Dlxsim

Dlxsim [DLX-Package] is an instruction accurate simulator for DLX assembly code. In this laboratory, we will use a modified version of dlxsim, which is changed in such a way, that it is behaving like the ASIP Meister specific implementation of the Brownie STD 32 Processor, which will be created and used in the later steps of the laboratory. In the first subchapter, some basic ideas about the Brownie architecture and the Brownie instruction set will be introduced. Afterwards the basic usage of dlxsim will be explained. In the last subchapter, it is shown how dlxsim can be extended to support new assembly instructions, which will be added to the Brownie processor with ASIP Meister.

Brownie STD 32 Architecture

Brownie STD 32 (browstd32.pdb) is a RISC-type pipeline processor architecture. The Brownie architecture is designed for an easy and fast **pipeline processor**. It is a **Load-/Store-architecture**, which means that there are dedicated commands for accessing the memory and that all the other commands only work on registers, but not on memory addresses. As an implication, the Brownie architecture has a **big uniform register file** that consists of 32 registers with 32 bits each, where some registers are special registers like the register r0 has a special meaning, as it is hard wired to zero. The pipeline stages for the Brownie processor are the following:

1. Instruction Fetch (**IF**): This phase reads the command on which the program counter (PC) points from the instruction memory into the instruction register (IR) and increases the PC.
2. Instruction Decode (**ID**): Here the instruction format is determined and the respectively needed parameters are prepared; e.g. reading a register from the register file or sign extending an immediate value.
3. Execute (**EXE**): The specific operation is executed in this phase for the parameters, which have been prepared in the preceding stage.
4. Memory Access and Write Back (**WB**): If the command is a memory access, then the access will be executed in this phase. Every non-memory access command will pass this stage without any activity. Finally, the result that has been computed or loaded will be written back to the register file.

   The memory architecture of Brownie STD is Harvard. Both of the instruction length and the data length are 32 bit. Addressing can be performed in byte. Brownie has 32 integer general-purpose registers and a 4-stage pipeline structure, each stage of which is named IF, ID, EXE and WB. Full forwarding to the pipeline makes the operation results of all the instructions immediately usable. Brownie STD executes delayed load for load type instruction, and the number of delayed slot is one. Brownie STD does not have a delayed branch slot (does not execute delayed branch). Brownie

STD does not have a floating-unit operation. A floating-point execution instruction can be added as a custom instruction. Basic architecture parameters are shown in the following table.

In Figure 3-2, an example for some overlapping commands in a 5-stage pipeline (DLX processor) is shown. In the first command, the values from the registers *r2* and *r3* are added and the result is stored in register *r1*. The write back to *r1* is done in clock cycle 4, so it cannot be read earlier than in clock cycle 5. The last command of the shown pipeline example is using *r1* as input and it is scheduled in such a way, that it is reading *r1* in clock cycle 5, so that it is using the latest value that has just been written back by the first command. *The example shows, that three successive NOPs are enough to resolve a data dependency.* The Brownie processor is using a **data forwarding** technique to resolve such data dependencies without the in-between NOPs, and ASIP Meister does support full forwarding i.e. data and branch forwarding is available in this version of ASIPmeister. Therefore, for ASIP Meister generated processors, the NOPs are not needed to resolve the data dependencies (as shown in Chapter 3.2.1 you can configure dlxsim to behave in both ways, i.e. with or without data forwarding).

Parameters	**Architecture**
Basic architecture	RISC
Memory architecture	Harvard
Instruction length	32
Data length	32
Addressing	Byte address
The number of general-purpose registers	32
The number of pipeline stages	4
The number of delayed branch slot	0
Floating-point unit	N/A
Forwarding	full forwarding
Alignment	1-byte, 2-byte and 4-byte
Endian-ness	Big Endian
Interrupts	Reset, Internal, External

Figure 3-1: Brownie STD 32 Architecture Parameters

The **instruction set** of the BROWSTD32 architecture is separated into four **instruction classes** (arithmetic for integer, arithmetic for float, load/store and branch), which are implemented in different **instruction formats**, as shown in Figure 3-3. The arithmetic instructions use either an instruction format for three registers or an instruction format for two registers and an immediate value. The load/store instructions always use the format with two registers and one immediate, where the effective address is computed as the

sum of one register (as base address) and the immediate value. The other register is used either as value to store in memory or as register where to place the loaded value. The branch instructions are divided into conditional branches and unconditional jumps. The jumps use an instruction format with a 26-bit immediate value as a PC-relative jump-target. The *jump register* instructions instead use the I-Format to declare in which register the absolute jump target is placed. The second register and the immediate value are not used by these instructions. The conditional branches need a field for a register that contains the condition, so they use the I-Format as well. They use the 16-bit immediate as PC relative jump target. The second available register is not used.

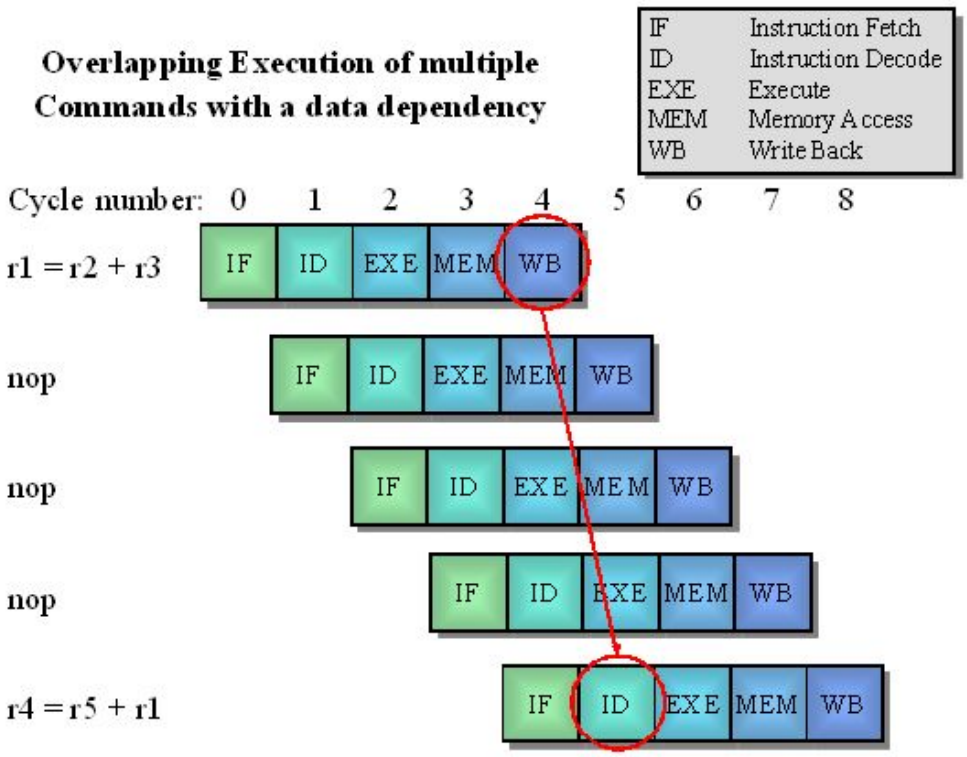

Figure 3-2: BROWSTD32 Pipeline Example with a Data Dependency

For many assembly-commands, there are special versions for dealing with unsigned values and for using immediate values as second input parameters. These versions have an attached "i" for "immediate" and/or an attached "u" for "unsigned" as suffix (e.g. addui). A summary of all assembly instructions that are available in the ASIP Meister specific implementation of the BROWSTD32 processor that is used in the laboratory (i.e. browstd32) is shown in Figure 3-4:. For a more detailed description of the assembly-commands have a look into [Sailer96].

Finally, some specialties in the architecture need to be mentioned:

- **Delay slots:** Without forwarding, an instruction, that is placed right after an unconditional jump or a conditional branch instruction is always executed. In fact, there are two instructions, that enter the pipeline, but only the first one is executed, the second one will not be allowed to write the computed result back to the register file. But, in Brownie this is handled automatically using full-forwarding.
- **Multicycle operations:** The operations mult, div and mod in their different versions are multicycle operations. That means, that these operations will not stay for one cycle in the execute phase, but for multiple cycles. The pipeline is stalled until the instructions finished their work in the execute stage.
- **Stalling:** The communication to the data memory is controlled by Request/Acknowledge-Signals to deal with memories of different speed. The corresponding assembly instructions might stay for multiple cycles in the memory stages, until the memory access is finished.

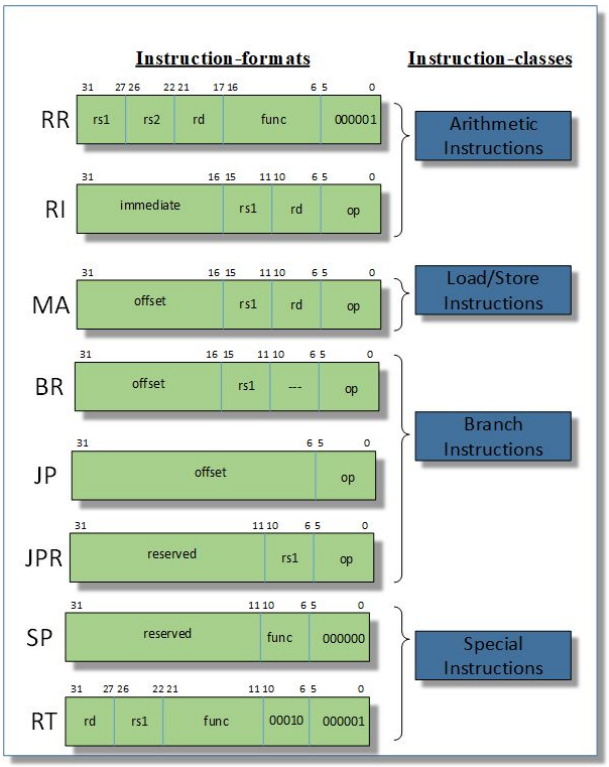

Figure 3-3: Instruction Formats and Classes of the BROWSTD32 Architecture

- **Special Registers:** Some registers in the general-purpose register file usually handle some special cases. However, contrary to the hard-wired zero-register *r0*, any register, as long it is done consistent in the assembly

file, can handle these other special cases. These special cases are the stack pointer (GPR*1*), the frame pointer (GPR4) and the link register (*GPR3*), which is then used as return address. For more details, please look into the Page-9 of Brownie Reference Manual [].

- **Instructions:** for more details of the instruction and their syntax refer to Page-44 [RM]

Instruction	**Description**
ADD, ADDI	Add; Syntax: *add rd rs0 rs1; rs1* can alternatively be the immediate
MUL	Multiply
MOD, MODU	Modulo
OR, ORI	Or
LLS, LLSI	Shift left logical
ARS, ARSI	Shift right arithmetic
LSOI	Left shift & OR immediate value
ENEQ	Set "not equal"
LB	Load Byte
LW	Load Word; Syntax: *load rd, imm(rs);* the effective address is computed by adding the im
SH	Store High
BRZ	Branch if "equal zero" Syntax: branch rs, imm; the branch target is a PC-relative address, given by the immediate
JP (JPR)	Jump (Register); Syntax*: j imm (jr rs);* The jump target is a PC-relative (absolute) addr
NOP	
EXBW	8-bit to 32-bit sign extension

Figure 3-4: Summary of All Assembly Instructions in the *browstd32* Processor

Extending dlxsim

Startup Parameters for dlxsim

Some parameters can be adapted when starting dlxsim. All mentioned parameters in Figure 3-5: do not allow blanks between a parameter, e.g. "*-pf0*" instead of "*-pf 0*".

Option	**Description**
-f*filename*	This parameter will load an assembly file after initialization. This par
-sf*filename*	This parameter loads a script file that can contain any command that

-dbb# (Debug Base Blocks)	This option only has an effect, if the later mentioned option "*Debug Assembly*" is enabled too. If both options are activated, then a register snapshot will automatically be printed at every base block start. This snapshot only includes registers, for which the value has changed since the last snapshot. By default, this option is turned off to avoid the enormous amount of output. To turn it on you have to enter *"-dbb1"*.
-da# (Debug Assembly)	This option helps you debugging the simulated assembly code. A debugging output is printed for all load/store and jump/branch instructions, including in which cycle the message was printed and from which address it was triggered. By default, this option is turned on. To turn it off you have to enter *"-da0"*.
-cdd# (Check Data Dependency)	With this option, you enable a warning message that appears when an unresolved data dependency is found in the executed assembly code. This means you will get a warning if for example, *r5* is read in cycle 10, but an earlier command has made a write access to *r5*, that cannot be read back before cycle 12. Therefore, you would read the 'old' value. By default, this option is turned on. To turn it off you have to enter "*-cdd0*".
-wsdo# (Warn Specific Dependency Once)	This option belongs to the before mentioned option Check Data Dependency. If there is an unresolved data dependency in a loop, then the warning message would usually be printed for every time the loop is executed. With this option defined, the warning will only be printed the first time the unresolved data dependency is noticed. By default, this option is turned on. To turn it off you have to enter *"-wsdo0"*.

-pf# (Pipeline Forwarding)	With this option, you can configure, whether you have a "Full Forwarding" (1) or not (0). In case of forwarding, your operands are forwarded to next stages and no branch delay slot is required. But, even in case of forwarding, you still require a delay slot (nop) for loar/store.
-ms# (Memory Size)	The size of the memory that is available within dlxsim. It is the common memory for instructions and data.
-lf*filename*	With this parameter, all print instructions for the LCD (as shown in C
-uf*filename*	With this parameter, all print instructions for the UART (as shown in
-af*filename*	With this parameter, all outputs to the AudioOut IP-Core (as shown

Figure 3-5: Summary of Helpful dlxsim Starting Parameters

Legend: *cursive*: replace with appropriate option, #: replace with a number

How to Add a New Instruction

The following list shows you the needed steps to add a new instruction for a given instruction-format. The given line numbers are rounded and might change during time, as the source code is partially under development. Every needed part of the source code is marked with a comment that includes the string "*ASIP NEW_INSTRUCTIONS*".

1. *dlx.h*:222 Add a new define for **OP_{CommandName}** with a unique number.
2. *sim.c*:227 Add the **name** for your assembly-command into the **operationNames**-array. The position of the name in this array has to correspond with the defined number in *dlx.h* from the preceding step.
3. *asm.c*:338 Add a new entry with your instruction name, instruction format and opcode into the ***opcodes***-table "OpcodeInfo opcodes[]". The **instruction name** has to be the same like in the previous step and completely written in lower-case! The already available **instruction formats** are shown in Figure 3-6:. There are two different possibilities for the **opcode**. Either you use the only "opcode" or you use "opcode" and "func" field collectively. The unused bits always stay 0 in the opcode field. These two possibilities differ in how dlxsim is internally handling the instruction. This will become clearer in the following step. Have a look at the following step to find free opcodes. The following values in the opcode table are usually not that important and might be filled out by copy-and-paste from

a similar instruction. Only the flags are important if you use instructions with immediate values as parameter. The flags are explained in *asm.c*.

4. *sim.c*:362 This step depends on your choice of the preceding step. You have used either the "opcode" or the "opcode" plus "func" field. If you have only "opcode" field, then you have to modify the ***opTable*** in *sim.c* otherwise in case of "opcode and func" field, then you have to modify the ***specialTable***. In both cases you replace the table entry at the position that corresponds to the chosen 6-bit opcode with your own ***OP_{CommandName}***. So if you have chosen the opcode value 5, then you replace the 5^{th} entry (start counting with 0) in the array with your own command.

5. *sim.c*:2197 Implement a new **case** for the big "***switch (wordPtropcode)***"-statement for your command. A good start is a copy-and-paste from a similar instruction. There are 2 different variables for the parameters. For example, there is a "***wordPtrrs1***" and a "***rs1***". In "***wordPtrrs1***" the number of the first source register is stored, while in "***rs1***" the value of this source register is stored. At the beginning of an implementation of one specific "***case***" some macros like "***LoadRegisterS1***" are called. These macros take care, that "***rs1***" is initialized with the current value of the register with the number "***wordPtrrs1***".

6. Compiling: To test your modified version of dlxsim you have to re-compile it. Simply type "make" in the dlxsim directory.

When the error "*Unknown Opcode*" occurs while loading the assembly file, then the corresponding instructions was not accepted. If you are sure, that it is not a typing error in the assembly file (register names, immediate values, ...) then have a look into points 1 – 4.

How to Add a New Instruction-Format

Adding a new instruction format is much more difficult than adding a new instruction for a given format. To add a new format you have to take care about how the parameters for your new format are to be stored in a 32-bit instruction word and how they are extracted out of it. Every needed part of the source code is marked with a comment that includes the string "ASIP NEW_FORMAT".

1. *asm.c:*140 Define a unique number for your instruction-format

2. *asm.c:*150 Add the *min* and *max* numbers of arguments, that your instruction format accepts in the arrays "**minArgs**" and "**maxArgs**". Use the position in the array, that corresponds to the defined number in the previous step.

3. *asm.c:*765 Implement how the parameters will be stored in the 32-bit instruction word within the "**switch (insPtr->class)**"-statement.

4. *asm.c*:1080 Implement what will be written if the instruction is disassembled in the "**switch (opPtr->class)**"-statement.

5. *sim.c:*2215 In the method "**compile**" you have to implement how the 32-bit instruction word will be expanded.

6. Compiling To test your modified version of dlxsim you have to re-compile it. Simply type "make" in the dlxsim directory.

Instruction Format	**Parameters**
NO_ARGS	no operands
LOAD	(register, address)
STORE	(address, register)
LUI	(dest, 16-bit expression)
ARITH_2PARAM	(dest, src) OR (dest, 16-bit immediate) OR "dest" replaced by "dest/src1"
ARITH_3PARAM	(dest, src1, sr2c) OR (dest/src1, src2) OR (dest, src1, 16-bit immediate) OR (dest/sr
ARITH_4PARAM	(rd, rs1, rs2, rs3) OR (rd, rs1, rs2, 5-bit immediate)
BRANCH_0_OP	(label) the source register is implied
BRANCH_1_OP	(src1, label)
BRANCH_2_OP	(src1, src2, label)
JUMP	(label) OR (src1)
SRC1	(src1)
LABEL	(label)
MOVE	(dest,src1)

Figure 3-6: Summary of Available dlxsim Instruction Formats

Using dlxsim

Dlxsim is distributed as source code, so before using it you have to compile it. Usually this is done, by just typing "*make"* in the "*tcl"* subdirectory and afterwards in the "*dlxsim"* directory. Then you can start the program by typing "dlxsim". If you want to use dlxsim for an ASIP Meister Project, then you have to set "*DLXSIM_DIR*" (see Figure 2-5) to the path where dlxsim is located e.g. "*/home/asip00/epp/dlxsim_Laboratory*", and then typing "*make dlxsim*" in an application's subdirectory.

Figure 3-7: shows some typical dlxsim commands. The list is not exhaustive, but it covers all the usual suspects.

Command	**Description**
load *filename*$^{+}$	Load a file, parse its content, and place the translated content to the s

get *address* {*options*}	Examples for "*address*" are *r0*... *r31*, *pc*, *npc* (next pc), 10 (memory address 10), 0x10 (memory address 16), _main (if _main is a label). Examples for "*options*" are: u (unsigned), d (decimal), x (hexadecimal, default), B (binary), i (instruction), v (do not read a value, but print the address itself; as example this can be used to translate from decimal to binary), s (interpret the upcoming sequence as 0-terminated string). In the options, you can also request to get the succeeding addresses from the determined base address. As an example, "*get _loop 10i*" would deliver the 10 first commands from the label *_loop* interpreted as instruction memory. This also works, if the address is a register, e.g. "*get r1 5u*".
put *address data*	Place some data at the given address. The address might be a registe
step {*number*}	Execute the next number of instructions given by "*number*", the defa
go {*address*}	Execute the assembly program, until an error occurs or a trap instruc
stats {*options*}	Print some statistics about the simulated assembly program. The *opt*
quit exit	Terminates the program
stop {*options*}	Manages break points: "*stop at address {command}*": Executes "*command*" whenever the given "*address*" is touched in any way. The default "*command*" is to abort the execution. "*stop info*": prints the list of break points. "*stop delete number*$^+$": deletes some specific break points. The numbers are according to the "*stop info*" list.
asm "*command*" {*pc*}	Return the opcode for "*command*". If the command needs to know th

Figure 3-7: Summary of Typical dlxsim Commands

Legend: *cursive*: replace with appropriate option, {braces}: optional, $^+$: one or multiple times

Some more points that have to be mentioned about dlxsim are:

- You can use the cursor buttons to navigate in your command history, i.e. in the previously entered commands. The up- and down-arrow-keys let you navigate inside the selected command, e.g. to correct typing errors.
- When you just enter an empty command in dlxsim (i.e. just press enter without having entered a command), then the last executed command will be repeated. This is for example useful, when you want to step through your code. You just have to execute the step instruction one time manually by typing "*step*" in the shell and afterwards you can repeatedly press enter to execute the next step instructions.
- When you press the Tab key, you will get an auto completion for filenames. The offered files are the files in the directory from where you started dlxsim. Although this auto completion will support the abbreviation "~" for your home directory, a load instruction with this abbreviation will fail. The same holds for loading files with the "*-f*" starting parameter, as shown in Figure 3-7:.
- After you have simulated an assembly code, you have to restart dlxsim to simulate another assembly code. The "*load*" instruction will not reset everything to its default value.
- Every assembly command that accepts an immediate value as parameter can also handle a label as immediate value. This is especially useful for the load/store, branch/jump and lhi commands.
- The ASIP Meister unit for data memory access does not accept an immediate change from a load command to a store command or vice versa. Dlxsim can handle this situation, but will print a warning to indicate, that this assembly code might produce a different result if it is simulated with the VHDL-code from an ASIP Meister CPU.

Statistics

There are many statistics available for the executed assembly code. You can get the statistics by typing "stats {*options*}", as shown in Figure 3-7:. The different available options are shown in Figure 3-8:. The different options may be combined; the default option is "*all*".

Debugging with dlxsim

This chapter assumes, that you are not only used to the assembler code and dlxsim, but that you are also used to the compiler and inline assembly (see Chapter 8).

General Points:

- **Compare the results** from the GCC compiled version and the gcc-compiled version. For the gcc compiled version you have to add printf statements for all essential variables, like:

#ifndef GCC

printf("temp1: %i\n", temp1);

#endif

For the GCC-compiled version, you have to make the important variables global, for example moving the variable "*temp1*" from inside the main method to a global part outside the main method. All global variables will get an own label in the assembly code with an underscore before the name, e.g. the variable "*temp1*" will get the label "*_temp1*". In dlxsim you can see the value of global variables with the "*get*" instruction, e.g. "*get _temp1 i*".

Option	**Description**
hw	Shows the memory size.
stalls	Shows the pipeline stalls (i.e. number of elapsed cycles, where the pip
all	Shows all statistics in the same order as they appear in this table.
reset	Resets all statistics to their initial values. This is useful if you want to
imcount imcount2 imcount3	Shows the number of executions for memory addresses. The output is organized into three columns. The first column shows how often a specific part of the instruction memory was executed. The second column shows the starting address of the specific memory part and the third column shows the size of the memory part. The different spellings of imcount (e.g. imcount2) refer to different sorting for the columns. This statistic merges neighbored memory addresses that are executed for nearly the same number into a single memory portion for which one output line is printed. The deviation to the average value is printed in the first column (e.g. "# of executions: 2 ± 1").
baseblocks	Shows the separation from the program into base blocks. This statisti

Figure 3-8: Summary of Available dlxsim Statistics

- Sometimes the **dlxsim simulation aborts** with an error message, e.g. when a load instruction is trying to access an address that is outside the simulated memory range. In such cases, you first have to find out, which instruction is causing this crash. With the instruction "*get pc*" you can see the address which is currently executed. With the instruction "*get {address} i*" you can see which instruction is placed at this address and at which label this instruction can be found. With the instruction "*get {Address}-0x10 20i*" you can see the context of this instruction.

- Getting more debugging information from dlxsim is very helpful to understand, what the assembly code is doing. Therefore, the dlxsim starting parameters *"-da#"* and *"-dbb#"* are useful. You have to replace the "#" with either a "1" to turn the option on or with a "0" to turn it off.

- da#: Debug Assembly. This option is turned *on* by default and it will print status information on the screen for every jump/branch/load/store-instruction. You can print additional status information by adding your needed information into the "*sim.c*" of dlxsim.

- dbb#: Debug Base Blocks. When you turn *on* this option then at every start of a base-block all changed register values will be printed. A base block is an elementary block of assembly code that is only executed sequentially. This means, that either all instructions of a base block are executed (one after the other) or none of them is executed at all. The borders of a base block (beginning/end) are jumps and labels. The simulation will create a huge amount of output on the screen if you turn this option on. Therefore it is recommended, that you copy the output to a text file for easier reading. You can automatically print everything into a text file if you start dlxsim like:
  "make dlxsim DLXSIM_PARAM="-fassembly.dlxsim -dbb1" | tee output.txt"
  The "*tee*" program will copy all output to the screen and to the file.

- Always have a look at the printed warnings when dlxsim runs the simulation. At the end of every simulation the warnings are summarized, i.e. the number of the printed warnings is shown. If the simulations aborts before its usual end this summary is not printed. To see the summary you can see them in the statistics with *"stats warnings"*.