<1> 20 Points – In this question you see a main program calling a subroutine. The condition of the stack at position **A** is shown (the stack pointer SP is pointing at the current top of the stack, which contains some arbitrary number we denote by XXXXXX).

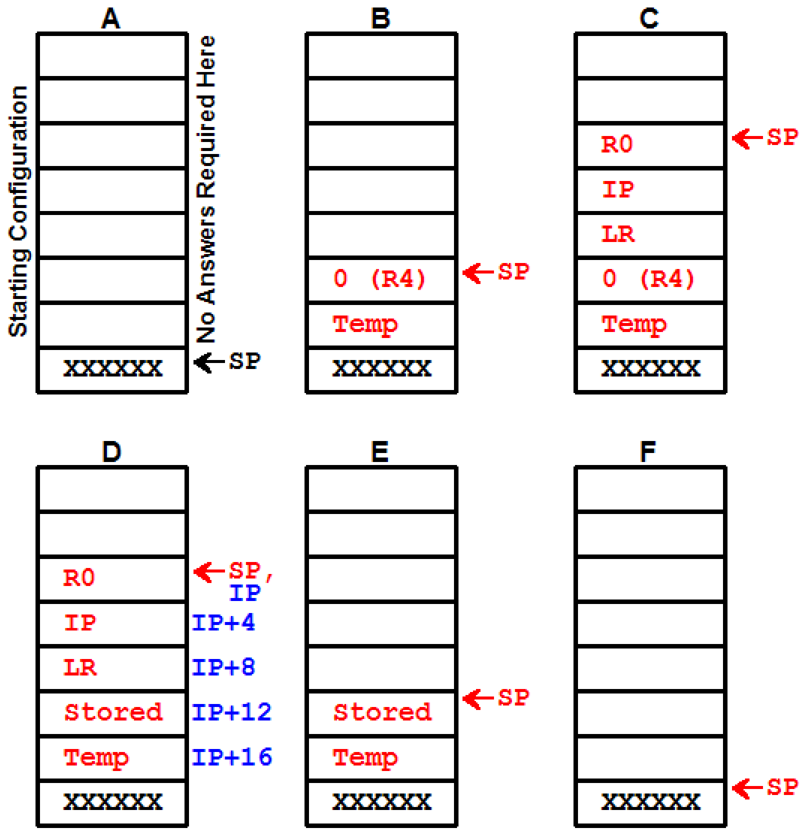
(1) **Trace the execution of the program from position A through position F**, and as you trace the program show the condition of the stack at positions **B**, **C**, **D**, **E**, and **F** by filling the appropriate values or symbols into the corresponding boxes (**do not** fill anything into the boxes for position **A**; it is given). For example, if the value being pushed onto the stack comes from variable Temp then write "Temp" into the correct box; if the value comes from a constant write the constant in the box, and if the value comes from a register write the name of the register in the box.

(2) **Fill in the blank for the LDR instruction** in SUB with the correct offset to load into R0 the first formal parameter to the subroutine (i.e., the value passed in from actual parameter Temp).

(3) **Fill in the blank for the STR instruction** in SUB with the correct offset to store R0 into the value returned by the subroutine (the second parameter slot in the call).

(4) **Yes or No:** as far as you can tell, is this subroutine completely transparent to all registers except the status flags? YES

```
(A)→LDR   R4,Temp
    STR   R4,[SP,#-4]!
    MOV   R4,#0
(B)→STR   R4,[SP,#-4]!
    BL    SUB
(E)→LDR   R0,[SP],#4
(F)→ADD   SP,SP,#4
────────────────────────
SUB STR   LR,[SP,#-4]!
    STR   IP,[SP,#-4]!
    STR   R0,[SP,#-4]!
    MOV   IP,SP
(C)→...
    LDR   R0,[IP,#_16_]
    ...
    STR   R0,[IP,#_12_]
(D)→...
    LDR   R0,[SP],#4
    LDR   IP,[SP],#4
    LDR   PC,[SP],#4
```



<2> 5 Points – I want to load the contents of integer variable `Temp` into floating point register `F0`. Describe in words the problem with the following code fragment. What will happen when it executes? Without changing the definition of `Temp`, fix the problem. Use any registers necessary.

```
          LDFS F0,Temp          LDR R0,Temp
                                FLT F0,R0
          ...

     Temp DCD  10
```

Integer variable `Temp` needs to be converted to a floating point number before it can go into `F0`, so it must be first loaded into an integer register and converted with the `FLT` instruction. Note that the code as written will actually load the integer 10 into `F0`, but the bit pattern corresponds to a small denormalized floating point number, very close to zero.

<3> 15 Points – In a high-level language such as Pascal, I declare an array of 32-bit integers with the statement **`Var A : Array [-10..10] Of Integer ;`** where the first element of the array is at `A[-10]` and the last element of the array is at `A[10]`. In translating this array declaration into ARM assembly language, I use the ARM directive **`A % 84`** to allocate and initialize to zero all 21 elements of the array (21 elements × 4 bytes per element = 84 bytes of memory).

(1) In pure algebra, write a mathematical expression that shows the mapping function from array index `X` to the offset in memory of the required item, relative to the base address of the array. Your answer should be a polynomial on `X` of the form: Offset ← *f(X)*. Tell me what *f* is.

Offset ← (X + 10) × 4
Offset ← (4X + 40)

The X+10 term is to normalize the Pascal array to a zero-based equivalent, then the multiplication by 4 is to convert the array index into a memory offset (since there are 4 bytes per array cell).

(2) Write the correct ARM assembly language statements to load into register `R0` the contents of `A[X]` where `X` is an integer variable stored in memory containing a number between -10 and 10. You do not need to perform range checking on `X`.

```
ADR R5,A                     ADR R5,A
LDR R1,X                     LDR R1,X
ADD R1,R1,#10                ADD R1,R1,#10
LDR R0,[R5,R1,LSL #2]        MOV R1,R1,LSL #2
                             LDR R0,[R5,R1]
```

(3) Write the correct ARM assembly language statements to load into register `R0` the contents of `A[7]`. By using a constant subscript (the 7) you are free to optimize your code in any manner you see fit.

```
ADR R5,A                    ADR R5,A
MOV R1,#7                   MOV R1,#17
ADD R1,R1,#10               LDR R0,[R5,R1,LSL #2]
LDR R0,[R5,R1,LSL #2]
                            ADR R5,A
ADR R5,A                    LDR R0,[R5,#68]
MOV R1,#68
LDR R0,[R5,R1]              ADR R5,A+68
                            LDR R0,[R5]
```

<4> 12 Points – In the subroutine call below, which of the parameters are ***call-by-value***, ***call-by-return***, ***call-by-value-return***, and ***call-by-reference***?

```
|                       ADR R0,Result        Reference
|                       STR R0,[SP,#-4]!
|   |                   LDR R0,Frog          Value-Return
|   |                   STR R0,[SP,#-4]!
|   |   |               SUB SP,SP,#4         Return
|   |   |   |           LDR R0,Toad          Value
|   |   |   |           STR R0,[SP,#-4]!
|   |   |   |   |       BL  SUBROUTINE
|   |   |   |           ADD SP,SP,#4
|   |   |               LDR R0,[SP],#4
|   |   |               STR R0,Newt
|   |                   LDR R0,[SP],#4
|   |                   STR R0,Frog
|                       ADD SP,SP,#4
```

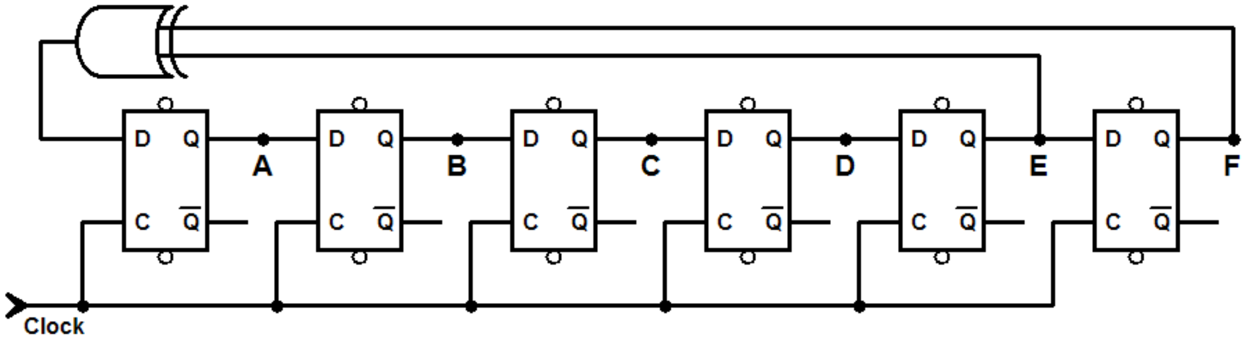
<5> 5 Points – In each of the following problems you are to multiply the contents of integer register `R0` by a constant value, in one instruction, without using any other registers, and without using any explicit multiplication instructions such as `MUL`, `MLA`, or `UMULL`. If the task cannot be accomplished in a single instruction, answer "Can't be Done".

```
(1)   R0 := R0 × 17      ADD R0,R0,R0,LSL #4

(2)   R0 := R0 × 15      RSB R0,R0,R0,LSL #4

(3)   R0 := R0 × 16      MOV R0,R0,LSL #4

(4)   R0 := R0 × -15     SUB R0,R0,R0,LSL #4

(5)   R0 := R0 × 10      Can't be done   ADD R0,R0,R0,LSL #2
                         in one line:    MOV R0,R0,LSL #1
```

<6> 8 Points – Short Answer – When is it necessary in a subroutine to use the IP register instead of the SP register for referencing parameters on the stack? When is it not necessary? How does recursion fit into your answers?

SP can be used instead of IP in any subroutine which does not use the stack after registers are pushed for transparency. If the subroutine uses the stack for temporary values then the stack pointer is modified and all references to variables in the current stack frame change offsets; use of IP allows stack modification without changing offsets into the stack frame. Recursive routines usually push parameters onto the stack before the internal calls can be executed; this essentially forces the use of IP to reference the stack frame.

<7> 12 Points – The following clocked circuit shows a 6-bit shift register where the left-most bit gets each new value from eXclusive-ORing the current values of the fifth and sixth bits. This forms a "63-step pseudo-random number generator" circuit. Assume that the current state of the shift register is `1 0 0 1 1 1`. Clock the circuit 6 times, and show the output state of the shift register after each clock pulse.

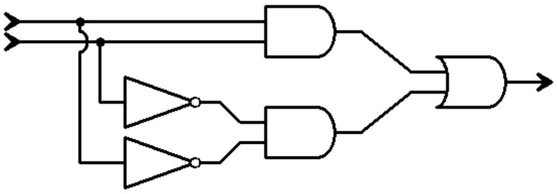


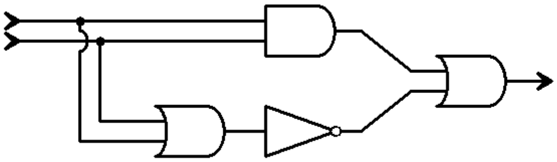
| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| Current Output: | 1 | 0 | 0 | 1 | 1 | 1 |
| After Clock #1: | 0 | 1 | 0 | 0 | 1 | 1 |
| After Clock #2: | 0 | 0 | 1 | 0 | 0 | 1 |
| After Clock #3: | 1 | 0 | 0 | 1 | 0 | 0 |
| After Clock #4: | 0 | 1 | 0 | 0 | 1 | 0 |
| After Clock #5: | 1 | 0 | 1 | 0 | 0 | 1 |
| After Clock #6: | 1 | 1 | 0 | 1 | 0 | 0 |

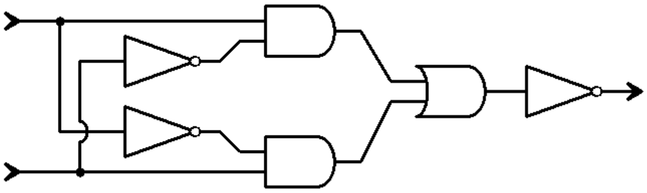
For each clock pulse the output of A will become the XOR of the previous values of E and F, and all other bits represent the previous state shifted right 1 bit (the old value of F is lost).

<8> 5 Points – Using only AND, OR, and NOT gates, draw a circuit below that performs the **exclusive-NOR** function for two inputs. Exclusive-NOR outputs 1 when the two inputs are the same, and outputs 0 when they are different.

Any of the following circuits will work. The first circuit outputs 1 if either AND-gate is triggered; the top one triggers if both inputs are 1, and the bottom one triggers through the NOT-gates if both inputs are 0. The two NOT-gates and the connected AND-gate performs the function of a NOR-gate. The second circuit is the same as the first, but it uses a different equivalent implementation of the NOR-gate using a NOT-gate and an OR-gate. In the third circuit the AND-gates are triggered if the inputs are different, so the output of the OR-gate is 1 if the inputs are different. This is an XOR-gate, which when run through the final NOT-gate becomes an XNOR-gate.

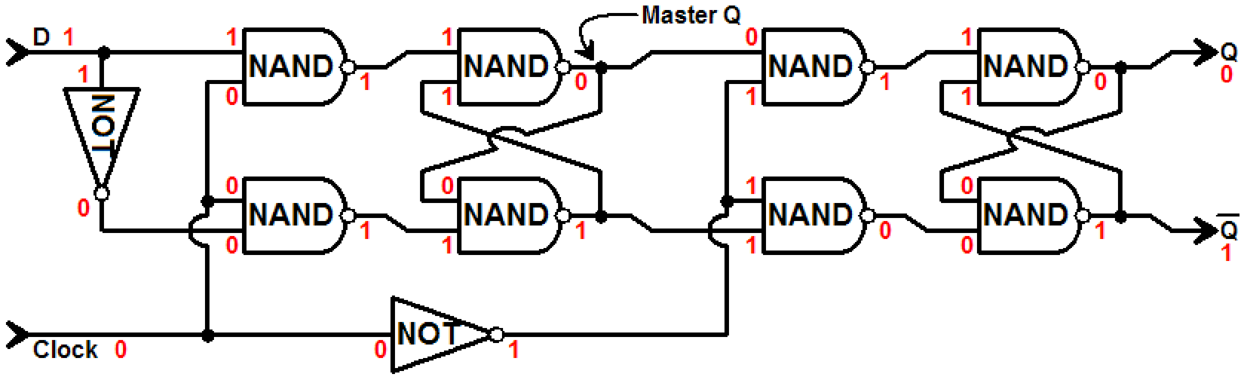




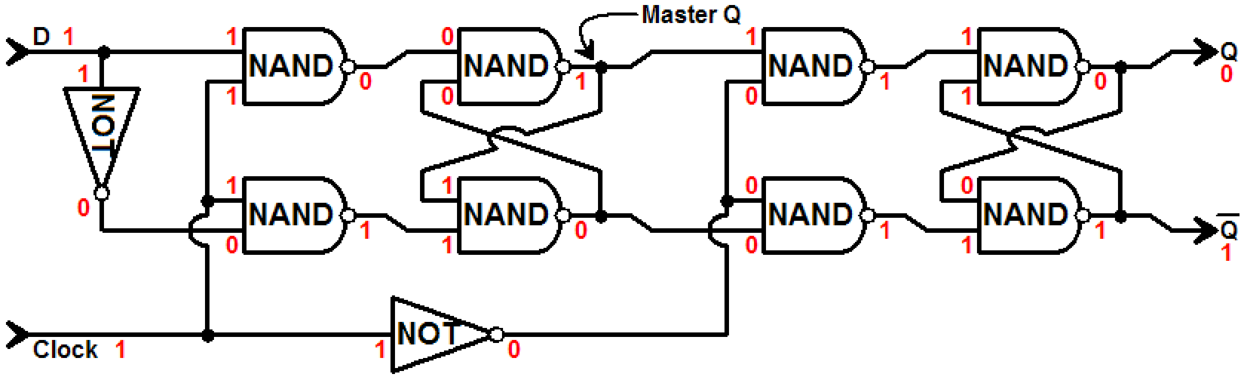


<9> 6 Points – In the following D-type flip-flop, assume that input **D = 1**, input **Clock = 0**, and internal point **Master Q = 0**. Fill in the outputs of ***every*** gate (yes, it is possible to do this from the given information).

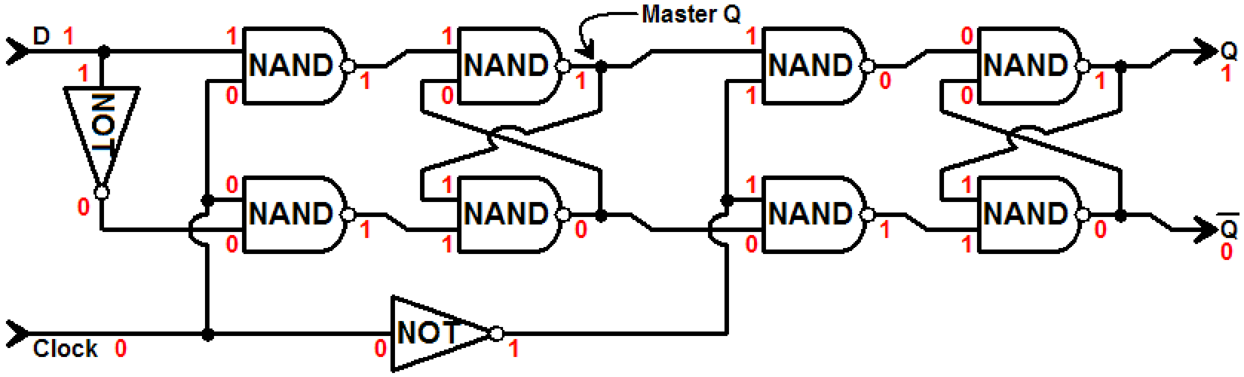
In the first view, you need to know that Master Q = 0 to force the rightmost six NAND-gates into a known state. After that, Master Q is free to change as needed.



<10> 6 Points – Based on the previous problem, leave D = 1 but **set Clock to 1**, and then fill in the outputs of ***every*** gate as before.



<11> 6 Points – Based on the previous problem, leave D = 1 and **clear Clock back to 0**, and then fill in the outputs of ***every*** gate as before.


D 1
Master Q
1
NAND
1
1
NAND
1
1
NAND
0
0
NAND
1
Q
1
1
NOT
0
0
1
1
0
0
1
1
NAND
1
1
NAND
0
1
NAND
1
1
NAND
0
Q̄
0
0
0
1
1
1
0
0
Clock 0
0
NOT
1