INTEGRATED DESIGN AND VERIFICATION: AN OUNCE OF PREVENTION IS WORTH A POUND OF CURE

CARL SEGER
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY, GOTHENBURG, SWEDEN
SECARL@CHALMERS.SE
SEPTEMBER 6, 2022
BAD NEWS

• To verify a HW design is:
  • Hard (60-80% of ASIC design effort)
  • Time consuming

• To debug a HW design:
  • Is even worse!

• To debug combined SW/HW:
  • Is cause of short life span...
  • ..and lots of grey hair!
GOOD NEWS! It could be worse...
RESEARCH HYPOTHESES

• Hypothesis 1:
  • There are way more SW engineer than HW design engineers.
  • Therefore, specification language should be “similar” to traditional SW languages.

• Hypothesis 2:
  • Ad-hoc/post design verification is not feasible
  • Therefore, design and verification must be tightly integrated

• Hypothesis 3:
  • Design is a highly interactive activity
  • Therefore, the design environment must be a highly interactive with fast interactions, much visualization and with plenty of guidance as well as re-use of earlier work.
THOR PROJECT: BIG PICTURE

Idea

Specification DSL

Static Analysis + Visualization

Property driven tests + Formal property verification

CPT: Proven correctness preserving transformation

LFV: Local formal refinement verification

DTS: Previously verified transformation sequence from database

(CPT | LFV | DTS)*

RTL

FEV: Formal equivalence verification

Layout

RTL synthesis

Refinement
PART 1:

Idea

Specification DSL

Static Analysis + Visualization

Property driven tests + Formal property verification

CPT: Proven correctness preserving transformation

LFV: Local formal refinement verification

DTS: Previously verified transformation sequence from database

(CPT | LFV | DTS)*

RTL synthesis

FEV: Formal equivalence verification

Bifrost
HIGH-LEVEL SPEC. LANG. GOALS

• Separate “what” from “how”
  • The goal is to succinctly specify what is needed.

• Allow algorithmic specifications (“software like”)
  • Most natural specification for many problems.

• Replace timing with protocols
  • Isolate the specifier from the subtleties of communication.

• Make validation as easy as possible
  • Clean (simple) semantics, strong typing, visualization, …
**BIFROST**

- Aimed at iterative algorithms and interaction between multiple modules
- Imperative language that is compiled into hardware
  - Both control machine and data path control is created.
- Protocols between units are user selectable
  - Can change protocol by changing 1-2 lines.
- Protocols include not only functional control but also power/voltage control.
for each chunk
create a 64-entry message schedule
(The initial values in 80 16
all elements are zero or the copy chunk into first 16 words w[0..15] of the message schedule array

Extend the first 16 words into the remaining 48 words w[16..63] of the message schedule
for i from 16 to 63
s0 := (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightrotate 3)
s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor (w[i-2] rightrotate 10)
w[i] := w[i-16] + s0 + w[i-7] + s1

Initialize working variables to current hash value:
\[
a := h0; 
b := h1; 
c := h2; 
d := h3; 
e := h4; 
f := h5; 
g := h6; 
h := h7;
\]

Compression function main loop:
for i from 0 to 63
S1 := (e rightrotate 5) xor (e rightrotate 11) xor (e rightrotate 25)
ch := (e and f) xor ((not e) and g)
temp1 := h + S1 + ch + k[i] + w[i]
S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
maj := (a and b) xor (a and c) xor (b and c)
temp2 := S0 + maj
\[
h := g; 
g := f; 
f := e; 
e := d + temp1; 
d := c; 
c := b; 
b := a; 
a := temp1 + temp2
\]

Add the compressed chunk to the current hash value:
\[
h0 := h0 + a; 
h1 := h1 + b; 
h2 := h2 + c; 
h3 := h3 + d; 
h4 := h4 + e; 
h5 := h5 + f; 
h6 := h6 + g; 
h7 := h7 + h;
\]

Produce the final hash value (big-endian):
digest := hash := h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7 append...
UNITS & PROTOCOL DECLARATIONS

```c
#include "types.inc"

// Macros to access HFL functions
define zx = "\z, ZX z";
// Right shift by constant
define rshift3 = "\w, do_rshift w '3';
define rshift10 = "\w, do_rshift w '10';
// Rotate by constant
define rrot2 = "\w, do_rrot w '2';
define rrot4 = "\w, do_rrot w '4';
define rrot6 = "\w, do_rrot w '6';
define rrot8 = "\w, do_rrot w '8';
define rrot11 = "\w, do_rrot w '11';
define rrot13 = "\w, do_rrot w '13';
define rrot15 = "\w, do_rrot w '15';
define rrot18 = "\w, do_rrot w '18';
define rrot19 = "\w, do_rrot w '19';
define rrot22 = "\w, do_rrot w '22';
define rrot25 = "\w, do_rrot w '25';

// Memory interface
action mem_read:MemRead provided by external via pulseecho alwayson;
action mem_write:MemWrite provided by external via pulseecho alwayson;

action w_read:wread provided by external via pulseecho alwayson;
action w_write:wwrite provided by external via pulseecho alwayson;
action k_lookup:Ktbl provided by external via combinational alwayson;

define N = 10; // Max number of big adders to use
action add:Add[N] provided by "add" via combinational alwayson;

subroutine main : read_addr:addr -> res:signature
{
    // Initialize Wmem
}
```
DEMO OF BIFROST
CAPTURE OF SHA256
Slow SHA256 version
Slow SHA256 version
Fast SHA256 version
EXAMPLE OF RESULTS: SHA256

<table>
<thead>
<tr>
<th>Name</th>
<th>Reg.file #rd-ports</th>
<th>Constant memory #rd-ports</th>
<th># adders</th>
<th>Memory protocol</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>slow</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Pulse</td>
<td>765</td>
</tr>
<tr>
<td>medium</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>Pulse</td>
<td>327</td>
</tr>
<tr>
<td>4phase</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>4-phase</td>
<td>342</td>
</tr>
<tr>
<td>fast</td>
<td>2</td>
<td>2</td>
<td>14</td>
<td>Pulse</td>
<td>230</td>
</tr>
</tbody>
</table>

NOTE: Only the number of units and protocols were changed*.
PART II:

Idea

Specification DSL

- Static Analysis
  + Visualization
- Property driven tests + Formal property verification
- CPT: Proven correctness preserving transformation
- LFV: Local formal refinement verification
- DTS: Previously verified transformation sequence from database
- (CPT | LFV | DTS)*
- RTL
- FEV: Formal equivalence verification

IDV

Refinement

RTL synthesis

Layout
SIMPLE NEURAL NETWORK EVALUATOR FOR LOW-PWR IOT

```c
int main() {
    // External unit interfaces
    action M_rd,M_Rev[N] provided by external via combinational alwayson;
    action R_rd,R_Rev[N] provided by external via pulseecho alwayson;
    action get_w: UInt1[N] provided by external via combinational alwayson;
    action add: UInt1[N] provided by "add" via combinational alwayson;
    action mult: UInt1[N] provided by "mul" via combinational alwayson;

    // Define N = 4; // Number of external units to use
    define N = 4;
    define WID = 28;
    define HT = 28;
    define HIDDEN_LAYERS = 3;
    define DEPTH = 16;
    define OUTPUTS = 10;

    // Layer 1
    for(l = 0; l < DEPTH; l = l+1) {
        for(x = 0; x < WID; x = x+1) {
            for(y = 0; y < HT; y = y+1) {
                j = x + (y*WID+x);
                a0 = de M_rd (read_addr+(x*HT+y));
                w = do get_w l j ;
                sum = do add sum (do mul w a0);
            }
        }
        res = do relu sum;
        do R_wr l res;
    }

    // For hidden layers
    N = HIDDEN_LAYERS + 2;
    for(layer = 2; layer < N; layer = layer+1) {
        for(l = 0; l < DEPTH; l = l+1) {
            for(j = 0; j < DEPTH; j = j+1) {
                a[j] = do R_rd (read_addr+(j*DEPTH+1));
                w = do get_w layer j ;
                sum = do add sum (do mul w a[j]);
            }
        }
        res = do relu sum;
        do R_wr ((read_addr+(layer-1)*DEPTH+1) res;
    }

    // Output layer: determine max
    max = 0;
    choice = 0;
    for(l = 0; l < OUTPUTS; l = l+1) {
        sum = do get_b (((l-1)+1)*DEPTH+1);
        for(j = 0; j < DEPTH; j = j+1) {
            a[j] = do R_rd (read_addr+(j*DEPTH+1));
            w = do get_w N j l;
            sum = do add sum (do mul w a[j]);
        }
        res = do relu sum;
        if(res > max) {
            choice = l;
            max = res;
        }
    }
    return;
}
```
If we dig into this (and clean up some), we get...
CORE DATAPATH FROM BIFROST
DEMO OF IDV
$2sum[31:0]$

$1[31:0]$

$2[31:0]$

$3[31:0]$

$8[31:0]$

$12[31:0]$
$2sum[31:0] \rightarrow i11
$1[31:0] \rightarrow i12
$2[31:0] \rightarrow i13
$3[31:0] \rightarrow i14
$8[31:0] \rightarrow i15
\rightarrow _$12[31:0]$
Details of Stage 2
STATUS

• Thor is a less than one year old project, but many of the ingredients have been in development for many years as part of other projects.

• For Thor, we are currently proceeding on two fronts:
  • Developing the system (coding, coding, testing, testing, …)
  • Using it designing an IoT processor (eat your own dog food)
    • Working on deciding our next domain to apply Thor to
SUMMARY

To enable domain-specific hardware we need:

- A method for describing the circuit as “SW”
- A method to ensure the “SW” model is correct.
- A method for compiling the “SW” to “HW”
- A method for refining the “HW” to a realistic HW implementation.

Thor provides a proof-of-concept for such a system.
THANK YOU

QUESTIONS?

VossII and IDV available at: https://github.com/TeamVoss/VossII