

A Fixed-Point DSP Architecture for Software-Defined Radio

Wouter Kriegler and Gert-Jan van Rooyen

Department of Electrical and Electronic Engineering, University of Stellenbosch

Abstract—Software-defined radios (SDR) perform signal processing in the digital domain, replacing hardware that previously performed such processing. The need exists to rapidly create new SDR applications without designing an entire system from the ground up, and without specialized knowledge of a target platform. This paper describes the design of a generic SDR topology that is highly reconfigurable and promotes a high level of code re-use. The research forms part of a larger project to design a domain-specific language (DSL) in which to describe SDR functionality in a platform-independent way. In this paper, the code synthesis from the DSL is extended to support the Freescale DSP563xx family, and the study provides a roadmap for the creation of such extensions to other embedded platforms.

Index Terms—Software-defined radio. Synchronous data flow. Domain-specific languages.

I. INTRODUCTION

THE most significant value of software-defined radio (SDR) is the re-usability and adaptability of the SDR hardware platform, since it can correct the functionality of a radio through software changes, or even completely change the radio application. This is due to the fact that tasks that would traditionally be performed by hardware can now be implemented using software running on a digital processor. Fig. 1 shows that some additional hardware may still be needed to create a complete SDR application.

However, a highly reconfigurable platform may not be highly flexible, because a significant coding effort may be needed to adapt the SDR. Ideally, SDR source code would be completely independent from the underlying hardware. In practice, various degrees of independence between a hardware platform and the SDR source code exist:

- 1) Software is custom written for a SDR platform, and a change in the SDR may require a major code rewrite. Each SDR instance is specific to the targeted hardware platform.
- 2) Software modules are pre-written for a specific SDR platform, and these modules can be configured and combined to produce a large variety of applications. The software written is still hardware specific.
- 3) Generic software modules are written that can describe a variety of SDR waveforms and actions in a hardware-independent manner. These modules can be configured

W. Kriegler is with the Department of Electrical and Electronic Engineering, University of Stellenbosch, c/o Banghoek Road & Joubert Street, Stellenbosch, 7600, South Africa (Fax: 021 808 4981, Telephone: 021 808 4478, e-mail: wouter@dsp.sun.ac.za)

G-J van Rooyen is with the University of Stellenbosch, South Africa

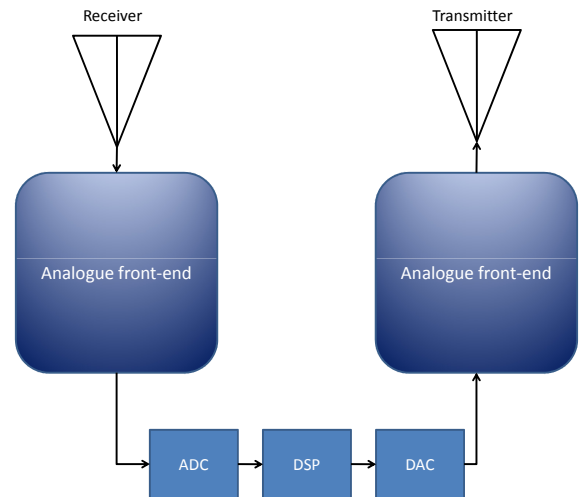


Fig. 1. Basic SDR configuration, showing the separation of hardware and software.

and combined to produce a radio application. For each target platform a code mapping between the generic modules and the hardware-specific high-level language is defined.

These three degrees of independence introduce increasing levels of code re-use in SDR. In Case 3 above, top-level generic definitions of various SDR modules are defined and configured. These modules are then migrated to a hardware-specific platform by means of a *languageA-to-languageB* translator. This means that an SDR can be highly reconfigurable by using these generic software modules, and also portable by means of the hardware-specific translators.

Such a system will consist of the following:

- 1) *Generic DSL modules*. A complete radio definition may consist of many modules or nodes that are linked together to form an SDR application. Fig. 3 illustrates this with an AM demodulation design. For each node in a communication system, a DSL module can be defined and configured and linked together to perform a task.
- 2) *A DSL translator*. This stage will be responsible for generating the hardware-specific high-level language (HLL) for the target platform on which the SDR will be implemented. This means that with a variety of DSL translators a single generic SDR definition can be compiled to many different platforms.
- 3) *Hardware-specific language*. Each platform on which a

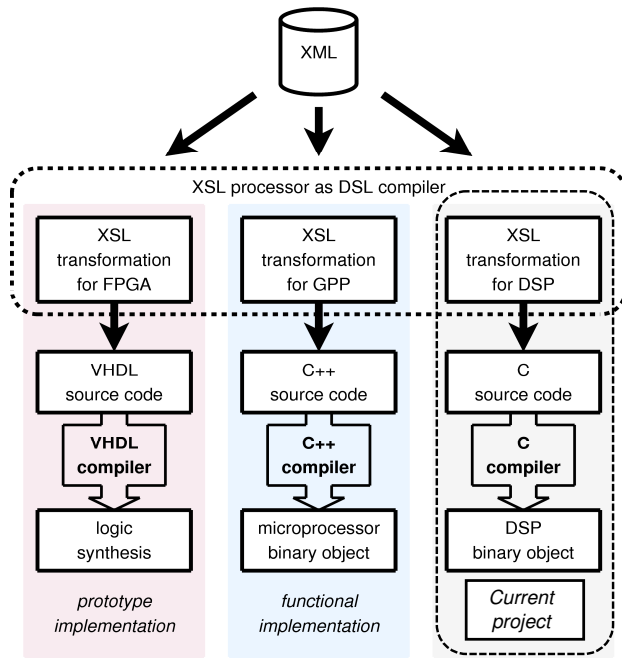


Fig. 2. A single generic XML definition for an SDR application exists that is then sent through the preferred DSL compiler. The appropriate platform-specific HLL source code is then generated and compiled to deliver an executable object. The project on the right is the one discussed in this paper.

SDR will be implemented may have its own high-level language (HLL).

We have taken the approach in our projects of firstly designing an HLL implementation of our SDR architecture for each hardware platform. This provides a reference implementation for later code generation, and is the main objective of this paper. The HLL implementation will be designed with code generation in mind. When a working hardware-specific implementation exists, a translator can be written to transform the generic DSL modules that reside in a SDR library, into a functioning hardware-specific HLL.

In our projects we chose to define these DSL modules in XML. XML is a markup language that lends itself to structured declarative definitions of systems. Translator software is readily available, which allows the XML modules to be translated into arbitrary high-level languages [2]. This is an essential feature, since the the ultimate goal of this project is the automatic code generation of a hardware-specific HLL from a generic XML SDR definition.

Previous research projects at Stellenbosch University have successfully implemented DSL translation to two hardware platforms. Fig. 2 shows a project overview, in which a fully functional implementation to a microprocessor has been completed [1], and a prototype implementation for a VHDL-based hardware platform for FPGA has been demonstrated [2]. This illustrates the re-use of code: A single generic XML definition for a SDR radio exists; different DSL translators then generate code, compatible with a variety of platforms.

This paper will describe the development of the hardware-specific solution for the Freescale DSP563xx family of DSPs (previously Motorola). This is done in preparation of the final

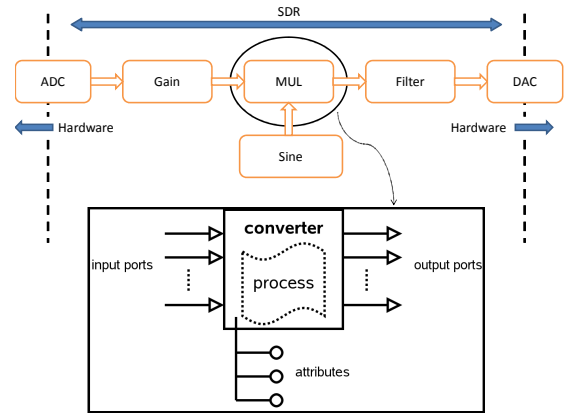


Fig. 3. SDF diagram of a simple SDR application using generic modules.

step in such an implementation, namely the construction of a DSL translator using XSL transformations (XSLT), which is beyond the scope of this paper.

II. A GENERIC SDR TOPOLOGY

In order to develop a platform-independent, reconfigurable SDR topology, a token flow network was developed where a variety of processing nodes (called *converters*) can be linked together into a radio application. Each converter may have several *attributes* that can be used to configure it. Fig. 3 shows the structure of a converter. Its primary task is to convert input tokens into output tokens. A converter must have at least one input port or one output port. Each converter must further contain a process declaration that describes the algorithm that converts inputs into outputs. These converters are then linked to each other to create a complete SDR application, which may be abstracted as a token flow network [3]. Each converter will produce output tokens and pass it on to the next converter in the synchronous data flow (SDF) network. The next converter will then consume that token and produce an output according to its process algorithm and current attributes. Synchronous data flow assumes that each node in the system has a static position, and that the number of samples produced and consumed by each node is known [3]. This is found to be an acceptable constraint for SDR design [1].

Since converters will execute synchronously, an output token of one converter must be passed to the next converter in the SDF graph. A converter may execute multiple times before the next converter in the SDF graph is scheduled. A single converter may therefore produce multiple output tokens before any of these are consumed. These tokens must all be passed to the next converter, and therefore an input queue should exist where these tokens will be stored until they are ready to be processed by the next converter. Tokens in the input queue will be processed in the same order in which they were produced, therefore the input of any converter should be a FIFO queue.

Once all converters of a complete SDR have been defined and linked together, these nodes must execute. Lee and

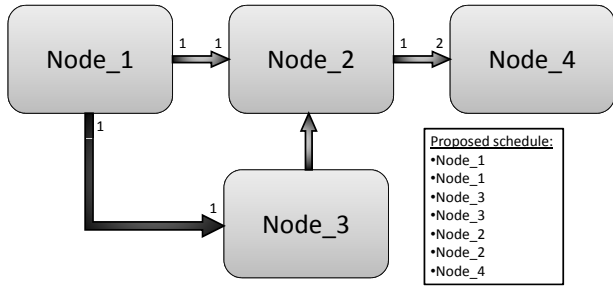


Fig. 4. Trivial SDR graph showing number of samples produced and consumed by each node. The appropriate schedule is also given.

Messerschmitt [3] propose a static scheduling algorithm for SDF networks. This algorithm ensures that each converter executes at the correct time and for the correct number of times. This algorithm can also test the validity of the SDF created. Since this is a static scheduling method, the schedule will only be determined once, and execution of this order will occur periodically. In a practical SDR, scheduling will be done whenever the token flow network (i.e. the SDR structure) is changed. This proposed schedule will then only be valid for that SDF network.

III. IMPLEMENTATION

Previous microprocessor-based SDR projects in this research group used C++, an object-orientated programming (OOP) language. OOP language features such as classes, inheritance, data abstraction and encapsulation, and their close relationship to the abstraction of an SDR as a token flow network, informed this choice of language. However, C++ is not supported on many embedded platforms such as DSPs, and the memory and processing overhead may be prohibitive. For this reason, the C programming language was chosen for the current project, because of the portability to a wide variety of embedded platforms. Some of the OOP features found in C++ did influence the C code design.

We already defined a converter as a node consisting of input ports, a process, and output ports. Consider the network in Fig. 4. *Node_1* has two outputs, linking it to two other nodes. It would be wasteful to duplicate data by using both an output queue and an input queue. The choice between an output queue and an input queue is arbitrary (in particular since only one-to-one interconnections are allowed). For this architecture only an input queue is maintained at each input port. *Node_2* now has memory allocated for two input queues. Each converter also has an output queue pointer, set equal to the memory address of the input queue it should be connected to. This means that a queue allocated in memory is now shared by two converters, thus linking them. If a converter pushes a sample into its output queue, that value would be available in the input queue of the converter sharing that queue as shown in Fig. 5.

As stated earlier, a converter may produce or consume multiple samples. Each queue in the SDF network must therefore be allocated a sufficient amount of memory to ensure that the maximum number of samples that can reside in the queue at any time can be accommodated. In addition to each converter

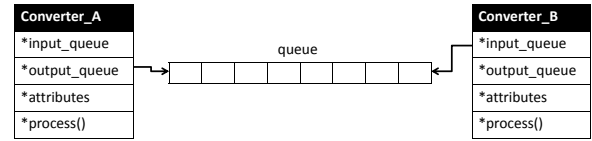


Fig. 5. Two converters sharing a queue allocated in memory. *Converter_A* and *Converter_B* both have read and write access to the queue.

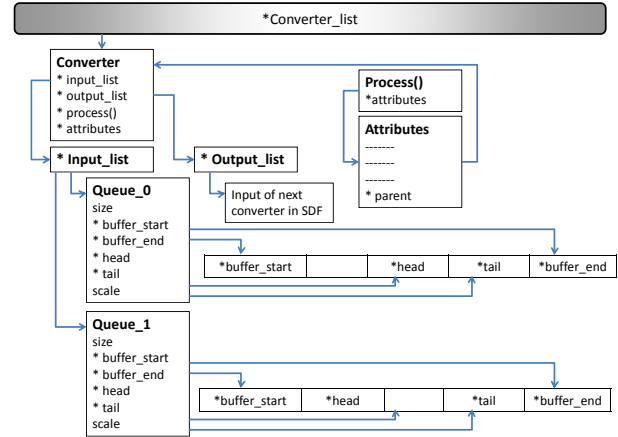


Fig. 6. Converter layout of a typical module in a SDR in a non-OOP language using a variety of pointers to duplicate some of the concepts found in a OOP language.

producing or consuming multiple samples, each converter may produce multiple outputs or consume multiple inputs. Instead of declaring a single input or output queue, we define an input list and an output list, that may consist of multiple pointers to queues, and represent the multiple ports of the converter. The convention of only allocating memory for the input queues still holds. Fig. 6 shows a converter that has two input queues, and a single output queue.

A converter may contain a set of attributes that are parameters to its processing algorithm. For example, this could be the gain of an amplifier block, or the frequency and phase angle of a sine generator. Since the number of attributes will vary between converters, a unique struct must be declared for each type of converter. This structure will contain all the attributes needed by that converter. An attribute of note is the parent pointer present in all attribute structs, that points to the current converter. This creates a convenient method to access the input and output queues.

Each converter must define a processing algorithm. When the empty converter shell is created, a C function pointer is defined, which points to this algorithm. When a converter executes, the attribute struct of that specific converter must be passed as parameter to the function. This is necessary because there may be more than one instance of a converter. This adds to the re-usability of code in the project, since only one instance of the process needs to exist in program memory, and provides a separation between converter class and instance.

Because the eventual goal of the project is to automatically

generate the C code described here, a structured approach must be taken when configuring a converter. This will simplify the code generation process. Before a converter can be created, an empty instance of it must be declared. A list of pointers to converters is first declared, as an array of type CONVERTER (Fig. 6). Once the list is created, an empty instance of each converter must be declared. This is done by allocating memory for each converter that will exist in the SDF network. A list of CONVERTER pointers is now stored in an array, each pointing to an empty instance of a converter. The following procedure must then be taken to create a new converter:

- 1) *Assign the process pointer.* Associate the converter with a specific process.
- 2) *Allocate memory for the attributes.* As stated earlier, each type of converter will have an attribute struct. Memory must be allocated using the malloc function.
- 3) *Set values of all attributes.* Once an instance of the attributes have been created, a setter must be used to assign values to all attributes in the struct.
- 4) *Allocate memory for the input list.* Because there may be more than a single input to the converter, a list containing pointers to all input queues must be declared using malloc.
- 5) *Allocate memory for an input buffer.* The size of the queue can be determined by means of the algorithm proposed by Lee and Messerschmitt [3].
- 6) *Allocate memory for QUEUE attributes.* QUEUE is a struct containing information regarding the current input queue. These include the start and end address of a queue, and pointers to the head and the tail of the queue. The size of the queue, and a flag indicating if the queue is full, are also included.
- 7) *Set up attributes of the queue.* Assign values to the start address of the queue, and the end address of the queue. The head and tail attributes will initially be set to the start address.
- 8) *Repeat steps 5 through 7 for all input queues.*
- 9) *Allocate memory for the output list.* As discussed earlier, memory will only be allocated for the input queues. During the final step, the value of the pointers in the output list will be set equal to the corresponding input queue.
- 10) *Link output buffers to input buffers.* This final step is separated from the rest, and only occurs after steps 1–8 have been completed for all converters in the SDF network.

Using these rules to create a converter ensures that all instances of all nodes in the SDF network are created in the same manner. This is essentially a copy-and-paste method when hand-coded, indicating that it is well-suited to automatic generation by machine.

At this phase of the project, an automated scheduler has not yet been implemented. Scheduling must be done by hand, and then coded into the program.

IV. HARDWARE PLATFORM

The target platform is the Freescale DSP56311 fixed-point 24-bit digital signal processor.

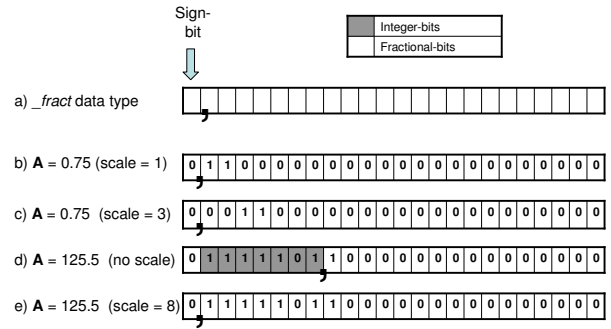


Fig. 7. Examples of the `_fract` data type. a) `_fract` as represented on a DSP. b) A value represented using the `_fract` data-type. c) The same value represented using the `_fract` data-type, where the value has been scaled. d) A value represented using a number of integer-bits and a number of fractional bits. e) The same value represented using the `_fract` data-type and a scale factor.

A few considerations should be kept in mind when porting the generic SDR architecture to the DSP. The first of these is the allocation of memory. When coding under an operating system, memory management schemes are present. When a program requests memory using a malloc function, the operating system will allocate free memory to the program. This functionality is typically not available in an embedded system. A memory allocation scheme must be written specifically for the platform, and for the purposes of this project an alternative allocation function named `dspmalloc` was implemented.

A variety of memory management schemes exist, of which the most common strategies are the first-fit, best-fit and worst-fit schemes. Simulations have shown that both first-fit and best-fit are better than worst-fit in terms of decreasing time and increasing storage utilization [4]. Speed is not an important factor in this project, since memory allocation will only occur during the configuration process. A more important factor would be the simplicity of implementation and the level of memory fragmentation. Because of the different needs of different systems, a method was implemented where the type of memory management can be chosen at compile time. Additional strategies can be implemented at a later stage, and then be selected at program startup. To increase flexibility and portability, the range across which memory may be allocated can also be specified, since the memory layout of all Motorola DSPs may not be the same. Memory will typically be allocated in the top half of available data memory.

The second consideration is that all signal processing must be done using fixed-point arithmetic. The TASKING EDE environment used in this project supports a data type called `_fract`, which can represent a value in the range $(-1; 1)$ using fixed-point representation. This is a useful data type, but it is obvious that values larger and smaller than the supported ranges will most probably occur in a telecommunications network. We need a method to use this data type to represent data in a much wider range.

As stated earlier, the Motorola DSP 56311 has a 24-bit memory interface. The `_fract` data type occupies 24 bits on the DSP. Fig. 7(a) shows how `_fract` data is represented

in the DSP. There is one sign bit and 23 fractional bits. We can scale a floating-point value to within the range $(-1;1)$ by dividing the value by a constant bigger than the value. The denominator should be a power of 2. Dividing by a power of two may be simplified to an arithmetic right-shift. For example:

$$\mathbf{A} = 125.5.0 \in [-128; 128] \quad (1)$$

$$\mathbf{A} = 0.98046875 \times 2^7 \in [-128; 128] \quad (2)$$

\mathbf{A} can now be of type `_fract` since it has a value of 0.98046875. Fig. 7(e) shows how this value would be represented on the DSP. We must remember that this is a scaled value; the actual value of \mathbf{A} is 128 times bigger. We must therefore keep track of the scale factors of all values currently in the SDF network. To accomplish this, each queue will have a scale factor as an attribute, defined at compile time. A sine wave generator, for example, will always produce values in the range $[-1;1]$. The scale of the connecting input queue will then have a scale of 1. We clarify that the scale is the number of integer bits before the decimal comma, including the sign bit. A queue with scale of 1 will be able to represent values in the range $[-1;1]$, while a queue with scale of 8 would be able to represent values in the range $[-128;128]$.

When scaling a value, the resolution of the value is reduced. This is due to the fact that a value will be shifted to the right, thus reducing the amount of fractional bits. To increase the resolution of a data type the number of bits used to represent that value must be increased. The resolution of the `_fract` data type is fixed at 24 bits. The TASKING compiler supports an additional data type, `long fract`, with double precision using 48 bits. Should the need exist for an increased resolution, mathematical expressions can be calculated using double precision and then cast back to 24-bits to be compatible with the single precision data type `_fract`. Alternatively all `_fract` data types can be substituted by `long fract` with minimal effort. For increased flexibility the preferred data type can be specified at compile time, and all fixed-point values and calculations will assume that data type.

Four functions were written to perform the fixed-point arithmetic found in the project. Addition [5], subtraction [5], multiplication and division. These functions all receive two values of type `_fract`, and the accompanying scale factors of these values. In addition to these values each function also receives the scale factor that the answer of the math operation should have. When math operations are performed on two values the resulting answer may be bigger than either value. This means the amount of fractional-bits needed to represent the value may increase. Therefore the number of fractional bits the answer must contain also needs to be specified.

V. TOOLCHAIN

As discussed in section I, a SDR application will be defined using XML. Such a definition will consist of many different converters (also defined in XML) that are configured and then linked together. This XML SDR definition must then be translated to C code supporting the generic SDR topology discussed in section II. XSLT is used for this purpose.

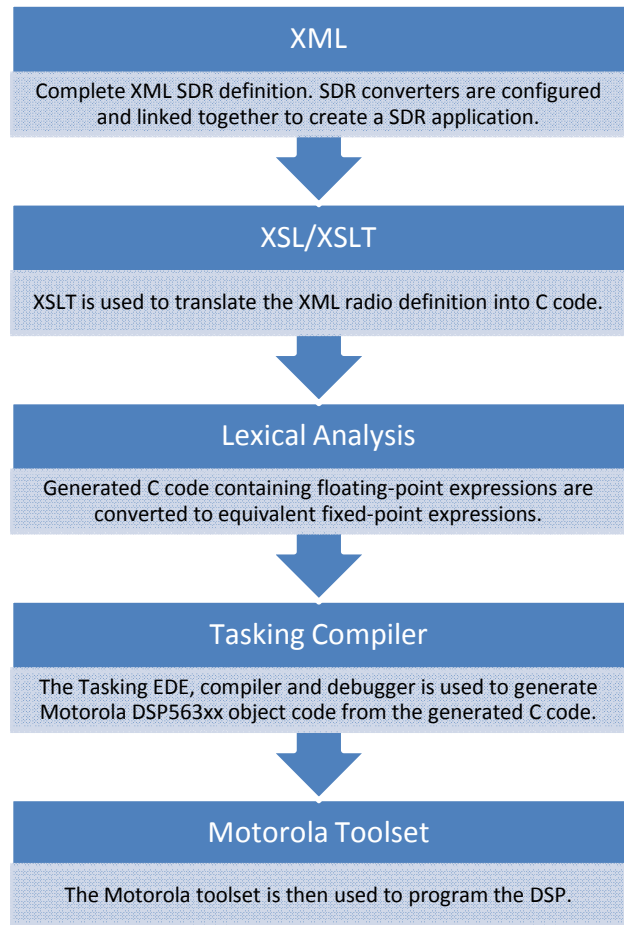


Fig. 8. The toolchain used to produce Motorola object code.

Each XML converter definition contains a process that describes the action of that converter. This consists of a block of code that will perform the action of the converter. To increase portability and flexibility of the XML converters, the ANSI-C style will be used in these process blocks. There is now an obvious inconsistency between this style of C code, and the style used to perform the fixed-point calculations in the previous section. All floating-point calculations have to be replaced by custom written fixed-point functions. For this purpose a lexical analyzer and parser will be used to transform floating-point expressions into fixed-point expressions.

The TASKING DSP56xxx C/C++/EC++ Compiler is able to generate code for the Motorola DSP 563xx. TASKING is a fully compliant ANSI-C compiler. Various optimization techniques exist to enhance performance. The generated object code is then loaded onto the Motorola DSP 563xx using the standard Motorola toolset.

VI. RESULTS

To verify the functionality of the hardware implementation of the SDR architecture, a simple SDR application was created, namely an AM modulator and demodulator. Fig. 9 shows a modulating signal that is modulated by multiplying it with a carrier signal. The AM modulated signal is then demodulated using synchronous (coherent) AM demodulation. The values

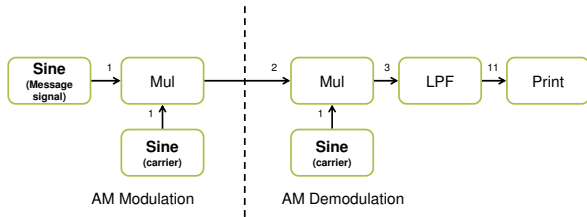


Fig. 9. A simple AM modulation-demodulation schema using synchronous demodulation. The scale factors of the input queues are also shown.

are then printed to a file and imported to Matlab. An equivalent application was created using Matlab, in order to verify the accuracy of the DSP implementation. The network consists of the following nodes or modules:

- 1) *Modulating signal*: A sine wave with reference frequency $f_o/f_s = 0.01$. Sine waves are generated by means of direct-digital synthesis (DDS) [7] using a 1024-entry look-up table. The value of the phase accumulator is:

$$\Delta\phi = 2^R \left(\frac{f_o}{f_s} \right) \quad (3)$$

where $R = 10$.

- 2) *Carrier signal*: Sine wave with reference frequency $f_o/f_s = 0.1$.
- 3) *Multiplication*: Multiplies two input samples and produces a single output.
- 4) *Filter*: second-order IIR filter (Direct form II realisation, $N = 2$) [6]. The filter coefficients were calculated using the `butter` function in Matlab.
- 5) *Print*: This module prints a sample to the screen and to an output file.

Fig. 10 shows the results for both the DSP and Matlab versions of the modulation and demodulation process. The waveform produced by the DSP SDR is identical to the reference signal produced in Matlab. The ripple in the demodulated signal is due to small frequency separation between the modulating signal and the carrier, and the low order of the filter.

VII. CONCLUSION

The purpose of this project was to create a SDR architecture that is highly reconfigurable and to implement that architecture on a hardware platform, for which the Motorola DSP563xx family of DSPs was chosen. It was shown that special consideration needed to be given to the embedded environment, where custom-written memory allocation algorithms were required. Also, the use of fixed-point processing required the implementation of custom mathematical routines. The results obtained illustrate that SDR applications can be implemented using this architecture on the DSP platform, and that waveforms retain numerical accuracy.

The work described in this paper focused on the creation of the C-based architecture for the DSP platform. The C code was designed in such a way that it lends itself to machine generation. In the next phase of the project, an XSLT parser

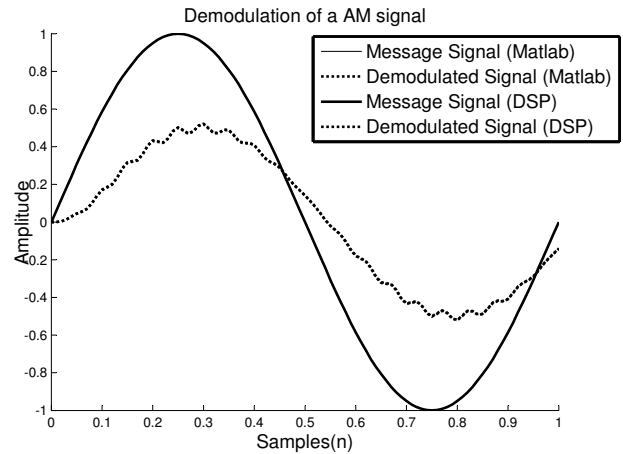


Fig. 10. Results obtained from the Matlab and DSP versions of the demodulation of an AM signal. Only a single signal is visible since the results are so closely matched.

will be implemented to automatically generate SDR C code for the DSP563xx from generic XML descriptions of radios.

ACKNOWLEDGMENT

The authors would like to thank the Telkom CoE programme for funding this project.

REFERENCES

- [1] Cronje, J.J., "Software Architecture Design of a Software-Defined Radio System", Master's thesis, University of Stellenbosch, 2004.
- [2] Brady, R., "A Cross Platform Framework For Software-Define Radio", Master's thesis, University of Stellenbosch 2007.
- [3] Lee, E.A and Messerschmitt, D.G., "Synchronous Data Flow", *Proceedings of the IEEE*, 1987, Vol.75, NO.9, pp. 1235-1245.
- [4] Silberschatz, A., Galvin, P. and Gagne, G., *Operating System Concepts*, 7th ed, Wiley, 2005.
- [5] Oberstar, E.L., *Fixed-Point Representation & Fractional Math*, <http://www.superkits.net/whitepapers.htm>.
- [6] Proakis, J.G and Manolakis, D.G., *Digital Signal Processing Principles, Algorithms and Applications*, 3rd ed, Prentice-Hall, 1996.
- [7] J. Vankka, "Methods of Mapping from Phase to Sine Amplitude in Direct Digital Synthesis," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 44, no. 2, pp. 526-534, March 1997.

Wouter Kriegler obtained his BEng degree from the University of Stellenbosch in 2006. He is currently studying towards an MScEng degree at the same university, and is part of the Signal Processing Laboratory at the Department of E&E Engineering.