# 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, 2017-18

## 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 simulator 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 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 VisUAL ARM emulator (1)

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)

- 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 this example, 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

- this cross-compiler and its binary utilities can be retrieved from rijndael.ece.vt.edu/gezel2repo/pool/main/a/arm-linux-gcc/

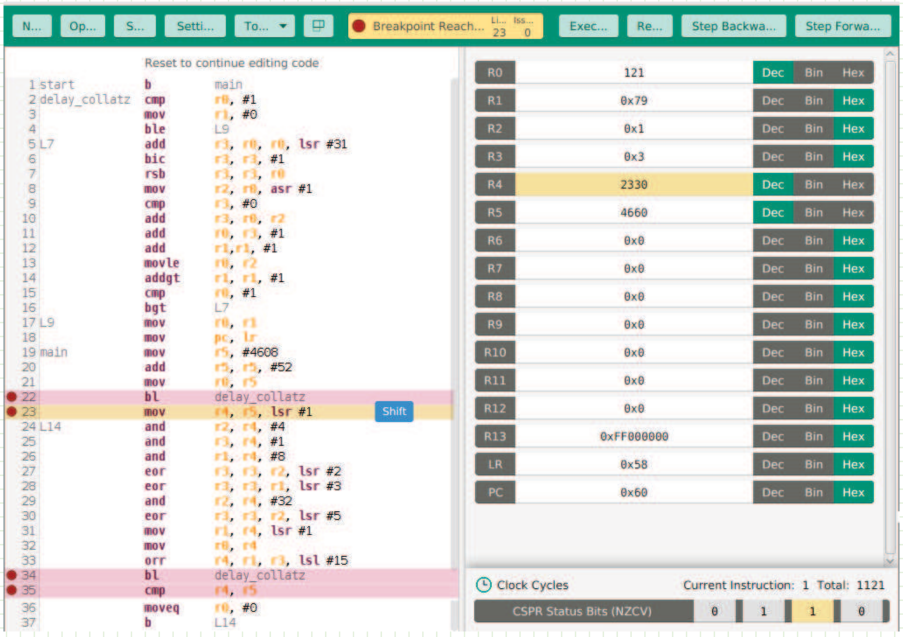
however, the assembly program so obtained does not meet the restrictions posed by the emulator, e.g. it contains assembly directives and other syntactic sugar that are not accepted by VisUAL

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

archive arm_visual_examples.tgz, available in folder crosstools of the reserved lab area, provides the C sources and the required scripts for the cross-compilation and VisUAL emulation of the GCD example and of a second example, that presents a software implementation of the Collatz delay calculation, similar in functionality to the hardware implementations considered in the previous lab experiences

- to run the examples, after unpacking the archive with tar xzpvf, in folder arm_visual_examples run respectively ./c2visual gcd, and ./c2visual delay_collatz, then launch VisUAL and open the respectively generated files gcd/gcd.s, and delay_collatz/delay_collatz.s (*warning:* with lowercase extension .s !)

## example for VisUAL ARM emulator (2)



```
start           b       main
delay_collatz   cmp     r0, #1
                mov     r1, #0
                ble     L9
L7              add     r3, r0, r0, lsr #31
                bic     r3, r3, #1
                rsb     r3, r3, r0
                mov     r2, r0, asr #1
                cmp     r3, #0
                add     r3, r0, r2
                add     r0, r3, #1
                add     r1,r1, #1
                movle   r0, r2
                addgt   r1, r1, #1
                cmp     r0, #1
                bgt     L7
L9              mov     r0, r1
                mov     pc, lr
main            mov     r5, #4608
                add     r5, r5, #52
                mov     r0, r5
                bl      delay_collatz
                mov     r4, r5, lsr #1
L14             and     r2, r4, #4
                and     r3, r4, #1
                and     r1, r4, #8
                eor     r3, r3, r2, lsr #2
                eor     r3, r3, r1, lsr #3
                and     r2, r4, #32
                eor     r3, r3, r2, lsr #5
                mov     r1, r4, lsr #1
                mov     r0, r4
                orr     r4, r1, r3, lsl #15
                bl      delay_collatz
                cmp     r4, r5
                moveq   r0, #0
                b       L14
```

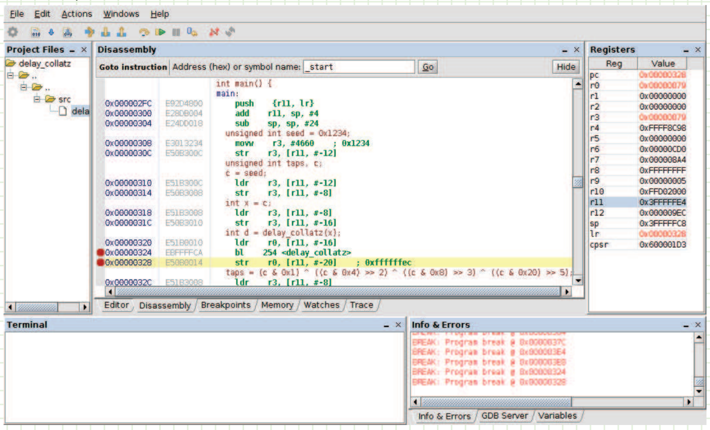
```
R0   121
R1   0x79
R2   0x1
R3   0x3
R4   2330
R5   4660
R6   0x0
R7   0x0
R8   0x0
R9   0x0
R10  0x0
R11  0x0
R12  0x0
R13  0xFF000000
LR   0x58
PC   0x60

Current Instruction: 1  Total: 1121
CSPR Status Bits (NZCV)  0  1  1  0
```

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



```
                                int main() {
                                main:
0x000002FC   E92D4800              push    {r11, lr}
0x00000300   E28DB004              add     r11, sp, #4
0x00000304   E24DD018              sub     sp, sp, #24
                                 unsigned int seed = 0x1234;
0x00000308   E3013234              movw    r3, #4660     ; 0x1234
0x0000030C   E50B300C              str     r3, [r11, #-12]
                                 unsigned int taps, c;
                                 c = seed;
0x00000310   E51B300C              ldr     r3, [r11, #-12]
0x00000314   E50B3008              str     r3, [r11, #-8]
                                 int x = c;
0x00000318   E51B3008              ldr     r3, [r11, #-8]
0x0000031C   E50B3010              str     r3, [r11, #-16]
                                 int d = delay_collatz(x);
0x00000320   E51B0010              ldr     r0, [r11, #-16]
0x00000324   EBFFFFCA              bl      254 <delay_collatz>
0x00000328   E50B0014              str     r0, [r11, #-20]     ; 0xffffffec
                                 taps = (c & 0x1) ^ ((c & 0x4) >> 2) ^ ((c & 0x8) >> 3) ^ ((c & 0x20) >> 5);
0x0000032C   E51B3008              ldr     r3, [r11, #-8]
```

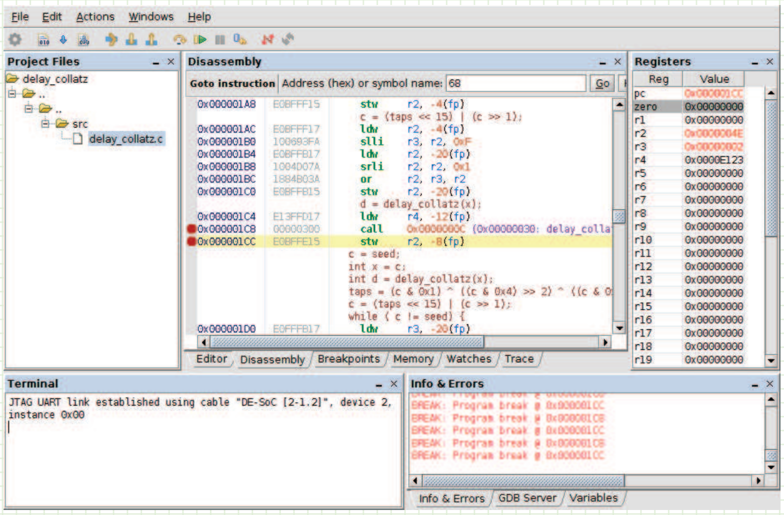
| Reg | Value |
|---|---|
| pc | 0x00000328 |
| r0 | 0x00000079 |
| r1 | 0x00000000 |
| r2 | 0x00000000 |
| r3 | 0x00000079 |
| r4 | 0xFFFF8C98 |
| r5 | 0x00000000 |
| r6 | 0x00000CD0 |
| r7 | 0x000008A4 |
| r8 | 0xFFFFFFFF |
| r9 | 0x00000005 |
| r10 | 0xFFD02000 |
| r11 | 0x3FFFFFE4 |
| r12 | 0x000009EC |
| sp | 0x3FFFFFC8 |
| lr | 0x00000328 |
| cpsr | 0x600001D3 |

```
BREAK: Program break @ 0x0000037C
BREAK: Program break @ 0x000003E4
BREAK: Program break @ 0x000003E8
BREAK: Program break @ 0x00000324
BREAK: Program break @ 0x00000328
```

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



```
0x000001A8   E0BFFF15          stw     r2, -4(fp)
                               c = (taps << 15) | (c >> 1);
0x000001AC   E0BFFF17          ldw     r2, -4(fp)
0x000001B0   100693FA          slli    r3, r2, 0xF
0x000001B4   E0BFFB17          ldw     r2, -20(fp)
0x000001B8   1004D07A          srli    r2, r2, 0x1
0x000001BC   1884B03A          or      r2, r3, r2
0x000001C0   E0BFFB15          stw     r2, -20(fp)
                               d = delay_collatz(x);
0x000001C4   E13FFD17          ldw     r4, -12(fp)
0x000001C8   00000300          call    0x0000000C (0x00000030: delay_colla
0x000001CC   E0BFFE15          stw     r2, -8(fp)
                             c = seed;
                             int x = c;
                             int d = delay_collatz(x);
                             taps = (c & 0x1) ^ ((c & 0x4) >> 2) ^ ((c & 0
                             c = (taps << 15) | (c >> 1);
                             while ( c != seed) {
0x000001D0   E0FFFB17          ldw     r3, -20(fp)
```

| Reg | Value |
|---|---|
| pc | 0x000001CC |
| zero | 0x00000000 |
| r1 | 0x00000000 |
| r2 | 0x0000004E |
| r3 | 0x00000002 |
| r4 | 0x0000E123 |
| r5 | 0x00000000 |
| r6 | 0x00000000 |
| r7 | 0x00000000 |
| r8 | 0x00000000 |
| r9 | 0x00000000 |
| r10 | 0x00000000 |
| r11 | 0x00000000 |
| r12 | 0x00000000 |
| r13 | 0x00000000 |
| r14 | 0x00000000 |
| r15 | 0x00000000 |
| r16 | 0x00000000 |
| r17 | 0x00000000 |
| r18 | 0x00000000 |
| r19 | 0x00000000 |

```
JTAG UART link established using cable "DE-SoC [2-1.2]", device 2,
instance 0x00
```

```
BREAK: Program break @ 0x000001CC
BREAK: Program break @ 0x000001C8
BREAK: Program break @ 0x000001CC
BREAK: Program break @ 0x000001C8
BREAK: Program break @ 0x000001CC
```

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

## low-level analysis of ARM executables

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

## low-level analysis of Nios II executables

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 both of the examples proposed and with the same compilation options; in particular, optimization level O2

the two C sources and the scripts to this purpose are available in the cross-compiled_sizes.tgz archive, within the crosstools directory in the reserved lab area

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, 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, 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:

> *GCC, the GNU Compiler Collection*
>
> *GNU Binary Utilities*