CS 4250

Computer Architecture

Exam 2

Prof. Nat Tuck
Plymouth State University

Fall 2026
Friday 2026-11-20 • 10:10–11:00 • D&M 417

Allowed: paper notes, pencil, a drink, a non-internet calculator. No phones. Show your work — an answer with no reasoning cannot earn partial credit.

Student Name:

Sample Code

```
/* uart_rx.c -- console UART receive path, polled by rx_task.
 * config:
 * - configTICK_RATE_HZ = 200
 */

#define RX_SIZE 64

static volatile char     rx_buf[RX_SIZE];
static volatile int      rx_count;    /* bytes waiting in rx_buf */
static volatile uint32_t rx_dropped;  /* bytes lost to overflow */
static volatile uint16_t latest_adc;

/* mtvec points here: one entry for every trap on the core.  Bit 63 of
   mcause says interrupt-or-exception, the low bits say which one. */
void trap_handler(void)
{
    unsigned long cause = read_csr(mcause);
    BaseType_t    woken = pdFALSE;

    if (!(cause & (1UL << 63)))     /* bit 63 clear: an exception */
        for (;;)
            ;                       /* this program expects none */

    switch (cause & 0x3f) {
    case 7:                         /* machine timer: the tick */
        latest_adc = adc_read();
        woken = xTaskIncrementTick();
        break;
    case 11:                        /* machine external: the UART */
        uart_isr();
        break;
    ...
    }

    portYIELD_FROM_ISR(woken);
}

void uart_isr(void)
{
    while (uart_status() & UART_RX_READY) {
        char b = uart_read();
        if (rx_count < RX_SIZE) {
            rx_buf[rx_count % RX_SIZE] = b;
            rx_count = rx_count + 1;
        } else {
            rx_dropped = rx_dropped + 1;
```

```
        }
    }
}

static void rx_task(void *arg)
{
    int i, n;

    for (;;) {
        vTaskDelay(pdMS_TO_TICKS(10));

        n = rx_count;
        for (i = 0; i < n; i++) {
            console_putc(rx_buf[i]);
        }
        rx_count = rx_count - n;
    }
}

static void ctrl_task(void *arg)
{
    for (;;) {
        vTaskDelay(pdMS_TO_TICKS(10));
        pwm_set(control_step(latest_adc));   /* takes ~1.5 ms */
    }
}

static void blink_task(void *arg)
{
    for (;;) {
        gpio_toggle(LED);
        vTaskDelay(pdMS_TO_TICKS(100));
    }
}

int main(void)
{
    /* xTaskCreate(pxTaskCode, pcName, usStackDepth, pvParameters, uxPriority, pxCreatedTask) =
    xTaskCreate(ctrl_task,  "ctrl",  1024, 0, 3, 0);
    xTaskCreate(rx_task,    "rx",    1024, 0, 2, 0);
    xTaskCreate(blink_task, "blink",  512, 0, 1, 0);

    uart_rx_irq_on();       /* the UART's own enable bit */
    vTaskStartScheduler();

    for (;;)
        ;
}
```

Part 1: FreeRTOS

Refer to the sample code above to answer the following questions:

Question 1.1: How many milliseconds per tick?

Question 1.2: What is the priority of the blink task? How can you tell?

Question 1.3: There's a data race in the code. Where is it? What problems can it cause? How can we fix it?

Question 1.4: Fill in this table. The first column is time after start. The second column is the thing that triggers a rescheule. The third column is the next task. The fourth column is the LED status.

Time	Event	Next Task	LED
0 ms	first scheduler call	ctrl_task	off
0	delay		off
0	delay		
0	delay		
5	tick		
10	tick		
11.5	delay		
11.5	delay		

Part 2: Pipelining

The machine below is a five-stage pipeline (IF, ID, EX, MEM, WB), one instruction per cycle, with **no forwarding**. A register written in the WB stage can be read in ID in that same cycle or in any later one, because the write happens in the first half of the cycle and the read in the second. A reader therefore has to be **three instructions behind** its writer. Ignore fetch stalls: every instruction is fetched one cycle after the one before it.

Branches are resolved in EX. A wrong prediction throws away the two instructions behind the branch and costs 2 cycles. The predictor is the simple one: **backward branches predicted taken, forward branches predicted not taken.**

```
loop:
        ld    x5, 0(x10)        # I1  node = *p
        ld    x6, 8(x5)         # I2  key  = node->key
        add   x7, x7, x6        # I3  sum += key
        addi  x9, x9, 1         # I4  count++
        ld    x10, 16(x10)      # I5  p = p->next
        bne   x10, x0, loop     # I6  while (p)
```

The schedule for the loop as written:

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	IF	ID	EX	MEM	WB											
I2		IF	ID	ID	ID	EX	MEM	WB								
I3			IF			ID	ID	ID	EX	MEM	WB					
I4				IF					ID	EX	MEM	WB				
I5					IF					ID	EX	MEM	WB			
I6						IF					ID	ID	ID	EX	MEM	WB

Question 2.1: How many stall cycles does one iteration take, and how many cycles does the body take in total? Say which instructions stall, and why.

Question 2.2: Reorder the loop body to remove as many stalls as you can. Write the new loop. If any stalls are left, why?

Question 2.3: Is the defined branch predictor better or worse than a random predictor on this code? Why?

Question 2.4: Would a random predictor be better than no predictor? Why?

Part 3: Other Stuff

Question 3.1: Consider a processor with two cores, where each core has 32kB of L1 data cache and the two cores share 128kB of L2 cache. Two take advantage of the two cores, you add a second thread to your single threaded program and it goes more than twice as fast operating on the same data. How is that possible?

Question 3.2: How does a two-way set associative cache differ from a direct mapped cache? Which is better? Give an example of a memory access sequence where one of these options would have more cache hits than the other - be specific with address bits.