Presentation

Project: Embedded Processor Design and Optimization

08.08.2024

Pei-Yu Lin

ICE Institute for Communication Technologies and Embedded Systems | RWTH AACHEN UNIVERSITY

Overview

- Identify the HOTSPOT
  - Found `shift_right_block` is the HOTSPOT
  - Set `shift_right_block` is the Primitive Function

- Implement the Primitive Function on trv32p3 RISC-V
  - < aes.p >
  - < aes.n>
  - < trv32p3.n >

- Optimize the Application for the AES-GCM algorithm
  - Method 1, Method 2
  - Method 3 -> big optimization
  - Method 4 -> reach the Lowest cycle count

- Apply Logic Synthesis
  - Print all result
  - Reach the min clock period / max clock frequency
  - Make an AT diagram

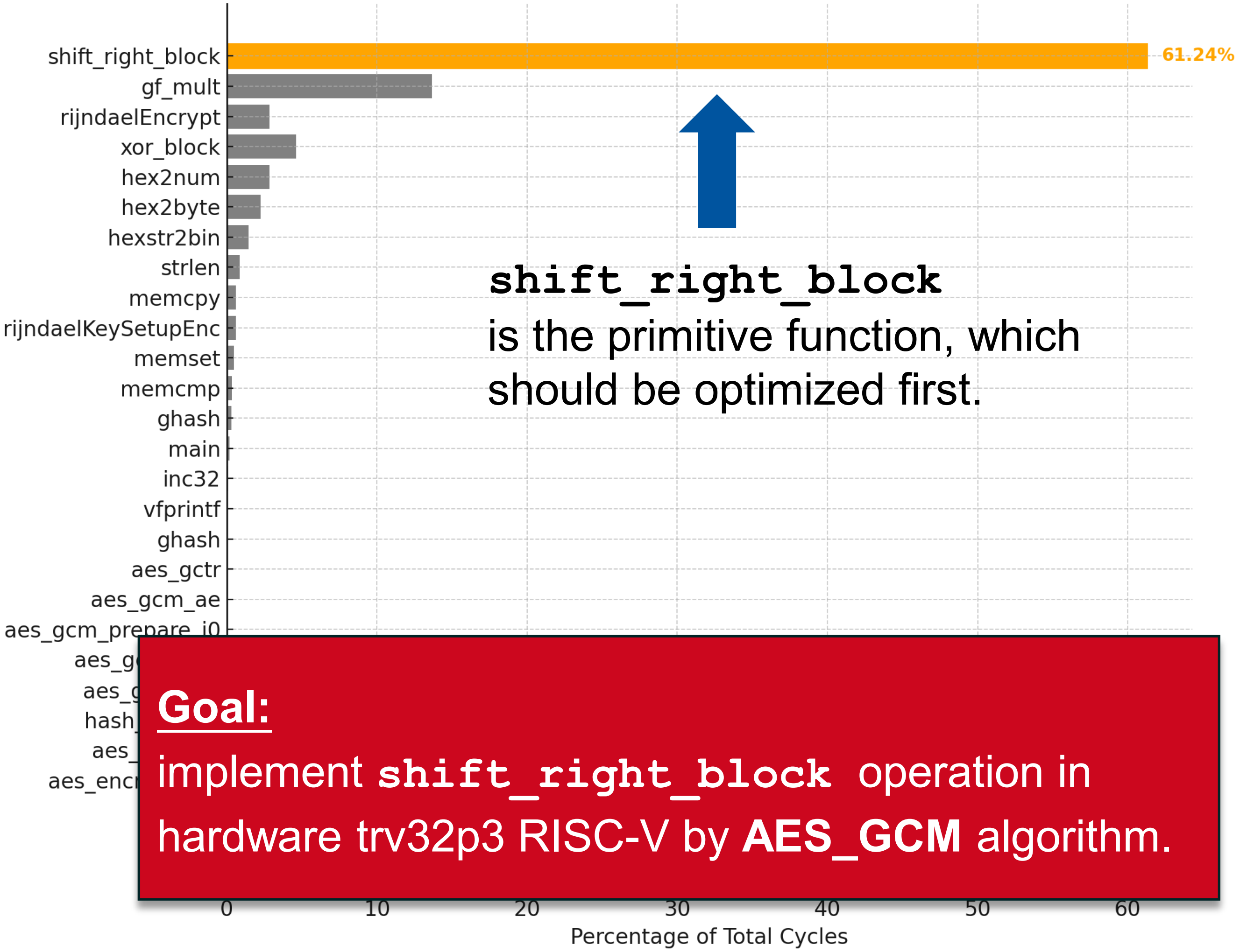
- My best Performance
  - Achieve the requirement

Apply profiling, found `shift_right_block` is the HOTSPOT

instruction_report of <aes-gcm.prx>

```
Profiling information for ::iss generated by Checkers R-2020.09#3904ab4039#201130 on Thu Jul 11 10:08:24 2024

Program being simulated:

    /net/heap/lin/asip-lab/trv32p3/applications/aes-256-gcm/Release/main

    Total cycle count          :              696016
    Report cycle count         :              696016
    Total instruction count    :              624020
    Report instruction count   :              624020
    Report instruction coverage : 95.06%
    Total size in program memory:               6876

Command used to generate this report: ::iss profile save /net/heap/lin/asip-lab/trv32p3/applications/aes-256-gcm/instruction_report.txt -type function_details -user

Function summary:

         Cycles % of total Instruction % of total % Coverage  Function                                                   Relative cycle use in simulation
    ----------- ---------- ----------- ---------- ----------- ---------------------------------------------------------- ----------------------------------------------
         426228     61.24%      398886     63.92%     100.00% shift_right_block void_shift_right_block___P__ucha ******************************
          95207     13.68%       75465     12.09%     100.00% gf_mult void_gf_mult___P__uchar___P__uchar___P__uc ******
          57362      8.24%       57072      9.15%     100.00% rijndaelEncrypt void_rijndaelEncrypt___P__uint___s ****
          44631      6.41%       39933      6.40%     100.00% xor_block void_xor_block___P__uchar___P__uchar     ***
          19935      2.86%       12063      1.93%      62.50% hex2num __sint_hex2num___cchar                     *
          15744      2.26%       11808      1.89%      75.00% hex2byte __sint_hex2byte___P__cchar                *
          10164      1.46%        7104      1.14%      85.71% hexstr2bin __sint_hexstr2bin___P__cchar___P__uchar
           5240      0.75%        2656      0.43%     100.00% strlen strlen
           4348      0.62%        4148      0.66%     100.00% memcpy memcpy
           4164      0.60%        4068      0.65%      97.97% rijndaelKeySetupEnc __sint_rijndaelKeySetupEnc___P
           3292      0.47%        3156      0.51%     100.00% memset memset
           2576      0.37%        1828      0.29%      66.67% memcmp memcmp
           1418      0.20%        1136      0.18%     100.00% ghash void_ghash___P__uchar___P__uchar___uint___P_
           1120      0.16%         935      0.15%      98.59% aes_gctr void_aes_gctr___Pvoid___P__uchar___P__uch
            919      0.13%         783      0.13%      80.34% main _main
            575      0.08%         500      0.08%     100.00% inc32 void_inc32___P__uchar
            378      0.05%         315      0.05%     100.00% vfprintf vfprintf
            378      0.05%         336      0.05%     100.00% aes_gcm_ghash void_aes_gcm_ghash___P__uchar___P__u
            375      0.05%         254      0.04%      74.19% div_called __sint_div_called___sint___sint_R__sint
            367      0.05%         327      0.05%      96.30% malloc malloc
            306      0.04%         252      0.04%      97.67% aes_gcm_ae __sint_aes_gcm_ae___P__uchar___P__uchar
            228      0.03%         192      0.03%     100.00% aes_gcm_prepare_j0 void_aes_gcm_prepare_j0___P__uc
            169      0.02%         129      0.02%     100.00% aes_gcm_gctr void_aes_gcm_gctr___Pvoid___P__uchar_
            156      0.02%         114      0.02%      95.00% aes_gcm_init_hash_subkey __Pvoid_aes_gcm_init_hash
            145      0.02%         116      0.02%     100.00% aes_encrypt void_aes_encrypt___Pvoid___P__uchar___
            144      0.02%         108      0.02%      81.82% aes_encrypt_init __Pvoid_aes_encrypt_init___P__uch
            120      0.02%          96      0.02%     100.00% free free
            105      0.02%          84      0.01%     100.00% printf printf
```

With the instruction report of `aes-gcm.prx`, we can find `shift_right_block` has the highest relative cycle counts with **61.24%** coverage in the total cycle usage.



`shift_right_block` is the primitive function, which should be optimized first.

Goal:

implement `shift_right_block` operation in hardware trv32p3 RISC-V by **AES_GCM** algorithm.

< aes.p >

Step1: Build a PDG file for implementing the primitive function

```
#define GETU32(pt) (((u32)(pt)[0] << 24) ^ ((u32)(pt)[1] << 16) ^ \
((u32)(pt)[2] << 8) ^ ((u32)(pt)[3]))

#define PUTU32(ct, st) { \
(ct)[0] = (u8)((st) >> 24); (ct)[1] = (u8)((st) >> 16); \
(ct)[2] = (u8)((st) >> 8); (ct)[3] = (u8)(st); }


static void shift_right_block(u8 *v)
{
    u32 val;
    val = GETU32(v + 12);
    val >>= 1;

    if (v[11] & 0x01)
    val |= 0x80000000;
    PUTU32(v + 12, val);
    val = GETU32(v + 8);
    val >>= 1;

    if (v[7] & 0x01)
    val |= 0x80000000;
    PUTU32(v + 8, val);
    val = GETU32(v + 4);
    val >>= 1;

    if (v[3] & 0x01)
    val |= 0x80000000;
    PUTU32(v + 4, val);
    val = GETU32(v);
    val >>= 1;

    PUTU32(v, val);
}
```

aes-gcm.prx

Primitive function
In C application

```
//#define GETU32(pt) (((u32)(pt)[0] << 24) ^ ((u32)(pt)[1] << 16) ^ ((u32)(pt)[2] << 8) ^ ((u32)(pt)[3]))
//#define GETU32(pt) (((pt >> 0) & 0xFF) << 24) ^ (((pt >> 8) & 0xFF) << 16) ^ (((pt >> 16) & 0xFF) << 8) ^
((pt >> 24) & 0xFF)
//#define PUTU32(ct, st) { \
(ct)[0] = (u8)((st) >> 24); (ct)[1] = (u8)((st) >> 16); (ct)[2] = (u8)((st) >> 8); (ct)[3] = (u8)(st); }
//#define PUTU32(st) (((st >> 0) & 0xFF) << 24) | (((st >> 8) & 0xFF) <<16)|(((st >> 16) & 0xFF) << 8) | ((st
>> 24) & 0xFF)
void aes_srb(w32 a, w32 b, w32 c, w32 d, w32& a_out, w32& b_out, w32& c_out, w32& d_out)
{
    uint32_t val;
    //val = GETU32(d); // v + 12
    val = revbyte(d);
    val >>= 1;

    if (c[31:24] & 0x01) // v[11]
    val |= 0x80000000;
    //d_out = PUTU32(val); // v + 12, val
    d_out = revbyte(val);
    //val = GETU32(c); // v + 8
    val = revbyte(c);
    val >>= 1;

    if (b[31:24] & 0x01) // v[7]
    val |= 0x80000000;
    //c_out = PUTU32(val);
    c_out = revbyte(val);
    val = revbyte(b);
    //val = GETU32(b); // v + 4
    val >>= 1;

    if (a[31:24] & 0x01) // v[3]
    val |= 0x80000000;
    //b_out = PUTU32(val);
    b_out = revbyte(val);
    val = revbyte(a);

    //val = GETU32(a); // v
    val >>= 1;
    //a_out = PUTU32(val);
    a_out = revbyte(val);
}
```

aes.p

Primitive function
in PDG <aes.p>

< aes.p >

What did I change the Primitive Function from C to PDG?

1

```
static void shift_right_block(u8 *v)
```

→

```
void aes_srb(w32 a, w32 b, w32 c, w32 d, w32& a_out, w32& b_out, w32& c_out, w32& d_out)
```

2

```
val = GETU32(v + 12);

PUTU32(v + 12, val);
```

→

```
val = revbyte(d);

d_out = revbyte(val);
```

bit	{07:00}	{15:08}	{23:16}	{31:24}
d	v[12], *(v+12)	v[13], *(v+13)	v[14], *(v+14)	v[15], *(v+15)
c	v[8], *(v+8)	v[9], *(v+9)	v[10], *(v+10)	v[11], *(v+11)
b	v[4], *(v+4)	v[5], *(v+5)	v[6], *(v+6)	v[7], *(v+7)
a	v[0], *v	v[1], *(v+1)	v[2], *(v+2)	v[3], *(v+3)

< aes.p >

2

Replace GETU32(pt), PUTU32(ct, st) with revbyte(val)

```
#define GETU32(pt) (((u32)(pt)[0] << 24) ^ ((u32)(pt)[1] << 16) ^ \
((u32)(pt)[2] << 8) ^ ((u32)(pt)[3]))
```

```
#define GETU32(pt) (((pt >> 0) & 0xFF) << 24) ^ (((pt >> 8) & 0xFF)
<< 16) ^ (((pt >> 16) & 0xFF) << 8) ^ ((pt >> 24) & 0xFF)
```

```
// Implementation of the revbyte
w32 revbyte(w32 input) {
    w32 byte0 = (input >> 0) & 0xFF;
    w32 byte1 = (input >> 8) & 0xFF;
    w32 byte2 = (input >> 16) & 0xFF;
    w32 byte3 = (input >> 24) & 0xFF;
    return (byte0 << 24) | (byte1 << 16)
| (byte2 << 8) | byte3;
}
```

```
#define PUTU32(ct, st) { \
(ct)[0] = (u8)((st) >> 24); (ct)[1] = (u8)((st) >> 16); \
(ct)[2] = (u8)((st) >> 8); (ct)[3] = (u8)(st); }
```

```
#define PUTU32(st) (((st >> 0) & 0xFF) << 24) | (((st >> 8) & 0xFF) << 16)
| \ (((st >> 16) & 0xFF) << 8) | ((st >> 24) & 0xFF)
```

revbyte(val);

< aes.n >

Step2: Make a new nML `aes.n` in lib folder.

1. Declare a new functional unit for aes.

1. Because no values pass through the function, we don't need any transistory.

2. The Primitive function needs 4 inputs and 4 outputs of 32-bit value. However, trv32p3 only has 2 read ports. So, we **create 4 alias registers RA, RB, RC, RD** and map them one-to-one to a register on data memory, and give each of them a read and a write port.

```
fu aes ; // Declares a new functional unit for aes
// no transitories

// 4 `alias`-register for aes_srb read and write
reg RA <w32> alias X[8] read(rar) write(raw);
reg RB <w32> alias X[9] read(rbr) write(rbw);
reg RC <w32> alias X[10] read(rcr) write(rcw);
reg RD <w32> alias X[11] read(rdr) write(rdw);

...
```

aes.n

< aes.n >

Step3: Define the aes_srb instruction `aes_srb_instr`

4. To implement the instruction, a, b, c, d values should be read from the data memory, after AES-GCM algorithm, the outputs should return to the memory. Both read and write are via the read and write ports of the alias registers RA, RB, RC, RD.

5. Define the **syntax** of the this `aes_srb` instruction in assembly language e.g. `aes_srb: RA, RB, RC, RD`

6. Because there is no parameter for the operation, **image** can be assigned a set of random 25-bit binary values.

```
fu aes ; // Declares a new functional unit for aes
// no transitories

// 4 `alias`-register for aes_srb read and write
reg RA <w32> alias X[8] read(rar) write(raw);
reg RB <w32> alias X[9] read(rbr) write(rbw);
reg RC <w32> alias X[10] read(rcr) write(rcw);
reg RD <w32> alias X[11] read(rdr) write(rdw);

// Define the aes_srb operation
opn aes_srb_instr(){ // don't need to create any parameter
action {
stage EX:
    aes_srb(rar=RA, rbr=RB, rcr=RC, rdr=RD, RA=raw, RB=rbw ,RC=rcw, RD=rdw) @aes;
}
// Define the syntax of the instruction in assembly language
syntax : "aes_srb:" RA", " RB", " RC", " RD;
image: "0000001001000000000101110";
}
```

aes.n

Institute for Communication Technologies and Embedded Systems
RWTH AACHEN UNIVERSITY

Step4: add the new instruction aes_srb_instr into the processor nML.

After creating an instruction operation aes_srb_instr , add it into bit-32 instruction, like majCUSTOM

Here, we don't need to add `revbyte_instr` , because `revbyte_instr` has been combined with `sha_instr` written in sha.n

```
opn majCUSTOM3 (alu_rriu_ar_instr | sha_instr | zlp_instr | aes_srb_instr) complete_image;
// added...
```

trv32p3.h

```
opn sha_instr (sha256_instr | sha256_step_instr | revbyte_instr);
```

Optimize the Application for the AES-GCM algorithm

I use 4 steps to reach the lowest cycle count.

Method 1:

In shift_right_block(), replace all of the code with a line: aes_srb(). Still implement the same function name of shift_right_block() in gf_mult()

```
void shift_right_block(){
    /*
    ...
    */
    aes_srb();
}
```

6 of 6 test cases executed and computed correctly!

Total cycle count in encryption function: 279055

Cycle count per test case in encryption function: 46509

Method 2:

In gf_mult(), directly replace the code related to shift_right_block() with the new aes_srb()

```
void gf_mult(){
    /*
    ...
    */
    // shift_right_block()
    aes_srb();
    /*
    ...
    */
}
```

6 of 6 test cases executed and computed correctly!

Total cycle count in encryption function: 252559

Cycle count per test case in encryption function: 42093

→ Directly delete a function is better

Optimize the Application

Method 3: Use u32 a, b, c, d variables to replace the u8 v array

The most Incredible optimization

(1) create **u32 a, b, c, d** variables and copy the value in **u8 v array** to **a, b, c, d.**

```
u32* d1;
u32 a,b,c,d;
a = *(u32*)(v);
b = *(u32*)(v+4);
c = *(u32*)(v+8);
d = *(u32*)(v+12);
```

(2) cast all the **u8 v** into **u32 a, b, c, d**. And we also need to change the condition in if-else

```
if(d>>24 & 0x01) { // v[15] --> d>>24
   a = a & 0xffffff00 | (a & 0xFF ^ 0xe1); //v[0] ^= 0xe1;
}
if (x[i] & BIT(7 - j)) {
    /* Z_(i + 1) = Z_i XOR V_i */
    //xor_block(z, v);
    d1 = (u32*) z; // add...
    *d1++ ^= a;
    *d1++ ^= b;
    *d1++ ^= c;
    *d1++ ^= d;
} else {
...
```

(3) return the value of a, b, c, d variables back to **u8 v array**

```
*(u32*)(v) = a;
*(u32*)(v+4) = b;
*(u32*)(v+8) = c;
*(u32*)(v+12) = d;
```

6 of 6 test cases executed and computed correctly!
Total cycle count in encryption function: 177955
Cycle count per test case in encryption function: 29659

Optimize the Application (Final Result)

Method 4: replace both of the GETU32() and PUTU32() with reverse_bytes()

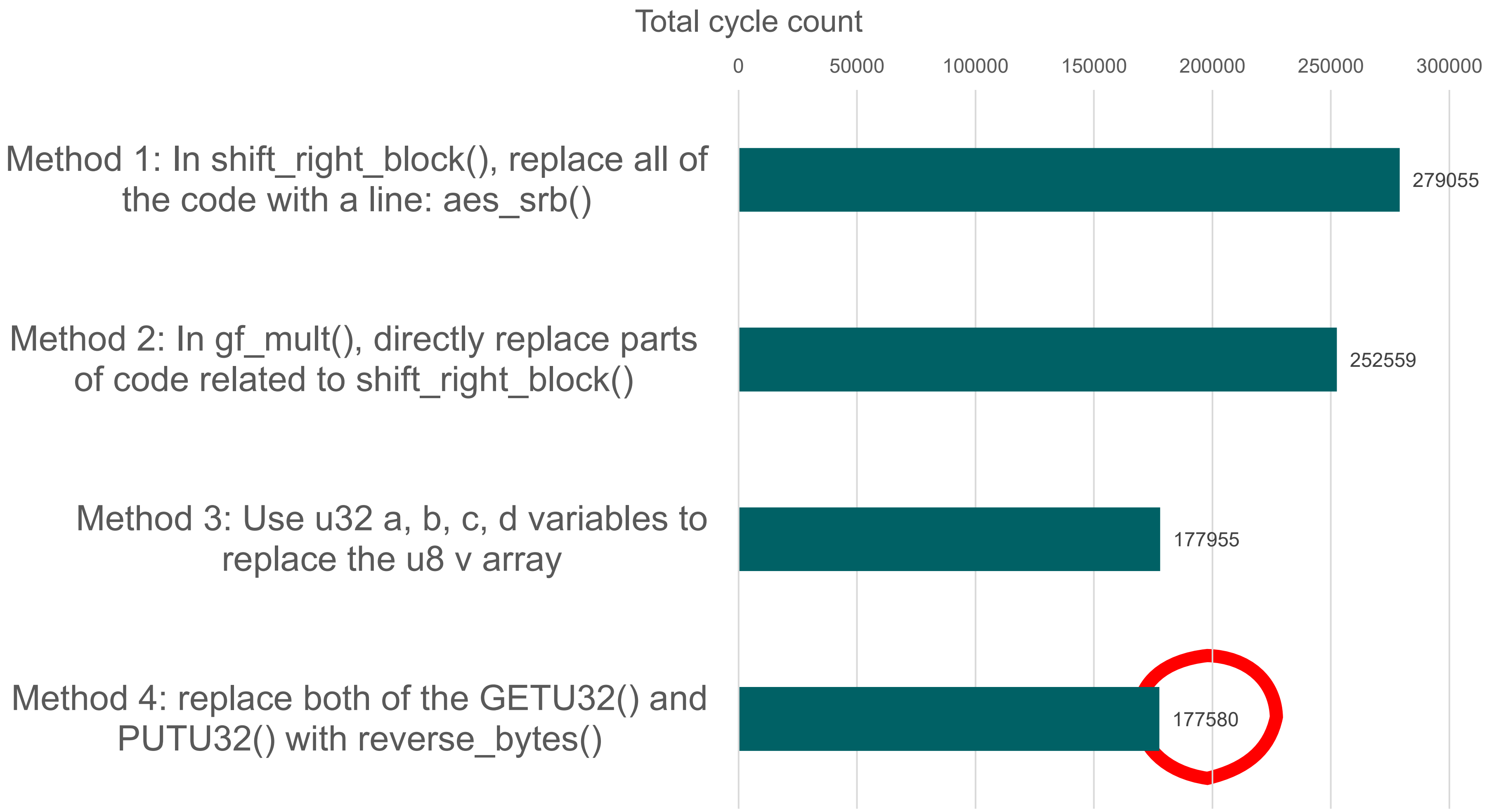
```
static void inc32(u8 *block) {
    u32 val;
    //val = GETU32(block + AES_BLOCK_SIZE - 4);
    val = reverse_bytes(*(u32*) (block + AES_BLOCK_SIZE - 4)); // add..
    val++;
    //PUTU32(block + AES_BLOCK_SIZE - 4, val);
    *(u32*) (block + AES_BLOCK_SIZE - 4) = reverse_bytes(val); // add..
}
```

A little improvement

We don't need the definition of GETU32 and PUTU32() anymore.

6 of 6 test cases executed and computed correctly!
Total cycle count in encryption function: 177580
Cycle count per test case in encryption function: 29596

Reach the Lowest Cycle Count



Do Logic Synthesis & AT diagram

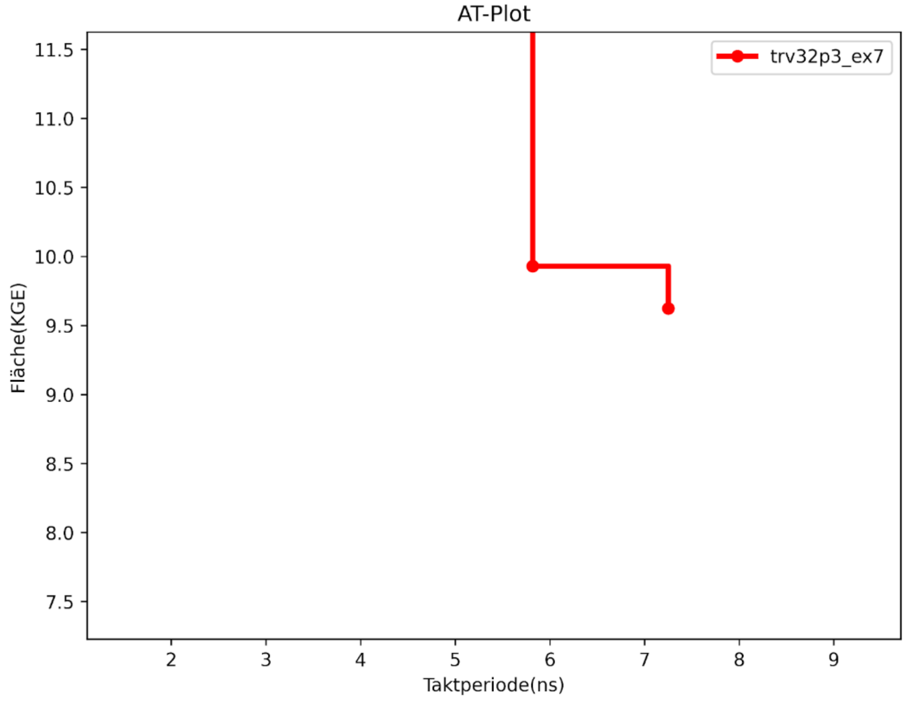
< print_all_results.py >

```
+------------------------------------+----------+-------+-------+--------+--------+--------+-------+-------+-------------+-------+-------+
| SynDcDir | Entity | kGE | Clk | Cstr | WNS | AT | RT | TAG | Flow | Job | Mem |
|------------------------------------+----------+-------+-------+--------+
| dc/trv32p3_c2.11_ex3_step | trv32p3 | 11.34 | 2.11 | 2.1
| dc/trv32p3_c2.12_ex3_step | trv32p3 | 10.98 | 2.12 | 2.1
| dc/trv32p3_c2.12_ex3_revbyte | trv32p3 | 11.15 | 2.12 |
| dc/trv32p3_c2.1_ex3_revbyte | trv32p3 | 11.29 | 2.12 | 2
| dc/trv32p3_c2.1_ex4.2 | trv32p3 | 12.34 | 2.12 | 2.10 | -0
| dc/trv32p3_c2.1_ex3_step | trv32p3 | 11.60 | 2.13 | 2.10
| dc/trv32p3_c2.13_ex3_init | trv32p3 | 9.41 | 2.13 | 2.13 |
| dc/trv32p3_c2.11_ex3_init_step_rev | trv32p3 | 12.19 | 2
| dc/trv32p3_c2.15_ex3_init | trv32p3 | 9.40 | 2.15 | 2.15 |
| dc/trv32p3_c2.15_ex3_revbyte | trv32p3 | 11.29 | 2.15 |
| dc/trv32p3_c2.15_ex3_init_step_rev | trv32p3 | 12.11 | 2
| dc/trv32p3_c2.12_ex3_init_step_rev | trv32p3 | 12.47 | 2.18 | 2.12 |  2m | | std.default | | |
| dc/trv32p3_c2.1_ex3_init_step_rev | trv32p3 | 12.44 | 2.19 | 2.10 |  2m | | std.default | | |
| dc/trv32p3_c2.2_ex3_init | trv32p3 | 9.48 | 2.20 | 2.20 | 0.00 | 20. d.default | | |
| dc/trv32p3_c2.2_ex4.2 | trv32p3 | 12.12 | 2.20 | 2.20 | -0.00 | 26 std.default | | |
| dc/trv32p3_c2.1_ex5 | trv32p3 | 14.36 | 2.22 | 2.10 | -0.12 | 31 std.default | | |
| dc/trv32p3_c2.23_ex4.2 | trv32p3 | 12.04 | 2.23 | 2.23 | 0.00 m | | std.default | | |
| dc/trv32p3_c2.24_ex4.2 | trv32p3 | 12.25 | 2.24 | 2.24 | 0.00 h 1m | | std.default | | |
| dc/trv32p3_c2.21_ex4.2 | trv32p3 | 12.55 | 2.25 | 2.21 | -0. | 0h 1m | | std.default | | |
| dc/trv32p3_c2.25_ex3_init | trv32p3 | 9.02 | 2.25 | 2.25 | .28 | 0h 0m | | std.default | | |
| dc/trv32p3_c2.25_ex4.2 | trv32p3 | 12.00 | 2.25 | 2.25 | 26.99 | 0h 1m | | std.default | | |
| dc/trv32p3_c2.2_ex5 | trv32p3 | 13.59 | 2.27 | 2.20 | -0 30.80 | 0h 1m | | std.default | | |
| dc/trv32p3_c2.25 | trv32p3 | 8.50 | 2.28 | 2.25 | -0.03 .35 | 0h 0m | | std.default | | |
| dc/trv32p3_c2.3_ex5 | trv32p3 | 13.49 | 2.30 | 2.30 | 00 | 31.03 | 0h 1m | | std.default | | |
| dc/trv32p3_c2.3_ex3_revbyte | trv32p3 | 10.85 | 2. 0 | 2.30 | 0.00 | 24.95 | 0h 1m | | std.default | | |
| dc/trv32p3_c2.3_ex4.2 | trv32p3 | 12.05 | 2.30 | 2.30 | 0.00 | 27.71 | 0h 1m | | std.default | | |
| dc/trv32p3_c2.33_ex7 | trv32p3 | 14.18 | 2.33 | 2.33 | -0.00 | 33.04 | 0h 1m | | std.default | | |
| dc/trv32p3_c2.34_ex7 | trv32p3 | 14.10 | 2.34 | 2.34 | 0.00 | 33.00 | 0h 1m | | std.default | | |
| dc/trv32p3_c2.11_ex3_revbyte | trv32p3 | 11.49 | 2.35 | 2.11 | -0.24 | 26.97 | 0h 1m | | std.default | | |
| dc/trv32p3_c2.36_ex7 | trv32p3 | 13.96 | 2.36 | 2.36 | 0.00 | 32.93 | 0h 1m | | std.default | | |
| dc/trv32p3_c2.38_ex7 | trv32p3 | 13.72 | 2.38 | 2.38 | 0.00 | 32.66 | 0h 1m | | std.default | | |
| dc/trv32p3_c2.2_ex7 | trv32p3 | 14.18 | 2.39 | 2.20 | -0.19 | 33.89 | 0h 2m | | std.default | | |
| dc/trv32p3_c2.4_ex3_init_step_rev | trv32p3 | 11.89 | 2.40 | 2.40 | 0.00 | 28.54 | 0h 1m | | std.default | | |
| dc/trv32p3_c2.4_ex7 | trv32p3 | 13.86 | 2.40 | 2.40 | 0.00 | 33.26 | 0h 1m | | std.default | | |
| dc/trv32p3_c2.3_ex7 | trv32p3 | 13.96 | 2.41 | 2.30 | -0.11 | 33.59 | 0h 1m | | std.default | | |
| dc/trv32p3_c2.31_ex7 | trv32p3 | 14.34 | 2.41 | 2.31 | -0.10 | 34.60 | 0h 1m | | std.default | | |
| dc/trv32p3_c2.45 | trv32p3 | 8.31 | 2.45 | 2.45 | 0.00 | 20.36 | 0h 0m | | std.default | | |
| dc/trv32p3_c2.4_ex3_revbyte | trv32p3 | 10.95 | 2.48 | 2.40 | -0.09 | 27.21 | 0h 1m | | std.default | | |
```

2.34 ns is the Min Clock Period
:= Max Clock Frequency

without negative slack !

AT diagram



Why does the diagram look abnormal?

The range of Taktperiode is set in 2.3 - 7.5 (ns), and the range of Fläche is set in 9.0 -15.0 (KGE). These x and y-axis constraints are based on the **sha256** algorithm. As for the AES algorithm, only a few of the values can be present.

My best Performance

The best total cycle count is 177580 by Method 4.

=> Max frequency from Logic Synthesis: 2.34 ns/period = 427.35 MHz

Total Computation Time = Total Cycle Count * Cycle_Period
= 177580 * 2.34 = **415,537 ns** **(much lesser than 702,000 ns)**

4 Optimization methods	Total cycle count	Cycle count per test case
Method 1: In shift_right_block(), replace all of the code with a line: aes_srb()	279055	46509
Method 2: In gf_mult(), directly replace parts of code related to shift_right_block()	252559	42093
Method 3: Use u32 a, b, c, d variables to replace the u8 v array	177955	29659
Method 4: replace both of the GETU32() and PUTU32() with reverse_bytes()	**177580**	29596
Min Clock Period / Max Clock frequency	2.34 ns/period = 427.35 MHz/period	

Thank you for your attention!

Pei-Yu Lin