A Verilog Translator in ACL2

Jared Davis

Centaur Technology

February 18, 2009

Introduction

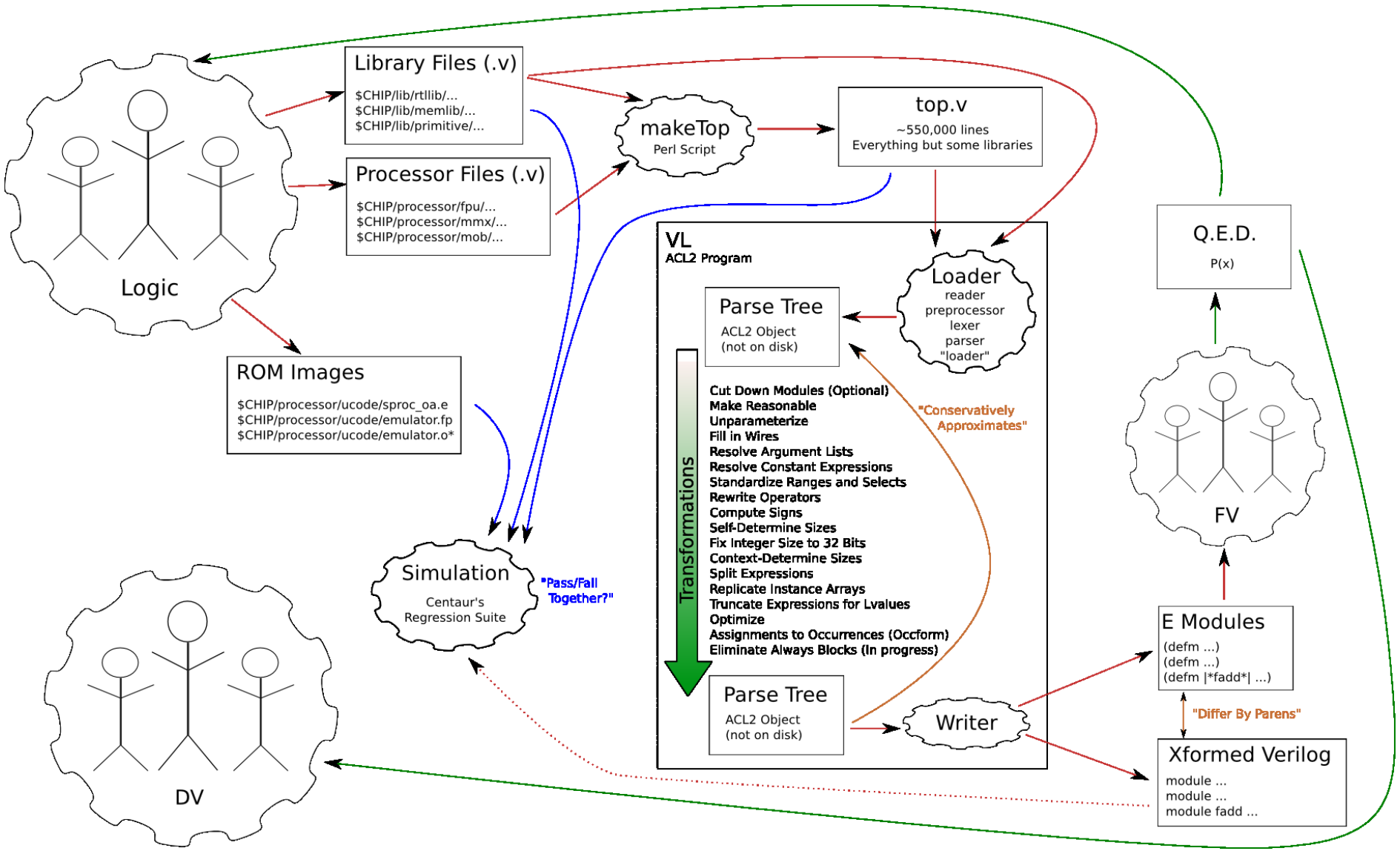
Previously — a preprocessor, lexer, and parser for Verilog 2005, mostly in logic mode with verified guards

- Simplicity over performance (1.5 mins., 10 GB memory)
- Elaborate well-formedness checks, unit testing

Today — a translator to convert the resulting parse tree into E modules

Think Verilog Simplifier + Paren Transposer

- We stay in Verilog as long as possible, rewriting modules into occurrence-based, register-transfer level descriptions
- We want to produce a conservative approximation of the input modules w.r.t. the semantics of Verilog

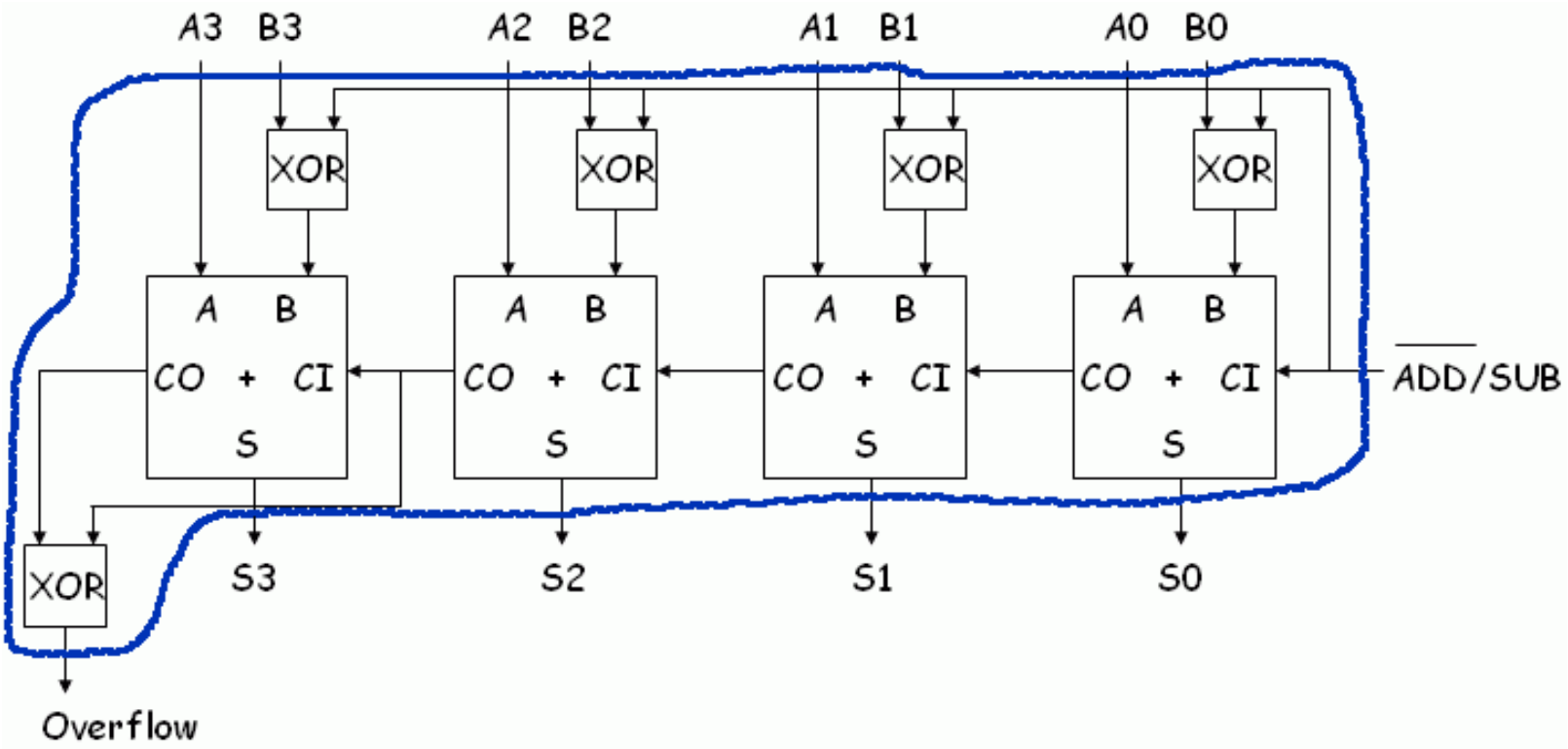

Logic
Library Files (.v)
$CHIP/lib/rtllib/...
$CHIP/lib/memlib/...
$CHIP/lib/primitive/...
Processor Files (.v)
$CHIP/processor/fpu/...
$CHIP/processor/mmx/...
$CHIP/processor/mob/...
makeTop
Perl Script
top.v
~550,000 lines
Everything but some libraries
ROM Images
$CHIP/processor/ucode/sproc_oa.e
$CHIP/processor/ucode/emulator.fp
$CHIP/processor/ucode/emulator.o*
VL
ACL2 Program
Loader
reader
preprocessor
lexer
parser
"loader"
Parse Tree
ACL2 Object
(not on disk)
Transformations
Cut Down Modules (Optional)
Make Reasonable
Unparameterize
Fill in Wires
Resolve Argument Lists
Resolve Constant Expressions
Standardize Ranges and Selects
Rewrite Operators
Compute Signs
Self-Determine Sizes
Fix Integer Size to 32 Bits
Context-Determine Sizes
Split Expressions
Replicate Instance Arrays
Truncate Expressions for Lvalues
Optimize
Assignments to Occurrences (Occform)
Eliminate Always Blocks (In progress)
"Conservatively Approximates"
Parse Tree
ACL2 Object
(not on disk)
Writer
Simulation
Centaur's
Regression Suite
"Pass/Fail Together?"
DV
Q.E.D.
P(x)
FV
E Modules
(defm ...)
(defm ...)
(defm |*fadd*| ...)
"Differ By Parens"
Xformed Verilog
module ...
module ...
module fadd ...

Outline

Verilog semantics — modules

Modules are the basic building blocks of Verilog designs

- They have an interface of input and output ports
- They may contain gates, registers, and instances of other modules



Verilog semantics — bits

We consider only register-transfer level stuff (not transistor-level stuff)

Bits (on wires, in registers) can have four values

- 0 — logical false (low)
- 1 — logical true (high)
- X — an unknown value
- Z — a high-impedance value (undriven)

Gate semantics are described in terms of these values with truth tables

buf

input	output
0	0
1	1
X	X
Z	X

not

input	output
0	1
1	0
X	X
Z	X

Verilog semantics — vectors

A wire/register can have a *range*, making it a vector of bits

- `wire [3:0] w;`

Such vectors can be interpreted in various ways

- Unsigned n-bit integers
- Signed n-bit integers
- "Real" numbers
- Strings, times, and realtimes

We deal almost exclusively with unsigned integers

- 4'b 0011, 4'd 10, 8'h FF, 2'b XX

Verilog semantics — expressions

Expressions can be used to concisely describe collections of gates

- `assign w = ~(a & b) ^ (c + 4'b0010) ;`

A complex set of rules are used to determine how wide each operation is

- Does `c + 4'b0010`, above, produce a carry?

The semantics are described w.r.t. the 4-valued logic

- `a & b` ands its arguments, bitwise, using the truth table for and
- `a + b` produces X if any bit of either argument is X or Z
- `a ? b : c` combines `b` and `c` bit-by-bit when `a` is X or Z

Verilog semantics — simulations

Verilog is simultaneously

- A language for describing circuits, and
- A language for simulating circuits over time

```
module test () ;
  reg a, b;
  wire o;
  and (o, a, b);

  initial
  begin
     a <= 0 ;
     b <= 1 ;
     $display("at time 0, o is %b", o);
     #1
     $display("at time 1, o is %b", o);
  end
endmodule
```

Conservative approximation

Spec: every simplified module M' should conservatively approximate the input module, M

Roughly, and too ambitiously,

- M' should have the same ports and internal state as M
- M' should have an instance of S' whenever M has an instance of S.
- For every port and internal value, M_p, at every time t of every simulation s, M'_p bit-approximates M_p,

Where "bit-approximates" means $M'_p = M_p$ or M'_p is X

The approximation must be close enough to facilitate verification (i.e., all X's is not useful)

The clock assumption

Certain ports are known to be clocks.

We assume there is enough time to update all other signals before any clock changes.

This is a huge assumption that rules out many Verilog simulations

Verilog constructs that break conservativity

Some Verilog constructs are broken w.r.t. conservativity

- `if` treats X/Z as false
- `===` and `!==` treat X,Z as knowns
- user-defined primitives may implement any X/Z behavior they like

We would eventually like to move away from using these constructs.

For now, we don't permit UDP's, and unsoundly

- Replace `if` as the ternary operator, ?:
- Treat `===` and `!==` as `==` and `!=`

Parse Trees

Our parser produces a vl-modulelist-p object, which is a list of vl-module-p's. Each of which has

- Name, ports
- Parameter declarations
- Port, register, variable, event, and wire declarations
- Gate instances (occurrences)
- Submodule instances (occurrences)
- Continuous assignments
- Always and initial statements

Most of these are compound structures. Defaggregate, deflist.

Basic module utilities

We develop a number of utilities for working with modules

- Accessor projections (defprojection)
- Modalists, module lookups
- Modnamespaces, item lookups
- Top-level/missing modules
- Dependent/necessary modules
- Dependency-order sorting
- Pruning modules w.r.t. a keep-list

Fun stuff. Logic mode, various theorems. Lots of MBE. Lots of Osets. Could prove lots more.

Reasonableness

A module is reasonable if it is semantically well-formed and does not contain "weird stuff" we do not handle

- Ports should have names and no complex expressions
- Port declarations should be unsigned, typeless, non-inout
- Compatible ports and port declarations, no duplicates
- Compatible port declaration and wire declarations
- No weird wire/reg types, multidimensional arrays, signed values
- No variables, event declarations
- Only simple gates (no transistors)
- Unique namespace

Most Centaur stuff is reasonable. We can generate reports of unreasonable modules.

Translator stages

The translator is written as a bunch of Verilog source transformations.

- Modulo certain extensions (e.g., size info on exprs)

Preamble.

- Read in the entire chip, as it is on disk (1.5 mins)
- Identify and throw away any portion of the chip which is unreasonable (reporting upon what has been done) ($<$ 1 minute)
- Optionally limit scope to particular modules for better translation speed.

Unparameterization

```
module plus(...) ;
  parameter width = 4 ;
  parameter strength = 10 ;
  wire [width-1:0] w;
  ...;
endmodule
```

Our first pass eliminates parameters (by expanding their uses)

- `plus$width=10$strength=13`
- multi-pass to resolve "width - 1" (very cautious)
- We about double the total number of modules
- We eliminate top-level modules with params left
- Result: parameter-free modules

Safe-mode

Unparameterization, and our later steps, produce new list of modules.

In safe-mode, we perform all kinds of well-formedness checks before and after each stage. After unparameterization,

- Do we still have a valid vl-modulelist-p
- Do the modules have unique names (identify name conflicts)
- Is the module list complete
- Is every module still reasonable
- Are all modules parameter-free (completeness of unparam)

Much like theorems, but no proof burden – just execution time.

Filling in wires

Two kinds of implicit wires

- Port implicit – "`input [3:0] a`" without also "`wire [3:0] a`"
- Other, undeclared names are implicitly one-bit wires (yuck)

Our next pass just adds appropriate wire declarations for all the implicit wires.

We can do all the same well-formedness checks from before.
Should add an "every name is declared" check

Resolving argument lists

The ports of a module are named

```
module adder(out, data_a, data_b);
```

Instances can refer to ports by position or name

```
adder a1(out1, data_a1, data_b1);
adder a2(.out(out2), .data_a(data_a2), .data_b(data_b2));
```

Argument list resolution involves

- Ensuring the actuals are compatible with the formals
- Canonicalize all instances to use the positional style
- Marking each argument as an input or output

Resolving constant expressions

We often have expressions in places we want constants.

- Declarations; `wire [6 - 1 : 0] w;`
- Bit selects; `assign msb = w[6 - 1];`
- Part selects; `assign x = w[6 - 1 : 3];`

We now evaluate these expressions, e.g., to 5.

- Spec is vague w.r.t. widths, signedness, etc.
- We only permit unsized integer literals (32-bit signed)
- We only allow overflow-free +, -, and *

Additional well-formedness checks.

- Ranges resolved (all constant indices)
- Selects in bounds (all constant indices in range)

Shifting ranges

Two ways to represent a six-bit vector:

- `wire [7:2] a; // a[7], ..., a[2]`
- `wire [5:0] a; // a[5], ..., a[0]`

We now shift all ranges over so that their rhs is 0.
We must simultaneously shift bit/part-selects.

WF checks: ranges/selects resolved, selects bound, ranges simple

Operator rewriting

We can make synthesis easier by rewriting away various operators.

`a ? b : c`	→	`(|a) ? b : c`	`a < b`	→	`^(a >= b)`
`a ? z : c`	→	`~(|a) ? c : z`	`a > b`	→	`^(b >= a)`
			`a <= b`	→	`b >= a`
`a && b`	→	`(|a) & (|b)`			
`a || b`	→	`(|a) | (|b)`	`a == b`	→	`&(a ~^ b)`
`!a`	→	`~(|a)`	`a != b`	→	`|(a ^ b)`
`~& (a)`	→	`~( &a )`			
`~| (a)`	→	`~( |a )`			
`~^ (a)`	→	`~( ^a )`			

Soundness — Reading the spec, testing with Cadence

Sign computation

Each expression in our parse tree has a sign field

- `nil`, `:vl-signed`, or `:vl-unsigned`.
- Our parser sets them all to `nil`

Rules for leaves

- Signed constants, e.g., 19, 3'bs 011, ...
- Unsigned constants, e.g., 3'b 011, 4'h A, ...
- Wire/port/register names, taken from declarations
- Strings, reals, etc., are left undecided

Rules for operators

- Selects, concatenates, compares are always unsigned
- Funcalls, syscalls, hierarchial id's, mintypmaxes are left undecided
- "All other operators" unsigned unless all args are signed

Width computation

Each expression in our parse tree also has a width field

- `nil` — not yet decided
- `:vl-not-applicable` — for non-integer expressions (strings, reals)
- `:vl-integer-size` — implementation dependent, 32+ bits
- naturals — fixed-width integers, zero included for multiconcats

This is complicated.
Also, the spec is very poorly written, or I am horribly stupid.

Widths are computed in two stages.

- First, we self-size each expression; bottom-up
- Next, we context-size expressions; top-down

An example

```
assign w = (14 + 3) >> 1;
```

Widths determine the value of `w`.

- 17 in binary is 10001.
- If the addition is 4-bit, `0001 >> 1` = 0
- If the addition is 5-bit, `10001 >> 1` = 1000 = 8.

The answer depends upon the size of `w`.

- If `w` is four or fewer bits, the answer is 0.
- If `w` is five or more bits, the answer is 8.

Computing widths, then, is important even for something as simple as constant folding.

Self-determined sizes

Expression	**Self size**
Unsized constants	"Same as integer"
Sized constants	As given
Wires	As declared
`i [+ - * / % & | ^ ^~ ~^] j`	max{ L(i), L(j) }
`[+ - ~] i`	L(i)
`i [=== !== == != > >= < <=] j`	1 bit
`i [&& ||] j`	1 bit
`[& ~& | ~| ^ ~^ ^~ !] i`	1 bit
`i [>> << ** >>> <<<] j`	L(i)
`i ? j : k`	max { L(j),L(k) }
`{i, ..., j}`	L(i)+...+L(j)
`{i {j, ..., k}}`	i*(L(j)+...+L(k))

Dealing with implementation-dependent sizes

We implement a 32-bit semantics, but

- work with symbolic integer sizes for as long as possible, and
- warn about implementation-defined widths

$$\textsc{MaxW}(a : \mathbb{N}, b : \mathbb{N}) = \max(a, b)$$

$$\textsc{MaxW}(\texttt{intsize}, \texttt{intsize}) = \texttt{intsize}$$

$$\textsc{MaxW}(a : \mathbb{N}, \texttt{intsize}) = \begin{cases} \texttt{intsize} & \text{if } a < 32 \\ \text{warn}, a & \text{otherwise} \end{cases}$$

$$\textsc{MaxW}(\_, \_) = \text{warn}, \texttt{nil}$$

$$\textsc{SumW}(a : \mathbb{N}, b : \mathbb{N}) = a + b$$

$$\textsc{SumW}(\_, \_) = \text{warn}, \texttt{nil}$$

Fixing to 32-bits

Since our self-sizing computation puts symbolic `:vl-integer-size` widths on some expressions, we now fix all of these to be 32 bits.

Additional well-formedness check: all expressions have a natural-numbered width

Context sizing algorithm

$$\textsc{CtxSize}(x, w) : \text{expr} \times \text{posp option} \rightarrow \text{expr}$$

Assumptions.

- x is an expression which has its self-sizes already determined
- All of the self-sizes in x are naturals
- x is *purenat*, "everything is unsigned," so all extensions are zero-extensions.

What is w?

- A positive number, "the size of the context", or
- Nil, meaning there is no context (port arguments??, concats, ...)

Context-determined operands

First, we recursively context-determine any context-determined operands

Operator	**Context-determined?**
`i [+ - * / % & | ^ ^~ ~^] j`	Yes
`[+ - ~] i`	Yes
`i [=== !== == != > >= < <=] j`	W.r.t. each other
`i [&& ||] j`	No
`[& ~& | ~| ^ ~^ ^~ !] i`	No
`i [>> << ** >>> <<<] j`	Only i
`i ? j : k`	Only j and k
`{i, ..., j}`	No
`{i {j, ..., k}}`	No

Claim. $\textsc{CtxSize}(x, w)_{ctxsize} = \max\{\text{nfix}(w), x_{selfsize}\}$

Context sizing algorithm

Claim (repeated). $\textsc{CtxSize}(x, w)_{ctxsize} = \max\{\text{nfix}(w), x_{selfsize}\}$

For constants and wires, context-sizing is zero-extension (since we require everything to be unsigned)

For operators like `a + b`, after `a` and `b` have been context-sized, they have the same width. This width becomes the width for the whole expression.

For operators like concatenation, we just need to zero-extend if we don't have enough bits.

For operators like `a == b`, let $a' = \textsc{CtxSize}(\mathtt{a}, \mathtt{b}_{selfsize})$, and let $b' = \textsc{CtxSize}(\mathtt{b}, \mathtt{a}_{selfsize})$. These have equal widths, and so we can compare them bitwise. Finally, the one-bit result can be zero-extended to the width of the external context.

Expression splitting

We now create temporary wires for subexpressions so that no assignment has more than a single operator.

Similarly, we split up complex expressions used as inputs (not outputs) to module instantiations.

```
assign w = (a + b) - c;
 --->
wire [width:0] temp;
assign temp = a + b;
assign w = temp - c;
```

```
mymod inst( a + b, ...);
 --->
wire [width:0] temp2;
assign temp = a + b;
mymod inst(temp, ...);
```

I should make a well-formedness check but haven't, yet. I can check idempotency, at least.

Making truncation explicit

Cases for `assign lhs = rhs;`

- lhs width $=$ rhs width (fine)
- lhs width $>$ rhs width (impossible — ctxsize)
- lhs width $<$ rhs width (implicit truncation!)

We now correct for this, so all assignments agree on width.

```
wire [rhswidth - 1:0] temp;
assign temp = rhs;
assign lhs = temp[lhswidth-1:0];
```

Final expression optimizations

Oprewrite, split, and trunc often introduce needless expressions. It's pretty easy to just remove them with an additional transformation.

|a	$\longrightarrow$	a	when a is one-bit
a[0]	$\longrightarrow$	a	when a is one-bit
a[0:0]	$\longrightarrow$	a	when a is one-bit
a[n:n]	$\longrightarrow$	a[n]	

Things like this are nice. Easy to write test code for Cadence to check that the transformations are sound.

Occforming

We now get rid of assignments altogether by replacing them with module occurrences.

```
assign w = a + b;
  --->
VL_13_BIT_PLUS gensym(w, a, b);
```

This involves

- writing module definitions (e.g., defining `VL_13_BIT_PLUS`), and
- replacing assignments with module instances.

We can, e.g., exhaustively test `VL_4_BIT_PLUS` with Cadence.

Eliminating instance arrays

Especially in parameterized modules, instance arrays are sometimes used

```
wire [13:0] o;
wire a;
wire [13:0] b;
and foo [13:0] (w, a, b);  // 13 and-gates
```

We transform this into

```
and foo0 (w[0], a, b[0]);
 ...
and foo13 (w[13], a, b[13]);
```

The rules for slicing up wires are not too bad.

Latches and flops

Latches and flops are described with always-blocks

```
always @ (posedge clk)   // a basic flop
  place <= val;

always @ (foo or bar)    // a basic latch
  if (clk)
     place <= foo & bar;
```

Difficult because statements can be very complicated

- We have a plan to handle simple cases automatically
- But for right now we do it by hand

Writing modules

Our parser returns two values

- A list of the parsed modules, and
- A comment map – alist of locations to strings

Each main module item (wire declarations, assignments, etc.) also is tagged with its location.

- And when we split, we keep that

We can write the translated verilog in "the same order", with comments preserved.

- We also output various annotations, e.g., "port implicit"
- Still needs some work to become more readable

```
module iumularray (eph1, mcenclk_p, aopinb, bopinb, quadcfg, sgnd_mul, mpsum_a,
     mpcar_a, vdd0, vbna, vss0, vbpa);
  input vdd0 ;
  input vbna ;
  input vss0 ;
  input vbpa ;
  wire vdd0 ;                 // Port Implicit
  wire vbna ;                 // Port Implicit
  wire vss0 ;                 // Port Implicit
  wire vbpa ;                 // Port Implicit

// auto-generated for well bias support
  input eph1 ;
  wire eph1 ;                 // Port Implicit

//clk
  input mcenclk_p ;
  wire mcenclk_p ;            // Port Implicit

//inverted clock enable
  input [31:0] aopinb ;
  wire [31:0] aopinb ;        // Port Implicit

// multiplicand operand (inverted)
  input [31:0] bopinb ;
  wire [31:0] bopinb ;        // Port Implicit

// multiplier operand (inverted)
  input [1:0] quadcfg ;
  wire [1:0] quadcfg ;        // Port Implicit
```

```
/* For bopin[7 : 0] */
  VL_8_BIT_BUF _gen_313 (_gen_24, bopin[7 : 0]) ;

  VL_32_BIT_BUF _gen_495 (bop8ze, {_gen_22, _gen_23, _gen_24}) ;

//************************************************************************
//  CONTROL SIGNALS:
//
//  quadcfg:
//      case 00 : 32 x 32 => 64
//      case 10 : 16 x 16 => 32
//      case 11 :  8 x  8 => 16
//
//  sgnd_mul:
//      case 0 : unsigned
//      case 1 : signed
//
//************************************************************************

/* For ((| (((~ sgnd_mul) & quadcfg[1]) & quadcfg[0])) ? {aop8ze} : {aop}) */
  wire [31:0] _gen_71 ;

/* For ((| ((sgnd_mul & quadcfg[1]) & quadcfg[0])) ? {aop8se} : ((| (((~ sgnd_m\
ul) & quadcfg[1]
     ) & quadcfg[0])) ? {aop8ze} : {aop})) */
  wire [31:0] _gen_72 ;

/* For ((| (((~ sgnd_mul) & quadcfg[1]) & quadcfg[0])) ? {aop8ze} : {aop}) */
  VL_32_BIT_MUX _gen_321 (_gen_71, _gen_68, _gen_69, _gen_70) ;
```

Direct translation to E

A few additional transformations

- Give names to any unnamed instances
- Eliminate supply0 and supply1 wires
- Split up n-ary gates, e.g., and(o, i1, i2, i3)

The main algorithm

- Explode wires into a fast-alist, ensure no duplicates
- Compute :I, :O, :C, and :CD for the module
- Compute :I, :O, :U, :OP for each occurrence
- Assemble the `defm` call, submit w/ make-event
- Extract the resulting `defm-raw` calls and save them in a file

Only around 900 lines of Lisp with comments, various theorems
A better version would be more like the Verilog writer

Usage model

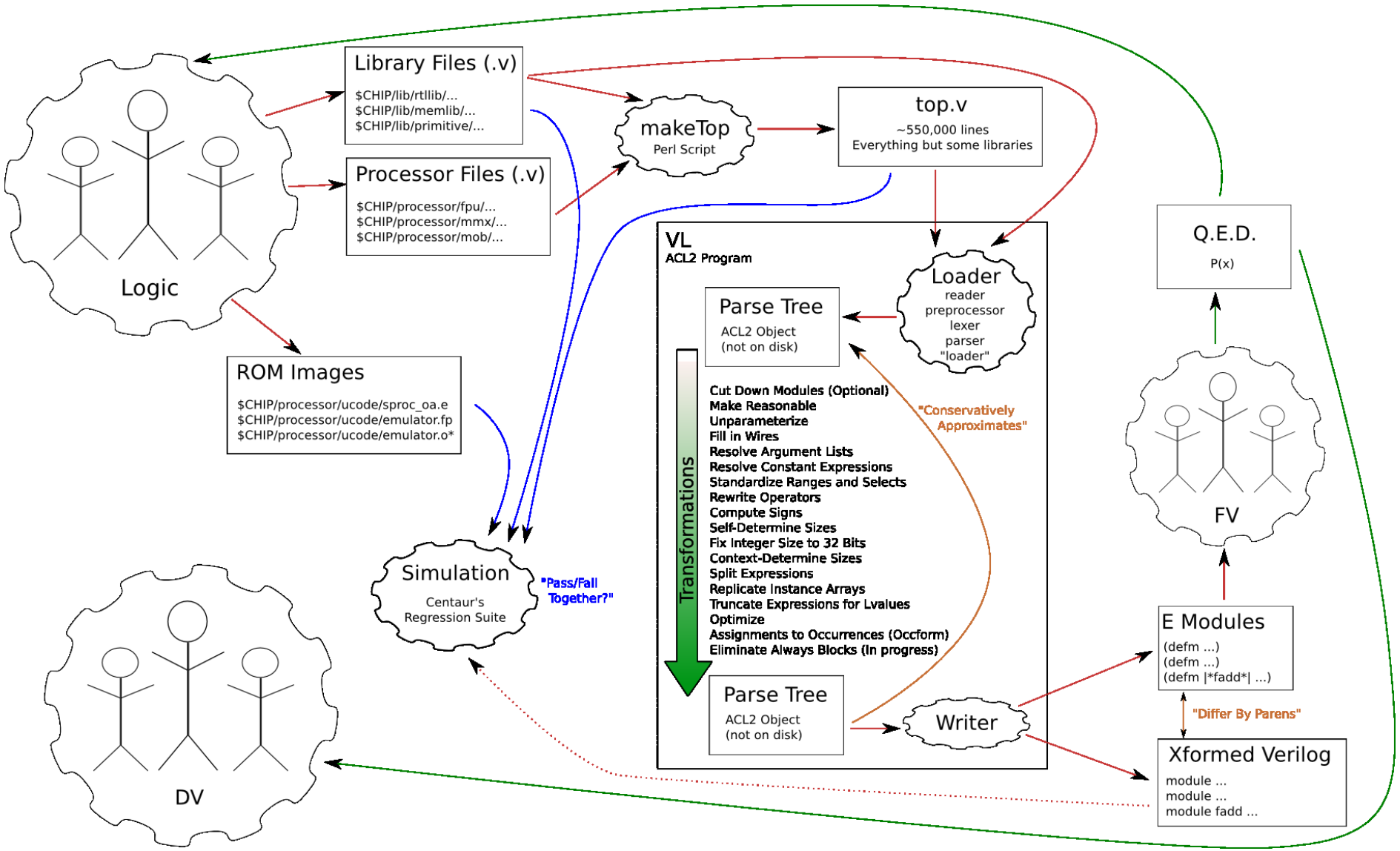
translate.sh runs the translator against the current copy of the chip

- Automatically run every night
- Stores everything we need into "/n/fv2/translated/[today]"
- Takes about 8 minutes (only doing certain modules)

refresh.sh builds an ACL2 image from the most recent translation

- Run by each user when they choose to update, undoable
- Copies today's translation into "mine" directory
- Builds **acl2cn** executable with E modules pre-loaded
- None of the translator books need to be included
- Takes about 4 minutes (only doing certain modules)

Actual Centaur proof-work is done with the acl2cn image.


Logic
Library Files (.v)
$CHIP/lib/rtllib/...
$CHIP/lib/memlib/...
$CHIP/lib/primitive/...
Processor Files (.v)
$CHIP/processor/fpu/...
$CHIP/processor/mmx/...
$CHIP/processor/mob/...
makeTop
Perl Script
top.v
~550,000 lines
Everything but some libraries
ROM Images
$CHIP/processor/ucode/sproc_oa.e
$CHIP/processor/ucode/emulator.fp
$CHIP/processor/ucode/emulator.o*
VL
ACL2 Program
Loader
reader
preprocessor
lexer
parser
"loader"
Parse Tree
ACL2 Object
(not on disk)
"Conservatively Approximates"
Transformations
Cut Down Modules (Optional)
Make Reasonable
Unparameterize
Fill in Wires
Resolve Argument Lists
Resolve Constant Expressions
Standardize Ranges and Selects
Rewrite Operators
Compute Signs
Self-Determine Sizes
Fix Integer Size to 32 Bits
Context-Determine Sizes
Split Expressions
Replicate Instance Arrays
Truncate Expressions for Lvalues
Optimize
Assignments to Occurrences (Occform)
Eliminate Always Blocks (In progress)
Parse Tree
ACL2 Object
(not on disk)
Writer
Simulation
Centaur's
Regression Suite
"Pass/Fail Together?"
DV
Q.E.D.
P(x)
FV
E Modules
(defm ...)
(defm ...)
(defm |*fadd*| ...)
"Differ By Parens"
Xformed Verilog
module ...
module ...
module fadd ...