Faculty of Electrical Engineering in Belgrade,
Department of Computer Technology and Informatics

Subject: System software (13E113SS, 13S113SS)
A teacher: assistant. Saša Stojanović, Ph.D
School year: 2021/2022. (The assignment is valid starting from the June 2022 deadline.)

Project for homework
- Projected task -

Document version:1.2

Important notes: Before reading this text,**mandatory**read the general rules of the course and the rules related to homework! Then read this text**as a whole and carefully**, before starting implementation or asking for help. If something is not defined precisely enough in the task or contradictory requirements are set, the student should introduce reasonable assumptions, thoroughly explain them and continue to build the remaining part of his solution based on the assumptions introduced. The requirements are intentionally insufficiently detailed, because students are expected to be creative and have a professional approach in solving practical problems!

Introduction

The goal of this project is the realization of a tool in the translation chain and an emulator for abstract computer system. A description of the abstract computer system is given in the attachment. The tools in the translation chain to be implemented include an assembler for the specified abstract computer system and a linker independent of the target architecture.

The project solution includes the source code that implemented the assembler, linker and emulator. In the following text, the project solution will be briefly referred to as just *implementation*. The source assembly code written for the abstract computer system, which will be translated by the assembler, linked by the linker and executed by the emulator, will be referred to below as *user program* in order to make a clear difference with respect to *implementation*.

In order to demonstrate and defend the project, *implementation* it must run successfully under the Linux operating system as a console application. Project protection is not possible under the Windows operating system.

Lexical analysis

The implementation of lexical analysis and parsing of input text files is not the main task of the project, but it is necessary for the successful development of the solution, which is why it is allowed to use lexer and parser generators for these needs. Without going into the details and purpose of these tools, which can be useful when creating a solution, attention is drawn to the fact that the tools are installed on the virtual machine *flex* and *bison* which represent exactly the lexer and parser generators respectively. In addition to the lexer generator and parser, it is allowed and recommended to use standard libraries such as STL (*Standard Template Library*). Also, it is allowed to use non-standard libraries as long as they do not implement things closely related to the core of the project, which includes the generation of machine code, accompanying metadata, the creation of relocation records, relocation and linking of the subject programs, emulation, etc.

Evaluation of the project

The requirements within the tasks listed below are classified according to the difficulty and scope of the three levels: A (30 points), B (35 points) and C (40 points). The division of requirements by level for each of the tasks was defined immediately after the introduction of the observed task. During the defense of the project, it is possible to achieve:

- 30 points, if and only if all the requirements prescribed for level A have been completed correctly
- 35 points, if and only if all the requirements prescribed for levels A and B are done correctly
- 40 points, if and only if all the requirements prescribed for levels A, B and C are done correctly

If the student does not correctly implement all the requirements prescribed for level A for each of the tasks, it is not possible to proceed to the defense of the project at all. Those involved in the case retain the discretionary right to mark the project defense as unsuccessful if they establish that the submitted project solution contains an error during execution or deviates in any way from the requirements of the project assignment.

Task 1: Assembler

Introduction

The goal of this task is realizationsingle-pass assembler for the processor described in the attachment. The assembler input is a text file with assembly source code written according to the syntax described below. The output of the assembler is a text file that represents the program in question (it is allowed to generate as output, in addition to the text file, a binary file for easier loading of the assembler output into the linker).

The format of the subject program should be based on the school version of the ELF format reference example text file as used in the exercises in the part of the material that concerns the construction of the assembler. It is allowed to make changes in the school version of the ELF format (sections, relocation record types, additional fields in existing record types, new data on the subject program, etc.) if necessary in accordance with the needs of the target architecture. When resolving all undefined details for the needs of the solution, be guided by the principles used by the GNU assembler.

Division of requirements by levels

The division of requirements by level for this task concerns assembler directives and commands (other requirements should be implemented by default). The level in which an assembler directive should be implemented is specified in square brackets in front of the requested assembler directive. The level at which an assembler command should be implemented is specified within the row of the corresponding table for the observed assembler command.

Launching from the terminal

Translation result*implementation*this task should haveassemblerfor the name. All information required for execution is provided as command line arguments. With a single launchassemblerassembles a single input file. The name of the source assembly code input file is given as a stand-alone command line argument. Boot methodassembleris as follows:

assembler [options] <input_file_name>

Command line options

A description of the command line options, along with a description of their parameters, which may be default at startupassemblerfound below:

- o <output_file_name>

Command line option-aboutsets its parameter<output_file_name>for the name of the output file representing the assembly result.

Startup example

An example of a command that initiates the assembly of source code within a file*entrance.s* with the aim of obtaining*exit.o*of the subject program, is given below:

. /assembler -o output.o input.s

Syntax of source assembly code

The syntax of source assembly code can be roughly divided into general details, assembly directives and assembly orders. General details define the appearance of a single line of source code in terms of label entries, assembly statements, assembly directives, comments, etc. An assembler directive represents an operation that the assembler should perform during assembly. An assembly instruction is a symbolic record of a machine instruction that the assembler needs to translate into a binary representation.

General details

General details, presented as (1) functional requirements that the assembler should meet and (2) syntax rules that the assembler should check if they are followed, listed in order by item below:

- one line of source code contains at most one assembly statement or directive,
- commented content is completely ignored during assembly,
- a single-line comment, which implicitly terminates at the end of a line, begins with the # character,
- label, whose definition ends with a colon, must be found at the very beginning of the source code line (optionally after an arbitrary number of whitespace characters) and
- label can stand alone, without an accompanying assembly statement or assembly directive in the same line of source code, which is equivalent to standing at the very beginning of the first subsequent line of source code that has content.

Assembly directives

*[level A].*global <symbol_list>

Exports the symbols specified within the parameter list. The parameter list may contain only one or more symbols separated by commas.

*[level A].*extern <symbol_list>

Imports the symbols specified within the parameter list. The parameter list may contain only one or more symbols separated by commas.

*[level A].*section <section_name>

Starts a new assembly section, automatically starting the previously started section ends, with an arbitrary name specified as a parameter to an assembler directive.

*[level A].*word <list_of_symbols_or_literal>

Allocates a fixed size space of two bytes for each initializer (symbol or literal) specified within the parameter list. The parameter list can contain only one initializer or multiple initializers separated by commas. The assembler directive initializes the allocated space with the value of the specified initializers.

*[level A].*skip <literal>

Allocates space equal to the number of bytes defined by the literal specified as a parameter. The assembler directive initializes the allocated space with zeros.

[level B].ascii <string>

Allocates space of a fixed size of one byte for each character of the string (character string between quotation marks). The assembler directive initializes the allocated space with values corresponding to the specified characters according to the ASCII table.

[level C].equ <new_symbol>, <expression>

Defines a new symbol whose value is equal to the specified expression.

[level A].end

Ends the process of assembling the input file. The rest of the input file is discarded, i.e. it is not assembled.

Assembly instructions

Mnemonic	The effect	Flags	Level
halt	Stops execution of instructions	-	A
int*regD*	push psw; pc <= mem16[(*regD*mode 8)*2];	-	A
iret	pop psw; pop pc;	psw	A
call*operand*	push pc; pc <=*operand*;	-	A
ret	pop pc;	-	A
etc*operand*	pc <=*operand*;	-	A
yes*operand*	if (equal) pc <=*operand*;	-	A
etc*operand*	if (not equal) pc <=*operand*;	-	A
jgt*operand*	if (signed greater) pc <=*operand*;	-	A
push*regD*	sp <= sp - 2; mem16[sp] <=*regD*;	-	A
priest*regD*	*regD*<=mem16[sp]; sp <= sp + 2;	-	A
xchg*regD*,*regS*	temp <=*regD*;*regD*<=*regS*;*regS*<=temperature;	-	A
add*regD*,*regS*	*regD*<=*regD*+*regS*;	-	A
Sat*regD*,*regS*	*regD*<=*regD*-*regS*;	-	A
mul*regD*,*regS*	*regD*<=*regD***regS*;	-	A
giant*regD*,*regS*	*regD*<=*regD*/*regS*;	-	A
cmp*regD*,*regS*	temp <=*regD*-*regS*;	ZOCN	A
not*regD*	*regD*<= ~*regD*;	-	A
and*regD*,*regS*	*regD*<=*regD*&*regS*;	-	A
Or*regD*,*regS*	*regD*<=*regD*|*regS*	-	A
xor*regD*,*regS*	*regD*<=*regD*^*regS*;	-	A
test*regD*,*regS*	temp <=*regD*&*regS*;	ZN	A
shl*regD*,*regS*	*regD*<=*regD*<<*regS*;	ZCN	A
shr*regD*,*regS*	*regD*<=*regD*>>*regS*;	ZCN	A
ldr*regD*,*operand*	*regD*<=*operand*;	-	A
p*regD*,*operand*	*operand*<=*regD*;	-	A

Label *regX* represents the designation of one of the programmatically accessible registers of the target architecture. Programmatically accessible registers are r0, r1, r2, r3, r4, r5, r6/sp, r7/pc and psw.

Label *operand* includes all syntax notations for specifying operands for different addressing modes. The syntax notations differ depending on whether they are assembly data statements or assembly jump statements.

Assembly commands for working with data support different syntax notations for operand, described below, which define the value data :

- $<literal> - value <literal>
- $<symbol> - value <symbol>
- <literal> - value from memory at address <literal>
- <symbol> - value from memory at address <symbol> absolute addressing
- %<symbol> - value from memory at address <symbol> PC by relative addressing
- <reg> - value in the registry <reg>
- [<reg>] - value from memory at address <reg>
- [<reg> + <literal>] - value from memory at address <reg> + <literal>
- [<reg> + <symbol>] - value from memory at address <reg> + <symbol>

The assembly jump and subroutine call statements support different syntax notations for operand, described below, which define the value destination hop addresses :

- <literal> - value <literal>
- <symbol> - value <symbol> absolute addressing
- %<symbol> - value <symbol> PC by relative addressing
- * <literal> - value from memory at address <literal>
- * <symbol> - value from memory at address <symbol>
- * <reg> - value in the registry <reg>
- * [<reg>] - value from memory at address <reg>
- * [<reg> + <literal>] - value from memory at address <reg> + <literal>
- * [<reg> + <symbol>] - value from memory at address <reg> + <symbol>

Task 2: Linker

Introduction

The goal of this task is the realization of a linker independent of the target architecture, which, based on metadata (symbol table, relocation records, etc.), connects one or more subject programs generated by the assembler from the first task.

The input of the linker is the output of the assembler, where it is possible to specify a larger number of subject programs that need to be linked. By default, the linker places the sections, starting at address zero, one right after the other in the order they are defined within the subject program. The linker processes a large number of subject programs at its input in the order in which they are specified via the command line. When placing a section with the same name as a previously placed section, it is inserted starting from the place where the section of the same name ended earlier, thus creating an aggregation of sections with the same name, where there is no overlap with the following sections, which is achieved by pushing them to higher addresses.

The output of the linker is a text file with contents as described below (it is allowed to generate as output, in addition to a text file, a binary file for easier loading of the linker output into the emulator).

Division of requirements by levels

The division of requirements by level for this task concerns the command line options (other requirements should be implemented by default). The level at which a command line option should be implemented is specified in square brackets before the requested command line option.

Launching from the terminal

Translation result*implementation*this task should havelinkerfor the name. All information required for execution is provided as command line arguments. With a single launchlinkerlinks one or more input files. The names of the input files, which represent the programs in question, are given as independent command line arguments. Boot methodlinkeris as follows:

linker [options] <input_filename>...

Command line options

A description of the command line options, along with a description of their parameters, which may be default at startuplinkerin arbitrary order is below:

[level A]-o <output_file_name>

Command line option-aboutsets its parameter<output_file_name>for the name of the output file representing the result of the link.

[level B]-place=<section_name>@<address>

Command line option-placeexplicitly defines the address starting from which it is located section of the specified name, where the address and name of the section are specified<section_name>@<address> parameter. This option can be specified multiple times for different section names to define addresses for multiple sections from input files. All sections for which this option is not

listed are placed in the default way, described in the introduction of this task, starting immediately after the end of the section that is placed at the highest address.

[level A]-hex

Command line option-hexrepresents a directive to the linker to generate a record as a result of linking, on the basis of which memory initialization can be performed, in the form of a set of pairs (*address, content*). The content represents the machine code that should be found at the given address. Pairs are generated only for those addresses where content with a defined initial value should be placed. The format of the record, with an example where the initial value of the content is equal to its address, is shown below:

0000: 00 01 02 03 04 05 06 07 0008: 08
09 0A 0B 0C 0D 0E 0F 0010: 10 11 12 13
14 15 16 17

Linking is only possible if there are no (1) multiple symbol definitions, (2) unresolved symbols and (3) overlaps between sections from input syllabuses when considered-placecommand line options. If one of the previous conditions is not met for the given input files of the linker, the linker must report an error with a corresponding message. The names of the symbols or sections causing the error should be part of the error message.

When starting the linker, specifying exactly one of the-relocateableand-hexcommand line option is required. The linker should not generate any output if exactly one of the two previously listed command line options is not specified.

[level C]-relocateable

Command line option-relocateablerepresents a directive to the linker that as a result connections is generated by the subject program, in the same format as the assembler output, in which all sections are placed also from the zero address (completely ignoring the potentially specified-place command line options). The object program obtained in this way can later be specified as input to the linker.

Linking is only possible if there are no multiple symbol definitions. If there are multiple symbol definitions for given linker input files, the linker must report an error with a corresponding message. The name of the multiple defined symbol should be part of the error message.

When starting the linker, specifying exactly one of the-relocateableand-hexcommand line option is required. The linker should not generate any output if exactly one of the two previously listed command line options is not specified.

Startup example

An example of a command that starts the linking of the programs in question*entrance 1.o*and *entrance2.o*which (1) defines the addresses where the corresponding sections are placed and (2) requires the generation of memory initialization records, is given below:

```
. /linker - hex
    - place=iv_table@0x0000  -place=text@0x4000
    - about mem_content.hex
    input1.o input2.o
```

Task 3: Emulator

Introduction

The goal of this task is the realization of the interpretive emulator for the computer system described in the attachment. The emulator input is the memory initialization file obtained as linker output with the specified-hexcommand line option. Emulation is only possible if the input file can be successfully loaded into the memory address space of the emulated computer system. After starting the emulator, the only printout in the console is printout directly from*user program*, while *implementation*it does not print anything on its own until the emulation is complete. Emulation ends when the emulated processor executeshaltinstruction*user program*. After the emulation is complete *implementation*prints the state of the emulated processor in the following format:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Emulated processor executed halt instruction Emulated
processor state: psw=0b0000000000000000 r0=0x0000
r1=0x0000 r2=0x0000 r3=0x0000 r4=0x0000 r5=0x0000
r6=0x0000 r7=0x0000
```

Division of requirements by levels

The breakdown of requirements by level for this task is defined below. For level A, it is necessary to emulate the entire observed computer system except for the terminal and timer peripherals. For level B, in addition to everything defined by level A, it is necessary to emulate the terminal peripheral as well. For level C, in addition to everything defined by level B, it is necessary to emulate the timer peripheral.

Launching from the terminal

Translation result*implementation*this task should haveemulatorfor the name. All information required for execution is provided as command line arguments. With a single launch emulatoremulates a single execution of a program from an input file. The name of the input file, which represents the output of the linker with the specified one-hexcommand line option, is given as a stand-alone command line argument. Boot methodemulatoris as follows:

```
emulator <input_filename>
```

Startup example

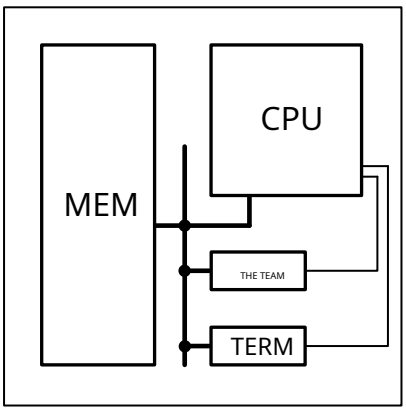
Example command, which starts emulation based on the memory initialization record in the input file*mem_content.hex*, is given below:

```
. /emulator mem_content.hex
```

Attachment: Description of the computer system

Introduction

An abstract computer system consists of a processor, operating memory, timers and terminal. All components of the computer system are connected to each other via the system bus. The timer and terminal are connected next to the system bus and directly to the processor via interrupt request lines. A simplified schematic representation of the considered abstract computer system is shown in the following figure:



Processor description

A section of a 16-bit dual-address processor with Von-Neumann architecture is described below. The addressable unit is one byte, and the order of bytes in a word is little-endian. The size of the memory address space is 2_{16}B.

Memory-mapped registers

Memory-mapped registers are registers that are accessed by instructions to access the memory address space. Starting with the address0xFF00of the memory address space, there is a 256-byte space reserved for memory-mapped registers. Memory-mapped registers are used to work with peripherals in a computer system.

Interrupt mechanism

The table of interrupt routines or IVT (interrupt vector table) is located starting from the address 0x0000memory address space and has eight inputs. Each entry occupies two bytes and contains the address of the corresponding interrupt routine. The inputs to IVT correspond to the following interrupt routines:

- input 0 contains the addressinterrupt routine that is executed during startup or reset of the entire processor (the complete interrupt processing sequence is not performed, but only a jump to the address that is within the given input is performed),
- input 1 contains the address of the interrupt routine, which is executed if an incorrect instruction is attempted (nonexistent operation code, incorrect addressing method, etc.),
- input 2 contains the address of the interrupt routine that is executed when a timer interrupt request arrives (a description of the timer's operating principle and how to configure it is given in a separate chapter),

- input 3 contains the address of the interrupt routine that is executed when an interrupt request arrives from the terminal (a description of the terminal's operating principle is given in a separate chapter) and
- the other inputs are free for use by the programmer.

Each machine instruction of the processor is executed atomically. Interrupt requests are serviced only after the current machine instruction has been atomically executed to completion. The processor, when accepting an interrupt request and entering the interrupt routine, only places the program status word and the return address on the stack.

Processor registers

The processor has eight general-purpose 16-bit registers marked with r<num> where <num> can have values from zero to seven. Register r7 is used as pc register. Registry value r7 contains the address of the next instruction to be executed. Register r6 is used as sp register. Registry value r6 contains the address of the occupied location at the top of the stack (the stack grows to lower addresses).

In addition to general-purpose registers, there is psw register (processor status word). Status the processor word consists of flags that provide the possibility to configure the interrupt mechanism and represent the status of the executed program. The layout of the processor status word is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I	Tl	Tr										N	C	ABOUT	Z

Meaning of flags in psw registry:

- Z (Zero) - the result of the previous operation is zero,
- ABOUT (Overflow) - exceeding,
- C (Carry) - transmission,
- N (Negative) - the result of the previous operation is negative,
- Tr (Timer) - masking of timer interruptions (0 - enabled, 1 - masked),
- Tl (Terminal) - masking of interruptions from the terminal (0 - enabled, 1 - masked) and
- I (Interrupt) - global masking of external interrupts (0 - enabled, 1 - masked).

Processor instruction format

Instructions can be from one to five bytes in size. Instruction in the most general case has the following format:

I	II	III	IV	V
InstrDescr	RegsDescr	AddrMode	DataHigh	DataLow

The first byte InstrDescr represents the description of the instruction, which contains the operation code and the instruction modifier. The operation code defines which instruction it is. The modifier further tells exactly what the instruction should do. The format of the first byte of the instruction is as follows:

7	6	5	4	3	2	1	0
OC_3	OC_2	OC_1	OC_0	MOD_3	MOD_2	MOD_1	MOD_0

Meaning of bits InstrDescr instruction byte:

- $OC_3OC_2OC_1OC_0$ - instruction operation code i
- $MOD_3MOD_2MOD_1MOD_0$ - instruction modifier.

Second byte RegsDescr defines the registers used by the instruction through their indices. Public registers are assigned indexes corresponding to their names. Registry index r0 is zero, the register index r1 is one etc. Index psw register is eight. The format of the second byte of the instruction is as follows:

7	6	5	4	3	2	1	0
RD_3	RD_2	RD_1	RD_0	RS_3	RS_2	RS_1	RS_0

Meaning of bits RegsDescr instruction byte:

- $RD_3RD_2RD_1RD_0$ - index of the first register (mainly *destination*) which the instruction uses and
- $RS_3RS_2RS_1RS_0$ - index of another register (mainly *source*) that the instruction uses.

The third byte AddrMode defines the way of addressing operands for instructions that do not only handle registers. The format of the third byte of the instruction is as follows:

7	6	5	4	3	2	1	0
Cf_3	Cf_2	Cf_1	Cf_0	AM_3	AM_2	AM_1	AM_0

Meaning of bits AddrMode instruction byte:

- $Cf_3Cf_2Cf_1Cf_0$ - method of updating in register-indirect addressing i
- $AM_3AM_2AM_1AM_0$ - addressing method.

Ways of addressing

The largest part of the instruction set of the processor uses only data from the registers, ie it exclusively and by default uses register direct addressing without support for any other addressing methods. A small subset of the instruction set supports the other addressing modes described below.

The addressing mode description uses several terms to define each of the addressing modes. A term *operand* the code of the group of instructions for working with data represents the data itself, while the code of the group of jump instructions represents the destination address of the jump. A term *selected registry* implies a register defined by bits $RS_3RS_2RS_1RS_0$ inside of RegsDescr instruction byte. A term *payload instructions* refers to DataHigh and DataLow byte instructions. Depending on the value of the bits $AM_3AM_2AM_1AM_0$ The processor supports the following addressing methods:

- 0b0000 - directly ; *operand* is equal *payload instructions*,
- 0b0001 - register directly ; *operand* is equal to the value of u *selected register*,
- 0b0101 - register direct with 16-bit signed sum ; *operand* is equal to the sum of the values in *selected register* and *payload instructions* which represents the marked addition.
- 0b0010 - registry indirect ; *operand* is in memory at the address indicated by the value u *selected register* (value *of the selected registry* is updated when the instruction is executed according to the value $Cf_3Cf_2Cf_1Cf_0$ bits),
- 0b0011 - register indirect with 16-bit signed shift ; *operand* is in memory at the address indicated by the sum of the values u *selected register* and *payload instructions* which represents the labeled displacement (value *of the selected registry* is updated when the instruction is executed according to the value $Cf_3Cf_2Cf_1Cf_0$ bits),
- 0b0100 - memory ; *operand* is in memory at the indicated address *payload instructions*.

Value$Cf_3Cf_2Cf_1Cf_0$of bits defines how the value will be updated *of the selected registry* when executing an instruction that uses one of the register-indirect addressing methods. Depending on the value$Cf_3Cf_2Cf_1Cf_0$bit value *of the selected registry* is updated as follows:

- 0b0000-no update,
- 0b0001-is decremented by two before forming the address *operand*,
- 0b0010-it is increased by two before forming the address *operand*
- 0b0011-is decremented by two after the address is formed *operand* and
- 0b0100-it is increased by two after forming the address *operand*.

Overview of processor instructions

An instruction to stop the processor

InstrDescr

7654 3210

0000	0000

Stops the processor from continuing to execute subsequent instructions. The instruction size is one byte.

Software interrupt instruction

InstrDescr		RegsDescr	
7654 3210		7654 3210	
0001	0000	DDDD	1111

Generates a software interrupt request. The IVT entry number for which the interrupt request is generated is in the destination register. The size of the instruction is two bytes.

push psw; pc<=mem16[(reg[DDDD] mod 8)*2];

Return instruction from interrupt routine

InstrDescr

7654 3210

0010	0000

Exits the interrupt routine by returning to the address from the stack and restores psw register value from the stack. The instruction size is one byte.

pop psw; pop pc;

Subroutine call instruction

InstrDescr		RegsDescr		AddrMode	
7654 3210		7654 3210		7654 3210	
0011	0000	1111	SSSS	UUUU	AAAA

Calls the subroutine by jumping to the address defined *operand* before saving the return address on the stack. The size of the instruction is three or five bytes (depending on the addressing method).

push pc; pc<=*operand*;

Return instruction from subroutine

InstrDescr

7654 3210

0100	0000

Exits the subroutine by returning to the address on the stack. The instruction size is one byte.

pop pc;

Jump instruction

InstrDescr RegsDescr AddrMode

7654 3210 7654 3210 7654 3210

0101	MMMM	1111	SSSS	UUUU	AAAA

Jumps unconditionally or conditionally, according to the instruction modifier, to the address defined *operand*. The size of the instruction is three or five bytes (depending on the addressing method). Depending on the modifier, the instruction performs the following type of jump:

MMMM==0b0000: /* jmp */ pc<=*operand*; MMMM==0b0001: /* equal */ if (equal) pc<=*operand*; MMMM==0b0010: /* jne */ if (not equal) pc<=*operand*; MMMM==0b0011: /* jgt */ if (signed greater) pc<=*operand*;

Atomic value substitution instruction

InstrDescr RegsDescr

7654 3210 7654 3210

0110	0000	DDDD	SSSS

Replaces the value of the destination and source registers atomically without the possibility of the replacement being interrupted by an asynchronous abort request. The size of the instruction is two bytes.

temp<=reg[DDDD]; reg[DDDD]<=reg[YYYY]; reg[SSSS]<=temp;

Instruction of arithmetic operations

InstrDescr RegsDescr

7654 3210 7654 3210

0111	MMMM	DDDD	SSSS

Performs the appropriate arithmetic operation, according to the instruction modifier, on the values in the destination and source registers. The size of the instruction is two bytes. Depending on the modifier, the instruction performs the following operation:

MMMM==0b0000: /* add */ reg[DDDD]<=(reg[DDDD] + reg[SSSS]); MMMM==0b0001: /* sub */ reg[DDDD]<=(reg[DDDD] - reg[YYYY]); MMMM==0b0010: /* mul */ reg[DDDD]<=(reg[DDDD] * reg[YYYY]); MMMM==0b0011: /* div */ reg[DDDD]<=(reg[DDDD] / reg[YYYY]); MMMM==0b0100: /* cmp */ temp<=(reg[DDDD] - reg[YYYY]); updt psw;

Instruction of logical operations

InstrDescr		RegsDescr	
7654 3210		7654 3210	
1000	MMMM	DDDD	SSSS

Performs the appropriate logical operation, according to the instruction modifier, on the values in the destination and source registers. The size of the instruction is two bytes. Depending on the modifier, the instruction performs the following operation:

```
MMMM==0b0000: /*     not */ reg[DDDD]<=~reg[DDDD];
MMMM==0b0001: /*     and */ reg[DDDD]<=(reg[DDDD] & reg[YYYY]);
MMMM==0b0010: /*      or */ reg[DDDD]<=(reg[DDDD] | reg[YYYY]); xor */
MMMM==0b0011: /*     reg[DDDD]<=(reg[DDDD] ^ reg[YYYY]);
MMMM==0b0100: /* test */ temp<=(reg[DDDD] & reg[YYYY]); updt psw;
```

Instruction of moving operations

InstrDescr		RegsDescr	
7654 3210		7654 3210	
1001	MMMM	DDDD	SSSS

Performs the appropriate shift operation, according to the instruction modifier, on the values in the destination and source registers. The size of the instruction is two bytes. Depending on the modifier, the instruction performs the following operation:

```
MMMM==0b0000: /* shl */ reg[DDDD]<=(reg[DDDD] << reg[YYYY]); updt psw;
MMMM==0b0001: /* shr */ reg[DDDD]<=(reg[DDDD] >> reg[YYYY]); updt psw;
```

Data loading instruction

InstrDescr		RegsDescr		AddrMode	
7654 3210		7654 3210		7654 3210	
1010	0000	DDDD	SSSS	UUUU	AAAA

Loads the data defined *operand* to the destination registry. The instruction size is three or five bytes (depending on the addressing method).

```
reg[DDDD]<=operand;
```

Instruction for storing data

InstrDescr		RegsDescr		AddrMode	
7654 3210		7654 3210		7654 3210	
1011	0000	DDDD	SSSS	UUUU	AAAA

Places the value from the destination register into the data defined *operand*. Size instruction is three or five bytes (depending on the addressing method).

```
operand<=reg[DDDD];
```

All combinations of instructions and operands, for which there is no reasonable interpretation, declare as an error.

Description of the terminal

The terminal is an input/output peripheral consisting of a display (output) and keyboard (input). The terminal has two programmatically accessible registers term_out and appointment which are accessed through the memory address space (memory mapped registers). The listed program accessible registers are mapped into the memory address space as follows:

Register	Address range
term_out	[0xFF00-0xFF01]
appointment	[0xFF02-0xFF03]

Register term_out represents the output data register. The terminal monitors entries in this register and every time a value is entered in term_out the register prints the character on the display. The character printed is determined by the contents of the ASCII table for the value typed in term_out register.

Register appointment represents the input data register. The terminal performs the following two operations each time any button on the keyboard is pressed: (1) writes the ASCII code of the pressed key to appointment register so that by reading its value it is possible to determine which button was pressed and (2) generate an interrupt request. The two previously described operations are performed by the terminal as soon as any button is pressed; where it should be emphasized that the terminal by no means waits for the input terminator, that is *enter* button. The terminal does not *echo* of the pressed button, i.e. it does not show it on the display. For the purposes of implementing this emulator functionality, it is possible, for example, to use <termios.h> header. If *user program* does not read the value appointment register in time, that is, before the user presses the next button, the ASCII code of the previously pressed button will be irretrievably lost. ASCII codes should not be buffered in the frame *implementation*. Consider that the processor of the observed computer system is fast enough to be guaranteed, if it is *user program* written correctly, it can manage to read the value appointment register in the interrupt routine before it is overridden.

Description of the timer

A timer is a peripheral that periodically generates an interrupt request. The timer has one programmable register team_cfg which is accessed through the memory address space (memory mapped register). The specified program accessible register is mapped into the memory address space as follows:

Register	Address range
team_cfg	[0xFF10-0xFF11]

Register team_cfg represents the timer configuration register. The period of generation of the interrupt request by the timer is defined by the value team_cfg registry as follows: 0x0 -> 500ms, 0x1 -> 1000ms, 0x2 -> 1500ms, 0x3 -> 2000ms, 0x4 -> 5000ms, 0x5 -> 10s, 0x6 -> 30s and 0x7 -> 60s. Initial value team_cfg registry, after starting or resetting the emulated computer system, it is 0x0000.

Example of a user program

The program shown below is an example of a program written in assembler described in the first task. The source assembly code of the program is separated into two text files *interrupts.s* and *main.s*. The program performs the following actions:

- defines the content of the IVT, i.e. sets the addresses of interrupt routines,
- the terminal interrupt routine prints on the display the character corresponding to the pressed button,
- the timer interrupt routine prints the character 'T' on the display,
- configure the timer to generate an interrupt request every second and
- waits for the user to press the button five times before stopping the processor.

```
# file:        interrupts.s
. section      ivt
      . word   isr_reset
      . skip 2 # isr_error
      . word   isr_timer
      . word   isr_terminal
      . skip   8
. external   my_start,      my_counter
. section      isr
. equ   term_out,     0xFF00
. equ   appointment,   0xFF02
. equ ascii_code, 84 # ascii('T')
# reset interrupt routine isr_reset:

      jmp my_start
# interrupt routine for timer
isr_timer:
      push  r0
      ldr   r0,   $ascii_code
      p     r0,   term_out
      priest r0
      iret
# interrupt routine for the isr_terminal
terminal:
      push  r0
      push  r1
      ldr   r0,   appointment
      p     r0,   term_out
      ldr r0, %my_counter # pcrel ldr
            r1,   $1
      add   r0,   r1
      str r0, my_counter # abs pop
            r1
      priest r0
      iret
. end
```

```
# file:        main.s
. global     my_start
. global     my_counter
. section my_code
. equ   team_cfg,   0xFF10
. equ   init_sp,       0xFEFE
my_start:
      ldr   r6,   $init_sp
      ldr   r0,   $0x1
      p     r0,   team_cfg
wait:
      ldr r0, my_counter
      ldr r1, $5
      cmp r0, r1
      etc   wait
      halt
. section      my_data
my_counter:
      . word 0
. end
```

The commands to compile, link, and emulate this example are:

```
. /assembler -o interrupts.o interrupts.s
. /assembler -o main.o main.s
. /linker -hex -place=ivt@0x0000 -o program.hex interrupts.o main.o
. /emulator program.hex
```

Project submission

It is necessary to implement the previously described tools according to the given requirements under the Linux operating system on the amd64 architecture using the C/C++ programming language. The necessary tools are described in the materials for lectures and exercises. It is recommended to create the solution within the VMware virtual machine that can be downloaded from the eLearning platform. The virtual machine contains all the necessary tools already installed, and as an integrated environment for program development it is possible to use VS Code, which should be connected to the GNU toolchain set.

Setting up the environment required to successfully compile the source code and run the program is also part of the task setup. The defense of the solution to the project task is performed exclusively under the previously described environment within the virtual machine. Access to the Internet during the defense of the project assignment solution will be disabled, which means that only the tools available on the virtual machine should be used.

Rules for project submission

The project is delivered exclusively as a single .zip archive containing only one name directory *solution* the contents of which are described below. Inside the directory *solution* can be found (1) optional *makefile* or some other script to translate the solution of the task, (2) an optional subdirectory *Misc* provided for additional things like flex and bison input files (output files generated by these tools should not be submitted at all as they will be generated) and (3) mandatory following three subdirectories: *tests*, *heart* and *Inc*. The contents of these subdirectories should be as follows:

- *tests*: input files to display the functionality of the task solution,
- *heart*: all .ci .cpp files with source code and
- *Inc*: all .hi .hpp files with source code.

The described content of the .zip archive should also be its only content. It is not allowed to submit executable files, files generated by flex and bison tools or anything else not explicitly mentioned above. Failure to comply with the rules for submitting the project regarding the content, structure and name of the directory will result in negative points or a ban on appearing for the defense in the given exam period.

A primitive example (the solution should definitely contain a much larger number of source code files than shown) of the contents of the .zip archive is below:

```
.
└── solution
    ├── makefile
    ├── Misc
    │   ├── flex
    │   └── bison
    ├── Inc
    │   ├── assembler.hpp
    │   ├── linker.hpp
    │   └── emulator.hpp
    ├── heart
    │   ├── assembler.cpp
    │   ├── linker.cpp
    │   └── emulator.cpp
    └── tests
        ├── test1.s
        ├── test2.s
        └── test3.s
```

Record of revisions

This record contains a list of changes and additions to this document by version.

Version 1.1

The side	Change
19	Added initialization of pointer to top of stack.

Version 1.2

The side	Change
4	Changed rules for project evaluation
20	Slightly modified description of project submission

Version 1.3

The side	Change