64-BIT ARM ASSEMBLY

From Basics to Party Tricks

BRUH I DO NOT CARE ABOUT ASM.

BRUH I DO NOT CARE ABOUT ASM.

» Nor do I. But I do care about fast code.

BRUH I DO NOT CARE ABOUT ASM.

» Nor do I. But I do care about fast code.

» We will focus on the basics of reading 64 bit ARM assembly, which we can leverage to understand if our compiler is emitting the instructions we want.

WHAT PLATFORM ARE WE LEARNING?

WHAT PLATFORM ARE WE LEARNING?

» Apple Silicon. The code herein will work nowhere else.

WHAT PLATFORM ARE WE LEARNING?

» Apple Silicon. The code herein will work nowhere else.

» But the ideas should transfer to programming Raspberry Pi/NVIDIA Jetson/Snapdragon/many more, with minor syntax changes for your assembler.

ALRIGHT I GUESS WE'RE DOING THIS. WHERE'S THE "HELLO, WORLD"?

ALRIGHT I GUESS WE'RE DOING THIS. WHERE'S THE "HELLO, WORLD"?

» "Hello, world" is too hard.

ALRIGHT I GUESS WE'RE DOING THIS. WHERE'S THE "HELLO, WORLD"?

» "Hello, world" is too hard.

» In assembly, we write exit_code.s, which gives an integer return status.

exit_code.s

```
.global _main
.balign 4
_main:
    mov w0, #7   // move '7' into 32 bit register
    mov w16, #1 // move '1' into 32 bit register-read by OS.
    svc #0x80    // "supervisor call"
```

exit_code.s

```
.global _main
.balign 4
_main:
    mov w0, #7   // move '7' into 32 bit register
    mov w16, #1 // move '1' into 32 bit register-read by OS.
    svc #0x80    // "supervisor call"
```

exit_code.s

```
.global _main
.balign 4
_main:
    mov w0, #7   // move '7' into 32 bit register
    mov w16, #1 // move '1' into 32 bit register-read by OS.
    svc #0x80    // "supervisor call"
```

exit_code.s

```
.global _main
.balign 4
_main:
    mov w0, #7   // move '7' into 32 bit register
    mov w16, #1 // move '1' into 32 bit register-read by OS.
    svc #0x80    // "supervisor call"
```

exit_code.s

```
.global _main
.balign 4
_main:
    mov w0, #7   // move '7' into 32 bit register
    mov w16, #1 // move '1' into 32 bit register-read by OS.
    svc #0x80    // "supervisor call"
```

exit_code.s

```
.global _main
.balign 4
_main:
    mov w0, #7   // move '7' into 32 bit register
    mov w16, #1 // move '1' into 32 bit register-read by OS.
    svc #0x80    // "supervisor call"
```

exit_code.s

```
.global _main
.balign 4
_main:
    mov w0, #7   // move '7' into 32 bit register
    mov w16, #1 // move '1' into 32 bit register-read by OS.
    svc #0x80    // "supervisor call"
```

exit_code.s

```
.global _main
.balign 4
_main:
    mov w0, #7   // move '7' into 32 bit register
    mov w16, #1 // move '1' into 32 bit register-read by OS.
    svc #0x80    // "supervisor call"
```

exit_code.s

```
.global _main
.balign 4
_main:
    mov w0, #7   // move '7' into 32 bit register
    mov w16, #1 // move '1' into 32 bit register-read by OS.
    svc #0x80    // "supervisor call"
```

ASSEMBLE, LINK, RUN:

```
$ as exit_code.s -o exit_code.o
$ ld -o exit_code.x exit_code.o \
    -lSystem -syslibroot `xcrun -sdk macosx --show-sdk-path`
$ ./exit_code.x
$ echo $?
7
```

32 AND 64 BIT REGISTERS

```
.global _main
.balign 4
_main:
    mov x0, #7   // move '7' into 64 bit register x0
    mov x16, #1 // move '1' into 64 bit register x16-read by OS.
    svc #0x80    // "supervisor call"
```

OK, NOW "HELLO WORLD"

```
.global _main
.balign 4
_main:
    mov x0, #1 // stdout = '1'
    # adr = 'Address to Register'
    adr x1, hello_world
    mov x2, 14  // 14 characters in our string
    mov x16, #4 // 4 is write syscall
    svc #0x80 ; "supervisor call"

    // Now exit:
    mov x0, #0
    mov x16, #1  // 1 = terminate syscall
    svc #0x80

hello_world: .ascii "hello, world!\n"
```

OK, NOW "HELLO WORLD"

```
.global _main
.balign 4
_main:
    mov x0, #1 // stdout = '1'
    # adr = 'Address to Register'
    adr x1, hello_world
    mov x2, 14  // 14 characters in our string
    mov x16, #4 // 4 is write syscall
    svc #0x80 ; "supervisor call"

    // Now exit:
    mov x0, #0
    mov x16, #1  // 1 = terminate syscall
    svc #0x80

hello_world: .ascii "hello, world!\n"
```

OK, NOW "HELLO WORLD"

```
.global _main
.balign 4
_main:
    mov x0, #1 // stdout = '1'
    # adr = 'Address to Register'
    adr x1, hello_world
    mov x2, 14  // 14 characters in our string
    mov x16, #4 // 4 is write syscall
    svc #0x80 ; "supervisor call"

    // Now exit:
    mov x0, #0
    mov x16, #1  // 1 = terminate syscall
    svc #0x80

hello_world: .ascii "hello, world!\n"
```

OK, NOW "HELLO WORLD"

```
.global _main
.balign 4
_main:
    mov x0, #1 // stdout = '1'
    # adr = 'Address to Register'
    adr x1, hello_world
    mov x2, 14  // 14 characters in our string
    mov x16, #4 // 4 is write syscall
    svc #0x80 ; "supervisor call"

    // Now exit:
    mov x0, #0
    mov x16, #1  // 1 = terminate syscall
    svc #0x80

hello_world: .ascii "hello, world!\n"
```

OK, NOW "HELLO WORLD"

```
.global _main
.balign 4
_main:
    mov x0, #1 // stdout = '1'
    # adr = 'Address to Register'
    adr x1, hello_world
    mov x2, 14  // 14 characters in our string
    mov x16, #4 // 4 is write syscall
    svc #0x80 ; "supervisor call"

    // Now exit:
    mov x0, #0
    mov x16, #1  // 1 = terminate syscall
    svc #0x80

hello_world: .ascii "hello, world!\n"
```

printf debugging is difficult in assembly

printf debugging is difficult in assembly

» So assemble with -g and use lldb

```
➜  example_code git:(master) ✗
➜  example_code git:(master) ✗
➜  example_code git:(master) ✗ ll
```

INTEGER ADDITION IN ARM ASSEMBLY . . . IS EASY?

```
.global _main
.balign 4
_main:
    mov x0, #7      // C++: long x0 = 7;
    add x0, x0, #8 // C++: x0 += 8;
    mov x16, #1 // move '1' into 64 bit register x16-read by OS.
    svc #0x80    // "supervisor call"
```

INTEGER ADDITION IN ARM ASSEMBLY . . . IS EASY?

```
.global _main
.balign 4
_main:
    mov x0, #7      // C++: long x0 = 7;
    add x0, x0, #8 // C++: x0 += 8;
    mov x16, #1 // move '1' into 64 bit register x16-read by OS.
    svc #0x80    // "supervisor call"
```

SUBTRACTION IS EASY TOO?

```
.global _main
.balign 4
_main:
    mov x1, #7          // C++: long x1 = 7;
    mov x2, #9          // C++: long x2 = 9;
    subs x0, x2, x1 // C++: long x0 = x2 - x1;
    mov x16, #1 // '1' = terminate syscall
    svc #0x80    // "supervisor call"
```

ONTO FLOATING POINT

```
.global _main
.balign 4
_main:
    fmov d1, #7.5    // double d1 = 7.5;
    fmov d2, #8.5    // double d2 = 8.5;
    fadd d0, d1, d2 // double d0 = d1 + d2;
    fcvtzs x0, d0
    mov x16, #1 // '1' = terminate syscall
    svc #0x80    // "supervisor call"
```

ONTO FLOATING POINT

```
.global _main
.balign 4
_main:
    fmov d1, #7.5    // double d1 = 7.5;
    fmov d2, #8.5    // double d2 = 8.5;
    fadd d0, d1, d2 // double d0 = d1 + d2;
    fcvtzs x0, d0
    mov x16, #1 // '1' = terminate syscall
    svc #0x80    // "supervisor call"
```

STRUCTURE OF AN AARCH64 INSTRUCTION

Studying how instructions are encoded gives us lots of insight into how ARM works.

Study the machine code with llvm-objdump

```
$ llvm-objdump -d exit_code.o
        0: d28000e0      mov x0, #7
        4: d2800030      mov x16, #1
        8: d4001001      svc #0x80
```

Study the machine code with llvm-objdump

```
$ llvm-objdump -d exit_code.o
       0: d28000e0      mov x0, #7
       4: d2800030      mov x16, #1
       8: d4001001      svc #0x80
```

» Each instruction is 4 bytes. This is general.

Naively, we might think we could encode ~4B instructions in 32 bits.

Naively, we might think we could encode ~4B instructions in 32 bits.

» But where is the '7' we return stored? It's not in a .data segment.

Naively, we might think we could encode ~4B instructions in 32 bits.

» But where is the '7' we return stored? It's not in a .data segment.

» Let's disassemble a different program to understand this further...

```
34: d2800000 mov x0, #0
38: d2800020 mov x0, #1
3c: d2800040 mov x0, #2
40: d2800060 mov x0, #3
44: d2800080 mov x0, #4
48: d28000a0 mov x0, #5
4c: d28000c0 mov x0, #6
50: d28000e0 mov x0, #7
54: d2800100 mov x0, #8
58: d2800200 mov x0, #16
5c: d2800400 mov x0, #32
```

```
34: d2800000 mov x0, #0
38: d2800020 mov x0, #1
3c: d2800040 mov x0, #2
40: d2800060 mov x0, #3
44: d2800080 mov x0, #4
48: d28000a0 mov x0, #5
4c: d28000c0 mov x0, #6
50: d28000e0 mov x0, #7
54: d2800100 mov x0, #8
58: d2800200 mov x0, #16
5c: d2800400 mov x0, #32
```

» The value is stored in the instruction!

The source and destination registers must also be stored in the instruction:

```
bc: 2a0003e0 mov w0, w0
c0: 2a0103e0 mov w0, w1
c4: 2a0203e0 mov w0, w2
c8: 2a0303e0 mov w0, w3
cc: 2a0403e0 mov w0, w4
d0: 2a0503e0 mov w0, w5
d4: 2a0603e0 mov w0, w6
d8: 2a0703e0 mov w0, w7
dc: 2a0803e0 mov w0, w8
e0: 2a0903e0 mov w0, w9
```

The source and destination registers must also be stored in the instruction:

```
 e4: 2a0003e1 mov w1, w0
 e8: 2a0003e2 mov w2, w0
 ec: 2a0003e3 mov w3, w0
 f0: 2a0003e4 mov w4, w0
 f4: 2a0003e5 mov w5, w0
 f8: 2a0003e6 mov w6, w0
 fc: 2a0003e7 mov w7, w0
100: 2a0003e8 mov w8, w0
104: 2a0003e9 mov w9, w0
```

Aside: Bit patterns for AArch64 can be found here

Representative example of an AArch64 encoding:

```
x00x 0010 1xxi iiii iiii iiii iiid dddd  -  movn Rd HALF
x10x 0010 1xxi iiii iiii iiii iiid dddd  -  movz Rd HALF
```

Representative example of an AArch64 encoding:

```
x00x 0010 1xxi iiii iiii iiii iiid dddd  -  movn Rd HALF
x10x 0010 1xxi iiii iiii iiii iiid dddd  -  movz Rd HALF
```

» In this case, only 6 bits are used for the opcode!

POP QUIZ

AArch64 provides 32 integer registers x0-x31.

Suppose the ARM designers decided 64 integer registers x0-x63 would be better.

What are the implications?

RISC Philosophy: Register-register moves

Consider these two instructions:

```
.global _main
.balign 4
_main:
    mov w0, w1  // register-register move
    orr w0, wzr, w1 // Logical or with zero register: w0 = w1|0
    mov x0, #0  // exit code = 7
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"
```

Let's disassemble this . . .

```
$ llvm-objdump -d instruction_aliases.o
        0: 2a0103e0      mov w0, w1
        4: 2a0103e0      mov w0, w1
        ...
```

wut??? where did the orr instruction go?

And why are the opcodes identical???

ASSEMBLY IS TOO HIGH-LEVEL

```
$ llvm-objdump -M no-aliases -d instruction_aliases.o
       0: 2a0103e0      orr w0, wzr, w1
       4: 2a0103e0      orr w0, wzr, w1
       ...
```

INSTRUCTION ALIASES

A

```
mov w0, w1
```

is an alias for

```
orr w0, wzr, w1
```

INSTRUCTION ALIASES

A

```
mov w0, w1
```

is an alias for

```
orr w0, wzr, w1
```

» A good way to keep the "R" in "RISC" is to reuse the same instructions!

ARM leverages the zero register and a logical OR to implement register-register moves.

ARM leverages the zero register and a logical OR to implement register-register moves.

» This frees up opcodes for stuff that's more fun and reduces silicon design burden!

Custom instructions

Custom instructions

» Learning about the custom instructions relevant to your field can have an extremely high ROI.

Our first party trick!

Our first party trick!

» Computing $1/\sqrt{2} \approx 0.7071067811865475$ with the legendary rsqrt trick from Quake:
$y = 1/\sqrt{x} \iff y^2 = 1/x \iff f(y) := 1/y^2 - x = 0$

Our first party trick!

» Computing $1/\sqrt{2} \approx 0.7071067811865475$ with the legendary rsqrt trick from Quake:

$$y = 1/\sqrt{x} \iff y^2 = 1/x \iff f(y) := 1/y^2 - x = 0$$

» Apply Newton's method to $f(y)$ and voila:

$$y_{n+1} := y_n \frac{(3 - x y_n^2)}{2}$$

Enter frsqrts

Enter frsqrts

» ARM provides the frsqrts instruction for computing these Newton iterates.

Enter frsqrts

» ARM provides the frsqrts instruction for computing these Newton iterates.

» But it's strange: Instead of computing $y_n(3 - xy_n^2)/2$, it computes $(3 - xz)/2$, and you need to assemble the rest yourself.

```
.global _main
.balign 4
_main:
    fmov d0, #2.0 // Goal: Compute 1/sqrt(x) for x=2
    fmov d1, #0.5 // initial guess: y0

    fmul d2, d1, d1    // square y0; z:=y0²
    frsqrts d3, d0, d2 // (3-xz)/2
    fmul d1, d1, d3    // y1=y0(3-xz)/2

    // Start the second iteration:
    fmul d2, d1, d1 // second guess squared
    frsqrts d3, d0, d2
    fmul d1, d1, d3

    // Start the third iteration:
    fmul d2, d1, d1 // third guess squared
    frsqrts d3, d0, d2
    fmul d1, d1, d3


    mov x0, 0
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"
```

```
.global _main
.balign 4
_main:
    fmov d0, #2.0 // Goal: Compute 1/sqrt(x) for x=2
    fmov d1, #0.5 // initial guess: y0

    fmul d2, d1, d1    // square y0; z:=y0²
    frsqrts d3, d0, d2 // (3-xz)/2
    fmul d1, d1, d3    // y1=y0(3-xz)/2

    // Start the second iteration:
    fmul d2, d1, d1 // second guess squared
    frsqrts d3, d0, d2
    fmul d1, d1, d3

    // Start the third iteration:
    fmul d2, d1, d1 // third guess squared
    frsqrts d3, d0, d2
    fmul d1, d1, d3


    mov x0, 0
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"
```

```
.global _main
.balign 4
_main:
    fmov d0, #2.0 // Goal: Compute 1/sqrt(x) for x=2
    fmov d1, #0.5 // initial guess: y0

    fmul d2, d1, d1    // square y0; z:=y0²
    frsqrts d3, d0, d2 // (3-xz)/2
    fmul d1, d1, d3    // y1=y0(3-xz)/2

    // Start the second iteration:
    fmul d2, d1, d1 // second guess squared
    frsqrts d3, d0, d2
    fmul d1, d1, d3

    // Start the third iteration:
    fmul d2, d1, d1 // third guess squared
    frsqrts d3, d0, d2
    fmul d1, d1, d3


    mov x0, 0
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"
```

```
.global _main
.balign 4
_main:
    fmov d0, #2.0 // Goal: Compute 1/sqrt(x) for x=2
    fmov d1, #0.5 // initial guess: y0

    fmul d2, d1, d1    // square y0; z:=y0²
    frsqrts d3, d0, d2 // (3-xz)/2
    fmul d1, d1, d3    // y1=y0(3-xz)/2

    // Start the second iteration:
    fmul d2, d1, d1 // second guess squared
    frsqrts d3, d0, d2
    fmul d1, d1, d3

    // Start the third iteration:
    fmul d2, d1, d1 // third guess squared
    frsqrts d3, d0, d2
    fmul d1, d1, d3


    mov x0, 0
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"
```

```
.global _main
.balign 4
_main:
    fmov d0, #2.0 // Goal: Compute 1/sqrt(x) for x=2
    fmov d1, #0.5 // initial guess: y0

    fmul d2, d1, d1    // square y0; z:=y0²
    frsqrts d3, d0, d2 // (3-xz)/2
    fmul d1, d1, d3    // y1=y0(3-xz)/2

    // Start the second iteration:
    fmul d2, d1, d1 // second guess squared
    frsqrts d3, d0, d2
    fmul d1, d1, d3

    // Start the third iteration:
    fmul d2, d1, d1 // third guess squared
    frsqrts d3, d0, d2
    fmul d1, d1, d3


    mov x0, 0
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"
```

Let's use a for-loop to remove the duplication:

```
.global _main
.balign 4
_main:
    fmov d0, #2.0 // Goal: Compute 1/sqrt(2)
    fmov d1, #0.5 // y0=0.5

    mov x1, #0 // iteration count
iteration:
    fmul d2, d1, d1 // square current estimate; z=yn²
    frsqrts d3, d0, d2 // (3-xz)/2
    fmul d1, d1, d3 // yn+1 = yn(3-xz)/2
    add x1, x1, 1
    cmp x1, #5
    b.ne iteration

    mov x0, 0
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"
```

Let's use a for-loop to remove the duplication:

```
.global _main
.balign 4
_main:
    fmov d0, #2.0 // Goal: Compute 1/sqrt(2)
    fmov d1, #0.5 // y_0=0.5

    mov x1, #0 // iteration count
iteration:
    fmul d2, d1, d1 // square current estimate; z=y_n^2
    frsqrts d3, d0, d2 // (3-xz)/2
    fmul d1, d1, d3 // y_n+1 = y_n(3-xz)/2
    add x1, x1, 1
    cmp x1, #5
    b.ne iteration

    mov x0, 0
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"
```

Let's use a for-loop to remove the duplication:

```
.global _main
.balign 4
_main:
    fmov d0, #2.0 // Goal: Compute 1/sqrt(2)
    fmov d1, #0.5 // y_0=0.5

    mov x1, #0 // iteration count
iteration:
    fmul d2, d1, d1 // square current estimate; z=y_n^2
    frsqrts d3, d0, d2 // (3-xz)/2
    fmul d1, d1, d3 // y_n+1 = y_n(3-xz)/2
    add x1, x1, 1
    cmp x1, #5
    b.ne iteration

    mov x0, 0
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"
```

Let's make it a function!

Let's make it a function!

» First we must study the ARM procedure call standard

Let's make it a function!

» First we must study the ARM procedure call standard

» Registers x9-x15 are "corruptible"-a function can change them to whatever without having to restore them afterwards.

Let's make it a function!

» First we must study the ARM procedure call standard

» Registers x9-x15 are "corruptible"-a function can change them to whatever without having to restore them afterwards.

» Registers d0-d7 are "parameter and results registers", so can the callee change those.

```
.global rsqrt

rsqrt:
    fmov d1, #0.5 // y₀=0.5, corrupts d1
    mov x9, #0    // x9 is a 'corruptible register'
rsqrt_iterate:
    fmul d2, d1, d1    // corrupts d2
    frsqrts d3, d0, d2 // corrupts d3
    fmul d1, d1, d3    // yₙ₊₁ = yₙ(3-xz)/2
    add x9, x9, 1
    cmp x9, #5
    b.ne rsqrt_iterate // branch to rsqrt_iterate

    fmov d0, d1 // return value in d0
    ret

_main:
    // Calling convention: First float arg to function is passed in d0:
    fmov d0, #2.0
    bl rsqrt // 'bl' = branch and update link record-tells `ret` where to go.
    mov x0, 0
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"
```

```
.global rsqrt

rsqrt:
    fmov d1, #0.5 // y₀=0.5, corrupts d1
    mov x9, #0    // x9 is a 'corruptible register'
rsqrt_iterate:
    fmul d2, d1, d1    // corrupts d2
    frsqrts d3, d0, d2 // corrupts d3
    fmul d1, d1, d3    // yₙ₊₁ = yₙ(3-xz)/2
    add x9, x9, 1
    cmp x9, #5
    b.ne rsqrt_iterate // branch to rsqrt_iterate

    fmov d0, d1 // return value in d0
    ret

_main:
    // Calling convention: First float arg to function is passed in d0:
    fmov d0, #2.0
    bl rsqrt // 'bl' = branch and update link record-tells `ret` where to go.
    mov x0, 0
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"
```

```
.global rsqrt

rsqrt:
    fmov d1, #0.5 // y₀=0.5, corrupts d1
    mov x9, #0    // x9 is a 'corruptible register'
rsqrt_iterate:
    fmul d2, d1, d1    // corrupts d2
    frsqrts d3, d0, d2 // corrupts d3
    fmul d1, d1, d3    // yₙ₊₁ = yₙ(3-xz)/2
    add x9, x9, 1
    cmp x9, #5
    b.ne rsqrt_iterate // branch to rsqrt_iterate

    fmov d0, d1 // return value in d0
    ret

_main:
    // Calling convention: First float arg to function is passed in d0:
    fmov d0, #2.0
    bl rsqrt // 'bl' = branch and update link record-tells `ret` where to go.
    mov x0, 0
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"
```

How does the function know where to return to? We just typed ret and it "did the right thing".

How does the function know where to return to? We just typed ret and it "did the right thing".

» A bl instruction "branches and updates the link record to PC+4"-i.e., it puts the address of the next instruction in x30.

How does the function know where to return to? We just typed ret and it "did the right thing".

» A `bl` instruction "branches and updates the link record to PC+4"-i.e., it puts the address of the next instruction in x30.

» ret without arguments is implicitly ret x30.

How does the function know where to return to? We just typed ret and it "did the right thing".

» A bl instruction "branches and updates the link record to PC+4"-i.e., it puts the address of the next instruction in x30.

» ret without arguments is implicitly ret x30.

» If you're crazy, you can return to the value in any register, e.g. ret x5. (It'll segfault.)

POP QUIZ: WHY IS THIS AN INFINITE LOOP?

```
.global foo

foo:
    ret

.global bar

bar:
    bl foo
    nop
    ret
```

```
.global _main
.balign 4
_main:
    bl bar

    mov x0, 0
    mov x16, #1
    svc #0x80
```

The value in the procedure link register gets clobbered!

Need to push procedure link register onto the stack or move it into callee saved register before the call.

Fix 1: Callee saved register

```
.global foo

foo:
    ret

.global bar

bar:
    mov x24, x30 // x24-x28 are "callee saved registers"
    bl foo
    ret x24

.global _main
.balign 4
_main:
    bl bar

    mov x0, 0
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"
```

Wait, isn't bar being called by main?

So shouldn't it save x24?

Fix 2 (better): Push the link record to the stack.

```
.global foo

foo:
    nop
    ret

.global bar

bar:
    str x30, [sp, #-16]!
    bl foo
    ldr x30, [sp], #16
    ret

.global _main
.balign 4
_main:
    nop
    bl bar

    mov x0, 0
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"
```

Wait, why did you adjust the stack by 16 bytes but only push an 8 byte register?

Wait, why did you adjust the stack by 16 bytes but only push an 8 byte register?

» In AArch64, the stack must be 16 byte aligned. The assembler won't reject 8 byte stack manipulations, but you'll terminate with a bus error.

Wait, why did you adjust the stack by 16 bytes but only push an 8 byte register?

» In AArch64, the stack must be 16 byte aligned. The assembler won't reject 8 byte stack manipulations, but you'll terminate with a bus error.

» Better question: Why did ARM insist on 16 byte stack alignment? Turns out not even they know.

If you have to adjust the stack by 16 bytes per op, might as well store 16 bytes.

So often you'll see

```
stp x0, x1, [sp, #-16]!
bl do_something
ldp x0, x1, [sp], #16
```

MORE MYSTERIES WITH `-M no-aliases`

Disassembly of

```
    add x9, x9, 1
    cmp x9, #5
    b.ne rsqrt_iterate // branch to rsqrt_iterate
```

gives

```
14: 91000529    add x9, x9, #1
18: f100153f    subs    xzr, x9, #5
1c: 54ffff61    b.ne    0x8 <rsqrt_iterate>
```

Why????

ARM has strong RISC heritage!

ARM has strong RISC heritage!

» cmp x9, #5 is an alias for subs xzr, x9, #5; this saves chip area and a codepoint.

ARM has strong RISC heritage!

» cmp x9, #5 is an alias for subs xzr, x9, #5; this saves chip area and a codepoint.

» Instructions ending in s generally mean "set flags"

If cmp is a subtraction anyway, we can refactor to save an instruction:

```
    mov x9, #3 // put final iteration count into register
rsqrt_iterate:
    fmul d2, d1, d1
    frsqrts d3, d0, d2
    fmul d1, d1, d3
    subs x9, x9, 1  // count downward
    b.ne rsqrt_iterate
```

If cmp is a subtraction anyway, we can refactor to save an instruction:

```
    mov x9, #3 // put final iteration count into register
rsqrt_iterate:
    fmul d2, d1, d1
    frsqrts d3, d0, d2
    fmul d1, d1, d3
    subs x9, x9, 1  // count downward
    b.ne rsqrt_iterate
```

READING FLAGS

Conditional branching instructions read bits set by the previous instruction in the current program status register.

READING FLAGS

Conditional branching instructions read bits set by the previous instruction in the current program status register.

» It can be read in lldb with reg read cpsr.[1]

[1] Technically the CPSR is replaced by PSTATE in AArch64-lldb hasn't yet gotten the memo.

```
➜  example_code git:(master) ✗
```

Our previous rsqrt was didactic. Let's do a good one.

FINAL RSQRT

Uses only 3 registers and 5 instructions!

```
.global _rsqrt
.balign 4
_rsqrt:
    frsqrte d1, d0 // "floating point rsqrt estimate"
    fmul d2, d1, d1 // square current estimate; z=y_n^2
    frsqrts d0, d0, d2 // (3-xz)/2
    fmul d0, d1, d0 // y_n+1 = y_n(3-xz)/2
    ret
```

"FEWER INSTRUCTIONS" != "BETTER PERFORMANCE"

Instructions can have different latencies.

Instruction latencies and throughputs can be found here, these do seem to indicate that this is a very fast rsqrt.

Check out rsqrt_perf.s

Demonstrates that the naive

```
fmov d0, 2.0
fsqrt d0, d0
fmov d1, #1.0
fdiv d0, d1, d0
```

takes about ~0.9ns on M2.

Check out rsqrt_perf.s

Demonstrates that the naive

```
fmov d0, 2.0
fsqrt d0, d0
fmov d1, #1.0
fdiv d0, d1, d0
```

takes about ~0.9ns on M2.

» our rsqrt takes . . . 293ps.

INCORPORATE IN C++

```
#include <iostream>

extern "C" {
    double rsqrt(double);
}

int main() {
    double rsqrt2_est = rsqrt(2.0);
    double rsqrt2_acc = 1/std::sqrt(2.0);
    std::cout << "rsqrt(2) (estimate) = " << rsqrt2_est << "\n";
    std::cout << "rsqrt(2) (cmath   ) = " << rsqrt2_acc << "\n";
}
```

EXERCISE

Write a reciprocal_estimate function using `frecpe` and `frecps`.

I'm not a graphics programmer; don't care about rsqrt

Fair enough. But there are other interesting hardware instructions; try this:

```
$ sysctl -a hw.optional
```

CPU feature flags are documented here.

Are you into crypto?

```
hw.optional.arm.FEAT_SHA256: 1
hw.optional.arm.FEAT_SHA512: 1
hw.optional.arm.FEAT_SHA1: 1
hw.optional.arm.FEAT_AES: 1
hw.optional.arm.FEAT_PMULL: 1
hw.optional.arm.FEAT_RNG: 0
```

Cybersecurity?

```
hw.optional.arm.FEAT_PAuth: 1
hw.optional.arm.FEAT_SB: 1
hw.optional.arm.FEAT_SPECRES: 1
```

Javascript?

hw.optional.FEAT_JSCVT: 1

ML?

```
hw.optional.arm.FEAT_FHM: 1     # half precision support
hw.optional.arm.FEAT_DotProd: 1 # hardware dot products
hw.optional.arm.FEAT_F64MM: 0   # matrix-matrix multiplication
```

READING CONTIGUOUS MEMORY

```
.global l2_norm

l2_norm:
    fmov d0, #0.0
    cbz x1, .bye // compare and branch on zero.
.loop:
    // post-indexed addressing: value at x0 loaded into d1, then x0 inc'd by 8
    ldr d1, [x0], #8
    // Fused-multiply add:
    fmadd d0, d1, d1, d0
    subs x1, x1, 1
    b.ne .loop
    fsqrt d0, d0
.bye:
    ret
```

```
.global _main
.balign 4
_main:
    adrp x0, u@page // address relative to page
    add x0, x0, u@pageoff // add page offset
    mov x1, 3 // length of vector
    bl l2_norm
    fcvtzs x0, d0 // round to integer, towards zero
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"

.text

u: .double 3.3, 4.7, 5.1
```

```
.global _main
.balign 4
_main:
    adrp x0, u@page // address relative to page
    add x0, x0, u@pageoff // add page offset
    mov x1, 3 // length of vector
    bl l2_norm
    fcvtzs x0, d0 // round to integer, towards zero
    mov x16, #1 // '1' = terminate syscall
    svc #0x80   // "supervisor call"

.text

u: .double 3.3, 4.7, 5.1
```

POP QUIZ

If "adrp x0, u@page" loads the address of u "relative to a page"-or 4096 bytes-what should the 12 lowest bits of x0 be after this instruction executes?

```
lldb l2_norm.x
(lldb) b main
Breakpoint 1: where = l2_norm.x`main + 4, address = 0x0000000100003f84
(lldb) run
   19   _main:
   20       adrp x0, u@page // address relative to page
-> 21       add x0, x0, u@pageoff // add page offset
   22       mov x1, 3 // length of vector
   23       bl l2_norm
   24       fcvtzs x0, d0 // round to integer, towards zero
(lldb) p/x $x0
(unsigned long) $0 = 0x0000000100003000
(lldb) ni
   19   _main:
   20       adrp x0, u@page // address relative to page
   21       add x0, x0, u@pageoff // add page offset
-> 22       mov x1, 3 // length of vector
   23       bl l2_norm
   24       fcvtzs x0, d0 // round to integer, towards zero
   25       mov x16, #1 // '1' = terminate syscall
(lldb) p/x $x0
(unsigned long) $1 = 0x0000000100003f9c
```

ARM NEON VECTORIZED DOT PRODUCT

```
.global neon_dot

neon_dot:
    movi v0.4s, #0
    cbz x2, .bye // compare and branch on zero.
.loop:
    ldr q1, [x0], #(4*4) // x0 contains address of first vector
    ldr q2, [x1], #(4*4) // x1 contains address of second vector
    // Fused-multiply add elementwise:
    fmla v0.4s, v1.4s, v2.4s
    subs x2, x2, 4
    b.ne .loop
    faddp v0.4s, v0.4s, v0.4s
    faddp v0.2s, v0.2s, v0.2s
.bye:
    ret
```

RETURNING TO A MYSTERY

RETURNING TO A MYSTERY

» Why do all our functions have .balign 4 on them???

RETURNING TO A MYSTERY

» Why do all our functions have .balign 4 on them???

» Let's investigate.

```
.text
b: .byte 0x7

.global _main
_main:
    mov x0, 0
    mov x16, 1
    svc 0x80
```

```
.text
b: .byte 0x7

.global _main
_main:
    mov x0, 0
    mov x16, 1
    svc 0x80
```

» We have a single byte of data before _main. Where is it stored?

```
$ llvm-objdump -d align_me.o
0000000000000000 <ltmp0>:
       0: 80000007      <unknown>

0000000000000001 <_main>:
       1: d2800000      mov x0, #0x0
       5: d2800030      mov x16, #0x1
       9: d4001001      svc #0x80
```

```
$ llvm-objdump -d align_me.o
0000000000000000 <ltmp0>:
       0: 80000007      <unknown>

0000000000000001 <_main>:
       1: d2800000      mov x0, #0x0
       5: d2800030      mov x16, #0x1
       9: d4001001      svc #0x80
```

» The instructions start at byte 1, the data at byte 0!

What if we request 4 byte alignment?

```
.text
b: .byte 0x7

.global _main
.balign 4
_main:
    mov x0, 0
    mov x16, 1
    svc 0x80
```

```
$ llvm-objdump -d align_me.o
0000000000000000 <ltmp0>:
       0: 00000007      udf #0x7

0000000000000004 <_main>:
       4: d2800000      mov x0, #0x0
       8: d2800030      mov x16, #0x1
       c: d4001001      svc #0x80
```

THE UDF INSTRUCTION

udf := undefined. All instructions beginning with 0000 are undefined; the low 16 bits can be used for data.

Why do we care about unaligned access?

“... if a datum is not aligned to its alignment requirement, memory access can take extra cycles because the datum spans more words than necessary. This penalty is avoided by aligning data sufficiently.”

fuz

“A64 instructions must be word-aligned. Attempting to fetch an instruction from a misaligned location results in a PC alignment fault.”

ARM Architecture reference guide

POP QUIZ: WHAT DOES THIS RETURN?

```
struct Foo {
    char c;
    float d;
    char e;
};

int main() {
    return sizeof(Foo);
}
```

POP QUIZ: WHAT DOES THIS RETURN?

```
struct Foo {
    float d;
    char c;
    char e;
};

int main() {
    return sizeof(Foo);
}
```

godbolt

We can generate packed structs:

```
struct __attribute__((__packed__)) Foo {
    float d;
    char c;
    char e;
};

int main() {
    return sizeof(Foo);
}
```

to get this down to the minimum of 6 bytes.

References:

References:

» ARM Developer Guide

References:

» ARM Developer Guide

» Programming with 64-Bit ARM Assembly Language

References:

» ARM Developer Guide

» Programming with 64-Bit ARM Assembly Language

» An overview of the ARM Assembly Instruction Set

References:

» ARM Developer Guide

» Programming with 64-Bit ARM Assembly Language

» An overview of the ARM Assembly Instruction Set

» Hello Silicon

References:

» ARM Developer Guide

» Programming with 64-Bit ARM Assembly Language

» An overview of the ARM Assembly Instruction Set

» Hello Silicon

» Learn the Architecture-A64 Instruction Set Architecture