NISC Technology and Preliminary Results

Mehrdad Reshadi, Bita Gorjiara, Daniel Gajski

Technical Report CECS-05-11
August 24, 2005

Center for Embedded Computer Systems
University of California Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{reshadi, bgorjiar, gajski}@cecs.uci.edu

Abstract

A common way of improving an application's performance is implementing it on a custom hardware. High level synthesis (HLS) and application specific instruction-set processor are two alternatives for automating this process. HLS techniques usually can handle small programs. Also, since the datapath is not available during scheduling, limited datapath and layout optimizations (such as interconnect pipelining) are possible in HLS. On the other hand, to run an application on a custom hardware, ASIP approaches require the extra phases of designing custom instructions and implementing them in both instruction decoder and compiler.

In this report, we present theNISC approach where an application is directly compiled on a given datapath. The compiler does not use any instruction abstraction and directly generates the control signal values of datapath components in each clock cycle. The generated architecture is called No-Instruction-Set-Computer (NISC). In this report, we present the core compiler algorithm followed by two sets of experimental results, one focusing on architecture reuse and the other one on designing a custom architecture for a given application. Having a fine-grained control over the datapath, our compiler can generate up to 70% better performance compared to an instruction-set- based compiler according to our preliminary results. Also, by designing a custom datapath for DCT algorithm, we could achieve 7 times speedup, 1.64 times power reduction, 12.5 times energy savings, and more than 3 times area reduction compared to a soft-core MIPS implementation.

Contents

1. Introduction 1
2. Comparing NISC with other approaches 3
   - 2.1 NISC vs. HLS 3
   - 2.2 NISC vs. ASIP 4
   - 2.3 NISC vs. Microcoded architectures 4
3. NISC flow 5
4. Algorithm overview and illustrative example 5
5. Simultaneous scheduling and binding algorithm for custom pipelined datapaths 9
   - 5.1 Scheduling the CFG of the program 9
   - 5.2 Scheduling the DFG of the program 10
6. Experiments 12
   - 6.1 Reusing datapath for different benchmarks 13
   - 6.2 Case study: DCT implementation 15
     - 6.2.1 Implementing DCT using general-purpose datapaths 15
     - 6.2.2 Designing a custom hardware for DCT 17
       - 6.2.2.1 Software transformations 17
       - 6.2.2.2 Initial Custom datapath: CDCT1 18
       - 6.2.2.3 CDCT2: Bus customization and adding a pipeline register to the datapath 19
       - 6.2.2.4 CDCT3: Eliminating the unused parts of ALU, comparator and RF 20
       - 6.2.2.5 CDCT4 and CDCT5: Controller pipelining 20
       - 6.2.2.6 CDCT6: bit-width reduction 21
     - 6.2.3 Comparing performance, power, energy and area of the NISCs 22
     - 6.2.4 Comparison with a manual design 24
7. Related works 25
8. Conclusion 26
9. Acknowledgements 26
10. References 27

List of figures

Figure 1- NISC flow........ 2
Figure 2- A sample NISC architecture........ 2
Figure 3- NISC based design flow........ 5
Figure 4- Partitioning a DFG into output sub-trees........ 6
Figure 5- (a) Sample DFG, (b) Sample datapath........ 7
Figure 6- Schedule of RTAs after scheduling >> operation........ 7
Figure 7- Schedule of RTAs after scheduling + operation........ 8
Figure 8- Schedule of RTAs after scheduling *h* sub-tree........ 8
Figure 9- Schedule of RTAs after scheduling all sub-trees........ 9
Figure 10- The ScheduleFunction procedure........ 10
Figure 11- The ScheduleBasicBlock procedure........ 11
Figure 12- The ScheduleOperation function........ 11
Figure 13- The ScheduleOperands function........ 12
Figure 14- The ScheduleRead function........ 12
Figure 15- Datapath with simple interconnects........ 13
Figure 16- Datapath with complex interconnects........ 13
Figure 19- C-Code of matrix multiplication........ 15
Figure 20- Block diagram of GPD........ 16
Figure 21- C-code of unrolled matrix multiplication........ 17
Figure 22- Transformed matrix multiplication C-code........ 18
Figure 23- Block diagram of CDCT1........ 18
Figure 24- Block diagram of CDCT2........ 20
Figure 25- Block diagram of CDCT6........ 21
Figure 26- Power breakdown of the DCT implementations........ 23
Figure 27- Comparing different DCT implementations........ 24
Figure 28- Comparing CDCT6 with a commercial manual design [29]........ 24

List of tables

Table 1- Execution cycles counts of benchmarks. 14
Table 2- Execution cycle counts and speedups on MIPS and MIPS-like NISCs.................................. 14
Table 3- Clock period of architectures after synthesis. 14
Table 4- Execution delay (us) of benchmarks and speedup vs. NP. 14
Table 6- ALU and Comparator operations.. 16
Table 7- Critical-path delay breakdown of CDCT1. 19
Table 8- Critical-path delay breakdown of CDCT2. 20
Table 9- Critical-path delay breakdown of CDCT3. 20
Table 10- Critical-path delay breakdown of CDCT4. 21
Table 11- Critical-path delay breakdown of CDCT5. 21
Table 12- Summary of the experiments. 22
Table 13- Performance, power, energy, and area of the DCT implementations. 22

An Algorithm for Compiling Applications on Custom Pipelined Datapaths and Synthesizing the Controller

Mehrdad Reshadi, Bita Gorjiara, Daniel Gajski
Center for Embedded Computer Systems
University of California Irvine, CA, 92697
{reshadi, bgorjiar, gajski}@cecs.uci.edu

Abstract

A common way of improving an application's performance is implementing it on a custom hardware. High level synthesis (HLS) and application specific instruction-set processor are two alternatives for automating this process. HLS techniques usually can handle small programs. Also, since the datapath is not available during scheduling, limited datapath and layout optimizations (such as interconnect pipelining) are possible in HLS. On the other hand, to run an application on a custom hardware, ASIP approaches require the extra phases of designing custom instructions and implementing them in both instruction decoder and compiler.

In this report, we present theNISC approach where an application is directly compiled on a given datapath. The compiler does not use any instruction abstraction and directly generates the control signal values of datapath components in each clock cycle. The generated architecture is called No-Instruction-Set-Computer (NISC). In this report, we present the core compiler algorithm followed by two sets of experimental results, one focusing on architecture reuse and the other one on designing a custom architecture for a given application. Having a fine-grained control over the datapath, our compiler can generate up to 70% better performance compared to an instruction-se- based compiler according to our preliminary results. Also, by designing a custom datapath for DCT algorithm, we could achieve 7 times speedup, 1.64 times power reduction, 12.5 times energy savings, and more than 3 times area reduction compared to a soft-core MIPS implementation.

1. Introduction

A common way of improving the performance of applications in embedded systems and SOCs is to implement them in custom hardware. Manual design of Application Specific Integrated Circuits (ASIC) is no longer practical due to the increasing size and complexity of applications and the shrinking time-to-market. High Level Synthesis (HLS) and Application Specific Instruction set Processors (ASIPs) are commonly used to automate the process of improving application performance. However, these techniques have limitations that restrict their usage and their effectiveness. HLS techniques support a subset of a programming language and can handle relatively small size programs. Additionally, since the final datapath is not available during synthesis, design decisions and optimizations are limited to estimations of final physical attributes. On the other hand, ASIPs rely on limited number of instruction extensions for improving the performance. Identifying the beneficial instructions, implementing an efficient instruction decoder, and incorporating the new instructions in the compiler are complex tasks and require special expertise; which may affect the time to market of these processors.

In this paper, we propose a new approach that combines some of the techniques of HLS and ASIP to overcome their limitations and provide an easy and efficient way of compiling programs on custom

pipelined datapaths. In our approach, we map a program (written in a high level language such as C) directly on the datapath and generate a controller that executes the program on the given datapath. We do not use predefined instruction semantics. Instead we have complete fine-grained control over datapath. Hence, we can achieve better parallelism and resource utilization. We call this architecture No-Instruction-Set-Computer (NISC). Figure 1 shows the overall flow of this process.

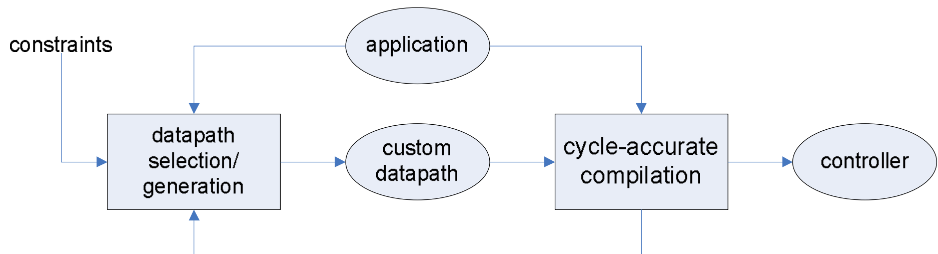

Figure 1- NISC flow

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 control values are stored in a control memory. For small size programs, the control values are generated via logic in the controller. The datapath of NISC 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.

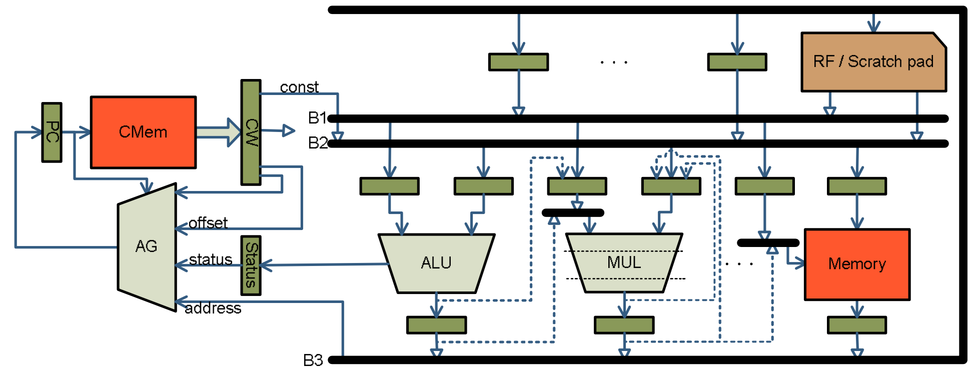

Figure 2- A sample NISC architecture.

In Figure 1, the custom datapath contains information such as clock period, net-list of components and their timings. The *cycle-accurate compiler* analyzes the datapath to determine what operations are possible and how data must flow between them. The core of this compiler is a scheduling and binding algorithm that maps the application on the given datapath and generates the control signals of components in every clock cycle. This algorithm must be efficient and impose little limitations on the structure of the input datapath. In the rest of this paper, we first compare NISC with other approaches and explain the NISC flow in more details; then we present an algorithm that performs the scheduling and binding simultaneously and maps an application onto a given NISC architecture.

This paper is organized as follows: Section 2 compares NISC with alternative approaches. Section 3 explains the NISC design flow. Section 4 illustrates the core algorithm of the compiler using an example. We describe the details of the algorithm in Section 5. The results of various experiments are shown in Section 6. Section 7 reviews related works and Section 8 concludes the paper.

2. Comparing NISC with other approaches

2.1 NISC vs. HLS

Traditional High Level Synthesis (HLS) techniques take an abstract behavioral description of a digital system and generate a register-transfer-level (RTL) datapath and controller. In these approaches, after scheduling, the components are connected (during binding) to generate the final datapath. 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 especially during scheduling. Since the generation of datapath and controller are tightly coupled, only those optimizations on the final datapath or layout are possible that do not invalidate the schedule.

Today, wiring and interconnect contribute to a significant portion of cost and delay of the circuit. The wire attributes depend on the technology as well as layout geometry (wire shape, length, fan out etc). An important technique for reducing the clock period of a design is pipelining the interconnects. Applying this 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.

There have been many attempts in the past to predict or estimate the physical attributes of the final datapath layout. Not only these attempts lack the necessary accuracy, they also lead to more complex allocation and scheduling algorithms.

The growing complexity of new manufacturing technologies demands synthesis techniques that favor 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.

In NISC, this goal is achieved by performing the scheduling and binding on a fully allocated datapath. The major benefits of this approach include:

- Design and manufacturing constraints can be easily applied to the datapath and its layout without modifying or complicating the scheduling algorithm.
- The timing information used by scheduling can be very accurate because the layout of datapath is available.
- The datapath can be optimized for area, power, speed, etc. without worrying about invalidating a schedule.
- The datapath can be selected from a pre-laid out and optimized set of IPs. Therefore the same way that standard cells simplified ASIC design, predefined datapaths can simplify system synthesis.
- It is possible to use traditional HLS techniques along with the application characteristics to generate an initial datapath. Then we can iteratively schedule the application and refine the datapath by adding/removing components or interconnects to improve the clock period, power consumption, area etc., and meet the constraints.

We can support all features of high level languages such as C by adding proper components and structures to the datapath. Therefore, we can target a broader application domain. Traditional HLS techniques could seldom achieve such language coverage.

2.2 NISC vs. ASIP

Compiling an application to a customized datapath has also been the goal of retargetable compilers and application specific instruction set processors (ASIPs) [23]. In these approaches, the compiler uses the high level instruction abstractions to indirectly control the datapath. It assumes that the processor already has a controller that translates the instructions into proper control signals for the datapath components.

In ASIPs, functionality and structure of datapath can be customized for an application through custom instructions. At run time, each custom instruction is decoded and executed by the corresponding custom hardware. Due to practical constrains on size and complexity of instruction decoder and custom hardware, only few custom instructions can be actually implemented in ASIPs. Therefore, only the most frequent or beneficial custom instructions are selected and implemented. In other words, the goal is to implement a custom hardware that executes critical portions of the program. In ASIPs, this requires several intermediate steps, i.e. designing custom instructions, implementing an efficient instruction decoder, and incorporating the new instructions in the compiler. These steps are complex and usually time consuming tasks that require special expertise.

In NISC, the instruction-set related intermediate phases are completely removed and the program is directly mapped to the datapath. If the datapath is designed to improve the execution of certain portions of program, the NISC compiler will automatically utilize it. Since the compiler is no longer limited by the fixed semantics of instructions, it can fully exploit datapath capabilities and achieve better parallelism and resource utilization.

2.3 NISC vs. Microcoded architectures

In horizontally microcoded architectures, a set of very simple operations in each cycle determines the behavior of datapath in that cycle. To enable automatic extraction of microcodes from datapath and using them in a compiler, the microcodes must execute in single cycle and have meaningful execution semantics for the compiler. However, consider operations such as *load* (Memory Read) or *pipelined multiplication*. These microcodes must be translated to multiple control values in different cycles. For example, during load, the *ChipSelect* control signal of memory must become 1 at some time (cycle) and then become 0 at some other time (cycle). Similarly, a pipelined multiply corresponds to events, i.e. multiple control values in multiple cycles. Note that, these events cannot be associated with different microcodes since such microcodes would not have meaningful execution semantics for compiler. Therefore, the microcode operations may require simple decoder in the architecture that translates them to actual control values. In fact, in most microcoded machines, the compiler only deals with microcodes and the structural details of datapath are often hidden from compiler.

In contrast, in NISC, each low-level action (such as accessing storages, transferring data through busses/multiplexers, and executing operations) is associated with a simple timing diagram that determines the values of corresponding control signals at different times. The NISC compiler eventually schedules these control values based on their timings and the given clock period of the system. Therefore, although a NISC may look similar to a horizontally microcoded architecture, the NISC compiler has much more low-level control over the datapath and hence is closer to a synthesis tool in terms of capability and complexity.

Nevertheless, because of similarities between NISC and a statically scheduled horizontally microcoded architecture, the presented algorithm in this paper can be used for compiling microcode operations as well.

3. NISC flow

Figure 3 shows a possible NISC based design flow for implementing an application on a custom hardware. In NISC, the controller is generated after compiling the application on a given datapath. The datapath can be generated (allocated) using different techniques. For example, it can be an IP, reused form other designs, generated by HLS, or specified by the designer. In our current implementation, we use an XML file to capture the netlist of components in the datapath. A component can be a register, register-file, bus, multiplexer, functional unit, memory etc. The functionalities of components are associated with timing information of corresponding control values. The program, written in a high level language such as C, is first compiled and optimized by a front-end and then mapped (scheduled and bound) on the given datapath. The compiler generates the stream of control values as well as the contents of data memory. The generated results and datapath information are translated to a synthesizable RTL design, described in Verilog, that is used for simulation (validation), synthesis (implementation), etc.

At any point, after scheduling down to the implementation, the results can be analyzed, and the design can be improved by refining the datapath and recompiling the program on the refined datapath. This kind of flow is possible only if we can compile an application directly to a given datapath.

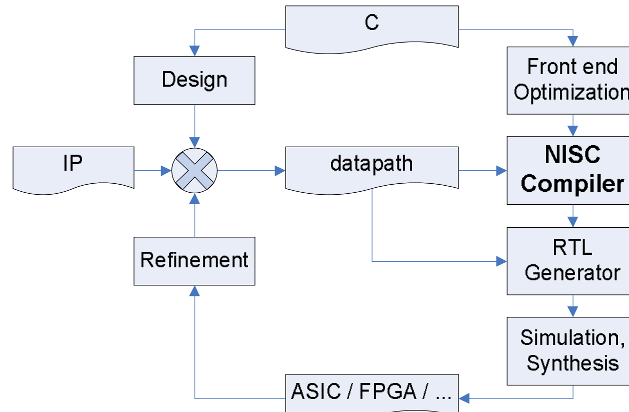

Figure 3- Possible NISC based design flow.

Since the NISC compiler compiles the application directly to the datapath, it can achieve better parallelism and resource utilization than conventional instruction-set based compilers. However, NISC compiler must solve a more complex problem because it must deal with all structural details of the datapath. The core of this compiler is a scheduling and binding algorithm that links the operations of the application to the clock cycles and datapath resources. The algorithm has to perform scheduling and binding simultaneously (see Section 4) and support features such as datapath/controller pipelining, data forwarding, operation chaining, multi-cycle and pipelined units. In the rest of this paper, we present one such algorithm. It processes the data flow graph (DFG) of basic blocks backward and generates a finite state machine (FSM) that executes the Control Data Flow Graph (CDFG) [4] of the program on a given datapath. The variable, operation, and interconnect bindings are performed during the schedule of each operation. We also allow pre-binding of variables and operations so that the designer or other algorithms can control the results. For examples, a partitioning algorithm may partition the variables and pre-bind them to two memory units.

4. Overview of the compiler algorithm by an 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 5(a) on datapath of Figure 5(b). Operation shift-left (>>) 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.

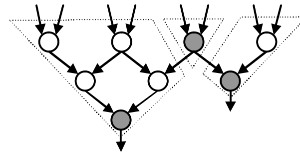

Figure 4- Partitioning a DFG into output sub-trees.

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 4 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 shaded 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 the end of the basic block (almost similar to ALAP schedule).

Consider the example DFG of Figure 5(a) to be mapped on the datapath of Figure 5(b). 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*. The *clk* starts from 0 and is decremented in each step until all sub-trees of a basic block are scheduled.

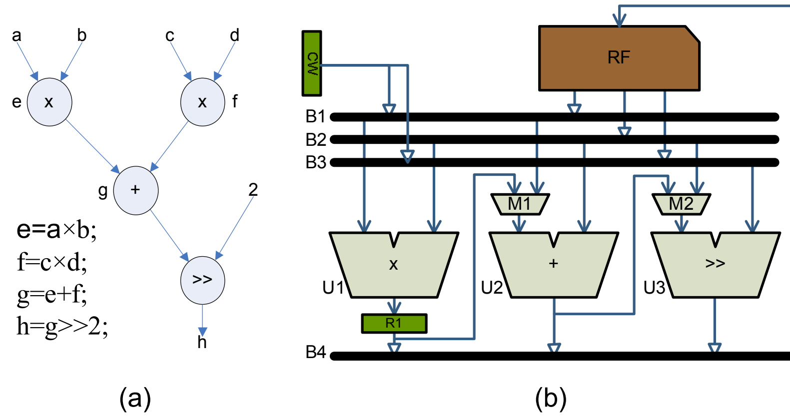

Figure 5- (a) Sample DFG, (b) Sample datapath.

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 the values of constant(s) in each cycle. It also schedules proper set of RTAs to transfer the value(s) 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 we 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 (interconnect binding). 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→ operation↓	-3	-2	-1
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 through register *R1* in cycle -2 and multiplexer *M1* in cycle -1. At this point, no other multiplier is 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(c);	
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.

5. Simultaneous scheduling and binding algorithm for custom pipelined datapaths

In this section we describe our algorithm for compiling the application to a custom datapath. When compiling the CDFG of each function of a program, we must consider the structure of the controller for compiling the control flow graph (CFG) and consider the structure of datapath for compiling the DFG. This process is described in the next two subsections.

In the description of the algorithm, we use the following definitions:

- Each basic block has 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*, and *ss.length* shows the number of scheduled states for that block.
- 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.

5.1 Scheduling the CFG of the program

The result of NISC compiler is an FSM that can be implemented in logic or using a memory. In a memory-based implementation the state register is a program counter register (PC). Therefore, a state change in the FSM corresponds to incrementing the PC or loading it with a new value using a jump operation. While incrementing PC always takes one cycle, loading it with a new value may take more than one cycle. The result of scheduling a basic block is always a sequence of states. We may only need a jump at the end of a basic block, if the last state of the block is not before the first state of the next

basic block. In the algorithm, we assume that the order of basic blocks is given, and that there may be jump operation at the end of some basic blocks.

Since we perform the scheduling backward, the result will be a set of states numbered from *–N* to *+bd*. The return address of a function is loaded into PC at state 0. Constant *bd* is the branch delay of the architecture, i.e. in a basic block, after loading the target address of a jump operation into PC, *bd* more control words will be executed from that basic block. Value of *bd* depends on the distance between *PC* and control word register, which is fixed and unique. Usually, this delay is 0 or 1 cycle in NISC.

In procedure *ScheduleFunction* (Figure 10), the *blkList* contains the topologically ordered list of basic blocks where the last element of the list is the *return block*. The blocks of this list are processed in reverse, starting from return block and after scheduling each block, the results are added to the *fsm*. In the main loop of *ScheduleFunction* (lines 3-8), before scheduling the body of a basic block, the jump operation at the end of block is scheduled. The same way that a + operation is mapped to an adder or ALU and writes its results to a register or register file, the jump is considered an operation that is mapped to address generator and writes its result to the PC register in cycle *clk*. In this way, we can schedule jump the same way that we schedule other operations (line 5). In order to make sure that the branch delay of the jump operation is filled by other operations in the basic block, we try to schedule the DFG of the basic block from cycle *clk+bd* (line 6). After scheduling each basic block, the new value of *clk* is calculated by decrementing the number of states that was added in the block (line 7). The *ScheduleBasicBlock* and *ScheduleOpertion* functions are described in Section 5.2. After scheduling all functions of a program, *fsm* will contain the final FSM of the design.

```
00 ScheduleFunction(FSM fsm, ordered list of basic blocks blkList)
01   clk = 0;
02   bd = branch delay;
03   foreach (blk ∈ reverse of blkList)
04     if (blk has a jump operation)
05       ScheduleOperation(blk.jump, clk, blk.ss, PC);
06     ScheduleBasicBlock(blk, clk+bd);
07     add blk.ss states to fsm;
08     clk = clk – blk.ss.length;
```

Figure 10- The ScheduleFunction procedure.

5.2 Scheduling the DFG of the program

Figure 11 shows the *ScheduleBasicBlock* procedure that performs the scheduling and binding for each basic block of a CDFG. In the main loop of this function (lines 3-16) 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. The algorithm tries to schedule as many of these operations as possible at clock cycle *clk*. 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 the set of sub-tree roots (*Roots*) and the newly generated internal outputs are added to the list in order to be processed later (lines 14-15). In each iteration of the loop, the *clk* is decremented and available output operations are collected and scheduled until all sub-trees in the block are processed.

```
00ScheduleBasicBlock(block blk, clock lastClock)
01  Roots = {output operations in blk.DAG};
02  clk = lastClock;
03  while(Roots ≠ ∅)
04    AvailableOutputs = ∅;
05    foreach (operation op ∈ Roots)
06      if (all successor of op are scheduled after clock clk)
07        AvailableOutputs = AvailableOutputs + {op};
08    Sort AvailableOutputs by OperationPriorities;
09    foreach (operation op ∈ AvailableOutputs)
10      internalOutputs=∅;
11      if (op.result is not pre-bound to a storage)
12        bind op.result
13      destination dst = storage of op.result
14      if ( ScheduleOperation(op, clock ,blk.ss, dst))
15        Roots = Roots – {op} + internalOutputs;
16    clk=clk-1;
```

Figure 11- The ScheduleBasicBlock procedure.

```
00bool ScheduleOperation(operation op, clock clk, schedule status ss, destination dst)
01  if (op is pre-bound to a functional unit)
02    F = functional unit to which op is pre-bound;
03  else
04    F = functional units that implement op sorted by UnitPriorities;
05  foreach(FU ∈ F)
06    P = paths from FU.output to dst sorted by PathPriorities;
07    foreach(p ∈ P)
08      p.timings.end = clock;
09      calculate p.timings.start;
10      if (resources of p are not reserved in ss.resTable)
11        FU.timing.end = p.timings.start;
12        calculate FU.timing.start;
13        if (FU is not reserved in ss.resTable)
14          copyStatus = ss;
15          if (ScheduleOperands(op, FU.timing.start, copyStatus, FU))
16            ss = copyStatus;
17            reserve FU and p in ss.resTable;
18            add corresponding RTAs to ss.RTAs;
19            return TRUE;
20  bind op.result;
21  if (ScheduleRead(op.result, clk, ss, dst));
22    internalOutputs = internalOutputs + {op};
23    return TRUE;
24  return FALSE;
25
```

Figure 12- The ScheduleOperation function.

The *ScheduleOperation* function (Figure 12) tries to schedule an operation *op* so that its result is available at *dst* at clock cycle *clk*. If *op* is not pre-bound to a specific functional unit, then the list of functional units that can execute *op* is stored in *F* and sorted by the UnitPriorities (lines 2-5). 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 8-13). 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 16-20). 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.

The *ScheduleOperands* function (Figure 13) 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.

```
00bool ScheduleOperands(operation op, clock clk, schedule status ss, functional unit FU)
01  foreach(operand o ∈ op.operands)
02    destination dst = FU.inputs corresponding to o;
03    if (o is a variable or a constant)
04      if (o is not pre-bound to a storage)
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;
11
```

Figure 13- The ScheduleOperands function.

In the *ScheduleRead* function (Figure 14), the best available path that can transfer a value from its storage to the specified destination at clock cycle *clk* is selected and scheduled.

```
00bool 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.timings.end = clk;
04    calculate p.timings.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 14- The ScheduleRead function.

6. Experiments

In this section we report 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 performed two sets of experiments. First, we compiled a set of benchmarks on a set of architectures and evaluated the schedules. We reused the same datapath to compile and implement different benchmarks. Next, for

DCT algorithm, we started from a simple architecture and iteratively refined and customized it to improve the performance, power, energy and area.

6.1 Reusing datapath for different benchmarks

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.

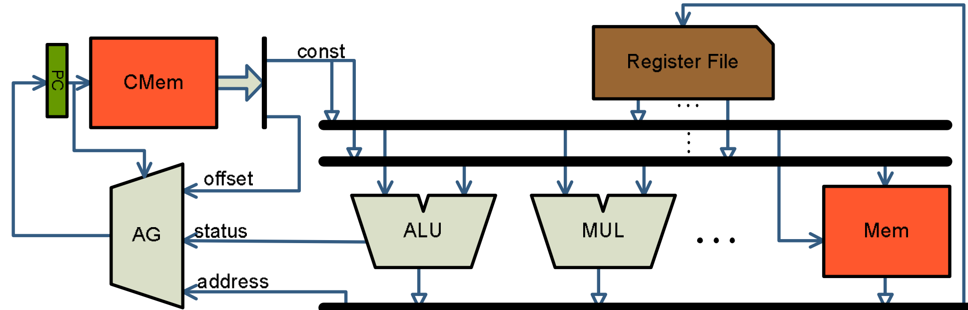

Figure 15- Datapath with simple interconnects.

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 15. 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 (CDP+F) 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 16.

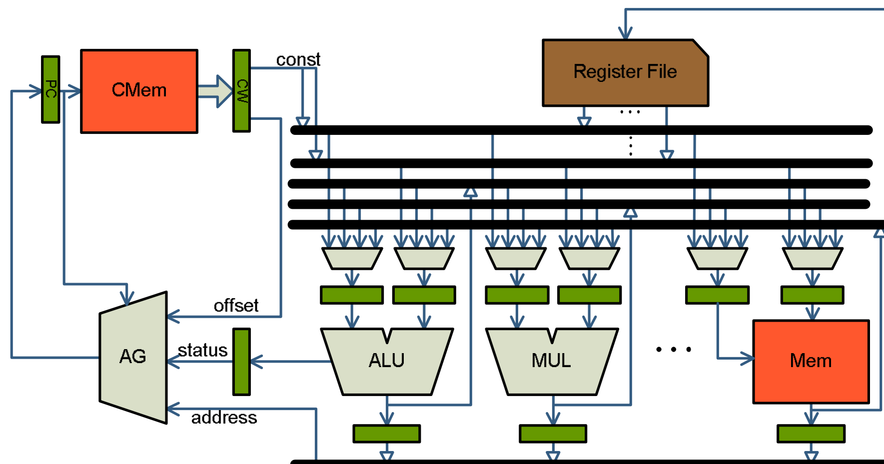

Figure 16- Datapath with complex interconnects.

We scheduled the benchmarks on the above datapaths and verified the results by simulating the generated Verilog files. The number of execution cycles 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. The considerable

decrease of execution cycle counts from CDP to CDP+F shows that the data forwarding paths between components are utilized well by our scheduling algorithm.

Table 1- Execution cycles counts of benchmarks.

	NP	CP	CDP	CDP+F
Bdist2: block 16x10	6143	6326	7168	5226
DCT 8x8	5225	5882	7146	6570
FFT	219	220	218	166
Sort: Best case (N=100)	25447	35349	84161	74162
Sort: Worst case (N=100)	35149	49902	98714	88715

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 [21]. 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 unnecessary accesses to 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- Execution cycle counts 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	6529	5386	5322	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

To evaluate the effect of each architecture modification on the clock period and the overall execution delay of benchmarks, we synthesized the architectures on a Xilinx VertixProII FPGA package using the Xilinx ISE tools. The clock period of each architecture after placement and routing, retiming, and buffer to multiplexer conversion; is reported in Table 3.

Table 3- Clock period of architectures after synthesis.

architecture	NP	CP	CDP	CDP+F	NM1	NM2
clock period (ns)	12.4	9.8	5.4	6.7	8.6	8.7

In Table 4 under each architecture column, the first element shows the execution delay (cycle count × cycle period) of the benchmark and the second element shows the speedup vs. architecture NP. Although, for each benchmark, the number of cycles in Table 1 increases from architecture NP to CP and CDP, the execution times have decreased due to improvements of cycle periods.

Table 4- Execution delay (us) of benchmarks and speedup vs. NP.

	NP		CP		CDP		CDP+F		NM1		NM2	
bdist2	76.2	1.0	62.0	1.2	38.7	2.0	35.0	2.2	44.8	1.7	38.0	2.0
DCT	64.8	1.0	57.6	1.1	38.6	1.7	44.0	1.5	46.3	1.4	46.3	1.4
FFT	2.7	1.0	2.2	1.3	1.2	2.3	1.1	2.4	1.4	1.9	1.2	2.3
Sort: Best	315.5	1.0	346.4	0.9	454.5	0.7	496.9	0.6	344.9	0.9	348.0	0.9
Sort: Worst	435.8	1.0	489.0	0.9	533.1	0.8	594.4	0.7	470.0	0.9	474.6	0.9

Note that in these experiments, we neither used any optimization (such as loop unrolling) nor modified the source code of benchmarks to increase the parallelism. The main goal of these experiments was to show that our algorithm can schedule and generate control words for each clock and execute programs correctly.

6.2 Case study: DCT implementation

In this section, a short introduction on DCT is presented, and then a custom datapath for the DCT is designed and refined using NISC methodology. The Discrete Cosine Transform (DCT) [22] and Inverse Discrete Cosine Transform (IDCT) are important parts of JPEG [25] and MPEG [26] standards. MPEG encoders use both DCT and IDCT, whereas MPEG decoders only use IDCT. The definition of DCT for a 2-D 8×8 matrix of pixels is as follows:

$$F[u,v] = \frac{1}{N^2}\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} f[m,n]\cos\frac{(2m+1)u\pi}{2N}\cos\frac{(2n+1)v\pi}{2N}$$

Where u, v are discrete frequency variables ($0 \le u, v \le 7$), $f[i, j]$ gray level of pixel at position (i, j), and $F[u,v]$ coefficients of point (u, v) in spatial frequency. Assuming N=8, matrix C is defined as follows:

$$C[u][n] = \frac{1}{8}\cos\frac{(2n+1)u\pi}{16}$$

Based on matrix C, an integer matrix $C1$ is defined as follows:

$$C1 = \text{round}(\mathit{factor} \times C)$$

The $C1$ matrix is used in calculation of DCT and IDCT:

$$F = C1 \times f \times C2$$

where, $C2 = C1^T$. As a result, DCT can be calculated using two consecutive matrix multiplications.

6.2.1 Implementing DCT using general-purpose datapaths

Figure 17 shows the C code for multiplying two given matrix A and B using three nested loops. Using a MIPS M4K ™ Core processor [27], the matrix-multiplication-based DCT takes 13058 cycles to compute [24]. However, given the MIPS datapath, the NISC implementation takes 10772 cycles. The 20% reduction in number of cycles is because of the finer-grained control that NISC compiler has over the datapath compared to traditional compilers that use instruction-set abstraction. We generated the synthesizable hardware description for our NISC-style MIPS (NMIPS), and synthesized it using Xilinx ISE 6.3. In our implementation, the bus-width of the datapath is 16-bit, and it does not have any integer divider or floating point unit. The clock frequency of 78.3MHz was achieved after synthesis and Placement-and-Routing.

```
for(int i=0; i<8; i++)
   for(int j=0; j<8; j++){
      sum=0;
      for(int k=0; k<8; k++)
         sum = sum + A[i][k] ×B[k][j];
      C[i][j] = sum;
   }
```

Figure 17- C-Code of matrix multiplication.

All the experiments in this section are synthesized on Xilinx FPGA package Virtex2V250-6 using Xilinx ISE 6.3 tool. Two synthesis optimizations of retiming and buffer-to-multiplexer conversions are

applied during optimization to improve the performance. In these experiments, we set the PAR effort to the highest level possible for maximum clock speed.

Figure 18 shows a simple general-purpose datapath (GPD) that includes an ALU, a Register File (RF), a multiplier (Mul), a data memory (Mem), a Comparator (Comp), and three buses. The RF has 32 registers, and ALU and Comp are designed to execute various C operations listed in Table 5. In NISC, the controller has a fixed structure and includes an Address Generator (AG), a Program Counter (PC), and a Control Memory (CMem). To support function call, a Link Register (LR) is added to the controller. The control and data memories are implemented using FPGA Block RAMs. The Block RAMs are synchronous and need to be driven by the clock. On each FPGA package, 24 Block RAMs exist where each has 16-Kb capacity. In our experiments, only two Block RAMs per architecture are used. Also, the FPGA package has 24 pre-synthesized multipliers, which only one is used.

Component	Operations
ALU	Add, Sub, And, Or, Xor, Shift-right, Shift-left, Shift-right-unsigned, Negate, Not
Comparator	Equal, Not-equal, Greater-or-equal, Greater-than, Less-than, Less-or-equal, Greater-or-equal-unsigned, Greater-than-unsigned, Less-than-unsigned, Less-or-equal-unsigned

Table 5- ALU and Comparator operations.

To support constant-based operations and jumps, a 10-bit constant and a 10-bit offset is added. The total number of bits in a single control word, including the constant and the offset bits, is 61. The NISC compiler generates about 50 control words. The clock frequency of GPD is 92.6 MHz.

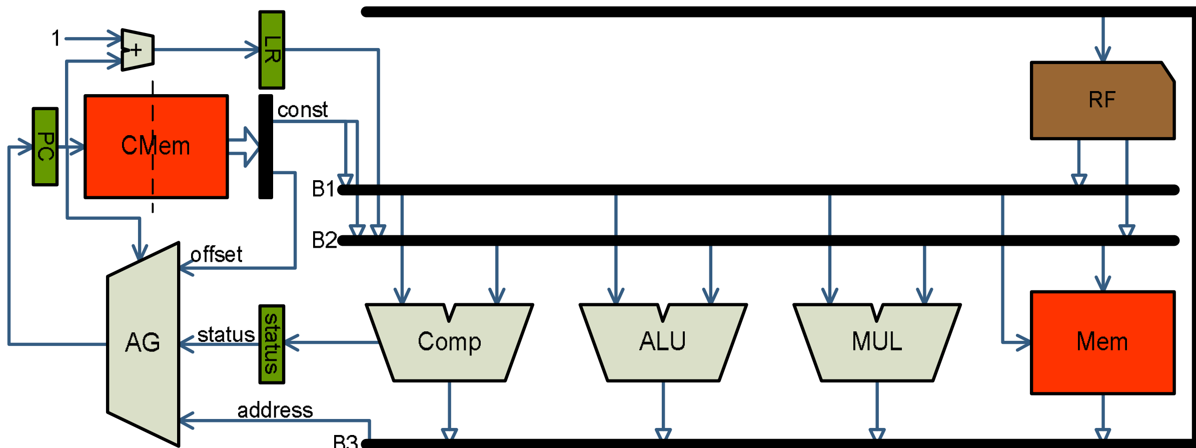

Figure 18- Block diagram of GPD.

In the rest of this section, we use different techniques to improve performance, area and power of the design. The techniques include:

1. Software transformations: unrolling the matrix multiplication loops to increase the parallelism in the code, and applying simple code transformations to reduce costly operations.
2. Using Multiply-and-Accumulate (MAC) unit: this technique improves the performance by chaining the two operations without accessing the Register File.
3. Adding pipeline registers to the datapath: if applied properly, this technique decreases the overall delay by reducing the clock period and increasing parallelism. Additionally, the power consumption decreases due to the reduction in switching activity.
4. Adding pipeline registers to the controller: although this technique increases branch delay (and hence the total number of cycles), the controller pipelining can help in reducing the critical path

5. Removing unused parts of ALU, comparator and register file: in a general-purpose datapath, all the operations supported in C, must be handled by the datapath. However, in a customized datapath, only the operations used by a specific application are supported. This optimization improves the area and performance.
6. Reducing the bit-width of some components without affecting the precision of the DCT calculations: this optimization reduces the area.

6.2.2 Designing a custom hardware for DCT

In general, customization of design involves both software and hardware transformations. In this section, we first apply the software transformations, and then customize and refine the datapath accordingly. Currently, the transformations are applied manually. In future, they can be applied automatically by tools.

6.2.2.1 Software transformations

To increase the parallelism, we unroll the inner-most loop of the matrix multiplication code. The transformed code is shown in Figure 19. Note that operation "*" represents accessing the value of a pointer (i.e. loading from memory). Next, we apply other software transformations to reduce the costly operations: To decrease the number of multiplications, we replace $i \times 8$ with i<<3 (i shift left three times). Additionally, to calculate the address, we need two consecutive additions, which may require two chained adders. However, if we replace one of the additions with an OR operation, then we can chain one adder with an OR unit, which is less costly than an extra adder. The conversion is possible in this particular application because of the special values of the constants. For example, $i8$+*const* is equal to $i8$|*const*, because $0 \leq const \leq 7$ at all time and the first three bits of $i8$ is always zero. Additionally, the two *for* loops can be merged to one, by combining the loops' counters. The new counter is represented by variable *ij*. Figure 20 shows the transformed code after the above modifications.

```
for(int i=0; i<8; i++)
   for(int j=0; j<8; j++){
      i8 = i × 8;
      sum = *(A + i8) × *(B + j);
      sum += *(A + i8 + 1) × *(B + 8 + j);
      sum += *(A + i8 + 2) × *(B + 16 + j);
      sum += *(A + i8 + 3) × *(B + 24 + j);
      sum += *(A + i8 + 4) × *(B + 32 + j);
      sum += *(A + i8 + 5) × *(B + 40 + j);
      sum += *(A + i8 + 6) × *(B + 48 + j);
      sum += *(A + i8 + 7) × *(B + 56 + j);
      C[i][j] = sum;
   }
```

Figure 19- C-code of unrolled matrix multiplication.

```
ij=0;
do {
  i8 = ij & 0xF8;
  j = ij & 0x7;
  aL = *(A+ (i8|0) ); bL = *(B + (0|j) );  sum =  aL × bL;
  aL = *(A+ (i8|1) ); bL = *(B + (8|j) );  sum += aL × bL;
  aL = *(A+ (i8|2) ); bL = *(B + (16|j) ); sum += aL × bL;
  aL = *(A+ (i8|3) ); bL = *(B + (24|j) ); sum += aL × bL;
  aL = *(A+ (i8|4) ); bL = *(B + (32|j) ); sum += aL × bL;
  aL = *(A+ (i8|5) ); bL = *(B + (40|j) ); sum += aL × bL;
  aL = *(A+ (i8|6) ); bL = *(B + (48|j) ); sum += aL × bL;
  aL = *(A+ (i8|7) ); bL = *(B + (56|j) ); *(C + ij) = sum + (aL × bL);
  ++ij;
} while(ij!=64);
```

Figure 20- Transformed matrix multiplication C-code.

6.2.2.2 Initial Custom datapath: CDCT1

By looking at the body of loop in Figure 20, four steps of computation can be identified:

- Calculation of the memory addresses of the relevant elements
- Loading the values of those elements from data memory,
- Multiplying the two values,
- Accumulating the multiplication results.

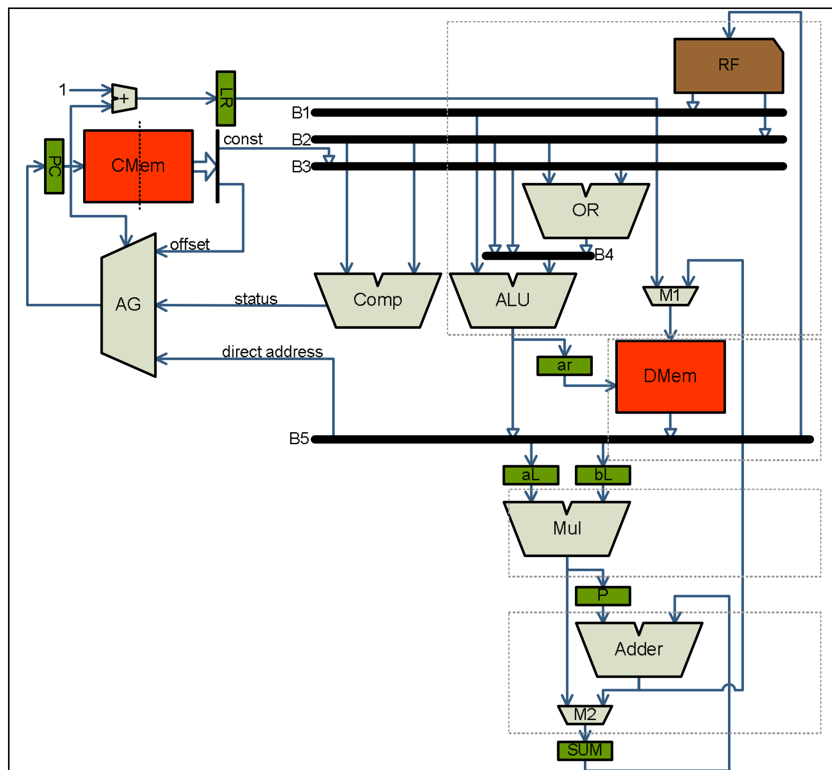

Figure 21- Block diagram of CDCT1.

We design our custom datapath in a way that each of these steps is a pipeline stage. Figure 21 shows the proposed custom pipelined datapath (CDCT1). The datapath includes four major pipeline stages that are marked in the figure. We have used operation chaining to reduce RF file accesses and decrease register pressure. Chaining the operations improves the energy consumption and performance. The OR and ALU are chained, as well as the Mul and Adder. Note that the chaining of multiply and add forms a MAC unit in the datapath. To assure proper usage of the MAC unit, we enforce mapping the *aL*, *bL*, and *sum* variables, to aL, bL and SUM registers in the datapath. After compilation, the total number of cycles of the DCT is 3080, and the maximum clock frequency is 85.7MHz.

Component	**CMem+CW**	**RF+RF_o**	**ALU+ALU_o**	**RF setuptime**
Delay (ns)	3.28	2.39	5.4	0.58

Table 6- Critical-path delay breakdown of CDCT1.

Table 6 shows the critical-path breakdown of CDCT1. Each column in the table shows the sum of a component delay and its output-interconnect delay. The critical path goes through CMem, RF, B2, B4, ALU, B5, and back to RF.

CDCT2: Bus customization and adding a pipeline register to the datapath

According to Table 6, ALU and the wire that connects ALU to RF are in the critical path. To reduce the critical path delay, we insert an additional pipeline register (i.e. *reg*1) in the output of the ALU, and call the new design CDCT2 (Figure 22). We also replace all the global buses, including B5, with point-to-point connections. Only the connections that are used by the DCT application are kept. Since there is no function call in DCT, the LR register can be removed. The NISC compiler automatically analyzes the new datapath and regenerates the control words to correctly handle the flow of the data. CDCT2 runs the DCT algorithm in 2952 cycles at the maximum clock frequency of 90MHz. The reduction in number of cycles is due to additional parallelism created by the separation of interconnects. Table 7 shows the breakdown of the critical path of CDCT2. Note that, in CDCT2, the critical path goes through the comparator instead of the ALU. In general, adding pipeline registers combined with retiming optimization is more effective; because, retiming balances the delay of the pipeline stages by moving some of the logic across the pipeline registers. In all the experiments here, we enabled retiming optimization to improve the clock frequency.

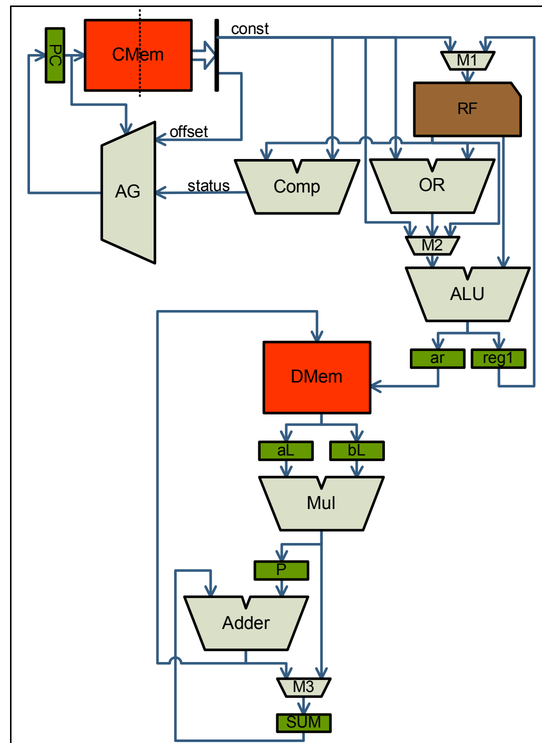

Figure 22- Block diagram of CDCT2.

Component	CMem+CW	RF+RF_o	Comp+comp_o	AG+PC setup
Delay (ns)	2.93	2.45	3.726	2.06

Table 7- Critical-path delay breakdown of CDCT2.

6.2.2.3 CDCT3: Eliminating the unused parts of ALU, comparator and RF

Next, we customize the ALU and comparator for the DCT application. In Figure 20, only Add, And, Multiply and Not-equal (!=) operations are used. The first two operations are executed by ALU, the third by Mul, and the last by Comp. We can simplify the ALU and comparator by eliminating the unused operations. NISC compiler allocates and uses nine registers in RF. Therefore, we reduce number of registers in RF from 32 to 16. The new architecture (CDCT3) runs much faster at the clock frequency of 114.4MHz. The breakdown of critical path delay (Table 8) shows a considerable reduction in the delay of the comparator. Also, the number of fanouts of RF output wires is reduced, and hence its interconnect delay is reduced. These modifications, also, reduce the area significantly.

Component	CMem+CW	RF+RF_o	Comp+comp_o	AG+PC setup
Delay (ns)	2.76	1.64	2.29	2.06

Table 8- Critical-path delay breakdown of CDCT3.

6.2.2.4 CDCT4 and CDCT5: Controller pipelining

Looking at the critical paths of the architectures, it is evident that the controller contributes to a major amount of the delay. The CMem, CW, and Address Generator (AG) delays are part of the critical path of CDCT3. To reduce the effect of the controller delay, we insert one pipeline register (i.e. CW register) in front of the CMem. The new architecture (CDCT4) can run much faster at the clock frequency of 147MHz. Table 9 shows a reduction in the critical path delay. On the downside however, the number of cycles of DCT increases to 3080 because of an extra branch delay cycle. Note that the NISC compiler

automatically analyzes the datapath and notices the extra branch delay. So, the user does not need to change the compiler manually.

Component	CMem+CW	RF+RF_o	Comp+comp_o	AG+PC setup
Delay (ns)	1.39	1.6	1.74	2.06

Table 9- Critical-path delay breakdown of CDCT4.

To further reduce the effect of controller's delay on the clock cycle, we insert another pipeline register (called status register) at the output of the Comp. This register eliminates the AG's delay from the critical path. Table 10 shows the breakdown of the critical path delay of the new architecture (CDCT5). In CDCT5, the critical path goes through the multiplier. Since the multiplier is a pre-synthesized unit in the FPGA package, it is not possible to reduce the critical path delay any further. Note that, CDCT5 has a branch delay of two and runs at the clock frequency of 170MHz. The total number of cycles of DCT has increased to 3208. CDCT4 and CDCT5 occupy larger area than CDCT3 due to the additional CW and status registers.

Component	bL+bL-o	Mul+Mul-o	P setuptime
Delay (ns)	1.29	4.25	0.3

Table 10- Critical-path delay breakdown of CDCT5.

6.2.2.5 CDCT6: bit-width reduction

In the final optimization, we reduce the bit-width of some of the components without affecting the precision of the calculations. The goal of this optimization is further reducing the area. We observed that the address-calculation pipeline stage does not need the 16-bit operations. In fact, all the address values are in the range of 0 to 255. Therefore, the bit width of RF, OR, ALU, and Comp are reduced to 8 bits. In this case, the clock frequency remains fixed at 170MHz. Figure 23 shows final design (CDCT6) after all the transformations.

Figure 23- Block diagram of CDCT6.

6.2.3 *Comparing performance, power, energy and area of the NISCs*

Table 11 summarizes all the experiments in Section 6.2. The second column briefly describes the experiments, and the third column shows the bit-width of Control Words. In these experiments, we first mapped DCT to two general-purpose datapaths (NMIPS and GPD). Then, we designed a custom pipelined datapath for DCT called CDCT1. Next, we added an additional pipeline register to CDCT1, simplified the functional units, and added controller pipelining. Finally, we optimized the bit-width of address-calculation pipeline stage and generated CDCT6.

	General Description	CW bit width
NMIPS	NISC with MIPS datapath	76
GPD	A general-purpose NISC architecture	61
CDCT1	Custom NISC for DCT	59
CDCT2	CDCT1 + additional pipeline register + bus transformation	60
CDCT3	CDCT2 with a simplified ALU, comparator and RF	50
CDCT4	CDCT3 + CW register	50
CDCT5	CDCT4 + status register	51
CDCT6	CDCT5 with a 8-bit-width address calculation pipeline stage	51

Table 11- Summary of the experiments.

Table 12 compares the performance, power, energy, and area of the all NISC implementations. We synthesized all the NISC architectures on FPGA. After placement and routing and based on the critical path delays, we extracted the maximum clock frequency of each design (shown in the third column).

In Table 12, column fourth shows the total execution time of the DCT algorithm calculated based on number of cycles and the clock frequency. Note that although in some cases (such as CDCT4 and CDCT5) the number of cycles increases, the clock frequency improvement compensates for that. As a result, the total execution delay maintains a decreasing trend.

Column fifth shows the average power consumption of the NISC architectures while running the DCT algorithm. All the designs are stimulated with the same data values. We used Post-Placement and Routing simulation to collect the signal activities, and computed the power consumption using Xilinx XPower tool. Figure 24 shows the power breakdown of different designs in terms of the clock, logic and interconnect power. Column sixth shows the total energy consumption calculated by multiplying power and execution time.

	No. of cycles	Clock freq	DCT exec. time(us)	Power (mW)	Enegy (uJ)	Normalized area
NMIPS	10772	78.3	137.57	177.33	24.40	1.00
GPD	11764	79.5	147.97	150.33	22.24	1.00
CDCT1	3080	85.7	35.94	120.52	4.33	0.81
CDCT2	2952	90.0	32.80	111.27	3.65	0.71
CDCT3	2952	114.4	25.80	82.82	2.14	0.40
CDCT4	3080	147.0	20.95	125.00	2.62	0.46
CDCT5	3208	169.5	18.93	106.00	2.01	0.43
CDCT6	3208	171.5	18.71	104.00	1.95	0.34

Table 12- Performance, power, energy, and area of the DCT implementations.

In these experiments, GPD consumes lower power than NMIPS because it does not have any forwarding path. Also, CDCT1 consumes less power than GDP because CDCT1 controls the activation of multiplier by aL and bL registers, while GDP wastes power by always activating ALU, Mul and Comp simultaneously.

CDCT2 consumes less power compared to CDCT1 because of the replacing shared bus B5 with short point-to-point connections. Instead of having a B5 with two fanins and four fanouts, three point-to-point

connections are used. This optimization reduces the total bus capacitance and hence, the total power consumption. The diagram of Figure 24 confirms the reduction in interconnect power consumption of CDCT2.

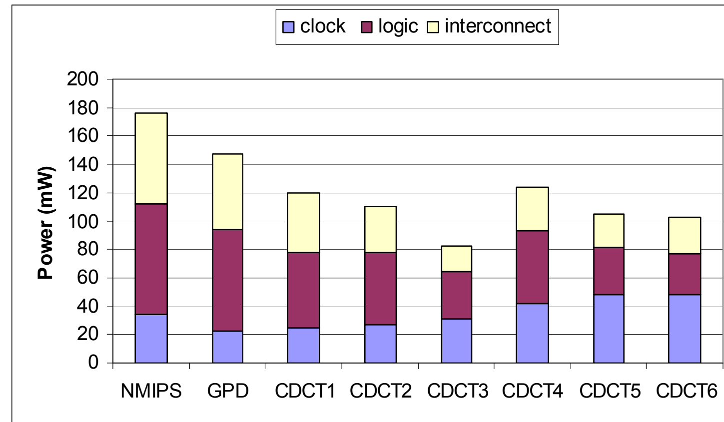

Figure 24- Power breakdown of the DCT implementations.

Power consumption of CDCT3 is lower than CDCT2 because of the elimination of unused operations in ALU and comparator. Elimination of operations reduces number on fan-outs of the RF output wires. Therefore, reduction in interconnect power, as well as logic power is achieved. The power breakdown of CDCT3 confirms this fact. Note that as the clock frequency goes up, the clock power gradually increases.

In CDCT4, the power consumption further increases, because of: (1) the higher clock power due to higher clock frequency and higher number of pipeline registers; (2) the higher logic power due to CW register gates; and more importantly, because of (3) the power consumption of logic and interconnects added by retiming algorithm. Since the difference between the delays of the two pipeline stages located before and after CW register is high, the retiming works aggressively to balance the delay. As a result it adds extra logic to the circuit.

In CDCT5, we added the status register to the output of Comp and reduced the critical path. In this case, the retiming algorithm works less aggressive because the delays of the pipeline stages are less imbalanced. As a result, we observe a reduction in logic and interconnect power. The last column of Table 12 shows the normalized area of different designs calculated based on the number of FPGA slices that each design (including memories) occupies. The area trend also confirms the increase in area in CDCT4 followed by a decrease in CDCT5, which we believe is because of the retiming.

Figure 25 shows the performance, power, energy and area of the designs normalized against NMIPS. The total execution delay of DCT algorithm has a decreasing trend except for the GPD that takes many cycles to finish the execution. The power consumption decreases up to CDCT3 and then increases. The energy consumption significantly drops at CDCT1, because of the reduction in number of cycles and power consumption. From CDCT1 to CDCT6, the energy decreases gradually in a slow paste.

As shown in Figure 25, CDCT6 is the best design in terms of delay, energy consumption and area. However, CDCT3 is the best in terms of power consumption. As a result, CDCT3 and CDCT6 are considered the pareto-optimial solutions. Compared to NMIPS, CDCT6 runs 7.14 times faster, consumes 1.69 times less power and 12.51 times less energy. Also CDCT6 occupies 3 times less area than NMIPS. Note that performance of NMIPS is 20% better than performance of a MIPS core. Also, since NMIPS does not have instruction decoder, its area is less than MIPS. In our experiments, we compared the results to NMIPS which is conservative relative to MIPS core.

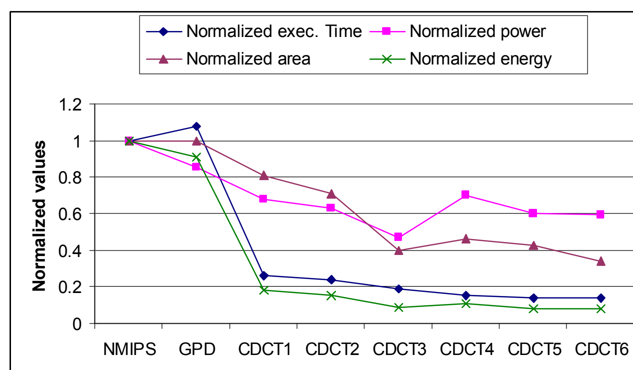

Figure 25- Comparing different DCT implementations.

The experiments show that using NISC methodology, a designer (or a tool) can start from a simple general-purpose datapath and iteratively refine and customize it for one or more applications. If properly designed, the custom hardware can be shared by several applications. It is also possible to trade some of the customizations for post-implementation re-programming by maintaining some of the general-purpose features of the initial design.

6.2.4 Comparison with a manual design

We have also compared the quality of our final design (CDCT6) with a commercial manual design [28]. In [28], the quality of a manual design after mapping to Xilinx Virtex2V250-6 package is reported. We also used the same package in our experiments to enable the comparison. Their design takes 82 cycles to compute an 8×8 DCT with a 15-bit precision (ours has a 16-bit precision). They have achieved maximum clock frequency of 74MHz on the FPGA package (we achieved 170MHz). Therefore, their total execution time of an 8×8 DCT is 1.1us. Compared to NMIPS that takes 137.57us, the manual design is 125 times faster. This clearly shows two orders of magnitude performance gap between the manual design and software implementation. Compared to CDCT6 that takes 18.71us to compute DCT, the manual design is 17 times faster. These results show that a custom NISC architecture can serve as an intermediate point between software and hardware implementations.

On the other hand, the total area of the manual design is 1365 FPGA slices, while the area of CDCT6 is 169 slices. Note that the low area of CDCT6 allows fitting eight of CDCT6 in the same area as of the manual hardware design. Since the DCT algorithm can usually run on different parts of an image in parallel, the performance of eight CDCT6 is almost eight times of the performance of one. This makes the CDCT6 only two times slower than the manual hardware design. Figure 26 compares the performance and normalized area of the two designs (power is not reported in [28]). Note that it took us about one week to explore different design alternatives while it usually takes significantly longer time to implement and verify a manual designs.

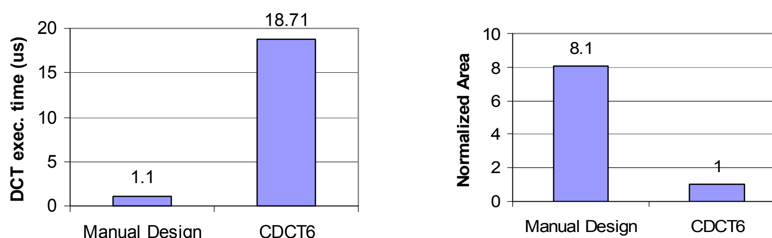

Figure 26- Comparing CDCT6 with a commercial manual design [28].

7. 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 and 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. 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-grained 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 4).

8. Conclusion

We introduce No-Instruction-Set-Computer (NISC) to enable design reuse and refinement. The NISC compiler has complete fine-grained control over the datapath and generates the controller by specifying the control values of components in each cycle. We also presented a scheduling and binding algorithm for compiling a program on a NISC.

Our algorithm is different from HLS techniques because it assumes that the datapath is fixed and performs the scheduling and binding simultaneously while processing the DFGs backward. It is also different from conventional instruction-set based compiler techniques because it directly maps the program on a given datapath without using any instruction abstraction. Consequently, it must deal with all structural details of the architecture and solve more complex problems.

Our experiments indicate that the algorithm efficiently supports features such as controller / datapath pipelining, data forwarding, multi-cycle and pipeline units, and operation chaining. We ran several benchmarks through our prototype compiler and simulated and synthesized the results on Xilinx FPGA using the Xilinx ISE tools. We showed that a NISC with a datapath similar to that of a MIPS M4K can perform comparably or better (up to 70%). We predict the same applies when using datapath of other embedded processor cores. We also presented a case-study of designing a custom datapath for DCT application using NISC. We started from a general-purpose pipelined datapath and iteratively refined it to achieve better performance, power and area. Our results show 7.14 times performance improvement, 1.64 times power reduction, 12.5 times energy savings, and more than 3 times area reduction compared to a soft-core MIPS implementation. We also compare the quality of our designs to a state-of-the-art commercial manual design.

Future works focuses on applying more optimizations in the compiler and simultaneous optimization of datapath for multiple applications.

9. Acknowledgements

This work is in part supported by SRC contact 1118.001.

10. 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] MIPS32® M4K™ Core, http://www.mips.com
[22] N. Ahmed, T. Natarajan, and K.R. Rao, Discrete Cosine Transform, *IEEE Trans. On Computers*, vol. C- 23, 1974.
[23] M.K. Jain, M. Balakrishnan, and A. Kumar, ASIP Design Methodologies: Survey and Issues, *In Proc. of International Conference on VLSI Design*, 2001.
[24] M. Reshadi, D. Gajski, An Algorithm for Compiling Programs to Custom Pipelined Datapaths, *In Proc. International Symposium on System Synthesis* (*ISSS05*), 2005.
[25] ISO/IEC JTC1 CD 10918. Digital Compression and Coding of Continuous-tone Still Images - part 1, requirements and guidelines, ISO, 1993 (JPEG)

[26] ISO/IEC JTC1 CD 13818. Generic Coding of Moving Pictures and Associated Audio: Video, ISO, 1994 (MPEG-2 standard)
[27] MIPS32® M4K™ Core, http://www.mips.com
[28] http://www.cast-inc.com/cores/dct/cast_dct-x.pdf