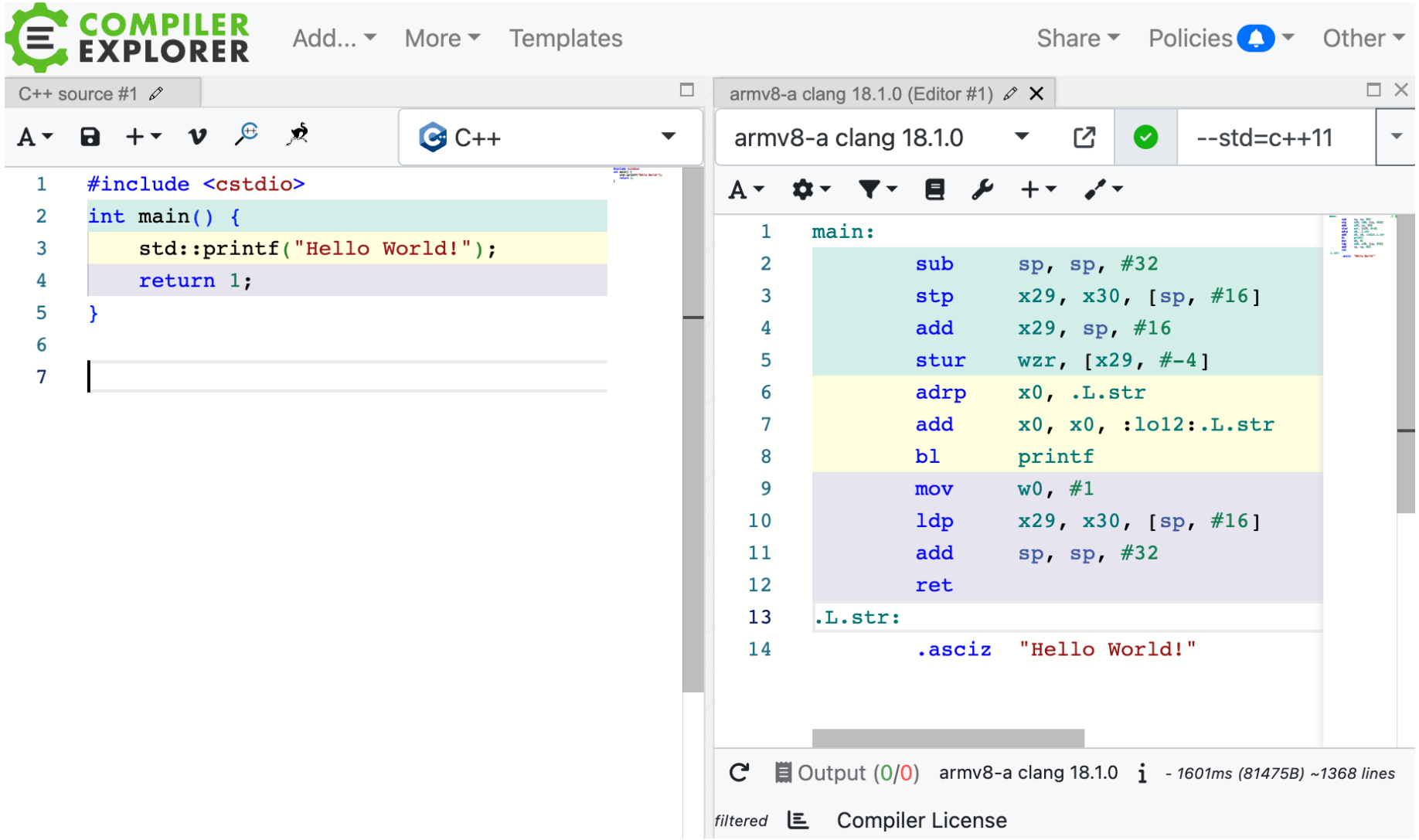
Passive ARM Assembly Skills for Debugging, Optimization (and Hacking)

Sebastian Theophil think-cell

21.11.2024

think-cell

Reading ARM Assembly



https://godbolt.org/z/jejT1vYbv

What is in a binary?

```
HEADER:0000000100000000 ; Mach-o header
HEADER:0000000100000000
HEADER:0000000100000000 ; Processor       : ARM
HEADER:0000000100000000 ; ARM architecture: metaarm
HEADER:0000000100000000 ; Target assembler: Generic assembler for ARM
HEADER:0000000100000000 ; Byte sex        : Little endian
HEADER:0000000100000000
HEADER:0000000100000000 ; ===========================================================================
HEADER:0000000100000000
HEADER:0000000100000000 ; Segment type: Pure data
HEADER:0000000100000000                 AREA HEADER, DATA, READONLY, ALIGN=0
HEADER:0000000100000000                 ; ORG 0x100000000
HEADER:0000000100000000                 EXPORT __mh_execute_header
HEADER:0000000100000000 ; const mach_header_64 _mh_execute_header
HEADER:0000000100000000 __mh_execute_header DCD 0xFEEDFACF      ; Magic number
HEADER:0000000100000004                 DCD 0x100000C           ; CPU type: ARM64
HEADER:0000000100000008                 DCD 0                   ; CPU subtype
HEADER:000000010000000C                 DCD 2                   ; File type: EXECUTE
HEADER:0000000100000010                 DCD 0x12                ; Number of load commands
HEADER:0000000100000014                 DCD 0x450               ; Size of load commands
HEADER:0000000100000018                 DCD 0x200085            ; Flags
HEADER:000000010000001C                 DCD 0                   ; Reserved
HEADER:0000000100000020 ; LC_SEGMENT_64 - 64-bit segment of this file to be mapped
HEADER:0000000100000020                 segment_command_64 <0x19, 0x48, "__PAGEZERO", 0, 0x100000000, 0, 0, 0,\
HEADER:0000000100000020                                    0, 0, 0>
HEADER:0000000100000068 ; LC_SEGMENT_64 - 64-bit segment of this file to be mapped
HEADER:0000000100000068                 segment_command_64 <0x19, 0x188, "__TEXT", 0x100000000, 0x4000, 0, \
HEADER:0000000100000068                                    0x4000, 5, 5, 4, 0>
HEADER:00000001000000B0 ; Sections
HEADER:00000001000000B0                 section_64 <"__text", "__TEXT", 0x100003F7C, 0x20, 0x3F7C, 2, 0, 0, \
HEADER:00000001000000B0                            0x80000400, 0, 0, 0>
HEADER:0000000100000100                 section_64 <"__stubs", "__TEXT", 0x100003F9C, 0xC, 0x3F9C, 2, 0, 0, \
HEADER:0000000100000100                            0x80000408, 0, 0xC, 0>
HEADER:0000000100000150                 section_64 <"__cstring", "__TEXT", 0x100003FA8, 0xD, 0x3FA8, 0, 0, 0, \
HEADER:0000000100000150                            2, 0, 0, 0>
HEADER:00000001000001A0                 section_64 <"__unwind_info", "__TEXT", 0x100003FB8, 0x48, 0x3FB8, 2, \
HEADER:00000001000001A0                            0, 0, 0, 0, 0, 0>
HEADER:00000001000001F0 ; LC_SEGMENT_64 - 64-bit segment of this file to be mapped
HEADER:00000001000001F0                 segment_command_64 <0x19, 0x98, "__DATA_CONST", 0x100004000, 0x4000, \
HEADER:00000001000001F0                                    0x4000, 0x4000, 3, 3, 1, 0x10>
HEADER:0000000100000238 ; Sections
HEADER:0000000100000238                 section_64 <"__got", "__DATA_CONST", 0x100004000, 8, 0x4000, 3, 0, 0, \
HEADER:0000000100000238                            6, 1, 0, 0>
HEADER:0000000100000288 ; LC_SEGMENT_64 - 64-bit segment of this file to be mapped
HEADER:0000000100000288                 segment_command_64 <0x19, 0x48, "__LINKEDIT", 0x100008000, 0x4000, \
HEADER:0000000100000288                                    0x8000, 0x292, 1, 1, 0, 0>
```

C++ linker produces executable files

These have OS-specific format

macOS: Mach-o
Linux: ELF
Windows: PE Format

Where is our code?

Where is our data?

What is in a binary?

```
struct segment_command_64 { /* for 64-bit architectures */
        uint32_t cmd; /* LC_SEGMENT_64 */
        uint32_t cmdsize; /* includes sizeof section_64 structs */
        char segname[16]; /* segment name */
        uint64_t vmaddr; /* memory address of this segment */
        uint64_t vmsize; /* memory size of this segment */
        uint64_t fileoff; /* file offset of this segment */
        uint64_t filesize; /* amount to map from the file */
        int32_t maxprot; /* maximum VM protection */
        int32_t initprot; /* initial VM protection */
        uint32_t nsects; /* number of sections in segment */
        uint32_t flags; /* flags */
};
```

What is in a binary?

```
HEADER:0000000100000308 ; LC_DYSYMTAB - dynamic link-edit symbol table info
HEADER:0000000100000308                 dysymtab_command <0xB, 0x50, 0, 0, 0, 2, 2, 1, 0, 0, 0, 0, 0, 0, \
HEADER:0000000100000308                                   0x80C8, 2, 0, 0, 0, 0>
HEADER:0000000100000358 ; LC_LOAD_DYLINKER - load a dynamic linker
HEADER:0000000100000358                 dylinker_command <0xE, 0x20, <0xC>>
HEADER:0000000100000364 aUsrLibDyld     DCB "/usr/lib/dyld",0   ; library's path name
HEADER:0000000100000372                 ALIGN 8
HEADER:0000000100000378 ; LC_UUID - the uuid
HEADER:0000000100000378                 DCD 0x1B                ; cmd
HEADER:000000010000037C                 DCD 0x18                ; cmdsize
HEADER:0000000100000380                 DCB 4, 0xD, 0x48, 0x71, 0xC1, 0xF4, 0x3F, 0xE1, 0x8F, 0x2E; uuid
HEADER:000000010000038A                 DCB 0xF1, 0x20, 0x67, 0xEB, 0xAB, 0x3A; uuid
HEADER:0000000100000390 ; LC_BUILD_VERSION - build for platform min OS version
HEADER:0000000100000390                 build_version_command <0x32, 0x20, 1, 0xD0000, 0xD0100, 1>
HEADER:00000001000003A8                 DCB    3
HEADER:00000001000003A9                 DCB    0
HEADER:00000001000003AA                 DCB    0
HEADER:00000001000003AB                 DCB    0
HEADER:00000001000003AC                 DCB    0
HEADER:00000001000003AD                 DCB    1
HEADER:00000001000003AE                 DCB 0x34 ; 4
HEADER:00000001000003AF                 DCB    3
HEADER:00000001000003B0 ; LC_SOURCE_VERSION - source version used to build binary
HEADER:00000001000003B0                 source_version_command <0x2A, 0x10, 0>
HEADER:00000001000003C0 ; LC_MAIN - replacement for LC_UNIXTHREAD
HEADER:00000001000003C0                 entry_point_command <0x80000028, 0x18, 0x3F7C, 0>
HEADER:00000001000003D8 ; LC_LOAD_DYLIB - load a dynamically linked shared library
HEADER:00000001000003D8                 dylib_command <0xC, 0x30, <<0x18>, 2, 0x5142400, 0x10000>>
HEADER:00000001000003F0 aUsrLibLibc1Dyl DCB "/usr/lib/libc++.1.dylib",0 ; library's path name
HEADER:0000000100000408 ; LC_LOAD_DYLIB - load a dynamically linked shared library
HEADER:0000000100000408                 dylib_command <0xC, 0x38, <<0x18>, 2, 0x5270000, 0x10000>>
HEADER:0000000100000420 aUsrLibLibsyste DCB "/usr/lib/libSystem.B.dylib",0 ; library's path name
HEADER:000000010000043B                 ALIGN 0x20
HEADER:0000000100000440 ; LC_FUNCTION_STARTS - compressed table of function start addresses
HEADER:0000000100000440                 linkedit_data_command <0x26, 0x10, 0x8090, 8>
HEADER:0000000100000450 ; LC_DATA_IN_CODE - table of non-instructions in __text
HEADER:0000000100000450                 linkedit_data_command <0x29, 0x10, 0x8098, 0>
HEADER:0000000100000460 ; LC_CODE_SIGNATURE - local of code signature
HEADER:0000000100000460                 linkedit_data_command <0x1D, 0x10, 0x8100, 0x192>
HEADER:0000000100000470                 DCB    0
HEADER:0000000100000471                 DCB    0
HEADER:0000000100000472                 DCB    0
HEADER:0000000100000473                 DCB    0
HEADER:0000000100000474                 DCB    0
HEADER:0000000100000475                 DCB    0
```

What to call first?

What to import?

Reading ARM Assembly

think-cell

Instruction Operands Register Immediate

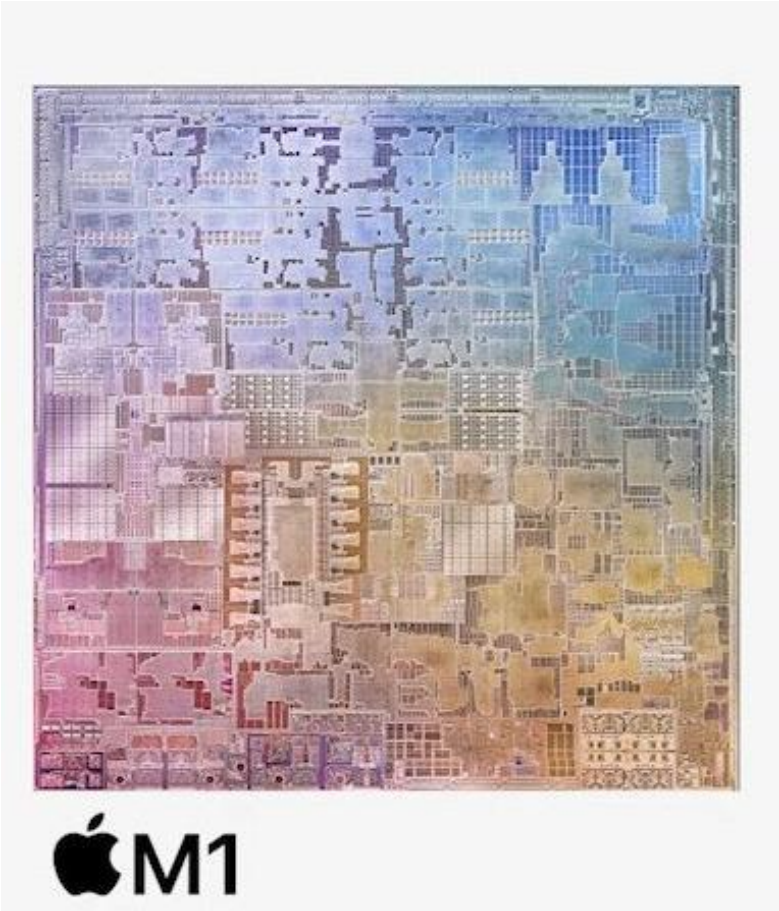
```
1   main:
2           sub     sp, sp, #32
3           stp     x29, x30, [sp, #16]
4           add     x29, sp, #16
5           stur    wzr, [x29, #-4]
6           adrp    x0, .L.str
7           add     x0, x0, :lo12:.L.str
8           bl      printf
9           mov     w0, #1
10          ldp     x29, x30, [sp, #16]
11          add     sp, sp, #32
12          ret
13  .L.str:
14          .asciz  "Hello World"
```

Memory Address (Register Value + Offset)

Memory Address (Label)

https://developer.arm.com/

What is in a CPU?



Execution Core @ 3.2 GHz

31 64bit Registers X0 - X30 (248 bytes, 1 cycle)

L1 Cache (~192 KB, 3 cycles)

L2 Cache (12 MB, 18 cycles)

RAM (16 GB, ~90ns / ~270 cycles)

https://github.com/ocxtal/insn_bench_aarch64/blob/master/optimization_notes_apple_m1.md

Using Registers

```
mov x0, xzr

mov w1, #1
```

x0-x30: 64-bit registers
w0-w30: 32-bit registers
(bottom bits of x0 – x30,
top bits are zeroed on write)

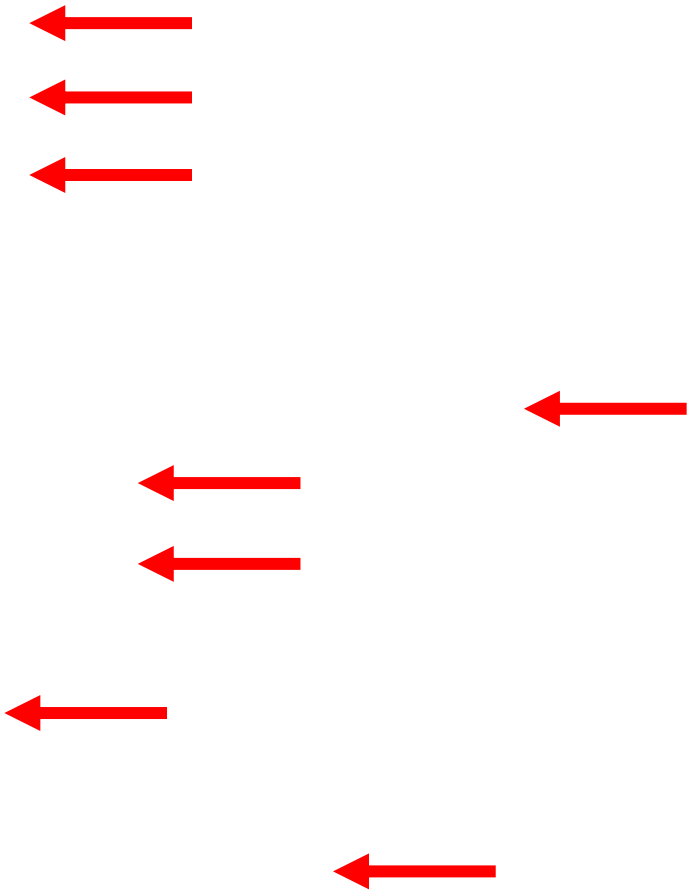
wzr/xzr: zero-valued registers
writes are ignored

https://godbolt.org/z/1bM6vsf9h

Using Registers

think-cell

```
mov x0, xzr ; fib1 = 0
mov w1, #1 ; fib2 = 1
mov w2, #5 ; i = 5

loop:
 add x3, x1, x0 ; fib = fib1 + fib2
 mov x0, x1 ; fib1 = fib2
 mov x1, x3 ; fib2 = fib

 sub x2, x2, #1 ; --i
 cmp x2, #0
 b.ne loop ; if i!=0 goto loop
```



https://godbolt.org/z/1bM6vsf9h

Using Registers

```cpp
std::int64_t fib(std::int64_t n) {
    std::int64_t fib1 = 0;
    std::int64_t fib2 = 1;
    for(std::int64_t j=0; j<n; ++j) {
        auto fib = fib1 + fib2;
        fib1 = fib2;
        fib2 = fib;
    }
    return fib2;
}
```

```
fib(long):
        cmp     x0, #1
        b.lt    .LBB0_4
        mov     x10, xzr
        mov     w9, #1
.LBB0_2:
        add     x8, x10, x9
        subs    x0, x0, #1
        mov     x10, x9
        mov     x9, x8
        b.ne    .LBB0_2
        mov     x0, x8
        ret
.LBB0_4:
        mov     w8, #1
        mov     x0, x8
        ret
```

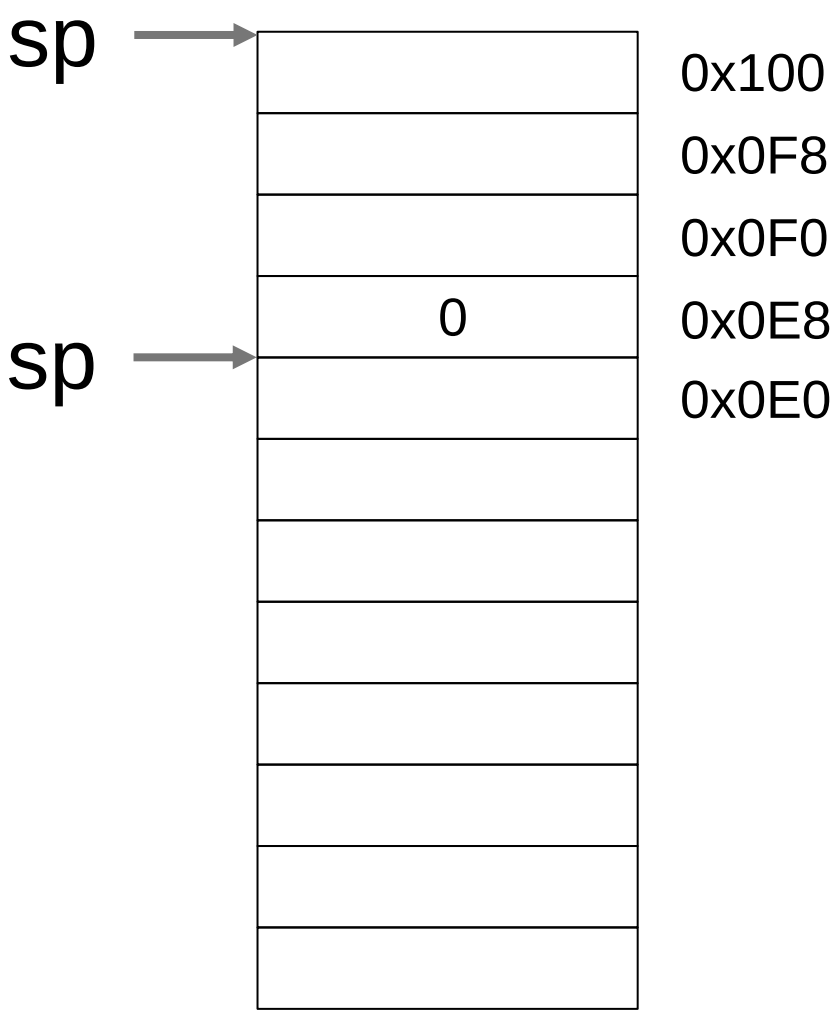
https://godbolt.org/z/1bM6vsf9h

Using the Stack

Stack memory allocated by OS
Stack pointer register points to stack
Stack grows down

```
sub sp, sp, #32

str wzr, [sp, #8]

ldr x0, [sp, #8]
```

Stack allocation is trivial
Stack access is cached and fast



Using the Stack

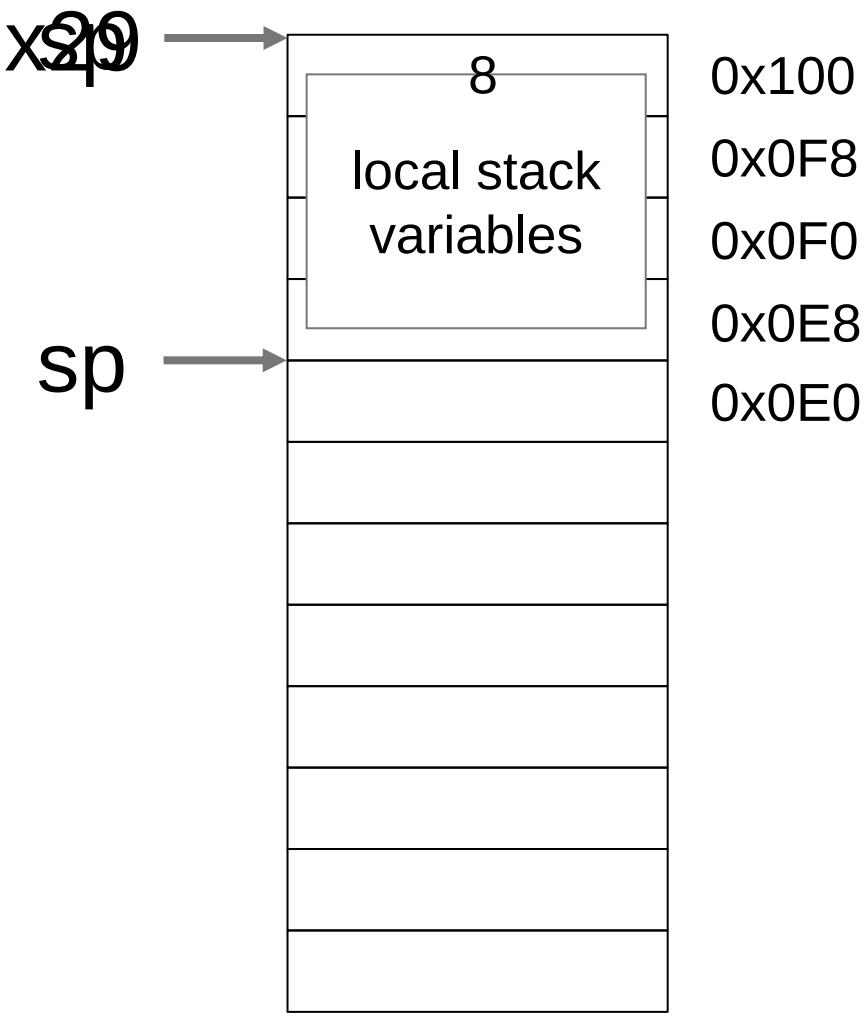
```
std::int64_t fib(std::int64_t n) {
    std::int64_t fib1 = 0;
    std::int64_t fib2 = 1;
    for(std::int64_t j=0; j<n; ++j) {
        auto fib = fib1 + fib2;
        fib1 = fib2;
        fib2 = fib;
    }
    return fib2;
}
```

```
fib(long):
        sub     sp, sp, #48
        str     x0, [sp, #40]
        str     xzr, [sp, #32]
        mov     x8, #1
        str     x8, [sp, #24]
        str     xzr, [sp, #16]
        b       .LBB0_1
.LBB0_1:
        ldr     x8, [sp, #16]
        ldr     x9, [sp, #40]
        subs    x8, x8, x9
        b.ge    .LBB0_4
        b       .LBB0_2
.LBB0_2:
        ldr     x8, [sp, #32]
        ldr     x9, [sp, #24]
        add     x8, x8, x9
        str     x8, [sp, #8]
        ldr     x8, [sp, #24]
        str     x8, [sp, #32]
```

https://godbolt.org/z/aWYTK9q46

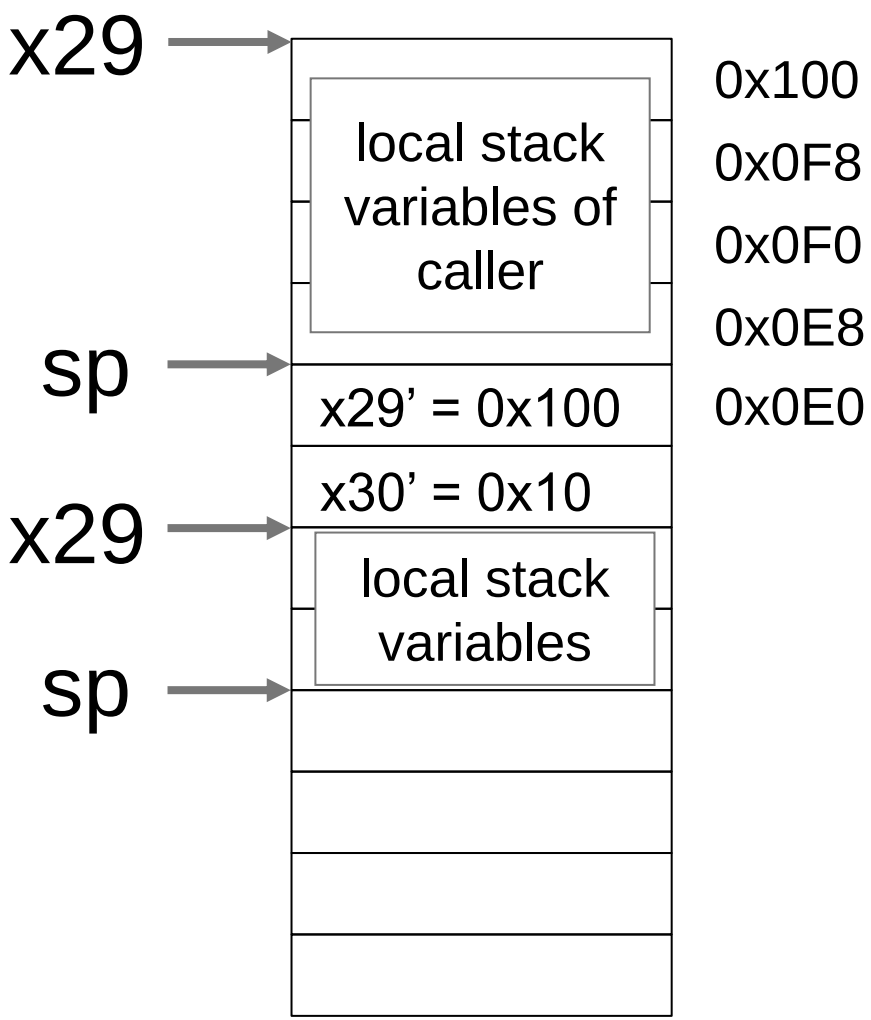
Calling Functions

```
0x0 sub sp, sp, #32
0x4 add x29, sp, #32
0x8 mov x0, #5
0xC bl fib
; sets x30 to 0x10
0x10 str x0, [x29, -#8]
```



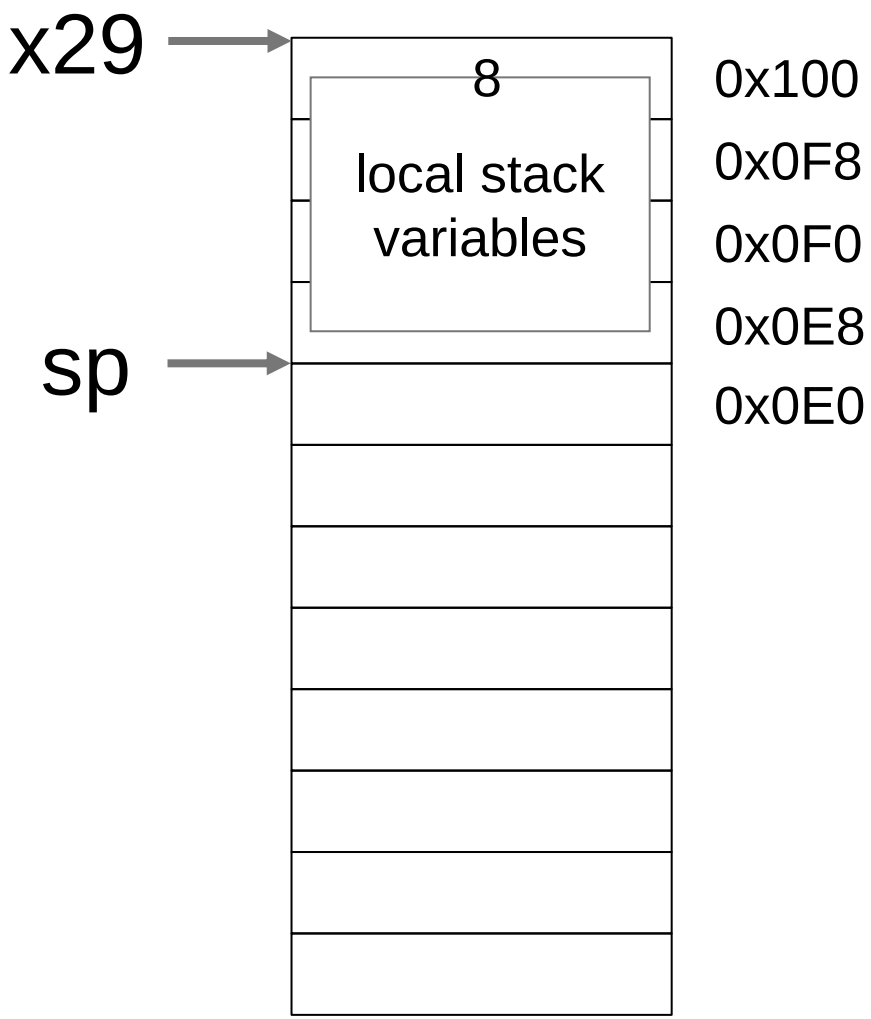
Calling Functions

```
fib:
  0x10 sub sp, sp, #32
  0x14 stp x29, x30,
       [sp, #16]

  0x18 add x29, sp, #16

  0x30 ldp x29, x30,
       [sp, #16]

  0x34 add sp, sp, #32
  0x38 ret ; jump to x30
```



Calling Functions

```
0x0  sub sp, sp, #32
0x4  add x29, sp, #32
0x8  mov x0, #5
0xC  bl fib
; sets x30 to 0x10
0xF  str x0, [x29, -#8]
```



Calling Functions (Simplified)

X0 – X7	Function arguments
X8	Indirect result
X9 – X15	Corruptible by callee
X16 – X17	Might be used by linker
X18	Reserved
X19 – X28	Callee-saved registers
X29	Frame pointer
X30	Link register

Similar rules exist for floating point registers

Stack must be 16 byte aligned

These rules are specific to each OS!

Binary code must be compatible

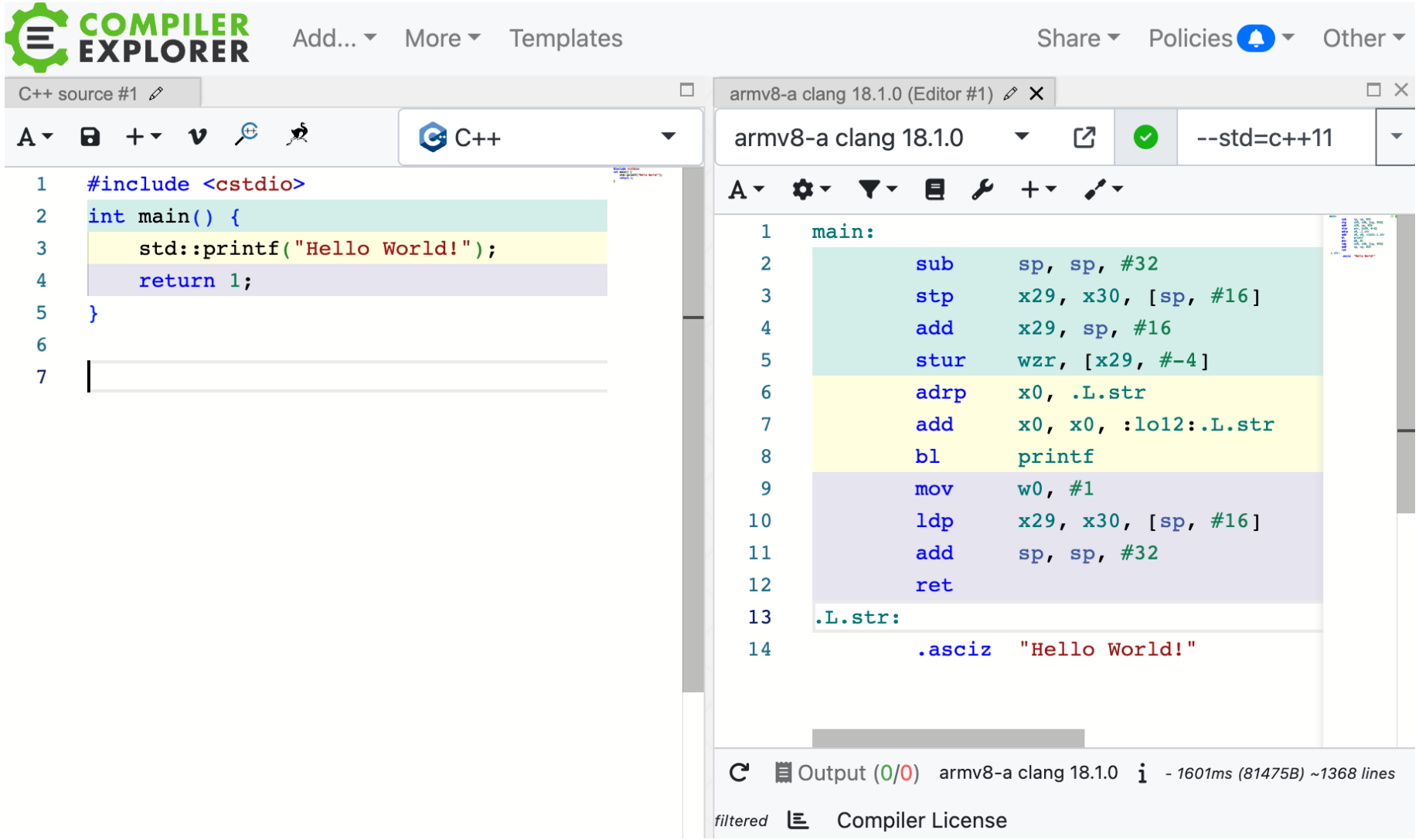
Application Binary Interface = ABI

https://developer.arm.com/documentation/102374/0101/Procedure-Call-Standard

https://learn.microsoft.com/en-us/cpp/build/arm64-windows-abi-conventions?view=msvc-170

https://developer.apple.com/documentation/xcode/writing-arm64-code-for-apple-platforms

Reading ARM Assembly (Revisited)



https://godbolt.org/z/zbn18xnnn

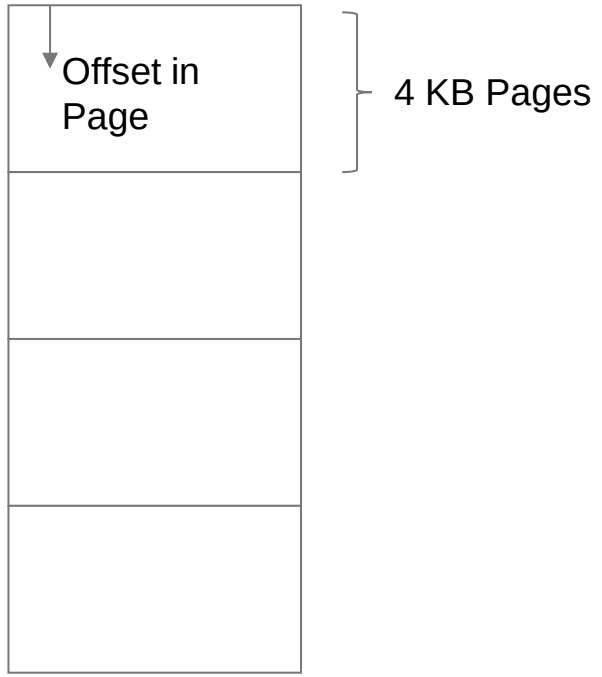
Reading ARM Assembly (Revisited)

```
1   main:
2           sub     sp, sp, #32
3           stp     x29, x30, [sp, #16]
4           add     x29, sp, #16
5           stur    wzr, [x29, #-4]
6           adrp    x0, .L.str
7           add     x0, x0, :lo12:.L.str
8           bl      printf
9           mov     w0, #1
10          ldp     x29, x30, [sp, #16]
11          add     sp, sp, #32
12          ret
13  .L.str:
14          .asciz  "Hello World"
```

We need to calculate 64-bit address

Each instruction is only 32 bits long!

Only 21 bits for address



Reading ARM Assembly (Revisited)

think-cell

```
0x2000 adrp x0, .ourstring
```

Page offset of `.ourstring: 1 << 12 + 0x2000`

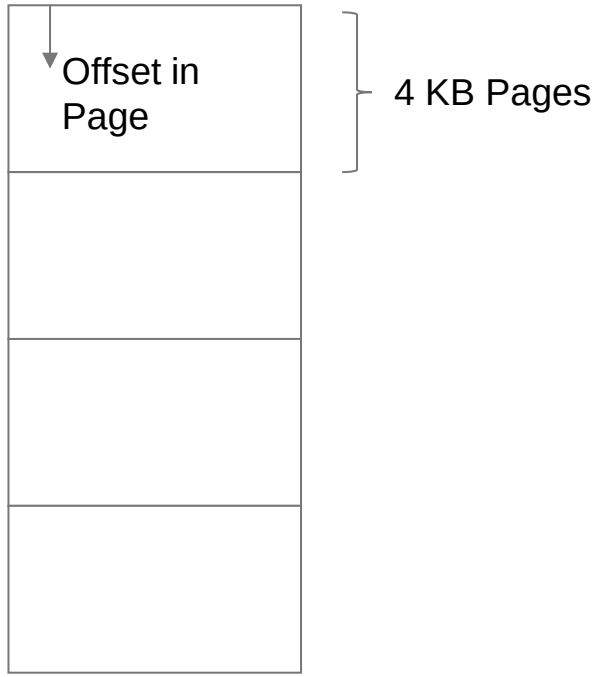
`x0 = 0x3000`

```
0x2004 add x0, x0, :lo12:.ourstring
```

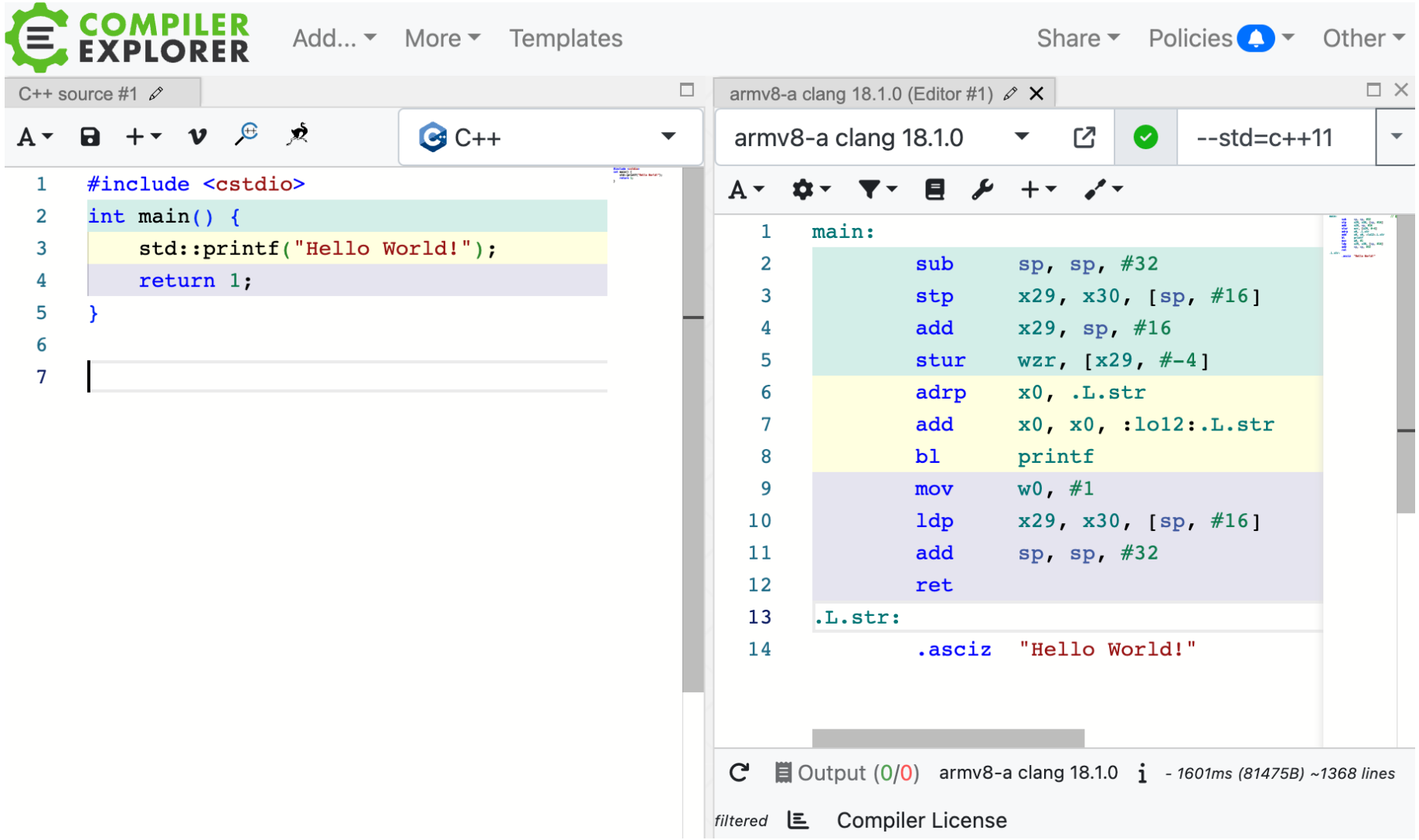
Offset inside page of `.ourstring: 0x3008 & 0xFFF = 0x8`

`x0 = 0x3008`

```
.ourstring
0x3008 "Hello World"
```



Reading ARM Assembly



https://godbolt.org/z/zbn18xnnn

ARM Memory Model

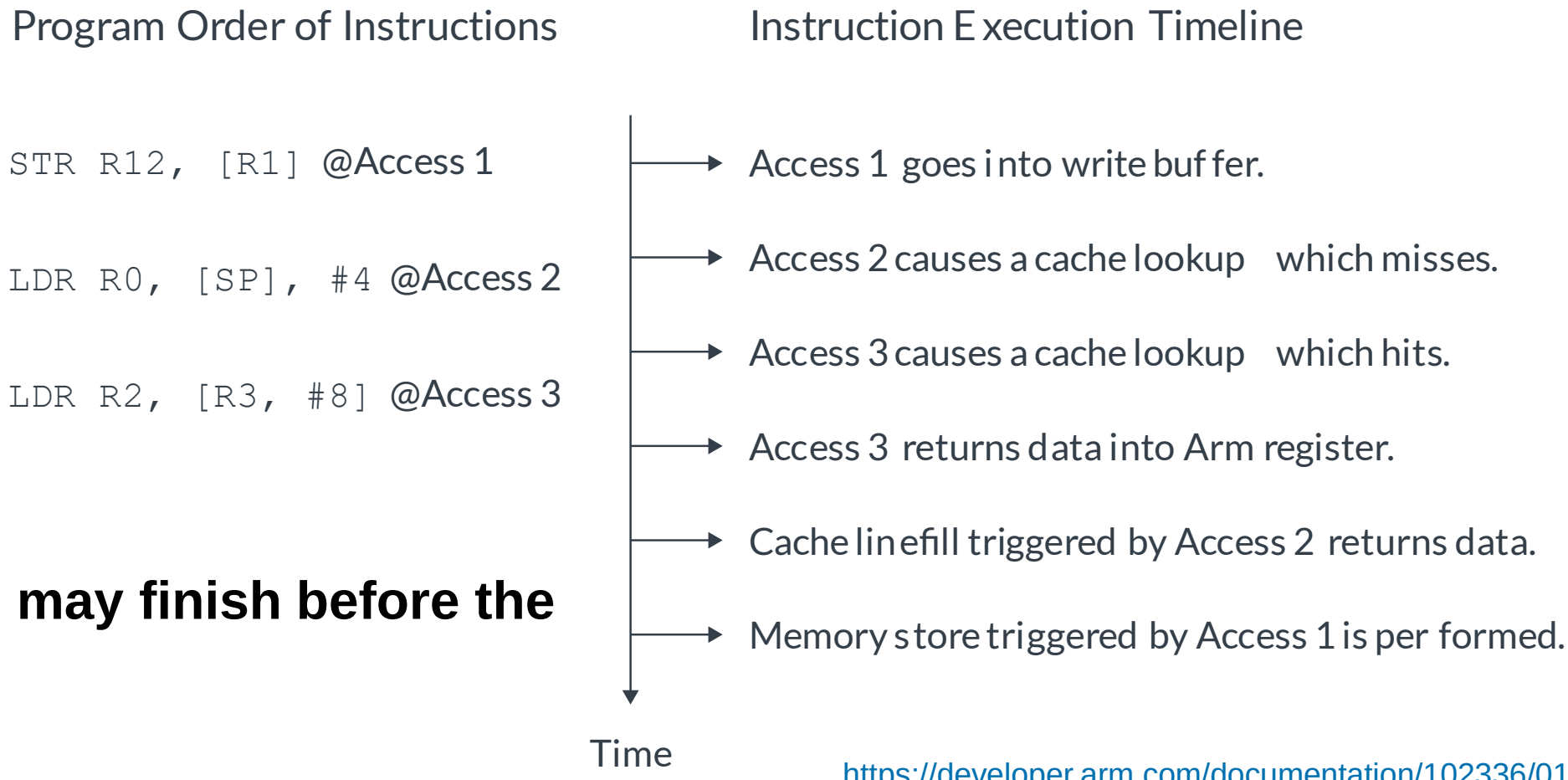
think-cell

Memory Systems, Ordering, and Barriers

https://developer.arm.com/documentation/102336/0100

“Armv8-A implements a weakly-ordered memory architecture. This architecture permits memory accesses which impose no dependencies to be issued or observed, and to complete in a different order from the order that is specified by the program order.”

ARM Memory Model



The loads may finish before the store!

https://developer.arm.com/documentation/102336/0100

ARM Memory Model

think-cell

CPU1

```
str [_x], 1 ; M0
str [_y], 1 ; M1
```

CPU2

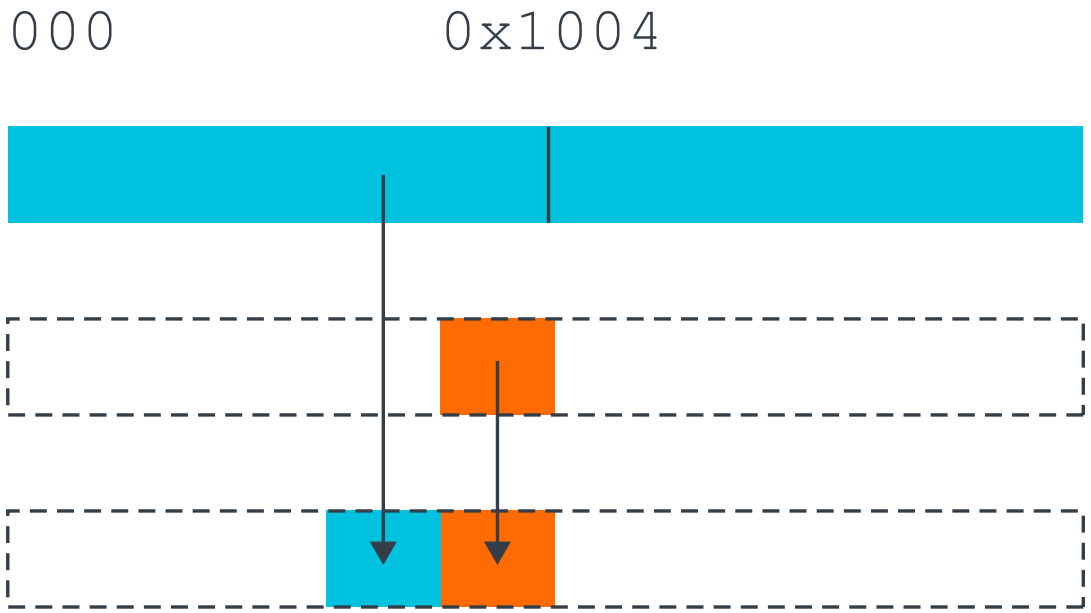
```
ldr r2, [_y] ; M2
ldr r1, [_x] ; M3
```

r1	r2				
0	0	M2	M3	M0	M1
0	1	M3	M0	M1	M2
1	0	M0	M2	M3	M1
1	1	M0	M1	M2	M3

ARM Memory Model

Access to the same bytes must be ordered

```
1.    STR   X0, [0x1000]
2.    STRB  W1, [0x1003]
3.    LDRH  W2, [0x1002]
```



https://developer.arm.com/documentation/102336/0100

ARM Memory Model

think-cell

Address Dependencies must be maintained

```
LDR X0, [X1]
STR X2, [X0] ; Result of previous load is the address in this
store.
```

Example 2:

```
LDR X0, [X1]
STR X2, [X5, X0] ; Result of previous load is used to
calculate the address.
```

ARM Memory Model

think-cell

How does this compare to Intel?

Intel® 64 Architecture Memory Ordering White Paper
https://www.cs.cmu.edu/~410-f10/doc/Intel_Reordering_318147.pdf

ARM Memory Model

think-cell

How does this compare to Intel?

Intel® 64 Architecture Memory Ordering White Paper
https://www.cs.cmu.edu/~410-f10/doc/Intel_Reordering_318147.pdf

1. Loads are not reordered with other loads
2. Stores are not reordered with other stores
3. Stores are not reordered with older loads
4. Loads may be reordered with older stores to different locations.

Intel Memory Model

think-cell

Intra-processor forwarding

CPU1

```
mov [_x], 1 ; M0
mov r1, [_x]
mov r2, [_y]
```

CPU2

```
mov [_y], 1 ; M1
mov r3, [_y]
mov r4, [_x]
```

`r2==0` and `r4==0` is allowed, no ordering between M0 and M1

Intel Memory Model

think-cell

Stores are transitively visible

CPU1	CPU2	CPU3
`mov [_x], 1 ; M0`	`mov r1, [_x] ; M1` `mov [_y], 1 ; M2`	`mov r2, [_y] ; M3` `mov r3, [_x] ; M4`

`r1==1, r2==1, r3==0` is not allowed

`r1==1 ->` M0 < M1 < M2 for all processors

No load reordering -> M3 < M4

`r2==1 ->` M2 < M3 and therefore M0 < M4

Intel Memory Model

Total store ordering

CPU1	CPU2	CPU3	CPU4
`mov [_x],1; M0`	`mov [_x],2; M1`	`mov r1,[_x]; M2` `mov r2,[_x]; M3`	`mov r3,[_x]; M4` `mov r4,[_x]; M5`

`r1==1, r2==2, r3==2, r4==1` is not allowed

All processors must agree on the order of M0 and M1

ARM Memory Model

think-cell

CPU1

```
str [_x], 1 ; M0
str [_y], 1 ; M1
```

CPU2

```
ldr r2, [_y] ; M2
ldr r1, [_x] ; M3
```

r1	r2				
0	0	M2	M3	M0	M1
0	1	M3	M0	M1	M2
1	0	M0	M2	M3	M1
1	1	M0	M1	M2	M3

ARM Memory Model

think-cell

CPU1

```
str [_x], 1 ; M0
str [_y], 1 ; M1
```

CPU2

```
ldr r2, [_y] ; M2
ldr r1, [_x] ; M3
```

r1	**r2**				
0	0	M2	M3	M0	M1
~~0~~	~~1~~	~~M3~~	~~M0~~	~~M1~~	~~M2~~
1	0	M0	M2	M3	M1
1	1	M0	M1	M2	M3

Not possible on Intel

ARM Memory Model

think-cell

Apple M processors implement both memory models

Allows performance comparison

Analyzing the memory ordering models of the Apple M1
Lars Wrenger∗, Dominik Töllner, Daniel Lohmann
Journal of Systems Architecture 149 (2024) 103102
https://www.sra.uni-hannover.de/Publications/2024/wrenger_24_jsa.pdf

Total-store ordering is 8.94% slower

ARM Memory Model

think-cell

Relaxed memory model only observable by other CPUs

On the same CPU, everything appears to happen in program order.

CPUs try to do out-of-order loads though!

https://github.com/travisdowns/uarch-bench/wiki/Memory-Disambiguation-on-Skylake

ARM and C++ abstract machine

think-cell

Evaluation order

https://godbolt.org/z/hePc6evc6

[intro.abstract]

“Conforming implementations […] need not copy or emulate the structure of the abstract machine. Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below.”

https://timsong-cpp.github.io/cppwp/intro#abstract-1

ARM and C++ abstract machine

think-cell

Aliasing rules

https://godbolt.org/z/Eod711Ez9

https://godbolt.org/z/333rcG6M1

ARM and C++ abstract machine

think-cell

Atomics

https://godbolt.org/z/8Trd6b16j

```
STR  #1, [X1]
STLR #1, [X3]  ; Cannot observe this STLR without observing the previous
STR.
```

https://developer.arm.com/documentation/102336/0100/Load-Acquire-and-Store-Release-instructions

Thank you

www.think-cell.com

think-cell

Join our team as a C++ Developer or Intern

think-cell.com/career