COSMIC Functional Size of ARM Assembly Programs

Ahmed Darwish[1] and Hassan Soubra[1]

[1]The German University in Cairo (GUC), New Cairo, Egypt,
ahmed.ahmeddarwish@student.guc.edu.eg
hassan.soubra@guc.edu.eg

Abstract. The COSMIC functional size measurement (FSM) method can be applied in different phases of software projects: early in the design phase or after the implementation has been delivered. Different software artifacts can be used to produce the COSMIC functional size of a piece of software: specification requirements, implemented code, etc. COSMIC has been used in different domains (e.g. Management Information Systems (MIS), Real-time Embedded Systems-RTES, etc.), and has been applied to different conceptual frameworks and programming paradigms. Assembly language is the lowest-level programming language designed for a specific type of processor executing machine code. Assembly can be compiled or interpreted from different high-level languages. ARM® processors account for 90% of those used in the mobile industry and controllers of IoT devices. In addition, 75% of the processors used in in-vehicle infotainment Advanced Driver-Assistance Systems (ADAS) are made by ARM. As a whole, they address 33% of the total addressable market. In this paper, we propose an FSM procedure based on COSMIC ISO 19761 to measure software artifacts expressed in ARM's base 32-bit Assembly code. An FSM automation prototype tool is also introduced.

Keywords: COSMIC FSM, ARM, Assembly Language, IoT, Compilers

1 Introduction

The world of technology is going through a new phase, and the workflow of software development is starting to take on a newer more efficient form. Due to the always changing requirements of the industry, newer programming languages with features more suited to the market keep springing up. In addition, older languages are often updated or used as a basis for newer languages with better specifications, lest they get ignored by a programming team.

Software Engineering is both a technical and a managerial process, where the best-suited programming language and style are picked by experts for building either a simple or a complex component of a project [1]. And to be able to plan these projects, it is necessary to be aware of the required effort and the estimated complexity of the system. This is where COSMIC software functional measurement [2] comes in. Functional size measurement (FSM) is a powerful tool for managing software projects and it provides an objective and quantitative base for management decisions. It can be applied either during the design or the implementation phase of a project. FSM also has the upper hand due to their language-agnostic rules and definitions, as well as their robustness against the difference in programming experience. They can even be applied to user requirements written in natural languages. This deems this type of measurement more objective.

The genericness of COSMIC's rules allows them to be easily adapted to different use cases. However, applying COSMIC to assembly languages, which has been largely ignored in literature, can be very useful. Any language, whether compiled or interpreted, or a hybrid of both, is bound to be represented in binary streams of 1s and 0s for a processing unit to execute [3]. Assembly Language is one layer of abstraction above this stream. That is why we argue that it is important to have measurement rules for assembly languages since having such rules would allow FSM measurement done in the implementation phase to be language-agnostic. Having an automated, generic measurement tool for implemented work would also provide a chance to detect any discrepancies between the expected software size and the actual one, in a smooth workflow. The increased demand of developing software at the lower levels of programming, which is a result of the recent IoT (Internet of Things) boom, is also a very strong incentive for automating low-level software measurement.

There are numerous microprocessor manufacturers in the market, many of which adhere to a specific base architecture and assembly language. ARM [4] provides a simplistic language due to its reduced instruction set computing (RISC) nature and it has a very high penetration ratio in the embedded systems industry. This paper presents the definition and design of a COSMIC FSM procedure for ARM-based devices. We are going to define the mappings of ARM's assembly to COSMIC's guidelines and rules, as well as reasons for chosen measurement granularity. We are also going to show an example of compiled machine code being measured using our method, dealing with different parts of a system. We believe that our approach can be generally applied to other architectures, with minor modifications.

The rest of the paper is organized in the following manner: Section 2 provides an exhaustive literature survey of different COSMIC FSM procedures using different programming and modelling languages, including attempts to implement tools which automate the measurement procedures. Section 3 contains overviews of the COSMIC method and the ARM microprocessor family. Section 4 presents our proposed measurement procedure, as well as an illustrative example using two different ARM instructions. Finally, sections 5 and 6 respectively present an automated measurement prototype and conclude the paper with a statement of our planned future work.

2 Related Work

Many approaches have been proposed in the literature to automate COSMIC-based measurements, for different types of input and in different domains. One of the most notable inputs are conceptual models, which have taken many different forms, and evolved throughout the years. One of the earliest attempts was undertaken by Jenner [5], who implemented a tool that measures the CFPs of a UML 1.0 model. Habela et al. [6] and Levesque et al. [7] then proposed manual procedures for the newer 1.5 and 2.0 versions of UML, respectively. Meiliana et al. [8] then went on to implement a tool that automates the measurement for models represented in XML. The previous works were concerned mainly with Management Information systems. Lind and Heldal [9] developed a tool using Java that estimates the functional points of embedded software represented in UML models. In the field of web development, Ceke and Milanisovic [10] developed a tool that takes web-oriented UML models as input. Similarly, Haoues et al. [11] proposed a procedure that could be applied to both Mobile and web applications. The latest work, done by De Vito et al. [12], consists of a generic automated estimation tool that is claimed to work accurately for any domain. More UML-related approaches are found in [13–15].

A lot of effort has also been made for other conceptual models. Diab et al. formalised the rules of applying to ROOM framework for real-time applications, and implemented a tool that works with RRRT models [16, 17]. Grau et al. [18] implemented a tool for the PRiM method. Abrahão et al.[16] implemented an automated measurement plug-in for VisualWADE, a popular tool for developing web applications using the Object-Oriented Hypermedia (OO-H) method.

Another kind of input that has been explored in literature is plain text. Ishrar et al. [20] were the first to carry out a statistical study, looking for correlations between the textual representation of the functionalities of an application and its COSMIC measurement. Their study encouraged further research in the area. Having a list of the use cases of an application, Ungan et al. [21] has been able to extract COSMIC measurements, by implementing an extension to the ScopeMaster® tool. In a very identical manner, Ecar et al. [22] estimated the measurements using User Stories (US).

Tools have also been developed for actual programming environments. Soubra et al. [23] implemented a tool that computes the CFP count of SimuLink files. Later on, the same authors [24] (with the exception of Sophie Stern) refined the procedure of the tool to account for variance in the account when the same functional requirement is implemented using different approaches. In the automotive industry, Soubra et al. [25] proposed a verification protocol for an FSM-automation tool for AUTOSAR-based software. In [26], an automated tool was developed by ESTACA for the LUSTRE-based SCADE development tool, and it was verified that its measurements match those done manually. For Management Information systems, Tarhan and Sağ [27] used execution traces from Java Business Apps to generate tagged representations of sequence diagrams, which were in turn used to automatically measure the functional size.

To the best of our knowledge, no attempts have been made to define rules and guidelines based on the COSMIC method, for any low-level assembly language. The closest thing to our approach was a work by Soubra and Abran who applied COSMIC rules to Arduino C, mapping COSMIC to the different components of an Arduino board [28].

3 Overviews of COSMIC and ARM Architecture

This section presents overviews of COSMIC and ARM architecture.

3.1 COSMIC Overview

COSMIC is an ISO-standardised method of quantifying FURs. It defines how to decompose the system into layers and how to differentiate between the various movements of data in a system according to what is called the boundaries of a Functional Process. It also dictates the guidelines of defining the granularity of the measurement. The measurement unit is coined a COSMIC Functional Point (CFP), and each data movement has the size of 1 CFP.

The four kinds of data movements according to COSMIC are as follows:-

1. Entry (E): This corresponds to a data group passing a boundary from a functional user **into** a functional process, as a reaction to some trigger.
2. Exit (X): This corresponds to a data group crossing a boundary and moving **outside** the functional process.
3. Read (R): Data is read from some persistent storage into the currently running functional process that requires it.
4. Write (W): Data is written to some persistent storage within the reach of the currently running functional process.

Persistent storage is defined as storage with means of storing data after a functional process terminates and/or of allowing a process to retrieve data that has been manipulated by some other functional process or another occurrence of the same process.

COSMIC rules can not be applied directly to measure the size of FURs; two other stages need to be carried out first:-

1. Measurement Strategy Phase, in which the scope and the purpose of the measurement is defined. This is done by applying the COSMIC Software Context Model.
2. Mapping phase, where the measurement rules of COSMIC are mapped and defined for the domain being measured.

After the measurement takes place, the final size of software is calculated by summing up the sizes of all the functional processes present within the defined scope. The FSM procedure proposed in this paper is based on version 4.0.2 of COSMIC.

3.2 ARM Overview

ARM is a family of RISC architectures for computer processors, based on the A32 Instruction Set Architecture -ISA, and its variations, the A64 and the T32 ISAs. The A64 ISA, the most recent addition, was introduced in tandem with the new ARMv8-A microarchitecture which supports 64-bit memory addressing. To maximise code density, T32 ISA have variable instruction lengths. To meet the typical need for floating point operations, an extension ISA exists. Other than that, additional ISA extensions are available for domains such as Machine Learning, which provide increased parallelism.

As of 2019, ARM processors account for 90% of those used in the mobile industry and controllers of IoT devices. In addition, 75% of the processors used in in-vehicle infotainment Advanced Driver-Assistance Systems (ADAS) are made by ARM. As a whole, they address 33% of the total addressable market [29].

At the time of writing, three different families of chips are in production: the (A)pplication family which is targeted at high-performance general applications, the (R)eal-time family which is optimised for time-critical and real-time domains, and finally the (M)icrocontroller family [4].

The register file in an ARM core consists of 31 registers, whose sizes are either 32- or 64-bits according to the version of the processor. Of those registers, 16 are always visible to the user, while the remaining are used to store program states, such as the program counter, and the stack pointer, and handle exceptions. As for memory, different processors have different levels of caching, and some have none at all, while providing a standard interface to an external memory [30].

One defining feature of ARM instructions is the ability to conditionally execute commands by setting a specific field to the required the status of the system that the instruction must run in. If the condition is not met, an instruction is treated as a No Operation (NOP) instruction. This is more efficient than relying on branch instructions.

4 An FSM Procedure for COSMIC Measurement on ARM

To correctly measure the functional size of a program represented in ARM instructions, we have to define the scope and the purpose of our measurement, and then we have to map COSMIC terms to ARM components.

4.1 The Measurement Strategy Phase

The purpose of this procedure is to apply the COSMIC method to the compiled ARM assembly code. As to the scope, it is at the hardware level of a computing entity, be it a stand-alone processor, or a separate core in a multi-core system. Since we are working at such a low level, the granularity here would be at the processor instruction level, since each instruction carries out a distinctive operation that affects the state of the system.

We consider the decoding circuitry (DC) to be our sole functional user. We say the decoding circuitry is the hardware unit responsible for extracting the different parameters from the fetched instruction, and analysing it so that the proper hardware signals, required for executing the instruction correctly, may be fired.

As for the persistent storage, we consider the register file, any caches present, and any existing co-processor register files/memories to be common persistent storage units for any ARM processor. In addition, if a processor is complying with the Harvard architecture, then the Data Memory is also a part of the persistent storage. However, if the processor is based on von Neumann's, such as pre-ARM9 processors [31], then we consider only the physical spaces storing data to be part of it.

4.2 The Mapping Phase

Our mapping is based on our abstract view of instructions, which was inspired from the official documentation of ARM's ISA. In our model, we view each instruction simply as a subroutine carrying out a specific operation, see figure 1. It must be stressed that this is not to be confused with the commonly known idea of an assembly subroutine which usually involves a branch and ending in a return. In our representation, every single instruction (including the branch and the return instructions) is a subroutine, carrying out a certain set of operations to achieve an intended change in the state of the system.

```
void instruction(halfbyte conditionField, {boolean S}, param1, param2,
    ...){
    // Optional status register check
    if(statusRegister[conditionBits] != conditionField)
        return;

    // Details of the instruction go here

    // Optional status register update
    // (for arithmetic instructions only)
    if(S == true)
        updateStatusRegsiter()

    return;
}
```

Figure 1. The abstract format of a single instruction

Moreover, for an instruction to be executed, the hardware needs to determine certain operands from the decoded instruction. The operands would naturally differ according to the nature of the instruction, and they can include information about the specific index or label of the needed data in the persistent storage. Therefore, we perceive the operands as entries coming from the instruction decoder to the subroutine, which is then later used to carry out the intended job, by using the operands to read a register, write to memory, or carry out an arithmetic operation, etc. It should be noted that due to the instruction format of ARM instructions, a Read can take place at the beginning of the execution of an instruction, where the condition field is compared to the status register bits that correspond to the needed comparison.

So based on the aforementioned idea, each instruction stored in the instruction memory is considered a separate independent functional process, which is triggered by fetching the instruction from the instruction memory according to the Program Counter, and passing it to the DC. This would be the Triggering Event. Of the extracted parameters, the condition field, shown in figure 1, is the Triggering Entry, since it is common for all instructions. This Entry will be followed by others carrying the operands and signals as explained before.

We define the start of the functional boundary for incoming data groups to be the circuitry connecting the instruction decoding mechanism (i.e. the DC) and the execution circuitry, which can, but need not, contain a mixture of ALUs, shifters, and/or sign extenders. Naturally, the boundary will contain the register file where register contents will be fetched according to the instruction operands and other persistent storage. The boundary also encapsulates the circuitry required for reading and writing from the other persistent storage members. This definition is arbitrary enough for and can be applied to any processor having a decoding and an execution stage in its pipeline.

The mappings of the COSMIC rules regarding data group movements into ARM terms are summarized in table 1.

4.3 The Measurement Phase

Using the rules defined in **Section 4.2** to measure the size in CFP of every instruction, we aggregate the results coming from all instructions at the end of the execution of our program. For some instructions, such as the ADC (Add with Carry) instruction, the CFP count of the actions carrying out the instruction's objective will always be the same for any instance of the instruction, no matter the value of the operands. In other words, apart from the optional condition field check, the CFP will be the same. On the other hand, other instructions, such as the PUSH instruction, would have different counts for different instances. The reason for this would be the nature of the parameters of the instruction, which can lead to different amounts of data movements based on their values. This is elaborated in the example below. Hence, we segregate ARM instructions into two types: fixed size and variable size instructions, from a COSMIC FSM point of view.

To show how an instruction is broken down to be measured, figures 2 and 3 shows examples of breaking down an instruction into measureable form. Both examples borrow the parameter names from the syntax definition of the corresponding instructions in ARM's ISA documentation. The CFP count of ADC instruction, where Operand2 is an immediate, not an optionally-shifted register value, is always 8 CFP, while the CFP count of PUSH instruction varies according to the *reglist* parameter. *reglist* is a mask that determines which registers will be pushed to the stack, as shown in the pseudo-code in the figure. Therefore, the CFP would vary based on the cardinality of *reglist*.

Table 1. Mapping COSMIC Data Group Movement rules to ARM Terms

COSMIC Data Group	Equivalent in ARM
Entry data group movements	Parameters required by the instruction to carry out its ISA-defined operation, such as names of operand registers, immediate values, memory addresses, or bit masks.
Exit data group movements	Instructions never move data outside the boundary, and are considered to systemically terminate. Consequently, all functional processes have an exit CFP count of 0.
Read data group movements	The reading of data from the persistent storage as per the parameters passed to the functional process
Write data group movements	The writing of data to persistent storage after the data processing unit carries out the required data manipulations

The status flags corresponding to the results of arithmetic operations are not automatically updated after each instruction in ARM. Instead, an optional identifier, **S**, has to be included in the instruction for this to happen. Therefore, an additional Write (1 CFP) updating the status register may take place depending on the presence of that identifier.

5 Prototype of an Automated Measurement Tool

In this section, we present a prototype tool for a program expressed in ARM assembly language.

5.1 Data Preparation

To retrieve a sample of machine code that can easily be analyzed and processed to carry out the proposed FSM procedure, we wrote a simple program in C as shown in figure 4. The C program contains two functions: square() and factorial(), of which the latter is recursive. The C file was compiled using GCC on a Raspberry Pi® 1 B+ board. The board runs on a Broadcom BCM2835 processor running an ARM ARM1176JZF-S™ core. This allowed us to have a binary

```
void ADC(halfbyte conditionField, boolean S, int rd, int rn, int
     Operand2){ ← 5 Entries
     if(statusRegister[conditionBits] != conditionField) ← 1 Read
         return;                                          (optional)

     tmp a = RegisterFile[rt]; ← 1 Read
     RegisterFile[rd] = a + Operand2 + statusRegister[CarryFlag]; ← 1
                                                            Write, 1 Read

     if(S == true)
         updateStatusRegister(); ← 1 Write (optional)

     return;
}
```

Figure 2. The ADC instruction represented as a subroutine

```
void PUSH(halfbyte conditionField, short reglist){ ← 2 Entries
     if(statusRegister[conditionBits] != conditionField) ← 1 Read
         return;                                          (optional)

     for i = 0 till 15:
         if reglist[i] == 1:
             Memory[address] = RegisterFile[i] ← 1 Read,
                                               1 Write (per register)
             address = address + 4

     RegisterFile[SP] = RegisterFile[SP] - 4*BitCount(reglist);← 1 Read,
                                                                 1 Write
     return;
}
```

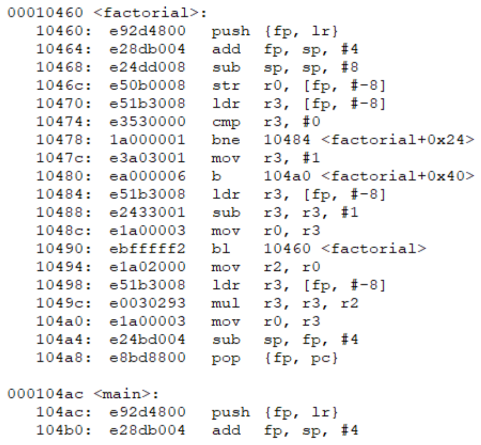
Figure 3. The PUSH instruction represented as a subroutine

executable in ARM machine language. The next step was to translate the binary file to a human-readable file that can be used to analyze the instructions more easily. This is where GNU's `objdump` tool came in. We used this tool to disassemble the binary file, separating the ARM instructions according to the subroutines, see figure 5. The `objdump` returns an instruction in the form of {hexadecimal representation, the natural language name of the instruction, list of needed operands}. This form adheres to ARM's assembly syntax.

```
#include <stdio.h>

int square(int x){ return x * x; }
int factorial(int x){
     if(x == 0){ return 1; }
     return x * factorial(x - 1);
}
int main(){
     int x = 5;
     x = x + 9;
     int y = square(x);
     y = factorial(y);
     printf("%d\n", y);
     return 0;
}
```

Figure 4. The sample C program used



```
00010460 <factorial>:
   10460:  e92d4800   push {fp, lr}
   10464:  e28db004   add  fp, sp, #4
   10468:  e24dd008   sub  sp, sp, #8
   1046c:  e50b0008   str  r0, [fp, #-8]
   10470:  e51b3008   ldr  r3, [fp, #-8]
   10474:  e3530000   cmp  r3, #0
   10478:  1a000001   bne  10484 <factorial+0x24>
   1047c:  e3a03001   mov  r3, #1
   10480:  ea000006   b    104a0 <factorial+0x40>
   10484:  e51b3008   ldr  r3, [fp, #-8]
   10488:  e2433001   sub  r3, r3, #1
   1048c:  e1a00003   mov  r0, r3
   10490:  ebfffff2   bl   10460 <factorial>
   10494:  e1a02000   mov  r2, r0
   10498:  e51b3008   ldr  r3, [fp, #-8]
   1049c:  e0030293   mul  r3, r3, r2
   104a0:  e1a00003   mov  r0, r3
   104a4:  e24bd004   sub  sp, fp, #4
   104a8:  e8bd8800   pop  {fp, pc}

000104ac <main>:
   104ac:  e92d4800   push {fp, lr}
   104b0:  e28db004   add  fp, sp, #4
```

Fig. 5. A snippet from the output of the objdump tool

5.2 The Automated Measurement Tool

We used Python[2] to implement our automated measurement tool, and figure 6 shows the UI of the tool. The tool accepts as input a text containing `objdump`'s disassembly, object, or C file. The tool also gives the user the option to exclude C's standard headers from the measurement. For an accurate measurement of the CFP count of the input program, we comply with the syntax and the pseudo-code defined in ARM's official specifications of their 32-bit ISA, A32[3]. The ARM Quick Reference Card[4] was also used as a guide.

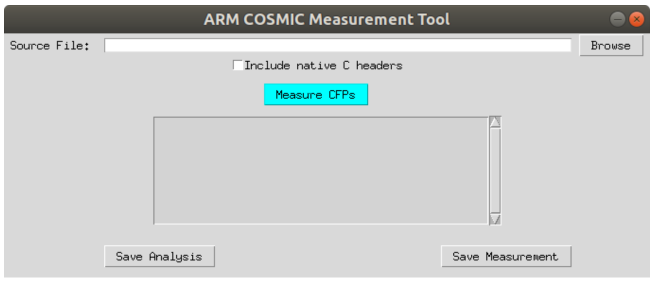


Fig. 6. The tool's UI

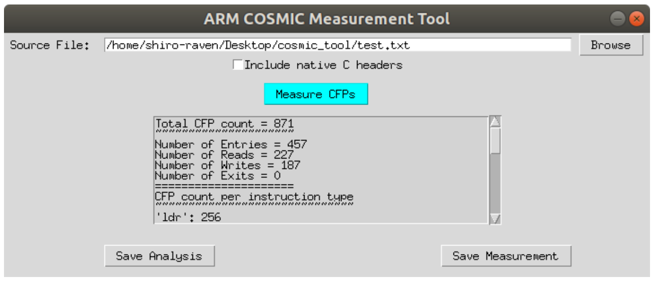
[2] https://www.python.org/
[3] https://developer.arm.com/docs/ddi0597/h
[4] https://www.ee.ryerson.ca/ courses/coe718/Data-Sheets/ARM/IS-arm.pdf

We start by using a regular expression to preprocess and filter out non-instruction elements of the file, mainly assembly labels of different sections in the code. After retrieving the list of instructions, we carry out the following steps:

1. For each fixed-size instruction, that always has the same functional size, we count its number of occurrences. We then multiply the occurrences by its size in CFP.
2. For variable size instructions, we further analyze the operands of each occurrence of such instructions to determine its exact COSMIC functional size.
3. For pseudo-instructions and aliases, we translate the instruction to an equivalent list of ARM instructions and use these resulting instructions for the measurement.

For our prototype, we only consider a subset of instructions, which are the ones present in our sample file. We plan to include all the instructions in the final version of our prototype tool.

Our prototype tool returns the total CFP count of the input file with its breakdown into the different types of data movements, as shown in figure 7. It also returns the count of each unique instruction in the program. The user can save the output as two different. The first of them is the functional size in CFP of the whole program, as well as the count of each unique instruction in the examined program. This is depicted in figure 8.



Source File: /home/shiro-raven/Desktop/cosmic_tool/test.txt

Include native C headers

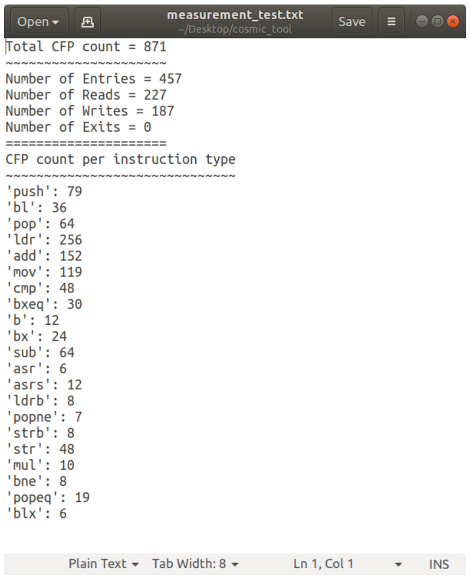
```
Total CFP count = 871
~~~~~~~~~~~~~~~~~~~~~
Number of Entries = 457
Number of Reads = 227
Number of Writes = 187
Number of Exits = 0
=====================
CFP count per instruction type
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'ldr': 256
```

Fig. 7. The UI with an output

The second output is a Comma-Separated Values (CSV) file, containing a list of the hexadecimal codes, the instruction names, the operands, as well as the individual CFP count of each instruction. figure 9. shows the CSV output file.

6 Conclusion

FSM is used to estimate development effort, manage project scope changes, measure productivity, benchmark, and normalize quality and maintenance ratios.



```
Total CFP count = 871
~~~~~~~~~~~~~~~~~~~~~
Number of Entries = 457
Number of Reads = 227
Number of Writes = 187
Number of Exits = 0
=====================
CFP count per instruction type
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'push': 79
'bl': 36
'pop': 64
'ldr': 256
'add': 152
'mov': 119
'cmp': 48
'bxeq': 30
'b': 12
'bx': 24
'sub': 64
'asr': 6
'asrs': 12
'ldrb': 8
'popne': 7
'strb': 8
'str': 48
'mul': 10
'bne': 8
'popeq': 19
'blx': 6
```

Fig. 8. The saved output file

COSMIC ISO 19761 is considered a second-generation FSM method that is designed to be independent of any implementation decisions embedded in the operational artifacts of the software to be measured.

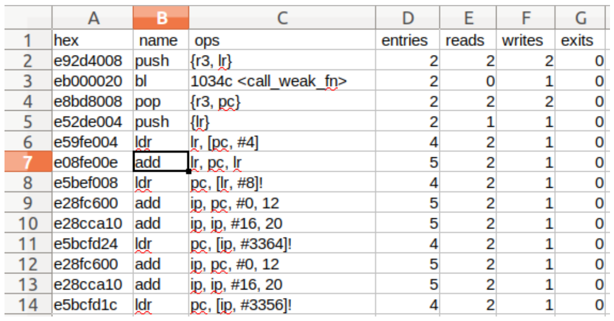
In this paper, we proposed an application of the COSMIC method to an assembly language. We explained how the COSMIC measurement methods could be used to measure the functionality of compiled ARM programs, with an illustrative example and we introduced an automated measurement tool prototype that can easily produce the functional size in CFP of an ARM program.

We believe that our work would influence the embedded systems industry, especially with the present boom in the domain of IoT and portable devices. In addition, we believe that our work would be useful to the compiler construction field, as the CFP count can provide insight into the difference of the efficiencies of two similar compilers.

In the future, we hope to apply our technique to a Complex Instruction Set Computer (CISC) language, and carry out a study to compare sizes in CFP as code is compiled into another.

References

1. Sommerville, Ian., Software Engineering, Addison Wesley; 2006
2. https://cosmic-sizing.org/
3. Compilers: Principles, Techniques, and Tools by Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman - Second Edition, 2007
4. https://www.arm.com/



	A	B	C	D	E	F	G
1	hex	name	ops	entries	reads	writes	exits
2	e92d4008	push	{r3, lr}	2	2	2	0
3	eb000020	bl	1034c <call_weak_fn>	2	0	1	0
4	e8bd8008	pop	{r3, pc}	2	2	2	0
5	e52de004	push	{lr}	2	1	1	0
6	e59fe004	ldr	lr, [pc, #4]	4	2	1	0
7	e08fe00e	add	lr, pc, lr	5	2	1	0
8	e5bef008	ldr	pc, [lr, #8]!	4	2	1	0
9	e28fc600	add	ip, pc, #0, 12	5	2	1	0
10	e28cca10	add	ip, ip, #16, 20	5	2	1	0
11	e5bcfd24	ldr	pc, [ip, #3364]!	4	2	1	0
12	e28fc600	add	ip, pc, #0, 12	5	2	1	0
13	e28cca10	add	ip, ip, #16, 20	5	2	1	0
14	e5bcfd1c	ldr	pc, [ip, #3356]!	4	2	1	0

Fig. 9. Part of the Output CSV file (examined in LibreOffice Calc)

5. Jenner, M.S.: COSMIC-FFP and UML: Estimation of the Size of a System Specified in UML – Problems of Granularity. In: 4th European Conference on Software Measurement and ICT Control, Heidelberg, pp. 173–184 (2001)
6. Habela, P., Glowacki, E., Serafinski, T., Subieta, K.: Adapting Use Case Model for COSMIC-FFP Based Measurement. In: 15th International Workshop on Software Measurement – IWSM 2005, Montréal, pp. 195–207 (2005)
7. Levesque, G., Bevo, V., Cao, D.T.: Estimating software size with UML models. In: Proceedings of the 2008 C3S2E Conference, Montreal, pp. 81–87 (2008)
8. Meiliana, et al. “Automating Functional and Structural Software Size Measurement Based on XML Structure of UML Sequence Diagram.” 2017 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom), 2017, pp. 24–28.
9. Lind, Kenneth, et al. “CompSize: Automated Size Estimation of Embedded Software Components.” 2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement, 2011, doi:10.1109/iwsm-mensura.2011.49.
10. Ceke, Denis, and Boris Milasinovic. “Automated Web Application Functional Size Estimation Based on a Conceptual Model.” 2015 23rd International Conference on Software, Telecommunications and Computer Networks (SoftCOM), 2015, pp. 234–241.
11. Haoues, Mariem, et al. “A Rapid Measurement Procedure for Sizing Web and Mobile Applications Based on COSMIC FSM Method.” Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement On , 2017, pp. 129–137.
12. Vito, Gabriele De, et al. “Design and Automation of a COSMIC Measurement Procedure Based on UML Models.” Software and Systems Modeling, vol. 19, no. 1, 2020, pp. 171–198.
13. Barkallah, S., Gherbi, A., & Abran, A. (2011). COSMIC Functional Size Measurement Using UML Models. FGIT-ASEA/DRBC/EL.
14. Lavazza, L., & Robiolo, G. (2010). Introducing the evaluation of complexity in functional size measurement: a UML-based approach. ESEM ’10.
15. Lavazza, L., & Bianco, V.D. (2009). A Case Study in COSMIC Functional Size Measurement: The Rice Cooker Revisited. IWSM/Mensura.
16. Diab, H., Frappier, M., St-Denis, R.: Formalizing COSMIC-FFP Using ROOM. In: ACS/IEEE International Conference on Computer Systems and Applications, Beirut (2001)

17. Diab, H., Koukane, F., Frappier, M., St-Denis, R.: μcROSE: Automated Measurement of COSMIC-FFP for Rational Rose Real Time. Information and Software Technology 47(3), 151–166 (2005)
18. Grau, G., Franch, X.: Using the PRiM method to Evaluate Requirements Model with COSMIC-FFP. In: Proceedings of the IWSM-MENSURA 2007, Mallorca, pp. 110–120 (2007)
19. Abrahão, Silvia, et al. “Definition and Evaluation of a COSMIC Measurement Procedure for Sizing Web Applications in a Model-Driven Development Environment.” Information & Software Technology, vol. 104, no. 104, 2018, pp. 144–161.
20. Hussain, Ishrar, et al. “Approximation of COSMIC Functional Size to Support Early Effort Estimation in Agile.” Data and Knowledge Engineering, vol. 85, 2013, pp. 2–14.
21. Ungan, Erdir, et al. “Automated COSMIC Measurement and Requirement Quality Improvement Through ScopeMaster® Tool.” IWSM-Mensura, 2018, pp. 1–13.
22. Ecar, Miguel, et al. “AutoCosmic: COSMIC Automated Estimation and Management Tool.” Proceedings of the XIV Brazilian Symposium on Information Systems, 2018, p. 61.
23. Soubra, Hassan, et al. “Design of a Functional Size Measurement Procedure for Real-Time Embedded Software Requirements Expressed Using the Simulink Model.” 2012 Joint Conference of the 22nd International Workshop on Software Measurement and the 2012 Seventh International Conference on Software Process and Product Measurement, 2011, pp. 76–85.
24. Soubra, Hassan et al. “A Refined Functional Size Measurement Procedure for Real-Time Embedded Software Requirements Expressed Using the Simulink Model.” 2012 Joint Conference of the 22nd International Workshop on Software Measurement and the 2012 Seventh International Conference on Software Process and Product Measurement (2012): 70-77.
25. Soubra, Hassan, et al. “Verifying the Accuracy of Automation Tools for the Measurement of Software with COSMIC – ISO 19761 Including an AUTOSAR-Based Example and a Case Study.” 2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement, 2014, pp. 23–31.
26. Soubra, Hassan, et al. Manual and Automated Functional Size Measurement of an Aerospace Realtime Embedded System: a case study based on SCADE and on COSMIC ISO 19761. 2015.
27. Tarhan, Ayça, and Muhammet Ali Sağ. “COSMIC Solver: A Tool for Functional Sizing of Java Business Applications.” Balkan Journal of Electrical and Computer Engineering, vol. 6, no. 1, 2018, pp. 1–8.
28. Soubra, Hassan and Alain Abran. “Functional size measurement for the internet of things (IoT): an example using COSMIC and the arduino open-source platform.” IWSM Mensura ’17 (2017).
29. ARM’s Q1 2019 Roadshow Presentation: https://www.arm.com/-/media/global/company/investors/PDFs/Arm_SBG_Q1_2019_Roadshow_Slides_FINAL.pdf
30. https://developer.arm.com/
31. Furber, Steve. ARM System-on-Chip Architecture. p. 344. ISBN 0201675196.