sf20
evaluation package

‘readme’ notes

Revision 1.0
22 December 2014

Author: Martin Raubuch

info@racors.com

Revision History

Revision	Date	
1.0	22Dec2014	Initial Revision

Table of contents

1 Introduction

1.1 Purpose

This evaluation package is intended for development engineers that want to take a look at the RACORS sf20 family of 16-bit microprocessors. It contains tools to build and test small software projects for the sf20b (base) ISA (Instruction Set Architecture).

The package also contains synthesizable RTL code of the sf20bu core, the ultra-light implementation of the sf20b ISA which may be used free of charge for commercial and non-commercial applications.

The second chapter is a tutorial that shows how to use the tools to create and test software projects.

1.2 License

This free of charge package is provided as is with no warranty or support. Licensing details are described in the **license.txt** file which is part of the package. The file is located in the same directory as this **readme** file.

1.3 Installation

The package is available for two platforms:

1. cygwin, a Linux environment for PCs with Windows OS
2. Linux, for PCs with native Linux OS

Only differences are the executables in the **`eval1/bin`** directory, all other files are identical. After unpacking the tar ball copy the executables from the **`eval1/bin`** directory to your **`usr/bin`** directory for convenient use from any shell window. Of course it's also possible to run the executables by specifying their file paths directly. In this case copying to **`usr/bin`** is not required.

1.4 Content

The following table lists all files of the package with their location and a short description.

File	location	description
`readme.pdf`	`eval1`	this readme file
`license.txt`		licensing terms
`sf20_qrg_revxx.pdf`	`eval1/docs`	sf20 quick reference guide, xx = revision
`sf20bISA_Revxx.pdf`		sf20b ISA reference manual, xx = revision
`sf20buIMA_Revxx.pdf`		sf20bu IMA (Implementation) reference manual
`sf20asmRevxx.pdf`		sf20 assembler user manual, xx = revision
`dcsliteRevxx.pdf`		dcslite simulator user manual, xx = revision
`sf20asm`	`eval1/bin`	sf20 assembler, executable
`eval1b`		sf20b ISS (Instruction Set Simulator), executable
`eval1bu`		sf20bu cycle accurate simulator, executable
`eval1bl`		sf20bl cycle accurate simulator, executable
`eval1.asm`	`eval1/projects/eval1/sf20b`	example project top-level assembly
`eval1.h`		example project header, definitions
`main.asm`		example project assembly sources
`intdiv.asm`		
`stdout.asm`		
`dcscontrol`		simulator control file
`dcscontrol`	`eval1/projects/eval1/sf20bu`	simulator control file
`dcscontrol`	`eval1/projects/eval1/sf20bl`	simulator control file
`sf20bu.v`	`eval1/rtl`	synthesizable Verilog of the sf20bu

2 Tutorial

2.1 Overview

This chapter shows how to create software projects for the sf20b architecture using the **sf20asm** standalone assembler and how to test/verify the software using the **dcs** simulator. The package contains a small example software project called **eval1**.

2.2 The dcs simulator

All RACORS processors and hardware IPs have been developed with the **dcs** modeling and simulation tool, a RACORS proprietary tool. The tool has been developed specifically for processor development and for hardware/software co-design. Dis-assembled instructions can be included in simulation listings which makes it also very useful for software development and verification.

This package contains a stripped down version of **dcs** called **dcs-lite**. It includes all features to simulate existing circuits and is primarily intended for software development and verification. The features for hardware development have been removed. A **dcs-lite** user manual can be found in the **`eval1/docs`** folder.

Hardware models for **dcs** are written in C language and linked with the simulation engine to generate simulator executables. Each unique circuit requires a separate simulator executable which contains the circuit model.

2.3 Processor models

The package contains three simulator executables in **`eval1/bin`**, one for ISS simulations of the sf20b ISA and one for each of the sf20bu and sf20bl processor implementations. The contained circuit models have the following common features:

- 64kx20-bit instruction memory mapped to **`0x0000-0xFFFF`** of the instruction address space (covers entire space)
- 63kBytes data memory mapped to **`0x0000-0xFBFF`** of the data address space, the remaining 1kBytes from **`0xFC00-0xFFFF`** are reserved for peripherals
- A **`stdout`** test bench peripheral to print text strings to the simulation shell window
- A test bench peripheral to print the current simulation cycle to the simulation shell window

The processor models are different between the executables but are fully software compatible because all three support the sf20b ISA.

sf20b ISS (Instruction Set Simulation) model

This ISS model implements the sf20b ISA but no features of a hardware implementation. It is the functional and performance reference for hardware implementations.

Functional reference means that program code that is executed on a hardware implementation (CSA model) of the sf20b ISA must generate the same results as the ISS model running this program code.

To be suitable as performance reference the ISS model's IPC (Instructions Per Cycle) is 1. All instructions are executed in one cycle except load/store instructions with multiple source/destination registers which take one cycle per register to be close to implementation realism.

ISS simulations are very fast (in the order of 10M cycles/s) and generate bit true results. They are well suited for software development and functional verification. Performance (cycle counts) reflects the IPC of 1 and cannot be used directly as performance indicator of a hardware implementation. However correlating with the average IPC of the target implementation provides a good estimate of the expected real life performance of hardware implementations.

sf20bu and sf20bl CSA (Cycle & Structure Accurate) models

These models reflect the actual hardware architecture and implementation of the respective processor. Synthesizable Verilog code of an implementation is derived from its corresponding CSA model and is verified against this model by co-simulation and cycle-by-cycle matching of all IOs, internal signals and registers.

In software development CSA simulations can be used to measure and optimize performance. Measurement is 100% accurate because the CSA models are cycle accurate. Simulation results of critical code sequences

can be used to analyze if cycles are lost e.g. due to data dependencies and if performance can be improved by instruction re-ordering or replacement.

2.4 The example project

The project is located at **`eval1/projects/eval1`**. This directory contains three sub-directories one for each of the sf20b, sf20bu and sf20bl based models and associated simulator executables in **`eval1/bin`**. The three sub-directories have different simulator control files to reference processor registers for list-vector generation and memories.

The assembly sources of the example project are located in the **`sf20b`** subdirectory. **`eval1.asm`** is the top-level. Most of the example source code is in **`main.asm`**. The program calls routines in **`stdout.asm`** to print messages to the screen. Header file **`eval1.h`** defines addresses of data objects. File **`intdiv.asm`** contains an integer divide routine which is used by the number-to-text conversion routine in **`main.asm`**. The example program performs the following actions:

- Prints “Hello World” to the screen (simulation shell window)
- Calculates the prime numbers from 1 to 1000 using the “Sieve of Eratosthenes” algorithm
- Prints the prime numbers found
- Prints “Program End”

2.5 Running simulations

Generating executable code from the assembly sources

This is done by running the assembler (executable in **`eval1/bin`**) in directory **`eval1/projects/eval1/sf20b`** with **`eval1.asm`** as argument. You may want to have a look at the generated list file **`eval1.log`**. The generated code is in **`eval1.cod`** which is a text readable file as well. When **dcs-lite** is started it loads the instruction and data memories of the simulation model with the code from **`eval1.cod`**. The **dcs-lite** control file **`dcscontrol`** contains memory load commands that references the **`eval1.cod`** file.

ISS simulation of the example program

In directory **`eval1/projects/eval1/sf20b`** run the simulator **`eval1b`** (executable in **`eval1/bin`**) with arguments **`49000 a`**. This runs the simulator for 49000 cycles and generates format **a** list vectors. The example program should print messages on the screen as described earlier. The “Program End” message is printed in cycle 48107.

List vectors are written into file **`listfile`** one line per vector/cycle. The format is defined in file **`dcscontrol`** and has the following elements from left to right:

element
Cycle number
Disassembled instruction
Instruction Address (hex)
Registers R0, R1, … RF (hex)
Special registers CC, CS, LC, U0, SA, IA, TA (hex)

In ISS simulations the effect of each instruction (updating of destination operands) happens in the cycle where the instruction is listed. Generated file **`memdump`** shows memory contents at the end of the simulation as defined in **`dcscontrol`**.

sf20bu CSA simulation of the example program

In directory **`eval1/projects/eval1/sf20bu`** run the simulator **`eval1bu`** (executable in **`eval1/bin`**) with arguments **`94000 a`**. This runs the simulator for 94000 cycles and generates format **a** list vectors. The instruction code is loaded from the **`.cod`** file in the neighboring **`sf20b`** directory to make sure that exactly the same code is executed as with the ISS simulation.

The same messages are printed as with the ISS simulation. The “Program End” message is printed in cycle 93454 which shows that the sf20bu implementation takes about twice the cycles as the theoretical ISS model to execute the same program.

The list vectors in **listfile** have almost the same format as the ISS list vectors. A minor difference is the printing of the flags of special register **CS** as a separate field in binary format following register **CC**. Otherwise the main difference is the extra marker character in front of each disassembled instruction. It indicates the execution status of the attached instruction, e.g. '*' means final execution cycle, '-' means stalled and '0','1','2', …. indicate intermediate cycles of multi-cycle instructions.

In the memory listing file **memdump** a major difference to the ISS simulation is that undefined memory locations are printed as XX characters. Memory locations are defined only if they are loaded from a **.cod** file when the simulator is started or if the processor writes a defined value to them.

sf20bl CSA simulation of the example program

In directory **eval1/projects/eval1/sf20bl** run the simulator **eval1bl** (executable in **eval1/bin**) with arguments **60000 a**. This runs the simulator for 60000 cycles and generates format **a** list vectors. As with the sf20bu simulation the instruction code is loaded from the **.cod** file in the neighboring **sf20b** directory.

The same messages are printed as with the ISS simulation. The "Program End" message is printed in cycle 58893 which shows that the sf20bl implementation is somewhat slower than the theoretical ISS model but significantly faster than the sf20bu.

The register elements of the list vectors in **listfile** have the same format as the ISS list vectors. Main difference compared to the ISS and sf20bu simulations is the printing of two disassembled instructions per vector and line. This is because the sf20bl has decoupled flow-control and computation/load/store execution units and it is possible that two instructions (one flow + one non-flow) are executed in the same cycle.

The marker characters in front of disassembled instructions indicate the execution status but have somewhat different meaning compared to the sf20bu simulation except for the '*' and '-' markers which indicate executed and stalled instructions respectively. Some extra markers are used in conjunction with branch speculation. Ignored 'i' is used as marker for conditional branch instructions that are speculatively not executed. Instructions that are stalled in their final execution stage because they have been fetched speculatively and the associated branch condition is not resolved yet have a (?) marker. When a branch condition is resolved showing that the speculation was wrong a '*' marker followed by "resume" is printed to indicate that instruction fetching and execution continues with the correct branch alternative. The execution pipeline is flushed and speculatively fetched instructions in the final execution stage are aborted which is indicated by a '!' marker.

2.6 Creating new software projects

Copy the example project directory **eval1/projects/eval1** and rename it to the desired project name. You may also want to rename the assembly source files. The top level assembly source file is used as argument of the assembler and contains "include" directives for all other source files of the project. The assembler generated **.cod** file has the same name as the top-level file, but with a **.cod** extension. It is referenced in the **dcscontrol** simulator control files (**MEMLD** commands), the file names there must be changed to the new top-level file name.