University of Toronto
Faculty of Applied Science and Engineering

Final Exam
December 2012

ECE253 – Digital and Computer Systems

Examiner – Prof. Stephen Brown

Print:

First Name . . . . Last Name . . . .

Student Number . . . .

1. There are **7** questions and **18** pages. Do **all** questions. The duration of the exam is 2.5 hours.
2. **ALL WORK IS TO BE DONE ON THESE SHEETS.** Use the back of the pages if you need more space. Be sure to indicate clearly if your work continues elsewhere.
3. Closed book. One 2-sided hand-written aid sheet is permitted.
4. No calculators are permitted.

1 [20]	
2 [15]	
3 [10]	
4 [10]	
5 [15]	
6 [15]	
7 [10]	
Total [95]	

1. Short answers:

[3 marks] (a) Convert the following decimal numbers to 16-bit 2's complement.

i. -800

Answer . . . .

ii. -5120

Answer . . . .

iii. -1028

Answer . . . .

[2 marks] (b) Consider the addition of 2 four-bit signed numbers $s_3s_2s_1s_0 = a_3a_2a_1a_0 + b_3b_2b_1b_0$. Give a logic expression for a function z that should be set to 1 only when the sum S is a valid 2's complement result.

Answer . . . .

[2 marks] (c) What are the differences between the Nios II *ldw* and *ldwio* instructions?

Answer . . . .

. . . .

. . . .

. . . .

[3 marks] (d) Assuming that interrupts are enabled for the Nios II processor, list the steps that the processor takes in response to an IRQ signal being asserted by an I/O device. Describe the various processor registers that are affected, and so on.

Answer . . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

[2 marks] (e) Use Boolean algebra to derive a minimal sum-of-products expression for the function f, below. In the boxes on the left specify the number of any identity that you used in that step. Identity numbers are shown at the end of this exam. Use as few steps as possible.

Identity

$f = x_1x_2 + x_3(\overline{x}_1 + \overline{x}_2)$

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

[2 marks] (f) Use Boolean algebra to show that the following equality is valid. In the boxes on the left specify the number of any identity that you used in that step. Identity numbers are shown at the end of this exam. Use as few steps as possible.

Identity

$\overline{k} \cdot x_3x_4 + k \cdot (x_3 + x_4) = x_3x_4 + k \cdot (x_3 + x_4)$

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

[3 marks] (g) Consider the function f shown in the Karnaugh map below. Give both a minimal SOP and POS expression for this function.

x_3x_4 \ x_1x_2	00	01	11	10
00	1	0	0	1
01	0	0	0	0
11	1	1	1	0
10	1	1	0	1

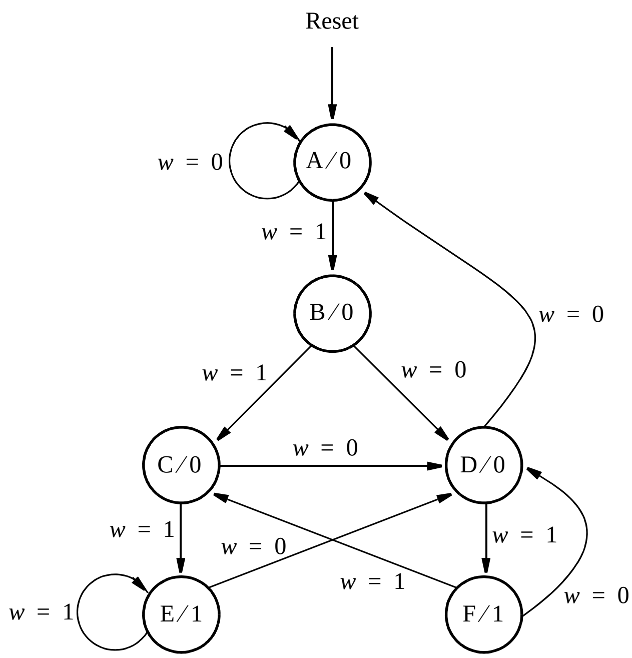
SOP ....

POS ....

[3 marks] (h) Derive and draw a circuit that implements the function $f = x_3 + x_1x_2$ using only 2-to-1 multiplexers. Your circuit should have **only** 2-to-1 multiplexer symbols, and no logic gates. Use the simplest circuit possible.

2. Finite State Machines:

[10 marks] (a) Consider the state diagram shown below.

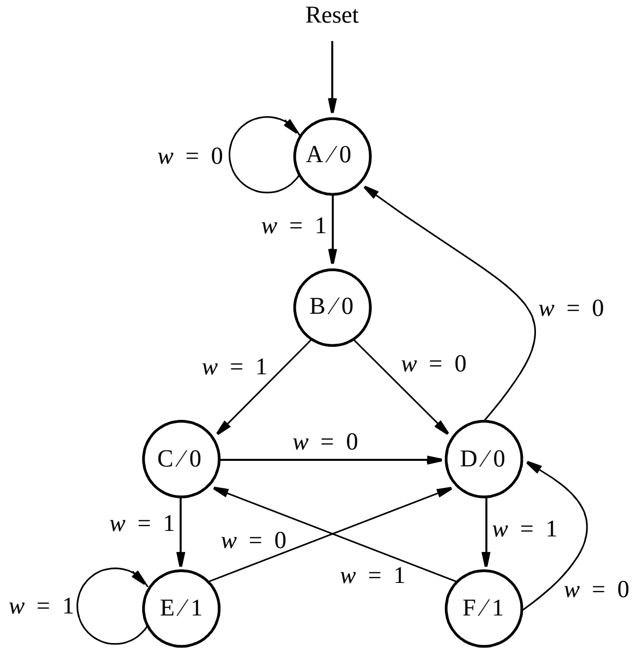


Write complete Verilog code that represents this FSM. Use separate **always** blocks for the state table and the state flip-flops, as done in lectures. Describe the FSM output, which is called z, using either continuous assignment statement(s) or an **always** block (at your discretion). Assign any state codes that you wish to use. Write your Verilog code in the space below and on the following page.

Answer:

Answer for Question 2 FSM Verilog code continued…

[5 marks] (b) The state diagram for this question is shown again below.



Assume that a one-hot code is used with the state assignment

$y_5y_4y_3y_2y_1y_0 = 000001(A), 000010(B), 000100(C), 001000(D), 010000(E), 100000(F)$

i. Write a logic expression for the signal Y_1, which is the input of state flip-flop y_1.

Answer . . . .

ii. Write a logic expression for the signal Y_3, which is the input of state flip-flop y_3.

Answer . . . .

3. Assembly Language Trace:

[10 marks] Consider the assembly language program shown below, which was also explained in lectures. This code calls the recursive subroutine named FINDSUM to sum the numbers from 0 to N. In this code N is set to the value 2. Note that the address that each instruction would have in memory, assuming that the program starts at address 00000000, is shown to the left of the code.

```
                  .text
                  .global  _start
              _start:
00000000          movia    sp, 0x20000
00000008          mov      r2, r0
0000000C          ldw      r4, N(r0)          # data is passed to subroutine in r4
00000010          call     FINDSUM            # result will be in r2
              END:
00000014          br       END                # wait here

              FINDSUM:
00000018          bne      r4, r0, RECURSE
0000001C          add      r2, r0, r0
00000020          ret
              RECURSE:
00000024          subi     sp, sp, 8
00000028          stw      r4, 0(sp)
0000002C          stw      ra, 4(sp)
00000030          subi     r4, r4, 1
00000034          call     FINDSUM

00000038          ldw      r4, 0(sp)
0000003C          add      r2, r4, r2
00000040          ldw      ra, 4(sp)
00000044          addi     sp, sp, 8
00000048          ret

                  .data
              N:
                  .word    2
                  .end
```

If this program is executed on the Nios II processor, what would be the values of the Nios II registers shown below when the program reaches, but has not yet executed, the instruction **add** r2, r0, r0. Also show in the space on the next page the contents of the stack in memory at this point in time.

pc		ra		sp	

Answer space for the contents of the stack in memory. For memory values that are not known, if any, write N/A in the corresponding box.

Memory Address	Content
....	
....	
....	
....	
....	
....	
20000	

4. Assembly Language Debug:

[10 marks] The Nios II program shown below performs a bubble sort—you implemented such an algorithm in Lab 7. Recall that the data, as shown on the next page, is stored in memory at the location LIST. The first item at LIST is the number of words to be sorted. The list is sorted by the program "in place" in the memory, in descending order. Unfortunately, this code contains some errors. Place an X to the left of each instruction that is erroneous, and indicate to the right of the instruction how you would fix this error. Assume that the program is loaded into memory starting at address 0. The code for the SWAP subroutine is on the following page.

Suggested correction

```
        .text
        .global   _start
_start:
[ ]     movia     sp, 0                     ...........................
[ ]     movia     r9, LIST                  ...........................
BEGIN_SORT:
[ ]     ldw       r20, 0(r9)                ...........................
RESTART_SORT:
[ ]     add       r18, r0, r0               ...........................
[ ]     addi      r19, r0, 1                ...........................
[ ]     addi      r4, r9, 4                 ...........................
SORT_LOOP:
[ ]     call      SWAP                      ...........................
[ ]     mov       r18, r2                   ...........................
[ ]     addi      r19, r19, 1               ...........................
[ ]     addi      r4, r4, 1                 ...........................
[ ]     beq       r19, r20, SORT_LOOP       ...........................
[ ]     addi      r20, r20, -1              ...........................
[ ]     bne       r18, r0, RESTART_SORT     ...........................
END:
        br        END
```

SWAP list elements
Register r4 has the address of the first element to swap, and
register r4 + 4 is the address of the other element
Return 1 in register r2 if swap performed, otherwise return 0

		Suggested correction
SWAP:		
addi	sp, sp, -8	. . . .
stw	r5, 0(sp)	. . . .
stw	r6, 4(sp)	. . . .
add	r2, r0, r0	. . . .
ldw	r5, 0(r4)	. . . .
ldw	r6, 4(r4)	. . . .
bgt	r5, r6, SKIP_SWAP	. . . .
stw	r6, 0(r4)	. . . .
stw	r5, 4(r4)	. . . .
addi	r2, r0, 1	. . . .
SKIP_SWAP:		
addi	sp, sp, 8	. . . .
ldw	r5, 0(sp)	. . . .
ldw	r6, 4(sp)	. . . .
eret		. . . .

```
    .data
LIST:
    .word   10, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    .end
```

5. Assembly Language Subroutine:

[15 marks] The Nios II processor does not include any hardware for performing a *division* operation, but the Nios II assembly language defines a divide instruction, as:

div rC, rA, rB

The result is defined as the integer operation $rC = rA/rB$. Since the **div** instuction isn't implemented in the Nios II processor, you are to write a subroutine, called DIVIDE, that performs a division operation. The inputs to DIVIDE should be passed to the subroutine in registers $r4$ and $r3$, and the result should be returned in register $r2$. The result should have the integer quotient of $r4/r3$ in the least-significant halfword of $r2$, and the remainder should be in the most-significant halfword of $r2$. Assume that $r4$ and $r3$ contain unsigned values, and that $r3 \neq 0$.

You need to provide a main program that calls your DIVIDE subroutine. Part of this main program is shown below; fill in the rest of the code. The values of arguments A and B that are used to produce the result $C = A/B$ are stored in memory as shown in the code—your main program needs to load these values from memory and pass them to the subroutine. Write the code for DIVIDE in the space on the next page. Include **meaningful comments** in your code!

```
        .text
        .global   _start
_start:









END:
        br        END                     # wait here

        .data
A:
        .word     10
B:
        .word     3
        .end
```

Answer space for the DIVIDE subroutine.

```
        .global DIVIDE
DIVIDE:
```

6. Exception Handler:

[15 marks] Consider the Nios II main program shown below. It enables interrupts for the pushbutton KEYs (but the interrupt service routine for the pushbutton KEYs isn't used in this question), and then initializes some registers and calls the **div** instuction. As we said in the previous question, the **div** instruction is not implemented in the Nios II processor. This means that when the processor tries to execute this instruction an exception occurs, and the processor automatically branches to the exception handler address (0x20).

```
        .text
        .global   _start
_start:
        movia     sp, 0x20000           # stack starts from a large memory address

# assume that we need pushbutton KEY interrupts to be enabled
        movia     r15, 0x10000050       # KEY base address
        movi      r7, 0b01010           # set interrupt mask bit for KEY3 and KEY1
        stwio     r7, 8(r15)            # interrupt mask register is (base + 8)

# enable Nios II processor interrupts
        movi      r7, 0b010             # set interrupt mask bit for irq1 (KEYs port)
        wrctl     ienable, r7           # write to control register
        movi      r7, 1
        wrctl     status, r7            # turn on Nios II interrupt processing

DO_DIV:
        movi      r4, 10
        movi      r3, 3
        div       r2, r4, r3

# there would be additional code here in a real program . . .
IDLE:
        br        IDLE
        .end
```

You are to write the exception handler for this main program, in the space on the next page. The exception handler has to check whether it has been called due to an IRQ, or due to an exception. If an exception, then your code has to check if the exception has been caused by the **div** instruction. If this is the cause of the exception, then simply call the DIVIDE subroutine that you wrote in Question 5. This means that your solution for this question works only for the exact instruction **div** r2, r4, r3, because these are the registers used in your DIVIDE subroutine. Make sure that you return properly from the exception handler to the main program, with the division result in the right register. Include **meaningful comments** in your code!

To check if the exception is caused by a **div** instruction, you need to know the format of the OPCODE for **div**, which would be produced by the Assembler for this program. The OPCODE format of the Nios II **div** instruction is shown at the end of this exam.

Answer space for the exception handler. Fill in the missing parts of the code.

```
            .section    .exceptions, "ax"
            .global     EXCEPTION_HANDLER
EXCEPTION_HANDLER:
            rdctl       et, ipending
            beq         et, r0, SKIP_DEC          # it is an exception, not an IRQ
            subi        ea, ea, 4                 # decrement ea by one instruction for IRQ
SKIP_DEC:
            subi        sp, sp, . . . . . . . . . # save registers, then check for div instruction











CHECK_irq1:                                       # KEYs are interrupt level 1
            andi        r22, et, 0b10
            beq         r22, r0, END_ISR          # other irq levels not handled in this code
            call        KEY_ISR                   # code for this ISR is not shown here
END_ISR:
                                                  # restore registers








            .end
```

7. Exception Handler (Advanced):

[10 marks] In Question 6 you called your DIVIDE subroutine from Question 5 to implement the division operation. But your DIVIDE subroutine works only for the exact instruction **div** r2, r4, r3. If the operand registers are not exactly $r2, r4, r3$, then your DIVIDE subroutine can't be used to implement the **div** instruction. For this questsion you are to write a somewhat more general DIVIDE subroutine that will work for any register operands from $r2$ to $r6$. For example, it has to work with **div** r2, r3, r4, or **div** r6, r2, r3, or **div** r5, r4, r2, and so on. This means that your new DIVIDE subroutine has to examine the OPCODE that caused the exception to determine what register operands are involved. Assume that only registers in the range of $r2$ to $r6$ will be used, and make the same assumptions as for Question 5 regarding the division operation ($rC = rA/rB$, unsigned integers, argument *rB*$\neq$ 0, quotient and remainder in *rC*).

Complete the code below (additional space is provided on the next page). Think about the comments that are shown in the code—they provide a *hint* about how your DIVIDE subroutine can access the contents of registers *rA*, *rB*, and *rC*. Include **meaningful comments** in your code!

```
        .global   DIVIDE
DIVIDE:
        subi      sp, sp, 28      # reserve space on the stack
        stw       r2, 8(sp)       # r2 saved at sp + 2 words
        stw       r3, 12(sp)      # r3 saved at sp + 3 words
        stw       r4, 16(sp)      # r4 saved at sp + 4 words
        stw       r5, 20(sp)      # r5 saved at sp + 5 words
        stw       r6, 24(sp)      # r6 saved at sp + 6 words
```

Answer for Question 7 DIVIDE code continued…

Boolean Identities

12a.	$x \cdot (y + z) = x \cdot y + x \cdot z$	*Distributive*
13a.	$x + x \cdot y = x$	*Absorption*
14a.	$x \cdot y + x \cdot \overline{y} = x$	*Combining*
15a.	$\overline{x \cdot y} = \overline{x} + \overline{y}$	*DeMorgan's theorem*
16a.	$x + \overline{x} \cdot y = x + y$	
17a.	$x \cdot y + y \cdot z + \overline{x} \cdot z = x \cdot y + \overline{x} \cdot z$	*Consensus*

Nios II div Instruction OPCODE Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A					B					C					0x25						0					0x3a					