University of L'Aquila
Embedded Systems
2021/2022

Teacher:
Dr. Luigi Pomante

Contributor:
Pietro Alfonsi
Edgardo Vittoria
Giannantonio Frittella
Andrea Rofini

HOMELAB

μC 8051 ISS/HDL

Contents

1. *Introduction: Micro Controller Intel 8051* ____ 4
   - 1.1. **Specifications** ____ 4
   - 1.2. **Instruction Set Architecture (I.S.A.)** ____ 5
   - 1.3. **I8051 Tools and Miscellanea** ____ 6
     - 1.3.1. I8051 Toolchains ____ 6
     - 1.3.2. Dalton Project Facilities ____ 6
     - 1.3.3. Hex Format ____ 8
2. *Setting Up the Environment* ____ 10
   - 2.1. **Software Requirements for Homelab execution** ____ 10
   - 2.2. **Preparing material** ____ 11
   - 2.3. **Compilers Tools** ____ 12
     - 2.3.1. SDCC ____ 12
     - 2.3.2. Keil ____ 14
   - 2.4. **Tools for Simulation** ____ 15
     - 2.4.1. Tools for Sw Simulation: Building ISASim ____ 15
     - 2.4.2. Tools for Hw Simulation: Building Hex2Rom ____ 18
3. *Compile for Intel 8051* ____ 19
   - 3.1. **Program example** ____ 19
     - 3.1.1. SDCC and Keil includes ____ 20
     - 3.1.2. Expected results ____ 20
   - 3.2. **SDCC compiler in action** ____ 20
     - 3.2.1. Build with plain SDCC ____ 20
     - 3.2.2. Build with SDCC flags ____ 22
   - 3.3. **Keil compiler in action** ____ 24
     - Keil output issue ____ 29
4. *Software simulation* ____ 30
   - 4.1. **Running ISASim** ____ 31
     - 4.1.1. Running ISASim on Linux ____ 31
     - 4.1.2. Running ISASim on Windows ____ 31
     - 4.1.3. ISASim debug information report ____ 32
   - 4.2. **Software Simulation results comparison** ____ 32
5. *Hardware simulation* ____ 36
   - 5.1. **Running Hex2Rom** ____ 36
     - 5.1.1. Running Hex2Rom on Linux ____ 36
   - 5.2. **Creating Vivado project for I8051 model** ____ 36
     - 5.2.1. Perform Vivado Hardware Simulation ____ 39
   - 5.3. **ROM memory generation (by using Hex2Rom)** ____ 41
   - 5.4. **Hardware simulation results comparison** ____ 41
6. *Conclusions* ____ 44

C/C++code simulation for μC Intel 8051

Homelab purpose is to build a development environment for Intel 8051 microcontroller and to use simulation techniques in order to verify the functionality and efficiency of developed software.

This Homelab proposes first to compile a C program, then to perform both software and hardware simulations. Software simulation will use an ISS, while the hardware simulation will use a VHDL model.

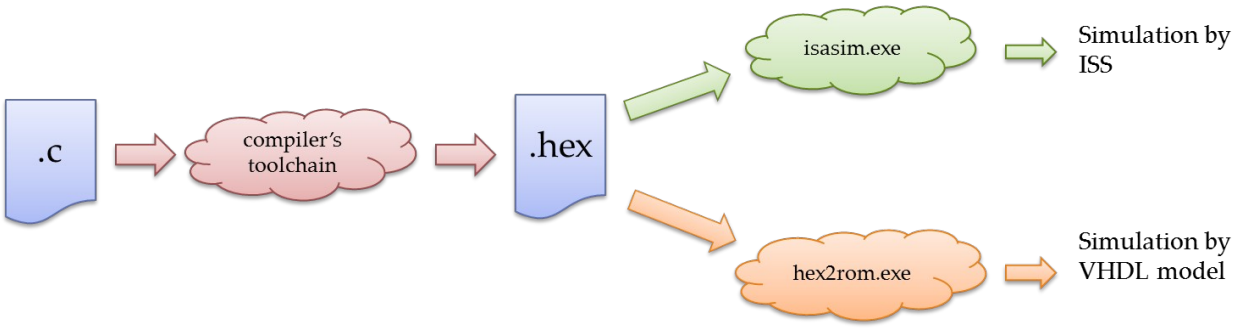


Figure 1 – Homelab workflow

In this tutorial we will first learn how to build a .c file for 8051. Various tools will be proposed and compared, executable file will be produced in hex format.

Once executable .hex is obtained, you can simulate its execution through the use of an ISS.
.hex executable also allows to produce a ROM image in vhdl format; its inclusion in VHDL model description you will be allowed to perform a complete hardware simulation.

Homelab purpose, rather than learn i8051 simulation process flow, is to apply the process itself to whatever function or program.

While this document is a guide through the i8051 simulation flow with respect to a toy example, each student will be asked to apply the same flow to one different function of its choice.

The function shall be of medium/low difficulty, similar in complexity and computational weight to the example proposed here in the document and/or the one proposed during the i8051 homelab class.

In the following, while describing homelab development, you will find five different sections named *Group/Student shall*. In each of those sections, are described the steps a student or a group of students shall to take in order to learn homelab notions.
It is worth to notice that last *Group/Student shall* section contains the real homelab assignment.

At last, please consider that some information you will find in this document, such as URLs, are valid at the time of writing but may change in time.

1. Introduction: Micro Controller Intel 8051

From Wikipedia (*http://it.wikipedia.org/wiki*), for *Intel 8051* entry:

> *The Intel MCS-51 (commonly termed 8051) is a Harvard architecture, complex instruction set computing (CISC) instruction set, single chip microcontroller (µC) series developed by Intel in 1980 for use in embedded systems.[1] Intel's original versions were popular in the 1980s and early 1990s and enhanced binary compatible derivatives remain popular today.*
>
> *The family was continued in 1996 with the enhanced 8-bit MCS-151 and the 8/16/32-bit MCS-251 family of binary compatible microcontrollers.[2] While Intel no longer manufactures the MCS-51, MCS-151 and MCS-251 family, enhanced binary compatible derivatives made by numerous vendors remain popular today. Some derivatives integrate a digital signal processor (DSP). Beyond these physical devices, several companies also offer MCS-51 derivatives as IP cores for use in field-programmable gate array (FPGA) or application-specific integrated circuit (ASIC) designs.*

1.1. Specifications

An 8-bit CPU is the core of the Intel 8051 microcontroller. Harvard Architecture is the architectural model adopted by the microcontroller; therefore, it parts data against instruction by the use of two memories and two buses; indeed 8051 presents a non-volatile PROM memory, which contains program instruction, and a RAM memory, which contains data. Furthermore, it presents an 8-bit Data Bus and a 16-bit Address Bus. I8051 registers are 8-bit registers. ALU works with 8-bit words and makes use of an accumulator register. Intel 8051 communicates with four I/O 8-bit ports.

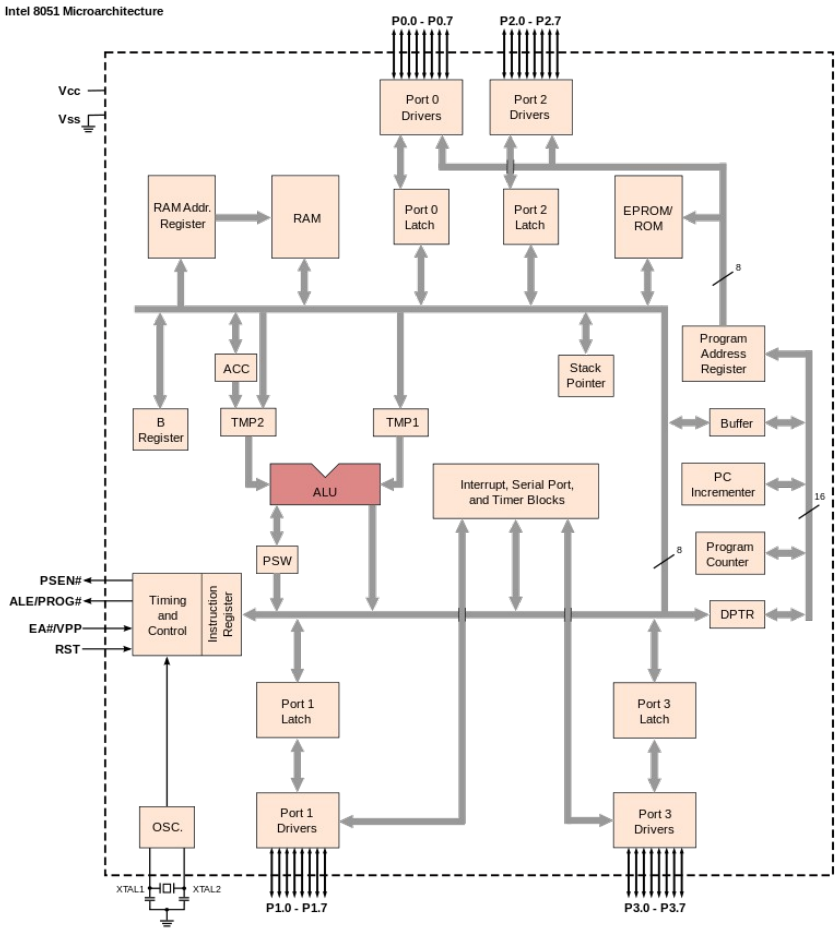


Figure 2 - Intel 8051 microarchitecture

1.2. Instruction Set Architecture (I.S.A.)

Instruction set groups its instructions into five distinct groups: arithmetic instructions, control instructions; logic instructions; single bit operations and finally data transfer instructions.
Note that the generic operation OP can be declined with several kinds of parameters: OP (*A*, *Rn*), OP (*A*, *direct*), OP (*A*, *@Ri*), OP (*A*, *#data*) where *Rn* is the name of a register, *direct* indicates a memory position, *Ri* is a pointer to a position, *#data* is a given data.
For an exhaustive list of instructions, please refer to https://it.wikipedia.org/wiki/Intel_8051. Below some of them.

1.2.1. Arithmetic instruction

Mnemonics	Description	Bytes	Cycles
ADD A,Rn	Sum A to content of Rn register	1	12
ADDC A,Rn	Sum A to content of Rn register and CY	1	12
SUBB A,Rn	Subtract to A the content of Rn register and CY	1	12
INC A	Unitary increment of A	1	12
DEC A	Unitary decrement of A	1	12
INC DPTR	Unitary increment of DPTR register	1	24
MUL AB	Multiply A content for B	1	48
DIV AB	Divide A per B; quotient in A e rest in B	1	48

1.2.2. Control instruction

Mnemonics	Description	Bytes	Cycles
ACALL addr11	Executes routine in a 2K segment	2	24
LCALL addr16	Executes routine	3	24
RET	Stop routine execution	1	24
AJMP addr11	Jump to specified address in a 2K segmento	2	24
LJMP addr16	Jump to specified address	3	24
SJMP rel	Jump to [rel] successive position of Program Counter	2	24
JMP @A+DPTR	Jump to PC position pointed by A plus DPTR content	1	24
JZ rel	Jump to [rel] position if A content is zero	2	24
NOP	No operation	1	12

1.2.3. Logic Instruction

Mnemonics	Description	Bytes	Cycles
ANL A,Rn	Logic AND between A and Rn register content	1	12
ORL A,Rn	Logic OR between A Rn register content	1	12
XRL A,Rn	Logic EX-OR between A Rn register content	1	12
CLR A	Put all A bits to zero	1	12
CPL A	Invert all A bits (1-Complement)	1	12
RL A	1 step left shift A bits	1	12
SWAP A	Swap two accumulator nibbles	1	12

1.2.4. Single bit operation instructions

Mnemonics	Description	Bytes	Cycles
CLR C	Set CY flag to 0	1	12
CLR bit	Set [bit] address bit to 0	2	12
SETB C	Set CY flag to 1	1	12
SETB bit	Set [bit] address bit to 1	2	12
CPL C	Invert CY flag	1	12
ANL C,bit	Logic AND between CY and [bit] address bit	2	24
ORL C,bit	Logic OR between CY and [bit] address bit	2	24
MOV C,bit	Copy carry bit in bit address	2	12

1.2.5. Data transfer instructions

Mnemonics	Description	Bytes	Cycles
MOV A,Rn	Copy A content in Rn register	1	12
PUSH direct	Copy in position pointed from SP the content of direct position and increment Stack	2	24
POP direct	Copy in position pointed from SP the content of direct position and decrement Stack	2	24
XCH A,Rn	Swap A content with Rn	1	12

1.3. I8051 Tools and Miscellanea

1.3.1. I8051 Toolchains

8051 Compilers' market is extensive; there are open source compilers and compilers for payment. Both have their strengths and weaknesses.
One of the most advertised products is MikroC PRO for 8051, it can be found at http://www.mikroe.com/mikroc/8051/. This software is available as a free trial or as a full version at the cost of $249.
An open source compiler is SDCC (Small Device C Compiler), whose website is http://sdcc.sourceforge.net/, is a completely free software that well suits our needs.
Another compiler for payment is Keil, it is used within the µVision IDE which is specifically designed for the compiler. Refer to the website http://www.keil.com/C51/. This software is not open source, but user is allowed to use some of its functionalities for free, without a current license it can still compile as long as code not exceeds 2Kbytes of object code.

SDCC compiler

From SDCC manual:

> *SDCC (Small Device C Compiler) is free open source, retargettable, optimizing standard (ISO C90, ISO C99, ISO C11) C compiler suite by Sandeep Dutta designed for 8 bit Microprocessors. [...] The entire source code for the compiler is distributed under GPL.*
> *SDCC uses a modified version of ASXXXX & ASLINK, free open source retargetable assembler & linker. SDCC has extensive language extensions suitable for utilizing various microcontrollers and underlying hardware.*

And more:

> *SDCC is not just a compiler, but a collection of tools by various developers. These include linkers, assemblers, simulators and other components.*

Keil compiler

From Keil compiler official website http://www.keil.com/, the product description:

> *Keil C51 is the industry-standard toolchain for all 8051-compatible devices, it supports classic 8051, Dallas 390, NXP MX, extended 8051 variants, and C251 devices. The µVision IDE/Debugger integrates complete device simulation, interfaces to many target debug adapters, and provides various monitor debug solutions*.

1.3.2. Dalton Project Facilities

There are several ways to simulate the execution of a program compiled on a hardware platform. Among them there are two categories: the software approach by ISS (Instruction Set Simulator) and the hardware approach using VHDL description (VHSIC Hardware Description Language).

The University of California developed a project centred on 8051 microprocessor, project provides a number of tools and examples useful for simulating C code on Intel 8051 microprocessor. The project name is *Dalton* developed by the *Dept. of computer Science of the University of California*.
The online project reference is http://www.ann.ece.ufl.edu/i8051/. It offers an ISS written in C ++ language for the Intel 8051 and a VHDL model of the processor.

To make best use of the VHDL model, it also provides a program to transform the Intel 8051 compiled code in VHDL memory ROM module containing instructions of the compiled code. Thus, it allows hardware simulation of C code; it even allows to synthesize the microcontroller 8051 with the compiled program loaded in the ROM.
From project website:

> *We developed a VHDL synthesizable model of the 8051 and a C++ based 8051 instruction-set simulator [...], on which we've based some research directions. One of those directions is a tuning environment [...], to assist a designer who wants to modify the 8051 architecture to be more power efficient for a particular program.*

Software simulation: Instruction Set Simulator (ISS)

From wikipedia (*http://en.wikipedia.org/wiki*), at Instruction Set Simulator page:

> *An instruction set simulator (ISS) is a simulation model, usually coded in a high-level programming language, which mimics the behavior of a mainframe or microprocessor by "reading" instructions and maintaining internal variables which represent the processor's registers.*

Dalton Project's Instruction Set Simulator (ISS) source code consists of the following files:

- Main.cc,
- i8051.cc,
- i8051.h.

From *Dalton Project* website:

> *The high level simulator for the Intel 8051, written in C++, allows a user to simulate simple programs written for the 8051. The simulator provides statistics on instructions executed, instructions executed per second, execution cycles required by the 8051, and average instructions per second for an 8051 executing the same program.*

ISS source code files are downloadable at Dalton Project website (http://www.ann.ece.ufl.edu/i8051/i8051sim/) or in *ISS* folder provided within material of this homelab.

In order to execute the program you have to compile these files and then you will able to run the executable file from Command Line Interface (CLI).
Refer to section 2.4.1. for ISASim building procedure.

Hardware Simulation (VHDL)

From wikipedia (*http://en.wikipedia.org/wiki*), at VHDL item:

> *VHDL (VHSIC Hardware Description Language) is a hardware **description** language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits. VHDL can also be used as a general purpose parallel programming language.*

VHDL (Very high speed integrated circuits Hardware Description Language) Intel 8051 microprocessor description provided by *Dalton Project* is composed by many files, each describes a specific hardware component.

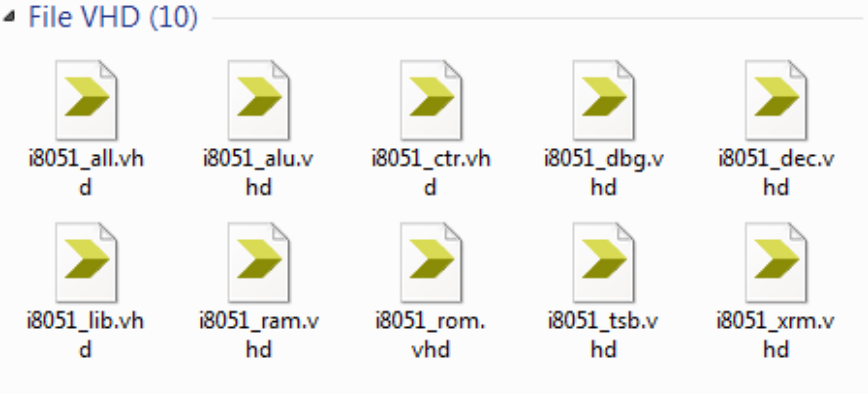


Figure 3 – VHDL model files

From project website:

> *The Intel 8051 is an 8-bit micro-controller. This micro-controller is capable of addressing 64K of program and 64K of data memory. The implementation [...] is written in Synthesizable VHDL (at least by Synopsys and Xilinx,) and models the actual Intel implementation rather closely, e.g., it is 100% instruction compatible.*

VHDL ROM memory component (modelled by i8051_rom.vhd file) is filled and configured with the program instructions; when started up the microprocessor will execute those instructions. By configuring this component accordingly to program willing to execute, we need a process/program to fill the rom with desired instructions.
To obtain ROM .vhd file filled with instructions accordingly to .hex file, Dalton project provides i8051_mkr.c program. i8051_mkr.c needs to be compiled in order to get an executable program which is runnable from Command Line Interface and allow to create i8051_rom.vhd file from .hex executable.

1.3.3. Hex Format

Hex is an Intel format that formats low-level hardware instructions in hexadecimal format. From Wikipedia (https://en.wikipedia.org/wiki/Intel_HEX) at *Intel Hex* item:

> *Intel HEX is a file format that conveys binary information in ASCII text form. It is commonly used for programming microcontrollers, EPROMs, and other types of programmable logic devices. In a typical application, a compiler or assembler converts a program's source code (such as in C or assembly language) to machine code and outputs it into a HEX file. The HEX file is then imported by a programmer to "burn" the machine code into a ROM, or is transferred to the target system for loading and execution.*

Despite instructions are hexdecimally coded, characters sequences are hardly comprehensible for a human being. Nonetheless, instructions are structured such that they are easier to be understand.
In Figure 4, the pattern of hex format instructions.

Pos 1	Pos 2-3	Pos 4-7	Pos 8-9	Pos 10-?	Pos10 + Value(Pos2-3)
:	10	0003	00	7F2F7E0BEF6E6015EFD39E4008C3EF9E	EC
:	10	0013	00	FFF58080EFC3EE9FFEF59080E78FA080	11
:	02	0023	00	FE22	BB
:	03	0000	00	020025	D6
:	0C	0025	00	787FE4F6D8FD758107020003	37
:	00	0000	01		FF

Positions 1 : record start
Positions 2-3: record length
Positions 4-7: data address. May acquire a value between 0000 – FFFF.
Positions 8-9: data type
Positions 10-?: data field
Position10 + Value(Pos2-3): checksum

Figure 4 – Intel Hex format

2. Setting Up the Environment

Group/Student Shall #1: Prepare its own environment

- Align student environment to Software Requirements; look at section 2.1..
- Create and fill directory tree as described in section 2.2.2.
- Build ISASim tool as described in section 2.4.1.
  - Follow steps into *Building ISASim: fixing code* section
  - Build ISASim without debug option and with debug option
- Build Hex2Rom tool as described in section 2.4.2.

2.1. Software Requirements for Homelab execution

Homelab, proposed in this document, uses third party software; it has been developed for Linux platform. Below the list of required software; before perform homelab assure tools/software availability on your platform. Other software will be introduced during Homelab development.

g++ installation

g++ is the linux compiler for c++ source code files.
In order to build a c++ program, g++ is required to be installed.

Check whether g++ is already installed.
Open terminal and type:

```
> g++ -v
```

If it is not installed you will receive the following:

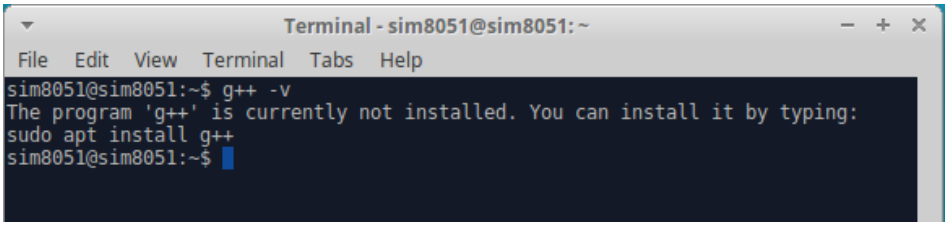


Figure 5 – g++ not installed

Then to install gcc, type:

```
> sudo apt install g++
```

Insert password if/when requested.
Then prompt shows something like:

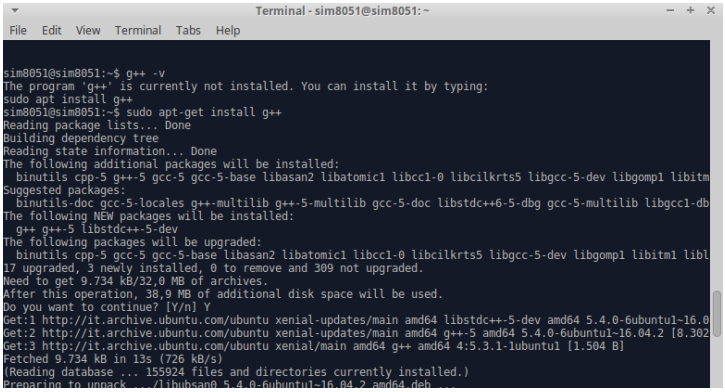


Figure 6 – g++ Installation

Then check g++ installation by typing again:

```
> g++ -v
```

Now you will see:

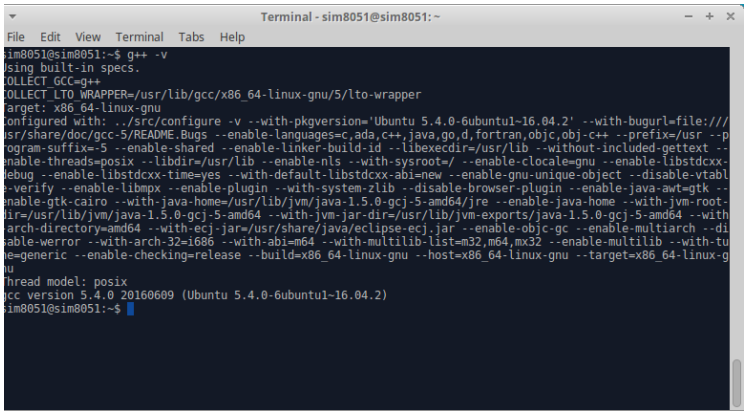


Figure 7 – g++ Installation check

gcc update

Check gcc version by typing

```
> gcc -v
```

If gcc version is lower than online update (4.9.4 at time of writing), perform following commands in order to update the software. Insert password when requested[1].

```
> sudo add-apt-repository ppa:ubuntu-toolchain-r/test
> sudo apt update
> sudo apt upgrade
> sudo apt dist-upgrade
```

Vivado Xilinx

Hardware simulation will be performed by usage of Xilinx software suite. Official description, from Xilinx site, is:

> *The Vivado® Design Suite offers a new approach for ultra high productivity with next generation C/C++ and IP-based design with the new HLx editions including HL System Edition, HL Design Edition and HL WebPACK™ Edition.*
> *The new HLx editions supply design teams with the tools and methodology needed to leverage C-based design and optimized reuse, IP sub-system reuse, integration automation and accelerated design closure. When coupled with the UltraFast™ High-Level Productivity Design Methodology*

Guide, this unique combination is proven to accelerate productivity by enabling designers to work at a high level of abstraction while facilitating design reuse.

The suite consists of, among others, a development environment for hardware description in VHDL and VERILOG; which allows to simulate and analyse described hardware behavior.
The software version is freely downloadable from URL:
http://www.xilinx.com/products/design-tools/vivado/vivado-webpack.html.

2.2. Preparing material

In order to proficiently perform the actions proposed in this document, you will need to fulfill following conditions and prepare files needed in the homelab actions.

Create *~/workspaces/i8051/* directory:

```
> cd ~/
> mkdir workspaces
> cd workspaces
> mkdir i8051
```

Access on your desktop and create the material folder which will contain all the files we will use for this homelab.

```
> cd ~/Desktop/
> mkdir material
```

Then create *ISS*, *VHDL* and *Hex2Rom* folders:

```
> cd ~/Desktop/
> mkdir ISS
> mkdir VHDL
> mkdir Hex2Rom
```

In *~/Desktop/material/ISS* place *Main.cc*, *i8051.cc* and *i8051.h* files, which you find in the 02_Material/ISS folder provided within the homelab.

In *~/Desktop/material/VHDL* place *i8051_all.vhd*, *i8051_all.vhd*, *i8051_alu.vhd*, *i8051_ctr.vhd*, *i8051_dbg.vhd*, *i8051_dec.vhd*, *i8051_lib.vhd*, *i8051_ram.vhd*, *i8051_rom.vhd*, *i8051_tsb.vhd* and *i8051_xrm.vhd* files, which you can find in the 02_Material/VHDL folder.

In *~/Desktop/material/Hex2Rom* place *i8051_mkr.c* file, which you can in the 02_Material/Hex2Rom folder.

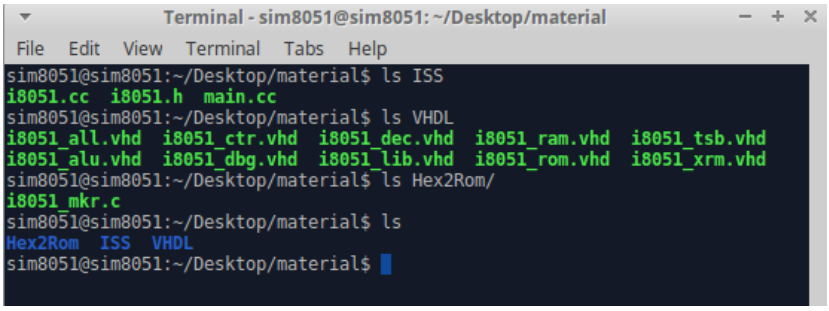


Figure 8 – Material Preparation

2.3. Compilers Tools

2.3.1. SDCC

As already mentioned, SDCC compiler's site is *http://sdcc.sourceforge.net/index.php*; here you can find compiler's manual and have access to download (latest available version is 4.1 at time of writing) section as shown in Figure 9.

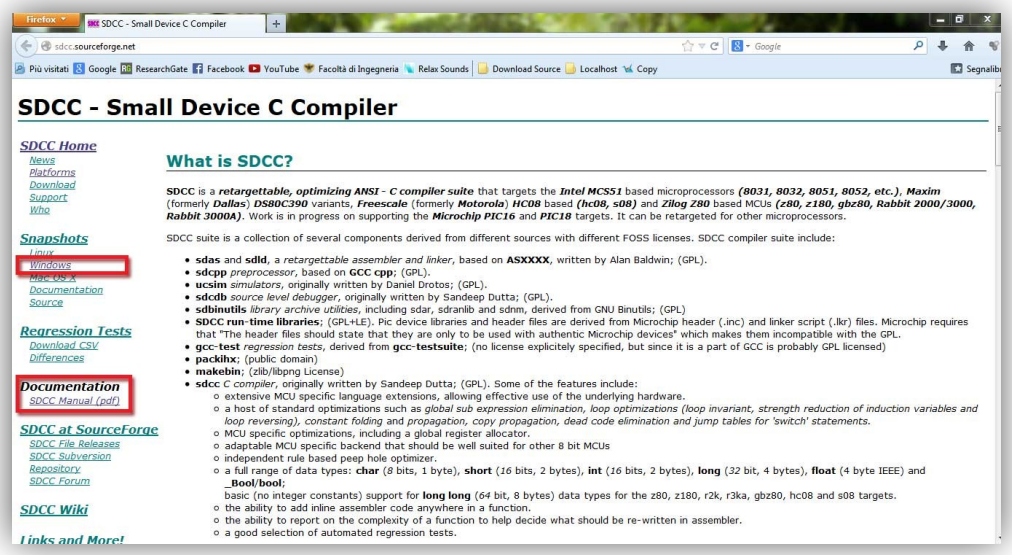


Figure 9 – SDCC website

Ubuntu Installation

For installation on Ubuntu-based machines (ex. The virtual machine provided for the course), SDCC can be installed with:

```
> sudo apt install sdcc
```

Manual Installation (Linux)

For **manual** SDCC installation on Linux based environments see the following steps.

Move (cut and paste) your SDCC object file (already decompressed) on your Desktop.
Open terminal and access Desktop folder.

```
> cd ~/Desktop
```

Login as root, provide root password when requested[1]:

```
> sudo su
```

Check sdcc-4.1.0 by typing

```
> ll ./sdcc-4.1.0
```

[1] Remember: user passsword set in Error: Reference source not found is *sim8051*. If you are using giacomo's virtual machine then password is *aswasw*

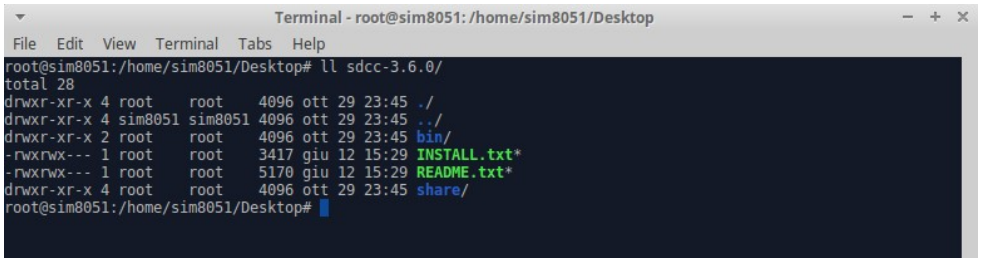


Figure 10 – SDCC permissions check

If you find out that the owner is root then you will need to change owner and permissions to all files and subdirectories contained in ./sdcc-4.1.0.

Type and execute command:

```
> chown –R user[2] ./sdcc-4.1.0
```

Now grant full permission by typing

```
> chmod 0777 –R user ./sdcc-4.1.0
```

Usually this is not a good way to proceed but it will grant full access to SDCC capabilities.

Now copy all sdcc-4.1.0 content into /usr/local/ with following commands:

```
> cd sdcc-4.1.0
> cp -r * /usr/local
```

Now consider that SDCC linux release is available only for 32 bit environment.
If you have a 64 bit linux environment the quickest solution is to install 32bit compatibility libraries.
If on Ubuntu-based distrubution, you can do this by following commands:

```
> sudo dpkg --add-architecture i386
> sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386
```

To check whether SDCC installation was successful type:

```
> sdcc -v
```

If the installation was successful, command should return:

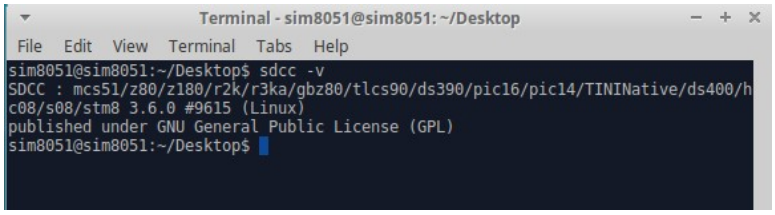


Figure 11 – SDCC successfull installation

Disconnect from root login

```
> exit
```

2.3.2. Keil

To freely download compiler from the web, open Keil website and get through the download page https://www.keil.com/download/product/.

[2] *user* is just a placeholder that student shall substitute with the name of the linux environment user. In pictures the user name is *sim8051*.

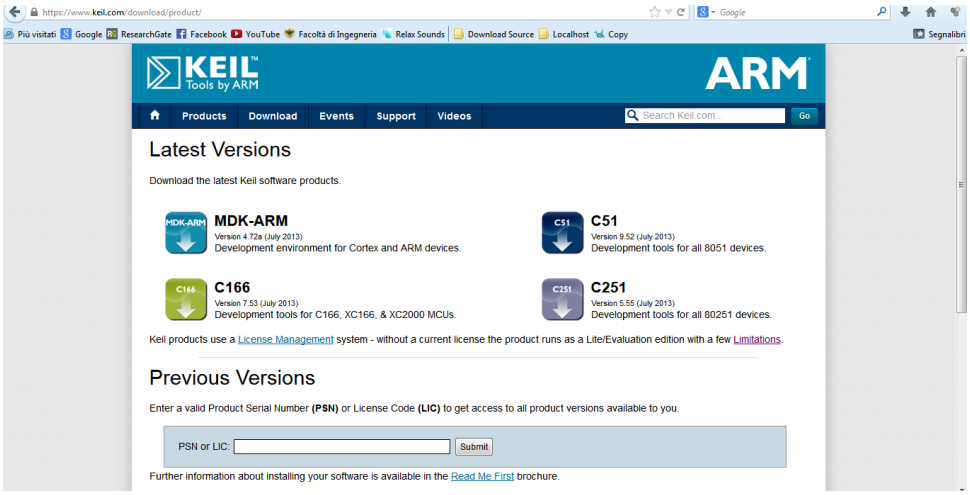


Figure 12 – Keil download page

Keil compiler is a tool developed for the only Windows environment. It is possible to use Keil on Linux through the wine (https://www.winehq.org/). Once wine is installed you can install the Keil four version (actual is five) also on Linux environment, this helps in order to have a single environment provided with all necessary tools.

Wine

Wine software is available for Linux systems, not for Windows.

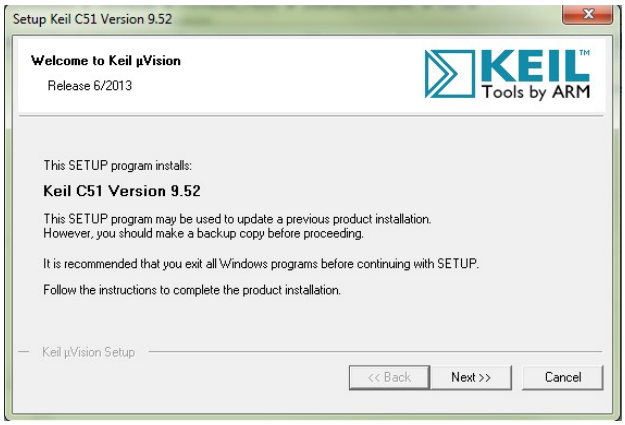
> *Wine enables Linux, Mac, FreeBSD, and Solaris users to run Windows applications without a copy of Microsoft Windows*

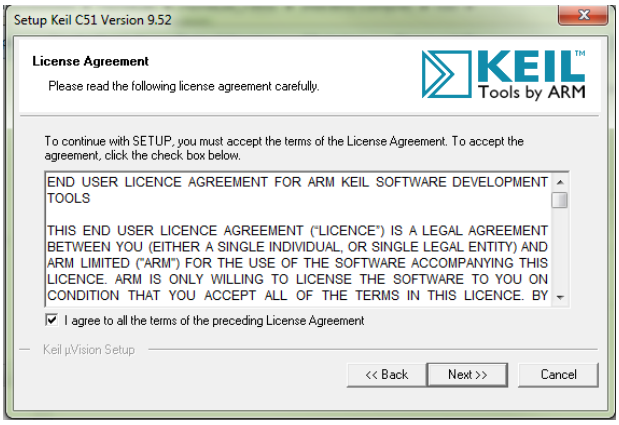
Installation procedures are available at: https://wiki.winehq.org/Ubuntu.

Keil Installation

Click on C51 item and proceed with registration until download executable file C51vXXX.exe and the uav4.pdf manual. .exe file. C51vXXX.exe allows to install both compiler and IDE.
Please, before proceed, consider that registration procedure ask for personal information such as phone and mail. Those info will be used from Italian Keil's satellites companies to contact registered user.

Let's start C51vXXX.exe downloaded file, a wizard will come up to guide through software installation. In Figure 13 are the snapshots for steps to be taken.





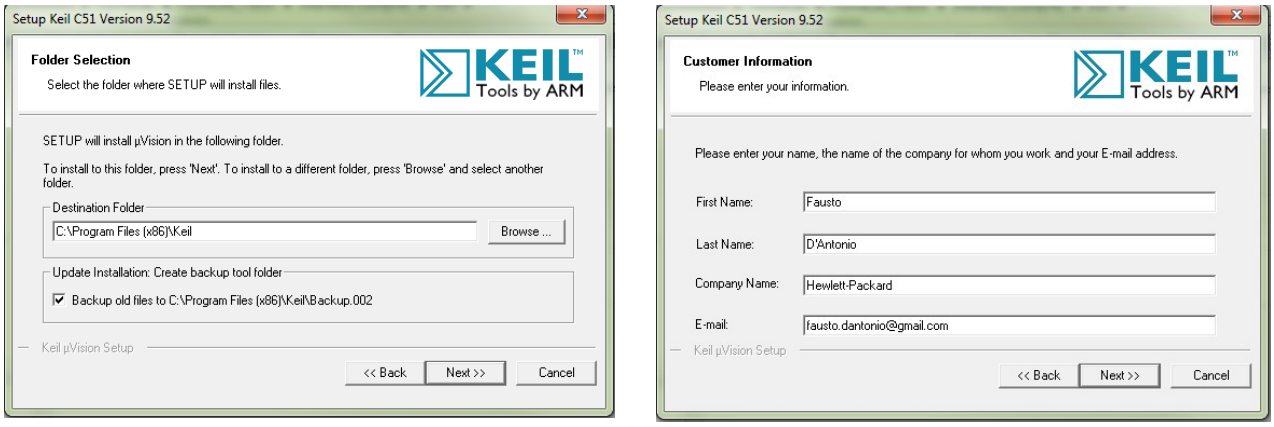


Figure 13 – Keil installation wizard

2.4. Tools for Simulation

2.4.1. Tools for Sw Simulation: Building ISASim

To get Instruction Set program available for execution, it needs to compile ISS source code; there are many ways to do that. The one we propose builds ISASim with the command line interface (CLI); see *Building ISASim: using Command Line Interface*.

Building ISASim: fixing code

Before build, is necessary to update and modify some code details.
ISS code was developed before official C++ standard was modified, indeed this update impacted on C++ compiler. Furthermore, different C++ compiler implementation may have some different interpretation of some code details. The needs to apply changes to code is almost natural.
To build ISS with last c++ compilers, code require following changes.
With reference to ISS folder in *~/Desktop/material/ISS*, a first change has to be done on the header file i8051.h. Add at line 25 the instruction `using namespace std`.

```
22  //#define DETAIL
23  […]
24
25
26
```

```
22  //#define DETAIL
23  […]
24
25  using namespace std;
26
```

Code 1 – i8051.h header file modifications

Then modify main.cc file, including *cstdlib* library, add following code line `#include <cstdlib>;` as shown in Code 2.

```
14  #include <iostream>
15  #include <signal.h>
16  #include "i8051.h"
17
18
```

```
14  #include <cstdlib>
15  #include <iostream>
16  #include <signal.h>
17  #include "i8051.h"
18
```

Code 2 – main.cc main file modifications

Save changes.

ISS is not provided with a mechanism that allows to understand when simulated program ends, and consequently ISS is not able to automatically stop the program execution.
Methods to end automatically a program are various. Here, one of the simplest but certainly not the most elegant.

In I8051.h file, consider the statement commented reported in Code 3.

```
//#define PROGRAM_COMPLETION ((unsigned char)RAM[P1] == 0x7F)
```

Code 3 – PROGRAM_COMPLETION variable

It means that ISASim terminates the program when P1 register assume 0x7F value (127 in decimal). This condition forces programmers to be sure that simulated programs never set register P1 to the 127 value, otherwise simulation would terminate immediately and improperly.
This instruction defines termination program condition. Such approach allows to trivially terminate programs, but precludes the possibility to set register P1 to 127 during the program execution. This restriction may exceed our will.

Following statement set a program completion condition, it states that all of the four registers have to be set to 0x0.

```
#define PROGRAM_COMPLETION ( ((unsigned char)RAM[P0] == 0x0) && ((unsigned char)RAM[P1]
== 0x0) && ((unsigned char)RAM[P2] == 0x0) && ((unsigned char)RAM[P3] == 0x0))
```

Code 4 – Modified PROGRAM_COMPLETION variable

Remove the statement reported in Code 3, with the one in Code 4 – Modified PROGRAM_COMPLETION variable.

This approach has the advantage to maintain the program flow under programmer control, but contrarily forces to terminate the program with four assignments. These assignments affect the number of instructions executed and, we will see, the information that ISASim provides.

Building ISASim: using Command Line Interface

We need two different configuration for ISASim executable, we will build it two times with different option.

In i8051 workspace folder, create a new subfolder with *ISASim* name.

```
> cd ~/workspaces/i8051/
> mkdir isasim
```

Move into *isasim* folder:

```
> cd isasim/
```

create *src* and *inc* subfolders

```
> mkdir src
> mkdir inc
```

create *Release* and *ReleaseForDebugging* subfolders

```
> mkdir Release
> mkdir ReleaseForDebugging
```

Copy all .cc files from Dalton Project ISS into *isasim/src/*.

```
> cp ~/Desktop/material/ISS/*.cc ./src/
```

Copy all .cc files from Dalton Project ISS into *isasim/src/*.

```
> cp ~/Desktop/material/ISS/*.h  ./inc/
```

ISS source code files are in *ISS* folder provided within material of this homelab.

Building Release configuration

Execute following commands:

```
> g++ -I "./inc" -O3 -Wall -c -o "./Release/i8051.o" "./src/i8051.cc"
> g++ -I "./inc" -O3 -Wall -c -o "./Release/main.o" "./src/main.cc"
> g++ -I "./inc" -o "./Release/ISASim" ./Release/*.o
```

ISASim source code composes of two .cc files: main.cc and i8051.cc. We will compile them separately and then link them together.
First of all compile and assemble i80151.cc file, with the following command:

```
> g++ -I "./inc" -O3 -Wall -c -o "./Release/i8051.o" "./src/i8051.cc"
```

This command uses **g++** compiler, builds `"./src/i8051.cc"` by using following flags:

- `-I "./inc"`
  Search for .h files into `./inc` subfolder (subfolder path refers to directory where g++ is called).
- `-O3`
  Set the maximum level of optimization.
- `-Wall`
  Display all warnings.
- `-c`
  Compile and assemble but do not link the input file.
- `-o "./Release/i8051.o"`
  Place output `i8051.o` file in `./Release` folder.

Similarily, compile and assemble main.cc file:

```
> g++ -I "./inc" -O3 -Wall -c -o "./Release/main.o" "./src/main.cc"
```

Link obtained object files with the following:

```
> g++ -I "./inc" -o "./Release/ISASim" ./Release/i8051.o ./Release/main.o
```

Building ReleaseForDebugging configuration

Compile and assemble i80151.cc file, with the following command:

```
> g++ -I "./inc" -O3 -Wall -c -o "./ReleaseForDebugging/i8051.o"
  -DDEBUG -DDEBUG_PC -DDETAIL "./src/i8051.cc"
```

Similarily, compile and assemble main.cc file:

```
> g++ -I "./inc" -O3 -Wall -c -o "./ReleaseForDebugging/main.o"
  -DDEBUG -DDEBUG_PC -DDETAIL "./src/main.cc"
```

Link obtained object files with the following:

```
> g++ -I "./inc" -o "./ReleaseForDebugging/ISASim" -DDEBUG -DDEBUG_PC
  -DDETAIL ./ReleaseForDebugging/i8051.o ./ReleaseForDebugging/main.o
```

You can enable the production of debugging information as a simulation output, by defining `DEBUG`, `DEBUG_PC` and `DETAIL` macros.
Flags `-DDEBUG -DDEBUG_PC -DDETAIL` allows you to define those Macros during building phase.
Alternatively, you can remove comments at 20, 21 and 22 lines in i8051.h file as follows, then build without `-DDEBUG -DDEBUG_PC –DDETAIL` flags:

```
20  //#define DEBUG
21  //#define DEBUG_PC
22  //#define DETAIL
```

```
20  #define DEBUG
21  #define DEBUG_PC
22  #define DETAIL
```

Code 5 – ISASim debug, condition enabling

2.4.2. Tools for Hw Simulation: Building Hex2Rom

Hex2Rom application may be acquired by build i8051_mkr.c file. *Dalton* project about it:

> *Program to convert an Intel 8051 HEX file into a ROM model, i.e., generates i8051_rom.vhd. You will need to compile this C/C++ file, say, gcc -Wall i8051_mkr.c, then run it with your HEX file as a command line argument to it, e.g., a.out myfile.hex.*

Building Hex2Rom: using Command Line Interface

We need only one configuration for Hex2Rom executable.

In i8051 workspace folder, create a new subfolder with *Hex2Rom* name.

```
> cd ~/workspaces/i8051/
> mkdir hex2rom
```

Move into *hex2rom* folder:

```
> cd hex2rom/
```

create *src*, *Release* and *ReleaseForDebugging(CALIBRI 11)* subfolders

```
> mkdir src
> mkdir Release
> mkdir ReleaseForDebugging
```

Copy all .cc files from Dalton Project ISS into *hex2rom/src/*.

```
> cp ~/Desktop/material/Hex2Rom/i8051_mkr.c ./src/
```

Compile the program with the following line:

```
> gcc -O3 -Wall -o "./Release/Hex2Rom.o" "./src/i8051_mkr.c"
```

3. Compile for Intel 8051

In this section we propose a procedure to compile a program in order to obtain *.hex* format executable file.

Group/Student shall #2: Experiment building opportunities

3.1. Preparing the directory tree

- Prepare directory tree in order to perform building phase.
  - Create a directory in *~/workspaces/i8051/* named *divmulSDCC*
  - Create an empty file in *~/workspaces/i8051/divmulSDCC/* named *divmul.c*
  - Create two more directories in *~/workspaces/i8051/* named *divmulSDCCRefined* and *divmulKeil*

3.2. Program example source code and changes

Dalton project, in addition to ISS and VHDL model, provides a set of program examples useful for testing their products. Each example provided consists in a .c language file and its relative .hex executable file.
We put our attention on *divmul* program example, it executes integer division between two numbers.
Source code is in Code 6.

Copy it into *~/workspaces/i8051/divmulSDCC/*divmul.c

```
/*--------------------------------------------------------------------------*/
#include <reg51.h> // To use within KEIL compiler
//#include <8051.h> // To use within SDCC compiler
/*--------------------------------------------------------------------------*/
void main() {
    unsigned x = 134;
    unsigned y = 1;
    unsigned q, r, p, i;
    for(i=0; i<12; i++) {
        y++;
    }
    q = x / y;
    r = x % y;
    p = q * y + r;
    P0 = q;
    P0 = r;
    P0 = p;
    while(1);
}
```

Code 6 – Source divmul program

Code realizes integer division between 134 and 13.
x variable value is 134, it is divided by y. y, starting from 1, is incremented for 12 times in a for loop. Then quotient q, the rest r and is verified that dividend is equal to sum of the rest and the product of divisor and quotient.

Some changes are proposed to source code in Code 6.
Change the way *divmul* uses registers, by enabling the usage of those that are not used (P1, P2 e P3), such that it will be possible to see and simulate all output ports working.
Is possible to access to I/O ports in the same way as done for registers; by using constants P0, P1, P2 and P3 we can drive device output ports.

Change output port assignment:

```
P0 = q;
P0 = r;
P0 = p;
```

```
P0 = q;
P1 = y;
P2 = r;
P3 = p;
```

Code 7 – New output port assignment for divmul.

Before while(1) instruction add following condition:

```
  P0 = 0;
  P1 = 0;
P P2 = 0;
P P3 = 0;
```

Code 8 – Program Completion condition

When simulation will run in CLI, it will stop before the while(1) instruction, and will avoid the need to interrupt and close the execution by typing ctrl+c input combination.

> Copy divmul.c file from *~/workspaces/i8051/divmulSDCC* into *~/workspaces/i8051/divmulSDCCRefined* and *~/workspaces/i8051/divmulKeil*

3.2.1. SDCC and Keil includes

Depending on compilation toolchain, we need to include different files.
SDCC and Keil compilers provide header files which define constants to model microcontroller's physical aspects, such as memory and registers addresses.

E.g. *reg51.h* (for Keil) and *8051.h* (for SDCC) defines P0 variable and its value to 0x80 which respond to the P0 port register address.
SDCC compiler provide and uses *i8051.h* header file, while Keil provide and use *reg51.h*.
It means that when we will compile using sdcc we will include *i8051.h* header file. While, when we use Keil we will include *reg51.h* file.

3.2.2. Expected results

x and y initial values are 134 and 1 respectively, y variable is incremented by 1 for 12 times reaching value y=13.

Expected results are:

$$q = \frac{x}{y} = 10 = 0x0A = 1010;$$

$$r = x\ \%\ y = 4 = 0x04 = 0100;$$

$$p = q * y + r = 134 = 0x86 = 10000110.$$

3.3. SDCC compiler in action

As already mentioned we will use open source compiler to get the hex file.

3.3.1. Build with plain SDCC

The compiler can be used from CLI (Command Line Interface) with the following command:

```
> sdcc sourcefile.c
```

To verify if it is working, we can get a complete list of available options by typing the command **`sdcc`** in the terminal, or equivalently **`sdcc`** `--help`, getting the list in Figure 14 – Help for SDCC command.

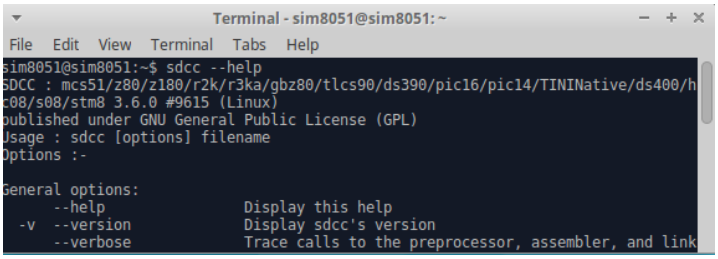


Figure 14 – Help for *SDCC* command

Move into *~/workspaces/i8051/divmulSDCC/* where you have *divmul.c* files.

```
> cd ~/workspaces/i8051/divmulSDCC/
```

Open *divmul.c* file and find instruction `#include <reg51.h>`. Substitute with `#include <8051.h>`.



Figure 15 – Header file substitution

Save changes and quit the editor.
Execute compilation command:

```
> sdcc divmul.c
```

Among others, build command generates divmul.ihx file, it is the final output in hexadecimal Intel. From it we can obtain divmul.hex, as is the executable result of compile process.
To better understand differences between divmul.ihx and divmul.hex files, we can refer to SDCC manual:

> *[…] the Intel Hex file which is generated by SDCC might include lines of varying length and the addresses within the file are not guaranteed to be strictly ascending. If your toolchain or a bootloader does not like this you can use the tool `packihx` which is part of the SDCC distribution,*

The .ihx file is not so different from the .hex file except for arrangement and structure; in other words packihx tool rearrange ihx file contents accordingly to hex format specifications. To use the tool, and thus obtain the file divmul.hex, type command:

```
> packihx sourcefile.ihx > destinationfile.hex
```

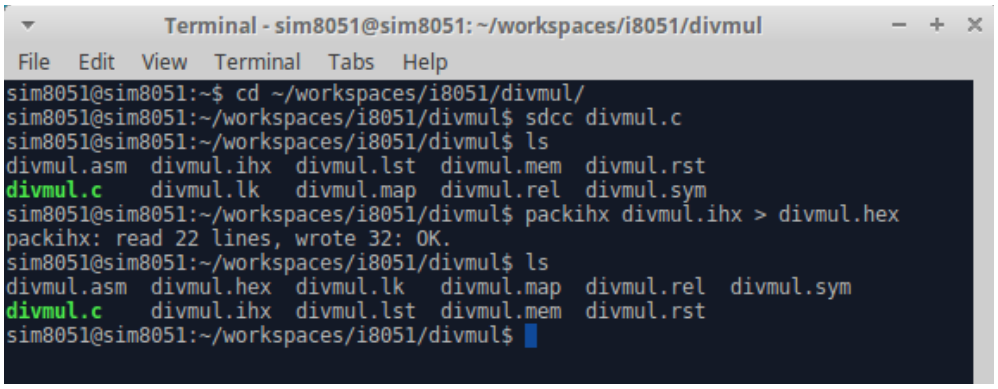


Figure 16 – SDCC building process

Compare divmul.ihx and divmul.hex. The content of the two files is almost the same.
Below, in Figure 17, a section of divmul.ihx and divmul.hex files; coloring highlights similarities and shows how divmul.hex is ordered and organized with respect to divmul.ihx.

divmul.ihx

```
:03000000020006F5
:03005F0002000399
:0300030002006296
:20006200 7E017F007C0C7D000EBE00010F1CBCFF011DEC4D70F28E088F09900086C007C044
:2000820006120 0DAAC82AD83D006D0078E088F09900086C007C006C005C004120120AA82A8
                                                    […]
:04005B00D8FCD9FAFA
:0D00060075810912016DE58260030200039F
:04016D007582002275
:00000001FF
```

divmul.hex

```
:03000000020006F5
:03005F0002000399
:0300030002006296
:10006200 7E017F007C0C7D000EBE00010F1CBCFFD8
:10007200 011DEC4D70F28E088F09900086C007C0FA
:10008200 061200DAAC82AD83D006D0078E088F0943
:10009200 900086C007C006C005C004120120AA82D3
                                                    […]
:04005B00D8FCD9FAFA
:0D00060075810912016DE58260030200039F
:04016D007582002275
:00000001FF
```

Figure 17 - .hex and .ihx comparison

3.3.2. Build with SDCC flags

We compiled divmul.c file using the basic SDCC compiler's settings. Compiler has options that allow to refine the output hex file such that it mostly fits the hardware device where executable is supposed to run. SDCC is a compiler capable of generating the .hex executable for various devices with different features, it is flexible enough to adapt to the needs of each of these. For example, is possible to define the family of the hardware device for which to compile.
We compile code for Intel 8051 microprocessor, whose family is Intel MCS 51. SDCC compiler, support this family and uses it as the default one. In addition it supports among others, families Zilog Z80, Z80 GameBoy, Microchip PIC 14-bit.

You can specify for which family to compile by adding to SDCC command the option -mfamilyname. For example, to compile the file sourcefile.c family MCS 51, we can use the command:

```
> sdcc sourcefile.c –mmcs51
```

Another aspect it is worth to refine is the memory RAM size; it is usual to have the need of informing the compiler of the memory size and structure.

By scrolling down SDCC options list the *Linker options* section is displayed; the option that fits our needs is --iram-size, it allows to enter the RAM size value for target device.

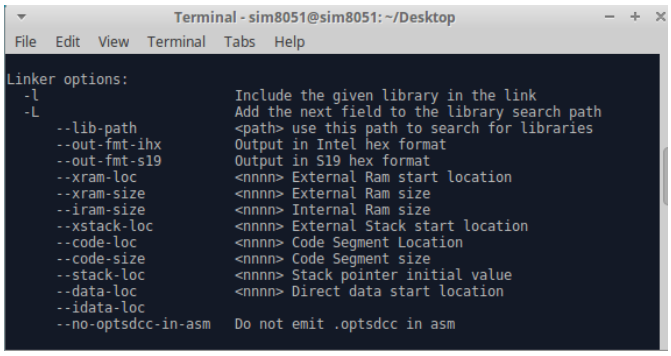


Figure 18 – SDCC linker options

RAM size dimension for our VHDL model is 128 byte; see documentation at
http://www.cs.ucr.edu/~dalton/i8051/

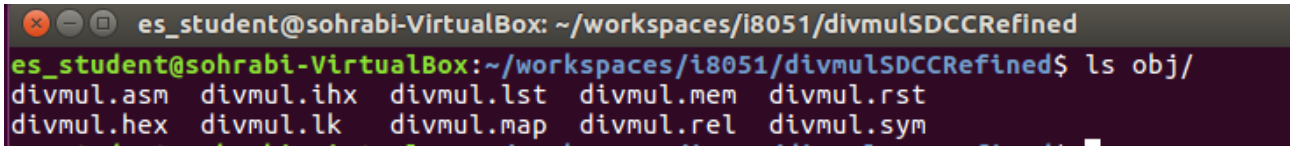
Command to compile for Intel 8051 with 128 byte RAM is

```
> sdcc sourcefile.c –mmcs51 –iram-size 128 –o ./obj/
```

Note the flag `–o ./obj/` move all output files into `/obj/`which shall already exists.
Once we obtained .ihx file, use command packihx to obtain .hex.

```
> packihx ./obj/sourcefile.ihx > sourcefile.hex
```

Apply these steps in *~/workspaces/i8051/divmulSDCCRefined/ to divmul.c file (replace sourcefile.c with divmul.c, same for .ihx and .hex). You should obtain these files in ~/workspaces/i8051/divmulSDCCRefined/obj:*



We can compare the two divmul.hex files, obtained with the procedure just outlined, with the file divmul.hex made available from *Dalton Project*.
Figure 19 is a comparison between the three divmul.hex files; at left the file obtained by the SDCC compiler with no options, in the center the .hex obtained from SDCC compiler with refining option while on the right the one provided by the *Dalton Project*.



Figure 19[3] – Comparison between .hex obtained files

Hex executables produced by different compilers is, of course, different in turn; indeed, among executables, length and structure are different. It is worth to notice that divmul.hex files obtained by SDCC compiler (with and without optimization) have the same structure and the same length; their only difference is a single character on the line 16, as shown in the table in Figure 20.

1	:0101020022DA	:0101020022DA
5	:06003500E478FFF6D8FD9F	:06003500E4787FF6D8FD1F
1	:100013007900E94400601B7A00900171780075	:100013007900E94400601B7A00900171780075
6	A0B3	A0B3
1		
7		
	Divmul.hex without refinement options	*Divmul.hex with refinement opdtions*

Figure 20 – SDCC divmul.hex files comparison

3.4. Keil compiler in action

Open Keil µVision4 program. As shown in Figure 21, the work area is divided into four parts: top menu with toolbars, the Project Navigator to the left, to the right the File Editor and at the bottom Build Output which is the output console.

[3] xyz.hex files were renamed in xyz.txt, due to display issues.

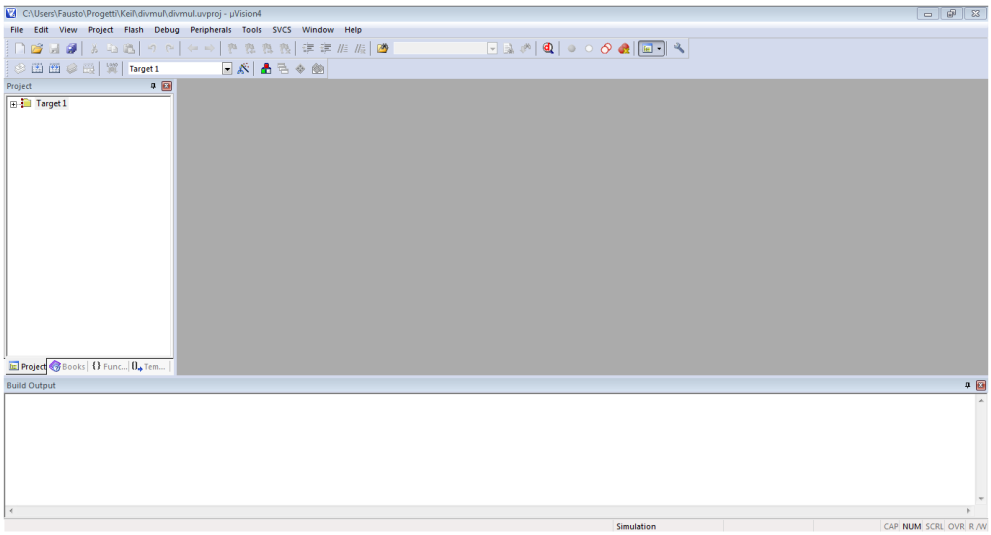


Figure 21 – uVision IDE

Before creating a new project, create an empty folder that contains all the files of our application, copy the divmul.c file in it as shown in Figure 22.

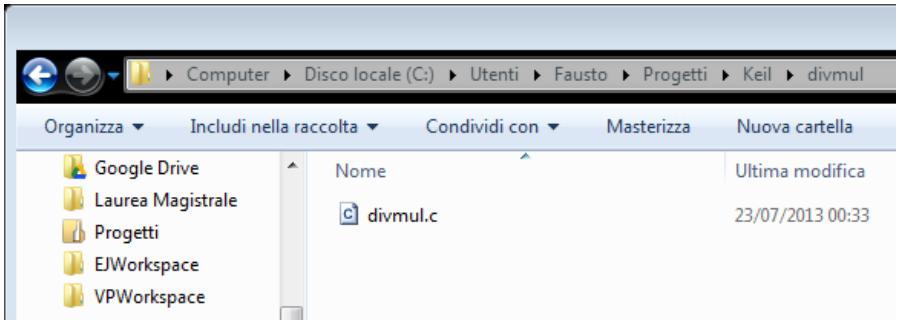


Figure 22 – divmul.c in uVision project folder

Create a new project, from navigation bar select *project->New µVision Project*.

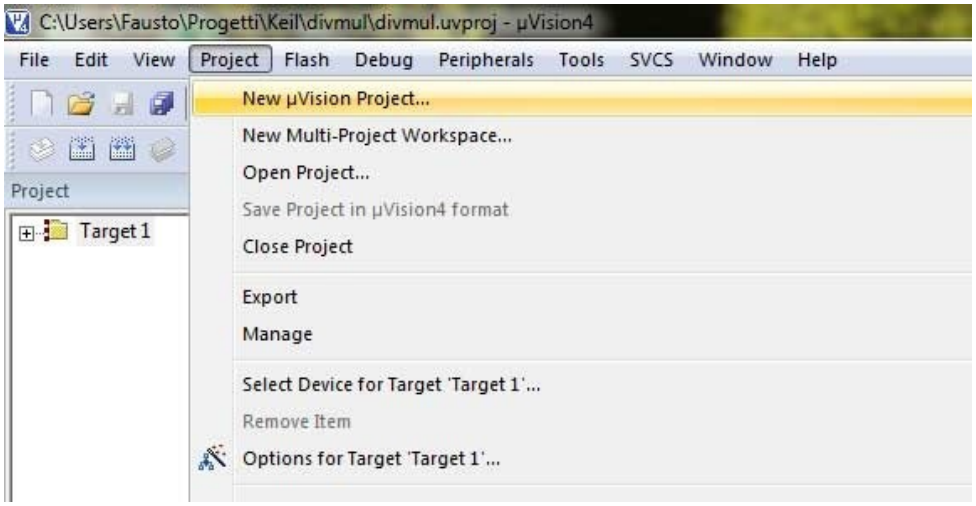


Figure 23 – New uVision project

Select directory previously created and name the project as *divmul.uvproj*.

A window will pop up, select *Generic->8051* as target device.

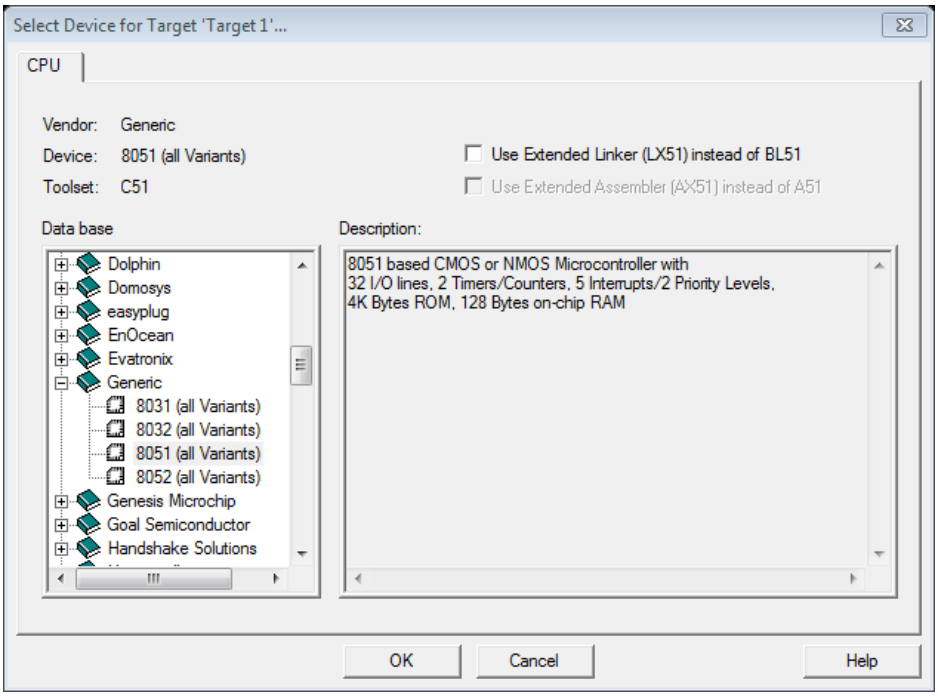


Figure 24 – uVision select target device

When asked for "*Copy 'STARTUP.A51' to Project Folder and add file to Project?*" answer Yes.
Add divmul.c file to project. In Project Navigator click with right mouse button on *Source Group 1* and select *Add existing file to group...* as in Figure 25, select and add divmul.c file from previously created directory.

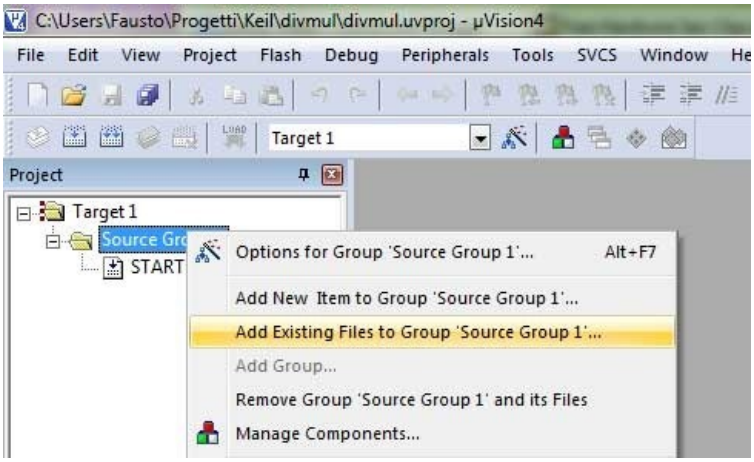


Figure 25 – uVision add files to project

Please remember that this compiler is compliant with reg51.h library and not with 8051.h. Refer to *SDCC and Keil* section.

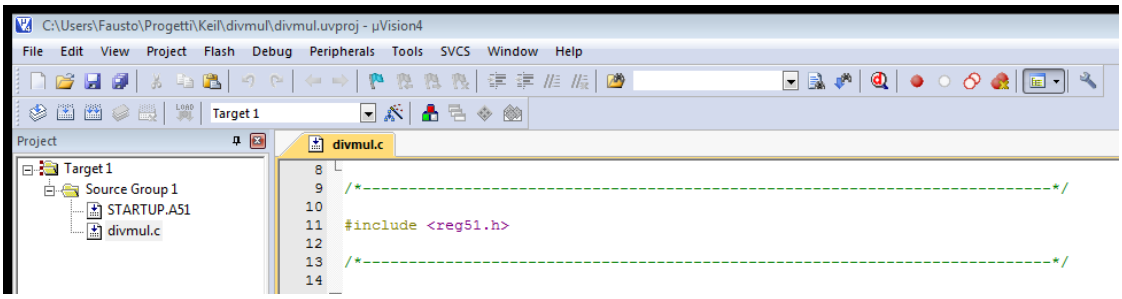


Figure 26 – include reg51.h for Keil compiler

Before build, inform the IDE we want to get .hex output file, go to Project Navigator click right mouse button on *Target1->Options for…*

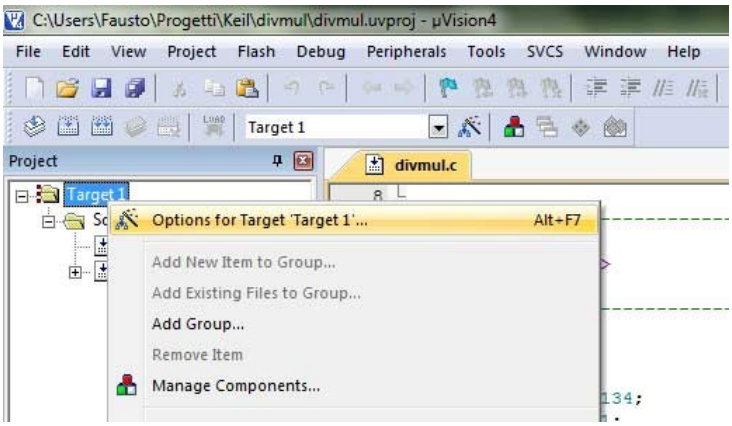


Figure 27 – uVision option for project

In Output tab, put a tick on *Create HEX File*.

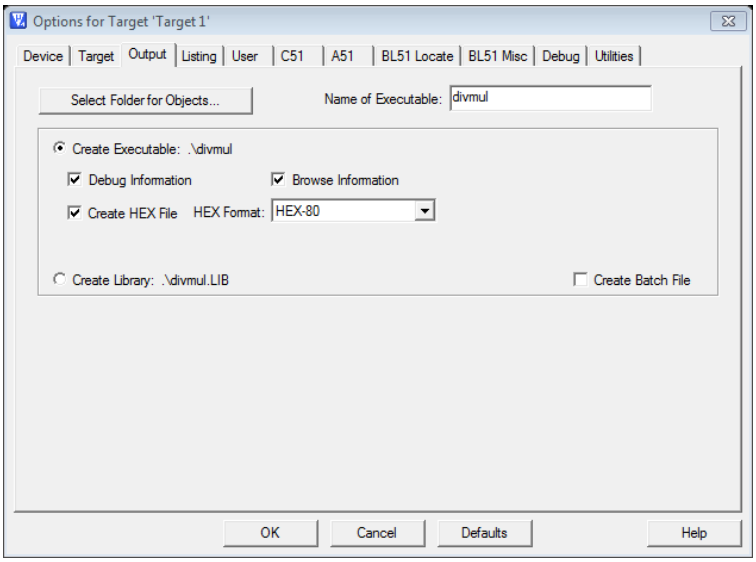


Figure 28 – uVision tick *Create HEX Fiie* option

To compile divmul.c file from menu bar click on *Project->Build target*.

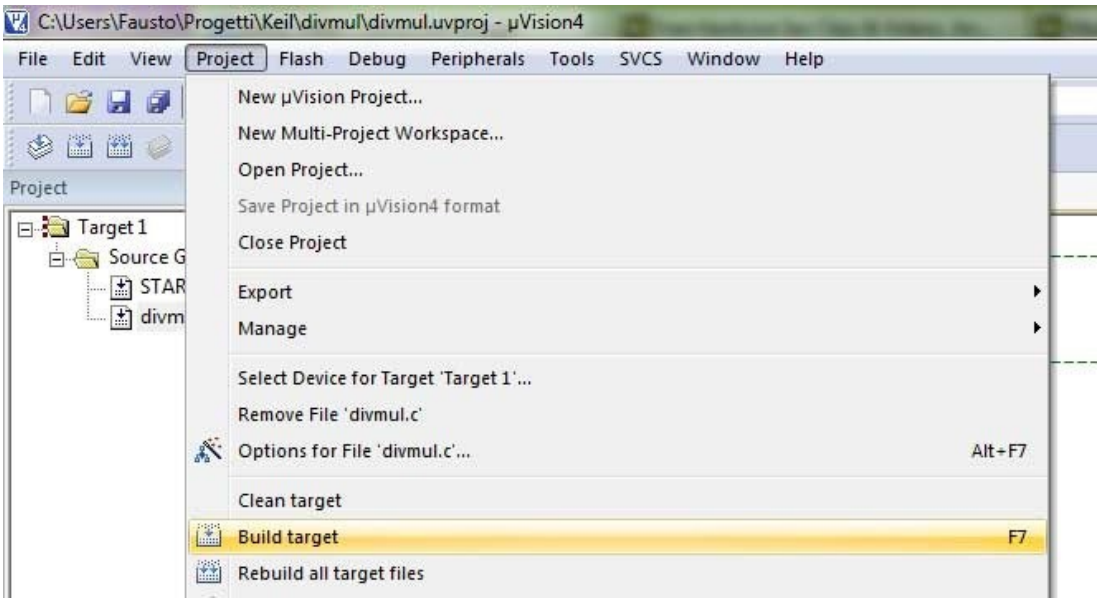


Figure 29 – uVision build target

At the end of building process, in the below *Build output* window will show up the message in Figure 30.

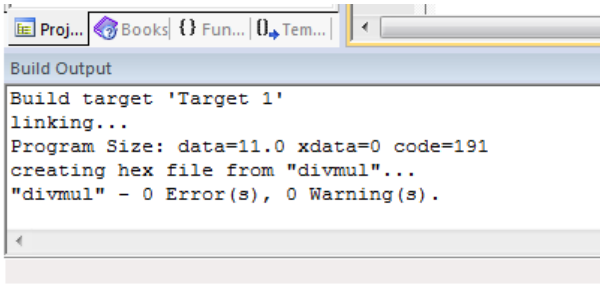


Figure 30 – uVision end building message

While in project folder:

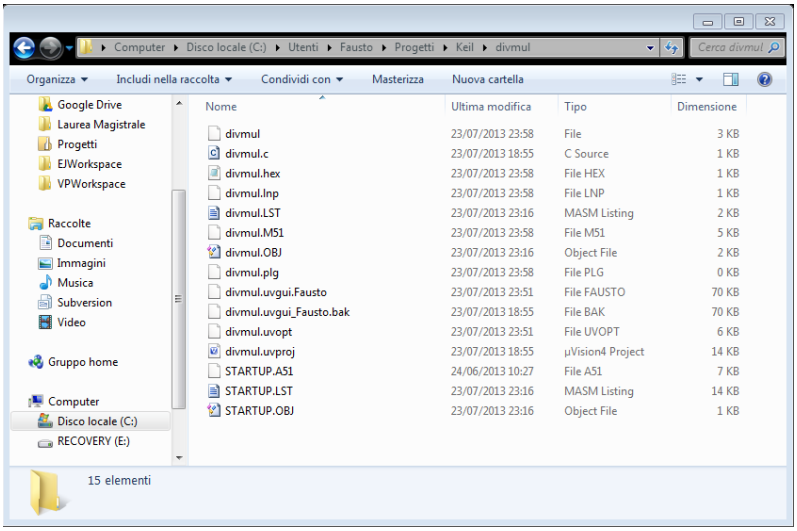


Figure 31 – uVision project folder

Between files created by IDEs in building process, there is also divmul.hex file, which is the file that can be used for simulation via ISS and VHDL model.

From comparison between .hex file just created by building process and divmul.hex file provided by *Dalton Project* we note that the content is different, even if length file seems to match.



Figure 32[4] – divmul.hex Keil produced files comparision

Keil output issue

The .hex Keil output file has a format not compliant with Linux system.
Indeed if we try to run hex2rom program with divmul.hex executable (obtained from Keil compilation process), we obtain following error:

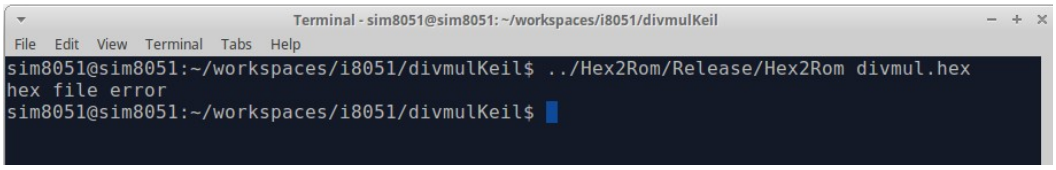


Figure 33 – hex2rom execution on divmul.hex file (obtained from keil)

To understand the problem see the output of command

```
> od -cx divmul.hex
```

It shows the hexadecimal representation of divmul.hex file:

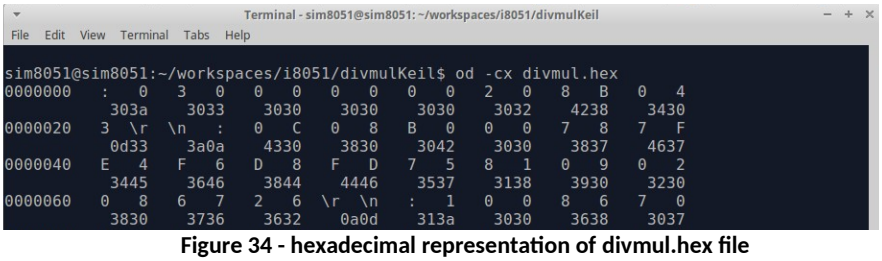


Figure 34 - hexadecimal representation of divmul.hex file

The output shows that End Of Line (EOL) characters are decoded with \r\n which is the format used by Windows systems. Linux environment uses \n

[4] xyz.hex files were renamed in xyz.txt, due to display issues.

The command:

```
> tr -d '\015' < divmul.hex > divmul_EOL.hex
```

substitutes the \r\n with \n in file divmul_EOL.hex as shown in figure:

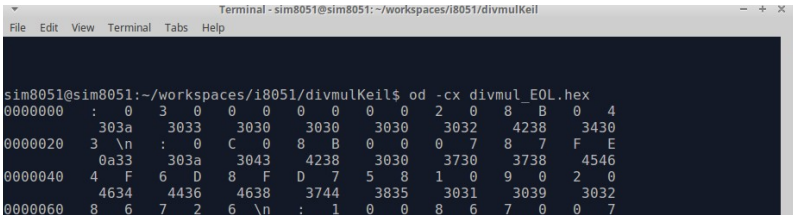


Figure 35 - hexadecimal representation of divmul_EOL.hex file

Now using hex2rom on divmul_EOL.hex work perfectly.

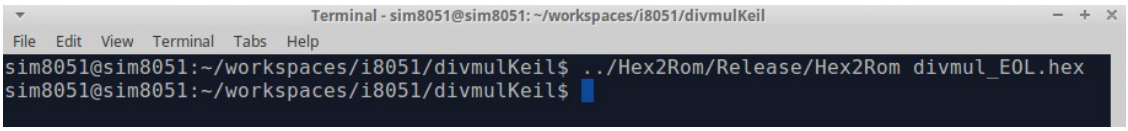


Figure 36 – hex2rom execution on divmul_EOL.hex file

4. Software simulation

According to previous sections, this section shows results of software simulations via the ISASim ISS.
Results refers to the Release and ReleaseForDebugging configurations. For each of the Release and ReleaseForDebugging configurations will be shown the results of the three builds simulations (SDCC with no option, SDCC with option, Keil).
Information provided by ISS, as a result of the simulation, are about registers' values evolution (Figure 37 – ISASim I/O ports evolution) and simulated program performances (Figure 38 – ISASim simulated program performances).

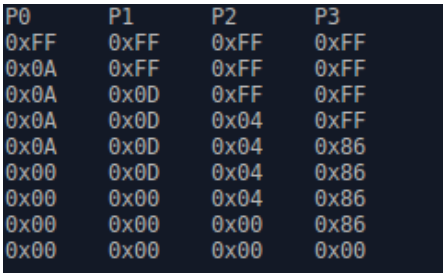


Figure 37 – ISASim I/O ports evolution

Figure 37 shows part of software simulation output. The result shows the evolution of output values on the four I/O ports. Each column refers to one of the four I/O ports available. Whenever register port value is updated, a new row is generated and displayed by ISASim; there is no temporal relationship according to which a new line is displayed.
Note that last four lines do not contribute to the program logic execution, but are part of the closure procedure, of course they impact on performance indices.

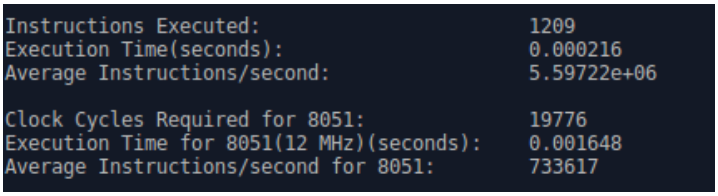


Figure 38 – ISASim simulated program performances

ISASim provides information (reported at Figure 38 – ISASim simulated program performances) at the end of the simulation. There are two kind of information; the first are indices about the host machine performances that run simulation, the latter are the performance projections that would take place at 8051 microcontroller execution.
In detail the provided indices are:

Instructions executed: the number of instructions simulated. In this item, we must consider the four assignments necessary for program closure.
Execution Time: the time, in seconds, that the host computer took to simulate the program instructions.
Average Instructions/second: the average number of instructions, simulated by the host computer, executed per second.
Clock Cycles Required for 8051: the number of clock cycles needed to run the program by 8051 microprocessor.
Execution Time for 8051<12 Mhz><seconds>: the time in seconds that microprocessor 8051 would have taken to execute the simulated program at a frequency of 12 MHz.
Average Instructions/second for 8051: average number of instructions per second that microprocessor 8051 would have run.

Group/Student shall #3: Applying software simulation on built programs

- Run ISASim on divmul.hex file into *~/workspaces/i8051/divmulSDCC/* following steps in section 4.1.1.
- Run ISASim on divmul.hex file into *~/workspaces/i8051/divmulSDCCRefined/* following steps in section 4.1.1.
- Run ISASim on divmul.hex file into *~/workspaces/i8051/divmulKeil/* following steps in section 4.1.1.
- Compare results with the one provided in 4.2..

4.1. Running ISASim

Once ISS code is compiled an executable file is produced, it takes two input parameters:

application-name <hex-file> <output-file>

application-name is the name of ISS executable file.
<hex-file> is the objct code results of building we want to simulate.
<output-file> is the text report file generated (if enabled). It will be generated only for *ReleaseForDebugging* configuration.

4.1.1. Running ISASim on Linux

Execute following commands both in *~/workspaces/i8051/divmulSDCC/ and ~/workspaces/i8051/divmulSDCCRefined/*

Execute Command for simulation without additional debug output information:

```
> ../ISASim/Release/ISASim ./divmul.hex
```

Execute Command for simulation with additional debug output information:

```
> ../ISASim/ReleaseForDebugging/ISASim ./divmul.hex ./divmulReport.txt
```

4.1.2. Running ISASim on Windows

Change to directory *~/workspaces/i8051/divmul*

Execute Command for simulation without additional debug output information:

```
C:/pathToDivmul > C:/pathToISASim/ISASim.exe divmul.hex
```

Execute Command for simulation with additional debug output information:

```
C:/pathToDivmul > C:/pathToISASimForDebugging/ISASim.exe divmul.hex
   divmulReport.txt
```

4.1.3. ISASim debug information report

ReleaseForDebugging simulation in addition to output indices provides a report file.

In Code 9 part of report contents.

```
00000 - LJMP
      LJMP6
00006 - MOV 12
      RAM(129) <- 0x09
00009 - LCALL
      LCALL 387
00387 - MOV 12
      RAM(130) <- 0x00
00390 - RET
      RET 12
00012 - MOV 2
      A <- RAM(130)
00014 - JZ
      if( A == 0 ) JMP 19
00019 - MOV 7
      R1 <- 0x00
00021 - MOV 1
      A <- R1
00022 - ORL 4
      A <- A | 0x00
```

divmulReport by SDCC

```
00000 - LJMP
      LJMP6
00006 - MOV 12
      RAM(129) <- 0x09
00009 - LCALL
      LCALL 387
00387 - MOV 12
      RAM(130) <- 0x00
00390 - RET
      RET 12
00012 - MOV 2
      A <- RAM(130)
00014 - JZ
      if( A == 0 ) JMP 19
00019 - MOV 7
      R1 <- 0x00
00021 - MOV 1
      A <- R1
00022 - ORL 4
      A <- A | 0x00
```

divmulReport by SDCC refined

```
00000 - LJMP
      LJMP191
00191 - MOV 7
      R0 <- 0x7F
00193 - CLR 1
      A <- 0x00
00194 - MOV 13
      RAM(R0) <- A
00195 - DJNZ 1
      R0--
      if( R0 != 0 ) JMP
194
00194 - MOV 13
      RAM(R0) <- A
00195 - DJNZ 1
      R0--
      if( R0 != 0 ) JMP
194
00194 - MOV 13
      RAM(R0) <- A
```

divmulReport by KEIL

Code 9 – divmulReport comparison

In report files there is the list of executed instructions. These instructions clearly refers to the assembly code, even if they are arranged in a quite different way.

Reports of simulations related to programs build with SDCC are very similar; indeed, the only difference is the number of instructions. Contrarily the simulation report of the Keil program clearly differs for instructions and instructions number.

4.2. Software Simulation results comparison

Release configuration simulations

Figure 39 – simulation of divmul build by SDCC (Release configuration), Figure 40 – simulation of divmul build by SDCC with option refinement (Release configuration) and Figure 41 – simulation of divmul build by KEIL (Release configuration) depict results of the simulation performed with ISASim ISS (build in Release configuration).

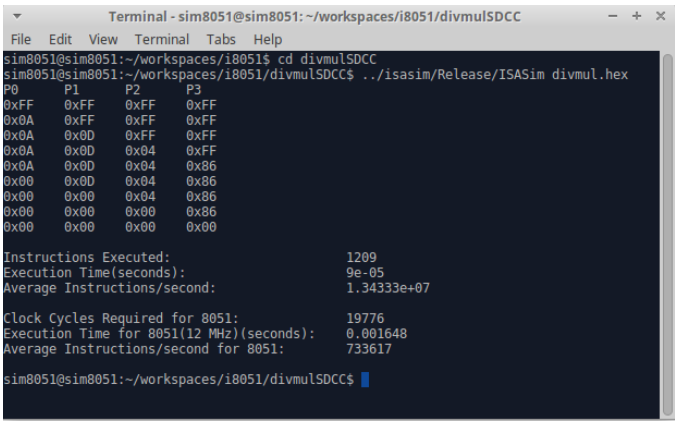


Figure 39 – simulation of *divmul* build by SDCC (Release configuration)

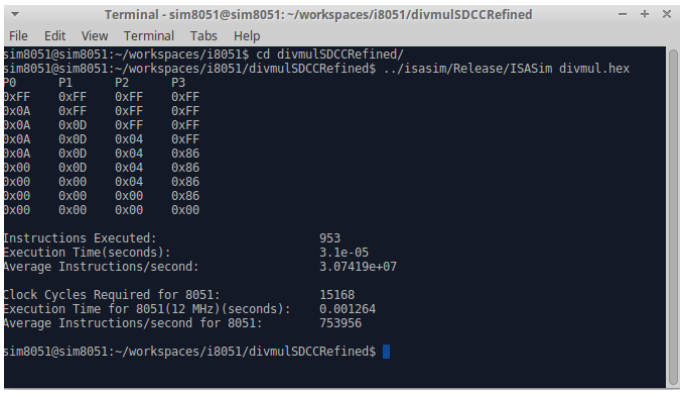


Figure 40 – simulation of *divmul* build by SDCC with option refinement (Release configuration)

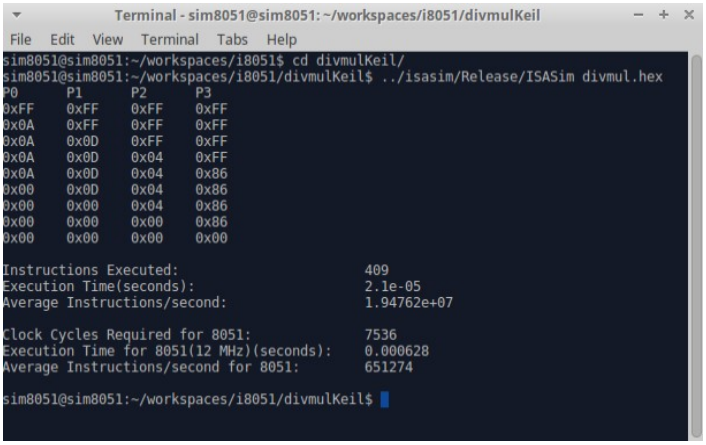


Figure 41 – simulation of *divmul* build by KEIL (Release configuration)

Registers change their values accordingly to *divmul* program statement. Figure 39 – simulation of divmul build by SDCC (Release configuration), Figure 40 – simulation of divmul build by SDCC with option refinement (Release configuration) and Figure 41 – simulation of divmul build by KEIL (Release configuration) show that computational results are the same for all performed builds. Note that last four lines correspond to the completion program procedure. Fifth from the last line indicates the results of the computation performed by *divmul* program. Expected results match in each of software simulations performed.

Looking at Figure 39 – simulation of divmul build by SDCC (Release configuration) and Figure 40 – simulation of divmul build by SDCC with option refinement (Release configuration) we can compare and evaluate SDCC builds, one without refinement option and one with.

Number of executed instruction for SDCC with refinement options is lower, instead host Execution Time seems to be higher. About I8051 projected data, number of clock cycles and execution time are both better for build with refinement options.
These data confirm the benefits derived from refinement options in the building process.

Figure 41 – simulation of divmul build by KEIL (Release configuration) shows that the number of executed instructions by *divmul* simulation build with KEIL, is much smaller than the buildings by SDCC, also the time of running on the Intel 8051 is smaller.

Software simulations show that different compilation (with respect to different copiler and different flags) change i8051 performances projection. Results show that the execution of the program build with Keil is more efficient. In addition SDCC compiler produces more efficient results with refinement options.

ReleaseForDebugging configuration simulations

Figure 42, Figure 43 and Figure 44 shows results of simulations by ISASim ISS build in ReleaseForDebugging configuration.

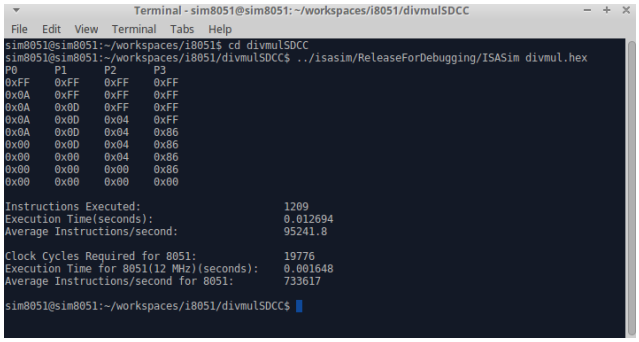


Figure 42 – simulation of Divmul build by SDCC (ReleaseForDebugging configuration)

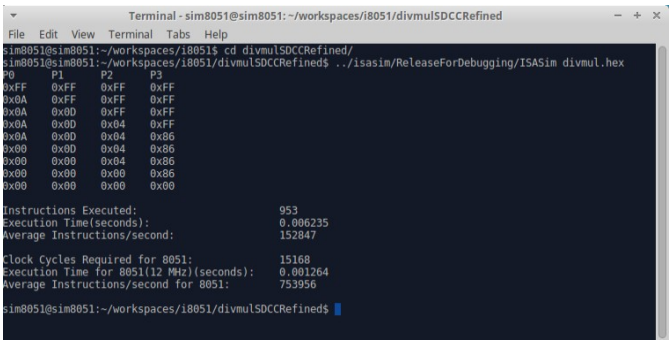


Figure 43 – simulation of Divmul build by SDCC with refinement options (ReleaseForDebugging configuration)

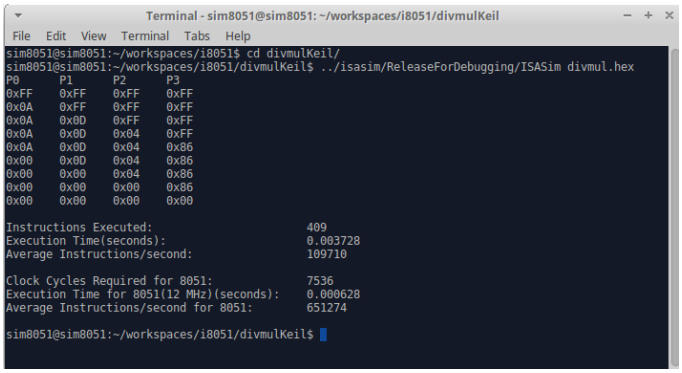


Figure 44 – simulation of Divmul build by KEIL (ReleaseForDebugging configuration)

Simulation results achieved with ISASim build in Release configuration, match perfectly with those produced by ISASim build in ReleaseForDebugging configuration, with the exception of host computer performance indices. Those, in addition to being subject to host computational load at the time of simulation, also depend on operations that ISASim executes during simulation.

5. Hardware simulation

Hardware simulation consists of two main steps: i8051 rom generation using Hex2Rom and

Group/Student shall #4: Applying hardware simulation on built programs

- Run Hex2Rom on divmul.hex file into *~/workspaces/i8051/divmulSDCC/* following steps in section 5.1.1
- Run Hex2Rom on divmul.hex file into *~/workspaces/i8051/divmulSDCCRefined/* following steps in section 5.1.1
- Run Hex2Rom on divmul.hex file into *~/workspaces/i8051/divmulKeil/* following steps in section 5.1.1
- Student shall prepare Vivado Project for i8051 as described in 5.2.
- Copy *~/workspaces/i8051/divmulSDCC/i8051_rom.vhd* into i8051 Vivado project folder, then start behavioural simulation.
- Copy *~/workspaces/i8051/divmulSDCCRefined/i8051_rom.vhd* into i8051 Vivado project folder, then start behavioural simulation.
- Copy *~/workspaces/i8051/divmulKeil/i8051_rom.vhd* into i8051 Vivado project folder, then start behavioural simulation.
- Compare results with the one provided in 5.3.

5.1. Running Hex2Rom

Once program is compiled we get an executable file that takes one input parameter:

application-name <hex-file>

application-name is the name of executable file.

<hex-file> is the object code results we want to build the ROM for.

The command creates a file named *i8051_rom.vhd*, it is the VHDL model memory filled with instructions of *<hex-file>* compiled code.

5.1.1. Running Hex2Rom on Linux

Apply following commands in all three directories *~/workspaces/i8051/divmulSDCC/*, *~/workspaces/i8051/divmulSDCCRefined/* and *~/workspaces/i8051/divmulKeil/*.

Execute Command for i8051 rom generation:

```
> ../hex2rom/Release/Hex2Rom ./divmul.hex
```

Check whether *i8051_rom.vhd* was correctly generated by typing:

```
> ls -l i8051_rom.vhd
```

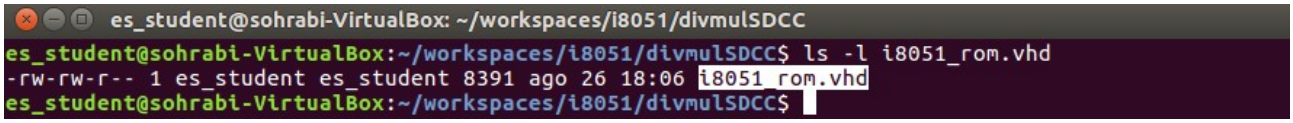


Figure 45 – i8051_rom.vhd generation

5.2. Creating Vivado project for I8051 model

In i8051 workspace folder, create a new subfolder with *i8051* name.

```
> cd ~/workspaces/i8051/
> mkdir i8051
```

Move into *i8051* folder:

```
> cd i8051/
```

create *model* subfolder:

```
> mkdir model
```

Copy all .vhd files from Dalton Project VHDL into *i8051/model/*.

```
> cp ~/Desktop/material/VHDL/*.vhd ./model/
```

Run Xilinx Vivado software, we will realize hardware simulation using it's features.

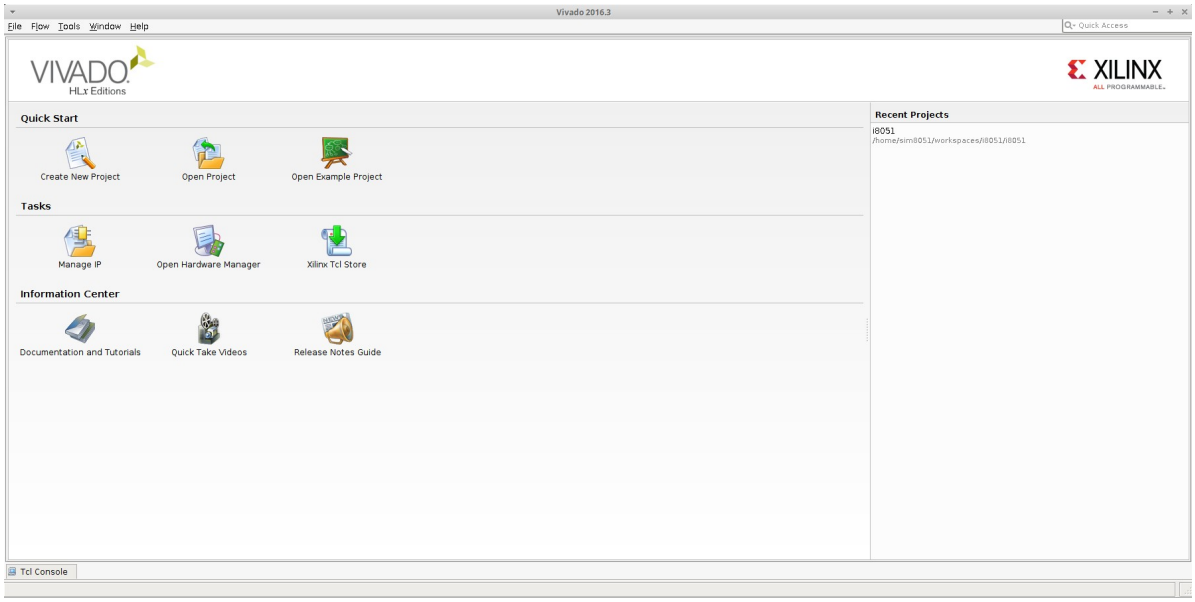


Figure 46 - Vivado main window

Create a project by *File->New Project*...

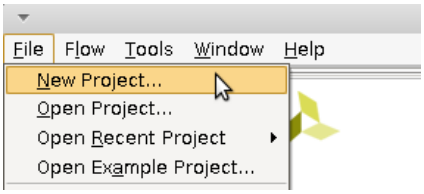
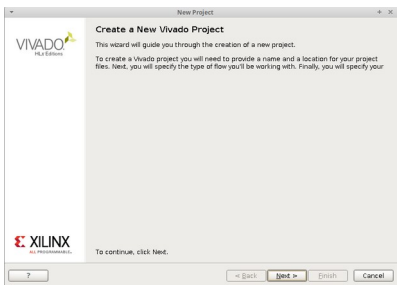
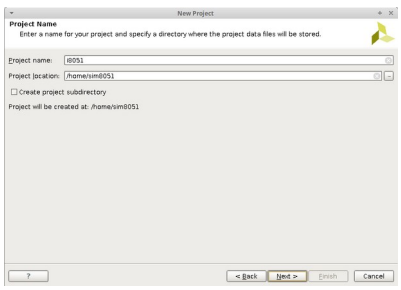


Figure 47 – Vivado new project

Set project name (as *i8051*) and project directory path (as *~/workspaces/i8051/*), then click on *Next*. Select RTL project; indeed we will only have simulation at behavioural level.





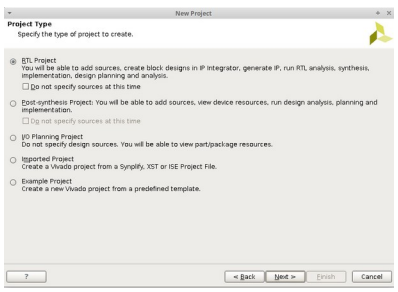


Figure 48 – Vivado new project wizard

Click on *Add files*… and select all .vhd files we moved in *~/workspaces/i8051/i8051/model* directory as a source for this project.

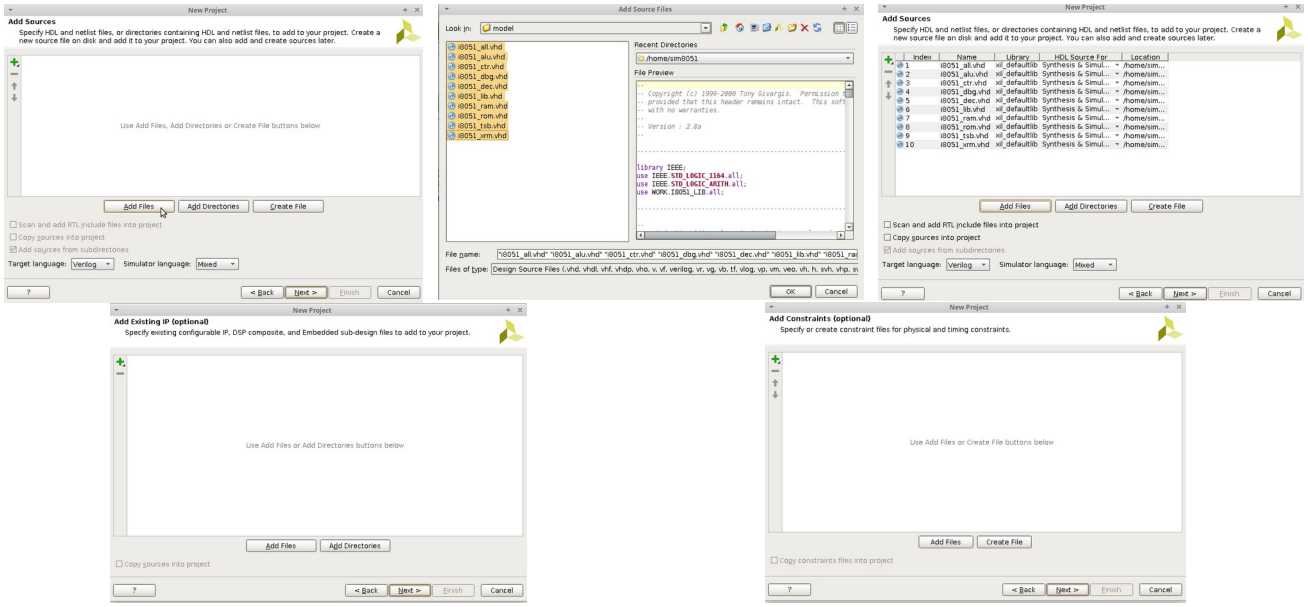


Figure 49 - Vivado selecting model's files

Select *Boards* perspective and select the FPGA board to use within simulation. Choose Artix7.

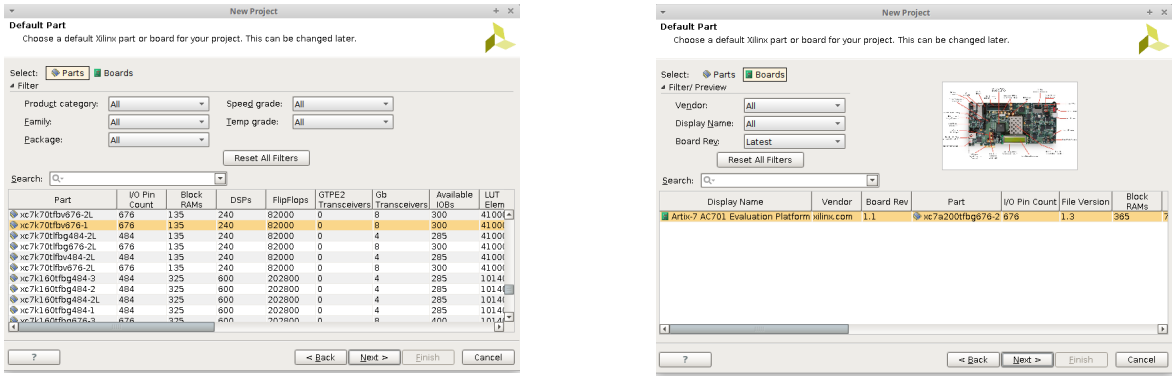


Figure 50 – Vivado board select

Check information about created project and click on *Finish* button.

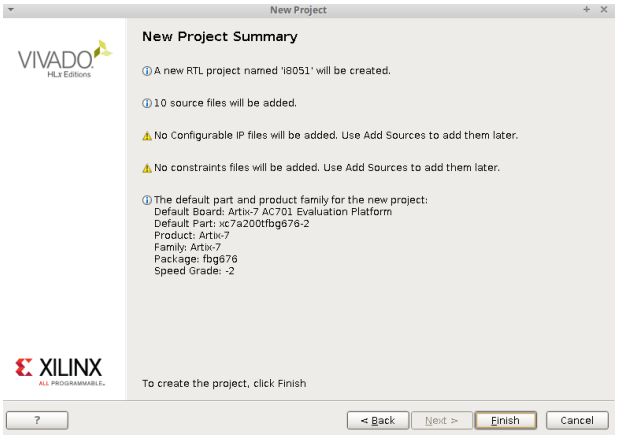


Figure 51 – Vivado project summary

5.2.1. Perform Vivado Hardware Simulation

To start simulation, click on *Run Simulation* in *Simulation*, on the left panel.

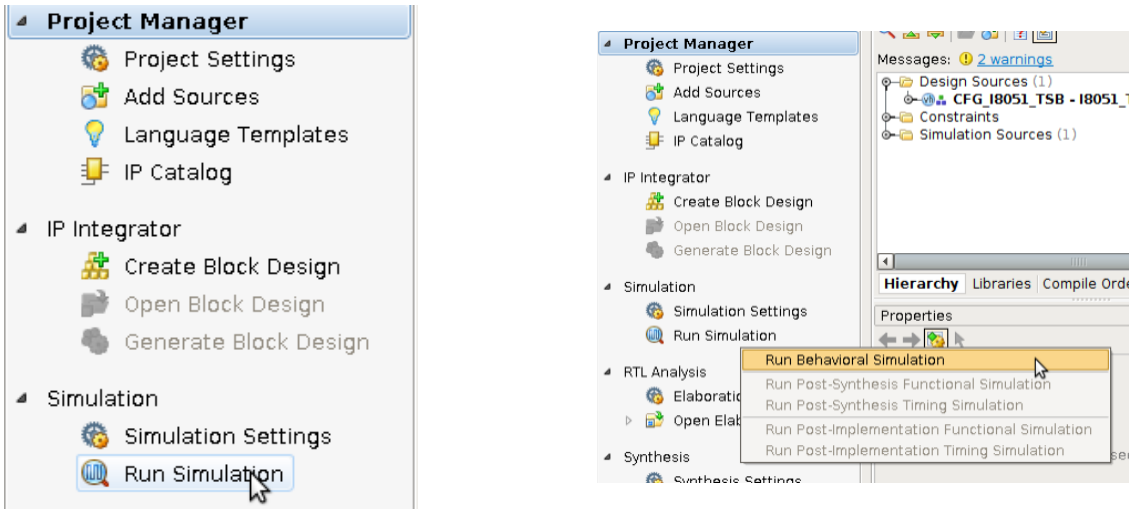


Figure 52 - Vivado Simulation start

Once simulation VHDL simulation process is started automatically will be open a simulation panel.

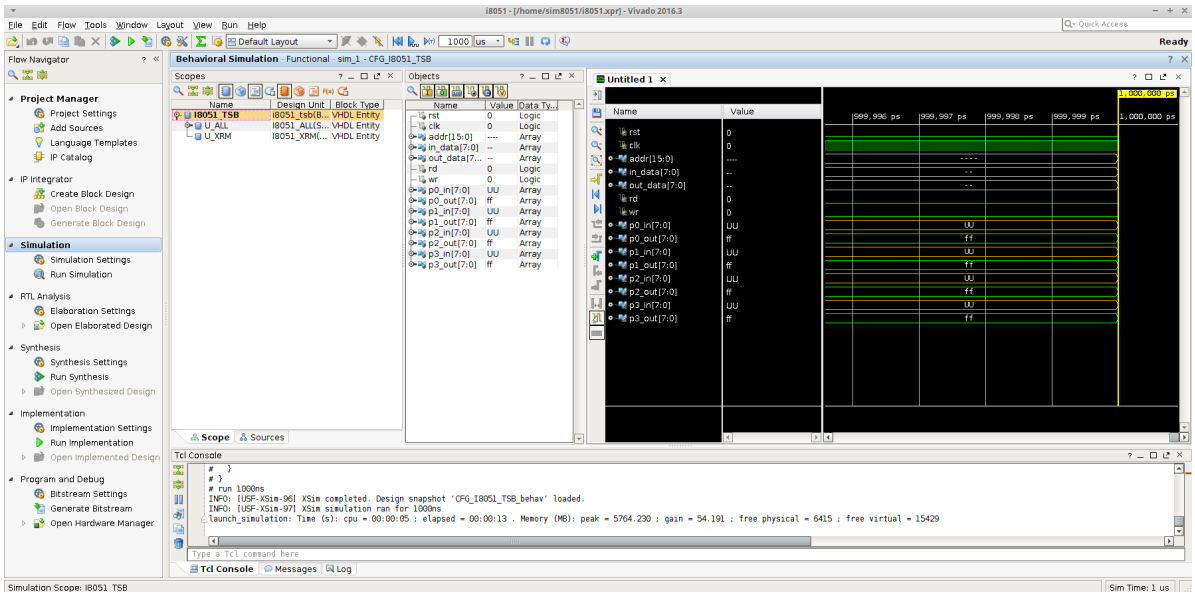


Figure 53 – Vivado Simulation Perspective

To run the simulation you may click on play button on the top right menu bar, this button won't stop the simulation so you will have to stop it manually.
You may also start the simulation only for a specific period of time, which you can set in the form.



Figure 54 – Simulation Time Control

Note that the simulation allows to monitor each signal temporal evolution. p0, p1, p2 and p3 output signals are the signals relative to the four output port.

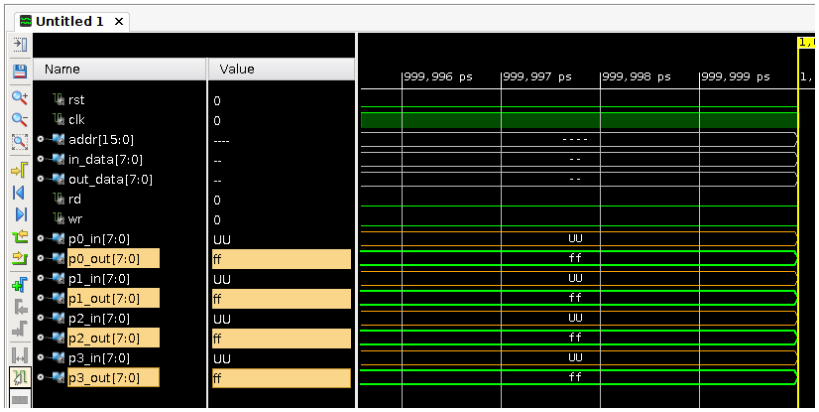


Figure 55 – Simulation output signals

For each displayed signal is possible to analyse every bit it is composed of, as shown in Figure 56.

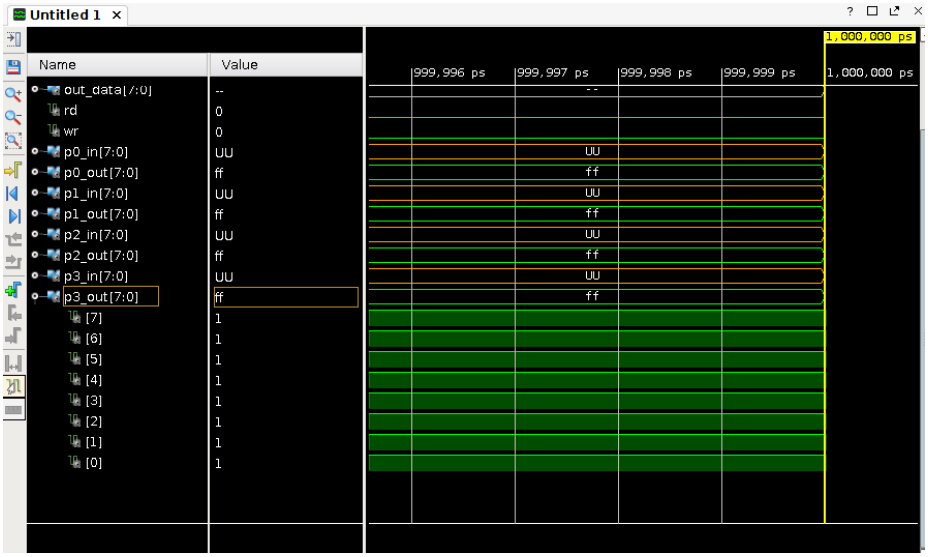
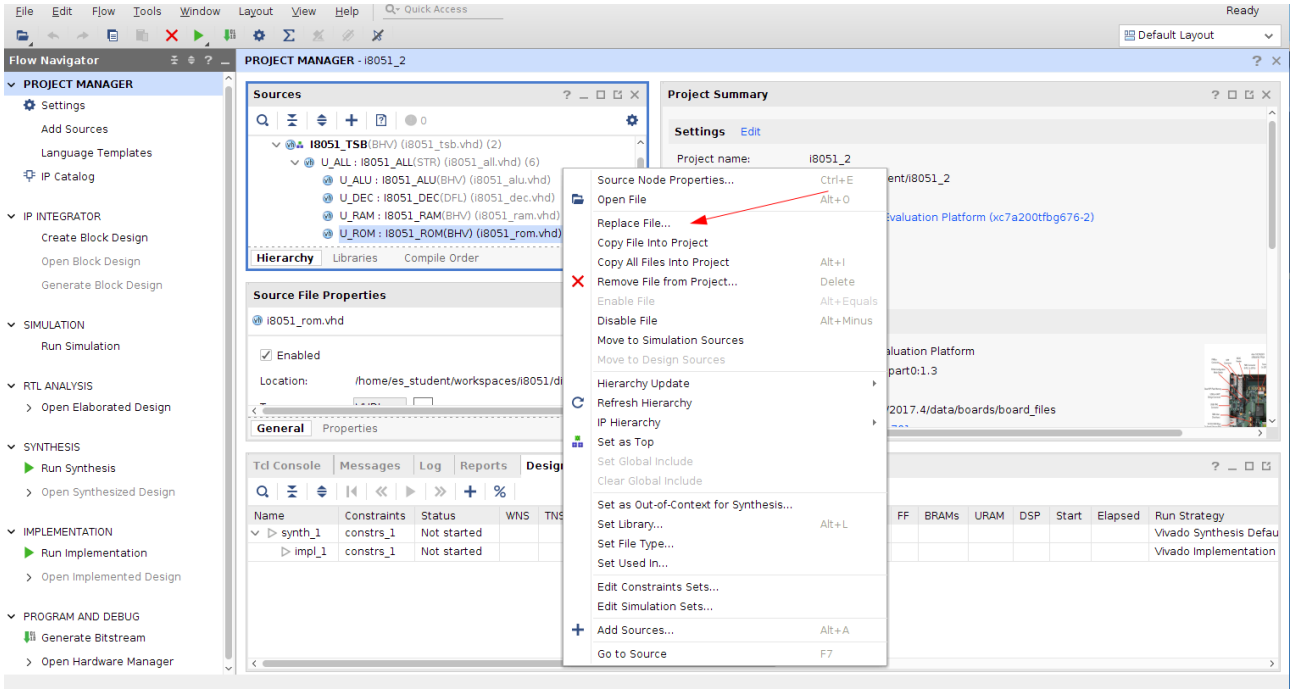
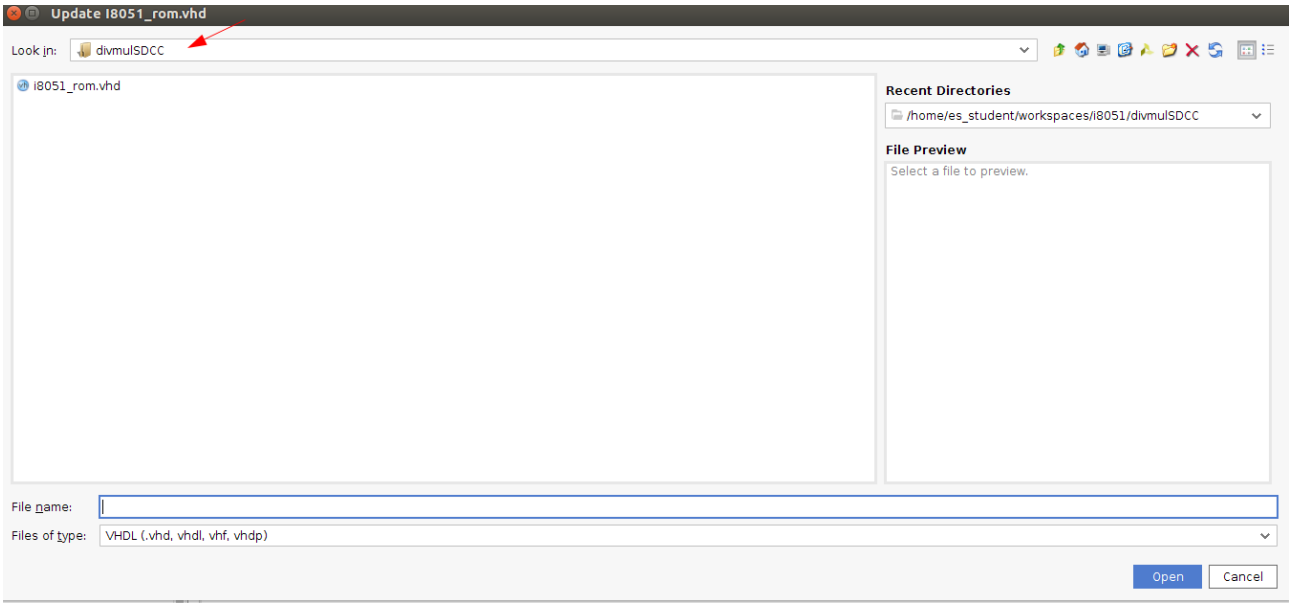


Figure 56 – Simulation output signal details

5.3. Hardware simulation results comparison

For each alternative (divmulSDCC, divmulSDCCRefined and divmulKeil), create a new Xilinx ISE project, by following procedure previously proposed.
Replace *I8051_rom.vhd* file with relative *I8051_rom.vhd* file generated by *Hex2Rom* (as shown in 5.1). To assure replacement takes place, first delete the model file then copy the file generated.



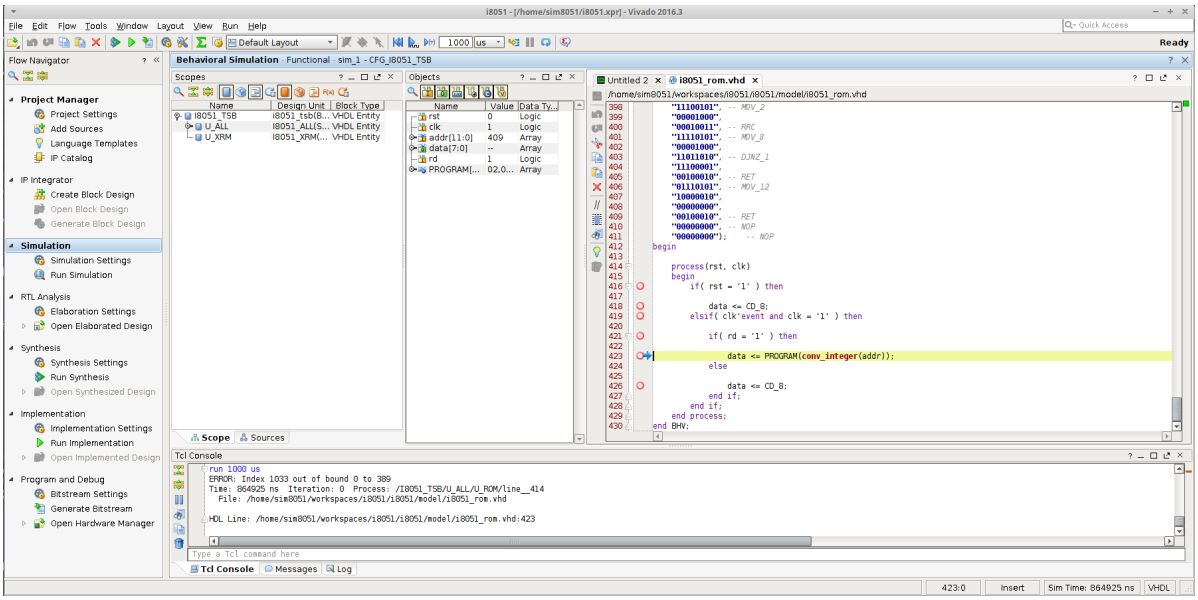


Repeat these steps in each case (SDCC, SDCCRefined and Keil).

Below are the results of hardware simulations.

SDCC simulation

VHDL model simulation of executable build with SDCC without refinement options, runs in error *ERROR: Index 1033 out of bound 0 to 370*, look at Console in Figure 57Figure 57 – hardware simulation of Divmul build by SDCCError: Reference source not found.



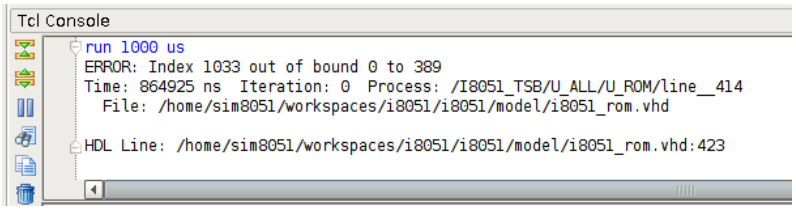


Figure 57 – hardware simulation of Divmul build by SDCC

As stated in message text the problem lies in the device memory size. The RAM memory is in fact too small to allow each instruction, contained in the file divmul.hex, to correctly allocate information.

This error is because the .hex executable, hardware simulated program, was build with SDCC without providing target device information. SDCC compiler can compile C code for device families, but not for specific devices.

SDCC with refinement options simulation

VHDL model simulation, for executable build with SDCC with refinement options, is depicted in Figure 58.

Due to the fact that values change very quickly, to see the results zoom in near the start of the simulation and look for changes in the register values.

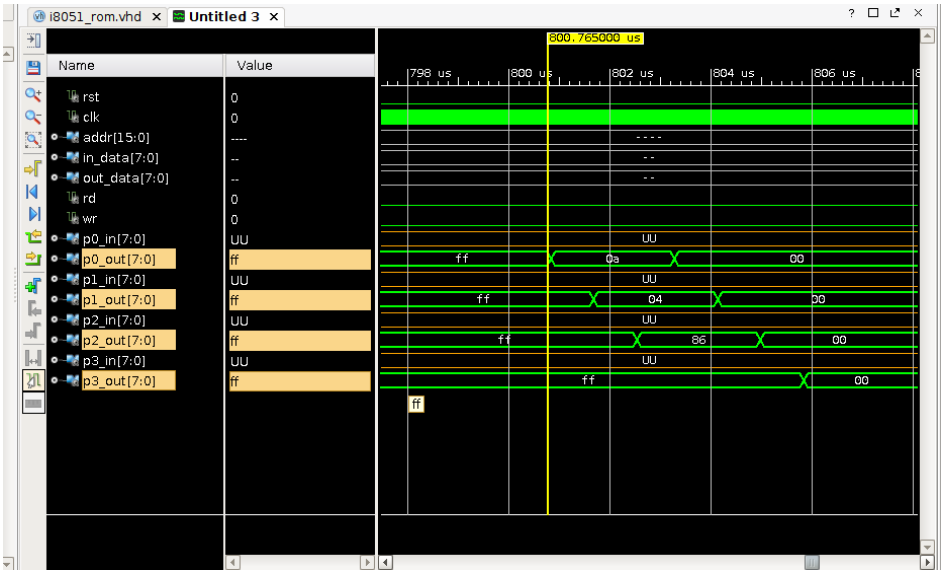


Figure 58 – hardware simulation of Divmul build by SDCC with refinement options

This simulation, differently than previous one, ended without error. The P0, P1, P2 and P3 output signals undergo two commutations.
The first commutation sets registers value with computation results, those are correct. The second commutation sets registers to 0, according to program completion procedure defined for ISASim.
The execution time of the program on microcontroller is 745 us.

Keil simulation

The simulation by VHDL model, for executable build with Keil, is depicted in Figure 59.

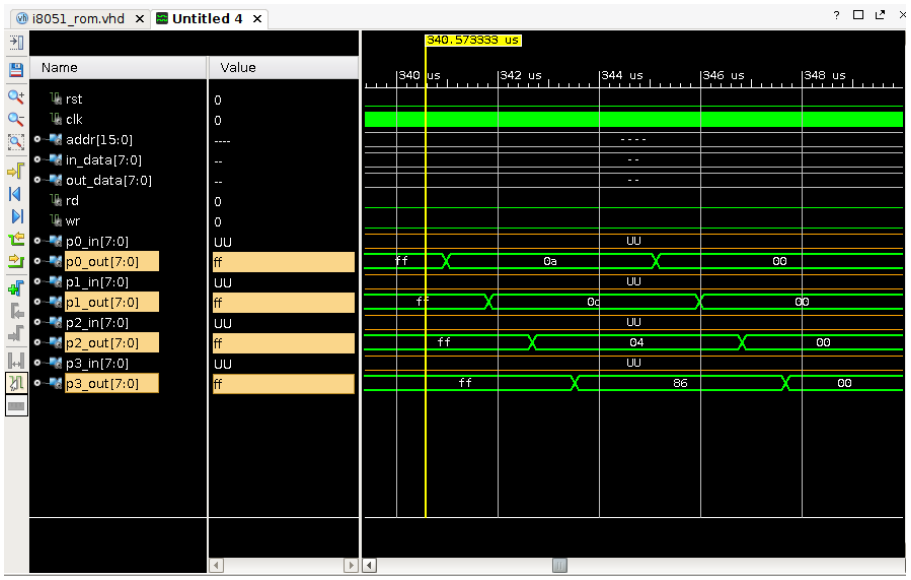


Figure 59 – hardware simulation of Divmul build by Keil

Simulation results match expectations, and are correct. Simulation time is 343 us.

6. Conclusions

Homelab aims to explore the Intel 8051 microcontroller world. Using tools and models made available by *Dalton Project* from University of California, we have built an environment which allows to develop applications for Intel 8051 microprocessor. This approach grant fast test and application prototyping by a heavy reduction of time to market and development cost..

During building process we compared different compilers and focus our attention on SDCC and Keil.
The SDCC compiler is an open source software, thanks to its high configurability, it well suited to our needs.
The Keil compiler with its IDE, despite license limitations, was easy to use and very efficient. Even if not reported in this document, mikroC Pro for 8051 compiler was also experienced, but only allowed us to successfully simulate by ISS.

Software simulation allows to get information about program implementation.
Information provided regards both host machine and 8051 microprocessor; it provides host machine timing performance and projected data for execution on 8051 microprocessor. It also displays output port values.
Performed simulations show that functionally all results were successful, while by a performance point of view the simulation by Keil was the most efficient both in time and in therefore in number of executed instructions.
Information obtained from software simulation do not return too much detail but provide a general idea about program execution. They are particularly suited to check performed computations and code optimization.

Hardware simulation allows to observe entire signals time evolution for microcontroller 8051. Each signal described in VHDL model, can be plotted and viewed in every bit at every instant of simulation time.
First hardware simulation proposal highlights a problem with the size of the memory that the ISS ISASim had not found. Instead simulations of divmul.c that was build with sdcc with options refining and with Keil have produced the correct expected results. Simulation program build by Keil was more performant.
Detail of information provided is very high, but hardware simulation is very costly for machine that runs it.

In conclusion, SDCC and Keil compilers were both suitable for Homelab purposes, SDCC was less performant than Keil, but of course cheaper.
Quality and detail level of produced information by hardware simulation is much higher than software simulation. However time and load of hardware simulation is very onerous. Contrarily software simulation disclose less valuable but easier to obtain and more usable data.

Group/Student shall #5

Please remember that this homelab will be fulfilled only after the application of the simulation flow to a function of your will.

- Chose and define a function alternative to the divmul.c presented in this homelab.
  Function shall be similar but not the same of those presented in this document and/or in class.
- Last assignment is to apply the whole i8051 flow to that function.
- When done prepare a couple of slides which describes and document the work performed.
  Slides shall refer only to the that chosen function (NOT to previous group/student shall).