Contents

[Lecture 1 - ISA 3](#_Toc87527533)

[Instruction Set Architecture (ISA) 3](#_Toc87527534)

[What are the functionalities required for ISAs? 3](#_Toc87527535)

[What are key ISA decisions? 3](#_Toc87527536)

[RISC vs CISC 3](#_Toc87527537)

[How many registers? 3](#_Toc87527538)

[Where operands reside? 4](#_Toc87527539)

[Harvard vs Von Neumann Architectures 4](#_Toc87527540)

[Microcontrollers vs Microprocessors 4](#_Toc87527541)

[Lecture 3 - SAM3x8e 5](#_Toc87527542)

[SAM3X8EModes 5](#_Toc87527543)

[General Purpose I/O Lines (GPIO) 5](#_Toc87527544)

[Input Schmitt Triggers 5](#_Toc87527545)

[Glitch Filters & Debouncing 5](#_Toc87527546)

[Timers 5](#_Toc87527547)

[Lecture 4 – Interrupt 1 6](#_Toc87527548)

[Interrupts 6](#_Toc87527549)

[Where to Branch? 6](#_Toc87527550)

[How to handle Nested Interrupts 6](#_Toc87527551)

[Nested Vectored Interrupt Controller (NVIC) 6](#_Toc87527552)

[Tail Chaining 7](#_Toc87527553)

[Late-Arriving 7](#_Toc87527554)

[Priority Masking 7](#_Toc87527555)

[How does the CPU know when data is available? 7](#_Toc87527556)

[How is data transferred into and out of the device? 7](#_Toc87527557)

[Lecture 5 – Interrupt 2 8](#_Toc87527558)

[Maximum Interrupt Rate 8](#_Toc87527559)

[Sharing Data Safely between ISRs and other Threads 8](#_Toc87527560)

[Lecture 6 - GPIO 9](#_Toc87527561)

[Parallel Input/Output Controller (PIO) 9](#_Toc87527562)

[Lecture 7 – SPI 10](#_Toc87527563)

[SPI capabilities 11](#_Toc87527564)

[SPI Protocol 11](#_Toc87527565)

[Embedded Characteristics 11](#_Toc87527566)

[Functional Description 12](#_Toc87527567)

[Data Transfer 13](#_Toc87527568)

[SPI Controller Register (Write-Only Register) 13](#_Toc87527569)

[PSI Mode Register (Read-Write Register) 13](#_Toc87527570)

[SPI Receive Data Register (Read-Only Register) 13](#_Toc87527571)

[SPI Transmit Data Register (Write-Only Register) 14](#_Toc87527572)

[Lecture 8 – I2C 14](#_Toc87527573)

[How does it transfer to a specific master/slave? 14](#_Toc87527574)

[Operating Roles 15](#_Toc87527575)

[I2C Modes 15](#_Toc87527576)

[Data Handshake Sequence 15](#_Toc87527577)

[Data Transfer Sequence 15](#_Toc87527578)

[Two-wire interface (TWI) 15](#_Toc87527579)

# Lecture 1 - ISA

## Instruction Set Architecture (ISA)

A set of instructions used by a machine to run programs. ISA can be interpreted as an interface between **Software** and **Hardware**.

Note: **Hardware implementation decides what and how instructions are implemented**.

## What are the functionalities required for ISAs?

* Load / Store
* Control
* Arithmetic / Logic

## What are key ISA decisions?

* Instruction Length
  + Variable Length
    - Require more steps to fetch and decode instructions
    - Allow for more flexible and compact instructions
    - ***CISC architecture***
  + Fixed Length
    - Allow easy instruction fetch and decode
    - Simplify *pipelines* and *parallelism*
    - ***RISC architecture***
* Number, Length and types of registers
* Where operands reside
* How to access the memory (Direct, Indirect, …)
* Instruction Format
* Operand Format (How many? How big?)

## RISC vs CISC

|  |  |
| --- | --- |
| **RISC** | **CISC** |
| Little load on single instruction | Heavy load on single instruction |
| Emphasis on ***Software*** | Emphasis on ***Hardware*** |
| Single clock needed | Multiple clocks needed |
| Heavy use of RAM | More efficient use of RAM |
| Complex and **variable length** | Simple and **fixed length** |
| More registers and cache needed | Less registers and cache needed |

### How many registers?

Why small number of registers?

* Fewer bits required to identify the register
* Less *Hardware* is used
* Faster access (Fewer and shorter wires needed)
* Faster context-switch

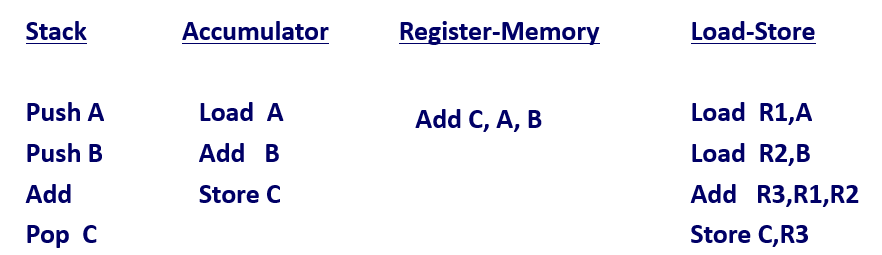
Why large number of registers?

* Fewer loads/stores needed (Less data transfer between CPU and MM)
* Easier to do several operations at once

### Where operands reside?

* **Stack** Machine (Decapitated)
* **Accumulator** Machine
* **Register-Memory** Machine (Used nowadays!)
* **Load-Store** Machine (Aka. **Register-Register** Machine)

Example of “C = A + B” in different machine architectures:

****

## Harvard vs Von Neumann Architectures

|  |  |
| --- | --- |
| **Von Neumann** | **Harvard** |
| Old architecture | Modern architecture |
| ***Same*** physical memory for data & instructions | ***Sperate*** physical memory for data & instructions |
| **Common** bus for data & instructions transfers | **Separate** bus for data & instructions transfers |
| **Two Clocks** required to execute instruction | **One Clocks** required to execute instruction |
| Cheap | Expensive |
| CPU **cannot** access instructions and read/write at the same time | CPU **can** access instructions and read/write at the same time |
| Used in personal computers | Used in Microcontrollers |

## Microcontrollers vs Microprocessors

|  |  |
| --- | --- |
| **Microprocessors** | **Microcontrollers** |
| Heart of **Computer** systems | Heart of **Embedded** systems |
| Only the **CPU** | **CPU, Memory and I/O** |
| Larger circuit (External components needed) | Smaller circuit (Components are internal) |
| **Cannot** be used in **compact systems** | **Efficient** to be used in **compact systems** |
| Hight energy consumption | Low energy consumption |
| Slow memory access | Fast memory access |
| **Von Neumann** architecture | **Harvard** architecture |

# Lecture 3 - SAM3x8e

## SAM3X8EModes

* Active Mode

Normal running mode with normal core clock. (Normal use of device)

* Low Power Modes
  + Backup Mode
    - **Lowest power consumption**
    - **Slow startup time**
    - **Core is offline**
  + Wait Mode
    - **Very low power, but still in powered state!**
    - **Fast startup**
    - **Clocks of Core, Peripherals and memories are stopped (But still powered!)**
  + Sleep Mode
    - Only core clock is stopped
    - **No instructions are executed, but Peripheral clock is still enabled! (I/O is working)**

## General Purpose I/O Lines (GPIO)

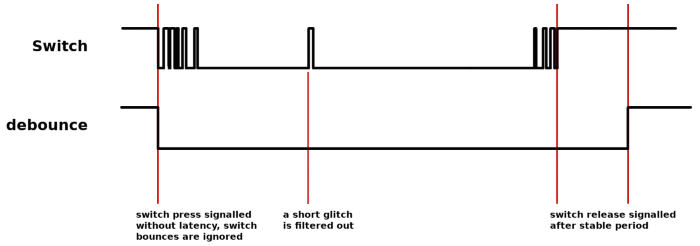
### Input Schmitt Triggers

0-5 Voltage is mapped into two binaries 0 and 1. But there is a *threshold* that defines the line between the two outputs. (Say threshold is 0.5, voltages from 3 and above are considered as 1 and others as 0).

### Glitch Filters & Debouncing

As shown below, bouncing happens when lots of signal bouncing appears in a very short period of time. Debouncing is an act of filtering those noises. (How? Switch releases after a short period of time to ignore the noises)

Glitch filter also filters out the random short noises in the signal.



## Timers

* Real-Time timer (RTT)

Generates **periodic interrupt (or triggers an alarm)**, **counting elapsed seconds**. (Read-only)

* Real-Time clock (RTC)

with **low power consumption** generates **Programmable** periodic interrupts. Stores a time-of-day and a 200-year **Gregorian calendar**.

* Watchdog-Timer (WDT)

Used to prevent system lock-up. If a software becomes trapped in a **deadlock**, an interrupt is generated with period of **16 seconds** (Countdown) to reset the **processer**.

# Lecture 4 – Interrupt 1

## Interrupts

An event that causes the CPU to **stop executing the current program** and **begin executing a special piece of code** called **an interrupt handler** or **interrupt service routine (ISR)**. Typically, the ISR does some work and then resumes the interrupted program. Interrupts:

* Can occur between any two instructions (**Asynchronous**)
* Are transparent to the running program (**Usually**)
* Not explicitly requested by the program (**Typically**)
* Call a procedure at an address by the **type of interrupt**, not the program itself!

### Where to Branch?

You either know what caused the interrupt or you don’t.

If you do, number (index) the possible interrupt cases and branch to a location using that number as an offset (Branch table).

If you don’t, you poll all possible sources of interrupts to see who caused it; Then branch to the right code. (**Bad design, high resource consumption rate**)

## How to handle Nested Interrupts

What if we get an interrupt while handling another? Three options: **Just handle it, ignore it, prioritize it.**

* Most instructions are short and fast to execute.
* Some instructions can many cycles to execute: ***Load Multiple (LDM), Store Multiple (STM), Push, Pop, MULS***. When these interrupts are executing and a new interrupt occurs:
  1. Abandon the current instruction
  2. Respond to the new interrupt
  3. Execute the ISR
  4. Return from the new interrupt
  5. ***Restart*** the abandoned instruction!!!!!!!!

# Nested Vectored Interrupt Controller (NVIC)

* NVIC manages and prioritizes external interrupts.
* There are 4 exception states:
  + **Inactive**: When we have no interrupts occurring.
  + **Pending**: Interrupt is occurring but CPU has not yet responded (Not entered in ISR)
  + **Active**: CPU is processing the Interrupt (ISR is working)
  + **A&P**: ISR is processing the interrupt, but **the same device** has sent another interrupt!
* Process state is automatically stored in Stack on an exception and is restored at the end of ISR.
* Interrupt priorities with **negative values** are by **Hardware** and are not configurable.
* Position 0 for interrupts is for ***Stack*** and nothing else. (That’s why indexing starts at 1!)

### Tail Chaining

This helps in executing the interrupts back-to-back without the problem of context switching by enabling back-to-back interrupts without the overhead of state saving and restoration. (Previous state isn’t restored and then backed-up again between each two interrupts)

### Late-Arriving

We are restoring registers for an interrupt and a higher priority interrupt occurs. Here we execute the late interrupt although it occurred later on. (Higher priority!)

### Priority Masking

Masking means ignoring some interrupts when we don’t want to be interrupted (Like being in *Critical Section*).

* Primask: 1-Bit register that masks all interrupts with priority lower than 0 (Higher number).
* Basepri: 8-bit register that takes a number in range of 0-240 and masks all interrupts with lower priority that that number. (Higher number)
* Faultmask: 1-Bit register like *Primask* but masks interrupt lower with priority lower than -1 (Higher number).

## How does the CPU know when data is available?

* Polling

Continuously scan for changes in signals. (CPU is always busy, bad design)

* Interrupts

Devices throw interrupt signals to inform CPU of any changes.

## How is data transferred into and out of the device?

* Programmed I/O

CPU itself handles the data transfer which prevents it from processing other tasks (I/O tasks are costly) and this decreases the CPU Utility.

* Direct Memory Access (DMA)

DMA is a feature of computer systems that allows certain hardware subsystems to access main system memory independently of the central processing unit. This causes **higher transfer rate**, **fewer CPU cycles for each transfer**.

This way I/O sends interrupts on batches (blocks) of data instead of each Byte that it processes.

# Lecture 5 – Interrupt 2

## Maximum Interrupt Rate

* How many interrupts can we handle per second?
* What is CPU utilization?
  + CPU looks like it’s running other codes with CPU speed clock of

## Sharing Data Safely between ISRs and other Threads

* **Volatile Data**

How is optimization done by compiler? When dealing with programs compiler reads data from Main Memory once and doesn’t do Load/Save in Main Memory unless necessary (Keeps data in the registers).

How can this optimization be bad? I/O and Interrupts might independently want to change the data (variables) we are using in our program from **Main Memory**. If they do so, the data will change in the Main Memory but your program is keeping the old values, but your program sees the data only in the registers due to optimization. So, it won’t detect the changes!

How to solve this problem? Variables that are accessed not only your program, but also from other devices, must be defined with **Volatile** keyword (So the optimization isn’t done).

* **Non-Atomic Shared Data**

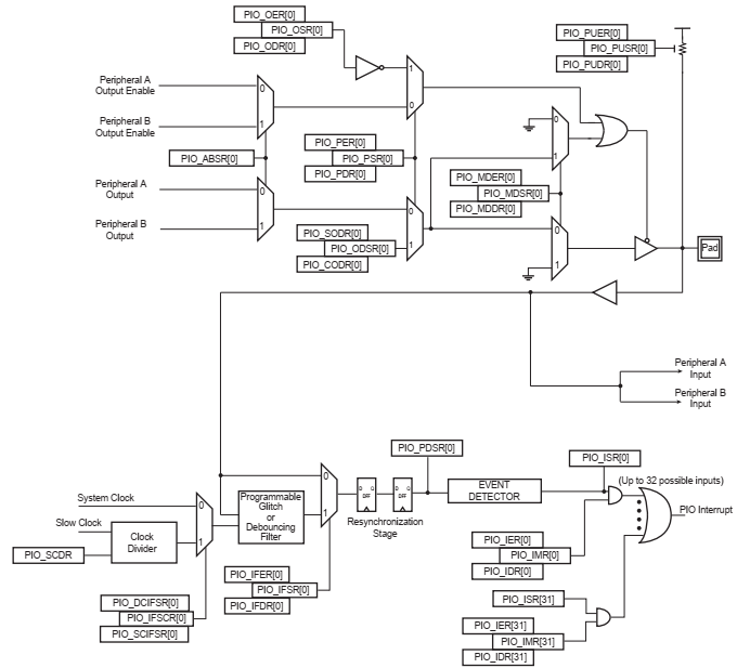
How it happens? This occurs when we have half-read a variable and then the variables value is changed by and external interrupt. When we continue to read the rest of the data (After the interrupt has completed) our data is corrected! So, we are vulnerable to **Race Conditions**.

How to prevent it? By guaranteeing that no interrupts are occurred when we are doing a specific task (E.g., Reading value of a variable). So, we disable incoming interrupts until our processing is finished.

# Lecture 6 - GPIO

## Parallel Input/Output Controller (PIO)

* Up to 32 fully-programmable I/O lines.
* You can’t add any PIO pin to any Peripherals (Check in sheet before hand)
* Each I/O be assigned as a general-purpose I/O to an embedded peripheral. (DMA, Lan, USB, ...)
* Can have many mods: Rising Edge, Falling Edge, Low Level, High Level, …
* We can enable/disable **Glitch Filter** and **Debouncing** for the PIO controller.
* Can control if line is *Pull-up.*
* *Allows for parallel transfer of 32-bit or 16-bit data.*
* The PMC (Power Management controller) controls the clock in order to save power.
* Power Management:
  + Writing registers of **User Interface** doesn’t require **PIO clock** to be enabled.
  + Not all features of the PIO controller are available when **clock is disabled**.
  + After a hardware reset PIO clock is disabled (Clock is disabled by default)
  + **Input Change Interrupt**, **Interrupt Modes** on a programmable event and the **read of the pin** level require the **clock to be validated**.
* Interrupt Generation:
  + PIO Controller interrupts are generated only when clock is enabled, duh.
  + PIO Controller requires NVIC to be programmed first.



* Why use three registers? One for enabling, one for disabling and one for keeping the status. We might want to change the status of a register for a short time without having the overhead of having to save/load its state. So, we may set disable=1 so status=0. We do our things and set disable=0, now the fate of the status relies on the enable register (We didn’t have to store its state anymore!)

# Lecture 7 – SPI

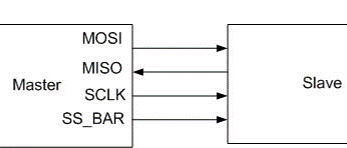
SPI is a serial-synchronous-protocol for transferring data with 4 wires. In synchronous protocol, there is always a wire for shared clock between sender and receiver. Synchronous serial data link:

* Provides communication with external devices in **Master** or **Slave** Mode
* Is essentially **a shift register** that serially transmits data bits to other SPIs
* Fast, Easy to use, Simple
* Used for small distance communication.

## SPI capabilities

* **Always Full-Duplex** (E.g., Telephone) (Can send and receive data at the same time) – In contrast to Half-Duplex (E.g., Walkie-Talkie)
* **Multiple Mbps transmission speed**
* **Transfers data in 8-bit to 16-bit characters**
* **Multiple Slaves**
  + **Daisy-chaining possible** (Devices are connected in a series – No direct connection between master and the target client)
* **Master controls the data flow**

Other devices act as slaves which have data shifted into and out by the master

****

## SPI Protocol

* ***Master Out Slave In (MOSI):***

Outputs data from master to slave(s).

* ***Master In Slave Out (MISO):***

Slave(s) output data to the master input.

* ***Serial Clock (SPCK)***

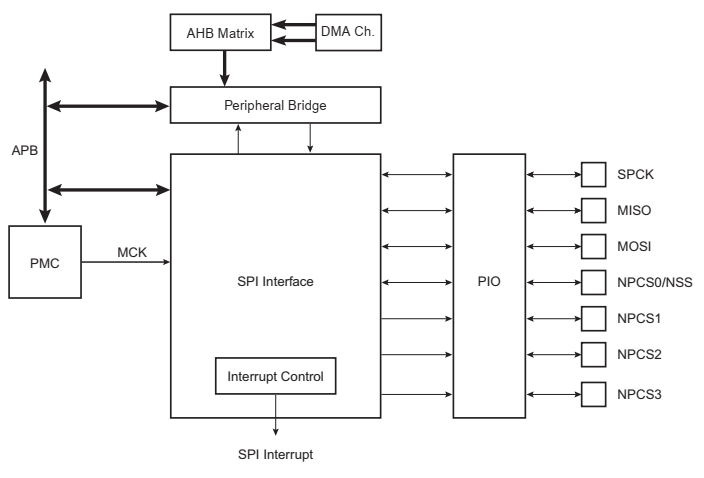
Is driven by the master and regulates the flow of the data bits. The master may transmit data at a variety of baud rates; the SPCK line cycles once for each bit that is transmitted.

* ***Slave Select (NSS)***

Allows slaves to be turned on and off by hardware.

## Embedded Characteristics

* There can be one master and multiple slaves. Multiple slaves can receive data from master at the same time, **but can’t send their data to the master simultaneously**!
* Microcontroller can communicate with maximum 15 peripherals at the same time.
* We can attach Co-Processor, LCD, CAN, Memories and … to the microcontroller.

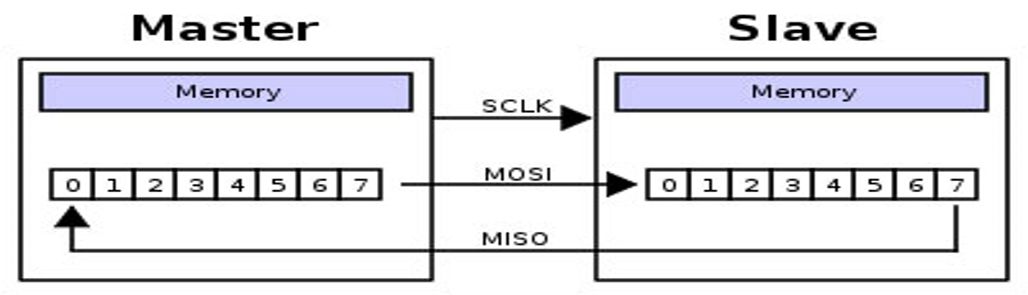


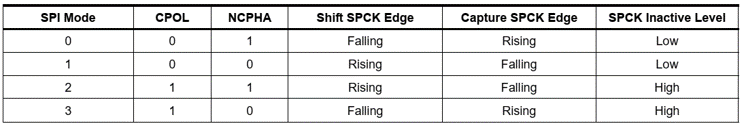
Note: The pins can also be used to for multiplexed with PIO lines.The programmer must first program the PIO controllers to assign the SPI pins to their peripheral functions.

## Functional Description

* **Master Mode**: MSTR = 1
  + The pins NPCS0 to NPCS3 are all configured as outputs.
  + SPCK pin is driven
  + The MISO line is wired on the receiver input
  + The MOSI line driven as an output by the transmitter
* **Slave Mode**: MSTR bit = 0 in the Mode Register (SPI\_MR)
  + The MISO line is driven by the transmitter output
  + The MOSI line is wired on the receiver input
  + The SPCK pin is driven by the transmitter to synchronize the receiver
  + The NPCS0 pin becomes an input, and is used as a Slave Select signal (NSS)
  + The pins NPCS1 to NPCS3 are not driven and can be used for other purposes

## Data Transfer

We should determine when the data is sent by the master and send it’s received by the slave in order to get the clean data. Our microcontroller can have 4 states:



## SPI Controller Register (Write-Only Register)

* SPIEN: SPI Enable
* SPIDIS: SPI Disable
* **If both SPIEN and SPIDIS are equal to one when the control register is written, the SPI is disabled**
* SWRST: SPI Software Reset
  + Reset the SPI. A software-triggered hardware reset of the SPI interface is performed.
  + **The SPI is in slave mode after software reset!!!**

## PSI Mode Register (Read-Write Register)

* **MSTR: Master/Slave Mode**
* PS: Peripheral Select
  + 0 = Fixed Peripheral Select
  + 1 = Variable Peripheral Select
* PCSDEC: Chip Select Decode
  + 0 = The chip selects are directly connected to a peripheral device.
  + 1 = The four chip select lines are connected to a 4- to 16-bit decoder (In this case the outputs are connected to a decoder)

## SPI Receive Data Register (Read-Only Register)

* RD: Receive Data

Data received by the SPI Interface is stored in this register right-justified.

* PCS: Peripheral Chip Select

In Master Mode only, these bits indicate the value on the NPCS pins at the end of a transfer.

## SPI Transmit Data Register (Write-Only Register)

* TD: Transmit Data

Data to be transmitted by the SPI Interface is stored in this register

* PCS: Peripheral Chip Select

This field is only used if Variable Peripheral Select is active

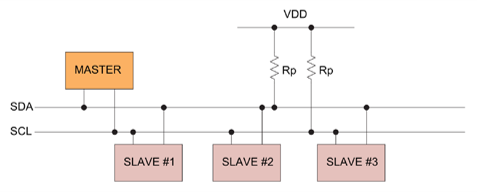
# Lecture 8 – I2C

* Serial Data Line (SDA)
* Serial Clock Line (SCL)

|  |  |
| --- | --- |
| **IPC** | **I2C** |
| Single-Master, Multi-Slave | Multi-Master, Multi-Slave |
| Short-range | Short-range |
| Simple and low cost | Simple and low cost |
| Low speed | Low speed |
| 4 communication wires | 2 communication wires |
| Synchronous | Synchronous |
| 4 pins – can support up to 15 devices | 2 wires – can support up to 128 devices  (Address is 7 Bits) |
| Mode: either master or slave (In any of the states, it could both send ***and*** receive) | Mode: either master or slave (In any of the states, it could both send ***or*** receive) |

The two resistors attach to the two wires are always 1 unless changed by the master (**Pull-up**)

* the Serial Clock Line (SCL) here is not the same as the microcontroller’s clock. It is actually much slower and used for transferring data.
* I2C is a two bidirectional open collector or open drain protocol
* **We can’t have multiple masters/slaves operating at the same time** (One at the time!) So only one master will generate clock rate the other master will just stand out. Also, master can send data to only one slave at the time.
* You can’t send and receive data at the same time. (IPC could!)



### How does it transfer to a specific master/slave?

It does so by sending the address of the target device using 7-bits (10-bits used rarely).

### Operating Roles

* Master: Generates the clock and initiates communication with slaves.
* Slave: Receives the clock and responds when addressed by a master.

Master and slave roles can change between messages.

## I2C Modes

* Master send – master node is sending data to a slave
* Master receive – master node is receiving data from a slave
* Slave transmit – slave node is sending data to the master
* Slave receive – slave node is receiving data from the master

## Data Handshake Sequence

START and STOP signals are different from data bits!

1. Master enables clock
2. Master sends START signal that wakes slaves in order to listen to the master (By **zeroing the data line (Writing a zero in the flow**)
3. Master then sends 7-bits that identifies who is the slave receiver. (Bits number **1** to **8**)
4. Master sends a bit that identifies where it wants to read (**1**) or write (**0**) which is Bit number **9**.
5. Master will stand by for slave response ACK bit (active low for acknowledged). Salve sends a **0**.
6. (*If slave exists and responds*) Data is sent with its **Most Significant Bit** at first. (By contract)

## Data Transfer Sequence

* The start bit is indicated by a **high-to-low** transition of SDA (with **SCL high**)
* The stop bit is indicated by a **low-to-high** transition of SDA (with **SCL high**)
* All other transitions of SDA take place with **SCL low** (W/R, ACK, Address, …)
* To write to the slave
  + The Master repeatedly sends a byte and salve reads it by sending an ACK bit back to master. (Until master sends STOP Bit)
* To read from slave
  + Master repeatedly receives a byte from the slave, and send an ACK bit after every byte it receives **except the last one!** (Equals to sending STOP signal)(If master doesn’t want any more data, doesn’t send the last ACK bit)

## Two-wire interface (TWI)

This is equivalent to I2C protocol (I2C-Compatible interface).

* We can configure two pins of the PIO to act as I2C Interface.

# Lecture 9 – UART

|  |  |  |
| --- | --- | --- |
| **IPC** | **I2C** | **UART** |
| Single-Master, Multi-Slave | Multi-Master, Multi-Slave | Single-Master, Single-Slave |
| Short-range | Short-range |  |
| Simple and low cost | Simple and low cost |  |
| Low speed | Low speed |  |
| 4 communication wires | 2 communication wires (Data and Clock) | 2 communication wires (Both Data) |
| Synchronous | Synchronous | Asynchronous |
| 4 pins – can support up to 15 devices | 2 wires – can support up to 128 devices  (Address is 7 Bits) |  |
| Mode: either master or slave (In any of the states, it could both send ***and*** receive) | Mode: either master or slave (In any of the states, it could both send ***or*** receive) |  |

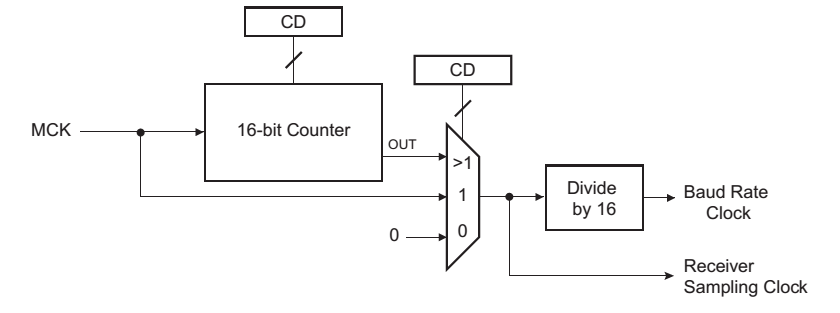
* UART can transfer data with only **one wire**.
* There is no **clock** in UART.
* Like I2C, UART has START and STOP signals
* UART functions like I2C in terms of being pull-up (Always 1 until proven otherwise)
* Transmitter and Receiver operate completely independent of each other, but have the same **Baud Rate**.
* UART cannot work without clock (PIO had some functionality though)
* Our microcontroller only supports 8 bits (with parity)

Baud Rate

* The frequency (rate) of sending bits in to the network. (**Baud Rate ≠ Clock Rate**)
* **Baud Generator**, converts the UART internal clock to a **lower** external clock for communication.
* Maybe you want to send bits every 10 clocks, so your *Baud Rate* is 1/10 of *Clock Rate.*
* Each device must adjust its clock to the *Baud Rate* in order to communicate. (Baud Rate can differ about 10%, not more)
* **Baud Rate** is calculated by where **CD** is defined by us, **MCK (Master Clock)** is from microcontroller. The result is stored as 16-bits in **UART\_BRGR** register.
* If **UART\_BRGR** is set to 0, the Baud Rate is disabled and UART is inactive. (The maximum is MCK devided by 16, or CD = 1. Minimum is when CD = 65536 which is the slowest)

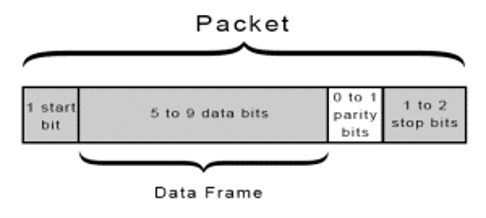
## Baud Rate Generator

Output of Baud Rate Generator are two signals:

* Baud Rate Clock (For transmitter)
* Receiver Sampling Clock (For receiver): which is 16 times faster than transmitter because it’s not devided by 16.

## Communication

1. First bit is the START bit (Zeroing the signal – Pullup keeps the signal at 1, we change it to zero)
2. The data bits can be 5-9 bits which is a contract between the two UARTs (Must be adjusted)
3. **Parity bit** is for identifying any errors.
4. 1 or 2 bits for STOP.



When UART is not transmitting, the wire is high (Pull-up)

### START Bit

The transmitting UART pulls the line from high to low for one clock cycle. (Shared clock!)

### STOP Bit

The sending UART drives the data transmission line from a low voltage to a high voltage for at least two-bit durations.

## Parity Bit

Can be **Evan**, **Odd**, **Mark** or **Space**. (Configurable).

* Space (0) or Mark (1), means the bit is always 0 or 1 regardless of what the data is.

### When is it used?

For example, we want to send STOP signal that is a **high** signal. To make sure we always have **0** before the STOP signal, we use parity of Mask.

### What can it detect?

* Framing: When the data is supposed to be 8 bits but it is actually not.
* Overrun: Before or while the process is reading the **UART\_RHR** since last transfer, another byte is received from UART and then the shift register automatically moves it into the **UART\_RHR**; Meaning the data is now override and old one is lost. RIP!
  + Then the **OVER** status bit in **UART\_SR** is set and an interrupt is sent to the process. Software clears **OVER** after it writes **RSTSA (Reset status)** at 1 for **UART\_CR**. The UART is then reset and overrun is stopped.

## UART Receiver in SAM3X8E

* Receiver **Reset**, **Enable** and **Disable**
  + **After device reset, the UART receiver is disabled**
  + To enable: RXEN = 1 in the control register (UART\_CR)
  + If enabled: the receiver starts looking for a START bit
  + To disable: RXDIS = 1 in the control register (UART\_CR)
    - **If the receiver is waiting for a start bit, it is immediately stopped**
    - **If the receiver has already detected a start bit and is receiving the data, it waits for the stop bit before actually stopping its operation**
  + To reset: RSTRX =1 in the control register (UART\_CR)
    - **The receiver immediately stops its current operations and is disabled, whatever its current state!**

### Start Detection and Data Sampling

* A low level (space) on URXD is interpreted as a valid start bit
  + if it is detected for more than 7 cycles of the sampling clock
  + Sampling clock = 16 x baud rate (In receive)

We wait for **half a clock** for the signal to stay low (be START) and then start communicating (*Why? There might be a shock, error, bounce and …*)

* The receiver samples the URXD at the theoretical **midpoint** of each bit
  + When a valid start bit has been detected
  + It is assumed that each bit lasts 16 cycles of the sampling clock (1-bit period): the bit sampling point is **eight** **cycles** (0.5-bit period) after the start of the bit.
  + The first sampling point is therefore 24 cycles (1.5-bit periods) after the falling edge of the start bit was detected.
  + Each subsequent bit is sampled 16 cycles (1-bit period) after the previous one.

### Receiver Ready

After 8 bits of data has been received, 8 bits are then moved from shift register to **UART\_RHR** that is a holding register. The **RXRDY** status bit in **UART\_SR** (Status Register) is set. **RXRDY** sends an interrupt to the processor and executes the ISR. Then ISR reads the 8 bits from **UART\_RHR** and then **RXRDY** is automatically cleared.

## Transmitter

* The UART transmitter is disabled after device reset
  + To enable: TXEN = 1 in the control register UART\_CR
    - The transmitter waits for a character to be written in the Transmit Holding Register (UART\_THR which is **NOT a shift register**) before actually starting the transmission
* To disable: TXDIS = 1 in the control register UART\_CR
  + **If the transmitter is not operating, it is immediately stopped**
  + **If a character is being processed into the Shift Register and/or a character has been written in the Transmit Holding Register, the characters are completed before the transmitter is stopped.**
* To reset: RSTTX = 1 in the UART\_CR

### Transmitter control

* When enabled, the bit **TXRDY** is set in the **UART\_SR**
* The transmission starts when
  + The programmer writes in the **UART\_THR** and
  + The written character is transferred from **UART\_THR** to the **Shift Register**
  + The **TXRDY** bit remains high until a second character is written in **UART\_THR**
* As soon as the first character is completed
  + The last character written in **UART\_THR** is transferred into the **shift register** and **TXRDY** rises again
    - Showing that the holding register is empty
* When both the Shift Register and **UART\_THR** are empty
  + All the characters written in **UART\_THR** have been processed
  + The **TXEMPTY** bit rises after the last stop bit has been completed