A CPLD Coprocessor for Embedded Cryptography

R. W. Ward, Dr. T. C. A. Molteno

Physics Department
University of Otago
Box 56, Dunedin, New Zealand

Email: tim@physics.otago.ac.nz

Abstract: Microcontrollers and CPLDs (Complex Programmable Logic Devices) are both low power components commonly found in embedded devices. We consider how a microcontroller and CPLD can work together to perform encryption and decryption with better power vs. performance characteristics than either device could manage alone.

Keywords: Microcontroller, PIC, CPLD, Coprocessor, Rijndael, AES, Encrpytion

Citing this work: Ward R.W., Molteno, T.C.A. "A CPLD co-proocessor for embedded cryptography", Proceedings of ENZCon 2003. Hamilton, New Zealand (2003).

1. INTRODUCTION

The Rijndael encryption algorithm [1], also known as the Advanced Encryption Standard (AES) is designed to be efficient to implement in both hardware and software. A CPLD (Complex Programmable Logic Device) is designed to implement small amounts of logic, although typically not enough to perform the full Rijndael algorithm.

A possible application for an implemenation of embedded Rijndael would be in a handheld device for transmitting audio information, either digitially (such as in a cell-phone), or requiring a modem at each end for tranmission over an analogue link such as a phone line. A related idea would be to use this for secure modem communication to transmit data.

We interface a microcontroller with a CPLD to perform Rijndael encryption and decryption using the CPLD as a coprocessor for the microcontroller. This configuration gives improved throughput/power characteristics over using a microcontroller alone. Microcontrollers and CPLDs are both relatively low power devices, so such an arrangement could be used for encryption and decryption in an embedded device where power consumption is an issue.

We compare our implementation (see Figure 1) with a software only implementation on a microcontroller, and a full implementation in hardware using an FPGA (Field Programmable Gate Array).

2. RIJNDAEL ENCRYPTION

Rijndael encryption is a form of symmetric encryption, in that it requires both the sender and receiver to have the same secret key. For encryption, the input (known as plaintext) is broken up into blocks with a particular bit width, and each block is encrypted using the key to produce an encrypted block of the same length (known as the ciphertext) See Figure 2 for an overview of this. This process is reversed for decryption.

The Rijndael algorithm is defined for key lengths of 128, 192 and 256 bits, and block sizes of 128, 192 and 256 bits. AES only allows block sizes of 128 bits. In this paper we consider the common case of 128 bit keys with 128 bit blocks.

3. DESIGN GOALS AND APPLICATIONS

Because we are looking at Rijndael for embedded devices, we consider:

- The typical speeds used for telecommunications are 14.4, 28.8, 57.6, 128 kilobits/s.
- Many applications will send and receive data simultaneously, so we our design must handle simultaneous encryption and decryption at the given throughput.
- Such a device is likely to be used in an environment where some information is lost in transmission, hence we only consider the non feedback mode (ECB [2]) for encryption.

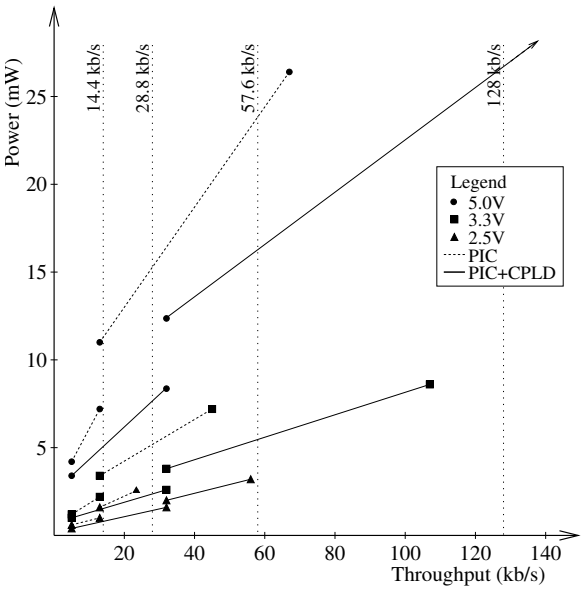

Figure 1. Power vs. Performance. The performance in kb/s, both encrypting and decrypting at that throughput. The power is the estimated power consumption of the PIC (Section 4) or PIC+CPLD (Section 5) system assuming that the clock is set as low as possible for the required performance, and no other operations are being performed. The discontinuities at 13.5 kb/s and 32 kb/s are where the clock speed reaches 4 MHz and the PIC has to be in HS rather than XT mode. For the CPLD, a Coolrunner II is assumed, except for at 5V where a Coolrunner XPLA3 is used.

3.1 Efficiency Considerations

3.1.1 Throughput. We calculate throughput by dividing the clock frequency of the microcontroller by the number of clock cycles required to perform the task. Note that in order to not be vulnerable to timing attacks, the throughput of the algoritm must be independent of the input values - it must take a fixed number of cycles. All of our implementations do this.

3.1.2 Power. For power use, we will use milliwatts, as calculated by typical current (according to datasheets) multiplied by voltage.

3.1.3 Utilization. For any practical application, components are are likely to be also serving other functions, so the utilization of components are worth noting.

3.2 Tools

Modern devices commonly contain a microcontroller to handle control functions, and the custom logic is commonly programmed into a Programmable Logic Device, typically a CPLD. This makes these devices attractive for adding extra functionality, as it may not require any extra hardware, or increased capacity

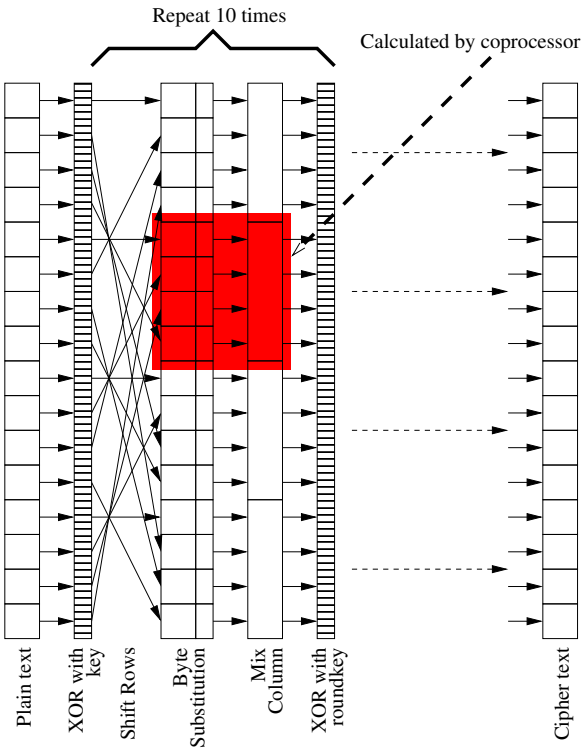

Figure 2. Overview of the Rijndael Encryption Algorithm. The portion surrounded by a the grey box will be performed by the coprocessor

microcontroller or CPLD.

3.2.1 Microcontroller. A microcontroller is a small processor designed for embedded devices. They typically run slowly, use relatively little power, have good I/O support and some built in memory. Often they have very limited memory capacity. Examples of Microcontrollers include the Motorola 68HC11, Microchip's PIC[1] family and the Intel 8051.

We will note the utilization of: program memory used; data memory used; execution time; I/O ports used.

3.2.2 CPLD. A CPLD is designed to provide a limited amount of configurable logic and to provide means to convert between different voltage standards. They are non-volitile, and operate at very low power, which makes them very suitable for embedded applications. Examples include Coolrunner XPLA3 and Coolrunner II CLPDs from Xilinx[2] or the MAX 7000 range from Altera.

We will note the utilization of amount of internal logic used. CPLDs have such a large number of I/O ports (typically 100 or more for the sizes of CPLD that we are considering) that I/O resource constraints are not likely to be a major consideration.

3.2.3 FPGA. An FPGA (Field Programmable Gate Array) is a large configurable array of gates. The

[1] www.microchip.com
[2] www.xilinx.com

provide much more capacity than CPLDs, but are volatile (hence requiring a means to program them on power on), and are not designed for power efficiency. There are typically either used for prototyping, or for larger devices.

4. MICROCONTROLLER IMPLEMENTATION

4.1 Microcontroller Choice

A PIC is a good choice for a microcontroller, as they are very low power, highly configurable and relatively cheap. There is a large range of PICs to choose from. We use a PIC16F870, as it is cheap (about NZD13), has sufficient resources to implement Rijndael encryption on (having 2K words of program memory, and 128 bytes of user registers), an 8-bit port available (which we will see is very important for coprocessor interfacing) and also has A/D for future expansion. The PIC16LF872 is upwardly compatible with the PIC16F870 and also operates at lower voltages.

4.2 Implementation details

4.2.1 Key calculation. We need to have the 16 byte key (see Figure 2) stored either somewhere non-volatile: either

- the 64 byte EEPROM
- a table in program memory
- supplied externally

As we have been prototyping, we store the key in RAM supply it externally.

Because of the very limited amout of RAM on a PIC, we cannot store the expanded key that the Rijndael algorithm uses, so have to calculate the key on the fly. Decrpyption uses the expanded key in reverse order, so rather than expanding the key each time before we start, we store the last 16 bytes of the key and work backwards to invert the key expansion algorithm [1].

4.2.2 Byte Substitution. For both byte substitution (see Figure 2) and inverse byte substitution, lookup tables are a clear choice on any microprocessor with sufficient memory. Due to the tables being stored in the program memory, a lookup takes 6 instruction cycles.

4.2.3 Inverse Mix Column transformation. In the forward mix column (see Figure 2), we want to calculate $b_0 = 02 \cdot a_0 + 03 \cdot a_1 + 01 \cdot a_2 + 01 \cdot a_3$, where the constants are hexadecimal representations of elements in $GF(2^8)$.

This can be arranged as $b_0 = a_0 + 02 \cdot (a_0 + a_1) + c$ where $c = a_0 + a_1 + a_2 + a_3$. b_1, b_2 and b_3 are done similarly without needing to recalculate c. Multiplication by 02 is cheap, being a rotate left (free) followed by 3 XOR operations. Pseudocode is given in [1].

Similarly, when for the inverse mix column, we want to calculate $b_0 = 0\text{E} \cdot a_0 + 09 \cdot a_1 + 0\text{D} \cdot a_2 + 0\text{B} \cdot a_3$. Let $c = a_0 + a_1 + a_2 + a_3$ and $k = 09 \cdot c = 02 \cdot (02 \cdot (02 \cdot (c))) \cdot c$.

Therefore $b_0 = a_0 + 02 \cdot (02 \cdot (a_0 + a_1) + a_0 + a_1) + c$.

b_1, b_2 and b_3 are done similarly without needing to recalculate c or k. These are both relatively efficient operations in PIC assembly.

Note that if we had more RAM or ROM, we could store several 256-byte lookup tables that store pre-calculated multiples of values in $GF(2^8)$ (combined with `ByteSub` or `InvByteSub`) as described in [5], which is somewhat faster. However, the PIC we are using does not have that much capacity.

The structure of decryption can be rearranged to be similar to that of encryptyion, which makes sharing of logic easier [1], [5] However, this requires applying the inverse mix column operation to the expanded key, so is not suitable for calculating the key on the fly as we are doing.

4.3 Timing

Table 1 gives the number of instruction cycles taken for each part of the Rijndael Algorithm.

TABLE 1. Timing (in instruction cycles) for Rijndael implemented on PIC alone. The Shift rows operation is shown as taking zero time, because it is a set of moves that can be made implicit in the other operations.

Operation	Encrypt	Decrypt	Total
Setup and control	121	135	256
Key expansion	740	740	1480
Key XOR	176	176	352
Register moves	352	352	704
Byte substitution	960	960	1920
Shift rows	0	0	0
Mix columns	1620	3168	4788
Total	3969	5531	9500

5. COPROCESSOR IMPLEMENTATION

As we see in Table 1, byte substitution and mix columns are two of the more expensive operations. They can also be folded together into an operation that takes 4 bytes of input and returns 4 bytes of output, providing a clear candidate for something that can be taken off to a coprocessor. These operations are the grey box in Figure 2, and their implementation on the CPLD is shown in Figure 3.

In our implementation of this PIC and the CPLD are interfaced using 8 bidirectional data lines and are

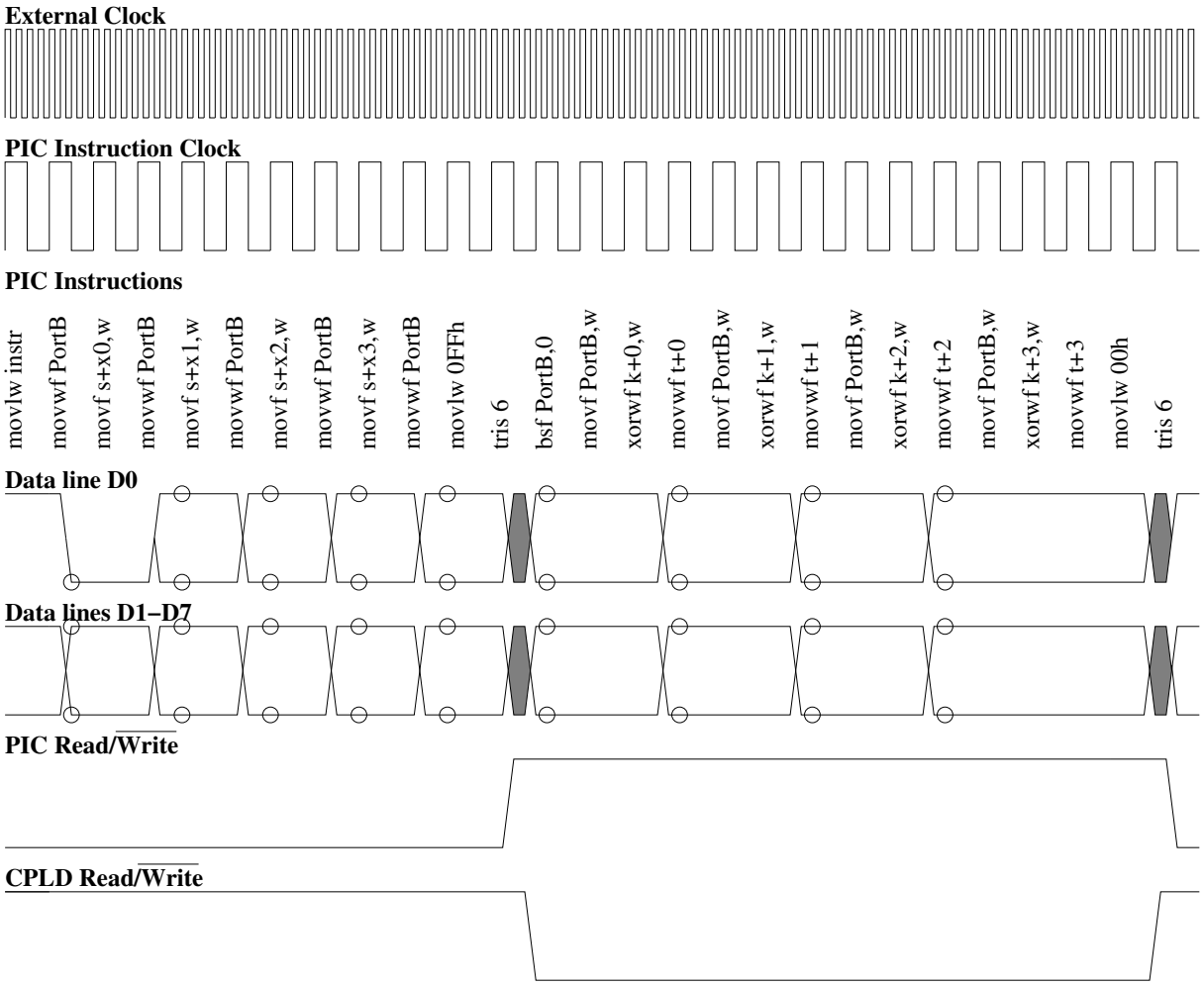

Figure 4. Instruction execution timing. The PIC instruction clock runs at a quarter the speed of the external clock. A typical instruction stream from the PIC is shown. Note that the xorwf instructions are not part of getting information on and off the CPLD, but are included here to avoid extra register shifts elsewhere. When the CPLD is not processing, the PIC holds D0 high, and D1-D7 are undefined. When the PIC has an instruction for the coprocessor, it drives D0 low, and puts the instruction on D1-D7. It then sends 4 data bytes, and switches to reading. The CPLD reads the instruction and the data (approximately where the circles are). After a 2 cycle pause so that the PIC and CPLD aren't trying to write at the same time, but not long enough for the signals to float into the linear region, the CPLD switches to write and sends the 4 bytes of result back, which the PIC reads approximately where the circles are. The CPLD then hands reading back to the PIC, and the CPLD is ready for the next instruction. This is all synchronized on the common clock

driven with a common clock (alternate designs might also include a control line for handshaking). The CPLD we use is a 256 macrocell Xilinx Coolrunner II (an XPLA3 could nearly equivalently be used), which can easily fit the logic required to perform this function for both encryption and decryption. For the inverse operation in $GF(2^8)$, we use the filed factoring approach [4], as it is small, but still fast enough. The CPLD is easily fast enough to keep up with the PIC - the bottleneck to the interface throughput is that it takes the PIC one instruction (4 clock cycles) to read or write from the data port, and another instruction to write that value to or read that value from memory.

This leads to a significant throughput improvement as shown in Table 2.

TABLE 2. Timing (in instruction cycles) for Rijndael implemented on PIC with CPLD

Operation	Encrypt	Decrypt	Total
Overhead	139	131	270
Key Expansion	530	550	1080
Key XOR	176	176	352
Coprocessor	920	920	1840
Total	1765	1777	3542

There are however costs to this approach. Because the PIC and CPLD are synchronized using the clock, there are 22 instruction cycles over which an interrupt wout disrupt communications, so interrupts must be disabled for that time or there must be some other mechanism to ensure interrupts don't occur.

Table 3 gives the resource usage of PIC vs.

PIC+CPLD

TABLE 3. Resources used by a PIC only implementation vs PIC+CPLD implementation

Resource	Total	PIC	PIC+CPLD
PIC program memory†	2048	1739	1028
PIC data memory‡	128	87	82
PIC I/O ports	22	0	8
CPLD macrocells	256	0	184
CPLD product terms	896	0	569
CPLD function blocks	640	0	385

†words ‡bytes

6. FPGA

An FPGA is typically much larger than a CPLD, and it is possible to implement the entire Rijndeal encryption and decryption algorithm on one, often at very high throughput indeed on one of the larger FPGAs, for example, 1.94 Gb/s on a Xilinx Virtex XCV1000BG560-4 FPGA [3]. However FPGAs are not designed for low power use - even the smallest ones have a quiescent power use of 40 mW or more and this will increase rapidly with work performed. Even though power/performance may be excellent, power will be too high to consider in a discussion on low power.

Of course, if the budget stretches to custom silicon, then a full rijndael in silicon implementation is likely to be a very good option.

7. POWER ANALYSIS

7.1 Modelling Power Use

From the datasheets, we determine approximate power use models for the devices that we use.

7.1.1 PIC. In terms of power usage, the PIC16F870 operates in one of four modes, in increasing order of power consumption:

Sleep In this mode, the PIC does no work, but consumes very little power - in the order of 0.01 μA plus whatever power is drawn from the peripherals.
LP (Low Power Crystal) 25 kHz to 100 kHz
XT (Crystal/Resonator) 100 kHz to 4 MHz
HS (High Speed Crystal) 4 MHz to 20 MHz

The power consumption at the slower modes is significantly lower than at faster modes, so there is for instance a real advantage to be able to able to reduce the clock from above 4 MHz to below 4 MHz and change the mode from HS to XT.

The PIC also draws much less power at lower voltages, but cannot be clocked as fast, and only some models of PIC can be driven below 4V.

7.1.2 CPLD. The power usage of a CPLD is a small base amount of power plus a certain amount of power per switching, which will be proportional to frequency. We estimate the power consumption using two methods: considering the amount of logic and looking at the data sheets, or by using the XPower tool that is part of the Xilinx software. These methods approximately agree.

There are two models of Xilinx CPLD we consider. The XCR3256 is a Coolrunner XPLA3 CLPD runs at 3.3V. Its power consumption can be approximated by $Power_{XCR3256} = 0.07 + 0.25f$ mW, where f is the external clock frequency in MHz. The XC2C256 is a Coolrunner II CPLD that runs at 1.8V, Its power consumption can be approximated by $Power_{XC2C256} = 0.04 + 0.07f$ mW. Note that the Coolrunner II cannot handle 5V I/O, whereas the older Coolrunner XPLA3 can.

In Figure 1, it can be seen that the addition of a CPLD dramatically increases the throughput/power ratio, allowing the PIC to perform at a lower clock than it could alone and often allowing a reduced voltage as well.

8. CONCLUSIONS

There are gains to be made in throughput or power use by including a coprocessor, particularly when there is already a CPLD already in the system. The cost of this is some increased complexity, required care over the interrupt system on the microcontroller, and a reduction in capacity for other task on the CPLD, and reduced available I/O ports on both the PIC and CPLD.

Whether this trade off is worth it depends on many parameters of the system, but this approach does provide an option for very low power cryptography.

8.1 Further work

8.1.1 Power Measurement. It would be useful to actually measure the power use of various configurations, rather than just relying on the datasheets. Our current setup (PIC and CPLD on separate boards, each with supporting circuitry) is not currently suited to this.

8.1.2 Further optimizations. Given that there is space unused on the CPLD, it would be posssible to move more of the working onto the CLPD for increased throughput and/or power efficiency. Good candidates for this would be to either move some of the key expansion work onto the CPLD, or to have some of the state stored on the CPLD.

Whether this is done in a practical application would depend on whether the CLPD needed some space left over for other functions. It would also add complexity

to the coprocessor interface.

8.2 Other Design Considerations

8.2.1 Other Choices of Microcontroller. Although we use a PIC16F870, this technique could easily be used with another microcontroller, particularly another PIC. Anything with less capacity is unlikely to be suitable, as the PIC16F870 is near the lower limit of resources.

In anything with more capacity, there would either be much more room for other functions, or optimisation techniques could be used that make the CLPD coprocessor option slightly less attractive (although it would still provide some throughput gain). One optimization that would be useful with more RAM would or EEPROM would be to store the complete expanded key, and not have to expand it on the fly.

8.2.2 Larger or Smaller CPLD. Although we were using a 256 Macrocell CPLD, it should be possible with careful optimisation to fit it onto a 128 Macrocell device. This would be particularly easy if only one of encryption or decryption were required, as the logic required becomes simpler.

With larger CPLDs, it is possible to do further optimizations as described in subsection 8.1.2. With a sufficiently large CPLD, it may be possible to do all the encryption/decryption on the CPLD, leaving the Microcontroller to handle the control logic. The largest available CPLD at the time of writing is 512 macrocells. We have not investigated how much it is possible to fit onto this.

8.2.3 Using RAM. The CPLDs we used can only store one bit per macrocell. The limitation to having more work done on the CPLD is the amount of state that can be stored. One approach may be to interface a low power RAM chip to the CPLD and store state in that. External RAM could also be used to store the expanded roundkey.

8.2.4 Larger Key or Block Sizes. It would probably not be very difficult to use a larger key size than 128 bits, as that mostly requires extra storage for the key. However a larger block size would be slower and require the storage of much more state. This would require a microcontroller with more RAM.

9. REFERENCES

[1] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. http://csrc.nist.gov/encryption/aes/rijndael/Rijndaelammended.pdf.

[2] Morris Dworkin. Recommendation for block cipher modes of operation - methods and techniques. http://csrc.nist.gov/publications/nistpubs/80038a/sp80038a.pdf.

[3] A.J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An fpga implementation and performance evaluation of the aes block cipher candidate algorithm finalists. In *The Third AES Conference (AES3), New York*, August 2000.

[4] R. W. Ward and T. C. A. Molteno. Efficient hardware calculation of inverses in $gf(2^8)$. In *Proceedings of the Tenth Electronics New Zealand Conference*, September 2003.

[5] Xinmiao Zhang and Keshab K. Parhi. Implementation approaches for the advanced encryption standard algorithm. *IEEE Circuits and Systems Magazine*, 2(4):24–46, 2002.

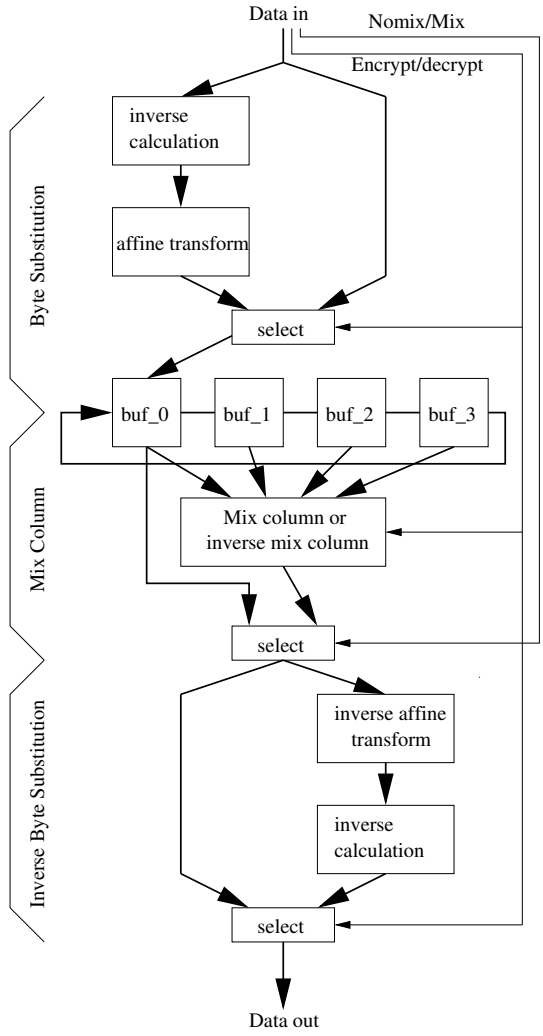

Figure 3. Internal workings of the coprocessor. Clocking and tristate logic is omitted. The first byte received is the instruction that determines whether encryption or decrpytion is being performed. The data is then stored in a buffer, with a byte substitution being performed first if encrytion is selected. Every time a byte is received, the buffers are rotated, so after 4 bytes, the buffers are full. The CPLD then switches to sending. The mix column or inverse mix column operation is performed and the information sent to the output, after the inverse byte susbstitution in the case of decryption. The buffers are rotated each time a byte is sent to give the 4 different output values. The 'inverse calculation' is a single execution unit is shared between the two substitution units, as it is expensive in area.