Algorithm Verification with Open Source and System Verilog

Andra Socianu

Daniel Ciupitu

accellera SYSTEMS INITIATIVE

2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

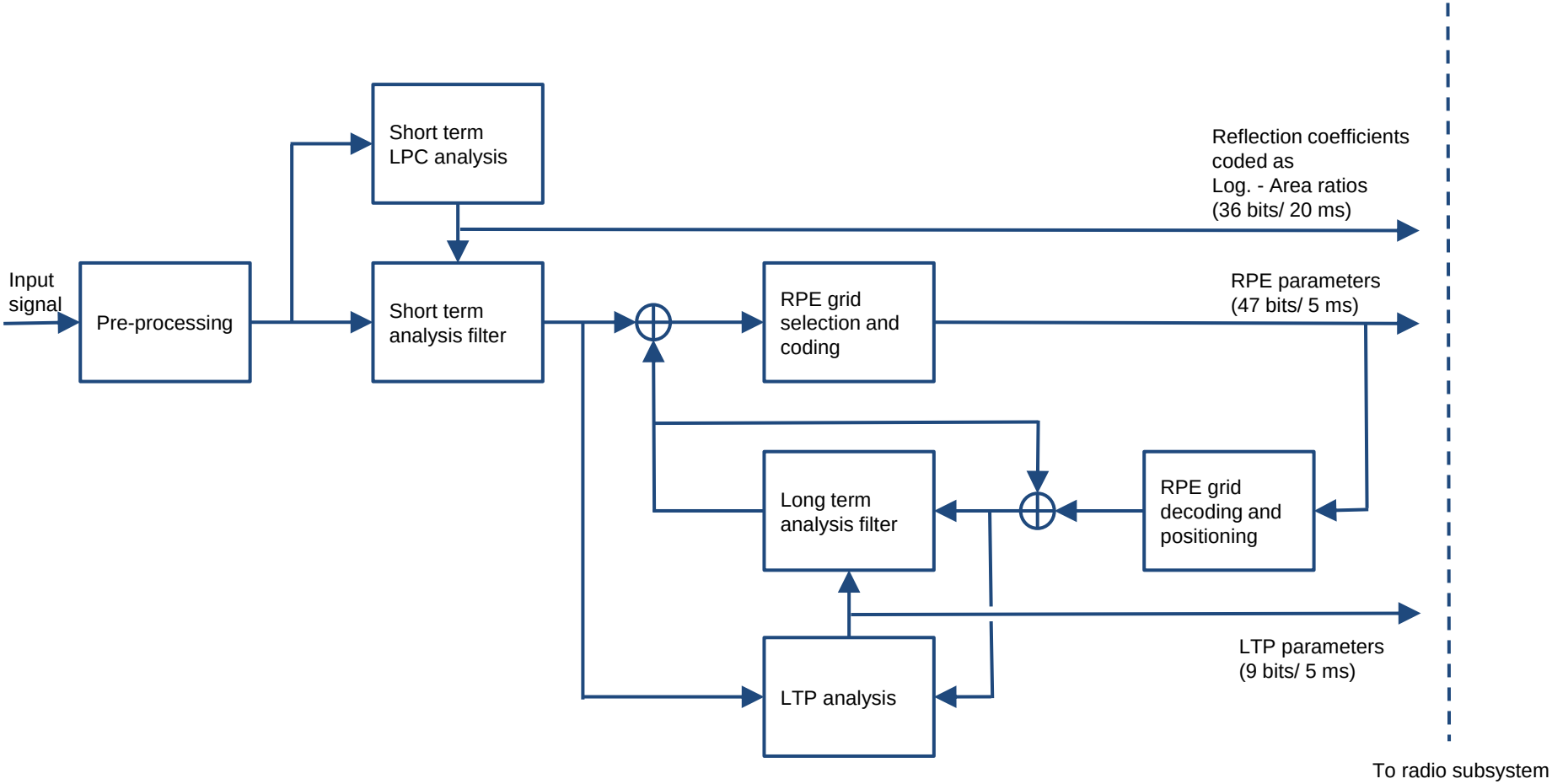
Agenda

- Verification of algorithmic blocks using Octave
- Case study I : SV – SHA-3 – Octave
- Case study II : SV – DSP functions – Octave
- Case study III : SV – DSP functions – SystemC
- Conclusions
- Q&A

Algorithmic Blocks

- Speech/signal/image processing and analysis
- Data encryption/decryption
- BB/radio signal modulation/demodulation
- Signal filtering
- Error detection and correction
- Data compression
- etc.

GSM Vocoder



Implementation Challenges

- Complexity
- Performance
- Debug
- Verification language limitations

Alternatives

- Matlab or Octave
- SystemC
- C++
- etc.

What is Octave?

GNU Octave is:

- a high-level interpreted language for numerical computations
- usually used through its interactive command line interface, but it can also be used to write non-interactive programs
- allows cross language communication
- open source (GNU-GPL)

Where to Get It

There are two Octave packages that need to be installed:

- Main application: ftp://ftp.gnu.org/gnu/octave
- Development package: http://goo.gl/yjHGbp

This paper uses version 3.4.3 of the above packages.

How to Install It

- Create a Makefile by running the script `configure‘ which you can find in the Octave package
- Run `make‘ to compile the sources
- Run `make install‘ to install octave and a copy of its libraries and its documentation

How to Test It

- Start Octave by typing *octave* in the Linux prompt

```
$: octave
```

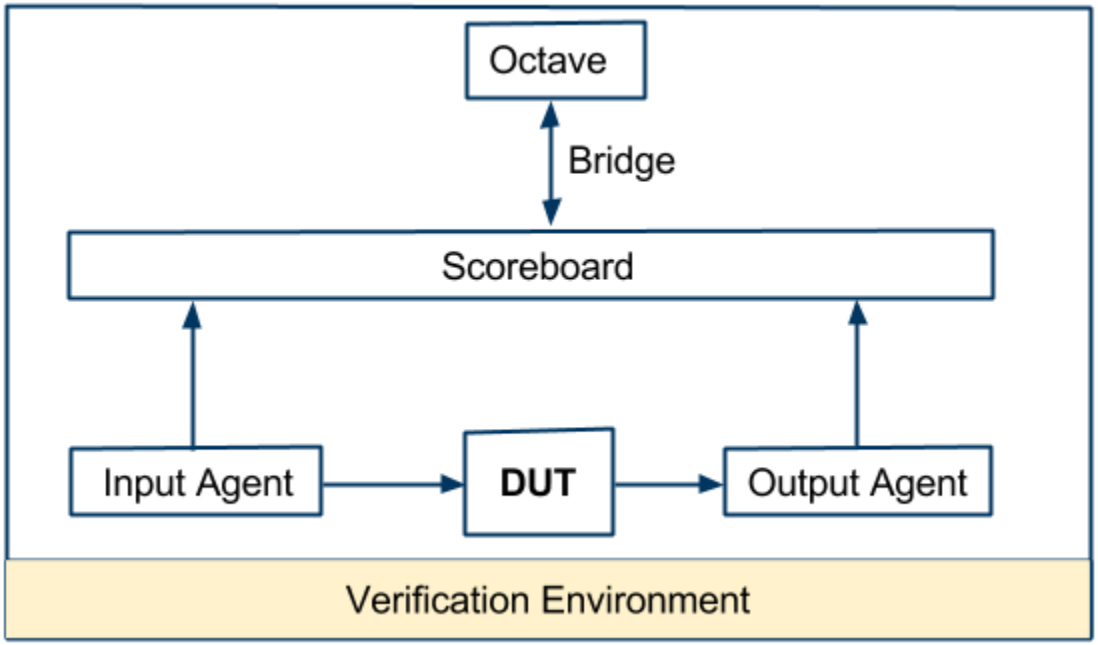
- Search for a known function

```
octave:1> which fft
`fft' is a function from the file /usr/lib64/octave/3.4.3/oct/x86_64-redhat-linux-gnu/fft.oct
```

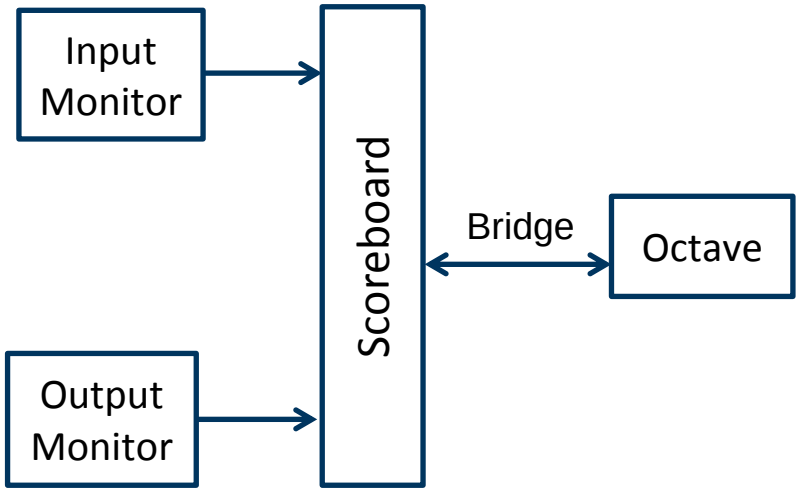
- Test a known function output

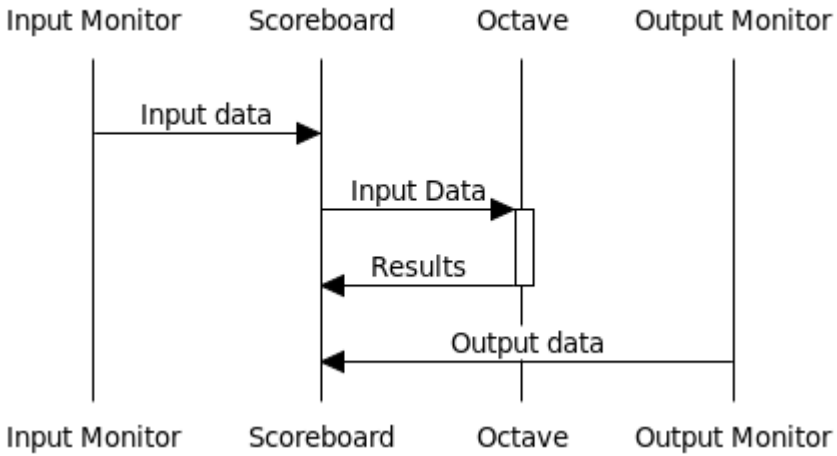
```
octave:2> cos(2*pi)
ans =  1
```

Verification Environment



Score Boarding



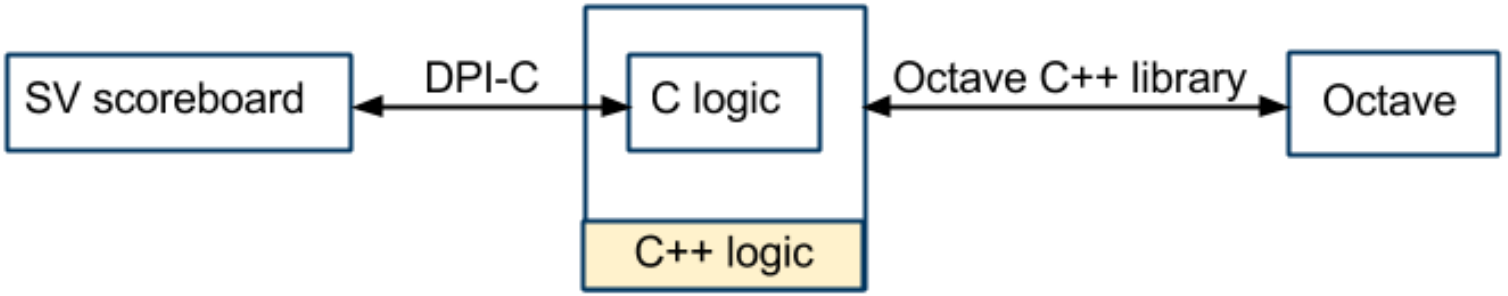


How to Connect SV and Octave

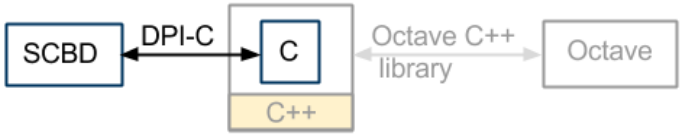
External interfaces:

- DPI-C (Direct Programming Interface)
- VPI (Verilog Procedural Interface)
- PLI (Programming Language Interface)
- etc.

The Bridge



How to Use DPI-C API



External interfaces:

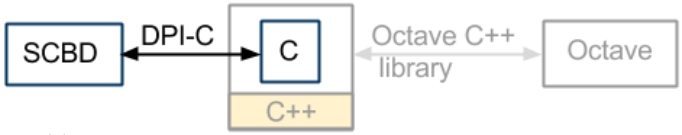
- **import** functions and tasks: implemented in C and called from SV

```
import "DPI-C" function int c_function (input int a, output int b,
inout int c);
```

- **export** functions and tasks: implemented in SV and called from C

```
export "DPI-C" sv_function;
```

System Verilog <-> C



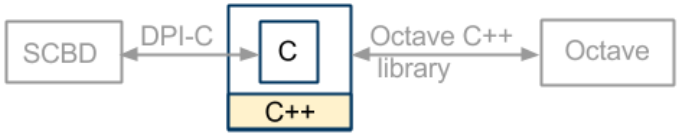
SV

```
import "DPI-C" function void c_hello_world();

class amiq_hello_world extends uvm_component;
 function sv_hello_world();
  `uvm_info("AMIQ_HELLO_WORLD", "Hello world from SV file", UVM_NONE);

  c_hello_world();
 endfunction
endclass
```

C

```
extern "C" {
   void c_hello_world() {
      printf("[AMIQ_HELLO_WORLD] Hello world from C file\n");

      cpp_hello_world();
   }
}
```

C <-> C++



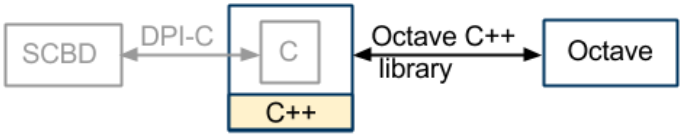
C

```
extern "C" {
   void c_hello_world() {
      printf("[AMIQ_HELLO_WORLD] Hello world from C file\n");

      cpp_hello_world();
   }
}
```

C++

```
void cpp_hello_world() {
   cout << "[AMIQ_HELLO_WORLD] Hello world from C++ file" << endl;

   oct_hello = load_fcn_from_file("hello.m", "", "", "hello", true);

   feval("hello", oct_in_list, 1);
}
```

Octave C++ Library



To get access to the octave C++ API you need to include its libraries. This will allow you to use script files, oct-files and built-in functions.

Libraries:

- main C++ library:

```
#include <octave/octave.h>
```

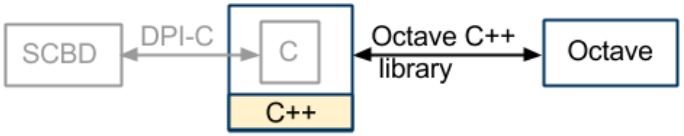
- Octave main() function:

```
#include <octave/oct.h>
```

- virtual terminal support:

```
#include <octave/parse.h>
```

C++ <-> Octave



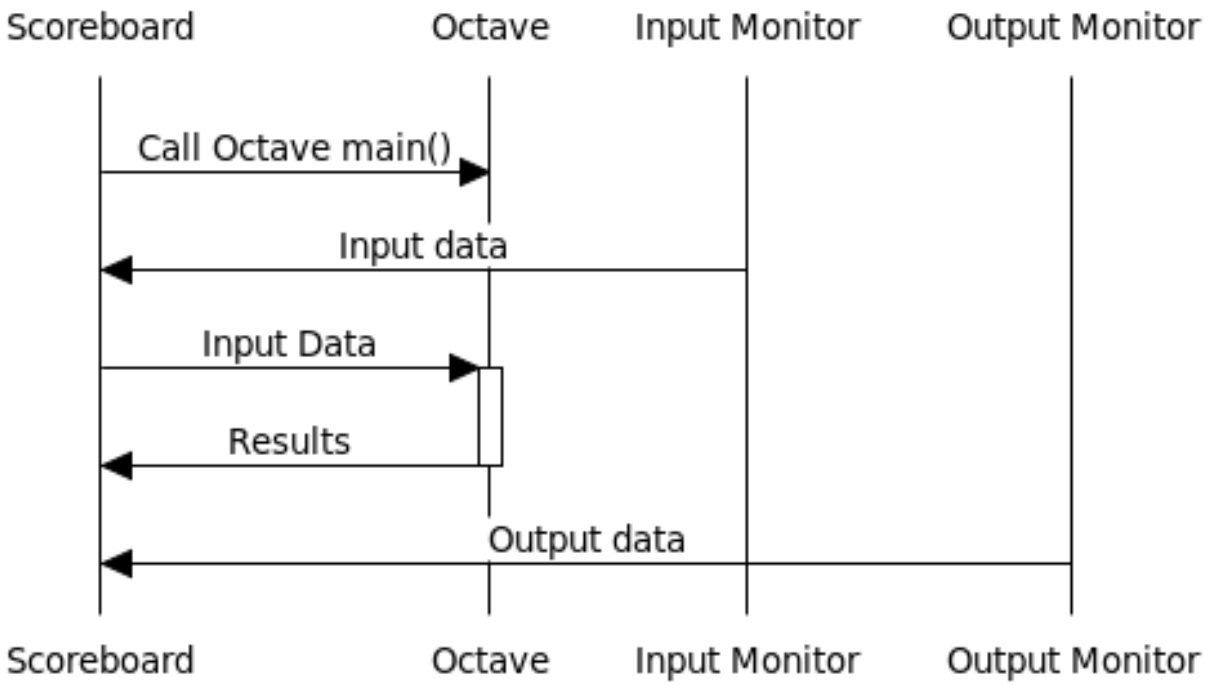
C++

```
void cpp_hello_world() {
   cout << "[AMIQ_HELLO_WORLD] Hello world from C++ file" << endl;

   oct_hello = load_fcn_from_file("hello.m", "", "", "hello", true);

   feval("hello", oct_in_list, 1);
}
```

Oct

```
function amiq_hw ();
   disp("[AMIQ_HELLO_WORLD] Hello world from Octave file");
end
```

Octave Initialization



Octave Initialization

```
// Initialize the Octave interpreter
void initialize_octave_cpp() {
    string_vector argv(2);
    argv(0) = "embedded";
    argv(1) = "-q";

    octave_main(2, argv.c_str_vec(), 1);
}
```

Calling Octave Built-In Functions

```
// Input parameters list for octave function
octave_value_list oct_in_list;

// Output message as a Matrix
Matrix oct_output_data(1, output_size);

// Get computed convolution to oct_output_data
oct_output_data = feval("conv", oct_in_list, 1)(0).matrix_value();
```

Calling Octave User-Written Functions

```
// Input parameters list for octave function
octave_value_list oct_in_list;

// Output message as a Matrix
Matrix oct_output_data(1, output_size);

// Pointer to Octave custom function
octave_function *conv_fct;

// Load Octave custom function
conv_fct = load_fcn_from_file ("amiq_conv.m", "", "", "amiq_conv ", true);

// Get computed convolution to oct_output_data
oct_output_data = feval("amiq_conv ", oct_in_list, 1)(0).matrix_value();
```

Compiling

Create a shared library that contains the Octave library and the C++ code. This library will be passed to the simulator which runs System Verilog.

TIP: Compiler options can be easily obtained by running:

```
$: mkoctfile -link-stand-alone -v cpp_code.cpp
```

Compile Command

```
mkoctfile -link-stand-alone -v my_file.cpp
g++ -c -fPIC -I/usr/include/octave-3.4.3/octave/.. -
I/usr/include/octave-3.4.3/octave -I/usr/include/freetype2 -O2 -
g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 -fexceptions -
fstack-protector --param=ssp-buffer-size=4 -m64 -
mtune=generic my_file.cpp -o my_file.o
g++ -shared -Wl,-Bsymbolic -o my_file.oct my_file.o -link-
stand-alone -L/usr/lib64/octave/3.4.3 -L/usr/lib64 -loctinterp -
loctave -lcruft -L/usr/lib64/atlas -llapack -L/usr/lib64/atlas -
lf77blas -latlas -lfftw3 -lfftw3f -lm -L/usr/lib/gcc/x86_64-redhat-
linux/4.4.6 -L/usr/lib/gcc/x86_64-redhat-
linux/4.4.6/../../../../lib64 -L/lib/../lib64 -L/usr/lib/../lib64 -
L/usr/lib/gcc/x86_64-redhat-linux/4.4.6/../../.. -lgfortranbegin -
lgfortran -lm
```

Compile Command

```
g++ \
      -Wall \
      -m64 \
      -I${PROJ_HOME}/octave \
      -I${PROJ_HOME}/c \
      -I${PROJ_HOME}/sim \
      -I/usr/include/octave-3.4.3/octave/.. \
      -I/usr/include/octave-3.4.3/octave \
      -I/usr/include/freetype2 \
      -L${PROJ_HOME}/sim \
      -L/usr/lib64/octave/3.4.3 \
      -L/usr/lib64 \
      -L/usr/lib64/atlas \
      -L/usr/lib/gcc/x86_64-redhat-linux/4.4.6 \
      -L/usr/lib/gcc/x86_64-redhat-linux/4.4.6/../../../../lib64 \
      -L/lib/../lib64 \
      -L/usr/lib/../lib64 \
      -L/usr/lib/gcc/x86_64-redhat-linux/4.4.6/../../.. \
      -Wl,-rpath \
      -Wl,/usr/lib64/octave/3.4.3 \
      -loctinterp \
      -loctave \
      -lcruft \
      -llapack \
      -lf77blas \
      -latlas \
      -lfftw3 \
      -lfftw3f \
      -lreadline \
      -lm \
      -lgfortranbegin \
      -lgfortran \
      -shared \
      -fPIC \
      -o libcpp_oct.so ${PROJ_HOME}/c/amiq_fft256_c_oct_container.cpp
```

Running

You can compile and run simulations with any of the 3 major EDA vendors simulators:

- *irun* (Cadence) and *vcs* (Synopsys): include the shared library at compile time
- *vlog/vsim* (Questa): include the shared library at run time

TIP: If you run into errors related to octave main() calls, the macro call "OCTINTERP_API" from "octave.h" library file has to be removed.

Success!!!

```
UVM_INFO @ 0 [AMIQ_HELLO_WORLD]: Hello world from SV file
[AMIQ_HELLO_WORLD]:              Hello world from C file
[AMIQ_HELLO_WORLD]:              Hello world from C++ file
[AMIQ_HELLO_WORLD]:              Hello world from Octave file
```

Octave Data Types

Special Octave types:

- octave_value_list (generic type)
- RowVector
- ColumnVector
- Matrix
- DiagMatrix
- etc.

How to Pass Data to/from DPI-C

System Verilog	C (input*)	C (output/inout*)
byte	char	char*
string	const char*	char**
real	double	double*
bit	unsigned char	unsigned char
logic	unsigned char	unsigned char*
int	int	int*
shortreal	float	float*
open array []	const svOpenArrayHandler	svOpenArrayHandler

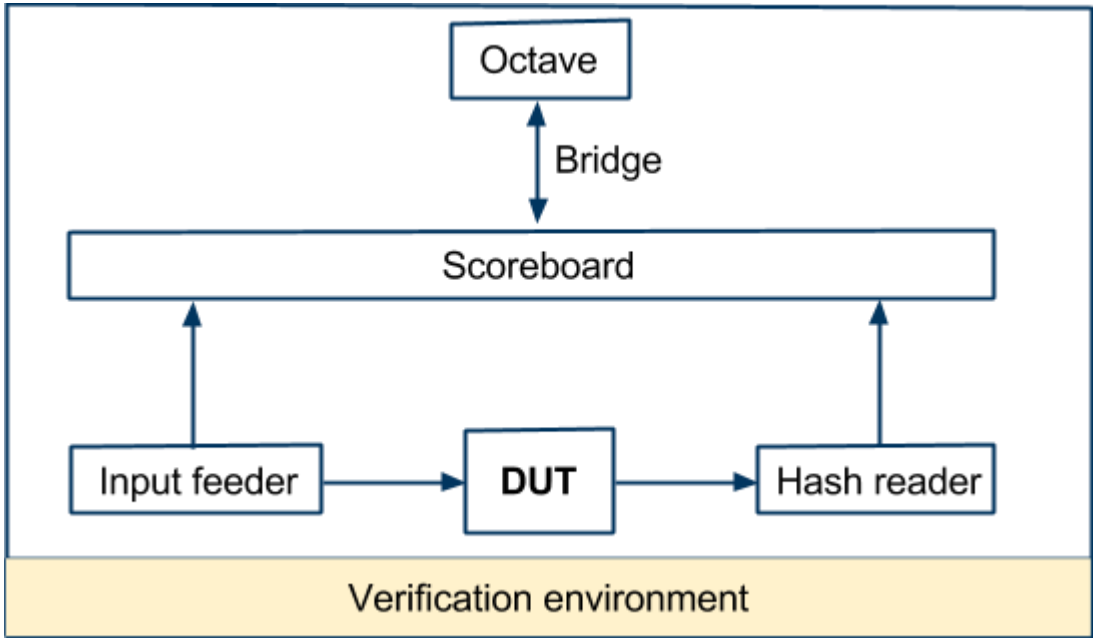
*direction is relative to System Verilog argument

Recap

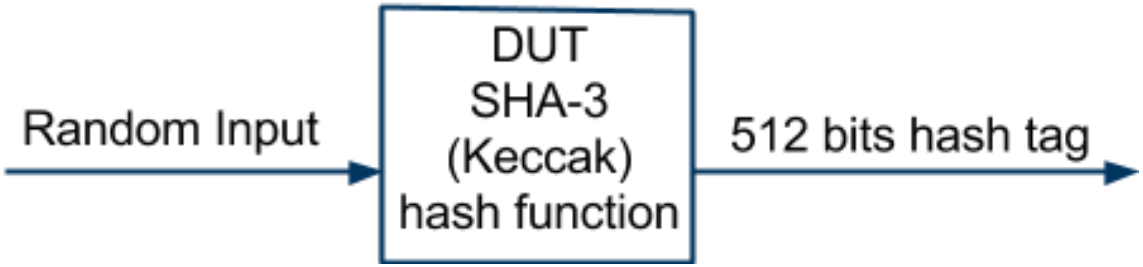
- What is Octave
- How to install Octave
- How to test Octave
- How to create a SV-Octave bridge
- How to compile and run a simulation

Case study I
SV – SHA 3 – Octave

Verification Environment



DUT

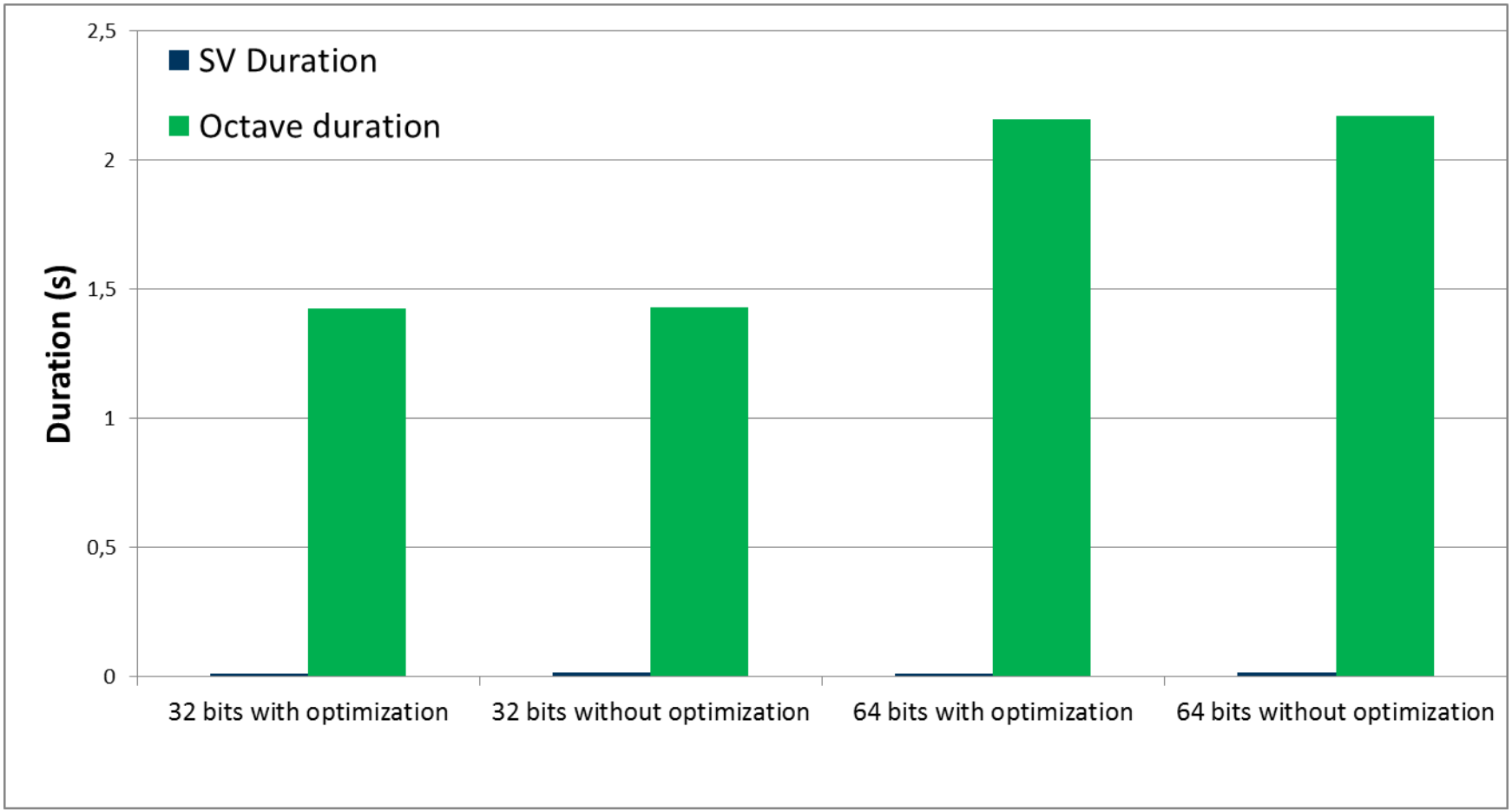

Random Input
DUT
SHA-3
(Keccak)
hash function
512 bits hash tag

What is SHA-3/Keccak?

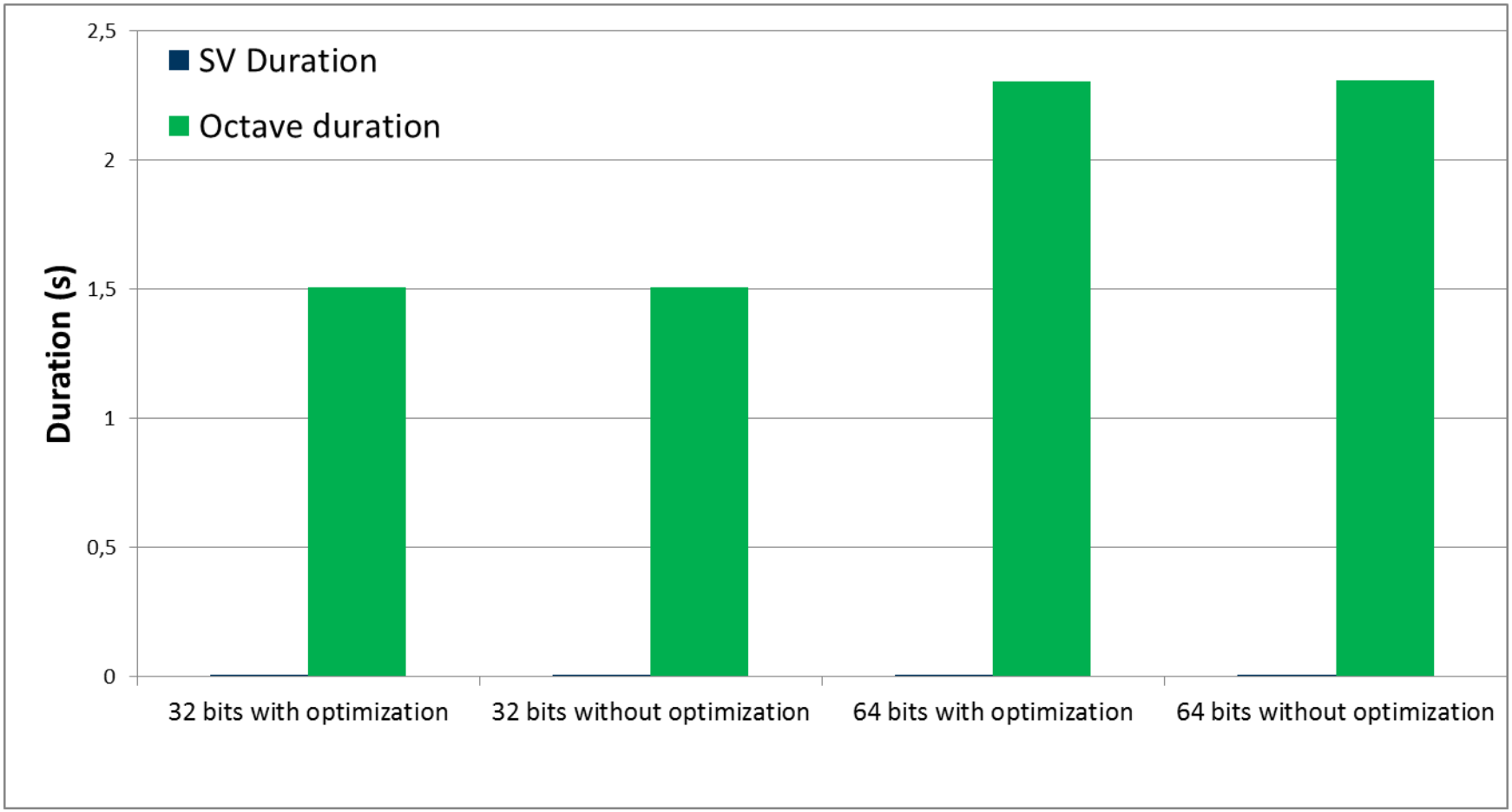
SHA-3 is:

- a cryptographic hash function
- a subset of the cryptographic primitive family Keccak
- an alternative to SHA-1 and SHA-2 which in theory are vulnerable

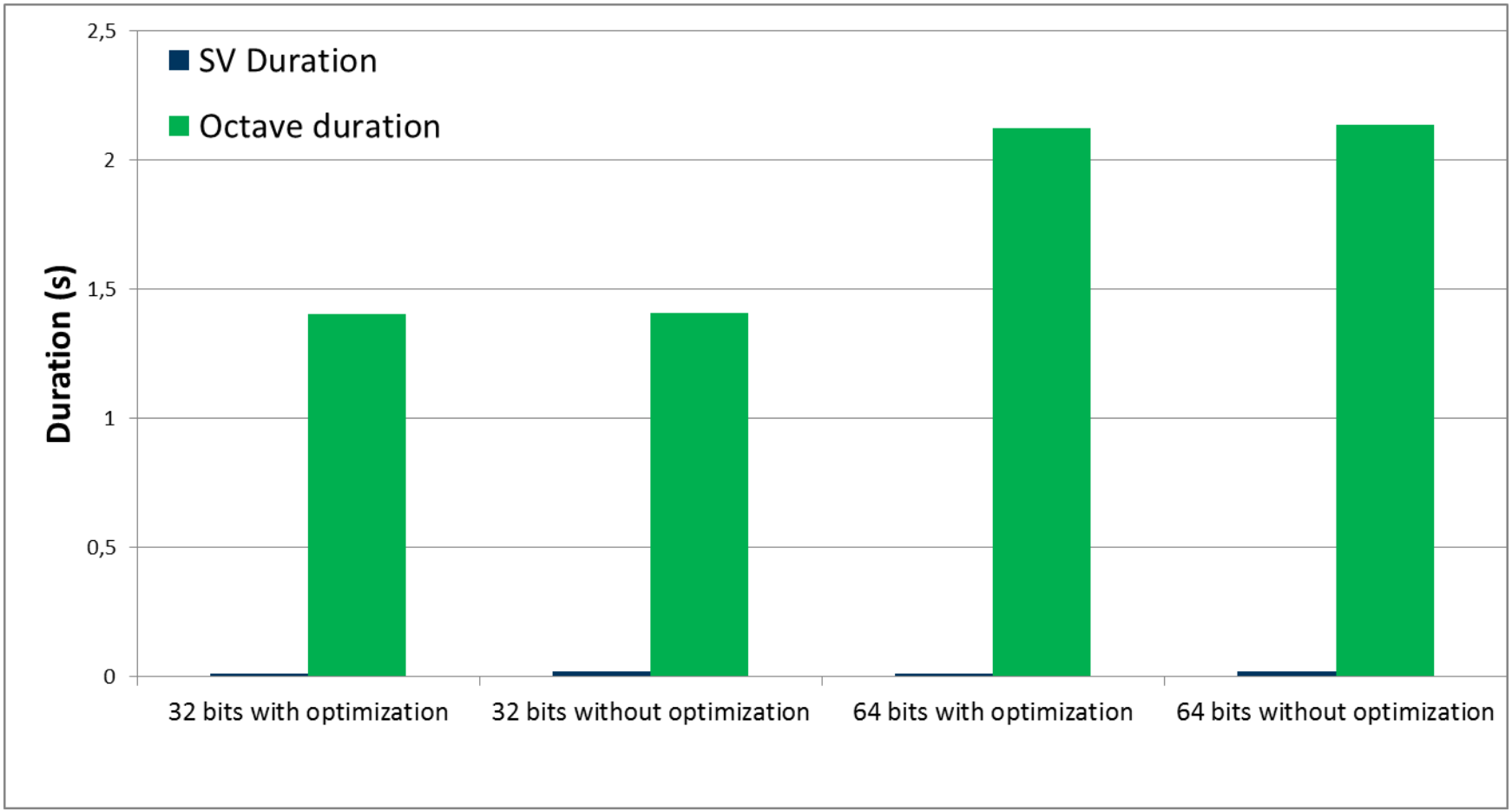
Performance – Simulator 1



Performance – Simulator 2



Performance – Simulator 3



Implementation Effort

	System Verilog	Octave
Number of code lines	223	132
Implementation time	1 day	1 day
Debug time	2 days	1 day

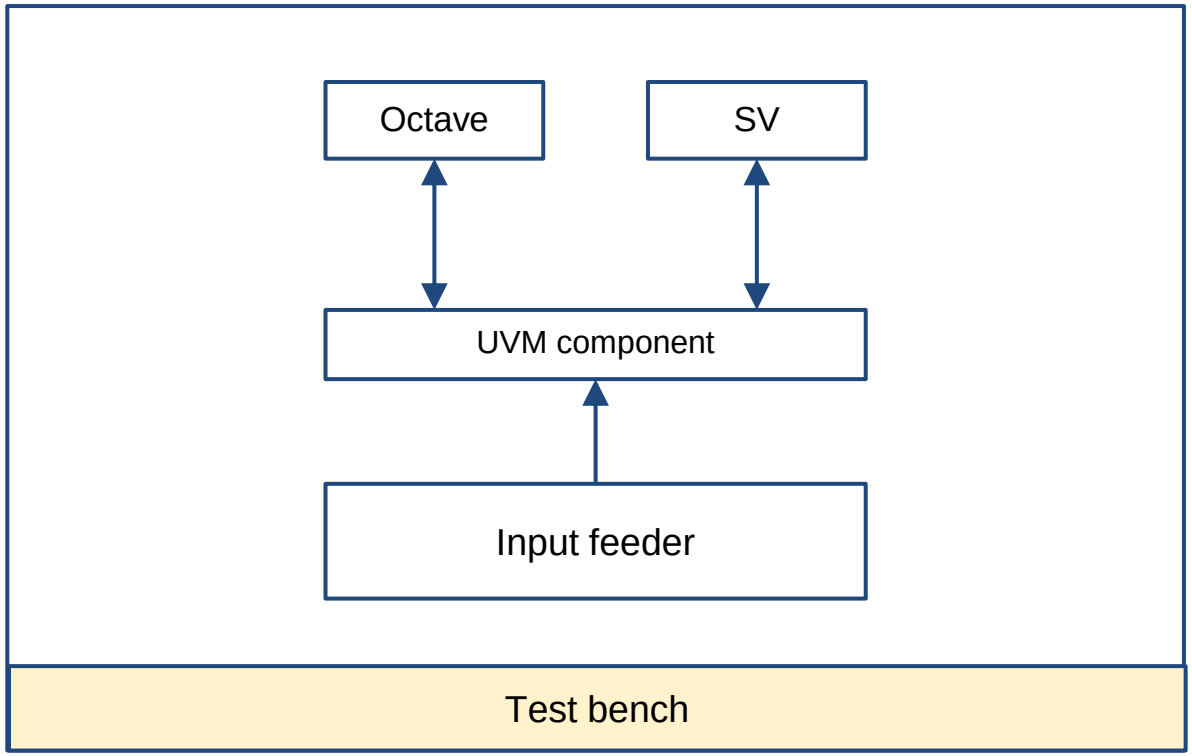
Why did we pushed forward?

- Keccak deals mainly with matrix operations
- System Verilog is optimized to work with multi-dimension arrays
- Things could be different for other types of functions

TIP: A simple multiplication of two matrixes takes 9.32 ms for the Octave computation and context switch, while the same operation takes 8.27 ms in SV.

Case study II
SV – DSP functions – Octave

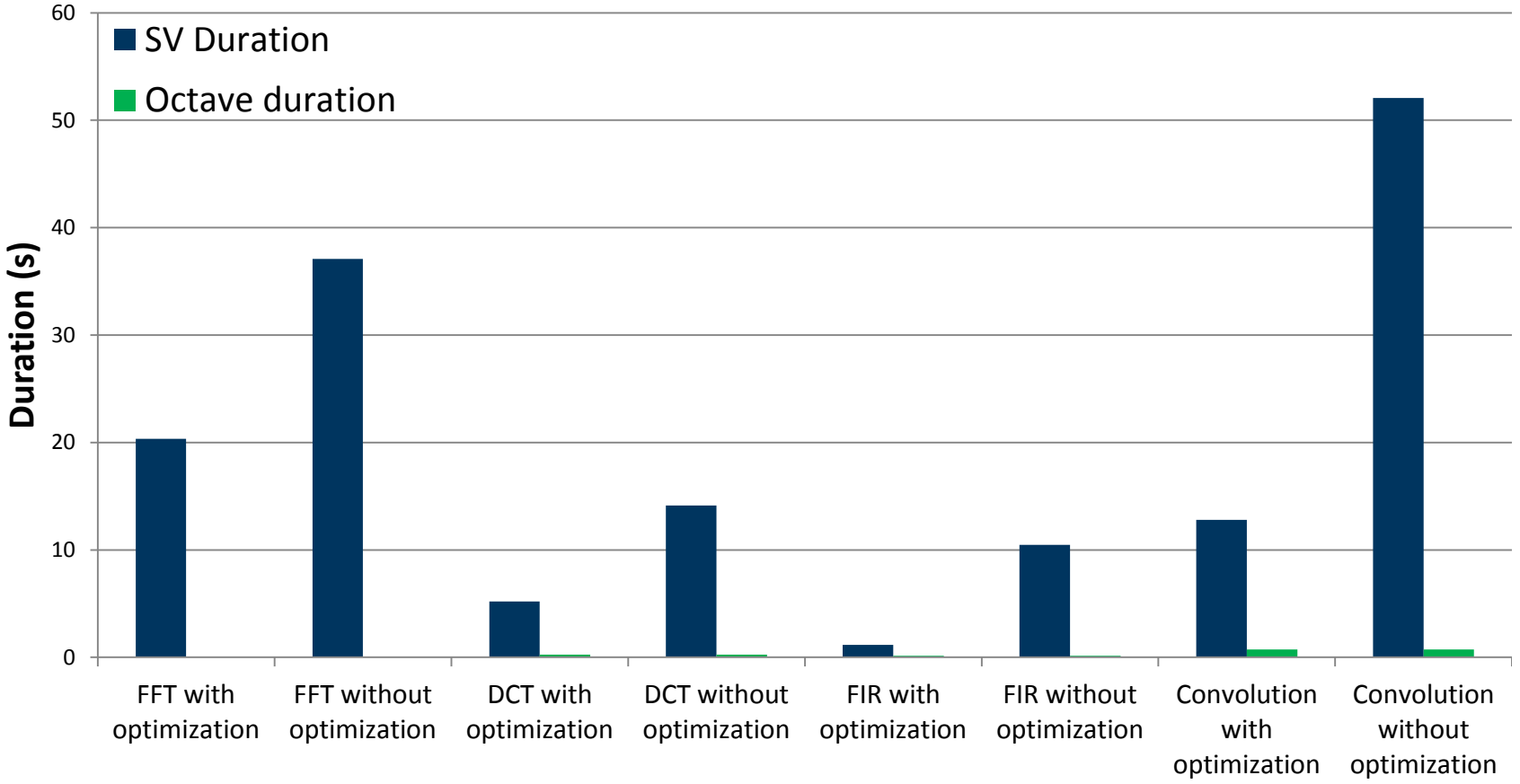
Test Bench



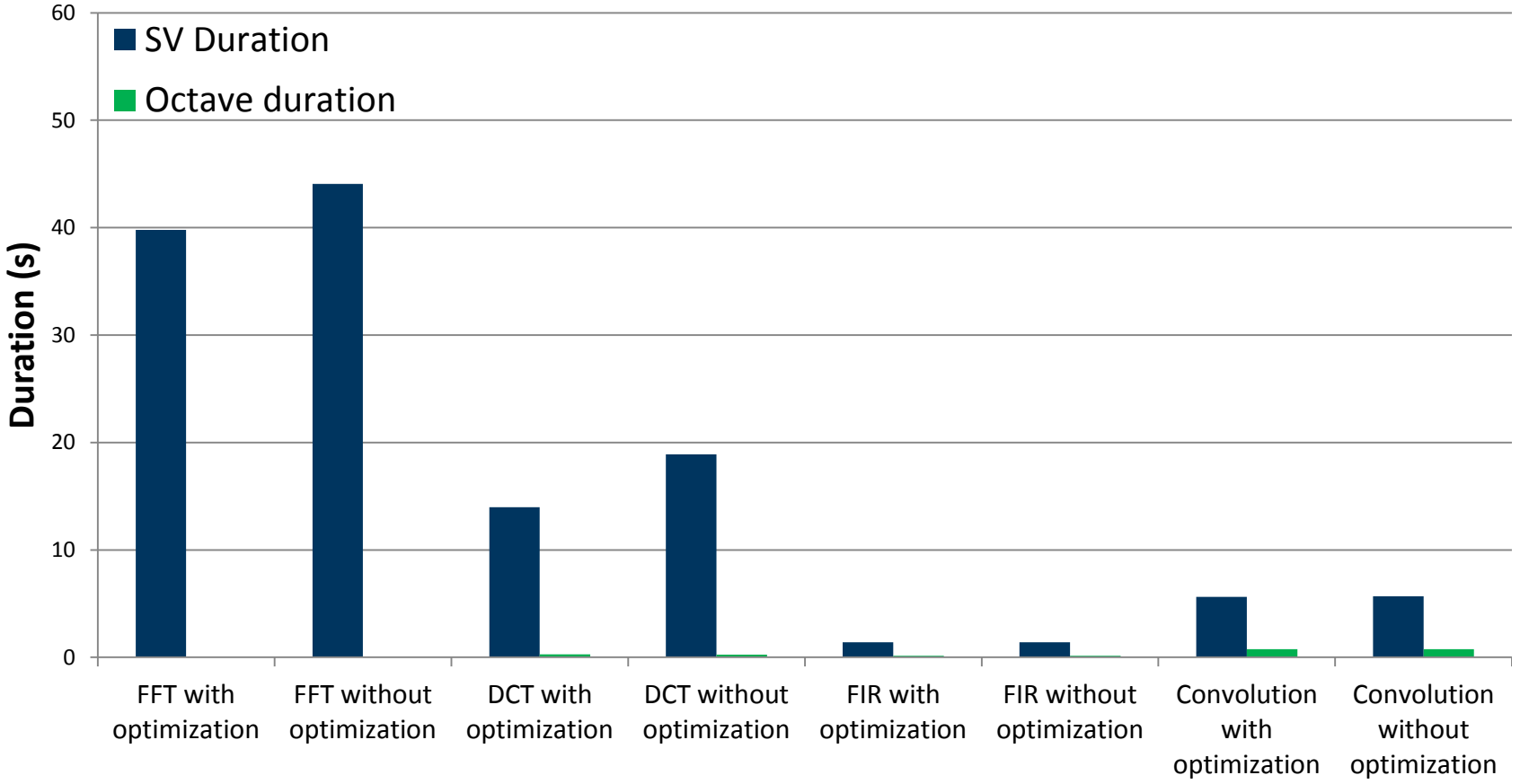
DSP Functions We Used

- FFT or Fast Fourier Transform
- DCT or Discrete Cosine Transform
- FIR or Finite Impulse Response filter
- Convolution

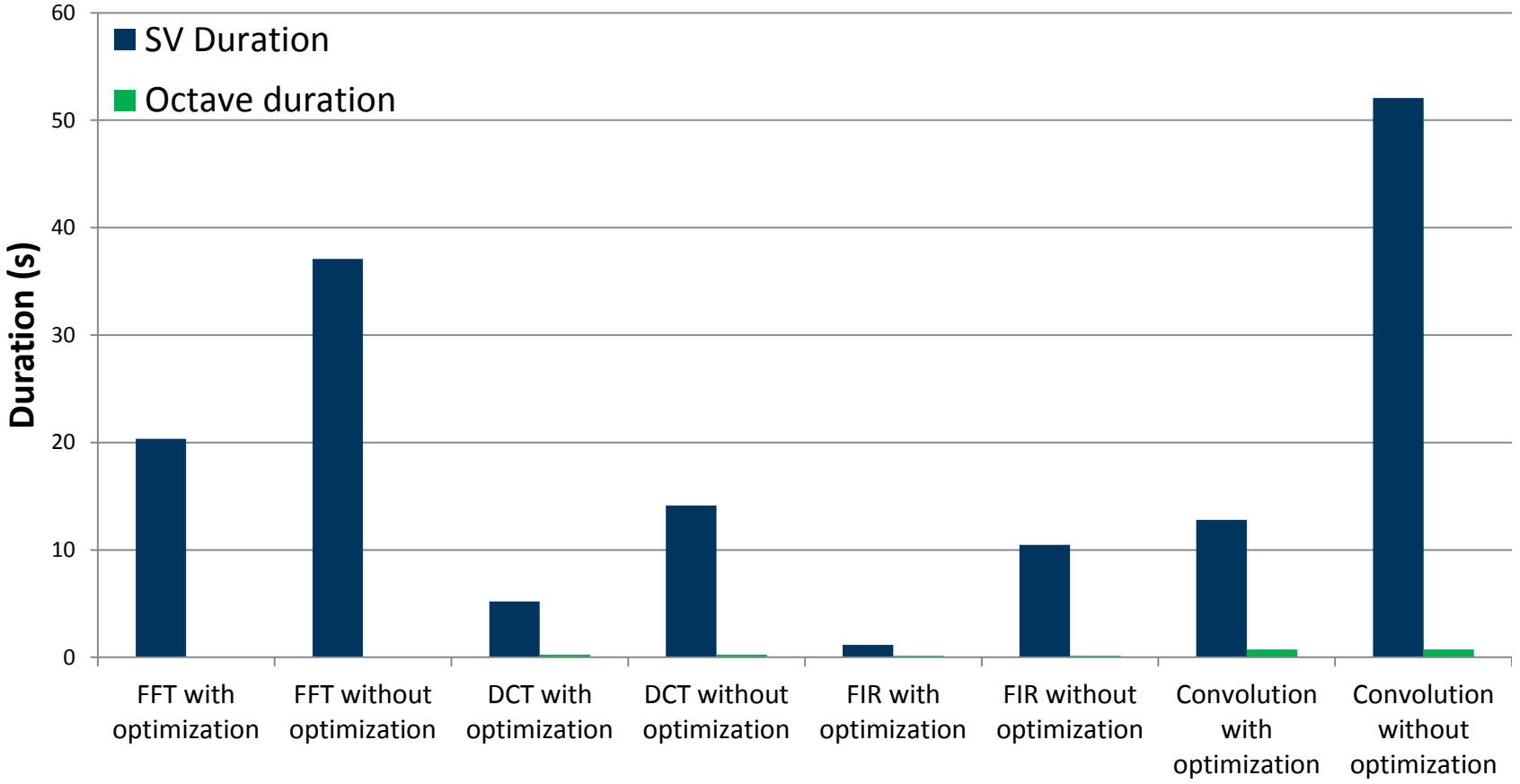
Performance – Simulator 1



Performance – Simulator 2



Performance – Simulator 3

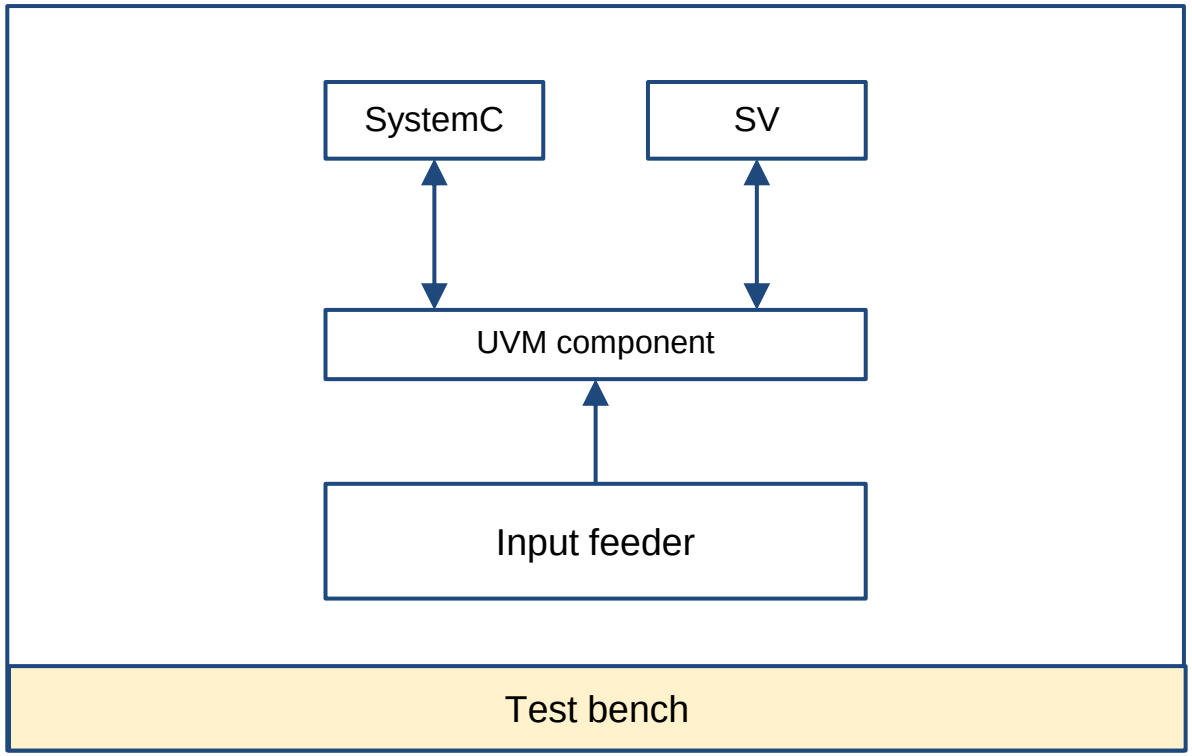


Implementation Effort

		System Verilog	Octave
FFT	Number of code lines	30	3
	Implementation time	1 day	0,5 days
	Debug time	1 day	0,5 days
DCT	Number of code lines	19	3
	Implementation time	1 day	0,5 days
	Debug time	1 day	0,5 days
FIR	Number of code lines	21	3
	Implementation time	1 day	0,5 days
	Debug time	1 day	0,5 days
Convolution	Number of code lines	12	3
	Implementation time	1 day	0,5 days
	Debug time	1 day	0,5 days

Case study III
SV – DSP functions – SystemC

Test Bench



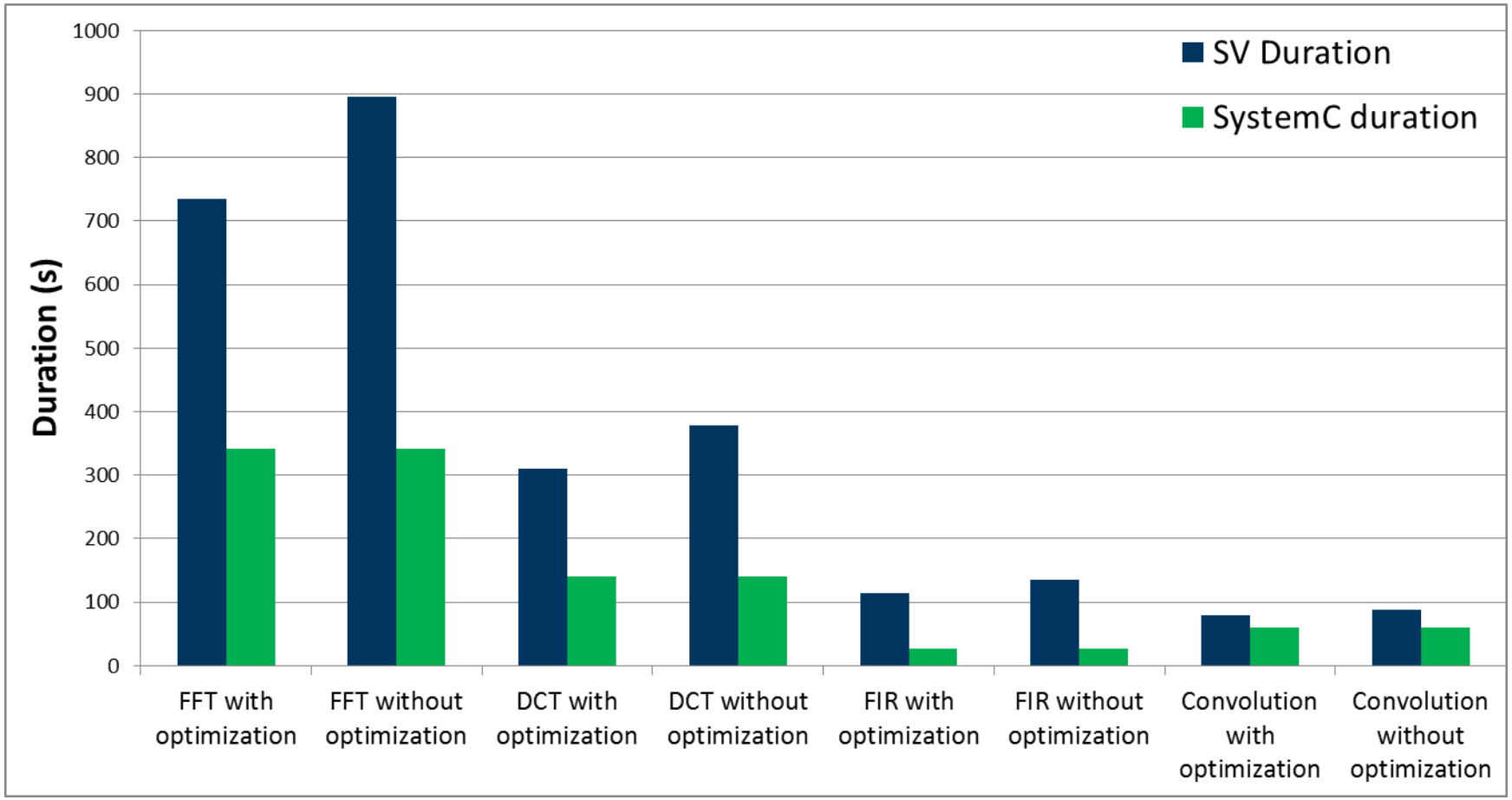
What is SystemC?

- a C++ library
- provides an event-driven simulation interface
- supports system level design
- inherits all C++ features
- adds new data types
- fixed point computation

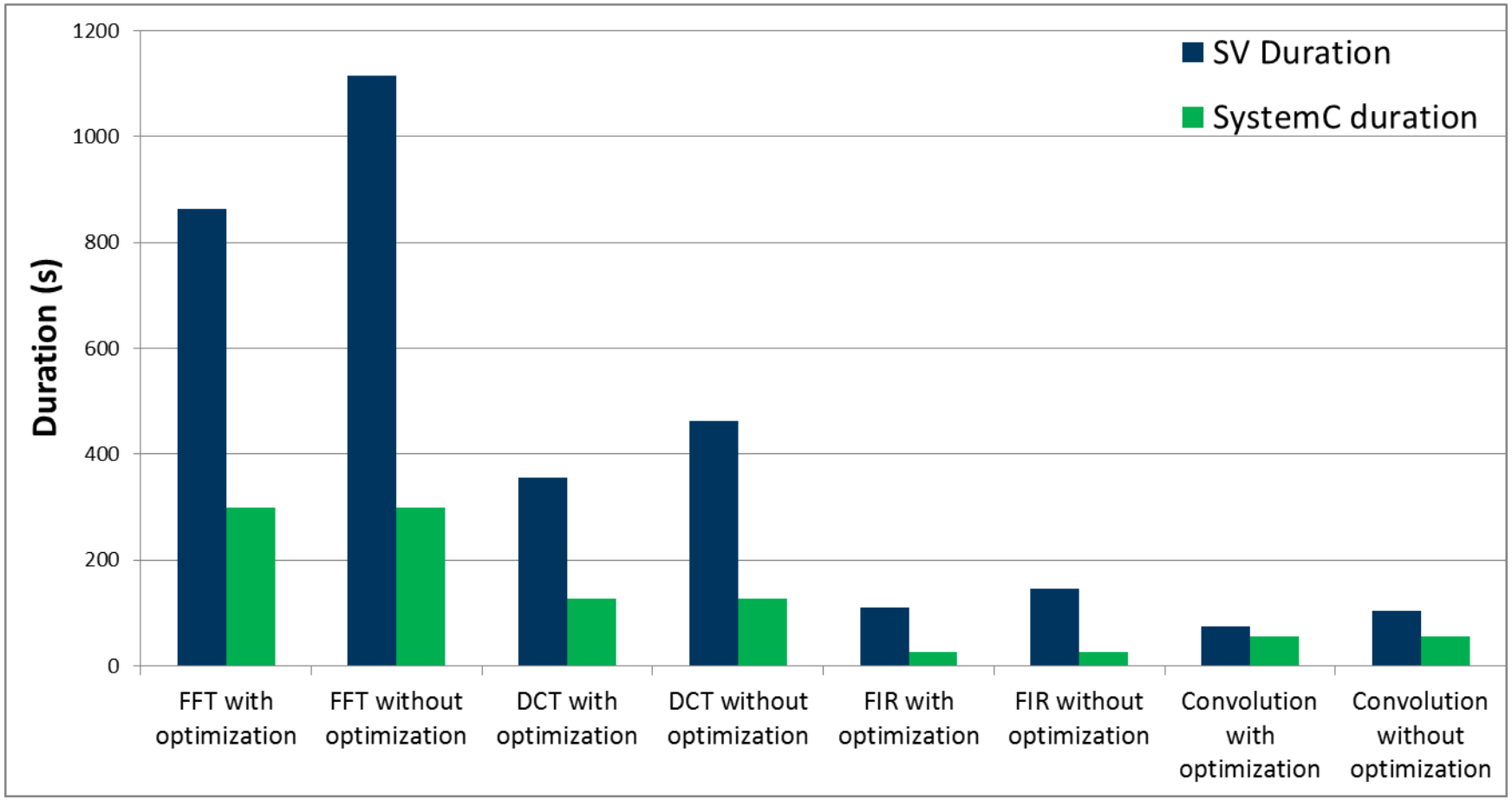
SystemC Specific Data Types

System Verilog	SystemC
bit, logic, reg wire	bool, sc_bit, sc_logic
bit, logic, reg, wire vector	sc_bv, sc_lv, sc_int, sc_uint
integer, int [unsigned]	[unsigned] int
real, shortreal	double/float
byte [unsigned]	[unsigned] char
enum	Enum
struct	Struct
-	sc_fixed, sc_ufixed

Performance – Simulator 1



Performance – Simulator 2



Implementation Effort

		System Verilog	SystemC
FFT	Number of code lines	30	37
	Implementation time	2 days	1 day
	Debug time	4 days	3 days
DCT	Number of code lines	20	20
	Implementation time	1 day	1 day
	Debug time	2 days	2 days
FIR	Number of code lines	13	16
	Implementation time	1 day	1 day
	Debug time	2 days	2 days
Convolution	Number of code lines	23	21
	Implementation time	1 day	1 day
	Debug time	1 day	1 day

Conclusions

Both Octave and SystemC can offer:

- greater performance
- lower implementation
- lower debug time

Conclusions

- SV is better when multi-dimensional vectors are involved
- Octave is better for particular functions (e.g. FFT, DCT, Convolution or FIR), but language context switch comes at a cost (~2.3 ms/call)
- SystemC can help you if you miss Octave fixed point libraries

Q&A

Q&A

Thank you!