp_2
  - DrawPTR: $0100

```
    // Multiply Y coord by 40
    asl                         // x2
    asl                         // x4
    adc ballCoordHiTemp_2 + 1   // x5
    asl                         // x10
!:
    asl                         // x20 and x40
    rol drawPtr + 1         // Shift hi byte to store carry and check higher bit to quit loop
    bcs !-
    sta drawPtr             // store lo byte
```

```
lda #WRITERINIT // All bits will be used for looping
sta writer      // but the variable will also store the lower bit of ball pattern
ldx #BALLPTRH   // 2 highest bits are used for looping,
stx drawPtr+1   // all remaining to store HI-byte of ball screen address
```

demo > every bit counts (1)

```
30    .const WRITERINIT = %10101000
31    .const BALLPTRH = %10000000 + (SCREENPAGE/$400)
367   // Prepare ball screen PTR and read current value
368   // ##########################################
369   // This is used for:
370   // - Drawing ball
371   // - Checking obstacles
372   // - Drawing wall
373   prepareBallPtr:
374       // TOPIC: mess with bits!
375       lda #WRITERINIT // All bits will be used for looping
376       sta writer      // but the variable will also store the lower bit of ball pattern
377       ldx #BALLPTRH   // 2 highest bits are used for looping,
378       stx drawPtr+1   // all remaining to store HI-byte of ball screen address
379
380   !:  // Prepare pattern based on fine tune bit
381       lda ballCoordHi - BALLPTRH,x        // Get least significant bit of the hi-byte of X,Y ball coords
382       lsr
383       sta ballCoordHiTemp_2 - BALLPTRH ,x
384       rol writer      // Store them in "writer" by shifting
385       inx
386       bcs !-          // Use higher bits of "writer" to exit at second iteration
```

```
410       // Draw ball pattern on screen
411   updateBall:
412       jsr prepareBallPtr
413       ldx writer
414   drawBallLoop:
415       lda ballchars-(<(WRITERINIT*4)), x
416       eor (drawPtr),y
417       sta (drawPtr),y
418       txa
419       axs #-4
420       iny
421       asl writer
422       bcs drawBallLoop
423       bpl quitPrepareBallPtr
424       tya
425       adc #(SCREENWIDTH-2)
426       tay
427       bcc drawBallLoop
428       inc drawPtr + 1
429       bcs drawBallLoop
```

demo > every bit counts (2)

```
319   // Bar update routine
320   // ##################
321   barControl:
322       ldx barPosition
323       lda JOYSTICK2   // Load Joystick port 2
324       lsr
325       lsr             // Skip bit 0-1
326       lsr
327       bcc barRight    // If bit 2 is clear => move right
328       lsr
329       bcs barSkip     // If bit 3 is clear => move left
330   barLeft:
331       cpx #((XYLIMIT - BARLENGHT) * 4)    // Right border
332       bcs barSkip
333       txa
334       // using illegal
335       axs #($100-BARSPEED*2)
336   barRight:
337       txa             // Left border
338       beq barSkip
339       axs #BARSPEED
340   barSkip:
341       stx barPosition
```

JOYSTICK2 ($DC00)

7	6	5	4	3	2	1	0
			Fire	Right	Left	Down	Up

X = (A & X) - #imm

demo > linear feedback shift register

Pattern	0	0	0	1	0	0	0	1	
Seed	0	0	0	0	0	0	0	1	
LSR	0	0	0	0	0	0	0	0	1
EOR	0	0	0	1	0	0	0	1	
LSR	0	0	0	0	1	0	0	0	1
EOR	0	0	0	1	1	0	0	1	
LSR	0	0	0	0	1	1	0	0	1
EOR	0	0	0	1	1	1	0	1	
LSR	0	0	0	0	1	1	1	0	1
EOR	0	0	0	1	1	1	1	1	
LSR	0	0	0	0	1	1	1	1	1
EOR	0	0	0	1	1	1	1	0	
LSR	0	0	0	0	1	1	1	1	0
LSR	0	0	0	0	0	1	1	1	1
EOR	0	0	0	1	0	1	1	0	
LSR	0	0	0	0	1	0	1	1	0
LSR	0	0	0	0	0	1	0	1	1
EOR	0	0	0	1	0	1	0	0	
LSR	0	0	0	0	1	0	1	0	0

```
170         lsr
171         bcc lfsr1
172         eor #LFSRPATTERN
173     lfsr1:
174         lsr
175         bcc lfsr2
176         eor #LFSRPATTERN
177     lfsr2:
178         pha
```

demo > SP *is* a register

```
269 invertSpeed:
270     txs                 // Abuse stack pointer to detect inversion, $ff means no inversion
271     ldx #$01
272 invertSpeedLoop:
273     lsr ballInvertFlags,x
274     bcc invertSpeedSkip
275     txs                 // $00 or $01 means inversion
276     lda #$00
277     sbc ballSpeedLo,x
278     sta ballSpeedLo,x   // Carry always clear here
279     lda #$00
280     sbc ballSpeedHi,x
281     sta ballSpeedHi,x   // Carry always clear here
282     lda #$01            // We avoid sec by adding 1
283     sbc angle,X
284     sta angle,x
285 invertSpeedSkip:
286     dex
287     bpl invertSpeedLoop
288
289     // Loop till no inversion
290     tsx
291     bpl mainLoop        // Check stack pointer msb to decide whether an inversion is needed
```

“The performance of the task provided intrinsic reward.”

food to go

- if you think something is foolish, most probably somebody already did it...
- ...and somebody else forked it!
- if you think something is useless/broken and fear bugs and issues...
- ...publishing it you'll receive, and this means somebody used it!
- if you think you are still able to have fun like a child...
- ...join us to the ASM-side of the force!

source: https://aceds.org/six-reasons-to-ask-a-question-from-the-audience-aceds-blog/