Name: ______________________________

First Name: __________________________

Student ID: __________________________

2nd session examination

Design of Digital Circuits SS2014

(252-0014-00S)

Srdjan Capkun, Frank K. Gürkaynak

Examination Rules:

1. Written exam, 90 minutes total.
2. No books, no calculators, no computers or communication devices. Six pages of handwritten 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.

Question:	1	2	3	4	5	6	7	Total
Points:	5	12	8	10	17	15	8	75
Score:								

This page intentionally left blank

1. (a) (3 points) Through a digital communication channel you have received the following information in hexadecimal format:

`0x002E 61A7 E82F`

How many bits of information have been transmitted?

Solution: There are 12 hex digits, each hex digit contains 4 bits, in total 48 bits

Using the table below, for each byte enter the corresponding binary information:

Hex	Binary							
`0x00`	0	0	0	0	0	0	0	0
`0x2E`	0	0	1	0	1	1	1	0
`0x61`	0	1	1	0	0	0	0	1
`0xA7`	1	0	1	0	0	1	1	1
`0xE8`	1	1	1	0	1	0	0	0
`0x2F`	0	0	1	0	1	1	1	1

(b) (2 points) How can you express decimal 171 and -40 using two's complement binary representation?

Solution: 171 $==$ **`0 1010 1011`**

Note that you have to have a leading zero, otherwise it would be a negative number (twos complement -85). Any number of leading zeroes is fine, but there should be at least one.
-40 $==$ **`101 1000`**

Similarly you need to have at least one leading '1' here as well.

2. For this question, use the following truth table for a 4-input logic function called Z.

Input				**Output**
A	B	C	D	Z
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	X
0	1	0	0	1
0	1	0	1	X
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	X
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

(a) (1 point) What is the meaning of X in this truth table?

Solution: The output value is not important for the functionality of the circuit. It can be taken as '0' or '1' to simplify the equations

(b) (6 points) A friend of yours has determined the following Boolean equation for Z:

$$Z = (B + D) \cdot (\overline{B} + \overline{C}) \cdot (A + \overline{C}) \cdot (\overline{A} + B + D) \cdot (A + B + \overline{C}) \cdot (A + C + \overline{D})$$

But he is not sure if this is correct. Verify whether or not the given equation matches the truth table given above. Is there something that your friend could have done better?

Solution:
The equation is not correct. You can see this if you mark the minterms on the truth table for each equation. The following are the problems:

- $(A + \overline{C})$ is redundant if the X there was chosen as '1'
- $(\overline{A} + B + D)$ is redundant.
- $(A + B + \overline{C})$ is redundant, but $(\overline{A} + \overline{B} + C)$ is missing
- $(A + C + \overline{D})$ is plain wrong. It covers 1 and X. Should not be there

The X values have not been optimally used, this results in a more complex equation, there are more 0s than 1s, so a SOP form would probably be better, in addition there are redundant terms, the equation is not simplified

(c) (5 points) Derive your own *optimized* boolean equation corresponding to the same truth table using *sums-of-products* form. Try to take advantage of the 'X' values to minimize the equation as much as possible. (*Hint: use a Karnaugh map*)

Solution: If you take all the Xs as '1', you can derive:
$Z = (\overline{B} \cdot D) + (\overline{C} \cdot D) + (\overline{A} \cdot B \cdot \overline{C})$

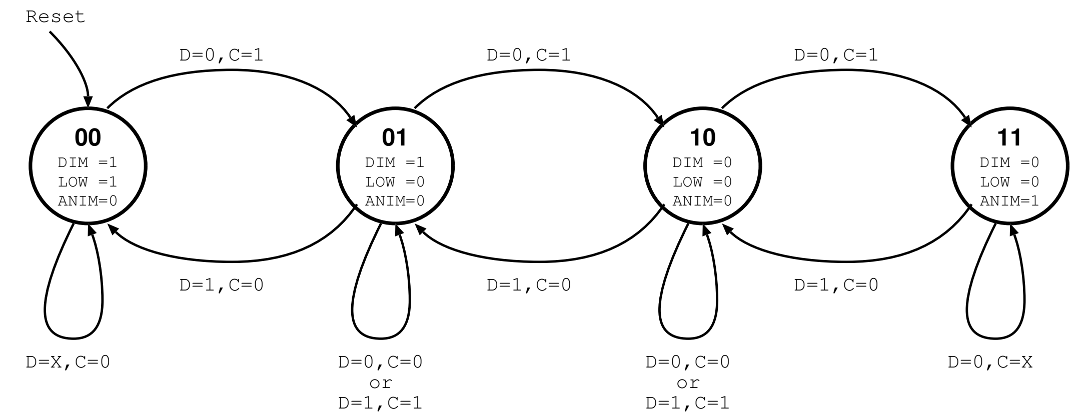
3. In this question you will be asked to draw design the FSM for a power saving control module of a mobile device.

   (a) (4 points) We want to design the power saving control module of a mobile device.

   - There are two inputs: C (charging) and D (discharging)
   - There are four power levels (0,1,2,3) for the device
   - When both inputs (C, D) are the same the power level does not change
   - When only C is active, power level increases until the last level (3) is reached
   - When only D is active, power level decreases until the lowest level (0) is reached
   - There are 3 outputs: DIM (dimmer), LOW (low power), $ANIM$ (animations)
   - DIM is active at power level 1 or lower
   - LOW is active at power level 0 only and signals that we are at low power
   - $ANIM$ is active at power level 3 only and enables power hungry animations on the device
   - the reset state corresponds to power level 2.

   Draw the State Transition Diagram for a Moore type FSM that implements this state machine

Solution:



(b) (4 points) Using the State Transition Diagram, complete the following table for both State Transitions and the outputs.

Present State	**Inputs**		**Next State**	**Outputs**		
name	C	D	name	DIM	LOW	$ANIM$
`00`	0	X	`00`	1	1	0
`00`	1	0	`01`	1	1	0
`00`	1	1	`00`	1	1	0
`01`	0	0	`01`	1	0	0
`01`	0	1	`00`	1	0	0
`01`	1	0	`10`	1	0	0
`01`	1	1	`01`	1	0	0
`10`	0	0	`10`	0	0	0
`10`	0	1	`01`	0	0	0
`10`	1	0	`11`	0	0	0
`10`	1	1	`10`	0	0	0
`11`	0	0	`11`	0	0	1
`11`	0	1	`10`	0	0	1
`11`	1	X	`11`	0	0	1

4. (10 points) There are four Verilog code snippets in this section. Some of these codes have a problem with the **syntax**. 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 one (input sel, input [1:0] data, output z);

        always @ (*)
            begin
              assign z= (sel) ? data[1]:data[0];
            end
endmodule
```

Solution: This code has mistakes. sequential assignments do not start with `assign`, these are reserved for combinational statements. In addition, if `z` was assigned in an `always` statement, it should have been declared as `reg`.

(b)

```
module two (input [1:0] sel, output reg [7:0] z);

  always @ (sel)
    if      (sel == 2'b01) z=8'b01010101;
    else if (sel == 2'b10) z=8'hAA;
    else                   z=8'd0;

endmodule
```

Solution: This code is correct, it is OK to write constants using different representations.

(c)

```
module three ( input [3:0] data, input sel1, input sel2, output z);

  wire [1:0] m;   // actual exam had (a typo) : wire m;

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

  mux2 i0 (.i(data[1:0]), .sel(sel1), .z(m[0]) );
  mux2 i1 (.i(data[3:2]), .sel(sel1), .z(m[1]) );
  mux2 i2 (.i(m), .sel(sel2), .z(z) );

endmodule
```

Solution: This code has mistakes. The sub module `mux2` should be defined outside the module `three` and not be part of it. Only the instantiations are part of the code. In the actual exam, there was a typo here, and `m` was declared as a single bit. Students that pointed out this typo got full points. This is the intended version.

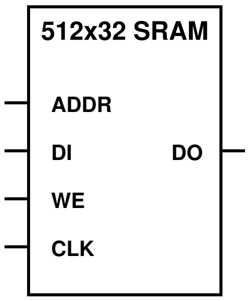
(d)

```
module four (input [3:0] data, input [1:0] sel, output reg [3:0] z);

  always @ (data, sel)
      z = data;
      if (sel[0])        z = ~data;
      else if (sel[1])   z = 4'b0000;

endmodule
```

Solution: This code has mistakes. If there is no `begin` following the `always` statement, then only the first statement will be a sequential statement, the rest starting with the `if` will be interpreted as a separate combinational statement.

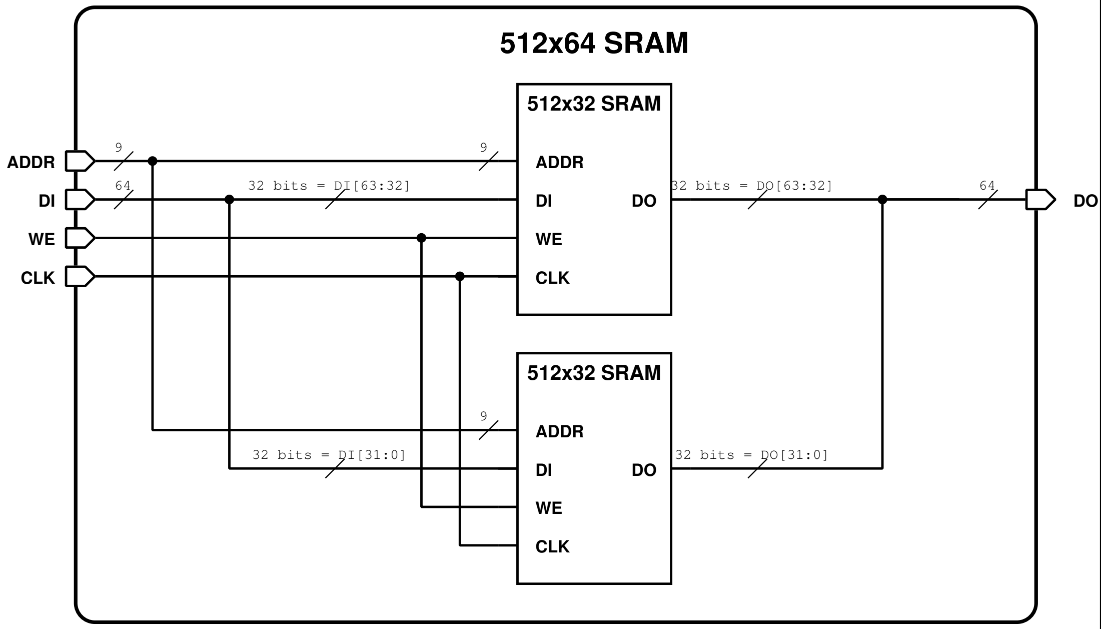
5. In this question, based on the topics we have covered in class, you will be asked to evaluate what would change if we modified the standard single cycle MIPS architecture from 32-bits to 64-bits.

   (a) (7 points) Using one or more of the following single port SRAM memories with 512-entries of 32-bit words



Draw the schematic of a main memory of 4 Kbytes capacity that is suitable for a 64-bit processor that operates on 64-bit words. You can use any combinational logic gates such as AND, OR, NOT gates and multiplexers if necessary.

Solution:



(b) (2 points) If we were to modify the ALU for the 64-bit MIPS processor, what changes would have to be made inside the ALU so that it could process 64-bits at a time? How would this affect the size of the ALU?

Solution: All arithmetic logic operations need to be defined over 64 bits. More or less the ALU would double in size.

(c) (2 points) Would the changes to the ALU that you have outlined above, also impact the propagation delay of the arithmetic and logic functions? Would the 64-bit ALU be faster, slower or exactly the same speed as a 32-bit adder.

Solution: The core of the ALU is an adder. For the 64-bit ALU we will need a 64-bit adder which will need more time to perform the operation (in the worst case twice as much). The propagation delay of the ALU will increase, ALU will be slower.

(d) (2 points) Assuming that the ALU is on the critical path of the processor, how would the clock frequency of the new 64-bit processor compare to the original 32-bit processor?

Solution: The propagation delay of the ALU will increase, which will decrease the maximum clock frequency of the processor.

(e) (4 points) As covered in class, the execution speed of a program on a processor can be given as:

$$Execution\ Time = N \times CPI \times 1/f$$

Where N is the number of instructions, CPI is clocks per instruction and f is the clock frequency.

Taking into account your answers from the previous parts, comment on the execution time of a program running on a single-cycle 64-bit MIPS architecture when compared to the same program running on a single-cycle 32-bit MIPS architecture. Do you expect the execution time to increase, to decrease, or would it stay the same? Briefly explain why. *Assume that only the width of the operands have changed, and the instructions were only modified to cope with the larger data width. No new instructions were added*

Solution:

Since both architectures are single cycle CPI will be 1 for both architectures, so the CPI will not affect the performance. Since the ALU is more complex, the clock frequency will be lower, which will increase the execution time. If the number of instructions can not be decreased by the same proportion the 64-bit processor will be slower.

There are cases when operating on larger numbers could reduce the number of instructions (N). I.e. a number exceeding 4 billion can not be expressed with only 32 bit. If you need to process such a number (for example add two such numbers) a 32-bit architecture will need multiple instructions, whereas a 64-bit architecture could use a single instruction. Reducing the run time.

This page intentionally left blank

6. In this question you will be asked to write a small subroutine using MIPS assembler. You will then write a second program that calls this subroutine more than once. *A copy of Appendix-B of your text book containing all MIPS Instructions has been provided to you.*

   (a) (9 points) Write a subroutine called `findmin` that will return the **minimum** value of an array. The location of the array in memory (`a0`) and the length of the array (`a1`) will be passed as parameters. The minimum value will be returned in the register `v0`.

Solution:

```
findmin:   lw $t4, 0($a0)          # t4 is minimum
           addi $t1, $0, 0         # loop counter t1 init 0

loop:      addi $t1,$t1,1          # t1 ++
           beq $t1, $a1, done      # loop reaches a1 --> done
           sll $t2,$t1,2           # byte addressing, multiply
           add $t2,$t2,$a0         # address of $t1 th member
           lw $t3,0($t2)           # load value from memory
           slt $t5, $t4,$t3        # compare to $t4
           beq $t5,$0,updatemin    # t3 is smaller
           j loop                  # repeat

updatemin: add $t4,$0,$t3          # update $t4
           j loop                  # continue loop

done:      add $v0,$0,$t4          # move result to $t4
           jr $ra                  # jump to $ra
```

(b) (6 points) Now that you have the subroutine `findmin`, write a small MIPS assembly subroutine that:

- finds the minimum of a first array of 64 values starting from the address `0x0000 0400`
- finds the minimum of a second array of 64 values starting from the address `0x0000 0824`
- jumps to label (`first`) if the minimum value of the first array is greater than the minimum value of the second array otherwise execution jumps to label (`second`)
- At the end, jump back to the calling program
- If necessary, save values in stack before calling `findmin`.

Solution:

```
sol:   addi $sp, $sp, -4      # make room on stack
       sw   $ra, 0($sp)       # save ra

       addi $a0, $0, 0x0400   # first address
       addi $a1, $0, 64       # number of elements
       jal  findmin           # v0=findmin(a0,a1)
       add  $s1,$0,$v0        # save result to $s1

       addi $a0, $0, 0x0824   # second address
       addi $a1, $0, 64       # number of elements
       jal  findmin           # v0=findmin(a0,a1)

       slt  $t0, $s1, $v0     # is $s1 less than v0
       beq  $t0, $0, first    # no : jump to first

second:                       # do something
       j    end               # jump over first

first:                        # do something

end:   lw   $ra, 0($sp)       # restore ra
       addi $sp, $sp,4        # restore stack
       jr   $ra               # jump to $ra
```

7. (8 points) What is the difference between a *set associative* and *direct-mapped* cache, **briefly** describe a situation where a *set associative* cache with the *same capacity* performs better than a *direct-mapped* cache version.

Solution:

In a direct mapped cache every memory location can map to only one cache location. In some cases this can cause conflict misses even though there is in principle room in the cache. Set associative caches allow a memory location to be mapped to a set of locations (i.e. a 2-way set associative cache allows mapping to 2 locations) in the cache. This reduces conflict misses.

For example assume a cache with a capacity of 8 words, and consider the code below:

```
one:   lw $s1, 0($s0)  # first read
two:   lw $s2, 4($s0)  # second read
three: lw $s3,32($s0)  # third read
four:  lw $s4,36($s0)  # fourth read
five:  lw $s5, 0($s0)  # re-read first
six:   lw $s6, 4($s0)  # re-read second
```

If you use a direct mapped cache the addresses 0, 32 and 4, 36 will map to the same cache location. So the first two accesses will be compulsory misses, but will fill the cache location 0 and 1. Although we still have room in the cache (only 2 out of 8 is occupied), the next two reads (three and four) will again map to locations 0 and 1 overwriting the old ones. The last two reads will then again be cache misses.

In a 2-way set associative cache, the reads at three and four will not overwrite the old content because there is *another way* to store them in the cache. Therefore, the last two accesses (five and six) will come from the cache.