16

ROM-based Multi Thread Controller

H.A. Hilderink and J.A.G. Jess
Eindhoven University of Technology
Eindhoven University of Technology, Department of Electrical Engineering, Design Automation Section, P.O. Box 513, 5600 MB Eindhoven, The Netherlands. email: H.A.Hilderink@ele.tue.nl

Abstract

A new controller architecture is presented which can handle multiple threads efficiently without state explosions. The architecture is based upon a ROM, where the address decoders are replaced by latches on each row, which makes it possible activating more rows (states) at the same time. The extra logic for controlling the control flow is minimal.

Keywords

Controller Generation, Controller Architectures

1 INTRODUCTION

The current generation DSP chips is able to handle large sequences of arithmetic operations in an efficient way. Nowadays the specifications become more and more complex, resulting in small data introduction intervals (the time between the current and the new data to be offered to the system: *dii*) for delivering data, like for video applications. At medium throughput applications the *dii* is larger than the time needed for the calculation of a result based on data, but at high throughput applications the wanted dii becomes smaller. This results that serialization of the algorithm becomes too slow and that other techniques must be found, like:

concurrency where two (or more) independent tasks are executed at the same time on different resources;
pipelining where the process is executed before its previous execution has finished.

A controller is fully specified by the results of a High Level Synthesis (HLS) system. Starting from a behavioral description operations are placed in time (scheduling) and each operation is assigned to a module which can perform the operation (module binding). The controller is specified as a token flow graph (similar to that of [Eijn93]), where nodes correspond with states and edges with state transitions. Each state is associated with a control vector indicating the operations the modules must perform. A state is active when it contains a token, which will be passed to the next state(s) at the end of the clock cycle.

For the implementation of the controller, different architectures can be used. A first consideration is the partitioning of the controller. For each set of resources a sub-controller is defined, while all these sub-controllers are

G. Saucier et al. (eds.), *Logic and Architecture Synthesis*

mastered by a global controller. Each sub-controller can be implemented on the most appropriate architecture. The next consideration is the choice of the implementation. There are different forms like a standard cell implementation, a Programmable Logic Array (PLA) or a ROM-based implementation. But all these implementations appear to be inefficient when the control flow contains a large grain of concurrency.

At the CATHEDRAL-II system [Zege90], the controller is implemented on a ROM-based architecture, which is accompanied by a branch and status finite state machine for the calculation of new (conditional) branch addresses, and an incrementer. The control output signals are directly provided by the ROM. At [Rane93] the controller specification is first partitioned, based upon sub-routine callings. Then the most suitable architecture is chosen for the implementation, which is based upon extended finite state models. The same method is also followed at [Gerb92]. There the implementation is based upon a ROM-based architecture, where control logic is added for the providing of the addresses by special structures according special constructions at the control specification, like conditional statements, loops and sub-routine callings. A modular interconnection of finite state machines architecture is used at [Ku91], where it is possible to implement multi thread control flows on separate finite state machines. The resulting set of communicating finite state machines are implemented on standard cells.

For all these implementations, except the last, holds that the concurrent specification is implemented on a single threaded architecture, i.e. an architecture where only one state can be active at the same time. This limitation often results in state explosions and consequently large implementations.

This paper presents a new controller architecture which can handle concurrency and pipelining efficiently. The architecture is based upon a ROM-structure, where the conventional ROM structure is slightly modified for permitting the activation of more than one row at a time. First the impact of concurrency and loop-pipelining is treated, followed by the description of the architecture. As result some implementation-templates are shown, illustrating the efficiency of the structure.

2 CONTROL STRUCTURES

A controller specification consists of an ordered set of sequences. A sequence is a consecutive list of states, where the states are executed one after the other. State transitions within a sequence are performed synchronous with a clock signal and are unconditional. A sequence can be executed independently of other sequences by using its own resources. Conditional execution of sequences is achieved by conditional state transitions between sequences.

Concurrency of sequences Concurrency is defined as the execution of two (or more) sequences independent of each other, where each sequence controls an own set of resources. Take for example the control flow of figure 1, where loops are started along sequence S. State T_n can only execute when all the loops have finished their executions. There are two forms of synchronization: first is when the number of iterations of each sequence is known beforehand, where synchronization is achieved by waiting for the slowest sequence (*implicit synchronization*); and second is when the number of iterations of one (or more) sequences is data dependent, requiring special actions which keep track whether all sequences are finished or not (*explicit synchronization*).

Pipelining of sequences Pipelining of sequences is defined when the sequence is restarted before its previous execution has been finished. This is only possible when the resources needed for the new invocation are released before the start of a new invocation, and that at every state of the sequence this criterion holds. Moreover it is needed that data, needed at the new invocation, has become available from the current invocation (or before). The data introduction interval (*dii*) is defined as the maximum time needed for the release of a particular resource controlled by states of the sequence.

If each state controls only one set of resources which is not controlled at other states, the dii is one cycle, like in figure 2(a). Here a loop with 6 states is pipelined by restarting the loop each cycle. Looking at the control of the

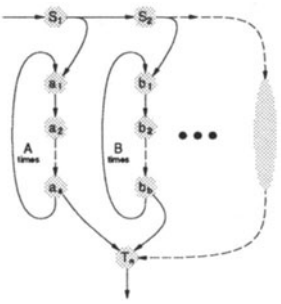

Figure 1 Set of concurrent sequences, where sequence `S` starts other sequences (loops) `a, b, ...`

loop, we see that its execution can be divided into three parts: first the prologue sequence, where the pipeline is filled, second the steady state sequence, and last the epilogue sequence, where the pipeline is being emptied. For the implementation on a single thread control flow this results in extra states for both the prologue and epilogue, but the body of the steady state sequence contains less states as the body of the original loop.

3 MULTI THREAD ROM BASED CONTROLLER ARCHITECTURE

Conventional controller models, based on a single ROM architecture, are characterized by the implementation of a single thread control flow, which is caused by the limitations of the conventional ROM. Although the ROM is efficient for large controller implementations [Gerb92], its structure is not suitable for a multi thread controller. Looking more in detail at the ROM structure, it is evident that the address decoders on its inputs (which selects one row at a time, according the provided address) forms the bottleneck. When these address decoders are replaced by latches, each row can be controlled separately (figure 2(b)).

The control signals (output signals of the controller), are generated by the the columns of the ROM. Each column (set of columns) controls one resource. During the synthesis of the behavioral description, it is guaranteed by the scheduler and allocator that a specific resource is controlled by only one state at a time. This results that for the controller implementation each column can be activated only by one row at each cycle.

A row is activated when a signal ("1") is clocked into the corresponding latch. At the next cycle, the signal can be shifted into the next latch, activating the next row. In this way, a simple sequence of states can be executed.

At a conditional statement, the signal must be passed to one of the possible latches, according the test signal (from the datapath). Only that latch is selected for which the test condition evaluates true. Thus the evaluated test signal for each alternative must be "anded" with the token for each conditional latch (figure 3). After termination of the conditional sequence, the token must be passed to the next latch. This results that both outputs of the conditional sequences (*then* and *else* body) must be "orred" for the next latch. Thus only two *and* gates, one *or* gate (and one invertor) are needed for the implementation of *if-then-else* control flows.

For a loop implementation the same number of gates are required.

Concurrency is achieved when the token is fed not to only one latch, but to more latches, each starting an independent sequence. The next state following the concurrent sequences can only be executed when all concurrent sequences are finished (figure 4b). Synchronization becomes *implicit* by taking the token from the slowest chain,

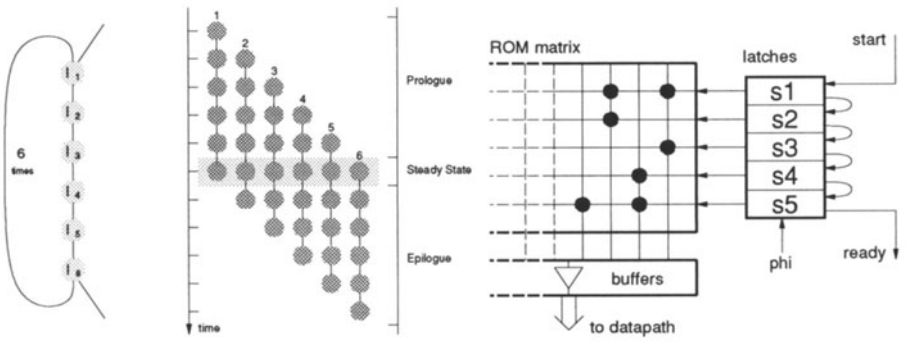

Figure 2 a: Example of a pipelined loop with 6 states, b: Proto Multi Thread ROM based controller

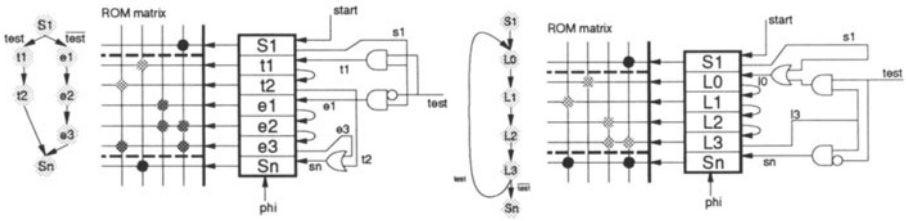

Figure 3 a: If-then-else implementation, b: Loop implementation

ignoring the other tokens (figure 4a). Synchronization becomes *explicit* when a state machine is needed which keeps track of sequences which are finished and sequences which are not. Only when all sequences are finished, a token is passed to the next latch. Take for example figure 4c, where explicit synchronization is implemented. If the concurrent sequences are started, the FSM is set to state q_0. When one of the concurrent sequences ends, the status of the FSM changes (to state q_a if sequence `b` is ended or to state q_b if sequence `a` is ended). If both (all) sequences end at the same time, the final state q_n is reached, which issues a signal to the next state s_n.

Pipelining

Pipelining of a sequence is started when a token is inserted while the previous token still has not leaved the sequence yet. This results that more than one states are active at the same time, resulting in two (or more) active rows within

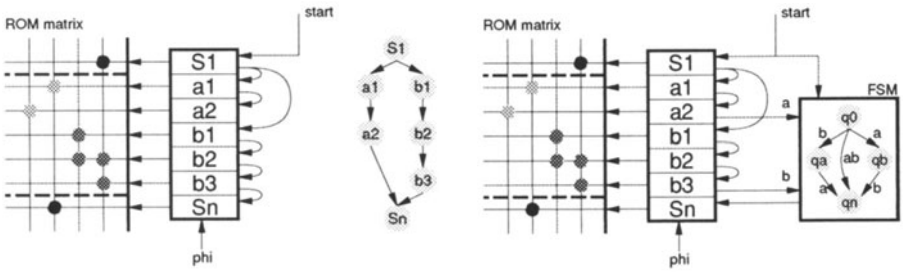

Figure 4 a: Implementation of concurrent sequences with implicit synchronization, b: A concurrent sequence construction, c: Implementation of concurrent sequences with explicit synchronization

the ROM. Take for example the implementation of a simple sequence of figure 2: a new token can be inserted at least three cycles after the previous token was inserted (the *dii* of the sequence is therefore 3 cycles). This does not mean that the next token must be inserted directly if possible, but it might also be delayed some cycles. This results in extra freedom for defining the pipelines while the implementation remains unaltered.

Also pipelining can be used within special constructs (i.e. *if-then-else* constructs, *loops* or *concurrency*), but then special attention must be paid to for maintaining the correct sequence of the tokens. Consider for example the *if-then-else* construction of figure 3 where the sequences of the alternatives do not have the same number of states. When a first token enters the construction and goes the longest way (*else-branch*), while a second token enters the construction a cycle later and goes the shortest way (*then-branch*), it can occur that both tokens arrive at the end state at the same moment. This non-determinism can be prohibited by either implementing a kind of synchronization within the construction, or by defining the *dii* of the construction equal to the number of states of the longest sequence.

Partitioning

The most efficient way for partitioning a controller is to define sub-controllers which control only a limited set of resources (probably two or more resources), and that each resource is controlled by only one controller. This results that the outputs of the controllers never have to be combined to deliver the final control word for the resource. Problems arise when a concurrent set of sequences is followed by another concurrent set of sequences, where the resource partition each sequence controls can differ (figure 5). So is resource $\mathcal{B}$ controlled together with resource $\mathcal{C}$ by sequence b, but is the resource together with resource $\mathcal{A}$ controlled by sequence p. This makes efficient partitioning difficult, but not entirely impossible.

4 EVALUATION

This adaptation of the ROM structure and the use of it as a controller has its impact on the total size of the implementation of the controller, and also on the delay and power consumption. This section compares the differences between the Multi Thread ROM controller implementation and a conventional ROM based controller implementation.

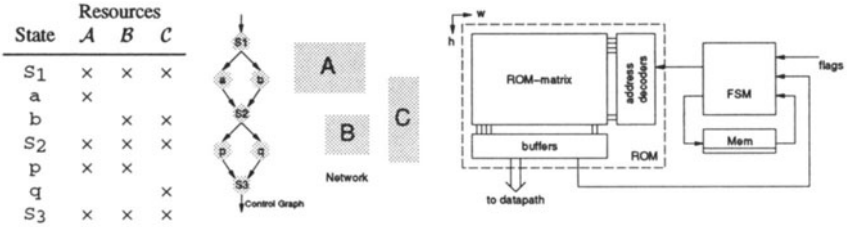

Figure 5 a: Resource occupation by different concurrent statements, b: Area occupation of a conventional ROM

Area A conventional ROM based controller consists of the following parts (figure 5):

- The ROM:
  Matrix: area depends on the number of columns and rows;
  Buffers: area depends on the number of columns;
  Address Decoders: area depends on the number of rows and the number of address lines;
- Finite State Machine: area depends roughly on the number of address lines for which it has to determine a value for each state transition, and on the number of states.

The Finite State Machine provides the next address of the ROM which depends on the output of the ROM of the current address, combined with the status signals from the data path. The more complex the state transitions are, the larger the area will be.

Comparing this conventional architecture with a Multi Thread ROM based solution, the following differences are evident:

- The address decoders are replaced with latches: the size of a single latch is about 4 times larger than an address decoder for 8 bits (The sizes are compared within the Compass Design System where the area of an address decoder, part of its generic ROM structure, is compared with the area of a single transparent latch: this is nevertheless a rough estimation).
- The area for the additional gates is slightly less than compared with the area needed for the Finite State Machine at the conventional solution.
- A minor difference is the size of the matrix: this is smaller caused by a smaller number of states (rows) and the fact that no state information is obtained from the matrix, which reduces the number of columns.

The area needed for the latches and the additional gates must be weighted against the area needed for the address decoders and the Finite State Machine. Especially when a concurrent controller description results in an "exploded" single thread controller description, the Multi Thread solution can be smaller. Also when flexible pipelining is wanted, this solution can be smaller than conventional designs.

Delay The difference between both architectures lies within the size of the FSM, whose critical path is for the Multi Thread ROM solution considerably shorter. This improves the overall speed off the controller and probably does not require extra pipelines within the controller.

Power A CMOS gate consumes power only when its output changes its value: switching. At the conventional ROM implementation this happens mostly at the address decoders, where for each state transition a new address

must be decoded, and at the Finite State Machine where a new address must be determined. At the Multi Thread ROM structure power is consumed when a token is shifted into a latch (and when it is shifted out of the latch), while the rest of the latches remain idle, and at the small number of gates when the transition is a conditional one.

The power consumption for the output buffers of the ROM matrix remain the same for both implementations. The Multi Thread ROM solution consumes therefore less power.

5 EXAMPLES

For proving the efficiency of the structure, examples have to be provided, but these are still under construction. This means that no comparison can be made yet between this multi thread solution and other conventional controller implementations. Also the lack of large examples, especially those with a large grain of concurrency, makes a comparison difficult. The controller generator is written as an application on the NEAT system [Heij93], using C++. As backbone the Compass Design System will be used, whereat the Multi Thread ROM module must be generated.

6 CONCLUSION

Although the Multi Thread ROM based controller has not yet been implemented, its preliminary results are encouraging. The advantages of the structure are its flexibility and relatively small area and power consumption also when the controller specification becomes more complex. The mapping of the control graph onto the network architecture is a straight forward job and does not require much time, this in contrary with the conventional state assignment procedures.

Remaining work will be the mapping of the controller on hardware and proving its efficiency, but also a more detailed study about partitioning and communicating controllers must be performed.

REFERENCES

[Eijn93] J.T.J. VAN EIJNHOVEN, J. JESS, AND J.P. BRAGE. Behavioral Specifiation for Synthesis. In F. CATTHOOR AND L. SVENSSON, editors, *Application–Driven Architecture Synthesis*, chapter 2, pages 23–45. Kluwer Academic Publishers, 1993.

[Gerb92] L. GERBAUX AND G. SAUCIER. Automatic Synthesis of Large Moore Sequencers. *Integration, the VLSI journal*, 13:259–281, September 1992.

[Heij93] M.J.M. HEIJLIGERS, H.M.A.M. ARTS, J.T.J. VAN EIJNDHOVEN, H.A. HILDERINK, J.A.G. JESS, W.J.M. PHILIPSEN, AND A.H. TIMMER. The New Eindhoven Architectural synthesis Toolbox. In *Workshop on Circuits, Systems and Signal Processing*, pages 71–75, Houthalen (B), March 1993. ProRisc/IEEE Benelux.

[Ku91] D. KU AND G. DE MICHELI. Optimal Synthesis of Control Logic from Behavioral Specifications. *Integration, the VLSI journal*, 10:271–298, October 1991.

[Rane93] K. RANERUP, L. PHILIPSON, J. MADSEN, O. OLESEN, AND G. JANSSEN. Controller Synthesis and Verification. In *Application–Driven Architecture Synthesis*, chapter 10, pages 211–230. Kluwer Academic Publishers, 1993.

[Zege90] J. ZEGERS, P. SIX, F. RABAEY, AND H. DE MAN. CGE: Automatic Generation of Controllers in the CATHEDRAL-II Silicon Compiler. In *Proceedings of the European Conference on Design Automation*, pages 617–621, Glasgow, February 1990. IEEE Computer Society Press.