Verification and support custom CSR’s to SHAKTI Ecosystem

A project report

Submitted by

KARABALWAD GANESH HANAMNLU

In the partial fulfilments of the requirements
For the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

JUNE 2021

THESIS CERTIFICATE

This is to certify that the thesis titled **Verification and support custom CSR's to SHAKTI Ecosystem,** submitted by **KARABALWAD GANESH HANAMNLU,** to the Indian Institute of Technology, Madras for the award of the degree **Master of Technology,** is a bonafide record of the research work done by him under our supervision. In full or in parts, the contents of this thesis have not been submitted to any other University or Institute for the award of any degree or diploma.

Prof. V. Kamakoti
Project guide
Department of Computer Science
IIT, Madras – 600 036

Place: Chennai
Date:

ACKNOWLEDGEMENTS

I wish to express my sincere thanks to Prof. V. Kamakoti for providing me all the necessary facilities for the project.

I place on record my sincere thanks to my mentor Mr. Sathyanarayan for continuous encouragement.

I am also grateful for the SHAKTI hardware and software team. I am extremely thankful and indebted to them for sharing their expertise, and sincere and valuable guidance and encouragement extended to me.

ABSTRACT

In this thesis, information regarding custom registers present in the c class core is provided. We need custom registers to control and monitor the system state. CSR’s can be read and written, and the bits can be set and reset. CSR’s are different than regular registers used for the standard computations such as arithmetic. They need a perticular way to enable them. We need assembly language to enable them. After enabling, verification takes place to check whether they are working as per required functionality.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

SoC	System On Chip
CSR	Control and Status Register
RISC	Reduced Instruction Set Computer
BSV	Bluespec System Verilog
DMA	Direct Memory Access
ASM	Assembly Language
ISA	Instruction Set Architecture

CHAPTER 1

INTRODUCTION

SHAKTI project is an open-source initiative by the Reconfigurable Intelligent Systems Engineering (RISE) group at IIT-Madras. The SHAKTI project is trying to build a family of six processors. It is based on the RISC-V ISA. The current SoC developments are for the Controller (C-Class) and Embedded (E- Class) classes.

1.1 Introduction to C Class?

C-class is a part of the SHAKTI family of processors. It is a highly configurable and commercial-grade-5-stage in order core supporting the standard RV64GCSUN ISA extensions. The core generator in this repository can configure the core to generate a large range of design instances from the same high-level source code. The design instances can serve domains ranging from embedded systems, motor-control, IoT, storage, industrial applications to low-cost, high-performance Linux-based applications such as networking, gateways, etc.

There have been a lot of successful silicon prototypes of the different instances of the c-class; thus proving its versatile property. The more significant parameterization of the design in conjuction with using an HLS like BSV makes it easy to continually add new features and design points.

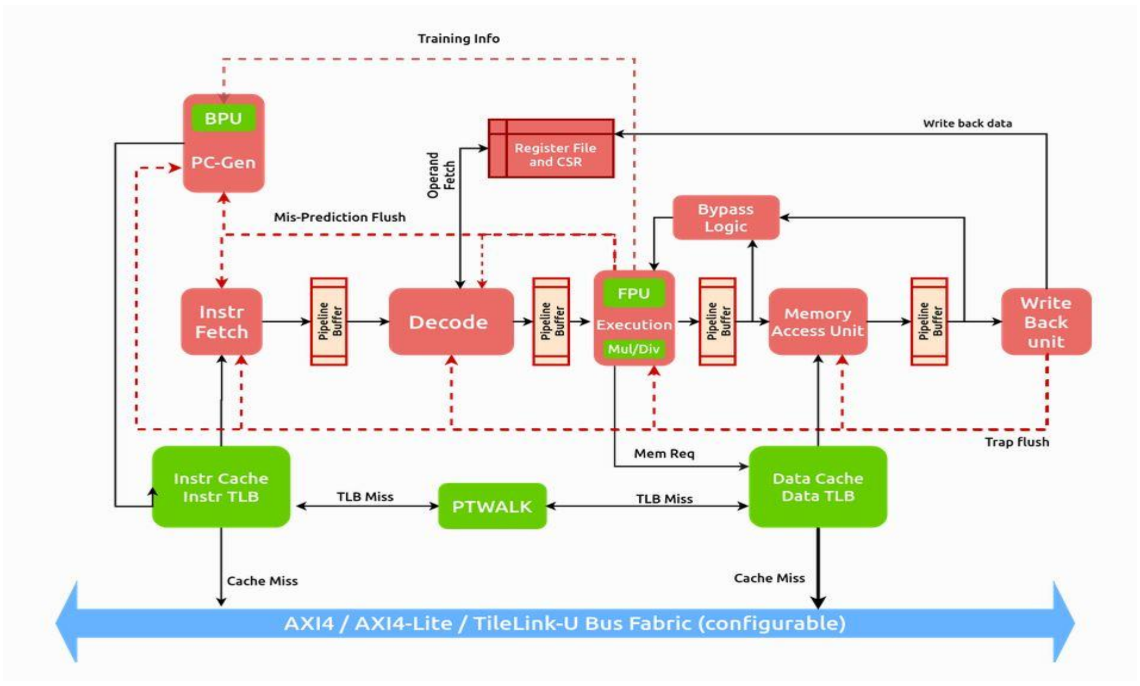


Figure 1. 1 Block diagram of a C-Class Core

1.2 Introduction to CSR’s

The Control and Status Register (CSR) are the system registers. RISC-V provides them to control and monitor system states. We can read the CSR’s, and write on them. Bits can be set or cleared. RISC-V provides distinct CSR’s for every privilege level. Each CSR has a particular name, and they are assigned a unique function. Processor operation will get affected when we read or write on CSR’s. We can’t use normal registers for all the operations; hence CSR’s are used where we can’t use regular registers. Such processes can be like knowing about system configuration; they can handle exceptions or switch to different privilege modes and handling interrupts, etc. They can’t be read or written the way a standard register can. They need a unique set of instructions called CSR instructions. CSR instructions require an intermediate base register for any operation on CSR registers. It is again possible to write immediate values to CSR registers.

CHAPTER 2

ESSENTIAL LEARNINGS

2.1 Bluespec

BSV (Bluespec System Verilog) is a language used to design of electronic systems such as ASICs, FPGAs, etc. BSV is used for the spectrum of applications such as processors, memory subsystems, interconnects, DMA's and data movers. It is used in multimedia and communication I/O devices, multimedia and communication codecs and processors as well. It is used in accelerators, high-performance computing accelerators, signal processing accelerators, etc. It is also used across markets, from low-power, portable consumer items to enterprise-class server-room systems.

It is used in many design activities because it is both a very high-level language as well as it synthesizes fully to hardware. The combination of high level and full synthesizability enables many of these activities that were previously only done in software simulation now to be moved easily to FPGA-based execution. It can help us to speed up performance by three to six orders of magnitude. Such a dramatic speed not only accelerates existing activities but enables new activities that were not feasible before.

The entire core is implemented in BSV. BSV guarantees synthesizable circuits, BSV also gives a high-level abstraction, like going from assembly to C. It enables to work at a much higher level, thereby increasing throughput.

2.2 RISC-V ASM

RISC-V pronounced as "RISC-five," is an open-source standard Instruction Set Architecture (ISA). It is designed based on Reduction Instruction Set Computer (RISC) principles. It has a flexible architecture so that it is used to build systems ranging from a simple microprocessor to complex multi-core systems, and hence RISC-V caters to any market. It provides two specifications, one, the User Level Instructions, which guides in development of simple embedded systems and connectivity applications. And another is the Privilege Level Instructions, which suggests in building secure systems, kernel, and stacks of protected softwares.

It currently supports three privilege levels, they are Machine/Supervisor/User, each group having dedicated CSR's to observe the system state and it's manipulation. In addition, RISC-V provides 31 write and registers. Well, they all can be used as general-purpose registers, and they have commited functions also. RISC-V is divided into various categories based on the maximum width of registers the architecture can support. For example, RV32 provides registers whose maximum width is 32-bits whereas for RV64 the maximum width is of 64-bits. The Processors with larger register widths can support data and instructions of smaller widths sizes. Hence, RV64 platform supports both RV32 and RV64.

2.3 Verilator

Verilator is open-source software tool which is used to convert Verilog to a cycle-accurate behavioral model in C++ or System C. It is fastest Verilog or SystemVerilog simulator. It accepts synthesizable Verilog or SystemVerilog. It performs lint code-quality checks and compiles into multithreaded C++, or SystemC. It does not simply convert Verilog HDL to C++ or SystemC, rather, verilator compiles the code into a much faster optimized and optionally thread-partitioned model, which is in turn wrapped inside a C++ or a SystemC module. The results are an assembled Verilog model that executes even on a single-thread over 10x faster than standalone SystemC. On a single thread it is about 100 times faster than interpreted Verilog simulators such as Icarus Verilog. By multithreading, 2-10x speedup can be gained.

2.4 GDB Setup

GDB, the GNU Project debugger, allows us to see what is going on 'inside' another program when it is executing a program. It also checks, what another program doing at the moment it crashed.

It can do four main types of things to help us to catch bugs in the program:

- Start the program, specifying anything that may affect its behavior.
- Make the program halt on a stated constraints.
- Inspect what happened, when the program has stopped.
- Change things in your program, so we can experiment with correcting the effects of one bug and go on to learn about next.

Those programs might be executing on the same machine as GDB (native), on another device (remote), or a simulator. GDB can run the most popular UNIX and Microsoft Windows variants, as well as on Mac OS X.

CHAPTER 3

Introduction to CSR’s and Configuration files

3.1 Custom CSR’s available in C-Class

The C-class includes the following custom CSR's implemented in the non-standard space for extra control and unique features.

- **Custom control csr (0x800)**

The CSR is used to enable or disable the caches, branch predictor, and arithmetic exceptions at run-time.

Bit position	Reset Value	Description
0	From config	Enable or disable the data cache
1	From config	Enable or disable the instruction cache
2	From config	Enable or disable the branch predictor
3	Disabled	Enable or disable arithmetic exceptions

Table 3. 1 Bit position and description

- **dtvec csr (0x7c0)**

XLEN register, which indicates the address of the debug loop when the debugger stops the core.

- **denable csr (0x7c1)**

1-bit csr indicating if the debugger can break the core.

- **mhpminterrupten csr (0x7c2)**

XLEN bit register following the same encoding as ‘mcounteren/mcountinhibit.’ A bit set to 1 indicates an interrupt will be generated when the corresponding counter reaches the value 0.

- **dtim base address csr (0x7c3)**

An XLEN bit register is holding the base address of the data tightly integrated scratch memory. It should correspond to the physical address space and not the virtual.

- **dtim bound address csr (0x7c4)**
  An XLEN bit register is holding the bound address of the data tightly integrated scratch memory. It should correspond to the physical address space and not the virtual.

- **itim base address csr (0x7c5)**
  An XLEN bit register is holding the base address of the instruction tightly integrated scratch memory. It should correspond to the physical address space and not the virtual.

- **itim bound address csr (0x7c6)**
  An XLEN bit register is holding the bound address of the instruction tightly integrated scratch memory. It should correspond to the physical address space and not the virtual.

3.2 Understanding the CONFIGURATION FILES

A configuration file, often shortened to 'config file', defines the parameters, options, settings, and preferences applied to operating systems, infrastructure devices, and applications.

Hardware and software devices can be profoundly complex, supporting countless options and parameters. The operator must explicitly delineate settings and choices appropriate for the specifics of the data center, cloud, or user environment. Configuration file information specifies, for example, where log files from an application are stored via the storage path, which plug-in are allowed in a given program, and ever the color scheme and dashboard widget preferences in a user interface.

The C- class core is configurable and highly parameterized. By changing a single configuration, one can generate core instances ranging from embedded micro-controllers to Linux-capable high-performance cores. The user in a YAML file should specify the configuration. Sample YAML files are available in the 'sample_config/' directory of the c-class repository. At times the user may specify conflicting configurations which are illegal and can be detected only during compile or simulation time. To catch them early, the configurator maintains a schema of valid structures and alerts the user when an illegal configuration is provided. The source code is compiled only when a legal arrangement is detected.

The output of the configurator is a makefile. inc file, which contains necessary variables, to be used in the master Makefile for bluespec compilation, verilator linking, simulation, verification, and other collateral information.

CHAPTER 4

DEBUGGING

4.1 Debugging with RISC-V GDB and Spike

Debugging takes place as part of both hardware and software development. In hardware development, peripherals, the memory, registers are initialized, and we instantiate a platform for development of an application. Software debugging is more accessible because, care of logs and memory dumps is taken by OS. Yet, the hardware debugging is challenging because it has to be done over bare metal with limited memory and specific capabilities.

4.1.1 Spike

Spike is a RISC-V instruction set simulator. When we don't have the physical device, then we can use spike for development of applications. Hence, the code can be easily migrated to real hardware later. Spike simulates the processor development environment

4.1.2 OpenOCD

The Open On-Chip Debugger (OpenOCD) is a chunk of software which provides an interface for the RISCV-GDB to connect to the target device. It allows the RISC-V GDB to connect to the target microcontroller through a debug adapter.

4.1.3 Debug adapter

In order to allow the Host Computer to communicate and introspect the target hardware, we need an additional hardware. These devices are often attributed as a Debugger devices/Debug adapters. Using the debugger devices, the compiled program is transferred to the target hardware.

4.1.4 RISC-V GDB

RISC-V GDB is the GNU debugger for RISC-V platforms. A debugger allows us to pause a program, examine various addresses, change variables, and step through the code. In this way, we can find the exact location and cause of the problem. The GDB controls the target hardware, while debugging. We can pass instructions through gdb to control the target device.

4.2 How OpenOCD and RISC-V GDB works in SHAKTI?

1. The FPGA board which is connected is the hardware, that can use many kinds of serial interfaces like SPI or JTAG, or GPIO.
2. Firstly, connect the FPGA board to PC using a USB cable.
3. Then run OpenOCD on your PC, which supports remote GDB protocol.
4. OpenOCD listens for GDB connections on the default port 3333. In RISC-V GDB, connect to OpenOCD by typing target remote localhost:3333.
5. It would cause RISC-V GDB to connect to the gdbserver on the local PC using port 3333. Now RISC-V GDB sends the commands it knows and receives data it understands.

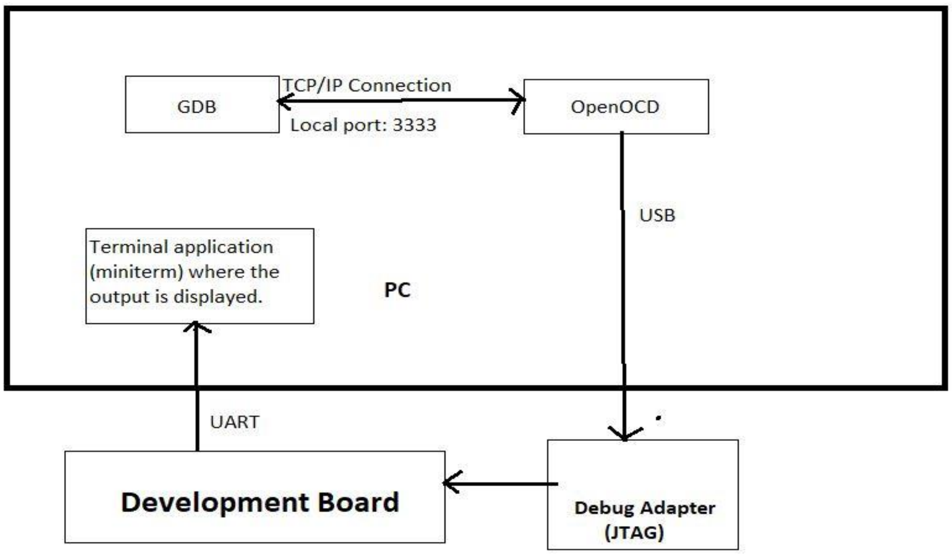


Figure 4. 1 Connection of GDB and OpenOCD to hardware

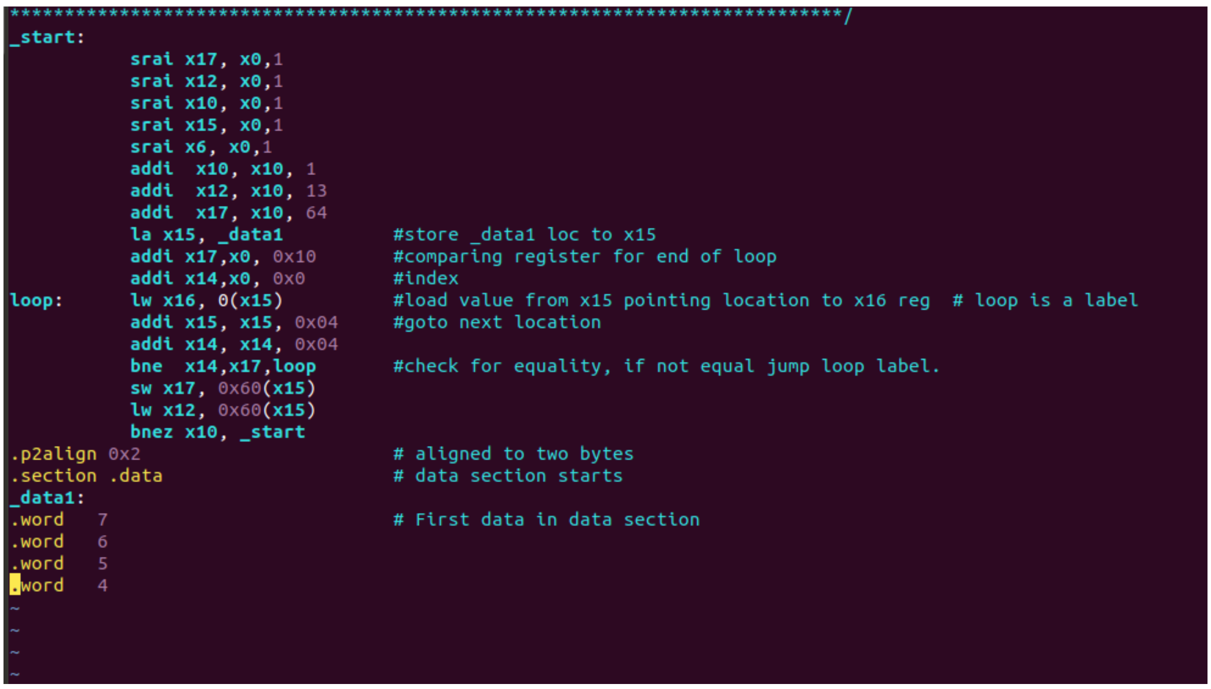
Thus RISC-V GDB and OpenOCD work together to interact with the device and perform any operation.

CHAPTER 5

Experiments and Results

5.1 Debugging of ASM Program

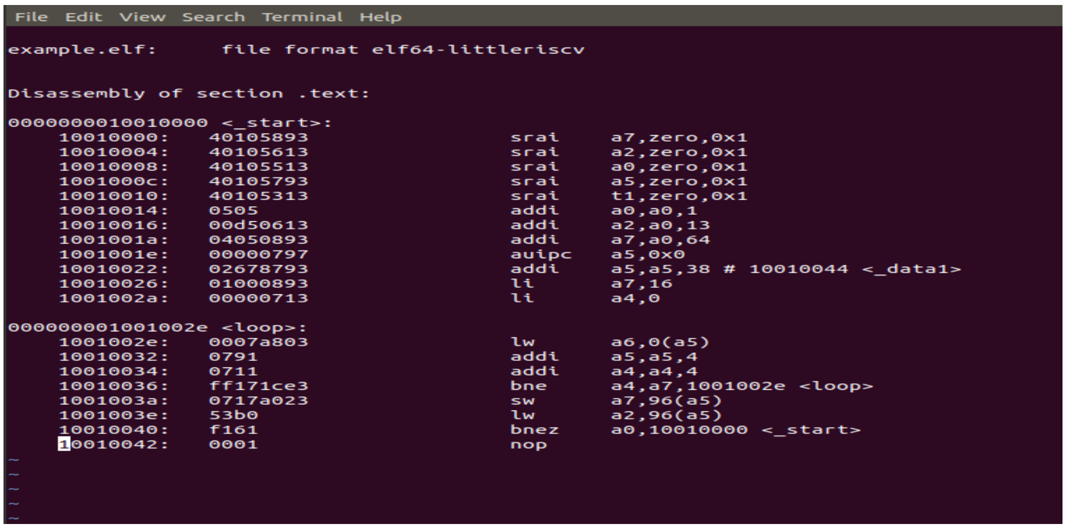
5.1.1 First ASM Program



```
****************************************************************************/
_start:
           srai x17, x0,1
           srai x12, x0,1
           srai x10, x0,1
           srai x15, x0,1
           srai x6, x0,1
           addi  x10, x10, 1
           addi  x12, x10, 13
           addi  x17, x10, 64
           la x15, _data1          #store _data1 loc to x15
           addi x17,x0, 0x10       #comparing register for end of loop
           addi x14,x0, 0x0        #index
loop:      lw x16, 0(x15)          #load value from x15 pointing location to x16 reg  # loop is a label
           addi x15, x15, 0x04     #goto next location
           addi x14, x14, 0x04
           bne  x14,x17,loop       #check for equality, if not equal jump loop label.
           sw x17, 0x60(x15)
           lw x12, 0x60(x15)
           bnez x10, _start
.p2align 0x2                       # aligned to two bytes
.section .data                     # data section starts
_data1:
.word   7                          # First data in data section
.word   6
.word   5
.word   4
~
~
~
~
```

Figure 5. 1 First ASM Program

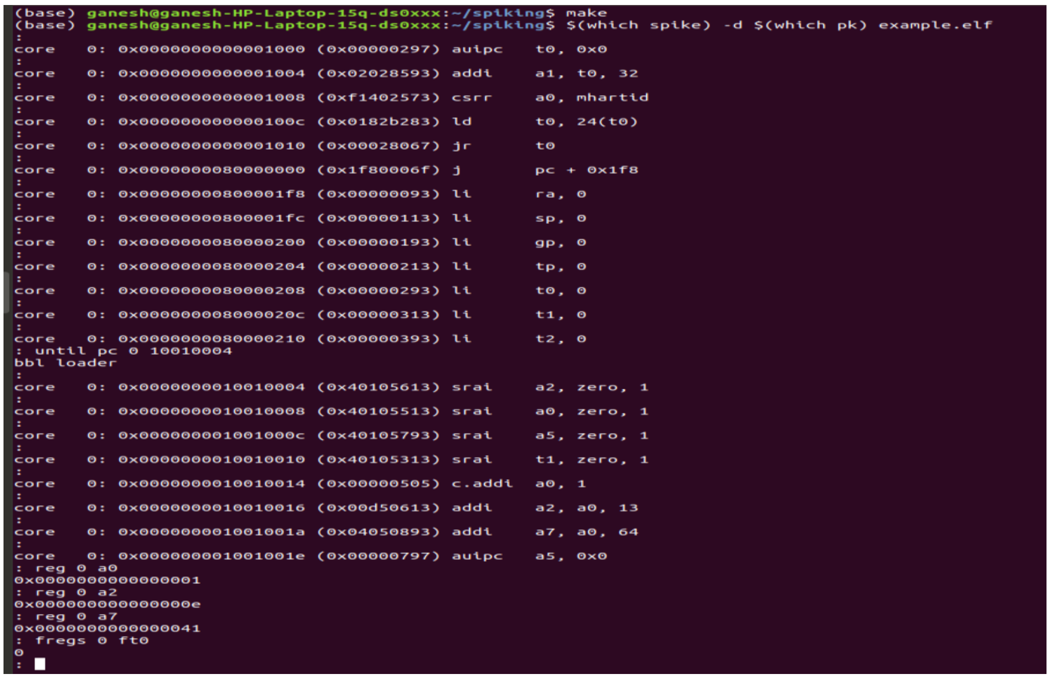
5.1.2 Object dump file for the compiled ASM Program



```
example.elf:     file format elf64-littleriscv


Disassembly of section .text:

0000000010010000 <_start>:
    10010000:   40105893                srai    a7,zero,0x1
    10010004:   40105613                srai    a2,zero,0x1
    10010008:   40105513                srai    a0,zero,0x1
    1001000c:   40105793                srai    a5,zero,0x1
    10010010:   40105313                srai    t1,zero,0x1
    10010014:   0505                    addi    a0,a0,1
    10010016:   00d50613                addi    a2,a0,13
    1001001a:   04050893                addi    a7,a0,64
    1001001e:   00000797                auipc   a5,0x0
    10010022:   02678793                addi    a5,a5,38 # 10010044 <_data1>
    10010026:   01000893                li      a7,16
    1001002a:   00000713                li      a4,0

000000001001002e <loop>:
    1001002e:   0007a803                lw      a6,0(a5)
    10010032:   0791                    addi    a5,a5,4
    10010034:   0711                    addi    a4,a4,4
    10010036:   ff171ce3                bne     a4,a7,1001002e <loop>
    1001003a:   0717a023                sw      a7,96(a5)
    1001003e:   53b0                    lw      a2,96(a5)
    10010040:   f161                    bnez    a0,10010000 <_start>
    10010042:   0001                    nop
~
~
~
~
~
```

Figure 5. 2 Object Dump file after compiling the ASM Program

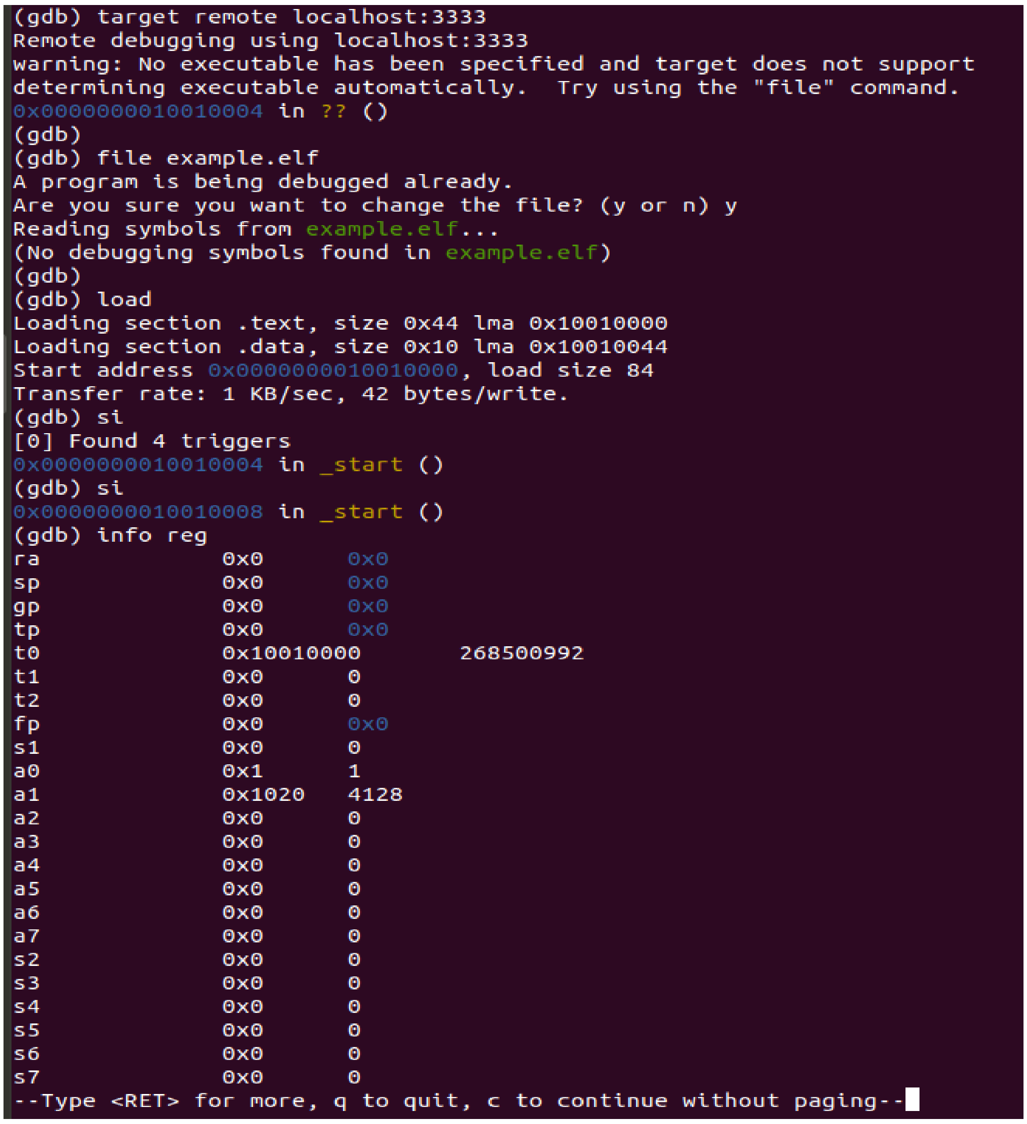
5.1.3 Debugging an ASM program with Spike



```
(base) ganesh@ganesh-HP-Laptop-15q-ds0xxx:~/spiking$ make
(base) ganesh@ganesh-HP-Laptop-15q-ds0xxx:~/spiking$ $(which spike) -d $(which pk) example.elf
:
core   0: 0x0000000000001000 (0x00000297) auipc   t0, 0x0
:
core   0: 0x0000000000001004 (0x02028593) addi    a1, t0, 32
:
core   0: 0x0000000000001008 (0xf1402573) csrr    a0, mhartid
:
core   0: 0x000000000000100c (0x0182b283) ld      t0, 24(t0)
:
core   0: 0x0000000000001010 (0x00028067) jr      t0
:
core   0: 0x0000000080000000 (0x1f80006f) j       pc + 0x1f8
:
core   0: 0x00000000800001f8 (0x00000093) li      ra, 0
:
core   0: 0x00000000800001fc (0x00000113) li      sp, 0
:
core   0: 0x0000000080000200 (0x00000193) li      gp, 0
:
core   0: 0x0000000080000204 (0x00000213) li      tp, 0
:
core   0: 0x0000000080000208 (0x00000293) li      t0, 0
:
core   0: 0x000000008000020c (0x00000313) li      t1, 0
:
core   0: 0x0000000080000210 (0x00000393) li      t2, 0
: until pc 0 10010004
bbl loader
:
core   0: 0x0000000010010004 (0x40105613) srai    a2, zero, 1
:
core   0: 0x0000000010010008 (0x40105513) srai    a0, zero, 1
:
core   0: 0x000000001001000c (0x40105793) srai    a5, zero, 1
:
core   0: 0x0000000010010010 (0x40105313) srai    t1, zero, 1
:
core   0: 0x0000000010010014 (0x00000505) c.addi  a0, 1
:
core   0: 0x0000000010010016 (0x00d50613) addi    a2, a0, 13
:
core   0: 0x000000001001001a (0x04050893) addi    a7, a0, 64
:
core   0: 0x000000001001001e (0x00000797) auipc   a5, 0x0
: reg 0 a0
0x0000000000000001
: reg 0 a2
0x000000000000000e
: reg 0 a7
0x0000000000000041
: fregs 0 ft0
0
: 
```

Figure 5. 3 Debugging of an ASM program using Spike

5.1.4 Debugging an ASM Program with GDB

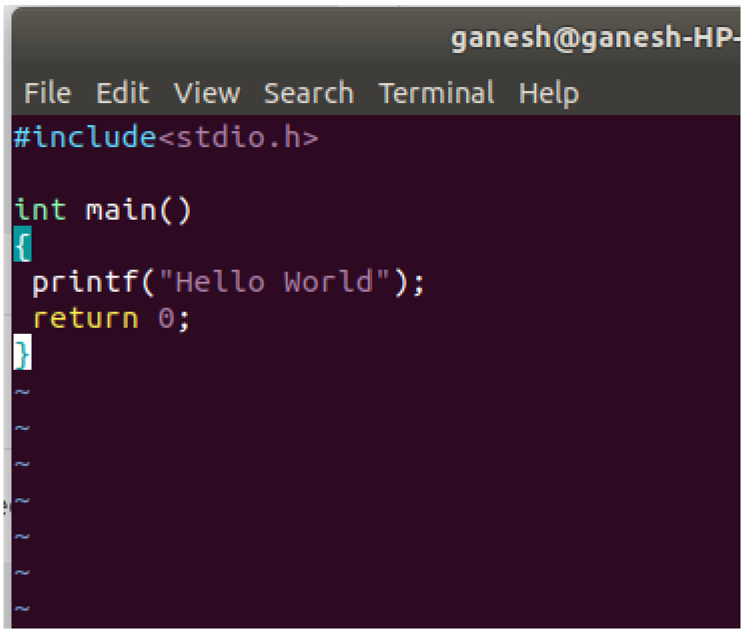


```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x0000000010010004 in ?? ()
(gdb)
(gdb) file example.elf
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Reading symbols from example.elf...
(No debugging symbols found in example.elf)
(gdb)
(gdb) load
Loading section .text, size 0x44 lma 0x10010000
Loading section .data, size 0x10 lma 0x10010044
Start address 0x0000000010010000, load size 84
Transfer rate: 1 KB/sec, 42 bytes/write.
(gdb) si
[0] Found 4 triggers
0x0000000010010004 in _start ()
(gdb) si
0x0000000010010008 in _start ()
(gdb) info reg
ra             0x0      0x0
sp             0x0      0x0
gp             0x0      0x0
tp             0x0      0x0
t0             0x10010000       268500992
t1             0x0      0
t2             0x0      0
fp             0x0      0x0
s1             0x0      0
a0             0x1      1
a1             0x1020   4128
a2             0x0      0
a3             0x0      0
a4             0x0      0
a5             0x0      0
a6             0x0      0
a7             0x0      0
s2             0x0      0
s3             0x0      0
s4             0x0      0
s5             0x0      0
s6             0x0      0
s7             0x0      0
--Type <RET> for more, q to quit, c to continue without paging--
```

Figure 5. 4 Debugging of an ASM using GDB

5.2 Debugging of a C Program

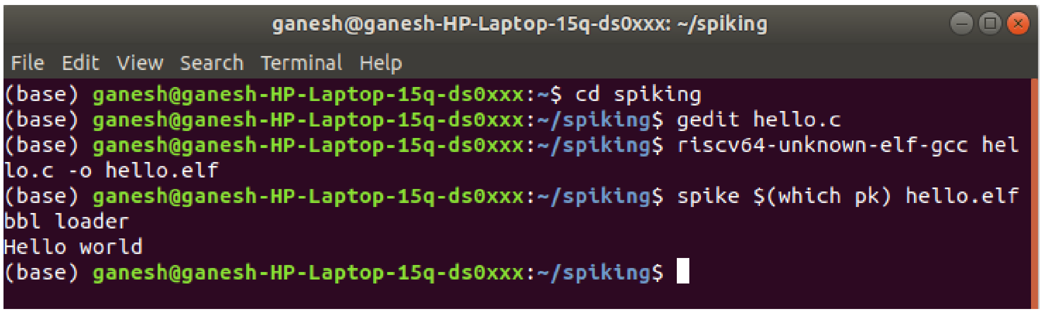
5.2.1 Simple C code



```
#include<stdio.h>

int main()
{
 printf("Hello World");
 return 0;
}
~
~
~
~
~
~
~
```

Figure 5. 5 A simple code for debugging

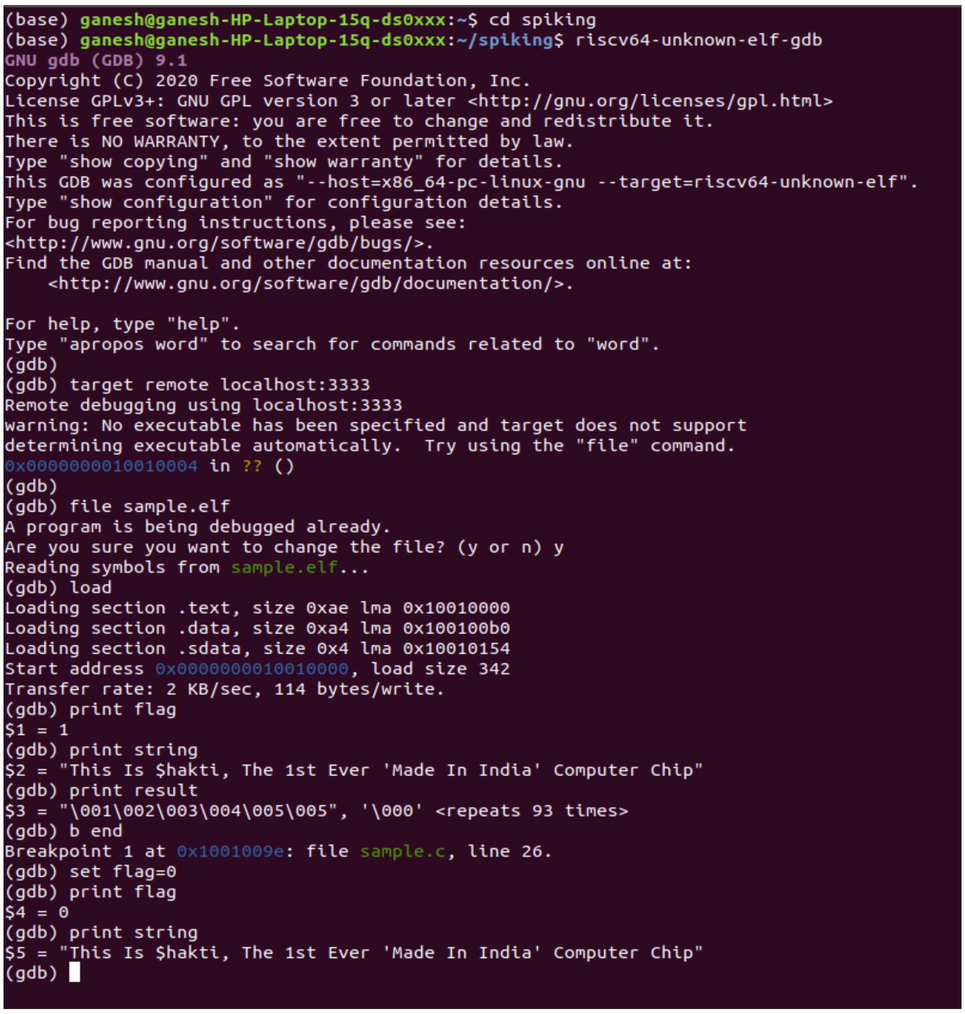
5.2.2 Output of the C code


ganesh@ganesh-HP-Laptop-15q-ds0xxx: ~/spiking
File Edit View Search Terminal Help

```
(base) ganesh@ganesh-HP-Laptop-15q-ds0xxx:~$ cd spiking
(base) ganesh@ganesh-HP-Laptop-15q-ds0xxx:~/spiking$ gedit hello.c
(base) ganesh@ganesh-HP-Laptop-15q-ds0xxx:~/spiking$ riscv64-unknown-elf-gcc hel
lo.c -o hello.elf
(base) ganesh@ganesh-HP-Laptop-15q-ds0xxx:~/spiking$ spike $(which pk) hello.elf
bbl loader
Hello world
(base) ganesh@ganesh-HP-Laptop-15q-ds0xxx:~/spiking$ 
```

Figure 5. 6 Output of the simple C code

5.2.3 Debugging of C code using GDB

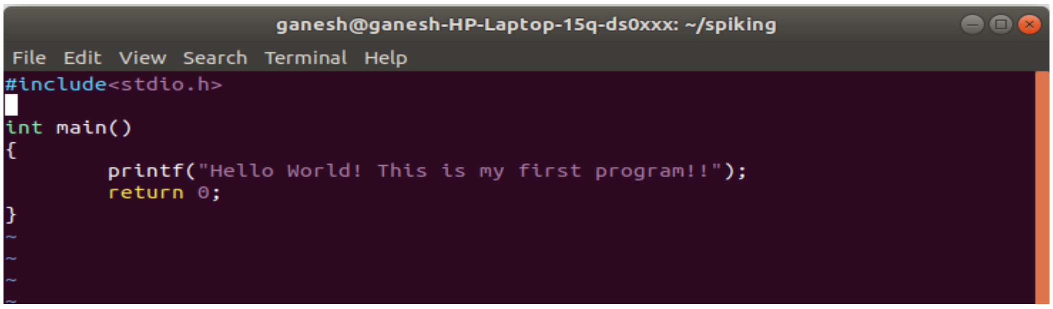


```
(base) ganesh@ganesh-HP-Laptop-15q-ds0xxx:~$ cd spiking
(base) ganesh@ganesh-HP-Laptop-15q-ds0xxx:~/spiking$ riscv64-unknown-elf-gdb
GNU gdb (GDB) 9.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=riscv64-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x0000000010010004 in ?? ()
(gdb)
(gdb) file sample.elf
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Reading symbols from sample.elf...
(gdb) load
Loading section .text, size 0xae lma 0x10010000
Loading section .data, size 0xa4 lma 0x100100b0
Loading section .sdata, size 0x4 lma 0x10010154
Start address 0x0000000010010000, load size 342
Transfer rate: 2 KB/sec, 114 bytes/write.
(gdb) print flag
$1 = 1
(gdb) print string
$2 = "This Is $hakti, The 1st Ever 'Made In India' Computer Chip"
(gdb) print result
$3 = "\001\002\003\004\005\005", '\000' <repeats 93 times>
(gdb) b end
Breakpoint 1 at 0x1001009e: file sample.c, line 26.
(gdb) set flag=0
(gdb) print flag
$4 = 0
(gdb) print string
$5 = "This Is $hakti, The 1st Ever 'Made In India' Computer Chip"
(gdb) 
```

Figure 5. 7 Debugging of a C code using GDB

5.3 C Class on Verilator

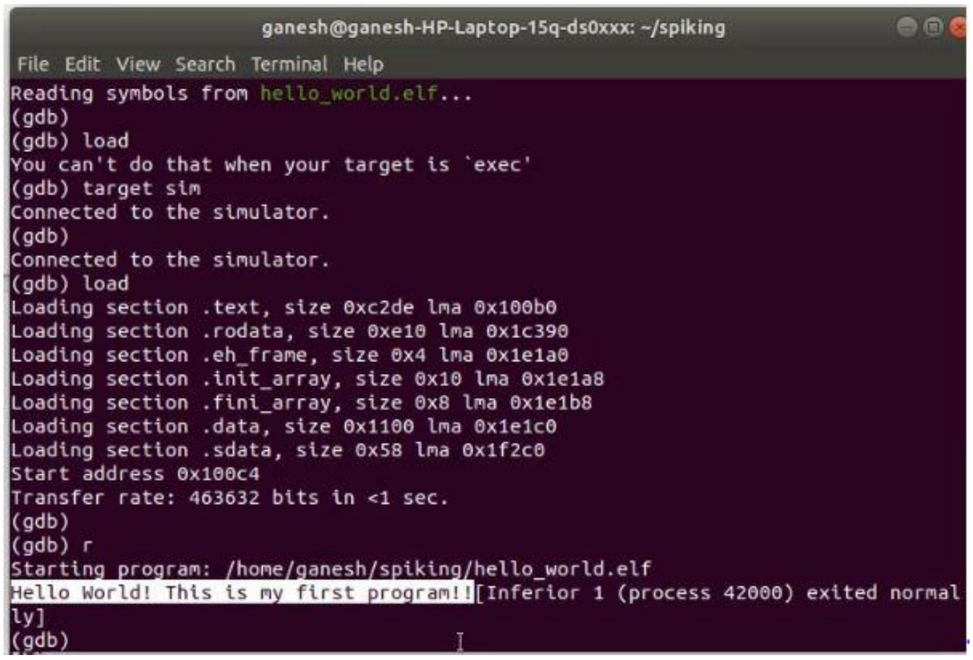
5.3.1 C code


ganesh@ganesh-HP-Laptop-15q-ds0xxx: ~/spiking
File Edit View Search Terminal Help

```
#include<stdio.h>

int main()
{
        printf("Hello World! This is my first program!!");
        return 0;
}
~
~
~
~
```

Figure 5. 8 Simple C code to debug using GDB

5.3.2 Output of the C code


ganesh@ganesh-HP-Laptop-15q-ds0xxx: ~/spiking
File Edit View Search Terminal Help

```
Reading symbols from hello_world.elf...
(gdb)
(gdb) load
You can't do that when your target is `exec'
(gdb) target sim
Connected to the simulator.
(gdb)
Connected to the simulator.
(gdb) load
Loading section .text, size 0xc2de lma 0x100b0
Loading section .rodata, size 0xe10 lma 0x1c390
Loading section .eh_frame, size 0x4 lma 0x1e1a0
Loading section .init_array, size 0x10 lma 0x1e1a8
Loading section .fini_array, size 0x8 lma 0x1e1b8
Loading section .data, size 0x1100 lma 0x1e1c0
Loading section .sdata, size 0x58 lma 0x1f2c0
Start address 0x100c4
Transfer rate: 463632 bits in <1 sec.
(gdb)
(gdb) r
Starting program: /home/ganesh/spiking/hello_world.elf
Hello World! This is my first program!![Inferior 1 (process 42000) exited normal
ly]
(gdb)
```

Figure 5. 9 Output on GDB debugger