Prototyping SIMD128 in V8

Haitao Feng
haitao.feng@intel.com

Sep 2013

Contents

1 Introduction

In this document, we would discuss prototyping SIMD128 in V8.

1.1 The straw man proposal

The 128-bit SIMD numeric value_type ECMAScript straw man proposal from John Mccutchan is at https://github.com/johnmccutchan/ecmascript_simd.

2 Implementation

2.1 V8 Execution Flow

The following diagram describes how V8 engine executes a JavaScript function.

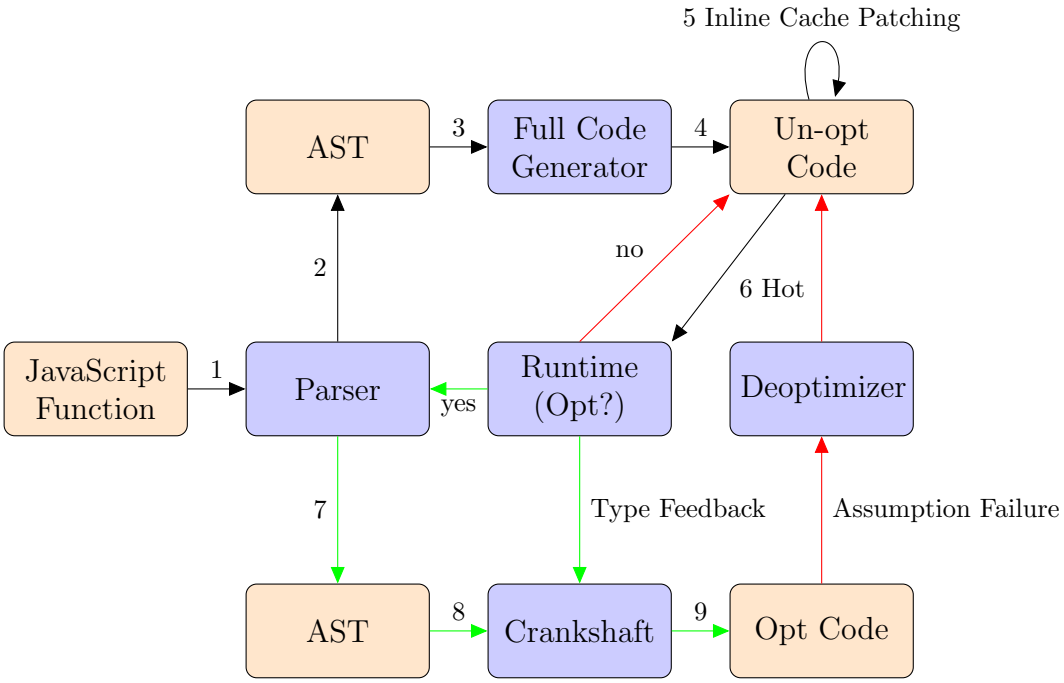

Figure 1: V8 Execution Flow

There are two compilers inside V8: the FullCodeGenerator and the Crankshaft. In the Crankshaft, there is a high-level IR called Hydrogen and a low-level IR called Lithium. Basically we need to implement SIMD128 for both compilers. If you are not familiar with V8 implementation, please read Andy Wingo's blogs first.

- v8: a tale of two compilers
- inside full-codegen, v8's baseline compiler
- value representation in javascript implementations
- a closer look at crankshaft, v8's optimizing compiler
- from ssa to native code: v8's lithium language
- on-stack replacement in v8

2.2 Implementing SIMD128 in the FullCodeGenerator

To implement SIMD128 proposal in the FullCodeGenerator, we need to introduce the float32x4 and uint32x4 primitive type and float32x4 and uint32x4 objects first and then introduce the float32x4 and uint32x4 constructors, next we need to introduce SIMD object into JavaScript global object and finally implement the SIMD operations.

2.2.1 Introducing float32x4 and uint32x4 primitive type and objects

From reading the straw man proposal and its JavaScript implementation, it seems that the float32x4 and uint32x4 are quite like the existing JavaScript Number concept. The Number primitive type is defined at ECMA262 8.5 and the Number Objects is at ECMA262 15.7. Number objects have the Object type defined at ECMA262 8.6.

In the FullCodeGenerator, a value from Number primitive type could be represented as SMI (SMall Integer) or HeapNumber. Both of them are tagged value, please read "value representation in javascript implementations" for more information on tagging. The HeapNumber's object layout and Number object layout are like this:

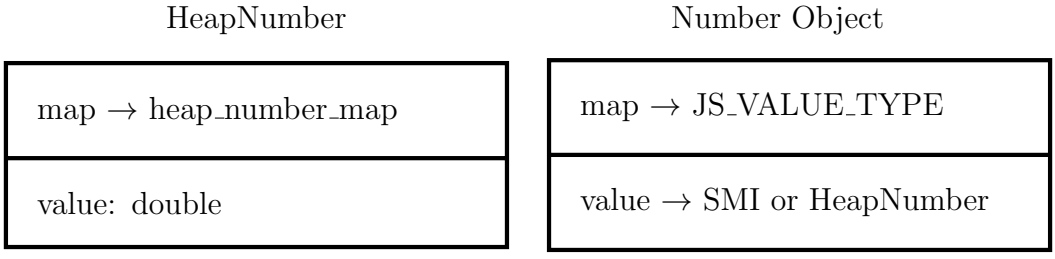

Figure 2: HeapNumber Layout and Number Object Layout

We will introduce float32x4 and uint32x4 primitive type and float32x4 and uint32x4 objects. When developers use the float32x4 and uint32x4 constructor in a new expression, we will create a float32x4 and uint32x4 object and this object wraps a float32x4 and uint32x4 value of the primitive type, like the Number object does. This is quite consistent with the current ECMA262 standard.

In the remaining document, we use the float32x4 object and uint32x4 object to mean a value of the float32x4 and uint32x4 primitive type, instead of the float32x4 and uint32x4 Object type.

2.2.2 Introducing float32x4 and uint32x4 constructors

We will introduce the float32x4 and uint32x4 JavaScript constructor in the JavaScript global object so that developers could write:

```
var f4 = float32x4(1.0, 2.0, 3.0, 4.0);
var u4 = uint32x4(1, 2, 3, 4);
var new_f4 = new float32x4(1.0, 2.0, 3.0, 4.0);
var new_u4 = new uint32x4(1, 2, 3, 4);
...
```

In the FullCodeGenerator, we will introduce Float32x4Object and Uint32x4Object classes for the float32x4 and uint32x4 objects and the object layouts in the heap are like this:

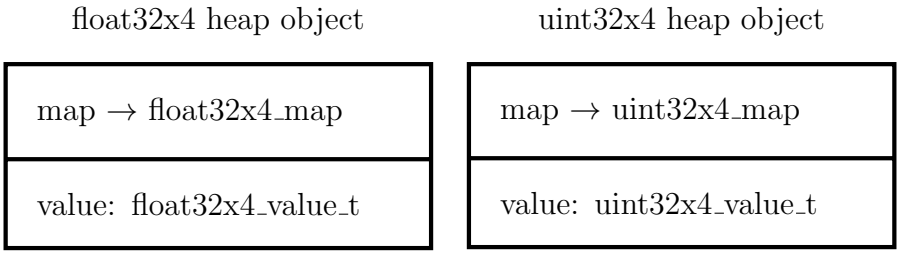

Figure 3: float32x4 and uint32x4 Object Layout

Each heap object has a map field with a pointer size (4 bytes on 32-bit system and 8 bytes on 64-bit system) pointing to its map (the hidden class). We will create maps for float32x4 and uint32x4. The float32x4_value_t and uint32x4_value_t are defined as:

```
struct float32x4_value_t { float storage[4]; };
struct uint32x4_value_t { uint32_t storage[4]; };
```

The float32x4 and uint32xt values are represented in this way as we hope most of the time we could use SIMD instruction on them. For float32x4, the instructions might include movaps, movups, addps, subps, mulps, divpsi, sqrtps, rcpps, shufps and rsqrtps. For uint32x4, the instructions might include movaps, movups, paddd, psubd, andps, orps, notps and xorps. In the Float32x4Object and Uint32x4Object classes, we provide methods to access each lane of the float32x4 and uint32x4 objects.

We might avoid using movaps (address aligned at 16 byte boundary) instruction first as it might need extra padding in the heap. We could implement the extra padding feature later to see the performance improvement. Currently for the IA32 architecture the HeapNumber (12 bytes) containing a double value in the heap is aligned by padding.

2.2.3 Introducing SIMD object

We will introduce the SIMD object into the JavaScript global object so that developers could write:

```
var neg_f4 = SIMD.neg(f4);
var abs_f4 = SIMD.abs(f4);
var lessthan_u4 = SIMD.lessThanOrEqual(neg_f4, abs_f4);
...
```

The SIMD object is quite like the Math object defined in ECMA262 15.8. We have two ways to implement the SIMD128 operations in the FullCodeGenerator:

1. Invoking C++ runtime by using the %SIMD_operation syntax in the JavaScript library

2. Generating machine codes by using the %_SIMD_operation syntax in the JavaScript library

In this step, we will use C++ runtime functions to implement the SIMD128 operations so that both FullCodeGenerator and Crankshaft works. Using the %_SIMD_operation could avoid the runtime call overhead but it requires the Crankshaft implementation also. We will revisit this after we implement SIMD128 in the Crankshaft.

An important implementation issue is whether we need to check operation overflow for SIMD operations on float32x4 and uint32x4 objects. In this prototype, we omit the overflow handling as we do not know what we need to do when it happens. It seems that Dart does not do this from reading the source codes.

2.2.4 Introducing Float32x4Array and Uint32x4Array (optional)

It might make sense to introduce Float32x4Array and Uint32x4Array typed array in the JavaScript. If we use the current float32x4 constructor for a Float32Array object, we have to construct a float32x4 object by float32x4(float32Array[4*i], float32Array[4*i+1], float32Array[4*i+2], float32Array[4*i+3]) and there are unnecessary float to double (from Float32Array element to HeapNumber or XMM double register) and double to float (from HeapNumber or XMM double register to float32x4 heap object or XMM register) conversion in the generated code. As they are not in the proposal, we might or might not implement them in this prototype.

They are quite like the existing Float32Array and Uint32Array, to implement them, we need to add Float32x4Array and Uint32x4Array into typedarray.js and implement the related functions.

2.3 Implementing SIMD128 in the Crankshaft

In the Crankshaft, we will un-box the float32x4 and uint32x4 heap object and represent them into an XMM register most of the time (they might be on stack when spilling) for efficiency. In order to do that, we need to introduce the Float32x4 and Uint32x4 representation, introduce instructions for representation inference and change, allocate xmm register for float32x4 and uint32x4 value and handle the deoptimization and OSR translation between optimized code generated from Crankshaft and un-optimized code generated from FullCodeGenerator.

2.3.1 Introducing SIMD instructions

Currently SIMD instructions are not in the V8 assembler and disassembler. We need to introduce them first. For this SIMD128 prototyping, the instructions might include movaps, movups, addps, subps, mulps, divpsi, sqrtps, rcpps, shufps, rsqrtps, paddd, psubd, andps, orps, notps and xorps.

2.3.2 Introducing Float32x4 and Uint32x4 type and representation

In the Crankshaft, each value has its type and representation which indicates what kind of register will be allocated for the value. For example, a SMI or Integer32 value will be allocated into a fixed register and a Double value will be allocated into a XMM register.

We will introduce a Float32x4 and Uint32x4 type and representation to represent the float32x4 and uint32x4 value.

2.3.3 Float32x4 and Uint32x4 representation inference and change

Once we have Float32x4 and Uint32x4 representation, we need to add instructions to unbox the float32x4 and uint32x4 heap object from FullCodeGenerator to Crankshaft and box the float32x4 and uint32x4 value from Crankshaft to FullCodeGenerator. The float32x4 and uint32x4 objects' representation is Tagged.

Crankshaft has a phase to infer the value representations for each Hydrogen instruction from the type feedback from machine code generated from FullCodeGenerator (the type feedback is mined from the inline caches) and the JavaScript source code. After the inference, it has a phase to insert the HChange Hydrogen instruction at the places where a value's representation changes.

For the representation inference phase, we need to specify the exact representation for the involved Hydrogen instructions. For example, after we inlined the SIMD.mul operation, we might use HMul Hydrogen instruction for the operation and specify the HMul instruction has a Float32x4 representation. For the representation change phase, we need to add Lithium instructions (LTaggedToFloat32x4, LTaggedToUint32x4 and LFloat32x4ToTagged and LUint32x4ToTagged) to lower the HChange instruction in the IA32 and X64 architectures.

2.3.4 Inlining SIMD operations

We need to inline the SIMD operations so that they could be optimized in the Crankshaft.

For the add, sub, mul, div operation, we might use the existing HAdd, HSub, HMul and HDiv Hydrogen instructions or introduce new Hydrogen instructions for the inlining.

For the other operations, such as shuffle, sqrt and reciprocalSqrt, we will introduce new Hydrogen instructions for the inlining.

The operands in the inlined Hydrogen instructions has the Float32x4 or Uint32x4 representation.

An important implementation issue is whether we need to check operation overflow for SIMD operations on float32x4 and uint32x4. In this prototype, we omit the overflow handling as we do not know what we need to do when it happens. It seems that Dart does not do this from reading the source codes.

2.3.5 Allocating registers for Float32x4 and Uint32x4 values

We need to specify that a value with Float32x4 or Uint32x4 representation needs to be allocated into a XMM register. When there is a spilling, we need to allocate 16-bytes on the stack. We might avoid using movaps instruction first as it might need extra padding in the stack. We could align the spilling location later to see

the performance improvement. Currently the spilled double location is aligned for the IA32 architecture.

2.3.6 Handling deoptimization for Float32x4 and Uint32x4 values

Crankshaft inserts type check instructions before using the type feedback information from the executed program behaviour. When the type assumption fails, the Crankshaft generated code will deoptimize to the FullCodeGenerator generated codes. This means there is a state (registers and stack) synchronization between the two worlds. For example, the FullCodeGenerator uses a stack slot for a HeapNumber and Crankshaft uses a XMM register, and the deoptimizer will serialize the XMM value, create a HeapNumber to contain the value and write the HeapNumber address into the corresponding stack slot.

For the Float32x4 and Uint32x4 values, if they are parts of the state synchronization, we need to write them into the translation, and box them into a heap object and write the heap object address into the corresponding stack slot.

2.3.7 Handling OSR for Float32x4 and Uint32x4 values

When a hot loop in the function executes enough iterations and makes the function hot, V8 might optimize the function by translating the program state from FullCodeGenerator generated code to Crankshaft generated code first and then executing Crankshaft generated codes from that state. This is called online stack replacement.

For the Float32x4 and Uint32x4 heap objects, if they are parts of the state synchronization, we need to unbox them first, put them into the right stack slots and make sure they will be loaded into the corresponding XMM register.

2.3.8 Introducing Float32x4Array and Uint32x4Array (optional)

In the Crankshaft, we need to implement loading and storing Float32x4Array and Uint32x4Array elements into the process of lowering HLoadKeyed and HStoreKeyed hydrogen instructions into machine instructions for X64 and IA32. We also need to make sure HLoadExternalArrayPointer works for Float32x4Array and Uint32x4Array.

2.4 Garbage Collection Impact

The float32x4 and uint32x4 object are allocated in the heap in the FullCodeGenerator, and stayed in the XMM registers in the Crankshaft. Implementing them should have little impact for the garbage collection:

- They are allocated in the NEW space first.

- They will be migrated to the OLD DATA space if they survive enough scavenging in the NEW space.

V8 supports allocating objects in the OLD DATA space directly, we might optimize the initial object location (in NEW space or OLD DATA space) in the future.

2.5 Testing and benchmarking

V8 has a unit testing framework and we will add SIMD128 test cases for the regression test. Peter's benchmark application is at https://github.com/PeterJensen/mandelbrot.

Beside the SIMD128 test cases and benchmarks written by ourselves, we could utilize the existing Dart SIMD128 test case and benchmarks.

- dart2js works for Dart SIMD programs. It creates a Float32x4 and Uint32x4 JavaScript objects and uses the JavaScript Float32Array to translate the Dart Float32x4List.
- We could modify dart2js to detect whether the underlying engine supports the float32x4 and uint32x4 constructors, if yes, generate JavaScript codes to use them; if no, fall back to create the Float32x4 and Uint32x4 objects. After this, we could use the SIMD test cases from Dart.
- The matrix_bench from https://github.com/johnmccutchan/vector_math could be used to benchmark our SIMD128 prototyping in V8.
- The Google IO 2013 Dart SIMD demo application (https://github.com/johnmccutchan/spectre/tree/master/web/skeletal_bench) could be used to benchmark our SIMD128 prototyping in V8. Currently there are some issues to run this application and we have discussed with John at https://github.com/johnmccutchan/spectre/issues/118.

3 Reference

1. https://github.com/johnmccutchan/ecmascript_simd
2. https://www.dartlang.org/articles/simd/
3. https://bugzilla.mozilla.org/show_bug.cgi?id=904913
4. http://www.ecma-international.org/ecma-262/5.1/
5. http://wiki.ecmascript.org/doku.php?id=strawman:value_objects

6. https://github.com/PeterJensen/mandelbrot

7. https://github.com/johnmccutchan/ecmascript_simd/issues/10

8. https://github.com/johnmccutchan/spectre/issues/118

9. Peter Jensen's SIMD prototype