Introduction to non-Intel Architectures

Reverse Engineering – Introduction to non-Intel Architectures
Sergi Àlvarez aka pancake

Introduction to non-intel architectures

5. Introduction to non-intel architectures (3h)

Theory

- Explain how different compilers generate code for non intel architectures
- This is: ARM, ARM64, MIPS and PowerPC
- Introduction to Intermediate Languages (Focus on ESIL)
- Caveats when doing static and dynamic analysis on them
- Tooling: Radare2 and Frida

Practice

- Learning more on r2, analyzing different simple binaries
- Emulate portions of the binary using ESIL from r2
- Understand the register profiles, memory layout
- How to find data/code references and resolve jump tables.

Introduction

Background

- What’s a CPU
- What’s the instruction set?
- How are those instructions encoded?
- Endianness
- How compilers generate code for them?
- Toolchains
- Golang/NDK/Rust

What’s a CPU?

CPU stands for Central Processing Unit.

- Carries out the instructions to perform the basic arithmetic, logical, control and io operations specified by the instructions.
- Performs the Fetch-Decode-Execute loop
- Connected to the RAM thru the MMU
- Run code and interacts with devices and memory

There are several ways to implement a Turing machine, and those end up into different architectures.

Architectures Poll

How many architectures do we know?

Architectures Poll

How many architectures do we know?

- We have seen X86

But there are more!

Architectures

- 8086 / 286, i386, X86-64
- ARM / THUMB
- ARM64
- MIPS
- POWERPC
- SPARC
- AVR
- ARC
- Z80
- 8051
- ...

- Hexagon
- PTX
- Cell
- Z/Arch
- XCore
- PIC
- PDP
- M68K
- 6502
- VAX
- XAP4
- ...

Models and Coprocessors

- There are different implementations from different vendors that behave different and support different instruction sets.
- Some may include different coprocessors to enable floating point or multimedia instructions.
- Undefined behaviour may be considered a security issue
- Undocumented instructions
- Vulnerabilities in privileged instructions

Spectra and Meltdown

According to Wikipedia...

Spectra

- **Spectre** is a vulnerability that affects modern microprocessors that perform branch prediction.[1][2][3]
  On most processors, the speculative execution resulting from a branch misprediction may leave observable side effects that may reveal private data to attackers.
  For example, if the pattern of memory accesses performed by such speculative execution depends on private data, the resulting state of the data cache constitutes a side channel through which an attacker may be able to extract information about the private data using a timing attack.[4][5][6]

Meltdown

- **Meltdown** is a hardware vulnerability affecting Intel x86 microprocessors, IBM POWER processors, and some ARM-based microprocessors.[1][2][3]
  It allows a rogue process to read all memory, even when it is not authorized to do so.

Spectra and undefined behaviour

- Yes, CPUs can be also buggy, and sometimes those bugs can be exploited to escalate privileges, cause a hw reboot or access to restricted resources.
- In this case, the cache was tricked with timings in order to guess the contents of memory from inaccessible memory. This is, being able to dump kernel memory from userland.
- Those bugs exist in all intel cpus, but also appear to be available on some ARMs
- Also affects many virtualized environments, like those running in the Cloud.

Protecting from Spectra

- No way to fix the wirings.
- As long as it’s a hardware design problem it is not easy to deploy an update because as Intel said, this can’t be fixed with a microcode update.
- Operating System vendors had to think in ways to protect their users, but how to fix a hardware bug with software?
    - Reducing the attack vector (verify apps before being published in the AppStore)
    - As long as the attack can be crafted in JS exploiting browser optimizations, V8, WebKit, Mozilla… provided updates
    - Perform static analysis looking for code blocks that use those special cache-related instructions in loop.

Protecting from Spectra

- https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/
  - The resolution of performance.now() will be reduced to 20µs. (UPDATE: see the MDN documentation for performance.now for up-to-date precision information.)
  - The SharedArrayBuffer feature is being disabled by default.

- https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/
  - Same as Mozilla, but introduces branchless security
  - Function pointers are indexed within an array
  - Array is accessed thru a binary mask to avoid oob indexes

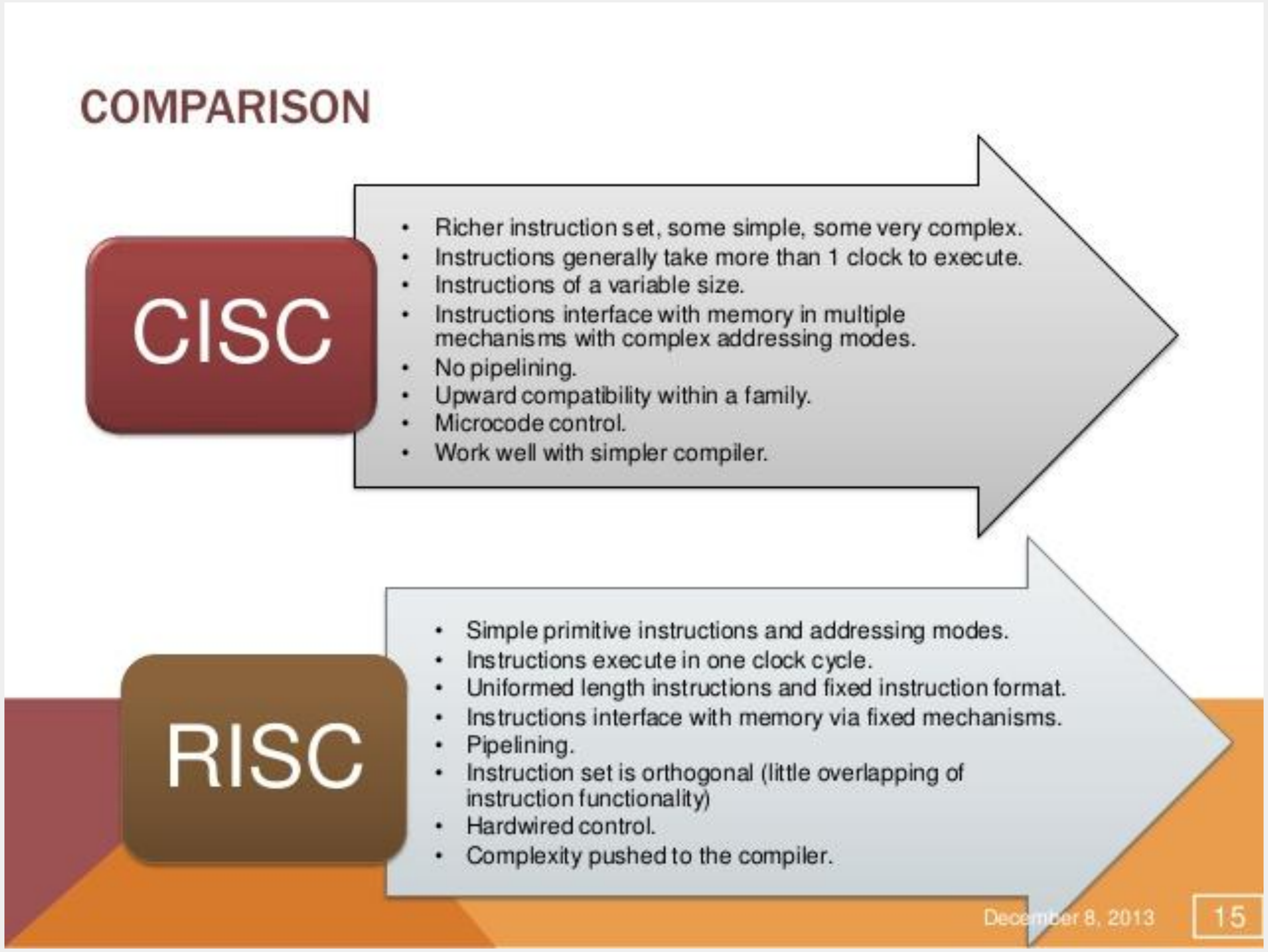
Microcode

I’ve mentioned microcode in the previous slide, so let’s explain what’s that.

- Each instruction can be defined in a low level assembly language named microcode.
- Microcode is some kind of privative RISC CPU implemented on each CISC
- Most of the CPUs supporting microcode have ways to update this “firmware” to introduce new instructions or fix bugs in other instructions.
- Microcode assembly should have no side effects and must be concise.

[CR]ISC

Types of Instruction Sets



COMPARISON

CISC

- Richer instruction set, some simple, some very complex.
- Instructions generally take more than 1 clock to execute.
- Instructions of a variable size.
- Instructions interface with memory in multiple mechanisms with complex addressing modes.
- No pipelining.
- Upward compatibility within a family.
- Microcode control.
- Work well with simpler compiler.

RISC

- Simple primitive instructions and addressing modes.
- Instructions execute in one clock cycle.
- Uniformed length instructions and fixed instruction format.
- Instructions interface with memory via fixed mechanisms.
- Pipelining.
- Instruction set is orthogonal (little overlapping of instruction functionality)
- Hardwired control.
- Complexity pushed to the compiler.

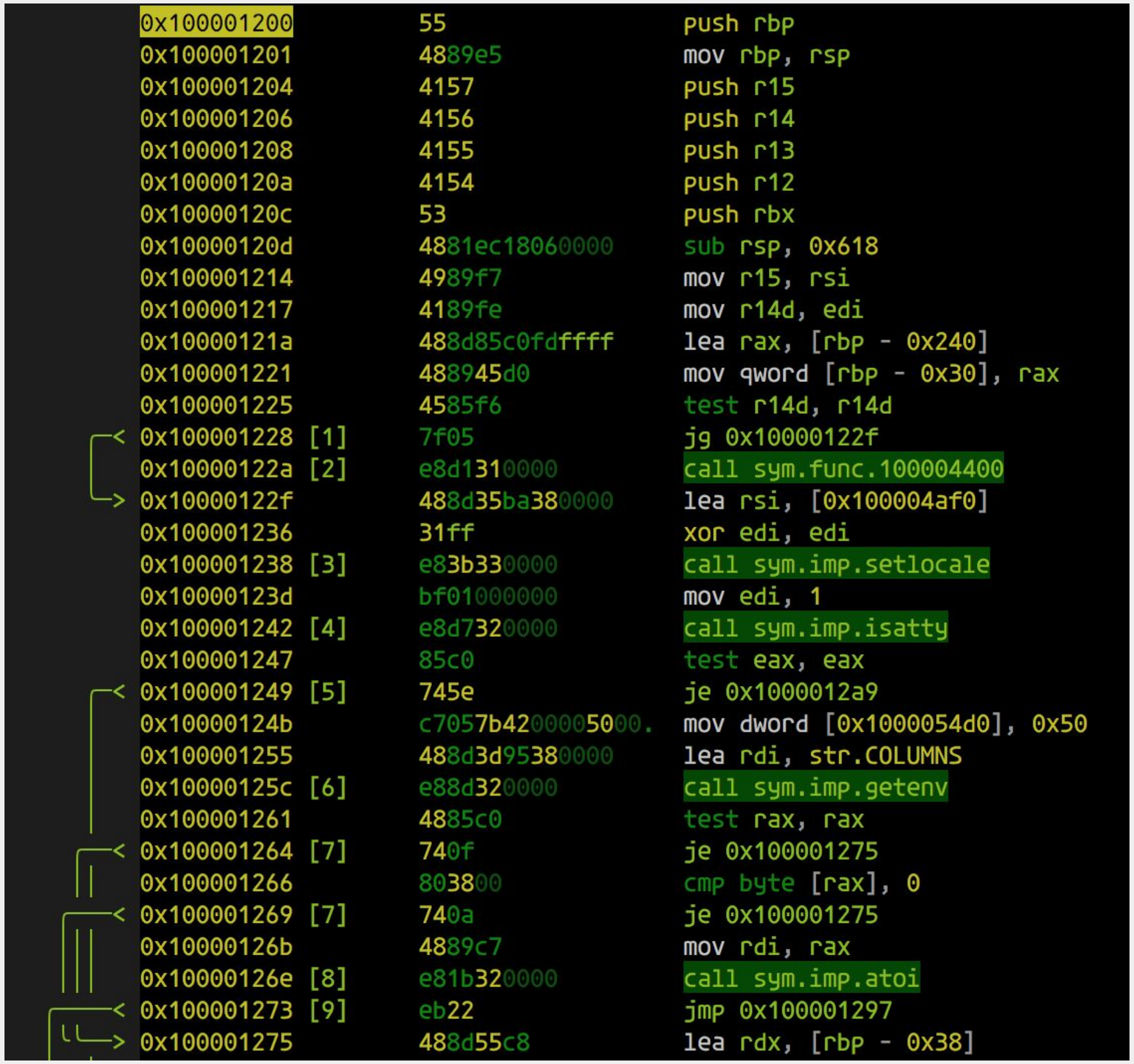
December 8, 2013 15

CISC

- Complex instruction sets use to have variable cost instructions, because what matters is to have less instructions in the code to achieve more.
- This forced intel to implement a complex MMU and variable length instructions to solve the problem of code size.
- Multiple clocks and parallel execution pipelines.
- Memory access and computation in a single instruction.

CISC: example

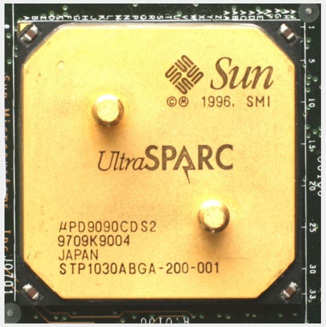
- mov performs additional arithmetics
- Variable size instructions
- Comparison with memory access
- No implicit alignment in instructions



```
0x100001200         55                  push rbp
0x100001201         4889e5              mov rbp, rsp
0x100001204         4157                push r15
0x100001206         4156                push r14
0x100001208         4155                push r13
0x10000120a         4154                push r12
0x10000120c         53                  push rbx
0x10000120d         4881ec18060000      sub rsp, 0x618
0x100001214         4989f7              mov r15, rsi
0x100001217         4189fe              mov r14d, edi
0x10000121a         488d85c0fdffff      lea rax, [rbp - 0x240]
0x100001221         488945d0            mov qword [rbp - 0x30], rax
0x100001225         4585f6              test r14d, r14d
0x100001228 [1]     7f05                jg 0x10000122f
0x10000122a [2]     e8d1310000          call sym.func.100004400
0x10000122f         488d35ba380000      lea rsi, [0x100004af0]
0x100001236         31ff                xor edi, edi
0x100001238 [3]     e83b330000          call sym.imp.setlocale
0x10000123d         bf01000000          mov edi, 1
0x100001242 [4]     e8d7320000          call sym.imp.isatty
0x100001247         85c0                test eax, eax
0x100001249 [5]     745e                je 0x1000012a9
0x10000124b         c7057b42000005000.  mov dword [0x1000054d0], 0x50
0x100001255         488d3d95380000      lea rdi, str.COLUMNS
0x10000125c [6]     e88d320000          call sym.imp.getenv
0x100001261         4885c0              test rax, rax
0x100001264 [7]     740f                je 0x100001275
0x100001266         803800              cmp byte [rax], 0
0x100001269 [7]     740a                je 0x100001275
0x10000126b         4889c7              mov rdi, rax
0x10000126e [8]     e81b320000          call sym.imp.atoi
0x100001273 [9]     eb22                jmp 0x100001297
0x100001275         488d55c8            lea rdx, [rbp - 0x38]
```

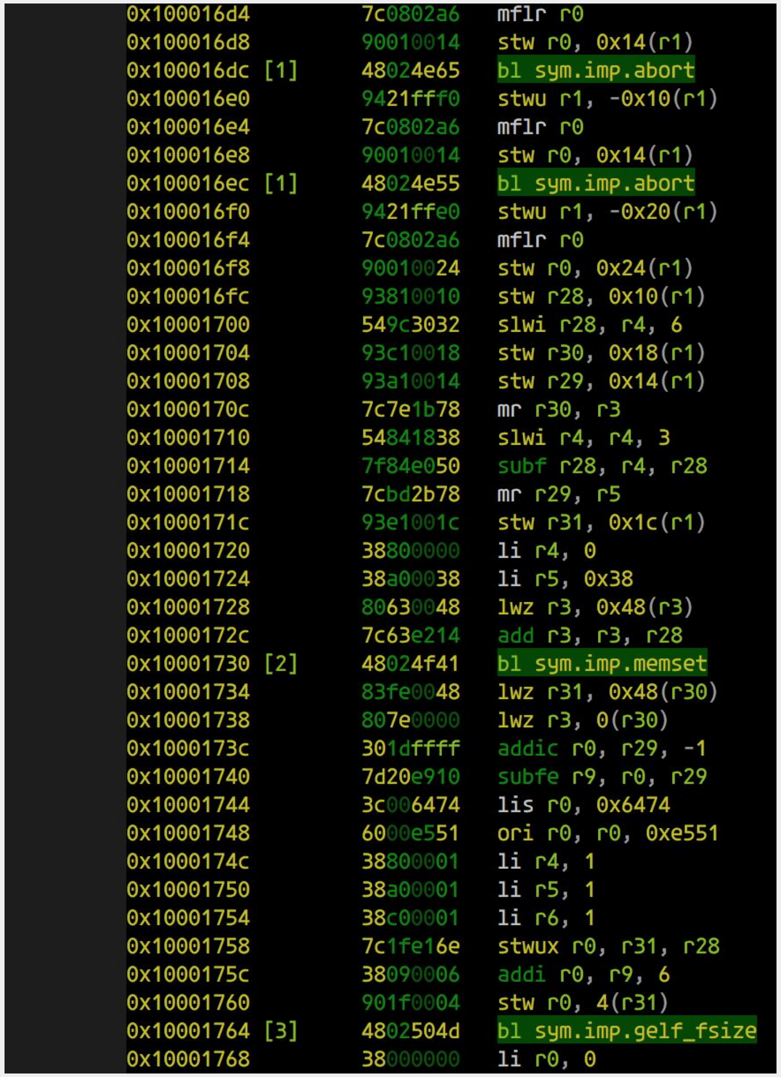
RISC

- Small and simple instruction sets trying to make them have pretty similar execution costs.
- Load/Store instructions to access memory.
- Requires more instructions to achieve the same as a single CISC instruction.
- Simplified MMU allows faster memory access without requiring bigger caches.
- Reduced complexity in design.



RISC: example

- **LI** : Load immediate
- **ORI**: only using registers and immediates
- Memory access instructions are **LOAD** or **STORE**, those don't perform any additional logic. (see **ST** and **LW**)
- Use **asm.describe=true**



```
0x100016d4          7c0802a6    mflr r0
0x100016d8          90010014    stw r0, 0x14(r1)
0x100016dc [1]      48024e65    bl sym.imp.abort
0x100016e0          9421fff0    stwu r1, -0x10(r1)
0x100016e4          7c0802a6    mflr r0
0x100016e8          90010014    stw r0, 0x14(r1)
0x100016ec [1]      48024e55    bl sym.imp.abort
0x100016f0          9421ffe0    stwu r1, -0x20(r1)
0x100016f4          7c0802a6    mflr r0
0x100016f8          90010024    stw r0, 0x24(r1)
0x100016fc          93810010    stw r28, 0x10(r1)
0x10001700          549c3032    slwi r28, r4, 6
0x10001704          93c10018    stw r30, 0x18(r1)
0x10001708          93a10014    stw r29, 0x14(r1)
0x1000170c          7c7e1b78    mr r30, r3
0x10001710          54841838    slwi r4, r4, 3
0x10001714          7f84e050    subf r28, r4, r28
0x10001718          7cbd2b78    mr r29, r5
0x1000171c          93e1001c    stw r31, 0x1c(r1)
0x10001720          38800000    li r4, 0
0x10001724          38a00038    li r5, 0x38
0x10001728          80630048    lwz r3, 0x48(r3)
0x1000172c          7c63e214    add r3, r3, r28
0x10001730 [2]      48024f41    bl sym.imp.memset
0x10001734          83fe0048    lwz r31, 0x48(r30)
0x10001738          807e0000    lwz r3, 0(r30)
0x1000173c          301dffff    addic r0, r29, -1
0x10001740          7d20e910    subfe r9, r0, r29
0x10001744          3c006474    lis r0, 0x6474
0x10001748          6000e551    ori r0, r0, 0xe551
0x1000174c          38800001    li r4, 1
0x10001750          38a00001    li r5, 1
0x10001754          38c00001    li r6, 1
0x10001758          7c1fe16e    stwux r0, r31, r28
0x1000175c          38090006    addi r0, r9, 6
0x10001760          901f0004    stw r0, 4(r31)
0x10001764 [3]      4802504d    bl sym.imp.gelf_fsize
0x10001768          38000000    li r0, 0
```

Fixed Instruction Size

- Most RISC architectures have a simplified memory management that restricts memory accesses to be bound to a specific alignment.
- This makes them design a fixed length instruction of that size. Which can be 2, 4 or even 8.
- In Thumb2 mode, instructions can be of 2 or 4 in size, but they must be always aligned to 4

Where Are They?

Where can we find them?

X86 is available in most desktop and servers nowadays.
So, where can we find those other architectures?

- IoT
- routers
- mobile
- smartwatches
- cars
- TVs
- sensors

Why are they interesting?

- Low power consumption
- Used in many more places than we can imagine
- Smaller sockets
- Optimized for low memory constraints
- Support better thermal conditions
- Embedded devices
- Proprietary hardware or CPU extensions
- Licensing
- Big Endian (better for networking)

ARM

ARM

- Originally designed from the 6502 CPU
- Quickly evolved into a consortium with multiple variants and implementations.
- Each CPU used different instruction sets, endianness and toolchains, so developing for ARM has been always a embedded-like style.
- The choice of this arch in mobile phones forced the market to make it standard this and simplify the toolchain ecosystem.

THUMB

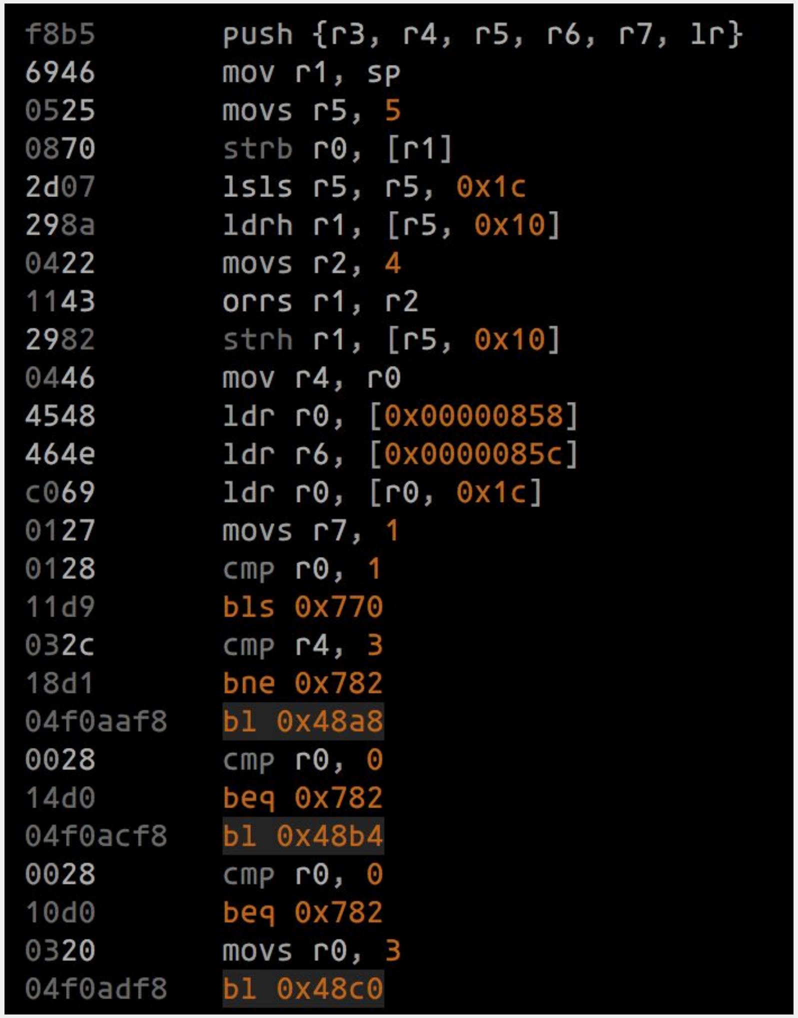
- Thumb and Thumb2 are the simpler than ARM
- Thumb are 2 byte in length
- Thumb2 are 2 or 4
- Focus on code density.
- Reuses same registers as ARM
- Switching from/to ARM by setting the T bit of the CPSR

ARM64

- Most modern design, evolution of the ARM architecture.
- Focus on supporting 64 bit memory read/writes
- Access to more RAM
- Simplified instruction set, compared to ARM
- No thumb mode.

Identify asm.bits

Take 3 raw binaries and identify if those are ARM, THUMB or ARM64.



```
f8b5        push {r3, r4, r5, r6, r7, lr}
6946        mov r1, sp
0525        movs r5, 5
0870        strb r0, [r1]
2d07        lsls r5, r5, 0x1c
298a        ldrh r1, [r5, 0x10]
0422        movs r2, 4
1143        orrs r1, r2
2982        strh r1, [r5, 0x10]
0446        mov r4, r0
4548        ldr r0, [0x00000858]
464e        ldr r6, [0x0000085c]
c069        ldr r0, [r0, 0x1c]
0127        movs r7, 1
0128        cmp r0, 1
11d9        bls 0x770
032c        cmp r4, 3
18d1        bne 0x782
04f0aaf8    bl 0x48a8
0028        cmp r0, 0
14d0        beq 0x782
04f0acf8    bl 0x48b4
0028        cmp r0, 0
10d0        beq 0x782
0320        movs r0, 3
04f0adf8    bl 0x48c0
```

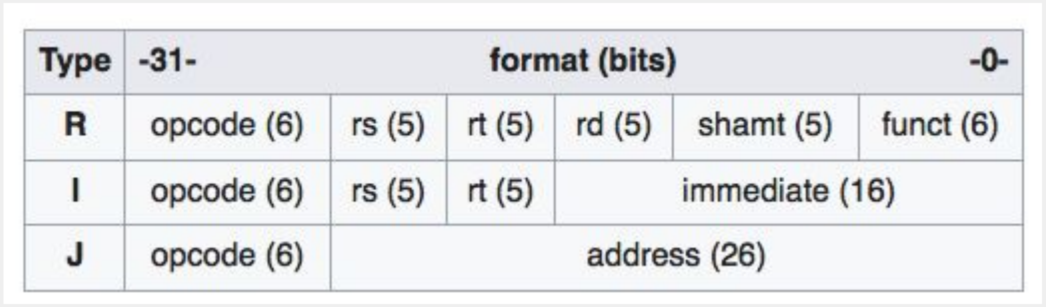
MIPS

MIPS

- Microprocessor without Interlocked Pipeline Stages
- Designed in 1985 (pretty modern)
- Have 32 registers
  - $0 is hardwired to ground (always zero)
- 32 → 64bit architecture
- Supports several optional CPU extensions (co-processors) (up to 4)
  - SIMD, MDMX, compression, crypto, threading, geometry ...
  - CP0 = system control coprocessor
  - CP1, optional FPU
  - CP2/3 multimedia or other
- Mostly used in Routers, but has been used in N64, PS, PS2 and PSP.

MIPS Instruction Set

- Load, Store (memory + regs)
- ALU (addition, substraction, multiply, and, xor …)
- Shifts (logical, arithmetic, + variable)
- Jump + Branch
- Exceptions (syscall + break)
- Fixed in size (4 bytes), easy to encode or decode.



Type	-31-			format (bits)		-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

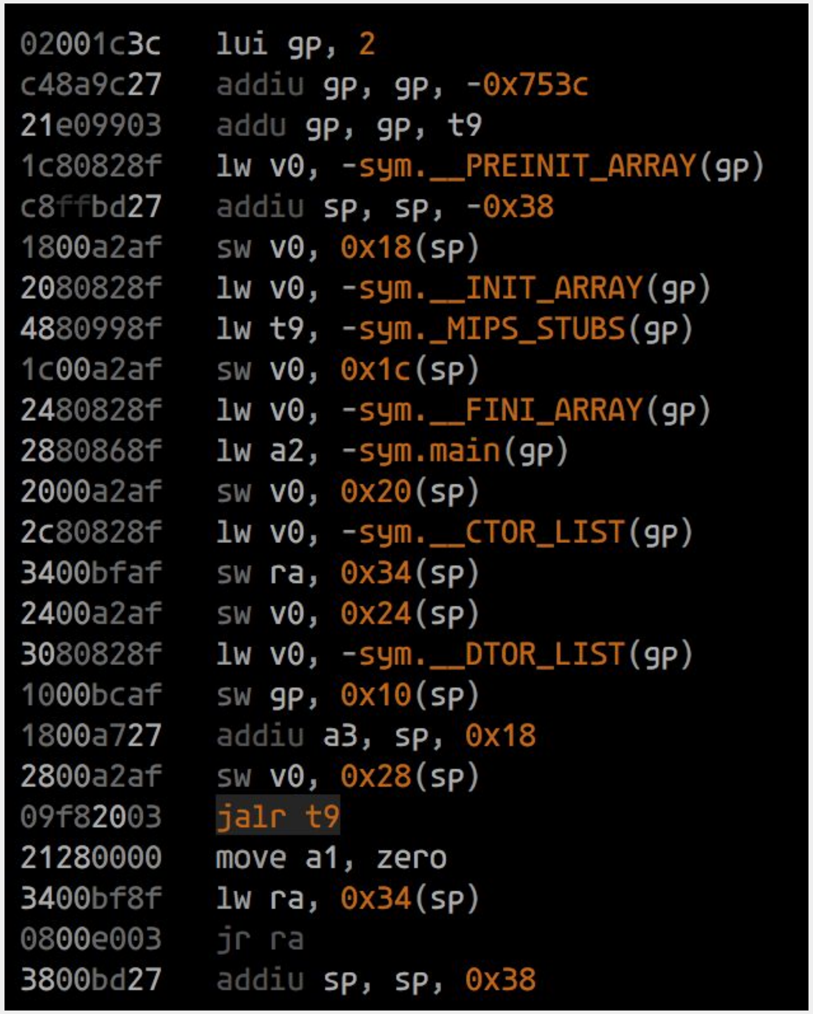
Delay Slot

- Some architectures like MIPS or SPARC have thing named “delay slot”.
- Instructions are fetched from aligned memory by blocks. This means that each block will be executed until the last instruction of the block is done.
- Compilers enforce those alignments as well as reorder the instructions to exploit the maximum performance of the pipeline.
- For example. Incrementing the counter register while the jump destination is computed in a loop.

MIPS Disassembly

Take a MIPS binary

- Find Main
- Resolve String Xref
-



```
02001c3c    lui gp, 2
c48a9c27    addiu gp, gp, -0x753c
21e09903    addu gp, gp, t9
1c80828f    lw v0, -sym.__PREINIT_ARRAY(gp)
c8ffbd27    addiu sp, sp, -0x38
1800a2af    sw v0, 0x18(sp)
2080828f    lw v0, -sym.__INIT_ARRAY(gp)
4880998f    lw t9, -sym._MIPS_STUBS(gp)
1c00a2af    sw v0, 0x1c(sp)
2480828f    lw v0, -sym.__FINI_ARRAY(gp)
2880868f    lw a2, -sym.main(gp)
2000a2af    sw v0, 0x20(sp)
2c80828f    lw v0, -sym.__CTOR_LIST(gp)
3400bfaf    sw ra, 0x34(sp)
2400a2af    sw v0, 0x24(sp)
3080828f    lw v0, -sym.__DTOR_LIST(gp)
1000bcaf    sw gp, 0x10(sp)
1800a727    addiu a3, sp, 0x18
2800a2af    sw v0, 0x28(sp)
09f82003    jalr t9
21280000    move a1, zero
3400bf8f    lw ra, 0x34(sp)
0800e003    jr ra
3800bd27    addiu sp, sp, 0x38
```

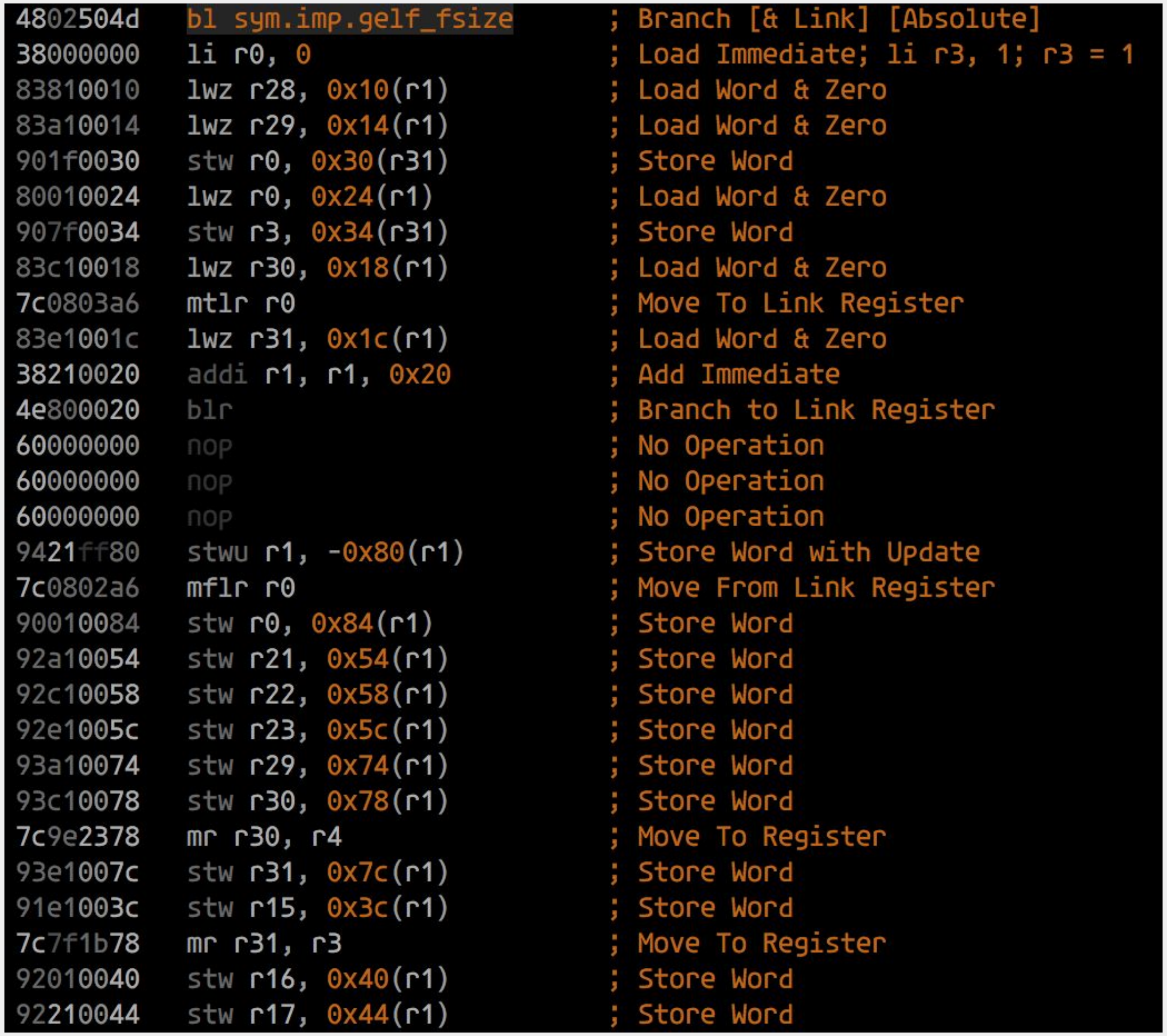
PowerPC

PowerPC

- Stands for Performance Optimization With Enhanced RISC
- Introduced in 1992 by IBM
- Mainly used as big-endian, but also has little-endian mode
- 64bit spec which is backwards compat with 32bit arch
- single and double precision FPU
- Conditional move instructions
- Extensions: AltiVec APU
- Used by Apple, Amiga, Wii, WiiU Xbox 360, GameCube, PS3
    - Apple moves: Motorola → PowerPC X86 → ARM64
- SuperScalar architecture

PowerPC: How it looks

- Toggle Endian
- Observe other characteristics
- ...



```
4802504d   bl sym.imp.gelf_fsize      ; Branch [& Link] [Absolute]
38000000   li r0, 0                   ; Load Immediate; li r3, 1; r3 = 1
83810010   lwz r28, 0x10(r1)          ; Load Word & Zero
83a10014   lwz r29, 0x14(r1)          ; Load Word & Zero
901f0030   stw r0, 0x30(r31)          ; Store Word
80010024   lwz r0, 0x24(r1)           ; Load Word & Zero
907f0034   stw r3, 0x34(r31)          ; Store Word
83c10018   lwz r30, 0x18(r1)          ; Load Word & Zero
7c0803a6   mtlr r0                    ; Move To Link Register
83e1001c   lwz r31, 0x1c(r1)          ; Load Word & Zero
38210020   addi r1, r1, 0x20          ; Add Immediate
4e800020   blr                        ; Branch to Link Register
60000000   nop                        ; No Operation
60000000   nop                        ; No Operation
60000000   nop                        ; No Operation
9421ff80   stwu r1, -0x80(r1)         ; Store Word with Update
7c0802a6   mflr r0                    ; Move From Link Register
90010084   stw r0, 0x84(r1)           ; Store Word
92a10054   stw r21, 0x54(r1)          ; Store Word
92c10058   stw r22, 0x58(r1)          ; Store Word
92e1005c   stw r23, 0x5c(r1)          ; Store Word
93a10074   stw r29, 0x74(r1)          ; Store Word
93c10078   stw r30, 0x78(r1)          ; Store Word
7c9e2378   mr r30, r4                 ; Move To Register
93e1007c   stw r31, 0x7c(r1)          ; Store Word
91e1003c   stw r15, 0x3c(r1)          ; Store Word
7c7f1b78   mr r31, r3                 ; Move To Register
92010040   stw r16, 0x40(r1)          ; Store Word
92210044   stw r17, 0x44(r1)          ; Store Word
```

SPARC

Sun
© Ⓜ 1996, SMI
Ultra SPARC II
C9209505 9950
PG2.0 GS2 USA
STP 1031 LGA
980
-300
EDO

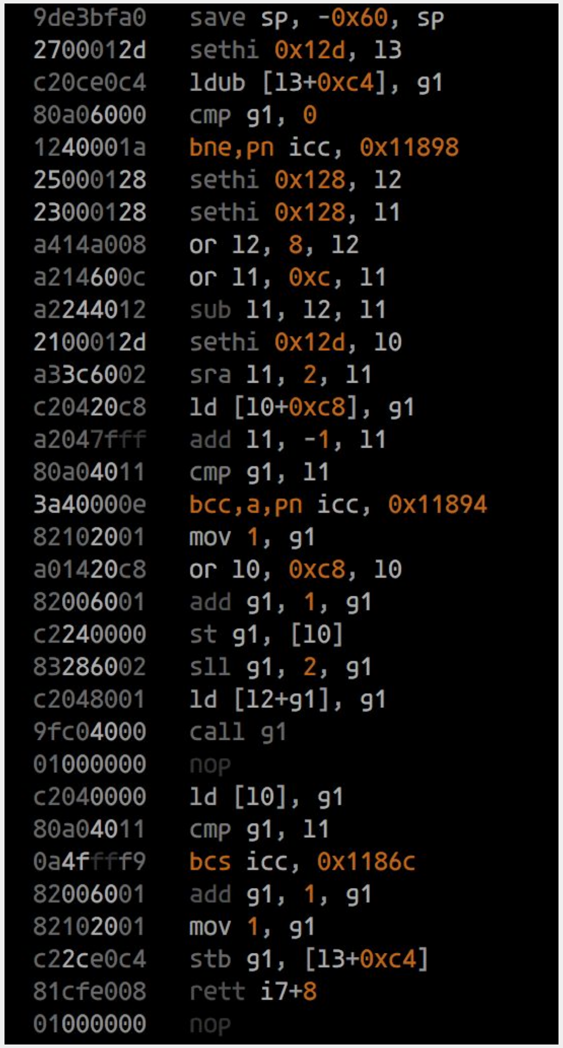
SPARC

- Stands for Scalable Processor Architecture
    - Designed by SUN as a replacement for the Motorola 68K
- Introduced in 1986 (SPARCv7)
- Started as a 32bit architecture, but it's 64bit since 1993
- Latest processors support SMP, NUMA
- Design was standardized and developed by the committee members
- Only used in High-End servers
- Focus on stability, constant response times and uptime
- 160 GPR originally. Oracle in 2015: from 72 to 640 GPR
- "Scalable" means that the specification allows to drop features without breaking to support from embeddeds to high end servers
- SPARCv8 have 16 double precision registers
    - Which can be used as 8 quad-precision fpu
- Only Big Endian

SPARC

SPARC

- This is how SPARC code looks like
- Notice the NOPs after the CALL instructions
- Notice the alignment of destinations
- How stackframe is constructed



```
9de3bfa0    save sp, -0x60, sp
2700012d    sethi 0x12d, l3
c20ce0c4    ldub [l3+0xc4], g1
80a06000    cmp g1, 0
1240001a    bne,pn icc, 0x11898
25000128    sethi 0x128, l2
23000128    sethi 0x128, l1
a414a008    or l2, 8, l2
a214600c    or l1, 0xc, l1
a2244012    sub l1, l2, l1
2100012d    sethi 0x12d, l0
a33c6002    sra l1, 2, l1
c20420c8    ld [l0+0xc8], g1
a2047fff    add l1, -1, l1
80a04011    cmp g1, l1
3a40000e    bcc,a,pn icc, 0x11894
82102001    mov 1, g1
a01420c8    or l0, 0xc8, l0
82006001    add g1, 1, g1
c2240000    st g1, [l0]
83286002    sll g1, 2, g1
c2048001    ld [l2+g1], g1
9fc04000    call g1
01000000    nop
c2040000    ld [l0], g1
80a04011    cmp g1, l1
0a4ffff9    bcs icc, 0x1186c
82006001    add g1, 1, g1
82102001    mov 1, g1
c22ce0c4    stb g1, [l3+0xc4]
81cfe008    rett i7+8
01000000    nop
```

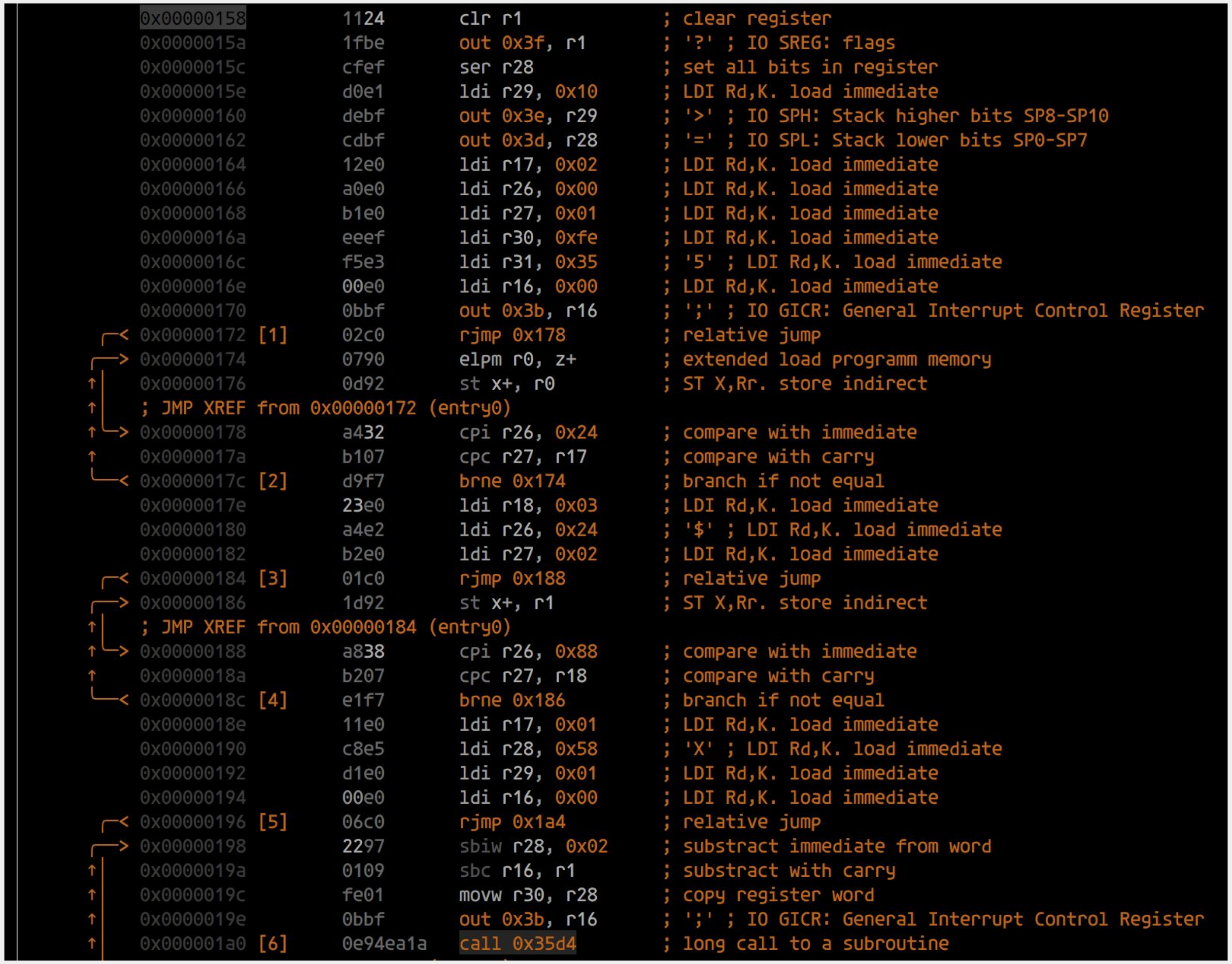
AVR

AVR

- Designed by two Norwegian students in 1997
- Developed by Atmel, being Arduino the most famous
- 8 bit CPU
- AVR32 appeared in 2006
- It’s a microcontroller
- Program memory inside the chip (from 16KB to 384KB)
- There’s also AVR32 (32bit arch with SIMD, DSP …)
  - Incompatible instruction set
- Advanced microcontroller

AVR

●



```
        0x00000158          1124       clr r1              ; clear register
        0x0000015a          1fbe       out 0x3f, r1        ; '?' ; IO SREG: flags
        0x0000015c          cfef       ser r28             ; set all bits in register
        0x0000015e          d0e1       ldi r29, 0x10       ; LDI Rd,K. load immediate
        0x00000160          debf       out 0x3e, r29       ; '>' ; IO SPH: Stack higher bits SP8-SP10
        0x00000162          cdbf       out 0x3d, r28       ; '=' ; IO SPL: Stack lower bits SP0-SP7
        0x00000164          12e0       ldi r17, 0x02       ; LDI Rd,K. load immediate
        0x00000166          a0e0       ldi r26, 0x00       ; LDI Rd,K. load immediate
        0x00000168          b1e0       ldi r27, 0x01       ; LDI Rd,K. load immediate
        0x0000016a          eeef       ldi r30, 0xfe       ; LDI Rd,K. load immediate
        0x0000016c          f5e3       ldi r31, 0x35       ; '5' ; LDI Rd,K. load immediate
        0x0000016e          00e0       ldi r16, 0x00       ; LDI Rd,K. load immediate
        0x00000170          0bbf       out 0x3b, r16       ; ';' ; IO GICR: General Interrupt Control Register
    ┌─< 0x00000172 [1]      02c0       rjmp 0x178          ; relative jump
   ┌──> 0x00000174          0790       elpm r0, z+         ; extended load programm memory
   ↑│   0x00000176          0d92       st x+, r0           ; ST X,Rr. store indirect
   ↑│   ; JMP XREF from 0x00000172 (entry0)
   ↑└─> 0x00000178          a432       cpi r26, 0x24       ; compare with immediate
   ↑    0x0000017a          b107       cpc r27, r17        ; compare with carry
   └──< 0x0000017c [2]      d9f7       brne 0x174          ; branch if not equal
        0x0000017e          23e0       ldi r18, 0x03       ; LDI Rd,K. load immediate
        0x00000180          a4e2       ldi r26, 0x24       ; '$' ; LDI Rd,K. load immediate
        0x00000182          b2e0       ldi r27, 0x02       ; LDI Rd,K. load immediate
    ┌─< 0x00000184 [3]      01c0       rjmp 0x188          ; relative jump
   ┌──> 0x00000186          1d92       st x+, r1           ; ST X,Rr. store indirect
   ↑│   ; JMP XREF from 0x00000184 (entry0)
   ↑└─> 0x00000188          a838       cpi r26, 0x88       ; compare with immediate
   ↑    0x0000018a          b207       cpc r27, r18        ; compare with carry
   └──< 0x0000018c [4]      e1f7       brne 0x186          ; branch if not equal
        0x0000018e          11e0       ldi r17, 0x01       ; LDI Rd,K. load immediate
        0x00000190          c8e5       ldi r28, 0x58       ; 'X' ; LDI Rd,K. load immediate
        0x00000192          d1e0       ldi r29, 0x01       ; LDI Rd,K. load immediate
        0x00000194          00e0       ldi r16, 0x00       ; LDI Rd,K. load immediate
    ┌─< 0x00000196 [5]      06c0       rjmp 0x1a4          ; relative jump
   ┌──> 0x00000198          2297       sbiw r28, 0x02      ; substract immediate from word
   ↑│   0x0000019a          0109       sbc r16, r1         ; substract with carry
   ↑│   0x0000019c          fe01       movw r30, r28       ; copy register word
   ↑│   0x0000019e          0bbf       out 0x3b, r16       ; ';' ; IO GICR: General Interrupt Control Register
   ↑│   0x000001a0 [6]      0e94ea1a   call 0x35d4         ; long call to a subroutine
```

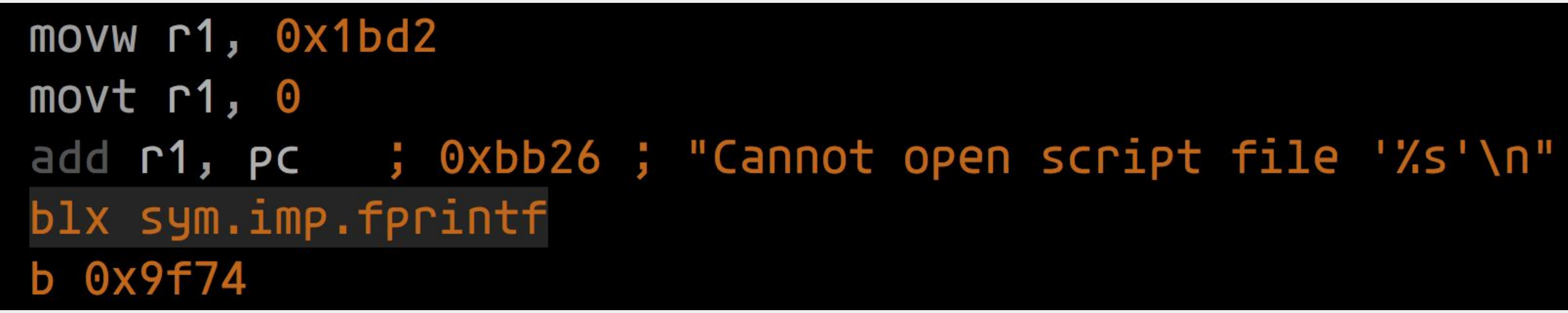
Relative Code

Relative Code

- As long as you can’t fit a 64 bit or 32 bit address encoded inside a 32 bit instruction, RISC architectures compute those addresses in different ways:
  - Loading a dword from a relative address from here
  - Compute the address by code, more than 1 instruction
  - Use PC as relative base address
- On arm64 there’s adrp which loads a relative address from PC aligned to 4096, which is making it more complex to assemble or relocate

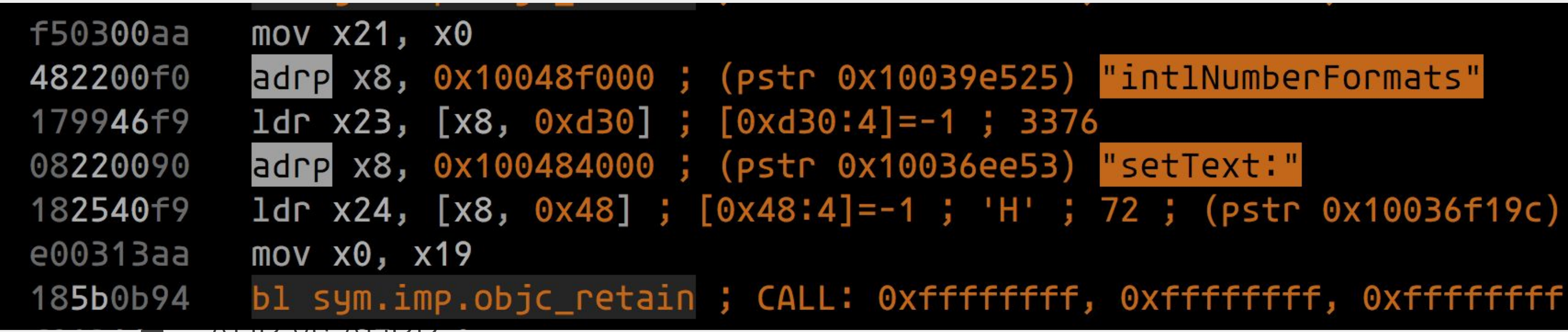
Relative Code: ARM

- ARM/Thumb code is usually PIC by default, because PC is r15 and can be accessed from any instruction without adding an extra cost



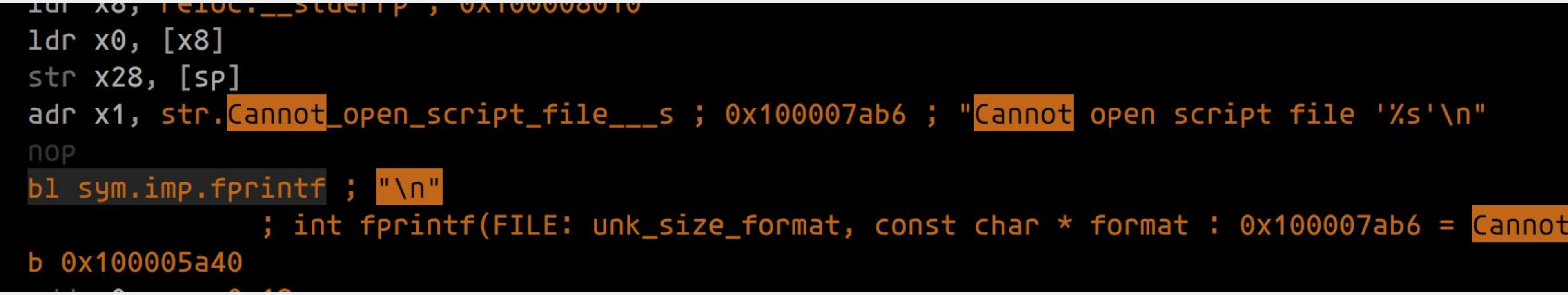
```
movw r1, 0x1bd2
movt r1, 0
add r1, pc    ; 0xbb26 ; "Cannot open script file '%s'\n"
blx sym.imp.fprintf
b 0x9f74
```

Relative Code: ARM64



```
f50300aa    mov x21, x0
482200f0    adrp x8, 0x10048f000 ; (pstr 0x10039e525) "intlNumberFormats"
179946f9    ldr x23, [x8, 0xd30] ; [0xd30:4]=-1 ; 3376
08220090    adrp x8, 0x100484000 ; (pstr 0x10036ee53) "setText:"
182540f9    ldr x24, [x8, 0x48] ; [0x48:4]=-1 ; 'H' ; 72 ; (pstr 0x10036f19c)
e00313aa    mov x0, x19
185b0b94    bl sym.imp.objc_retain ; CALL: 0xffffffff, 0xffffffff, 0xffffffff
```

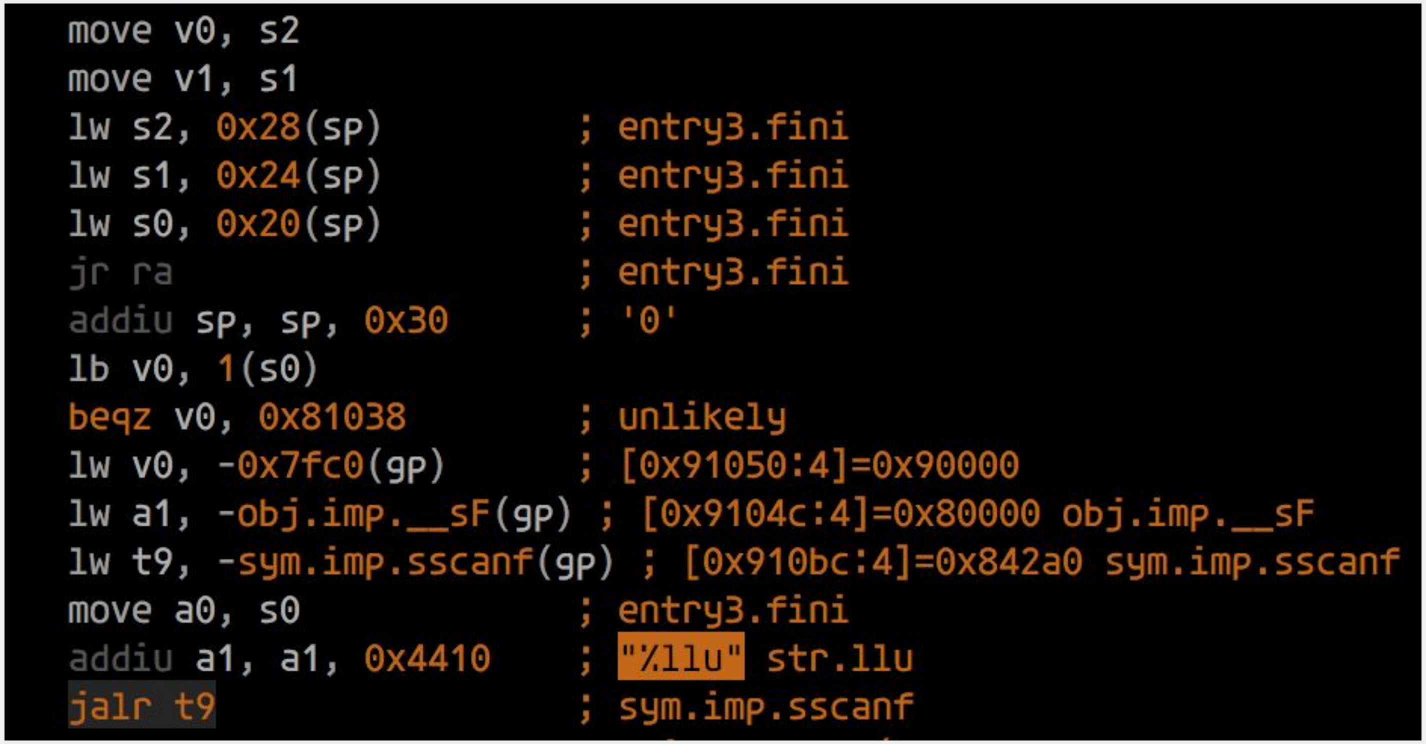
- ADR vs ADRP
- No direct access to PC
- LDR when absolute addresses



```
ldr x0, [x8]
str x28, [sp]
adr x1, str.Cannot_open_script_file___s ; 0x100007ab6 ; "Cannot open script file '%s'\n"
nop
bl sym.imp.fprintf ; "\n"
                ; int fprintf(FILE: unk_size_format, const char * format : 0x100007ab6 = Cannot
b 0x100005a40
```

Relative Code: MIPS

- GP points to a global pointer table
- Symbols loaded from that table + delta



```
move v0, s2
move v1, s1
lw s2, 0x28(sp)         ; entry3.fini
lw s1, 0x24(sp)         ; entry3.fini
lw s0, 0x20(sp)         ; entry3.fini
jr ra                   ; entry3.fini
addiu sp, sp, 0x30      ; '0'
lb v0, 1(s0)
beqz v0, 0x81038        ; unlikely
lw v0, -0x7fc0(gp)      ; [0x91050:4]=0x90000
lw a1, -obj.imp.__sF(gp) ; [0x9104c:4]=0x80000 obj.imp.__sF
lw t9, -sym.imp.sscanf(gp) ; [0x910bc:4]=0x842a0 sym.imp.sscanf
move a0, s0             ; entry3.fini
addiu a1, a1, 0x4410    ; "%llu" str.llu
jalr t9                 ; sym.imp.sscanf
```

Endianness

Endianness

- Intel architectures are little endian.
- Network endian is big
- Most CPUs designed to work on network devices are big endian: mips, sparc, powerpc, arm
- Some of those are big endian.
  This means that the cpu can be configured to work on le or be.
- Most of those instructions on arm are being deprecated and the trend is now little endian only.
- Middle endian architectures
- Little, Big and Middle!

Middle Endian

Which architectures are middle endian?

Tell me at least two.

Middle Endian

Which architectures are middle endian?

- PDP-11
- Clemency (Defcon CTF 2017)

Using rasm2 to assemble and disassemble

- Use -d to disassemble
- Use -a to specify the arch
- Use -L to list all the supported plugins
- Use r2pm to install more
- Use -b to set the bits size, 16, 32, 64..
- Use -e to swap endian
- Use -o to specify the offset

Memory Alignment

- 4, 32 …
- Depends on the granularity of the MMU
- Segmentation
- Page size
- Cache size
- PreCache

Comparing instructions sets

- How to make a system call?
- How to nop
- How to call
- How to make conditions
- Basic registers: pc, sp, bp, a0, a1 ...

Toolchains

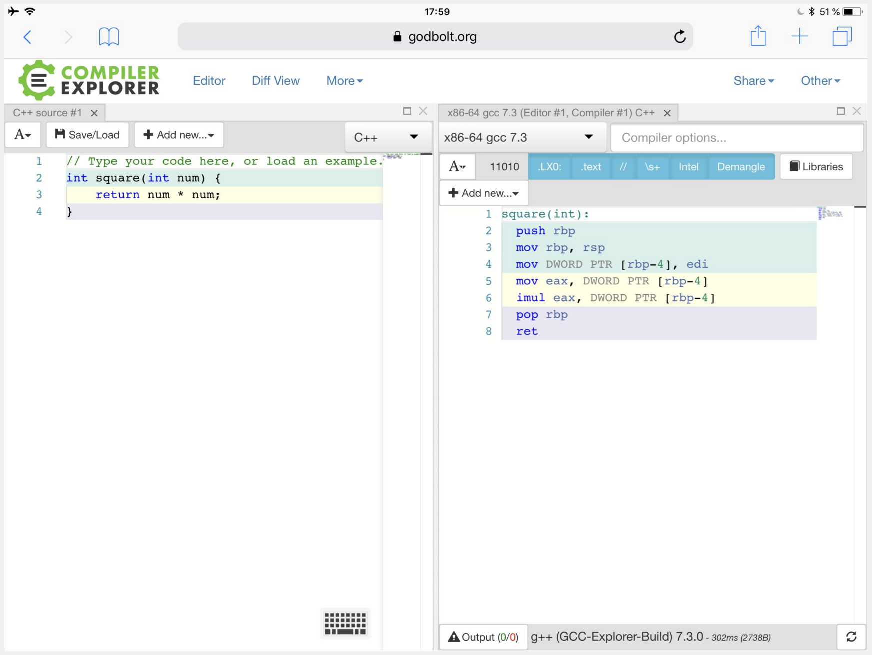
Compilers

How do they generate code?

- See god compiler website
- On mac we can use
  - xcrun —sdk iphoneos gcc -arch arm64
- NDK supports mips, arm, arm64, x86
- Distro facilities
- SDKs of specific product
- Check the radare2-regressions repo for bins

GodBolt

- GodBolt is a website that allow us to compile code using different compilers, languages and target CPUs.
- We can use this to see how the generated code looks.



```cpp
// Type your code here, or load an example.
int square(int num) {
    return num * num;
}
```

```asm
square(int):
  push rbp
  mov rbp, rsp
  mov DWORD PTR [rbp-4], edi
  mov eax, DWORD PTR [rbp-4]
  imul eax, DWORD PTR [rbp-4]
  pop rbp
  ret
```

g++ (GCC-Explorer-Build) 7.3.0 - 302ms (2738B)

Compilers, Languages and Flags

- Compiling a hello world in different languages
  - Go
  - C
  - Dlang
  - Java
- Disassemble the binaries
- Use different compiler flags
  - -O0
  - -O3
  - -Oz

Compilers

- If you use macOS, Install XCode and use xcrun
  - xcrun --sdk iphoneos gcc
    - -arch arm64
    - -arch armv7
    - -arch armv7 -mno-thumb

- This works for C, C++, ObjC and Swift.

CrossCompiling with Go

- go build main.go

```
$ GOOS=linux GOARCH=arm go build hello.go
```

```
package main

func main() {

    println (“hello world”)

}
```

CrossCompiling with Rust

- Rustup toolchain
  - *curl https://sh.rustup.rs -sSf | sh*
- List all available toolchains:
  - rustup target list
  - https://forge.rust-lang.org/platform-support.html
- Install your favourite target and make it the default
  - rustup target install $targetName
- Compile Hello World
  - rustc --target $targetName hello.rs

CrossCompiling with the NDK

- Download the Android NDK use r2’s sys scripts
  - sys/android-shell.sh [arm, arm64, mips, x86]
  - Then you get ndk-gcc in the PATH
- But we can also do the same by hand but you have to
  - Specify the --sysroot
  - Set platform includes with -I
  - Set PATH to get the linker, ar …
- Google use to release buggy NDKs, so get the last one, and rollback to previous if anything’s wrong.

Other Toolchains

Each manufacturer use to provide its own toolchain.

- Same goes for distros, which use to distribute
  - Gentoo’s crossdev
- Search for “arch” or mingw, toolchain …
  - clang, gcc are the main C compilers
- CrossDev

DockCross

- Docker images containing toolchains to cross-compile for several operating systems and architectures.
    - git clone https://github.com/dockcross/dockcross
    - cd dockcross
    - make # generate base docker image

Exercise: cross compiling

- Choose one of those cross-compiling methods
- Experiment with different compilation flags

De-Compilers

Decompilers

The inverse process to compilation.

Takes a binary and generates C-like code to ease the reading

- Usually it doesn’t compile
- Many constructions and optimizations can’t be decoded easily
- Arch/Compiler/Flags/Version-specific

Decompilers

Opensource

- R2dec
- Snowman
- Retdec
- Radeco

Commercial

- Hexrays
- Hopper

DEMO: Using a Decompiler

- Let’s try to decompile some code by hand
  - Take the binary of interest
  - Find the target function
  - Read the disassembly and write the decompiled C-like

- Then do the same with r2dec
  - r2pm -i r2dec

Better disassembly with emulation

- e asm.emu=true
- e asm.emu.str = true (asm.emu.* -> emu.*)
- e asm.pseudo=true
- e??asm.

- pds
- pdc
- ...

Manual Inspection

Inspecting the binaries

- Boilerplate
- How to find main from entrypoint?
- Runtime library
- Statically linked libraries
- Structure of the binary

Tools used in this class

During this course we will be using Radare2 and Frida

- Install r2 from git
- Pip install frida

At least one cross compiler toolchain (rust, ndk, go, xcrun …)

Exercise: Find the Main

- Use r2 enter in Vp and follow the code path starting from the entrypoint.
- Check this on binaries generated by different compilers
- See how strings are processed by the language, how arguments are passed to functions, etc

Comparing Code

- Use different compiler flags and get the disasm
- Write a shell script that uses objdump and diff to get the differences
- Port the script to use r2
- Use radiff2

Reverse Engineering non x86 bins

- How to find xrefs to strings?
- How to find xrefs to functions?
- Callgraph
- Visualizing references

XRefs

- What is an xref?
- How are they constructed?

Intermediate Languages

Intermediate Languages

- Internally each instruction is encoded as a bunch of other microinstructions
- Analysis tools and emulators use intermediate languages to abstract the logic of the arch
- The design of the IR depends on the use cases that is trying to solve.
  - To be transpiled to other languages
  - High or low level
  - Executable code, can be interpreted
  - To be optimizable with multiple passes
  - For decompilation

Microcode

- Intermediate abstraction between assembly code and the hardware.
- Used by the manufacturers to provide CPU updates
- Re-implement, extend instruction set
- Each CPU instruction is implemented in several small ones
- CPUs have its own protection methods to avoid anyone to replace that microcode.
- Some vulnerabilities has been found for some AMD and Intel processors.
- Think of it as a limited FPGA

Intermediate Languages

- REIL
- ESIL (radare2)
- .net MSIL
- LLVM Bitcode
- IDA Microcode
- BN
- Valgrind
-

https://hex-rays.com/products/ida/support/ppt/recon2018.ppt

IR: For compilation

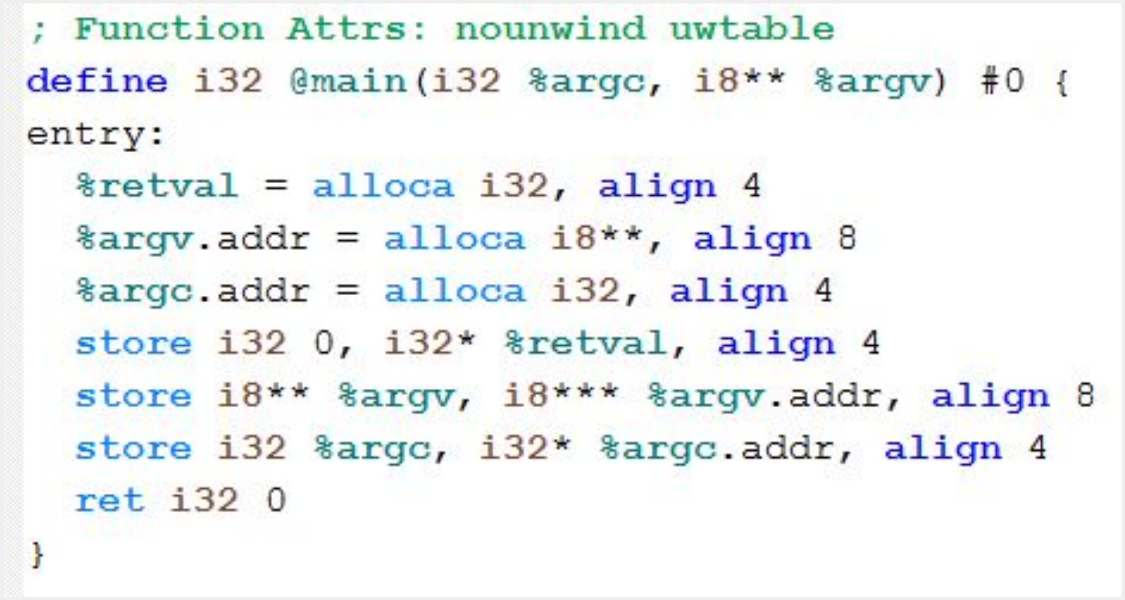
- Those are used internally by the compilers in order to get an arch-independent way to represent the code and write generic optimization and analysis tools on top of it.
- Provide extra information, compiler IRs expose type, size, version, alignment and many other information
- More verbose than pure assembly
- Most known is the one in LLVM, but there are more!

LLVM IR: demo

- Compile ired using clang
- Apple only accepts apps with bitcode (*-fembed-bitcode)*
- This allows them to recompile and optimize apps for new CPUs or compiler versions.

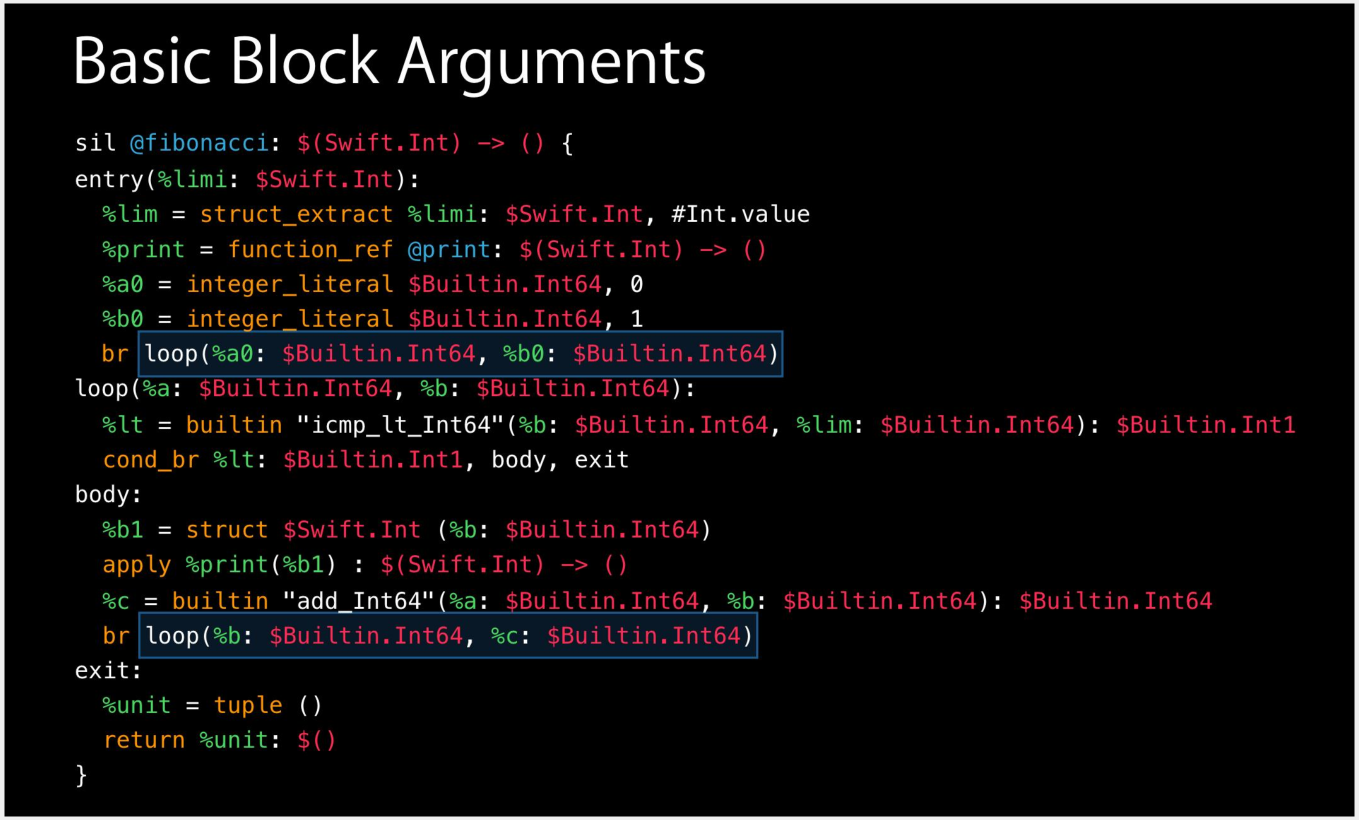
- clang -S -llvm-ir -c ired.c
  - Generate a .ll file
  - If we don’t pass the -S we get a .bc

LLVM IR



```
; Function Attrs: nounwind uwtable
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %retval = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
  %argc.addr = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i8** %argv, i8*** %argv.addr, align 8
  store i32 %argc, i32* %argc.addr, align 4
  ret i32 0
}
```

SIL: Swift Intermediate Language



Basic Block Arguments

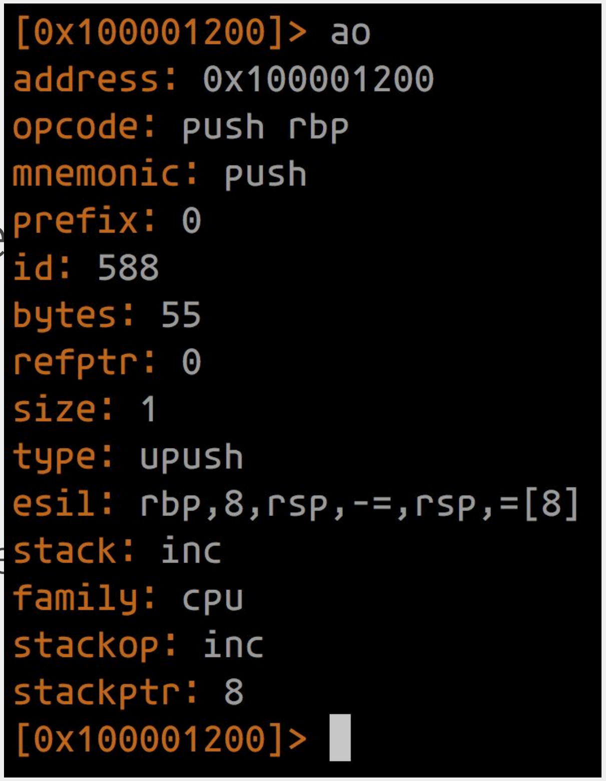
```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Int64, 0
  %b0 = integer_literal $Builtin.Int64, 1
  br loop(%a0: $Builtin.Int64, %b0: $Builtin.Int64)
loop(%a: $Builtin.Int64, %b: $Builtin.Int64):
  %lt = builtin "icmp_lt_Int64"(%b: $Builtin.Int64, %lim: $Builtin.Int64): $Builtin.Int1
  cond_br %lt: $Builtin.Int1, body, exit
body:
  %b1 = struct $Swift.Int (%b: $Builtin.Int64)
  apply %print(%b1) : $(Swift.Int) -> ()
  %c = builtin "add_Int64"(%a: $Builtin.Int64, %b: $Builtin.Int64): $Builtin.Int64
  br loop(%b: $Builtin.Int64, %c: $Builtin.Int64)
exit:
  %unit = tuple ()
  return %unit: $()
}
```

IR: For reversing

- Those intermediate representations are designed to be used for reverse engineering.
- To allow us to emulate code or analyze as an individual or as a block the logic of the code.
- Depending on the level of detail we want the representation can infer variables, types, arguments...
- Use to be simple, Risc style
- Sometimes considered microcode

IR: Registers

- Infinite or large number of virtual registers
- Size of register specified in the operand
- Subregisters prefixed with different name
  - b for byte, w for word, d for dword…
- Some IR append the register size to the register
  - Others (like ESIL) just make this implicit by the name
  - Requires a register profile to work



```
[0x100001200]> ao
address: 0x100001200
opcode: push rbp
mnemonic: push
prefix: 0
id: 588
bytes: 55
refptr: 0
size: 1
type: upush
esil: rbp,8,rsp,-=,rsp,=[8]
stack: inc
family: cpu
stackop: inc
stackptr: 8
[0x100001200]>
```

IR: Memory

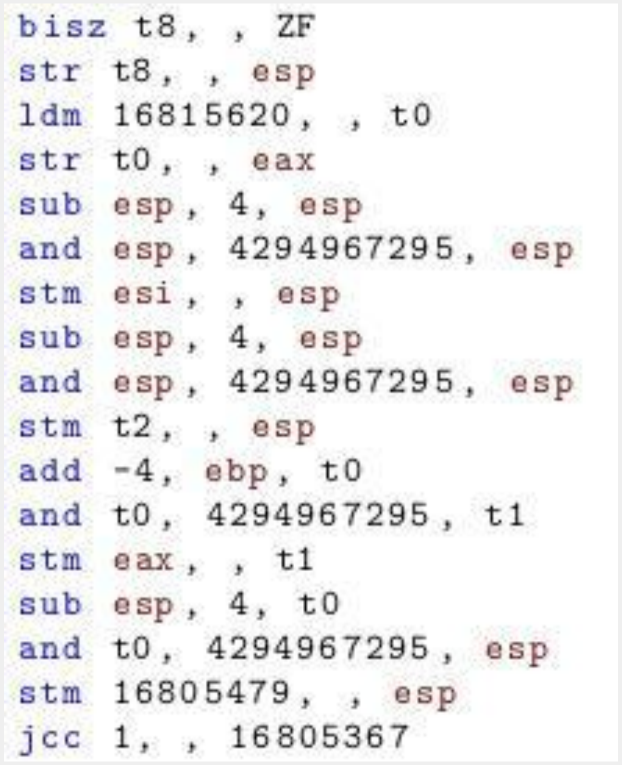
- Selectors (bank switch, segment register...)
- Specify base pointer (for structs…)
- Specify alignment
- Page permissions emulation
- Exception handler

- store %struct.r_num_t* %0, %struct.r_num_t** %4, align 8
- store i8* %1, i8** %5, align 8

REIL

- Used by zynamics bindiff
- To bindiff across archs

- R2 can spit REIL from ESIL
  - See the aetr command



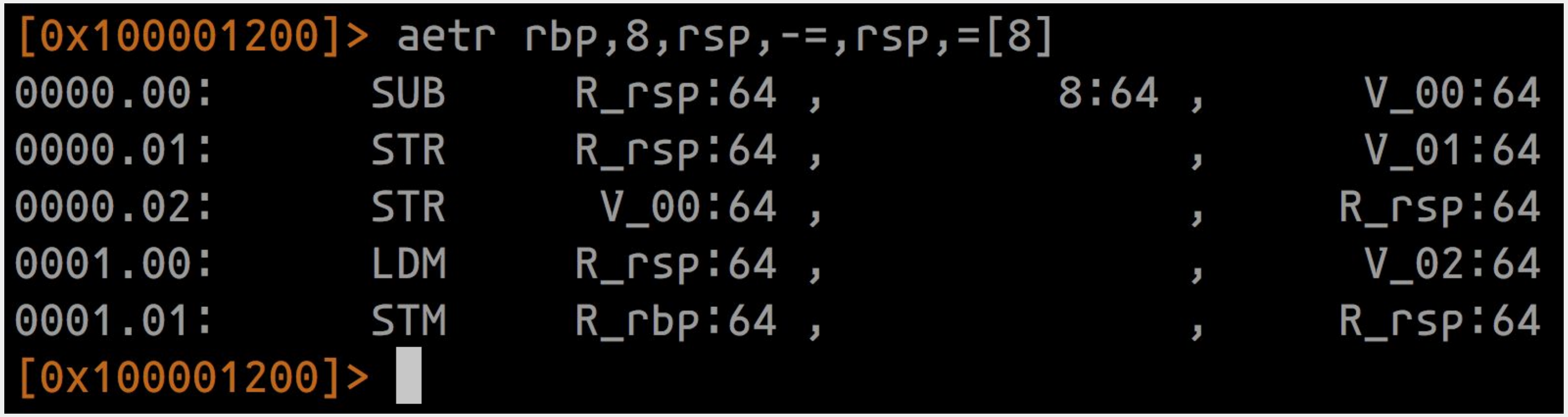
```
bisz t8, , ZF
str t8, , esp
ldm 16815620, , t0
str t0, , eax
sub esp, 4, esp
and esp, 4294967295, esp
stm esi, , esp
sub esp, 4, esp
and esp, 4294967295, esp
stm t2, , esp
add -4, ebp, t0
and t0, 4294967295, t1
stm eax, , t1
sub esp, 4, t0
and t0, 4294967295, esp
stm 16805479, , esp
jcc 1, , 16805367
```

REIL: Links

https://www.zynamics.com/binnavi/manual/html/reil_language.htm

https://moflow.org/ref/REIL%20-%20A%20platform-independent%20intermediate%20representation%20of%20disassembled%20code%20for%20static%20code%20analysis.pdf

https://github.com/Cr4sh/openreil



```
[0x100001200]> aetr rbp,8,rsp,-=,rsp,=[8]
0000.00:      SUB     R_rsp:64 ,            8:64 ,       V_00:64
0000.01:      STR     R_rsp:64 ,                 ,       V_01:64
0000.02:      STR      V_00:64 ,                 ,      R_rsp:64
0001.00:      LDM     R_rsp:64 ,                 ,       V_02:64
0001.01:      STM     R_rbp:64 ,                 ,      R_rsp:64
[0x100001200]> 
```

REIL: Specifications

- Small instruction set (only 17 instructions)
- Very similar to real hardware
- Simple register-based VM with no internal stack
- RISC style, so each instruction makes a clear action
- Most instructions have 3 operands
    - ADD, SUB, MUL, DIV, MOD, BSH
    - AND, OR, XOR
    - LDM, STM, STR
    - BISZ, JCC
    - NOP, UNKN, UNDEF
- Types of operands:
    - Immediates
    - Registers (size specified in the operand itself, unlinked to the real arch)
    - SubAddress, like immediates but hold address of reil instructions
- Call instructions are hinted, because each arch do different things.

REIL: Limitations

- UNDEF instruction undefines the value of a register
  - Useful for emulation, but this should be a hint for the VM not an instruction
- UNKN means the instruction can’t be (or it isn’t) translated to REIL
- Original implementation in Java
- Flat memory model, some MMUs can’t be emulated here
- Doesn’t works with self-modifying code
- Doesn’t works with code that changes endianness
- No FPU, MMX/SSE support
- REIL is closed source, bought by Google, and abandoned
- OpenREIL is the OS reimplementation in C
  - Limited to X86
  - Abandoned (last commit 6 months ago)

SSA

- Static Single Assignment
- SSA is a common assembly language code analysis technique that consists in generating virtual registers everytime a register value is overwritten, making all register contents constant.
- This eases the analysis and allows to identify which are the input parameters of a function.
- Used by many optimization engines

https://en.wikipedia.org/wiki/Static_single_assignment_form

Other ILs

- RREIL - modern and flexible REIL (fpu, mmx, …)
- MAIL - for malware analysis (polymorphic code…)
- VEX - used by valgrind
- IDA Microcode
    - https://recon.cx/2018/brussels/resources/slides/RECON-BRX-2018-Decompiler-internals-microcode.pdf
    - Supports memory selectors
    - Decompiler written on top of it
    - Several optimization passes
    - 72 instructions
    - Added some syntax features on top of it
    - infinite registers and support for micro-registers (aka sub-regs)

ESIL

- Represented as strings
- Comma separated list of tokens strings
- Internally each instruction is encoded as a bunch of other microinstructions
- Each token is pushed in order to the ESIL stack and evaluated if the name is a recognized token.
- Stack based virtual machine similar to FORTH

ESIL

ESIL

- Intermediate Language designed by me
- Expressions are comma separated strings
    - Easy to review, test and maintain
- Similar to Forth
    - Reverse Polish notation
    - Stack based VM
    - Implicit side-effects
    - Low Level
    - program counter disassociated from the VM
        - Allows to emulate self-modifying code
    - Initial support for MMX, FPU is planned
    - Support delayed execution (for MIPS at least)

ESIL

- Register size is specified by the register profile
    - Expressions drop that logic
    - VM is available in C and Rust
    - Hooks to change the behaviour of anything
        - When instruction is fetched
        - When memory is accessed or written
        - When registers are read or modified
        - Exceptions
        - Syscalls / IO / ...
        - When a specific address is going to be executed
            - Emulate function logic
        - Anal plugs can specify custom ops specific for the target

What can we do with it?

- Emulate code with a single virtual machine
- Compare binaries of different architectures
- Use its expressions to match in search operations
- Assisted debugging (guess what next instruction will do before executing it)
- Predefine state
- Unpack encrypted code
-

Demo ESIL in r2

- Emulate some code with radare2
- Visualize the results of each emulated instruction
- Catch new references thanks to the emulation
- Recursive emulation in graph
- Pds
  - aeim
  - e io.cache=true
  - aeip
  - aecu $$
  - ...

Dynamic Analysis

Static vs Dynamic

- There are cases where we will need to emulate or directly execute a program.
- This is somewhat dangerous depending on the target (malware?) so we should be careful and use virtual machines as much as possible.
- There are different tools that can be useful for dynamic analysis.
  - Debuggers (r2, lldb, gdb, …)
  - Tracers (frida)
  - Syscall Tracer (strace)
  - Other: Valgrind, ltrace, dtrace

Process State

Register profiles

- A register profile is a way to describe how registers are mapped inside an arena buffer.
- That’s how kernels and debuggers map process states
    - In r2 use the `drp` command to get that
    - Depends on debugger backend, but r2 have its own for esil
    - Just a TSV file
- GDB also uses register profiles, but in XML
- Describes the following information:
    - Type of register
    - Name of register
    - Offset in buffer
    - Size in buffer
    - Packed/flagstring

Register Types

- GPR
- FPU
- DRX
- MMX/XMM
- SEGMENT
- FLAGS
- PRIVILEGED
- ZERO GROUND

Register Classification by Use

- Passing arguments (A0..3)
- Returning result (A0)
- Hold Stackframe (SP, BP)
- Program Counter (PC)

Those are the most basic types.

Memory Layout

- Programs need memory to be ready to run
- There are different memory layouts that are configured in the MMU of the processor.
- Protected mode isolates process memory from other processes. Kernel can see everyone's memory, but kernel mappings are done on top of the physical, linear ones.
- Stack, Program code, Heap, Libraries, … require
  - Base address + Size
  - Page permissions
  - Page size (4KB, 8KB, 64KB, ..)
  - Associated physical address

Memory Layout with radare2

- r2 provides all the abstractions inside libr/io and the ‘o’ command to emulate an MMU
- Let’s do a practice to map few bins in memory
  - What’s PADDR, VADDR, BADDR, LADDR
  - Define permissions and how this affects r2 emulation
  - Priorities (bankswitching)

Emulators, Virtualizers, Simulators

- There are different tools that we can use to run a program in a sandboxed environment
- It depends on the needs we have and the target architecture to choose the option.

Type of Tools:

- Emulator (interpret or JIT native instructions)
- Tracer (inject code in the app to trace the app flow)
- Simulator (same arch, but run different OS or env)
- Sandbox (restrict execution boundaries)

Core Files

- Core files are (elf, mach0, …)
- Can be generated by the kernel, or by the debugger.
- Usually when the process crashes, but can be generated at any time.
- Useful to load with the debugger and get a snapshot of the state of the process when it died.
    - Memory maps
    - Register state
    - Loaded files
    - ...

Demo: Frida tracing ARM binary

- Start a process in an ARM VM or device (any android)
  - Hook some functions
  - Trace the arguments
  - Inspect memory
- Install and use r2frida

Problem Solvers

Problem Solvers

- Problem solvers are mathematical engines that compute equations in order to find the solution to the missing values in it.
- Require some restrictions, preconditions and postconditions
- Angr, z3, ….

DEMO with r2angr.py

TODO

Questions?