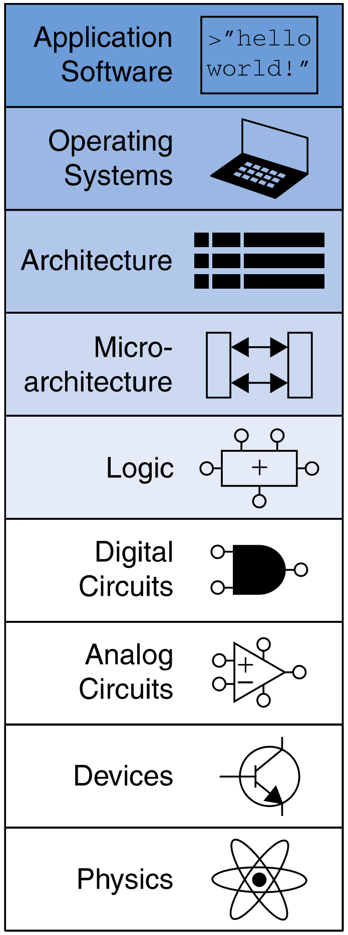
RISC-V Architecture & Processor Design

Week 14

IO and Peripherals

Topics

- Microcontrollers
- RISC-V Microcontrollers
- Memory-Mapped I/O
- General-Purpose I/O
- Device Drivers
- Delays & Timers
- Example: Morse Code
- Interfacing
- Serial Peripheral Interface
- Example: SPI Accelerometer



Microcontrollers

- **Microprocessors** are **computers on a chip**
- **Microcontrollers** are microprocessors with **flexible input/output (I/O) peripherals**
- **Peripheral examples:**
  - General-purpose I/O (GPIO): turn pins ON and OFF
  - Serial ports
  - Timers
  - Analog/Digital and Digital/Analog Converters (ADC/DAC)
  - Pulse Width Modulation (PWM)
  - Universal Serial Bus
  - Ethernet

Embedded Systems

- Microcontrollers are commonly used in embedded systems
- An embedded system is a system whose user may be unaware there is a computer inside
- **Examples:**
  - Microwave oven
  - Clock/radio
  - Electronic fuel injection
  - Annoying toys with batteries for toddlers
  - Implantable glucose monitor

Microprocessor Architectures

- The architecture is the native machine language a microprocessor understands.
- **Commercial Examples:**
  - RISC-V
  - ARM
  - PIC
  - 8051

RISC-V Microcontrollers

RISC-V Microcontrollers

- RISC-V is an **open-standard** architecture
  - Developed at Berkeley starting in 2010
  - No licensing fees
  - Increasingly popular, especially for system-on-chip
- **SiFive Freedom E310-G002**
  - Second generation RISC-V microcontroller (2019)
  - Found on several low-cost boards

SiFive Freedom E310 Microprocessor

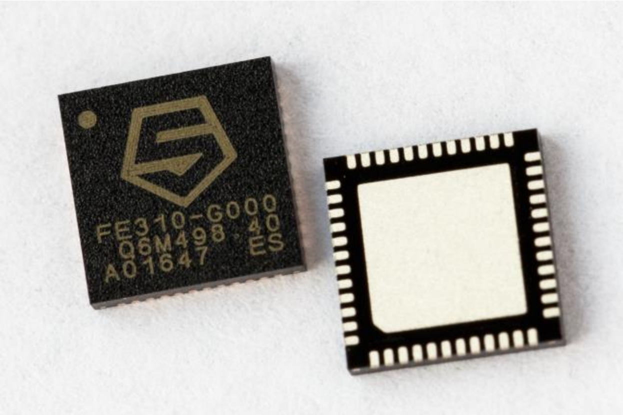
- E31 microprocessor with **5-stage pipeline**
  - Similar to the one we will discuss in class
  - RV32IMAC architecture
    - Baseline **32**-bit **R**ISC-**V** **I**nteger instructions
    - Plus **M**ultiply/Divide, **A**tomic Memory, **C**ompressed operations
  - 2.73 Coremark/MHz
- **Electrical Specs**
  - Up to 320 MHz
  - Built in antiquated 180 nm process
  - 1.8V core and 3.3V I/O

FE310 Memory & I/O

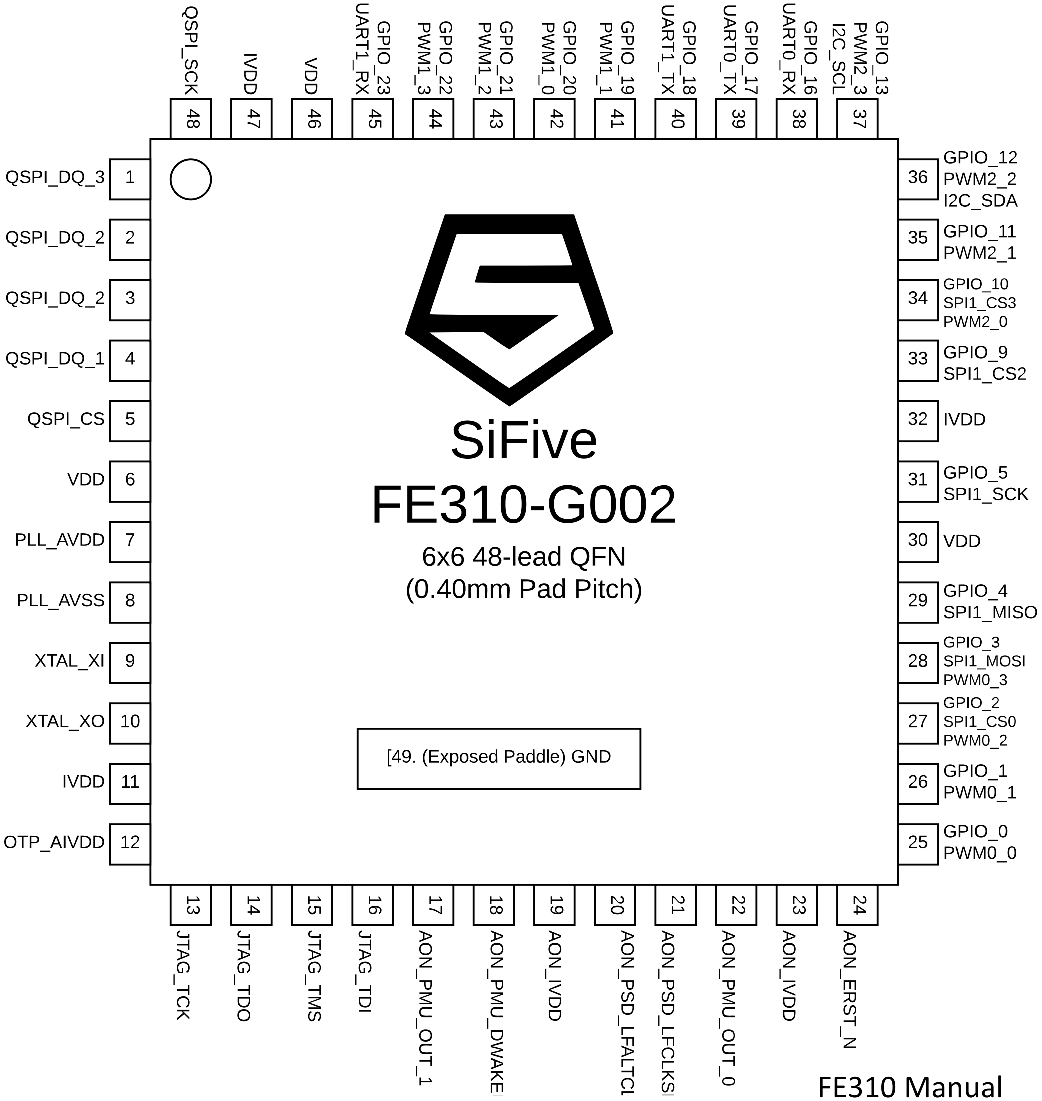
- **Onboard Memory**
  - 16 KB Data SRAM
  - 16 KB Instruction Cache
  - 8 KB Mask ROM & 8 KB OTP (one-time-programmable) Program Boot Memory
  - Most instructions located on external Flash
- **I/O Peripherals**
  - 19 GPIO Pins
  - Serial Ports: SPI x2, I2C, UART x2
  - PWM x3
  - Timer
  - JTAG Debugger Interface

FE310 Pinout

- **48-pin QFN** (quad flat pack, no leads)
  - Hard to hand solder
- **Pins:**
  - 12 power (GND pad on back)
  - 19 GPIO
  - 4 JTAG programming
  - 6 QSPI Flash code link
  - 2 clock crystals
  - 6 control



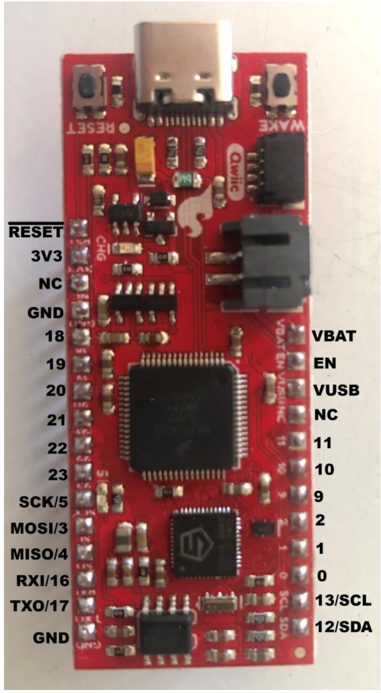
designnews.com



FE310 Manual

FE310 Boards

SparkFun RED-V Thing Plus

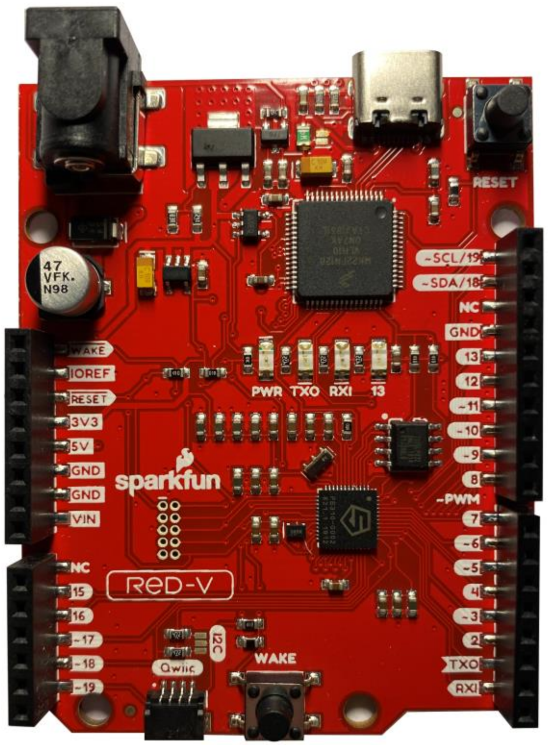


$30

Fits breadboard (2.3x0.9”)

Requires soldering header pins

SparkFun RED-V RedBoard



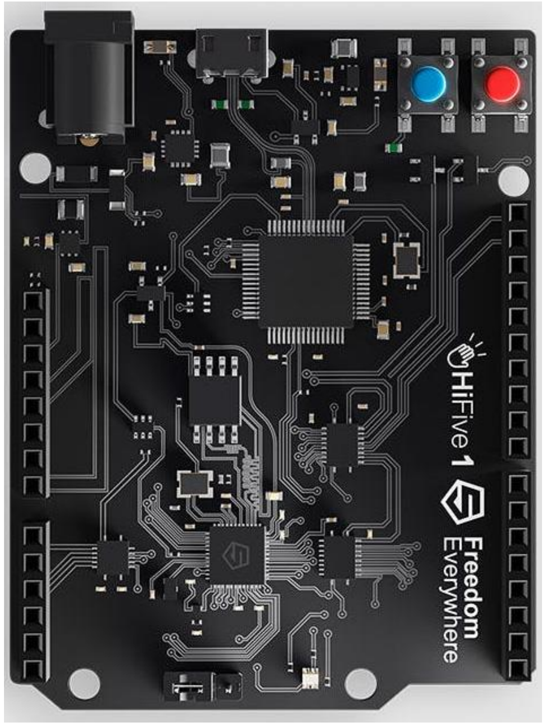
sparkfun.com

$40

Headers soldered

Nonstandard pin labels

HiFive 1



sifive.com

$60

WiFi & Bluetooth

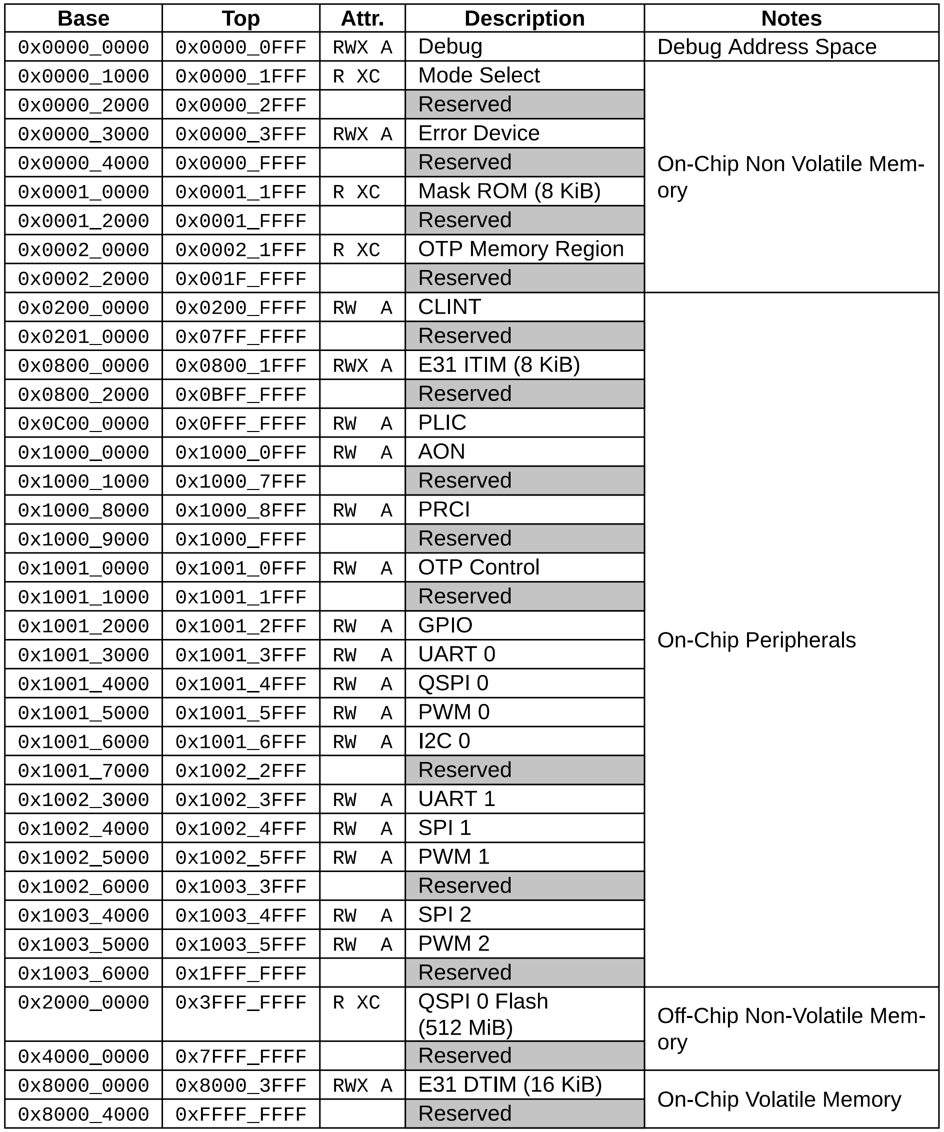
Memory-Mapped I/O

Memory-Mapped I/O

- Control peripherals by **reading or writing memory locations** called ***registers***
  - Not really the same as a bank of flip-flops
- Just like accessing any other variable, but these registers cause **physical things** to happen.
- Portion of the address space is **reserved for I/O** registers rather than program or data.
- In C, use **pointers** to specify addresses to read or write.

Example: FE310 Memory Map

- **Boot ROM:**
  - 0x00010000-1FFFF
- **Code Flash:**
  - 0x20000000-3FFFFFFF
- **Data SRAM:**
  - 0x80000000-80003FFF
- **Peripherals:**
  - 0x02000000-0x1FFFFFFF



Base	Top	Attr.	Description	Notes
0x0000_0000	0x0000_0FFF	RWX A	Debug	Debug Address Space
0x0000_1000	0x0000_1FFF	R XC	Mode Select	On-Chip Non Volatile Memory
0x0000_2000	0x0000_2FFF		Reserved	
0x0000_3000	0x0000_3FFF	RWX A	Error Device	
0x0000_4000	0x0000_FFFF		Reserved	
0x0001_0000	0x0001_1FFF	R XC	Mask ROM (8 KiB)	
0x0001_2000	0x0001_FFFF		Reserved	
0x0002_0000	0x0002_1FFF	R XC	OTP Memory Region	
0x0002_2000	0x001F_FFFF		Reserved	
0x0200_0000	0x0200_FFFF	RW A	CLINT	On-Chip Peripherals
0x0201_0000	0x07FF_FFFF		Reserved	
0x0800_0000	0x0800_1FFF	RWX A	E31 ITIM (8 KiB)	
0x0800_2000	0x0BFF_FFFF		Reserved	
0x0C00_0000	0x0FFF_FFFF	RW A	PLIC	
0x1000_0000	0x1000_0FFF	RW A	AON	
0x1000_1000	0x1000_7FFF		Reserved	
0x1000_8000	0x1000_8FFF	RW A	PRCI	
0x1000_9000	0x1000_FFFF		Reserved	
0x1001_0000	0x1001_0FFF	RW A	OTP Control	
0x1001_1000	0x1001_1FFF		Reserved	
0x1001_2000	0x1001_2FFF	RW A	GPIO	
0x1001_3000	0x1001_3FFF	RW A	UART 0	
0x1001_4000	0x1001_4FFF	RW A	QSPI 0	
0x1001_5000	0x1001_5FFF	RW A	PWM 0	
0x1001_6000	0x1001_6FFF	RW A	I2C 0	
0x1001_7000	0x1002_2FFF		Reserved	
0x1002_3000	0x1002_3FFF	RW A	UART 1	
0x1002_4000	0x1002_4FFF	RW A	SPI 1	
0x1002_5000	0x1002_5FFF	RW A	PWM 1	
0x1002_6000	0x1003_3FFF		Reserved	
0x1003_4000	0x1003_4FFF	RW A	SPI 2	
0x1003_5000	0x1003_5FFF	RW A	PWM 2	
0x1003_6000	0x1FFF_FFFF		Reserved	
0x2000_0000	0x3FFF_FFFF	R XC	QSPI 0 Flash (512 MiB)	Off-Chip Non-Volatile Memory
0x4000_0000	0x7FFF_FFFF		Reserved	
0x8000_0000	0x8000_3FFF	RWX A	E31 DTIM (16 KiB)	On-Chip Volatile Memory
0x8000_4000	0xFFFF_FFFF		Reserved	

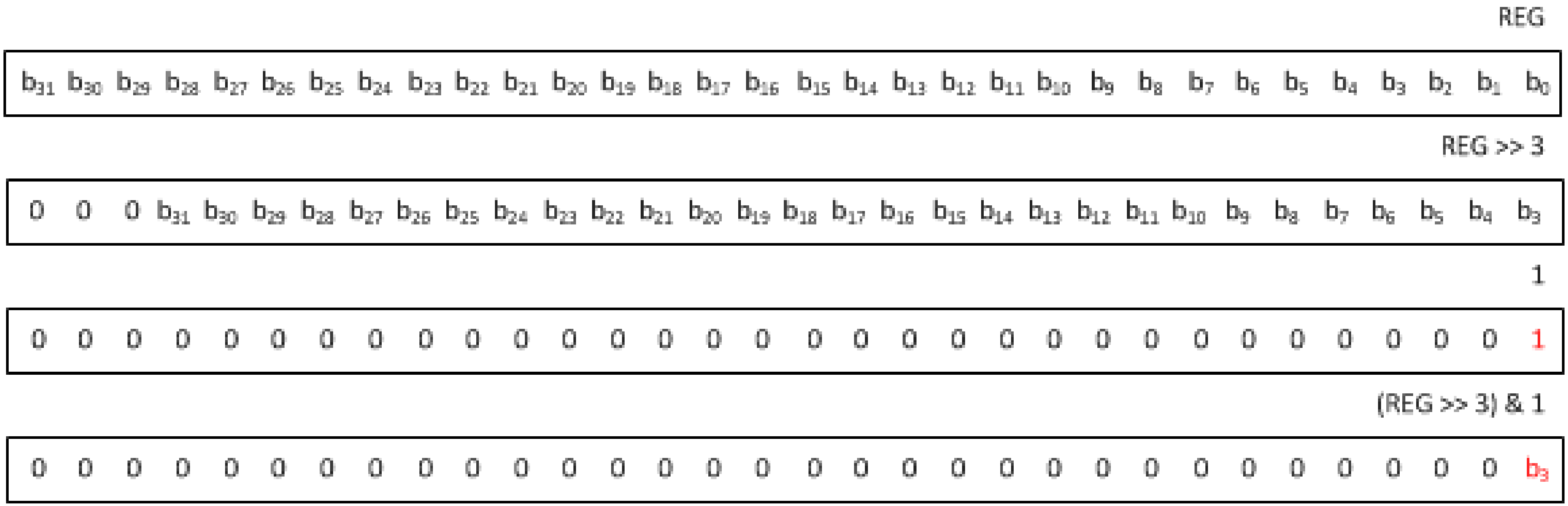
From FE310 Manual

Memory Mapped I/O in C

```
#include <stdint.h>
// Pointers to memory-mapped I/O registers
volatile uint32_t *GPIO_INPUT_VAL  = (uint32_t*)0x10012000;
volatile uint32_t *GPIO_OUTPUT_VAL = (uint32_t*)0x1001200C;

// read all the GPIO inputs
uint32_t allInputs = *GPIO_INPUT_VAL;

// read GPIO bit 19
int gpio19 = (*GPIO_INPUT_VAL >> 19) & 0b1;

// Wait for bit 19 to be 0
while ((*GPIO_INPUT_VAL >> 19) & 0b1);

// Wait for bit 19 to be 1
while (!((*GPIO_INPUT_VAL >> 19) & 0b1));

// Write 1 to GPIO bit 5
*GPIO_OUTPUT_VAL |= (1<<5);

// Write 0 to GPIO bit 5
*GPIO_OUTPUT_VAL &= ~(1<<5);
```

C Idioms: Read value of bit

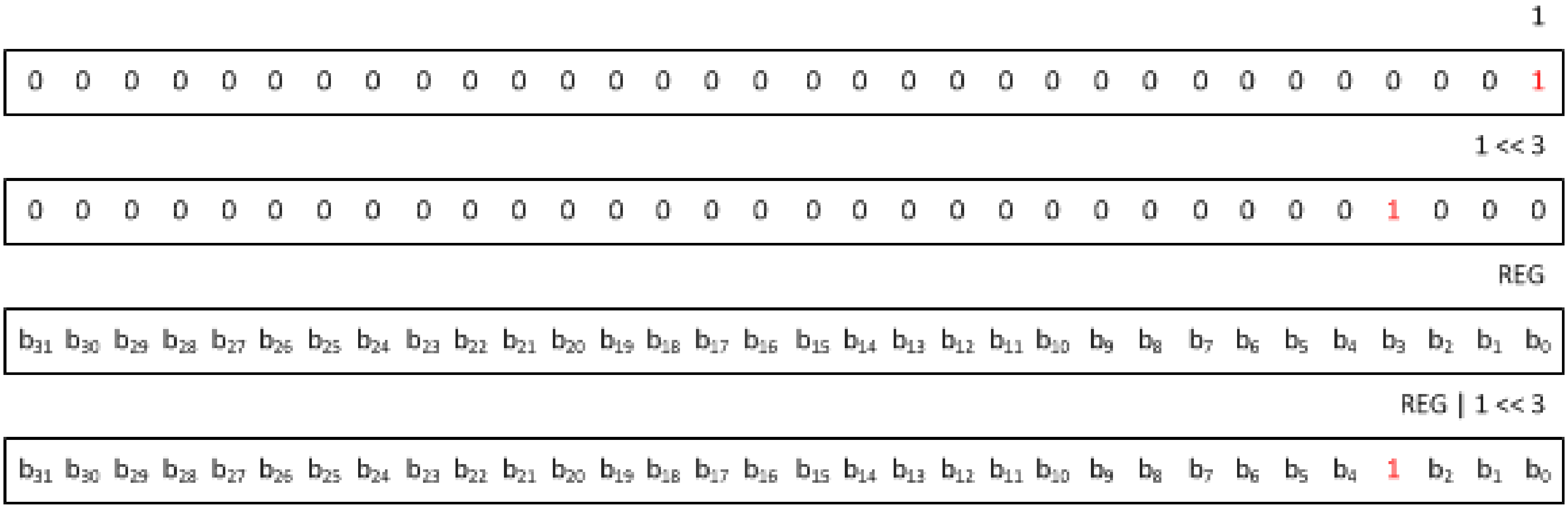
```
int bit = (*REG >> 3) & 1;   // get value of bit 3
```



REG	b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	b_{23}	b_{22}	b_{21}	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_{9}	b_{8}	b_{7}	b_{6}	b_{5}	b_{4}	b_{3}	b_{2}	b_{1}	b_{0}
REG >> 3	0	0	0	b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	b_{23}	b_{22}	b_{21}	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_{9}	b_{8}	b_{7}	b_{6}	b_{5}	b_{4}	b_{3}
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
(REG >> 3) & 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	b_{3}

C Idioms: Write Bit to 1

*REG |= (1 << 3); **// turn on bit 3**



1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

1 << 3

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0

REG

b_{31} b_{30} b_{29} b_{28} b_{27} b_{26} b_{25} b_{24} b_{23} b_{22} b_{21} b_{20} b_{19} b_{18} b_{17} b_{16} b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0

REG | 1 << 3

b_{31} b_{30} b_{29} b_{28} b_{27} b_{26} b_{25} b_{24} b_{23} b_{22} b_{21} b_{20} b_{19} b_{18} b_{17} b_{16} b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 1 b_2 b_1 b_0

C Idioms: Write Bit to 0

```
*REG &= ~(1 << 3);          // turn off bit 3
```

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1 << 3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
~(1 << 3)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
REG	b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	b_{23}	b_{22}	b_{21}	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
REG & ~(1 << 3)	b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	b_{23}	b_{22}	b_{21}	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	0	b_2	b_1	b_0

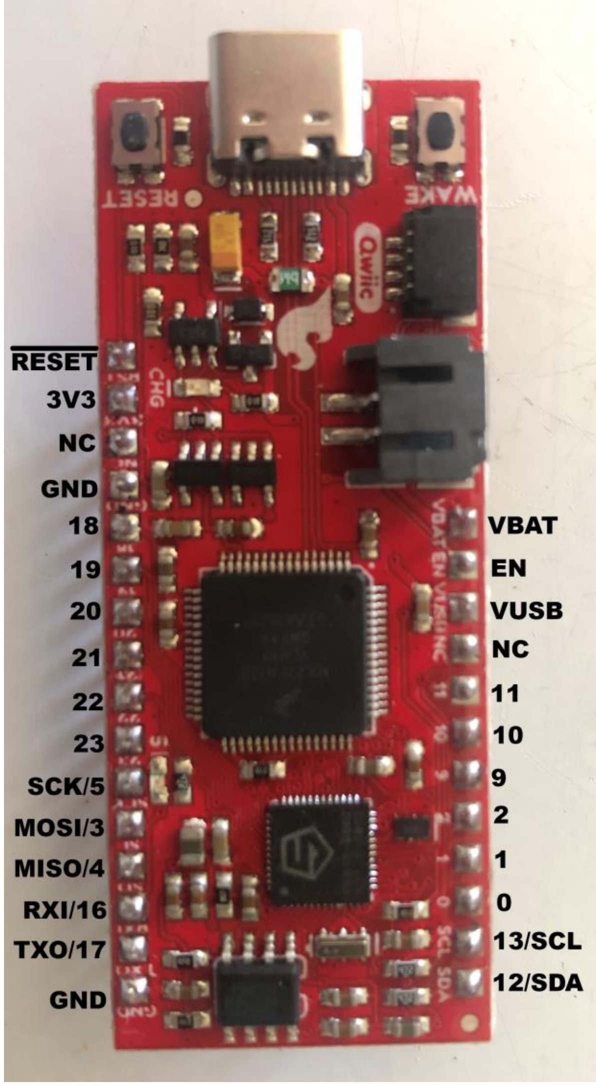
General Purpose I/O (GPIO)

General-Purpose I/O (GPIO)

- GPIOs can be written (driven to 0 or 1) or read.
- **Examples:** LEDs, switches, connections to other digital logic
- **In general:**
  - Look up how many pins your microcontroller has, how they are named.
  - Each pin needs to be configured as an input, output, or special function.
  - Then read or write the pin.

FE310 GPIOs

- **19 GPIOs connected to external pins**
  - Not enough pins for all 32 GPIOs
- **Some GPIOs have multiple functions**
- **Blue LED on RED-V board**
  - Connected to GPIO5



FE310 GPIO Registers

- Base Address
  - – 0x10012000
- 32 bits/reg
  - – For 32 GPIOs
- * means enable resets to 0
- Turn on enable to use GPIO

From FE310 Manual

Offset	Name	Description
`0x00`	`input_val`	Pin value
`0x04`	`input_en`	Pin input enable*
`0x08`	`output_en`	Pin output enable*
`0x0C`	`output_val`	Output value
`0x10`	`pue`	Internal pull-up enable*
`0x14`	`ds`	Pin drive strength
`0x18`	`rise_ie`	Rise interrupt enable
`0x1C`	`rise_ip`	Rise interrupt pending
`0x20`	`fall_ie`	Fall interrupt enable
`0x24`	`fall_ip`	Fall interrupt pending
`0x28`	`high_ie`	High interrupt enable
`0x2C`	`high_ip`	High interrupt pending
`0x30`	`low_ie`	Low interrupt enable
`0x34`	`low_ip`	Low interrupt pending
`0x40`	`out_xor`	Output XOR (invert)

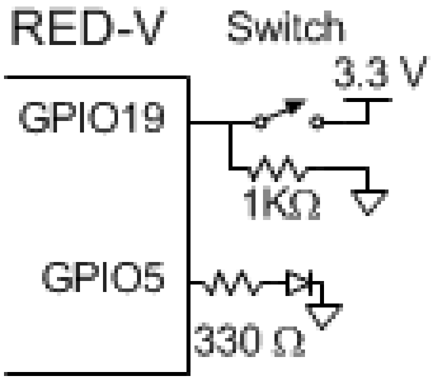
0x38	iof_en	Enable I/O function
0x3C	iof_sel	Select I/O function

FE310 GPIO With Enables

```
#include <stdint.h>
volatile uint32_t *GPIO_INPUT_VAL  = (uint32_t*)0x10012000;
volatile uint32_t *GPIO_INPUT_EN   = (uint32_t*)0x10012004;
volatile uint32_t *GPIO_OUTPUT_EN  = (uint32_t*)0x10012008;
volatile uint32_t *GPIO_OUTPUT_VAL = (uint32_t*)0x1001200C;

// Enable input 19 and output 5
*GPIO_INPUT_EN |= (1 << 19);
*GPIO_OUTPUT_EN |= (1 << 5);

// read GPIO bit 19
int gpio19 = (*GPIO_INPUT_VAL >> 19) & 0b1;

// Wait for bit 19 to be 1
while (!((*GPIO_INPUT_VAL >> 19) & 0b1));

// Write 1 to GPIO bit 5
*GPIO_OUTPUT_VAL |= (1<<5);

// Write 0 to GPIO bit 5
*GPIO_OUTPUT_VAL &= ~(1<<5);
```

Example: Switches & LEDs

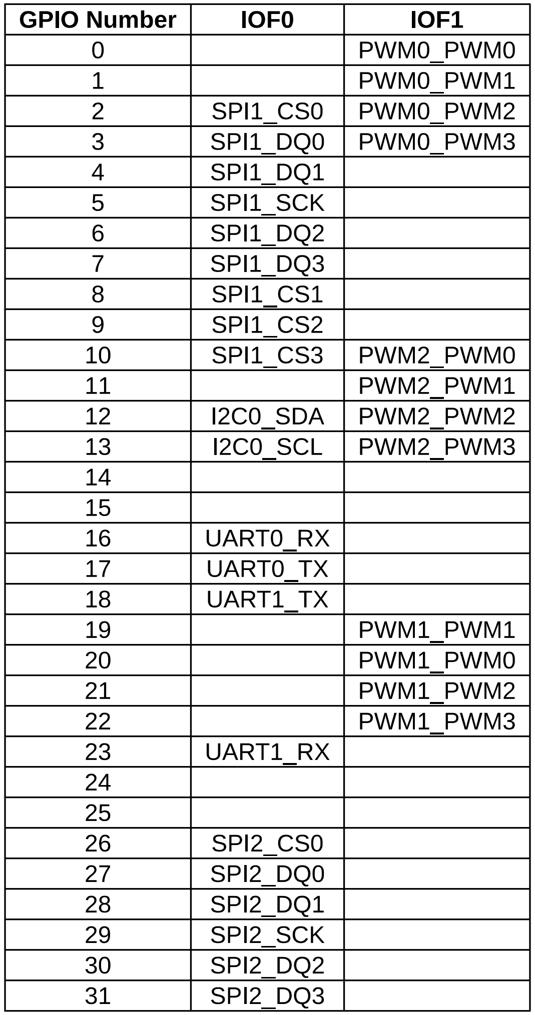
- Read a switch and drive an LED
  - (Could have done this with a wire!)



```
#include <stdint.h>
void main(void) {
    volatile uint32_t *GPIO_INPUT_VAL  = (uint32_t*)0x10012000;
    volatile uint32_t *GPIO_INPUT_EN   = (uint32_t*)0x10012004;
    volatile uint32_t *GPIO_OUTPUT_EN  = (uint32_t*)0x10012008;
    volatile uint32_t *GPIO_OUTPUT_VAL = (uint32_t*)0x1001200C;
    int val;

    *GPIO_INPUT_EN  |= (1 << 19);   // Set pin 19 to input
    *GPIO_OUTPUT_EN |= (1 << 5);               // set pin 5 to output
    while (1) {
        val =  (*GPIO_INPUT_VAL >> 19) & 1;   // Read value on pin 19
        if (val) *GPIO_OUTPUT_VAL |=  (1 << 5); // Turn ON pin 5
        else     *GPIO_OUTPUT_VAL &= ~(1 << 5); // Turn OFF pin 5
    }
}
```

Other I/O Functions

- Most GPIO pins also have **other special functions**
  - – Serial Ports: SPI, I2C, UART
  - – Pulse Width Modulation
- Activate with GPIO registers
  - – iof_en enables special function
  - – iof_sel selects the function



GPIO Number	IOF0	IOF1
0		PWM0_PWM0
1		PWM0_PWM1
2	SPI1_CS0	PWM0_PWM2
3	SPI1_DQ0	PWM0_PWM3
4	SPI1_DQ1	
5	SPI1_SCK	
6	SPI1_DQ2	
7	SPI1_DQ3	
8	SPI1_CS1	
9	SPI1_CS2	
10	SPI1_CS3	PWM2_PWM0
11		PWM2_PWM1
12	I2C0_SDA	PWM2_PWM2
13	I2C0_SCL	PWM2_PWM3
14		
15		
16	UART0_RX	
17	UART0_TX	
18	UART1_TX	
19		PWM1_PWM1
20		PWM1_PWM0
21		PWM1_PWM2
22		PWM1_PWM3
23	UART1_RX	
24		
25		
26	SPI2_CS0	
27	SPI2_DQ0	
28	SPI2_DQ1	
29	SPI2_SCK	
30	SPI2_DQ2	
31	SPI2_DQ3	

From FE310 Manual

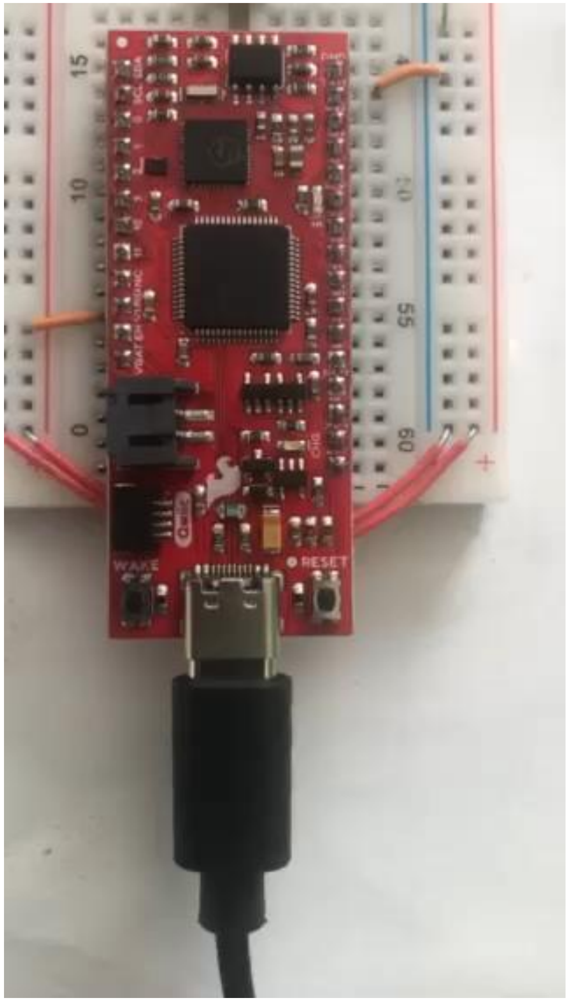
Delays & Timers

Generating Delays

```
#define COUNTS_PER_MS 1600

void delayLoop(int ms) {
	// declare loop counter volatile so it isn't optimized away
	// COUNTS_PER_MS empirically determined such that
	// delayLoop(1000) waits 1 sec
	volatile int i = COUNTS_PER_MS * ms;
	while (i--); // count down time
}
```

Blink Light

```
void flash(void) {
    pinMode(5, OUTPUT);

    while (1) {
        digitalWrite(5, 1);
        delayLoop(500);
        digitalWrite(5, 0);
        delayLoop(500);
    }
}
```

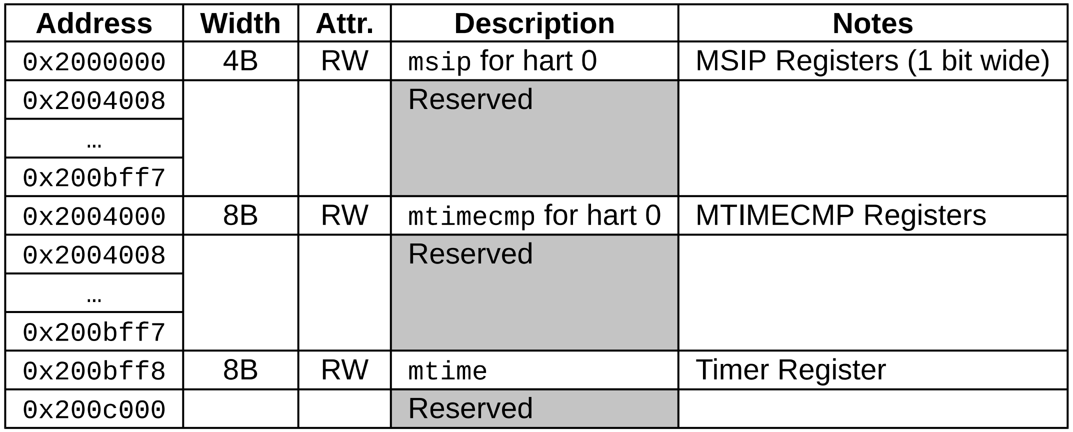


Timers

- **delayLoop** requires calibrating **COUNTS_PER_MS**
  - Tedious to calibrate by hand
  - Inexact
  - Could break if compiler optimizes better
- **Timers** are peripherals for time measurement
- FE310 has one **64-bit timer**
  - Counts ticks of an external 32.768 KHz oscillator

Timer Register

- Timers accessed through memory-mapped I/O
- FE310 timer in Core-Local Interruptor (CLINT)
  – 64-bit **mtime** register at **0x0200BFF8**



Address	Width	Attr.	Description	Notes
`0x2000000`	4B	RW	`msip` for hart 0	MSIP Registers (1 bit wide)
`0x2004008`			Reserved	
…				
`0x200bff7`				
`0x2004000`	8B	RW	`mtimecmp` for hart 0	MTIMECMP Registers
`0x2004008`			Reserved	
…				
`0x200bff7`				
`0x200bff8`	8B	RW	`mtime`	Timer Register
`0x200c000`			Reserved	

Table 24: CLINT Register Map

From FE310 Manual

Delay with Timer

```
void delay(int ms) {
   volatile uint64_t *mtime = (uint64_t*)0x0200bff8;
   uint64_t doneTime = *mtime + (ms*32768)/1000;

   while (*mtime < doneTime); // wait until time is reached
}
```

Interfacing Peripherals

Interfacing

- **Interfacing:** connecting external devices to a microcontroller
  - Sensors
  - Actuators
  - Other Processors
- **Interfacing Methods**
  - **Parallel**
  - **Serial**
    - **SPI:** Serial Peripheral Interface
      - 1 clock, 1 data out, 1 data in pin
    - **UART:** Universal Asynchronous Receiver/Transmitter
      - no clock, 1 data out, 1 data in pin, agree in software about intended data rate
    - **I2C:** Inter-Integrated Circuit
      - 1 clock, 1 bidirectional data pin
  - **Analog**
    - **ADC:** Analog/Digital Converter
    - **DAC:** Digital/Analog Converter
    - **PWM:** Pulse Width Modulation

Parallel Interfacing

- Connect **1 wire / bit** of information
  - **Example:** 8-bit parallel interface to send a byte at a time
- Also send **clock** or **REQ/ACK** (request/acknowledge) to indicate when data is ready
- Parallel busses are **expensive** and **cumbersome** because of the large number of wires
- Mostly used for **high-performance applications** such as DRAM interfaces

Serial Interfacing

- Serial interface sends **one bit at a time**
  - Use many clock cycles to send a large block of information
  - Also send timing information about when the bits are valid
- **Common Serial Interfaces:**
  - Serial Peripheral Interface (**SPI**)
    - Serial clock, 2 unidirectional data wires (MOSI, MISO)
    - Very common, easy to use
  - Inter-Integrated Circuit (**I2C**) “I squared C”
    - 1 clock, 1 bidirectional data wire
    - Fewer wires, more complex internal hardware
  - Universal Asynchronous Receiver/Transmitter (**UART**) “you-art”
    - 2 unidirectional data wires (Tx, Rx)
    - Asynchronous (no clock); configure (agreed upon) speed at each end

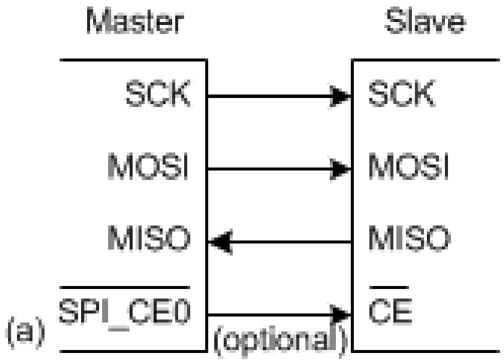
SPI

Serial Peripheral Interface (SPI)

- SPI is an easy way to connect devices
- **Master device** communicates with one or more **slave devices**
  - **Slave select signals** indicate which slave is chosen
  - **Master sends clock** and **data out**. **Slave** sends back **data in**.
- **Signals:**
  - **SCK/SCLK:** (Serial Clock) Generated by master; one pulse per bit
  - **MOSI:** (Master Out Slave In) serial data from master to slave
  - **MISO:** (Master In Slave Out) serial data from slave to master
  - **SS/CE/CS:** (Slave select/Chip Enable/Chip Select) optional, one or more, may be active low

SPI Waveforms

- SPI Slave hardware can be just a **shift register**

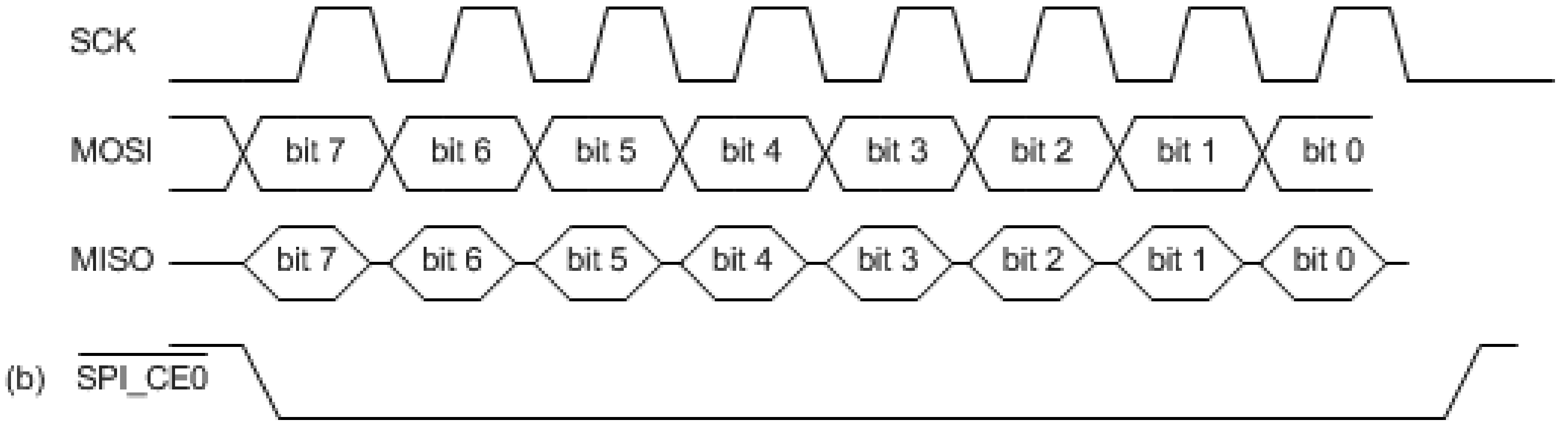


SCK: Generated by master; one pulse per bit

MOSI: (Master Out Slave In) serial data from master to slave

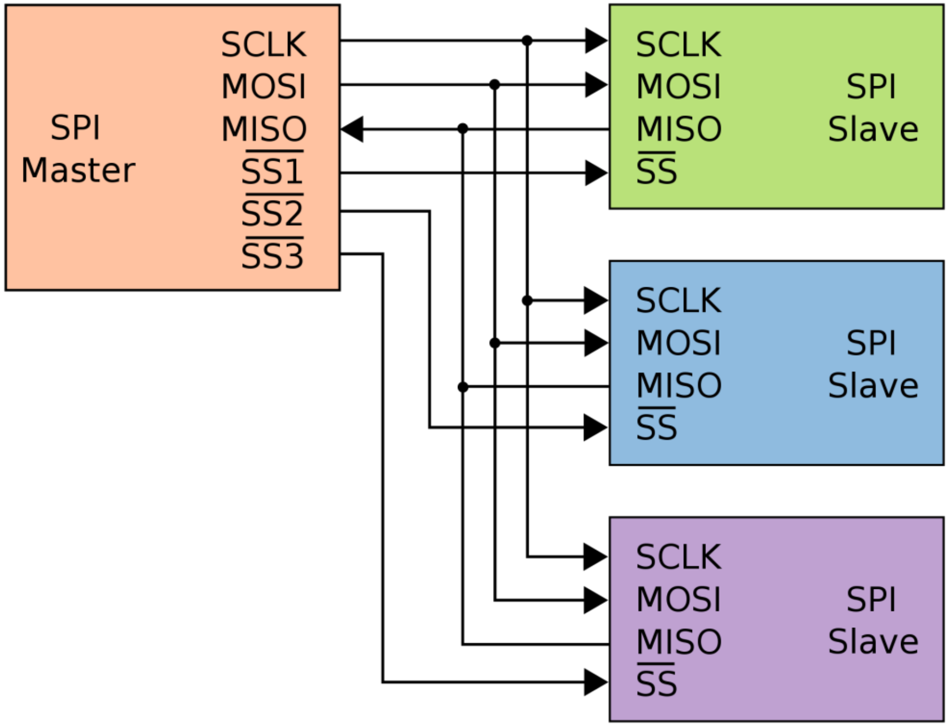
MISO: (Master In Slave Out) serial data from slave to master

CE: (Chip Enable) optional, one or more, may be active low

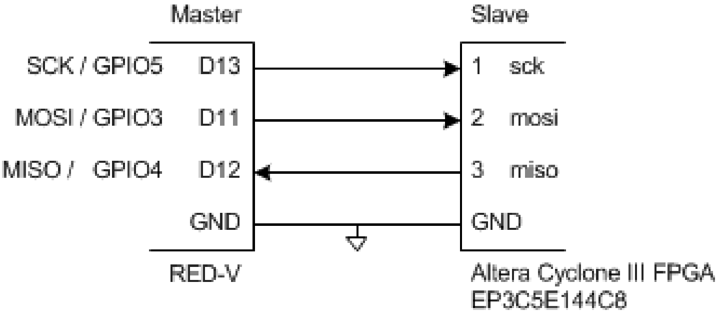


SPI Connection

Generic Master to Several Slaves



RED-V Master to One Slave



en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus#/media/File:SPI_three_slaves.svg

Slave Select

- Slave Select is **active low**
  - Variously named NSS or $\overline{\text{SS}}$ or $\overline{\text{CE}}$ or $\overline{\text{CS}}$
  - Turns ON device when 0, OFF when 1
- Turning **OFF** the slave device may **save power**
- **Options:**
  - **1 slave device:** tie slave select active
  - **1 or more slave devices:** use GPIO pins to turn desired device ON before SPI transaction, turn device back OFF afterward
  - **1 or more slave devices:** use SPI controller to automatically pulse the desired SS pin during a transaction

SPI Communication

- **Configure**
  - **Choose port**
    - FE310 has SPI1 and SPI2. Let’s use SPI1.
  - **Set SPI SCK, MISO, MOSI, CS0 pins to IOF0 mode**
  - **Set Baud Rate** (1 MHz or less prudent on a breadboard)
  - **Clock Polarity**
    - CPOL = 0 (default): clock idles at 0
    - CPOL = 1: clock idles at 1
  - **Clock Phase**
    - CPHA = 0: sample MOSI on the first clock transition
    - CPHA = 1: sample MOSI on the second clock transition
  - **Set other control registers as needed**
    - See EasyREDVIO code

SPI Communication Continued

- **Transmit data**
  - **Wait** for the **txdata full flag** to be 0 indicating SPI is ready for new data.
  - **Write byte** to **txdata** data field.
  - **SPI** will generate 8 clocks and send the 8 bits of data, while receiving 8 bits back.
- **Receive data**
  - **Read rxdata register.**
  - If **rxdata empty** flag is 1, there is no data yet. Repeat read.
  - When empty flag is 0, there is valid data in the **rxdata** field.

SPI Registers

- **Base Address**
  - SPI1 0x10024000
  - SPI2 0x10034000
- **Baud Rate**
  - sckdiv bits 11:0 indicate div
  - $F_{sck} = F_{in} / 2(div+1)$
  - F_{in} is 16 MHz coreclk
- **Clock Mode**
  - sckmode bit 0: pha (Clock Phase)
  - sckmode bit 1: pol (Clock Polarity
- **Transmit Data Register**
  - txdata bits 7:0: data (write to transmit)
  - txdata bit 31: full (1 indicates FIFO can accept data)
- **Receive Data Register**
  - rxdata bits 7:0: data (read received data)
  - rxdata bit 31: empty (1 indicates no data ready)