Improving on the experience of hand-assembling programs for application-specific processors

Ian Piumarta
Kyoto University of Advanced Science
Kyoto, Japan
ian.piumarta@kuas.ac.jp

ABSTRACT
Creating an application-specific processor is an effective and popular way to solve many problems in embedded hardware design using FPGAs, ASICs, or custom silicon. Programming these processors is complicated by the lack of toolchain support for creating the necessary binary code as part of hardware design, implementation, and evaluation. Hardware developers who cannot create their own ad-hoc assembler are left to hand-assemble their code into binary instructions which is both painful and error prone. We present a tool that supports the rapid creation of assemblers for application-specific processors. A single language is used to specify both instruction formats as collections of bit fields and the instantiation of those formats into sequences of binary instructions as a single, homogeneous activity that is designed to be as familiar and accessible to hardware designers as possible. The output from the tool can be used directly by hardware synthesis tools to initialise the program memory of an application-specific processor.

KEYWORDS
embedded programming, configuration memory, state machines, microcode, micro assemblers, macro languages, application specific computers, domain specific languages

1 INTRODUCTION
Application-specific processors, in the form of finite state machines, can be found in many embedded hardware designs. Small state machines, in which states represent the progress through a task with simple external inputs, are easy to design and implement using a hardware description language (HDL). A serial transceiver can be implemented in a page of HDL using perhaps a dozen states whose sequencing depends on only a couple of external inputs [6]. Control outputs from this kind of machine, for example ‘data received’, can be encoded as part of the current state number.

At the other extreme of complexity are embedded processors that implement general-purpose instruction sets and which interpret machine code programs embedded in the hardware device. These can also be easily incorporated into larger designs when the instruction set architecture is stable and assemblers for it are readily available. Popular examples include the Xilinx PicoBlaze and MicroBlaze soft-core RISC processors for which several assembler implementations are freely available [13]. Sequencing in these processors uses a combination of program counter and the contents of program store (indexed by the program counter) that contain instructions specifying how the next state, including the next program counter value, should be calculated. Fields within the instructions control which register values should be tested or combined, which arithmetic operation to use, and where to store any outputs generated.

Between these two extremes is a huge space of domain- and application-specific processor designs that combine elements of both approaches. Examples include sequences of control signal patterns, large state machine tables, or microcode that interprets the ‘vertically encoded’ instructions of a CISC processor. Microcode instruction words, for example, tend to be wide, regular, and explicit, with different fields of instruction bits connected directly to control inputs of components in the surrounding hardware. Such instructions might contain a field of N bits that are connected directly the ‘write-enable’ inputs of the N registers in the processor’s implementation, allowing any combination of the registers to be updated at the end of the current microcycle.

Unlike common general-purpose processors these application-specific instruction processors (ASIPs) have no standard assembly language or assemblers to convert a symbolic representation of an assembly language program into binary form. Writing a dedicated assembler causes additional work for a developer who is not necessarily a software programmer, introduces another source of potential bugs, and results in a single-use tool that may never be reused beyond the original project. Application-specific programs for these processors may well be hand-assembled, meaning the designer writes the programs using some form of symbolic notation that is converted manually into the binary constants representing the actual instructions.

Programmable hardware such as the Virtex-7 FPGAs from Xilinx [15] can contain as much as 68 megabits of on-chip
memory suitable for storing firmware programs written in application-specific instructions [14, Ch. 1].

Manual construction of instructions as binary constants is error-prone, assembling large programs is tedious, and even small modifications to the program cause difficulties such as recalculating branch target addresses that cross regions where instructions were added or removed. Instruction formats can also change, sometimes rapidly, as requirements change or when new features are evaluated experimentally by exploratory implementation which necessitates re-assembling the entire program, even though the program itself may not have changed.

Hand-assembling programs for these processors therefore poses problems of reliability, scalability, and maintainability. Hardware description languages (HDLs) offer little help. Within a VHDL project, hand-assembled instructions might often appear as a list of hexadecimal constants.

```vhdl
type ram_t is array (0 to 7) of std_logic_vector(31 downto 0);
signal ROM : ram_t := (
    ... X"2310", X"203B", X"8300", X"4002",
    ...
);
```

Verilog’s bit field concatenation operator does allow the designer to assemble instructions field-by-field, revealing some of the structure within each word and helping make the process a little less painful and possibly a little less error-prone. Each explicitly sized bit field inside curly braces is concatenated with those adjacent to it to form a correspondingly wider value. While not ideal this can be applied to simple, regular instruction formats such as in this excerpt of an embedded RISC-V [10] softcore processor program.

```vhdl
reg [31:0] rom [0:1023]; // program ROM in block RAM initial begin
    ...
    // ADDI x2, x0, 1
    rom[100] = (12'd001, 5'd0, 3'b000, 5'd2, 7'b0010011);
    // LW x1, x2, 0
    rom[101] = (12'd000, 5'd2, 3'b010, 5'd1, 7'b0000011);
end
```

Here a read-only memory (ROM) containing the device’s firmware is declared as an 1024-element array of 32-bit ‘registers’. The initial block then initialises the contents of the ROM, using binary representations of the processor instructions in the firmware program. The two instructions shown occupy the 100th and 101st words in the ROM.

Even in this example the designer has to remember that these particular instructions contain, from left to right, a 12-bit signed constant (immediate value for ‘add immediate’ or address offset for ‘load word’), five bits containing the source register number, three bits of the (ten-bit) opcode, a five-bit destination register number, and the remaining seven bits of the opcode.

A significant technical challenge of a tool that might help to assemble the above constants symbolically is providing almost total flexibility in how those constants are represented while remaining simple to understand and use. Total flexibility is necessary because the structure and content of application-specific instructions is unpredictable, as is the amount of implicit information that the instruction might contain versus information given explicitly as ‘operands’ to that instruction. In the RISC-V example above, each instruction has both an ‘opcode’ and a ‘function’ number which together specify the operation to be performed. Whether the programmer has to write both of these values explicitly, using symbolic names, or whether the values are both implicitly contained within a single ADDI instruction (for example) is entirely a design decision that should be made by the programmer and not by the implementer of the supporting tool; the tool therefore cannot make any assumptions about instruction structure, such as the use of exactly one opcode or whether operands are written in the same order as the corresponding bit fields within the assembled instruction.

Better language support might therefore allow the designer to assign symbolic names to the bit fields that concatenate to form an instruction, as well as symbolic names for the different values that populate those fields.

```vhdl
```

The structure of the ADDI and LW instructions can then be described in terms of these fields, along with the specific values for the fn3 and op fields.

```vhdl
ADDI = fn3 0b000 0 0b0010011 rd rs imm
LW = fn3 0b010 0 0b0000011 rd rs imm
x0 = 0
x1 = 1
x2 = 2
```

Having also included some symbolic constants for the register numbers, the above memory initialisation might now be written in an entirely symbolic way.

```vhdl
ADDI x2 x0 1
LW x1 x2 0
```

(Here, for example, x2, x0, and 1 provide the ‘missing’ values for the rd, rs, and imm fields in the description of the ADDI instruction. We chose to make the op and fn3 values be provided implicitly by the symbolic instruction name, but that decision was ours alone and was not imposed on us by the semantics or syntax of the language support that we are using.)

In this paper we describe a tool whose goal is to help the hardware designer to achieve exactly this kind of flexible,
symbolic representation of binary constants and in particular when they represent instructions for an application-specific processor. Given an instruction set description, similar to the two example instructions above, and a program written using that instruction set, our tool produces a memory initialisation file containing the corresponding binary constants that can be included directly in the HDL program.

Ad-hoc assemblers for user-defined ISAs have been written in high-level languages. An example of a small, educational, open-source, embedded CPU and its ad-hoc assembler (written in Java) can be found at Alchitry [9]. An assembler for a general-purpose ISA (such as RISC-V) would be considerably larger and more difficult to modify, making reuse in a dedicated processor project difficult.

3 DESIGN

Our goal is to produce an HDL-compatible external data file containing binary encoded instructions for a user-defined instruction set in the most reliable, scalable, and maintainable manner possible. It should be simple to operate and integrate easily into existing workflows and tool chains.

The tool will implement a language for specifying the bit-level structure of data words to be generated, giving symbolic names to fields within structures and to the values populating them. Output is generated by instantiating the patterns with specific values for their fields. To remain flexible, it should make few assumptions about the instructions being assembled or the way they will be described symbolically.

The description language should look familiar to hardware designers, introduce minimal syntax and semantics, and present a programming model and paradigm that is easy to understand.

A secondary goal is to make the implementation of the tool easy to understand and modify, should the need arise. The use of simple, self-contained data structures and a small parsing expression grammar [3] for input tokenisation are steps towards providing this level of accessibility.

3.1 Language design

The language design aspires to scope, economy, and elegance. Scope is not artificially restricted by making assumptions about, or placing restrictions on, the way a designer can choose to describe binary structures. Economy means introducing only the necessary and sufficient concepts and values needed to fulfill its role, without redundancy, using the simplest syntax. To be elegant it should make the relationships between concepts and values clear and natural.

We illustrate the design of the language by considering how instruction descriptions (bit-level structure and field encoding) are typically presented in a design document and then how they are used to manually assemble a binary instruction. The language design will then follow from the description and assembly process, constrained by the three principles explained in the previous paragraph.

A running example will create a binary file containing instructions for a hypothetical, 16-bit, two-operand, RISC-like, embedded CPU. While this is considerably simpler than

```plaintext
reg [31:0] rom [0:1023]; // program ROM in block RAM
initial begin
$readmemb("rom.data", rom, 0, 63); // assembled data
end
```

The remainder of this paper is structured as follows. In Section 2 we present some related work and context. Section 3 discusses the design goals of our language and tool, and how the design tries to attain those goals. Section 4 evaluates our tool relative to the design goals. Section 5 discusses the strengths and limitations of our language and tool. Section 6 offers some concluding remarks and briefly outlines our current and future plans for the language and tool.

2 RELATED WORK

Compilers for simple state machines have been written, for example Ragel [7]. They generate a software state machine as a source program that can be included in traditional software programs. They are not designed to produce the kind of table-driven state machine that is the basis of sequential control in an FPGA or soft-core processor.

Very little work has been published on the development of micro-assemblers for user-defined instruction sets. Most micro assemblers were written by manufacturer in the 1970s and early 1980s, were specific to a single machine architecture, and were supplied only to customers of those architectures. A survey of the history of micro-code and micro-assemblers can be found in [2].

Assembler generators are available as components within large tool-chains supporting the design of System-on-Chip (SoC) devices and hardware accelerators. Commercial systems include Synopsis ASIP Designer [12] which uses a formal ISA description language to create architectures powerful enough to run compiled C programs, including an assembler for their specialised machine code. Research in ASIPs also concentrates on generating SoC-level ASIPs specified using formal description languages, in many cases producing a design including data paths and control logic. The assembler is a by product of these approaches, not the main result.
a microcode instruction set, the concepts and their application are identical. Only their quantity would be increased in the microcode instruction set.

Arithmetic and logical instructions in our 16-bit architecture will consist of a 6-bit opcode, occupying the most significant six bits, followed by two 5-bit register numbers. A design document might well present that format using the following diagram.

```
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**opcode** | **destination d** | **source s**

Hardware designers are familiar with a square bracket notation for bit fields. We could write the opcode field in a familiar way as "[15 10]". Placing the three field descriptions together on one line even makes a semi-pictorial description of the instruction.

```
[15 10][9 5][4 0]
```

Populating the opcode, d, and s fields with values yields a finished binary instruction. If the opcode for addition is 42 then the design document might describe the ADD instruction as follows.¹

```
ADD Rd, Rs
```

**opcode** = 42
**destination** = Rd
**source** = Rs

To add register 6 to register 7 in an embedded program we would hand-assemble the instruction 'add r7, r6' by converting the opcode and register names to numbers: 42 (addition), 7 (destination), and 6 (source). The numbers are inserted into the format by shifting them left by 10, 5, and 0 bits, respectively.

```
```

Values too large to fit into the width of a field are truncated so their least significant bits will fit, which simplifies the handling of negative numbers. The output word is sized implicitly by the most significant bit position appearing in any of the field descriptions. Bits not explicitly set in the output word contain zeroes. (Options to issue a warning for unset bits, for bits set more than once, or for output words that are not a specified length, are potentially useful for debugging an instruction set description but are not provided in the prototype described here.)

```
[7 0] 11 ⇒ 00001011
[0 7] 11 ⇒ 11010000
```

With respect to the above items, our language now has two distinct entities (item 2): the *bit field* descriptions and numeric values that populate them. Our tool generates binary output (item 1) from a sequences of those entities. The semantics are easy to grasp: bit fields dictate where values should be placed in the output word (and how wide they are allowed to be). A bit field can be thought of as a ‘function’ that takes one ‘argument’, a numeric value, and inserts it into the output word. Applying the three bit field ‘functions’ to the three ‘argument’ values shown above generates the binary form of the ‘add r7, r6’ instruction. Assigning symbolic names to sequences of entities (Section 3.2) will let the hardware designer write ‘add r7 r6’ instead of ‘[15 10] 42 [9 5] 7 [4 0] 6’ (item 3).

Given one line of input ‘[15 10] 42 [9 5] 7 [4 0] 6’ our tool produces one line of output ‘1010100011100110’ by inserting the three values (42, 7, and 6) into the three corresponding fields ([15 10], [9 5], and [4 0]) in the output word.

Overlapping fields are allowed and any bit position written more than once will be updated accordingly. Single bits can be written by setting $h$ and $l$ to the same bit number in the "[h l]" syntax. Integer values and bit field contents are all represented with most-significant bit first, which is typically how we write binary numbers and instruction formats on paper. Reversing the bit numbers in a field, writing the lower bit number first, causes the contents of the field to be filled in reverse order which supports filling LS bit first fields within instructions.

```
[7 0] 11 ⇒ 00001011
[0 7] 11 ⇒ 11010000
```

```
[7 0] 11 ⇒ 00001011
[0 7] 11 ⇒ 11010000
```

3.2 Macros

Writing bit field descriptions and literal values such as ‘[15 10] 42 [9 5] 7 [4 0] 6’ is an improvement over writing ‘1010100011100110’, but is still prone to errors and not conducive to changes in (for example) opcode encodings. Our language provides a macro system that allows developers to give symbolic names to sequences of entities. A macro definition consists of a back-tick character followed by the name of the macro being defined and then a replacement

¹The order of fields in the binary format can, but need not, be the same as the order of corresponding operands in the assembly language. In this example the source and destination could appear in the assembly language representation in the opposite order, by swapping the order of the fields ‘[9 5]’ and ‘[4 0]’, along with their corresponding values. The final assembled binary instruction would be identical.
list which is a sequence of zero or more entities. Whenever
the name appears in a program it is removed and the cor-
responding replacement list of entities inserted in its place. After the definition
\`op_add 42
we could subsequently write our addition instruction as
\[15 10\] op_add \[9 5\] 7 \[4 0\] 6
⇒ \[15 10\] 42 \[9 5\] 7 \[4 0\] 6
⇒ 1010100011100110
Furthermore, given the definition
\`opcode \[15 10\]
we could write
opcode op_add \[9 5\] 7 \[4 0\] 6
⇒ \[15 10\] 42 \[9 5\] 7 \[4 0\] 6
⇒ 1010100111000110
The sequence of tokens making up the replacement text
of the macro extends until the end of the source line or the
next back-tick character. This permits multiple definitions
to appear on one line, which in turn allows a semi-pictorial
layout when describing bit fields within a format. Defini-
tions for all three bit fields (opcode, destination, source) can
therefore be written
\`opcode [15 10] \`rd [9 5] \`rs [4 0]
which is very close to (and easy to transcribe from) the de-
sign documentation diagram shown above. Our example in-
struction can now be written like this:
opcode 42 rd rs 7 6
⇒ \[15 10\] 42 \[9 5\] \[4 0\] 7 6
⇒ \[15 10\] 42 \[9 5\] \[4 0\] \[9 5\] \[4 0\] 7 6
⇒ 1010100011100110

3.3 Order of entities
We can now almost write an instruction ‘template’ in a nat-
ural and obvious way. Consider this template for the add
instruction.
\`add opcode 42 rd rs
Using the add instruction in a program would result in the
following macro replacements.
add 7 6
⇒ opcode 42 rd rs 7 6
⇒ \[15 10\] 42 \[9 5\] \[4 0\] 7 6
⇒ \[15 10\] 42 \[9 5\] \[4 0\] \[9 5\] \[4 0\] 7 6
The bit field "[9 5]" is separated from its argument "7"
by an intervening bit field "[4 0]". Does this matter? Only
if the language arbitrarily requires each ‘function’ bit field
to be followed immediately by its value ‘argument’. Since
there is no reason to impose that requirement, we don’t.
The syntax and semantics of the language can now be
expressed as follows. Each (non empty) line either contains
one or more definitions, or outputs a single binary instruc-
tion constructed from one or more bit fields and their cor-
responding values. Each bit field in the line, read from left
to right, is matched with a corresponding value in the line,
read from left to right. The only syntactic rule is that the
number of fields and values must be equal; how they are
interleaved is irrelevant.
Conceptually the tool collects all the bit field descriptions
(preserving their order) and all values (preserving their or-
der), ‘zips’ the two sequences together to make field-value
pairs (the fourth expansion step shown below), then applies
each field to its attached value.
add 7 6
⇒ opcode 42 rd rs 7 6
⇒ \[15 10\] 42 \[9 5\] \[4 0\] 7 6
⇒ \[15 10\] \[9 5\] \[4 0\] 42 7 6
⇒ \[15 10\] 42 \[9 5\] \[4 0\] \[9 5\] \[4 0\] 7 6
⇒ 1010100011100110
This flexibility in how fields and values are interleaved af-
fords the designer much freedom to create natural abstrac-
tions describing the structure of instructions. Continuing
the running example, a more convenient and scalable defi-
nition of the arithmetic and logical instruction format might
be this:
\`opcode [15 10] \`rd [9 5] \`rs [4 0]
`alu_op opcode rd rs
`add alu_op 42
`sub alu_op 43
`and alu_op 44
add 7 6
⇒ alu_op 42 rd rs 7 6
⇒ \[15 10\] \[9 5\] \[4 0\] \[9 5\] \[4 0\] 7 6
⇒ \[15 10\] \[9 5\] \[4 0\] \[9 5\] \[4 0\] 7 6
⇒ \[15 10\] \[9 5\] \[4 0\] \[9 5\] \[4 0\] \[9 5\] \[4 0\] 7 6
⇒ 1010100011100110
Here first line defines three instruction fields, the second
line defines an instruction format called ‘alu_op’ that uses
those three fields, and the third through fifth lines define
three instructions using the ‘alu_op’ format. Note that the
only thing that differs between these three instructions is
their opcode value; the fields rd and rs, which appear in
the instruction format, are supplied with values later when
the ‘program’ finally uses the add, sub, or and macros.

3.4 Labels
Labels associate a symbol with a location in the program. They are provided using the same macro definition mecha-
nism described above. If an undefined name appears first on
a line immediately followed by a colon, that name is defined
as a macro whose value is a single integer constant whose
value is equal to the number of words that have been output
so far.
\[7 0\] 0
⇒ 00000000
the use of that instruction. Similarly when the ‘addr’ value is supplied as an operand in
structures and interpretations that are possible.

drawing multiple instruction diagrams, one for each of the
the opcode field. Documentation typically reflects this by
icant 10 bits of the instruction depending on the value of
structures and interpretations can be placed on the least signif-
the instruction set of our embedded CPU. Multiple struc-
Jump instructions introduced an element of
bit steering into

3.5 Multiple instruction formats

In our running example, opcodes 1, 2, and 3 might be jumps
to a 10-bit, absolute location. The addr field is 10 bits wide
and occupies the same bits as the rd and rs fields. It can
be written pictorially, under the rd and rs fields, occupying
the same number of columns, in the same way that it might
appear in a design document.

```
opcode [15 10] 'rd [9 5] 'rs [4 0]
opcode [15 10] 'jump_op opcode rd rs
jump_op opcode addr
jmp jump_op 1
jeq jump_op 2
jilt jump_op 3
...'
add alu_op 42
loop: add  7  6       ⇒ 01010 00111 00110
     jeq 42                ⇒ 00001 00001 01010
    jmp  loop             ⇒ 00001 00000 00000
```

Here the first line defines an instruction format for
register-register instructions and the second line defines an instruction format for jump instructions that take a 10-bit address
as operand. The third and fourth lines give names to those
formats. The next three lines define three instructions using
the ‘jump’ format and that differ only in their opcode.
Line 9 defines an instruction using the ‘alu’ format using
opcode 42. When any of the ‘jump’ format instructions
are used, the missing ‘addr’ value is supplied as an operand in
the use of that instruction. Similarly when the ‘add’ instruction
is used, the missing ‘rd’ and ‘rs’ values are supplied as operands in each use of that instruction.

3.6 Symbol redefinition

Jump instructions introduced an element of bit steering into
the instruction set of our embedded CPU. Multiple structures
and interpretations can be placed on the least significant
10 bits of the instruction depending on the value of
the opcode field. Documentation typically reflects this by
drawing multiple instruction diagrams, one for each of the
structures and interpretations that are possible.

| 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|---|---|---|
| opcode | dest d | source s |
| opcode | address a |

Transcribed to bit field descriptions, we have:

```
`opcode [15 10] 'rd [9 5] 'rs [4 0]
`opcode [15 10] 'addr [9 0]
```

Disallowing the redefinition of opcode would be the sim-
plest implementation but would prevent this kind of picto-
rial representation of instruction structure. We therefore al-
low a symbol to be redefined provided that the second (and
any subsequent) definitions are identical to the first defini-
tion, permitting the limited (but representationally invalu-
able) redefinition shown above.

3.7 Literals and comments

For convenience numeric literals can also be written in hexa-
decimal, octal, or binary using C-like or Verilog-like syntax.
C-like literals are written with a prefix of “0x”, “0”, or “0b” to
indicate hexadecimal, octal, or binary, respectively. Verilog-
like literals are written with a prefix of “n’h”, “n’o”, or “n’b”,
for hexadecimal, octal, or binary, respectively, or with a pre-
fix of “n’d” to indicate the default decimal format. (The “n”
can be any decimal number. In Verilog it specifies the num-
ber of bits in the value; we ignore it but allow it to be present
for compatibility with Verilog syntax.)

3.8 Diagnostics

When the input cannot be parsed a syntax error is reported
that consists of the name of the current input file, the line
number, and all input text on the line following the last suc-
cessfully parsed definition or expression. For example, at-
tempts to parse

```
`opcode [15 10] '42 [9 5] 'rs [4 0]
```

on line 7 of Example.m3 causes the language to print
Example.m3:7 syntax error near: ‘42 [9 5] ‘rs [4 0]
and then exit. Syntactically invalid label definitions, bit field
specifications or literals will also cause a syntax error.

An incompatible redefinition of a macro causes a similar
error. If the first two lines of example.m3 contain

```
`opcode [15 10]
`opcode [15 11]
```

then processing the file with our tool causes it to print
Example.m3:2 multiple definition of '/opcode' and
then exit. The same mechanism detects and disallows
the redefinition of a label at multiple locations.

4 EVALUATION

Our language and tool have been used successfully in a num-
er of projects. An early version was used to prepare short
‘ROM’ programs (such as bootloaders) for an experimental
RISC-V core running on FPGAs. Reports from those users
4.1 A Complete Example: TOY

Sedgewick and Wayne developed the Princeton TOY computer for teaching computer architecture. For educational purposes we implemented the architecture in Verilog and used our language and tool to help us write programs for it running both under simulation and in FPGA hardware.

Figure 1 shows the definitions of the instruction formats and instructions of the TOY architecture. A 4-bit opcode in the most significant bits steers the interpretation of the least significant 8 bits. Figure 2 shows the transcription of this specification into our language along with a small example program that outputs a sequence of integers to a memory-mapped output port. Running this file through our tool produces the following binary output:

```
0111000100000001
0111010000000000
1001000111111111
0111011111111111
0010000110010001
1101001100000010
0001000100000001
1111000000000010
```

This output is included in a Verilog source file using an initial `readmemb` task and initializes the contents of block RAM acting as program ROM. The following `Makefile` entries incorporate the tool into our workflow.

```
make toy

iverilog -o toy toy.v

mv toy.m3 > toy.m3
```

(Our tool is called `m3` in the above example.)

4.2 Implementation

The tool implementation is less than 400 lines of C code.
We believe that our language and tool met their initial design goals and they have already proven themselves valuable in several projects, greatly reducing development time while significantly increasing reliability and maintainability.

5 DISCUSSION

We believe that our language and tool met their initial design goals and they have already proven themselves valuable in several projects, greatly reducing development time while significantly increasing reliability and maintainability.

Several potentially useful additions could be made quickly and easily to the language by any programmer who can read and understand the short parsing expression grammar and its parser actions (written in C).

- Allowing symbolic values to be used as bit field indices.
- Allowing fields to be specified as bit position plus length.
- Adding a directive to include other source files. This could be useful if maintaining a single file describing instruction formats and register names for a given ISA, subsequently included in multiple program files.
- Adding command-line options to output binary in other formats.

Conversely the scope of our language is hindered by several shortcomings.

- Macros have no arguments, although we are not sure that this is actually a shortcoming (because there is no evaluation of arithmetic expressions and successive values can be placed into a bit field located anywhere in the output word).
- Macro assembler constructs from more traditional assemblers are not supported, e.g: looping to repeatedly generate a sequence of words, conditional sections, etc. This would be needed to implement alignment.
- There is no provision to include other source files. For example, preparing a single file specifying bit field and register names for a given ISA and then including it in multiple application-specific program files.
- The output is simple, absolute binary in a text file. No binary libraries or linking is available, although we are not convinced that this is a limitation either (because the programs we are targeting are self-contained, the tool is very fast, and code reuse and sharing can easily be done by composing source files together).

6 CONCLUSION

Several indicators suggest a renewed and growing interest in hardware-software co-design. Intel’s 2015 acquisition of Altera demonstrates a belief within the embedded industry that FPGAs have a future as accelerators for traditional CPUs on a single die. High-speed FPGAs (such as those from Achronix [1]) can almost eliminate the gap in performance between an ASIC and a soft-core CPU, and in that case there is no reason to use a standard ISA for the end-user programmable part of the system. AI/ML [5] and cryptocurrency [4] applications are also starting to use programmable hardware for performance gains, as are 5G network designers. The ability to design and rapidly prototype with new ISAs, and to evolve those ISAs painlessly as new requirements emerge, will be critical. We believe that the kind of language and tool presented here will be invaluable in these
application areas, especially during a research and rapid prototyping phase.

We are currently developing a new language and tool that address all the shortcomings mentioned in the previous section. The major difficulty is not technical but rather conceptual: how to design and integrate the missing features while preserving the simplicity and flexibility of both the language and the tool.

DOWNLOAD
The tool and short demonstrations as shown in this text can be downloaded from: https://piumarta.com/tmp/px21.tgz

ACKNOWLEDGEMENTS
The author would like to thank James Noble and the anonymous referees for their very helpful comments and feedback on earlier drafts of this paper.

REFERENCES

A LANGUAGE GRAMMAR
Figure 3 presents the parsing expression grammar for the language described in this paper. The grammar and parser actions are taken ‘verbatim’ from the source code, with a handful of small edits to improve readability.

B GLOSSARY
This section presents a glossary of abbreviations and technical terms used in the text that are related to hardware and hardware description languages.

5G Fifth-generation, high-capacity wireless telephony system. Application-specific hardware (in the form of ASICs) is often used in the base stations of these networks.

AI Artificial Intelligence. ASIPs can be used to implement algorithms for highly parallel computations used by (e.g.) neural networks. Intel acquired Altera (an FPGA manufacturer) so they could integrate FPGAs into server CPUs to accelerate AI applications.

ASIC Application Specific Integrated Circuit. An integrated circuit performing a function dedicated to one application or application domain. ASICs are faster than FPGAs but are not programmable.

ASIP Application Specific Instruction Processor. A processor that implements instructions whose behaviour supports a particular application or domain particularly efficiently. Programming tools (such as compilers) and the implementation of the ASIP are often generated together from the same specification.

Bit steering
The use of one bit field within an instruction (typically the opcode, or part of it) to select which format (location and meaning of the remaining bit fields) is used to decode the instruction.

CISC Complex Instruction Set Computer. A computer in which each instruction is complex and performs a lot of work but runs relatively slowly compared to a RISC computer.

CPU Central Processing Unit. The part of the computer that performs computation by fetching, decoding, and executing program instructions in order to
Figure 3: The grammar of the m3 language written as a parsing expression grammar [3] for the ‘leg’ parser generator [8]. Within the parser actions the global ‘yylval’ contains a token representing the parsed line, ‘tokens’ is a temporary list used to build lists of other tokens, input text matched by rules between ‘<' and '>' characters is made available to parser actions as a string called ‘yytext’, ‘-’ is a valid rule name that matches zero or more spaces, and the ‘error’ rule prints a syntax error message using ‘context’ to collect any unparsed source code from the current input line and ‘@’ to make the following action run immediately during (rather than after successful) parsing.
move data around or to perform arithmetic and logical operations on it.

**FPGA** Field Programmable Gate Array. An integrated circuit that contains many general purpose elements whose logical functions and inter-element connections can be configured dynamically, for example when power is applied to a system.

**HDL** Hardware Description Language. A language intended to describe hardware designs at the gate and register level. Verilog and VHDL are two well known examples that can be used to produce integrated circuits, configuration data for FPGAs, or simulations of hardware for testing and verification.

**ISA** Instruction Set Architecture. An abstract model of a computer and its behaviour, including the instruction set and (typically) its binary encoding, independent of any particular implementation. (‘amd64’ or ‘x86_64’ is a single ISA having multiple implementations provided by companies such as AMD and Intel.)

**ML** Machine Learning. A subdomain of AI in which a hardware or software system performs a specific task with increasing precision or accuracy as the number of cases it considers increases. The training of ML systems involves repeated massively parallel computations performed on huge data sets and can benefit greatly from application-specific hardware or instruction processing.

**RAM** Random Access Memory. Volatile memory that contains transient data or programs that can change rapidly.

**RISC** Reduced Instruction Set Computer. A computer in which each instruction is simple and performs very little work but runs relatively fast compared to a CISC computer.

**ROM** Read Only Memory. Persistent memory that contains firmware or other fixed instructions that do not change.

**SoC** System on a Chip. A complete computer on a single integrated circuit providing most or all of CPU cores, peripheral devices, bus controllers, graphics accelerator, and RAM.