The Wayback Machine - http://web.archive.org/web/20160310173708/https://pragprog.com/magazines/2012-03/the-nor-…
Skip to main content

The Pragmatic Bookshelf

- Log in
-
- Books/Videos
  - **Back**
  - Books/Videos
  - Books
  - Video
  - Audio
  - Training
  - Magazine
-

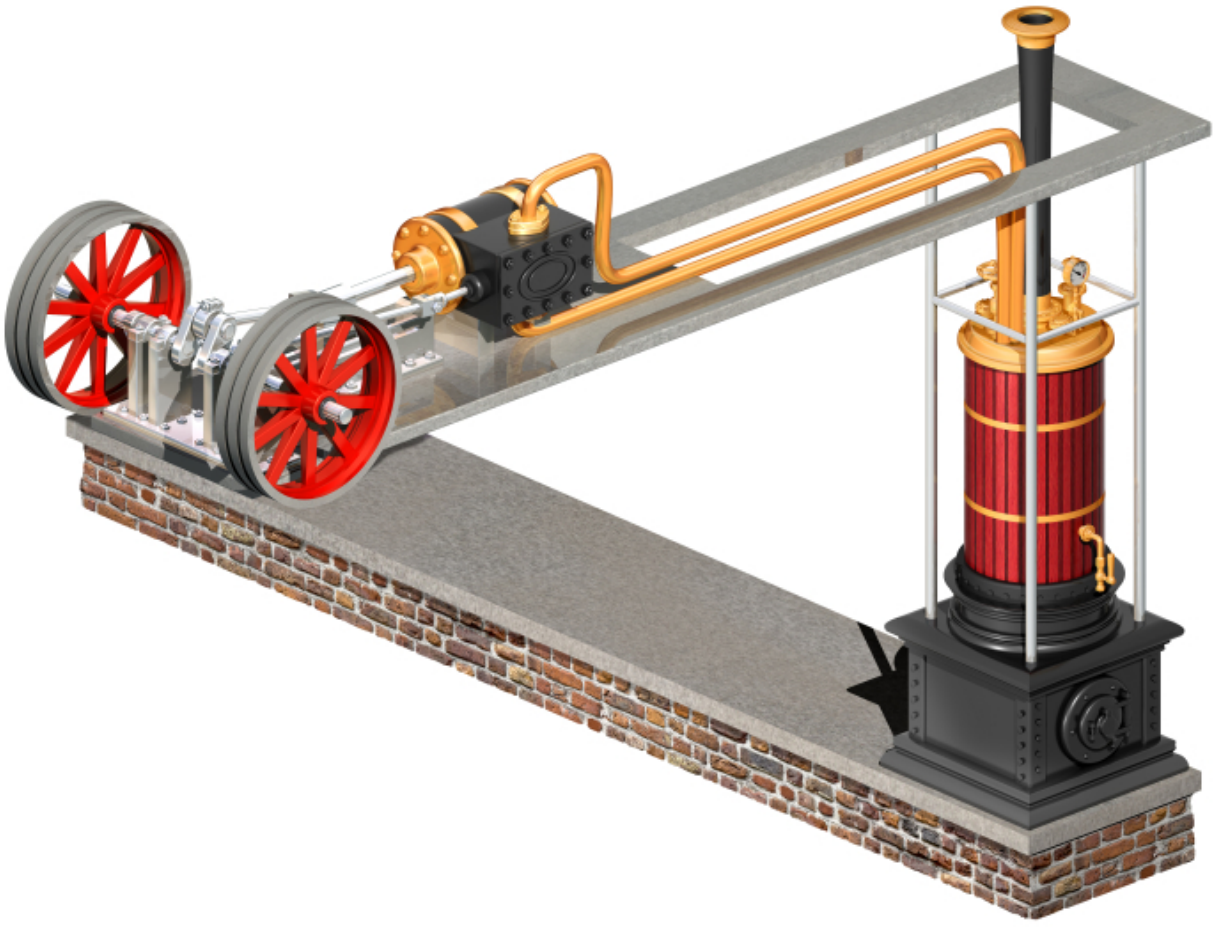
Cart (0)

- Search our catalog...
  Search
- Log in
- Books/Videos
  - Books
  - Video
  - Audio
  - Training
  - Magazine
- Help
  - Forums
  - Resources

The NOR Machine

Build a CPU with Only One Instruction

by Alexander Demin



Build an assembler and an emulator for a single-instruction CPU and implement a non-trivial algorithm on it, using Ruby as a macro DSL to compile it all.

Have you ever developed in an assembly language? Have you developed an assembly language? Ever developed a CPU running your own assembly language?

There may be some who said “yes” to all three questions. But even for those readers, this article may have something new to offer. We are going to:

1. Develop a CPU that performs only one primitive instruction.

2. Write a non-trivial calculation in this CPU's single-instruction assembly language.
3. Use Ruby as a macro domain-specific language to compile it all.

At the end we will have an assembler and an emulator of that single instruction CPU, along with an implementation of a non-trivial algorithm on it, and some Ruby code that looks like no Ruby code you've ever seen.

Why would we want to do this? What, doesn't it sound like fun?

One Function to Build Them All

I picked up an idea years ago in a FIDO discussion group, RU.HACKER. With a minor improvement, that idea will become the heart of our CPU.

The idea is to use just one function to compute everything. It suffices to do this for bits: bytes consist of bits, and if we can compute bits, we also can compute bytes, and everything made up of bytes. That is, everything.

There are sixteen boolean functions of two arguments. Two of them are special: NOR and NAND. The other fourteen functions can be computed using only either NOR or NAND. We'll use NOR.

Not surprisingly, NOR can be expressed via other functions:

```
NOR(a, b) = NOT(OR(a, b))
```

NOR's truth table is as follows:

```
===== ========
Input  Output
===== ========
 A B  A NOR B
----- --------
 0 0     1
 0 1     0
 1 0     0
 1 1     0
===== ========
```

Remember, all other boolean functions of one or two arguments can be computed from NOR. For example:

```
NOT(a) = NOR(a, a)
AND(a, b) = NOT(OR(NOT(a), NOT(b)))
OR(a, b) = NOT(NOR(a, b))
XOR(a, b) = OR(AND(a, NOT(b)), AND(NOT(a), b)))
```

Defining Our CPU's Architecture

So NOR gives us everything we need to compute anything in bits. All right, we have established our single function. Now let's sketch out an overall architecture for our CPU.

We will have a linear memory organization containing 2^16 (65536 or 0xFFFF), cells, and consequently a range of valid addresses from 0 (0x0000) to 65535 (0xFFFF), 16 bits per address. So far this CPU is similar to classical microprocessors like the Intel 8080 or Zilog Z80. But from here on it's going to get really different.

The first significant difference is that memory cells in our CPU will be 16-bit, not 8-bit, so one cell can hold a range of values from 0x0000 to 0xFFFF.

The next difference is more important. Normally, in both RISC and CISC architectures of modern CPUs, each individual instruction consists of two parts: an operation code, or opcode, and some number of arguments. The opcode specifies an operation to perform and the arguments define the details of a particular instruction.

But remember, our CPU understands only one instruction. So we don't need the opcode, because we always compute NOR! Our instructions will only contain the arguments—the arguments of NOR.

Now we have to settle on a binary format for our instructions. Each instruction will reside in three consecutive cells in the memory. The first and second cells will contain addresses of the NOR arguments, a and b respectively. The third argument will contain the address where the result of the NOR will be placed.

Imagine our memory as a linear array mem of 16-bit values:

```
mem[0], mem[1], mem[2], mem[3], ..., mem[65534], mem[65535]
```

Consider an instruction at the address 1. It occupies three cells: 1, 2, and 3. To process this instruction, our CPU has to execute the following:

1. take a value from mem[1] and name it addr_a
2. take a value from mem[addr_a] and name it a
3. take a value from mem[2] and name it addr_b
4. take a value from mem[addr_b] and name it b
5. compute r = NOR(a, b)
6. take a value from mem[3] and name it addr_r
7. put r to mem[addr_r]

We can generalize this by introducing a register called IP (we'll consider registers in a minute), an instruction pointer, holding an address of the current instruction. Let's write down the algorithm in a pseudo code:

```
mem[mem[IP + 2]] = NOR(mem[mem[IP + 0]], mem[mem[IP + 1]])
```

Once an instruction is processed, we increment IP by 3 and repeat again. You may notice that this loop is infinite, similar to a real CPU. Of course, when we develop an emulator for our CPU we will define a special case when the CPU halts.

Creating Registers

Okay, we have the memory model and the instruction format. Now we need to think about registers. In real CPUs the registers are a bunch of cells usually residing on the main chip and can be read or written very quickly. These cells are supposed to contain data that will be accessed frequently.

Our CPU is not real hardware and we don't have that fast on-chip memory for registers, so we'll just put the registers of our CPU into the main memory (from now on, just memory), in fact, into the mem array. This is a very interesting trick. We'll see why in a second.

We have already declared one register, called IP, holding an address of the current instruction. Because our registers are in the memory, the IP register is also in the memory. We said the instructions in our CPU get data from two cells, calculate NOR, and put a result into another cell. Our IP register is a regular cell in the memory, which means it can be changed similarly to other cells by any instruction. This is how we're going to implement branching.

Looking ahead, you could already guess that our CPU holds the registers, the code, and data in the same memory (the mem array).

Everything is ready to implement the main loop of our CPU. Let's do it in Ruby:

```
def nor(a, b)
  ~(a | b) & 0xFFFF
end

while true do
  i = mem[IP]
  a = mem[i + 0]
  b = mem[i + 1]
  r = mem[i + 2]
  mem[ip] = i + 3
  f = nor(mem[a], mem[b])
  mem[r] = f
end
```

A Ruby IP variable specifies an address on the IP register in the mem array.

Also, you have probably noticed that a function nor(a, b) doesn't do just a single 1-bit NOR operation, but it computes sixteen NORs in parallel (remember, our registers and memory cells are 16-bit): bit-wise NOR between a and b.

Creating More Instructions

All right, our CPU is almost ready. Now we need to learn how to write programs on it. We can compute NOR and other boolean functions. But this is not enough. We need to implement data copying, branching, and multi-bit arithmetic.

At this point it is worthwhile to introduce a slightly different notation for our NOR-based assembly language. Previously we used a notation of functions (for example, NOT(a) = NOR(a, a)), but let's change it to an assembly language.

Our primitive instruction NOR will have following syntax:

```
NOR a, b, r
```

The first word in the line is an instruction name (NOR) and then arguments follow.

Let's rewrite other functions into this notation. Also, for compound functions, we will use a pseudo macro assembler syntax.

NOT, defined as NOT(a) = NOR(a, a):

```
macro NOT a, r
  NOR a, a, r
endm
```

More complex operations require temporary registers. We will declare them using local keywords. These registers are similar to IP in that they reside somewhere in the memory.

AND, defined as AND(a, b) = NOT(OR(NOT(a), NOT(b))):

```
macro AND a, b, r
  local t1, t2
  NOT a, t1
  NOT b, t2
  OR t1, t2, t1
  NOT t1, r
```

```
endm
```

OR, defined as OR(a, b) = NOT(NOR(a, b)):

```
macro OR a, b, r
  local t
  NOR a, b, t
  NOT t, r
endm
```

XOR, defined as XOR(a, b) = OR(AND(a, NOT(b)), AND(NOT(a), b))):

```
macro XOR a, b, r
  local t1, t2
  NOT b, t1
  AND a, t1, t1
  NOT a, t2
  AND t2, b, t2
  OR t1, t2, r
endm
```

A MOV command, copying data from one place to another, can be implemented relatively easily. For example, we want to copy a word from a from address to a to address:

```
mem[to] = OR(mem[from], mem[from])
```

or in the form of a macro:

```
macro MOV from, to
  OR from, from, to
endm
```

A JMP command organically follows from MOV. If the to argument of MOV is the IP register, we effectively write a value of to to IP and make the CPU execute the next instruction from another address, from:

```
mem[IP] = OR(mem[from], mem[from])
```

or in the form of a macro:

```
macro JMP to_addr
  OR to_addr, to_addr, IP
endm
```

Conditional branching is more interesting. Let's set a condition parameter to be 0xFFFF if the condition is true and we do a jump, and 0x0000 otherwise. So the conditional branching can be implemented as:

```
mem[IP] = OR(AND(to_addr, condition), AND(mem[IP], NOT(condition)))
```

or as a macro:

```
macro BRANCH true_addr, false_addr, condition
  local t1, t2
  AND true_addr, condition, t1
  NOT condition, t2
  AND false_addr, t2, t2
  OR t1, t2, IP
endm
```

If condition is 0xFFFF, we write true_addr to IP and jump, but if condition is 0x0000, we write false_addr to IP. Writing to IP means that on the next CPU cycle the execution will continue from another address.

Implementing Basic Math

Let's pause for a second and think what we've done so far.

Having only one instruction, NOR, we have built NOT, AND, OR, XOR, MOV, JMP, and BRANCH. So we now know how to do boolean computations, copying data, and branching. Isn't it cool?

Agreed, we lack multi-bit arithmetic. We really only need summation. With summation we can build subtraction, multiplication, and division. And once we have all of these, we can program any algorithm on our CPU.

A very important point is that we need 16-bit summation. Remember, one instruction in our CPU computes bit-wise NOR. This means that one bit from the a argument and its corresponding bit from b affect only one resulting bit of r. But an algorithm of the full binary adder entails the concept of carry, where we have to carry an extra bit to the summation of the next binary digit.

Currently we cannot move bits from one position to another. We have no shift operation. Let's improve the main CPU loop.

We introduce one extra register and call it SHIFT_REG. This register will contain the result of the last NOR operation cyclicaly shifted to the left.

```
while true do
  i = mem[IP]
  a = mem[i + 0]
  b = mem[i + 1]
  r = mem[i + 2]
  mem[ip] = i + 3
  f = nor(mem[a], mem[b])
  mem[r] = f
  mem[SHIFT_REG] = ((f >> 15) & 1) | ((f & 0x7FFF) << 1)
end
```

The last line computes a cyclic left shift of the last NOR and stores the result to the SHIFT_REG register.

Now let's build a full binary adder.

The full binary adder has the following logic (suppose that a and b are bits):

```
sum = (a ^ b) ^ carry
carry = (a & b) | (carry & (a ^ b))
```

Here a and b are arguments of summation and carry equals 1 if the previous summation had an overflow and we need to take this fact into account.

Initially carry equals 0, and after computing sum we also update carry.

Below is an implementation of the full binary adder. This is a macro based on existing primitive functions (XOR, AND, OR and MOVE).

```
; Input:
;  mask  - a current bit mask (0x0001, 0x0002, 0x0004, 0x0008 etc)
;          This parameter specifies which bit of "a" and "b" we are
;          going to add.
;  carry - a carry from the previous summation
;          This values is "masked" (ANDed) by the "mask" argument.
;  a, b  - an argument addresses
;  r     - an address of the result
; Output:
;  r     - a result
;  carry - a carry to the next bit (already shifted to the left
;          with respect to the "mask")
;  mask  - a mask shifted to the left by one bit
;
macro FADD mask, carry, a, b, r
  local t1, t2, bit_r       ; Local variables
  XOR a, b, t1              ; Formula: sum = (a ^ b) ^ carry.
  XOR t1, carry, bit_r      ;
  AND bit_r, mask, bit_r    ; Mask all bits in 'bit_r'
                            ; except the current one
  OR bit_r, r, r            ; Save the bit to the result: r |= sum
  AND a, b, t2              ; Formula:
                            ; carry = (a & b) | (carry & (a ^ b))
  AND carry, t1, t1        ;
  OR t2, t1, carry          ; The carry is calculated. Its left
                            ; shifted copy is in SHIFT_REG.
  MOV SHIFT_REG, carry      ; (1): Assign the carry to itself but
                            ;      shifted the next bit.
  MOV mask, mask, mask      ; (2): Dummy assignment to just get:
                            ;      SHIFT_REG = mask << 1.
  MOV SHIFT_REG, mask       ; (3): mask = SHIFT_REG = mask << 1
  AND carry, mask, carry    ; Mask all bits in 'carry'
                            ; except the current one
endm
```

A trick occurs at the line labeled (1). On the line previous to (1) we update carry and after that OR operation, the SHIFT_REG register holds a value of carry shifted to the left. At line (1) we copy that shifted value of carry to carry itself.

Another interesting trick is at lines (2) and (3). At line (2) we do a “weird” operation MOV mask, mask, mask which seems to be doing nothing at all and definitely doesn’t change mask. But behind the scenes, the MOV operation also shifts the value of mask to the left and stores the result into S. Then on line (3) we copy that shifted value back to mask. By doing those lines (2) and (3) we prepare mask for summation of the next bit. Finally we clean up carry by zeroing to all except the current, pointed to by mask.

Okay, are you ready for the full 16-bit adder? Here we go.

To make the source shorter we need to introduce one more feature to our pseudo macro assembler: a construct to repeat lines. Its syntax will be *[number]. If a line has such prefix, for example, *16, this line will be repeated a [number] of times.

Now the full 16-bit adder macro:

```
; Input:
;  a, b  - arguments
;  carry - a carry (the least significant bit only matters)
; Output:
;  r     - the result: r = a + c + carry
;  carry - a carry (the least significant bit only matters)
;
macro ADC a, b, carry, r
  local mask                        ; Local variables.
  XOR r, r, r                       ; Set r = 0
  MOV CONST_1,  mask                ; The initial mask value = 0x0001.
  *16 FADD mask, carry, a, b, t     ; Repeat FADD 16 times (no loops,
                                    ; just a repetition).
  MOV t, r                          ; Move the result 'r'.
endm
```

We have called the macro ADC which stands for “Addition with Carry.”

We initialize a result to 0 and then put 0x0001 to mask. We introduce a special cell called CONST_1. This cell simply contains a value of 0x0001. Then we repeat the FADD macro sixteen times. Remember, on each step the FADD updates carry and mask for the next iteration. After sixteen FADD repetitions the carry has been cyclicaly shifted from the most significant sixteenth bit to the lowest significant first bit.

That is it. The full 16-bit summation with carry is ready.

Programming a Real Algorithm

Let me recap once again what we have so far. We have 64K memory of 16-bit values and two registers, IP and SHIFT_REG. We can do boolean functions, we can copy cells and do branching, and finally we have 16-bit summation with carry.

Now we have enough to program any algorithm on our CPU.

You may say we lack the call stack and the concept of subroutines. Do you want to implement them? Let’s implement two auxiliary operations for indirect data access first.

We call them PEEK and POKE, similar to operators in the BASIC language, where the first one returns a value from a given address and the second one puts a value to a given address. Implementations of these operations are interesting because we will write code modifying itself on the fly. This is possible because the registers, code, and data in our CPU share the same address space, the memory.

Okay, let’s do PEEK first.

```
; This macro reads a value from the address given in "a"
; and stores it to the cell whose address is "b."
;
; Input:
;   a - an address
; Output
;   b - a values from a cell addressed by "a"
;
; This is an indirect addressing: [a] -> b
```

```
;
macro PEEK a, b
  local t
  MOV a, label1  ; In these two lines we create
  MOV a, label2  ; a "NOT x, y, t" command on-the-fly.
 label1:         ; Initially there is a "NOT 0, 0, t"
  dw 0           ; command here but by two previous "MOV"s
 label2:         ; we replace two zeros from with
  dw 0           ; a concrete value of "a"
  dw t
  NOT t, b       ; Final NOT to "b".
endm
```

Technically, PEEK is similar to MOV. It does two subsequent NOTs to copy a value from one cell to another. The trick is that we modify the first NOT by injecting the concrete address a. The second NOT is a regular one. The label1 and label2 are local for this macro.

Let’s implement POKE.

```
; This macro writes a given value "a"
; to a cell addressed by "b."
;
; Input:
;   a - a value
;   b - an address
; Output:
;   None.
;
; This is an indirect addressing: a -> [b]
;
macro POKE a, b
  local t
  MOV b, label    ; We create a "NOT t, t, x" command on-the-fly.
  NOT a,  t       ; This is a regular "NOT" operation.
  dw @t           ; This is a "NOT t, t, 0" command
  dw @t           ; and by the first "MOV" in the macro
 label:           ; we replace that "0" with a value of "b"
  dw 0
endm
```

POKE also uses the MOV technique. It has two subsequent NOT operations and the second one is being modified on the fly by injecting the concrete address of b.

How cool is that?!

PEEK and POKE are very powerful operations. Having them, it is very easy to implement a concept of the call stack to be able to call subroutines. I leave it as an exercise for the readers.

Let’s make our CPU compute something useful. For example, CRC16.

The algorithm in Ruby might look like:

```
def calc_crc16(data)
  crc = 0xFFFF
  data.each_byte do |x|
    crc = crc ^ (x & 0xFF)
    8.times do
      shift = (crc & 1) != 0
      crc = crc >> 1
      crc = crc ^ 0x8401 if shift
    end
  end
  return crc
end
```

Usually when people develop in an assembly language they use a translator, an assembler, converting a text representation of CPU instructions into machine code. Also the assembler can do macro substitution (we have already used macros, and we do need this functionality), data allocation, and symbolic labels.

The assembler in most cases is a standalone application that takes a source in the assembly language and produces a binary.

To make our stuff a bit more fun we'll use another approach. We will use Ruby as our assembler. We will develop a few helper functions, allowing us to write regular Ruby code, and the code will automatically produce our NOR-based machine code. You'll be amazed at how great Ruby is at doing that.

And finally we'll run a compiled binary on the emulator.

Let's begin not from the low-level helpers, but from the CRC implementation.

The code below computes the CRC-16 from a string given in a needle variable. I know, some macros and commands may look new to you, but we'll consider them shortly.

Just reflect on this code for a second. Does it look like Ruby at all? To me it looks like assembly language. Here you can unfold the magic of Ruby because it is Ruby. The tiny helper functions make it possible.

```
needle = literal Example + "\x00"

i, ch, t, ptr, crc16 = var 5

label :entry
      MOVi 0xFFFF, crc16
      MOVi needle, ptr
label :crc_loop                # crc_loop:
      PEEK ptr, ch             #   ch = *ptr
      IS_0 ch                  #   is ch == 0?
      JZi  :exit               #   if yes, goto "exit"
      ANDi ch, 0xFF, ch        #   ch &= 0xFF
      XOR  crc16, ch, crc16    #   crc16 ^= ch
      MOVi 8, i
label :loop_i                  # crc_loop_i:
      ANDi crc16, 1, t         #   t = crc16 & 1
      SHR crc16, crc16         #   crc16 >>= 1
      IS_0 t                   #   is t == 0?
      JZi :skip_xor            #   if yes, goto "skip_xor"
      XORi crc16, 0x8401, crc16  #   crc16 ^= 0x8401
label :skip_xor                  # skip_xor:
      ADDi i, 0xFFFF, i          #   i -= 1
      IS_0 i                     #   is i == 0?
      JNZi :loop_i               #   if not, goto "loop_i"
      ADDi ptr, 0x0001, ptr      #   ptr += 1
      JMPi :crc_loop             #   goto crc_loop

      label :exit
      EXITi()
```

The Full Source

All right, now we are approaching the finish line. We'll go through the entire source of the program in Ruby, which contains this CRC-16 calculation code, the helper functions, NOR-based binary generation routines, and, finally, the emulator of our CPU.

When this code runs, it calculates CRC-16 twice, using our NOR CPU and a regular Ruby implementation. I hope the results will match!

Line 1 sets up a string we use to calculate CRC-16.

```
1  Example = "String for testing"
```

From lines 2 to 37 there are helper functions allowing us to make Ruby code look so close to the assembly. We have functions:

- to declare and set symbolic labels (make_label() in line 8 and label(), line 14)
- to place NOR code into the assembly (code() in line 15)
- to place a string into the assembly (literal() in line 16)
- to declare and place into the assembly different types of variables and constants (static() in line 21, var() in line 24, init_var() in line 29, and const in line 34)

```
 2  Text = []
 3  Labels = []
 4  Statics = {}
 5  Consts = {}
 6  Vars = {}
 7  Literals = {}

 8  def make_label(name = "")
 9    name = name.to_s
10    name = "label_%d" % Labels.size if name.empty?
11    Labels.push name
12    return name
13  end

14  def label(name) Text.push "#" + name.to_s end
```

```
15  def code(value) Text.push "  %s" % value end

16  def literal(value)
17    name = "str_%d" % Literals.size
18    Literals[name] = value
19    return name
20  end

21  def static(*names)
22    names.each { |name| Statics[name] = 0 }; names
23  end

24  def var(count = 1)
25    names = []
26    count.times { Vars[(names.push "var_%d" % Vars.size).last] = 0 }
27    return names
28  end

29  def init_var(value)
30    name = "var_%d" % Vars.size
31    Vars[name] = value
32    return name
33  end

34  def const(value)
35    return Consts[value] if Consts.has_key? value
36    Consts[value] = "const_%d" % Consts.size
37  end
```

In lines 38-39 we define the main registers of our CPU: IP, SHIFT_REG, CARRY_REG, and ZERO_REG.

```
38  IP = init_var :entry

39  SHIFT_REG, CARRY_REG, ZERO_REG = var 3
```

The function NOR() in line 40 is our main building block. All other functions are built using this one. This function places into the assembly three words of the instruction.

```
40  def NOR(a, b, r)
41    code a
42    code b
43    code r
44  end
```

In lines 45-89 we define the bit-wise boolean functions jump and move. Their implementations repeat exactly what we've discussed previously. The static helper function allows us to define temporary variables for each macro. These temporary variable are singletons. There is one instance of each variable (for example, OR_t) across all invocation of a macro where it is defined.

It is worthwhile to look at functions having "i" in the name (ANDi() in line 60, XORi() in line 73, JMPi() in line 84 and

MOVi() in line 87). These functions have an immediate value in the second parameter. Normally all arguments of our macros are addresses pointing to the values, not values per se. Each of these functions has to place the immediate value to a temporary variable first and then use the const() helper to create an unnamed constant.

```
45  def NOT(a, r)
46    NOR a, a, r
47  end

48  def OR(a, b, r)
49    t = static :OR_t
50    NOR a, b, t
51    NOT t, r
52  end

53  def AND(a, b, r)
54    t1, t2 = static :AND_t1, :AND_t2
55    NOT a, t1
56    NOT b, t2
57    OR t1, t2, t1
58    NOT t1, r
59  end

60  def ANDi(a, imm, r)
61    t = static :ANDi_t
62    MOVi imm, t
63    AND a, t, r
64  end

65  def XOR(a, b, r)
66    t1, t2 = static :XOR_t1, :XOR_t2
67    NOT a, t1
68    NOT b, t2
69    AND a, t2, t2
70    AND b, t1, t1
71    OR t1, t2, r
72  end

73  def XORi(a, imm, r)
74    t = static :XORi_t
75    MOVi imm, t
76    XOR a, t, r
77  end

78  def MOV(a, b)
79    OR a, a, b
80  end

81  def JMP(a)
```

```
 82    MOV a, IP
 83  end

 84  def JMPi(a)
 85    JMP const(a)
 86  end

 87  def MOVi(imm, a)
 88    MOV const(imm), a
 89  end
```

In lines 90-105 there is an implementation of a PEEK() function. As we have discussed previously this function applying indirect addressing has to modify its code on the fly by injecting the a value to the second NOR (line 97-103). We create two labels l1 and l2 (lines 92 and 93) and use them (lines 98 and 100) to address the locations where a has to be injected.

```
 90  # [a] -> b
 91  def PEEK(a, b)
 92    l1 = make_label
 93    l2 = make_label
 94    MOV a, l1
 95    MOV a, l2
 96    t = static :PEEK_t
 97    # BEGIN: NOR 0, 0, t
 98  label l1
 99    code 0   # <- a [initially '0']
100  label l2
101    code 0   # <- a [initialli '0']
102    code t
104    NOT t, b
105  end
```

In lines 106-118 there is a POKE() function. This function also does indirect addressing; that’s why it modifies itself similar to PEEK(). We create a label l in line 108 and place it in line 115. The label l identifies a location to inject the value of b.

```
106  # a -> [b]
107  def POKE(a, b)
108    l = make_label
109    MOV b, l
110    t = static :POKE_t
111    NOT a, t
112    # BEGIN: NOR t, t, 0
113    code t
114    code t
115  label l
116    code 0   # <- b [initially '0']
118  end
```

In lines 119-121 there is a function halting our CPU. We haven’t discussed yet how to stop the CPU. Below is one possible way. Say we stop the CPU when IP becomes equal to 0xFFFF. Of course, the emulator needs to implement this behavior, and you’ll see it further down.

```
119  def EXITi
120    MOVi 0xFFFF, IP
121  end
```

The Full Binary Adder

In lines 122-149 there is an implementation of the full binary adder. The function FADD() does what we discussed, and the implementation is nicely wrapped into Ruby helpers. Note that this function doesn't use the CARRY_REG register directly (as with SHIFT_REG). Instead it uses the carry argument passed from a caller function.

```
122  def FADD(mask, carry, a, b, r)
123    tmp_a, tmp_b, bit_r = static :FADD_a, :FADD_b, :FADD_bit_r
124    t1, t2 = static :FADD_t1, :FADD_t2
125
126    AND a, mask, tmp_a      # zeroing bits in 'a' except mask'ed
127    AND b, mask, tmp_b      # zeroing bits in 'b' except mask'ed
128    AND carry, mask, carry  # zeroing bits in 'carry' except mask'ed

129    # SUM = (a ^ b) ^ carry
130    XOR a, b, t1
131    XOR t1, carry, bit_r

132    # Leave only 'mask'ed bit in bit_r.
133    AND bit_r, mask, bit_r

134    # Add current added bit to the result.
135    OR bit_r, r, r

136    # carry = (a & b) | (carry & (a ^ b))
137    AND a, b, t2
138    AND carry, t1, t1

139    # carry is calculated, and 'SHIFT_REG' contains the same value
140    # but shifted to the left by 1 bit.
141    OR t2, t1, carry

142    # CARRY is shifted to the left by 1 bit for the next round.
143    MOV SHIFT_REG, carry

144    # SHIFT_REG = mask << 1
145    MOV mask, mask
146    # mask = shift (in fact, "mask = mask << 1")
147    MOV SHIFT_REG, mask

148    AND carry, mask, carry
149  end
```

A function ZERO() in line 150-152 uses the XOR trick to put 0 into its argument.

```
150  def ZERO(a)
151    XOR a, a, a
152  end
```

In lines 153-166 there is an ADC() function implementing 16-bit summation with carry. In line 157 it calls FADD() sixteen times and then copies the result to r in line 158.

Please consider the lines 159-166. After iterating the FADD() function we have CARRY_REG equal to 0x0000 if there was no overflow during summation, or 0x0001 if there was. As you remember, where we were talking about the BRANCH() function, we said that it expects the condition to be either 0x0000 (false) or 0xFFFF (true). We are going to use CARRY_REG as the condition later on, so we have to convert 0x0001 to 0xFFFF.

In lines 159-165 we rotate the values of CARRY_REG (0x0001) across all sixteen bits of the word and in each position we OR it with t. By doing that we’re spreading that 1 in the lowest significant position to all the bits in the word, and eventually 0x0001 becomes 0xFFFF.

```
153  def ADC(a, b, r)
154    mask, t = static :ADC_mask, :ADC_t
155    ZERO t
156    MOVi 0x0001, mask
157    16.times { FADD mask, CARRY_REG, a, b, t }
158    MOV t, r

159    ZERO t                      # Set t = 0.
160    16.times do                 # Rotate "CARRY_REG" 16 times.
161      OR t, CARRY_REG, t        # t |= CARRY_REG
162      MOV CARRY_REG, CARRY_REG  # SHIFT_REG = CARRY_REG << 1
163      MOV SHIFT_REG, CARRY_REG  # CARRY_REG = CARRY_REG << 1
164    end
165    MOV t, CARRY_REG            # CARRY_REG = t
166  end
```

In lines 167-175 there are two wrappers of ADC performing summation without using CARRY_REG.

```
167  def ADD(a, b, r)
168    ZERO CARRY_REG
169    ADC a, b, r
170  end

171  def ADDi(a, imm, r)
172    t = static :ADDi_t
173    MOVi imm, t
174    ADD a, t, r
175  end
```

In lines 176-187 there are functions for conditional branching. The main idea is to use OR() (line 182) as a decision maker.

```
176  # Jump 'a', if cond = FFFF, and 'b' if cond = 0000
177  def BRANCH(a, b, cond)
178    t1, t2 = static :BRANCH_t1, :BRANCH_t2
179    AND a, cond, t1     # t1 = a & cond
```

```
180   NOT cond, t2        # t2 = !cond
181   AND b, t2, t2       # t2 = b & t2 = b & !cond
182   OR t1, t2, IP       # ip = (a & cond) | (b & !cond)
183 end

184 # Jump 'a', if cond = FFFF, and 'b' if conf = 0000
185 def BRANCHi(a, b, cond)
186   BRANCH const(a), const(b), cond
187 end
```

In line 188-194 there is a function we haven't seen yet. It checks whether the a argument is zero or not. It adds 0xFFFF to a, and there is only one value of a possible when the overflow doesn't happen and CARRY_REG is 0 afterwards. This value is 0. If a is non-zero, there will be an overflow and CARRY_REG will become 0xFFFF. Then we negate the value of CARRY_REG and assign it to ZERO_REG.

So, if a is 0 this function sets ZERO_REG to 0xFFFF, or 0x0000 otherwise.

```
188 # if a != 0 then ZERO_REG = FFFF else ZERO_REG = 0000
189 def IS_0(a)
190   t = static :IS_0_t
191   ZERO CARRY_REG
192   ADC a, const(0xFFFF), t
193   NOT CARRY_REG, ZERO_REG
194 end
```

In lines 709-716 there are two functions doing conditional branching. They almost repeat the functionality of BRANCHi except that if the condition is false they carry on execution from the next instruction. This is implemented by introducing a bypass label pointing to an address right after the macro.

```
195 # ip = (ZERO_REG == FFFF ? a : ip)
196 def JZi(a)
197   bypass = make_label
198   BRANCHi a, bypass, ZERO_REG
199 label bypass
200 end

201 # ip = (ZERO_REG == FFFF ? a : ip)
202 def JNZi(a)
203   bypass = make_label
204   BRANCHi bypass, a, ZERO_REG
205 label bypass
206 end
```

In lines 725-224 there are functions doing different kinds of shifts. They all are based on the SHIFT_REG register.

```
207 def ROL(a, b)
208   MOV a, a             # SHIFT_REG = a << 1
209   MOV SHIFT_REG, b
210 end

211 def ROR(a, b)
```

```
212    t = static :ROR_t
213    MOV a, t
214    15.times { ROL t, t }
215    MOV t, b
216  end

217  def SHL(a, b)
218    ROL a, b
219    ANDi b, 0xFFFE, b
220  end

221  def SHR(a, b)
222    ROR a, b
223    ANDi b, 0x7FFF, b
224  end
```

The CRC-16 Algorithm

All right, at last in lines 225 to 251 we have our CRC-16 algorithm implemented via the functions we created previously. There are two loops. The outer one (crc_loop label) iterates over the source string via the ptr pointer up to the \x00 character. The inner loop iterates eight times over the current byte and applies the CRC-16 routine.

Ignoring for a moment what this code is doing, let’s just reflect on it. It doesn’t really look like Ruby. It looks like assembly language!

```
225  # NORCPU code

226  needle = literal Example + "\x00"

227  i, ch, t, ptr, crc16 = var 5

228  label :entry
229        MOVi 0xFFFF, crc16
230        MOVi needle, ptr
231  label :crc_loop                # crc_loop:
232        PEEK ptr, ch             #   ch = *ptr
233        IS_0 ch                  #   is ch == 0?
234        JZi  :exit               #   if yes, goto "exit"
235        ANDi ch, 0xFF, ch        #   ch &= 0xFF
236        XOR  crc16, ch, crc16    #   crc16 ^= ch
237        MOVi 8, i
238  label :loop_i                  # crc_loop_i:
239        ANDi crc16, 1, t         #   t = crc16 & 1
240        SHR crc16, crc16         #   crc16 >>= 1
241        IS_0 t                   #   is t == 0?
242        JZi :skip_xor            #   if yes, goto "skip_xor"
243        XORi crc16, 0x8401, crc16  #   crc16 ^= 0x8401
244  label :skip_xor                # skip_xor:
245        ADDi i, 0xFFFF, i        #   i -= 1
```

```
246       IS_0 i                     #   is i == 0?
247       JNZi :loop_i               #   if not, goto "loop_i"
248       ADDi ptr, 0x0001, ptr      #   ptr += 1
249       JMPi :crc_loop             #   goto crc_loop

250  label :exit
251       EXITi()
```

In lines 252-278 we build our assembly. In lines 253-257 we join all the parts (code, labels, variables, constants, and strings) together into the assembly array. The labels have a # prefix.

```
252  assembly = []
253  Vars.each { |k, v| assembly.push "#%s" % k, v }
254  Text.each { |x| assembly.push x }
255  Statics.each { |k, v| assembly.push "#%s" % k, v }
256  Consts.each { |k, v| assembly.push "#%s" % v, k }
257  Literals.each do |k, v|
258    assembly.push "#%s" % k
259    v.each_byte { |x| assembly.push x }
260  end
```

In lines 261-270 we walk through the assembly, calculate the labels' addresses, and store them into the labels dictionary.

```
261  offset = 0
262  labels = {}
263  assembly.each do |x|
264    x = x.to_s
265    if x.start_with? '#' then
266      labels[x[1..-1]] = offset
267    else
268      offset = offset + 1
269    end
270  end
```

Finally, in lines 271-278 we cut out the labels from the assembly and replace the symbolic label names with concrete values.

```
271  # Remove all list having labels.
272  assembly.delete_if { |x| x.to_s.start_with? "#" }

273  # Substitute labels by values.
274  assembly.collect! do |x|
275    x = x.to_s.strip
276    subst = labels[x]
277    (if subst == nil then x else subst end).to_i
278  end
```

Now we have our code fully compiled and built into a sequence of NOR commands. It is in the assembly array.

In line 279-297 there is the NOR CPU emulator—the heart of our long journey. It executes the NOR code taken from the assembly until a halt condition when IP becomes equal to 0xFFFF (remember the EXITi macro?)

```
279  def nor(a, b)
280    ~(a | b) & 0xFFFF
281  end

282  ip = labels[IP]
283  shift_reg = labels[SHIFT_REG]

284  puts "Start"
285  mem = assembly
286  while true do
287    i = mem[ip]
288    a = mem[i + 0]
289    b = mem[i + 1]
290    r = mem[i + 2]
291    mem[ip] = i + 3
292    f = nor(mem[a], mem[b])
293    mem[r] = f
294    mem[shift_reg] = ((f >> 15) & 1) | ((f & 0x7FFF) << 1)
295    break if mem[ip] == 0xFFFF
296  end
```

Once the execution is complete, we print the content of the crc16 variable, which should contain our CRC16 value (line 297).

```
297  puts "NOR CPU CRC16: %04X" % mem[labels[crc16]]
```

In lines 298-310 we calculate CRC16 again using a purely Ruby implementation to make sure that our wonderful NOR-based program works correctly.

```
298  def calc_crc16(data)
299    crc = 0xFFFF
300    data.each_byte do |x|
301      crc = crc ^ (x & 0xFF)
302      8.times do
303        shift = (crc & 1) != 0
304        crc = crc >> 1
305        crc = crc ^ 0x8401 if shift
306      end
307    end
308    return crc
309  end

310  puts "Ruby CRC16: %04X" % calc_crc16(Example)
```

Wow! We're done!

At last we can run it (presuming you have the source in the file norcpu.rb).

```
ruby norcpu.rb
```

And hopefully it should print:

```
Start
NOR CPU CRC16: F6AD
Ruby CRC16: F6AD
```

If the values match, it proves that our NOR-based CPU works.

To play with the source, you can download it from the magazine website.

Reflecting on what we have done, you might notice a serious drawback to our CPU: the number of NOR instructions to do even simple calculations is really big. For example, the size of our assembly code for calculating the tiny CRC16 algorithm is already 20273 words, a third of our memory size! So, an implementation of more sophisticated algorithms would require us to change the memory model to be at least 32-bit.

To be honest I don't see any real applications for such a CPU, but in educational terms I believe it has served well. Anyway, I hope it was a fun little exercise!

Alexander Demin is a software engineer and Ph.D. in Computer Science. Constantly exploring new technologies, he believes that something amazing is always out there. He can be contacted at alexander@demin.ws or found at his home page or blog.

Send the author your feedback or discuss the article in the magazine forum.