Hardware Design for Cryptographers

P. Schaumont

Bradley Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, VA

Why should a cryptographer care about hardware?

- Many interesting application domains (RFID, Wirespeed processing, ..) require hardware implementation.
- Hardware gives the best performance. Eg. ECC2K-130 cryptanalysis, bitcoin mining.
- Competitions. Keccak (SHA-3 Winner) was recognized early-on as having superior hardware performance.
- Improved algorithm design by taking implementation constraints into account. Eg. Lightweight crypto: PRESENT (CHES07), photon (CHES11).
- Implementation attacks (faults and SCA) are tightly connected to implementation.
Why should a cryptographer care about hardware?

- Many interesting application domains (RFID, Wirespeed processing, ..) require hardware implementation.
- Hardware gives the *best* performance. Eg. ECC2K-130 cryptanalysis, bitcoin mining.
- Competitions. Keccak (SHA-3 Winner) was recognized early-on as having superior hardware performance.
- Improved algorithm design by taking implementation constraints into account. Eg. Lightweight crypto: PRESENT (CHES07), photon (CHES11).
- Implementation attacks (faults and SCA) are tightly connected to implementation.
Why should a cryptographer care about hardware?

- Many interesting application domains (RFID, Wirespeed processing, ..) require hardware implementation.
- Hardware gives the best performance. Eg. ECC2K-130 cryptanalysis, bitcoin mining.
- Competitions. Keccak (SHA-3 Winner) was recognized early-on as having superior hardware performance.
- Improved algorithm design by taking implementation constraints into account. Eg. Lightweight crypto: PRESENT (CHES07), photon (CHES11).
- Implementation attacks (faults and SCA) are tightly connected to implementation.
Why should a cryptographer care about hardware?

- Many interesting application domains (RFID, Wirespeed processing, ..) require hardware implementation.
- Hardware gives the *best* performance. Eg. ECC2K-130 cryptanalysis, bitcoin mining.
- Competitions. Keccak (SHA-3 Winner) was recognized early-on as having superior hardware performance.
- Improved algorithm design by taking implementation constraints into account. Eg. Lightweight crypto: PRESENT (CHES07), photon (CHES11).
- Implementation attacks (faults and SCA) are tightly connected to implementation.
Why should a cryptographer care about hardware?

- Many interesting application domains (RFID, Wirespeed processing, ..) require hardware implementation.
- Hardware gives the best performance. Eg. ECC2K-130 cryptanalysis, bitcoin mining.
- Competitions. Keccak (SHA-3 Winner) was recognized early-on as having superior hardware performance.
- Improved algorithm design by taking implementation constraints into account. Eg. Lightweight crypto: PRESENT (CHES07), photon (CHES11).
- Implementation attacks (faults and SCA) are tightly connected to implementation.
But (too) often, hardware is an afterthought!

E.g. NIST Call for SHA-3 submissions:

- A reference implementation shall be submitted in order to promote the understanding of how the candidate algorithm may be implemented. This implementation shall consist of source code written in ANSI C.

- To demonstrate the efficiency of a hardware implementation of the algorithm, the submitter may include a specification of the algorithm in a nonproprietary Hardware Description Language (HDL).
Hey. Why didn’t NIST say:

- A reference implementation shall be submitted in order to promote the understanding of how the candidate algorithm may be implemented. This implementation shall consist of source code written in Verilog 2001.

- To demonstrate the efficiency of a software implementation of the algorithm, the submitter may include a specification of the algorithm in a nonproprietary Software Programming Language (ANSI C).
Hardware Misconceptions

- **To design hardware, I will need to dig deep into technology.** Not true. You can create technology-independent hardware descriptions.

- **Hardware design tools are complex and expensive.** Not true. HDL simulators are free. FPGA implementation tools are free.

- **I can write C and then use a C-to-HDL converter. Why bother?** Not efficient. Certainly, some tools may help you part of the way, but none are as smart as you.

- **Writing algorithms (ciphers) in hardware is more difficult than in software.** Not true. This is a matter of practice.
Hardware Misconceptions

- *To design hardware, I will need to dig deep into technology.*
  Not true. You can create technology-independent hardware descriptions.

- *Hardware design tools are complex and expensive.*
  Not true. HDL simulators are free. FPGA implementation tools are free.

- *I can write C and then use a C-to-HDL converter. Why bother?*
  Not efficient. Certainly, some tools may help you part of the way, but none are as smart as you.

- *Writing algorithms (ciphers) in hardware is more difficult than in software.*
  Not true. This is a matter of practice.
Hardware Misconceptions

- To design hardware, I will need to dig deep into technology. Not true. You can create technology-independent hardware descriptions.

- Hardware design tools are complex and expensive. Not true. HDL simulators are free. FPGA implementation tools are free.

- I can write C and then use a C-to-HDL converter. Why bother? Not efficient. Certainly, some tools may help you part of the way, but none are as smart as you.

- Writing algorithms (ciphers) in hardware is more difficult than in software. Not true. This is a matter of practice.
Hardware Misconceptions

- *To design hardware, I will need to dig deep into technology.*  
  Not true. You can create technology-independent hardware descriptions.

- *Hardware design tools are complex and expensive.*  
  Not true. HDL simulators are free. FPGA implementation tools are free.

- *I can write C and then use a C-to-HDL converter. Why bother?*  
  Not efficient. Certainly, some tools may help you part of the way, but none are as smart as you.

- *Writing algorithms (ciphers) in hardware is more difficult than in software.*  
  Not true. This is a matter of practice.
Objectives of this presentation

- Explain basic principles of **hardware description** (as opposed to software programming).
- Demonstrate technology-independent hardware description with ANSI C.
- We will not discuss low-level technology mapping (= the path from hardware description to gates).
- Underlying idea: if you (the cryptographer) make a hardware-friendly spec, hardware designers will deliver better results.
Outline

1. Hardware-oriented Specification
2. Hardware Circuits
3. Hardware Description in C
In this talk, we’ll make a few assumptions

<table>
<thead>
<tr>
<th>In software programs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- operations execute sequentially, instruction by instruction</td>
</tr>
<tr>
<td>- storage is central and may be indexed (arrays)</td>
</tr>
<tr>
<td>- variables have a native data type (eg 32 bit)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In hardware descriptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- operations execute in parallel, cycle by cycle</td>
</tr>
<tr>
<td>- storage is distributed and cannot be indexed</td>
</tr>
<tr>
<td>- variables can have any length, but we’ll assume 32 bit</td>
</tr>
</tbody>
</table>
Software variables

- A software variable is a container for values of a predefined type.
- A software variable can be read and written.

```c
int a; // a 32-bit integer
```
Hardware variables

- Example uses a hypothetical hardware description language.
- A `reg` is a hardware variable with storage. It may be written and read in a different clock cycle. When writing a value in cycle \( n \), the value cannot be read before cycle \( n + 1 \).
- A `wire` is a hardware variable without storage. It must be written and read in the same clock cycle. When writing a value in cycle \( n \), the value cannot be read after cycle \( n \).
Hardware Counter

reg a;
a = a + 1;

- In cycle n, write the value of \( a \) plus one into \( a \).
- The value of \( a \) in cycle \( n+1 \) will be the value written into \( a \) in cycle \( n \).
- This is a counter incrementing once per clock cycle.
- What is \( a \)’s initial value? The description does not say. We’ll assume it’s zero. Hardware implementations need to take care of initialization by implementing proper reset logic.
What does this circuit do?

Hardware Counter

```
reg a;
a = a + 1;
```

- In cycle n, write the value of $a$ plus one into $a$.
- The value of $a$ in cycle $n+1$ will be the value written into $a$ in cycle $n$.
- This is a counter incrementing once per clock cycle.
- What is $a$’s initial value? The description does not say. We’ll assume it’s zero. Hardware implementations need to take care of initialization by implementing proper reset logic.
What does this circuit do?

Hardware Counter

```plaintext
wire a;

a = a + 1;
```

- In cycle n, write the value of `a` plus one into `a`.
- The value of `a` in cycle n will be the value written into `a` in cycle n.
- This is a counter that increments infinitely fast.
- This is bad hardware design! It’s asynchronous and unstable.
What does this circuit do?

Hardware Counter

```plaintext
wire a;
a = a + 1;
```

- In cycle \( n \), write the value of \( a \) plus one into \( a \).
- The value of \( a \) in cycle \( n \) will be the value written into \( a \) in cycle \( n \).
- This is a counter that increments infinitely fast.
- This is **bad hardware design**! It’s asynchronous and unstable.
What does this circuit do?

Hardware Counter

\begin{verbatim}
reg a, b;
a = a + 1;
b = b + 1;
\end{verbatim}

- In cycle \( n \), the values of registers \( a \) and \( b \) are incremented.
- This is a dual counter. The two statements appear to execute in parallel.
What does this circuit do?

**Hardware Counter**

```c
reg a, b;
a = a + 1;
b = b + 1;
```

- In cycle n, the values of registers `a` and `b` are incremented.
- This is a dual counter. The two statements appear to execute in parallel.
What does this circuit do?

**Hardware Counter**

```plaintext
reg a;
wire b;
a = b + 1;
b = a + 1;
```

- In cycle $n$, write the value of $b$ plus one into $a$.
- In cycle $n$, write the value of $a$ plus two into $a$.
- This is a counter that increments in steps of two.
- Again, both statements execute in parallel. The result is determined by data dependencies and by the variable types (reg, wire).
What does this circuit do?

Reg  a;
wire  b;
a  =  b + 1;
b  =  a + 1;

- In cycle n, write the value of $b$ plus one into $a$.
- In cycle n, write the value of $a$ plus two into $a$.
- This is a counter that increments in steps of two.
- Again, both statements execute in parallel. The result is determined by data dependencies and by the variable types (reg, wire).
Indexed Variables

- In software, *arrays* and *pointers* represent indexed storage, and they assume a central, shared memory.

- In hardware, *arrays* can be treated either as separate registers (a[0], a[1], ..), or as a memory (reading a[i] is the same as reading address i from memory a).

- This distinction (register or memory) is important for the interpretation of a snippet of code such as

  for (i=0; i<10; i++)
    a[i] = a[i] + 1;

- In this presentation, we’ll stick to the *register* interpretation. A consequence is that *pointers* have no meaning.
An expression is a circuit

- With the proper interpretation of hardware variables (reg or wire), an expression or set of expressions becomes a circuit.
Linear Feedback Shift Register

```vhdl
reg lfsr;
wire next;
next = (lfsr & 1) ? 0xD0000001u : 0;
lfsr = (lfsr >> 1) ^ next;
```

- In cycle n, test the LSB of register `lfsr`, and use it to select a constant value `0xD0000001u` or `0x0`.
- In the same clock cycle, shift `lfsr` one position down, xor it with the constant value (from the same clock cycle), and update `lfsr`.
- Initialization of `lfsr` is missing; this cannot work.
Linear Feedback Shift Register

reg lfsr;
wire next;
next = (lfsr & 1) ? 0xD0000001u : 0;
lfsr = (lfsr >> 1) ^ next;

- In cycle n, test the LSB of register `lfsr`, and use it to select a constant value `0xD0000001u` or `0x0`.
- In the same clock cycle, shift `lfsr` one position down, xor it with the constant value (from the same clock cycle), and update `lfsr`.
- Initialization of `lfsr` is missing; this cannot work.
Linear Feedback Shift Register (2)

Works as before, but this circuit has an extra input `load`. When `load` is non-zero during clock cycle n, the `lfsr` register will hold the value 1 in clock cycle n+1.
Linear Feedback Shift Register (2)

Works as before, but this circuit has an extra input `load`.

When `load` is non-zero during clock cycle n, the `lfsr` register will hold the value 1 in clock cycle n+1.
Control deals with conditional execution of operations.

In software, control statements such as if, while, for deal with conditional execution.

In hardware descriptions, there are no control statements. (Loops etc. serve the purpose of syntactical sugar rather than functionality. See further).

Control is a source of great confusion and pain for aspiring hardware designers. It shouldn’t.

In hardware (= a parallel, distributed execution environment), control design requires careful treatment.
Control as conditional update

The easiest form of hardware control is conditional state update.

Up-down counter

```c
reg count;
reg down;
count = down ? count - 1 : count + 1;
down = (count == 9) ? 1 :
    (count == 1) ? 0 : down;
```

What will be the range of counter values? 0 to 10
Control as conditional update

The easiest form of hardware control is conditional state update.

Up-down counter

```c
reg count;
reg down;

count = down ? count - 1 : count + 1;
down = (count == 9) ? 1 :
      (count == 1) ? 0 : down;
```

What will be the range of counter values? 0 to 10
Control as conditional update

Finite State Machines - if you’re familiar with those - can be expressed with conditional state update too.

Up-down counter

```c
reg count;
reg state; // 0=up, 1=down
count = state ? count - 1 : count + 1;
state = ((state == 0) && (count == 9)) ? 1 :
    ((state == 1) && (count == 1)) ? 0 :
        state;
```

That’s right. Essentially the same as previous design ...
Summary - Fundamental Hardware Concepts

1. A hardware program is a set of expressions with \texttt{reg} or \texttt{wire} variables
2. A hardware program represents a circuit
3. The meaning of the program is independent of the lexical order of statements
4. Control is implemented by conditional update of \texttt{reg} variables

Claim: If cryptographers would provide such designs as reference, hardware designers can deliver more efficient implementations and with less 'dumb mistakes'.
Outline

1. Hardware-oriented Specification
2. Hardware Circuits
3. Hardware Description in C
We can support the fundamental concepts of hardware description with C.

In the following, we’ll discuss how to convert general C programs into ’hardware oriented’ C programs.
If we implement this C program in hardware, we intend to run it *many* times. For every new \( b \), we will compute a new \( a \).
The hardware interpretation of this C program

```c
infinite loop {
    read input b;
    do in hardware {
        int a, b;
        a = a + b;
        a = a * 5;
        a = a + 3;
    }
    write output a;
}
```

Next step: pick a cycle budget. We’ll assume one cycle to compute `do in hardware`. 
The hardware interpretation of this C program

```c
every_clock_cycle {
    read b;
    a = a + b;
    a = a * 5;
    a = a + 3;
    write a;
}
```

Next step: analyze read-write dependencies and map `int` into `reg` or `wire`. In C, we will keep using `int`, but we’ll make explicit what should be `reg` and what should be `wire`. 
The hardware interpretation of this C program

every_clock_cycle {
    read b;
    a1 = a3@1 + b;
    a2 = a1 * 5;
    a3 = a2 + 3;
    write a3;
}

To help this analysis, we use a single-assignment formulation. 
a3@1 means: the a3 value from the previous clock cycle.
The hardware interpretation of this C program

```c
input b;
output a3;
wire b, a1, a2, a3_next;
reg a3;
a1 = a3 + b;
a2 = a1 * 5;
a3_next = a2 + 3;
a3 = a3_next;
```
Mapping it in C

For this hardware model ..

```plaintext
input b;
output a3;
wire b, a1, a2;
wire a3_next;
reg a3;
a1 = a3 + b;
a2 = a1 * 5;
a3_next = a2 + 3;
a3 = a3_next;
```

.. you’d write this in C

```c
int a3, a3_next;
int a1, a2, b;
a1 = a3 + b;
a2 = a1 * 5;
a3_next = a2 + 3;
// state update
a3 = a3_next;
```
That’s it!

.. and you’d write this in Verilog

```verilog
define module any(input wire[31:0] b,
                    input wire clk,
                    output reg[31:0] a3)
  wire [31:0] a3_next, a1, a2;
  always @(posedge clk)
    a3 = a3_next;
  assign a1 = a3 + b;
  assign a2 = a1 * 5;
  assign a3_next = a2 + 3;
endmodule
```
Multi-cycle C programs

```c
int a, b;
a = a + b;
```

Let’s do another example, and assume a cycle budget of two clock cycles. How do you split one single addition?
This transformation is called bitslicing: computing an $N$-bit operation in $k$ cycles by processing $N/k$ bit per sub-operation.
### Single-cycle model with control

```c
every_cycle {
    read16 b;
    phi = phi ? 0 : 1; // control
    if (phi == 0) {
        q = al + b;
        c = (q >> 16);
        al = q & 0xFFFF;
    } else {
        q = ah + b + c;
        ah = q;
    }
    write16 q;
}
```
Hardware Version of the Multicycle Program

```c
reg phi, c, al, ah;
wire b, t, q;
phi = phi ? 0 : 1;
t = phi ? ah : al;
q = t + b + c;
c = phi ? 0 : q[16];
al = phi ? al : q;
ah = phi ? q : ah;
```
int1 phi, phi_next, c, c_next;
int16 al, ah, al_next, ah_next;
int16 b, t;
int17 q;
phi_next = phi ? 0 : 1;
t = phi ? ah : al;
q = t + b + c;
c_next = phi ? 0 : (q >> 16);
al_next = phi ? al : q;
ah_next = phi ? q : ah;
// update
phi = phi_next;
c = c_next;
al = al_next;
ah = ah_next;
Recipe for Straight Line C

1. Decide cycle budget
2. Convert code to single-assignment code
3. Partition expressions over clock cycles
   - Partition horizontally: *scheduling*, reuse hardware to compute similar expressions
   - Partition vertically: *bitslicing*, reuse hardware to compute long operations
4. Identify `reg` and `wire` variables
5. Express hardware model in C
The general strategy is to unroll, and convert the result as straight-line C.

Data-dependent loops cannot be unrolled; they will need additional control hardware.
Cubehash example

```c
static void transform(unsigned x[]) {
    int i;
    int r;
    unsigned y[16];

    for (r = 0; r < CUBEHASH_ROUNDS; ++r) {
        for (i = 0; i < 16; ++i) x[i + 16] += x[i];
        for (i = 0; i < 16; ++i) y[i ^ 8] = x[i];
        for (i = 0; i < 16; ++i) x[i] = ROTATE(y[i], 7);
        for (i = 0; i < 16; ++i) x[i] ^= x[i + 16];
        for (i = 0; i < 16; ++i) y[i ^ 2] = x[i + 16];
        for (i = 0; i < 16; ++i) x[i + 16] = y[i];
        for (i = 0; i < 16; ++i) x[i + 16] += x[i];
        for (i = 0; i < 16; ++i) y[i ^ 4] = x[i];
        for (i = 0; i < 16; ++i) x[i] = ROTATE(y[i], 11);
        for (i = 0; i < 16; ++i) x[i] ^= x[i + 16];
        for (i = 0; i < 16; ++i) y[i ^ 1] = x[i + 16];
        for (i = 0; i < 16; ++i) x[i + 16] = y[i];
    }
}
```
Cubehash mapping - 1 cycle per round

1-cycle design: Unroll and convert to single-assignment form

```c
for (i = 0; i < 16; ++i)
    x[i + 16] += x[i];
```

```c
x1_16 = xin_16 + xin_0;
x1_17 = xin_17 + xin_1;
x1_18 = xin_18 + xin_2;
x1_19 = xin_19 + xin_3;
x1_20 = xin_20 + xin_4;
x1_21 = xin_21 + xin_5;
x1_22 = xin_22 + xin_6;
x1_23 = xin_23 + xin_7;
x1_24 = xin_24 + xin_8;
x1_25 = xin_25 + xin_9;
x1_26 = xin_26 + xin_10;
x1_27 = xin_27 + xin_11;
x1_28 = xin_28 + xin_12;
x1_29 = xin_29 + xin_13;
x1_30 = xin_30 + xin_14;
x1_31 = xin_31 + xin_15;
```
Cubehash mapping - Throughput and Efficiency

Performance optimized UMC 130nm \((fsc0g_d_sc_tc.lib)\)

\[ Tp = \frac{\text{Clock}}{\text{cycles}} / 16 \times 256 \text{ (in Gbps)} \]

\[ \text{Eff} = \frac{Tp}{\text{GE}} \text{ (in kbps/gate)} \]

<table>
<thead>
<tr>
<th>Clock MHz</th>
<th>GE cycles per round</th>
<th>Tp</th>
<th>Eff</th>
</tr>
</thead>
<tbody>
<tr>
<td>347</td>
<td>30553</td>
<td>5.55</td>
<td>181</td>
</tr>
</tbody>
</table>
Cubehash mapping - 2 cycles per round

Notice the symmetry in:

```c
for (r = 0; r < CUBEHASH_ROUNDS; ++r) {
    //------------------------------------------
    for (i = 0; i < 16; ++i) x[i + 16] += x[i];
    for (i = 0; i < 16; ++i) y[i ^ 8] = x[i];
    for (i = 0; i < 16; ++i) x[i] = ROTATE(y[i], 7);
    for (i = 0; i < 16; ++i) x[i] ^= x[i + 16];
    for (i = 0; i < 16; ++i) y[i ^ 2] = x[i + 16];
    for (i = 0; i < 16; ++i) x[i + 16] = y[i];
    //------------------------------------------
    for (i = 0; i < 16; ++i) x[i + 16] += x[i];
    for (i = 0; i < 16; ++i) y[i ^ 4] = x[i];
    for (i = 0; i < 16; ++i) x[i] = ROTATE(y[i], 11);
    for (i = 0; i < 16; ++i) x[i] ^= x[i + 16];
    for (i = 0; i < 16; ++i) y[i ^ 1] = x[i + 16];
    for (i = 0; i < 16; ++i) x[i + 16] = y[i];
    //------------------------------------------
}
```
Cubehash mapping - 2 cycles per round

2-cycle design: Unroll, convert to single-assignment form, and merge

(cycle 1):
\[
\begin{align*}
    \text{for } (i = 0; i < 16; ++i) & \quad y[i \oplus 8] = x[i]; \\
    \text{for } (i = 0; i < 16; ++i) & \quad x[i] = \text{ROTATE}(y[i], 7);
\end{align*}
\]

(cycle 2):
\[
\begin{align*}
    \text{for } (i = 0; i < 16; ++i) & \quad y[i \oplus 4] = x[i]; \\
    \text{for } (i = 0; i < 16; ++i) & \quad x[i] = \text{ROTATE}(y[i], 11);
\end{align*}
\]
Cubehash mapping - 2 cycles per round

.. expands to:

\[
\begin{align*}
  t0 &= \text{cycle} \ ? \ \text{ROL}(x04,11) : \ \text{ROL}(x08,7); \\
  t1 &= \text{cycle} \ ? \ \text{ROL}(x05,11) : \ \text{ROL}(x09,7); \\
  t2 &= \text{cycle} \ ? \ \text{ROL}(x06,11) : \ \text{ROL}(x0a,7); \\
  t3 &= \text{cycle} \ ? \ \text{ROL}(x07,11) : \ \text{ROL}(x0b,7); \\
  t4 &= \text{cycle} \ ? \ \text{ROL}(x00,11) : \ \text{ROL}(x0c,7); \\
  t5 &= \text{cycle} \ ? \ \text{ROL}(x01,11) : \ \text{ROL}(x0d,7); \\
  t6 &= \text{cycle} \ ? \ \text{ROL}(x02,11) : \ \text{ROL}(x0e,7); \\
  t7 &= \text{cycle} \ ? \ \text{ROL}(x03,11) : \ \text{ROL}(x0f,7); \\
  t8 &= \text{cycle} \ ? \ \text{ROL}(x0c,11) : \ \text{ROL}(x00,7); \\
  t9 &= \text{cycle} \ ? \ \text{ROL}(x0d,11) : \ \text{ROL}(x01,7); \\
  ta &= \text{cycle} \ ? \ \text{ROL}(x0e,11) : \ \text{ROL}(x02,7); \\
  tb &= \text{cycle} \ ? \ \text{ROL}(x0f,11) : \ \text{ROL}(x03,7); \\
  tc &= \text{cycle} \ ? \ \text{ROL}(x08,11) : \ \text{ROL}(x04,7); \\
  td &= \text{cycle} \ ? \ \text{ROL}(x09,11) : \ \text{ROL}(x05,7); \\
  te &= \text{cycle} \ ? \ \text{ROL}(x0a,11) : \ \text{ROL}(x06,7); \\
  tf &= \text{cycle} \ ? \ \text{ROL}(x0b,11) : \ \text{ROL}(x07,7); \\
\end{align*}
\]
Cubehash mapping - Throughput and Efficiency

Performance optimized UMC 130nm ($\text{fsc0g_d_sc_tc.lib}$)

$$T_p = \frac{\text{Clock}}{\text{cycles}} \times 16 \times 256 \text{ (in Gbps)}$$

$$\text{Eff} = \frac{T_p}{\text{GE}} \text{ (in kbps/gate)}$$

<table>
<thead>
<tr>
<th>Clock MHz</th>
<th>GE cycles per round</th>
<th>$T_p$</th>
<th>Eff</th>
</tr>
</thead>
<tbody>
<tr>
<td>347</td>
<td>30553</td>
<td>1</td>
<td>5.55</td>
</tr>
<tr>
<td>558</td>
<td>21053</td>
<td>2</td>
<td>4.47</td>
</tr>
</tbody>
</table>
Cubehash mapping - 4 cycles per round

4-cycle design: Bit-sliced 2-cycle design

2-cycle code:

\[
\begin{align*}
    s0 &= x_{10} + x_{00}; \\
    s1 &= x_{11} + x_{01}; \\
    &\ldots
\end{align*}
\]

4-cycle code:

\[
\begin{align*}
    s0 &= x_{10} + x_{00} + \text{carry}_0; \\
    s1 &= x_{11} + x_{01} + \text{carry}_1; \\
    \ldots \\
    \text{newcarry}_0 &= \text{cycle} \& (s0 < x_{00}); \\
    \text{newcarry}_1 &= \text{cycle} \& (s1 < x_{01}); \\
    \ldots \\
    \text{carry}_0 &= \text{newcarry}_0; \\
    \text{carry}_1 &= \text{newcarry}_1; \\
    \ldots
\end{align*}
\]
Cubehash mapping - Throughput and Efficiency

Performance optimized UMC 130nm (*fsc0g_d_sc_tc.lib*)

\[ Tp = \frac{\text{Clock}}{\text{cycles}} \times \frac{16}{256} \text{ (in Gbps)} \]

\[ \text{Eff} = \frac{Tp}{\text{GE}} \text{ (in kbps/gate)} \]

<table>
<thead>
<tr>
<th>Clock MHz</th>
<th>GE</th>
<th>cycles per round</th>
<th>Tp</th>
<th>Eff</th>
</tr>
</thead>
<tbody>
<tr>
<td>347</td>
<td>30553</td>
<td>1</td>
<td>5.55</td>
<td>181</td>
</tr>
<tr>
<td>558</td>
<td>21053</td>
<td>2</td>
<td>4.47</td>
<td>212</td>
</tr>
<tr>
<td>621.1</td>
<td>15698</td>
<td>4</td>
<td>2.48</td>
<td>158</td>
</tr>
</tbody>
</table>
Even more cycles per round

See D.J. Bernstein’s implementations:

- 8 cycles per round:
  http://cubehash.cr.yp.to/hardware8/hash.c

- 16 cycles per round:
  http://cubehash.cr.yp.to/hardware16/hash.c

- 32 cycles per round:
  http://cubehash.cr.yp.to/hardware32/hash.c
Conclusions

- In most cases, you - the algorithm designer (or cryptographer) - are in the best position to come up with guidelines for hardware implementation.

- To suggest a hardware implementation, you do not have to write HDL. C is adequate to describe the most important ideas.

- The best practice is to try it. Send your code to a hardware designer and evaluate. Collaborate on CHES submissions, CAESAR submissions, ...

Hardware Design is not the exclusive domain of HDL programmers.