A Verilog Parser in ACL2

Jared Davis

Centaur Technology

September 10, 2008

Introduction

A preprocessor, lexer, and parser for Verilog 2005 (IEEE-1364)

- Basically complete for modules

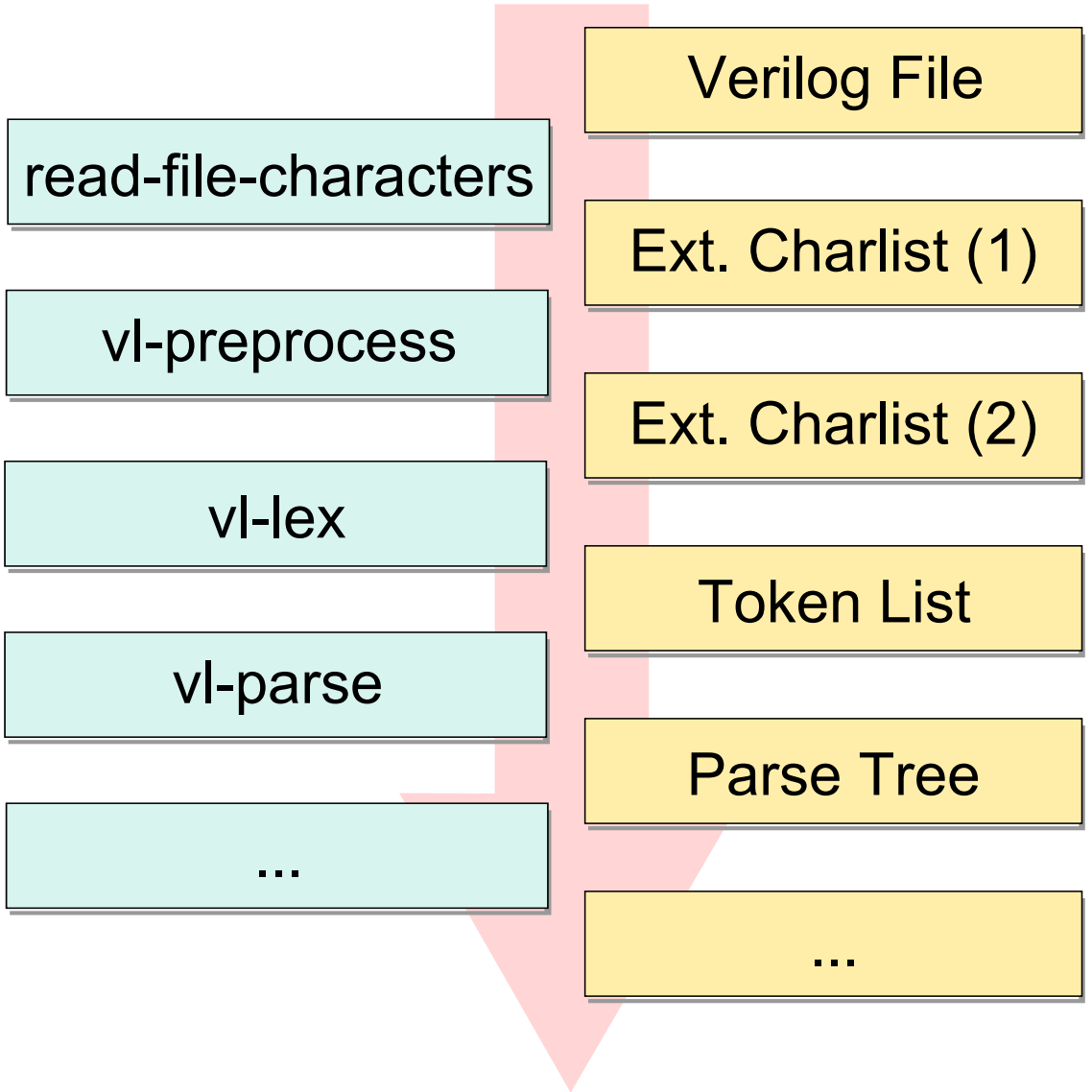
Verilog is a pretty big language

- Long history, many-level modelling, simulation mixed in
- Preprocessor, 125 keywords, 50 other token types, 20-page grammar

Primary concern: correct translation

- Simplicity over performance
- Elaborate well-formedness checks
- Mostly in logic-mode with verified guards
- Unit tests to promote semantic correctness

High-level design



Results

Performance is quite acceptable (550K LOC)

Read top.v	6s	2.6 GB
Preprocess top.v	4s	1 GB
Lex top.v	28s	2.5 GB
Parse top.v	20s	1.4 GB
Load libraries	20s	1.7 GB
Total	78s	9.3 GB

Working on making an ACL2 image with the chip pre-loaded

Outline

Reading files

Verilog sources are just ASCII text files

We read in whole files as lists of extended characters

- $\langle character, filename, line, column \rangle$

Inefficient, but has advantages:

- Minimizes use of state
- Automatic position tracking
- Easy to write tests for list-based tools

The preprocessor

Verilog has a C-style preprocessor

- define and undef for constants
- nestable ifdef, ifndef, elsif, else
- other stuff that we don't support (like include)

vl-preprocess : echarlist $\longrightarrow$ successp $\times$ echarlist

- Program mode
- Reuses some lexer routines
- 1,200 lines (about 45% comments and whitespace)
- Includes 250 lines of unit tests

Preprocessor implementation

Woefully underspecified, so we try to defensively mimic Cadence

```
‘ifdef foo
‘define myendif ‘endif
‘myendif
```

```
‘define a 0
‘define b ‘a
‘define a 1
wire w = ‘b ;
```

Not a blind textual substitution

```
// comment about ‘ifdef
$display("The value of ‘a’ is %d\n", a);
```

No nice way to relate input and output

The lexer

The lexer is quite basic.

- "Optimized" based on the first-remaining character
- Verilog is pretty amenable to this

vl-lex : echarlist $\rightarrow$ successp $\times$ tokenlist

- A mixture of program and logic mode
- 1400 lines (about 40% comments and whitespace)
- Includes 400 lines of unit tests
- About 40% is related to numbers

Written with some "theorems" in mind:

- tokenlistp(output)
- flatten(output) = input

Token definition

Our lexer produces a list of *tokens*.

- Plain tokens (ws, comments, operators, punctuation, keywords)
- String, real, and integer literals
- Identifiers and system identifiers

Each token has

- A symbolic type (which can be accessed quickly)
- The echars the lexer created it from

We have some basic well-formedness checks, e.g., an integer literal should be in the specified range.

Lexer utilities: literals

Reasonably-efficient utilities for handling literals

- **vl-matches-string-p** : string $\times$ echars $\rightarrow$ bool
- **vl-read-literal** : string $\times$ echars $\rightarrow$ prefix $\times$ remainder
- **vl-read-some-literal** : strings $\times$ echars $\rightarrow$ prefix $\times$ remainder
- **vl-read-until-literal** : string $\times$ echars $\rightarrow$ bool $\times$ prefix $\times$ remainder
- **vl-read-through-literal** : string $\times$ echars $\rightarrow$ bool $\times$ prefix $\times$ remainder

We also prove some basic theorems about these

Lexer utilities: defchar

We also automate the introduction of character types. For instance:

```
(defchar whitespace
  (or (eql x #\Space)
      (eql x #\Tab)
      (eql x #\Newline)
      (eql x #\Page)))
```

Introduces efficient functions (w/ theorems):

- **vl-whitespace-p** : characterp $\longrightarrow$ bool
- **vl-whitespace-echar-p** : echar $\longrightarrow$ bool
- **vl-whitespace-list-p** : character-listp $\longrightarrow$ bool
- **vl-read-while-whitespace** : echars $\longrightarrow$ nil $\times$ prefix $\times$ remainder

Classic recursive-descent parsing

$$\textit{range} ::= \textbf{[}\ \textit{expression}\ \textbf{:}\ \textit{expression}\ \textbf{]}$$

```
File in;

Token match_token(type) {
    Token t = lex();
    if (t.getType() == type) return t;
    throw new Exception("Expected " + type);
}

Range parse_range() {
    match_token(LBRACK);
    Expression e1 = parse_expression();
    match_token(COLON);
    Expression e2 = parse_expression();
    match_token(RBRACK);
    return new Range(e1, e2);
}
```

Observations

A reasonable way to write parsers

- follows the grammar rule quite closely
- implicitly propagates errors upwards (nice)
- implicitly advances through the file (nice)

Let me emphasize these last two points:

- parse_range can fail (by propagating an exception)
- parse_range changes state (the file pointer)

Not straightforward to do in ACL2.

Explicit state and exception handling = pain

```
(defun parse-range (tokens)
  (mv-let (err val tokens) (match-token :LBRACK tokens)
   (if err
       (mv err val tokens)
     (mv-let (err e1 tokens) (parse-expression tokens)
      (if err
          (mv err e1 tokens)
        (mv-let (err val tokens) (match-token :COLON tokens)
         (if err
             (mv err val tokens)
           (mv-let (err e2 tokens) (parse-expression tokens)
            (if err
                (mv err e2 tokens)
              (mv-let (err val tokens) (match-token :RBRACK tokens)
               (if err
                   (mv err val tokens)
                 (mv nil (make-range e1 e2) tokens)))))))))))
```

The SEQ language

SEQ is a macro-language for writing parsers

- Makes exception-propagation implicit
- Makes advancing the token list implicit

```
(defun parse-range (tokens)
  (declare (xargs :guard (tokenlistp tokens)))
  (seq tokens
       (:= (match-token :LBRACK tokens))
       (e1 := (parse-expression tokens))
       (:= (match-token :COLON tokens))
       (e2 := (parse-expression tokens))
       (:= (match-token :RBRACK tokens))
       (return (make-range e1 e2))))
```

Actions and streams

SEQ is a language for applying *actions* to a *stream*.

Actions are ACL2 expressions which return

$$(\texttt{mv}\ \mathit{error}\ \mathit{val}\ \mathit{stream}')$$

Where

- *error* is non-nil if an error has occurred,
- *val* is the return value of this action
- stream' is the updated stream

Streams are basically any ACL2 object you wish to sequentially update

SEQ language: returns

Every seq program must end with a return statement.

Successful returns

```
(return (foo ...))
  --->
(mv nil (foo ...) streamname)
```

Raw returns

```
(return-raw (mv "Bad!" nil streamname))
  --->
(mv "Bad!" nil streamname)
```

SEQ language: voids

Void Statements can update the stream and cause errors

```
(:= (action ... args ...))
... more statements ...
 --->
(mv-let ([err] [val] streamname)
        (action ... args ...)
        (if [err]
            (mv [err] [val] streamname)
          (check-not-free ([err] [val])
             ... more statements ...)))
```

SEQ language: simple binds

Simple Binds can update the stream, cause errors, and bind a name

```
(foo := (action ... args ...))
... more statements ...
 --->
(mv-let ([err] foo streamname)
        (action ... args ...)
        (if [err]
            (mv [err] foo streamname)
          (check-not-free ([err])
             ... more statements ...)))
```

SEQ language: destructuring binds

Destructuring Binds can update the stream, cause errors, and bind many names

```
((foo . bar) := (action ... args ...))
... more statements ...
 --->
(mv-let ([err] [val] streamname)
        (action ... args ...)
        (if [err]
            (mv [err] [val] streamname)
          (let ((foo (car [val]))
                (bar (cdr [val])))
            (check-not-free ([err] [val])
               ... more statements ...))))
```

SEQ language: when and unless

When/Unless Blocks are useful for matching optional stuff

inoutdecl ::= **inout** *nettype* [**signed**] [*range*] *list_of_port_identifiers*

```
(seq tokens
     (:= (match-token :kwd-inout tokens))
     (type := (parse-net-type tokens))
     (when (is-token :kwd-signed tokens)
       (signed := (match-token :kwd-signed tokens)))
     (when (is-token :lbrack tokens)
       (range := (parse-range tokens)))
     (ids := (parse-list-of-port-ids tokens))
     (return (make-inoutdecl type signed range ids)))
```

SEQ language: early returns

Early Returns are useful for choosing between alternative productions.

$$\textit{nonempty\_list\_of\_ids} ::= \textit{identifier}\ \{\ ,\ \textit{identifier}\ \}$$

```
(defun parse-nonempty-list-of-ids (tokens)
  (seq tokens
       (first := (match-token :identifier tokens))
       (unless (is-token :comma tokens)
         (return (list first)))
       (:= (match-token :comma tokens))
       (rest := (parse-nonempty-list-of-ids tokens))
       (return (cons first rest))))
```

SEQ language: looking ahead

Arbitrary lookahead is trivial when you are traversing a list. For stobjs, you would need some kind of unget operation.

```
(defun parse-nonempty-list-of-ids (tokens)
  (seq tokens
       (first := (match-token :identifier tokens))
       (when (and (is-token :comma tokens)
                  (is-token :identifier (cdr tokens)))
         (:= (match-token :comma tokens))
         (rest := (parse-nonempty-list-of-ids tokens)))
       (return (cons first rest))))
```

SEQ language: backtracking

Backtracking is also relatively straightforward by "trapping" errors.

```
(defun parse-foo-or-bar (tokens)
  (mv-let (erp foo updated-tokens)
          (parse-foo tokens)
          (if (not erp)
              (mv nil foo updated-tokens)
            (parse-bar tokens))))

(defun parse-foo-or-bar (tokens)
  (seq-backtrack tokens
                 (parse-foo tokens)
                 (parse-bar tokens)))
```

Sensible error reporting for primitives

My **match-token** and **is-token** are macros instead of functions.

For error reporting, we can introduce our parsing functions with **defparser** instead of defun.

```
(defparser foo (... tokens)
  body)
 --->
(defun foo (... tokens)
  (let ((__function__ 'foo))
    (declare (ignorable __function__))
    body))
```

```
(defmacro is-token (type &optional (tokens 'tokens))
  (declare (xargs :guard (tokentypep type)))
  `(and (consp ,tokens)
        (eq (token-type (car ,tokens)) ,type)))

(defmacro match-token (type &optional (tokens 'tokens))
  (declare (xargs :guard (tokentypep type)))
  (let ((tokens ,tokens))
       (if (not (consp tokens))
           (mv [[error in __function__, unexpected eof ]]
               nil tokens)
         (let ((token1 (car tokens)))
           (if (not (eq ,type (token-type token1)))
               (mv [[error in __function__ at [place] ...]]
                   nil tokens)
               (mv nil token1 (cdr tokens)))))))
```

Efficiency note

Creating error strings was really slow.

Consing together error structures instead reduced parser time by 80% even though there isn't that much backtracking.

```
(list "foo is ~x0 and bar is ~x1.~%" foo bar)
  vs.
(concatenate 'string "foo is "
             (coerce (explode-atom foo 10) 'string)
             "and bar is "
             (symbol-name bar)
             ".~%")
```

More syntactic sugar

Defparser generates a macro alias with implicit tokens
And automates the tokenlistp guard

```
(defparser parse-range (tokens)
  (seq tokens
       (:= (match-token :LBRACK))
       (e1 := (parse-expression))
       (:= (match-token :COLON))
       (e2 := (parse-expression))
       (:= (match-token :RBRACK))
       (return (make-range e1 e2))))

(defparser parse-foo (tokens)
  (seq tokens
       (range := (parse-range))
       ...))
```

Logic-mode parsing

With 170 defparser functions, we need to automate theorem creation.

We classify our parsers as having various properties, and use these properties to decide what theorems to prove about them.

Sometimes, combinations of properties can lead to better theorems.

Net effect: logic mode and guard verification are fairly easy.

Termination behavior

Every defparser we've written is at least **weakly decreasing**:

- `(<= (acl2-count (third (parse-foo)))`
  `(acl2-count tokens))`

Most are also **strongly decreasing**:

- `(implies (not (first (parse-foo)))`
  `(< (acl2-count (third (parse-foo)))`
  `(acl2-count tokens)))`

While some others are only **strong on value**:

- `(implies (second (parse-foo))`
  `(< (acl2-count (third (parse-foo)))`
  `(acl2-count tokens)))`

Failure behavior

Most (all?) of our parsers fail **gracefully**:

- `(implies (first (parse-foo))`
  `(not (second (parse-foo))))`

But some **never** fail: (e.g., optional or zero+ productions)

- `(not (first (parse-foo)))`

Result characterization

On success, we usually have some idea what the **result** ought to be:

- `(implies (and (not (first (parse-foo)))`
  `...guards...)`
  `(foop (second (parse-foo))))`

We've also found it useful to know if the result is a **true-listp**.

- `(true-listp (second (parse-foo)))`

It's also useful to know if **resultp-of-nil** is true.

- If `(foop nil)` and fails gracefully, omit first hyp.
- If never fails, omit first hyp.

Actual examples (verbatim)

```
(defparser vl-parse-range (tokens)
  :result (vl-range-p val)
  :resultp-of-nil nil
  :fails gracefully
  :count strong
  (seq tokens
       (:= (vl-match-token :vl-lbrack))
       (left := (vl-parse-expression))
       (:= (vl-match-token :vl-colon))
       (right := (vl-parse-expression))
       (:= (vl-match-token :vl-rbrack))
       (return (vl-range left right))))
```

Actual examples (verbatim)

```
(defparser vl-parse-optional-drive-strength (tokens)
  :result (vl-maybe-gatestrength-p val)
  :resultp-of-nil t
  :fails never
  :count strong-on-value
  (mv-let (erp val explore)
          (vl-parse-drive-strength)
          (if erp
              (mv nil nil tokens)
            (mv nil val explore))))
```

Actual examples (verbatim)

```
(defparser vl-parse-list-of-param-assignments (tokens)
  :result (vl-param-assignment-tuple-list-p val)
  :resultp-of-nil t
  :true-listp t
  :fails gracefully
  :count strong
  (seq tokens
       (first := (vl-parse-param-assignment))
       (when (vl-is-token? :vl-comma)
         (:= (vl-match-token :vl-comma))
         (rest := (vl-parse-list-of-param-assignments)))
       (return (cons first rest))))
```

Mutual recursion termination problems

We can't expand definitions of mutually-recursive functions until they've been admitted, so this doesn't work:

```
(vl-mutual-recursion
 (defparser vl-parse-lvalue (tokens)
   ...)

 (defparser vl-parse-list-of-lvalues (tokens)
   (declare ...)
   (seq tokens
        (first := (vl-parse-lvalue))
        ...
        (rest := (vl-parse-list-of-lvalues tokens))
        ...))
```

We hackishly extend SEQ with :s= and :w= to address this.

The parser

vl-parse : tokenlist $\rightarrow$ successp $\times$ modulelist

- Entirely logic mode, guards verified
- 7,800 lines (more than half comments/whitespace)
- 1,400 lines of unit tests (want more)
- Almost $\frac{1}{3}$ deals with expressions and statements (mutual recursion)

Similar to Terry's original parser (Common Lisp)

- Loop is about the only thing seq doesn't handle well
- Backtracking is quite nice

Parse trees

We introduce parse trees in a separate file (parsetree.lisp) which has few dependencies.

Mostly we just introducing various aggregates, for instance:

```
(defaggregate regdecl
 (name signedp range arrdims initval atts)
 :tag :regdecl
 :require
 ((stringp-of-regdecl->name          (stringp name))
  (booleanp-of-regdecl->signedp      (booleanp signedp))
  (maybe-range-p-of-regdecl->range   (maybe-range-p range))
  (rangelist-p-of-regdecl->arrdims   (rangelist-p arrdims))
  (maybe-expr-p-of-regdecl->initval (maybe-expr-p initval))
  (atts-p-of-regdecl->atts           (atts-p atts))))
```

Parse tree implementation

Defaggregate and deflist get me pretty far, but they don't do everything.

- Not useful in mutually-recursive cases
- I need to write a good sum-of-products macro

For now, some custom-written recognizers, constructors, and accessors to handle these cases — ugh!

2,100 lines (35% ws/comments), all logic-mode, guards-verified, lots of basic theorems (mostly automatic)

Used by our translation process

- Completeness, non-conflicting names, reasonable constructs
- Unparameterization, expression simplification, e-ification

Demo