Interrupt and Low-level Programming Support for Expanding the Application Domain of Statically-Scheduled Horizontal-Microcoded Architectures in Embedded Systems

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

Abstract

The increasing role of software in the embedded systems has made processor an important component in these systems. However, to meet the tight constraints of embedded application, it is often required to customize the processor for the application. Customizing instruction-based processors is difficult and very challenging. Design approaches based on statically-scheduled horizontal-microcoded architectures have been proposed to simplify the architecture customization. In these approaches, first the datapath is specified by the designer, and then the operations of the datapath are extracted automatically. Since the operations are statically scheduled in these architectures (i) low-level programming using assembly is impossible or very tedious; and (ii) execution of programs cannot be interrupted arbitrarily. In this paper, we address the above problems. We show how to efficiently handle interrupts in such architectures and also propose an elegant way of controlling low-level hardware resources in a general way in C language. We also show that after adding interrupt and low-level programming we could use the above architectural style in a multi-core system to implement a complete MP3 decoder that can process 122 frames per second while the standard requirement is 38 frames per seconds.

1. Introduction

Due to the productivity gain of using software in the design of embedded systems, processors are increasingly used in these systems. Embedded processors often run only one or a few applications in the life-time of the system. Therefore, they can be customized for the target applications and significantly improve the quality of the embedded system in terms of cost or other constraints such as performance, and power consumption. Instruction-based architectures limit the customizations because: (a) hardware designer is limited by instruction coding, size and complexity of the decoder; (b) compilers can support certain class of instructions and hence instructions cannot be very complex; and (c) manually updating compilers to incorporate the custom instructions is not practical and developing compilers that automatically utilize hardware customizations through new custom instructions is very complex.

An alternative design approach is to let an experienced ASIC designer specify the datapath of the processor and then automatically compile the program on the given datapath by explicitly controlling the machine activities. Based on this design approach, MIMOLA [1], TIPI [2], and NISC [3] use a statically-scheduled Horizontal Microcoded Architecture (HMA) style to maximize the explicit control of the programmer (or the compiler) over the datapath. In these approaches, the microcode is used to execute the program on the given datapath. In contrast, typically in traditional microcoded processors [4], [5], the microcode was used inside the processor to implement the instructions of the instruction-set. In other words, the instructions, rather than the microcode, would define the processor's external behavior seen by the programs. The instruction abstraction (a) enables backward binary compatibility, (b) simplifies low-level programming through assembly, and (c) defines fine-grained intervals where interrupts could be handled by the processor. By using microcodes instead of instructions all these benefits are lost. In embedded and custom processors, backward binary compatibility is not as important as it is in the general-purpose processors. However, interrupt and assembly programming are necessary features. For example, developing different communication protocols rely on interrupts and low-level access to the hardware.

In statically-scheduled pipelined architectures, different stages of execution of an operation (e.g. read, execute, write-back) are implemented with several micro-operations. The overlapping execution stages of different operations are combined in micro-instructions which determine the control-word (CW) for each clock cycle. As a result, execution of micro-instructions cannot be arbitrarily interrupted; otherwise, the interrupt routine may need to store/restore datapath registers in addition to the registers of the register-file. A safe and efficient interrupt mechanism is needed in statically-scheduled HMAs before they can be used in embedded systems. On the other hand, practical use of such architectures mandates that programs are written in a high-level and architecture-independent language (such as C). However, use of low-level assembly programming is inevitable in firmware code (e.g. device drivers). Since instruction abstraction is removed, an alternative approach must be developed that allows the low-level programming in such architectures. These issues have not been addressed in the past. Since MIMOLA does not support pipelined datapaths, interrupts does not impose a big challenge. On the other hand, the compiler of TIPI solves a Boolean satisfiability problem and can be used only for very small functional blocks. None of these approaches have considered interrupts. Low level programming in MIMOLA and TIPI is done directly with the microcodes. Since the target architecture is statically scheduled, the programmer must manually schedule the microcodes as well. This is a very tedious and error prone task.

In NISC design approach, a cycle-accurate compiler uses a mixture of standard compiler and High-Level-Synthesis (HLS) algorithms to generate the control signals of a given datapath in every clock cycle. Therefore, the architecture designer focuses only on datapath design and provides the netlist of the datapath components along with timing, cost and other attributes of components as an input to the compiler. The compiler, then maps the application directly on the given datapath. Experiments on several embedded and real life applications have shown [6] that NISC can perform on the average 5 times better than a RISC processor while having only 15% larger code size on average. The NISC design tools and sample architectures are publicly available at [7]. In this paper, we focus on two problems: (1) adding interrupt support to the architecture and the tools; and (2) enabling low-level programming (similar to assembly) in C language. After adding interrupt and low-level programming,

978-3-9810801-2-4/DATE07 © 2007 EDAA

we developed two multi-core systems to implement an Mp3 decoder using NISC approach. The final system could process 122 frames per second while the Mp3 standard requirement is 38 frames per second.

The rest of this paper is organized as follows: Section 2 presents an overview of NISC. In Section 3 we explain how interrupt handling is added to NISC, In Section 4 we explain our solution for supporting low-level programming in C language. Use of NISC in two multi-core systems is shown in Section 5. Section 6 presents the related work and Section 7 concludes the paper.

2. NISC overview

In NISC design approach, the target architecture is composed of a pipelined datapath and a pipelined controller that drives the control signals of the datapath components in each clock cycle. The datapath can be simple or as complex as datapath of a processor. The controller has a fixed template and is usually composed of a Program Counter (PC) register, an Address Generator (AG) and a Control Memory (CMem). The control values are stored in a control memory. For small size programs, the control values are generated via logic in the controller. The NISC *cycle-accurate compiler* generates the control values that define what the datapath should do in every clock cycle. Figure 1 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. In presence of controller pipelining (i.e. CW and Status registers in Figure 1), the compiler should also make sure that the branch delay is considered correctly and is filled with other independent operations. Compilation algorithm detail is presented in [3] and [8].

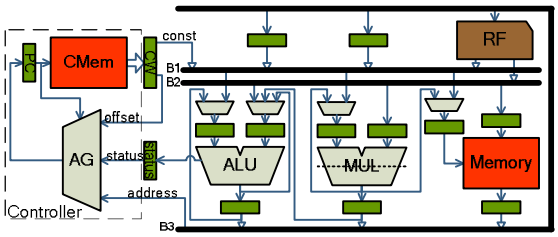

Figure 1- NISC architecture example.

Figure 2 shows the NISC flow of designing a custom architecture for a given application. The datapath can be generated (allocated) using different techniques. For example, it can be an IP, specified by the designer, reused form other designs, or generated automatically by algorithms similar to HLS. The datapath description is captured in a Generic Netlist Representation (GNR)[9]. A component in datapath can be a register, register-file, bus, multiplexer, functional unit, memory etc. The program, written in a high level language such as C, is first compiled and optimized by a front-end and then mapped 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 that is used for simulation and synthesis. After synthesis and Placement and Routing, the accurate timing, power, and area information can be extracted and used for further datapath *refinement*. For example, the user may add functional units and pipeline registers, or change the bit-width of the components and observe the effect of modifications on precision of the computation, number of cycles, clock period, power, and area. In NISC, there is no need to design the instruction-set because the compiler automatically analyzes the datapath and extracts possible operations and branch delay. Therefore, the designer can refine the design very fast.

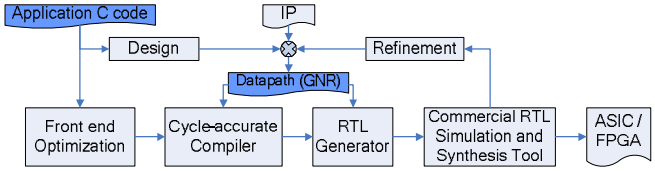

Figure 2- NISC flow.

While MIMOLA and TIPI focus only on single-cycle operations, in NISC operation chaining (sub-cycle operations) and multi-cycle operations are also supported. 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 compiler eventually schedules these control values based on their timings and the given clock period of the system. Therefore, the compiler has much more low-level control over the datapath and hence is closer to a synthesis tool in terms of capability and complexity.

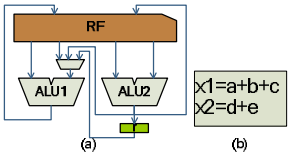

Figure 3- (a) Sample datapath, (b) sample code.

Consider the datapath of Figure 3(a) that is used to compile the set of expressions shown in Figure 3 (b). Depending on the clock frequency of the system and the delay of the components, the compiler can choose to chain two operations in one cycle or execute one operation over multiple cycles. Assume that clock period of the system is T, delay of *ALU1* is $d1$, and delay of *ALU2* is $d2$. Also assume that *ALU2* is slower but consumes less power ($d1 < d2$). Depending on the values of T, $d1$, and $d2$ three cases are possible:

- If $d1 < T$ and $d2 < T$ but $T < d1+d2$, then each operation must be scheduled in one cycle and intermediate data must be stored in the register-file or datapath register r (Figure 4(a)).
- If $d1+d2 \leq T$, then two operations can be chained in one cycle and register-file is accessed only once for writing back the final results (Figure 4(b)).
- If $d1 < T < d2$, then the faster *ALU1* can be used to execute two operations in two consecutive cycles while the slower *ALU2* executes the other operation in two cycles (Figure 4(c)).

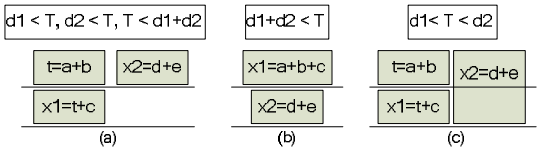

Figure 4- (a) single-cycle, (b) chained, (c) multi-cycle operations.

As this example illustrates, in NISC the datapath can be utilized very efficiently because the compiler has complete control over it. While instruction-set based compilers are mainly concerned with performance, the NISC compiler can also consider other design parameters such as timing and power consumption of individual datapath components. However, as mentioned before, this architectural style introduces new challenges for supporting interrupts and low-level programming.

3. Adding interrupt handling to NISC

In traditional processors, the interrupt is checked between every two instructions. The execution flow can be interrupted between instructions because all instructions store their result back to the

register-file. Therefore, the interrupt routine may only need to store/restore the value of registers in the register-file in its prolog/epilog.

In NISC, the intermediate results of operation may be stored in the internal registers of the datapath. Furthermore, an operation may take more than one cycle (see Section 2) and hence span across multiple CWs. Therefore, in NISC the execution flow cannot be interrupted between any two arbitrary CWs. Detecting the dependencies between CWs at run time is very difficult (if not impossible). Also, in addition to the registers of the register-file, an interrupt routine may need to store/restore the intermediate registers of the datapath as well.

To address this problem, we need to find an easily identifiable location in the program where execution flow can be safely interrupted. The boundary of basic blocks is a good candidate for this purpose. A basic block is a sequence of operations that always execute together. The execution sequence of basic blocks of the program is data or control flow dependent. Consequently, every basic block *must* read its inputs from memory or register-file and *must* write its outputs back to memory or register-file. In other words, since execution of operations of a basic block cannot depend on the intermediate datapath values of other basic blocks, the interrupt can be safely serviced at the end of basic blocks. In fact, one of the goals of NISC is to execute each basic block *as if* it was executed with one custom instruction.

We modified the controller of NISC to check for interrupts only when bits corresponding to jump operations are set, i.e. at the end of basic blocks. After a jump operation, the execution flow goes to the target of the jump or an interrupt routine. In presence of an interrupt, the target of the original jump is passed to the interrupt routine as its return address. Note that this scheme also simplifies the implementation of atomic functionalities because the programmer can now count on atomic execution of basic blocks.

The only concern is that servicing the interrupt only between basic blocks may increase the overall interrupt service delay if the basic blocks are very large. There are two contributing factors to the interrupt service delay: (1) interrupt latency, i.e. the time between when the interrupt is activated and when the execution flow is transferred to the interrupt service routine; and (2) the delay of interrupt service routine (ISR), i.e. the time it takes to execute the code in the ISR.

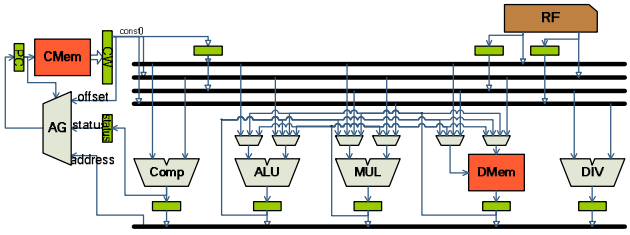

Figure 5- Architecture used for analyzing size of basic blocks.

In our proposed approach, the size of basic blocks in the running application can affect the interrupt latency. To examine this effect, we ran a series of embedded benchmarks on a generic architecture (GA) shown in Figure 5. The benchmarks include *qsort*, *dijkstra*, *sha*, *adpcm.coder*, *adpcm.decoder* and *crc32* from MiBench (the free version of EEMBC embedded benchmarks available at [10]), and a fixed-point Mp3 decoder (more than 10,000 lines of C code available at [11]). We generated the RTL Verilog code of the design and used Xilinx ISE 8.1 toolset for simulation and synthesis of the results. We synthesized the GA (Figure 5) on a Xilinx Virtex4 (90-nm) FPGA package and achieved a clock frequency of 80 MHz. The Xilinx toolset also provides a soft-core 32-bit RISC processor (MicroBlaze) that is already optimized Xilinx technology. On a Vertix4 FPGA package, MicroBlaze runs at 105 MHz. MicroBlaze core comes with specific fine-grained timing constraints that direct the synthesis tool to achieve the highest possible clock frequency. For synthesizing GA we only used a general clock constraint and we expect that the clock frequency of GA can be further improved by using more specific constraints. In any case, the achieved 80 MHz clock frequency for GA seems to be reasonable enough to be used in our calculations.

Figure 6 shows the distribution of number of basic blocks that take less than 100 clock cycles to execute. The first column in this figure shows the number of basic blocks that take 0 to 9 cycles to execute; the second column shows the number of basic blocks that take 10 to 19 cycles, and so on. It is clear that in these benchmarks, the majority of basic blocks take between 10 to 30 cycles. In other words, if we service interrupts in between basic blocks, most of the time the interrupt latency will be less than 0.5 μ sec (=50 cycles / 80 MHz).

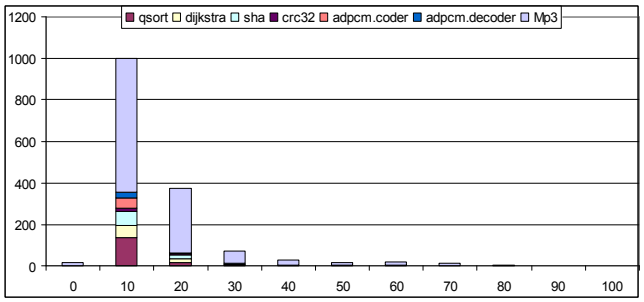

Figure 6- Distribution of basic blocks shorter than 100 cycles.

Figure 7 shows the distribution of number of basic blocks that are longer than 100 cycles. Overall, there are 13 basic blocks in all of the benchmarks that are longer than 100 cycles. In general, although large basic blocks are rare in applications, in cases where interrupt delay is critical, the compiler can break large basic blocks into a sequence of smaller blocks whose size is determined by the frequency of the interrupts or the upper bound of their delay. Note that large basic blocks are typically the result of techniques that improve the operation-level parallelism of the code. The compiler can break large blocks into smaller ones after or during operation scheduling without negatively affecting the utilization of parallelism. Compiler can also enable interrupt handling after fall-through basic blocks (not ending with a jump) by adding a jump to the next block.

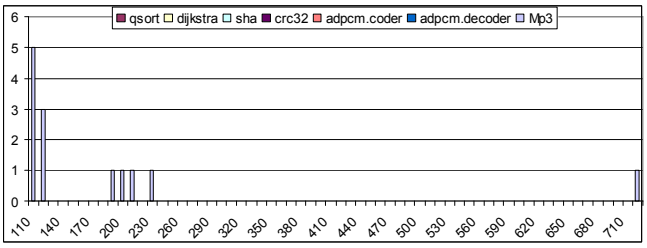

Figure 7- Distribution of basic blocks longer than 100 cycles.

A more important factor in servicing interrupts is the ISR execution delay. We ran the aforementioned benchmarks on both MicroBlaze and GA to compare their performance. On average, the benchmarks ran 5 times faster on GA than MicroBlaze. We believe the performance of a typical ISR routine benefits similarly from execution on GA. Additionally, in NISC, we can customize the architecture to further improve the performance of particular piece of code, including an ISR.

The above experiments show that by processing interrupts in between basic blocks, statically-scheduled architectures can handle interrupts almost as efficiently as their RISC counterparts.

4. Low-level programming in NISC using C

Languages such as C are generic enough to cover majority of the application needs, but sometimes in applications, the underlying hardware must be controlled directly through special registers or instructions. In instruction-based processors, programmers use assembly code to perform tasks such as peripheral IO operations, or configuring the interrupt unit. Since in NISC, the architecture has no predefined instruction-set, it does not have any assembly code either. This is specially limiting when an application requires interrupt or needs to communicate with other cores in a system. In statically-scheduled HMAs, use of microcode for low-level programming requires that the programmer also provide an accurate cycle-by-cycle schedule of the microcodes. This makes direct use of microcodes (a) tedious and error prone, and (b) impractical in C language.

To address this limitation, we added support for *pre-bound* functions and variables to the NISC compiler. These functions and variables have common C syntax but instead of implementing them in the normal way, the compiler maps them to specific hardware resources. During code generation, the compiler generates proper control bits to access their corresponding hardware resources. Note that pre-bound functions are different from intrinsic functions commonly used in the compilers. Pre-bound functions affect the functionality of the application but have no implementation and are treated similar to other operations. Therefore, they can be scheduled in parallel with other operations. On the other hand, the intrinsic functions are implemented in the same way as other normal functions, i.e. inlined or jumped to. But since the compiler has a built-in knowledge of how the intrinsic functions behave, it can optimize them more than normal code. Also, some intrinsic functions only provide hints to the compiler (e.g. for optimizations) but have no implementation or have no effect on the program.

The NISC tools use an XML base description format (called GNR) for capturing components and the netlist of the datapath. For each component, the ports, operations and their timing are captured in GNR. We described pre-bound functions for functional units the same way that their operations are defined. The description also maps the function output and parameters to the ports of the component and specifies the timing and corresponding control bit values. We also specify whether the scheduler can freely move the function and schedule it with other operations, or it should preserve the order of the function with respect to operations that appear before and after it in the code. For example, if we have a function pre-bound to a unit that calculates the minimum of two values, that function can be scheduled with other operations in the program.

Figure 8 shows the GNR code of an Interrupt Unit (IU) that has thee pre-bound functions, i.e. *setMask*, *clearInterrupt*, and *interruptNumber*. The component has a set of input, output and control ports. Function descriptions specify the mapping between their inputs/output and the input/output ports of the component. The description also determines the control values that must be assigned to corresponding control ports for execution of the function. The functions in this example indicate *stateDependency*="*all*". This means that the compiler must preserve the order of operations before and after these functions during scheduling. Figure 9 shows a sample C code for using the above pre-bound functions. After receiving an interrupt, the `interruptHandlerMain` function is called. In this function, first the current interrupt number is read and it is handled after masking all other interrupts. Finally, the corresponding interrupt is cleared and interrupts a re-enabled.

To support pre-bound functions and variables, we added a new tool, *PreboundCGenerator*, to the flow of Figure 2. The new flow is shown in Figure 10. Before compiling the application on the given datapath, the *PreboundCGenerator* tool processes the architecture description and generates a C header (.h) and source (.c) file that contains the declarations of the pre-bound variables and functions. For every register in the datapath (including registers in the register-file) a variable is declared in the generated source file, the function descriptions of the functional units are also translated to proper C function declarations. The tool also provides this information to the NISC compiler so that it knows which functions and variables are pre-bound to what hardware components. The generated source files are included in the application and the programmer can use them the same way they are normally used in C.

```
<FU type="InterruptUnit">
  <Params>
  <Ports>
    <Clock n="clk" bitWidth="1"/>
    <InPort n="reset" bitWidth="1"/>
    <CtrlPort n="clearCurrInterrupt" default="0" bitWidth="1"/>
    <CtrlPort n="loadMask" default="0" bitWidth="1"/>
    <OutPort n="interrupt2Controller" bitWidth="1"/>
    <InPort n="interrupts" bitWidth="(@INTERRUPT_COUNT)"/>
    <InPort n="i" bitWidth="(@BUS_WIDTH)"/>
    <OutPort n="o" bitWidth="(@BUS_WIDTH)"/>
  </Ports>
  <Annot_verilog>
  <Annot_compiler>
    <Functions>
      <Function n="setMask" delay="(@DELAY)" stateDependency="all">
        <Input port="i"><Type n="unsigned char"/></Input>
        <Ctrl port="loadMask" val="1"/>
      </Function>
      <Function n="clearInterrupt" delay="(@DELAY)" stateDependency="all">
        <Input port="i"><Type n="unsigned char"/></Input>
        <Ctrl port="clearCurrInterrupt" val="1"/>
      </Function>
      <Function n="interruptNumber" delay="(@DELAY)" stateDependency="all">
        <Output port="o"><Type n="unsigned char"/></Output>
      </Function>
    </Functions>
  </Annot_compiler>
</FU>
```

Figure 8- The GNR code for an Interrupt Unit (IU).

```
void interruptHandlerMain()
{
  int iNum = __$IU_interruptNumber();
  __$IU_setMask(0);
  //handling the interrupt
  __$IU_clearInterrupt(iNum);
  __$IU_setMask(-1);
}
```

Figure 9-Sample C code for using pre-bound functions of IU.

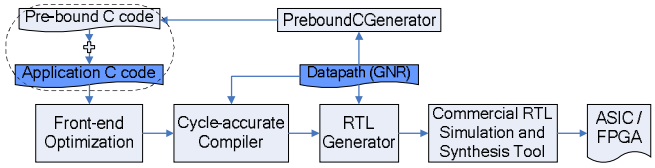

Figure 10- NISC tool flow with pre-binding.

During compilation, instead of binding variables to global memory, or stack, they are bound to their corresponding registers. Similarly, instead of implementing calls to pre-bound functions with *jump* operations, these calls are treated the same way that for example an *add* or *multiply* operation is treated. While providing similar capabilities, our pre-binding approach is more flexible than using assembly in instruction based processors. The pre-bound constructs have C syntax and can be merged with the rest of the application much easier than assembly code. Additionally, the programmer does not need to worry about the scheduling of these constructs.

In NISC, the main goal is to develop the application in an architecture independent high-level language (e.g. C) so that it can be mapped on different custom architectures. The benefit of our pre-binding approach is that a C code using pre-bound functions or variables can execute on any architecture as long as that architecture contains the corresponding hardware resources. Additionally, with this approach, the backward compatibility can be maintained at

source code level without imposing as tight constraints as backward binary compatibility requires.

5. Using NISC in a system

Typically in embedded systems, applications can be partitioned into parallel processes implemented by different components of a System-on-Chip (SoC). Adding pre-binding and interrupt to NISC and its tools makes it possible to use multiple NISC cores in a SoC. In order to facilitate communication between two NISCs, we designed a double-handshake bus protocol with proper communication interface (CI) unit [12]. The CI unit has two queues for send and receive, and it provides several pre-bound functions such as *push*, *pop*, *StartSend*, etc. Each function corresponds to a specific pattern on the control ports of the CI unit. These functions were described in the GNR format and used in the C code of the application. Similarly, we designed an interrupt handling unit (IU) and described it pre-bound functions for setting the interrupt mask, clearing the interrupt, reading the current interrupt number, etc. The IU and CI components where used to implement a message passing protocol on top of our double hand-shake bus.

In this section, we describe the implementation results of two multi-NISC systems for a fixed-point Mp3 benchmark downloaded from [11]. In general, an Mp3 audio file contains several frames. For a stereo file, each frame has two channels (i.e. left and right channels). In the Mp3 decoder, the frames go through three main phases, namely, *decode_frame*, *synthesis_frame* and *output_pcm*. Profiling the Mp3 decoder on the generic NISC architecture of Figure 5 showed that 63% of execution time is spent in *decode_frame*, 25% in *synthesis_frame*, 11% in the *output_pcm*. We realized that there are two approaches to parallelize the Mp3 application: (a) processing each channel separately, or (b) pipelining the phases. However, the Mp3 decoder was originally targeted for desktop PCs and separating the channels completely requires rewriting most of the code. Alternatively, we decided to separate the *synthesis_frame* phase for each channel because it required minimum code modifications. Such partitioning can reduce the execution time of *synthesis_frame* to half and hence can at most improve the performance by 12.5%. As for the second system, we pipelined the application into two stages where the first pipeline stage implements *decode_frame* phase and the second stage implements *synthesis_frame* and *output_pcm* phases. In this approach, processing delay of one frame is expected to increase due to the communication overhead. However, since the *decode_frame* of one frame is overlapped with the *synthesis_frame* and *output_pcm* of another frame, the overall performance can be improved by up to 36% (=min(63, 25+11)).

Table 1- Area and clock frequency of MicroBlaze and GA

Processors	Clock freq.(MHz)	Area (gates)	#cycles for 1 frame	speedup
MicroBlaze	105	39574	8,861,336	1
GA	80	35632	897,452	7.28
multi-core GA	80	73046	-	-

We implemented the Mp3 decoder on a MicroBlaze, a single GA, and two multi-core configuration of GA. Table 1 shows the clock speed and area of each architecture as well as their performance for decoding one frame of audio. For simulating the Mp3 decoder, we used the *scope1.mp3* (44.1KHz, 96kbit/s, stereo) available at [13].

Table 2- Throughput of three Mp3 implementations

Systems	#cycles for 1 frame	speedup for 1 frame(%)	#cycles for 25 frames	speedup for 25 frames (%)	frames/sec
SingleCore	897,452	0.00	22,800,961	0.00	88
Coprocessor	803,357	10.48	20,205,994	11.38	99
Pipelined	917,204	-2.20	16,433,655	27.93	122

Table 2 shows the results of implementing the Mp3 decoder in three configurations. The second and fourth columns show number of cycles for processing one frame, and 25 frames in each configuration and the third and fifth columns show the respective speedups. Figure 11 shows the block diagram of the three implementation configurations. Figure 11 (a) shows the *SingleCore* configuration in which the entire Mp3 decoder runs on one GA. Figure 11 (b) shows the *Coprocessor* configuration in which the Mp3 decoder runs on two GAs. In this case, one of GA acts as a coprocessor for the main GA and runs the *synthesis_frame* phase for left channel while the main GA runs the same phase for the right channel. The main GA also runs the other phases for both channels. The total performance improvement in this case is 10.48% which is close to the expected 12.5%. For each channel, the main GA sends 1152 words to the coprocessor GA and then receives 1152 words from it. The communication overhead is responsible for the 2% performance loss from the expected upper bound, i.e. 12.5%. Figure 11 (c) shows the *Pipelined* configuration, where one GA runs the *decode_frame* of both channels and send 2×1152 words to the second GA to perform *synthesis_frame* and *output_pcm*. In this configuration, the processing time for a single frame is increased by 2% but the overall throughput of the system is increased by 28%. Similarly, the communication overhead is responsible for the 8% performance loss from the expected upper bound, i.e. 36%. The communication overhead in the *Pipelined* configuration has increased because of the extra synchronization which was not necessary in *Coprocessor-Sys* configuration.

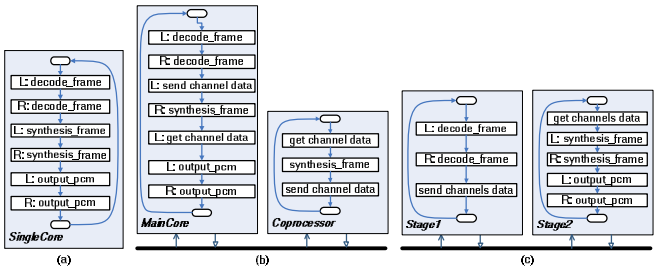

Figure 11- Implementing Mp3 with (a) SingleCore, (b) Coprocessor, and (c) Pipelined cores,

According to the Mp3 standard, at least 38 frames must be played per second. MicroBlaze can only run 12 frames per second. The last column of Table 2 shows the throughput of the tree configuration. Clearly, this throughput is much more than what the standard required. To save power, the SingleCore and Coprocessor configuration can run with half their clock frequency. The clock frequency of the Pipelined system can be reduced by two thirds while still meeting the throughput constraints of the standard.

6. Related work

Before RISC processors become popular, microcode processors [5] were extensively studied for several years. Today microcodes are mainly used inside processors for implementing complex instructions or for controlling programmable coprocessors such as ARM OptimoDE [14], [15]. In these cases, handling interrupt or low-level programming in a high-level language has not been an issue. This is because processors have instructions and coprocessors do not need these features.

Many retargetable design approaches ([16], [17]) have proposed techniques that generate software development tools from the description of instruction-set of the architecture. These approaches abstract out the architectural implementation details. Those who

attempted to generate the architecture from the description resulted in poor quality implementations. For example, LISA [18], and Target Chess [19] compilers use a behavioral instruction description mixed with structural architecture information and mainly focus on code generation and simulation. Absence of implementation details in the input description of these techniques degrades the quality of their recently added HDL generation. However, as in any other instruction based processor, handling interrupts and assembly programming is not a problem in these approaches.

An alternative approach is to describe the architecture structure and automatically extract the ISA. In the MIMOLA project [1] the RECORD [20] compiler extracts the behavioral model of instructions from MIMOLA HDL and targets a horizontal microcode machine with single-cycle operation. The MIMOLA HDL describes both datapath and the instruction decoder (controller). 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. This approach was suitable for architecture implementation but had two drawbacks: (a) they did not support pipelined datapaths or multi-cycle units, and (b) the designer had to describe the controller explicitly. Interrupt handling did not impose any challenge in this architecture because they did not support pipelined datapaths. However, to use low-level programming, the programmer had to use the microcodes and manually schedule them.

Similar to MIMOLA, the TIPI (Tiny Instruction-set Processors and Interconnect) [2] targets statically-scheduled HMAs with single-cycle instructions. The main difference is that instead of relying on specification of the controller, the TIPI uses the speciation of non-deterministic atomic actions on architectural state and outputs. While MIMOLA uses binary decision diagrams (BDDs) [21] to extract the valid instructions, in TIPI they extract the instruction-set as a set of *operations* and *conflict table* from the programmability constraint descriptions using Boolean satisfiability (SAT) algorithm. Cycle-accurate simulator and HDL generation from TIPI has been reported. Currently, TIPI does not have a compiler and all programming must be done manually. Also, interrupt support has not been addressed or considered.

7. Conclusion

NISC is viable option for implementing embedded applications. However, to use it in practical situations (a) it must support interrupts, and (b) it must provide a mechanism for low-level programming. NISC has no predefined instruction-set (hence no assembly) and instead executes the program using very tightly coupled control words generated by the compiler. We showed that by adding pre-bound function and variables to the NISC and its tools, NISC can support low-level programming in C. These functions and variables are mapped directly to the hardware resources by the compiler. We also showed that the interrupts can be safely serviced between basic blocks. After adding these features, we used NISC in two SoC designs to implement an Mp3 decoder. The generated systems were running several times faster than what is required by the standard.

8. References

[1] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, D. Voggenauer, "The MIMOLA Language - Version 4.1. Technical Report." Computer Science Dpt., University of Dortmund, Sept. 1994.

[2] S. Weber and K. Keutzer, "Using Minimal Minterms to Represent Programmability", International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS), September 2005.

[3] M. Reshadi, D. Gajski, "A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths", CODES+ISSS, September 2005.

[4] A. Agrawala, T. Rauscher, Foundations of Microprogramming: Architecture, Software, and Applications, Academic Press, 1976.

[5] S. Habib, Microprogramming and Firmware Engineering Methods, John Wiley & Sons, Inc., 1988.

[6] B. Gorjiara, D. Gajski, "FPGA-friendly Code Compression Technique for Statically Scheduled Horizontal Microcoded Custom IPs", International Symposium on Field-Programmable Gate Arrays (FPGA), February 2007.

[7] NISC Technology website http://www.cecs.uci.edu/~nisc/.

[8] M. Reshadi, B. Gorjiara, D. Gajski, "Utilizing Horizontal and Vertical Parallelism Using a No-Instruction-Set Compiler and Custom Datapaths", International Conference on Computer Design (ICCD), pages 69-76, October 2005.

[9] B. Gorjiara, M. Reshadi, D. Gajski, "Generic Architecture Description for Retargetable Compilation and Synthesis of Application-Specific Pipelined IPs", International Conference on Computer Design (ICCD), October 2006.

[10] MiBench benchmark: http://www.eecs.umich.edu/mibench/

[11] MPEG Audio Decoder: http://www.underbit.com/products/mad/

[12] B. Gorjiara, M. Reshadi, D. Gajski, "NISC Communication Interface", Center for Embedded Computer Systems (CECS), TR 05-18, December 2005.

[13] Fraunhofer-Gesellschaft website: ftp://ftp.fhg.de/pub/layer3/mp3-bitstreams.tgz

[14] N. Clark, H. Zhong, K. Fan, S. Mahlke, K. Flautner, K. Van Nieuwenhove, "OptimoDE: Programmable Accelerator Engines Through Retargetable Customization", Hot Chips, 2004.

[15] M. Byatt, "Data plane processing with configurable architectures", ARM white paper, 2003.

[16] P. Mishra and N. Dutt, "Architecture Description Languages for Programmable Embedded Systems", IEE Proc. on Computers and Digital Techniques (CDT), Special issue on Embedded Microelectronic Systems: Status and Trends, vol. 152, no 3, 2005.

[17] W. Qin and S. Malik, "Architecture Description Languages for Retargetable Compilation", in The Compiler Design Handbook: Optimizations & Machine Code Generation. Y. N. Srikant and Priti Shankar, CRC Press, 2002.

[18] O. Schliebusch, A. Chattopadhyay, R. Leupers, G. Ascheid, H. Meyr, M. Steinert, G. Braun, A. Nohl, "RTL Processor Synthesis for Architecture Exploration and Implementation", Design, Automation and Test in Europe (DATE), 2004.

[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] R. Leupers and P. Marwedel, "Retargetable Generation of Code Selectors from HDL Processor Models", Design, Automation and Test in Europe (DATE), 1997.

[21] R. E. Bryant. "Graph-based Algorithms for Boolean Function Manipulation." IEEE Trans. on Computers 24.3 (1992): 293-318.