Assignment

ELECTENG702 Advanced Embedded Systems

Improving AES128 software for Altera Nios II processor using custom instructions

October 1. 2005

Professor Zoran Salcic

by

Kilian Foerster

10-8 Claybrook Road, Parnell
Auckland 1001
email: kfoe001@ec.auckland.ac.nz
UID: 4115147

Abstract

This report describes how custom instructions for the Altera Nios II processor can be used to improve embedded system implementations. This is done by example on an AES128 software implementation. All steps from analysing the software, creating the custom instructions to the final tests are covered. The final comparison between the initial and the improved implementation shows that custom instructions can lead to faster and cheaper designs.

Contents

1 Introduction

The aim of this assignment is to show how custom instructions for the Altera Nios II processor can be used to improve Nios II embedded system implementations in terms of speed and area. The Nios II processor enables the designer to change the hardware/software partitioning during system implementation. This is often done iteratively by replacing different parts of the software with hardware followed by an evaluation of the result. As the hardware design of the Nios II processor can be changed easily the development time for the whole system drops dramatically. To show how that design process works an existing AES128 software implementation will be improved using custom instructions. Therefore one full design iteration from analysing the software to the final evaluation of the result will be accomplished.

2 Nios II custom instruction overview

By using custom instructions with the Altera Nios II embedded processor time-critical software algorithms can be accelerated. This is achieved by replacing a complex sequence of standard instructions by a single instruction implemented in hardware. Up to 256 custom instructions can be added to the Nios II processor. [2]

A custom instruction is a custom logic block directly connected to the Nios II arithmetic logic unit (ALU) [2]. This way the design of the FPGA-based Nios II processor can be changed easily. Therefore the designer still has all freedoms in terms of hardware/software partitioning during implementation.

There are different types of custom instructions available. The main types are "combinatorial" and "multi-cycle". The combinatorial custom instruction block reads a 32 bit input vector from "dataa" and/or "datab" and generates a 32 bit "result" vector within one clock cycle (figure 1). [2]

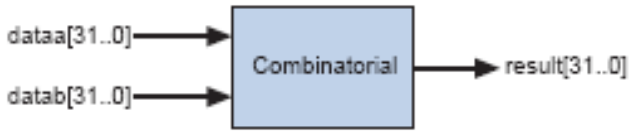

Figure 1: Combinatorial custom instruction block [2]

The multi-cycle custom instruction block has additional inputs for "clk", "clk_en", "start", "reset" and "done"; although some of the inputs are optional (figure 2). The multi-cycle custom instruction block also reads the input from "dataa" and/or "datab" and produces a "result". This is done within a finite number of clock cycles. The maximum number of clock cycles the multi-cycle block needs to perform the operation has to be known when adding the instruction to the Nios II processor. [2]

The multi-cycle custom instruction block can be extended to perform multiple operations. The operation selection is done by an input vector "n". This vector is up to eight bit wide and allows up to 256 different operations. These different operations are simply mapped to different opcodes. [2] A custom instruction can incorporate further functionalities like the access to internal register files. Logic outside the Nios II processors' data path can be interfaced, too. [2]

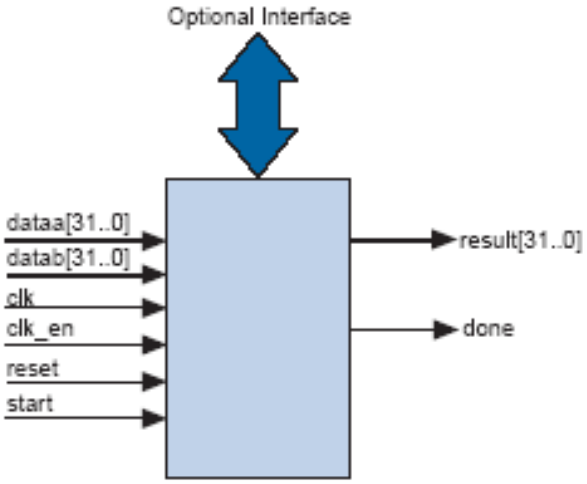

Figure 2: Multi-cycle custom instruction block [2]

3 The AES128 software implementation for Nios II processors

This section gives a very short overview of the AES algorithm and a special way to implement AES on 32 bit platforms. Also the used AES software implementation by Chrisophe Devine is introduced.

3.1 Overview over AES (Advanced Encryption Standard)

The National Institute for Standards and Technology (NIST) ran a contest for the new Advanced Encryption Standard (AES) for the United States beginning in January 1997. This new standard would replace the old Data Encryption Standard (DES) and triple-DES. Rijndale was announced as the winner of the five finalist algorithms in October 2000. Rijndael was designed by John Daemen and Vincent Rijmen, two Belgian designers. [5]

AES is a symmetric block cipher with a fixed block length of 128 bit and a variable key length of 128 bit, 192 bit or 256 bit ([5] p. 31). For an encryption the following steps have to be performed ([5] p. 34):

- key expansion
- key addition
- encryption rounds (number depending on key length)
  - sbox substitution
  - shift row
  - mix column
  - key addition

- final round
  - sbox substitution
  - shift row
  - key addition

The most complex step in the algorithm is the mix column step. This step performs a polynomial multiplication over GF(2^8). All other steps are substitutions, rotations and shifts. ([5] pp. 34-41)

3.2 Short analysis of the chosen AES implementation

There are many different ways of implementing the AES algorithm on a 32 bit processor. Rijmen and Daemen ([5] pp. 56-59) describe a very fast implementation by combining the different steps of the round transformation in a single set of look-up tables. For encryption four tables with each 256 4-byte entries requiring 4 kB of storage space are used. One encryption round then takes 16 table look-ups and 16 XOR operations. Decryption can be implemented in a similar way using four different tables requiring also 4 kB of storage space. The performance of the decryption is equal to the encryption.

Christophe Devine published his 32 bit AES implementation [4] based on that principle under the GNU General Public License. His code was configured to use 8 kB of precomputed tables for encryption and decryption. He also uses a look-up table for the key expansion. This table requires 4 kB of storage space and is created during runtime. In addition the forward and reverse S-Boxes, each has 256 1-byte entries, add another 512 bytes to the required storage space. All together his implementation requires 12.5 kB of storage space for look-up tables.

This implementation originally supported 128 bit, 192 bit and 256 bit keys. As the different key lengths only affect the number of encryption/decryption rounds the support for 192 bit and 256 bit keys was removed. All tests and comparisons were done using the 128 bit algorithm.

4 Improving the software implementation using Nios II custom instructions

The first step is to figure out which parts of the code need the most time or area. With this knowledge custom instructions can be created to improve the software implementation.

Of course the big look-up tables consume a lot of area. Since the way the algorithm was implemented will not be changed these tables are still needed. However the four tables for encryption are only rotated versions of one another ([5] p. 59). Thus it is possible to replace four tables by one in addition to some rotating logic. The same applies to the decryption and the key scheduling tables.

In concerns of time it turns out quickly that the encryption/decryption rounds are worth to be improved. One encryption round consists of 16 table look-ups, 16 XOR operations and 12 shift operations. For the encryption of one block nine rounds plus a slightly different last round are necessary.

The encryption round macro (figure 3) takes four 32 bit inputs (Y0 .. Y3) and returns four 32 bit outputs (X0 .. X3). A closer look to the macro shows that each output depends on all four inputs. As a custom instruction can only take two 32 bit inputs it is not possible to build an instruction that just returns one 32 bit output element. A possible way is to combine one forward table look-up with one shift and one XOR operation. This means the custom instruction will take one 32 bit input on "dataa" (e.g. Y0) and perform the table look-up (e.g. on FT0) after the shift (e.g. >> 24). The result of the look-up is than XORed with the 32 bit input given on "datab" (e.g. RK[0]) and returned via "result". To be able to perform the look-up the look-up table has to be included in the custom instruction block.

```
X0 = RK[0] ^ FT0[ (uint8) ( Y0 >> 24 ) ] ^  \
             FT1[ (uint8) ( Y1 >> 16 ) ] ^  \
             FT2[ (uint8) ( Y2 >>  8 ) ] ^  \
             FT3[ (uint8) ( Y3       ) ];   \
                                            \
X1 = RK[1] ^ FT0[ (uint8) ( Y1 >> 24 ) ] ^  \
             FT1[ (uint8) ( Y2 >> 16 ) ] ^  \
             FT2[ (uint8) ( Y3 >>  8 ) ] ^  \
             FT3[ (uint8) ( Y0       ) ];   \
                                            \
X2 = RK[2] ^ FT0[ (uint8) ( Y2 >> 24 ) ] ^  \
             FT1[ (uint8) ( Y3 >> 16 ) ] ^  \
             FT2[ (uint8) ( Y0 >>  8 ) ] ^  \
             FT3[ (uint8) ( Y1       ) ];   \
                                            \
X3 = RK[3] ^ FT0[ (uint8) ( Y3 >> 24 ) ] ^  \
             FT1[ (uint8) ( Y0 >> 16 ) ] ^  \
             FT2[ (uint8) ( Y1 >>  8 ) ] ^  \
             FT3[ (uint8) ( Y2       ) ];   \
```

Figure 3: Encryption round [4]

To calculate one 32 bit encryption round output four different custom instructions have to be used. Each has to use a different table and shift by a different number of bits.

The final round (figure 4) is similar to the other rounds but uses a different table (the forward S-Box) and needs 24 shift operations. The custom instructions for the last round perform one table look-up each, two shift and one XOR operation. Four additional custom instructions are needed for the last round.

The decryption rounds are similar to the encryption rounds but use different tables and different shift operations. To improve also the decryption rounds eight more custom instructions have to be created.

The operations for the key expansion are similar to those used in the encryption/decryption rounds. Therefore custom instructions should be created for the key expansion, too.

```
X0 = RK[0] ^ ( FSb[ (uint8) ( Y0 >> 24 ) ] << 24 ) ^
             ( FSb[ (uint8) ( Y1 >> 16 ) ] << 16 ) ^
             ( FSb[ (uint8) ( Y2 >>  8 ) ] <<  8 ) ^
             ( FSb[ (uint8) ( Y3       ) ]       );

X1 = RK[1] ^ ( FSb[ (uint8) ( Y1 >> 24 ) ] << 24 ) ^
             ( FSb[ (uint8) ( Y2 >> 16 ) ] << 16 ) ^
             ( FSb[ (uint8) ( Y3 >>  8 ) ] <<  8 ) ^
             ( FSb[ (uint8) ( Y0       ) ]       );

X2 = RK[2] ^ ( FSb[ (uint8) ( Y2 >> 24 ) ] << 24 ) ^
             ( FSb[ (uint8) ( Y3 >> 16 ) ] << 16 ) ^
             ( FSb[ (uint8) ( Y0 >>  8 ) ] <<  8 ) ^
             ( FSb[ (uint8) ( Y1       ) ]       );

X3 = RK[3] ^ ( FSb[ (uint8) ( Y3 >> 24 ) ] << 24 ) ^
             ( FSb[ (uint8) ( Y0 >> 16 ) ] << 16 ) ^
             ( FSb[ (uint8) ( Y1 >>  8 ) ] <<  8 ) ^
             ( FSb[ (uint8) ( Y2       ) ]       );
```

Figure 4: Last encryption round [4]

Now all table look-ups are performed within a custom instruction. Therefore all look-up tables can be moved to the custom instruction block.

4.1 Creation of the custom instruction block

Knowing which functionalities have to be implemented in the custom instructions, the first step is to decide which type of custom instruction block to take.

Memory elements are needed within the custom instruction block. Only a multi-cycle custom instruction can be used because reading from memory always takes a multiple number of clock cycles. At least one to set the address and one to present the data.

As different custom operations use the same look-up table all custom operations should be integrated in one extended custom instruction block. The selection of the operation is then done by the input vector "n".

The custom instructions were designed using Altera Quartus II software. Figure 5 shows the internal structure of the custom instruction block. The "polmul_lut" block was designed using VHDL.To that block the "lpm_rom" memory blocks which contain the look-up tables are connected. As all tables consist of 256 elements, one 8 bit wide address bus can be used for all memories.

The "polmul_lut" block is a finite state machine with two states. In the first state ("set_addr") the 32 bit value of "dataa" is read and mapped to the eight bit of the "addr" bus. The firt two bits of "n" select which bits of "dataa" are written on "addr. Thus the first two bits of "n" select the first shift operation. These two bits also select which rotation is performed on the table output. In the second state ("output") the

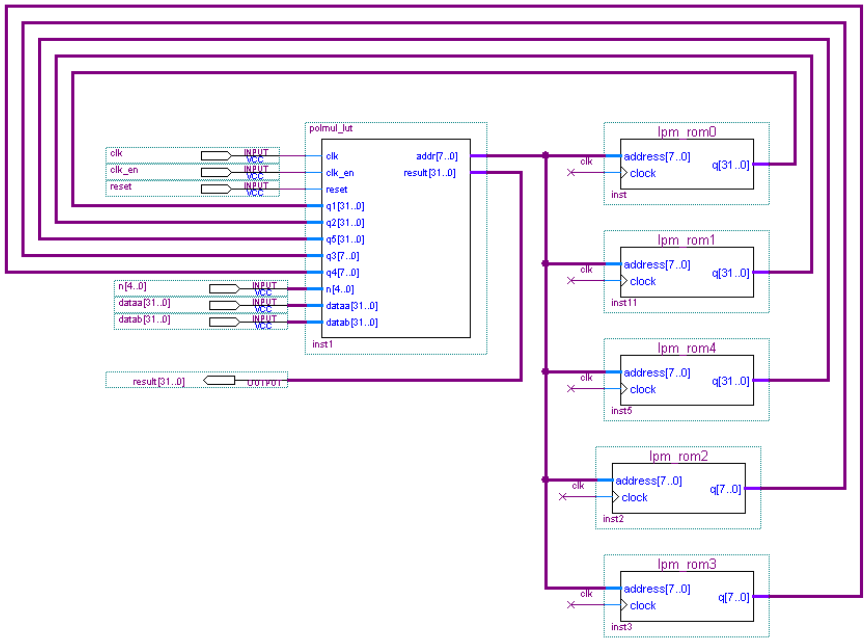

Figure 5: Internal structure of the custom instruction block

requested value is read from the memory. The third, fourth and fifth bit of "n" select which table will be read. In this state the resulting look-up value is also XORed with the value from "datab" and written to "result".

4.2 Adding the custom instruction block to the Nios II processor

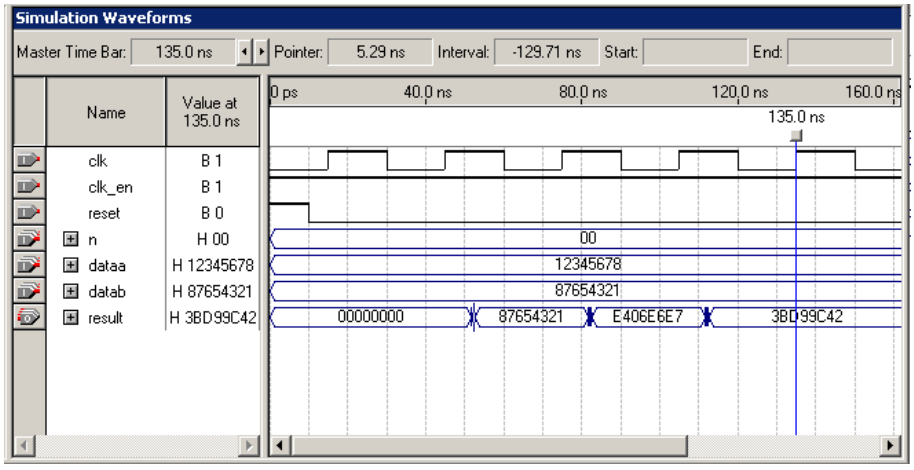

Figure 6: Timing analysis of the custom instruction block

To create a Nios II processor the SOPC Builder software is used. It allows to configure the processor core and to add a huge variety of predesigned peripherals. It is also used to add and configure custom instructions. When adding a multi-cycle custom

instruction using SOPC Builder it is required to enter the number of clock cycles the instruction needs to be executed. A timing analysis of the custom instruction block using Altera Quartus II showed that the output "result" is stable after five clock cycles (figure 6).

4.3 Inserting the custom instructions in the AES128 software implementation

Before a custom instruction can be used, it has to be defined in a header file. The data types and the numbers of the inputs and outputs are set by the custom instruction built-in functions. In addition to that the opcode as set by SOPC Builder is assigned. Figure 7 shows the definition of the forward table instructions for the encryption round. They take tow integers as input and return one integer as output [2].

```
#define FT30_N  0x00 //00000000b
#define FT28_N  0x01 //00000001b
#define FT116_N 0x02 //00000010b
#define FT024_N 0x03 //00000011b

#define FT30(A,B) __builtin_custom_inii(FT30_N,(A),(B))
#define FT28(A,B) __builtin_custom_inii(FT28_N,(A),(B))
#define FT116(A,B) __builtin_custom_inii(FT116_N,(A),(B))
#define FT024(A,B) __builtin_custom_inii(FT024_N,(A),(B))
```

Figure 7: Definition of the forward table custom instructions

After including the header file into the source of the AES128 implementation the custom instructions can be used. Figure 8 shows the encryption round macro using the new instructions. This part replaces the code in Figure 3. The instructions are nested into each other. The result of FT024 is an input of FT116, the result of FT116 is an input of FT28 and the result of FT28 is an input of FT30. This nesting is necessary as the XOR operations are performed within the instructions.

```
X0 = FT30(Y3,FT28(Y2,FT116(Y1,FT024(Y0,RK[0]))));
X1 = FT30(Y0,FT28(Y3,FT116(Y2,FT024(Y1,RK[0]))));
X2 = FT30(Y1,FT28(Y0,FT116(Y3,FT024(Y2,RK[0]))));
X3 = FT30(Y2,FT28(Y1,FT116(Y0,FT024(Y3,RK[0]))));
```

Figure 8: Encryption rounds using custom instructions

The improved code for the decryption rounds and the key scheduling is implemented in the same way just using different custom instructions.

4.4 Test of the new implementation

The first step is to test the new custom instructions to ensure they behave like expected. For testing their functionality ModelSim was used. Each instruction was tested with

two test vectors. The results show that the right tables were used and that the additional shift and rotation logic works properly. This test does not check on all table elements. The correctness of the precomputed tables has to be checked at a later point by testing the whole implementation.

After checking the functionality a small timing analysis was done with Quartus II to ensure all instructions take the same number of clock cycles to finish. It was confirmed that the output of all instructions is stable after five clock cycles (figure 6). Also the maximum clock frequency was checked as the custom instruction should not slow down the whole processor. The maximum clock frequency of the custom instruction block is 221 MHz whereas the maximum clock frequency of the Nios II/s processor core is about 180 MHz [3]. Both for an Altera Stratix II FPGA. Thus the maximum clock frequency of the design is not affected by the custom instructions.

As already mentioned the above tests do not check all table entries nor the whole implementation. One possibility to test the whole implementation is the "Rijndael Monte Carlo Test" which is already included in the code from Christophe Devine [4]. This test performs $4 \cdot 10^6$ encryption and decryption operations and compares the results to expected values. Thus it ensures the correctness of the whole implementation including the look-up tables. Unfortunately the improved implementation could not be simulated using the Altera Nios II IDE as the simulation of multi-cycle custom instructions is not supported. Even if the simulation was supported it would take too long as the simulator is very slow. Therefore the test can only be executed on real hardware which wasn't done yet.

5 Comparison of the custom instruction solution to the plain software implementation

To compare the performance of both implementations different aspects must be taken into account. One of them is the area used on the target device. This means the number of logic and memory elements. The number of logic and memory elements depends on the processor core and its additional peripherals. The codesize and the ram usage of the software select the amount of memory needed. Also hardware multiply units or custom instructions increase the number of logic and memory elements.

Another aspect is the time that is the algorithms' execution time. This time depends on the number and kind of instructions and the clock frequency of the processor.

5.1 Comparison in concerns of area

	Implementation with custom instructions	Implementation in plain software
Codesize	17kB	45 kB
Additional memories	3,5 kB (look-up tables)	4 kB (RAM for key scheduling table)
Logic Elements	352 ALUTs 41 register	? (optional hardware multiplier)

Table 1: Comparison of both variants in concerns of area

The same processor core can be used for both software variants. For the further calculations we assume that the Nios II/s core is used on a Stratix II target device. The only difference for both variants is the amount of memory needed and the additional hardware blocks. For the improved implementation the previously created custom instruction block must be added. For the plain software implementation we could add a hardware multiply unit to speed up shift operations. Table 1 shows that the improved variant consumes far less memory blocks than the plain software variant. It also turns out that the custom instructions need additional logic elements.

5.2 Comparison in concerns of time

The clock frequency for both variants is restricted by the used processor core. Thus the differences in execution time are only due to the number and the type of the instructions. Different instructions take a different number of clock cycles to execute. The previously created custom instructions for example need five clock cycles to present the result and an additional cycle to be executed by the processor. A shift operation takes one cycle per bit. If a hardware multiplier is used only three to four cycles are needed for any shift [1].

By looking at the dissassembly of the compiled code the number of instructions can be counted. Afterwards the clock cycles, needed for these instructions to be executed, can be added together. As the difference of both variants is important, the differing parts were compared only. Table 2 shows that the improved variant is about two times faster than the plain software variant. Without a hardware multiplier it would be about four times faster.

	with custom instructions	plain software	plain software with multiplier	number of executions per 128 Byte block
Setup enc. round keys	30	155	75	10
Setup dec. round keys	30	100	55	8
encryption round	120	400	220	9
encryption last round	30	155	75	1
decryption round	120	400	220	9
decryption last round	30	155	75	1
total per 128 Byte block	2760	9860	5300	-

Table 2: Clock cycles needed by both variants

6 Conclusion

By improving the existing plain software AES128 implementation using custom instructions the execution time was reduced significantly. Also less area is needed. This means the new implementation is more cost effective than the initial one. The next step in this design process would be to test the new implementation on an evaluation board. After that another iteration cycle could follow. Therefore other parts of the software would be replaced by hardware or vice versa. If after some iterations the result still doesn't meet the required specifications it might become necessary to try a complete different way. This could be a plain hardware implementation for example.

7 Appendix

custom.bdf	Altera Quartus II block design file for the custom instruction block
polmul_lut.vhd	VHDL file implementing the finite state machine used in custom.bdf
custom.vwf	Altera Quartus II waveform file for the timing simulation of custom.bdf
custom_test.vhd	Modelsim test bench for the functional test of custom.bdf
aes.c	C source file of the AES128 implementation by Christophe Devine
aes.h	C header file for use with aes.c
aes_modified.c	With custom instructions modified version of aes.c
system.h	C header file to define the custom instruction used in aes_modified.c

References

[1] Altera, *Nios II Process Reference Handbook*, ver. 5.0, July 2005

[2] Altera, *Niso II Custom Instruction User Guide*, ver. 1.2, January 2005

[3] Altera, *Nios II Performance Benchmarks*, ver. 1.0, October 2004

[4] Christophe Devine, *AES source code*, http://www.cr0.net:8040/code/crypto/aes/

[5] J.Daemen and V. Rijmen, *The Design of Rijndael*, Springer-Verlag: 2002