THEJAS32 Programmers Manual

Release 1.0.0

HDG, CDAC, Trivandrum

Jul 26, 2023

CONTENTS

This Programmer's Manual provides an overview of the RISC-V THEJAS32 SoC, covering the architecture overview, instruction set, memory organization, registers, exceptions, interrupts, programming model and development tools.

CHAPTER

ONE

THEJAS32 SOC PROGRAMMER'S MANUAL

1.1 Introduction

The THEJAS32 SoC (System-on-Chip) is a flexible and customizable platform designed for embedded systems. This programmer's manual aims to provide an overview of the architecture, instruction set, memory organization, programming model, and development tools for programming applications on an THEJAS32 SoC.

1.2 Architecture Overview

The The THEJAS32 SoC is based on RISC-V RV32IM architecture of the RISC-V Instruction Set Architecture (ISA). It stands for RV32 (32-bit address space) and IM (integer multiplication and division). The architecture supports a set of instructions, including basic integer arithmetic, logical operations, load/store instructions, control flow instructions, and privileged instructions for system-level operations.

An THEJAS32 SoC typically consists of a ET1031 Processor core, memory subsystem, input/output interfaces, and various peripherals tailored for specific applications.

1.2.1 Features

The THEJAS32 SoC provides the following features:

- 32-bit instruction and data formats
- 32 general-purpose registers (x0-x31)
- Machine Mode
- Interrupt and exception handling
- System control and status registers (CSRs)
- Integer multiplication and division operations

1.3 Instruction Set Architecture

The RV32IM architecture is a 32-bit RISC-V instruction set architecture that supports integer and multiplication/division operations. The architecture is divided into the following major components:

- Integer base instructions (RV32I)
- Integer multiplication and division extension (RV32M)

The THEJAS32 SoC supports a wide range of instructions, including but not limited to:

- Arithmetic and logical instructions (add, sub, and, or, xor, etc.)
- Load and store instructions (lw, sw, lb, sb, etc.)
- Branch and jump instructions (beq, bne, jal, jalr, etc.)
- Shift and comparison instructions (sll, srl, slt, etc.)
- Multiplication and division instructions (mul, div, etc.)

Refer to the RISC-V specification for a complete list of instructions supported by the RV32IM architecture.

CHAPTER

TWO

MEMORY ORGANIZATION

The memory organization in an THEJAS32 SoC typically consists of several memory regions, including instruction memory (often referred to as "text" segment), data memory (often referred to as "data" segment), stack memory, and peripheral memory regions.

The instruction memory stores executable code, and the data memory stores variables and data used by the program. The stack memory is used to store function call frames and local variables. The peripheral memory regions are used to interface with external devices and peripherals connected to the SoC.

The memory organization can vary depending on the specific THEJAS32 SoC implementation and the chosen memory map.

2.1 Memory Map

The THEJAS32 SoC's memory map is as follows:

ROM

Memory Address	Content	Size
10000	Boot firmware	32KB
17FFF	End of Memory	

IRAM

Memory Address	Content	Size
20000	User defined firmware	96KB
37000	Data segment for firmware	32KB
3FFFF	End of Memory	

Flash

Memory Address	Content	Size
000000	Reserved for Program Flashing	256KB
040000	User defined memory	1.75MB
200000	End of Memory	

CHAPTER

THREE

REGISTERS

The RV32IM architecture defines a set of 32 general-purpose registers (x0 to x31), which are 32 bits wide. These registers are used to store data and perform computations within the CPU core.

Name	Width	Number	Description
x0	32	0	Hard-wired zero
x1	32	1	Return address
x2	32	2	Stack pointer
x3	32	3	Global pointer
x4	32	4	Thread pointer
x5	32	5	Temporaries (1)
x6	32	6	Temporaries (2)
x7	32	7	Temporaries (3)
x8	32	8	Temporaries (4)
x9	32	9	Temporaries (5)
x10	32	10	Temporaries (6)
x11	32	11	Temporaries (7)
x12	32	12	Reserved for platform use
x13	32	13	Reserved for platform use
x14	32	14	Reserved for platform use
x15	32	15	Reserved for platform use
x16	32	16	Reserved for platform use
x17	32	17	Reserved for platform use
x18	32	18	Reserved for platform use
x19	32	19	Reserved for platform use
x20	32	20	Reserved for platform use
x21	32	21	Reserved for platform use
x22	32	22	Reserved for platform use
x23	32	23	Reserved for platform use
x24	32	24	Reserved for platform use
x25	32	25	Reserved for platform use
x26	32	26	Reserved for platform use
x27	32	27	Reserved for platform use
x28	32	28	Reserved for platform use
x29	32	29	Reserved for platform use
x30	32	30	Reserved for platform use
x31	32	31	Reserved for platform use

Additionally, there are special-purpose registers, including the program counter (PC), stack pointer (SP), and various control and status registers (CSRs) used for managing exceptions, interrupts, and other system-level operations.

The THEJAS32 SoC may also include additional registers for specific purposes, such as peripheral control and configuration.

CHAPTER

FOUR

SYSTEM CONTROL AND STATUS REGISTERS (CSRS)

The THEJAS32 SoC includes a set of control and status registers (CSRs) that allow software to control and monitor the processor's behavior. The CSRs provide access to various system configuration, status, and control registers, including:

- Machine status registers (mstatus, misa, etc.)
- Machine trap handling registers (mtvec, mcause, mie, etc.)
- Machine memory management registers (satp, etc.)

Refer to the RISC-V specification for a complete list of CSRs supported by the RV32IM architecture.

CHAPTER

FIVE

EXCEPTIONS AND INTERRUPTS

The RV32IM architecture supports exceptions and interrupts, which are used to handle exceptional events and asynchronous events, respectively. Exceptions can occur due to events like illegal instructions, divide-by-zero, and page faults, while interrupts are typically generated by external devices or timers.

When an exception or interrupt occurs, the CPU core saves the current context, including the program counter and registers, and transfers control to the exception or interrupt handler. After handling the exception or interrupt, the core resumes execution from the saved context.

CHAPTER

SIX

PROGRAMMING MODEL

The programming model for an RV32IM-based SoC follows the general principles of RISC-V programming. Programs are typically written in assembly language or compiled from high-level languages using a RISC-V toolchain.

To develop software for an THEJAS32 SoC, you need to understand the instruction set architecture, memory organization, and register usage. You also need to be familiar with the available development tools, such as assemblers, compilers, linkers, and debuggers, to build and debug your applications.

The specific programming model details, including calling conventions, stack usage, and system-level operations, may vary depending on the RV32IM implementation and the chosen software development environment.

6.1 Hello World Program

To create a “Hello, World!” C program to display text in a UART terminal on THEJAS32 SoC, you’ll need a few components: the C code, the linker script (lds), the C runtime startup code (crt.S), and a Makefile to build the project. Below are the steps to achieve this:

1.Write the “Hello, World!” C program (main.c):

```c
/*Function to print the string using UART Peripheral*/
void print_string(const char *str)
{
    // Address of UART TX register
    volatile char *tx = (volatile char *)0x10000100;
    // Address of UART LSR register
    volatile char *lsr = (volatile char *)0x10000114;
    while (*str)
    {
        *tx = *str; //Tranxmit a single char
        while ((*lsr& 0x20) != 0x20); //Check LSR for TX complete
        str++;
    }
}

/*Main program*/
void main() {
    const char *message = " Hello, World!\n";

```

(continues on next page)

(continued from previous page)

```
    print_string(message);

        while(1);/*Infinite Loop*/
}
```

2. Write the linker script (linker.lds):

```
OUTPUT_ARCH(riscv)
ENTRY(_start)

MEMORY
{
  ram (rwx) : ORIGIN = 0x200000, LENGTH = 250K
}

SECTIONS
{
  . = 0x200000;                      /*Start Address*/
  .text.init : { *(.text.init) }  /*Init code*/
  .text : { *(.text) }               /*Main Program*/
  .data : { *(.data) }               /*Data Section*/
  _end=.;                            /*End of section*/
}
```

3. Write the C runtime startup code (crt.S):

```
  .section ".text.init"
  .globl _start

  #init all registers with zero
_start:
  li  x1, 0
  li  x2, 0
  li  x3, 0
  li  x4, 0
  li  x5, 0
  li  x6, 0
  li  x7, 0
  li  x8, 0
  li  x9, 0
  li  x10,0
  li  x11,0
  li  x12,0
  li  x13,0
  li  x14,0
  li  x15,0
  li  x16,0
  li  x17,0
  li  x18,0
  li  x19,0
```

(continues on next page)

(continued from previous page)

```
  li  x20,0
  li  x21,0
  li  x22,0
  li  x23,0
  li  x24,0
  li  x25,0
  li  x26,0
  li  x27,0
  li  x28,0
  li  x29,0
  li  x30,0
  li  x31,0

#set the stack pointer as the end+1K
  la  tp, _end
  add tp, tp, 1024
  add sp, sp, tp

#jump to main
  j main
```

4. Write the Makefile (Makefile):

```
CC=riscv64-vega-elf-gcc
OC=riscv64-vega-elf-objcopy

CFLAGS=-march=rv32im -mabi=ilp32 -Os -nostdlib -nostartfiles

all: hello.elf

hello.elf: main.c crt.S linker.lds
        $(CC) $(CFLAGS) -T linker.lds -o hello.elf crt.S main.c
        ${OC} hello.elf hello.bin -O binary
clean:
        rm -f hello.elf hello.bin
```

5. Install any riscv toolchain that supports ‘rv32im’ extension, here we are using vega-tools(Prebuilt toolchain available in VEGA Gitlab repository)

```
$ git clone https://gitlab.com/cdac-vega/vega-tools.git
```

6. Export the toolchain PATH in your terminal, or save it in your .bashrc file

```
$ export PATH=$PATH:/**<replace with your toolchain path>**/vega-tools/
↪toolchain/bin/
```

7. Build the project: Open a terminal in the directory containing the main.c, crt.s, linker.lds, and Makefile files. Then run the ‘make’ command to build the project:

```
$ make
```

(continues on next page)

(continued from previous page)

```
riscv64-vega-elf-gcc -march=rv32im -mabi=ilp32 -Os -nostdlib -nostartfiles -T␣
↪linker.lds -o hello.elf crt.S main.c
riscv64-vega-elf-objcopy hello.elf hello.bin -O binary
```

8. Transfer the 'hello.bin' file to any of the Aries Development boards using xmodem file transfer, Press Enter after file transfer, You can see the output in terminal

```
+-----------------------------------------------------------------------------
↪+
|           VEGA Series of Microprocessors Developed By C-DAC, INDIA          ␣
↪|
|     Microprocessor Development Programme, Funded by MeitY, Govt. of India   ␣
↪|
+-----------------------------------------------------------------------------
↪+
| Bootloader, ver 1.0.0 [  (hdg@cdac_tvm) Tue Dec  15 16:50:32 IST 2020 #135]␣
↪/
|                                                                             ␣
↪|
|  ___    _________________________          ISA   : RISC-V [RV32IM]          ␣
↪|
|  __ |  / /__  ____/_  ____/__    |                                          ␣
↪|
|  __ | / /__  __/  _  / __ __  /| |         CPU   : VEGA ET1031              ␣
↪|
|  __ |/ / _  /___  / /_/ / _  ___ |                                          ␣
↪|
|  _____/  /_____/  \____/  /_/  |_|         SoC   : THEJAS32                 ␣
↪|
+---------------------------------------+-------------------------------------
↪+
|         www.vegaprocessors.in         |             vega@cdac.in            ␣
↪|
+---------------------------------------+-------------------------------------
↪+

Transfer mode   : UART XMODEM

IRAM            : [0x200000 - 0x23E7FF] [250 KB]

Please send file using XMODEM and then press ENTER key.
CCCCCCCCC
Starting program ...




Hello, World!
```

Note:

1. To clean the project you can type 'make clean' command

```
$ make clean
rm -f hello.elf hello.bin
```

2. Board may need to reset before you try to transfer a new program. You can see the 'CCC' is displayed on the terminal for the xmodem handshake.

CHAPTER

SEVEN

DEVELOPMENT TOOLS

To develop software for an THEJAS32 SoC, you can use a range of development tools, including:

Assembler: Converts assembly code into machine code.

Compiler: Translates high-level programming languages into assembly or machine code.

Linker: Combines object files and libraries to create an executable program.

VEGA SDK: Enable developers to assist in creating applications for THEJAS32 platform. SDKs are designed to simplify and streamline the development process by offering pre-built components, APIs (Application Programming Interfaces), and examples that enable developers to interact with and utilize the features of the THEJAS32 platform.

Arduino IDE: is a widely-used platform for prototyping and developing projects with microcontrollers, VEGA BSP is available for arduino.

Integrated Development Environments (IDEs): Provide a comprehensive set of tools for code editing, building, debugging, and project management, streamlining the development process. Depending on your preferences and requirements, you can choose from various open-source and commercial development tools available for the RISC-V ecosystem.

CHAPTER EIGHT

CONCLUSION

This Programmer's Manual provides an overview of the RISC-V THEJAS32 SoC, covering the architecture overview, instruction set, memory organization, registers, exceptions, interrupts, programming model and development tools.

By understanding these key aspects of RV32IM-based SoC programming, you'll be equipped to develop software applications for embedded systems using the RISC-V RV32IM architecture.

Remember to refer to the official RISC-V specification, the documentation provided by the specific THEJAS32 SoC implementation, and the available development tools' documentation for detailed and up-to-date information when working on a particular project.