A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths

Mehrdad Reshadi, Daniel Gajski
Center for Embedded Computer Systems (CECS)
University of California Irvine
Irvine, CA 92697, USA
{reshadi, gajski}@cecs.uci.edu

Abstract

Traditional high level synthesis (HLS) techniques generate a datapath and controller for a given behavioral description. The growing wiring cost and delay of today technologies require aggressive optimizations, such as interconnect pipelining, that cannot be done after generating the datapath and without invalidating the schedule. On the other hand, the increasing manufacturing complexities demand approaches that favor design for manufacturability (DFM).

To address these problems we propose an approach in which the datapath of the architecture is fully allocated before scheduling and binding. We compile a C program directly to the datapath and generate the controller. We can support the entire ANSI C syntax because the datapath can be as complex as the datapath of a processor. Since there is no instruction abstraction in this architecture we call it No-Instruction-Set-Computer (NISC). As the first step towards realization of a NISC-based design flow, we present an algorithm that maps an application on a given datapath by performing scheduling and binding simultaneously. With this algorithm, we achieved up to 70% speedup on a NISC with a datapath similar to that of MIPS, compared to a MIPS gcc compiler. It also efficiently handles different datapath features such as pipelining, forwarding and multi-cycle units.

Categories and Subject Descriptors

B.5.2 [Register-Transfer-Level Implementation]: Design Aids---Automatic synthesis; C.1.3 [Processor Architectures]: Other Architecture Styles---Pipeline processors; B.5.1 [Register-Transfer-Level Implementation]: Design---Control design, Datapath design, Styles; D.3.4 [Programming Languages]: Processors---Code generation, Compilers, Retargetable compilers.

General Terms

Algorithms, Performance, Design, Standardization, Languages.

Keywords: NISC, scheduling, cycle-accurate compiler.

1. Introduction

Traditional High Level Synthesis (HLS) techniques take an abstract behavioral description and generate a register-transfer-level (RTL) datapath and controller. The generated datapath is in form of a netlist and must be converted to layout for the final physical implementation. Lack of access to layout information limits the accuracy and efficacy of design decisions (or optimizations) during synthesis. For example, applying interconnect pipelining technique is not easy during scheduling, because wire information is not available yet. It is not also possible to efficiently apply it after generating the datapath because it invalidates the schedule. The growing complexity of new manufacturing technologies demands synthesis techniques that support Design-For-Manufacturability (DFM). However, the interdependent scheduling, allocation and binding tasks in HLS are too complex by themselves and adding DFM will add another degree of complexity to the design process. This increasing complexity requires a design flow that provides a practical separation of concerns and supports more aggressive optimizations based on accurate information.

We believe that the best way to achieve this goal is to separate the generation of datapath and controller as shown in Figure 1. This new approach combines HLS, Application Specific Instruction set Processor (ASIP) design, and retargetable compiler techniques. First the datapath is designed and remains fixed during compilation, then the controller is generated by mapping (scheduling and binding) the application on the given datapath. In this way, DFM and other layout optimizations are handled independently from compilation/synthesis. Furthermore, accurate layout information can be used by scheduler.

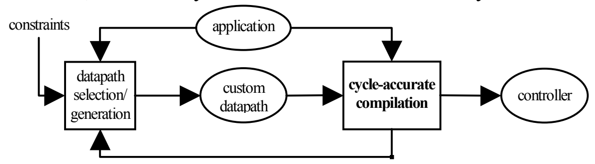

Figure 1- Proposed custom hardware design flow.

In some aspects, the proposed design flow is similar to the compilation of applications for processors because in both cases the datapath is fixed during the mapping process. However, compilers rely on instruction-set (or microcode) to abstract out the functionality of processor's datapath and assume that the processor translates such abstractions to proper control signals. In our approach, the *cycle-accurate compiler* directly maps the application on the given datapath by (1) binding operations, storages, and interconnects, and (2) scheduling the control signal values of datapath components in proper clock cycles. Therefore, it has complete fine-grain control over datapath and can achieve better parallelism and resource utilization. Since we do not use predefined instruction semantics, we call the result architecture No-Instruction-Set-Computer (NISC).

A NISC is composed of a pipelined datapath and a pipelined controller that drives the control signals of the datapath components in each clock cycle. The datapath can be generated in several ways. It can be selected from available IPs or reused from previous designs. It can also be designed manually or automatically using techniques from HLS or AISP design that analyze the application behavior and suggest a custom datapath. Such datapath can be iteratively refined and optimized as shown in Figure 1. The datapath can be simple or as complex as datapath of a processor. Figure 2 shows a sample NISC architecture with a memory based controller and a pipelined datapath that has partial data forwarding, multi-cycle and pipelined units, as well as data memory and register file. Depending on the required features (e.g. interrupt handling) the controller is selected from a set of predefined templates. The control values are stored in the control memory. For small applications, they can also be generated via logic.

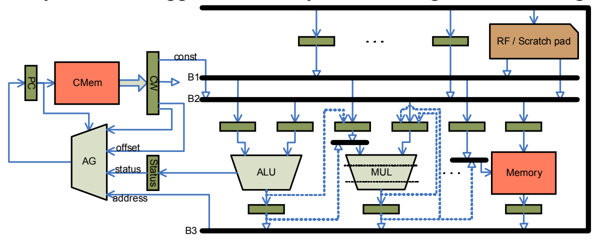

Figure 2- A sample NISC architecture.

The core of NISC design flow is the cycle-accurate compiler that maps the application directly on the given datapath. To show that such a design flow is feasible, in this paper, we present an algorithm that compiles the application by performing the scheduling and binding simultaneously. The paper is organized as follows: related works are reviewed in Section 2. In Section 3 we illustrate the algorithm using an example and then describe the details of the algorithm in Section 4. Various experiments and their results are shown in Section 5. Finally, Section 6 concludes the paper.

2. Related works

Because the architecture style of NISC is new, little research has been done on the mapping algorithms for NISC. However, there has been an extensive body of work on scheduling and binding algorithms in the area of high level synthesis, retargetable compilers.

Force directed scheduling (FDS) [1], [2] is commonly used to solve the timed constrained scheduling problem. This algorithm, distributes the execution of similar operations in different control steps in order to achieve high utilization of functional units while meeting the time deadline. Path-based scheduling algorithm [3] tries to minimize the number of control steps needed to execute the critical paths that exist in the given CDFG [4]. To do so, the algorithm gives emphasis to conditional branching i.e. it starts by extracting all possible execution paths from the given CDFG and schedules them independently. Then the schedules of different paths are combined to generate the final schedule for the whole design. However, the path-based approach restricts the execution order of the operations before scheduling.

List-based scheduling techniques [5] are used to solve resource constrained scheduling problem in which the number of resources of different types are limited. List scheduling processes each control step sequentially. At each control step, it tries to choose the best operation from the list of candidate operations, subject to resource constraints. List scheduling uses a ready-list, which keeps all nodes that their predecessors are already scheduled. The ready-list is always sorted with respect to a priority function. The priority function always resolves the resource contention among operations, i.e. operations with lower priority will be deferred to the next or later control steps. The quality of the results produced by a list-based scheduler depends predominantly on its priority function.

Mobility of the operation, i.e. the difference between ASAP (as soon as possible) and ALAP (as late as possible) times, is commonly used as the priority function in many HLS systems. Different priority functions and heuristics have been proposed to improve the quality of list scheduling. The proposed list scheduling algorithms in [6] and [7] uses mobility as the primary priority functions. To break the tie among a set of available operations with similar mobility, they assign higher priority to those operations that contribute to the same output. Before scheduling begins, they analyze the outputs of operations in the DFG by constructing a set of trees (cones) that start from output nodes as roots. However, they use a conventional scheduler that starts from inputs and proceeds forward, and the output trees are only used to break the tie during schedule. A similar approach is used in [8] and [9] for scheduling on VLIW architectures. Output trees in DFG are also used for instruction selection using the maximal-munch algorithm. Processing the DFG backward, from outputs towards inputs, has proven to be very fruitful. However, this idea has been mainly used in priority functions but not the scheduling algorithm itself.

Many researchers ([10], [11], [12], [13], [14]) have also attempted to incorporate layout information in the synthesis process, especially in scheduling. However, similar to traditional HLS, these approaches generate the datapath after scheduling and therefore they can only predict or estimate layout information during scheduling.

While most HLS techniques use list-based scheduling and perform allocation and binding separately, some approach, such as [15] and [16], try to perform scheduling, allocation and binding simultaneously using integer linear programming or branch-and-bound algorithms. Although they may achieve optimal results, complexity restrains the practical applicability of such approaches.

Getting a fixed architecture model as input is a common assumption in retargetable compilers, mostly used for Application Specific Instruction set Processors (ASIPs). But usually in these compilers the architecture model is described in terms of instructions, which is a much higher level of abstraction than the structural details of the architecture. Even compilers such as RECORD [17] and CHESS [19] that use a structural description of architecture, extract the higher level instruction information for using in the compiler. The RECORD compiler extracts behavioral model of instructions from MIMOLA HDL [18]. They assume a horizontal microcode machine with single cycle operation. They process the structure of the datapath from destination storages towards source storages to extract valid register transfers (RTs). After analyzing the controller, they reject illegal RTs that do not correspond to an instruction, and use the remaining RTs in the compiler. The CHESS compiler uses the nML language [20] to extract the instruction set graph (ISG) that captures structural resources in the architecture that are used by each instruction.

Regardless of the approaches, every compiler generates a stream of processor instructions and assumes that the processor itself deals with the control signals of its component. Since there is no instruction in NISC, the compiler directly maps the program to the datapath. In this way, compiler has complete fine-grain control over datapath and can achieve better parallelism and resource utilization. However, not only the compiler should generate the schedule, it should also generate the control values of architecture component in each cycle. Therefore, the NISC compiler must deal with much more structural details and solve a more complex problem than traditional processor compilers.

In all HLS approaches scheduling is done mainly based on the delay of functional units, while all or part of binding (especially interconnect binding) is done afterwards. This is not possible in NISC and scheduling and binding *must* be done simultaneously (see Section 3).

In the next section, we present an efficient simultaneous scheduling and binding algorithm that is inspired by the benefits of backward processing of DFG. It is very suitable for dealing with structural details of NISC datapaths. Nevertheless, the algorithm is very general and can be used in other domains as well.

3. Algorithm overview and illustrative example

In this section we illustrate the basis of our scheduling and binding algorithm using an example. The input of algorithm is the CDFG of application, netlist of datapath components and the clock period of system. The output is an FSM in which each state represents a set of register transfers actions (RTAs) that execute in one clock cycle. An

RTA can be either a data transfer through buses / multiplexers / registers, or an operation executed on a functional unit. The set of RTAs are later used to generate the control bits of components.

As opposed to traditional HSL, we can not schedule operations merely based on the delay of the functional units. The number of control steps between the schedule of an operation and its successor depends on both the binding of operations to functional units (FU) and the delay of the path between corresponding FUs. For example, suppose we want to map DFG of Figure 4 on datapath of Figure 5. Operation >> can read the result of operation + in two ways. If we schedule operation + on *U2* and store the result in register file *RF*, then operation >> must be scheduled on *U3* in next cycle to read the result from *RF* through bus *B2* and multiplexer *M2*. Operation >> can also be scheduled in the same cycle with operation + and read the result directly from *U2* through multiplexer *M2*. Therefore, selection of the path between *U2* and *U3* can directly affect the schedule. Since knowing the path delay between operations requires knowing the operation binding, the scheduling and binding must be performed simultaneously.

Binding itself involves three subtasks: *variable binding* assigns a value to a storage; *operation binding* assigns an operation to an FU; and *interconnect binding* selects a path between two FUs, or a storage and a FU. In our algorithm, these three subtasks are done during schedule of each operation.

The basic idea in the algorithm is to schedule an operation and all of its predecessors together. An *output operation* in the DFG of a basic block is an operation that does not have a successor in that basic block. We start from output operations and traverse the DFG backward. Each operation is scheduled after all its successors are scheduled. The scheduling and binding of successors of an operation determine when and where the result of that operation is needed. This information can be used for: utilizing available paths between FUs efficiently, avoiding unnecessary register file read/writes, chaining operations, etc.

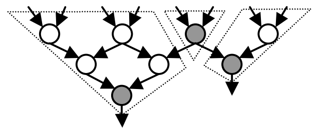

Figure 3- Partitioning a DFG into sub-trees.

We partition the DFG of the basic block into sub-trees. The root of a sub-tree is an output operation. The leaves are input variables, constants, or output operations from other basic blocks. If the successors of an operation belong to different sub-trees, then that operation is considered as an *internal output* and will have its own sub-tree. Such nodes are detected during scheduling. Figure 3 shows an example DFG that is partitioned into three sub-trees. The roots of the sub-trees are the output operations and are shown with dark nodes. The algorithm schedules each sub-tree separately. If during scheduling of the operations of a sub-tree, the schedule of an operation fails, then that operation is considered an internal output and becomes the root of a new sub-tree.

A sub-tree is available for schedule as soon as all successor of its root (output operation) are scheduled. Available sub-trees are ordered by the mobility of their root. The algorithm starts from output nodes and schedules backward toward their inputs, therefore more critical outputs tend to be generated towards end of the basic block (almost similar to ALAP schedule).

Consider the example DFG of Figure 4 to be mapped on the datapath of Figure 5. Assume that the clock period is 20 units and delays of *U1*, *U2*, *U3*, multiplexers and busses are 17, 7, 5, 1 and 3 units, respectively. We schedule the operations of basic block so that all results are available before last cycle, i.e. 0; therefore, the RTAs are scheduled in negative cycle numbers. In each step, we try to schedule the sub-trees that can generate their results before a given cycle *clk*. *clk* starts from 0 and is decremented in each step until all sub-trees of a basic block are scheduled.

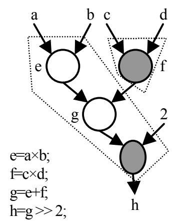

Figure 4-Sample DFG

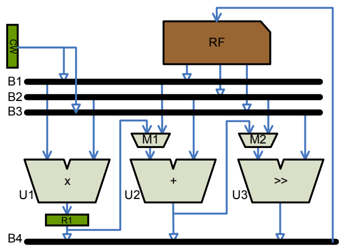

Figure 5-Sample datapath

During scheduling, different types of values may be bound to different types of storages (variable binding). For example, global variables may be bound to memory, local variables to stack or register file, and so on. A constant is bound to memory or control word (CW) register, depending on its size. A control word may have limited number of constant fields that are generated in each cycle along with the rest of control bits. These constant fields are loaded into the CW register and then transferred to a proper location in datapath. The NISC compiler determines what constant(s) should be generated in each cycle. It also schedules proper RTAs to transfer the value to where it is needed.

When scheduling an output sub-tree, first step is to know where the output is stored. In our example, assume *h* is bound to register file *RF*. We must schedule operation >> so that its result can be stored in destination *RF* in cycle -1 and be available for reading in cycle 0. We first select a FU that implements >> (operation binding) then make sure that a path exists between selected FU and destination *RF* and all elements of the path are available (not reserved by other operations) in cycle -1. In this example we select *U3* for >> and bus *B4* for transferring the results to *RF*. Resource reservation will be finalized if the schedule of operands also succeeds. The next step is to schedule proper RTAs in order to transfer the value of *g* to the left input port of *U3* and constant *2* to the right input port of *U3*. Figure 6 shows the status of schedule after scheduling the >> operation. The figure shows the set of RTAs that are scheduled in each cycle to read or generated a value. At this point *B3* and *M2* are considered the *destinations* to which values of *2* and *g* must be transferred in clock cycle -1, respectively.

clock→	-3	-2	-1
operation↓			
a			
b			
c			
d			
e			
f			
g			M2=?;
2			B3=?;
h			B4=U3(M2, B3); RF(h)=B4;

Figure 6- Schedule of RTAs after scheduling >> operation.

In order to read constant *2*, we need to put the value of CW register on bus *B3*. As for variable *g*, we schedule the + operation on *U2* to perform the addition and pass the result to *U3* though multiplexer *M2*. Note that delay of reading operands of + operation and executing it on *U2*, plus the delay of reading operands of >> operation and executing it on *U3* and writing the results to *RF* is less than one clock cycle. Therefore, all of the corresponding RTAs are scheduled together in clock cycle –1. The algorithm chains the operations in this way, whenever possible. The new status of scheduled RTAs is shown in Figure 7. In the next step, we should schedule the × operations to deliver their results to the input ports of *U2*.

Clock→ operation↓	-3	-2	-1
a			
b			
c			
d			
e			M1=?;
f			B2=?;
g			M2=U2(M1, B2);
2			B3=CW;
h			B4=U3(M2, B3); RF(h)=B4;

Figure 7- Schedule of RTAs after scheduling + operation.

The left operand (*e*) can be scheduled on *U1* to deliver its result though register *R1* in cycle –2 and multiplexer *M1* in cycle –1. At this point, there is no more multiplier left to generate the right operand (*f*) and directly transfer it to the right input port of *U2*. Therefore, we assume that *f* is stored in the register file and try to read it from there. If the read is successful, the corresponding × operation (*f*) is considered as an internal output and will be scheduled later. Figure 8 shows the status of schedule at this time. The sub-tree of output *h* is now completely scheduled and the resource reservations can be finalized.

Clock→ operation↓	-3	-2	-1
a			
b			
c		B1=RF©;	
d		B2=RF(d);	
e		R1=U1(B1, B2);	M1=R1;
f			B2=RF(f);
g			M2=U2(M1, B2);
2			B3=CW;
h			B4=U3(M2, B3); RF(h)=B4;

Figure 8- Schedule of RTAs after scheduling *h* sub-tree.

The sub-tree of internal output *f* must generate its result before cycle -1 where it is read and used by operation +. Therefore, the corresponding RTAs must be scheduled in or before clock cycle –2 and write the result in register file *RF*. The path from *U1* to *RF* goes through register *R1* and hence takes more than one cycle. The second part of the path (after *R1*) is scheduled in cycle –2 and the first part (before *R1*) as well as the execution of operation × on *U1* is scheduled in cycle –3. The complete schedule is shown in Figure 9.

clock→ operation↓	-3	-2	-1
a	B1=RF(a);		
b	B2=RF(b);		
c		B1=RF(c);	
d		B2=RF(d);	
e		R1=U1(B1, B2);	M1=R1;
f	R1=U1(B1, B2);	B4=R1; RF(f)=B4;	B2=RF(f);
g			M2=U2(M1, B2);
2			B3=CW;
h			B4=U3(M2, B3); RF(h)=B4;

Figure 9- Schedule of RTAs after scheduling all sub-trees.

In the above example, we showed how the DFG is partitioned into sub-trees during scheduling. We also showed how pipelining, operation chaining, and data forwarding are supported during scheduling of sub-trees.

4. Simultaneous scheduling and binding algorithm for custom pipelined datapaths

In this section we describe the main body of the algorithm that performs the scheduling and binding for each basic block of the application.

In the algorithm we use the following definitions:

For an operation *op*, *op.result* is the value generated by *op* and *op.operands* is the list of results of predecessors of *op*.

For a functional unit *FU*, *FU.output* is the output port of *FU* and *FU.inputs* is the set of input ports of *FU*. A functional unit may implement multiple operations. For each operation, *FU.timing* represents the delay of the unit (or its stages if it is pipelined) as well as the duration of applying the control signals to the unit.

A path *p* is the list of resources that can transfer a value from one point to another. These resources include busses, multiplexers and registers. The timing of resources of *p* is stored in *p.timings* and is calculated base on delay of buses or multiplexers, or setup time and read delay of registers or register-files.

A destination *dst* is a storage or an input port of a functional unit.

Each basic block as a schedule status *ss*, where *ss.RTAs*(*clk*) stores the set of scheduled RTAs in clock cycle *clk*, and *ss.resTable*(*clk*) stores the reservation status of resources in clock cycle *clk*.

In the *ScheduleBasicBlock* function (Figure 10), before scheduling the body of the basic block, the jump operation at the end of block is scheduled. If the controller is pipelined, then the branch delay of the jump operation must be filled by other operations in the block. Our algorithm schedules operations backward, i.e. from last operation in the basic block towards the first one. Scheduling jump before other operations guarantees that branch delay will be filled if resource constraints allows. In the main loop of *ScheduleBaiscBlock* function (lines 6-18) the *available* output operations, i.e. sub-tree roots that can generate their results at clock cycle *clk*, are collected and sorted based on a priority function, such as operation mobility. During scheduling of each of these output operations, some internal outputs may be generated. If the schedule of the operation is successful, then the operation is removed from sub-tree roots (*Roots*) and the newly generated internal outputs are added to the list in order to be processed later (lines 16-17). In each iteration of the loop, the *clk* is decreased and available output operations are collected and scheduled until all sub-trees in the block are processed. At the end, the sequence of control steps in *blk.ss* contains the exact schedule of RTAs that execute the basic block on the given datapath. After scheduling all basic blocks, the FSM of controller is generated by combining the sequence of control steps in each block based on the CDFG of the program.

```
00 ScheduleBasicBlock(block blk)
01    initialize the blk.ss schedule status;
02    if (blk has a jump operation)
03       ScheduleOperation(blk.jump, 0, blk.ss, PC);
04    Roots = {output operations in blk.DAG};
05    clk = 0;
06    while(Roots ≠ ∅)
07       AvailableOutputs = ∅;
08       foreach (operation op ∈ Roots)
09          if (all successor of op are scheduled after clock clk)
10             AvailableOutputs = AvailableOutputs + {op};
11       Sort AvailableOutputs by OperationPriorities;
12       foreach (operation op ∈ AvailableOutputs)
13          internalOutputs=∅;
14          bind op.result
15          destination dst = storage of op.result
16          if ( ScheduleOperation(op, clock ,blk.ss, dst))
17             Roots = Roots – {op} + internalOutputs;
18       clk=clk-1;
```

Figure 10- The ScheduleBasicBlock function

The *ScheduleOperation* function (Figure 11) tries to schedule an operation *op* so that its result is available at *dst* at clock cycle *clk*. The list of functional units that can execute *op* is stored in *F* and sorted by the UnitPriorities (line 2). This priority function depends on the delay of the unit as well as the paths from output of the unit to the destination *dst*. After selecting a functional unit *FU*, all paths from *FU* to *dst* are stored in *P* and sorted by a PathPriority. The timings of *FU* and a selected path *p* are calculated so that the output of *FU* is available at *dst* at clock cycle *clk* (lines 5-10). If *FU* and all of the resources on the path *p* are not reserved in the *ss.resTable* at the corresponding calculated times, then algorithm tries to schedule the operands of *op* by calling the *ScheduleOperands* function. If the schedule of operands succeeds, then selected functional unit *FU* and path *p* are reserved (operation and interconnect binding) (lines 13-17). We pass a copy of scheduling status (*copyStatus*) to function *ScheduleOperands* to make sure that original status changes only if all operands are successfully scheduled. If scheduling failed after trying all functional units, the *ScheduleOperation* function tries to bind the result of operation to a storage and schedule a read from that storage. If the read succeeds, the operation is added to the *internalOutputs* for later processing.

```
00 bool ScheduleOperation(operation op, clock clk, schedule status ss,
01 destination dst)
02    F= functional units that implement op sorted by UnitPriorities;
03    foreach(FU ∈ F)
04       P=paths from FU.output to dst sorted by PathPriorities;
05       foreach(p ∈ P)
06          p.timing.end=clock;
07          calculate p.timing.start;
08          if (resources of p are not reserved in ss.resTable)
09             FU.timing.end=p.timing.start;
10             calculate FU.timing.start;
11             if (FU is not reserved in ss.resTable)
12                copyStatus = ss;
13                if (ScheduleOperands(op, FU.timing.start, copyStatus, FU))
14                   ss=copyStatus;
15                   reserve FU and p in ss.resTable;
16                   add corresponding RTAs to ss.RTAs;
17                   return TRUE;
18    bind op.result;
19    if (ScheduleRead(op.result, clk, ss, dst));
20       internalOutputs = internalOutputs + {op};
21       return TRUE;
22    return FALSE;
```

Figure 11- The ScheduleOperation function.

The *ScheduleOperands* function (Figure 12) schedules the operands of an operation *op* on a selected functional unit *FU* so that their values are available on corresponding input ports of *FU* at clock cycle *clk*. If an operand is a variable or a constant, then this function tries to schedule a read from the corresponding storage. Otherwise, it calls the *ScheduleOperation* function. The function succeeds only if all operands can be scheduled.

```
00 bool ScheduleOperands(operation op, clock clk, schedule status ss, functional
01 unit FU)
02    foreach(operand o ∈ op.operands)
03       destination dst= FU.inputs corresponding to o;
04       if (o is a variable or a constant)
05          bind o to a storage;
06          if (! ScheduleRead(o, clk, ss, dst))
07             return FALSE;
08       else if (! ScheduleOperation(o, clk, ss, dst))
09             return FALSE;
10    return TRUE;
```

Figure 12- The ScheduleOperands function.

In the *ScheduleRead* function (Figure 13), the best available path that can transfer a value from its storage to the specified destination at clock cycle *clk* is selected and scheduled.

```
00 bool ScheduleRead(value v, clock clk, schedule status ss, destination dst)
01    P=paths from storage of v to dst sorted by PathPriorities
02    foreach(p ∈ P)
03       p.timing.end=clk;
04       calculate p.timing.start;
05       if (resources of p not reserved in ss.resTable)
06          reserve p in ss.resTable;
07          add corresponding RTAs to ss.RTAs
08          return TRUE;
09    return FALSE;
```

Figure 13- The ScheduleRead function.

5. Experiments

In this section we report preliminary results of implementing our algorithm in a NISC compiler that is being developed as part of the NISC based design tool set. The input to the compiler is the netlist of datapath components as well as the application written in ANSI C. To evaluate our algorithm we compiled a set of benchmarks on a set of architectures and evaluated the schedules. For benchmarks we used the *bdist2* function (from MPEG2 encoder), *DCT* 8x8, *FFT*, and a *sort* function (implementing the bubble sort algorithm). The *FFT* and *DCT* benchmarks have data independent control graphs. The *bdist2* benchmark works on a 16×h block and we used h=10 in our experiments. For the *sort* benchmark, we calculated the best case and worst case results for sorting 100 elements. Among these benchmarks, *FFT* has the most parallelism and *sort* is a fully sequential code. A demo of the tool and the details of benchmarks and architectures is available at [21].

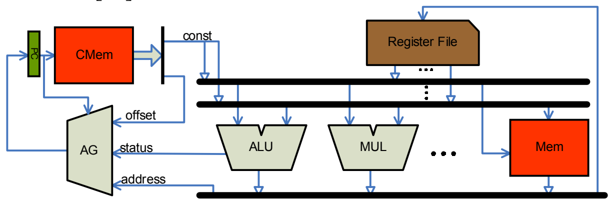

Figure 14- Datapath with simple interconnects.

To evaluate the effect of interconnects, we used a set of architectures that had the same number and type of functional units and storages but had different interconnect configuration. We started with an architecture with no pipelining (NP) similar to Figure 14. Then we added controller pipelining (CP) by adding *CW* and *status* registers in front of control memory and address generator (*AG*), respectively. We then added datapath pipelining (CDP) by adding registers to the input/output ports of functional units and data memory. At the end, we added data forwarding (CDPF) by adding interconnects from output of functional units to the input registers of other functional units. The final architecture is similar to what is shown in Figure 15.

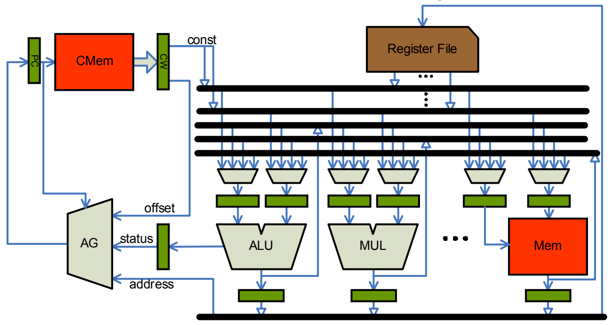

Figure 15- Datapath with complex interconnects.

We scheduled the benchmarks on the above datapaths and verified the results by simulating the generated Verilog files. We then synthesized the generated architectures on a Xilinx VirtexProII FPGA package using the Xilinx ISE tool set. After placement and routing, retiming, and buffer to multiplexer conversion; the tool reported 12.4, 8.9, 5.2, and 5.6 ns for the clock periods of NP, CP, CDP, and CDPF architectures, respectively. The number of execution cycles and total execution time of each benchmark on different architecture is shown in Table 1. While adding pipelining reduces the clock period, it may increase the cycle counts especially if there is not enough parallelism in the benchmark. Therefore, except for *FFT*, the cycle count of other benchmarks increases when we move from NP, to CP and CDP. However, the overall exaction time has improved in all cases, except in *sort* which is a fully sequential code. The considerable decrease of execution cycle counts from CDP to CDPF shows that the data forwarding paths between components are utilized well by our scheduling algorithm.

Table 1- Cycles, execution time and speedup of benchmarks.

	Cycle count				Total execution time (us)				Speedup vs. NP			
	NP	**CP**	**CDP**	**CDPF**	**NP**	**CP**	**CDP**	**CDPF**	**NP**	**CP**	**CDP**	**CDPF**
bdist2: block 16x10	6143	6326	7168	5226	76.2	56.3	37.3	29.3	1.00	1.35	2.04	2.60
DCT 8x8	10450	11764	14292	13140	129.6	104.6	74.4	73.6	1.00	1.24	1.74	1.76
FFT	219	220	218	166	2.7	2.0	1.1	0.9	1.00	1.35	2.45	3.00
Sort: Best case (N=100)	25447	35349	84161	74162	315.5	314.6	437.6	415.3	1.00	1.00	0.72	0.76
Sort: Worst case (N=100)	35149	49902	98714	88715	435.8	444.1	513.3	496.8	1.00	0.98	0.85	0.88

We also evaluated the schedule of benchmarks on two processor-like NISC architectures. The datapath of NM1 architecture is the same as a MIPS M4K Core [22]. The NM2 architecture extends the datapath of NM1 by adding one more ALU and 2 more register file read ports. Because of their similar datapath, the clock periods of these architectures are similar. The second, third and forth columns in Table 2 show the execution cycle counts of benchmarks on MIPS, NM1, and NM2, respectively. The last three columns show the corresponding speedups vs. MIPS. We used a gcc-based cross compiler to compile and optimize the benchmarks for MIPS. Note that although NM1 and MIPS have the same datapath, the benchmarks run up to 70% faster on NM1. The parallelism in NM1 (and MIPS) is limited by the number of register file read/write ports. However, our algorithm has well utilized the pipelining and data forwarding paths between components and achieved the speedup by avoiding accessing the register file. Our scheduling algorithm did utilize the extra resources in NM2 (especially for FFT) and the result was up to 100% faster than MIPS.

Table 2- Cycles and speedups on MIPS and MIPS-like NISCs.

	Cycle count			Speedup vs. MIPS		
	MIPS	**NM1**	**NM2**	**MIPS**	**NM1**	**NM2**
bdist2: block 16x10	6727	5204	4363	1.00	1.29	1.54
DCT 8x8	13058	10772	10644	1.00	1.21	1.23
FFT	277	162	133	1.00	1.71	2.08
Sort: Best case (N=100)	45642	40103	40004	1.00	1.14	1.14
Sort: Worst case (N=100)	50493	54656	54557	1.00	0.92	0.93

We neither used any optimization (such as loop unrolling) nor modified the source code of benchmarks to increase the parallelism. However, the results indicate that our compile utilizes the parallelism in the application and the datapath, and its results are comparable or better than that of a standard gcc-base compiler.

6. Conclusion

In this paper we presented (1) a design flow in which we map an application directly on a given datapath and generate the corresponding controller (2) and an algorithm for doing so.

Our approach is different from compiling for processors because in processors, the compiler uses the instruction (or microcode) abstraction to control the datapath and assumes that the processor translates instructions to control signals. In our approach, the architecture has no instruction abstraction and the cycle-accurate compiler must generate the control signals of datapath components in each clock cycle. We call this architecture No-Instruction-Set-Computer (NISC). A NISC compiler has complete fine-grain control over datapath and hence can achieve better parallelism and resource utilization.

The NISC approach is also different from traditional HLS because in HLS, the datapath and controller are generated **after** scheduling and binding; while in NISC, the datapath is available **before** scheduling and binding and the controller is generated afterwards. NISC simplifies DFM and use of IPs, and enables complete coverage of high level languages as well as more efficient and aggressive optimizations such as interconnect pipelining.

We also presented a compilation algorithm that traverses the DFG backward and performs scheduling and binding simultaneously. The algorithm inherently supports pipelining, data forwarding and operation chaining. Compared to a gcc-based MIPS compiler, our algorithm generates up to 70% faster results for the same datapath. Nevertheless, our algorithm is general and can be used in other domains.

7. References

[1] P. G. Paulin, J. Knight, "Algorithms for High-Level Synthesis", IEEE Design & Test of Computers, 1989.

[2] P. G. Paulin, J. P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", IEEE Transactions on Computer-Aided Design, 1989.

[3] R. Camposano, "Path-Based Scheduling for Synthesis", IEEE Transactions on Computer-Aided Design, 1991.

[4] A. Orailoglu and D.D. Gajski, "Flow graph representation", Design Automation Conference, 1986.

[5] D. Gajski, N. Dutt, A. Wu, S. Lin, "High-Level Synthesis Introduction to Chip and System Design", Kluwer Academic Publishers, The Netherlands, 1994.

[6] S. Govindarajan, R. Vemuri, "Cone-Based Clustering Heuristic for List-Scheduling Algorithms", Proceedings of European Design & Test Conference (ED&TC), 1997.

[7] A.M. Sllame, V. Drabek, "An efficient list-based scheduling algorithm for high-level synthesis", Proceedings of the Euromicro Symposium on Digital System Design, 2002.

[8] E. Ozer, S. Banerjia, "Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures", MICRO-31, 1998.

[9] J. R. Ellis, "Bulldog: A compiler for VLIW architectures", Cambridge, MA: The MIT Press, 1986.

[10] M. Xu, F. J. Kurdahi, "Layout-driven high level synthesis for FPGA based architectures", DATE, 1998.

[11] S. Y. Ohm, F. J. Kurdahi, N. Dutt, M. Xu, "A comprehensive estimation technique for high-level synthesis", International Symposium on Systems Synthesis, 1995.

[12] D. Kim, J. Jung, S. Lee, J. Jeon, K. Choi, "Behavior-to-placed RTL synthesis with performance-driven placement", International Conference Computer Aided Design, 2001.

[13] J. Zhu, D. Gajski, "Soft scheduling in high level synthesis", Design Automation Conference, 1999.

[14] W.E. Dougherty, D.E. Thomas, "Unifying behavioral synthesis and physical design", Design Automation Conference, 2000.

[15] B. Landwehr, P. Marwedel, and R. Dömer, "OSCAR:Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming", Proc. of European Conference on Design Automation, 1994.

[16] N. Berry, B.M. Pangrle, "SCHALLOC: an algorithm for simultaneous scheduling & connectivity binding in a datapath synthesis system", Design Automation Conference, 1990.

[17] R. Leupers, P. Marwedel, "Retargetable Generation of Code Selectors from HDL Processor Models", European Design and Test, 1997.

[18] P. Marwdedel, "The MIMOLA Design System: Tools for the Design of Digital Processors", Design Automation Conference, 1984.

[19] J. Van Praet, D. Lanneer, G. Goossens, W. Geurts, H. De Man, "A Graph Based Processor Model for Retargetable Code Generation", European Design and Test Conference, 1996.

[20] A. Fauth,J. Van Praet, M. Freericks, "Describing instruction set processors using nML", European Design and Test Conference, 1995.

[21] http://www.cecs.uci.edu/~reshadi/projects/nisc/

[22] MIPS32® M4K™ Core, http://www.mips.com