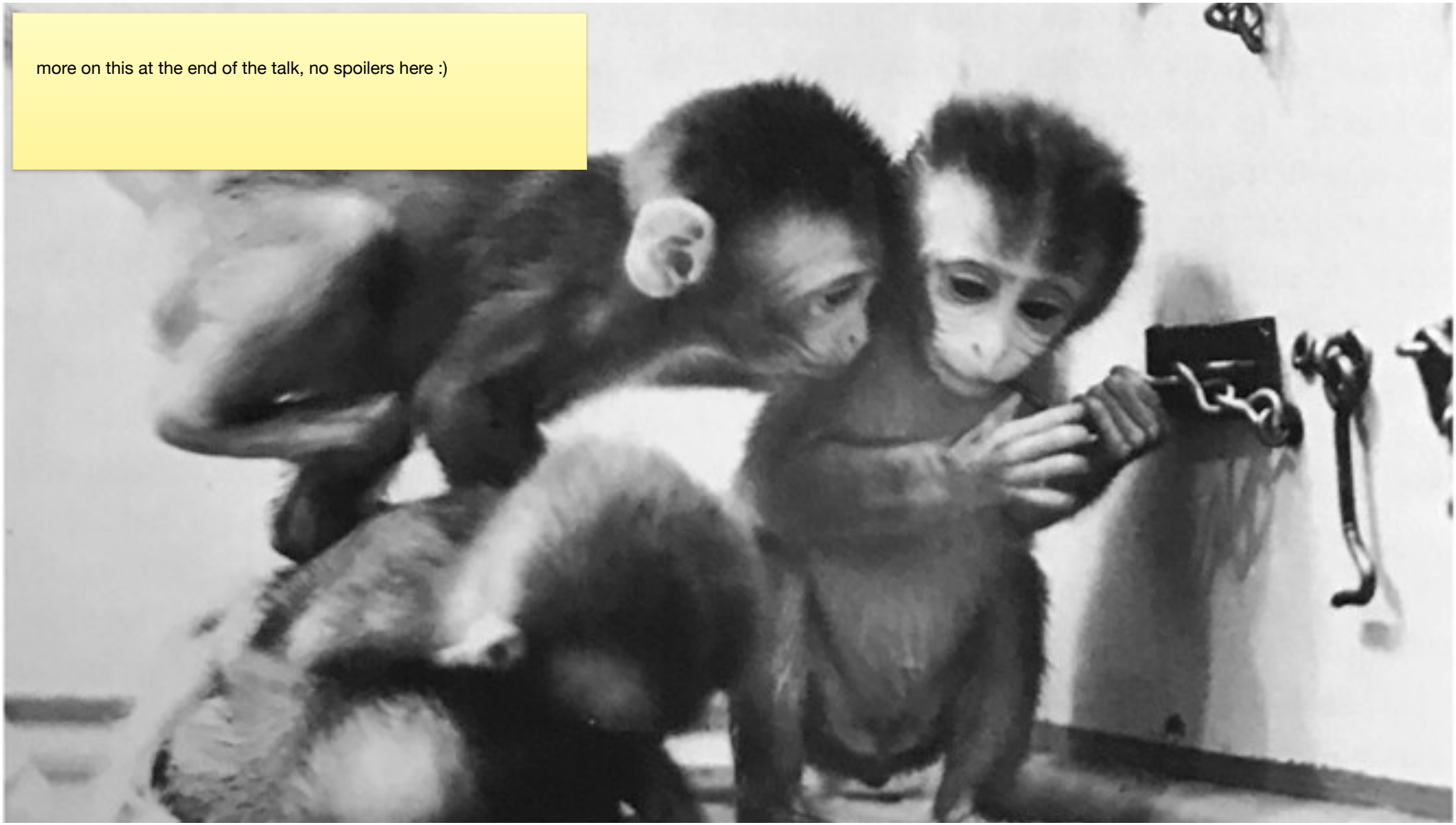
Assembly Coding 2024

introduzione al vostro prossimo hobby preferito

Sandro "guly" Zaccarini, Diego "freshness" Barzon

31/01/2024 - Modena Full Stack


more on this at the end of the talk, no spoilers here :)

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 dependant (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

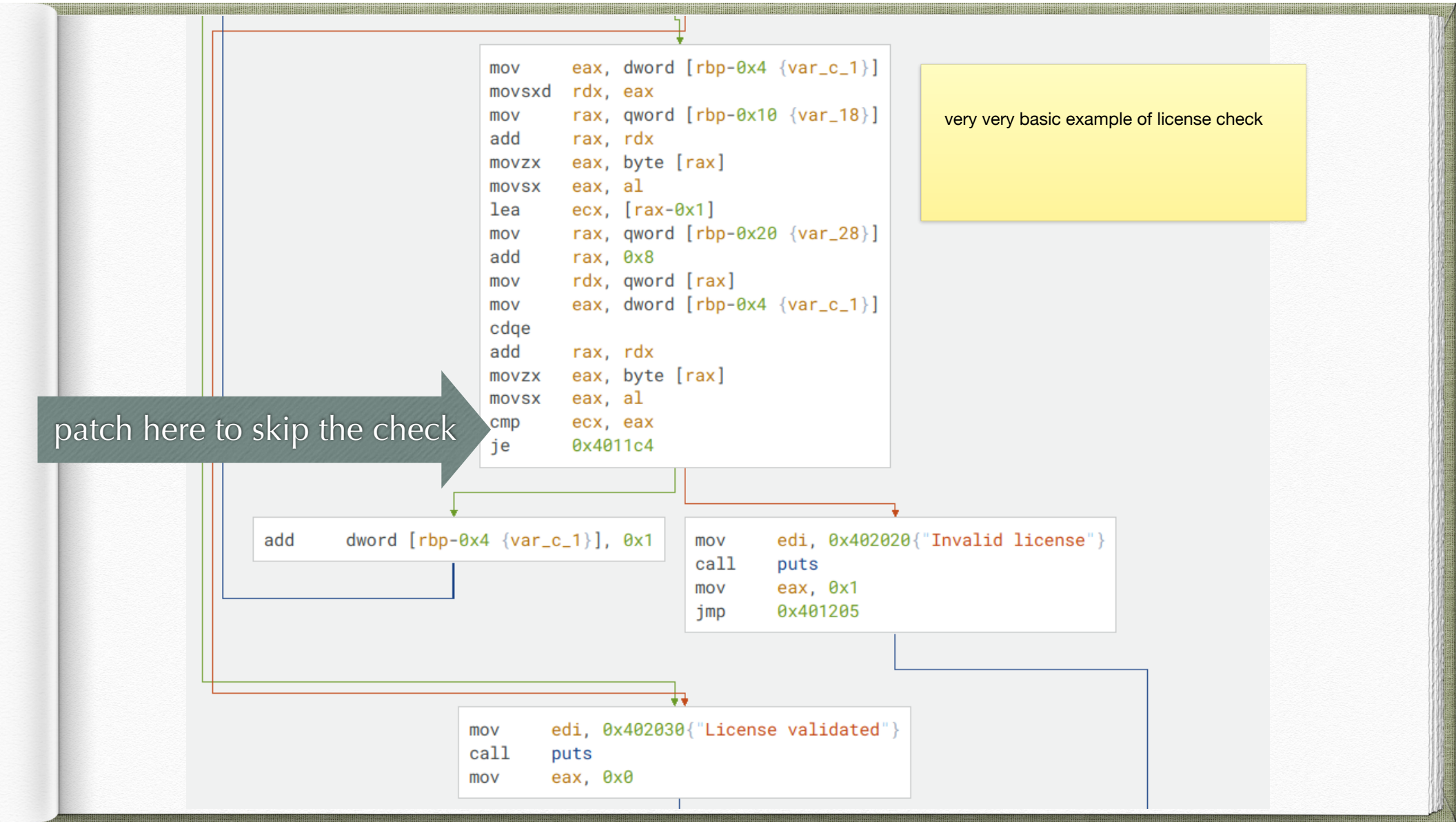
- reverse engineering
- pwn/exploit
- high performances
- embedded/very low resources
- very specific tasks

use case > reversing

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

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
very very basic example of license check
patch here to skip the check
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
- a different team write code based on the documentation

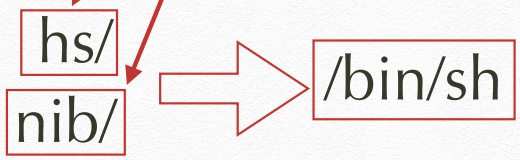
source: https://reactos.org/

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

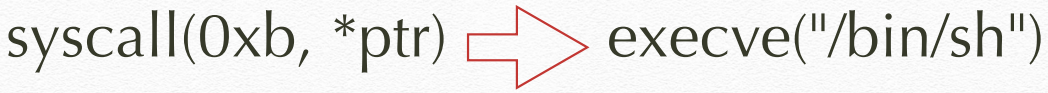
```
08049000   6a0b          push    0xb {var_4}
08049002   58            pop     eax {var_4}  {0xb}
08049003   682f736800    push    0x68732f {var_4}
08049008   682f62696e    push    0x6e69622f {var_8}
0804900d   89e3          mov     ebx, esp {var_8}
0804900f   cd80          int     0x80
08049011 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



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

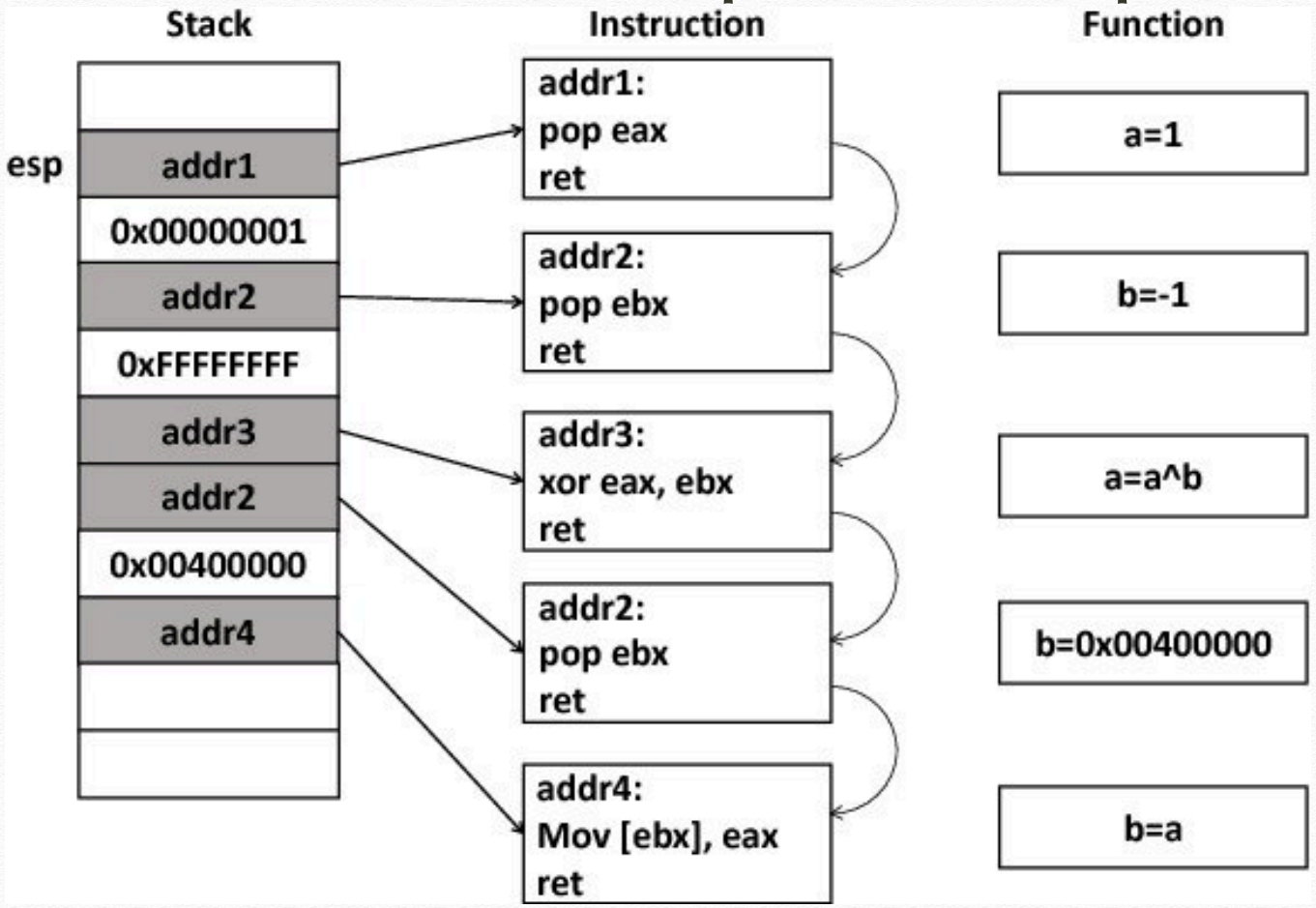
use case > pwn > shellcode

```
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;
```

lots of gadgets from our helloworld

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 > 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

```
 90   // Test C function
 91   printf("Starting C optimized function:\n");
 92     for(int i=0;i<64000;i++)
 93         test2[i] = 0;
 94   c_start = clock();
 95   for(long i=0;i<50000;i++) {
 96     unsigned int data;
 97     ((unsigned char *)&data)[0] = 0;
 98     ((unsigned char *)&data)[1] = 64;
 99     ((unsigned char *)&data)[2] = 128;
100     ((unsigned char *)&data)[3] = 192;
101     for(int i=0;i<16000;i++) {
102         ((char *)&data)[0] += 1;
103         ((char *)&data)[1] += 1;
104         ((char *)&data)[2] += 1;
105         ((char *)&data)[3] += 1;
106         ((unsigned int *)test2)[i] = data;
107       }
108   }
109   c_end = clock();
110     total = (float)((c_end - c_start)/50000);
111   printf("Done - Elapsed: %5.1f\n",total);
```

```
 1 BITS 64
 2 ; RDI has "test" 64 bit buffer pointer (standard arg passing)
 3 ; We need to save only RBX, which must not be clobbered
 4 ; across function calls (RBX, RBP, R12-R15)
 5
 6   push rbx
 7   mov dx, 0xc080
 8   mov ebx, 0x4000
 9   mov ecx, 16000
10 mainloop:
11   add dl,1
12   add dh,1
13   mov eax,edx
14   shl eax,16
15   add bl,1
16   add bh,1
17   add eax,ebx
18   stosd
19   dec ecx
20   jne mainloop
21   pop rbx
22   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 (handheld devices) 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

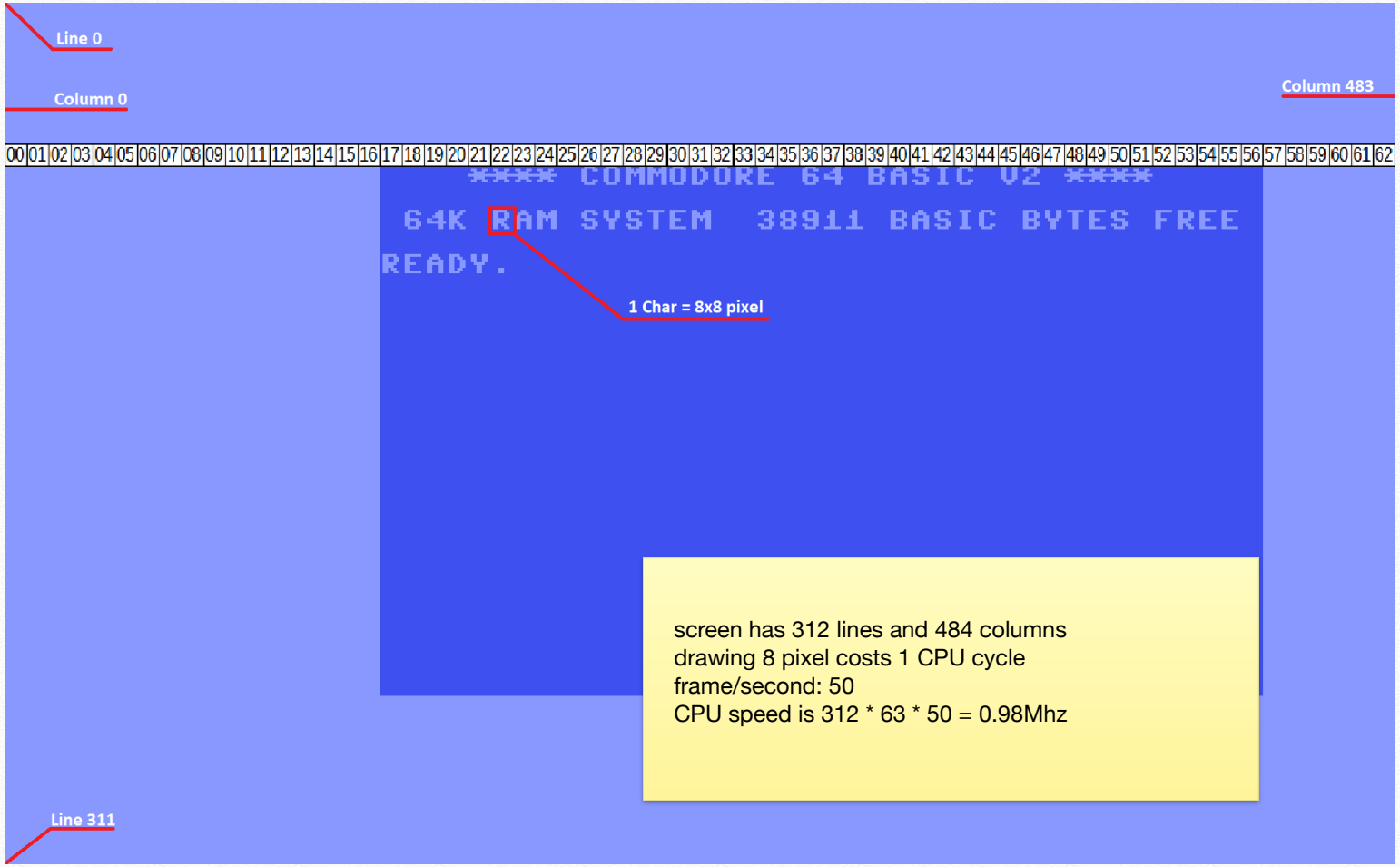
demo time

- demo1: abuse instruction behaviour to overcome hardware limits
- demo2: playable arkanoid-clone in 507bytes

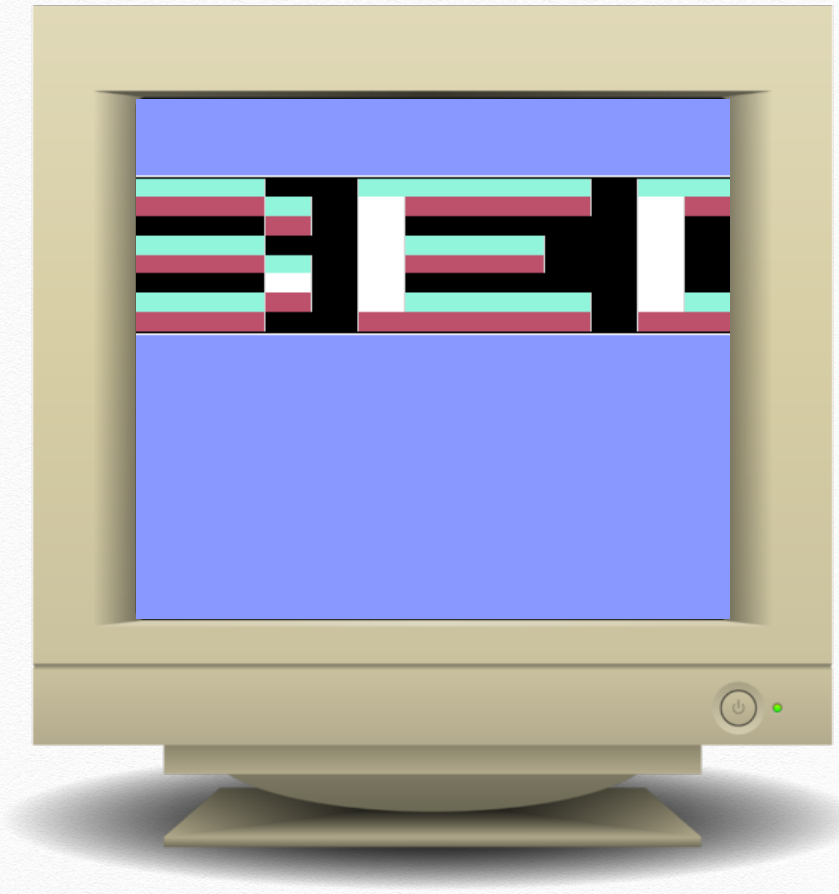
demo > platform

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

demo > video details



demo > scroller



as of 20240201 not public, will be in the next few weeks on csdb.dk under freshness user https://csdb.dk/scener/?id=9788

demo > synchronization loop

```
// Synchronize timer to raster (countdown 6 => 0)
synctimer:
    lda #$00
    sta $dc0e           // Stop timer
    lda #$08
    sta $dc04
    lda #$00            // Set timer to 9 cycles
    sta $dc05
    ldx #$ff            // Wait for lower part of the screen
waitlowerscreen:
    cpx $d012           // 4
    bne waitlowerscreen // 3/2 -> Max jitter 7 cycles
waitline:
    ldy #10             // 2
waitcycles:             // 49 <- (5*10)-1
    dey                 //
    bne waitcycles      //
    lda #$11            // 2
    inx                 // 2
    cpx $d012           // 4
    beq waitline        // 3/2
    // Next instruction on cycle 3
    nop         // 2
    nop         // 2
    nop         // 2
    sta $dc0e   // 4 Start timer
    rts
```

waitline function waits for line 255
waitcycles absorbs jittering in 1-7 lines using a loop made of 62 cycles

Synced code has to take 63 cycles per line otherwise image will be distorted

demo > synced loop

```
.align $0080
syncedblock:
    ldx #$00

openloop:
    cpx upperline       // 3

    bne !+              // 2/3 - 17
    nop                 // 2  - If upper line reached
    nop                 // 2
    nop                 // 2
    lda #$06            // 2  - Use color 06 (Blue)
    bne bgsetcolor      // 3
!:
    cpx lowerline       // 3
    bne !+              // 2/3
    lda #$0e            // 2  - If lower line reached
    bne bgsetcolor      // 3  - Use color 0e (Light blue)
!:
    bit $ff             // 3
    clc                 // 2
    bcc nosetcolor      // 3  - No color changes on all other lines
bgsetcolor:
    sta $d021           // 4

nosetcolor:
```

```
nosetcolor:
    cpx #$02            // 2  - 18
    bcs !+              // 2/3
    nop                 // 2  - Enable 24 rows screen mode
    bit $ff             // 3
    lda #$00            // 2
    beq openscreen      // 3
!:
    cpx #$c6            // 2
    bcs !+              // 3/2
    lda stophs:#$08     // 2
    bpl openscreen      // 3
!:
    lda #$00            // 2
    nop                 // 2 - Enable 25 rows screen mode
openscreen:
    sta $d011           // 4

    .fill 9,NOP         // 18  - 25 - Waste some cycles
    inx                 // 2
    cpx #205            // 2
    bne openloop        // 3 - Loop for the whole screen
```

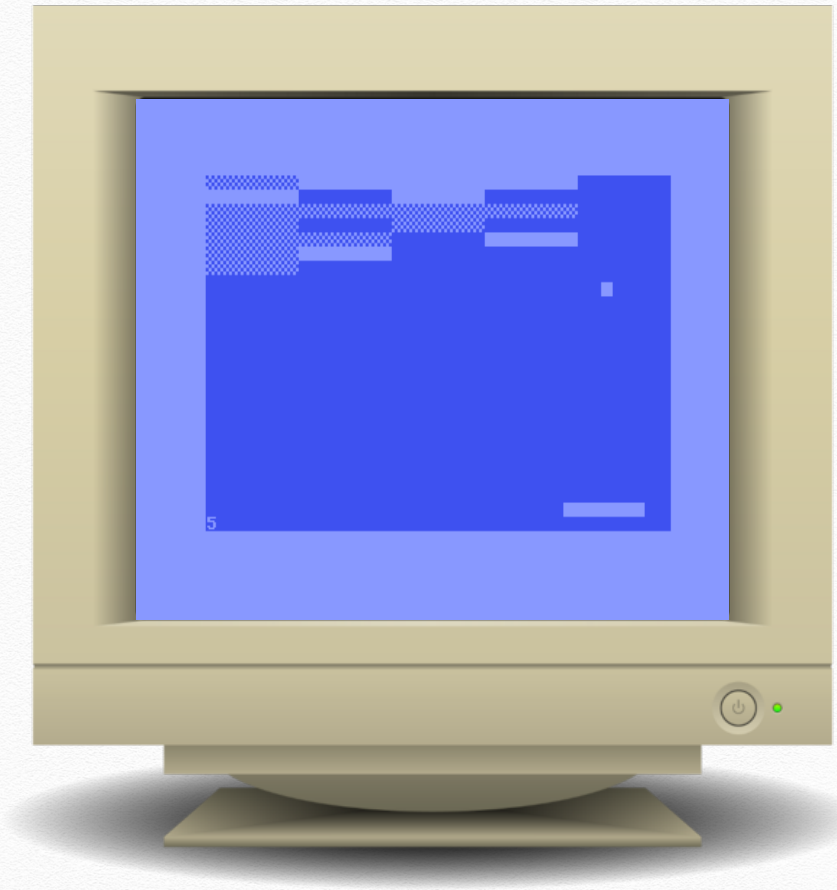
demo > synced loop

```
// Speed code will be created this way:
// sta $d020,x  // 5 - 63 (5 + 13*4 + 6)
// stx $d020    // 4
// sty $d020    // 4
// sax $d020    // 4
// sta $d020    // 4
// sta $d020    // 4
// stx $d020    // 4
// sta $d020    // 4
// sty $d020    // 4
// sta $d020    // 4
// sax $d020    // 4
// sty $d020    // 4
// stx $d020    // 4
// sax $d020    // 4
// rts          // 6
```

Each sta/x/y store a color on background color register, effectively drawing characters

demo > PETSCIInoid

downloadable at https://csdb.dk/release/?id=239089



demo > multiplication

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

6502 instruction set doesn't support multiplication, this is an easy way to write a custom x40

```
    // 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
```

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
EOR	0	0	0	1	0	0	0	1
LSR	0	0	0	0	1	0	0	0
EOR	0	0	0	1	1	0	0	1

LSR	0	0	0	0	1	1	0	0
EOR	0	0	0	1	1	1	0	1
LSR	0	0	0	0	1	1	1	0
EOR	0	0	0	1	1	1	1	1

LSR	0	0	0	0	1	1	1	1
EOR	0	0	0	1	1	1	1	0
LSR	0	0	0	0	1	1	1	1

LSR	0	0	0	0	0	1	1	1
EOR	0	0	0	1	0	1	1	0
LSR	0	0	0	0	1	0	1	1

LSR	0	0	0	0	0	1	0	1
EOR	0	0	0	1	0	1	0	0
LSR	0	0	0	0	1	0	1	0

```
    // TOPIC: LFSR pattern/random generator (Seed is level)
    lsr
    bcc lfsr1
    eor #LFSRPATTERN
lfsr1:
    lsr
    bcc lfsr2
    eor #LFSRPATTERN
lfsr2:
```

pattern generator used to draw the wall

demo > SP *is* a register

```
    // X is always $ff here
invertSpeed:
    txs                 // Abuse stack pointer to detect inversion, $ff means no inversion
    ldx #$01
invertSpeedLoop:
    lsr ballInvertFlags,x
    bcc invertSpeedSkip
    txs                 // $00 or $01 means inversion
    lda #$00
    sbc ballSpeedLo,x
    sta ballSpeedLo,x   // Carry always clear here
    lda #$00
    sbc ballSpeedHi,x
    sta ballSpeedHi,x   // Carry always clear here
    lda #$01            // We avoid sec by adding 1
    sbc angle,X
    sta angle,x
invertSpeedSkip:
    dex
    bpl invertSpeedLoop

    // Loop till no inversion
    tsx
    bpl mainLoop        // Check stack pointer msb to decide whether an inversion is needed
```

tsx and txs abuse the SP register as a temporary storage, shaving more bytes

writer is used to store some bits on the lower nibble while highest bits are used as condition to exit the loop

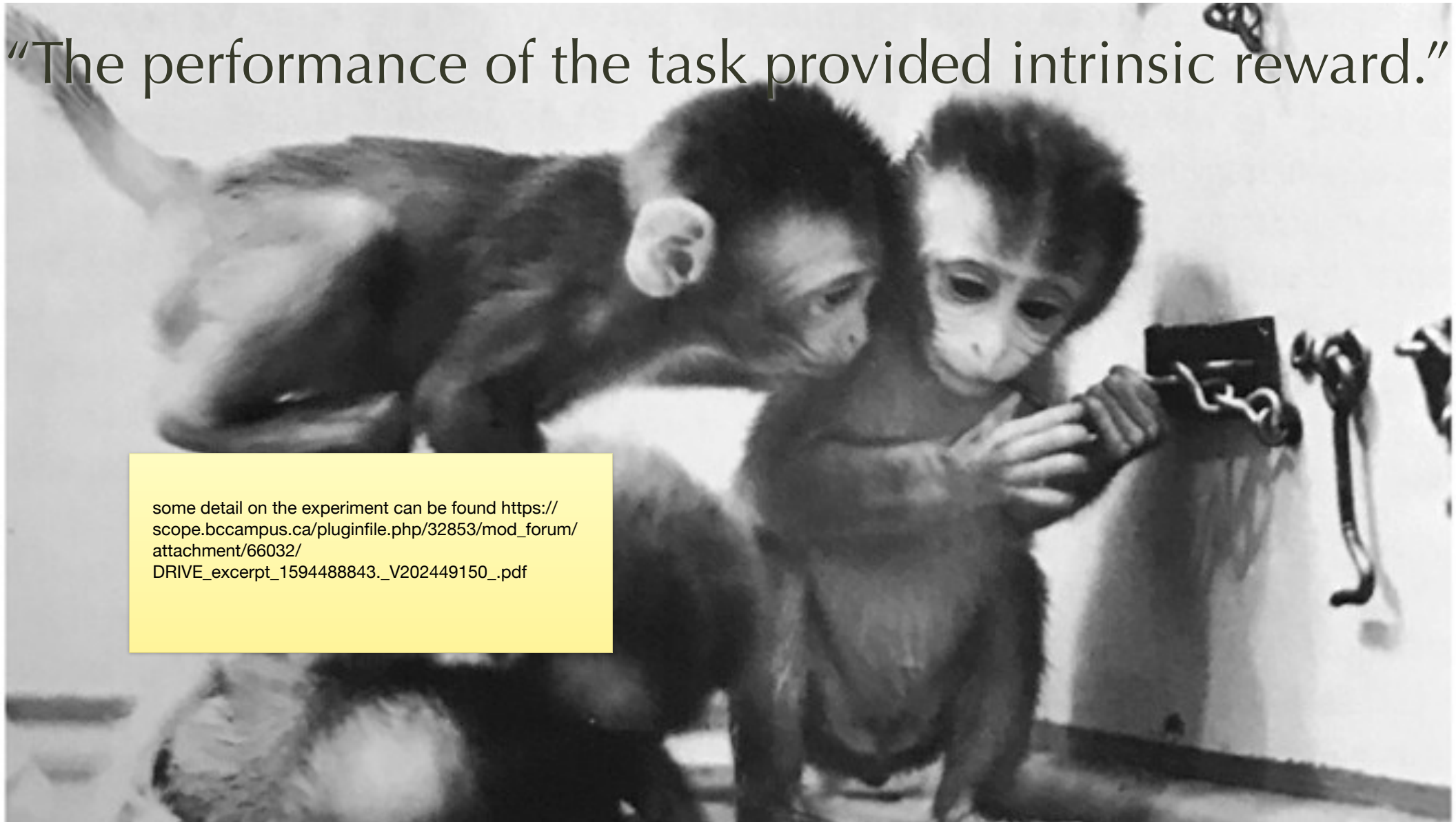
demo > every bit counts

```
// Prepare ball screen PTR and read current value
// ############################################
// This is used for:
// - Drawing ball
// - Checking obstacles
// - Drawing wall
prepareBallPtr:
    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

!:  // Prepare pattern based on fine tune bit
    lda ballCoordHi - BALLPTRH,x        // Get least significant bit of the hi-byte of X,Y ball coords
    lsr
    sta ballCoordHiTemp_2 - BALLPTRH ,x
    rol writer      // Store them in "writer" by shifting
    inx
    bcs !-          // Use higher bits of "writer" to exit at second iteration
```

```
.const WRITERINIT = %10101000
.const BALLPTRH = %10000000 + (SCREENPAGE/$400)
```

```
    // Draw ball pattern on screen
updateBall:
    jsr prepareBallPtr
    ldx writer
drawBallLoop:
    lda ballchars-(<(WRITERINIT*4)), x
    eor (drawPtr),y
    sta (drawPtr),y
    txa
    axs #-4
    iny
    asl writer
    bcs drawBallLoop
    bpl quitPrepareBallPtr
    tya
    adc #(SCREENWIDTH-2)
    tay
    bcc drawBallLoop
    inc drawPtr + 1
    bcs drawBallLoop
```


"The performance of the task provided intrinsic reward."
some detail on the experiment can be found https://scope.bccampus.ca/pluginfile.php/32853/mod_forum/attachment/66032/DRIVE_excerpt_1594488843._V202449150_.pdf

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/