Syntax-Driven Implementation of Software Programming Language Control Constructs and Expressions on FPGAs

Neil C. Audsley and Michael Ward
Dept. of Computer Science, University of York
York, UK
neil.audsley@cs.york.ac.uk, michael.ward@cs.york.ac.uk

ABSTRACT

This paper considers the efficient parallel implementation of control constructs and expressions written in a common software programming language and synthesised to FPGA platforms. The context of this work are *Syntax-Driven Language Specific Processors* (SDLSP). An SDLSP for a given software programming language has its architecture defined by the grammar rules of the language itself. Each statement and expression rule in the grammar is implemented on the FPGA, together with sufficient control logic to load program statements sequentially onto the processor, and interface with program store. The instructions executed are a high-level (effectively one-to-one) encoding of the application software program. The advantages of this approach lie in its parallelism and space-efficiency. Syntax-driven language processors take less space than a full CPU on FPGA, and execute statements with a comparable speed; take significantly less space in general than directly compiled approaches (such as Handel-C), although have longer execution times for the same code.

Categories and Subject Descriptors: D.3 [Programming Languages]: Processors *Compilers* C.1 [Processor Architectures]: Other Architecture Styles *High-level language architectures*

General Terms: languages

Keywords: compilation, fpga, language

1. INTRODUCTION

Language Specific Processors (LSP) execute programs written in a single language, unlike a CPU which can execute any suitably compiled program. Common LSPs are interpreters or virtual machines [1]. The executable instructions of LSPs are usually at a higher level of abstraction than for CPUs (eg. compare Java bytecodes [2] to CPU machine instructions). LSPs can be implemented on top of CPUs, thus performing a run-time translation to CPU executable code; or directly on FPGA (eg. [3]). LSPs are of fixed size for a given programming language, the size of the code to be interpreted being proportional to the size of the source program.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'06, October 23–25, 2006, Seoul, Korea.
Copyright 2006 ACM 1-59593-543-6/06/0010 ...$5.00.

In this paper, we discuss an LSP whose architecture resembles the structure of the grammar of the language it is intended to process, termed the *Syntax-Driven LSP (SDLSP)*. The SDLSP executes instructions that are at the level of source language statements, ie. at a higher-level of abstraction than conventional LSP approaches. In general, instructions are a one-to-one mapping with language constructs and statements, with expressions taking multiple instructions depending upon their size. The main advantages of the SDLSP approach are its space-efficiency and inherent parallelism, due to the high level (ie. language grammar) of SDLSP instructions. For example, parallelism is achieved by allowing independent parts of a single language statement, construct or expression to execute in parallel. An important aspect of the SDLSP approach is the parallel evaluation of expressions. In this paper the architecture and operation of an SDLSP is detailed, focussing upon control constructs and expressions. An example is provided, in terms of an implementation (on FPGA) of an SDLSP for the TINY programming language [4].

It is noted that although SDLSPs are specific to the programming language that are intended to process, many aspects of common (imperative) programming languages are identical (eg. control-flow constructs, expressions).

The remainder of the paper is structured as follows. Section 2 presents background and previous work. Section 3 describes the SDLSP architecture and operation, with section 4 outlining an SDLSP for TINY. Section 5 extends the general SDLSP approach for arithmetic expressions, with section 6 extending the TINY SDLSP for expression evaluation. Implementation of the TINY SDLSP on an FPGA is detailed in section 7, together with evaluation results. Finally, conclusions are presented in section 8.

2. BACKGROUND AND PREVIOUS WORK

Programs written in a software programming language can be compiled to execute on FPGAs in a number of ways:

1. *Application-specific Circuit (ASC)*: Both subsets of standard languages (eg. C [5] and Ada [6]) and language variants (eg. Impulse-C [7], Handel-C [8]) have been successfully compiled to FPGA, forming a program-specific (ie. application specific) implementation. Such approaches can include library modules from other languages (eg. VHDL).

2. *CPU-Specific (CS)*: This represents a conventional compilation of a program to CPU-specific executable form. Clearly, many languages are compiled in this manner. In general, the compiled executable can only be executed on the target CPU (or CPU family). We note that the CPU can be a softcore, targeted at an FPGA; or a hardcore CPU embedded within an FPGA (eg. PowerPC with Xilinx Virtex Pro).

3. *Language Specific Processor (LSP)*: An interpreter or virtual machine for a specific programming language. Programs in the language are compiled to a high-level language specific executable form, which is then interpreted by the LSP. One key example of an LSP is the Java Virtual Machine, where the executable form is Java byte code [2]. LSPs can be implemented as a translation layer on top of a CPU, ie. performing runtime translation to CPU instructions; or directly on the FPGA. In the latter case, the architecture of the LSP is similar to that of a CPU (eg. consider hardware implementations of the Java Virtual Machine [3, 9, 10].

4. *Hybrid Approach*: Two or more of the above can be combined. For example, where a CPU is combined with an application-specific accelerator or co-processor to speed up a particular application function.

The CS and LSP approaches are similar, in that both fetch, decode and execute a sequence of instructions. However, a key difference is that LSPs often contain language specific features within the architecture, eg. a Java LSP (ie. a Java VM) contains security, dynamic memory management and other language specific facilities [2].

The four approaches highlight different trade-offs that can be exploited for an efficient implementation. When the target is a single FPGA, the finite available resources (logic cells, routing and memory) suggest that in general, as source programs get bigger (eg. in lines of code), more resources are required for implementation. When the resources available on a single FPGA are not sufficient for the implementation of an application program, a number of strategies can be adopted, from changing the platform (eg. larger FPGA, different device family, multiple FPGAs, additional RAM for program store); changing the source (eg. redesign of the application). When neither of these strategies are appropriate, implementations tend towards hybrids, utilising both CS or LSP together with some ASC for speed. However, moving from a pure ASC to an implementation including a LSP or CS is significant, due to the space overheads of the CPU or LSP (ie. they require accompanying buses, memory and other devices). Hence, it necessarily efficient to include a CPU or LSP unless significant parts of the implementation are executed on these devices.

The compilation process for the ASC, CS and LSP approaches are similar (ie. involving lexical, syntactic and semantic analysis, together with code generation and optimisation). For circuit generation (for ASC) or code-generation (for CS and LSP) the compilers instantiate mappings for each source language statement/construct/expression to target circuit / executable. Essentially, there is a mapping from statement/construct in the language to a separate subcircuit (ASC), set of CPU instructions (CS) or set of language specific high level instructions (LSP). In ASC, some degree of sharing of resources can be achieved (thus reducing FPGA space requirements), particularly with expensive mathematical operations (eg. divide); although this is more complex for software languages with concurrency (eg. Ada) where contention is harder to prevent.

As stated above, instructions interpreted by the LSP are of a higher abstraction level compared to machine instructions – eg. object create Java bytecode. Usually, one source statement / construct / expression is compiled to many LSP instructions. Each LSP instruction can take many cycles to execute, depending upon its complexity.

In CS, optimisation techniques can reduce the amount of space required to store the program [11], so reducing overall resources required. It is noted that the CPU is not affected (nor buses linking CPU and memory), with the resources taken by the CPU constant.

3. THE SDLSP APPROACH

An SDLSP is distinct from a LSP in the following ways:

1. the architecture of an SDLSP follows the grammar rules of the language;

2. SDLSP instructions are simple encodings of the source language;

3. an SDLSP executes instructions in a non-atomic nested manner;

4. an SDLSP allows independent parts of statements / constructs / expressions to be executed in parallel.

The following sections detail the SDLSP approach.

3.1 Intuition

An SDLSP executes each statement, construct or expression in the source program sequentially. Thus, the high-level instructions interpreted by the SDLSP correspond to syntactic elements of the language. For example, one instruction may be an "if" statement. The expression representing the condition expression would form another instruction.

To interpret syntactic-level instructions, the SDLSP contains a separate component for each syntactic element in the language – ie. separate components for both the "if" and for expression evaluation. The "if" instruction is presented to both the "if" and expression components in parallel. When all components involved in the execution of a high-level instruction have completed, the instruction completes.

Instruction execution is nested, in that one instruction A may commence execution before B, but B may complete before A. Such nesting follows the permissible nesting of constructs within the grammar of the language. For example, an "if" instruction will start prior to those in the condition expression and the body of the "then" or "else" blocks; however, the condition expression and "then" body or "else" body will all complete prior to the "if" instruction completing. Hence, the operation is faithful to the grammar of the language.

3.2 Architecture

The SDLSP architecture is based upon the structure of a formal BNF (Backus-Naur Form) [12] representation of the grammar of the programming language. This is different than conventional LSPs, which are based on the architecture

of CPUs and operate on low-level bytecodes (ie. not at the syntactic level of the language) [1].

The remainder of this section considers BNF, together with the derivation of the structural architecture of an SDLSP from a given BNF grammar.

3.2.1 BNF (Backus-Naur Form)

BNF (Backus-Naur Form) [12] is an unambiguous mathematical (ie. formal) description of a language, having been used extensively for over 40 years for software programming languages (and many others). BNF represents a language as a set of terminals (ie. keywords) and non-terminal symbols (ie. syntactic structures), with production rules mapping non-terminal symbols onto non-terminals and/or terminal symbols. BNF is limited to describing context-free grammars [12], sufficient for common software programming languages (eg. Ada, C, Pascal, Java), and others (eg. VHDL, Verilog).

A simple example of BNF is given in Figure 1 where the start symbol for the grammar is “A”. The grammar describes a language where programs begin with symbol “start” (ie. rule “A”), have one or more statements (rules “NT1” and “NT2”)), and finish with symbol “end”. Each statement can be a conditional (rule “NT3”), or assignment of expression to an identifier (rule “NT4”). Expressions for the conditional and right-hand side of the assignment are simple, being constant integers (rule “NT5”). An example program conformant to the grammar is given in Figure 2.

3.2.2 Structural Architecture

An architecture based upon the BNF (ie. syntax) of a language represents all run-time semantically significant elements of the language (eg. a control statement) as distinct components in the architecture. These components are linked according to the syntax rules (ie. BNF production rules) of the language. For example, the “if” statement in Figure 1 has an expression to determine whether to execute a set of statements; this requires the corresponding “if” component within the architecture to be connected to the component responsible for expression evaluation (to obtain the result).

Each component implements the semantics of the statement/construct that it represents. For example, the “if” component waits for the condition expression to complete, and initiates execution of either the “then” or “else” set of statements as appropriate.

The architecture only contains components that are significant at run-time. For example, most programming languages contain syntactic elements that are present for scoping or to permit unambiguous parsing at compile-time. These are not required at run-time, after code generation, and so are not needed in the architecture.

As an example, Figure 3 shows the components and connections required for the BNF of Figure 1. Note that symbols “start” and “end” do not appear in the architecture as they are non-semantically contributing at run-time. Component “NT3” implements “if”; “NT4” implements assign. The grammar start symbol “A” is shown as the root for the component architecture. “NT1” and “NT2” are not shown, effectively as they are only in the grammar to enable programs to contain one or more statements in any order.

The symbol “NT5” is included as it represents simple expressions within this grammar. It passes the result of evaluating an integer (“0” or “1”) as a condition to the “if” node (ie. to symbol “NT3”); or to the right-hand-side of the “assign” node (ie. to symbol “NT4”).

```
A    ::=  ``start'' NT1 ``end''
NT1  ::=  NT2 | NT2 NT1
NT2  ::=  NT3 | NT4
NT3  ::=  ``if'' NT5 ``then'' NT4
                ``else'' NT4
NT4  ::=  string ``:='' NT5
NT5  ::=  0 | 1
```

Figure 1: Example: BNF Grammar

```
start
  tmp := 0
  if (1) then tmp1 := 1
  tmp2 := 1
end
```

Figure 2: Example: Source Program

Identifier and constant stores are added to hold (in distinct locations) identifiers declared within the grammar and constants respectively. Noting that some constants maybe including implicitly within an instruction (see section 3.3).

By adding store and instruction decode to the component architecture, a full SDLSP structural architecture is realised. This is illustrated in Figure 4 for the component architecture of Figure 3. The grammar start symbol is replaced by a fetch/decode component, responsible for loading the next instruction from program store; decoding that instruction; initiating execution of that instruction.

Further consideration of the architecture is given later in the paper, particularly in terms of implementation.

3.3 Instruction Selection

Instruction selection for a SDLSP is significantly less complex than during conventional code generation during compilation for a CPU target. Essentially, there is an exact one-to-one mapping of language construct to SDLSP instruction. For a CPU there it is usually a one-to-many mapping from language construct to machine instructions, with many different combinations of instructions to choose from, including different addressing modes, with optimal instruction selection difficult [12].

The instructions to be interpreted by the SDLSP are formed by a conventional parse and code generation phase. We as-

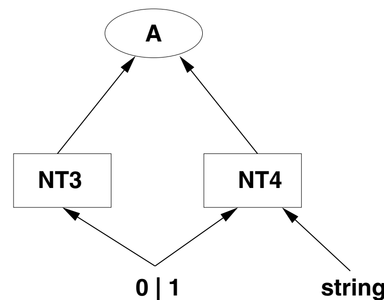

Figure 3: Example: Components

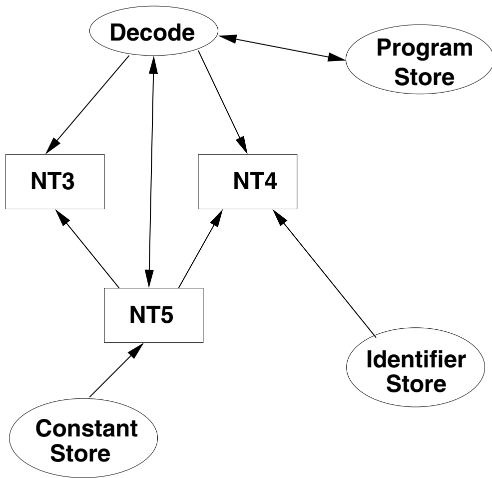

Figure 4: Example: Structural Architecture

	Opcode	Stop	Field	Instruction Width
expr	00	-	constant	$2+d$
assign	01	0/1	ident_addr : expr_addr	$3+2a$
if	10	0/1	expr_addr : then_addr : else_addr : after_addr	$3+4a$

Table 1: Example SDLSP Instructions (Simplistic Encoding).
Key: a = address width (bits); d = data width (bits)

sume that an appropriate parser is used for the language grammar – often an LR bottom-up parser is used for BNF grammars [12], eg. those produced by the YACC parser-generator [13]. We also assume that the parser produces an Abstract Syntax Tree (AST) and symbol table that can be used during code generation.

Instructions are generated by walking the AST, with 1 instruction generated for each run-time semantically significant language level statement / construct; 1 or more for each expression. The tree walk does not differ from that used in conventional compilers, indeed any approach can be used that is suitable for context-free grammars [12].

Instructions are binary, in the basic form:

OPCODE : FIELD

The opcode is fixed width (for a particular SDLSP), with each statement / construct mapped to a distinct opcode. Clearly, the number of bits required by the opcode is dependent upon the number of distinct statements / constructs in the language.

The field varies according to the opcode, being an encoding of all other pertinent information required by that opcode to execute. For example, an "if" construct typically has a condition expression, with the field of the "if" instruction containing the address of the first instruction representing the expression. The field is a fixed width, dependent upon the language.

A simplistic instruction set is given in Table 1 for an SDLSP for the language given in Figure 1. An opcode width of 2 bit is used to differentiate between the two statement forms (ie. "NT3" and "NT4") and the condition expression ("NT5").

	Opcode	Stop	Field	Instruction Width
assign	0	0/1	ident_addr : constant	$2+a+d$
if	1	0/1	else_addr : after_addr	$2+2a$

Table 2: Example SDLSP Instructions (Efficient Encoding).
Key: a = address width (bits); d = data width (bits)

	Source	Opcode	Stop	Field	Field Comment
0	tmp :=	01	0	0:0	tmp in location 0 : expression at line 1
1	expr:0	00	-	0	constant 0
2	if<x	10	0	3:4:-:6	expression at line 3 : then block at line 4 else block at line - : after starts at line 6
3	expr:1	00	-	1	constant 1
4	tmp1 :=	01	1	1:5	tmp1 in location 1 : expression at line 5
5	expr:1	00	-	1	constant 1
6	tmp2 :=	01	1	2:6	tmp2 in location 2 : expression at line 7
7	expr:1	00	-	1	constant 1

Table 3: Example SDLSP Using *Simplistic* Instructions for Figure 2 Program.

An instruction "stop" bit indicates whether the instruction is the last of a block. For example, an "if" needs to know when the block of statements for the "then" or "else" has completed, so the "if" itself can complete execution.

For the three instructions in Table 1, the field contains relevant information. The constant value of the expression is contained in the field of "expr"; the identifier location and expression location are in the field of "assign"; the addresses of the condition expression, "then" and "else" blocks, together with first instruction after the complete "if" block are contained in the field of the "if" instruction.

Table 3 shows shows the instructions required for the program of Figure 2. For an address width of 3 and data width of 1 (since only binary values are permitted by the grammar), the number of bits required is 54.

Note that for the grammar being considered, a number of efficiency savings in the instruction set can be achieved:

- remove "expr" instruction and encoding constants as immediate within "if" and "assign" – this saves one bit of opcode, and removes need for storage of "expr_addr" within "if" instruction field;
- ensure that the "then" block of statements occurs immediately after the "if" – – this removes need for storage of "then_addr" within "if" instruction field.

The net effect is that the number of instructions falls from 3 to 2; with fewer overall bits required to store the program. Table 4 shows the instructions required for the program of Figure 2. For an address width of 3 and data width of 1 (since only binary values are permitted by the grammar), the number of bits required by the program falls from 54 to 27.

3.4 Storage Allocation

Conventional code generation during compilation for a CPU target must consider low-level architectural storage is-

	Source	Opcode	Stop	Field	Field Comment
0	tmp := 0	0	0	0	tmp in location 0
1	if<x	1	0	-:3	else block at line - : after starts at line 3
2	tmp1 := 1	0	1	1	tmp1 in location 1
3	tmp2 := 1	0	1	2	tmp2 in location 2

Table 4: Example SDLSP Using *Efficient* Instructions for Figure 2 Program.

sues, including register allocation and temporary variable storage. For compilers targeting an SDLSP, storage allocation issues are constrained to:

1. the mapping of variables declared in the source program to locations in the identifier store;
2. the mapping of constants declared in the source program to locations in the constant store and/or to an immediate field in the instruction;
3. the use of temporary variables.

All variables declared within the source program are mapped to distinct locations in the identifier store. Similarly, constants are mapped to distinct locations in the constant store, if they are not stored in the field of the instruction (see above and section 6). Temporary variables are only required during expression evaluation, and are considered further in section 5).

We note that for compilers targeting CPUs, the use of procedure/function frames on the stack [12] allows (dependent on the scoping rules of the programming language) the effective mapping of several variables to the same location of physical storage, so reducing the overall storage requirement. This approach could be utilised for an SDLSP, although could require the addition of a stack and frame-pointer, particularly for multiple copies of the same variable (ie. if the language allows recursive procedure/function calls). However, this is outside the scope of the paper, as procedure/function calls are not considered herein.

3.5 Operation

The basic operation of an SDLSP is that instructions are fetched and decoded in order, with the actual execution of the instructions is non-atomic and nested (as described in section 3.1). The remainder of this section describes the operation of an SDLSP in more detail.

3.5.1 Initialisation

The decode node of the architecture (see Figure 3(b)) contains a Program Counter (PC). The PC is initialised to 0, or the location in program store of the first instruction to execute.

3.5.2 Fetch and Decode

The SDLSP initiates a fetch of the instruction in program store at the address contained in the PC. The decode node then passes the field of the instruction to the node responsible for processing the corresponding opcode – termed the *target node*. The decode node now waits for a node to signal that it can fetch the next instruction – termed the *fetch signal*.

Note, the fetch signal does not necessarily come from the target node; nor does the fetch signal imply that the instruction has completed execution. To illustrate, we consider an "if" control statement. This will send fetch signals to initiate fetching of the instruction for the condition, to fetch statements for either the "then" or 'else" bodies, and at completion of the "if" block.

Some instruction executions (eg. "if") may need to change the PC. Hence, the fetch signal can be accompanied by an address. If the decode unit receives an address with the fetch signal, the PC is updated to this new value prior to the next fetch.

3.5.3 Execute

When a node receives an instruction from the decode node (ie. becomes the target node) it executes the logic required to fulfill the semantics of the instruction. This may require waiting for completion of execution of other nodes. We now consider the execution of the three instructions of Table 1:

- "expr" – this instruction evaluates a constant (in the instruction field) passing the value of the constant to the "assign" and "if" nodes (noting exactly one of these will be active, waiting for the value).
- "assign" – this instruction stores the address of the target identifier, then issues a fetch signal to decode, as it needs to wait for the expression to evaluate. The "assign" node subsequently receives the value, stores it in the appropriate identifier location, and issues a fetch signal.
- "if" – this instruction stores the field addresses, then issues a fetch signal to decode, as it needs to wait for the condition expression to evaluate. The "if" node subsequently receives the value to determine whether the "then" or "else" block of statements is to be executed. If the "then" is to be executed, the "if" node issues a fetch signal and waits for the "then" block to complete. If the "else" block is to be executed, the "if" node issues a fetch signal together with the address of the "else" block (from the instruction field). When the appropriate "then" or "else" block has completed, the "if" node issues a fetch signal together with the address of the first instruction after the "if" node (from the instruction field).

The last instruction in each grammatical block of statements has the stop bit set. This enables the node representing the surrounding construct to recognise the block completion. For example, during the execution of an "if" statement, the "if" node needs to know when the "then" or "else" block has completed execution.

4. CASE STUDY: CONTROL STATEMENTS IN TINY

This section provides further insight into the SDLSP approach by describing an SDLSP for the TINY software programming language [4]. TINY contains conventional imperative language control structures and expression arithmetic; it does not contain functions / procedures, or data types other than integers. TINY contains sufficient complexity to illustrate the SDLSP approach. The remainder of this section discusses the overall architecture of the TINY SDLSP,

```
program ::= stmt-sequence
stmt-sequence ::= statement (; statement)
statement ::= if-stmt | repeat-stmt |
          assign-stmt | read-stmt |
          write-stmt
if-stmt ::= if exp then stmt-sequence
            [else stmt-sequence] end
repeat-stmt ::= repeat stmt-sequence
                until exp
assign-stmt ::= identifier := exp
read-stmt ::= read identifier
write-stmt ::= write exp
exp ::= simple-exp [comparison-op simple-exp]
comparison-op ::= < | =
simple-exp ::= term [addop term]
addop ::= + | -
term ::= factor [mulop factor]
mulop ::= * | /
factor ::= ( exp ) | number | identifier
```

Figure 5: TINY Grammar.

```
read x ;
if 0 < x then fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact
end       {end of if}
```

Figure 6: Example: TINY Program

concentrating upon the control statements of the language. Discussion of expressions is left to section 6.

4.1 Architecture

The TINY grammar is given in Figure 5, with example program in Figure 6. The architecture derived from the TINY grammar is given in Figure 7. The basic statements in the grammar (read, write, assign, if, repeat) are at the top-level, under the decode node. The "read" statement only uses the identifier store (ie. a read is to a destination variable); the remaining statement types utilise expressions (for condition expressions in the "if" and "repeat" statements). Expressions in turn utilise the identifier store (noting constants are literals within instructions – see section 6.1).

The operation of the architecture is similar to that described in section 3.5. Each of the components implement the appropriate semantics for the associated language statement / construct / expression. The semantics are conventional control flow / arithmetic expression evaluation (further details in [4]).

Note that both the 'if" and "repeat" instructions may change the PC when sending a fetch signal to the decode node.

The TINY grammar permits nested "if" and "repeat" statements. This requires local storage to hold statement context (ie. addresses in the field part of the instruction) until the instruction completes. Local storage is provided within the component. The amount of local storage needed is a static property of the input program, reflecting the maximum nesting of control statements within the program. If the SDLSP were generated for a particular program, such

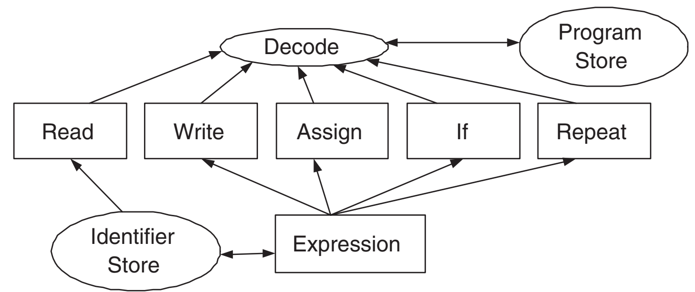

Figure 7: SDLSP for Tiny Language.

	Opcode	Stop	Field	Instruction Width
read	000	0/1	ident_addr	4 + a
write	001	0/1	expr_addr	4 + a
if	010	0/1	else_addr : after_addr	4 + 2a
repeat	011	0/1	expr_addr : after_addr	4 + 2a
assign	100	0/1	ident_addr	4 + a
expr	101	0/1	configuration (*see Section 6* : upto 8 addresses)	4 + c

Table 5: TINY SDLSP Instructions.
Key: a = address width (bits); d = data width (bits); c = expression configuration width

lexical analysis can define the amount of local storage provided. Otherwise, a maximum nesting can be defined by an SDLSP – eg. nesting of 8 depth could be provided, noting that few programs will nest even this deeply. If the local store is exhausted, it needs to save to general memory, although this is not part of the current TINY SDLSP implementation.

4.2 Instructions

Similar to the example in section 3.3, distinct opcodes are required for each of the top-level statements (read, write, assign, if, repeat). Note that a separate opcode is not required to indicate an expression, as the decode component is always aware whether it is loading a statement instruction or an expression instruction (see section 4.1). The basic instructions are defined in Table 5. Instructions for the program of Figure 6 are given in Table 6 (expression fields for TINY are discussed in section 6.1).

5. EXPRESSION EVALUATION FOR SDLSPS

The evaluation of expressions for SDLSPs provides an opportunity to exploit natural parallelism within the source program, by calculating several sub-parts of the expression simultaneously. This raises two main issues: provision of a parallel architecture upon which expressions are evaluated; generation of appropriate instructions.

5.1 Architecture

An initial approach for a syntax-driven implementation of expressions is to essentially mimic the approach taken in compilation for a conventional processor. Each operator is included separately, with two inputs (as all are binary operators) and one output returning the result of the operation. The order of the operators in program store enforces

	Source	Opcode	Stop	Field	Field Comment
0	read x	000	0	0	x in loc. 0
1	if	010	0	-:11	no else
2	expr:0<x	101	1	configuration : 0 : 0	addresses of x,0
3	fact :=	100	0	1	fact in loc. 1
4	expr:1	101	1	configuration : 1	address of constant 1
5	repeat	011	0	10:11	
6	fact :=	100	0	1	fact in loc. 1
7	expr:fact*x	101	1	configuration : 1:0	addresses of fact,x
8	x :=	100	1	0	x in loc. 0
9	expr:x-1	101	1	configuration:0:1	addresses of x,1
10	x=0	101	1	configuration:0:0	addresses of x,0
11	write fact	001	1	1	fact in loc. 1

Table 6: TINY SDLSP Instructions for Figure 6.
Details of expression instruction configuration fields in Table 7

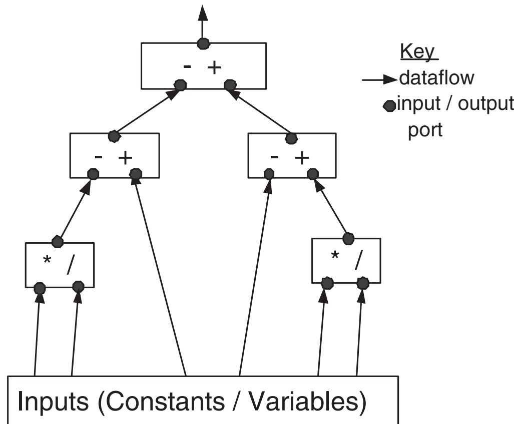

Figure 8: Example Graph-Based Expression Architecture.

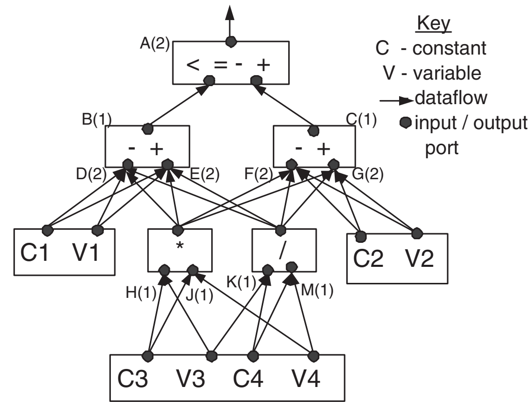

Figure 9: Expression Architecture.

precedence. This approach has no parallelism, in that each operation forming part of the expression is taken sequentially, in a similar manner to CPU operation, with a similar number of instructions (essentially one per operator).

One approach for exploiting available parallelism is to form a graph of operators, with inputs (either constants or variables) flowing from the leaves to the top of the tree in a data-flow manner. By including multiple copies of operators within the graph, parallelism can be exploited. This is shown in Figure 8, where inputs flow from bottom, through operator blocks (which calculate only when both inputs are available), to a returned result. In the figure, pairs of operators are included within the same operator block – largely this is for implementation considerations, as the operators within a block largely share logic.

5.2 Instructions and Operation

For an operator graph such as that shown in Figure 8, an expression instruction field must contain the required configuration of the tree. That is, for each operator block, one operator must be selected; together with the addresses or values of the inputs. For Figure 8, this requires 3 bits of operator block configuration and upto 12 addresses (ie. 6 constants and 6 identifiers). We note that the number of addresses can be reduced in many ways. For example, by utilising a separate bit field to specify whether an address is for constant or identifier store; or by optimising constants within expressions. This is considered further in the next section.

Importantly, both simple (eg. single-operator) or more complex (ie. multi-operator) expressions can be configured. For example, a simple expression can be expanded to one that more naturally maps onto the operator graph:

$$a + b \;=\; ((1 * 0) + a) + (b + (1 * 0))$$

Note that there is a time cost in expanding simple expressions to include relatively expensive multiply operations, although in practice a number of operator graphs could be available (from simple to complex). An appropriate graph could be selected at compile time for each source expression; each expression instruction would identify which graph to be used.

6. CASE STUDY: EXPRESSIONS IN TINY

TINY defines rules of precedence and associativity over arithmetic operators that enable an unambiguous arithmetic expression to be described in the source program, and parsed into an AST by the compiler.

For the TINY SDLSP, one potential architecture for expression evaluation is shown in Figure 9 (which is essentially an expansion of the "Expression" component of Figure 7). Multiple copies of "+" and "-" operators are provided to allow greater evaluation parallelism than if only a single copy of each were provided. The result is that independent parts of the same expression can be evaluated in parallel, but are still expressed within one instruction (see section 6.1) – eg.

	Expression Configuration
2	A < : B + : C + : D C1 : E C1 : F C2 : G V2 : C1 0 : C2 0 : V2 x
4	A + : B + : C + : D C1 : E C1 : F C2 : G C2 : C1 0 : C2 1
7	A + : B + : C + : D C1 : E * : H V3 : J V3 F C2 : G C2 : C1 0 : C2 0 : V3 fact : V4 x
9	A - : B + : C + : D V1 : E C1 : F * : G * H C3 : G C3 : C1 0 : V1 x : C2 1 : C3 0
10	A = : B + : C + : D C1 : E V1 : F C2 : G C2 : C1 0 : V1 x : C2 0

Table 7: Expression Opcode Configurations for Table 6.

consider the following expression (where C represents constants and V variables):
((C3 * V3) + V1) - ((C4 / V4) + V2)
Both the multiply and divide can occur in parallel, and with different variables and constants.

6.1 Instructions for Expressions

For the expression architecture of Figure 9 (and referring to the "expr" instruction in Table 5), the configuration requires 16 control bits to select the required operators and paths – these bits are shown in Figure 9 in parentheses. Note that simple expressions, such as "C+V", can be achieved by utilising constants. The field also contains upto 8 addresses – ie. constant or variable addresses. The exact number of addresses can be inferred from the configuration. Examples of expression opcodes are given in Table 7, where the configuration data corresponds to the labels in Figure 9.

Expressions of arbitrary length must be broken into suitable parts, where each part is of suitable form for the architecture of Figure 9. This can occur by transformation of the source code prior to code-generation, or during code generation by outputting a number of expression instructions for a given source expression. This paper assumes the former.

6.2 Operation

Each expression instruction involves execution of all parts of the expression evaluation architecture (of Figure 9). At the end of an expression instruction, the top-most node (ie. that containing "< = - +" in Figure 9) outputs the result to all higher nodes (see Figure 7). Exactly one of these nodes will be active waiting for an expression result.

7. EVALUATION

The SDLSP for TINY described above has been implemented using Handel-C targeting a Spartan-2 device. For a full SDLSP with 16-bit variables, 1176 LUTs (147 flip-flops) are required. Note that 860 LUTs are due to to utilising Handel-C multiply and divide. In terms of speed, the circuit clocks at around 20MHz for single cycle multiply and divide; 52MHz for multi-cycle.

The number of instructions loaded and executed by SDLSPs compared to other approaches is lower. However, one instruction on an SDLSP is equivalent to many machine instructions on a CPU. This can be seen by noting the TINY source program of Figure 6 compiles to 40 instructions using the TM compiler [4], whilst the TINY SDLSP requires only 12 instructions, of which 5 are expressions (see Table 6). Over many programs (using 16 bit variables and 8 bit store addresses), the SDLSP implementation uses only 32.3% of instructions compared to conventional compilation.

An important comparison in terms of program size (total number of bits required for all instructions) For these programs, the instructions ranged between 12 and 28 bits, mean average 16; expressions between 25 and 41 bits, mean average 34.

Overall, the implemented SDLSP will process a single instruction at the stated frequency, whatever the bit size of the instruction, assuming a sufficiently wide memory (ie. at least that of the widest instruction).

Further comparison is difficult, as the SDLSP implemented is that for a limited language. However, we note that JOP LSP for Java is around a factor of 5 bigger; MicroBlaze and other softcore (32-bit) CPUs are around 1.5 to 5 times as big – all clock at around 70MHz for Spartan-2 (same FPGA as the implemented SDLSP).

8. CONCLUSIONS

The contribution of this paper is an investigation into an alternative implementation approach for software programming languages implemented on FPGAs. Syntax-driven language specific processors (SDLSP) were proposed to interpret a specific software programming language at a statement/construct/expression level, with an architecture based upon the structure of the language grammar.

SDLSPs exploit statement level parallelism (ie. execute independent parts of the same statement in parallel); show good space efficiency for FPGA implementations (due to executing instructions at a high level of abstraction); and execute at a comparable speed to a softcore CPU.

Current work is concentrating on extending the approach to encompass all typical imperative programming language features (eg. procedure/function calls, parameter passing, non-integer data types); and concurrency (eg. thread operations, context-switch).

9. REFERENCES

[1] J. Smith and R. Nair, *Virtual Machines*. Elsevier, 2005.
[2] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Addison-Wesley, 1997.
[3] M. Schoeberl, "A Time Predictable Java Processor," in *Proc. DATE*, 2006, pp. 800–805.
[4] K. C. Louden, *Compiler Construction: Principles and Practice*. PWS, 1997.
[5] M. Reshadi and D. Gajski, "A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths," in *Int. Symp. CODES+ISSS*, 2005.
[6] M. Ward and N. C. Audsley, "Hardware Implementation of the Ravenscar Ada Tasking Profile," in *Proceeedings of CASES 2002*, 2002, pp. 59–68.
[7] D. Pellerin and S. Thibault, *Practical FPGA Programming in C*. Prentice Hall, 2005.
[8] *Celoxica Product Information*, Celoxica Ltd, 2006. [Online]. Available: http://www.celoxica.com
[9] *Lightfoot Product Information*, Digital Communication Technologies, 2006. [Online]. Available: http://www.dctl.com
[10] *Espresso Product Information*, Aurora VLSI, Inc., 2006. [Online]. Available: http://vodka.auroravlsi.com/
[11] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.
[12] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
[13] J. R. Levine, T. Mason, and D. Brown, *Lex and Yacc*. O'Reilly and Associates, Inc., 1992.