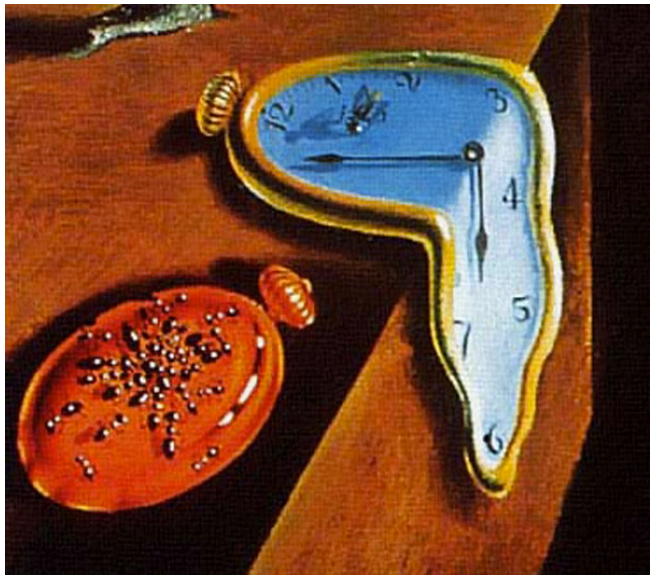
EECS 373
Design of Microprocessor-Based Systems

Ron Dreslinski
University of Michigan



Clocks, Counters, Timers, Capture, and Compare

Some slides by Mark Brehob, Prabal Dutta and Thomas Schmid

Announcements

- HW3 Posted on web (Due Next Wednesday)
- Project Meeting
  - Monday 10/10 @ 6:30pm
  - 1003 EECS
- Project Description Posted on Website this afternoon

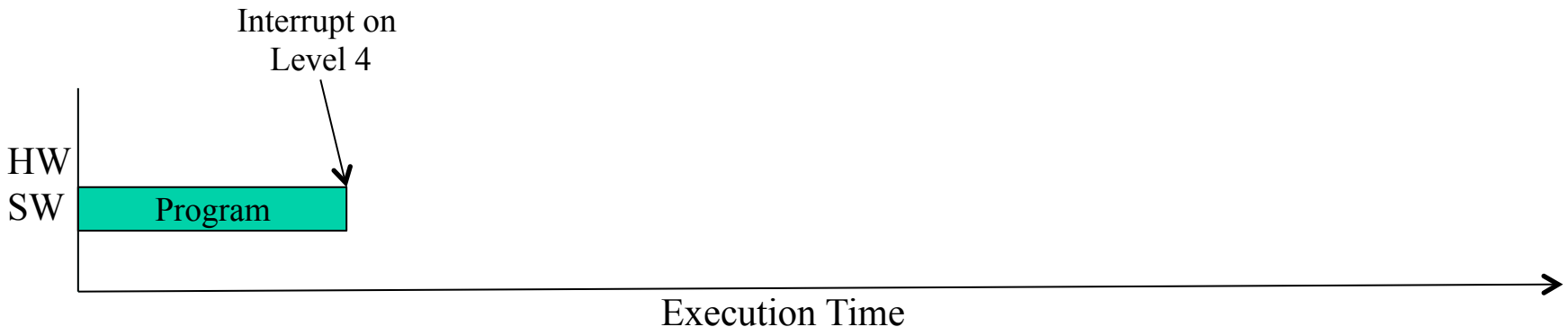
What happens when we return from an ISR?

- Interrupt exiting process
  - System restoration needed (different from branch)
  - Special LR value could be stored (0xFFFFFFFx)

- Walkthrough a few examples, simplified system assumptions:
  - 5 Interrupt Levels (0-4)
  - 256 Priority Levels (0 Highest, 256 Lowest)

- First time presenting, so errors in the animation are unintentional but may be present (lets find out together)

Single Interrupt



Program Stack

Priority Table

0	8
1	10
2	5
3	3
4	6

Pending

4				0
0	0	0	0	0

Enabled

4				0
1	1	1	1	1

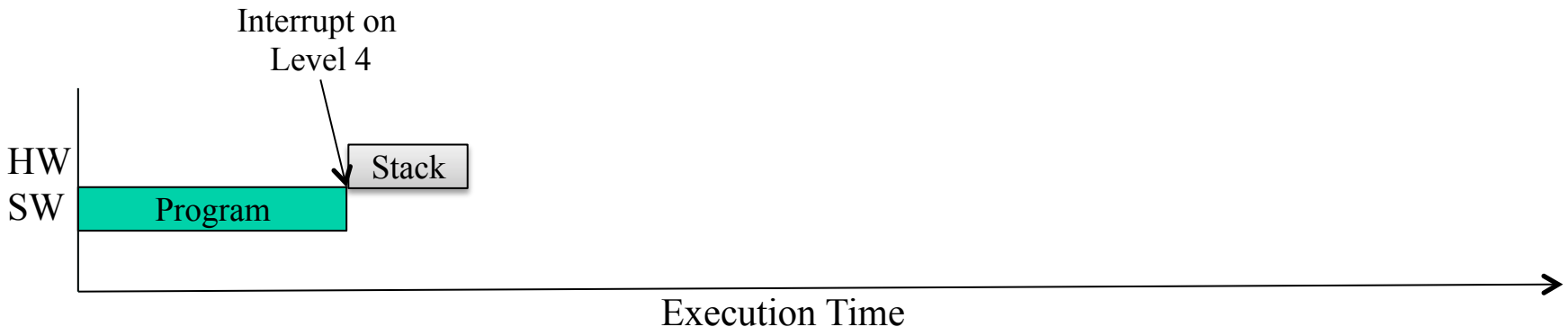
Base Priority

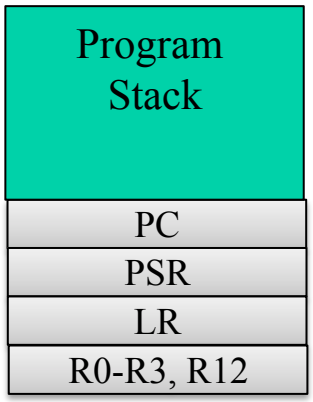
256

Active

4				0
0	0	0	0	0

Single Interrupt





Priority Table

0	8
1	10
2	5
3	3
4	6

Pending

4				0
1	0	0	0	0

Enabled

4				0
1	1	1	1	1

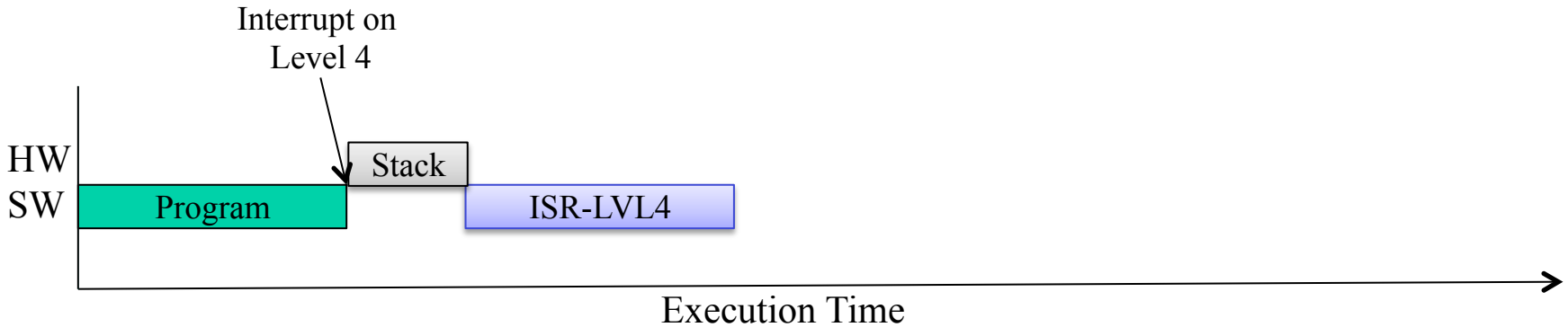
Base Priority

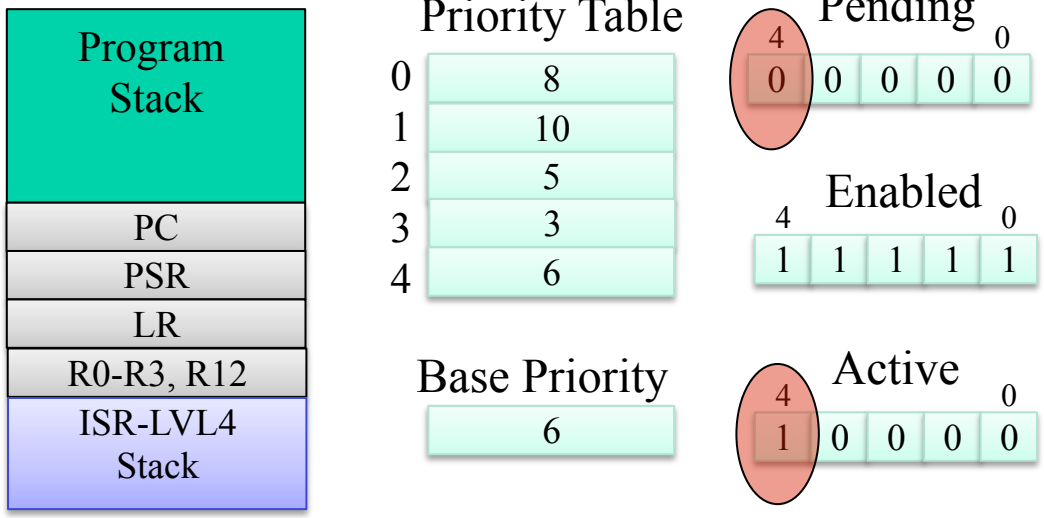
6

Active

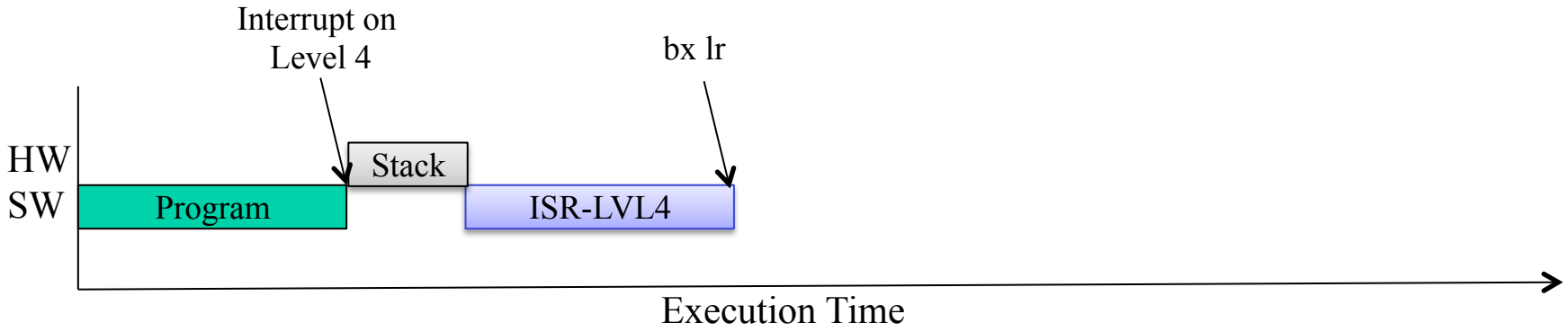
4				0
0	0	0	0	0

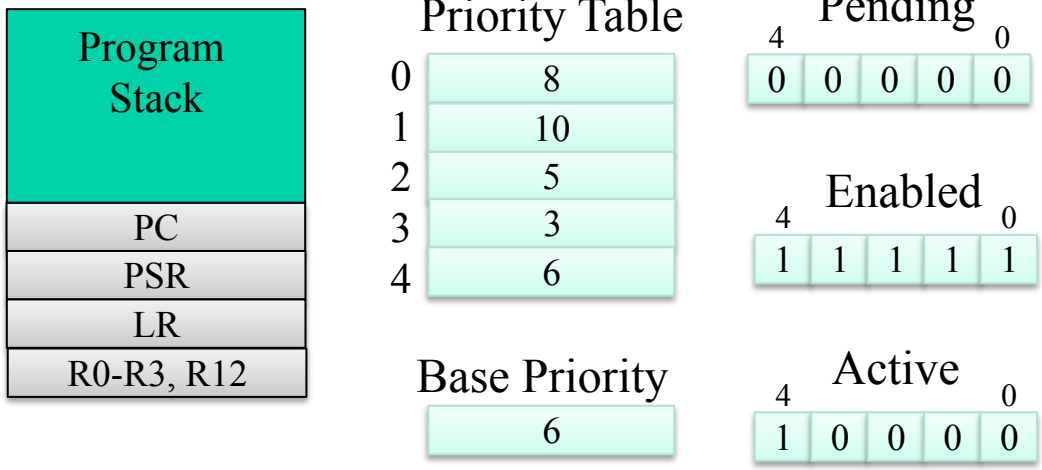
Single Interrupt



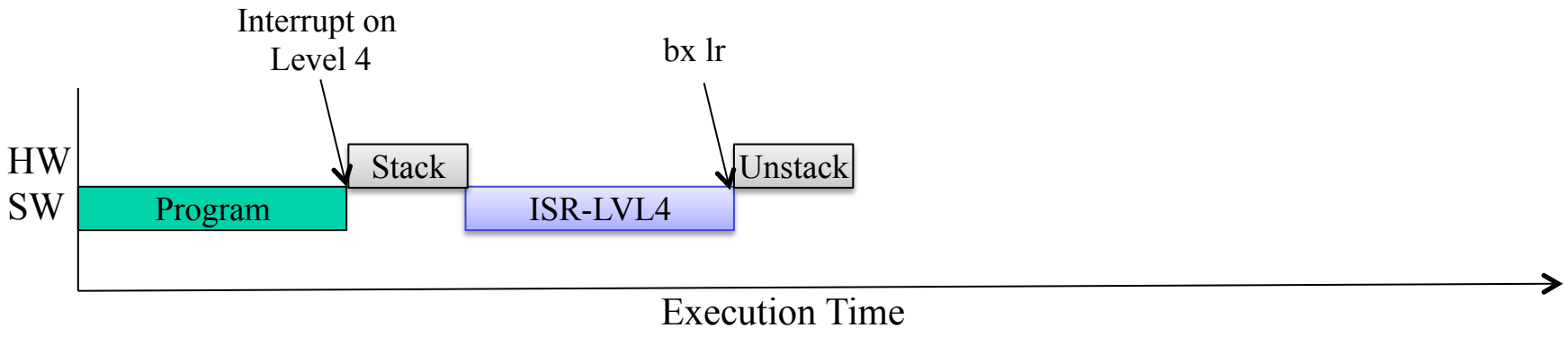


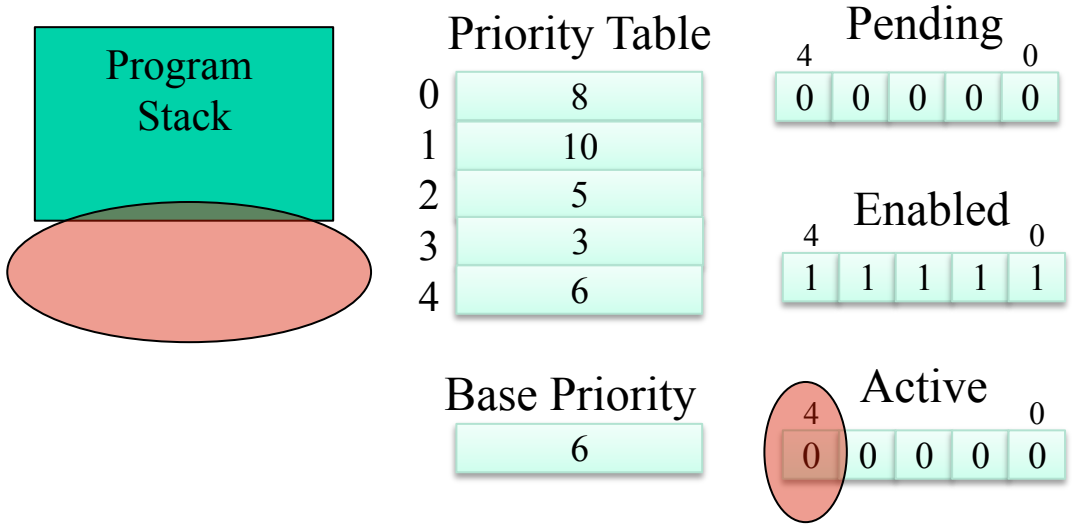
Single Interrupt



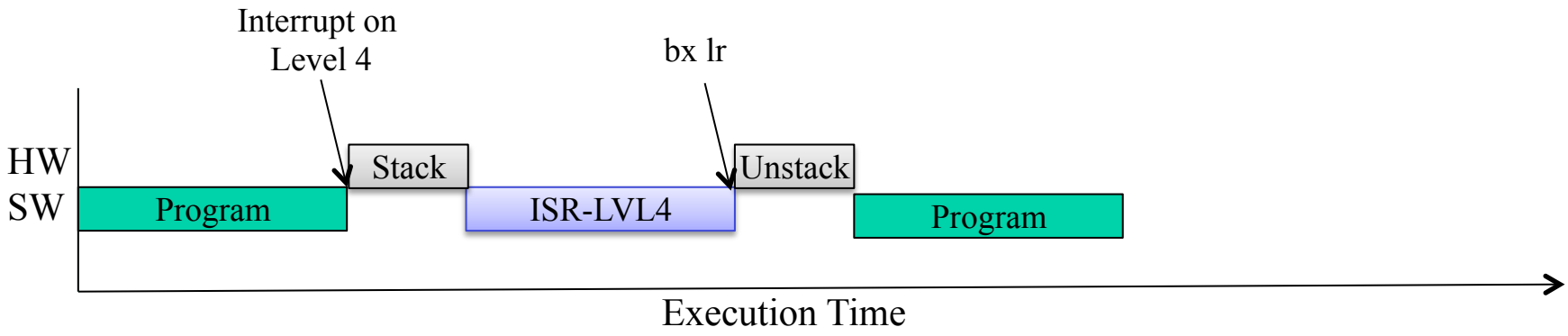


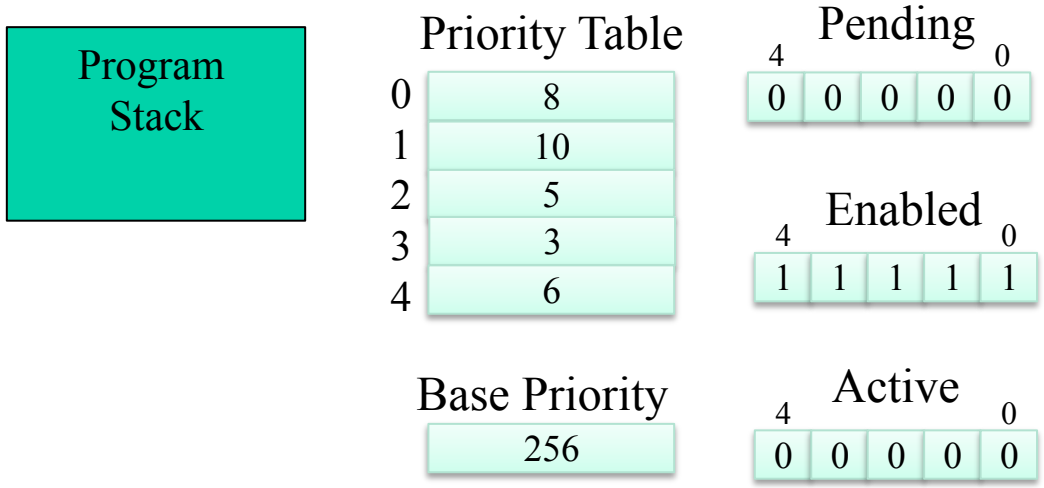
Single Interrupt





Single Interrupt

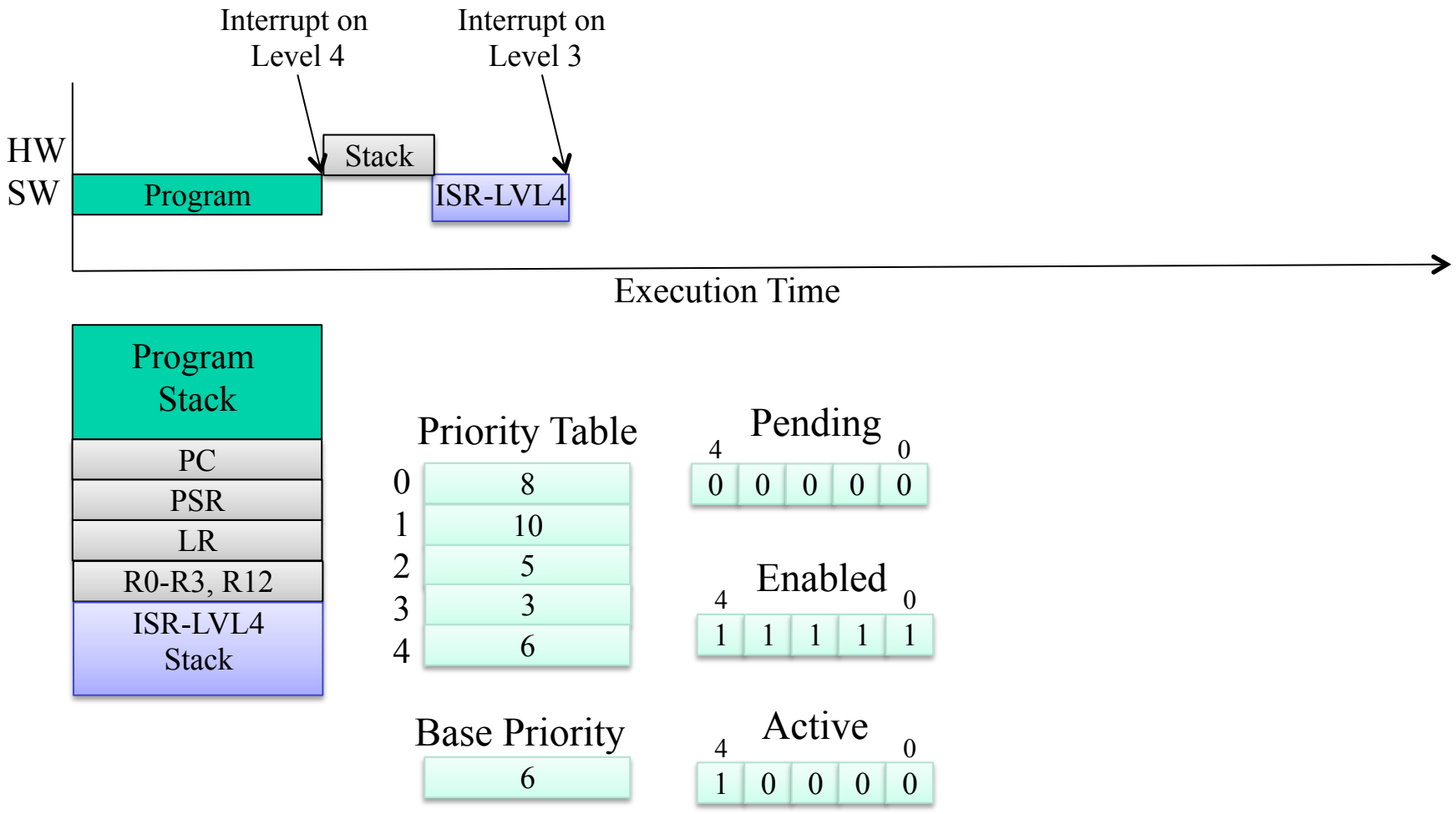




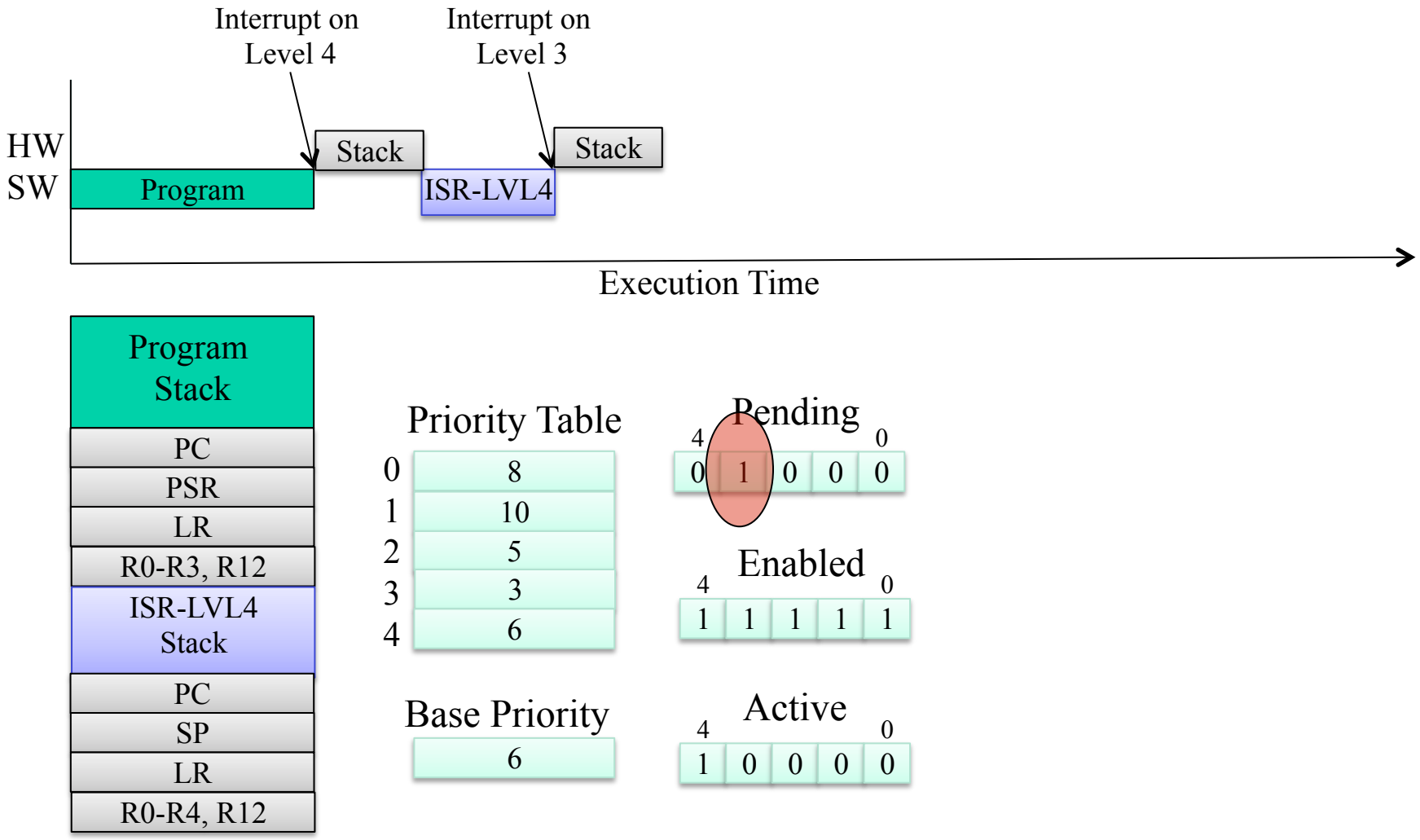
Nested Interrupts

- When new exception occurs
- And CPU handling another exception of lower priority (incoming request is higher priority)
- New exception will interrupt the current ISR
- Will generate a new ISR stack on the stack

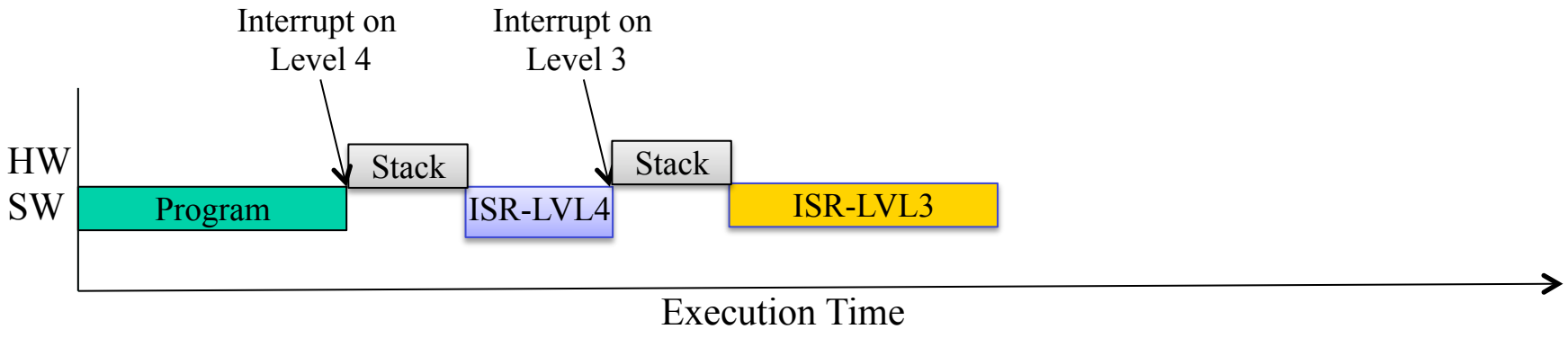
Nested Interrupt

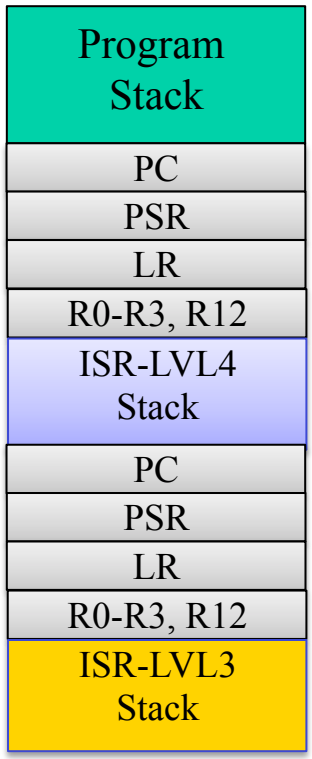


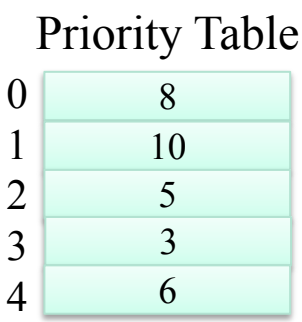
Nested Interrupt

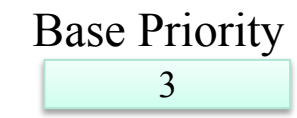


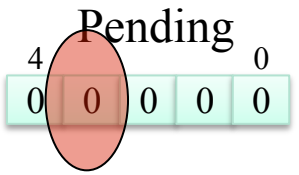
Nested Interrupt

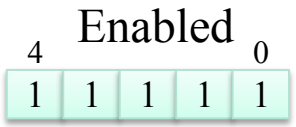


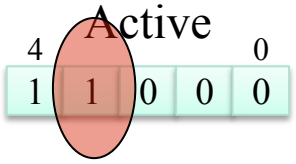




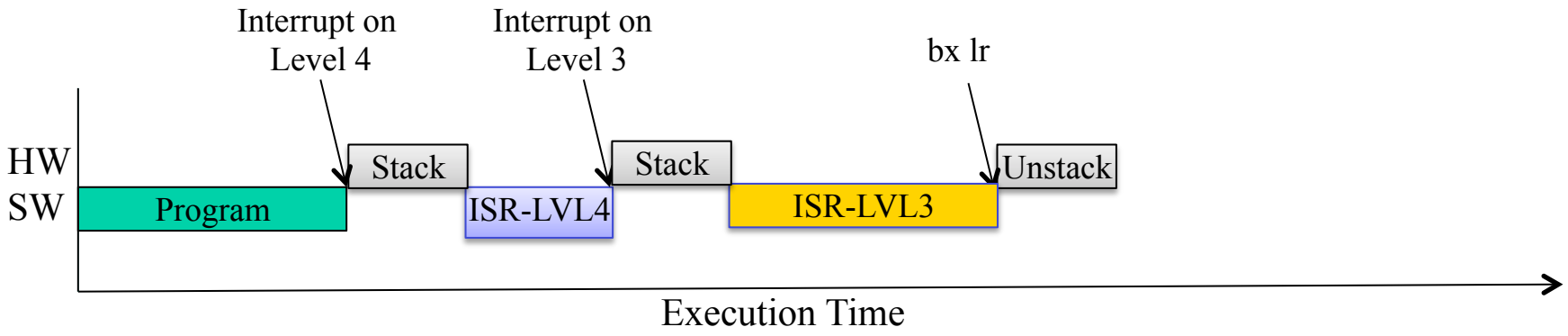


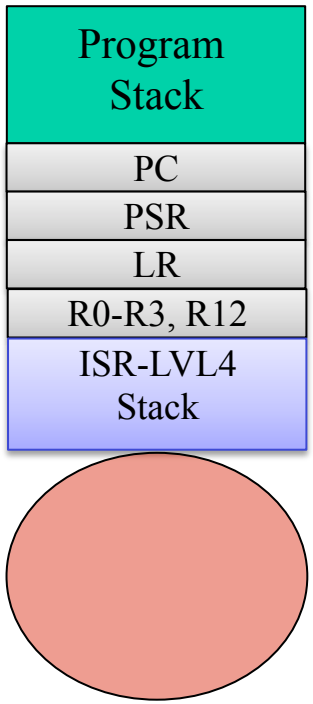






Nested Interrupt





Priority Table

0	8
1	10
2	5
3	3
4	6

Pending

4				0
0	0	0	0	0

Enabled

4				0
1	1	1	1	1

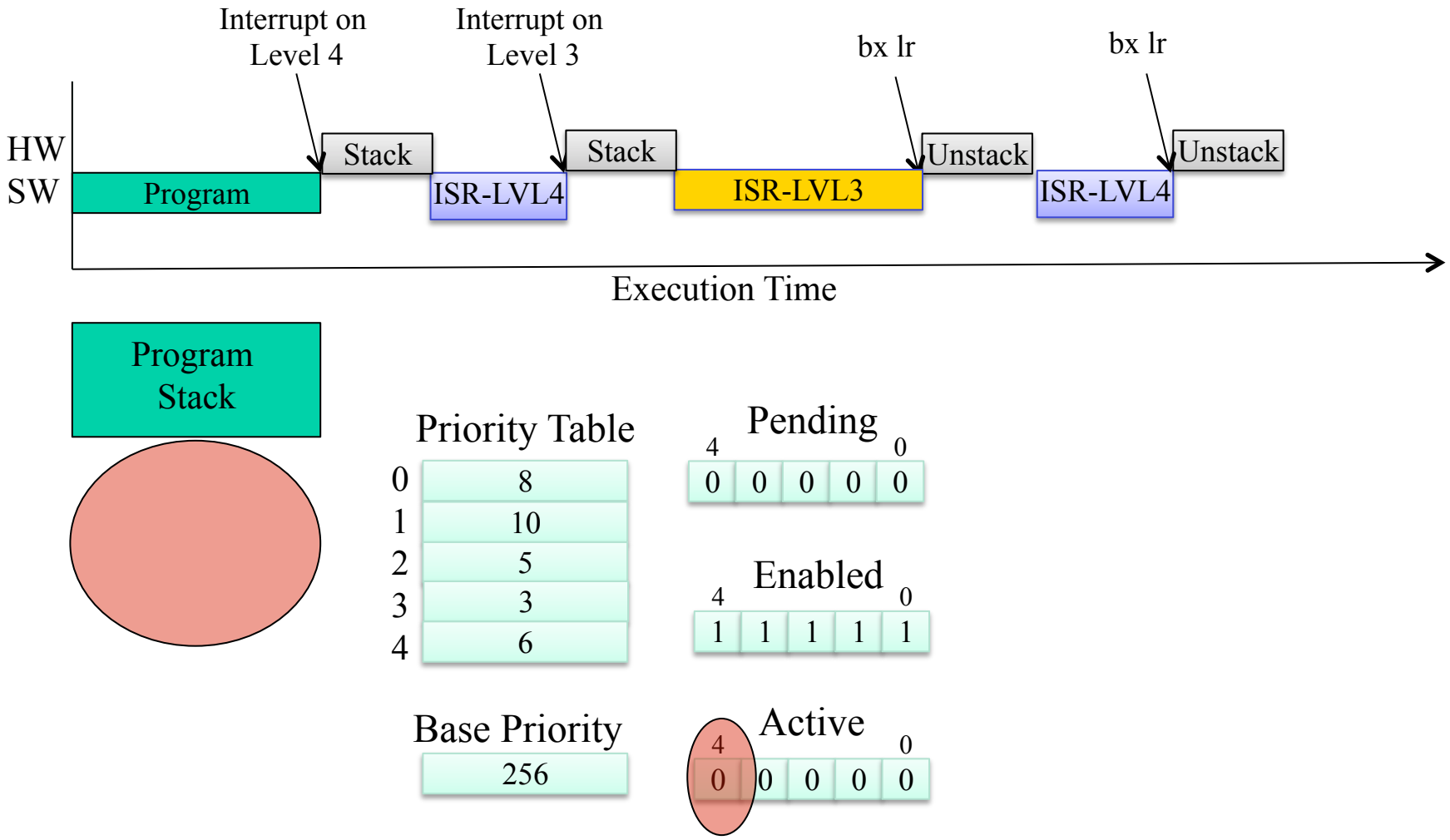
Base Priority

6

Active

4				0
1	0	0	0	0

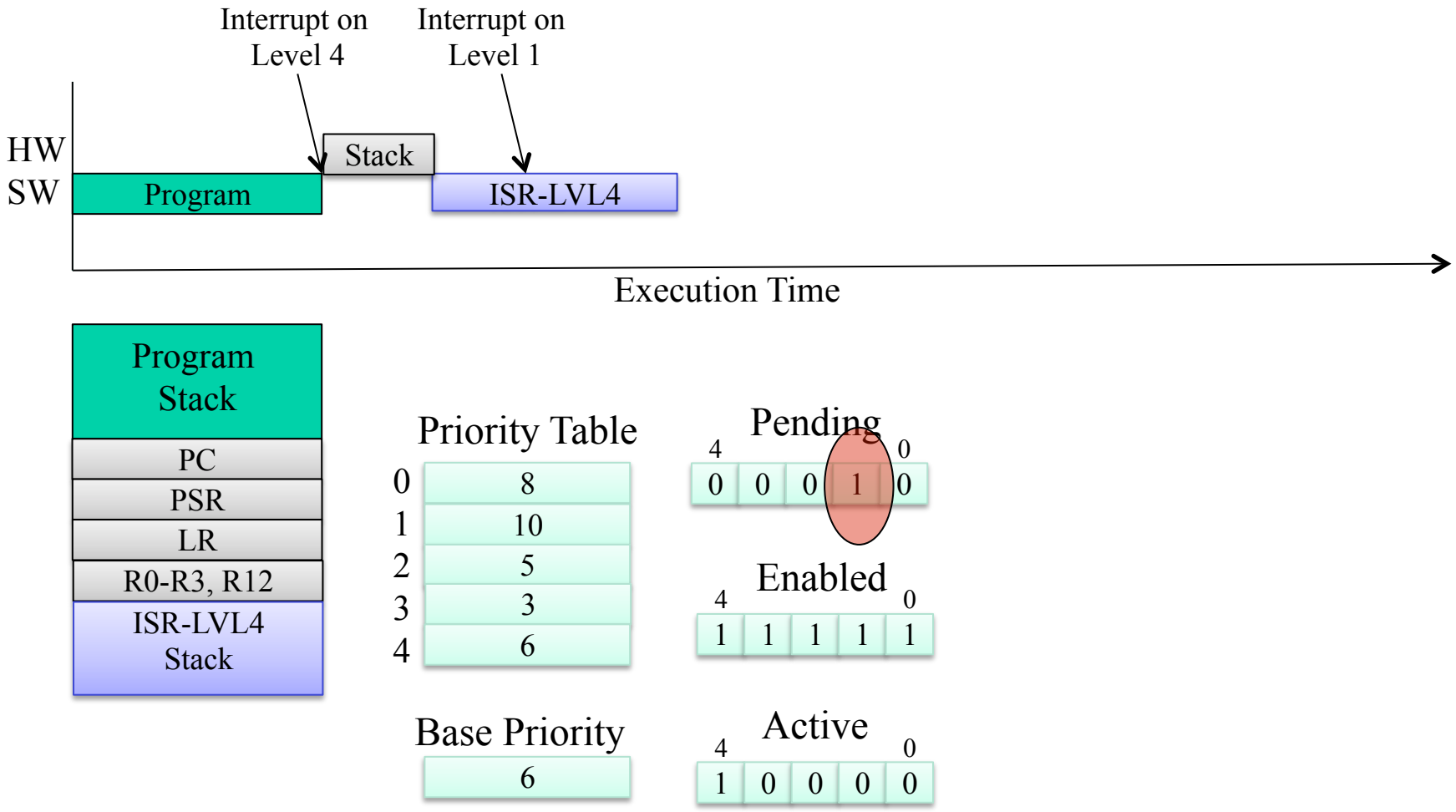
Nested Interrupt



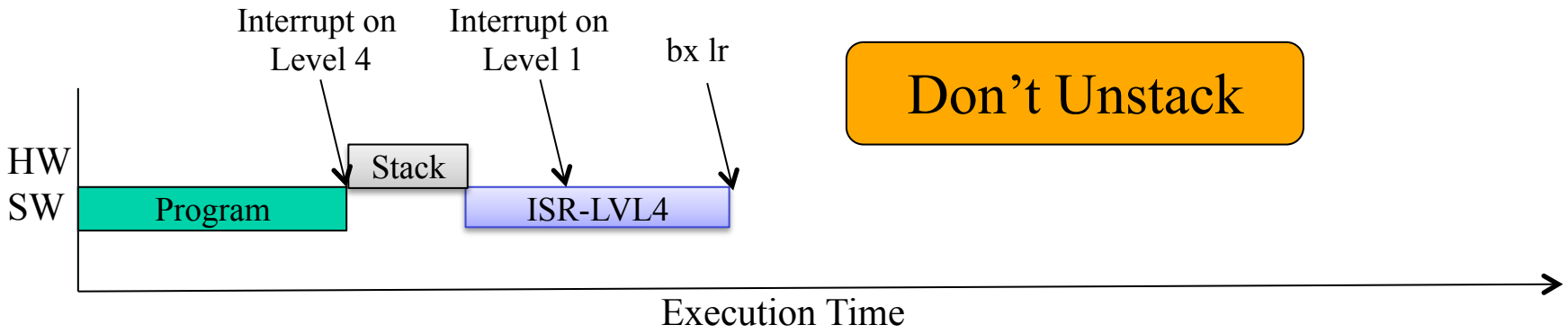
Tail Chaining

- When new exception occurs
- But CPU handling another exception of same/higher priority (incoming request is lower priority)
- New exception will enter pending state
- But will be executed before register unstacking
- Saving unnecessary unstacking/stacking operations
- Can reenter hander in as little as 6 cycles

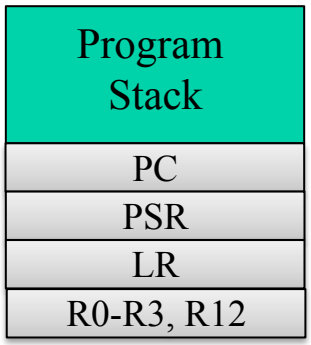
Tail-Chaining



Tail-Chaining



Don't Unstack



Priority Table

0	8
1	10
2	5
3	3
4	6

Pending

4				0
0	0	0	1	0

Enabled

4				0
1	1	1	1	1

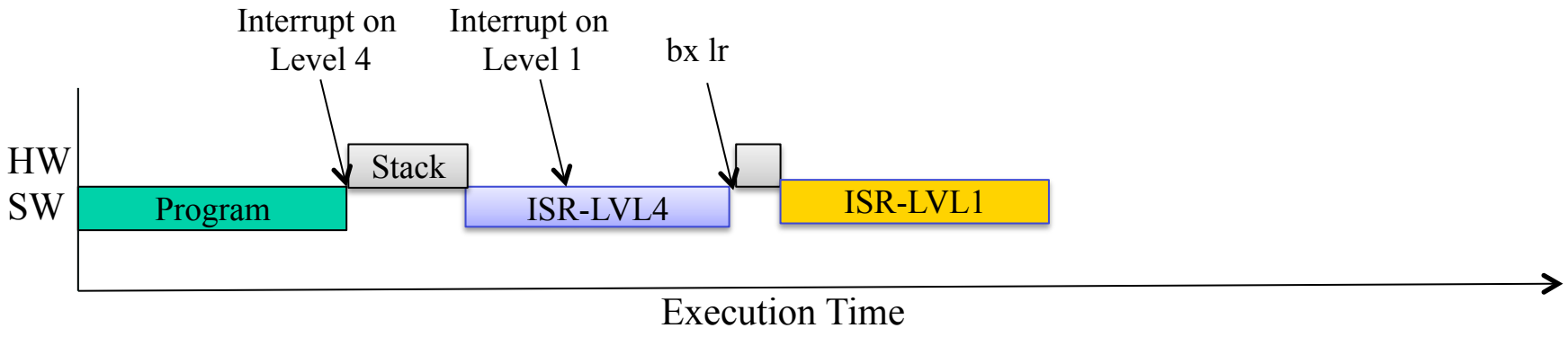
Base Priority

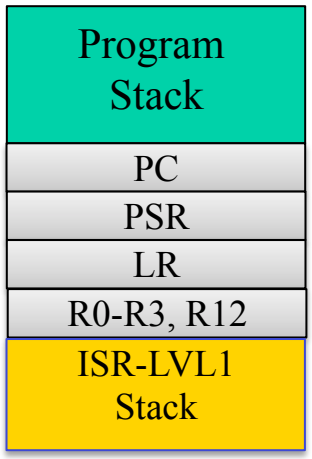
6

Active

4				0
0	0	0	0	0

Tail-Chaining





Priority Table

0	8
1	10
2	5
3	3
4	6

Pending

4				0
0	0	0	0	0

Enabled

4				0
1	1	1	1	1

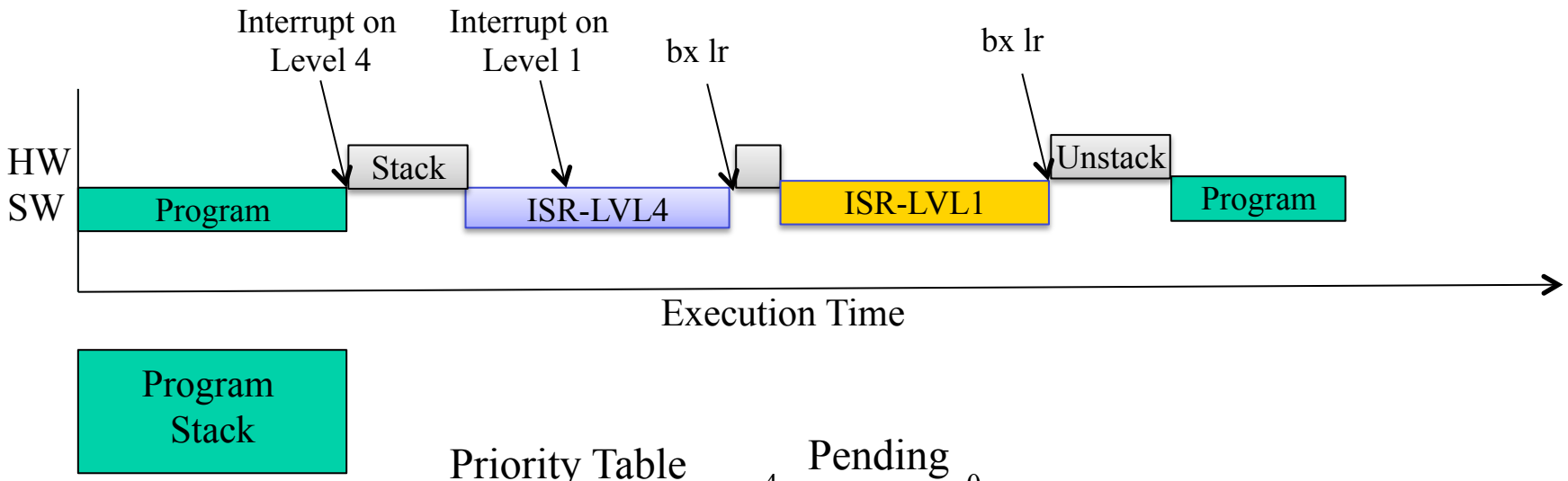
Base Priority

10

Active

4				0
0	0	0	1	0

Tail-Chaining



Program Stack

Priority Table

0	8
1	10
2	5
3	3
4	6

Pending

4				0
0	0	0	0	0

Enabled

4				0
1	1	1	1	1

Base Priority

256

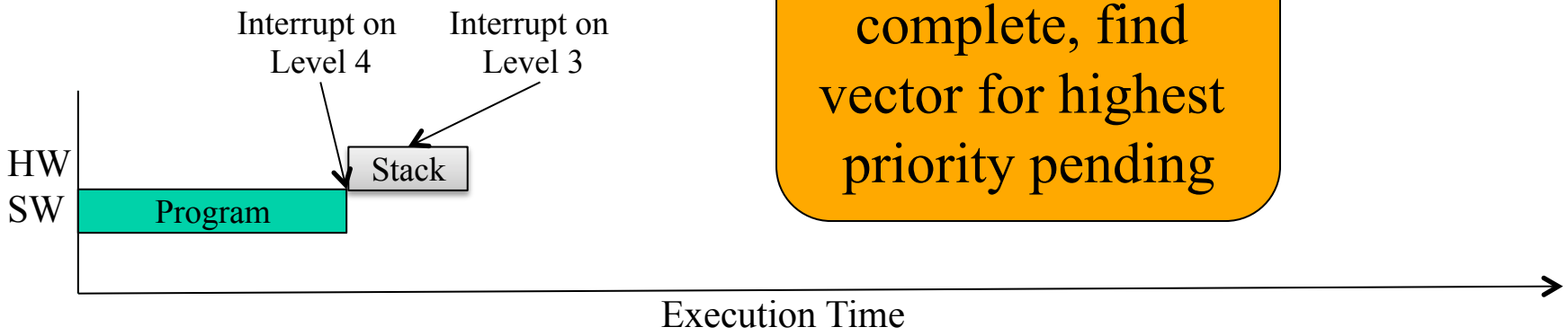
Active

4				0
0	0	0	0	0

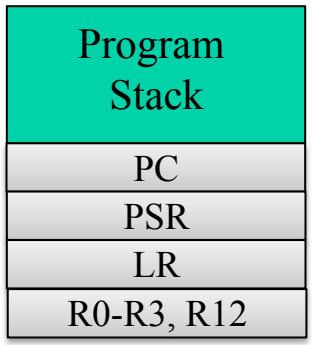
Late Arrival

- Late arrivals (ok, so this is actually on entry)
  - When one exception occurs and stacking commences
  - Then another exception occurs before stacking completes
  - And second exception of higher preempt priority arrives
  - The later exception will be processed first

Late Arrival



Once stacking complete, find vector for highest priority pending



Priority Table

0	8
1	10
2	5
3	3
4	6

Pending

4				0
1	1	0	0	0

Enabled

4				0
1	1	1	1	1

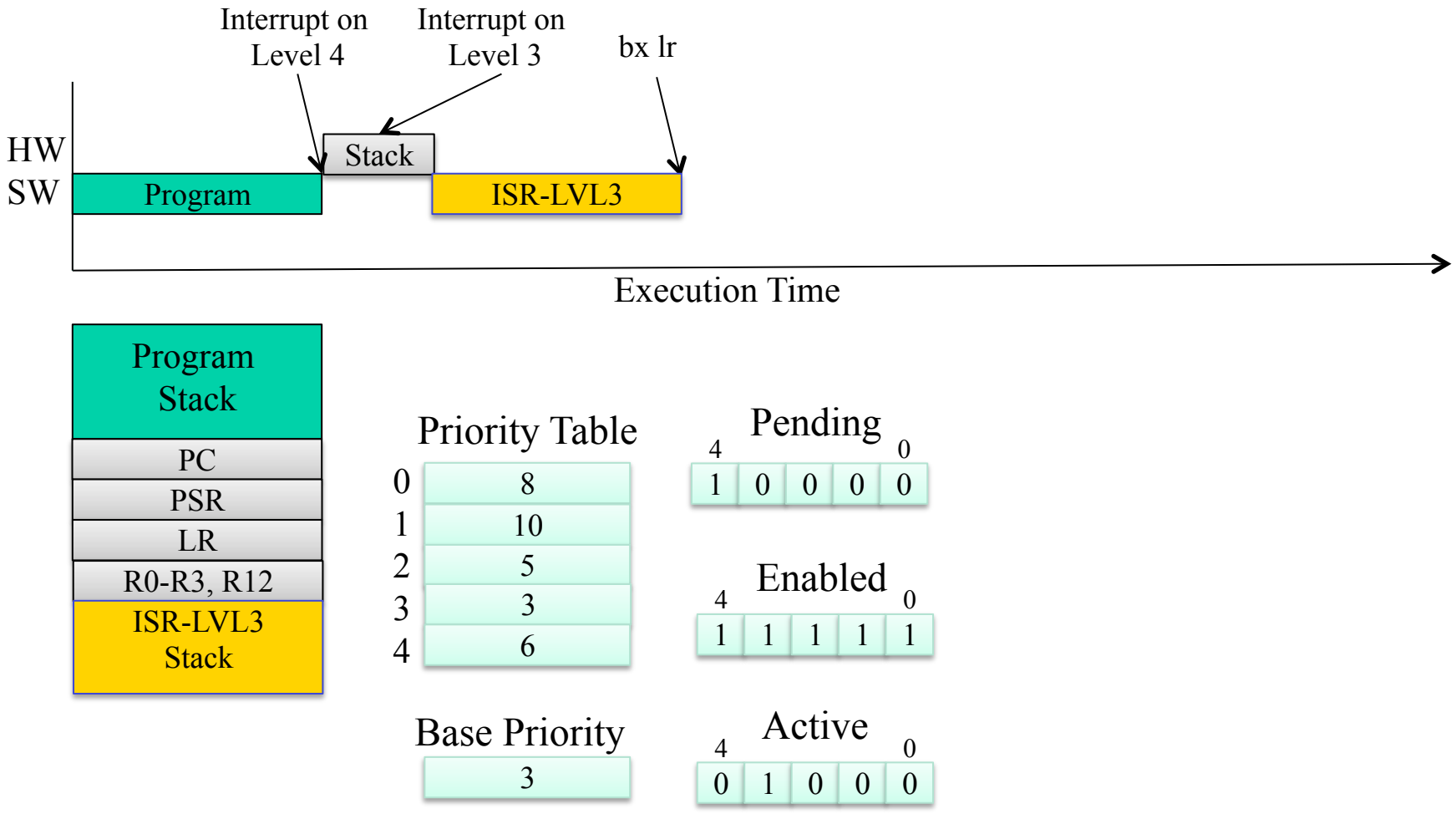
Base Priority

256

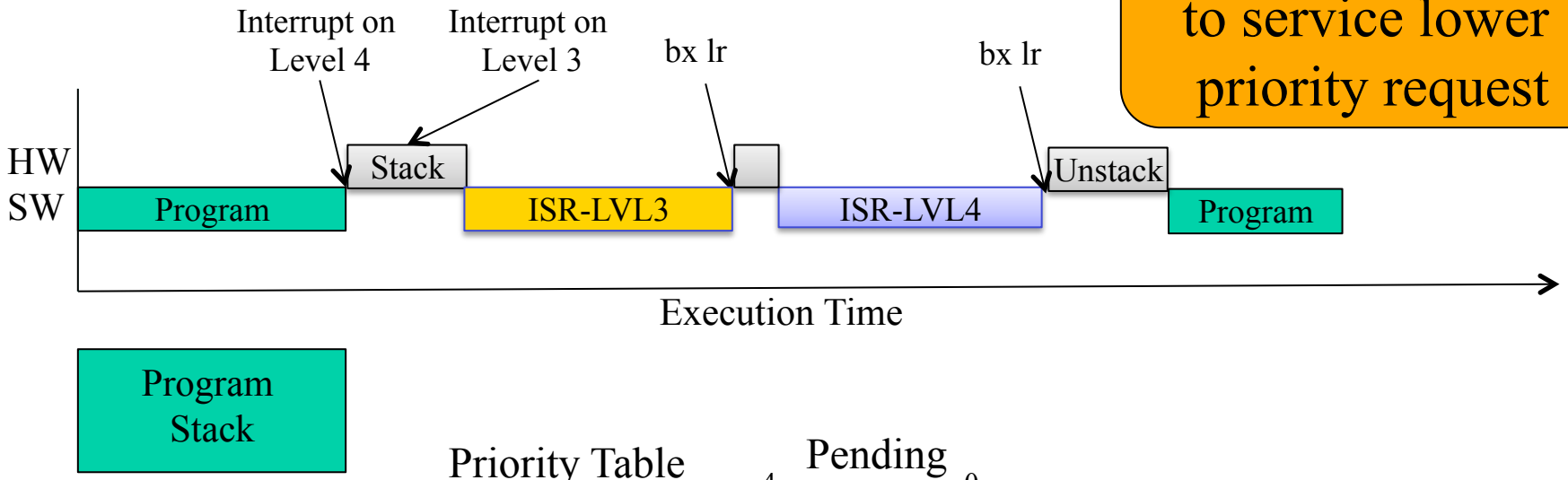
Active

4				0
0	0	0	0	0

Late Arrival



Late Arrival



Tail Chaining used to service lower priority request

Program Stack

Priority Table

0	8
1	10
2	5
3	3
4	6

Pending

4				0
0	0	0	0	0

Enabled

4				0
1	1	1	1	1

Base Priority

256

Active

4				0
0	0	0	0	0

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)

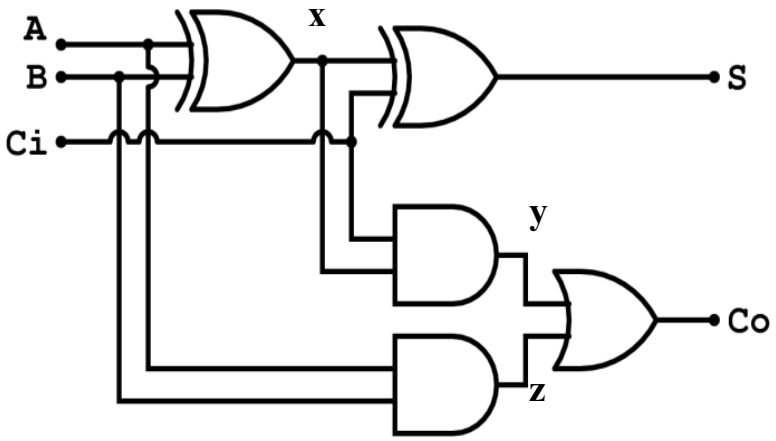
- What data structure?
  - Data needs be sorted
    - Inserting one thing at a time
  - We always pop from one end
  - But we add in sorted order.

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

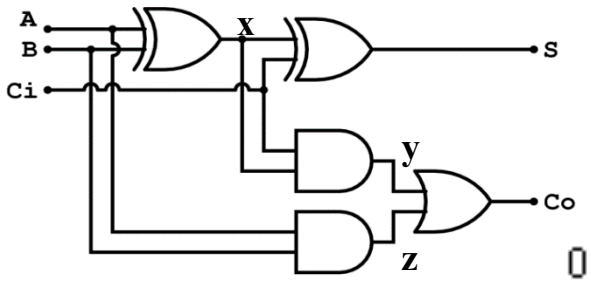
Timing delays and propagation



Full adder (from Wikipedia)

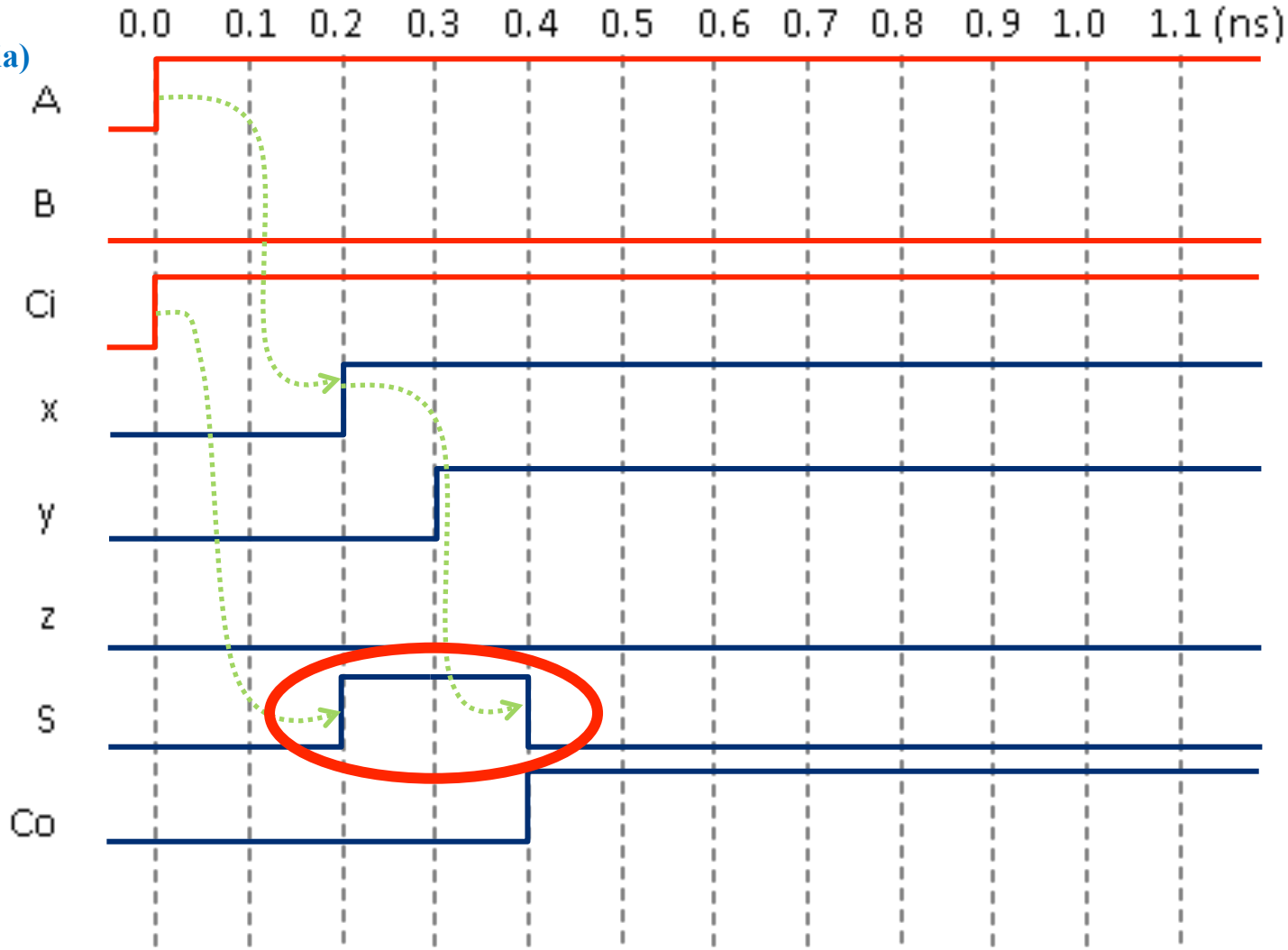
- Assume
  - XOR delay = 0.2ns
  - AND delay = 0.1ns
  - OR delay = 0.1 ns
- What is the worst case propagation delay for this circuit?

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.



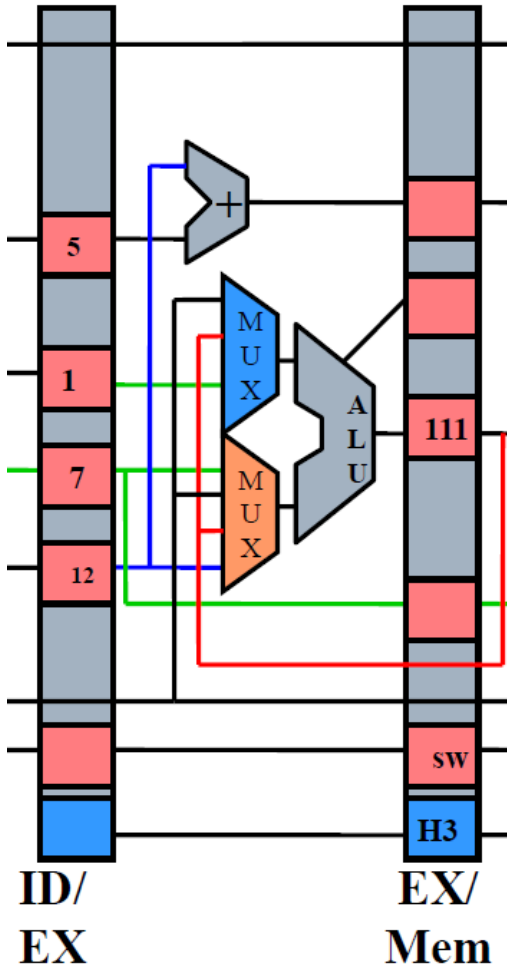
Only selected causality arrows shown…

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" even if the output ends up where it started!

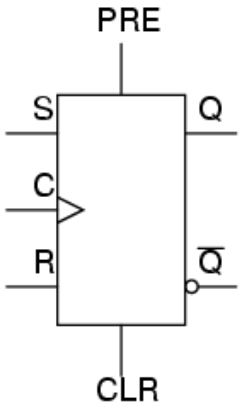
Effect of Glitches

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

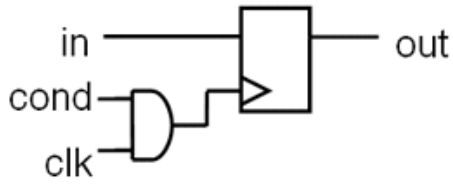


So, how can glitches hurt us?

- 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.

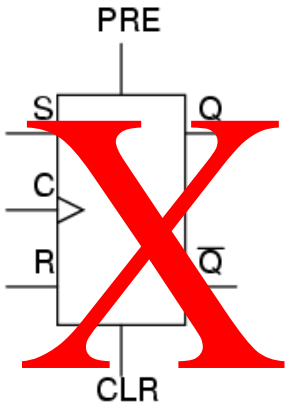


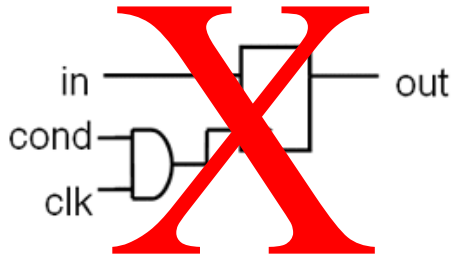
If clk is high and cond glitches, you get extra edges!

Design rules

1. Thou shalt not use asynchronous resets

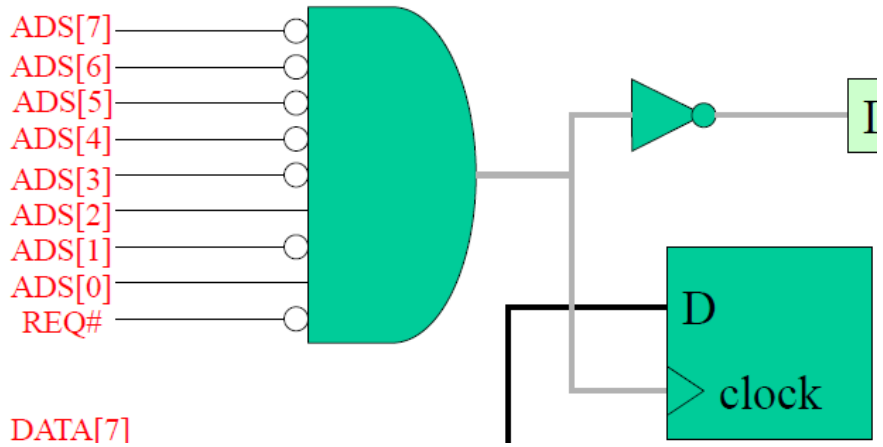
2. Thou shalt not drive a clock with anything other than a clock or directly off of a flip-flop's output



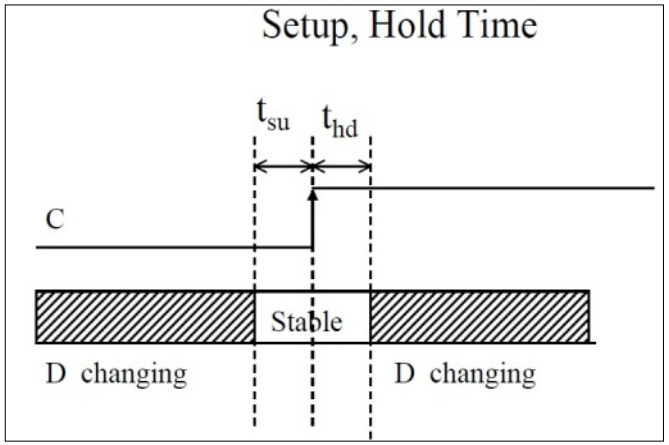


Really? Seriously?

- 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

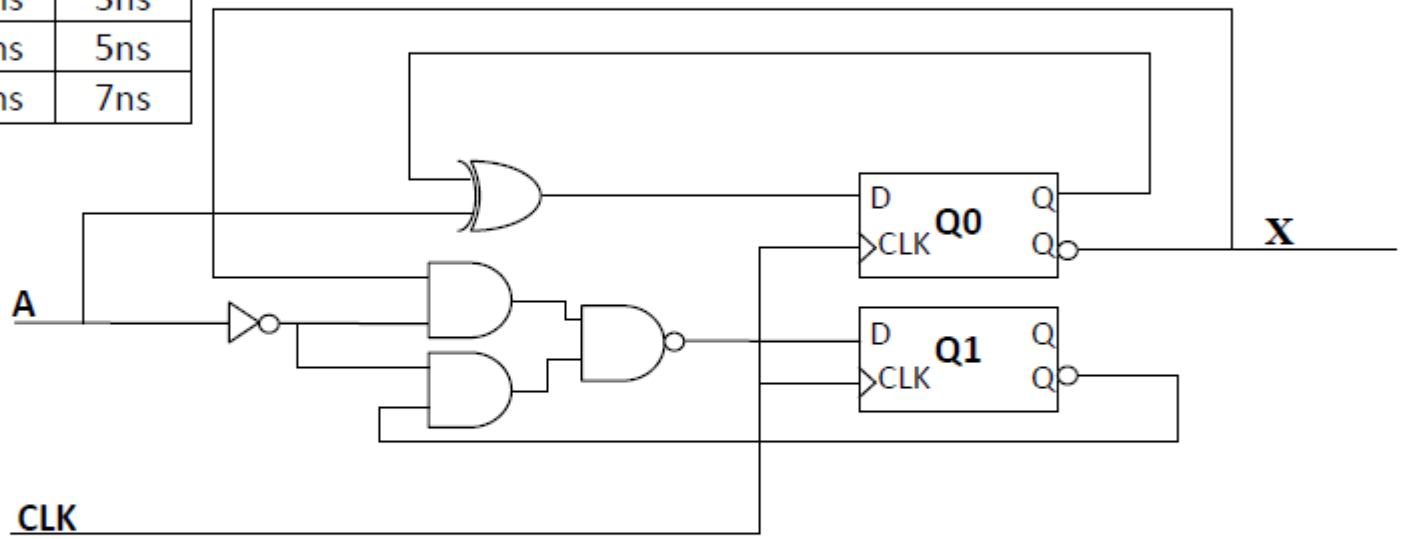


- The idea is simple.
  - When the clock is changing if the data is also changing it is hard to tell what the data ***is***.
    - Hardware can't always tell
      - And you can get meta-stable behavior too (very unlikely but…)
  - So we have a "guard band" around the clock rising time during which we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

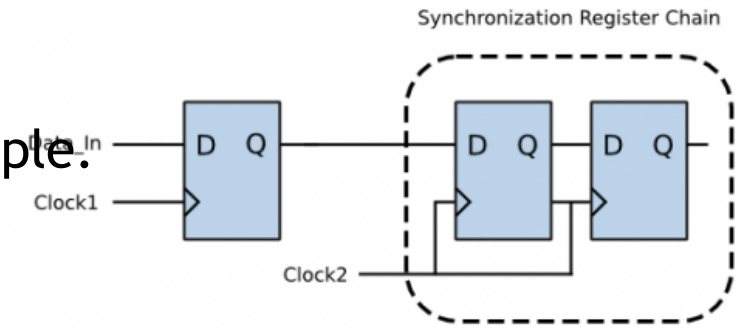
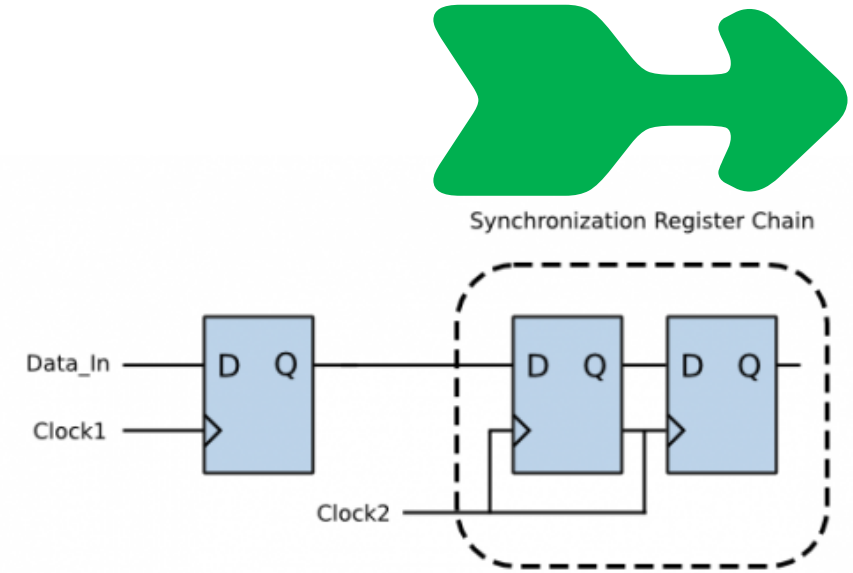


Figure from http://www.eeweb.com/electronics-quiz/solving-metastability-design-issues, we use the same thing to deal with external inputs too!

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?