

Supporting FPGA Microprocessors through Retargetable Software Tools*

David A. Clark and Brad L. Hutchings
Department of Electrical and Computer Engineering
Brigham Young University
Provo, UT 84602

Abstract

FPGA systems outperform many ASIC and super computer systems through effective use of the reconfigurable resource. Reusing design effort across different applications requires a standard, flexible software environment. Driving FPGA systems from ANSI C is possible using `lcc` (an ANSI C compiler) targeted at an FPGA system and `dasm` (a retargetable, flexible assembler). The compiler supports custom hardware capabilities of FPGA systems, as well as all constructs of C. The assembler reads instruction definitions at assemble time, allowing the user to add new custom hardware functions which `dasm` can assemble correctly to an instruction stream the hardware executes. A source code debugger has been implemented for this system.

1 Introduction

FPGAs are capable of achieving high performance on many application-specific tasks. In many cases performance achievable with FPGAs on certain applications exceeds comparable ASIC designs or even super computers[2, 7].

One approach used in obtaining this high performance on application-specific designs is through a framework of application-specific modules and a programmable core like the DISC[12] processor. The programmable core is functionally capable of supporting high level languages like ANSI C. The application-specific modules are custom hardware modules which seamlessly interface with the programmable core and provide high levels of performance on specific application's tasks.

Using this approach, applications can be organized as 'C' code and a set of application-specific hardware modules. The 'C' code is compiled in conventional fashion with the application-specific modules accessed as subroutines from a run-time library. Unlike other approaches, the program itself is *not* compiled into hardware but rather is used to control the operation and sequence of the application-specific hardware modules. In addition, the 'C' code can be used to implement parts of the application when application-specific hardware modules are either not required nor available.

This paper will focus on the software tools for the DISC processor which enable users to combine ANSI C with application-specific hardware modules and obtain high levels of performance. The next section introduces the DISC architecture, and how applications are developed for it.

1.1 DISC Architecture

DISC is a partially reconfigurable FPGA-based processor consisting of a programmable core and application-specific hardware modules. It is capable of reconfiguring at run-time to replace idle hardware circuitry with active hardware modules. In this way DISC can be thought of as a processor with a potentially unlimited instruction set.

Each instruction is implemented with independent circuit modules which are paged into the hardware as dictated by the program execution. When the program needs to execute an instruction module not found in the FPGA, a new instruction module is placed in an unused area of the FPGA and execution is resumed.

The programmable core of this machine implements basic instructions used in all applications necessary to support C. It makes a broad range of applications possible, but has relatively low performance when compared to typical microprocessors.

*This work was supported by ARPA/CSTO under contract number DABT63-94-C-0085 under a subcontract to National Semiconductor.

An application-specific hardware module is designed to accomplish a specific task efficiently. It is treated as a hardware instruction and is generally applied to a small class of applications. They are generally more hardware intensive and accomplish a relatively large amount of work in a short time.

A program for this type of a processor consists of a sequence of instructions and a library of hardware circuitry. Utilizing the application-specific modules for a large portion of the algorithm provides high performance. In our experience this approach provides a working prototype with minimal design time.

1.2 Development Process

Applications for this hardware framework are written in ANSI C using procedure calls to call application-specific modules. The program is compiled with `lcc` which generates assembly code to run on DISC. This code is converted to machine readable form by `dasm`, a retargetable assembler.

Developing new applications consists of partitioning the algorithm into hardware and software, coding the application, designing the hardware modules and debugging the implementation. The algorithm is partitioned into two pieces, a central time critical piece to be designed in hardware, and the rest of the application. Typically, the whole application is first coded in C, each hardware function being implemented as a procedure. This provides a working model quickly and a basis from which to design the hardware modules.

Once the software is working, application-specific hardware modules are designed to replace the software procedures. The C code is then recompiled with a flag to call the hardware instead of software procedure. Additional hardware is designed and tested until the application's performance is acceptably high.

The remainder of this paper is outlined as follows. Section 2 describes the types of support required in order to support a high level language and debugging environment. Sections 3-5 describe the tool suite used to support DISC[12] namely, `lcc`, `dasm`, and `ddb`. The final two sections are devoted to future work and conclusions.

2 Hardware Support needed for High Level Languages and Debugging

Building a FPGA-based system capable of supporting high level languages and debugging, requires hardware support to operate efficiently.

This section discusses the hardware constructs DISC uses to support ANSI C. It will cover addressing modes, instruction set and debug support from a hardware perspective.

2.1 Addressing Modes

Addressing modes are important to generating efficient assembly code from ANSI C. The modes implemented in the DISC processor are shown in Table 1. These four modes are implemented in a central control block which is used by all instruction modules. This maximizes instruction module design reuse and provides a clean interface for each instruction to use.

Name	Meaning
immediate	Use given value in computation
direct	Fetch operand from given location
indirect	Fetch from address given at location specified
indirect offset	Add offset to given location and fetch from address found there

Table 1: C addressing modes.

It is possible to support C with a smaller set of addressing modes, but it would result in larger code size and lower performance. Immediate mode is useful for variable initialization. Direct addressing is commonly used in many instructions. Indirect addressing is vital in supporting pointers, arrays and structures. Indirect offset mode is not as necessary, but used often enough to justify its existence.

Adding these addressing modes to a system will greatly increase efficiency and system functionality in supporting high level languages.

2.2 Instruction set

The `lcc` code generator assumes the underlying hardware can support the following types of instructions: *load*, *store*, *negation*, *NOT*, *AND*, *OR*, *XOR*, *ADD*, *SUB*, *multiply*, *divide*, *modulus*, *shift right*, *shift left*, *compare instructions* (*eq*, *lt*, *gt*, *le*, *ge*, *ne*), *conditional jump*, *unconditional jump*, *function call*, and *return*. The designer has many options in implementing the functional equivalent of these basic instructions.

The DISC processor does not implement all of the above instructions as single executable entities. Some

of them are implemented using a sequence of one or more instructions. For example, the *multiply* was implemented using 65 lines of assembly code. This reduces the hardware design time and increases execution time. *Negation* and *NOT* were implemented with an *XOR* command, reducing the number of primitive hardware modules to design, but not affecting execution speed. Similarly, all instructions were examined and the design time, chip area, execution speed and frequency of use trade offs were made.

2.3 Support for Debugging

Debugging a process running on an FPGA system is a key part of a successful project. A debugger must be able to interact with the FPGA system, retrieve system state and assign values to specified locations. A condensed list of debugger actions and the required hardware support is listed in Table 2.

Debugger command	Hardware Support
Set/Clr BP	R/W instruction memory
Print Value	Read memory (reg, stack, mem)
Show state	Read registers and state

Table 2: Debugger support.

3 lcc - ANSI C Compiler

`lcc` is a retargetable compiler for ANSI C Standard X3.159-1989. It is in production use at Princeton University and AT&T Bell Laboratories, written by Fraser and Hanson[5, 6]. The distribution comes with source code for three target architectures, namely MIPS, SPARC and X86 machines.

The compiler consists of about 10,000 lines of front-end code and each target specific portion consists of 700-1200 lines of back-end code. The front end of the compiler parses the input file, does a few code optimizations and generates an intermediate language, to be read by the back end. Using a few front end data structures and procedures the back end takes the intermediate code and produces assembly language for the selected target. Those interested in obtaining a copy of `lcc` should refer to <http://www.cs.princeton.edu/software/lcc/>.

`lcc` supports the full ANSI C standard, but due to some hardware size and design constraints, the DISC

version does not support floating point or doubles. Addresses and data are represented with 16-bits, limiting the size of the programs to 64K. The hardware limits instructions to no more than one operand. Command line options supported by `lcc` are shown in Table 3.

Option	Description
-f=<file>	Specify follow-on file to include before assembling
-g	Emit debug symbols
-h=<file>	Specify header file to include before assembling
-p	Print intermediate code
-t=<arch>	Specify target architecture

Table 3: `lcc` command line options.

In order to generate code for the DISC processor, a set of base instructions, stack protocol, register convention, and instruction mnemonics must be defined. This section will describe the machine-dependent portion of code used to generate code for DISC (about 1500 lines of C code).

3.1 Generating DISC code

Before generating code for the DISC processor the assembly language instruction format and mnemonics must be defined. Table 4 shows a few examples of DISC instructions. This should give the reader a flavor of the addressing modes and basic syntax used for DISC instructions.

Assembly	Meaning
ld #0x25	Load accumulator with 0x25
sd -4+25(sp)	Store accumulator to offset 19 from the stack pointer
add R24	Add accumulator with contents of register 24
sub (R28)	Subtract value pointed to by register 28 from accumulator
jmp foo	Jump to label foo

Table 4: Sample DISC assembly instructions.

Once the mnemonics, addressing modes and operand formats have been decided, code for the machine can be generated by writing the back end code for `lcc`. This

is done by creating a `.md` file which specifies the rules for emitting code, defines target specific functions and defines interface definitions. This file is in the following format:

```
%{
  <interface prototypes>
  <DISC prototypes & data>
%}
  <terminal declarations>
%%
  <DISC code emitting rules>
%%
  <DISC functions>
  <DISC interface definition>
```

`lburg`[3, 4] takes the `.md` specification as input and emits a `.c` file which is compiled as part of `lcc`. Interface prototypes and data define back end procedures and variables available to the front end. Terminal declarations are the basis of the intermediate language emitted by the front end and consumed by the back end. The DISC code emitting rules and functions take the output from the front end and actually produce the assembly code.

3.2 Register Allocation

Thirty-two memory locations are used as pseudo-registers to compensate for DISC's accumulator based approach and make it look more like a register-rich target like the MIPS or SPARC processors. Starting with the MIPS processor back-end for `lcc`, a working back-end for DISC was successfully coded in about a week.

Through analysis of the number of references to each register within 300K lines of assembly code, a register allocation scheme was developed. This was done by assembling 300K lines of assembly and counting the number of references to each register within that code. After studying how `lcc`'s register allocation scheme behaved, the scheme shown in Table 5 was chosen.

Due to limited routing resources on the CLAYFUN board[11], DISC's implementation makes it impossible for programs to read the value of the hardware stack pointer into the accumulator. Block copies need to perform operations on the stack pointer so a copy of it is kept into one of the pseudo registers and updated the same time as the stack pointer. The stack pointer is only changed entering and exiting functions, so keeping two copies of the stack pointer is easily maintained.

Registers	Use
R0-R3	First procedure arguments
R4	Function return value
R5-R7	Temporaries used during mult and div
R8-R9	Scratch Registers used in block copies
R10-R25	Variables
R26-R29	Temporaries
R30	Stack pointer
R31	Return Address

Table 5: DISC Register Convention.

3.3 Stack Frame

In order to support procedure calls and recursion, a simple stack frame (see Figure 1) was implemented for the DISC architecture. The calling convention assumes callee-saved registers. Sizes of the frame are determined at compile time, depending on the number of locals, arguments and registers within the frame. As typical `lcc` target architectures have no stack manipulation instructions, the stack frame size is set at compile time by the largest number of items it uses at once. This `lcc` methodology speeds up processor execution, but consumes more stack memory.

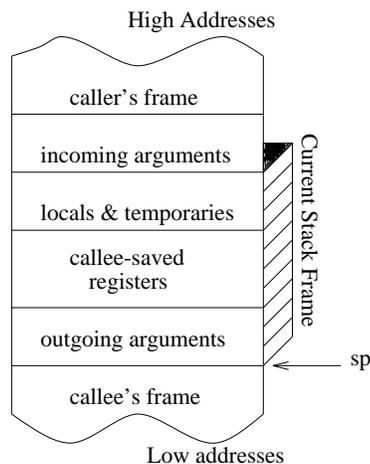


Figure 1: DISC stack frame.

To allow for a contiguous image processing space, the top of the stack was chosen to be at `0x8000`, instead of at the very top of memory. Typical images used in the DISC system are `256x256` bytes which consume `0x4000`

bytes of memory. This enables two images to fit above the stack space for easy manipulation.

3.4 Calling hardware routines

To support flexible calls to custom hardware modules from within C code, `lcc` was extended to treat hardware routines like procedure calls. This allowed zero or more parameters to be passed to the hardware routine and zero or one return values to be used in the program. This section discusses two ways to call DISC hardware routines from within C code.

The first hardware calling convention is to append the key word `native_` to the front of the custom instruction mnemonic. This tells `lcc` to load the given arguments into the correct locations (see Table 5), and use the string following `'_'` and before `'('` as the mnemonic to call the hardware instruction directly. A custom module called `mean`, would be called from within C code using `native_mean()`;

The second method calls a user-defined assembly language routine, passing arguments and using the return value in the same way as regular function calls. This is implemented by loading the return address into R31 and jumping to the function label (using the same string as defined by the procedure). The user then can write an arbitrary assembly language routine, using the arguments passed to it if desired. At the end of the routine, the user must insert a `jmp (R31)` to return to the calling routine.

The user's assembly routine is included in the `.asm` file through the command line option `-f <filename>`. This produces a `#include "<filename>"` statement which includes the user defined assembly program during the assemble stage.

With these two methods, the user has full flexibility in using custom hardware instructions with C code. A simple C program that incorporates both of these methods is seen below. It assumes custom instruction `'bar'` and user defined assembly subroutine `'foo'` exist.

```
main () {
    int i=0, j=0, z;
    j = native_bar(i);
    z = foo(j);
    return z;
}
```

After compilation with `lcc`, the following assembly program is produced. This output is simplified to hide unnecessary detail from the reader.

```
#include "definition of instruction set"
<set up stack frame>
<save registers>
<initialize variables>
sd R0      ; Put i's value in parameter 1
bar R0     ; Call custom module.
ld R4      ; Load return value (j)
sd R0      ; Store j in parameter 1 register
ld #G_4    ; Load return addr
sd R31     ; Store to return register
jmp foo    ; Jump to user defined subroutine.
G_4:       ; Return address
<restore registers>
#include "user_defined_foo"
stop      ; Processor halt instruction.
```

The user defined assembly routine may contain any amount of instructions, procedure calls, branches etc. as long as stack protocol is not violated. Return values for these functions must be placed in register R4. An example of such a routine is shown below:

```
bar:      ; Function label
    ld R0
    eq #1
    jnc second
        mean #0, #1
second:
    threshold #1, #0
    jmp (R31)
```

In this example, the `mean` instruction is called only if the parameter passed in is equal to one, otherwise only the `threshold` operation will be performed. `Threshold` and `mean` are both custom instructions with two immediate operands. Since `lcc` can not call functions with immediate operands, (a limitation in `lcc`'s code generation), this user defined assembly code must be written. If these hardware modules accepted arguments from memory, this would not be necessary.

3.5 Debugging software

Having the compiler treat special purpose hardware as a function call gives greater flexibility in testing the software portion of the application. This allows algorithms for custom hardware instructions to be tested as a software algorithm on the processor before implementing it in hardware. This has proved useful in debugging the software algorithm and later in testing the hardware modules. An example of this method is shown here.

```

main () {
    int i=0, j=0, z;
    j = native_bar(i);
    z = foo(j);
    return z;
}

#ifdef SOFTWARE_DEBUG
int native_bar(int i) {
    /* C Code necessary to implement function */
}
int foo(int i) {
    /* C Code necessary to implement function */
}
#endif

```

One advantage of this approach is the freedom to use hardware or the software version of the program to get the highest performance. Some application-specific hardware modules take more time to load into the FPGA than they take executing. If the module is only going to be executed once, the designer may choose to use the software version of the module.

Treating hardware routines like procedure calls in C enables the programmer to easily use either the hardware routine or a software version of the routine without changing the source code.

3.6 Debug Symbols

Invoking `lcc` with the `-g` option emits debugging symbols. To successfully debug a program, the name, type, location and size of each variable in the source code must be communicated to the debugger. `lcc` accomplishes this through assembler directives which are read by the assembler and a pre-assembler script.

Variable types are defined with the `.stabdef` directive. Below are the definitions for `char` and `int` type declarations. Structures, arrays and pointers are defined through combining previous definitions.

```

.stabdef <int> 1=INT
.stabdef <char> 2=char

```

Before the variable is used in the assembly code, `lcc` emits a `.stabsym_` directive showing where it is stored (local, global, parameter), its definition number and symbolic name. The final few characters give the major storage class as shown in Table 6.

```

.stabsym_reg j=1, R25
.stabsym_loc i=1, -1
.stabsym_loc z=1, -2

```

Storage Class	Definition
reg	Register
loc	Local variable on stack
global	Global variable
param	Function call parameter
function	Function name, return type

Table 6: Storage locations.

The current scope of the program variables is exported by `.stabblock` directives. Variables declared inside each block have visibility within the block. `lcc` emits one directive for each C code brackets '{' and '}' in the source code.

```

; .stabblock Left 0 ; (l|r level)
; Code in the block...
; .stabblock Right 0 ; (l|r level)

```

This allows variables of the same name to exist within the same function, each with a specific scope. The debugger determines the correct scope of each variable and is able to use the correct variable when it is needed.

4 `dasm` - Dynamic Assembler

`dasm` is a dynamic assembler, one that does not need to know the target instruction set until assemble time. Each target machine is fully specified in a header file, including word length, instruction mnemonics and bit placement for each instruction.

The goal of `dasm` was to be able to assemble programs for DISC though it has been successfully demonstrated on DLX, MIPS and other architectures. `dasm`'s flexibility is necessary when developing in an FPGA environment where the hardware is extremely flexible. This gives the system designer freedom to change the design without being hindered by assembler constraints.

`dasm` accepts an assembly file (traditionally `.asm`) consisting of directives, instructions and machine definitions and produces three files, `.obj`, `.lst` and `.dbg`. The `.obj` file is an ASCII output of the fully assembled program. ASCII was chosen for the ease of converting to other forms of output. The `.lst` file is a listing of the assembly output, including locations of errors and warnings. Symbols are emitted to the `.dbg` file, when the `-g` option is invoked. `dasm` supports command line options outlined in Table 7.

Option	Meaning
-g	Emits symbols and debug information to .dbg file
-h	Prints header file information to .lst file to aid in header file debugging
-o <file>	Put output files in <file>

Table 7: `dasm` Command line options.

This section will discuss assembler directives, templates, addressing modes and debug hooks.

4.1 Defining the Target Machine

This section describes the method of representing a new target architecture to `dasm`. This target specific information is placed in a header file to be included in the assembly file before assembling.

Within the header file, the architectural features that must be specified for each machine include: minimum number of addressable bits, length of data and instructions, number and mnemonics of registers and byte order. The specific mnemonics of the architecture defining directives are given in Table 8. Limitations of `dasm` in defining architectures are; only fixed length instructions are possible and multi-file compilation is not supported.

Mnemonic	Definition
<code>.Align_Boundary</code>	Specifies minimum number of addressable bits
<code>.AlignPerInst</code>	Number of ABs per instruction
<code>.AlignPerDb</code>	Number of ABs per piece of data
<code>.BigEndian</code>	Emit Big Endian output
<code>.LittleEndian</code>	Emit Little Endian output
<code>.Registers</code>	Specify special character used to denote registers
<code>.SpecRegister</code>	Specify additional reg by name

Table 8: `dasm` Directives.

The DISC architecture is defined for `dasm` by using the following directives:

```
.Align_Boundary 16
.AlignPerDb 1
.AlignPerInst 2
.LittleEndian
.SpecRegister "sp" 1
```

These tell `dasm` that the smallest addressable word on DISC is 16 bits, instructions are 32-bits long (two addressable words), data is defined as 16-bit quantities, byte order is little endian and “sp” is a register. Once this information is defined, it is static for the entire compilation process.

4.2 Templates

Instruction templates give the user an easy mechanism to define new instruction sets similar to Ramsey’s[9] method of templates. A template defines a collection of one or more sub-templates. Each sub-template defines which bits it maps to, with the freedom to choose non-contiguous patterns of bits. A checking mechanism ensures multiple sub-templates within a template don’t write to the same bit location.

A simple example of how templates work is seen using the DLX instruction set. For this architecture, instructions are defined using R, I and J templates consisting of Op, Rs1, Rs2, Rd, Const16, Const26, and Opx sub-templates. These three templates and sub-templates are shown in Figure 2. The `dasm` directives which describe this information are given by:

```
.TemplateType R (Op, Rs1, Rs2, Rd, Opx)
.TemplateType I (Op, Rs1, Rd2, Const16)
.TemplateType J (Op, Const26)
```

```
.SubType Op <31:26>
.SubType Rs1 <25:21>
.SubType Rs2 <20:16>
.SubType Rd <15:11>
.SubType Rd2 <20:16>
.SubType Opx <10:0>
.SubType Const16 <15:0>
.SubType Const26 <25:0>
```

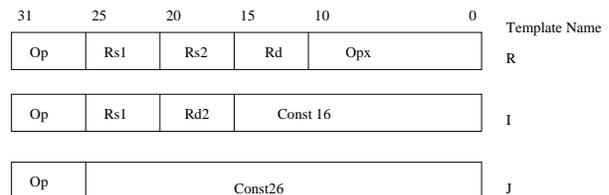


Figure 2: DLX Template Example.

Notice that the sub-template Op and Rs1 are instantiated in more than one template, while two separate

sub-template definitions are required for Rd and Rd2 because they use two different bit positions within the template.

The DISC architecture requires a single DISC template but uses non-contiguous bits within the sub-templates as seen in Figure 3 and the `dasm` directives below. This feature gives maximum flexibility when selecting pins on the FPGA as breaking busses apart or interleaving bits could provide greater design density.

```
.TemplateType D (Id, Ucode, Operand, Sp)

.SubType Id <30:28,23:20>
.SubType Ucode <27:24,19:16>
.SubType Operand <15:0>
.SubType Sp <31>
```

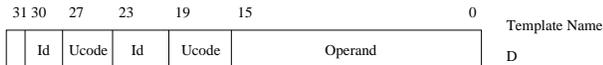


Figure 3: DISC Template Example.

4.3 Defining instructions

Once templates and sub-templates are defined, they are used to specify the machine’s instruction set. Instructions are defined by specifying the mnemonic, number of operands, template name, a list of the non-zero sub-templates and their values or the operand number and type. Operands can have types as defined in Table 9. All of this information is used to check the input assembly program for syntax errors.

Type	Meaning
immediate	Operands preceded with #
direct	No special modes specified
indirect	Denoted by parenthesis ()
address	Address of operand &
register	As specified by the .Registers directive

Table 9: `dasm` operand types.

Each instruction must have a unique combination of mnemonic string, types and number of operands. An example of two assembly instructions are given for the DISC processor.

```
ld -3+6(sp)
ld #5
```

The instruction directives that match these definitions are defined as:

```
.inst ld 2 D (Id, 0b001010), \
(Ucode, 0b10100000), \
(Operand, direct, 1), \
(Sp, register, 2)
.inst ld 1 D (Id, 0b001010), \
(Ucode, 0b01100000), \
(Operand, immediate, 1)
```

These definitions tell `dasm` to look for instructions named “ld” with one or two operands. The operand types must either be immediate or direct and a register (“sp” was defined as a register using `.SpecRegister` directive). `dasm` will match the first instruction “ld -3+4(sp)” to the second instruction definition.

Once a match has been made, it places the value of the operands in appropriate fields as defined in the templates and sub-templates. The Id and Ucode fields are specified in the definition, “-3+4” is placed in the Operand field and the register “sp” is placed in the Sp sub-template field. `dasm` checks consistency with the previously defined templates and stores each instruction in its database.

5 DDB - DISC Debugger

`ddb` is a debugger used in debugging applications on the DISC processor, modeled after Ramsey’s[8, 10] work with retargetable debuggers. It is a source level debugger, and can debug assembly files or C files. It runs on the host processor and debugs processes that are running on the DISC processor remotely. Cooperation from the DISC processor is assumed and basic debugging commands are shown in Table 10.

`ddb` was successfully implemented for the DISC architectures. Details are available in Clark[1].

6 Future Work

Several extensions could be made to each of the tools discussed in this paper to further the usability and functionality each interacts with the system.

`lcc` could be extended to support doubles and even floating point. This extension could help study a larger class of scientific code and algorithms.

Command	Definition
br line#	Break at C line number
c [#]	Continue 1+ times from current bp
d bp#	Delete specified breakpoint number
f	Print Current focus
h	Print Help
list	List breakpoints
l <file>	Loads <file> to debug
n [#]	Continue one [n] C code line #s
p <name>	Print symbol name
r bp#	Redo breakpoint number
run	Begin execution of program
s [#]	Single step machine [n] times
u bp#	Undo breakpoint number

Table 10: `ddb` commands.

`dasm` could be extended to support a standard object format, and a retargetable linker could be written to allow multiple input files. This would allow larger, more complex programs to be examined on the FPGA systems. With this capability, the compiler itself could be ported to the system, enabling development of operating systems, applications and developmental tools.

These tools could also be ported to other types of systems, such as DSPs to utilize advantages in that system. A general purpose simulation environment could aid in developing the FPGA system by allowing fast tradeoffs between system configurations as the system is developed.

7 Conclusion

It has been shown that applications may be developed for FPGA-based systems using conventional C programs. This also allows application-specific hardware modules to be integrated into the code with almost no effort. This method provides a programming interface to the FPGA-based system that does not require special extensions, is well understood and a standard that many programmers use.

References

- [1] David A. Clark. Supporting FPGA Microprocessors through Retargetable Software Tools. Thesis, Brigham Young University, April 1996.
- [2] C. P. Cowen and S. Monaghan. A reconfigurable Monte-Carlo clustering processor (MCCP). In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 59–65, Napa, CA, April 1994.
- [3] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a Simple, Efficient Code-generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.
- [4] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG - A Fast Optimal Instruction Selection and Tree Parsing. *ACM SIGPLAN Notices*, 27(4):68–76, April 1992.
- [5] Christopher W. Fraser and David Hanson. A Retargetable Compiler for ANSI C. *ACM SIGPLAN Notices*, 26(10):29–43, 1991.
- [6] Christopher W. Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Publishing Company, Inc., 1995. ISBN 0-8053-1670-1.
- [7] D. P. Lopresti. Rapid implementation of a genetic sequence comparator using field-programmable gate arrays. In C. Sequin, editor, *Advanced Research in VLSI: Proceedings of the 1991 University of California/Santa Cruz Conference*, pages 138–152, Santa Cruz, CA, March 1991.
- [8] Norman Ramsey. *A Retargetable Debugger*. PhD thesis, Princeton University, 1993.
- [9] Norman Ramsey and Mary F. Fernandez. The New Jersey Machine-Code Toolkit. In *USENIX Technical Conference*, pages 289–302, New Orleans, LA, January 16-20 1995.
- [10] Norman Ramsey and David R. Hanson. A Retargetable Debugger. *ACM SIGPLAN Notices*, 27(7):22–31, 1992.
- [11] C. R. Rupp. *CLAYFUN Reference Manual*. National Semiconductor, version 2.00 edition, September 1994.
- [12] M. J. Wirthlin and B. L. Hutchings. A Dynamic Instruction Set Computer. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 99–107, Napa, CA, April 1995.