MIPS Assembly Language Implementation of GIFT-64-128 Encryption

William Diehl

George Mason University, Fairfax, VA, 22030, USA
wdiehl@gmu.edu

Abstract. The GIFT-64-128 block cipher encryption is implemented in MIPS assembly language. The program is assembled and simulated using the QtSPIM simulator and produces functionally correct results. This implementation requires 22,764 clock cycles per 64-bit block encryption, as well as 1,296 bytes of code, and 192 bytes of data memory. The major functional units of the implementation are analyzed in terms of cycle count and bytes of code.

Keywords: GIFT, Encryption, Block Cipher, MIPS, Assembly Language

1 Introduction

Most modern software is written in high-level languages, such as C, C++, Java, or Python. Compilers and production tools have improved to the point that the tedious work of assembly language coding is rarely required. However, in cases where we wish to attain the maximum possible performance, or study in-depth the instruction sequences or architectural components which realize maximum performance (or which are roadblocks), we program in assembly language.

Cryptography is a required element of security in nearly all modern information technology. In particular, block ciphers are useful for realizing the cryptographic service of confidentiality, i.e., preventing plaintext from being read by any party without the corresponding secret key. Current research is ongoing to develop cryptographic block ciphers, which have good performance, but which are also lightweight i.e., are low-resource or energy-efficient, in both hardware in software.

In this research, we investigate an assembly language implementation of one block cipher, GIFT-64-128, using the MIPS instruction set architecture (ISA). Our MIPS assembly language implementation of GIFT-64-128 encryption is assembled and executed in the QtSPIM simulator and is, to the best of our knowledge, the first implementation of GIFT in the MIPS architecture. We first design the program based on the GIFT specification, then verify functionality in the QtSPIM simulator, and finally analyze the resulting code from the standpoint of performance (clock cycles) and required resources (code and data memory). The MIPS assembly code is included as an appendix at the end of this paper.

2 Background

2.1 GIFT Cipher

The GIFT cipher is a lightweight block cipher described in [1]. Descended from the PRESENT cipher [2], it is designed primarily for energy-efficient hardware implementations. GIFT is a basic substitution-permutation cipher; in this research we implement GIFT 64-128, which is encryption on a 64-bit block of plaintext using a 128-bit secret key. Decryption is similar (through in reverse order) and is not considered in this work. GIFT consists of a substitution layer with 4-bit S-Boxes (SubCells), a 64-bit bitwise permutation (PermCells), and mixing of round key and round constants into the state in each round (AddRoundKey). The key is updated per KeySchedule once per round, and after the round key is provided to the state. 28 rounds are required for one block encryption in GIFT-64-128. The basic flow of the GIFT cipher is shown in Fig. 1.

Since its introduction, GIFT has been investigated in both hardware and software implementations, e.g., [3 – 6]. It has also been incorporated as the primitive in lightweight authenticated encryption with associated data in the form of GIFT-COFB [7]. Hardware and software implementations of GIFT-COFB have likewise been investigated in [8 – 10].

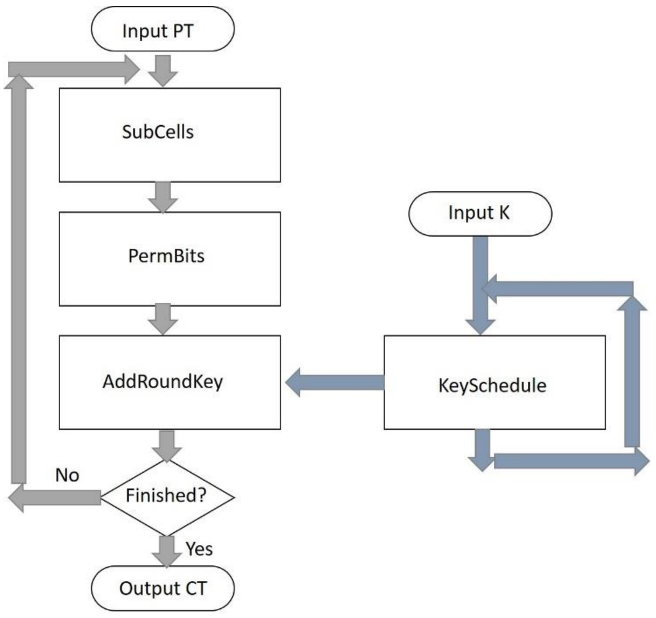

Fig. 1. GIFT basic encryption flow, including SubCells, PermBits, AddRoundKey, and KeySchedule. PT is Plaintext, CT is ciphertext, and K is the secret key.

2.2 MIPS

MIPS (Microprocessor without Interlocked Pipelined Stages) is a widely-known RISC (reduced instruction set computer) microprocessor [11]. It is a load-store architecture with 32 general purpose registers and can be tightly coupled with accelerators such as a floating point (FP) coprocessor. Instructions are word-aligned to processor data bus width (e.g., 32 bit) boundaries, and operations are permitted on byte, half word, and word quantities. Although emphasis on general purpose open-source RISC architecture has generally transitioned to the RISC-V, MIPS remains an important benchmark for performance, and enjoys wide use in computer architecture textbooks and university curricula [12]. Cryptographic functionality has been previously demonstrated in MIPS; examples are [13 – 17].

QtSPIM is a popular open-source assembly and simulator environment for a 32-bit version of MIPS and can be downloaded from [18]. The GIFT implementation in this research uses the QtSPIM environment and conventions.

3 Implementation

A simplified and generic MIPS assembly language implementation of GIFT-64-128 encryption using the QtSPIM assembler and simulator environment is produced. The source code is shown in the appendix. The program iteratively loops through 28 rounds. Each round consists of SubCells, PermBits, AddRoundKey, KeySchedule, and minimal ancillary control instructions. PermBits and AddRoundKey each call subroutines; there are no nested subroutine calls. S-Boxes and round constants are pre-stored in look up tables, as discussed below.

4 Analysis

The MIPS assembly language implementation is assembled and simulated in the QtSPIM simulator. All default options of QtSPIM are enabled. Results of block encryptions are verified against test vectors in [19] and are functionally correct. Analytic results are shown in Table 1 in terms of clock cycles and corresponding bytes of code in the .text segment. The function with both the longest run time and largest code usage is PermCells. In fact, PermCells requires 355 clock cycles per round, representing 44% of total required cycles. This is expected as GIFT (like its predecessor PRESENT) has a complex bitwise permutation function which is difficult to render using general purpose instructions such as shifts and rotates. Use of instruction set architectures (ISA) with custom or programmable permutators (e.g., RISC-V), or innovative methodologies (e.g., [3]) can improve performance. The second most complex function in terms of both run time and code size is "Extract Round Key," i.e., generating the round key $U||V$ as defined in [1] and adding it to state. The round key generation likewise is dependent on shifts and rotations, and is more easily implemented in hardware.

SubCells is computed by pre-storing 4-bit S-Boxes in the .data segment, requiring 16 bytes. This can be compacted into 8 total bytes, or computed on-the-fly, but at the cost of higher computational complexity. Round constants are retrieved from a pre-stored look up table, which requires 112 bytes of memory in the .data segment. Round constants can be generated on-the-fly using methods described in [1] but at increased computational complexity. Additional functions are unremarkable.

A total of 813 clock cycles are required per round. As GIFT-64-128 consists of 28 rounds, a total of 22,764 cycles are required for one 64-bit block encryption. A total of 1,296 bytes, and 192 bytes of code (.text segment) and memory (.data segment) are required, respectively. The stack is used for subroutine calls, however, stack memory usage is negligible. System calls and housekeeping functions required by the QtSPIM environment are not considered.

Table 1. Analysis of functional units by clock cycles and bytes of code (text). PT is Plaintext.

Function	Subfunction	Cycles	Bytes
SubCells			
	Initialization	8	32
	Sbox (Per byte of PT)	16	64
	Sbox (8 bytes of PT)	128	
PermCells		355	816
AddRoundKey			
	Extract Round Key	267	152
	Add Round Constant	7	28
	Update PT in Memory	4	16
Key Schedule			
	Update Key State	33	128
	Update Key in Memory	8	80
Loop Control		3	12
Misc			32
Per Round		813	1296

5 Conclusion

The first-known GIFT 64-128 block cipher encryption in MIPS assembly language is achieved. The program is assembled and simulated in QtSPIM and achieves functionally correct block encryption results. This implementation of 64-bit block encryption requires 22,764 clock cycles, 1,296 bytes of code, and 192 bytes of data memory. The permutation function requires 44% of total clock cycles, and round key extraction is the second most costly function in terms of performance. Future work can consider innovative permutation processing structures to improve performance, and the optimized ratio of performance vs. code density should be explored. Results are generally instructive for the future generation of RISC architectures, such as RISC-V.

References

1. A Banik, S., Pandey, S.K., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y. (2017). GIFT: A Small Present. In: Fischer, W., Homma, N. (eds) Cryptographic Hardware and Embedded Systems – CHES 2017. CHES 2017. Lecture Notes in Computer Science(), vol 10529. Springer, Cham. https://doi.org/10.1007/978-3-319-66787-4_16.
2. A Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An ultra-lightweight block cipher. In Paillier, P., Verbauwhede, I., eds.: CHES 2007. Volume 4727 of LNCS., Springer, Heidelberg (September 2007) 450-466.
3. Adomnicai, A., Najm, Z., & Peyrin, T. (2020). Fixslicing: A New GIFT Representation: Fast Constant-Time Implementations of GIFT and GIFT-COFB on ARM Cortex-M. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020(3), 402–427. https://doi.org/10.13154/tches.v2020.i3.402-427.
4. Lara-Nino, C.A., Diaz-Perez, A., Morales-Sandoval, M. (2018). FPGA-Based Assessment of Midori and GIFT Lightweight Block Ciphers. In: , et al. Information and Communications Security. ICICS 2018. Lecture Notes in Computer Science(), vol 11149. Springer, Cham. https://doi.org/10.1007/978-3-030-01950-1_45.
5. Jamuna Rani, D., Emalda Roslin, S. Optimized Implementation of Gift Cipher. Wireless Pers Commun 119, 2185–2195 (2021). https://doi.org/10.1007/s11277-021-08325-2.
6. Gheorghe Pojoga and Kostas Papagiannopoulos. 2022. Low-latency implementation of the GIFT cipher on RISC-V architectures. In Proceedings of the 19th ACM International Conference on Computing Frontiers (CF '22). Association for Computing Machinery, New York, NY, USA, 287–295. https://doi.org/10.1145/3528416.3530996.
7. S. Banik, A. Chakraborti, A. Inoue, T. Iwata, K. Minematsu, M. Nandi, T. Peyrin, Y. Sasaki, S. Meng Sim, Y. Todo, “GIFT-COFB,” Cryptology ePrint Archive, Paper 2020/738.
8. Caforio, A., Collins, D., Banik, S., Regazzoni, F. (2022). A Small GIFT-COFB: Lightweight Bit-Serial Architectures. In: Batina, L., Daemen, J. (eds) Progress in Cryptology - AFRICACRYPT 2022. AFRICACRYPT 2022. Lecture Notes in Computer Science, vol 13503. Springer, Cham. https://doi.org/10.1007/978-3-031-17433-9_3.
9. B. Rezvani, F. Coleman, S. Sachin, W. Diehl, Hardware Implementations of NIST Lightweight Cryptographic Candidates: A First Look, Cryptology ePrint Archive, Paper 2019/824.
10. B. Rezvani, T. Conroy, L. Beckwith, M. Bozzay, T. Laffoon, D. McFeeters, Y. Shi, M. Vu, W. Diehl, “Efficient Simultaneous Deployment of Multiple Lightweight Authenticated Ciphers,” Cryptology ePrint Archive, Paper 2020/609.
11. David Patterson, John Hennessy, Computer Organization and Design MIPS Edition: The Hardware/Software Interface, 5th Edition (2013), Elsiver.
12. Turley, Jim. "Wait, What? MIPS Becomes RISC-V". Electronic Engineering Journal. Mar. 21, 2021.
13. Wang, X., Gordon, S.D., McIntosh, A., Katz, J. (2016). Secure Computation of MIPS Machine Code. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds) Computer Security – ESORICS 2016. ESORICS 2016. Lecture Notes in Computer Science(), vol 9879. Springer, Cham. https://doi.org/10.1007/978-3-319-45741-3_6.

14. Singh, Kirat, Biometric Based Network Security Using MIPS Cryptography Processor (May 9, 2015). Available at SSRN: https://ssrn.com/abstract=2777942 or http://dx.doi.org/10.2139/ssrn.2777942.
15. T. Hiscock, O. Savry and L. Goubin, "Lightweight Software Encryption for Embedded Processors," 2017 Euromicro Conference on Digital System Design (DSD), 2017, pp. 213-220, doi: 10.1109/DSD.2017.25.
16. H. Anwar, M. Daneshtalab, M. Ebrahimi, J. Plosila and H. Tenhunen, "FPGA implementation of AES-based crypto processor," 2013 IEEE 20th International Conference on Electronics, Circuits, and Systems (ICECS), 2013, pp. 369-372, doi: 10.1109/ICECS.2013.6815431.
17. Z. Zang, Y. Liu and R. C. C. Cheung, "Reconfigurable RISC-V Secure Processor And SoC Integration," 2019 IEEE International Conference on Industrial Technology (ICIT), 2019, pp. 827-832, doi: 10.1109/ICIT.2019.8755206.
18. J. Larus, QtSPIM, https://sourceforge.net/projects/spimsimulator/files/, Accessed: Oct. 8, 2022.
19. T. Peyrin, GIFT Block Cipher, https://github.com/giftcipher/gift, Accessed: Oct. 8, 2022.

Appendix

MIPS Assembly Source Code for GIFT-64-128 Encryption

```
# GIFT-64-128 Encrypt (gift64enc.asm)
# MIPS QtSPIM
# William Diehl
# 08-18-2022
# Encrypts one 64 bit block of plaintext (PT) to one 64 bit block of ciphertext (CT)
# Using 128 bit secret key (K)

.data

sbox: .byte 0x1, 0xa, 0x4, 0xc, 0x6, 0xf, 0x3, 0x9, 0x2, 0xd, 0xb, 0x7, 0x5, 0x0, 0x8, 0xe

#PT and CT are located in indata; PT is overwritten by CT

indata: .byte 0x10, 0x32, 0x54, 0x76, 0x98, 0xba, 0xdc, 0xfe
inpermdata: .space 8
outpermdata: .space 8

#Secret Key K is below.  K is overwritten by successive round key updates

inkey: .byte 0x10, 0x32, 0x54, 0x76, 0x98, 0xba, 0xdc, 0xfe
       .byte 0x10, 0x32, 0x54, 0x76, 0x98, 0xba, 0xdc, 0xfe

outkey: .space 16
roundkey: .word 0x00000008, 0x00000088, 0x00000888, 0x00008888
          .word 0x00088888, 0x00888880, 0x00888808, 0x00888088
          .word 0x00880888, 0x00808888, 0x00088880, 0x00888800
```

```
        .word 0x00888008, 0x00880088, 0x00800888, 0x00008880
        .word 0x00088808, 0x00888080, 0x00880808, 0x00808088
        .word 0x00080880, 0x00808800, 0x00088000, 0x00880000
        .word 0x00800008, 0x00000080, 0x00000808, 0x00008088
numbytes: .word 8

#main routine

.text
.globl main
.ent main
main:

    li $s2, 0 # round counter (round 0)

roundloop:

#start subcells
    la $t0, indata
    la $t1, sbox
    la $t2, inpermdata
    lw $t3, numbytes

sboxloop:

    lbu $t4, ($t0)
    andi $t4, $t4, 0x0f
    addu $t5, $t1, $t4
    lbu $t6, ($t5)
    lbu $t4, ($t0)
    srl $t4, $t4, 4
    addu $t5, $t1, $t4
    lbu $t7, ($t5)
    sll $t7, $t7, 4
    or $t7, $t7, $t6
    sb $t7, ($t2)

    sub $t3, $t3, 1
    addu $t0, $t0, 1
    addu $t2, $t2, 1
    bgt $t3, 0, sboxloop

#subcells complete
#start permcells
```

```
    la $t1, inpermdata
    la $t2, outpermdata
    li $t0, 0

    lw $a0, ($t1)
    jal perm1

    ror $t0, $t0, 8
    lw $a0, 4($t1)
    jal perm1

    ror $t0, $t0, 8
    lw $a0, ($t1)
    jal perm2

    ror $t0, $t0, 8
    lw $a0, 4($t1)
    jal perm2

    ror $t0, $t0, 8

    sw $t0, ($t2)

    li $t0, 0
    lw $a0, ($t1)
    jal perm3

    ror $t0, $t0, 8
    lw $a0, 4($t1)
    jal perm3

    ror $t0, $t0, 8
    lw $a0, ($t1)
    jal perm4

    ror $t0, $t0, 8
    lw $a0, 4($t1)
    jal perm4

    ror $t0, $t0, 8
    sw $t0, 4($t2)

# permcells complete
# start extract round key
```

```
    la $t0, inkey
    la $t1, outpermdata
    lw $s0, ($t1)  # lower state
    lw $s1, 4($t1) # upper state

    li $t2, 0
    li $t3, 0

    lbu $t2, ($t0) # k0l
    lbu $t3, 1($t0) # k0h

    jal extkeystate

    xor $s0, $s0, $t5
    xor $s1, $s1, $t6

    lbu $t2, 2($t0) # k1l
    lbu $t3, 3($t0) # k1h

    jal extkeystate

    sll $t5, $t5, 1
    sll $t6, $t6, 1

    xor $s0, $s0, $t5
    xor $s1, $s1, $t6

#extract round key complete
#start add round constant

    la $t2, roundkey
    addu $t2, $t2, $s2
    lw $t3, ($t2) # get the next round key
    xor $s0, $s0, $t3
    li $t3, 0x80000000
    xor $s1, $s1, $t3

#add round constant complete
#update indata with new state

   la $t0, indata
    sw $s0, ($t0)
    sw $s1, 4($t0)

#update key state
```

```
    la $t0, inkey
    la $t1, outkey

    lhu $t2, 2($t0)
    ror $t2, $t2, 2
    move $t3, $t2 # save $t2 in $t3
    srl $t2, $t2, 16 # shift upper 16 bits to lower 16 bits
    or $t3, $t2, $t3 # combine upper and lower halves to complete the rotate on 32-bit field
    sh $t3, 14($t1)

    lhu $t2, 0($t0)
    ror $t2, $t2, 12
    move $t3, $t2 # save $t2 in $t3
    srl $t2, $t2, 16 # shift upper 16 bits to lower 16 bits
    or $t3, $t2, $t3 # combine upper and lower halves to complete the rotate on 32-bit field
    sh $t3, 12($t1)

    lhu $t2, 14($t0)
    sh $t2, 10($t1)

    lhu $t2, 12($t0)
    sh $t2, 8($t1)

    lhu $t2, 10($t0)
    sh $t2, 6($t1)

    lhu $t2, 8($t0)
    sh $t2, 4($t1)

    lhu $t2, 6($t0)
    sh $t2, 2($t1)

    lhu $t2, 4($t0)
    sh $t2, 0($t1)

#put updated round key back in inkey

    lw $t2, ($t1)
    sw $t2, ($t0)

    lw $t2, 4($t1)
    sw $t2, 4($t0)

    lw $t2, 8($t1)
    sw $t2, 8($t0)

    lw $t2, 12($t1)
    sw $t2, 12($t0)

#update key state complete
#round loop control

    add $s2, $s2, 4
    blt $s2, 112, roundloop

#done
    li $v0, 10
    syscall
.end main

.globl perm1
.ent perm1
perm1:

    and $t3, $a0, 0x00000001
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000002
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000004
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000008
    or $t0, $t0, $t3 #target register

    and $t3, $a0, 0x00000010
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000020
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000040
    or $t0, $t0, $t3 #target register
    ror $a0, $a0, 4
    and $t3, $a0, 0x00000080
    or $t0, $t0, $t3 #target register

    jr $ra

.end perm1

.globl perm2
.ent perm2
perm2:

    ror $a0, $a0, 12
    and $t3, $a0, 0x00000001
    or $t0, $t0, $t3 #target register

    rol $a0, $a0, 12
    and $t3, $a0, 0x00000002
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000004
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000008
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 16
    and $t3, $a0, 0x00000010
    or $t0, $t0, $t3 #target register

    rol $a0, $a0, 12
    and $t3, $a0, 0x00000020
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000040
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000080
    or $t0, $t0, $t3 #target register
    jr $ra

.end perm2

.globl perm3
.ent perm3
perm3:

    ror $a0, $a0, 8
    and $t3, $a0, 0x00000001
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000002
    or $t0, $t0, $t3 #target register

    rol $a0, $a0, 12
    and $t3, $a0, 0x00000004
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000008
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 16
    and $t3, $a0, 0x00000010
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000020
    or $t0, $t0, $t3 #target register

    rol $a0, $a0, 12
    and $t3, $a0, 0x00000040
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000080
    or $t0, $t0, $t3 #target register

    jr $ra

.end perm3

.globl perm4
```

```
.ent perm4
perm4:

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000001
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000002
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000004
    or $t0, $t0, $t3 #target register

    rol $a0, $a0, 12
    and $t3, $a0, 0x00000008
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 16
    and $t3, $a0, 0x00000010
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000020
    or $t0, $t0, $t3 #target register

    ror $a0, $a0, 4
    and $t3, $a0, 0x00000040
    or $t0, $t0, $t3 #target register

    rol $a0, $a0, 12
    and $t3, $a0, 0x00000080
    or $t0, $t0, $t3 #target register

    jr $ra

.end perm4

.globl extkeystate
.ent extkeystate
extkeystate:

    li $t7, 8
    li $t5, 0
    li $t6, 0

extkeyloop:

    and $t4, $t2, 0x00000001
    or $t5, $t5, $t4
    ror $t5, $t5, 4

    and $t4, $t3, 0x00000001
    or $t6, $t6, $t4
    ror $t6, $t6, 4
    srl $t2, $t2, 1
    srl $t3, $t3, 1

    sub $t7, $t7, 1
    bgt $t7, 0, extkeyloop

    jr $ra

.end extkeystate
```