THEJAS32

Thejas32 uses the ET1031 CPU core based on RISC-V ISA. The salient features include :

- RISC-V `rv32im` ISA, including CSR and fence instructions.
- 100 MHz clock frequency
- 250 KB RAM
- 2 x 16-bit GPIOs
- 3 x 32-bit Timers
- 3 x UARTs
- 4 x SPIs
- 3 x I2Cs
- 8 x PWM Channels

Memory Map

Peripheral Register	Start	End	Interrupt Number
RAM	0x00200000	0x00237FFF	-
UART0	0x10000100	0x100001FF	0
UART1	0x10000200	0x100002FF	1
UART2	0x10000300	0x100003FF	2
SPI0	0x10000600	0x100006FF	3
SPI1	0x10000700	0x100007FF	4
I2C0	0x10000800	0x100008FF	5
I2C1	0x10000900	0x100009FF	6
TIMER0	0x10000A00	0x10000A10	7
TIMER1	0x10000A14	0x10000A24	8
TIMER2	0x10000A28	0x10000A38	9
TIMERS INTERRUPT STATUS	0x10000AA0		
TIMERS INTERRUPT CLEAR	0x10000AA4		
TIMERS RAW INTERRUPT STATUS	0x10000AA8		
I2C2	0x10001000	0x10001FFF	22
GPIOA	0x10080000		10 - 21
GPIOB	0x10180000		
SPI2	0x10200100	0x102001FF	23
SPI3	0x10200200	0x102002FF	
PWM	0x10400000	0x104000FF	24 - 31
PLIC	0x20010000	0x2001FFFF	

Boot-up

On PoR, the boot-loader in the on-chip rom 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 boot-loader waits for the user to upload the executable code (.bin format) over UART. The downloaded code is copied to RAM and code execution begins.

```
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 using `minicom`, 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.

Here's the console log when BOOTSEL=1 :

```
Copying from FLASH to IRAM

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

Starting program ...
```

NOTE : In this mode, 8KB of RAM is reserved for another boot-loader present on flash.

All examples in this repo assume `BOOTSEL = 1`. Define `BOOTSEL = 0` in your Makefile to compile for boot-mode 0.

GPIO

The GPIO controller on Thejas32 is based on ARM PL061 and features two 16-bit wide GPIO instances.

GPIOs are controlled using 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 = (1 << 5) << 2
```

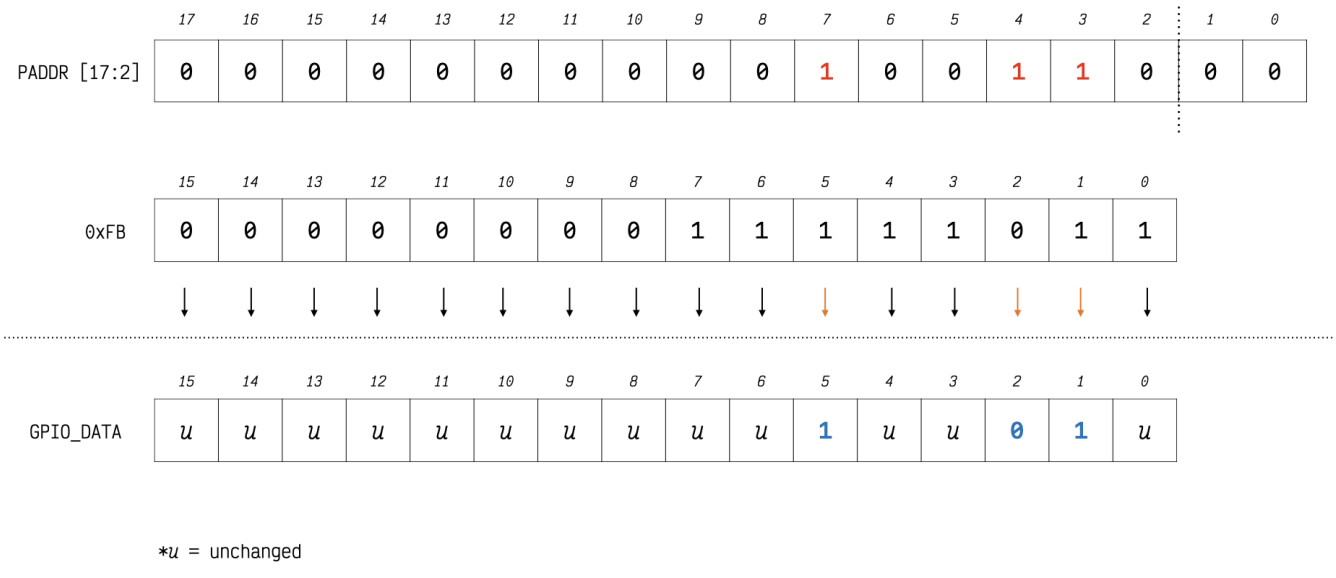
Then the GPIO_DATA register occurs at address :

```
GPIO_DATA = GPIOA_BASE + PADDR
```

Now write desired data to this address at desired pin bit:

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

Here's an illustration explaining the operation by writing `0xFB`



Data `0xFB` is written to **GPIO_DATA** register. Only the bits set to 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, on loading a value into the `TIMER_LOAD` register, 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 setting a bit in the timer's local interrupt status register.

- When timer interrupt is masked, the raw interrupt status can be read in `TIMERS_RAWISR`. The bits in this register correspond to Timer number, eg, 0 $\rightarrow$ `TIMER0`. So to check the raw interrupt status of `TIMER2`, check if bit 2 is set to 1.
- When unmasked, this interrupt status can be read in `TIMERx→ISR` (x = 0,1,2) at the 0th bit of register. The bit is set to 1 when the interrupt occurs.

Load Count

At a $100\ MHz$ clock frequency, the timer counter decrements by 1 every $10\ ns$ $\left(\frac{1}{100*10^6}\right)$

For example, to achieve a period of $1\ \mu s$

$10\ ns \quad \rightarrow \quad 1\ tick$

So, $1\ \mu s \ \rightarrow \frac{1\mu s}{10ns} \ \rightarrow \ \frac{10^{-6}}{10*10^{-9}} \quad ticks$

$\therefore 1\ \mu s \quad \rightarrow \ 100 \quad ticks$

So load a value of 100 for $1\ \mu s$ time period.

Interrupts and Exceptions

While Thejas32 does not have standard name for the interrupt controller used, all interrupts are handled globally by one single controller, so henceforth, the interrupt controller would be referred to as **PLIC** (Platform-Level Interrupt Controller).

ET1031 **does NOT support nested interrupt/exception handling**. Exceptions inside interrupt/exception handlers cause another exception, thus exceptions during the critical part of the exception handlers, will cause those registers to be overwritten. Interrupts during interrupt/exception handlers are disabled by default, but can be explicitly enabled if desired.

Terminologies

The following are derived from official RISC-V ISA manual:

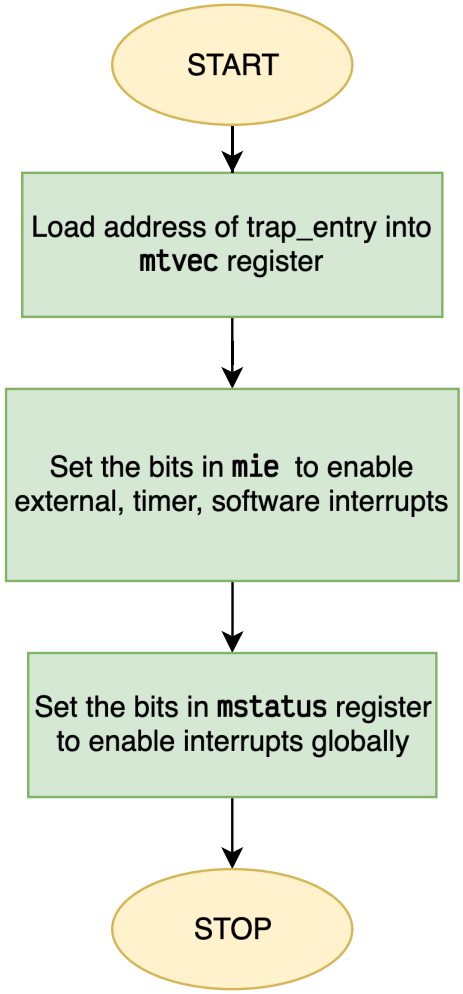
- **Exception** : Used to refer to an unusual condition occurring at runtime (ie *synchronous* in nature) associated with an instruction in the current core. Eg : illegal instruction, division-by-zero etc.
- **Interrupt** : Refers to an external *asynchronous* event that may cause a hart to experience unexpected transfer of control. Typically triggered by peripherals.
- **Trap** : The transfer of control to a trap handler caused by either an exception or an interrupt.

A subset of CSRs (**C**ontrol and **S**tatus **R**egisters) assist in handling traps. The registers are discussed briefly here, for detailed information, consult RISC-V ISA manual vol2 (Privileged).

- `mtvec` (Machine Trap-Vector base-adress register) - Holds the trap vector configuration. Prominently the base address of the trap vector.
- `mie` (Machine Interrupt Enable) - Used to enable machine-mode interrupts. These are classified as :
  - External interrupts
  - Timer interrupts
  - Software interrupts (exceptions)
- `mstatus` (Machine Status) - Keeps track of and control the CPU's current operating status. Setting `MIE` bit in `mstatus` registers enables machine-mode interrupts globally.
- `mepc` (Machine Exception Program Counter) - Holds the PC (program counter) value when an interrupt/exception is encountered.
- `mcause` (Machine Cause Register) - The highest bit (XLEN-1) indicates whether the trap was cause by exception(0) or interrupt(1). The rest of the bits contain exception code.

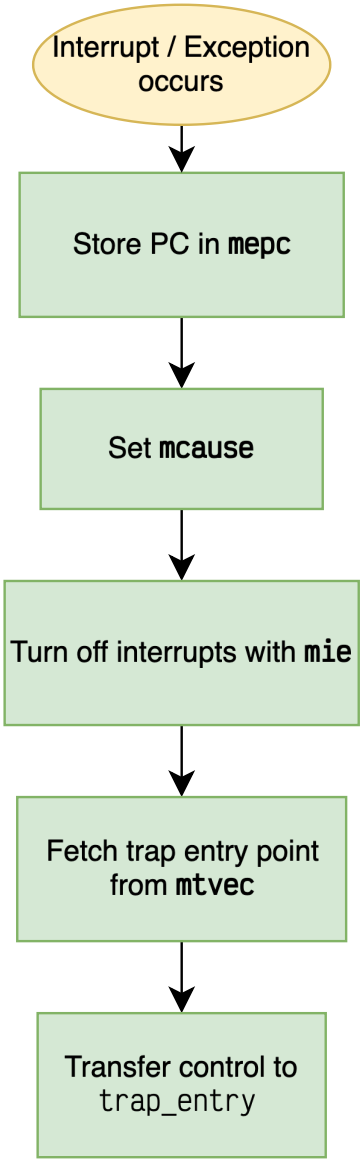
Setting trap entry

This is implemented in software.



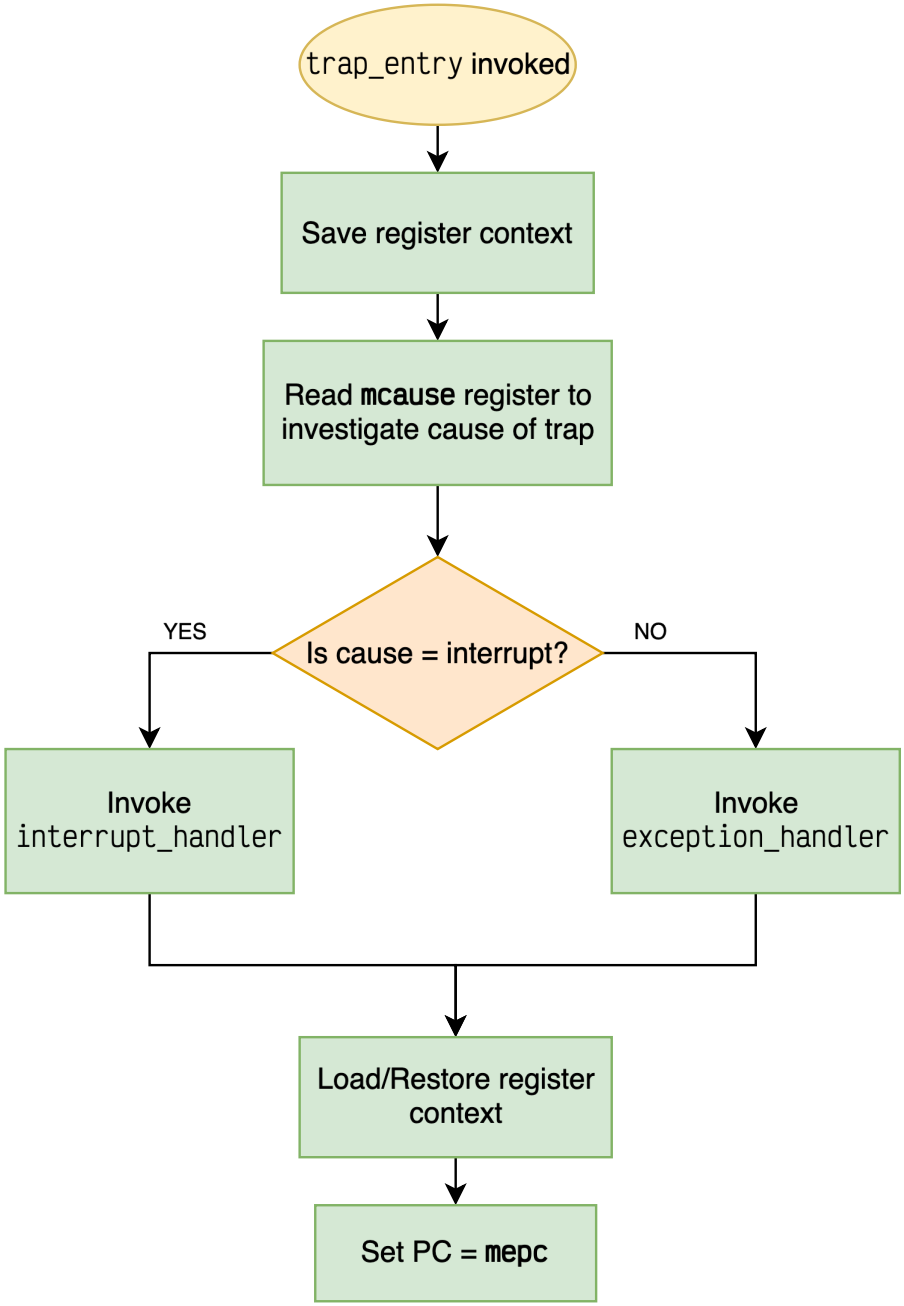
Interrupt Arrival

Performed by hardware.



Handling interrupts

Implemented in software.



More resources

- Blog by Muller Lee
- RISC-V ISA Manual Vol-2 (Privileged)
- Youtube : John's Basement Introduction to RV32I Interrupts and Traps
- Youtube : Robert Baruch - Interrupts and Exceptions