Final Exam- 9047

Joshua Bainbridge 250869629

Part 1: Short Answers

a) In the context of this course, what is the best definition of a **chip select logic**?

In the context of this course the term **chip select logic** refers to the logical circuits used to select a memory chip by the microprocessor on a microcontroller. When constructing memory for a microcontroller different RAM and ROM chips are used. Additionally multiple RAM or ROM chips are combined to increase the number of address spaces or the width of the memory. When a microprocessor is fetching or storing data to memory the chip select logic is needed to ensure the data goes to the right memory chip.

b) What must the status flags be for a mnemonic to execute if it has the lo condition code?

The mnemonic lo is an unsigned comparison for less than. To test this the processor looks for the carry flag (C) to be 0.

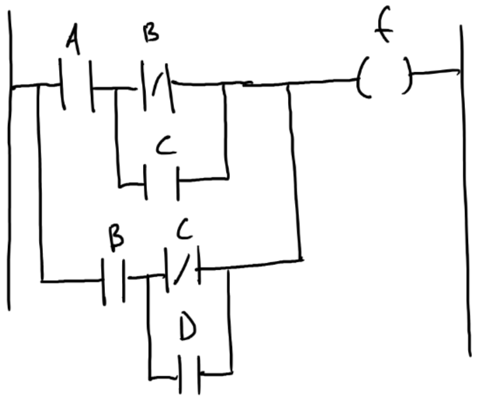
c) A UART peripheral is configured to 9-1-2. What does this mean?

9 pulses of transmitted data followed by 1 odd parity bit and finally two pluses as the stop signal.

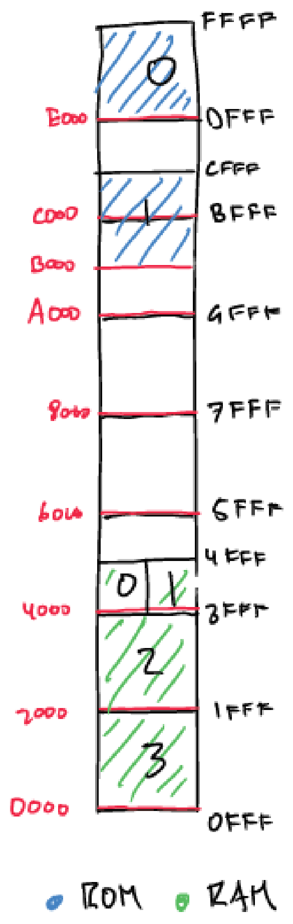
d) In the context of this course, what is the best definition of full coverage?

Full coverage refers to the sensing area of a wireless sensor network. A sensor network has full coverage if in a given LxL area every point is being monitored by at leas one sensor node.

e) Draw the relay ladder logic (RLL) program to implement the following in one rung. Here A, B, C, D are the inputs.



Part 2: Memory Mapping



ROM: 8k top: $\boxed{\text{E000 – FFFF}}$

8k middle: $\boxed{\text{B000 – CFFF}}$

RAM: 20k bottom: $\boxed{\text{0000 – 4FFF}}$

$(4FFF - 0000) + 1 = 5000 = 2^{2.3}(2^4)(2^4)(2^4)$

$= 2^{14.3} = 2^{4.3}(2^{10}) = 20k$

ROM0:	1110	0000	0000	0000
(8k×8)	1111	1111	1111	1111
ROM1:	1011	0000	0000	0000
(8k×8)	1100	1111	1111	1111
RAM 0,1:	0100	0000	0000	0000
(4k×4)	0100	1111	1111	1111
RAM 2:	0010	0000	0000	0000
(8k×8)	0011	1111	1111	1111
RAM 3:	0000	0000	0000	0000
(8k×8)	0001	1111	1111	1111

Chip select:

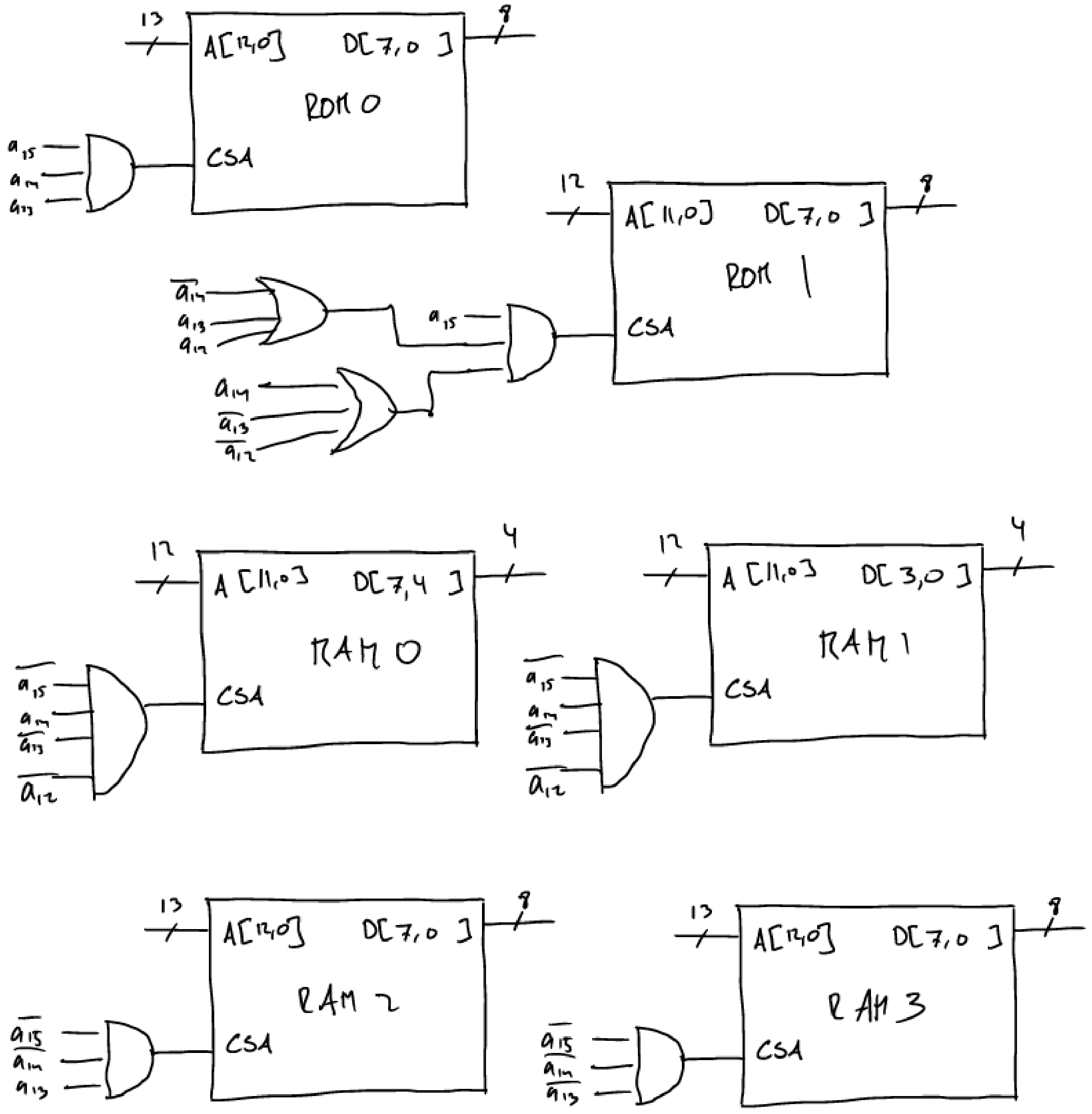
ROM 0: $a_{15}\, a_{14}\, a_{13}$

ROM 1: $a_{15}\,(\overline{a_{14}}\, a_{13}\, a_{12} + a_{14}\, \overline{a_{13}}\, \overline{a_{12}})$

RAM 0,1: $\overline{a_{15}}\, a_{14}\, \overline{a_{13}}\, \overline{a_{12}}$

RAM 2: $\overline{a_{15}}\, \overline{a_{14}}\, a_{13}$

RAM 3: $\overline{a_{15}}\, \overline{a_{14}}\, \overline{a_{13}}$


13
A[12,0]
D[7,0]
8
ROM 0
CSA
a_{15}
a_{14}
a_{13}
12
A[11,0]
D[7,0]
8
ROM 1
CSA
$\overline{a_{14}}$
a_{13}
a_{12}
a_{15}
a_{14}
$\overline{a_{13}}$
$\overline{a_{12}}$
12
A[11,0]
D[7,4]
4
RAM 0
CSA
$\overline{a_{15}}$
a_{14}
$\overline{a_{13}}$
$\overline{a_{12}}$
12
A[11,0]
D[3,0]
4
RAM 1
CSA
$\overline{a_{15}}$
a_{14}
$\overline{a_{13}}$
$\overline{a_{12}}$
13
A[12,0]
D[7,0]
8
RAM 2
CSA
$\overline{a_{15}}$
$\overline{a_{14}}$
a_{13}
13
A[12,0]
D[7,0]
8
RAM 3
CSA
$\overline{a_{15}}$
$\overline{a_{14}}$
$\overline{a_{13}}$

Part 3: Assembly Language

a)

1) move 188 → 0xbc

2) load value at r2 → 0x1000 → r3 = 0xdd

3) load value at r2+4 → 0x1004 → r0 = 0x62
   ↳ keep change

4) store 0xbc at 0x1004

5) stor 0xdd at 0x100C

r0 :	0x62	0x1000 :	0xdd
r1 :	0x00	0x1004 :	0xbc
r2 :	0x1004	0x1008 :	0x81
r3 :	0xdd	0x100C :	0xdd
r4 :	0xbc	0x1010 :	0x5b

b)

Answer the following questions. [5 marks]

- Which line of code is the start of a subroutine?
- How many times does the code on line 4 execute?
- What is the final value in memory address 0xAA002004?

a) The name of the subroutines are located at 2,8,10,12. The first line of code that is executed from each subroutine is on 3,9,11,13. Line 3 is the first line of the first subroutine executed.

b) 15

c) 0x800000bb

Part 4: Peripherals

a) How can you tell which pins are set to output and which are set to input for this GPIO port?

All GPIO have along with the given data registers mapped in memory space a control register mapped just above or below the data register. For example in the problem the give GPIO is mapped to 0xE000. Let assume the control register is located at memory address 0xE004. The operation of the pin corresponds to the bit value of the pin in the control register. The protocol for the ARM processor is a bit value of 1 means the pin is an output pin and a bit value of 0 means the pin is an input pin.

To determine if a specific pin in an input or output the following **pseudo** code can be used:

- Read the values of the control register and store then in a register → ldr r1 [0xE004]
- Use a bit mask to isolate the pin being looked at. → and r1, #1
    - For example if pin 0 is the pin of interest, using the AND operator with 1 will and the value of the control register with 0x0001. All other values c_{15} to c_1 with be set to 0 and c_0 will be the only value left.
- Compare masked value to value of GPIO pin → cmp r1, #1
- If r1 = 1 the pin is an output, if r1= 0 the pin is an input

b) Following **pseudo** code can be used:

```
ldr r1, [0xE004] → load in the content of the control register

ldr r2, 0xFFDB → load 1111 1111 1101 1011, only bits 2 and 5 are 0

and r1, r2 → will return: c15 c14 c13 c12 c11c10 c9c8 c7c60c4 c30c1c0

str r1, [0xE004]
```

c) Following **pseudo** code can be used:

```
ldr r1, [0xE004] → load in the content of the control register

ldr r2, 0xFFDB → load 0001 0100 0000 0000, only bits 12 and 10 are 1

or r1, r2 → will return: c15 c14 c131 c111c9c8 c7c6 c5c4 c3 c2c1c0

str r1, [0xE004]
```

d) Following **pseudo** code can be used:

```
ldr r1, [0xE004] → load in the content of the control register

ldr r2, 0x4 → load 0000 0000 0000 0100

and r3, r1, r2 → will return: 0000 0000 0000 0c200

lsr r3, 2 → will return: 0000 0000 0000 000c2

ldr r2, 0x20→ load 0000 0000 0010 0000

and r4, r1, r2 → will return: 0000 0000 00c50 0000

lsr r4, 5 → will return: 0000 0000 0000 000c5

add r4, r3
```

e) Following **pseudo** code can be used:

ldr r1, [0xE004] $\rightarrow$ load in the content of the control register

and r2, r0, 2 $\rightarrow$ load r0_1 into r2

lsl r2, 2 $\rightarrow$ will return: r$0_1$00

and r3, r0, 1 $\rightarrow$ load r0_0 into r2

add r2, r3 $\rightarrow$ will return: r$0_1$00 r0_0

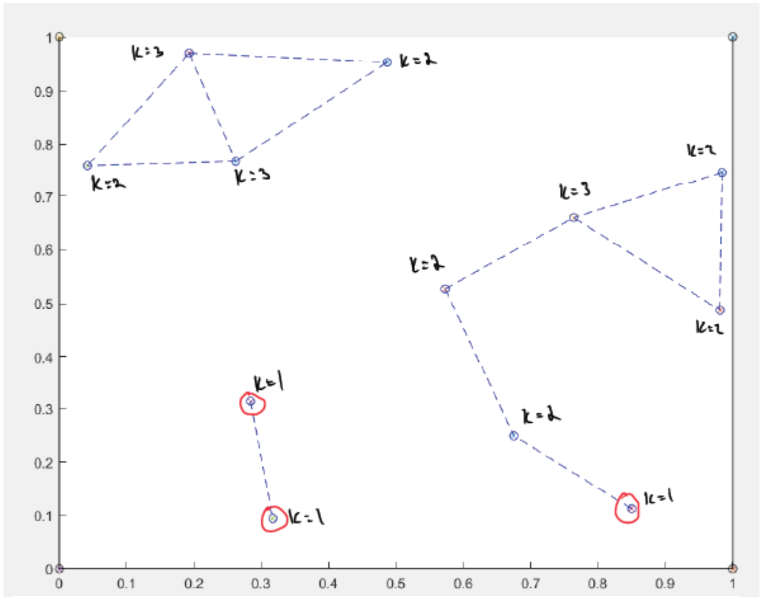
lsl r2, 10 $\rightarrow$ will return: 000 r0_1 0r$0_0$00 0000 0000

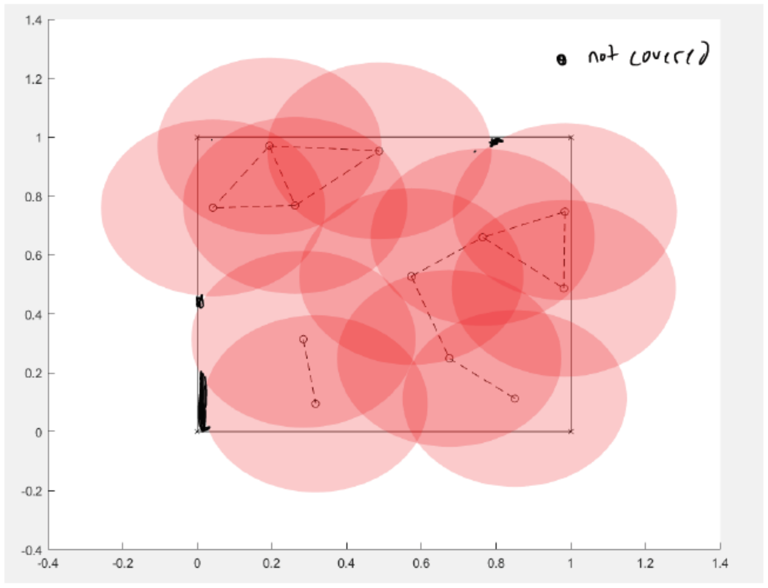
Part 5: Embedded System Design

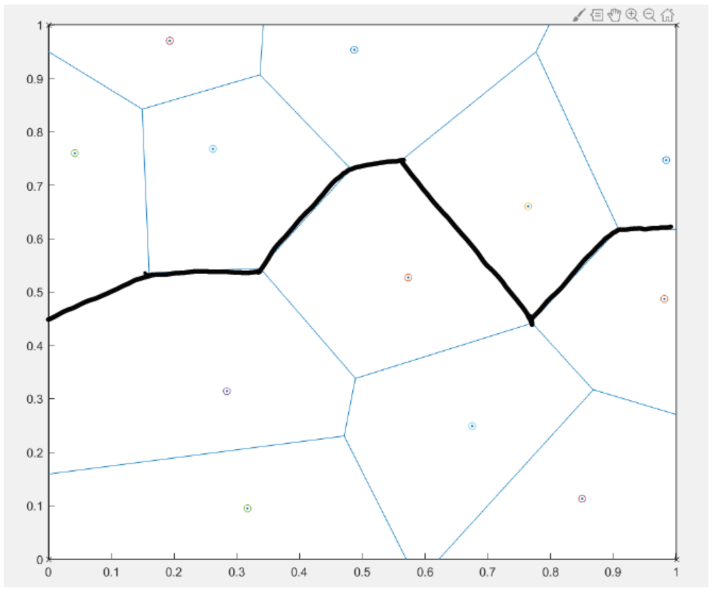
The code for this section can be found in the Appendix. In short it is a modification of the lab 2 assignment. While working on it for this exam I has some issues with the code, it won't compile properly but is good pseudo code if I had more time to do it for homework. The code functions as follows:

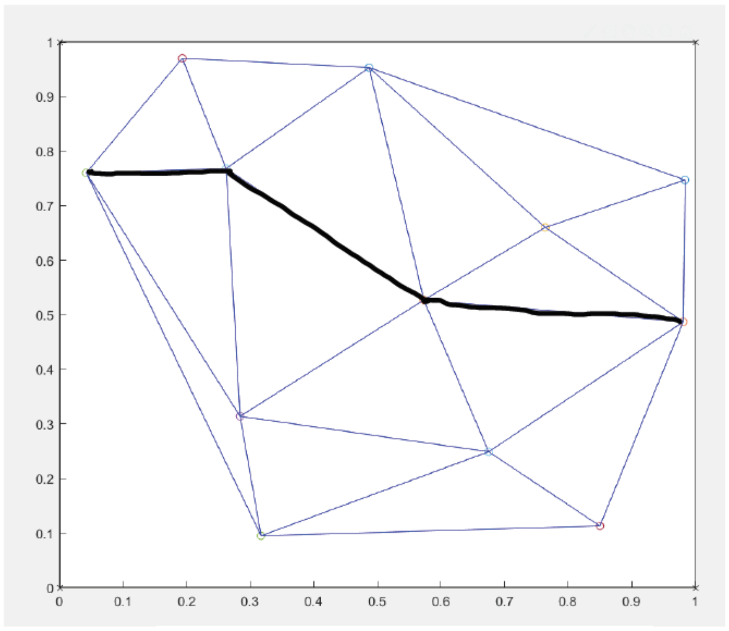
- The assignments were written same as the lab before. The code under .global _start sets up the interrupt and the clock. When I was attempting to write 1 000 000 000 to create a 10 second time the value was to large to fit into the timer registers. So for now the counter is kept at 5 seconds
- The main look is set to an idle state. It loops to itself without preforming any actions. This is done to prevent the timer from starting before button 0 is pushed.
- When button 0 is pushed it triggers the interrupt subroutine. The subroutine check which button was pushed and branches to the correct subroutine. In this case if button 0 is push the counter subroutine starts.
- The counter subroutine starting the timer by writing junk to the register. At this point the time begins to count. The code has the timer counter to 5. To get it to 10 second an flag register (r8) is used. R8 has 1 bit with is toggled using the XOR function. The number count register r12 is only incremented when r8 = 0. R12 is only increment every other timer cycle so as to count every 10 seconds.
- Finally the two remain interrupt buttons. Button 1 causes the program to branch back to the idle main_loop subroutine. Thus the counting stops. The timer continues to loop, however.
- Button 2 functions similar to button 0 and returns the program to the count subroutine.

Part 6: WSN






Breach Path


Support Path

a) the k-connectivity of the network is 1

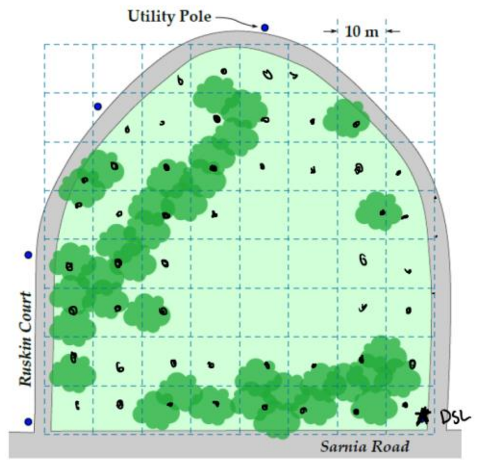
b) the k-coverage is 0 because some areas are not covered.

Part 7: Sensor Design

a)

Nodes	Components	Cost	Unit Quantity
Sensor	Base	1	50
	Sensor x2	3	
	High Batter x2	50	
	ZR (not boosted)	10	
		76	3800
DSL Nodes	Base	10	
	Power Converter	5	1
	DSL	20	
	ZC (boosted)	10	
		60	60

b)



c)

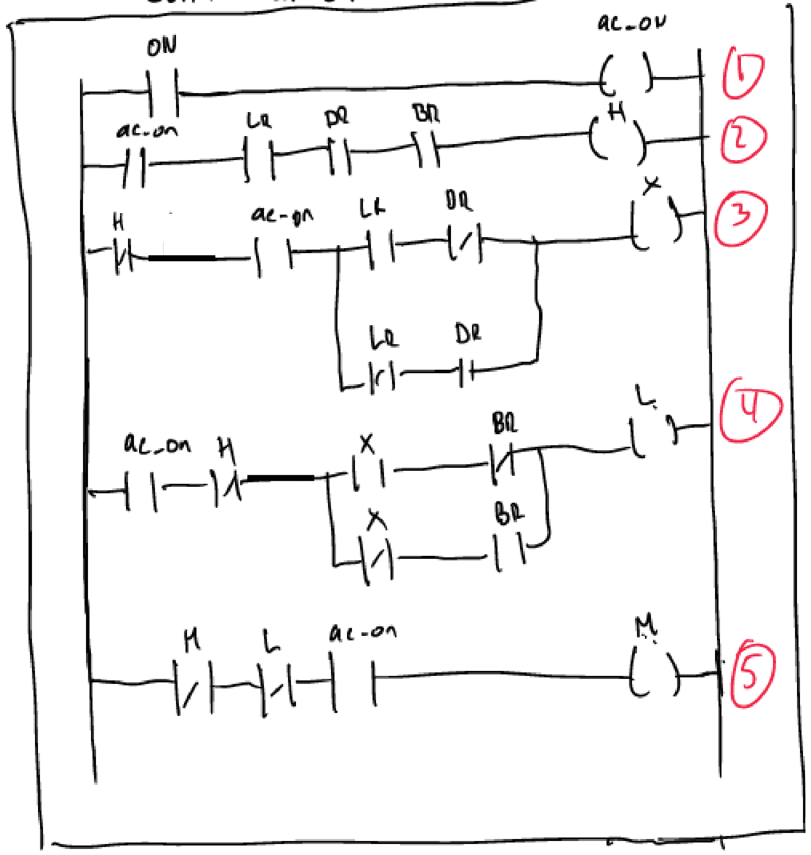
- The network has been built 1000 dollars under budget. The extra money can be used for maintenance of the system. Depending of the design of the base and amount of beer being consumed it is very possible some nodes will be ruined by passing miscreants and their actives in the park.
- The connectivity of the WSN is every storge with the overall k-coverage value of 3 and some nodes having up to 6 connections. This was achieved because no end nodes where deployed. (Partially due to time constraints)
- There are areas with no sensor coverage. The assumption was made that the majority of miscreants (even drunk ones) are unlikely to urinate in the middle of an open area. It is much more likely they would target partially covered areas like under trees. As a result the sensor network heavily covered the wooded areas with robust coverage.
- The one essential node on the network is the DSL node. To mitigate the likelihood of that node going dark it was attacked to a utility pole and should be placed high off the ground.
- The lifespan of this network should be medium to long. The signals should be sent via and interrupt protocol and parts of the network can likely be in an idle mode during the day as miscreants tend to come out at night.

Part 8: PLC

Sensors: living room : LR
dining room : DR
bedroom : BR

normal open
↳ returns 0
unless above 25 degrees

States: ac high : H
ac medium : M
ac low : L
ac ON : ac_on
Suith an ON : ON



1) Check the switch on the AC. If it is in the ON position set the state of the ac_on to be true
   - ON = ac_on
2) If the AC is on and all three sensor are active set AC to high
   - ac_on(LR)(DR)(BR) = H
3) If no high and AC is still on. Begin xoring all the sensors. (‘ is a NOT operator)
   - H’(ac_on)(LR) ⊕ (DR) = X
4) Second part of xoring.
   - H’(ac_on)(X) ⊕ (BR) = L
   - if the results in 1 it means only 1 of the sensors is active
5) Final step if it is not high or low and AC is on
   - H’(L’)(ac_on)=M

Appendix

```
.section .vectors, "ax"
    B     _start              // reset vector
    B     SERVICE_UND         // undefined instruction vector
    B     SERVICE_SVC         // software interrupt vector
    B     SERVICE_ABT_INST    // aborted prefetch vector
    B     SERVICE_ABT_DATA    // aborted data vector
    .word 0                   // unused vector
    B     SERVICE_IRQ         // IRQ interrupt vector
    B     SERVICE_FIQ         // FIQ interrupt vector

.global  _start
.data
timer_T: .word 500000000
timer_sT: .word 100000000
.text
_start:
    and r0, #0
    and r1, #0
    and r2, #0
    and r3, #0
    and r4, #0
    and r5, #0
    and r6, #0
    and r7, #0
    and r8, #0
    and r9, #0
    and r10, #0
    and r11, #0
    and r12, #0
    /*Set up stack pointers for IRQ and SVC processor modes*/
    MOV    R1, #0b11010010         // interrupts masked, MODE = IRQ
    MSR    CPSR_c, R1              // change to IRQ mode
    LDR    SP, =0xFFFFFFFF - 3     // set IRQ stack to A9 onchip memory
    /*Change to SVC (supervisor) mode with interrupts disabled*/
    MOV    R1, #0b11010011         // interrupts masked, MODE = SVC
    MSR    CPSR, R1                // change to supervisor mode
    LDR    SP, =0x3FFFFFFF - 3     // set SVC stack to top of DDR3 memory

    BL     CONFIG_GIC              // configure the ARM GIC

    // write to the pushbutton KEY interrupt mask register
    LDR    R0, =0xFF200050         // pushbutton KEY base address
    MOV    R1, #0xF                // set interrupt mask bits
    STR    R1, [R0, #0x8]          // interrupt mask register (base + 8)

    // enable IRQ interrupts in the processor
    MOV    R0, #0b01010011         // IRQ unmasked, MODE = SVC
    MSR    CPSR_c, R0

    // TIMER
    // get the addresses
    ldr r4 , =0xff202000
    ldr r5 , =0xff200000
    ldr r6 , adr_T
    ldr r7 , adr_sT
    ldr r0 , [r6]
    //initialize the 5 s count
    ldr r8, =0x00006500
    ldr r9,=0x00001dcd
    str r8 , [r4 , #8]
    str r9 , [r4 , #12 ]
    // start the timer for continuous
    // counting ( 0b0110 )
    mov r1 , #6
    str r1 , [r4 , #4]
    // get the 1 s interval
    ldr r1 , [r7]
main_loop:
    add r12, #1
    b main_loop
counter:
    // main program simply idles
    // get the current count
    // first write junk to counter
    str r1 , [r4 , #16 ]
    // now get the low count
    ldr r2 , [r4 , #16 ]
    // get the high count
    ldr r3 , [r4 , #20 ]
    // combine them
    add r2 , r3 , lsl #16
    // check if current count has
    // passed 1 s
    cmp r2 , r0
    bhi counter
    // increment LED pattern
    sublo r0 , r1
    blo counter
    ldr r0 , [r6]
    eor r11, #1
    cmp r11,#1
    addeq r10,#1
    b counter
    adr_T: .word timer_T
    adr_sT: .word timer_sT
/*Define the exception service routines*/
/*--- Undefined instructions --------------------------------------------------*/
SERVICE_UND:
    B SERVICE_UND
/*--- Software interrupts -----------------------------------------------------*/
SERVICE_SVC:
    B SERVICE_SVC
/*--- Aborted data reads ------------------------------------------------------*/
SERVICE_ABT_DATA:
    B SERVICE_ABT_DATA
/*--- Aborted instruction fetch -----------------------------------------------*/
SERVICE_ABT_INST:
    B SERVICE_ABT_INST
/*--- IRQ ---------------------------------------------------------------------*/
SERVICE_IRQ:
    PUSH    {R0-R7, LR}

    /*Read the ICCIAR from the CPU Interface*/
    LDR    R4, =0xFFFEC100
    LDR    R5, [R4, #0x0C]        // read from ICCIAR

FPGA_IRQ1_HANDLER:
    CMP    R5, #73
UNEXPECTED:
    BNE    UNEXPECTED             // if not recognized, stop here

    BL      KEY_ISR
EXIT_IRQ:
    /*Write to the End of Interrupt Register (ICCEOIR)*/
    STR    R5, [R4, #0x10]        // write to ICCEOIR

    POP    {R0-R7, LR}
    SUBS    PC, LR, #4

/*--- FIQ ---------------------------------------------------------------------*/
SERVICE_FIQ:
    B   SERVICE_FIQ
```

```
/**Configure the Generic Interrupt Controller (GIC)*/
.global CONFIG_GIC
CONFIG_GIC:
    PUSH   {LR}
    /*To configure the FPGA KEYS interrupt (ID 73):
     *1. set the target to cpu0 in the ICDIPTRn register
     *2. enable the interrupt in the ICDISERn register*/
    /*CONFIG_INTERRUPT (int_ID (R0), CPU_target (R1));*/
    MOV    R0, #73         // KEY port (Interrupt ID = 73)
    MOV    R1, #1          // this field is a bit-mask; bit 0 targets cpu0
    BL     CONFIG_INTERRUPT

    /*configure the GIC CPU Interface*/
    LDR    R0, =0xFFFEC100  // base address of CPU Interface
    /*Set Interrupt Priority Mask Register (ICCPMR)*/
    LDR    R1, =0xFFFF      // enable interrupts of all priorities levels
    STR    R1, [R0, #0x04]
    /*Set the enable bit in the CPU Interface Control Register (ICCICR).
     *This allows interrupts to be forwarded to the CPU(s)*/
    MOV    R1, #1
    STR    R1, [R0]

    /*Set the enable bit in the Distributor Control Register (ICDDCR).
     *This enables forwarding of interrupts to the CPU Interface(s)*/
    LDR    R0, =0xFFFED000
    STR    R1, [R0]
    POP    {PC}

/*
 *Configure registers in the GIC for an individual Interrupt ID
 *We configure only the Interrupt Set Enable Registers (ICDISERn) and
 *Interrupt Processor Target Registers (ICDIPTRn). The default (reset)
 *values are used for other registers in the GIC
 *Arguments: R0 = Interrupt ID, N
 *       R1 = CPU target
*/
CONFIG_INTERRUPT:
    PUSH   {R4-R5, LR}

    /*Configure Interrupt Set-Enable Registers (ICDISERn).
     *reg_offset = (integer_div(N / 32)*4
     *value = 1 << (N mod 32)*/
    LSR    R4, R0, #3          // calculate reg_offset
    BIC    R4, R4, #3          // R4 = reg_offset
    LDR    R2, =0xFFFED100
    ADD    R4, R2, R4          // R4 = address of ICDISER

    AND    R2, R0, #0x1F       // N mod 32
    MOV    R5, #1              // enable
    LSL    R2, R5, R2          // R2 = value

    /*Using the register address in R4 and the value in R2 set the
     *correct bit in the GIC register*/
    LDR    R3, [R4]            // read current register value
    ORR    R3, R3, R2          // set the enable bit
    STR    R3, [R4]            // store the new register value

    /*Configure Interrupt Processor Targets Register (ICDIPTRn)
     *reg_offset = integer_div(N / 4)*4
     *index = N mod 4*/
    BIC    R4, R0, #3          // R4 = reg_offset
    LDR    R2, =0xFFFED800
    ADD    R4, R2, R4          // R4 = word address of ICDIPTR
    AND    R2, R0, #0x3        // N mod 4
    ADD    R4, R2, R4          // R4 = byte address in ICDIPTR

    /*Using register address in R4 and the value in R2 write to
     *(only) the appropriate byte*/
    STRB   R1, [R4]

    POP {R4-R5, PC}

/*************************************************************************
 *Pushbutton - Interrupt Service Routine
 *
 *This routine checks which KEY has been pressed. It writes to HEX0
 ************************************************************************/
.global KEY_ISR
KEY_ISR:
    LDR     R0, =0xFF200050  // base address of pushbutton KEY port
    LDR     R1, [R0, #0xC]   // read edge capture register
    MOV     R2, #0xF
    STR     R2, [R0, #0xC]   // clear the interrupt

    LDR     R0, =0xFF200020  // based address of HEX display

CHECK_KEY0:
    MOV     R3, #0x1
    ANDS    R3, R3, R1      // check for KEY0
    BEQ     CHECK_KEY1      // display "0"
    B       counter

CHECK_KEY1:
    MOV     R3, #0x2
    ANDS    R3, R3, R1      // check for KEY1
    BEQ     CHECK_KEY2
    B       main_loop

CHECK_KEY2:
    MOV     R3, #0x4
    ANDS    R3, R3, R1      // check for KEY2
    BEQ     IS_KEY3
    B       counter

IS_KEY3:
    MOV     R2, #0b01001111
    STR     R2, [R0]         // display "3"

END_KEY_ISR:
    BX LR

    .end
```