CKONE: A MINIMAL TTK-91 SIMULATOR

Lassi Kortela
Programming in C project (Fall 2011)
University of Helsinki

Overview

Ckone is a command-line-driven simulator for TTK-91, a simplified computer architecture designed for teaching purposes at the University of Helsinki. The canonical TTK-91 simulator is Titokone, a more mature package written in Java featuring an assembler and a graphical front-end. Titokone was used as a reference when writing Ckone.

Supported platforms

Ckone is written in almost bare ISO C99 and should therefore compile on a wide variety of platforms. However, see the section ”Limitations on simulator implementation” for some caveats.

Ckone has compiled without warnings and successfully run the TTK-91 programs in the examples/ directory on the following platforms.

OS	OS version	Compiler	Computer
Mac OS X	10.6	Xcode gcc 4.2	x86_64
Linux	2.6	gcc 4.4	x86_64
Linux	3.0	gcc 4.6	x86

How to build

Open a shell prompt at the ckone directory and type make. To use a compiler other than gcc you need to edit the Makefile.

How to use

Open a shell prompt at the ckone directory and type ./ckone program.b91 where program.b91 is the file name of any pre-assembled TTK-91 program. Example programs are provided in the examples/ directory.

The following command line options are available.

-h provides a short usage summary.

-v enables verbose mode, in which detailed information about simulator state is emitted; see the section ”Outline of simulator operation” for details.

The simulated program may do input and output on the standard input and standard output streams. When the simulator prompts ”Input: ”, type in a signed decimal integer and press Enter. Output is displayed as a signed decimal integer prefixed by ”Output: ”.

Outline of simulator operation

The simulator first loads a program by parsing a .b91 file that describes a pre-assembled TTK-91 program. Such files are generated by assembling .k91 source files in Titokone.

The simulator keeps track of the simulated computer's entire memory in a single array of words.

The initial size of the memory is zero; it is expanded as needed. The loader fills in the memory such that program code is loaded at offset zero, followed immediately by the data area which contains initialized variables. Later, when simulation starts, a stack is established immediately following the data area.

Before commencing simulation a disassembly of the entire program code is written to standard output if verbose mode is enabled. Then program execution starts at address zero. The fetch-decode-execute cycle of the CPU is simulated. In the fetch phase the next instruction word is read from the memory location pointed to by the TTK-91 PC register and stored in the TTK-91 IR register. The decode phase uses a routine shared with the disassembler. In verbose mode each instruction is also disassembled to standard output as it is decoded. This makes it possible to follow the execution of the program and was a crucial debugging aid in developing the simulator itself.

The execute phase is done via a big switch table with one case per instruction, keyed on the opcode decoded from the instruction word. Before opcode-specific operations the TTK-91 temporary register TR is set to the operand of the instruction, decoded from the instruction word and retrieved using memory fetches if a memory or indirect operand was specified.

Simulation ends when the program either issues the HALT supervisor call or does something that causes an exception; all exceptions are fatal. After a HALT the values of all data variables are displayed on standard output if verbose mode is enabled. This is done heuristically by traversing the symbol table looking for symbols whose values are valid addresses in the data area then displaying the data values at those addresses.

The supervisor calls as well as the input and output ports of the simulated computer are mapped such that the simulated program may do input and output on the simulator's standard input and standard output streams. Inputs and outputs are represented to the user as signed decimal integers.

Testing

The program was tested by running the programs in the examples/ directory. The output was verified against Titokone on the same input. The output was also checked for consistency across the different platforms that Ckone was tested on.

Limitations on simulator behavior

There is no stepwise debugger.

One cannot set breakpoints.

Not all supervisor calls are implemented.

Error messages are rudimentary, especially in the parser.

Neither the parser nor the disassembler validate instructions; only the simulator does.

The disassembler displays some instruction forms differently from their canonical assembler syntax.

The disassembler does not heuristically substitute symbols for instruction operands.

Limitations on simulator implementation

The C size_t type is assumed to be at least 32 bits wide.

The C ssize_t (signed size_t) type is assumed to be available in <unistd.h>.

It is assumed that C uses two's complement representation for signed integers.

The simulator's integer wraparound behavior on 64-bit machines has not been tested.

GCC is passed -Wno-unused-parameter to muffle some spurious warnings.

Notes on C style

Since this is a C programming course, some commentary on coding style is in order.

I follow Rob Pike's recommendation that ”include files should never include include files”. I find that this makes it easier to understand the inclusion hierarchy and obviates the need to protect each include file against being included multiple times. I find that the extra effort to list all pertinent include files and put them in the proper order in each .c file is negligible.

I sometimes condense if-statements and for/while-loops into a single line instead of putting the body on a separate line. I may even chain together several nested control structures on a single line. I was introduced to this practice when reading Dan Bernstein's code and I find that the increased code density makes my intent clearer and the program more legible. The downsides are that bugs involving extraneous semicolons are on rare occasions harder to catch and that debugging each part of the nested control structure requires editing the source to break each part onto its own line. I encounter these downsides so rarely that I find the loss negligible.

I never use typedef. I find that it makes code very hard to understand especially when pointer types are involved, as there is no consensus on whether the typedef should denote a pointer type or the type being pointed to. Loss of precision when converting between numerical types is also hard to grasp when the bit-widths of the types involved are concealed behind typedefs.