Lab 6 - Memory Instructions, Virtual Memory & Page Tables

This lab can only be marked once and marking commences whenever the demonstrator starts marking it (the highest grade is 100 marks). The lab is split into three parts:

* **Exercise 1** is about memory instructions in machine code and relates to the provided C code “stack.c” [10+10+10 marks]
* **Exercise 2** is about the virtual memory of processes and the virtual address space they interact with and relates to the provided C code “virtual.c” [15+15+15 marks]
* **Exercise 3** is about simulating paging to look into the representation of page tables and the role of the translation lookaside buffer (TLB) and relates to the provided python script “paging.py” [10+5+10 marks + 10 bonus marks]

It follows a quick introduction to concepts related to the exercises:

* **Source code** is written in languages that help us express what we want to happen through:
  + variables that reference memory addresses of typed data objects by name
  + functions/methods that take inputs, do actions and return outputs
  + types/classes/structs that organise data objects
  + if-then-else blocks to control which code is executed in which case as well as for/while loops (e.g., to repeat actions until a desired outcome is reached)
* **Machine code** is a sequence of instructions the CPU can directly execute such as:
  + various instructions to read/write to a handful CPU registers, (e.g. arithmetic/logic operations, putting subroutine parameters into registers used for inputs, backing up registers)
  + memory instructions to read/write something to/from a register from/to main memory at a certain address specified in a register (e.g., related to variables/call stack)
  + branch instructions to jump to a certain position in the code based on the value of a register (this is how functions/if-then-else/loops are implemented in machine code)
* **Compilers** translate source code files to machine code files (libraries/executables).
* **Disassemblers** translate machine code files to an output where each line is a machine code instruction translated to a human-readable assembler programming language.

# Exercise 1

The provided source code “stack.c” calls a recursive function. Compilers usually translate recursive function to a subroutine that implements the function and a jump/branch to the subroutine where the function is called. If the compiler cannot eliminate the recursion it usually grow the call stack each time it jumps to the subroutine (which implements recursively calling the function).

# Tasks:

* Compile the source code file “stack.c” to a machine code file (e.g., “gcc -O3 -o stack3 stack.c”)
* Use a disassembler to read the machine code file (e.g., “objdump -d stack3” or on MacOSX “otool -vt stack3”)
* Execute the machine code file (“./stack3”)
* Answer the questions below.

1. A stack pointer (SP) is the virtual address of the first byte of the stack. How many bytes does the SP move down per each recursive call? Does this number change when compiling either with the “-O3” or “-O0” flags?

# [10 marks]

The stack pointer (SP) moves downwards with each recursive call, allocating space for function call data (such as local variables, function arguments, return addresses, and saved registers). The movement of the SP depends on how much memory the function call needs.

### **Analysis with** -O0 **(No Optimization)**

* The output shows that the SP moves **48 bytes per recursive call**:
  + **1 call → 48 bytes**
  + **2 calls → 96 bytes**
  + **3 calls → 144 bytes**
* This pattern suggests a consistent movement of **48 bytes per call**.

### **Analysis with** -O3 **(Optimization Level 3)**

* At first, the SP remains unchanged for the first **7 recursive calls**, meaning the compiler may have optimized out unnecessary stack frames or inlined the calls.
* Starting from **8 calls**, the SP moves by **64 bytes per recursive call**.
* This movement remains consistent for higher numbers of calls:
  + **8 calls → 64 bytes**
  + **9 calls → 64 bytes**
  + **10 calls → 64 bytes**
* This indicates that when optimizations are enabled, function frames are larger, possibly due to additional alignment or padding, register spilling, or optimized memory management.

### Summary of Stack Pointer Movement

* **With -O0**:
  + SP moves **48 bytes per call**, indicating a standard unoptimized function call with minimal memory use.
* **With -O3**:
  + SP moves **64 bytes per call**, which is larger due to the compiler's alignment or optimization choices.
* The initial lack of movement (0 bytes for the first few calls) is likely caused by aggressive optimization like function inlining or tail-call optimization.

1. Use a disassembler tool to analyse the compiled machine code of the program. In which instruction (should correspond to a disassembler output line) of the machine code does it jump/branch from the main function into the subroutine that implements the “f” function? (copy that line in your answer)

# [10 marks]

116f: e8 9c 01 00 00 call 1310 <f>

* e8: This is the opcode for a call instruction, which transfers control to a subroutine.
* 9c 01 00 00: This is the relative offset from the current instruction to the f function (address 1320).
* call 1320 <f>: This is the actual call to the f function.

It uses a relative call, meaning it adds the offset to the instruction pointer to jump to the correct address.

1. Which machine code instructions read something from memory and which ones write something to memory? Copy the whole disassembler output and prepend an “R” to at least one read instruction and a “W” to at least one write instruction.

# [10 marks]

stack3:     file format elf64-x86-64

Disassembly of section .init:

0000000000001000 <\_init>:

    1000:   f3 0f 1e fa             endbr64

    1004:   48 83 ec 08             sub    $0x8,%rsp

    1008:   48 8b 05 d9 2f 00 00    mov    0x2fd9(%rip),%rax        # 3fe8 <\_\_gmon\_start\_\_@Base> -------R

    100f:   48 85 c0                test   %rax,%rax

    1012:   74 02                   je     1016 <\_init+0x16>

    1014:   ff d0                   call   \*%rax

    1016:   48 83 c4 08             add    $0x8,%rsp

    101a:   c3                      ret

Disassembly of section .plt:

0000000000001020 <.plt>:

    1020:   ff 35 8a 2f 00 00       push   0x2f8a(%rip)        # 3fb0 <\_GLOBAL\_OFFSET\_TABLE\_+0x8>

    1026:   f2 ff 25 8b 2f 00 00    bnd jmp \*0x2f8b(%rip)        # 3fb8 <\_GLOBAL\_OFFSET\_TABLE\_+0x10>

    102d:   0f 1f 00                nopl   (%rax)

    1030:   f3 0f 1e fa             endbr64

    1034:   68 00 00 00 00          push   $0x0

    1039:   f2 e9 e1 ff ff ff       bnd jmp 1020 <\_init+0x20>

    103f:   90                      nop

    1040:   f3 0f 1e fa             endbr64

    1044:   68 01 00 00 00          push   $0x1

    1049:   f2 e9 d1 ff ff ff       bnd jmp 1020 <\_init+0x20>

    104f:   90                      nop

    1050:   f3 0f 1e fa             endbr64

    1054:   68 02 00 00 00          push   $0x2

    1059:   f2 e9 c1 ff ff ff       bnd jmp 1020 <\_init+0x20>

    105f:   90                      nop

Disassembly of section .plt.got:

0000000000001060 <\_\_cxa\_finalize@plt>:

    1060:   f3 0f 1e fa             endbr64

    1064:   f2 ff 25 8d 2f 00 00    bnd jmp \*0x2f8d(%rip)        # 3ff8 <\_\_cxa\_finalize@GLIBC\_2.2.5>

    106b:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

Disassembly of section .plt.sec:

0000000000001070 <putchar@plt>:

    1070:   f3 0f 1e fa             endbr64

    1074:   f2 ff 25 45 2f 00 00    bnd jmp \*0x2f45(%rip)        # 3fc0 <putchar@GLIBC\_2.2.5>

    107b:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

0000000000001080 <puts@plt>:

    1080:   f3 0f 1e fa             endbr64

    1084:   f2 ff 25 3d 2f 00 00    bnd jmp \*0x2f3d(%rip)        # 3fc8 <puts@GLIBC\_2.2.5>

    108b:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

0000000000001090 <\_\_printf\_chk@plt>:

    1090:   f3 0f 1e fa             endbr64

    1094:   f2 ff 25 35 2f 00 00    bnd jmp \*0x2f35(%rip)        # 3fd0 <\_\_printf\_chk@GLIBC\_2.3.4>

    109b:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

Disassembly of section .text:

00000000000010a0 <main>:

    10a0:   f3 0f 1e fa             endbr64

    10a4:   55                      push   %rbp

    10a5:   bf 0a 00 00 00          mov    $0xa,%edi

    10aa:   48 89 e5                mov    %rsp,%rbp

    10ad:   41 57                   push   %r15

    10af:   4c 8d 3d 8c 0f 00 00    lea    0xf8c(%rip),%r15        # 2042 <\_IO\_stdin\_used+0x42>

    10b6:   41 56                   push   %r14

    10b8:   49 89 ee                mov    %rbp,%r14

    10bb:   41 55                   push   %r13

    10bd:   4c 8d 2d 44 0f 00 00    lea    0xf44(%rip),%r13        # 2008 <\_IO\_stdin\_used+0x8>

    10c4:   41 54                   push   %r12

    10c6:   49 89 ec                mov    %rbp,%r12

    10c9:   53                      push   %rbx --------W

    10ca:   bb 01 00 00 00          mov    $0x1,%ebx

    10cf:   48 83 ec 08             sub    $0x8,%rsp

    10d3:   e8 98 ff ff ff          call   1070 <putchar@plt>

    10d8:   eb 56                   jmp    1130 <main+0x90>

    10da:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)

    10e0:   b9 01 00 00 00          mov    $0x1,%ecx

    10e5:   31 d2                   xor    %edx,%edx

    10e7:   4c 89 ee                mov    %r13,%rsi

    10ea:   bf 01 00 00 00          mov    $0x1,%edi

    10ef:   31 c0                   xor    %eax,%eax

    10f1:   bb 02 00 00 00          mov    $0x2,%ebx

    10f6:   e8 95 ff ff ff          call   1090 <\_\_printf\_chk@plt>

    10fb:   4c 39 e5                cmp    %r12,%rbp

    10fe:   4c 89 e0                mov    %r12,%rax

    1101:   4c 89 e2                mov    %r12,%rdx

    1104:   b9 02 00 00 00          mov    $0x2,%ecx

    1109:   48 0f 4e c5             cmovle %rbp,%rax

    110d:   4c 89 ee                mov    %r13,%rsi

    1110:   bf 01 00 00 00          mov    $0x1,%edi

    1115:   48 29 c2                sub    %rax,%rdx

    1118:   31 c0                   xor    %eax,%eax

    111a:   e8 71 ff ff ff          call   1090 <\_\_printf\_chk@plt>

    111f:   83 c3 01                add    $0x1,%ebx

    1122:   81 fb 10 27 00 00       cmp    $0x2710,%ebx

    1128:   0f 8f a3 00 00 00       jg     11d1 <main+0x131>

    112e:   66 90                   xchg   %ax,%ax

    1130:   83 fb 01                cmp    $0x1,%ebx

    1133:   74 ab                   je     10e0 <main+0x40>

    1135:   48 63 fb                movslq %ebx,%rdi

    1138:   83 fb 02                cmp    $0x2,%ebx

    113b:   74 be                   je     10fb <main+0x5b>

    113d:   b9 03 00 00 00          mov    $0x3,%ecx

    1142:   48 83 ff 03             cmp    $0x3,%rdi

    1146:   0f 84 9e 00 00 00       je     11ea <main+0x14a>

    114c:   48 83 ff 04             cmp    $0x4,%rdi

    1150:   0f 84 92 00 00 00       je     11e8 <main+0x148>

    1156:   4c 89 e0                mov    %r12,%rax

    1159:   48 83 ff 05             cmp    $0x5,%rdi

    115d:   74 2a                   je     1189 <main+0xe9>

    115f:   48 83 ff 06             cmp    $0x6,%rdi

    1163:   74 1d                   je     1182 <main+0xe2>

    1165:   48 83 ff 07             cmp    $0x7,%rdi

    1169:   74 10                   je     117b <main+0xdb>

    116b:   48 83 ef 07             sub    $0x7,%rdi

    116f:   e8 9c 01 00 00          call   1310 <f>

    1174:   48 39 c5                cmp    %rax,%rbp

    1177:   48 0f 4e c5             cmovle %rbp,%rax

    117b:   49 39 c4                cmp    %rax,%r12

    117e:   49 0f 4e c4             cmovle %r12,%rax

    1182:   49 39 c4                cmp    %rax,%r12

    1185:   49 0f 4e c4             cmovle %r12,%rax

    1189:   49 39 c4                cmp    %rax,%r12

    118c:   4c 89 e2                mov    %r12,%rdx

    118f:   89 d9                   mov    %ebx,%ecx

    1191:   49 0f 4e c4             cmovle %r12,%rax

    1195:   4c 89 ee                mov    %r13,%rsi

    1198:   bf 01 00 00 00          mov    $0x1,%edi

    119d:   4c 39 f0                cmp    %r14,%rax

    11a0:   49 0f 4f c6             cmovg  %r14,%rax

    11a4:   48 29 c2                sub    %rax,%rdx

    11a7:   31 c0                   xor    %eax,%eax

    11a9:   e8 e2 fe ff ff          call   1090 <\_\_printf\_chk@plt>

    11ae:   83 fb 0a                cmp    $0xa,%ebx

    11b1:   74 05                   je     11b8 <main+0x118>

    11b3:   83 fb 64                cmp    $0x64,%ebx

    11b6:   75 43                   jne    11fb <main+0x15b>

    11b8:   8d 1c 9b                lea    (%rbx,%rbx,4),%ebx

    11bb:   4c 89 ff                mov    %r15,%rdi

    11be:   e8 bd fe ff ff          call   1080 <puts@plt>

    11c3:   01 db                   add    %ebx,%ebx

    11c5:   81 fb 10 27 00 00       cmp    $0x2710,%ebx

    11cb:   0f 8e 5f ff ff ff       jle    1130 <main+0x90>

    11d1:   48 83 c4 08             add    $0x8,%rsp

    11d5:   31 c0                   xor    %eax,%eax

    11d7:   5b                      pop    %rbx

    11d8:   41 5c                   pop    %r12

    11da:   41 5d                   pop    %r13

    11dc:   41 5e                   pop    %r14

    11de:   41 5f                   pop    %r15

    11e0:   5d                      pop    %rbp

    11e1:   c3                      ret

    11e2:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)

    11e8:   89 d9                   mov    %ebx,%ecx

    11ea:   31 d2                   xor    %edx,%edx

    11ec:   4c 89 ee                mov    %r13,%rsi

    11ef:   bf 01 00 00 00          mov    $0x1,%edi

    11f4:   31 c0                   xor    %eax,%eax

    11f6:   e8 95 fe ff ff          call   1090 <\_\_printf\_chk@plt>

    11fb:   81 fb e8 03 00 00       cmp    $0x3e8,%ebx

    1201:   74 b5                   je     11b8 <main+0x118>

    1203:   e9 17 ff ff ff          jmp    111f <main+0x7f>

    1208:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)

    120f:   00

0000000000001210 <\_start>:

    1210:   f3 0f 1e fa             endbr64

    1214:   31 ed                   xor    %ebp,%ebp

    1216:   49 89 d1                mov    %rdx,%r9

    1219:   5e                      pop    %rsi

    121a:   48 89 e2                mov    %rsp,%rdx

    121d:   48 83 e4 f0             and    $0xfffffffffffffff0,%rsp

    1221:   50                      push   %rax

    1222:   54                      push   %rsp

    1223:   45 31 c0                xor    %r8d,%r8d

    1226:   31 c9                   xor    %ecx,%ecx

    1228:   48 8d 3d 71 fe ff ff    lea    -0x18f(%rip),%rdi        # 10a0 <main>

    122f:   ff 15 a3 2d 00 00       call   \*0x2da3(%rip)        # 3fd8 <\_\_libc\_start\_main@GLIBC\_2.34>

    1235:   f4                      hlt

    1236:   66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)

    123d:   00 00 00

0000000000001240 <deregister\_tm\_clones>:

    1240:   48 8d 3d c9 2d 00 00    lea    0x2dc9(%rip),%rdi        # 4010 <\_\_TMC\_END\_\_>

    1247:   48 8d 05 c2 2d 00 00    lea    0x2dc2(%rip),%rax        # 4010 <\_\_TMC\_END\_\_>

    124e:   48 39 f8                cmp    %rdi,%rax

    1251:   74 15                   je     1268 <deregister\_tm\_clones+0x28>

    1253:   48 8b 05 86 2d 00 00    mov    0x2d86(%rip),%rax        # 3fe0 <\_ITM\_deregisterTMCloneTable@Base>

    125a:   48 85 c0                test   %rax,%rax

    125d:   74 09                   je     1268 <deregister\_tm\_clones+0x28>

    125f:   ff e0                   jmp    \*%rax

    1261:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)

    1268:   c3                      ret

    1269:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)

0000000000001270 <register\_tm\_clones>:

    1270:   48 8d 3d 99 2d 00 00    lea    0x2d99(%rip),%rdi        # 4010 <\_\_TMC\_END\_\_>

    1277:   48 8d 35 92 2d 00 00    lea    0x2d92(%rip),%rsi        # 4010 <\_\_TMC\_END\_\_>

    127e:   48 29 fe                sub    %rdi,%rsi

    1281:   48 89 f0                mov    %rsi,%rax

    1284:   48 c1 ee 3f             shr    $0x3f,%rsi

    1288:   48 c1 f8 03             sar    $0x3,%rax

    128c:   48 01 c6                add    %rax,%rsi

    128f:   48 d1 fe                sar    %rsi

    1292:   74 14                   je     12a8 <register\_tm\_clones+0x38>

    1294:   48 8b 05 55 2d 00 00    mov    0x2d55(%rip),%rax        # 3ff0 <\_ITM\_registerTMCloneTable@Base>

    129b:   48 85 c0                test   %rax,%rax

    129e:   74 08                   je     12a8 <register\_tm\_clones+0x38>

    12a0:   ff e0                   jmp    \*%rax

    12a2:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)

    12a8:   c3                      ret

    12a9:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)

00000000000012b0 <\_\_do\_global\_dtors\_aux>:

    12b0:   f3 0f 1e fa             endbr64

    12b4:   80 3d 55 2d 00 00 00    cmpb   $0x0,0x2d55(%rip)        # 4010 <\_\_TMC\_END\_\_>

    12bb:   75 2b                   jne    12e8 <\_\_do\_global\_dtors\_aux+0x38>

    12bd:   55                      push   %rbp

    12be:   48 83 3d 32 2d 00 00    cmpq   $0x0,0x2d32(%rip)        # 3ff8 <\_\_cxa\_finalize@GLIBC\_2.2.5>

    12c5:   00

    12c6:   48 89 e5                mov    %rsp,%rbp

    12c9:   74 0c                   je     12d7 <\_\_do\_global\_dtors\_aux+0x27>

    12cb:   48 8b 3d 36 2d 00 00    mov    0x2d36(%rip),%rdi        # 4008 <\_\_dso\_handle>

    12d2:   e8 89 fd ff ff          call   1060 <\_\_cxa\_finalize@plt>

    12d7:   e8 64 ff ff ff          call   1240 <deregister\_tm\_clones>

    12dc:   c6 05 2d 2d 00 00 01    movb   $0x1,0x2d2d(%rip)        # 4010 <\_\_TMC\_END\_\_>

    12e3:   5d                      pop    %rbp

    12e4:   c3                      ret

    12e5:   0f 1f 00                nopl   (%rax)

    12e8:   c3                      ret

    12e9:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)

00000000000012f0 <frame\_dummy>:

    12f0:   f3 0f 1e fa             endbr64

    12f4:   e9 77 ff ff ff          jmp    1270 <register\_tm\_clones>

    12f9:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)

0000000000001300 <getStackPointer>:

    1300:   f3 0f 1e fa             endbr64

    1304:   55                      push   %rbp

    1305:   48 89 e5                mov    %rsp,%rbp

    1308:   48 89 e8                mov    %rbp,%rax

    130b:   5d                      pop    %rbp

    130c:   c3                      ret

    130d:   0f 1f 00                nopl   (%rax)

0000000000001310 <f>:

    1310:   f3 0f 1e fa             endbr64

    1314:   55                      push   %rbp

    1315:   48 89 e5                mov    %rsp,%rbp

    1318:   41 56                   push   %r14

    131a:   48 89 e8                mov    %rbp,%rax

    131d:   41 55                   push   %r13

    131f:   41 54                   push   %r12

    1321:   53                      push   %rbx

    1322:   48 83 ff 01             cmp    $0x1,%rdi

    1326:   74 50                   je     1378 <f+0x68>

    1328:   48 89 eb                mov    %rbp,%rbx

    132b:   48 83 ff 02             cmp    $0x2,%rdi

    132f:   74 40                   je     1371 <f+0x61>

    1331:   49 89 ec                mov    %rbp,%r12

    1334:   48 83 ff 03             cmp    $0x3,%rdi

    1338:   74 30                   je     136a <f+0x5a>

    133a:   49 89 ed                mov    %rbp,%r13

    133d:   48 83 ff 04             cmp    $0x4,%rdi

    1341:   74 20                   je     1363 <f+0x53>

    1343:   49 89 ee                mov    %rbp,%r14

    1346:   48 83 ff 05             cmp    $0x5,%rdi

    134a:   74 10                   je     135c <f+0x4c>

    134c:   48 83 ef 05             sub    $0x5,%rdi

    1350:   e8 bb ff ff ff          call   1310 <f>

    1355:   48 39 c5                cmp    %rax,%rbp

    1358:   48 0f 4e c5             cmovle %rbp,%rax

    135c:   49 39 c6                cmp    %rax,%r14

    135f:   49 0f 4e c6             cmovle %r14,%rax

    1363:   49 39 c5                cmp    %rax,%r13

    1366:   49 0f 4e c5             cmovle %r13,%rax

    136a:   49 39 c4                cmp    %rax,%r12

    136d:   49 0f 4e c4             cmovle %r12,%rax

    1371:   48 39 c3                cmp    %rax,%rbx

    1374:   48 0f 4e c3             cmovle %rbx,%rax

    1378:   5b                      pop    %rbx

    1379:   41 5c                   pop    %r12

    137b:   41 5d                   pop    %r13

    137d:   41 5e                   pop    %r14

    137f:   5d                      pop    %rbp

    1380:   c3                      ret

Disassembly of section .fini:

0000000000001384 <\_fini>:

    1384:   f3 0f 1e fa             endbr64

    1388:   48 83 ec 08             sub    $0x8,%rsp

    138c:   48 83 c4 08             add    $0x8,%rsp

    1390:   c3                      ret

 mov 0x2fd9(%rip),%rax moves data from memory into the %rax register.

 The 0x2fd9(%rip) refers to a memory location **relative to the Instruction Pointer (RIP)**.

 This means the instruction **reads** from memory and places the value into %rax.

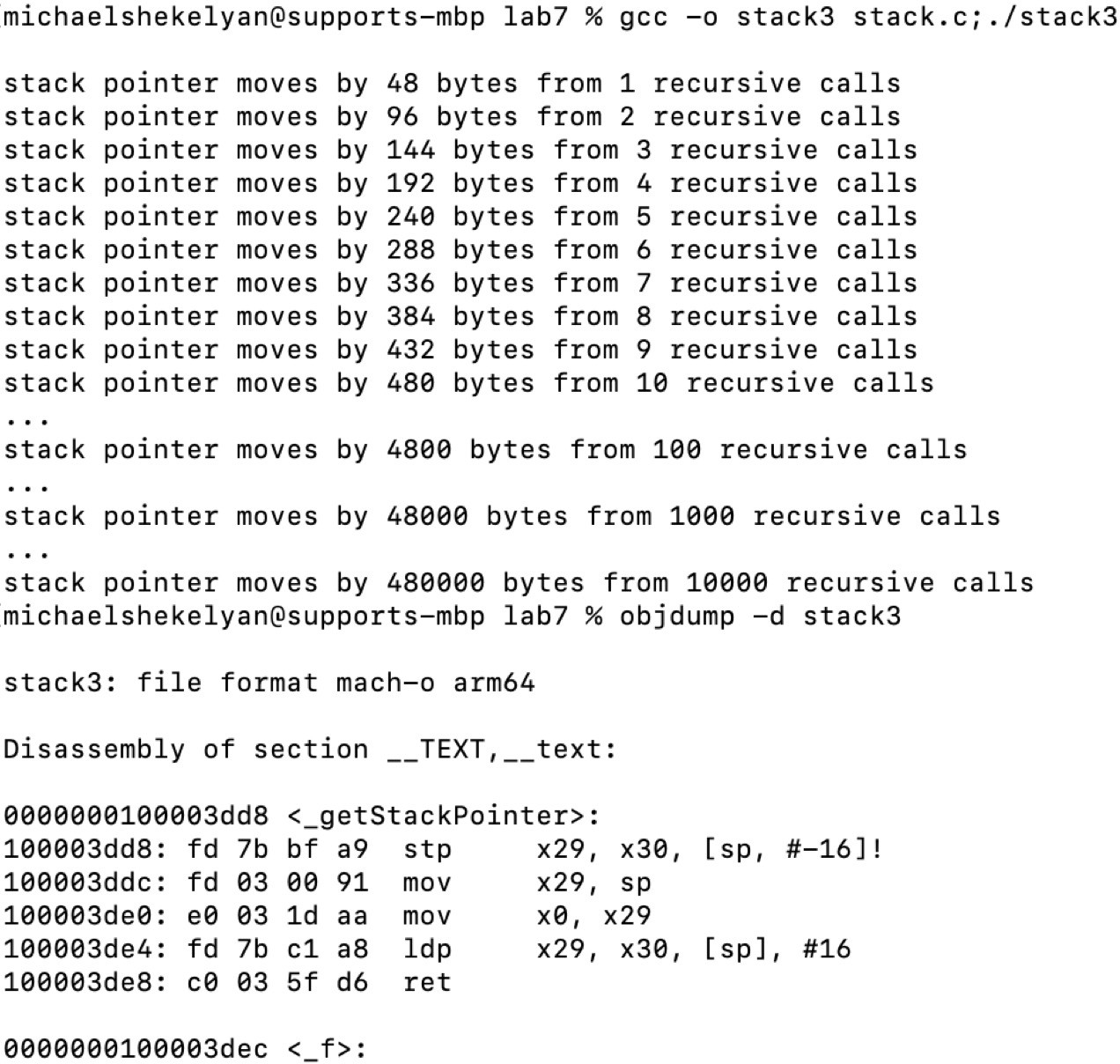


Figure 1: Exemplary disassembled output of “stack.c”

# Exercise 2

Virtual Memory

* + In C the expression &x gives the virtual memory address of the value stored in the variable x
  + In C if “x” has the integer type long (a signed 64-bit integer), then “&x” has the pointer type long\* (a pointer to a memory address where a long value begins)
  + In C the expression “long\* ptr = &x” would assign the variable ptr with the memory address of x.
  + In C the expression “\*ptr” gives the value stored at the virtual address referenced by ptr (a pointer is a variable that has a memory address as a value)

# Tasks:

* + Analyse the C code in the file “virtual.c” file (e.g., using a text editor)
  + Compile “virtual.c” (e.g. “gcc -o virtual virtual.c” to create the executable “virtual”)
  + Execute the compilation output (e.g. “./virtual”) and analyse the outputs by the program.
  + Answer the questions below and be able to explain how you are arrived at your answers when getting it marked.

1. The C program with source code “virtual.c” outputs virtual addresses in decimal notation written as PN\*PS+OS = VA. After inspecting the code, what does each of the variables PN,PS,OS and VA describe (e.g., what would be a good name for each of them)?

# [15 marks]

PN = Page number within the virtual address

Refers to the index of the page where the data resides. A virtual address in a system is divided into a page number, for identifying the page, and an offset, for locating the data within that page.

PS = Page size

Number of bytes each page can contain.

OS = Offset

Tells you the position of the specific byte within a page. The offset allows you to pinpoint the exact location withing the page.

VA = Virtual address

Refers to the complete address used by the program to locate data.

1. The compiled program with source code “virtual.c” outputs virtual addresses both in decimal and hexadecimal notation. One could also manually translate between both (e.g., 0xEFF1

= 14 *∗* 163 + 15 *∗* 162 + 15 *∗* 161 + 1 *∗* 160). Pick any printed virtual address (copy the related

outputs): how would one manually translate the hexadecimal value into a decimal one?

# [15 marks]

virtual address pointing to 1st byte of arrayOnHeap[3]:

hexadecimal: 0x556d60b9c6c8

decimal: 22931704732 \* 4096 + 1736 = 93928262584008

Each digit in the hexadecimal number corresponds to a power of 16, with the rightmost digit having the lowest power (starting from 160160).

The hexadecimal value 0x556d60b9c6c8 consists of the following digits:

5,5,6,d,6,0,b,9,c,6,c,85,5,6,d,6,0,b,9,c,6,c,8

where:

• d=13

• b=11

• c=12

We now expand the hexadecimal value into its decimal form using powers of 16:

(5×1611)+(5×1610)+(6×169)+(13×168)+(6×167)+(0×166)+(11×165)+(9×164)+(12×163)+(6×162)+(12×161)+(8×160)(5×1611)+(5×1610)+(6×169)+(13×168)+(6×167)+(0×166)+(11×165)+(9×164)+(12×163)+(6×162)+(12×161)+(8×160)

You then sum all of them up:

87960930222080+5497558138880+412316860416+55834574848+1610612736+0+11534336+589824+49152+1536+192+887960930222080+5497558138880+412316860416+55834574848+1610612736+0+11534336+589824+49152+1536+192+8

Summing these up:

9392826258400893928262584008

Thus, the decimal equivalent of 0x556d60b9c6c8 is 93928262584008.

1. The compiled program with source “virtual.c” outputs multiple virtual addresses. Going through all printed virtual addresses, what are the lowest and highest virtual addresses? (copy printed lines of those) Considering that virtual addresses point to a single byte, how many bytes are addressable based on the smallest and highest address? (does the system have that much RAM?) How many frames do contain some bytes of arrayOnHeap and presuming the same frame size what is the maximal number of frames that could theoretically contain some bytes of arrayOnHeap?

# [15 marks]

Lowest:

virtual address pointing to 1st byte of global offsetBits variable:

hexadecimal: 0x556d34025014

decimal: 22931521573 \* 4096 + 20 = 93927512363028

Highest:

virtual address pointing to 1st byte of arrayOnHeap[3]:

hexadecimal: 0x556d60b9c6c8

decimal: 22931704732 \* 4096 + 1736 = 93928262584008

Addressable bytes (highest minus lowest+1) =93928262584008−93927512363028+1=750000000

So, there are 750,000,000 bytes addressable between the lowest and highest virtual addresses.

(you add the one since both addresses represent individual bytes)

The system will most likely have this much RAM ass 750 million bytes approximates to 714.3 MB and most systems today typically have GB or TB of RAM.

How many frames?

The system's page size is 4096 bytes (since the virtual address uses 12 offset bits, meaning the offset part is 12 bits wide, and the page size is 212=4096 bytes).

To calculate how many frames are used by the arrayOnHeap entries, considering that the arrayOnHeap addresses range from arrayOnHeap[0] to arrayOnHeap[3]:

• First byte of arrayOnHeap[0]: 0x556d60b9c6b0

• First byte of arrayOnHeap[1]: 0x556d60b9c6b8

• First byte of arrayOnHeap[2]: 0x556d60b9c6c0

• First byte of arrayOnHeap[3]: 0x556d60b9c6c8

Each entry in arrayOnHeap spans only 8 bytes, and since the page size is 4096 bytes, all of them fit within the frame. The 8-byte span of each arrayOnHeap element will be entirely within a single 4096-byte page.

Since these four addresses are within a contiguous block of memory and the page size is 4096 bytes, all four entries of arrayOnHeap occupy 1 page in total.

The Maximal Number of Frames That Could Theoretically Contain Some Bytes of arrayOnHeap

We can assume that the addresses are spread out over the maximum number of frames, considering that each frame is 4096 bytes.

Since each entry (arrayOnHeap[0], arrayOnHeap[1], arrayOnHeap[2], arrayOnHeap[3]) occupies only 8 bytes, and 8 bytes fit within a single frame, each entry is contained within a single frame. In the worst-case scenario, each entry could be in a separate frame.

However, since these entries are very close in memory (with only 8-byte gaps between them), they all fall into the same single frame in this specific case.

But, theoretically, if the entries were spread out across multiple pages (or frames), 4 frames could contain the arrayOnHeap elements.

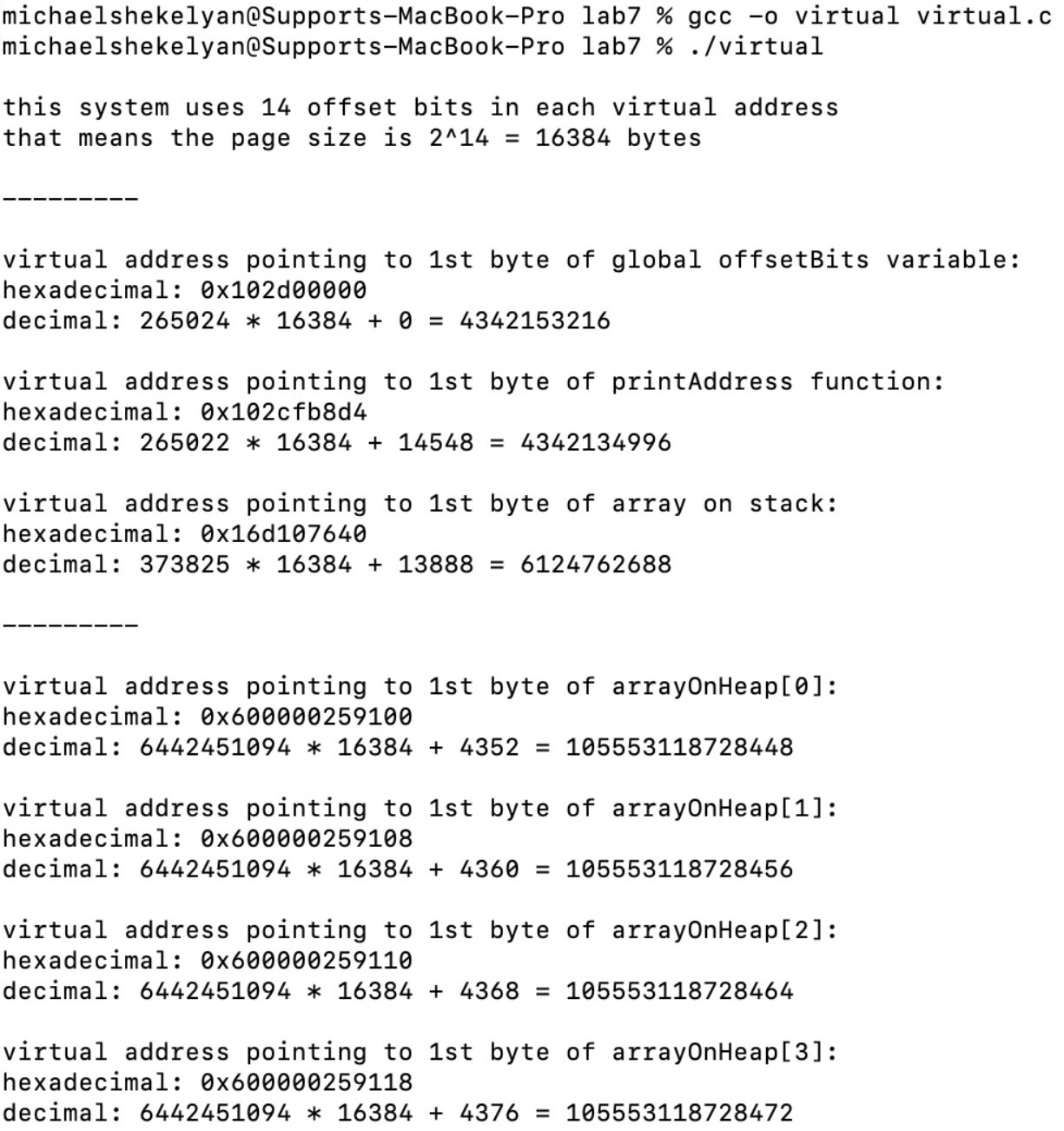


Figure 2: Exemplary output of program with source code “virtual.c”

# Exercise 3

The provided python script “paging.py” implements a simulator with x86-64 four-level paging with 36-bit page numbers (split into 9-bit sub parts for each level) and 12-bit offsets (the page/frame size is therefore 212 bytes = 4096 bytes = 4 kilobytes). Note that 64-bit systems reserve some of the bits for other purposes and usually do not use all 64 bits for virtual addresses (here only 48 bits are used). Each page table entry is 64 bits, which is 8 bytes. For 12-bit offsets, each page table is indexed by a 9-bit subpart of the page number and therefore has 29 = 512 entries. As each entry is 8 bytes, that means each page table perfectly fits into a frame (8 *∗* 512 = 4096).

# Tasks:

* + Familiarise yourself with the paging simulator in “paging.py”, its limitations and how it matches with the example in Figure 3.
  + Modify/extend “paging.py” as instructed in the questions below and briefly answer the questions.

1. Add some print statements to display the memory, so one can check if the state of the page simulator matches Figure 3 at the end and display how much memory is used by all page tables. Compare it to how much space would be used if only one level was used with 36-bit page numbers and 12-bit offsets.

# [10 marks]

1. As it is a simulator it will have some shortcomings. What are the limitations apart from lacking swapping and page faults? How could swapping and page faults be implemented in the simulator? (some ideas are sufficient, it is not necessary to implement it)

# [5 marks]

* No Access Permissions
* Fixed-Sized Frames Only:  
  It does not simulate cases where larger pages can improve performance by reducing page table depth.

1. Add a translation lookaside buffer (TLB) to the paging simulator with 512 entries which either clears the entries after each context switch or indicates which process id an entry relates to. *Note:* The paging simulator is not about achieving high performance and just about spelling out the logical flow of events in address translation.

# [10 marks]

1. **Bonus tasks** (not needed for 100 marks, but adds up to 10 marks until 100 marks are reached).

*Note*: page tables are unfortunately smaller than frames in case of two/three-level modes.

* 1. Add “x86-64” three-level mode (the “if self.offsetbits == 21” blocks) with 27-bit page numbers (each level has 9 bits) and 21-bit offsets (2MB page size).
  2. Add “x86-64” two-level mode (the “if self.offsetbits == 30” blocks) with 18-bit page numbers (each level has 9 bits) and 30-bit offsets (1GB page size).

# [10 marks]

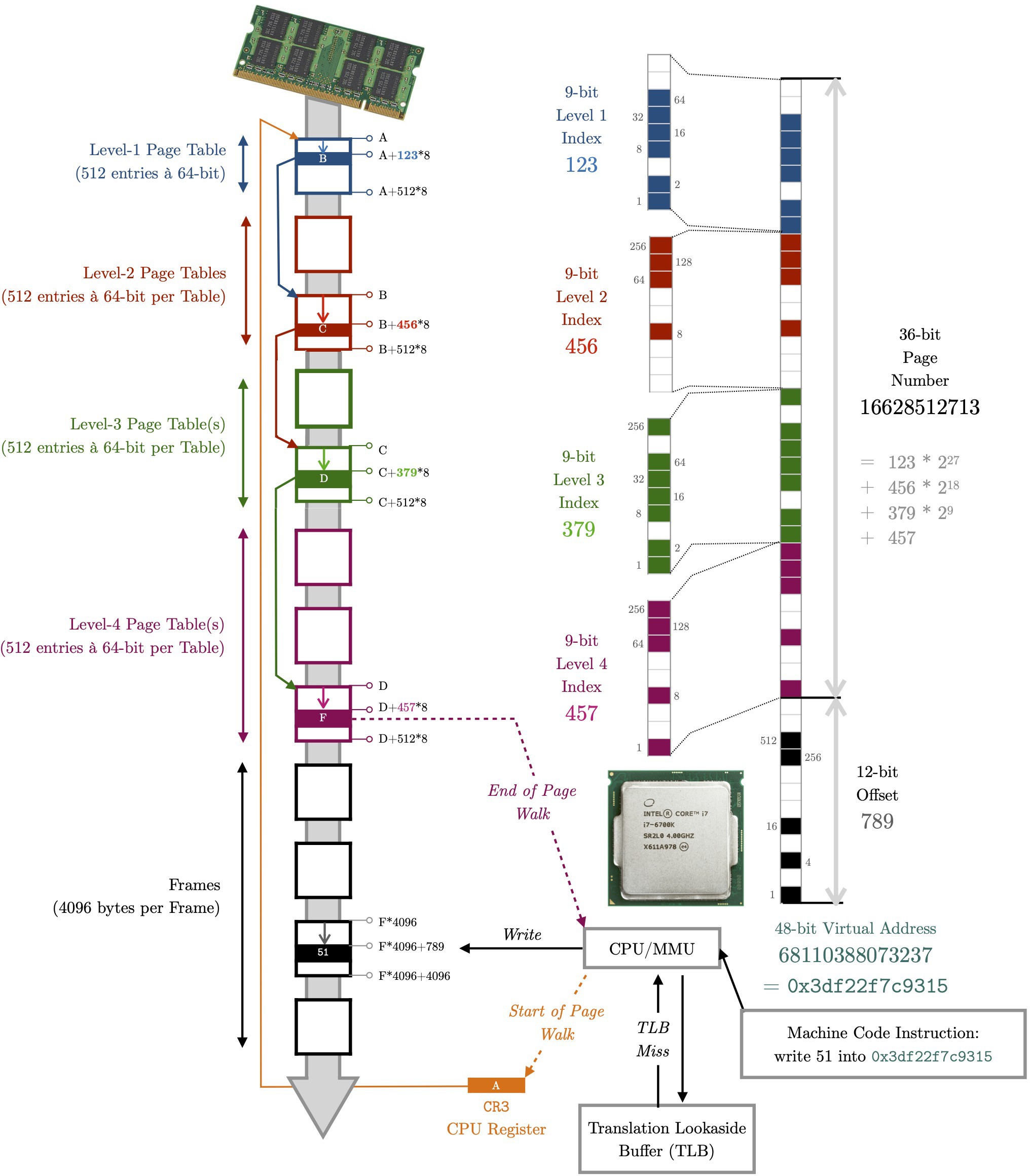


Figure 3: Exemplary address translation using four-level paging