# THEJAS32

## Boot-up

On PoR, the boot-loader starts executing and the BOOTSEL jumper selects where to execute the code from

| Jumper Position | BOOTSEL | Boot Mode |
|---|---|---|
| Open | 0 | UART XMODEM |
| Closed | 1 | Boot from SPI Flash |

- When **BOOTSEL=0**, the bootloader waits for the user to upload the executable code (.bin format) over UART. The downloaded code is copied to RAM and code execution begins.

```
+--------------------------------------------------------------------------+
 |           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.
 CCCCCCCCCCCCC
```

To upload code in this mode, press `Ctrl+A S` to send file over serial, select xmodem and navigate to the program binary location.

```
+-[Upload]--+
| zmodem    |
| ymodem    |
▷xmodem<    |
| kermit    |
| ascii     |
+-----------+
```

---

- When **BOOTSEL=1**, the bootloader copies the code from external SPI flash memory to the SRAM. Flashing to this flash memory is achieved using `xmodemflasher` tool provided by CDAC.

The program `xmodemflasher` takes 2 arguments :

`xmodemflasher <device> <executable_binary>`

Example : `xmodemflasher /dev/ttyUSB0 build/app.bin`

Here's the console log when BOOTSEL=1 :

```
+-------------------------------------------------------------------------+
 |           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            |
 +-------------------------------------+-----------------------------------+

 Copying from FLASH to IRAM

 [INFO] Flash ID: 1f:86:01 Flash initialized
 [INFO] Copying  250KB from address: 0x000000.

 Starting program ...
```

---

# GPIO

Thejas32 has two GPIO controllers - GPIOA, GPIOB - each of them 16 bit wide.

GPIO is accessible with two registers :

- DIR (Direction Register)
- DATA (Data Register)

`GPIOA_DIR` - 0x100C0000

`GPIOB_DIR` - 0x101C0000

The DATA register for each GPIO appears at 16 locations in memory, according to the Pin number selected.

The PADDR register needs to be set/reset at appropriate location and bit for writing to GPIO.

`GPIOA_BASE` = 0x10080000

`GPIOB_BASE` = 0x10180000

Suppose you wish to write logical *LOW* to GPIOA Pin 5 :

We define address `PADDR` as `( (1 << pin) << 2 )`. `PADDR` acts as a mask for `GPIO_DATA`

```
PADDR = (1 << 5) << 2
```

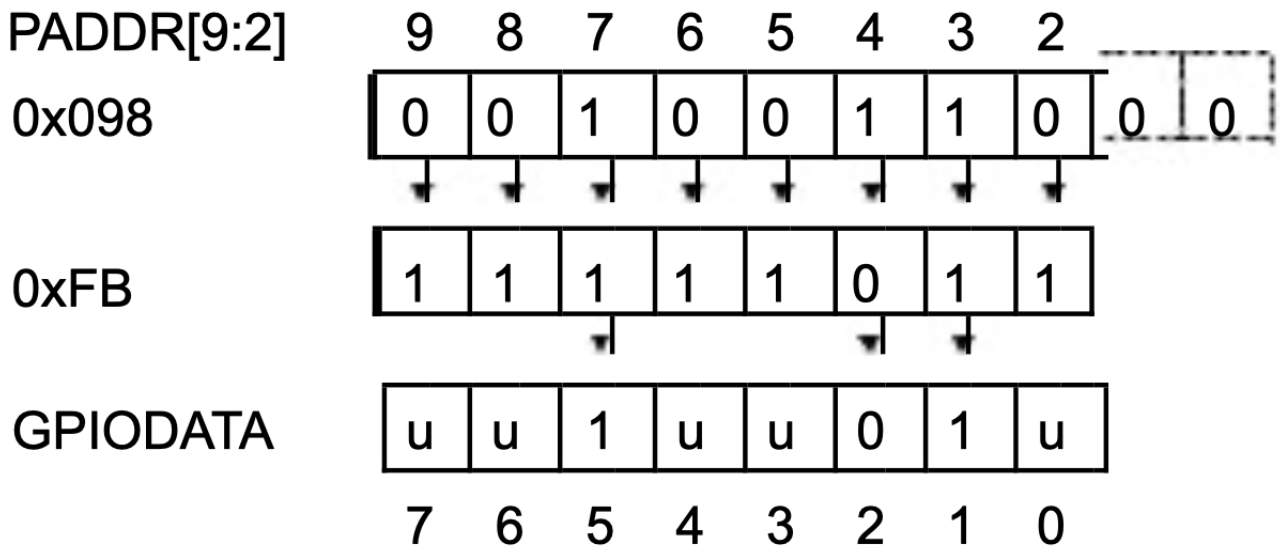
Then the GPIO DATA registers occurs at address :

```
GPIO_DATA = GPIOA_BASE + PADDR
```

Now write to address appropriate bit as per pin:

```
*GPIO_DATA = 0 << 5;
```

Here's a figure explaining the operation on a similar 8-bit implementation (ET1031 is a 16-bit implementation) :




*u = unchanged

Data `0xFB` is written to **DATA** register. Only the bits masked with 1 in **PADDR** get modified in GPIO_DATA (bits 5 and 2).

---

# Timer

Thejas32 has 3 timers, each 32-bit auto-reload down counter : TIMER0, TIMER1, TIMER2

The timers feature 2 modes :

## Free-running Mode

Counter starts with the value 0xFFFFFFFF and counts down to 0.
Alternatively, a value can be loaded into the TIMER_LOAD register can timer starts counting down from this value.

When the count reaches zero (0x00000000), an interrupt is generated and the counter wraps around to 0xFFFFFFFF irrespective of the value in the TIMER_LOAD register.

If the counter is disabled by clearing the TIMER_CTRL_EN bit in the Timer Control Register, the counter halts and holds its current value. If the counter is re-enabled again then the counter continues decrementing from the current value.

## Periodic Mode

An initial counter value can be loaded by writing to the `TIMER_LOAD` Register and the counter starts decrementing from this value if the counter is enabled.

The counter decrements each cycle and when the count reaches zero, `0x00000000`, an interrupt is generated and the counter reloads with the value in the TIMER_LOAD Register. The counter starts to decrement again and this whole cycle repeats for as long as the counter is enabled.

In both modes the end of timer count is signalled by an interrupt.

- When masked, the raw interrupt status can be read in `GLOBAL_TIMER_INT_STATUS`. The bits in this register correspond to Timer number, eg, 0 = TIMER0. So to check the raw interrupt status of TIMER2, check the bit 2 ((0x1 << 2) or 0x4).
- When unmasked, this interrupt status can be read in `TIMERx→ISR` (x = 0,1,2) at the 0th bit of register. The bit sets to 1 when the interrupt occurs.

### Prescalar

There is no clock tree diagram available, so my guess is as good as yours. We consider the CPU bus clock to be 100 MHz as advertised.

From experimentation with the timer :

- Case 1 : Timer initialized with interrupt masked ( `Timer_Init()` ), TIMER_LOAD = 50 for a 1us delay and (50*1000) for 1ms delay.
- Case 2 : Timer initialized with interrupt unmasked ( `Timer_Init_IT()` ), TIMER_LOAD = 100 for a 1us delay and (100*1000) for 1ms delay.

---

# Interrupts and Exceptions

These are RISC-V definitions :

- **Exception** : Used to refer to an unusual condition occurring at runtime (ie synchronous in nature) associated with an instruction in the current core. Eg : Illegal instruction
- **Interrupt** : Refers to an external asynchronous event that may cause a hart to experience unexpected transfer of control. Typically done by peripherals.

The transfer of control in both cases is called a **Trap**.

# Interrupt

Interrupts can only be **enabled/ disabled on a global basis** and not individually. There is an event/interrupt controller outside of the core that performs masking and buffering of the interrupt lines.

ET1031 **does support nested interrupt/exception handling**. Exceptions inside interrupt/exception handlers cause another exception, thus exceptions during the critical part of the exception handlers, i.e. before having saved the `MEPC` and `MSTATUS` register, will cause those register to be overwritten. Interrupts during interrupt/exception handlers are disabled by default, but can be explicitly enabled if desired.

Example for timer interrupt :

- Initialize timer in interrupt mode `Timer_Init_IT()`.
- Enable global interrupts `__enable_irq()`
- Enable timer interrupt in PLIC (Platform-level Interrupt Controller) with `PLIC_Enable()`, which takes the interrupt number as argument.
- Define the IRQHandler routine. Make sure to clear the timer interrupt with `Timer_ClearInterrupt()` before return.
- Start Timer