PRACTICAL PART – THE TC COMPUTER EMULATOR

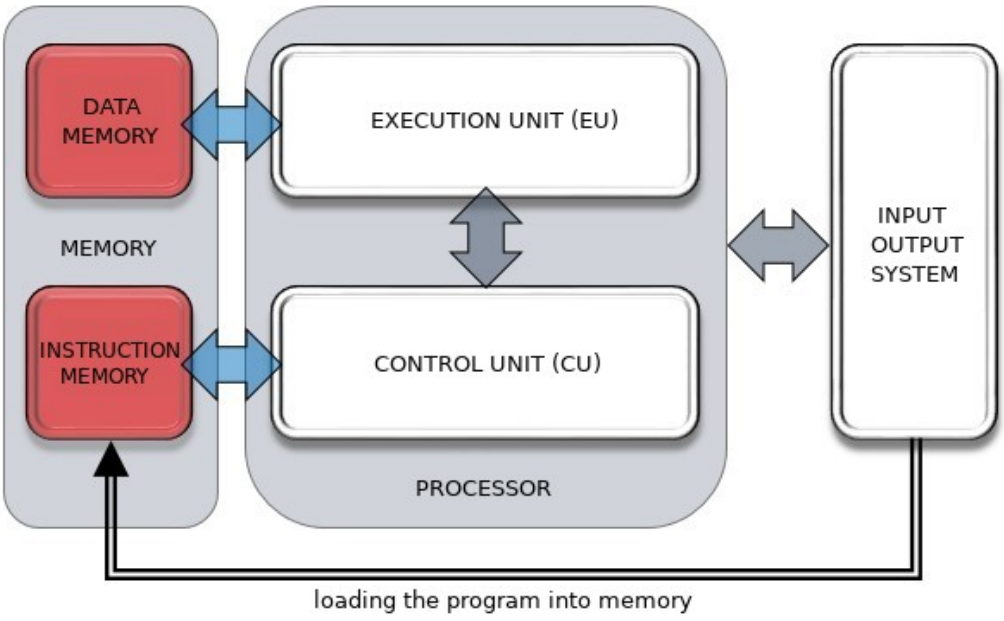
The practical part of this course will involve **low-level programming** for a fake computer designed only for the purpose of this course, called **TAK computer** (or **TC**).

Since TC does not exist physically, we will in fact write programs for an **emulator** of its architecture, running on standard (e.g. x86) machines. The emulator will accept a program, written in an assembly-style **TAK computer language** (**TCL**), and simulate how it would execute on a TC computer. The emulator will be called **TCE** (**TAK Computer Emulator**).

Some aspects of the language will be denoted as **[OPTIONAL]** – this means that you won't strictly need them to solve any programming task; however, they touch important and standard aspects of low-level programming, so you're welcome to understand them (and potentially use them to simplify or optimize your code).

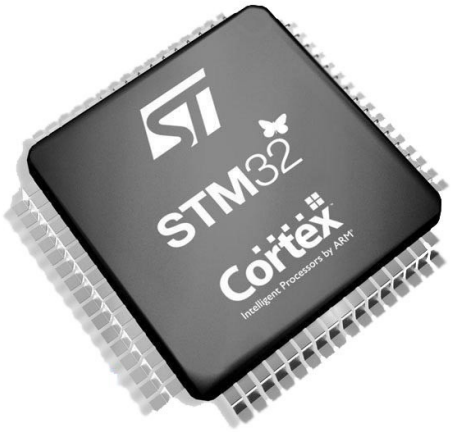
TC computer – architecture model

The architecture model of the TC computer is based on the von Neumann model, though with one important difference – the memory block will be split to two disjoint sub-blocks: **data memory** and **instruction memory**.



The architecture model of a TC computer.

This complies with the *Harvard computer model*, which analogously distinguishes data memory and instructions memory. Such split can be found in practice in **micro-controllers**, where data memory is often of a volatile RAM type, while the instructions are essentially fixed and hence stored in permanent memory of ROM type.



A 32-bit CORTEX microcontroller – with clock frequency 24 MHz, 8 kiB RAM and 64 kiB ROM (source: www.botland.com.pl)

Note

Micro-controllers play a crucial role in many areas where electronics touches everyday life. They are installed e.g. in cars, intelligent fridges, TVs, and even toys.

TC computer – architecture details

To define the architecture of TC computer in full detail, we need to specify e.g. the sizes of various types of its memory. For a broader generality, we will introduce several **constructional parameters**:

- ***K*** – the size of a processor register or a data memory cell (in bits),
- ***N*** – the number of regular processor registers,
- ***S*** – the number of data memory cells,
  - ***B*** – the address of the beginning of stack inside data memory,
- ***T*** – the number of instruction memory cells.

Let's now discuss the computer units in more detail.

Processor

The processor consists of the following binary **registers** (of size *K* bits each):

- **regular registers** in the execution unit (EU):

    - *N* registers purposed for storing temporary data: arguments for or results of processor instructions,
    - indexing: from 0 up to *N*-1: R0, R1, …, R(*N*-1),
- **special registers** in the control unit (CU):
    - **IAR** (Instruction Address Register) – always storing the address of the currently executed instruction in the instruction memory,
    - **IRCR** (Instruction Return Code Register) – storing the **return code** of most recently executed instruction, i.e. 0 if the instruction succeeded, and a number indicating the type of error otherwise; the details will depend on instruction type and will be discussed below,
    - **SHR** (Stack Head Register) – storing the current address of the stack head in the data memory.

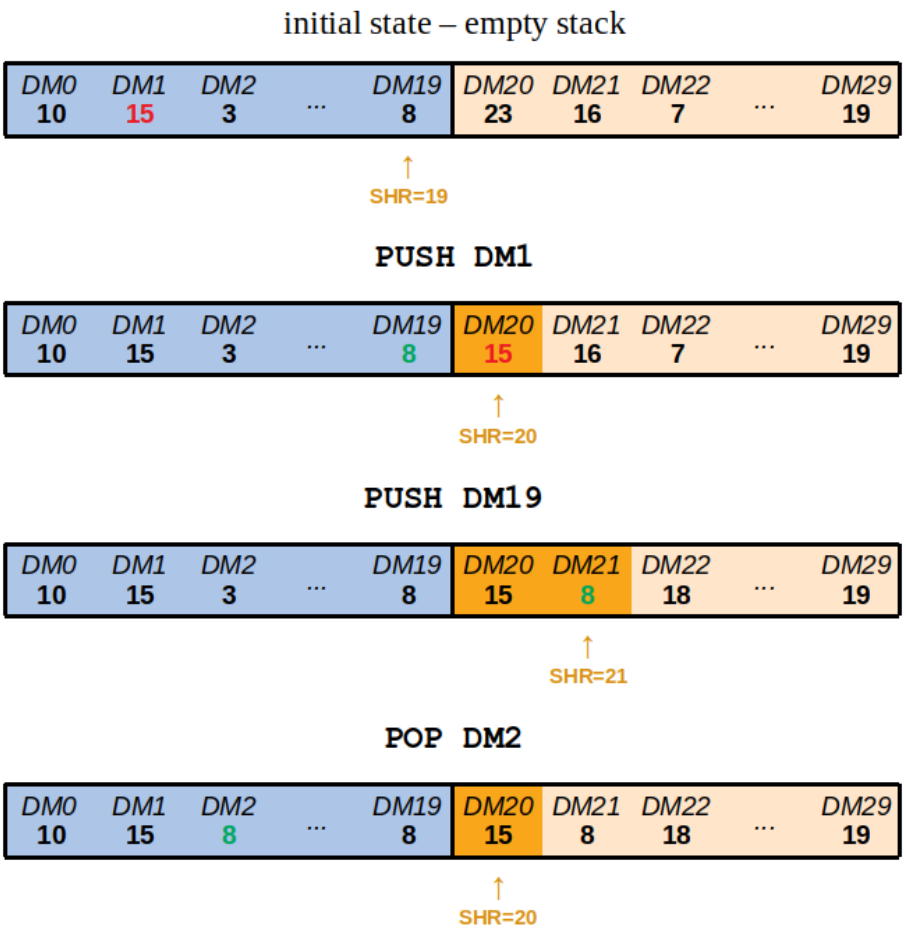
  Special registers can be used in TCL code just like normal registers, except for that only control unit can modify their values – they **can't be modified** directly **in the code**. Any such attempt (like e.g. `MOV IAR IRCR`) will produce an error.

Data memory

The data memory (in short: DM) consists of *S* **cells** (of size *K* bits each), denoted DM0, DM1, …, DM(*S*-1). It is further split into **two blocks** with different type of access:

- the first *B* cells (DM0, DM1, …, DM(*B*-1)) are accessible **directly** by their indices (e.g. „store the value from R0 into DM3" or „load the value from DM5 into R1"),
- the remaining cells (DM(*B*), DM(*B*+1), …, DM(*S*-1)) constitute the memory buffer for the **stack** – a list of values which can be **pushed** (added at the end) or **popped** (removed from the end), rather than accessed directly by their index.
  At every moment, the stack (if not empty) extends from the memory cell DM(*B*) up to its **head** which is located in the cell DM(SHR), i.e. the cell

indexed by the current value of SHR (Stack Head Register).



An example sequence of actions on the stack (dark orange) within the second (light orange) part of the data memory, for S = 30 and B = 20. The bold numbers are the values stored in the data memory cells

The initial value of SHR is *B*-1, which means that the stack is **empty**. **Pushing** the first element would increase SHR to *B*, and store the value at DM(*B*). Pushing the next element would increase SHR to *B*+1, and store the value at DM(*B*+1), and so on.
Popping works conversely: reads the value of DM(SHR), and then decreases SHR by one.

The memory split described above makes TC partly a **stack machine**. This functionality will be important for **procedural programming**; see the final section of this document.

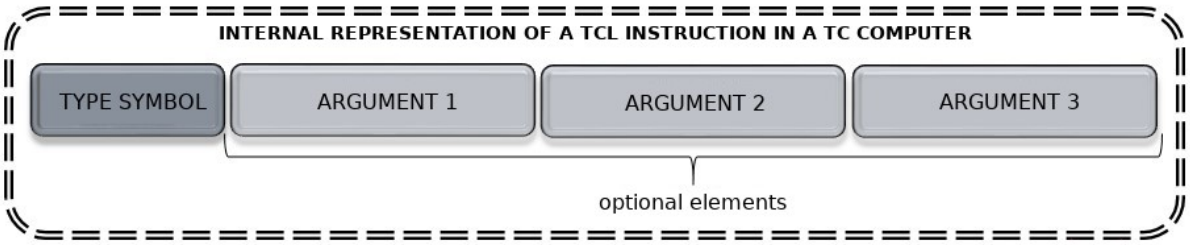
Instruction memory

The instruction memory (IM) consists of T **cells**, IM0, IM1, ..., IM(*T*-1), each representing one **instruction** together with any its arguments. To have TC execute a program, the consecutive instructions of that program should be stored in IM0, IM1, and so on.

Each instruction consists of:

- a **type symbol**, indicating the type of instruction (e.g. „add two numbers", „move a value from one location to another", etc.),
- optionally, up to three **arguments** specifying the details (e.g. which numbers should be added).

  While in TCL instructions have a **symbolic** representation (e.g. „ADD R0 R1 R2" stands for „add values from R0 and R1 and store the result in R2"), the instruction memory cells contain **binary representations** of such instructions:



The structure of a binary representation of an instruction within the TC computer.

An IM cell is split to four blocks of bits, encoding the type symbol and arguments (or null value when a corresponding argument is absent). Also, the whole cell can be null, indicating that it is empty (there's no instruction stored in it).

The length of an IM cell (in bits) is chosen so that each block can fit the necessary information (instruction type, index of a register, index of a data memory cell etc. - note that all of these are bounded by the constructional parameters). We will not dive into deep details here as they are invisible from the programmer's perspective.

Input-output system

The input-output system of the TC computer has the following functionality:

- **input**: reading values from the standard input and storing them in registers or data memory;
- **output**: sending to the standard output values of registers, data memory cells, as well as reporting when program execution has finished.

  In particular, the input-output system converts all numeric values between the internal binary and external decimal representation.

  In our TCE emulator, the standard input and output will be available through the standard command line console. The emulator will also be responsible for reading all the constructional parameters and the code of a program, which will be treated as loaded into the instruction memory of the emulated TAK computer.

TC computer – computing model

Let us now describe in detail how the program specified by the user is loaded into a TC computer and executed by it.

A **program** is a sequence of **instructions**. Each instruction occupies one cell of the instruction memory, so the total number of instructions in the program is limited by T, the size of instruction memory.

Loading the program

When TCE is asked to run a program consisting of L instructions, it first checks whether $L \leq T$ and whether all instructions are syntactically correct, throwing a **program loading error** if necessary. If all is correct, TCE stores the instructions into the first L cells of instruction memory, and sets the remaining cells to be empty.

Executing the program

At the beginning of executing any TC program, all the registers are set to zero, except for SHR (Stack Head Register), whose value is set to B-1 (indicating that the stack is empty). In particular, IAR (Instruction Address Register) is set to 0, which guarantees that executing the program will start from its first instruction.

Then, the following actions are taken in a loop, until the execution finishes:

- the current instruction (indexed by the value of IAR) is loaded into the execution unit (EU),
- the EU performs the computations related to the instruction, optionally using the data memory block,
- the control unit (CU) determines the value of return code from the instruction and stores it in the IRCR register; value 0 denotes successful execution, a positive value denotes an error; in the latter case **executing the program is halted**,
- the value of IAR is incremented by one.

Successful termination of a program

Assuming that all instructions in the program are syntactically correct and do not cause any execution error, executing the program can be terminated in two situations:

- IAR $\geq L$ – execution has just reached an empty memory cell,
- IAR $= T$ (possible when $L = T$) – we have just executed the last instruction of the program, stored in IM(T – 1), which was not a jump instruction. In this case, there are no further instructions to be executed.

Execution errors

An execution error occurs whenever an attempt to execute a (syntactically valid) instruction leads to an action which would be incorrect **algorithmically** (e.g. division by 0) or due to hardware **limitations** (e.g. accessing memory cell with index out of bounds; arithmetic overflow). In each of these cases, the control unit halts program execution, and – through the input-output system – passes the values of IAR and IRCR to the outside environment. This lets the user know which kind of error has occurred and which instruction has triggered it.

Introduction to the TCL language

We will now describe the basic rules of the **TAK Computer Language** (**TCL**) which will be used to program the TC computer. TCL is an assembly language, sticking to machine-level concepts but offering their symbolic representation, in a fashion similar to existing assembly languages.

A **program** is represented as a text input (provided in a file or directly from terminal), in which **each line** always corresponds to one **instruction**. For instance, the following program:

```
SET R0 1   // store value 1 in register R0
SET R1 13  // store value 13 in register R1
OUT R0     // print the value of R0 to the output
```

consists of three instructions, which will be stored correspondingly in IM0, IM1, IM2.

The **instruction structure** is simple: it always starts with a type symbol, followed by all the arguments. For programmer's convenience (in particular, for best compatibility with various conventions of existing assembly languages), we introduce the following conventions:

- the language is case-insensitive,
- type symbol and arguments can be separated by any non-empty sequence of whitespace characters (spaces, tabs) and commas,
- any of characters: `/ # ;` starts a comment (ignored portion of code) spanning until the end of the current line.

  To summarize, the following lines of code are all valid, and mutually equivalent:

```
ADD R1 R2 R3          // add values of R1 and R2; store result in R3
ADD   R1,   R2,   R3  ; this does the same
add   r1,r2,r3        # so does this
```

Addressing modes for instruction arguments

Before we list all the instruction types, let us discuss in general what kinds of arguments are available in the language.

In every instruction, every argument must denote a **value** and/or refer to a **memory cell**. In TCL, there are three ways to do this (called **addressing modes**):

- **immediate addressing** – the argument is just a number (given in decimal form) whose value should be used in the computation, for example:

```
SET R0 3 // store the value 3 in register R0
```

- **direct addressing** – the argument specifies a data memory cell, which should be either written to or read from. Examples:

```
IN DM0  // store the value read from input in cell DM0
OUT DM1 // print the value stored in cell DM1 to the output
```

  Note that only the first *B* data memory cells (DM0, DM1, ..., DM(*B*-1)) are directly accessible; the remaining cells serve as stack and an attempt to access them directly will result in a program loading error.

- **register addressing** – the argument specifies a register, which should be either written to or read from. Examples:

```
IN R0  // store the value read from input in register R0
OUT R1 // print the value stored in register R1 to the output
```

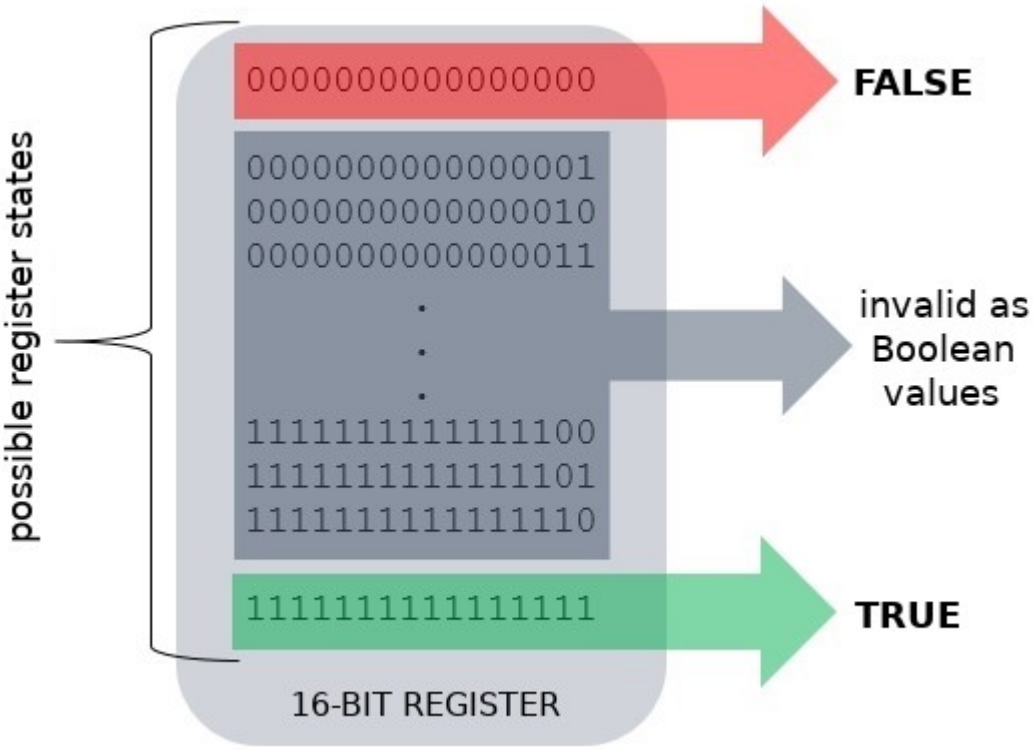
This list is simplified comparing to real-life assembly languages; for instance, the Intel 8086 processors (introduced in 1978) supported 7 addressing modes. Our choice of 3 basic addressing modes will be enough for our simple applications.

Note that a single instruction may combine **different addressing modes**. On the other hand, keep in mind that most instruction types place **restrictions** on addressing modes of their arguments – for instance, the single argument of `IN` cannot use immediate addressing (indeed, `IN 15` would not make any sense). The details for every instruction type will be described below.

Data types

TCL supports just two data types:

- **Signed integers** which can fit within one register (or memory cell). We use the **two's complement** convention (see Lecture 7). This means that, given the *K*-bit cells, the range of numbers supported by the language is from -2^{K-1} to $2^{K-1} - 1$, inclusive. Whenever execution unit gets involved in an arithmetic operation whose result would fall outside this range, the result will be an execution error of **arithmetic overflow**. Also, specifying numeric constants outside this range gives an error.
- **Boolean values**: true or false. Since registers and memory cells have *K* bits, we need to clarify how Boolean values will be represented in them. Our convention is to replicate the same bit *K* times, where 1 stands for truth, and 0 for false (see the picture).



Representation of boolean values in a 16-bit register.

TCL instruction types

We will now present all the instruction types available in TCL. For each of them, we will describe its syntax and semantics, including:

- the number of arguments and their allowed addressing modes
- the possible **return codes** – values of IRCR register after executing the instruction,
- **[OPTIONAL]** number of **clock cycles** spent on executing the instruction - this won't play any direct role in the programming tasks, but will be useful for those who'd like to fight for better efficiency.

System instructions

These instructions allow global manipulations on the memory state.

CLEAR_REG

Syntax: `CLEAR_REG`

Semantics: Clears all regular registers in the machine (R0, R1, ..., R(*N*-1)), setting their value to 0.

Addressing: N/A (no arguments)

Return code: 0 – success

Clock cycles: *N* (the number of regular registers)

CLEAR_DATA_MEM

Syntax: `CLEAR_DATA_MEM`

Semantics: Clears all memory cells in the machine (DM0, DM1, …, DM(*S*-1)), setting their value to 0.

Addressing: N/A (no arguments)

Return code: 0 – success

Clock cycles: 5·*S* (where *S* = the number of data memory cells)

DUMP_REG

Syntax: `DUMP_REG`

Semantics: Prints consecutively values of all registers (R0, R1, …, R(*N*-1)) to the output.

Addressing: N/A (no arguments)

Return code: 0 – success

Clock cycles: 51·*N* (where *N* = the number of registers)

DUMP_DATA_MEM

Syntax: `DUMP_DATA_MEM`

Semantics: Prints consecutively values of all data memory cells (DM0, DM1, …, DM(*S*-1)) to the output.

Addressing: N/A (no arguments)

Return code: 0 – success

Clock cycles: 51·*S* (where *S* = the number of data memory cells)

Data transmission

IN

Syntax: `IN ARG1`

Semantics: Reads a number (specified in decimal form) from the input and stores its value in the register/memory cell specified by ARG1.

Addressing: register/direct

Return code: 0 – success

1 – the provided string did not represent an integer

2 – overflow: the given integer did not fit within *K* bits

3 – empty input

Clock cycles: 51 – in register addressing mode

55 – in direct addressing mode

Examples: `IN R1` – reads a number and stores it in R1

`IN DM2` – reads a number and stores it in DM2

OUT

Syntax: `OUT ARG1`

Semantics: Prints (in the decimal form) the value stored in ARG1 to the output.

Addressing: register/direct

Return code: 0 – success

Clock cycles: 51 – in register addressing mode

55 – in direct addressing mode

Examples: `OUT R1` – prints the value of R1

`OUT DM2` – prints the value of DM2

MOV

Syntax: `MOV ARG1 ARG2`

Semantics: Copies the value stored in register/memory cell specified by ARG1 to the register/cell specified by ARG2.

Addressing: register/direct

Return code: 0 – success

Clock cycles: 2 – in register-register addressing mode

6 – in register-direct or direct-register addressing mode

10 – in direct-direct addressing mode

Example: `MOV R1 DM2` – copies the value from R1 into DM2

SET

Syntax: `SET ARG1 ARG2`

Semantics: Sets the new value of register/memory cell specified by ARG1 to be the number specified by ARG2.

Addressing: ARG1: register/direct

ARG2: immediate

Return code: 0 – success

Clock cycles: 11 – in register addressing mode

16 – in direct addressing mode

Example: `SET DM2 -15` – stores the number -15 in memory cell DM2

PUSH

Syntax: `PUSH ARG1`

Semantics: Pushes the value of register/memory cell specified by ARG1 to the top of the stack (growing the stack by 1). This means three steps:

(i) increase the value of SHR by one,

(ii) read the value of ARG1,

(ii) store that value in the new stack head, i.e. the cell DM(SHR).

Addressing: register/direct

Return code: 0 – success

1 – stack overflow error (SHR = *S*) – the stack has filled the whole available memory buffer; no room for the new value

Clock cycles: 8 – in register addressing mode

12 – in direct addressing mode

Example: `PUSH R1` – pushes the value stored in R1 to the head of the stack

POP

Syntax: `PUSH ARG1`

Semantics: Pops the value off the current head of the stack (decreasing the stack size by 1), and stores it in the register/memory cell specified by ARG1. This means three steps:

(i) read the current stack head, i.e. the value of DM(SHR),

(ii) store that value in the register/memory cell specified by ARG1,

(iii) decrease the value of SHR by one.

Addressing: register/direct

Return code: 0 – success

1 – empty stack (SHR = *B*-1) – no value to be popped from it

Clock cycles: 8 – in register addressing mode

12 – in direct addressing mode

Example: `POP DM3` – pops the value from the head of the stack to DM3

Logic operations

SETT / SETF

Syntax:
```
SETT ARG1
SETF ARG1
```

Semantics: Stores the Boolean truth (11...1) / false (00...0) in the register ARG1.

Addressing: register

Return code: 0 – success

Clock cycles: 1

NOT

Syntax:
```
NOT ARG1 ARG2
```

Semantics: Computes the negation of the Boolean value from register ARG1; stores it in the register ARG2

Addressing: register

Return code: 0 – success

1 – the value of register ARG1 did not represent a Boolean value

Clock cycles: 1

AND / OR / XOR / NAND / NOR

Syntax:
```
AND  ARG1 ARG2 ARG3
OR   ARG1 ARG2 ARG3
XOR  ARG1 ARG2 ARG3
NAND ARG1 ARG2 ARG3
NOR  ARG1 ARG2 ARG3
```

Semantics: Performs the specified logic operation on the Boolean values from registers ARG1 and ARG2; stores the result in the register ARG3. For explanation of the five operations (AND / OR / XOR / NAND / NOR), see Lecture 6.

Addressing: register

Return code: 0 – success

1 – register ARG1 or ARG2 did not contain a Boolean value

Clock cycles: 2

Example: `AND R1 R3 R0` – stores the conjuction of the Boolean values from R1 and R3 into R0

Arithmetic operations

NEG

Syntax: `NEG ARG1 ARG2`

Semantics: Computes the negation (e.g. 5 → -5) of the value of register ARG1, and stores it in ARG2. (The arithmetic analogue of `NOT`)

Addressing: register

Return code: 0 – success

1 – arithmetic overflow: the result does not fit in a register

Clock cycles: 1

ABS

Syntax: `ABS ARG1 ARG2`

Semantics: Computes the absolute value (e.g. 5 → 5, -5 → 5) of the value of register ARG1, and stores it in ARG2.

Addressing: register

Return code: 0 – success

1 – arithmetic overflow: the result does not fit in a register

Clock cycles: 1

ADD / SUB / MUL / DIV / MOD

Syntax:

```
ADD ARG1 ARG2 ARG3
SUB ARG1 ARG2 ARG3
MUL ARG1 ARG2 ARG3
DIV ARG1 ARG2 ARG3
MOD ARG1 ARG2 ARG3
```

Semantics: Performs an arithmetic operation (**ADD**ition / **SUB**traction / **MUL**tiplication / integer **DIV**ision / **MOD**ulo) on the numbers stored in registers ARG1 and ARG2, and stores the result in ARG3.

For DIV / MOD involving negative numbers, we follow the C99 convention: *a* MOD *b* has **same sign as *a*** (see „Note" below).

Addressing: register

Return code: 0 – success

1 – arithmetic overflow: the result does not fit in a register

2 – division by 0 (for DIV / MOD operations)

Clock cycles: 4 / 4 / 10 / 30 / 30 – for ADD / SUB / MUL / DIV / MOD

Note

In the leading languages, there are **two** competing **standards** of defining how DIV and MOD should behave for **negative numbers**:

a	*b*	**floor convention** (Python, Ruby, R, Matlab)		**C99 (truncate) convention** (Java, modern C/C++, C#, JS)	
		a DIV *b*	*a* MOD *b*	*a* DIV *b*	*a* MOD *b*
13	10	1	3	1	3
-13	10	-2	7	-1	-3
13	-10	-2	-7	-1	3
-13	-10	1	-3	-1	-3

To help understanding, let's note that both conventions adhere to the **main principle** relating DIV and MOD operations to each other:

$$a = (a \text{ DIV } b) \cdot b + (a \text{ MOD } b).$$

Now, in the **floor convention**, the division result is **rounded down**, or *equivalently*, the **remainder** (MOD) has always the **sign of *b***.

In the **truncate (C99) convention**, on the other hand, the result is **rounded towards zero** (so down if it's positive, up if it's negative), or *equivalently*, the **remainder** has the **sign of *a***.

While the floor convention is more compatible with standard math (at least for $b > 0$), **in TCL** we'll follow the **C99 convention**, as the leading one particularly for lower-level languages.

INC / DEC

Syntax:
```
INC ARG1 ARG2
DEC ARG1 ARG2
```

Semantics: Increases / decreases the value stored in register ARG1 by the number represented by ARG2

Addressing: ARG1 – register
ARG2 – immediate

Return code: 0 – success

1 – arithmetic overflow: the result does not fit in a register

Clock cycles: 14

Example: `INC R1 12` – increases the number stored in R1 by 12.

Arithmetic relations

CMP(EQ / NEQ / LT / GT / LE / GE)

Syntax:
```
CMPEQ  ARG1 ARG2 ARG3
CMPNEQ ARG1 ARG2 ARG3
CMPLT  ARG1 ARG2 ARG3
CMPGT  ARG1 ARG2 ARG3
CMPLE  ARG1 ARG2 ARG3
CMPGE  ARG1 ARG2 ARG3
```

Semantics: Compares the numbers stored in registers ARG1 and ARG2; stores in ARG3 whether the value from ARG1 is **EQ**ual / **N**ot **EQ**ual / **L**ess **T**han / **G**reater **T**han / **L**ess than or **E**qual / **G**reater than or **E**qual the value from ARG2.

Addressing: register

Return code: 0 – success

Clock cycles: 3

Example: `CMPLT R1 R5 R2` – stores true (as Boolean value) in R2 if the value from R1 is less than the value from R5; stores false (as Boolean value) in R2 otherwise

Flow control instructions

JMP

Syntax: `JMP ARG1`

Semantics: Makes a **jump** to the instruction indexed by the value of ARG1 – the next instruction to be executed will be IM(ARG1).

Internally, this is achieved by setting IAR to the value of ARG1 *minus one* – as if the current instruction was the one preceding IM(ARG1). Since IAR is increased by one at the end of executing every instruction, this results in executing IM(ARG1) in the next turn.

Addressing: immediate/register

Return code: 0 – success

1 – wrong jump target (invalid instruction memory cell index)

Clock cycles: 4

Example: `JMP 1` – jumps so that the next instruction to be executed is IM1, i.e. the second instruction in the program (reminder – indexing starts from zero!)

JMPT / JMPF

Syntax:
```
JMPT ARG1 ARG2
JMPF ARG1 ARG2
```

Semantics: Makes a **conditional jump** to the instruction indexed by the value of ARG1 – the jump occurs if the value of ARG2 is **T**rue / **F**alse; nothing happens otherwise.

Addressing: ARG1 – immediate/register
ARG2 – register

Return code: 0 – success

1 – wrong jump target (invalid instruction memory cell index)

2 – the value of register ARG2 did not represent a Boolean value

Clock cycles: 6

Example: `JMPF 7 R2` – jumps to IM7 if the value of R2 is a Boolean false; no action if the value of R2 is a Boolean truth; execution error otherwise

JMP(EQ / NEQ / LT / GT / LE / GE)

Syntax:
```
JMPEQ  ARG1 ARG2 ARG3
JMPNEQ ARG1 ARG2 ARG3
JMPLT  ARG1 ARG2 ARG3
JMPGT  ARG1 ARG2 ARG3
JMPLE  ARG1 ARG2 ARG3
JMPGE  ARG1 ARG2 ARG3
```

Semantics: Makes a conditional jump to the instruction indexed by the value of ARG1, under the condition that the value of ARG2 is **EQ**ual / **N**ot **EQ**ual / **L**ess **T**han / **G**reater **T**han / **L**ess than or **E**qual / **G**reater than or **E**qual the value from ARG3.

Addressing: ARG1 – immediate/register

ARG2, ARG3 – register

Return code: 0 – success

1 – wrong jump target (invalid instruction memory cell index)

Clock cycles: 6 / 6 / 7 / 7 / 9 / 9 – for JMP(EQ / NEQ / LT / GT / LE / GE)

Example: `JMPGE 12 R5 R2` – jumps to IM12 if the value of R5 is greater or equal than the value of R2; no action otherwise

SKIP

Syntax: `SKIP`

Semantics: Increases IAR by one.

Effect: skips the execution of the following instruction: if IM(x) is SKIP, then executing it results in a jump to IM(x + 2).

Addressing: N / A – no arguments

Return code: 0 – success

1 – wrong jump target (invalid instruction memory cell index)

Clock cycles: 4

NOP

Syntax: `NOP`
`<empty line>`

Semantics: No action – this instruction does nothing.

Yet, it is a non-trivial instruction. As a result, when IM(IAR) = NOP, the execution unit will do nothing, but the control unit will continue executing the program, proceeding to IM(IAR + 1). On the other hand, when IM(IAR) has zero value, program execution is immediately terminated.

Syntactically, NOP can be represented either with `NOP` or with an **empty line**. As a result, your TCL program may contain empty lines (for readability purposes) which will be interpreted as NOP instructions, and hence will not halt executing your program :)

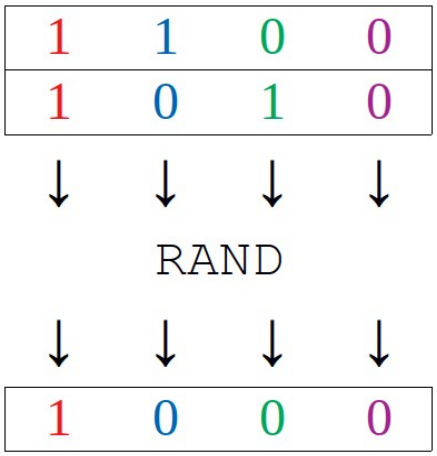
Addressing: N / A – no arguments

Return code: 0 – success

Clock cycles: 1

Bit-wise arithmetics [OPTIONAL]

Bit-wise arithmetics mean applying logical operators to sequences of bits, in a bit-wise fashion. For example, bit-wise AND on sequences 1100 and 1010 gives 1000 (see the picture).



The principle of bit-wise operations. („RAND" denotes here „bit-wise AND").

A related concept is the **bit shift** of a binary sequence. The **left shift** involves erasing the left-most bit of a sequence (hence all the following bits move by one place to the left), and appending a zero at the end. Arithmetically, the result is **multiplication** by 2, though **modulo 2^K**.

The **right shift**, conversely, involves erasing the right-most bit (so moving all preceding one place to the right), and … What should we prepend on the start? One intuitive answer is „a zero, just like for the left shift" - that convention is called the **logical right shift**. On the other hand, if we prefer an operation with a good arithmetic intepretation (namely, integer division by 2), then, to obtain good results for negative numbers, we need the prepended bit to be a **copy** of the **original left-most bit**. This variant is called the **arithmetic right shift**.

These operations are not useful in most intuitive contexts, and do not increase the expressive power of the language. Yet, they gained much popularity in low-level programming due to the simplicity of their hardware implementation, which allows many processors to perform these much faster than the typical arithmetic operations. (For example, on some processors, the most efficient way to store 0 in a register is to tell the processor to replace this value with the bit-wise xor of it with itself).

RNOT

Syntax: `RNOT ARG1 ARG2`

Semantics: Computes the bit-wise negation (e.g. 3 = 0...011 → 1...100 = -4) of the value in register ARG1 and stores it in register ARG2.

In the 2's complement convention, this operation always transforms an integer value *x* into -1 – *x*.

Addressing: register

Return code: 0 – success

Clock cycles: 1

RAND / ROR / RXOR / RNAND / RNOR

Syntax:

```
RAND  ARG1 ARG2 ARG3
ROR   ARG1 ARG2 ARG3
RXOR  ARG1 ARG2 ARG3
RNAND ARG1 ARG2 ARG3
RNOR  ARG1 ARG2 ARG3
```

Semantics: Performs a bit-wise logic operation (AND / OR / XOR / NAND / NOR) on the values of registers ARG1 and ARG2, and stores the result in the register ARG3.

Addressing: register

Return code: 0 – success

Clock cycles: 2

RSL / ASL / RSR / ASR

Syntax:

```
RSL ARG1
ASL ARG1
RSR ARG1
ASR ARG1
```

Semantics: Performs a bit shift (**L**eft or **R**ight) of the value of register ARG1, storing the result back in ARG1.

RSL and ASL both perform the (unambiguous) left shift.

RSR performs the logic variant of the right shift, while ASR the arithmetic variant.

Addressing: register

Return code: 0 – success

Clock cycles: 1

Procedural programming

The elements of TCL listed above are powerful enough to realize most basic aspects of imperative programming, as known from popular languages like Java or C++. For instance, proper combinations of **conditional jumps** (JPMT etc.) can be used to build an **equivalent** of „**if-else**" block or „**while**" loop (and then – with some arithmetic – also the „**for**" loop).

However, the story becomes more tricky when it comes to **procedures** (or **functions**), which we understand as pieces of code which you can **call** and then **return** from. How to do this in TCL?

A simple print procedure

Let's start from a simple example, and try to code it with a simple machinery – just jumps.

For the start, let's say we want to code a simple procedure which prints out its single argument *a*. We want to call it two times: for $a = 10$ and $a = 9$. Since the core of data flow in TC is registers, we'll adopt the convention that arguments are passed through initial registers – in this case, through R0.

For better orientation in the code, we provide line numbers on the left – keep in mind though that they are not part of the actual code.

```
0     # start of the program
1     SET R0 10  # set argument value
2     JMP 9      # call „print"
3
4     SET R0 9   # set argument value again
5     JMP 9      # call „print" again
6
7     JMP 11     # halt program execution (jump to empty instruction)
8
9     OUT R0     # „print" STARTS HERE
10    JMP ???    # yeah... where do we jump to return???
```

This **„almost" works**. The first call of our print procedure (i.e. jump to line 9) happens in line 2, so at the end of executing the procedure (in line 10) we'd like to jump back to line 3 – just after the calling point. But the next time, we call the procedure from line 5, and we want the returning jump in line 10 to target line 6!

How to do this? The simple answer is: **store** the calling location based on **IAR**, and **use** this knowledge at return time. Below, we use R1 for that storage:

```
0     # start of the program
1     SET R0 10  # set argument value
2     MOV IAR R1 # store where we are
3     JMP 11     # call „print"
4
```

```
5     SET R0 9    # set argument value again
6     MOV IAR R1  # store where we are
7     JMP 11      # call „print” again
8
9     JMP 15      # halt program execution (jump to empty instruction)
10
11    OUT R0      # „print” STARTS HERE
12    INC R1 2    # compute the instruction to which we want to return
13    JMP R1      # return!
```

Here's the trick: before each call of „print”, we store the current IAR value (number of line we're at) in R1. At the end of the „print” procedure, we'd like to return to the next line after the calling jump, that is, **two lines after** the line in which we saved IAR (assuming that we wisely always save **right before** the calling jump). For this, it's sufficient to increase R1 by 2, and do „JMP R1”.

The more general picture

As we saw above, the store-caller-place-in-register trick works for the above simplistic example. But what if we'd like a procedure to **call another** procedure?

They can't both use R1 for keeping the return instruction number, as one would destroy the value saved by the other one. Should the other one use R2? That would work for a simple picture, but not if we want to have **recursion** (a procedure calling directly or indirectly itself).

The general solution here is to use **stack** for accumulating and disposing respective return locations, correspondingly to the **call stack** of invocations of all our functions. In each call, we push the current IAR to the stack; in each return, we pop the last-pushed value, increase it by 2, and return there. That's simple, and has an additional advantage that no registers are *permanently* occupied for the purpose of return address tracking. (Well, some register is still needed *temporarily*, for just the „increase by 2” part).

To take an example, we will write a „print2” procedure which, given a non-negative number *n*, prints out the numbers *n*, *n*-1, ..., 1, 0. We will implement it **recursively**: „call print(*n*); then, if *n* > 0, call print2(*n*-1)”. Here's the code, together with two sample calls, print2(10) and then print2(9):

```
0     # start of the program
1     SET R0 10   # set argument value
2     PUSH IAR    # store (push to stack) where we are
3     JMP 16      # call „print2”
4
5     SET R0 9    # set argument value again
6     PUSH IAR    # store (push to stack) where we are
7     JMP 16      # call „print2” again
8
9     JMP 30      # halt program execution (jump to empty instruction)
10
11    OUT R0      # „print” STARTS HERE
```

```
12    POP R1      # get (pop from stack) the last stored IAR value
13    INC R1 2    # compute the instruction to which we want to return
14    JMP R1      # return!
15
16                # „print2” STARTS HERE
17    PUSH IAR    # store (push to stack) where we are
18    JMP 11      # call „print” for „n”
19    SET R1 0
20    JMPEQ 24 R0 R1  # skip recursive part if „n” equals 0
21    DEC R0 1    # decrease „n” by one
22    PUSH IAR    # store (push to stack) where we are
23    JMP 16      # call „print2” for „n-1”
24                # RETURNING FROM „print2” STARTS HERE
25    POP R1      # get (pop from stack) the last stored IAR value
26    INC R1 2    # compute the instruction to which we want to return
27    JMP R1      # return!
```

Yay, **this works**! Pity though that we get the numbers printed in the reversed order: *n*, *n*-1, ..., 1, 0. Let's make it increasing: 0, 1, ..., *n*-1, *n*, by switching the order of actions within „print2”: first call print2(*n*-1) (if suitable), and only then call print(*n*). It looks like only „print2” part needs changes, so let's rewrite only that:

```
16                # „print2” STARTS HERE
17    SET R1 0
18    JMPEQ 23 R0 R1  # skip recursive part if „n” equals 0
19    DEC R0 1    # decrease „n” by one
20    PUSH IAR    # store (push to stack) where we are
21    JMP 16      # call „print2” for „n-1”
22    INC R0 1    # get the value of „n” back – SUBOPTIMAL!
23    PUSH IAR    # store (push to stack) where we are
24    JMP 11      # call „print” for „n”
25                # RETURNING FROM „print2” STARTS HERE
26    POP R1      # get (pop from stack) the last stored IAR value
27    INC R1 2    # compute the instruction to which we want to return
28    JMP R1      # return!
```

OK, so the blue lines went down, as expected, but why did the red line appear? This is to **restore** the desired value of a „**local variable**” - in this case, R0 – which had been earlier decreased by one. Well, this trick **worked luckily** in our simple case, but generally, it often happens that procedures compute some local values to be used **after** some sub-routine calls which *inrecoverably* mess up the state of memory – quite possibly *all* the memory that we could use as a backup. (Think of recursion again! If you plan to backup R0 in, say, DM237, then your recursive sub-call of the same function will very likely mess up the very same

DM237!) Therefore, the **safe general solution** is just what we did previously with the return address: **save locally used values on the stack**. Like this:

```
16              # „print2” STARTS HERE
17    SET R1 0
18    JMPEQ 24 R0 R1  # skip recursive part if „n” equals 0
19    PUSH R0    # save the original „n” for later
20    DEC R0 1   # decrease „n” by one
21    PUSH IAR   # store (push to stack) where we are
22    JMP 15     # call „print2” for „n-1”
23    POP R0     # restore the original „n”
24    PUSH IAR   # store (push to stack) where we are
25    JMP 11     # call „print” for „n”
26              # RETURNING FROM „print2” STARTS HERE
27    POP R1     # get (pop from stack) the last stored IAR value
28    INC R1 2   # compute the instruction to which we want to return
29    JMP R1     # return!
```

This approach – when followed correctly – allows expressing **arbitrarily complex** recursive computation in an assembly-level language. The most important thing to watch for here is not to mess the order in which local values and return addresses land on the stack. For instance, in the above code, it is crucial that line 18 precedes line 20: since the value of *n* will be popped from the stack (line 22) *after* returning from „print2” sub-call (line 21), it must be pushed to the stack *before* the return address. Otherwise the code would crash.