# A Comparison of the AES Candidates Amenability to FPGA Implementation

Nicholas Weaver, John Wawrzynek
{nweaver,johnw}@cs.berkeley.edu*

March 15, 2000

**Abstract**

The 5 final AES candidates, MARS, RC6, Rijndael, Serpent, and Twofish, are all intended to run well both on hardware and software implementations. However, the different algorithms may result in significant differences in cost and performance when implemented on FPGAs or in small custom devices. This document discusses the various algorithms from the perspective of a potential FPGA implementer. Rijndael and Twofish are excellent candidates from a hardware designer's viewpoint, while MARS is particularly expensive and inefficient.

## 1 Introduction

The 5 final AES candidates, MARS[2], RC6[6], Rijndael[4], Serpent[1], and Twofish[7], are all designed to run efficiently on a wide variety of hardware and software. However, the candidates vary in their amenability to hardware implementations. In this paper we estimate the relative cost and performance for various possible implementations of the different AES algorithms. Although no actual implementations are realized, it is straightforward to estimate the cost of various possible realizations of the AES candidates.

## 2 Observations on the candidates

The following observations will be justified throughout the paper.

MARS is not a very suitable cipher for a hardware implementation. The three separate round types, the use of both an expensive multiplier and numerous large S-box references, and a complicated subkey generation, all combine to make it a poor candidate.

RC6, though it uses comparatively expensive operations, is a reasonable candidate unless subkey generation is important. The ability to reasonably reduce the hardware requirements without sacrificing too much performance is present, a useful feature when a low cost implementation is desired. However, the subkey generation, which has a tight dependency and needs to visit elements multiple times, poses a considerable challenge for any application which needs to change keys frequency.

Rijndael is probably the best candidate when subkey flexibility isn't essential. All operations are highly parallel but comparatively inexpensive in hardware, and the subkey generation is both fast and compact. However, the additional cost of creating a separate datapath if decryption is required somewhat hampers the design, and the subkey generation may still have an impact, depending on the application.

Serpent, surprisingly enough, is not the best candidate from a hardware standpoint. Although it uses very short operations which map naturally to hardware, 32 instances of

*This work was suppored by DARPA, contract number DABT63-96-C-0048. Further support comes from the California State MICRO Program.

each of the 8 types of S-boxes quickly add up, and if a compact implementation is desired, the bandwidth is considerably reduced. Also, there is essentially no sharing between encryption and decryption pipelines.

Twofish is the best overall from a hardware viewpoint. Although not as fast as Rijndael and Serpent, the ability to perform encryption and decryption with a trivially modified pipeline is quite valuable. Also, there is a nice tradeoff space between area and performance. If subkeys are not changed, the subkey generation can largely be folded into the pipeline. If area is still tight, the pipeline can be folded in half. However, if subkeys are changed often and performance is critical, the ability to change subkeys from block to block with almost no performance penalty, whether encrypting or decrypting, is of significant potential benefit. This degree of flexibility is unique to Twofish, and is a very desirable property.

# 3 Possible implementation techniques

There are 3 primary criteria when measuring the candidates using a hardware metric: latency, bandwidth, and area. Latency is the amount of time required to encrypt a single block of data. If the cipher is operating in CFB or similar modes, the latency of encryption may be the critical factor. Bandwidth is the number of blocks which can be computed in a given period of time. If there is no feedback on the ciphertext, such as in ECB mode, bandwidth indicates how fast data can be encrypted. Area is a specific metric, which generally suggests the cost for an implementation. Lower area is generally beneficial, as this allows lower cost parts to be used.

The implementation fabric being considered is the Xilinx Virtex[9] Field Programmable Gate Array, which consists of an array of 4 input lookup tables (4-LUTs)[1] and associated flip flops, plus a perimeter of medium sized, dual ported, 512 byte BlockRAM memories[2]. Each 4-input lookup table can also act as a 16-bit RAM, for storing temporary values.

Embedded, small to medium sized memory blocks are becoming ubiquitous on modern FPGAs, although many older devices (such as the Xilinx 4000 series) lack such features. Thus, the use of such memories needs to be considered separately. It is, however, safe to assume that practically all future devices will have such capabilities.

It is comparatively easy to estimate the size of a hand layed out datapath for these applications, as the dataflows are suitably regular to allow the functional units to be packed together. The cost of the control logic is not considered, because for the AES candidates the primary cost is the datapath.

Similarly, the cost of generating the encryption subkeys is considered separately; for some algorithms subkey generation may be better implemented on a small microcontroller[3]. Some applications may use constant subkeys or subkeys which change only rarely, in which case subkey generation time is not a concern. However, in other applications where encryption keys may change on a packet-by-packet basis, subkey generation can become the dominant factor in the time it takes to encrypt a block.

Although the sketches described are geared towards FPGAs, a good rule of thumb is that, except for memories, logic in an FPGA takes roughly ten times the silicon area of an ASIC, while using very similar design techniques. Thus, these implementation techniques and relative cost metrics could carry over into the ASIC realm.

There are three common hardware implementation techniques considered. These are small microcoded datapaths; a pipelined, single or multiple round, C-slow[4] structure; and a fully unrolled datapath.

A microcoded datapath may be the most compact design, but often suffers from very poor bandwidth. It consists of a register file, a datapath of several functional units cus-

[1] A 4-input lookup table can realize any boolean of 4 inputs

[2] These are small, 8 address, 16b wide memories, which have two separate address and data ports. This allows two separate memory locations to be written or read in a single cycle.

[3] There is a current trend towards FPGAs with microcontrollers, such as the Triscent parts[8].

[4] Two paragraphs further defines C-slow. Be patient.

tomized to the application at hand, and a small program (usually contained in a small ROM) which controls the datapath. The problem with such implementations is that the aggregate bandwidth is usually very low and the design is unable to utilize the parallelism inherent in the algorithm.

A $C$-slow datapath implements a single round or group of rounds, separated into $C$ pipeline stages which operate on different blocks. This allows for considerably higher bandwidth then a single iterative round, as $C$ independent blocks can be processed through the pipeline. The number $C$ is usually chosen to match the desired clock rate. A $C$-slow pipeline can run at a high clock rate and adding more register stages can allow an even higher clock rate (and therefore higher bandwidth) without affecting the latency[5]. Furthermore, since more operations can be done in parallel, this technique may improve the overall latency when compared with a microcoded implementation.

For most algorithms, a $C$-slow, single round pipeline should require roughly the same area as a microcoded datapath or an unpipelined round, while offering a considerable improvement in bandwidth, as the functional units are more highly utilized. Thus, a $C$-slow technique should always be utilized unless such a design simply can not be implemented in the available area or the implementation fabric is flip-flop poor.

A fully unrolled datapath, where each round is separatly implemented in hardware, can be similarly pipelined to run at a high speed. This offers essentially no latency advantage over a $C$-slow datapath, but allows for the maximum bandwidth possible. The number of pipeline stages is chosen in a similar way to the C-slow implementations, to provide operation which matches a target clock rate. The area cost and available bandwidth of a full pipeline are a simple multiple of the area and bandwidth for a $C$-slow implementation, so unrolled pipelines are not considered in detail in this analysis.

In general, we will attempt to roughly estimate the number of pipeline stages which would be required to allow a Virtex implementation to run at a 50 MHz clock cycle. A typical, modern, midsized FPGA such as the Virtex XCV200 contains 5000 4-LUTs and 14 BlockRAMs, while a typical compact, low cost FPGA such as the Xilinx Spartan2[6] XC2S50 contains 1,700 LUTs and 8 BlockRAMs.

Though these implementation sketches are for a particular FPGA fabric, these comparisons should carry over[7] to other FPGAs and small ASICs.

# 4 Cryptographic core

The cost for the different implementation's cryptographic cores were estimated by summarizing the costs of their respective subcomponents.

Serpent is the best from a pure performance viewpoint in hardware, although Rijndael and Twofish are close to it in performance and area/performance. The significant problem with Serpent is there is a significant minimum size for the implementation to be effective. MARS is comparatively awkward, requiring both a relatively large amount of logic and a large amount of ROM for table lookups.

The other problem with Rijndael and Serpent is that separate pipelines are required for encryption and decryption. Having to implement separate pipelines for encryption and decryption doubles the area of an implementation if both operations are required.

A RC6 pipeline can be easily modified to perform both encryption and decryption by replacing the adders with adder/subtracters (a no cost or very low-cost transformation). The Feistel basis of MARS and Twofish allow a slightly tweaked pipeline to handle both rolls effectively.

[5] This technique has a limit of the setup and hold time of the flip flop, and the granularity at which different paths may require different latencies.

[6] A low cost revision of the Virtex

[7] Although with some caveats, usually dealing with local memories and the use of tristate buffers to implement wide muxes, which may not be present in other FPGA fabrics

| Algorithm | Implementation | Latency (cycles) | Bandwidth (blocks/cycle) | Size (4-LUTs) | Size (BlockRAM) |
|---|---|---|---|---|---|
| MARS | Microcoded datapath | 480 | 1/480 | 770 | 8 |
| | 6-Slow, single round | 190 | 1/16 | 1500 | 12 |
| RC6 | 5-Slow, single round | 102 | 1/20 | 1700 | 0 |
| | 8-Slow, folded round | 164 | 1/40 | 950 | 0 |
| Rijndael | 2-Slow, single round | 20 | 1/10 | 780 | 8 |
| Serpent | 8 slow, 8 round | 32 | 1/4 | 3800 | 0 |
| | single round | 32 | 1/32 | 1600 | 0 |
| Twofish | 3-Slow, single round | 50 | 1/16 | 1350 | 0 |
| | 4-Slow, folded round | 66 | 1/32 | 870 | 0 |

Figure 1: A comparison of the implementation costs for the various algorithms

A mixed Rijndael pipeline can share the S-boxes by separating the transformation from the S-box, which adds a small step and some area. However, this approach still requires a completely different column mixing step and therefore a fairly significant area cost to handle both encryption and decryption. Some implementations would probably just use separate pipelines, since depending on the implementation technology, the cost of the S-boxes may be dwarfed by the remaining costs.

Serpent can share almost no area between encryption and decryption, since it is dependent on inverse-sboxes and inverse-transformations for decryption. This essentially doubles the cost of a Serpent device which performs both encryption and decryption.

## 4.1 MARS

MARS is unfortunately comparatively costly to implement on small devices, as a microcoded datapath is more compact then a C-slow pipeline. There are also several comparatively expensive elements: variable rotations, the numerous, large S-box references, and the 32 bit multiplier. Since the multiplier and rotates are on the critical path and can not have their latencies hidden, a fast array multiplier and a barrel rotator are necessary to achieve good performance.

### 4.1.1 Microcoded datapath

A microcoded datapath would require 4 BlockRAMs for a 32 bit, 2 read, one write port register file for a scratchpad and subkey storage, another 4 BlockRAMs used as ROMs to store the S-Box, 32 LUTs for the XOR, 32 LUTs for the adder/subtracter, 160 LUTs for a barrel rotator, and 512 LUTs for an array multiplier[8]. Thus, this datapath requires roughly 8 BlockRAMs and 768 LUTs.

Assuming a single cycle latency for all operations but the multiplier, and assuming 3 cycles for the multiplier, it would take roughly 13 operations for each round of forward mixing, 18 for one round of the cryptographic core, and 12 rounds for the backwards mixing. Thus, it would require at least 480 cycles of latency for a single encryption. Furthermore, it is very difficult to run more than one or two blocks through such a datapath.

## 4.2 Full Round, C-slow

A $C$-slow pipeline is less compact on MARS when compared to other algorithms, due to the expense of various components and the 3 separate round types. The forward mixing would require 4 BlockRAMs for the S-box halves and 172 LUTs for the logic. The core would

[8] A booth encoded, shift and add multiplier could probably be constructed for only 64 to 128 LUTs, but would require 16 cycles/multiply instead of 3 cycles cycles

require roughly 4 BlockRAMs for the sbox, 64 LUTs to store the subkeys, 512 LUTs to compute R, 172 LUTs to compute M, and 172 LUTs to compute L, plus an additional 150 LUTs to compute B, C, and D, for a total of 1070 LUTs and 4 BlockRAMs for the core. The final mixing would require another 4 BlockRAMs and 172 LUTs for logic, for a total of 12 BlockRAMs and nearly 1500 LUTs. The number of 4-LUTs is reasonable, but the large number of S-box references are a considerable expense, making this implementation prohibitive on devices without local memories to use for the S-boxes.

In order to run the central core at a desired 50 MHz, it would probably be necessary to run it 6-slow[9], with the forward and backward mixings running 3 slow This would require 192 cycles of latency to encrypt a single block, but would produce one block every 16 clock cycles.

The biggest problem with MARS is the numerous references to the large S-Boxes. If a bandwidth-oriented implementation is desired, the number of S-Box references becomes very expensive. The 32 bit, modulo $2^{32}$ multiplier is expensive, but not prohibitively slow. Finally, the 2 variable rotations are moderately expensive operations. The biggest expense is the three different round types: although not a concern for a software implementor, it is a significant handicap for hardware designs.

## 4.3 RC6

RC6 uses operations which, while inexpensive in a modern microprocessor, are moderately expensive in hardware. A 32 bit, modulo $2^{32}$ multiplier require 512 LUTs, and a 32 bit rotator would require 160 LUTs to accomplish. However, there is a nice ability to trade off performance for area in this design.

### 4.3.1 Full round, C-slow

The most straightforward, compact implementation of RC6 is a single round, C-slow implementation. The initial and final keys are best stored in registers, while the remaining keys would fit in 128 LUTs. The MUXes on the start of the pipeline (to select between the input and the result from the previous round) require 128 LUTs, and the input and output whitening each require 64 LUTs.

The pipeline for the round itself would need 512 LUTs for each F function to perform the 32 bit multiplication. The variable rotations require 160 LUTs but can be combined with the XOR operation, and each of the subkey additions requires 32 LUTs. When added to the hardware required for muxing plus the initial and final adders, the total comes to roughly 1700 LUTs for the pipeline.

It should take 3 cycles to perform the F function, another cycle for the rotation, and a final cycle for the subkey addition, suggesting that a 5-slow pipeline would be sufficient. This would require 102 cycles latency to produce a result but would be able to produce a result every 20 cycles.

### 4.3.2 Compact, half-round, C-slow design

There are some tricks which can be used for a more compact RC6 design. Since both sides of a round are identical, the implementer could build a half-round, C-slow implementation which folds the two halves together. This roughly cuts the resource requirements and bandwidth in half, and adds three cycles of latency per round in order to exchange $t$ and $u$ and to perform the exchange at the end of each round, with an addititonal cycle of padding to implement a round in an even number of clock periods. In a case where bandwidth is as important as latency while resources are heavily constrained, this technique would be significantly prefered over a microcoded datapath.

[9] 3 cycles to compute R, 1 cycle to compute M, 1 to compute L, and 1 cycle to compute the new values of B, C, and D

The additional costs of such a datapath are one extra cycle for each swap and one cycle for padding, making the pipeline 8-slow and uping the latency to 164 cycles, and the bandwidth reduced to one block every 40 cycles. This allows the core to be almost cut in half, to 870 LUTs, with another 32 LUTs to store the remaining subkeys. Also, an additional 40 LUTs are required for various MUXes, and the subkey storage and whitening remain unchanged. Thus, the cost of such an implementation would be roughly 950 LUTs.

## 4.4 Rijndael

Rijndael's number of rounds depends on the key size. For this analysis both the block and key size are 128 bits. Rijndael has a high degree of parallelism, with very short operations and a small number of rounds, which makes it one of the fastest candidates for a hardware implementer.

The Mix-column operation of Rijndael would require 8 LUTs for the accumulation of each byte, with each multiplication probably reducable into 8 LUTs similar to the technique in [3]. Thus, the entire mix column for one 32-bit word would probably require on the order of 100 4-LUTs.

A round of Rijndael requires 8 BlockRAMs to store the S-boxes for the byte substitution[10], no area for the row shifting operation, 400 LUTs for the 4 column mixes, 128 LUTs for the key xors and the bypassing of the final column mix, 128 LUTs for the input subkey addition and pipeline MUXes, and 128 LUTs to store the subkeys.

The net result is probably 780 LUTs and 8 BlockRAMs for a single round implementation. With a critical path of 1 memory access, 3 LUTs for the column mixing, and one for the round key addition, a one or two cycle latency is reasonable for a round. With only 10 rounds of encryption, this results in an incredibly low 20 cycles of latency, with a block every 10 cycles.

Rijndael performs a greater number of rounds when used with a larger subkey. This would not affect the area required but would increase the latency and reduce the bandwidth. With 2 clock cycles for each round, it is straightforward to extrapolate the cost of a larger subkey.

## 4.5 Serpent

Serpent's operations, being very DES-like, map extremely well into hardware. The choice of 4 input, 4 output S-boxes allow each S-box to occupy only 4 4-LUTs, while XORs are very inexpensive, and constant rotations and permutations are free. However, although the algorithm is very fast, a considerable amount of area is required for the S-boxes which make Serpent surprisingly costly in hardware, even though its basic operations are very inexpensive.

### 4.5.1 Serpent 8-round, 8-slow

Due to the nature of Serpent's S-box use, the sweet spot for a serpent implementation is to unroll 8 rounds. The initial and final permutations require only wiring, not lookup tables, so the entire cost is in the encryption core.

A single round requires 128 LUTs for the key XORs, 128 LUTs to store the subkeys for the round, 128 LUTs for the S-boxes, and 160 LUTs for linear transformation, for a total of 544 LUTs for a single round. In a pipeline, a savings of 64 LUTs/round could be achieved by combining two of the key XORs with the linear transformation from the previous round, at the cost of some design complexity. For an 8 round pipeline, the total comes to 3800 LUTs for the entire pipeline.

[10] There is some wasted memory here due to the size of the BlockRAMs. Only half of the bits are actually used, which indicates that in a technology where the 8x8 S-boxes are directly implemented the area occupied would be smaller

Since each round consists only of bitwise operations and fixed rotations with a critical path of only 5 LUT evaluations, it should be pipelineable with only one cycle/round. It may even be possible to complete 1.5 to 2 rounds in a single cycle, reducing the latency further, since this critical path is so short. Thus, the 8 round pipeline would be run 8 slow, producing a result every 4 cycles, with a low latency of 32 cycles to encrypt a single block.

### 4.5.2 Serpent single round

A single round implementation would still need to implement all possible S-Boxes, a wide muxing step to combine the results would best be implemented with tristate buffers. Thus, 1024 LUTs would be required for the S-Boxes, 256 LUTs to store the round subkeys, 128 LUTs for the key XOR, and 160 for the linear transform. The resulting single-round implementation would require 1600 LUTs. This also introduces one more evaluation (the muxing of the S-boxes to select the correct one) into the critical path.

If pipelined at the same rate as the 8-round version, this would produce a result every 32 clock cycles, with an identical latency of 32 cycles. Since this only represents a 40% savings in area but an 8-fold reduction in bandwidth, this is not a beneficial tradeoff in most cases.

## 4.6 Twofish

Twofish works well in hardware without requiring memory to implement S-boxes. Though it is not the fastest or the most compact, it is reasonably small and has other advantages, including a nice area/performance tradeoff and the ability to perform encryption and decryption with a slightly modified pipeline.

The building block of Twofish, the $h$ function, maps reasonably well to FPGA logic. Each $q$ permutation requires 24 LUTs to implement, integrated with the S-box key XORing, for a total of 288 LUTs. The critical path is 12 LUT evaluations, short enough to expect to implement in a single cycle.

The MDS Galois matrix multiplication also maps very well. [3] shows how the multiplication by **0x5b** can be implemented in 8 LUTs, and the multiplication by **0xEF** requires 9 LUTs. It requires a further 8 LUTs to add each output together. The net result is that the matrix requires 135 LUTs to compute, with a critical path of 3 LUTs, allowing it to be combined with the PHT.

### 4.6.1 Twofish single round

A single round would require 846 LUTs for the two $h$ functions, another 64 LUTs for the PHT, 64 LUTs to store the subkeys, 64 LUTs for the subkey addition, and 64 for the subkey XORing. A final 256 LUTs are required for the whitening steps, resulting in roughly 1360 LUTs for the entire pipeline.

A reasonable expectation would be for this round to take 3 cycles, one for the S-boxes, one for the MDS and PHT, and one for the key addition and XOR[11]. Such a pipeline would take 48 cycles to encrypt a single block, producing a block every 16 cycles.

### 4.6.2 Twofish folded

Like RC6, the symmetries in Twofish allow the pipeline to be folded in half. This would require an additional cycle to do the PHT, because the MDS would need to be split out, as well as additional logic for the PHT operation. This would require 423 LUTs for the $h$ function, 64 LUTs for the PHT[12], 64 LUTs to store the subkeys, and 64 LUTs to perform

[11] Carries on FPGAs tend to propagate faster then the sum, but if it is necessary to develop a 3 stage pipeline, it might be best to place the pipeline in the middle of the carry of the PHT and key addition, so that the first cycle does the low 16 bits of the PHT and the key addition, and the second cycle does the high bits and the XOR operation

[12] for a 32 bit adder and the additional logic to shift or not and to select the proper input

| Algorithm | Implementation | Latency (cycles) | Bandwidth (subkey sets/cycle) | Size (4-LUTs) | Size (BlockRAM) |
|---|---|---|---|---|---|
| MARS | New microcoded datapath | 270 | 1/270 | 300 | 8 |
| | Existing datpath modified | 270 | 1/270 | 50 | 0 |
| RC6 | Specialized datapath | 264 | 1/264 | 290 | 0 |
| Rijndael | New specialized datapath | 36 | 1/36 | 128 | 2 |
| | Shared S-boxes | 36 | 1/36 | 160 | 0 |
| Serpent | 8 slow, 8 round | 32 | 1/4 | 2060 | 0 |
| | 2 slow, 2 round | 32 | 1/16 | 1500 | 0 |
| Twofish | Shared H-func | 20 | 1/20 | 512 | 0 |
| | Separate H-func | 4 | 1/4 | 1260 | 0 |

Figure 2: Comparative performance and cost of subkey generation

the feistel network XOR and to rotate the output if necessary. 128 LUTs would still be needed for each of the whitening steps. It would also require an additional cycle for the PHT, in order to delay the proper element.

Such a pipeline would require roughly 870 LUTs and would require 4 cycles to complete each block, increasing the latency to 64 cycles, and reducing the bandwidth to one block every 32 cycles.

# 5 Subkey generation

Although subkey generation is not always on the critical path, it is may be necessary to do the subkey expansion within the device, often as a microcoded datapath or customized logic. Some applications, like point-of-sale terminals, may rarely or ever need to change their keys, in which case subkey generation isn't a priority and can be performed external to the device.

Applications such as an encrypting packet router or disk controller may require changing subkeys on a packet-by-packet or block-by-block basis. In such applications, the key setup time and parallelism may prove to be the critical factor. An important consideration for hardware implementations is how agile the key scheduling is. Being able to pipeline subkey generation at the same rate as encryption allows subkeys to be generated concurrent to encryption.

Note, though, that Rijndael and Serpent allow concurrent keyscheduling only in the encryption direction, not for decryption. These ciphers require some additional buffering for the expanded subkeys for decryption, which would make decryption latency for a changed key to be different than the encryption latency for a changed key.

The ideal case, which only occurs in Twofish, is subkeys which can be generated independently. This allows encryption and decryption subkeys to be generated on the fly regardless of whether the data is being encrypted or decrypted. This is a great advantage for devices which need to encrypt and decrypt a large number of differently keyed blocks.

In general, the datapath will only be described for a keysize of 128 bytes, if there is a significant difference in the pipeline structure for different key sizes.

Both MARS and RC6 have considerably slower subkey generation when compared with the other candidates. Neither can be effectively pipelined or accelerated, and any attempt to simultaneously produce multiple subkeys for different initial keys requires duplication of the subkey-creating hardware.

Rijndael's subkey generation is considerably shorter and takes up a small amount of area.

Although it can not be pipelined, it is small enough to duplicate if subkeys are changed often. Creating the Serpent subkeys, on the other hand, favor a heavily pipelined design due to the comparatively high cost of all the S-boxes.

Twofish's key generation can share hardware with the encryption pipeline, if a low cost implementation is required. Alternatively, it may contain it's own copy of the S-box logic and generate the subkeys concurrently with encryption, essentially eliminating all the latency involved in subkey creation.

## 5.1 MARS

The MARS subkey generation is best implemented in a custom microcoded datapath. If such a datapath is used for encryption, the incremental cost of subkey generation is minor, just a fair amount of expanded code with all indexes recalculated. The only addition would be a logical structure to compute $M_n$ requiring some 100 LUTs to accomplish. If a microcoded datapath is not used, essentially the full microcoded datapath from the encryption description (sans multiplier), would be necessary, roughly 300 LUTs and 8 BlockRAMs, and roughly 270 cycles to generate the subkeys.

## 5.2 RC6

The RC6 subkey generation is probably best implemented with a custom datapath, using 2 BlockRAMs to store the subkeys during computation. Since the number of user key blocks is rather small, 32 LUTs used as a small RAM is sufficient. 2, 32 bit registers can store $A$ and $B$, with 32 LUTs for a dedicated adder to always compute $A + B$. The only additional logic to calculate $A$ is 2 adders, one to generate the initial value of the $S$ array, and the second to add the current value of the $S$ array to $A + B$, 64 LUTs in all. For updating $B$, this requires 160 LUTs for the rotation and 32 LUTs for another adder. Thus, the total datapath would occupy 290 LUTs.

The control logic for this structure consists only of a couple of counters and some simple state for the state machine, so it should not require significant resources.

It should be reasonable to update $A$ in 1 cycle as it only requires 3 additions or two additions plus a memory lookup, and a constant rotation. Similarly, $B$ should be computable in a single cycle as well. Thus, for 20 round RC6, this datapath requires 132 executions, for 264 cycles to generate the subkeys.

## 5.3 Rijndael

Rijndael's subkey generation is very compact. It can only produce four bytes per cycle as each word is dependent on the previous word, so an implementation which changes keys often would be still dominated by the latency of subkey generation.

Subkey generation requires 4 copies of the S-boxes in 2 BlockRAMs (either shared with the encryption pipeline or independent), enough buffering for 128b with a 128b key, 32 LUTs for the Rcon table, and 32 LUTs for the various XORs and selections. Since the buffering is dominant, the total would probably require 128 LUTs, as the flip flops end up dominating the cost. Each subkey word could be generated in a single cycle, requiring 38 cycles to generate all the subkeys.

## 5.4 Serpent

Just as the best Serpent implementation is an 8 round, 8 slow pipeline, the same holds for the subkey generation. Since the structure is very similar to the round itself, the same techniques can be used. It requires 64 LUTs to calculate the XORs for each of the 4 subkeys generated for each round, another 128 LUTs for the sbox substitution, and 32 LUTs for calculating the index, for 224 LUTs for each round. At 8 rounds, this comes to 1800 LUTs plus another 260 LUTs for the MUXes at the end of the pipeline, for a total of 2060 LUTs.

A more compact, 2 round design would still require 128 LUTs for the XORs, 1024 LUTs for the S-boxes with the results muxed by tristate buffers, 64 LUTs for the indexes, and 256 LUTs for the MUXes at the start of the pipeline. This would total to roughly 1500 LUTs, while still only requiring 32 cycles to generate a full set of subkeys. Although it might be possible to reuse the S-boxes from the encryption pipeline, the additional muxing would probably swamp most of the savings achieved by this reuse unless only a single-round serpent implementatino is used.

### 5.5 Twofish

The key generation in Twofish occurs in two parts, the first generating the two keys for the S-boxes and the second generating the round keys. The S-Box subkeys require a constant GF($2^8$) matrix multiplication. Using specialization, assuming an average of 8 LUTs/constant, 256 LUTs are required to generate the subterms, and another 96 LUTs are required to perform the XORs to generate the sbox subkeys.

For implementations where subkey generation is not in the critical path one can use the S-boxes from the encryption pipeline. The modifications to the existing pipeline would add 64 LUTs to mux the inputs into the S-boxes, 64 LUTs to mux the S-box subkeys between the encryption subkeys and the input keys, and another 32 LUTs to modify the PHT, for a total of 160 LUTs, a very small addition to the pipeline. This approach would require a total of 512 LUTs of datapath to generate the subkeys and 20 cycles to generate the complete set of subkeys.

A separate round subkey datapath could be implemented, requiring an additional copy of the 2 H-functions and a PHT (910 LUTs). This would require a total area of 1260 LUTs. This is comparable to the cost of the encryption pipeline, but has the advantage that subkeys can be generated on the fly concurrently with encryption, except for those subkeys required for the input and output whitening. This allows a hardware implementation of twofish to operate at almost maximum bandwidth while able to change subkeys on a block by block basis, and to shift between encryption and decryption at will. This has the effect of reducing the key setup time to only the 4 cycles needed to generate the input and output whitening subkeys.

## 6 Other implementations

Twofish and Serpent have hardware implementations[3] [5] reported in the literature which can be used to help calibrate the quality of our estimates. Both implementations used HDL synthesis, which hurts performance but does not significantly affect the area required.

The Twofish implementation in [3] requires roughly 900 Xilinx 4000 CLBs, or 1800 LUTs, with the hardware for the round itself requiring roughly 1400 LUTs. A pipelined version used 7 cycles/round, running at 35 MHz. Three considerations reduced their performance: the implementation overhead of VHDL, routing congestion and tools, and an older generation FPGA.

The area numbers for this implementation are very close to the estimates for Twofish, a very good sign. Also, the performance degradation present in the HDL version is expected. HDL synthesis[13] techniques tend to produce significantly lower performing designs[14], and the Xilinx 4000 series is also significantly slower then the current generation of Xilinx FPGAs.

[13] This is where the logic is described in a High level Description Language and then compiled to form the actual circuitry of the implementation, as opposed to a lower level approach of a had specified and hand placed datapath which is assumed in the estimates.

[14] There are two factors involved: HDL synthesis on a design like Twofish is usually constructed without detailed placements for the individual modules of the datapath, and the place-and-route tools are not intelligent about placing or reconstructing datapaths in designs.

The Serpent implementation [5] is unfortunately harder to use as a calibration. It required 18,000 LUTs at 37 MHz for a fully unrolled, pipelined (1 stage/round) version, 15,000 LUTs at 13 MHz for an unpipelined, 8 round version, and 11,000 LUTs at 15 MHz for a single round when implemented in a Virtex 1000. The performance numbers are very good, and although some improvement may be achieved by a manually layed-out design, the nature of serpent doesn't have heavy datapath regularity to exploit.

The single and eight round versions can not be used to calibrate the area estimates, as the design used flipflops and MUXes for subkey storage, instead of the luts-as-memory ability present in the Virtex. Furthermore, the single round implementation used MUXes instead of the internal tristate lines to mux the S-boxes, a serious inefficiency in the implementation.

The best mechanism for attempting to calibrate area is to quadruple the area estimate for an eight round version of serpent, as a first approximation. With 15,000 LUTs for the estimated area, and 18,000 LUTs for the HDL implementation, the comparison is pretty close. The additional area for the HDL version undoubtedly includes the logic for setting the subkeys and performing I/O, while the estimate in this paper only considers the cryptographic core.

# 7 Conclusions and Lessons Learned

Both Rijndael and Twofish are very amenable to hardware implementations. Rijndael is the fastest, with a great degree of parallelism and very quick operations, but area requirements increase substantially if encryption and decryption is required in the same device. Although Twofish is somewhat slower, there is an excellent degree of flexibility in subkey generation and in area/performance tradeoffs.

The numerous, large S-boxes are one of the features which greatly cripple MARS hardware implementations. Having to implement 9 large 32bit S-boxes to create a single $C$-slow pipeline impose a significant cost on any implementation. Also, the heterogeneous round types cause a significant area penalty when compared to other implementations. The use of both S-boxes and multiplication further compounds the cost, requiring both considerable storage and considerable logic to implement.

The subkey generation for both MARS and RC6 have serial steps which require all subkeys to be modified several times. This causes subkey generation to be very slow in hardware, a significant defect when dealing with applications which require rapidly changing subkeys.

Serpent ends up being surprisingly awkward, mostly due to the large number of S-boxes required. It takes 1024 LUTs just to store all the S-boxes. Although the performance is excellent, the bandwidth quickly drops for smaller implementations and the area/performance suffers greatly.

Similarly, two operations which are cheap software, multiplication and rotation, end up being comparatively expensive in hardware. A multiplier occupies much more logic then an addition in hardware, and large multiplier are much costlier[15]. Similarly, variable rotations are much more expensive in hardware when compared to constant rotations, XORs, or additions.

Independently generated subkeys such as those in Twofish offer a great benefit for some applications, as this allows almost complete hiding of the subkey generation time. This property allows a hardware implementation to almost completely overlap subkey generation with encryption and can remove the need for any expanded subkey storage.

[15] As an example, a $32x32$ modulo $2^{32}$ multiplier is four times the area of a $16x16$ modulo $2^{16}$ multiplier.

# 8 Acknowledgments

Many thanks to David Wagner for explaining the design decisions and operations of various aspects of the ciphers and to Eylon Caspi for his capable editing.

# References

[1] Anderson, Biham, and Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard", *http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Serpent/Serpent.pdf*

[2] Burnwick *et al*, "The MARS encryption algorithm", *http://csrc.nist.gov/encryption/aes/round2/AESAlgs/MARS/mars-int.pdf*

[3] Chodowiec and Gaj, "Implementation of the Twofish Cypher Using FPGA Devices", George Mason University Techinical Report, *http://www.counterpane.com/twofish-fpga.html*

[4] Daemen and Rijmen, "AES Proposal: Rijndael", *http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Rijndael/Rijndael.pdf*

[5] Elbirt and Parr, "An FPGA Implementation and Performance Evaluation of the Serpent Block Ciphen", in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, February, 2000.

[6] Rivest, Robshaw, Sidney, and Yin, "The RC6 Block Cypher", *http://csrc.nist.gov/encryption/aes/round2/AESAlgs/RC6/cipher.pdf*

[7] Schneier *et al*, "Twofish: A 128-Bit Block Cypher", *http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Twofish/Twofish.pdf*

[8] Triscend Inc, "Triscend E5 Configurable System-on-Chip Family", *http://www.triscend.com/products/dse5csoc.pdf*

[9] Xilinx Inc, "Virtex 2.5V Field Programmable Gate Arrays", *http://www.xilinx.com/partinfo/ds003.pdf*