A Unified Scheduling Model for High-Level Synthesis and Code Generation

Augusli Kifli Gert Goossens Hugo De Man*

IMEC, Kapeldreef 75, B-3001 Leuven, Belgium

Abstract

Scheduling is an essential task both in high-level synthesis and in code generation for programmable processors. In this paper we discuss the impact of the controller model on the scheduling task *for DSP applications. Existing techniques in* high-level synthesis *mostly assume a simple controller model in the form of a single FSM. However, in reality more complex controller architectures are often used. On the other hand, in the case of* programmable processors*, the controller architecture is largely defined by the available control-flow instructions in the instruction set.*

In this paper, a unified scheduling model *is presented to handle a wide range of controller architectures, from the application-specific to programmable processor solutions. The impact of chosing a certain controller architecture on the scheduling phase is investigated. Finally, the tasks of controller generation and code assembly are discussed, which will generate the FSM or machine code description from the correct schedule.*

1 Introduction

In the past few years, many high-level synthesis systems have been developed to map behavioral descriptions into *application-specific IC architectures* (ASICs). Often these systems focused on real-time DSP applications. The advantage of ASIC solutions is their cost efficiency. For competitive markets like consumer electronics or telecommunications, ASICs however often lack flexibility and programmability.

In the past, *commercial (DSP) processors* were the only choice when programmability was desired. More recently, a trend emerged in the DSP community to build *application-specific integrated processors* (ASIPs), which offer programmability, while maintaining low cost and power consumption as well as high speed. The design of a compiler for each new ASIP is however a time consuming and tedious task. This has lead to a renewed search for efficient techniques for *retargetable code generation.*

In a *real-time signal processing system*, the advancement in processing technology has enabled the possibility to integrate the complete system on one chip. The so-called *heterogeneous ASIC architecture* contains the necessary memory, accelerator data paths, an embedded programmable processor and the glue logic to put these components together [3]. The DSP core can come in the form of a commercially available DSP processor or ASIP. It is used to implement the low to medium throughput data processing parts and some control functions of the system. The accelerator data paths on the other hand can be used to implement the more time critical (high throughput) functions of the system. They can be synthesized by a high-level synthesis tool.

This paper is concerned with control strategies for the different parts of a heterogeneous ASIC. Accelerator data paths which are usually non-programmable can be steered by a FSM type of controller or a microcoded controller. The embedded programmable processor has its own sequencer and instruction decoder. Different types of controllers will strongly influence the *scheduling task* in order to produce a valid and effective schedule. Most high-level synthesis systems, [10] [9] among others, assume a controller in the form of a single FSM. Some support a more complex microcoded controller as in [6] [11] [7] [8]. However no system that is known currently to the author supports a range of controller architectures.

We present a unified model to capture the control flow of an application program and the characteristic of the underlying controller architectures, which allows the scheduler to generate a valid and efficient schedule. The contributions of this paper are the following:

- In Section 2, we will first review a broad *range of different controller architectures* which are commonly used in the DSP domain.
- Next, in Section 3 we will introduce an efficient model to represent conditional branches in a control-data flow graph (CDFG).
- It will be shown in section 4 how this model enables the formulation of additional *scheduling constraints*, which exactly model the influence of each target controller architecture on the synthesis or code generation process. In this way, the scheduler can *guarantee the correctness* of the produced FSM or machine code descriptions with respect to the given controller architecture.
- For ASICs, existing high-level synthesis systems usually try to optimize the operator or storage (and sometimes interconnect) cost. However, in some cases the eventual control logic is dominating the silicon area and delay. In the case of code generation for ASIPs and commercial DSP processors, scheduling aims at reducing the number of instructions and the execution time of the program. Excessive amounts of branch instructions may however deteriorate these parameters and should therefore be avoided. This will the topic of discussion in section 5.

The models and techniques described in this paper are part of the *Chess* compiler, a retargetable code generation system with extensions for high-level synthesis.

2 Controller architectures

Synthesized hardwired FSM. Most high level synthesis systems assume this type of controller. This is a non programmable controller and hence can only execute one specific application program. The control logic is synthesized after scheduling, when the state transition graph of the application program is known. The state transition graph may contain m*-way branches*, with m an arbitrary integer. We call these *multi-way branches.*

Hardwired microcoded controller. This type of controller architecture also executes only one application program since it is

*Professor at Katholieke Universiteit Leuven, Belgium

non programmable. It consists of two major parts : a *microprogram ROM*, which stores the micro-instructions, and a *sequencer block*, which controls the execution sequence of the program. A micro-instruction usually takes only one cycle to execute. Since the sequencer block is to be synthesized, it basically has the power of hardwired FSM controllers (i.e. multi-way branch). The sequencer itself usually contains an incrementer/adder to generate sequential addresses. The non-sequential (branch) addresses are generated by a dedicated logic block. This type of controller architecture is used in *Cathedral* [6] [11], *ASYL* [8] and [7] among others.

Programmable microcoded controller. As mentioned in Section 1, there is a trend in the DSP community to build programmable architectures in the form of ASIPs. In this case the controller architecture itself must be programmable. Usually a programmable microcoded controller is chosen. Unlike the hardwired microcoded controller, the sequencer and instruction decoder blocks of the programmable microcoded controller are *fixed in advance*. This means that there is a limitation on the *type of branches* that can be executed. In most cases, only the *2-way conditional branch* and the *unconditional branch* are available. Depending on the actual architecture, these branch operations may or may not be executed in parallel with other operations. The micro-instructions can be stored in RAM, EPROM or ROM.

The controller architecture in this case can be partly represented by the available *control flow instructions*(conditional and unconditional branch) defined in the instruction set. In this paper, the term *controller architecture* will be used to represent the control flow instructions when we talk about programmable processors.

Programmable macrocoded controller. In this case, the instructions in the program ROM are *macro*-instructions rather than *micro*-instructions. The instructions can take more than one cycle to execute in the execution unit of the processors. This controller architecture is found in a number of commercial DSP and microprocessors [1].

We do not claim that the above classification is complete but we believe that most existing controller architectures fall into one of the above categories.

3 Modeling of conditional branch

A classical approach used in software compilation and in high level synthesis, is to break up the CDFG into basic blocks which depend on the control constructs. Control dependencies are introduced *between* basic blocks, but do not occur *inside* a basic block. Older scheduling tools are mostly basic block schedulers. A basic block scheduler only considers data dependencies and resource conflicts within basic blocks but does not move operations across the basic block boundaries. The latter is however crucial to obtain highly optimized schedules.

We use a different model for control constructs, with two major advantages :

- It facilitates moving of operations across traditional basic blocks boundaries during scheduling, thus optimizing the condition structure.
- It can be used by the scheduler to generate a controller implementation for a wide range of controller architectures, including most of those introduced in Section 2.

A *conditional branch* consists of a number of *conditional paths* and a *condition select node*. The conditional paths will be activated depending on the value of the signal evaluated by the condition select node. For example, a 2-way conditional branch has two

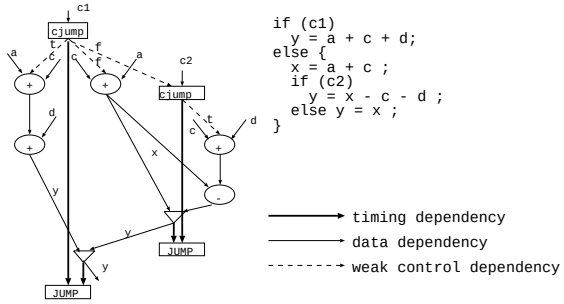

Figure 1: CDFG model of conditional branch

conditional paths, referred to as the "true" and "false" conditional paths. In [9], several representations of a conditional branch have been discussed. The representation we use is similar to the one used by [9] and [2] in a number of aspects.

Figure 1 shows a *2-way conditional branch* and the corresponding CDFG representation. A 2-way conditional branch is represented by one "CJUMP" node, one "JUMP" node, and one or several "MERGE" nodes. The MERGE nodes are represented by reverse triangles in the figure. A *case statement* is represented as nested 2-way conditional branches.

The CJUMP node starts the conditional branch by evaluating the values of certain status signals (conditions). The MERGE nodes mark the definition of signals based on different values produced in each conditional path. The values (signals) being transferred out of the conditional branch must go through MERGE nodes. The inputs of a MERGE node are two signals which are defined in a mutually exclusive way, one coming from the true and the other from the false conditional path. The operations that consume the signal produced by a MERGE node can only be scheduled after the MERGE node is scheduled. The different MERGE nodes associated with a CJUMP node can be scheduled at different time steps.

To represent optimizations of the condition structure in an easy way, the concept of *weak control dependencies* is introduced. Weak control dependencies denote dependencies that can be violated. They are added between every *CJUMP node* and *all operations inside the conditional branch* started by this CJUMP and terminated by the corresponding JUMP (see Figure 1). Whether or not a weak control dependency is violated, is decided by the *scheduler*. A violation of a weak control dependency means that the target operation to which it is pointing will be executed *irrespective of the condition value* (i.e., it will be executed *unconditionally* with respect to this condition). The position of the CJUMP and JUMP nodes in the schedule defines the *scope* of the conditional operations.

3.1 Checking mutual exclusivity

The scheduler must be able to find out whether two operations are mutually exclusive. For that purpose, *condition vectors* have been introduced in [10]. However, as pointed out in [9], these condition vectors may yield wrong results in specific configurations of conditional branches. The alternative *condition tag* method proposed in [9] itself is a pessimistic approach. This means that when the condition tags can not determine whether two operations are mutually exclusive, they are assumed to be non-exclusive. To alleviate this problem, a *tautology checking* tool is used in our scheduling algorithm to check mutual exclusivity when simple heuristics cannot decide.

4 Constraints imposed by the chosen controller architecture

With the introduced condition representation, the control flow of the application is captured. In this section, it will be shown how the scheduler is able to capture the characteristic of the chosen controller architecture and generate a correct and efficient schedule.

We introduce virtual resources in the controller that can execute the CJUMP and JUMP operations. The resource that is capable of executing CJUMP operations is called *CJUMP operator* and the resource that executes the JUMP operations is called *JUMP operator*. The conflict behaviour of both the CJUMP and JUMP operations is modeled in a similar way as other data operations like addition or subtraction.

The different types of controllers described in section 2 can be characterized by the following parameters:

- Number of CJUMP and JUMP operators that is allocated.
- The delay slot of the CJUMP operation.
- Whether the CJUMP operation is a delayed-branch or not.
- Conflict information between CJUMP, JUMP and other operations.

High-level synthesis systems typically explore different allocations of operators in the data path. In a similar fashion, we can explore different allocations of CJUMP and JUMP operators in the controller. By having more CJUMP operators allocated, we can potentially schedule more conditional branches in parallel. In this way, area/time tradeoffs can be made with respect to the controller architecture. This issue has been ignored hitherto in most high-level synthesis systems. The same approach can also be followed by designers of an ASIP. By iteratively calling a code generator that supports our model, different controller architecture for the ASIP can be explored.

We will now described how the different types of controllers can be characterized in terms of the above parameters.

Synthesized hardwired FSM. Since the FSM which contains the CJUMP and JUMP operators is to be synthesized, we can allocate an *unlimited number of CJUMP and JUMP operators* to execute the CJUMP and JUMP operations. In this way, the scheduler is allowed to generate an unrestricted multi-way branch schedule. The conditional branch representation introduced in Section 3 only has two branches, namely the true and the false branch. Multi-way branches are realized by scheduling several mutually non-exclusive CJUMP operations on the same control step (c-step). In general, when n mutually non-exclusive conditional branches are scheduled on the same c-step, we can have at most a 2^n-way branch in the schedule. Scheduling parallel conditional branches is useful to maximize resource sharing and minimize the schedule length. The delay slot of the CJUMP is the number of pipelines in the FSM controller minus one. The CJUMP operations will always be a delayed-branch. The CJUMP and JUMP operations will not be in conflict with other operations.

Hardwired microcoded controller. This type of controller contains a dedicated logic block to test on the conditions and generate the target jump addresses. Since this logic block will be synthesized only after the CDFG is scheduled, we can again allocate an *unlimited number of CJUMP and JUMP operators*. However, similar to the case of FSM controller architectures, we can play with the number of branch operators to make area/time tradeoffs.

The microcoded controller in most cases is a *pipelined* architecture. We can easily model the *delay slot* [4] of the CJUMP operations by putting a weight on the weak control dependencies that exist between a CJUMP operations and all operations in the conditional branch started by the CJUMP operation. For example, when we have a delay slot of 1, the weight of the weak control dependency is set to 2, which means that the dependency's target operation is only executed conditionally with respect to the given condition when it is scheduled 2 c-steps after CJUMP is scheduled. Other operations may or may not be scheduled in the delay slot, depending on whether we have a *delayed-branch* [4] architecture or not. Delayed-branch is useful in order to reduce the penalty of control hazards. Together with speculative scheduling of operations, the penalty of control hazard is more or less minimized.

Programmable microcoded controller. Programmable processors usually contain conditional branch instructions(for example: branch if zero) in their instruction set. The CJUMP operations will eventually be mapped into one of these control flow instructions available in the instruction set of the programmable processor depending on the condition flag that the CJUMP operation is evaluating.

By limiting the number of CJUMP and JUMP operators to execute the CJUMP and JUMP operations, we can model a programmable controller architecture. In most DSP processors and ASIPs, only *2-way* conditional branch and unconditional branch instructions are available. We can model this by allocating *only one CJUMP operator*. The number of parallel instruction streams that can be issued by the processor is defined by its instruction set. In some high performance processors, branch operations (CJUMP, JUMP) can be put in the same instruction with other data operations. Some processors support *fall through m-way* conditional branches. In the fall-through m-way conditional branch, $m-1$ branch target addresses are put in the instruction word. To model the fall through m-way conditional branch, $m-1$ *branch operators* are allocated. In addition, the constraint is imposed that only the CJUMP operations belonging to the same nested conditional branch can be scheduled on the same c-step.

The controller of the ADSP-2100 from Analog Devices can be categorized in this class of controllers. The CJUMP operation has no delay slot which means that no penalty is incurred by executing a CJUMP operation. The CJUMP and JUMP operations can not be executed in parallel with other operations. It allows only 2-way conditional branch which means only one CJUMP operator is allocated.

Programmable macrocoded controller. This type of controller is mainly present in commercial DSP processors or cores. The instructions can take more than one cycle to execute in the data path. In this case, structural hazards can be present. Structural hazard exists when two instructions want to use the same execution unit. We are currently working on a model [5] which will be able to capture structural hazards.

For example: the 2-way conditional branch instructions in the TMS320C5x instruction set have 3 delay slots. They can not be issued in parallel with other operations. The 2-way conditional branch instructions of Motorola 96002 have no delay slot and can not be issued with in parallel other operations.

Explicit modeling of the various controller architectures allows the scheduler to generate a correct schedule for the underlying controller. The schedule which is achieved by assuming a certain controller architecture may not be a valid schedule for another controller architecture, not to mention the optimality of the schedule.

5 Global scheduling and control generation

5.1 Global scheduling

The scheduler used in our system is a list-based scheduler. It is not the intention of this paper to discuss the scheduling technique. We will briefly point out some features that are present in our scheduler.

- With the weak dependencies that are introduced, the scheduler is able to schedule an operation speculatively. The execution condition of an operation is updated whenever a weak dependency to that operation is violated.
- The list scheduling priorities of operations that violate the weak dependencies are lowered such that they will not compete too heavily for resources with other operations which retain their execution condition.
- The *priorities* of two *mutually exclusive operations* that utilize the same type of resource are increased by our list scheduler. This is done to increase their chances to share resources.

The controller architecture that is chosen not only imposes constraints on the scheduler but also affects the scheduling strategy in order to produce a better result. This will be demonstrated by the following small example. Suppose we are given a CDFG as shown in figure 2(a). We would like to map the CDFG into an architecture with two general purpose ALUs which can perform comparison and arithmetic operations. We will show three different schedules, corresponding to three different controller architectures, each time assuming the same data path composition.

The first controller architecture has 1 CJUMP and 1 JUMP operator allocated (2-way branch controller). The CJUMP operation is a delayed-branch and has a delay slot of 1. The second one is the same as the first one except that 2 CJUMP operators are allocated. The third one has unlimited CJUMP and JUMP operators allocated and has no delay slot for the CJUMP operation (m-way branch controller). In all three controllers, the CJUMP and JUMP operations can be scheduled in parallel with the addition and subtraction operations. The CJUMP and JUMP operations however can not be scheduled in the delay slot of a CJUMP operation in the first 2 cases.

The scheduling results for all three cases are shown in figure 2. The scheduler has changed some execution condition of some operations. These operations are shown shaded in figure 2. In figure 2(a), the second CJUMP operation can not be scheduled on cstep-2 because we have only 1 CJUMP operator allocated for the controller. The two CJUMP operations can be scheduled in the same cstep in figure 2(b) since we have two CJUMP operators in the controller in the second case. The schedule can be further compacted in figure 2(c) when there is no delay slot for the CJUMP operations. The delay slots of the CJUMP operations are filled with useful operations in the first two schedules. This small example shows how a valid and efficient schedule can be produced for different types of controller architectures by using the model introduced in this paper. The rightmost schedule for the m-way branch controller is not a valid one when we choose a controller of type assumed in case 1 or 2. The schedule in figure 2(a) is also a valid schedule for the other 2 types of controllers but clearly not an efficient one.

5.2 Control generation

After scheduling, we can construct the state transition graph in a Moore model. The state transition graph can then be mapped into the chosen controller architecture. When the FSM is chosen as the controller architecture, logic synthesis tools can be used to map the state transition graph into PLA or multilevel logic. When a microcoded type of controller is chosen, address assignment is done and the states in the state transition graph correspond to the instruction words in the microprogram.

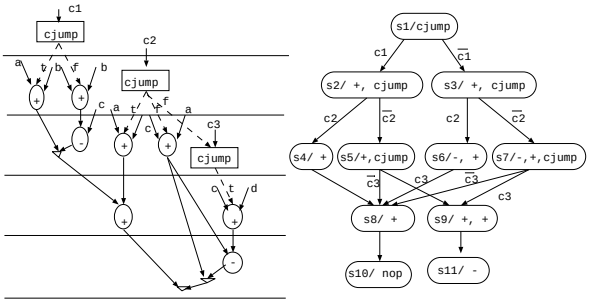

Figure 3: Optimized schedule and state transition graph .

In some DSP algorithms where fixed sample rate is desired, worst case schedule length is of interest. Fixed sample rate means that we can introduce input samples at a fixed time interval which will reduce the complexity of the interface logic. If fixed sample rate is indeed desired, all possible conditional paths must be given the same length. Shorter execution paths due to conditional branches are therefore padded with NOPs (no-operations), in order to achieve a fixed schedule length. This also means that each operation in the CDFG is mapped to one fixed c-step, regardless of which conditional path is taken. In the sequel, we will assume that all conditional paths must be given the same length.

The execution of two parallel conditional branches can be overlapped. Overlapping execution of conditional branches incurs a certain cost in the controller. The complexity of the controller is proportional to the number of states and transitions in the state transition graph.

Figure 3 is an example taken from [10]. It shows the schedule of two parallel conditional branches and the corresponding state transition graph. Since we require that all the execution paths have equal length, shorter execution paths are padded with NOPs. States which contain NOPs only are present in the state transition graph. This schedule executes the two conditional branches in parallel to minimize the machine cycle count. It assumes a FSM type of controller. The schedule shown in Figure 3 takes 5 c-steps to execute and needs 11 states in the state transition graph. The schedule would require 6 c-steps and 11 states if the two conditional branches are scheduled in sequence. This demonstrates that some optimization can be done to *reduce the number of machine cycles and states* when parallel conditional branches are overlapped. The operations that are executed in each state are shown after the state number.

In general, overlapping of parallel conditional branches results in a certain degree of *state duplication*. When the CJUMP operation a has delay slot and only 2-way conditional or fall-through m-way conditional branch is allowed, uncareful overlapping of conditional branches can result in excessive amounts of code duplication. The scheduler tries to schedule operations that belong to the same condition tree as closely to each other as possible. By doing so, the code duplication is minimized.

6 Experiments

We have implemented the scheduler which uses the models described in Section 3 in order to produce valid and efficient schedule for different types of controllers.

We have tested the scheduling algorithm on *real life design examples*. Table 1 shows the result of the schedule with different

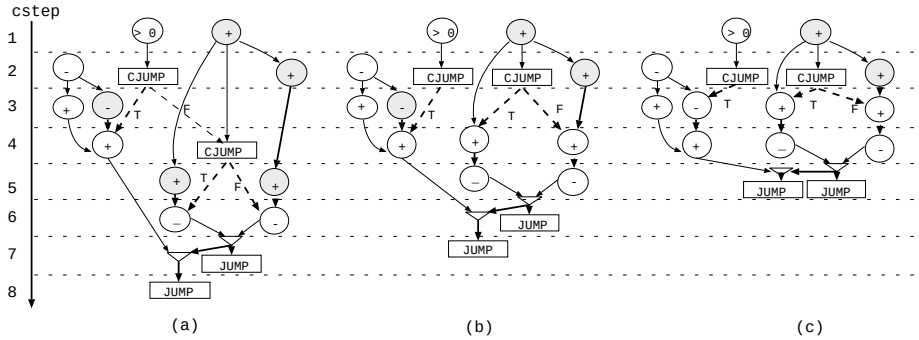

Figure 2: Schedules for different controller architectures.

design	# nodes	ctrl arch.	*method 1*		*method 2*	
			# c-s	# sts	# c-s	# sts
test	30	FSM	12	20	11	16
		2-way μc	16	45	14	21
		m-way μc	15	26	12	20
ched	67	FSM	25	40	24	38
		2-way μc	25	82	24	57
		m-way μc	26	43	25	40
echo	160	FSM	43	60	40	53
		2-way μc	44	73	45	68
		m-way μc	43	60	40	51

Table 1: Experimental results

controller architectures. The three examples shown in Table 1 have a quite complex condition structure in the CDFG. The “test” example contains 2 parallel conditional branches. One of them is a nested conditional branch, while the other contains a loop structure. The “ched” example is a part of a GSM channel decoder algorithm. It has a loop structure which contains 5 parallel conditional branches and one of them is nested. The “echo” example is a full echo canceler algorithm. The input to the scheduler is a CDFG in which the functional unit and register assignment as well as interconnect synthesis are already performed.

Method 1 in the table refers to the schedule where speculative execution is not allowed while *method 2* allows speculative execution. Both methods uses the combination of scheduling beyond basic blocks and parallelization of conditional branches.

The following controller architectures are used. “FSM” refers to a single non-pipelined FSM controller. “2-way μ-coded” is a microcoded controller with a pipeline delay of 3 and a delay slot of 1 for CJUMP operations. Only one CJUMP operator is allocated which means that only 2-way branches can be generated in the schedule. However, the CJUMP and JUMP operations do not conflict with other operations. Finally, “m-way μ-coded" is similar to “2-way μ-coded”, except that an unlimited number of CJUMP and JUMP operators are now allocated.

7 Conclusion

We have presented a scheduling model which can cope with the restrictions imposed by a wide range of controller architecture types. Scheduling operations independent of control dependencies, parallel execution of conditional branches and optimization of resource sharing are features present in our scheduling algorithm. The scheduling model we introduce allows the scheduler to produce a valid and efficient schedule for the chosen controller architecture. This versatile scheduling model is an essential component of a hardware/software codesign environment, where the use and co-existence of different architecture targets (accelerator data paths, various programmable processor cores) must be explored.

Acknowledgments. The research described in this paper has been carried out in the context of the ESPRIT 2260 (Sprite) project of the European Union. The “ched” example has been worked out in cooperation with Alcatel-Bell (Antwerp).

References

[1] Edward A.Lee. "Programmable dsp architectures, part i, ii". In *IEEE ASSP Magazine*, October 1988.

[2] A. Fauth and A.Knoll. "Translating signal flowcharts into microcode for custom digital signal processors". In *ISCP 93*, 1993.

[3] G. Goossens, I. Bolsens, B. Lin, F. Catthoor, "Design of heterogeneous ICs for mobile and personal communication systems", *Proceedings IEEE International Conference on Computer-Aided Design* (ICCAD-94), San Jose (Calif., U.S.A.), November 1994.

[4] D. Patterson J. Hennessy. "Computer Architecture - A Qualitive Approach". Morgan Kaufmann Publishers, 1990.

[5] J. Van Praet, G. Goossens, D. Lanneer, H. De Man. "Instruction Set Definition and Instruction Selection for ASIPs." *Seventh International Symposium on High-Level Synthesis*, Niagara-on-the-lake, Ontario, Canada, May 1994.

[6] A. Kifli and et. al. "Flag/Condition Handling and Branch Assignment for Large Microcoded Controllers" . In G. Saucier, editor, *Control dominated Synthesis From a Register Transfer Description*. Elsevier Science Publisher, 1992.

[7] S.Z. Lin, H.T. Hwang, and Y.C. Hsu. "Efficient Microcode Arrangement and Controller Synthesis for Application Specific Integrated Circuits". In *Proc. Int. Conf. on Comp. Aided Design*, pages 38–43, 1991.

[8] F. Poirot and G. Saucier. "Controller Synthesis in the ASYL System". In North Holland, editor, *International Workshop on Logic and Architecture Synthesis for Silicon Compilers*, 1989.

[9] Minjoong Rim and Rajiv Jain. "Representing conditional branches for high-level synthesis applications". In *Proc. 29th Design Automation Conference*, pages 106–111, 1992.

[10] K. Wakabayashi and H. Tanaka. "Global scheduling independent of control dependencies based on condition vectors". *Proc. 29th Design Automation Conference*, pages 112–115, 1992.

[11] J. Zegers, P. Six, J. Rabaey, and H. De Man. "CGE: Automatic Generation of Controllers in the CATHEDRAL-II Silicon Compiler". In *Proc. European Design Automation Conference*, pages 617–621, March 1990.