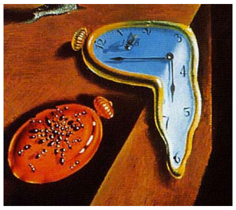
EECS 373
Design of Microprocessor-Based Systems

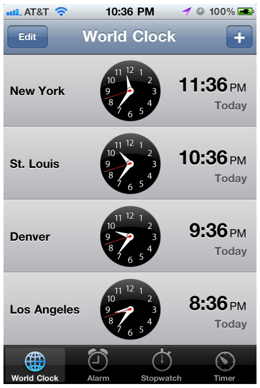


Prabal Dutta
University of Michigan

Lecture 8: Clocks, Counters, Timers, Capture, and Compare
September 25 & 30, 2014

Some slides by Mark Brehob and Thomas Schmid

iPhone Clock App



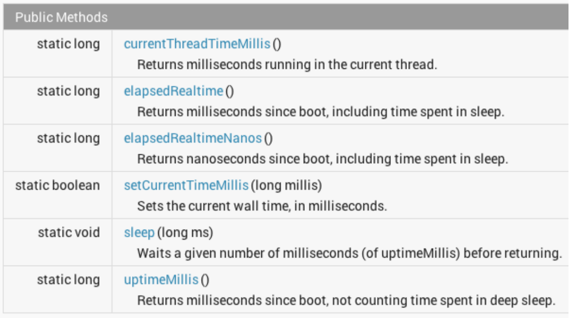
- World Clock – display real time in multiple time zones
- Alarm – alarm at certain (later) time(s).
- Stopwatch – measure elapsed time of an event
- Timer – count down time and notify when count becomes zero

Motor/Light Control



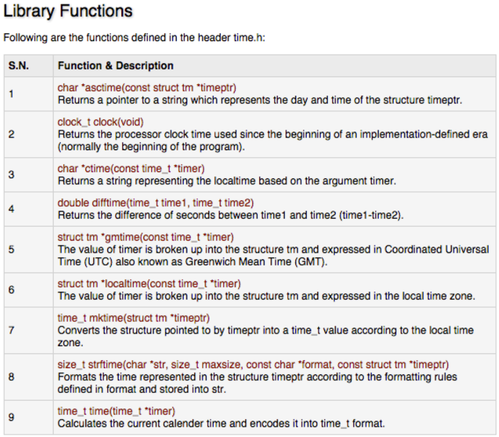
- Servo motors – PWM signal provides control signal
- DC motors – PWM signals control power delivery
- RGB LEDs – PWM signals allow dimming through current-mode control

Methods from android.os.SystemClock



Public Methods	
static long	currentThreadTimeMillis() Returns milliseconds running in the current thread.
static long	elapsedRealtime() Returns milliseconds since boot, including time spent in sleep.
static long	elapsedRealtimeNanos() Returns nanoseconds since boot, including time spent in sleep.
static boolean	setCurrentTimeMillis(long millis) Sets the current wall time, in milliseconds.
static void	sleep(long ms) Waits a given number of milliseconds (of uptimeMillis) before returning.
static long	uptimeMillis() Returns milliseconds since boot, not counting time spent in deep sleep.

Standard C library's <time.h> header file

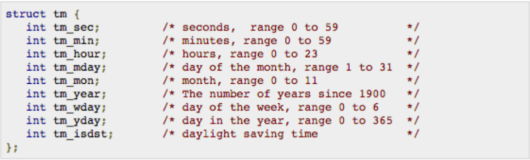


Library Functions

Following are the functions defined in the header time.h:

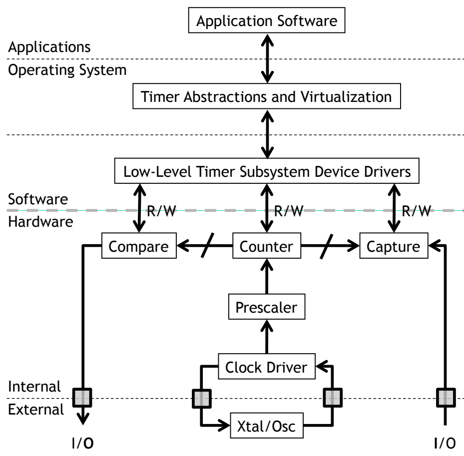
S.N.	Function & Description
1	char *asctime(const struct tm *timeptr) Returns a pointer to a string which represents the day and time of the structure timeptr.
2	clock_t clock(void) Returns the processor clock time used since the beginning of an implementation-defined era (normally the beginning of the program).
3	char *ctime(const time_t *timer) Returns a string representing the localtime based on the argument timer.
4	double difftime(time_t time1, time_t time2) Returns the difference of seconds between time1 and time2 (time1-time2).
5	struct tm *gmtime(const time_t *timer) The value of timer is broken up into the structure tm and expressed in Coordinated Universal Time (UTC) also known as Greenwich Mean Time (GMT).
6	struct tm *localtime(const time_t *timer) The value of timer is broken up into the structure tm and expressed in the local time zone.
7	time_t mktime(struct tm *timeptr) Converts the structure pointed to by timeptr into a time_t value according to the local time zone.
8	size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *timeptr) Formats the time represented in the structure timeptr according to the formatting rules defined in format and stored into str.
9	time_t time(time_t *timer) Calculates the current calender time and encodes it into time_t format.

Standard C library's <time.h> header file: struct tm



```
struct tm {
   int tm_sec;         /* seconds,  range 0 to 59          */
   int tm_min;         /* minutes, range 0 to 59           */
   int tm_hour;        /* hours, range 0 to 23             */
   int tm_mday;        /* day of the month, range 1 to 31  */
   int tm_mon;         /* month, range 0 to 11             */
   int tm_year;        /* The number of years since 1900   */
   int tm_wday;        /* day of the week, range 0 to 6    */
   int tm_yday;        /* day in the year, range 0 to 365  */
   int tm_isdst;       /* daylight saving time             */
};
```

Anatomy of a timer system



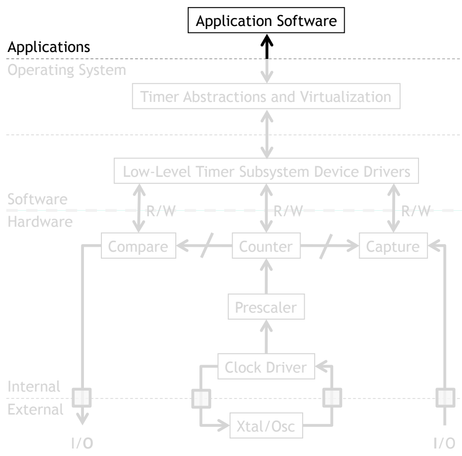
```
...
timer_t timerX;
initTimer();
...
startTimerOneShot(timerX, 1024);
...
stopTimer(timerX);
```

```
typedef struct timer {
  timer_handler_t handler;
  uint32_t time;
  uint8_t mode;
  timer_t* next_timer;
} timer_t;
```

```
timer_tick:
  ldr r0, count;
  add r0, r0, #1
  ...
```

```
module timer(clr, ena, clk, alrm);
  input clr, ena, clk;
  output alrm;
  reg alrm;
  reg [3:0] count;

  always @(posedge clk) begin
    alrm <= 0;
    if (clr) count <= 0;
    else count <= count+1;
  end
endmodule
```

Anatomy of a timer system



```
timer_t timerX;
initTimer();
...
startTimerOneShot(timerX, 1024);
...
stopTimer(timerX);
```

```
typedef struct timer {
  timer_handler_t handler;
  uint32_t time;
  uint8_t mode;
  timer_t* next_timer;
} timer_t;
```

```
timer_tick:
  ldr r0, count;
  add r0, r0, #1
  ...
```

```
module timer(clr, ena, clk, alrm);
  input clr, ena, clk;
  output alrm;
  reg alrm;
  reg [3:0] count;

  always @(posedge clk) begin
    alrm <= 0;
    if (clr) count <= 0;
    else count <= count+1;
  end
endmodule
```

What do we really want from our timing subsystem?

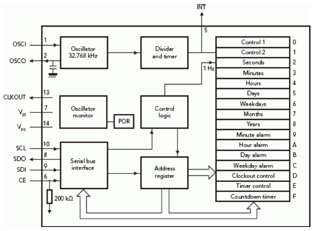
- Wall clock date & time
  - Date: Month, Day, Year
  - Time: HH:MM:SS:mmm
  - Provided by a “real-time clock” or RTC
- Alarm: do something (call code) at certain time later
  - Later could be a delay from now (e.g. Δt)
  - Later could be actual time (e.g. today at 3pm)
- Stopwatch: measure (elapsed) time of an event
  - Instead of pushbuttons, could be function calls or
  - Hardware signals outside the processor
- Timer – count down time and notify when count = 0
  - Could invoke some code (e.g. a handler)
  - Could take some action (e.g. set/clear an I/O line)

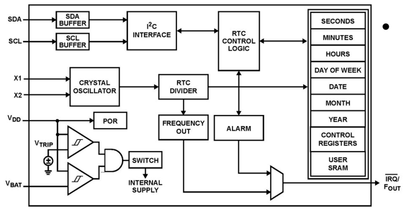
What do we really want from our timing subsystem?

- Wall clock
  - datetime_t getDateTime()
- Alarm
  - void alarm(callback, delta)
  - void alarm(callback, datetime_t)
- Stopwatch: measure (elapsed) time of an event
  - t1 = now(); … ; t2 = now(); dt = difftime(t2, t1);
  - GPIO_INT_ISR:

    LDR R1, [R0, #0] % R0=timer address
- Timer – count down time and notify when count = 0
  - void timer(callback, delta)
  - Timer fires → Set/Clear GPIO line (using DMA)

Wall Clock from a Real-Time Clock (RTC)





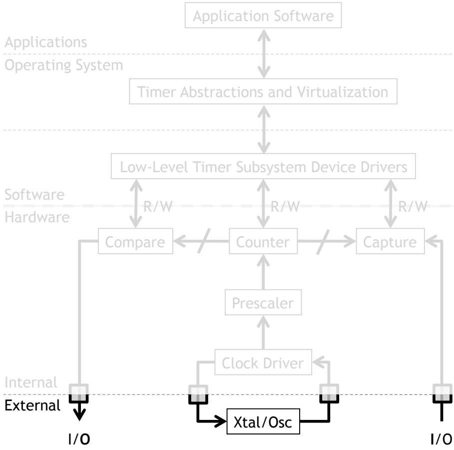
- Often a separate module
- Built with registers for
  - Years, Months, Days
  - Hours, Mins, Seconds
- Alarms: hour, min, day
- Accessed via
  - Memory-mapped I/O
  - Serial bus (I2C, SPI)

What do we really want from our timing subsystem?

- Wall clock
  - datetime_t getDateTime()
- Alarm
  - void alarm(callback, delta)
  - void alarm(callback, datetime_t)
- Stopwatch: measure (elapsed) time of an event
  - t1 = now(); … ; t2 = now(); dt = difftime(t2, t1);
  - GPIO_INT_ISR:

    LDR R1, [R0, #0] % R0=timer address
- Timer – count down time and notify when count = 0
  - void timer(callback, delta)
  - Timer fires → Set/Clear GPIO line (using DMA)

Anatomy of a timer system



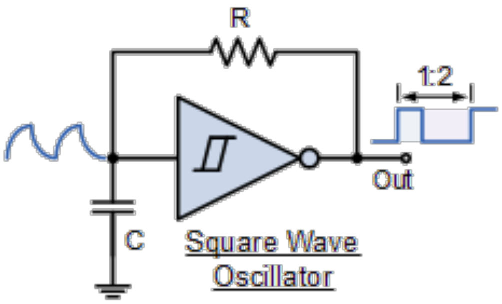
```
timer_t timerX;
initTimer();
...
startTimerOneShot(timerX, 1024);
...
stopTimer(timerX);
```

```
typedef struct timer {
  timer_handler_t handler;
  uint32_t time;
  uint8_t mode;
  timer_t* next_timer;
} timer_t;
```

```
timer_tick:
  ldr r0, count;
  add r0, r0, #1
  ...
```

```
module timer(clr, ena, clk, alrm);
  input clr, ena, clk;
  output alrm;
  reg alrm;
  reg [3:0] count;

  always @(posedge clk) begin
    alrm <= 0;
    if (clr) count <= 0;
    else count <= count+1;
  end
endmodule
```

Oscillators - RC



Oscillators - Crystal

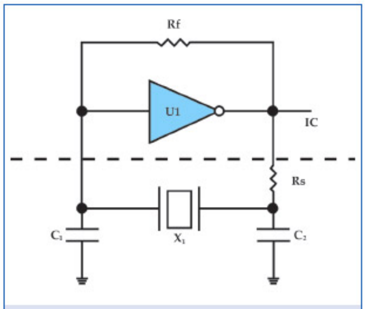
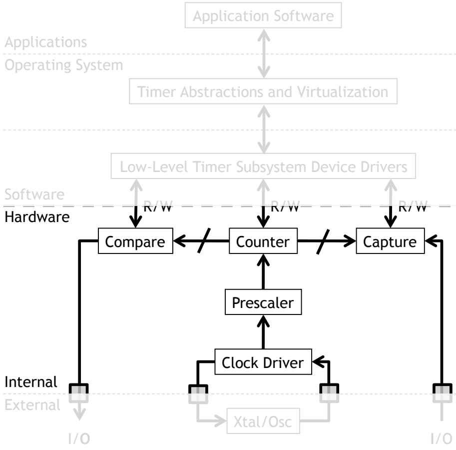


Figure 1: Fundamental Mode Isolated Pierce-Gate Oscillator

Anatomy of a timer system



```
timer_t timerX;
initTimer();
...
startTimerOneShot(timerX, 1024);
...
stopTimer(timerX);
```

```
typedef struct timer {
  timer_handler_t handler;
  uint32_t time;
  uint8_t mode;
  timer_t* next_timer;
} timer_t;
```

```
timer_tick:
  ldr r0, count;
  add r0, r0, #1
  ...
```

```
module timer(clr, ena, clk, alrm);
  input clr, ena, clk;
  output alrm;
  reg alrm;
  reg [3:0] count;

  always @(posedge clk) begin
    alrm <= 0;
    if (clr) count <= 0;
    else count <= count+1;
  end
endmodule
```

What do we really want from our timing subsystem?

- Wall clock
  - datetime_t getDateTime()
- Alarm
  - void alarm(callback, delta)
  - void alarm(callback, datetime_t)
- Stopwatch: measure (elapsed) time of an event
  - t1 = now(); … ; t2 = now(); dt = difftime(t2, t1);
  - GPIO_INT_ISR:

    LDR R1, [R0, #0]   % R0=timer address
- Timer – count down time and notify when count = 0
  - void timer(callback, delta)
  - Timer fires → Set/Clear GPIO line (using DMA)

Why should we care?

- There are two basic activities one wants timers for:
  - Measure how long something takes
    - “Capture”
  - Have something happen once or every X time period
    - “Compare”

Example # 1: Capture

- FAN
  - Say you have a fan spinning and you want to know how fast it is spinning. One way to do that is to have it throw an interrupt every time it completes a rotation.
    - Right idea, but might take a while to process the interrupt, heavily loaded system might see slower fan than actually exists.
    - This could be bad.
  - Solution? Have the timer note *immediately* how long it took and then generate the interrupt. Also restart timer immediately.
- Same issue would exist in a car when measuring speed of a wheel turning (for speedometer or anti-lock brakes).

Example # 2: Compare

- Driving a DC motor via PWM.
  - Motors turn at a speed determined by the voltage applied.
    - Doing this in analog land can be hard.
      - Need to get analog out of our processor
      - Need to amplify signal in a linear way (op-amp?)
    - Generally prefer just switching between “Max” and “Off” quickly.
      - Average is good enough.
      - Now don’t need linear amplifier—just “on” and “off”. (transistor)
  - Need a signal with a certain duty cycle and frequency.
    - That is % of time high.

Servo motor control: class exercise

- Assume 1 MHz CLK
- Design “high-level” circuit to
  - Generate 1.52 ms pulse
  - Every 6 ms
  - Repeat
- How would we generalize this?

SmartFusion Timer System

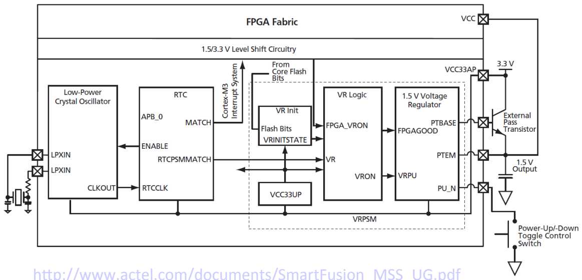
Timers on the SmartFusion

- SysTick Timer
  - ARM requires every Cortex-M3 to have this timer
  - Essentially a 24-bit down-counter to generate system ticks
  - Has its own interrupt
  - Clocked by FCLK with optional programmable divider
- See Actel SmartFusion MSS User Guide for register definitions

---

Timers on the SmartFusion

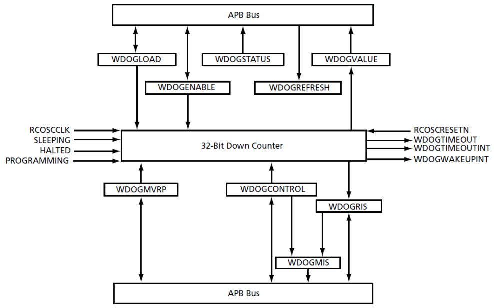
- Real-Time Counter (RTC) System
  - Clocked from 32 kHz low-power crystal
  - Automatic switching to battery power if necessary
  - Can put rest of the SmartFusion to standby or sleep to reduce power
  - 40-bit match register clocked by 32.768 kHz divided by 128 (256 Hz)



http://www.actel.com/documents/SmartFusion_MSS_UG.pdf

Timers on the SmartFusion

- Watchdog Timer
  - 32-bit down counter
  - Either reset system or NMI Interrupt if it reaches 0!

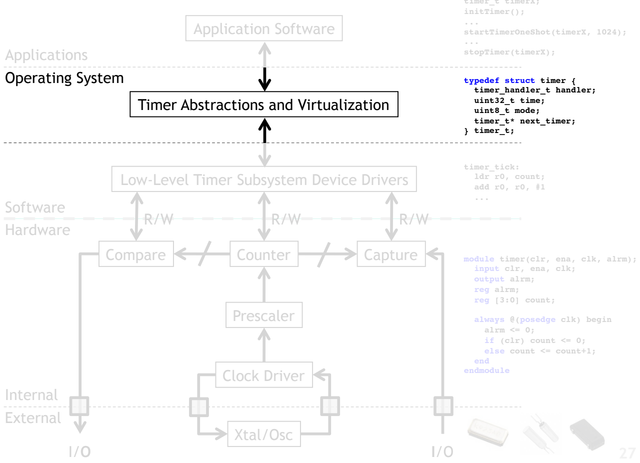


Timers on the SmartFusion

- System timer
  - "The System Timer consists of two programmable 32-bit decrementing counters that generate interrupts to the ARM® Cortex™-M3 and FPGA fabric. Each counter has two possible modes of operation: Periodic mode or One-Shot mode. The two timers can be concatenated to create a 64-bit timer with Periodic and One-Shot modes. The two 32-bit timers are identical"

http://www.actel.com/documents/SmartFusion_MSS_UG.pdf

Anatomy of a timer system



```
timer_t timerX;
initTimer();
...
startTimerOneShot(timerX, 1024);
...
stopTimer(timerX);
```

```
typedef struct timer {
  timer_handler_t handler;
  uint32_t time;
  uint8_t mode;
  timer_t* next_timer;
} timer_t;
```

```
timer_tick:
  ldr r0, count;
  add r0, r0, #1
  ...
```

```
module timer(clr, ena, clk, alrm);
  input clr, ena, clk;
  output alrm;
  reg alrm;
  reg [3:0] count;

  always @(posedge clk) begin
    alrm <= 0;
    if (clr) count <= 0;
    else count <= count+1;
  end
endmodule
```

Virtual Timers

- You never have enough timers.
  - Never.
- So what are we going to do about it?
  - How about we handle in software?

Virtual Timers

- Simple idea.
  - Maybe we have 10 events we might want to generate.
    - Just make a list of them and set the timer to go off for the *first* one.
      - Do that first task, change the timer to interrupt for the next task.

Problems?

- Only works for "compare" timer uses.
- Will result in slower ISR response time
  - May not care, could just schedule sooner…

Implementation Issues

- Shared user-space/ISR data structure.
  - Insertion happens at least some of the time in user code.
  - Deletion happens in ISR.
    - We need critical section (disable interrupt)
- How do we deal with our modulo counter?
  - That is, the timer wraps around.
  - Why is that an issue?
- What functionality would be nice?
  - Generally one-shot vs. repeating events
  - Might be other things desired though
- What if two events are to happen at the same time?
  - Pick an order, do both…

Implementation Issues (continued)

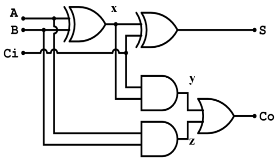
- What data structure?
  - Data needs be sorted
    - Inserting one thing at a time
  - We always pop from one end
  - But we add in sorted order.

Data structures

```
typedef struct timer
{
    timer_handler_t handler;
    uint32_t        time;
    uint8_t         mode;
    timer_t*        next_timer;
} timer_t;

timer_t* current_timer;

void initTimer() {
    setupHardwareTimer();
    initLinkedList();
    current_timer = NULL;
}

error_t startTimerOneShot(timer_handler_t handler, uint32_t t) {
    // add handler to linked list and sort it by time
    // if this is first element, start hardware timer
}

error_t startTimerContinuous(timer_handler_t handler, uint32_t dt) {
    // add handler to linked list for (now+dt), set mode to continuous
    // if this is first element, start hardware timer
}

error_t stopTimer(timer_handler_t handler) {
    // find element for handler and remove it from list
}
```

Some loose ends…glitches and all that

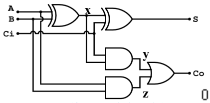
Timing delays and propagation



Full adder (from Wikipedia)

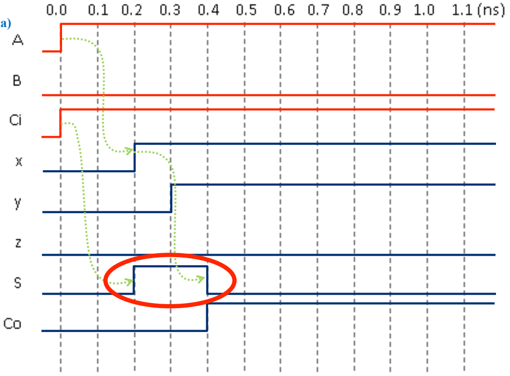
- Assume
  - XOR delay = 0.2ns
  - AND delay = 0.1ns
  - OR delay = 0.1 ns
- What is the worst case propagation delay for this circuit?

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.



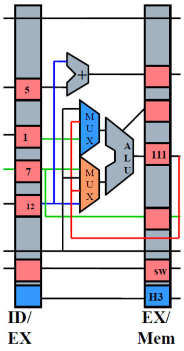
Only selected causality arrows shown…

Glitching: a summary

- When input(s) change
  - The output can be wrong for a time
  - However, that time is bounded
- And more so, the output can change during this "computation time" <u>even if the output ends up where it started</u>!

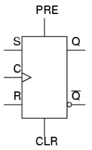
Effect of Glitches

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

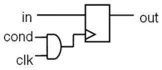


So, how can glitches hurt us?

- 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.



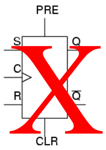
Traditionally, CLR is used to indicate async reset. "R" or "reset" for sync. reset.

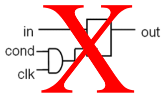


If clk is high and cond glitches, you get extra edges!

Design rules

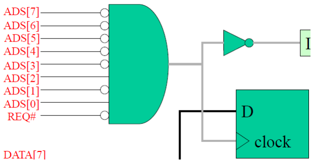
1. Thou shall Not use asynchronous resets
2. Thou shall not drive a clock with anything other than a clock or directly off of a flip-flop's output





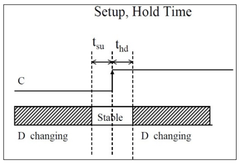
Really? Seriously?

- People *do 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.



Setup and hold time

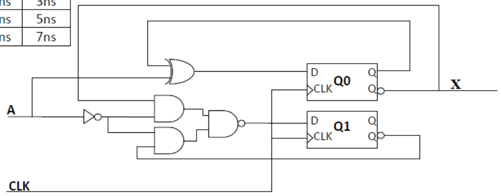
- The idea is simple.
  - When the clock is changing if the data is also changing it is hard to tell what the data ***is***.
    - Hardware can't always tell
      - And you can get meta-stable behavior too (very unlikely but…)
  - So we have a "guard band" around the clock rising time during which we 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:		
Clock to Q	1ns	4ns
Set-up time	4ns	
Hold time	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.

a. 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).
b. *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"
- Important: don't feed an async input to multiple places!

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.

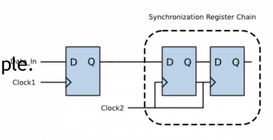
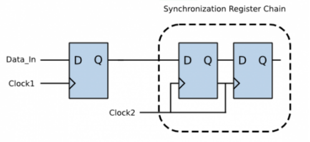


Figure from http://www.eeweb.com/electronics-quiz/solving-metastability-design-issues, we use the same thing to deal with external inputs too!

Design rules

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



```
/* Synchonization 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};
```

Questions?

Comments?

Discussion?