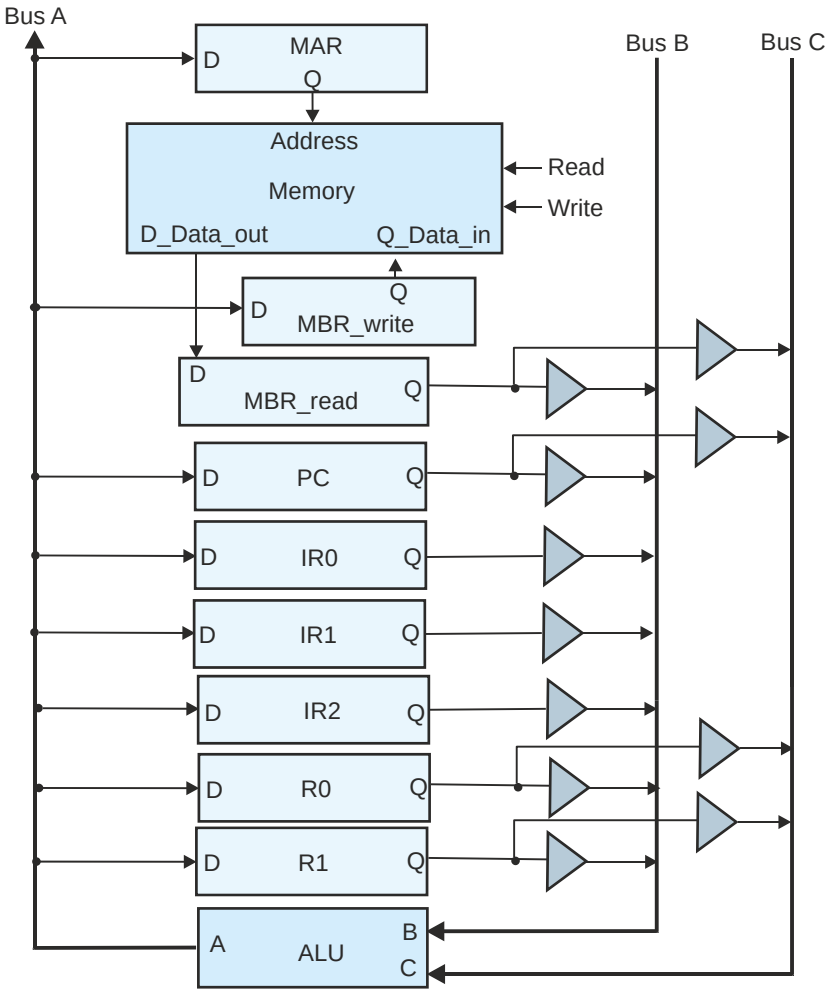
Example of A Microprogrammed Computer

The purpose of this example is to demonstrate some of the concepts of microprogramming. We are going to create a simple 16-bit computer that uses three buses A, B, and C. Bus C receives its input from the ALU and provides an output to all registers. Buses A and B provide inputs to the ALU from the registers. The following figure illustrates the CPU's bus and register structure.



All registers apart from the memory address register are connected to Bus, B but only some registers are connected to bus C.

The computer has two general-purpose registers R0 and R1, and three instruction registers I0, I1 and I2. The three instruction registers are an unusual feature of this processor.

The program and data memory has three dedicated registers. There is a memory address register, MAR, and two memory buffer registers, MBR_read and MBR_write, The difference between the two MBRs is that MBR_read receives data from memory during a read cycle, and MBR_write provides data to memory in a write cycle. The use of separate read and write MBRs simplified control and multiplexing arrangements.

A key feature of the ISA (instruction set architecture) of this processor is that all instructions are three words long and consist of an op-code, an address, and an operand. Both the address and operand are 16 bits wide. This arrangement solves the problem of operand range (it's the same as the register width), and addressing range (the full 16-bit address space is supported). Each instruction is, therefore 48 bits in total and must be read as three consecutive words. Many instructions will not need all 48 bits and, therefore, the architecture does not use program space efficiently.

The forms of the instructions supported are given as follows. Note that the destination operand is first (leftmost) and is in bold font.

Register-to-register:	`MOVE` **`R0,`**`R1,`	`ADD` **`R0,`**`R0,R1`
Register-to-memory:	`MOVE` **`0x1234,`**`R0,`	`ADD` **`0x1234,`**`R0`
Memory-to-register:	`LOAD` **`R0,`**`0x1234,`	`ADD` **`R0,`**`0x1234`
Indexed memory:	`MOVE` **`R0,`**`0x1234(R1)`	
Literal:	`ADD` **`R0,`**`#0x1234`	

The classes of instruction are:

- Data movement
- Program flow control: `JMP 0x1234`, `JMP R0`, `JMP 0x1234(PC)`, `JMP 0x1234(PC,R0)`
- Data processing: Arithmetic, shift, logical, compare
- Conditional branch

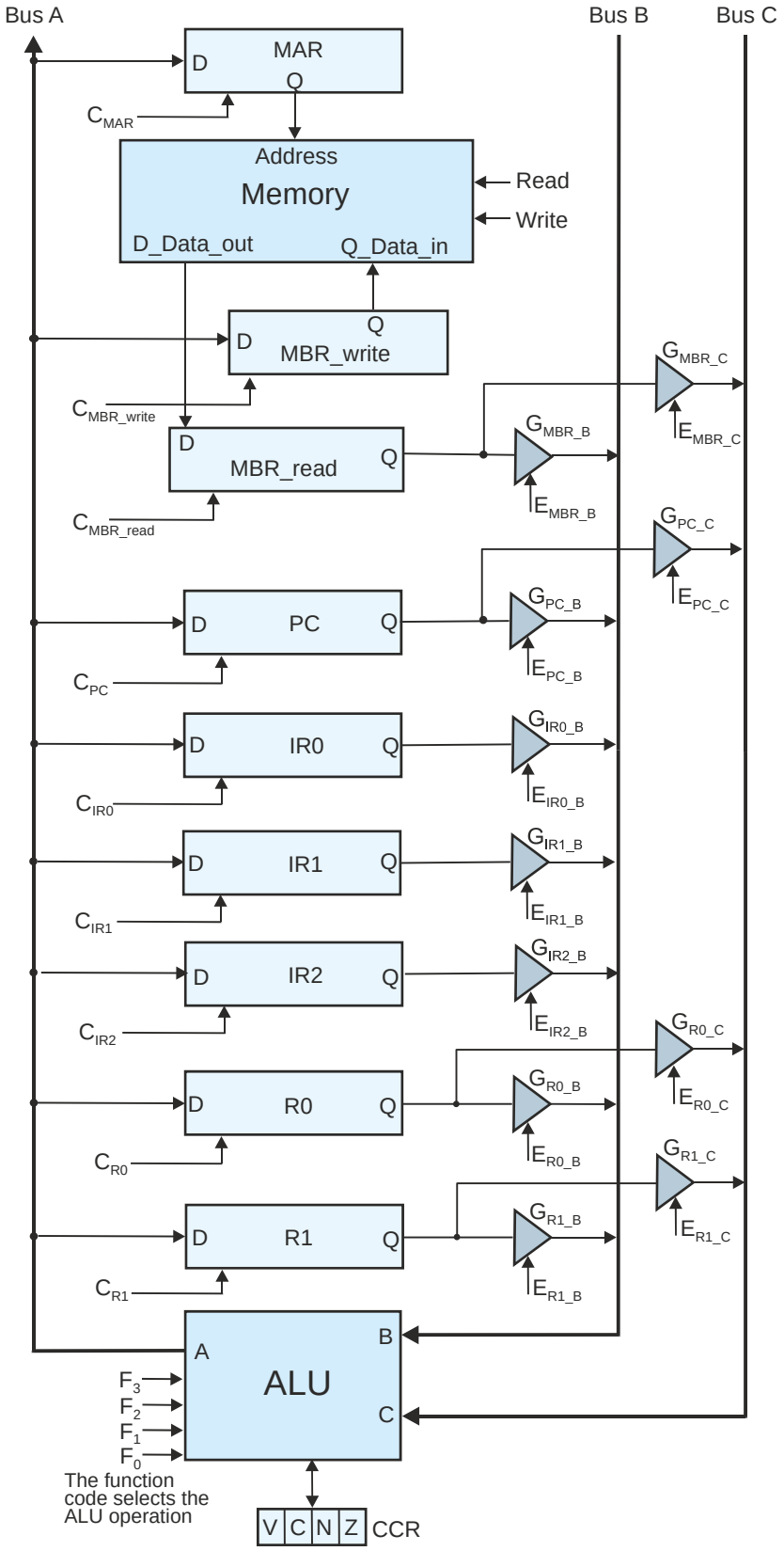
The instruction format is:

Bits Function
15 – 13 Predicate
12 – 10 Instruction class
9 – 8 Source register
7 – 6 Destination register
5 – 2 Instruction parameter
1 – 0 Constant 0 to 3

The control signals are:

- Memory: read, write
- Registers: 9 clock, 11 tri-state enables
- ALU: 4 function selects.

The next figure shows the structure of the computer with the bus control and ALU signals.


Bus A
Bus B
Bus C
MAR
D
Q
C_{MAR}
Address
Memory
Read
Write
D_Data_out
Q_Data_in
MBR_write
C_{MBR_write}
MBR_read
C_{MBR_read}
G_{MBR_C}
E_{MBR_C}
G_{MBR_B}
E_{MBR_B}
G_{PC_C}
E_{PC_C}
PC
C_{PC}
G_{PC_B}
E_{PC_B}
IR0
C_{IR0}
G_{IR0_B}
E_{IR0_B}
IR1
C_{IR1}
G_{IR1_B}
E_{IR1_B}
IR2
C_{IR2}
G_{IR2_B}
E_{IR2_B}
G_{R0_C}
E_{R0_C}
R0
C_{R0}
G_{R0_B}
E_{R0_B}
G_{R1_C}
E_{R1_C}
R1
C_{R1}
G_{R1_B}
E_{R1_B}
A
B
C
ALU
F_3
F_2
F_1
F_0
The function code selects the ALU operation
V C N Z
CCR

The fetch cycle

The fetch phase of each instruction requires the reading of three consecutive words, even if the instruction does not need all three words. If we assume that the contents of the PC point to the next instruction to be fetched, the fetch phase can be represented in register transfer language, RTL, as:

[MAR] ← [PC] ; Fetch first word (the op-code)
[PC] ← [PC] + 1
[MBR_{read}] ← [[MAR]]
[IR0] ← [MBR_{read}]

[MAR] ← [PC] ; Fetch second word (the 16-bit address)
[PC] ← [PC] + 1
[MBR_{read}] ← [[MAR]]
[IR1] ← [MBR_{read}]

[MAR] ← [PC] ; Fetch third word (the 16-bit literal)
[PC] ← [PC] + 1
[MBR_{read}] ← [[MAR]]
[IR2] ← [MBR_{read}]

The notation [R0] means the contents of register R0, and the notation [0x1234] means the contents of memory location 0x1234. The notation [[R0]] means the contents of memory whose address is given by the content's of register R0. This is, of course, register-indirect addressing.

Note that the PC is incremented by 1 rather than 2 because this is a 16-bit word-addressed machine. Individual bytes cannot be accessed.

We can represent these operations in terms of control signals as:

E_{PC_B}, F(pass), C_{MAR}
Read, C_{MBR_read}
$E_{MBR\_read\_B}$, F(pass), C_{IR0}
E_{PC_B}, F(pass), C_{MAR}
Read, C_{MBR_read}
$E_{MBR\_read\_B}$, F(pass), C_{IR1}
E_{PC_B}, F(pass), C_{MAR}
Read, C_{MBR_read}
$E_{MBR\_read\_B}$, F(pass), C_{IR2}

Note that in the above microinstruction sequence, the same three microoperations are repeated three times and the only difference in each case is which destination instruction register is clocked.

The Execute Phase

Having fetched the instruction as three words, the next step is to execute it. Let's look at a few operations.

`ADD` **`R0,`**`R1` defined as [R0] ← [R0] + [R1]

In this case, we only have to copy R0 and R1 to the B and C buses, set the ALU function code to add, and then clock R0; that is E_{R0_B}, E_{R1_C}, F(add), C_{R0}

Suppose the instruction had been `ADD` **`R0,`**`P`. In this case, we would have to obtain memory address P from instruction register IR1 and use it to access memory; that is,

[MAR] ← [IR1] ; Copy the address in IR1 to the memory address register
[MBR_{read}] ← [[MAR]] ; Read from memory into the MBR
[R0] ← [MBR_{read}] + [R0] ; Add the data from memory to R0 and store the result in R0

The sequence of microoperations corresponding to these machine level instruction are:

E_{IR1_B}, F(pass), C_{MAR}
Read, C_{MBR_read}
E_{R0_B}, MBR_{read_C}, F(add), C_{R0}

Let's consider a more adventurous `ADD **R0,**(R1,0x1234)`. In this case we are using address register indirect with indexing. The index value is in register IR2.

The RTL form of the instruction is [R0] ← [R0] + [[R1] + [IR2]]. We first have to add together the contents of pointer register R1 and instruction register IR2 that contains the offset.

E_{IR2_B}, E_{R1_C}, F(add), C_{IR2}	; This sequence ends with the operand address in IR2
E_{IR2_B}, F(pass), C_{MAR}	; now proceed as before and pass the address to the MAR
Read, C_{MBR_read}	; read the actual operand
E_{R0_B}, EMR_{read_C}, F(add), C_{R0}	; and add it to R0

Literal Operations

Finally, consider an operation with a literal. The instruction `ADD **R0,**0x1234` is represented in RTL by [R0] ← 0x1234.

Implementing this instruction couldn't be easier, because it is the same as adding two registers, except that one register is instruction register IR2. That is, is E_{R0_C}, E_{IR2_B}, F(add), C_{R0}

Store Literal

Now consider a store literal indexed in memory that uses two constants. In this case, we are going to load a 16-bit constant onto 16-bit memory with `STORE **(0x1234,R0),**0x5678`. The RTL version is [[R0] + 0x1234] ← 0x5678. Here we use both instruction registers IR1 and IR2 that contain the address and the constant (offset), respectively.

E_{IR2_B}, E_{R0_C}, F(add), C_{MAR}	; This gives us the operand address of R0 + IR2 in the memory address register
E_{IR1_B}, F(pass), C_{MBR_write}	; Put the literal that's in IR1 in the memory buffer register for writing
Write	; and write to memory

Branch and Jump Operations

Branch and jump operations all lead to the reloading of the program counter. In general (but not exclusively) the term branch tends to indicate a program-counter relative branch, whereas a jump tends to mean a jump to an absolute address.

The program counter can be loaded from instruction register IR1 (absolute jump), a data register (register indirect jump), program counter plus the offset in IR2 (relative address), or even program counter plus register (program counter register indirect). Moreover, we can make loading the program counter dependent on the condition codes to provide conditional branches or jumps.

Let's consider a conditional program counter relative branch of `BEQ target` which is expressed as

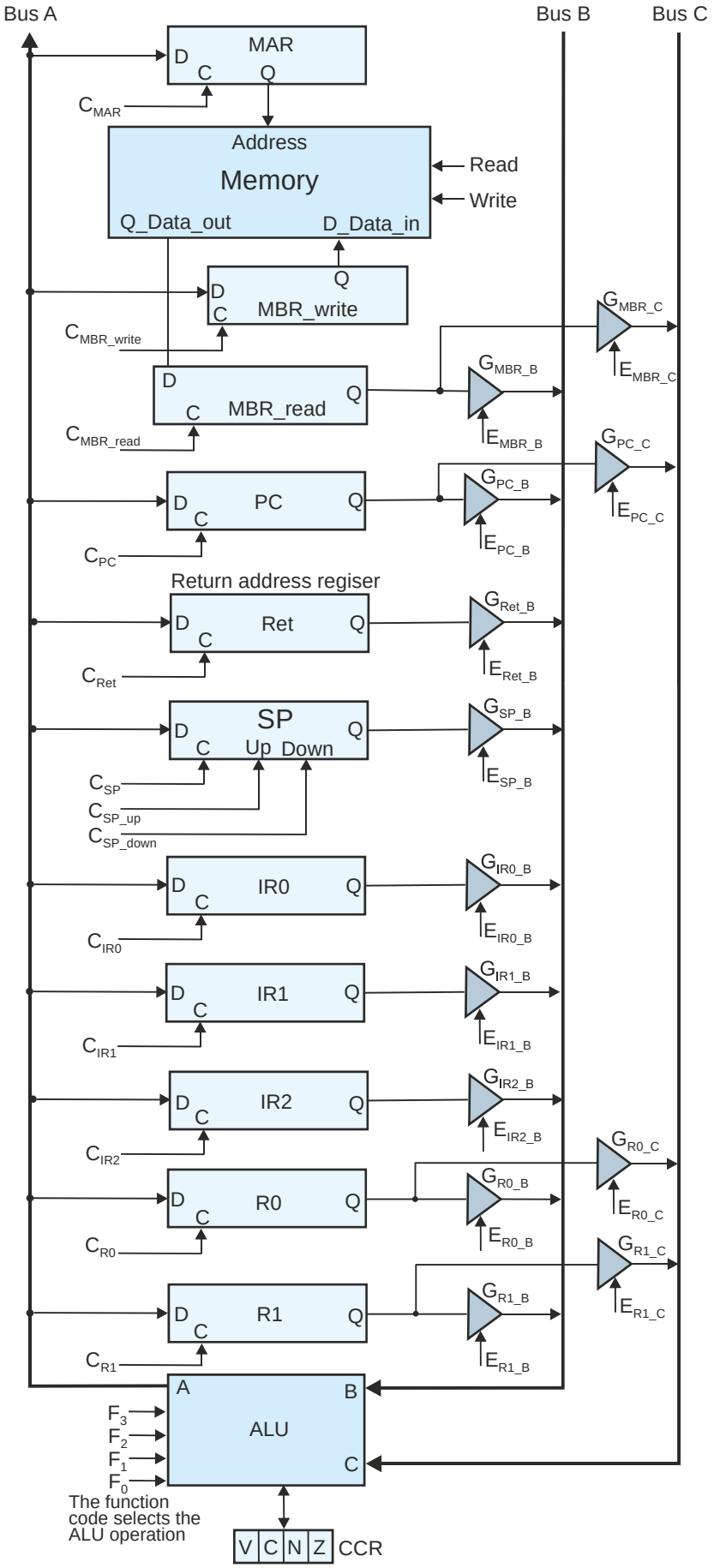
IF Z THEN [PC] ← [PC] + [IR2]

The sequence of microoperations for this instruction is remarkably simple.

E_{IR2_B}, E_{PC_C}, F(add), IF Z = 1 THEN C_{PC}	; This generates a new target address and ; clocks the PC if the Z-bit is set.

Extending the Processor

We can extend the processor in two simple ways to increase its flexibility. First, we are going to give it a return address register, Ret, to allow returns from a subroutine, and, second, we are going to add a stack pointer. The return address register will be used in the same way as ARM's r14, the link register.


Bus A
Bus B
Bus C
MAR
D
C
Q
C_{MAR}
Address
Memory
Read
Write
Q_Data_out
D_Data_in
MBR_write
C_{MBR_write}
MBR_read
C_{MBR_read}
PC
C_{PC}
Return address regiser
Ret
C_{Ret}
SP
Up Down
C_{SP}
C_{SP_up}
C_{SP_down}
IR0
C_{IR0}
IR1
C_{IR1}
IR2
C_{IR2}
R0
C_{R0}
R1
C_{R1}
G_{MBR_C}
E_{MBR_C}
G_{MBR_B}
E_{MBR_B}
G_{PC_C}
E_{PC_C}
G_{PC_B}
E_{PC_B}
G_{Ret_B}
E_{Ret_B}
G_{SP_B}
E_{SP_B}
G_{IR0_B}
E_{IR0_B}
G_{IR1_B}
E_{IR1_B}
G_{IR2_B}
E_{IR2_B}
G_{R0_C}
E_{R0_C}
G_{R0_B}
E_{R0_B}
G_{R1_C}
E_{R1_C}
G_{R1_B}
E_{R1_B}
A
B
ALU
C
F_3
F_2
F_1
F_0
The function code selects the ALU operation
V C N Z
CCR

We have created a stack pointer, SP, that can be loaded from the A bus by clocking C_{SP}. Similarly, the stack pointer's output can be deposited on the B bus by enabling bus control E_{SP_B}. The stack pointer is an up/down counter with up-clock and a down-clock inputs. These two clock inputs allow us to increment it, and to decrement it, respectively.

The return addresst register is used to store subroutine return addresses. Consider the instruction pair `CALL target` and RETURN. The RTL forms of these instructions are:

`CALL target`: [Ret] ← [PC]; [PC] ← [PC] + [IR2]; Save PC and branch to PC plus offset

`RETURN`: [PC] ← [Ret]

We can implement these two instructions in terms of the following microoperations.

`CALL target`:	E_{PC_B}, F(pass), C_{Ret}	;save return address
	E_{IR2_B}, E_{PC_C}, F(add), C_{PC}	;jump to new address
`RETURN`:	E_{Ret_B}, F(pass), C_{PC}	;restore return address to PC

Push and Pull Operations

We can load and store the contents of the stack pointer using any of the microoperation sequences we used for the same operations on the R0 and R1 registers; all we need do is exchange R0 to SP, etc. Here we are ging to implement `PUSH R0` and `PULL R0` operations. We assume a full descending stack; that is the stack pointer points at the top item on the stack and that the stack pointer grows upwards to low addresses and is decremented before pushing and incremented after popping.

`PUSH R0` is defined in RTL as [SP] ← [SP] – 1; [[SP]] ← [R0]

`PULL R0` is defined in RTL as; [R0] ← [[SP]]; [SP] ← [SP] + 1

The microoperations required to implement these are:

`PUSH R0`:	C_{SP_down}	; decrement the stack pointer
	E_{SP_B}, F(pass), C_{MAR}	**;** SP to MAR for memory write
	E_{R0_B}, F(pass), C_{MBR_write}	; R0 to MBR for memory write
	Write	; do the write and push R0
`PULL R0`:	E_{SP_B}, F(pass), C_{MAR}, C_{SP_up}	; SP to MAR for memory read, and post increment SP
	Read, C_{MBR_read}	; read memory and clock top of stack into MBR
	$E_{MBR\_read\_B}$, F(pass), C_{R0}	; MBR to R0

Subroutine Call and Return Using the Stack

Now let's implement the traditional CISC-style instructions `JSR target`, and `RTS` that perform a subroutine call and return by pushing the return address on the stack (call) and pulling the return address off the stack (return). Some CISC processors used an absolute address with a JSR, rather than a program counter relative address. We can easily do this because all instructions have a 16-bit address field.

The RTL forms of these two instructions are:

`JSR target`: [SP] ← [SP] - 1; [[SP]] ← [PC]; [PC] ← [IR1]; Save PC and load target address in PC to call

`RTS`: [PC] ← [[SP]]; : [SP] ← [SP] + 1

We can implement these in terms of the following microoperations.

`JSR target:`	C_{SP_down}	;decrement the stack pointer
	E_{SP_B}, F(pass), C_{MAR}	;SP to MAR for memory write
	E_{PC0_B}, F(pass), C_{MBR_write}	;PC to MBR for memory write
	Write	;do the write and push R0
	E_{IR1_B}, F(pass), C_{PC}	;jump to new address
`RTS:`	E_{SP_B}, F(pass), C_{MAR}, C_{SP_up}	;SP to MAR for memory read, and post increment SP
	Read, C_{MBR_read}	;read memory and clock top of stack into MBR
	$E_{MBR\_read\_B}$, F(pass), C_{PC}	;MBR to PC