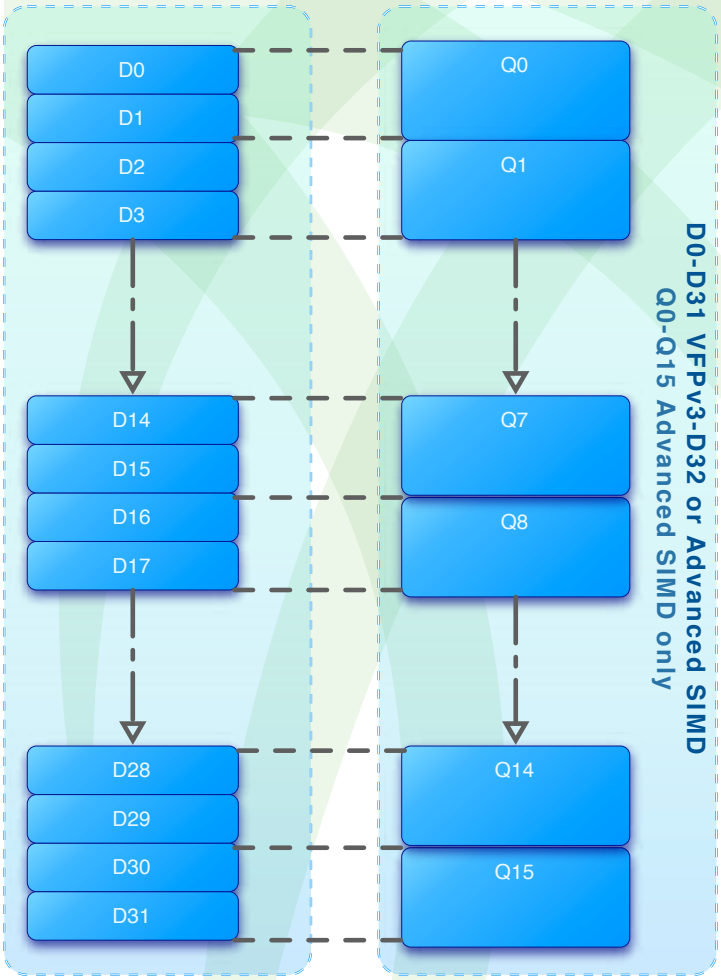
neon-test

TEST CODE TUTORIAL

In this document, we step through the test suite "neon-test" in order to learn about coding for the neon processor.



Neon Register Set

The NEON register bank consists of thirty-two 64-bit registers. If both Advanced SIMD and VFPv3 are implemented, they share this register bank. The same register bank can be viewed as sixteen 128-bit quadword registers [Q0-Q15], or thirty-two 64-bit doubleword registers [D0-D31]. Each of the Q0-Q15 registers maps onto a pair of D registers. When you are writing software, you can freely switch between either view of the registers. The instruction determines whether a D register or a Q register is being accessed. Most instructions can operate on different data types. This is encoded in the instruction.

What is NEON?[1]

The implementation of the Advanced Single Instruction Multiple Data (SIMD) extension used in ARM processors is called NEON. NEON technology is implemented on all current ARM Cortex-A series processors.

SIMD allows a processor to operate on multiple sets of data in parallel with one instruction. For the right type of problem, meaning one that can be divided up into parallel operations, this speeds up performance.

NEON instructions are executed as part of the ARM or Thumb instructions that make up a program. ARM or Thumb instructions take care of program flow and synchronization. NEON instructions handle memory accesses, data copying between NEON and general purpose registers, data type conversion, and data processing.

Usually, both Vector Floating Point (VFP) extension and NEON extension are implemented together. They have many common features, so using them together makes sense.

NEON instructions support 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers, as well as 32-bit single-precision floating point elements, and 8-bit and 16-bit polynomials. There is an additional extension called half-precision extension which enables NEON to convert between single-precision floating point and half-precision floating point. In other words, there are extensions to the extension! Knowing what extensions are implemented on the processor in use is a necessary early step to successful code writing.

---

1 Adapted from "Introducing NEON" ARM DHT 0002A ID060909

The following test code, complex_mult.c, was part of neon-test as of mid-October 2009. There may be some variations in the current test code from this package.

COMPLEX_MULT.C

Font Key for Tutorial
`#include <stdio.h>` indicates source code directive to include standard c library input and output header file indicates tutorial commentary

`#include <stdio.h>`
directive to include standard c library input and output header file

`#include <stdlib.h>`
directive to include standard c library header file

```
void
complex_mult(const float *a, const
float *b, float *r)
{
```
Here is where the function complex_mult begins. It takes three arguments and does not return a result. The three arguments are three pointers to float values. The value pointed to by *a and *b are designated as const, which means they do not change within this function. A and B are the numbers to be multiplied. R is the result. *a dereferences to A, *b dereferences to B, and *r dereferences to R.

```
   double d0, d1, d2, d3;
```
Four double values are declared. These are automatic variables, which means they exist within the function. Notice that these are not guaranteed to be initialized.

Now, how large are the things we have declared? float is 32 bits and double is 64 bits.

```
#if  0
```
This is a shorthand for a section of code that will never be compiled, but that one may want to keep for either archival or future purposes. The #else block that follows the #if 0 block does get compiled.

```
 asm volatile (
"vld1.32   {%P0}, [%5] \n\t"
"vld1.32   {%P1}, [%6] \n\t"
"vmul.f32  %P2, %P0, %P1       \n\t"
"vrev64.32 %P3, %P0            \n\t"
"vmul.f32  %P3, %P3, %P1       \n\t"
"vneg.f32  %2,  %2             \n\t"
"vtrn.32   %P2, %P3            \n\t"
"vsub.f32  %P2, %P2, %P3       \n\t"
"vst1.64   {%P2}, [%4] \n\t"
: "=&w"(d0), "=&w"(d1), "=&w"(d2), "=&w"(d3)
: "r"(r), "r"(a), "r"(b)
: "memory");
```

```
#else
```
This #else is related to the previous #if statement.

```
     asm volatile (
```
asm indicates that assembly instructions follow, within the pair of parentheses. volatile instructs the compiler to compile everything as written, and do not attempt to delete, move, or combine the instructions. This is because it's hand-written to proceed in a particular way.

```
"vld1.32   {d0}, [%1]          \n\t"
```
Vector load one element 32 bits wide from our input operand [%1] to our NEON register {d0}. This moves data from memory to a register. Operands have a single % sign as prefix to distinguish them from registers.

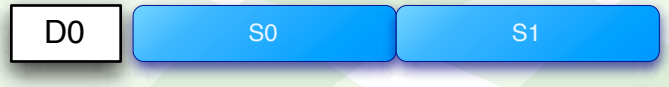
The brackets in [%1] are much like an asterisk in *ptr.

When we see [%1], then we interpret it as "Take the register the compiler assigned as the 1st operand, treat the contents as an address, and look up that address in memory". In this vector load instruction, [%1] provides a base address.

There are several vector loads in the instruction set. Each does different things based on the way the instruction is written. Here is what d0 looks like:

NEON-TEST

Nostrud dolute facinit iriurero odio do et dit, commy nisl ea faccum quip enim del eros nim dolorem zzrit, quatuer alis alisi tio od duis esse-nit, quate commodo odolore consed magna.



Note that d0 is 64 bits wide. It's composed of s0 and s1, each being a VFP/NEON register 32 bits wide. It may seem like we're trying to load 32 bits into a 64-bit register and then moving on, but that's not what ends up happening. In order to understand what is going on, we have to jump ahead in the source code.

Look at the end of the block of assembly code below - specifically, the two instructions right before the closing parenthesis. We'll jump back to this point after the explanation. Hang in there!

```
"vld1.32   {d1}, [%2]          \n\t"
"vmul.f32  d2, d0, d1          \n\t"
"vrev64.32 d3, d0              \n\t"
"vmul.f32  d3, d3, d1          \n\t"
"vneg.f32  s5, s5              \n\t"
"vtrn.32   d2, d3              \n\t"
"vadd.f32  d2, d2, d3          \n\t"
"vst1.64   {d2}, [%0]          \n\t"
```

Skip over the above assembly instructions for now - we'll get back to this shortly.

```
          :: "r"(r), "r"(a), "r"(b)
          : "d0", "d1", "d2", "d3",
"memory");
```

"Wot's all this?"

Notice what appears to be a double colon, followed by some symbols, then another colon, followed by some more symbols? Here is what this format indicates[1].

```
asm volatile(
lots of assembly instructions
:"=r"(b)      /* output */
:"r"(a)       /* input */
:"%eax"       /* clobbered register */
);
```

What looks like a leading double colon in our test code is actually two single colons with nothing in between them. The leading double colon (::) means that there is no output. This makes sense because our function is type void. The input parameters follow the second colon and are "r"(r), "r"(a), "r"(b). What do these mean? They correspond to the arguments in the function definition. We are setting up a place for these incoming arguments to live.

"r"(a) is the programmer telling the compiler "this assembly code expects a register to contain a. Please arrange for this to be true before calling my assembly code. You can pick any register you prefer for this purpose, and I'll just put %1 when I mean that register. "r"(r) is the 0th operand. "r"(a) is the 1st operand. "r"(b) is the 2nd operand.

Being able to find the address of these set-

Code Design

"What do you think about the way colons are used here?"

"The colons are ok with me. What bugs me is that this stuff is set up is at the end. It should be at the beginning so that the restrictions are in context. this would make reading the code easier."

1 referenced from http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

aside storage locations is very important, especially when we want to store the result of our work back to memory. [%0] is the address of the register set aside to contain operand 0, [%1] is the address of the register set aside to contain operand 1, and [%2] is the address of the register set aside to contain operand 2. Anytime we need the addresses of these operands, we use [%n], where n is the number of the operand, starting with 0.

In summary, "r"(r), "r"(a), "r"(b) direct the compiler to generate the code to load the addresses of the two operands we're going to multiply, as well as the address where we will store the result.

After the third colon, we indicate what's "clobbered". This tells the compiler that the value in registers d0, d1, d2, d3 and something called memory are to be modified by the block of assembly code. In other words, we don't want the compiler to use these registers to store any other value, because the value will change. This is a safety lockout on the registers.

Now, let's skip back to the top of the block of assembly code and step through it.

```
"vld1.32   {d1}, [%2]              \n\t"
```

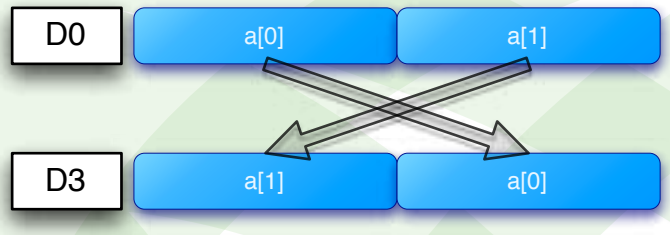
Vector load one element 32 bits wide from our input operand [%2] (associated with a register by that colon magic) to register d1.

```
"vmul.f32  d2, d0, d1              \n\t"
```

In this instruction set, the flow is from right to left. This is a vector multiplication of two 32-bit floating point datatypes that happen to be housed in a 64-bit wide set of registers. This means that 32-bit wide lanes of d0 and d1 are multiplied together and these results are placed in d2.

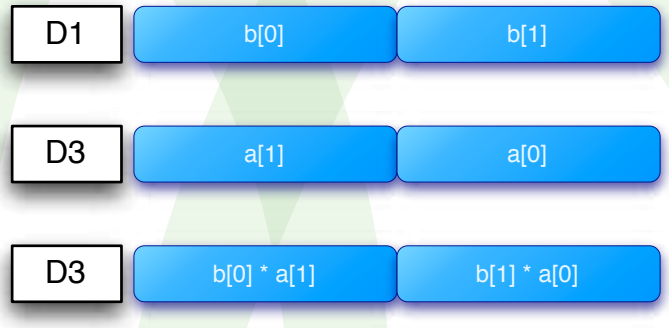

```
"vrev64.32 d3, d0                  \n\t"
```

This instruction reverses the order of 32-bit elements within each double word of the vector. The source register is d0 and the destination register is d3. Here is a graphic of what is happening. The two-element array that had been loaded into d0 has been loaded into d3, but with each element swapped. As we will see shortly, his makes the complex multiplication easier.



"vmul.f32 d3, d3, d1 \n\t"

Vector multiplication of two 32-bit floating point datatypes that happen to be housed in a 64-bit wide set of registers. This means that 32-bit wide lands of d1 and d3 are multiplied together and these results placed in d3.



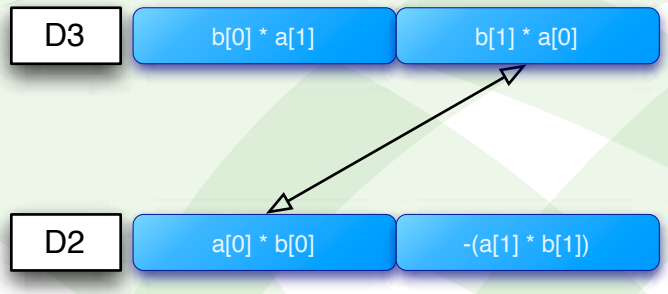
"vneg.f32 s5, s5 \n\t"

Next, we negate. This instruction takes s5, negates it, and puts it back in s5. On page 3 we showed that each 64-bit d register is composed of two 32-bit wide s registers. Here is where s5 is:

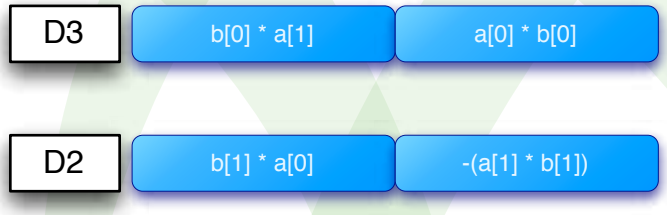
s5 is the latter half of d2, and is (a[1] * b[1]). We need -(a[1] * b[1]) later on, so this operation is useful.

"vtrn.32 d2, d3 \n\t"

This is a vector transpose, where the elements are 32-bits wide. Here is what's happening.



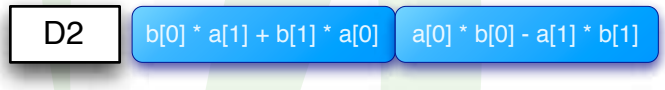
And here are the values of d3 and d2 after the transpose.



The second half of d3 is swapping places with the first half of d2. Both registers are modified. The 32-bit wide lanes of each register are treated like elements of a 2 by 2 matrix.

```
"vadd.f32  d2, d2, d3              \n\t"
```

Taking d2 and d3, we add them element-wise. Each 32-bit wide lane is added, and the result placed in d2.




```
"vst1.64   {d2}, [%0]              \n\t"
```

Store the contents of d2, a 64-bit register which contains our result, into memory. We previously set aside memory with compiler directives. %0 is a register, which happens to contain an address. [%0] is the value found at that address. The brackets are much like an asterisk in *ptr.

When we see [%0], then we interpret it as "Take the register the compiler assigned as the 0th operand, treat the contents as an address, and look up that address (and as many subsequent consecutive addresses as you need to fill up the right size vector) in memory". In this vector store instruction, [%0] provides a base address, and we move data from d2 into memory until we've moved all of it.

`#endif`

This marks the end of the #if 0, #else thingie that had us choose between two blocks of assembly code. As mentioned above, there isn't a test here, to choose between them. The second block is compiled and the first one is not.

```
}
```
This closing curly bracket marks the end of the complex_mult function. There is no return value because it is a void function.

```
int main()
{
```
Here is where the function main begins. The `int` indicates that main returns an integer result. It appears to take no arguments, but interestingly enough, main is a special case because its arguments are provided by the runtime assembly language that comes with the compiler (This will not be on the exam!).

```
    volatile float a[2], b[2], r[2];
```
Volatile instructs the compiler to compile everything as written, and do not attempt to delete, move, or combine the instructions. This is because it's hand-written to proceed in a particular way. a[2] and b[2] are the multiplicands. r[2] is the result. a[2], b[2], and r[2] are each two-element-wide arrays of type float.

```
    int i;
```
We set up a variable i as an integer.

```
    a[0] = 1;
    a[1] = 2;
    b[0] = 3;
    b[1] = 4;
```
Here is where we initialize the elements of the array. `a[0] = 1;` is a bit terse. It should be
`a[0] = 1.0;`
or
`a[0] = (float)1;`
or, if you are Franklin Antonio, you would use
`a[0] = 1.;`

```
    for (i=0; i<1e8; i++) {
```
This is a loop. i is set to zero and is post-incremented once per loop. When i reaches 1e8, then execution continues past the closing curly brace.

There is a possible portability problem here that's worth a word or two. i<1e8 might be too large if the code was ported to another processor where 1e8 was too large for an integer to hold. However, we're always running on an ARM with NEON code, so this is perfectly fine. The purpose of the number 1e8 as the loop limit is to cause the calculations to run enough times in order to show any speedup from the optimization.

In order to make this particular type of statement as portable as possible, it's a good idea to use types, like INT32, when declaring integers.

```
#ifdef NEON
```
if the macro NEON is defined, then execute the following conditional block.

```
            complex_mult(a, b, r);
```
Call the complex_mult function with the arguments a, b, and r.

```
#else
```
if NEON is not defined, then do this conditional block.

```
r[0] = (a[0]*b[0] - a[1]*b[1]);
r[1] = (a[1]*b[0] + a[0]*b[1]);
```

The elements of the array are multiplied together as a complex multiplication, using the definition of complex multiplication[2]. The arrays a and b stand for complex numbers, where the first element of each array is the real component, and the second element of each array is the imaginary component.

```
#endif
      }
```
End the test as to whether or not NEON is defined.

```
      printf("Result = (%f, %f)\n",
r[0], r[1]);
```
Present the result.

```
      return 0;
}
```
We return a value of 0 and the function main ends. It is traditional to return 0 if the program operated correctly.

2 http://en.wikipedia.org/wiki/Complex_number#Operations

NEON-TEST

The following test code, dotproduct.c, was part of neon-test as of mid-October 2009. There may be some variations in the current test code from this package.

DOTPRODUCT.C

```
#include <stdio.h>
```
directive to include standard c library input and output header file

```
#include <stdlib.h>
```
directive to include standard c library header file

```
#ifdef U0
```
if the macro UO is defined, then execute the following conditional block.

```
float
dotprod_fff_cortex_a8(const float
*a, const float *b, size_t n)
{
```
Here is where the function dotprod_fff_cortex_a8 begins. It takes three arguments and returns a result of type float. The three arguments are two pointers to float values and size_t n, which is an integer large enough to hold the SIZE of any data type. The "t" stands for type.

The value pointed to by *a and *b are designated as const, which means they do not change within this function. A and B are the numbers to be "dot productized". *a dereferences to A, *b dereferences to B. We set up a loop to iterate over the n elements of the input vector. n is used in that loop. The value for n is passed in from main as the constant value 256 when main calls dotprod_fff_cortex_a8(a, b, 256).

```
  float sum = 0;
```
A float is declared and initialized to zero.

```
  size_t i;
```
An integer "large enough to hold the SIZE of any data type" is declared.

```
  for (i = 0; i < n; i++){
```
We set up a loop to iterate over the n elements of the input vector. i starts at zero and is post-incremented after each time through the loop. When it reaches the value 256, we exit the loop.

```
    sum += a[i] * b[i];
```
Within the loop, for each iteration, we multiply what looks like the i^{th} element of a times the i^{th} element of b. We accumulate the result in the float variable called sum. This is C pointers at work. They are passing around a pointer to a float, which through the magic of pointer arithmetic can be used to address any number of floats contiguous in memory (that is, an array of floats).

```
  }
  return sum;
}
```
We return "sum", which is the result of the accumulation.

```
#endif
```
The conditional block started by #ifdef U0 is concluded.

```
#ifdef U1
```
if the macro U0 is defined, then execute the following conditional block.

```
float
dotprod_fff_cortex_a8(const float
*a, const float *b, size_t n)
{
```
Here is where the function dotprod_fff_cortex_a8 begins. It takes three arguments and returns a result of type float. The three arguments are two pointers to float values and size_t n, which is an integer large enough to hold the SIZE of any data type. The "t" stands for type.

The value pointed to by *a and *b are designated as const, which means they do not change within this function. A and B are the numbers to be "dot productized". *a dereferences to A, *b dereferences to B. Here, n is used in a branch instruction. The value for n is passed in from main as the constant value 256 when main calls dotprod_fff_cortex_a8(a, b, 256)

```
    float s = 0;
```
A float is declared and initialized to zero. We use this only inside the function.

```
    asm volatile (
```

asm indicates that assembly instructions follow, within the pair of parentheses. volatile instructs the compiler to compile everything as written, and do not attempt to delete, move, or combine the instructions. This is because it's hand-written to proceed in a particular way.

Now, even though we have assembly code to look at, let's skip ahead to the compiler constraints section to make sure we know what to do with the operands.

```
        "vmov.f32  q8, #0.0                    \n\t"
        "vmov.f32  q9, #0.0                    \n\t"
        "1:                                    \n\t"
        "subs      %3, %3, #8                  \n\t"
        "vld1.32   {d0,d1,d2,d3}, [%1]!        \n\t"
        "vld1.32   {d4,d5,d6,d7}, [%2]!        \n\t"
        "vmla.f32  q8, q0, q2                  \n\t"
        "vmla.f32  q9, q1, q3                  \n\t"
        "bgt       1b                          \n\t"
        "vadd.f32  q8, q8, q9                  \n\t"
        "vpadd.f32 d0, d16, d17                \n\t"
        "vadd.f32  %0, s0, s1                  \n\t"

        : "=w"(s), "+r"(a), "+r"(b), "+r"(n)
        :: "q0", "q1", "q2", "q3", "q8", "q9");
```

As a reminder, the compiler constraint format is as follows.

```
asm volatile(
lots of assembly instructions
:"=r"(b)      /* output */
:"r"(a)       /* input */
:"%eax"       /* clobbered register */
);
```

So, following the first colon are four output constraints. Let's look at the first one, "=w"(s). The w, used here, is a constraint letter for assembly in gcc that indicates the register is a VFP floating point register. This is specific to ARM. The "=" means write only and is a natural fit for the output section. The next three constraints are all "+r". The + means "read and write, operand is both input and output". Even though these constraints are in the output section, they may act as both input and output. The r indicates that a register in memory is set aside for the operand. The code decrements n, so it's used as a variable. It's decremented in place. The inputs are float a, float b, and integer n. There are no inputs listed.

Now that we know what registers have been set aside in memory, let's go back and step through the assembly code block.

```
"vmov.f32   q8, #0.0                    \n\t"
```

Taking 32 floating point bits at a time, load zeros into q8. The constant specified by # is replicated to fill the NEON register.

```
"vmov.f32   q9, #0.0                    \n\t"
```

Taking 32 floating point bits at a time, load zeros into q9.The constant specified by # is replicated to fill the NEON register.

```
"1:                                     \n\t"
```

A label is written as a symbol followed by a colon. This will be used to jump back to this location, when we get to the branch instruction.

```
"subs       %3, %3, #8                  \n\t"
```

This is a subtraction instruction. The s added to sub is an optional suffix. It indicates that condition code flags are updated upon the result of the subtraction. The first %3 is the destination register, where the result will be placed. The second %3 is the first operand of the subtraction. The #8 is an immediate value, and is allowed to be in the range of 0 - 4095. This instruction decrements n by 8. N is the number of floating point words in each vector, and was input initially as 256 (that happens when this function is called by main, so skip ahead to the main function if you want to see where that occurs).

We decrement by 8 because we can do 8 loops at a time, instead of just one. This gives a hint of the power of SIMD architecture in terms of being able to parallel instructions to a greater degree than "regular" processors.

We use %3 instead of [%3] because we are working with the contents of the register, and not using the register to store an address.

```
"vld1.32    {d0,d1,d2,d3}, [%1]!        \n\t"
```

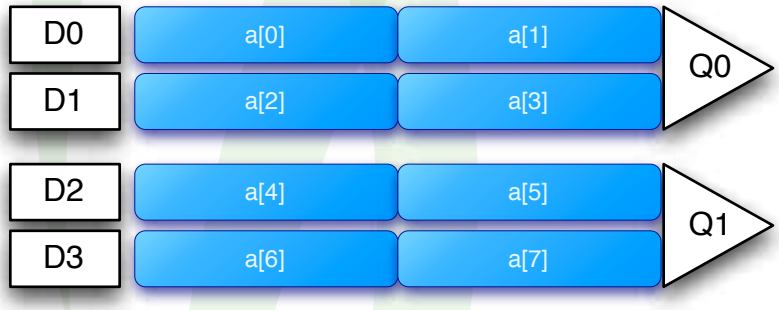
So what does this do? Vector load multiple n-element structures. Here, n = 1. [%1] points to a 1-element structure that is 32 bits wide. We put the contents pointed to by the base address of [%1] into each of the NEON registers listed. We start with [%1] as if it was the first address of an array of 32-bit wide elements. We ratchet through, filling up d0, d1, d2 and then d3 with the contents at the addresses following [%1].

The ! means that [%1] is updated to (Rn plus the number of bytes transferred by the instructions) after all the loads and stores.

d0 is 64 bits wide. There are two spots for 32-bit wide elements in each d register. Here is a diagram of the relationship between the 32-bit s registers, the 64-bit d registers, and the 128-bit q registers.

[%1] is the base address of the array of 256 elements that we are going to dot product with a second 256-element array. After we accomplish the vector load, we update [%1] by the number of bytes transferred by the instruction. Each element is 32 bits wide. We are using 8-bit bytes, so each 32-bit load is 4 bytes. We loaded 8 registers. 8 registers x 4 bytes = an update to [%1] of 32. This means that [%1] is now pointing at the 9th element in the 256 element array.

Here is what the registers look like after the load. The "a" corresponds to the "a" in the second operand listed in the compiler constraints after the first colon. "a[0]" is the first element of the array. "a[1]" is the second element of the array, and so on.




```
"vld1.32    {d4,d5,d6,d7}, [%2]!        \n\t"
```

Here we do the same type of vector load for d4, d5, d6, and d7. We're using the base address specified by [%2]. This is a second 256-element array that will be dot-productized with the first 256-element array.

Here is what the registers look like after the load. The "b" corresponds to the "b" in the third operand listed in the compiler constraints after the first colon. "b[0]" is the first element of the array. "b[1]" is the second element of the array, and so on.


```
"vmla.f32  q8, q0, q2                    \n\t"
```

q8 and q9 were zeroed out earlier by a `"vmov.f32  q9, #0.0"` and `"vmov.f32  q9, #0.0"` instructions. This instruction does a multiply and accumulate. q8 is the destination. q0 is the 1st operand and q2 is the second operand. q0*q2 += q8.

Quick quiz: Why is VLD1.32 not have a float datatype and VLMA.f32 have a float datatype? (a quick reminder: in general, the datatype is the part of the instruction following the opcode)

Answer: loads don't care about the datatype. It's just bits when you load. Multiply, however, needs to know what kind of number it's dealing with.

Here is what the q8 register looks like after the first multiply and accumulate.


```
"vmla.f32  q9, q1, q3                    \n\t"
```

We do another multiply and accumulate. q1*q3 += q9.


```
"bgt        1b                           \n\t"
```

This instruction is a branch if greater than. What are we comparing? OK, remember those condition code flags from `"subs %3, %3, #8"` ? We look at those condition codes here in order to decide whether or not to jump back to label 1 and load more elements of the 256-element

arrays.

The 1b means to look back for label 1. We're looking backwards, up the code, for a label equal to 1. That's where we jump if the result of the subtraction is still positive. In essence, this test asks whether or not we have run out of our 256 elements yet. So, when the subtraction reaches zero, which it will after 32 rounds, then we fall through to the next instruction. q8 and q9 look something like this:

D16	a[0]*b[0] + a[8]*b[8] + a[16]*b[16] + ... + a[252]*b[252]	a[1]*b[1] + a[9]*b[9] + a[17]*b[17] + .. + a[253]*b[253]
D17	a[2]*b[2] + a[10]*b[10] + a[18]*b[18] + ... + a[254]*b[254]	a[3]*b[3] + a[11]*b[11] + a[19]*b[19] + ... + a[255]*b[255]

D18	a[4]*b[4] + a[12]*b[12] + a[20]*b[20] + ... + a[248]*b[248]	a[5]*b[5] + a[13]*b[13] + a[21]*b[21] + .. + a[249]*b[249]
D19	a[6]*b[6] + a[14]*b[14] + a[22]*b[22] + ... + a[250]*b[250]	a[7]*b[7] + a[15]*b[15] + a[23]*b[23] + ... + a[251]*b[251]

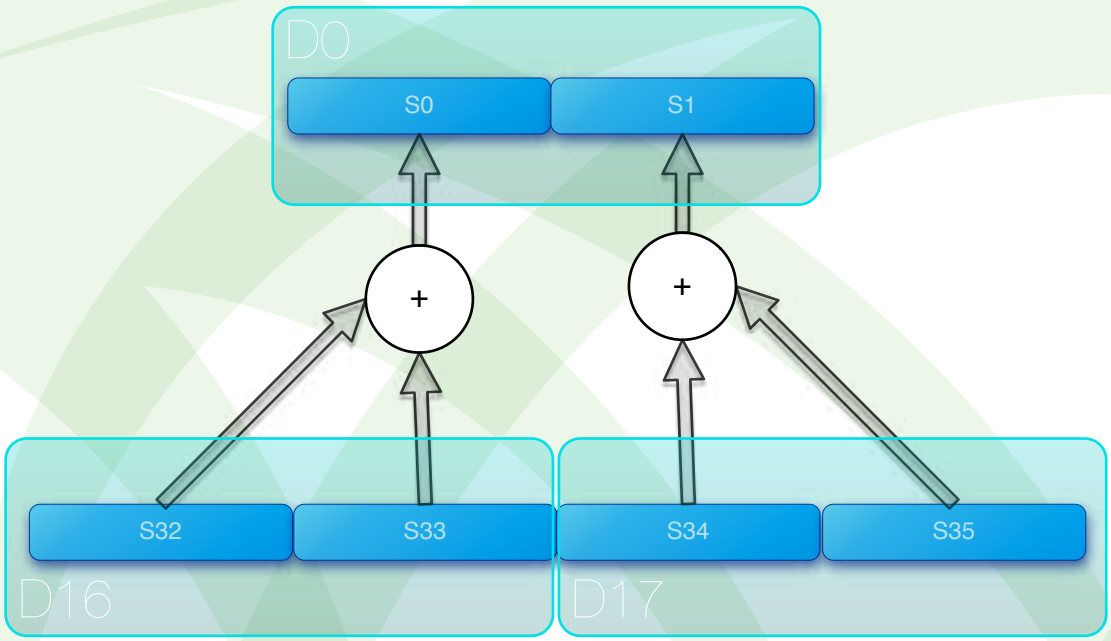
```
"vadd.f32  q8, q8, q9                    \n\t"
```

Vector Add adds corresponding elements in two vectors, and places the result in the destination vector. f32 (32-bit wide floating point "lane") is our datatype. The first argument (q8) is our destination. The second argument (in this case also q8) is our first operand. q9 is our second operand. This is a quadword operation, where corresponding 32-bit lanes of q8 and q9 are added together and then the result is placed in q8. Here is what q8 looks like after the add.

D16	a[0]*b[0] + a[4]*b[4] + a[8]*b[8] + a[12]*b[16] + ...	a[1]*b[1] + a[5]*b[5] + a[9]*b[9] + a[13]*b[13] + ...
D17	a[2]*b[2] + a[6]*b[6] + a[10]*b[10] + a[14]*b[14] + ...	a[3]*b[3] + a[7]*b[7] + a[11]*b[11] + a[19]*b[19] + ...

```
"vpadd.f32 d0, d16, d17                  \n\t"
```

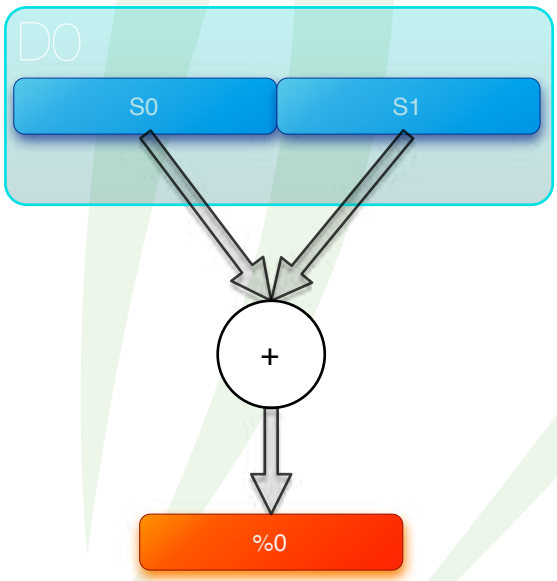
This is a pairwise add. It takes two vectors and adds the adjacent pairs of elements. d0 is the destination and d16 and d17 are the source. f32 is the datatype. It tells us that the elements to be added together are 32-bit floating point. d registers are 64-bit registers, so each d register has two lanes of data to be added together. Here is a diagram of what's going on.



This fills d0 with the a[0]b[0] + a[1]b[1] + a[4]b[4] + a[5]b[5] + ... series and fills d1 with the a[2]b[2] + a[3]b[3] + a[6]b[6] + a[7]b[7] + ... series of data.

```
"vadd.f32  %0, s0, s1                    \n\t"
```

d0 is composed of s0 and s1. Add s0 and s1 to bring the two series of data together in one continuous sum. The destination is the %0 register, which was setup with the compiler constraints in the colon section. The source registers are s0 and s1. Our datatype is 32-bit wide floating point lanes.



```
        : "=w"(s), "+r"(a), "+r"(b), "+r"(n)
        :: "q0", "q1", "q2", "q3", "q8", "q9");
```
Here is the end of our assembly code block, where the compiler constraints live.

```
    return s;
```
We return our result. s is the name of the register associated with operand %0. It's where we put our result in the final assembly code instruction.

```
}
#endif
```
The conditional block started by #ifdef U1 is concluded.

```
int main()
{
```
Here is where the function main begins. The `int` indicates that main returns an integer result. It appears to take no arguments, but interestingly enough, main is a special case because its arguments are provided by the runtime assembly language that comes with the compiler.

```
    volatile float a[256];
    volatile float b[256];
```
Volatile instructs the compiler to compile everything as written, and do not attempt to delete, move, or combine the instructions. Here we have two arrays, named a and b. They each have 256 elements of type float.

```
    float dp;
```
We set up variable dp as type float.

```
    size_t i;
```
we set up variable i as type size_t, which is an integer large enough to hold the SIZE of any data type. The "t" stands for type.

```
    for (i=0; i < 256; i++) {
```
This is a loop. i is set to zero and is post-incremented once per loop. When i reaches 256, then execution continues past the closing curly brace.

```
        a[i] = (rand() - RAND_MAX/2.0);
        b[i] = (rand() - RAND_MAX/2.0); }
```
These two instructions put random numbers into each of the elements of the arrays a and b

```
    for (i=0; i<1e6; i++) {
```
This is a loop. i is set to zero and is post-incremented once per loop. When i reaches ie6, then execution continues past the closing curly brace.

```
        dp = dotprod_fff_cortex_a8(a, b, 256); }
```
The function dotprod_fff_cortex_a8 is called. The addresses of the arrays a and b and the integer 256 are passed in as arguments. The result, when returned, is assigned to the variable

dp. a, n, 256, and dp all correspond to the operands in the compiler constraint section that is indicated by the colon formatting immediately following the block of assembly code we looked at earlier.

```
    printf("Result = %f\n", a[0]);
```

Present the result.

```
    return 0;
}
```

We return a value of 0 and the function main ends. It is traditional to return 0 if the program operated correctly. This concludes the code tutorial of the neon-test test suite.

Discussion

Here is where a short discussion needs to be placed. Some sort of summary about the broader application of the techniques, the place for NEON in signal processing, and the next step of our particular project, which is to add NEON support to FFTW.

In general, does neon-test have techniques that are commonly applied in SIMD coding? What are the commonly encountered SIMD coding techniques?

Why is NEON interesting?

The next step in our particular project is to add NEON support to a software coding project called Fastest Fourier Transform in the West. This library of routines allows one to implement fast fourier transform. (description, website, title of next document in the series.)

Optimized Tomfoolery
5371 Carmel Knolls Drive
San Diego, CA 92130

Company Name
123 Everywhere Avenue
City, ST 00000