Multiprocessor DSP Code Synthesis in Ptolemy

Praveen Murthy[1]
Master's Report
UCB/ERL Tech. Memorandum No. M66/93
Electronics Research Lab
College Of Engineering, UC Berkeley
Berkeley, CA 94720
August 1993

1 Abstract

Ptolemy is a flexible tool that has been developed for simulation, prototyping, and software synthesis for large heterogenous systems. The objectives of Ptolemy encompass virtually all aspects of designing signal processing and communication systems, ranging from algorithms and communication strategies, through simulation, parallel computing, and generation of real-time prototypes. This project report describes the multiprocessor code synthesis aspect of Ptolemy, and in particular, code synthesis for the Sproc digital signal processor (DSP) made by Star Semiconductor.

[1] This research was supported by Star Semiconductor and the State of California MICRO Program.

2 Introduction

The reality with most high-performance DSP chips is that they are very tedious and complex to program. In addition, programming parallel DSP architectures has required adaptation of algorithms to parallel implementations on a case-by-case basis, with each application being carefully analyzed for concurrency. The bulk of coding for DSPs is done using assembly language techniques and requires the programmer to be familiar with many details of the chips architecture, such as pipelines. Although C compilers exist for many popular DSPs, they produce code that is 4-5 times less efficient than hand-assembled code and thus are not a feasible solution for demanding DSP tasks. The problem with high level language compilers is that they are often not able to use specialized addressing modes (like bit-reversed addressing for FFT's, hardware support for circular buffers) efficiently. Another option is to use an applicative language like Silage to specify DSP programs [Gen90]. The declarative semantics of the language, and its support for fixed-point arithmetic, makes highly efficient code-generation possible. The Mentor/EDC DSPstation is based on Silage [Men92].

Neither assembly language nor C coding are the most natural ways of specifying DSP algorithms. A more natural method is to use block diagrams and signal flow graphs [Mes84]. In this approach, any complete system design methodology must include software synthesis from the high level block diagram of the system. This can be done by having hand written assembly code segments in the individual blocks. The hope is that such blocks will be stored in standard libraries rendering them modular, reusable software components. Software synthesis then consists of two phases: scheduling and code generation. During the scheduling phase, the blocks are partitioned onto parallel processors, and for each processor, a sequence of block invocations is determined. Code generation consists of piecing together the assembly code segments from the blocks in the order determined by the scheduler. This methodology seems to be a popular one and is being used commercially in the Star Semiconductor Sproc system [SS92], in the Comdisco DPC system [Pow92], and in the CADIS Descartes system.

2.1 Overview of Ptolemy.

Ptolemy is a software tool that has been developed at Berkeley to facilitate the simulation of large heterogenous systems, for example, communication networks and signal processing systems [Buc93][Alm92], and is based on dataflow semantics [Den80]. Heterogenous computational models are achieved through the *domain* abstraction which enables each subsystem to be modeled in a way that is natural and efficient for that subsystem. Combination of these heterogenous subsystems is enabled by the definition of a standard interface called the *event horizon*. Ptolemy is implemented in C++ and uses object oriented programming (OOP) methodology to achieve extensibility to new domains without the need to modify or understand existing ones.

The basic unit of modularity in Ptolemy is the Block. A Block contains a method called `go()` whose action can be different depending on the model of computation used. In the code generation domains, this method synthesizes code for the target processor. Its invocation is directed by a Scheduler (another modular object) which, in the case of the synchronous dataflow (SDF) model of computation, determines the order of firing of the Blocks at compile time. Another type of object called the Target, describes the specific features of a target for code generation. An interconnected block diagram constitutes the user-interface view of the system (or *universe*). For an example, see figure 20 at the end of the report.

There are many domains in Ptolemy, including synchronous dataflow (SDF), discrete event (DE), and dynamic dataflow (DDF). Of chief interest for this project was the SDF domain since that is the model of computation for which the code generation capability has been completely developed. By model of computation, we mean the operational semantics and the scheduling strategy that are used in simulating computations in that domain. For example, the DE domain has a different model of computation than SDF because in the DE domain, the notion of time is very important, whereas in SDF, time is not an issue. For DE stars, the management of timestamps (which indicate the precise time at which the corresponding data was generated) is as important as the functionality of the star. The order in which events are generated can be nondeterministic (for example, queueing networks); hence, we cannot do static scheduling in DE and have to do scheduling dynamically. Therefore, the operational semantics and scheduling strategy used for DE is different from that of SDF; we refer to these two as having different models of computation.

While the domain abstraction is primarily for mixing different models of computation, in the code generation domains, different domains can have the same model of computation but distinct target languages. For example, Blocks that generate Sproc assembly code using the SDF model of computation form their own domain, separate from Blocks that generate Motorola 56000 code using SDF. As an example of how Blocks, Targets, and Schedulers can interact, consider a set of Blocks in the Sproc domain. Any of several targets can be chosen including one that generates code in a higher level form consisting of statements containing Block names and parameters instead of in-line assembly code. Any of two schedulers, each of which uses different heuristics to obtain parallel schedules, can also be chosen.

2.2 Synchronous Dataflow

Synchronous Dataflow is a special case of dataflow in which algorithms are described as directed graphs where the number of samples produced and consumed by each actor on each invocation is fixed and known at compile time [Lee86][Lee87]. This model is appropriate for signal processing because a large set of DSP algorithms are easily specified under the SDF paradigm. Advantages of SDF over dataflow are a greater degree of setup-time syntax checking (since sample-rate inconsistencies which can cause deadlock can easily be detected), and run-time efficiency (since the schedules generated are fully static rather than dynamic). In addition, this model exposes the inherent concurrency in an algorithm that can be exploited in parallel hardware [Sih91].

3 The Sproc DSP

The Sproc signal processing chip is a multiprocessor DSP made by Star Semiconductor [SS92]. The architecture uses fixed point arithmetic and has 16 bit address paths and 24 bit data paths. There are four so called general signal processors (GSPs) on the chip and these share a central multi-ported program and data memory unit (fig. 1). Contention for memory is eliminated by

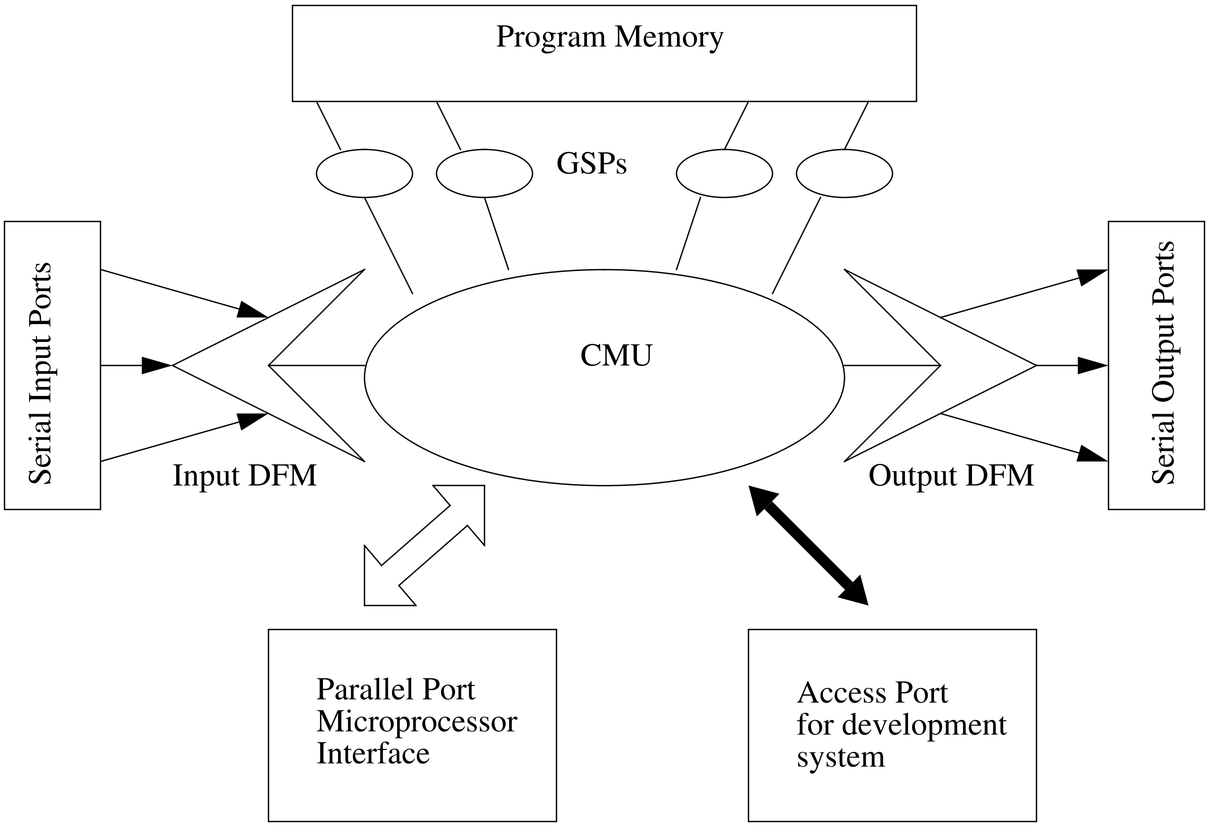

Fig 1. Sproc central memory architecture

giving access to each GSP in a time division multiplexed manner. Input/output data flow managers (DFMs) coordinate simultaneous data streams, serial channels interface signals, and parallel interfaces enable connections to external processors. The use of time division multiplexed memory and the DFMs eliminates the need for interrupts to handle multiple data streams.

3.1 The Central Memory Unit

The CMU uses a frame composed of time slots, or memory access periods, allotted for each GSP and for the I/O. The basic frame represents one Sproc chip machine cycle (five master clock cycles) and includes five time slots of one master clock cycle each (fig. 2). Time slots 1 through 4 are used by the GSPs. During time slot 1, GSP 1 can read or write the CMU; during slot 2, GSP 2 can read or write and so on. Time slot 5 is used by I/O operations. It is submultiplexed into 8 divisions for parallel I/O and other operations. During every other machine cycle, slot 5 is used by the parallel port and can support signal or data flow without interrupting the processing of any of the GSPs. During the other machine cycle, slot 5 supports the serial ports, the access port, and the on- chip probing port.

1 machine cycle (5 clock cycles)

Slot 1 GSP 1	Slot 2 GSP 2	Slot 3 GSP 3	Slot 4 GSP 4	Slot 5 I/O

Fig 2. CMU Time Slot Divisions

3.2 The General Signal Processors

The GSPs are 24-bit fixed-point processors and have Harvard-like architectures having separate program and data buses. The word widths for both program and data are 24 bits, although some registers are narrower. There are 5 subcycles within each GSP machine cycle (corresponding to each clock cycle). These are instruction fetch, decode, data fetch, and two cycles for the execute phase (fig. 3). All instructions except two execute in one machine cycle, the two exceptions being the multiply and multiply-and-accumulate instructions, where the result is available on the third machine cycle. However, non-multiply instructions can be executed while the multiplier completes. This is useful in constructs like loops where there are necessarily two other instructions (one to check loop termination and one to load the multiplier's register), resulting in zero waiting time for the multiply. Two data formats are accommodated in the machine: a fixed point format extending from -2.000 to +1.9999999762 and an integer format from -8388608 to 8388607.

The bits in a 24-bit Wait register in each GSP can be set or cleared using an LDWS instruction. A 24-bit Trigger bus connects this register and serial output ports to the first 24 locations in the data memory so that a write to any of these addresses will clear the corresponding bits (if set) in the Wait registers. This feature is of potential use for interprocessor communication; more will be said about this later.

3.3 The Data Flow Managers

The DFMs coordinate the filing of input and output data into or out of the CMU without interrupting any of the GSPs. Because the CMU is a multi-ported memory space, this activity can take place concurrently with signal processing. The DFMs set up either double buffers or programmable vector FIFO queues in the CMU. The FIFO can be up to 256 words long, meaning that 256 samples will be written sequentially into memory before overwriting occurs. This implies that one does not have to worry about losing samples if a particularly time consuming task is being executed since an appropriate buffer size can be determined at compile time and set (as long as the size is less than 256). Of course, if the task takes longer than was estimated, we would still lose samples since the buffer would not be big enough. A worst case estimate of task times should therefore be used in order to calculate the buffer size.

3.4 The board

The chip and its associated circuitry reside on a board and are connected to a host IBM PC via the SprocBox which contains a Motorola 68000 microprocessor. The box communicates with

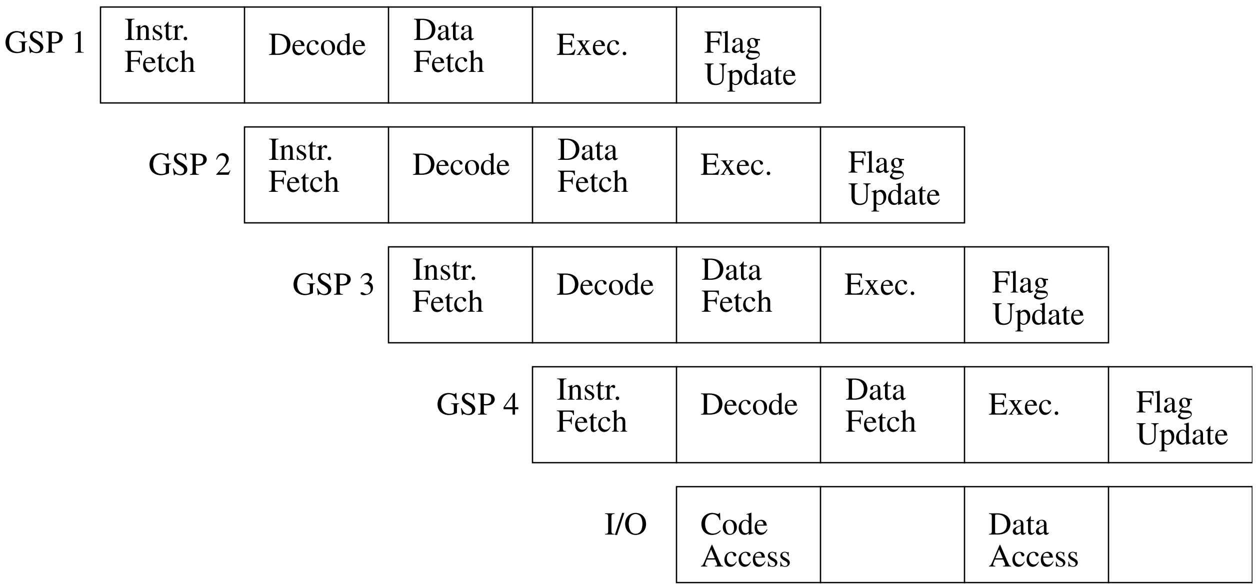

Fig 3. Timing from the GSP's perspective.

the PC over a RS232 port. All user requests to the Sproc chip (such as loading the chip, changing parameters while a program is being run) use the SPROCDrive Interface (SDI) software provided by Star and are serviced by the SprocBox. The board also contains two A/D converters and two D/A converters hooked up to the chips serial ports. Sample rates of upto 20kHz are possible with the 20MHz board. In addition there is a probe port to allow monitoring of signals at any memory location.

3.5 The Sproc development system

The Sproc development system is similar to Ptolemy in that programming is via block diagrams. As in Ptolemy, the blocks contain assembly code to perform the required functions. However, the Sproc system does not use SDF semantics to the full extent possible (as in Ptolemy) and a result of this is that multirate graphs are not cleanly developed on the system. This became evident in the early stage of the project when a couple of applications were developed in order to gain familiarity with the chip. The applications developed were the relatively simple Karplus-Strong plucked string algorithm and a more complicated LPC vocoding system. The LPC coder is described after a discussion on the scheduling strategy used by Star.

3.5.1 SSIMD scheduling

The Star development system uses SSIMD (skewed single instruction multiple data) scheduling [Bar82][Bar83]. In this scheme, the entire application is run on all processors (or as many needed to meet real-time throughput requirements), and skewed in time with respect to one another so that different samples are processed by different processors. This is an attractive scheme for many signal processing applications and works well in practice. Figure 4 shows an example of SSIMD scheduling. Note that processor 2 cannot execute block A until after processor 1 has executed block B because of the precedence constraint. For SSIMD, the scheduling problem is just to determine a uniprocessor schedule and to determine the optimum starting time for each processor (in other words, the skew.)

However, it is easy to see that this method will be suboptimal if large feedback loops are present [Lee86]. As shown in figure 5, a spatially partitioned schedule will do better.

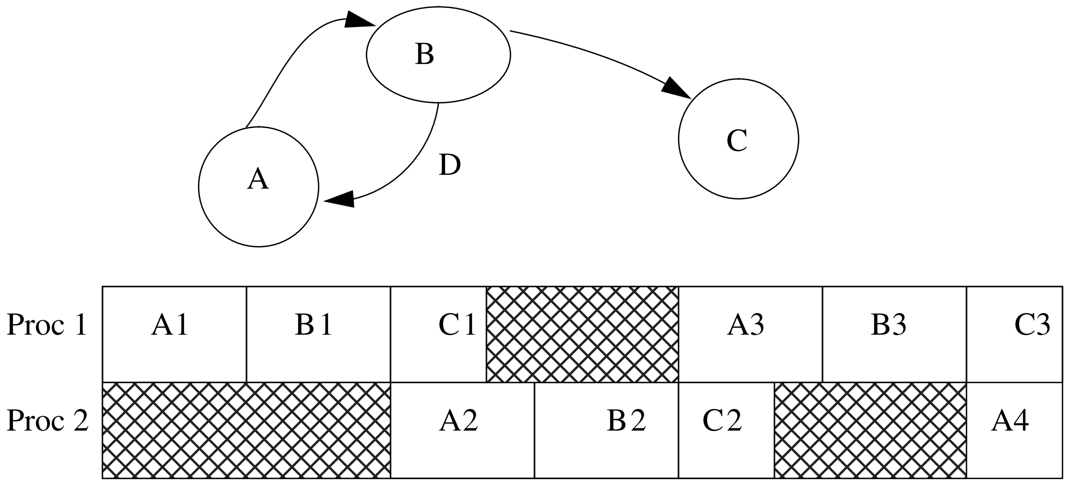

Fig 4. An example of SSIMD scheduling

3.5.2 The LPC vocoder

This required the development of a few cells that were not available in the Sproc library. These were an autocorrelation block, a block implementing the Levinson-Durbin algorithm for calculating the linear prediction filter coefficients, a time-varying FIR filter, and a block to do adaptive quantization. The implementation of the autocorrelation and Levinson-Durbin blocks were the most challenging because the system does not allow the connecting wires between blocks to be arrays. This is essential for implementing multirate graphs because the programmer is saved the trouble of having to ensure that blocks do not fire before data is available. As a result, the cells had to be implemented with synchronization issues in mind. A further optimization was also necessary; the blocks had to be written so that their computation was distributed over several samples rather than just occurring over one sample period, because the SSIMD scheduling scheme used by Star is inefficient for large granularity stars (meaning stars with relatively long execution times). This will be explained later on.

There were two main concerns when implementing these blocks on the Sproc. The first was scaling. Since fixed point numbers are constrained in magnitude to be within 2.0 on the Sproc, the algorithms had to have appropriate scaling at various points. The other issue is the real-time constraint. Cells have to be written so that the maximum execution time does not exceed the maximum allowed for a particular sampling rate (400 GSP cycles at a sampling rate of 10kHz and chip rate of 20MHz.)

To meet the real-time constraint, a pipelining strategy was used. Since the algorithm requires $M + 1$ lags of autocorrelation, these have to be computed. With a frame size even as small as 32, not all the lags can be generated in one sample interval. So, the lags are computed one per sample interval. Once there are 32 samples in the buffer, these are copied to another buffer (within the autocorrelation cell) because, while the lags are being computed one per sample interval, new data continues to arrive, and this data has to be stored in the original buffer. In each sample interval, one lag is computed and output. Similarly, the Levinson-Durbin cell does one pass of the recursion in each sample period. One pass means computing the $k + 1^{th}$ order coefficients

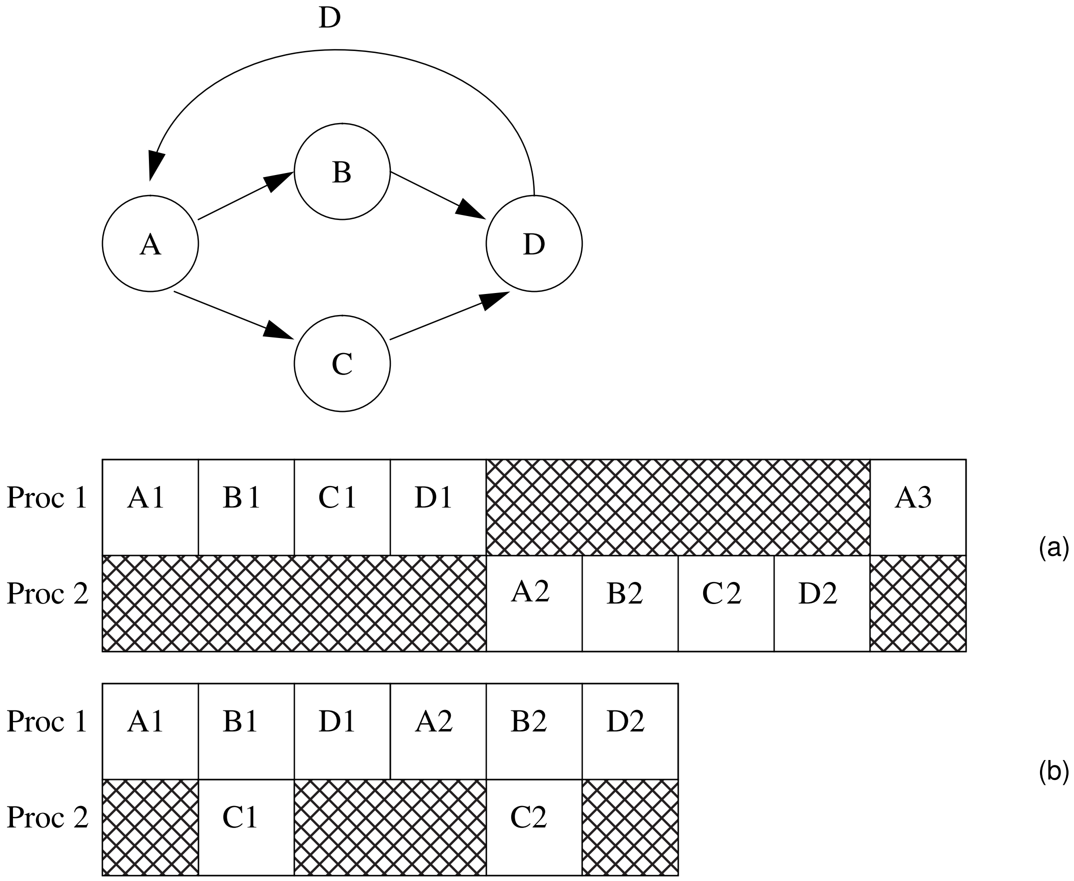

Fig 5. **a)** A suboptimal SSIMD schedule. **b)** A spatially partitioned schedule that's optimal

using equation 1, given below. Therefore, there is a delay of about M samples before the M^{th} order prediction-error filter coefficients have been computed. These coefficients are then copied into the output array so that the error-filter and the reconstruction filter can use the coefficients. Again, because of the pipelined manner of the computation, the cell will start overwriting the coefficient values $32 - M$ samples later instead of 32 samples later. Since the validity of the filter coefficients is assumed to be the frame size, we have to copy it into another array where they will stay that long. Figure 6 gives an illustration of this procedure. The equations that need to be computed are given below. The algorithm is an old one and can be found in an advanced DSP textbook, e.g. [Hay86].

$$a_{k+1}(m) = a_k(m) + \gamma_{k+1} a_k(k-m+1) \qquad m = 0, \dots, k+1 \qquad k = 0, \dots, M-1 \quad \text{(EQ 1)}$$

$$\gamma_{k+1} = \frac{-1}{P_k} \sum_{m=0}^{k} a_k(m)\, r_x(k+1-m) \qquad k = 0, \dots, M-1 \quad \text{(EQ 2)}$$

$$P_{k+1} = P_k\left(1 - \gamma_{k+1}^2\right) \qquad P_0 = r_x(0) \quad \text{(EQ 3)}$$

$$r_x(k) = \frac{1}{N} \sum_{n=|k|+1}^{N} x(n)\, x(n-k) \qquad -N+1 \le k \le N-1 \quad \text{(EQ 4)}$$

The k^{th} reflection coefficient is γ_k and is given by eqn. 2. The $a_k(i)$ are the k^{th} order filter coefficients, P_k is the k^{th} order error power, and $r_x(k)$ is the k^{th} autocorrelation lag. M is the order of the desired filter. Note that equation 4 gives a biased estimate of the autocorrelation; this is done to ensure that the autocorrelation is positive semi-definite. Also note that $a_0(0) = 1.0$ in equation 1.

The amount of effort required by the programmer to make cells like these work clearly shows that a better way is needed to express cells that generate more than one sample. The SDF model allows such “multirate” cells to be expressed cleanly in that it hides, from the programmer, all of the synchronization steps necessary to ensure that there is enough data before a cell can fire. The scheduler ensures that cells only fire after they have enough data. For example, above, we had to write the autocorrelation and Levinson-Durbin cells so that they counted how many samples they had received before they started computing. By contrast, the Ptolemy versions of these cells do not have any of the messy “control” code; they only contain the computation code.

It is also worth mentioning why the cells have to do the computations in a pipelined manner. Consider the case of scheduling an acyclic graph. If any node in such a graph has an execution time of P sample intervals, then we need at least P processors to meet the throughput (which is assumed to be the sample rate). Hence, if the execution time of the autocorrelation cell every 32 sample intervals were 8 sample intervals, then we would not be able to meet the throughput during those 8 sample intervals because we only have 4 processors. Adding the Levinson-Durbin cell to this will make the worst-case execution time even longer. Therefore, we have to restrict the duration of each cell to be at most 4 sample periods. This is why these cells have to be written as if they were being interrupted so that other cells in the graph (e.g., the FIR, A/D, and D/A) can fire. By contrast, with spatial partitioning we won't have this restriction because a cell that fires infrequently, but has a long execution time, could be partitioned onto a different processor from one on which frequently firing cells get partitioned. Figure 7 illustrates a case where we have 2 processors and where the overall execution time of the graph is longer than 2 sample intervals. B and C can represent the autocorrelation and Levinson-Durbin cells for example. Hence, the worst case execution (in the Star context) occurs every fourth sample period. During the other sample periods, B and C execute some synchronization code; the duration is much shorter. As can be seen, D (which can represent a D/A) won't be executed during sample periods 3,4,5, and 6. In

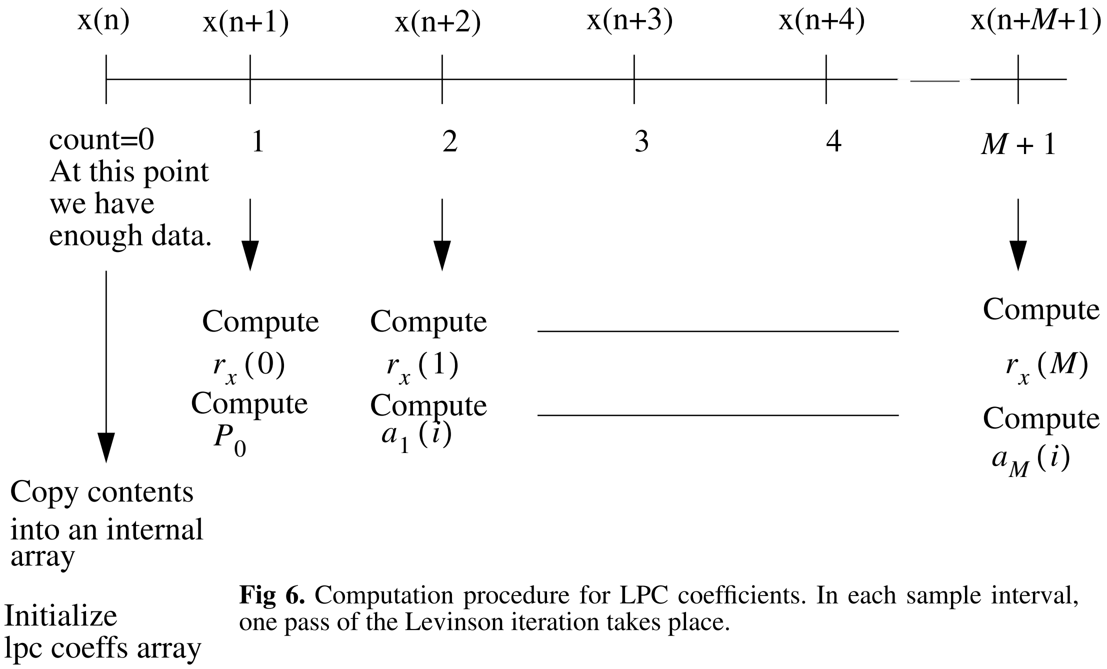

Fig 6. Computation procedure for LPC coefficients. In each sample interval, one pass of the Levinson iteration takes place.

contrast, the spatially partitioned schedule in figure 7a shows that after the 7th sample period, D will be executed every sample period. It is easy to see that if all of the nodes combined take less than two sample periods to execute, than the two-processor system can meet the throughput using SSIMD scheduling.

Scaling is done in several places. In the autocorrelation cell, the individual products are scaled by $\frac{1}{N}$ and then added. Even though this increases the number of multiplies by $N-k$ when the k^{th} lag is computed, it is necessary to avoid overflow. This makes using larger frame sizes trickier since not only are there more computations, but if the numbers get too small (because of the scaling), then inaccuracies will creep in. The scaling is needed because even if $r_x(k)$ is less than 2.0, the sum could still be larger than 2.0 but less than N. Scaling the individual terms by $\frac{1}{N}$ is one way to ensure that if $r_x(k)$ is indeed less than 2.0 (and hence representable in fixed point), then we do the computation correctly. If $r_x(k)$ is greater than 2.0, no attempt is made to represent this correctly since the input would have to be scaled; this is left to the user.

In the Levinson-Durbin recursion, scaling is required in two places. The first is the division by the error-power in the computation of reflection coefficients. Clearly, if P is smaller than 0.5, then the inverse cannot be represented in fixed point notation. Hence, P has to be scaled to lie between 0.5 and 2.0. Since γ is guaranteed to lie between -1 and 1 (because the autocorrelation is positive semi-definite), the numerator (the summation) can be scaled by the same amount too,

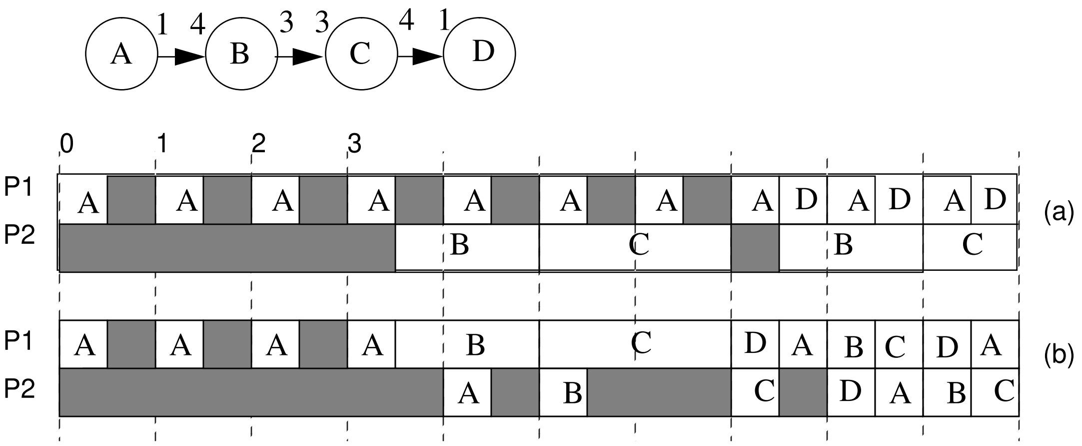

Fig 7. a) A spatially partitioned schedule that meets the throughput **b)** An SSIMD schedule that does not meet the throughput.

without fearing overflow. The second scaling has to be done for the filter coefficients themselves. Equation 1 shows that the initial value, $a_0(0)$, determines the value of the coefficients for all orders greater than 1. To see how starting the recursion with $a_0(0) = b$ affects the computation, where b is some number less than 1.0, denote by γ' the values of the reflection coefficients, and by a_k' the filter coefficients generated by the algorithm when $a_0'(0) = b$. We get that $\gamma_1' = -br_x(1)/r_x(0)$. So $\gamma_1 = \gamma_1'/b$. If we use γ_1 in equation 1, we get $a_1'(0) = b$ and $a_1'(1) = \gamma_1 a_0'(0) = \gamma_1 b$. Therefore, $a_1'(i) = ba_1(i), i = 0, 1$. Now we can see that $\gamma_2' = b\gamma_1$ when $a_1'(i), i = 0, 1$ is used in equation 2. So we can always compute γ_k from γ_k' by $\gamma_k = \gamma_k'/b$ and use γ_k in equation 1 to compute $a_k'(m)$ where $a_k'(m) = ba_k(m)$. What this shows is that starting the recursion with $a_0'(0) = b$ results in the filter coefficients being scaled by b. Therefore, we would like to know what b should be to ensure that $a_k'(m)$ is always less than 1.0 in magnitude. It turns out that choosing $b = 0.25$ works well in practice, but an exact bound can be derived, as is done in section 3.5.2.1.

The cells were tested by constructing a simple speech coder. The Levinson-Durbin cell is used to generate the prediction error filter coefficients, which is used by an FIR filter to generate the error sequence. The error sequence, which should hopefully have much less energy than the speech signal if the prediction is good, is coded using a 3-bit adaptive quantizer. The speech signal is reconstructed by inverse filtering which can be achieved by having an FIR filter in a feedback loop, using the same coefficients generated by the Levinson-Durbin recursion (fig. 8).

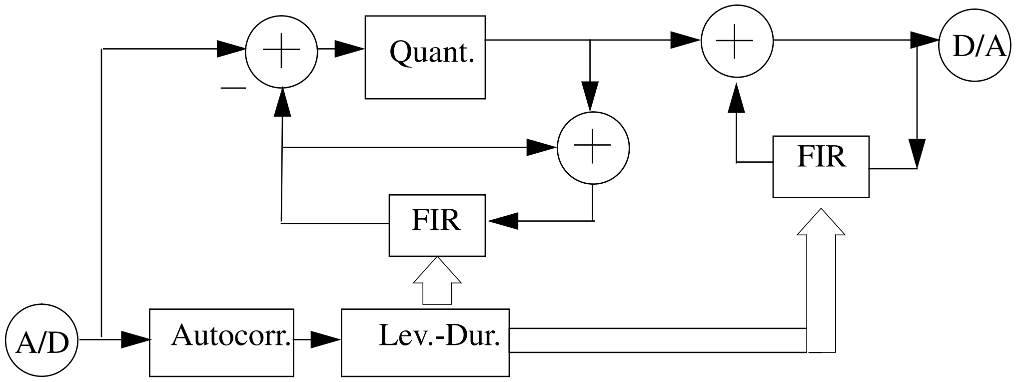

Fig 8. A simple LPC coder

3.5.2.1 An upper bound for $a_0(0)$

By upper bound we mean the value b such that

$$|a_0(0)| \le b \Rightarrow |a_i(m)| \le 1.0 \qquad \forall i, m = 0, 1, \ldots, M \tag{EQ 5}$$

Theorem 1: Equation 5 is satisfied if $b = 1 \Big/ \left(\binom{M}{M/2} \right)$.

We define $\binom{N}{k} \equiv \frac{N!}{(N-k)!k!}$.

Proof:

First we need a few facts and lemmas.

Fact 1: $\binom{N-1}{m} + \binom{N-1}{N-m} = \binom{N}{m}$

Fact 2: $\binom{N}{m} + \binom{N}{m+1} = \binom{N+1}{m+1}$

Lemma 1:

$$|a_i(m)| \le \sum_{j=0}^{k-1} (|a_{i-k}(m-j)| + |a_{i-k}(i-m-j)|)\binom{k-1}{j} \qquad \forall k = 1, \ldots, i \tag{EQ 6}$$

Proof of lemma 1:

The proof is by induction on k. We have from equation 1 that

$$|a_i(m)| \le |a_{i-1}(m)| + |a_{i-1}(i-m)|$$

Letting $k = 1$ in equation 6, we see that the base case holds. Now assume it holds for $k-1$. From equation 1, we have that

$$|a_{i-k}(m-j)| \le |a_{i-k-1}(m-j)| + |a_{i-k-1}(i-k-m+j)|, \tag{EQ 7}$$

and that

$$|a_{i-k}(i-m-j)| \le |a_{i-k-1}(i-m-j)| + |a_{i-k-1}(m+j-k)| \tag{EQ 8}$$

Now, $0 \le j \le k-1 \Rightarrow k \ge k-j \ge 1$. If we substitute equations 7 and 8 in equation 6, we will get four terms in the summation. Consider the term $\left|a_{i-k-1}(i-m-(k-j))\right|$. The summation

$$\sum_{j=0}^{k-1} \left|a_{i-k-1}(i-m-(k-j))\right| \binom{k-1}{j} =$$

$$\left|a_{i-k-1}(i-m-k)\right| \binom{k-1}{0} + \ldots + \left|a_{i-k-1}(i-m-1)\right| \binom{k-1}{k-1} =$$

$$\sum_{j=0}^{k-1} \left|a_{i-k-1}(i-m-(j+1))\right| \binom{k-1}{j}$$

. Similarly, we can replace $m-(k-j)$ in equation 8 by $m-(j+1)$. If we substitute equations 7 and 8 into equation 6 with these index changes, we get two summations, each of the form

$$\sum_{j=0}^{k-1} (a(m-j) + a(m-j-1)) \binom{k-1}{j} \qquad \text{(EQ 9)}$$

where the subscript $i-k-1$ has been omitted for clarity. Equation 9 can be expanded as follows:

$$(a(m) + a(m-1)) \binom{k-1}{0} + (a(m-1) + a(m-2)) \binom{k-1}{1} + \ldots$$

$$\ldots + (a(m-k+1) + a(m-k)) \binom{k-1}{k-1} =$$

$$a(m) \binom{k-1}{0} + a(m-1) \left(\binom{k-1}{0} + \binom{k-1}{1} \right) + \ldots + a(m-k) \binom{k-1}{k-1} =$$

$$\sum_{j=0}^{k} a(m-j) \binom{k}{j}$$

where we have used fact 2.

Applying this to equation 6, we see that it is satisfied for k if it's satisfied for $k-1$. **QED**

Lemma 2: $\left|a_i(m)\right| \le \left|a_0(0)\right| \binom{M}{M/2} \qquad \forall 0 \le i, m \le M$

Proof of lemma 2:

Letting $k = i$ in equation 6, we get

$$|a_i(m)| \le \sum_{j=0}^{i-1} (|a_0(m-j)| + |a_0(i-m-j)|)\binom{i-1}{j}$$

since the summation for $k-1$ is less than the summation for k. Suppose that $1 \le m \le i-1$. Then, $1 \le i-m \le i-1$. Since $a_0(j) = 0, \forall j \ne 0$, we get

$$|a_i(m)| \le \left[\binom{i-1}{m} + \binom{i-1}{i-m}\right]|a_0(0)| = \binom{i}{m}|a_0(0)| \qquad \text{(EQ 10)}$$

where we have used Fact 1. For the case $m = i$, we have that $|a_i(i)| \le |a_0(0)|$ from equation 1. For $m > i$, $a_i(m) = 0$ and for $m = 0$, $a_i(0) = a_0(0)$. We have

$$MAX_{1 \le i, m \le M}\binom{i}{m} = \binom{M}{M/2}.$$

Therefore, the lemma follows. **QED**

This implies that if $|a_0(0)| \le 1/\binom{M}{M/2}$, then $|a_i(m)| \le 1.0, \forall 0 \le i, m \le M$. Hence the theorem is proved. **QED**.

For $M = 12$, a reasonable value, we should make $a_0(0) \cong 0.001$ to ensure that the computations never overflow. Since b will typically be small enough that b^{-1} is greater than 2.0, for efficiency, we should set $a_0(0)$ to be the largest power of two that is less than the calculated bound. This will allow the reflection coefficients to be scaled using shifts instead of multiplies. The cell was implemented with these considerations in mind.

4 Code Generation in Ptolemy

A domain in Ptolemy consists of Stars, Targets, and Schedulers. The Wormhole interface is optional and is used only if the domain being designed is going to be used with other domains in a single universe. The algorithm to be implemented is represented as a hierarchical dataflow graph. The graph is built hierarchically out of standard library Stars or user-defined Stars that can be linked in dynamically.

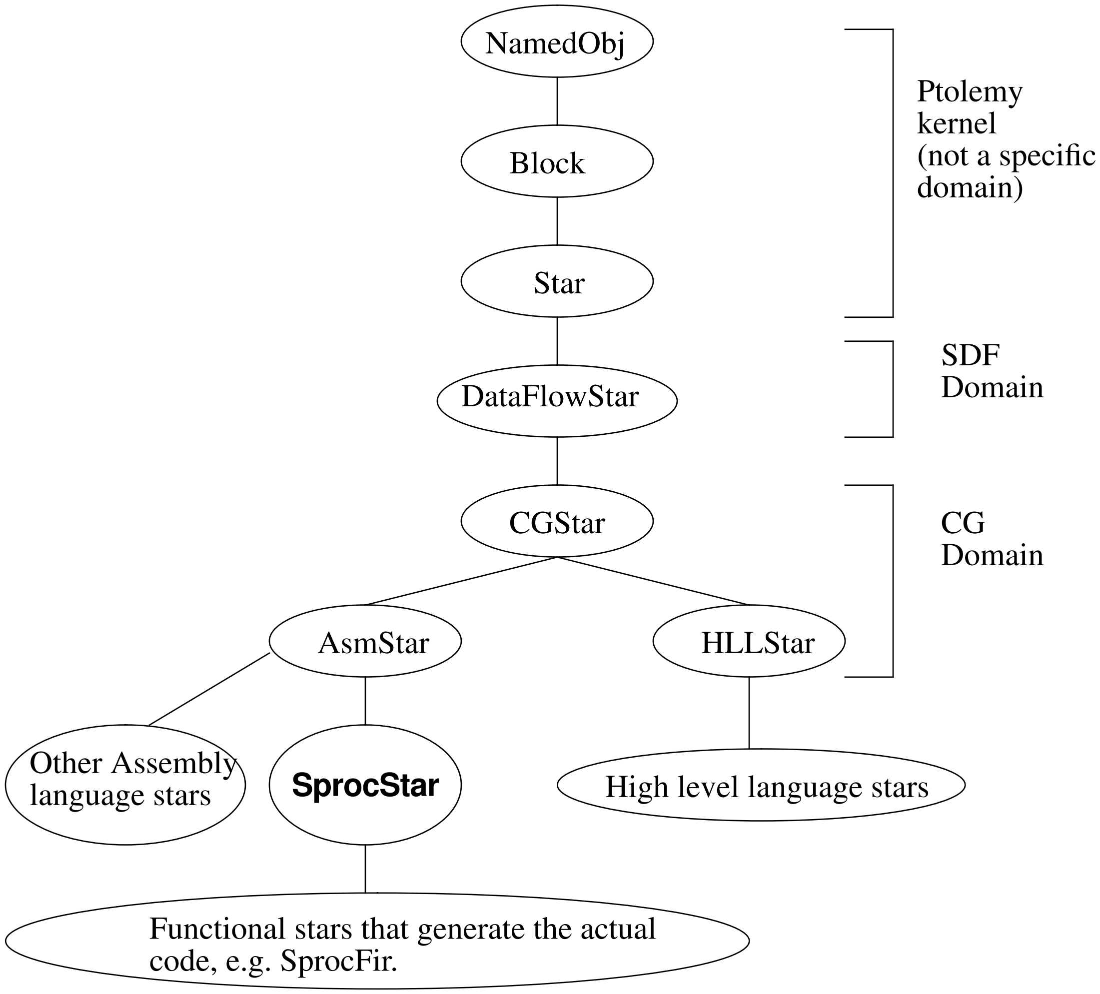

Fig 9. Derivation hierarchy for SprocStar. SprocStar is derived from AsmStar in the CG domain

4.1 Stars

Currently, there are two types of stars in Ptolemy, code-generation stars and simulation stars. Figure 9 illustrates the derivation hierarchy for stars in the Sproc domain. As can be seen, the CG domain (which is the baseclass for all code-generation domains) is derived from the SDF domain and the SDF domain is derived from the kernel as is every domain in Ptolemy. Note that this derivation hierarchy can change in the future when code-generation will be extended to more general models of computation like dynamic dataflow and boolean dataflow [Buc92]. Derivation here refers to the object-oriented inheritance concept in which one class can be *derived* from another class. The derived class will be a superset of its parent class meaning that methods and certain types of data members from the parent class are available to the derived class. In addition, the derived class can have new data members, new methods, and re-definition of methods from the parent class if those methods are *virtual* methods. In SprocStar some simple functions have been redefined; for example, the function that prints out state values is redefined to do it in hexadecimal.

Stars are relatively easy to write because there is a preprocessor language called *ptlang* that translates a star description to C++. The star description is given in a "boilerplate" format by defining states, ports, and various methods. *ptlang* then parses the file and generates the appropriate cc and h files that the compiler can understand. An example of this code for the star SprocGain is given in figure 11. The `go()` method in the star is called when the universe in run. In simulation stars, this method will perform the actual function of the star, whereas in code-generation stars, this method will call the `addCode(CodeBlock&)` method which will add the code contained in codeblocks to a code stream in the target. There are two other functions that are used: the `setup()` method and the `initCode()` method. The `setup()` method is called before `go()` and is responsible for setting up information that will be needed by the scheduler and memory allocator, such as the size of arrays, the number of samples read from a particular port etc. For example, in figure 12 the `setup()` method resizes the coefficients vector and sets a parameter for the input port indicating that D samples will be read, where D is the downsampling factor for the star. The `initCode()` method is called before the `go()` method and is used to generate code that appears before the main loop; code to initialize portholes for example (fig. 10).

The assembly language code is contained in codeblocks. A complete star description is given in figure 11 for the star SprocGain. Notice that the codeblock has constructs of the form $addr or $val. These are called *macros.* The addCode method will substitute the appropriate values when the codeblock is parsed. The most commonly used macros in Sproc stars are given in table 1.

Calling the `noInternalState()` method in the constructor tells the scheduler that the star has no dependence on past computations. This is to allow the scheduler the option of schedul-

```
codeblock (initfill) {
// initiallization code for $starname()
lda         #$val(fill)
ldb         #$addr(output)
ldd         #1
ldl         #$size(output)-1
$label(lp1):
sta         [B+L]
djne        $label(lp1)
}
initCode { if ((factor > 1)&(fill != 0.0) ) addCode(initfill); }
```

Fig 10. The InitCode method from SprocUpsample initializes the output port to a value.

ing multiple invocations of this star simultaneously on different processors, thereby extracting more parallelism (see section 4.4.2). The `execTime()` method is used by the scheduler to find out how much time this star takes to execute; the star writer must provide this.

One of the advantages of star-writing in Ptolemy is the possibility of conditional code generation. A good example of this is the `go()` method from the SprocLMS star (figure 12) which implements an adaptive filter using the LMS algorithm. There are two parameters for the star: the down-sampling factor and the number of delays on the error input (since the error is computed from the output at some stage, there is a feedback loop implying an error delay of at least 1.) There are various optimizations possible in the code for particular combinations of particular values of these parameters, and this results in the code being stitched together appropriately by the `go()` method. Also, note how the execution time is calculated in the `execTime` method for the various combinations of the input parameters. The execution times for each of the code sections are provided by the programmer.

Table 1. Commonly used macros in SprocStars

Macro	Use
$addr(memory_name)	Substitute memory address.
$addr(portname, offset)	Substitute memory address with added offset.
$val(state_name)	Substitute numerical value of the state.
$starname()	Substitute the name of the star.
$label(foo)	Create unique label from “foo” by adding a unique identifier.

```
defstar {
            name { Gain }
            domain { Sproc }
            desc {
The output is set to the input multiplied by a gain term.
The gain must be in [-1,1].
            }
            version { @(#)SprocGain.pl1.512/8/92 }
            author { P. Murthy }
            copyright {
Copyright (c) 1990, 1991, 1992 The Regents of the Univer-
sity of California.
            }
            location { Sproc stars library }
            explanation {
            }
            execTime {
                  return 5;
            }
            input {
                  name {input}
                  type {FIX}
            }
            output {
                  name {output}
                  type {FIX}
            }
            constructor {
                  noInternalState();
            }
            defstate {
                  name {gain}
                  type {FIX}
                  default {1.0}
                  attributes {A_RAM}
                  desc {Gain value}
            }
            codeblock (std) {
//_$starname()_begin:
            ldx   $addr(input)
            mpy   $addr(gain)
            nop
            nop
            stmh  $addr(output)
            }
            go {
                  addCode(std);
            }
}
```

Fig 11. Code for star SprocGain

```
setup {
     D = int(downsmpl);
     E = int(errorDelay);
     input.setSDFParams(D,D-1);
     if (int(flag)) {
           length = coef_vec.size();
           coef_vec.resize(2*int(length)+D*E);
           flag = 0;
     }
}

go {
     addCode(fir_init);
     if (D == 1) addCode(no_downsmpl_init);
     else if (D == 2) addCode(downsmpl_2_init);
     else addCode(downsmpl_n_init);
     addCode(update);
     if (E > 1) addCode(errordelay);
     addCode(fir_do);
}
exectime {
     int fir_init = 2;
     int no_downsmpl_init = 2;
     int downsmpl_2_init = 4;
     int downsmpl_n_init = 2+5*(D-1);
     int fir_do = 6+4*(length-1);
     int update = 10+5*(length-1);
     int errordelay = 1+3*(E-1)*D;
     int common = fir_init + update + fir_do;
     if (E == 1) {
           if (D == 1) return no_downsmpl_init + common;
           else if (D == 2) return downsmpl_2_init + common;
           else return downsmpl_n_init + common;
     }
     else {
           if (D == 1)
                 return no_downsmpl_init + errordelay + common;
           else if (D == 2)
                 return downsmpl_2_init + errordelay + common;
           else return downsmpl_n_init + errordelay + common;
     }
}
```

Fig 12. Conditional code generation in SprocLMS.

4.2 Targets

A target in Ptolemy defines those features of an architecture that are pertinent to code generation. It will specify how the generated code will be collected, specify and allocate resources such as memory, and define the code necessary for proper initialization of the platform. It may also specify how to compile and run the code if the platform being targeted is directly accessible from the platform that Ptolemy runs on. There are two types of targets in Ptolemy, multiprocessor targets and single processor targets. The difference is that a multiprocessor target will contain several child targets that are single processor targets. This allows code re-use in that once a single processor target has been defined for a particular processor, any architecture containing multiple such processors need only define a parent multi-target, making use of the single-targets that already exist. An example of this is the OMA architecture [Bie90] where the OMATarget is built from existing Motorola 96000 targets [Sri93]. The derivation hierarchy for targets is shown in figure 13.

CGTarget is the baseclass for all code generation targets including multiprocessor targets. The CGMultiTarget represents a target for a generic, fully connected architecture. No assumption

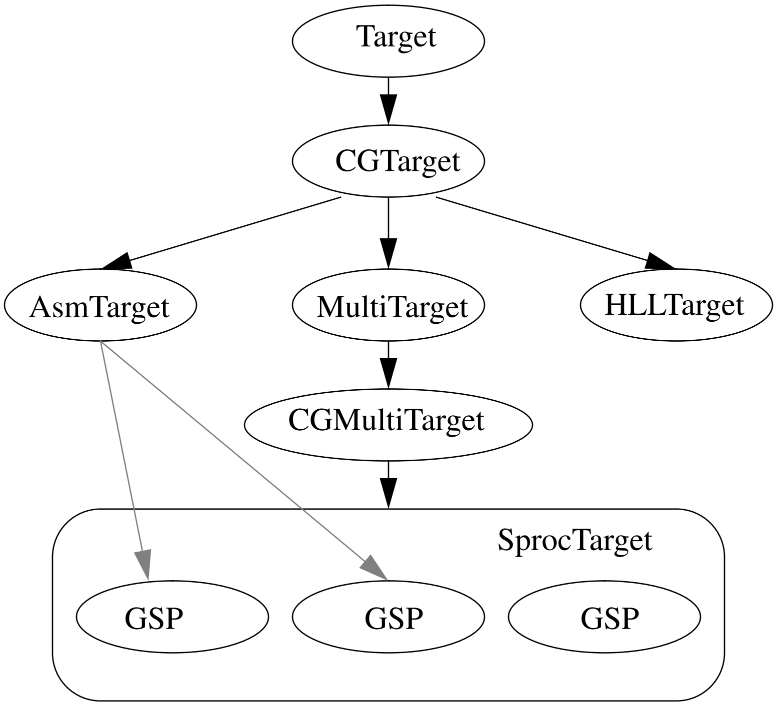

Fig 13. Derivation hierarchy for targets

is made about the memory resources available. The main job of this target is to define various parameters for the multiprocessor schedulers assuming the fully connected topology (communication costs, schedules for use of communication facilities etc.) Fully connected means that every processor in the network can communicate with every other processor in the network, and that the communication time is not a function of a particular communicating processor pair. This model is appropriate for the Sproc since the shared memory allows each processor to communicate with any other processor. The shared memory also implies that the communication time is not dependent on the processors. There is another target called CGSharedBus which assumes that communication among processors occurs through a shared bus and hence sets up the schedulers so that bus contention cost is taken into account. These two generic targets fit most parallel DSP architectures quite well. In addition to having methods for defining various parameters for schedulers, CGMultiTarget has a method for creating and initializing child targets, which are derived from AsmTarget, and a method for choosing and initializing the multiprocessor scheduler based on the user's selection. The general method by which a particular architecture is targeted is then to derive a parent target from CGMultiTarget or CGSharedBus. The parent target is responsible for any shared resources and the child targets are responsible for private resources local to the processor.

4.3 Schedulers

There are two multiprocessor schedulers of interest to the Sproc domain. One is Hu's level based list scheduling [Hu61] and the other is Gil Sih's declustering scheduler [Sih91]. There is also Gil Sih's dynamic level scheduler, but this assumes that processors have the ability to do computations in parallel with communication. Since the Sproc does not have this ability, it is not used. The parent target contains parameters that allow the user to choose between either of these schedulers. Since the schedulers behave differently for all but the most trivial graphs, this flexibility is useful for selecting the best schedule. In addition, the user can manually schedule the graph by setting a state in each star that tells the scheduler the processor on which the star should be scheduled. The schedulers are responsible for partitioning the graph across the different processors, taking interprocessor communication into account. Once the graph has been thus partitioned, sub-graphs are created for each processor and an order for firing the blocks is determined. Since each uniprocessor target has a uniprocessor scheduler object, which in turn has a data structure for storing the schedule, the scheduled sub-graphs are handed down to the uniprocessor schedulers' schedule data structure. This allows the child target to execute its sub-graph without knowing where it came from, thus preserving the modularity of the target hierarchy. The parent target, however, has to ensure that the uniprocessor scheduler in each child target does not get invoked since the schedule will be provided by the multiprocessor scheduler.

4.4 Interprocessor communication

4.4.1 Send/Receive

Interprocessor communication (IPC) is implemented using the send-receive model. In this model, send stars are inserted whenever one processor wishes to communicate data to another processor. The receiving processor will have a receive star that will receive the data. No assumption is made about how these stars will be implemented; that is part of the target/domain design. The scheduler is merely told the cost involved (i.e., the execution times of the send and receive). Once the sub-graphs are created, the scheduler inserts these communication stars at the appropriate places and schedules them along with the other stars in the sub-graph. For an example, see figure 14. A side benefit of this approach is that no sub-graph will have unterminated connections. This is important because each child target behaves exactly as if its sub-galaxy were the whole universe and unconnected terminals are not allowed in any graph the user might create.

4.4.2 Spread/Collect

In multirate applications, it is often possible to extract more concurrency out of the graph by executing multiple invocations of a block simultaneously on different processors. This is possible with actors that do not contain state variables; the noInternalState() directive mentioned in section 4.1 informs the scheduler of this. Figure 15 shows a simple example of this.

Since B1 (the first invocation of B) requires a sample from A2 and A2 is scheduled on processor P2, a way is needed of collecting the three samples required for B1 (2 from A1, 1 from A2). The reason we have to collect these samples is that the original node B only has one input

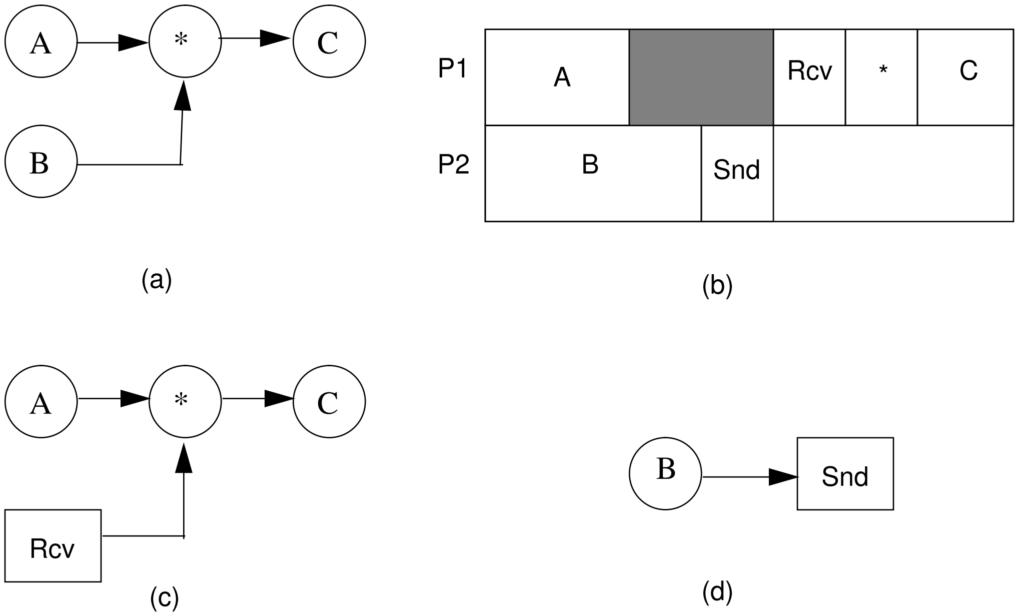

Fig 14. Send/Receive Model. a) A graph, b) A two processor schedule with IPC stars, c) sub-graph for P1, d) sub-graph for P2

arc. The code inside will make reference only to this input arc. In the APEG graph, however, each invocation of B has three arcs. We cannot allocate memory separately for these arcs because they won't be referenced in the codeblocks. Hence, a "Collect" star is inserted as shown in figure 15. This star tells the memory allocator that the three output locations on the Collect stars output arc should be aliased to the two outputs on A and one to the receive star output. Similarly, of the four samples produced by A2 and A3, one is sent to P1 while the rest are used by B2. This behavior can be implemented via a "Spread" star. The subgalaxies created by the scheduler are shown in fig.15d,15e.

5 The Sproc domain

The Sproc domain consists of a set of targets, an experimental scheduler, and a library of stars.

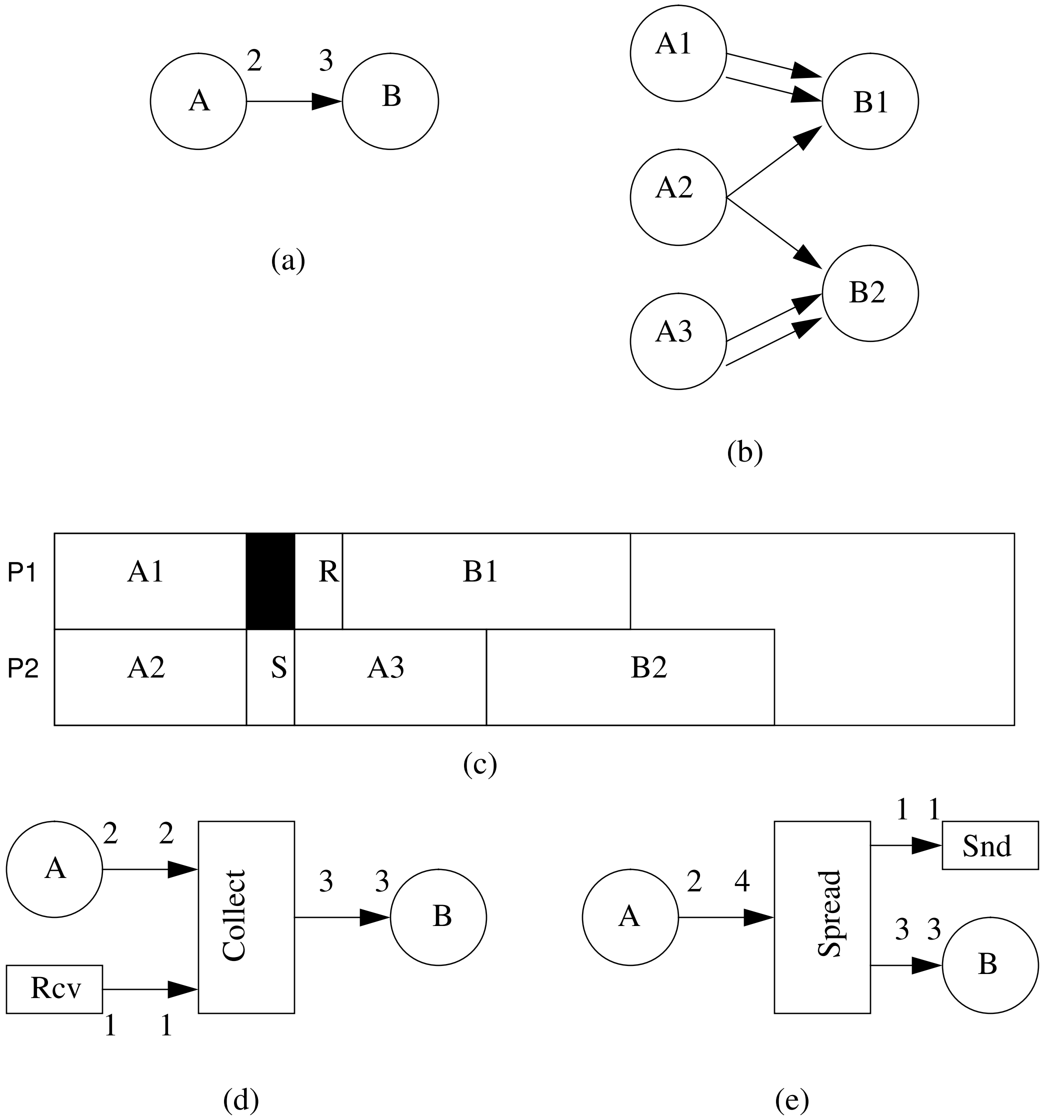

Fig 15. Spread/Collect **a)** Multirate graph, **b)** Acyclic precedence expanded graph (APEG), **c)** Schedule, **d)** Sub-galaxy for P1, **e)** Sub-galaxy for P2

5.1 SprocTarget

This is the main parent target object derived from CGMultiTarget as shown in figure 13. It controls the creation and pairing of IPC stars, it is responsible for initializing the child targets, and it coordinates the memory allocation process with the child targets. Figure 16 shows the sequence of function calls that take place. Only the important calls are shown. Some of the calls have a name beside them in brackets; this name indicates the class from which the called function will be executed because of the virtual mechanism. The basic function troika is `setup()`, `run()`, and `wrapup()` (recall that these are the main functions in stars as well). At what level in the derivation tree one of these gets called depends on the function. For example, SprocTarget has `setup()` and `wrapup()` but no `run()` because the definition in CGMultiTarget suffices. The scheduler object is chosen during `setup()` in the Target as shown. The schedule is computed during `setup()` and the code generation occurs during `run()`. Memory allocation is done by the `allocateMemory` function and functions such as `beginIteration()` and `endIteration()` are used for generating looping code. In SprocTarget the number of iterations is currently ignored but in general these methods should generate code taking that number into account because loop schedulers call the same functions to generate the looping code within a schedule. Since loop schedulers are not used in parallel code-generation, this is not an issue in the Sproc domain.

```
SprocTarget :: setup()
    create the memory object
    CGMultiTarget :: setup()
        createChild() [SprocTarget] //create child targets
        chooseScheduler()
        Target :: setup()
            scheduler :: setup() [SDFScheduler]
                computeSchedule() [ParScheduler] // main schedule
                    createSend()
                    createReceive()
                    pairSendReceive() [SprocTarget] // pair each snd w/ rcv
    headerCode()
CGMultiTarget :: run()
    CGTarget :: run()
        generateCode() [CGMultiTarget]
            scheduler()->compileRun() [ParScheduler]
                UniProcessor(i) :: prepareCodeGen()
                    convertSchedule() // copy par.sched to targets'
                    simRunSchedule() // trace schedule for right buffer sizes
                mtarget->prepareCodeGen [SprocTarget]
                    build a galaxy from several sub-galaxies
                    instantiate()
                        resizeBuffer() // for I/O stars based on makespan
                    firstChild->setup() [SprocGspTarget]
                        allocateMemory() [AsmTarget]
                        codeGenInit() [AsmTarget]
                            doInitialization() [SprocGspTarget]
                                writeInt,Fix etc
                UniProcessor(i) :: generateCode()
                    child(i)->generateCode() [SprocGspTarget]
                        headerCode()
                        initCode() // fire stars initcode methods
                        mainLoopCode() [CGTarget]
                            beginIteration() [SprocGspTarget]
                            scheduler()->compileRun [SDFScheduler]
                                fire go() methods in stars
                            endIteration() [SprocGspTarget]
                        Target::wrapup()
                            wrapup each star
                    addProcessorCode() // append code to stream in parent
SprocTarget :: wrapup()
    frameCode() // generate any framing code required for program
    printDataRam [SprocMemory] // generate memory initialization map
    Dump code to files
```

Fig 16. Function call sequence for code-generation

5.1.1 Send/Receive

As mentioned in section 3.2, the Sproc has a trigger-bus mechanism which could be of potential use for low-overhead IPC. A possible implementation using this mechanism is shown in figure 17.

Send:

```
LDA     $addr(input)
STA     $addr(location)
```

Receive:

```
wait:
LDA      WS
AND      #$val(location_bit)
JNE      $label(wait)
LDA      $addr(location)
STA      $addr(output)
```

"Location" is an address for the trigger-bus memory locations and "location_bit" is a number with a 1 is the bit position corresponding to the address of location. E.g., the trigger-bus locations are 0800h to 0817h. For location 0805h, the location_bit is 2^5 or 32. The WS register is loaded with an appropriate "bit-mask" at the beginning of the overall program loop. The bit-mask is simply a number with ones is the positions of address locations.

Fig 17. A possible Send/Rcv implementation

In the above implementation, we simply poll the appropriate bit in the WS register to see when a write has occurred. The trigger-bus memory location is used to transfer the data. The above implementation will be fine for homogenous graphs (meaning that all blocks produce and consume at most one sample) but will fail for multirate rate graphs because there is no mechanism for ensuring that the send star does not overwrite the previous data in `location` if it hasn't been read yet. The reason that it will work for homogenous graphs is that each send/receive pair is called only once; hence, if we do some sort of barrier synchronization at the beginning of the main loop (for instance, by waiting for the next sample to arrive), the send need not worry about overwriting good data. On the Sproc, this type of barrier synchronization is easy because the beginning of each sample period results in the least significant bit in the WS being cleared. Hence, by setting this bit at the beginning of the main loop, each processor can wait until the bit is cleared; this will be the indication that a new sample period has started. Of course, for this to work, the main loop on each processor should be completed within a sample period. Also, the WS register has to have the appropriate bit set again after the receive in order for the next receive to occur (this is assuming that a particular send/receive pair is invoked more than once in a schedule period). Again, for homogenous graphs, this won't be a problem since with barrier synchronization, we can assume that all send/receive pairs have executed and reload the bit mask into the WS register at the beginning of the main iteration. It turns out that setting a bit in the WS register and ensuring that the send star does not overwrite previous data requires more overhead than the usual semaphore approach. In particular, a single instruction for setting a bit in the WS register does not exist; so three instructions would be required to set a bit. In addition to these disadvantages, there are only 18 trigger bus locations which puts a limit on the number of IPC stars, an undesirable prospect.

Therefore the approach used is a semaphore mechanism. In this scheme, the Send star will write the datum on its input to the output of the Receive star if the semaphore location contains a one. Otherwise, it waits until the condition is met. After the write to the Receive's output, the semaphore location is reset to a zero. The Receive star merely examines the semaphore location and if it is one (meaning that a new datum has not arrived), it waits until the location has a zero. The code for doing this is shown in figure 18.

Send:

```
ldl     #0
ldx     $addr(input)
$label(wait):
lda     $ref(pollAddr)
jeq     $label(wait)
stl     $ref(pollAddr)
stx     $ref(output)
```

Receive:

```
lda       $addr(pollAddr)
jne       $label(wait)
std       $addr(pollAddr)
```

Fig 18. Send/Rcv implementation

It is worth noting that a rather roundabout method is used to enable the Send star to write directly to the output on the *Receive* star. Recall that a Send star does not have an output port and a Receive star does not have an input port because one is a sink while the other is a source in the sub-galaxy in which each exists. However, the macro function that actually computes the address of a location can be re-defined in the Send star because this is a virtual function in CGStar. The re-defined method simply looks up the Receive star's output location even though the codeblock in the Send star refers to a non-existent “output” port. The parent target ensures that each Send star has a pointer to the Receive star that it has been paired with. The same technique is used for the semaphore location as well, since this has to be a state in either the Receive or the Send.

The overall execution time is 9 cycles, with the Send star requiring 6 cycles. An advantage of this scheme is that the semaphore location can be allocated just like any other memory location. It does not require a special region of the memory like the trigger-bus region. Even though it is a waste of memory to use 24-bit locations for storing single bits, a single location or register for multiple bits cannot be used without instructions for bit setting and clearing.

5.1.2 Memory Allocation

Since the Sproc has a shared-memory architecture, only one memory object is required in the Target definition. The parent target creates this object during `setup()` and a pointer to the object is passed to each child target when the child target is created so that each child has access to the same object. The memory allocator functions are all called by AsmTarget (the base class for the child targets). The order in which this occurs is as follows: requests for all memory are posted first. For the child this means all the states and ports in its subgalaxy. Then the `performAllocation()` method is called to process all the requests. The request-allocation sequence, however, can only happen once for each memory object because of the way the allocation routines are written. So for a shared memory object, just having each child go about memory allocation on its turn fails because the request-allocate sequence will occur once for each child and we want it to occur only once. Therefore, the parent intervenes with a method called `prepareCodeGen()`, which is called before the `generateCode()` method (the `generateCode()` method is the usual way of passing control to the child targets for memory allocation and code generation). In `prepareCodeGen()`, one galaxy is created out of the subgalaxies for each of the processors. This galaxy is then re-instantiated because the original instantiation is lost when the sub-galaxies were created in the first place. After the re-instantiation, the memory allocation methods are called through one of the child targets. Now when the `generateCode()` method is called, the child targets simply fire the stars in their subgalaxies and add the generated code directly to a stream in the parent target. The `wrapup` method in the parent then writes the contents of the stream to a user-specified file. The initialization code for the memory is also generated during `wrapup()` via a method in the SprocMemory class.

Normally the initialization is done in AsmTarget via “org” like statements. The Sproc compiler, however, does not have such a statement, and the only method for initializing the memory is by statements such as `variable fixed foo = 1.5`. Moreover, these statements have to occur in order of the memory map since there is no way (without run-time code) to initialize specific memory locations. The initialization method in AsmTarget does not initialize port locations, something that is required for the Sproc. Hence, the `doInitialization()` method is redefined in the child targets to permit port initialization. `doInitialization()` calls virtual methods such as `writeInt()` (for writing integers), `writeFix()`, and `writePort()`. These would normally generate compiler statements such as `dc 1.5` (on the Motorola 56000) but here these methods write the values to global arrays maintained in the parent target. This is done so that the memory initialization can be done in order of memory rather than the order of stars. Three arrays are maintained: one for the name of the state or port, one for the type, and one for the value. These arrays are then read by the method `printDataRam()` in the SprocMemory class during the `wrapup()` in the parent target and the “`variable`” declarations are generated and written to another file. Having the names in the initialization serves another useful purpose, namely to allow efficient debugging and change of state values while the program is running on the chip through the SDI interface on the PC.

5.1.3 I/O

Programmable buffering is used for the input/output stars. One weakness of the SDF paradigm is that since there is no concept of time, it is assumed that source stars can be fired at any time. This is obviously not true in the case of A/D stars where the star is actually synchronous with an external sample rate. Therefore the schedulers, which are based on SDF, will schedule input/output stars without taking the sample rate into account. When a schedule period contains multiple invocations of an input star, it is crucial that some method be used to ensure that samples are not lost. In the Motorola 56000 domain, interrupts are used if there are multiple invocations. The Sproc has programmable buffers where the size can be set before-hand; this loosens the constraint of an input block having to fire every sample period to an input block having to fire N times in N sample periods (if the schedule requires N firings of this input block). This eliminates the need for interrupts and the loss in processing time that results. The buffer size is set automatically by the target by calculating the number of sample periods over which the schedule will run.

5.1.3.1 A possibility for future work

Even though programmable buffering is a better solution than interrupts, the best solution would be for the scheduler to take into account periodicities of certain stars. This could then result in schedules where the I/O stars are executed in each sample period, eliminating the need for any buffering at all. This is generally a difficult thing to do because not only does the I/O star have to fire every sample period but all stars that work at the same rate must also fire every sample period. This is especially difficult since all such stars may not be on the same processor and some may even have execution times greater than a sample period. If I/O stars can be scheduled on different processors for different invocations, then meeting the no-buffering goal might be possible. But given that the number of different combinations of schedules for scheduling N invocations of an I/O star on P processors is P^N, the problem may not have a polynomial time solution in general. Figure 19 shows a simple example of an efficient schedule if source stars are scheduled on different processors.

5.2 Other Targets

Two other targets have been written. One target generates code in the form of high level function calls. In other words, instead of in-line assembly code in the program, a statement containing the name of the block, its states, and its input and output ports is generated. The Sproc compiler then generates code for these functions from the Sproc library of assembly language cells. Therefore this requires there to be an exact match between Ptolemy stars and cells in the Sproc library. This requirement is not very attractive because as discussed before, stars in Ptolemy are more general in terms of their multirate capability and their ability to do conditional code generation.

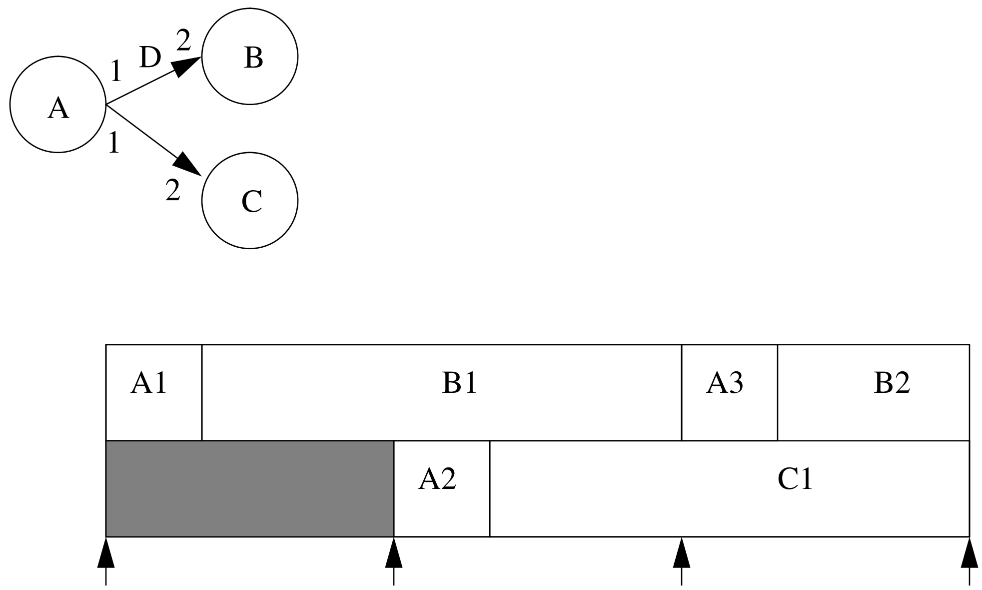

Fig 19. Example of source stars fired on different processors for alternate samples.

The other target is a test for an experimental scheduler under study. This target just sets up the scheduler object and does not do anything else. It illustrates the ease with which new targets and schedulers can be written based on existing code.

5.3 Some examples

Some applications developed include the Karplus-Strong algorithm for simulating plucked string sounds (this is done in a way that turns the PC into a mini keyboard), reverberation of an audio signal, a three band QMF filter bank, and ADPCM encoding and decoding of speech using LMS filters. Three of these are described below.

5.3.1 Plucked strings

The Karplus-Strong algorithm is a simple, elegant way of simulating plucked string instruments like guitars and harps [Moo90]. A plucked string can be modeled as a digital delay line with a low- pass filter in a feedback loop. The delay line is initially filled with random samples; this signal represents an excitation signal that is rich in harmonics. As the signal circulates in the feedback loop, the low pass filter attenuates the higher frequencies. This results in a signal that sounds like a plucked string instrument when played through a speaker. The length of the delay line, which is equivalent to the length of the string, determines the pitch of the resulting sound. While the delay line can only contain integer values, it is possible to get fractional delays by including an all-pass filter in series with the delay. This would then allow the model to be tuned to desired frequencies quite precisely.

Figure 20 shows seven of these modules connected to a D/A converter. The pulse star in each module generates a rectangular pulse that has a duration equal to the length of the delay line for that module. The values of the delays in each module are chosen to roughly approximate an incomplete scale. Through the access port on the Sproc board, the user can set a flag in the pulse star. The pulse star generates a pulse only if this flag is set. After generating the pulse, the star will reset the flag and generate zeros. Therefore, by tying commands that set this flag to particular keys on the IBM PC keyboard, one gets the effect of a musical keyboard by pressing different keys that trigger one or more of the modules. Note that even though the behavior of the pulse star appears to be asynchronous, it is not really being used in that manner since it is fired on every iteration. The star decides, based on its flag, whether to generate a pulse or not, rather than generating one periodically. This application is partitioned onto all four processors and the makespan is exactly 200 cycles.

5.3.2 QMF Filter bank

Figure 21 shows a three-band quadrature mirror filter (QMF) bank. The principle behind QMF filter banks is as follows. Signals of practical interest, for example, music signals or speech signals, do not contain equal amounts of energy in all parts of the spectrum. Hence, if it were possible to separate the signal into components for different frequency regions, each frequency region could be coded separately based on the amount of energy it has in that part of the spectrum. One way of obtaining these components is to recursively divide the spectrum by half at each stage. In other words, a high-pass and a low-pass component is created by filtering and then one of these is further subdivided until one has enough bands. For speech signals, where the energy is typically in the low-pass region, it's the low-pass component that is subdivided at each stage. Note that by dividing the spectrum by two, we obtain two band-limited signals that can be deci-

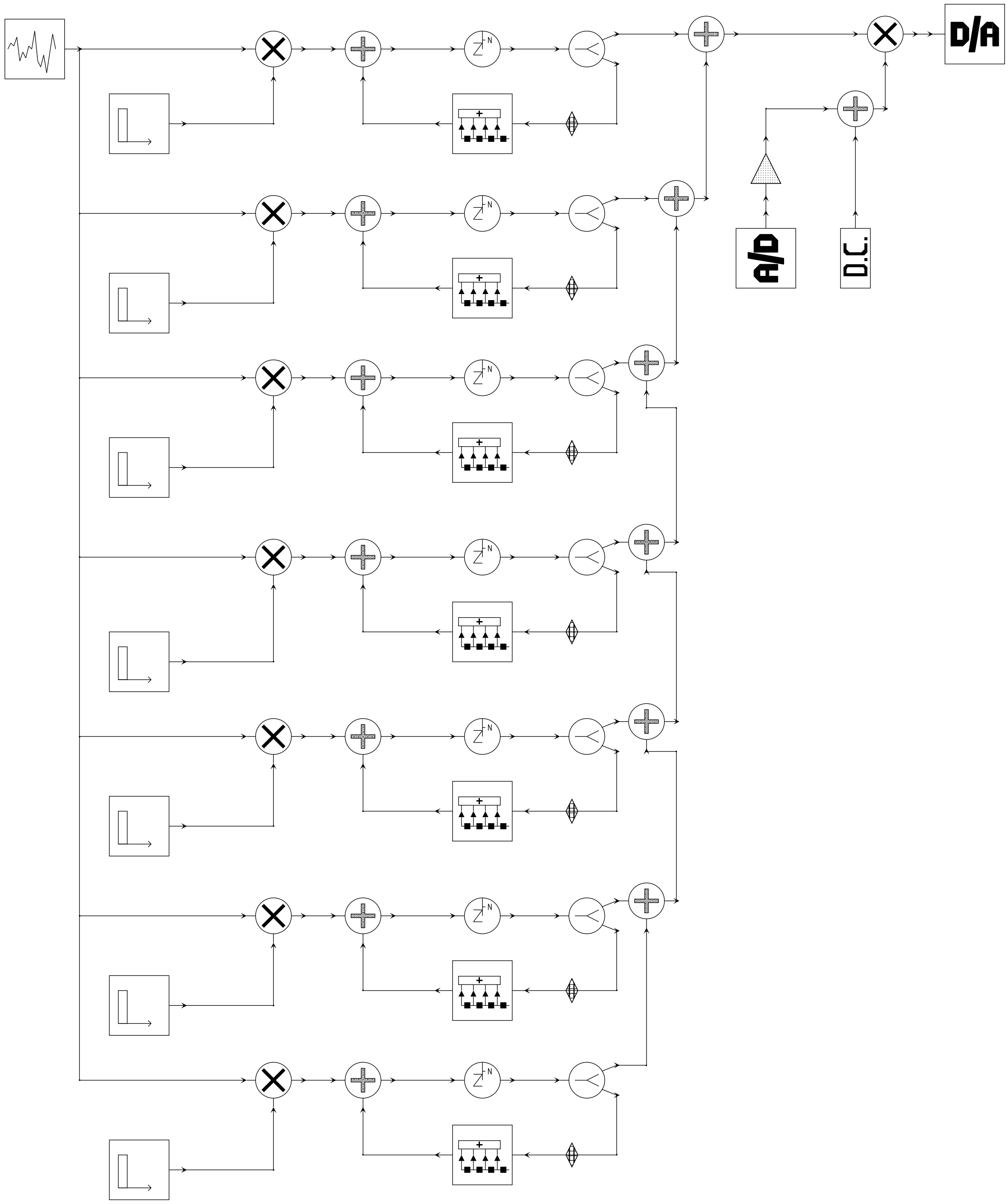

Fig 20. The multi-string bank. Each bank generates a plucked string sound at a given frequency using the Karplus-Strong algorithm

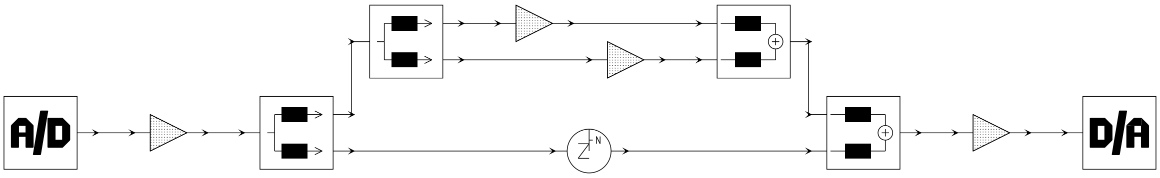

Fig 21. A 3-band QMF filter bank. The left side consists of the analysis filters and the right side the synthesis filters

mated by two. So even though we have two signals, each is transmitted at half the original rate. The reduction in the transmission rate comes about because fewer bits will typically be required to code one of the components. The choice of the analysis filters is very important since the reconstructed signal usually differs from the original signal due to aliasing, amplitude distortion, and phase distortion. However, the filters can be designed in such a way that some (or all) of these distortions are eliminated. See [Vai93] for a good introduction to the theory and design of filter banks.

Figure 21 contains 2 stages: the low-pass component of the first stage is subdivided again into two bands. We get three bands overall, and in the example, we can vary the gain in each of the paths to get the effect of an audio equalizer. The FIR filters that implement the high-pass and low-pass filters (figure 22) also do the decimation (interpolation) in the analysis (synthesis) stages. This is because the implementation of the filter is much more efficient if the decimation is taken into account; samples that are thrown away, as is the case during decimation, are not computed, and multiplies with zeros, as is the case during interpolation, do not occur. More stages are not used because each additional stage causes the code space to more than double because of the decimation/interpolation. Given that there are only 1k words of program memory available, anything more than 2 stages is currently not feasible. However, if loop scheduling techniques [Bha93] are extended to the multiprocessor case (possible future work), then the code space problem might be mitigated.

A small fraction of the generated code for this universe is given in the appendix (all of it isn't given since many trees would have suffered). The filter bank runs at a sample rate of 9.7kHz.

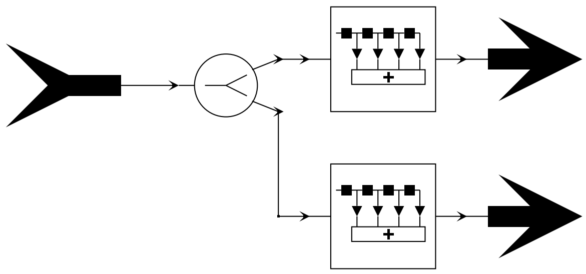

Fig 22. The analysis block expanded; the top is a low-pass FIR and the bottom a high-pass FIR

5.3.3 ADPCM coding of speech

A third example developed is an adaptive differential pulse code modulation (ADPCM) coder for speech. The coder/decoder shown in figure 23 achieves a bit rate of about 29000 bits per second at a sampling rate of 9.7kHz and three bits per sample. The idea behind ADPCM coding is to code an error signal that is obtained as the difference of the input sample with a predicted version of the input sample [Rab78][Jay84]. In other words, because speech signals have a high degree of correlation for short periods of time, it is possible to use a certain number of past samples (say 14) to predict the next sample. If this prediction is good, then the error signal will be small, and we can use many fewer bits to encode the error signal. The LMS filter shown in the figure does linear prediction from past values of the quantized input samples using the LMS algorithm with an instantaneous estimate of the cross-correlation between the error and the input. The adaptive quantizer codes the error signal to 8 levels (3 bits) using an estimate of the power in the error signal to determine the step size. If the actual bits were transmitted (instead of the levels), then we would need to transmit the step size as well so that the decoder could reconstruct the quantized error signal. Note that the feedback-around-quantizer structure ensures that both of the LMS filters adapt using the same error signal; this avoids having to transmit the filter coefficients periodically. Figure 24 shows a Gantt chart display that illustrates the partitioning of the blocks onto three of the four processors.

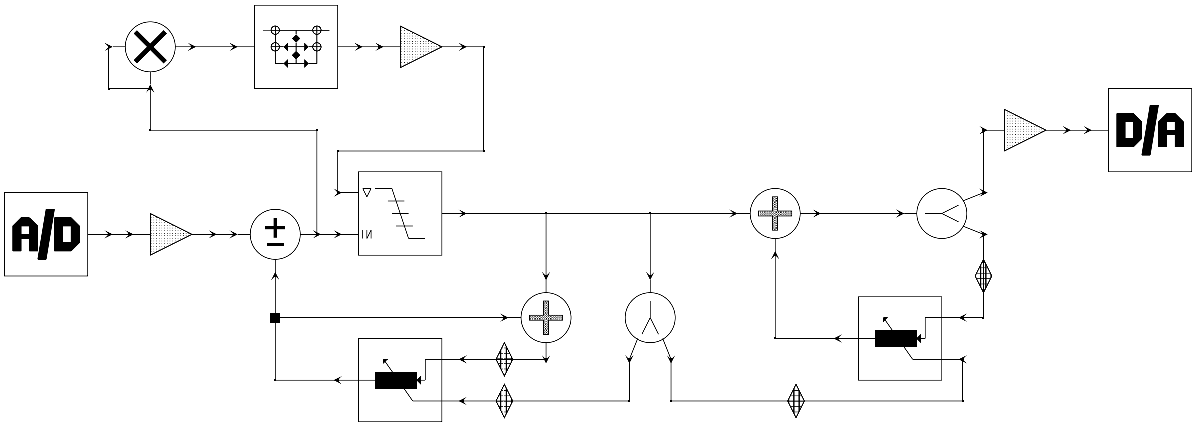

Fig 23. The ADPCM coder schematic. The left half is the coder and the right half is the decoder.

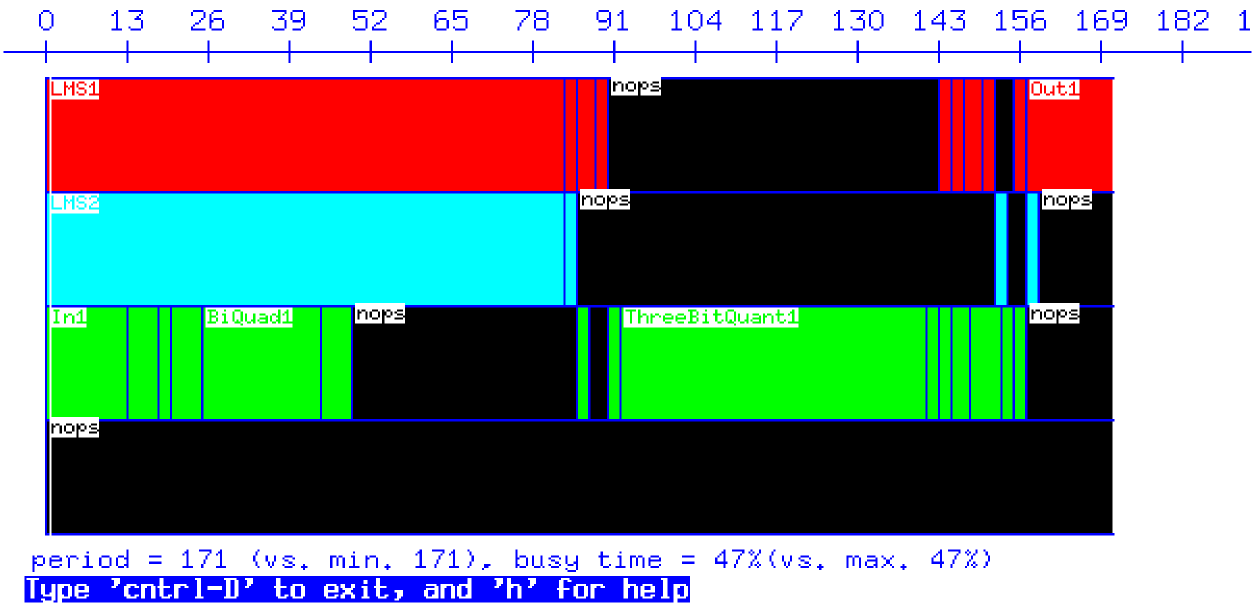

Fig 24. Gantt chart showing the partitioning of the blocks for the ADPCM example

6 Conclusion

A new domain targeting a parallel architecture has been built and demonstrated in Ptolemy. The architecture is the shared-memory, 4-processor Sproc DSP made by Star Semiconductor. Limitations of the synthesis environment provided by Star were explored; these were found to be a lack of multirate capability and SSIMD scheduling. The first limitation requires blocks to be written with synchronization issues in mind. The second limitation is more serious in that it requires large-granularity cells to be written in a manner that allows the computation to occur over several sample periods. An application was developed to illustrate these limitations. Advantages of the code-generation environment in Ptolemy over the environment provided by Star Semiconductor are:1) the SDF model which allows multirate applications to be expressed neatly, 2) an object-oriented approach that permits any of several schedulers and any of several targets to be used, or for new ones to be incorporated easily, 3) an ability for blocks to do conditional code-generation, and 4) the existence of schedulers in Ptolemy that do not have the limitations of SSIMD scheduling. A library of stars for the domain has been developed and the library is extensive enough to allow applications such as QMF filter banks and ADPCM speech coders to be rapidly prototyped. As is the case with all Ptolemy domains, the library can be easily extended once the user has the required knowledge of assembly language programming for the Sproc.

7 Acknowledgments

I am grateful to Star Semiconductor and the State of California MICRO program for supporting this project. I would also like to thank Soonhoi Ha for being ever-willing to answer questions and fix bugs in the schedulers and the CG kernel. Without his help the Sproc domain might not have been a reality! Last but not least, I would like to thank my advisor, Professor Edward Lee, for his many helpful comments for improving this report. I am grateful for his patience!

8 References

[Alm92] The Almagest, EECS/ERL ILP Office, Software Distribution, 479 Cory Hall, UCB 1992

[Bar82] T.P.Barnwell III, C.J.M.Hodges, M.Randolf, "Optimal Implementation of Single Time Index Signal Flow Graphs on Synchronous Multiprocessor Machines", Proc. of the ICASSP, Paris France May,1982

[Bar83] T.P.Barnwell III, D.A.Schwartz, "Optimal Implementations of Flow Graphs on Synchronous Multiprocessors", Proc. 1983 Asilomar Conf. on Circuits and Systems, Pacific Grove, CA, Nov. 1983

[Bha93] S.S.Bhattacharyya, J.Buck, S.Ha, E.A.Lee, "A Compiler Scheduling Framework for Minimizing Memory Requirements of Multirate DSP systems represented as Dataflow Graphs", Tech. Report, Memorandum UCB/ERL M93/31, Electronic Research Laboratory, College of Engineering, UC Berkeley, Berkeley, CA 94720, 1993

[Bie90] J.Bier, S.Sriram, E.A.Lee, "A Class of Multiprocessor Architectures for Real-Time DSP", VLSI Signal Processing IV, 1990

[Buc93] J.Buck, S. Ha, E.A.Lee, D.G.Messerschmitt, "Ptolemy: a Framework for Simulating and Prototyping Heterogenous Systems", to appear in the International Journal of Computer Simulation, special issue on "Simulation Software Development," 1993

[Buc92] J.Buck, E.A.Lee, "The Token Flow Model", Dataflow workshop, Hamilton Island, Australia, May 1992

[Den80] J.B.Dennis, "Dataflow Supercomputers", Computer, 1980

[Gen90] D. Genin, P.Hilfinger, J.Rabaey, C.Scheers, H. de Man, "DSP Specification Using the Silage Language", Proc. of the ICASSP 1990

[Hay86] S.Haykin, "Modern Filters", MacMillan, 1986

[Hil89] P.N.Hilfinger, "Silage Reference Manual, DRAFT Release 2.0", Computer Science Division, EECS Dept., UC Berkeley, Berkeley, CA 94720, 1989

[Hu61] T.C.Hu, "Parallel Sequencing and Assembly Line Problems," Operations research 9(6), November 1961

[Jay84] N.S.Jayant, P.Noll, "Digital Coding of Waveforms", Prentice Hall, 1984

[Lee86] E.A.Lee, "A Coupled Hardware and Software Architecture for Programmable DSPs", Ph.D. thesis, UCB, 1986

[Lee87] E.A.Lee, D.Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing", IEEE Transactions on Computers, January 1987

[Men90] Mentor Graphics Corp., "DSP Station User's and Reference Manual", 1992

[Mes84] D.G.Messerschmitt, "Structured Interconnection of Signal Processing Programs," Proceedings of the Globecom 1984

[Moo90] R.Moore, “Elements of Computer Music”, Prentice Hall, 1990

[Pow92] D.B.Powell, E.A.Lee, W.C.Newmann, “Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams,” Proceedings of ICASSP 92, March 1992

[Rab78] L.R.Rabiner, R.W.Schafer, “Digital Processing of Speech Signals”, Prentice Hall, 1978

[Sih91] G.Sih, “Multiprocessor Scheduling to Account for Interprocessor Communication”, Ph.D. thesis, UCB 1991

[Sri93] S.Sriram, E.A.Lee, “Design and Implementation of an Ordered Memory Access Architecture”, ICASSP 93

[SS92] Sproc Signal Processor Databook, Star Semiconductor, 1992

[Vai93] P.P.Vaidyanathan, “Multirate Systems and Filter Banks”, Prentice Hall, 1993

Appendix: Code for QMF filter bank

```
/***************************************
#User: murthy
#Date: Thu Mar 11 16:51:06 1993
#Target: default-Sproc
#Universe: qmf
***************************************/

asmblock qmf {} ()
#include “qmf.var”
duration 0;
begin

// ----------------------------------------
// Code for GSP1:
// ----------------------------------------

_$start_gsp1:
            ldd   #1
// initiallization code for In0
            ldf   #1
            lda   #8
            ldb   #096fh
            stb   0991h
            ldx   #15h
            stf   440h
            sta   441h
            stb   442h
            stx   443h
// initiallization code for Out0
            lda   #8
            ldb   #0977h
            stb   099dh
            ldx   #5h
            ldf   #1
            sta   451h
            stb   452h
            stx   453h
            stf   454h  // decimation register
            stf   456h  // wait trigger mask
main_loop_gsp1:
// Synchronize on incoming sample
            ldws  #1
_sync_gsp1:
            jwf   _sync_gsp1
// code from star code_proc0.In0 (class SprocIn)
//_In0_begin:
begin_0:
            lda   2048
            and   #16383
            cmp   0991h
            jeq   begin_0
            lda   0991h
            ldb   A
            add   #1          // buf_size = 1
            cmp   #096fh+8
            jlt   ok_1
            lda   #096fh
```

```
ok_1:
            sta   0991h
            lda   [B]
            sta   0994h
.
.
            sta 099dh
            stx 1109
// code from star code_proc0.Receive1 (class SprocReceive)
//_Receive1_begin:
wait_33:
            lda   09a0h
            jne   wait_33
            std   09a0h
// code from star code_proc0.Receive0 (class SprocReceive)
//_Receive0_begin:
wait_34:
            lda   099fh
            jne   wait_34
            std   099fh
// code from star code_proc0.Add20 (class SprocAdd2)
//_Add20_begin:
            lda   0999h       // 1st input -> A
            add   0984h       // 2nd input added to A
            sta   099ah
// code from star code_proc0.Gain2 (class SprocGain)
//_Gain2_begin:
            ldx   099ah
            mpy   099bh
            nop
            nop
            stmh  099ch
// code from star code_proc0.Out0 (class SprocOut)
//_Out0_begin:
            ldb 099dh
            lda 099ch
            sta [B]
            ldx #-1
            lda 099dh
            cmp #0977h+8/2
            jeq cont_35
            ldx #0
cont_35:
            add #1      //buf_size = 1
            cmp #0977h+8
            jlt ok_36
            lda #0977h
ok_36:
            sta 099dh
            stx 1109
            jmp   main_loop_gsp1

// ----------------------------------------
// Code for GSP2:
// ----------------------------------------

_$start_gsp2:
            ldd   #1
// initiallization code for DelayN0
            lda   #0818h
            sta   09a2h
```

```
main_loop_gsp2:
// Synchronize on incoming sample
.
.
Variable Declarations generated for QMF (Not all are shown)
-----------------------------------------------------------
variable fixed Fir7_coef_vec = -0.00122400000691414;// 094bh, length 36
variable fixed Fir7_coef_vec_1 = -0.000698000018019229;// 094ch
variable fixed Fir7_coef_vec_2 = 0.011832999996841;// 094dh
variable fixed Fir7_coef_vec_3 = 0.0116820000112057;// 094eh
variable fixed Fir7_coef_vec_4 = -0.0712829977273941;// 094fh
variable fixed Fir7_coef_vec_5 = -0.0309859998524189;// 0950h
variable fixed Fir7_coef_vec_6 = 0.226242005825043;// 0951h
variable fixed Fir7_coef_vec_7 = 0.0692479982972145;// 0952h
variable fixed Fir7_coef_vec_8 = -0.73157399892807;// 0953h
variable fixed Fir7_coef_vec_9 = 0.73157399892807;// 0954h
variable fixed Fir7_coef_vec_10 = -0.0692479982972145;// 0955h
variable fixed Fir7_coef_vec_11 = -0.226242005825043;// 0956h
variable fixed Fir7_coef_vec_12 = 0.0309859998524189;// 0957h
variable fixed Fir7_coef_vec_13 = 0.0712829977273941;// 0958h
variable fixed Fir7_coef_vec_14 = -0.0116820000112057;// 0959h
variable fixed Fir7_coef_vec_15 = -0.011832999996841;// 095ah
variable fixed Fir7_coef_vec_16 = 0.000698000018019229;// 095bh
variable fixed Fir7_coef_vec_17 = 0.00122400000691414;// 095ch
variable fixed Fir7_coef_vec_18 = 0.0;// 095dh
variable fixed Fir7_coef_vec_19 = 0.0;// 095eh
variable fixed Fir7_coef_vec_20 = 0.0;// 095fh
variable fixed Fir7_coef_vec_21 = 0.0;// 0960h
variable fixed Fir7_coef_vec_22 = 0.0;// 0961h
variable fixed Fir7_coef_vec_23 = 0.0;// 0962h
variable fixed Fir7_coef_vec_24 = 0.0;// 0963h
variable fixed Fir7_coef_vec_25 = 0.0;// 0964h
variable fixed Fir7_coef_vec_26 = 0.0;// 0965h
variable fixed Fir7_coef_vec_27 = 0.0;// 0966h
variable fixed Fir7_coef_vec_28 = 0.0;// 0967h
variable fixed Fir7_coef_vec_29 = 0.0;// 0968h
variable fixed Fir7_coef_vec_30 = 0.0;// 0969h
variable fixed Fir7_coef_vec_31 = 0.0;// 096ah
variable fixed Fir7_coef_vec_32 = 0.0;// 096bh
variable fixed Fir7_coef_vec_33 = 0.0;// 096ch
variable fixed Fir7_coef_vec_34 = 0.0;// 096dh
variable fixed Fir7_coef_vec_35 = 0.0;// 096eh
variable fixed In0_fifo[8];// 096fh
variable fixed Out0_fifo[8];// 0977h
variable fixed Send0_input[2];// 097fh
variable fixed Send1_input[2];// 0981h
variable fixed Receive1_output[2];// 0983h
variable fixed Send2_input[2];// 0985h
variable fixed Receive2_output[2];// 0987h
variable fixed Receive3_output[2];// 0989h
variable fixed Add21_input2[2];// 098bh
variable fixed Send4_input[2];// 098dh
variable fixed Send5_input[2];// 098fh
variable integer In0_out_ptr = 0;// 0991h, length 1
variable integer In0_fifo_buffer_length = 0;// 0992h, length 1
variable integer In0_fifo_index_length = 8;// 0993h, length 1
variable fixed Gain0_input;// 0994h, length 1
variable fixed Gain0_gain = 0.5;// 0995h, length 1
variable fixed Gain1_input;// 0996h, length 1
variable fixed Gain1_gain = 1.0;// 0997h, length 1
variable fixed Fir2_input;// 0998h, length 1
```