Standard Cells for Hardware Synthesis of LUCID Programs

Abhay Kejriwal
Texas Instruments, Inc.

Ben M. Huey
Arizona State University

Abstract

We describe the design of standard cell libraries in the hardware description language VHDL for Lucid operators using both demand-driven and data-driven dataflow computation. These standard cells are useful for synthesizing a VLSI system from a given design description given in Lucid. Almost all the operators for the Lucid language, and at least one operator from each of the different classes of Lucid operators for the data-driven and demand-driven model was implemented. Interfaces for hybrid models were designed and a few hybrid structures were implemented to verify the functionality and interfacing capabilities of the cells. While the use of Lucid as a hardware synthesis language seems to be promising, much additional work will be required before it becomes an effective tool and can be widely accepted.

I. Introduction

The trend in the design of VLSI systems has been to reduce design cycle time and at the same time improve performance. Reduced design cycle time is achieved by designing at a higher level of abstraction and using CAD tools to translate the design into a low level implementation. Improved performance is achieved by using more advanced, high-speed technologies. Further performance improvement can be achieved by utilizing parallelism. The use of faster technologies, automated CAD tools and parallelism has provided multi-fold improvement in performance.

However, current high level hardware description languages require explicit management of parallelism that exceeds the ability of the human designer unless the design is highly regular. A tool supporting design at a high level of abstraction with implicit parallelism will be more suitable to design a parallel architecture system and should encourage design with a new view of design space.

Lucid encourages algorithm design that maximizes parallelism constrained only by the data dependencies of a program. This property of Lucid makes it an obvious choice for describing massively parallel architecture systems. Objects in Lucid are infinite sequences and a Lucid program is an expression. This makes Lucid especially suitable for digital signal processing and real time processing applications [3]. The ultimate goal of this work is to develop Lucid as a synthesis language for designing parallel architecture systems; in this paper we address the more modest aim of building a foundation for synthesis of Lucid programs as hardware.

We have designed standard cell libraries for Lucid operators using both demand-driven and data-driven dataflow computation. These standard cells have been designed in the IEEE Std. 1076 VHDL and are useful for generating a VLSI system synthesized from a given description in Lucid. VHDL is one of the two languages now in almost universal use as a language for digital design [1],[2]. The choice of VHDL was influenced by the following factors: (1) The availability of various well-developed synthesis tools. Instead of having to develop a complete synthesis tool, only a simple tool will be required to convert a given Lucid program into VHDL description. This reduces the cost of development as well as the time required to develop the tool. (2) VHDL is widely accepted in industry and the design will be portable. (3) Design verification can be done by simulating the description before actually building the system. (4) VHDL allows mixing of descriptions at different levels of abstraction. Thus, if the product is a part of a larger system, an abstract level behavioral description of the product can be used to develop the complete system.

To the best of our knowledge this work is the first attempt to use Lucid as a hardware design language. Of particular interest are the use of an alternative set of primitive operations in hardware design, and the implications for datapath control and buffering that arise from the underlying computational models employed by Lucid.

II. Classification of Lucid Operators

Lucid operators have been classified into four broad categories [3] based on whether an operator produces an incomplete or complete data object and whether the subset of input data required to produce a result can be syntactically determined or not [4]. Used in conjunction with the information about a given application, these classifications are used to decide between data-driven and demand-driven implementation for a particular operator and to decide the buffering method to be used between operators.

Total and Partial Operators

A total operator produces a complete data object. A complete data object is one which does not contain bottom, i.e., it has no holes or gaps, and it cannot be further completed [7]. A partial operator is one which may produce an incomplete data object when applied to complete operand values.

Predictably and Unpredictably-heedless Operators

A predictably-heedless operator is one in which the subset of inputs required to compute each result can be predicted syntactically; i.e., the subset can be determined by static analysis. While building a specific system it is necessary to know beforehand the exact path taken by data during the computation, and hence predictably-heedless operators can be easily implemented using any of the models of computation.

An unpredictably-heedless operator is one in which the subset of input data values required can not be determined syntactically and depends on some of the data values; i.e., the subset is run time dependent. If the system does not have a reconvergent path for input data of a unpredictably-heedless operator, then again any of the models of computation can be used for implementation. Due to the dynamic nature of these operators, the model becomes quite complex in the case of reconvergent fanout. It may also be impossible to implement a perfect data-driven model which works well in all possible conditions since buffer sizes cannot be known a priori. If application-specific constraints can be applied, then it will be possible to use the data-driven model. The demand-driven model is suitable for these type of operators, but memory management then becomes a critical issue and some form of garbage collection strategy will be required to determine what data can be flushed.

Implications for Hardware Implementation

Some of the Lucid operators introduce an offset in the data path of a circuit with fanout. This offset can be either static or dynamic in nature. For example, consider the expression Y = X + *next* X. The corresponding dataflow graph is shown in Fig. 1. To compute Y(t) we require the values of X(t) and X(t+1). Thus, the next operator in this example introduces an offset of one in the fanout path. This offset is static in nature as it can be determined at compile time by analyzing the given program. To counter the effect of this offset and ensure the correct operation, a one stage buffer has to be connected in the fanout path. However, if the fanout in the circuit is not reconvergent, then it can be implemented without the buffer. In this case the two outputs will have an offset of one.

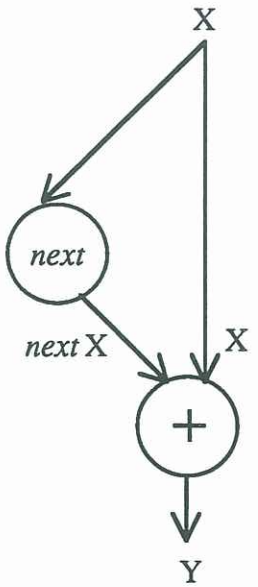

Figure 1. Dataflow graph for the expression Y = X + *next* X.

The operators that introduce a dynamic offset are most difficult to implement as the size of the offset can not be determined at the compile time. The size of the offset depends on some of the input values, and can only be determined at run time.

Thus, to implement these types of operators some dynamic decision making capabilities are required in order to select the correct offset. For example, consider the Lucid program shown in Fig. 2(a).

Fig. 2(b) displays the corresponding dataflow graph. This program contains the *upon* operator with a reconvergent fanout. To evaluate Z(t) at $t = t_1$ we need the values of $X(t_1)$ and $Y(t_1)$. The input values required to compute $Y(t_1)$ are $P(t_1)$ and the X(t') where t' depends upon the history of P. If the values of P have been false for n times before (excluding the value at t = 0), then we will need the value of $X(t_1 - n)$ in order to evaluate $Y(t_1)$. Thus, the two fanout paths of input data X have an offset of n in the data path of the *upon* operator. This offset is dynamic and depends on the values of P.

```
Z = X + Y;
where
    Y = X upon P;
end
```

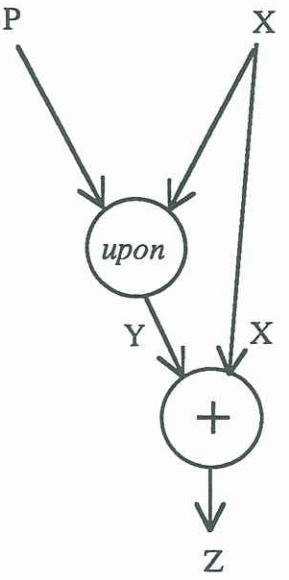

Figure 2. (a) A program containing *upon* operator with reconvergent fanout.

(b) corresponding dataflow graph.

This operator can be implemented using data-driven methodology by providing a buffer embedded in the *upon* operator, which can be reconfigured so as to choose the correct offset. Furthermore, the size of the buffer depends on the total number of times the value of P may be false. Thus, this operator cannot be implemented using a data-driven method without applying any system constraints regarding the total number of times the values of P can be false, otherwise it requires a re-configurable buffer of infinite size. In demand-driven methodology too, some constraints have to be applied in order to handle this operator with a finite storage. The demand-driven methodology handles it more efficiently since at a particular time only single fanout path requests for data and the data is used only by that operator, not by all the operators. A number of operators using a datum can request the data at different offsets. Thus, the demand-driven methodology does not require any extra hardware or overhead to make sure that correct offset is maintained, but does increase the storage requirements considerably.

Total and predictably-heedless operators introduce only a static offset, if any, and hence can be implemented using either data-driven or demand-driven methodology without any problems.

Some of the total and unpredictably-heedless operators such as the *if-then-else* operator, do not introduce any offset and can be implemented using any of the methods. Some operators, such as *upon*, introduce a dynamic offset as explained above, and require a complex implementation methodology.

Partial and unpredictably-heedless operators such as *whenever*, *asa* operators are similar to the *upon* operator. They introduce a dynamic offset and require a similar methodology as that of *upon* operator. These operators are different from the *upon* operator in the sense that the offset introduced is in the fanout path. Thus, the buffer has to be provided in the fanout path. Furthermore, if these operator are not handled correctly, a deadlock condition may arise in data-driven implementations. The demand-driven methodology is more suitable for the reasons explained earlier, and also prevents a deadlock.

III. Asynchronous Design

To improve the throughput and hence the performance of the system, a piped dataflow has been implemented. The pipeline is asynchronous with each stage having handshaking signals for

synchronization. Although a clocked pipeline would reduce the overhead of handshaking signals and hardware complexity, the asynchronous approach was selected for the following reasons:

1. In a clocked pipeline, clock skew becomes a critical issue. To reduce the clock skew to an acceptable value, a large portion of chip's power budget must be dedicated to driving the clock. The clock skew and the power dissipation in driving the clock become more important as the clock frequency increases. Also in synchronous circuits a large power is dissipated during each clock transition. Asynchronous circuits, in contrast, dissipate power only when they are active and only in the active regions. Thus, asynchronous designs can achieve near-zero standby power when quiescent [8]-[10].

2. Asynchronous designs do not use a global clock, simplifying global chip routing, and potentially reduce the space used for busing.

3. In a clocked pipeline scheme, it is necessary to perform a prior analysis of every stage in order to find the maximum allowed clock speed which can accommodate the operation of each stage in the worst case. If it is possible to perform a static analysis, then it is trivial for a synthesis tool to remove the extra hardware for handshaking and replace it with a clocked system.

4. A rare worst case delay in one stage may be much slower than the more common general case. Accommodating the worst case in synchronous design with a slower clock cycle may significantly reduce the overall performance. An asynchronous pipeline with handshaking signals can have different speeds in case the delays in the stages are different for different cases. Thus, asynchronous designs can provide high-speed operation, depending only on causal relation of signal transitions with an average delay instead of a worst-case delay [10].

5. Asynchronous designs have a better timing fault tolerance, because of insensitivity to delay variance in layout design, fabrication process and operating environment [10].

6. Asynchronous design can provide design flexibility and cost reduction, with higher level logic design separated from lower level timing design [10].

Synchronizing signals offsets the gain to some extent and requires extra hardware. A clocked pipeline can be created from our design by simply replacing the synchronizing hardware with a clock input. This can be performed by a synthesis tool in an optimization step. Asynchronous design also has the advantage of being a more generalized design.

Two standard cell libraries have been designed, one using the data-driven model and the other using the demand-driven model, for each operator. Our main aim throughout this design process has been to build generalized models which may later be simplified using optimization strategies. Next we discuss the general design and system level structure, followed by the design of representative individual operators for each model of computation.

Throughout we use a consistent model of computation that ensures that data-driven and demand-driven subsystems can be interfaced together. This decision generally increases the complexity of the design, but has the advantage that it allows the designer to build a hybrid structure without modifying the models. Once a general model is available, the synthesis tools used to implement the machine for the given Lucid description can perform this optimization. This would not be possible if a simpler, but more specific model had been designed.

IV. Data Driven Models

System Level Considerations for Data-driven Models

Some of the operators introduce an offset in the data path of the input data if the input has a fanout. In the case of non-reconvergent fanout, the system will perform correctly, but the outputs of the two paths will have an offset. Reconvergent fanout causes erroneous operation as incorrect data values may combine. In order to ensure correct operation of the system, buffering has to be introduced in the data path. The size of the buffer has to be determined before implementing the system.

The partial and unpredictably-heedless operators, and some of the total and unpredictably-heedless operators in a system, with reconvergent fanout, may introduce a dynamic offset in the data path. Theoretically, it is not possible to implement these operators with finite storage. We have assumed some system constraints

in order to implement these operators. The data-driven model has been designed on the basis that the maximum value of the dynamic offset is known before designing the operator. This maximum offset value can be determined statistically for a particular application by doing some sample simulations.

An asynchronous reset has been provided for all the sequential elements in the design. Providing the reset signal greatly simplifies the testing procedure as the system can be brought to a known initial state.

Bus Structure for Data-driven Models

The bus structure is designed so that it would be able to support other methods of computation and should provide the capability to interface the models using different methods of computation. The bus consists of a *n-bit data_bus*, *m-bit tag_bus* and two handshaking signals, namely *data_req and data_ack*. The handshaking signals are used for synchronizing the pipeline. Fig. 3 displays the timing diagram for a typical bus operation. The *tag_bus* provides the capability to interface these models with tagged data-driven and tagged demand-driven models of computation.

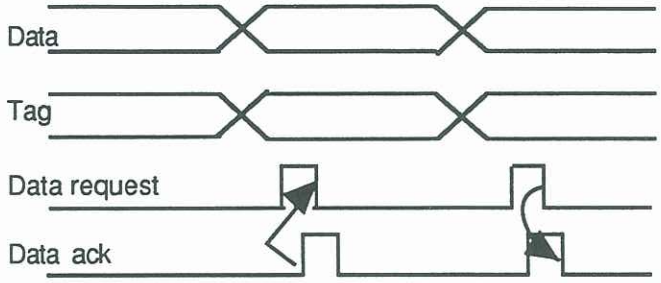

Figure 3. Timing diagram for a typical bus operation.

General Structure of a Data-driven Operator

Each operator is divided into two modules namely, an input stage and the execution unit with an output stage. The outputs of each operator are latched. The input data and tag are latched (by the previous stage or by the driving unit) on the *input_data* bus and the *input_tag* bus. The *data_in_req* is then issued to indicate that the data is available. The input unit detects the availability of data at the positive edge of *data_in_req* and stores this information if the execution unit is busy. The execution unit after completing the previous operation, checks for the availability of data. If data is available, it starts evaluating the particular operation. Once the computation has been performed and the last output cycle is finished, it latches the result on the *output_data_bus* and *output_tag_bus*. It then issues a *data_out_req* signal to indicate that the result is available and waits for the acknowledge signal, *data_out_ack* from the next stage. At the same time a signal is issued to the input unit to indicate that the current operation is done and the execution unit is ready to perform the next operation. The input unit issues an acknowledge signal to the previous stages, and then waits for the availability of the next data.

Design of Individual Data-driven Operators

The general structure for a data-driven model has been described in the previous section. A brief description of the design of some of the operators is presented in this section. The operators that are implemented in a similar methodology with a different functionality are indicated. Only operator-specific design considerations are discussed in this section, and the input unit and output stages are not discussed as they have been described in the previous section.

Next will be equal to the value of X at the next time. Thus, the value of Y can be given as

$$Y(t) = X(t+1) \quad \text{where t is time.}$$

The design can be explained as, the first input data request is ignored and the subsequent values of input data is latched at the output. The subsequent tag value is decremented and latched at the output. The execution unit consist of a *set* register and a decrementer. On the availability of first data, if the *set* bit is zero, then it is set to one and only the input handshaking signals are issued. If the *set* bit is one, tag value is decremented, and the input data and tag are latched at the output. The input and output handshaking signals are then issued. Fig. 4 displays the block schematic for the data-driven implementation of *next* operator.

If the input of this operator has a reconvergent fanout, then it produces an offset of one in the data path of the fanout. In order to ensure correct operation, a one stage buffer has to be connected in the fanout path for each *next* operator.

The design methodology, of *first, if-then-else, current* operators, is almost similar to that of *next* operator, but for the differences required to implement the functionality of each operator.

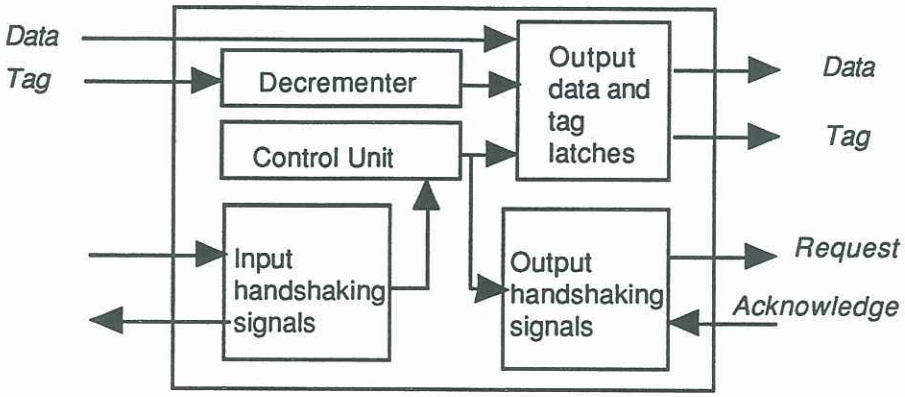

Figure 4. Schematic for the data-driven implementation of *next* operator.

Fby (read as *followed by*) operator is total and predictably-heedless. Consider the expression Z = X *fby* Y. The first value of Z will be the first value of X and subsequent value of Z will be the previous values of Y. Thus, the values of Z can be defined as

$$Z(0) = X(0) \text{ and}$$
$$Z(t) = Y(t-1) \text{ for } t > 0 \text{ where t is time.}$$

The first output of the *fby* operator is the first value of the X input. The subsequent output values are the previous values of Y input. It consists of a 1-bit *set* register, an incrementer and a control unit. When the X input data is available, the *set* bit is checked. If it is zero, then X input data and tag are latched at the output, and *set* bit is set to one. The input handshaking signals to X input and output are then issued. On subsequent availability of X data only the handshaking signals to X input stage are issued. When a data is available at the Y input, the *set* bit is checked. If the *set* bit is zero, it waits for it to become one. When the *set* bit becomes one, the tag value of Y input is decremented, and the Y input data and tag are latched at the output. The input handshaking signals to Y input and the output are then issued. The *fby* operator creates an offset of one in the fanout path of Y data, and requires a single stage buffer in the fanout path to ensure correct operation.

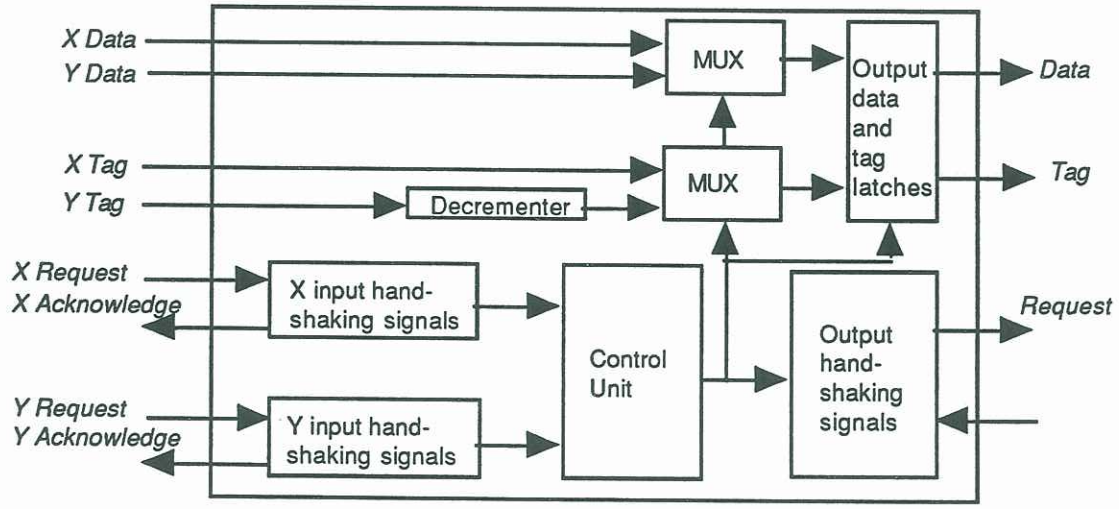

Figure 5. Schematic for the data-driven implementation of *fby* operator.

As it is evident from the above description, this operator is activated or fired whenever any one of the input data is available. The reason for using this strategy is that for evaluating the output at time 0, only the value of X(0) is required. For subsequent outputs at time t, only the value of Y(t-1) is required. Consider the expression N = 0 *fby* N + 1. The value of Y(0) input is N(0) + 1 and depends on N(0). If we activate the operator only when both the inputs are available, then to evaluate the output N(0), the *fby* operator must wait for both the inputs, 1 and Y(0), to be available. To evaluate the value of Y(0), the add operator must wait for N(0) to be available. Thus, a deadlock condition will occur which can not be resolved. This problem is solved by activating the operator whenever any one of the input data is available.

Whenever is partial and unpredictably-heedless. Consider the expression Y = X *whenever* P. The value of Y will be the value of X whenever P is true. The whenever operator can best be thought of as a selective filter that only passes certain data objects. The value of Y can be given as

$$Y(t') = X(t) \text{ whenever } P(t) \text{ is true, where t and t' represents time.}$$

This operator basically filters an input sequence and produces a collapsed output sequence. The subset of values of X required to evaluate Y cannot be determined syntactically as it depends on the values of P. Also, it may not be possible to complete the evaluation even if the input data object is complete because P may never be true. Thus, *whenever* operator is partial and unpredictably-heedless.

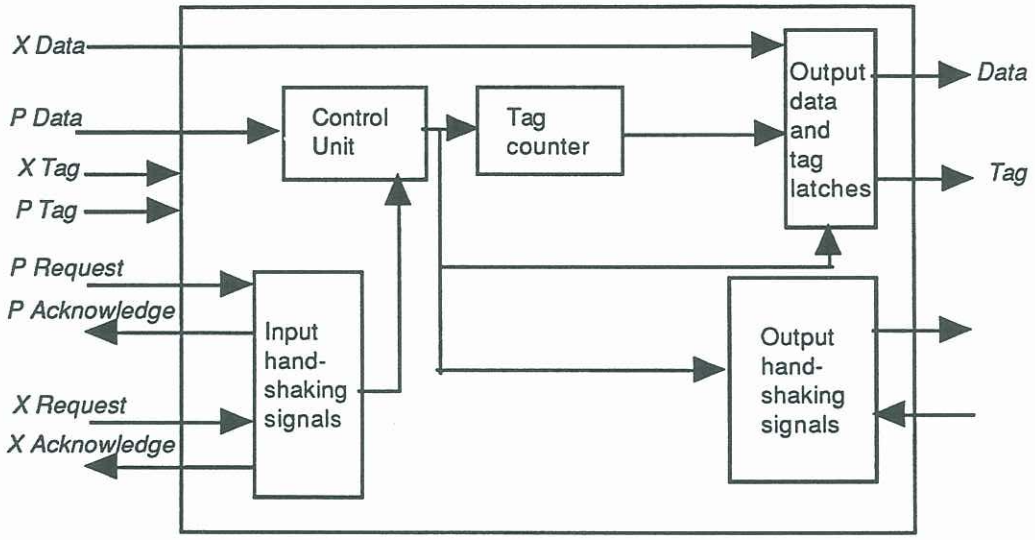

Figure 6. Schematic for the data-driven implementation of *whenever* operator.

This operator introduces a dynamic offset in the data path of the fanout, and will produce an output only for the input values for which the P input is true. It consists of a tag counter, an offset counter and a control unit. The tag counter stores the tag value of the previous output. The offset counter records the size of the offset introduced by the fanout operator. When all the input data is available, the value of P is checked. If P is true, then the input data and value of the tag counter are latched at the output and the tag counter is incremented. The value of the offset counter is also available at an output, and can be used to select the proper data from the buffer in the fanout path of the input variable. The buffer is a 16-word dual port cache with one read and one write port. The read port is random access while the write port operates in first-in first-out fashion. The offset count is checked on each computation and if it exceeds the maximum size of the buffer, an error signal is generated. Fig. 6 displays the block schematic for the data-driven implementation of this operator.

Other operators The *asa* operator is modeled in a fashion similar to *whenever* . In the case of the *asa* operator only the first data value is computed. The offset depends upon this computation and remains constant throughout. The *upon* operator is implemented using a similar design methodology except for a few deviations, which are required to implement the functionality of the operator. The 16-stage buffer is embedded in the model of the *upon* operator as the offset produced by the *upon* operator is in the data path of the *upon* operator.

A number of arithmetic operators have also been implemented for the purpose of verifying the design for a few sample programs. In this chapter the design of specific Lucid operators will be described. The design of arithmetic operators follows a similar methodology and are trivial to implement.

V. Demand Driven Models

System Level Considerations for Demand-driven Models

Demand-driven computations are suitable for computations which are not used very frequently. They have the advantage that only the useful computations are performed. In certain cases, the demand-driven model is more suitable for implementing partial and unpredictably-heedless operators. The disadvantage of demand-driven computations is the overhead in demand generation and propagation mechanism. The computation starts only when a result is needed and generally these computations are not pipelined. Thus, the performance of demand-driven computations is much lower (high latency and less throughput) than the data-driven computations.

The data values are supplied to the different operators upon request. This implies that a data value has to be stored until it is demanded. A dual port cache has been provided for the input data at the root level, i.e., only the primary input values are stored and any intermediate value will be computed only when an output is required. Each data object is provided with only one cache having one read port. Thus, if there is a fanout for that data object, a bus arbitration scheme is needed. Our bus arbitration is accomplished by daisy chaining. Each input of an operator is provided with two sets of *bus_request* and *bus_grant* signals. One set of these signals is used to obtain the bus from the next higher level and the other to provide a grant to the next lower level. At the top level, i.e., for the first operator accessing the cache, one set of these two signals can be shorted to grant use of the bus. The priority is decided by the level at which the operator is connected. If the data objects have a reconvergent fanout with at least one of the paths having a partial or non-predictable operator, then a set associative cache seems to be the best solution as it allows more than one set of data to be available. One set can have the data for the partial or incomplete object and other set can have the data for other operators with the required offset. This reduces the cache traffic, and to a great extent, the miss rate as well. The degree of associativity depends upon the system being implemented. For a given application, the cache parameters to optimize the performance can be determined statistically.

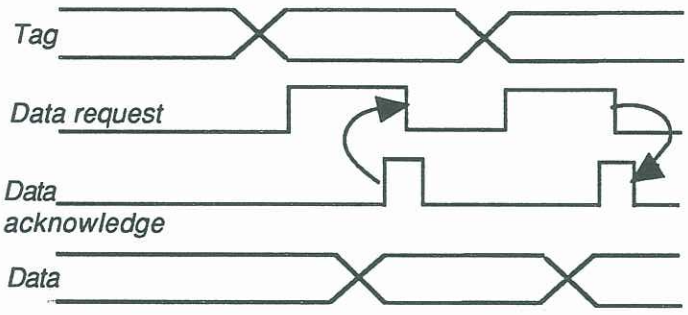

Figure 7. Timing diagram for a typical bus operation.

Bus Structure for Demand-driven Models

The bus structure is designed to be compatible with other models of computation and allows interfacing of these different models. The bus consists of a *n-bit data_bus*, *m-bit tag_bus* and two handshaking signals, *data_req and data_ack.*. This structure is similar to the bus structure of data-driven models except that in a demand-driven model a *data_req* signal is a request for the input data and a *data_ack* signal means that the requested data has been supplied. The *tag_bus* is provided in order to support tagged data-driven model of computation and to provide the capability for interfacing these models with tagged data-driven model of computation. The cache for a data object having a fanout will be accessed by more than one operator. A daisy chain architecture has been implemented for bus arbitration. Fig. 7 shows the timing diagram for a typical bus operation.

General Structure of a Demand-driven Operator

In demand-driven computation an operator is activated when a demand for the output is issued. Upon activation, the operator issues its own demands for the inputs needed to compute the result. Once the inputs are available, it performs the desired computation and passes the result to the output. A demand for an output results in generation and propagation of demands for inputs until finally the demand propagates to the root

level and data is requested from the cache. Once the data is supplied, computation propagates towards the output. The operator connected to the cache issues a request for the bus and then waits until the bus is granted. Once the bus is available, the operator issues a request for the data from the cache. Bus arbitration is provided locally by each operator connected in a daisy chain fashion as explained earlier.

Design of Individual Demand-driven Operators

The design methodology for demand-driven models is the same for all operators. The difference between the different models is for implementing the desired functionality. The design of a few operators will be described below.

Next is activated when an output is demanded. The tag value at the output is incremented and passed to the input. A demand for the input is issued and then the operator waits until the input data is available. When data is available at the input, a acknowledge signal is issued by the driving stage. On receipt of the input acknowledge signal, the input data is passed to the output and an output acknowledge signal is issued. Fig. 8 displays the block schematic for the demand-driven implementation of *next* operator. The *first* and *current* operators are implemented using a similar design methodology.

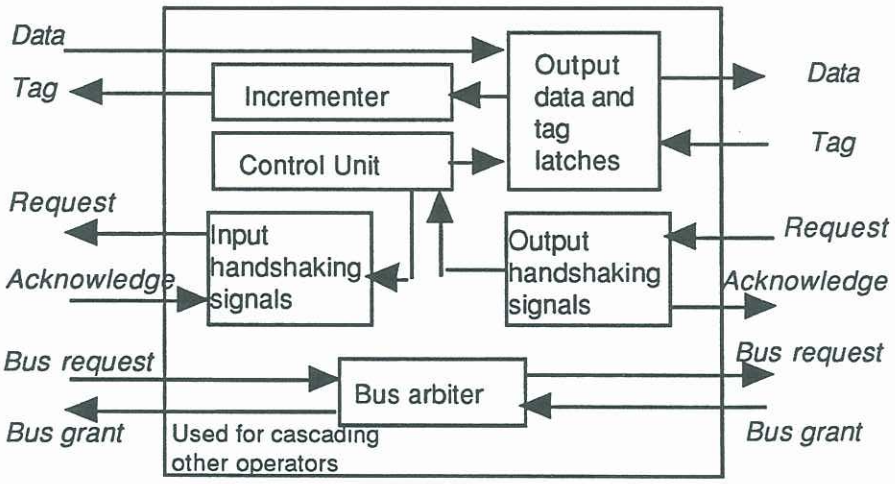

Figure 8. Schematic for the demand-driven implementation of the *next* operator.

Fby is activated when an output is demanded. If the output is demanded for tag value 0, a demand for X input at time 0 is issued. Once X(0) is available it is passed to the output and an output acknowledge signal is issued. If the output is demanded for tag t ($t > 0$) then a demand for Y(t-1) is generated. When Y(t-1) is available, this value is passed to the output and an output acknowledge signal is issued. Fig. 9 displays the block schematic for the demand-driven implementation. The *if-then-else* operator is modeled using a similar design methodology.

Whenever is a partial and unpredictably-heedless operator and one of the most difficult operator to model. The model consist of two m-bit registers. One of them is used for storing the last tag value of P which has been checked and the other one for storing the last output tag value. The operator is activated whenever there is a demand for output. Upon activation, it issues a demand for the P data from the last tag value of input, which had been checked. The counter, used for storing the tag value of the last input data which had been checked, is incremented. If the value of P is true, then the output tag counter is incremented. This tag value is then compared with the tag value of the requested output. If they are not equal, this process is repeated until the two values are equal. Once these two values becomes equal, the X input data is requested for the tag value for which the last P value was true. This data is then passed on to the output with an appropriate acknowledge signal. Fig. 10 displays the block schematic for the demand-driven implementation. The *asa* and *upon* operators are implemented using a similar design methodology.

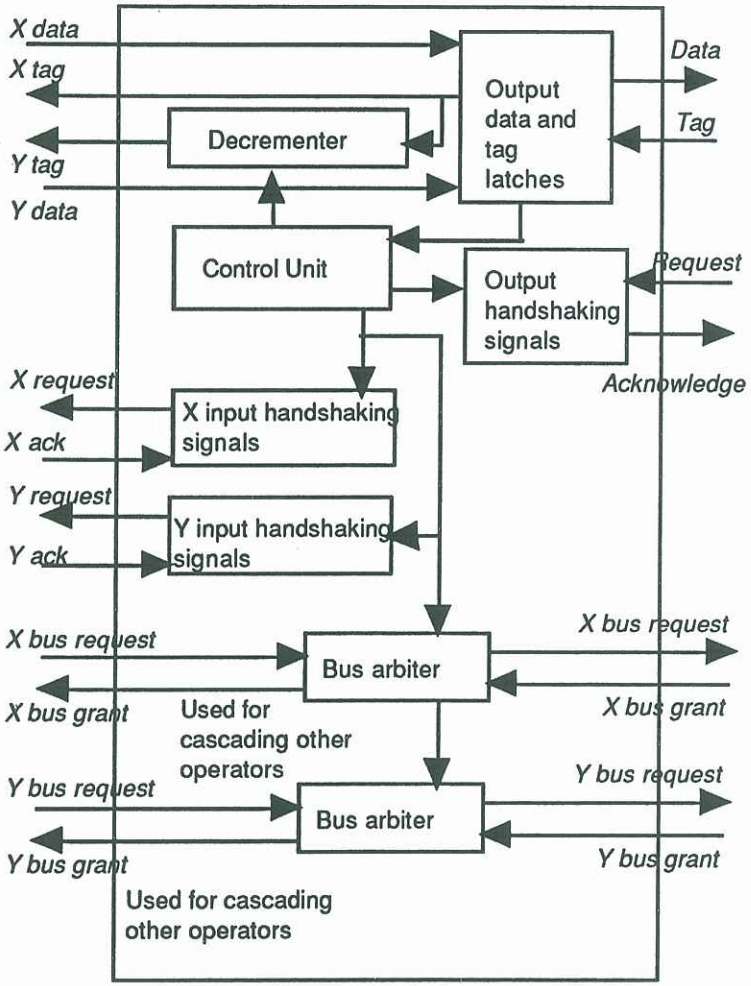

Figure 9. Schematic for the demand-driven implementation of *fby* operator.

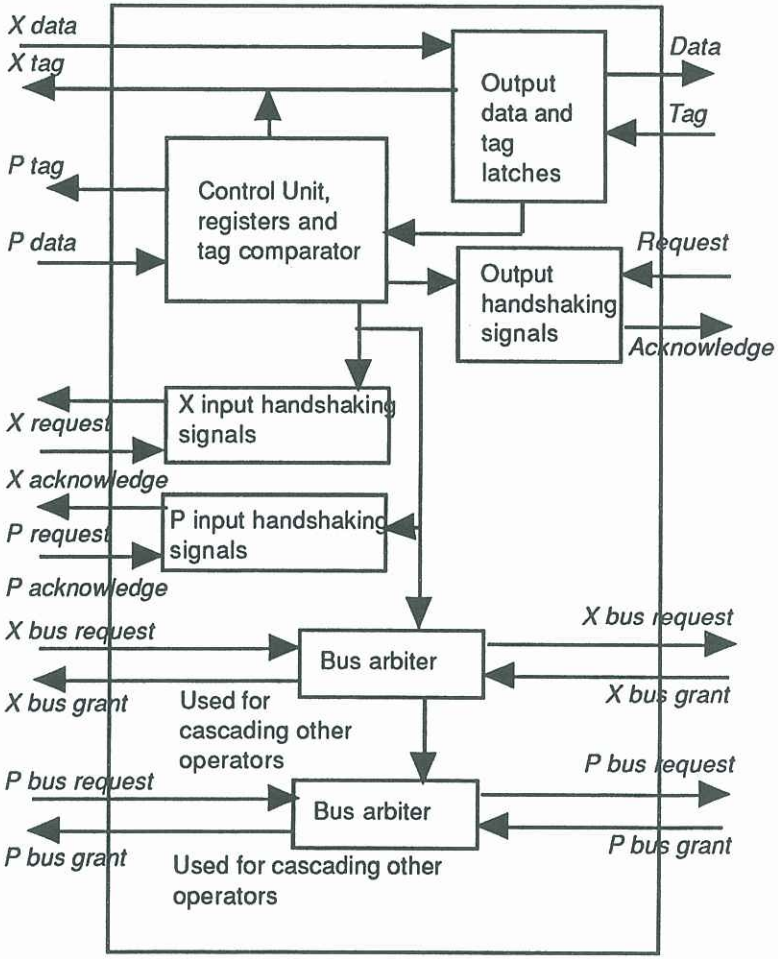

Figure 10. Schematic for the demand-driven implementation of *whenever* operator.

VI. Conclusion

Limitations

In data-driven models, the operators which introduce an dynamic offset in the data path have been designed assuming that the maximum allowed offset is known. However, with a finite buffer size, only a finite number of correct computations can be performed. Once the offset is exceeded, there should be some mechanism for supplying the correct data.

In demand-driven models, a cache controller with the ability to supply the requested data on a cache miss is required. This implies either a secondary data storage or a way to recalculate the requested data. With a secondary storage system, garbage collection techniques have to be employed for determining the data values which can be discarded. Fig. 11 displays the block schematic for such a scheme. All the caches in this scheme use the same main memory, cache controller and garbage collector. In very large circuits it may be advantageous to use more than one memory management system. Also, the main memory and the secondary storage system together can be used to provide a multilevel memory management scheme. The cache is provided with only one read port. Thus, with a large fanout, it can become a bottleneck. The number of ports or the number of caches, for an optimum performance, can be determined depending upon the particular application.

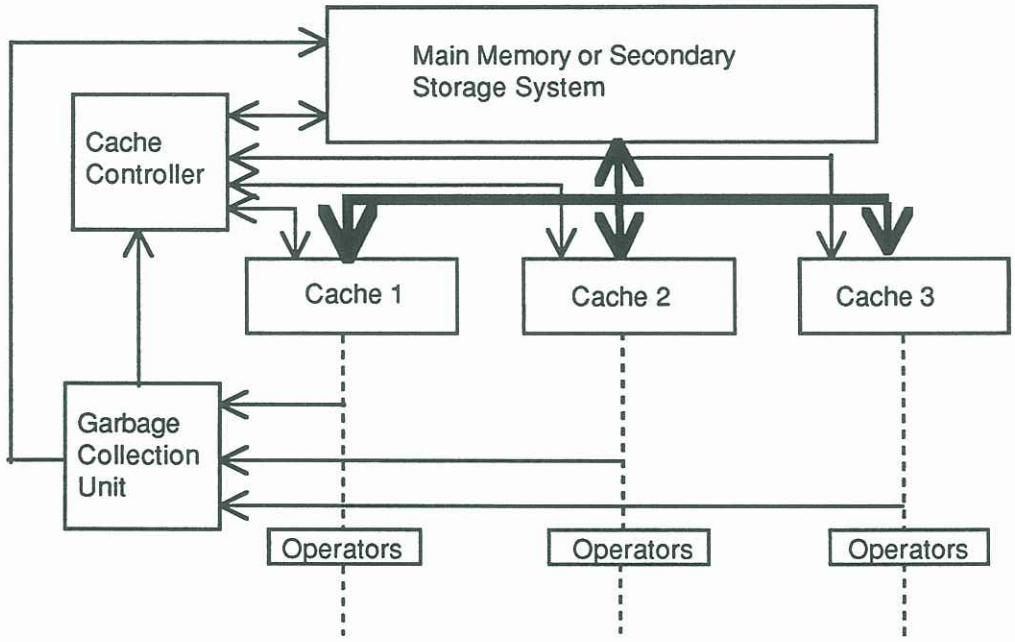

Figure 11. Block diagram of the memory management scheme for demand-driven implementations.

Testing

We have tried to both exhaustively test individual cells for each operation, and once each model was tested exhaustively, to implement and verify some sample programs that represent critical cases and test the interface design by combining operators of the different classes, and combining data-driven and demand-driven implementations. Some programs were also created by randomly connecting these operators. The results of all these tests are as per our expectations and are reported in detail in [11].

The initial goal was to model one operator from each of the different classes of Lucid operators for the data-driven and demand-driven model. We not only achieved this goal, but also implemented almost all the operators of Lucid and designed interfaces for hybrid models. A few hybrid structures were implemented and the results verify the functionality and interfacing capabilities of the cells.

The results of testing verified the functional correctness of the designs. Full performance evaluation can be done only after a system using this methodology is built and its performance is compared with an equivalent system based on conventional control flow methodology. This methodology implements a system with the maximum parallelism that could be achieved, and does not require data to be moved to or from the memory. Thus, even though the performance is not evaluated, we expect the performance of a system using this methodology to be much higher than a conventional system.

Summary

The development of a new tool or a language is an ongoing process. The prototype has to go through several cycles of modification before it is widely accepted. We have been successful in introducing a new idea for developing a synthesis language suitable for implementing parallel architecture systems. The standard cell libraries designed in this work present a strong foundation for realizing the ultimate goal of developing Lucid as a synthesis language for hardware design.

Our research has been aimed at reducing the complexity of designing parallel architecture systems. Thus, not only do we introduce an automated design scheme for a parallel architecture system, but due to wide availability of hardware synthesis tools for VHDL, we do so in a form that reduces the tool development cost and time considerably. It is our hope that this contribution will spur further efforts toward creating a fully automated tool for designing a parallel machine.

While the use of Lucid as a hardware synthesis language seems to be promising, much more work will be required before it becomes an effective tool and can be widely accepted.

References

[1] D. L. Perry, *VHDL*. New York: McGraw-Hill, 1994.

[2] Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems*. New York: McGraw-Hill, 1993.

[3] I. P. Radivojevic and J. Herath, "Executing DSP applications in a fine-grained dataflow environment," *IEEE Transactions on Software Engineering*, vol. 17, pp. 1028-41, October 1991.

[4] W. W. Wadge and E. A. Ashcroft, *Lucid the Dataflow Programming Language*. U. K.: Academic Press, 1985.

[5] R. Jagannathan and E. A. Ashcroft, "Eazyflow: a hybrid model for parallel processing," in *Proc. International Conference on Parallel Processing*, pp. 9-15, August 1984.

[6] A. A. Faustini, S. G. Mathews and A. G. Yaghi, The pLucid Programmer's Manual, Technical Report TR-83-004, Computer Science Department, Arizona State University, October 1983.

[7] W. W. Wadge, "An extensional treatment of dataflow deadlock," in G. Kahn (editor), *Lecture Notes in Computer Science: Semantics of Concurrent Computation*, Springer-Verlag, pp. 285-299, 1979.

[8] K. V. Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij and A. Peeters, "Asynchronous circuits for low power: a DCC error corrector," *IEEE Design and Test of Computers*, vol. 11, pp. 22-32, Summer 1994.

[9] A. Marshall, B. Coates, P. Siegel, "Designing an asynchronous communications chip," *IEEE Design and Test of Computers*, vol. 11, pp. 8-21, Summer 1994.

[10] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako and A. Takamura, "TITAC: design of a quasi-delay-insensitive microprocessor," *IEEE Design and Test of Computers*, vol. 11, pp. 50-63, Summer 1994.

[11] A. Kejriwal, *Standard Cell Designs for Hardware Synthesis with LUCID Operators*, M.S. Thesis, Arizona State University, June 1994.