Testing for oneself brings more answers than a thousand theoretical questions.

No questions allowed

Abstract

Innovation in programming relies on the ability to transcend the limitations of existing languages. While these languages are powerful in terms of expressiveness, they can constrain developers creativity by imposing rigid structures. This note proposes the creation of an assembly language based on Presburger arithmetic, a mathematical theory that provides a decidable framework for manipulating integers.

Processor Architecture

Pipeline Cycle. Each instruction follows a processing cycle of up to five steps, according to the following scheme :

1. `IF (Instruction Fetch)` - Loading the instruction from memory.
2. `ID (Instruction Decode)` - Decoding the instruction to identify operands and registers.
3. `EX (Execution)` - Performing arithmetic, logical, or other operations.
4. `MEM (Memory Access)` - Accessing memory to read or write data.
5. `WB (Write Back)` - Writing results into registers.

Registers. The processor has 5 unsigned four-byte (4-byte) registers : R_0, R_1, R_2, R_3, R_4, as well as several special registers :

— `pc (Program Counter)` - Register pointing to the currently executing instruction.
— `sp (Stack Pointer)` - Register pointing to the top of the stack.
— `lr (Link Register)` - Register for storing the return address.

Stack. The processor has a P stack bounded by 32 elements of four (4) unsigned bytes each.

Memory Model

The memory visible to an assembly program is segmented into two distinct parts : read-only memory (ROM) and write-only memory (RAM). Memory is addressed using unsigned integers of four (4) bytes. The following keywords are used :

— `MEM (memory)` - An array of bytes representing the read-only memory (ROM).
— `MAP (mapping)` - An array of bytes representing the write-only memory (RAM).

Instruction Set

Instruction	*Description*	*IF*	*ID*	*EX*	*MEM*	*WB*
Stack						
`PUSH Rx`	$P[sp] \leftarrow R_x; \quad sp \leftarrow sp+1$	x	x		x	
`POP Rx`	$R_x \leftarrow P[sp]; \quad sp \leftarrow sp-1$	x			x	x
`TOP Rx`	$R_x \leftarrow P[sp]$	x			x	x
`SWAP`	$P[sp] \leftrightarrow P[sp-1]$	x				x
Arithmetic						
`ADD Rx, Ry, Rz`	$R_x \leftarrow R_y + R_z$	x	x	x		x
`MOV Rx, Ry`	$R_x \leftarrow R_y$	x	x			x
`INC Rx`	$R_x \leftarrow Rx + 1$	x	x	x		x
`ZER Rx`	$R_x \leftarrow 0$	x				x
Memory						
`LDR Rx, Ry, Rz`	$R_x \leftarrow MEM[R_y : R_y + min(R_z, 4)]$	x	x	x	x	x
`STR Rx, Ry`	$MAP[R_x : R_x + 4] \leftarrow MAP[R_x : R_x + 4] \mid R_y$	x	x	x	x	
Control						
`JLE Rx, Ry, label`	$pc \leftarrow$ label if $x \leq y$	x	x	x		x
`CALL label`	$lr \leftarrow pc; \quad pc \leftarrow$ `label`	x				x
`RET`	$pc \leftarrow lr$	x	x			x
`NOP`	*Does nothing*	x				

Thus, the goal is to demonstrate that the minimal assembly language presented is powerful enough to express a calculation that allows the assembly of two-dimensional blocks using an instruction manual.

There are two formats for the instruction manual that the assembler program can interpret in language L' :

1. *First format*
   - `1st byte` : width (4 bits) and height (4 bits) of the piece.
   - `2nd byte` : horizontal (4 bits) and vertical (4 bits) offset of the piece.
   - `3rd byte` : color of the piece.

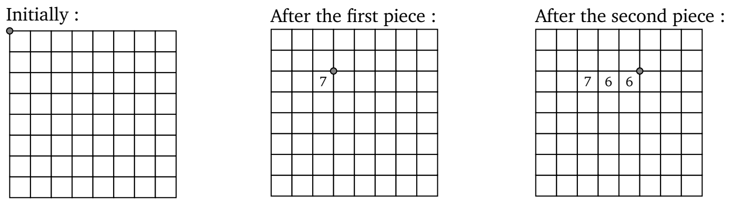
   For example, `MEM`:=$[0x11, 0x22, 0x07, 0x21, 0x10, 0x06]$ can be translated as :
   1. Place a 1×1 piece with a displacement of $(2, 2)$ and color $0x07$.
   2. Place a 2×1 piece with a displacement of $(1, 0)$ and color $0x06$.
2. *Second format*
   - `1st byte` : width of the piece.
   - `2nd byte` : height of the piece.
   - `3rd byte` : horizontal offset of the piece.
   - `4th byte` : vertical offset of the piece.
   - `5th byte` : color of the piece.

   For example, `MEM`:=$[0x01, 0x01, 0x02, 0x02, 0x07, 0x02, 0x01, 0x01, 0x00, 0x06]$ can be translated as :
   1. Place a 1×1 piece with a displacement of $(2, 2)$ and color $0x07$.
   2. Place a 2×1 piece with a displacement of $(1, 0)$ and color $0x06$.

The assembler program must be able to read an instruction manual (found in MEM) in one of the two formats and assemble the blocks to form a two-dimensional construction.

The positioning follows a cumulative logic : initially at $(0, 0)$, the top-left corner, each piece is placed relative to the previous one. Its anchor is set at its top-left corner. To make this decidable, the size of the grid used to place the pieces (`MAP`) is set to 8×8. When a displacement causes an overflow of the grid, the program must reposition its anchor to position 0 in that dimension. That is, if a piece is displaced by $(4, 3)$ and we are at position $(5, 6)$, the piece will be placed at position $((5 + 4) \mod 8, (6 + 3) \mod 8) = (1, 1)$. This same logic applies when part of the piece exceeds the grid. The final anchor point after placing a piece is given by the following formula : $((p_{ix} + w + dx) \mod 8, (p_{iy} + h + dy - 1) \mod 8)$ where p_{ix} and p_{iy} are the coordinates of the top-left corner of the initial anchor point, w and h are the width and height of the piece respectively, and dx and dy are the horizontal and vertical displacements of the piece.

The illustration below shows the evolution of the grid after placing the pieces for the memory `MEM`:=$[0x11, 0x22, 0x07, 0x21, 0x00, 0x06]$ (first format) :



The goal is therefore, in the first phase, to translate an instruction manual into a construction in the grid `MAP`. In the second phase, we conjecture that there are several arrangements of instructions that allow the creation of a minimal assembler program L' capable of performing this task. The program should therefore be optimized to use as few cycles as possible before completing construction.

Debugging Tool

A rudimentary tool has been put in place to assist in the design of programs in assembly language. This tool allows you to visualize the state of the program at each step of its execution. This program, called *asm*, is used as follows :

```
./asm [-p <path>] [-m <path>] [-b breakpoints]
```

Where

— `-p <path>` : specifies the path to the assembly program to be executed.

— `-m <path>` : specifies the path to the instruction manual.

— `-b breakpoints` : specifies the breakpoints separated by commas. Line numbers referring to the assembly file should be written.

Example of usage : `./asm -p program.asm -m manual.txt -b 3,6,42`

Program Visualization. The debugging program consists of six distinct areas as well as six commands to interact with the program. Below are tables listing the different available areas and commands :

Command	*Purpose*
`q`	Quit the program.
`i`	Change the selected area.
`w`	Scroll up the selected area.
`x`	Scroll down the selected area.
`s`	Execute an instruction.
`c`	Continue execution until the next breakpoint.

List of commands to interact with the debugger.

Area	*Description*
`Instructions`	List of program instructions.
`Processor`	Information about the stack and registers.
`Memory`	Content of the ROM memory (instruction manual).
`Map`	Content of the RAM memory (construction grid).
`Pipeline`	Status of the execution pipeline.

List of areas displayed by the debugger.

For the `Map` area, the colors will be represented not by hexadecimal values, but by their equivalent color.

$0 \Rightarrow$ White, $1 \Rightarrow$ Blue, $2 \Rightarrow$ Green, $3 \Rightarrow$ Cyan, $4 \Rightarrow$ Red, $5 \Rightarrow$ Magenta, $6 \Rightarrow$ Yellow, $7 \Rightarrow$ Black

Instruction Manuals. Different instruction manuals are provided to test your programs. Here is how they are structured :

— *First line* : manual format number (1 or 2).

— *Following lines* : each line contains an instruction in the form of `<width> <height> <horizontal_offset> <vertical_offset> <color>`.

It can be assumed that the instructions have valid and minimum values. For example, a horizontal offset will never exceed the width - 1 of the grid.

Ranking

Let the set of tests provided by default for the project be as follows :

— `manuals_f1/` : folder containing the instruction manuals with the first format for the instructions.

— `manuals_f2/` : folder containing the instruction manuals with the second format for the instructions.

❗ Only one (1) file per instruction format will be used for the tests. Since there are two formats, you will need to submit two files. If multiple (more than two) files are submitted, one file will be randomly chosen for each format.

Scoring. For each of the manuals (from the `manuals_f1` and `manuals_f2` folders), a test will be performed to verify that the program gives the expected result (drawing in the `MAP` grid).

Point allocation for each manual :

— If it is the correct expected result :
The team that implemented the solution using the least number of clock cycles will receive one (1) point for the `m1_x` manuals, two (2) points for the `m2_x` manuals, and so on. The team that implemented the slowest solution will receive three-quarters (0.75) of a point less than the maximum, i.e., 0.25 for the `m1_x` manuals, 1.25 for the `m2_x` manuals, and so on. Other teams that succeeded will have their score distributed linearly between these two bounds.

— Otherwise :
A score ranging from zero (0) to one-tenth (0.1) of a point will be awarded based on the progress of the construction.

— If the program is hardcoded :
A score of 0.25 will be awarded.

The final score will be the sum of the points from each manual.