
# EECS 373
## Design of Microprocessor-Based Systems

Website: **www.eecs.umich.edu/courses/eecs373/**

Ronald Dreslinski
University of Michigan

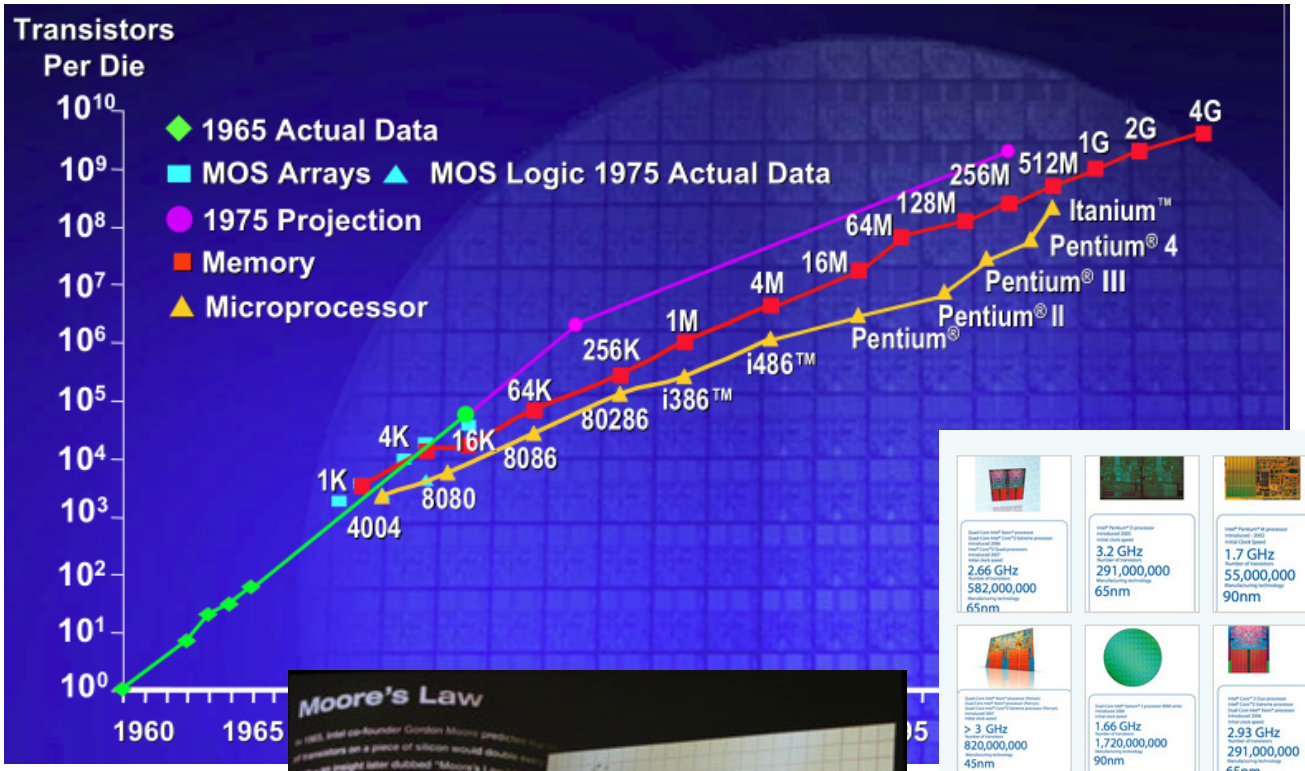
Midterm Review

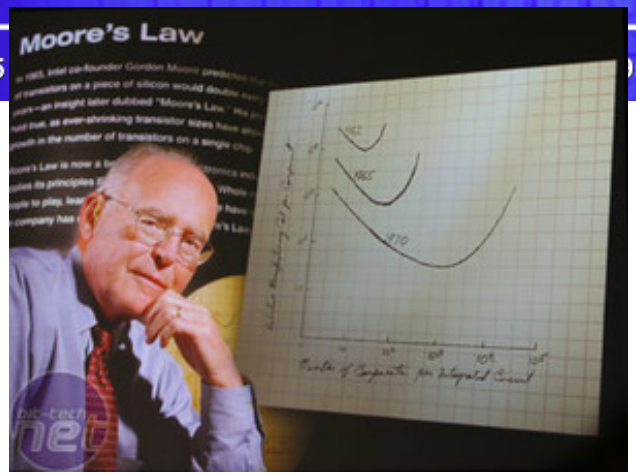
Slides developed in part by
Prof. Dutta and Dr. Brehob

# What is an embedded system?


What is driving the
embedded everywhere explosion?

# Moore's Law (a statement about economics): IC transistor count doubles every 18-24 mo



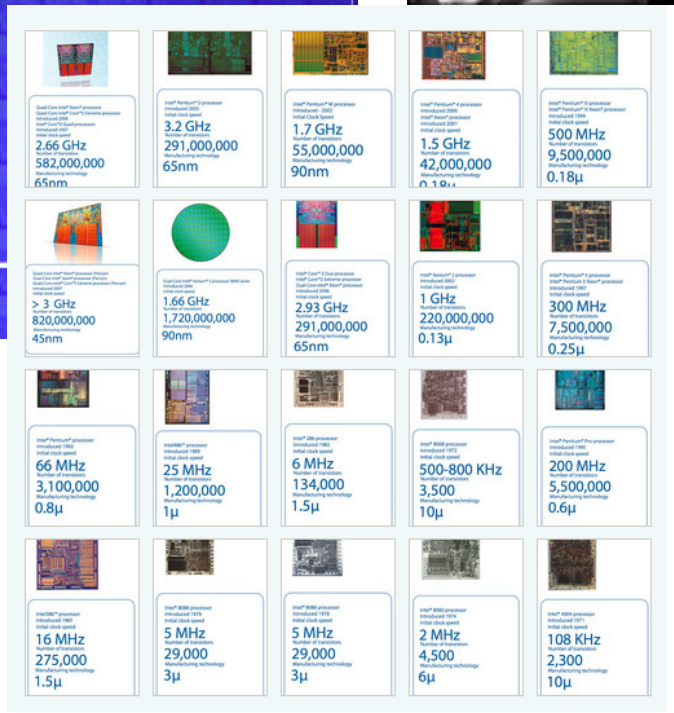
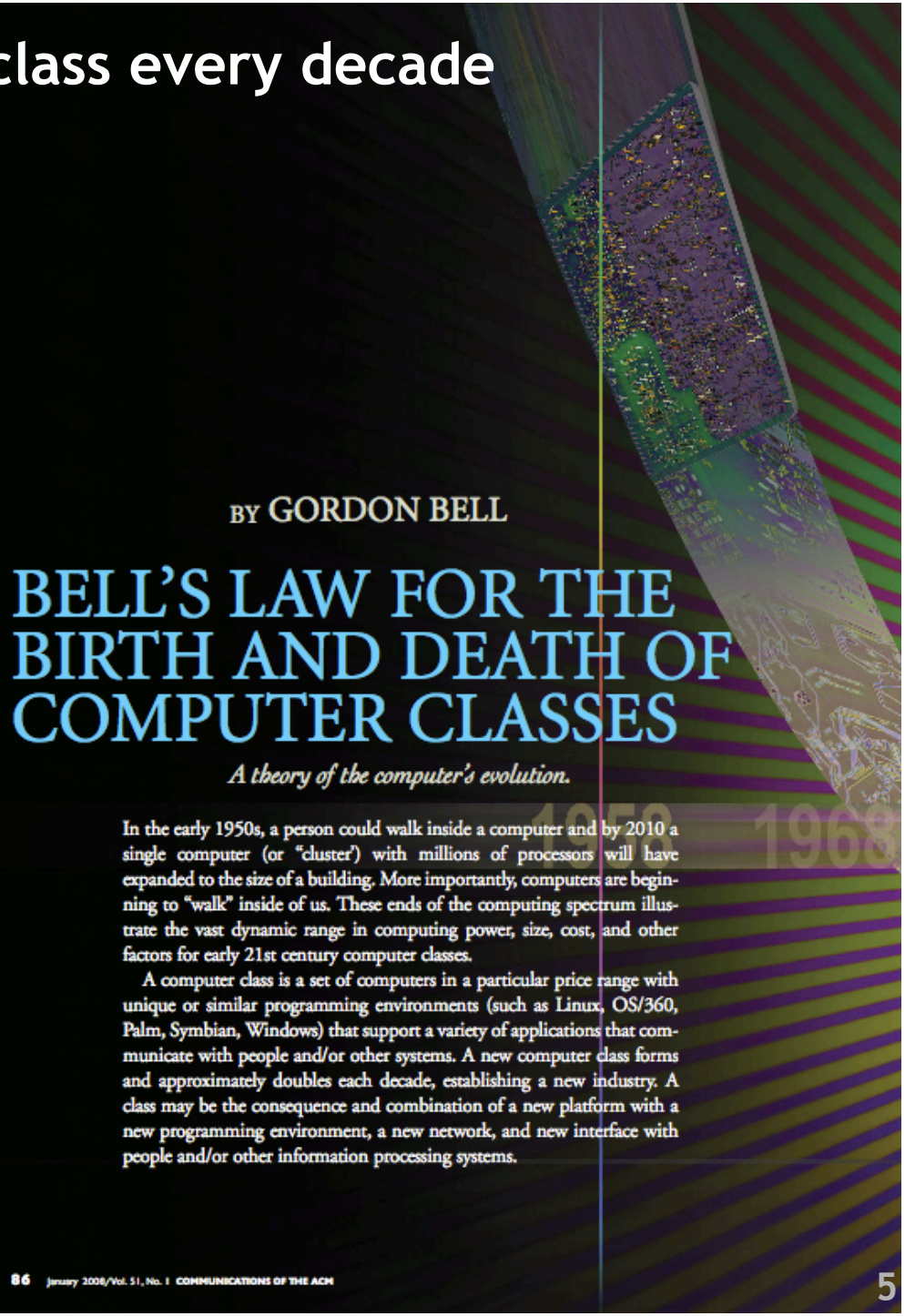




Photo Credit: Intel

# Bell's Law: A new computer class every decade

*"Roughly every decade a new, lower priced computer class forms based on a new programming platform, network, and interface resulting in new usage and the establishment of a new industry."*

- Gordon Bell [1972,2008]



BY GORDON BELL

BELL'S LAW FOR THE BIRTH AND DEATH OF COMPUTER CLASSES

*A theory of the computer's evolution.*

In the early 1950s, a person could walk inside a computer and by 2010 a single computer (or "cluster") with millions of processors will have expanded to the size of a building. More importantly, computers are beginning to "walk" inside of us. These ends of the computing spectrum illustrate the vast dynamic range in computing power, size, cost, and other factors for early 21st century computer classes.

A computer class is a set of computers in a particular price range with unique or similar programming environments (such as Linux, OS/360, Palm, Symbian, Windows) that support a variety of applications that communicate with people and/or other systems. A new computer class forms and approximately doubles each decade, establishing a new industry. A class may be the consequence and combination of a new platform with a new programming environment, a new network, and new interface with people and/or other information processing systems.

# What is driving Bell's Law?

## Technology Scaling

- Moore's Law
  - Made transistors cheap
- Dennard's Scaling
  - Made them fast
  - And low-power
- Result
  - Holding #T's constant
    - Exponentially lower cost
    - Exponentially lower power
  - Small, cheap & low-power
    - Microcontrollers
    - Memory
    - Radios

## Technology Innovations

- MEMS technology
  - Micro-fabricated sensors
- New memories
  - New cell structures (11T)
  - New tech (FeRAM, FinFET)
- Near-threshold computing
  - Minimize active power
  - Minimize static power
- New wireless systems
  - Radio architectures
  - Modulation schemes
- Energy harvesting

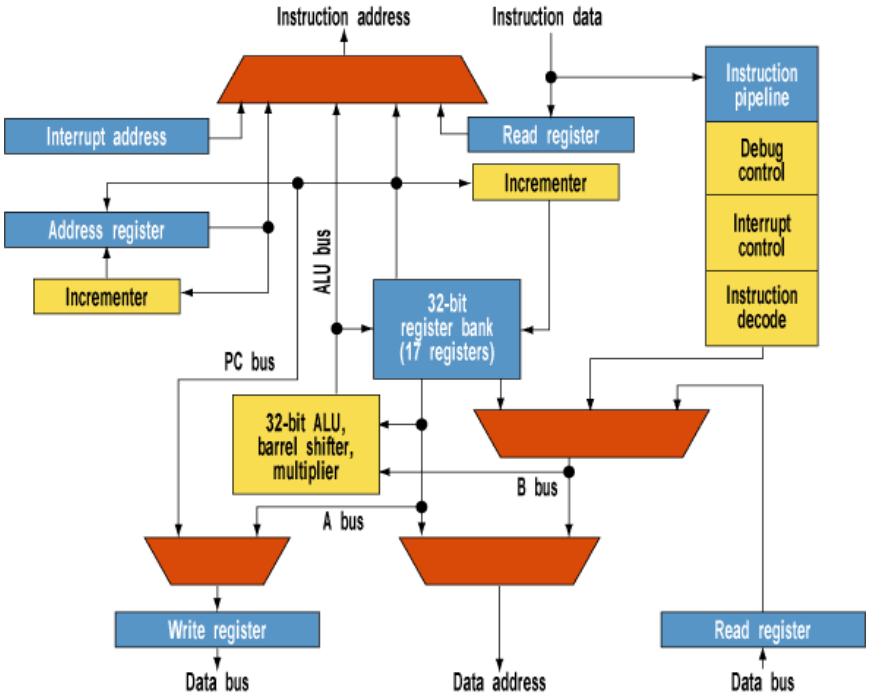

# Why study 32-bit MCUs and FPGAs?


What differentiates these products from one another?

FPGA
=====

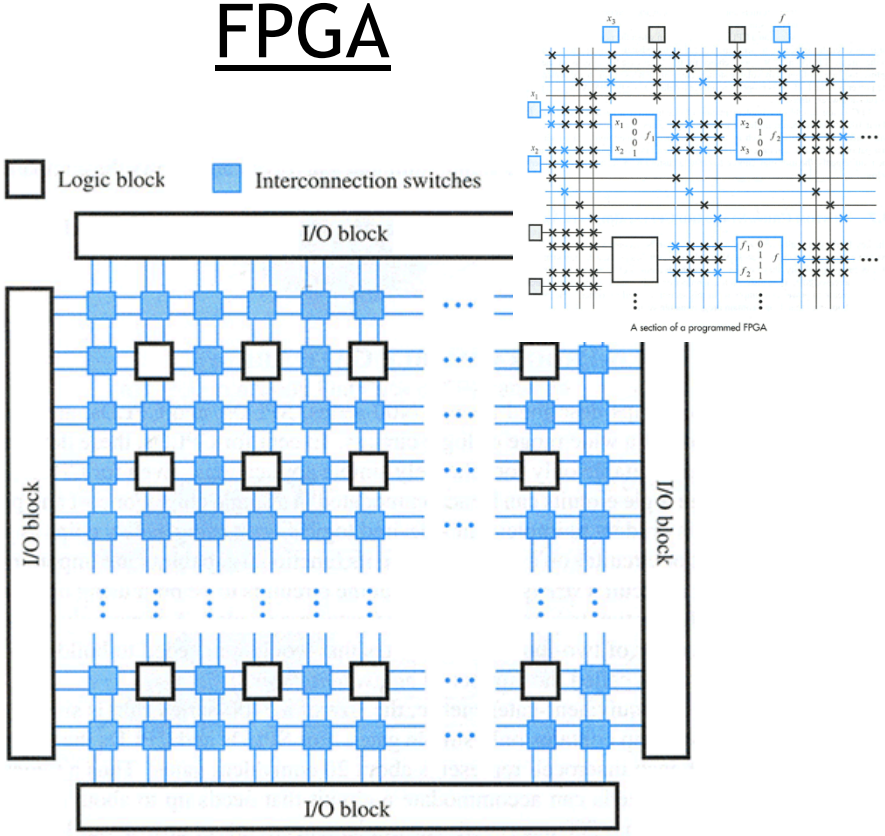
Microprocessor
==========

# MPU



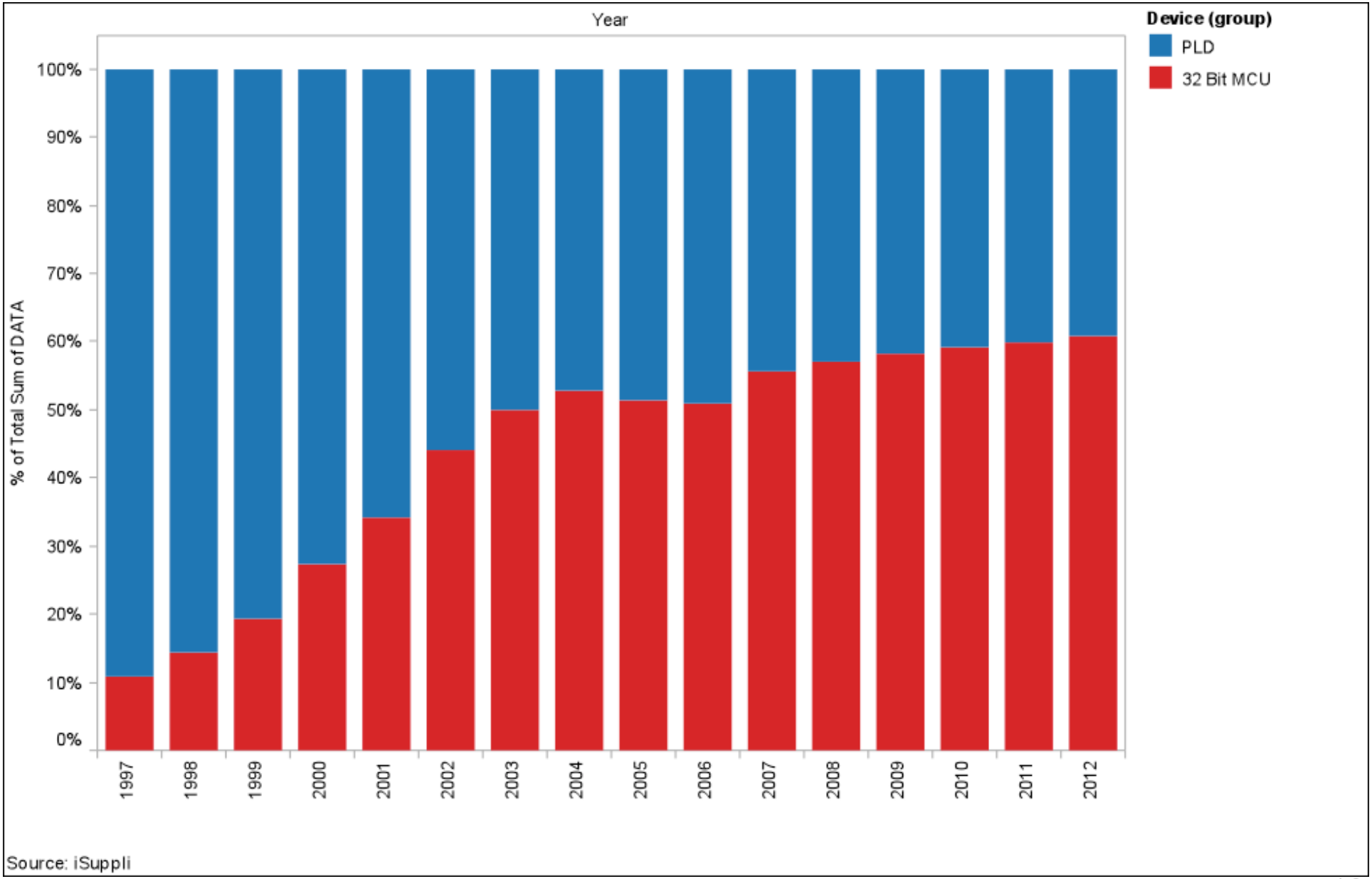

The Cortex M3's Thumbnail architecture looks like a conventional Arm processor. The differences are found in the Harvard architecture and the instruction decode that handles only Thumb and Thumb 2 instructions.

# FPGA



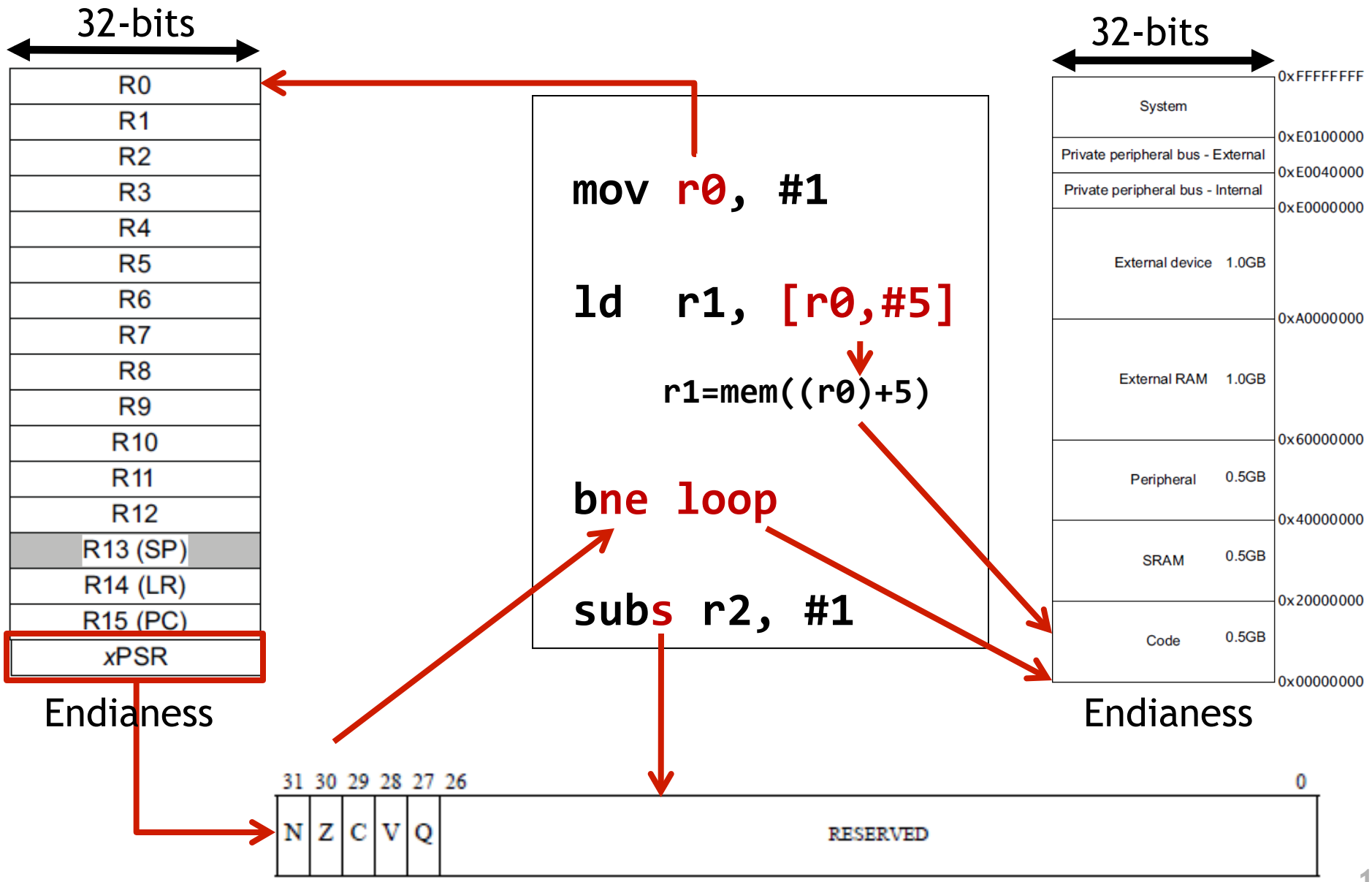

General structure of an FPGA

# MCU-32 and PLDs are tied in embedded market share



# Major elements of an Instruction Set Architecture

(registers, memory, word size, endianess, conditions, instructions, addressing modes)



# The endianess religious war: 284 years and counting!


- Modern version
  - Danny Cohen
  - IEEE Computer, v14, #10
  - Published in 1981
  - Satire on CS religious war

- Historical Inspiration
  - Jonathan Swift
  - *Gulliver's Travels*
  - Published in 1726
  - Satire on Henry-VIII's split with the Church
    - Now a major motion picture!

- Little-Endian
  - LSB is at lower address

```
                                Memory     Value
                                Offset   (LSB) (MSB)
                                ======   ===========
uint8_t a  = 1;                 0x0000   01 02 FF 00
uint8_t b  = 2;
uint16_t c = 255; // 0x00FF
uint32_t d = 0x12345678;        0x0004   78 56 34 12
```

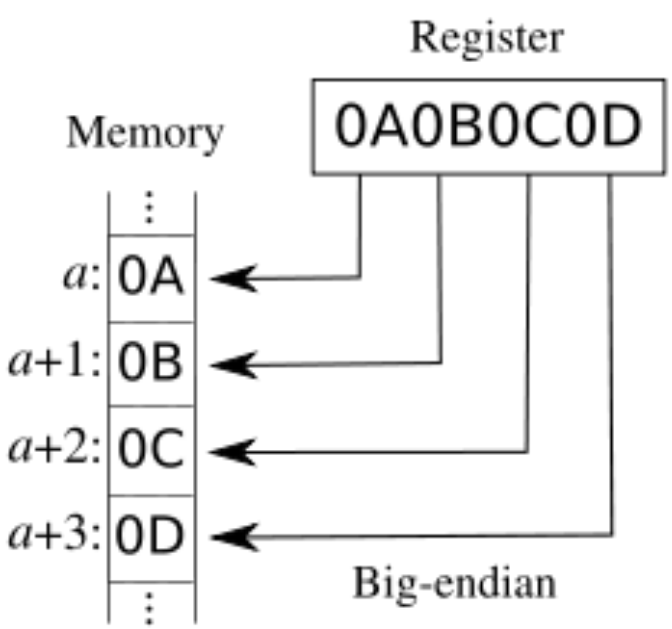
- Big-Endian
  - MSB is at lower address

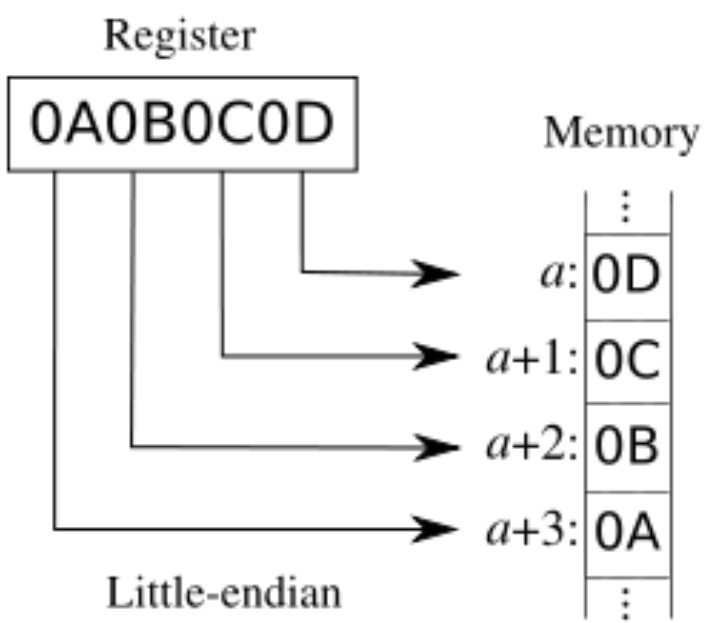
```
                                Memory     Value
                                Offset   (LSB) (MSB)
                                ======   ===========
uint8_t a  = 1;                 0x0000   01 02 00 FF
uint8_t b  = 2;
uint16_t c = 255; // 0x00FF
uint32_t d = 0x12345678;        0x0004   12 34 56 78
```

# Addressing: Big Endian vs Little Endian (370 slide)




- Endian-ness: ordering of bytes within a word
  - Little - increasing numeric significance with increasing memory addresses
  - Big – The opposite, most significant byte first
  - MIPS is big endian, x86 is little endian



# Instruction encoding

- Instructions are encoded in machine language opcodes
- Sometimes
  - Necessary to hand generate opcodes
  - Necessary to verify assembled code is correct
- How?

| Instructions |
| --- |
| `movs r0, #10` |
| `movs r1, #0` |

| Register Value | Memory Value |
| --- | --- |
| 001\|00\|000\|00001010 | (LSB) (MSB) |
| (msb) (lsb) | 0a 20 00 21 |
| 001\|00\|001\|00000000 | |

ARMv7 ARM

**Encoding T1** All versions of the Thumb ISA.

MOVS <Rd>,#<imm8> Outside IT block.

MOV<c> <Rd>,#<imm8> Inside IT block.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 0 | 0 | Rd | | | imm8 | | | | | | | |

```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

# Assembly example


```
data:
     .byte 0x12, 20, 0x20, -1
func:
          mov r0, #0
          mov r4, #0
          movw   r1, #:lower16:data
          movt   r1, #:upper16:data
top:      ldrb   r2, [r1],1
          add r4, r4, r2
          add r0, r0, #1
          cmp r0, #4
          bne top
```

# An ISA defines the hardware/software interface


- A “contract” between architects and programmers
- Register set
- Instruction set
  - Addressing modes
  - Word size
  - Data formats
  - Operating modes
  - Condition codes
- ***Calling conventions***
  - Really not part of the ISA (usually)
  - Rather part of the ABI
  - But the ISA often provides meaningful support.

# ARM Architecture roadmap


**ARM7TDMI**
**ARM922T**

Thumb instruction set


**ARM926EJ-S**
**ARM946E-S**
**ARM966E-S**

Improved ARM/Thumb Interworking

DSP instructions

**Extensions:**

Jazelle (5TEJ)


**ARM1136JF-S**
**ARM1176JZF-S**
**ARM11 MPCore**

SIMD Instructions

Unaligned data support

**Extensions:**

Thumb-2 (6T2)

TrustZone (6Z)

Multicore (6K)


**Cortex-A8/R4/M3/M1**

Thumb-2

**Extensions:**

v7A (applications) – NEON

v7R (real time) – HW Divide

V7M (microcontroller) – HW Divide and Thumb-2 only

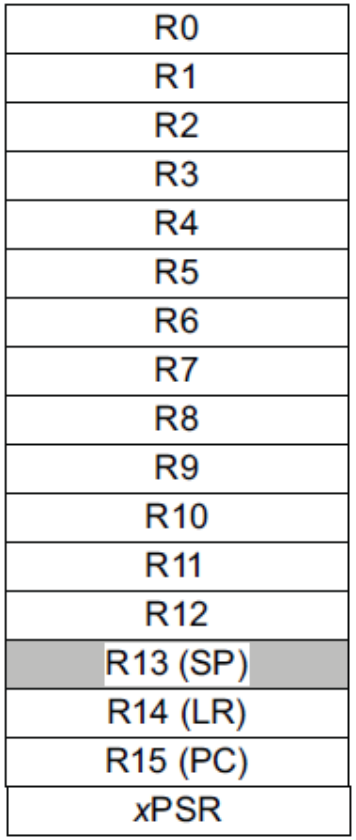
# ARM Cortex-M3 ISA

## Instruction Set

```
ADD Rd, Rn, <op2>
```

Branching
Data processing
Load/Store
Exceptions
Miscellaneous

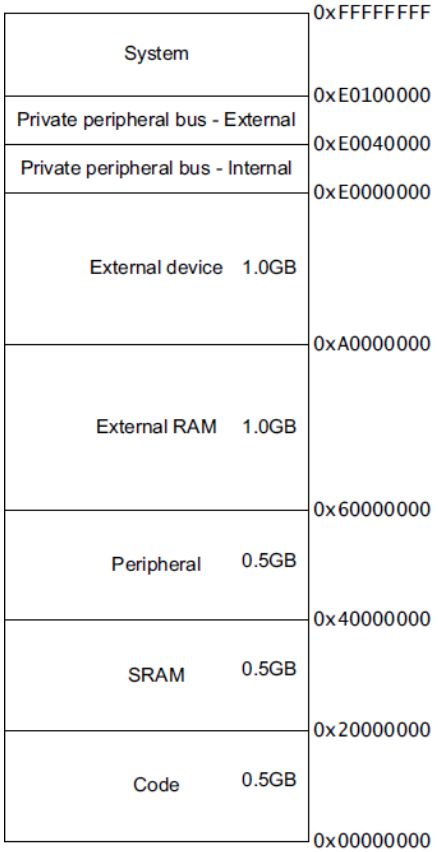
## Register Set




32-bits

Endianess

## Address Space



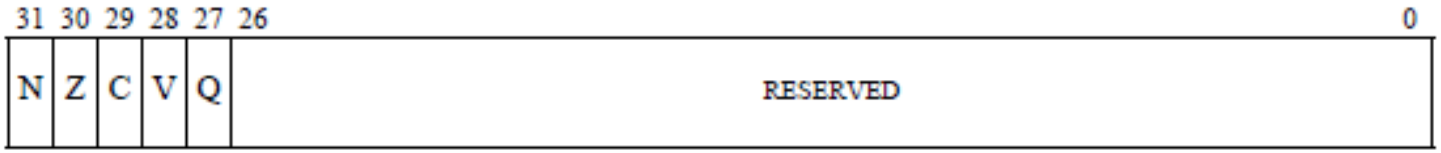

32-bits

Endianess

# Addressing Modes (again)


- Offset Addressing
  - Offset is added or subtracted from base register
  - Result used as effective address for memory access
  - [<Rn>, <offset>]
- Pre-indexed Addressing
  - Offset is applied to base register
  - Result used as effective address for memory access
  - Result written back into base register
  - [<Rn>, <offset>]!
- Post-indexed Addressing
  - The address from the base register is used as the EA
  - The offset is applied to the base and then written back
  - [<Rn>], <offset>

# Application Program Status Register (APSR)




APSR bit fields are in the following two categories:

- Reserved bits are allocated to system features or are available for future expansion. Further information on currently allocated reserved bits is available in *The special-purpose program status registers (xPSR)* on page B1-8. Application level software must ignore values read from reserved bits, and preserve their value on a write. The bits are defined as UNK/SBZP.
- Flags that can be set by many instructions:

  **N, bit [31]** Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then N == 1 if the result is negative and N = 0 if it is positive or zero.

  **Z, bit [30]** Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

  **C, bit [29]** Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.

  **V, bit [28]** Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

  **Q, bit [27]** Set to 1 if an SSAT or USAT instruction changes (saturates) the input value for the signed or unsigned range of the result.

| Register | Synonym | Special | Role in the procedure call standard |
|---|---|---|---|
| r15 | | PC | The Program Counter. |
| r14 | | LR | The Link Register. |
| r13 | | SP | The Stack Pointer. |
| r12 | | IP | The Intra-Procedure-call scratch register. |
| r11 | v8 | | Variable-register 8. |
| r10 | v7 | | Variable-register 7. |
| r9 | | v6<br>SB<br>TR | Platform register.<br>The meaning of this register is defined by the platform standard. |
| r8 | v5 | | Variable-register 5. |
| r7 | v4 | | Variable register 4. |
| r6 | v3 | | Variable register 3. |
| r5 | v2 | | Variable register 2. |
| r4 | v1 | | Variable register 1. |
| r3 | a4 | | Argument / scratch register 4. |
| r2 | a3 | | Argument / scratch register 3. |
| r1 | a2 | | Argument / result / scratch register 2. |
| r0 | a1 | | Argument / result / scratch register 1. |

# ABI quote


- A subroutine must preserve the contents of the registers r4-r8, r10, r11 and SP (and r9 in PCS variants that designate r9 as v6).

# ABI Basic Rules


1. A subroutine must preserve the contents of the registers r4-11 and SP
   - Let's be careful with r9 though.

2. Arguments are passed though r0 to r3
   - If we need more, we put a pointer into memory in one of the registers.
     - We'll worry about that later.

3. Return value is placed in r0
   - r0 and r1 if 64-bits.

4. Allocate space on stack as needed. Use it as needed.
   - Put it back when done…
   - Keep word aligned.

# Other useful factoids


- Stack grows down.
  - And pointed to by “sp”
- Address we need to go back to in “lr”

## And useful things for the example

- Assembly instructions
  - add adds two values
  - mul multiplies two values
  - bx branch to register

# Memory-mapped I/O


- The idea is really simple
  - Instead of real memory at a given memory address, have an I/O device respond.
    - Huh?
- Example:
  - Let's say we want to have an LED turn on if we write a "1" to memory location 5.
  - Further, let's have a button we can read (pushed or unpushed) by reading address 4.
    - If pushed, it returns a 1.
    - If not pushed, it returns a 0.

# Now…


- How do you get that to happen?
  - We could just say “magic” but that’s not very helpful.
  - Let’s start by detailing a simple bus and hooking hardware up to it.
- We’ll work on a real bus next time!

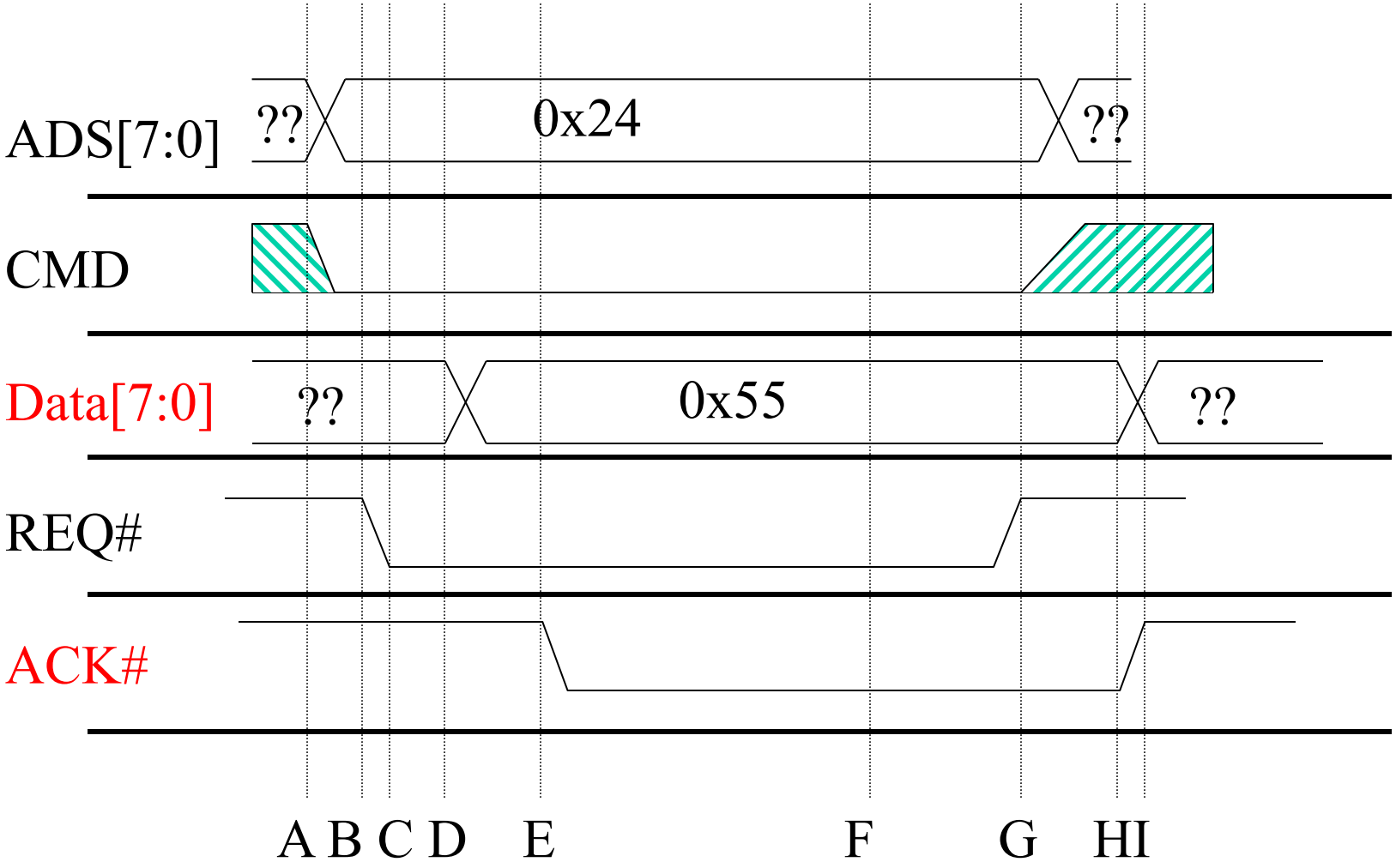
# Basic example


- Discuss a basic bus protocol
  - Asynchronous (no clock)
  - Initiator and Target
  - REQ#, ACK#, Data[7:0], ADS[7:0], CMD
    - CMD=0 is read, CMD=1 is write.
    - REQ# low means initiator is requesting something.
    - ACK# low means target has done its job.

# A read transaction


- Say initiator wants to read location 0x24
  - Initiator sets ADS=0x24, CMD=0.
  - Initiator *then* sets REQ# to low. (why do we need a delay? How much of a delay?)
  - Target sees read request.
  - Target drives data onto data bus.
  - Target *then* sets ACK# to low.
  - Initiator grabs the data from the data bus.
  - Initiator sets REQ# to high, stops driving ADS and CMD
  - Target stops driving data, sets ACK# to high terminating the transaction

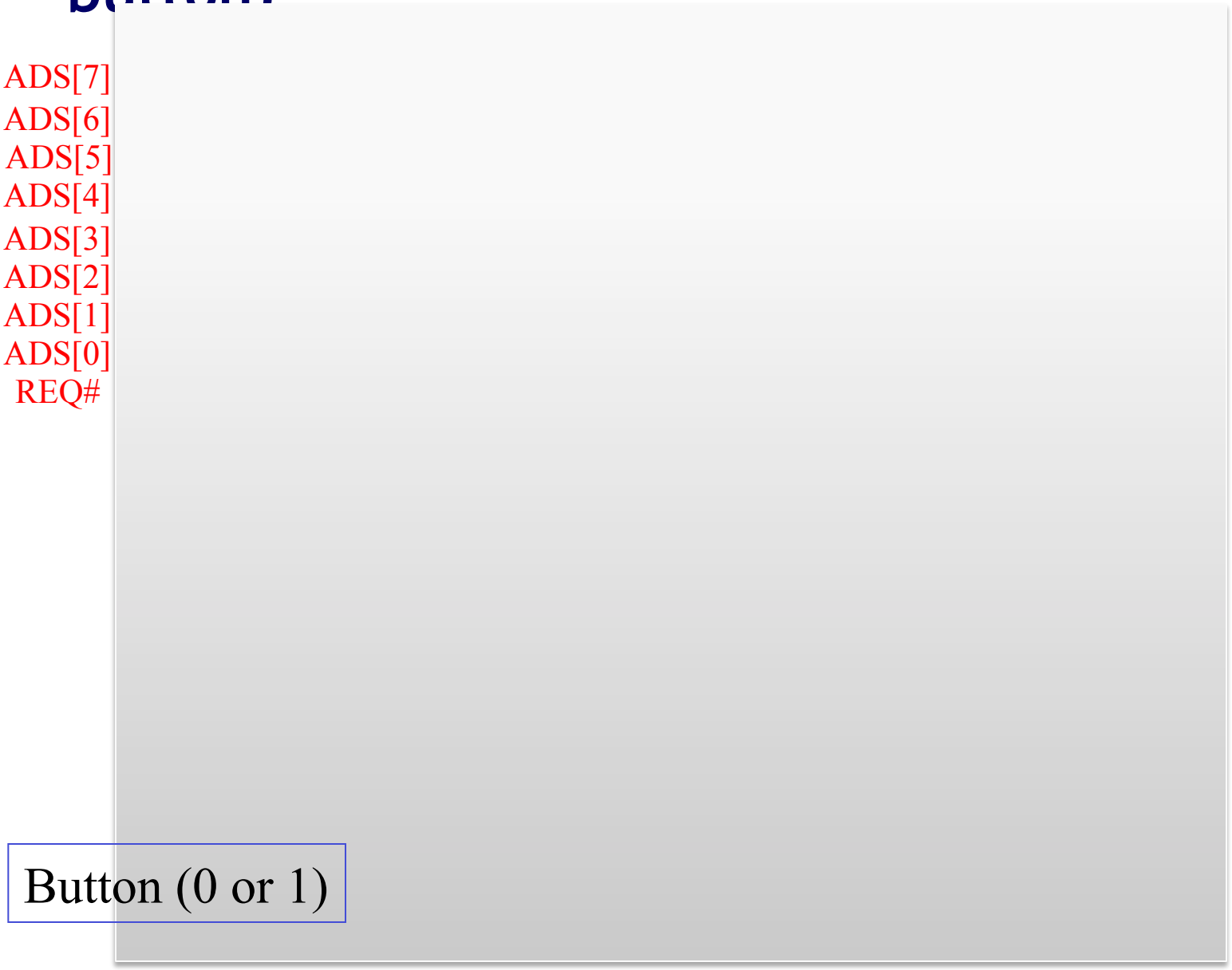
# Read transaction



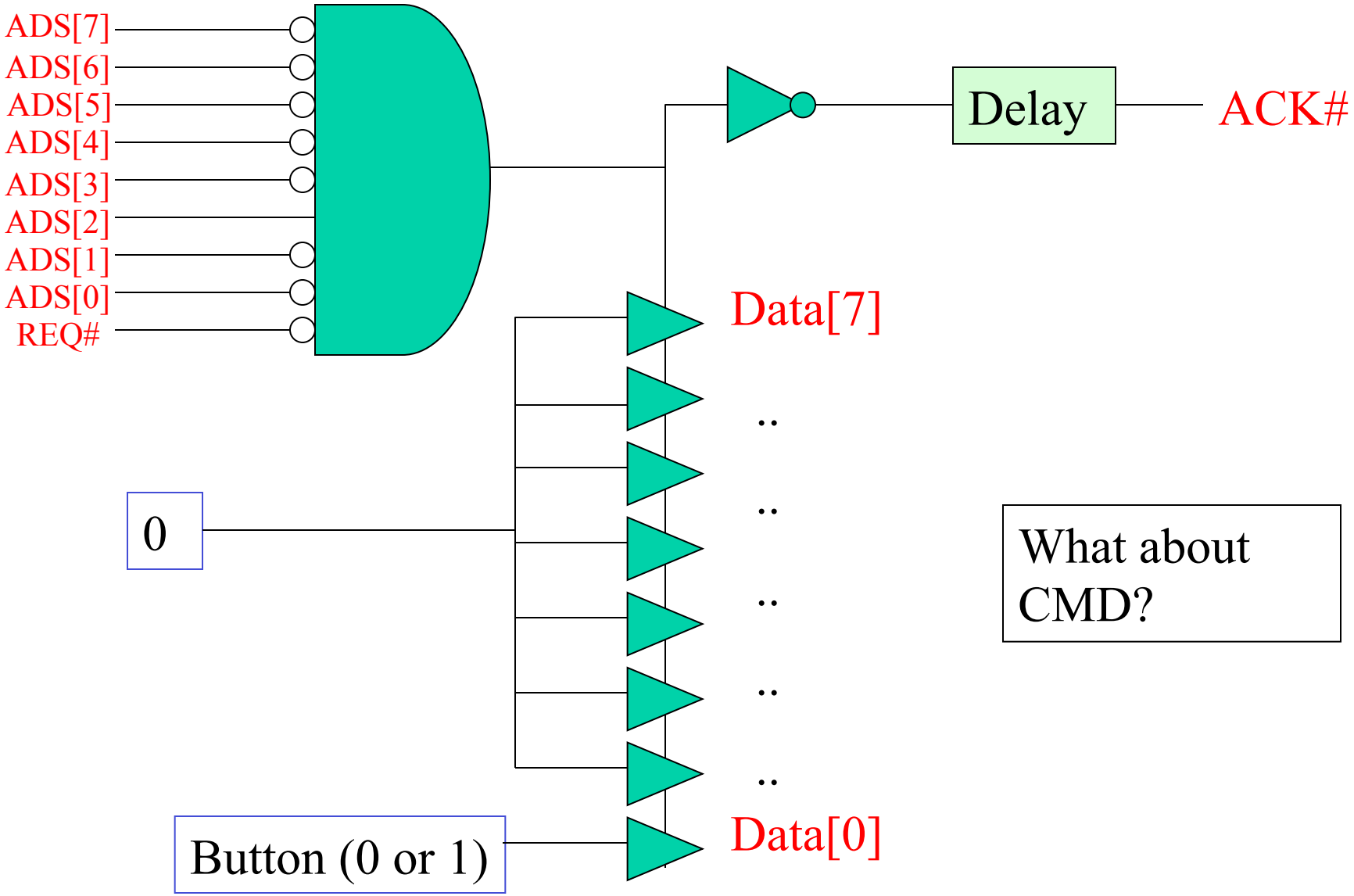
# A write transaction
# (write 0xF4 to location 0x31)


- Initiator sets ADS=0x31, CMD=1, Data=0xF4
- Initiator *then* sets REQ# to low.
- Target sees write request.
- Target reads data from data bus. (Just has to store in a register, need not write all the way to memory!)
- Target *then* sets ACK# to low.
- Initiator sets REQ# to high & stops driving other lines.
- Target sets ACK# to high terminating the transaction

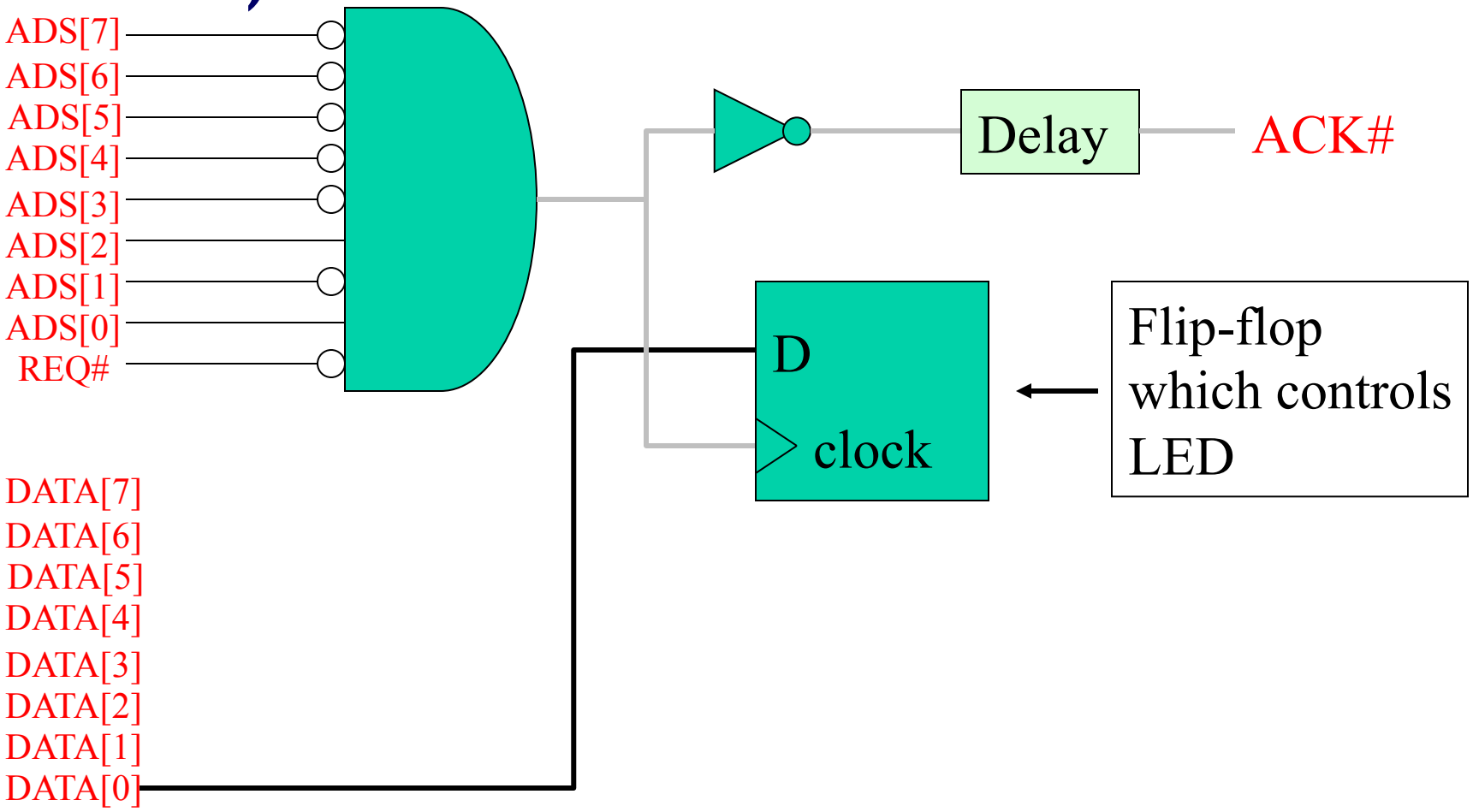
# The push-button (if ADS=0x04 write 0 or 1 depending on button)



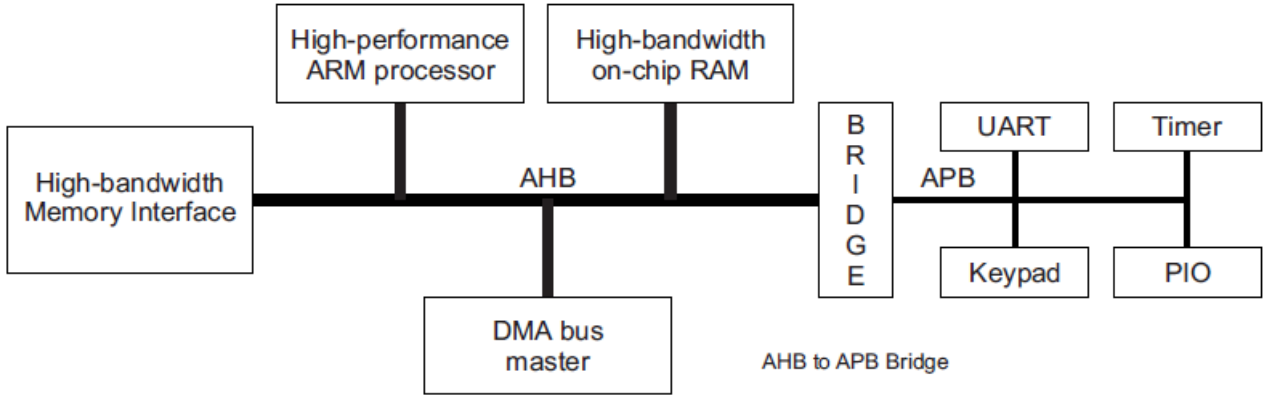
# The push-button (if ADS=0x04 write 0 or 1 depending on button)



# The LED
# (1 bit reg written by LSB of address 0x05)



# Advanced Microcontroller Bus Architecture (AMBA)
# - Advanced High-performance Bus (AHB)
# - Advanced Peripheral Bus (APB)



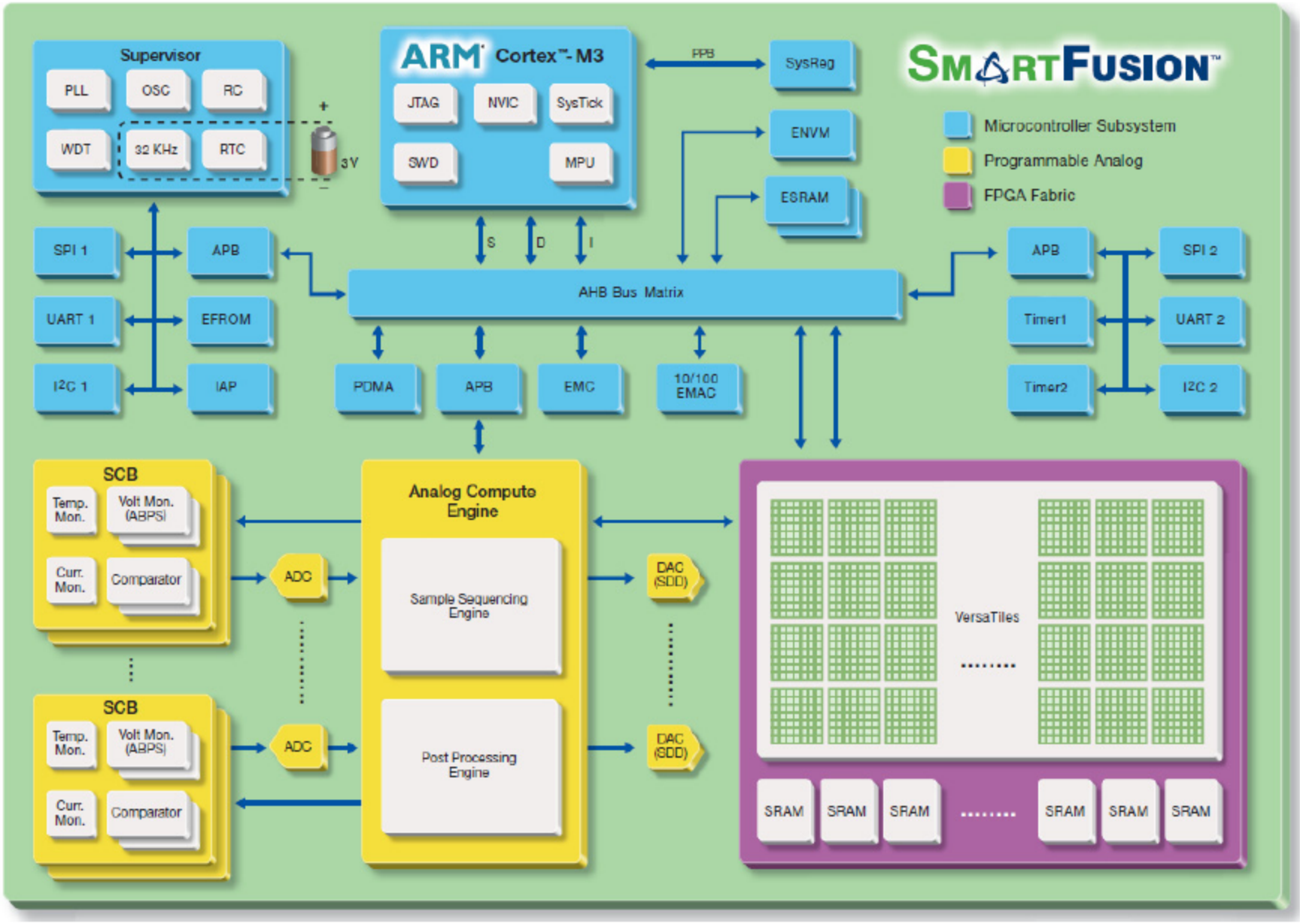

## AHB

- High performance
- Pipelined operation
- Burst transfers
- Multiple bus masters
- Split transactions

## APB

- Low power
- Latched address/control
- Simple interface
- Suitable of many peripherals

# Actel SmartFusion system/bus architecture



# Bus terminology


- Any given transaction have an "*initiator*" and "*target*"
- Any device capable of being an initiator is said to be a "*bus master*"
  - In many cases there is only one bus master (*single master* vs. *multi-master*).
- A device that can only be a target is said to be a slave device.
- Some wires might be shared among all devices while others might be point-to-point connections (generally connecting the master to each target).

# Driving shared wires


- It is commonly the case that some shared wires might have more than one potential device that needs to drive them.
  - For example there might be a shared data bus that is used by the targets and the initiator. We saw this in the simple bus.
  - In that case, we need a way to allow one device to control the wires while the others "stay out of the way"
    - Most common solutions are:
      - using tri-state drivers (so only one device is driving the bus at a time)
      - using open-collector connections (so if any device drives a 0 there is a 0 on the bus otherwise there is a 1)

# Or just say no to shared wires.

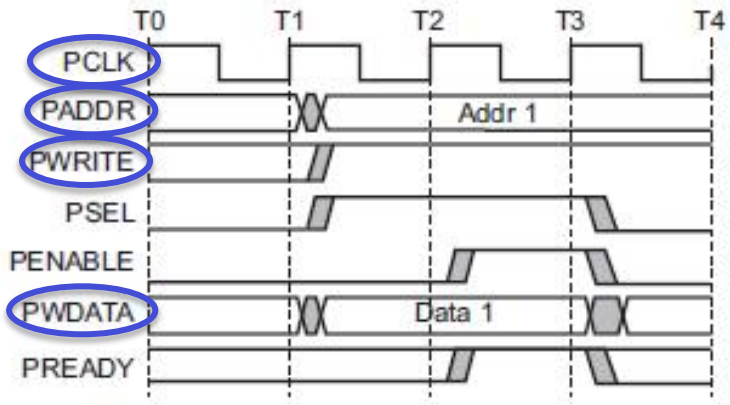

- Another option is to not share wires that could be driven by more than one device...
  - This can be really expensive.
    - Each target device would need its own data bus.
    - That's a LOT of wires!
  - Not doable when connecting chips on a PCB as you are paying for each pin.
  - Quite doable (though not pretty) inside of a chip.

## APB is a fairly simple bus designed to be easy to work with.


- Low-cost
- Low-power
- Low-complexity
- Low-bandwidth
- Non-pipelined
- Ideal for peripherals

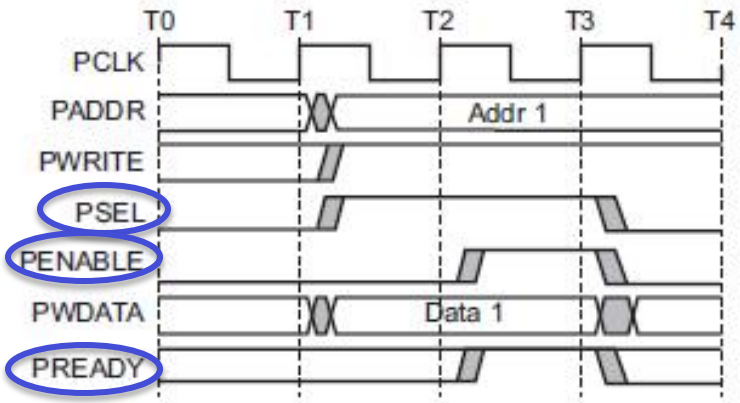
# APB bus signals

- PCLK
  - Clock
- PADDR
  - Address on bus
- PWRITE
  - 1=Write, 0=Read
- PWDATA
  - Data written to the I/O device. Supplied by the bus master/ processor.

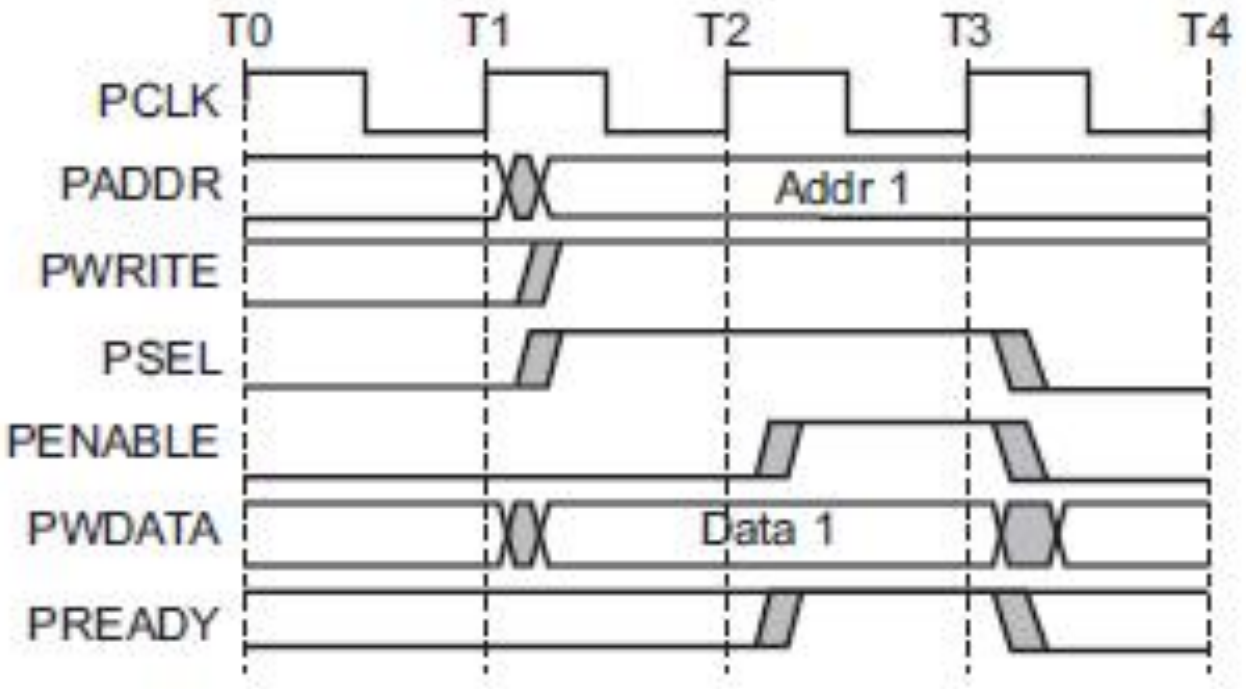


# APB bus signals

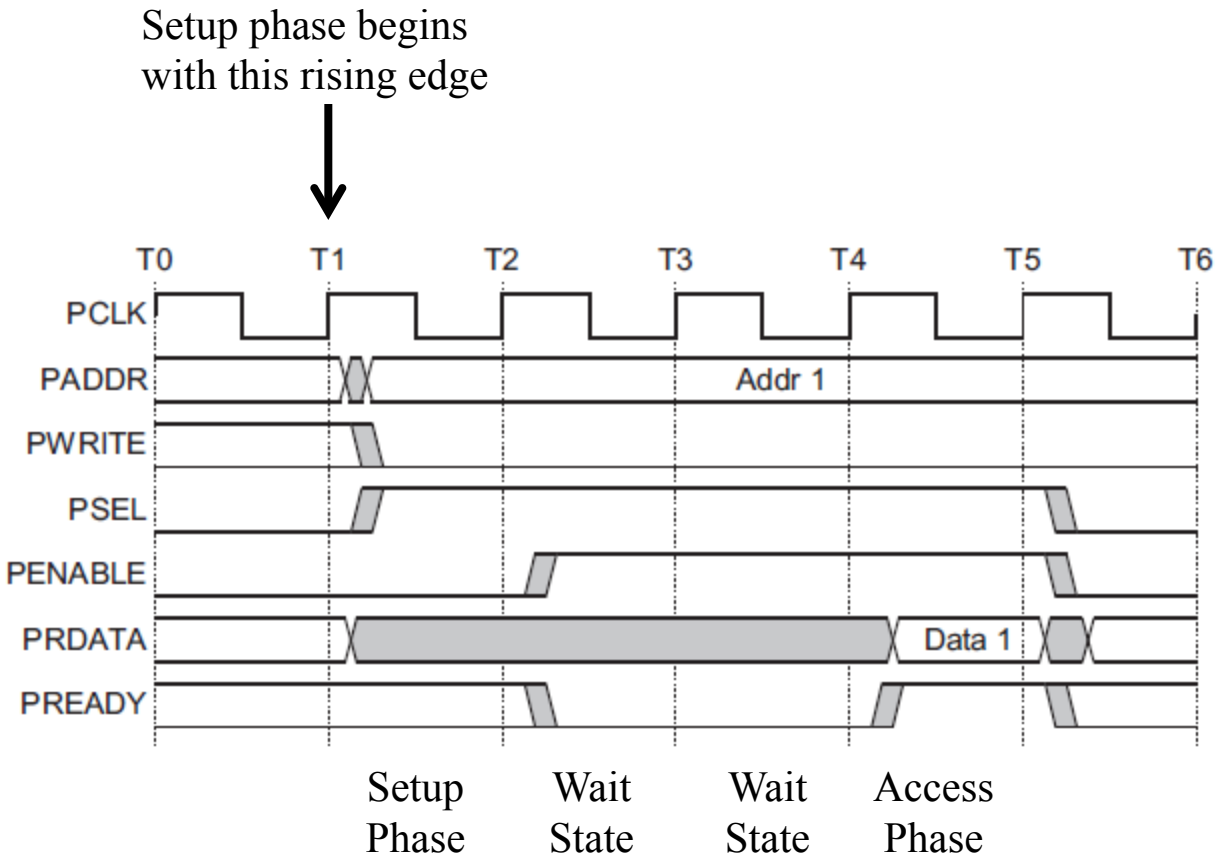
- PSEL
  - Asserted if the current bus transaction is targeted to ***this*** device
- PENABLE
  - High during entire transaction *other than* the first cycle.
- PREADY
  - Driven by target. Similar to our #ACK. Indicates if the target is *ready* to do transaction.
    Each target has it's own PREADY



# So what's happening here?



# A read transfer with wait states



# Interrupts


Interrupt (a.k.a. exception or trap):

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

Interrupts are really glorified procedure calls, except that they:

- **can occur between any two instructions**
- are transparent to the running program (usually)
- are not explicitly requested by the program (typically)
- call a procedure at an address determined by the type of interrupt, not the program

# Two basic types of interrupts (1/2)


- Those caused by an instruction
  - Examples:
    - TLB miss
    - Illegal/unimplemented instruction
    - div by 0
  - Names:
    - Trap, exception

## Two basic types of interrupts (2/2)


- Those caused by the external world
  - External device
  - Reset button
  - Timer expires
  - Power failure
  - System error
- Names:
  - interrupt, external interrupt

# How it works


- Something tells the processor core there is an interrupt
- Core transfers control to code that needs to be executed
- Said code “returns” to old program
- Much harder then it looks.
  - Why?

## … is in the details


- How do you figure out *where* to branch to?
- How do you ensure that you can get back to where you started?
- Don't we have a pipeline? What about partially executed instructions?
- What if we get an interrupt while we are processing our interrupt?
- What if we are in a "critical section?"

## Where


- If you know *what* caused the interrupt then you want to jump to the code that handles that interrupt.
  - If you number the possible interrupt cases, and an interrupt comes in, you can just branch to a location, using that number as an offset (this is a branch table)
  - If you don't have the number, you need to *poll* all possible sources of the interrupt to see who caused it.
    - Then you branch to the right code

# Enabling and disabling interrupt sources

- Interrupt Set Enable and Clear Enable
  - 0xE000E100-0xE000E11C, 0xE000E180-0xE000E19C

| | | | | |
|---|---|---|---|---|
| 0xE000E100 | SETENA0 | R/W | 0 | Enable for external interrupt #0–31<br>bit[0] for interrupt #0 (exception #16)<br>bit[1] for interrupt #1 (exception #17)<br>...<br>bit[31] for interrupt #31 (exception #47)<br>Write 1 to set bit to 1; write 0 has no effect<br>Read value indicates the current status |
| 0xE000E180 | CLRENA0 | R/W | 0 | Clear enable for external interrupt #0–31<br>bit[0] for interrupt #0<br>bit[1] for interrupt #1<br>...<br>bit[31] for interrupt #31<br>Write 1 to clear bit to 0; write 0 has no effect<br>Read value indicates the current enable status |

# Polling-Driven Application


- Recall pushbutton-LED example

```
        mov     r0, #0x4        % PBS MMIO address
        mov     r1, #0x5        % LED MMIO address
loop:   ldr     r2, [r0, #0]    % Read value from switch [1 cycle]
        str     r2 [r1, #0]     % Save value to LED [1 cycle]
        b       loop            % Repeat these steps [1 cycle]
```

- This is a polling-driven application
- Software constantly loops, polling and (re)acting
- However, it doesn't do anything else useful!

# The Problem with Polling

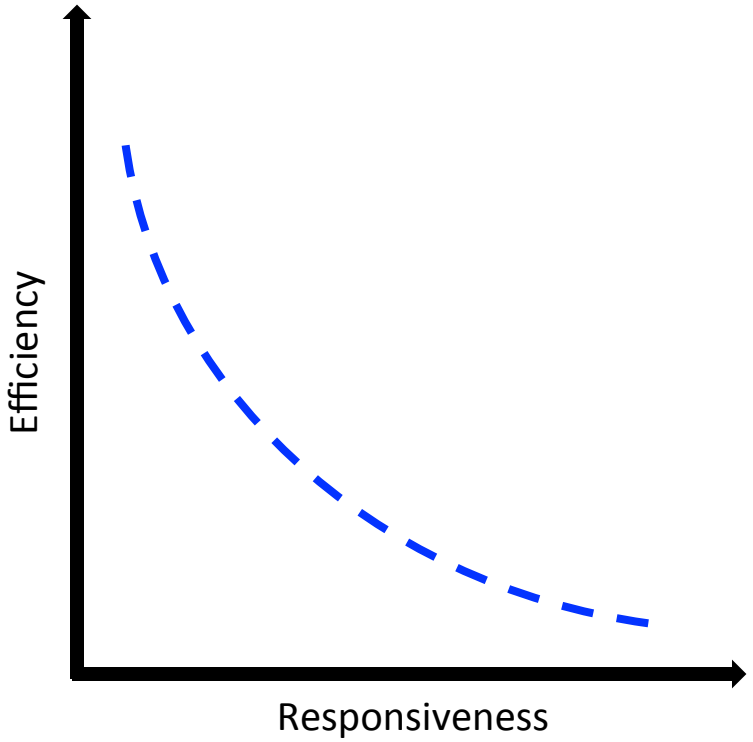

- If we want to do other work, we might call a routine:

```
                mov     r0, #0x4          % PBS MMIO address
                mov     r1, #0x5          % LED MMIO address
loop:           ldr     r2, [r0, #0]      % Read value from switch [1 cycle]
                str     r2 [r1, #0]       % Save value to LED [1 cycle]
                bl      do_some_work      % Do some other work [100 cycles]
                b       loop              % Repeat these steps [1 cycle]
```

- Polling affects the responsiveness of PBS ⇔ LED path!
  - Whenever we're "doing some work," we not polling PBS
  - And the more "other work" we do, the worse the latency gets
- And it affects the efficiency of the processor
  - The ldr/str values don't change very either much
  - So, the processor is mostly wasting CPU cycles (and energy)

# Polling trades off efficiency and responsiveness

```
        mov     r0, #0x4          % PBS MMIO address
        mov     r1, #0x5          % LED MMIO address
loop:   ldr     r2, [r0, #0]      % Read value from switch [1 cycle]
        str     r2 [r1, #0]       % Save value to LED [1 cycle]
        bl      do_some_work      % Do some other work [100 cycles]
        b       loop              % Repeat these steps [1 cycle]
```




- Efficiency
  - Minimizing useless work
  - Maximizing useful work
  - Saving cycles & energy
- Responsiveness
  - Minimizing latency
  - Tight event-action coupling
- Can we do better? Yes!

# Level-triggered interrupts


- Basics:
  - Signaled by asserting a line low or high
  - Interrupting device drives line low or high and *holds it there until it is serviced*
  - Device deasserts when directed to or after serviced
    - Requires some way to tell it to stop.

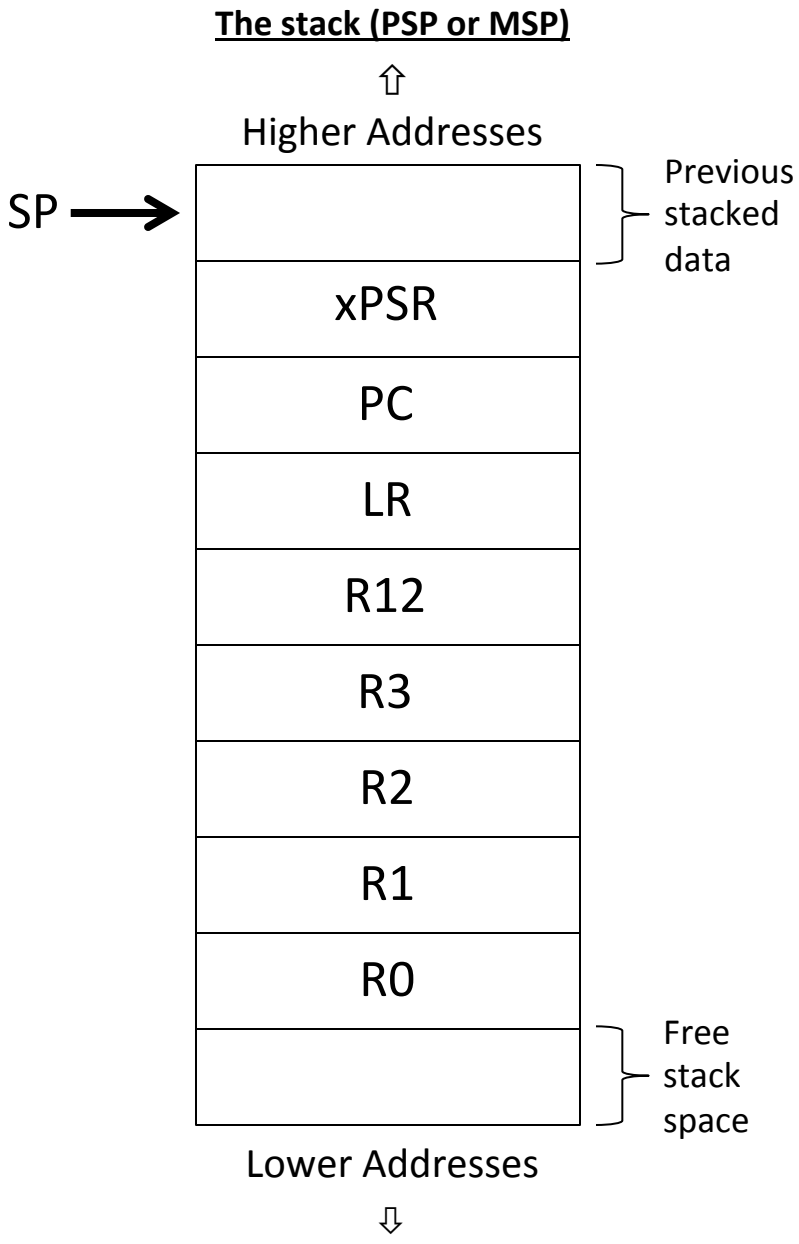
- Sharing?
  - Can share the line among multiple devices
  - Often open-collector or HiZ
    - Active devices assert the line, inactive devices let the line float
  - Easy to share line w/o losing interrupts
  - But servicing increases CPU load
  - And requires CPU to keep cycling through to check
  - Different ISR costs suggests careful ordering of ISR checks
  - Can't detect a new interrupt when one is already asserted

# Edge-triggered interrupts


- Basics:
  - Signaled by a level *transition* (e.g. rising/falling edge)
  - Interrupting device drives a pulse onto INT line
- Sharing *is* possible
  - INT line has a pull up and all devices are OC/OD.
  - Could we miss an interrupt?  Maybe...if close in time
  - What happens if interrupts merge?  Need one more ISR pass
  - Easy to detect "new interrupts”
  - Pitfalls: spurious edges, missed edges
- Source of "lockups" in early computers

# Basic interrupt processing

- Stacking
  - Automatically by CPU
  - Maintains ABI semantics
  - ISRs can be C functions
- Vector Fetch
  - We'll see this next
- Exit: update of SP, LR, PC



# NVIC/Interrupt configuration registers


- ICTR Interrupt Controller Type Register (RW)
- ISER Interrupt Set-Enable Register (RW)
- ICER Interrupt Clear-Enable Register (RW)
- ISPR Interrupt Set-Pending Register (RW)
- ICPR Interrupt Clear-Pending Register (RW)
- IABR Interrupt Active Bit Register (RO)
- IPR Interrupt Priority Register (RW)
- AIRC Application Interrupt and Reset Control

# Enabling and disabling interrupt sources


- Interrupt Set Enable and Clear Enable
  - 0xE000E100-0xE000E11C, 0xE000E180-0xE000E19C

| | | | | |
|---|---|---|---|---|
| 0xE000E100 | SETENA0 | R/W | 0 | Enable for external interrupt #0–31<br>bit[0] for interrupt #0 (exception #16)<br>bit[1] for interrupt #1 (exception #17)<br>…<br>bit[31] for interrupt #31 (exception #47)<br>Write 1 to set bit to 1; write 0 has no effect<br>Read value indicates the current status |
| 0xE000E180 | CLRENA0 | R/W | 0 | Clear enable for external interrupt #0–31<br>bit[0] for interrupt #0<br>bit[1] for interrupt #1<br>…<br>bit[31] for interrupt #31<br>Write 1 to clear bit to 0; write 0 has no effect<br>Read value indicates the current enable status |

# Configuring the NVIC (2)


- Set Pending & Clear Pending
  - 0xE000E200-0xE000E21C, 0xE000E280-0xE000E29C

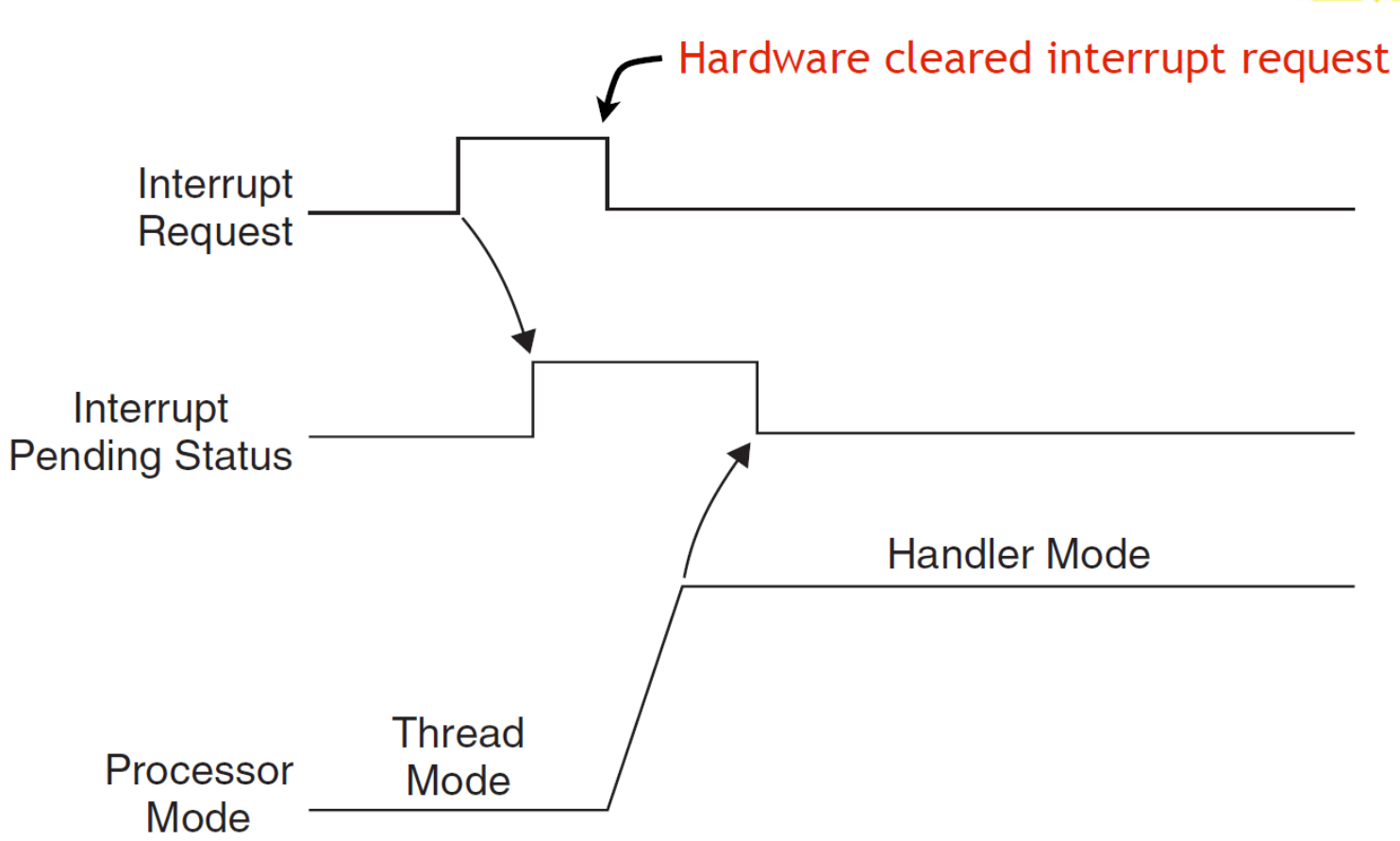
| | | | | |
|---|---|---|---|---|
| 0xE000E200 | SETPEND0 | R/W | 0 | Pending for external interrupt #0–31<br>bit[0] for interrupt #0 (exception #16)<br>bit[1] for interrupt #1 (exception #17)<br>…<br>bit[31] for interrupt #31 (exception #47)<br>Write 1 to set bit to 1; write 0 has no effect<br>Read value indicates the current status |
| 0xE000E280 | CLRPEND0 | R/W | 0 | Clear pending for external interrupt #0–31<br>bit[0] for interrupt #0 (exception #16)<br>bit[1] for interrupt #1 (exception #17)<br>…<br>bit[31] for interrupt #31 (exception #47)<br>Write 1 to clear bit to 0; write 0 has no effect<br>Read value indicates the current pending status |

# Configuring the NVIC (3)


- Interrupt Active Status Register
  - 0xE000E300-0xE000E31C

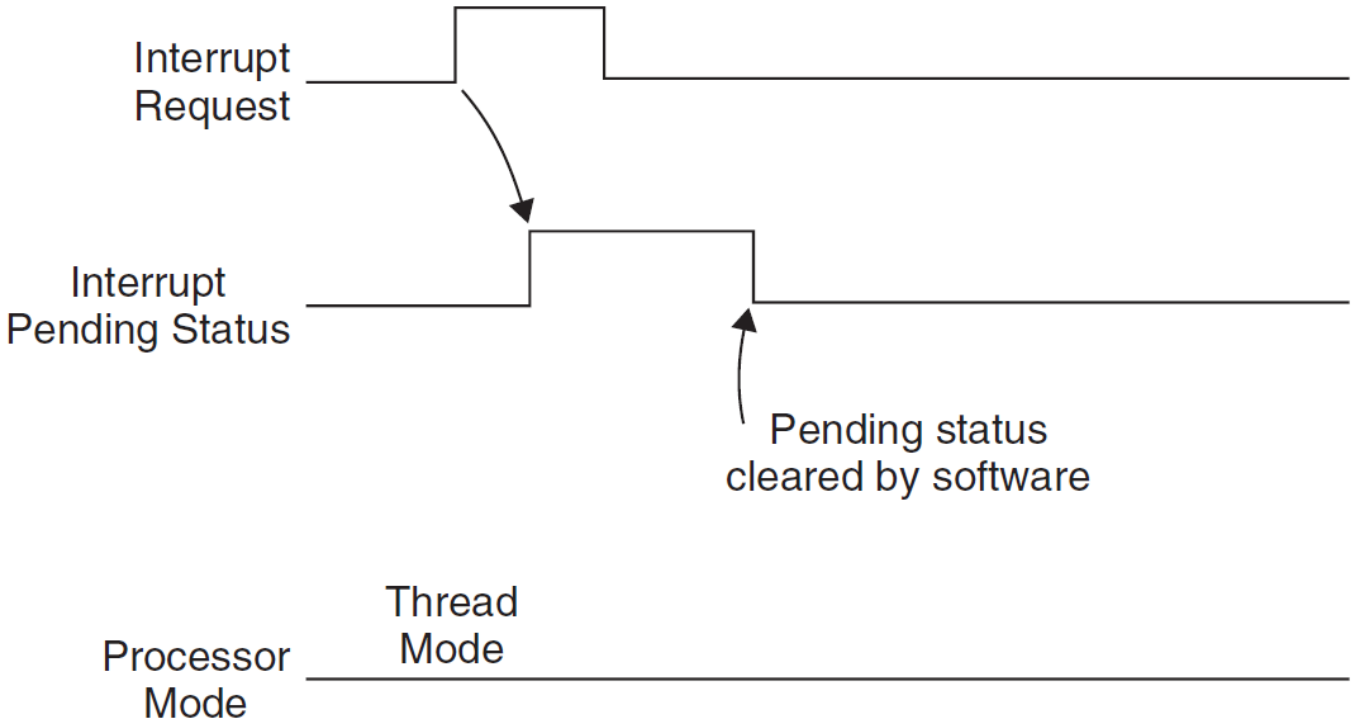
| Address | Name | Type | Reset Value | Description |
|---|---|---|---|---|
| 0xE000E300 | ACTIVE0 | R | 0 | Active status for external interrupt #0–31<br>bit[0] for interrupt #0<br>bit[1] for interrupt #1<br>…<br>bit[31] for interrupt #31 |
| 0xE000E304 | ACTIVE1 | R | 0 | Active status for external interrupt #32–63 |
| … | – | – | – | – |

# Pending interrupts



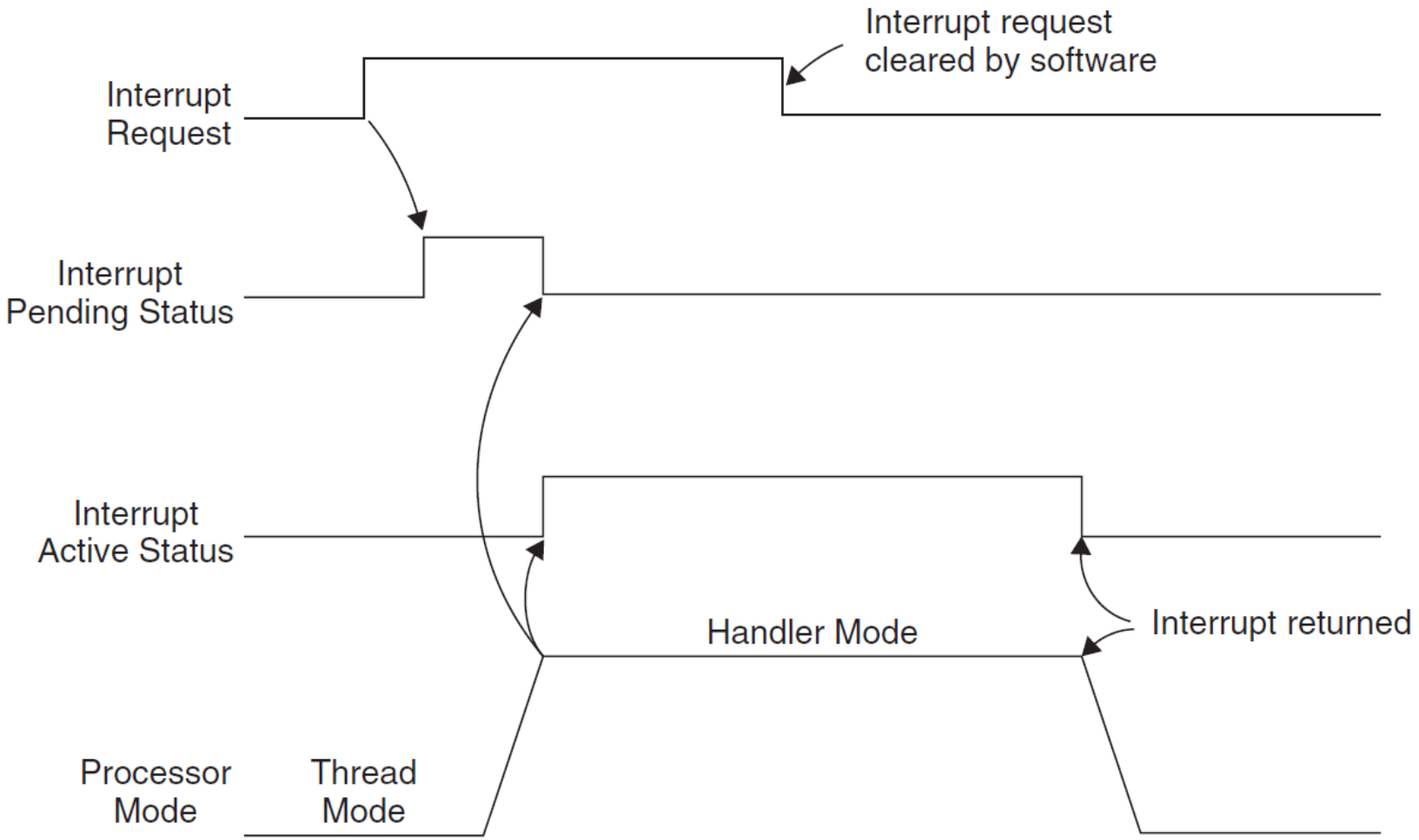

The normal case. Once Interrupt request is seen, processor puts it in "pending" state even if hardware drops the request.
IPS is cleared by the hardware once we jump to the ISR.

This figure and those following are from *The Definitive Guide to the ARM Cortex-M3, Section 7.4*

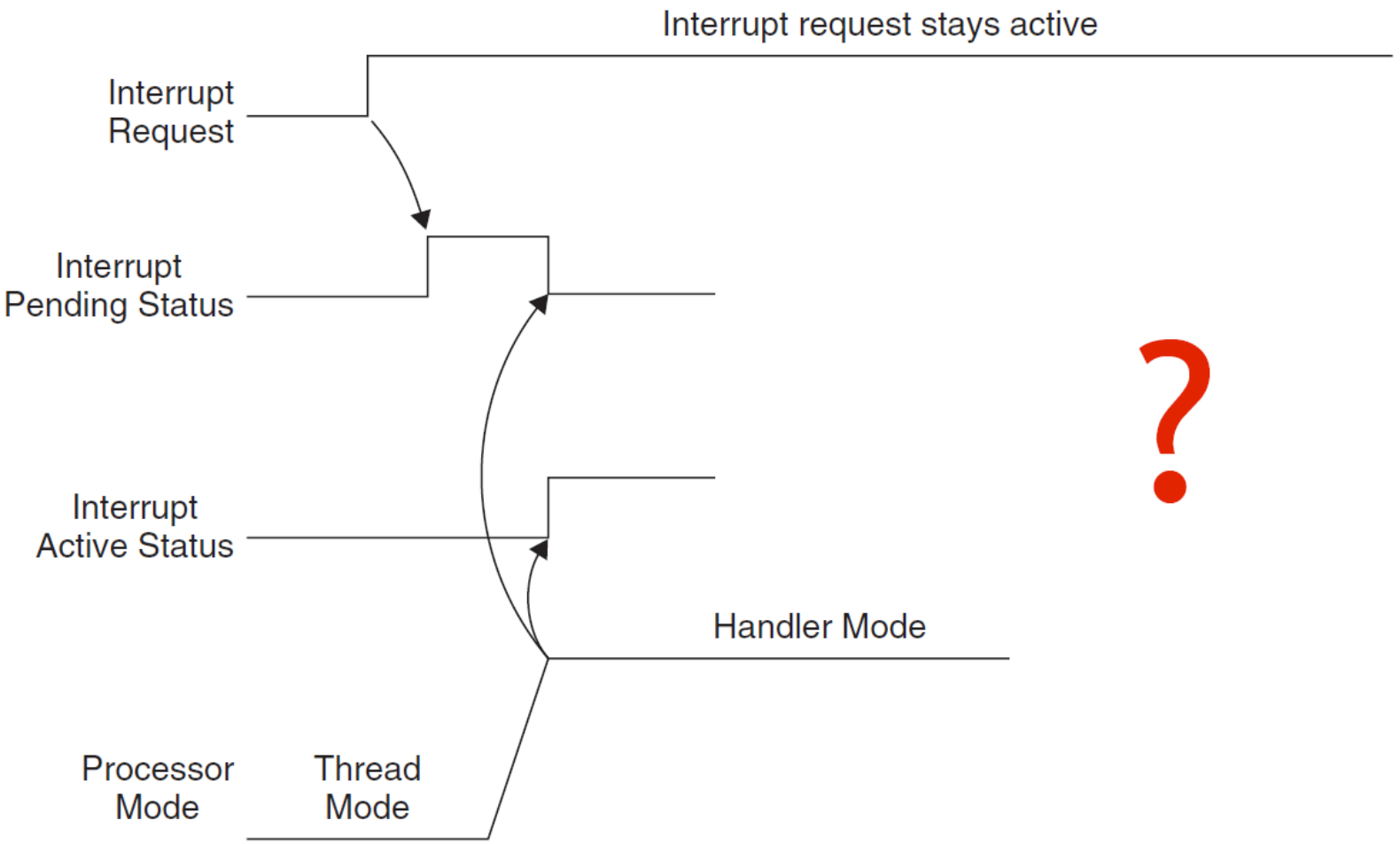



In this case, the processor never took the interrupt because we cleared the IPS by hand (via a memory-mapped I/O register)

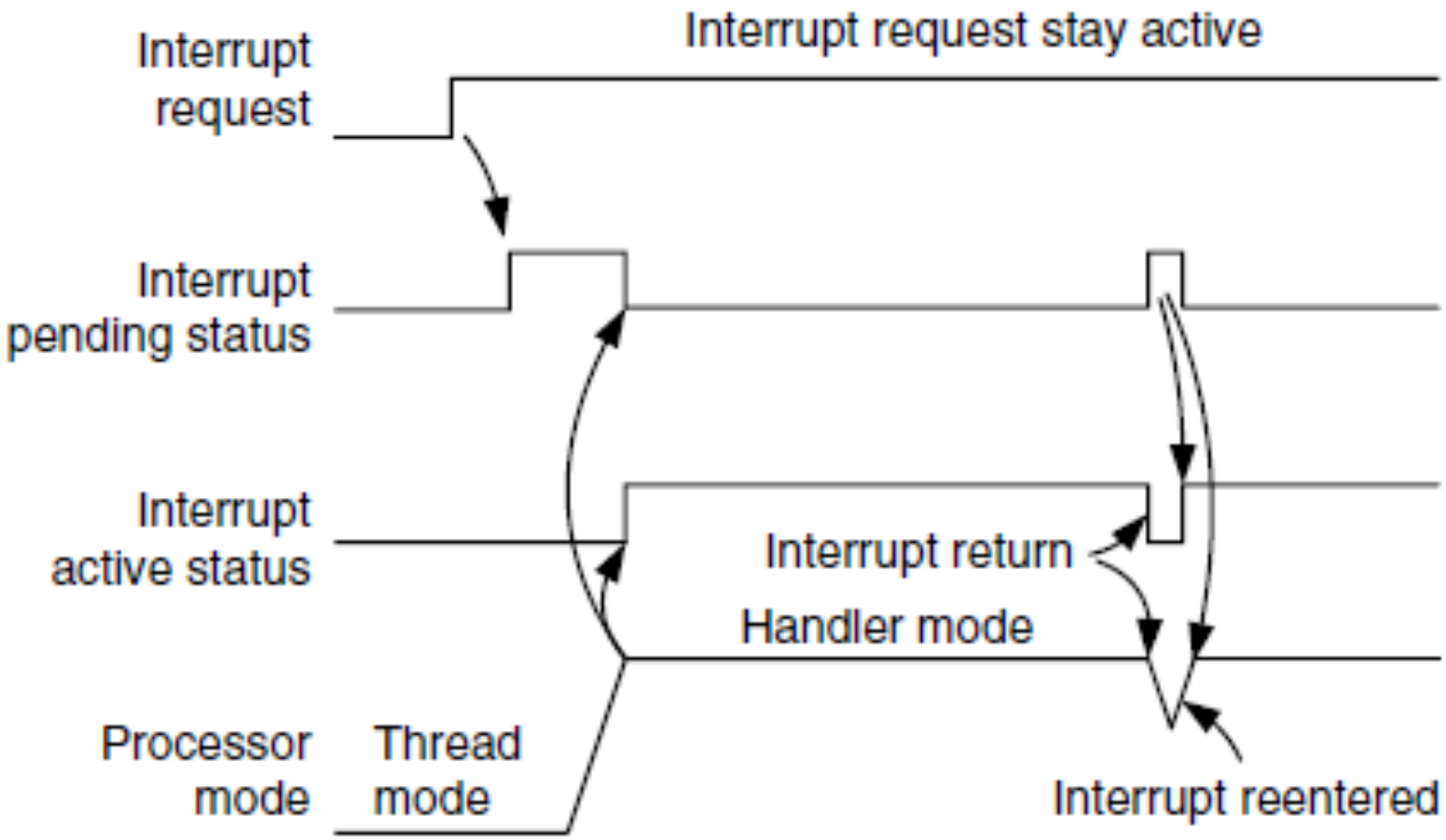
# Active Status set during handler execution



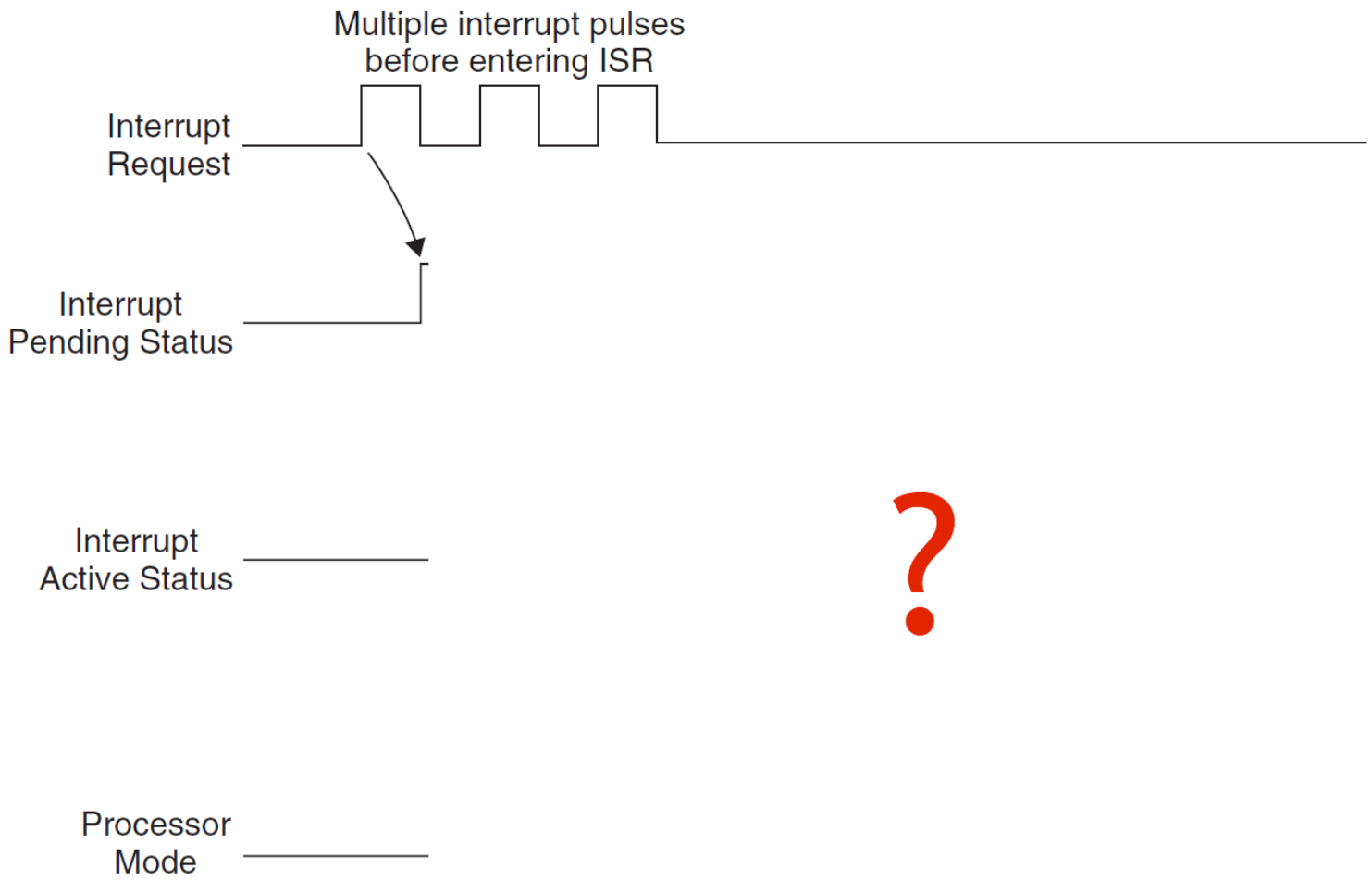
# Interrupt Request not Cleared



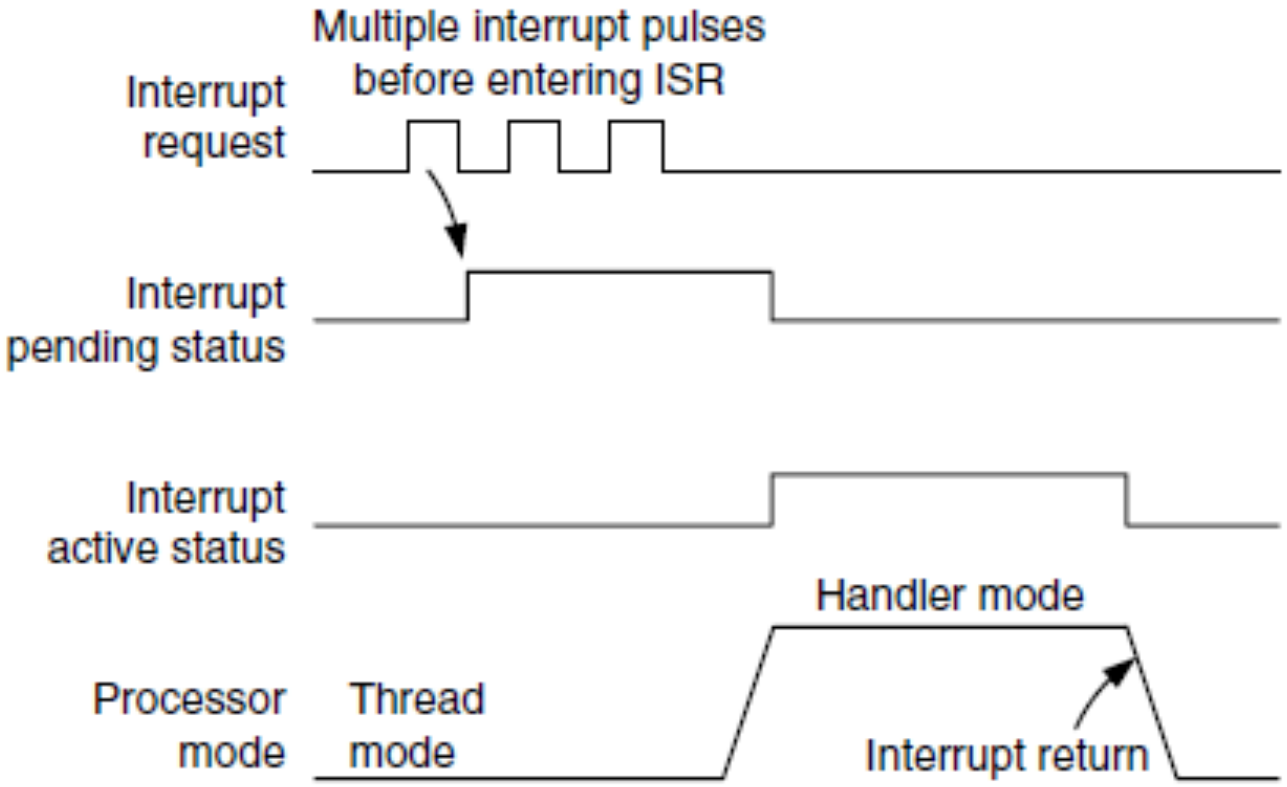
# Answer


Interrupt request stay active
Interrupt request
Interrupt pending status
Interrupt active status
Interrupt return
Handler mode
Processor mode
Thread mode
Interrupt reentered

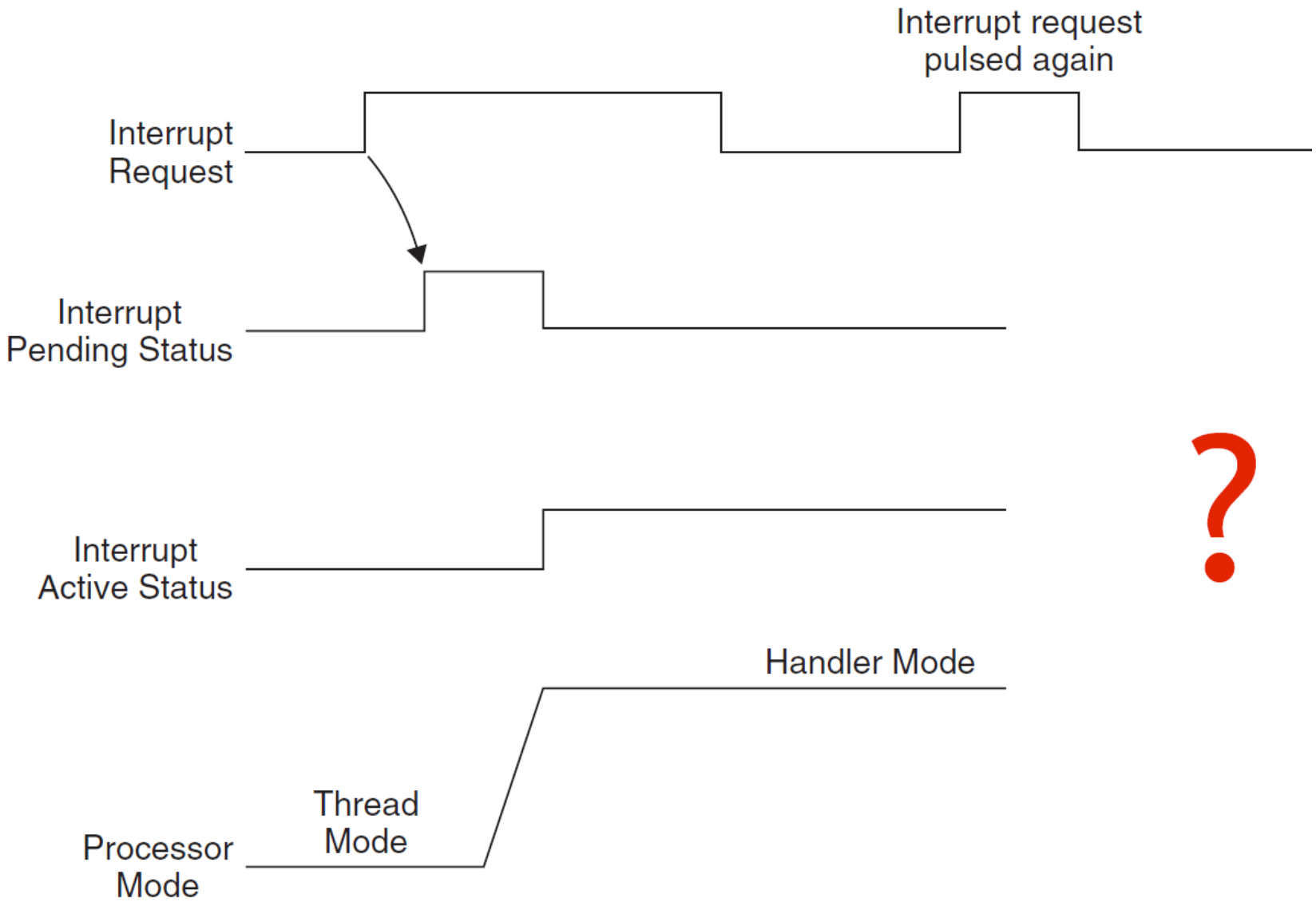
# Interrupt pulses before entering ISR



# Answer


Multiple interrupt pulses
before entering ISR
Interrupt
request
Interrupt
pending status
Interrupt
active status
Handler mode
Processor
mode
Thread
mode
Interrupt return

# New Interrupt Request after Pending Cleared




?

# Interrupt Priority


- What do we do if several interrupts arrive simultaneously?
- NVIC allows priorities for (almost) every interrupt
- 3 fixed highest priorities, up to 256 programmable priorities
  - 128 preemption levels
  - Not all priorities have to be implemented by a vendor

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| Implemented | | | Not implemented, read as zero | | | | |

  - SmartFusion has 32 priority levels, i.e. 0x00, 0x08, … , 0xF8
- Higher priority interrupts can pre-empt lower priorities
- Priority can be sub-divided into priority groups
  - Splits priority register into two halves, preempt priority & subpriority
  - Preempt priority: indicates if an interrupt can preempt another
  - Subpriority: used to determine which is served first if two interrupts of same group arrive concurrently

# Interrupt Priority (2)


- Interrupt priority level registers
  - Range: 0xE000E400 to 0xE000E4EF

| Address | Name | Type | Reset Value | Description |
|---|---|---|---|---|
| 0xE000E400 | PRI_0 | R/W | 0 (8-bit) | Priority-level external interrupt #0 |
| 0xE000E401 | PRI_1 | R/W | 0 (8-bit) | Priority-level external interrupt #1 |
| … | – | – | – | – |
| 0xE000E41F | PRI_31 | R/W | 0 (8-bit) | Priority-level external interrupt #31 |
| … | – | – | – | – |

# Preemption Priority and Subpriority


| Priority Group | Preempt Priority Field | Subpriority Field |
|---|---|---|
| 0 | Bit [7:1] | Bit [0] |
| 1 | Bit [7:2] | Bit [1:0] |
| 2 | Bit [7:3] | Bit [2:0] |
| 3 | Bit [7:4] | Bit [3:0] |
| 4 | Bit [7:5] | Bit [4:0] |
| 5 | Bit [7:6] | Bit [5:0] |
| 6 | Bit [7] | Bit [6:0] |
| 7 | None | Bit [7:0] |

## Application Interrupt and Reset Control Register (Address 0xE000ED0C)

| Bits | Name | Type | Reset Value | Description |
|---|---|---|---|---|
| 31:16 | VECTKEY | R/W | – | Access key; 0x05FA must be written to this field to write to this register, otherwise the write will be ignored; the read-back value of the upper half word is 0xFA05 |
| 15 | ENDIANNESS | R | – | Indicates endianness for data: 1 for big endian (BE8) and 0 for little endian; this can only change after a reset |
| 10:8 | PRIGROUP | R/W | 0 | Priority group |
| 2 | SYSRESETREQ | W | – | Requests chip control logic to generate a reset |
| 1 | VECTCLRACTIVE | W | – | Clears all active state information for exceptions; typically used in debug or OS to allow system to recover from system error (Reset is safer) |
| 0 | VECTRESET | W | – | Resets the Cortex-M3 processor (except debug logic), but this will not reset circuits outside the processor |

# PRIMASK, FAULTMASK, and BASEPRI registers


- What if we quickly want to disable all interrupts?
- Write 1 into PRIMASK to disable all interrupts except NMI
  - MOV R0, #1
  - MSR PRIMASK, R0 ; MSR and MRS are special instructions
- Write 0 into PRIMASK to enable all interrupts

- FAULTMASK is the same as PRIMASK, but it also blocks hard faults (priority = -1)

- What if we want to disable all interrupts below a certain priority?
- Write priority into BASEPRI register
  - MOV R0, #0x60
  - MSR BASEPRI, R0

# Masking

### B1.4.3 The special-purpose mask registers

There are three special-purpose registers which are used for the purpose of priority boosting. Their function is explained in detail in *Execution priority and priority boosting within the core* on page B1-18:

- the exception mask register (PRIMASK) which has a 1-bit value
- the base priority mask (BASEPRI) which has an 8-bit value
- the fault mask (FAULTMASK) which has a 1-bit value.

All mask registers are cleared on reset. All unprivileged writes are ignored.

The formats of the mask registers are illustrated in Table B1-4.

**Table B1-4 The special-purpose mask registers**

| | 31 ... 8 | 7 ... 1 | 0 |
|---|---|---|---|
| PRIMASK | RESERVED | | PM |
| FAULTMASK | RESERVED | | FM |
| BASEPRI | RESERVED | BASEPRI | |

# Interrupt Service Routines


- Automatic saving of registers upon exception
  - PC, PSR, R0-R3, R12, LR
  - This occurs over data bus
- While data bus busy, fetch exception vector
  - i.e. target address of exception handler
  - This occurs over instruction bus
- Update SP to new location
- Update IPSR (low part of xPSR) with exception new #
- Set PC to vector handler
- Update LR to special value EXC_RETURN
- Several other NVIC registers gets updated
- Latency can be as short as 12 cycles (w/o mem delays)

# The xPSR register layout

The APSR, IPSR and EPSR registers are allocated as mutually exclusive bitfields within a 32-bit register. The combination of the APSR, IPSR and EPSR registers is referred to as the xPSR register.

**Table B1-2 The xPSR register layout**

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 16 | 15 | 10 | 9 | 8 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| APSR | N | Z | C | V | Q | | | | | | | | | | |
| IPSR | | | | | | | | | | | | | | 0 or Exception Number | |
| EPSR | | | | | | ICI/IT | | T | | | | ICI/IT | a | | |

# ARM interrupt summary


1. We've got a bunch of memory-mapped registers that control things (**NVIC**)
   - Enable/disable individual interrupts
   - Set/clear pending
   - Interrupt priority and preemption

2. We've got to understand how the hardware interrupt lines interact with the NVIC

3. And how we figure out where to set the PC to point to for a given interrupt source.

# 1. NVIC registers (example)


- Set Pending & Clear Pending
  - 0xE000E200-0xE000E21C, 0xE000E280-0xE000E29C

| | | | | |
|---|---|---|---|---|
| 0xE000E200 | SETPEND0 | R/W | 0 | Pending for external interrupt #0–31<br>bit[0] for interrupt #0 (exception #16)<br>bit[1] for interrupt #1 (exception #17)<br>…<br>bit[31] for interrupt #31 (exception #47)<br>Write 1 to set bit to 1; write 0 has no effect<br>Read value indicates the current status |
| 0xE000E280 | CLRPEND0 | R/W | 0 | Clear pending for external interrupt #0–31<br>bit[0] for interrupt #0 (exception #16)<br>bit[1] for interrupt #1 (exception #17)<br>…<br>bit[31] for interrupt #31 (exception #47)<br>Write 1 to clear bit to 0; write 0 has no effect<br>Read value indicates the current pending status |

# 1. More registers (example)


- Interrupt Priority Level Registers
  - 0xE000E400-0xE000E4EF

| Address | Name | Type | Reset Value | Description |
|---|---|---|---|---|
| 0xE000E400 | PRI_0 | R/W | 0 (8-bit) | Priority-level external interrupt #0 |
| 0xE000E401 | PRI_1 | R/W | 0 (8-bit) | Priority-level external interrupt #1 |
| … | – | – | – | – |
| 0xE000E41F | PRI_31 | R/W | 0 (8-bit) | Priority-level external interrupt #31 |
| … | – | – | – | – |

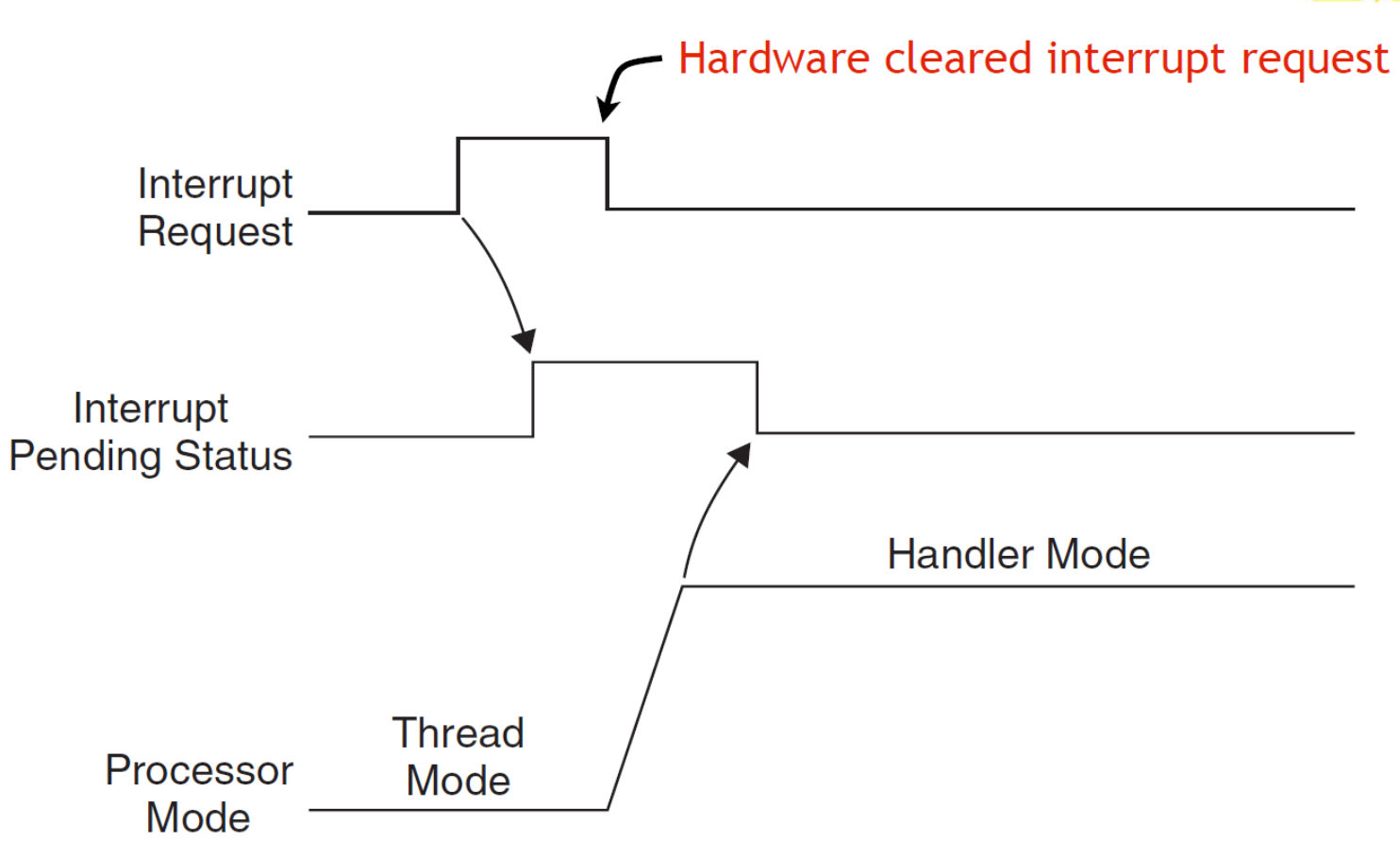
# 1. Yet another part of the NVIC registers!


| Priority Group | Preempt Priority Field | Subpriority Field |
|---|---|---|
| 0 | Bit [7:1] | Bit [0] |
| 1 | Bit [7:2] | Bit [1:0] |
| 2 | Bit [7:3] | Bit [2:0] |
| 3 | Bit [7:4] | Bit [3:0] |
| 4 | Bit [7:5] | Bit [4:0] |
| 5 | Bit [7:6] | Bit [5:0] |
| 6 | Bit [7] | Bit [6:0] |
| 7 | None | Bit [7:0] |

Application Interrupt and Reset Control Register (Address 0xE000ED0C)

| Bits | Name | Type | Reset Value | Description |
|---|---|---|---|---|
| 31:16 | VECTKEY | R/W | – | Access key; 0x05FA must be written to this field to write to this register, otherwise the write will be ignored; the read-back value of the upper half word is 0xFA05 |
| 15 | ENDIANNESS | R | – | Indicates endianness for data: 1 for big endian (BE8) and 0 for little endian; this can only change after a reset |
| 10:8 | PRIGROUP | R/W | 0 | Priority group |
| 2 | SYSRESETREQ | W | – | Requests chip control logic to generate a reset |
| 1 | VECTCLRACTIVE | W | – | Clears all active state information for exceptions; typically used in debug or OS to allow system to recover from system error (Reset is safer) |
| 0 | VECTRESET | W | – | Resets the Cortex-M3 processor (except debug logic), but this will not reset circuits outside the processor |

# 2. How external lines interact with the NVIC




The normal case. Once Interrupt request is seen, processor puts it in "pending" state even if hardware drops the request.
IPS is cleared by the hardware once we jump to the ISR.

This figure and those following are from *The Definitive Guide to the ARM Cortex-M3, Section 7.4*

# 3. How the hardware figures out what to set the PC to


```
g_pfnVectors:
    .word  _estack
    .word  Reset_Handler
    .word  NMI_Handler
    .word  HardFault_Handler
    .word  MemManage_Handler
    .word  BusFault_Handler
    .word  UsageFault_Handler
    .word  0
    .word  0
    .word  0
    .word  0
    .word  SVC_Handler
    .word  DebugMon_Handler
    .word  0
    .word  PendSV_Handler
    .word  SysTick_Handler
    .word  WdogWakeup_IRQHandler
    .word  BrownOut_1_5V_IRQHandler
    .word  BrownOut_3_3V_IRQHandler
.............. (they continue)
```

**Table 7.1** List of System Exceptions

| Exception Number | Exception Type | Priority | Description |
|---|---|---|---|
| 1 | Reset | –3 (Highest) | Reset |
| 2 | NMI | –2 | Nonmaskable interrupt (external NMI input) |
| 3 | Hard fault | –1 | All fault conditions if the corresponding fault handler is not enabled |
| 4 | MemManage fault | Programmable | Memory management fault; Memory Protection Unit (MPU) violation or access to illegal locations |
| 5 | Bus fault | Programmable | Bus error; occurs when Advanced High-Performance Bus (AHB) interface receives an error response from a bus slave (also called *prefetch abort* if it is an instruction fetch or *data abort* if it is a data access) |
| 6 | Usage fault | Programmable | Exceptions resulting from program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor) |
| 7–10 | Reserved | NA | — |
| 11 | SVC | Programmable | Supervisor Call |
| 12 | Debug monitor | Programmable | Debug monitor (breakpoints, watchpoints, or external debug requests) |
| 13 | Reserved | NA | — |
| 14 | PendSV | Programmable | Pendable Service Call |
| 15 | SYSTICK | Programmable | System Tick Timer |

**Table 7.2** List of External Interrupts

| Exception Number | Exception Type | Priority |
|---|---|---|
| 16 | External Interrupt #0 | Programmable |
| 17 | External Interrupt #1 | Programmable |
| … | … | … |
| 255 | External Interrupt #239 | Programmable |

# What happens when we return from an ISR?


- Interrupt exiting process
  - System restoration needed (different from branch)
  - Special LR value could be stored (0xFFFFFFFx)
- Tail chaining
  - When new exception occurs
  - But CPU handling another exception of same/higher 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
- 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

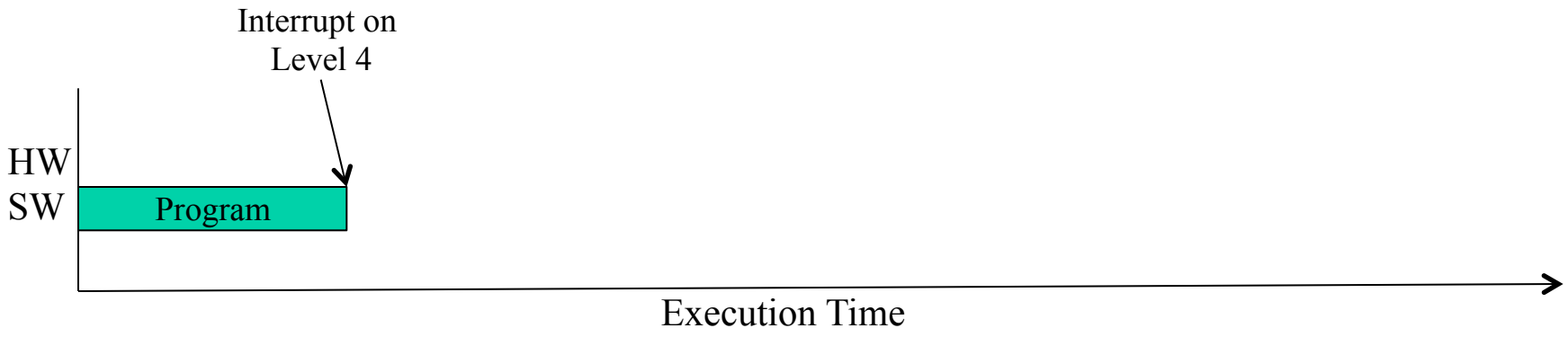
# What happens when we return from an ISR?

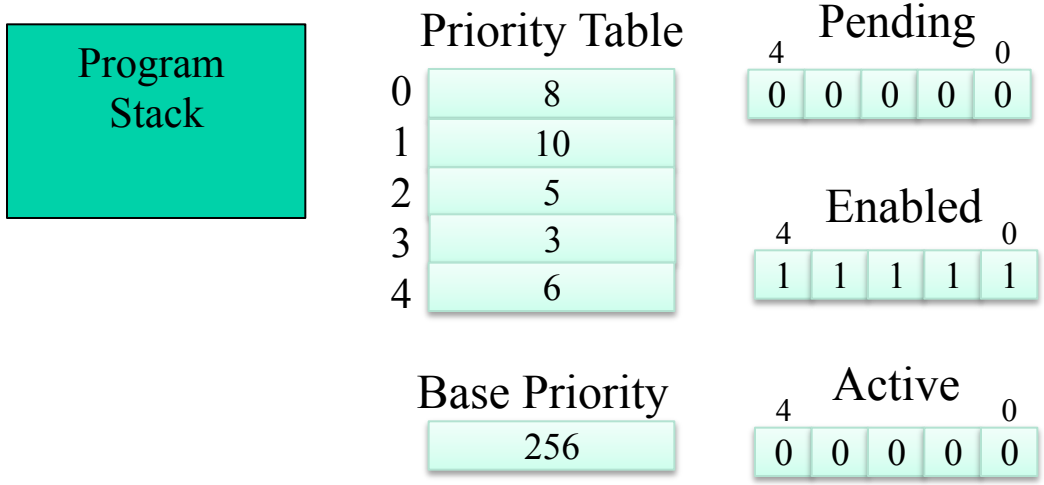

- Interrupt exiting process
  - System restoration needed (different from branch)
  - Special LR value could be stored (0xFFFFFFFx)

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

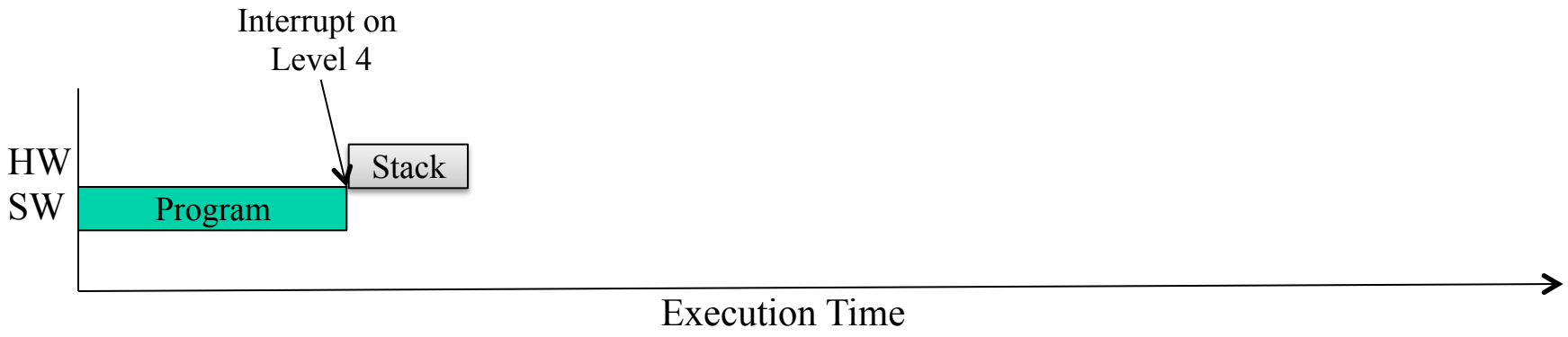
- First time presenting, so errors in the animation are unintentional but may be present (lets find out together)

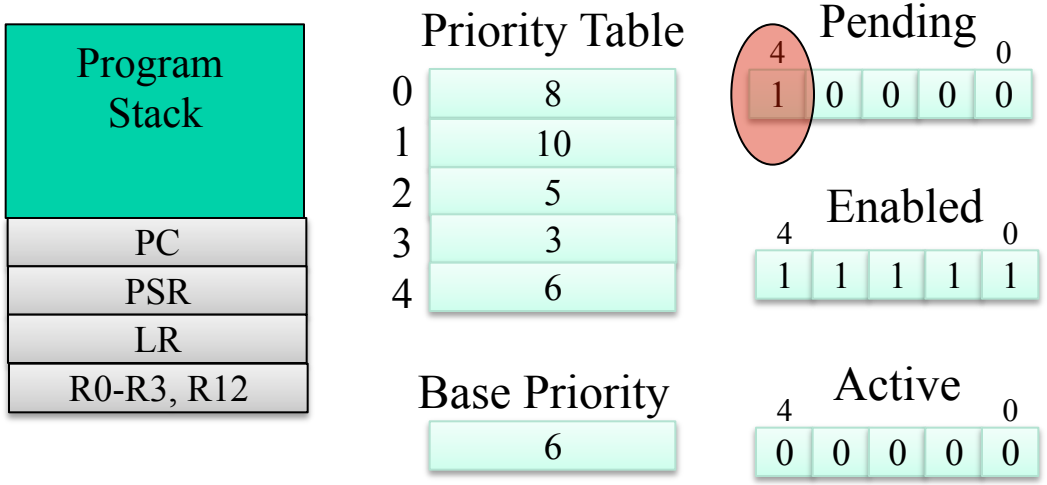
# Single Interrupt



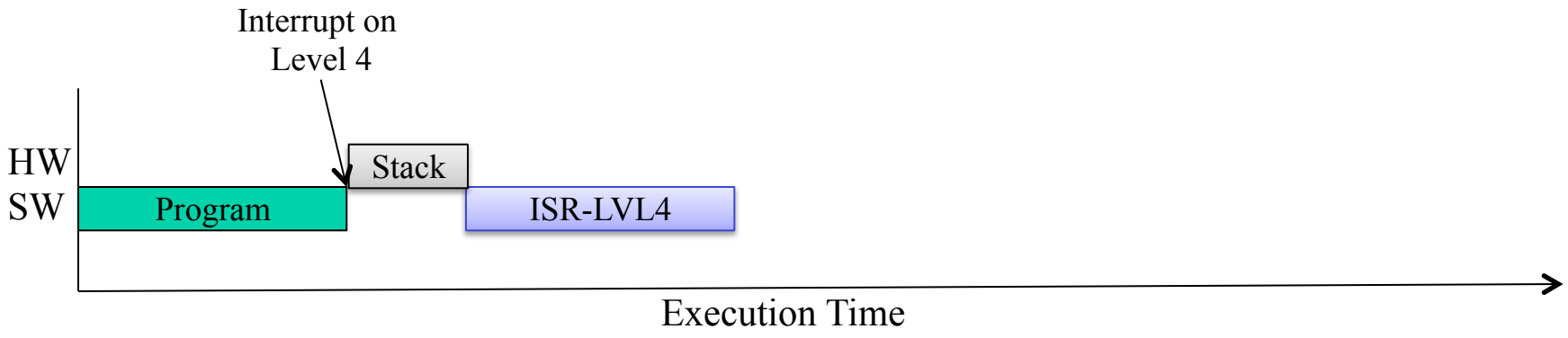



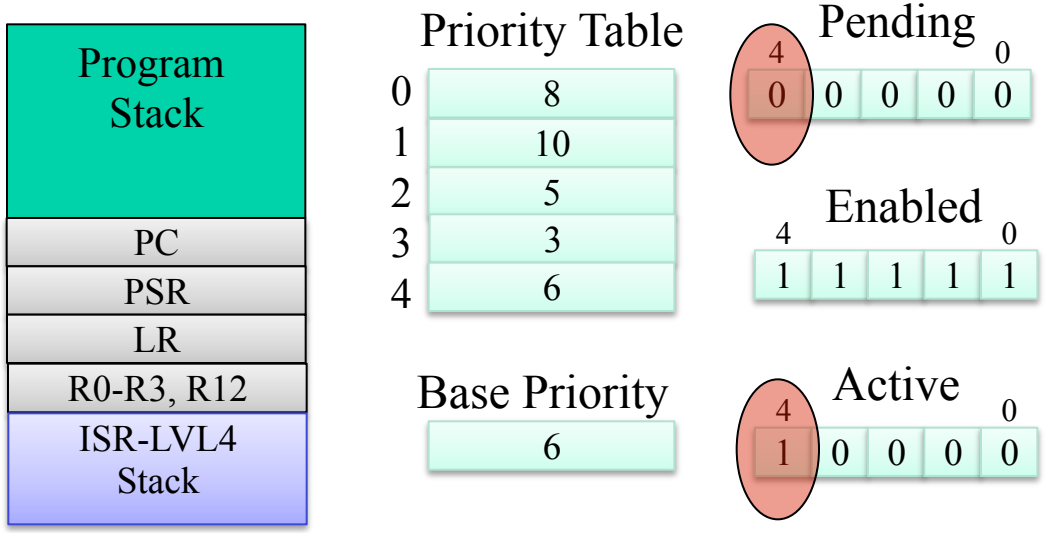
# Single Interrupt



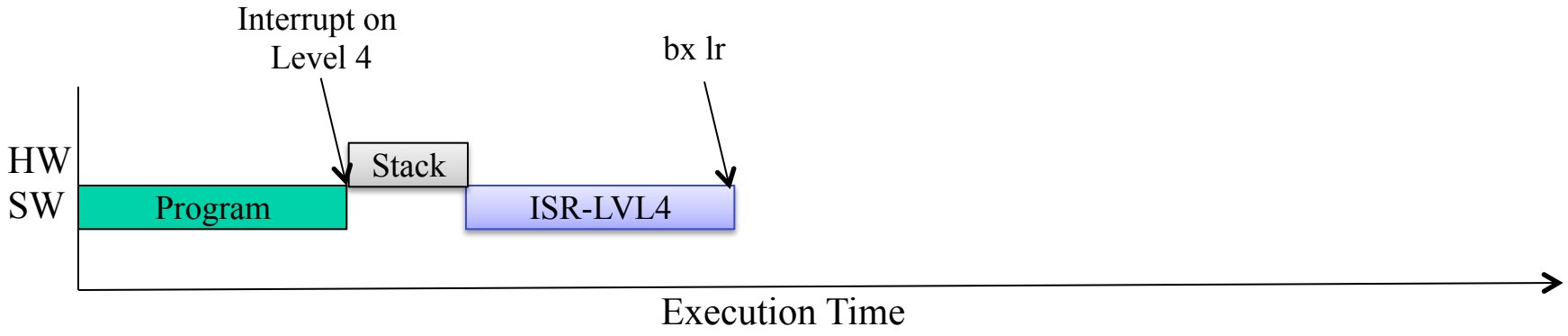



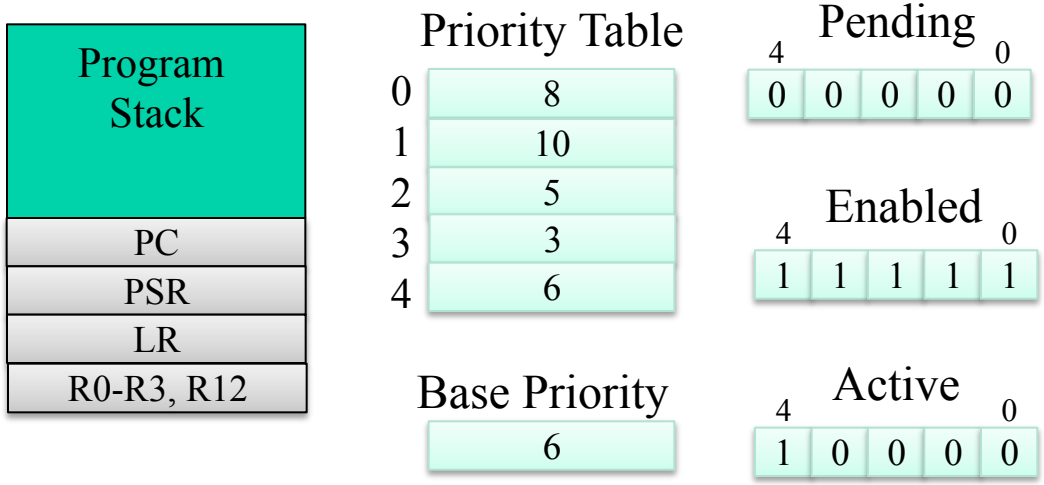
# Single Interrupt



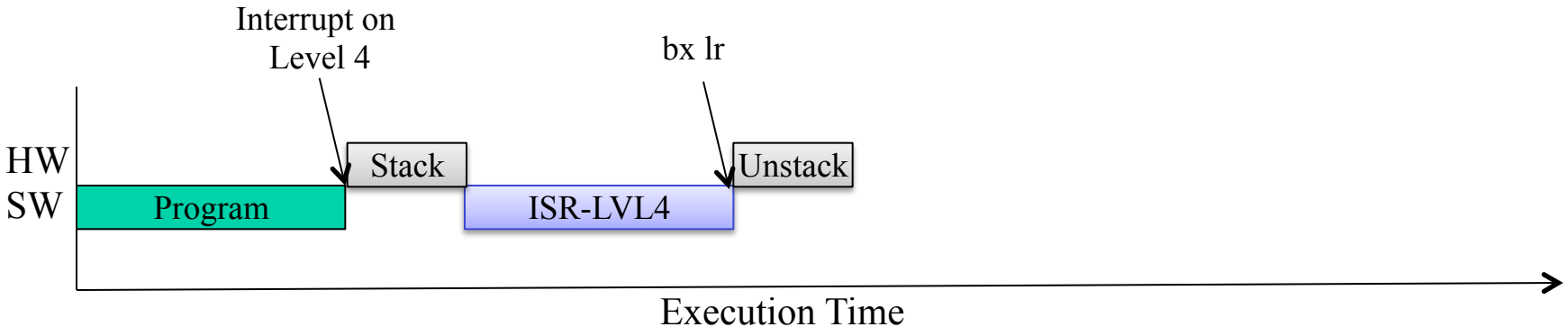



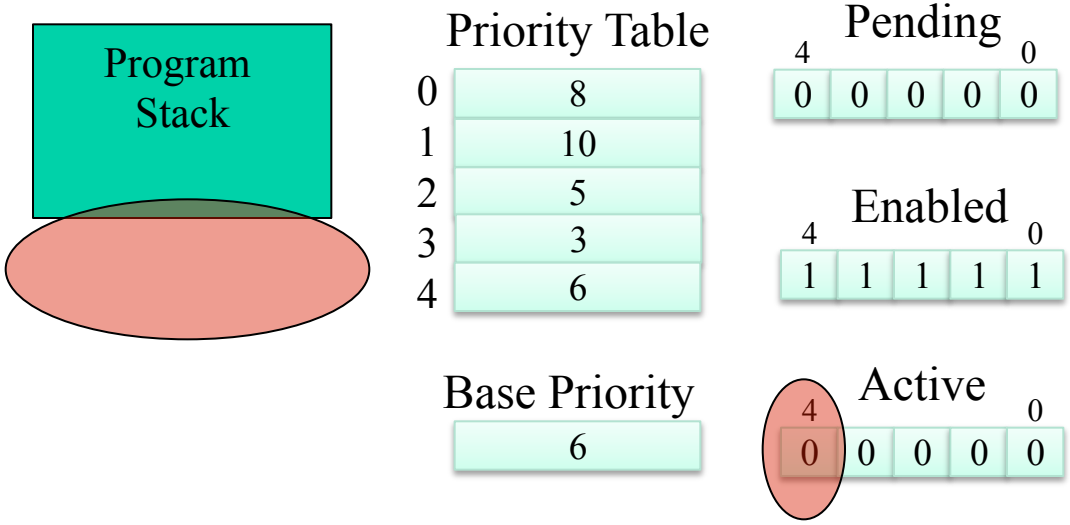
# Single Interrupt



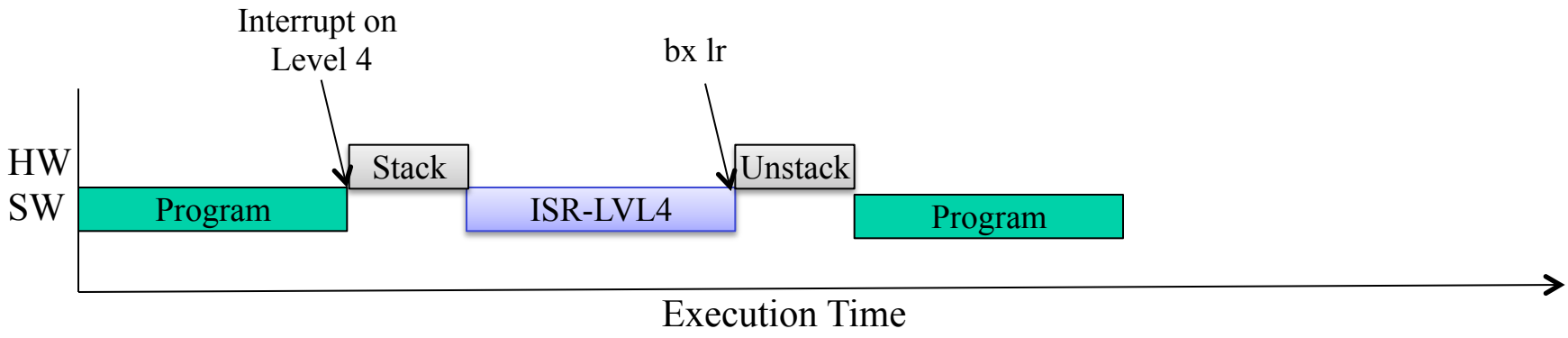



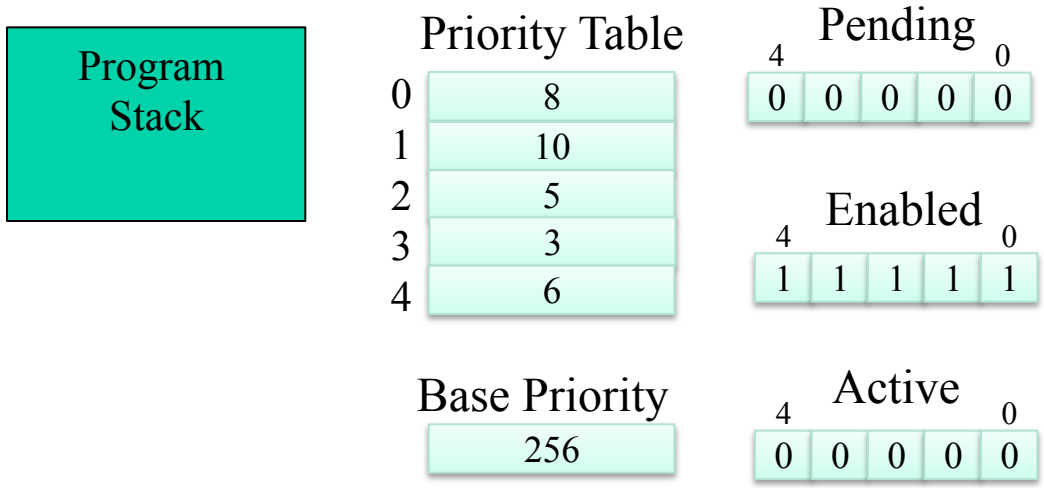
# Single Interrupt



# Single Interrupt

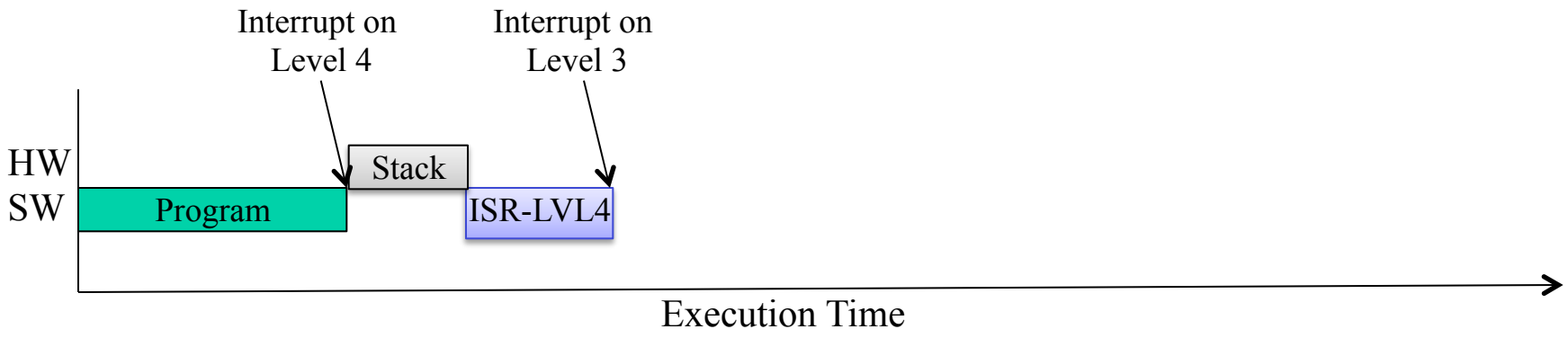


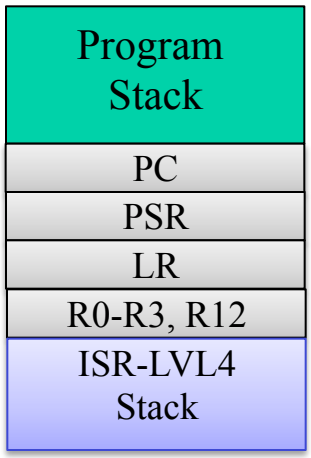



# Nested Interrupts


- 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




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 |

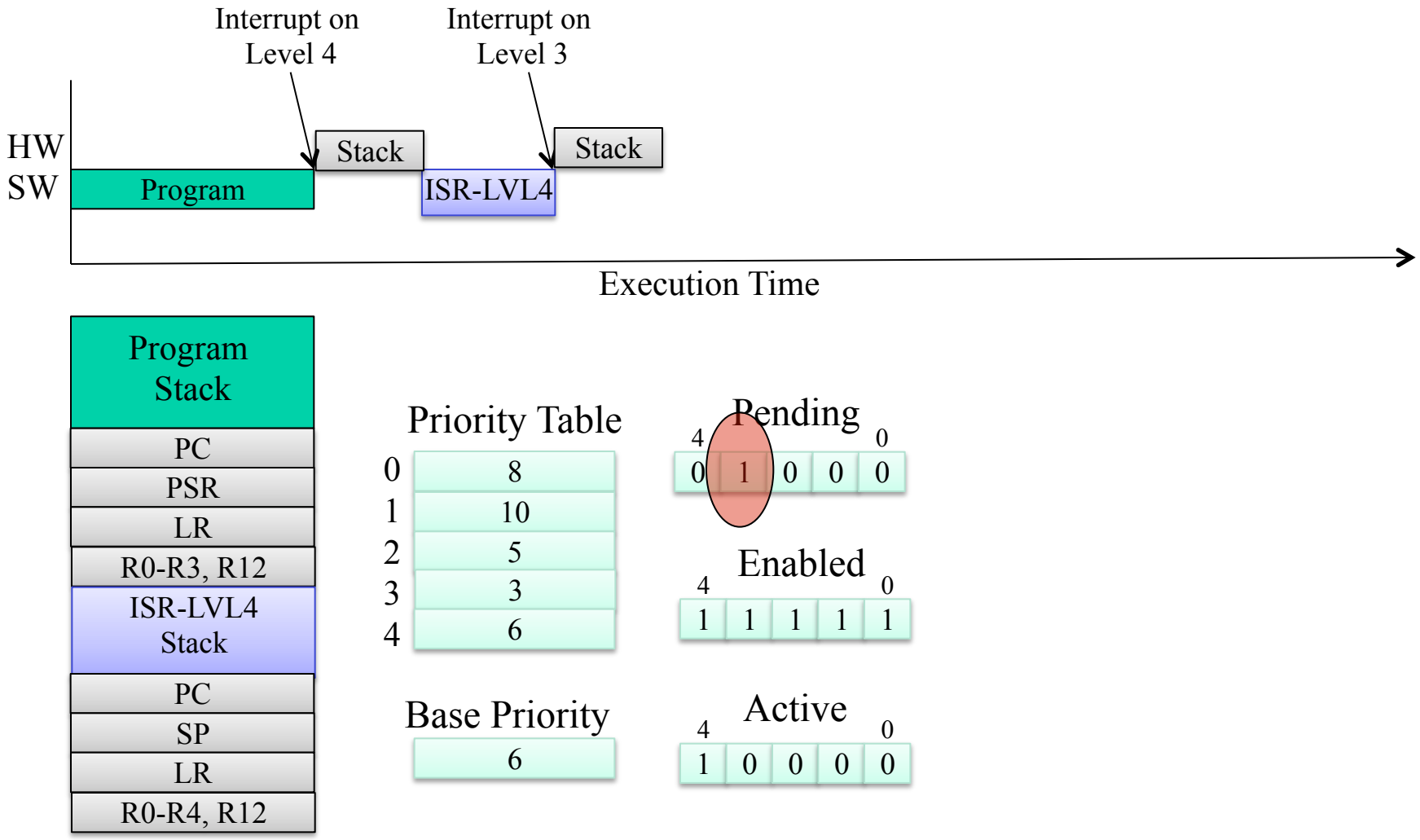
Base Priority

| |
|---|
| 6 |

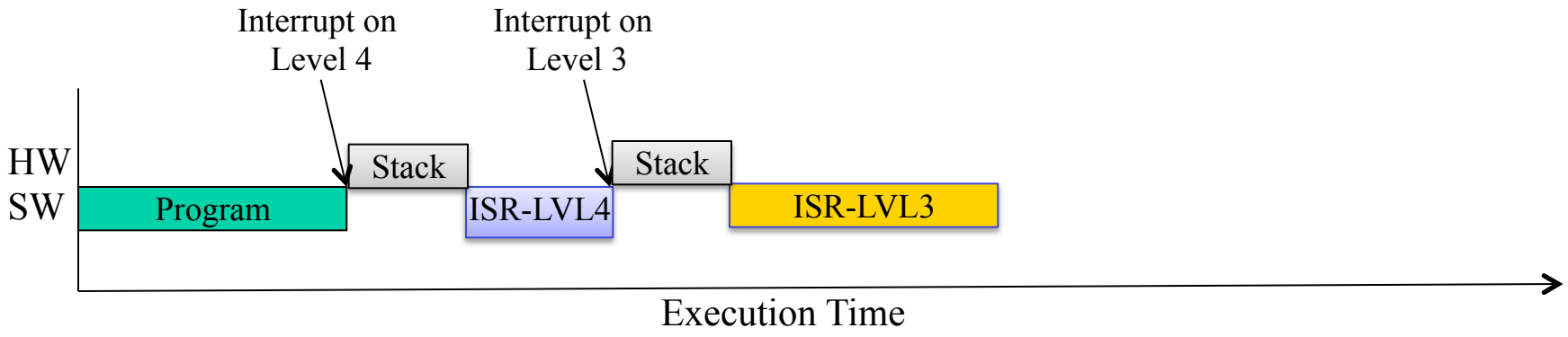
Active

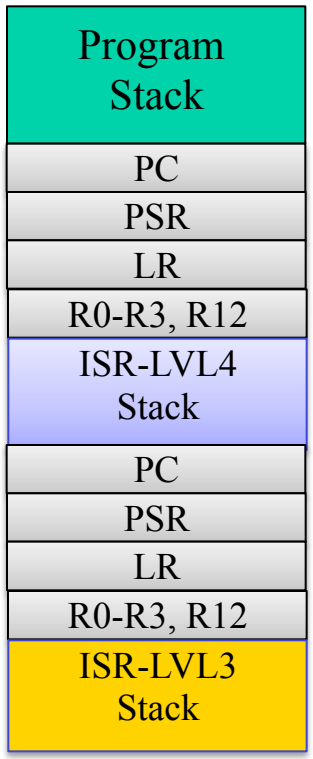
| 4 | | | | 0 |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 |

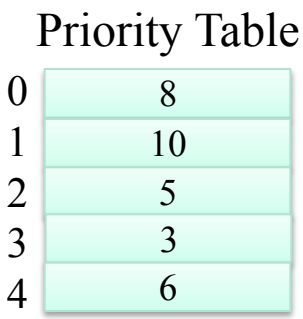
# Nested Interrupt

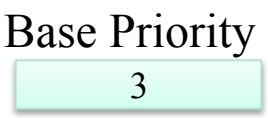


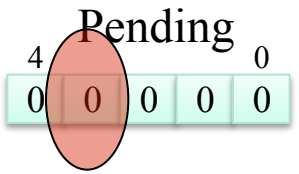
# Nested Interrupt

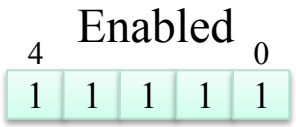


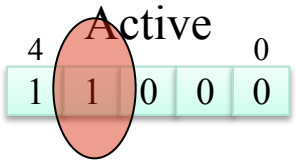



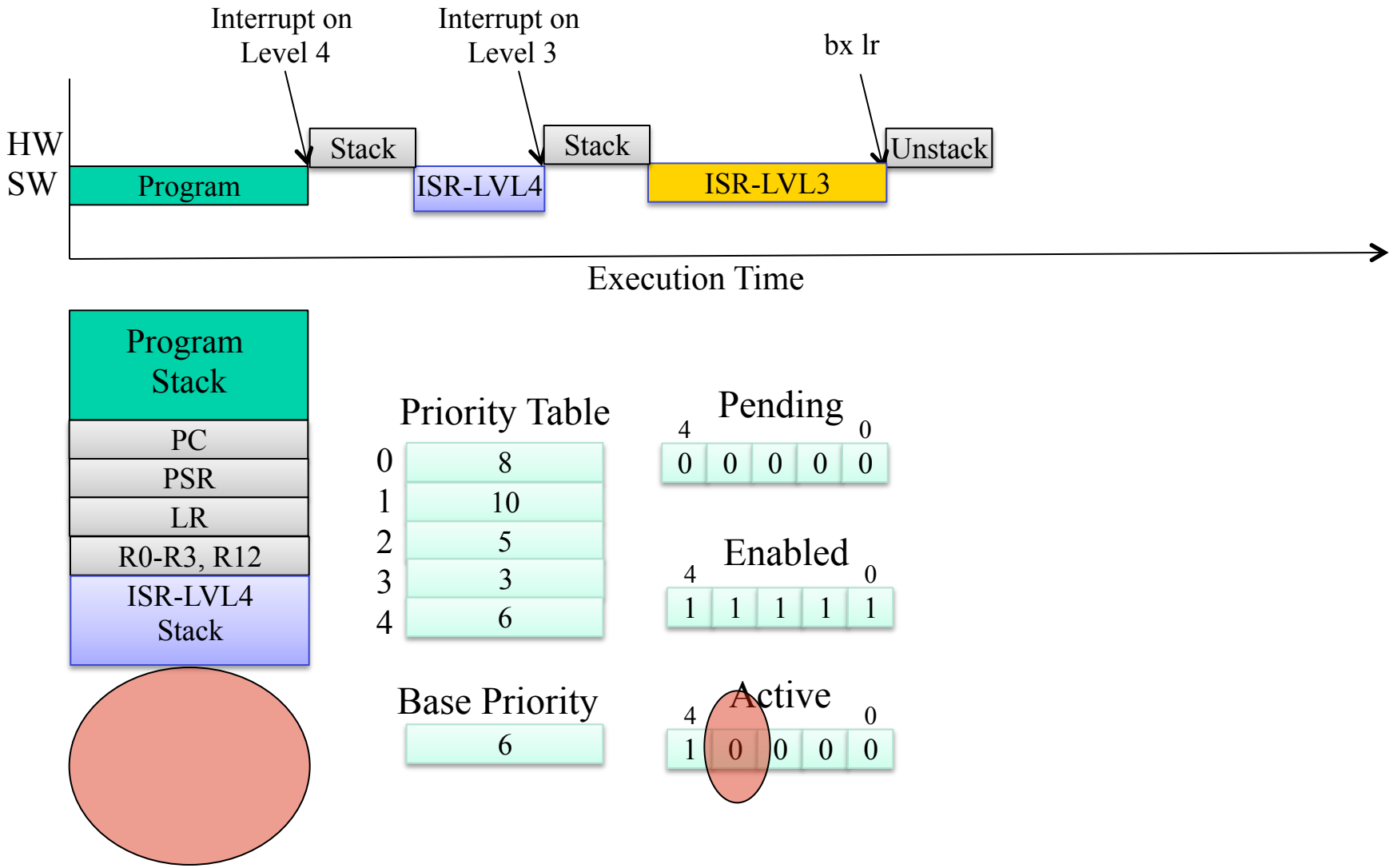



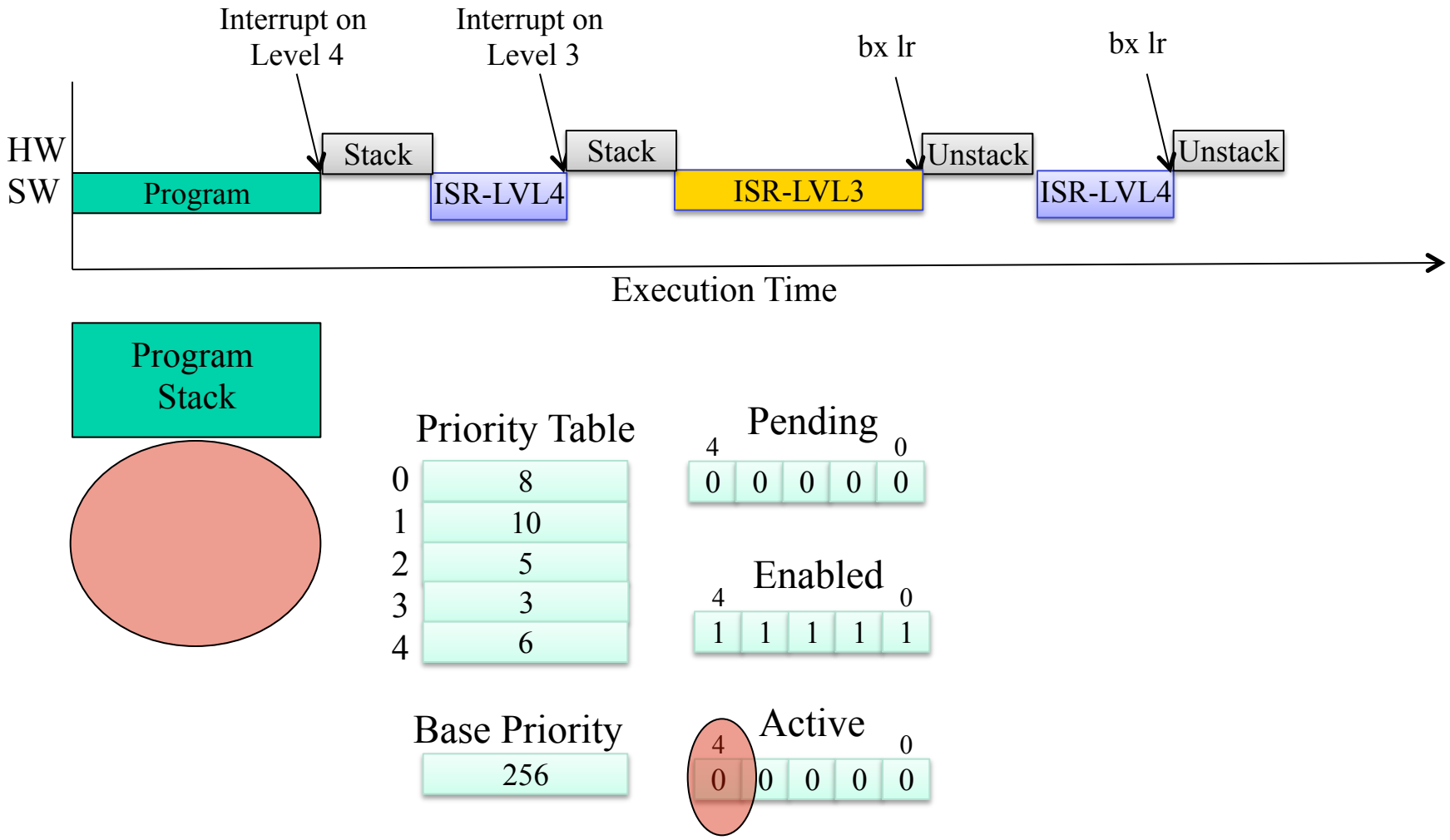



# Nested Interrupt



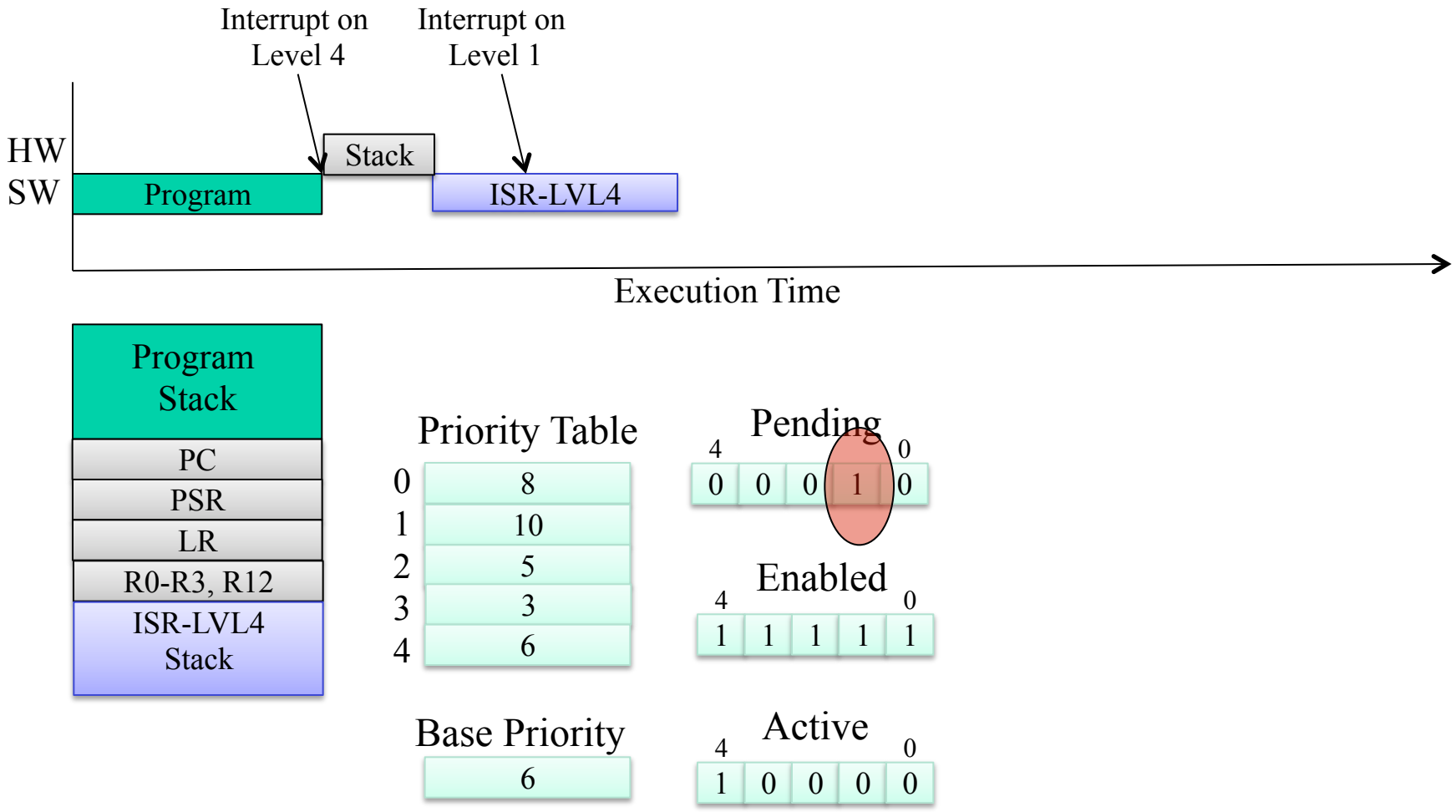
# Nested Interrupt



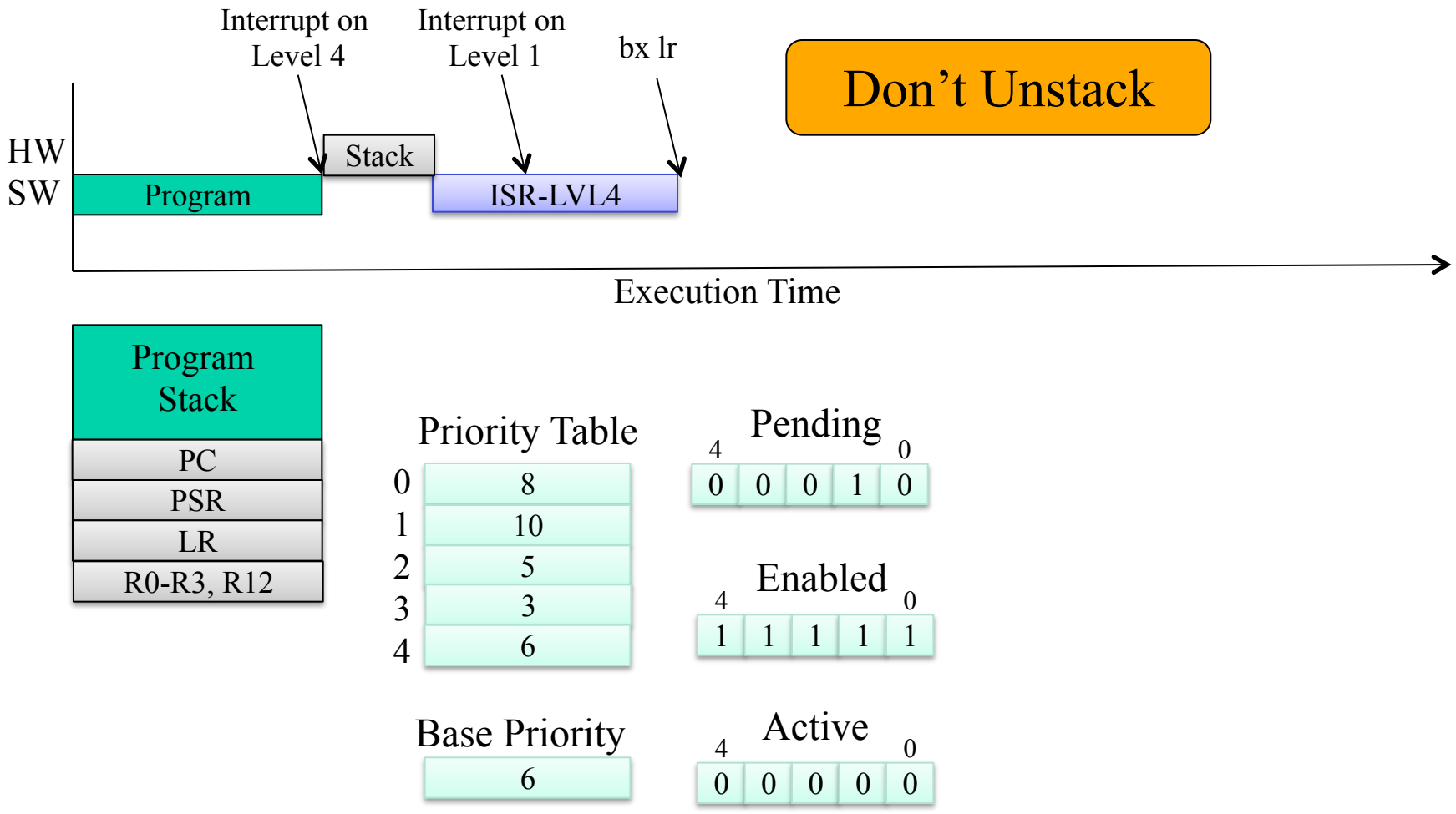
# Tail Chaining


- 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

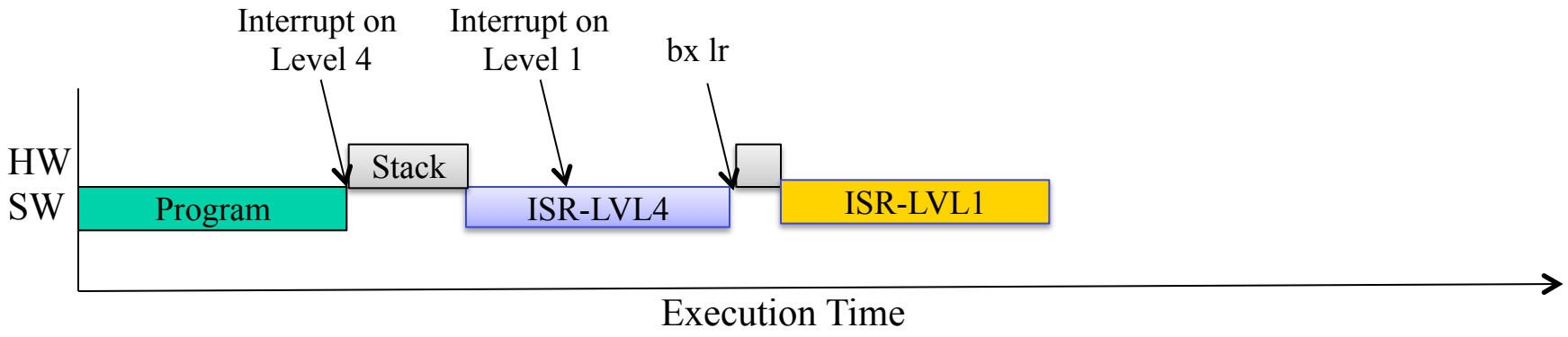
# Tail-Chaining

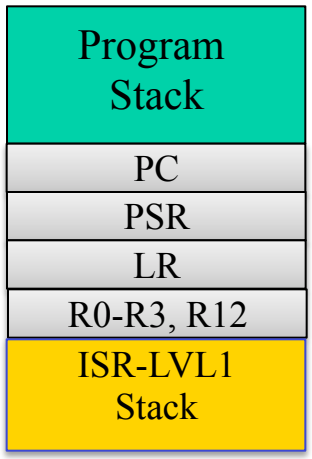


# Tail-Chaining



# 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 |

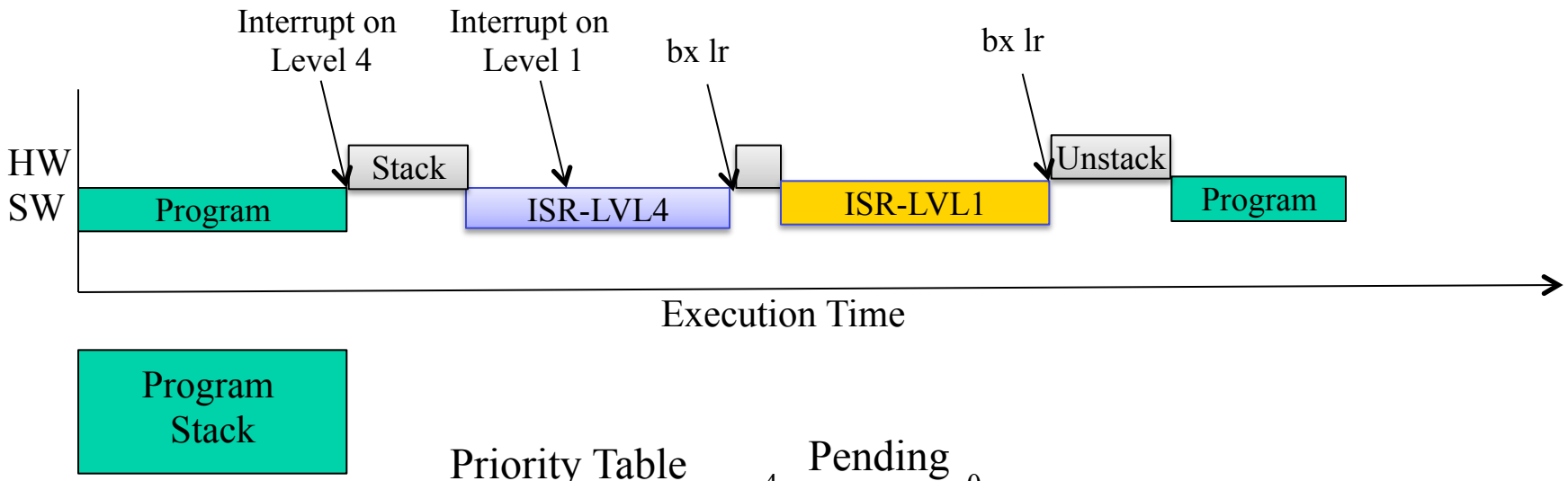
Base Priority

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 |

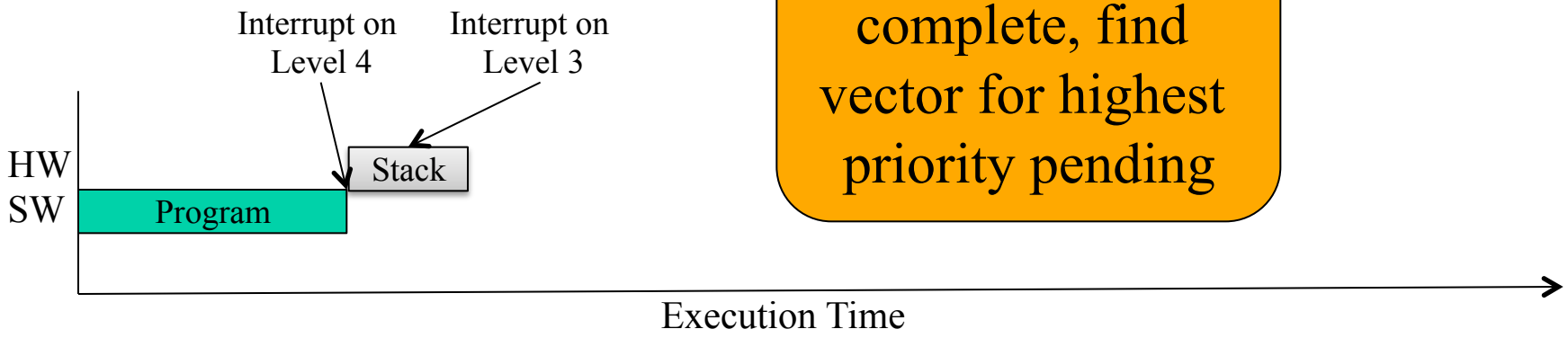
Active

| 4 | | | | 0 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |

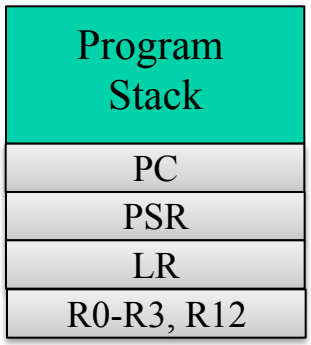
# Late Arrival


- 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 |

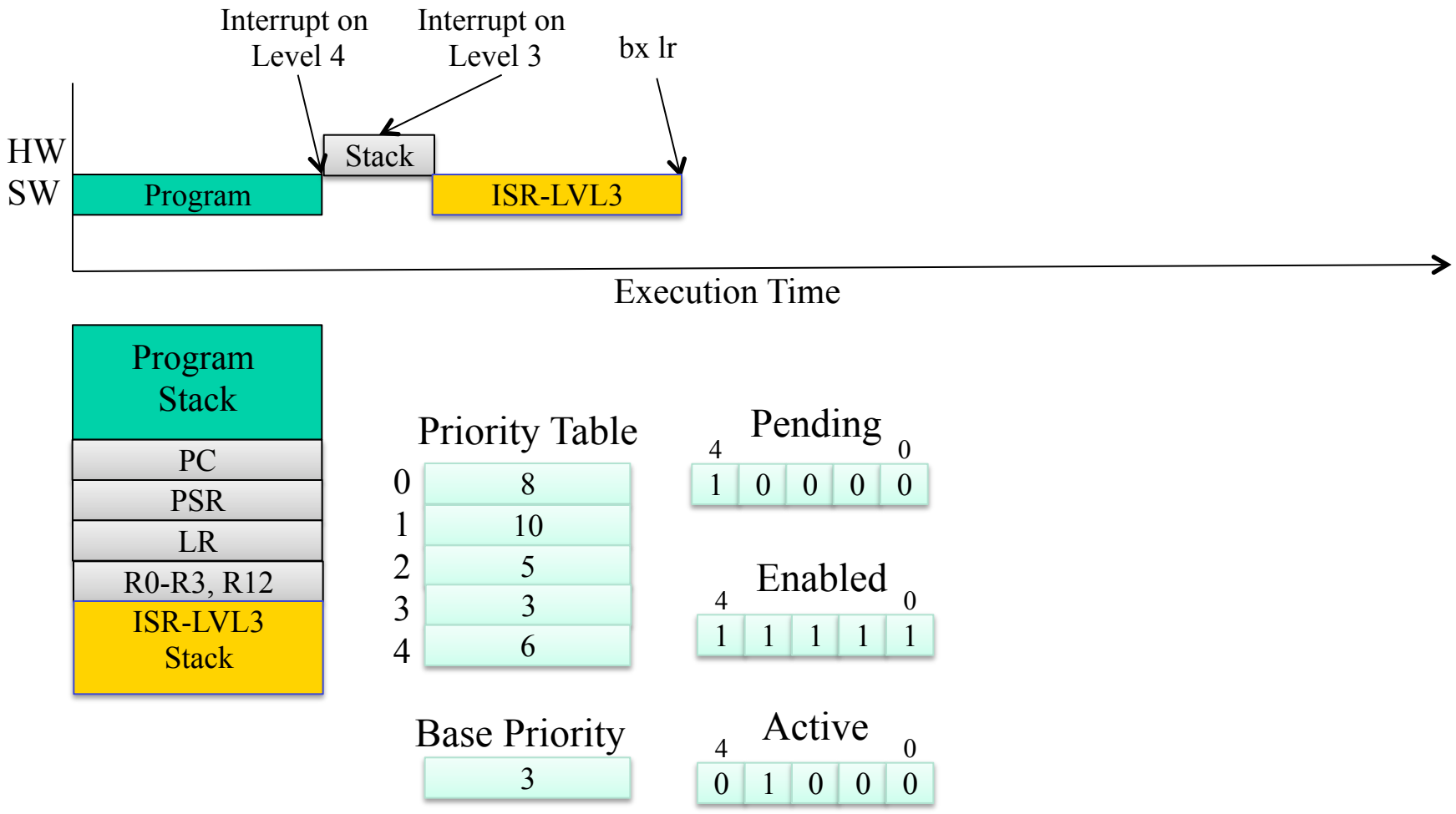
Base Priority

256

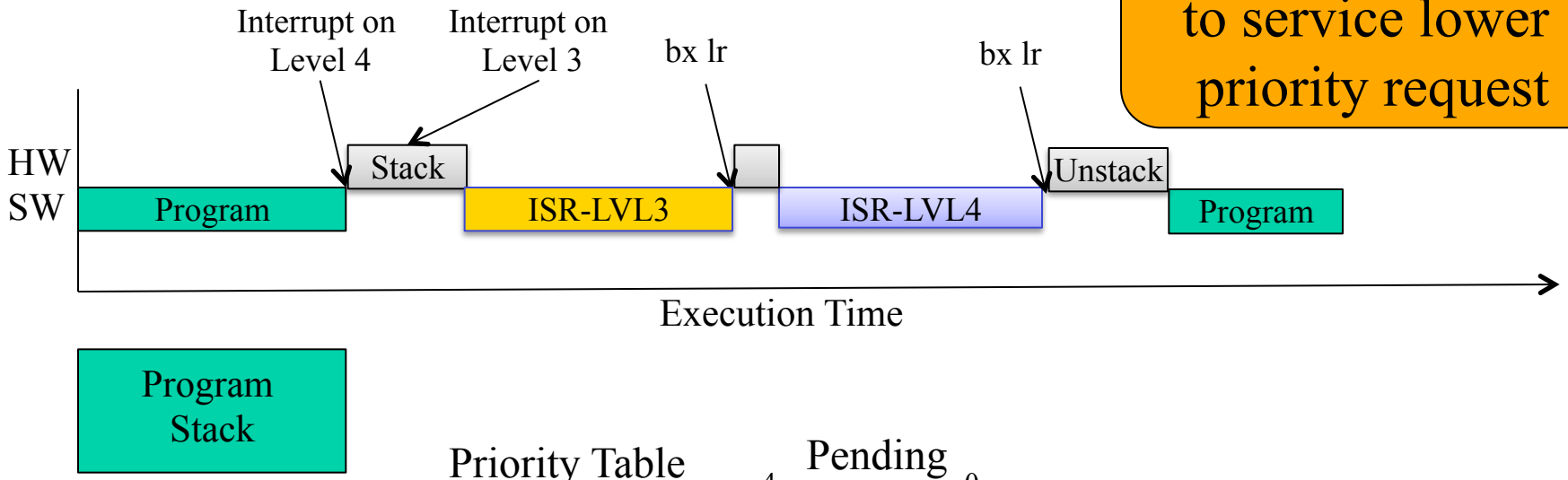
Active

| 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)

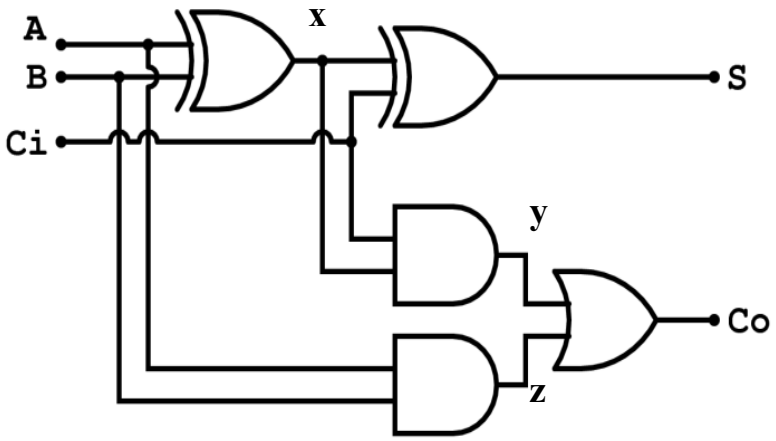

- What data structure?
  - Data needs be sorted
    - Inserting one thing at a time
  - We always pop from one end
  - But we add in sorted order.

# 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

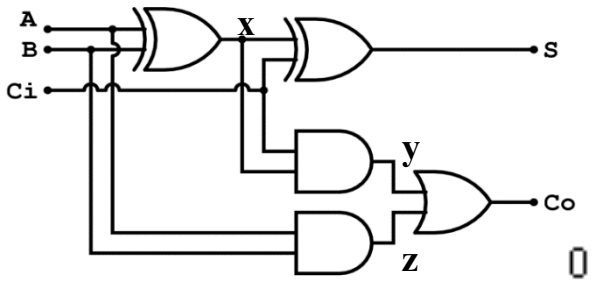
# Timing delays and propagation




Full adder (from Wikipedia)

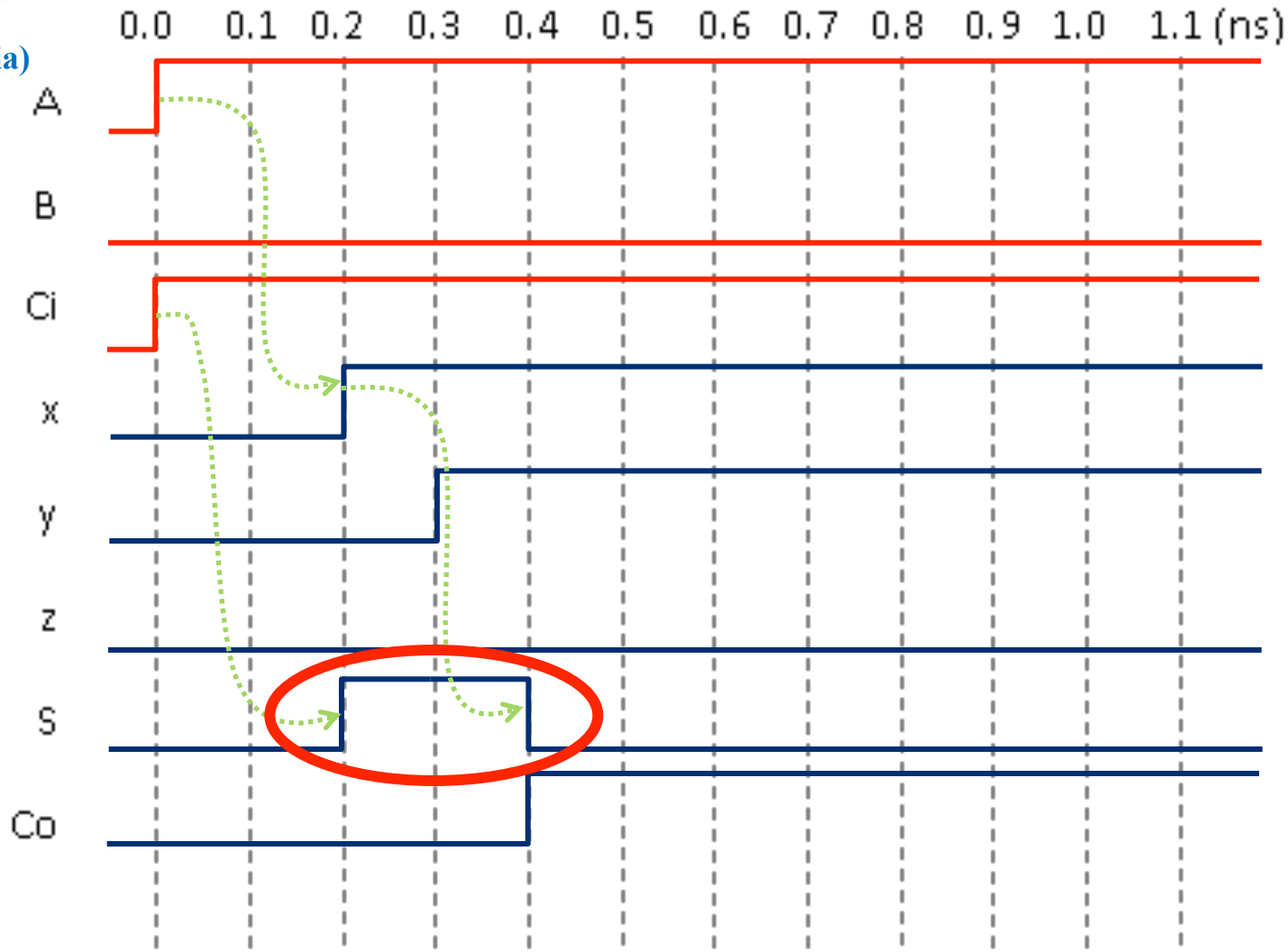
- Assume
  - XOR delay = 0.2ns
  - AND delay = 0.1ns
  - OR delay = 0.1 ns
- What is the worst case propagation delay for this circuit?

# 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.



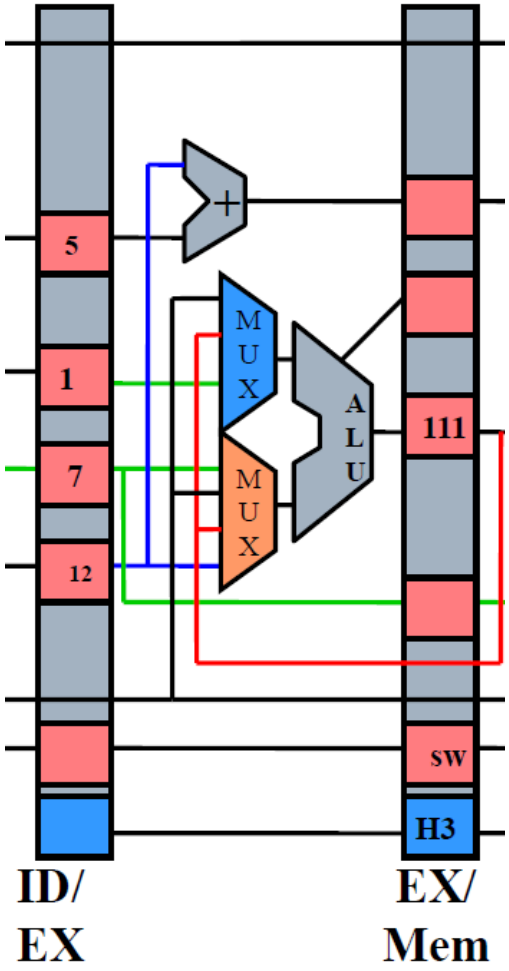

Only selected causality arrows shown…

# 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!

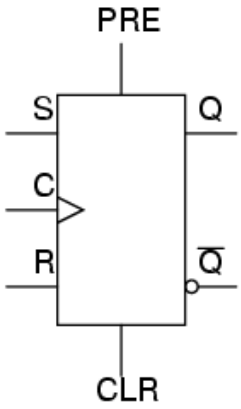
# Effect of Glitches

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

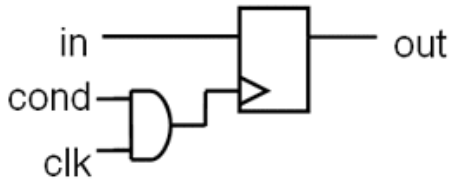


# So, how can glitches hurt us?

- 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.**

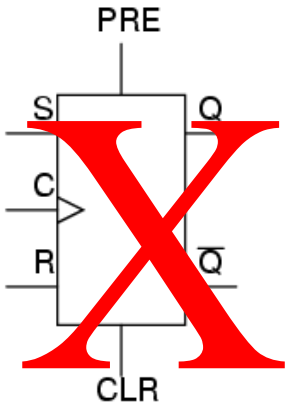


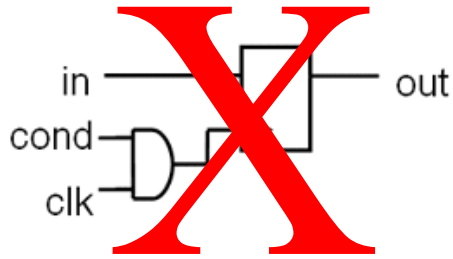

**If clk is high and cond glitches, you get extra edges!**

# Design rules

1. Thou shalt not use asynchronous resets

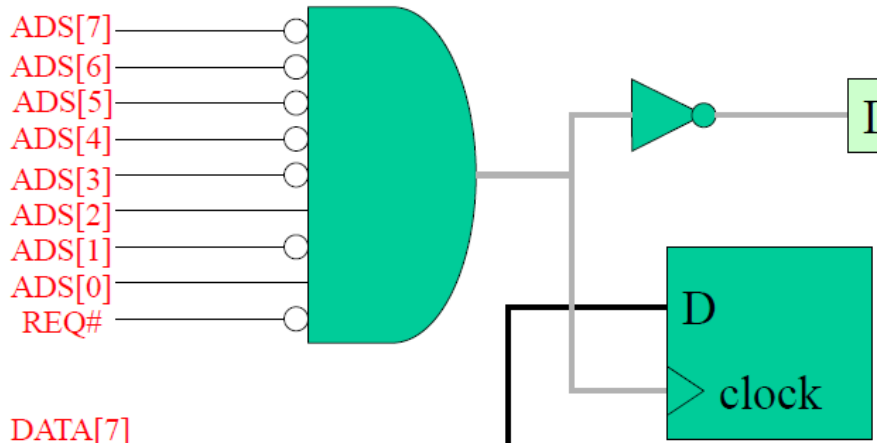
2. Thou shalt not drive a clock with anything other than a clock or directly off of a flip-flop's output



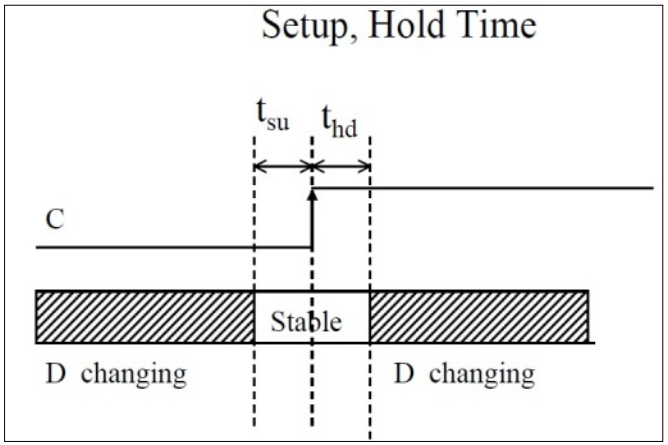



# Really? Seriously?

- 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

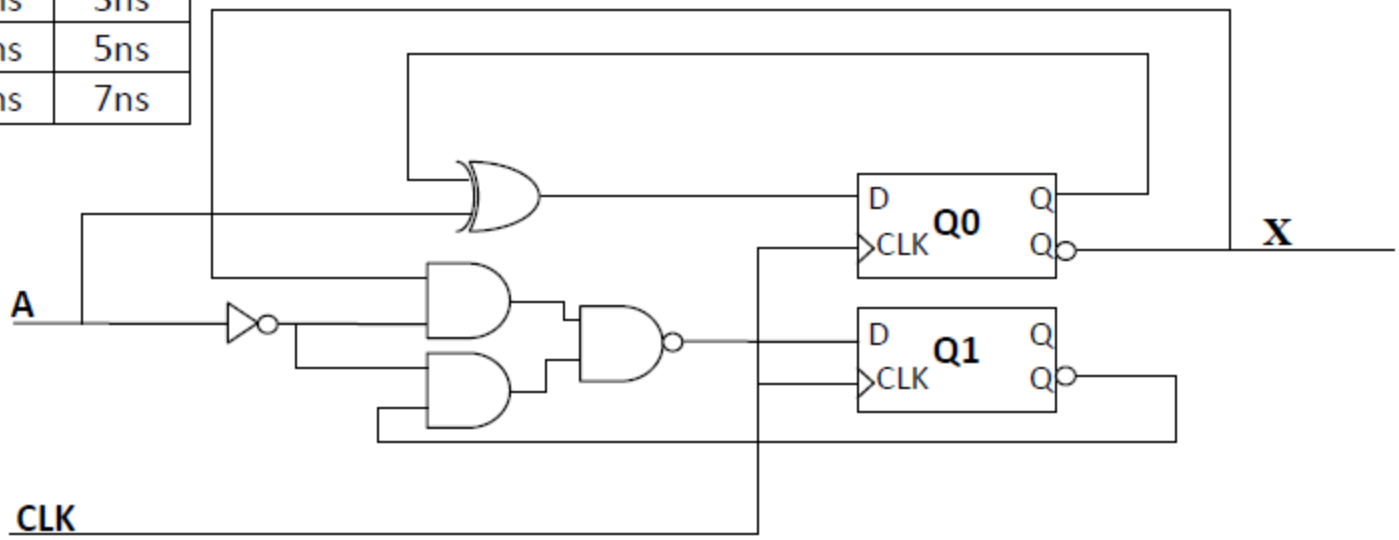



- The idea is simple.
  - When the clock is changing if the data is also changing it is hard to tell what the data ***is***.
    - Hardware can't always tell
      - And you can get meta-stable behavior too (very unlikely but…)
  - So we have a "guard band" around the clock rising time during which we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

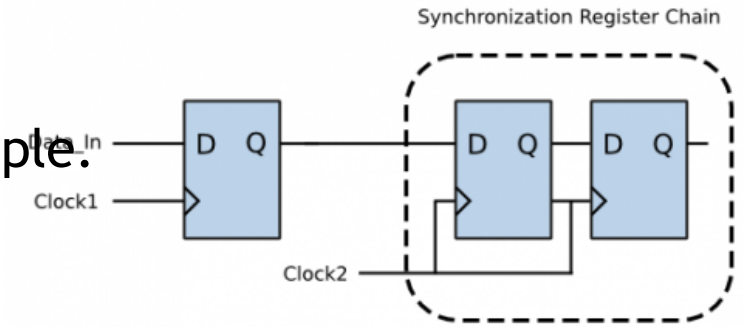
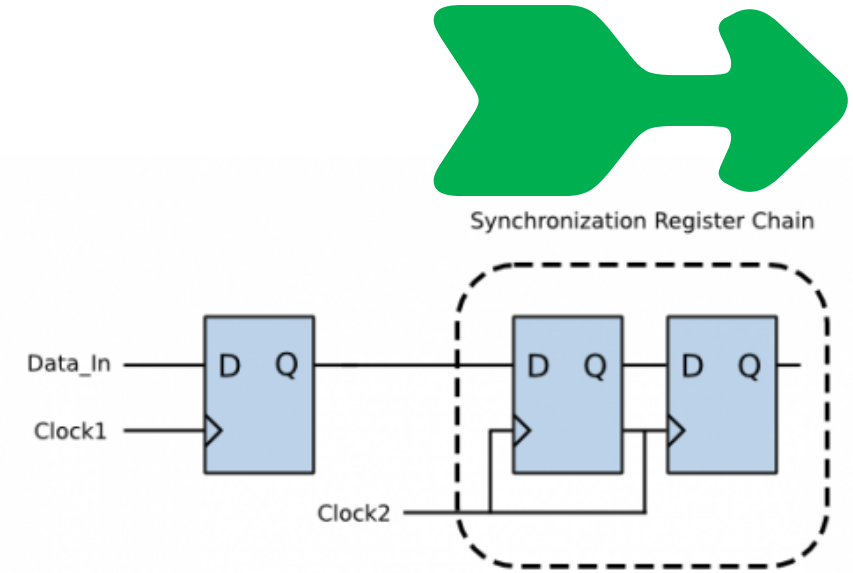



Figure from http://www.eeweb.com/electronics-quiz/solving-metastability-design-issues, we use the same thing to deal with external inputs too!

# 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};
```