1

# RETARGETABLE CODE GENERATION FOR PARALLEL, PIPELINED PROCESSOR STRUCTURES.

**Wolfgang Schenk**

*Computer Science Department*

*University of Dortmund*

*Germany*

## ABSTRACT

The demand for decreased turn around time in the design of programmable digital circuits requires CAD tools for synthesis, verification and code generation. Usually a RT level netlist is available as soon as the datapath is designed. Given the netlist and the behavior of the RT level modules, the proposed compiler maps a source program to the binary code of the target machine.

The main tasks of the compiler are allocation, register allocation, scheduling and compaction. These tasks are highly interdependent. Some machine features such as operator chaining, multi-cycle operations, pipeline latency, load delay, delayed branch, or residual control give raise to instruction dependencies, which can be automatically extracted from the structural description.

From the netlist the proposed compiler derives an internal target machine representation, that is general enough to support all target architecture features mentioned above.

In case the hardware supports different operators for a given operation the code generator must not commit to one of them, until a suitable alternative can be determined. In order to generate high quality code and to support irregular architectures, the code generator examines the alternative code versions.

## 1 INTRODUCTION

With a retargetable compiler it is easy to generate code for different target machines. The code generator has knowledge about the target machine features. There are different approaches where to specify the target machine:

1. The machine description is coded in the compiler.
2. There is an external description of the target machine, which is incorporated into the compiler by a pre-processing tool (e.g. a code-generator-generator).
3. The machine description is external and considered at compilation time.

The proposed compiler follows the 3rd approach, which is probably better called target independent compilation, because the compiler needs not to be recompiled for a new target. A machine description must include the operators of the target machine and the resources they are using in each cycle.

The machine model of the proposed compiler is a purely structural description of the modules in the machine and their interconnections. A processor description usually includes ALUs, registers, and memories as well as the instruction counter, the instruction memory and the instruction register. An instruction field, that specifies an operator is simply an interconnection from the instruction register to the control input of some ALU, either directly or via some other modules (e.g. decoder). An immediate operand (address) is a connection from the instruction register to the data (address) input of some module. Again this might be a direct connection or the operand may pass through some other modules. Consequently there is no notion for instruction formats. It is not needed, because all information is present in the netlist. Moreover there is no need to distinguish between modules belonging to the datapath and those which constitute the controller. Similarily, in this model there is no distinction between hardware for data and address calculation, because either one may serve the other purpose.

Based on a structural processor model the proposed compiler supports all of the following architectural features present in modern programmable processors:

- vertical and horizontal instruction encoding

- residual control
- pipelining with and without interlocks
- delayed branches, delayed load

As in classical compilers, the retargetable compiler also has to perform the code selection- and register-allocation tasks. Moreover, depending on the architecture features, the scheduling and compaction tasks may be required to improve the quality of the generated code.

The compiler accomplishes this by extracting the register transfers from the structural machine description. Operators, register transfers as well as instructions are represented in an unique data structure called SRU (set of resource usages). This data structure is consistently used for all of the compiler tasks.

In order to achieve high quality code, the compiler has to explore as much machine features as possible. The code selection task becomes a hard problem for complex instruction set processors. Sometimes not even the basic block boundaries are easily determined, due to conditional executable instructions [1]. If the target processor has a irregular register file (e.g. register windows, stacks, etc.) the register allocation task becomes more intricate because the access method may prevent the usage of a particular register at later time. If there are special purpose registers like address registers or stack pointers, the data routing task shall respect for the lack of available interconnect hardware. Nevertheless, the code generator should be able to use special purpose hardware.

For pipelined processors with or without interlocks the compiler is expected to anticipate pipeline hazards. The register allocation and scheduling task shall produce code sequences, that avoids unnecessary pipeline stalls. Good code quality for VLIW processors on the other hand requires a sophisticated allocation of partial instructions together with a global scheduling/compaction scheme. As the machine features affect the relative importance of the compiler tasks, the order in which the tasks are carried out is crucial as well.

This contribution is organized as follows. In the next section the most important compiler task are described and how they relate to each other. The phase coupling problem is discussed and the possible effect on the resulting code with respect to the target machine features. In the 3rd section closely related work on retargetable compilers in the MIMOLA software system is reviewed. These compilers already accept structural target machine specifications and take care of alternative code versions. In section 4 the style of the external description

**Figure 1** Compiler phases

## 2.1 Code Selection

Code selection is the task of mapping the source language statements to an intermediate representation, which contains operations only that are available in the instruction set of the target hardware. The code generator must be able to map the intermediate representation into target machine code in turn. Note that the term code selection is used in a different context elsewhere. Some authors regard the allocation task as part of the code selection, and others subsume code selection under the compaction task, which is considered to select from alternative instructions. Code selection for conventional processors is a straight forward task. How to map each source language construct to intermediate code is guided by simple rules. The intermediate code is usually represented as three address code or quadruples. Each statement of the intermediate code corresponds directly to a single assembly instruction. In case the target processor has multiple functional units like in VLIW processors, the statements of the intermediate code may map to more than one possible partial instructions. If a resource conflict hampers the compaction of a partial instruction the current code selection should be undone, whenever an alternative code compacts better. In the presence of multi-cycle instructions like in pipelined processors, resource conflicts may arise when succeeding instructions refer to the same resource. Although the intermediate code corresponds to a complete instruction, the code selection shall be undone unless the pipeline has interlocks to resolve such hazards.

## 2.2 Allocation

The allocation task maps the operations in the intermediate code to the resources in the target hardware. Variable allocation is the task of the global memory management instance, which is usually implemented as part of the code selection task. The stack layout as well as the references to structured variables are implicit in the intermediate code. Register allocation is the special case for the selection of storage resources for temporary values. The reverse mapping of variable- and register allocation is a subset of the state of the target hardware, which comprises the contents of all storage resources at a given execution point.

The allocation task is not an issue for conventional processors, where only a single functional unit is available. If there are multiple functional units then there may be alternative bindings for a given operation. The best binding is not determined until compaction time. Thus allocation should be backtrackable. Therefore, code selection and allocation should be carried out in a single phase. In the absence of multi-cycle instructions, the allocation of a functional unit to an operation may be implemented by allocation of instruction fields and residual control values, since all conflicts with respect to hardware components are mapped to conflicts at the control word (instruction conflict). Thus checking for resource conflicts can be replaced by checking for instruction conflicts. This observation was used in the compilers MSSV [7, 6] and MSSQ [11, 10]. The latter approach avoids backtracking by allocating all possible partial code versions for a statement. The code versions are stored in a special data structure called I-tree, which gives a concise representation and is well suited for checking for resource conflicts. At compaction time all versions for all statements are available and the best binding can be selected. If the processor has multi-cycle operations, the allocation becomes even more complicated. Replacing resource conflicts with instruction conflicts like in the case where only single cycle operations are present, is no longer valid, because resource conflicts may arise when succeeding instructions refer to the same resource. In order to keep track which resources are used during the execution of an operation, detailed knowledge about the architecture is necessary.

## 2.3 Scheduling

The scheduling task establishes an order of operations in time. Any order which satisfies the partial order implied by the data dependencies is valid. The critical path is the longest chain. The length is measured in the number of cycles needed to execute the operations along the path. The delay of an operation is known after the code generator has committed to a binding established in allocation. The critical path is not known until compaction, because then the delays that contribute to the length of the path can be distinguished from the delays that vanish, due to parallel execution.

## 2.4 Register Allocation

In register allocation it is determined which values are stored at which storage locations. The code generator makes use of the parallelism of the target architecture to avoid serialization of the code. This reduces the need for temporaries also. The life-span of a temporary does not exceed a basic block boundary. A temporary value is either inherent in the source program, or it is introduced by the code generator. Nevertheless the code generator has to introduce a temporary if there are insufficient resources to compile a statement into a single instruction. This is detected in the allocation phase, when there are not enough functional units or other intra-instruction conflicts to perform all operations of the statement. The introduction of a temporary updates the intermediate code with new read and write accesses to the temporary cell and the corresponding data dependencies. These accesses must also be allocated. Because it may be impossible to write the desired value directly to the newly assigned temporary location, it may be necessary to introduce another temporary cell. This is commonly known as the data routing problem. After allocation and compaction it can be told which copy of the value should be used and which data route fits best.

## 3 RELATED WORK

There are only few retargetable compiler known, that rely on structural hardware descriptions. The approach of Paulin et.al. [12], has a built-in processor model. For the register allocation and data routing tasks, the storage resources are classified into non disjunct register classes. The pattern matcher makes use of the dynamic programming concept from [2] (and [3, pp. 567–572]), which is considerably fast, but relies on costs, which are computed for each node of an expression tree. These costs must be uncertain, because it is not known in advance, how tight the associated code can be compacted.

A more recent approach is the CHESS compiler from Goossens et.al., which is also presented in this book. It uses a structural hardware description for the data path modules, and the nML-model from Fauth et.al. [5] for the specification of the instruction formats. In a hardware analysis phase partial code versions are attached to the operators found in the hardware, which are used to select from the possible instruction formats. But the allocation scheme does not provide multi-cycle operations, thus inter-instruction conflicts are hardly supported.

The design of the proposed compiler is based on the experience with earlier compilers MSSV [7, 6] and MSSQ [11, 10] of the MIMOLA software system [8]. Both compilers generate code for processor structures described in the MIMOLA language [4]. In the pre-processing stage, the high-level statements of the source program are mapped to register transfer statements: Variables are bound to storage cells, and high-level control constructs like loops and procedure calls are expanded to register transfer statements [7]. The compilers MSSV and MSSQ treat conditional statements in an unique way [9]. There may be different hardware features for implementing conditionals. Depending on which hardware support is detected, several versions of conditional statements are generated. Usually it is profitable to avoid jumps. But it is not known in advance, whether a conditional statement shall be compiled into a conditional jump. Since conditional expressions and conditional load operations may be implemented without conditional jumps, the different versions for conditional statements lead to different partitions into basic blocks. Each of the alternative basic blocks is allocated, and at compaction the best version is selected. This way the code selection task is done for complete basic blocks.

## 4 EXTERNAL MACHINE REPRESENTATION

The compiler relies on a detailed knowledge about the machine structure. The description contains the definition of data types, as well as the hardware parts and the interconnections among them. The parts are instantiations of modules from a component library. The modules in turn are described in their behavioral view, thus exposing the primitive operators of the hardware. The specification of the target machine allows a broad range of abstraction. The description may contain simple gates, ALUs, register, and memories as well as more complex modules. Complex modules may consist of operators and storages.

This way complete (sub-)systems can be specified. A pipelined ALU with input and output latches may be specified by a single module description. The ALU

```
TYPE Ctr= ENUM(mul,div); Data=(31:0); Word=(63:0);
MODULE Alu( IN c: Ctr; IN x, y: Data; OUT z: Word);
VAR x0, y0: Data; c0: Ctr;
CONBEGIN
   c0 := c; x0 := x; y0 := y;
   CASE c0  OF
     mul: z ← x0*y0;
     div: z ← x0/y0;
   END
CONEND
```

**Figure 2** Example of a pipelined ALU

has the input ports c,x,y, and the output port z. Ports define the signals, that are accessible outside the module. They are connected with nets, which are specified as sets of ports. A variable declaration defines the pipeline registers x0,y0,c0. Variables specify storage components. They are the only containers, that preserve their value across machine cycles. All storages (if enabled) are assumed to receive a new value synchronously at the end of the machine cycle. The behavior of the ALU is a single concurrent block. Each statement of a concurrent block is executed in every cycle. For instance, the concurrent block '**CONBEGIN** a:=b;b:=a **CONEND**' would exchange the values of a and b. The internal control logic of a module is represented by conditional statements (i.e. CASE-/IF-statement). The signal assignments to z in the ALU are mutually exclusively selected by the contents of the pipeline register c0. An operator may be described by an expression or a procedure call. Operators may have multiple outputs, which correspond to the output values of a procedure call. By default a signal assignment has a delay of 0 cycles. Multi-cycle operators would be specified as follows: The signal assignment 'c $\leftarrow$ a+b **AFTER** 1' introduces a one cycle delay for an adder until the output becomes valid. Note, that the arguments of an operator must be stable during all cycles in order to produce a valid result. Similarily, a setup time (in cycles) for a storage would be specified by an **AFTER**-construct attached to a variable assignment. In contrast, a delayed load must be specified with pipeline registers in front of the data input of the storage.

Informally the modules can be classified on how the data and control inputs are used.

- If a multi-cycle operator requires a control code, whose value is supplied in the first cycle of the activation only, the module is said to be of an *instruction issuing* type. Usually the control code is latched, and the latch register can be regarded as a residual control register. A floating point coprocessor is a typical example for this type of control.
- If the control input of the module must be constant across all the cycles of the activation, the module is said to be of the *fixed control* type. This is a common case when a controller has to make use of a slow operator with a delay larger than the cycle time.
- The most complex modules are of the *variant control* type. The control input has to receive different values in each cycle of the activation. Those modules are regarded as programmable devices, where the output is produced by applying a sequence of primitive operators on the arguments. The controller of a floating point unit for example makes use of this con-

trol type. The values must be normalized, calculated and rounded in sequence. For machine code generation these details can almost always be hidden from the code generator by replacing the variant controlled module with a compound operator of the instruction issuing type. Otherwise the code selection task shall provide the knowledge on how to decompose the operation to be compiled into the sequence of primitive operators.

The compiler generates binary code for the specified hardware structure. The user supplies the machine specification and tags the instruction register. Depending on whether the micro instruction register or the machine instruction register is specified, micro code or machine code is generated respectively. For machine code generation, of course the micro controller must be specified. A controller may be constituted from micro code (which may be given by ROMs and their contents), and decoder circuitry, and decoding logic, which is integrated into the behavioral description of data path modules. Micro code generation can even be used in the manual design and verification of processor structures [10]. In case the data path is already designed, the controller can be generated with the retargetable compiler.

## 5 INTERNAL MACHINE REPRESENTATION

The code generator uses an unique data structure, which resembles the module activation statements, known from the MIMOLA language in a more general sense. A module activation in MIMOLA specifies for each of the input ports a value. The activation determines the operator completely, because there are only simple, single-cycle operators, with a single output.

The module activation for issuing a specific operator can be generalized. This motivates the definition of SRUs.

### 5.1 Resources Usages

For each operator, the resources together with the values and relative cycles are collected in a set of resource usages (SRU). Formally a resource usage (RU) is a triple consisting, of a resource, a value, and an interval of integers. It denotes, that the resource is occupied by the value during the cycles indicated by the interval.

In general any hardware aspect, a code generator shall take care of, is a resource. A resource allowed in a RU may be a port or a variable. A port carries an input or an output signal of a part, and a variable represents a register or memory. Resources are typed, thus it can be determined, which bit fields are affected in the resource usage. Resources with structured types allow for more advanced features. A memory for example is described by a variable of an array type. If the memory has different access methods like word and byte addressing, it is described by a variable with a variant type, which is the union of different array types.

A value engaged in a resource usage, is a constant, a pattern or an operator, whose parameters are values in turn. The operator is characterized by its signature, which specifies the type of the parameters and whether they are used as input argument to the operator or as a result. There are four built-in operators for storage accesses. The unary *read* operator has the signature $read(OUT)$, for accessing a register, where the result argument yields the current contents of the register. For memory accesses the binary *read* operator with the signature $read(OUT, IN)$ is used, where the second argument specifies the memory address. If an operator has multiple outputs, the output actually used is selected by a projection operator. For single output operators, the projection may be omitted for readability. Similarily there are two *load* operators for write accesses to registers and memories, with the signatures $load(IN)$ and $load(IN, IN)$ respectively. The *load* operators are the only operators that yield no result. There is another special operator, which is heavily used in allocation. The *assert* operator ! with the signature $!(OUT, IN)$ implements the identity function on its input. It is used to distinguish a value, which is produced, from a value, which must be asserted, at the resource.

The timing is the third component of a RU. The interval indicates the first and last cycle at which the value is associated with the resource. For the description of the storage property, the last cycle of a *load* operator is usually infinite.

The SRUs constitute a very general data structure. SRUs are used to describe single operators, (micro-) operations, module activations, register transfers, and complete instructions. The code generator makes use of SRUs in hardware analysis, allocation, register allocation, scheduling and compaction.

The basic SRUs are extracted from the behavior module description. The extraction process relies on data flow information. A preceding data flow analysis yields the information about when the signals are produced and consumed.

According to the dataflow information the basic SRUs are combined to module activations or operator SRUs. In general the SRU for an operator contains RUs for the guarding conditions, the arguments, and the result. The control code of an operator is usually described within the module behavior using conditionals like IF- and CASE-statements for example. The result of the operator is associated with a signal resource or a storage resource. An assignment or reference to a variable introduces a *load* or *read* operator respectively.

Whenever an argument of an operator refers to a signal, it is represented by a pattern symbol. A pattern symbol is local to the SRU, and all occurrences of the pattern inside the SRU refer to the same signal.

For the pipelined ALU in fig. 2 the following basic SRUs are related to the activation of a multiplication:

$$\begin{Bmatrix}(Alu.c\ , & p1 & ,[1,1]\,) \\ (Alu.c0, & load(p1), & [1,\infty])\end{Bmatrix} \qquad \begin{Bmatrix}(Alu.x\ , & p1 & ,[1,1]\,) \\ (Alu.x0, & load(p1), & [1,\infty])\end{Bmatrix}$$

$$\begin{Bmatrix}(Alu.y\ , & p1 & ,[1,1]\,) \\ (Alu.y0, & load(p1), & [1,\infty])\end{Bmatrix} \qquad \begin{Bmatrix}(Alu.c0\ , & read(!mul), & [1,1]) \\ (Alu.x0, & read(p1) & ,[1,1]) \\ (Alu.y0, & read(p2) & ,[1,1]) \\ (Alu.z\ , & p1 * p2 & ,[1,1])\end{Bmatrix}$$

There are four concurrent SRUs. Three of them represent the assignments to the registers $c0$, $x0$, and $y0$. The first SRU consists of a RU for the signal resource $Alu.c$, which represents the input port $c$, and a RU for the storage resource $Alu.c0$. The *load* operator for a register $c0$ gets a value to be stored, which is represented by the pattern symbol $p1$. The timing is represented by the intervals. If a new value is supplied for a single cycle at the input port, the register assumes the new value, and stores it forever or until it will be overwritten. Since the scope of a pattern symbol is local to a SRU, it cannot interfer with any pattern symbol in a different SRU. In the SRU with the multiplication operator, the registers are accessed. The *read* operators at $Alu.x$ and $Alu.y$ produce the argument values, which are represented by the pattern symbols $p1$ and $p2$, for the multiplication, whose result is visible at the signal resource $Alu.z$. The value, read from the register $c0$, has to be asserted as the constant *mul* in order to use the SRU.

## 5.2 Extraction of Register Transfers

The hardware analysis phase provides all the information about the available operators and their usage to the code generator. The dataflow analysis and the extraction of the basic SRUs is carried out for the behavior of each module.

The code generator can do the allocation on basic SRUs. But for performance reasons basic SRUs are condensed as far as reasonable. Because the contents of the storages must be tracked, a reasonable granularity for allocation is at the register transfer level. A register transfer may occur within a module or beyond several modules, in which case the interconnections add the new dataflow information. But not all storages are relevant for code generation. Especially the pipeline registers and latches, present in pipelined processors, need not to be tracked, since they receive a new value in each clock cycle anyway. If hardware analysis determines a register, which is unconditionally loaded in each cycle, it is regarded as a pipeline register. With this knowledge it is possible to extend the register transfers beyond pipeline registers. The programable storage resources become the boundaries of the register transfers.

Starting at a *load* operator of a storage resource, which is no pipeline register, the register transfers are constructed. A second SRU is combined with the current one, if it contains a RU, for which the dataflow information indicates a supplied value at the resource, which is needed in the current SRU. During the combination of the SRUs, different pattern symbols for identical signals are merged into another. The data dependencies for signals must be maintained without storing it in a temporary. This is achieved if the interval of the supplied value covers the interval of its use. Thereby the setup- and hold times in the combined SRU are verified. When eliminating a pipeline register, the timing of the participating SRUs must be adjusted. The module activation of the pipelined ALU for the multiplication is:

$$\left\{\begin{array}{l}(Alu.c,\ !mul\ ,[1,1])\\(Alu.x,\ \ p1\ \ ,[1,1])\\(Alu.y,\ \ p2\ \ ,[1,1])\\(Alu.z, p1*p2,[2,2])\end{array}\right\}$$

Note that the pipeline registers $c0$, $x0$, and $y0$ have vanished. The timing has been adjusted accordingly.

## 6 THE CODE GENERATION ALGORITHM

The hardware analysis constructs the basic SRUs from the module behavior. For each operator of a module a basic SRU is created. Using the interconnections, the basic SRUs are in turn combined to register transfers. This process yields so called *unbound* SRUs, which serve as pattern for the allocation task, where the intermediate code is matched against those SRUs. The compiler uses trees as the intermediate representation of the program to be compiled. Whenever a node in the expression tree matches the value of a RU, an instance of the SRU is created, and the bindings from the expression to the RU and vice versa are established. These instances are called *bound* SRUs, and the expression is said to be partially allocated using the matching SRU. Register allocation uses the bindings established in allocation to determine, which values should reside in which storages and for checking for free storage resources when allocating temporaries. The compaction phase deals with bound SRUs only. The main task is checking for resource conflicts. It has to be assured, that no resource is used with different values at the same time.

The code generation algorithm proceeds through the phases allocation, register allocation, scheduling and compaction as depicted in fig. 3. In order to explore all possible code versions, it is fully backtrackable. There is no ordering among the unbound SRUs, and alternative matches can only established via backtracking. If there are no instruction versions without temporaries, storage locations are inserted tentatively at the input of the current instantiated SRUs. After register allocation, it is tested whether a valid schedule for the current allocated partial code version exists. If no schedule exists due to a dead-lock situation, the allocation must be backtracked. Furthermore a decision is made on whether spill code should be inserted before backtracking takes place. Once a suitable schedule is found, it can be used for compaction, which finishes the code generation for the current basic block.

### 6.1 Allocation

The allocation task establishes the bindings of the program operations to the operators in the hardware. Value tracking is an important subtask of allocation. For each value in the program it has to be determined, at which signals and storages it is present. In general a value must be routed via connections, multiplexors or other circuitry to the argument of an operator where it is used. Therefore it lives at different resources at the same time. The same is true for

```
FOR each basic block bb  DO
BEGIN
   WHILE not all nodes n in bb allocated  DO
     allocate n with some matching sru {backtrackable}
     create instance srui of sru and establish bindings
      FOR each RU ru in srui,
           whose value has to be asserted (!)  DO
         CASE resource(ru)  OF
           storage: {register allocation}
             IF not already there  THEN
               allocate temporary
               for value(ru) at resource(ru)  FI
           port:
            allocate value(ru)
            with some matching sru {backtrackable}
      END {case}
   END {while}
  schedule srui's
   IF schedule exist for srui's  THEN compact srui's
                                 ELSE backtrack allocation  FI
END {for}
```

**Figure 3** The code generation algorithm

different copies of a value, which may reside at different storage resources. The machine state, on the other hand, is the mapping from the storage resources to the values in the program.

The actual allocation is done for a total basic block at once. It is interleaved with the register allocation and scheduling phases, which may reject the current bindings and cause a reallocation of alternative bindings via backtracking.

The basic block is given as the set of statements in their intermediate representation together with the dataflow information. Spill code and other code for accessing temporaries may already have been inserted. Each assignment statement is represented by a single tree. The allocation algorithm does a depth first traversal through the tree. As each node is visited, it is allocated with a matching SRU. A node of a statement tree may be a variable access, an operation or a constant. If a variable is accessed, it may be bound or unbound depending on whether it is already associated with at least one storage or not. Write accesses to unbound variables are bound in advance to suitable storage locations. A write access for a bound variable matches with any SRU, that contains a RU with a *load* operator for one of the storages the variable is bound to. An instance of the matching SRU is created. The arguments in the expression tree become the instances of the arguments of the operator. Bindings which refer to patterns symbols are propagated through the SRU. They give rise to newly introduced assertions, which still have to be allocated. Whenever an assertion refers to a storage resource, the register allocation will try to take that storage as a temporary location, if it is not already there. If the assertion refers to an input port, the allocation recurs, trying to route the required value to that port. If there are other assertion operators in the SRU, they are allocated in turn. The allocation of other operations and constants proceeds accordingly. Anyway, beside *load*, *read*, !, and pass-through operators there is no semantic knowledge about the operators built into the allocation.

The allocation process stops in one of the following states:

1. Allocation succeeded, and leaves assertions for the instruction memory only.
2. Allocation succeeded, and leaves assertions for other storage resources.
3. Allocation failed, because no match was found.

In the first case, the assertions represent a code fragment ready for scheduling and compaction. In the second case register allocation takes place for the values to be asserted. The corresponding storage resources are assigned to the asserted values. In the last case the given program cannot be compiled onto the target machine due to missing operators or interconnections. The code generator issues an error message and exits.

## 6.2 Scheduling and Compaction

The data dependencies in the intermediate representation of the program imply the corresponding data dependencies between the bound RUs in the allocated SRUs. Although the data dependencies occur between RUs, the scheduling is carried out for SRUs. In this context a SRU can be regarded as an already compacted code fragment. This is because a SRU consists of RUs, which are fixed in their relative timing.

A valid schedule maintains data dependencies. It assures, that the live spans of different values that use the same resource are mutually exclusive.

Since register allocation is done tentatively, dead-locks may occur. A dead-lock is caused, when a value is written to a storage, and no ordering of the subsequent accesses to it constitute a valid schedule. Before backtracking is initiated, the decision is made whether or not spill code should be inserted for one of the participating values. Spill code is generated for temporaries only, because other variables are assigned to non-critical memories. No spilling takes place if this value is supposed to have alternative storage resources. For the most critical resource, the associated value with the fewest succeeding usages is selected for spilling. The selected value is bound to a new resource. If no such value exist, spill code is inserted.

Compaction is initiated for a totally allocated block only, after a valid schedule has been determined. The compaction algorithm preserves the order established during scheduling. Thus the compaction task has to pack the allocated SRUs as tight as possible together. Thereby data dependencies must not be violated, and no resource must be used with different values at the same time. Because several data dependencies may occur between two SRUs, and each of them is implied by storage accesses, the minimum distance between the SRUs is determined by the relative timing as recorded in the corresponding RUs. The first time the value is produced must not be later then the first time it is used in the dependent RU.

## 7 CONCLUSION

The compiler presented in this contribution has a target independent code generator, which relies on a purely structural hardware description. A wide range of possible target processors are covered. In addition to architectural features such as horizontal and vertical instruction encoding, pipelined and multi-cycle operators, the compiler automatically takes care of delayed branches, delayed load, residual control, data setup- and hold-time and other hardware properties. The code generation algorithm features an unique data structure (SRU) for the allocation, register allocation, scheduling and compaction tasks. High quality code is achieved by exploring different code versions via backtracking.

Thus most of the flaws from the sequential ordering of compiler phases are avoided.

## REFERENCES

[1] *ADSP2101/2102 User's Manual (Architecture), Analog Devices, 2nd Edition.* 1991.

[2] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[3] Alfred V. Aho, Ravi Sethi, and Jefferey D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[4] R. Beckmann, P. Marwedel, D. Pusch, and W. Schenk. The MIMOLA language reference manual – version 4.0 –, 2nd edition. Technical Report 401, Computer Science Department, University of Dortmund, February 1992.

[5] A. Fauth and A. Knoll. Translating signal flowcharts into microcode for custom digital signal processors. *Proc. ICSP*, 1993.

[6] P. Marwedel. Tree-based mapping of algorithms to predefined structures. *Int. Conf. on Computer-Aided Design*, 1993.

[7] Peter Marwedel. A retargetable compiler for a high-level microprogramming language. In *MICRO-17*, pages 267–274, New Orleans, Louisiana, October-November 1984.

[8] Peter Marwedel and Wolfgang Schenk. Cooperation of synthesis, retargetable code generation and testgeneration in the MSS. *EDAC-EUROASIC'93*, pages 63–69, 1993.

[9] Peter Marwedel and Wolfgang Schenk. Implementation of IF-statements in the TODOS-microarchitecture synthesis system. In *Synthesis for control dominated circuits*, pages 249–262. North-Holland, 1993.

[10] L. Nowak and P. Marwedel. Verification of hardware descriptions by retargetable code generation. In *Proc 26th Design Automation Conference*, pages 441–447, June 1989.

[11] Lothar Nowak. Graph based retargetable microcode compilation in the MIMOLA design system. *MICRO-20*, pages 126–132, 1987.

[12] Pierre G. Paulin, Clifford Liem, Trevor C. May, and Shailesh Sutarwala. CodeSyn: A retargetable code synthesis system. *Proc. 7th Int. Symp. on High-Level Synthesis*, 1994.

[13] Steven R. Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. *MICRO-15*, pages 125–133, 1982.