LAB 6

Lab 6 involves interfacing to the IBM PC parallel port. Use the material on www.beyondlogic.org for reference.

This lab requires the use of a Digilab board. Everyone should have one, but at least each group working together needs to have a board.

Parallel Port Registers

Address: Base + 0 => Data Port
         Base + 1 => Status Port
         Base + 2 => Control Port

Status port bits:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Signal Name</th>
<th>Pin #</th>
<th>Inverted</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Busy</td>
<td>11</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>Ack</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Paper Out</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Select In</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Error</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IRQ</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>1</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Control Port Bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Signal Name</th>
<th>Pin #</th>
<th>Inverted</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Unused</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Unused</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Enable bi-directional port</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Enable IRQ via ACK line</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Select Printer</td>
<td>17</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Initialize Printer (Reset)</td>
<td>16</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>Auto Linefeed</td>
<td>14</td>
<td>Yes</td>
</tr>
<tr>
<td>0</td>
<td>Strobe</td>
<td>1</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Digilab Board Initialization:**

A program will be provided on the lab machines that is used to initialize the Digilab boards. It is called `cfglab6.exe`, and will be placed in a directory on the path, so that it can be executed simply by typing its name anywhere from a DOS prompt.

**C Programming Language Semantics**

Parameter passing to procedures:

C is ‘call by value’ for scalar variables, ‘call by reference’ for vector variables.

‘Call by value’ means that the value of the parameter is pushed onto the stack and passed to the procedure as the parameter.

‘Call by reference’ means that the address of the parameter is pushed onto the stack.

When a variable is passed by value, the called procedure can’t modify the value of the parameter in the caller’s context.

When a variable is passed by reference, then the called procedure has a pointer to the actual variable and not a copy of its value. In this case, the called procedure can modify the value of the caller’s variable.

(In some languages, such as FORTRAN, which are call by reference for scalar variables, it is possible to modify the value of constants)

**Variable Initialization:**
In C, when a variable is declared, it is possible to give it an initial value by specifying an initializer. When the variable has static storage class (either by using the static keyword when declaring a local variable, or by declaring it outside of a procedure) the initialization is done statically by the compiler when the program is compiled, and no runtime code is required to perform the initialization.

However, when an automatic variable, in a procedure, is initialized, it is necessary for the compiler to generate code to perform the initialization. Initialization of automatic variables in a procedure must be performed each time the procedure is entered, before any other code in the procedure is executed.

**In-Line Assembler:**

Most C compilers support some form of in-line assembler. This is a non-standard extension to the C language, therefore,

- the syntax varies from one vendor’s assembler to the next.
- Also, any in-line assembler code is non-portable between processors and sometimes between operating systems on the same processor.

With the Borland C++ compiler used in our lab, the syntax is:

```
_asm opcode operands
```

or:

```
_asm {
    opcode operands
    opcode operands
    .
    .
    .
}
```

- The first syntax allows a single assembly language statement to be entered.
- The second syntax allows a block of assembly language statements to be entered.

**Please note:** With the Borland compiler, the opening brace (\{) must be on the same line as the _asm keyword. If the brace isn’t on the same line, the compiler won’t recognize it as an entry into in-line assembly mode and will report numerous syntax errors on the assembly language statements.
The Borland compiler also does not allow the definition of statement labels within the in-line assembly code. If you need to create a label for the target of a jump in assembler, you need to close the in-line assembly block, create the label using the C syntax for labels, and then enter a new in-line assembly block.

```
_asm {
    cmp ax,bx       ;asm code
    jnz foo         ;jump to other asm code
    xchg ax,bx
}
foo:              //C statement label
_asm {
    mov dx,ax      ;other asm code
    ...
    ...
}
```

**Initialization of Variables**

In the C (or C++) programming language any variable definition can have an optional initializer for the variable. How and when the initialization occurs is determined by the storage class of the variable. The syntax is similar to the following:

```
int foo = 10;
```

The term storage class refers to how and when memory space (storage) for the variable is allocated.

In C there are basically two options for storage class: static and automatic.

- The memory for static variables is allocated by the compiler at the time of compilation. In an ‘x86 system, the compiler will reserve space in the program’s data segment at compilation time to store the variable.
- The memory space for automatic variables is allocated at run time on the stack. The compiler will generate code to allocate space for automatic local variables on entry to a procedure and de-allocate the space on exit from the procedure.
The storage class assigned to a variable is determined by how and where the variable is defined. **All variables defined outside the scope of any procedure (i.e. outside the body of any subroutine or function) must be static.** Thus, global variables are, and must be static.

By default, **variables defined within a procedure (local variables) will be of storage class automatic.** It is possible to specify that a local variable within a procedure be given static storage class by using the keyword `static` at the beginning of the declaration, as follows:

```
static int foo = 10;
```
The assignment of initial values is determined by the variable’s storage class. All static variables will be initialized once before execution of the program begins. Generally, this is done by the compiler initializing the memory allocated in the data segment for all initialized variables.

Most C compilers for the ‘x86 processors actually generate two data segments one for initialized variables and one for uninitialized variables, which will be combined by the linker into a single segment.

Automatic variables are initialized each time that the procedure where they are declared is executed. Because the memory is allocated on the stack automatically on entry to the procedure, the initialization must also be performed on each entry to the procedure.
For scalar variables, the initialization could look something like this:

C Code:
```c
void subr(void)
{
    int x = 1;
    int y = 2;
    x = x+y;
    return x
}
```

ASM Code
```
subr proc near
push bp
mov bp,sp
sub sp,4
mov [bp-2],1  ;initialize x
mov [bp-4],2  ;initialize y
mov ax,[bp-2]
add ax,[bp-4]
mov sp,bp
pop bp
ret
subr endp
```

Notice that the compiler would generate code to perform the initialization that would explicitly assign a constant value to the variable. This is essentially the same code that would be generated by an assignment statement.

In C, initializing a scalar variable via an initializer on the declaration and via an explicit assignment statement cause the compiler to generate essentially the same code.