Telescope Need-To-Know

Apr 30, 2022

This is a compilation of all the important bits of information about the TASL language, TASL UDL, Telescope Computer, Glass CPU, and TBOS v1.0. I don't know what kind of document I want this to be, so some things might get covered extensively, while I might gloss over other details. I just want to get as much need-to-know information into one document as possible. Here's a quick description of the project:

What do we mean by the word "computer"? Images that come to mind might include a keyboard, mouse, and display of some kind, or perhaps the box that each of those is connected to, or maybe the hardware resting inside that box. At the opposite end, one also might think of the window manager, file system, or even browsers considering how digital things have become in the recent two years. This shows just how many different facets there are to what we consider a "computer" nowadays. So, while the direct answer of "a Turing Machine" might be technically correct, it by no means captures this immense complexity wrapped inside the (usually) intuitive UX of the modern computer system. This project is an attempt to peel back those layers of complexity, and scale the ladder of abstraction from the ground up. This not only includes making hardware like a CPU and computer, but also the software that makes the hardware easier to use, like an assembly language and a self-assembler for writing code. Finally, this involves writing an operating system to manage the computer's resources, to support multiprogramming, and even support multithreading. We often take for granted just how high "high-level" actually is.

Instructions:

ISA Instructions

These are the normal instructions that come as part of the ISA. These are all supported by the TASL language.

`Control:` `HLT, SYS, NOP, RSM`

`Memory:` `MOV A←B, LDI A←Word, LOD A←[B+Address], STR A→[B+Address]`

`Branching:` `JIF CND Address`

`ALU:` `ALU/ALF OP A←B, ALI/AIF OP A←I, AFN OP A B, AIN OP A I`

- HLT can act as a pause in programs.
- SYS triggers the SYS exception (taking a total of 3 clock cycles)

- RSM is the only instruction which can enable LO offsetting
    - RSM also jumps to EPC, and sets the current U/E to the previous.
- JIF (when true) and LDI take 2 clock cycles
- STR and LOD take 3 clock cycles
- In STR, if A=B, it behaves like a direct write to the address (this does **not** work for LOD)
- AIN also writes to flags (if it didn't, it would do nothing)

TASL Additional Instructions

While the standard ISA is more than sufficient to write any program with, it's not expressive enough. While we can just `MOV PC RA` and `AFN SUB RG0 RG1`, it would be much nicer if we could just `RET`urn and `CMP`are `RG0` with `RG1`. This is where these "additional" instructions come in, which are just expressive mnemonics for common operations.

`RET` - Set PC to RA
`ZRO A` - Set A to 0
`NUL A` - Set A to 0xFFFF
`PAS A` - Pass A through the ALU, setting flags.
`INC A` - Increment A, setting flags
`DEC A` - Decrement A, setting flags
`PSH I` - Subtract I from SP
`POP I` - Add I to SP
`CMP A B` - Compare A and B (for use with `JEQ/JNE/JLT/JGE`)
`CMI A I` - Compare A to a fixed value I (like CMP)
`JMP Addr` - Jump to Addr unconditionally (takes 2 cycles)
`JEQ Addr` - Jump to Addr if the previous CMP found that A=B (takes 2 cycles)
`JNE Addr` - The logical inverse of JEQ (takes 2 cycles)
`JLT Addr` - Jump to Addr if the previous CMP found that A<B (takes 2 cycles)
`JGE Addr` - The logical inverse of JLT (note this is **not** A>B) (takes 2 cycles)

`CAL Addr` - A unique macro instruction for function calls. Jumps to Addr unconditionally, and sets RA to the value after the CAL instruction. Takes 4 clock cycles to execute.

Operands & Modifiers

Registers

`RG0-7`
`RG6` is OUT
`RG7` is RA
`PC` - Program counter

`SP` - Stack pointer
`IO` - Currently selected I/O device
`EXA` - Exception handler address (jumped to on exception)
`EPC` - Address to return to after handling the exception.
`CTD` - Countdown timer (does not count down or signal CTDZ if E=0)
`HLB` - HI/LO bounds (HI is relative to LO)
HI **must** be >0 for RSM to not immediately trigger OOB.
`FLG` - (5) IOP, (3) EXC, (4) ALUF (Ovr, Car, Sign, 0), (4) U/E stack (prvU, prvE, curU, curE)

- `EXC` in order: `INT0-3, CTDZ, SYS, PRIV, OOB`.
- `IOP` = (3) Group #, (2) Device #
    - Groups in order: 0 Drives, 1 Drive Pointers, 2 Keyboard/Terminal, 3 GPIO
      4 TNET, 5 Network, 6 Expansion Slots, 7 Immutable.
    - Immutable: Cycles Lower/Stop, Cycles Upper/Start, Random/Status, BOOT
      Setting status triggers a breakpoint to let the user know the OS is running.

ALU Operations

```
NOT (~B)
AND
OR
XOR
SLB (logical shift A left by B)
SRB (^ right)
SET (set bit B in A to 1)
UST (set bit B in A to 0)
NEG (two's-complement of B)
ADD
ADC (add + carry)
SUB (A-B)
SBB (sub with borrow)
MUL (returns lower word of product)
DIV (unsigned division)
MOD
```

- Any operation which does not support a given flag will set that flag to 0.
- All arithmetic operations set the overflow bit if the sign of the output is different from what it should be (except DIV and MOD, which turn on overflow if dividing by 0).
- SRB and SLB turn on carry if any bits that were on are shifted over the edge.
- The carry of SET & UST is what bit B in A **was**.
- MUL turns on carry if the upper word of the product is non-zero.
- DIV sets carry if the remainder is non-zero, and MOD does if the quotient is non-zero.

Jump Conditions

```
Format: Condition/LogicalInverse - Condition description
EQZ/NEZ  - equal to zero
LTZ/GEZ  - less than zero
CAR/NCR  - carry
OVR/NOV  - overflow
PRD/PNR  - current I/O device is reading
PWT/PNW  - current I/O device is writing
TRU      - unconditional true
RND      - random chance to jump
```

Telescope Computing System

As of right now, here is what makes the telescope computing system different from the CPU:

Boot Loader

At the heart of the TCS is the bootloader machine, which serves as a shell around the CPU and removes access to IOP device 31. This means that the TCS's "OS Adr" and "OS Data" should be connected to ROM with a **loadable** file in it.

Interrupts

The interrupts on the standard CPU are not assigned to anything in particular, but they are in the TCS.

0 - TICK: interrupt occurs once every 2^16 cycles using computer's cycle counter
1 - PAUS: somewhat like ctrl+alt+del. Should bring the user to some kind of UI
2 - PVIO: non-maskable, user program attempted to read/write privileged I/O device.
3 - STOP: non-maskable, stop computer right where it is. Like HLT, but at the OS level.

IMPORTANT: TICK is currently not configured.

I/O Devices

The TCS splits I/O devices into 8 groups of 4.
This means the top 3 bits of IOP are the group number, and the bottom 2 are the device number
Device number 0 for all groups is restricted to kernel processes only.

0 - Drives
    Drives should have 1 word addressing, with 1 word data
1 - Drive Pointers
    The address of each corresponding drive.
    Increments on each read/write to the corresponding drive.
    When written to, sets the drive pointer to the given value.

2 - Terminal/Keyboard

Input is from the keyboard

Output is to the terminal

IOW signal is from keyboard

3 - GPIO

General purpose I/O, like on an Arduino or Raspberry Pi

Provides raw access to the signals the CPU sends and receives from I/O

4 - TNET

Specially created I/O device for primitive networking.

Contains two registers, one facing outwards and one facing inwards

Each register has a corresponding R/S flip-flop which is on if it contains data

Flags:

writeIN (to CPU)	if inbound full
readOUT (from TCS)	if inbound empty
readIN (to CPU)	if outbound empty
writeOUT (from TCS)	if outbound full

In other words: If it has data it's writing, and if it doesn't it's reading.

The pins are configured in such a way that 2 computers can connect their TNET I/O ports to communicate. When this happens, the inbound register is bypassed so that, in each direction, there is only 1 register between each computer.

5 - Network (unused for now)

6 - Internal Expansion (unused for now) (kernel only)

7 - Immutable (kernel only)

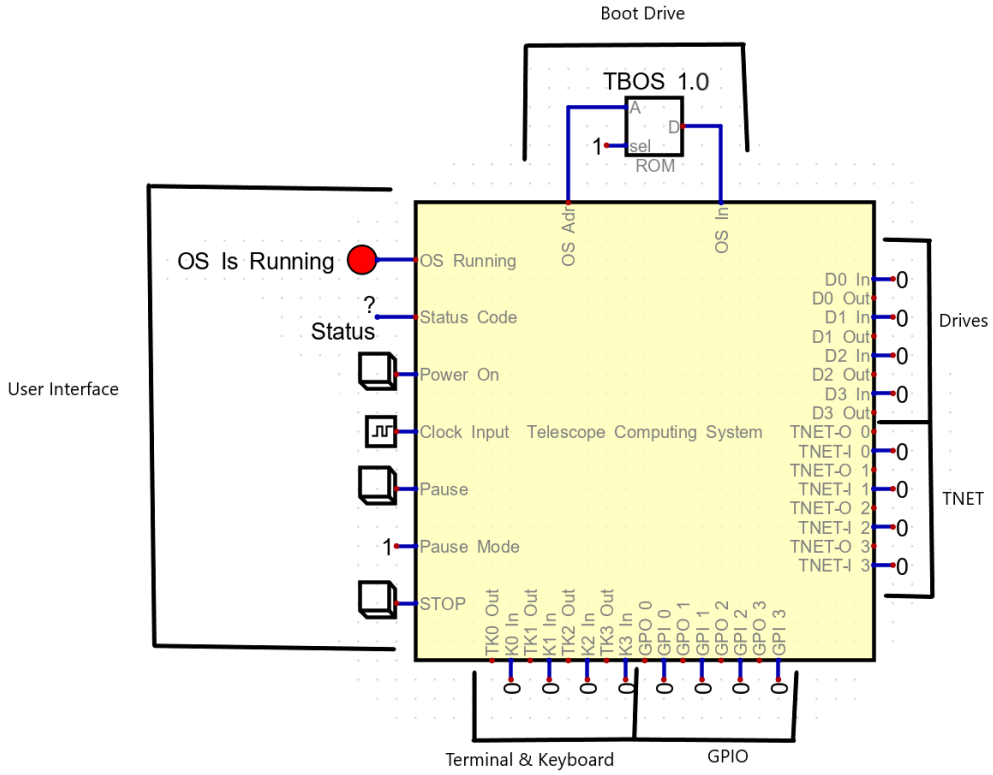
0 - Cycles counter (read lower word) / Start counter (write)

1 - Cycles counter (read upper word) / Stop counter (write)

2 - Random number generator (read) / PC status code register (write)

3 - External boot (unusable by any program)

Physical Interface



Each device has it's own pair of inputs and outputs (except drive pointers, which provide the address for drives)

Note that, in Digital, all inputs need to have some value (hence all of the 0's)

If "Pause Mode" is 1, "Pause" directly pauses the CPU. If "Pause Mode" is is 0, "Pause" triggers the PAUS interrupt

Finally, STOP triggers the STOP interrupt

TASL Features, Syntax, and Style

Example Program

```
JMP >Init
(% An example TASL program

   This is an example of what a TASL program looks like.
   Hopefully from this, you can get a rough understanding of TASL.
   Take note of the style choices, some are really important.
%)

(/ ------- Data -------

@MY_CONSTANT = 15
@MY_OTHER_CONSTANT = 0xF0
@MY_CONST_CHAR = ‘c’
@MY_NEWLINE_CHAR = ‘
‘

@TERM_FLG = 0x4000

#MyVariable = 0b010111011
#MyOtherVar = 583

#MyString = “Hello World”  / Null terminated in memory
#MyEmptyList = (~53)  / Must be a decimal number after ~ (for now)

#MyList = (23, 0x8A, ‘c’, ~20, “HI”, 8)  / Lists can be anything

@MY_HACK_CONSTANT = >MY_CONSTANT  / You can also assign constants to others
)

/ ------- Code -------

(#Init  / In this program, init prints MyString and calls MyFunction
      / Initializing the registers
      NUL RG1; ZRO RG0; LDI FLG >TERM_FLG

      / The loop
      #I.StringLoop
            LOD RG2 RG0 >MyString         / Load MyString[RG0] into RG2
            CMP RG2 RG1; JEQ >I.Done      / Compare this with RG1(NULL)
            MOV IO RG0                    / If not equal, write to I/O
            INC RG0; JMP >I.StringLoop    / And loop
      #I.Done

      / Now that we’re done, call MyFunction
      CAL >MyFunction; MOV IO OUT
```

```
      HLT
)


(#MyFunction  / MyFunction returns the 2nd character from MyList
      LDI RG0 2; LOD RG1 RG0 >MyList
      MOV OUT RG1; RET
)
```

How TASL Works

So you might be wondering how an assembly language supports this level of complexity, and the answer may or may not surprise you: It doesn't! Most of what you see above is completely ignored by the assembler, like equals signs, semicolons, commas, parentheses, etc. Since this is important for making labels, here is the list of important characters highlighted:

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Another important thing to note is that each time you type a number, character, reference (like >MyLabel), empty list, or string, the value is taken and placed directly where you typed it in memory. For instance, I could do this:

```
LDI RG0 “”
```

And the assembler would load null into RG0. If I actually typed a string, it would try to execute the second character of the string as an instruction after loading the first into RG0, because like I said, it places what you type directly where you type it into memory. This is why the jump to Init at the beginning is so important, because it skips over all of the "variable" definitions.

The only exception to this is constants, which do not follow this rule. A constant is basically a label that you can define yourself, which means that its value is not placed into memory anywhere. Constants are only visible from within the assembler, and their references, like all other label references, are replaced by their value when assembled.

TASL Style

So now comes the big question: Why are you allowed to do this?

```
JMP<>Init?@MY_CONST[23]#Init;LDI=RG0;>MY_CONST]]]MOV,IO[RG0[HLT
```

I will leave the function of this program as an exercise to the reader, but the reason why is twofold:

a) It's really easy to make an assembler whose only understanding of lexemes is important vs unimportant characters.
b) I want to leave as much room for stylistic choices as possible.

While it can be incredibly difficult to read if done improperly, anyone who is really trying to use this language shouldn't come up with a bad style, because there would be easier ways to do things. While beginners, including myself, will have an inconsistent and not very good style, it won't be terrible so long as they are actually trying to use the language to write good code. Speaking of style choices, here are some important ones:

- Use parentheses to indicate collapsable sections of code, and only use them for functions, data, and big comments. The UDL supports this.
- A sequence of 4 or less instructions, which work together to perform a higher-level task, should be on the same line and separated by semicolons.
- Use Pythonic comments and indentation.
- Always put a multi-line comment at the beginning of your program describing it (and probably make it collapsable, because it should be a few lines long).
- Use equals signs when assigning values to variables and constants for easier readability.
- If you have a function `MyFunctionHere`, any labels within this function should be called `MFH.MyLabel`. This is to prevent you from accidentally naming labels the same thing.

But, like in every language:

Be consistent.

Setting Up the Mini-IDE

Here's a how-to guide for setting up the mini-IDE that I have put together:

To get syntax highlighting

1. Run Notepad++, and go to "Language" →"User Defined Language" → "Define Your Language"
2. Select "Import," then load the "TAS Light v3 UDL.xml" file.
3. Now close and re-open Notepad++
4. You should see "TAS Light v3" under "Language" now towards the bottom. Click on that.

To make assembling, compressing, and file encoding easier, here's how to add buttons:

1. Go to "Plugins" → "Plugins Admin"
2. Search for and install "NppExec" and "Customize Toolbar" by checking the box next to them and pressing the "Install" button on the top right.
3. Now copy the "Notepad++ Config Files" from the project into %APPDATA%\Notepad++\plugins\config
4. Inside of the "npes_saved.txt" file, there are two types of lines that begin with "set" after the lines starting with "::". ASSEMBLER_PATH should be followed by the path to the TestingAssembler.py script, and FILE_ENC_PATH should be followed by the path to the FTLV1.py script.
5. Now make those changes, save the file, and restart notepad++

Now you should have access to the buttons inside of the customize toolbar plugin.

What the Scripts Do

The FTLV1 script, as was mentioned previously, takes a file as an input (or the current file you're working on if you click the button) and converts it into ASCII numbers in a Logisim drive format. Digital just so happens to use the same format for its drives, except they require .hex files.

The TestingAssembler script is meant to speed up development, but it was also meant to ensure that the self-assembler would run. You'll notice if you run the TestingAssembler that it mentions running a simulation, because that's exactly what it does: it simulates the self-assembler in Python. However, over time new features were necessary to speed up development, so there are a few extra things it supports that the self-assembler doesn't. For instance:

- By specifying -c after the file name, it will run "lexical compression" which produces a much smaller file that will assemble to the same machine code. This is useful for running files through the self-assembler quickly. It's also useful for combining multiple files together to simulate importing.

- By specifying -d after the file name, it will produce multiple files. It will make a .tlo.hex file like usual, but it will also make a .rtl file and a .table file.
    - The .table file is the final label table that the assembler produces internally, so you can see where everything is in the machine code relative to each other.
    - The .rtl file is like a translated version of the machine code, where RAM address x corresponds to RTL line x+2.

The Operating System

The Telescope Basic Operating System version 1.0, or TBOS v1.0, is a really simple OS as of right now. It only supports a few different things:

- Running root as a user process from drive 0
- Running 16 processes at once (this is configurable, though)
- Switching to the next process after the CTD countdown timer reaches 0.
- Loading programs from the start of a given drive as child user processes.
    - After doing a syscall, the OS checks the FLG of the process and, if it's looking at a drive device that is not drive 0, it will load the program from the start of that drive.
- Terminating running processes and their children.
    - Any unsupported exception will cause the process to be terminated. This means that HLT will behave somewhat as expected and terminate the process.

TBOS 1.0 is meant to be a proof-of-concept that future versions will build off of to achieve the final additional goals:

- Multiple syscalls
- Loading programs from anywhere in a given drive.
- Spawning child threads with given properties
- Support for both custom semaphores and I/O semaphores
- Simulated I/O device interrupts & wait for I/O
- Support for all 4 CPU interrupts
- Support structures for processes