# Introducing I/O

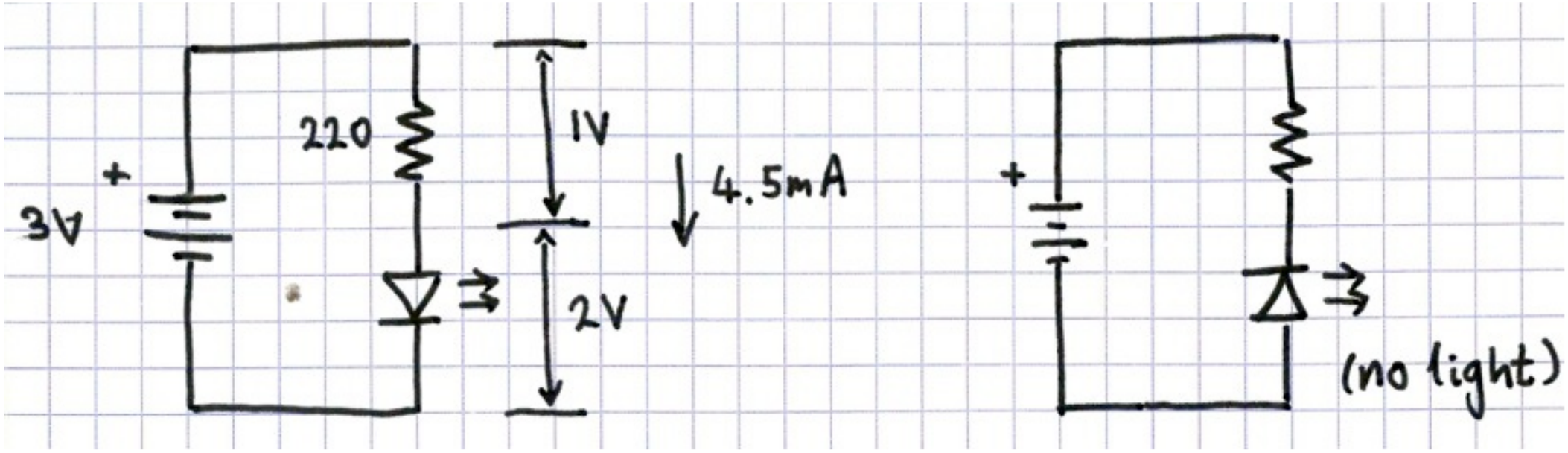
Mike Spivey

Hilary Term 2020

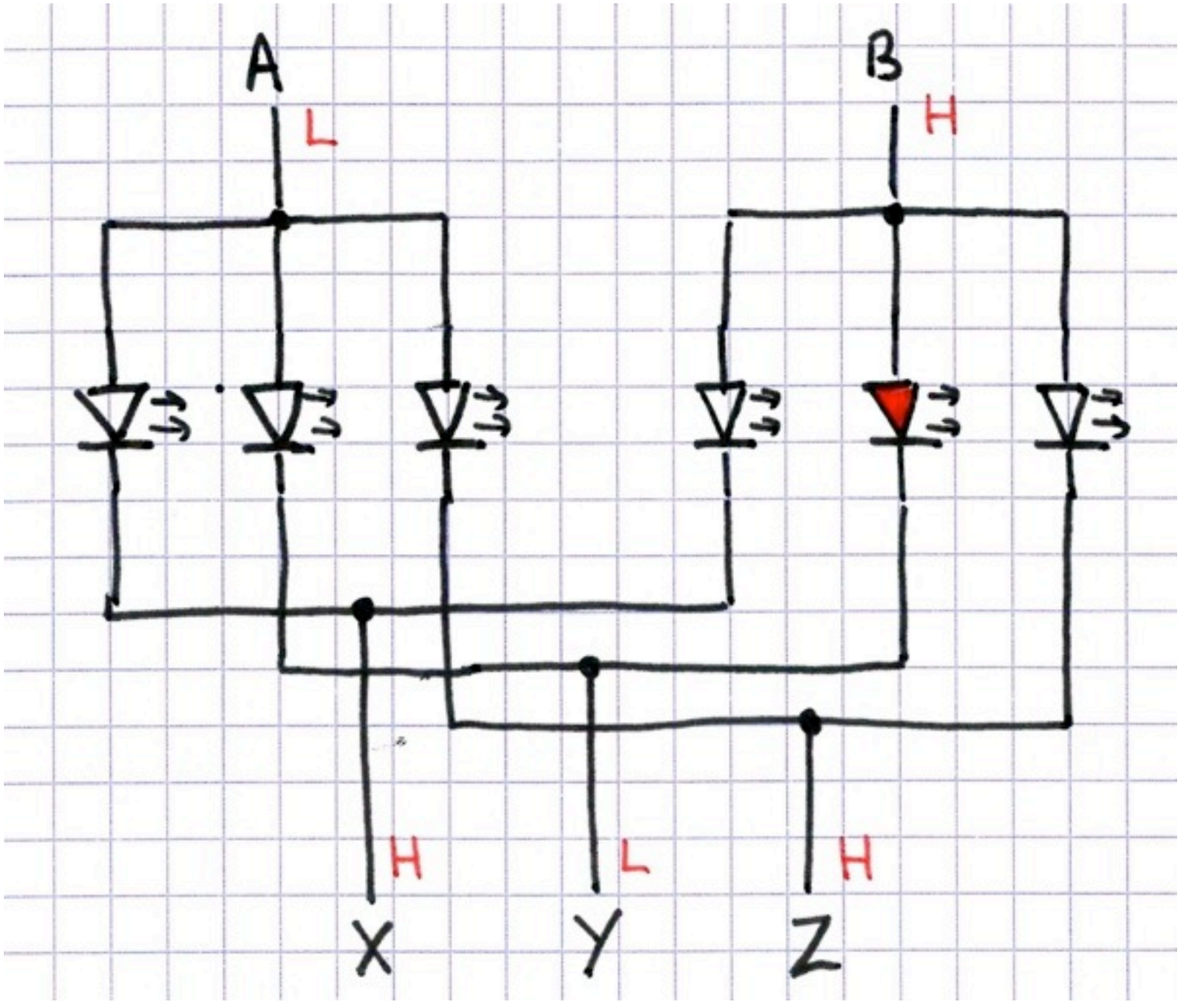

Department of
COMPUTER
SCIENCE

# [8.1] Basics of LEDs

## Forward and reverse bias

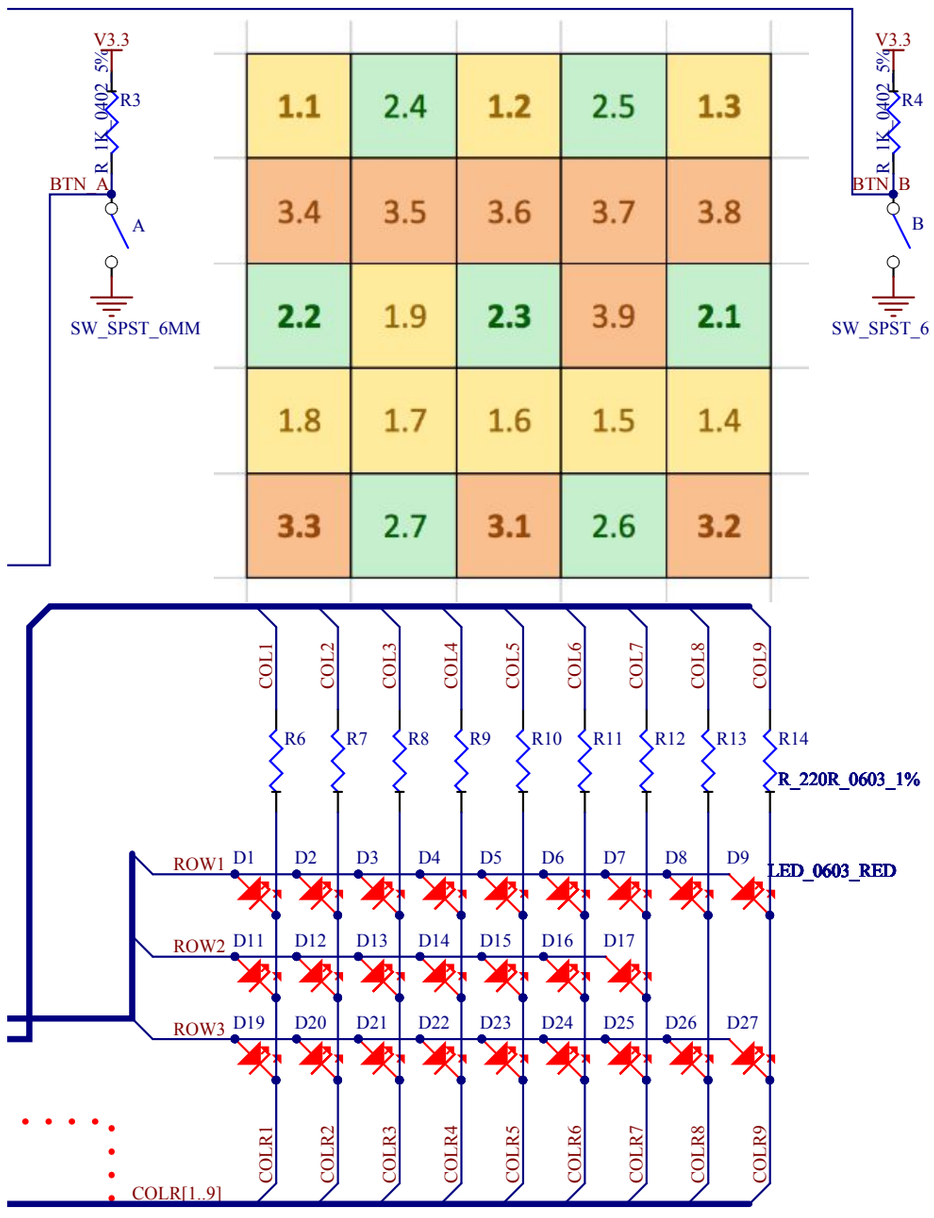


# [8.2] LED multiplexing



# [8.3] LEDs on the micro:bit

- Physically: 5 × 5
- Electrically: 3 × 9 (with 2 gaps)
- Uses 12 bits of GPIO
- Two *active-low* pushbuttons use 2 more GPIO bits



# [8.4] I/O registers

| | | |
|---|---|---|
| `GPIO_DIR` | `0x50000514` | Controls in/out direction |
| `GPIO_OUT` | `0x50000504` | High or low for each pin |

For example:

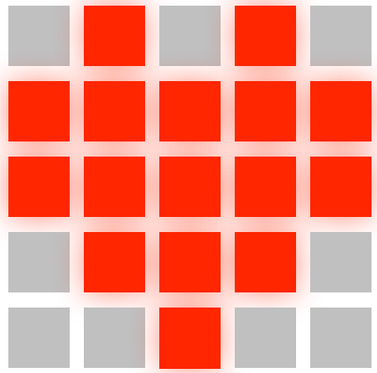
```
ldr r0, =0x50000504
ldr r1, =0x5fb0
str r1, [r0]
```

or in C:

```
#include ”hardware.h”
...
GPIO_OUT = 0x5fb0
```

# [8.5] Multiplexing the display

A pattern like this can be obtained by lighting *successively* ...



LEDs 5, 6, 7, 9 in row 1;
LEDs 1, 2, 3, 4, 5 in row 2;
LEDs 1, 4, 5, 6, 7, 8, 9 in row 3.

```
0010 1000 1111 0000 = 0x28f0
0101 1110 0000 0000 = 0x5e00
1000 0000 0110 0000 = 0x8060
```

# [8.6] Code for multiplexing

```
while (1) {
    GPIO_OUT = 0x28f0;
    delay(JIFFY);
    GPIO_OUT = 0x5e00;
    delay(JIFFY);
    GPIO_OUT = 0x8060;
    delay(JIFFY);
}
```

Use say `JIFFY` = 5000 for 67 updates/sec.

# [8.7] Better: make it data-driven

```
static const unsigned short heart[] = {
    0x28f0, 0x5e00, 0x8060
}

/* frame -- show three rows n times */
void frame(const unsigned short *img, int n) {
    while (n > 0) {
        for (int p = 0; p < 3; p++) {
            GPIO_OUT = img;
            delay(JIFFY);
        }
        n--;
    }
}
```

# [8.8] Implementing `delay()`

```
void delay(unsigned usec) {
    unsigned n = 2 * usec;
    while (n > 0) {
        nop(); nop(); nop();
        n--;
    }
}
```

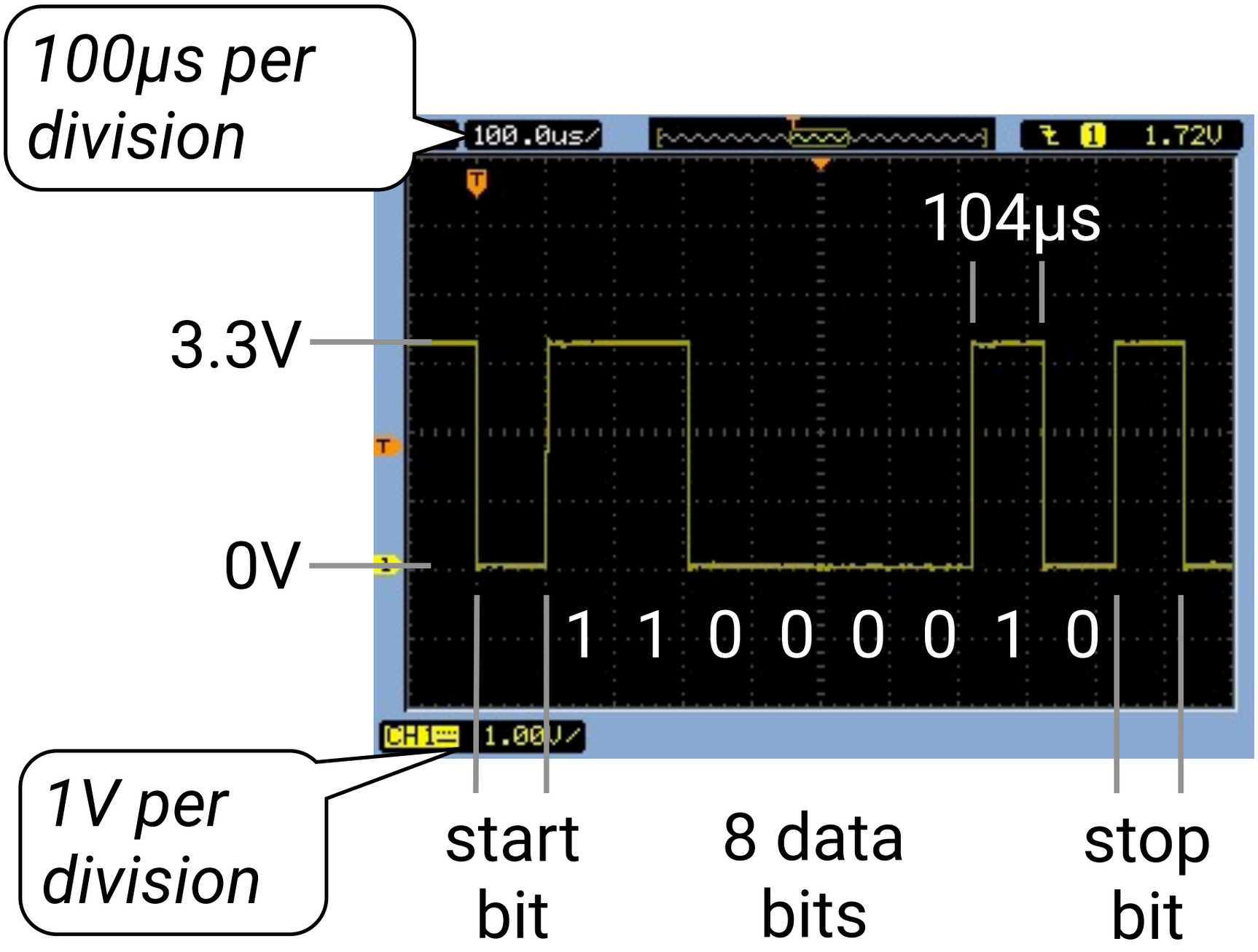
- Experiment shows that each iteration takes 8 cycles = 0.5µs at 16MHz.

# Serial I/O

## Digital Systems – Lecture 9


Department of
COMPUTER
SCIENCE

# [9.1] One character on the serial port



# [9.2] A basic UART driver

```
void serial_putc(char ch) {
    while (! UART_TXDRDY) { /* idle */ }
    UART_TXDRDY = 0;
    UART_TXD = ch;
}
```

- This uses *polling* to wait for the previous character to finish transmitting.
- `serial_printf` is a wrapper around this

(Detail for first character omitted.)

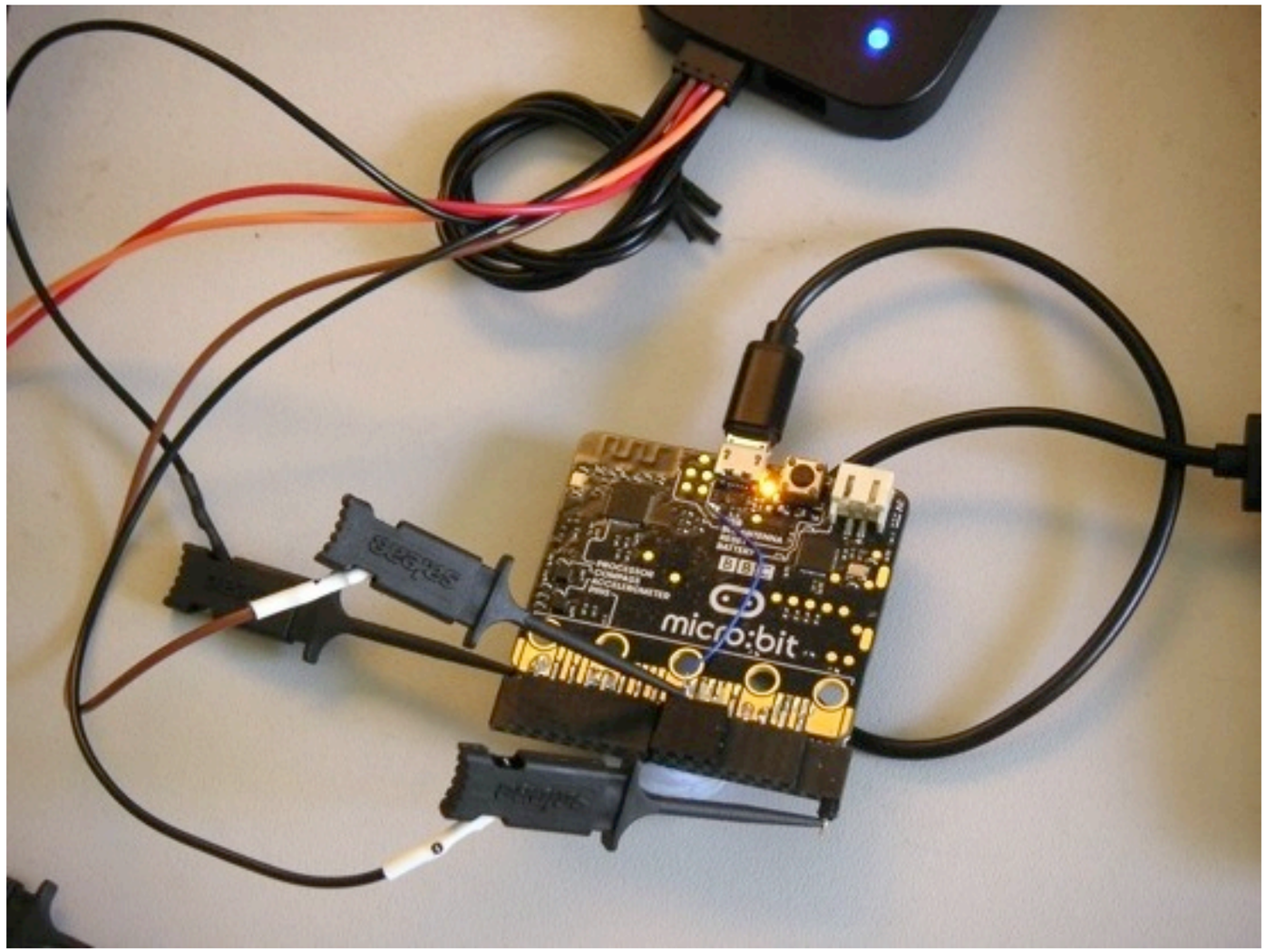
# [9.3] Testing the driver

```
start_timer();
while (count < 500) {
    if (prime(n)) {
        count++;
        serial_printf(“prime(%d) = %d\r\n”,
                      count, n);
    }
    n++;
}
stop_timer();
```

# [9.4] Setting things up

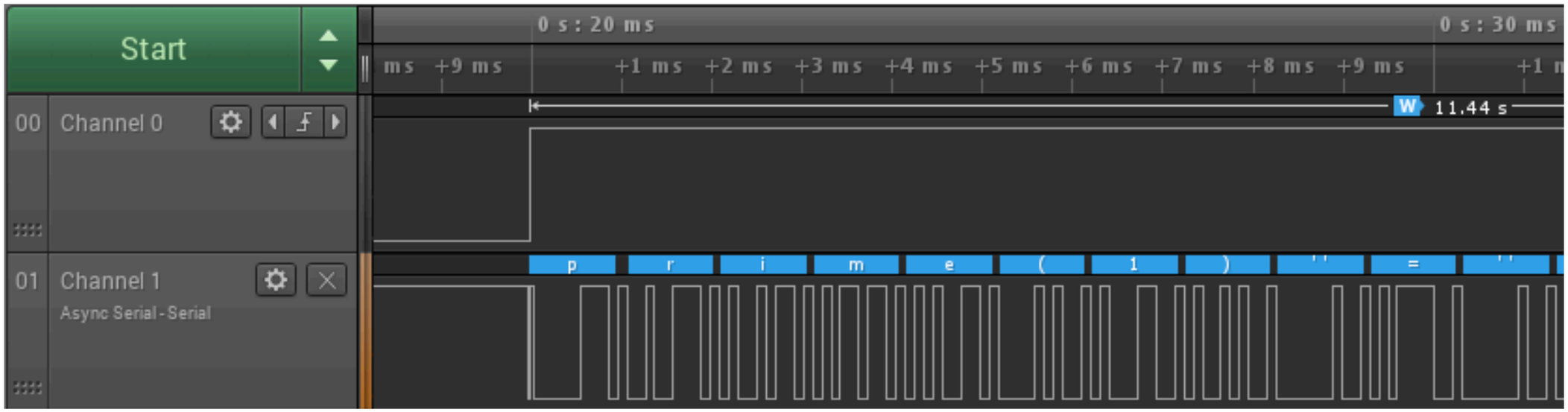
```
void serial_init(void) {
 UART_ENABLE = 0;

 GPIO_DIRSET = BIT(USB_TX);
 GPIO_DIRCLR = BIT(USB_RX);
 SET_FIELD(GPIO_PINCNF[USB_TX], GPIO_PINCNF_PULL, GPIO_Pullup);
 SET_FIELD(GPIO_PINCNF[USB_RX], GPIO_PINCNF_PULL, GPIO_Pullup);

 UART_BAUDRATE = UART_BAUD_9600;      // 9600 baud
 UART_CONFIG = 0;                     // format 8N1
 UART_PSELTXD = USB_TX;               // choose pins
 UART_PSELRXD = USB_RX;
 UART_ENABLE = UART_Enabled;
 UART_STARTTX = 1; UART_STARTRX = 1;
 UART_RXDRDY = 0; UART_TXDRDY = 0;
 txinit = 1;
}
```

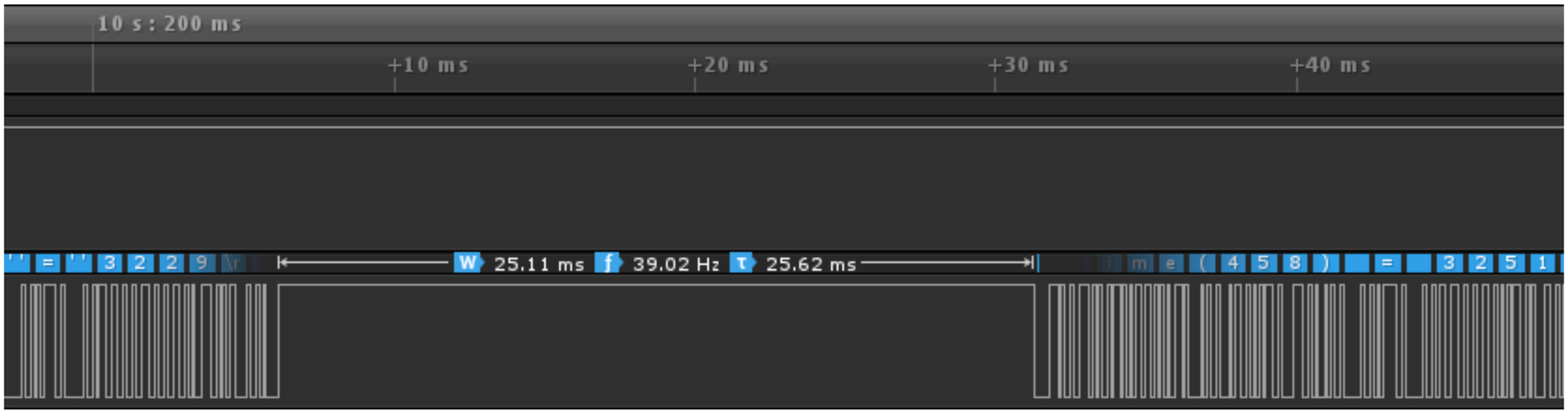
# [9.5] Connecting a logic analyser




Monitoring both an LED pin and the UART

# [9.6] Logic analyser traces



# Programming with interrupts

Digital Systems – Lecture 10


Department of
COMPUTER
SCIENCE

# [10.1] Without interrupts

```
int prime(int n) {
    int k = 2;

    while (k * k <= n) {
        if (n % k == 0) return 0;
        poll_uart();
        k++;
    }

    return 1;
}
```

```
void poll_uart(void) {
    if (UART_TXDRDY) {
        // send another char
    }
}
```

# [10.2] Using interrupts

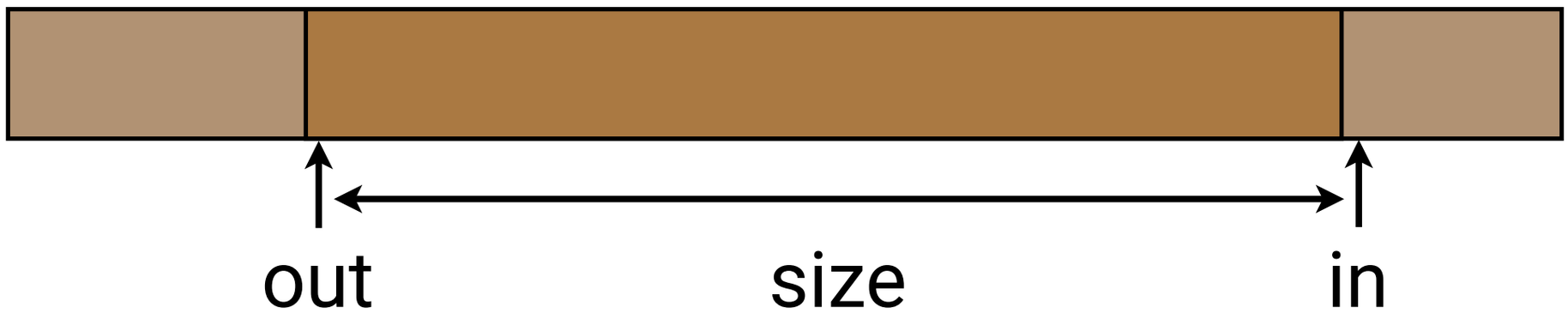
```
int prime(int n) {
    int k = 2;

    while (k * k <= n) {
        if (n % k == 0) return 0;
        poll_uart();
        k++;
    }

    return 1;
}
```

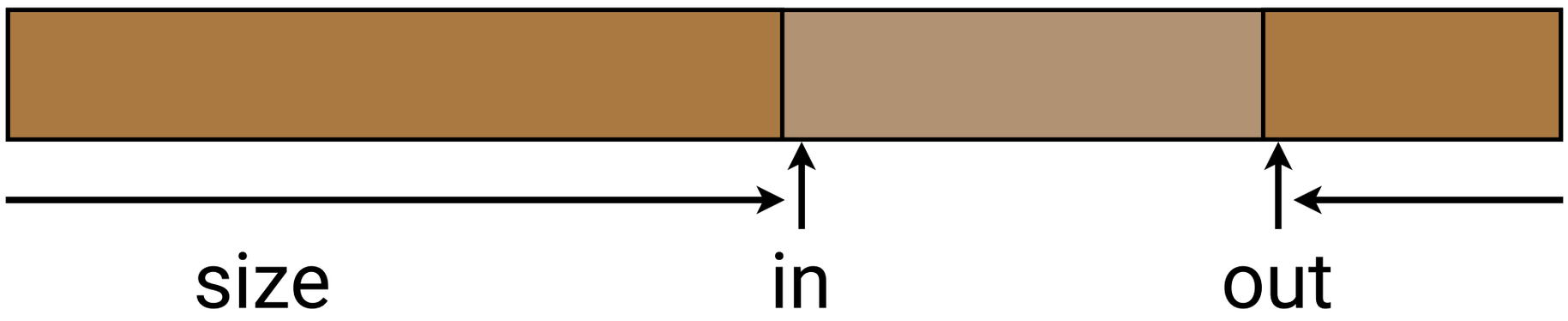
```
void uart_handler(void) {
    if (UART_TXDRDY) {
        // send another char
    }
}
```

# [10.3] A circular buffer

```
#define NBUF 64

static volatile int bufcnt = 0;
static int bufin = 0;
static int bufout = 0;
static volatile char txbuf[NBUF];
static volatile int txidle;
```

# [10.4] Wrapping around



# [10.5] The interrupt handler

```
void uart_handler(void) {
    if (UART_TXDRDY) {
        UART_TXDRDY = 0;
        if (bufcnt == 0)
            txidle = 1;
        else {
            UART_TXD = txbuf[bufout];
            bufcnt--;
            bufout = (bufout+1) % NBUF;
        }
    }
}
```

# [10.6] Rewriting `serial_putc`

```
void serial_putc(char ch) {
    while (bufcnt == NBUF) pause();

    intr_disable();
    if (txidle) {
        UART_TXD = ch;
        txidle = 0;
    } else {
        txbuf[bufin] = ch; bufcnt++;
        bufin = (bufin+1) % NBUF;
    }
    intr_enable();
}
```

# [10.7] Why disable interrupts?

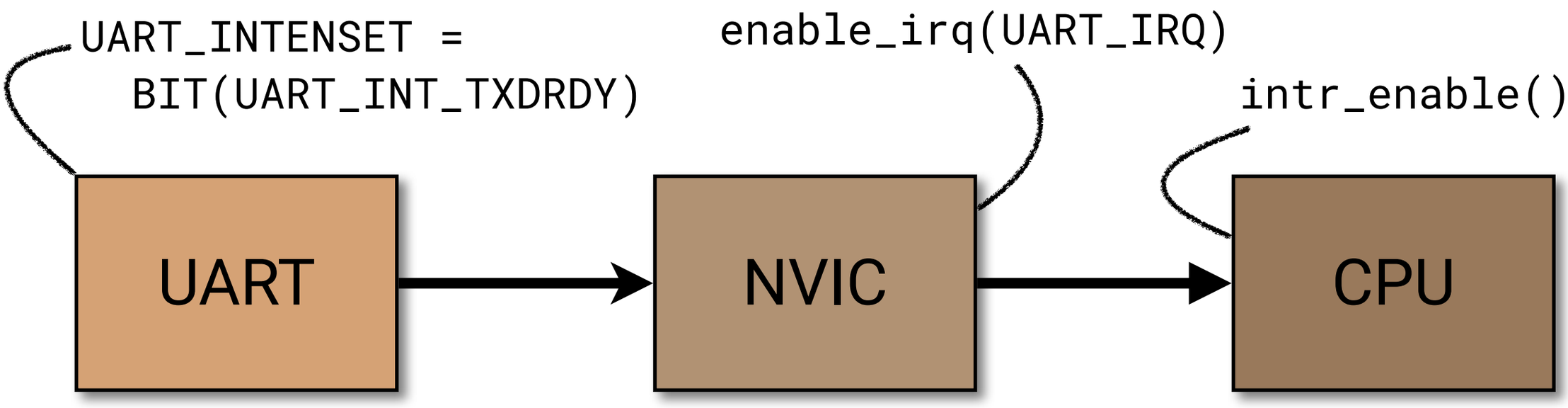
The compiler will implement `bufcnt++` with

```
ldr r0, =bufcnt
ldr r1, [r0]
add r1, r1, #1
str r1, [r0]
```

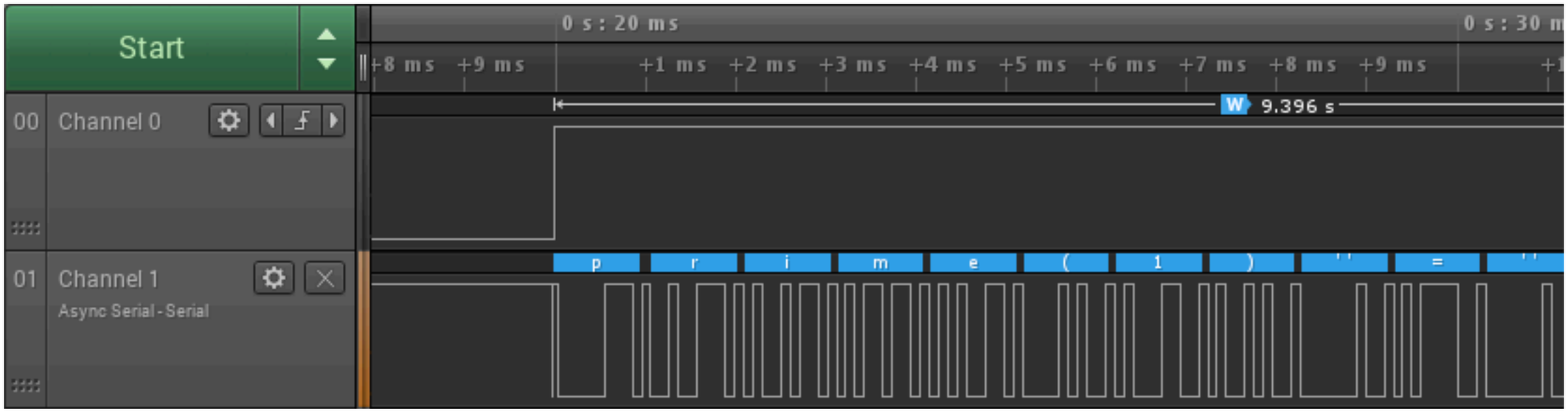
What if an interrupt happens here?

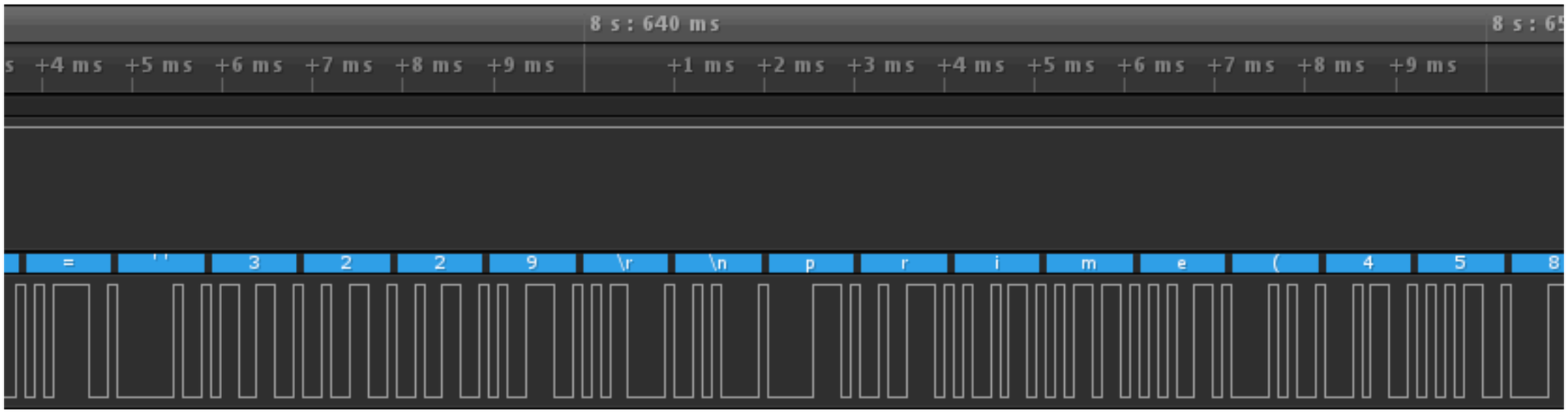
# [10.8] Setting up the interrupt

```
void serial_init(void) {
    ...

    UART_INTENSET = BIT(UART_INT_TXDRDY);
    enable_irq(UART_IRQ);
    txidle = 1;
}
```

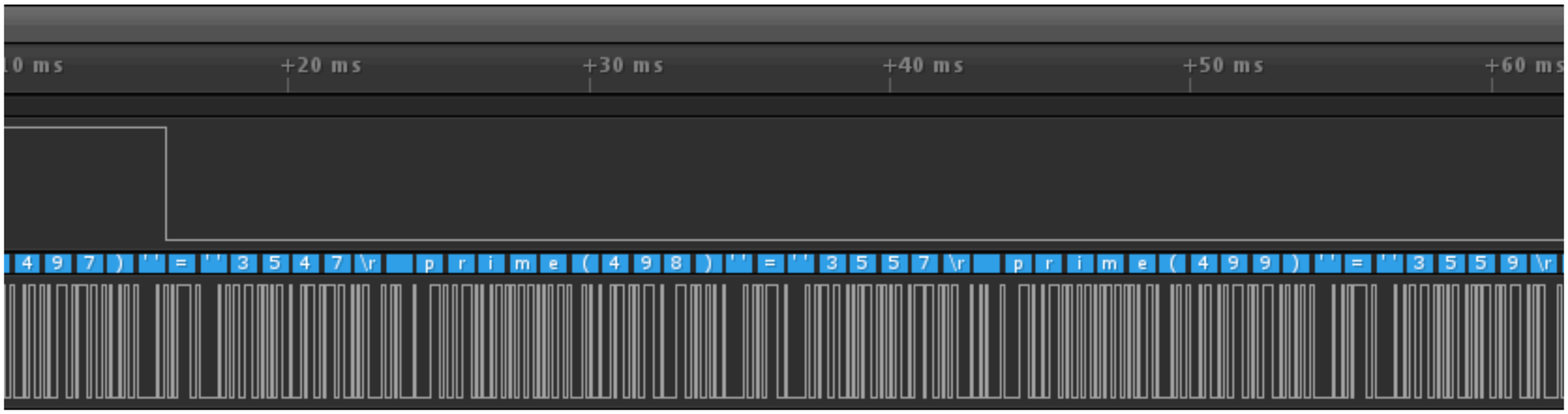


# [10.9] Updated results



# [10.10] And a surprise



# The interrupt mechanism

Digital Systems – Lecture 11


Department of
COMPUTER
SCIENCE

# [11.1] Interrupt mechanism

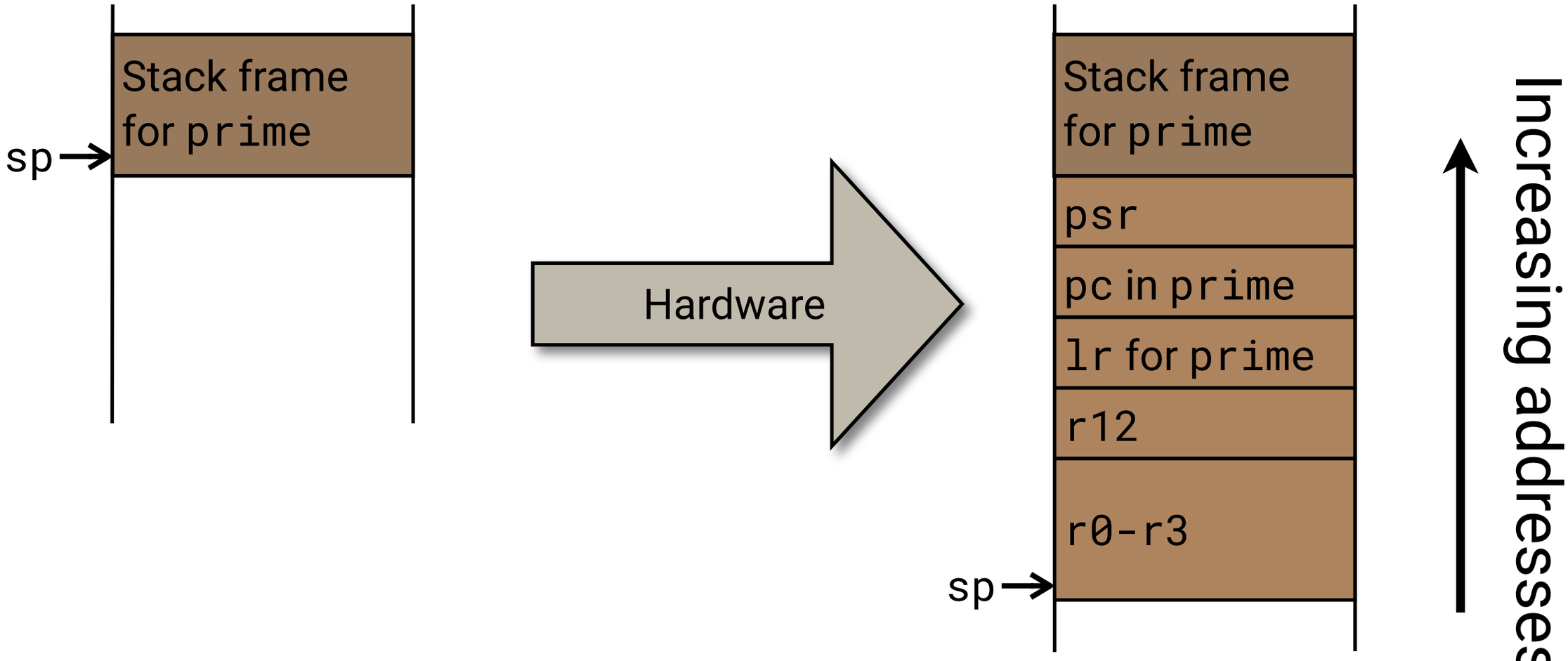
```
int prime(int n) {
    int k = 2;

    while (k * k <= n) {
        if (n % k == 0) return 0;
        k = ... k+1;
    }

    return 1;
}
```

Interrupts must save the processor state

Interrupt handlers can be ordinary subroutines

```
void uart_handler(void) {
    if (UART_TXDRDY) {
        // send another char
    }
}
```

# [11.2] Interrupt entry



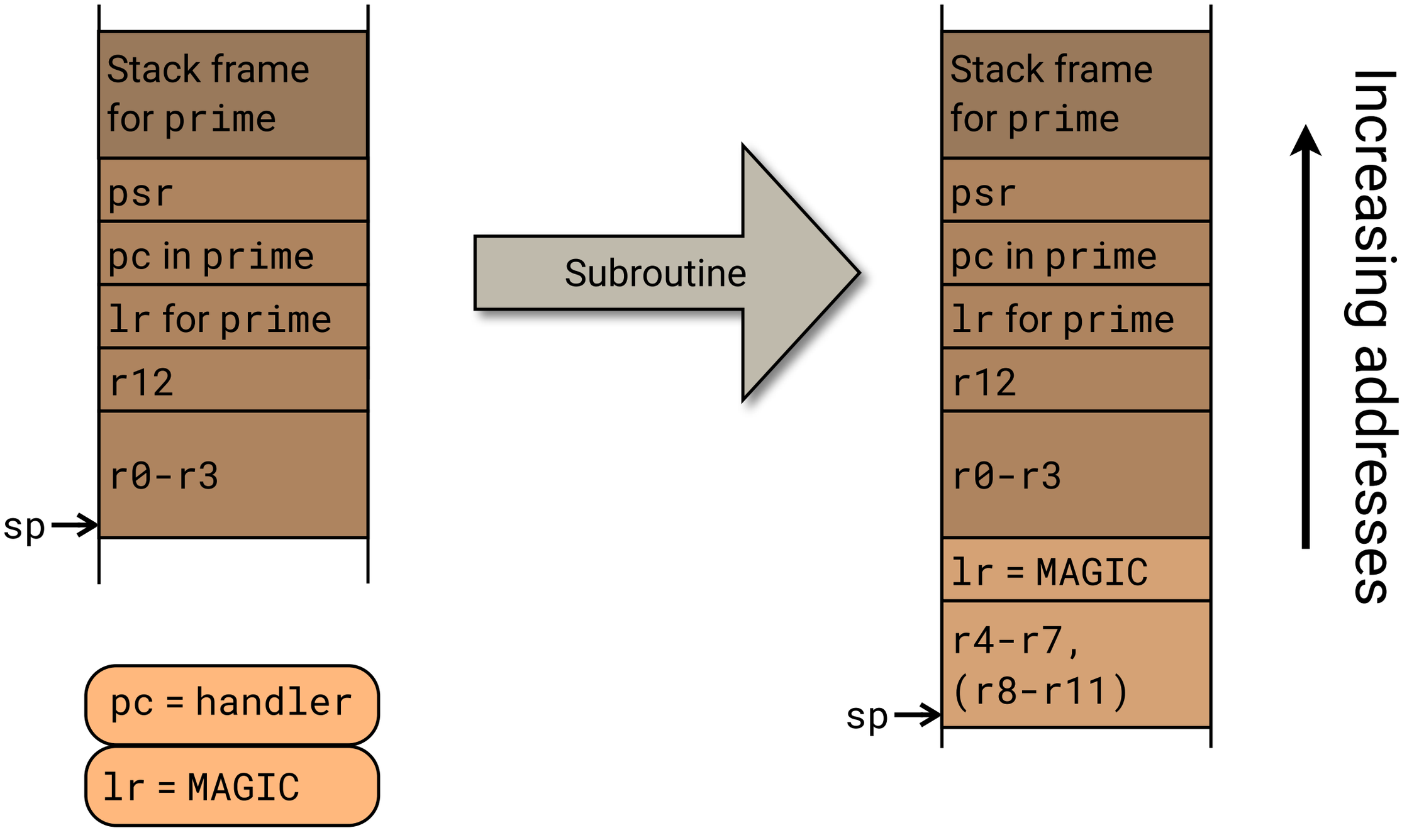

pc in prime

lr for prime

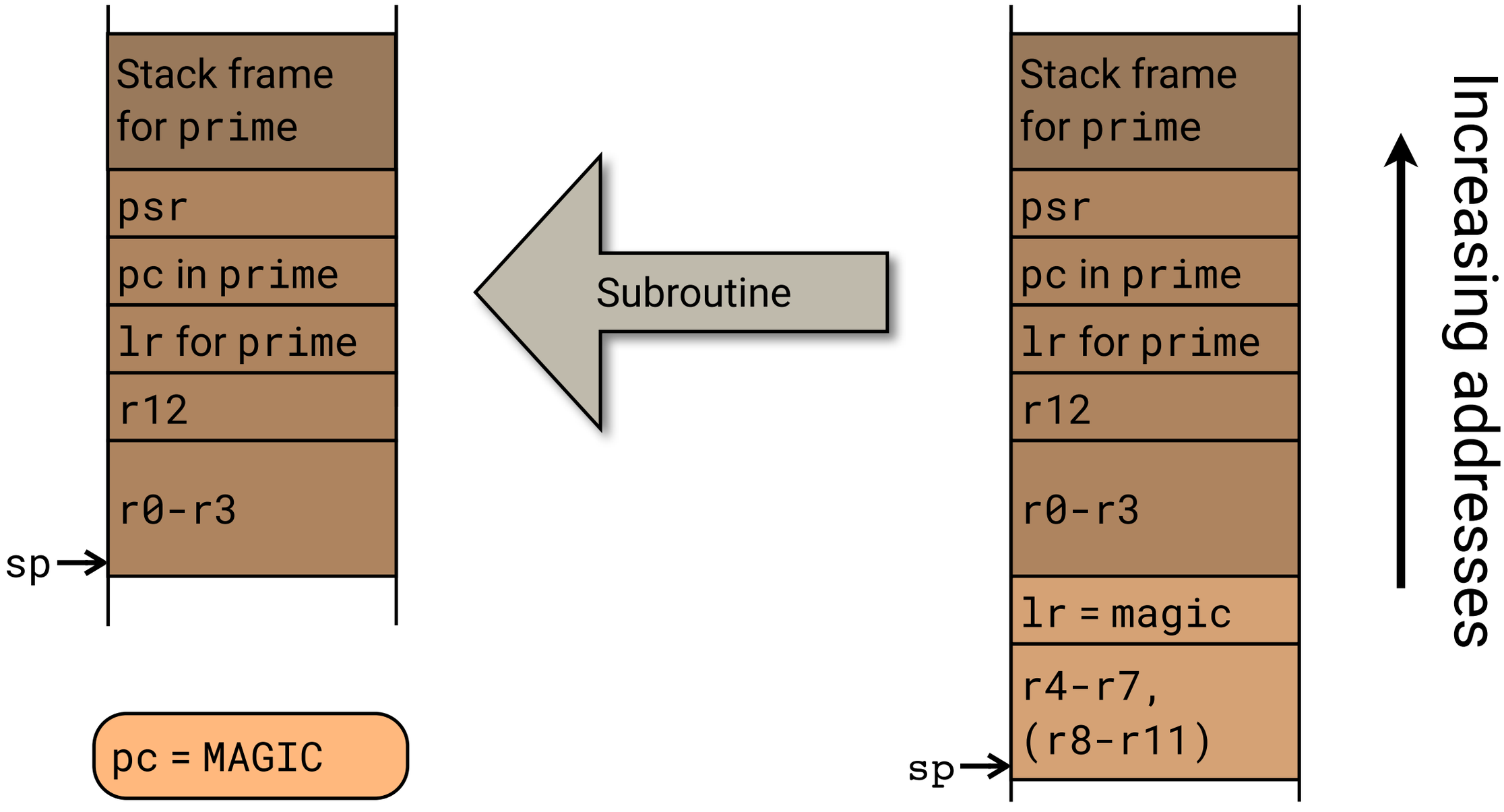
pc = handler

lr = MAGIC

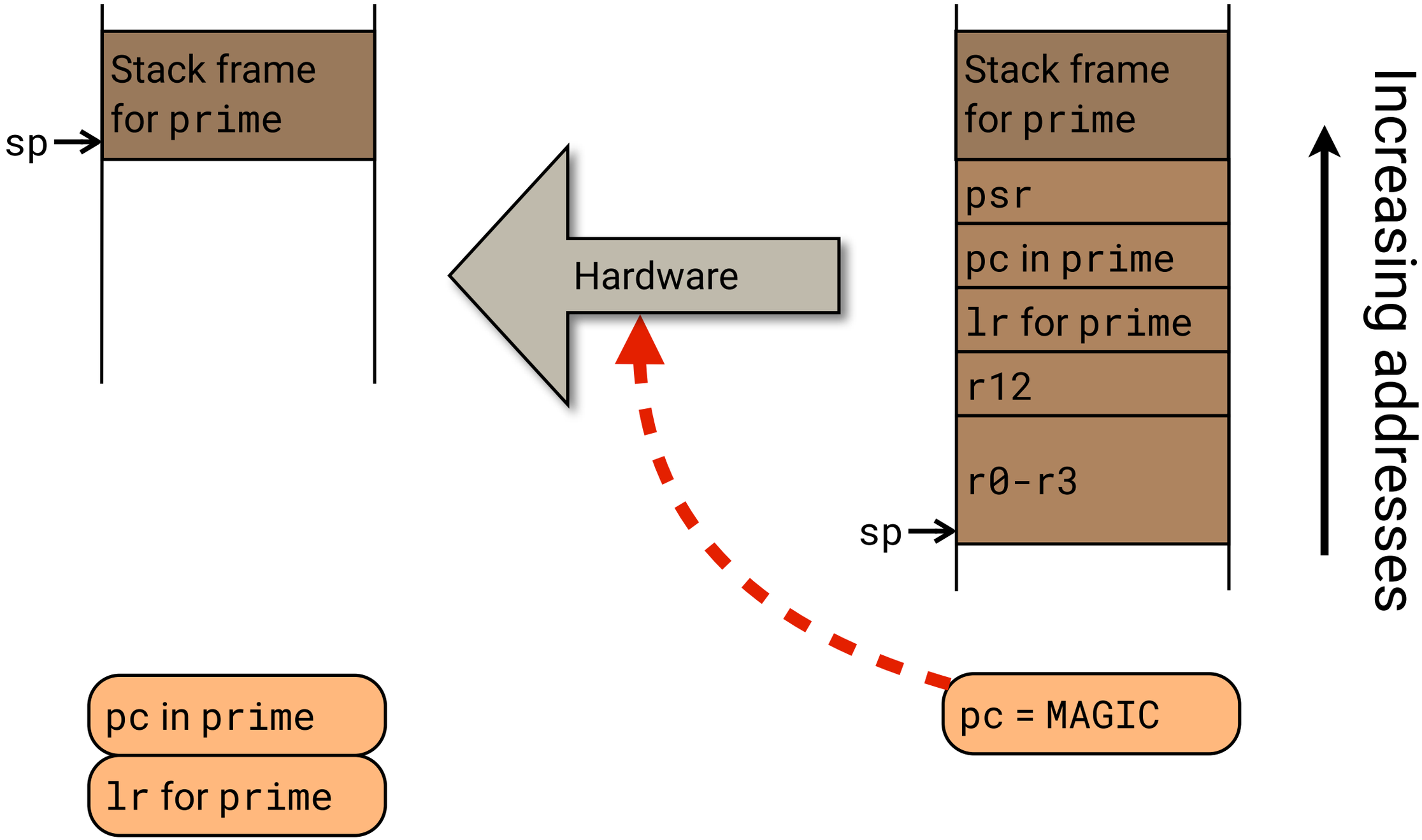
# [11.3] Entering `handler`



# [11.4] Exiting `handler`



# [11.5] Interrupt exit



# [11.6] Scheduling regular actions

Version 0: delay loops (already seen).

- *Wasteful of time and power.*

Version 1: use a timer for delays (this lecture).

- *Still wastes time.*

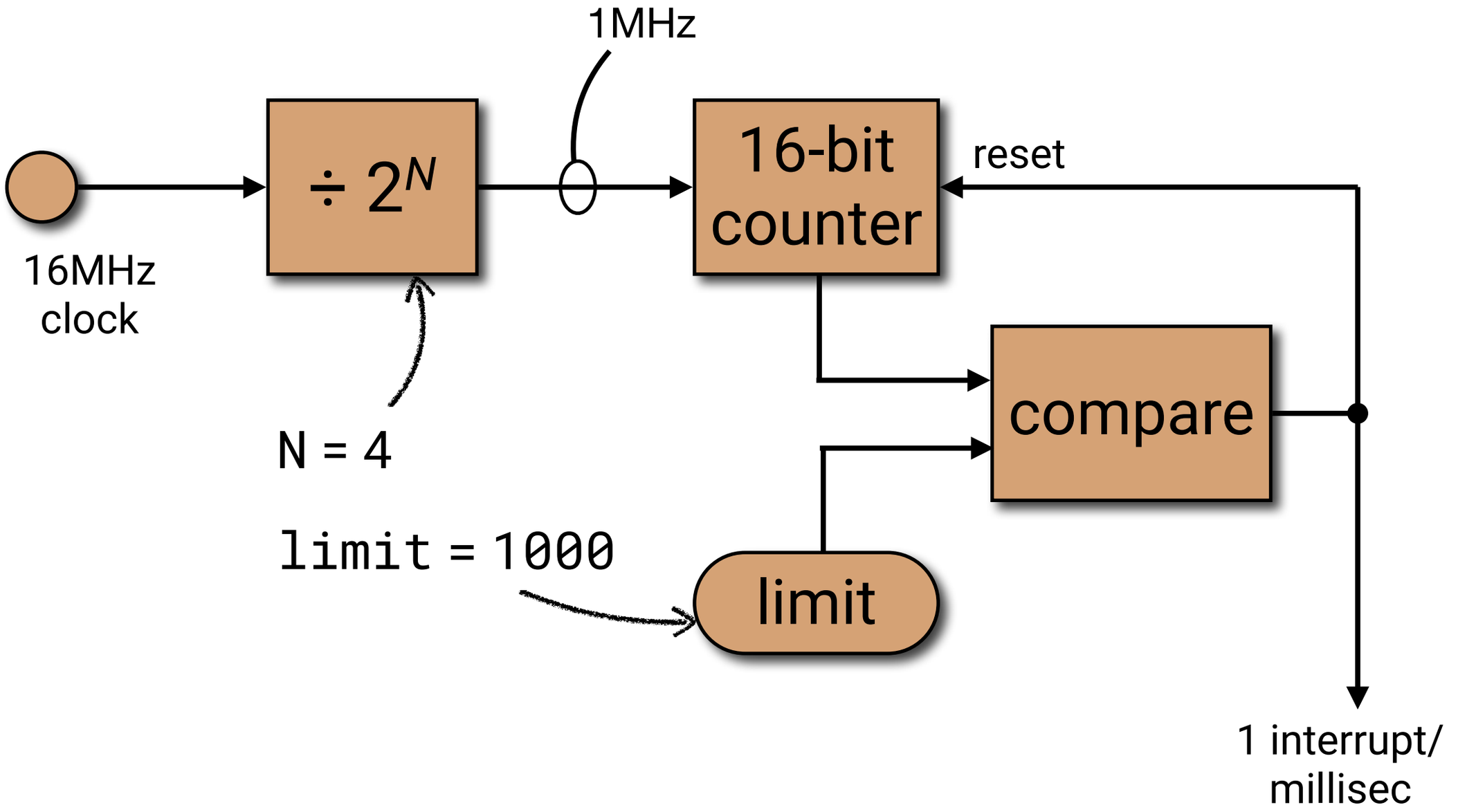
Version 2: purely interrupt driven (this lecture).

- *Efficient but inflexible.*

Version 3: use an operating system (next time).

- *Best of all worlds!*

# [11.7] Timer hardware



# [11.8] Reimplementing `delay()`

```
unsigned volatile millis = 0;

void timer1_handler(void) {
    if (TIMER1_COMPARE[0]) {
        millis++;
        TIMER1_COMPARE[0] = 0;
    }
}

void delay(unsigned usec) {
    unsigned goal = millis + usec/1000;
    while (millis < goal) {
        pause();
    }
}
```

Uses `wfe` instruction

# [11.9] Idea 2: interrupt driven

Make timer_interrupt call this at 5ms intervals:

```
static int row = 0;

void advance(void) {
    row++;
    if (row == 3) row = 0;
    GPIO_OUT = heart[row];
}
```

- No internal control structure allowed.
- Efficient but inflexible.