Australian National University

First Semester Examination 2011

Introduction to Computer Systems

(COMP2300/COMP6300)

Writing Period: 3 hour duration
Study Period: 15 minutes duration
Permitted Materials: One A4 page with notes on both sides.
NO calculator permitted.

The questions are followed by labelled, framed blank panels into which your answers are to be written. An additional answer panel is provided (at the end of the paper) should you wish to use more space for an answer than is provided in the associated labelled panels. If you use the additional panel, be sure to indicate clearly the question and part to which it refers to.

Questions 1 and 2 of this exam redeem the marks of questions 1 and 2 of the mid-semester exam.

The marking scheme will put a high value on clarity so, as a general guide, it is better to give fewer answers in a clear manner than to outline a greater number in a sketchy, half-answered fashion. The Appendix contains a table of powers of 2 in decimal, ASCII codes for characters 'A' to 'Z' in hexadecimal, and the rPeANUt specifications.

Write your answers using a black or blue pen.

Please write clearly – if we cannot read your writing you may lose marks!

Student Number (please write this one very very clearly):

u							

Official use only:

Q1:	Q2:	Q3:	Q4:	Q5:	Total:

Question 1 [15 marks] Student number: .....

(a) How many different values can be stored in a 4 bit word? What is the range of numbers that is stored in a 4 bit word when the two's complement representation is used?

Q1(a) [2 marks]

(b) Convert the hexadecimal number 0xA5B to octal.

Q1(b) [1 marks]

(c) The IEEE 32-bit single-precision floating-point standard is: 1 bit sign, 8 bits exponent with a bias of 127, and the remaining 23 bits are the significand (mantissa). What decimal number does the float 0x42E48000 represent?

Q1(c) [3 marks]

(d) Suppose we had an integer array of length 256, and the size of an integer is 4 bytes. How many such arrays could fit into an 8 KB area of memory?

Q1(d) [3 marks]

(e) In *C,* write a program that reads a (single-word) name from the standard input stream then prints (to the standard output stream) “Hello ” followed by the input name.

Q1(e) [4 marks]

(f) What would the below code print if it was compiled and run?

```
#include <stdio.h>
#include <string.h>
char *foo(char *str) {
   int len = strlen(str);
   char *p1, *p2;
   int i;
   for (i=0;i<=len/2;i++) {
      p1 = &(str[i]);
      p2 = &(str[len-i-1]);
      *p1 = *p1 ^ *p2;
      *p2 = *p1 ^ *p2;
      *p1 = *p1 ^ *p2;
   }
   return str;
}
int
main() {
   char str[] = "0123456";
   printf("%s\n",foo(str));
   return 0;
}
```

Q1(f) [2 marks]

Question 2 [15 marks] Student number: .....

(a) Could any more registers be added to the rPeANUt computer without major changes to the instruction format? If so how many extra registers could be added? If not why not?

Q2(a) [3 marks]

(b) Disassemble the program given in the image below (note your disassembled program should be able to be given to the rPeANUt assembler). Also exactly what will the program output to the terminal when it is run? (note the appendix contains some ASCII codes and the rPeANUt specifications)

Q2(b) [5 marks]

address	data
0x0100	0xc0010107
0x0101	0xc2120000
0x0102	0xa4120106
0x0103	0xd120fff0
0x0104	0x1b110000
0x0105	0xa4000101
0x0106	0x00000000
0x0107	0x00000043
0x0108	0x0000004f
0x0109	0x0000004d
0x010a	0x00000050
0x010b	0x00000055
0x010c	0x00000054
0x010d	0x00000045
0x010e	0x00000052
0x010f	0x00000000

(c) The *C* code below prints a message based on a key press. Convert this *C* code into rPeANUt assembler code. Your solution must implement a procedure for 'printstr'. Also, use the conventional rPeANUt stack frame approach for this procedure.

Q2(c) [7 marks]

```
#include <stdio.h>
void printstr(char *str) {
  while (*str != 0) {
    putchar(*str);
    str++;
  }
}
int main() {
  char c = getchar();
  if (c == 'a') {
    printstr("a was pressed");
  } else {
    printstr("key pressed");
  }
  return 0;
}
```

Question 3 [20 marks] Student number: .....

a) What interrupts are there in rPeANUt? How would you characterise them?

Q3(a) [3 marks]

b) What is a system call? Why are they important for modern operating systems? How are system calls implemented in Linux based x86 machines?

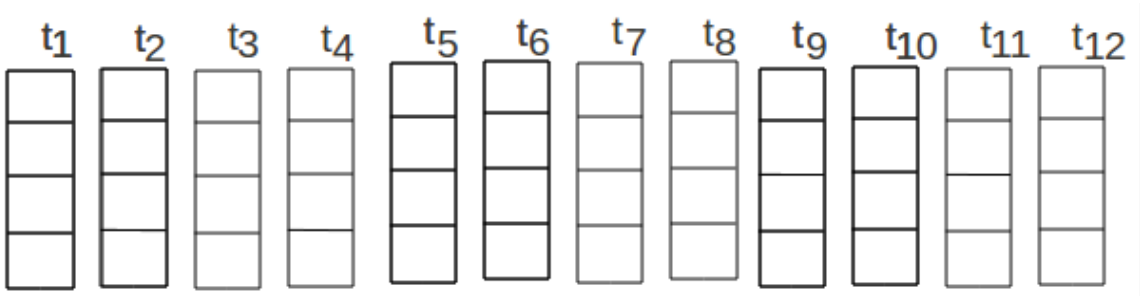
Q3(b) [5 marks]

c) Consider a virtual memory system with a main memory that can hold four memory pages. Assume the least recently used page replacement policy is used by the operating system. The following sequence of 12 page accesses at times t_1 to t_{12} to 6 different pages (numbered 1 to 6) is given:

Time:	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
Page:	3	2	1	6	4	1	5	6	3	2	4	3

(i) Assuming the main memory has been empty at the beginning, complete the following diagram of the four main memory pages at times t_1 to t_{12}. Please write your answers – the page numbers – into the appropriate boxes.

Q3(c.i) [3 marks]



(ii) How many pages have to be loaded in this page access sequence?

Q3(c.ii) [1 marks]

(d) In both paging and set-associative caches, the least recently used (LRU), first-in-first-out (FIFO) and random replacement policies may be used. In general, which of these performs best and which of these performs worst? What general property of the memory access patterns of programs gives rise to this? Briefly explain why. If one policy is generally better, explain in the context of caches why it is not always used in modern computers.

Q3(d) [4 marks]

(e) Briefly describe the Internet Protocol (IP) and the Transmission Control Protocol (TCP). What is the relationship between these two protocols?

Q3(e) [4 marks]

Question 4 [20 marks] Student number: .....

(a) Explain why the latency of a disk read or write, i.e. the time to access the first byte of a sector, can be relatively large. Explain how it is possible to achieve a reasonably high rate of data transfer for subsequent bytes.

Q4(a) [3 marks]

(b) What is a *cache line*, and what lengths are these typically?

Q4(b) [2 marks]

(c) Define the terms *direct-mapped cache* and *E-way set-associate cache*. What are typical values for *E*?

Q4(c) [4 marks]

(d) Suppose you had a CPU with: 8 bit addresses, byte addressable memory, a 2-way set associative data cache with a total of 128 bytes, each cache line containing 16 bytes, a LRU replacement approach is used within each of the sets; also there is no pre-fetching.
(d.i) How many cache lines are there in the cache?

Q4(d.i) [1 marks]

(d.ii) How is the 8 bit address partitioned into tag, set, and word sections (give the number of bits in each section)?

Q4(d.ii) [1 marks]

(d.iii) Assuming the cache is initially completely empty and the below sequence of addresses is read by the CPU. Circle the reads that generate a cache miss.

Q4(d.iii) [2 marks]

0x80, 0x3A, 0xB0, 0x30, 0x8F, 0x7A, 0x30, 0xB7, 0x8A, 0x31

(d.iv) Now assume a direct mapped cache is used with the same size (128 bytes) with the same line size (16 bytes), also assume the cache is initially completely empty. Circle the reads that generate a cache miss (using the same sequence as the above question).

Q4(d.iv) [2 marks]

0x80, 0x3A, 0xB0, 0x30, 0x8F, 0x7A, 0x30, 0xB7, 0x8A, 0x31

(e) Why are caches so important for modern computer systems in terms of performance?

Q4(e) [2 marks]

(f) Suppose you had a computer with a 512 KB level-2 cache. If a program had a working set of 1 MB, would it be possible that the program would run with a low rate of level-2 cache misses? Briefly explain your answer.

Q4(f) [3 marks]

Question 5 [20 marks] Student number: .....

Suppose you have been asked to develop a program that determines if two files are 'similar'. Two files are defined to be 'similar' if and only if:

- the files are exactly the same length, and
- all but at most 10 bytes in the files have the same value in the same position.

The program is provided the names of the two files as arguments. The program gives the result to standard out saying either "similar" or "different".

a) The 'similar' program is a lot like the standard 'diff' program on unix systems. What approaches could you use to determine how 'diff' is implemented to help you develop the 'similar' program? Compare these approaches and give some advantages and disadvantages of each of them.

Q5(a) [5 marks]

b) Implement the 'similar' program in C.

Some lines from the manual pages of open, read, and stat are given below:

open:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
The argument flags must include one of the following access modes:
O_RDONLY, O_WRONLY, or O_RDWR. These request opening the file read-
only, write-only, or read/write, respectively.
 In addition, zero or more file creation flags and file status flags can
be bitwise-or'd in flags. The file creation flags are O_CREAT, O_EXCL,
O_NOCTTY, and O_TRUNC.
mode specifies the permissions to use in case a new file is cre-
ated. This argument must be supplied when O_CREAT is specified
in flags; if O_CREAT is not specified, then mode is ignored.
```

read:

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
read() attempts to read up to count bytes from file descriptor fd into
the buffer starting at buf.
On success, the number of bytes read is returned (zero indicates end of
file), and the file position is advanced by this number.
```

stat:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int fstat(int fd, struct stat *buf);
These functions return information about a file.
All of these system calls return a stat structure, which contains the
following fields:
    struct stat {
        :           :
        off_t     st_size;    /* total size, in bytes */
        :           :
    };
```

Q5(b) [10 marks]

Q5(c) Suppose you have been asked to evaluate the performance of your 'similar' program for different sized 'similar' files (assume these files are stored on a hard-disk). You could time the execution of the program on different sized 'similar' files and produce a graph that has 'file size' on the x-axis and 'time taken divided by file size' on the y-axis. What would you expect this graph to look like? Explain and justify your answer.

Q5(c) [5 marks]

Additional answers to Question .....

Additional answers to Question .....

Appendix

Powers of 2 in decimal:

$2^{-5}=0.03125$ $2^{-4}=0.0625$ $2^{-3}=0.125$ $2^{-2}=0.25$ $2^{-1}=0.5$ $2^{0}=1$

$2^{1}=2$ $2^{2}=4$ $2^{3}=8$ $2^{4}=16$ $2^{5}=32$ $2^{6}=64$

$2^{7}=128$ $2^{8}=256$ $2^{9}=512$ $2^{10}=1024$ $2^{11}=2048$ $2^{12}=4096$

$2^{13}=8192$ $2^{14}=16384$ $2^{15}=32768$ $2^{16}=65536$ . .

ASCII codes in hexadecimal:

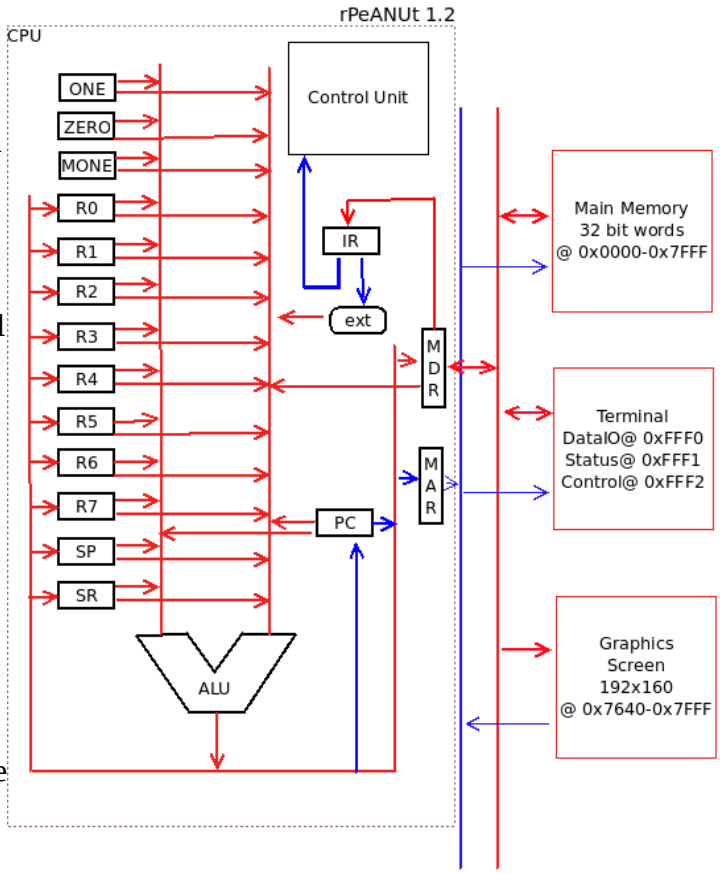
```
0x41 'A',  0x42 'B',  0x43 'C',  0x44 'D',  0x45 'E',  0x46 'F',  0x47 'G',
0x48 'H',  0x49 'I',  0x4A 'J',  0x4B 'K',  0x4C 'L',  0x4D 'M',  0x4E 'N',
0x4F 'O',  0x50 'P',  0x51 'Q',  0x52 'R',  0x53 'S',  0x54 'T',  0x55 'U',
0x56 'V',  0x57 'W',  0x58 'X',  0x59 'Y',  0x5A 'Z'
```

Specification of the rPeANUt Computer and Assembler (v1.2)

Eric McCreath - 2011- Research School of Computer Science - ANU

Introduction: The rPeANUt Computer is a simple microprocessor which was created for teaching computer systems at the ANU. There is a Java implementation of a simulator along with an assembler. This document aims to precisely describe the microprocessor along with the assembler so students can develop code and in so doing gain an idea of what is involved in developing code for a real microprocessor.

Overview: The rPeANUt is a 32 bit microprocessor with: 16 bit addresses, 32 bit register, and memory that is addressable in words of 32 bits. So the total maximum amount of addressable memory is $2^{16}=65523$ words or 262144 bytes (256k). Only the addresses 0x0000 to 0x7FFF are connected to actual memory. Address between 0x8000 and 0xFFFF are used for memory mapped IO (although only 3 of these addresses are actually used). When the microprocessor is reset the instruction pointer (IP) is set to 0x0100, so normally a program will be load at this point for execution. Addresses 0x0000 to 0x00FF are reserved for the interrupt vector and other OS code. Also the last 960 words of actual memory is used for the frame buffer.



The microprocessor contains the following 32 bit registers:

- 8 generally purpose registers these may be used for storing either data or addresses. These are denoted R0, R1, ... R7.
- An instruction register (IR) - which holds the current

instruction that is being executed.

- A status register (SR) - contains status information about the CPU. Bit 0 is used for integer overflow (OF), and bit 1 is used for interrupt mask (IM).
- Three constant registers called ONE, ZERO, and MONE. They contain the constants 1, 0, and -1 respectively.

The microprocessor contains the following 16 bit registers:

- A stack pointer (SP) - this points to the top of the stack and is used for method calls, method returns, and interrupts (SP is set to 0x7000 when the microprocessor is reset).
- A program counter (PC) - which contains the address of the next instruction to execute.

Note that the IR registers is not directly accessible via the instruction set. Although clearly the execution of instructions will effect this register.

The microprocessor also contains a control unit which sequences the movement of data around the CPU. The microprocessor goes through the follow execution cycle:

```
do {
    IR = mem[PC];
    PC = PC + 1;
    execute_instruction in IR;
    check for interrupts;
} while(!halt);
```

Instruction set

Instruction are all 1 word long (32 bits). Registers have the labels R0,R1,...R7, SP, SR, PC, ONE, ZERO, MONE and take a nibble (4 bits) in the machine code. The encoding of this nibble is: R0 is 0x0, R1 is 0x1, ... , R7 is 0x7, SP is 0x8, SR is 0x9, PC is 0xA, ONE is 0xB, ZERO is 0xC, and MONE is 0xD. Addresses and values take 16 bits of the 32 bit machine code instruction. Values are sign extended from 16 bits to 32 bits. The description in the table below assumes the word of the instruction has been loaded into the IR and the PC has been moved to point to the next instruction word.

Name	Assembly Instruction	Machine code	Description
addition	add <RS1> <RS2> <RD>	0x1<RS1><RS2><RD>0000	RD <- RS1 + RS2
subtraction	sub <RS1> <RS2> <RD>	0x2<RS1><RS2><RD>0000	RD <- RS1 - RS2
multiply	mult <RS1> <RS2> <RD>	0x3<RS1><RS2><RD>0000	RD <- RS1 * RS2
divide	div <RS1> <RS2> <RD>	0x4<RS1><RS2><RD>0000	RD <- RS1 / RS2
modulo	mod <RS1> <RS2> <RD>	0x5<RS1><RS2><RD>0000	RD <- RS1 % RS2
bit and	and <RS1> <RS2> <RD>	0x6<RS1><RS2><RD>0000	RD <- RS1 & RS2
bit or	or <RS1> <RS2> <RD>	0x7<RS1><RS2><RD>0000	RD <- RS1 | RS2
bit xor	xor <RS1> <RS2> <RD>	0x8<RS1><RS2><RD>0000	RD <- RS1 ^ RS2
negate	neg <RS> <RD>	0xA0<RS><RD>0000	RD <- - RS
bit not	not <RS> <RD>	0xA1<RS><RD>0000	RD <- ~ RS
copy register	move <RS> <RD>	0xA2<RS><RD>0000	RD <- RS
call immediate	call <address>	0xA300<address>	SP <- SP +1 mem[SP] <- PC PC <- address

Name	Assembly Instruction	Machine code	Description
return from call	return	0xA3010000	PC <- mem[SP] SP <- SP-1
trap	trap	0xA3020000	SP <- SP +1 mem[SP] <- PC PC <- 0x0004 SR <- SR | (1<<1)
jump	jump <address>	0xA400<address>	PC <- address

jump if zero	jumpz <R> <address>	0xA41<R><address>	if (R == 0x00000000) {
			PC <- address
			}

jump if negative	jumpn <R> <address>	0xA42<R><address>	if ((R&0x80000000) == 0x00000000) {
			PC <- address
			}

jump if not zero	jumpnz <R> <address>	0xA43<R><address>	if (R != 0x00000000) {
			PC <- address
			}

reset status bit	reset <BIT>	0xA50<BIT>0000	SR <- SR & ~(1<<BIT)
set status bit	set <BIT>	0xA51<BIT>0000	SR <- SR | (1<<BIT)
push onto stack	push <RS>	0xA60<RS>0000	SP <- SP + 1 mem[SP] <- RS
pop from stack	pop <RD>	0xA61<RD>0000	RD <- mem[SP] SP <- SP - 1
bit left rotate (using immediate)	rotate #<amount> <RS> <RD>	0xB0<RS><RD>00<amount>	RD <- RS << amount | RS >>> (32 - amount)
bit left rotate (using register)	rotate <RA> <RS> <RD>	0xE<RA><RS><RD>00000	RD <- RS << RA | RS >>> (32 - RA)
immediate load from memory	load #<label or value> <RD>	0xC00<RD><value>	RD <- ext(value)
absolute load from memory	load <address> <RD>	0xC10<RD><address>	RD <- mem[address]
indirect load from memory	load <RSA> <RD>	0xC2<RSA><RD>0000	RD <- mem[RSA]
base + displacement load from memory	load <RSA> #<value> <RD>	0xC3<RSA><RD><value>	RD <- mem[RSA + ext(value)]
absolute store to memory	store <RS> <address>	0xD1<RS>0<address>	mem[address] <- RS
indirect store to memory	store <RS> <RDA>	0xD2<RS><RDA>0000	mem[RDA] <- RS
base + displacement store to memory	store <RS> #<value> <RDA>	0xD3<RS><RDA><value>	mem[RDA+ext(value)] <- RS
halt	halt	0x00000000	fetch execution stops!

Hardware and Interrupts
A simple terminal is provided via memory mapped IO. Interacting with this device is done via three addresses: dataIO at 0xFFF0, status at 0xFFF1, and control at 0xFFF2. The least most significant bit of the status register (bit 0) is 1 when data is available and 0 otherwise. Bit 1 of the status register is 0 when it is ready to receive data and 1 if not ready. To write to this device simply write to the memory address of the dataIO location, and to read from this device just read from the dataIO address. Note the status register should be check prior to reading or writing to this device (although good practice to check the status bit before writing, in this simulator it will always be ready for writing, so you may just write directly to the dataIO location). The control address is used to set interrupts on for this device (bit 0 of the control address is 0 if interrupts are disabled and 1 if enabled). If the interrupt bit is set then when the a key is hit an interrupt is generated. Note interrupts are disabled by default.

The simulated computer has a black and white screen which is 192 pixels wide and 160 pixels high (or 0xC0 wide and 0xA0 high). The contents of this screen is determined by a frame buffer which starts at address 0x7C40 and ends at 0x7FFF. Assuming (0,0) is the top left corner of the screen then pixel (x,y) will be bit x%32 of the word at address 0x7C40 + 6*y + x/32. If this bit is 1 then the pixel is white and if the bit is 0 then the pixel is black.

When an interrupt occurs the current PC is pushed onto the stack and the PC is set to the address associated with that interrupt. Any registers used by the interrupt service routine must be saved and restored. The interrupt event also sets the interrupt mask high which should be cleared before the interrupt service routine finishes. The standard 'return' instruction is used to return from interrupts. The interrupts and their addresses are given below:

interrupt	address	description
memory fault	0x0000	This happens when memory is accessed that is not addressable.
IO device	0x0001	This happens when interrupts are enabled on the terminal device and a key is hit.
trap	0x0002	This interrupt happens when the trap instruction is executed.

rPeANUt Assembler

The rPeANUt assembler provides a simple way of converting assembly code into the rPeANUt machine code. It works using two phases. The first is a line by line translation into machine code. As this translation takes place both a symbol table is created and a list of addresses that need resolving. The second phase involves resolving all these missing addresses. Note the assembler writes directly into the hardware's simulated main memory (this is just for simplicity).

Each instruction must be placed on a single line. Lines with no instructions or labels are simple ignored. Any characters after the first “;” on a line are considered comments and ignored.

As the code is assemble instructions and data are placed into the next available address. The process starts at address 0x0000 and can only move forward. If you wish to skip forward to a new address location then you can simply place the address before a “:”. You can not go backwards!

Address labels may consist of alpha numeric characters but must not start with a numeric character. They also must not be keywords (keywords are instructions, register names, and assembler directives). A single address can have multiple names, however, separate lines must be used to achieve this.

Instructions may be placed on a line by them self or after a ":". Instructions have the instruction name followed its parameters. These are all space separated. The names of instructions are given in the table in the instruction section of this document. Please note the order of the instruction parameters is important.

The registers are denoted: R0, R1, R2, … , R7, SP, SR, PC, ONE, ZERO, MONE. Addresses can be given either using an integer (given in base 10 or as hex number, hex numbers are prefixed with 0x) or simply the label. Immediate addresses or values are prefixed with the # symbol. The addressing mode of the load and store operations is determined by the list of parameters.

The "block" keyword is used to reserve a block of memory. If the block keyword then a positive integer k is given then k words are reserved (these are initialised to 0). If block followed by an immediate integer or character then one word is reserved and this word is initialised to the given value. If a string is given using #"<string>" (e.g. #"Hello World") then a block of words is reserved and initialised to that string. A null terminator is used and the characters are stored in the least most significant byte of each word.