Software Synthesis for Embedded Processors

Yigitcan Aydin
Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
Email: yigitcan.aydin@stud.hshl.de

***Abstract*—21st century is all about computers and software. From agriculture to space research, from gaming to bank systems, etc. Everyday millions of computers in different types and sizes are being produced. For most of the people, it does not matter how computers work and provides us what we need and want in just seconds. How computers can do this? They are made of chips and programmers can write programs which is similar to human language and computer understands it. Throughout this paper, software synthesis will be discussed with all of its steps and phases in basic and comprehensible way.**

I. Introduction

Synthesizing software for any given type of processors has many different and complex levels as we go deeper in the computer organization. Usually programmers write the code and run them by only compiling or interpreting. However, it is not that straightforward to synthesize a software. Since the computers do not have the same understanding of the environment and have different methodologies to think about a process, conceptualize it and realize it. Therefore, in order to get how computers can come up with excellent representations of the software such as real-time critical systems, games, simulation programs and so on, we should be familiar with under the hood.

Firstly, a programmer begins with his/her journey by choosing a proper language for the job she needs it to get done. In our case let's take C language as our high-level programming language. By high-level language, we mean that it is the most similar language to the human language so far. After finishing writing the C code, we need to go one more level down to get closer to the machine language. Therefore, we need to use one of the C compilers, e.g. GCC, to translate the source code into the assembly language which is specific to the ISA (Instruction Set Architecture) which we will be discussing about later in the paper. Each ISA has its own assembly language but most of the times they are separated from each other with minor differences (if they are under the same instruction set category, i.e. RISC or CISC).

Once we have the assembly language, we are ready to generate the machine code which is the real language that computers can understand literally and by using an assembler, assembly code is translated into the machine code. The machine code consists of only binary numbers, i.e. 0s and 1s. After the assembler finishes its job, it gives the output as the object code which is also a machine executable code.

Since we are using some standard libraries and maybe some of the previously created special libraries which are already compiled and assembled into machine code, we need to combine them as we desire to be able to make them run together. This task is done by the linker. The linker combines the object code and the library routines an executable code which is also a machine code as well.

As a very last step for our program to be run properly, we need to load it into the memory to make to be ready for the CPU operations. This job is done by the loader. The loader gets the executable machine code and stores it in the memory.

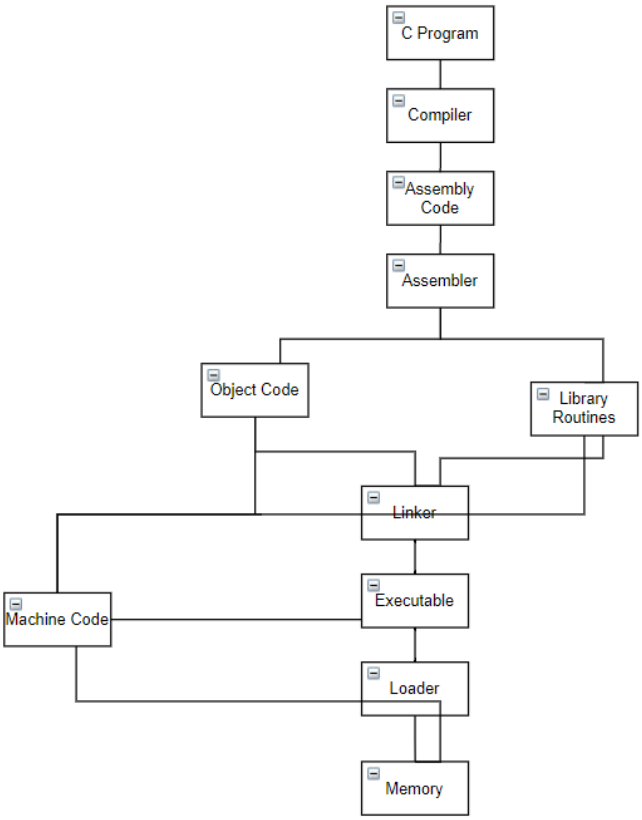

Fig. 1. Software synthesization layout [1]

So far what we have been discussing is the main flow or the architecture to synthesize any kind of software either for PCs or embedded devices. Of course there are major differences between PC and embedded processor architectures in terms of size, CPU time, ISA and costs. In the following sections we will be discussing a bit more deeply about compilers, ISAs and processor architectures.

II. Programs in High-Level Languages

High-level programming languages are considered as the languages which are the closest languages to the human languages that both computers and humans can understand. Although, high-level languages are considered as similar to the human language, e.g. English, in reality they are still pretty different from human languages in terms of grammatical (syntax) and of meaning (semantics).

In order to write a program in one of the high-level languages (throughout this paper we will be using C high-level programming language), we only need a text editor or one of the IDEs such as Visual Studio Code and save the file as .c for C language. Up to now everything is clear and simple. However, if we want to run the code and see the results we cannot see anything at all yet. Because computers or machines do not understand any of the high-level languages at all, machines can only understand binary numbers (machine language). Therefore, we need another step to translate our C code into a bit more similar form of machine language, i.e. leveling down to the low-level programming language level which is also called as the assembly level.

Leveling down to the assembly level is done by the compilers. We will go deeper about the compilers in the next section along with the simple C codes and the related assembly code together.

III. Compilers

Compilers take the high-level code and translate it into either assembly code or machine code depending on which type of operation is being done. Compilers can generate object files (.o) which are basically the executable machine code, assembly codes (.S), linker operations such as the creation of the executable programs (.bin, .exe, .hex, .out, .elf) and the creation of the libraries (.a, .dll) [2].

In order to be clearer with the compilers: a compiler takes the source files and the header files as input, and generates object files, assembly files, library files and binary files [2].

For all of the above explained operations for the C language GCC GNU C Compiler can be freely used for our desired programs. From now on, let us go deeper with the compiler operations and how compilers should be used to generate the desired output files.

Compilers have to be designed carefully and acquire scalability to generate the most efficient form of the assembly code and the machine code. Therefore, being aware of the compiler design phases would be beneficial to the programmers and engineers.

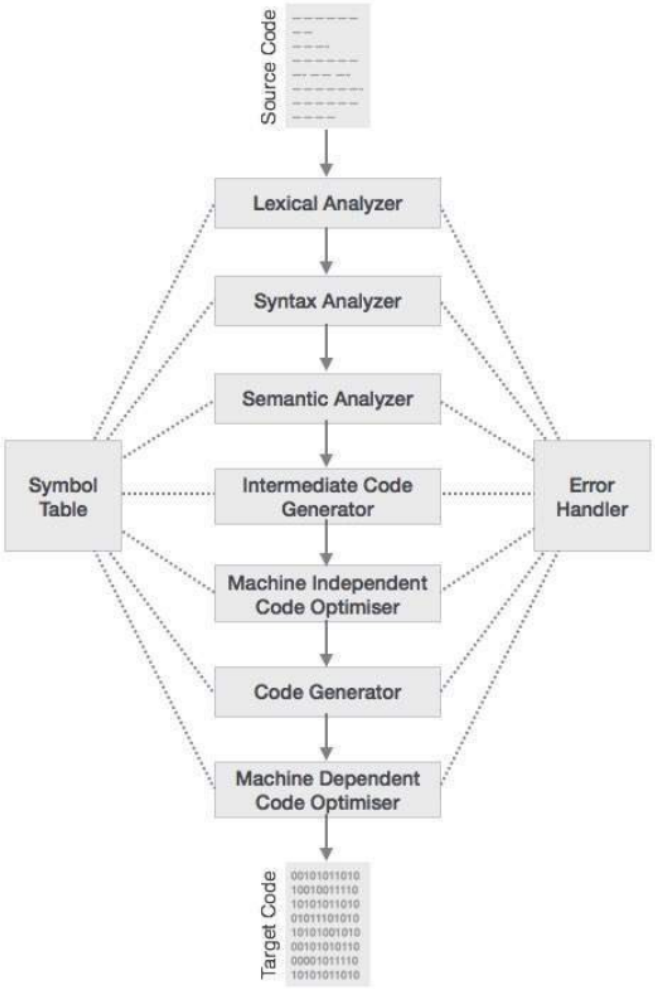

Fig. 2. Phases of the compiler design [3]

First of all compiler begins with its job by scanning the source code in order to create a dictionary out of the characters and represent them as tokens. This stage is called as lexical analyzer [3]. Then syntax analyzer takes these tokens to create a syntax tree and checks them whether they are acceptable according to predefined C language syntax rules [3]. Semantic analysis determines if the syntax tree created adheres to linguistic norms. For example, variable assignment is among appropriate data types, and string addition to an integer. The semantic analyzer also maintains record of symbols, their types, and expressions, as well as whether or not signifiers are proclaimed prior to its usage. As a result, the semantic analyzer generates an enriched syntax

tree [3]. Following semantic analysis, the compiler creates assembly language for the machine from the source code [3]. Optimization is the extraction of unneeded lines of codes of the source code and the rearranging of phrase sequences to create a faster running program with less amount of usage of the CPU and the memory [3]. After this phase, generating a machine code which is made of binary numbers so that the computer hardware can understand it and be able to realize the operation.

After getting used to how compilers are designed and know how do they generate the target code, now we can proceed with some examples to show some of the features of the compilers. Let us assume that we have the following C code saved as main.c which simply prompts "Hello World!" on the terminal screen:

```
// main.c

#include <stdio.h>

int main(void){
    printf("Hello World!");

    return 0;
}
```

In order to compile this program, we type the following commands on the terminal:

```
$ gcc main.c
```

and we get the binary file with the default name a.out on Linux or a.exe on Windows. If we want to change the file name from a.out to main.out we need to use -o flag to do this:

```
$ gcc main.c -o main
```

this command will generate the file with the name main.out. If we are using a source or library file which is not included in the compiler's standard path finder we can check and add the desired file path as following:

```
$ gcc -v        -- this command shows
the compiler's standard paths

$ gcc main.c -I <path_name> -o main
```

With this -I flag we are able to compile a file which is stored in a folder outside of the compiler's standard paths [2].

Following set of gcc compiler flags will produce preprocessor output, assembly code and object file respectively:

```
$ gcc -E main.c > main.i
$ gcc -S main.c > main.s
$ gcc -C main.c
```

Instead of writing all of these commands separately, we can use -save-temps flag to generate all of the above outputs mentioned [4]:

```
$ gcc -save-temps main.c

$ ls
a.out main.c main.i main.o main.s
```

If we are interested in reading the source code related assembly code for example:

```
$ cat main.s
    main:
        addiu   $sp,$sp,-32
        sw      $31,28($sp)
        sw      $fp,24($sp)
        move    $fp,$sp
        li      $2,1919680512 # 0x726c0000
        ori     $4,$2,0x6421
        jal     printf
        nop

        move    $2,$0
        move    $sp,$fp
        lw      $31,28($sp)
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
        jr      $31
        nop
```

Here we come to a very important point which is the assembly language and the instruction set architectures which describes and determines the behavior of the program and how can it be handled in the hardware. Therefore, the following section is the most important part of this paper. As we understand more about the assembly language and the instruction set architectures, we will be able to design more efficient and reliable systems.

IV. Instruction Set Architectures (ISA)

In designing of the computers, as we discussed before briefly, engineers had to find out a way to connect the hardware with the software. Naturally, this process was not that simple to figure out. In order to understand how engineers found out a perfect solution to this problem, we should focus on the fundamentals of the computer abstraction levels. Just for convention, we are going to separate the computer abstraction layers into three: software layer, middle layer and hardware layer.

In hardware layer, we have several chips that each one of them have crucial roles. These hardware components are nothing more than the different combinations of the transistors which operate under the rules of the semi-conductor physics. Only thing transistors can understand is the voltage values on them such as 0V and 5V. Scientists named these voltage values using binary number system 0s and 1s (0V: 0 and 5V: 1). They used this new method to direct orders to the computer components in a standardized way in which contained specific binary number patterns to determine the

behavior of the targeted component. At this point they already created the machine language literally which made up of only 0s and 1s (e.g. 10001001 00001101).

On the other hand, in software level we have programs such as games, simulators, web-browsers, etc. which are written in one of high-level programming languages (C, C++, Python, Java, etc.). These programming languages look so similar to English so that the programmers can create whatever software they want in a less exhausting way. But we have been discussing about binary numbers in the hardware level, and here we are talking about languages similar to the human language. Therefore, there must be a magical process happening in the middle layer. Here arises this section's main concern which is the design of the instruction set architectures (ISA).

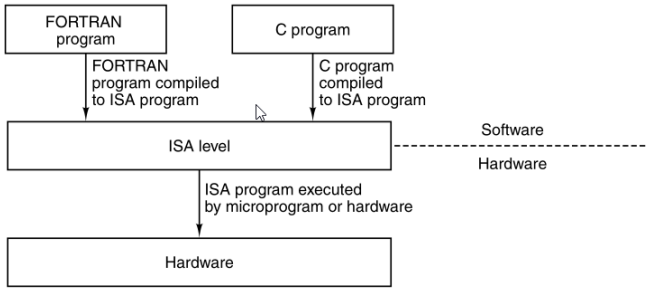

Fig. 3. Simple computer organization [5]

The contact in between CPU and the programmer is described as the instruction set [6]. Processors have their own instruction sets that is implemented in hardware to perform specific requirements including such moving, adding, or multiplying data [6]. There are two major ISAs that used widely in the processor manufacturing industry: Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC).

A. RISC

Reduced Instruction Set Computers were designed to be simple in terms of instructions so that the speed of the program executions can be higher. In order to implement a RISC architecture, in hardware level there has to be done a lot of work. Because of this, although RISC instructions simple and easy to use, the hardware implementation of RISC is complex. RISC is meant to be fast in executing the instructions since there are less amount of instructions and less usage of the main memory. In RISC, mostly the access to the memory is required only by store and load instructions. All the remaining instructions are stored in the registers. Therefore, register-register implementation is considered as the fastest way of executing the instructions.

Although it is fast in executing the instructions, at the same time it causes pretty big amount of instruction code implementation and complexity in programming.

As a last point for the RISC, it enables programmers implementation of pipelines, in other words parallelism. This feature becomes so handy in many cases, when designers try to increase the speed of the instruction executions by dividing instructions into different parts. This method is called as ISA pipelining. Another pipelining method is to use more CPU cores so that the separate CPU cores can focus on the same problem.

In the following two subsections we are going to go a bit deeper about two different ISA designs and focus more on assembly language basic for the MIPS32 ISA and few simple examples on how registers are used to execute assembly instructions. In ARM part, we will focus on general architecture basics of the ARM processors and their importance in the embedded systems.

1) MIPS: MIPS32 ISA is the simplest architecture to understand what is happening in the processor. As the MIPS32 name suggests that all the MIPS instructions are 32 bits long and the amount of the registers used in this architecture is 32. All the registers are classified according to specific tasks that MIPS processor may have. Register names and their roles are listed in the following lines: [1]

```
-name-  -register #-  -explanation-
$zero       0         const. 0
$at         1         reserved
$v0-$v1     2-3       return values
$a0-$a3     4-7       arguments
$t0-$t7     8-15      temporary
$s0-$s7     16-23     saved values
$t8-$t9     24-25     temporary
$gp         28        global pointer
$sp         29        stack pointer
$fp         30        frame pointer
$ra         31        return address
```

We will return at the usage of the listed registers later on. At the moment, we are going to mention about the operation formats of the MIPS32 architecture. We have three different formats that allows us to implement any kind of programs as the following: [1].

```
R format:
| op | rs | rt | rd | sa | function |
I format:
| op | rs | rt | immediate |
J format:
| op | jump target |
```

The type of the operation is specified in the 'op' section. 'rs' is resource 1, 'rt' is either the resource 2 or the destination. 'rd' is the destination. 'sa' is the shift amount. 'function' specifies the function of the operation type. 'jump target' is

simply the target address.

R format allows us to implement arithmetic operations with two variables. I format either runs the arithmetic operation with one variable and one constant or the load/store operations. J format is responsible for the branching and jumping to the specific address.

Let us try to observe some these terms within the following simple C code to MIPS32 assembly code example: [1]

```
C code:

f = (g + h) - (i + j);

registers -> f,g,i,j:$s0,...,$s4

assembly code:

add $t0, $s1, $s2 # g+h
add $t1, $s3, $s4 # i+j
sub $s0, $t0, $t1 # f=(g+h)-(i+j)
```

As easily can be seen from the example, operations are not done at once, instead done in separate steps. First step is to add the s1 and the s2 registers and store the value in the t0 register. Then same procedure for the second add instruction. Finally, we subtract t1 register value from t0 and store it into the s0 register (saved value register). In this example we used the R format.

What if we want to load data from the memory and store data to the memory. Then we should be using I format to do this. For example;

```
lw $t0, 4($s3)
sw $t0, 8($s3)
```

In the first instruction we are loading a word (lw) from the memory. Every word has the size of 4 bytes. Therefore, we are getting the value stored in 4 bytes away from the base address of the s3 and loading it into one of the temporary registers. In second instruction, we are doing exactly the vice versa. Storing a word (sw) in the t0 register into the memory address which is 8 bytes (2 words) away from the base address of the s3 register.

Now let us have another example in which we have a while loop and if the test condition is satisfied we have to increment the value of a variable by one: [1]

```
C code:

while (s[i] == k){
    i += 1;
}

registers -> s3: i, s5: k, s: s6

assebly code:

Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: ...
```

With the sll operation (shift left), we are calculating the distance of the first element of the array s[i] from its base address. Shifting it by 2 bytes (its result is 4 bytes from 2^2) and then adding it with the first element which is already stored in the firs place in the array gives us the address of the first word in the array. Then we load the first word from the memory (lw). 'bne' operation helps us to branch to Exit if the first word of the array is not equal to the s5 register value (k). If this condition is satisfied, we continue with the loop by incrementing the s3 (i) register value by one and proceed with the jump instruction (j). If the condition is not satisfied, we directly exit the loop and branch to the Exit.

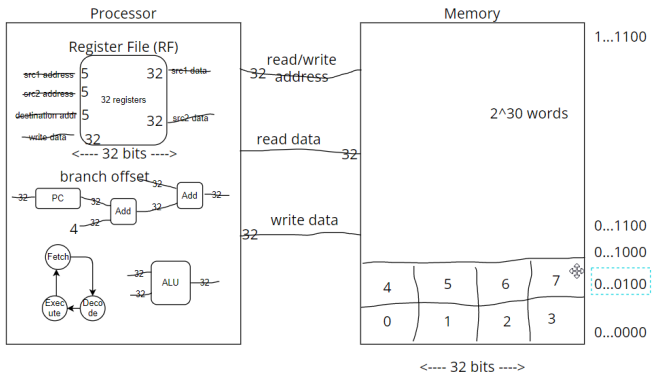

Fig. 4. MIPS32 organization with the processor and the memory

In the figure above (Fig. 4) we can see the simplified organization of the MIPS32 architecture. In the processor, we have register file (RF) that deals with source allocation and determining the addresses of the sources. Just below the RF, we have a circuit that calculates the branch addresses and in the ALU, the arithmetic and the logic operations are done. MIPS processor is simply using the cycle routine depicted by circles, first fetch the data, decode the data and execute the data.

Only three operation can be proceeded between the processor and the memory according to this organization: Get the addresses of where we want to read the data from and write the data to.

We can keep giving many more examples about MIPS32 assembly instructions and floating point operations (which is very complex and needs to studied alone to understand the

floating point operations), however it is out of scope of this paper.

Since we have acquired some of the basic ideas about how ISA deal with the high-level languages in terms of translating them into the assembly language, we can move on to the another RISC based ISA called ARM without mentioning its assembly instructions.

2) ARM: ARM is one of the processor architectures which is widely used in embedded devices. It is also mostly based on the RISC architecture model and register-register operations as in the MIPS architecture. ARM uses 32 bits registers and instructions are pretty similar to the MIPS instructions.

ARM architecture is designed to be fast (natural effect of using register-register mode which is implemented in the processor itself) and more efficient in energy consumption. As a plus, because of the reduced chip size of the ARM processors, it enables embedded system engineers to use them more frequently. It also makes it possible to reach the main memory via load and store instructions when needed.

Most of the people who are not deeply interested in this field would think that Intel or AMD (biggest personal computer processor manufacturers) are earning more than any other processor manufacturers. It is a pure misconception of course. Since the embedded devices are available all around in our daily life such as mobile phones, embedded automobile computers, TVs, watches, airplanes, etc. we can expect that the need of these processors is extremely high. Therefore, ARM processors are being used in most of the systems and produced huge amount of them yearly.

B. CISC

Complex Instruction Set Computers (CISC), are very complex in terms of number of instructions and implementation of them. They are complex in software part but not complex in hardware implementation. Compared to the number of instructions provided by RISC, CISC number of instructions are way more. However, this does not mean that it is harder than the RISC to implement. CISC architecture is mainly used by the well-known processor manufacturer company Intel.

V. Conclusion

As a conclusion to the problem that we stated in the abstract, in order to design a complete system, it is not enough to be able to write and understand one of the high-level programming languages. There was a gap between the high-level languages and the machine language. Therefore, we focused on the compilers that enables us to get closer to the machine language level by translating the high-level languages into the assembly instructions which are specified by the processor manufacturers according to the global standards. With the help of the ISA we are able to close the gap between the software and the hardware.

References

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design MIPS Edition The Hardware/Software Interface*. Elsevier Science, 2020.

[2] P. Johnston, "Compilers," https://embeddedartistry.com/lesson/compilers, November 2019, (Accessed on 06/05/2022).

[3] TutorialsPoint, "Compiler design - phases of compiler," https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm, (Accessed on 15/06/2022).

[4] H. Arora, "15 most frequently used gcc compiler command line options," https://www.thegeekstuff.com/2012/10/gcc-compiler-options/, October 2012, (Accessed on 15/06/2022).

[5] A. Tanenbaum and T. Austin, *Structured Computer Organization 6th Edition*. Pearson, 2013.

[6] Studypool, "Embedded processor architecture," https://www.studypool.com/documents/4844361/embedded-processor-architecture, (Accessed on 15/06/2022).