Telescope Need-To-Know

Apr 6, 2022

This is a compilation of all the important bits of information about the TASL language, the Telescope Computer, and the Glass CPU. This assumes that you already have some knowledge of the whole project, so a lot of details are going to be glossed over here. This is meant for people who want to program on the system that already have a good idea of what they're doing, and just need a quick reference.

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, (4) U/E stack (cur/prev)

- `EXC` in order: `INT0-3, CTDZ, SYS, PRIV, OOB`.
  - INT2 and INT3 are non-maskable
- `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
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 1 bits are shifted over the edge.
- The carry of SET & UST is what bit B in A **was** (the old SEL instruction).
- 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
```

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 definitions
@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 ~

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

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

#Object
      #Object.Prop1 = 23
      #Object.Name = “Hello”
      #Object.Prop2 = ‘F’
)



(#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 RG0 RG0 >MyString
            CMP RG0 RG1; JEQ >I.Done
```

```
            MOV IO RG0
            INC RG0; JMP >I.StringLoop
      #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. Each time you type a number, character, reference (like >MyLabel), empty list, and string, the value is taken and placed directly where it is 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, 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 that.

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, the whole motto of TASL is "Why would you do it bad?" It does nothing but make your job and other people's jobs more difficult. Sure there are good and bad style *choices* that people make, but no one honestly using this will end up with an overall bad style, because it would just be hard to use. 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.
- 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.
- Use equals signs when assigning values to variables and constants
- Use colons & indentation when making objects like from the previous section.
- 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.