## CPE 442 : Neon intrinsics

## Author(s)


- Victor Delaplaine
- Tristan Chutka


## Video link(s)

- Video demonstrating the sobel program with Neon intrinsics
- Video demonstrating the sobel program with Neon intrinsics vs without

## Neon Extension

The Neon engine that exits in some new ARM processors allow you to perform Single Instruction Multiple Data(SIMD). These SIMD instructions are just an extension of ARM instructions. They just use their own unit(ALU and Registerfile) known as the Neon Unit which is integrated in the processor and share resources of the main unit (CPU). These SIMD instructions are implemented on the Neon Unit and can do vector operations: add two vectors in parallel, multiply, and other operations. The Neon unit consits of Single word(S), Double word(D), and Quad word(Q) registers, these registers will determine the length of our vectors. Each vector has lanes associated with them. These are the number of elements stored in each vector.

For our case in this class our maxium width vector can be 128 bits. The register will hold a maximum of 128-bits is known as a Q register, a D register holds 64 bits, and finally S registers hold 32bits. S registers can only be used by the VFP(vector floating point) Unit.

- 64-bit NEON vectors(D Registers) can contain:
  - Eight 8-bit elements.
  - Four 16-bit elements.
  - Two 32-bit elements.
  - One 64-bit element.
- 128-bit NEON vectors(Q Registers) can contain:
  - Sixteen 8-bit elements.
  - Eight 16-bit elements.
  - Four 32-bit elements.
  - Two 64-bit elements.

## Neon Intrinsics (ARM-7)

- Neon intrinsics allow you to replace ordinary instructions with Neon instructions by the compiler.
- You must include the `<arm_neon.h>` header in your code to use the following neon intrinsics

**Neon Intrinsic Data Types**

- Neon vector datatypes are named in according to the pattern: `<type><size>x<number_of_lanes>[x<number_of_vectors>]_t`
  - For our case `<type>` look at `Table 1.1` below
  - For `<size>` that would be the associated number of bits for a given type
  - For `<num_of_lanes>` the max number of lanes is followed by the equation: `total_bits/(sizeof(<type>)*[<number_of_vectors>])`
    - Where total_bits are determined by either Q(128-bits) registers or D(64-bits) registers
  - Only if `<number_of_vectors>` is greater than one fill in the number, this option is if you want an array of vectors

| | 8-bit | 16-bit | 32-bit | 64-bit |
|---|---|---|---|---|
| Unsigned integer | U8 | U16 | U32 | U64 |
| Signed integer | S8 | S16 | S32 | S64 |
| Integer of unspecified type | I8 | I16 | I32 | I64 |
| Floating-point number | DNE | F16 | F32/F | DNE |
| Polynomial over {0,1} | P8 | P16 | DNE | DNE |

[ Table 1.1: this shows the various datatypes you can use ]

**Neon Variables and constants in use**

- `uint32x2_t vec64a, vec64b` // Create two D-reg variables
- `uint8x8_t start_value = vdup_n_u8(0)` // To replicate a constant into each element of a vector

**Neon Function Prototypes**

- Neon Intrinic function prototypes have the following form: `<operation_name><flags>_<type>`
  - The `<operation_name>` what operation is being used
    - Example `vld1_u8`: loads a uint8 pointer into a 8x8 vector
    - Example `vmull_u8`: mulitplys two uint8x8 into a longer int with 16 bytes and 8 channels
  - The `<flags>` determine certain things about the operations such as:
    - Example: uin8x8_t `vadd_u8(uint8x8_t a, uint8x8_t b)`
      - The input and outputs are using D registers
    - Example: uint8x8_t `vaddq_u8(uint8x16_t a, uint8x16_t b)`
      - The inputs and outputs are using Q registers `flag=q`
  - `<type>` determines what type the inputs are.

**Neon Functions in use**

- `result = vget_lane_u32(vec64a, 0)` // this will get lane 0 of the D reg
- `byteval = vreintrpret_u8_u32(wordval)` // cast 32bit datatype to 8bit datatype vector
- `v=vld1_u16(A)` // load an array (elemnts uint16 to a D vector)
  - Note: When loading in arrays dont overlap in memory, to reiterate this to the compiler denote your destination as variable ___restrict

**Neon Instrinsics using inline assembly**

- There are different ways to achieve the above operations, this other way is called inline assembly.
  - Instruction for Neon asembly can be found here
- Neon assembly instruction example: Add two Q registers with 16-bit elements:
  - `VADD.I16 q0, q1, q2` // q0 is the result register and q1 and q2 are the inputs

**Inline assembly using in c/c++ code**

- gcc/g++ offers a way to execute assembly code in c/c++ code.
- To tell the compiler your going to write assembly code, you have to use keyword `__asm__`
  - For example the below function will add 100 to the parmaeter passed in and then return it.

```
__asm__( /* Assembly function body */
"my_fn:\n"
    "  mov %rdi,%rax\n"
    "  add $100,%rax\n"
    "  ret\n"
);
int main(){
    return my_fn(3);
}
```

## Flags Needed!

- `-mfpu=neon`
  - Tells gcc/g++ we are working with neon instruction in ARM
- `-mcpu=Cortex-A8`
  - Tells gcc/g++ what specific architecture we are working on
- `-ftree-vectorize`
  - Explicitly tells the compliler to try to vectorize
  - Usually if you want to use auto-vectorization
- `-O3`
  - Tells gcc/g++ to vectorize or optimize

– Usually if you want to use auto-vectorization