Name: ____________________

First Name: ____________________

Student ID: ____________________

## 2nd session examination

# Design of Digital Circuits WS2015

## (252-0014-00S)

Srdjan Capkun, Frank K. Gürkaynak

Examination Rules:

1. Written exam, 90 minutes in total.
2. No books, no calculators, no computers or communication devices. Six pages of hand-written notes are allowed.
3. Write all your answers on this document, space is reserved for your answers after each question. Blank pages are available at the end of the exam.
4. Put your Student ID card visible on the desk during the exam.
5. If you feel disturbed, immediately call an assistant.
6. Answers will only be evaluated if they are readable.
7. Write with a black or blue pen (no pencil, no green or red color).
8. Show all your work. For some questions, you may get partial credit even if the end result is wrong due to a calculation mistake.
9. Points for every part are indicated in the exam. They should correspond to the expected difficulty of the questions. Consider this when allocating your time.

| Question: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Total |
|---|---|---|---|---|---|---|---|---|
| Points: | 5 | 18 | 12 | 10 | 10 | 10 | 10 | 75 |
| Score: | | | | | | | | |

*This page intentionally left blank*

1. (a) (4 points) For the following four numbers given in decimal or hexadecimal notation, write the corresponding binary number using the indicated format.

| | |
|---|---|
| $(-11)_{10}$ using 6-bit two's complement: | $(11\,0101)_2$ |
| $(51)_{10}$ using 6-bit unsigned: | $(11\,0011)_2$ |
| $(-17)_{10}$ using 6-bit sign magnitude: | $(11\,0001)_2$ |
| $(39)_{16}$ using 6-bit unsigned: | $(11\,1001)_2$ |

(b) (1 point) What are the problems with the sign/magnitude representation of binary numbers, why are they not used more often than two's complement representation?

**Solution:**
Zero is represented twice (+0, -0), and more importantly subtraction is more complex when you have sign/magnitude. In two's complement the same circuit can handle both positive and negative numbers exactly the same way.

2. A circuit has four inputs ($A$, $B$, $C$, $D$) and two outputs ($E$, $T$).

- Output $E$ should be '1' if there are **equal number** of '1's and '0's in the input.
- Output $T$ should be '1' if **two or more** of the inputs are '1'

(a) (2 points) Complete the following truth table

| **Inputs** | | | | **Outputs** | |
|---|---|---|---|---|---|
| $A$ | $B$ | $C$ | $D$ | $E$ | $T$ |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 |

(b) (3 points) Write a Boolean equation for $E$ using either SOP or POS representation. (*Hint: use a Karnaugh map*)

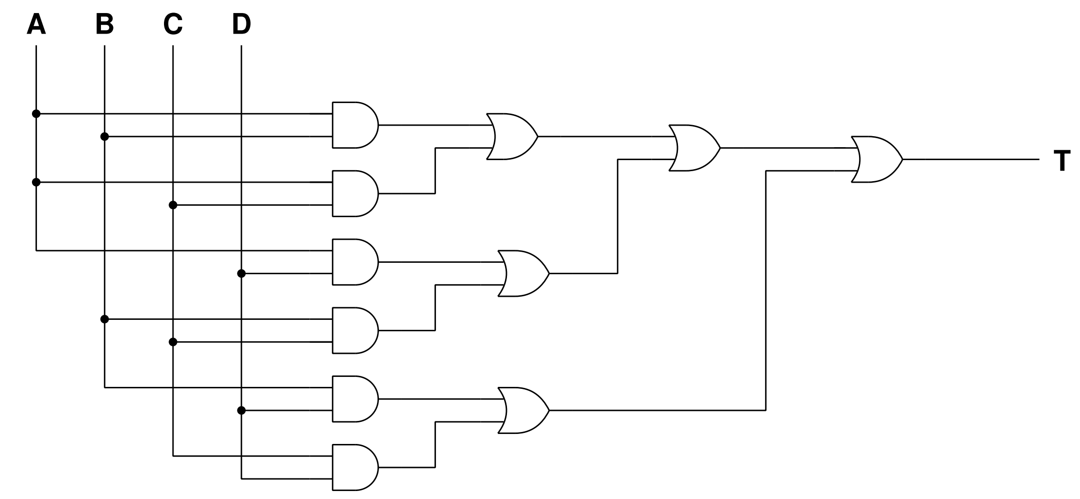
**Solution:** One possible solution is:
$E = (\overline{A} \cdot \overline{B} \cdot C \cdot D) + (\overline{A} \cdot B \cdot \overline{C} \cdot D) + (\overline{A} \cdot B \cdot C \cdot \overline{D}) + (A \cdot \overline{B} \cdot \overline{C} \cdot D) + (A \cdot B \cdot \overline{C} \cdot \overline{D}) + (A \cdot \overline{B} \cdot C \cdot \overline{D})$

(c) (3 points) Write a Boolean equation for $T$ using either SOP or POS representation. (*Hint: use a Karnaugh map*)

**Solution:** One possible solution is:
$T = (A \cdot B) + (A \cdot C) + (A \cdot D) + (B \cdot C) + (B \cdot D) + (C\dot{D})$

(d) (3 points) Draw a gate level diagram implementing $T$ using **only two-input** AND, OR gates and/or inverters.

**Solution:**



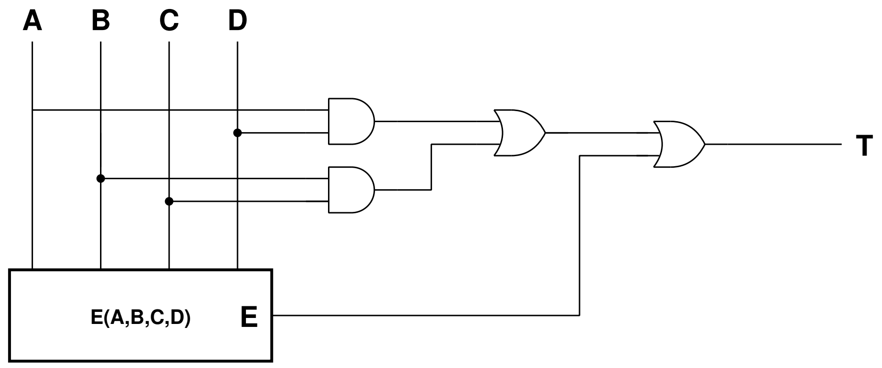
(e) (2 points) Assuming that the propagation of all 2-input gates and inverters is 100 ps and the contamination delay is 50 ps, what is the critical path (longest path), and the shortest path of the circuit you have drawn in Question 2d?

**Solution:**
For the solution given in Question 2d, the critical path goes through 1 AND gate and 3 OR gates = 4 propagation delays = 400ps.

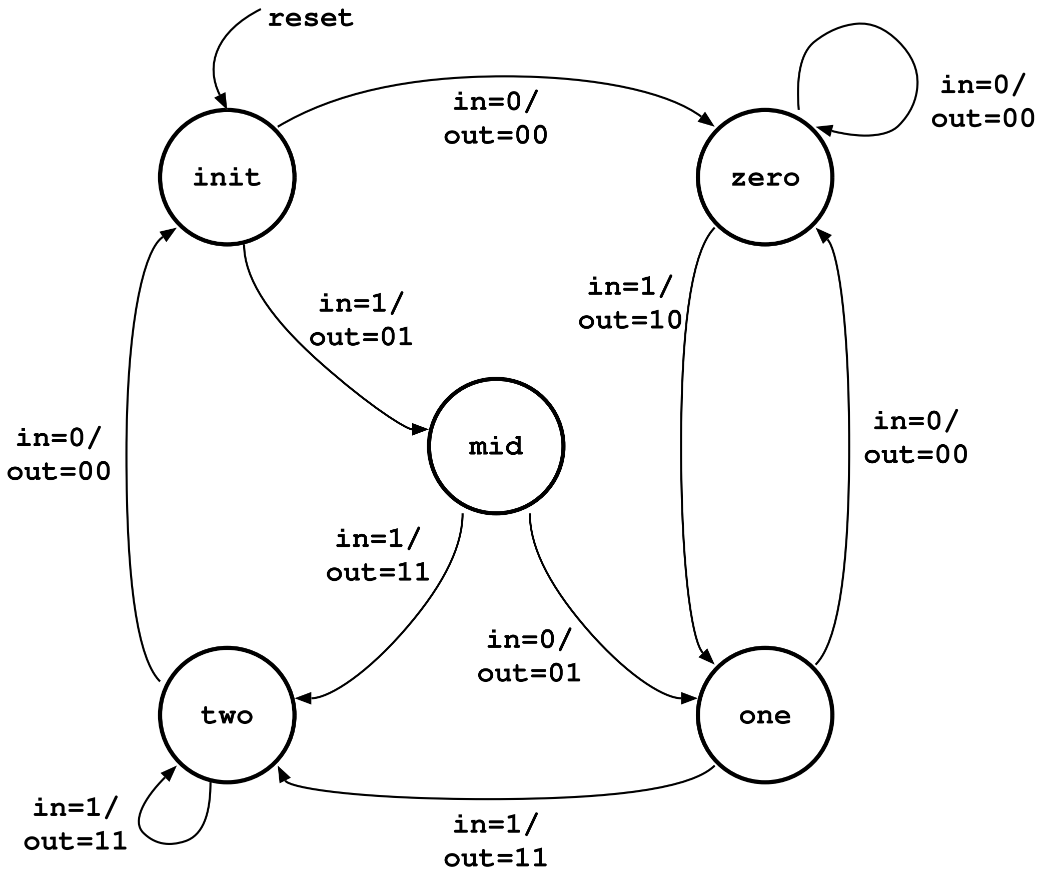
The shortest path goes through one AND gate and two OR gates = 3 contamination delays = 150ps.

(f) (5 points) Note that the function $T$ contains $E$ (meaning $T$ is '1' for every input that results in a '1' for $E$). Assume that you already have a small circuit that implements $E$, design a simpler Boolean function that implements $T$ by using the output of $E$ and draw the gate level schematic once again only using $E$, 2-input AND/OR gates and inverters.

**Solution:** One of the simpler solutions is: $T = (A \cdot D) + (B \cdot C) + E$



3. In this question you will be given the state transition diagram for a Finite State Machine (FSM). The FSM has a two-bit output "`out[1:0]`" as well as a single bit input "`in`".



(a) (2 points) Is this a Mealy or Moore type of FSM, briefly explain why?

**Solution:** This is a Mealy type of FSM, as the output depends not only on the state but also on the present value of the input.

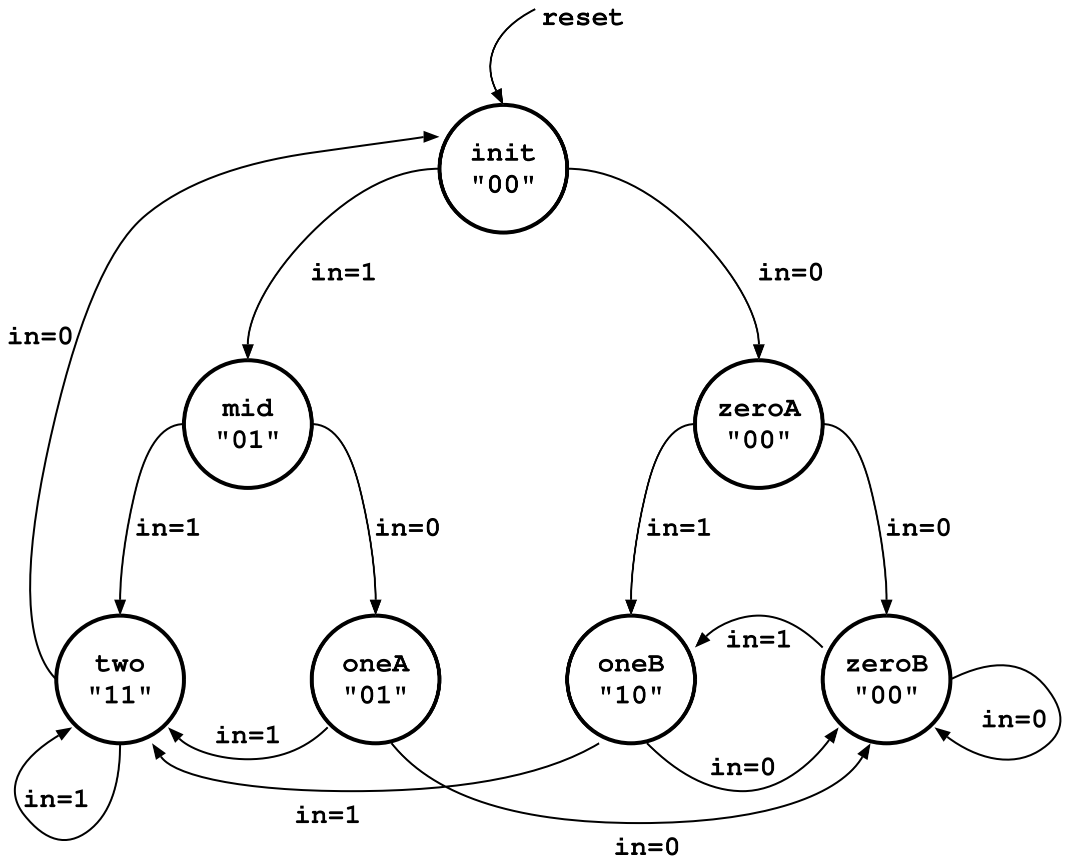
(b) (5 points) Using the following state encoding table, complete the state transition and output table on the following page.

| State | Encoding | | |
|---|---|---|---|
| Name | $S_2$ | $S_1$ | $S_0$ |
| `init` | 1 | 1 | 1 |
| `mid` | 1 | 0 | 0 |
| `zero` | 0 | 0 | 0 |
| `one` | 0 | 0 | 1 |
| `two` | 0 | 1 | 0 |

| **Present State** | | | **Input** | **Next State** | | | **Outputs** | |
|---|---|---|---|---|---|---|---|---|
| $P_2$ | $P_1$ | $P_0$ | $in$ | $N_2$ | $N_1$ | $N_0$ | $out_1$ | $out_0$ |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |

(c) (5 points) Draw a new state transition diagram for this FSM using a different type (if the original was a Moore, then draw a Mealy type or vice versa) that has the same functionality.

**Solution:** This is one possible solution. Note that states `ZeroA` and `ZeroB` can be merged, however this organization is slightly more regular, and shows the thinking behind the original FSM.



4. (10 points) There are four Verilog code snippets in this section. For each code, first state whether or not there is a mistake. If there is a mistake explain how to correct it. *Note: Assume that the behavior as described, is correct*

(a)

```
module mux2 ( input [1:0] i,  output s, input z);
  assign s= (z) ? i[1]:i[0];
endmodule

module one (input [3:0] data, input sel1, input sel2, output z );
  wire [1:0] temp ;

  mux2 i0 (data[1:0], sel1, temp[0]);
  mux2 i1 (data[3:2], sel1, temp[1]);
  mux2 final (temp, sel2, z);
endmodule
```

**Solution:** This code is not correct. It uses an ordered instantiation template where the ordering of the pins should correspond to the order they are declared. This itself is not wrong, but the *input* signals on the second position (sel1,sel2) connect to the *output* of the instance *mux*. Either the ordering of *mux2* needs to be changed, or the ordering of the elements the instantiation. The better option would be to use a *connect by name*.

(b)

```
module two (input [3:0] a, input [0:3] b, output reg [3:0] z);
 always @ ( *)
     if (a == 0)
       z={b[0],b[1],b[2],b[3]};
     else
       z=a+b;
endmodule
```

**Solution:**

The code is syntactically correct. There is a lot of unnecessary code: using one input as [0:3] and the other as [3:0] is not the smartest choice, the entire code could be written as *assign z=a+b*, but syntactically the code is correct.

(c)

```
module three (clk, rst, a, b, c, z);
  input a,b,c,clk,rst;
  output reg z;
  reg q;

  always @ (*)
    begin
      q <= a ^ b;
      if (c) q <= ~(a^b);
    end
  always @ (negedge clk)
     if (rst) z= 1'b0;
     else     z= q;
endmodule
```

**Solution:** The code is syntactically correct. Once again, it is a bit unconventional. One always block use blocking statements and the other non-blocking statements. Both would work, although it would probably be more efficient to write them otherwise.

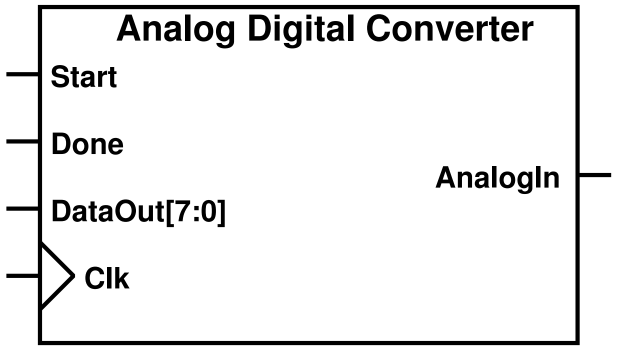
(d)

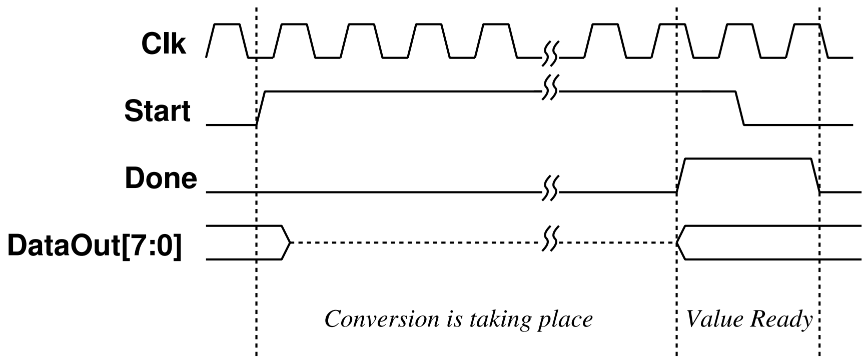
```
module four (input [2:0] sel, output reg [5:0] z);
 case (sel)
   0: z = 6'b00_0000;
   1: z = 6'b00_0001;
   2: z = 6'b00_0011;
   3: z = 6'b00_0111;
   4: z = 6'b00_1111;
   5: z = 6'b01_1111;
   6: z = 6'b11_1111;
   default: z= 6'b00_0000;
 endcase
endmodule
```

**Solution:** This code is not correct. The case statement is a sequential statement that needs to be within an always statement.

For the following questions 5 and 6 we will attempt to add a simple peripheral to a standard single cycle MIPS processor. In question 5 you will make the modifications to the hardware and in question 6 you will write MIPS code to help you access this peripheral. *Note that, you can answer the questions independently. i.e. it is possible to answer 6 even if you have not answered 5.*

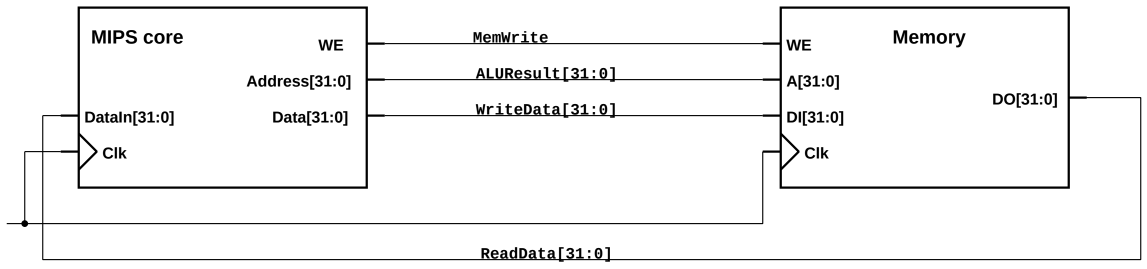
The peripheral we want to add is a simple Analog to Digital converter (ADC) as shown below.





- The ADC has to be initiated by setting the `Start` signal to '1'.
- The ADC will then start converting the `AnalogIn` signal into a digital value. This could take several clock cycles to finish. You do not need to know in advance exactly how many clock cycles the conversion will require.
- As soon as the conversion is done, the ADC will pull the `Done` signal to '1'.
- At this point the analog input has been converted to an 8-bit value that will be available as the `DataOut` signal and can safely be read back.
- The `DataOut` will stay valid as long as the `Start` signal is 1. Once you have recovered the data, you can complete the action by resetting the `Start` signal back to '0'.
- As soon as you reset the `Start` signal the ADC will also lower the `Done` signal back to '0' and the ADC will be ready to make another conversion.

5. (10 points) Seen below is a simplified block diagram of a single-cycle MIPS architecture showing the core to memory interface as covered in the class and lab exercises.



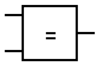
The ADC is connected to the system through a memory mapped interface:

- The MIPS processor can write to `Start` pin of the ADC.
- The MIPS processor can read the `Done` pin and the 8-bit `DataOut` value from the ADC.

The three pins of the ADC are mapped to the following memory addresses:

```
Start   0xFFFF FF00
Done    0xFFFF FF40
DataOut 0xFFFF FF80
```

Make the necessary connections and if necessary add multiplexers, registers, flip-flops so that the processor can read and write to the ADC just as reading/writing to the memory without interfering with the memory. In case you need to compare a value with a memory address, you may use a comparator block that takes two 32-bit inputs and outputs 1 if they match, as shown below.

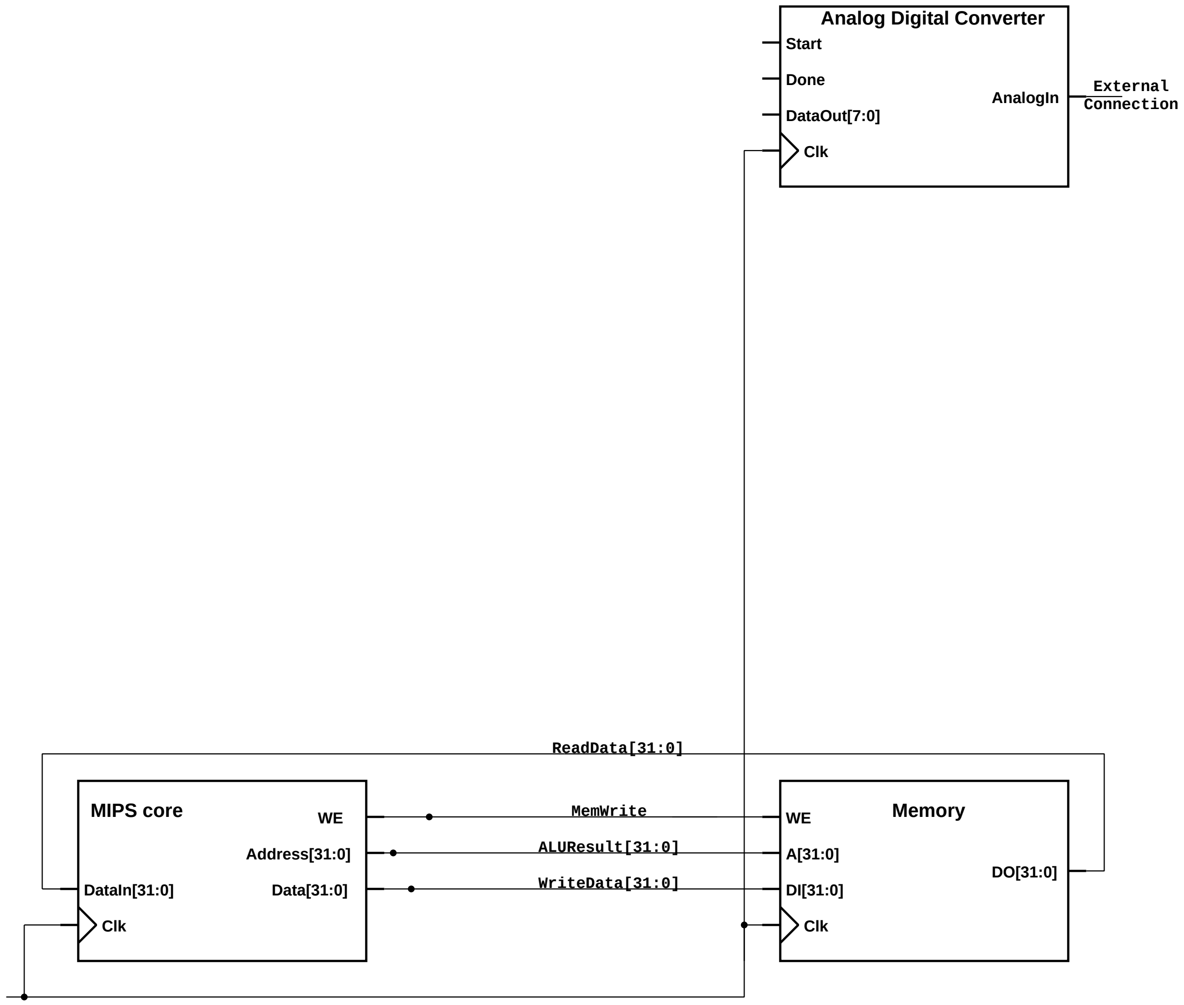


More specifically, complete the following two functions:

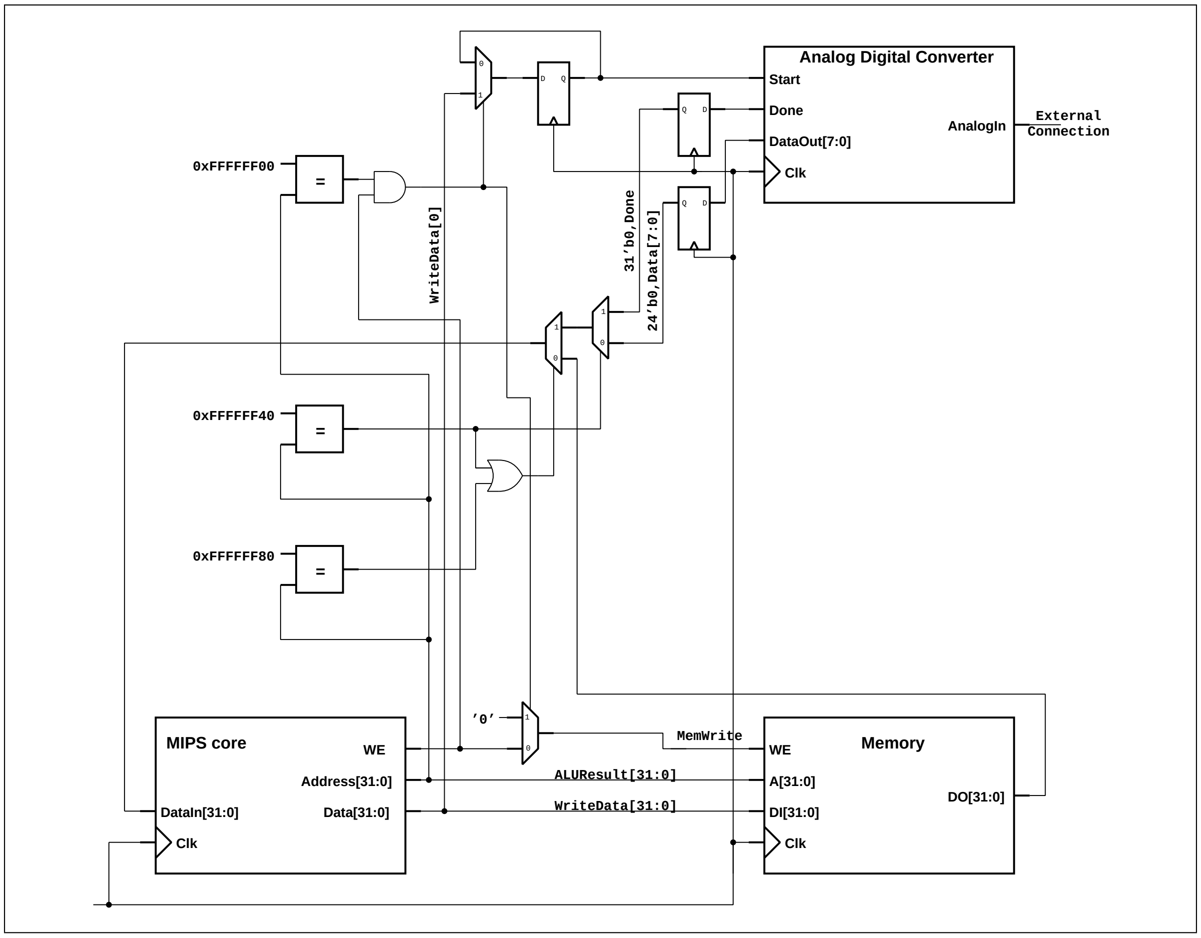
(a) The input for `Start`

(b) The reading of the output for `Done` and `DataOut`

Please draw on the schematic on the next page.


Analog Digital Converter
Start
Done
DataOut[7:0]
Clk
AnalogIn
External Connection
ReadData[31:0]
MIPS core
WE
Address[31:0]
DataIn[31:0]
Data[31:0]
Clk
MemWrite
ALUResult[31:0]
WriteData[31:0]
Memory
WE
A[31:0]
DI[31:0]
Clk
DO[31:0]

**Solution:**


Analog Digital Converter
Start
Done
DataOut[7:0]
Clk
AnalogIn
External Connection
0xFFFFFF00
=
WriteData[0]
31'b0,Done
24'b0,Data[7:0]
0xFFFFFF40
=
0xFFFFFF80
=
'0'
MemWrite
MIPS core
WE
Address[31:0]
DataIn[31:0]
Data[31:0]
Clk
ALUResult[31:0]
WriteData[31:0]
Memory
WE
A[31:0]
DI[31:0]
DO[31:0]
Clk
D
Q

6. In this question, we will write subroutines to access the ADC we have added to our MIPS system. As before, the three pins are mapped to the following addresses:

```
Start   0xFFFF FF00
Done    0xFFFF FF40
DataOut 0xFFFF FF80
```

(a) (5 points) Write a subroutine called `readADC` that will

1. Activate the ADC
2. Wait until the conversion is over
3. Read the Value converted by the ADC and return it to the calling program.

**Solution:**

```
readADC:   addi $t1, $0, 1         # We need '1' in one register
                                   # Optionally check if 'Done'=0
           sw   $t1, 0xff00($0)    # Activate ADC by writing '1' to 'Start'
waitloop:  lw   $t2, 0xff40($0)    # Read 'Done'
           beq  $t2, $0, waitloop  # if 'Done'= 0 keep waiting
           lw   $v0, 0xff80($0)    # Read the 'DataOut'
           sw   $0,  0xff00($0)    # Set 'Start' back to 0
           jr   $ra                # Return to $ra
```

(b) (5 points) Now write a function `averageADC` that will collect 256 values from the ADC by calling the subroutine you have written above. Average these 256 values and return the result back to a calling subroutine.

**Solution:**

```
averageADC : addi $s0, $0, 0       # this will be our count
             addi $sp, $sp, -4     # make room on stack
             sw   $ra, 0($sp)      # push the return address
             addi $s2, $0, 256     # end count for the loop
    loop :   jal  readADC          # call subroutine we have written
             add  $s0, $v0, $s0    # accumulate values in $s0
             addi $s2, $s2, -1     # reduce count
             beq  $s2, $0, fin     # jump to loop end
             j    loop
      fin :  sra  $v0, $s0, 8      # divide by 256 for averaging
             lw   $ra, 0($sp)      # read back the return address
             addi $sp, $sp, 4      # restore stackpointer
             jr   $ra              # return to $ra
```

7. For this question consider the following MIPS assembly code.

```
      lui  $s0, 0x2000   # Initialize addresses
      ori  $s0, 0x0000   # First vector
      lui  $s1, 0x1001
      ori  $s1, 0x5000   # Second vector
      lui  $s2, 0x1001
      ori  $s2, 0x6000   # End address
      addi $s3, $0, 0    # initialize
loop: lw   $t0, 0($s0)   # Read vector 1
      lw   $t1, 0($s1)   # Read vector 2
      add  $t2, $t0, $t1 # Add both to $t2
      add  $s3, $s3, $t2 # Accumulate at $s3
      addi $s0, $s0, 4   # increment vector 1
      addi $s1, $s1, 4   # increment vector 2
      beq  $s1, $s2, end # end reached?
      j    loop          # no, loop.
end:
```

This code is being executed on a MIPS processor that executes all instructions in a single cycle, except for memory accesses (i.e. `lw` and `sw` instructions) which take 20 clock cycles. The processor will be waiting during this time.

(a) (2 points) How many clock cycles (approximately) does it take to execute the above mentioned program?

**Solution:** There are 7 instructions before the loop. The loop is executed 1024 times. There are 6 instructions that take 1 clock cycle and 2 `lw` instructions that take 20 cycles so the total is:

```
Total = 7 + 1024 * (2*20 + 6)
      = 47111
      ~ 45000 - 50000 cycles
```

(b) (4 points) We want to speed up the operation by using a cache memory. By using a cache with a capacity of 256 words, we can achieve 1 clock cycle memory accesses for data that is within the cache.

How fast would the above program execute if a direct mapped cache with block size of one word is used?

**Solution:**

In principle, all memory accesses would be cache misses, no data is read twice (no temporal locality), and block size is one word so no spatial locality can be exploited. In fact each `lw` instruction will be longer, as now we have one cycle for the cache miss (21 instead of 20 cycles).

```
  Total = 7 + 1024 * (2*21 + 6)
        = 49159
        ~ 50000 cycles
```

(c) (4 points) What would be the best possible cache configuration with a capacity of 256 words for running this program?

**Solution:**

There are two independent memory locations being read. So the cache needs to be 2-way set associative so that the consecutive `lw` accesses on `$s0` and `$s1` do not invalidate the data.

Since there is no temporal locality (data is not read again in this program), spatial locality is the only way to speed up. The block size should be as large as possible. Since we have 2-ways, the largest block size we can have is 128 words.

In this case we would have 2 cache misses for each way every 128 memory accesses that will require 21 cycles each. All other accesses will have a single cycle access time.

```
  Total = 7 + 8 * (2 * 21 +6) + 1016 * 8
        = 8519
        ~ 8000-9000 cycles
```