BAC Computer

1 Introduction

BAC is an extremely simple computer conceived as a design example for FPGA. The aim was to achieve a CPU with less than 300 logic cells in an FPGA of the ICE40HX type, with 8 bits of data width, and with memory storage in the internal BRAM blocks of the FPGA.

Regarding the name, it is derived from BAC-teria, the simplest possible self-sufficient living beings. Furthermore, the -AC ending puts it in line with other famous computers ("ENIAC", "UNIVAC"… ;)

This design had its origin in a discussion on the "FPGA-Wars" forum ([https://groups.google.com/g/fpga-wars-explorando-el-lado-libre/c/BpvDRmLioS4]) in which the basic ideas of what this design would end up being were presented.

2 Programmer's model

BAC has the following features::

- Harvard Architecture: Separate program and data memory.
  - Program memory: 256 x 16 max.
  - Data memory: 256 x 8 max.
- 16 bit wide instructions: 8 bits for op-code + 8 bits for a literal operand
- Registers:
  - PC: 8 bits, counter, write only. Points to the instruction to be executed in program memory. It is incremented every clock cycle except when written when executing jump instructions.
  - X: 8 bits, pointer, write only, points to data memory
  - Acc: 8 bits, accumulator, contains one of the data for two-operand instructions and can also store the results of the instructions
  - Flags: 3 bits, carry (C), zero (Z), and negative (N).
- Addressing modes:
  - Immediate or Literal: The operand is in the 8 least significant bits of the instruction.
  - Direct: The operand is in the data memory position pointed by the 8 least significant bits of the instruction.
  - Indexed: The operand is at the data memory position pointed by register X.
- The results of two-operand instructions can be written to the Acc register or to data memory (only if direct or indexed addressing is selected), as desired. Some single-operand instructions always store the result in data memory (for example INC).
- Specific instructions for input and output: IN, OUT.

2.1 Instruction set

The coding of the instructions follows this format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction						INDX	NLIT	Literal data							

The instructions are:

Instruction	Mod. Flags	Mnemonic	Addr. modes			Description
			Lit	Dir	inX	
0000.0x	-,-,-	NOP	implicit			
0000.10	-,-,-	JMP	y	y	y	unconditional jump
0000.11	-,-,-	JMPD	y	y	y	unconditional jump, delayed
0001.00	-,-,-	JNC	y	y	y	jump if C==0
0001.01	-,-,-	JNCD	y	y	y	jump if C==0, delayed
0001.10	-,-,-	JC	y	y	y	jump if C==1
0001.11	-,-,-	JCD	y	y	y	jump if C==1, delayed
0010.00	-,-,-	JNZ	y	y	y	jump if Z==0
0010.01	-,-,-	JNZD	y	y	y	jump if Z==0, delayed
0010.10	-,-,-	JZ	y	y	y	jump if Z==1
0010.11	-,-,-	JZD	y	y	y	jump if Z==1, delayed
0011.00	-,-,-	JPL	y	y	y	jump if N==0
0011.01	-,-,-	JPLD	y	y	y	jump if N==0, delayed
0011.10	-,-,-	JMI	y	y	y	jump if N==1
0011.11	-,-,-	JMID	y	y	y	jump if N==1, delayed
0100.00	N,Z,-	LDA	y	y	y	Acc = op
0100.01	N,Z,-	IN	-	y	y	Acc = IO[addr]
0100.10	-,-,-	LDX	y	y	y	X = op
0101.00	-,-,-	STA	-	y	y	Mem[addr] = Acc
0101.01	-,-,-	OUT	-	y	y	IO[addr] = Acc
0101.10	-,-,-	TAX	Implicit			X = Acc
1000.00	N,Z,C	ADDA	y	y	y	Acc = op + Acc
1000.01	N,Z,C	ADDM	-	y	y	Mem[addr] = Mem[addr] + Acc
1000.10	N,Z,C	ADCA	y	y	y	Acc = op + Acc + C
1000.11	N,Z,C	ADCM	-	y	y	Mem[addr] = Mem[addr] + Acc + C
1001.00	N,Z,C	SUBA	y	y	y	Acc = op - Acc
1001.01	N,Z,C	SUBM	-	y	y	Mem[addr] = Mem[addr] - Acc
1001.10	N,Z,C	SBCA	y	y	y	Acc = op - Acc - /C
1001.11	N,Z,C	SBCM	-	y	y	Mem[addr] = Mem[addr] - Acc - /C

Instruction	Mod. Flags	Mnemonic	Addr. modes			Description
			Lit	Dir	inX	
1010.00	N,Z,C	CMP	y	y	y	op - Acc (result not stored)
1010.01	N,Z,-	TST	y	y	y	op & Acc (result not stored)
1010.10	N,Z,C	ROR	-	y	y	Mem[addr] = {C,$Mem_{7:1}$} , C=Mem_0
1011.00	N,Z,-	ANDA	y	y	y	Acc = op & Acc
1011.01	N,Z,-	ANDM	-	y	y	Mem[addr] = Mem[addr] & Acc
1011.10	N,Z,-	ORA	y	y	y	Acc = op | Acc
1011.11	N,Z,-	ORM	-	y	y	Mem[addr] = Mem[addr] | Acc
1100.00	N,Z,-	XORA	y	y	y	Acc = op ^ Acc
1100.01	N,Z,-	XORM	-	y	y	Mem[addr] = Mem[addr] ^ Acc
1101.00	N,Z,-	INC	-	y	y	Mem[addr] = Mem[addr] + 1
1101.01	N,Z,-	INCA	-	y	y	Acc = Mem[addr] = Mem[addr] + 1
1101.10	N,Z,-	INCX	-	y	y	X = Mem[addr] = Mem[addr] + 1
1101.11	N,Z,-	INCAX	-	y	y	Acc = X = Mem[addr] = Mem[addr] + 1
1110.00	N,Z,-	DEC	-	y	y	Mem[addr] = Mem[addr] - 1
1110.01	N,Z,-	DECA	-	y	y	Acc = Mem[addr] = Mem[addr] - 1
1110.10	N,Z,-	DECX	-	y	y	X = Mem[addr] = Mem[addr] - 1
1110.11	N,Z,-	DECAX	-	y	y	Acc = X = Mem[addr] = Mem[addr] - 1

Beware of the order of the operands in the subtraction and comparison instructions: **Acc is the value that is subtracted** (this order is similar to that in 8-bit PIC microcontrollers).

Bits 8 and 9 select the addressing mode for the instruction:

INDX,NLIT	Addressing mode	
x0	Lit	Inmediate / Literal
01	Dir	Direct
11	inX	Indexed

2.2 Assembler language syntax

A line of code has the following parts:

```
label:    MNEMONIC  operand  ; comments
```

The label and comment fields are optional. The operand field is not present for the NOP and TAX instructions, but for all other instructions must always be present. And its syntax depends on the addressing mode used.

Examples:

```
LDA  65         ; Immediate, decimal constant
LDA  0x41       ; Immediate, hexadecimal constant
LDA  'A'        ; Immediate, ASCII code
LDA  (0xB+2)*5  ; Immediate, expression
LDA  [0xE];     ; Direct, hexadecimal address
LDA  [(8+6)]    ; Direct, address in an expression
LDA  [X]        ; Indexed
```

3 Programming tricks

Let’s look at some program examples:

- Data stacks:

```
sp = 0x1F     ; memory location used as stack pointer
    ; push
    DECX [sp] ; decrement stack pointer and leave copy in X
    STA  [X]

    ; pop
    LDX  [sp]
    LDA  [X]
    INC  [sp]
```

- Subrutines

```
    ; Subrutine call
    LDA  .+2  ; Return address 2 instructions ahead
    JMP  rutine
    ; returns here
    ...
rutine:
    DECX [sp] ; decrement stack pointer and leave a copy in X
    STA  [X]  ; return address stored on stack
    ...
    LDX  [sp]
    INC  [sp]
    JMP  [X]  ; return to main program
```

- Setting the carry flag

```
    ADDA 0 ; C = 0
    ...
    LDA  255
    ADDA 1 ; C = 1
```

- Comparisons

```
CMP op compares (subtracts) Acc from 'op'. The value of the flags will be:
    C == 1  if op >= Acc (unsigned values)
    Z == 1  if op == Acc
    N == 1  if op <  Acc (unsigned values)

  TST op is a logical comparison (AND). The Z flag will be:
    Z == 1  if there is no bit set to 1 simultaneously in Acc and 'op'
```

- Shifts and rotations to the left

```
    LDA  [variable]   ; shl(variable) = variable*2
    ADDM [variable]   ; C = variable7

    LDA  [variable]   ; rol(variable) = variable*2 + C
    ADCM [variable]   ; C = variable7
```

- Shift right (unsigned)

```
    ADDA 0          ; Sets C = 0
    ROR [variable]  ; variable = variable / 2
```

- Shift right (signed)

```
    LDA  0x80
    ADDA [variable]  ; C = variable7
    ROR  [variable]  ; variable = variable / 2
```

- memset

```
    LDA  dest-1      ; address for writting (-1)
    STA  [pointer]
    LDA  nbytes      ; number of data to write
    STA  [counter]
    LDA  value       ; fill value
et1:
    INCX [pointer]   ; X = ++pointer
    STA  [X]         ; *X = value
    DEC  [counter]   ; --counter
    JNZ  et1
```

4 “Pipelining” (synchronous program memory)

If the program ROM memory is synchronous and has the same active edge in its clock as the CPU, what will happens is that all instructions are executed with one cycle delay. This only poses a problem for jump instructions because when you jump, the instruction that was after the jump has already been loaded into the ROM output register, and if nothing is done about it it will be executed.

This has been prevented by having the current instruction executed as a NOP if the previous instruction was a taken jump. In this way, the unconditional jump lasts two effective clock cycles, and the conditional jumps lasts two cycles when taken or only one if not taken.

Although the delayed jumps have also been preserved. They have been coded the same way as the normal jumps but with bit #10 set to 1. In these jumps, the instruction that follows them is not invalidated and therefore will always be executed, also for the conditional jumps. It is as if the jump was executed one instruction later than it should be according to its position in the program, and that is why we call it delayed. We can take advantage of this behavior by placing useful instructions after the jumps. As an example we have the return of a subroutine:

```
rutine:
    DECX [sp]
    STA  [X]
    ...
    LDX  [sp]
    JMPD [X]  ; returns to main program
    INC  [sp] ; Executed after a delayed jump
```

Here we take advantage of the instruction that follows the jump (in bold) to adjust the stack pointer upon return and thus save a clock cycle in the execution.

A more sophisticated example of the use of delayed jumps is in a subroutine to print text strings from program memory:

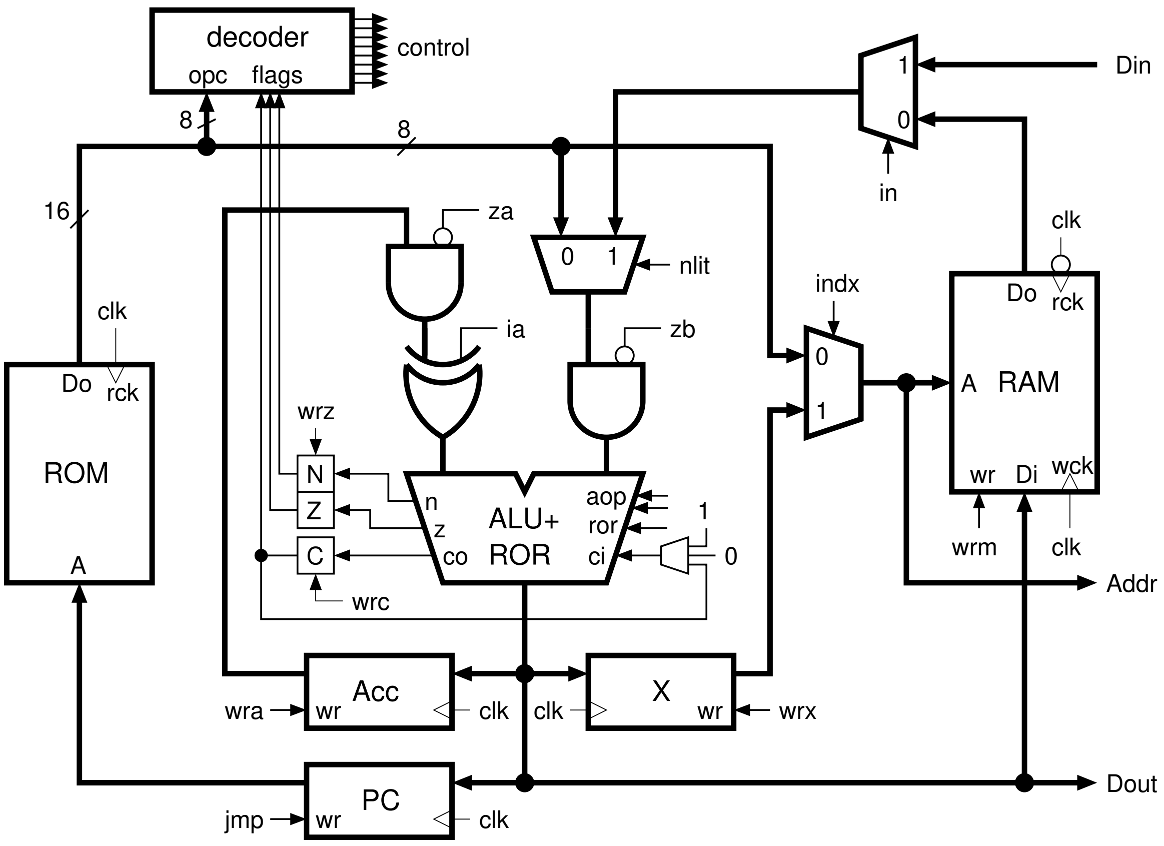
```
; Print string from program memory
; par1 : pointer to the beginning of the ASCIIZ string
; trick learned from the GIGATRON computer
pputs:  decx    [sp]
        sta     [x]
pp1:    jmpd    [par1] ; Jump to table and run LDA n
        jmpd    .+1    ; But immediately jump back (to JZD)
        jzd     pp2
        inc     [par1] ; pointer++
        jmpd    pp1
        out     [0]    ; data to terminal
pp2:    ldx     [sp]   ; subroutine return
        jmpd    [x]
        inc     [sp]
; text strings stored in program memory
txt:    lda     'H'
        lda     'e'
        lda     'l'
        lda     'l'
        lda     'o'
        lda      0
```

Notice that in this example all the instructions that follow the jumps (in bold) are useful instructions, and that by placing two consecutive jumps in the code we can execute a single LDA instruction from the text string (This trick is used in the GIGATRON computer firmware). This way of managing constant tables is only possible if jumps are of the delayed type. I must confess this has been the real motivation I had for including delayed jumps in the instruction set.

5 Hardware

5.1 Blocks

The block diagram of the BAC computer follows this figure:



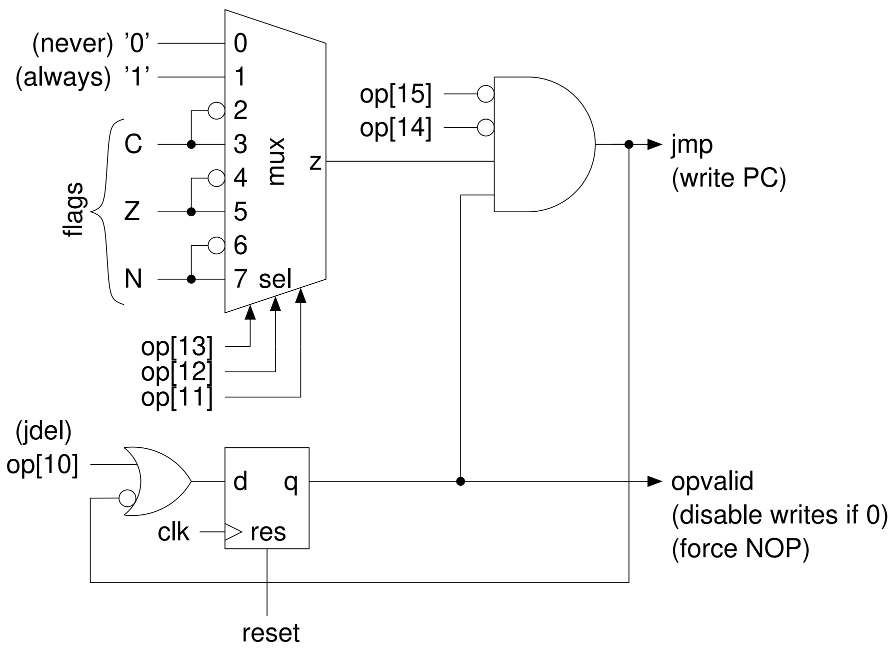
In the center of the diagram we have an ALU capable of performing addition, AND, OR, and exclusive OR operations, to which a multiplexer has been added to perform rotations to the right. One of the ALU operands comes from the Acc register, although you can invert its bits (for subtractions), force a value of zero (for LDA, LDX, IN, INCs, and jumps), or a value of 0xFF (for DECs). In the other operand we can also force a value of zero, which will allow us to output the value of Acc without modifying it (for STA, OUT, and TAX). A pair of multiplexers allow us to route to this operand the 8 least significant bits of the instruction, the output of the data RAM, or an external data (IN instruction).

X and PC are pointer registers that contain memory addresses, either for data (X) or program (PC). The latter is also a counter register that is incremented every clock cycle, except when written during jump instructions.

The ALU also provides us with the values that we must write to the flag registers: Carry, C, zero, Z, and negative, N. These flags are written when executing some of the instructions, but not all. And there is actually a fourth flag, 'opvalid', not shown in the figure, which does not come from the ALU but from the jump logic, and which indicates whether the current instruction should be executed or not. These flags, along with the 8 MSB bits of the instruction, read from the program ROM, are used in the "decoder" block to generate the signals that control each of the individual blocks of the core. "decoder" has two parts: the logic for executing jumps, and a combinational block similar to a PLA for generating the other control signals.

5.2 Conditional execution (jumps)

Jumps are the only instructions that can be executed or not depending on the value of the flags. When executed, they write a new value to the PC register, and in the case of normal (non-delayed) jumps, they invalidate the next instruction, preventing it from being executed. The logic for executing the jumps is shown in the following figure:

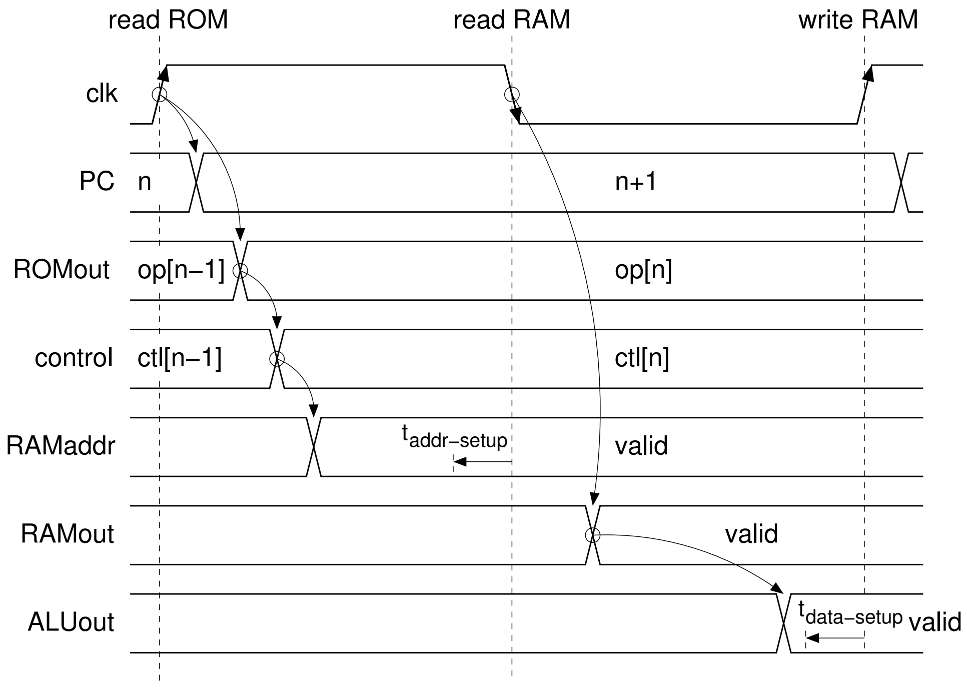


Here I resorted to an 8-input multiplexer to route the logical value corresponding to the selected condition to a common signal, that is validated with the two MSB bits of the operation code and with the output of the 'opvalid' flag, to generate a control signal, 'jmp', that enables a write to the PC register. If the jump wasn't of the delayed type, its opcode will have its bit #10 set to 0, and therefore the 'opvalid' flag will be loaded with 0, so, the next instruction is invalidated. In the case of delayed jumps 'op[10]' is 1, 'opvalid' will remain at 1, and the instruction following the jump will be executed.

When 'opvalid' is 0, all write signals are invalidated, not just 'jmp', and the 'in' and 'out' outputs are also inhibited. When this happens, the instruction doesn't modify anything and is equivalent to a NOP.

The 'reset' signal also sets the 'opvalid' flag to 0, since we do not know what op-code we could have in the ROM output register when starting the system.

5.3 Timing



An important design constraint has been the use of the FPGA's BRAM blocks for system memories, both program and data. In the case of program memory we already have taken into account its synchronous nature, which gives rise to 'pipelining' and delayed jumps, but we also have to consider it for data memory. If we want to read and write a memory location during a single clock cycle we have no choice but to do the reading on a different edge than the writing. This is possible because BRAMs have separate clock signals for reading and writing, so one clock can be the complement of the other.

This results in the timing diagram shown in the above figure, where the program ROM is read on the rising edges of the clock while the data RAM is read on the falling edges but written on the rising edges. In order to have everything operating correctly, the signals have to be valid a certain time before the corresponding clock edge: the address of the data RAM a time $t_{addr-setup}$ before the falling edge of the reading, and the output of the ALU another time $t_{data-setup}$ before the rising edge of the write. The longest delay would be for instructions that read and write to RAM, with direct or indexed addressing, and with a carry propagating within the ALU adder. For example: `INC [pos]` when the memory location 'pos' contains the data 0xff. The sum of all the propagation delays plus the write setup time would give us the minimum duration for the half cycle with the clock low. When the clock is high, the delays are limited to the ROM's own propagation delay plus that of the address multiplexer for the RAM, so we will surely have a lot of idle time left if the clock wave is more or less square. From this consideration, it may be advisable to use a rectangular clock wave with less time high than low.

6 Results

The designed core is in a Verilog module with the following interface signals:

- 'clk'. Clock input
- 'reset'. Reset input, active high and asynchronous.
- '[7:0]din'. Input bus for peripherals.
- '[7:0]addr'. Address output for peripherals.
- '[7:0]dout'. Output bus for peripherals.
- 'out'. Writing pulse in peripherals.
- 'in'. Reading pulse in peripherals.

This module has been instantiated in two test designs. One of them simply routed each signal from the module to an I/O pin of the FPGA. It did not include any peripherals, and had been designed as a mechanism to measure the resources that the micro actually occupies in the FPGA.

The other is a practical system that includes as peripherals a simple UART with a predefined baud rate and a 3-bit register that is displayed on LEDs, as well as a PLL that allows us to test different clock frequencies. In this design it has been possible to run the "Hello World" program.

The results are:

	Conditions	Core only	Core+UART+LEDs
Logic cells		162	262
BRAMs		2	2
Maximum clock frequency (nextpnr)		59.01 MHz	43.78 MHz
Maximum clock frequency (tested)	50% clk duty	-	65 MHz
Maximum clock frequency (tested)	25% clk duty	-	85 MHz

In conclusion, compared to other 8-bit micros this design has turned out to be really tiny (6502: 673 LCs, Z80: 2247 LCs), and although logically it has limitations, especially in relation to the size of the program memory, it can still have applications of interest.

7 The 64K BAC

The weak point about the BAC processor is the small address space it has, both for the data and program memories. So, something had to be done in order to increase its addressing capabilities. In the case of data memory it will be desirable to have at least 512 bytes because this is the size of a single BRAM block. And for program memory 256 instructions are easily too few if programs are a bit complex. In both cases we need some storage for the extra address bits, and a new write-only register, PG, was included along with an instruction to load data into it:

Instruction	Flags	Mnemonic	Description
0100.11	-,-,-	LDPG	PG = op

The PG register has a maximum of 8 bits, but it could have less. For instance, for 512 bytes of data memory PG only has its bit #0 physically implemented. The new BAC03 core has two parameters, ROMSIZE and RAMSIZE, both defaulting to 256. If one of these parameters is higher than 256 the PG register and its related logic is synthesized.

7.1 More than 256 bytes of data memory

For data memory PG can be considered to be the high byte of the index register, X. Therefore, the indexed addressing mode now uses the registers PG and X to point to a data memory position in a memory space up to 64KB in size.

But the direct addressing mode only has the low 8 bits of the address encoded in the instructions, and only the first 256 bytes of the data RAM can be addressed in this way. This resembles the "zero-page" mode of the 6800 and 6502 CPUs. Let's present some examples:

```
; Setting [0x1234] to 0xFF
    lda  0xff
    ldpg >0x1234  ; high byte (0x12)
    ldx  <0x1234  ; low byte  (0x34)
    sta  [x]      ; data_mem[PG:X] = Acc
; Fill 512 bytes of memory form 0x100 to 0x2FF
; ptr, ptr+1, are in Zero Page
    lda  0
    sta  [ptr]
    lda  2
    sta  [ptr+1]
    ldpg [ptr+1]
    lda  fill_value
l1: decx [ptr]
    jnzd l1
    sta  [x]         ; dmem[PG:X]=Acc
    dec  [ptr+1]
    jnzd l1
    ldpg [ptr+1]
```

7.2 More than 256 instructions in program memory

This requires a PC register with more than 8 bits, easy? No, because we only have 8 bits for the destination address of jumps. So, the PC is divided into two sections: The lower 8 bits are loaded in the case of jumps while the high bits are left unmodified. This limits the jumps to addresses in the same 256 instruction page as we are when executing the jump (and remember: jumps are executed with one cycle delay, so, a jump instruction located at address 0x0FF will have the PC as 0x100 when jumping).

It would be desirable to have also “long” jumps in order to jump to code in a different page, but, I didn’t want to include another 16 possible jumps into the instruction set, so, I resorted to a little dirty trick here:

- If the jump is preceded by an ’LDPG” instruction the high bits of the PC are copied from the PG register. Otherwise they are left unmodified.

I know this will result in a lot of complications if some day the core is modified to support interrupts, but for now the ’long’ jumps only require an extra flip-flop to remember the write to PG register during the previous cycle, a very small amount of logic.

As an example lets present how to code a call to a “far” subroutine:

```
; Far Call
         lda  >(.+5)     ; return page (MSBs)
         sta  [rpg]
         lda  <(.+3)     ; return address (LSBs)
         ldpg  >routine  ; Page of the subroutine
         jmp   routine   ; long jump (after ldpg)
         ; return here
         ....
routine: ldpg 0          ; Stack on page 0
         decx [sp]
         sta  [x]        ; Save return address
         decx [sp]
         lda  [rpg]      ; Save return page
         sta  [x]
         ...
lret:    ldx  [sp]       ; restore return page
         lda  [x]
         sta  [rpg]
         incx [sp]
         ldpg [rpg]
         jmpd [x]        ; long jump (after ldpg)
         inc  [sp]
```

Of course, much of this mess can be avoided if subroutines are located on the same page as the code, or at least if they don’t call other nested subroutines and there is no need to use the stack.

This “more than 256 instructions” mod, implied some work also on the assembler tool. Now, the assembler will display a warning message if a jump instruction changes pages and isn’t preceded by an LDPG instruction.

I must recognize these address space extensions look crappy and that a 64K space should have been considered from the beginning, resulting in a quite different processor architecture (maybe in a 6502 clone ;) But there are also crappy commercial examples around. Take for instance the data memory addressing of the Intel 8052, or the memory banking of the Microchip PICs.

8 Tools

8.1 Assembler

An specific assembler program has been written for this micro, bac02asm, which has the following command line:

```
bac02asm [options] sourcefile.asm
options:
-o outputfile.hex Name of the output file. Default is "out.hex"
-l listfile.lst   Name of the list file. Default is "out.lst"
-s statfile.lst   Name of the instruction statistics file
```

Regarding the assembler source file, it may contain the following elements:

- Simbols / labels:

Labels start at the first text column and must end with ':' or '='. In the first case the current address is assigned to the label while in the second the result of an expression is explicitly assigned. Examples:

```
        ORG 0x20
label1: LDA <const16 ; label1 becomes 0x20
label2= 0x14         ; label2 becomes 0x14
```

- Directives. They are the following:

```
INCLUDE "file"    Includes the indicated file into the source code.
ORG <expression>  Sets the address counter to the indicated value.
WORD <expression> Directly generates a word in the code.
```

- Expressions.

  Expressions include numerical constants, predefined variables, labels, and their possible combinations through arithmetic and logical operations.

  - Numerical constants:

    ```
    65   Constant expressed in decimal base
    0x41 The same constant in hexadecimal base
    'A'  ASCII code value of the uppercase letter A.
    ```

  - Predefined variables:

    ```
    . (dot) It is the current value of the address counter.
            Usage example: "    JMP . ; never ending loop"
    ```

  - Unary operators:

    ```
    -value The sign of expression "value" is changed (two's complement).
    ~value The bits of the value expression are inverted.
    <value The 8 least significant bits of the value expression.
    >value The 8 most significant bits of the value expression.
    ```

- Binary operators:

```
a+b  The sum of the expressions a and b
a-b  The subtraction of a and b
a*b  The product of a and b
a/b  The integer division of a by b
a%b  The remainder (module) of the integer division of a by b
a&b  The logical AND of the bits of a and b
a|b  The logical OR of the bits of a and b
a^b  The exclusive-OR (XOR) function of the bits of a and b
a>>b a shifted b bits to the right. Zeros are shifted into MSBs
a<<b a shifted b bits to the left. Zeros are shifted into LSB
```

- Parentheses: Specify the order of operations. Examples:

```
pos= base+(resvd+index)*2
divider= (FCLK/16+BAUD/2)/BAUD-1
    ANDA (~((1<<ENAB)|(1<<PWON)))&0xf
    org  (.+0x0f)&0xf0 ; align to multiple of 16
```

• Comments: They are all the text that follows ';' in each line.

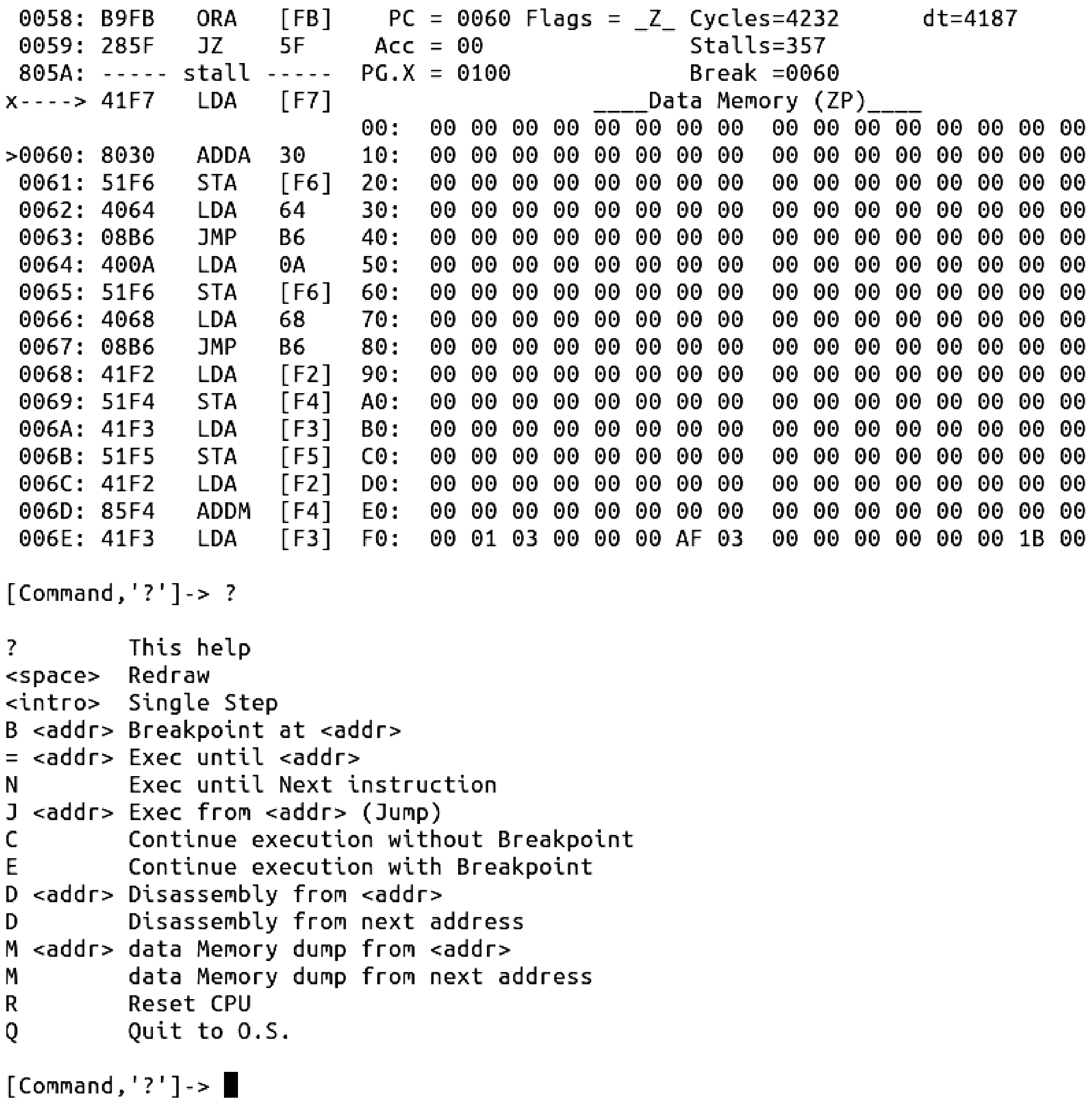
Output file format.

The output file is a sequence of segments with the contents of memory that can be imported from Verilog using the command `$readmemh("file",memory)`. Includes address markers of the type `"@hex"` followed by the data in hexadecimal. Example with a segment of 3 data at address 0 and 2 data at address 16:

```
@0000
40FF
4DFF
4D05
@0010
0818
E505
```

8.2 Emulator

The BAC computer can be simulated using an emulator program, bac02emu, which shows us the contents of the registers and data memory interactively while we execute the code. Let's see a screenshot:



```
 0058: B9FB   ORA   [FB]    PC = 0060 Flags = _Z_ Cycles=4232      dt=4187
 0059: 285F   JZ    5F     Acc = 00               Stalls=357
 805A: ----- stall -----  PG.X = 0100             Break =0060
x----> 41F7   LDA   [F7]                  ____Data Memory (ZP)____
                          00:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
>0060: 8030   ADDA  30    10:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 0061: 51F6   STA   [F6]  20:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 0062: 4064   LDA   64    30:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 0063: 08B6   JMP   B6    40:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 0064: 400A   LDA   0A    50:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 0065: 51F6   STA   [F6]  60:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 0066: 4068   LDA   68    70:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 0067: 08B6   JMP   B6    80:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 0068: 41F2   LDA   [F2]  90:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 0069: 51F4   STA   [F4]  A0:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 006A: 41F3   LDA   [F3]  B0:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 006B: 51F5   STA   [F5]  C0:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 006C: 41F2   LDA   [F2]  D0:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 006D: 85F4   ADDM  [F4]  E0:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 006E: 41F3   LDA   [F3]  F0:  00 01 03 00 00 00 AF 03  00 00 00 00 00 00 1B 00

[Command,'?']-> ?

?        This help
<space>  Redraw
<intro>  Single Step
B <addr> Breakpoint at <addr>
= <addr> Exec until <addr>
N        Exec until Next instruction
J <addr> Exec from <addr> (Jump)
C        Continue execution without Breakpoint
E        Continue execution with Breakpoint
D <addr> Disassembly from <addr>
D        Disassembly from next address
M <addr> data Memory dump from <addr>
M        data Memory dump from next address
R        Reset CPU
Q        Quit to O.S.

[Command,'?']-> █
```

The emulator must be run from a terminal window capable of interpreting ANSI escapes. In the capture of this example, the `LDA [F7]` instruction, that was read from the program ROM during the previous cycle, will be executed. Above it are shown the instructions executed in the 3 previous cycles, the immediately preceding one being a `--- stall ---` (nop), due to the `JZ 5F` jump. Below we have the code disassembled from the address pointed by PC. Above, further to the right, we have the value of the registers (PC, Acc, PG, and X), the active flags (Zero), the simulated time since the reset, and the time since the last stop (1 cycle when executing step by step). Below we have the 256 bytes of the zero page of the data memory.

The emulator simulates a system that, in addition to the core, and full 64K program and data memories, includes an UART. Transmitted data is displayed on <stdout>, while text typed on <stdin> is interpreted as received data (as long as we are not executing the code step by step). The UART registers are those of the following addresses:

1. Data to be transmitted on writes. Data received on reads. A read also clears the receiver flags: DV and OV.
2. Flags. Read only. The bits in this register are:
   - Bit 0: DV. Valid data on reception.
   - Bit 1: FE. Format error (stop bit was 1). In the emulator this bit is always 0.
   - Bit 2: OV. Overrun. A data was received when DV was 1.
   - Bits 3 to 6. Not used.
   - Bit 7. TRDY. Transmitter ready if 1.

The TRDY bit is disabled when transmitting data for 352 cycles, which is equivalent to having two stop bits and a divisor of 32 in the simulated UART. When the code is executed without stopping at each instruction we can see on the screen the data that the simulated micro sends to the UART, in this example the text “1609 Hello World”. Here the micro executes code until reaching the “breakpoint” at address 0x11:

```
[Command,'?']-> =11
<Running... press <ctrl>-c to stop>
1609 Hello World
<press <enter> to resume>
```

9 Verilog sources (Original 0.25K BAC)

```
///////////////////////////////////////////
//          BAC-02 computer by
//            J. Arias (2023)
// Public domain source.
///////////////////////////////////////////

//`define ROBUSTDEC

module bac_computer (
    input clk,          // Reloj
    input reset,        // Reset, asíncrono, activo en alto
    input [7:0]din,     // Datos desde periféricos
    output [7:0]addr,   // Dirección de periféricos
    output [7:0]dout,   // Datos hacia periféricos
    output out,         // Pulso de escritura en periféricos
    output in           // Pulso de lectura de periféricos
);

///////////////////////////////////////////
// ROM de programa en BRAM
///////////////////////////////////////////
reg [15:0]ROM[0:255];
initial $readmemh("ROM.hex",ROM);

reg [15:0]romout;
always @(posedge clk) romout<=ROM[pc];

/////////////////////////////////////////////////
// RAM de datos
//  lectura   en bajada de clk
//  escritura en subida de clk
// (así se parece a una RAM con lectura asíncrona)
/////////////////////////////////////////////////
reg [7:0]RAM[0:255];
reg [7:0]ramout;
always @(negedge clk) ramout <= RAM[aram];
always @(posedge clk) if (wrm)  RAM[aram] <=aluout;

///////////////////////////////////////////
// registros
///////////////////////////////////////////

reg [7:0]pc;
reg [7:0]acc;
reg [7:0]xreg;
reg C,Z,N;          // flags

always @(posedge clk or posedge reset)
    if (reset) pc<=0; else pc <= (jmp) ? aluout : pc + 1;
always @(posedge clk) if (wra) acc<=aluout;
always @(posedge clk) if (wrx) xreg<=aluout;
always @(posedge clk) if (wrc) C <= ror ? alub[0] : co;
always @(posedge clk) if (wrz) {N,Z}<={aluout[7],(aluout==0)};

///////////////////////////////////////////
// data mux
///////////////////////////////////////////
wire nlit = romout[8];
wire indx = romout[9];

wire [7:0]alub = zb ? 0 : (nlit ? (in ? din : ramout) : romout[7:0]);
wire [7:0]alua = (za ? 0 : acc)^(ia ? 8'hff : 0);
wire [7:0]aram = indx ? xreg : romout[7:0];
```

```
///////////////////////////////////////////
// ALU
///////////////////////////////////////////
reg [7:0]aluo;       // combinacional (salida intermedia)
reg co;              // combinacional (salida de acarreo)
// códigos de operación de la ALU
parameter SUM=2'd1;  // cualquier permutación de los códigos sirve
parameter AND=2'd0;  // esta es la que resulta en menos celdas lógicas
parameter OR =2'd2;
parameter XOR=2'd3;

always @*
    case (aop)
        SUM: {co,aluo} <= alua + alub + ci;
        AND: {co,aluo} <= {1'bx, alua & alub};
        OR : {co,aluo} <= {1'bx, alua | alub};
        XOR: {co,aluo} <= {1'bx, alua ^ alub};
    endcase
wire [7:0]aluout = ror ? {C,alub[7:1]} : aluo;

///////////////////////////////////////////////////////////
// condiciones saltos
//     000        001   010 011 100 101     110      111
// nunca (NOP), siempre, NC,  C, NZ,  Z, positivo, negativo
///////////////////////////////////////////////////////////
wire [7:0]jmpcond={N,~N,Z,~Z,C,~C,1'b1,1'b0};
wire jmp = jmpcond[romout[13:11]] & (~romout[14]) & (~romout[15]) &opvalid;

////////////////////////////////////////////////////////////////
// Tras los saltos normales se descarta la siguiente instrucción
////////////////////////////////////////////////////////////////
reg opvalid=0; // Vale 0 después de un salto no retardado
always @(posedge clk or posedge reset)
    if (reset) opvalid<=0; else opvalid<=(~jmp)|romout[10];

////////////////////////////////////////////////////////
// decoder
//  si opvalid==0 se inhiben todas las escrituras (NOP),
//  incluyendo IN y OUT
////////////////////////////////////////////////////////
reg  [10:0]control;    // combinacional
wire [1:0]aop = control[10:9];
wire wrm      = control[8] & opvalid;
wire wra      = control[7] & opvalid;
wire wrx      = control[6] & opvalid;
wire wrc      = control[5] & opvalid;
wire wrz      = control[4] & opvalid;
wire za       = control[3];
wire zb       = control[2];
wire ia       = control[1];
wire ci       = control[0];

// algunas señales simples
wire ror = (romout[15:10]==6'b1010_10);
wire in  = (romout[15:10]==6'b0100_01) & opvalid;
wire out = (romout[15:10]==6'b0101_01) & opvalid;
```

```
// Para hacer más legible la tabla
parameter _1_=1'b1;
parameter ___=1'b0;
parameter _x_=1'bx;
parameter _xx=2'bxx;

always @*
    casex (romout[15:10])
                          //  aop  wrm, wra  wrx  wrc  wrz  za   zb   ia   ci
        6'b00xx_xx: control<={ OR, ___, ___, ___, ___, ___, _1_, ___, ___, _x_}; // JMPs

        6'b0100_00: control<={ OR, ___, _1_, ___, ___, _1_, _1_, ___, ___, _x_}; // LDA
        6'b0100_01: control<={ OR, ___, _1_, ___, ___, _1_, _1_, ___, ___, _x_}; // IN
        6'b0100_10: control<={ OR, ___, ___, _1_, ___, ___, _1_, ___, ___, _x_}; // LDX
        6'b0101_00: control<={ OR, _1_, ___, ___, ___, ___, ___, _1_, ___, _x_}; // STA
        6'b0101_01: control<={ OR, ___, ___, ___, ___, ___, ___, _1_, ___, _x_}; // OUT
        6'b0101_10: control<={ OR, ___, ___, _1_, ___, ___, ___, _1_, ___, _x_}; // TAX

        6'b1000_00: control<={SUM, ___, _1_, ___, _1_, _1_, ___, ___, ___, ___}; // ADDA
        6'b1000_01: control<={SUM, _1_, ___, ___, _1_, _1_, ___, ___, ___, ___}; // ADDM
        6'b1000_10: control<={SUM, ___, _1_, ___, _1_, _1_, ___, ___, ___,  C }; // ADCA
        6'b1000_11: control<={SUM, _1_, ___, ___, _1_, _1_, ___, ___, ___,  C }; // ADCM
        6'b1001_00: control<={SUM, ___, _1_, ___, _1_, _1_, ___, ___, _1_, _1_}; // SUBA
        6'b1001_01: control<={SUM, _1_, ___, ___, _1_, _1_, ___, ___, _1_, _1_}; // SUBM
        6'b1001_10: control<={SUM, ___, _1_, ___, _1_, _1_, ___, ___, _1_,  C }; // SBCA
        6'b1001_11: control<={SUM, _1_, ___, ___, _1_, _1_, ___, ___, _1_,  C }; // SBCM

        6'b1010_00: control<={SUM, ___, ___, ___, _1_, _1_, ___, ___, _1_, _1_}; // CMP
        6'b1010_01: control<={AND, ___, ___, ___, ___, _1_, ___, ___, ___, _x_}; // TST
        6'b1010_10: control<={_xx, _1_, ___, ___, _1_, _1_, _x_, ___, _x_, _x_}; // ROR

        6'b1011_00: control<={AND, ___, _1_, ___, ___, _1_, ___, ___, ___, _x_}; // ANDA
        6'b1011_01: control<={AND, _1_, ___, ___, ___, _1_, ___, ___, ___, _x_}; // ANDM
        6'b1011_10: control<={ OR, ___, _1_, ___, ___, _1_, ___, ___, ___, _x_}; // ORA
        6'b1011_11: control<={ OR, _1_, ___, ___, ___, _1_, ___, ___, ___, _x_}; // ORM
        6'b1100_00: control<={XOR, ___, _1_, ___, ___, _1_, ___, ___, ___, _x_}; // XORA
        6'b1100_01: control<={XOR, _1_, ___, ___, ___, _1_, ___, ___, ___, _x_}; // XORM

        6'b1101_00: control<={SUM, _1_, ___, ___, ___, _1_, _1_, ___, ___, _1_}; // INC
        6'b1101_01: control<={SUM, _1_, _1_, ___, ___, _1_, _1_, ___, ___, _1_}; // INCA
        6'b1101_10: control<={SUM, _1_, ___, _1_, ___, _1_, _1_, ___, ___, _1_}; // INCX
        6'b1101_11: control<={SUM, _1_, _1_, _1_, ___, _1_, _1_, ___, ___, _1_}; // INCAX

        6'b1110_00: control<={SUM, _1_, ___, ___, ___, _1_, _1_, ___, _1_, ___}; // DEC
        6'b1110_01: control<={SUM, _1_, _1_, ___, ___, _1_, _1_, ___, _1_, ___}; // DECA
        6'b1110_10: control<={SUM, _1_, ___, _1_, ___, _1_, _1_, ___, _1_, ___}; // DECX
        6'b1110_11: control<={SUM, _1_, _1_, _1_, ___, _1_, _1_, ___, _1_, ___}; // DECAX
`ifdef ROBUSTDEC
        default:    control<={_xx, ___, ___, ___, ___, ___, _x_, _x_, _x_, _x_}; // NOP
`else
        default:    control<={_xx, _x_, _x_, _x_, _x_, _x_, _x_, _x_, _x_, _x_}; // ???
`endif
    endcase

///////////////////////////////////////////
// I/O
///////////////////////////////////////////
assign dout = aluout;
assign addr = aram;

endmodule
```

10 Verilog sources (64K BAC)

```
///////////////////////////////////////////
//         BAC-03 computer by
//           J. Arias (2023)
// Public domain source.
///////////////////////////////////////////

//`define ROBUSTDEC

module bac_computer (
    input clk,        // Reloj
    input reset,      // Reset, asíncrono, activo en alto
    input [7:0]din,   // Datos desde periféricos
    output [7:0]addr, // Dirección de periféricos
    output [7:0]dout, // Datos hacia periféricos
    output out,       // Pulso de escritura en periféricos
    output in         // Pulso de lectura de periféricos
);
parameter ROMSIZE=256;
parameter RAMSIZE=512;
localparam PAW=$clog2(ROMSIZE-1);
localparam DAW=$clog2(RAMSIZE-1);
localparam PGW=(PAW > DAW)? PAW-8 : DAW-8;

///////////////////////////////////////////
// ROM de programa en BRAM
///////////////////////////////////////////
reg [15:0]ROM[0:(ROMSIZE-1)];
initial $readmemh("ROM.hex",ROM);

reg [15:0]romout;
always @(posedge clk) romout<=ROM[pc];

/////////////////////////////////////////////////
// RAM de datos
//  lectura   en bajada de clk
//  escritura en subida de clk
// (así se parece a una RAM con lectura asíncrona)
/////////////////////////////////////////////////
reg [7:0]RAM[0:(RAMSIZE-1)];
reg [7:0]ramout;
always @(negedge clk) ramout <= RAM[aram];
always @(posedge clk) if (wrm)  RAM[aram] <=aluout;

///////////////////////////////////////////
// registros
///////////////////////////////////////////

reg [PAW-1:0]pc;
reg [7:0]acc;
reg [7:0]xreg;
reg [PGW-1:0]pg;
reg C,Z,N;        // flags
reg ljmp=0;
if (PAW>8) begin
    always @(posedge clk) ljmp<=wrpg;
end
always @(posedge clk or posedge reset)
    if (reset) pc<=0; else begin
        pc[7:0] <= (jmp) ? aluout : pc[7:0] + 1;
        if (PAW>8)
            pc[PAW-1:8] <= (jmp & ljmp) ? pg[PAW-9:0] : pc[PAW-1:8] + ((&pc[7:0])&(~jmp));
    end

always @(posedge clk) if (wra) acc<=aluout;
always @(posedge clk) if (wrx) xreg<=aluout;
always @(posedge clk) if (wrpg) pg<=aluout[PGW-1:0];
always @(posedge clk) if (wrc) C <= ror ? alub[0] : co;
always @(posedge clk) if (wrz) {N,Z}<={aluout[7],(aluout==0)};
```

```verilog
///////////////////////////////////////////
// data mux
///////////////////////////////////////////
wire nlit = romout[8];
wire indx = romout[9];

wire [7:0]alub = zb ? 0 : (nlit ? (in ? din : ramout) : romout[7:0]);
wire [7:0]alua = (za ? 0 : acc)^(ia ? 8'hff : 0);

wire [(DAW-1):0]aram = (DAW>8)? (indx ? {pg,xreg} : { {(DAW-8){1'b0}},romout[7:0]}):
    (indx ? xreg : romout[7:0]);

///////////////////////////////////////////
// ALU
///////////////////////////////////////////
reg [7:0]aluo;       // combinacional (salida intermedia)
reg co;              // combinacional (salida de acarreo)
// códigos de operación de la ALU
parameter SUM=2'd1;  // cualquier permutación de los códigos sirve
parameter AND=2'd0;  // esta es la que resulta en menos celdas lógicas
parameter OR =2'd2;
parameter XOR=2'd3;

always @*
    case (aop)
        SUM: {co,aluo} <= alua + alub + ci;
        AND: {co,aluo} <= {1'bx, alua & alub};
        OR : {co,aluo} <= {1'bx, alua | alub};
        XOR: {co,aluo} <= {1'bx, alua ^ alub};
    endcase
wire [7:0]aluout = ror ? {C,alub[7:1]} : aluo;

///////////////////////////////////////////////////////////
// condiciones saltos
//     000        001   010 011 100 101     110      111
// nunca (NOP), siempre, NC,  C, NZ,  Z, positivo, negativo
///////////////////////////////////////////////////////////
wire [7:0]jmpcond={N,~N,Z,~Z,C,~C,1'b1,1'b0};
wire jmp = jmpcond[romout[13:11]] & (~romout[14]) & (~romout[15]) &opvalid;

////////////////////////////////////////////////////////////////
// Tras los saltos normales se descarta la siguiente instrucción
////////////////////////////////////////////////////////////////
reg opvalid=0; // Vale 0 después de un salto no retardado
always @(posedge clk or posedge reset)
    if (reset) opvalid<=0; else opvalid<=(~jmp)|romout[10];

/////////////////////////////////////////////////////////
// decoder
//  si opvalid==0 se inhiben todas las escrituras (NOP),
//  incluyendo IN y OUT
/////////////////////////////////////////////////////////
reg  [10:0]control;    // combinacional
wire [1:0]aop = control[10:9];
wire wrm      = control[8] & opvalid;
wire wra      = control[7] & opvalid;
wire wrx      = control[6] & opvalid;
wire wrc      = control[5] & opvalid;
wire wrz      = control[4] & opvalid;
wire za       = control[3];
wire zb       = control[2];
wire ia       = control[1];
wire ci       = control[0];

// algunas señales simples
wire ror = (romout[15:10]==6'b1010_10);
wire in  = (romout[15:10]==6'b0100_01) & opvalid;
wire wrpg= ((DAW>8)||(PAW>8))? (romout[15:10]==6'b0100_11) & opvalid : 0;
wire out = (romout[15:10]==6'b0101_01) & opvalid;
```

```
// Para hacer más legible la tabla
parameter _1_=1'b1;
parameter ___=1'b0;
parameter _x_=1'bx;
parameter _xx=2'bxx;
always @*
    casex (romout[15:10])
                          //  aop  wrm, wra  wrx  wrc  wrz  za   zb   ia   ci
        6'b00xx_xx: control<={ OR, ___, ___, ___, ___, ___, _1_, ___, ___, _x_}; // JMPs

        6'b0100_00: control<={ OR, ___, _1_, ___, ___, _1_, _1_, ___, ___, _x_}; // LDA
        6'b0100_01: control<={ OR, ___, _1_, ___, ___, _1_, _1_, ___, ___, _x_}; // IN
        6'b0100_10: control<={ OR, ___, ___, _1_, ___, ___, _1_, ___, ___, _x_}; // LDX

        6'b0100_11: control<=((DAW>8)||(PAW>8))?                                 // LDPG
                             { OR, ___, ___, ___, ___, ___, _1_, ___, ___, _x_}:
`ifdef ROBUSTDEC
                             {_xx, ___, ___, ___, ___, ___, _x_, _x_, _x_, _x_};
`else
                             {_xx, _x_, _x_, _x_, _x_, _x_, _x_, _x_, _x_, _x_};
`endif
        6'b0101_00: control<={ OR, _1_, ___, ___, ___, ___, ___, _1_, ___, _x_}; // STA
        6'b0101_01: control<={ OR, ___, ___, ___, ___, ___, ___, _1_, ___, _x_}; // OUT
        6'b0101_10: control<={ OR, ___, ___, _1_, ___, ___, ___, _1_, ___, _x_}; // TAX

        6'b1000_00: control<={SUM, ___, _1_, ___, _1_, _1_, ___, ___, ___, ___}; // ADDA
        6'b1000_01: control<={SUM, _1_, ___, ___, _1_, _1_, ___, ___, ___, ___}; // ADDM
        6'b1000_10: control<={SUM, ___, _1_, ___, _1_, _1_, ___, ___, ___,  C }; // ADCA
        6'b1000_11: control<={SUM, _1_, ___, ___, _1_, _1_, ___, ___, ___,  C }; // ADCM
        6'b1001_00: control<={SUM, ___, _1_, ___, _1_, _1_, ___, ___, _1_, _1_}; // SUBA
        6'b1001_01: control<={SUM, _1_, ___, ___, _1_, _1_, ___, ___, _1_, _1_}; // SUBM
        6'b1001_10: control<={SUM, ___, _1_, ___, _1_, _1_, ___, ___, _1_,  C }; // SBCA
        6'b1001_11: control<={SUM, _1_, ___, ___, _1_, _1_, ___, ___, _1_,  C }; // SBCM

        6'b1010_00: control<={SUM, ___, ___, ___, _1_, _1_, ___, ___, _1_, _1_}; // CMP
        6'b1010_01: control<={AND, ___, ___, ___, ___, _1_, ___, ___, ___, _x_}; // TST
        6'b1010_10: control<={_xx, _1_, ___, ___, _1_, _1_, _x_, ___, _x_, _x_}; // ROR

        6'b1011_00: control<={AND, ___, _1_, ___, ___, _1_, ___, ___, ___, _x_}; // ANDA
        6'b1011_01: control<={AND, _1_, ___, ___, ___, _1_, ___, ___, ___, _x_}; // ANDM
        6'b1011_10: control<={ OR, ___, _1_, ___, ___, _1_, ___, ___, ___, _x_}; // ORA
        6'b1011_11: control<={ OR, _1_, ___, ___, ___, _1_, ___, ___, ___, _x_}; // ORM
        6'b1100_00: control<={XOR, ___, _1_, ___, ___, _1_, ___, ___, ___, _x_}; // XORA
        6'b1100_01: control<={XOR, _1_, ___, ___, ___, _1_, ___, ___, ___, _x_}; // XORM

        6'b1101_00: control<={SUM, _1_, ___, ___, ___, _1_, _1_, ___, ___, _1_}; // INC
        6'b1101_01: control<={SUM, _1_, _1_, ___, ___, _1_, _1_, ___, ___, _1_}; // INCA
        6'b1101_10: control<={SUM, _1_, ___, _1_, ___, _1_, _1_, ___, ___, _1_}; // INCX
        6'b1101_11: control<={SUM, _1_, _1_, _1_, ___, _1_, _1_, ___, ___, _1_}; // INCAX

        6'b1110_00: control<={SUM, _1_, ___, ___, ___, _1_, _1_, ___, _1_, ___}; // DEC
        6'b1110_01: control<={SUM, _1_, _1_, ___, ___, _1_, _1_, ___, _1_, ___}; // DECA
        6'b1110_10: control<={SUM, _1_, ___, _1_, ___, _1_, _1_, ___, _1_, ___}; // DECX
        6'b1110_11: control<={SUM, _1_, _1_, _1_, ___, _1_, _1_, ___, _1_, ___}; // DECAX
`ifdef ROBUSTDEC
        default:    control<={_xx, ___, ___, ___, ___, ___, _x_, _x_, _x_, _x_}; // NOP
`else
        default:    control<={_xx, _x_, _x_, _x_, _x_, _x_, _x_, _x_, _x_, _x_}; // ???
`endif
    endcase

///////////////////////////////////////////
// I/O
///////////////////////////////////////////
assign dout = aluout;
assign addr = aram;

endmodule
```