What you should know by today

CPU basic architecture
Registers
Instructions
Data representation

- Hexadecimal, little-big endian, 2's representation

Basic architecture of an assembly program

- Included files, comments, structure
- Passing arguments to subroutines
- Returning values when leaving functions

Lecture notes at the following address:

http://www.citr.auckland.ac.nz/~patrice/lecture_notes.html

The Layout of an Assembly Language Program

Example 1:

What does an assembly language program look like?

```
entry main.enter;

import "../IMPORT/register.h";
import "../IMPORT/callsys.h";

//      void main()
//              {
//              while ( TRUE )
//                      {
//                      char c;
//                      c = getChar();
//                      putchar( c );
//                      }
//              }
block main uses register, CALLSYS {
        code {
        public enter:
        loop:
        ldiq $a0, CALLSYS_GETCHAR;
        call_pal CALL_PAL_CALLSYS;
        mov $v0, $a1;
        ldiq $a0, CALLSYS_PUTCHAR;
        call_pal CALL_PAL_CALLSYS;
        br loop;
        end:
        }
}
```

System calls

In fact the instructions for making system call requests are usually put inside functions, and the functions are called instead.

```
block Sys {
	// char getChar()
	//	{
	//	Read a character from the simple terminal;
	//	}
public block getChar uses proc, CALLSYS
	{
	 code
		{
		public enter:
		lda $sp, -sav0($sp);
		stq $ra, savRet($sp);
		body:
		ldiq $a0, CALLSYS_GETCHAR;
		call_pal CALL_PAL_CALLSYS;
		return:
		ldq $ra, savRet($sp);
		lda $sp, +sav0($sp);
		ret;
		}
	}
```

```
// void putChar( char c )
//      {
//      Write a character to the simple terminal;
//      }
public block putChar uses proc, CALLSYS
        {
        code
                {
                public enter:
                //Next 2 lines to store registers into memory
                lda $sp, -sav0($sp);
                stq $ra, savRet($sp);
                body:
                //move content of register a0 into register a1
                mov $a0, $a1;
                // put constant value (quadword) CALLSYS_PUTCHAR into register a0
                ldiq $a0, CALLSYS_PUTCHAR;
                //make a request to the OS to do something (print a Char)
                call_pal CALL_PAL_CALLSYS;
                return:
                //Next 2 lines to restore registers value previously store into memory
                ldq $ra, savRet($sp);
                lda $sp, +sav0($sp);
                ret;
                }
        }
...
}
```

lda $sp, -sav0($sp);
stq $ra, savRet($sp); save registers on entry of a function

ldq $ra, savRet($sp);
lda $sp, +sav0($sp);
ret; restore registers on exit of a function

Printing a character: performed by calling the function *Sys.putChar.enter*
Getting a character: performed by calling the function *Sys.getChar.enter*

Memory allocation for variables, data, constants, strings

By definition saved registers can be used to store the values of your program variables.

- Good for small programs
    - easy to run out of registers to use for simple variables:
        - Only 6 saved registers
- Registers (8 bytes long) can only be used to contain simple values:
    - Integers, characters, boolean values, etc…
- Arrays and strings are too big to be stored in a register, and have to be stored in memory.

Space for string constants can be allocated in the constant section.
Space for variables and arrays can be allocated in the data section.

Rules:

To allocate space, we need:

- An alignment statement
- A label to refer to the memory address where data is stored
- A memory allocation statement. We can initialise memory, by specifying a data type, followed by the initial value, then a ";".

```
const {
align quad;
message1:
asciiz "Type some input: ";
align quad;
message2:
asciiz "The input was: ";
}
```

Data types can be keywords such as byte, ubyte, quad, ascii, asciiz, etc, to allocate space for a signed byte, unsigned byte, signed quadword, unterminated ASCII string, null terminated ASCII string, etc.
Apart from the data types corresponding to strings, memory allocation instructions allocate the appropriate amount of memory in the relevant section (1 byte for byte and ubyte, 2 bytes for word and uword, 4 bytes for long and ulong, 8 bytes for quad and uquad, 4 bytes for float, 8 bytes for double).

Difference between the signed and unsigned variants:

- Check if the value is in the range.

For the ascii directive:

- The number of bytes allocated is equal to the length of the string.
- The content is the data within the string.

For the asciiz directive:

- Similar with an extra zero byte allocated and added on the end.

To allocate data that is initially zero.

```
data {
c: quad;
d: quad;
}
```

To allocate blocks of memory, by declaring an array:

```
data {
align quad;
buffer:
byte [ BUFFERSIZE + 1 ];
}
```

Memory statements (with no initial values provided) usually only occur within a data section.

Data has to be aligned to be accessed properly

Alignment statements are used to round the current address up to a multiple of the size of a specified type.

- Good idea to align data labels to quadwords, no matter what the size of the data.
- If labels are not at least aligned to longwords, then the memory display in the simulator will be confused.

Exercise

Suppose we have the following alpha assembly language

```
data {
align quad;
message:
asciiz "0x12";
align quad;
value:
quad 0x123456789a;
}
```

Indicate the contents of each byte of memory in hexadecimal.

label	address	content	label	address	content
	0x1000000			0x1000008	
	0x1000001			0x1000009	
	0x1000002			0x100000a	
	0x1000003			0x100000b	
	0x1000004			0x100000c	
	0x1000005			0x100000d	
	0x1000006			0x100000e	
	0x1000007			0x100000f	

The label **message**, is at address 0x1000000

Getting character from the screen:

New section definitions

A const section is composed of the data for string constants, etc., that will not be altered.

A data section is composed of the data for global variables that might be altered.

```
// char buffer[ BUFFERSIZE + 1 ];
// void main() {
// while ( TRUE ) {
// print( "Type some input: " );
// readline( buffer, BUFFERSIZE );
// print( "The input was: " );
// print( buffer );
// newline();
// }
// }
block main uses proc {
        abs { //absolute section: provide symbolic names for constants->easier
                NEWLINE = '\n';
                BUFFERSIZE = 200;
                }
        const {//allocate memory for data which will not changed
                message1: //contain the memory address of the first byte of string
                //asciiz: extra zero byte allocated and added at the end of a string
                asciiz "Type some input: ";
                message2:
                asciiz "The input was: ";
                }
        data {//allocate memory for data which may be altered
                buffer: //allocate blocks of memory by declaring an array
                byte [ BUFFERSIZE + 1 ];
                }
```

```
code { //code section :specify instructions to execute
    public enter:
        {
        loop:
        //ldiq: load immediate quadword
        ldiq $a0, message1; //load value message1 into register $a0
        bsr IO.print.enter; //branch to subroutine IO.print.enter
        ldiq $a0, buffer;
        ldiq $a1, BUFFERSIZE; $a1 contains the value BUFFERSIZE
        bsr IO.readLine.enter;
        ldiq $a0, message2;
        bsr IO.print.enter;
        ldiq $a0, buffer;
        bsr IO.print.enter;
        bsr IO.newline.enter; //function which position cursor to the next line
        br loop;
        end:
        }
    }
}
```

Several calls to functions *IO.print.enter*, *IO.readLine.enter*, etc… to generate actions such as reading a line, writing a line, going to the next line, etc…

- <u>These functions have to be programmed: not already included in the assembly simulator !!!</u>

Example: printing a line

```
block IO
    {
    ...
// void print( char *s ) {
// while ( *s != 0 ) {
// putChar( *s );
// s++;
// }
// }
    public block print uses proc
        {
        abs
            {s = s0;}
        code {
        public enter:
        lda $sp, -sav1($sp);
        stq $ra, savRet($sp);
        stq $s0, sav0($sp);
        body:
        mov $a0, $s; // Pointer to char in string

        while:
            ldbu $a0, ($s); // Get character
            beq $a0, end; // Break if at end of string
        do:
            bsr Sys.putChar.enter; // Print char
            addq $s, 1; // Increment pointer
            br while;
        end:
        return:
        ldq $s0, sav0($sp);
        ldq $ra, savRet($sp);
        lda $sp, +sav1($sp);
        ret;
        }
  }
  ….
```

Assume the following directives have been used to reserve locations in the memory:

```
data{
align quad;
a:  byte 0x12
align quad;
b:  word 0x9876
align quad;
c:  long 0x89012345
align quad;
d:  quad 0x1234567890123456
}
```

Fill the memory assuming that label **a** is stored at starting address 0x1000000

	address	content		address	content		address	content		address	content
a	00			08							
	01			09							
	02			0a							
	03			0b							
	04			0c							
	05			0d							
	06			0e							
	07			0f							

The following directives have been used to reserve memory locations.

```
data {
a:   byte 0x12;
b:   long 0x34567890;
c:   byte 0xab;
d:   word 0xcdef;
e:   long 0x87654321;
     }
```

Show the contents of the memory and the labels for various locations assuming that label **a** is stored at starting address 0x10…00.

label	address	content	label	address	content
	0x10…00			0x10…08	
	0x10…01			0x10…09	
	0x10…02			0x10…0a	
	0x10…03			0x10…0b	
	0x10…04			0x10…0c	
	0x10…05			0x10…0d	
	0x10…06			0x10…0e	
	0x10…07			0x10…0f	