Low-Area Implementations of SHA-3 Candidates

Jens-Peter Kaps

Cryptographic Engineering Research Group (CERG)
http://cryptography.gmu.edu
Department of ECE, Volgenau School of IT&E,
George Mason University, Fairfax, VA, USA

SHA-3 Project Review Meeting
Outline

1. Introduction
2. Implementations
3. Results

Kaps
Low-Area Implementations of SHA-3 Candidates
Motivation

- There have been several comparison of Throughput/Area optimized implementations [Gaj],[Matsuo],[Baldwin],[Guo].
- Only few low-area implementations of single SHA-3 algorithms on FPGAs.
- Not all fully autonomous, Varying interface assumptions.
- Low-area implementations highlight flexibility of algorithm designs.
Motivation

- There have been several comparison of Throughput/Area optimized implementations [Gaj],[Matsuo],[Baldwin],[Guo].
- Only few low-area implementations of single SHA-3 algorithms on FPGAs.
- Not all fully autonomous, Varying interface assumptions.
- Low-area implementations highlight flexibility of algorithm designs.

Goal

- First comprehensive comparison of low-area implementations of Round 2 SHA-3 Candidates.
- All use the same standardized interface.
- All optimized for the same parameters under the same assumptions.
Implementing for minimum area alone can lead to unrealistic run-times.

⇒ Goal: Achieve the maximum Throughput/Area ratio for a given area budget.

Realistic scenario:

- System on Chip: Certain area only available.
- Standalone: Smaller Chip, lower cost, but limit to smallest chip available, e.g. 768 slices on smallest Spartan 3 FPGA.
Implementing for minimum area alone can lead to unrealistic run-times.

⇒ Goal: Achieve the maximum Throughput/Area ratio for a given area budget.

Realistic scenario:
- System on Chip: Certain area only available.
- Standalone: Smaller Chip, lower cost, but limit to smallest chip available, e.g. 768 slices on smallest Spartan 3 FPGA.

Target
- Xilinx Spartan 3e, low cost FPGA family
- Budget: 500 slices, 1 Block RAM (BRAM)
Interface

- Based on Interface and I/O Protocol from [Gaj], \( w = 16 \).
- \( \text{msg}_\text{len}_\text{ap}, \text{seq}_\text{len}_\text{ap} \) (after padding) in 32-bit words.
- \( \text{msg}_\text{len}_\text{bp}, \text{seq}_\text{len}_\text{bp} \) (before padding) in bits.

\[
\text{msg}_\text{len}_\text{bp} = \sum_{i=0}^{n-2} \text{seq}_\text{len}_\text{ap}_i \cdot 32 + \text{seq}_\text{len}_\text{bp}_{n-1}
\]

\[
\text{msg}_\text{len}_\text{ap} = \sum_{i=0}^{n-1} \text{seq}_\text{len}_\text{ap}_i \cdot 32
\]

\( w \) bits

<table>
<thead>
<tr>
<th>seq_len_ap_0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq_0</td>
<td></td>
</tr>
<tr>
<td>seq_len_ap_1</td>
<td>0</td>
</tr>
<tr>
<td>seq_1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>seq_len_ap_{n-1}</td>
<td>1</td>
</tr>
<tr>
<td>seq_{n-1}</td>
<td></td>
</tr>
</tbody>
</table>

- SHA Interface
- SHA Protocol
Minimization Technique

Datapath
- Use BRAM to store state, initialization vectors, constants
- Use BRAM in each clock cycle
- Avoid temporary storage or use:
  - Free registers, i.e. unused flip-flops after LUTs
  - Shift Registers (1x16 bit / Distributed RAM (1x16 bit)
    \[\Rightarrow 1 \text{ LUT} = \frac{1}{2} \text{ Slice}\]

Control Logic
- Small main state machine, up-to 8 states
- Counter for clock cycles in longest state
- Stored Program Control within states
- BRAM addressing must follow regular sequence, can have offset between rounds
We use exclusively \textit{read\_first} mode, i.e. old value is read, new value is written.

Saves clock cycles, however, leads to address offset.

Control logic might become difficult.

**Limits**

- Maximum 2 input and 2 output ports, 2 addresses (dual port).
- Maximum single port w/ 64 bits or dual port w/ 32 bits each.
Smallest implementation would be $\frac{1}{2}$ G-function $\rightarrow$ BRAM contention.

Best result: 2 G-functions, pipelined as shown above. Keeps data in-flight longer, eliminates BRAM contention.

However, generation of addresses difficult $\Rightarrow$ Large control logic.
CubeHash

- Initialization vector pre-processed stored in BRAM.
- BRAM contention requires swapping data between BRAM and Distributed RAM (DRAM).
  - This requires additional clock cycles.
  - Leads to simpler control logic.
- Finalization very costly at 9,296 clock cycles.
- Mix-Columns contains 2 Mix-Column units with total 8x8 bit register.
- 4 logic based S-Boxes with integrated pipeline stage.
- Faster Mix-Columns would exceed 500 slices.
- Key Generation uses DRAM, 32 bit adder. Allows store salt.
- Small control unit with room for improvement.
4 logic based S-Boxes followed by pipeline stage.
- Only fixed rotations.
- Most time consuming function: SMIX.
- Challenge to store the S-Mix table.
Grøstl

- Serialized P and Q.
- Only 2 S-Boxes due to Shift Rows → can only use 2x8 bits.
- Uses single set of registers for two Mix Column units.
- 16 / 32 bit datapath → 7 clock cycles per column.
Uses BRAM and two 32x8 DRAMs.

- Grouping and de-grouping are the most expensive operations.
  - 160 clock cycle operation.
  - multiple narrow memory accesses.
- Message injection uses serialized XORs.
- SubCrumb is implemented as DROM.
- Constraints: DRAM had to be used to keep control logic simple.
Based on paper by [Detrey].

- BRAM contains state register C and initialization vectors.
- Our I/O is more complex than [Detrey].
- BRAM makes controller more complex.
SHAvite-3

- 4 ROM based S-Boxes.
- Regular path of the mds matrix is an advantage.
- Mix column is realized through a shift register.
SHA-2

- Full Datapath
  - 700 slices
  - 65 clock cycles
SHA-2

- Full Datapath
  - 700 slices
  - 65 clock cycles

- Small Datapath
  - 520 slices
  - 595 clock cycles
SHA-2

- Full Datapath
  - 700 slices
  - 65 clock cycles

- Small Datapath
  - 520 slices
  - 595 clock cycles

SHA-2 is not well suited for very small implementations.
### Performance Equations

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Block Size (bits) $b$</th>
<th>Clock Cycles to hash $N$ blocks $clk =$</th>
<th>Throughput $b/(l + p) \cdot T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAKE</td>
<td>512</td>
<td>$18 + (32 + 480) \cdot N + 65$</td>
<td>$512/(512 \cdot T)$</td>
</tr>
<tr>
<td>CubeHash</td>
<td>256</td>
<td>$2 + (16 + 928) \cdot N + 9312$</td>
<td>$256/(928 \cdot T)$</td>
</tr>
<tr>
<td>ECHO</td>
<td>1536</td>
<td>$16 + (96 + 2449) \cdot N + 17$</td>
<td>$1536/(2545 \cdot T)$</td>
</tr>
<tr>
<td>Fugue</td>
<td>32</td>
<td>$33 + (2 + 61) \cdot N + 990$</td>
<td>$32/(63 \cdot T)$</td>
</tr>
<tr>
<td>Grøstl</td>
<td>512</td>
<td>$2 + (32 + 1120) \cdot N + 577$</td>
<td>$512/(1152 \cdot T)$</td>
</tr>
<tr>
<td>JH</td>
<td>512</td>
<td>$35 + (32 + 1574) \cdot N + 17$</td>
<td>$512/(1606 \cdot T)$</td>
</tr>
<tr>
<td>Luffa</td>
<td>256</td>
<td>$2 + (16 + 606) \cdot N + 647$</td>
<td>$256/(622 \cdot T)$</td>
</tr>
<tr>
<td>Shabal</td>
<td>512</td>
<td>$36 + (32 + 48) \cdot N + 208$</td>
<td>$512/(80 \cdot T)$</td>
</tr>
<tr>
<td>SHA[3]vite-3</td>
<td>512</td>
<td>$18 + (32 + 648) \cdot N + 17$</td>
<td>$512/(680 \cdot T)$</td>
</tr>
<tr>
<td>SHA-256</td>
<td>512</td>
<td>$18 + (32 + 563) \cdot N + 17$</td>
<td>$256/(595 \cdot T)$</td>
</tr>
</tbody>
</table>
## Implementation Results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Area (slices)</th>
<th>Block RAMs</th>
<th>Maximum Delay (ns) T</th>
<th>Throughput (Mbps) Large m</th>
<th>Throughput/Area (Mbps/slice)</th>
<th>Throughput (Mbps) Small m</th>
<th>Throughput/Area (Mbps/slice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAKE</td>
<td>538</td>
<td>1</td>
<td>9.73</td>
<td>102.7</td>
<td>0.19</td>
<td>88.4</td>
<td>0.164</td>
</tr>
<tr>
<td>CubeHash</td>
<td>292</td>
<td>1</td>
<td>9.84</td>
<td>28.0</td>
<td>0.09</td>
<td>2.5</td>
<td>0.008</td>
</tr>
<tr>
<td>ECHO</td>
<td>499</td>
<td>1</td>
<td>10.87</td>
<td>55.5</td>
<td>0.11</td>
<td>54.8</td>
<td>0.110</td>
</tr>
<tr>
<td>Fugue</td>
<td>480</td>
<td>1</td>
<td>11.52</td>
<td>44.0</td>
<td>0.09</td>
<td>2.5</td>
<td>0.005</td>
</tr>
<tr>
<td>Grøstl</td>
<td>474</td>
<td>1</td>
<td>8.96</td>
<td>49.6</td>
<td>0.10</td>
<td>33</td>
<td>0.070</td>
</tr>
<tr>
<td>JH</td>
<td>371</td>
<td>1</td>
<td>10.98</td>
<td>29.0</td>
<td>0.08</td>
<td>28.6</td>
<td>0.077</td>
</tr>
<tr>
<td>Luffa</td>
<td>453</td>
<td>1</td>
<td>9.42</td>
<td>43.6</td>
<td>0.10</td>
<td>21.4</td>
<td>0.047</td>
</tr>
<tr>
<td>Shabal</td>
<td>498</td>
<td>1</td>
<td>8.84</td>
<td>723.9</td>
<td>1.45</td>
<td>178.8</td>
<td>0.359</td>
</tr>
<tr>
<td>SHAvite-3</td>
<td>496</td>
<td>1</td>
<td>6.99</td>
<td>107.7</td>
<td>0.22</td>
<td>102.4</td>
<td>0.206</td>
</tr>
<tr>
<td>SHA-256</td>
<td>520</td>
<td>1</td>
<td>10.01</td>
<td>42.9</td>
<td>0.08</td>
<td>40.6</td>
<td>0.070</td>
</tr>
</tbody>
</table>
Results for Large Messages

![Diagram showing throughput vs. area for various SHA-3 candidates.]

- **Kaps**: Low-Area Implementations of SHA-3 Candidates
Results for Short Messages

![Graph showing latency vs. message size for various SHA-3 candidates: BLAKE, CubeHash, ECHO, Fugue, Groestl, JH, Luffa, Shabal, and SHAvite−3. The x-axis represents message size in bytes, ranging from 0 to 1600 bytes. The y-axis represents latency in nanoseconds, ranging from 0 to 6 x 10^5 nanoseconds. Each candidate is represented by a different line color. The graph illustrates the performance of each candidate across the given message sizes.]
Control Logic vs. Datapath

![Bar chart showing the comparison between Datapath and Controller for SHA-3 candidates. The candidates include CubeHash, JH, Luffa, Groestl, Fugue, SHA-3, Shabal, ECHO, SHA-2, and BLAKE. The chart displays the area in slices for each candidate, with a focus on low-area implementations.](image-url)
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Reference</th>
<th>Area (slices)</th>
<th>Block RAMs</th>
<th>Maximum Delay (ns)</th>
<th>I/O Width</th>
<th>Datapath Width</th>
<th>Clock Cycles (l + p)</th>
<th>Device</th>
<th>Functionality</th>
<th>Throughput (Mbps)</th>
<th>Throughput/Area (Mbps/slice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAKE-32</td>
<td>TW</td>
<td>124</td>
<td>2</td>
<td>5.20</td>
<td>32</td>
<td>32</td>
<td>816</td>
<td>xc3s50</td>
<td>FA</td>
<td>115.0</td>
<td>0.93</td>
</tr>
<tr>
<td>BLAKE-32</td>
<td>TW</td>
<td>538</td>
<td>1</td>
<td>9.73</td>
<td>16</td>
<td>32</td>
<td>512</td>
<td>xc3s100e</td>
<td>FA</td>
<td>102.7</td>
<td>0.19</td>
</tr>
<tr>
<td>BMW</td>
<td></td>
<td>895</td>
<td>1</td>
<td>26.00</td>
<td>32</td>
<td>32</td>
<td>1060</td>
<td>xcv300</td>
<td>FA</td>
<td>9.0</td>
<td>0.01</td>
</tr>
<tr>
<td>CubeHash</td>
<td>[TW]</td>
<td>292</td>
<td>1</td>
<td>9.84</td>
<td>16</td>
<td>32</td>
<td>944</td>
<td>xc3s100e</td>
<td>FA</td>
<td>28.0</td>
<td>0.10</td>
</tr>
<tr>
<td>ECHO</td>
<td>[TW]</td>
<td>499</td>
<td>1</td>
<td>10.87</td>
<td>16</td>
<td>32</td>
<td>2545</td>
<td>xc3s100e</td>
<td>FA</td>
<td>55.5</td>
<td>0.11</td>
</tr>
<tr>
<td>ECHO</td>
<td>[?]</td>
<td>127</td>
<td>1</td>
<td>2.80</td>
<td>8</td>
<td>8</td>
<td>6593</td>
<td>xc5vlx50-2</td>
<td>FA</td>
<td>72.0</td>
<td>0.57</td>
</tr>
<tr>
<td>Fugue</td>
<td>[TW]</td>
<td>480</td>
<td>1</td>
<td>11.52</td>
<td>16</td>
<td>32</td>
<td>63</td>
<td>xc3s100e</td>
<td>FA</td>
<td>44.0</td>
<td>0.09</td>
</tr>
<tr>
<td>Grøstl</td>
<td>[?]</td>
<td>1276</td>
<td>0</td>
<td>16.67</td>
<td>64</td>
<td>64</td>
<td>1152</td>
<td>xc3s1500</td>
<td>FA</td>
<td>192.0</td>
<td>0.15</td>
</tr>
<tr>
<td>Grøstl</td>
<td>[TW]</td>
<td>474</td>
<td>1</td>
<td>8.96</td>
<td>16</td>
<td>32</td>
<td>1641</td>
<td>xc3s100e</td>
<td>FA</td>
<td>49.6</td>
<td>0.10</td>
</tr>
<tr>
<td>JH</td>
<td>[TW]</td>
<td>371</td>
<td>1</td>
<td>10.98</td>
<td>16</td>
<td>32</td>
<td>3870</td>
<td>xc5vlx50</td>
<td>FA</td>
<td>29.0</td>
<td>0.08</td>
</tr>
<tr>
<td>Keccak</td>
<td>[?]</td>
<td>444</td>
<td>1</td>
<td>3.77</td>
<td>64</td>
<td>64</td>
<td>622</td>
<td>xc3s100e</td>
<td>FA</td>
<td>70.0</td>
<td>0.16</td>
</tr>
<tr>
<td>Luffa</td>
<td>[TW]</td>
<td>453</td>
<td>1</td>
<td>9.42</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>xc3s200</td>
<td>FA</td>
<td>43.6</td>
<td>0.10</td>
</tr>
<tr>
<td>Shabal</td>
<td>[?]</td>
<td>499</td>
<td>0</td>
<td>1.25</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>xc3s100e</td>
<td>FA</td>
<td>800</td>
<td>1.60</td>
</tr>
<tr>
<td>Shabal</td>
<td>[TW]</td>
<td>498</td>
<td>1</td>
<td>8.84</td>
<td>16</td>
<td>32</td>
<td>80</td>
<td>xc3s100e</td>
<td>FA</td>
<td>723.9</td>
<td>1.45</td>
</tr>
<tr>
<td>Shavite-3</td>
<td>[TW]</td>
<td>496</td>
<td>1</td>
<td>6.99</td>
<td>16</td>
<td>32</td>
<td>680</td>
<td>xc3s100e</td>
<td>FA</td>
<td>107.7</td>
<td>0.22</td>
</tr>
</tbody>
</table>
Comparison of Candidate Implementations

![Graph showing Throughput/Area comparison for various SHA-3 candidates]

- **BLAKE-32**
- **BMW**
- **Cubehash**
- **ECHO**
- **Fugue**
- **Groestl**
- **JH**
- **Keccak**
- **Luffa**
- **Shabal**
- **Shavite-3**
- **SHA-2**

The graph illustrates the throughput-area comparison for different implementations, with a focus on low-area implementations of SHA-3 candidates.
Best Candidate Implementations

Throughput/Area

Virtex

Spartan

Kaps

Low-Area Implementations of SHA-3 Candidates
Thanks for your attention.