dcs-lite

digital circuit simulator lite

User Manual

Revision 1.0
27. December 2013

Author: Martin Raubuch

info@racors.com

Revision History

Revision	Date	
0.9	07Sep2012	First version derived from **dcs** user manual
1.0	27Dec2013	Review, property note changed to RACORS GmbH

Table of contents

1 Overview

1.1 Introduction

The **dcs** simulator is a tool for the development and verification of digital circuits. Focus is on high efficiency and high simulation speed to handle complex circuits and long simulation sequences. The tool has been created primarily for the development of the eco and sf families of microprocessors but can as well be used for any other digital circuits.

The **dcs** based development flow is specifically optimized for processor cores and for hardware/software co-development. The cycle based simulation mode is used to develop bit true models, e.g. ISS (Instruction Set Simulation) models of microprocessors. It provides very high simulation speed (~10Mcycles/s) and is well suited for software and programming tools development.

The event based simulation mode is used to develop cycle and structure accurate hardware models. It provides higher simulation speed than Verilog RTL simulators (some 100kcycles/s depending on circuit complexity and host performance). Structure accurate means that the circuit hierarchy and signal and register names of logic modules are defined in the **dcs** model and are later transferred one by one to a Verilog RTL model.

After verification in the **dcs** environment circuits are translated to synthesizable Verilog RTL. **Dcs** generates skeleton Verilog modules (I/Os, registers, sub-module instantiations and sub-module connections). The actual translation of primitive modules to synthesizable RTL is done manually but is an easy and quick step compared to the development and verification phase. Verilog RTL modules are verified against the corresponding **dcs** models using cycle by cycle matching of the **dcs** and Verilog simulation outputs. **Dcs** generates Verilog stimulus files, Verilog wrapper files and utilities for the matching of simulation outputs.

Circuits consist of a hierarchy of interconnected modules. Modeling is done directly in C language. Circuit models are compiled and linked with the simulator kernel to generate the simulator executable. Primitive modules have one or more functions that implement the RTL or bit true behavior.

Besides the high simulation speed and the ability to create cycle & structure accurate models one major benefit of **dcs** over other C language modeling techniques is the X-state (undefined state) handling which is extremely helpful for digital circuit debugging.

Dcs is a command line tool with a simple but efficient user interface. To run simulations using an existing circuit model, e.g. for software development or functional/performance evaluation of IP blocks is easy also for new users. Modeling and circuit development however requires in depth knowledge of digital circuits, modeling and simulation techniques and the C language.

1.2 dcs-lite

This user manual is for **dcs-lite** a stripped down version of **dcs**. The lite version is provided to simulate existing circuits. The features for circuit development have been removed. Executables are circuit specific and contain the simulation engine and the circuit model.

1.3 Feature summery

- Command line tool for digital circuit simulation with high speed
- Cycle based, 2-state (0,1) simulation mode for bit true models
- Event based, 3-state (0,1,X) simulation mode for cycle accurate models
- Generation of simulation listings with disassembled processor instructions
- Generation of memory dumps
- Command to load memory models from eco and sf assembler output (.cod files)

2 User Interface

2.1 Concept

Dcs-lite is not an interactive tool. Each run simulates the specified number of cycles and then the tool exits. Two output files with simulation results are generated. Per cycle list vectors are written into the file 'listfile'. At the end of the simulation user specified sections of memory content are written into file 'memdump'.

The number of cycles to simulate and the cycles for which to generate list vectors is specified as command line arguments when the tool is invoked. All other control parameters and commands (simulation mode, list vector definition, memory loading from files, etc.) are read from the file 'dcscontrol' in the local directory.

Because circuit models are C files that are compiled and linked with the simulation kernel **dcs-lite** executables are circuit specific and have circuit specific names. Circuit models are different between the 2-state cycle based and the 3-state event based simulation modes and separate executables are generated for the two modes.

2.2 Invocation

Dcs-lite is started by typing the name of the executable in the command shell followed by an optional sequence of simulation cycle numbers and list vector specifiers as arguments. If no arguments are given the default number of simulation cycles is 1000.

Arguments are integer numbers that specify the number of cycles to simulate. The maximum number is the maximum value of an unsigned 32-bit integer. Each cycle number can be optionally followed by a list-vector specifier (letter 'a', 'b', 'c', or 'd') to indicate that **dcs-lite** should generate list vectors for the specified number of simulation cycles. The examples below illustrate the concept.

Example 1

```
executable_name
```

Simulates 1000 cycles with no list vector generation, (default, if no **run** command is in the control file)

Example 2

```
executable_name 500000
```

Simulates 500.000 cycles with no list vector generation

Example 3

```
executable_name 34621 b
```

Simulates 34.621 cycles with format **b** list vector generation

Example 4

```
executable_name 65000 1520 d
```

First simulates 65.000 cycles with no list vector generation, then simulates 1.520 cycles with format **d** list vector generation

Example 5

```
executable_name 100000 4784 c 23000 2500 a
```

First simulates 100.000 cycles with no list vector generation, then simulates 4.784 cycles with format **c** list vector generation, then simulates 23.000 cycles with no list vector generation, finally simulates 2.500 cycles with format **a** list vector generation.

Example 4 is the most commonly used format for hardware/software debugging and for IP block evaluation. The first phase generates no list vectors for the best possible simulation speed. The second phase is the interesting section and list vectors are generated. The generation of list vectors and writing into 'listfile' significantly slows down the simulation speed. When running high cycle count simulations it is a good idea to generate list vectors only for interesting sections of the simulation time window.

2.3 Control File

Format

Dcs-lite expects a control file with the name 'dcscontrol' in the local directory. The file has a simple, line based format. Lines that start with a blank, ';' or '#' character are treated as comment lines and are ignored. All other lines must start with a command keyword. The following paragraphs describe the syntactical elements of control file commands.

- **Numbers**: unsigned integers, can be specified in decimal format (sequence of the letters 0-9), or in hexadecimal format (prefix 0x followed by a sequence of the letters 0-9,a-f,A-F). Numbers specified in decimal format are converted to 32-bit integers and the maximum value is 2**32 - 1. Numbers specified in hexadecimal format are converted to 64-bit integers.
- **Strings**: sequences of any printable characters and blanks (except for the double quote “” character) enclosed in double quote “” characters, CR (carriage return) control characters can be specified with the \n 2-letter sequence, maximum length is 256 characters.
- **Keywords**: Keywords are not case sensitive. The following is a list of all keywords in alphabetical order: **a**, **all**, **b**, **bin**, **byte**, **c**, **cycle**, **d**, **data**, **ev3st**, **hex**, **input**, **item**, **long**, **lva**, **lvb**, **lvc**, **lvd**, **memdmp**, **memld**, **nospace**, **off**, **on**, **out**, **output**, **register**, **run**, **scmsg**, **short**, **space**, **text**, **usrstring**, **variable**

Module instances of the simulated circuit and their inputs, outputs, registers and variables are referenced by strings. For instances the path within the circuit hierarchy is specified in the same way as file paths. For example if the top level of the circuit contains an instance named CPU which contains a sub-instance CACHE which contains a sub-instance CTR then the instance CTR would be referenced by the string “/CPU/CACHE/CTR”.

Command Set

The following paragraphs are detailed descriptions of the **dcs-lite** control file commands in alphabetical order. Command descriptions start with the command syntax in the following format:

Command-key <mandatory argument(s)> [optional argument(s)]

For each argument the syntax type is specified first followed by the name of the argument in bold letters. Elements with multiple options are separated by ‘|’ (logical or) character.

lva, **lvb**, **lvc**, **lvd**

These four commands are used to define the list vector formats **a**, **b**, **c**, and **d** respectively. The commands and their effect on the respective list vector formats is identical for list vectors **a**, **b**, **c** and **d**. Each command adds an item to the respective list vector format. The second keyword determines the type of item added and the syntax of the parameters that follow. The next paragraphs describe the **lva** command options and their effect on the list vector format **a** as example. More details can be found in the separate **list vectors** section of this chapter.

lva output <keyword **hex**|**bin**|**vsp**> <keyword **space**|**nospace**> <string **instance**> <string **output-signal**>

Adds the specified **output-signal** of the specified **instance** to list vector format **a**. If the **instance** path or **output-signal** doesn’t exist an error message is printed.

The next keyword following **output** specified the print format.The options are **hex** (hexadecimal) or **bin** (binary). With the **hex** and **bin** formats the **output-signal** is printed with the minimum number of digits to represent the bit width of **output-signal**. The mandatory **space** or **nospace** options determine if a leading space character is printed before the **output-signal** is printed.

lva input <keyword **hex**|**bin**|**vsp**> <keyword **space**|**nospace**> <string **instance**> <string **input**>

Adds the specified **input** of the specified **instance** to list vector format **a**. If the **instance** path or **input** doesn’t exist an error message is printed. The remaining elements of the command are the same as for the **lva output** command.

lva register <keyword **hex**|**bin**|**vsp**> <keyword **space**|**nospace**> <string **instance**> <string **register**>

Adds the specified **register** of the specified **instance** to list vector format **a**. If the **instance** path or **register** doesn’t exist an error message is printed. The remaining elements of the command are the same as for the **lva output** command.

lva variable <keyword **hex**|**bin**|**vsp**> <keyword **space**|**nospace**> <string **instance**> <string **variable**>

Adds the specified **variable** of the specified **instance** to list vector format **a**. If the **instance** path or **variable** doesn’t exist an error message is printed. The remaining elements of the command are the same as for the **lva output** command.

lva text <string **text**>

Adds the specified **text** string to list vector format **a**. Strings can contain '\n' characters to define multi line list vector formats or to add a blank line as separator.

lva userstring <number **n**> <keyword **space**|**nospace**> <string **instance**>

Adds the **userstring** of the specified instance to list vector format **a**. Parameter **n** specifies the length of the printed string. Let the length of the module defined user string be **k**. If **k**>**n** the first **n** characters of the user string are printed. If **k**<**n** then the extra characters are printed as blanks. The **space** or **nospace** parameter determines if a leading space character is printed before the user string.

memld <keyword**text|data**> <string **instance**> <number **offset**> <string **file**>

The workspace of the specified **instance** is loaded with data from the specified **file**. **Dcs** modules can have a workspace which is typically used for memory models or for large register files.

With the **memld** command memory models can be loaded from eco and sf assembler output files. In the event based 3-state simulation mode, memory locations that are loaded from a .cod file are set to the defined state.

The .cod files that are generated by the eco and sf standalone assemblers contain **text** and **data** sections. The **text** or **data** parameter of the **memld** command specifies from which section the memory model is loaded.

The specified **offset** is a 32-bit integer that is added to the address of **text** or **data** records from the .cod file. For negative offsets a 2's complement 32-bit number, e.g. 0xFFFFFF80 for -128 must be specified.

Errors are printed if the **instance** does not exist, if the **instance** has no workspace or if an address (.cod record address + **offset**) is out of range with respect to the **instance**'s workspace.

memdmp <string **instance**> <number **addr**> <number **items**> <keyword **byte**|**item**>
<keyword **byte**|**short**|**long**> <number **items_per_line**> <string **name**>

An entry with the specified parameters is added to the memory dump list. At the end of a simulation **dcs** generates the output file 'memdump' with sections of memory content listings. Each **memdmp** command defines one section that is written into 'memdump'.

The **instance** (1st parameter) defines the instance path of the memory. An error is generated if the **instance** does not exist or if the **instance** has no workspace (is not a memory).

The item type (5th parameter) has the options **byte**, **short** or **long**. With the **byte** option, 8-bit values are printed with 2 hexadecimal digits per value. With the **short** option, 16-bit values are printed with 4 hexadecimal digits per value.With the **long** option, 32-bit values are printed with 8 hexadecimal digits per value. Printed values are separated with one space character.

The address format parameter (4th parameter) has the options **byte** or **item**. With the **byte** option the start address **addr** (2nd parameter) is interpreted as a **byte** address and the addresses printed at the beginning of each line of the memory content listing are **byte** addresses. With the **item** option the start address **addr** (2nd parameter) is interpreted as an **item** address and the addresses printed at the beginning of each line of the memory content listing are **item** addresses. Item address means that if for example the item type (5th parameter) is **long** the address n references the nth 32-bit word of the memory.

The **addr** value (2nd parameter) specifies the start address of the memory content that is listed. An error message is printed if the address range that is specified by **addr** and **items** does not lie completely inside the **instance** workspace.

The **items** value (3rd parameter) specifies the number of items to be listed. An error message is printed if the address range that is specified by **addr** and **items** does not lie completely inside the **instance** workspace.

The **items_per_line value** (6th parameter) defines how many items are printed per line.

The **name** string (7th parameter) specifies a name for the section. This name is printed in a separate line before the listing starts. Purpose is to make it easier to find sections in the 'memdump' file.

run <number **cycles**> [keyword **a**|**b**|**c**|**d**|**v**]

Specifies the number of **cycles** to simulate and an optional list vector format. The list vector format can be **a**, **b**, **c**, **d**, or **v**. If no list vector format is specified no list vectors are generated.

Multiple **run** commands can be present in the command file. The simulator executes them in the order of their appearance. With multiple **run** commands it is possible to define time windows with and without list vector generation similar as it can be done by specifying command line arguments when **dcs** is invoked.

scmsg <keyword **on**|**off**>

Switches screen messaging **on** or **off**. This is for messages that are generated by circuit modules, typically by test bench modules. **Dcs** provides a global variable (must be declared external in message generating modules) for module implementations to check if screen messaging is switched on or off. Purpose is to switch screen message generation off when not needed to increase simulation speed.

simod <keyword **cycle**|**ev3st**>

Sets the simulation mode to **cycle** (cycle based, 2-state 0 and 1) or **ev3st** (event based, 3-state, 0,1,X). The selected mode must correspond with the circuit model. Selecting the wrong mode generates meaningless simulation results.

2.4 List Vectors

The purpose of list vectors is to visualize simulation results of longer time periods.During periods where list vector generation is enabled **dcs-lite** writes one list vector per simulation cycle into the file 'listfile' in the local directory. The generated file is an ASCII text file and can be viewed with any text editor.

Four list vector formats **a**, **b**, **c** and **d** can be defined. All list vector formats start with the simulation cycle number. The cycle number is printed as a 10-digit decimal number followed by two space characters as separator. The cycle item is part of all list vector formats and does not need to be specified by an **lva**, **lvb**, **lvc** or **lvd** command. The remaining items of user formats **a**, **b**, **c**, and **d** are defined by a series of **lva**, **lvb**, **lvc** and **lvd** commands in the **dcs-lite** control file. Each command adds one item to the respective list vector format. The following item types are supported:

- **inputs**, **outputs**, **registers**, **variables**: these items refer to the state of module instances of the simulated circuit. The **input**, **output**, **register** or **variable** state of the specified module instance is printed in hexadecimal or binary format. Values are printed with the minimum number of digits required to represent the bit width of the specified item. A leading optional space character can be printed as separator. In the **ev3st** (event based 3-state) simulation mode undefined bits are printed as 'X' characters. For values that are printed in hexadecimal format an 'X' is printed if any of the bits represented by a digit is undefined.
- **text**: with this item type user defined text strings can be included. Main use cases are separators (multiple space characters) and names of instance state items. Text strings can include '\n' characters to create multi line list vector formats or to separate multi line vectors with a blank line.
- **userstring**: **dcs-lite** module definitions can include a programmable length user string. For modules with a user string the simulation data structure of module instances provides a pointer for module implementation functions to write to the user string. Main use case is to include disassembled processor instructions in list vectors.

The reason for having multiple user defined list vector formats is to avoid the need for changing a single format every time different signals should be listed, e.g. during circuit debugging. The fact that list vectors are helpful for both hardware and software debugging is another good reason for having multiple user defined formats.

For software debugging typical list vector formats contain the disassembled instruction and the processor registers of the programming model. The ability to include disassembled instructions is one of the most powerful features of **dcs-lite**. In most cases the cycle based, 2-state simulation mode is used for software and programming tools development. It has the advantage of very high simulation speed. Runs with billions of cycles complete in short times.

The event based 3-state simulation mode can also be useful for software and programming tools development, e.g. in the following cases:

- **Accurate performance evaluation**. With cycle and structure accurate circuit models simulation results 100% reflect the cycle by cycle behavior and performance of the hardware circuit. List vectors show processor stalls caused by either hardware (bus wait states) or data dependencies. Especially if circuits include peripherals and memories **ev3st** mode simulations can be used to analyze performance bottlenecks and optimize software, e.g. by instruction reordering or alignment of loop entry points.
- **Debugging**. In **ev3st** mode simulations memory locations that are not preloaded with a **memld** command in the control file are initially undefined. If software reads and uses an undefined variable the entire processor state goes to undefined after a few clock cycles. This makes it easy to find software bugs that are related to uninitialized variables.

2.5 Memory Dump

File ‘memdump’ in the local directory is generated at the end of a simulation run and lists the content of memories of the simulated circuit. The file contains multiple sections each section is defined by a **memdmp** command in the control file.

Sections start with a separator line filled with ‘*’ characters and the memory’s instance name in the middle. The next line prints the section’s name which is defined by the associated **memdmp** command (last parameter). The following lines are the actual memory content listing. They start with an address followed by a two-character separator followed by a user defined number of memory content values, separated by space characters.

Values are printed as 2 (byte), 4 (short) or 8 (long) digit hexadecimal numbers with no prefix. In the **ev3st** simulation mode digits are printed as ‘X’ (undefined) if any of the four bits represented by a digit is undefined. In the **cycle** simulation mode memory content is always defined and is initialized to all zeros at the beginning of the simulation.

The address at the beginning of each line can be a byte address or an item address as defined by the associated **memdmp** command. Item addresses point at the n^{th} item (byte, short or word) of the memory.