# Hardware and Interrupts
# Introduction to GPIO


Santosh Sam Koshy

Principal Technical Officer

C-DAC Hyderabad

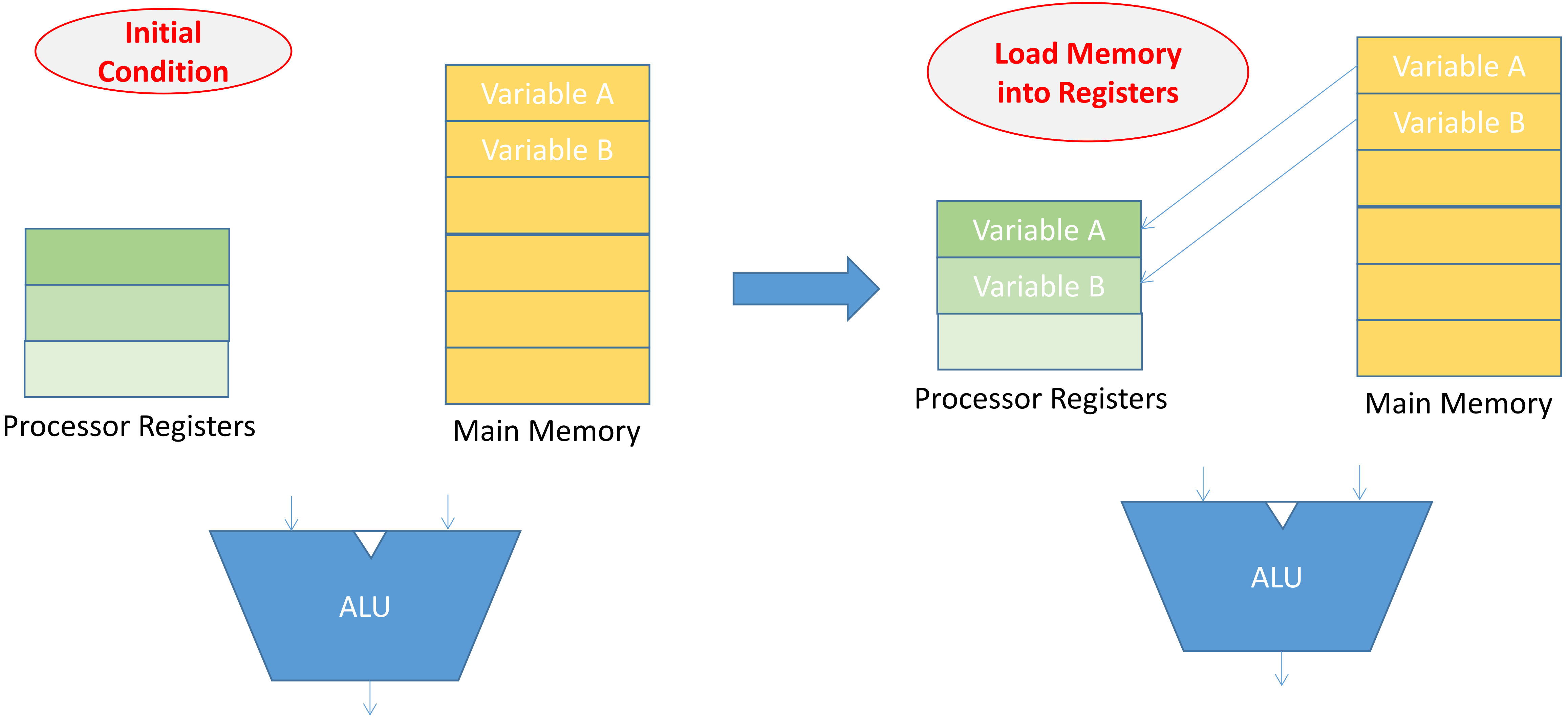
# Content

- Device Description
- Memory mapped IO and IO mapped IO
- Handling Interrupts in the kernel
- BBB Header Information
- GPIO Functions
- GPIO as an Input Device
- GPIO as an Output Device

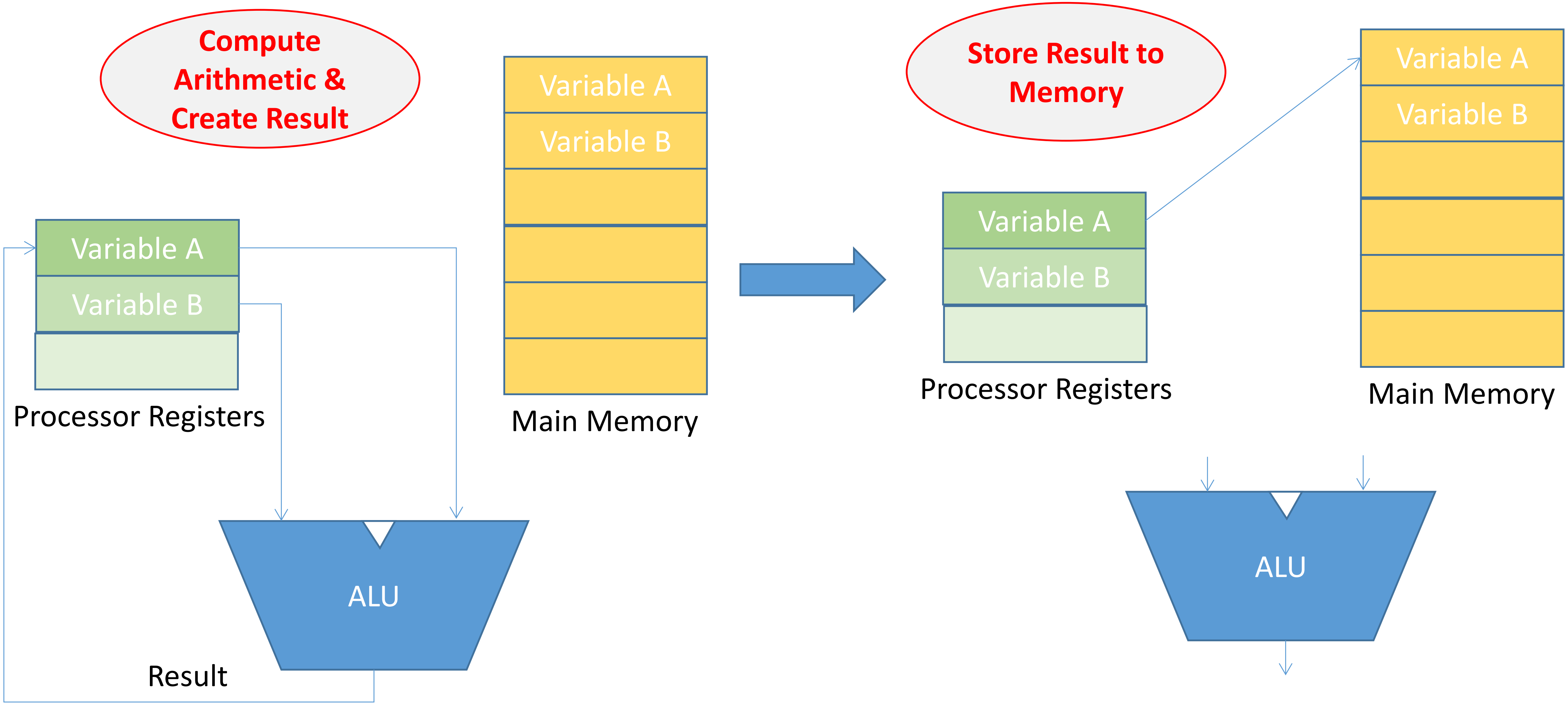
# Device Memory

- Peripheral devices are controlled by writing and reading its registers
- Registers
  - Data Registers (Reading and Writing)
  - Control Registers (Configuring the device. Supports R&W Operations)
  - Status Registers (Maintains state of the device. Generally Read Only)
- Every peripheral has a number of such registers, generally mapped contiguously in the Processor’s Memory Address Space or IO Address Space

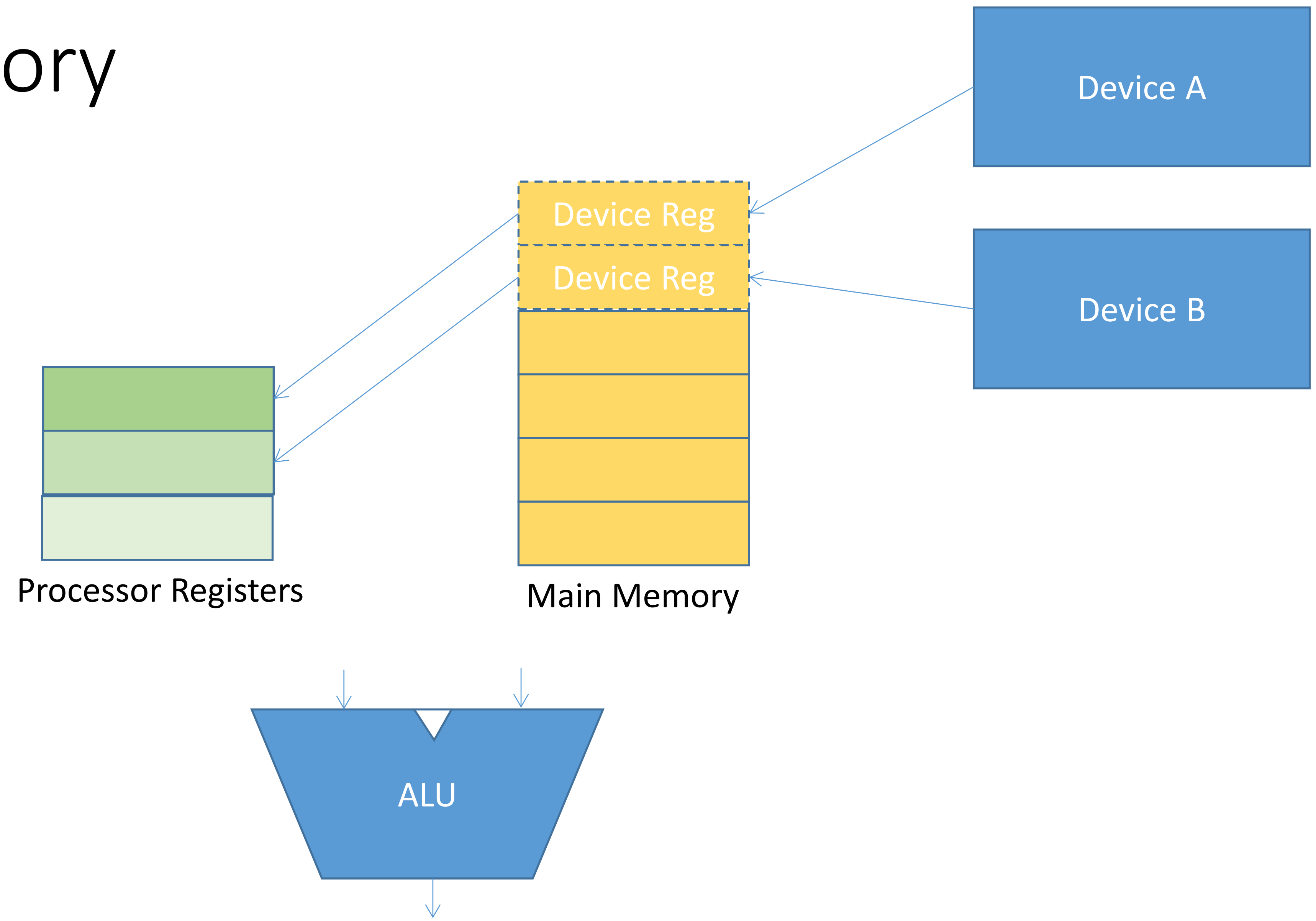
# Conventional Memory



# Conventional Memory



# IO Memory


Device A
Device B
Device Reg
Device Reg
Processor Registers
Main Memory
ALU

# Memory Caching in IO Ports

- Volatile Keyword
- Disable hardware caching during IO access. Done by the kernel itself

# Data Reordering

- Consider the Following Code Snippet
  - *Write (PIN FUNCTION, CLOCK DEVICE)*
  - *Write (FREQUENCY SETTING, CLOCK DEVICE)*
  - *Write (PRESCALAR VALUE, CLOCK DEVICE)*
  - *Write (START CLOCK, CLOCK DEVICE)*
- Compiler Reorders the data for optimized performance
  - *Write (PIN FUNCTION, CLOCK DEVICE)*
  - *Write (PRESCALAR VALUE, CLOCK DEVICE)*
  - *Write (START CLOCK, CLOCK DEVICE)*
  - *Write (FREQUENCY SETTING, CLOCK DEVICE)*

# Barriers

- The code will be modified like this
  - *Write (PIN FUNCTION, CLOCK DEVICE)*
  - *Write (FREQUENCY SETTING, CLOCK DEVICE)*
  - *Write (PRESCALAR VALUE, CLOCK DEVICE)*
  - *BARRIER();*
  - *Write (START CLOCK, CLOCK DEVICE)*

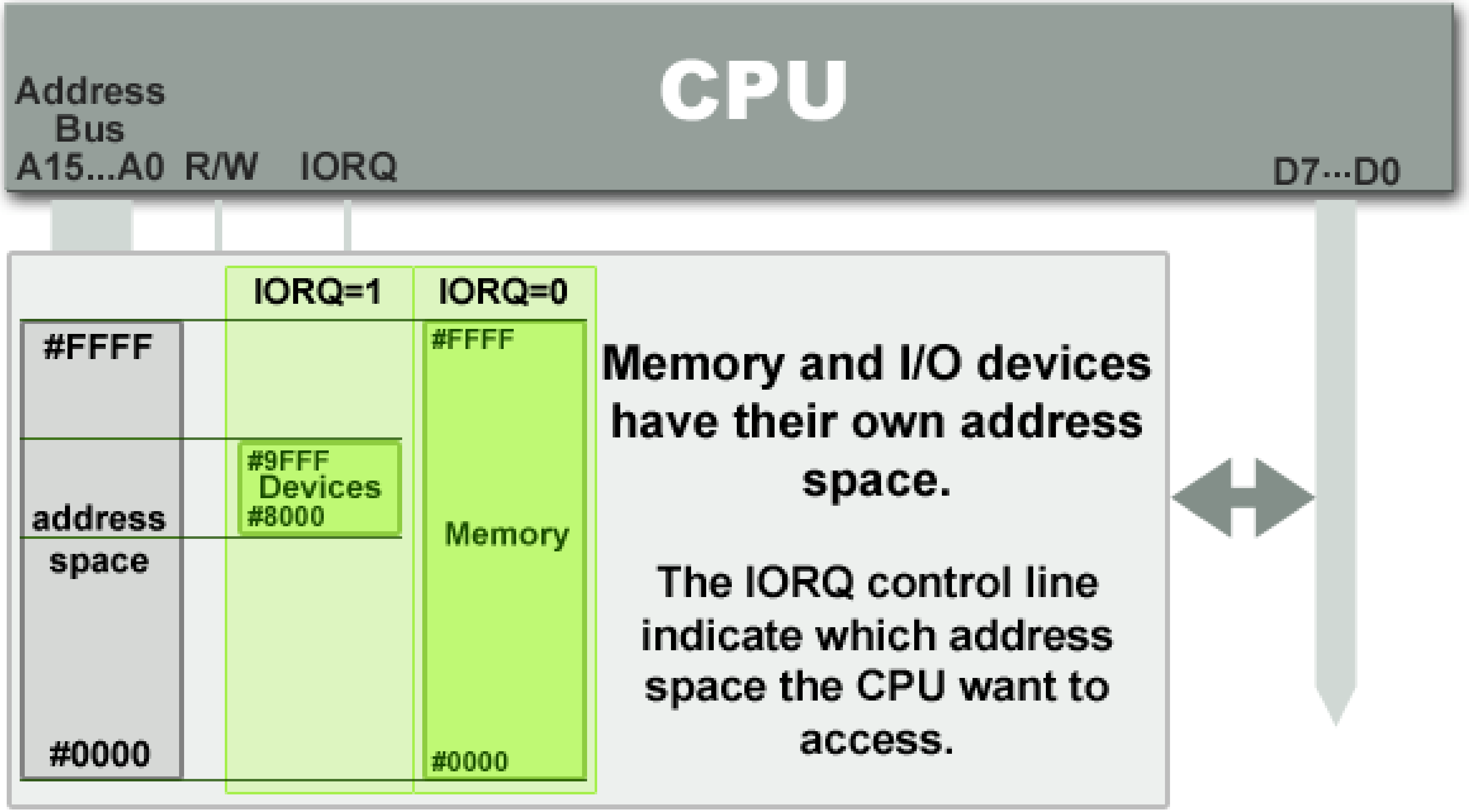
# Barriers in Linux Kernel

- The linux kernel provides 4 macros to cover all possible ordering needs
  - void barrier(void)
    - The kernel makes it a point that the compiler optimizations are absent across the barrier. The memory values present in the CPU registers are immediately stored in the memory
  - void rmb(void);
  - void read_barrier_depends(void);
  - void wmb(void);
  - void mb(void);
    - These functions insert hardware memory barriers in the compiled instruction flow. They are supersets of the barrier macro.
    - An rmb guarantees that any reads appearing before the barrier are completed prior to the execution of any subsequent read.
    - wmb orders write and mb orders both read and write.
- Header - #include <asm/system.h>

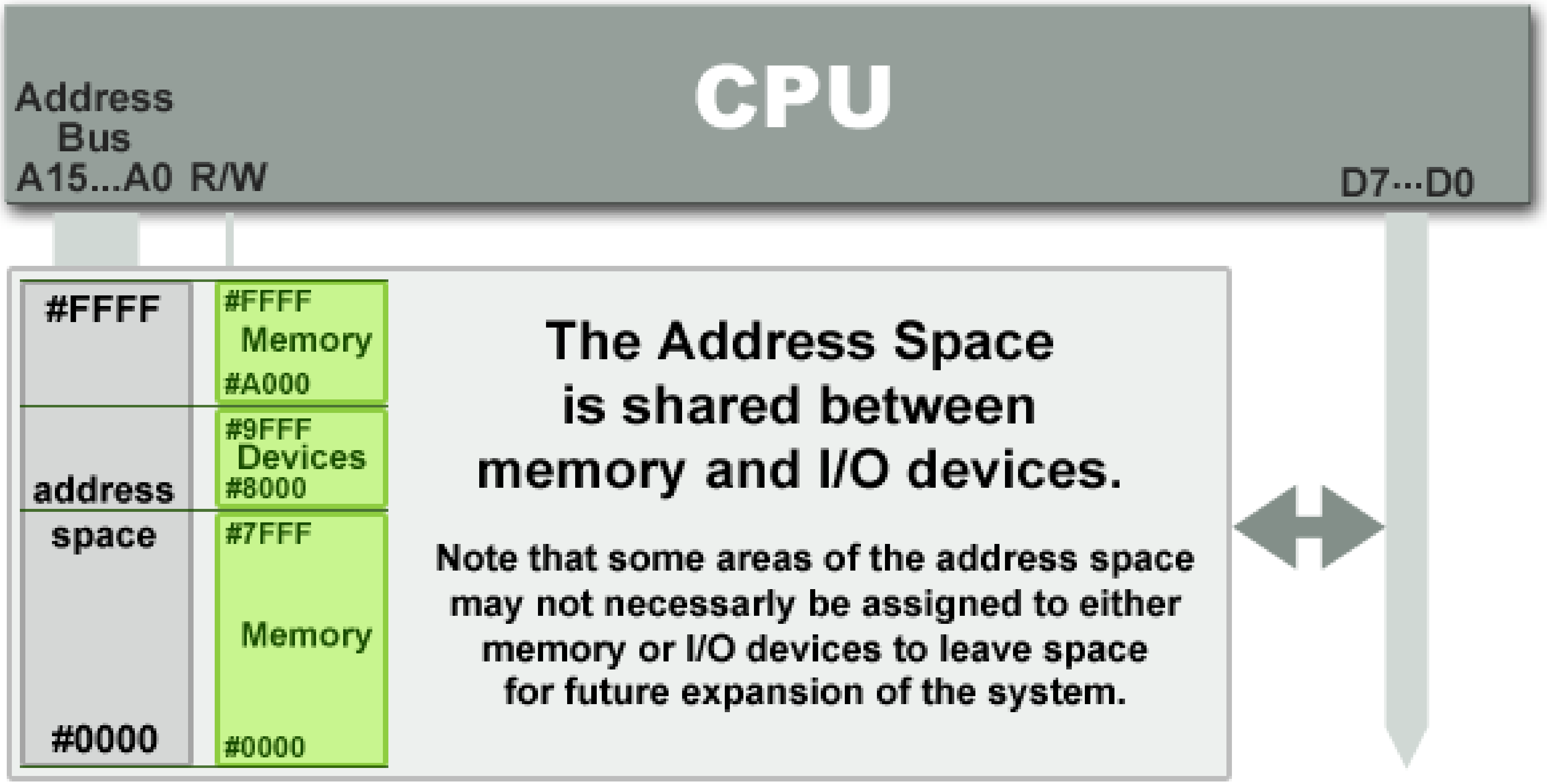
# Device Addressing

- IO Mapped IO or IO Ports
  - Peripheral devices are mapped into IO space and are accesssed through special instructions
  - Additional control lines are required to segregate IO access
  - Memory is better utilized
- Memory Mapped IO or IO Memory
  - Peripheral devices are mapped into the system memory
  - All access to this device space is performed using similar instructions that access memory
  - Effective memory gets reduced

# IO mapped IO (IOIO) or IO Ports



# Memory Mapped IO (MMIO) or IO Memory



# Communicating with IO Ports

- Drivers communicate with many devices through the I/O ports.
- Linux introduces a set of functions that may be used to gain exclusive access to an I/O region and communicate data transfers to and from these ports.
- Before embarking into communication with the I/O ports, the driver must gain access to the required ports by calling the function
  - struct resource *request_region(unsigned long first, unsigned long n, const char *name);
    - *The argument first is the first address requested*
    - *The argument long specifies the length of the addresses requested*
    - *The argument name identifies the requested region by this name in /proc/ioports*
- Header - #include <linux/ioport.h>

# Communicating with IO Ports

- When the I/O ports have been used as per requirement, they should be returned to the kernel so that other drivers may avail their existence. The I/O ports are returned by the function
  - void release_region(unsigned long start, unsigned long n);
- Before using a I/O region, a check on the availability has to be conducted to be certain that the requested region will be available for our use. A special function allows this check.
  - int check_region(unsigned long first, unsigned long n);
    - *This function returns a negative error code if the region is not available. This is a deprecated function and may not always prove to be true since it does not run atomic with the request_region function.*

# Communicating with IO Ports

- On successfully being alloted the region, the actual communication to the ports may be carried out using the kernel provided functions.
- These functions are specific to the port sizes on the I/O devices and provide interfaces for 8-bit, 16-bit and 32-bit ports
  - unsigned inb (unsigned port);
  - unsigned inw (unsigned port);
  - unsigned inl (unsigned port);
    - *Get data from the port specified as an argument*
  - void outb (unsigned char byte, unsigned port);
  - void outw (unsigned short word, unsigned port);
  - void outl (unisgned long word, unsigned port);
    - *Send data to the port specified in the argument*
- Header - #include <asm/io.h>

# Communicating with IO Ports

- The kernel also supports string operations that render its services in allowing a string of 'n' bytes to be transferred between the driver and the I/O port
  - void insb (unsigned port, void *addr, unsigned long count);
  - Similarly..insw, insl
    - *Copies count bytes of information from the port to addr*
  - void outsb(unsigned port, void *addr, unsigned long count);
    - *Write count data pointed by addr to the port*
- The kernel provides mechanisms of synchronization between a high end processor and a relatively slower I/O by allowing a pause functionality in data transfer.
- This feature may be accessed by ending the function names previously discussed with an '_p', such as inb_p, outb_p....

# IO Memory

- IO Memory regions must be allocated prior to use
  - struct resource *request_mem_region(unsigned long start, unsigned long len, char *name);
- IO Memory regions should when no longer in use
  - void release_mem_region(unsigned long start, unsigned long len);
- Accessing IO Memory regions should be preceded by a call to
  - void *ioremap(unsigned long phys_addr, unsigned long size);
  - void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
- After accessing the IO Memory region, unmap it using
  - void iounmap(void * addr);

# IO Memory

- To read from IO memory,
  - unsigned int ioread8(void *addr);
  - unsigned int ioread16(void *addr);
  - unsigned int ioread32(void *addr);
- Writing to the IO memory through
  - void iowrite8(u8 value, void *addr);
  - void iowrite16(u16 value, void *addr);
  - void iowrite32(u32 value, void *addr);
- Reading a series of values
  - void ioread8_rep(void *addr, void *buf, unsigned long count);
  - void iowrite8_rep(void *addr, const void *buf, unsigned long count);

# Interrupts - Definition

- Means to notify the kernel that a device is requesting attention.
- Characteristics of Interrupts
  - Interrupts occur Asynchronously
    - Processor completes the current instruction being executed
    - Jumps to Interrupt Service Routine
    - Handle the Interrupt
    - Return from Interrupt
  - Global Interrupts are disabled. It may be enabled in the handler by the driver
  - Interrupts must be handled quickly because the longer interrupts take to execute, the longer interrupts remain disabled
  - No Sleep, or other delay functions should be called in an interrupt
  - Less important functions of the interrupt should be performed in a bottom half (tasklet)

# Interrupts

- Components
  - IRQ Line – Interrupt Number
    - These are specific to a peripheral
    - It could be shared between number of devices
    - Limited number (32 in x86). You have to find out your IRQ number before you can use it. It is very specific to the hardware that you are using
  - Interrupt Handler
    - Each handler has to register itself with the kernel whenever an operation is to be performed on the device.
    - Registering a handler is a notification by the driver to the kernel, claiming authority for access to the device through a requested IRQ number.
    - On completion of access to the device, the handler may be unregistered and the irq number freed for allowing access to other applications.

# Interrupts – Requesting an IRQ and binding a handler

- Registering an interrupt handler to an irq number and notifying the kernel is done by
  - int request_irq ( unsigned int irq,
    irqreturn_t (*handler) (int, void *, struct pt_regs *),
    unsigned long flags,
    const char *dev_name,
    void *dev_id);
    - The value returned from request_irq to the is either 0 to indicate success or a negative error code, as usual.
    - It is not uncommon for the function to return -EBUSY to signal that another driver is already using the requested interrupt line.
- The handler is freed by calling
  - void free_irq (unsigned int irq, void *dev_id);

# Interrupts – Where to request for IRQ

- There are two places where an IRQ can be requested.
  - During the driver initialization at module_init function
    - For this, it is advisable that the interrupt is requested as a shared IRQ
  - In the open method of the driver
    - If called in the open call, it allows the use of the interrupt line only when the application requests for it.
    - The disadvantage of this technique is that a per device open count has to be maintained to know when interrupts can be disabled.

# Interrupts – FLAGS passed to request_irq

- FLAGS
  - SA_INTERRUPT (May be deprecated.. Just check)
    - This bit indicates a “fast” handler.
    - Fast interrupt handlers run with all interrupts disabled on the local processor
  - SA_SHIRQ
    - This bit indicates that the interrupt can be shared between devices.
    - The dev_id must be unique to each registered handler. A pointer to any per-device structure is sufficient. NULL cannot be passed to this field
    - The interrupt handler must be capable of detecting whether its device generated an interrupt. This requires both hardware support and associated logic in the handler
  - SA_SAMPLE_RANDOM
    - Adds to the kernel entropy pool to increase the randomness.

# Interrupts – The Handler Function

- The Interrupt Handler
  - static irqreturn_t intr_handler (int irq, void *dev_id, struct pt_regs *regs);
    - *The return value of an interrupt handler is the special type irqreturn_t.*
    - *An interrupt handler can return two special values, IRQ_NONE or IRQ_HANDLED.*
    - *IRQ_NONE is returned when the handler detects an interrupt for which its device was not the originator.*
    - *IRQ_HANDLED is returned if the interrupt handler was correctly invoked*

# Interrupts – Disabling and Re-enabling

- Single Interrupts
  - Sometimes the driver may need to disable interrupt delivery for a specific line. This is achieved by using one of the functions
    - void disable_irq(int irq);
    - void enable_irq(int irq);
- Disable all Interrupts
  - void local_irq_save(unsigned long flags);
  - void local irq_disable(void);
    - shuts off the interrupts
  - void local_irq_restore(unsigned long flags);
  - void local_irq_enable(void);
    - Re-enables the interrupts

# Doing it on the Beagle Bone Black

# Beaglebone Black Pinout Diagram

## P9

| Function | Physical Pins | | Function |
|---|---|---|---|
| DGND | 1 | 2 | DGND |
| VDD 3.3 V | 3 | 4 | VDD 3.3 V |
| VDD 5V | 5 | 6 | VDD 5V |
| SYS 5V | 7 | 8 | SYS 5V |
| PWR_BUT | 9 | 10 | SYS_RESET |
| UART4_RXD | 11 | 12 | GPIO_60 |
| UART4_TXD | 13 | 14 | EHRPWM1A |
| GPIO_48 | 15 | 16 | EHRPWM1B |
| SPIO_CSO | 17 | 18 | SPIO_D1 |
| I2C2_SCL | 19 | 20 | I2C_SDA |
| SPIO_DO | 21 | 22 | SPIO_SLCK |
| GPIO_49 | 23 | 24 | UART1_TXD |
| GPIO_117 | 25 | 26 | UART1_RXD |
| GPIO_115 | 27 | 28 | SP11_CSO |
| SP11_DO | 29 | 30 | GPIO_112 |
| SP11_SCLK | 31 | 32 | VDD_ADC |
| AIN4 | 33 | 34 | GND_ADC |
| AIN6 | 35 | 36 | AIN5 |
| AIN2 | 37 | 38 | AIN3 |
| AIN0 | 39 | 40 | AIN1 |
| GPIO_20 | 41 | 42 | ECAPWM0 |
| DGND | 43 | 44 | DGND |
| DGND | 45 | 46 | DGND |


| LEGEND |
|---|
| Power, Ground, Reset |
| Digital Pins |
| PWM Output |
| 1.8 Volt Analog Inputs |
| Shared I2C Bus |
| Reconfigurable Digital |

## P8

| Function | Physical Pins | | Function |
|---|---|---|---|
| DGND | 1 | 2 | DGND |
| MMC1_DAT6 | 3 | 4 | MMC1_DAT7 |
| MMC1_DAT2 | 5 | 6 | MMC1_DAT3 |
| GPIO_66 | 7 | 8 | GPIO_67 |
| GPIO_69 | 9 | 10 | GPIO_68 |
| GPIO_45 | 11 | 12 | GPIO_44 |
| EHRPWM2B | 13 | 14 | GPIO_26 |
| GPIO_47 | 15 | 16 | GPIO_46 |
| GPIO_27 | 17 | 18 | GPIO_65 |
| EHRPWM2A | 19 | 20 | MMC1_CMD |
| MMC1_CLK | 21 | 22 | MMC1_DAT5 |
| MMC1_DAT4 | 23 | 24 | MMC1_DAT1 |
| MMC1_DAT0 | 25 | 26 | GPIO_61 |
| LCD_VSYNC | 27 | 28 | LCD_PCLK |
| LCD_HSYNC | 29 | 30 | LCD_AC_BIAS |
| LCD_DATA14 | 31 | 32 | LCD_DATA15 |
| LCD_DATA13 | 33 | 34 | LCD_DATA11 |
| LCD_DATA12 | 35 | 36 | LCD_DATA10 |
| LCD_DATA8 | 37 | 38 | LCD_DATA9 |
| LCD_DATA6 | 39 | 40 | LCD_DATA7 |
| LCD_DATA4 | 41 | 42 | LCD_DATA5 |
| LCD_DATA2 | 43 | 44 | LCD_DATA3 |
| LCD_DATA0 | 45 | 46 | LCD_DATA1 |

# Introducing GPIO on BBB

- Header - #include <linux/gpio.h>
- Functions
  - static inline bool gpio_is_valid(int number)
    - check validity of GPIO number (max on BBB is 127)
  - static inline int  gpio_request(unsigned gpio, const char *label)
    - allocate the GPIO number, the label is for sysfs
  - static inline void gpio_free(unsigned gpio)
    - deallocate the GPIO line
  - static inline int  gpio_export(unsigned gpio, bool direction_may_change)
    - make available via sysfs and decide if it can change from input to output and vice versa
  - static inline void gpio_unexport(unsigned gpio)
    - remove from sysfs

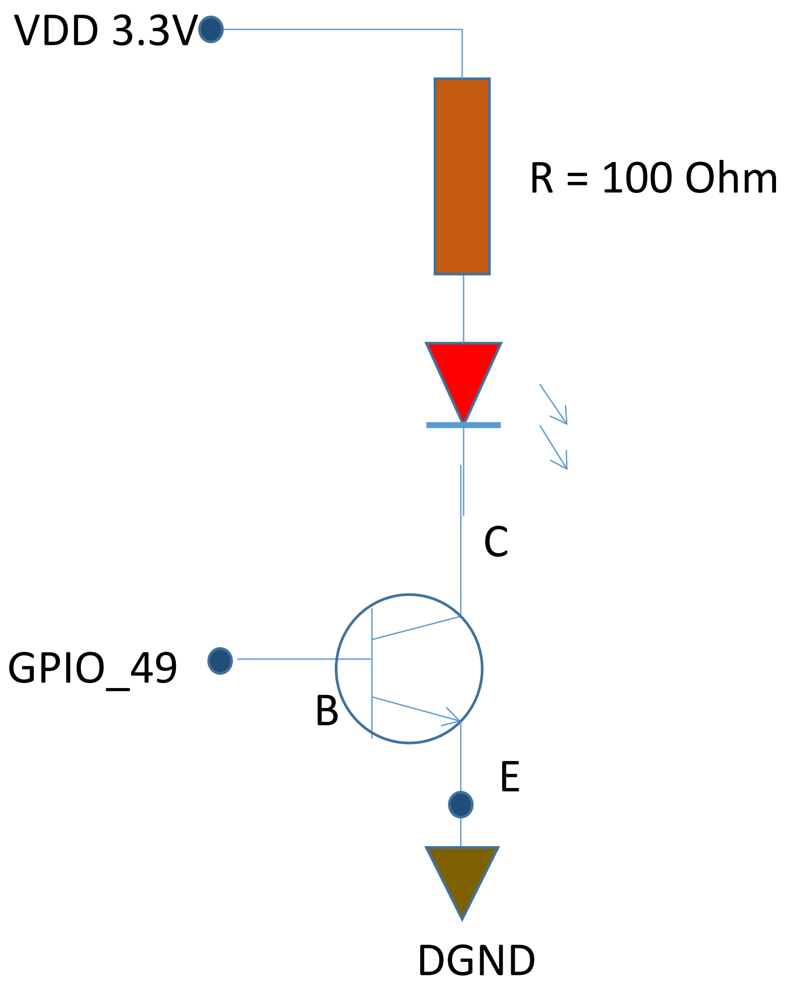
# Introducing GPIO on BBB

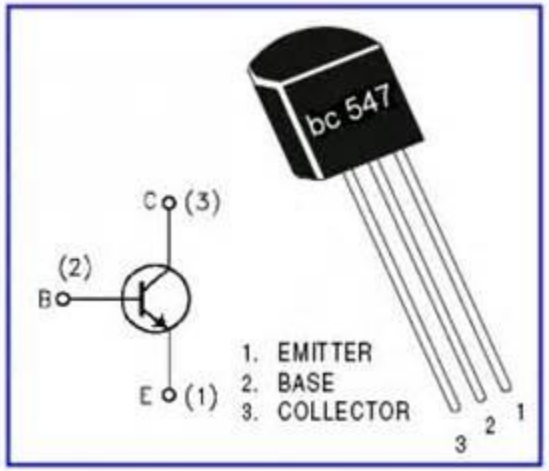
- static inline int  gpio_direction_input(unsigned gpio)
  - an input line (as usual, return of 0 is success)
- static inline int  gpio_direction_output(unsigned gpio, int value)
  - value is the state
- static inline int  gpio_get_value(unsigned gpio)
  - get the value of the GPIO line
- static void gpio_set_value(unsigned gpio, value)
  - Set the value of the GPIO line
- static inline int  gpio_set_debounce(unsigned gpio, unsigned debounce)
  - set debounce time in ms (platform dependent)
- static inline int  gpio_sysfs_set_active_low(unsigned gpio, int value)
  - set active low (invert operation states)
- static inline int  gpio_to_irq(unsigned gpio)
  - associate with an IRQ

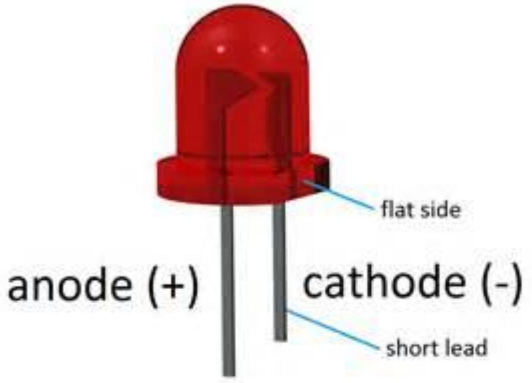
# Case Study – 1: Variable Frequency LED Toggle

- Involves IOCTL, Kernel Timers and GPIO
- Toggle an LED connected to GPIO_49 of your Beaglebone in a specific periodicity
- Use Kernel Timers to maintain toggle periodicity
- Using IOCTL from the user space, change the frequency of toggling by sending out a command through the driver to the LED
- Demonstrate…

# Required Circuit



# Case Study – 2: Toggle an LED when Switch is Pressed

- Connect a tactile switch to GPIO_115 on your BBB
- The switch should be pulled up, through a 15K Ohm resistor
- Whenever the switch is pressed, an interrupt should be generated and in the handler, the LED should be toggled
- Register the corresponding interrupt handler and IRQ number through GPIO access

# Required Circuit