BIT.C MACHINE (REV 3.3)

A simple, Turing-complete and easy to recreate CPU architecture.

SPECS

Registers:

- 16, 64-bit general purpose registers all initialized to zero. Can be accessed with the notion `rA`, `rB`, `rC`, ..., `rP`.
- 3 special registers: `LAST`, `PC` and `ERR`. `LAST` can only be set by instruction outputs and not directly, And cannot be read. `PC` can be set and read using the `pc` instruction. `ERR` will only be set to an error code in case of an error and followed immediately by a CPU halt.

Memory:

By default, The bit machine has `131,072` bytes of random-access memory. `16,384` bytes of this memory starting from `0` are for the system code. The usual, Non-magic-addressed memory starts at 20,001. And inbetween `16,384 … 20,000` exist the magic addresses needed to interact with modules outside the CPU and RAM. Extra memory could theoretically be achieved using 3rd-party input devices.

Addressing modes:

`1 <constant>`: A constant value. Cannot be used in `addr` mode for addressing modes.

`01 <register>`: If used as a destination, It will write to the specified register. Otherwise, It will read the value from said register.

`00 <register>`: If used as a destination, It will write to the memory address pointed to by the value of the specified register. Otherwise, It will read from the same memory address.

INSTRUCTIONS:

`00`: Manager instruction (2^{x+3})

- `00` Halt: This instruction will stop all execution immediately forever. Should be preceded by a shutdown preparation.
- `11` (2^{3+3}) 64-bit mode: Puts the machine in 64-bit mode. In this setting, Each instruction is 64 bits and an address mode is 30 bits.
- `10` (2^{2+3}) 32-bit mode: Puts the machine in 32-bit mode. In this setting, Each instruction is 32 bits and an address mode is 14 bits.
- `01` (2^{1+3}) 16-bit mode: The smallest instruction size. In this mode, Each instruction is 16 bits and an address mode is 6 bits. Useful for saving space when working only with regsiters or pre-set registers containing a memory address.

`01` : MOV instruction

- Syntax:
    - mov(**data** from, **addr** to)
- Moves data between two places, Which each could be a register, A memory address-containing register, etc.
- **Notice**: This instruction can be written using the math instruciton as follows:
-
```
macro mov(SRC, DEST)
  math nand DEST, $0    ; Becomes all 1s
  math nand DEST, DEST ; Becomes all 0s
  math add DEST, SRC    ; 0 + x == x
end mov
```

`10` : PC (program counter) instruction

- Syntax:
    - pc(**bit** get/set, **bit[3]**? flags, **data** target)
    - pc(<set>, **addr** target)
    - jmp(**bit(3)** flags, **data** target) == pc(get, …)
- If the first bit is set, The machine will jump to said target in memory (Sets `PC` to target) if any of the flags match the `LAST` register.
- If the first bit is not set, The target type changes to an `addr`. The flags are ignored and the current value of `PC` incremented by 1 is written to `target`.
- The `LAST` register is a special register that cannot be read from the code and can only be set by the code and used by the JMP instruction. It is set to `from` in `mov`, `0` in `manager`, Doesn't change in `jmp` and set to the result in `mth`.

`11` : MATH instruction

- Syntax:
    - math(**bit** operation, **addr** left, **data** right)
- If the first bit is set, The operation is `nand`. Equivalent C code would be `left = ~(left & right);`
- If it is not set, The operation is `add`. Equivalent C code would be `left += right;`

MAGIC ADDRESSES:

`17,000`: Shutdown byte. If non-zero, The CPU is ready for shutdown and can safely be powered off. Usually followed by a halt instruction to prevent further changes.

`17,001`: Display ready byte. If non-zero, The CPU is ready to write to the screen. Should be set to zero by the display after the write is done and a clock cycle has passed.

`17,002`: Color mode byte. If non-zero, The CPU writes one RGB pixel at a time. If zero, The CPU writes 24 B&W pixels to the screen. This setting should only be set by the CPU.

`17,003 … 17,005`: If in B&W mode, 24 B&W pixels in these locations are written to the display. If in color mode, They each indicate a setting of RGB.

`17,006 and 17,007`: X position bytes. Indicate where the display should write the said pixel(s). Maximum width is therefore capped to 65,536 pixels.

`17,008 and 17,009`: Y position bytes. Indicate where the display should write the said pixel(s). Maximum height is therefore capped to 65,536 pixels.

`17,010`: Input ready byte. If non-zero, It means that input from the keyboard is available. Can be set by the CPU and the keyboard. The keyboard should only write to this if it is already `0`.

`17,011`: Input byte. Should only be set by the keyboard if it is zero. The CPU should clear this byte after it is done.

`17,012 … 17,020`: The current time as a unix timestamp. Should only be set by the cmos clock.

`17,021 … 17,037`: The current clock cycle count. Should only be set by the system clock.

`17,038`: 3rd-party input type. Can support up to 256 types. Reads/Writes 8 bytes at a time. Type 0 should be the hard drive, Type 1 should be the mouse if available, And the rest are up to the user.

`17,039 … 17,046`: Input index bytes. Indicates a `uint64`.

`17,047`: 3rd-party input ready byte. Can be set by the CPU and the keyboard. The 3rd-party input should only write to this if it is already `0`.

`17,048 … 17,057`: 3rd-party input value. 8 bytes of input. Should be set to zero after the CPU is done reading them.

TODO: Magic address documentation for writing data to 3rd-party devices

ASSEMBLER USAGE GUIDE

HLT

- Subset of <manager> instruction.

Halts the program. Instruction is `0000`.

MODE64: Subset of <manager> instruction.

Switches the program to 64-bit mode. Instruction is `0001`.

MODE32: Subset of <manager> instruction.

Switches the program to 32-bit mode. Instruction is `0010`.

MODE16: Subset of <manager> instruction.

Switches the program to 16-bit mode. Instruction is `0011`.

MOV

```
mov $20001, rA             ; set rA to 20,001
mov $5, mA                 ; set memory address 20,001 to $5
```

MATH

```
math nand rA, $5           ; rA = ~(rA & 5)
math add rA, rB            ; rA += rB
```

PC/JMP

```
jmp (zero or more) rA      ; jump to memory in rA if LAST >= 0
pc set (less or more) rB   ; jump to memory in rB if LAST != 0
pc get rD                  ; write PC to rD
```

Macros

```
macro name(arg1, arg2)
  mov arg1, rA
  mov arg2, rB
end name

@name($5, $6)              ; moves 5 to rA and 6 to rB.
```

```
define name $20001
mov !name, rF              ; equivalent to mov $20001, rF
```

BIT.C STANDARD LIBRARY

Below you can find a set of macros that can be used to improve developer experience with the bit machine.

```
macro mov(SRC, DEST)
  math nand DEST, $0     ; DEST = -1
  math nand DEST, DEST   ; DEST  = 0
  math add DEST, SRC     ; DEST = SRC
end mov

macro Decrement(x)            ; just a neat trick to only use one storage
                              ; service when decremeneting.

  math nand x, x              ; invert once
  math add x, $1              ; add 1
  math nand x, x              ; invert again
  ; x - 1 = -(-x + 1)

define StackPointer $20001
define StackValue $20002      ; upwards growing stack

macro InitStack()
  mov !StackPointer, rP       ; move the stack pointer to rP
  mov !StackValue, mP         ; move the stack pointer's value to mP
end InitStack

macro Push(arg)               ; arg cannot be rA or rB.
  mov !StackPointer, rA
  mov mA, rB
  mov arg, mB                 ; move arg to current stack location
  mov !StackPointer, rA
  math add mA, $1             ; increment the stack pointer
end Push

macro Pop(put)                ; do NOT set arg to rA, rB or rC.
  mov !StackPointer, rA
  mov mA, rB
  mov mB, put                 ; get the stack value
  mov !StackPointer, rA
  @Decrement(mA)
end Pop

macro Sub(vA, vB)
  mov vB, rA
  math nand rA, rA
  math add rA, $1             ; two's complement
  math add vA, rA             ; add vA to vB
end Sub

macro GetChar(destination)    ; dest shouldn't be rA or rB
  mov $17010, rA              ; input ready byte
  pc set rB                   ; store program counter
  math add mA, $0             ; get value of input ready byte
  jmp (zero) rB               ; jump back if zero
  mov $17011, rA
  mov mA, destination         ; get char byte
  mov $0, mB                  ; zero out input ready byte to read more
  mov $0, mA                  ; zero out the char byte
end GetChar

macro ShutDown()
  mov $17000, rA
  mov $0, mA                  ; set shutdown byte to zero
  math nand mA, mA            ; set shutdown byte to 1
  halt                        ; stop cpu execution
end ShutDown
```

```
macro Multiply(left, right)
  mov right, rA
  mov $0, rB
  mov $0, rD
  math nand rB, rB            ; -1
  pc set rC
  math add rD, left           ; rD += left
  math add rA, rB
  jmp (less or more) rC       ; repeat until zero
  mov rD, left
end Multiply

macro Read3rdParty(type, dest)
  @Write(type, $17038)        ; 3rd-party input type magic address
  mov $17039, rA
  pc set rB
  math add mA, $0             ; get value in mA
  jmp (zero) rB               ; block until non-zero
  mov rA, rE                  ; stash for clearing later
  mov $17040, rA
  mov dest, rB
  mov $8, rC
  pc set rD
  mov mA, mB
  mov $0, mA
  math add rA, $1
  math add rB, $1
  @Decrement(rE)
  jmp (less or more) rD
  mov $0, mE                  ; clear everything up
end Read3
```