Ternary Emulation Manual

Scott Carda

Contents

Introduction

This manual describes the Emu Emulator, an emulation of a ternary logic machine. This emulator exists as a python package and can be implemented in any python3 script. The basics of ternary logic are described as well.

Ternary Logic

While the base unit of binary computers is a bit, having two states, the base unit of ternary computers is a trit, having three states. These states are represented in two ways: as the numbers 0, 1, and 2, and and the truth values False, Neutral, and True, or F, N, and T, respectively. The mapping between these two representations is 0 = F, 1 = N, and 2 = T.

Representation

Trits can be agglomerated into larger units called trytes, a tryte containing nine trits, having 3^9 states. This is the ternary complement to the binary byte. Because this is a large number of bits, it is advantageous to use a more compact way of representing trytes rather than spelling out the value of each trit. One way of accomplishing this is to interpret the series of trits as digits of a number represented in the number base 3, and convert the number to base 10. In this way, groups of trits can represent large numbers. This conversion, while human readable, is too computationally intensive and inefficient to be used to significant degree internally in the emulator. We will use the heptavigesimal numbering system, or hept for short, which is base 27. Because the hept is base 27, which is a power of 3, the representation will map directly to the values of the trits, unlike base 10. A single character in the hept system will represent all 27 possible states of a group of three trits, so we will need three hept characters to represent one tryte. The characters we will use for our implementation of the hept system will start with the ampersat, or the "@" symbol, representing the zero state. Then states 1 through 26, those being the base 10 conversions of the numbers acquired from interpreting the trits as base 3 numbers, will be represented by the letters A-Z of the alphabet, respectively.

Logic Gates

The three-valued nature of the trits makes them incompatible with the traditional binary logic gates. A new set of logic gates must be used to construct logic on a ternary machine. We will be using truth tables to represent the logic of an individual gate. For single input gates, the left column represents the single trit input into the gate, and the right column represents the single trit output. For two input gates, the leftmost column acts as an axis and represents the first of the two trit inputs. The topmost row acts as an axis and represents the second of the two trit inputs. The rest of the table represents the output values of the single output trit. The following are several logic gates that are defined:

FNOT	
F	F
N	T
T	N

NNOT	
F	T
N	N
T	F

TNOT	
F	N
N	F
T	T

SHIFT_M	
F	T
N	F
T	N

SHIFT_P	
F	N
N	T
T	F

ISF	
F	T
N	F
T	F

ISN	
F	F
N	T
T	F

IST	
F	F
N	F
T	T

AND	F	N	T
F	F	F	F
N	F	N	N
T	F	N	T

OR	F	N	T
F	F	N	T
N	N	N	T
T	T	T	T

XOR	F	N	T
F	F	N	T
N	N	N	N
T	T	N	F

LOCATOR	F	N	T
F	F	F	F
N	F	F	F
T	F	F	T

SUM	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

SUM_C	0	1	2
0	0	0	0
1	0	0	1
2	0	1	1

The Emu Emulator

The Emu Emulator is an emulator for a ternary computing system. It uses a 27-trit instruction set, and a word in this machine is 27 trits, or three trytes. The 9 programmable index registers, as well as the program counter, or PC, are all single word registers. The current program status register, or CPSR, is a register with three trits flags in it that keep track of the qualities of the results of certain operations, which can be used to dictate the program flow. Programs are stored in RAM as a series of instructions and data. RAM is a 27-trit address space that is word-addressable.

The Emu emulator can be used in a python script by importing the Emulator package. Create an Emu object, then use the member function "read_object", passing the path to an object file, prepared ahead of time, as a string. This will read the program, represented by the object file, into the RAM of the Emu object. Finally, call the member function "start" to have the program executed by the Emu Emulator object.

Instruction Format

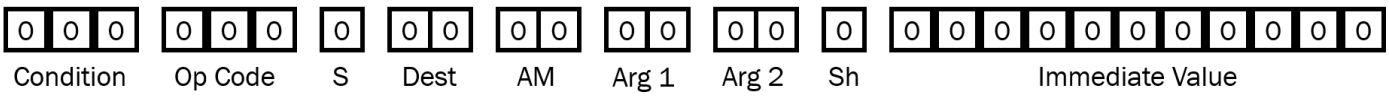

Figure 1: 27-trit Instruciton Format

Condition: The 3-trit condition code the machine uses to determine whether the instruction is executed.

Op Code: The 3-trit code that indicates which operation is performed with this instruction.

S: The instruction's flag mode. The jump operation has unique behavior associated with this.

Dest: The 2-trit indicator for the destination register.

AM: The instruction's 2-trit address mode.

Arg 1: The 2-trit indicator for the register to be used as the first argument to the operation.

Arg 2: The 2-trit indicator for the register to be used as the second argument to the operation.

Sh: The instruction's shift mode. Indicates which shift operation is performed on Arg 2.

Immediate Value: This 11-trit field is reserved for embedding numerical values in the instruction.

Conditions

The Current Program Status Register or CPSR is a special 3-trit register that keeps track of several flags. There are three flag, one associated with each of the three trits in the register. The most significant trit is the **S** flag, which indicates the sign of the result of an operation. Table 1 shows the meaning of the values for the **S** flag. The middle of the three trits is the **C** flag, which contains the value of any unsigned overflow resulting from an operation. The least significant trit is the **V** flag, which contains the value of any signed overflow resulting from an operation.

Table 1:

S Flag	Meaning
0	Result was Positive
1	Result was Zero
2	Result was Negative

Instruction are conditionally executed based on the values of these flag. Common conditions have been abstracted from these flag and made available to the system. Each condition is assigned a 3-trit code for use in the most significant field of an instruction. If the condition is false, the rest of the instruction is ignored, and the machine will move on to the next instruction in its RAM without executing the encoded operation. The conditions available to the system are found in Table 2. As this is a 3-trit code, each of these condition codes can be represented as a single heptavigesimal character, which are found in parentheses in the first column of the table.

Table 2:

Condition Code	Meaning	Flags
000 (@)	Always	Always
001 (A)	Equal	$\mathbf{S} = 1$
002 (B)	Carry Clear	$\mathbf{C} = 0$
010 (C)	Negative	$\mathbf{S} = 2$
011 (D)	Overflow Clear	$\mathbf{V} = 0$
012 (E)	Unsigned Lower/Equal	$\mathbf{C} = 0$ **OR** $\mathbf{S} = 1$
020 (F)	Less Than	IS_T($\mathbf{S}$) = IS_F($\mathbf{V}$)
021 (G)	Less Than or Equal To	$\mathbf{S} = 1$ **OR** IS_T($\mathbf{S}$) = IS_F($\mathbf{V}$)
022-200 (H-R)	*bad condition code*	*bad condition code*
201 (S)	Greater Than	$\mathbf{S} \neq 1$ **AND** IS_T($\mathbf{S}$) $\neq$ IS_F($\mathbf{V}$)
202 (T)	Greater Than or Equal To	IS_T($\mathbf{S}$) $\neq$ IS_F($\mathbf{V}$)
210 (U)	Unsigned Higher	$\mathbf{C} \neq 0$ **AND** $\mathbf{S} \neq 1$
211 (V)	Overflow Set	$\mathbf{V} \neq 0$
212 (W)	Positive	$\mathbf{S} \neq 2$
220 (X)	Carry Set	$\mathbf{C} \neq 0$
221 (Y)	Not Equal	$\mathbf{S} \neq 1$
222 (Z)	Never	Never

Operations

The emulator supports 24 operations, which are listed in Table 3 along with their associated opcodes.

Table 3:

Operation Code	Operation
000 (@)	HALT
001 (A)	NO_OP
002 (B)	CLEAR
010 (C)	LOAD
011 (D)	STORE
012 (E)	EXCHANGE
020 (F)	ADD
021 (G)	SUB
022 (H)	MOVE
100 (I)	AND
101 (J)	OR
102 (K)	XOR
110 (L)	FNOT
111 (M)	NNOT
112 (N)	TNOT
120 (O)	ISF
121 (P)	ISN
122 (Q)	IST
200 (R)	LSR
201 (S)	ASR
202 (T)	LSL
210 (U)	SHIFT_M
211 (V)	SHIFT_P
212 (W)	*bad operation code*
220 (X)	JUMP
221 (Y)	*bad operation code*
222 (Z)	*bad operation code*

HALT: Raises a HALT exception, causing the current program to stop executing. This should be considered the proper way for a program to terminate.

NO_OP: Will not perform any operation, other than to move the current program counter to the next instruction.

CLEAR: Sets the word of memory at the specified address to all zeros, clearing the memory location.

LOAD: Loads the word of memory at the specified address into the destination register.

STORE: Stores the contents of the destination register into memory at the specified address. The register specified by the **Dest** field is actually used as the data source in this instruction.

EXCHANGE: Loads the word of memory at the specified address into the destination register, while storing the original value of the destination register at the specified address in memory, effectively swapping the two values.

ADD: Adds together the values in the registers specified by **Arg1** and **Arg2**, storing the result in the destination register.

SUB: Subtracts the value in the register specified by **Arg2** from the value in the register specified by **Arg2**, storing the result in the destination register.

MOVE: Copies the value specified into the destination register.

AND: Tritwise AND the values in the registers specified by **Arg1** and **Arg2**, storing the result in the destination register.

OR: Tritwise OR the values in the registers specified by **Arg1** and **Arg2**, storing the result in the destination register.

XOR: Tritwise XOR the values in the registers specified by **Arg1** and **Arg2**, storing the result in the destination register.

FNOT: Tritwise FNOT the value in the register specified by **Arg2**, ignoring **Arg1**.

NNOT: Tritwise NNOT the value in the register specified by **Arg2**, ignoring **Arg1**.

TNOT: Tritwise TNOT the value in the register specified by **Arg2**, ignoring **Arg1**.

ISF: Tritwise ISF the value in the register specified by **Arg2**, ignoring **Arg1**.

ISN: Tritwise ISN the value in the register specified by **Arg2**, ignoring **Arg1**.

IST: Tritwise IST the value in the register specified by **Arg2**, ignoring **Arg1**.

LSR: Shifts all the trit values in the register specified by **Arg1** to the right by the specified number of trits, resulting in a division of the number by a power of three. The result is stored in the destination register. The trit value shifted into the most significant trit will be zero.

ASR: Shifts all the trit values in the register specified by **Arg1** to the right by the specified number of trits, resulting in a division of the number by a power of three. The result is stored in the destination register. The trit value shifted into the most significant trit will be two if the original value was negative, or zero otherwise, preserving the sign of the value.

LSL: Shifts all the trit values in the register specified by **Arg1** to the left by the specified number of trits, resulting in a multiplication of the number by a power of three. The result is stored in the destination register. The trit value shifted into the most significant trit will be zero.

SHIFT_M: Tritwise SHIFT_M the value in the register specified by **Arg2**, ignoring **Arg1**.

SHIFT_P: Tritwise SHIFT_P the value in the register specified by **Arg2**, ignoring **Arg1**.

JUMP: Puts the value of the register specified by **Arg2** into the PC register, causing the program execution to jump to that address in RAM. This operation has special logic associated with the flag mode of the instruction.

Flag Mode

This is a single-trit field in the instruction that denotes the flag mode for the instruction. There are three flag modes that can be used, the value this field takes for each mode is outlined in Table 4. 'No-Set' mode will prevent the instruction from affecting the CPSR register flags. 'Set' mode will allow for the instruction to set the flags in the CPSR register. 'Exclusive-Set' mode will allow for the instruction to set the flags in the CPSR register, while throwing out the usual result of the instruction, instead of storing it to the destination register.

Table 4:

S Field	Flag Mode
0	Result was No-Set
1	Result was Set
2	Result was Exclusive-Set

The jump instruction has speacial logic associated with this field. As the jump instruction cannot affect the CPSR flags, the regular flag modes are not applicable. Instead this value is used to determine how to handle capturing the current value in the PC register. If the value of this field is 0, the current location in code, before the jump, is not recorded, and Destination field is ignored. If the value is 1, the current location in code, before the jump, is stored in the register denoted in the Destination field. This is done to make it easier to implement functions by capturing where the function will return to. If the value of the field is 2, the jump will not affect the PC register, resulting in no jump in code, but will store the current value in the PC register in the register denoted in the Destination field.

Address Mode

There are 8 address modes, which are listed in table Table 5. The first 3 address modes, 00 through 02, are used by non-addressing operations, such as add or sub. The last 5 address modes are used by addressing instructions, such as ldr and str. The address mode dictates the value of the second argument to the operation, or the only argument to the operation for operations that only have one argument. For addressing operations, this value must be an address.

Table 5:

AM Field	Address Mode
00	Immediate
01	Register Direct
02	Register Direct Scaled
10	*bad address mode*
11	Addressed
12	Register Indirect
20	Immediate Offset
21	Direct Offset
22	Scaled Offset

Immediate: The value of the immediate field is used as the argument.

Register Direct: The value of the register indicated by Arg 2 is used as the argument.

Register Direct Scaled: The value of the register indicated by Arg 2 undergoes a shift operation, indicated by the instruction's shift mode, whose magnitude is the value of the immediate field. The result is used as the argument.

Addressed: The value of the immediate field is the address used as the argument.

Register Indirect: The address found in the register indicated by Arg 1 is used as the argument.

Immediate Offset: The sum of the address found in the register indicated by Arg 1 and the value of the immediate field is the address used as the argument.

Direct Offset: The sum of the address found in the register indicated by Arg 1 and the value of the register indicated by Arg 2 is the address used as the argument.

Scaled Offset: The value of the register indicated by Arg 2 undergoes a shift operation, indicated by the instruction's shift mode, whose magnitude is the value of the immediate field. The result is added to the address found in the register indicated by Arg 1, resulting in the address used as the argument.

Shift Mode

This is a single-trit field in the instruction that denotes the flag mode for the instruction. There are three flag modes that can be used, the value this field takes for each mode is outlined in Table 6. There are two address modes which perform a scaling operation, Register Direct Scaled and Scaled Offset. In these address modes the value of the register indicated by Arg 2 undergoes a shift operation whose magnitude is the value of the immediate field. The shift mode indicates which shift operation is performed during these instructions.

Table 6:

Sh Field	Shift Mode
0	Logical Shift Right
1	Arithmetic Shift Right
2	Logical Shift Left

Immediate Value

The last 11 trits of an instruction represent a numerical value embedded in the instruction. This field is further subdivided into a rotation field, which is the first 2 trits, and the value field, which is the remaining 9 trits. The number that the immediate represents is calculated by treating the value field as the least significant 9 trits in a 27-trit number. That number is then rotated to the right by 3 times the number specified in the rotation field. The resulting number may be interpreted as a signed or unsigned number depending on its use.