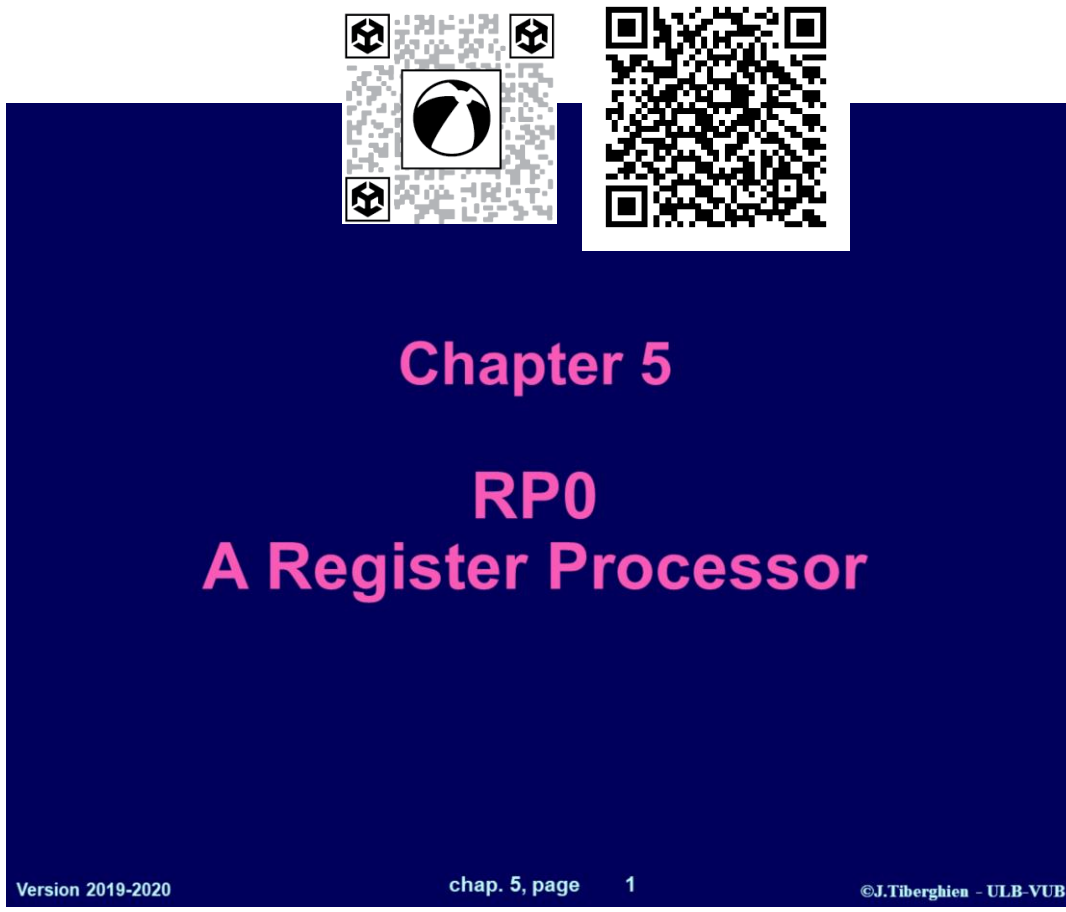
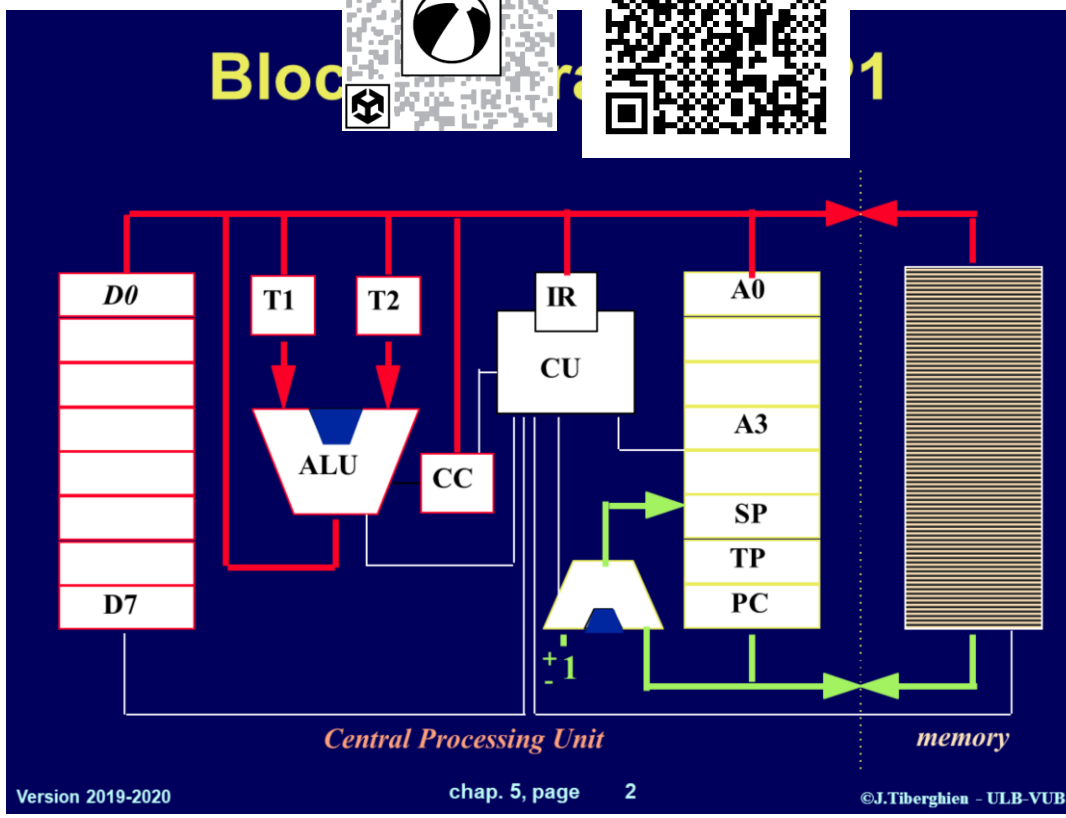
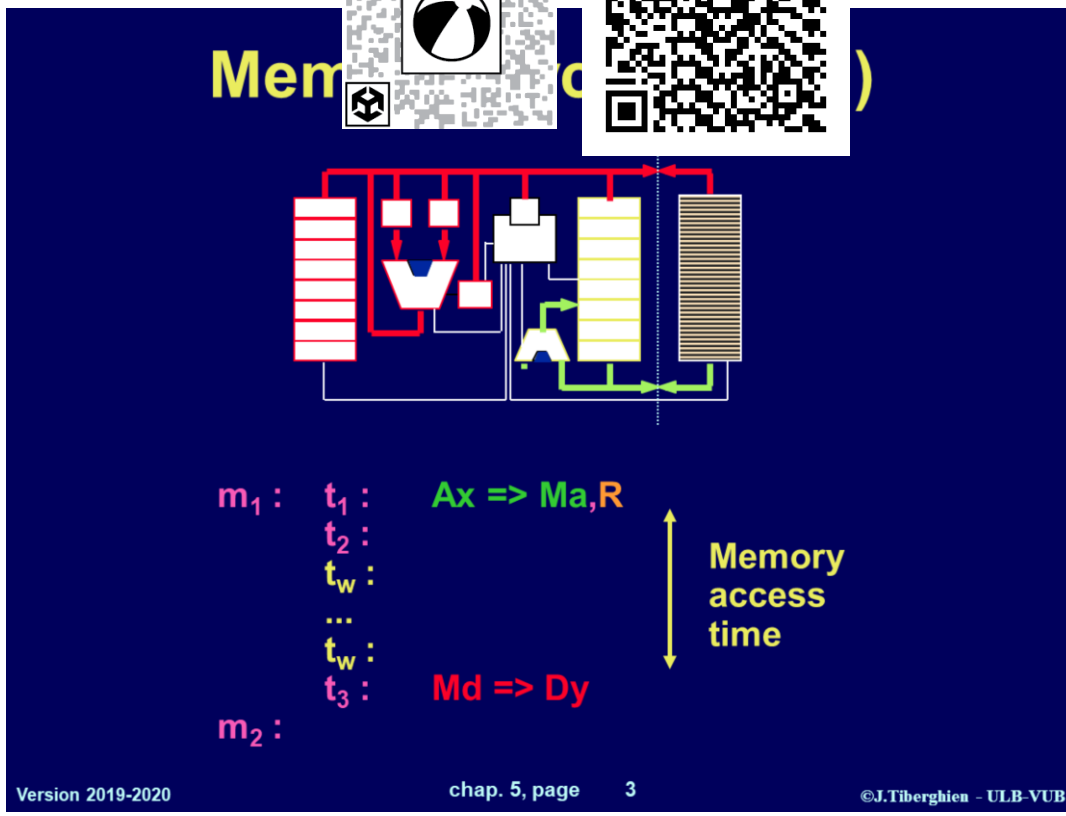


In the previous chapter we could see that a stack architecture is excellent to manage activation blocks and evaluate expressions but we also saw that at certain times, such as during the execution a **for** loop, the stack organization can be very hard. In this chapter a register architecture will be described. This architecture can emulate a stack architecture where this is useful, but also avoids heaviness caused by the rigidity a stack.
It would be perfectly possible to describe this processor as was done in previous chapter (this is even be an excellent exercise to check understanding of these two chapters) but another method closer to the physical reality of the processor will be used here. This method of describing all computer devices through which data can flow and analyse these transits in time. The descriptions used in the previous chapter and in this one are complementary, the first being more abstract and essentially intended to understand the logic of the instruction set, regardless of the speed of execution, while the second, more concrete, consider not only the logic of the instructions but also the temporal sequence of their execution.



The processor RP1 does not exist commercially, but its architecture, reduced to the minimum necessary to illustrate architectural concepts, inspired by that of the Motorola 68000 family.
On the slide, from left to right, we can see
- A set of eight registers, called D0 to D7, intended to contain data
- An arithmetic unit (ALU) with, at its inputs, two temporary registers T1 and T2.
- A CC register for holding the properties of arithmetic results
- A control unit (CU) with its instructions for register (IR)
- An arithmetic unit to increase or decrease addresses
- A set of 8 registers to keep addresses, called A0 to A7; In the last three registers A have specific functions. Thus, A7, also called PC is the program counter, which keeps the address of the next instruction to execute.
- The central computer memory.

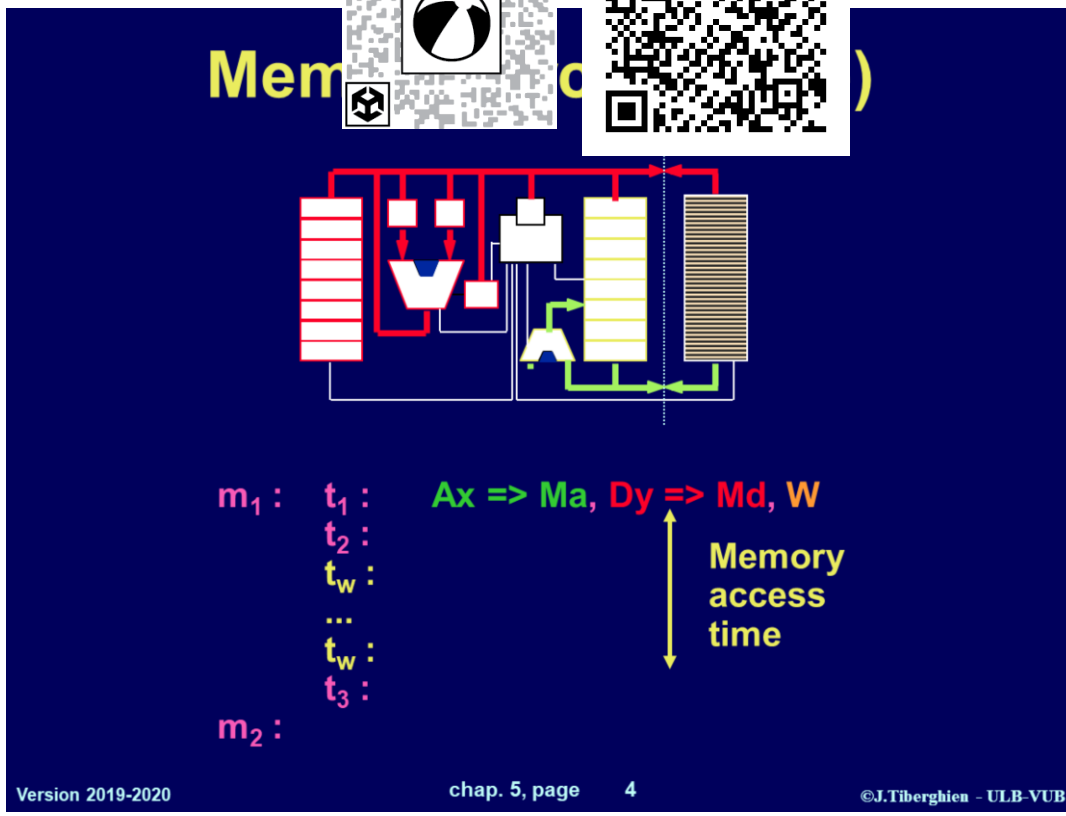
All these components are interconnected by means of two 'bus'. These are the wire sets to transport the content of a register to another.
Both bus present in RP1 are the data bus, shown in red at the top of the slide and the address bus, shown in green at the bottom of the slide.
Besides the busses a number of wires that go from the control unit to the different computer parts. These wires allow the control unit to transmit its commands.



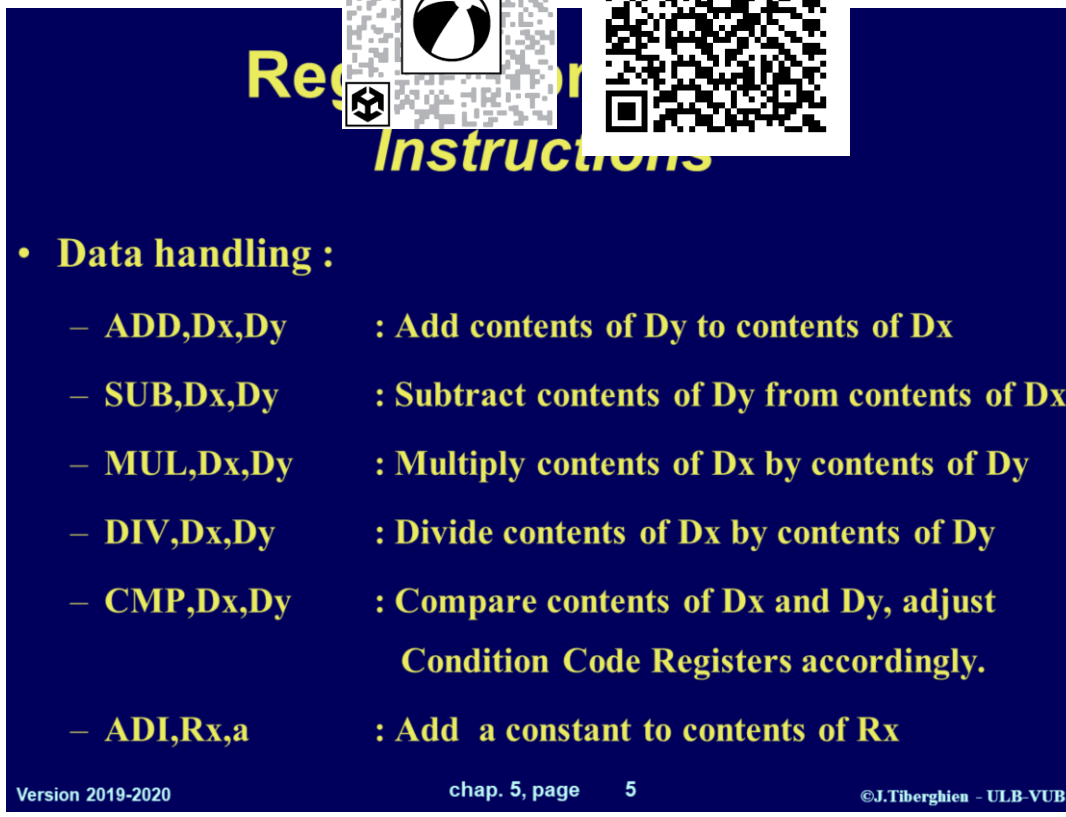
To describe a transfer between memory and CPU, we must first consider the clock that determines the timing of all computer operations. This clock is part of the control unit and defines fixed time intervals called "clock cycle". In contemporary computers, the duration of a clock time is of the order of nanoseconds. A write or read operation in the main memory takes a few clock cycles. The totality of the required clock cycles is called a "memory cycle."

A typical read cycle takes place as follows:

- At time t1, the address of the data to be read is sent from one of the A registers to the memory via the address bus and a read command is sent by the control unit to the memory.
- At time t2, the control unit asks the memory the status of the read operation. If ready, proceed to t3. Otherwise, the control unit inserts a waiting time tw before the control unit asks again the memory for a status.
- At time t3 the read operation is completed by the transfer of the read value via the data bus to a central processing register.



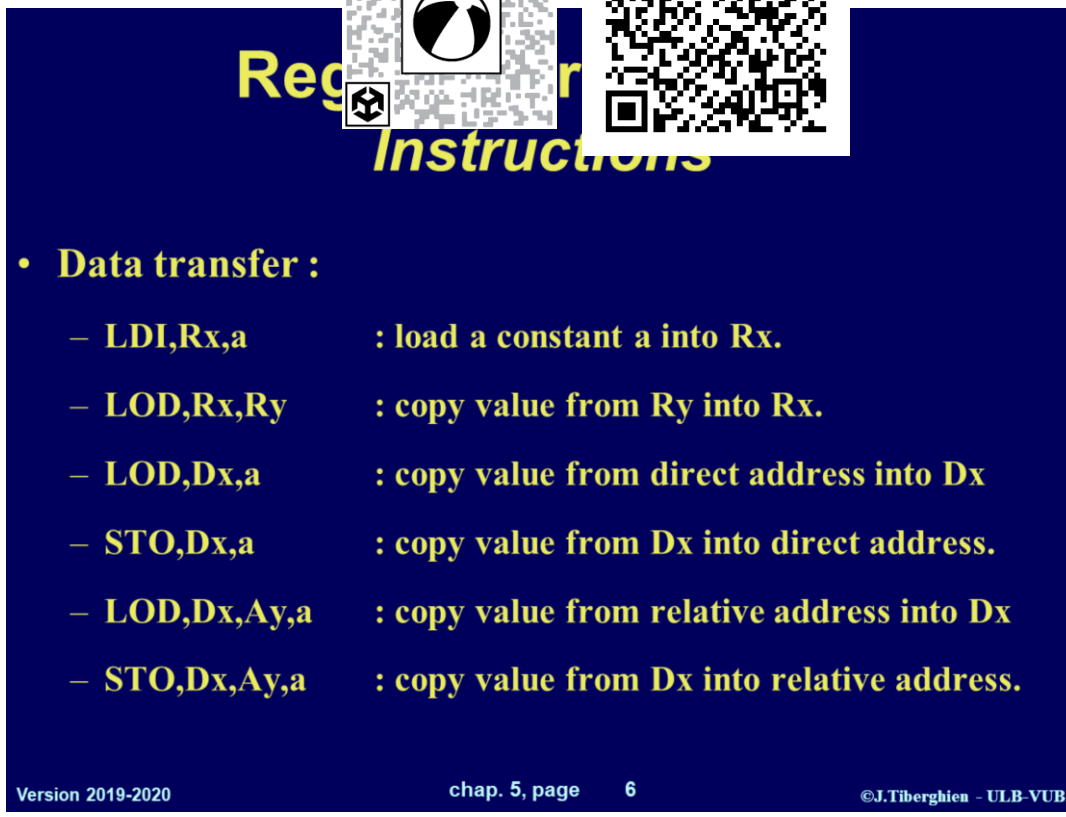
A write cycle begins with a time t1 during which the address is placed on the bus addresses, the data on the data bus and a write command is given by the control unit. The rest of the memory cycle is used partly by the memory to execute the write operation and partly by the control unit checking the memory status at each clock cycle. A write cycle also ends with a clock cycle t3, during which nothing here described happens.
If we consider normal memories that have an access time of about 50ns we understand that for each read operation many tw waiting times will be inserted. This finding helps explain why the clock speed is a poor indicator of the performance of a computer, since an increase in the clock speed may have the primary effect of increasing the number of times tw to insert into each memory cycle.



The RP1 data processing instructions operate on data in the processor registers. For most instructions this is exclusively for Dx data registers, but for some instructions, registers Ax addresses are also allowed. This latitude is indicated by the use of Rx notation meaning Dx or Ax. The RP1 data processing instructions have two fields to reference operands, the result of the operation being systematically relocated to the place where was the first operand.

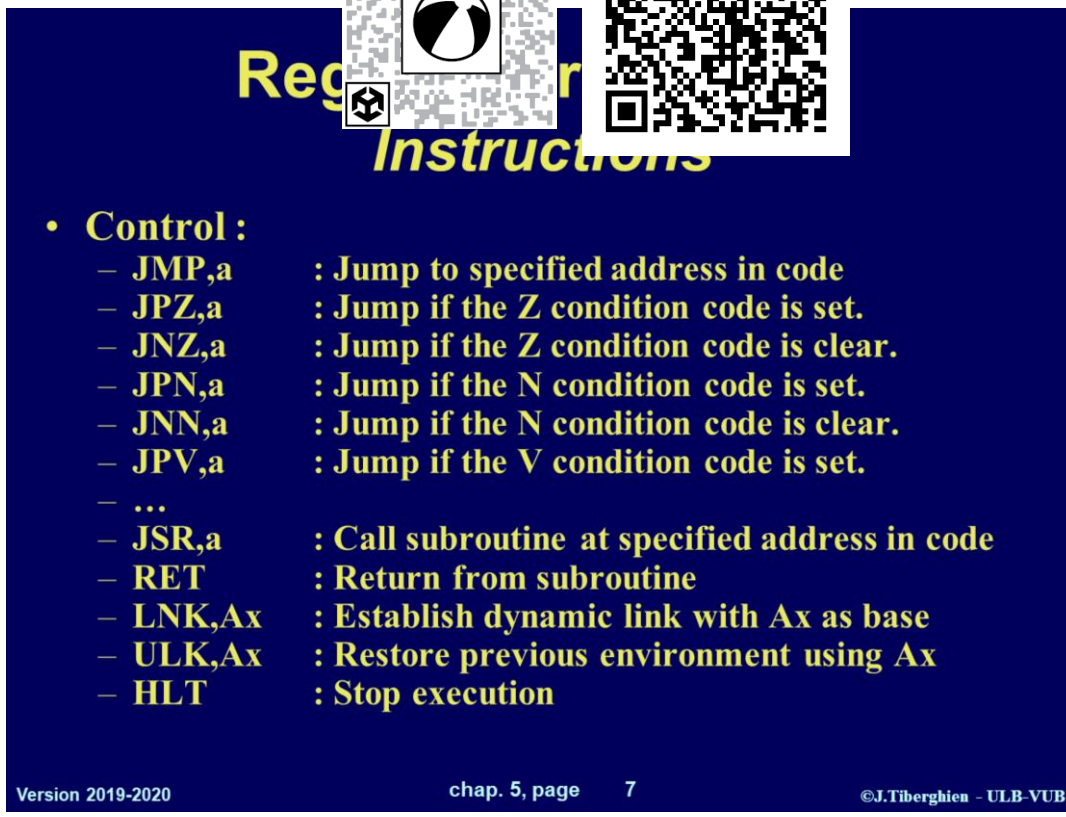
Besides the four arithmetic operations, RP1 has a CMP compare instruction which is in fact identical to the subtraction operation but does not replace the first operand with the result. The properties of the result of the arithmetic operation is noted in the CC register. This is so for all other information processing instructions.

Finally, there is an instruction, ADI that can add to the contents of any data or address register the value of a constant included in place of the reference of the second operand in the instruction.

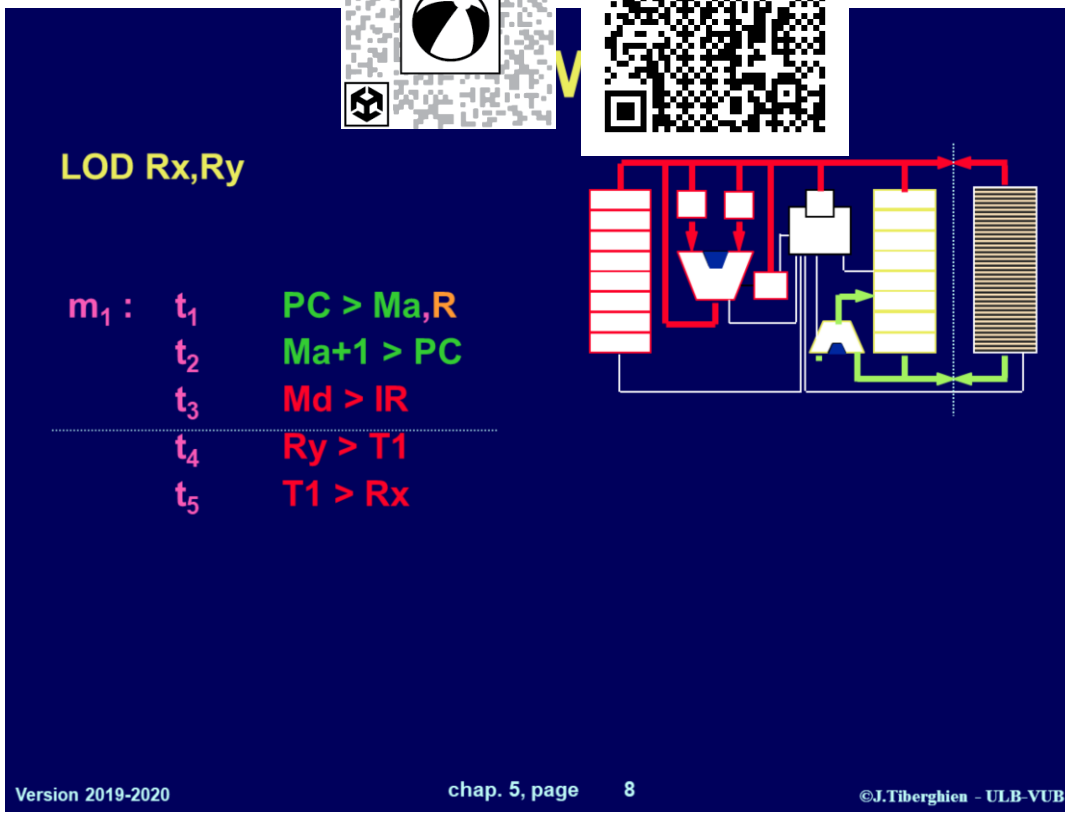


In a register machine, one needs transfer instructions to:
- load constants into registers (LDI, Rx, a)
- copy the contents of a register to another (LOD Rx, Ry)
- copy the contents of a word from memory into a register (LOD Dx, ...)
- copy the contents of a register in a word of memory (STO Dx, ...)

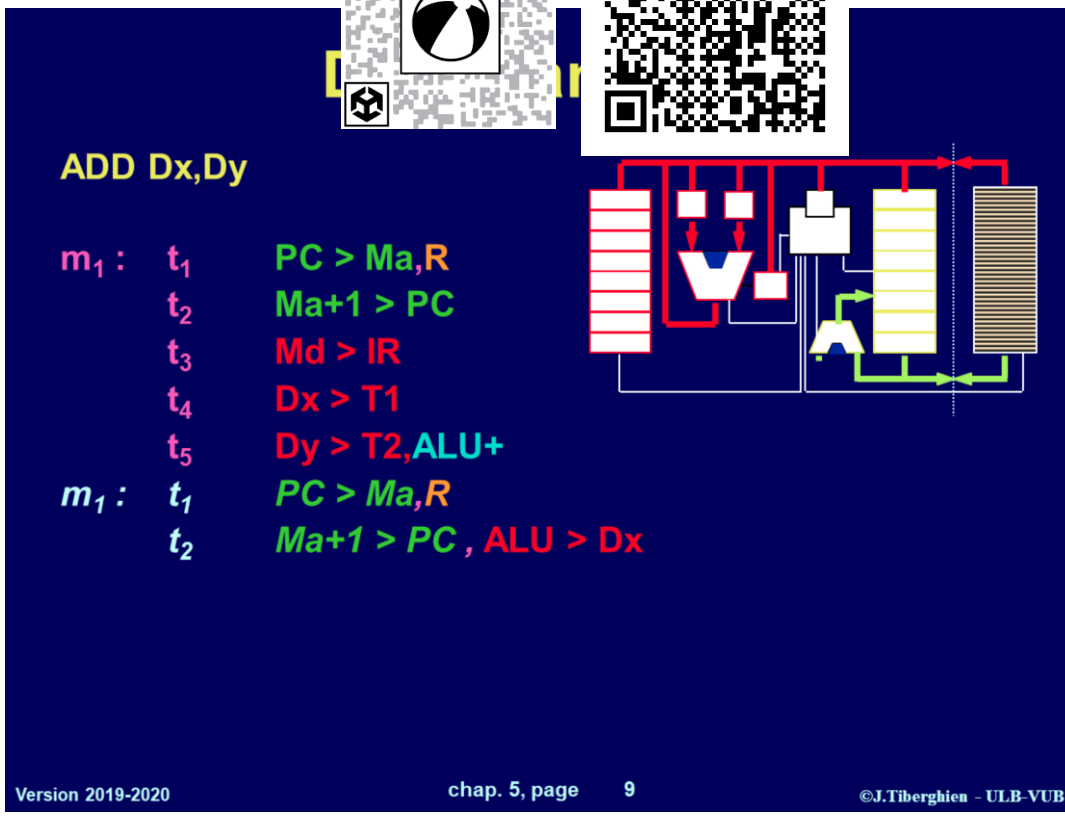
The architecture of RP1 offers for the last two transfer instruction categories two alternative transfer instructions to reference a memory address, the direct addressing and relative addressing to the contents of a address register.



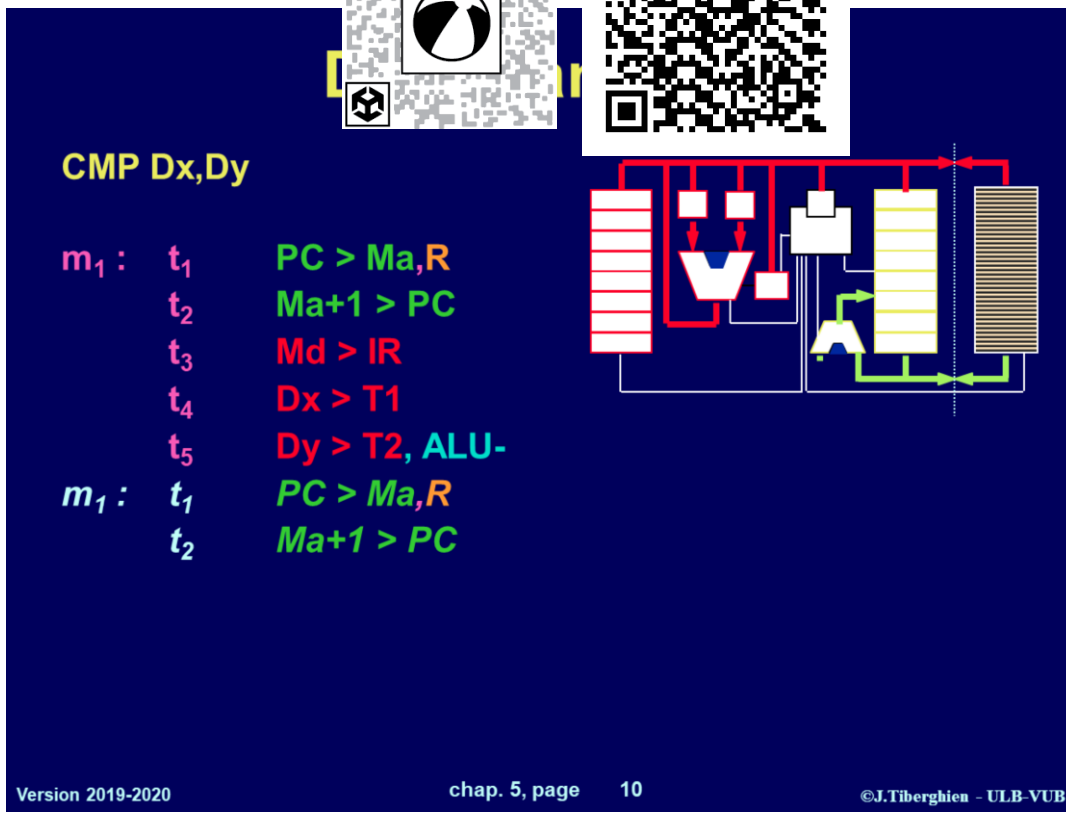
Besides the unconditional jump (JMP) one can find among the control instructions a series of instructions for conditional jumps. As explained in Chapter 3.2, it's the status of specific bits of the CC register that determines if the jump is to be performed or not. Only a few RP1 instructions for conditional jumps are shown here. They are those that depend on the bit Z (zero), N (negative) and V (overflow during the calculation). Among the instructions for managing the sub routines there is obviously the call subroutine (JSR) and the return from subroutine (RET) but there are also instructions for managing dynamic links (LNK and ULK).
Finally, there is, similar for the stack processor described in the previous chapter, a HLT instruction useful if one wishes to write a simulator for this processor.



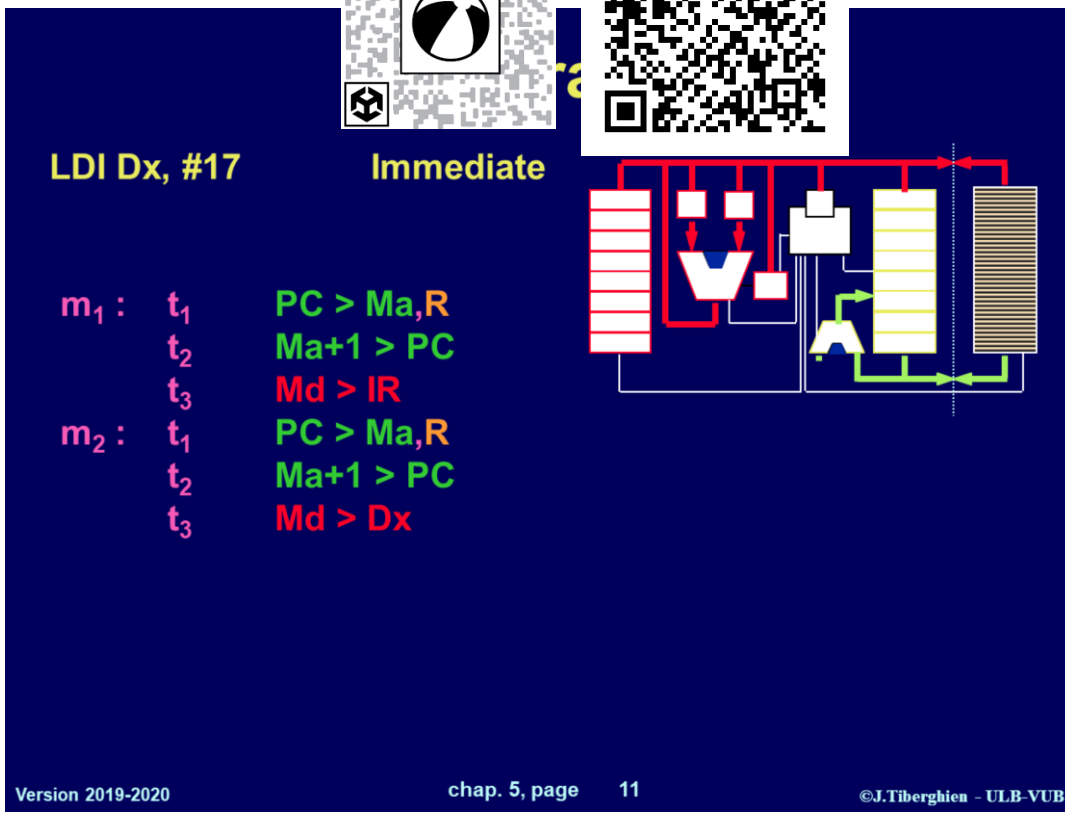
The first instruction that will be analysed here is the instruction LOD Rx, Ry which copies the contents of the register Ry to the register Rx.
It requires only a single memory cycle. The first time the contents of the program counter PC is on the bus addresses and a read command is given.
The second time, the contents of the program counter that was still on the address bus is increased by one with the little arithmetic unit associated with the address registers, and rewritten in PC. Thus it is already preparing the start of the execution of the next instruction.
At the time t3, possibly after some time of tw unrepresented waiting, the instruction is transferred from the memory in the IR register to be decoded and executed. It is not until that the controller 'knows' what the instruction is that the instruction is executed. To copy the contents of Ry in Rx not additional memory cycle is required, just one clock cycle is needed to copy the contents of the register Ry in the temporary register T1 and a second clock cycle to copy the contents of this temporary register into Rx. Both transfers between T1 and another register are obviously via the data bus.
The copy of the content of Ry into Rx can not be done without intermediate step as the 8 D registers and 8 A registers are organized as addressable memories in which, each time, only one register is accessiible.



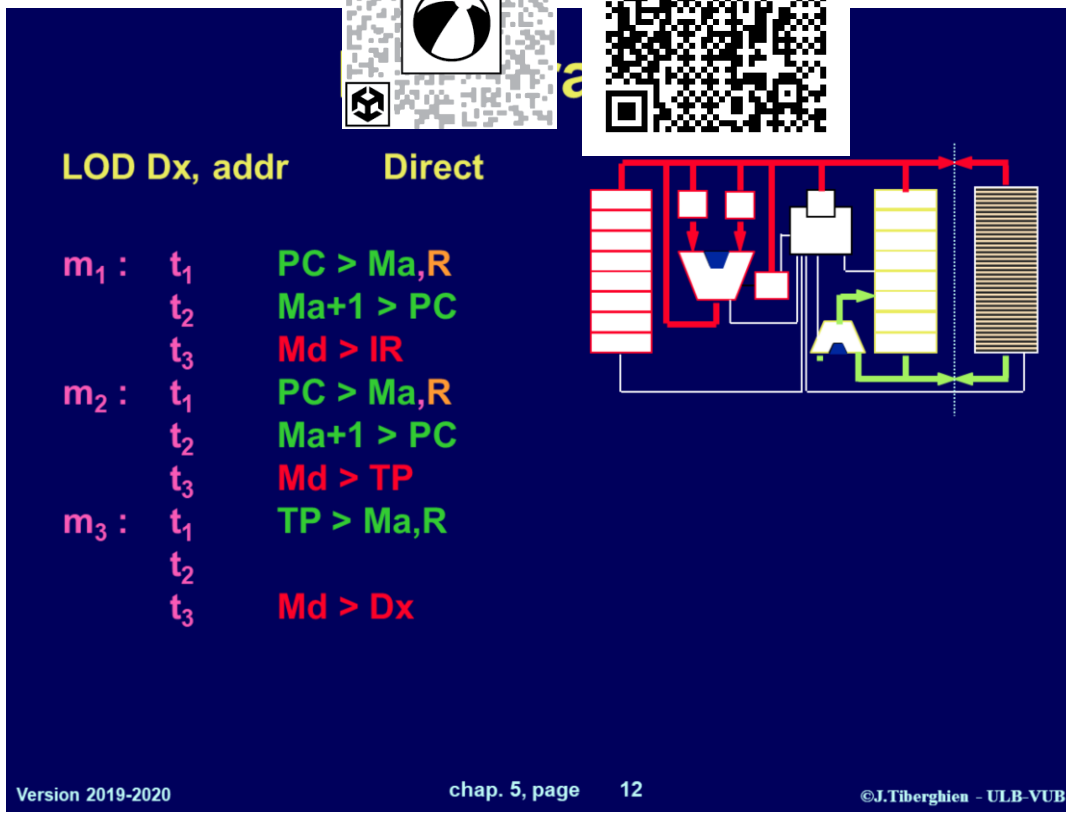
Another instruction that requires only one memory cycle and which introduces an important architectural concept is the ADD Dx, Dy instruction that serves to add to the contents of Dy to Dx.
The first three clock cycles are obviously identical to those described for the LOD instruction Rx, Ry, since the nature of the instruction to be executed is known only from the end of t3.
At t4 the content of Dx is transferred to the temporary register T1, then, at t5, the Dy content is transferred to the T2 register and the command to add is given to the arithmetic unit.
Normally there should be a t6 to transfer the result of the addition in Dx. Instead, it immediately starts executing the next instruction and takes advantage of the t2 of the instruction to transfer the result of the addition. Indeed, during the cycles t1 and t2 of the first memory cycle instruction execution, the data bus is not used and can be used to terminate execution of the previous instruction.
This temporal superposition of the end of the execution of one instruction and the beginning of the next is called "pipelining" and is a technique that is used extensively in all modern processors to increase performance.



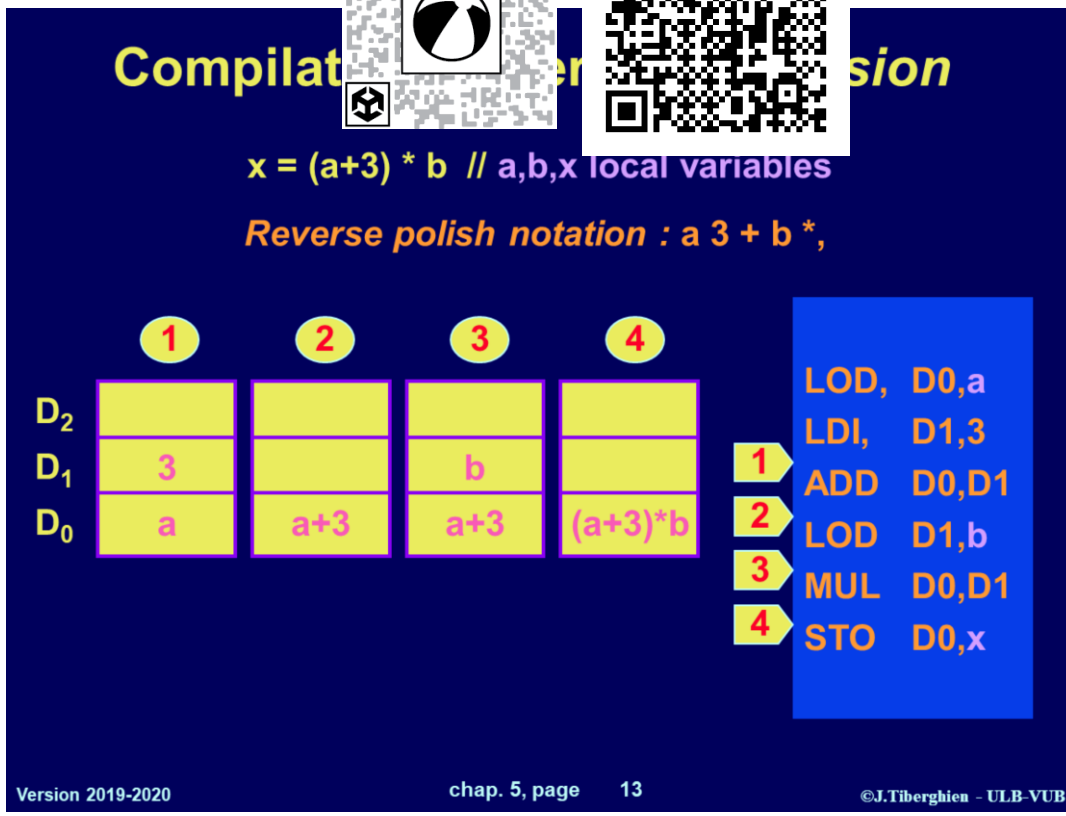
The instruction CMP Dx, Dy compares the values in the Dx and Dy registers. It requires 5 clock cycles and is therefore completely identical to what was described for the ADD Dx, Dy instruction, except that the command provided to the arithmetic unit is a subtraction instead of an addition.
The main difference is seen during the execution of the next instruction, since at t2 the results available at the output of the arithmetic unit is not copied into a register.
The only result of the CMP Dx, Dy instruction remains in the CC register to which, at each arithmetic operation the properties of the result are reported.



The LDI Dx c instruction allows to load the constant c into the Dx register. It is exists of two words, the first is the actual instruction, the second is the constant c. It therefore requires two memory cycles.
The first is limited to three clock cycles and ends when the first word of the instruction loaded in the IR register.
During the second cycle, the incremented content of PC once again put on the address bus and, once again incremented. At t3 of the second cycle, the constant c is placed on the data bus by the memory and copied into the Dx register.



The LDI Dx, a instruction allows to load the value stored at memory address **a** in the Dx register. It is a long, two words instruction, the first is for the actual instruction, the second is the address **a**.
It requires three memory cycles, two to read the instruction and to read the value to be transferred.
The first is limited to three clock cycles and ends when the first word of the instruction in the IR register.
In the second cycle, the content of PC is incremented, put on the address bus and, once again incremented. At the t3 of the second cycle, the content of memory address **a** is put on the data bus by the memory and copied to the A6 register also called TP.
In the third cycle, the TP content is placed on the address bus and a read command is given to thememory. At the t3 of the third cycle, the value in the memory at the address **a** has become available on the data bus and is copied to the Dx register.



Les explications données à propos des instructions arithmétiques et de transfert devraient suffire pour permettre de comprendre comment une expression peut être évaluée dans RP1.

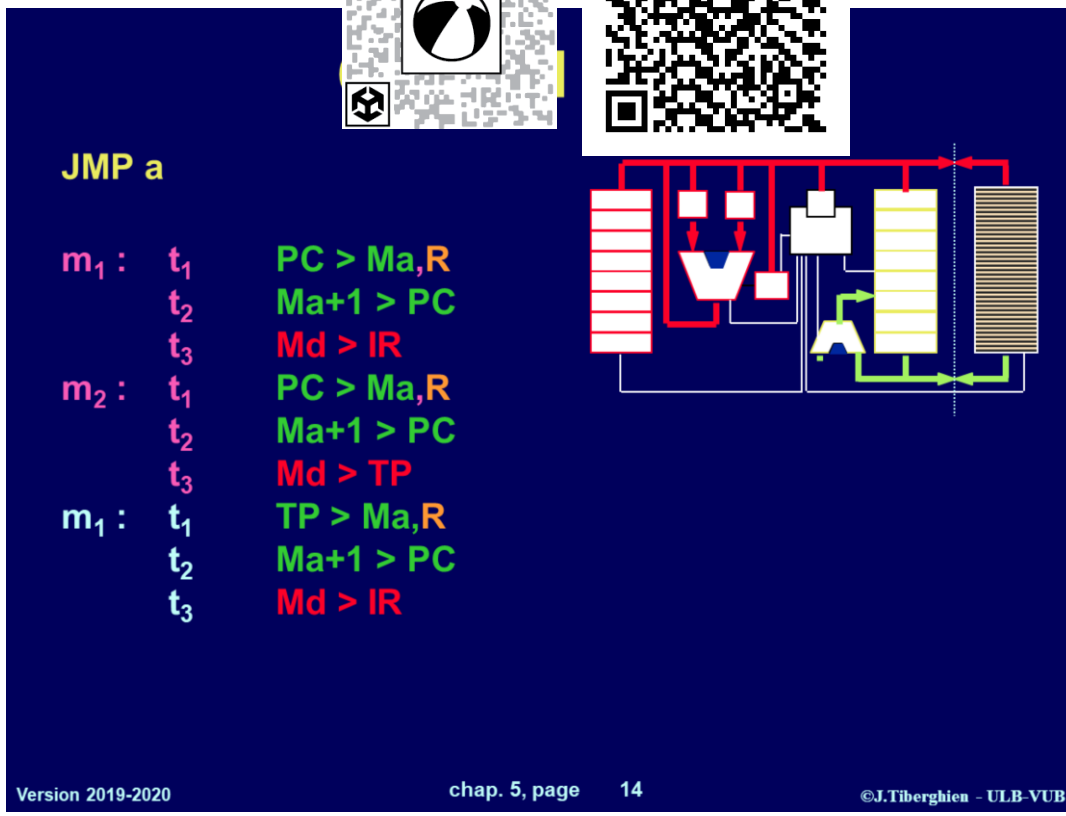
Les registres D vont être utilisés pour former une pile sur laquelle on peut évaluer une expression comme cela a été fait sur la pile de l'ordinateur décrit au chapitre précédant.

L'expression est d'abord réécrite en notation polonaise inversée. Le compilateur va ensuite lire cette expression et traduire les opérandes rencontres par des instructions LOD ou LDI et les opérateurs par les instructions arithmétiques appropriées.

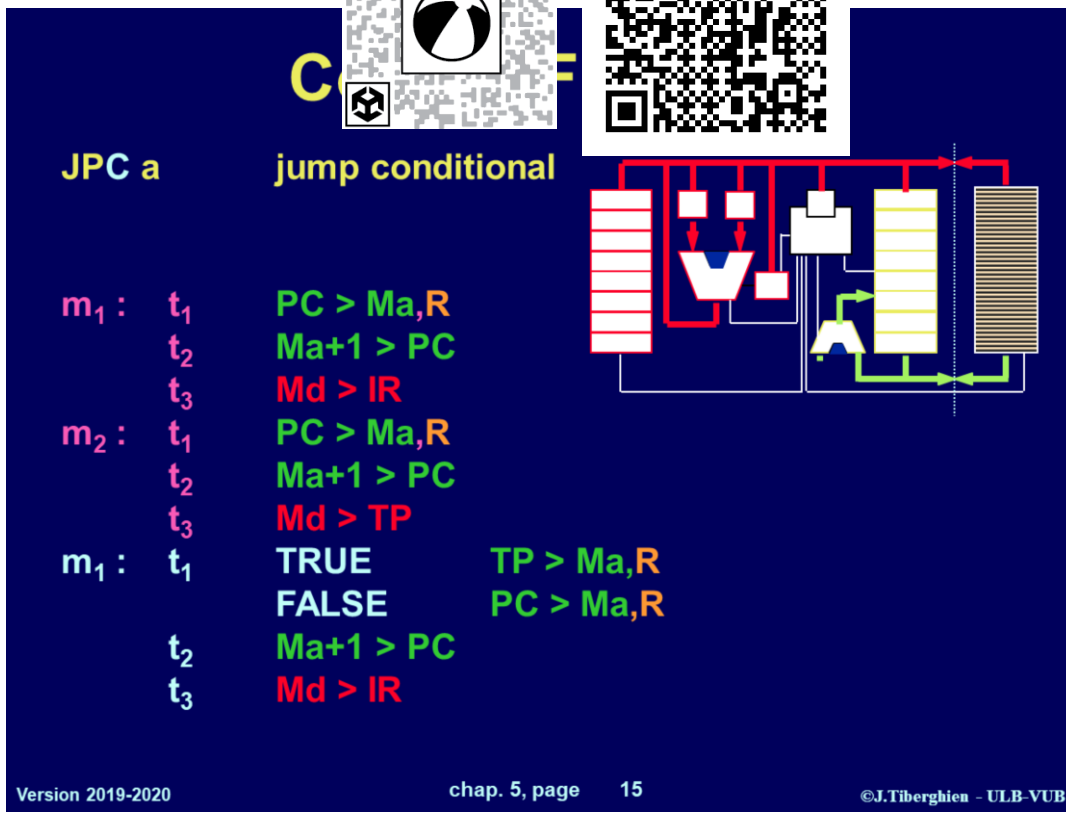
Ainsi, pour l'expression polonaise inversée **`a 3 + b *`** le programme commencera par placer la valeur de **`a`** dans le registre D0 et la constante **3** dans le registre D1. Ensuite le contenu de D1 sera additionné au contenu de D0. Après cela, la valeur de **`b`** sera placée dans D1 et le contenu de D0 multiplié par le contenu de D1. Il ne restera ensuite plus qu'à transférer le contenu de D0 dans la variable **`x`** en mémoire.

The explanations about arithmetic instructions and transfer should be sufficient for understanding how an expression can be evaluated in RP1.
The D registers will be used to form a stack on which to evaluate an expression as has been done on the stack of the computer described in the previous chapter.
The term was first rewritten in reverse Polish notation. The compiler will then read and translate this expression operands meetings with LOD or LDI instructions and operators through appropriate arithmetic instructions.
Thus for Reverse Polish expression 3 + b * the program will start by placing the value in a register D0 and the constant 3 in the register D1. Then the contents of D1 will be added

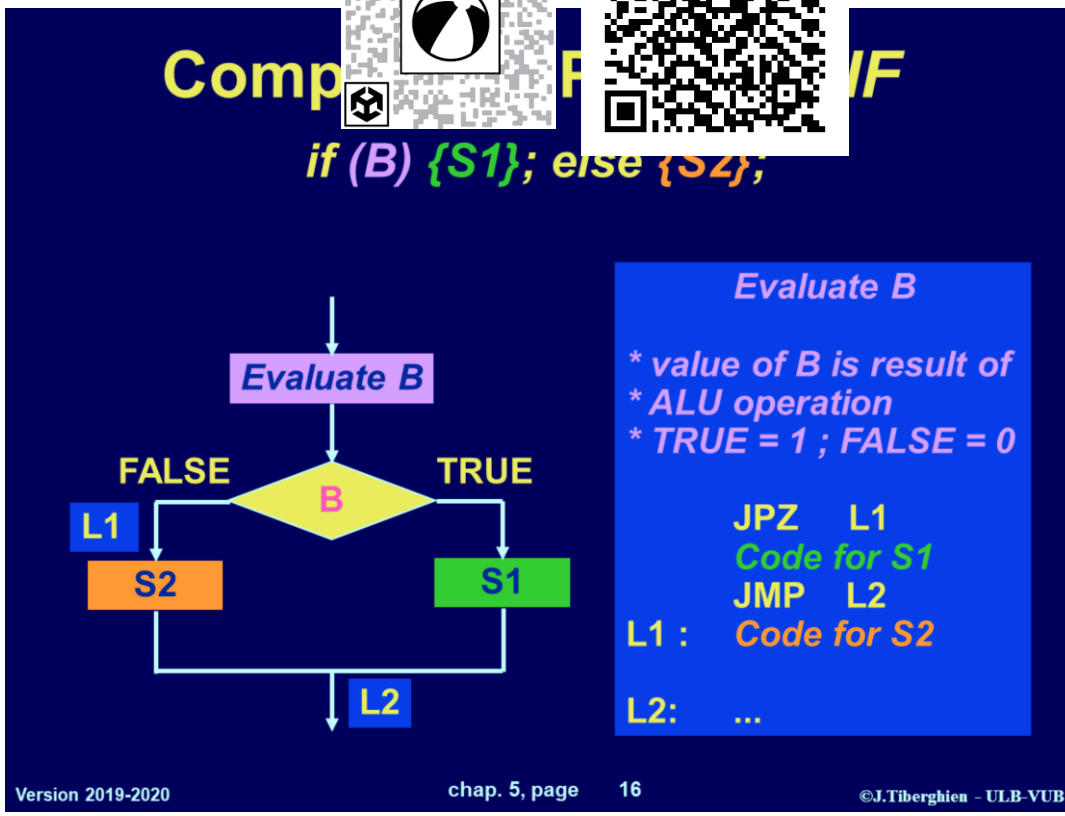
to the contents of D0. After th w ontent D1 and D0 multiplied by the contents of l er contents of D0 in the variable x in memory.



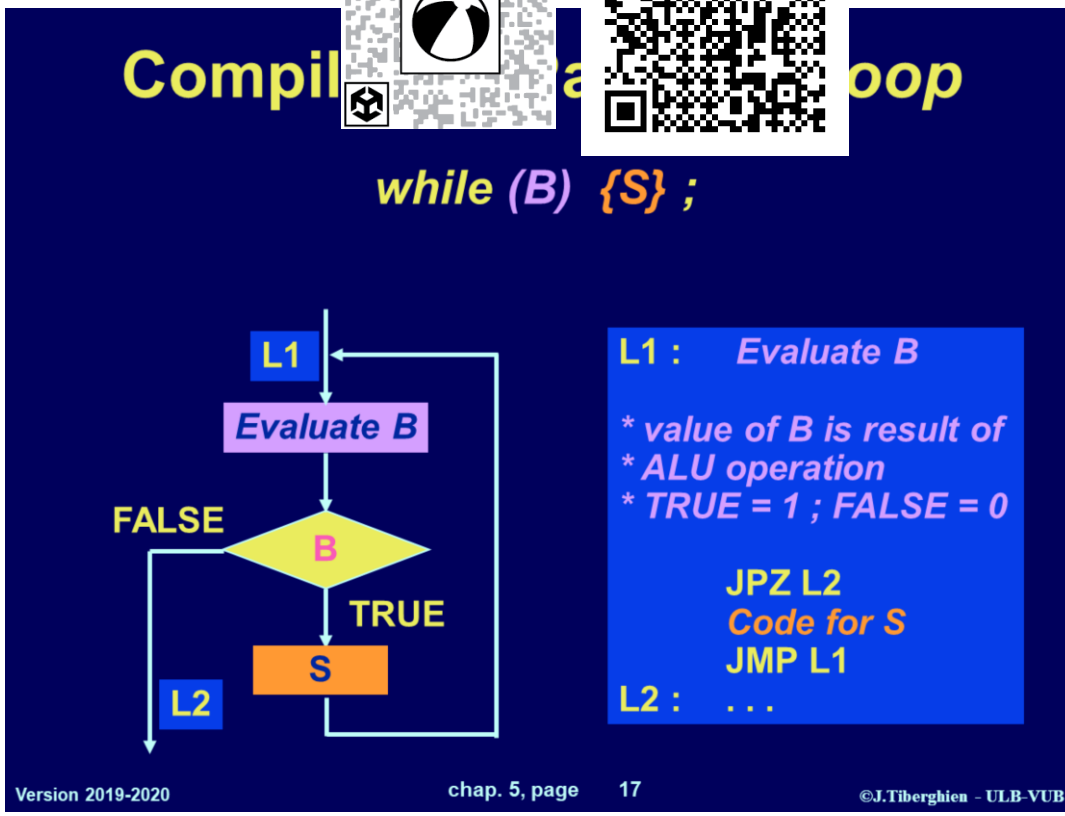
The JMP **a** instruction allows an unconditional jump to the address **a**. This instruction has a length of two words, the first being the actual instruction, the second the address to which it is necessary to jump to. During the second memory cycle of the execution, this address is transferred to the TP register.
Then begins execution of the next instruction, but instead of placing the PC content on the address bus during the first clock cycle the content of TP that is placed on the bus. At t2, as for any other execution, the small additional arithmetic unit adds 1 to the contents of the bus addresses and places the result in the PC registry. In this way, program execution continues well from the address that was contained in the statement of unconditional jump.



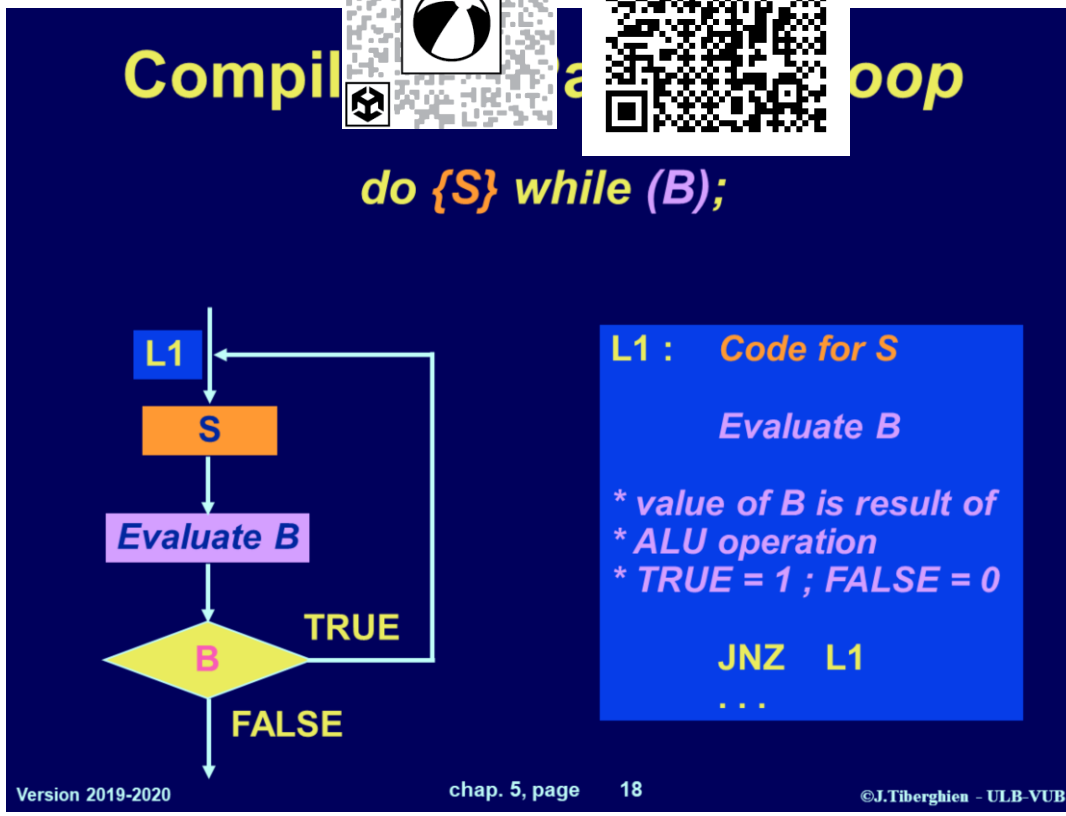
The execution of a conditional branch instruction is quite similar to that of an unconditional jump instruction, one major difference, which is at the beginning of the execution of the next instruction.
At the end of the second memory cycle, the TP register contains the address of the instruction that will execute if the condition of the jump is satisfied while the PC register contains the address of the instruction following the conditional jump instruction, that it will run if it is not satisfied with the condition of the jump. The control unit therefore decides at that moment, according to the CC register status if it's required to place the contents of TP or PC on the address bus.



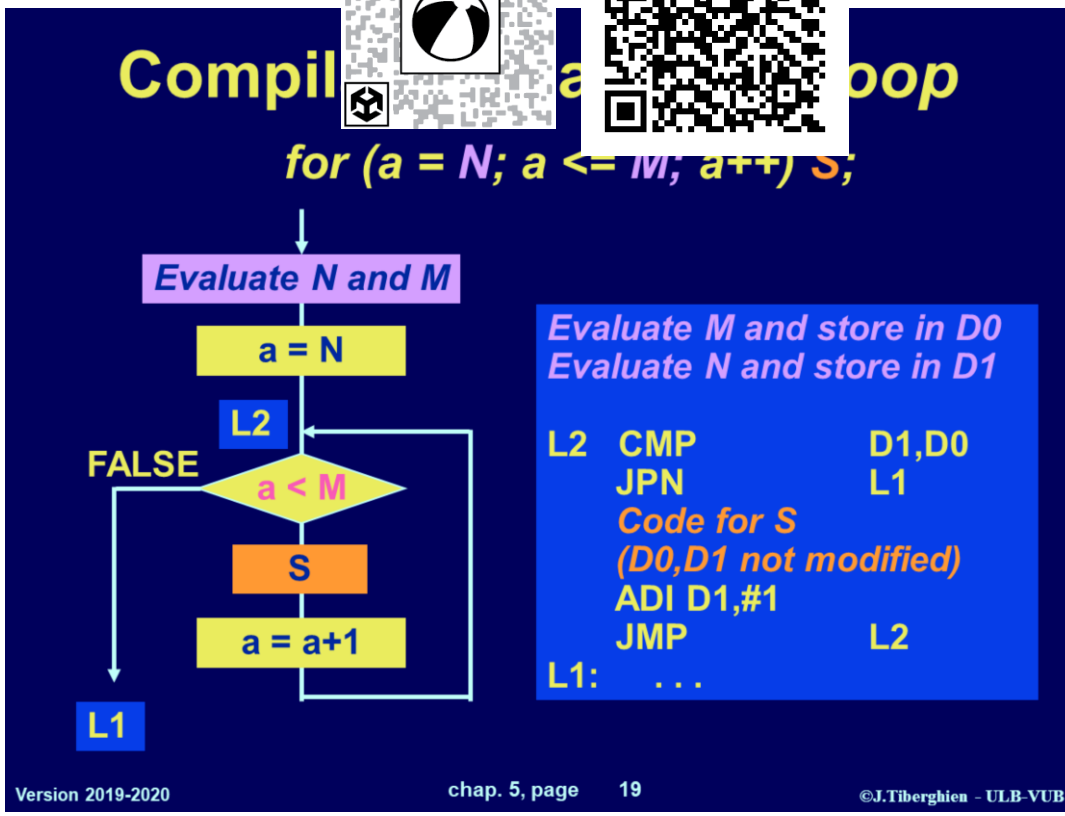
Using the conditional and unconditional jump instructions one can implement conventional control instructions of high-level languages. The slide above shows how an **if** statement is translated into assembler RP1.
First the Boolean expression B is evaluated so that the evaluation ends with a calculation that results in a 0 result if the expression is **false** and a 1 result if the expression is **true**. After evaluating B the instruction **JPZ L1** skips to the label L1 if the value of B was false. If, on the other hand it were **true**, the code for S1 is executed. After this code unconditional jump to the label L2 is done which is the continues the rest of the program. The code for S2 is placed between the labels L1 and L2.



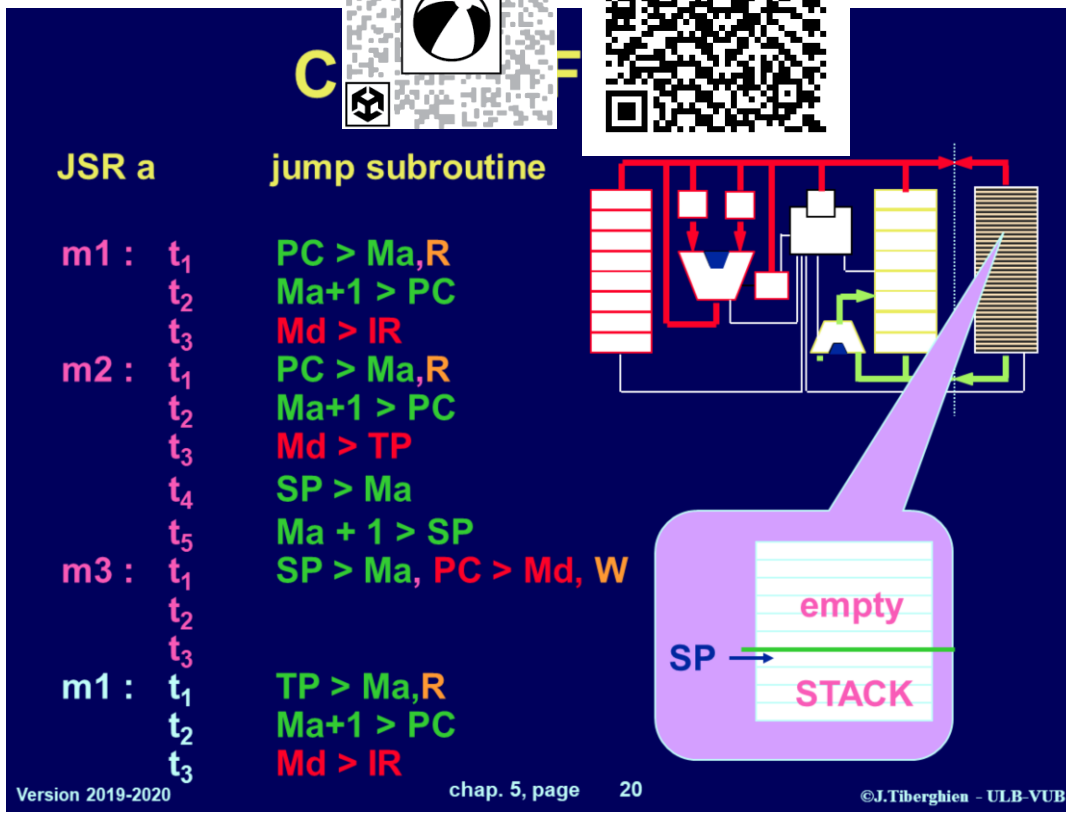
The translation in assembler of the **while … do** loop is easy. first the expression B is evaluated as in the above example. A JPZ L2 instruction skips to the rest of the program if the value of B was **false**. If, on the other hand, it was **true** the code for S is executed and at the end of it, an unconditional jump JMP L1 causes a reassessment B.



The code resulting from the translation of a **do..while** statement shown above.
On can see that it starts with the code that corresponds to S, followed by the code corresponding to the evaluation of the Boolean expression B. Following this assessment a JNZ instruction skips to the beginning of the S execution if the value B was TRUE, otherwise continue with the rest of the program.



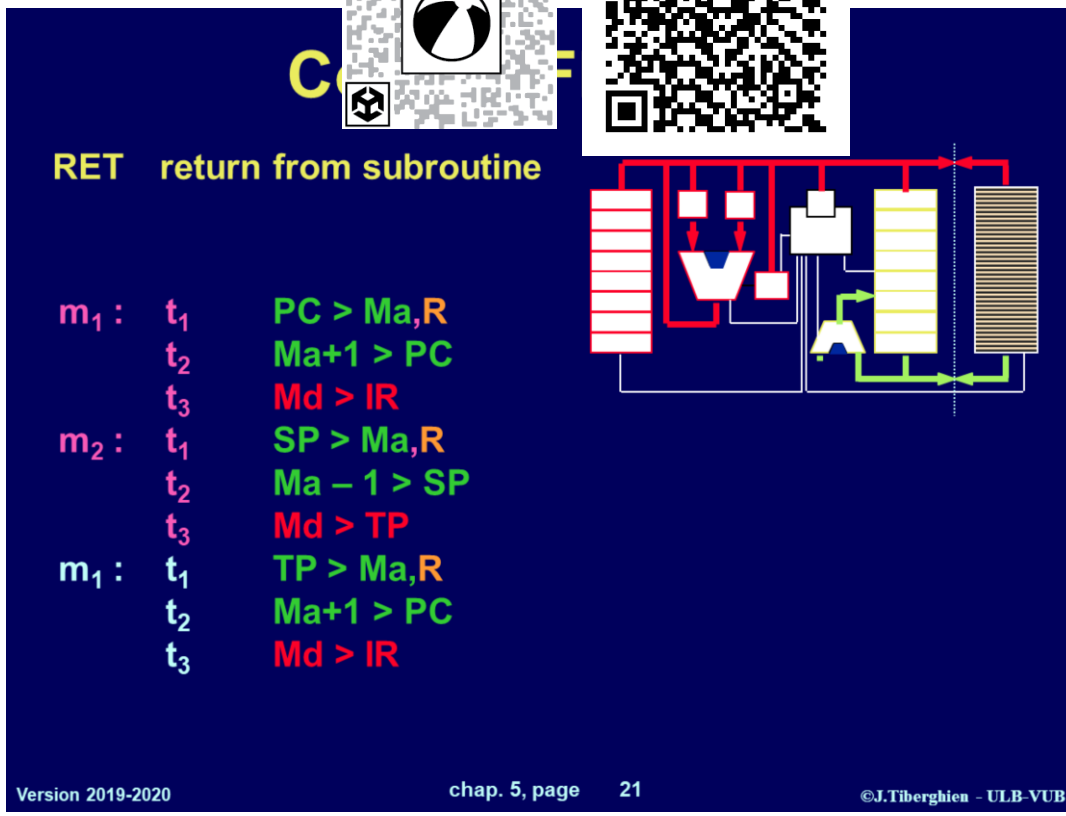
The translation in assembler of the **for** loop allows to highlight the advantages of an register architecture with respect to a stack architecture.
First expressions M and N are evaluated and their values are placed in the D0 and D1 registers. The CMP D1, D0 instruction allows to verify if it's necessary to continue the execution of the **for** loop or if we can move on rest of the program. An instruction JPN L1 skips to the rest of the program if the contents of D1 was larger than D0. After JPN instruction, the code for S is executed. It should be noted that this code can not use the D0 and D1 registers, since these are necessary for the management of the **for** loop. After the S code, an ADI D1,#1 instruction is used to increment the variable **a** stored in the D1 register. After incrementing the JMP L2 instruction causes the return to the beginning of the loop.
The big difference between this code and code for the stack processor is the result of the fact that the D0 and D1 remain accessible while with the stack processor only top is accessible and that any reading of this top makes it inaccessible.



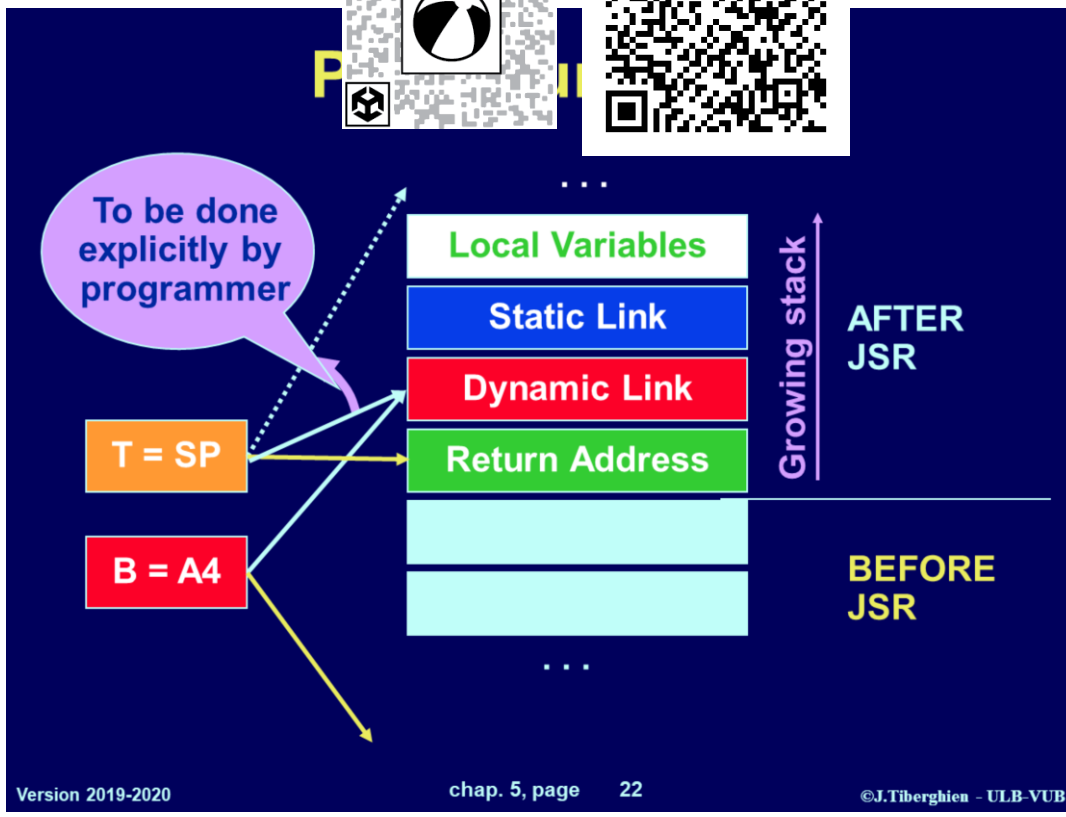
The most interesting aspect of a subroutine call instruction is the safeguard mechanism of the return address. In the RP1 a part of the memory is used to build a stack in which the return address, the dynamic and static links, activation blocks and the function parameters are stored. The address of the top of the stack is retained in the register A5, also called SP.

During the first 6 clock cycles (plus any waiting time tw) the execution of a subroutine call instruction is quite the same as a regular jump instruction. At the end of t3 of the second memory cycle one can find the TP address of the subroutine and in PC the address of the instruction following the call. The second memory cycle is extended by two clock cycles during which it increments the contents of the stack pointer SP, so that it points to the first available space on the stack. Then starts the third memory cycle in which we write the contents of the PC register in the memory at where the SP points.

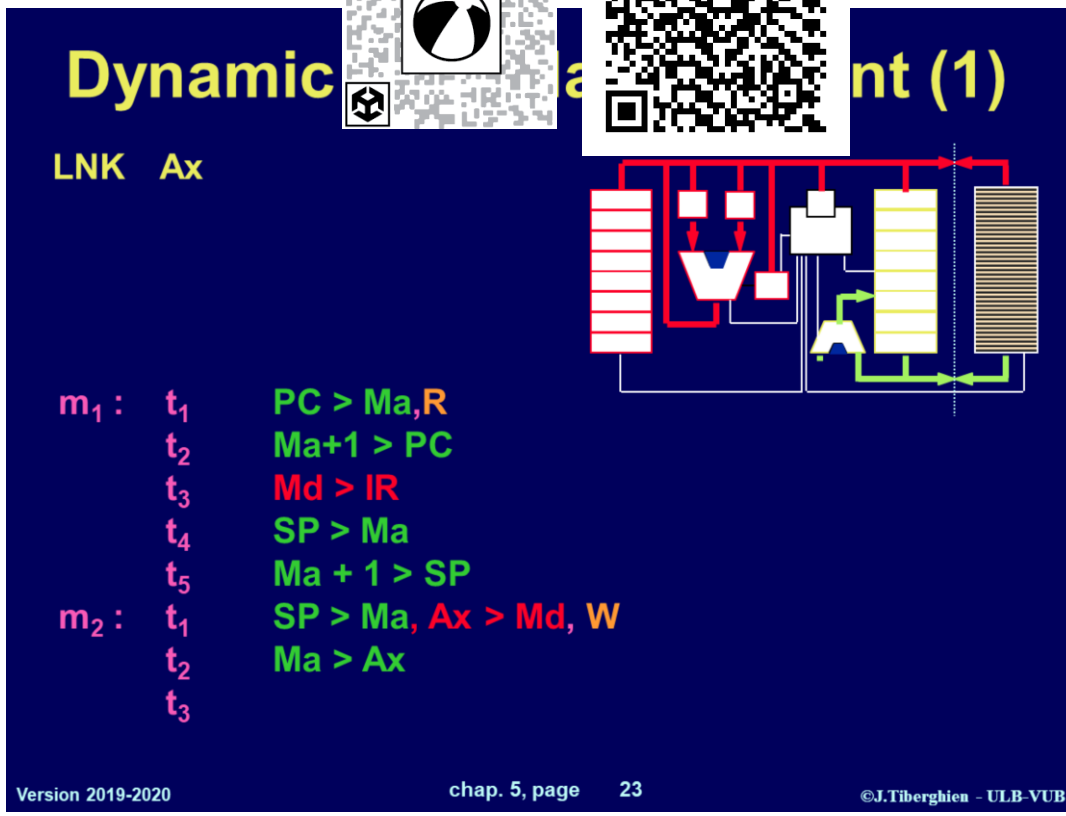
At the end of the third cycle the executing of the next instruction starts whose address will be found in TP register instead of the PC register.



The execution of a subroutine return comes down to moving the return address stored on the stack to the program counter PC.
When at the end of the first memory cycle the RET instruction is identified, the second cycle begins immediately. At t1, the content of the stack pointer SP is placed on the address bus with a reading command. At the t2 the contents of the stack pointer is decremented by one and at t3 the return address is loaded into the TP register.
After the end of the second cycle the execution of the next instruction begins by putting the contents of TP on the address bus.

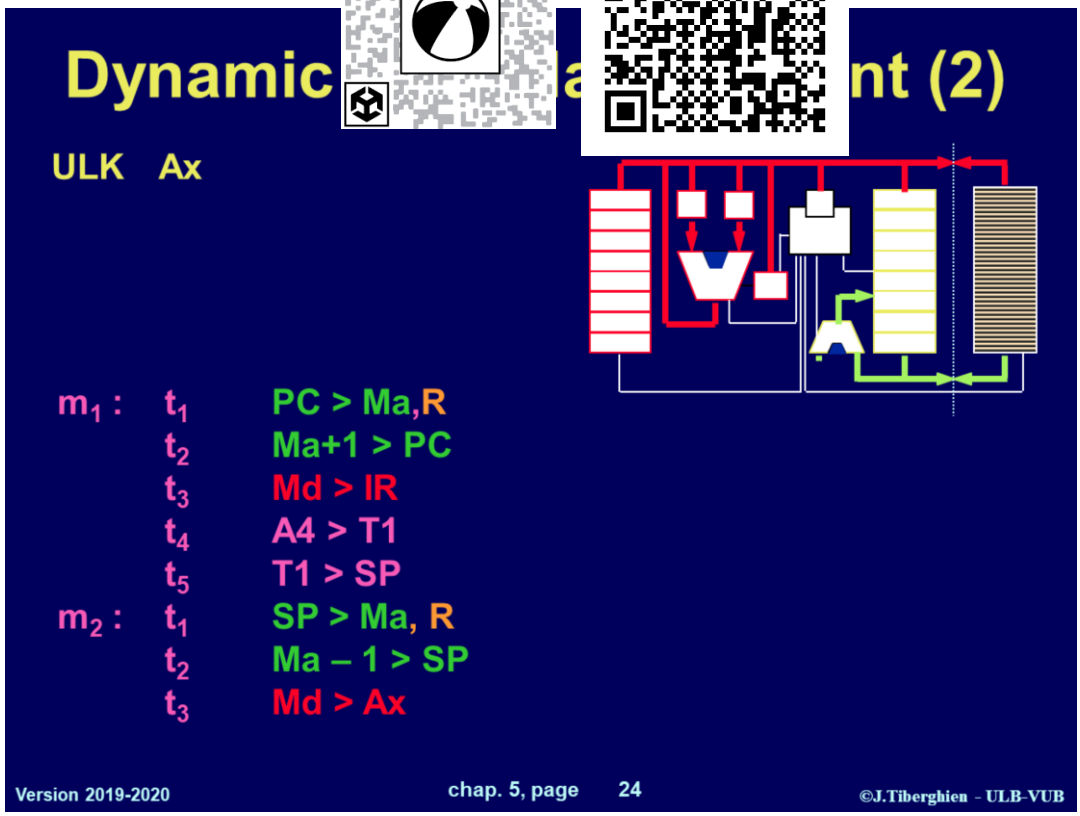


In the stack processor described in the previous chapter, call and return instructions subroutines managed not only the return addresses but also the dynamic and static links. We saw that this was not the case for RP1. This computer has two specific instructions (LNK and ULK) to manage dynamic links. It does not have specific instructions for static links because their management is so, usually, very simple.
As in the case of the stack machine, there must be a base register. Here, the A4 register will be used for this function.

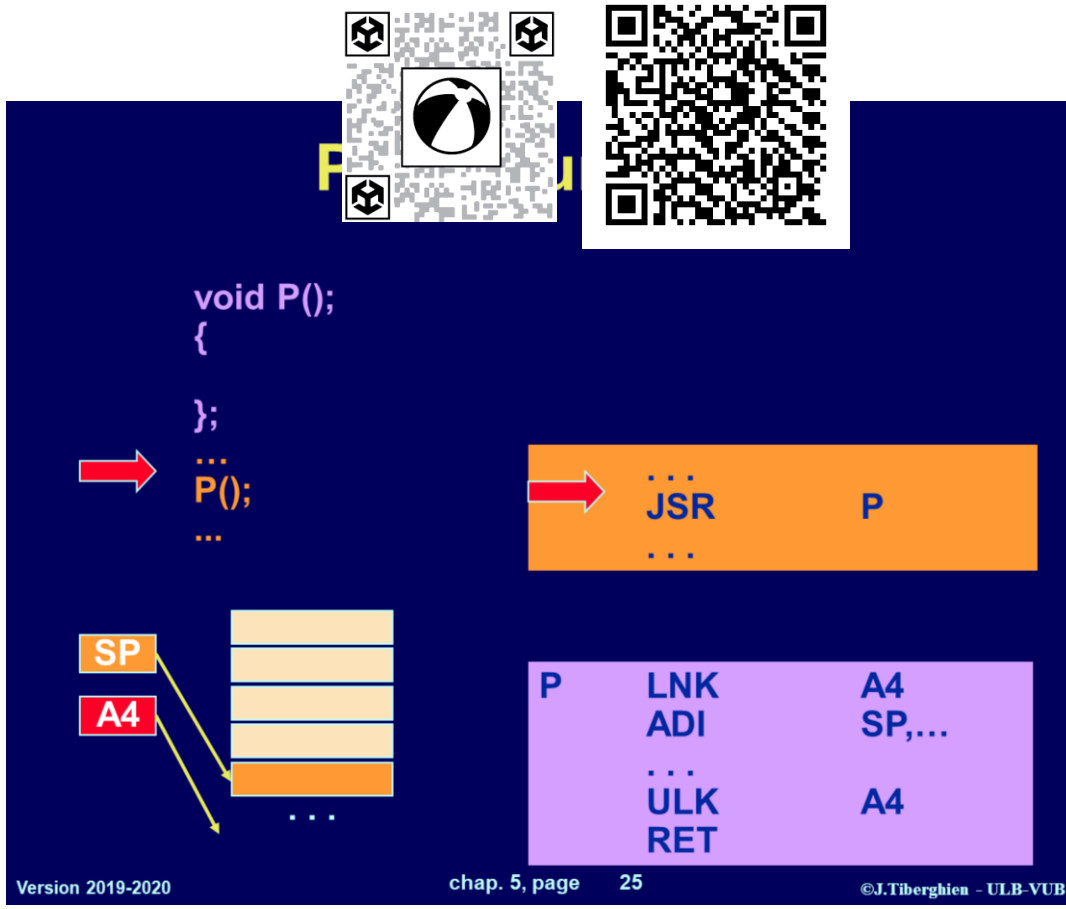


The LNK instruction is used to update the registry after a call subroutine. It is normally the first instruction of the subroutine.
The first memory cycle of the execution of LNK has 5 clock cycles the latter two are used to increment the stack pointer.
At t1 of the second cycle, the content of the stack pointer is placed on the address bus, the content of the registry is placed on the data bus and a write command is given.
At t2 of the same cycle the base register is updated by copying the present value on the address bus into the base register.

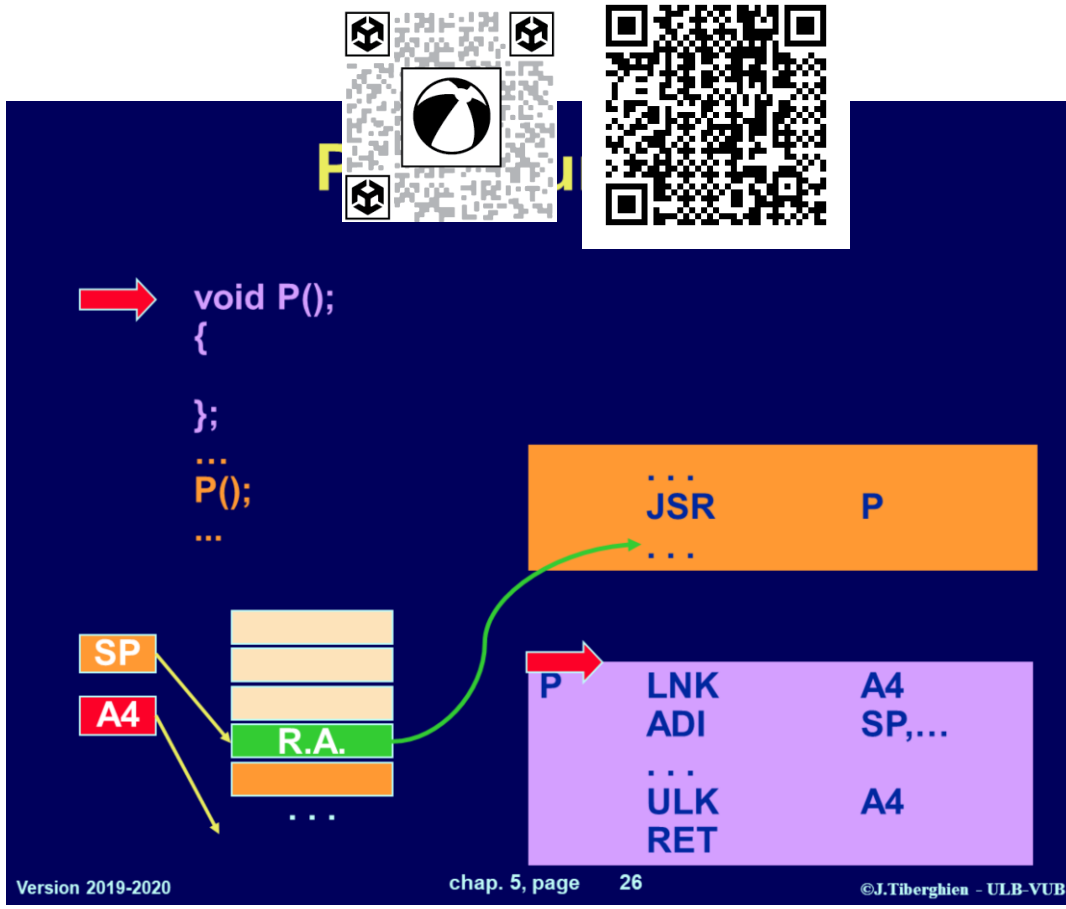
Any of the A0 to A4 can be used for that purpose, but, in general it is A4 that is used, leaving A0 to A3 for data indexing purposes.



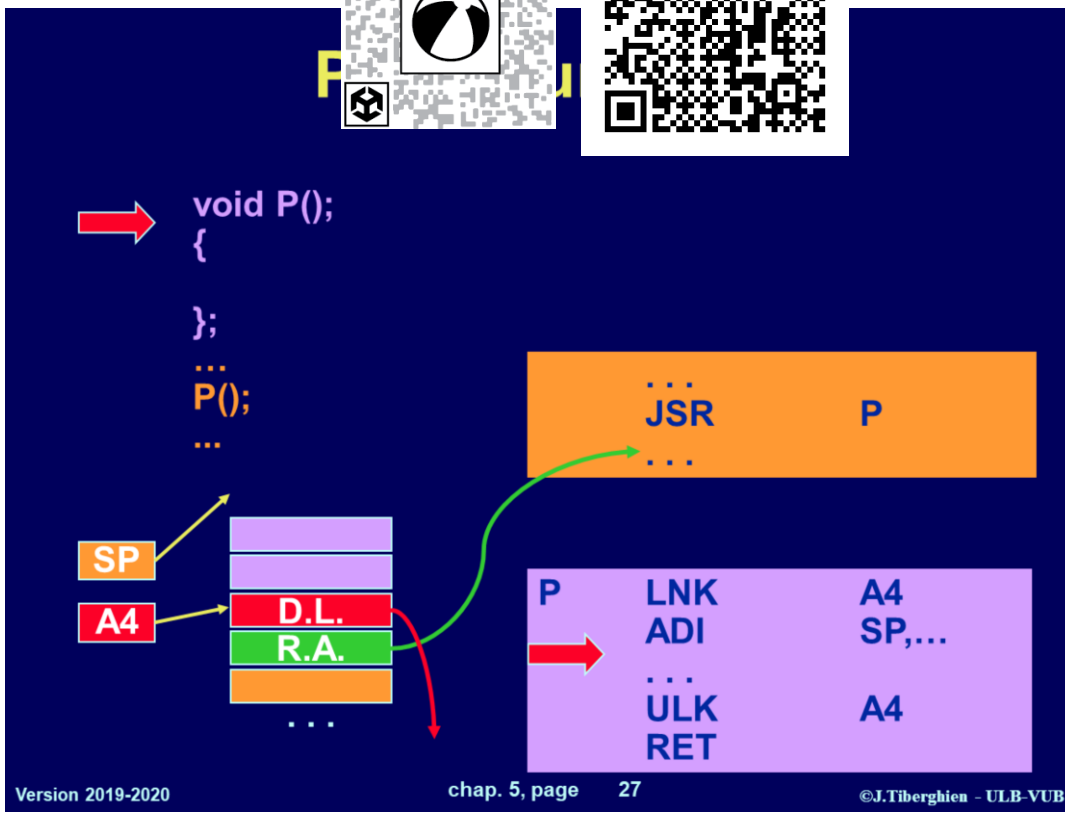
The ULK instruction is used to restore the old values of the stack pointer and the base register just before leaving the subroutine. During the last two clock cycles of the first memory cycle, the value in the base register is copied to the stack pointer to release all the space occupied by the subroutine. In the second cycle, the value of dynamic link is removed from the stack and placed in the base register.



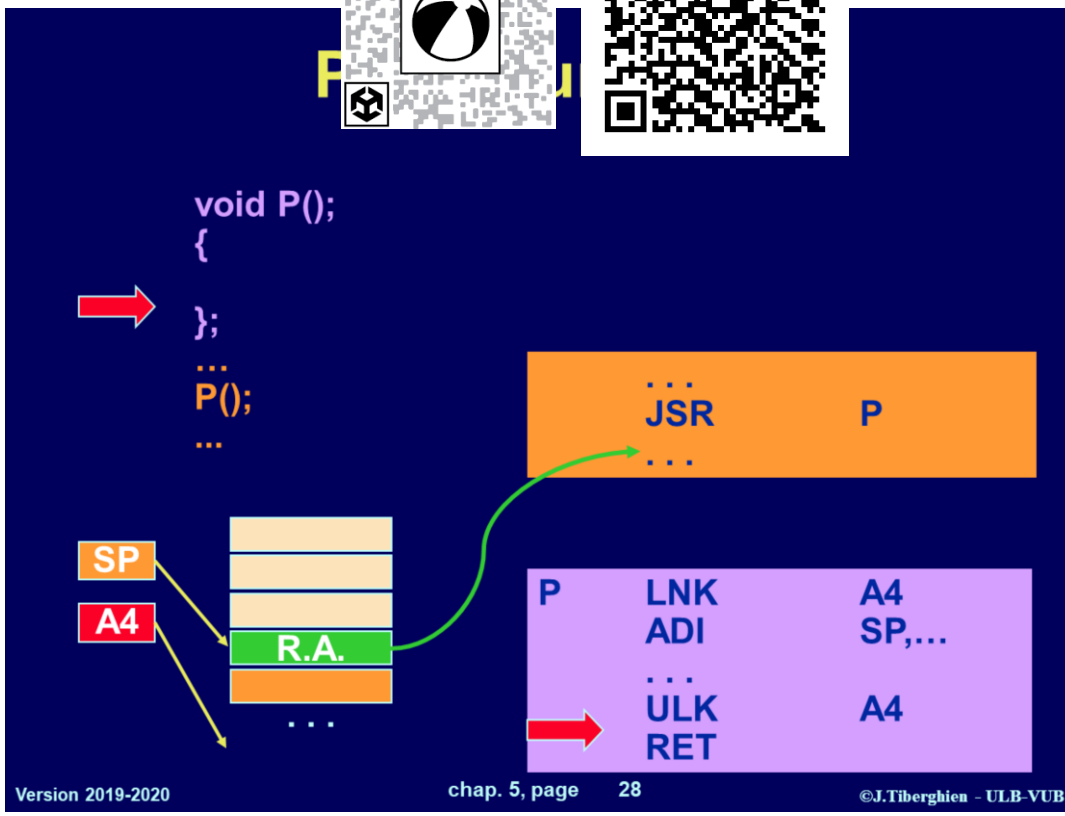
The above slide shows the use of the instructions JSR, LNK, ULK and RET.
Before calling the function P, the stack pointer SP points to the last used word on the stack and the base register A4 points at the start of the activation block of the function in which the call to P is found.



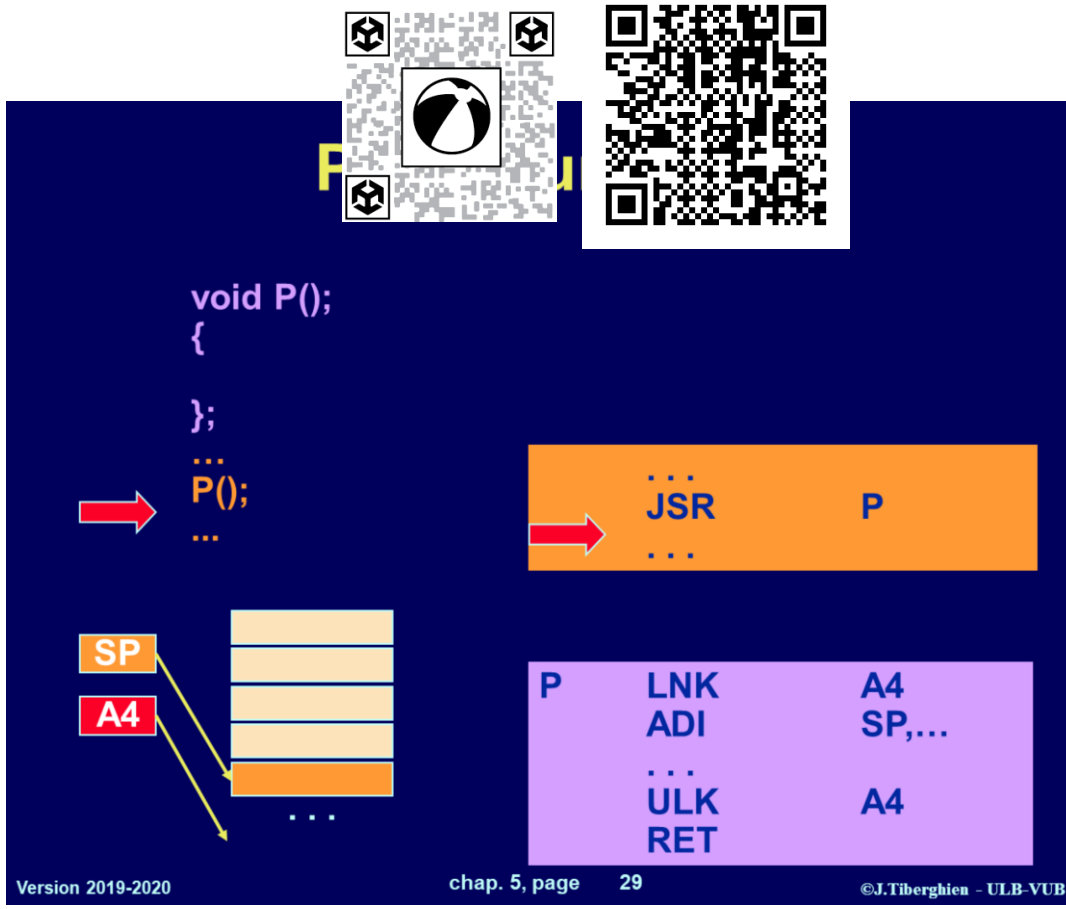
After calling the function P, the address of the following instruction in the calling function, the call (the return address RA) is on the stack.



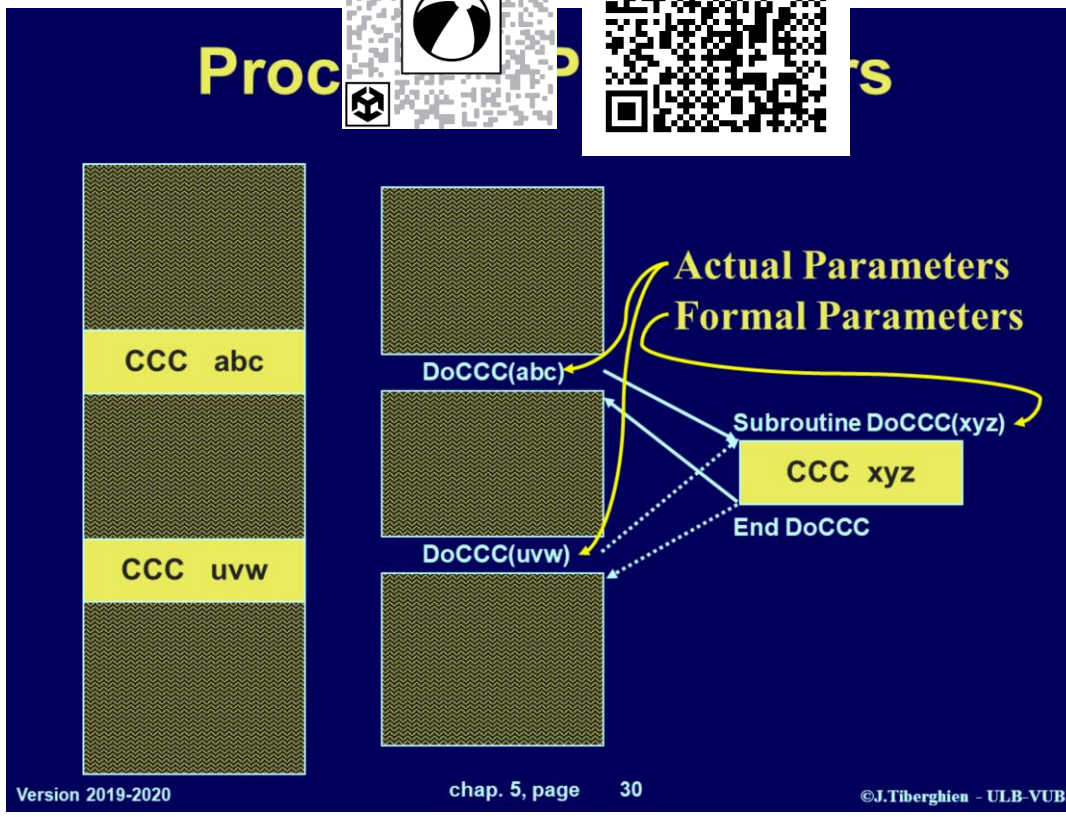
The first instruction in the function P is the LNK A4 instruction to update the base register A4. It puts on the stack, above the return address, the old value in A4 and places in A4 the address on the stack in this dynamic link.
The following statement (ADI SP ...) increments the contents of the stack pointer to reserve space for the activation block of the function P.



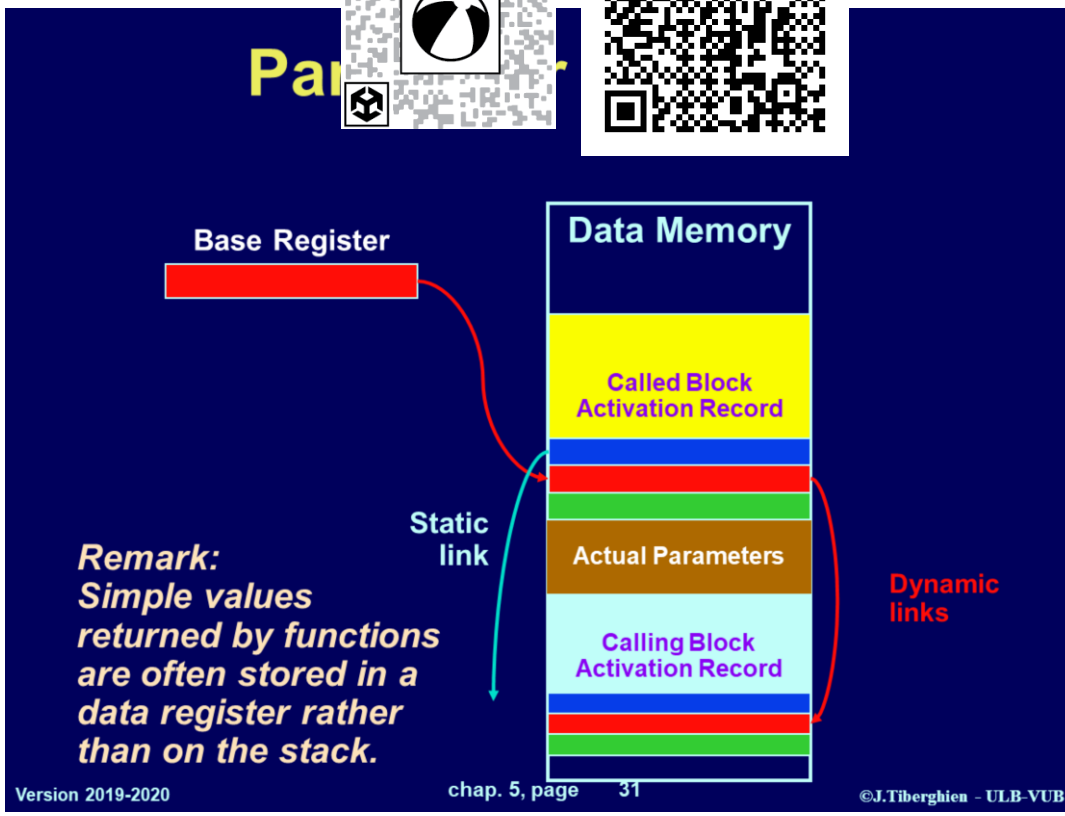
The last but one instruction of the function P is ULK A4 instruction that restores the value in A4 before executing the LNK instruction at the start of the function P. It also frees all the space on the stack for the function P by placing in SP the address return address.



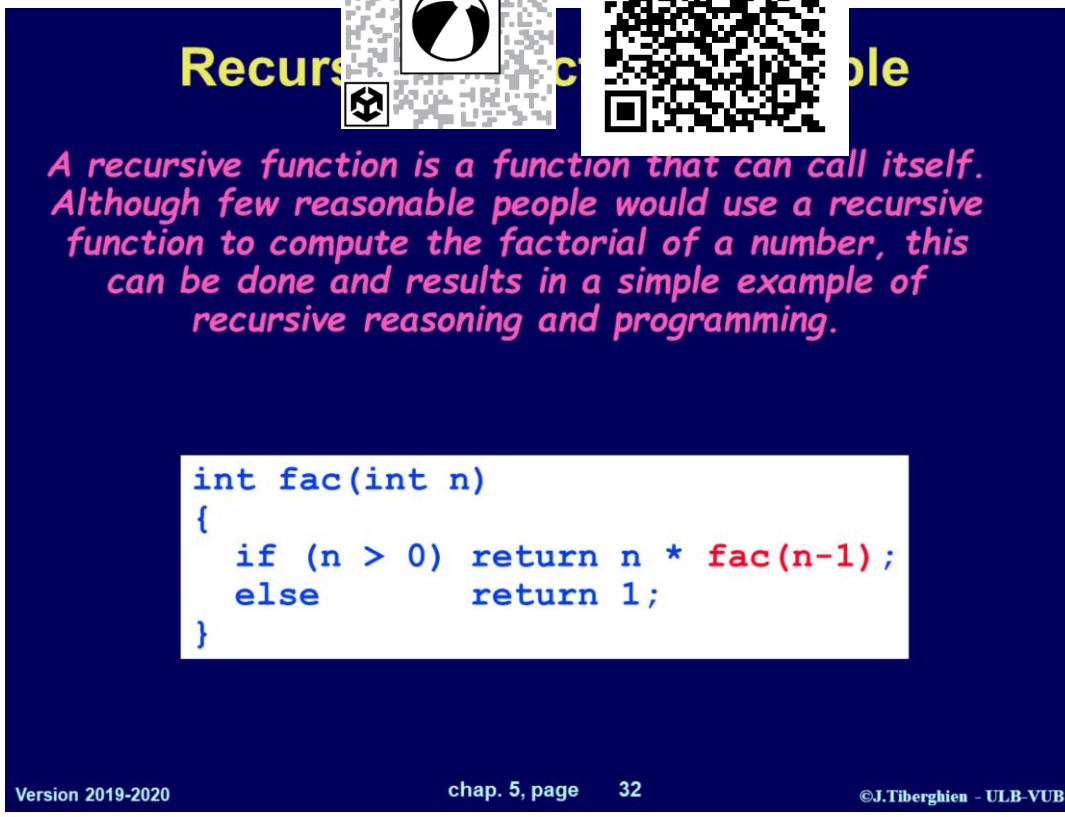
After the return of the function P execution continues from the return address RA, which is also removed from the stack. (Actually, it remains on the stack, but the stack pointer was decremented, it became practically inaccessible and will be replaced by other data at the next write on the stack)



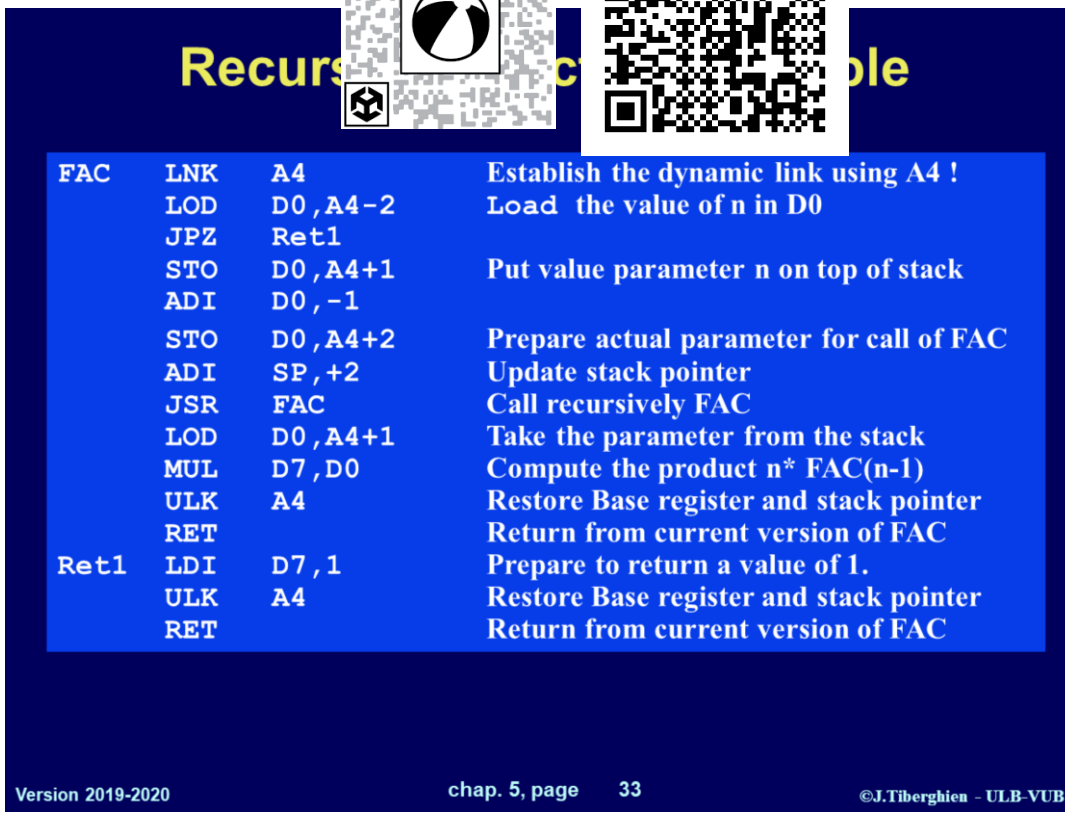
To complete the study of the call subroutines in RP1, we must also consider the parameter passing mechanism.



The parameter passing is done in exactly the same way in RP1 as in the stack processor studied in the previous chapter. The parameters are placed on the stack, on top of activation black of the calling function and below the return address. For parameters passed by value it is the current value of the parameter that is placed on the stack while for parameters passed by reference, this is their address.
The only difference with the stack processor lies in the values returned by functions. While in the stack processors a place was reserved for this value below the stack parameters, it's usually in a data register that the value of a function is returned in a register processor.



A recursive function is a function that can call itself.
Although for performance reasons it is not recommended to calculate the factorial function recursively, we can do it to have a very simple example of using recursion.
The factorial function written in C is simple since it literally transpose the mathematical definition of the factorial function in the C language



The slide shows the translation in assembler, with comments, of the recursive factorial function.
It is strongly recommended the reader to follow step by step the execution of this function when it has to calculate 3! or 4! It is particularly important to draw the successive states of the stack for this execution to understand the mechanisms of implementation of a recursive function.