EECS 373
Design of Microprocessor-Based Systems

Robert Dick
University of Michigan

Lecture 8: Project Overview, Timers, and Hazards

31 January 2017

Slides inherited from Mark Brehob.

Outline

- Context and review
- Midterm exam
- Project
- Timers
  - General characteristics
  - SmartFusion board
- Hazards

Context and review

- Interrupts
  - Polling
  - Event-driven vs. procedural programming
  - Interrupt vectors / jump tables
  - Timing diagrams
  - Prioritization and masking
- Timers
  - Architecture and source of complexity
  - Pulse width modulation

NVIC

- Set/inspect interrupt enable, pending, and active status bits.
- Set/inspect priority level.
- Only highest four bits considered by our processor.
- See pages 34-42 of lecture 7 for additional details.

Bonus and timer tie-in: higher-level APIs atop interrupts

- Interrupt vector / jump table used to indicate ISR for each interrupt.
- Callbacks
  - Similar concept at a higher level.
  - Pass a function pointer (a callback) into another function.
  - We saw this with the sort routine.
  - But now the function can simply register the callback somewhere for later execution.
  - E.g., pass in a function to execute at a particular time.

```
int execute_when(void (*callback)(int when), int when);
```

Sharing data with ISR

- What if an ISR/program shared data structure requires multiple instructions to modify?
- E.g., deleting an element from a linked list.
- Program

  Get pointer to relevant list element.

  [What if interrupt happens here?]

  Read and write data in list element.
- ISR

  Delete list element.

Sharing data with ISR

- Solution: make atomic operations atomic.
- New program version

  Disable interrupts.

  Just those that care about inconsistent state.

  Very briefly.

  Get pointer to relevant list element.

  [What if interrupt happens here?]

  Read and write data in list element.

  Enable interrupts.

Debugging ISRs

- Set a breakpoint at interrupt handler.
  - Is it ever called?
- Examine NVIC registers.
  - Are they set correctly?
- Use oscilloscope to look at interrupt signal.
- Default interrupt vector table traps.
  - To infinite loop.

Outline

- ~~Context and review~~
- Midterm exam
- Project and topic talks
- Timers
  - General characteristics
  - SmartFusion board
- Hazards

Midterm exam conflicts

- 8-9:30pm on 22 February in 2505 GGBL.
- Deadline for pointing out a hard conflict and requesting an alternative time is today.
- Email me (dickrp@umich.edu).

Midterm exam preparation

- Material up to and including 9 February lecture may be tested.
- New material presented on or after 14 February will not be tested.
- HW 5 is largely a review assignment.
- 7 Feb: Will post practice midterms based on prior exams but adjusted to match material this semester.
- Not graded. You are welcome to discuss in office/lab hours.
- 14 Feb: Will post solutions to those practice midterms.

Review sessions

- Several review sessions will be held in the week before the exam.
- Times and locations will be posted to the website by 7 February.

Outline

- ~~Context and review~~
- ~~Midterm exam~~
- Project and topic talks
- Timers
  - General characteristics
  - SmartFusion board
- Hazards

Project

- Read the project handout if you haven't done so.
  - Click “project” in website menu.
- 31 January: topic proposals assigned
  - Two terse (<= 1 page) topic proposals due 3 February at 8pm.
  - Will provide format.
- 4 February: I post all proposals.
- 7 February: project team formation meeting.
- 13-15 February: 15-minute project proposals to Matthew and me.

Topic talk

- 1 February: topic talk time slot signup.
  - 15-minute educational talk on topic of interest.
  - Can be on topic relevant to project.
- 23 February: finalize talk title.

Outline

- ~~Context and review~~
- ~~Midterm exam~~
- ~~Project and topic talks~~
- Timers
  - General characteristics
  - SmartFusion board
- Hazards

Timers

- Why they matter?
- Avoid pitfalls of loop-based delays.
  - Waste power.
  - Prevent other useful work from being done.
- Why they are complex?
  - Span HW/SW boundary.

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 and 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 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

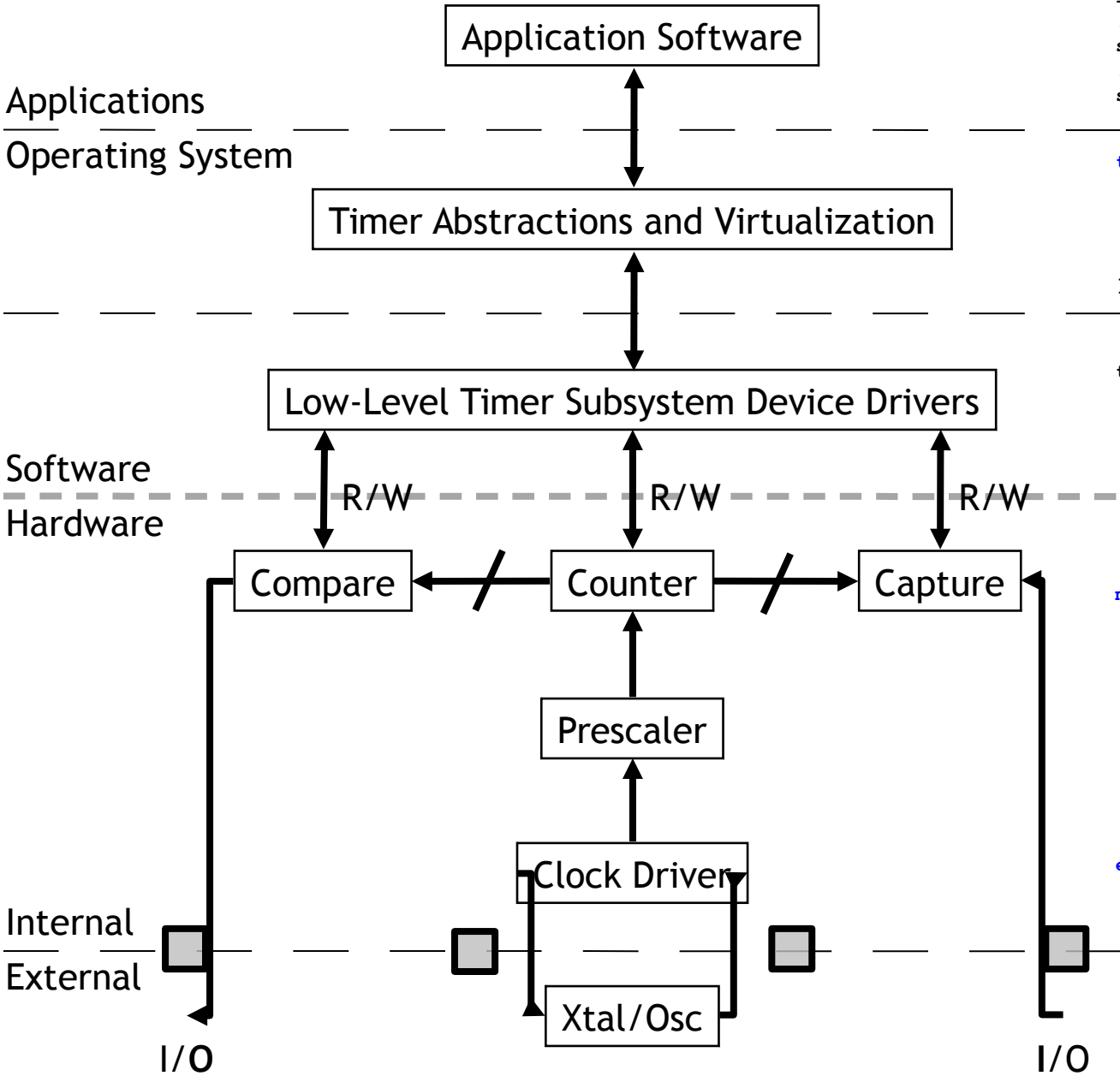
Following are the functions defined in the header time.h:

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);
```

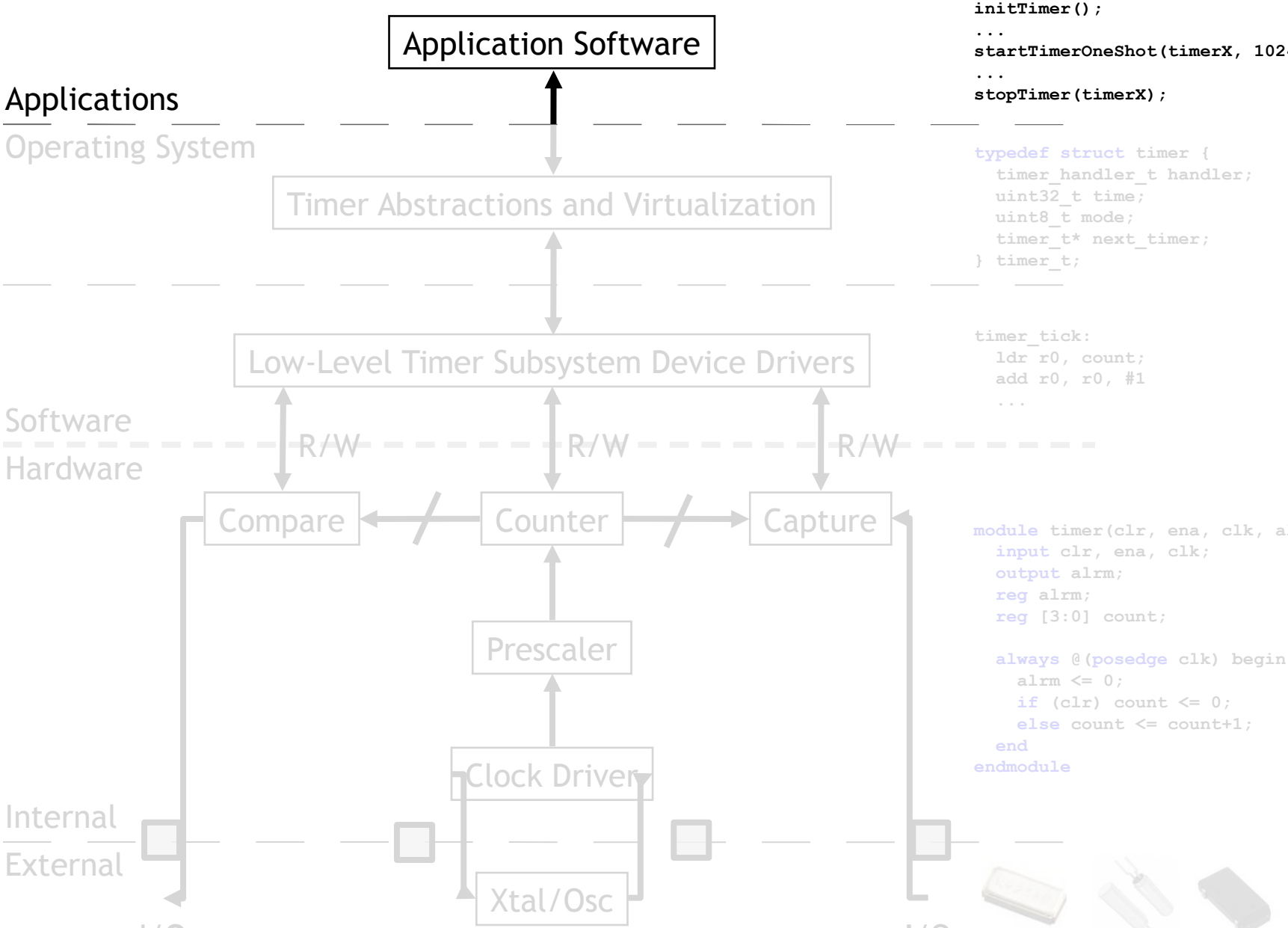
```
typedef struct timer {
  timer_handler_t handler;
  uint32_t time;
  uint8_t mode;
  timer_t* next_timer;
} timer_t;
```

```
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
```

Timer requirements

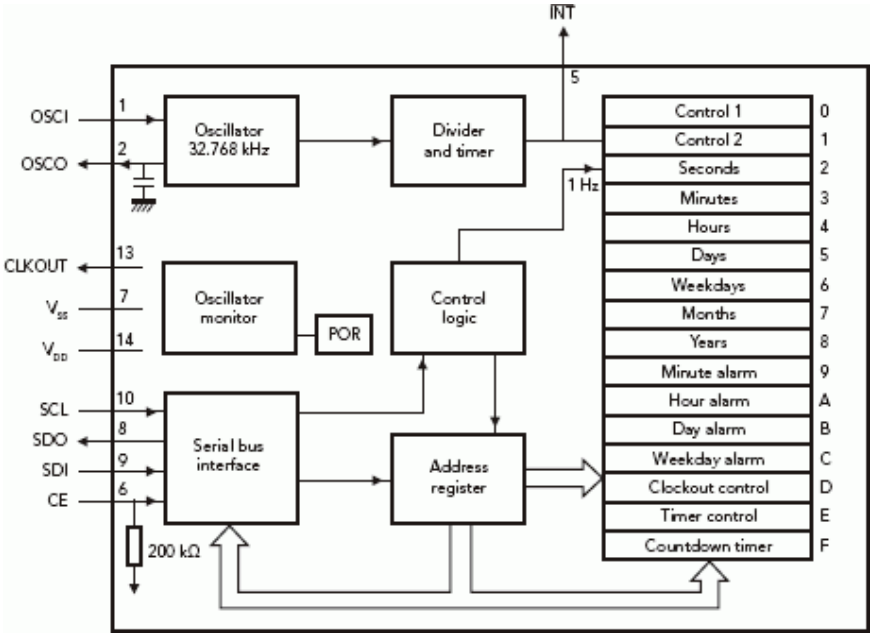
- Wall clock date & time
  - Date: Month, Day, Year
  - Time: HH:MM:SS:mmm
  - Provided by a “real-time clock” or RTC
- Alarm: do something (call code) at certain time later
  - Later could be a delay from now (e.g., Δt)
  - Later could be actual time (e.g., today at 3pm)
- Stopwatch: measure (elapsed) time of an event
  - Instead of pushbuttons, could be function calls or
  - Hardware signals outside the processor

Timer requirements

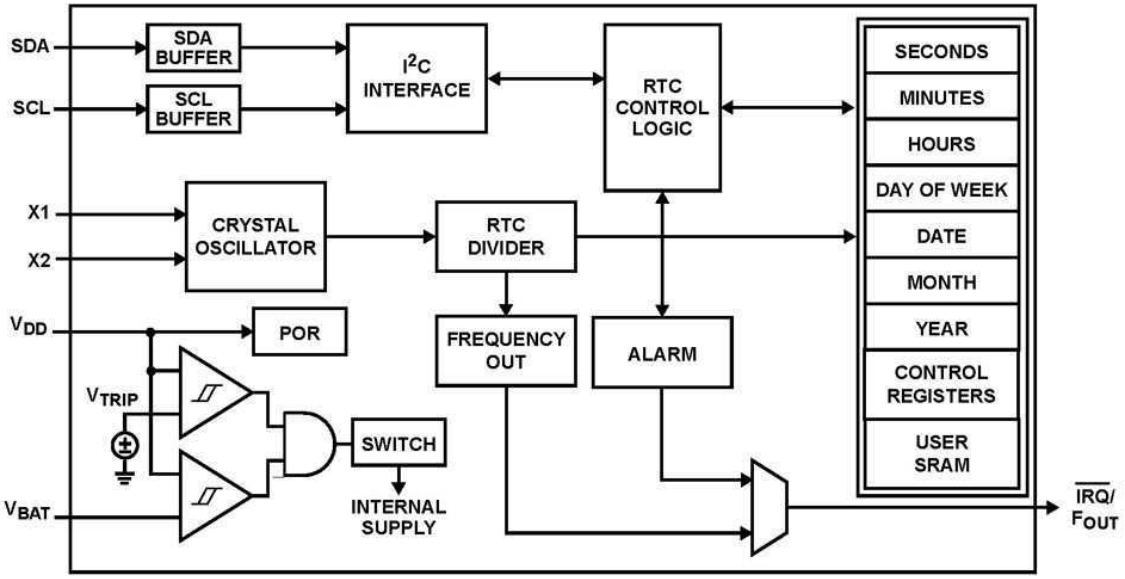
- Wall clock
  - datetime_t getDateTime()
- Alarm
  - void alarm(callback, delta)
  - void alarm(callback, datetime_t)
- Stopwatch: measure (elapsed) time of an event
  - t1 = now(); … ; t2 = now(); dt = difftime(t2, t1);
  - GPIO_INT_ISR:

```
LDR R1, [R0, #0]      % R0=timer address
```

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)

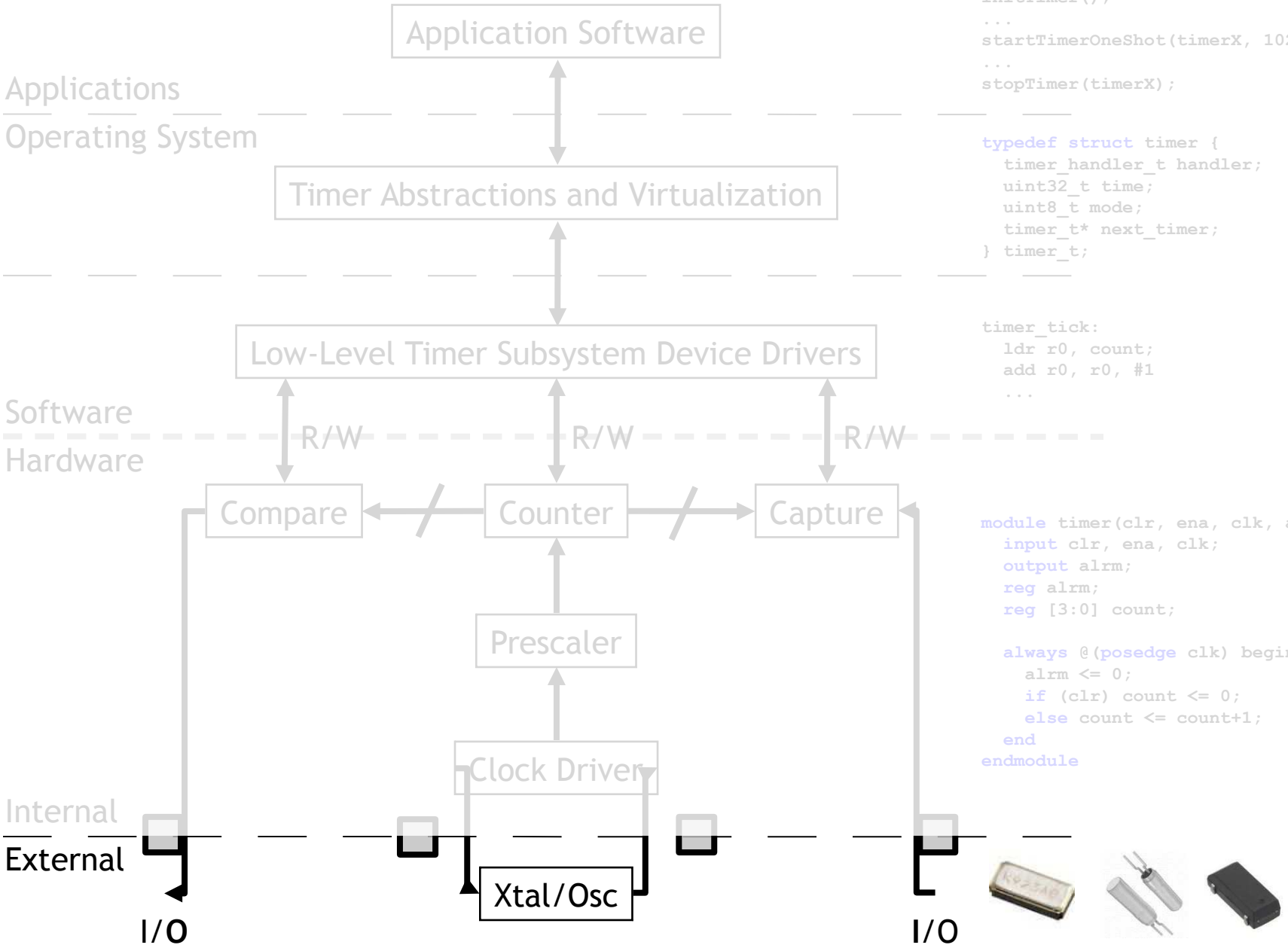


Timer requirements

- 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
```

Anatomy of a timer system




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

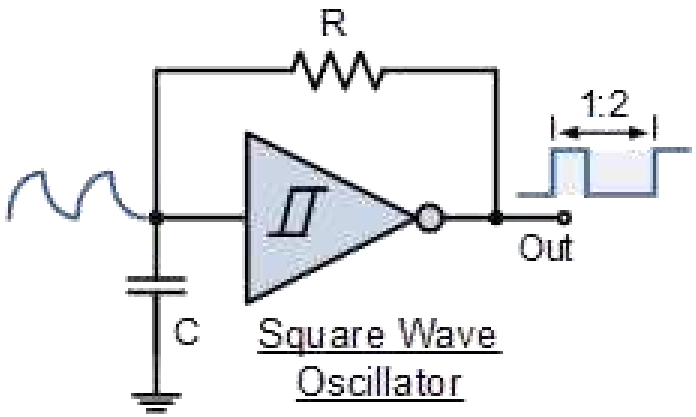
```
typedef struct timer {
  timer_handler_t handler;
  uint32_t time;
  uint8_t mode;
  timer_t* next_timer;
} timer_t;
```

```
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

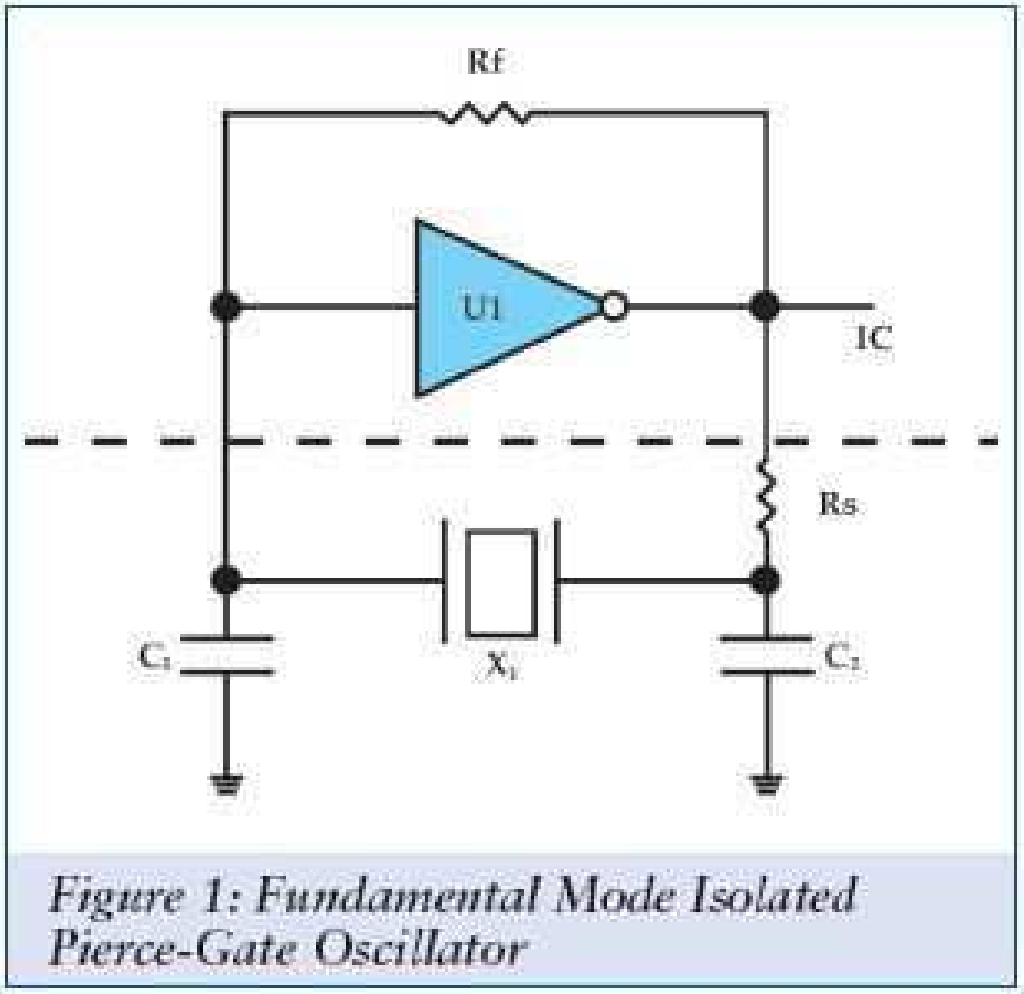
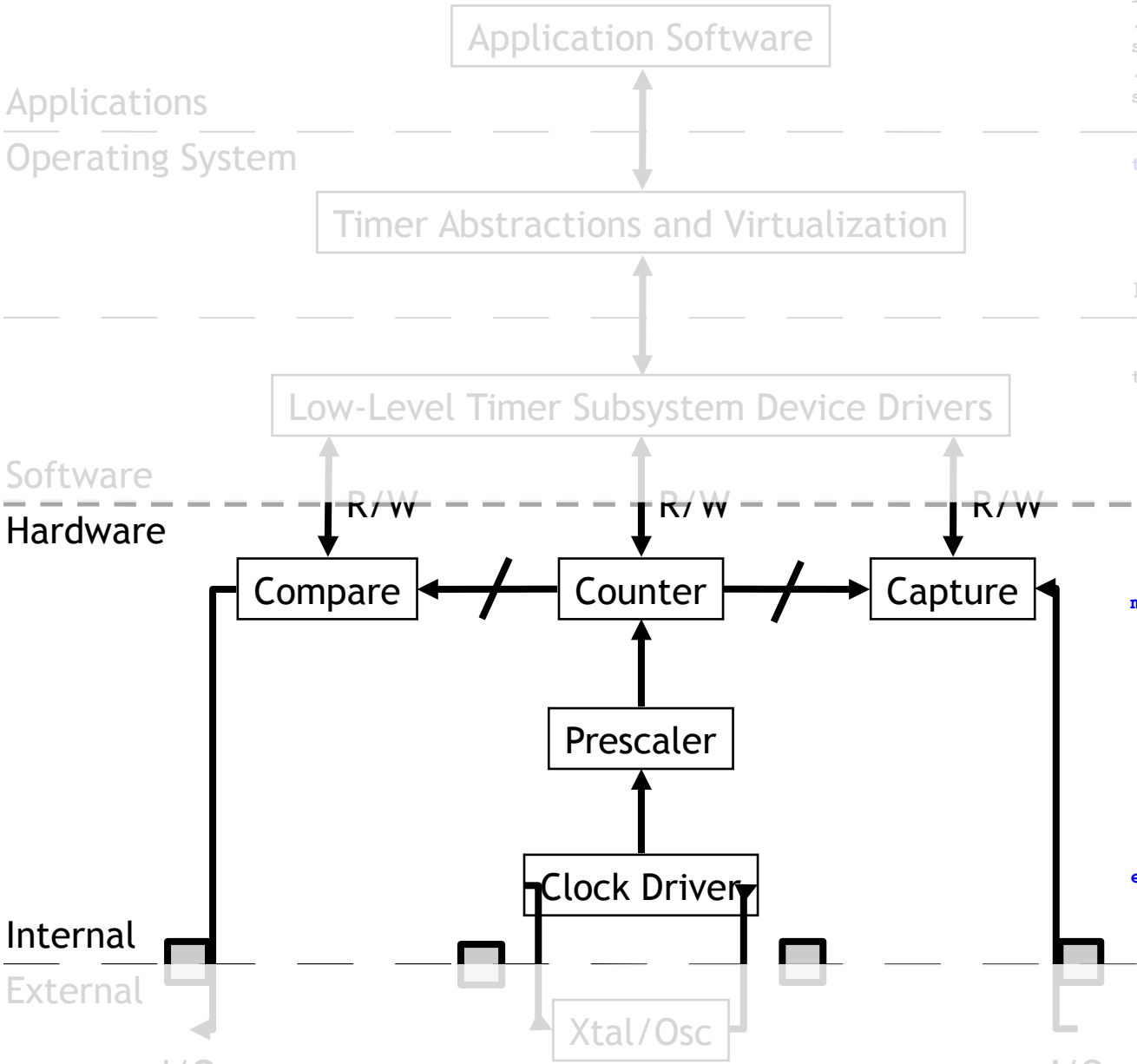


Figure 1: Fundamental Mode Isolated Pierce-Gate Oscillator

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
```

Timer requirements

- 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 applications

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 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?

Outline

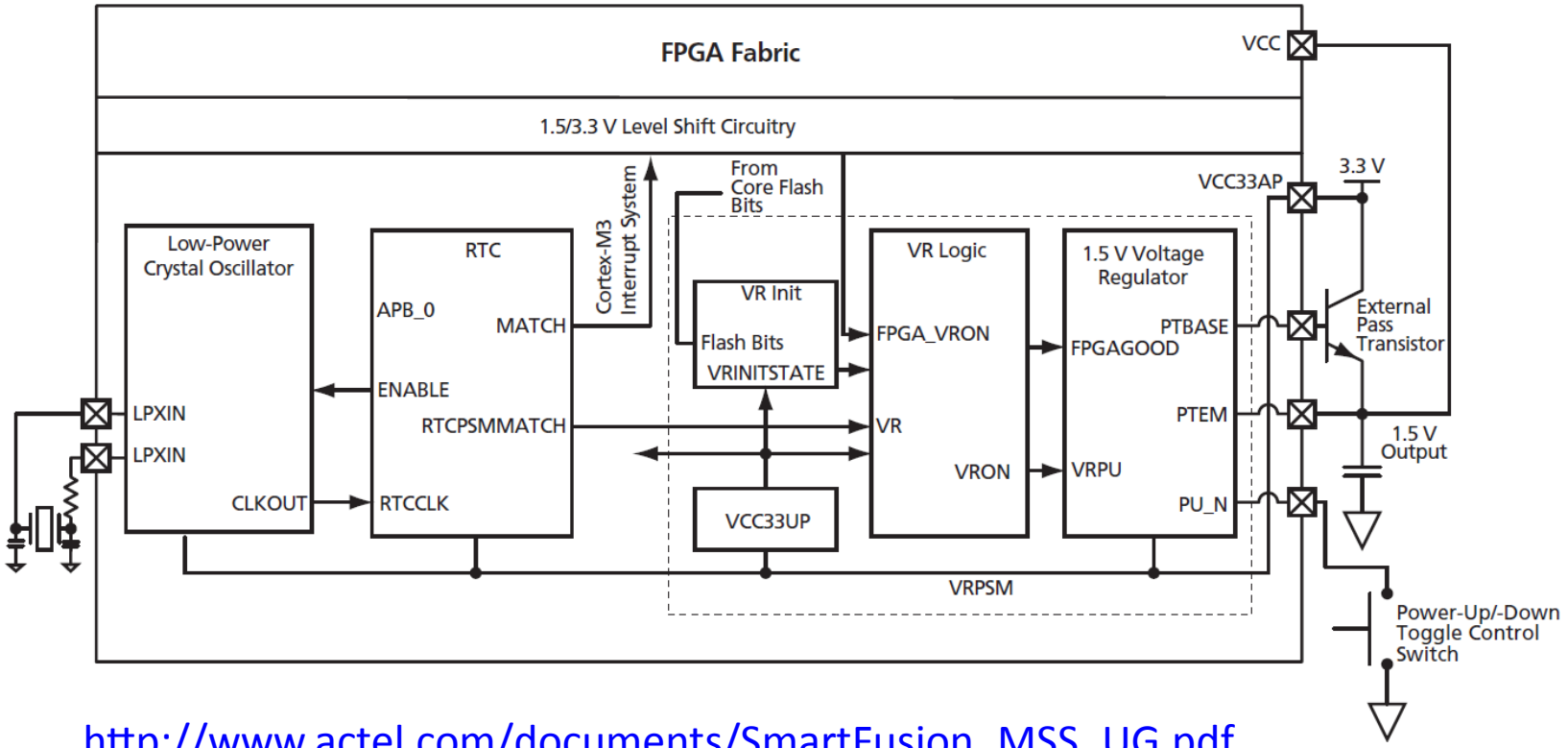
- ~~Context and review~~
- ~~Midterm exam~~
- ~~Project and topic talks~~
- Timers
  - ~~General characteristics~~
  - SmartFusion board
- Hazards

Timers on the SmartFusion

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

Timers on the SmartFusion

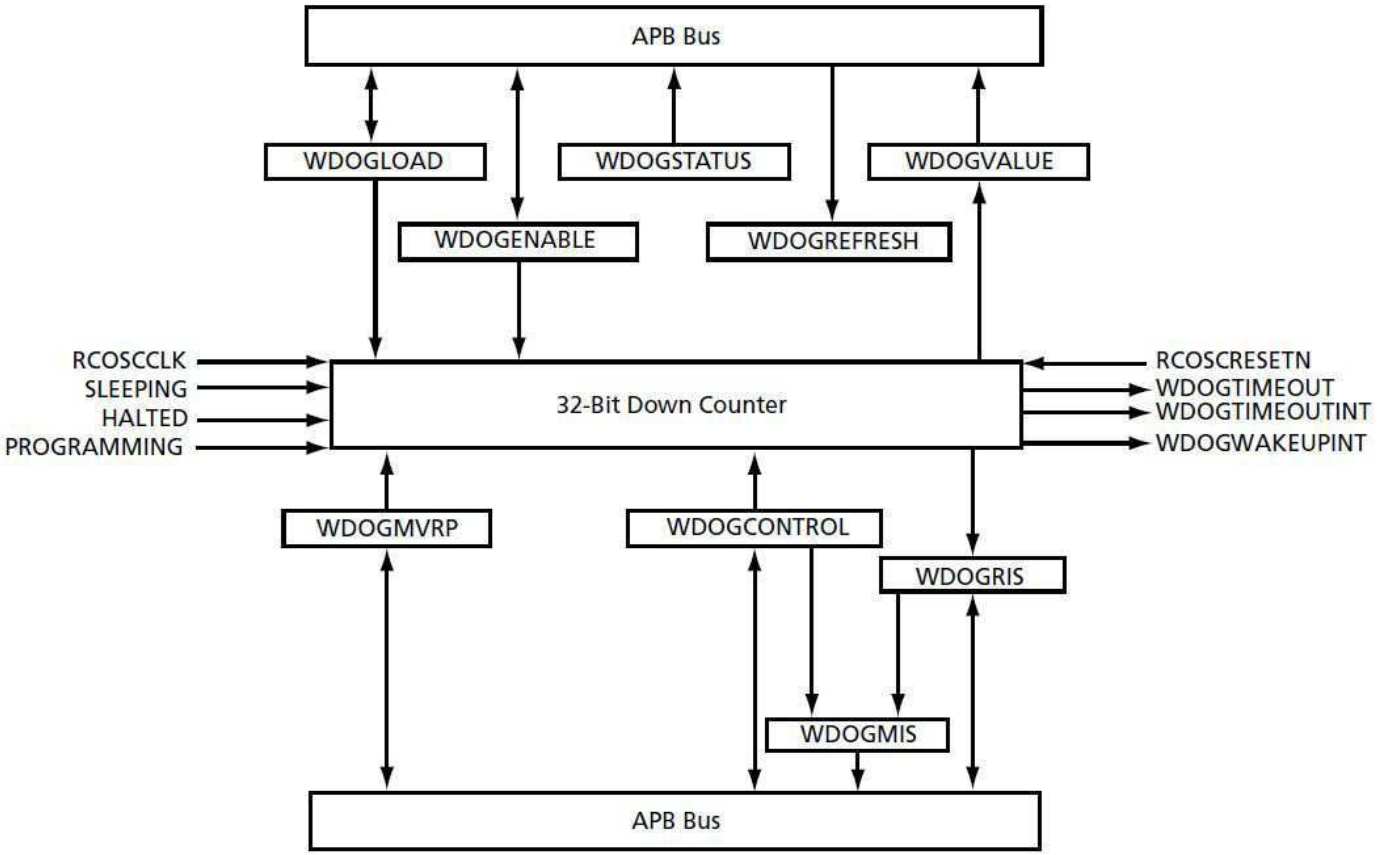
- Real-Time Counter (RTC) System
  - Clocked from 32 kHz low-power crystal
  - Automatic switching to battery power if necessary
  - Can put rest of the SmartFusion to standby or sleep to reduce power
  - 40-bit match register clocked by 32.768 kHz divided by 128 (256 Hz)



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!

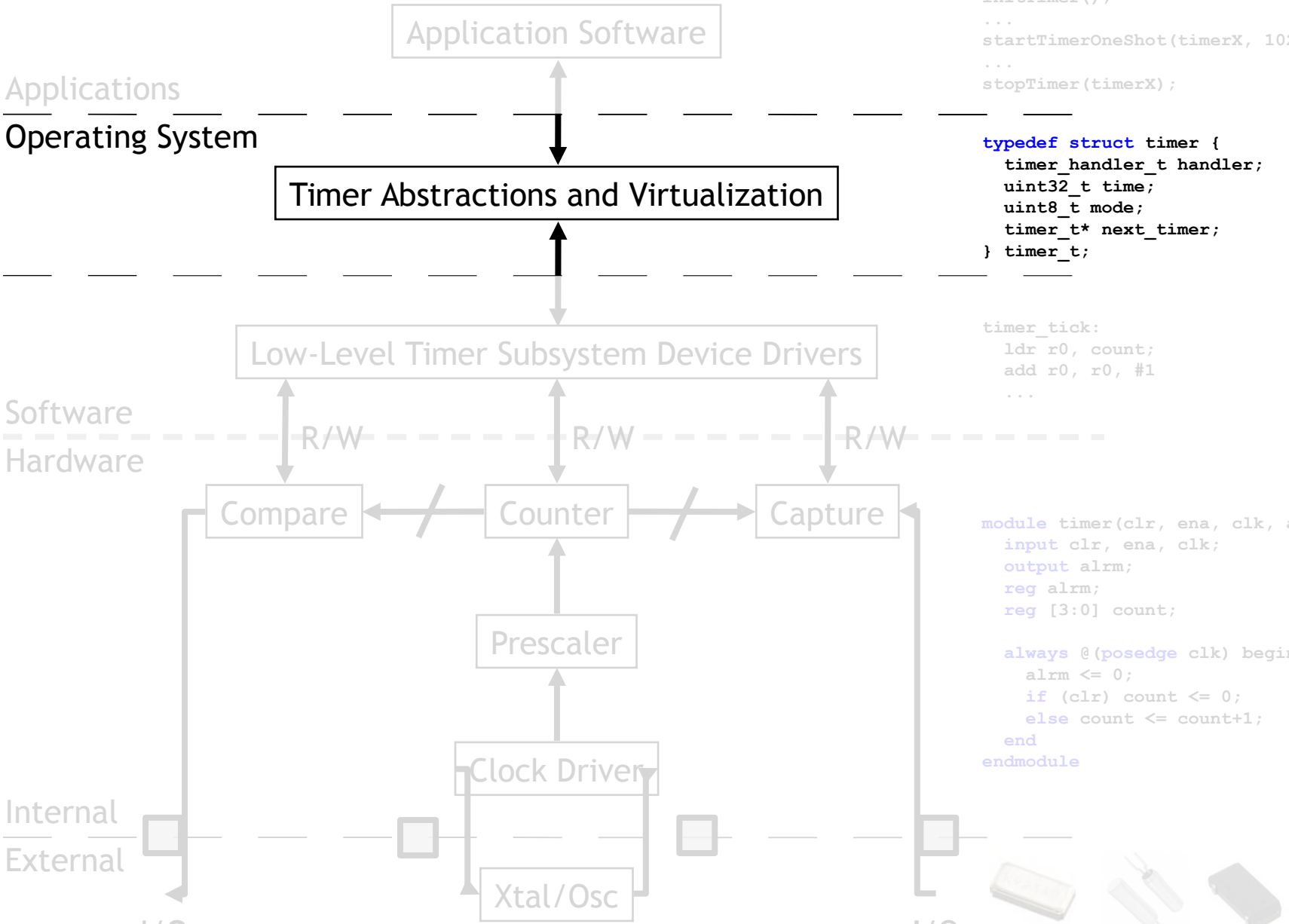


Timers on the SmartFusion

- 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

- Can we use more timers than exist in hardware?
- Yes. Use hardware timers as a foundation for software-controlled virtual timers.
- Maybe we have 10 events we might want to generate.
- Make a list of them and set the timer to go off for the first one.
- Repeat.

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
}
```

Outline

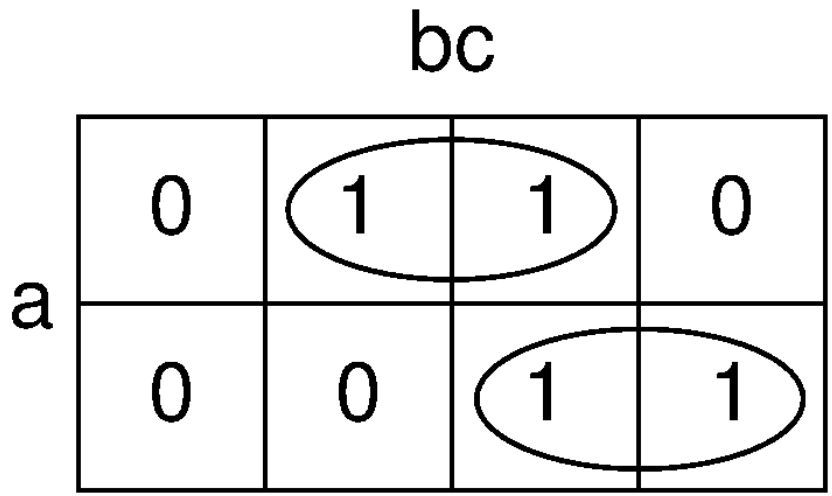
- ~~Context and review~~
- ~~Midterm exam~~
- ~~Project and topic talks~~
- ~~Timers~~
  - ~~General characteristics~~
  - ~~SmartFusion board~~
- Hazards

Hazards

- Race between variable transitions.
- May, but not must, produce a glitch.
- Glitch
  - Static glitch: transient pulse of incorrect value when output should be stable.
  - Dynamic glitch: transient pulse of incorrect value when output should be changing.
- Consider a minimal implementation of
  - f(a, b, c) = a'b'c + a'bc + abc + abc'

Hazards

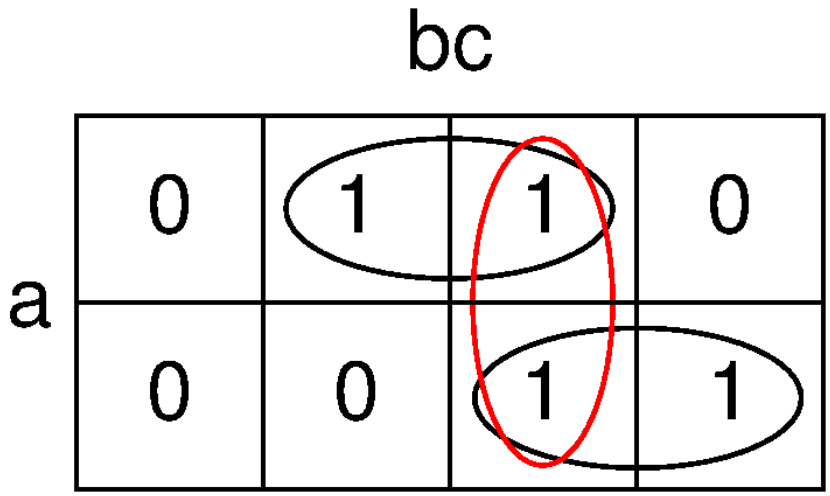
- Consider a minimal implementation of
  - f(a, b, c) = a'b'c + a'bc + abc + abc'



- f(a, b, c) = a'c + ab
- What if b=1, c=1?

Hazards

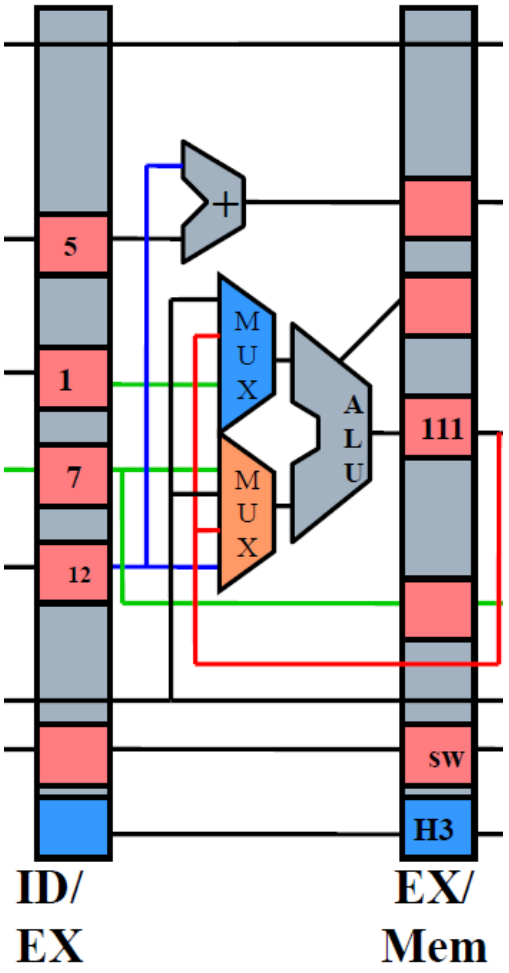
- How to eliminate
  - Limit logic to two levels
  - Cover all transitions



- f(a, b, c) = a'c + ab + bc
- What if b=1, c=1?

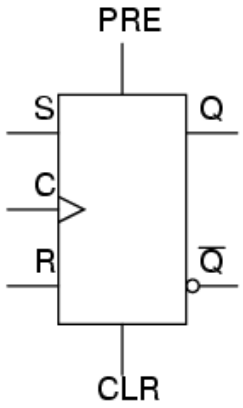
Effect of hazards

- Hazards can often be ignored in synchronous systems.
- Only sampling on clock edges.
- Make clocks slow enough to permit glitching to finish before next edge.
- Still wastes power.
- Causes major problems in asynchronous systems.
  - Different design style required.

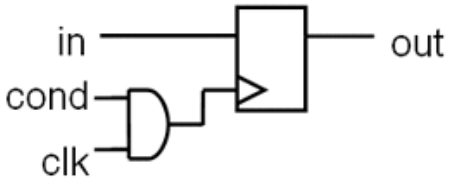


When hazards need special attention

- 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.
    - Unless the number of times reset doesn't matter.
  - Can use a flip-flop on the input.
    - You probably needed a synchronous reset in this case.
  - Clocks
    - Hazards in logic driving clocks can produce spurious clock edges.



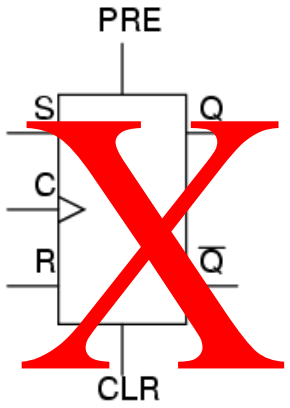
Traditionally, CLR is used to indicate async reset. "R" or "reset" for sync. reset.

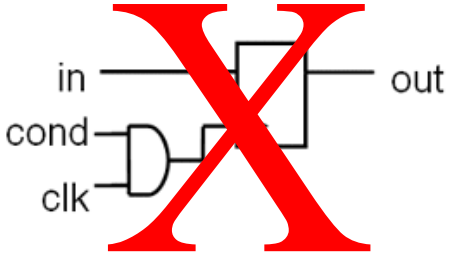


If clk is high and cond glitches, you get extra edges!

Simple design rules

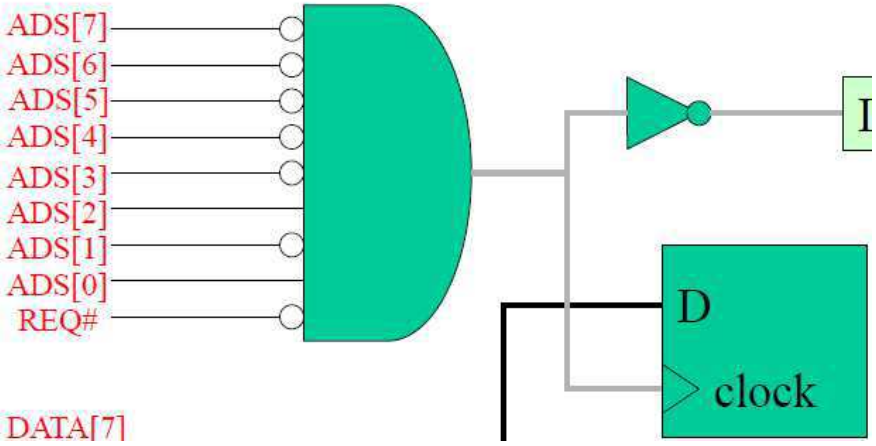
- Don't use asynchronous resets unless you understand the implications fully.
- Don't drive a clock with logic containing hazards.
- Hazard-free guarantee.
  - Only two levels.
  - Cover transitions.
    - Literal or complement, not both.





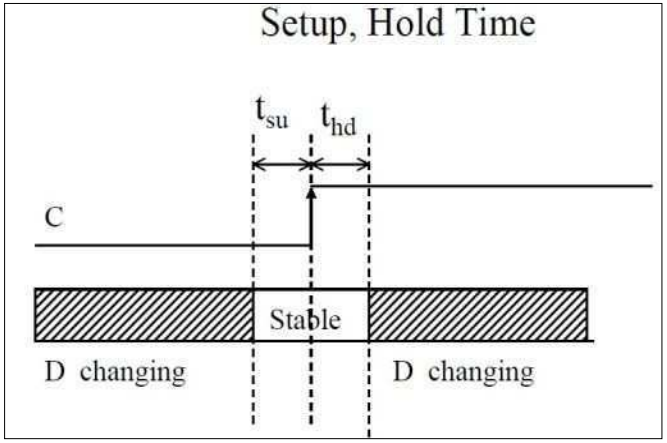
Glitches

- People use asynchronous resets and clock gating!
  - Think carefully
- Our “simple” bus used
- Combinational logic for the clock
  - Works because REQ goes low only after everything else has stopped switching.
  - Might also be safe even if this weren't true.
  - Need detailed analysis of implementation to know.



Setup and hold time

- If active clock edge and data change at same time.
- Then data latched is unclear.
- Often worse for registers than single flip-flops.
  - Inconsistent state.
- Use temporal “guard band” around clock edge
- Data must be stable there.
- Setup time.
- Hold time.



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 to a "0" or a "1" quickly.
    - That could cause later 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.

Example

- 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.
  - Or explicitly design a fundamental mode AFSM.
    - I get yelled at for teaching this to undergrads but tell me if you are curious. Could do an "optional" talk in second half of course.

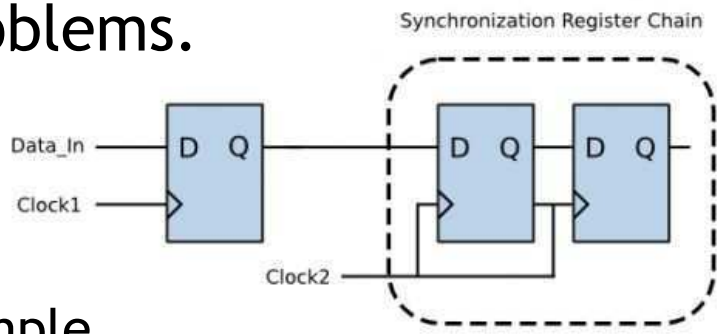
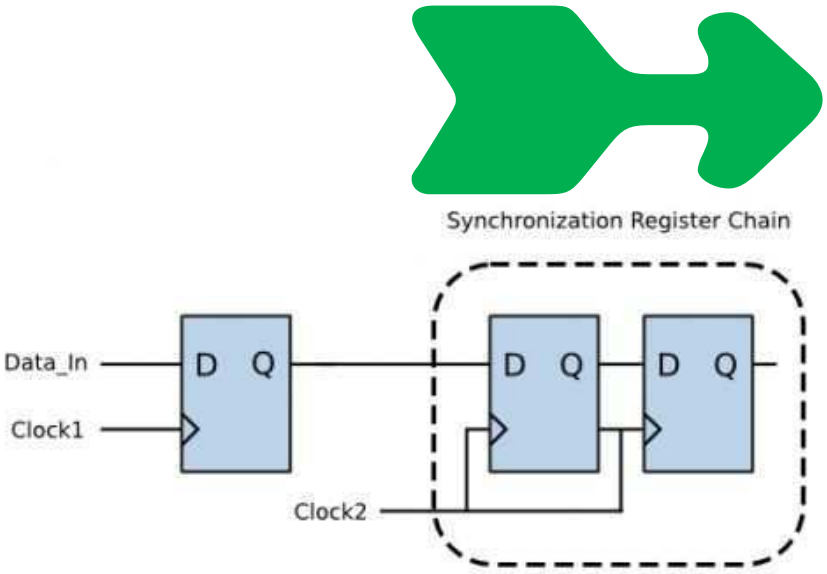


Figure from http://www.eeweb.com/electronics-quiz/solving-metastability-design-issues, we use the same thing to deal with external inputs too!

Design rules

3. 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};
```