NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

EXAMINATION FOR
Semester 2 AY2005/2006

CS2271 – Embedded Systems

Apr/May 2006 **Time Allowed: 2 Hours**

INSTRUCTIONS TO CANDIDATES

1. This examination paper contains **FOUR** questions and comprises **TWELVE (12)** printed pages, including this page. There is also a **THREE (3)** page appendix attached to the end of this paper. Each question has different weightage, and total 50 marks for the paper.
2. Answer **ALL** questions within the spaces provided in this booklet.
3. This is an Open Book examination.
4. Please write your Matriculation Number below.

MATRICULATION NO: _____

This portion is for examiner's use only

Question	Marks	Remarks
Q1 (5 marks)		
Q2 (18 marks)		
Q3 (17 marks)		
Q4 (10 marks)		
Total		

Question 1 Programmable Logic Devices **(5 MARKS)**

Consider this statement:

"Any circuit that can be implemented with an FPGA can also be implemented using Programmable Logic Arrays (PLAs) and some other stuff".

a. Do you agree with this statement? Why or why not? **(3 marks)**

b. Assuming that the statement is true, and assuming that you have a circuit implemented on an FPGA. What are the "other stuff" that you would require to implement an equivalent circuit using PLAs? **(2 marks)**.

Question 2 Handel-C Analysis **(18 Marks)**

The attached Appendix A **(DO NOT DETACH!)** shows the Handel-C source code for a pipelined ECPU. There are several complex issues that have not been handled by this implementation, and in this question we will consider two of them.

a. One critical issue is that several instructions following a branch instruction always ends up being executed regardless of the outcome of the branch. Based on the source code given, estimate how many such instructions will be executed. **(3 marks)**

b. Does the same problem exist with the JMP instruction? Why or why not? **(2 marks)**

An interrupt is "precise" if, when it is triggered and interrupts are not disabled, no program instruction after the interrupted instruction is executed until the interrupt service routine has completed. So given the following fragment:

Instr #		**Instruction**
I1		cmpi r1, r2
I2		blt skip
I3		addi r1, r1, 1
I4	skip:	muli r2, r2, 23
I5		...

If instruction I3 is interrupted and interrupts are not disabled, then instructions I4, I5, etc will not be executed until the interrupt service routine has completed.

c. The pipelined version of the ECPU does not support precise interrupts. Explain why. **(5 marks)**

d. Modify the code in Appendix A so that the ECPU now supports precise interrupts. Show your modifications in Appendix A (by cancelling, rewriting etc), and not here. **(8 marks)**.

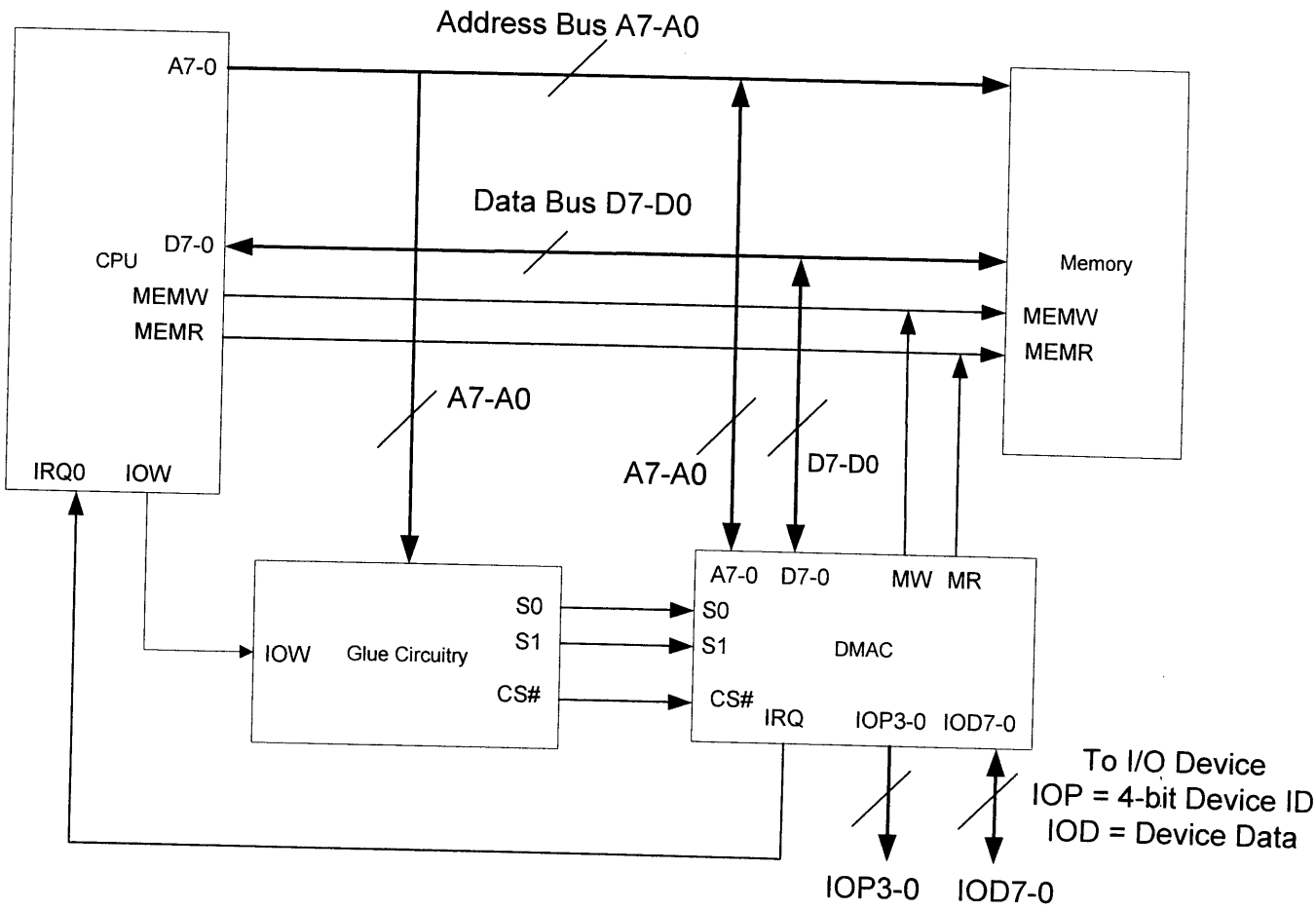
Question 3 HDL Design **(17 Marks)**

A "direct memory access controller" or DMAC is a device that, when set up correctly by the microprocessor, automatically reads from I/O devices and transfers data to memory without CPU intervention. When the transfer is completed, the DMAC interrupts the CPU. A DMAC is also able, when set up correctly by the CPU, to read from memory and write to the I/O device.

The CPU will write to the following I/O addresses to set up the DMAC (note: Numbers prefixed with 0x are in hexadecimal, while numbers prefixed with 0b are binary. Numbers with no prefixes are decimal).

I/O Port Address	Function
0xFC	CPU writes to this port to tell the DMAC the starting address for the transfer.
0xFD	CPU writes to this port to tell the DMAC how many bytes to transfer.
0xFE	CPU writes a "0x0*N*" to this port to tell the DMAC to begin reading from the I/O device *N*, and a "0xF*N*" to tell the DMAC to begin writing to I/O device *N*, where *N* is a 4-bit device identifier..
0xFF	CPU writes to this port to acknowledge the interrupt from the DMAC. The DMAC de-asserts the interrupts once the CPU has written to this port.

This block diagram shows how the DMAC is connected to the CPU.



The OE# pin on the DMAC is asserted low to activate the DMAC. Otherwise the DMAC is inactive. The SETUP pins (S1 and S0) correspond to the following DMAC setup modes:

Mode (S1S0)	**Action**
0b00	Read the CPU data bus for the starting address
0b01	Read the CPU data bus for the number of bytes to transfer
0b10	Read the CPU data bus for the "read I/O" or "write I/O" start commands
0b11	IRQ acknowledged. De-assert the IRQ line.

a. Using ABEL, write a HDL description of the "glue circuitry" to be implemented on the 82S100 PLA. Pins 2-9 and 20-27 are input pins, and pins 10-18 are output pins. **(3 marks)**

b. The DMAC is to be implemented using Handel-C on an FPGA. Write all the necessary interface declarations here (you may use any pin names like "P2", "P3", etc. You do not need to assume any particular FPGA family or device). **(6 marks)**.

c. Implement the DMAC using Handel-C. Assume that the internal clock is ½ the speed of the external clock, which is connected to pin P1, and that the DMAC does not need to arbitrate for the CPU/Memory bus.

Further, to write to memory, MEMW and A7-0 must be held for at least 2 external clock cycles prior to placing data onto the D7-0 data bus, during which time MEMW and A7-0 must still be held constant. Similarly, to read from memory, MEMR and A7-0 must be held for at least 2 external clock cycles prior to reading from D7-0. MEMR and A7-0 must be held steady during the read. **(8 marks)**

(Page intentionally left blank for your solution)

(You may continue writing on the back of this page. Please indicate "PTO" if you are doing so)

Question 4 – Real Time Operating Systems (10 Marks)

a. The following code contains a potential bug even if each task code has no bugs at all. What is this bug? What situation will produce this bug? **(3 marks)**.

```
semaphore sem1, sem2;

taskA()
{
            ... Some other code here
      take_sema(sem1);
      !! Do long computation
      release_sema(sem1);
}

taskB()
{
      ... Some other code here
      take_sema(sem2);
      !! Do long computation
      release_sema(sem2);
}

taskC()
{
      ... Some other code here
      take_sema(sem1);
      !! Do long computation
      release_sema(sem1);
}
```

The table below shows the interface to a re-entrant queuing library:

Pre-Condition:	Subroutine Entry Point	Post Condition:
R31 = Base address of queue R30 = Task number to be enqueued	0xF000	Task number in R30 is placed into the queue indicated by R31.
R31 = Base address of queue	0xF100	The task number at the head of the queue is placed into R30 is the queue is not empty. Otherwise a "-1" is put into R30.

b. Write an ECPU routine that does the following:

i. Saves the context of the current running task.
ii. Moves the current task to the READY queue.
iii. Selects a task from the head of the READY queue.
iv. Restore its context.

There are some things you need to note:

i. Position your routine to start at memory location 100.
ii. You need to maintain the task number of the currently running task.
iii. You can expect to have >2 tasks that have to be run, including the currently running task.
iv. You may use two ECPU instructions called SAVE and RESTORE to save and restore a task context:

```
save src_reg ; Save registers R0-R31, PC and stack pointer to memory
             ; pointed to by register src_reg

restore src_reg ; Restore registers R0-R31, PC and stack pointer from
                ; memory location pointed to by register src_reg.
```

So if R0 contains 100, then `save R0` will write R0-R31, PC and SP to locations 100-133. Since you have >2 tasks including the currently running task, you must be careful not to over-write other task's context when saving. Assume that registers R29-R31 are reserved for your use and that tasks never use these registers. **(6 marks)**

(You may continue your code on the back of this page. Please indicate "PTO" if you are doing so).

c. Suggest how your routine in part b can be used as part of a pre-emptive scheduler. **(1 mark).**

End of Paper

```
chan  <unsigned INSTR_SIZE> ir_reg;

/* These replace the original variables defined for
storing the decoded portions of ir_reg */

chan <int WORD_SIZE> const_reg, op1_reg, op2_reg;
chan <unsigned 4> opcode_reg, src1_reg, dest_reg,
        wb_destreg;
chan <int WORD_SIZE> opout_reg;

/* The fetch stage */
void inst_fetch()
{
   par
   {
            /* Fetch the instruction from the memory
            location pointed  by PC */
         ir = program[pc];
         pc++;
   }

   ir_reg ! ir; // starts the ID stage

}

/* The decode stage */

void inst_decode()
{
   unsigned INSTR_SIZE my_ir;

   ir_reg ? my_ir;

   // Reads in instruction from IF stage
   // Decode it, passing parts of the
   // instruction to the
   // EX stage.
   par
   {
         constant_reg ! my_ir<-8;
         op1_reg ! read_reg((my_ir\\4)<-4);
         dest_reg ! (my_ir\\8)<-4;
         opcode_reg ! my_ir\\12;
   }

         op2_reg ! read_reg(my_ir<-4);
}

void inst_execute()
{
   int WORD_SIZE my_const, my_op1, my_op2,
         opout;

   unsigned 4 my_opcode, my_destreg;

   /* Read stuff from previous stage */
   par{
```

```
            const_reg ? my_const;
            op1_reg ? my_op1;
            dest_reg ? my_destreg;
            opcode_reg ? my_opcode;
            op2_reg ? my_op2;
        }

        switch(my_opcode)
        {

        ..... /* Other instructions */

            case BEQ: if(eqFlag)
                        pc = my_const;
                        break;
            case BLT: if(ltFlag)
                        pc = my_const;
                        break;
            case BGT: if(gtFlag)
                        pc = my_const;
                        break;
            case JMP: pc = my_const;
                      break;
        ..... /* Other instructions */
        }

        /* Send results to WB stage */
        par{
            wb_destreg ! my_detreg;
            opout_reg ! opout;
            opcode_reg ! my_opcode;
        }
    }
```

```
/* The writeback stage */
void inst_writeback()
{
   unsigned 8 IRQValues;
   unsigned 3 i;
   unsigned 1 quit = 0;

/* We will always writeback except for SW, OUT and HLT.
        These opcodes are all > LW */

usigned WORD_SIZE opout;
usigned 4 destreg, opcode;

par
{
   wb_destreg ? destreg;
   opout_reg ? opout;
   opcode_reg ? opcode;
}

if(opcode != CMP && opcode != BEQ && opcode != BLT  &&
      opcode != BGT && opcode != JMP && opcode != SW
               && opcode != PRN && opcode != HLT)
                     write_reg(opout, destreg);

   IRQValues = IRQBus.IRQ;
   if(IRQValues > 0)
   {
         push((int WORD_SIZE) pc);
         /* Now check which interrupt handler is set */
         for(i=7; i<=0 && !quit; i--)
               if(IRQValues[i] == 1)
               {
   /* Get the IRQ handler address from data memory */
                     pc = (unsigned WORD_SIZE) data[i];
                     quit = 1;
               }
   }
}


void main()
{
      par
      {
            while(run_flag)inst_fetch();
            while(run_flag)inst_decode();
            while(run_flag)inst_execute();
            while(run_flag)inst_writeback();

      }
}
```