Scheduling of Outputs in Grammar-based Hardware Synthesis of Data Communication Protocols

Johnny Öberg, Anshul Kumar[1], Ahmed Hemani

Electronic Systems Design Laboratory,
Royal Institute of Technology (KTH),
ESDLab/KTH-Electrum, Electrum 229, S-164 40 Kista, Sweden
[1] Dept. of Computer Science & Engineering,
Indian Institute of Technology, New Delhi, India.
Email: {johnny, ahmed}@ele.kth.se, anshul@cse.iitd.ernet.in

Abstract

We present a grammar based specification method for hardware synthesis of data communication protocols in which the specification is independent of the port size. Instead, it is used during the synthesis process as a constraint. When the width of the output assignments exceed the chosen output port width, the assignments are split and scheduled over the available states. We present a solution to this problem and results of applying it to some relevant problems.

1. Introduction

System descriptions are composed of three elements: Data structuring, Computation and Communication. Languages like VHDL, C++ etc. have evolved to support computation intensive applications. Communication in such languages are weakly supported using data structuring elements to specify the *interface* and computation elements to specify the *implementation of protocol* for communication between systems.

ProGram (Protocol Grammar) is a grammar based specification language, inspired by YACC and LEX, intended to serve the need of communication intensive applications, where the functionality is dominated by the task of *recognising* messages in the input stream(s) and *producing* output stream(s) based on message contents and the sequence. The specification essentially specifies the vocabulary and the grammar for communication between systems. A crucial detail that is not part of specification is the width of the communication channel. Just as in a telephone conversation, the communicating parties do not worry about the bandwidth of the communication channel.

Computational aspects of the system description are modelled as the actions that need to be performed when a message has been recognised. Depending on the chosen width of input and output streams, the actions are spread over several cycles. This is the task of output scheduling that is the focus of this paper.

Section 2 relates our work to similar work done recently. Section 3 gives a brief introduction of ProGram. Section 4 illustrates the synthesis process using a small example and present the output scheduling algorithms. Section 5 shows the results of doing design space exploration and synthesising a small part of the Operation and Maintenance (OAM) functionality of the ATM-protocol together with initial experiments on the quality of the produced VHDL code. Finally, section 6 summarizes our contribution, draws some conclusions and discusses future research.

2. Related Research

Automatic generation of language recognizers from grammar specifications has been extensively used in the software area for a long time [7]. Synthesis of hardware from such specifications was recently reported by Seawright et. al. [1]-[4]. They describe a system, called Clairvoyant, which is used to synthesize some small to medium sized examples from Production-Based Specifications. The output of Clairvoyant is an FSM described in VHDL that is synthesizable by logic synthesis tools. This system is targeted for detailed specification of communication interfaces and other control-dominated circuits including Communication Protocols. In Clairvoyant a design entity with a single process and a well defined boundary and interface is specified. All inputs and outputs in the design entity are described at clock cycle level.

Our approach is similar to this to the extent that the input is a production based specification and the output is in RT-level VHDL. However, ProGram is targeted for specification and synthesis of large systems and differs from the above approach in being at a higher level of abstraction. The synthesizer allows fast exploration of the lower level design alternatives and frees the designer of clock cycle level details and exact input-output description. Furthermore, multiple processes can be specified in ProGram and reading and writing of the multiple input and output streams can take place independently. This is very useful for structuring large designs. In addition, the ProGram

descriptions are independent of the width of I/O. The I/O sizes are derived to satisfy the throughput constraints posed in the grammar description. This entails splitting and scheduling the logic for message recognition from input stream and assignment to output stream. This is the key extension to our previous work ([15]) in this field.

In Clairvoyant all actions are specified in VHDL-code, similar to the software approaches used by, for instance, YACC [6][7]. ProGram, however, uses actions specified as lists of assignments. This helps in analysing the specifications for design space exploration, as well as for design verification. ProGram is also inspired by YACC in terms of its notations, but has been designed with the aim of hardware synthesis. In spirit, it has some similarity with LOTOS [8] in the sense that like LOTOS, specifications in ProGram deal with sequences of allowed events rather than states and state transitions as is the case with Estelle, SDL [8] and Promela [5].

DALI [14], is a commercial system based on the Clairvoyant system. It uses a graphical interface for entering the production rules, called frames, in a hierarchical manner together with their actions. As with Clairvoyant, DALI supports actions in a “host” language (VHDL or Verilog). In addition, DALI has some inbuilt simple data manipulation and communication primitives.

3. Describing Hardware Protocols as a Grammar

Data communication protocols can be viewed as a language in which two systems communicate. ProGram is based on a BNF like notation to specify a) the vocabulary of the language and b) the grammar rules to compose a message in terms of the vocabulary. A third element specifies the action to be taken when a message has been recognised. Two additional elements specify the memory layout and interface.

A. Interface declaration section

In the interface section, see Figure 3.1., the external interfaces and internal signals (interfaces between processes) are declared. Input declaration specifies an input port and consists of the name, the width and the bitrate of the input. The bitrate is optional and if given, ProGram uses the information as a constraint during the synthesis process and will introduce pipelining if necessary. If the bitrate is not given, the synthesised circuit will not accept another message until the completion of the current one. An Internal specification specifies the name and width of an internal signal, that are used for communication between concurrent processes inside the protocol circuit. An Output specification similarly specifies the name and width of the output port. Note that, though the width of input, output and internal signals is specified, the grammar is completely independent of the bitwidth. By varying bitwidth, the user easily explores the design space. This will cause the internal and output assignments to be partitioned and scheduled over the input tokens constructed from the input port size constraint.

```
%input input_cell_1 [bit]8 rate 155 Mbps
%internal vci [bit]16
%internal address [bit]8
%output output_cell_11 [bit]53*8

%memory connection_memory [[bit]8] connection_status

%start Input_Handler(input_cell_1)
```

Figure 3.1. Interface declaration for the F4 OAM example.

Memories are specified in a slightly different way. The size of the memory is specified in terms of number of address lines followed by the name of the rule that specifies the memory field layout.

Start rules mark the top rule in the rule hierarchy. Since there can be multiple input streams, the start rules also specify the current input stream. The input stream is then inherited downwards in the hierarchy of rules until a redirection is found, as explained later.

B. Token definitions section

A Token is a pattern of bits read from an input stream or written to an output stream. Reading and writing tokens are the primary events. Unlike YACC & LEX, tokens are not specified separately nor is a separate recognizer built for them.

Symbolic names are given to tokens as shown in Figure 3.2. The Token SIXA42 is an example of multiplicity where the pattern 6A (hexformat) is repeated 42 times.

```
// GENERAL TOKENS
VCI_SEGMENT             0000 0000 0000 0011
VCI_CONNECTION          0000 0000 0000 0100
VCI_USER_1              0000 0000 0000 0001
VCI_HIGH_IS_ZERO              0000 0000 0000
VCI_IDLE                0000 0000 0000 0000

SIXA42      [0110 1010]42
```

Figure 3.2. Examples of F4 OAM specific tokens.

C. Memory Layout Section

Memories are interpreted as a list of records. The record is a collection of named fields. A field consists of a name together with its bit size. Fields inside a record can be accessed separately. The record to access is identified by the address to the memory.

D. Action Macro Section

Actions in the grammar specify assignment of values to signals. To ease the task of specification, actions that are frequently used can be specified as a macro which can be referenced later in the action part of the production rules. An example of action macros is shown in Figure 3.3.

```
q = $a next CPM16 ($trcc01+1)16;

next = if (inp=00) then 1001 else 0110 end if;
```

Figure 3.3. Example of an action value specification.

Expressions to compute values may be put directly in the assignments with a size specification. The expressions allow concatenation and conditionals in addition to the usual arithmetic and logic operations. The operands can be constants, signals, other action value symbols or bit patterns recognized by grammar symbols. Hierarchical descriptions are supported. The assignments can then simply refer to the name of the macro. A symbol prefixed with $ refers to the bit pattern recognized by that symbol. The scope of such a reference is limited to the symbols already recognized in the current alternative.

E. Grammar Rules

A grammar rule consists of a grammar symbol which serves as a rule identifier and a list of alternatives. Each alternative is a sequence of non-terminal symbols and/or terminals followed by actions enclosed in curly brackets as shown in Figure 3.4. A redirection of the input stream is done by passing the new signal stream as a parameter to the subtree of productions.

```
input_cell: gfc vpi VCI_SEGMENT
      { vci=VCI_SEGMENT; }
   pti clp hec
      { address = $vpi; } oam_segment_types
   | gfc vpi vci_user {vci=$vci_user;} pti clp hec
      { address = $vpi; } user_cell_body
      special_user_cell(connection_memory[address])
      { output_cell_11 = special_user_cell_action;
       priority=PRIORITY_1; };
```

Figure 3.4. Partial grammar specification for the OAM/ATM example.

In the signal assignment in Figure 3.4., the signal address is assigned the value of the rule production vpi. The symbol special_user_cell parses the full record obtained from the memory connection_memory at address address.

Symbols that are not hierarchical are called Terminals. A terminal can be any of the four following things: a token, a bit string, a special pattern or the keyword error. The keyword error denotes an error condition. There are two types of special patterns:

```
[bit]k
[others]k
```

where k is an integer. The first one specifies a string of k don't cares (either a 1 or a 0). The second one is the same as specifying an else clause i.e. all other combination of bits that has not previously been specified. This also matches any string of k bits. The others type of pattern or an error keyword can only appear as the last alternative in a grammar rule. In addition, it is also possible to negate a pattern, i.e. to specify that anything but this pattern matches the description. Any pattern or error condition not directly given, corresponds to an error state in the synthesized hardware.

4. Synthesis from the Protocol Grammar

The synthesis procedure is based on a series of transformations performed on the grammar specification, each transformation taking the specification closer to a possible implementation. To illustrate the synthesis process we will walk through the synthesis of a self synchronizing Manchester encoder, whose ProGram description is shown in Figure 4.1.

```
// ProGram Specification of a Manchester Encoder
%input inp bit
%output q bit
%start encode(inp) no_reset clk 20 MHz single_FSM
%% // No constants are specified
%% // No memories are specified
%% // No actions macros are specified
%%
encode: 00 { q = 01 ; }
   | 01 bit { q = 010 ; }
   | 10 bit { q = 100 ; }
   | 11 { q = 10 ; };
```

Figure 4.1. ProGram Specification of a self-synchronizing Manchester encoder

The manchester encoder has one input, named inp, and one output, named q, of the size bit. The encoder does not have a reset input since it is self synchronizing. It should be implemented as a single statemachine clocked at 20 MHz. The clock period does not affect the structure of the produced statemachine, but is instead used by the backend tool as a constraint for the subsequent logic synthesis. The encoder input must be oversampled with a factor of two in order to have enough transitions to output the manchester encoded output. If the sampled sequence is 00 or 11, the encoder is in sync and the corresponding manchester encoded output is assigned to q. If however the sampled sequence is 01 or 10, the encoder is out of sync and need to wait one additional bit in order to get in sync again. Therefore, one additional bit is added to these transition

```
CheckConsistency();
for i=1 to Number_of_Start_Rules loop
   start_symbol=new symbol(ENTER,stream(i));
   exit_symbol=build_grammar_DAG(start_rule(i),
         start_symbol);
   ReduceExits(start_symbol);
   WordAlignTerminals(start_symbol,stream_size(i));
   ReduceGrammar(start_symbol);
   WordAlignOutputs(start_symbol);
   ScheduleUpwards(start_symbol);
   ScheduleDownwards(start_symbol);
   ReduceTailStates(exit_symbol);
   MarkAllStates(start_symbol);
   Output_FSM(start_symbol);
end loop;
```

Figure 4.2. The synthesis procedure.

sequences. The output assignments during these phases could be used to output a correct value but instead we choose values that allows us to minimize the number of states in the final FSM.

In order to synthesize a state machine, we transform the grammar specification into a Directed Acyclic Graph (DAG), called the Grammar DAG, which is a series-parallel graph in which the parallel subgraphs correspond to the alternatives for a grammar symbol and the subgraphs in series correspond to the sequence of items within an alternative. All non-recursive references to non-terminal symbols are expanded to their subgraphs in the Grammar DAG. Thus, the nodes of the Grammar DAG correspond to the terminals and recursive references to non-terminals.

The input is parsed and the productions stored in tables and checked for consistency and conformance to the restrictions. Then the Grammar DAG is constructed for each start symbol and optimised using the procedure shown in Figure 4.2. Each start symbol will result in one parallel communicating VHDL process. The details of the first four algorithms were covered in [15] so they will only be covered briefly here. The main focus will be on the two output schedule algorithms.

First, the grammar is build using the build_grammar_DAG() algorithm. Each item of all the alternatives in a grammar production rule is expanded and linked together with a dummy exit state. The dummy exit states are reduced out of the Graph by the ReduceExits() algorithm. One Exit node remains after reduction: the exit state of the uppermost level of the grammar that collects all expanded branch alternatives for the DAG of that start symbol.

When the grammar DAG has been constructed and all dummy Exits have been deleted, the resulting grammar DAG is word aligned onto the input stream of its start symbol. Terminals of smaller size than the width of the input stream are grouped together and terminals that are wider than the width of the input stream are split into smaller pieces. Actions are kept with the last terminal of the split sequence. The result of applying this algorithm to the Manchester Encoder example is shown in Figure 4.3. The input transition sequences are split into portions that have the same size of the input inp.

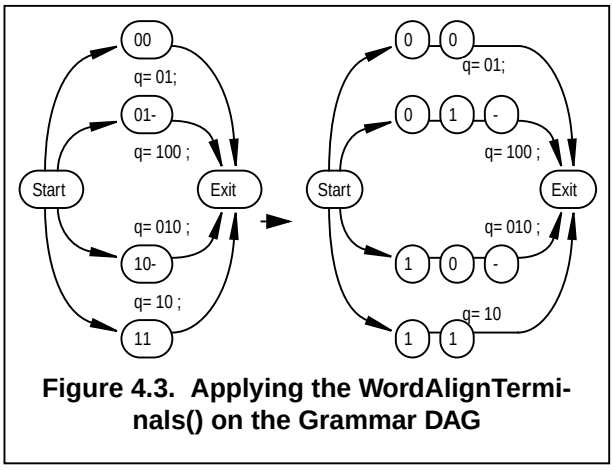

Figure 4.3. Applying the WordAlignTerminals() on the Grammar DAG

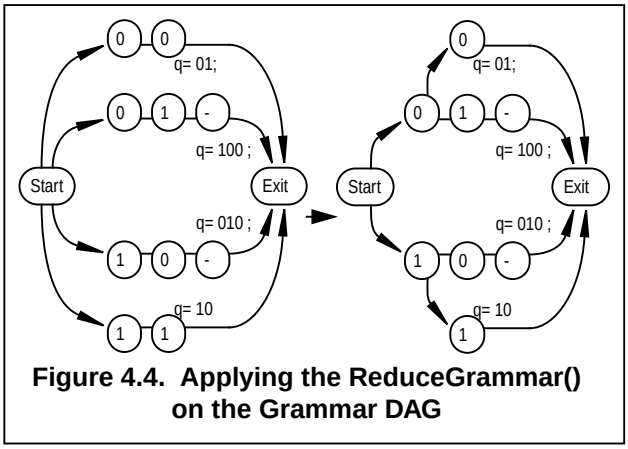

Figure 4.4. Applying the ReduceGrammar() on the Grammar DAG

The grammar DAG could at this point be slightly ambiguous. For ease of description a designer is allowed to duplicate terminals and patterns in multiple branches. After applying WordAlignTerminals(), two alternatives in a rule may end up in having the same condition for transition. This ambiguity is resolved by reducing out all duplications using the ReduceGrammar() algorithm. Input sequences that are identical are merged. Two sequences are identical if they have the same transition and the same actions assigned to it. The result of applying this algorithm on the Manchester Encoder example is shown in Figure 4.4.

When the input sequence ambiguities have been removed out of the grammar DAG, the output sequences, i.e. the actions, must be word aligned to their respective outputs. Action items, that are smaller than their output stream, are grouped together and actions that are wider than their output stream are split into appropriate chunks. Application of this transformation on the encoder is shown in Figure 4.5. All bit sequences associated to the output q is split into a list of actions; all items in the list having the size bit.

The resulting grammar DAG is again slightly ambiguous. To output all bits during the same state is not meaningful. The output assignments need to scheduled over the

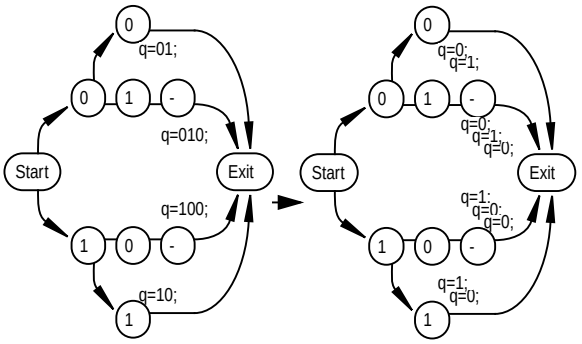

Figure 4.5. Applying the WordAlignActions() on the Grammar DAG

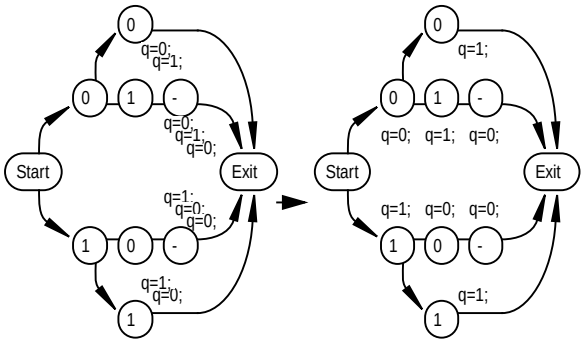

Figure 4.7. Applying the ScheduleUp() on the Grammar DAG

```
ScheduleUp()
begin
   Denote the set of all successors as Succ()
   if symbol has no successors then return;
   for i in all successors loop
      ScheduleUp();
   end loop;
   if symbol has one successor then
      Denote the set of action lists that are written to in
            the successor as Outputs()
      for i in Outputs() loop
         Denote the list of items in the action list Output(i)
               as OutputsToBeScheduled
         Move all elements in OutputsToBeScheduled
               except the last one up to this symbol
      end loop;
   else
      Denote the set of action lists that are written to by the
            successor S as Output(S)
      for i in all Output(S) that are written to by all
            successors S loop
         Denote the list of consequtive items, except the
               last one, that are identical for all action lists
               Output(S,i) as OutputsToBeScheduled.
         Move all elements that are identical in
               OutputsToBeScheduled up to this symbol
      end loop;
   end if;
end ScheduleUp();
```

Figure 4.6. The Up Scheduling Algorithm.

```
ScheduleDown()
begin
   Denote the set of all successors as Succ()
   if symbol has one successor and the successor is the exit
         state then return;
   Denote the set of action lists that are written to by this
         symbol as Outputs()
   for i in all Outputs() loop
      for jin all Succ() loop
         if the Ouput(i) is writing to an output also written to by
               Succ(j)
            Copy all the items, except the first, in the list
                  Output(i) to the beginning of the action list of
                  the output that is also written to by Succ(j)
         else
            Copy all the items, except the first, in the list
                  Output(i) to Succ(j)
         end if;
      end loop;
      Delete all the items, except the first, in the list Output(i)
   end loop;
   for i in all successors loop
      ScheduleDown();
   end loop;
end ScheduleDown();
```

Figure 4.8. The Down Scheduling Algorithm.

transitions. This is done using the ScheduleUp() algorithm shown in Figure 4.6.

The ScheduleUp() algorithm is a recursive algorithm that is applied depth-first on all the symbols in the grammar DAG. If there is only one successor, all but the last item in all the action lists of the successor are moved up to this symbol. If there are many successors, only those outputs which have their action lists written by all successors are considered for scheduling. To do otherwise would violate the specification. All the items but the last in the considered action lists are then compared, starting from the first item, to find out how many consecutive items that are identical between the action lists. The consecutive items that are identical are then moved up to this symbol. The result of applying the ScheduleUp() algorithm on the encoder example is shown in Figure 4.7.

The assignment 0 followed by 1 is moved up in the hierarchy, starting from the x transition in the upper part of the figure, to the preceding transition, the 1. Then the assignments of the transition 0, the 0, is compared with the assignments of the transition 1, also a 0. Because the assignments to be moved are identical, the assignments can be moved up to the preceding transition, the first 0 transition.

When the ScheduleUp() algorithm has been applied, there could be several outputs that have not been scheduled. This happens when there are more than one item left after identifying the sequence of identical consecutive items. These are taken care of by the ScheduleDown() algorithm shown in Figure 4.8.

The ScheduleDown() algorithm is applied recursively in a top-down manner to all symbols in the grammar DAG.

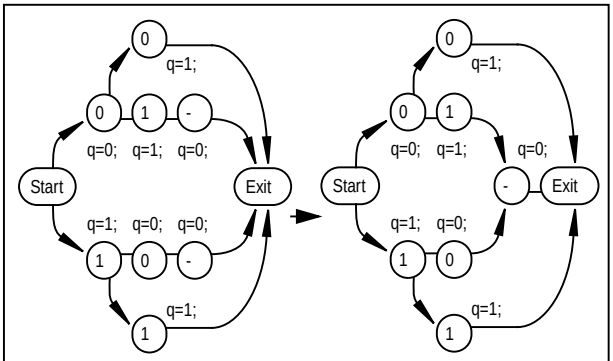

Figure 4.9. Applying the ReduceTailStates() on the Grammar DAG

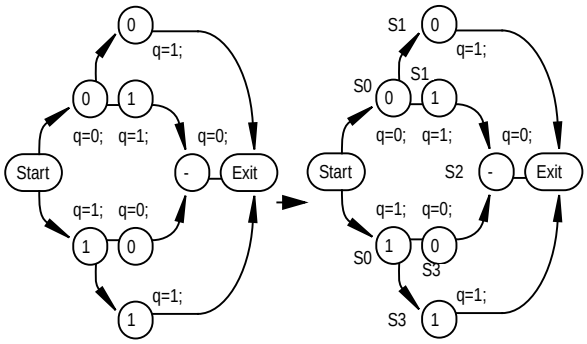

Figure 4.10. Applying the MarkAllStates() on the Grammar DAG

If a symbol has more than one item left in an action list the first item is kept and the remaining items are copied to the successors. If a successor has an output with an associated action list that are writing to the same output stream as the action list that is to be copied, the copied list is inserted in the beginning of the associated list. If the successor does not have an output that are writing to the same output stream, a new output, with the list to be copied associated to it, is added to the successor's list of outputs. When the ScheduleDown() algorithm has been applied, it can happen that the outputs of the symbols preceding the exit symbol have more than one item in their associated action lists. This case is the result of a grammar that does not contain enough transitions to schedule the output, which has resulted in a solution that is partly unschedulable and is reported as an error.

In the Manchester encoder example, this error would have occurred, had the assignments not been identical in the two alternative branches. The ScheduleDown() algorithm would have moved the assignments down through the hierarchy ending up with two assignments in the states preceding the exit state. The specification does not contain enough transitions for the outputs to be scheduled properly and the error would be reported.

After the output assignments have been scheduled over the transitions, there is still an opportunity for optimising the final FSM. Transitions with multiple preceding transitions, like the exit state, may have identical tokens and identical assignments on some of the incoming transitions. In this case it is useful to merge these transitions. The application of the ReduceTailStates() algorithm on the example is shown in Figure 4.9. The ReduceTailStates() algorithm is almost identical as the ReduceGrammar() algorithm presented in [15] but is applied Bottom-Up starting from the exit symbol and comparing and merging predecessors instead of being applied Top-Down starting from the start symbol and comparing and merging successors.

After the outputs have been scheduled and the grammar has been optimised, states are assigned to the grammar DAG. This can be done in two ways. For designs that have long decision trees and the branches of the tree are of approximately equal size, there is an advantage in splitting

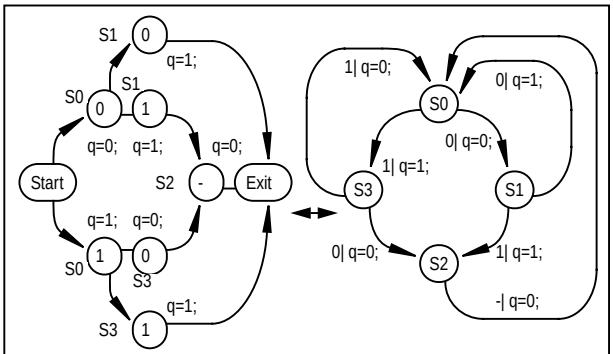

Figure 4.11. Final Grammar DAG vs. State Diagram of the Manchester Encoder

the controller into a depth counter and a branch selector. The depth counter would then keep track of how far down in the tree the controller is and the branch selector would keep track of in which branch of the tree the controller is in. The partition of the controller into two parts reduces the complexity of the problem, and thereby the time and memory spent in state optimisation, and thus allows synthesis of much larger designs. For smaller designs and for designs that don't have near equal length of their branches, it is more advantageous to assign all states to the same FSM.

The result of applying the MarkAllStates() algorithm on the Manchester encoder is shown in Figure 4.10. The first state is state_0. Then the trailing transitions are followed from the left to the right. The state_1 is assigned to the transitions going out from transition 0 of state_0 and state_2 is assigned to the transition x going out from transition 1 from state_1. The state_3 is assigned to the transitions going out from transition 1 of state_0. Since the transition x going out from transition 0 of state_1 already has a state assigned to it, this transition branch is not followed. The final state diagram of the self synchronizing Manchester Encoder is shown in Figure 4.11.

When the states has been marked the final Grammar DAG is output as an FSM in VHDL in a format suitable for synthesis by Synopsys. The two possible target architectures of the synthesized design are shown in Figure 4.12. The first architecture consists of a depth counter in the form

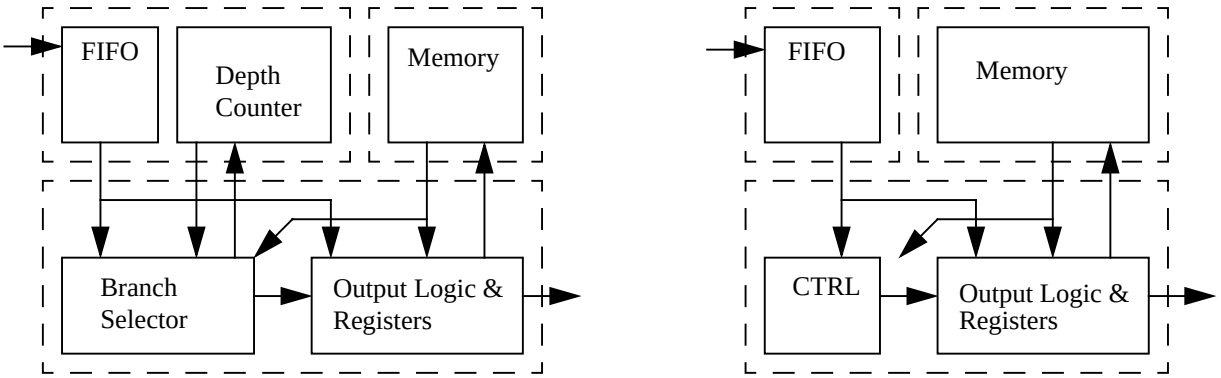

Figure 4.12. The two Possible Target architectures.

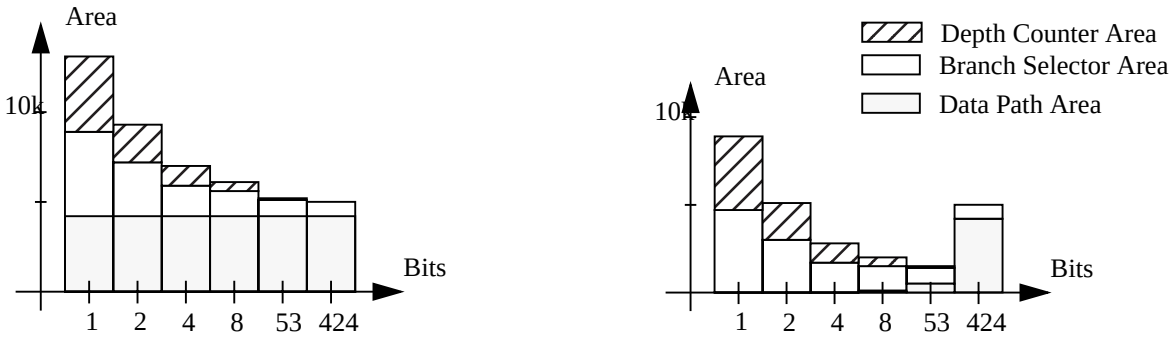

Figure 5.1. Design Space Exploration. a) Area vs. input port width. b) Area vs. input & output port width.

of a token pipeline, one FIFO per Input Stream to allow the actions to refer to the values of previous parsed symbols, a branch selector in the form of a symbolic State Machine, memories and output logic plus registers. Before Logic Synthesis, the token pipeline, the FIFO and the memories need to be extracted from the VHDL-code since these are poor candidates for optimisation by Logic Synthesis. The second architecture is similar to the first one but the Branch Selector and the Depth Counter is replaced with a single FSM controller. It should be noted that both the FIFO and the memory are optional and are only instantiated if they are needed. The Manchester encoder example uses the second architecture with a single controller. The FIFO and the memory are not needed so the design will consist only of a single controller together with output logic and registers.

5. Results & Discussion

A small part of the F4 Operation and Maintenance Protocol for ATM-switch systems [9] has been synthesised for different input port widths to test the efficacy of our approach. The design parses the incoming ATM-stream and extracts different types of OAM-cells. The synthesis results for different bitwidths are shown in Figure 5.1. The datapath area includes the area of muxes and output registers. Memory and FIFO area are not included in the area figures. The lsi_10k was used as a target technology.

In Figure 5.1. a) the output width is fixed to 424 bits, which means that the protocol is performing a series to parallel conversion, and therefore the datapath area is constant. If we instead would have selected the output width to be the same as the input width we get Figure 5.1. b). Here we can see that the design has a minimal area at a port size of 53 bits. However, the design with 8 bits is almost equal in size. Since the Pad area is not included in the figures and since it is always easier to handle 8 bit ports than 53 bit ports, the 8 bit design is probably a better choice. The Depth Counter was realized as a Johnson-counter, i.e. a one-hot encoded counter and the Branch Selector was implemented using binary encoding. It should be noted that the designs with bit widths one and two did not meet the timing requirements because of too long a critical path in the next state logic. For these two cases, the whole state machine needs to be one-hot encoded in order to meet the timing constraints. Thus, the proper selection of target architecture would in this case have been to select a single FSM as the controller.

We also synthesized a smaller design, the earlier presented Self-Synchronizing Manchester Encoder. This was compared to a) a manually designed encoder expressed at RTL in VHDL and b) a behavioural description synthesised with a commercial HLS tool. Both designs were written and manually optimised by an experienced designer. The results are displayed in Table 5.1.

Table 5.1. Program vs. VHDL coded for HLS and manually coded RT level VHDL

	HLS	RTL	ProGram
Number of lines in the specification	30	40	10
Resulting Area after Logic Synthesis (Gates)	75	28 (31)	28
Minimum Clock Period (ns)	6.0	3.6	3.6
Critical path (clock cycles)	2	1	1
Number of states	9	4	4

The first thing important to note is that the HLS-*code* is not so much smaller than the manually written RTL VHDL-code. This is because the functionality contains many transitions. The ProGram code is 3 times smaller than the HLS code. The second thing to note is that the ProGram code got identical result after synthesis as the manually optimised code, both in terms of performance and in area. The manually synthesized design was actually larger in the first try. This was caused by the synthesis tool which was given more freedom to find a solution, but after constraining the search space the manual design was of equal size. The HLS coded design was 2.7 times bigger than the RTL and ProGram designs. In addition, it required two clock cycles to perform the functionality with a 1.67 times slower clock and five more states.

6. Conclusions and Future Research

We have extended our methodology for specifying data communication protocols in an implementation independent manner and synthesizing hardware from such specifications with output scheduling. In addition, we have added a state reduction algorithm to improve the produced state machine. The synthesis tool developed can be used to explore alternative realisations with different widths of the I/O ports.

The initial experiments on the quality of the produced code is promising, but it is too early to draw any conclusions from it since the number of synthesized designs are small. The quality of the produced VHDL code for small designs seems to be as good as the code produced by an experienced designer. However, to draw any general conclusion, an extensive case study needs to be performed.

A problem with our current approach is that the output assignments that consist of datapath calculations that are wider than the stream output needs to be split over the states. Ponder the case where an addition is performed and only the lower half of the result should be used in a cycle and the higher half in the next cycle. In the current implementation, a full size addition is scheduled and the result stored in a register, whose value is kept for two cycles. This results in an unnecessary slow hardware. Instead the addition could be performed in two steps, the lower half of the result calculated during the first cycle and the second half calculated during the second cycle.

The case of a solution that is partly unschedulable caused by a reduce-conflict in the grammar can be solved using a pipelined controller mechanism that output the values during the first transitions of the next pass of the controller. As long as there are no output assignments scheduled in these control steps for the output in question, the reduce-conflict is resolved.

A number of other design options can be included to explore a larger design space. These include alternative state partitioning and encoding choices, alternative synchronisation schemes and various pipelining options. The synchronisation schemes could be organized in the form of a macro library which could be accessed by the synthesizer.

REFERENCES

[1] A. Seawright, F. Brewer, “Synthesis from Production-Based Specifications”, 29th DAC, pp 194-199, June 1994.

[2] A. Seawright, F. Brewer, “High Level Symbolic Construction Techniques for High Performance Sequential Synthesis”, 30th DAC, pp 424-428, June 1994.

[3] A. Seawright, F. Brewer, “Clairvoyant: A Synthesis System for Production-Based Specification,” IEEE Trans. on VLSI Systems, vol. 2, pp 172-185, June 1994.

[4] A. Seawright, Grammar-Based Specifications and Synthesis for Synchronous Digital Hardware Design, Ph. D. Thesis, Univ. of California, Santa Barbara, June 1994.

[5] G. J. Holzmann, “Specification and validation of Protocols”, Prentice-Hall International Inc., 1991.

[6] S. C. Johnson, “YACC, Yet another compiler compiler”, Computing Science Tech. Rep. 32, AT&T Bell Lab., Murray Hill, 1975.

[7] A.V. Aho, R. Sethi, J.D. Ullman, “Compilers, Principles, Techniques and Tools”, Addison-Wesley Publishing Company, 1986.

[8] Edited by K. J. Turner, “Using Formal Description Techniques”, John Wiley & Sons Ltd., 1993.

[9] “B-ISDN Operation and Maintenance interface principles and functions”, ITU-T Recommendation I.610.

[10] M. E. Lesk, “Lex-A lexical analyzer generator”, Computing Science Tech. Rep. 39, AT&T Bell Lab., Murray Hill, 1975.

[11] D. D. Gajski, F. Vahid, S. Narayan, J. Gong, “Specification and Design of Embedded Systems”, Prentice Hall, 1994.

[12] F. Vahid, S. Narayan, D. D. Gajski, “SpecCharts: A VHDL frontend for embedded systems”, IEEE Trans. on CAD, vol. 14, pp 694-706, 1995

[13] D. D. Gajski, S. Narayan, L. Ramachandran, F. Vahid, “System Design Methodologies: Aiming at the 100 h Design Cycle”, IEEE Trans. on VLSI Systems, vol. 4, pp 70-82, March 1996.

[14] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe, and J. Buck, “A System for Compiling and Debugging Structured Data Processing Controllers”, in Proc. of EuroDAC ‘96, Geneva, Switzerland, September 1996.

[15] J. Öberg, A. Kumar, A. Hemani, “Grammar-based Hardware Synthesis of Data Communication Protocols”, In Proc. of ISSS’96, pp 14-19, La Jolla, California, Nov 6-8, 1996.