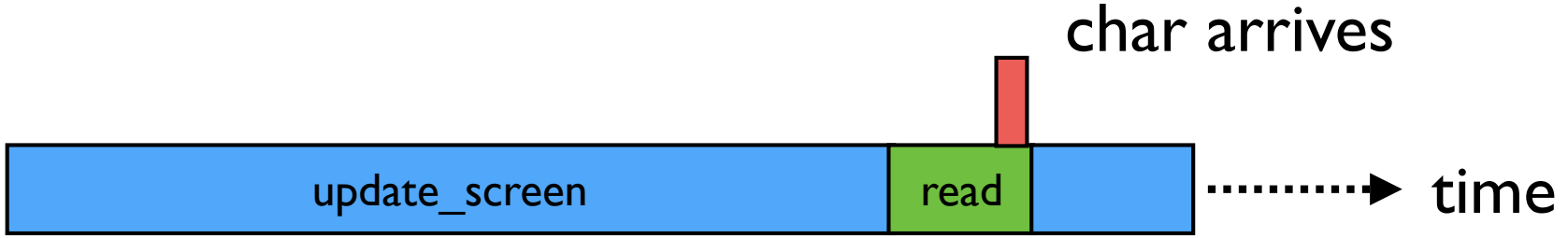
Interrupts and Concurrency

Blocking I/O

```
while (1) {
  read_char_to_screen();
  update_screen();
}
```

`read_char_to_screen` loops until char is received
call that a blocking read



Blocking I/O

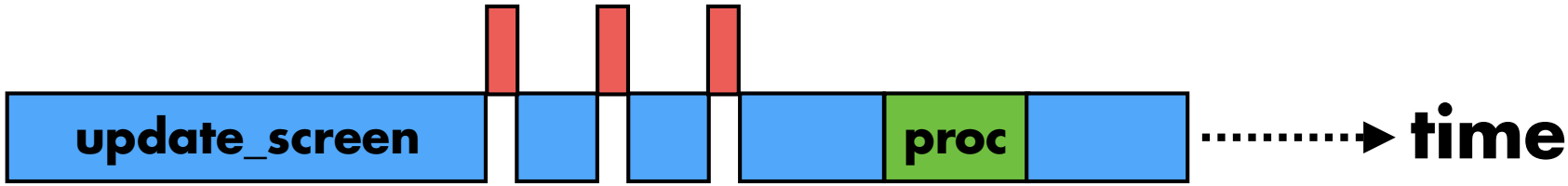
```
while (1) {
  read_char_to_screen();
  update_screen();
}
```

glkeyboard

Concurrency

```
when a scan code arrives {
  add_scan_code_to_buffer();
}

while (1) {
  // Doesn't block
  while (read_chars_to_screen()) {}
  update_screen();
}
```



Interrupts to the Rescue

Cause processor to pause what it's doing and immediately execute interrupt code, returning to original code when done.

- **External events (reset, timer, GPIO)**
- **Internal events (bad memory access, software trigger)**

Last Lecture

Interrupt vectors are placed at a known location (`0x0`)

- **Processor looks for the interrupt handlers in this table**
- **Table consists of "vectors"; Vector *points* to the interrupt handler code**
- **Have to copy this block of memory to `0x0` in `cstart.c`**

```
_vectors:
    ldr pc, _reset_asm
    ldr pc, _undefined_instruction_asm
    ldr pc, _software_interrupt_asm
    ldr pc, _prefetch_abort_asm
    ldr pc, _data_abort_asm
    ldr pc, _reserved_asm
    ldr pc, _interrupt_asm
    ldd pc, _fast_interrupt_asm

_reset_asm:                     .word reset_asm
_undefined_instruction_asm:     .word undefined_instruction_asm
_software_interrupt_asm:        .word software_interrupt_asm
_prefetch_abort_asm:            .word prefetch_abort_asm
_data_abort_asm:                .word data_abort_asm
_reserved_asm:                  .word reset_asm
_interrupt_asm:                 .word interrupt_asm
_fast_interrupt_asm:            .word fast_interrupt_asm
_vectors_end:

interrupt_asm:
    sub   lr, lr, #4
    push  {lr}
. . .
```

libpi/src/interrupt_asm.s

Interrupt Handler

Interrupt occurs right before instruction

```
Disassembly of section .text:

00008000 <_start>:
    8000:       e3a0d902        mov     sp, #32768      ; 0x8000
    8004:       eb000001        bl      8010 <_cstart>

00008008 <hang>:
    8008:       eb000039        bl      80f4 <led_on>
    800c:       eafffffe        b       800c <hang+0x4>

00008010 <_cstart>:
    8010:       e92d4800        push    {fp, lr}
```

What is the pc when the interrupt occurs? Where can we store that information?

Interrupt Handler in ASM

```
interrupt_asm:
  sub   lr, lr, #4          @ Have to subtract 4 from LR
  push  {lr}
  push  {r0-r12}
  mov   r0, lr              @ Pass pc as argument
  bl    interrupt_vector    @ C function
  pop   {r0-r12}
  ldm   sp!, {pc}^          @ Pop LR to PC, restore CPSR
```

`lr` register is stores `pc` where interrupt occurred

Interrupt Handler in ASM

```
interrupt_asm:
  sub   lr, lr, #4          @ Have to subtract 4 from LR
  push  {lr}
  push  {r0-r12}
  mov   r0, lr              @ Pass pc as argument
  bl    interrupt_vector    @ C function
  pop   {r0-r12}
  ldm   sp!, {pc}^          @ Pop LR to PC, restore CPSR
```

Save registers on the stack
Interrupt stack and `sp`

Processor Modes

User - unprivileged mode

IRQ - interrupt mode

FIQ - fast interrupt mode

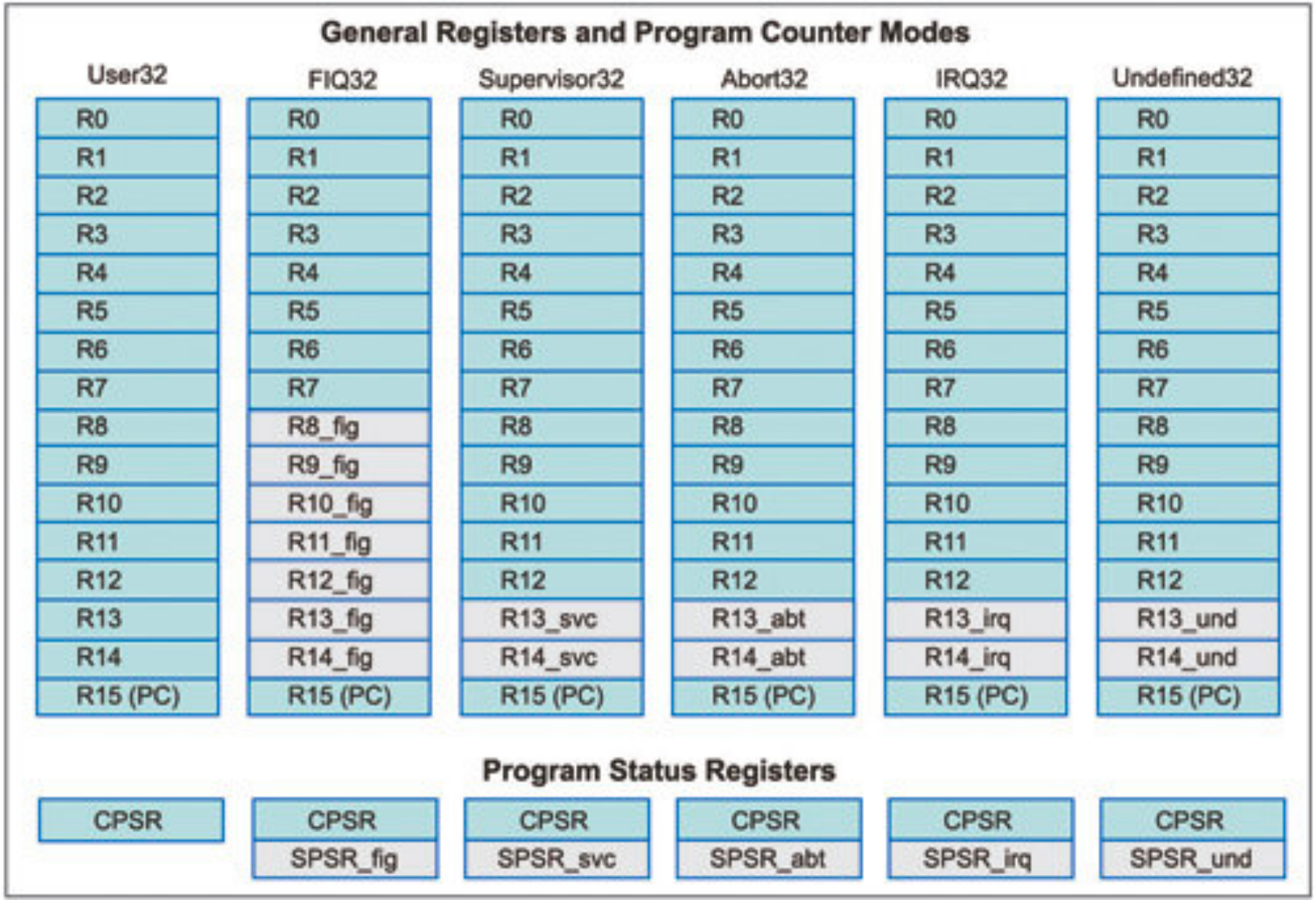
Supervisor - privileged mode, entered on reset

Abort - memory access violation

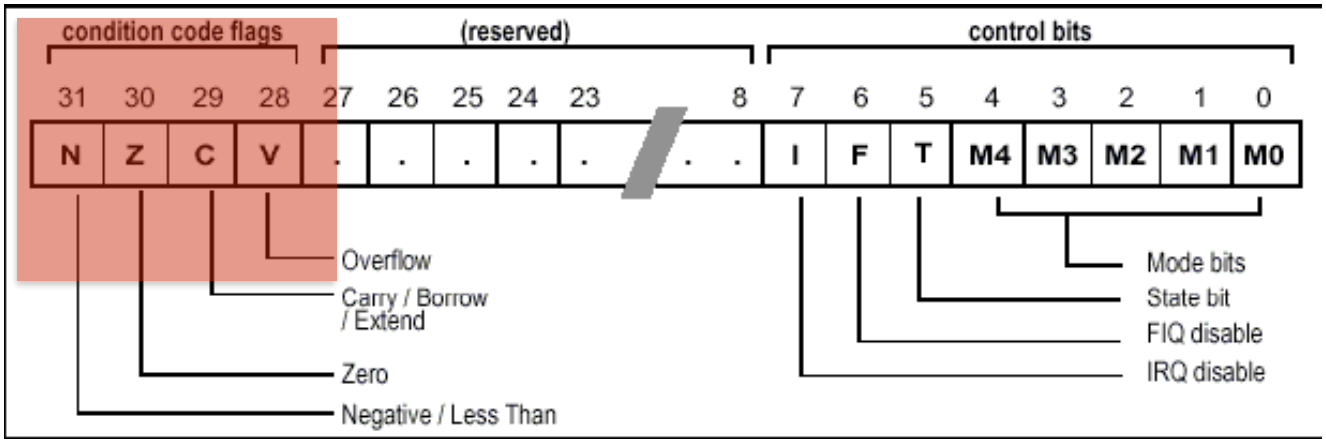
Undefined - undefined instruction

System - privileged mode that shares user regs

Shared / Unshared Registers



CPSR

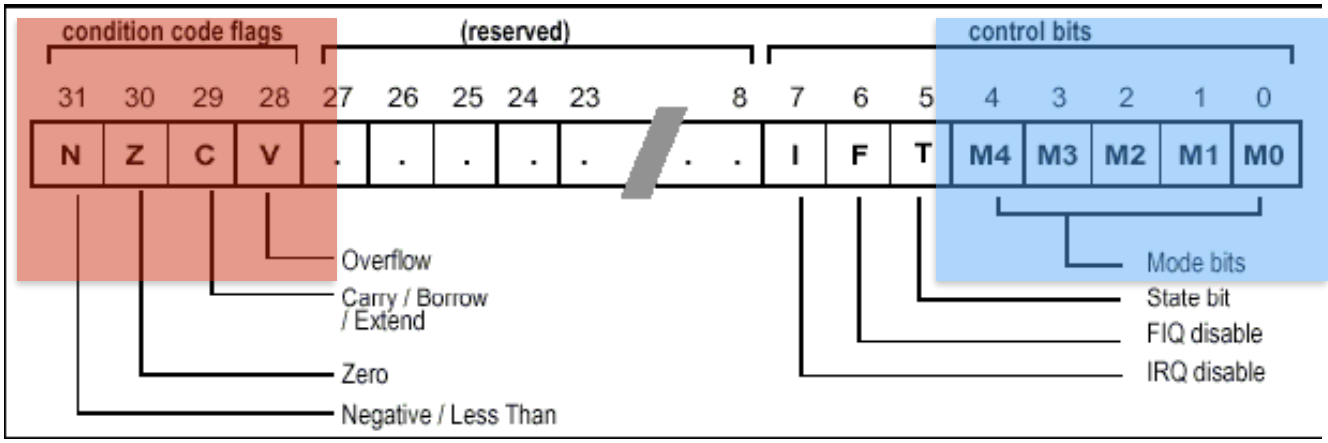


M[4:0]	Mode
b10000	User
b10001	FIQ
b10010	IRQ
b10011	Supervisor
b10111	Abort
b11011	Undefined
b11111	System

```
msr psr, Rm      <- Store Rd into psr
mrs Rd, psr      <- Load Rd with psr

msr cpsr_c, r0   <- Store CPSR with r0
mrs r0, cpsr_c   <- Load r0 with CPSR
```

CPSR

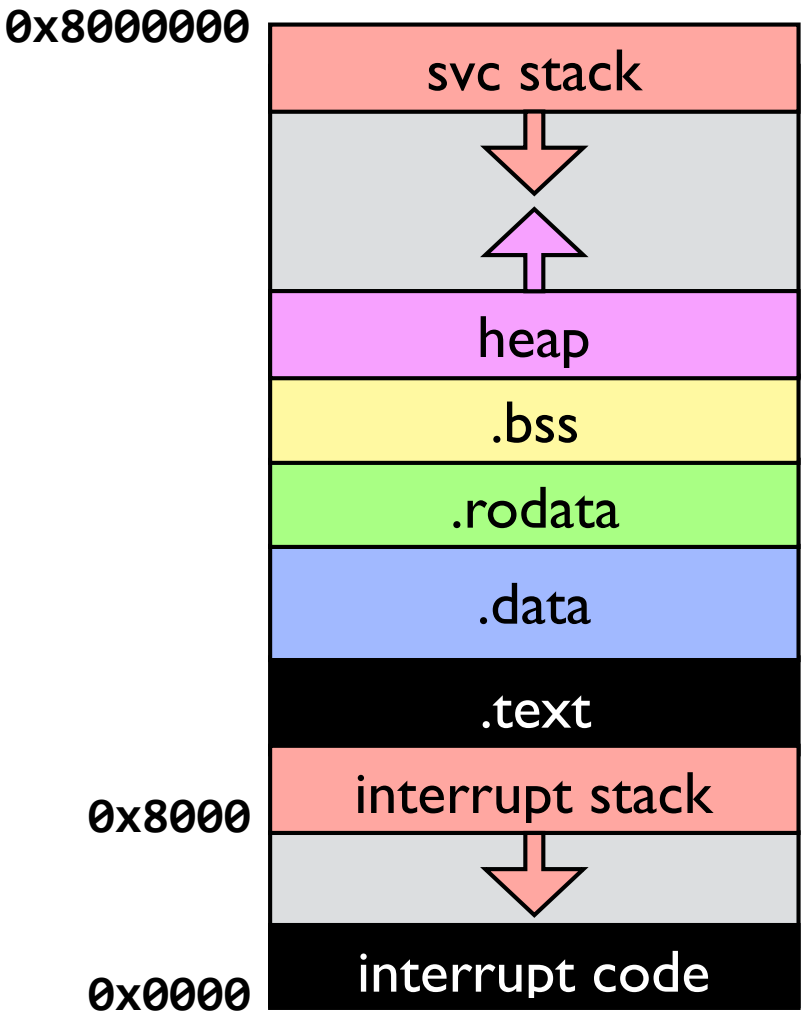


M[4:0]	Mode
b10000	User
b10001	FIQ
b10010	IRQ
b10011	Supervisor
b10111	Abort
b11011	Undefined
b11111	System

```
msr psr, Rm      <- Store Rd into psr
mrs Rd, psr      <- Load Rd with psr

msr cpsr_c, r0   <- Store CPSR with r0
mrs r0, cpsr_c   <- Load r0 with CPSR
```

Set up Interrupt Stack



start.s

```
_start:
    mov r0, #0xD2  // Interrupts
    msr cpsr_c, r0
    mov sp, #0x8000
    mov r0, #0xD1  // Fast interrupts
    msr cpsr_c, r0
    mov sp, #0x4000
    mov fp, #0     // FIQ has fp
    mov r0, #0xD3  // Supervisor
    msr cpsr_c, r0
    mov sp, #0x8000000
    mov fp, #0
    bl _cstart
```

Interrupt Handler in ASM

```
interrupt_asm:
  sub   lr, lr, #4          @ Have to subtract 4 from LR
  push  {lr}
  push  {r0-r12}
  mov   r0, lr              @ Pass pc as argument
  bl    interrupt_vector    @ C function
  pop   {r0-r12}
  ldm   sp!, {pc}^          @ Pop LR to PC, restore CPSR
```

Return from interrupt

What is

```
ldm sp!, {pc}^
```

Enabling Interrupts

Three Layers

1. Enable/disable a specific interrupt source
   - For example, when we detect a falling clock edge on GPIO_PIN23 (PS/2 CLK)
2. Enable/disable type of interrupts
   - E.g., GPIO interrupts
3. Global interrupt enable/disable

Interrupt fires if and only all three are enabled

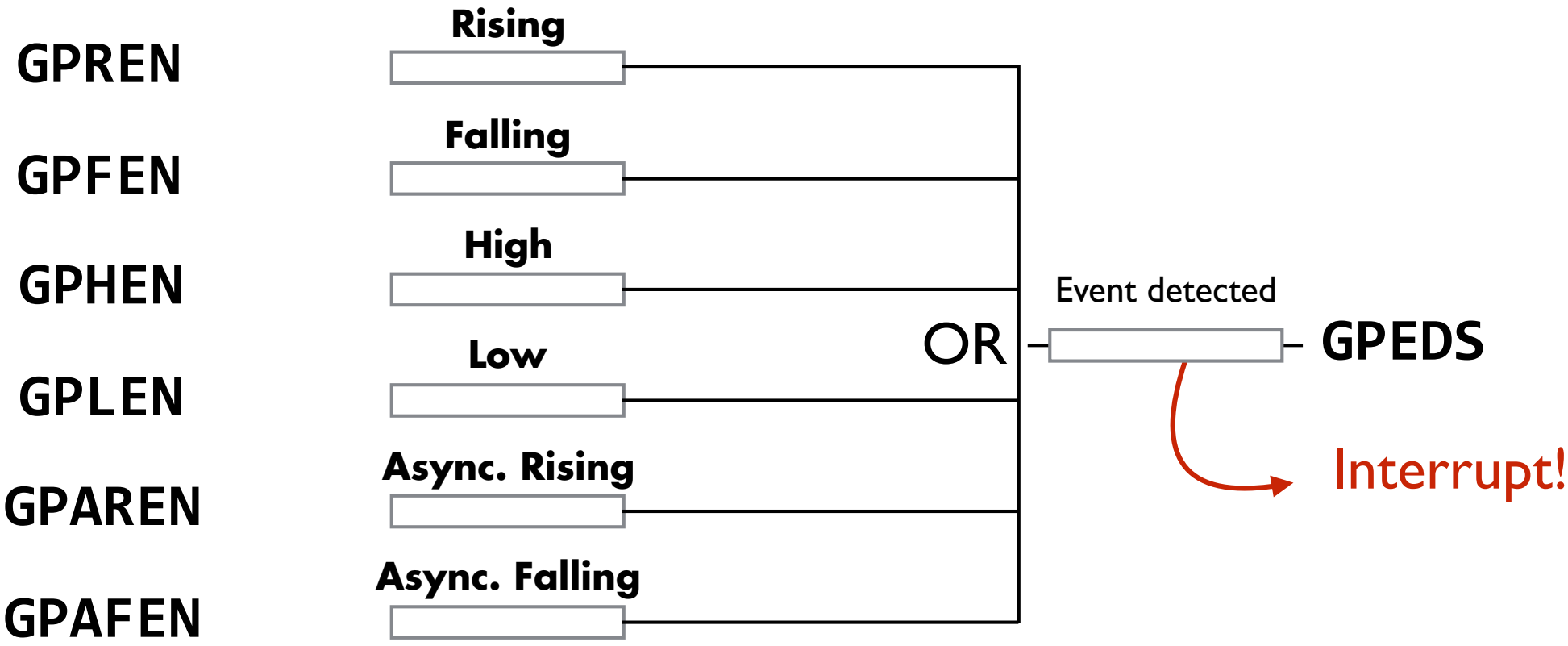
Forgetting to enable one is a common bug

armtimer/blink.c

```
void wait_for_falling_clock_edge() {
  while(gpio_read(CLK) == 0) {}
  while(gpio_read(CLK) == 1) {}
}
```

GPIO Events

Peripheral Registers



See `gpioextra.h` and `gpioextra.c`

GPIO Interrupts (pg. 96-98)

Goal: Trigger interrupt on falling edge of clock, read data line in interrupt handler.

Falling edge detect enable register (GPFENn)

- **Lots of other options! High level, low level, rising edge, etc.**

Event detect status register (GPEDSn)

- **Bit is set when an event on the given pin occurs**
- **Clear event by writing 1 to position, or will re-trigger an interrupt!**

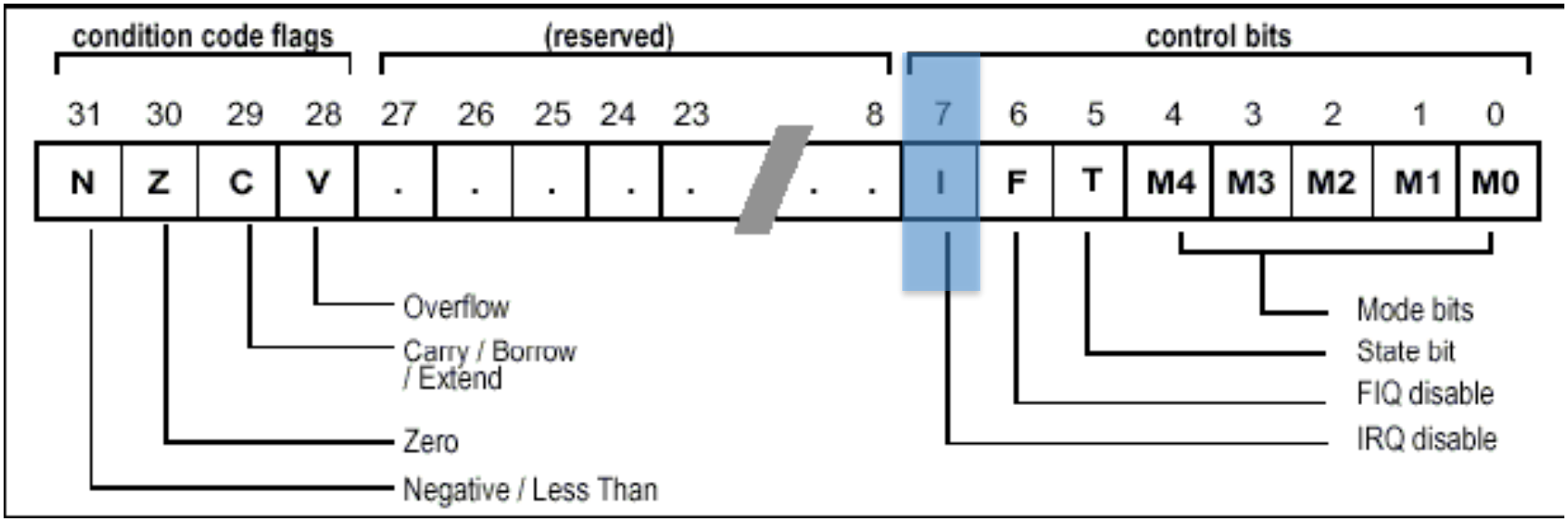
BCM2835, Sec 7.5

#	IRQ 0-15	#	IRQ 16-31	#	IRQ 32-47	#	IRQ 48-63
0		16		32		48	smi
1		17		33		49	gpio_int[0]
2		18		34		50	gpio_int[1]
3		19		35		51	gpio_int[2]
4		20		36		52	gpio_int[3]
5		21		37		53	i2c_int
6		22		38		54	spi_int
7		23		39		55	pcm_int
8		24		40		56	
9		25		41		57	uart_int
10		26		42		58	
11		27		43	i2c_spi_slv_int	59	
12		28		44		60	
13		29	Aux int	45	pwa0	61	
14		30		46	pwa1	62	
15		31		47		63	

From the "internet"

GPIO pin:	4	17	30	31	47
gpio_irq[0] (49)	Y	Y	Y	Y	N
gpio_irq[1] (50)	N	N	Y	Y	N
gpio_irq[2] (51)	N	N	N	N	Y
gpio_irq[3] (52)	Y	Y	Y	Y	Y

Enabling Global Interrupts




```
.global interrupts_global_enable
interrupts_global_enable:
    mrs r0,cpsr
    bic r0,r0,#0x80
    // I=0 enables interrupts
    msr cpsr_c,r0
    bx lr
```

```
.global interrupts_global_disable
interrupts_global_disable:
    mrs r0,cpsr
    orr r0,r0,#0x80
    // I=1 disables interrupts
    msr cpsr_c,r0
    bx lr
```

code/button-interrupts

We’re done!

Not Quite

An interrupt can fire at any time

- Interrupt handler may put a PS/2 scan code in a buffer
- Could do so in the middle of when main() code is trying to pull a scan code out of the buffer
- Need to make sure the interrupt doesn't corrupt the buffer

Need to write code that can be safely interrupted

code/race

One Problem

main code

```
extern int a;

a = a + 1;
```

interrupt

```
extern int a;

a = a - 1;
```

danger

```
00008000 <inc>:
    8000:   e52db004    push    {fp}        ; (str fp, [sp, #-4]!)
    8004:   e28db000    add fp, sp, #0
    8008:   e59f3018    ldr r3, [pc, #24]   ; 8028 <inc+0x28>
    800c:   e5933000    ldr r3, [r3]
    8010:   e2832001    add r2, r3, #1
    8014:   e59f300c    ldr r3, [pc, #12]   ; 8028 <inc+0x28>
    8018:   e5832000    str r2, [r3]
    801c:   e24bd000    sub sp, fp, #0
    8020:   e49db004    pop {fp}        ; (ldr fp, [sp], #4)
    8024:   e12fff1e    bx  lr
    8028:   00010070    .word   0x00010070
```

```
0000802c <dec>:
    802c:   e52db004    push    {fp}        ; (str fp, [sp, #-4]!)
    8030:   e28db000    add fp, sp, #0
    8034:   e59f3018    ldr r3, [pc, #24]   ; 8054 <dec+0x28>
    8038:   e5933000    ldr r3, [r3]
    803c:   e2432001    sub r2, r3, #1
    8040:   e59f300c    ldr r3, [pc, #12]   ; 8054 <dec+0x28>
    8044:   e5832000    str r2, [r3]
    8048:   e24bd000    sub sp, fp, #0
    804c:   e49db004    pop {fp}        ; (ldr fp, [sp], #4)
    8050:   e12fff1e    bx  lr
    8054:   00010070    .word   0x00010070
```

If interrupt happens at danger point, inc() will lose the result of dec(): it has copied a into register r3.

Why won't volatile solve this?

Disabling Interrupts

```
main                                    interrupt handler

  interrupts_global_disable();
  a++;                                          a++;
  b++;;                                         b++;
  reenable_interrupts();
```

Preemption and Safety

Very hard, lots of bugs.

You'll learn more in CS110/CS140.

Two simple answers

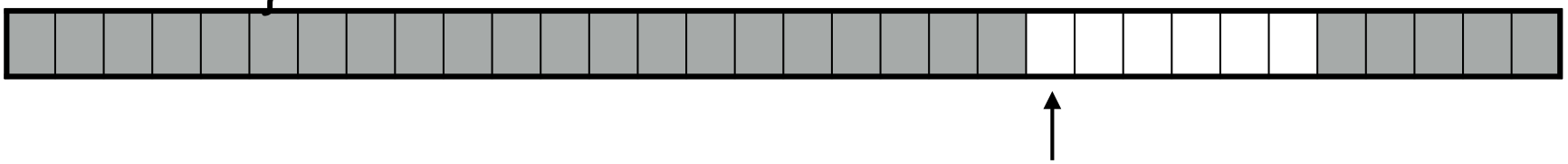
1. **Use simple, safe data structures**
   - write once, but not always possible

2. **Otherwise, temporarily disable interrupts**
   - always works, but easy to forget

Safe Ring Buffer

```
int rb_enqueue(rb_t *rb, int elem) {
    if (rb_full(rb)) {
        return 1;
    } else {
      rb->entries[rb->tail] = elem;
      rb->tail = (rb->tail + 1) % LENGTH; // only writes tail
      return 0;
   }
}
```

```
bool rb_empty(rb_t *rb) {
  return rb->head == rb->tail;
}
```




```
bool rb_dequeue (rb_t *rb, int *elem) {
    if (rb_empty(rb)) return false;
    *elem = rb->entries[rb->head];
    rb->head = (rb->head + 1) % LENGTH; // only writes head
    return true;
}
```

Ringbuffer (rb)

```
void interrupt_handler(void) {
  read_data_bit();
  if (scancode_complete) {
    rb_enqueue(rb, scancode);
  }
}

keyboard_read_scancode(void) {
  while (rb empty(rb) {}
  if( !rb_empty(rb) )
    rb_dequeue(rb, &scancode);
}
```

This Lecture

Writing the code that runs in interrupts

- **Assembly code needed to change to processor models and special registers**
- **Interrupt table copied to `0x0` in `cstart.c`**

Setting up the CPU to issue interrupts

- **3 levels: cause, type, global**

Writing code that can be safely interrupted

- **Race conditions though interrupt-safe ring buffer**

Summary

Interrupts allow external events to preempt what's executing and run code immediately

- **Needed for responsiveness, e.g., do not missing PS/2 scan codes from keyboard when drawing**

Simple goal, but working correctly is very tricky!

- **Deals with many of the hardest issues in systems**

Assignment 7: update keyboard to use interrupts

- **Each clock edge is an interrupt; after 11 interrupts, push scan code into a ring buffer; keyboard driver pulls scan codes from ring buffer**

code/shell-interrupts