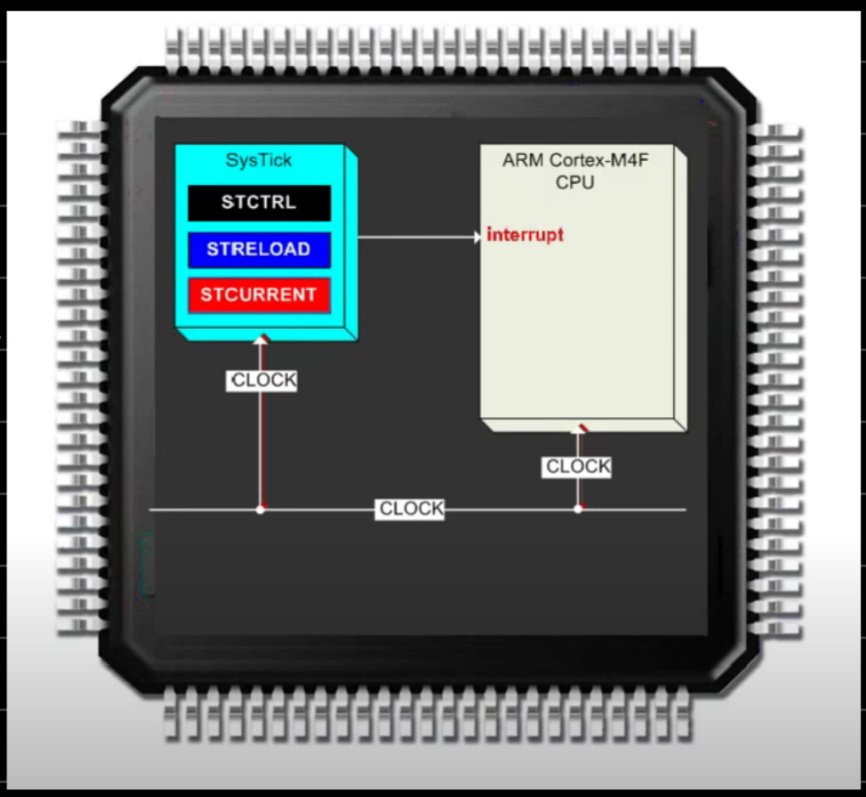
What are interrupts and how do they work

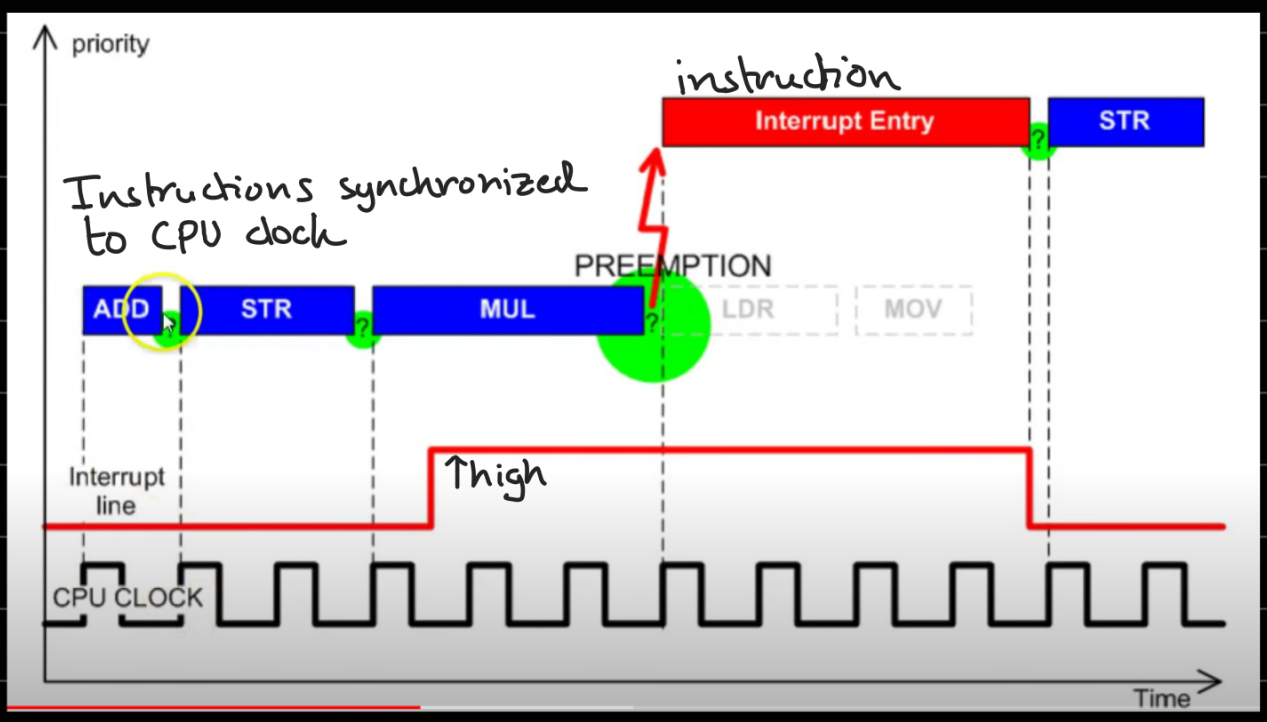
During the delay function in our blinky program, the CPU is always "polling" or waiting for the iteration counter in our delay loop to fall to 0.

Unfortunately our delay function polling is quite an inefficient method. The answer to this is:

Interrupts ⟶ TIVA C handles and times interrupts using system time hardware (SysTick in datasheet)



Systick has 3 registers
- ⟶ STcurrent register (SysTick→VAL in datasheet) is a 24bit down counter. Generates an alarm to CPU when value reaches 0.
- ⟶ STRELOAD register (SysTick→Load) reloads the STcurrent register when it reaches 0.
- ⟶ STCTRL to control Systick and STcurrent



ARM CPU uses special htw to sample interrupt line. As long as interrupt is low, CPU fetches next instruction in pipeline. If interrupt line is high then CPU executes interrupt entry (pre-emption)

Going back to our basic LED blinking code, we can remove the line that clears the output data register to turn off the LED. We do this by using the XOR operator (^=) to toggle our LED on and off. In the while(1) loop, if LED was previously on then:

previous bit to LED pin

XOR

R_LED position bit used for toggling (this bit is constant)

=

output bit to LED (toggled value)

```
while(1)
{
  // Use of new XOR operator (^=) to toggle LED on/off
  GPIOF_HS->DATA_Bits[0x02] ^= R_LED;

  delay(500000);
}
```

For interrupts, start by setting up the systick registers. These are found in the header file "core_cm4.h":

```
*/
typedef struct
{
  __IOM uint32_t CTRL;                   /*!< Offset: 0x000 (R/W)  SysTick Control and Status Register */
  __IOM uint32_t LOAD;                   /*!< Offset: 0x004 (R/W)  SysTick Reload Value Register */
  __IOM uint32_t VAL;                    /*!< Offset: 0x008 (R/W)  SysTick Current Value Register */
  __IM  uint32_t CALIB;                  /*!< Offset: 0x00C (R/ )  SysTick Calibration Register */
} SysTick_Type;

/* SysTick Control / Status Register Definitions */
#define SysTick_CTRL_COUNTFLAG_Pos          16U                                        /*!< SysTick CTRL: COUNTFLAG Position */
#define SysTick_CTRL_COUNTFLAG_Msk          (1UL << SysTick_CTRL_COUNTFLAG_Pos)        /*!< SysTick CTRL: COUNTFLAG Mask */

#define SysTick_CTRL_CLKSOURCE_Pos           2U                                        /*!< SysTick CTRL: CLKSOURCE Position */
#define SysTick_CTRL_CLKSOURCE_Msk          (1UL << SysTick_CTRL_CLKSOURCE_Pos)        /*!< SysTick CTRL: CLKSOURCE Mask */

#define SysTick_CTRL_TICKINT_Pos             1U                                        /*!< SysTick CTRL: TICKINT Position */
#define SysTick_CTRL_TICKINT_Msk            (1UL << SysTick_CTRL_TICKINT_Pos)          /*!< SysTick CTRL: TICKINT Mask */

#define SysTick_CTRL_ENABLE_Pos              0U                                        /*!< SysTick CTRL: ENABLE Position */
#define SysTick_CTRL_ENABLE_Msk             (1UL /*<< SysTick_CTRL_ENABLE_Pos*/)       /*!< SysTick CTRL: ENABLE Mask */

/* SysTick Reload Register Definitions */
#define SysTick_LOAD_RELOAD_Pos              0U                                        /*!< SysTick LOAD: RELOAD Position */
#define SysTick_LOAD_RELOAD_Msk             (0xFFFFFFUL /*<< SysTick_LOAD_RELOAD_Pos*/)  /*!< SysTick LOAD: RELOAD Mask */

/* SysTick Current Register Definitions */
#define SysTick_VAL_CURRENT_Pos              0U                                        /*!< SysTick VAL: CURRENT Position */
#define SysTick_VAL_CURRENT_Msk             (0xFFFFFFUL /*<< SysTick_VAL_CURRENT_Pos*/)  /*!< SysTick VAL: CURRENT Mask */

/* SysTick Calibration Register Definitions */
#define SysTick_CALIB_NOREF_Pos             31U                                        /*!< SysTick CALIB: NOREF Position */
```

① Enable SysTick and set 1's to interrupt, clock source and enable fields in SysTick→CTRL register

② SysTick→Val is a clear on write register, so it will always be cleared no matter what we write to it

③ For SysTick load register we need to know speed of CPU clock cycle (16MHz), we need to define this speed in our code by → #define SYSCLOCK_HZ 16000000U. Then we load Systick→load with $\frac{sysclock\_Hz - 1}{2}$ value

Once registers are set up, it will interrupt CPU every half a second. This will call the systick handler of the vector table. We therefore need to set this function up as well (contained in the bsp.c file). Thus we can move our code that toggles the LED → GPIOF_HS → Data_Bits ^= R_LED. The delay.c and delay.h files become obsolete then.

```c
#ifndef __BSP_H__
#define __BSP_H__

// #define statements moved to header file
#define R_LED (1U << 1)
#define B_LED (1U << 2)
#define G_LED (1U << 3)

#define SYSCLOCK_HZ 16000000U   // Default clock speed of TIVA

#endif // __DELAY_H__
```

in.c * | tm4c_cmsis.h | Startup_tm4c.c | **bsp.c** × | core_cm4.h | bsp.h

```c
/* Board Support Package */

#include "tm4c_cmsis.h"
#include "bsp.h"

__stackless void assert_failed (char const *file, int line)
{
  /* This code is modified by user according to project requirements */

  NVIC_SystemReset(); //System reset by TMSIS
}

// Systick handler functions for when interrupts happen
void SysTick_Handler(void)
{
  // Use of new XOR operator (^=) to toggle LED on/off
    GPIOF_HS->DATA_Bits[0x02] ^= R_LED;
}
```

Now we need to enable interrupts by clearing the PRIMASK bit. We do this using the function:
"__iar_builtin_enable_interrupt()"

To see how SysTick_Handler now works, set a breakpoint in the function:

```c
// Systick handler functions for when interrupts happen
void SysTick_Handler(void)
{
  // Use of new XOR operator (^=) to toggle LED on/off
    GPIOF_HS->DATA_Bits[G_LED] ^= G_LED;
}
```

Every time this breakpoint is hit, the LED gets toggled.

Interrupts : How do CPU's handle interrupt