NISC Modeling and Simulation

Mehrdad Reshadi and Daniel Gajski

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

{reshadi, gajski}@cecs.uci.edu

CECS Technical Report 04-08
March, 2003

NISC Modeling and Simulation

Mehrdad Reshadi and Daniel Gajski

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

{reshadi, gajski}@cecs.uci.edu

CECS Technical Report 04-08
March, 2003

Abstract

Running an application on a general purpose processor is not very efficient and implementing the whole application in hardware is not always possible. The best option is to run the application on a customized data path that is designed based on the characteristics of the application. The instruction set interface in normal processors limits the possible customization of the data path. In NISC, the instruction interface is removed and the data path is directly controlled. Therefore, any customization in the data path can be easily utilized by the supporting tools such as compiler and simulator and one set of such tools is enough to handle all kinds of data paths. To achieve this goal, a generic model is needed to capture the data path information that these tools require. In this report we present one such model and show how it can be used for simulation and compilation. We also discuss the issues that must be addressed during compilation and simulation based on the proposed model.

Contents

List Of Figures

NISC Modeling and Simulation

Mehrdad Reshadi and Daniel Gajski

Center for Embedded Computer Systems
University of California, Irvine

{reshadi, gajski}@cecs.uci.edu

1 Introduction

In a Soc design, the system is usually partitioned into smaller components that execute a specific set of tasks or applications. These components are usually in the form of an IP, developed by a third party or reused from previous designs. The efficiency of the SoC design relies on *efficient use* of IP cores as well as use of *efficient IP* cores. Efficient use of an IP core depends on the tools and techniques that are utilized for mapping the application to the IP. On the other hand the efficiency of an IP depends on how well it matches the needs and behavior of the application. The system components may be implemented by pure software, using a general purpose processor; pure hardware using logic gates; or a mixture of software and customized hardware.

General purpose processors are fairly easy to program and the size of the application is bounded by the size of the program memory, which is normally very large. However, because of their generality, these processors are not very efficient in terms of quality metrics such as performance and energy consumption. Such processors use an *instruction set* as their interface which is used by a compiler to map the application. The available compilers can perform well as long as the instruction set is fixed, very generic and fairly simple (RISC type). In other words, the efficiency of general purpose processors and the efficient use of them are very much limited by their instruction set.

Implementing a system component in hardware only using logic gates and blocks can be very efficient in terms of performance and energy consumption. However, they have very little flexibility and the size of the applications that they can implement is bounded by the available chip area. Because of the complexities involved in developing such components, the synthesis tools can work only on relatively small applications. High design complexity and cost of these components limits their use in SoC.

A better option for implementing a system component or an IP is to have a customized hardware for each application that runs an optimized software. This approach is more efficient than using only general purpose processor and more flexible than using logic blocks. Application Specific Instruction set Processors (ASIPs) tried to achieve this goal by performing the following steps:

1. Finding a proper and critical sub-functionality of the application as a candidate for speedup.
2. Designing an efficient data path for executing the candidate sub-functionality.
3. Designing an instruction and updating the processor instruction decoder accordingly.
4. Updating the compiler to take advantage of the new complex instruction.
5. Compiling the application to use the new improved data path.

Typically in ASIPs, the complexity of the instructions generated in the third step is very much limited by the instruction decoder. This in turn limits the type of possible data paths and hence limits the range and complexity of possible application sub-functionalities that can be improved. On the other hand, it is very difficult to have the compiler use such complex instructions

efficiently. Capturing and modeling the internal behavior of the processor in terms of instruction set and then using the model for generating compiler have proven to be very difficult or even impossible. This is one of the reasons behind failure or limitations of retargetable compilers. We believe that the instruction set is an extra unnecessary layer of abstraction between the processor behavior and the compiler. In processors, the most complexity and also the critical path is usually in the instruction decoder. In compilers, instruction selection performs very poorly for complex instructions (which led to replacement of CISC with RISC). NISC, No Instruction Set Computer, addresses this problem by removing the instruction set interface. In other words, Steps 3 and 4 from the above list will be removed and instead the compiler directly maps the application to the customized data path by using register transfer (RT) matching techniques rather than instruction set matching techniques. We will show that by using the NISC concept, virtually any type of data path can be supported. This has two very important implications: first, any application characteristics can be utilized to generate the most optimized possible data path for executing that application; and second, a single set of supporting tool chain; such as compiler and simulator; are enough to map any application to any customized (and proper) data path.

The controller (instruction decoder, etc.) is usually the most complex part of a normal processor and therefore it contains the critical path and also requires the most verification effort. In addition to the two formerly described advantages, the NISC concept significantly simplifies the controller and therefore the NISC processors can run much faster than their traditional ASIP counterpart and require much less development (especially verification) effort.

The rest of this report is organized as follows. Section 2 compares NISC with instruction set based processors. Section 3 describes how optimized NISC processors can be obtained for an application. Section 4 explains how a NISC processor is modeled and the issues involving in simulation and compilation are discussed in Sections 5 and 6, respectively. Finally, Section 7 concludes the report.

2 NISC's View of the Processor

In general a processor is a sequential hardware that can be viewed as a *controller* and a *data path* as shown in Figure 1. In each step, the controller generates control bits for the data path to perform a specific task and gets a feedback from data path to determine the state of the circuit in the next step. To execute an application efficiently, we need to have an optimized data path for the application and map (compile) the application on to the data path by generating, customizing or configuring the controller. The utilization of the data path depends on its controllability and the interface that the controller provides for this purpose.

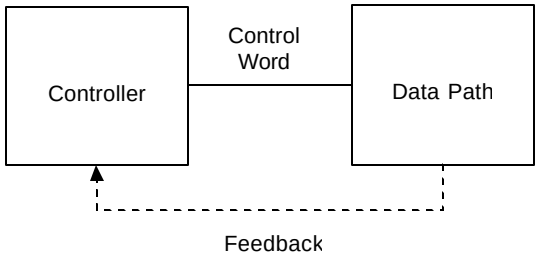

Figure 1 - General sequential hardware

In today's processors, the exposed interface of the controller is the Instruction Set and the compiler or the programmer can only control the sequence of these instructions in the program memory.

Every processor has a complicated controller that converts the sequence of instructions in memory (address space dimension) to a sequence of control words for each clock cycle (time dimension). An instruction implicitly describes the behavior of the data path at different times. In a pipelined data path, the behavior of instructions overlap in time and the controller must extract the overall exact behavior of the data path at each clock cycle. Figure 2 shows the typical structure of RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer) processors [1]. In CISC, each complex instruction is translated to a sequence of µ-instructions and then each µ-instruction participates in the generation of the control words at each cycle. In other words, CISC requires two translations in address space dimension (application → sequence of complex instructions, and complex instruction → sequence of µ-instructions) and one translation in time dimension (µ-instruction → control word). In RISC, the instructions are simpler and directly participate in generation of control words.

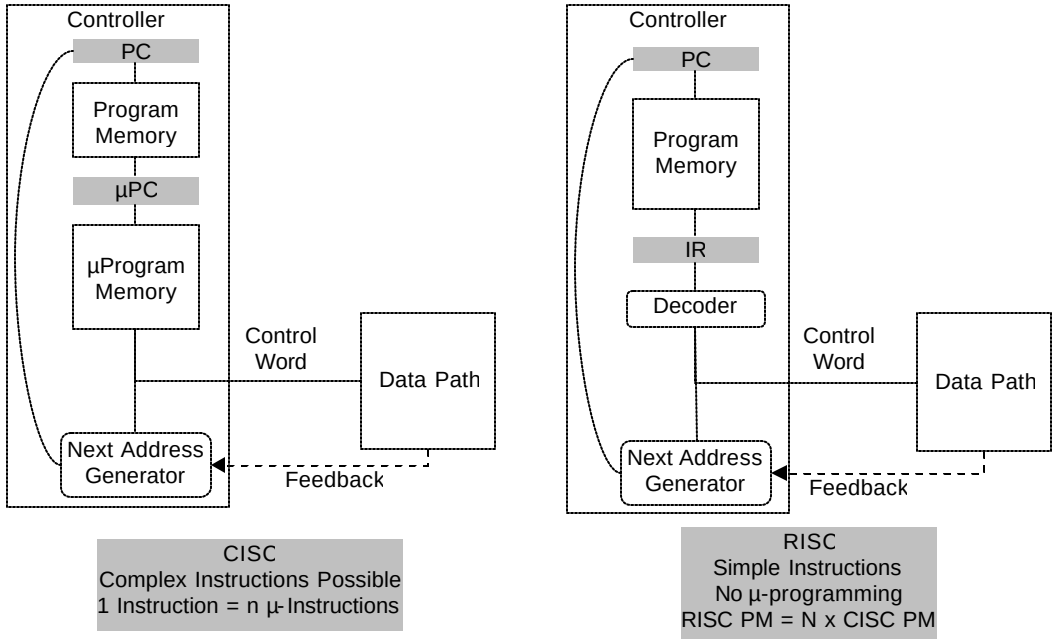

Figure 2 - RISC and CISC block diagram

The complexity of the controller and the one dimensional control (address space dimension only) in RISC and CISC processors limit the compiler or programmer's control for utilizing the data path. Therefore, any data path customization is very difficult to be reflected in the processors interface let alone, updating the compiler to use it. VLIW machines tried to address this problem by limitedly adding the time dimension to their interface and allowing the compiler to determine what part of the data path it wants to use. However, they still use the instruction set interface which requires a decoder and limits the control over the data path. For example, in TI-C6x [2] VLIW processors, the fetch and decode take six pipeline stages while execution takes only five pipeline stages.

Because of the above complexities, the controller and data path in normal processors are fixed or have very limited customization capability. For some changes in the data path, the whole controller as well as a significant part of the supporting tools such as compiler and simulator must be redesigned. On contrary, any data path customization is possible in NISC and is easily supported by the tools.

The instruction set in normal processors represents the behavior of both the controller and the data path. By removing the instruction set abstraction in NISC, we can consider the behavior of controller and data path separately. With a proper controller, the compiler can efficiently compile an application to any custom data path. In other words, the CFG (control flow graph) of the application is compiled based on the features of the controller and the DFG (data flow graph) of the application is compiled to the custom data path. To achieve this goal, we eliminate any translation between the address space dimension and the time dimension as much as possible. The NISC compiler generates the proper control words and loads them in the program memory. As Figure 3 shows, the controller should only load the control words and apply them to the data path. The control words in NISC may be wider than the RISC instructions. However, due to better utilization of parallelism, the number of NISC control words may be less than the number of RISC instructions. Therefore the size of program memory in both NISC and RISC is not significantly different.

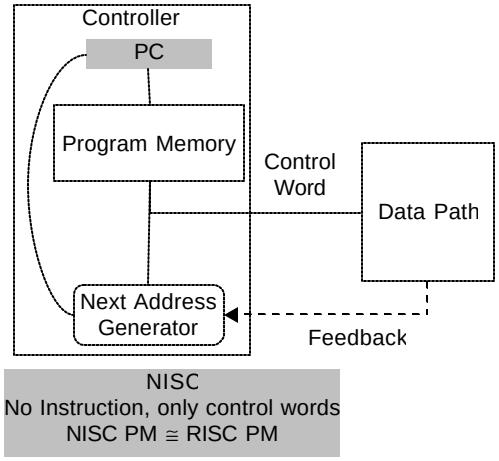

Figure 3 - NISC block diagram

Because of the clear separation of controller and data path in NISC, a wide range of customizations are now possible in the system. When compiling the basic blocks (a set of operations that *always* run consecutively, i.e. there is no jump operation among them) of the application, the compiler only needs to know the features of the data path and generate the corresponding control words. In the next sections we will describe how data path is modeled and will point out some possible approaches for using the model in simulator and compiler. The controller of NISC can also be customized to improve the control flow of the application. Changing the number of pipeline stages in the controller, adding a hardware stack for storing the return address of a function call and adding a branch prediction are examples of possible customizations of the controller. This information must be considered by the compiler when handling the CFG of the application.

3 Generating the Optimized NISC for an Application

An efficient implementation of an application can be achieved by compiling the application on a customized data path that has been designed for that application. Figure 4 shows how this can be done using NISC processors. In this methodology, first, the application is profiled and analyzed to extract its important characteristics that can be used for generating a customized data path. This process can be done automatically or by the designer and should provide information such as depth and structure of pipeline, level of parallelism, type and configuration of components and so on. This information is captured in the NISC model and is used to derive the compiler and simulator.

The compiler gets the application and the NISC model as the input and generates the corresponding control words. It translates each operation of the application into a set of register transfers and then schedules them in order to meet the resource and timing constraints.

The simulator gets the sequence of the control words and simulates them on the model of the target NISC processor. The simulator serves two purposes: first, it validates the correctness of the timing and functionality of the compiler's output; and second, it evaluates the performance metrics such as speed and energy consumption. The performance results of the simulator can be analyzed to fine tune the structure of the customized NISC processor.

As Figure 4 shows, the NISC processor model plays the key role in this methodology. Its structure determines the flexibility of the analyzer or designer for suggesting more optimized processors. It also affects the quality and complexity of the simulator and compiler. In this report, we mainly focus on the details of this model and describe what information it should capture and how this information can be later used in the simulator or compiler.

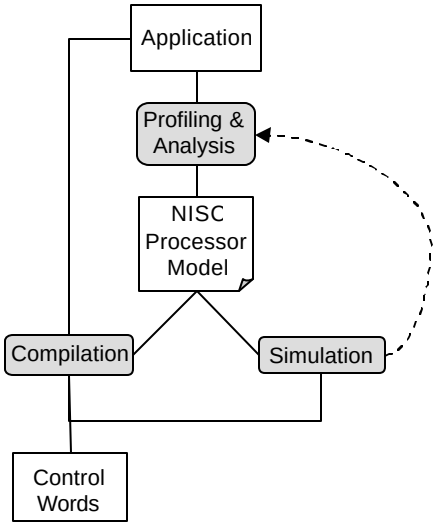

Figure 4 - NISC based methodology

4 NISC Processor Model

The NISC processor model is used mainly for three purposes: translating application operations to register transfers, constructing the control words from the set of register transfers, and decoding the control words back to their corresponding register transfers. Our NISC model includes three category of information: the clock frequency, model of controller, and model of data path. The model of controller captures the functionality of "Next Address Generator" unit, number of pipeline stages in the controller and the possible branch prediction strategy. The model of data path captures the flow and timing of possible data transfers in the processor as well as the mapping between the control bits of the units and their functionality. In this section, we first describe the model of data path and then review the modeling and support of multi-cycle components as well as pipelined data paths.

4.1 Model of Data Path

The model of data path captures the type and direction of possible register transfers in the system and shows how components of the system are connected. In other words, it shows how data should be moved between two points in the system and how long will it take. The data path is captured

through a connectivity graph. The connectivity graph is a directed graph that its nodes are the components of the data path and its edges represent a direct connection between the corresponding components. Each component has a set of input, output and control ports and the edges of the graph connect only ports of components. Figure 5 shows an example of a simple data path and its corresponding connectivity graph. In Figure 5(a), the register file *RF* is connected to bus *B* which derives one of the two input registers of the *ALU*. The *ALU* gets its inputs from registers *a1* and *a2* and writes its output to register *o1.* Figure 5(b) shows the connectivity graph of this simple data path. In this graph, each component has a name and specifies its type. For example, component *B* of type *Bus1* corresponds to the bus *B* in the data path of Figure 5(a). The ports of the components in the graph are determined by its type and are connected to the ports of other components to represent a point to point connection. For example, component *ALU* has two input ports *i1* and *i2* and one output port *o*. The output port *o* of *ALU* is connected to the input port *i* of register *o1*. Therefore, the *ALU.o→o1.i* edge indicates that data can be transferred from *ALU* to *o1*.

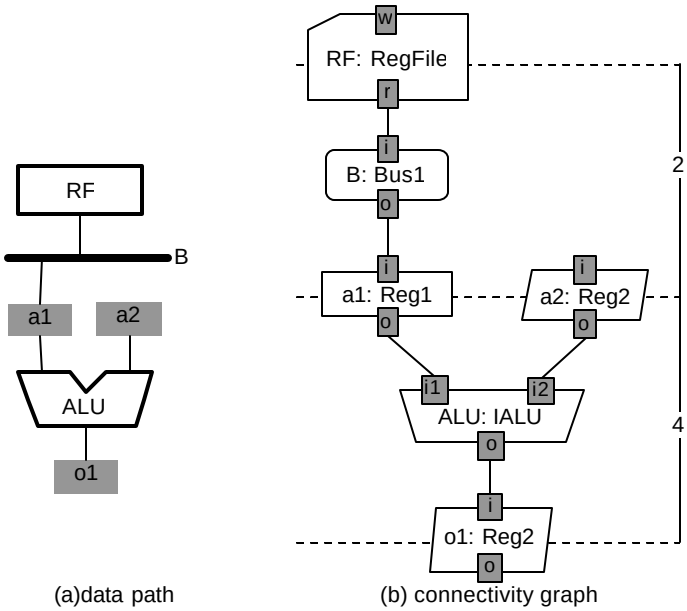

Figure 5 - NISC model of a simple data path

Figure 5(b) shows only the input/output ports and does not show the control ports. The component types that are used in the connectivity graph of Figure 5(b) are described in Figure 6 where the port configuration, timing information and behavior of each component type is shown For example, the component type *Reg1* defines a register with one input port *i* and one output port *o* and control ports *ld* and *clk*. It also defines the setup time for input port *i*, $T_s(i)$, the setup time for control port *ld*, $T_s(ld)$, and the delay of output port *o*, $T_d(o)$. The behavior of the component is described based on the values of ports. The *Reg1* component type has a behavior that is synchronized with the clock and determines that on a clock edge if the value of control port *ld* is true, then the output port *o* gets the value of the input port *i.* Not that he *RegFile* component has two behaviors. The one that writes to the internal storage is synchronized with the clock while the behavior that reads from the internal storage is independent of the clock. s

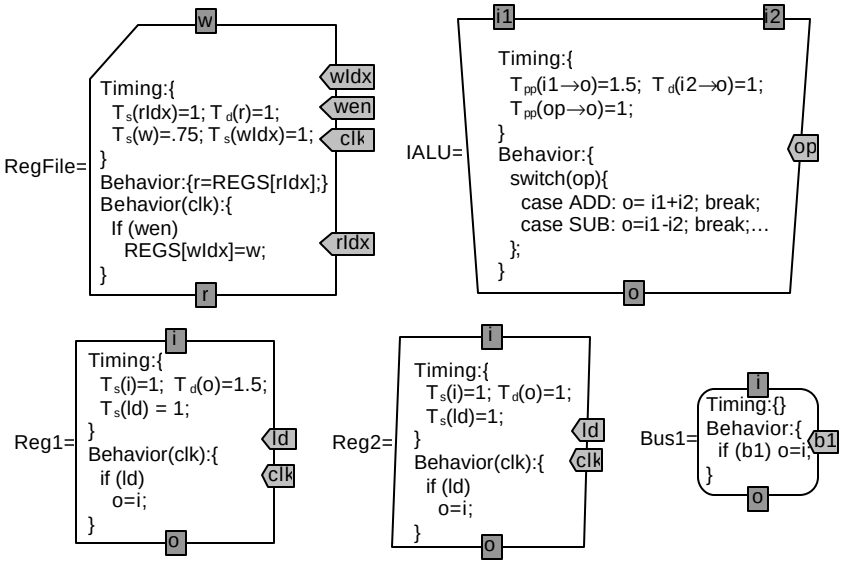

Figure 6 - Component database for Figure 5

In general, the components of the connectivity graph are selected from a component database which defines the *ports*, *timing*, *control bits*, *behavior* and *attributes* of each component type. The component attributes are mainly used by the compiler to map and bind the application operation and variables to the components of the connectivity graph. Examples of these attributes are: typed operations, such as unsigned addition, singed addition; typed storages, such as floating point storage, short integer storage. The behavior of a component describes the data transfers and operations between the ports of that component. The behavior is used by simulator to execute the sequence of control words generated by compiler.

The control bits define a mapping between the attribute values and the actual values of control ports of a component. This information is used for generating the control words after compilation and decoding the control words during simulation.

The timing information of the components is used: by a data path analyzer to extract information such as the critical path; by compiler to determine number of required clock cycles for a register transfer; and by simulator to validate the timings and evaluate the performance. The timing information is represented in four different forms:

- Setup time $T_s(x)$ for input or control ports. It shows the amount of time that the data must be available on ports *x* before the clock edge.
- Output Delay $T_d(x)$ for output ports. It shows the amount of time after the clock edge that it takes for the value to appear on output port *x*.
- Port-to-Port Delay $T_{pp}(x \rightarrow y)$ from input or control ports to output ports. It shows the amount of time that it takes for the effect of the value of input or control port *x* to appear on the output port *y*.
- Sequence relation or time difference $T_{dif}(x<y)$ between any two ports. It shows the amount of time that an event on *y* should or will happen after an event on *x*. Note that all other timing forms can be represented with T_{dif}. For example, $T_s(x) = T_{dif}(x<clk)$ and $T_d(x)=T_{dif}(clk<x)$. However, the other forms carry more semantic information that can be used by the tools.

The structure of ports (and hence the interpretation of timing information) for each component depend on its type. We define seven types of components as follows:

1. *Register* is a single-storage clocked component that has at least one input port and one output port.
2. *Register File* is a multi-storage clocked component. For each input or output port, there is at least one address control port whose value determines which internal storage element is read by the corresponding output port or is written by the corresponding input port. Usually, for each input port a single bit control port enables the write. Writes are always synchronized with the clock, i.e. the address and input port value are used to update the internal storage only at the edge clock. Therefore, for input ports and their corresponding address control ports a setup time, T_s, must be defined. The reads from register file are independent of the clock and any time the value of a output address changes, the value of the corresponding internal storage appears on the output port. Therefore, for an output port *r* and its corresponding control port *rIidx*, $T_{pp}(rIidx \rightarrow r)$ must be defined. Other control ports may be used for enabling/disabling the reads or writes.
3. *Bus* is component that has multiple input and control ports but one output port. It does not have any internal storage and therefore is not clocked.
4. *Memory* is also a multi-storage component similar to Register File. However, it usually works as a combinational logic (clock independent) and may use only one address control port to determine the internal storage to be used for read or write. Typically, a chip-enable control signal enables/disables the memory and a read-write control signal determines the behavior of the memory. The timing protocol of memory can be captures through several T_{pp} and T_{dif} timings.
5. *Computational Unit* is a combinational unit that performs an operation on the input data and produces an output. It may have control signals for determining the type of its operation.
6. *Pipelined Computational Unit* is similar to the Computational Unit but has a specific number of internal pipeline stages. Therefore, instead of having port-to-port delays, a setup time is defined for the input and control ports relative to the first internal pipeline stage and an output delay is defined for the output ports relative to the last internal pipeline stage.
7. *Compound* component is an abstraction of a connectivity graph that can be used as a component in another connectivity graph. By using this component, the model can support hierarchical designs. The ports, timing and behavior of this component are defined by its corresponding connectivity graph. Before any processing in the tools, these components may be expanded and the connectivity graph if flattened.

To enable further customization of the data path towards the characteristics of the application, the NISC data path model should also support multi-cycle and pipelined data paths to provide faster and better performance. Pipelining not only improves the resource utilization but also can help reducing the clock period. By supporting multi-cycle component, the clock period is no longer limited by the critical path. In the next sections, we describe how these features are captures and review their possible effects on the simulator and compiler.

4.2 Multi-cycle Component support

A multi-cycle component takes more than one clock cycle to generate its output. This means the clock frequency is not necessarily bounded by the critical path. In the NISC processor model, whenever the clock period is less than the delay of component, that component should be treated as

multi-cycle. Therefore, no special representation is needed for such components in the model of data path. Whenever a register transfer involves a multi-cycle component, the NISC compiler must expand the corresponding control bits over multiple cycles during the generation of control words. In this way, the control words guarantee the correct execution and therefore the simulator does not need to handle multi-cycle components different than the normal ones.

To understand this concept better, consider the following example based on data path of Figure 5. Assume the operation $x=y+z$ is translated to $o1=ALU(+, a1, a2)$ register transfer which takes four units according to the connectivity graph of Figure 5. If the clock period is at least 4 units, then the whole operation can be done in one cycle and the control word would be:

```
clock n:    o.load = 1; ALU.op = +;
```

Now assume that the clock period is only two units. For a correct execution, the control signals of the *ALU* must remain unchanged during two consecutive cycles and the result must be loaded in the output register at the end of the second cycle. Therefore the control word would be:

```
clock n:    ALU.op = +;
clock n+1:  o1.load = 1; ALU.op = +;
```

A multi-cycle component can be pipelined. This means that the control and input values need to remain stable only during one cycle but the output will be generated after multiple cycles. Assume that the ALU component is one stage pipelined and $T_s(i1)=0.5$, $T_s(i2)=0$, $T_d(o)=1$. With a clock period of two units, the control word would be:

```
clock n:    ALU.op = +;
clock n+1:  o.load = 1;
```

All of these cases can be simply detected by comparing the clock period with the timings of data transfers captured in the connectivity graph.

4.3 Pipelined data path

Pipelining is a powerful parallelizing feature that inevitably appears in any modern processor. The pipeline in a NISC processor is divided into two parts. The initial part of the pipeline is in the controller and only fetches the control words from the memory. This part of the pipeline is very simple and almost identical in all NISC processors. The second part of the processor pipeline resides in the data path. The flow of data in the data path pipeline is solely controlled by the fetched control word. In connectivity graph of the data path, the stages of the pipeline are represented by registers and pipelined computational units. Since the control words describe the complete behavior of the data path pipeline, the simulator can easily handle any complex pipeline structure by simply applying the control bits to the components and simulating individual component behaviors. However, the compiler must generate the exact and detailed behavior of the pipeline and therefore, the complexity and the structure of the pipeline have a direct effect on the complexity of the algorithms used in the compiler. In Section 6, we briefly describe what the compiler needs to do to handle the pipeline.

5 Simulation

The simulator is used for both validating the output of the compiler and evaluating the performance of the NISC processor. The output of the compiler has both timing and functionality information that must be validated. Validating the timing of register transfers can be statically done using the connectivity graph and comparing its information against the value of control bits in the control words. Therefore, the simulator needs only to validate the functionality of the output of the compiler. On the other hand, since each control word represents the behavior of the processor in

one clock cycle, the number of executed control words is equal to the number of clock cycles that the execution takes. In this way, a functional simulation of the control words also represents the performance of the processor.

Considering the general structure of a NISC processor shown in Figure 3, the main loop of the simulator is very simple and straight forward. Figure 7 shows the pseudo code of the simulation main loop. In order to calculate the next value of PC, the first part of the loop executes the functionality of the "Next Address Generation" unit using the content of control word and the feedback from data path. Then the data path pipeline is executed using the contents of the current control word. After the next control word is fetched from memory, the value of PC is set to the calculated new address.

```
ControlWord cw = 0;
while (not end of program)
{
  nPC = calculate next PC from cw and data path feedback;
  execute ( cw );
  cw = Mem[PC];
  PC = nPC;
}
```

Figure 7 - Pseudo code of the simulation main loop

The main part of the simulator is a big function that decodes the control word and performs the corresponding register transfers in the data path. To generate this function, the flattened connectivity graph is traversed in reverse topological order to construct the register transfer blocks. A register transfer block is a sequence of operations that start with one or more reads from storage components and ends with only one write to a storage component. The register transfer blocks are generated in reverse topological order to ensure all storages are read before they are written. Figure 8 shows the simulation main function of the data path of Figure 5. In this pseudo code, the control word *cw* contains all of the control bits of the components of the data path. For example, *cw.a1.ld* is the control bit of the control word that is connected to the *ld* bit of component *a1.* During the traversal of the connectivity graph, the behavior of each component is copied from its corresponding component type. In the second register transfer block of Figure 8, the first, third and fifth lines are the behaviors of the *RegFile, Bus1* and *Reg1* from Figure 5(c) respectively. Also, for every edge in the connectivity graph, an assignment is generated in the simulation function. For example, in the second register transfer block of Figure 8, the second and forth lines represent the *RF.r*→*B.i* and *B.o*→*a1.i* edges of the connectivity graph of Figure 5(b) respectively.

```
void execute(ControlWord cw)
{
  //Register transfer Block 1
    ALU.i2 = a2.o;
    ALU.i1 = a1.o;
    switch (cw.ALU.op)
    {
      case Add: ALU.o = ALU.i1 + ALU.i2; break;
      case Sub: ALU.o = ALU.i1 - ALU.i2; break;
      //…
    }
    o1.i = ALU.o;
    if (cw.o1.ld) o1.o = o1.i;

  // Register transfer Block 2
    RF.r = RF.regs[cw.RF.idx];
    B.i = RF.r;
    if (cw.B.b1) B.o = B.i;
    a1.i = B.o;
    if (cw.a1.ld) a1.o = a1.i;

  // Register transfer Block 3
    //other part of the data path that writes into a2
    a1.i = …
    if (cw.a2.ld) a2.o = a1.i;
}
```

Figure 8 - Main simulation function for data path of Figure 5

The generated code for the *execute* function in the simulator is compiled and optimized by a conventional C or C++ compiler. During this optimization, each register transfer block is reduced to manipulation of register data only. Figure 9 shows the optimized version of Figure 8.

```
void execute(ControlWord cw)
{
  //Register transfer Block 1
    if (cw.o1.ld)
      switch (cw.ALU.op)
      {
        case Add: o1 = a1 + a2; break;
        case Sub: o1 = a1 - a2; break;
        //…
      }

  // Register transfer Block 2
    if (cw.a1.ld && cw.B.b1)
      a1 = RF.regs[cw.RF.idx];

  // Register transfer Block 3
    //other part of the data path that writes into a2
    if (cw.a2.ld) a2 = …;
}
```

Figure 9 - Optimized main simulation function for data path of Figure 5

6 Compilation

To execute an application on a given data path, the compiler must translate each operations of the application to sequences of register transfers by generating proper control words for each cycle. As in a typical synthesis tool, the compilation starts by converting the operations in application to three address operations. Then the typical phases of register allocation, operation binding and bus selection are performed.

Register allocation can be done automatically or manually by the designer. The typical approach is to map all of the variables of the application to a register file or a scratch pad. While this is a straight forward approach, it can limit or prevent the utilization of features such as operation chaining and feedback paths. As an alternative, it is possible to map only the inputs and outputs of the basic blocks to the register file or scratch pad and let the compiler decide where the intermediate values in basic block should be stored. This approach can reduce the register pressure and significantly improve the quality of the results.

After knowing where the data are stored (register allocation), the operation binding phase determines what unit in the data path must be used to perform the required operation on the data. Among the available units that perform a specific operation, the selection can be done based on the unit's performance, power consumption or any other desired criteria.

Finally, if the allocated registers that store the data and the selected unit that perfoms the operation are not connected directly, the data must be moved from the registers to the unit via one or more busses. This is done in the bus selection phase.

The output of the compiler is a sequence of register transfers scheduled at different clock cycles. The set of register transfers that happen in the same cycle determine the value of control bits in that cycle.

As we mentioned earlier in Section 4, compiling simple operations and supporting multi-cycle components is straight forward and is done by finding data transfer paths in the connectivity graph and generating the control words by comparing the timings of the selected path with the clock cycle period. In the rest of this section, we briefly mention the issues that are involved in handling the data path pipeline in the compiler.

Since the data are read at the beginning of a pipeline path and the processor status (register file, memory and status bits) is updated at the end of it, the register transfers must be carefully scheduled to make sure the data and control dependencies are satisfied. For any pipeline structure, data dependency can be handled by fairly simple algorithms. However, handling control dependency especially in presence of speculative execution requires extra care and the complexity of corresponding algorithms vary depending on the structure of pipeline.

In a pipelined processor, there is always a delay between issuing a branch command and updating the PC. This is delay, known as branch delay, is because of: first, delay of calculating the target address; and second, delay of calculating the condition for conditional branches. In the simplest case, the control words that are executed during the branch delay should do nothing (equivalent to NOP in normal processors). To mask the branch delay, either the compiler moves some operations after the branch (branch slot) or the processor predicts the target address and executes the operations speculatively (branch prediction). In case of branch prediction, only the operations that do not change the processor status can be executed speculatively.

In a *uniform* pipelined data path, all pipeline paths in the processor have the same length and the branch delay is similar for all operations. In this way branch handling is significantly simplified

and the compiler performs a fixed algorithm irrespective of the current schedule of pipeline paths. Figure 10 shows a NISC processor with a uniform pipelined data path. In this processor, comparison, ALU operation and multiplication finish in the same pipeline stage. Because of the pipeline in the controller, a non-conditional branch takes at least one cycle to be resolved while a conditional branch will take four cycles to resolve. If the address generation uses a branch prediction, the first four control words (parallel operations) of a basic block must not update the processor status, i.e. they must not write to register file or memory. In absence of branch prediction, for unconditional branches one NOP and for conditional branches four NOPs must be inserted after the branch.

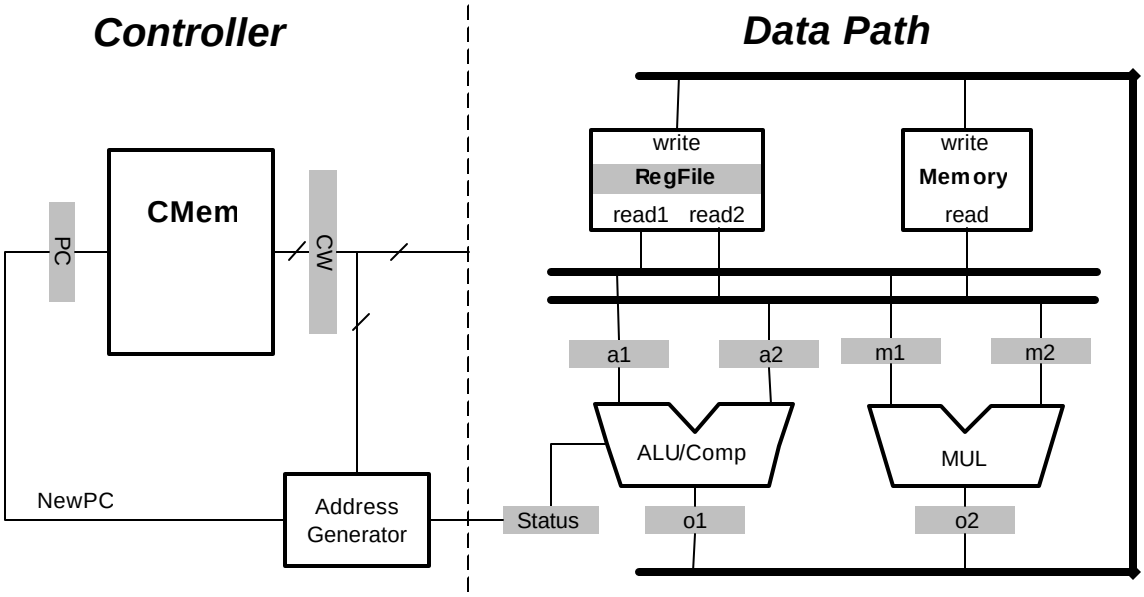

Figure 10 - NISC with uniform pipeline paths

A very general pipelined data path may contain multi-cycle units with different delays or different number of pipeline stages in different pipeline paths. We refer to this general case as *heterogeneous* pipelined data path. In a heterogeneous pipelined data path, the branch delay depends on the controller pipeline and the pipeline paths that update the *status* register. Therefore, branch delay of a conditional branch depends on the path that has been selected for calculating the condition and updating the status bits. For example, in Figure 11 two different paths with different delays can update the *status* register. The compiler may schedule a condition on any of these paths depending on the availability of the resources. In other words, the branch delay depends on the schedule and is not fixed any more. The number of NOP operations or the number of initial operations of a basic block that can be speculatively executed depends on their delay as well as the schedule of previous conditional operations.

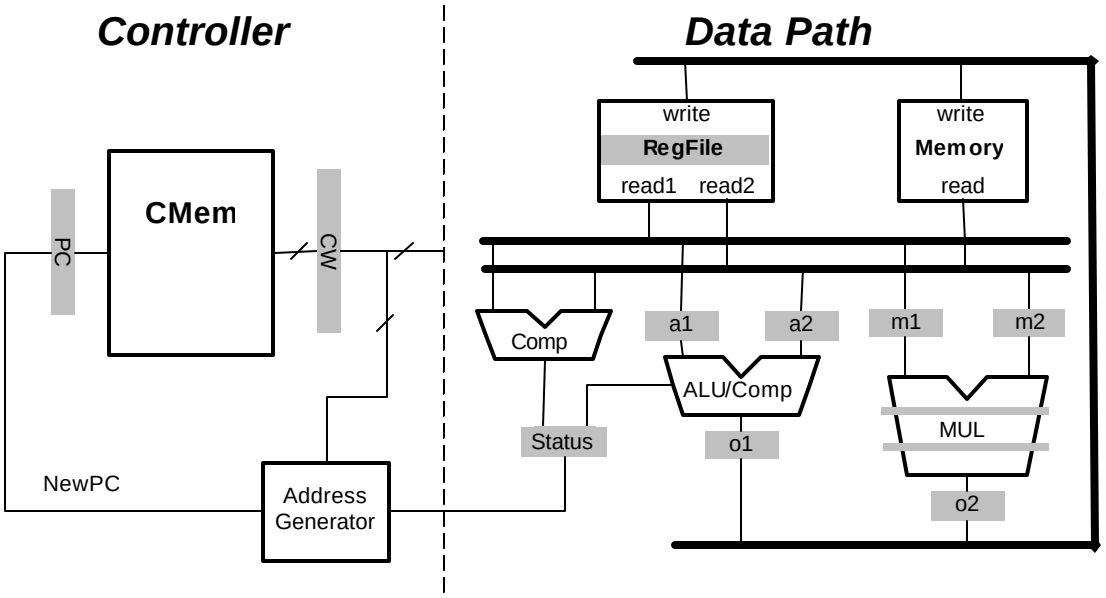

Figure 11 - NISC with heterogenous pipeline paths

In this section we presented the problems and issues that must be addressed in the compiler. The suggested approaches are included in this report to show that the information captured by the model are sufficient for addressing the issues involved in compilation of applications for a NISC processor.

7 Conclusion and Future work

In this report, we elaborated the concept of NISC and compared it with normal processors. By removing the instruction interface in NISC any application characteristics can be utilized to generate the most optimized possible data path for executing that application. Also, a single set of supporting tool chain; such as compiler and simulator; are enough to map any application to any customized (and proper) data path. To achieve this goal a general model is necessary that can capture any data path characteristics and can be used by simulator and compiler.

NISC data path is captured by a connectivity graph whose nodes represent the components of the data path and edges show directed connections between the ports of the components. The components of the graph are selected from a database of component types. We showed that this model is general enough to capture advanced features such as multi-cycle components and pipelined data paths. We also showed how information of the model can be used to generate a simulator and to address the issues involved in compilation.

Our work will focus on mainly three areas:

- Modeling: we have done preliminary investigation on the proposed model in this report to make sure it can handle general cases and can provide the necessary information for simulator and compiler. More extensive analysis will be done in the future to make sure that the proposed model can capture all the necessary information of complex real life components.

- Simulation: As described in the report, the functional and cycle accurate simulators will be the same for a NISC processor. Therefore, it is important to have a fast simulator and further research will focus on developing high performance simulation techniques for NISC.

- Compilation: the NISC compiler must translate the operations of the application to a sequence of control words based on the model of data path (captured in the connectivity graph). This compiler may use techniques from both the traditional compilers and the synthesis tools. In long term, the following optimizations and techniques must be studied and implemented in the compiler:
    - Multi-cycle component support which will remove the upper bound on the clock cycle.
    - Better register allocation. Since there is more control over data path of NISC, the variables can be assigned to storages other than register file. For example, input/output registers of the components in the pipeline can be used to store the temporary values and improving the performance by chaining the operations.
    - Heterogeneous pipeline support to improve performance through better customization of data path. Heterogeneous pipelines require more complex algorithms mainly for handling the flow of the program, i.e. branch operations.
    - Speculative execution support to improve the utilization of resources and improving the performance.

The research on simulator and compiler will also help us verifying information and structure of the model and improving it, if necessary.

8 References

[1] D. Gajski, "NISC: The Ultimate Reconfigurable Component," CECS Technical Report 03-28, October 1, 2003.

[2] http://www.dspvillage.ti.com TMS320C6000 CPU and Instruction Set Reference Guide.