TECHNICAL NOTE	ONDEMAND MICROELECTRONICS
Project:	SVENm
Project No:	n.a.
Author:	L. Kervella
Document ID:	Svn_note036
Last modified:	2007-07-19, 5:14 PM
Version:	1.0
File: \\Odaxsun08\prj_svenm\pdoc\note\svm_note36_v1r1x.doc	

CHILI Instruction Set Simulator short manual

1 Introduction

This document describes how to run the CHILI Instruction Set Simulator.

Note : The simulator executable is dynamically linked and requires that libstdc++.so.5 is installed. (gcc 3.3.5 or similar).

2 Revisions

Version	Date	Author	Description
1.0	2007-03-09	L. Kervella	Initial version

1. Compiling for the Simulator

The chilli gcc is used to produce an ELF binary for the simulator:

```
% chilli-odm-elf-gcc test.c –o test.exe
```

2. Starting the Simulator

The simulator for CHILI (chilisim) can be executed from the command line as mentioned below. It takes an absolute file <file>, generated by the gcc compiler, as the input.

Syntax :

```
% chilisim –z <file>
```

By default, data and code sections are stored in the 64K Core RAM.

Command line arguments to the simulated application are set with the –a switch.

Examples:

```
/* standard C main fuction */
int main(int argc, char **argv)
{
}

# calling the simulator with 2 application parameters arg1 and arg2
%chilisim –z text.exe –a “arg1 arg2”

argc = 2
argv[0]=test.exe
argv[1]=arg1
argv[2]=arg2

# string parameter to –a is empty
%chilisim –z text.exe –a “”

argc = 1
argv[0]=test.exe

# -a is not used here
%chilisim –z text.exe

argc = 0
```

3. Arguments for the DMS (Data Memory Subsystem)

By default, the DMS is not simulated (DMA commands and DMA controller).
To simulate with the CHILI DMS, the dynamic library “libdms2.so” (file included in the chili toolchain package) needs to be specified by the –d switch.

Example:

```
-d libdms2.so
```

Scratch pads models need also to be specified with –e, -f, -g, -j switches.
The scratch pas models available in the dms2.so library need to be specified with their function entry point which are “createPort0” and “createPort1”

Example:
```
-e libdms2.so:createPort0 -flibdms2.so:createPort1
```

The layout of the data memory model needs to be specified with the switch “–x”.

They consist of :

-c[address range] : defines the address range of the core memory (coreRAM)
-l[address range] : defines the address range of the local memory (sRAM)
-d[addess range]: defines the address range of the external memory (DRAM).

Example:
```
-x"-c0:0x200000 -l0x1000000:0x100FFFF -d0x10000000:0x10100000"
```

NOTE1: the memory layout specified within the –x arguments has to match the section layout specified in the linker script.

NOTE2: the dms2.so library will output by default warnings when uninitialized memory are read. To stop the output of warnings, use the –W switch.

4. Debug Interactive mode

Use the –D switch to start the simulator in debugging mode.

```
% chilisim –D
Interactive debugging mode started. Try help for information.
chilisim>
```

Use “help” to get the list of available commands.

5. Simulation Interruption

It is possible to stop a running simulation with a CTRL-C interruption. The simulation will switch to interactive debug mode. To exit the debugger and continue the simulation in standard mode, use the command “exit”. To quit the simulation, use the “quit” command.

6. Verbose Output

A verbose mode can be set with the –v switch following by a digit representing the verbose level. The 5 verbose levels are the following:

-v1: cycle counter + instruction address

Example:
```
#CYCLE       2645  #ADDR 0004978f
```

```
#CYCLE        2646   #ADDR 00049794
#CYCLE        2647   #ADDR 00049798
#CYCLE        2648   #ADDR 0004979c
```

-v2: register contents

Example:

```
Cycle: 607
   R0=00000002       R1=20000000       R2=00000400       R3=1FFFFF3C       R4=00000000
R5=00000000   R6=00000000   R7=00000000
   R8=00000000        R9=00000000      R10=00000400      R11=00000000      R12=00000000
R13=1FFFFFD0  R14=00000000  R15=00000000
  R16=10000000      R17=00000000      R18=00000006      R19=00000008      R20=00000000
R21=00000000  R22=00000000  R23=20000000
  R24=00000000      R25=00000000      R26=00000000      R27=0000001F      R28=00000000
R29=00000000  R30=00000000  R31=00000000
  R32=00000000      R33=00000000      R34=00000000      R35=00000000      R36=00000000
R37=00000000  R38=00000000  R39=00000000
  R40=00000000      R41=00000000      R42=00000000      R43=00000000      R44=00000000
R45=00000000  R46=00000000  R47=00000000
  R48=00000000      R49=00000000      R50=00000000      R51=00000000      R52=00000000
R53=00000000  R54=00000000  R55=00000000
  R56=00000000      R57=00000000      R58=00000000      R59=00000000      R60=00000000
R61=1FFFFF14  R62=1FFFFED0  R63=0005094C
```

-v3 : pipeline content

```
#CYCLE 239:
#ADDR 3b18d:
spy_src_0_dat_s:  >00000000   >00000000   >00000000   >00000000
spy_src_1_dat_s:  >00000000   >00000000   >00000000   >00000000
spy_dst_r_dat_s:  >00000000   >00000000   >00000000   >00000000
dst_dat_comb:     >00000000   >00000000   >00000000   >00000000
dst_dat_ss:       >00000000   >00000000   >00000000   >00000000
dst_dat_sss:      >00000000   >00000000   >00000000   >00000000
dst_dat_post:     >00000000   >00000000   >00000000   >00000000
register file:    >00000000   >0000088c   >00036946   >00000000
                  >00000000   >00000000   >00000000   >0000088c
                  >0000b31c   >00000000   >00000000   >00000000
                  >00000000   >00000000   >00000000   >00000000
                  >00000064   >0000089c   >0000089c   >0000088c
                  >0000b468   >00000000   >00000000   >00000000
                  >0000b468   >00000000   >00000000   >00000000
```

-v4 : Disassembler output

At every cycle, the simulator output the 4-slots disassembled instructions and the contents of the 64 registers.

Example:

```
Cycle: 531
#ADDR 00037F8E:
jump(0x38012)
if(R16 == 0x0)
port32[R62 + -4] = R63;
R62 = R62 + -4;

   R0=00000000     R1=0000B31C     R2=00000400     R3=1FFFFF3C     R4=00000000
R5=00000000   R6=00000000   R7=00000001
   R8=0000B664     R9=0000B664    R10=00000000    R11=00000000    R12=00000000
R13=1FFFFFD0  R14=00000000  R15=00000000
```

```
  R16=00000000    R17=00000000    R18=0000B6C0    R19=00042A28    R20=00000000
R21=00042C63  R22=00000000  R23=00000000
  R24=00000000    R25=00000000    R26=00000000    R27=00000000    R28=00000000
R29=00000000  R30=00000000  R31=00000000
  R32=00000000    R33=00000000    R34=00000000    R35=00000000    R36=00000000
R37=00000000  R38=00000000  R39=00000000
  R40=00000000    R41=00000000    R42=00000000    R43=00000000    R44=00000000
R45=00000000  R46=00000000  R47=00000000
  R48=00000000    R49=00000000    R50=00000000    R51=00000000    R52=00000000
R53=00000000  R54=00000000  R55=00000000
  R56=00000000    R57=00000000    R58=00000000    R59=00000000    R60=00000000
R61=1FFFFF78  R62=1FFFFF2C  R63=0004450D
```

-v5 : Function calls output

With this verbose level, the simulator outputs a line every time a function is called and returned. The function name and arguments values (2) are displayed.
The cycle count, pipeline and DMS stalls are also displayed.

Example:

```
%chilisim -z functionPointer.exe -v5

FUNCTION_CALL (main , 0x00000000 , 0x00000000) 12:3:0
-FUNCTION_CALL (__main , 0x00000000 , 0x00000000) 21:3:0
--FUNCTION_CALL (atexit , 0x00000642 , 0x00000000) 53:3:0
---FUNCTION_CALL (__register_exitproc , 0x00000000 , 0x00000000) 59:9:0
---110 end_func 128:21:0
--121 end_func 142:24:0
-132 end_func 156:27:0

Core:0 Program has finished within 156 cycles (PIPELINE stalls: 27, DMS
stalls: 0).
R0: 0x00000000
```

Function names can be added to –v5 to select only the function to be outputted.

Example:

```
 -v5:main:f1
```

Only trace for function “main” and “f1”

7. Profiler Output

-p <file> : output profile information in file <fiel>

Example of profiling output (no DMS here):

```
Total cycles: 3302936
with DMS stalls: 0
Pipeline stalls: 0

Cycles per function (including subfunctions):
Cycles:  Total          pl stalls       dms stalls

99.76%   3295044        0               0               MpegAudioDecoder
```

```
63.91%    2111017         0                0              mad_frame_decode
63.49%    2097341         0                0              mad_layer_III
62.36%    2059945         0                0              III_decode
18.35%    606403          0                0              mad_synth_frame
18.34%    605808          0                0              synth_full
18.32%    605150          0                0              III_imdct_l
4.6%      152207          0                0              mad_bit_read
3.37%     111384          0                0              dct32
1.6%      52946           0                0              _vfprintf_r
1.1%      36400           0                0              fastsdct
0.94%     31277           0                0              __sfvwrite
0.8%      26661           0                0              fprintf
0.8%      26523           0                0              vfprintf
0.47%     15834           0                0              memmove
0.4%      13359           0                0              fwrite
0.39%     12907           0                0              mad_header_decode
0.3%      10030           0                0              __udivsi3
0.23%     7818            0                0              puts
0.22%     7327            0                0              memcpy
....
....
....
Function self cycles (without subfunctions):
Function BstdFileEofP at address: 0x51
    Calls: 1 Cycles: 11
    Average Cycles: 11
    No min/max information
    Max local stack in function: 80120 Byte
    Max stack in function: 80192 Byte
    Smallest heap address in function: 4294967295 Byte

Function BstdRead at address: 0xab
    Calls: 2 Cycles: 174
    Average Cycles: 87
    Min Cycles: 51 Max Cycles: 123
    Max local stack in function: 80148 Byte
    Max stack in function: 80220 Byte
    Smallest heap address in function: 469760640 Byte
```

8. Multi-core simulation

-M <n> : number of cores
.Z core1:core2:…:core<n> list of Elf binaries files for each core

When profiling is used, a suffix corresponding to the core-id is added to
To the output profiled filenames.

Example:

```
chili-odm-elf-gcc core0.c -o core0
chili-odm-elf-gcc core1.c -o core1
chili-odm-elf-gcc core2.c -o core2

chilisim -M 3 -Z core0:core1:core2 -p profile.txt
```

Profiler files created are:

```
0_profile.txt
1_profile.txt
2_profile.txt
```

Debug PLI

This mechanism allows the user to customize the debug output of the simulator without

changing the source code of the simulated C application. (same concept as Verilog API).

A C file need to be compiled as a shared library and contains callback routines that will be executed
at the beginning or end of a function call during simulation.
(at jsr(r63, func) or ret(63) chili instructions).

The C file should also contain the registration of the callbacks.

An API provides a set of functions to access the register or memory values.

In the PLI:
2 functions need to be defined:

```
void ChilisimDebugPLIinit(void *core);
void ChilisimDebugPLIclose(void *core)
```

They are executed once by each core:

ChilisimDebugPLIinit at the start of simulation and
ChilisimDebugPLIclose at the end.

Callbacks registration should be done in ChilisimDebugPLIinit() with the routine ChilisimDebugPLIAddFunction(void *core, char *fname**,** void (*f)(void *), int when) (see example for usage).

The access to the simulator internal values is done with the following routines:

```
#include "pli/chilisim_pli.h"

extern unsigned int chilisim_debug_pli_get_register_value(void
*core, int regno);
extern unsigned char chilisim_debug_pli_get_memory_value(void
*core, unsigned int address);
extern unsigned int chilisim_debug_pli_get_cycle(void *core);
extern unsigned int chilisim_debug_pli_get_coreid(void *core);
```

Compile PLI with:

```
gcc -g -shared -fPIC my_debug.c -o my_debug.so -
I/prj/cad/odc/chili/test64/include/chili
```

Run with Chilisim by adding following switch:

```
  -l ./my_debug.so
```

Example:

```
/* ---------------------------------------------------------------------------
   Chilisim Debug PLI example

Compiled with:
gcc      -g      -shared      -fPIC      check_fifo.c      -o      check_fifo.so      -
I/prj/cad/odc/chili/test64/include/chili

Run with Chilisim simulator by adding following switch:
  -l ./check_fifo.so

ex:
chilisim -W -d libdms2.so -e libdms2.so:createPort0 -flibdms2.so:createPort1
         -x"-c0:0x200000 -l0x1000000:0x150FFFF -d0x10000000:0x12000000"
         -M 2 -Z reader.exe:writer.exe
         -l ./check_fifo.so
--------------------------------------------------------------------------- */

#include<stdio.h>

/* API header, NEEDED */
#include "pli/chilisim_pli.h"

#define MAXARRAY 512000

/* user file pointers */
unsigned char core0[MAXARRAY];
unsigned char core1[MAXARRAY];

int idx_core0 = 0;
int idx_core1 = 0;

FILE *ferr=NULL;

/* callback when entering user C function "writeFifo" - core0 */
void my_debug_writeFifo_begin(void *core)
{
    int r1= chilisim_debug_pli_get_register_value(core, 1);
    int r2= chilisim_debug_pli_get_register_value(core, 2);
    int i=r1;
    unsigned char val;

    printf("[check_fifo.so]  my_debug_writeFifo_begin  r1=%x  r2=%x  idx_core0=%d\n",
r1, r2, idx_core0);

    const char *fname = chilisim_debug_pli_get_parent_function_name(core);

if (fname)
        printf("[check_fifo.so:my_debug_writeFifo_begin]  inside  function:  %s  \n",
fname);

    if (idx_core0+r2>=MAXARRAY)
        return;

    while(i<r1+r2)
    {
        val = chilisim_debug_pli_get_memory_byte_value(core, i);

        core0[idx_core0] = val;
        i++;
        idx_core0++;
    }
}

/* callback when exiting user C function "readFifo" - core1 */
void my_debug_readFifo_end(void *core)
{
    int r1= chilisim_debug_pli_get_register_value(core, 1);
    int r2= chilisim_debug_pli_get_register_value(core, 2);
    int i=r1;
    unsigned int val;
    int ok = 1;

    if (idx_core1+r2>=MAXARRAY)
        return;
```

```
    while(i<r1+r2)
    {
        val = chilisim_debug_pli_get_memory_byte_value(core, i);

        if (val != core0[idx_core1])
        {
            fprintf(ferr, "[check_fifo.so] diff on Element %d, at cycle:%d\n",
                    idx_core1,
                    chilisim_debug_pli_get_cycle(core));
            ok = 0;
        }

        core1[idx_core1] = val;
        i++;
       idx_core1++;
    }

    chilisim_debug_pli_dump_callstack(core);

}

/* NEEDED -- called by simulator */
void ChilisimDebugPLIinit(void *core)
{
    /* register callbacks */


ChilisimDebugPLIAddFunction(core,"writeFifo",&my_debug_writeFifo_begin,CHILISIM_AT_
BEGIN);

ChilisimDebugPLIAddFunction(core,"readFifo",&my_debug_readFifo_end,CHILISIM_AT_END)
;

    if (chilisim_debug_pli_get_coreid(core) == 0)
        ferr = fopen("./t_check_fifo.txt","w");
}

/* NEEDED -- called by simulator */
void ChilisimDebugPLIclose(void *core)
{
    /* core0 makes the check */
    if (chilisim_debug_pli_get_coreid(core) == 0)
    {
        if (idx_core0 != idx_core1)
        {
            printf("Nb Elements are diff idx_core0=%d idx_core1=%!!\n",
                   idx_core0, idx_core1 );
        }

        int i=0;
        while(i<idx_core1)
        {
            if (core0[i] != core1[i])
            {
                printf("Diff  on  element  %i  core0=%d  core1=%d!!\n",  i,  core0[i],
core1[i]);
            }
            i++;
        }

        printf("FIFO: all %d bytes match!!\n", idx_core0);

      fclose(ferr);
    }
}
```