# Blowfish & IDEA in Silicon

David Honig
honig@alum.mit.edu

**Keywords**: *Encryption Hardware; Blowfish Encryption; IDEA Encryption; ASIC Implementation of Software-Oriented Block Ciphers;*

**Abstract**

*We describe designs for circuits that compute two modern software-optimized encryption algorithms, Blowfish and IDEA. For each algorithm we discuss the complexity of these algorithms in terms of Verilog RTL source code, standard cell and PLD gate-equivalents, and RAM and ROM consumption. We describe the critical paths and sequencing and review various architectural tradeoffs and design tricks we found noteworthy.*

REV 18 Aug 00
for netpub
(missing refs)

# Blowfish & IDEA in Silicon

Contents

# Introduction

**History**

DES [] was useful 20+ years ago, when hardware was expensive. With cheap, fast CPUs came the development of software- and processor- optimized block cipher algorithms, such as Blowfish [] and IDEA (tm) []. For instance, Blowfish was optimized for (or designed around) the Pentium's 4Kbyte cache; IDEA uses 16-bit integer multiplies and 16 bit registers which were common in CPUs at the time the algorithm was designed. In contrast, DES is extremely efficient in hardware and inefficient on generic CPUs.

But bandwidth doubled twice as fast as Moore’s law, and CPUs couldn’t keep up, especially if they were to do anything else at the same time. Although great things can be done with reconfigurable logic, DSPs, etc., the most efficient implementation of ciphers is a dedicated ASIC.

**Motivation**

Hardware is good for encryption for two reasons. First of course is performance. More specific to security, though, is the immutability of hardware, vs. software on networked machines.

Block cipher algorithms don’t change over time, and they are relatively simple. Thus they are suitable for putting in an ASIC. Higher-level protocols tend to evolve over time and to be too complex for direct implementation, preferring programmable CPUs or even reconfigurable logic.

The block ciphers described here were designed to be chained. Since we want to pipeline the flow of blocks through this chain, we want our implementations of Blowfish and IDEA to use the same number of clocks per block. It turns out that this leaves IDEA with a slower critical path than Blowfish.

## Caveats & Colophon

These designs have *not* been prototyped on a PLD or fabbed when this was written. The Verilog *has* been extensively simulated and synthesized to both CPLDs and commercial standard cell library. Our methods and tools are acknowledged below.

The Verilog simulations used Cadence *Verilog XL* vers. 2.2
The synthesis used either Synplicity's *Synplify* vers 2.5 and its Altera APEX 20K 400 -1 characterizations, or Synopsys' *Design Compiler* vers 1998.08 and a proprietary , but not unusual commercial .22 micron CMOS standard cell library.

Gates were estimated like so: Altera "LC's" were taken to be equivalent to 10 gates each. Design Compiler's "grids" are equivalent to three per 2-input NAND gate. Routing area is *not* included in Design Compiler estimates.

Timing was estimated like so: Synplify returns a clock-frequency estimate for the targeted part. Design Compiler returns the slack for a given clock []; a 1.5 nsec input delay was included in its estimates.

The source RTL and design flow would eventually be published (not, however, for free use) to facilitate users' verification of the hardware's integrity ---no trojan gates, no leaky channels, no sneaky bits.

This paper assumes a familiarity with both Blowfish and IDEA; for space reasons, we have omitted a decent discussion of their workings, except for the architecturally significant parts.

All trademarks acknowledged.

## The Blowfish Core

**Algorithm**

Blowfish[] is a 16 round Feistel cipher operating on 64 bit blocks. The key can be up to 448 bits long.

Two things stand out about Blowfish from an architectural viewpoint. First, Blowfish's use of four large S-tables requires embedded RAM. Second, it has a lengthy recursive key schedule. Since we expect to have to quickly swap a few session contexts, the lengthy key schedule could be a problem, so we add Dump/Restore functionality to cache the cipher's internal state without going through the lengthy re-initialization.

[FIG Round Structure]

**Core Description**

The core executes the following functions:

SETKEY This initiates the loading of a user key on the keybus. Blowfish’s user key is variable length, so the key length is supplied to the core followed by the requisite number of bytes. The core uses some 26,000 clocks before it signals that it is done expanding the key. The key is sequenced on the 8-bit keybus which was kept separate from the data bus.

ENCRYPT This initiates the encryption of data present on the input bus. The current internal state of the cipher is used. The ciphertext is present on the output bus 34 clocks later.

DECRYPT Exactly like encryption; there is no time required to switch between encryption and decryption, which would be useful for symmetrically keyed IPSec or disk encryption.

DUMP This sequences the core’s internal state on the output bus. Blowfish’s internal state is four S tables of 256 words of 32 bits each, plus a P table of 18 words of 32 bits each. That’s 1042 words or 4168 bytes. The output bus is 64 bits wide, so we can do this in just over 500 clocks, a big savings over the 26,000 clocks to initialize with a new key.

RESTORE This is the inverse of the DUMP operation.

CLEAR This is a fast zeroization of the core’s internal state.

**Pins**

We use a separate byte-wide bus for the keying.

| | |
|---|---|
| Din[63:0] | Data input bus |
| Dout[63:0] | Data output bus |
| Clk, Reset_n | System signals |
| Clear | Input, initiates zeroization |
| Encmode | Input, Encrypt/Decrypt signal |
| Go | Input, Initiate crypt operation on Din |
| Done | Output, signals valid data on Dout, ready state |
| Loadkey | Input, initiates key loading (first byte is key length in 32 bit words) |
| Keybus[7:0] | Input, key data supplied here, 8 bits at a time |
| Restore | Input, initiates loading of data on Din into internal memories |
| Dump | Input, initiates sequencing of internal memories on Dout |

[FIG Pinout]

## Internal Structure

Blowfish is implemented as a "codec" module which performs the encryption rounds, and a "controller" module which performs initialization. The codec has four big (256x32) RAMs and a smaller (18x32) RAM. During initialization, the controller updates the codec's RAMs. When loading a key, the controller reads from a ~4 Kbyte ROM which contains digits of Pi, used for initializing the codec's internal state when a key is loaded.

[FIG Internal structure]

## Issues

### Key Scheduling

Blowfish has a lengthy, recursive initialization sequence. This procedure expands the key into 1042 (1024 + 18) words of 32 bits each. This expanded key is the internal state of the cipher engine, the S and P tables, used for encryption and decryption.

Blowfish's key expansion runs the Blowfish algorithm 521 times (1042 / 2) to securely mix a fixed, entropic, initial internal state (the first 1042x32 bits of Pi - 3.0) with the input key. This recursive key expansion takes time, but it strengthens the algorithm by making exhaustive key search more expensive. For applications with frequent context switching, it forces the dump/restore operations.

### ROM Usage

Blowfish requires the first 1042 x 32 bits of Pi to load into its internal RAMs at the start of key scheduling. This is stored in ROM (11 address lines, 32 data lines).

RAM usage

Blowfish is notable in that it uses large substitution tables. Blowfish uses four S tables, each containing 256 entries of 32 bits. (4x8x32 bits) There is also a P table, 18 deep and also 32 bits wide (18x32 bits). All this state is derived from the user key which is up to 448 bits long.

Critical Path

The critical path in our Blowfish design is the F() function, which includes 3 bitwise XORs and two 32-bit adders.

### Code

The Verilog consists of about 400 lines for the codec and 500 for the controller. Each was developed and tested as a separate module, using highly instrumented, reference C code as a "gold standard". There is also a top-level file and a file containing the Pi initialization data in a ROM.

Even though the controller's source code was larger, after synthesis the controller was only 1/3 of the gates, with 2/3 being consumed by the codec.

### Optimizations

The first and last P-table entries are mirrored in registers to shave 1 clock at the beginning and 1 at the end of the rounds.

The "Save/Dump" logic was moved to the "codec" because its faster than if the "controller" managed them. The controller still needs to read and write to the codec's memory (at a slower rate) during key scheduling.

### Performance

Clocks/Block: 34. Here's how. Blowfish has 16 rounds. These are computed in 2 clocks each, one for the S-table lookup, one for the F() function. We also use one extra clock at the beginning and end of the rounds for Blowfish's prewhitening and postwhitening steps.

Encrypt/Decrypt switching time: 0. When an cipher operation is initiated, an ENCRYPT/DECRYPT line indicates the mode; the difference for Blowfish is the order in which the P-table entries are used.

## Blowfish Resource Summary

| | |
|---|---|
| Gates | 13 Kgates |
| RAM | ~33 Kbits |
| ROM | ~33 Kbits |

Altera 1200 LCs, 40 Mhz

## Future Work

In retrospect, these alternative designs are interesting.

<u>Paging</u>
Instead of having a 4 Kbyte internal memory dumped to an externally managed session-cache, Blowfish could use deeper internal memories and switch contexts by addressing. This is reasonable so long as the total memory-read time remains less than the critical path.

<u>Pipelined S-Table Lookup</u>
The lookup time on 256 x 32 SRAMs may be short enough so that the S-table need not be registered, but part of the addition-xor-addition-xor Feistel round computation.
The choice is between 18 (16+2) faster clocks, with S-table lookup alternating with the F() function, and 10 (8+2) slower clocks that include the S-table lookup and the F() function.

## The IDEA Core

IDEA has been used as a benchmark for crypto hardware, in part because it does not require much memory. The following performance figures have been published[]:

| | | |
|---|---|---|
| Pentium-Pro, reference code | 16 Mbps | 180 Mhz |
| Buldas, Poldre -.EE | 32 | 25 Mhz |
| 1 u; also supports RSA | | |
| Bonneburg et al. | 44 Mbps | 25 Mhz |
| VINCI 1993 -Zurich | 177 Mbps | 25 Mhz |
| 107mm^2 in 1.2 u 250K trans (~60K gates?) | | |
| 8-stage pipeline, full custom multipliers | | |
| Ascom | 100/300 Mbps | 40 Mhz |
| .25u; 35K gates; 3 pipelines; 4 mults | | |
| Mencer et al | 528 Mbps | 33 Mhz |
| fully pipelined 4 Xilinx FPGAs | | |

These designs differ in the number of multipliers implemented and their pipelining, number of clocks per block, use of subkey memory for cacheing state, whether they used custom vs. standard cell design, and their use of BIST. We discuss our design below.

### Algorithm

IDEA []….. DESCRIPT

IDEA is architecturally noteworthy in that it lacks S-tables, using modular multiplication (hereafter "modmults") to emulate them instead. It uses four modmults in each of eight rounds. The modmults are implemented using a "hi-lo" algorithm that uses a 16x16 multipliers, adder-subtractors, and logic. IDEA has a simple key schedule, but (unlike Blowfish) the decryption subkeys are different (decryption subkeys are the multiplicative inverses of the encryption subkeys) and must be computed from the latter. In software this is done with Euclid's algorithm, but we use a ROM. This has the side benefit of eliminating timing attacks on the iterative Euclid algorithm. Much like Blowfish's ROM, IDEA's ROM is only used when a key is loaded. Unlike Blowfish's ROM, spying on IDEA's ROM usage reveals its internal state –a good reason to integrate the ROM on chip.

[FIG Round Structure]

## Core Description

The IDEA core has exactly the same functions and pins as the Blowfish core. The sequencing of signals differs, e.g., because Blowfish's key schedule takes longer than IDEA's, and because IDEA has much less internal state to Save or Dump.

## Internal Structure

Like Blowfish, the IDEA core has a "codec" module which performs the round functions and a "controller" module which performs key expansion. Like Blowfish, the codec contains a number of RAMs to hold the subkeys. Unlike Blowfish (and much like DES) we could have computed these round subkeys "on the fly". But IDEA decryption requires further computations on the encryption subkeys, so the codec-with-SRAM plus controller architecture was adopted. (Throughout this project, we assumed building embedded SRAM was not a problem.)

The codec contains two implementations of the hi-lo function. Synthesis tools can be sensitive to how you write the modmult function, and are not that smart. This was important because the modmults are the critical path. Recent PLDs with support for fast multiplication (optimized carry chains) can do quite well compared to standard cells. And many IDEA designs have custom (hand tweaked) multipliers. We used the generic multipliers generated by the synthesis tools.

## Issues

### Mod Inv

IDEA decryption requires the computation of the modular inverses of the encryption subkeys. In software, this is done with Euclid's recursive algorithm. In our implementation, we use a ROM containing 65536 words of 16 bits each.
This has the benefit of resisting timing attacks possible with the algorithm used in software implementations []; but it may not be worth it.

### Hi Lo ModMult Algorithm Critical Path

The critical path in the IDEA algorithm is the modmult operation, multiplication of two unsigned 16 bit integers modulo 65537. This is efficiently performed using the "hi-lo" algorithm. In this algorithm, a 16x16->32 bit unsigned multiplication is performed. Then the upper and lower halves are subtracted, and depending on the sign, this difference may be incremented by one. (If the high or low halves are zero, other computations with shorter critical paths are used.)

### RAM

6 x 18x16bit

ROM

1 Mbit, (16b addr, 16b data)

## Optimizations

We explored different ways of coding the modmults and how the synthesis tools interpreted them. In retrospect we might have looked into pipelining the modmults and reconciling the altered clocks per block with a more pipelined Blowfish design, to keep the same clocks per block for each.

The use of a ROM instead of computation was a design simplification.

## Code

The size and distribution of RTL source for IDEA is similar to that for Blowfish. IDEA's controller is about 550 lines, and the codec about 400. And like Blowfish, when synthesized, 2/3 of the logic is devoted to the codec.

## Performance

Clocks/Block: 34. Here's how. IDEA has 8 rounds. These are computed in 4 clocks each. One of those phases uses both modmult instances; two others use just one, but can't be pipelined because of dependancies. We also use one extra clock at the beginning and end of the rounds for mixing in additional subkeys.

Encrypt/Decrypt switching time: 0. This is because both the encryption subkeys and the decryption subkeys derived from them are computed when the key is loaded. Switching between encryption and decryption is a matter of setting an address line into the RAM banks.

## IDEA Resources

Altera speed 23.1 Mhz, 2313 LCs

Gates 26 Kgates
RAM 2 Kbit
ROM 1 Mbit

## Future Work

Computing subkeys on the fly
We don't need memory for encryption in IDEA, we could have generated the subkeys on the fly. But you have to store your decryption subkeys somewhere.

Pipelined Idmul
The lengthy critical path computation can be pipelined, requiring more clocks, but the clocks can be faster.

Whether a long combinatorial path should be pipelined is not obvious: sometimes doing more work in fewer, slower cycles pays off. A recent FPGA Twofish design study [] found that a slow 10 Mhz design worked twice as well (~80 vs. 40 Mbps) as a pipelined design with a thrice faster clock but seven times as many clocks per block (16 vs. 112). In our case, we wanted to match IDEA to Blowfish and we weren't trying to design the fastest IDEA one could build. Area concerns constrained the number of multipliers we included, for instance.

Logic instead of ROM Instead of using a ROM to hold 64K multiplicative inverses, we could use the iterative algorithm which is used in software. Without padding, this might yield a timing attack ---the number of clocks taken to complete leaking (albeit very little) about the key. This might end up saving chip real estate.

## Chip Testing Notes

BIST may not be appropriate for a crypto chip where you don't want side channels or visible internal registers. Debug-mode pins are a vulnerability.

Crypto has the useful property that a single bit change 'avalanches' to affect other bits in successive rounds. Ie, make a mistake in the design, or (later) have a faulty transistor, and your answer looks nothing like its supposed to.

The RAM tables in Blowfish are accessed 16 times per block; encrypting a few hundred different blocks will likely test every cell, and by feeding the output into the input repeatedly, you need only compare the final result against the known correct answer.

## Conclusions

We have presented designs for Blowfish and IDEA encryption cores. We use an architecture using Codecs, which perform the round functions to effect ciphering operations, and Controllers, which perform subkey expansion when a key is loaded. The Codecs require several SRAMs; the Controllers read ROMs during initialization. Both cores share the same pinouts and operations, can switch between encryption and decryption in no time. They also can save and restore internal state to support session-context cacheing.

Blowfish is about 13K gates of logic, with five wide SRAMS totalling 33 Kbits. At 6 transistors per SRAM cell and 4 per gate, the RAM is about 3 times the size of the logic. The 33 Kbit ROM would not be so large, as ROMs are denser.

IDEA is about 23K gates, with less than 2 Kbits of RAM. The megabit of ROM might be replaced with logic to save space.

We targeted Altera parts for two reasons. First, Blowfish clearly needed blocks of RAM, and at the time (ca. 1998) Altera had these. Second, in-house experience favored those devices at the time. The latest (ca. 2000) Altera and Xilinx parts have plenty of gates and RAM.

These designs produce approximately 2 output bits per clock (actually 64/34), so at 80 Mhz, e.g., on an ASIC, they could drive an OC-3 link.

One could achieve arbitrarily scalable encryption rates with these cores without using faster clocks. Instead, you use *parallel counter mode*. You replicate N cipher cores, key them identically, and drive each of them with their own increment-by-N counter. The counters are initialized with some IV, IV+1, IV+2, … IV+N-1. Their outputs would be harvested (by sequencing their tri-stated outputs onto a common bus), to be xored with the plaintext. This has the advantage of not requiring feedback (yet not being succeptible to codebook attacks), being arbitrarily linearly scalable, and operates *transparently* as a single block cipher (in counter mode) as seen from the outside. In contrast, an interleaved CBC architecture requires specifying the interleave factor, and has serious inefficiencies if the hardware on either end of a link have different interleave factors.

[Fig ]

Although PLDs can do extremely well [], ASICs will always be cheaper and lower power. In this paper I have demonstrated that encryption algorithms designed for general purpose microprocessors can be efficiently implemented in fixed circuits. Since encryption algorithms don't change (cf. higher level protocols), they are an appropriate target for carving in stone. This is also preferable from a security perspective, where an alpha particle or trojan software can ruin more than your day.

## References

DES
FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION 1981 GUIDELINES FOR IMPLEMENTING AND USING THE NBS DATA ENCRYPTION STANDARD

USPatent 3798359 : BLOCK CIPHER CRYPTOGRAPHIC SYSTEM
Feistel; Horst, Mar. 19, 1974

Blowfish
Description of a New Variable-Length Key 64-bit Block Cipher (Blowfish) B Schneier

IDEA
*http://www.ascom.ch/infosec/idea/coprocessor.html*

*A VLSI implementation of RSA and IDEA encryption engine.*
Ahto Buldas, Jüri Põldre

*A 177 Mbit/s VLSI Implementation of the International Data Encryption Algorithm*
R. Zimmermann, A. Curiger, H. Bonnenberg, H. Kaeslin, N. Felber, and W. Fichtner
1993 Swiss Federal Institute of Technology Zurich

A proposal for a new block encryption standard X. Lai J.L. Massey Advn. in Cryptology Eurocrypt 90 389-404, 1991

The following taken from Mosanya et al., Crytobooster: a reconfigurable and Modular cryptographic coprocessor, CHES 1999.

Bonnenberg et al, VLSI implementation of a new block cipher. Proc Int Conf Comp Design: VLSI in Computer and Processors p510-513 Wash 1991 IEEE Comp Sci Press

Mencer et al Hardware software tri-design of encryption for mobile communication units. Proc Int Conf Acoustics, Speech, and Signal Processing Seattle May 1998

CHES 99

P Chodowiec, K Gaj FPGA implementation of Twofish GMU Elec & Comp Sci TR Jul 1999

## Figures

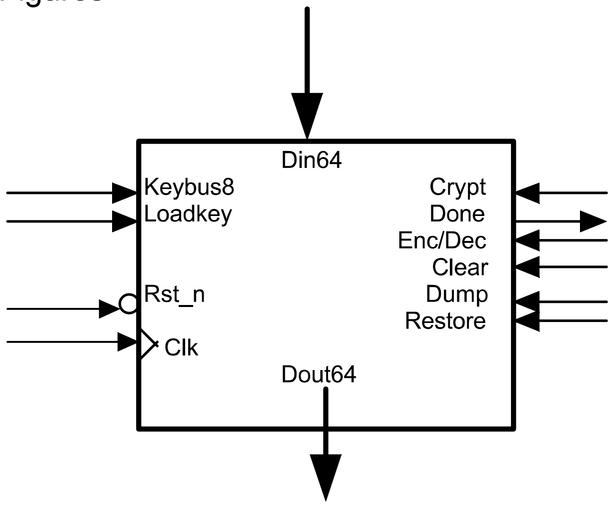

**Figure 1 Pinouts for Blowfish, IDEA.**

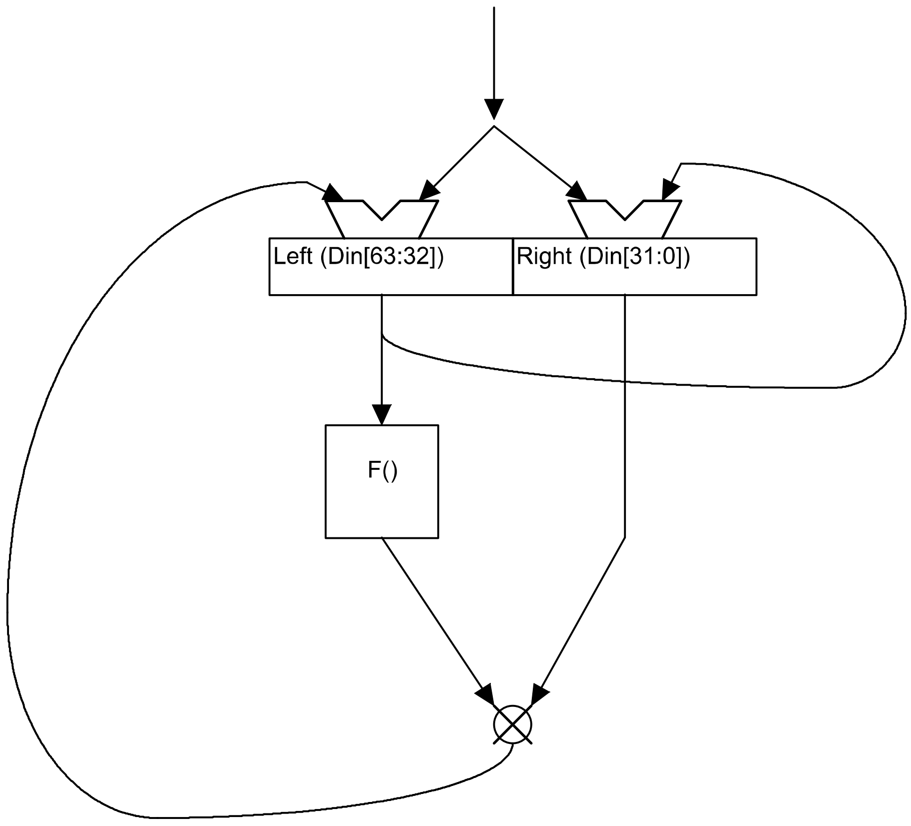

**Figure 2 Basic Structure of a Feistel Cipher**

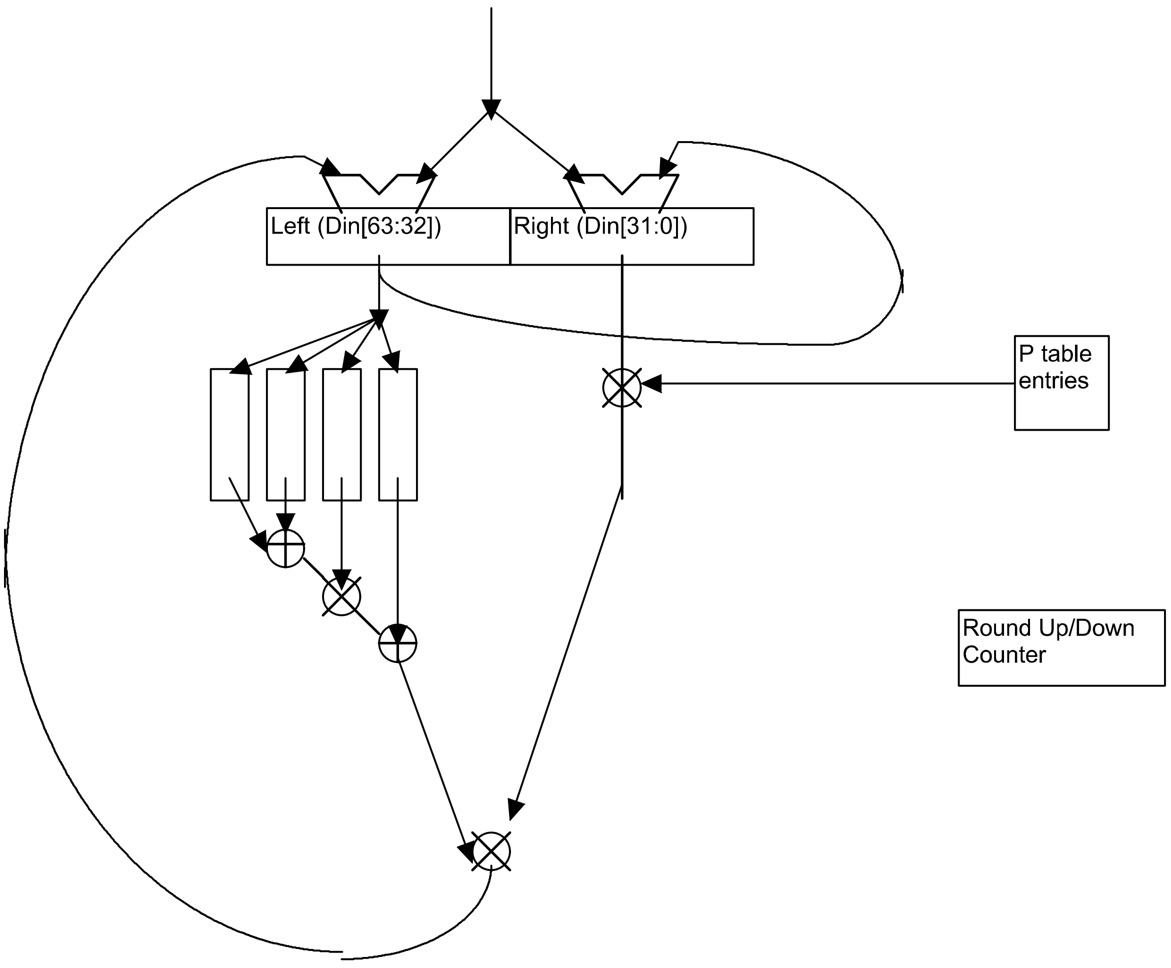

**Figure 3 Blowfish F() showing S-tables, adders and xors.**

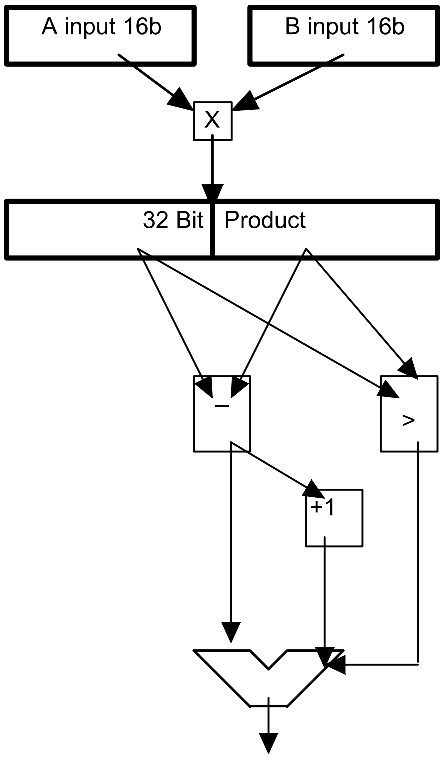

**Figure 4 IDEA hi-lo circuit for modmult, the critical path**