Theorem-based Circuit Derivation in Cryptol

John Launchbury | GPCE | 2011
Challenge of Implementing Crypto

Creating a crypto algorithm requires skills in math AND programming

Variety of target architectures

Validation is complex and tedious

Variety of requirements
Lack of clear reference implementations

It's hard to relate implementations to the underlying math
Cryptol: Specifications and Formal Tools

- **Declarative specification language**
  - Language tailored to the crypto domain
  - Designed with feedback from NSA

- **Execution and Validation Tools**
  - Tool suite for different implementation and verification applications
  - In use by crypto-implementers

- **Automated Synthesis down to FPGA**
  - Algebraic rewrite-based compilation
  - Theorem-based derivation
One Specification - Many Uses

**Design**

\[ w_0 = u_I^I \mod p + u_I^I w_1 \mod p \]

\[ s = f^* (w_0 + pw_2) \mod q \]

**Validate**

**Build**

**Domain-specific design capture**

**Assured implementation**

- **Formal Models and test cases**
- **Hardware Implementation**
- **Software Implementation**
  - C, Haskell,…
- **FPGA**
- **Special purpose processor**
- **Verify crypto implementation**

- **Cryptol Workbench**
- **Cryptol**

Monday, October 24, 11
Cryptol Programs

- File of mathematical definitions
  - Two kinds of definitions: values and functions
  - Definitions may be accompanied by size-type declarations

- Definitions are computationally neutral
  - Cryptol tools provide the computational content (interpreters, compilers, code generators, verifiers)

```plaintext
x : [4][32];
x = [23 13 1 0];

F : ([16],[16]) -> [16];
F(x, x') = 2 * x + x';
```
Traversals

- Cartesian traversal

\[
\begin{align*}
[ | [x \ y] | | x &\leftarrow [0 \ 1 \ 2], \ y &\leftarrow [3 \ 4] | ] \\
= &\begin{bmatrix}
[0 & 3] \\
[1 & 3] \\
[2 & 3]
\end{bmatrix}
\begin{bmatrix}
[0 & 4] \\
[1 & 4] \\
[2 & 4]
\end{bmatrix}
\end{align*}
\]

- Parallel traversal

\[
\begin{align*}
[ | x + y | | x &\leftarrow [1 \ 2 \ 3] \\
| | y &\leftarrow [3 \ 4 \ 5 \ 6 \ 7] | ] \\
= & [4 \ 6 \ 8]
\end{align*}
\]
Stream Equations

\[
\begin{align*}
as &= \{\text{0x3F 0xE2 0x65 0xCA}\} \quad \# \text{ new; } \\
\text{new} &= \begin{array}[t]{c}
| \quad a \land b \land c \\
\lor \quad a \leftarrow \text{as} \\
\lor \quad b \leftarrow \text{drop}(1, \text{as}) \\
\lor \quad c \leftarrow \text{drop}(3, \text{as}) \\
| \end{array}
\end{align*}
\]
AES Structure

PT → Key addition → Byte substitution → Shift row → Mix column → Key addition → Byte substitution → Shift row → Key addition

T-box

Key Expansion runs in lockstep with encryption

CT

Key0 → Key1 → Key2 → Key3

Next Word → Round #
Cryptol: Specify interfaces unambiguously

From the Advanced Encryption Standard definition†

3.1 Inputs and Outputs

The input and output for the AES algorithm each consist of sequences of 128 bits (digits with values of 0 or 1). These sequences will sometimes be referred to as blocks and the number of bits they contain will be referred to as their length. The Cipher Key for the AES algorithm is a sequence of 128, 192 or 256 bits. Other input, output and Cipher Key lengths are not permitted by this standard.

blockEncrypt : {k} (k >= 2, 4 >= k) => ([128], [64*k]) -> [128]

For all k  ...between 2 and 4  First input is a sequence of 128 bits  Second input is a sequence of 128, 192, or 256 bits  Output is a sequence of 128 bits
Cryptol in practice

- Create a Cryptol reference specification.
- Execute the specification, including assertion checks.
- Generate test vectors with Quickcheck to bundle with the reference specification.
Why implement crypto algorithms on FPGAs?

- For high-grade crypto, software on commodity hardware is not trusted
  - Hardware-only solutions are the norm
- Custom crypto chips are increasingly common
  - But they are complex, and there is no standard
  - FPGAs are more flexible, and built on simple building blocks
- Natural match between Cryptography and FPGAs
  - Manipulation of arbitrary length bit sequences
  - Highly parallel stream processing
Why use Cryptol to produce FPGAs?

- Cryptol’s fixed-length sequences naturally model hardware bit-vectors
  - A Cryptol function maps inputs to outputs, with no side effects
  - Model state-holding elements within a stream
- Sequentialization in Cryptol comes only from data-dependency
  - Sequences can be mapped over space or time
  - There is an implicit clock signal (one clock period per element in sequential sequences)
  - The user can explore space-time tradeoffs without significantly changing the Cryptol source

Cryptol naturally describes hardware
Cryptol in FPGA Development

- It is easier to experiment in Cryptol than in VHDL
- Explore many architectures
- Generate circuits and evaluate performance

- Cryptol reference specification
- Symbolic evaluator
- Cryptol interpreter
- Equivalence checker
- Test Vectors
- Implementation model
- Symbolic evaluator
- Netlist model
- Symbolic simulator
- Equivalence checker
- System Simulation
- Cryptol implementation specification
- Cryptol compiler
- Synthesis
- C
- VHDL
- Netlist
- Bitfile

Galois tools
FPGA Vendor tools
Specification
Data files produced by Cryptol tools
Input to tool
Data files produced by vendor tools
Feedback to designer
Source files

Monday, October 24, 11
Producing AES Cores

**AES Encrypt**
- Reference
- Fuse KeyGen
- T-Boxes
- Pipelined

**AES Decrypt**
- Reference
- Fuse KeyGen
- T-Boxes
- Pipelined
Non-linear Components

SBox : [256][8];
SBox = [ 0x63 0x7c 0x77 0x7b 0xf2 0x6b 0x6f 0xc5 0x30 0x01 0x67 0x2b 0xfe 0xd7 0xab 0x76
0xca 0x82 0xc9 0x7d 0xf0 0xad 0xd4 0xa2 0xaf 0x9c 0xa4 0x72 0xc0
0xb7 0xfd 0x93 0x26 0x36 0x3f 0xf7 0xcc 0x34 0xa5 0xe5 0xf1 0x71 0xd8 0x31 0x15
...
0x8c 0xa1 0x89 0x0d 0xbf 0xe6 0x42 0x68 0x41 0x99 0x2d 0x0f 0xb0 0x54 0xbb 0x16 ];

UBox : [256][8];
UBox = [ 82 9 106 213 48 54 165 56 191 64 163 158 129 243 215 251
124 227 57 130 155 87 167 241 120 204 246 147 150 162 245
84 123 148 50 166 195 78 224 106 119 214 38 225 105 20
...
23 43 4 126 186 119 214 38 225 105 20 ];

aes-utils> :check
*** Checking "SUBoxes" ["aes-utils.cry", line 76, col 1]
Checking case 256 of 256 (100.00%)
Q.E.D.
[Coverage: 100%. (256/256)]

theorem SUBoxes: {b}. UBox @ (SBox @ b) == b;
Mix Columns

- Treat each column as a cubic polynomial over GF(2^8)
- Generate new columns by multiplying by a fixed cubic polynomial \(3X^3+X^2+X+2\), modulo \(X^4+1\)
- In decrypt, the polynomial is \(11X^3+13X^2+9X+14\)
MixColumns

mixColumn : [4][8] -> [4][8];
mixColumn([y0 y1 y2 y3]) = 
    (gTimes2(y0) ^ gTimes3(y1) ^ y2 ^ y3)
    (y0 ^ gTimes2(y1) ^ gTimes3(y2) ^ y3)
    (y0 ^ y1 ^ gTimes2(y2) ^ gTimes3(y3))
    (gTimes3(y0) ^ y1 ^ y2 ^ gTimes2(y3))

mixColumnD : [4][8] -> [4][8];
mixColumnD([y0 y1 y2 y3]) = 
    (gTimese(y0) ^ gTimesb(y1) ^ gTimesd(y2) ^ gTimes9(y3))
    (gTimes9(y0) ^ gTimese(y1) ^ gTimesb(y2) ^ gTimesd(y3))
    (gTimesd(y0) ^ gTimes9(y1) ^ gTimese(y2) ^ gTimesb(y3))
    (gTimesb(y0) ^ gTimesd(y1) ^ gTimes9(y2) ^ gTimese(y3))

theorem mixColumnOk: {x}. (mixColumnD (mixColumn x) == x);
MixColumns

mixColumn : [4][8] -> [4][8];
mixColumn([y0 y1 y2 y3]) = 
\[(y0 ^ gTimes2(y1) ^ gTimes3(y2) ^ y3) \]
\[(y0 ^ y1 ^ gTimes2(y2) ^ gTimes3(y3)) \]
\[(gTimes3(y0) ^ y1 ^ y2 ^ gTimes2(y3)) \];

mixColumnD : [4][8] -> [4][8];
mixColumnD([y0 y1 y2 y3]) = 
\[(gTimese(y0) ^ gTimesb(y1) ^ gTimesd(y2) ^ gTimes9(y3)) \]
\[(gTimes9(y0) ^ gTimese(y1) ^ gTimesb(y2) ^ gTimesd(y3)) \]
\[(gTimesd(y0) ^ gTimes9(y1) ^ gTimese(y2) ^ gTimesb(y3)) \]
\[(gTimesb(y0) ^ gTimesd(y1) ^ gTimes9(y2) ^ gTimese(y3)) \];

aes-imp1> :check
*** 2 Theorems to be checked.
*** [1/2] Checking "mixColumn'Ok" ["aes-imp1.cry", line 55, col 1]
Checking case 1000 of 1000 (100.00%)
1000 tests passed OK
[Coverage: 2.33e-5%. (1000/4294967296)]

theorem mixColumnOk: {x}. (mixColumnD (mixColumn x) == x);
Key Expansion

KeyExpansion : [4][32] -> [11][4][4][8];
KeyExpansion keyAsWords = [\ transpose s \ s <- ss ]
  where { ss = groupBy(4, [\ reverse(splitBy(4,w)) \ w <- ws ]);
    ws = keyAsWords # ([\ nextWord(i,w,w')
      || i <- [4 .. 43]
      || w <- ws
      || w' <- drop(3,ws) ]); }

nextWord : ([8],[32],[32]) -> [32];
nextWord(i,w,w') = w ^ temp
  where { temp = if i \% 4 == 0
    then SubWord(RotWord(w')) ^ Rcon' (take (4, i / 4))
    else w'; }

Key expansion will be merged to run in lockstep with encryption — so we need to create a reversed version.
Compute a Reversed Key Expansion

KeyDExpansion : [4][32] -> [11][4][4][8];
KeyDExpansion keyAsWords = [ [ [ reverse v || v <- transpose s ] || s <- ss ] ]
where { ss = groupBy(4,[ [ reverse(splitBy(4,w)) || w <- ws ] ])
    ws = keyAsWords # ([ [ nextWord(i,w,w') || i <- [43 -- 4]
        || w <- ws ] ])
}

*** [2/5] Checking "KeyDExpansionThm" ["aes-imp1.cry", line 119, col 1]
Checking case 1000 of 1000 (100.00%)
1000 tests passed OK
[Coverage: 0.00%. (1000/340282366920938463463374607431768211456)]

flipKey KeyAsWords
    = splitBy(4,join (reverse (join (transpose (KeyExpansion KeyAsWords ! 0))))));

theorem KeyDExpansionThm:
    {k}. KeyDExpansion (flipKey k) == reverse (KeyExpansion k);
oneRound : ([4], ([4][4][8],[4][32])) -> ([4][4][8],[4][32]);

oneRound (round, (state, key)) = (next_state, next_key)
where { state' = if round == 1 then state else d
    next_state = AddRoundKey(state',key);
    next_key = nextKey(round, key);
    d = if round == 11 then SubBytes(ShiftRows(state))
        else MixColumns'(SubBytes(ShiftRows(state))); };

Key expansion is now merged to run in lockstep with encryption
Producing AES Cores

AES Encrypt
- Reference
- Fuse KeyGen
- T-Boxes
- Pipelined

AES Decrypt
- Reference
- Fuse KeyGen
- T-Boxes
- Pipelined
oneRound_Tbox : ([4], ([4][4][8],[4][32])) -> ([4][4][8],[4][32]);

oneRound_Tbox (round, (state, key)) = (next_state, next_key)
where { state' = if round == 1 then state else transpose d;
    next_state = AddRoundKey(state',key);
    next_key = nextKey(round, key);
    d = [| if (round == 11) then [(t0@1) (t1@2) (t2@3) (t3@0)] else t0 ^ t1 ^ t2 ^ t3
        where { b0 = state @ 0 @ (j+0); b1 = state @ 1 @ (j+1);
            b2 = state @ 2 @ (j+2); b3 = state @ 3 @ (j+3);
            t0 = T0 b0; t1 = T1 b1; t2 = T2 b2; t3 = T3 b3; } |
        || j <- [0 .. 3] : [4][2] // note: j is 2 bits, so arith is mod 4 |]; |};

theorem TboxOk: {r s k}. oneRound(r,(s,k)) == oneRound_Tbox(r,(s,k));
\[
T_0, T_1, T_2, T_3 : [8] \rightarrow [4][8];
\]
\[
T_0(a) = T_0\_table \_at \ a;
\]
\[
T_1(a) = T_0(a) \ggg 1;
\]
\[
T_2(a) = T_0(a) \ggg 2;
\]
\[
T_3(a) = T_0(a) \ggg 3;
\]
\[
T_0\_table = \text{const } [\| T_0\_func(a) \| \ a <- [0..255] ];
\]
\[
T_0\_func : [8] \rightarrow [4][8];
\]
\[
T_0\_func(a) = [(g\_Times2 s) \ s \ s \ (g\_Times3 s)] \text{ where } s = \text{SBox } \_at \ a;
\]
\[\]
\[\text{theorem TFns: } \{a \ b \ c \ d\}\.\]
\[\text{T0 a } ^ \text{T1 b } ^ \text{T2 c } ^ \text{T3 d } = = \text{mixColumn } [\| \text{SBox } \_at \ x \| \ x <- [a \ b \ c \ d ] ];\]
\[\text{theorem UFns: } \{a \ b \ c \ d\}\.\]
\[\text{U0 a } ^ \text{U1 b } ^ \text{U2 c } ^ \text{U3 d } = = \text{mixColumnD } [\| \text{UBox } \_at \ x \| \ x <- [a \ b \ c \ d ] ];\]
Building U-Boxes

oneRound_Ubox : ([4], ([4][4][8],[4][32])) -> ([4][4][8],[4][32]);

oneRound_Ubox (round, (state, key)) = (state", next_key)
  where { state' = if round == 10 then state else transpose d;
    state" = AddRoundKey(state', if round == 10 | round == 0 then key_state
               else MixColumnsD key_state);
    key_state = ProjectKey(key);
    next_key = nextKeyD(round, key);
    d = [| if (round == 0) then [(UBox @ b0) (UBox @ b1) (UBox @ b2) (UBox @ b3)]
      else u0 ^ u1 ^ u2 ^ u3
      where { b0 = state @ 0 @ (j-0); b1 = state @ 1 @ (j-1);
        b2 = state @ 2 @ (j-2); b3 = state @ 3 @ (j-3);
        u0 = U0 b0; u1 = U1 b1; u2 = U2 b2; u3 = U3 b3;  }
    || j <- [0 .. 3] |];

theorem UboxOk: {r s k}. (oneRoundD(r,(s,k)) == oneRound_Ubox(r,(s,k)));
oneRound_Ubox : ([4], ([4][4][8],[4][32])) -> ([4][4][8],[4][32]);
oneRound_Ubox (round, (state, key)) = (state'', next_key)
  where { state' = if round == 10 then state else transpose d;
          state'' = AddRoundKey(state', if round == 10 | round == 0 then key_state
                      else MixColumnsD key_state);
          next_key = nextKeyD(round, key);
          d = [ | if (round == 0) then [(UBox @ b0) (UBox @ b1) (UBox @ b2) (UBox @ b3)]
                      else u0 ^ u1 ^ u2 ^ u3
                      where { b0 = state @ 0 @ (j-0); b1 = state @ 1 @ (j-1);
                                b2 = state @ 2 @ (j-2); b3 = state @ 3 @ (j-3);
                                u0 = U0 b0; u1 = U1 b1; u2 = U2 b2; u3 = U3 b3; } ]
          || j <- [0 .. 3] |];

theorem UboxOk: {r s k}. (oneRoundD(r,(s,k)) == oneRound_Ubox(r,(s,k)));
Results

- Modes for Cryptol
  - LLSPIR: gate level counts etc.
  - VHDL
  - FSIM: simulation flow, without place & route;
  - TSIM: full simulation flow
- Xilinx Virtex-4
- Simulation with place & route reports slightly lower number:
  - *The Best Case achievable is 3.074ns, which corresponds to 325MHz*
  - This is *very good* performance.... :-)
- Size usage is higher than encrypt, but for good reasons

```
aes-ubox-reg> :translate Cipher_Tbox_reg  // encrypt

......

            Number of Slices:                     6438  out of  63168    10%
            Number of Slice Flip Flops:          11968  out of  126336     9%
            Number of 4 input LUTs:               6304  out of  126336     4%
```

Minimum period: 2.943ns (Maximum Frequency: 339.784MHz)
Minimum input arrival time before clock: 2.378ns
Maximum output required time after clock: 4.376ns
Maximum combinational path delay: No path found

Device Utilization (size summary):
-----------------------------------
Selected Device : 4vlx60ff668-12

```
Number of Slices:                     10571  out of  26624    39%
Number of Slice Flip Flops:          19256  out of  53248     36%
Number of 4 input LUTs:               13501  out of  53248     25%
```
Cryptol Assurance Tools

- **“Quickcheck” property-based testing**
  - Cryptol automatically tests property on random inputs.

- **Translators to SAT- and SMT-based property checkers**
  - SAT: Checks for satisfiability of large Boolean formulas
  - SMT extends SAT with higher-level constraint solvers (linear arithmetic, arrays, functions, etc.)

- **Safety checking**
  - Automatically checks that a Cryptol function will never raise an exception
    - Divide-by-zero, Out-of-bounds array access, assertion failures

- **Semi-automatic theorem proving**
  - Translator from Cryptol to Isabelle theorem prover
Cryptol in the evaluation process

A crypto-device evaluator:
- Creates a reference specification and associated formal model
- Checks the equivalence of the implementation models at several points in the tool

The process works for both hand-written and Cryptol-generated designs.
Equivalence Checking: Some Lessons Learned

- Technique is very useful in practice
  - Equivalence Checking of various versions of AES and DES against both reference models and internal FPGA models take < 5 minutes (most take < 30 seconds)
  - Models typically have $\sim10^6$ nodes

- However, it has its limitations
  - The Hash Function MD5 on 2 bits takes nearly 10 minutes…
  - The block cipher RC6, however takes, uh, too long (32-bit multiply)
  - Only works on core ciphers — not modes (finite input/output)
  - The equivalence checker typically yields an answer promptly, or it timeouts
Questions?

- Free download of interpreter
- Documentation
- Evaluation license for FPGA and equivalence checking tool flows
- Cryptol courses

www.cryptol.net