

Xiaoshi Wang

Professor Pfaffmann

CS 203 Project 1

October 22, 2017

User Manual

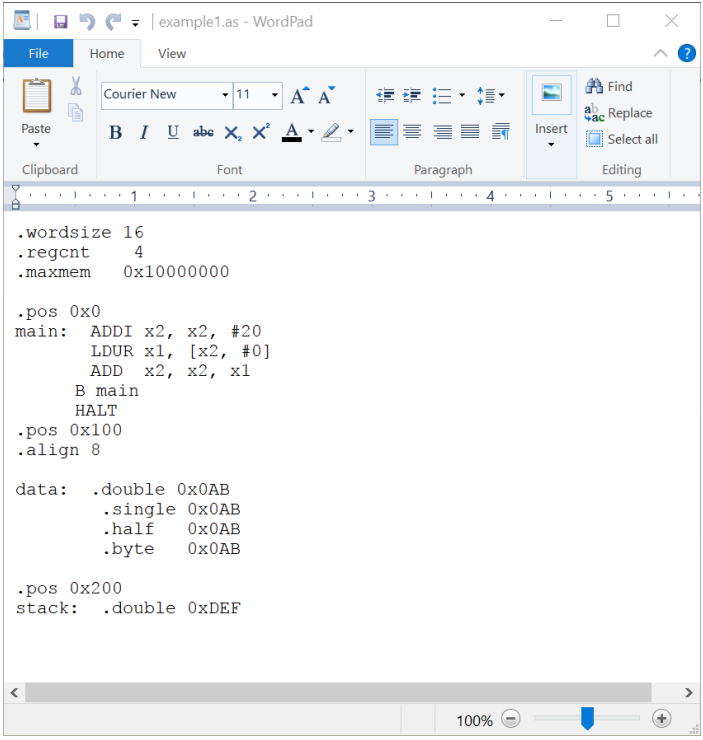
Overview

There are totally five classes: Assembler, Simulator, Processor_Window, Memory_Window, and Visualizer_Window. We translate our assembly language into machine code in “Assembler”.

And “Simulator” would do jobs like a CPU. Also, three windows (Processor_Window, Memory_Window, and Visualizer_Window) will pop-up while our simulator is working.

Assembler

A typical input file for Assembler is shown below. (Figure 1)



```
.wordsize 16
.regcnt 4
.maxmem 0x10000000

.pos 0x0
main: ADDI x2, x2, #20
      LDUR x1, [x2, #0]
      ADD x2, x2, x1
      B main
      HALT
.pos 0x100
.align 8

data: .double 0x0AB
      .single 0x0AB
      .half 0x0AB
      .byte 0x0AB

.pos 0x200
stack: .double 0xDEFF
```

Figure 1

Notice that the file type should be “.as”. And the first three lines are shown the word size, register count, and maximum memory respectively. Codes in assembly language is followed by those constants.

A file of machinal codes is generated by calling the main method of Assembler class. Notice that the input should be the file name without its suffix. Then, a file with the same name by with suffix of “.o” is generated. The corresponding file for “example1.as” is shown in figure 2.

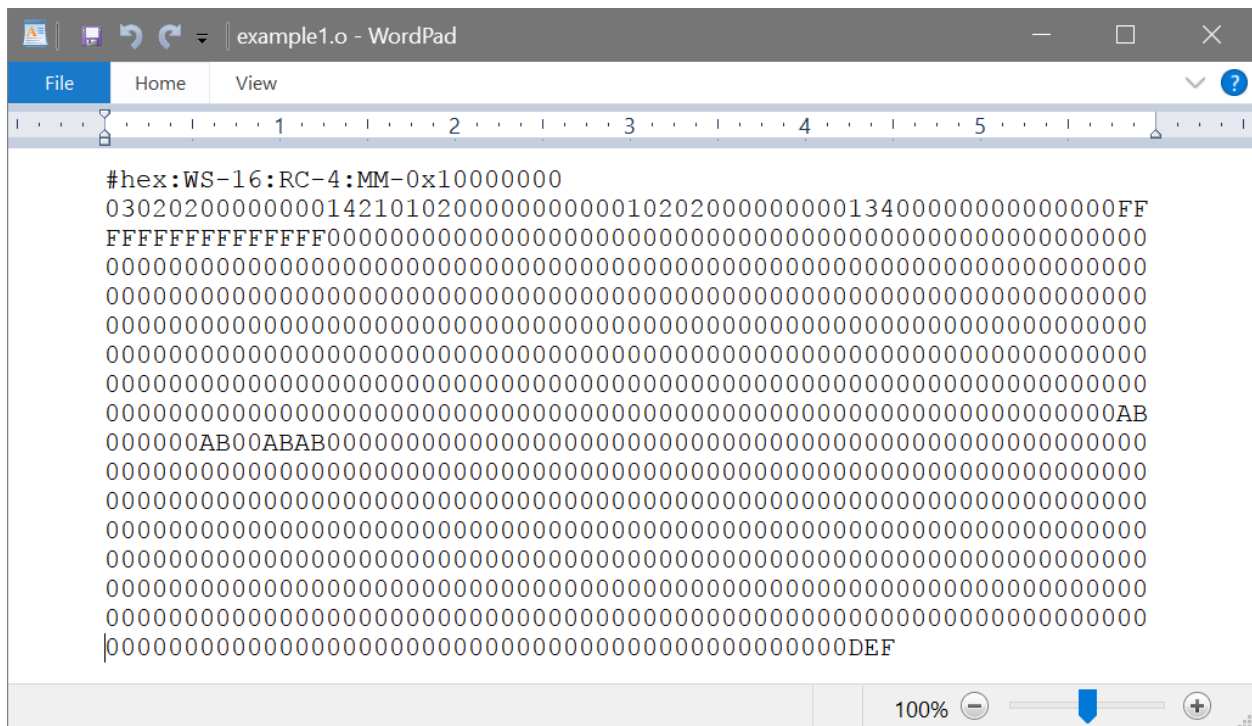


Figure 2

A file in another version is created simultaneously. Here is “example1Version2.o”. (Figure 3)

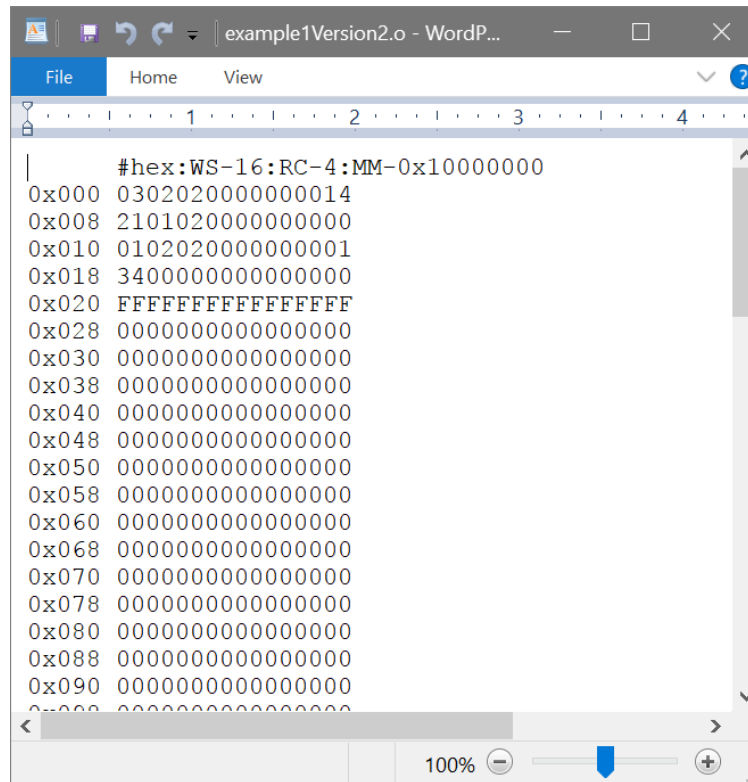


Figure 3

Notice that the hex in the first line indicate our code is in hexadecimal. And “WS”, “RC”, “MM” are stand for word size, register count and maximum memory respectively.

We the Assembler finishes its job, “*Finish Assembler program for 'example1.o'.*” is shown on the terminal.

Simulator

As in the Assembler, the input for the main method in Simulator should be the file name without suffix. Then, after we call the main, three windows are popped up. Two of them are shown below. (Figure 4 and Figure 5)

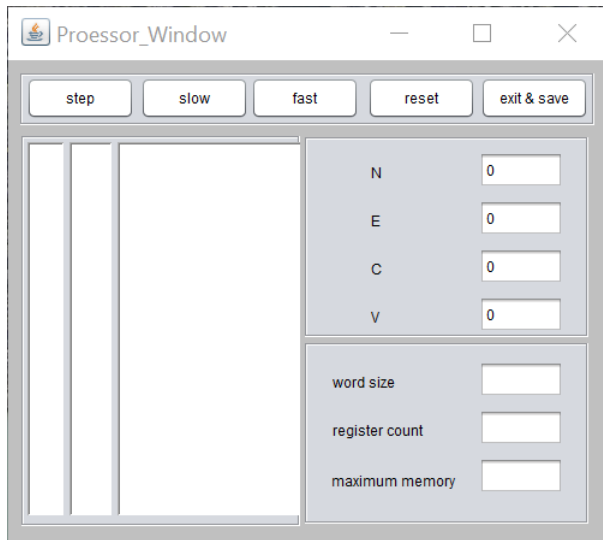


Figure 4

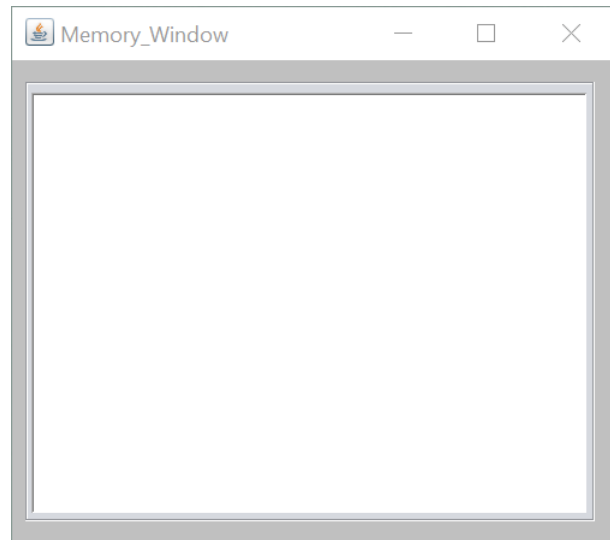


Figure 5

Buttons “step”, “slow” and “ fast” are different buttons used to proceed the code line by line. And reset button can be used to reset to the beginning of our code. The three columns on the left part shown the registers state. The fist columns is the noise mode. That is, if the register’s value changes during the previous instruction, the Boolean would be “T”. Otherwise, we get “F”. The second column shows the index of each register, and its value is present on the third column.

Figure 6 shown a Proessor_Window after several clicks on step button.

For the Meomory_Window, it would show the next line needed to be proceeded. Figure 7 is an example.

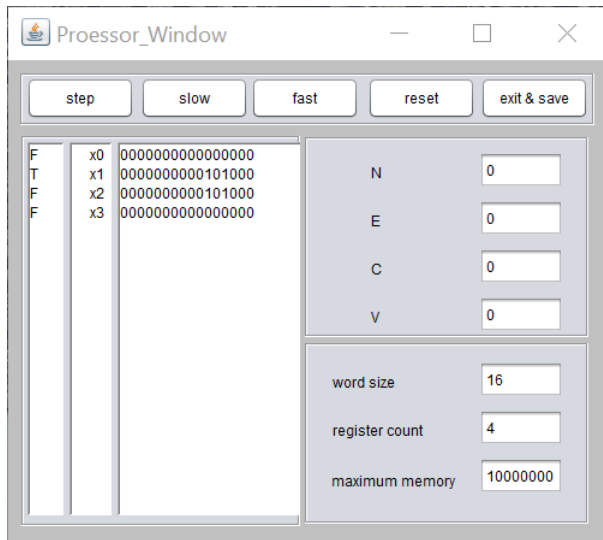


Figure 6

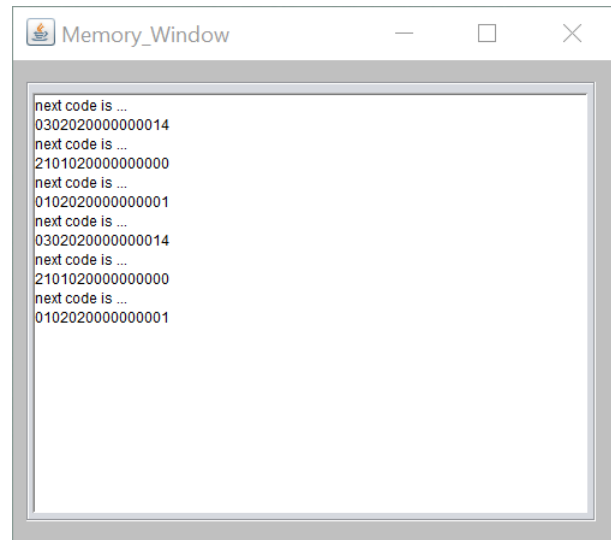


Figure 7

If we press the “exit & save” button, we will exit the program, and an empty window like Figure 8 is shown. Also, two files (Figure 9 and Figure 10) are generated. In terminal, we have

“You have pressed 'exit' button, and a file named 'example1ProcessorState.o' and 'example1MainMemory.o' are just created. ”

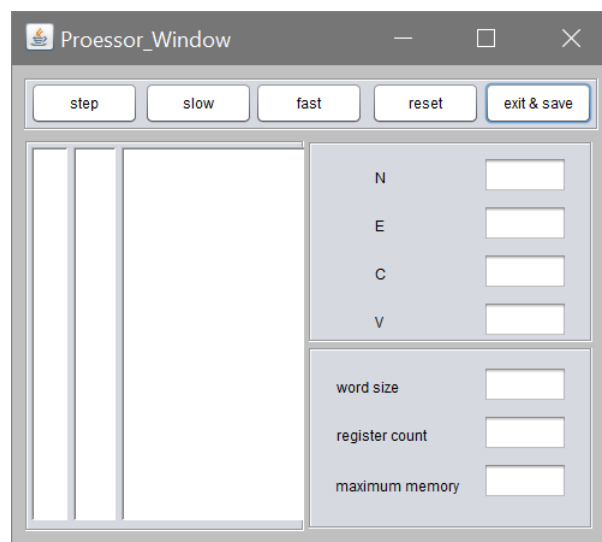


Figure 8

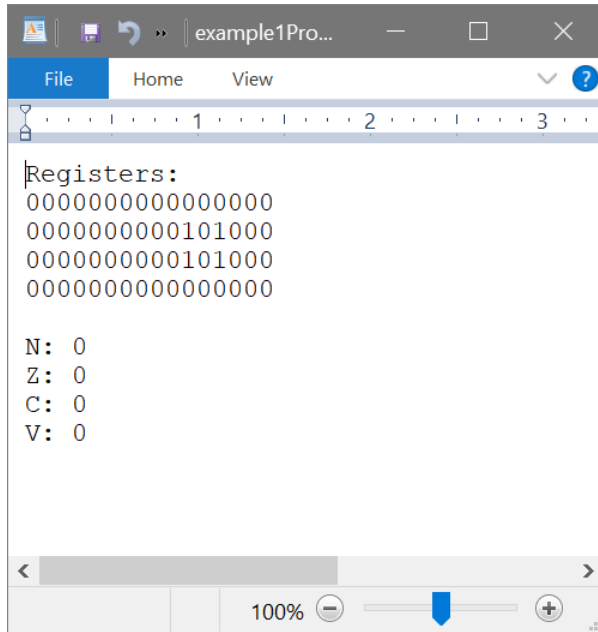


Figure 10

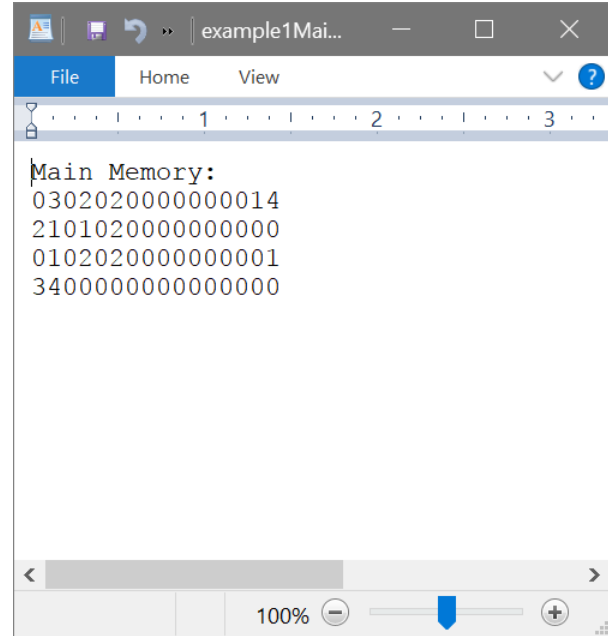


Figure 11

Visualizer_Window is the third window goes along with the simulator. The initial state of this window is shown in Figure 11. Basic instructions is shown above each mini windows. Figure 13 illustrate values this window returns if the examples are typed.

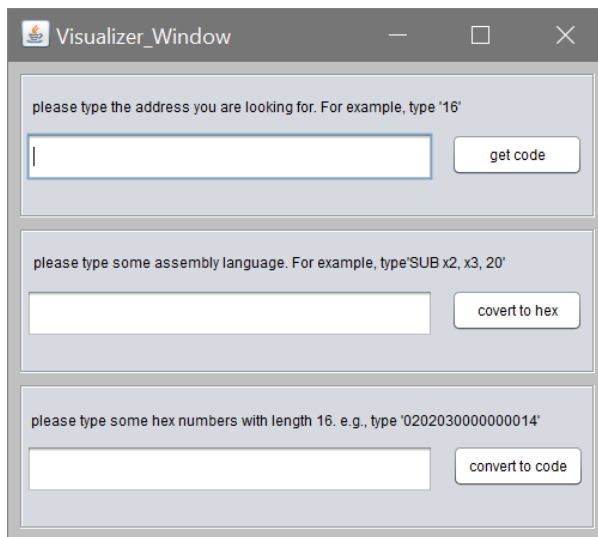


Figure 12

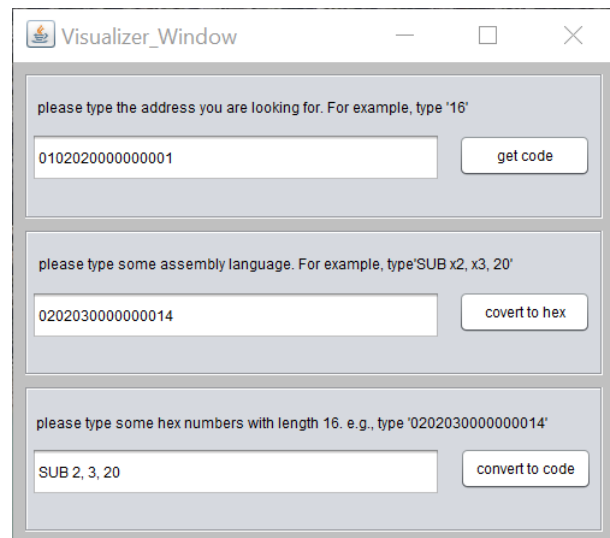


Figure 13

Project Report

Introduction

In this project, we need to design our own assembly language named *Little Finger*, and write three programs for its assembler, main-memory emulator and debugger respectively. Our assembly language should be able to handle stacks, and basic arithmetic, logical, transfer and branch operations. Here are some requirements for our three programs.

- 1) The assembler program is capable to translate the assembly language into machine code.
- 2) CPU is used to read a file with machine code and execute those code in main memory.
- 3) Processor state and main memory can be saved when CPU is working.
- 4) An image visualizer is created as the debugger such that codes are provided if an address range is given.

Assembler

For the design of our assembly language *Little Finger*, we first need to decide how long the machine code needed. Let us set the length to be 16. (Any sizes greater than 9 works. The reasoning is explained later in this report.) Since we have 32 operations in total, at least two hexadecimal digits needed to illustrate all operations. And table 1 show how I encode those 32 operations in two digits.

Operation	Hex	Operation	Hex
ADD	01	AND	11
SUB	02	ORR	12
ADDI	03	EOR	13

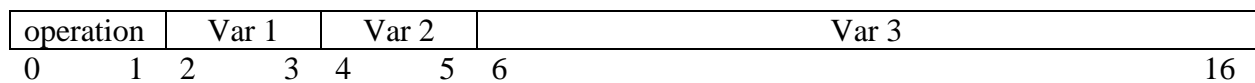
SUBI	04	ANDI	14
ADDS	05	ORRI	15
SUBS	06	EORI	16
ADDIS	07	LSL	17
SUBIS	08	LSR	18
LDUR	21	CBZ	31
STUR	22	CBNZ	32
LDURSW	23	B.cond	33
STURSW	24	B	34
LDURH	25	BR	35
STURH	26	BL	36
LDURB	27	PUSH	EF
STURB	28	POP	DF

Table 1

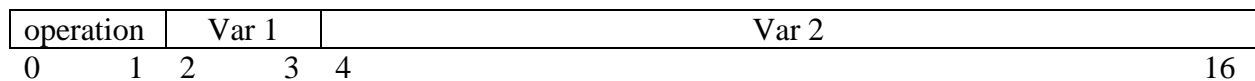
Notice that those operations within the same category have the same value for their first index.

This would help us to read the machine code easier later in our CPU. And we store the information of those operation in a hash table. Then, we can easily access an arbitrary operations and find their corresponding machine code through this hash table.

There are always some variables followed by the operations, and we have 10 digits left to encode those variables. Since the number of those inputs are varied from different operations, we have different methods to encode them. Here are the structures of our machine code for three variables (Graph 1), two variables (Graph 2), and one variable (Graph 3).



Graph 1



Graph 2

operation			Var 1												
0	1	2													

Graph 3

For example, if we want to encode “SUB x2, x3, 20”, the code for “SUB” is “02”, and the hexadecimal forms for 2, 3, and 20 are 2, 3, 14 respectively. Then, the final machine code is “0202030000000014”. Notice that in our code, we do not distinguish the index of a register and a constant because we know what the number refers to, when we read the operation.

Also, there may have labels in our assembly language. For example, “main: ” indicates the beginning of our main method. We may use these addresses somewhere when we call branches. Thus, we store those addresses in our pervious hash table as well. Then, if we want to refer those label, we can simply find their location by call it in hash table.

We need to fill nonmeaningful zeros sometimes. When we want to jump to further location, by “.pos 0x200” for example, we need to calculate the difference between current location and the targeted location, and fill those positions with zeros. Thus, we should have a global variable as a counter to record the current location. This counter is also helpful when we try to align our memory into certain form. For instance, if our current location is 44, and we want to align by 6, the difference between 6 and the reminder of $44/6$, which is 4, is the number of zeros needed to return the preferred form.

Machine Simulator

In this simulator, we need to execute the machine code from the image file we just created in Assembler. Notice that there are three part we need to consider: main memory, registers, and flags.

First, we read those machine code needed from our input. All we need to execute CPU is those code in main method. We can create an arrayList to store those codes line by line. This would be our main memory. Notice that our main method end if we meet “HALT” in our Also, another arrayList needed for storing values of all registers. (The number of registers and the size of each registers are provided in the beginning of our image file.)

Then, our CPU process the fetch execute cycle one by one. For each chunk of code, we first detect its operation by reading the first two hexadecimal numbers in this line. Notice that since our translation from operations to hexadecimal numbers is one-to-one, we can decode it without ambiguity. Then, after we know its operation, we decode the variables and follow the instruction of this operation. Values inside the registers may be modified during that process. Also, if there are any specified cases, flags should be changed to the appropriate states.

Image Visualizer

While we are proceeding our image file, we may need a debugger to check some specified line of our code. Thus, an image visualizer is constructed along our CPU.

We can find the machine code for a specified address in that Visualizer. For example, if we type “16” in our text space, the memory at address 16 would be return.

Another two features in that visualizer is to translating an arbitrary legal assembly language to machine code, and machine code to assembly language. The first feature is the same as what we did in Assembler for each line. And the second one is the same as that in Simulator. Then, if we think some lines wrong in our CPU, we can first get it by typing its address, and then translate it in the translation interface.

Conclusion

We construct a series of three programs in this project, and by which we can proceed an arbitrary assembly code. We first translate the assembly language into machine code by *Little Finger* we defined. Then, we modify how CPU works for main memory, registers, and flags in our second program. A Image Visualizer is provided as a debugger to better understanding the process of our CPU.

References

Java Platform SE 7, docs.oracle.com/javase/7/docs/api/.

Patterson, David A., et al. *Computer organization and design: the hardware/Software interface*. Elsevier/Morgan Kaufmann, 2018.

Svatek;(reaver(at)centrum.cz), all:Tomáš. “Simple GUI Extension for BlueJ.” *Simple GUI Extension for BlueJ*, gbluej.slunecnisoustava.eu/.