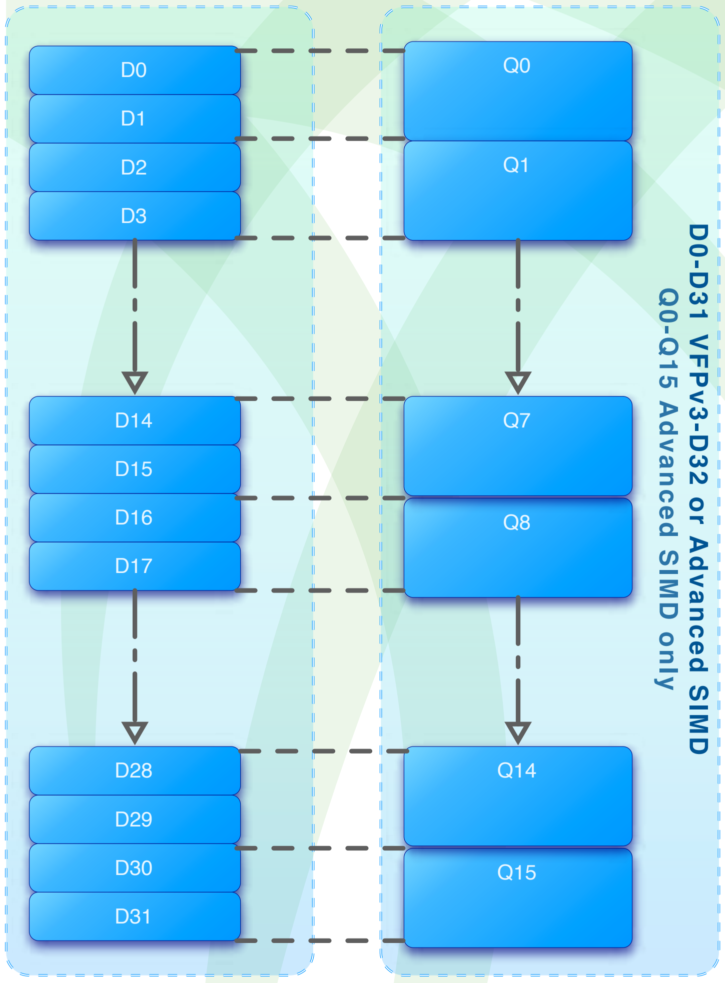
neon-test

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

NEON-TEST

The neon-test package contains two programs, complex_mult and dotproduct.

At the time this document was written, neon-test.tgz could be obtained from the following web page.

http://www.opensdr.com/node/13

Each program is outfitted with conditional compilation preprocessor directives that allow it to be built in one of two ways. One version of each program is a straightforward C implementation of the operation, and the other is a NEON implementation written as inline assembly language. By compiling each program both ways and comparing the results, you can verify correct operation of the NEON code and measure the speed-up achieved by switching from C to NEON.

One of these programs, complex_mult, is basically a non-vector operation. This example shows what sort of hoops one has to jump through in order to write efficient non-vector code using the NEON instruction set. The other program, dotproduct, is a very simple computation repeated on a long vector, so it shows the other end of the scale.

We will study each program line-by-line to become more familiar with the NEON extension.

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>
```

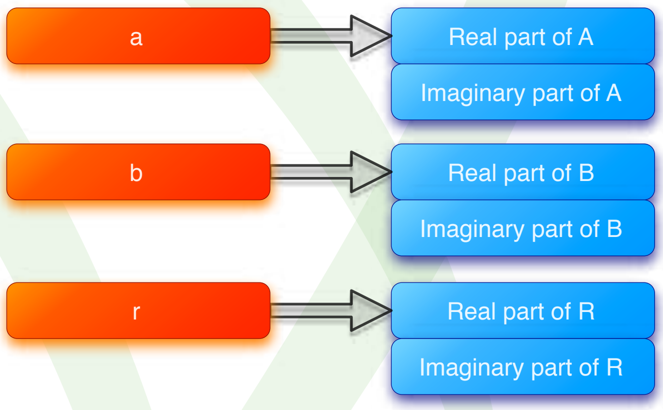
directive to include standard c library input and output header file

```
#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 values pointed to by *a and *b are designated as const, which means they do not change within this function. Let's call the complex numbers to be multiplied A and



B, and the resulting complex product R. Each of these complex numbers is represented by two floats, the real part and the imaginary part. *a dereferences to the first (real) part of A, *b dereferences to the first (real) part of B, and *r dereferences to the first (real) part of 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.

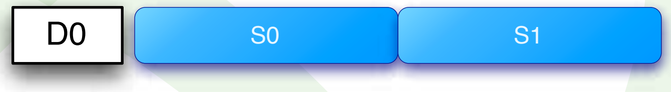
`"vld1.32   {d0}, [%1]        \n\t"`
Vector load multiple 1-element structures from our input operand [%1] to our NEON register d0. Every element of d0 is loaded.

This instruction moves data from memory to a register. The memory location is a base address and the register is completely filled.

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.

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 (and as many subsequent consecutive addresses as you need to load up the right size vector) in memory".

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:



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. S0 is loaded from [%1], and S1 is loaded from the next 32-bit memory location after [%1].

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 first operand. "r"(a) is the second operand. "r"(b) is the third operand.

Being able to find the address of these set-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 memory location set aside to contain the first operand, [%1] is the address of the memory location set aside to contain the second operand, and [%2] is the address of the memory location set aside to contain the third operand. Anytime we need the addresses of these operands, we use [%n], where n is the number of the operand, starting with 0.

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 for setup is at the end. It should be at the beginning so that the restrictions are in context. This would make reading the code easier."

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 values 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 d0, d1, d2, and d3 to store any other value, because the value will change. This is a safety lockout on the registers. Adding "memory" to the list requires the compiler to assume that any variable might be changed by the assembly code. The code generated by the compiler will need to reload variables from memory after the assembly code completes, rather than trying to reuse a value it might have previously loaded into a register.

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

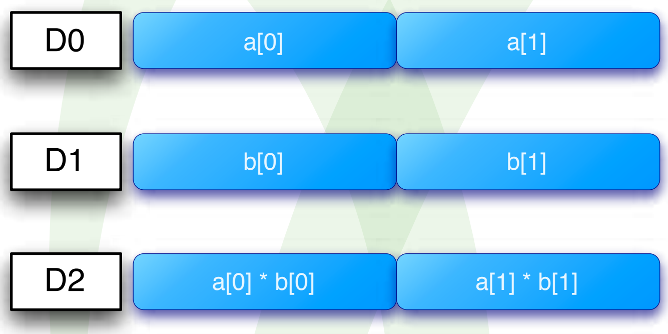
Now, let's skip back to the top of the block of assembly code and step through it.

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

Vector load multiple 1-element structures from our input operand [%2] (associated with a register by that colon magic) to our NEON register d1. Every element of d1 is loaded. This is just like the previous vld1.32 instruction.

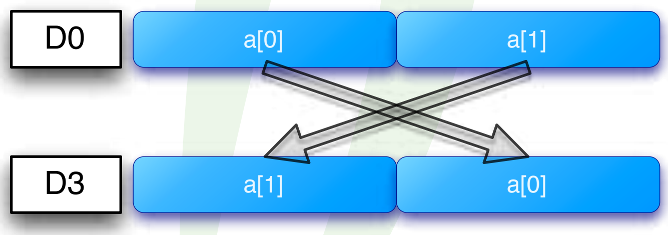
```
    "vmul.f32  d2, d0, d1             \n\t"
```

Here is our first example of computation on multiple data. 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 replaced by the other. As we will see shortly, this makes the complex multiplication easier.

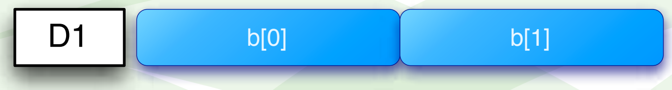


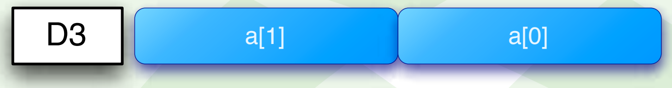
```
    "vmul.f32  d3, d3, d1             \n\t"
```

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 lanes of d1 and d3 are multiplied together and these results placed in d3.

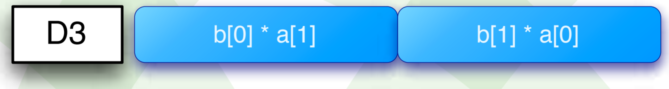
Below are the contents of d1 and d3 before the multiplication.

NEON-TEST



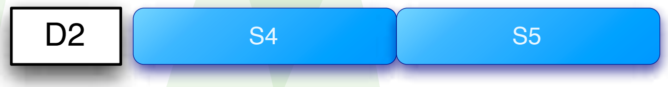


Below are the d3 register contents after multiplication.



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

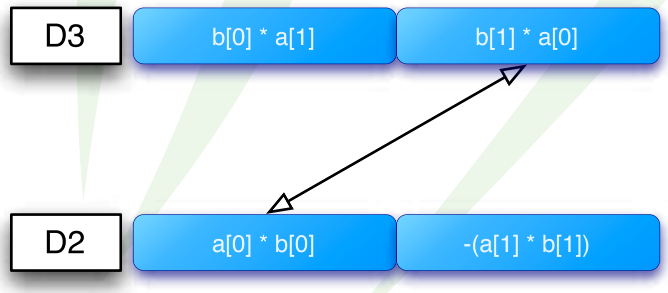
Next, we negate. This instruction takes s5, negates it, and puts it back in s5. On page 3 we showed that each 64-bit d register is composed of two 32-bit wide s registers. Here is where s5 is:



s5 is the latter half of d2, and is (a[1] * b[1]). What we actually need to do is subtract (a[1] * b[1]), which is equivalent to adding its negative. We also need to add another term, and we're always on the lookout for ways to parallelize operations like that. We can add two terms in parallel, so by negating this term we're setting up to do these two operations in parallel. This sort of roundabout method is perhaps typical of SIMD code.

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

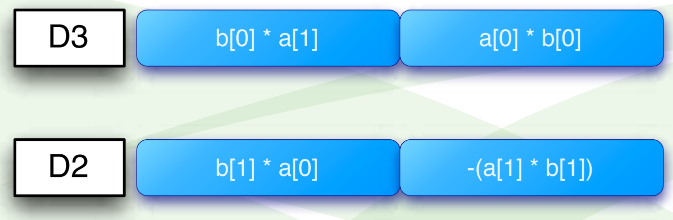
This instruction transposes a 2 by 2 matrix formed by d2 and d3, where the elements are 32-bits wide. Here is what's happening.



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

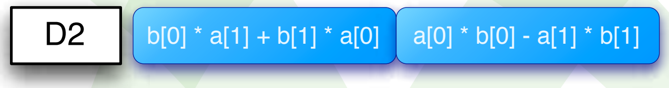
"SIMD computers require less hardware than MIMD computers because they have only one global control unit. Furthermore, SIMD computers require less memory because only one copy of the program needs to be stored. In contrast, MIMD computers store the program and operating system at each processor. However, the relative unpopularity of SIMD processors as general purpose compute engines can be attributed to their specialized hardware architectures, economic factors, design constraints, product life-cycle, and application characteristics."[1]

1 Grama, Ananth, Anshul Gupta, George Karypis, Vipin Kumar. Introduction to Parallel Computing. England: Addison-Wesley. 2003



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.



Here at last is the payoff for all the setup we've had to do. In one instruction we've effectively done an addition and a subtraction. In this case, since the operation is basically scalar, we don't get a huge payoff.

```
      "vst1.64   {d2}, [%0]              \n\t"
```
Store the contents of d2, a 64-bit register which contains our result, into memory. %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. 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];
```
We are setting aside memory here. The word "volatile" warns the compiler that these variables in

memory are liable to change in ways the compiler is not aware of. In this case, the in-line assembly language instructions are making changes to these variables. The compiler doesn't try to interpret the assembly. It's up to us, as programmers, to make sure the compiler is aware of any special needs the data might have. Volatile means that the compiler must generate code that does not try to optimize by holding these values in registers. Since the values may change in memory, the values in registers would be wrong. Any case where a variable serves as an interface between in-line assembly and c is an excellent candidate for being declared as volatile. Worst case, some optimization is lost. Best case, wrong code becomes right code. In this case, there is no particular reason for this volatile keyword. This is a very conservative example of usage.

The rule of thumb is to use volatile whenever it's important that the compiler not get fancy with memory.

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 assigns the (integer) value 1 to the (float) variable a[0], which involves an implicit type conversion. 1, 2, 3, and 4 are arbitrary test data to exercise the code.

It would be more explicit to say `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 incremented once per loop. When i reaches 1e8 (1 times 10 raised to the 8th power, or 100,000,000), 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 speed-up 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 compile the following conditional block.

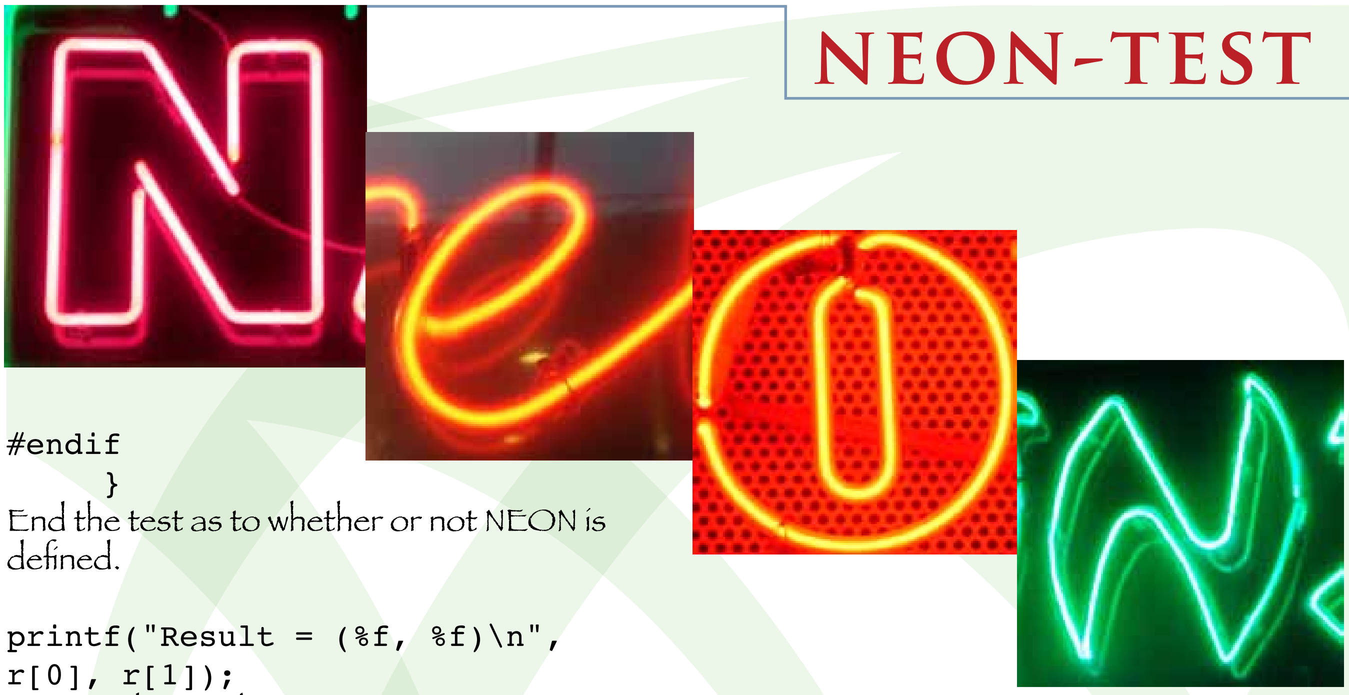
```
complex_mult(a, b, r);
```
Call the complex_mult function with the arguments a, b, and r.

```
#else
```
if NEON is not defined, then compile 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.

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



```
#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.

"Effective use of SIMD instructions still largely relies on hand coding in assembly language."[1]

1 Ericson, Christopher. Real-Time Collision Detection. San Francisco: Morgan Kaufman, 2005

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 U0 is defined, then execute the following conditional block. Remember that in complex_mult.c, we had an #ifdef NEON. NEON in the previous code and U0 here (and, later on in this file, U1) all serve the same purpose. They control which parts of the program are compiled. The difference is arbitrary. Different authors will use different #ifdef macros.

```
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.

This is not the best name for a function as written under #ifdef U0. It's a very reasonable name for the function as written for NEON (under #ifdef U1), but the plain C code in the U0 implementation has nothing to do with the Cortex A8. Probable, the programmer wrote the U1 version first, and then had to use the same name for the U0 version to avoid having a confusing extra #ifdef in the main() function.

The values pointed to by *a and *b are designated as const, which means they do not change within this function. Let's call the vectors to be "dot productized" A and B. *a dereferences to the first element of A, *b dereferences to the first element of 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 literal 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 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 U1 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. This is exactly the same function definition as in the plain C version, above.

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, for the same purpose as before: to control termination of the loop. The value for n is passed in from main as the literal 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 (GNU compiler collection[3]) 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 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 in the input section.

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"
```
Fill up the quadword register q8 with the 32-bit floating point value 0.0. The constant specified by # is replicated to fill the NEON register.

```
          "vmov.f32   q9, #0.0                         \n\t"
```
We're doing what we did to q8, but now doing it for q9. Fill up the quadword register q9 with the 32-bit floating point value 0.0. The constant specified by # is replicated to fill the NEON register.

We now have eight separate 32-bit registers zeroed out and ready to serve as parallel accumulators.

```
          "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.

We decrement by 8 because we can do 8 pairs of input values 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.

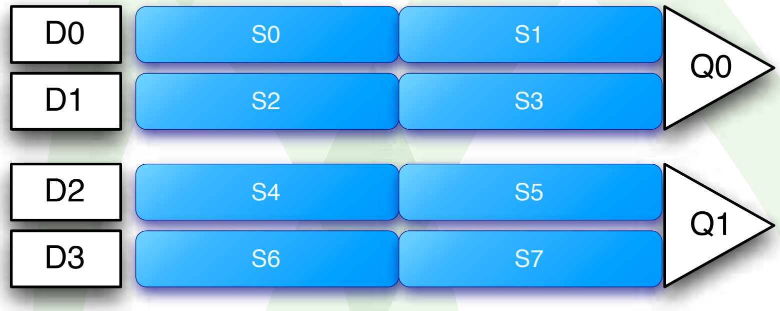
```
          "vld1.32    {d0,d1,d2,d3}, [%1]!         \n\t"
```
So what does this do? Vector load multiple 1-element structures. Here, n = 1. [%1] points to a

3 http://www.gnu.org/

1-element structure that is 32 bits wide. We copy values out of memory into the NEON registers listed as {d0,d1,d2,d3}. This destination is presented as a "register list". We start loading from the address currently in the register %1 and increment the address for each 32-bit value loaded. This results in a total of eight 32-bit floats loaded from memory.

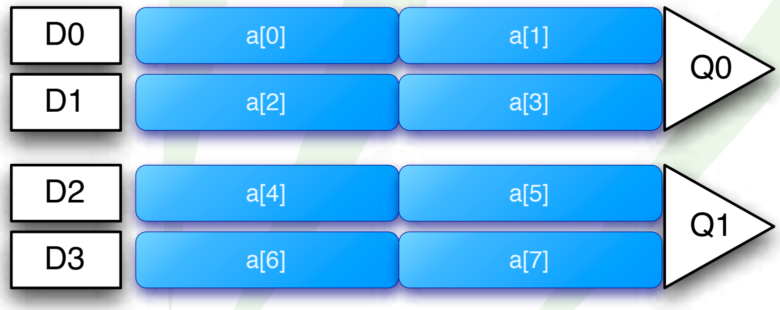
The ! means that %1 is updated to (Rn plus the number of bytes transferred by the instructions) after all the loads and stores.

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.



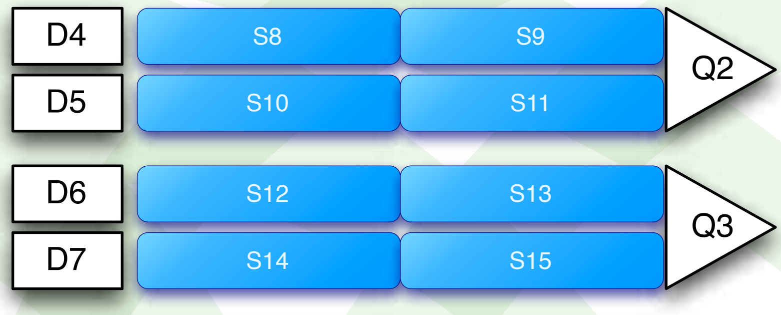
[%1] is the base address of the array of 256 elements that we are going to dot product with a second 256-element array. After we accomplish the vector load, we update [%1] by the number of bytes transferred by the instruction. Each element is 32 bits wide. We are using 8-bit bytes, so each 32-bit load 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 on the first time through the loop. 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.

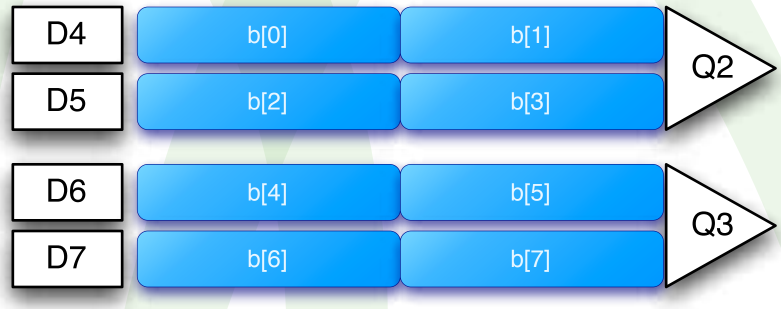


```
        "vld1.32   {d4,d5,d6,d7}, [%2]!       \n\t"
```
Here we do the same type of vector load for d4, d5, d6, and d7. We copy values out of memory into the NEON registers listed started at the address currently in the register %2. This is a sec-

ond 256-element array that will be dot-productized with the first 256-element array.



Here is what the registers look like after the load on the first time through the loop. 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.



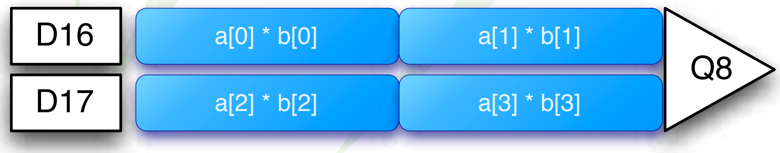
```
          "vmla.f32   q8, q0, q2                \n\t"
```

q8 and q9 were zeroed out earlier by a `"vmov.f32   q9, #0.0"` and `"vmov.f32   q9, #0.0"` instructions. This instruction does four multiply and accumulates in parallel. q8 is the destination. q0 is the 1st operand and q2 is the second operand. q8 += q0*q2 in 32-bit wide lanes.

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)

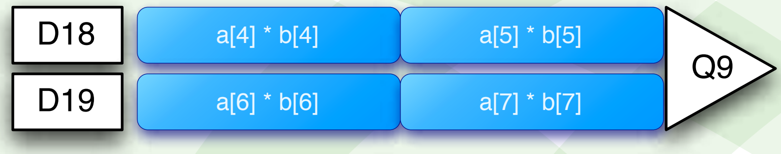
Answer: loads don't care about the datatype. It's just bits when you load. Multiply, however, needs to know what kind of number it's dealing with.

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. q9 += q1*q3.



```
        "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.

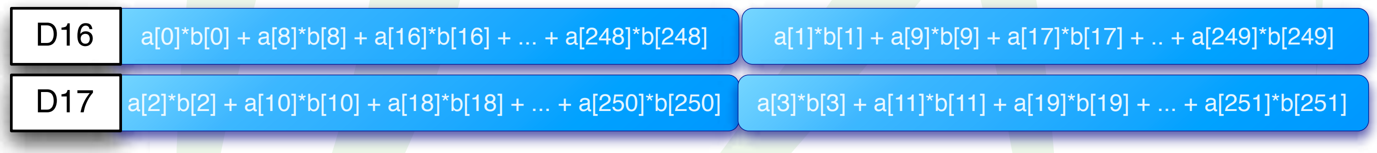
Here is what's going on under the hood with the condition code flags affected by the subtraction.

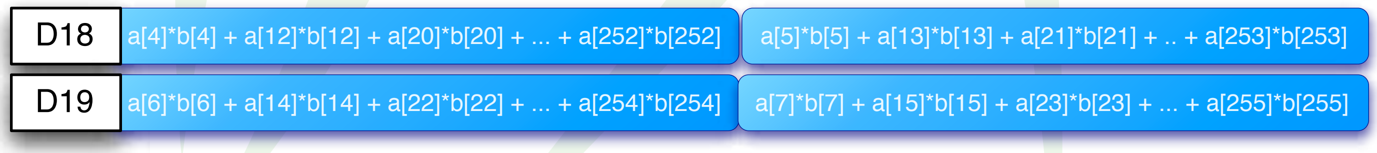
```
(if (not (or zbit (xor nbit vbit)))
  (set program counter to address indicated by label))[4]
```

The `1b` means to look back for label 1. B stands for back. 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, as indicated by the calculation with condition flags (listed above).

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.

q 8 and q9 look something like this:


D16
D17
a[0]*b[0] + a[8]*b[8] + a[16]*b[16] + ... + a[248]*b[248]
a[1]*b[1] + a[9]*b[9] + a[17]*b[17] + .. + a[249]*b[249]
a[2]*b[2] + a[10]*b[10] + a[18]*b[18] + ... + a[250]*b[250]
a[3]*b[3] + a[11]*b[11] + a[19]*b[19] + ... + a[251]*b[251]


D18
D19
a[4]*b[4] + a[12]*b[12] + a[20]*b[20] + ... + a[252]*b[252]
a[5]*b[5] + a[13]*b[13] + a[21]*b[21] + .. + a[253]*b[253]
a[6]*b[6] + a[14]*b[14] + a[22]*b[22] + ... + a[254]*b[254]
a[7]*b[7] + a[15]*b[15] + a[23]*b[23] + ... + a[255]*b[255]

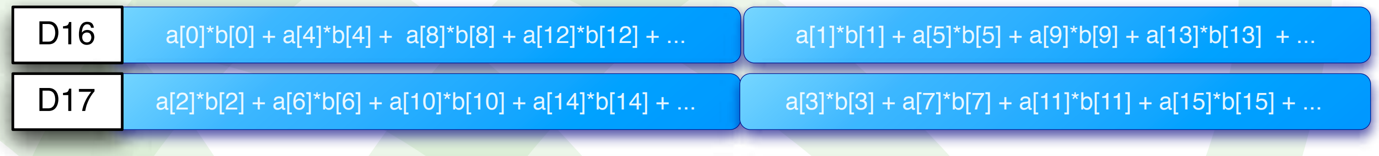
We have eight partial sums from the dot product. Now we need to combine them.

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

4 referenced from http://sourceware.org/cgen/gen-doc/arm-thumb-insn.html#insn-bgt

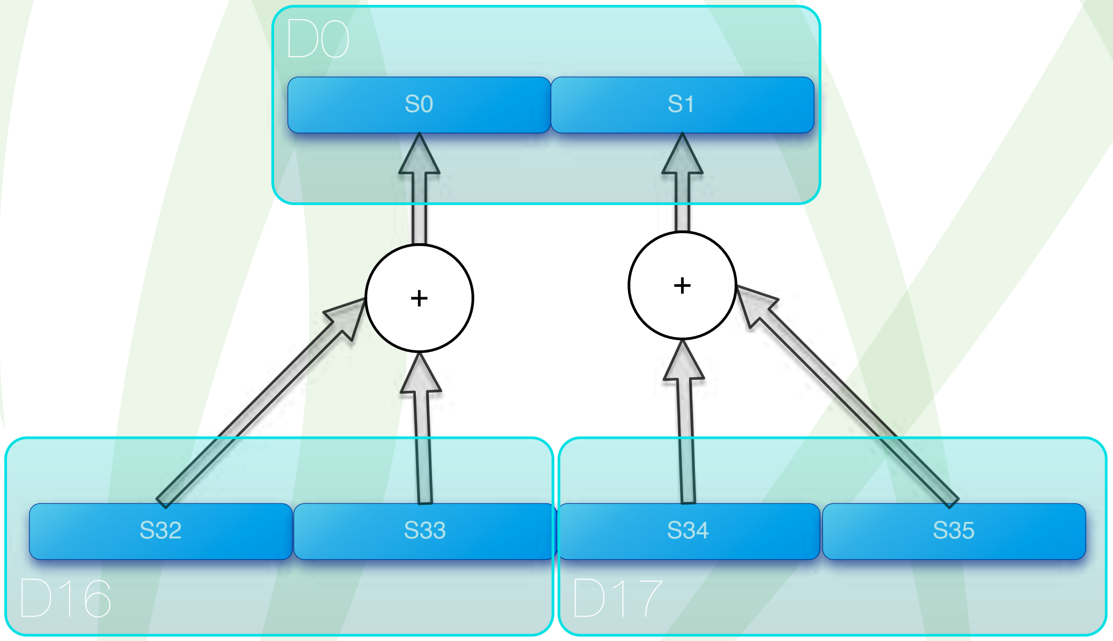
NEON-TEST

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[12] + ...
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[15]*b[15] + ...

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

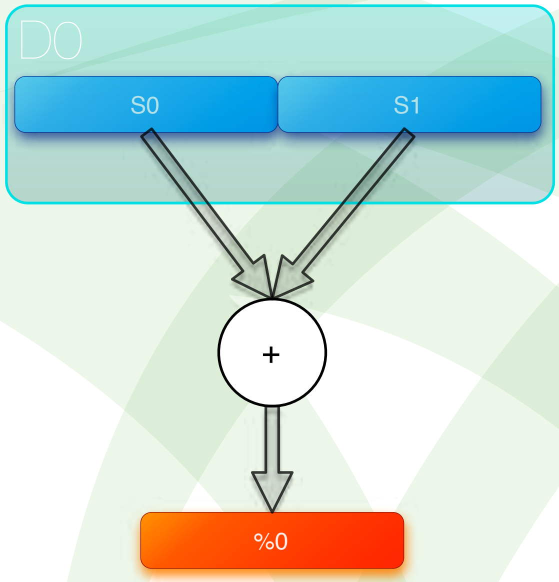
This is a pairwise add. It takes two vectors and adds the adjacent pairs of elements. d0 is the destination and d16 and d17 are the source. f32 is the datatype. It tells us that the elements to be added together are 32-bit floating point. d registers are 64-bit registers, so each d register has two lanes 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 1e6, then execution continues past the closing curly brace.

```
        dp = dotprod_fff_cor-
tex_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. For the third argument, not just any value will do. 256 works because it is a multiple of 8. What happens if you pass in a number that isn't a multiple of 8? You access memory you shouldn't accessing, and get the wrong answer.

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]);
```
Print one of the random inputs.

```
    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. Next, let's look at the makefile and see how to run and measure execution time of the code we create.

MAKEFILE

A makefile provides a set of recipes for common operations in a project[1]. Here is a discussion of the makefile for neon-test.

```
CFLAGS=-march=armv7-a -mtune=cortex-a8 -mfpu=neon -mfloat-abi=softfp
```

We set up a local variable called CFLAGS, which is short for c-compiler flags. This variable is invoked in the make file commands.[2]

-march=armv7-a	This specifies the name of the target ARM architecture.
-mtune=cortex-a8	This option is very similar to the -mcpu= option, which specifies the target processor type. However, instead of strictly setting the processor type, and hence restricting which instructions can be used, it specifies that GCC should tune the performance of the code as if the target were of the type specified in this option, but still choosing the instructions that it will generate based on the cpu specified by a -mcpu= option. For some ARM implementations better performance can be obtained by using this option.
-mfpu=neon	This specifies what floating point hardware (or hardware emulation) is available on the target.
-mfloat-abi=softfp	Specifies which floating-point ABI to use. Permissible values are: `soft', `softfp' and `hard'. Specifying `soft' causes GCC to generate output containing library calls for floating-point operations. `softfp' allows the generation of code using hardware floating-point instructions, but still uses the soft-float calling conventions. `hard' allows generation of floating-point instructions and uses FPU-specific calling conventions. The default depends on the specific target configuration. Note that the hard-float and soft-float ABIs are not link-compatible; you must compile your entire program with the same ABI, and link with a compatible set of libraries.

```
all : complex_mult_neon complex_mult dotproduct dotproduct_neon
```

The colon separates the target from the dependency. By convention, "all" is a target that makes all of the different parts of the project. When you type "make all", then this recipe is run. The list that follows the colon is the list of files that all depends on. Everything in the list following the colon has to be made fresh in order for all to be made fresh. "all" isn't constructed. There are no commands listed below the target and dependencies. "all" is something known as a pseudo target. Because all is the first thing listed, then if you just type "make", then you get "make all".

```
run :
	echo Benchmark complex multiplication
```

1 reference http://www.gnu.org/software/make/manual/

2 CFLAGS options referenced from http://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html

```
	time ./complex_mult
	time ./complex_mult_neon
	echo Benchmark dot product
	time ./dotproduct
	time ./dotproduct_neon
```

This is another example of a pseudo target. It's called run. It has no dependencies. We can tell that it has no dependencies because there is nothing listed after the colon. The commands are the six lines following the first line. They are indented by a tab.

"echo" prints the string following it to (usually) the screen for you to read.
"time ./complex_mult" measures how long complex_mult takes to run and displays the duration on the screen for you to read.
"time ./complex_mult_neon" measures how long complex_mult_neon takes to run and displays the duration on the screen for you to read.
"echo" prints the string following it to (usually) the screen for you to read.
"time ./dotproduct" measures how long dotproduct takes to run and displays the duration on the screen for you to read.
"time ./dotproduct_neon" measures how long dotproduct_neon takes to run and displays the duration on the screen for you to read.

```
clean :
	rm -f complex_mult_neon complex_mult dot_product dot_product_neon
```

This is another example of a pseudo target. It's called clean, and has no dependencies. If you type "make clean", then it removes (with the long arm of the law) the files complex_mult_neon, complex_mult, dot_product, and dot_product_neon. Namely, the ones built by this makefile. Make clean is useful when things get bejiggered, or before distribution.

```
complex_mult_neon : complex_mult.c
	gcc $(CFLAGS) -DNEON -o $@ $<
```

If complex_mult_neon.c is newer than complex_mult_neon, then make complex_mult_neon using the command listed.

```
complex_mult : complex_mult.c
	gcc $(CFLAGS) -o $@ $<
```

If complex_mult.c is newer than complex_mult, then make complex_mult using the command listed.

```
dotproduct : dotproduct.c
	gcc $(CFLAGS) -DU0 -o $@ $<
```

If dotproduct.c is newer than dotproduct, then make dotproduct using the command listed.

```
dotproduct_neon : dotproduct.c
```

```
    gcc $(CFLAGS) -DU1 -o $@ $<
```

If dotproduct_neon.c is newer than dotproduct_neon, then make dotproduct_neon using the command listed.

complex_mult_neon and complex_mult are two different programs. They are created from the same source code file. The different programs are made when different commands are used. Complex_mult_neon is created when "gcc $(CFLAGS) -DNEON -o $@ $<" is executed.

gcc is the compiler. $(CFLAGS) is a variable that was defined at the top of the makefile.

-DNEON is passed directly to the compiler and defines a symbol called NEON. The -D option sets pre-processor variables. This is done so that all code that is in the specified "#ifdef NEON #endif" blocks will be compiled.

-o means create an output file (the executable). $@ is a special makefile syntax. It gets replaced with the name of the target, which in this case is complex_mult_neon. The target is before the colon. This is an argument to the -o flag. $< is a special makefile syntax. It gets replaced with the name of the source code, which is listed as the dependency (after the colon).

If you were typing this like a human would, then it would translate to the following command.

gcc -march=armv7-a -mtune=cortex-a8 -mfpu=neon -mfloat-abi=softfp -DNEON -o complex_mult_neon complex_mult.c

MAKE AND RUN

Here is an example of making and running the four executables.

```
abraxas3d@beagleboard:~$ pwd
/home/abraxas3d

abraxas3d@beagleboard:~$ ls
neon-test.tgz

abraxas3d@beagleboard:~$ tar xvf neon-test.tgz
./neon-test/
./neon-test/Makefile
./neon-test/complex_mult.c
./neon-test/dotproduct.c

abraxas3d@beagleboard:~$ cd neon-test/

abraxas3d@beagleboard:~/neon-test$ ls
```

```
Makefile  complex_mult.c  dotproduct.c

abraxas3d@beagleboard:~/neon-test$ make all
gcc -march=armv7-a -mtune=cortex-a8 -mfpu=neon -mfloat-abi=softfp
-DNEON -o complex_mult_neon complex_mult.c
complex_mult.c: In function 'main':
complex_mult.c:55: warning: passing argument 1 of 'complex_mult'
discards qualifiers from pointer target type
complex_mult.c:55: warning: passing argument 2 of 'complex_mult'
discards qualifiers from pointer target type
complex_mult.c:55: warning: passing argument 3 of 'complex_mult'
discards qualifiers from pointer target type
gcc -march=armv7-a -mtune=cortex-a8 -mfpu=neon -mfloat-abi=softfp  -o
complex_mult complex_mult.c
gcc -march=armv7-a -mtune=cortex-a8 -mfpu=neon -mfloat-abi=softfp  -DU0
-o dotproduct dotproduct.c
dotproduct.c: In function 'main':
dotproduct.c:66: warning: passing argument 1 of 'dotprod_fff_cortex_a8'
discards qualifiers from pointer target type
dotproduct.c:66: warning: passing argument 2 of 'dotprod_fff_cortex_a8'
discards qualifiers from pointer target type
gcc -march=armv7-a -mtune=cortex-a8 -mfpu=neon -mfloat-abi=softfp  -DU1
-o dotproduct_neon dotproduct.c
dotproduct.c: In function 'main':
dotproduct.c:66: warning: passing argument 1 of 'dotprod_fff_cortex_a8'
discards qualifiers from pointer target type
dotproduct.c:66: warning: passing argument 2 of 'dotprod_fff_cortex_a8'
discards qualifiers from pointer target type

abraxas3d@beagleboard:~/neon-test$ ls
Makefile      complex_mult.c    dotproduct    dotproduct_neon
complex_mult  complex_mult_neon  dotproduct.c

abraxas3d@beagleboard:~/neon-test$ time ./complex_mult
Result = (-5.000000, 10.000000)
real 0m 31.02s
user 0m 30.30s
sys  0m 0.00s

abraxas3d@beagleboard:~/neon-test$ time ./complex_mult_neon
Result = (-5.000000, 10.000000)
real 0m 17.98s
user 0m 17.61s
sys  0m 0.00s
abraxas3d@beagleboard:~/neon-test$

abraxas3d@beagleboard:~/neon-test$ time ./dotproduct
```

```
Result = 730547584.000000
real 0m 43.16s
user 0m 42.10s
sys  0m 0.02s
abraxas3d@beagleboard:~/neon-test$

abraxas3d@beagleboard:~/neon-test$ time ./dotproduct_neon
Result = 730547584.000000
real 0m 1.23s
user 0m 1.17s
sys  0m 0.00s
```

RESULTS OF THE OPTIMIZATION

Here is a table showing the speed improvement in the execution time between the optimized (complex_mult_neon, dotproduct_neon) and non-optimized (complex_mult, dotproduct) code.

Executable	Execution Time
complex_mult	30.30 seconds
complex_mult_neon	17.61 seconds
dotproduct	42.10 seconds
dotproduct_neon	1.17 seconds

NEON-TEST CONCLUSION

NEON is intended to enable signal and media processing. Since these are very important functions in amateur radio, NEON is a coprocessor that amateur radio enthusiasts need to know more about.

The next step in our particular project (MEP) is to add NEON support to a software coding project called Fastest Fourier Transform in the West (FFTW).

This library of routines allows one to implement the fast fourier transform. Read more about FFTW at their website:

http://www.fftw.org/

Find out more about our project, the Microwave Engineering Project (MEP), at our website:

http://www.delmarnorth.com/microwave

Optimized Tomfoolery
5371 Carmel Knolls Drive
San Diego, CA 92130

Company Name
123 Everywhere Avenue
City, ST 00000