Crazy Plan To Implement an AArch64 Code Generator

by Walter Bright
https://x.com/WalterBright
DConf Aug 2025

Rationale

It's insane to write another backend, but I am
doing it anyway. The D compilers come in three variants:
GDC (based on GCC), LDC (based on LLVM), and DMD
(based on Digital Mars C/C++). I've implemented code
generators for 8086, 386, 486, Pentium, Pentium Pro,
and x86_64 because I had to to get a working compiler.
The GDC and LDC D compilers already support the AArch64,
so why write another? If you are on the spectrum, this
presentation is for you!

Advantages of DMD's Code Generator

- It's fast
- I like to control the whole process from front to back
  - Don't want to have to extend someone else's back end
- It's not *that* hard to write a code generator
  - Mostly lots of details

Sometimes Crazy Things Can Have Unexpected Results

- Like ImportC that turned out to be able to translate C code to D code
  - That was certainly not part of the plan
  - I did not code it to do that
  - It was discovered
  - And turned out to be very valuable
- Though I cannot expect his code generator will do something unexpected (other than bugs!)

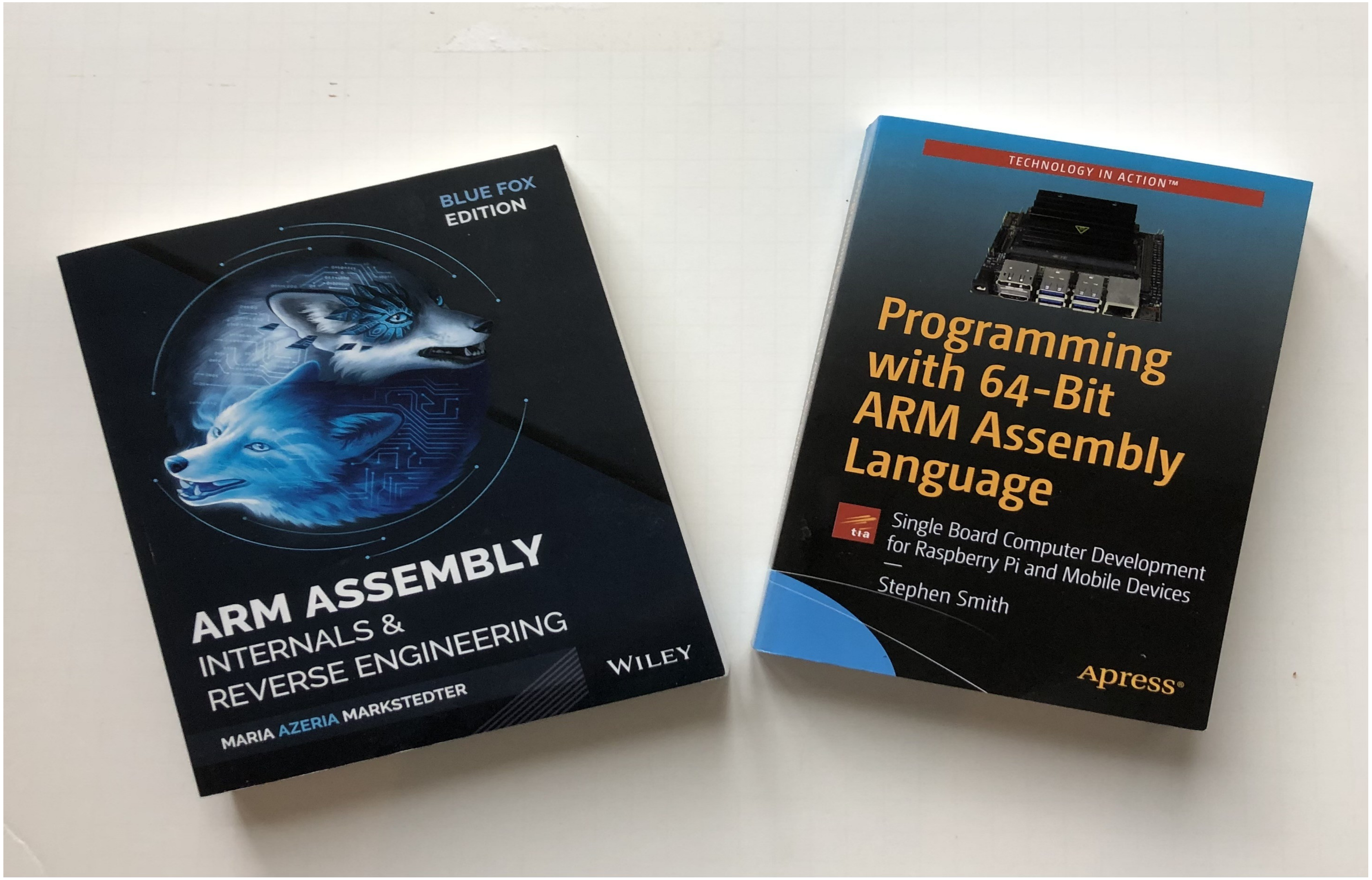
Ok, Ok, I Admit It

- I am a nerd
- And this is going to be fun!

Simplifying Constraints

- 64 bit ARM only
- Re-use as much of the X86_64 as possible
  - Register allocator
  - Structure of the code generator
  - Common subexpression logic
  - Stack layout
  - Same intermediate code
  - Object file generation
  - Test suite

Embarked Knowing Essentially Nothing about the AArch64


BLUE FOX EDITION
ARM ASSEMBLY
INTERNALS & REVERSE ENGINEERING
WILEY
MARIA AZERIA MARKSTEDTER
TECHNOLOGY IN ACTION™
Programming with 64-Bit ARM Assembly Language
Single Board Computer Development for Raspberry Pi and Mobile Devices
Stephen Smith
Apress®

Things To Leave Behind

- 32 bit code generation
- XMM instructions
- Op memory,immediate
- Op memory,register
- Op register,memory

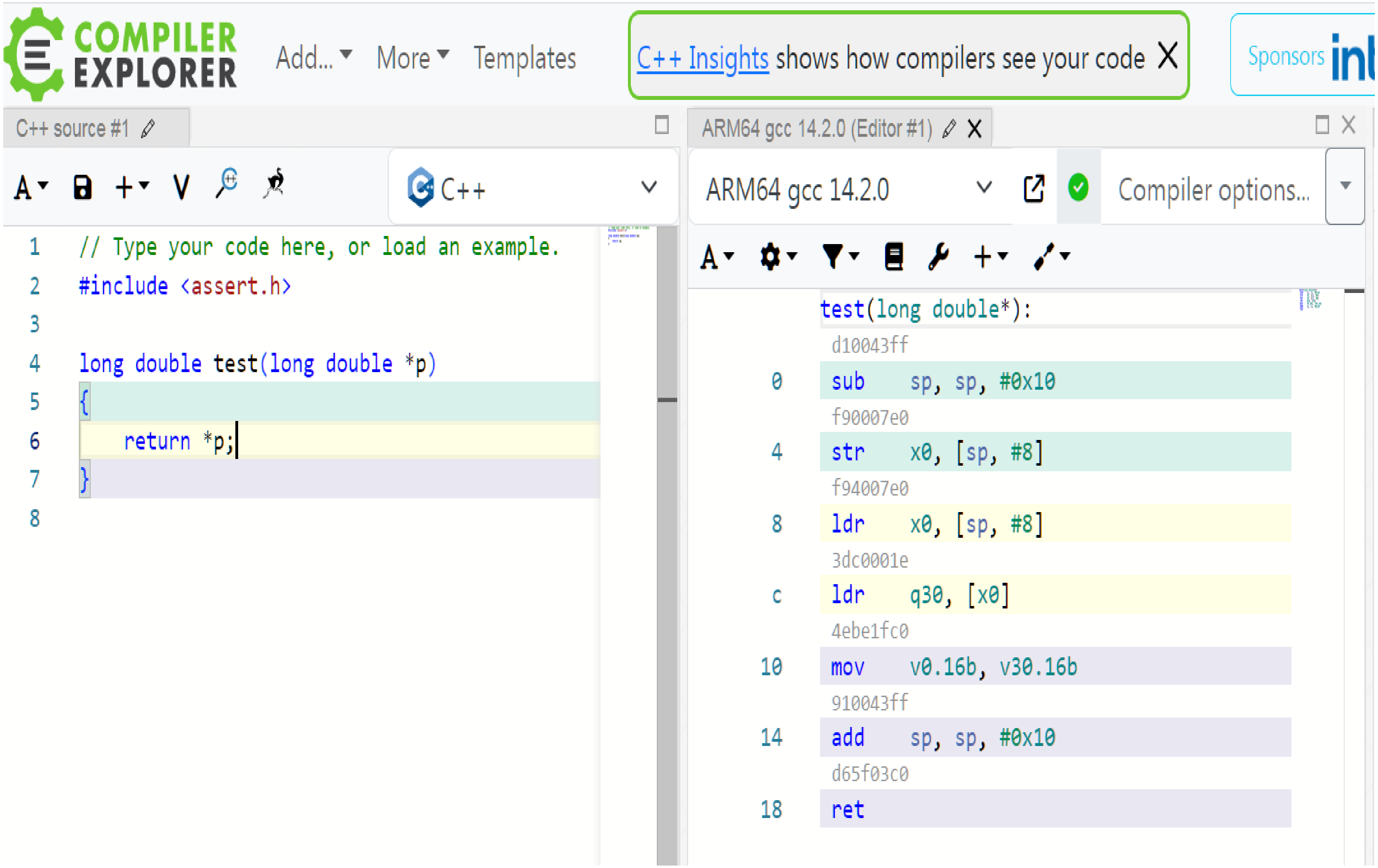
More To Leave Behind

- String instructions
- Larger struct parameters (pass by ref)
- Fake “floating point” register
- LEA x+c*i+offset in one operation

Things We Cannot Leave Behind

- Complex numbers (because of C)

Godbolt.org



```cpp
// Type your code here, or load an example.
#include <assert.h>

long double test(long double *p)
{
    return *p;
}
```

```
test(long double*):
     d10043ff
0    sub     sp, sp, #0x10
     f90007e0
4    str     x0, [sp, #8]
     f94007e0
8    ldr     x0, [sp, #8]
     3dc0001e
c    ldr     q30, [x0]
     4ebe1fc0
10   mov     v0.16b, v30.16b
     910043ff
14   add     sp, sp, #0x10
     d65f03c0
18   ret
```

Disassembler

```
real test(real *p) {
    return *p;
}
```

```
dmd test.d -arm -vasm
```

```
0000:   3D C0 00 00  ldr q0,[x0] // ldr_imm_fpsimd.html
0004:   D6 5F 03 C0  ret         // encodingindex.html#branch_reg
```

```
objdump -d test.o
```

```
   0:   3dc00000        ldr     q0, [x0]
   4:   d65f03c0        ret
```

Can Mechanically Convert

- Binary => Assembler
- Binary => Spec URL
- Spec URL => Binary
- Spec URL => Assembler

But Not:

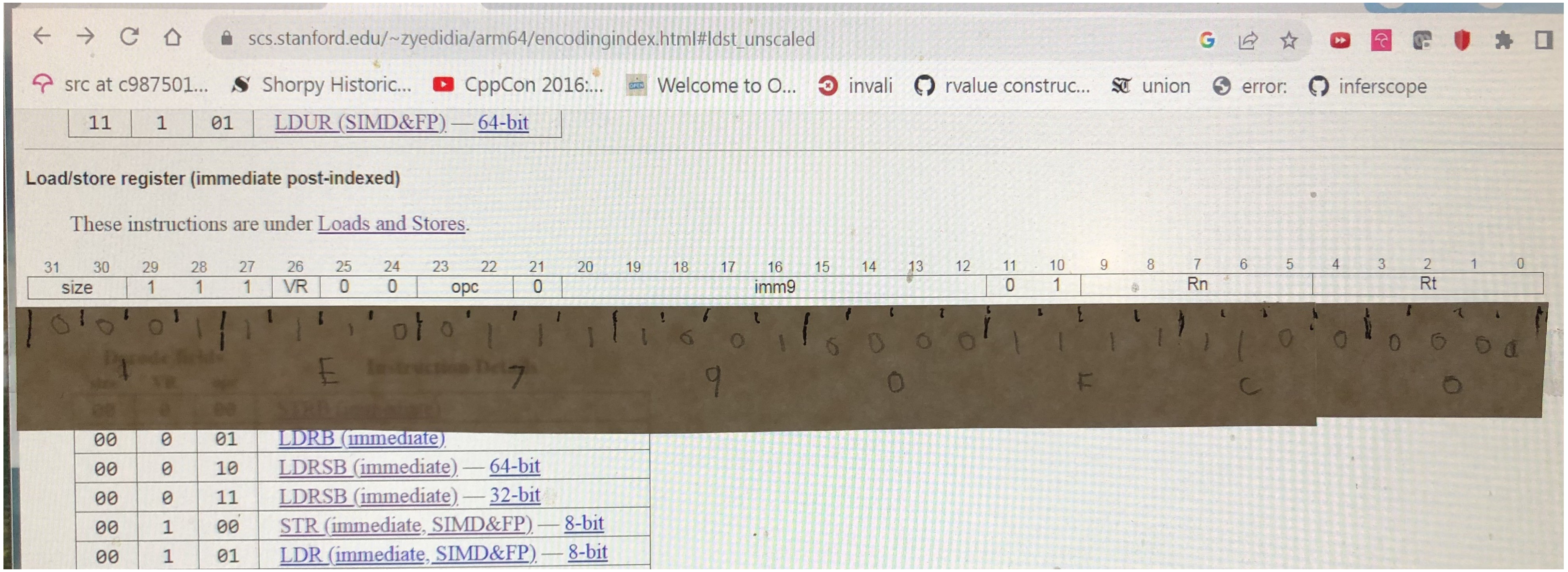
Assembler => Binary

Need to write an inline assembler for that

Instruction Layouts

- Done in “groups”
- Each group is defined by which bits are set
  - With a function to encode it
- Each instruction has a function to encode it
  - Which then calls its “group” function to finish it
- Generating a 32 bit value

Hex To Spec



scs.stanford.edu/~zyedidia/arm64/encodingindex.html#ldst_unscaled

src at c987501... Shorpy Historic... CppCon 2016:... Welcome to O... invali rvalue construc... union error: inferscope

11	1	01	LDUR (SIMD&FP) — 64-bit

Load/store register (immediate post-indexed)

These instructions are under Loads and Stores.

31 30	29	28	27	26	25	24	23 22	21	20 … 12	11	10	9 … 5	4 … 0
size	1	1	1	VR	0	0	opc	0	imm9	0	1	Rn	Rt

1 E 7 9 0 F C 0

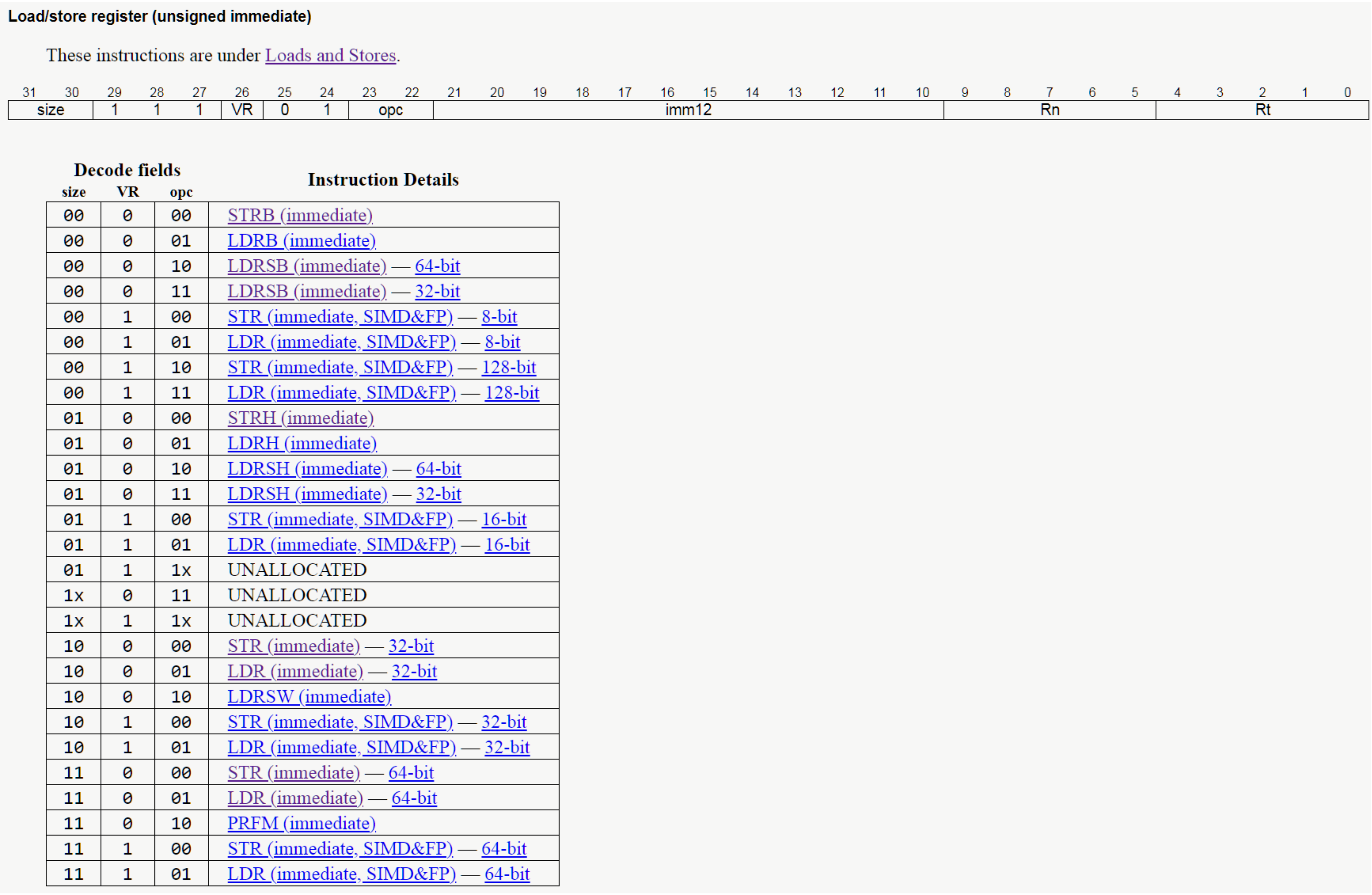
00	0	01	LDRB (immediate)
00	0	10	LDRSB (immediate) — 64-bit
00	0	11	LDRSB (immediate) — 32-bit
00	1	00	STR (immediate, SIMD&FP) — 8-bit
00	1	01	LDR (immediate, SIMD&FP) — 8-bit

But before the disassembler can handle the entire instruction set, one is reduced to converting the hex to binary, and then matching the binary up with the group patterns in the spec, one by one.

Tedious, but only has to be done once.

Nothing says auld skool like scissors, tape, pen and pencil.

ldst_pos



Load/store register (unsigned immediate)

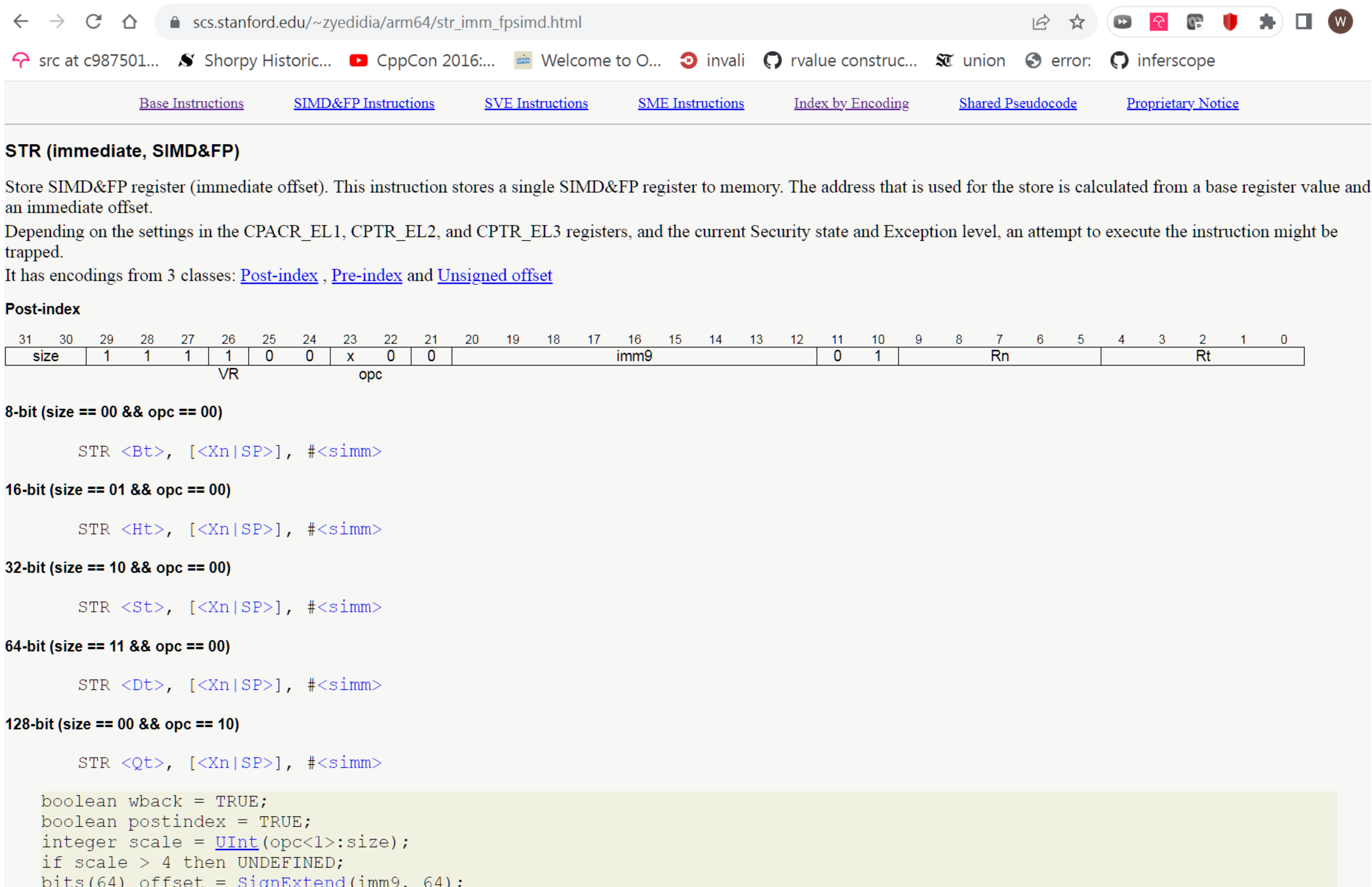
These instructions are under Loads and Stores.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	VR	0	1	opc		imm12												Rn					Rt				

Decode fields			Instruction Details
size	VR	opc	
00	0	00	STRB (immediate)
00	0	01	LDRB (immediate)
00	0	10	LDRSB (immediate) — 64-bit
00	0	11	LDRSB (immediate) — 32-bit
00	1	00	STR (immediate, SIMD&FP) — 8-bit
00	1	01	LDR (immediate, SIMD&FP) — 8-bit
00	1	10	STR (immediate, SIMD&FP) — 128-bit
00	1	11	LDR (immediate, SIMD&FP) — 128-bit
01	0	00	STRH (immediate)
01	0	01	LDRH (immediate)
01	0	10	LDRSH (immediate) — 64-bit
01	0	11	LDRSH (immediate) — 32-bit
01	1	00	STR (immediate, SIMD&FP) — 16-bit
01	1	01	LDR (immediate, SIMD&FP) — 16-bit
01	1	1x	UNALLOCATED
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	STR (immediate) — 32-bit
10	0	01	LDR (immediate) — 32-bit
10	0	10	LDRSW (immediate)
10	1	00	STR (immediate, SIMD&FP) — 32-bit
10	1	01	LDR (immediate, SIMD&FP) — 32-bit
11	0	00	STR (immediate) — 64-bit
11	0	01	LDR (immediate) — 64-bit
11	0	10	PRFM (immediate)
11	1	00	STR (immediate, SIMD&FP) — 64-bit
11	1	01	LDR (immediate, SIMD&FP) — 64-bit

```
/* Load/store register (unsigned immediate)
 * https://www.scs.stanford.edu/~zyedidia/arm64/
     encodingindex.html#ldst_pos
 */
static uint ldst_pos(uint size, uint VR, uint opc,
                     uint imm12, reg_t Rn, reg_t Vt) {
    //debug printf("imm12: %x\n", imm12);
    assert(imm12 <= 0xFFF);
    assert(VR == (Vt > 31));
    reg_t Rt = Vt & 31;
    return (size  << 30) |
           (7     << 27) |
           (VR    << 26) |
           (1     << 24) |
           (opc   << 22) |
           (imm12 << 10) |
           (Rn    <<  5) |
            Rt;
}
```

str_imm_fpsimd



STR (immediate, SIMD&FP)

Store SIMD&FP register (immediate offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	0	0	imm9									0	1	Rn					Rt				
					VR			opc																							

8-bit (size == 00 && opc == 00)

```
STR <Bt>, [<Xn|SP>], #<simm>
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>], #<simm>
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>], #<simm>
```

64-bit (size == 11 && opc == 00)

```
STR <Dt>, [<Xn|SP>], #<simm>
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Generating STR Instruction

```
/* https://www.scs.stanford.edu/~zyedidia/arm64/str_imm_fpsimd.html
 * STR <Vt>,[<Xn|SP>,#<simm>]  Unsigned offset
 */
static uint str_imm_fpsimd(uint size, uint opc, uint imm12,
     reg_t Rn, reg_t Vt)
{
    assert(imm12 < 0x1000);
    assert(size < 4);
    assert(opc  < 4);
    return ldst_pos(size,1,opc,imm12,Rn,Vt);
}
```

Disassembling STR/LDR

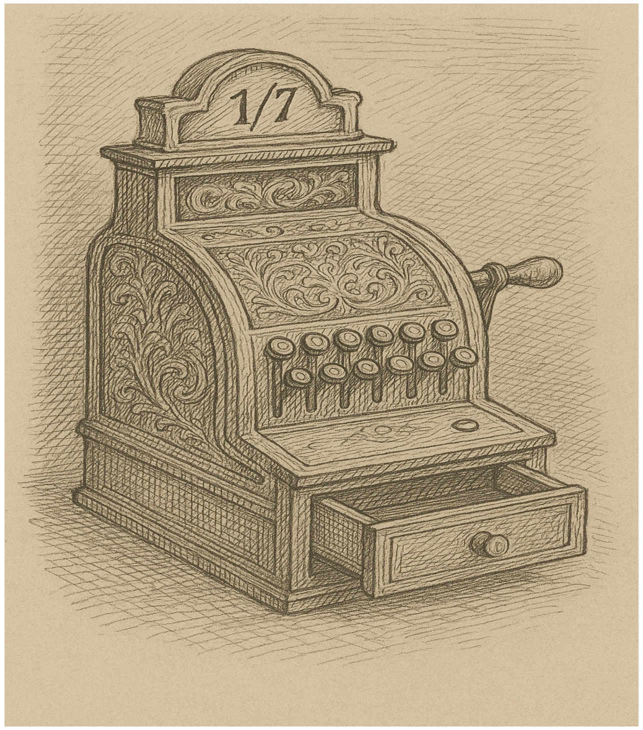
```
// Load/store register (unsigned immediate)
if (field(ins, 29, 27) == 7 && field(ins, 25, 24) == 1) // #ldst_pos
{
    url = "ldst_pos";

    uint size = field(ins, 31, 30);
    uint VR = field(ins, 26, 26);
    uint opc = field(ins, 23, 22);
    uint imm12 = field(ins, 21, 10);
    uint Rn = field(ins, 9, 5);
    uint Rt = field(ins, 4, 0);

    // str_imm_gen.html STR (immediate)
    // ldr_imm_gen.html LDR (immediate)

    uint ldr(uint size, uint VR, uint opc) { return (size << 3) | (VR << 2) | opc; }

    uint factor = 4;
    const(char)* format = "%s_imm";
    switch (ldr(size, VR, opc))
    {
        case ldr(0,0,0): p1 = "strb";  goto Lldr8;  // strb_imm.html
        case ldr(0,0,1): p1 = "ldrb";  goto Lldr8;
        Lldr8:
            p2 = regString(factor == 84, Rt);
            p3 = eaString(0, cast(ubyte)Rn, imm12);
            break;
...etc...
```

Registers

X86_64
AX BX CX DX SP BP SI DI R8..15 XMM0..31 ES NOREG ST01 ST0 STACK PSW

AArch64

r31 SP	Stack Pointer
r30 LR	Link Register
r29 FP	Frame Pointer (BP)
r19-r28	Callee-saved registers (if Callee modifies them)
r18	Platform Register
r17 IP1	intra-procedure-call temporary register
r16 IP0	intra-procedure-call scratch register
r9-r15	temporary registers
r8	Indirect result location register
r0-r7	Parameter / result registers



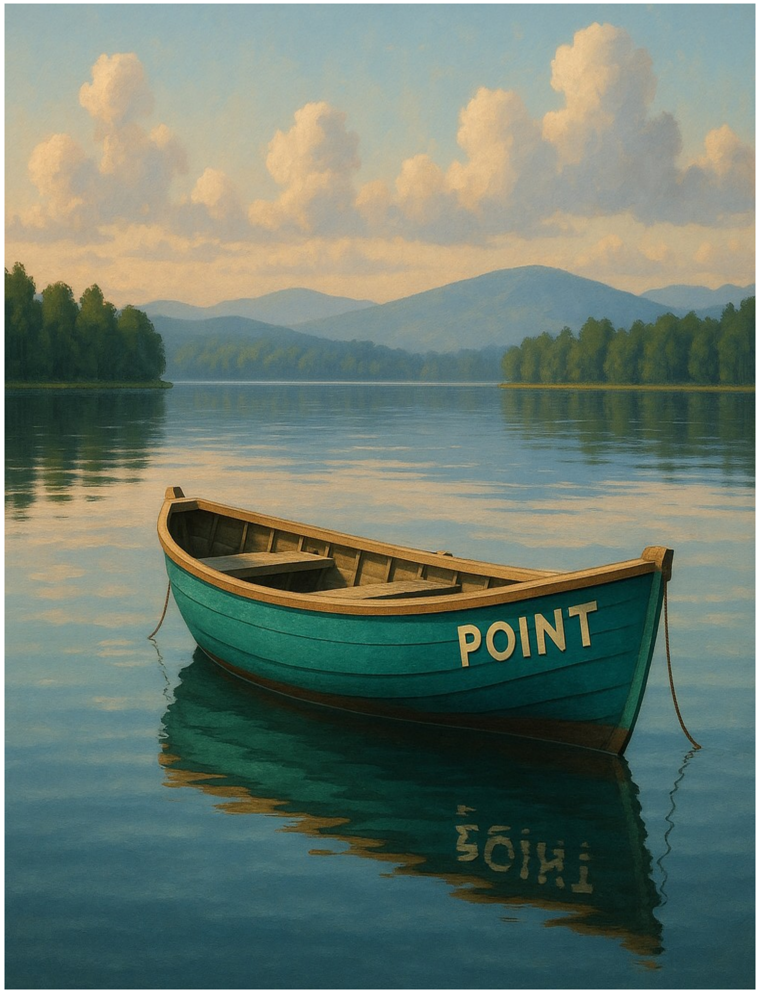
Floating Point Registers

v0-v7	Parameter / result registers
v8-v15	Callee-saved registers (only bottom 64 bits need to be saved)
v16-31	Temporary registers

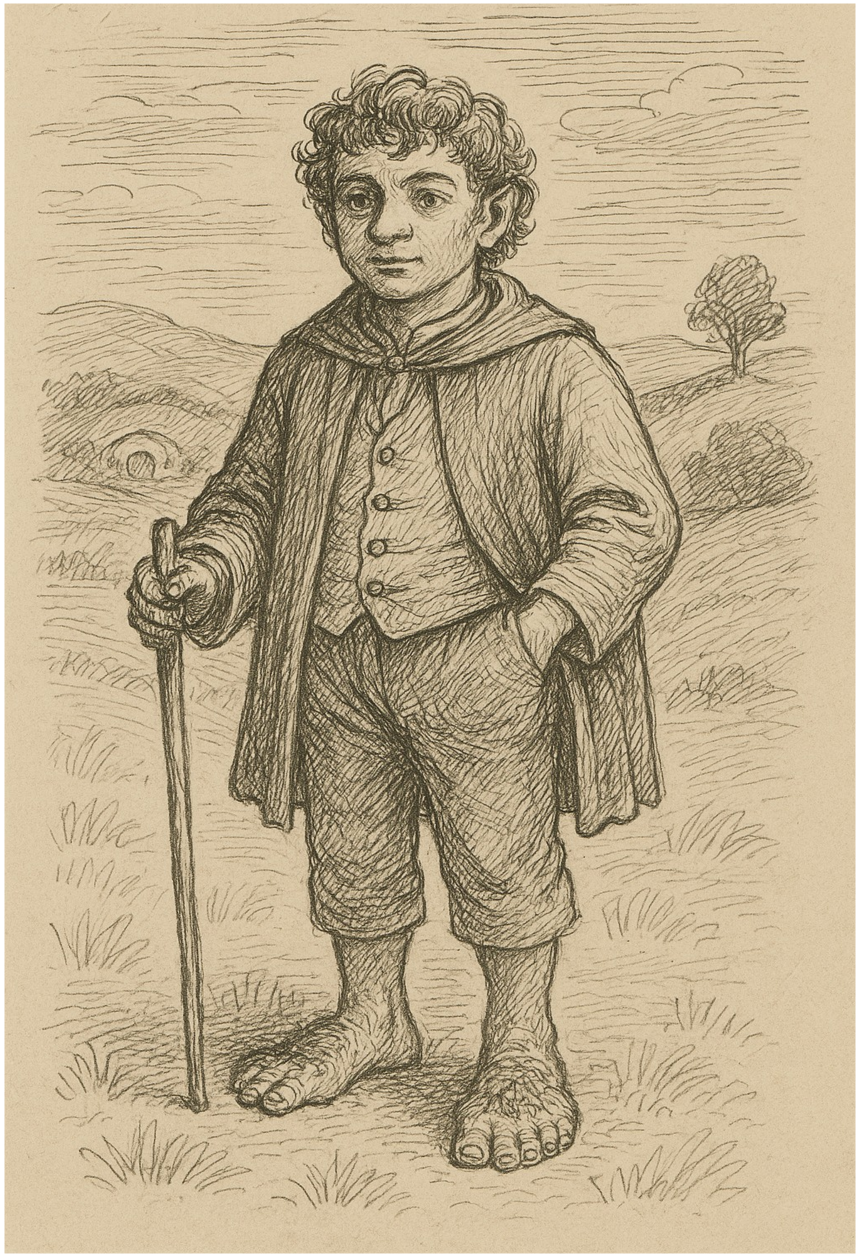
R0-28 … V0-24

Floating Point Types

- 16 bit
- 32 bit
- 64 bit
- 128 bit



Halfling Type



I Meant Half Float

- 16 bit floats not supported by D
- But falls out of the instruction set
- So backend is supporting it as I go, and it will eventually find its way into D as an extension

The Q Factor

- 128 bit floating point
- 32 Q registers, from Q0..Q31
- Overlaps the V floating point registers
- Uses library functions for arithmetic
- Contents are not preserved across function calls
  - Code generator does not track register types
  - So we cannot use Q8..Q15, as only bottom 64 bits are preserved

Function Calls

- Pretty ordinary
- structs <= 2 registers in size are passed in registers
  - Like Microsoft C
- Larger and non-POD are passed by reference
- Caller cleans the stack

Variadic Function Calls

```
void prolog_genvarargs(ref CGstate cg, ref CodeBuilder cdb, Symbol* sv)
{
    printf("prolog_genvarargs()\n");
    /* Generate code to move any arguments passed in registers into
     * the stack variable __va_argsave,
     * so we can reference it via pointers through va_arg().
     *   struct __va_argsave_t {
     *     ulong[8] regs;      // 8 byte
     *     ldouble[8] fpregs;  // 16 byte
     *     struct __va_list_tag // embedded within __va_argsave_t
     *     {
     *         void* stack;  // next stack param
     *         void* gr_top; // end of GP arg reg save area
     *         void* vr_top; // end of FP/SIMD arg reg save area
     *         int gr_offs;  // offset from gr_top to next GP register arg
     *         int vr_offs;  // offset from vr_top to next FP/SIMD register arg
     *     }
     *     void* stack_args_save; // set by prolog_genvarargs()
     *   }
     * The instructions seg fault if data is not aligned on
     * 16 bytes, so this gives us a nice check to ensure no mistakes.
    STR     x0,[sp, #voff+0*8]
    STR     x1,[sp, #voff+1*8]
    STR     x2,[sp, #voff+2*8]
    STR     x3,[sp, #voff+3*8]
    STR     x4,[sp, #voff+4*8]
    STR     x5,[sp, #voff+5*8]
    STR     x6,[sp, #voff+6*8]
    STR     x7,[sp, #voff+7*8]

    STR     q0,[sp, #voff+8*8+0*16]
    STR     q1,[sp, #voff+8*8+1*16]
    STR     q2,[sp, #voff+8*8+2*16]
    STR     q3,[sp, #voff+8*8+3*16]
    STR     q4,[sp, #voff+8*8+4*16]
    STR     q5,[sp, #voff+8*8+5*16]
    STR     q6,[sp, #voff+8*8+6*16]
    STR     q7,[sp, #voff+8*8+7*16]

    ADD     reg,sp,Para.size+Para.offset
    STR     reg,[sp,#voff+8*8+8*16+8*4] // set __va_argsave.stack_args
*/
```

```
/* Save registers into the voff area on the stack
 */
targ_size_t voff = cg.Auto.size + cg.BPoff + sv.Soffset;  // EBP offset of start of sv

if (!cg.hasframe || cg.enforcealign)
    voff += cg.EBPtoESP;

regm_t namedargs = prolog_namedArgs();
printf("voff: %llx\n", voff);
foreach (reg_t x; 0 .. 8)
{
    if (!(mask(x) & namedargs))  // unnamed arguments would be the ... ones
    {
        //printf("offset: x%x %lld\n", cast(uint)voff + x * 8, voff + x * 8);
        uint offset = cast(uint)voff + x * 8;
        if (!cg.hasframe || cg.enforcealign)
            cdb.gen1(INSTR.str_imm_gen(1,x,31,offset)); // STR x,[sp,#offset]
        else
            cdb.gen1(INSTR.str_imm_gen(1,x,29,offset)); // STR x,[bp,#offset]
    }
}

... etc ...
```

AArch64 Literal Encoding

- A 64 bit address does not fit
- A 32 or 64 bit value does not fit
- A 32 or 64 bit floating point value does not fit

X86_64 Literal Encoding

- Has variable length instructions
- Integer literals and addresses can be easily encoded in the instruction
- Floating point constants are loaded from memory

Integer Literals

- orr w0,w31,#0
  - mov w0,#0
- orr w0,w31,#12345
  - mov w0,#12345
- movn w0,#value
  - mov w0,#-value
- Two instructions to load 0x56781234
  - mov w0,#1234
  - movk w0,#5678,lsl #16

Floating Point Literals

- Encoded as an 8 bit floating point value!
  - 7: sign bit
  - 6-4: exponent
  - 3-0: mantissa
- More complex ones are loaded from memory

More Complex Floats

```
adrp    x0,0  R_AARCH64_ADR_PREL_PG_HI21 .rodata.cst4
ldr     s0,[x0] R_AARCH64_LDST32_ABS_LO12_NC .rodata.cst4
```

Taking The Address

```
adrp    x0, 0 R_AARCH64_ADR_PREL_PG_HI21 .bss
add     x0, x0, #0x0 R_AARCH64_ADD_ABS_LO12_NC .bss
```

State of Play

- Linux is the target (maybe Mac Mini)
- No complex numbers
- No SIMD
- Partial half float
- No inline assembler
- Get it to work, don't worry about optimization
- Partial 128 bit floats

Conclusions

- AArch64 is not simpler than X86_64
  - It is just complicated in different ways
- Mastering the instruction set is like trying to master C++
- The way to make progress on it is to not worry about optimal code yet
- The existing X86_64 code is a great guide to making the AArch64 generator work