ARM Hardware and Assembly Language

Basically every smartphone on the planet currently uses an ARM processor, an inexpensive and energy-efficient microprocessor. The design dates back to the 1980's, when ARM stood for the "Acorn RISC Machine"--Acorn was the company, and more on RISC below. Compared to x86, which is a high performance but extremely complicated machine, ARM is much simpler technology and much easier to license, which makes it very popular for custom chip designs. In addition to cell phones, ARM is becoming popular in "embedded" systems like inside a hard drive, and NVIDIA has licensed ARM, probably for their upcoming massively parallel supercomputer chips.

From C or C++, an ARM machine is difficult to distinguish from any other 32-bit machine (64-bit ARM was quite late, and is only now starting to appear in high-end devices).

```
std::cout<<"Yes, this is ARM.\n";
return 37;
```

(Try this in NetRun now!)

However, note that the disassembly is completely different from x86:

ARM

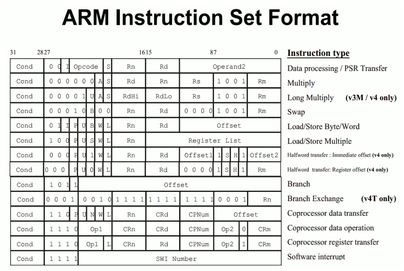
```
00000030 <foo>:
  30:   e92d4010 push   {r4, lr}
  34:   e59f0010 ldr    r0, [pc, #16] ...cout
  38:   e59f1010 ldr    r1, [pc, #16] ...str
  3c:   e3a02012 mov    r2, #18 ; 0x12
  40:   ebffffe bl     ...ostream...
  44:   e3a00025 mov    r0, #37 ; 0x25
  48:   e8bd8010 pop    {r4, pc}
```

x86

```
0000000000000027 <foo>:
  27:   48 83 ec 08             sub    rsp,0x8
  2b:   ba 11 00 00 00          mov    edx,0x11
  30:   be 00 00 00 00          mov    esi,...str
  35:   bf 00 00 00 00          mov    edi,...cout
  3a:   e8 00 00 00 00          call   ...ostream...
  3f:   b8 25 00 00 00          mov    eax,0x25
  44:   48 83 c4 08             add    rsp,0x8
  48:   c3                      ret
```

First, note that the machine code, on the left, is all in one uniform-sized block of binary data, not ragged like x86 machine code. This is because ARM is a "**Reduced Instruction Set Computer (RISC)**" machine, while x86 is a "Complex Instruction Set Computer (CISC)" machine. RISC refers to the fact that every ordinary ARM instruction is a uniform 32 bits long, while CISC machines use variable-length instructions: x86 uses 5 bytes for "mov eax,3" and just 1 byte for "ret". The advantage with RISC is the fixed-size instructions are simpler for the CPU to decode quickly; the advantage with CISC is you save space on short instructions, and can take as much room as you need for long instructions, like to load a 64-bit constant into a register. In the 1980's, many groups were building a RISC machine, such as MIPS (nearly extinct, clinging to a tiny niche in routers), Power/PowerPC (used in mid 1990's to mid 2000's Macs, and modern server versions sold by IBM), SPARC (now sold by Oracle), and DEC Alpha (now extinct).

ARM Instruction Set Format



Regarding **registers**, briefly:

Register	*AKA*	*Use*
r0		Return value, first function argument
r1-r3		Function arguments and general scratch
r4-r11		Saved registers
r12	ip	Intra-procedure scratch register, rarely used by the linker
r13	sp	Stack pointer, a pointer to the end of the stack. Moved by push and pop instructions.
r14	lr	Link register, storing the address to return to when the function is done. Written by "bl" (branch and link, like function call), often saved with a push/pop sequence, read by "bx lr" (branch to link register) or the pop.
r15	pc	Program counter, the current memory address being executed. It's very unusual, but handy, to have the program counter just be another register--for example, you can do program counter relative addressing very easily, by just loading from [pc+addr].

Like x86 64-bit, you need to align the stack *if* you're calling a function that uses floating point, but only to 8 bytes (not 16 bytes). All the gory details are in the ARM Architecture Procedure Call Standard, if you care.

ARM Instructions

First, note that the assembler syntax is quite different from x86:

- To make a **comment**, use @ to start the comment. x86 uses a semicolon for this. // comments also work, unlike x86!
- To insert a **constant**, by default you need to decorate it with "#" like "#3" (the corresponding GNU/AT&T x86 syntax uses "$3"). NetRun puts in ".syntax unified", so you don't actually need to do this explicitly, like x86 with NASM.

Here's how you return a small **constant**, in the return register r0. "mov" works exactly like x86. "bx lr" is a rough equivalent to "ret".

```
mov r0,#17  @ r0 is return value register
bx lr    @ return from function
```

(Try this in NetRun now!)

Now that we can load constants, we can do some **arithmetic**. The ARM "add" instruction takes three operands: the destination, and two separate source values. This is surprisingly handy for keeping you registers organized in a complicated function!

```
mov r2,#13
mov r3,#100
add r0,r2,r3

bx lr
```

(Try this in NetRun now!)

Because ARM is a RISC machine, there's not enough bits in a 32-bit mov instruction to fit a full 32-bit constant. They chose to combine an 8-bit constant with a 4-bit rotate(!), so you can mov any 8-bit value (0-255), or any value with up to 8 most significant bits, like #65536, but you can't mov #257 ("Error: invalid constant (#) after fixup"). For **bigger constants**, you need to load them from memory, by making a label, loading the address of the label with "adr" (address load), and then using the "ldr" (load register) instruction to load the data.

```
adr r1, .myconst  @ put address into r1
ldr r0, [r1]      @ load data from r1
bx lr

.myconst:
   .word 257
```

(Try this in NetRun now!)

Push and pop on ARM have super powers: you give them a *list* of registers to save and restore. They're smart enough to push and pop them in the opposite order too!

Here's how you save some preserved registers, and do some three-operand arithmetic. If you save the link register (lr), and pop into the program counter (pc), you'll magically return from the function!

```
push {r4-r7,lr}

mov r4,#10
mov r5,#100
add r0,r4,r5

pop {r4-r7,pc}   @ interesting hack: pop into the program counter to return from function
```

(Try this in NetRun now!)

Here's how you **call a function**, using "bl" or branch-and-link. This stores your return address into register lr, not the stack like x86, so you need to explicitly save main's lr so you can return there after you run. The first function argument goes into r0, just like the return value.

```
push {lr}   @ must save link register if we call our own function

mov r0,#123   @ r0 is first function parameter
bl print_int  @ branch-and-link (exactly like PowerPC)

pop {pc}   @ interesting hack: pop into the program counter to return from function
```

(Try this in NetRun now!)

Comparisons

Comparisons use the "cmp" instruction, followed by a conditional operation, exactly like x86. Unlike x86, *every* ARM instruction can be made conditional, not just jumps. This means you can compar and then do an "addgt" (add if greater-than), or a "movgt" (conditional move), or a "bgt" (conditional branch), etc.

```
mov r0,10
mov r1,7
mov r2,5
cmp r1,r2
addgt r0,r0,100
bx lr
```

(Try this in NetRun now!)

Here we're printing the integers from 0 to 10. We use r4 as the loop counter, because it is a preserved register. That tasty "blt" instruction does a branch if the compare came out less-than.

```
push {r4,lr}

mov r4,0
start:
        mov r0,r4
        bl print_int
        add r4,r4,1
        cmp r4,10
        blt start

pop {r4,pc}
```

(Try this in NetRun now!)

See this full list of comparison codes.

Here we're loading the address of an array to use as a function argument.

```
push {lr}     @ must save lr since we call a function

adr r0,mydata           @ first parameter: array memory address (program counter relative)
mov r1,#2               @ second parameter: array length
bl iarray_print

pop {pc}     @ function return

mydata:
        .word 123
        .word 456
```

(Try this in NetRun now!)

There's a pretty good summary of all ARM instructions over at HeyRick.

ARM Performance

Yeah, about that.

First, the NetRun ARM machine is only a Raspberry Pi 3 at 1.2GHz, so you'd expect it to be maybe 3x slower than a 4GHz x86. But it's actually a bit worse than that:

	ARM	**x86**
Empty function	7.5ns	1.2ns
Return constant	7.5ns	1.2ns
Load 2 constants	8.3ns	1.2ns
Load 2 and add	8.3ns	1.2ns

Note that the ARM is not only quite slow, additional instructions slow it down even further, while on x86 most instructions can be done simultaneously with other instructions, so they cost no time. This is because x86 is more deeply superscalar, having enough circuitry and data analysis to do more instructions at the same time.

Now that people are using ARM for real work, like in smartphones, performance suddenly matters, so folks are now working to build high performance ARM cores. While x86 has a several decade advantage in designing for speed, ARM is a simpler architecture, so it's not clear which architecture will win in the long run.
