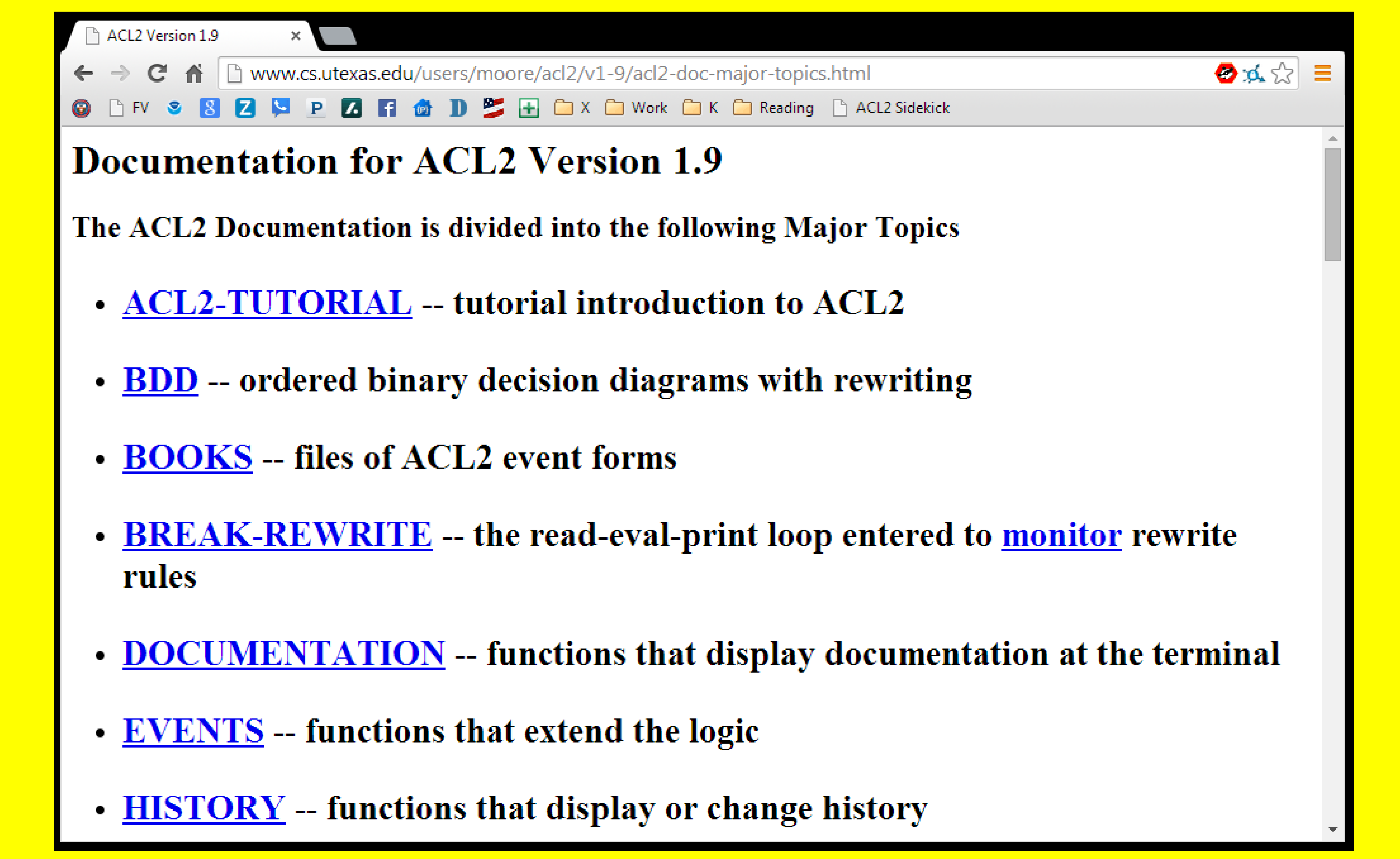
Industrial Strength Documentation for ACL2

Jared Davis
jared@centtech.com

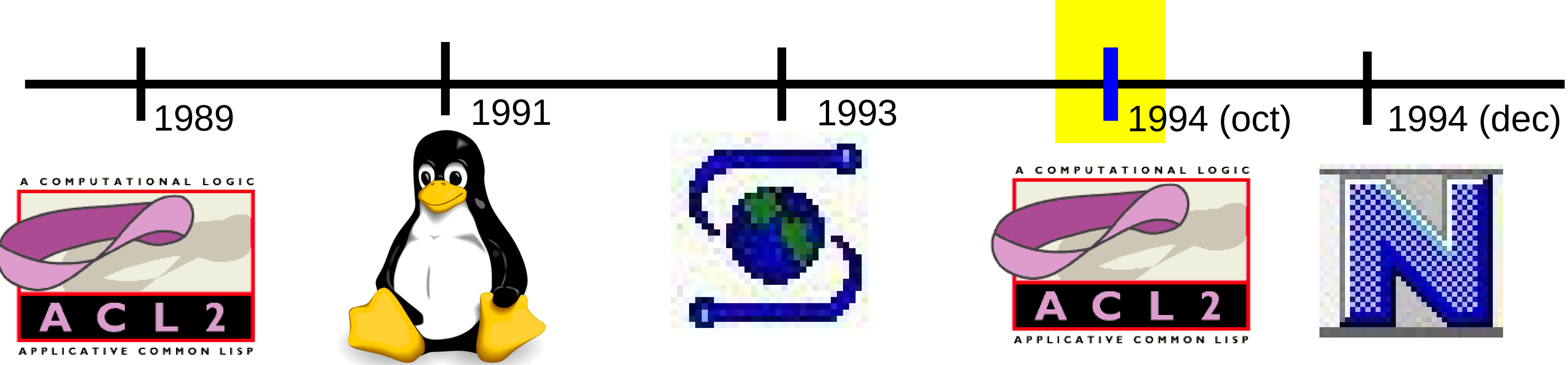
Matt Kaufmann
kaufmann@cs.utexas.edu

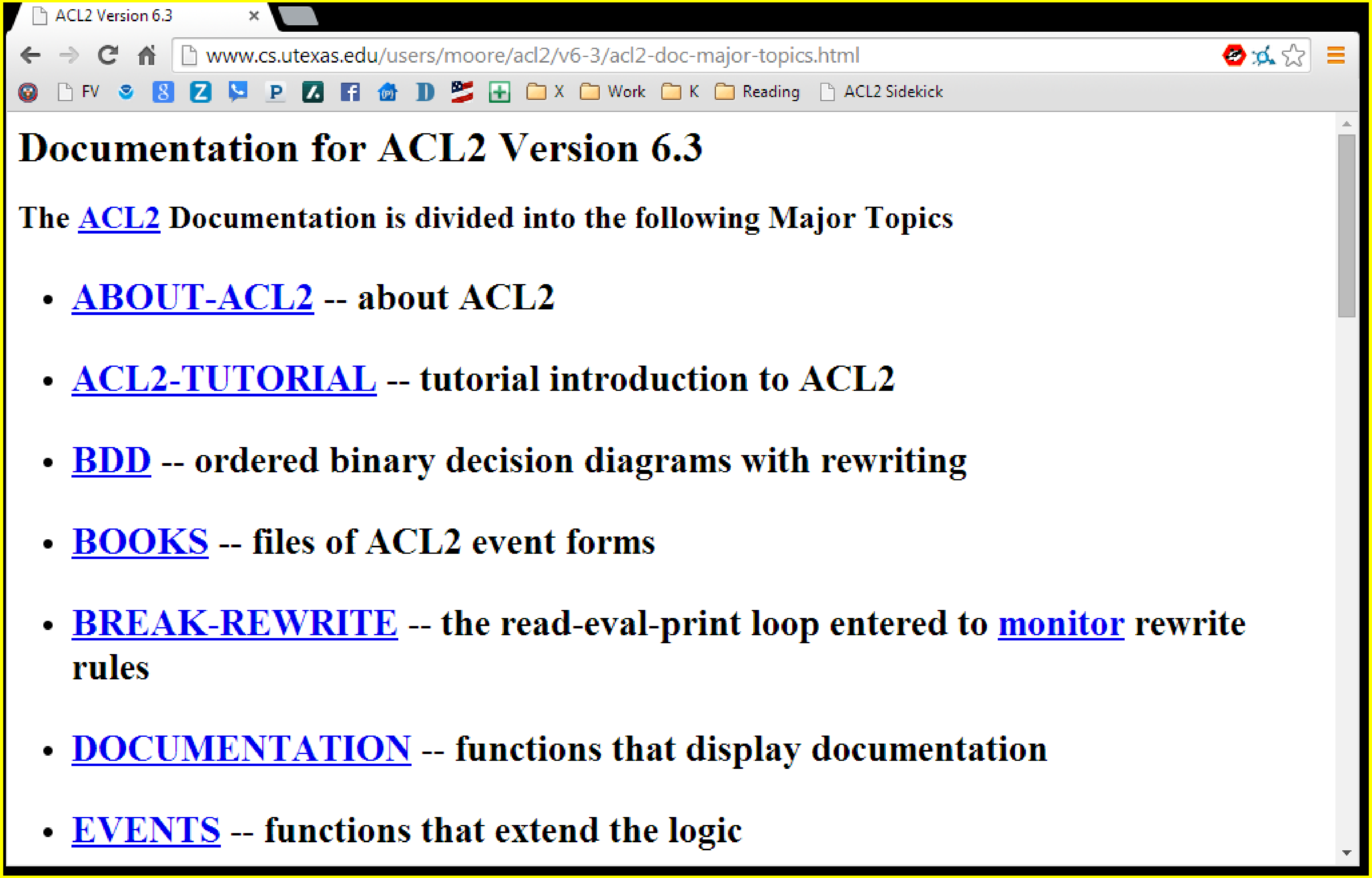

ACL2 Version 1.9
www.cs.utexas.edu/users/moore/acl2/v1-9/acl2-doc-major-topics.html
FV
X
Work
K
Reading
ACL2 Sidekick

Documentation for ACL2 Version 1.9

The ACL2 Documentation is divided into the following Major Topics

- ACL2-TUTORIAL -- tutorial introduction to ACL2
- BDD -- ordered binary decision diagrams with rewriting
- BOOKS -- files of ACL2 event forms
- BREAK-REWRITE -- the read-eval-print loop entered to monitor rewrite rules
- DOCUMENTATION -- functions that display documentation at the terminal
- EVENTS -- functions that extend the logic
- HISTORY -- functions that display or change history

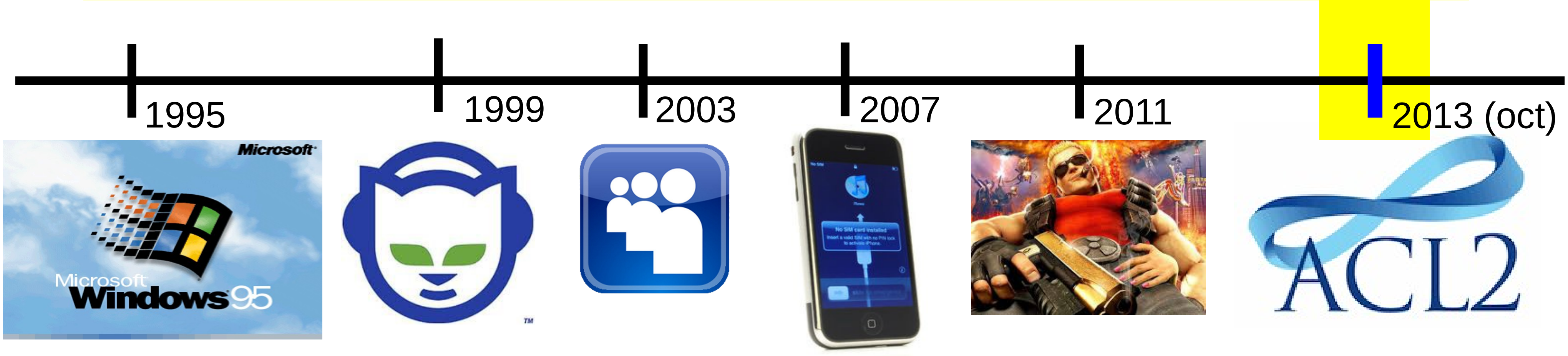


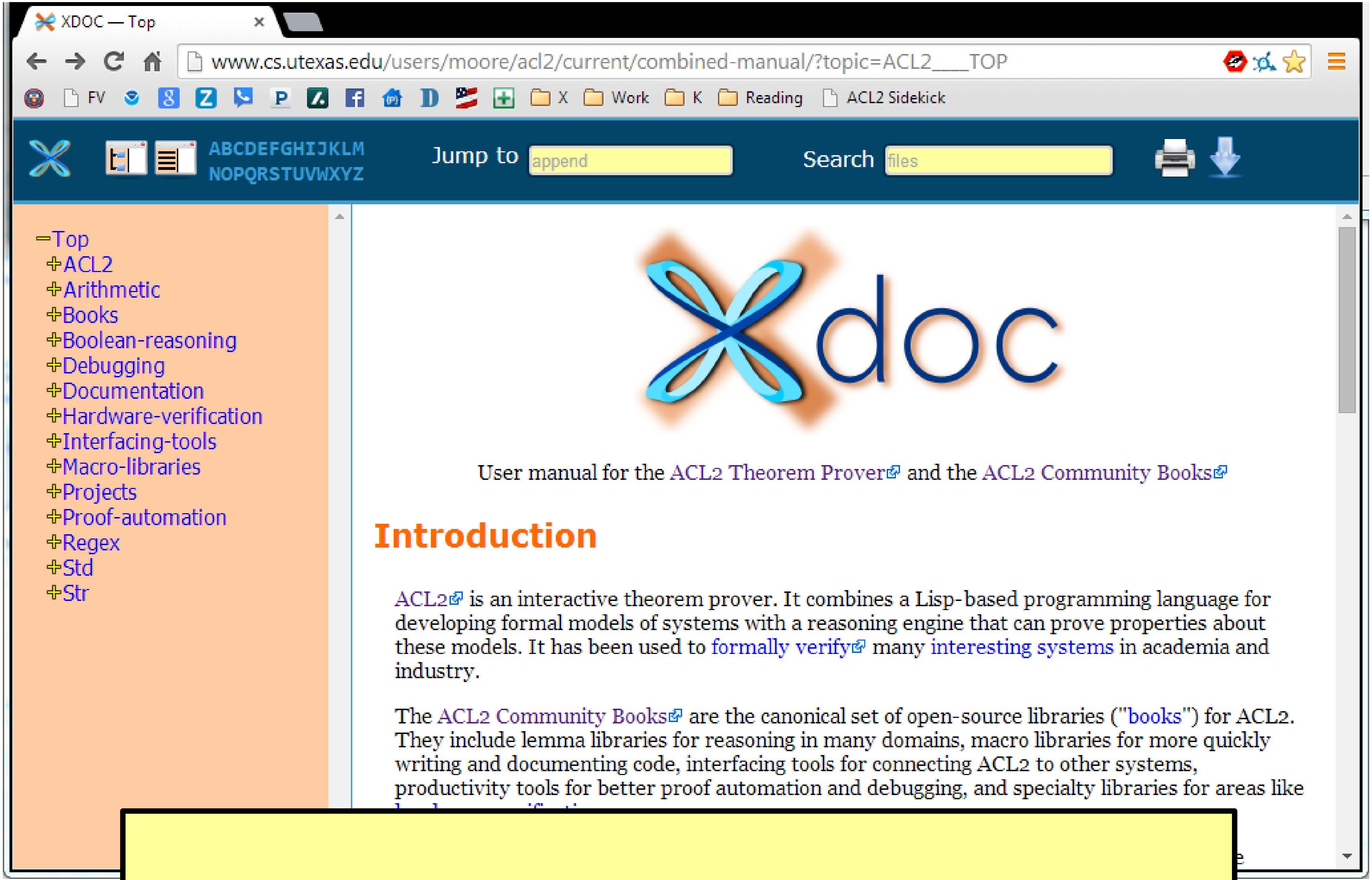

ACL2 Version 6.3
www.cs.utexas.edu/users/moore/acl2/v6-3/acl2-doc-major-topics.html
FV
X
Work
K
Reading
ACL2 Sidekick

Documentation for ACL2 Version 6.3

The ACL2 Documentation is divided into the following Major Topics

- ABOUT-ACL2 -- about ACL2
- ACL2-TUTORIAL -- tutorial introduction to ACL2
- BDD -- ordered binary decision diagrams with rewriting
- BOOKS -- files of ACL2 event forms
- BREAK-REWRITE -- the read-eval-print loop entered to monitor rewrite rules
- DOCUMENTATION -- functions that display documentation
- EVENTS -- functions that extend the logic





User manual for the ACL2 Theorem Prover and the ACL2 Community Books

Introduction

ACL2 is an interactive theorem prover. It combines a Lisp-based programming language for developing formal models of systems with a reasoning engine that can prove properties about these models. It has been used to formally verify many interesting systems in academia and industry.

The ACL2 Community Books are the canonical set of open-source libraries ("books") for ACL2. They include lemma libraries for reasoning in many domains, macro libraries for more quickly writing and documenting code, interfacing tools for connecting ACL2 to other systems, productivity tools for better proof automation and debugging, and specialty libraries for areas like

Demo

How to document your books

(the tedious, manual way, for starters)

```
(include-book “xdoc/top” :dir :system)

(defxdoc str
  :short "ACL2 String Library"
  :long "<p>This is a rudimentary string library
for ACL2.</p>

<p>The functions here are all in logic mode, with
verified guards.  In many cases, some effort has
been spent to make them both efficient and relative
straightforward to reason about.</p>

<p>Ordinarily, t                                    p
@({
 (include-book \
})

<p>The documentation is then available by typing
```

Docs in Code

```
(include-book “xdoc/top” :dir :system)
```

```
(defxdoc str
  :short "ACL2 String Library"
  :long "<p>This is a rudimentary string library
for ACL2.

<p>The fu                                            with
verified                                             has
been spen                                          elative
straightf

<p>Ordinarily, to use the library one should run</p
@({
 (include-book \"str/top\" :dir :system)
})

<p>The documentation is then available by typing
```

Lightweight

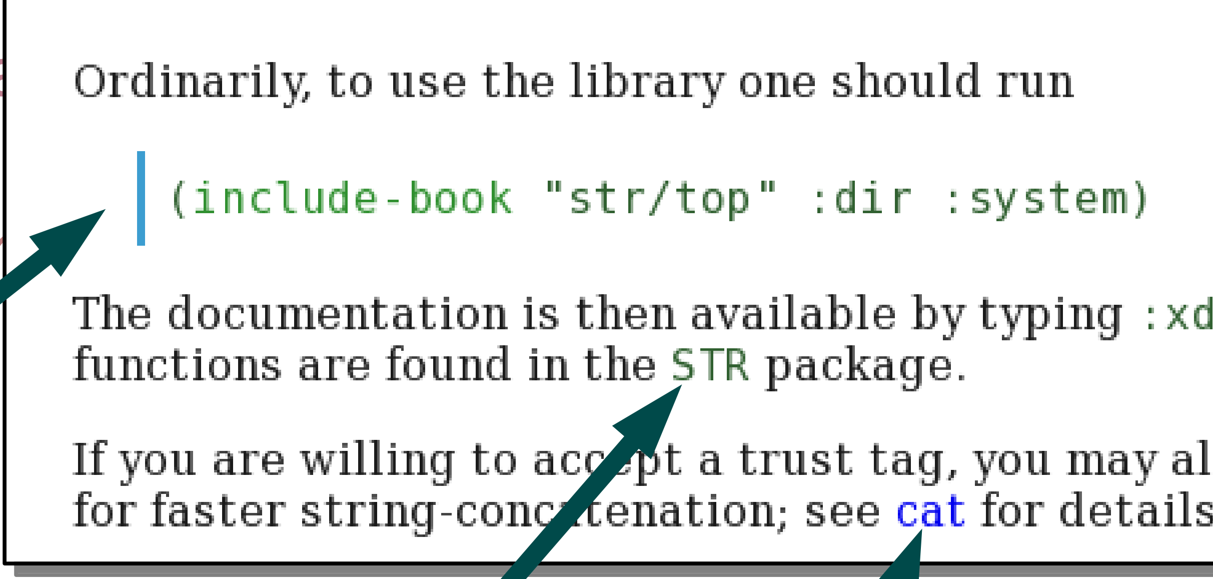
Loads Quickly (< 0.1 sec)

Standard XML Syntax

Tags must be balanced!

```
(include

(defxdoc
  :short
  :long                                          library
for ACL2

<p>The functions here are all in logic mode, with
verified guards.  In many cases, some effort has
been spent to make them both efficient and relative
straightforward to reason about.</p>

<p>Ordinarily, to use the library one should run</p
@({
 (include-book \"str/top\" :dir :system)
})
```

```
straightforward to rea

<h3>Loading the librar

<p>Ordinarily, to use

@({
 (include-book \"str/top\" :dir :system)
})

<p>The documentation is then available by typing
@(':xdoc str').  All of the library's functions
are found in the @('STR') package.</p>

<                    to accept a trust tag, you
m                    @('fast-cat') book for faste
s                    see @(see cat) for
```



Ordinarily, to use the library one should run

```
(include-book "str/top" :dir :system)
```

The documentation is then available by typing :xd
functions are found in the STR package.

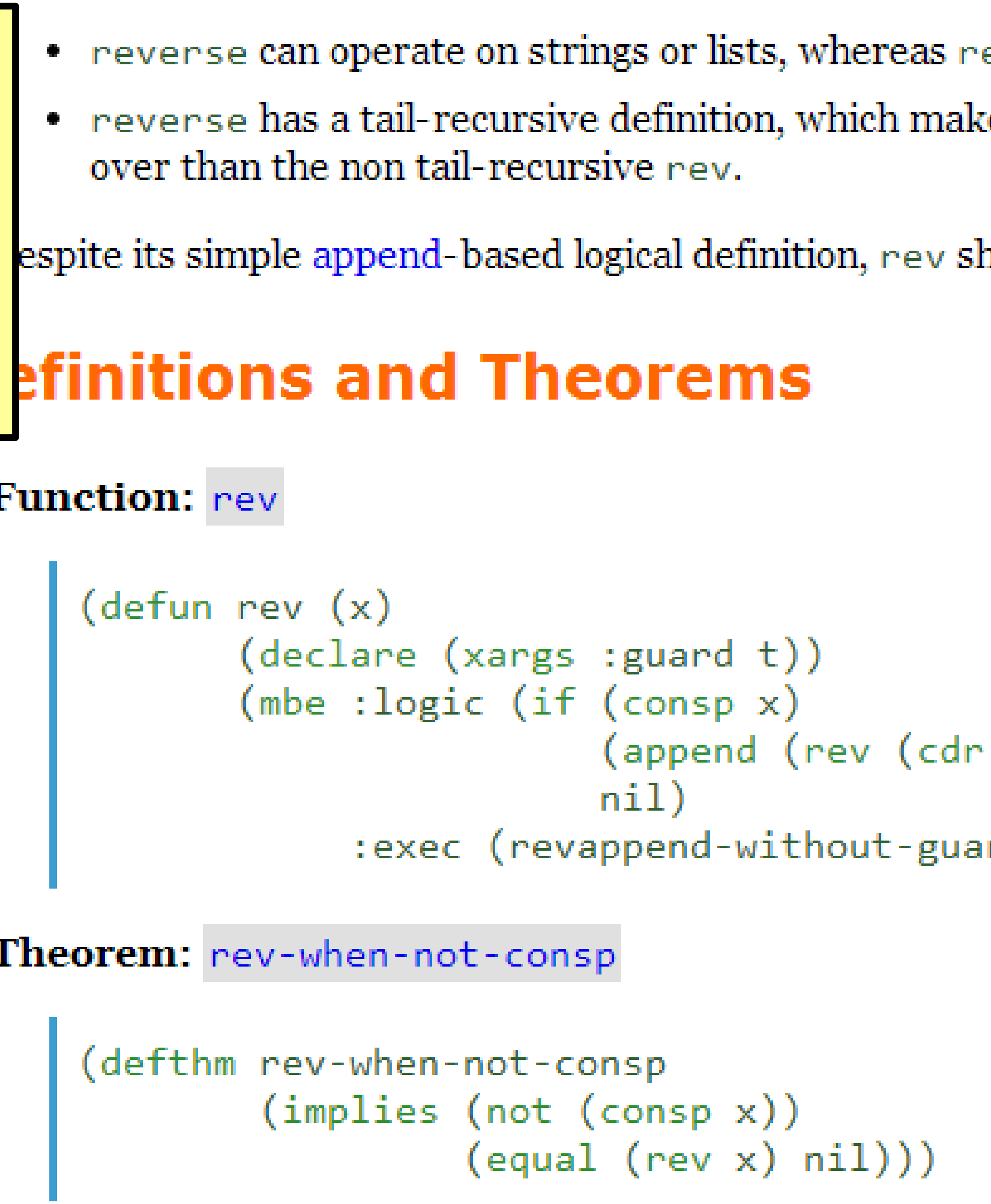
If you are willing to accept a trust tag, you may al
for faster string-concatenation; see cat for details

Preprocessor!

Fights Bitrot!

Auto Links!

@(def rev)



- reverse can operate on strings or lists, whereas re
- reverse has a tail-recursive definition, which mak over than the non tail-recursive rev.

espite its simple append-based logical definition, rev sh

efinitions and Theorems

Function: rev

```
(defun rev (x)
       (declare (xargs :guard t))
       (mbe :logic (if (consp x)
                       (append (rev (cdr
                       nil)
            :exec (revappend-without-guar
```

Theorem: rev-when-not-consp

```
(defthm rev-when-not-consp
        (implies (not (consp x))
                 (equal (rev x) nil)))
```

@(def rev-when-not-consp)

Xdoc

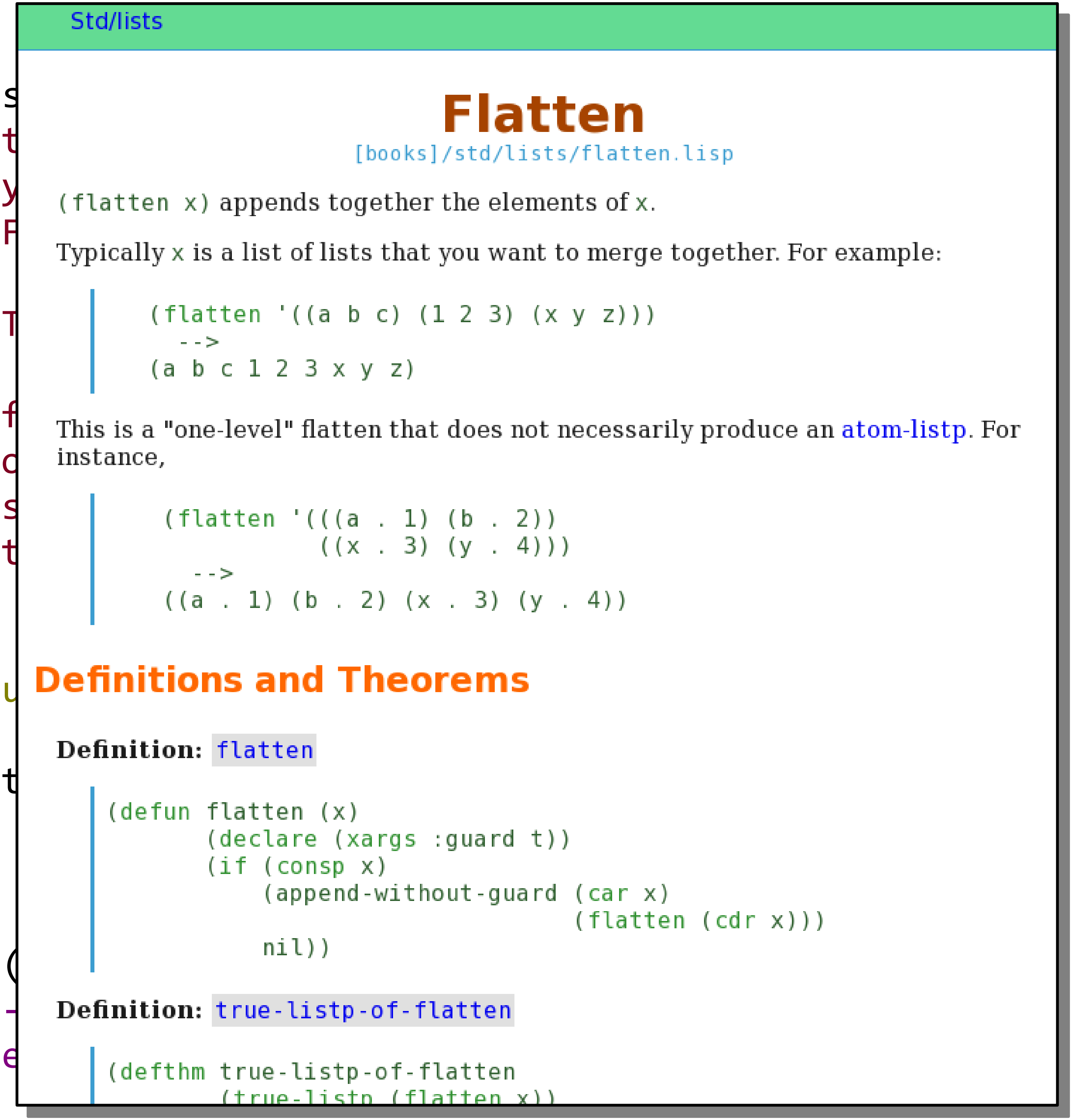
How to document your books

organize and

(the fancy, less tedious way)

```
(defxdoc flatten
  :parents (std/lists)
  :short "@(call flatten) appends together the elements of @('x').”
  :long "<p>Typically @('x') is a list of lists that you want
To merge together.  For example:</p>
...
<h3>Definitions and Theorems</h3>
@(def flatten)
@(thm true-listp-of-flatten)
@(thm flatten-when-not-consp)
@(thm flatten-of-cons)
@(thm flatten-of-list-fix) ...”)

(defund flatten (x)
  (declare (xargs :guard t))
  (if (consp x)
      (append-without-guard (car x) (flatten (cdr x)))
    nil))

(encapsulate ()
  (local (in-theory (enable flatten)))
  (defthm true-listp-of-flatten ...)
  (defthm flatten-when-not-consp ...)
  ...))
```

```
(defxdoc flatten
  :parents (std/lists
  :short "@(call flat
  :long "<p>Typically
To merge together.  F
...
<h3>Definitions and T
@(def flatten)
@(thm true-listp-of-f
@(thm flatten-when-no
@(thm flatten-of-cons
@(thm flatten-of-list

(defund flatten (x)
  (declare (xargs :gu
  (if (consp x)
      (append-without
    nil))

(encapsulate ()
  (local (in-theory (
  (defthm true-listp-
  (defthm flatten-whe
  ...))
```



Std/lists

Flatten

[books]/std/lists/flatten.lisp

(flatten x) appends together the elements of x.

Typically x is a list of lists that you want to merge together. For example:

```
(flatten '((a b c) (1 2 3) (x y z)))
  -->
(a b c 1 2 3 x y z)
```

This is a "one-level" flatten that does not necessarily produce an atom-listp. For instance,

```
 (flatten '(((a . 1) (b . 2))
            ((x . 3) (y . 4)))
  -->
 ((a . 1) (b . 2) (x . 3) (y . 4))
```

Definitions and Theorems

Definition: flatten

```
(defun flatten (x)
       (declare (xargs :guard t))
       (if (consp x)
           (append-without-guard (car x)
                                 (flatten (cdr x)))
           nil))
```

Definition: true-listp-of-flatten

```
(defthm true-listp-of-flatten
        (true-listp (flatten x))
(defxdoc flatten
  :parents (std/lists)
  :short "@(call flatten) appends together the elements of @('x').”
  :long "<p>Typically @('x') is a list of lists that you want
To merge together.  For example:</p>
...
<h3>Definitions and Theorems</h3>
@(def flatten)
@(thm true-listp-of-flatten)
@(thm flatten-when-not-consp)
@(thm flatten-of-cons)
@(thm flatten-of-list-fix) ...”)

(defund flatten (x)
  (declare (xargs :guard t))
  (if (consp x)
      (append-without-guard (car x) (flatten (cdr x)))
    nil))

(encapsulate ()
  (local (in-theory (enable flatten)))
  (defthm true-listp-of-flatten ...)
  (defthm flatten-when-not-consp ...)
  ...))
```

un-DRY!

```
(defsection flatten
  :parents (std/lists)
  :short "@(call flatten) append                                ')
  :long "<p>Typically @('x') is
To merge together.  For example:
[example1]
[example2]”

  (defund flatten (x)
    (declare (xargs :guard t))
    (if (consp x)
        (append-without-guard (car x) (flatten (cdr x)))
      nil))

  (local (in-theory (enab
  (defthm true-listp-of-
  (defthm flatten-when-nc
  ...)
```

DRYer
Organizes books
Better :pbt
Indents nicely

```
<h3>Definitions and The   ms</h3>
@(def flatten)
@(thm true-listp-of-f  tten)
@(thm flatten-when-no  consp)
@(thm flatten-of-cons )
@(thm flatten-of-list-f   )
```

Xdoc

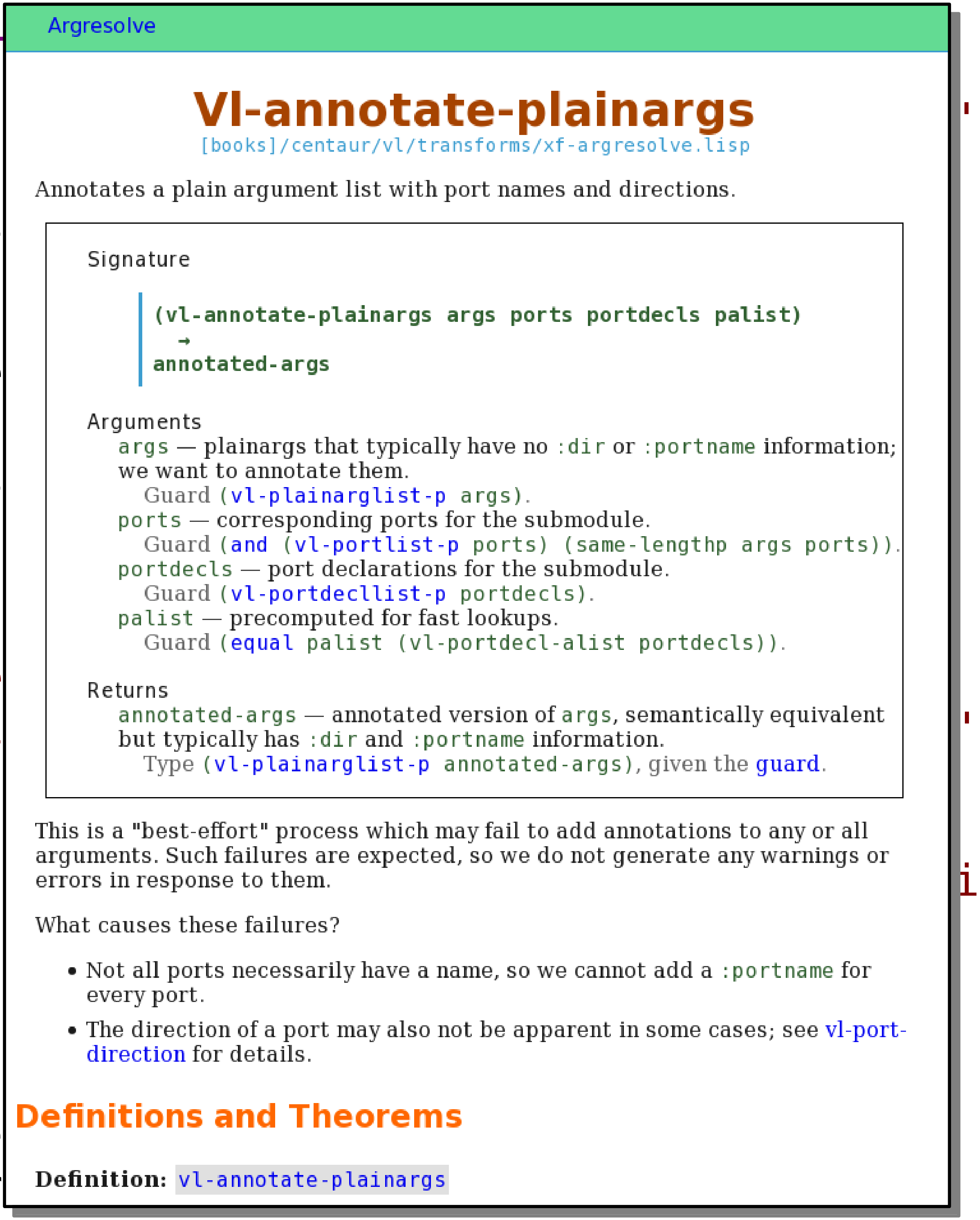
How to organize and document your books

even better

(with less typing and stuff)

```
(define vl-annotate-plainargs
  ((args         "plainargs that typically have no @(':dir') or @('
                  information; we want to annotate them."
                 vl-plainarglist-p)
   (ports        "corresponding ports for the submodule"
                 (and (vl-portlist-p ports)
                      (same-lengthp args ports)))
   (portdecls "port declarations for the submodule"
                 vl-portdecllist-p)
   (palist       "precomputed for fast lookups"
                 (equal palist (vl-portdecl-alist portdecls))))
  :returns
  (annotated-args "annotated version of @('args'), semantically e
                   but typically has @(':dir') and @(':portname')
                  vl-plainarglist-p :hyp :fguard)
  :parents (argresolve)
  :short "Annotates a plain argument list with port names and di
  :long "<p>This is a \"best-effort\" process ...”

  (b* (((when (atom args))
        nil)
       (name (vl-port->name (car ports)))
       (expr (vl-port->expr (car ports)))
    ...)
```

```
(define vl-annotate-p
  ((args       "plainargs th
                information;
               vl-plainargli
   (ports      "correspondin
               (and (vl-port
                    (same-le
   (portdecls "port declara
               vl-portdeclli
   (palist     "precomputed
               (equal palist
  :returns
  (annotated-args "annotate
                   but typi
                  vl-plaina
  :parents (argresolve)
  :short "Annotates a plain
  :long "<p>This is a \"bes

  (b* (((when (atom args))
        nil)
       (name (vl-port->name
       (expr (vl-port->expr
    ...)
```



Argresolve

Vl-annotate-plainargs

[books]/centaur/vl/transforms/xf-argresolve.lisp

Annotates a plain argument list with port names and directions.

Signature

```
(vl-annotate-plainargs args ports portdecls palist)
  →
annotated-args
```

Arguments

args — plainargs that typically have no :dir or :portname information; we want to annotate them.
Guard (vl-plainarglist-p args).

ports — corresponding ports for the submodule.
Guard (and (vl-portlist-p ports) (same-lengthp args ports)).

portdecls — port declarations for the submodule.
Guard (vl-portdecllist-p portdecls).

palist — precomputed for fast lookups.
Guard (equal palist (vl-portdecl-alist portdecls)).

Returns

annotated-args — annotated version of args, semantically equivalent but typically has :dir and :portname information.
Type (vl-plainarglist-p annotated-args), given the guard.

This is a "best-effort" process which may fail to add annotations to any or all arguments. Such failures are expected, so we do not generate any warnings or errors in response to them.

What causes these failures?

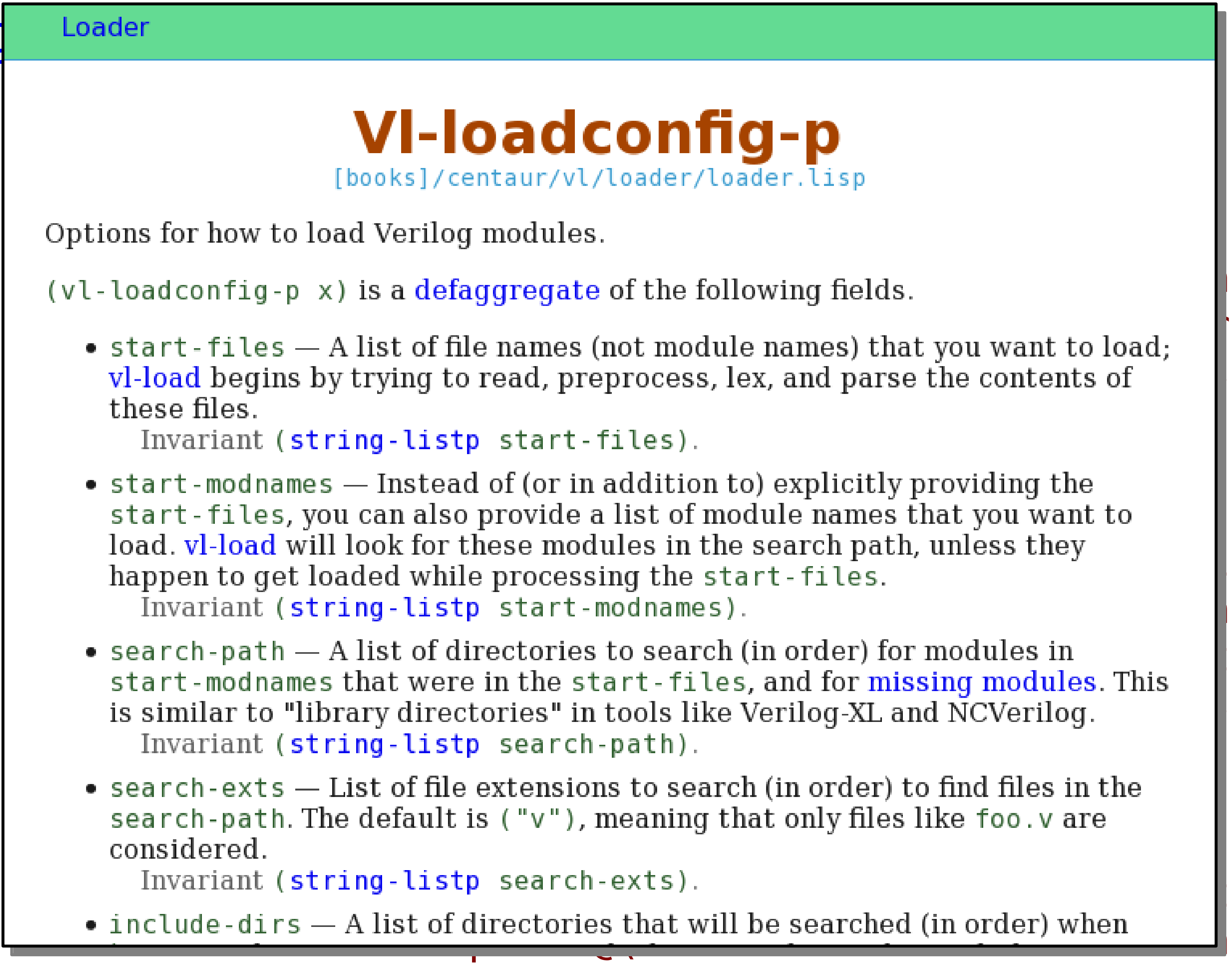
- Not all ports necessarily have a name, so we cannot add a :portname for every port.
- The direction of a port may also not be apparent in some cases; see vl-port-direction for details.

Definitions and Theorems

Definition: vl-annotate-plainargs

```
(defaggregate vl-loadconfig
  :parents (loader)
  :short "Options for how to load Verilog modules."

  ((start-files    string-listp
                   "A list of file names (not module names) that
                    load; @(see vl-load) begins by trying to read
                    lex, and parse the contents of these files.")

   (start-modnames string-listp
                   "Instead of (or in addition to) explicitly pro
                    @('start-files'), you can also provide a list
                    names that you want to load.  @(see vl-load)
                    these modules in the search path, unless they
                    loaded while processing the @('start-files').

   (search-path    string-listp
                   "A list of directories to search (in order) fo
                    @('start-modnames') that were in the @('start
                    for <see topic='@(url vl-modulelist-missing)'
                    modules</see>.  This is similar to \"library
                    in tools like Verilog-XL and NCVerilog.")
   ...)
```

```
(defaggreg
  :parents (
  :short "Op

  ((start-fi                                                      at
                                                                  ead
                                                                  .")


    (start-mo                                                     pro
                                                                  ist
                                                                  d)
                                                                  hey
                                                                  ')


    (search-p                                                      fo
                                                                  art
                                                                  g)
                   modules</see>.  This is similar to \"library
                   in tools like Verilog-XL and NCVerilog.")

    ...)
```



Loader

Vl-loadconfig-p

[books]/centaur/vl/loader/loader.lisp

Options for how to load Verilog modules.

`(vl-loadconfig-p x)` is a defaggregate of the following fields.

- `start-files` — A list of file names (not module names) that you want to load; vl-load begins by trying to read, preprocess, lex, and parse the contents of these files.
  Invariant `(string-listp start-files)`.
- `start-modnames` — Instead of (or in addition to) explicitly providing the `start-files`, you can also provide a list of module names that you want to load. vl-load will look for these modules in the search path, unless they happen to get loaded while processing the `start-files`.
  Invariant `(string-listp start-modnames)`.
- `search-path` — A list of directories to search (in order) for modules in `start-modnames` that were in the `start-files`, and for missing modules. This is similar to "library directories" in tools like Verilog-XL and NCVerilog.
  Invariant `(string-listp search-path)`.
- `search-exts` — List of file extensions to search (in order) to find files in the `search-path`. The default is `("v")`, meaning that only files like `foo.v` are considered.
  Invariant `(string-listp search-exts)`.
- `include-dirs` — A list of directories that will be searched (in order) when

Macros like these aren't hard.

Documentation as Data

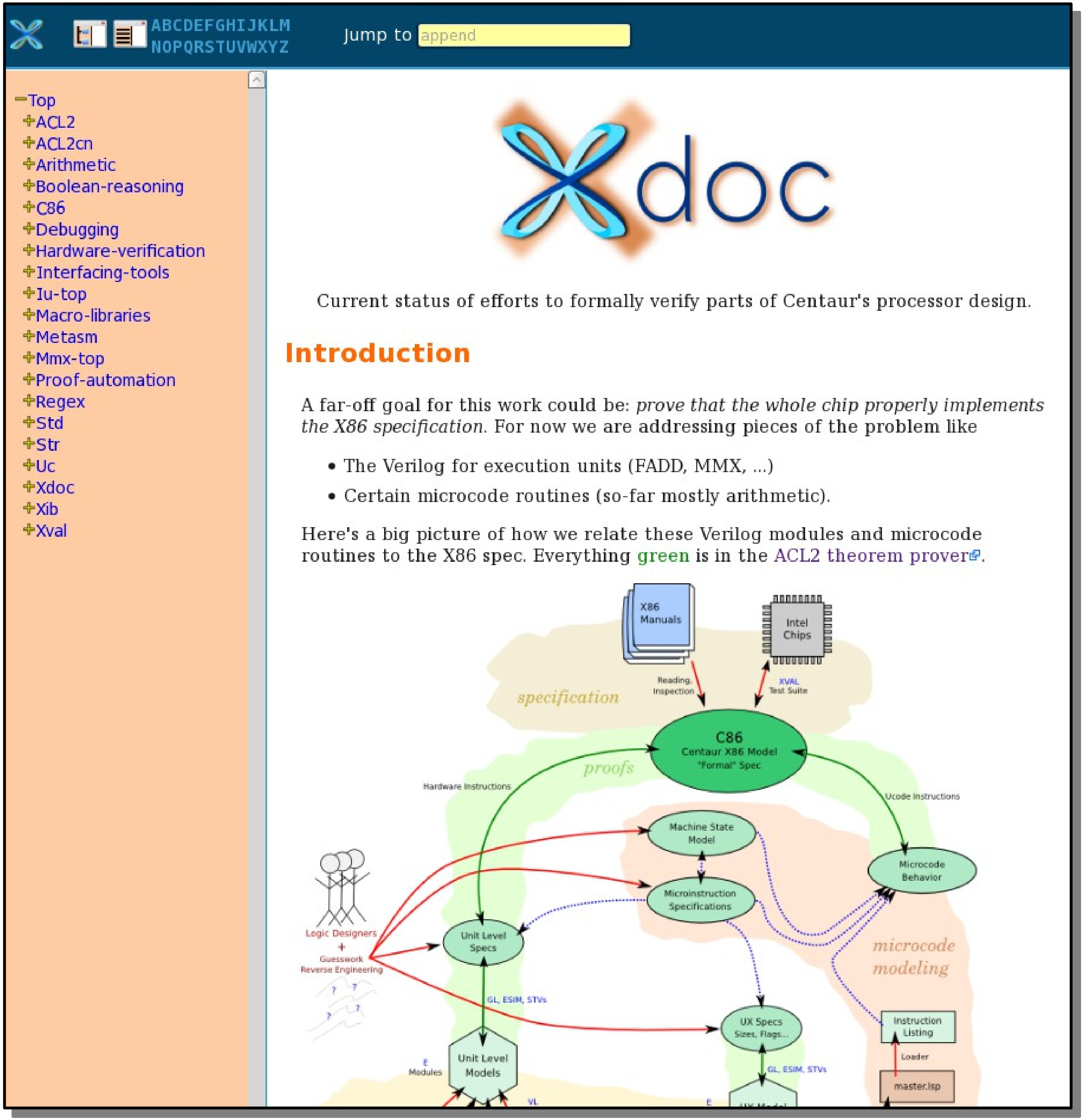
The full docs are just a table with a list of topics.

xdoc

How to get a fancy manual with your stuff in it

```
(include-book “your-books”)
(xdoc::save “./my-manual”)
```

(by the way, it's embeddable)



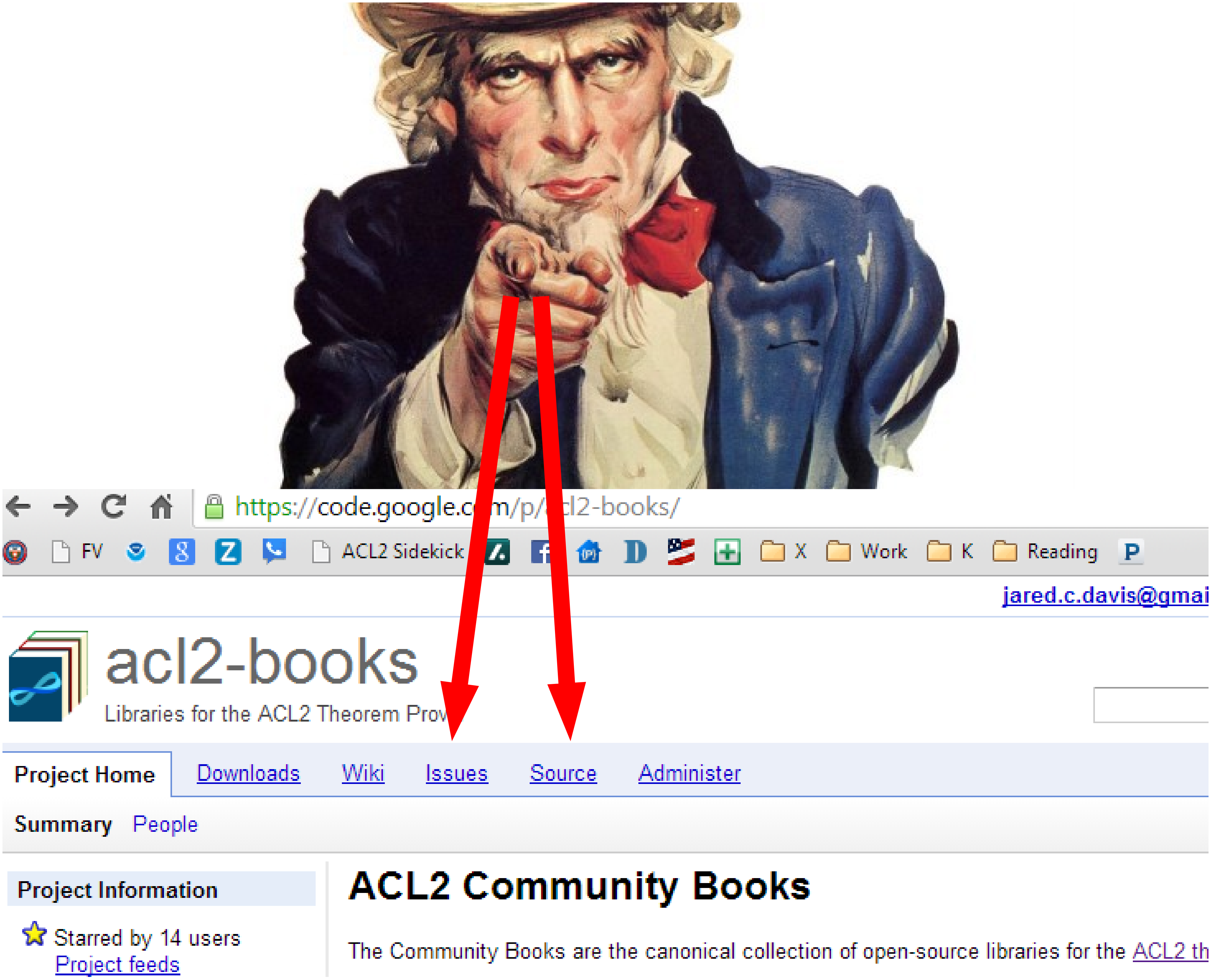
Current status of efforts to formally verify parts of Centaur's processor design.

Introduction

A far-off goal for this work could be: *prove that the whole chip properly implements the X86 specification*. For now we are addressing pieces of the problem like

- The Verilog for execution units (FADD, MMX, ...)
- Certain microcode routines (so-far mostly arithmetic).

Here's a big picture of how we relate these Verilog modules and microcode routines to the X86 spec. Everything green is in the ACL2 theorem prover.

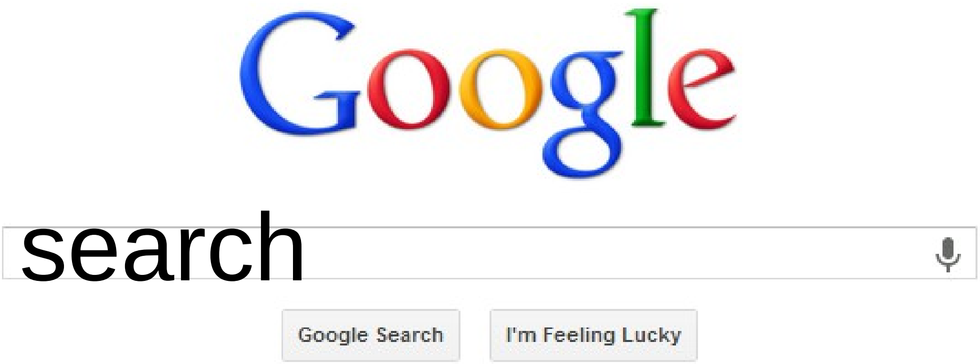


acl2-books

Libraries for the ACL2 Theorem Prov

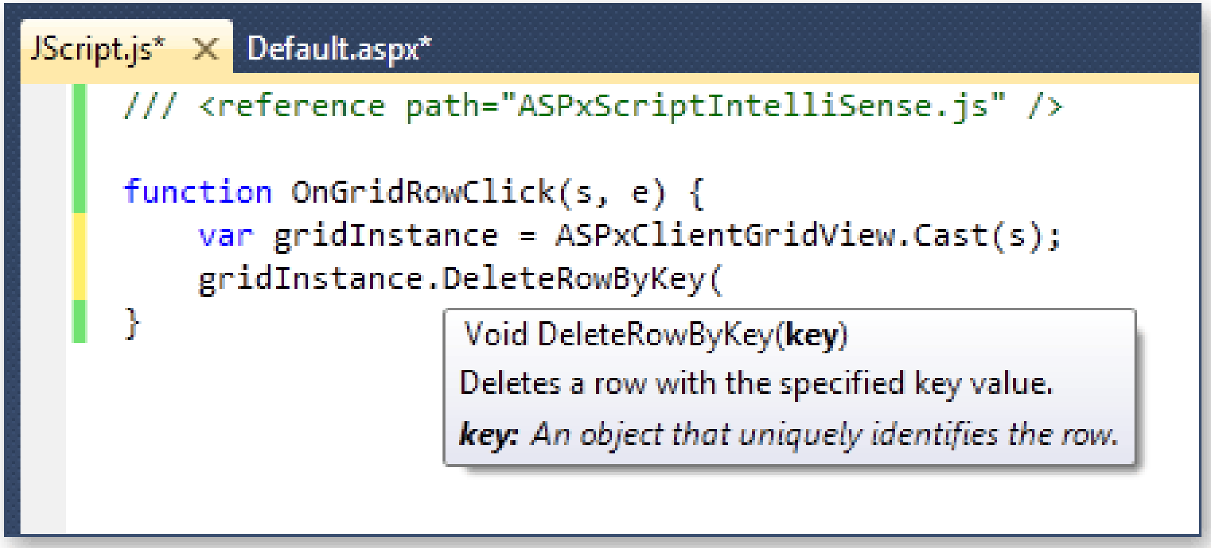
ACL2 Community Books

The Community Books are the canonical collection of open-source libraries for the ACL2 th



search

intellisense



```
/// <reference path="ASPxScriptIntelliSense.js" />

function OnGridRowClick(s, e) {
    var gridInstance = ASPxClientGridView.Cast(s);
    gridInstance.DeleteRowByKey(
}
```

Void DeleteRowByKey(**key**)
Deletes a row with the specified key value.
key: *An object that uniquely identifies the row.*

[edit source]

[add a note]

Thanks!