EECS 373 – Homework #3 & Practice Exam – Solution Key
Prabal Dutta

1. Fill-in-the-blank or circle the best answer
   a. R0; R14 or LR
   b. 16-bit and 32-bit
   c. 5 ns
   d. sometimes
   e. does not
   f. UART
   g. synchronous; addresses embedded in frames
   h. flip-flop
   i. grow quadratically
   j. .text (lec 2, slide 47)

2. Rewrite the add function in EABI assembly. Key things to consider are:
   a) caller (r0-r3)/callee save (r4-r11) registers
   b) parameter passing (r0-r3) and return (r0/r0+r1)conventions
   c) assigning variable registers: r4-r8, r10, r11 (r9 is special)
   d) dealing with the link register: LR (r15)

   Notes:
   - a is passed in r0 (this is the a pointer to a[0])
   - n is passes in r1
   - Must push/pop LR since add is not a leaf function
   - Need storage for local variables i and sum
   - Must save/restore r0-r3, if live, across call to printint(sum)
   - Don't need i after printint, so we can put it in r2 and skip save/restore
   - Need to either save/restore sum across printint call or put it in a callee save register. If we save and restore, that's (another) push/pop memory access (in addition to the LR), so we might as well just push/pop a callee save register (e.g. r4)
   - Must return sum in r0
   - Putting all this into an assembly function gives us:
     a in r0
     n in r1
     i in r2
     sum in r4
     sum in r0 (for call to printint)
     sum in r0 (for return value)

```
add:  push {r4, lr}    ; r0 = a, r1 = n
      mov  r2, #0      ; r2 = i = 0
      mov  r4, #0      ; r4 = sum = 0
loop: cmp  r2, r1      ; r2-r1
      bge  done        ; branch to done when i>=n
      ldr  r3, [r0, r2, lsl#2] ; load r3 with value of mem[r0+r2*4]
      add  r4, r4, r3  ; sum += a[i]
      add  r2, #1      ; i++
      b    loop        ; iterate loop
done: mov  r0, r4      ; prepare to pass sum to printint in r0
      bl   printint    ; call printint
      mov  r0, r4      ; restore r0 as printint might have clobbered it
      pop  {r4, pc}    ; restore r4 and load pc with original value of lr
```

3. ARM Assembly

   a. 0b 0100 0100 1101 1001 = 0x44D9
   b. The key here is:
      i) Remembering your addressing modes:
        [<Rn>, <offset>] - Offset: EA = <Rn> + <offset>
        [<Rn>, <offset>]! - Pre-idx: EA = <Rn> + <offset>; <Rn> += <offset>
        [<Rn>], <offset> - Post-indexed: EA = <Rn>; <Rn> += <offset>
      ii) Paying attention to whether ld and str size (byte, halfword, word)
        str - stores a word (32 bit)

```
    ldrb - loads a byte (8 bits)
    strh - stores a half word (16 bits)
    ldr - loads a word (32 bit)
  r1 = 0x00000FFE
  r2 = 0xDD00CCDD
  r3 = 0x0000000F
c. 0x1000 = 0b 0001 0000 0000 0000 = 2^12 = 4096 (msb = 0, so pos number)
   0xfffc = 0b 1111 1111 1111 1100 (msb = 1, so neg number)
          = 0b 0000 0000 0000 0011 invert the bits
          + 0b 0000 0000 0000 0001 add one
            ======================
          - 0b 0000 0000 0000 0100 = -(2^2) = -4
   0xffff = 0b 1111 1111 1111 1111 (msb = 1, so neg number)
          = 0b 0000 0000 0000 0000 invert the bits
          + 0b 0000 0000 0000 0001 add one
            ======================
          - 0b 0000 0000 0000 0001 = -(2^0) = -1
```

4. Memory mapped I/O.

a.
```
// Declare reg and add 7 to REG_FOO's value
uint32_t* reg = (uint32_t*)REG_FOO;
*reg += 7;
```

b. Sketch glue logic to interface DFF to APB

```
           +----*
   PSEL ---+     \
           |      \
 PWRITE --O| AND   )--------------------+
           |      /                     |
PENABLE ---+     /                      |
           +----*     +---------+      |\|
                      |         |      | \
        PWDATA[0] +---+ D     Q +------+  +----- PRDATA[0]
                      |         |      | /
                      |   DFF   |      |/
                      |         |
             PCLK +---+ CLK     |
                      |   ENA   |
           +----*     +----+----+
   PSEL ---+     \         |
           |      \        |
 PWRITE ---+ AND   )-------+
           |      /
PENABLE ---+     /
           +----*
```

Note: The APB *spec* doesnt say anything about whether tristate buffers are needed (and if your bus is multiplexed, it's not). But, to demonstrate the importance of only driving the bus when you (and you alone) are being read, it's useful to add the tristate driver.

Note: Drawn using asciiflow.com

5. Serial busses

a.
```
v(t) = Vcc (1 - e^(-t/T)
t = -R*C*ln(1-v(t)/Vcc) [at (v(t) = 2.4V)
t = -(10e3)(50e-12)(ln(1-2.4/3.0))
t = 0.8 us
```

b. There are 8 data bits for 20 bits transmitted, giving 40% data throughput. With four different devices, each will get 1/4 of the this overall throughput, giving each device 10% of the data throughput, of 400 kHz * 1 bit/Hz * 10% = 40 kbps/device.

6. Logic Design

N = fin/fout = 100 MHz/5MHz = 20
M = N*DC = 20 * 30% = 6

```
                                        +--------+        +---------+
                                        |        |        |         |
            +-----------+       +---/---+ [ <6 ] +--------+ D     Q +--> CLKOUT
            |           |       |   5   |        |        |         |
            |           |       |       +--------+        |   DFF   |
            |   5-bit   | D4:D0 |               CLKIN >---+ CLK     |
CLKIN >-----+           +---/---+                         |         |
            |  Counter  |   5   |                         +---------+
            |           |       |       +--------+
            |    RST    |       |       |        |
            +-----+-----+       +---/---+ [ =19] +---+
                  |                 5   |        |   |
                  |                     +--------+   |
                  |                                  |
                  +----------------------------------+
```

Where:

[<6] = ~D4 & ~D3 & ~(D2 & D1)
[=19] = D4 & ~D3 & ~D2 * D1 * D0

Note: Drawn using asciiflow.com

7. AHB Write.

```
              _____       _____       _____       _____       _____
        FCLK |     |_____^     |_____^     |_____^     |_____^     |____
              _ ___________ ___________ ___________ ___________ _________
 HADDR[31:0] |_X_20000604__X___________X___________X___________X_________
              _ ___________ ___________ ___________ ___________ _________
      HWRITE |_/           \___________X___________X___________X_________
              _ ___________ ___________ ___________ ___________ _________
HWDATA[31:0] |_X___________X_76543210__X___________X___________X_________
              _ ___________ ___________ ___________ ___________ _________
 HREADY(OUT) | V           V           V           V           V
```

8. Enable Interrupts. The challenge here is to map the interrupt number x to the particular memory word and bit offset. First, note that a word has 32 bits, so interrupts 0-31 map to the word at 0xE000E1000, and interrupts 32-63 map to the word at 0xE000E1004, and so on. So, to determine the "word," we can divide x by 32 (or, equivalently, shift if by 5 bits). To get the bit position within that word, we actually need to extract the bottom 5 bits of x which contain the a value in the range 0-31 and turn that 5-bit value into a bit mask with the appropriate bit set. Finally, we just need to set the bit that corresponds to that value by writing the mask value into the appropriate register. Note one really important thing: we *don't* need to read, update, and write back the ISER registers because the only thing that matters is writing 1s to enable interrupts (i.e. writing 0s does *not* disable interrupts!). The following code snippet does all of the following.

```
void enable_interrupts(int x) {
  uint32_t base = 0xE000E100;
  uint32_t idx = x >> 5;
  uint32_t bit = x & 0x1f;
  uint32_t mask = 1 << bit;
  *((uint32_t*)(base+4*idx)) = mask;
}
```