Exercises 2
Memory mapped I/O, Timers, and Interrupts

Computer Organization and Components / Datorteknik och komponenter (IS1500), 9 hp
Computer Hardware Engineering / Datorteknik, grundkurs (IS1200), 7.5 hp

KTH Royal Institute of Technology
Friday 18th December, 2020

Suggested Solutions

Memory Mapped I/O

1. (a)
```
lui     $t1,0x8000
ori     $t1,$t1,0x20f0
addi    $t0,$0,0x4F
sw      $t0,0($t1)
```

The address is 32-bit long. We therefore need two instructions `lui` and `ori` to write the address into a register. To display digit 3, the bits for indices 0,1,2,3, and 6 need to be set. Hence, the binary value that should be written is `1001111`, which is the same as the hexadecimal value `0x4F`.

(b) An alternative would be to use the pseudo instruction `li` (load immediate). See the MIPS reference sheet for details.

(c) An alternative to use `ori` could, in this case, have been to use `addi`.

```
lui     $t1,0x8000
addi    $t1,$t1,0x20f0  # potential problem
addi    $t0,$0,0x4F
sw      $t0,0($t1)
```

Note, however, that this does not work in the general case. For instance, if the most significant bit of the 16-bit immediate value is 1, then `addi` (the one marked "potential problem") would perform a subtraction, since the immediate value is negative. Hence, this approach cannot be used when the lower 16 bits can be interpreted as a negative number because `addi` is always sign extending its immediate value.

An even shorter version would have been to use the index of the store word instruction.

```
lui     $t1,0x8000
addi    $t0,$0,0x4F
sw      $t0,0x20f0($t1)
```

This version works in this case, but the index value for `sw` has the same problem as above because the index immediate value is sign extended.

2. The following C code shows a possible solution.

```
volatile int* push_buttons = (volatile int*) 0x8000abc0;
volatile int* LED = (volatile int*) 0x80007bc0;
while(1){
  if((*push_buttons) & 8)
    *LED = 0x3f;
  else
    *LED = 0x0;
}
```

   Note that the volatile keyword must be used to make sure that the compiler does not optimize the memory mapped I/O accesses.

3. (a) The direction of PORTE is controlled by the TRISE device-register. This register can be accessed at address `0xbf886100`.

   To make the C statement more readable, we add the following pointer definition to the beginning of the program:

   ```
   volatile int * trise = (volatile int *) 0xbf886100;
   ```

   To change a PORTE pin into an output pin, the corresponding bits in the TRISE register must be cleared to zero.

   We must not change any other bits, so a read-modify-write sequence is necessary:

   ```
   *trise = *trise  & 0xfff1;
   ```

   As you see, we only care about the 16 least significant bits of TRISE. Even though TRISE is a 32-bit device-register, only the lower 16 bits are used. As an alternative, note how we can use the bit-wise negation operator $\sim$.

   ```
   *trise = *trise & ~0xe;
   ```

   There is, however, another way. The TRISECLR device-register, at address `0xbf886104`, has the following useful properties when a value is written:

   - Any bits which are ones (1's) when writing to TRISECLR will clear-to-zero the corresponding bits in TRISE.
   - Any bits which are zeroes (0's) when writing to TRISECLR will not change the corresponding bit in TRISE.

   You can read a little more about the TRISECLR device-register in Section 12.3.8 (I/O Ports) of the PIC32 Family Reference Manual. Therefore, we can achieve our goal in a different way, by writing `0xE` to TRISECLR.

   Our previously declared pointer is a pointer-to-int (or, more precisely, a pointer to volatile `int`); our MIPS system uses 4 bytes for an `int`, and the address of TRISECLR is exactly 4 bytes greater than the address of TRISE. Therefore, TRISECLR can be accessed as `(trise + 1)`:

   ```
   *(trise + 1) = 0xe; /* TRISECLR = 0xe */
   ```

   The following statement has the same effect, and is a little more elegant:

   ```
   trise[1] = 0xe; /* TRISECLR = 0xe */
   ```

   It is of course also possible to define a new pointer:

```
volatile int * triseclr = (volatile int *) 0xbf886104;
*triseclr = 0xe;
```

Finally, the one-liner, which is concise but hard to understand:

```
*((volatile int *) 0xbf886104) = 0xe;
```

(b) To be able to use the names TRISE and/or TRISECLR, our program file must include the header-file `pic32mx.h`:

```
#include <pic32mx.h>
```

This file contains definitions for all device-registers in our MIPS processor. Now we can write:

```
TRISE = TRISE & ~0xe;
```

or use TRISECLR:

```
TRISECLR = 0xe;
```

If you like, you can calculate any device-register address from the information in the file pic32mx.h. There is, however, a pitfall. Since PIC32 processors are configurable, only the 20 lower bits are listed for each entry. The C compiler combines these bits with 12 bits from another source, to generate the complete address.

In our case, the higher 12 bits are always `0xbf8`. As an example, look up `TRISE` in the file. You will see the address listed as `0x86100`. Put `bf8` before the `86104`, and remember that `0x` just means hexadecimal, and you get `0xbf886104`.

(c) When the C compiler optimizes your program, the compiler looks for code that seems to read the same location several times in a loop, or something similar. If the compiler finds such code, it is simplified into code that only reads the location once. This is fine for ordinary program variables, but will destroy code that performs input/output.

The `volatile` keyword means that a location may change in ways the compiler can't see, and stops optimization from changing how a volatile location is used.

(d) We can use the address – PORTE is at `0xbf886110`:

```
volatile int * porte = (volatile int *) 0xbf886110;
*porte = 0xa;
```

The code will, however, be more readable if we instead use the PORTE name:

```
#include <pic32mx.h> /* This line should appear only once, */
                     /* near the beginning of the file */
PORTE = 0xa;
```

Sometimes, a shift-operator can increase readability:

```
PORTE = (5 << 1); /* Write binary 101 to bits 3, 2, and 1 */
```

Note that this shift operation does not imply any extra cost when running the program; the shift operation is performed at compile time by the compiler.

(e) Yes, PORTE is connected to 8 green LEDs (light-emitting diodes) on the Basic I/O Shield. Writing a '1' will light up the corresponding LED.

(f) The PORTE device-register behaves like a memory cell. Reading PORTE will return the value that was last written.

Not all device-registers have this memory-like behavior. For example, reading the TRISECLR register will return an undefined value.

Timers

4. (a) For TMR2, TCKPS is a three-bit field consisting of bits 6, 5, and 4, of the T2CON device-register. See the TxCON part of the figure on page 14-6 in the PIC32 Family Reference Manual. The bits of the TxCON register are described in greater detail on pages 14-11 and 14-12 of the manual.

The timer has several options, which can be enabled by setting (to '1') various bits in T2CON. To get a well-defined behavior, we want to disable these options. Furthermore, we want to stop the timer from running until we have finished initializing it. Therefore, we should not use a read-modify-write sequence. Instead, we simply write our desired value.

To get a 1:64 prescale value, the bits should be 110. Therefore, we write:

```
T2CON = 0x60;
```

This also stops the timer, by clearing (to zero) the ON bit, which is bit 15 in T2CON.

(b) To arrive the correct period value, consider the base clock-rate of 80 000 000 clock cycles per second. The clock signal is divided by 64 in the prescaler, so the timer counts at a rate of (80 000 000 / 64).

We don't want a one-second period, we want the period to be 1/100 of a second. Therefore, the correct period is (80 000 000 / 64) / 100.

We don't want to use a pocket calculator to perform the necessary arithmetic. The compiler can do it for us, and including the calculation into the code will make the program easier to understand. So we may write:

```
PR2 = (80000000 / 64) / 100;
```

There is a pitfall here: the timer uses a 16-bit counter. If the result from our calculation is too large to fit into 16 bits, this problem may go undetected. So we may want to use the pocket calculator after all, unless we are prepared to let the compiler do a little extra work. This is a little advanced, so if you're in a hurry, just skip to the next question.

We define a compile-time constant for our period:

```
#define TMR2PERIOD ((80000000 / 64) / 100)
```

Then, we use the `#if`, `#error`, and `#endif` compiler directives to check the value:

```
#if TMR2PERIOD > 0xffff
#error "Timer period is too big."
#endif
```

Finally, we use the value to initialize the timer:

```
PR2 = TMR2PERIOD;
```

At the `#if`, the compiler will check the constant TMR2PERIOD against the other constant, `0xffff`. If TMR2PERIOD is greater, the `#error` directive will stop the compilation with an error. The `#endif` is necessary to specify the end of the test.

(c) Resetting the counter will ensure that the timer starts at a well-defined value, when we start it, sometime later.

The counter can be accessed directly as TMR2:

```
TMR2 = 0;
```

(d) Before starting the timer, all other initialization of the timer should be finished. The timer can then be started by setting the ON bit to a '1'. The ON bit is bit 15 in T2CON, and can be set to '1' as follows:

```
T2CONSET = 0x8000; /* Start the timer */
```

(e) This question is left as an exercise without an explicit solution.

(f) There must be a way for the statement to indicate the result, so we write a function. The function will return 1 if there was a time-out, and 0 otherwise.

```
int timertest(void)
{
  if(IFS(0) & 0x100){     /* Test time-out event flag */
    IFS(0) = 0;           /* Reset all event flags (crude!) */
    return(1);
  }
  else
    return(0);
}
```

Resetting all event flags can cause problems if your program should receive events from more than one device. However, in lab 3, we will only receive events from TMR2.

Interrupts

5. (a) Registers $1 through $15, $24, and $25 can be changed by any C function, or any assembler subroutine. Register $ra will be changed by the call to the C function, i.e., the jal instruction. Saving these registers is necessary before calling the C function `user_isr`.

(b) To enable interrupts from TMR2, three steps are necessary.

The first two steps are to write a one to the T2IE bit in IEC0, and to write a non-zero priority value to the three T2IP bits in IPC2. See page 53 in PIC32MX3XX/4XX Family Data Sheet. Please note that the syntax is different in our system: instead of IEC0 and IPC2, we must write IEC(0) and IPC(2).

The third and final step is to execute an `ei` instruction to enable interupts globally. To this end, we call the function `enable_interrupt` (which can be written in assembly code).

```
IEC(0) = (1 << 8);
IPC(2) = 4; /* Values between 4 and 31 will work */
enable_interrupt();
```

(c) This statement is the same as without interrupts. However, the conditions for executing the statement is quite different.

Here is the statement itself. Please note that the syntax in our system is slightly different from the manual: instead of IFS0, we must write IFS(0).

```
IFS(0) = 0; /* Reset all event flags (crude!) */
```

But in this case, we must place this statement inside an interrupt service routine (ISR):

```
void user_isr(void){
  IFS(0) = 0;
  /* More code must follow here! */
}
```