The Manual of Tak's Spreadsheet Assembler for Tak's Toy Processor

# Copyright, copyleft, open source or just freeware?

This manual, the spreadsheet assembler and the toy processor design in Logisim are all licenced under [GPLv2](https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html). This essentially means the material is copyrighted. However, the licence also grant everyone the right to use the material and derive work from it. For derived work, however, the original copyright must be retained while adding the copyright of the party performing the derivation. GPLv2 is contagous, meaning that all derived work must also be licenced under GPLv2.

So formally:

**Copyright (C) 2016,2017,2018 Tak Auyeung  
  
This program is free software; you can redistribute it and/or  
modify it under the terms of the GNU General Public License  
as published by the Free Software Foundation; either version 2  
of the License, or (at your option) any later version.  
  
This program is distributed in the hope that it will be useful,  
but WITHOUT ANY WARRANTY; without even the implied warranty of  
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
GNU General Public License for more details.**

# How to use

This section is intended for people who want to make use of this assembler.

## What is it?

Tak's Toy Processor (TTP) is a simplistic RISC processor implemented in Logisim (version 2.7.1). It has a simple assembler that is implemented as a spreadsheet. The assembler allows the use of mnemonic notations, label definitions to generate the actual opcodes and RAM content of a program.

## Syntax

The syntax of the assembly language is pretty simple. The following describes what you can do with this assembler.

The assembler is line-based. This means it is *not* like C/C++ where a newline is viewed merely as a space. Each line can be one of the following:

* empty except for optional comments
* instruction that generates code
* label definition: same restriction as a C/C++ variable name, followed by a colon

Comments can be used on any line. Content after "//" is ignored.

In these descriptions, x and y are registers, and i is an immediate value. A register can be a,b,c or d. An immediate value can be a literal base-10 unsigned integer or the reference of a label.

### Label Definitions and Postfix Expressions

To define a label, spell out the name of the label and place a colon (:) immediately at the end without any space between the name of the label and the punctuation.

Note that label definitions can utilize postfix expressions to specify the value of a label. A label definition that is "empty" is defined to the address of the current location:

L1:

However, a label definition can also utilize a postfix expression:

L2: 2 4 + // L2 is define to 2+4=6

A postfix expression can involve the use of labels, as long as there is no circular definition:

L3: L1 + L2

A postfix expression may use a dot (.) to refer to the current location:

L4: . 3 + // L4 is the symbolic name of 3 bytes after the current address

Postfix expressions can also be used to specify an immediate operand, for example:

ldi a, L2 L3 + 2 - // a is L2+L3-2

### Instruction Mnemonics

The assembler is case-insensitive, so be careful with labels! Instructions that are recognized are as follows:

* utility type
  + halt // for(;;);
  + nop // {}
* ALU type (affect flags)
  + add x,y // x+=y, updates CZSOL
  + sub x,y // x-=y, updates CZSOL
  + cmp x,y // x-y, updates CZSOL
  + and x,y // x&=y, updates ZSL
  + or x,y // x |= y, updates ZSL
  + not x // x=~x, updates ZSL
  + rsh x,y // x>>=y, updates ZSL
* ALU type (no change to flags)
  + inc x // ++x
  + dec x // --x
* copying type
  + ldi x,i // x=i; actually x=(\*PC++)
  + st (y),x // \*y=x
  + ld x,(y) // x=\*y
  + cpr x,y // x=y
* jump type
  + jmp x // PC=x
  + jmpi i // PC=i; actually PC=(\*PC)
  + jc x // PC=c?x:PC
  + jci i // PC=c?i:PC; actually PC=c?(\*PC):(PC+1)
  + jz x // PC=z?x:PC
  + jzi i // PC=z?i:PC; actually PC=z[[1]](#footnote-0)?(\*PC):(PC+1)
  + js x // PC=s?x:PC
  + jsi i // PC=s?i:PC; actually PC=s?(\*PC):(PC+1)
  + jo x // PC=o?x:PC
  + joi i // PC=o?i:PC; actually PC=o?(\*PC):(PC+1)
  + jl x // PC=l?x:PC
  + jli i // PC=l?i:PC; actually PC=l?(\*PC):(PC+1)
* memory allocation type
  + byte i // allocates and initializes a byte in RAM

# Basic architecture

To understand the architecture, it is best to review the Logisim files and find out how it is implemented. From the *software* perspective, this is a quick summary.

The architecture has four general purpose registers that can be specified in some instructions. These four registers are simply designated as 'a', 'b', 'c' and 'd'. Each one can be used as register x and/or register y of an instruction.

The program counter (PC) is a register that is not directly specifiable by instructions. The program counter stores the address of the next instruction to execute. As a result, changing the PC changes the location of the next instruction. This is accomplished by the unconditional branch 'jmp' instruction. The 'jmp' instruction has one register operand that points to the location to continue execution.

The ALU implements arithmetic operations. The result of such operations overwrites certain flags that are not directly software accessible.

* The Carry flag is overwritten only after 'add', 'sub' and 'cmp'. It is 1 iff the result of the 'add' ends with a carry, or the result of 'sub' or 'cmp' ends with a borrow.
* The Zero flag is overwritten after any ALU operation. It is 1 iff the result of an operation is 0.
* The Sign flag is overwritten after any ALU operation. It is the most significant bit of the result of an operation.
* The Overflow flag is overwritten only after 'add', 'sub' and 'cmp'. It is 1 iff the result of the instruction has a sign that does not make sense.
* The Less-than flag is overwritten after any ALU operation because it is the exclusive-or of the Sign and Overflow flag.

These flags are used in the conditional branch instructions 'jc', 'jz', 'js', 'jo' and 'jl'. Each of these instructions has a register operand. These conditional branch instruction continues execution at the location pointed to by the register operand iff the flag specified is a 1. For example, 'jc a' continues execution where register 'a' points to iff the Carry flag is a 1. Iff the flag specified is a 0, execution continues with whatever instruction is right after the conditional branch instruction.

The other family of conditional branch instructions end with "i" like "jci". This family of instructions uses an immediate value after the opcode to specify where to branch to.

# How to assemble

Using the assembler is relatively easy. Editing the text of source code is best done in a plain text editor such as Notepad in Windows or Mousepad/vi in Linux. In a GUI editor (use gvim instead of vim for this purpose), copy the program to the clipboard. In sheet "source" of the assembler spreadsheet, go to cell A1 and paste.

This should paste the source code so that each line in the text file copies to one cell in column A. Currently, the spreadsheet is hard coded so that only the first 100 rows are used.

To get the output, go to sheet "output" of the "assembler" spreadsheet. This sheet displays the memory content. Column A indicates the start address, column B and C indicate the content of memory. Everything in the "output" sheet is in hexadecimal. This content must be hand entered into the RAM content editor of Logisim.

As of 2016/12/08, the assembler now checks and reports certain types of errors. Errors are indicated in column B of the "source" sheet. When at least one error is presented, the first row of "RAM file" indicates that. So check "RAM file" first before getting the RAM content!

# Implementation

This section is for the curious kind. Skip this if you only want to know how to use the assembler!

The "mnemonics" sheet defines the instructions of the processor.

* Column A specifies the name of each mnemonic.
* Column B is the "base opcode" corresponding to the mnemonic (if any). Note that "byte" has no opcode because it is used only to reserve a byte.
* Column C indicates the position of register "x" as an operand. 0 means the first, and 1 means the second.
* Column D indicates the position of register "y" using the same convention.
* Column E indicates the position of immediate constant.
* Column F specifies how to recognize register x (the regular expression in un-escaped parentheses), and
* column G specifies how to recognize register y.
* Column H is the amount of bits to shift for register x as a part of the opcode,
* Column I is the amount of bits to shift for register as a part of the opcode.

The "heavy lifting" of assembling is done in the "assemble" sheet.

* Column A is the cleaned up version of each line, removing comments and empty spaces before and after instructions.
* Column B marks labels being defined (if any)
* Column C looks up the mnemonic table and specifies the index (as row number) into the mnemonics sheet for the instruction of that line.
* Column D extracts the operands of an instruction.
* Column E is the first operand
* Column F is the second operand. Note that both columns E and F are the result of the formula of column E.
* Column G looks up the mnemonic sheet to find the position of register x
* Column H is the operand containing register x
* Column I looks up the mnemonic sheet to find the position of register y
* Column J is the operand containing register y
* Column K is register x
* Column L is register y
* Column M is register x shifted by the bit position amount
* Column N is register y shifted by the bit position amount
* Column O is the base opcode
* Column P integrates the base opcode with the bits to specify register x and register y
* Column Q is the position of immediate constant
* Column R is the text of the immediate constant
* Column S resolves a label reference to its actual value
* Column T counts the number of bytes of the instruction
* Column U repeats the label being defined on that row
* Column V is the actual start address in RAM for that row
* Column W is the address like column V but in hexadecimal
* Column X repeats the opcode with register specification
* Column Y is the immediate value

Notable special features used in the assembler include the following:

* Lookup functions in a spreadsheet like MATCH, VLOOKUP. These function uses a cell to specify a key to locate it in a range of cells. MATCH returns the row number of the found item in the range of cells, whereas VLOOKUP is a short hand to specify the "value" column corresponding to the found item.
* Addressing and dereferencing cells. ROW() and COLUMN() can be used to discover the row and column number of a cell. ADDRESS is used to formulate a cell specification dynamically, and INDIRECT makes use of the text of a cell specification to access the content of a cell. Look at INDIRECT as dereference in C++.
* Regular expression. Regular expression is somewhat supported in Google Sheets. This allows the matching, extraction and/or replacement of parts of a string.
* Conditional formulae. This is the same as ternary operators in C/C++!

1. [↑](#footnote-ref-0)