**Subroutine**

In computers, a subroutine is a sequence of program instructions that perform a specific task, packaged as a unit. This unit can then be used in programs wherever that particular task have to be performed. A subroutine is often coded so that it can be started (called) several times and from several places during one execution of the program.

Once subroutines task is done, we can implemente it by call and return instruction.

**There are different types of subroutine instruction:**

**Unconditional Call instruction:** CALL address is the format for unconditional call instruction. After execution of this instruction program control is transferred to a sub-routine whose starting address is specified in the instruction. Value of PC (Program Counter) is transferred to the memory stack and value of SP (Stack Pointer) is decremented by 2.

**Conditional Call instruction**: In these instructions program control is transferred to subroutine and value of PC is pushed into stack only if condition is satisfied.

| INSTRUCTION | PARAMETER | COMMENT |
| --- | --- | --- |
| CC | 16-bit address | Call at address if cy (carry flag) = 1 |
| CNC | 16-bit address | Call at address if cy (carry flag) = 0 |
| CZ | 16-bit address | Call at address if ZF (zero flag) = 1 |
| CNZ | 16-bit address | Call at address if ZF (zero flag) = 0 |
| CPE | 16-bit address | Call at address if PF (parity flag) = 1 |
| CPO | 16-bit address | Call at address if PF (parity flag) = 0 |
| CN | 16-bit address | Call at address if SF (signed flag) = 1 |
| CP | 16-bit address | Call at address if SF (signed flag) = 0 |

**Unconditional Return instruction** –

RET is the instruction used to mark the end of sub-routine. It has no parameter. After execution of this instruction program control is transferred back to main program from where it had stopped. Value of PC (Program Counter) is retrieved from the memory stack and value of SP (Stack Pointer) is incremented by 2.

**Conditional Return instruction** –

By these instructions program control is transferred back to main program and value of PC is popped from stack only if condition is satisfied. There is no parameter for return instruction.

| INSTRUCTION | COMMENT |
| --- | --- |
| RC | Return from subroutine if cy (carry flag) = 1 |
| RNC | Return from subroutine if cy (carry flag) = 0 |
| RZ | Return from subroutine if ZF (zero flag) = 1 |
| RNZ | Return from subroutine if ZF (zero flag) = 0 |
| RPE | Return from subroutine if PF (parity flag) = 1 |
| RPO | Return from subroutine if PF (parity flag) = 0 |
| RN | Return from subroutine if SF (signed flag) = 1 |
| RP | Return from subroutine if SF (signed flag) = 0 |

**Advantages of Subroutine –**

1. Decomposing a complex programming task into simpler steps.
2. Reducing duplicate code within a program.
3. Enabling reuse of code across multiple programs.
4. Improving tractability or makes debugging of a program easy.

**Passing Parameters via the Stack**

The stack is usually the preferred way to pass arguments to a subroutine. Although this technique is a bit more involved once you get it right, it is bullet proof, and allows one to pass as many parameters to a subroutine as desired.

To pass parameters to a subroutine, the calling program pushes them on the stack in the reverse order so that the last parameter to pass is the first one pushed, and the first parameter to pass is the last one pushed. This way the first parameter is on top of the stack and the last one is at the bottom of the stack.

**The four major ways of passing parameters to and from a procedure are: -**

Passing parameters using registers

Passing parameters using memory

Passing parameters using pointers

Passing parameters using stack

**Passing parameters using registers-**

The data to be passed is stored in the registers and these registers are accessed in the procedure to pro. The disadvantage of using registers to pass parameters is that the number of registers limits the number of parameters you can pass.

E.g. An array of 100 elements can’t be passed to a procedure using registers.cess the data.

**Passing parameters using memory-**

In the cases where few parameters have to be passed to and from a procedure, registers are convenient. But, in cases when we need to pass a large number of parameters to procedure, we use memory. This memory may be a dedicated section of general memory or a part of it.

**Passing parameter using pointers-**

A parameter passing method which overcomes the disadvantage of using data item names (i.e. variable names) directly in a procedure is to use registers to pass the procedure pointers to the desired data.

**Passing parameters using stack-**

In order to pass the parameters using stack we push them on the stack before the call for the procedure in the main program. The instructions used in the procedure read these parameters from the stack. Whenever stack is used to pass parameters it is important to keep a track of what is pushed on the stack and what is popped off the stack in the main program.

**What is Stack?**

In computer science, a stack is an abstract data type that serves as a collection of elements, with two main principal operations:

**Push**, which adds an element to the collection, and

**Pop**, which removes the most recently added element that was not yet removed.

The order in which elements come off a stack gives rise to its alternative name, **LIFO** (last in, first out). Additionally, a peek operation may give access to the top without modifying the stack.

**Basic Architecture of a stack**

A typical stack is an area of computer memory with a fixed origin and a variable size. Initially the size of the stack is zero. A stack pointer, usually in the form of a hardware register, points to the most recently referenced location on the stack; when the stack has a size of zero, the stack pointer points to the origin of the stack.

The two operations applicable to all stacks are:

a **push** operation, in which a data item is placed at the location pointed to by the stack pointer, and the address in the stack pointer is adjusted by the size of the data item;

a **pop** or pull operation: a data item at the current location pointed to by the stack pointer is removed, and the stack pointer is adjusted by the size of the data item.

**Duplicate**: the top item is popped, and then pushed again (twice), so that an additional copy of the former top item is now on top, with the original below it.

**Peek**: the topmost item is inspected (or returned), but the stack pointer and stack size does not change (meaning the item remains on the stack). This is also called top operation in many articles.

**Swap** or **exchange**: the two topmost items on the stack exchange places.

**Rotate** (or **Roll**): the n topmost items are moved on the stack in a rotating fashion. For example, if n=3, items 1, 2, and 3 on the stack are moved to positions 2, 3, and 1 on the stack, respectively. Many variants of this operation are possible, with the most common being called left rotate and right rotate.

**Recursion**

In computer science, recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem. Such problems can generally be solved by **iteration**, but this needs to identify and index the smaller instances at programming time. Recursion solves such recursive problems by using functions that call themselves from within their own code. The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science

Most computer programming language support recursion by allowing a function to call itself from within its own code. A recursive function definition has one or more base cases, meaning input(s) for which the function produces a result trivially (without recurring), and one or more recursive cases, meaning input(s) for which the program recurs (calls itself).

**Types of Recursion**

**Single recursion and multiple recursion**

Recursion that only contains a single self-reference is known as single recursion, while recursion that contains multiple self-references is known as multiple recursion. Standard examples of single recursion include list traversal, such as in a linear search, or computing the factorial function, while standard examples of multiple recursion include tree traversal, such as in a depth-first search

**Indirect recursion**

Most basic examples of recursion, and most of the examples presented here, demonstrate direct recursion, in which a function calls itself. Indirect recursion occurs when a function is called not by itself but by another function that it called (either directly or indirectly).

**Anonymous recursion**

Recursion is usually done by explicitly calling a function by name. However, recursion can also be done via implicitly calling a function based on the current context, which is particularly useful for anonymous functions, and is known as anonymous recursion.

**Limit of recursion depth**: The depth of recursion refers to the number of levels of actiavation of a procedure which exist during the deepest call of the procedure.

Maybe 773

**Reentrancy**

In computing, a computer program or subroutine is called reentrant if multiple invocations can safely run concurrently on a single processor system, where a reentrant procedure can be interrupted in the middle of its execution and then safely be called again ("re-entered") before its previous invocations complete execution. The interruption could be caused by an internal action such as a jump or call, or by an external action such as an interrupt or signal, unlike recursion, where new invocations can only be caused by internal call.

**Relocation**:

A relocatable program is one that can be read into memory at any address and executed without modification. This mainly means using relative offsets for data accesses and jump instructions. If this is easy/possible is based on the type of architecture, the size of the address space and the size of the program.

**Call cycle:**

CALL: DR after m.ts. ОF contains the address of the transition

● DR → BR; Write the transition address in BR

● IP → DR; Prepare a return address for

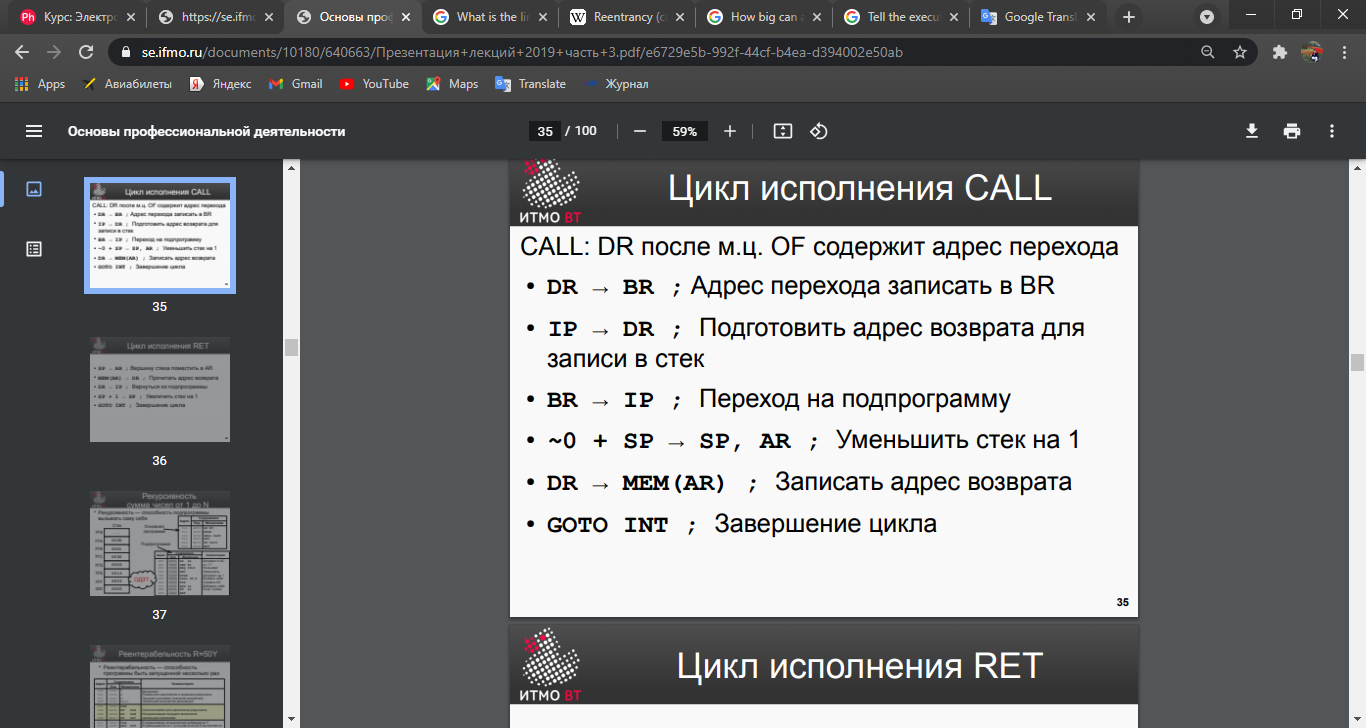
stack writes

● BR → IP; Go to subroutine

● ~ 0 + SP → SP, AR; Reduce stack by 1

● DR → MEM (AR); Record return address

● GOTO INT; End of cycle



**Cycle of RET**

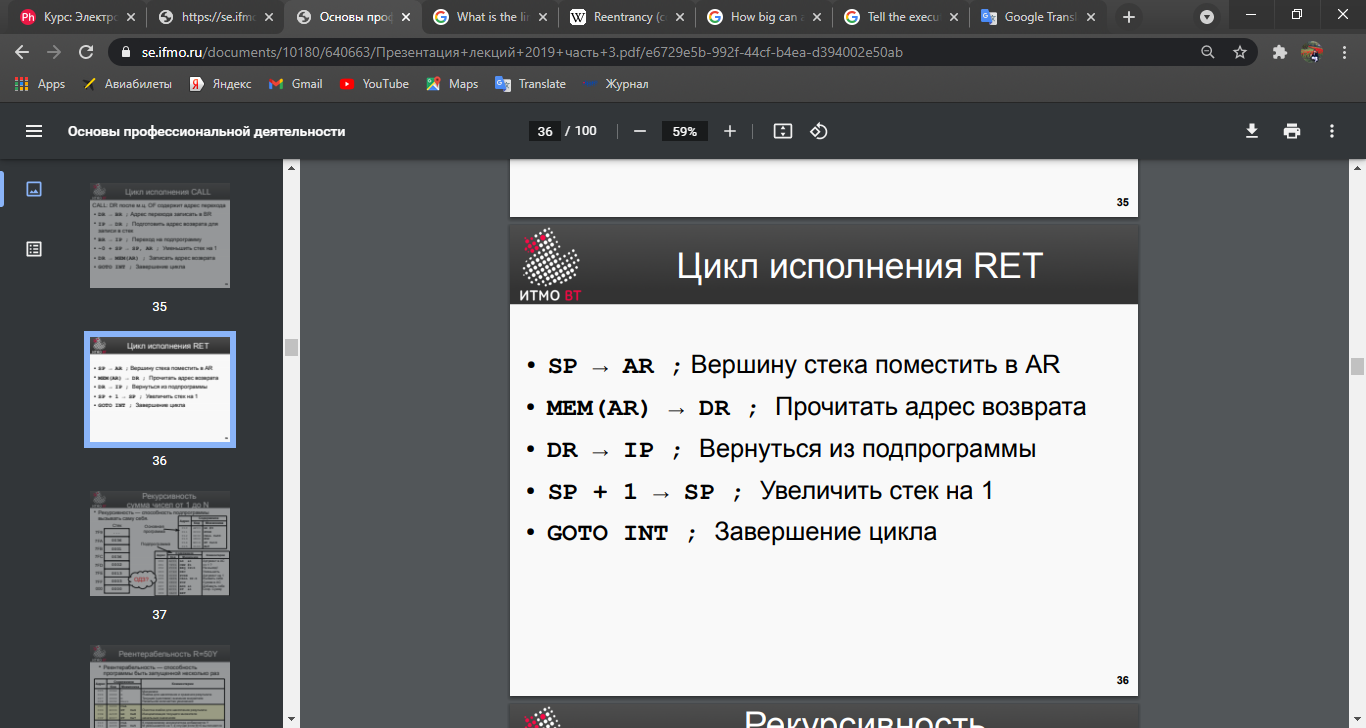
SP → AR; Place the top of the stack in AR

● MEM (AR) → DR; Read return address

● DR → IP; Return from subroutine

● SP + 1 → SP; Increase the stack by 1

● GOTO INT; End of cycle



**Recursion** is possible, in bvm it is limited by the number of free cells. If you call infinitely deep recursion, sooner or later it will start to spoil our program and erase it (since every time a subroutine is called, the return address is written to the stack, and the stack size is not programmatically limited)

the maximum depth of recursion depends on the location of the program in memory. the stack grows from bottom to top

The depth of recursion is limited only when the stack starts to erase our program and it breaks

**Насколько большую перемещаемую программу можно написать на БЭВМ?**

The size of the program is limited by the number of memory cells - 2048 cells, respectively, as many instructions we can write in memory.

**Combination of which two commands can work like RET**

I think RET can be replaced with JUMP and POP commands.

You will first need to correctly select the return address (we will have it written to the stack when the CALL command is executed), then use JUMP to return to this address, delete this address using POP and continue the program execution.

But this method may work incorrectly if some additional data is written to the stack.

You can, of course, not delete the return address (using POP) when using jump, but this will lead to the fact that garbage will accumulate on the stack (return addresses at which we have already returned, and we no longer need them) and this garbage can again, start erasing our program or interfere in any way.

**Machine cycles of different commands:**

**machine cycle :**

**1. Instruction fetch**

**2. Address fetch**

**3. Operand fetch**

**4. Execution**

**5. Interrupt**

**Instruction fetch**

1) instruction pointer register(IP) ->memory buffer register( BR), AR

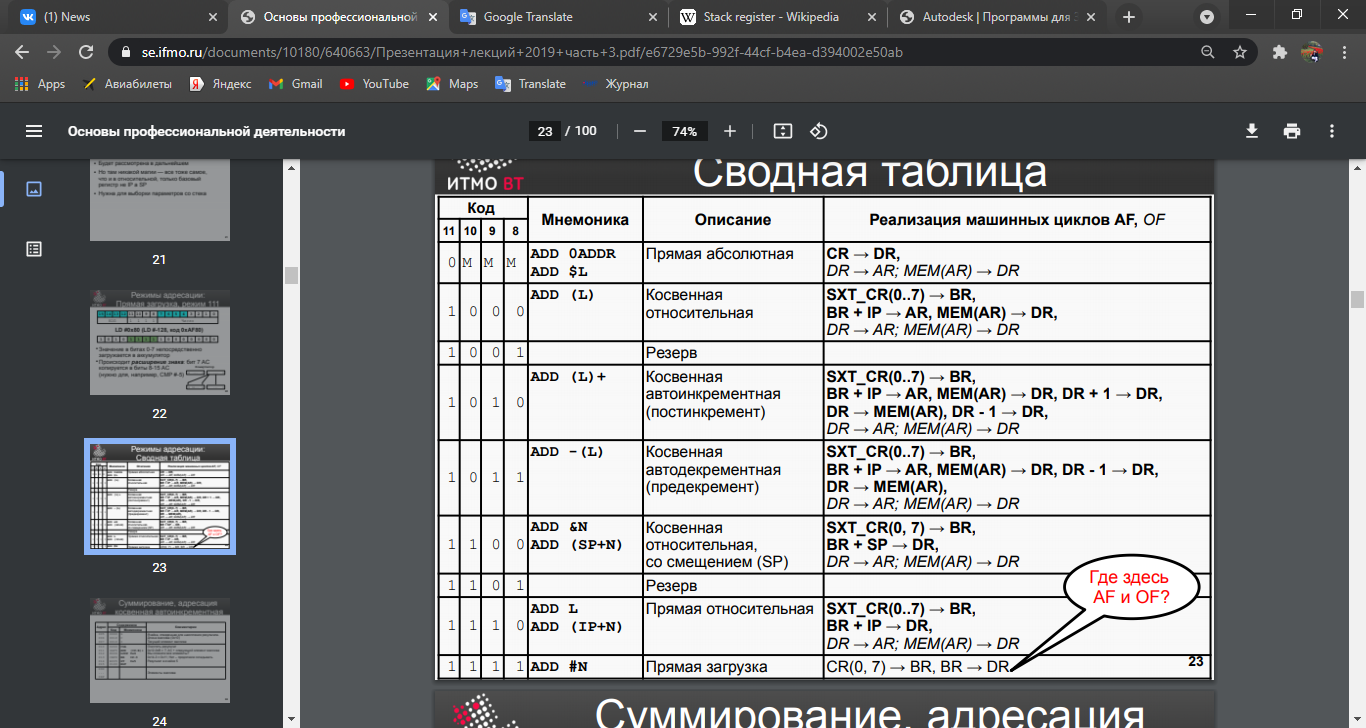
Instruction pointer IP content by ALU is written to BR and AR.

2) BR + 1 -> IP

The content of BR is increased by 1 and is written to IP, at the same time the content is read to DR.

3) DR -> CR

(Data register) DR content via ALU is written to (counter register)CR



**Operand fetch cycle:**

1) DR -> AR

DR (operand address from the instruction) are forwarded to AR(accumulator

**Execution fetch:**

**Push**:

AC → DR

~0 + SP → SP, AR

DR → MEM(AR)

if PS(W) = 0 then GOTO STOP ended here

if PS(IRQ) = 0 then GOTO INFETCH @ 01 goes to instruction fetch

**POP**:

DR → AC, N, Z, V

**POPF**

DR → PS

if CR(10) = 1 then GOTO AL11XX @ B5

if CR(9) = 1 then GOTO AL111X @ BB

if CR(8) = 1 then GOTO RESERVED @ E0

**DEC**

AC + ~0 → AC, N, Z, V, C

if PS(W) = 0 then GOTO STOP @ DE ended

if PS(IRQ) = 0 then GOTO INFETCH @ 01

**LD**

DR → AC, N, Z, V

GOTO INT @ C4

if PS(W) = 0 then GOTO STOP @ DE ended

if PS(IRQ) = 0 then GOTO INFETCH @ 01 Instruction fetch

**SUB**

AC + ~DR + 1 → AC, N, Z, V, C

**BPL F303**

if PS(N) = 0 then GOTO BR @ 5C

extend sign CR(0..7) → BR

BR + IP → IP

**CMP**

AC + ~DR + 1 → N, Z, V, C

**BMI F202**

if PS(N) = 1 then GOTO BR

extend sign CR(0..7) → BR

BR + IP → IP

**ASL 0500**

AC → DR

AC + DR → AC, N, Z, V, C

**RET**

DR → IP

SP + 1 → SP

**IRET**

DR → PS

SP + 1 → SP, AR

MEM (AR) → DR

DR → IP

SP + 1 → SP