

CGE *: Automatic Generation of Controllers in the CATHEDRAL-II Silicon Compiler.

J. Zegers, P. Six, J. Rabaey,[†]H. De Man. [‡]

IMEC, Kapeldreef 75, B-3030 Leuven (Belgium)

Abstract

This paper describes the efforts done within the framework of the CATHEDRAL-II silicon compiler towards automatic controller generation. The program CGE (Controller Generation Environment) maps a microcode description generated by Atomics, an RT scheduler, onto a target controller architecture. The program produces logic, structure and layout descriptions of the constituent blocks. The controller architecture has been chosen to suit most of the audio, speech, telecom and back-end image processing real time algorithms (throughputs up to 1MHz).

1 Introduction

Fast design of complex ASIC's for digital signal processing algorithms requires the use of silicon compilation tools. CATHEDRAL-II [DeM 88] is a silicon compilation environment for multiprocessor real time information processing chips. The architectural choices have been reported in [Cat 88]. A high level applicative description of the algorithm, written in SILAGE [Hil 85], is first translated in register transfers, to be executed on predefined modules. This mapping is performed based on rules, described in [VaH 87]. These register transfers are then scheduled in an optimal way using the ATOMICS tool, [Goo 87], taking into account the pipelining of the controller architecture.

Then, the task of CGE is to map the output of the scheduler in an optimal way onto a control architecture. The architecture of the controller has an associated timing behavior, that imposes some constraints on the schedule of the instructions. These pipelining characteristics are input to the scheduler. This means that the controller architecture can be chosen freely.

Our approach to controller generation is different from [DeM 87] and others, which try to solve the controller design as a general single-FSM problem, uncommitted to a certain application field. Since in ASIC design the *algorithm is fixed* at compile time, we can tune and optimize our control architecture to this target application. To handle the control functions in an optimal way, our controller will be composed of *four cooperating blocks*. In particular, an incrementer is used to reduce the complexity of the next

state computation part. The increment problem and its solution, as described in section 3.1, can be used in all control designs using a counter.

Section 2 describes how the control architecture was chosen to fit most of the real-time signal processing algorithms for which CATHEDRAL-II is intended. Then, the operation and features of the controller will be explained. Section 3 describes the optimization steps in the controller generation. Section 4 gives results of applying CGE to industrial controllers. Finally, future research and improvements will be stated and some conclusions will be drawn (sections 5 and 6).

2 Control Architecture and Design Environment

2.1 Choice of the target architecture

The algorithms, used in most medium-throughput (8kHz-1MHz) information processing applications, perform a *lot of decision making*. To avoid extra machine cycles, we therefore opted for a control architecture allowing *multiple branches* as opposed to the two-way branching provided in most microprocessors.

A study of 12 alternative controller architectures was therefore started. These architectures included single PLA controllers, microprocessor-like control structures,... Typical real-world design examples, such as echo canceling, discrete fourier transform and pitch tracking, were mapped onto the different architectures and the area and cycle count they required were evaluated. The results of this study can be found in [DeC 87].

The applications revealed that, when the complexity increased, only two control architectures were able to execute the algorithms in an efficient way. One of these architectures, called the multi-branch controller, has been chosen as a test vehicle for CGE and is depicted in figure 1. An in depth report of the architectural strategies used and trade-offs made in CATHEDRAL-II can be found in [Cat 88].

This architecture contains *four main blocks* and some registers :

- a microcode ROM (μ ROM),
- a status finite state machine (SFSM),
- a branch finite state machine (BFSM) and
- an incrementer (INC).

2.2 Operation of the controller

The μ ROM contains the instructions (i.e. control signals) for the datapath modules and two extra fields, the status and branch jump

*Research sponsored by the EC under the ESPRIT-87 project.

[†]Currently assistant professor at U.C.Berkeley

[‡]Professor at the Katholieke Universiteit Leuven

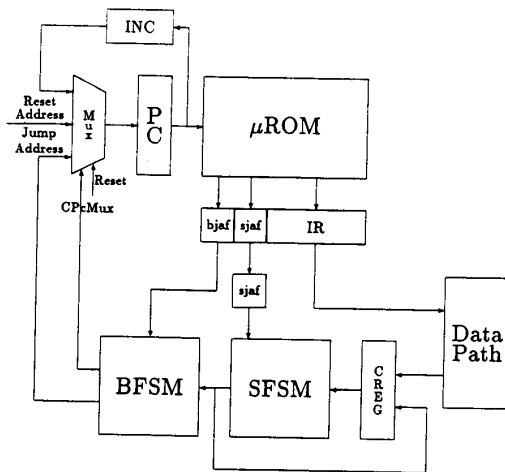


Figure 1: Multi-branch Control Architecture

address fields (SJAF and BJAF). These fields can be seen as instructions for the corresponding FSM's.

The SFSM has to process the status signals, generated by the datapath modules. Under control of the SJAF-bits, it can pass a combination of the status signals to the BFSM or store this information in the condition registers (CREG), for use later on in the algorithm. The BFSM generates next state addresses whenever the control flow has to jump to a state, other than the incremented present state. It also controls the CPcMux signal to load the appropriate address in the PC. The address generated is function of the instruction coming from the ROM (BJAF) and the stored status information (CREG).

As can be seen in figure 1, several registers are present in the architecture. They determine the pipelining of the controller. The Program Counter stores the actual state of the controller. It points to an instruction in the μ ROM. The instruction for the SFSM is delayed by two cycles, to make sure that the instruction for the SFSM arrives at the same time as the corresponding status signals, generated by the datapath. The register at the input of the SFSM (CREG) is present to split the combinatorial delay of the datapath and the SFSM and BFSM delays. The instruction for the BFSM is delayed over only one cycle to cut the combinatorial delay of the μ ROM and the BFSM.

These delays are taken into account when the register transfers are scheduled. The delay for status information to reach the program counter is three cycles. This can be seen in the following way. Execution of an instruction generates a status bit in the datapath. In the next cycle this information is available to the SFSM and the BFSM. One cycle later a new address is loaded into the program counter and still one cycle later a new instruction can be loaded in the instruction register (IR). The delay to start a branch is two cycles, because the program counter will only get its next address after it has rippled through the BJAF-register. This means that the microcode schedule doesn't contain the next-address specification, but the next-next address.

2.3 Why this architecture ?

The following facts are in favor of the chosen architecture :

- Multiple branching is possible with an almost unrestricted branching complexity, with zero overhead in machine cycles (the same holds for loops).
- Several status signals can be treated in parallel, again with no restriction on the complexity.
- For sequential parts of the algorithm the incrementer will provide the necessary addresses for the program counter.
- The architecture can be optimized for particular algorithms. For instance, the incrementer can be stripped in applications where almost no increments can be realized.
- Status and branching information are separated and can be optimized independently.

Negative aspects of the controller architecture are :

- Only for application specific IC's.
- A three cycle delay between the instruction generating a certain status signal and the next useful instruction in the control flow, resulting in possibly two no-operation instructions.

This last drawback of the control architecture can however be overcome by the scheduler, provided it can find instructions that are independent of the status to be generated. Details can be found in [Goo 87].

2.4 The CGE environment

The input to the CG-environment consists of the scheduled microcode and a library with the corresponding detailed control signals for the modules.

For each machine cycle, the microcode schedule symbolically describes :

- CTL-field : The control signals that have to be generated.
- UPD-field : The update equations (for the SFSM).
- JMP-field : The branch address and conditions (for the BFSM).

An example of a typical state description is given :

```
<S1> CTL reg_2_r:W[2,3], mult_1:map;
      UPD CREG[0]=alu_1, CREG[1]=CREG[1];
      JMP IF CREG[1]&!CREG[2]&CREG[0] -> S2
          || !CREG[1]|CREG[2]!|CREG[0] -> S3
      FI;
```

The control field (CTL) specifies the module's name (eg. mult), instance name (eg. 1) and mode (eg. map). The update conditions (UPD) specify which CREG-register has to be updated with a status signal and which have to be stored. The branch information (JMP) specifies to which state the controller has to jump under which condition.

Every module, present in the module library, has several control signals associated with it. The intention of the control library is to have a single description of the control signals for each module. Every module is given a name, followed by the different

modes it can operate in, followed by the corresponding boolean pattern of the control signals. There are two ways to specify these control signals : fully instantiated or vector type. The modules that have fully instantiated control signals are typically ALU's, multipliers,... Every control signal is given explicitly. However, for modules like a register file, it is more convenient to describe the control signals in an indexed way. The program will expand every vectortype description in $\log(M)$ bits, with binary value N .

```
mult:map -> acc = T, inv = F;
reg:R[N,M] -> r[N,M] = T;
```

For each of the blocks of the controller, CGE generates a logic description, either as a two-level table (ESPRESSO format [Bra 84]) or with boolean equations in EQNTOTT-format. These descriptions can be minimized from within CGE, using the espresso- and eqntott-tools. The minimized descriptions of the ROM and FSM's can then be used to generate PLA's using a cell placer and a symbolic library of cells [Rij 87]. Each PLA can also be folded using a constrained folding program, EPSILON [Bar 84]. Multilevel logic implementations are also possible for the SFSM and BFSM.

Finally, the program generates a structure description containing the different blocks of the controller and their interconnection. This netlist and the description of the different blocks can then be used to generate the full layout of the controller, using a module generator [Six 86] or a floorplanning tool.

2.5 Main program loop

To conclude this section we give the main program loop of the CG-environment :

```
main() {
  MicroParser(); /* Parse microcode */
  PreProcess(); /* Expand macro's, ... */
  CreateFlowGraph(); /* Next-instruction graph */
  SFsm(); /* Generate status FSM */
  BFsm(); /* Generate branch FSM */
  MicroRom(); /* Generate microcode ROM */
  Minimization(); /* Logic Minimization */
  Structure(); /* Control structure */
}
```

The MicroParser() routine is a yacc and lex-based microcode and library parser. The CreateFlowGraph() routine will build the next-instruction graph. The SFsm() routine finds the minimal number of SJAF codes and generates the boolean description of the SFSM. The BFsm() routine creates the spanning tree and optimizes it for increments, finds the minimal number of BJAF codes and generates the equations for the BFSM. The MicroRom() routine generates a two-level ROM table containing the PC-value, the BJAF and SJAF codes and the control signals to be generated in each state.

3 Optimizations

The optimizations performed by the CG-environment are mainly :

- Optimal use of the incrementer.
- Minimal number of SJAF and BJAF codes.
- Assignment of SJAF and BJAF codes.

First, we will treat the increment problem, that is most affecting the size of both the BJAF field and the BFSM (section 3.1). Then we will describe how a minimal number of SJAF and BJAF codes is found and how the assignment is done (section 3.2).

3.1 The increment problem

As stated before, the microcode input of CGE contains the next-next address specification as a result of the controller pipelining. The first task is to find the next-instruction graph and to assign a code to every state to maximize the use of the incrementer. This will reduce the number of BJAF's (thus the number of BJAF bits). Moreover, the number of productterms in the BFSM will decrease, since state transitions are shifted to the incrementer and no program counter addresses have to be stored in the BFSM.

The microcode graph (solid lines) and the corresponding next-instruction graph (dotted lines) are shown in figure 2.

The algorithm to find the next-instructions from the microcode uses a depth-first search algorithm, taking into account the conditions on the different branches in the JMP specifications and the update equations, as specified in the UPD part of the microcode. To speed up the operations when comparing the conditions, all the conditions on the branches are minimized using an exact minimization (Quine-McCluskey algorithm), since we have to minimize only single output functions.

The algorithm can be described in the following way :

```
NextInstructionGraph(State, NextStateList, PreviousCondition) {
  for (every NextState in the NextStateList) {
    if (ValidNextState(CurrentCondition, PreviousCondition)) {
      State->child = ListAppend(NextState);
      NextState->parent = ListAppend(State);
      UpdatedCondition = UpdateCondition(CurrentCondition);
      NextInstructionGraph(NextState, State, UpdatedCondition);
    } } }
```

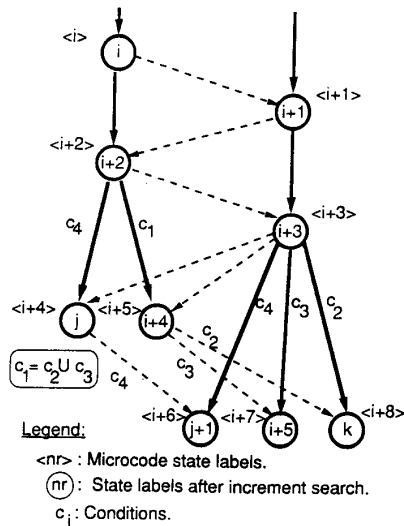


Figure 2: Microcode and next-instruction graph

The next-instructions are determined from the state that we are currently visiting (State) and the JMP specification of the previous state (NextStateList). The latter state description gives the value of CurrentCondition, which is the condition on that state transition in the next-next instruction graph. The results are stored in the child and parent fields of every state. As can be seen in the above algorithm, we include an edge in the next-instruction graph if the condition intersection of the previous condition and the condition on the next-next instruction specification of the microcode is not empty (function ValidNextState). Conditions have to be updated according to the UPD-specifications (function UpdateCondition). The next-instruction and the next-next instruction graph are known and one can decide where the incrementer can be used and when the address should be generated by the BFSM.

The second step will be to assign a code to every state taking into account that whenever an increment is possible we will choose that option. This is done by generating a spanning tree on the next-instruction graph by a depth-first search and then optimizing this tree (see figure 3). This means that whenever there is a branch in the spanning tree, we will try to assign the leafs of that state to another state in the tree, which does not yet have a successor in the spanning tree, and thus no increment. The algorithm is as follows :

```
OptimizeSpanningTree() {
  for (Every state with a branch in spanning tree) {
    find a state with a next-state as child and
    not yet having an arc in the spanning tree
    if found { replace_arc }
  }
}
```

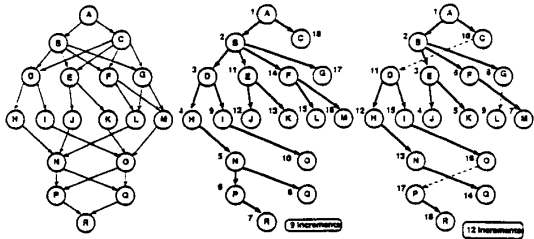


Figure 3: Spanning tree of next-instruction graph : optimisation

After the optimization, we can number the states according to the spanning tree in a depth-first search manner. The graph on the left in figure 3 shows the edges present in the next-state graph. The graph in the middle shows the spanning tree, before starting optimization. The graph on the right shows the result after optimization. The resulting state codes are also shown in figure 3. Edges are removed (e.g. from state b to d) and new edges are included (e.g. from c to d). The new edges are shown as dotted lines. After optimization the number of increments has increased from nine to twelve.

3.2 Optimization of JAF instruction codes

Another optimization performed by CGE is the minimization of the number of different JAF codes for the BFSM and SFSM. This means that CGE will determine how many different instructions each finite state machine has to perform and assign a unique code to every distinct operation. This will result in a minimal code length for the JAF's.

For the case of the SJAF, for example, the program will check if the update specification is larger, part of, smaller or equal to previously encountered updates. In this way a list can be built, containing a minimal number of different updates covering all the updates specified in the state descriptions of the controller. Every update set is then assigned a code and that code is used to build the logic description of the SFSM. For the BJAF, a similar procedure can be followed.

After finding the minimal number of JAF-codes, one should take care to assign to every instruction a code, that allows to minimize the logic implementation of the SFSM and BFSM. This problem is known in the literature as state assignment. In CGE, the codes for the JAF-fields are assigned with a KISSL-style strategy, see [DeM 85].

4 Examples

The CGE program is written in C and has been run on different designs, with varying complexities.

Table 1: Results of CGE on Apollo/DN4000.

Design	St	Edg	Eli	S	B	C	CPU
Filter	11	13	10	2	4	54	7
Ind1	12	19	13	5	6	23	7
Cordic	25	52	31	7	16	87	24
Ind2	32	47	31	9	12	78	18
Echo	99	156	86	17	53	77	41
Ind3	118	146	121	10	16	54	44
Speech	212	245	223	9	21	95	72
Ind4	227	257	219	10	31	47	65
Ind5	318	411	261	40	110	157	326

Table 1 gives the design name (names starting with Ind are industrial examples), number of states, number of state transitions, eliminated state transitions (i.e. implemented as increments), number of SJAF, BJAF codes and control signals. The last column gives the cpu-time of CGE on an APOLLO/DN4000 in seconds.

Ex. Ind1 from table 1 was generated with the CATHEDRAL-II silicon compiler in which CGE is integrated. The layout can be seen in figure 4. It realizes a four-tap FIR filter and uses two alu's. The multi-branch controller is shown in the top part of the figure.

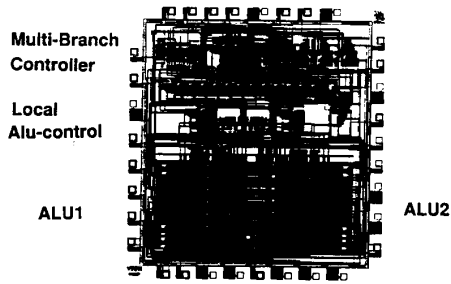


Figure 4: CATHEDRAL-II design example

5 Further research and optimizations

Sometimes it is possible to merge the BJAF and SJAF codes into one field, resulting in a decrease in the number of bits one has to store in the μ ROM. This will cause conflicts that will have to be taken into account, when assigning codes to SJAF and BJAF. The optimizations performed by CGE do not yet make any attempt to choose an optimal assignment of the state codes, BJAF and SJAF codes, aiming at multilevel logic implementation. This topic is also being investigated.

Some large design examples contain rather large SFSM and BFSM blocks. The combinatorial delay can become too large. This can partly be solved by multilevel logic implementations, but if PLA's are to be used, the CG-environment should attempt to partition the FSM's in an intelligent way.

The aim of CGE is to provide the designer with several alternative architectures from which (s)he can choose the most optimal one. The controller generation environment can therefore be seen as a toolbox of different state minimization and state assignment algorithms valid for several control architectures. Alternative architectures will be included in the future. These alternatives will be stripped versions of the multibranch controller of figure 1. In a later stage, different architectures will be included.

6 Conclusion

The CGE system that generates the logic, structure and layout descriptions of the multi-branch controller architecture shown in figure 1, from the microcode of the algorithm, has been described. It performs an automatic mapping of a microcode schedule, generated by Atomics, on a control architecture. The controller is optimized by optimal use of the incrementer and optimal instruction codes for FSM's.

It has been successfully integrated in the CATHEDRAL-II silicon compiler. Further improvements and optimizations (state assignment, other architectures) are being studied.

References

- [Bar 84] M. Bartholomeus et al. : "PLASCO : A procedural silicon compiler for PLA based systems", CICC 85, Portland, May 20-23, 1985, pp. 226-229
- [Bra 84] R. Brayton et al. : "Logic Minimisation Algorithms for VLSI Synthesis", Kluwer Publishing Co., Boston, 1984
- [Cat 88] F. Catthoor et al. : "Architectural Strategies for an Application-Specific Synchronous Multiprocessor Environment", IEEE Trans. on ASSP, Vol. 36, No. 2, Februari 1988, pp. 265-284.
- [DeC 87] I. De Cauwer and H. De Keulenaer : "Stuurlogica architecturen voor klantgerichte multiprocessor signaalverwerkingschips", Master's Thesis, KULeuven, 1987.
- [DeM 88] H. De Man et al. : "CATHEDRAL-II : A Synthesis System for Multiprocessor DSP Systems", in Silicon Compilation, Ed. by D. Gajski, Addison-Wesley Publishing Company Inc., 1988, pp. 311-360.
- [DeM 85] G. De Micheli : "Optimal State Assignment for Finite State Machines", IEEE Trans. on CAD, Vol. CAD-4, No. 3, July 1985, pp. 269-285.
- [DeM 87] G. De Micheli : "Synthesis of Control Systems" in Design Systems for VLSI Circuits (Logic Synthesis and Silicon Compilation), Ed. by G. De Micheli, A. Sangiovanni-Vincentelli and P. Antognetti, Martinus Nijhoff Publishers, pp. 327-364.
- [Goo 87] G. Goossens et al. : "An efficient microcode-compiler for custom DSP-processors", Proc. ICCAD, 1987, pp. 24-27.
- [Hil 85] P. Hilfinger : "Silage, a high-level language and silicon compiler for digital signal processing", Proc. IEEE-CICC 1985, Portland, May 1985, pp. 213-216.
- [Rij 87] L. Rijnders et al. : "Design of a process tolerant cell library for regular structures using symbolic layout and hierarchical compaction", Proc. ESSCIRC, 1987, pp. 197-200.
- [Six 86] P. Six et al. : "An intelligent module generator environment", Proc. 23rd ACM/IEEE DAC, Las Vegas, 1986, pp. 730-735.
- [VaH 87] J. Vanhoof et al. : "A knowledge-based CAD system for synthesis of multi-processor digital signal processing chips", Proc. VLSI 1987, Vancouver, August 1987.