Module 5: Lecture Questions (Partial Solutions)

Module 5 – Lecture 17(Tuesday 4/27)

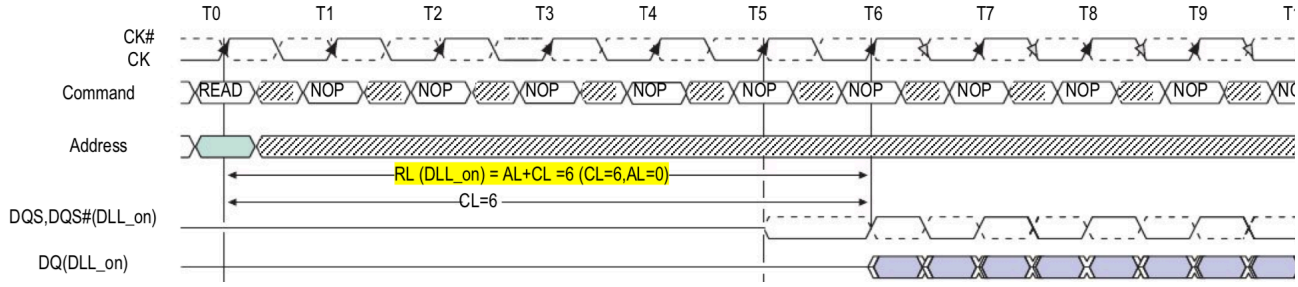
1. How is the TM4C1294 similar to a motherboard? Why can't embedded application simply use motherboards instead of microcontrollers?
2. Name four types of memory that are the TM4C1294 device and what buses the CPU uses to access them.
3. What three types of signals are implemented are implemented in the on-chip bus connections?

Module 5 – Lecture 18(Thursday 4/28)

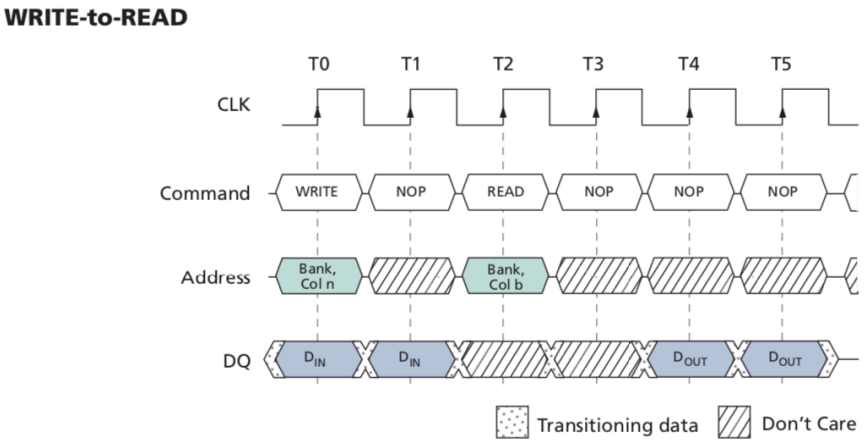
4. Many systems employ bridges to access lower performance interfaces. What is the advantage of using a bridge? In large systems, what would happen to your bus performance if you tried to put all your peripherals on a single bus?
5. In class we looked at the block diagram of the system-on-chip device, Atmel SAM9G45 and also the TI TM4C1294. Why is the Atmel device so much more complex? Why does it have so many external memory interfaces, while the TM4C1294 does not? What additional module is added to the Atmel device to compensate for the reduced memory performance of external devices? What module is added to improve bus performance?
6. What are the major buses in the TM4C1294? What are their primary connections?
7. What are bandwidth and latency? Under what conditions does latency significantly affect the bandwidth of the bus and under what conditions is it negligible. Explain.
8. I have a 32-bit bus with a 50 Mhz system clock. The system can perform one transaction per system clock cycle. What is its maximum bandwidth in bits/sec and in bytes/sec?
9. Below is a timing diagram of a sequential read operation, once data starts returning it returns read data every clock cycle. The transaction starts when the READ command and address are valid at time = T0. The data is returned on the DQ lines, the blue/grey color indicate data is valid? What is the latency of the read command in clock cycles? If 54 sequential reads in a row, what is my bus utilization.

   The latency of the read command is 6 clocks.

   If 54 sequential reads are executed in a row the total time is 6 clocks of latency + 54 clocks of data transmission. 54 transactions happen in 60 clock cycles = 54/60 = 0.9 => 90%



10. Below is a timing diagram of a write-to-read operation. There are two clock cycles of latency when alternating between read and write operations denoted Don't care values of the DQ, data values. If I execute a sequential write for 10 transactions and then a sequential read for 5 transactions followed by another sequential write for 20 transactions. What will be the bus utilization?
    The total number of clocks to perform 35 transactions = 41 clocks. Utilization = 35/41 = 85%.
    2 clocks for 1st write latency + 10 clocks for write data transmission +
    2 clocks for 1st read latency + 5 clocks for read data transmission +
    2 clocks for 2nd write latency + 20 transactions.



11. Why are microcontrollers preferred for real time applications even though their maximum performance can be an order of magnitude less than that of a SoC device.
12. Why are caches included in SoC devices? How do they improve bus utilization and system performance? What is the disadvantage of using caches in a real-time application?
13. What are two advantages of using a FIFO versus performing multiple single access operations?
14. In the ece3849_adc_fifo example,
15. What is the difference in ADC configuration? What do the ADC_CTL_IE and ADC_CTL_END flags in the ADCSequenceStepConfigure() functional call do?
16. What major change is needed to ADC_ISR functionality to support the FIFO operation instead of the original non-FIFO version?
17. Write the ADCSequenceStepConfigure() function calls needed to configure ADC FIFO with a depth of 6 that triggers an interrupt every 3 samples. What is the ADC_ISR period and ISR relative deadline for this configuration?
    3 samples are read for each interrupt, ISR period = 1usec * 3 samples = 3 usec.
    ISR relative deadline = 4 usec: It will take 3 additional samples to fill the 6 sample FIFO + 1 sample to reach the overflow condition.

Module 5 – Lecture 19 (Tuesday 5/4)

18. When using the FIFO in the ADC example what was the performance trade-off when selecting the ISR period?

19. A typical hardware FIFO for an input peripheral interrupts when the FIFO is half-full. What is the reasoning behind this? Why is it better than interrupting when full or non-empty?

    If the FIFO interrupts when full, the relative deadline (until FIFO overflow) is much

    shorter (ISR can tolerate very little latency) than interrupting when half-full or non-empty.

    If the FIFO interrupts when non-empty, the CPU load is much higher (due to much shorter period) than interrupting when half-full or full.

    Interrupting when half-full is a compromise between CPU load and the relative deadline (latency tolerance) for the ISR/task handling the FIFO.

20. What is an advantage of using a DMA?
21. List two potential advantages of a hardware FIFO over a DMA.

    Hardware FIFOs are less costly because they do not require a DMA controller, so the CPU remains the only bus mater. Software for DMA is also more complex. (Cost is only lower if FIFOs are short.)

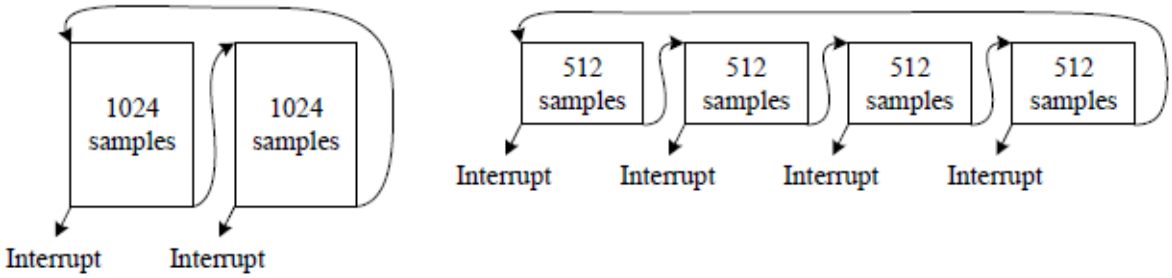
    Hardware FIFOs do not steal memory cycles from the CPU as DMA might. So, hardware FIFOs do not affect the CPU performance when I/O transfers occur to/from the FIFOs. In contrast, DMA can introduce additional latency and effective "CPU load." (Note that in the MCU we used in this course, the DMA controller never causes the CPU to stall.)

    A FIFO provides more deterministic performance. In systems where the DMA controller cannot stall the CPU, it is unclear what the maximum response time of the DMA operation will be.

22. Name an advantage and a disadvantage of a bus arbiter versus a DMA controller.
23. What is the difference between a dual address and single address DMA transfer?

More questions on next page

24. You configured the ADC DMA in ping-pong mode with two 1024- sample buffers (left diagram). For a generic real-time system (not your lab), what is the benefit to using four 512-sample buffers (right diagram) instead? What is a disadvantage? Assume that on every DMA interrupt, the DMA ISR keeps the continuous DMA transfer going, and signals a Task to process the completed DMA buffer. Hint: What is the latency between when the first sample is taken and when it can be processed by the data? What is the relative deadline and interrupt period for each of the configuration.



Advantage 2-buffer system

- The two-buffer configuration will have a longer ISR period and fewer interrupts, lowering the CPU utilization.

Advantage 4-buffer system

- The buffers are shorter in length, therefore the delay between when the data is originally sampled and when the data can be processed is half as long.
- The relative deadline for the 2-buffer system is the time to take 1024 samples. The relative deadline for the 4-buffer system is 3*512samples = 1536 samples.

25. List three advantages of ARM based designs.

Module 5 – Lecture 20 (Thursday 5/6)

26. How does pipelining improve the performance of the TM4C1294 processor. How many stages are there and what does each stage do?
27. Using the ARM CPU block diagram, describe the instruction execution flow through the processor. Where must the instruction operands be located, for an instruction to execute?
28. There are 16 CPU registers in the ARMv7-M architecture. What are each used for?
29. In addition to the 16 CPU registers there are a set of Program Status Registers what is the function of the APSR register. How are the flags in this register used by the instruction set?
30. Why is word alignment important for 32-bit and 16-bit words? What condition must be met for 32-bit words to be aligned? When aligned how is each byte of the word stored given that the processor is little Endian?
31. I have an array of short variables that are aligned on 4-byte boundaries. What negative affect will this have?
32. I have an integer variable stored at address 0x2202. What negative affect will this have?
33. Many instructions use constants. Some constants are limited to 8, 12 or 16-bits in size. Why are they limited?

Module 5 – Lecture 21 (Friday 5/7)

34. Convert the following C code fragment into ARM assembly. Register assignments are indicated in the comments (do not write assembly code for the variable declarations). You may use other general-purpose registers as temporary storage. Use only the ARM instructions in the lecture notes and the ARM assembly summary (Pages/Datasheets section on Canvas). You do not need to use any of the advanced instruction formats, such as applying a shift to the second source operand.

```
uint32_t i, m;          // r0 = i; r1 = m;
uint32_t A[20], B[20];  // r2 = &A[0]; r3 = &B[0];

B[i] = ((m - A[i]) & 0x0f) | (B[i] << 4);
```

Solution #34

This version uses only the basic instruction formats (all you need for the exam):

```
lsl r4, r0, #2     ; r4 = i*4; // array offset
ldr r5, [r2, r4]   ; r5 = copy of A[i];
sub r5, r1, r5     ; r5 = m - r5;
and r5, r5, #0x0f  ; r5 = r5 & 0x0f;
ldr r6, [r3, r4]   ; r6 = copy of B[i];
lsl r6, r6, #4     ; r6 = r6 << 4;
orr r5, r5, r6     ; r5 = r5 | r6;
str r5, [r3, r4]   ; B[i] = r5; // write to B[i] in memory
```

The following solution takes advantage of more advanced instruction formats:

```
ldr r5, [r2, r0, lsl #2] ; r5 = copy of A[i];
sub r5, r1, r5           ; r5 = m - r5;
and r5, r5, #0x0f        ; r5 = r5 & 0x0f;
ldr r6, [r3, r0, lsl #2] ; r6 = copy of B[i];
orr r5, r5, r6, lsl #4   ; r5 = r5 | (r6 << 4);
str r5, [r3, r0, lsl #2] ; B[i] = r5; // write to B[i]
```

More questions on next page

35. Convert the following C code fragment into ARM assembly. Register assignments are indicated in the comments (do not write assembly code for the variable declarations). You may use other general-purpose registers as temporary storage. Use only the ARM instructions in the lecture notes and the ARM assembly summary (Pages/Datasheets section on Canvas). You do not need to use any of the advanced instruction formats, such as applying a shift to the second source operand.

```
int32_t i, x;      // r0 = i; r1 = x;
int32_t A[32];     // r2 = &A[0];

i = 0;
x = 100;
while (i != 32) {
    x -= A[i];
    if (x <= 0) {
        break;     // exit the loop
    }
    i++;
}
```

Solution #35 (option #1)

```
        mov r0, #0        ; i = 0;
        mov r1, #100      ; x = 100;

loop1   cmp r0, #32
        beq done1         ; if i == 32, exit the loop

        lsl r3, r0, #2    ; r3 = i * 4; // array offset
        ldr r4, [r2, r3]  ; r4 = copy of A[i];
        sub r1, r1, r4    ; x -= (copy of A[i]);

        cmp r1, #0
        ble done1         ; if x <= 0, exit the loop

        add r0, r0, #1    ; i++;
        b   loop1         ; continue loop
done1
```

The version on the next page applies a few optimizations: using the shifter and ALU in one instruction, setting flags in the sub instruction, eliminating the unconditional jump in the loop.

Solution #35 (option #2)

Partially optimized version (other optimizations are possible)

```
        mov r0, #0          ; i = 0;
        mov r1, #100        ; x = 100;

        b   cond1           ; jump to the loop condition

loop1   ldr r4, [r2, r0, lsl #2]  ; r4 = copy of A[i];
        subs r1, r1, r4     ; x -= (copy of A[i]); // set flags
        ble done1           ; if x <= 0, exit the loop

        add r0, r0, #1      ; i++;

cond1   cmp r0, #32
        bne loop1           ; if i != 32, continue loop
done1
```

36. Convert the following C code fragment into ARM assembly. Register assignments are indicated in the comments (do not write assembly code for the variable declarations). You may use other general-purpose registers as temporary storage. Use only the ARM instructions in the lecture notes and the ARM assembly summary (Pages/Datasheets section on Canvas). You do not need to use any of the advanced instruction formats, such as applying a shift to the second source operand.

```
int32_t i, x;   // r0 = i; r1 = x;
int32_t A[64];  // r2 = &A[0];

if ((i == 0) || (A[i] <= A[i - 1])) {
       x += A[i];
}
else {
       x--;
}
```

Solutions on next page

Question 36 Option #1

```
        lsl r3, r0, #2     ; r3 = i*4;
        add r3, r2, r3     ; r3 = &A[i];
        ldr r4, [r3]       ; r4 = copy of A[i];
                           ; need to read A[i] in all cases

        cmp r0, #0         ; compare i to 0
        beq then1          ; if i == 0, goto then1
                           ; evaluate second condition only if i != 0
        ldr r5, [r3, #-4]  ; r5 = copy of A[i - 1];
        cmp r4, r5         ; compare copies of A[i] and A[i - 1]
        bgt else1          ; if A[i] > A[i - 1], goto else1

then1   add r1, r1, r4     ; x = x + (copy of A[i] in r4);
        b   done1          ; goto done1

else1   sub r1, r1, #1     ; x--;

done1
```

Question 36 Option #2:

(alternative 1)

```
        cmp r0, #0         ; compare i to 0
        beq then1          ; if i == 0, goto then1
                           ; evaluate second condition only if i != 0
        lsl r3, r0, #2     ; r3 = i*4;
        add r3, r2, r3     ; r3 = &A[i];
        ldr r4, [r3]       ; r4 = copy of A[i];
        ldr r5, [r3, #-4]  ; r5 = copy of A[i - 1];
        cmp r4, r5         ; compare copies of A[i] and A[i - 1]
        bgt else1          ; if A[i] > A[i - 1], goto else1

then1   lsl r3, r0, #2     ; r3 = i*4;
        ldr r4, [r2, r3]   ; r4 = copy of A[i];
        add r1, r1, r4     ; x = x + (copy of A[i]);
        b   done1          ; goto done1

else1   sub r1, r1, #1     ; x--;

done1
```

Option #3 on next page.

Question 36 Option #3

```
        ldr r4, [r2, r0, lsl #2] ; r4 = copy of A[i];

        cmp r0, #0          ; compare i to 0
        beq then1           ; if i == 0, goto then1
                            ; evaluate second condition only if i != 0
        sub r3, r0, #1      ; r3 = i - 1;
        ldr r5, [r2, r3, lsl #2] ; r5 = copy of A[i - 1];
        cmp r4, r5          ; compare copies of A[i] and A[i - 1]
        bgt else1           ; if A[i] > A[i - 1], goto else1

then1   add r1, r1, r4      ; x = x + (copy of A[i]);
        b   done1           ; goto done1

else1   sub r1, r1, #1      ; x--;

done1
```

37. Convert the following C code fragment to ARM assembly. Register assignments are indicated in the comments. You may use other general-purpose registers as temporary storage. Notes: This is not a complete function. You do not need to worry about the function call convention. You do not need to write any assembly code for the variable declarations. The array is stored in memory.

```
uint32_t b, i;   // r0 = b; r1 = i;
uint32_t A[64];  // r2 = &A[0];

if (i == 64) {
    i = 0;
}
else {
    b = A[i];
    if (b > 100) {
         A[i] = b >> 1;
    }
}
```

Solution #37

```
        cmp r1, #64         ; compare i to 64
        bne else1           ; if i != 64, goto else1

        mov r1, #0          ; i = 0;
        b   done1           ; goto done1

else1   lsl r3, r1, #2      ; r3 = i * 4; // array offset
        ldr r0, [r2, r3]    ; b = copy of A[i];
        cmp r0, #100        ; compare b to 100
        bls done1           ; if b <= 100, goto done1

        lsr r4, r0, #1      ; r4 = b >> 1;
        str r4, [r2, r3]    ; A[i] = r4;
```

done1

38. Convert the C function `max4()` into ARM assembly. Your resulting assembly functions must be callable from C, so should adhere to the C register convention discussed in lecture (also in the ARM assembly summary on Canvas). You do not need to use any assembler directives or optimize your code. Hints: Carefully study which registers a function must preserve upon return, and which registers the caller can expect to be modified across a function call. You will need to save registers onto the stack to implement this function.

```
int32_t max2(int32_t a, int32_t b); // implemented elsewhere

int32_t max4(int32_t a, int32_t b, int32_t c, int32_t d)
{
    return max2(max2(a, b), max2(c, d));

}
```

Solution #38

```
; int32_t max4(int32_t a, int32_t b, int32_t c, int32_t d)
; upon entry (by the C convention):
;   r0 = argument #1 = a
;   r1 = argument #2 = b
;   r2 = argument #3 = c
;   r3 = argument #4 = d
;   note: r0-r3, r12 and lr do not need to be preserved upon return
; local variables (in r4-r11 - preserved across function calls):
;   r4 = c
;   r5 = d
;   r6 = max2(a,b) return value
; upon return (by the C convention):
;   r0 = return value
max4    push {r4-r6, lr} ; save registers on the stack
        mov  r4, r2      ; r4 = c;
        mov  r5, r3      ; r5 = d;

                         ; argument #1 is already a
                         ; argument #2 is already b
        bl   max2        ; call max2();// r0-r3, r12, lr not preserved
        mov  r6, r0      ; r6 = max2(a,b) return value;

        mov  r0, r4      ; argument #1 = c;
        mov  r1, r5      ; argument #2 = d;
        bl   max2        ; call max2();

        mov  r1, r0      ; argument #2 = max2(c,d) return value;
        mov  r0, r6      ; argument #1 = max2(a,b) return value;
        bl   max2        ; call max2(); // return value in r0

        pop  {r4-r6, pc} ; restore registers and return
```

39. Convert the C function `f2()` to ARM assembly. Assume the function `f1()` is implemented elsewhere. Make sure to follow the C calling conventions for passing arguments and return values, as well as preserving registers that need to be preserved. Remember that certain registers can be overwritten by a function call.

```
int32_t f1(int32_t x);

int32_t f2(int32_t a, int32_t b)
{
    return a - f1(b - f1(b));
}
```

Solution #39

```
; upon entry into the function:
;   r0 = a;
;   r1 = b;
; inside the function:
;   r4 = a;
;   r5 = b;
; upon return:
;   r0 = return value;
f2      push {r4, r5, lr} ; preserve registers on the stack

        mov  r4, r0       ; r4 = a;
        mov  r5, r1       ; r5 = b;

        mov  r0, r1       ; 1st argument = b;
        bl   f1           ; call f1();

        sub  r0, r5, r0   ; 1st argument = b - (f1() return value);
        bl   f1           ; call f1();
        sub  r0, r4, r0   ; return val. = a - (f1() return value);

        pop  {r4, r5, pc} ; restore registers and return
```

40. Convert the C function f2() to ARM assembly. Assume the function f1() is implemented elsewhere. Make sure to follow the C calling conventions for passing arguments and return values, as well as preserving registers that need to be preserved. Remember that certain registers can be overwritten by a function call.

```
int32_t f1(int32_t x, int32_t y);

int32_t f2(int32_t a)
{
    int32_t b = f1(a, 5);
    return b + f1(a, b - 5);
}
```

Solution #40

```
; upon entry into the function:
;   r0 = a;
; inside the function:
;   r4 = a;
;   r5 = b;
f2      push {r4, r5, lr} ; preserve registers on the stack

        mov  r4, r0       ; r4 = a;

                          ; 1st argument is already a
        mov  r1, #5       ; 2nd argument = 5;
        bl   f1           ; call f1();
        mov  r5, r0       ; b = return value of f1();

        mov  r0, r4       ; 1st argument = a;
        sub  r1, r5, #5   ; 2nd argument = b - 5;
        bl   f1           ; call f1();

        add  r0, r5, r0   ; r0 = b + (return value of f1());

        pop  {r4, r5, pc} ; restore registers and return (value in r0)
```

More questions on next page

Module 5 – Lecture 23 (Tuesday 5/11)

41. Determine the number of clock cycles required to execute the following ARM assembly function on a Cortex-M4 system. Show your work. You need to analyze the control flow in this function.

```
; void initab(int32_t a[], int32_t b[], int32_t c);
initab  mov r3, #0
loop1
lsl r12, r3, #2
str r2, [r0, r12]
str r2, [r1, r12]
add r3, r3, #1
cmp r3, #3
ble loop1
bx  lr
```

Solution #41

```
; void initab(int32_t a[], int32_t b[], int32_t c);

initab  mov r3, #0
; Data operation: 1 cycle

loop1   lsl r12, r3, #2
; Data operation: 1 cycle

        str r2, [r0, r12]
; STR with register offset,
; not pipelined: 2 cycles
        str r2, [r1, r12]
; STR with register offset,
; not pipelined: 2 cycles
        add r3, r3, #1
; Data operation: 1 cycle

        cmp r3, #3
; Data operation: 1 cycle

        ble loop1
; Branch to label
; Taken: 2 cycles (when r3 <= 3)
; Not taken: 1 cycle (when r3 > 3)
        bx  lr
; Branch to register,
; always taken: 3 cycles
```

Loop executes 4 times:
at cmp r3, #3
r3 = 1, 2, 3, 4
ble not taken in the last iteration

Total cycles = 1 (mov)
+ (1 (lsl) + 2 (str) + 2 (str) + 1 (add) + 1 (cmp) + 2 (bne)) * 3
+ 1 (lsl) + 2 (str) + 2 (str) + 1 (add) + 1 (cmp) + 1 (bne)
+ 3 (bx)

= 39

42. The following is a block copy function that copies data one word (4 bytes) at a time. Determine the number of clock cycles needed to execute the assembly version of this function on a Cortex-M4, assuming `count=8` and no memory access stalls. Use the instruction timing summary from lecture notes or the ARM assembly summary on Canvas. If you are interested, detailed timing information is in the ARM Cortex-M4 Technical Reference Manual section 3.3. Hint: The assembly code is not a one-to-one conversion from C, but the loop runs for the same number of iterations.

```
void block_copy4(uint32_t dst[], uint32_t src[], uint32_t count)
{
    uint32_t i;
    for (i = 0; i < count; i++) {
        dst[i] = src[i];
    }
}
```

```
; void block_copy4(uint32_t dst[], uint32_t src[], uint32_t count);
; Arguments:
; r0 = dst = destination pointer
; r1 = src = source pointer
; r2 = count = number of 4-byte words to copy
; Local variables:
; r2 = number of bytes to copy
; r3 = offset = i * 4  (does not follow the C code exactly)
; r12 = temp
block_copy4
        lsl   r2, r2, #2     ; r2 = count * 4; // # of bytes to copy
        mov   r3, #0         ; offset = 0;

loop1   cmp   r3, r2
        bhs   done1          ; if (offset >= total bytes), done

        ldr   r12, [r1, r3]  ; load word from source
        str   r12, [r0, r3]  ; store word to destination
        add   r3, r3, #4     ; offset += 4;
        b     loop1          ; continue loop

done1   bx    lr             ; return
```

Solution on next page

Solution #42

```
block_copy4
        lsl   r2, r2, #2      ; data operation: 1 cycle
        mov   r3, #0          ; data operation: 1 cycle

loop1   cmp   r3, r2          ; data operation: 1 cycle
        bhs   done1           ; branch to label: 1 cycle if not taken
                              ;                  2 cycles if taken

                              ; loop 8 cycles/iteration
        ldr   r12, [r1, r3]   ; LDR, 1st in sequence: 2 cycles
        str   r12, [r0, r3]   ; STR with register offset
                              ;    2nd in sequence: 1 cycle
        add   r3, r3, #4      ; data operation: 1 cycle
        b     loop1           ; branch to label, always taken: 2 cycles

done1   bx    lr              ; branch to register, always taken:
                              ;    3 cycles
```

9th iteration

The loop is executed 8 times (`count = 8`), `bhs` is not taken in these iterations.
cycles each iteration = 1 (cmp) + 1 (bhs) + 2 (ldr) + 1 (str) + 1 (add) + 2 (b) = 8

In the 9th iteration, `bhs` is taken.
cycles for 9th iteration = 1 (cmp) + 2 (bhs) = 3

The remaining instructions are executed once.
cycles for instructions executed once = 1 (lsl) + 1 (mov) + 3 (bx) = 5

Total clock cycles = 8 (cycles/iteration) * 8 (iterations) + 3 (9th iteration) + 5 = 72

Note that on the exam you may need to interpret assembly language to determine the number of iterations. In this problem, the following instructions determine the number of iterations. It may help to write out the value of `r3` each time it is used in the `cmp` instruction.

```
        lsl   r2, r2, #2      ; r2 = 8 << 2 = 32;
        mov   r3, #0          ; r3 = 0;

loop1   cmp   r3, r2          ; r3 = 0,4,8,12,16,20,24,28,32 (last)
        bhs   done1           ; if r3 >= 32, exit loop

        add   r3, r3, #4      ; r3 += 4;
        b     loop1           ; continue loop
done1
```

43. Determine the **maximum** number of clock cycles required to execute the following ARM assembly function on a Cortex-M4 system. Show your work. You need to analyze the control flow in this function.

```
; void f3(int32_t *p1, int32_t *p2);

f3      ldr r2, [r0]

        ldr r3, [r1]

        cmp r2, r3

        blt else1

        str r2, [r1]

        b   done1

else1   str r3, [r0]

done1   bx  lr
```

Solution on next page

Solution #43

```
; void f3(int32_t *p1, int32_t *p2);

f3      ldr r2, [r0]
; LDR without offset, not following another LDR/STR: 2 cycles

        ldr r3, [r1]
; LDR without offset, pipelined after another LDR/STR: 1 cycle

        cmp r2, r3
; Data operation: 1 cycle

        blt else1
; Branch with immediate destination: taken        2 cycles
;                                    not taken    1 cycle
        str r2, [r1]
; STR without offset: 1 cycle

        b   done1
; Branch with immediate destination: always taken   2 cycles

else1   str r3, [r0]
; STR without offset: 1 cycle

done1   bx  lr
; Branch with register destination: always taken   3 cycles
```

There are two paths of execution, depending on whether the `blt` branch is taken or not.

`blt` not taken:

clock cycles = 2 + 1 + 1 + 1 (blt) + 1 (str) + 2 (b) + 3 (bx) = 11

`blt` taken:

clock cycles = 2 + 1 + 1 + 2 (blt) + 1(str) + 3 (bx) = 10

<u>maximum</u> number of clock cycles = <u>11</u>.