

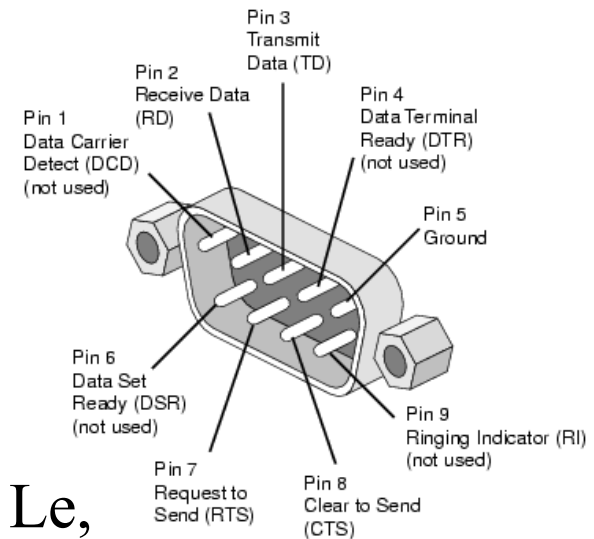
EECS 373

Design of Microprocessor-Based Systems

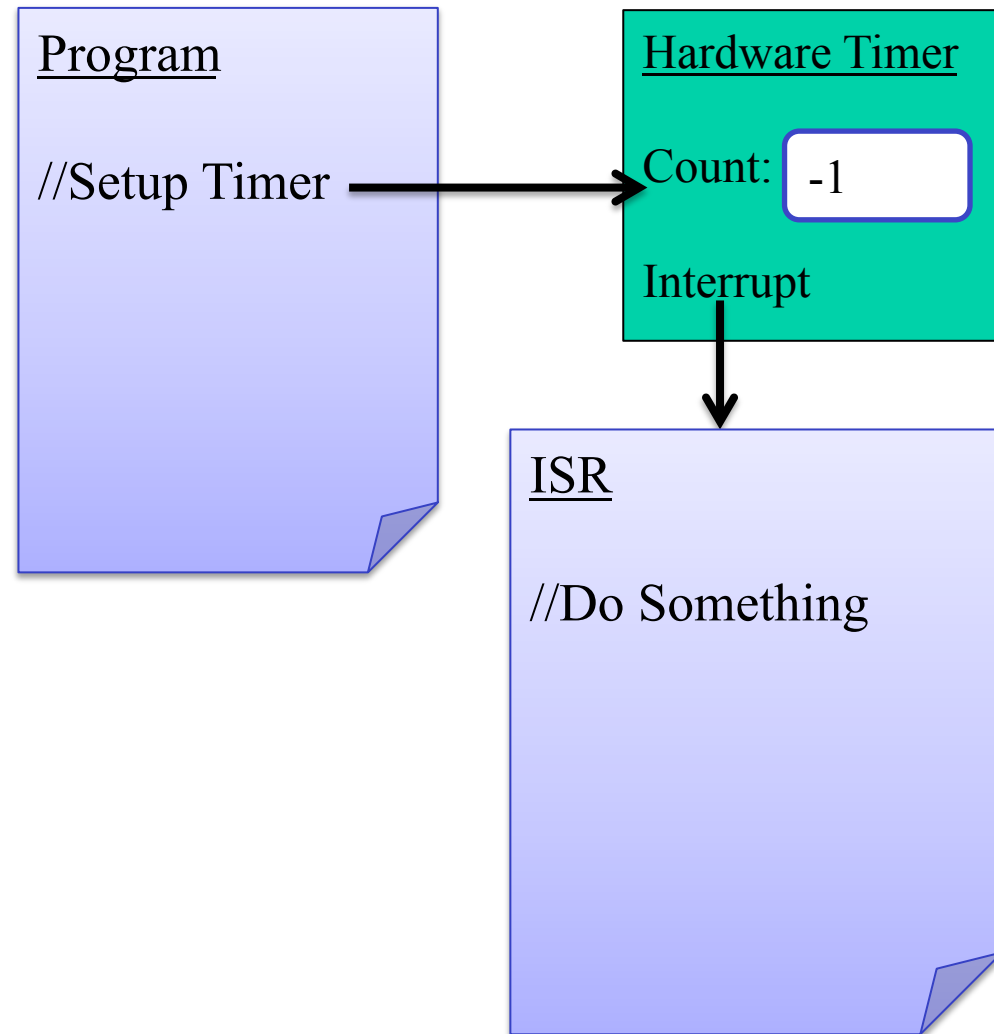
Ron Dreslinski
University of Michigan

Serial buses, digital design

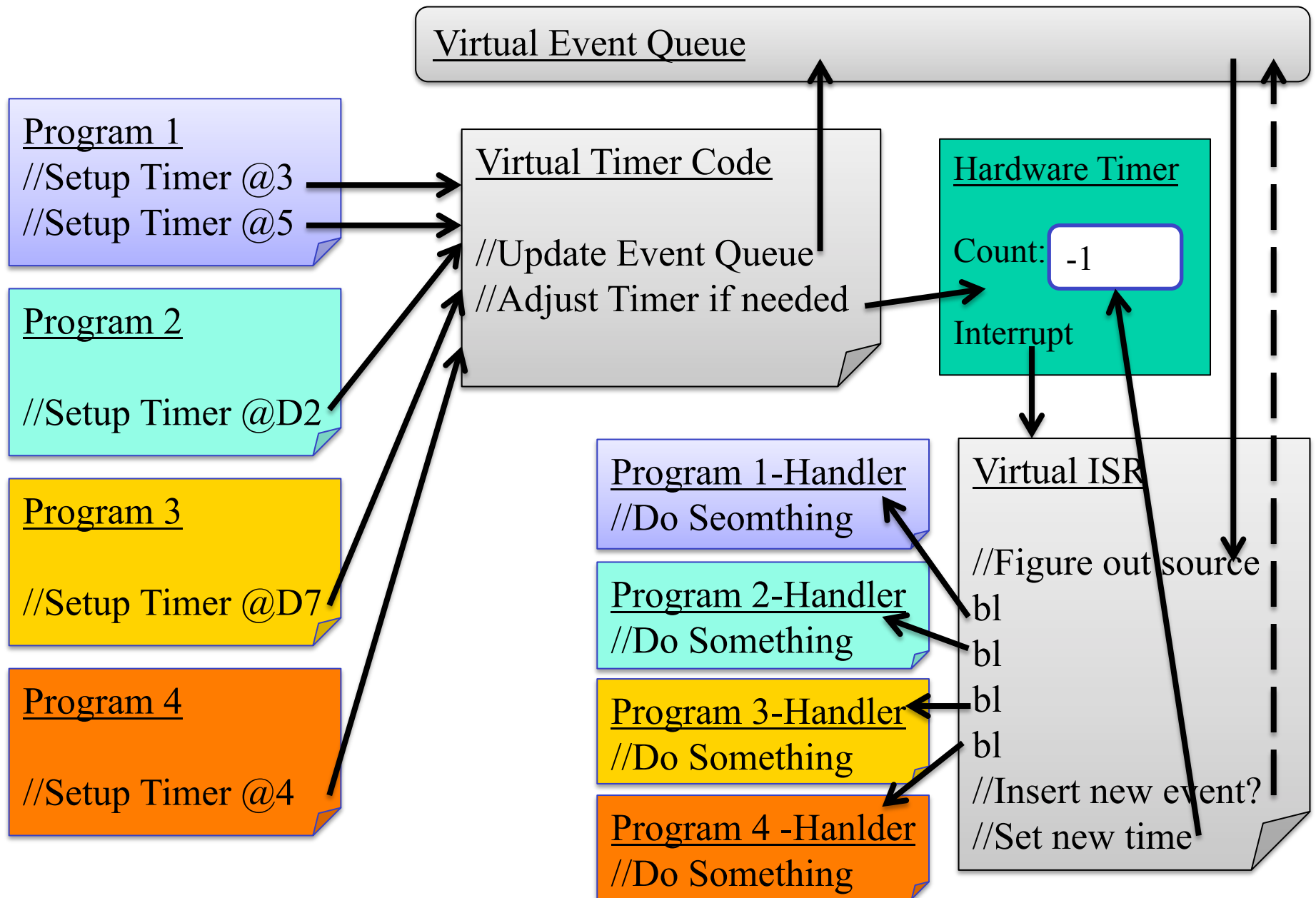
Material taken from Brehob, Dutta, Le,
Ramadas, Tikhonov & Mahal



Timer



Virtual Timer



Event Queue



Program 1

//Setup Timer @3

//Setup Timer @5

Program 2

//Setup Timer @D2

Program 3

//Setup Timer @D7

Program 4

//Setup Timer @4

Head Ptr



Hardware Timer

Count: -1

Event Queue



Program 1

//Setup Timer @3

//Setup Timer @5

Program 2

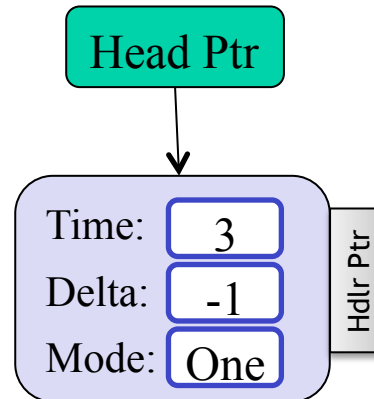
//Setup Timer @D2

Program 3

//Setup Timer @D7

Program 4

//Setup Timer @4



Hardware Timer

Count: 3

Event Queue

Program 1

//Setup Timer @3

//Setup Timer @5

Program 2

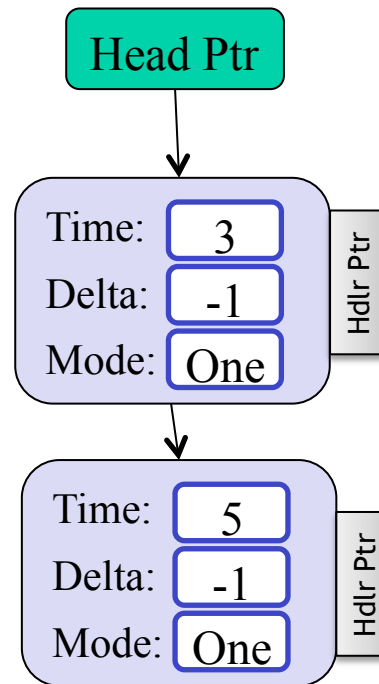
//Setup Timer @D2

Program 3

//Setup Timer @D7

Program 4

//Setup Timer @4



Hardware Timer

Count: 3

Event Queue

Program 1

//Setup Timer @3
//Setup Timer @5

Program 2

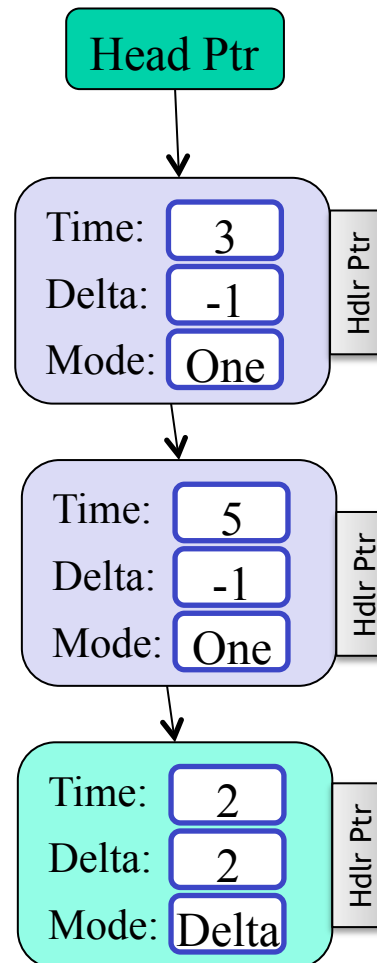
//Setup Timer @D2

Program 3

//Setup Timer @D7

Program 4

//Setup Timer @4



Hardware Timer

Count: 3

Event Queue

Program 1

//Setup Timer @3
//Setup Timer @5

Program 2

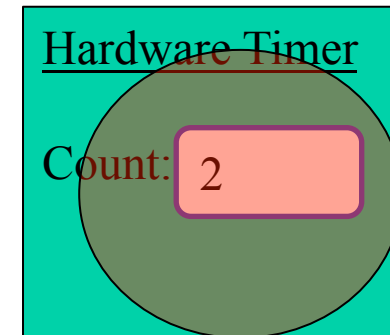
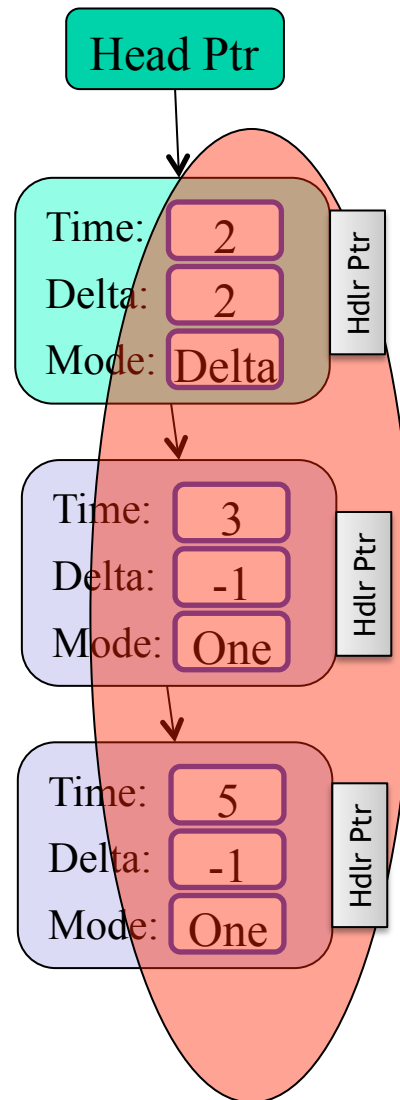
//Setup Timer @D2

Program 3

//Setup Timer @D7

Program 4

//Setup Timer @4



Event Queue



Program 1

//Setup Timer @3
//Setup Timer @5

Program 2

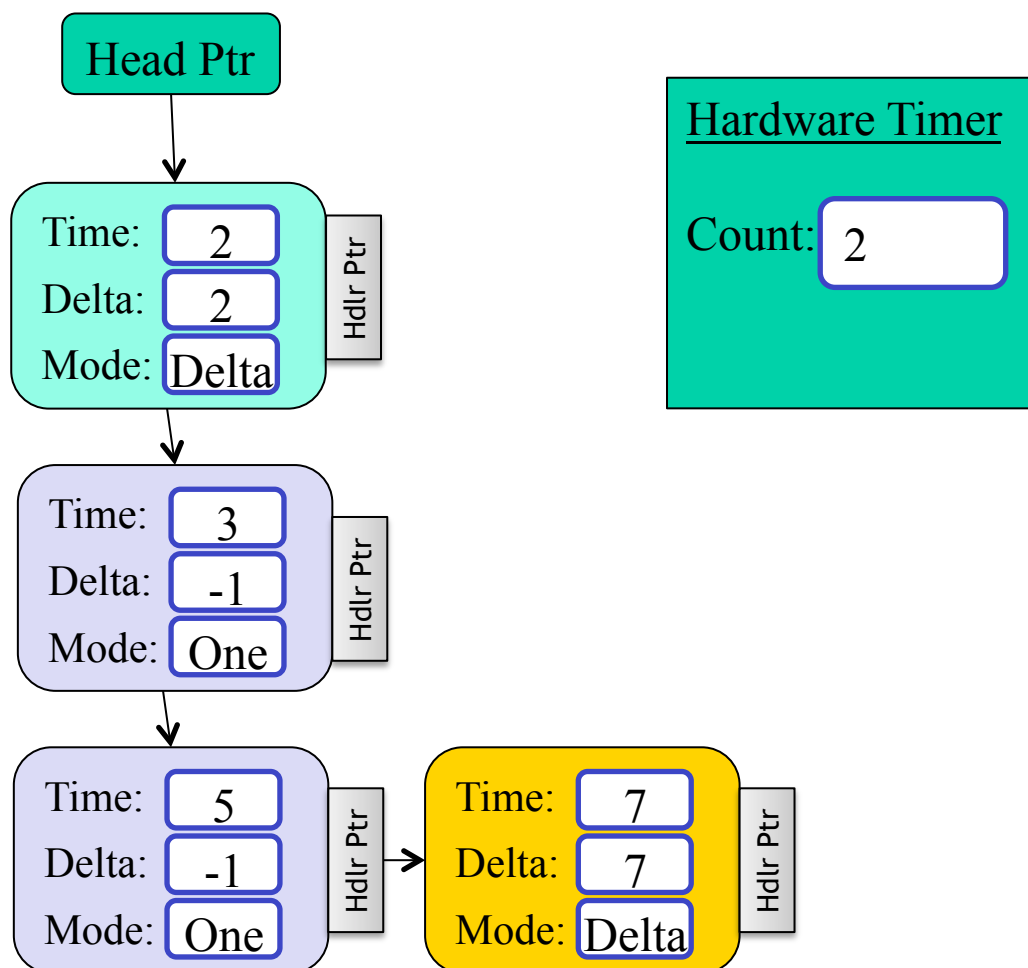
//Setup Timer @D2

Program 3

//Setup Timer @D7

Program 4

//Setup Timer @4



Event Queue



Program 1

//Setup Timer @3
//Setup Timer @5

Program 2

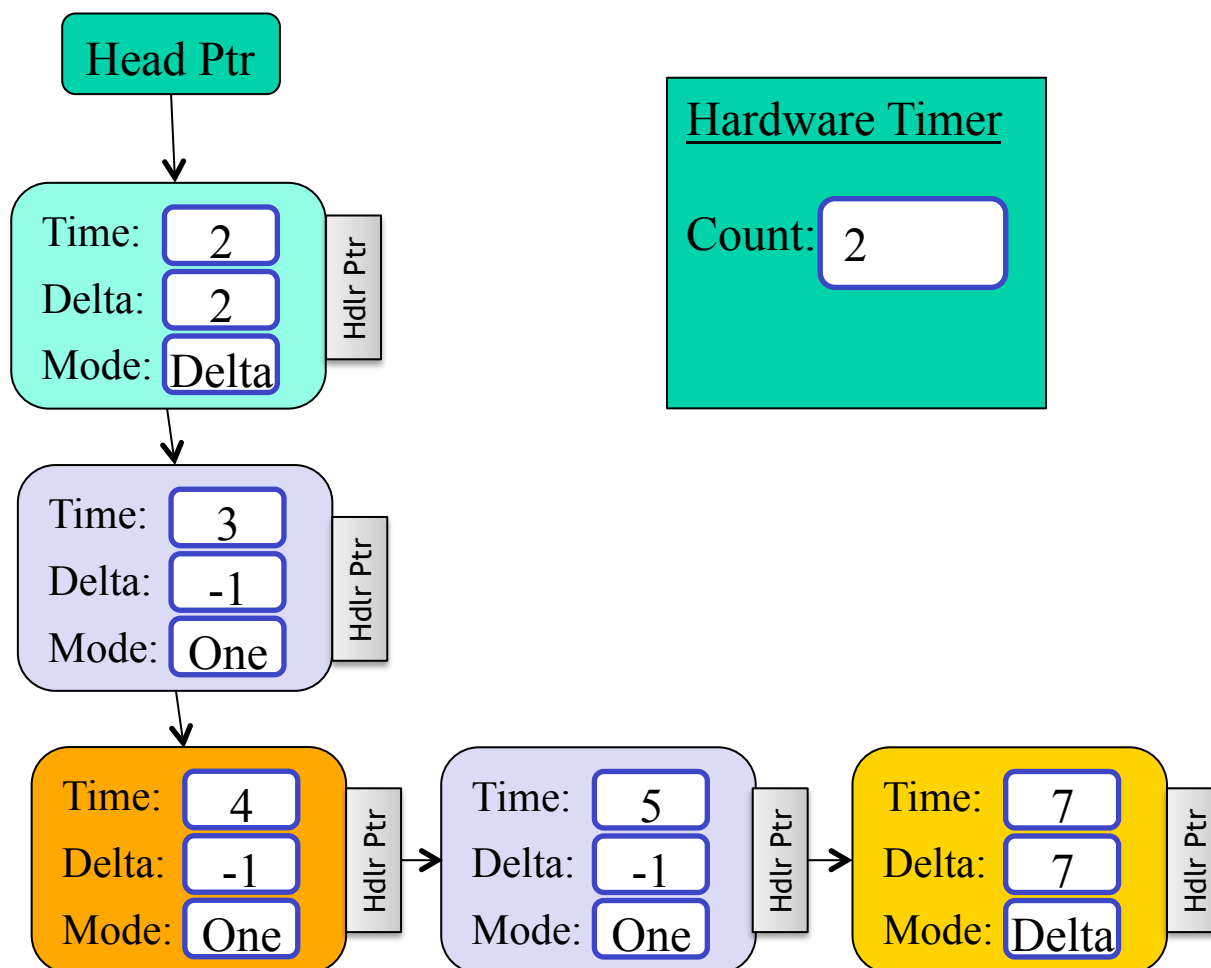
//Setup Timer @D2

Program 3

//Setup Timer @D7

Program 4

//Setup Timer @4



Event Queue



Program 1

//Setup Timer @3
//Setup Timer @5

Program 2

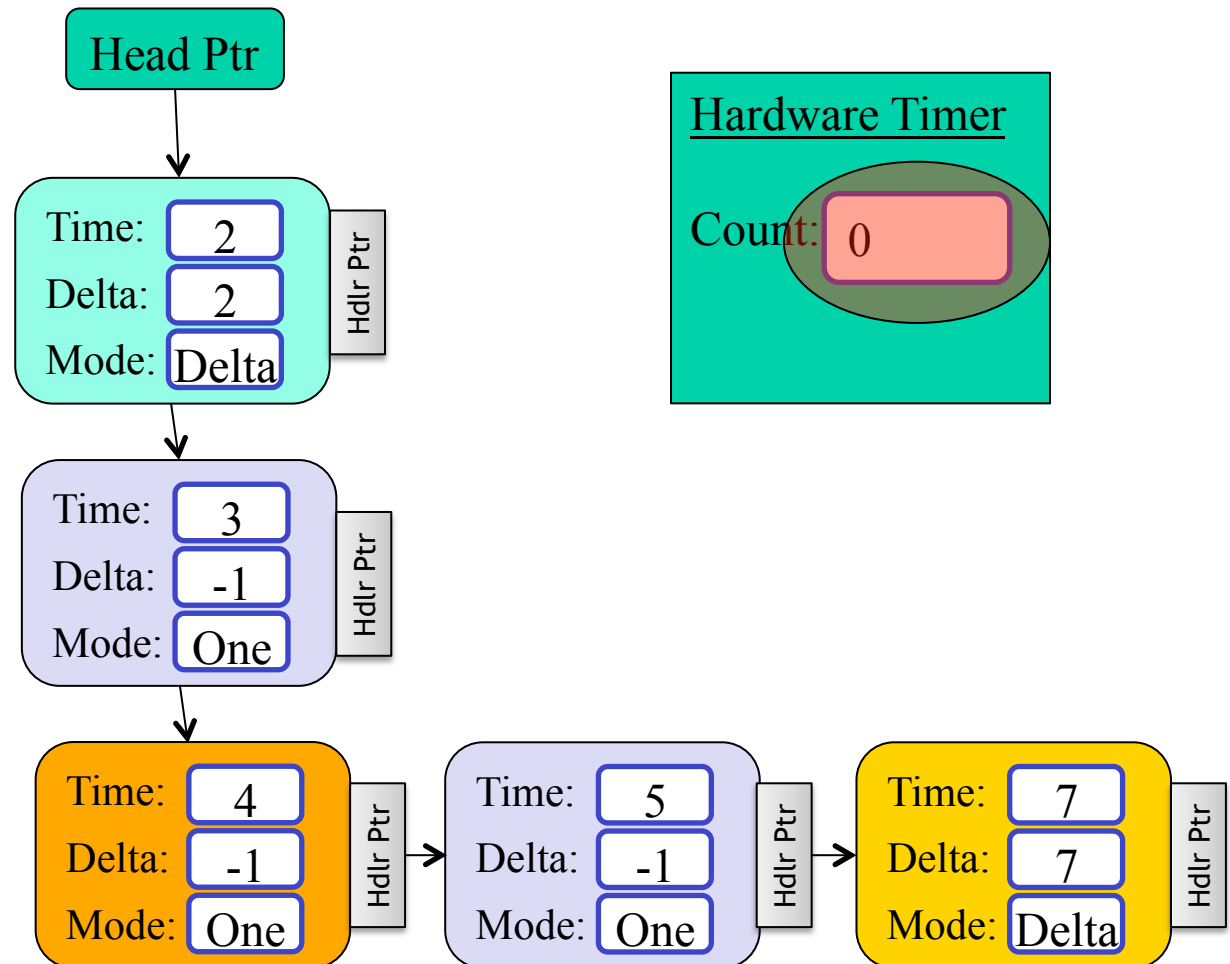
//Setup Timer @D2

Program 3

//Setup Timer @D7

Program 4

//Setup Timer @4



Event Queue



Program 1

//Setup Timer @3
//Setup Timer @5

Program 2

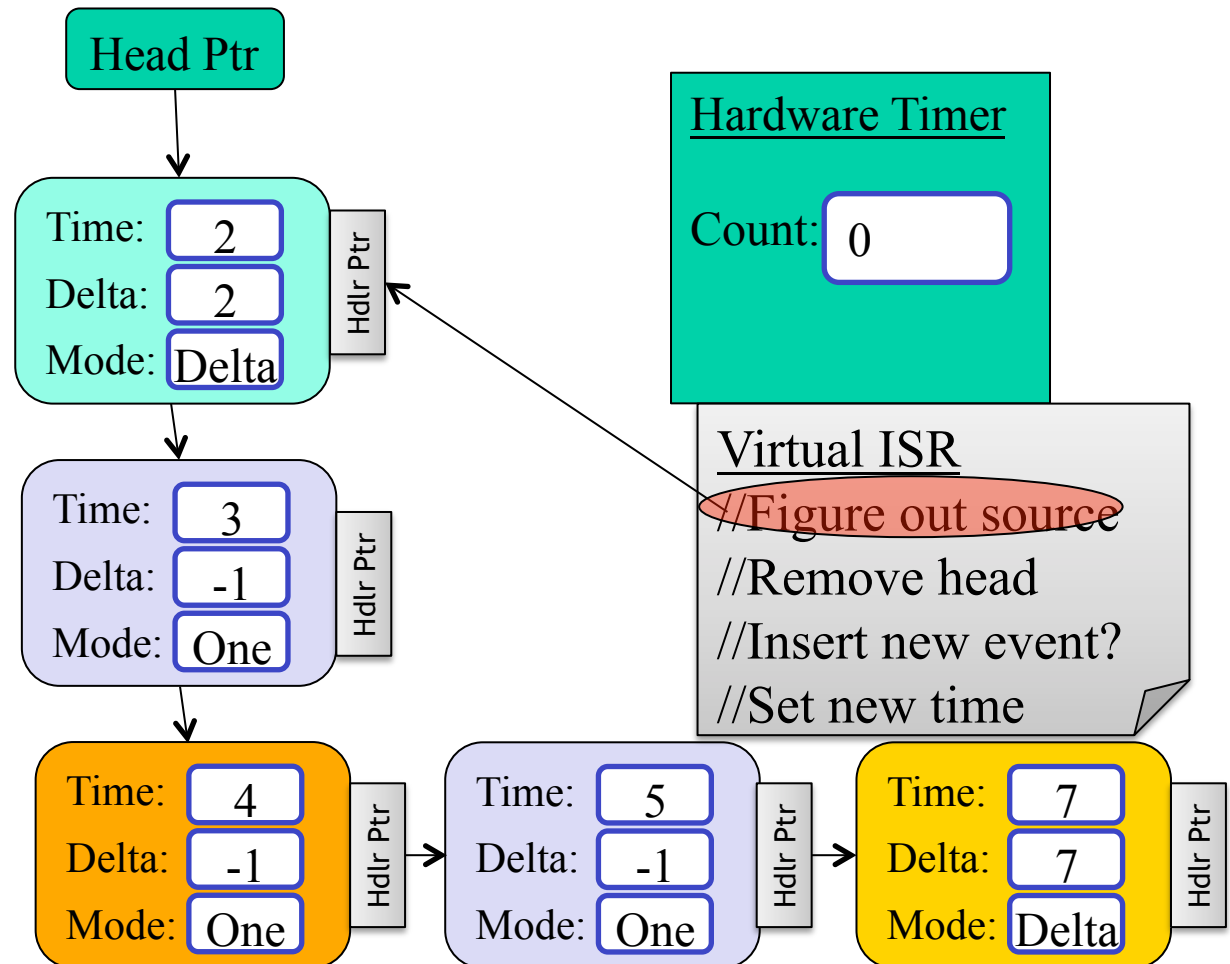
//Setup Timer @D2

Program 3

//Setup Timer @D7

Program 4

//Setup Timer @4



Event Queue

Program 1

//Setup Timer @3
//Setup Timer @5

Program 2

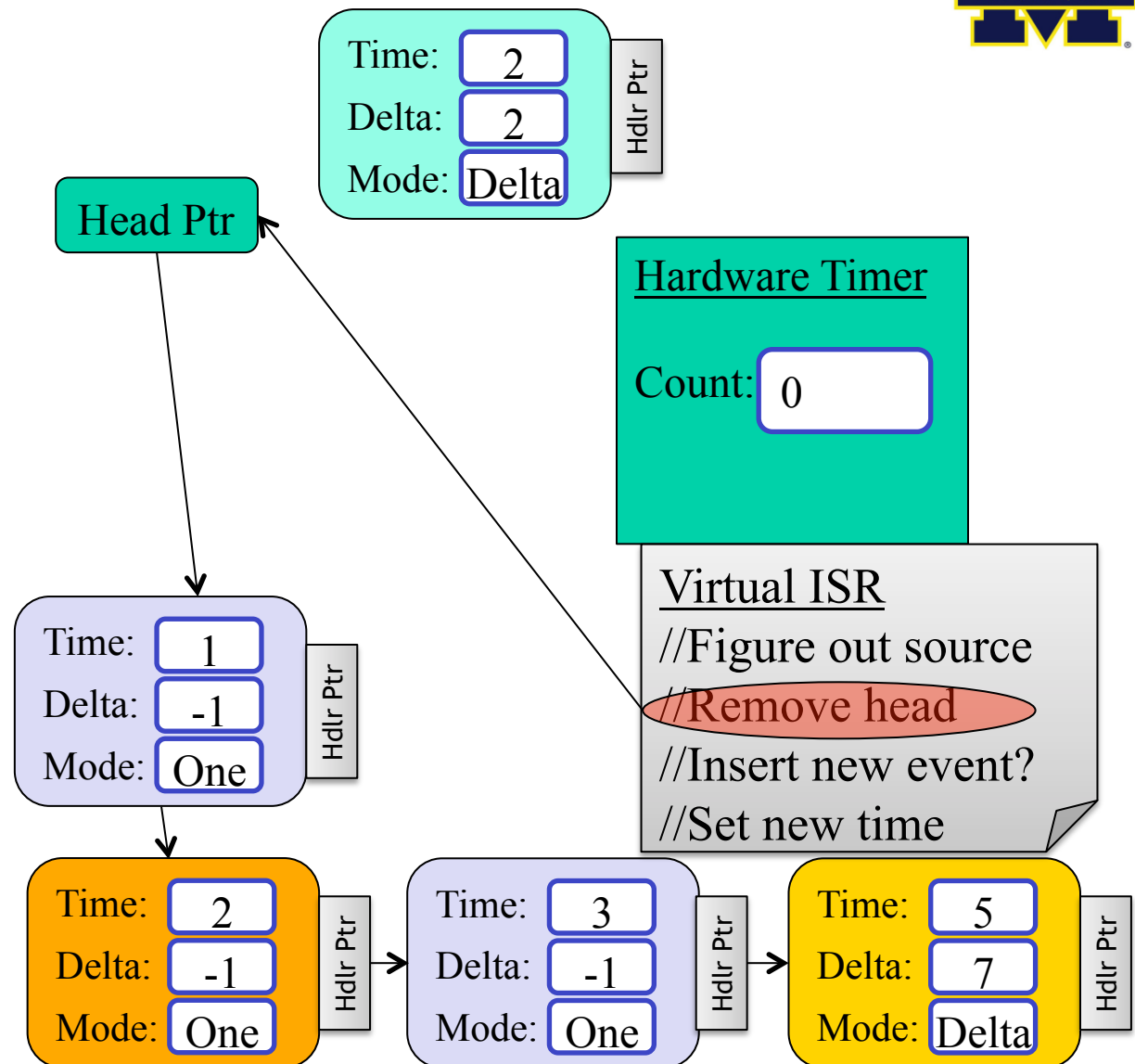
//Setup Timer @D2

Program 3

//Setup Timer @D7

Program 4

//Setup Timer @4



Event Queue



Program 1

//Setup Timer @3
//Setup Timer @5

Program 2

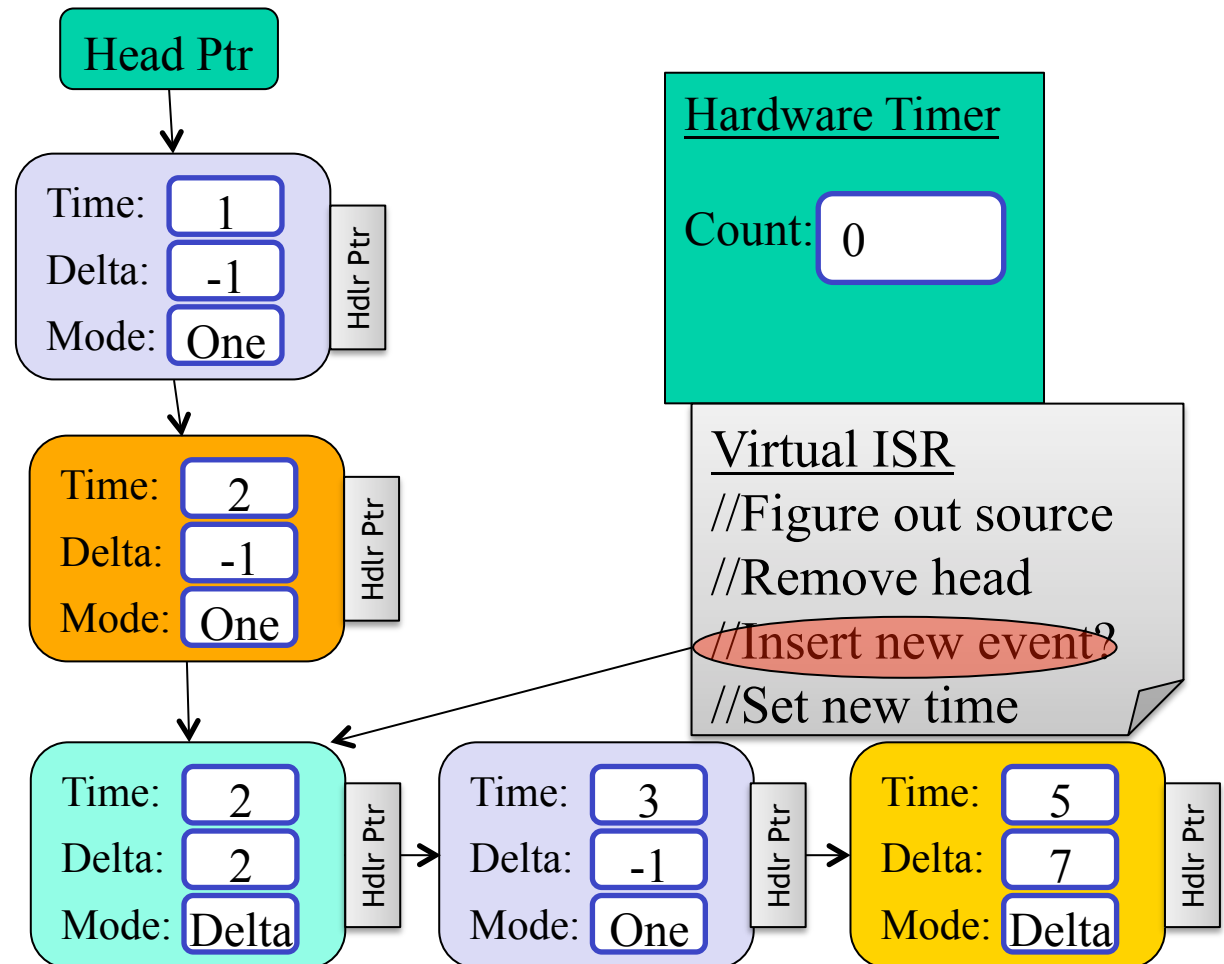
//Setup Timer @D2

Program 3

//Setup Timer @D7

Program 4

//Setup Timer @4



Event Queue



Program 1

//Setup Timer @3
//Setup Timer @5

Program 2

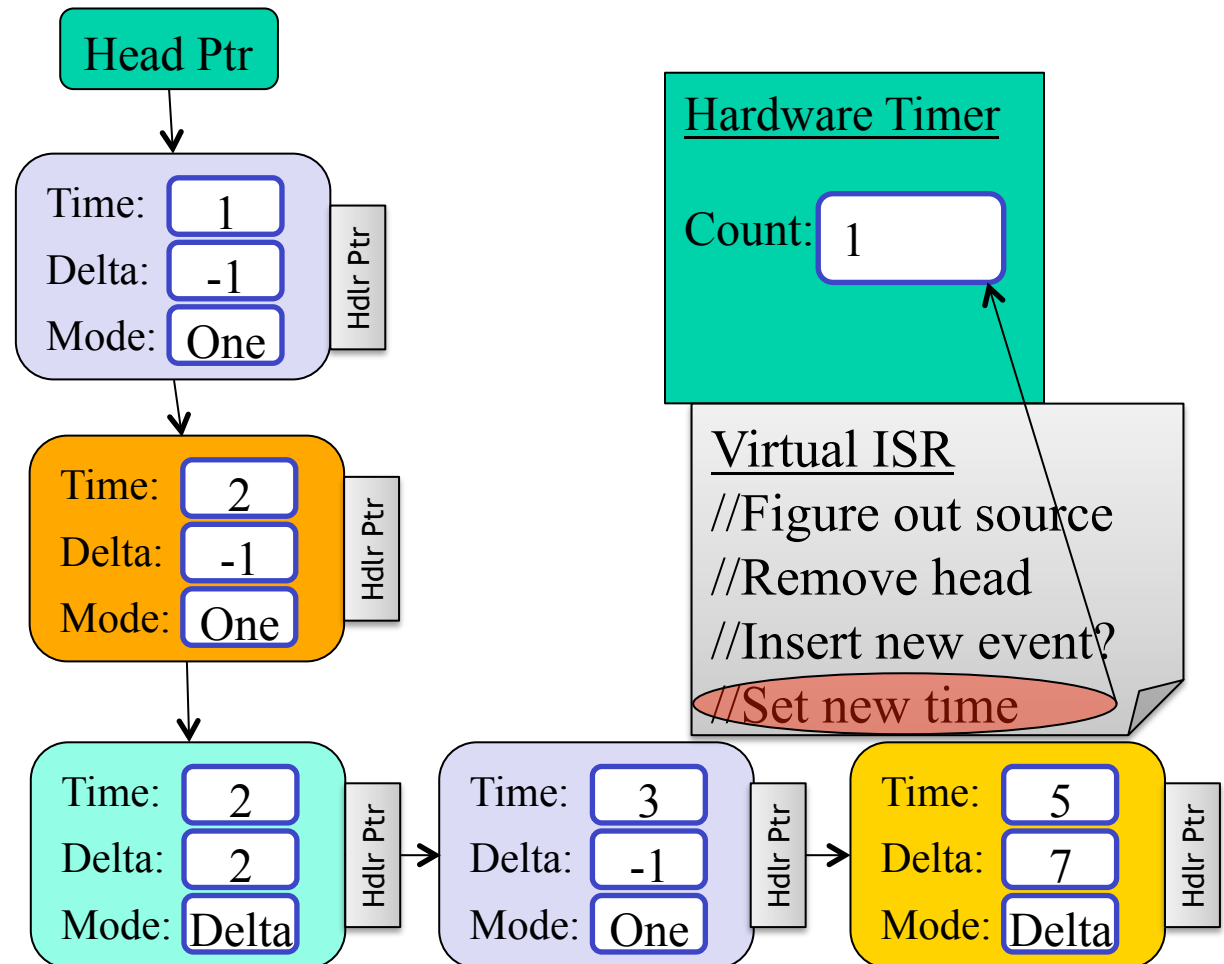
//Setup Timer @D2

Program 3

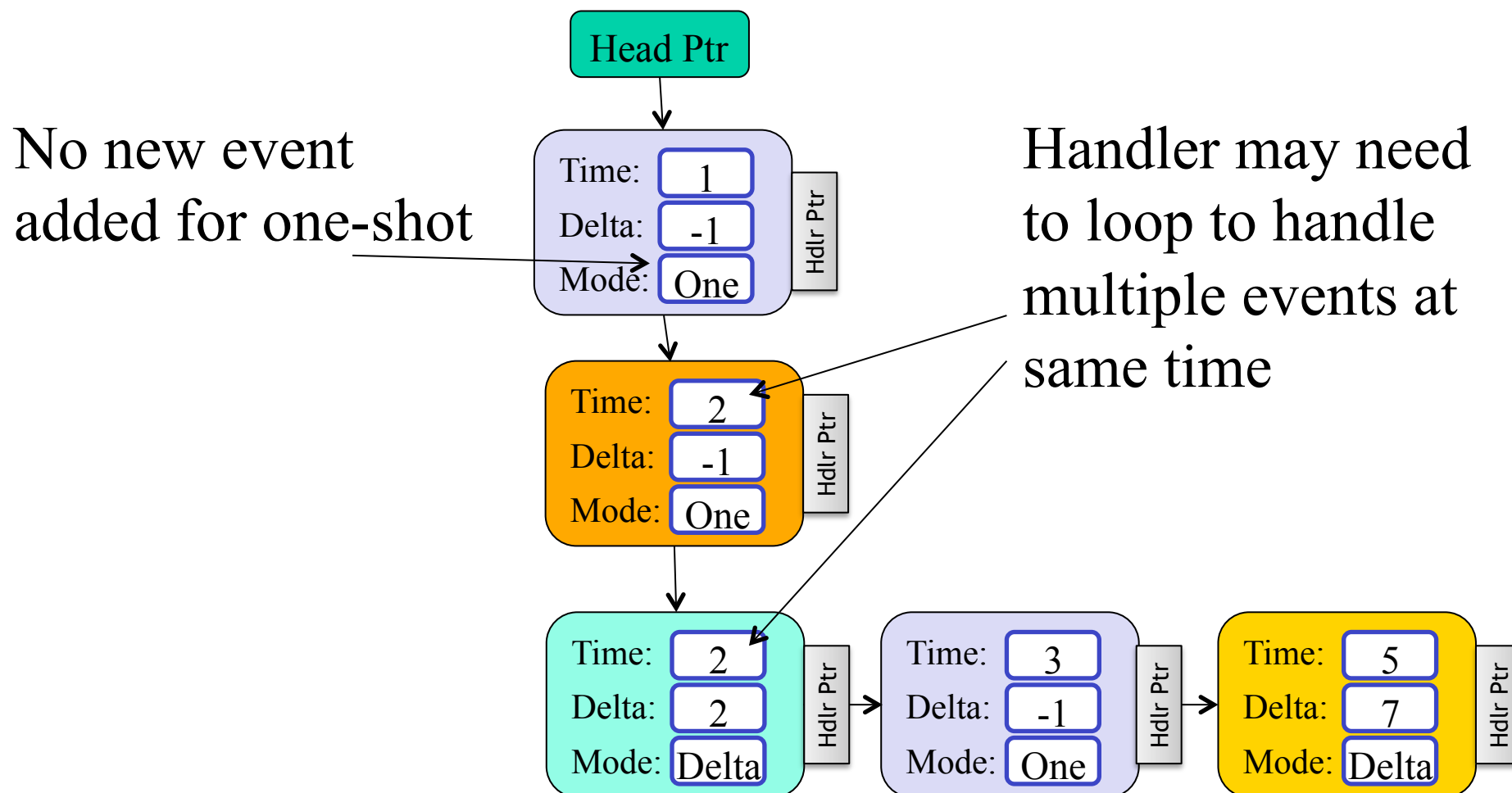
//Setup Timer @D7

Program 4

//Setup Timer @4



Event Queue - Caveats

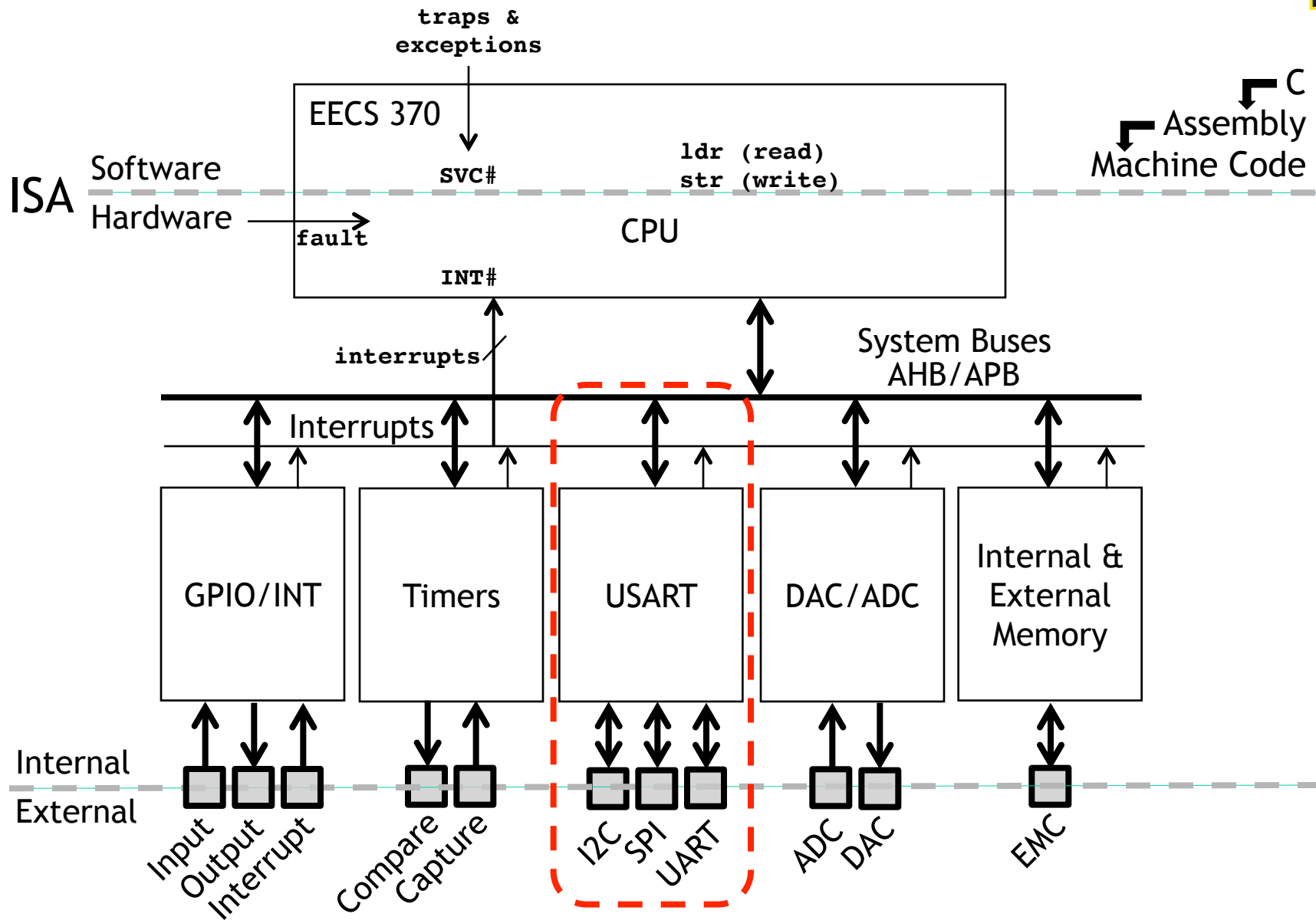


Agenda

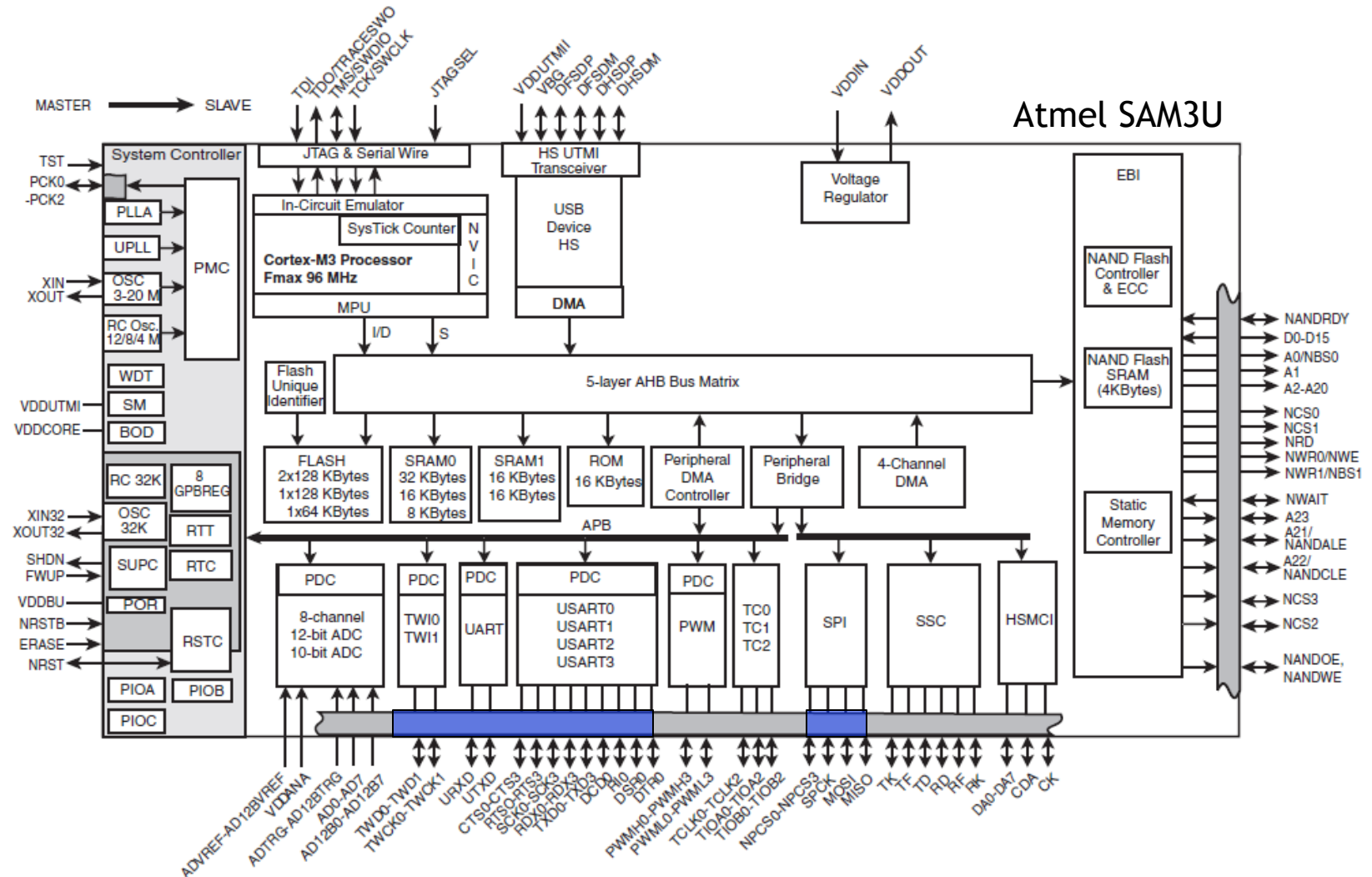


- Serial Buses
 - Introduction
 - UART
 - SPI
 - I2C
- Glitches
 - Asynchronous resets and glitches
 - Design rules
- Set-up and hold time.
 - Review
 - Dealing with external inputs
 - Design rules

Serial interfaces



External memory attaches to the processor via the external memory controller and bus

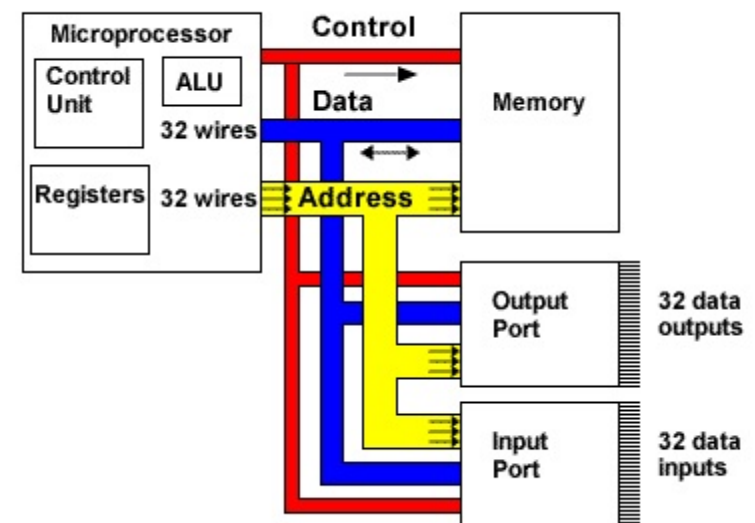


- Universal Asynchronous Receiver/Transmitter
 - a type of "asynchronous receiver/transmitter", a piece of computer hardware that translates data between parallel and serial forms.
 - UARTs are commonly used in conjunction with communication standards such as EIA, RS-232, RS-422 or RS-485.
 - The universal designation indicates that the data format and transmission speeds are configurable and that the actual electric signaling levels and methods (such as differential signaling etc.) typically are handled by a special driver circuit external to the UART.

Fun with buses



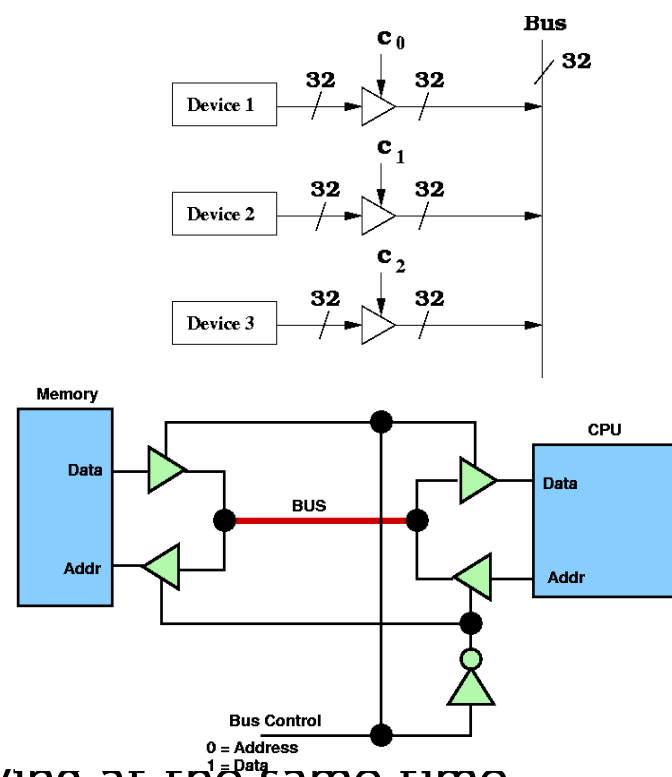
- A multidrop bus (MDB) is a computer bus in which all components are connected to the same set of electrical wires. (from Wikipedia)
 - In the general case, a bus may have more than one device capable of driving it.
 - That is, it may be a “multi-master” bus as discussed earlier.



How can we handle multiple (potential) bus drivers? (1/3)



- Tri-state devices, just have one device drive at a time. Everyone can read though
 - Pros:
 - Very common, fairly fast, pin-efficient.
 - Cons:
 - Tri-state devices can be slow.
 - Especially drive-to-tristate?
 - Need to be sure two folks not driving at the same time
 - Let out the magic smoke.
 - Most common solution (at least historically)
 - Ethernet, PCI, etc.



How can we handle multiple (potential) bus drivers? (2/3)

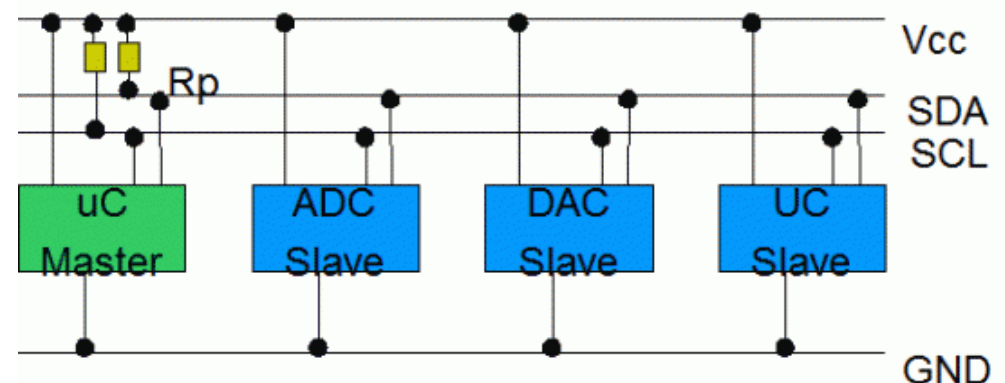
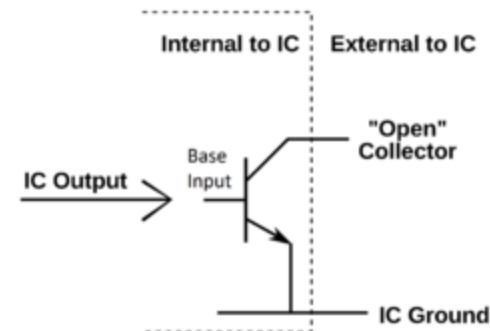


- MUX
 - Just have each device generate its data, and have a MUX select.
 - That's a LOT of pins.
 - Consider a 32-bit bus with 6 potential drivers.
 - » Draw the figure.
 - » How many pins needed for the MUX?
 - Not generally realistic for an “on-PCB” design as we’ll need an extra device (or a lot of pins on one device)
 - But reasonable on-chip
 - In fact AHB, APB do this.

How can we handle multiple (potential) bus drivers? (3/3)



- “pull-up” aka “open collector” aka “wired OR”
 - Wire is pulled high by a resistor
 - If any device pulls the wire low, it goes low.
- Pros:
 - If two devices both drive the bus, it still works!
- Cons:
 - Rise-time is very slow.
 - Constant power drain.
- Used in I2C, CAN



Agenda



- Serial Buses
 - Introduction
 - UART
 - SPI
 - I2C
- Glitches
 - Asynchronous resets and glitches
 - Design rules
- Set-up and hold time.
 - Review
 - Dealing with external inputs
 - Design rules

UART

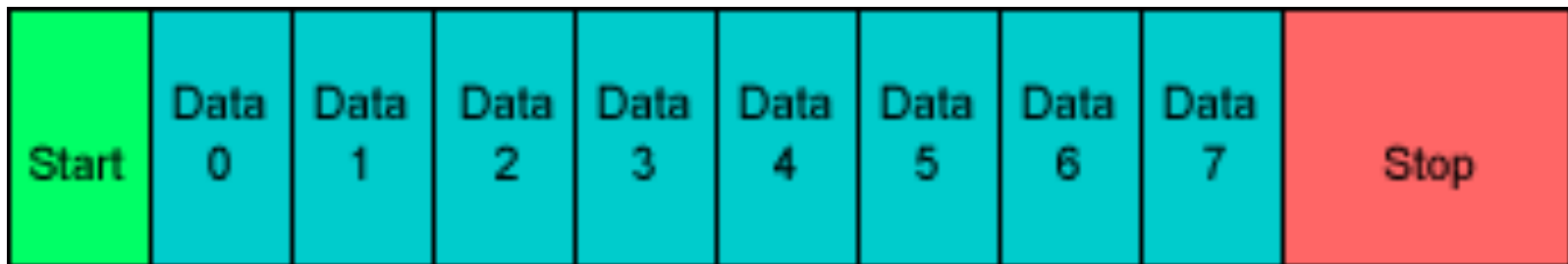


- Universal Asynchronous Receiver/Transmitter
- Hardware that translates between parallel and serial forms
- Commonly used in conjunction with communication standards such as EIA, RS-232, RS-422 or RS-485
- The universal designation indicates that the data format and transmission speeds are configurable and that the actual electric signaling levels and methods (such as differential signaling etc.) typically are handled by a special driver circuit external to the UART.

Protocol



- Each character is sent as
 - a logic *low* start bit
 - a configurable number of data bits (usually 7 or 8, sometimes 5)
 - an optional parity bit
 - *one or more logic high* stop bits.
 - with a particular bit timing (“baud” or “baudrate”)
- Examples
 - “9600-N-8-1” → <baudrate><parity><databits><stopbits>
 - “9600-8-N-1” → <baudrate><databits><parity><stopbits>



Variations and fun times



- UART is actually a generic term that includes a large number of different devices/standards.
 - RS-232 is a standard that specifies
 - “electrical characteristics and timing of signals, the meaning of signals, and the physical size and pin out of connectors.

Signals (only most common)

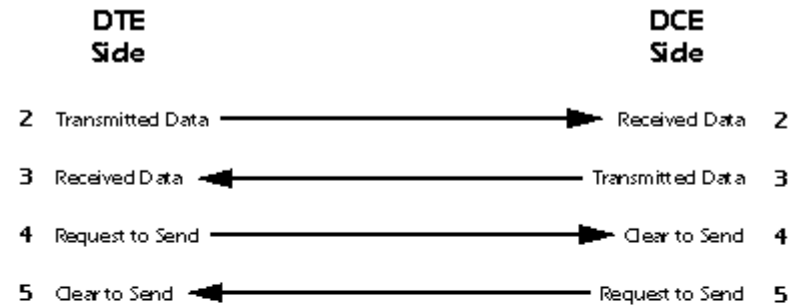
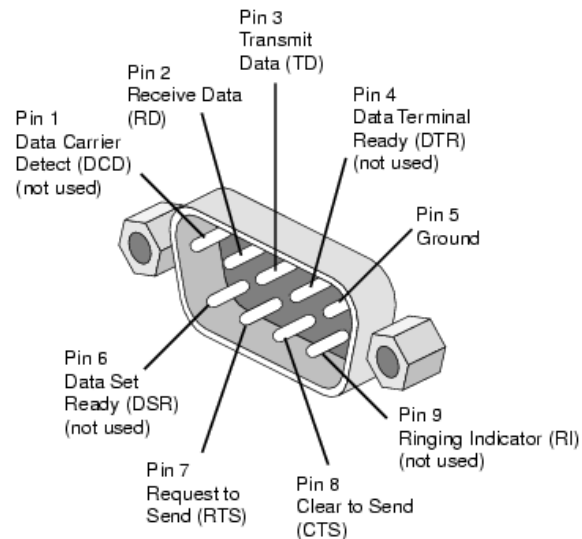


- The **RXD** signal of a UART is the signal receiving the data. This will be an input and is usually connected to the TXD line of the downstream device.
- The **TXD** signal of a UART is the signal transmitting the data. This will be an output and is usually connected to the RXD line of the downstream device.
- The **RTS#** (Ready to Send) signal of a UART is used to indicate to the downstream device that the device is ready to receive data. This will be an output and is usually connected to the CTS# line of the downstream device.
- The **CTS#** (Clear to Send) signal of a UART is used by the downstream device to identify that it is OK to transmit data to the upstream device. This will be an input and is usually connected to the RTS# line of the upstream device.

DB9 stuff



- DTE vs DCE
- Pinout of a DCE?
- Common ground?
- Noise effects?



Pin Number	Signal	Description
1	DCD	Data carrier detect
2	RxD	Receive Data
3	TxD	Transmit Data
4	DTR	Data terminal ready
5	GND	Signal ground
6	DSR	Data set ready
7	RTS	Ready to send
8	CTS	Clear to send
9	RI	Ring Indicator

Wiring a DTE device to a DCE device for communication is easy.

The pins are a one-to-one connection, meaning all wires go from pin x to pin x.

A straight through cable is commonly used for this application.

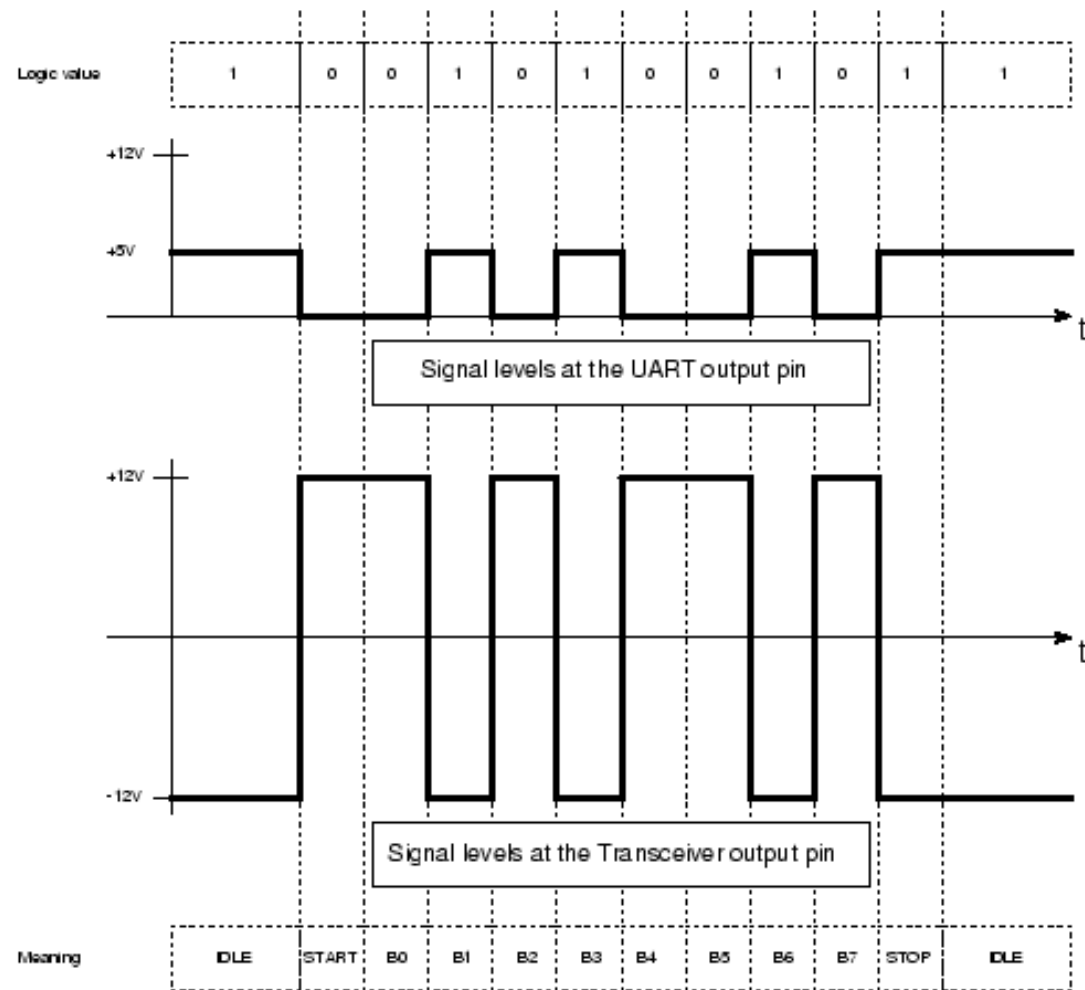
In contrast, wiring two DTE devices together requires crossing the transmit and receive wires.

This cable is known as a null modem or crossover cable.

RS-232 transmission example



RS232 Transmission of the letter 'J'



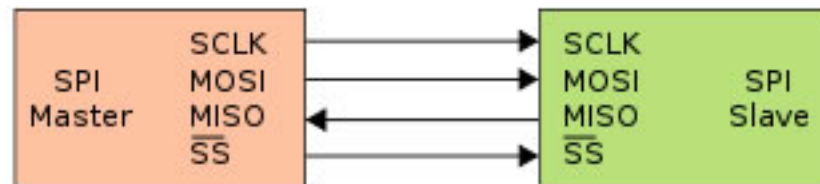
Agenda



- Serial Buses
 - Introduction
 - UART
 - SPI
 - I2C
- Glitches
 - Asynchronous resets and glitches
 - Design rules
- Set-up and hold time.
 - Review
 - Dealing with external inputs
 - Design rules

Introduction

- What is it?
- Basic Serial Peripheral Interface (SPI)
- Capabilities
- Protocol
- Pro / Cons and Competitor
- Uses
- Conclusion

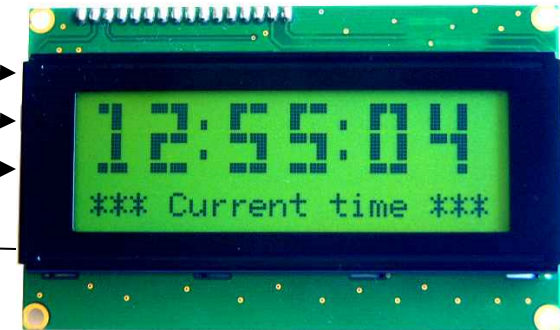


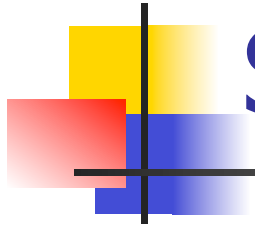
Serial Peripheral Interface

http://upload.wikimedia.org/wikipedia/commons/thumb/e/ed/SPI_single_slave.svg/350px-SPI_single_slave.svg.png

What is SPI?

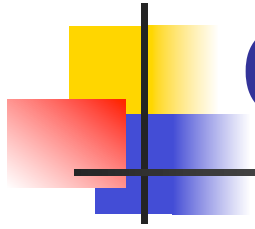
- Serial Bus protocol
- Fast, Easy to use, Simple
- Everyone supports it





SPI Basics

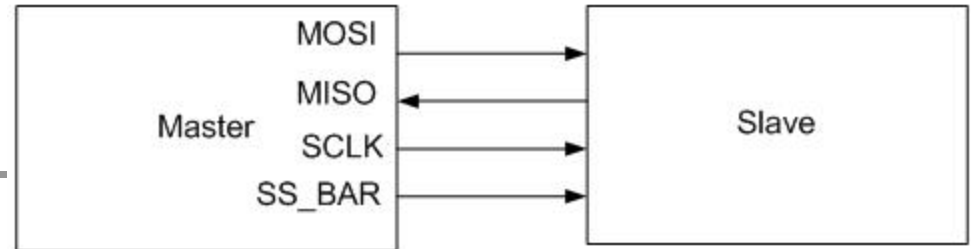
- A communication protocol using 4 wires
 - Also known as a 4 wire bus
- Used to communicate across small distances
- Multiple Slaves, Single Master
- Synchronized



Capabilities of SPI

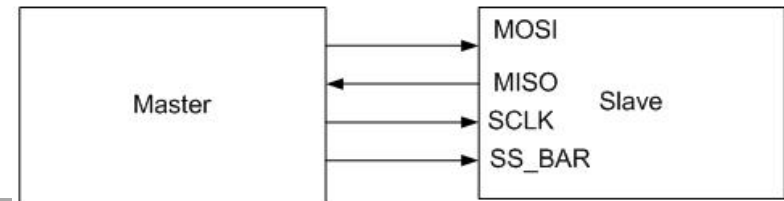
- Always Full Duplex
 - Communicating in two directions at the same time
 - Transmission need not be meaningful
- Multiple Mbps transmission speed
- Transfers data in 4 to 16 bit characters
- Multiple slaves
 - Daisy-chaining possible

Protocol



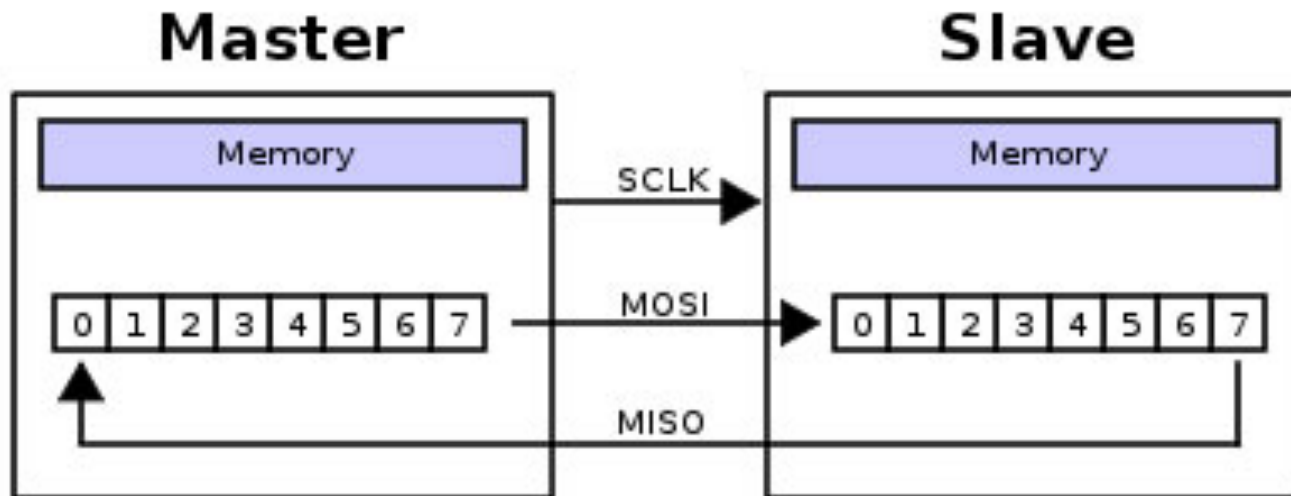
- Wires:
 - Master Out Slave In (MOSI)
 - Master In Slave Out (MISO)
 - System Clock (SCLK)
 - Slave Select 1...N
- Master Set Slave Select low
- Master Generates Clock
- Shift registers shift in and out data

Wires in Detail



- MOSI – Carries data out of Master to Slave
- MISO – Carries data from Slave to Master
 - Both signals happen for every transmission
- SS_BAR – Unique line to select a slave
- SCLK – Master produced clock to synchronize data transfer

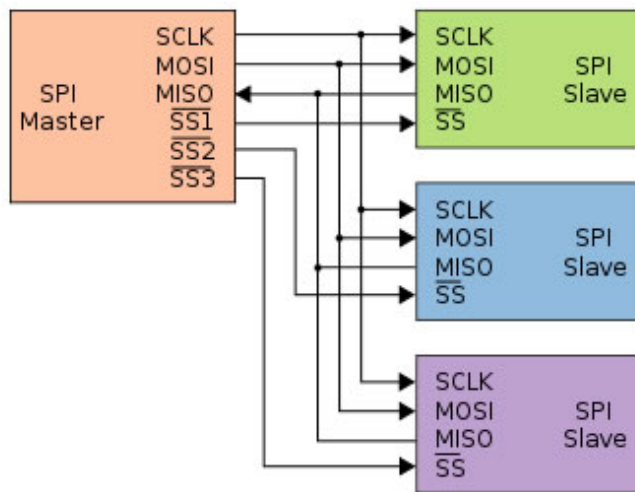
Shifting Protocol



Master shifts out data to Slave, and shift in data from Slave

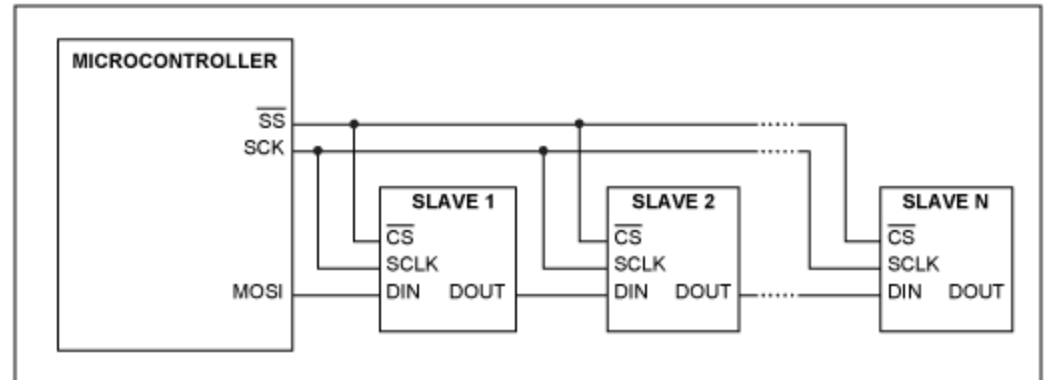
http://upload.wikimedia.org/wikipedia/commons/thumb/b/bb/SPI_8-bit_circular_transfer.svg/400px-SPI_8-bit_circular_transfer.svg.png

Diagram



Master and multiple independent slaves

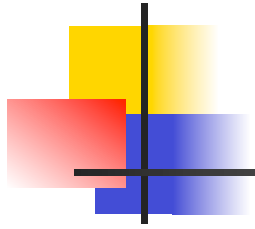
http://upload.wikimedia.org/wikipedia/commons/thumb/f/fc/SPI_three_slaves.svg/350px-SPI_three_slaves.svg.png



Some wires have been renamed

Master and multiple daisy-chained slaves

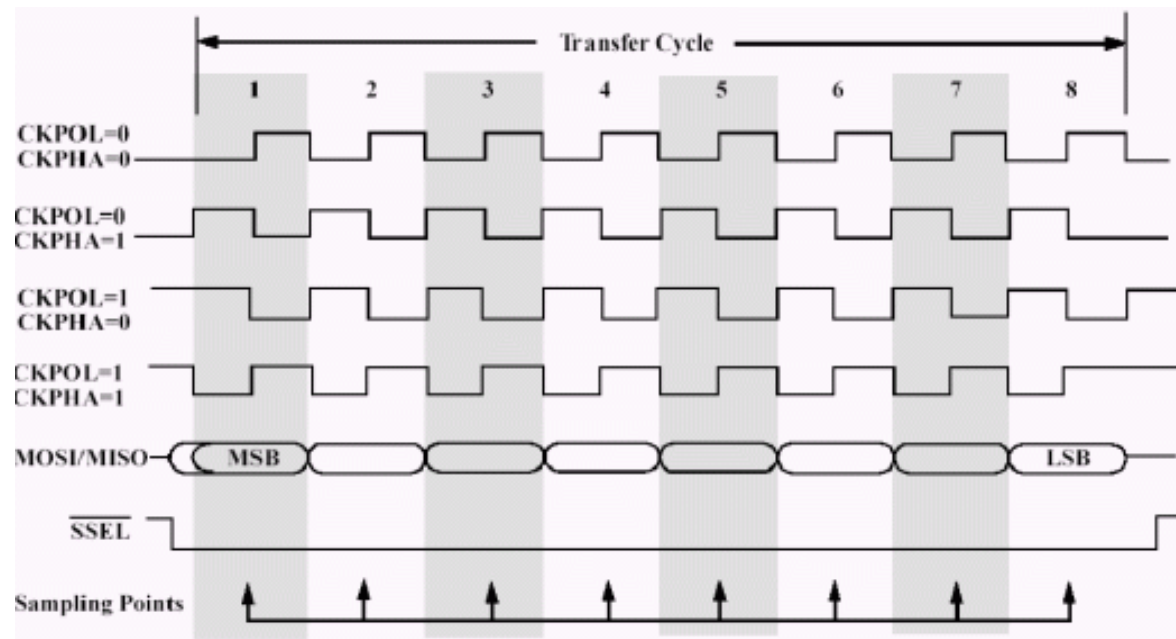
http://www.maxim-ic.com/appnotes.cfm/an_pk/3947



Clock Phase (Advanced)

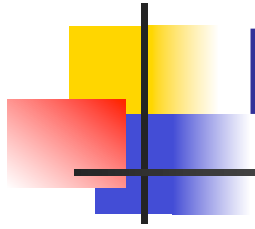
- Two phases and two polarities of clock
- Four modes
- Master and selected slave must be in same mode
- Master must change polarity and phase to communicate with slaves of different numbers

Timing Diagram



Timing Diagram – Showing Clock polarities and phases

<http://www.maxim-ic.com.cn/images/appnotes/3078/3078Fig02.gif>



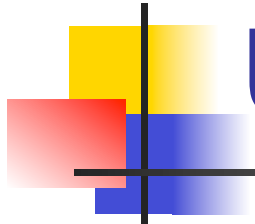
Pros and Cons

Pros:

- Fast and easy
 - Fast for point-to-point connections
 - Easily allows streaming/Constant data inflow
 - No addressing/Simple to implement
- Everyone supports it

Cons:

- SS makes multiple slaves very complicated
- No acknowledgement ability
- No inherent arbitration
- No flow control



Uses

- Some Serial Encoders/Decoders, Converters, Serial LCDs, Sensors, etc.
- Pre-SPI serial devices



Summary

- SPI – 4 wire serial bus protocol
 - MOSI MISO SS SCLK wires
- Full duplex
- Multiple slaves, One master
- Best for point-to-point streaming data
- Easily Supported

Agenda



- Serial Buses
 - Introduction
 - UART
 - SPI
 - I2C
- Glitches
 - Asynchronous resets and glitches
 - Design rules
- Set-up and hold time.
 - Review
 - Dealing with external inputs
 - Design rules



What is I²C?

- Inter-Integrated Circuit
- Pronounced “eye-squared-see”
- Two-wire serial bus protocol
- Invented by Philips in the early 1980's
 - That division now spun-off into NXP



Where is it used?

- Originally used by Philips inside television sets
- Now very common in peripheral devices intended for embedded systems use
 - Philips, National Semiconductor, Xicor, and Siemens , ...
- Also used in the PC world
 - Real time clock
 - Temperature sensors



Technical Description

- Two-wire serial protocol with addressing capability
- Speeds up to 3.4 Mbit/s
- Multi-master/Multi-slave



Wiring

- Two lines
 - SDA (data)
 - SCL (clock)
- Open-collector
 - Very simple interfacing between different voltage levels

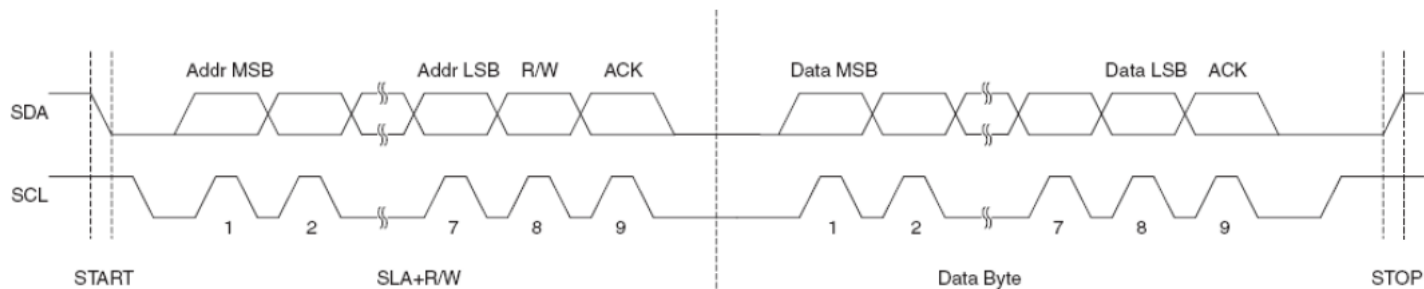


Clock

- Not a traditional clock
- Normally high (kept high by the pull-up)
- Pulsed by the master during data transmission (whether the master is transmitter or receiver)
- Slave device can hold clock low if it needs more time

A Basic I²C Transaction

- Master always initiates transactions
- Start Condition
- Address
- Data
- Acknowledgements
- Stop Condition



Source: ATmega8 Handbook



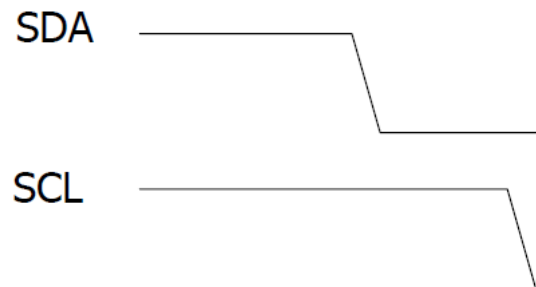
A Basic I²C Transaction

- Transmitter/Receiver differs from Master/Slave
- Master initiates transactions, slave responds
- Transmitter sets data on the SDA line, Receiver acknowledges
 - For a read, slave is transmitter
 - For a write, master is transmitter



Start Condition

- Master pulls SDA low while SCL is high
 - Normal SDA changes only happen while SCL is low

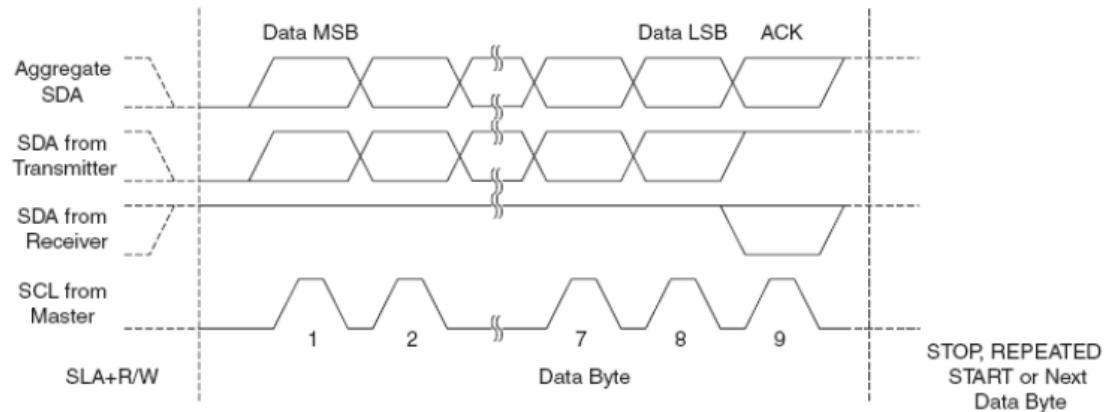




Address Transmission

- Data is always sampled on rising edge of clock
- Address is 7 bits
- An 8th bit indicates read or write
 - High for read, low for write
- Addresses assigned by Philips/NXP (for a fee)

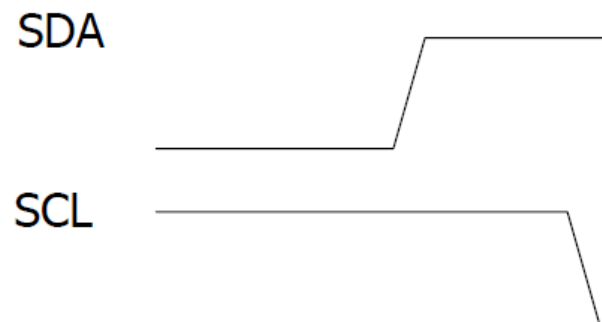
- Transmitted just like address (8 bits)
- For a write, master transmits, slave acknowledges
- For a read, slave transmits, master acknowledges
- Transmission continues with subsequent bytes until master creates stop condition



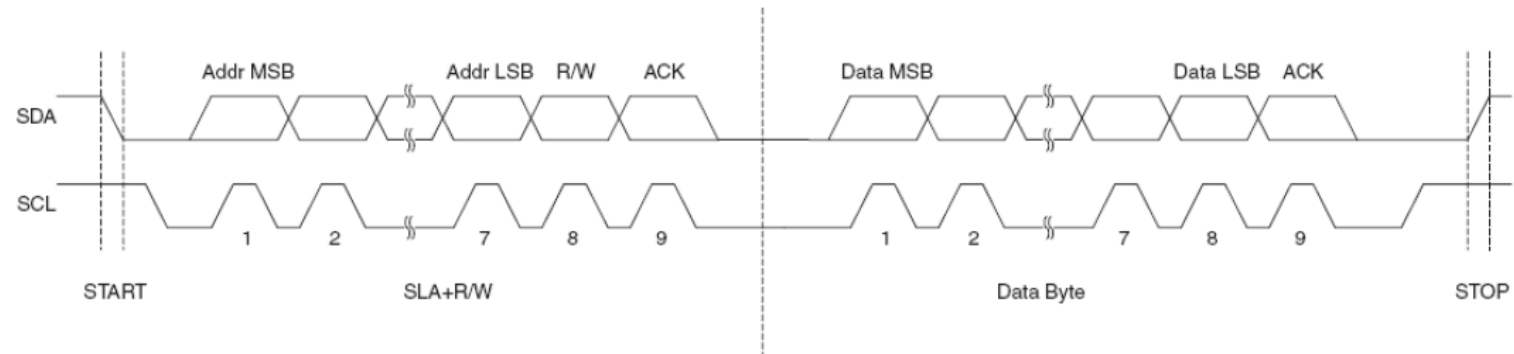
Source: ATMega8 Handbook

Stop Condition

- Master pulls SDA high while SCL is high
- Also used to abort transactions



Another Look



Source: ATmega8 Handbook

Agenda



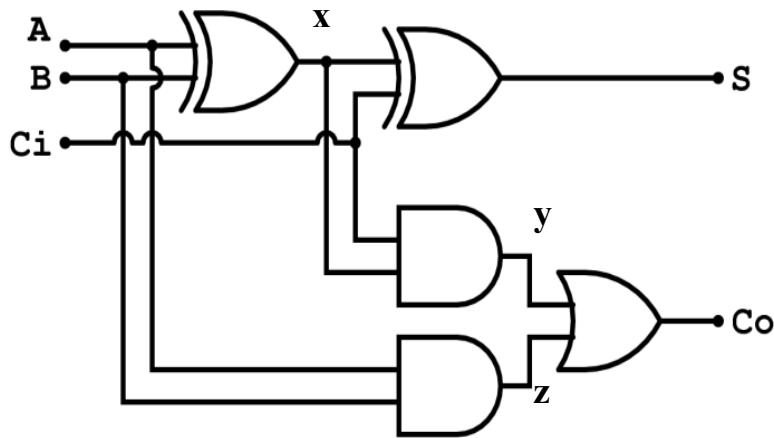
- Serial Buses
- Glitches
 - Asynchronous resets and glitches
 - Design rules
- Set-up and hold time.
 - Review
 - Dealing with external inputs
 - Design rules

Glitches



- Combinational logic can glitch
 - What is a glitch?
 - How do we normally deal with it?
 - Where can it hurt us?

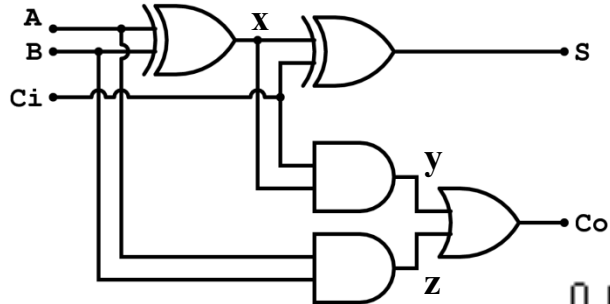
Timing



Full adder (from Wikipedia)

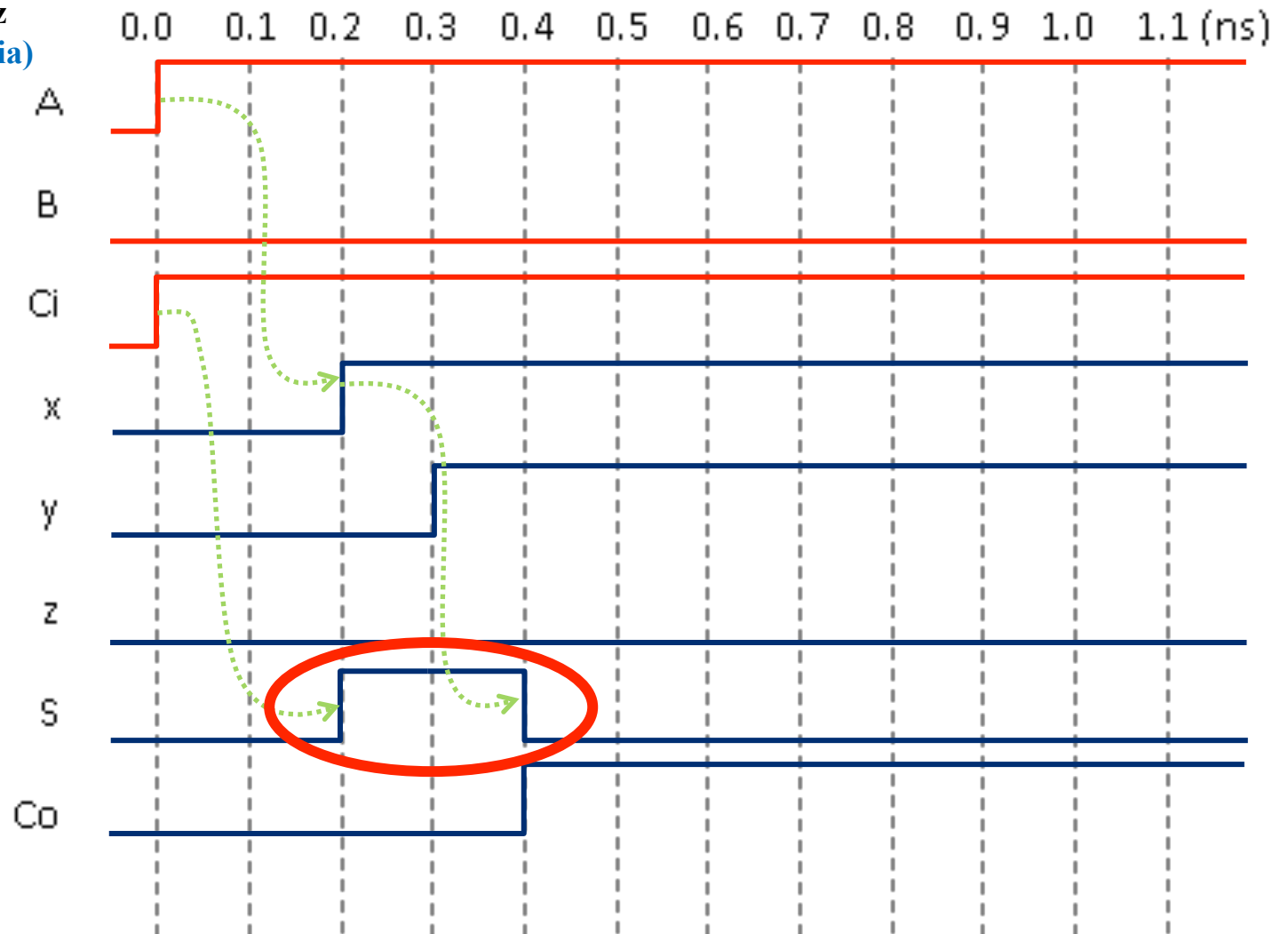
- Assuming the XOR gates have a delay of 0.2ns while AND and OR gates have a delay of 0.1ns
 - What is the worst case propagation delay for this circuit?

Glitches



Full adder (from Wikipedia)

Consider the adjacent circuit diagram. Assuming the XOR gates have a delay of 0.2ns while AND and OR gates have a delay of 0.1ns, fill in the following chart.



Only selected causality
arrows shown...

Glitching: a summary

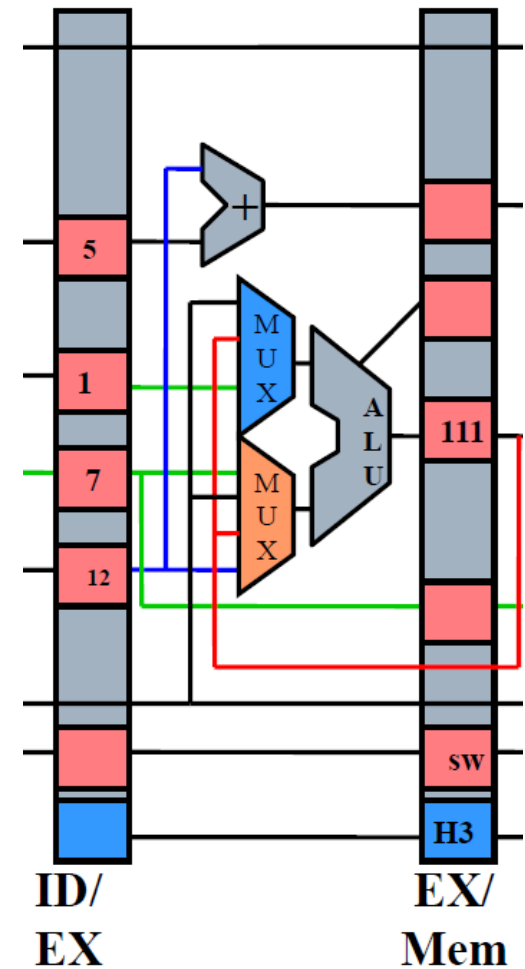


- When input(s) change, the output can be wrong for a time. However, that time is bound.
 - And more so, the output can change during this “computation time” even if the output ends up where it started!

Effect of Glitches



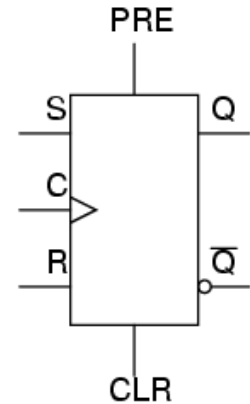
- Think back to EECS 370.
 - Why don't glitches cause errors?



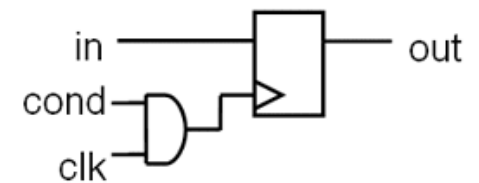
So, how can glitches hurt us?



- There are a handful of places:
 - Asynchronous resets
 - If you've got a flip-flop that has an asynchronous reset (or "preset") you need to be sure the input can't glitch.
 - That pretty much means you need a flip-flop driving the input (which means you probably should have used a sync. reset!)
 - Clocks
 - If you are using combinational logic to drive a clock, you are likely going to get extra clock edges.



Traditionally, CLR is used to indicate async reset. "R" or "reset" for sync. reset.

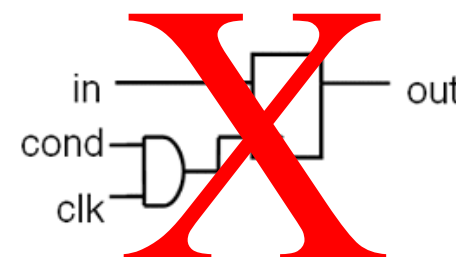
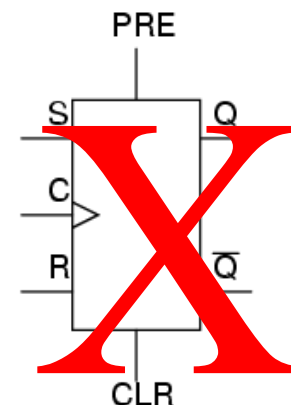


If clk is high and cond glitches, you get extra edges!

Design rules



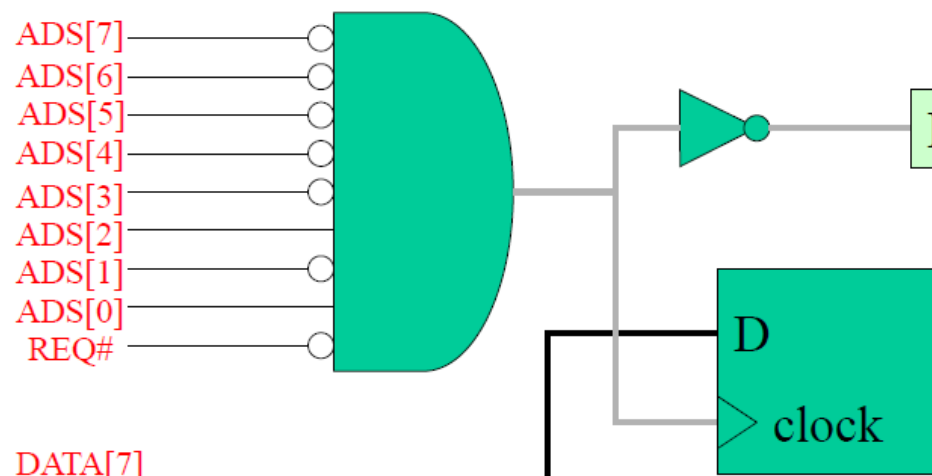
1. Thou shall Not use asynchronous resets
2. Thou shall not drive a clock with anything other than a clock or directly off of a flip-flop's output



Really?



- I mean people *use* asynchronous resets and clock gating!
 - Yep. And people use `goto` in C programs.
 - Sometimes they are the right thing.
 - But you have to think *really* hard about them to insure that they won't cause you problems.
 - Our “simple” bus used combinational logic for the clock
 - Works because REQ goes low only after everything else has stopped switching
 - So no glitch.
 - Not fun to reason about...
- Avoid unless you must
 - Then think *really* carefully.



Agenda

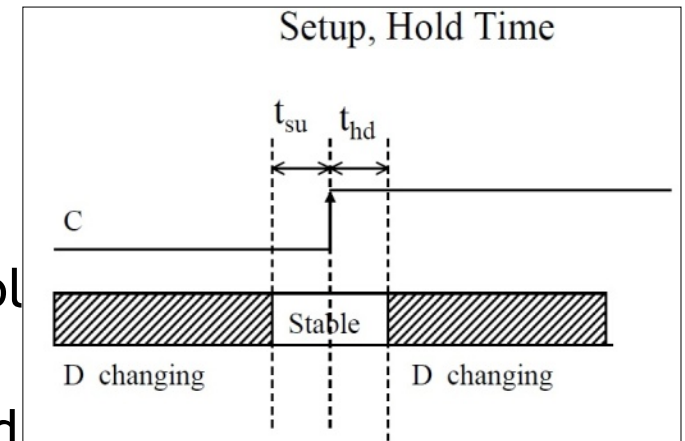


- Serial Buses,
- Glitches
 - Asynchronous resets and glitches
 - Design rules
- Set-up and hold time.
 - Review
 - Dealing with external inputs
 - Design rules

Setup and hold time



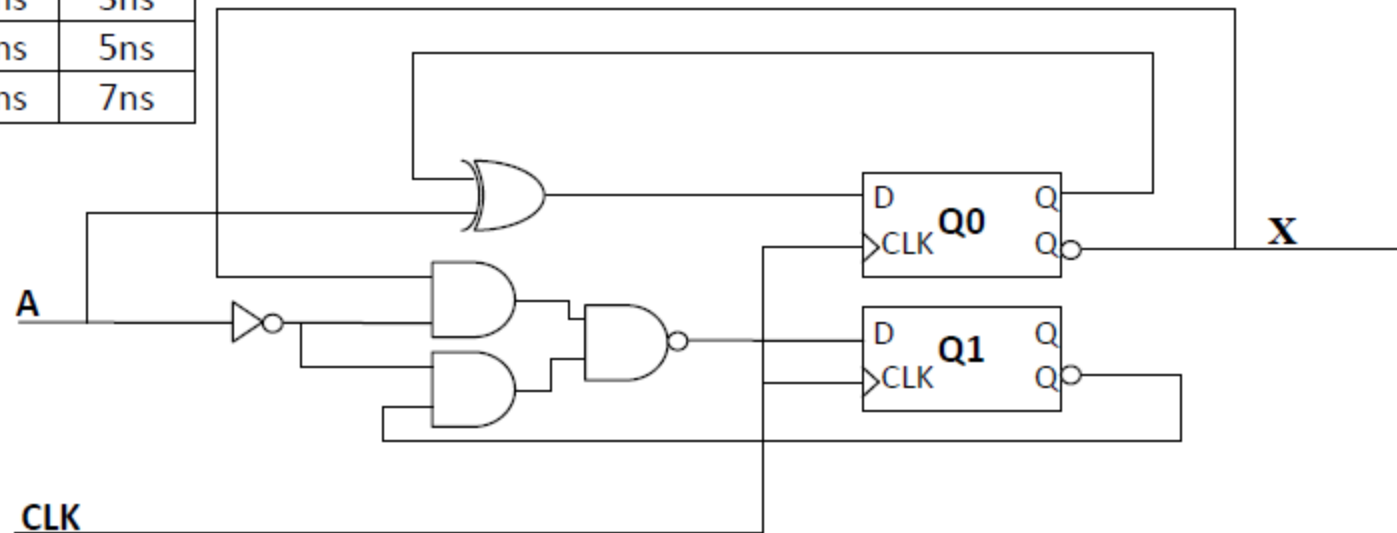
- The idea is simple.
 - When the clock is changing if the data is also changing it is hard to tell what the data is.
 - Hardware can't always tell
 - And you can get meta-stable (unlikely but...)
 - So we have a “guard band” around the clock rising time during which we don't allow the data to change.
 - See diagram. We call the time before the clock-edge “setup time” and the time after “hold time”



Device	Min	Max
DFF:		
<i>Clock to Q</i>	1ns	4ns
<i>Set-up time</i>	4ns	
<i>Hold time</i>	5ns	
OR/AND	2ns	6ns
NOT	1ns	3ns
NAND/NOR	2ns	5ns
XOR	3ns	7ns

Example:

Fast and slow paths; impact of setup and hold time



Assume that the input A is coming from a flip-flop that has the same properties as the flip-flops that are shown and is clocked by the same clock.

- Add inverter pairs as needed to the above figure to avoid any “fast path” problems. Do so in a way that has least impact on the worst-case delay (as a first priority) and which keeps the number of inverter pairs needed to a minimum (as a second priority).
- After you’ve made your changes in part a, compute the maximum **frequency** at which this device can be safely clocked.

So what happens if we violate set-up or hold time?



- Often just get one of the two values.
 - And that often is just fine.
 - Consider getting a button press from the user.
 - If the button gets pressed at the same time as the clock edge, we might see the button now or next clock.
 - Either is generally fine when it comes to human input.
 - But bad things could happen.
 - The flip-flop's output might not settle out to a "0" or a "1"
 - That could cause latter devices to mess up.
 - More likely, if that input is going to two places, one might see a "0" the other a "1".

Example

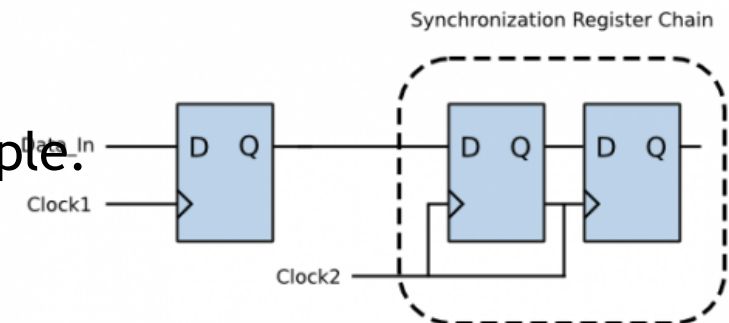


- A common thing to do is reset a state machine using a button.
 - User can “reset” the system.
- Because the button transition could violate set-up or hold time, some state bits of the state machine might come out of reset at different times.
 - And you quickly end up at a wrong or illegal state.

So...



- Dealing with inputs not synchronized to our local clock is a problem.
 - Likely to violate setup or hold time.
 - That could lead to things breaking.
- So we need a clock synchronization circuit.
 - First flip-flop might have problems.
 - Second should be fine.
 - Sometimes use a third if really paranoid
 - Safety-critical system for example.



Design rules



3. Thou shalt use a clock synchronization circuit when changing clock domains or using unclocked inputs!

```
/* Synchronization of Asynchronous switch input */
```

```
always@(posedge clk)
```

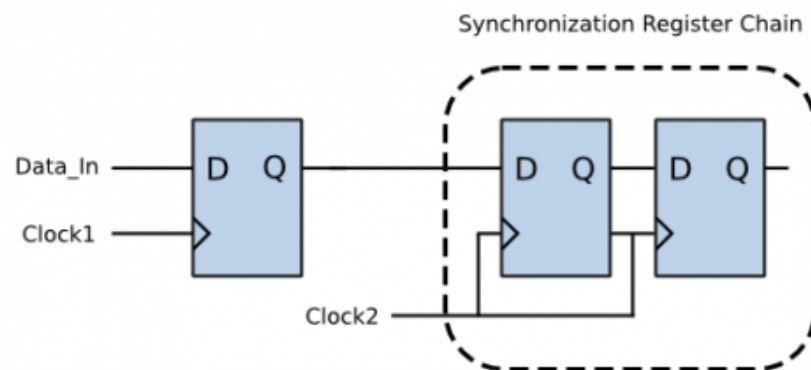
```
begin
```

```
    sw0_pulse[0] <= sw_port[0];
```

```
    sw0_pulse[1] <= sw0_pulse[0];
```

```
    sw0_pulse[2] <= sw0_pulse[1];
```

```
end
```



```
always @(posedge clk) SSElr <= {SSElr[1:0], SSEl};
```