TH-2.2 - Audio and Control: Simulation to Embedded in Seconds

N. Bentall

nathan.bentall@oxford-digital.com

Abstract:

In real-time audio or control applications, algorithms are typically developed using simulation tools such as Matlab/Simulink. Traditionally, this development is followed by a conversion to fixed point, and a labour-intensive manual optimisation and conversion to the assembly code of a signal processor or ASIC/FPGA core. The prudent will also run a series of test vector comparisons to check the conversion process. By using custom simulation models, and an automated process of net-list extraction and code generation, these time consuming steps may be eliminated, going from simulation to running embedded DSP code in seconds.

1. INTRODUCTION

The idea of graphical representation of computer programs has been around a long time. Though probably earlier references could be found, Sutherland's 1966 thesis [1] details this approach, and opens with the statement “Men find pictures useful for communicating with computers as well as for conversing with each other”. Flick through any DSP text-book and you will likely find many pictorial representations of the algorithms and systems, such diagrammatic descriptions used to convey the concepts most succinctly. While a graphical representation does not necessarily offer the greatest insight for every algorithm, it is certainly an effective method for many, particularly in the area of audio and control systems. Visual Programming Languages (VPLs) (using graphical representations of procedures and/or algorithms to define a computer program) are often used within the ‘Model Based Design' (MBD) paradigm. Within a MBD approach, attempts are made to model closely all relevant parts of the system, often including external physical factors. A solution (algorithm or process) can then be rapidly developed and tested within the simulation environment and without recourse to costly real-world tests (at least for a substantial part of the development process).

For this reason, MBD methods are frequently used to reduce time-to-market and design costs, at least in part by reducing the risk factor associated with errors made during the implementation phase.

While a MBD approach has long been in common use in safety-critical applications such as automotive and aerospace (using tools such as Boeing's Easy5, which began life in the mid 1970's), there is growing interest both in wider use MBD, VPLs and also (of relevance to this paper) in conjunction with Automatic Code Generation (ACG) direct from the model. Partly, this is driven by cost constraints, with the faster time-to-market offered using an ACG approach, but ACG also potentially offers a reduction in implementation error and ultimately, more reliable systems which match exactly the intended function (or at least match the model), and by combining an evolved work flow, modern simulation tools and specifically developed signal processing architecture/tool-chain, there is real potential for products and development cycles which are faster, better AND cheaper.

1.1. Brief History

Of particular interest to the author is the use of VPL methods for the development of audio algorithms.

In 1984, a small Oxfordshire company ‘Solid State Logic’ (SSL) began work on a new concept for the design of digital audio mixing consoles. (For those unfamiliar with SSL, they were described in 1988 Drummond [2] as being to mixing consoles what ‘Hoover/Sellotape’ were to their respective industries). The SSL work established something of a manifesto for (amongst other things) the philosophy of VPLs as applied to algorithm development and automatic code generation. The work was published in a series of papers including Eastty [3], McCulloch [4], Kentish [5], and laid many of the foundations upon which later tools, (either directly or indirectly), were built. In the late 1980's, a tool called (first ‘Patcher’ but later) MAX/MSP emerged from the research group, IRCAM. MAX provided a graphical interface for describing audio algorithms, intended mainly for use in sound synthesis and music composition. The software (in a much-evolved form) is still available today. In 1995, and based on the philosophy of the earlier SSL work, Eastty et al [6] presented details of a Sony large-scale professional audio mixer (Figure 1) which was used in both recording studio and mission-critical broadcast environments.

Figure 1 Sony OXF-R3 Mixing Console

In its day, the hardware and processing capability of this machine was formidable. Consisting of 400 Digital Signal Processors (DSPs) connected in a massively-parallel array (Figure 2), the DSPs were programmed entirely using a VPL approach combined with Automatic Code Generation.

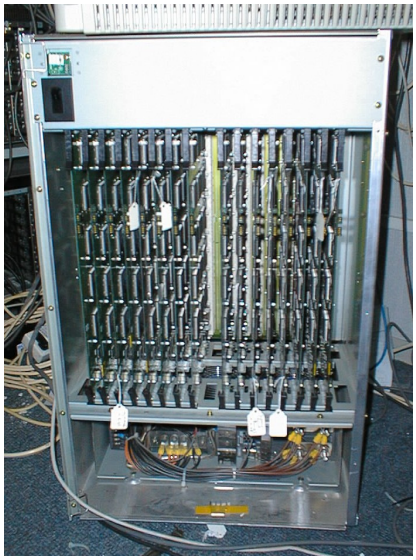


Figure 2 Sony OXF-R3 Processing Rack

To generate a 'program', a representation of the signal processing algorithm was drawn using schematic symbols for multiply, add, delay etc. in most cases without any need on the part of the algorithm developer to consider the details of implementation. The basic arithmetic operations could be combined using multiple levels of hierarchy, to form sub-systems such as second-order filter sections, equalizers, channel strips and eventually, a design for the entire mixer. A 'netlist' representing the connectivity of the schematic was exported and used as input to a 'compiler' which would attempt to fit the specified DSP algorithms into the processing available. Some examples of the schematics used can be found in Frindle [7] which details the development of an audio dynamic range processor.

2. ALGORITHM DEVELOPMENT

For audio algorithm development, the 'quality' of the algorithm is often very difficult to quantify objectively. For this reason, audio algorithm development frequently relies, on an empirical approach (in addition to more readily quantifiable measurement techniques). A very short idea-hear loop can be invaluable, allowing a designer to quickly try ideas and listen to the results, make adjustments, and try again.

For both audio and control systems development, the complexity of interaction between various sub-systems can also make accurate analytic predication difficult. For both applications, simulation can be a very useful tool.

2.1. Simulation

Thanks to the advances made in personal computing power, algorithm development for many useful applications using this approach no longer requires a fan-cooled processor rack to provide the ability to simulate/run the algorithm, and there are numerous desktop VPLs available [8]. Algorithm ideas can be quickly prototyped and auditioned using these types of tools. One of the more popular commercial applications is Simulink from Mathworks.

Simulink offers many tools and an easy-to-use interface for developers wishing to quickly try out an idea or check the efficacy of some new/altered method, or test a new optimisation of a known method. These tools include the ability to hear audio results, extensive graphing capability, the ability to perform spectral analysis and capture data etc. without requiring the developer to become too involved in the details of the implementation, and thus avoid many of the irritating and petty troubles of other programming languages such as syntax, variable declaration/data management, displaying data or outputting it to a sound card or gathering data from an acquisition board.

While such details can be interesting and challenging in themselves, when the goal is to develop an algorithm, such tasks can distract from the main objective.

2.2. Benefits of Simulation

Simulation tools that allow the developer to sufficiently abstract away the implementation minutiae can significantly enhance productivity and reduce both development time and frustration, and may even result in the discovery of new and innovative solutions that might have not been attempted were the process too difficult or time consuming to try out.

For control systems, it is also now practical to include hardware-in-the-loop simulations. Usually using additional hardware, the simulation can be interfaced to a real control system, allowing a further stage of verification before the algorithm under test is targeted at the embedded system.

Without some kind of easy-to-use simulation environment, the developer is forced to hand-code ideas within the target system to test and check performance. If the results are not as hoped, the problem could be with the concept (algorithm), the implementation or even the hardware on which it is running. Finding the cause can be very time consuming.

In summary, simulation is beneficial because it allows the developer to try and check ideas in a low-cost, low-risk environment and encourages innovation through a what-if approach.

3. IMPLEMENTATION

Once the algorithm is believed to be working within the simulation environment, invariably it must be transferred to a target system, often some kind of embedded processor. There are a number of possible approaches.

3.1. Hand-Coding

The traditional method, and probably the one most commonly used, is to hand-code the algorithm using a text-based programming language such as C or assembler. This can be very time consuming and error prone.

In [9], an estimate of coding productivity is given at 6 lines of hand-generated code per day for a servo control system. While it is acknowledged that lines of code per day is a rather tenuous measure (and doubly so for highly optimised assembler, where one can spend many days trying to *reduce* the number of lines!), this is not dissimilar to the authors experience for hand coded-&-optimised assembler for audio applications.

During a recent port of a commercially licensed DSP algorithm to an embedded DSP target, the reference algorithm was provided as C-code, but the final implementation was written in assembler. Looking at one sub-algorithm, which ended up at around 100 lines (excluding parameterisation aspects, comments and some framework) of executable assembler code, implementation took about 4 weeks. This translates to around 5 lines per day. As part of the port, the sub-algorithm was input into Simulink using the generic block set, which took around 1 day.

Much of the time spent in the implementation involved verification and correction of the inevitable coding/translation errors. Although it was a successful project, the time spent coding could have been significantly reduced (in this case, by at least 4:1) if only some kind of automated implementation process existed...

3.2. Automatic Code Generation

From a developers point of view, the 'Holy Grail' (at least in the present context) might be to complete the algorithm development within a simulation environment, and then, with the click of a button, have that algorithm accurately and efficiently converted to running code for the target system, safe in the knowledge that the implementation generated would match precisely the behaviour seen in the simulator.

In fact, there are various solutions which offer, to varying degrees, this kind of functionality. Without exception, all have limitations or restrictions which unfortunately mean there is no universally applicable solution. For the purposes of this discussion, only Simulink based solutions will be considered, though it is acknowledged that many others exist.

The solutions may be divided into two basic sets – those that rely on using special Simulink Blocksets (in Simulink parlance, a Blockset is a library or set of pre-configured processing elements or operations, for example, multiply, add, delay, bi-quad filter section etc.) and those which output another standard language such as C or Verilog.

The Real-Time Workshop add-on from Mathworks, takes the standard-language approach. It attempts to generate generic C code, sometimes targeted at a specific architecture, for example [10] introduces the use of Real Time Workshop to generate C code to target a Texas Instruments DSP. Some common errors associated with this type of automatic code generation are reported in [11], being type definition and scaling errors, which can ultimately result in a difference in real-world behaviour between the simulation and the target system. For the developer, this means a requirement to test (sometimes extensively) the generated/compiled code to verify correct operation. While this kind of solution can reduce the effort required for the initial translation, the requirement for testing and debugging appears still present.

The situation can be improved by the supply of a special Blockset that is designed specifically for the target device. Such Blocksets are available from several DSP vendors including Analog Devices and Texas Instruments (and also for the TinyDSP Core

from Oxford Digital detailed in this paper). By limiting the range of blocks a developer can use to a sub-set of specially made blocks, the task of guaranteeing a match between target and simulation behaviour becomes at least easier (and in many cases, may not be practical without this approach).

Another problem with Automatic Code Generation is that most processors were not designed with this kind of VPL programming in mind. This can result in a very complex compiler and/or models, inefficient implementations, or worse, make it (almost) impossible to achieve a reliable match between the models and the target.

Unlike most processor architectures, the TinyDSP Core presented here was designed specifically for use within an automatic code generation flow (and never to be programmed in assembler). For this reason, the generated code is highly optimised and the simulation results are bit-accurate.

3.3. FPGA/Vendor Blocksets

Another automatic code generation approach is to generate code/designs which describe the hardware interconnection within a target silicon technology. Several of the FPGA vendors provide Simulink Blocksets to target their devices. An example of some of the blocks supplied with Xilinx System Generator Blockset is given in Figure 3.

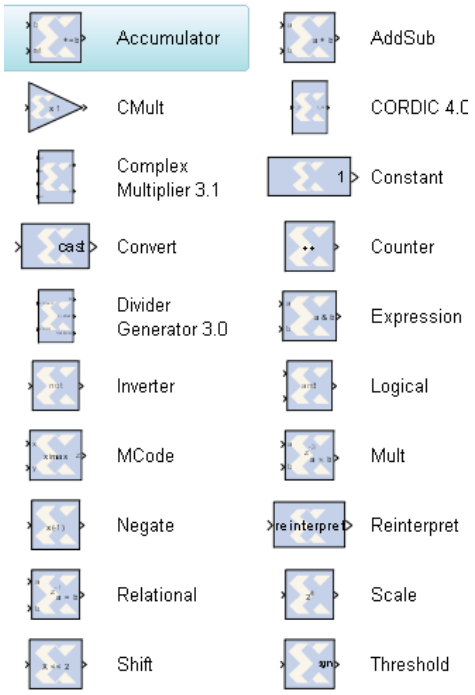


Figure 3 Example Xilinx System Generator Blocks

This approach can provide great flexibility as almost any operation can be implemented and very significant work-flow benefits for certain algorithms, particularly where the developer is lucky enough to find suitable pre-written macro-blocks for use within the design (Filter blocks etc). In [12], a person-hours comparison was presented between a hand-coded-HDL versus System Generator flow (employed in the implementation of a SATCOM waveform generator), and concluded an impressive >10:1 improvement in design time effort.

It can, however, be difficult to make an efficient implementation for many use cases, particularly for control and audio applications. Typically, the achievable clock rate in an FPGA/ASIC is much higher than the input data rate (perhaps a 100MHz clock rate compared to a 44.1 kHz sample rate, for example). In order to make an efficient implementation, it is usual to attempt to re-use the 'expensive' (in terms of silicon area) elements such as the multiplier, and gather data storage into ram-blocks. Although this can be created with tools such as System Generator, the end result often ends up looking similar to the design of a DSP processor core (indeed it would not be unreasonable for the System Generator Blockset to *include* a TinyDSP Core block).

Another disadvantage of the vendor-specific blocks is that they are (unsurprisingly) specific to particular FPGA vendor's devices and in general, not easily transportable to an ASIC flow or between FPGA vendors. This is understandable from a commercial view-point, but can be a nuisance as a customer.

Sharing many of the benefits that contributed to the 10:1 effort ratio in [12], a similar effort reduction ratio can be expected with TinyDSP Core/Simulink flow presented here.

4. THE TinyDSP Core FLOW

The preceding sections have provided some background, and hopefully some insight, into the reasoning behind the development of the TinyDSP Core, associated simulation and automated code generation design flow. By designing a DSP core specifically for automatic code generation from a VPL representation of the algorithm required, excellent efficiency, both in terms of silicon area/FPGA usage and workflow can be achieved.

A simplified work-flow diagram is given in Figure 4. There are two parallel paths. On the left hand side is the DSP algorithm flow, and the right-hand side is the ASIC/FPGA flow.

They both have a dependency on the parameter 'TINY_DATAWIDH'. This parameter represents

the number of bits used in the calculations, both for the Simulink models and the hardware. In the Simulink environment, this data width parameter can be changed at any time, and may form part of the algorithm design.

In practice, it is usually set somewhere between 24 and 32 bits for most audio applications (though 48-bits has been used for a very high-end audio filtering implementation).

The numbered elements will now be described in more detail.

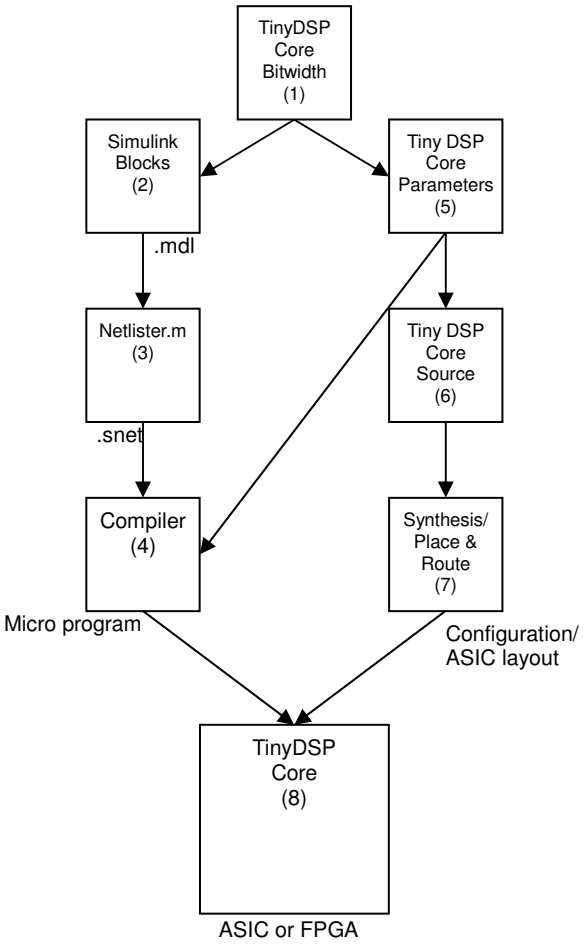


Figure 4 Simplified workflow diagram.

4.1. TinyDSP Core Data Width (1)

Because the ultimate target is the fixed-point DSP core, in order for the simulation behaviour to match the target exactly, fixed-point data types have been specified during the library creation (making use of the Simulink Fixed-Point package). TINY_DATAWIDTH must be initially assigned before running a simulation or generating an output.

For a 24-bit DSP core, the user would assign this value by typing TINY_DATAWIDTH=24 at the Matlab console. This globally sets correctly the precise numerical behaviour of all the blocks in the Blockset.

(Note that using the Fixed-Point tool in Simulink, the data types may be overridden for simulation purposes, for example, all types may be set to 'doubles' or 'singles'. This can speed up algorithm development in the early stages.)

4.2. Simulink Blocks (2)

Programming the DSP device is achieved using TinyDSP Core Blockset. The DSP program is entered using symbolic representations of basic mathematical and logic operations. No other source-code or framework code of any kind is required. The Simulink model *is* the source-code for the implemented algorithm.

There are a few common-sense rules for the use of these blocks.

1) A single output must feed any given input (single driver for any given node)
2) Any output must connect via the library blocks to an input or source (no floating inputs)
3) Other blocks outside of the library may be used for analysis/simulation purposes, but only blocks from within the library will ultimately generate running code in the target system.

There are 41 primitive blocks from which the program can be made; 16 of these are multiply instructions with different scaling factors. Treating these 16 multiply-scale instructions as a single overloaded multiply, there are 14 basic DSP instructions (Figure 5) plus a few symbols for audio and control IO.

Any available Simulink blocks may be used and may remain within a design for testing/simulation purposes, but blocks outside of the TinyDSP Core library will be ignored by the tool chain when converting the design to run in the target processor.

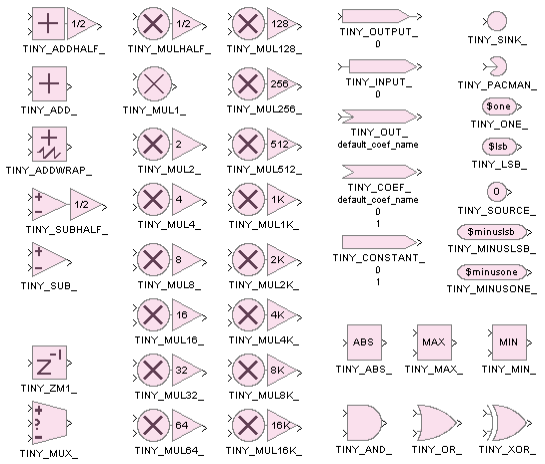


Figure 5 DSP primitives in the TinyDSP Blockset

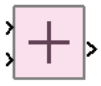
4.3. Blockset Operation

Some of the blocks have additional parameters which can be changed from their default values; for example, a constant may have a numerical value assigned, and an input, a channel number. The operation of key blocks will now be briefly described.

ABS

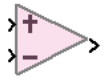
Takes the absolute value of the input – e.g. turns -5 into 5.

ADD



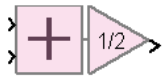
Add with saturate. Adds two values. If the result would be greater than the word width can represent (in either positive or negative sense) the result is limited to the largest value possible (positive or negative).

SUBTRACT



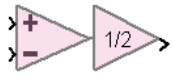
Subtract with saturate. Like Add, but subtracts one value from the other instead of adding.

ADD-Half



Adds two values and divides result by two. No saturation can occur.

SUBTRACT-Half

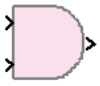


Like Add-Half, but subtracts one value from the other instead of adding.

ADDWRAP

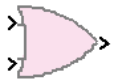
Adds two values. If the output exceeds the representation of the word-width, the result ‘wraps’, rather than saturating.

AND



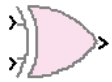
Bit-wise logical AND of two inputs (i.e. bit 0 of input A is AND-ed with bit 0 of input B to make output bit 0 and so on for each output bit)

OR



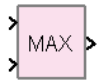
Bit-wise logical OR of two inputs.

XOR



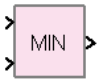
Bit-wise logical Exclusive OR of two inputs.

MAX



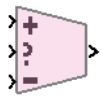
Output is the Greater (more positive) of the two inputs. (max of (-3, -5) is -3)

MIN



Output is the lesser (more negative) of the two inputs. (min of (-3, -5) is -5)

MUX

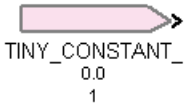


Multiplex of two inputs dependent on third input. If the control input is greater than or equal to zero, output gets value of '+' input, else output gets value of '-' input.

DELAY

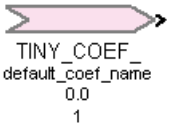
Single-sample delay (at the audio/control sample rate). In hardware, these are implemented using automatically rotating memories - from an instruction point of view, these are 'for free' as the address calculation is performed in hardware.

CONSTANT



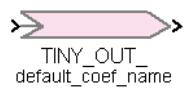
A constant value set at compile-time that cannot be changed while the program is running. The block accepts two parameters, the value (default 0.0) and the number of integer bits (default 1) used to represent the value.

COEF



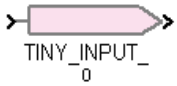
A volatile constant (from DSP point of view) that may be changed by external control software. The block accepts three parameters. The first is the coefficient name. This will appear in the output files, and by setting this to some pre-prepared values, a quick link may be made with the Control Application in section 6. The second parameter sets the initial value (default 0.0). The third assigns the number of integer bits used to represent the value, effectively providing a scaling factor.

OUTPUT



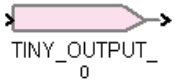
A value that can be read by control software (e.g. for meter display value). This block accepts a single parameter, the coefficient name. This name is processed by the compiler and can be used to refer by-name to the memory location which the value represents.

AUDIO INPUT



Numbered audio channel input. The output value is updated each sample-period.

AUDIO OUTPUT



Numbered audio channel output. At the end of computation for each sample, the value of the output ports are updated.

ZERO

A source of the numerical value '0'.

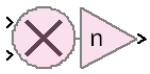
Code Eater

Similar to the Stub, but anything connected to the code eater gets 'eaten' by an optimisation process (assuming its calculation does not contribute to an input elsewhere in the programme).

Output Stub

All outputs must feed an input and cannot be left 'dangling'. If, during development, the user wishes to keep some piece of the DSP but its output is not yet used, this Stub can be connected to its output. The DSP code will be retained in the object code.

MULTIPLY



Multiplies the two m-bit inputs to make an intermediate wide result with (2m -1) bits which is further multiplied (bit-shifted) by value of 'n'; n can take any one of 16 pre-determined values, and is called the scaling factor. The value n is fixed at compile time and is controlled by choosing the symbol with the required value (graphically, the triangle represents a linear gain value).

VERSION

Inclusion of single version block is a requirement of the compiler. It should be connected only to a 'code eater' block.

4.4. Block Implementation

Each block within the Blockset is implemented in Simulink as a masked subsystem. Beneath each of the symbols the behaviour for simulation is defined using the standard Simulink primitives.

An example is given in Figure 6 of the MUL32 block.

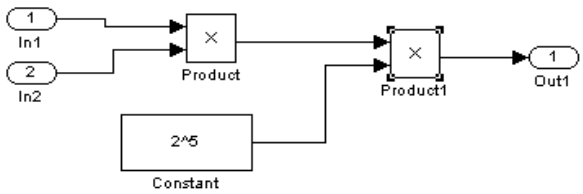


Figure 6 Inside the MUL32 block

Within the Simulink environment, the shift (this being a MUL32) is implemented by multiplying by a constant of 2^5. The data types for each operation (multiply, add etc.) have been specified as fixed point types whose parameters depend upon the global TINY_DATAWIDTH variable.

Each of the blocks also has a special 'tag'. This is a text field which is attached to the block within the library. It is this tag which identifies the type of block to the process which extracts the connectivity of the blocks and ultimately the compiler which generates the final code, for example, the MUL32 block has the tag 'TINYPRIMITIVE_MUL32'.

4.5. Netlist Extractor (3)

In order to convert the simulation design created in Simulink to object code, first the blocks and their connectivity must be extracted from the Simulink model (.mdl file). This is achieved using a custom .m file which is run within Matlab. One of the key operations within this .m script is the 'find_system' command. This is a built-in Matlab command which can return an array of handles to all objects within a design. Each of the handles is then processed, with some checking for particular types which is based on the special tags attached to each block. The processing includes inspection of each of the blocks input and output ports to extract the connectivity between blocks.

Once the pre-processing is complete, an intermediate file (with suffix .snet) is output which contains a complete, though not optimal, description of the type and connectivity of each of the blocks in the design (including elements which will not ultimately generate code, such as blocks used for simulation purposes).

4.6. Compiler (4)

'The' compiler actually consists of a number of steps, all of which are carried out automatically. The first job is to convert the .snet to an intermediate netlist format (for historical reasons), a spice netlist. The compiler then examines this netlist and ultimately generates object code. For further information on the compiler process, see [13]. Some additional tools generate a Windows DLL which can be loaded by the graphical control application detailed later. Included in the DLL is the object code which runs in the TinyDSP Core. The compiler also generates output files which can be used to load the TinyDSP Core via other methods (for example from a host micro-controller, or a state-machine that can read directly from a serial ROM). For a fixed-function ASIC implementation, the TinyDSP Core executable can also be included as pre-programmed ROM within the ASIC.

An algorithm may require coefficient value changes at run-time. Connection between the algorithm's coefficients and the controls on the graphical user interface is made by the use of special coefficient names (or for complex coefficient calculation, c-code can be included which runs as part of the control application). The memory locations are also reported by the compiler in text files which can be included in the build process of a host micro-controller executable.

4.7. TinyDSP Core Parameters (5)

A text file, ‘tinydefs.vp’, contains all the necessary parameters to tailor the TinyDSP Core to a specific requirement and is used as input to the TinyDSP source code. Key parameters are:

- The Data width (the precision used for internal number representation)
- The number of instructions implemented (depends on the achievable clock rate of the target architecture and input/output data sample rate)

- The data memory size (depends on how much is needed for the algorithms to be implemented)

A very important point to note is that changing these parameters does **not** require any change to the algorithm source code (the Simulink model).

The only parameter which affects the numerical output is the data width described in section 4.1. All other parameters may be changed to suit the silicon architecture or meet gate-count targets.

(Of course, if the memory size or instruction count available is reduced below the minimum required to implement the algorithm, the compiler will throw an error.)

4.8. TinyDSP Core Source (6)

Because of the good level of abstraction provided by the graphical programming interface and the optimising compiler, it is easy to make efficient programs without intimate knowledge of the internal operation of the DSP core itself, but an overview is given here. A generalised block diagram of the core is shown in Figure 7.

The TinyDSP Core source code consists of parameterised Verilog HDL, and can be compiled for a target architecture (FPGA or ASIC).

Key features of the DSP core include:

- Fixed-point
- Low gate-count and low power
- No 'branch' instruction (fixed program flow) keeps gate count and power low – processor can't 'crash'
- From 128 to 8192 instructions per sample (depending on core parameters – this is a HDL-compile-time setting and cannot be altered once 'in silicon')
- Clock_Freq=Sample_rate * Instruction_count
- Data_Memory_Access_Rate=Clock_Freq * 4
- Single instruction multiply-add, allowing 5 cycle second order section IIR filter
- Built in shift and limit circuitry in every instruction allows graceful numerical limiting (as opposed to wrapping/overflow)
- Program controlled data address calculation in-hardware (rotating memory area) for low overhead IIR and FIR filters (delay lines can be implemented without requiring a 'move' instruction)
- Address Memory from 512k to 8k, 7 to 11 bit words
- Data Memory from 128 to 2k, 16 to 48 bit words (depending upon the chosen data-width)
- Upto 32 independent input & 32 output channels.

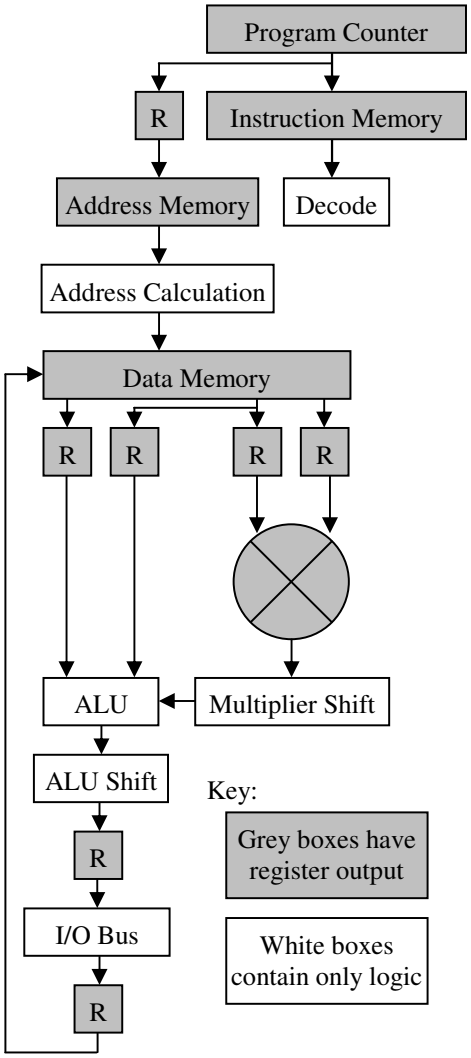


Figure 7 General Block Diagram of TinyDSP Core

4.9. Internal Number representation

Although the internal representation is signed-fractional (1 sign bit and the rest of the bits are considered fractional bits), of course numbers with value greater than '1' can easily be represented simply by accounting for shifts in the radix point using a multiply with scaling factor (the MUL-N blocks detailed earlier). For seasoned DSP coders, this is already obvious, but for the less seasoned, or those more used to floating point units, a brief discussion follows.

A notation scheme of i.f will be adopted, where i represents the sign and integral bits and f the fractional bits. Using this notation to represent a 24

bit number with one sign/integral bit and 23 fractional bits yields a '1.23'.

Internally, a multiply of two m-bit numbers generates a (2m - 1) bit wide result. All the scaling factor does is to select which m-bits out of the wide result are selected to make the m-bit output. By default (using the multiply with x1 scaling) it takes the top bits (the extra sign bit is automatically discarded) such that 1.23 * 1.23 = 1.23 (schematic symbol shown in Figure 8).

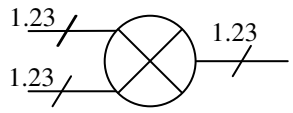


Figure 8 Mul1 for 1.23*1.23=1.23

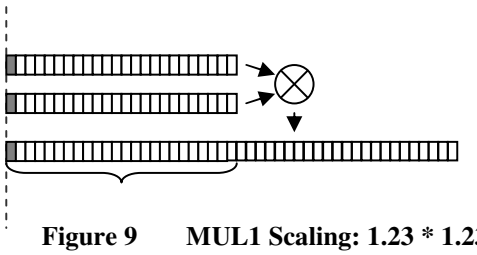


Figure 9 MUL1 Scaling: 1.23 * 1.23

Figure 9 makes an attempt to represent this visually. Each rectangle represents a single bit. A grey rectangle represents a sign bit for a given representation. The horizontal brace represents which bits will make the final output.

When the multiply is between a pair of 2.22 numbers, the wide result is a 3.44 (when the duplicate sign bit is removed), so if, as is often the case, the output of the multiply is to be the same scaling as the inputs (a 2.22), a left-shift of 1-bit must be applied before the lower bits are discarded. This is shown in Figure 10 and is achieved in the schematic by using the 'mul2' block.

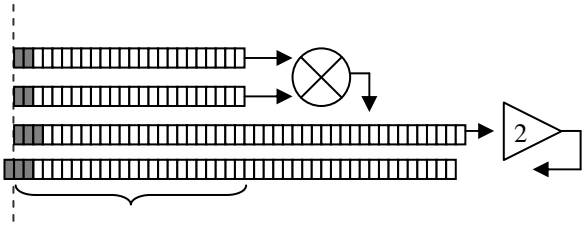


Figure 10 MUL2 Scaling: 2.22 * 2.22

If in any doubt, simply set all inputs to '1' in the chosen representation(s) and check that the result is also '1' given the expected output representation. This can be easily achieved using the Simulink Blockset.

This is the only scaling required and the only instructions this type of scaling affects are the multiply instructions. All other instructions, by their nature, behave identically and require no scaling. (Note that during the shift, should an overload occur, it is automatically limited to generate the final result)

4.10. P&R, TinyDSP Core (7) & (8)

These processes are necessary if using a custom silicon implementation, either FPGA or ASIC. If using a commercially available device, these steps have been completed already.

5. USEAGE

In summary, the TinyDSP Core with Simulink Blockset usage process is:

1) Using the Simulink library of blocks, create the algorithm in Simulink.
2) Set/check the data bitwdith.
3) Test the algorithm's performance in Simulink (optional)
4) Run the Matlab script 'tiny_netlist.m', to extract a netlist from a Simulink model.
5) Run the compiler tools to generate executable code for the TinyDSP Core
6) Load the code in the target and run

If the aim is to target a custom ASIC or FPGA, additional steps are required to instantiate the TinyDSP core in the target architecture. This is only required (usually) once per target hardware type.

For implementation of the TinyDSP Core in silicon:

1) If required, modify the default TinyDSP Core parameters
2) Instantiate the core within the target architecture
3) Run place & route.

5.1. Creating a Schematic

The parts detailed in section 4.2 can be placed to form an algorithm. Figure 11 shows a simple, but complete design. A triangle wave is generated at the output of TINY_ABS_1 using an integrator implemented with a wrapping adder. This triangle wave is multiplied by audio input channel 0 and used to make audio output channel 0. The result at the output is an amplitude-modulated version of the input.

Simulink elements can be added for simulation purposes, as per Figure 12 which shows the simple design with additional blocks to listen to some audio processing use Simulink audio IO.

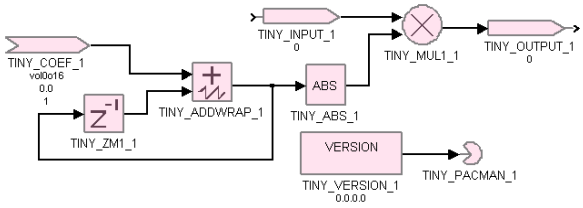


Figure 11 Simple, but complete design

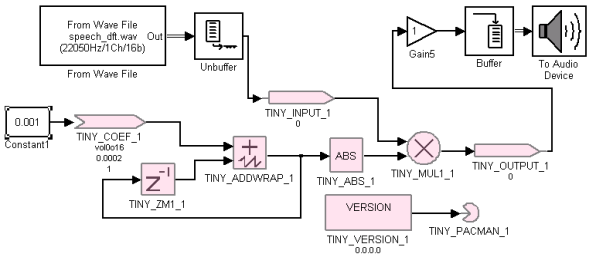


Figure 12 Simple design with Simulink simulation blocks

After inputting the schematic in Figure 12, and ensuring that the TINY_DATAWIDTH has been set, the result can be heard by starting the simulation. Other blocks for analysis/verification can be added as-required. To verify the shape of the saw-tooth wave, a Simulink 'scope' block may be connected. Running this simulation, the scope display appears as in Figure 13.

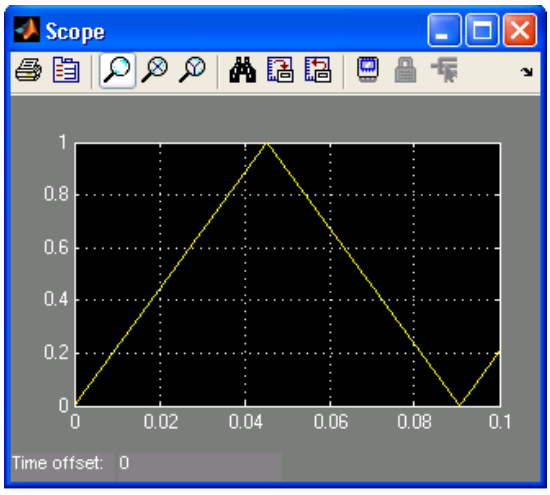


Figure 13 Triangle wave output viewed by Simulink Scope

At any time, tiny_netlist.m can be run, passing the name of the Simulink model (same name as the .mdl file-name) to generate a new .snet file. To achieve this from the Matlab console, and assuming the design has been saved/named 'fade_in_out', type:

```
tiny_netlist('fade_in_out')
```

A .snet file will be created.

The next step is to run the compiler process. This is simply achieved by running a short batch file which first calls a program to convert the .snet into a .net (a spice netlist) and a second program TinyBP.exe which ultimately generates all required output files including a .dll which the graphical Control Application can load directly.

6. CONTROL APPLICATION

In order to facilitate algorithm development, a generic front-panel containing hundreds of assignable user controls was made.

The application runs on a PC, and connects to the hardware development board via a USB-to-I2C converter. A screen-shot is given in Figure 14.

Each of the knobs, sliders and switches can be assigned to control some aspect of the DSP. Coefficients can be calculated on the host PC and communicated to the running target hardware via the USB/I2C interface on the development hardware.

As can be seen in Figure 14, 8 vertical 'strips' can be viewed simultaneously, while additional 8-strip pages can be accessed using the row of numbered tabs across the top. There are 16 pages in total, giving 128 strips. Running down the right-hand side are the 8 'scene' tabs. A 'scene' contains a complete positional setup of each control on all 128 strips. This allows 8 entirely different DSP parameter sets to be held and edited. Built in is the ability to copy from one scene setup to another. In this way, a current scene as a work-in-progress can be copied to another scene, incrementally (or indeed entirely) altered, and quickly compared to the original scene. This kind of real-time control can be useful for optimising or tailoring the algorithm's coefficients to a particular application (for example, the frequency and gain settings of a tone control to suit a particular speaker)

The complete settings can be saved to a file for transfer from a host micro-controller at boot-time. For a typical audio application of these tools see [14].

6.1. Export parameters to real device

When using the TinyDSP Core within a product such as a mobile telephone, it is usual for the host CPU in the phone to control the loading of object code and parameters/coefficients.

The TinyDSP Core object code is usually loaded at boot time, while the parameters are changed according to user preferences. To facilitate this, an

additional console-based application takes as input the settings created with the Generic Control Application and exports them for inclusion (as a c header file) in the build process of the other (host) CPU.

With the capability for many parameters, and potentially 8 scene settings, this application also performs some data compression such that the minimum requirement (in difference terms) is output. For example, if 100 parameters were the same across all 8 scenes, and only 4 parameters differed between all 8 scenes, the entire data for all 8 scenes is not required.

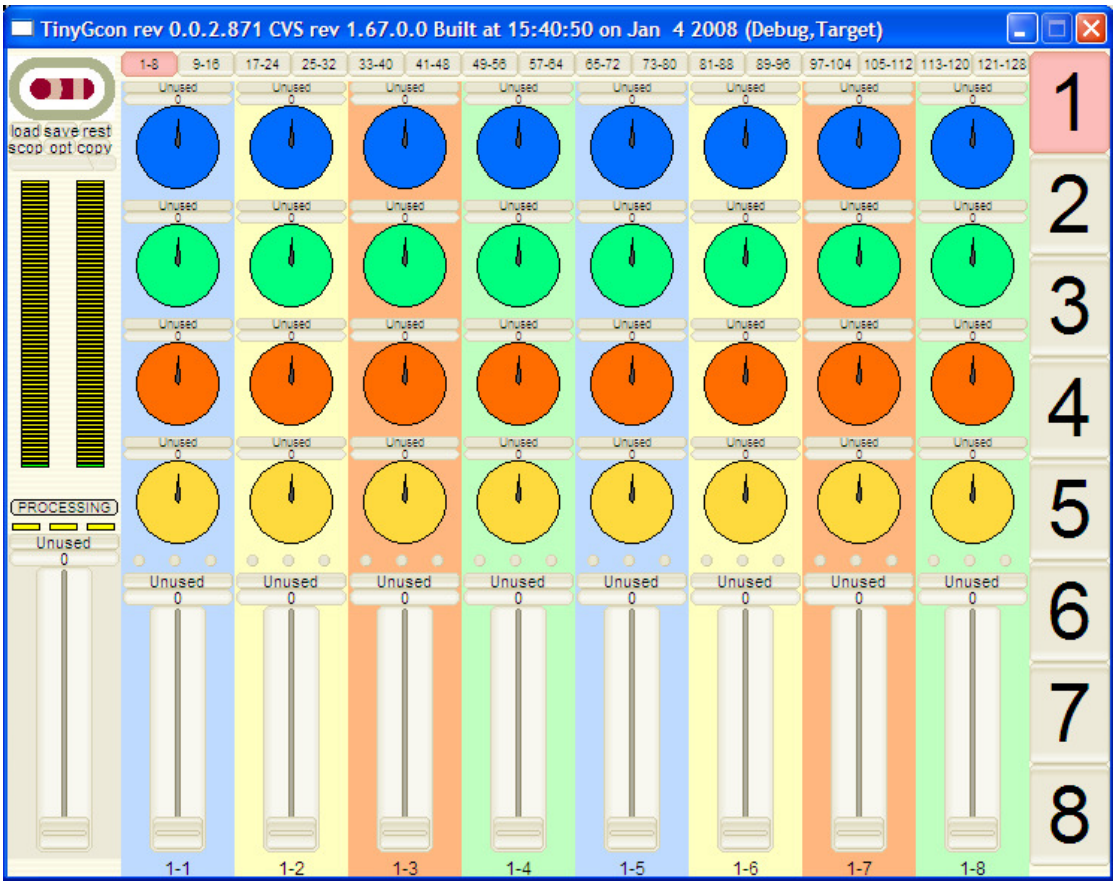


Figure 14 Generic Control Application

7. HARDWARE – SILICON & DEV BOARD

A DSP-module is available, which contains a micro controller with a boot ROM and USB interface (the large device in the middle of the board) and the chip with the TinyDSP Core implemented as an ASIC (this is the small device to the left in Figure 15).

The TinyDSP Core in this instance is packaged in a 0.4mm pitch micro-BGA, which can be difficult to build into prototypes such as flat panel televisions and desktop speakers for initial prototyping. To facilitate easy (hand-soldered) evaluation, the module of has all necessary pins brought out to 0.1” pitch – the whole package then can be used like a 0.6” wide 24 pin DIL chip.

This module can function stand-alone (boots and loads on power-up) and can be programmed by the Control Application such that a complete program and any settings made are first downloaded to the on-board ROM via the USB interface, and subsequently (with PC disconnected) loaded automatically at power on. This enables easy inclusion of the DSP in any prototype system.

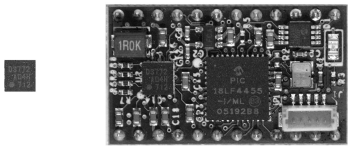


Figure 15 DSP Chip (left) and Development Board (actual size)

In addition to the DSP module board, a mother board was designed (Figure 16). This contains some additional analogue IO circuitry (it can drive

headphones directly) and also some break-out pins where knobs and switches can be connected.

The micro-controller on the DSP module includes control-rate A-D and switch inputs that facilitate the connection of small control panels directly. When prototyping, this makes for easy inclusion of such things as bypass-switch, level control and scene-mode selection.

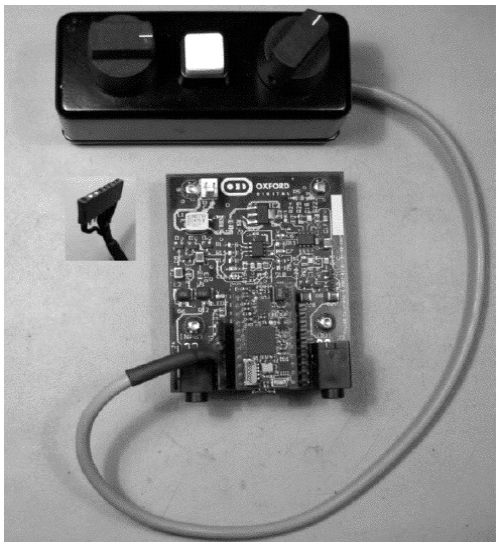


Figure 16 DSP Module mounted on Mother-board and showing Switch/knob unit

7.1. Power Requirement

In battery powered devices, particularly mobile telephones, MP3 players and docking stations, battery life, and therefore power consumption, is a critical issue.

The TinyDSP Core ASIC in Figure 15 has a power consumption of only 80mW when running a full program (including the on-board ADCs and DACs, but not including the micro-controller).

8. CONCLUSION

A complete solution, from bit-accurate Simulink simulation through automatic code generation, to execution in an embedded system has been presented. By designing the target DSP core with automatic code generation in mind, and by keeping the TinyDSP Core architecture simple and focussed, an extremely efficient and reliable end-to-end solution can be provided that has wide ranging application possibilities.

By using industry-standard simulation software as a front-end for algorithm development, and generating object code directly from the Simulink representation, very significant reductions in development time (as much as 10:1) and cost, as well as increased reliability of the generated code can be achieved.

9. ACKNOWLEDGEMENTS

Thanks go to the team at Oxford Digital, especially Peter Eastty, John Richards, Duncan Stott, Mike Smith, Dave Cowan and Chris Gerard. Thanks also to Graham Reith at Mathworks.

10. REFERENCES

[1] Sutherland, William Robert, “The on-line graphical specification of computer procedures”, Massachusetts Institute of Technology, 1966

[2] B. Drummond & Jimmy Cauty (a.k.a.‘The Timelords’) “The Manual”, 1st Edition 1988. ISBN 0-86359-616-9

[3] P. Eastty “Digital Audio Processing on a Grand Scale”, presented at the 81st AES Convention (1986) – AES Paper Number 2381.

[4] C. McCulloch “Automatic Generation of Microcode for Digital Audio Signal Processor”, presented at the 81st AES Convention (1986) – AES Paper Number 2382

[5] W. Kentish & D. Bell, “An Automated Approach to Digital Console Software Design” presented at the 81st AES Convention (1986) – AES Paper Number 2373

[6] Eastty et al “The Software Behind Large Digital Mixer” presented at the 99th AES Convention (1995) – AES Paper Number 4125

[7] P. Frindle, “Implementation of Dynamics Section in a Digital Console Design” presented at the 100th AES Convention (1996) – AES Paper Number 4166

[8] Wikipedia list of Visual Programming Languages: http://en.wikipedia.org/wiki/Visual_programming_language

[9] Bartosinski et. al. “Integrated Environment for Embedded Control Systems Design”, Processing Symposium, 2007

[10] M Donovan, “Deploying Simulink designs on your DSP”, EE Times 8/17/2006, URL: http://www.eetimes.com/design/automotive-design/4016960/Deploying-Simulink-designs-on-your-DSP

[11] A. Burnard, “Verifying and Validating Automatically Generated Code”, Int. Auto-motive Conference (IAC ‘04) Stuttgart, Germany, 2004

[12] D. Haessig, et al, 2005. Case-study of a Xilinx system generator design flow for rapid development of SDR waveforms. Proc. of SDR 05 Technical Conference and Product Expo, Nov. 14-18, California, USA. pp: 1-6.

[13] P. Eastty, “Digital Audio Processing on a Tiny Scale” presented at the AES 123rd Convention (2007 Oct.)

[14] N. Bentall et al, “Tiny DSP: DSP Core, Algorithm Development and 'Device Mastering”, presented at the AES 34th International Conference (2008, August)