

Autocoding Sensor Processing Applications to Run on DSPs and FPGAs

William I. Lundgren, James W. Steed, Kerry B. Barnes
1247 N. Church Street, Suite 5
Moorestown, NJ 08057

Abstract

When running a Gedae application on a homogenous board of PowerPCs, the Gedae runtime kernel (RTK) executes on each processor and handles firing functions and moving data to and from other processors. This paper focuses on the processing of the data by the individual PEs once the data has arrived in the processor. We explore the automated conversion of a signal flow graph (using Gedae-SFG here) into a sequential set of instructions for a DSP. The same techniques can be used when building multiplexed implementations for FPGAs by adding a step to convert the sequential code into an RTL like language. This RTL multiplexes the operations onto available devices This paper includes an example application (a FIR filter) that illustrates the efficient implementation on an example DSP and FPGA processors.

Keywords: FPGAs, DSPs, autocoding, vector, compiler

Introduction

Gedae is an integrated design environment for building the software used in deployed systems and advanced demonstrators based on boards of digital signal processors (DSP) (e.g., AltiVec, PowerPC, TigerSHARC) or distributed networks (e.g., Linux clusters). The programming problem for these systems can be divided into 2 major components – 1) getting the data into and out of local fast memory including interprocessor communications and 2) efficiently processing the data that is in local memory. The movement of data is a complex problem and not addressed in this paper. Gedae Data Flow Graph (Gedae-DFG) deals with this problem.

The Gedae-DFG capability automatically implements the distributed control needed to move the data on a multiprocessor system into local memory for processing. The implementation is accomplished in 2 pieces – pre-runtime graph compilation and a runtime kernel. The graph compilation uses of hundreds of algorithms and makes dozens of passes over the data flow graph. Each pass solves a problem or a set of problems such as dataflow deadlock. Gedae's Runtime Kernel (RTK) runs on each processor on the system and implements decisions that have to be made at runtime. The developer is left to concentrate on the algorithmic details of the application.

Efficient processing of data in local memory is accomplished in 3 ways – 1) the vendor (or end user) hand-codes vector routines to intelligently use the ALUs, 2) the compiler vectorizes for-loops and 3) Gedae autocodes an optimized implementation.

Gedae's primitives are built on top of the Gedae E library and are written in C code open to compiler optimizations. If the vendor provides an optimized vector library, then the optimized routines are used in place of the compiled C code. This approach suffers from lack of completeness and requires management of separate implementations for each chip architecture.

Gedae's Signal Flow Graph (Gedae-SFG¹) language builds optimized code dependent on the architecture of the processor, allowing for vector optimizations without a vector library. When algorithms are constructed using Gedae-SFG, the low-level ALU, register, and memory interaction can be automated by Gedae just as the high-level processor interaction is with Gedae-DFG. Gedae-SFG provides optimizations similar to a vector compiler but without hard coding those optimizations to a

¹ Gedae-SFG has been integrated with Gedae for research purposes. A first release of the capability was made during spring of 2006. Once the concepts discussed here have sufficiently matured they will be released as part of the commercial product.

specific chip architecture. Gedae-SFG has been designed to handle a wide range of processors – from Pentiums to FPGAs – and is ideal for targeting simple, compute-intensive architectures.

The combination of these techniques provides a very high level of portability – an application mapped to one board can be run on another board without modification.

Issues in Vector Programming

Our research is directed at automating the generation of efficient code for processing data in local memory using arbitrary architectures including fixed architectures like DSPs and flexible architectures like FPGAs. When programming hardware at this low level, 5 common problems must be addressed:

- 1) Timing related issues where some components have latencies that have to be accounted for in parallel paths.
- 2) Multiplexing of processing on limited components – for example, we need 256 multipliers but there are only 64 available.
- 3) Integration of application with hardware including ADCs and external IO.
- 4) Integration of 3rd party or user IP like optimized routines and FPGA cores.
- 5) Integration of FPGA processing with general purpose processors in heterogeneous systems – particularly those with multiple general purpose processors and multiple FPGAs.

This paper focuses on the 2nd problem – multiplexing operations on fixed architectures and flexible architectures with limited resources.

Example Processors

To show how Gedae-SFG can be used to target a special purpose DSP, we will study its use on a simplified model of the Sharc ADSP-21160N. We will illustrate the FPGA programming concepts with a fictional chip that has 2 hardware multipliers and 10 hardware adders. It also has “unlimited” registers. The discussion of the Sharc will demonstrate how Gedae’s single sample language can be used to automatically generate highly efficient code for simple processor architectures.

Optimizing to a Fixed Architecture

A Language Support Package (LSP) allows target-optimized code to be exported from single sample components in the Gedae-RTL graph. To allow for these target-based optimizations based on this information, we must provide Gedae with information about the target architecture on the processor level, including the operations and byte widths for each ALU path, the size of the register file, the size of the local storage, and how all the components are connected. By entering this information into Gedae’s embedded configuration, Gedae is able to create a virtual model of the target processor that includes those components, and use that model to tailor the code generation to make maximum use of the components.

The concept requires that we have a simple Hardware Description Language (HDL) for defining the hardware architecture. Using a set of common components like ALU and memory, the HDL describes the architecture as a selection of these components and their interconnection. As an example, a simplified version of the Sharc ADSP-21160N is shown in Figure 1. This depiction is simplified because we are only showing the parts of the hardware necessary to create an optimized vector routine.

We are not trying to build a complete application – just an optimized routine for a

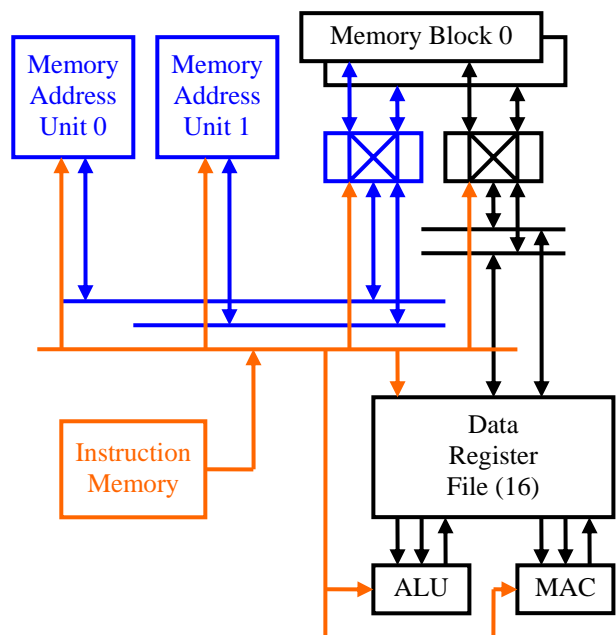


Figure 1 The architecture of the ADSP-21160N

very specific vector operation. The output of this programming will be encapsulated in a C routine and invoked from a C program in the same way we invoke an optimized routine now. For purposes of illustration we have also left the second CPU off the diagram. In what follows we will first define the architecture and each of the components. In order to automate the programming of such a processor we need to know the characteristics of each of the components. Each of these components is defined in the paragraphs below.

The **memory address units** have 2 levels of indexing and N individual addresses. The instruction for a MAU consists of an optional I indicating increment and a number indicating the address to use. The increment takes place after the address is used. The address units will be set up in a separate operation. There are six values that can be set. A0 is the starting address of the primary loop. O1 is the offset for the nested loop. S0 is the stride for the primary loop. S1 is the stride for the secondary loop. N0 is the number of iterations in the primary loop and N1 is the number of iterations in the secondary loop. The behavior of a MAU is succinctly defined in the snippet of C code shown below. There is a set of the values for each address in each address unit.

```
for (A0, i=0; i<N0; A0+=S0, i++) {
  for (A1=A0+O1, j=0; j<N1;
      A1+=S1, j++) {
    /* Wait for I<n> */
  }
}
```

The **memory blocks** are dual ported but only one port is shown here since it is the only one involved with the CPU. Each memory port has an address port and a data port. There are 2 instructions that are available for each memory – R and W. Each instruction can have one or command for each memory block.

The **data register file (DRF)** has 16 registers (Figure 2). The reading and writing to registers is controlled by the operations on either the memory unit connected via the cross bar (XB), ALU or the MAC unit. The ports for the data register file are named as in the following diagram. The connectivity of the port to a register is represented by a number in the instruction – in this case 0-15. A register can be

read from and written to on the same clock cycle.

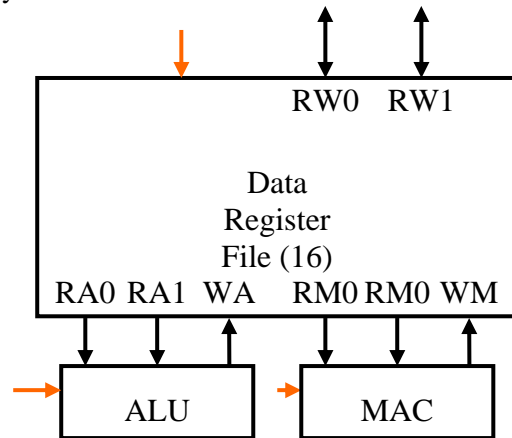


Figure 2 Processor Structure

The **arithmetic logic unit (ALU)** (figure 2) executes any of a number of operations that have been predefined and places the results into the destination register in a single clock cycle. The input and output registers are determined by the DRF port settings.

The **multiply accumulate unit (MAC)** (figure 2) does a multiply and then accumulates the results into one of its internal registers. It can also do the multiply and place the result back into the DRF. We have 3 separate arguments to the MAC. OP indicates whether the multiply takes place. The value in A indicates that the result is accumulated into the corresponding register. OUT identifies when the result is to be written to the output port. An M indicates the multiplier and an A<n> indicates accumulator n. If the M or A<n> is followed by a C then the accumulator is cleared as well.

The **cross bar (XB)** is illustrated in Figure 3. If the instruction is 0 the cross bar is configured as

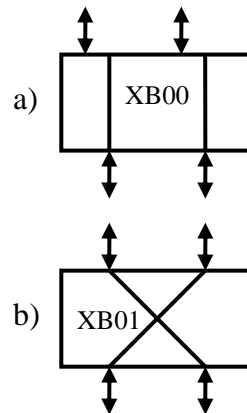


Figure 3 Structure of the crossbar

shown in Figure 3-a) for straight feed through of the data from MB0 to RW0, and MB1 to RW1. If the instruction 1 is issued then the cross bar is configured as shown in Figure 3-b) so MB0 is connected to RW1 and MB1 to RW0.

Example: 3 Point FIR Filter

The FIR filter is described in a fully parallel way free of implementation detail. The block diagram is shown in Figure 4. To implement this algorithm on the Sharc hardware, we will map this graph to our internal model of the Sharc. While it is possible to recognize the multiply accumulate nature of the algorithm and use the MAC for that operation in a single cycle, for the purposes of this illustration, we are going to use the MAC only for the multiply and will use the ALU to do the accumulation. The FIR filter is going to operate on a buffer of 5 values and there will be 3 coefficients. The call will be fir3 (float *in, float *C, float *out). It will be the responsibility of the calling program to properly buffer the data to provide continuous outputs. In this case it will be necessary to overlap by 2 samples (size (C) - 1).

The diagram in Figure 5 shows all the operations that must be completed for a 3 point FIR filter on a buffer of 5 samples. A variety of implementations are possible. In this case, we are reading the coefficients from the C buffer into registers during startup since they will not change. Also at startup, we initialize counters in the two MAUs to iterate through the input and output buffers (in and out). In Figure 5, the orange/yellow bars show two steady-state iterations. The blue bars at the beginning and end show the pipeline filling and emptying.

Once each operation in the block diagram (Figure 4) is mapped to a component (Figure 5), defining the pipeline behavior, we can deduce a set of long instruction words that describe the vector routine. The long instruction words for our 3 point FIR filter are shown in Figure 6. Each instruction is simply a list of control signals that define the state of all the hardware components. These long instruction words are "virtual machine code," i.e., machine code that programs the virtual machine. The internal program of long instruction words can then be directly translated by the LSP to assembly code which implements the vector routine.

Targeting Flexible Architectures - FPGAs

The process is similar when targeting FPGAs but there is an additional step. As an example, we will consider an FPGA with at most 2 multipliers and 10 adders. Such a processor is

shown in Figure 7. The Data Register File encapsulates multiplexers and connections needed to direct data from the registers to the ALUs and back again. We map a 3-point FIR filter to this programmable architecture (Figure 4).

Clearly we need to multiplex the multiplications to fit the 3 point FIR filter on our FPGA processor. The design uses three multipliers, but our FPGA only has two multipliers. To create the multiplexed FPGA design, first we create the LIW for this design on our FPGA processor. Figure 8 on page 6 is an example LIW program that executes the 3 point FIR filter on the example FPGA. We assume the coefficients (C0, C1, C2) are preloaded into registers. Then there is a simple 2-step steady-state operation to compute each output sample.

This LIW program can be converted easily to an output Gedae-SFG graph and processed using our available tools to produce synthesizable VHDL. Registers in the register file can be specified as registers in the single sample language. The adders and multipliers can be specified as the add and multiply primitives. The multiplexing needed to map the registers to the ALUs can be specified by multiplex primitives, in this case simple two-input MUX primitives specified using the IF-statement.

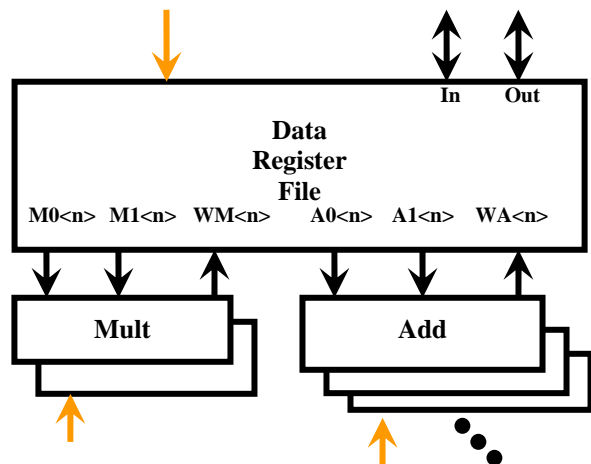


Figure 7 The architecture of the ADSP-21160N

The Gedae-SFG graph created from this LIW program is shown in Figure 9 on page 6. This design has equivalent functionality to the original Gedae-SFG design but requires additional clock cycles. The clock is specified as "Clock(2)", meaning two clock pulses will be required to process each sample.

Multiplexers surround the “i32_mult” primitive, controlling which inputs are connected during each of the 2 pulses. A multiplexer also controls the 2nd input to the “i32_add” primitive – on the 1st pulse we add the results from the two multipliers, and on the 2nd pulse we add that sum to the value from the reused multiplier.

Conclusions

Gedae research under the DTC program has led to the formation of a solid concept for implementation of multiplexed processing on both fixed and flexible architectures. While there is much work to be completed, the

foundation for the future work is in place and we forge ahead optimistically.

Reference

1. Analog Devices, System Development and Programming for the ADSP-21161 SHARC Processor Workshop Slides

Acknowledgements

The work reported in this paper was funded by the Electro-Magnetic Remote Sensing (EMRS) Defence Technology Centre, established by the UK Ministry of Defence and run by a consortium SELEX Sensors and Airborne Systems, Thales Defence, Roke Manor Research and Filtronic.

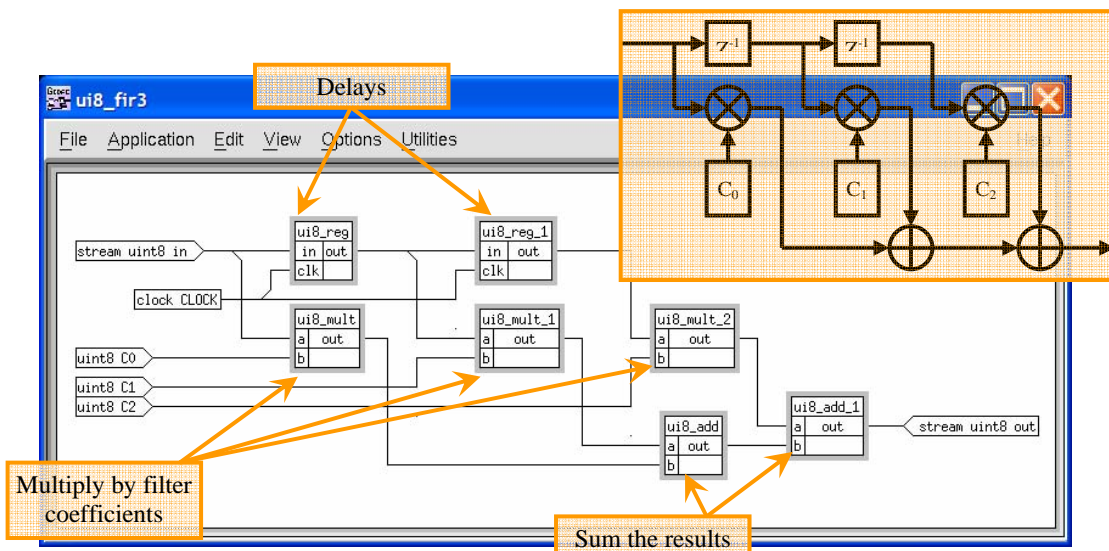


Figure 4 Gedae-SFG of a 3-point FIR filter much like a signal flow graph from a DSP book.

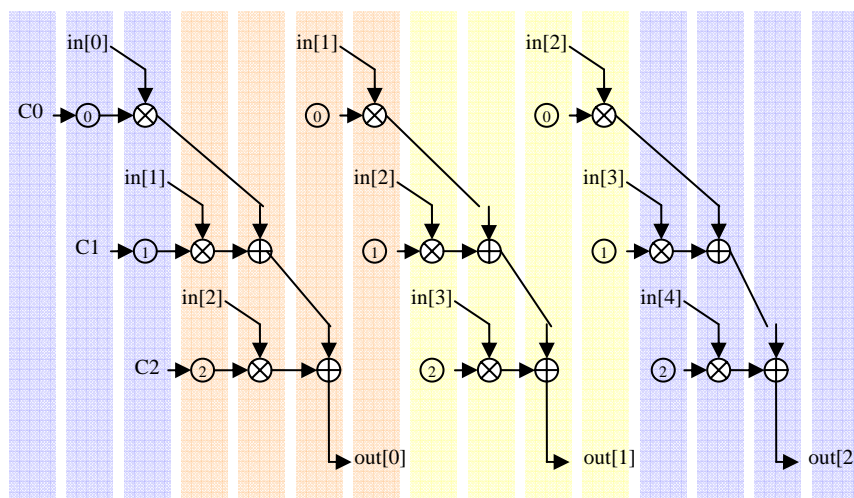


Figure 5 Implementation of a 3-Tap FIR filter with pipe filling, steady state and pipe emptying.

Reading Data Memory with Coefficients Loaded and Stored in Registers																
		0	1	2		4	5	6	7		8	9	10	11		
	MAU0		I0	I0		I0		I0	I0		I0					
	MAU1	I0	I0	I0					I1					I1		
	XBA	0	0	0		0		0	0		0			0		
	XBD	0	0	0		0		0	0		0			0		
	MB0		R	R		R		R	R		R					
	MB1	R	R	R					W					W		
DRF	RW0		3	3	REPEAT M-N TIMES	3		3	3	END REPEAT	3					
	RW1	0	1	2					4							4
	RA0							4	4					4	4	
	RA1							5	5					5	5	
	WA							4	4					4	4	
	RM0			0			1	2			0		1	2		
	RM1			3			3	3			3		3	3		
WM			4			5	5		4			5	5			
MAC	ALU															
	OP			*			*		+		*		*		+	*
	AC															
	OUT			M			M	M			M		M	M		

Figure 6 The LIW program that implements the FIR filter on our example fixed architecture

Long Instruction Word Program for Executing 3-Point Fir Filter on Resource Limited FPGA							
		Setup	0	1		0	1
	Mult0		*	*	REPEAT CONTINUOUSLY	*	*
	Mult1		*			*	
	Add0			+			+
	Add1						
DRF	In	Load (C0)	M0-0			M0-0	
	R1	Load (C1)	M1-0			M1-0	
	R2	Load (C2)		M0-0			
	R3	Cnct (R4)	M0-1	In			M0-0
	R4	Cnct (R5)	M1-1			In	
	R5			M0-1			
	R6		M0-W	A0-0, M0-W			A0-0, M0-W
	R7		M1-W	A0-1		A0-1	
	R8						
	R9			A0-W		A0-W	
	R10					Out	
						END REPEAT	

Figure 8 The LIW program that implements the FIR filter on our example FPGA architecture

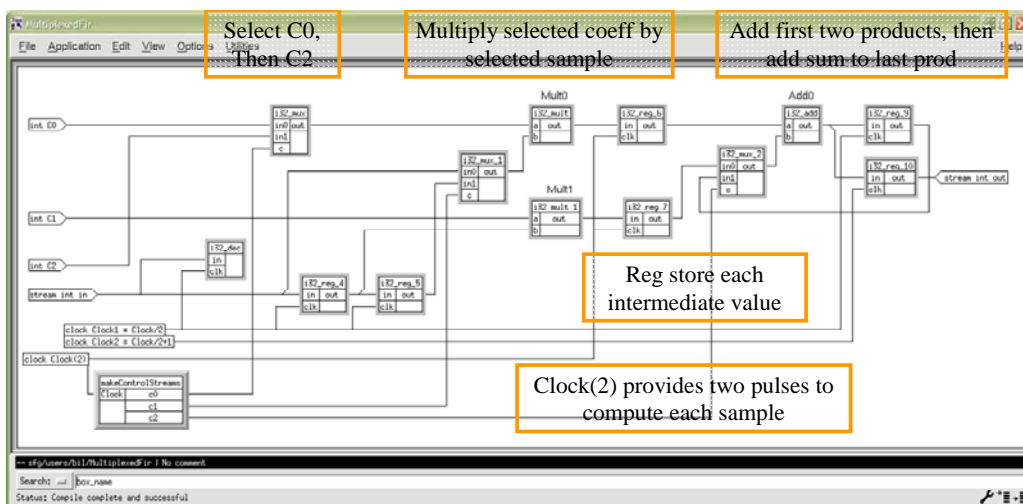


Figure 9 Geda-SFG with sufficient detail for compilation using available Geda and vendor tools