NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

EXAMINATION FOR
Semester 1 AY2009/2010

CS2271 – Embedded Systems

Nov/Dec 2009 **Time Allowed: 2 Hours**

INSTRUCTIONS TO CANDIDATES

1. This examination paper contains **THREE** questions and comprises **EIGHTEEN (18)** printed pages, including this page. There is also an appendix printed on **TWENTY-TWO (22)** printed pages. For readability reasons, there are some blank pages in between the 22 printed pages. The printed pages are numbered while the blank pages are not.

2. The weightage for each question is as indicated, and total 100 marks. Note that the marks distribution is not equal across questions.

3. Answer **ALL** questions within the boxes provided. Anything written outside of the boxes will not be graded.

4. This is an open book examination. All printed or hand-written materials are allowed. However programmable calculators and computing devices including smart phones and PDAs are not allowed.

5. Please write your Matriculation Number below and at the top-left corner of each page in the spaces provided. Do not write your name.

MATRICULATION NO: ______________________________

This portion is for examiner's use only

Question	Marks	Remarks
Q1	/35	
Q2	/50	
Q3	/15	
Total	/100	

Question 1 Fundamentals (35 MARKS)

Answer all parts of this question.

a. For each of the following statements, indicate whether you agree with the statement by circling "T", or disagree by circling "F". (4 MARKS TOTAL)

   i. The µC/OS-II operating system uses RMS priority scheduling. T / F

   ii. FPGAs have cost advantage over ASICs when production of a design is limited to a small to moderate quantities, but not in large quantities. T / F

   iii. Both the Liu and Leyland criteria and critical instance analysis are not applicable to multi-processor or multi-core machines. T / F

   iv. Multiple parallel reads from a non-RAM variable in Handel-C is likely to result in incorrect results when rendered to hardware. T / F

b. Explain why the `ram` structure in Handel-C is not suited for general-purpose I/O on an FPGA. (5 MARKS)

c. Study the process schedule below and answer the questions that follow. Some tasks may be aperiodic, and the periodic tasks are scheduled using a real-time schedule algorithm. Tasks A, B and C have CPU times of 1, 3 and 1 respectively. (20 MARKS TOTAL).

Time	Running Task	Time	Running Task	Time	Running Task
0	A	12	A	24	A
1	B	13	B	25	C
2	B	14	B	26	B
3	A	15	A	27	A
4	B	16	B	28	B
5	Idle	17	C	29	B
6	A	18	A	30	A
7	B	19	B	31	B
8	B	20	B	32	B
9	A	21	A	33	A
10	B	22	B	34	B
11	Idle	23	Idle	35	C

i. Is the underlying operating system pre-emptive or co-operative? Explain your answer. (3 MARKS)

ii. Are the tasks periodic or aperiodic? Estimate the period of each task. For aperiodic tasks, take the worst-case assumption (i.e. shortest interval between runs) in estimating their periods using the information given. (6 MARKS)

Task	Periodic/Aperiodic	C_i	P_i
A		1	
B		3	
C		1	

iii. List the tasks in order of priority. Explain how you derived your answer (5 MARKS)

iv. Given your answers in part ii, are the tasks schedulable under RMS? EDF? Explain your answer (6 MARKS)

d. Correct the errors in the following µC/OS-II application by amending directly on the code provided. For your convenience, relevant portions of the uC/OS-II API are shown in the appendix. Assume that the program logic is correct overall, and that errors are confined to the way the RTOS features are used. The RTOS is configured to grow the task stacks downwards. (6 MARKS):

```
#include <app_cfg.h>
#include <ucos_ii.h>

OS_EVENT *mySem, *myQ;
OS_FLAG_GRP *myFlags;
void *QArray[10];
int myVar=0;
OS_STK taskA_stack[1024], taskB_stack[1024], taskC_stack[1024],
                taskD_stack[1024];

void taskA(void *ptr)
{
      INT8U err;
      /* Suppress compiler warnings */
      ptr=ptr;

      /* Sets some flags, then sleeps for 5 seconds, repeats */
      for(;;)
      {
            OSFlagPost(myFlags, 0x53, OS_FLAG_SET, &err);
            OSTimeDlyHMSM(0,0,5,0);
      }
}

void taskB(void *ptr)
{
      INT8U err;
      int val;

      /* Waits on the flags set by TASK A, then sends a message to
      Task C via a queue  The message is a fixed value passed in via
      ptr and the flags are cleared */

      val=(int) ptr;

      OSFlagPend(myFlags, 0x41, OS_FLAG_WAIT_SET_ALL, 0, &err);
      OSQPost(myQ, (void *) val);
      OSFlagPost(myFlags, 0x41, OS_FLAG_CLR, &err);
}

void taskC(void *ptr)
{
      /* Receives a message from task B, then takes a semaphore and
      updates a shared variable */

      int *msg;
      INT8U err;

      /* Suppress compiler warnings */
      ptr=ptr;
```

```
        mySem=OSSemCreate(1);
        for(;;)
        {
                msg=(int *) OSQPend(myQ, 0, &err);
                OSSemPend(mySem, 0, &err);
                myVar += *msg;
                OSSemPost(mySem);
        }
}

void taskD(void *ptr)
{
        /* Prints out the value of myVar every 3 seconds */
        INT8U err;
        /* Suppress compiler warnings */
        ptr=ptr;
        unsigned time=0;

        for(;;)
        {
                OSTimeDlyHMSM(0,0,3,0);
                OSSemPend(mySem, 0, &err);
                OS_Printf("Time: %-3u Var: %-3d\n", time, myVar);
                OSSemPost(mySem);
                time+=3;
        }
}

int main()
{
        INT8U err;
        myQ=OSQCreate(QArray, 10);
        OSTaskCreate(taskA, NULL, &taskA_stack[0], 7);
        OSTaskCreate(taskB, (void *) 5, &taskB_stack[0], 6);
        OSTaskCreate(taskC, NULL, &taskC_stack[0], 5);
        OSTaskCreate(taskD, NULL, &taskD_stack[0], 4);
        OSStart();
}
```

Question 2 Handel-C Design (50 MARKS)

When sensitive or private data needs to be transmitted electronically, people often use various encryption techniques to ensure that the data cannot be (easily) read other than by the intended recipients of the message. A key is used to encrypt the message, and the same key (which must be kept secret) is used to decrypt the message.

In this question, we will explore the situation where we have both the original message ("plaintext") and the encrypted version ("ciphertext"), and we want to recover the key that produced the ciphertext from the plaintext. We will use a simplified version of the key generation and encryption algorithms used in the "Wired Equivalent Privacy" or WEP algorithm. This algorithm is obsolete but is still often used to secure wireless 802.11 networks, particularly in private homes.

Key Generation Algorithm

In our encryption system, we will use a 64-bit (8-byte) key that is generated from a passphrase using the algorithm shown below.

Assuming a passphrase "Never the twain shall meet!", the key generation algorithm first pads in spaces to ensure that the total number of characters is a multiple of 8. The algorithm then arranges the passphrase as shown below, and takes an XOR (⊕) of all the ASCII values of the letters in a column to derive the key byte for that column:

N	e	v	e	r	space	t	h
⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
e	space	t	w	a	i	n	space
⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
s	h	a	l	l	space	m	e
⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
e	t	!	space	space	space	space	space

Taking the XOR of all key characters in a column yields the following 64-bit key:

3D	59	42	5E	5F	49	57	0D

Encryption/Decryption Algorithm

Once the key is generated from the pass phrase, we can now apply a simple "block cipher" to encrypt the message. To do this, we split the plaintext into chunks of 8 bytes, then perform an XOR of the ASCII code of the plaintext character with the corresponding key. So to encrypt the phrase "I love CS2271!" We start with the first 8 characters and XOR with the key to generate the first 8 bytes of the cipher text.

I	space	l	o	v	e	space	C
⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
3D	59	42	5E	5F	49	57	0D

Taking the XOR of the key and the ASCII code of the character in each column yields an 8-character cipher block. We repeat with subsequent 8-character blocks to generate the remaining cipher blocks.

S	2	2	7	1	!	space	space
⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
3D	59	42	5E	5F	49	57	0D

Note that we pad the plaintext with spaces to ensure that the total number of characters XORed is a multiple of 8.

Since XOR is a reversible process, decryption involves just taking the XOR of the ciphertext with the same key bytes. The figure below shows the reversibility of XOR:

1	0	1	1	0	1	0	1
⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
1	1	0	1	1	0	1	1

This yields the encrypted string:

0	1	1	0	1	1	1	0

Taking the output pattern 01101110 and XOR with the same key 11011011:

0	1	1	0	1	1	1	0
⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
1	1	0	1	1	0	1	1

This produces:

1	0	1	1	0	1	0	1

Which is exactly our original binary string.

a. Ideally the keys produced by the key generator would cover the entire range from 00 00 00 00 00 00 00 00 to FF FF FF FF FF FF FF FF. Show, however, that if the key phrase consists entirely of ASCII characters, the actual key range only covers from 00 00 00 00 00 00 00 00 to 7F 7F 7F 7F 7F 7F 7F 7F. (Hint: the last ASCII character has a code of 127) (3 MARKS)

b. The following C code is an implementation of the XOR encryption algorithm above. Translate this into Handel-C. Assume that KEYLEN has been declared elsewhere, and call your Handel-C function "cipher". Exploit as much parallelism as possible.
(10 MARKS)

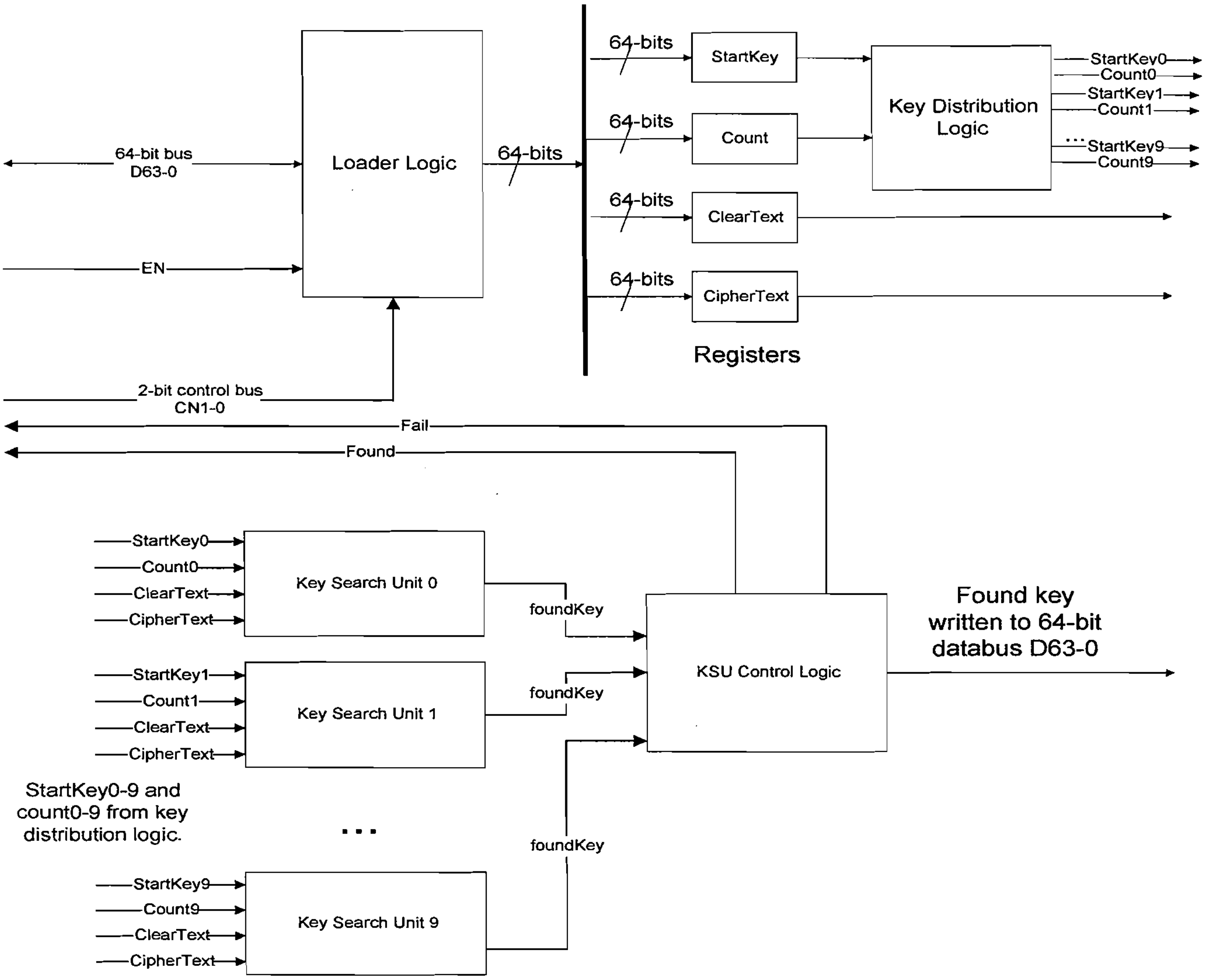
```
/* Encrypts or decrypts the string in t1, storing the result
in t2. Cipher is symmetric: placing cleartext in t1 results
in encrypted text in t2, and vice versa. Decryption does not remove
additional spaces at the end of the cleartext.*/
void cipher(char *t1, char *t2, char key[KEYLEN])
{
      int bal, i, j;

      /* Ensure that t1 is a multiple of KEYLEN. Pad with spaces
      if not. */

      if(strlen(t1) % KEYLEN)
      {
            bal=KEYLEN - strlen(t1) % KEYLEN;

            for(i=0; i<bal; i++)
                  strcat(t1, " ");
      }

      /* Now we encrypt/decrypt the data */
      for(i=0; i<strlen(t1) / KEYLEN; i++)
            for(j=0; j<KEYLEN; j++)
                  t2[i*KEYLEN+j]=t1[i*KEYLEN+j]^key[j];
      t2[strlen(t1)]='\0';
}
```

The above program converted to Handel-C is:

(Continue on next page)

Page is intentionally left blank.

We will now design a key-cracker that, given a cleartext sample and its ciphertext equivalent, will search part of the keyspace for a possible key match. The block diagram of the cracker is shown below. Note that this is a logical block diagram and blocks may not correspond to actual Handel-C functions or circuits.



Briefly, this is how the key-cracker is used by an external circuit whose design is beyond the scope of this question.

i. The EN line is asserted and stays asserted throughout.

ii. CN1-0 is set to 0b00 (the "0b" prefix indicates a binary number) and the starting 64-bit key (StartKey) is put onto the data bus D63-0. This is copied by the key cracker into the 64-bit StartKey register.

iii. CN1-0 is then set to 0b01, and a 64-bit count is put onto D63-0. This is copied into the 64-bit Count Register.

iv. CN1-0 is set to 0b10, and the 64-bit clear-text is put onto D63-0. This is copied into the 64-bit ClearText Register. Assume that the clear-text has been space-padded if the actual text has fewer than 8 bytes.

v. CN1-0 is set to 0b11, and the 64-bit cipher-text is put onto D63-0. This is copied into the 64-bit CipherText Register.

vi. Once the ciphertext is loaded, the key cracker immediately distributes the keys (ranging from StartKey to StartKey+Count-1) to the 10 Key Search Units (KSUs) so that they each search a mutually-exclusive sub-range. For example, if the key cracker is to search 100 keys starting from key 0, the key distribution logic distributes the keys as follows:

Key Search Unit	StartKey	Count
0	0	10
1	10	10
2	20	10
…	…	…
9	90	10

So KSU 0 seaches 10 keys from key 0 to 9, KSU 1 searches 10 keys from key 10 to 19, etc.

vii. Each KSU that completes its search either returns the found key to the KSU Control Logic, or a "-1" value indicating that the KSU had failed to find a key.

viii. When all 10 KSUs have returned a result (either the found key or -1), the KSU control logic either asserts the FOUND line and places the found key onto D63-0 (the same data bus used to load the registers), or asserts the FAIL line.

ix. The external circuit de-asserts EN, after which the key-cracker de-asserts the FOUND/FAIL lines and takes the key, if any, off the data bus.

The StartKey, Count and ClearText registers can be loaded in any order, but the CipherText register must be loaded last as loading this register triggers the search. Although the FPGA itself is clocked, the operation of the key-cracker is asynchronous and depends only on the assertion of control lines. Therefore there are no timing diagrams for this circuit.

c. Modify your function in Question 2b so that up to 10 units can concurrently access your cipher routine. You do not need to rewrite the entire function here, only the parts that you changed. (3 MARKS)

d. Based on the block diagram and description above, briefly describe how your cipher routine in Question 2b can be simplified and/or optimized for use in the key-cracker. (4 MARKS)

e. Write a function called "keysearch" that takes the starting key, the number of keys to search, the clear and cipher text, and any other arguments you deem necessary. This function then searches the subrange for a possible key that maps the cleartext to the ciphertext. Ensure that your function can be called up to 10 times concurrently. If the function finds a key, it returns the key via its corresponding entry in the channel array `keys`. If it does not find any matching key, it returns a `-1`. The `keys` channel array is declared as shown below. (10 MARKS)

```
chan <int 65> keys[10]; /* Note: 65 bits, not 64 */
```

(Continue next page)

f. Write the remaining functions and data/register declarations you need to implement the key cracker. Assume that the buses have been declared using the following `interface` statement, and you do not need to declare them yourself. To save time, you can use ... for code that is repetitious, for example “P63”, “P62”, ..., “P0”.(20 MARKS)

```
signal unsigned 64 key_out=0; // For returning found keys
signal unsigned 1 rw=0; // To control the tristate bus.
signal unsigned 2 status=0; // 10=FAIL, 01=FOUND.
                              //00 and 11 are illegal values.

interface bus_ts(usigned 64 key_in) DataBus(unsigned 64
      bus_out=key_out, unsigned 1 cond=rw) with {data="P63", "P62",
            … "P0"};

interface bus_in(unsigned 1 en_in) EnBus() with {data="P64"};

interface bus_in(unsigned 2 ctl_in) CtlBus() with {data="P66", "P65"};

interface bus_out() StatusBus(unsigned 2 statout=status) with
      {data="P68", "P67"};
```

Continue on next page.

Continue on next page.

Page is intentionally left blank.

Question 3 Real Time System Design (15 MARKS)

a. In the ECPU as given in the lecture notes, the compare (CMP) and branch (BEQ, BNE, BGT) are separate. Illustrate with an example why the ECPU's design makes it difficult, if not impossible, to implement fully re-entrant routines. (4 MARKS)

b. Would the use of semaphores solve the problems highlighted in Question 3a? Why or why not? (3 MARKS)

c. Device drivers that drive complex devices like network cards or Universal Serial Bus (USB) devices often have to use relatively complex algorithms involving state machines and packet exchanges to maintain control. Explain why such devices are not well suited to a hard real-time environment. (4 MARKS)

d. Sometimes usage of such devices in a hard real-time system is unavoidable. Show what features of a real-time operating system can be used to mitigate the risks associated with using these devices, and explain your answer. You may assume that these devices themselves do not have strict deadlines. (4 MARKS)

End of Paper