Program analysis tools and examples of their use

Tutorial 08 on Dedicated systems

Teacher: Giuseppe Scollo

University of Catania
Department of Mathematics and Computer Science
Graduate Course in Computer Science, 2016-17

Table of Contents

tutorial outline

this tutorial deals with:

- GNU cross-compilers and utilities for low-level program analysis
- examples of use of such tools for various processors and program development systems:
  - for NIOS II processor with GCC 4.7.3 and 5.2.0 cross-compiler and utilities
  - for SimIt-ARM simulator v. 2.1-sfu with GCC 3.2 cross-compiler and utilities
  - for ARM Cortex-A9 processor with GCC 5.2.0 cross-compiler and utilities
  - execution with debugger for NIOS II processor on FPGA
- lab experience: reproduction of the execution of the examples and production of tutorial documentation

GNU cross-compilers and binutils

program development for dedicated systems often takes place on machines that are based on a different processor than the target one, viz. that which is meant for their execution

- a *cross-compiler* is a compiler which produces assembly code and executable programs for a different architecture than that on which it is executed
- just like ordinary compilers, cross-compilers are often equipped with a collection of utilities for low-level program analysis

the cross-compilers and utilities here considered are GNU free software, that typically have names `<target>-gcc` for the compiler, where `<target>` indicates the target architecture, and `<target>-<util>` for the utility `<util>`, such as for example

- `size`: list of sizes of sections and total size of an object module or of an executable
- `objdump`: contents of an object module or of an executable
- `nm`: list of symbols of an object module or of an executable

example for NIOS II processor

the use of the nios2-elf-gcc cross-compiler and of a couple of binary utilities, exposed in sect. 7.4 of (Schaumont 2012) for the example in Listing 7.7, can be reproduced by running the scripts provided in the nios2binutils_example.tgz archive, available in the crosstools folder of the reserved lab area

> the scripts are available in two versions, each relating to a version (13.1 or 16.0) of the nios2eds suite which contains the cross-compiler and related binary utilities, respectively of versions 4.7.3 and 5.2.0 of GCC

for each version, two scripts are available:

- nios2gccToolchain.sh for the production of assembly source, object file and executable
- ext_build/nios2gcc_extToolchain.sh for the analysis of the object and executable files using the size and objdump utilities, and for the generation of the linker memory map

example for SimIt-ARM simulator

the example showcases the use of the simulator SimIt-ARM v. 2.1 both for the analysis of the executable and for the command-line simulation of a program for GCD computation with Euclid's algorithm (Listing 7.11 in (Schaumont, 2012))

> version 2.1 of the simulator is made use of, rather than the current 3.0, because the former supports cycle-accurate simulation, whereas the latter only supports functional simulation

compilation goes through the cross-compiler arm-linux-gcc, for ISA ARMv5, implemented in the StrongARM processor among others

> this cross-compiler with related binary utilities as well as the simulator can be found in repository rijndael.ece.vt.edu/gezel2repo

archive arm_simitarm_example.tgz, available in folder crosstools of the reserved lab area, provides two scripts for this tutorial:

- arm-linux-gccToolchain.sh for the production of assembly source, object file and executable
- sim/SimIt-ARM.sh for cycle-accurate analysis of the execution (simulator sima) and for simulation with the ema debugger, with command-line input in file emadCommands.txt

example for ARM Cortex-A9 processor

the example deals with the execution, on an ARM Cortex-A9 in the Altera DE1-SoC system, of a variation of the program in Listing 7.1, where the main() function body is replaced by

```
printf("maxgcd(a,b)=%d\n", gcd(a,b));
return 0;
```

loading and execution take place by means of the program altera-monitor-program, under control of its included GDB debugger and through the JTAG connection between the program and the DE1-SoC

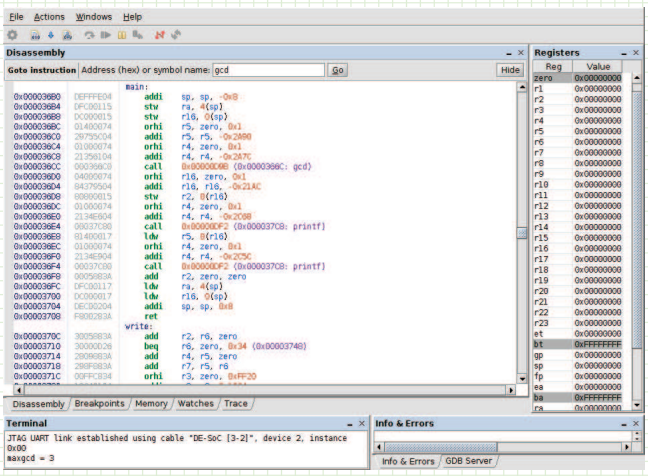
> the system allows the loading of a C or assembly source, whereby compilation is done through the monitor program itself, however, in this example, scripts are provided that are similar to those in the previous examples, in order to allow low-level analysis with binary utilities, and the executable is loaded that is generated by those scripts

the archive arm_cortex-a9_example.tgz, available in the crosstools folder of the reserved lab area contains the following scripts:

- arm_altera_gccToolchain.sh for the production of assembly source, object and executable
- ext_build/arm_altera_gcc_extToolchain.sh for the analysis of the object and executable files

execution with debugger for NIOS II processor on FPGA

the same C source from the previous example may be compiled and executed by a NIOS II softcore on the FPGA of the DE1-SoC system, in this example with no other script, just to explore the use of the debugger included in the monitor program



a snapshot of execution on an Altera DE1-SoC board, controlled by the Altera Monitor Program debugger

lab experience

run the scripts provided with this tutorial, examine the files produced by their execution, find a few not well-known aspects of their contents, search the web for information about them and produce some tutorial documentation in form of a list of questions and answers

each answer should give a reference to its source; if the answer is drawn from a directly accessible web source, insert the specific link; if this points to information that only answers the question, then giving the link as answer is enough

here are three examples of question and answer, one for each of the three types described above:

Q: What is the meaning of the acronyms VMA and LMA in the output of `objdump`?

A: Respectively *virtual memory address* and *load memory address*; the former is the address of a program section during execution, the latter is its address when the program is first loaded in memory; the two addresses may differ when the program is first loaded on a different memory than the one which holds it at runtime, for example it is loaded on a Flash memory and then copied into RAM when it executes. (Schaumont, 2012, pp. 216-217)

Q: What does the ARM instruction `rsb` do?

A: It stands for *reverse subtract*, as it subtracts the first source operand from the second one, that is, the two arguments are in reverse order with respect to `sub`; please note that arithmetic ARM instructions allow the second source operand to be immediate, whereas the first one and the destination must be registers, so this instruction allows subtraction from a constant. (www.heyrick.co.uk/armwiki/RSB)

Q: What is the origin of the name `bss` of the executable program section that contains the global variables?

A: en.wikipedia.org/wiki/.bss#Origin

references

recommended readings:

Schaumont (2012) Ch. 7, Sect. 7.4, 7.6.2

readings for further consultation:

Schaumont (2012) Ch. 7, Sect. 7.6.1, 7.6.3

useful materials for the proposed lab experience:

GCC, the GNU Compiler Collection

GNU Binary Utilities

Introduction to the the Altera Nios II Soft Processor, Altera University Program, May 2016

Introduction to the ARM® Processor Using Altera Toolchain, Altera University Program, May 2016

SimIt-ARM Users' Guide

Altera Monitor Program Tutorial for Nios II, Altera University Program, May 2016

Altera Monitor Program Tutorial for ARM, Altera University Program, May 2016