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, 2018-19

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 ARM emulator VisUAL v. 1.27 with GCC 3.2 cross-compiler, GNU Binutils 2.13, and ad-hoc script
  - execution with debugger on ARM Cortex-A9 processor (DE1-SoC HPS)
  - execution with debugger on Nios II processor (DE1-SoC FPGA)
  - for ARM Cortex-A9 processor with GCC 4.8.1 cross-compiler and GNU Binutils 2.23
  - for Nios II processor with GCC 5.3.0 cross-compiler and GNU Binutils 2.25
- lab experience: reproduction of the execution of the examples and analysis of the problem that was mentioned near the end of the previous lecture

  C sources and scripts for the execution of the lab experience are available in the `classroom.tgz` archive, which can be found in folder `crosstools/e08` of the reserved lab area

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

examples for VisUAL ARM emulator

a first example showcases the use of the VisUAL v. 1.27 ARM emulator, for the simulation of the execution of a program for GCD computation with Euclid's algorithm (Listing 7.11 in Schaumont, with modified `main` because of the different simulation target)

a second example, illustrated by the next figure, presents a software implementation of the Collatz delay calculation, similar in functionality to the hardware implementations produced in the previous lab experiences

here the emulation gets a lexical problem with the (pre-UAL) code produced by the cross-compiler

the program seen at the end of the previous lecture (Listing 7.4 in Schaumont), with an initialization of the array contents, forms the third example

compilation without optimization produces a more serious kind of troubles in this case

the emulator accepts an ARM source assembly program coded in a subset of the instruction set defined by the UAL (*Unified Assembly Language*) syntax

in these examples, the assembly program is obtained from the C source by means of the `arm-linux-gcc` cross-compiler, for ISA ARMv4, implemented in the StrongARM processor

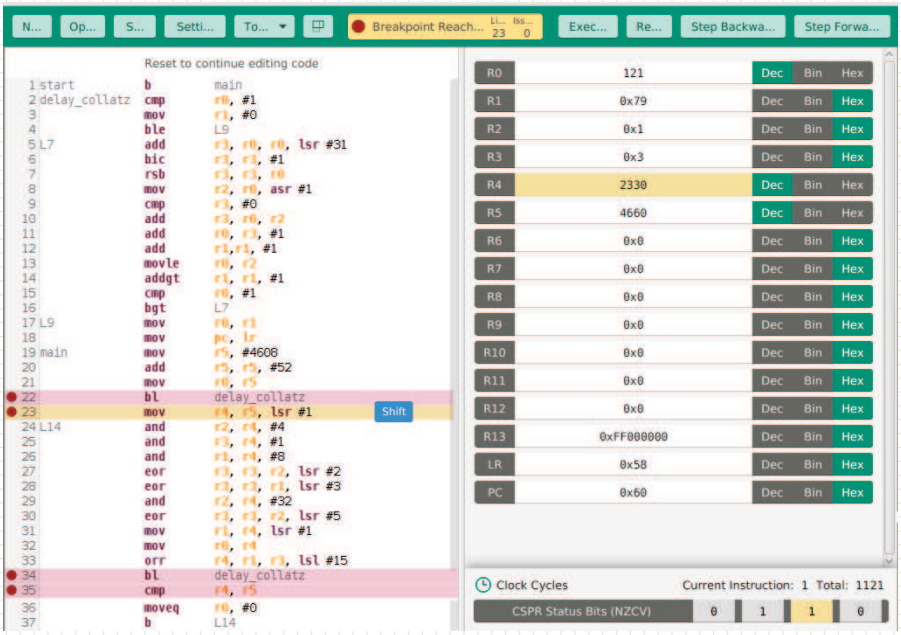
the Debian package installation provides both this cross-compiler and its binary utilities

however, the assembly program so obtained does not meet the restrictions posed by the emulator, e.g. it contains assembler directives and other syntactic sugar that are not accepted by VisUAL

this obstacle is mostly overcome by an ad-hoc script, that edits a copy of the previous assembly source to produce a version of it that is *almost* conforming to VisUAL restrictions

folder `arm_visual` in the provided archive contains the C sources of the three examples, together with the `c2visual` script for the cross-compilation and aforementioned editing of the produced assembly code (script `c02visual` only differs in the compilation without optimization, for the analysis of the problem mentioned in the previous lecture

VisUAL emulation of an example

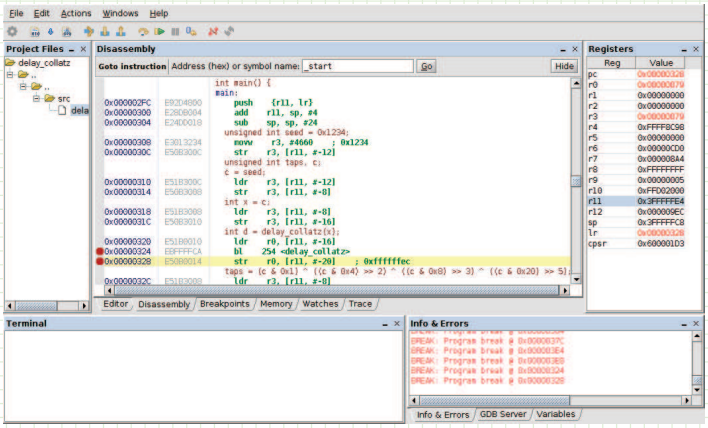


a snapshot of VisUAL emulation of the `delay_collatz.s` ARM assembly program

execution with debugger for ARM Cortex-A9 processor

the figure shows an execution snapshot of program delay_collatz.c on the DE1-SoC ARM processor
cross-compilation, loading and execution take place by means of the altera-monitor-program, under control of its GNU debugger GDB

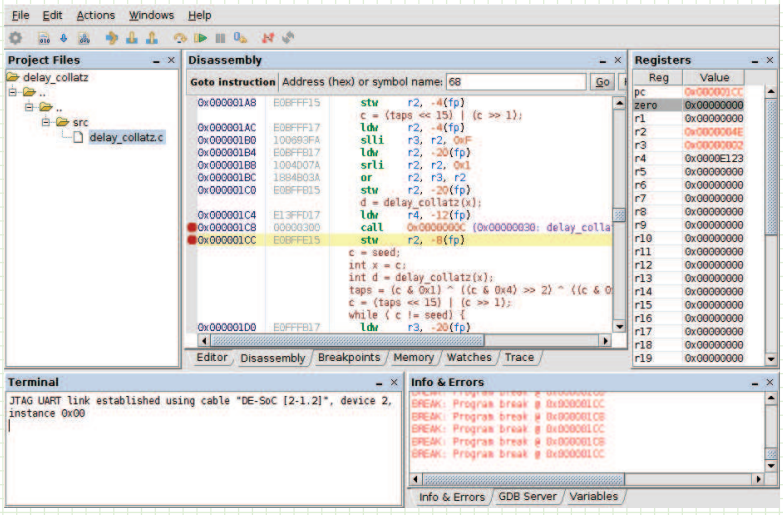
- the system allows the loading of a C or assembly source, whereby cross-compilation is done through the monitor program itself



execution snapshot of an ARM program under control of the GDB debugger within the Monitor Program

execution with debugger for Nios II processor

the same C source from the previous example may be cross-compiled and executed by a Nios II softcore on the FPGA of the DE1-SoC system



execution snapshot of a Nios II program under control of the GDB debugger within the Monitor Program

execution of the delay_collatz example on the ARM processor by means of the monitor program produces two files in the project directory that are relevant to low-level analysis:

- delay_collatz.axf: the ELF executable
- delay_collatz.axf.objdump: its disassembly produced by the objdump utility

in order to carry out low-level analysis even further, it helps one to know where the binary utilities are located, for the gcc cross-compiler used by the monitor program

> in the installation directory of the Quartus Prime software, the cross-compiler and the binutils are located in directory
> University_Program/Monitor_Program/arm_tools/baremetal/bin

N.B. actually, the monitor program makes use of specialized versions of some of these tools, that are placed elsewhere; however, for general use, independent of any execution in the contest of the monitor program, the aforementioned tool collection is the proper one

processing the C sources by the monitor program for execution on the Nios II processor produces the .elf executable in the project directory, but not its disassembly

> in the installation directory of the Quartus Prime software, the cross-compiler and the binutils are found in directory
> nios2eds/bin/gnu/H-x86_64-pc-linux-gnu/bin

hre is a simple example of use of the size utility: comparing the sizes of the executables produced by the three cross-compilers of interest here, for the C sources of two of the examples proposed and with the same compilation options; in particular, optimization level O2

> the scripts to this purpose are available in the sizes folder of the provided archive

the following table presents the outcome of this exercise

C source	cross-compiler	text	data	bss	total
gcd	arm-linux	820	260	4	1084
	arm-altera-eabi	492	16	28	536
	nios2-elf	1056	1068	0	2124
delay_collatz	arm-linux	864	260	4	1128
	arm-altera-eabi	628	16	28	672
	nios2-elf	1196	1068	0	2264

lab experience

run the examples provided with this tutorial and produce documentation of any difficulties met and of the solutions found to overcome them

in particular, the VisUAL emulation of example `accumulate` requires further editing of the code produced by the compilation + editing script, both with and without optimization, in order to solve the following problems:

- with optimization (script `c2visual`): it seems that VisUAL only admits *one* occurrence of the `DCD` directive; furthermore, it makes use of instruction `ADR` (unknown in ARMv4) to load an address to a register, whereas the code produced by the script makes use of instruction `LDR` with a source location that holds the subject address
- without optimization (script `c02visual`): besides the aforementioned problems, the multiple transfer instructions produced by the script violate some of the restrictions posed by VisUAL, according to similar prescriptions introduced in ARM ISA versions higher than v4

with respect to the last mentioned problem, it is proposed to:

- solve it, to the purpose of emulation, by using instruction `LDMFD sp! ...` instead of `LDMEA fp ...` to restore saved registers and return, with consistent modifications to handling of the activation frame and of registers `fp` and `sp`
- explain why the handling of the stack and of the activation frame made by the ARMv4 code produced by the cross-assembler is correct, unlike figure 7.9, which is aimed at illustrating it

references

recommended readings:

Schaumont, Ch. 7, Sect. 7.4

readings for further consultation:

Schaumont, Ch. 7, Sect. 7.5, 7.6

The ARM Instruction Set, tutorial, ARM University Program, V1.0

VisUAL - A highly visual ARM emulator

tutorials from source Intel® FPGA University Program (November 2016):

Introduction to the ARM® Processor Using Intel FPGA Toolchain - For Quartus Prime 16.1

Introduction to the the Intel Nios II Soft Processor - For Quartus Prime 16.1

Intel FPGA Monitor Program Tutorial for ARM - For Quartus Prime 16.1

Intel FPGA Monitor Program Tutorial for Nios II - For Quartus Prime 16.1

useful materials for the proposed lab experience:

ARM Architecture Reference Manual, ARMv7-A and ARMv7-R Edition, ARM DDI 0406C.c (2014)

GCC, the GNU Compiler Collection

GNU Binary Utilities