ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Prof. L. Thiele

Technische Informatik 1 - HS 2011

Übung 5

Datum: 10.11.2011

1 Bit-Test

Zum überprüfen des Status eines I/O- Devices muss oft ein einziges Bit geprüft werden. Implementieren Sie sowohl in C als auch in MIPS-Assembler dafür eine Funktion `int bittest(int x, int n)`. `bittest` soll `1` zurückgeben, falls das Bit an der Position `n` in `x` gesetzt ist, andernfalls `0`.
Hinweis 1: Die MIPS Procedure Call Convention, die schon in der Vorlesung bzw. früheren Übungen besprochen worden ist, können Sie dem Unterkapitel A.6 in "Patterson, Hennessy: Computer Organization and Design" entnehmen (online verfügbar: http://www.cs.wisc.edu/~larus/HP_AppA.pdf).
Hinweis 2: Zur Implementierung der Funktion können Sie die folgenden Instruktionen verwenden:

sllv (shift left logical variable):	`sllv $rd, $rs, $rt`	\$rd = \$rs << \$rt
srlv (shift right logical variable):	`srlv $rd, $rs, $rt`	\$rd = \$rs >> \$rt

2 Polling eines Memory-Mapped I/O Devices

In dieser Übung wird die I/O-Verarbeitung mit einem MIPS Prozessor betrachtet. Dazu wird der SPIM (SPIM = MIPS rückwärts gelesen) Simulator benutzt, der neben dem CPU Kern auch *memory-mapped I/O* und *Interrupt-Verarbeitung* simuliert. Der SPIM Simulator verarbeitet ausschliesslich Assembler Code und hat eine einfache, grafische Benutzeroberfläche, die mit dem Kommando `xspim` gestartet wird.

SPIM simuliert zwei memory-mapped Devices: Das Eingabe-Device ermöglicht die Eingabe von Zeichen über die Tastatur. Das Ausgabe-Device ermöglicht die Anzeige von Zeichen auf dem Bildschirm. Prinzipiell sind die beiden Devices voneinander unabhängig. Im Folgenden wird kurz die Funktion der beiden Devices erläutert, siehe auch Abbildung 1.

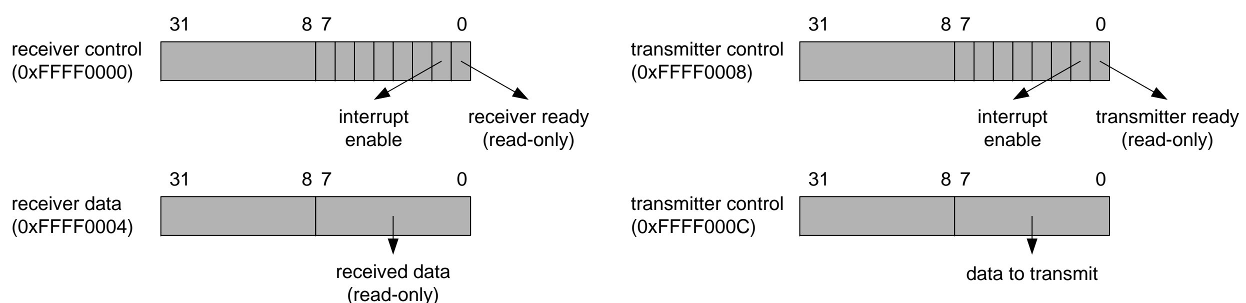

Abbildung 1: Register zur Steuerung der I/O-Devices.

Eingabe-Device. Das Eingabe-Device wird über zwei Register gesteuert, die in den Speicherbereich des MIPS Prozessor eingeblendet sind (daher der Name memory-mapped Device). Das Register *receiver control* ist an der Speicheradresse 0xFFFF0000 eingeblendet. Für diese Aufgabe ist nur das Bit *receiver*

ready des Registers *receiver control* relevant: Das Bit *receiver ready* wechselt von 0 auf 1, sobald ein Zeichen auf der Tastatur eingegeben wurde. Bei jedem Lesezugriff auf das Register *receiver data* wechselt das Bit *receiver ready* von 1 auf 0. Im Register *receiver data*, das an der Speicheradresse 0xFFFF0004 eingeblendet ist, wird der ASCII-Wert des zuletzt eingegebenen Zeichens gespeichert.

Ausgabe-Device. Das Ausgabe-Device wird ebenfalls über zwei Register gesteuert. Das Register *transmitter control* ist an der Speicheradresse 0xFFFF0008 eingeblendet. Für diese Aufgabe ist nur das Bit *transmitter ready* des Registers *transmitter control* relevant: Das Bit *transmitter ready* ist dann 1, wenn das Ausgabe-Device bereit ist, ein Zeichen auf dem Bildschirm anzuzeigen. Wenn das Bit *transmitter ready* 1 ist, kann in das Register *transmitter data*, das an der Speicheradresse 0xFFFF000C liegt, der ASCII-Wert eines Zeichens geschrieben werden, das als nächstes angezeigt werden soll. Ein Schreibzugriff auf das Register *transmitter data* hat zur Folge, dass das Bit *transmitter ready* auf 0 gesetzt wird.

2.1 Aufgaben

In dieser Aufgabe soll ein Programm erstellt werden, das die Echo-Funktion implementiert. Unter *Echo* versteht man, dass ein vom Benutzer eingegebenes Zeichen am Ausgabe-Device angezeigt wird.

Es soll Polling verwendet werden, um die Echo-Funktion in SPIM zu implementieren. Das folgende Pseudo-Programm zeigt grob die Implementierung der Echo-Funktion mittels Polling:

```
while (1) {
  wait_until_character_received();
  x = read_character();
  wait_until_transmitter_ready();
  write_character(x);
}
```

---

a) Zeichnen Sie ein Zustandsdiagramm, das die Echo-Funktion modelliert. Verwenden Sie die Bits *receiver ready* und *transmitter ready* zur Definition von Ereignissen, die Zustandsübergänge auslösen.
b) Geben Sie ein C Programmfragment an, das ein Zeichen vom Receiver-Device liest. Implementieren Sie dazu in C die beiden Funktionen `wait_until_character_received()` und `read_character()`. Verwenden Sie in `wait_until_character_received()` Polling, um auf die Verfügbarkeit eines Zeichens zu warten.
c) Erstellen Sie ein Assembler Programm, das die Echo-Funktion wie im oben gezeigten Pseudo-Programm implementiert. Im Verzeichnis "polling" finden Sie dazu ein entsprechendes Template `polling.s`:

```
.text
.globl main
main:
    addi $sp, $sp, -4   # save $ra.
    sw   $ra, 0($sp)
    lui  $t0, 0xffff    # $t0 = base address for I/O register
                        # use lw $t1, offset ($t0) to load the word at memory address
                        # (0xffff0000 + offset) into register $t1
loop:
    # ***** YOUR CODE GOES HERE *****
    j    loop           # endless loop
```

Um das Programm mit SPIM zu simulieren, muss die memory-mapped I/O Funktion explizit aktiviert werden. Starten Sie dazu xspim mit der entsprechenden Kommandozeilenoption:
`xspim -mapped_io polling.s`.

3 Interrupts (Zusatzaufgabe)

Verwenden Sie nun Interrupts, um die Zeichen vom Receiver Device zu empfangen. Lesen Sie zunächst dazu die Unterkapitel A.7 und A.8 aus "Patterson, Hennessy: Computer Organization and Design" (online verfügbar: http://www.cs.wisc.edu/~larus/HP_AppA.pdf).

a) Zunächst müssen Sie die CPU und das I/O-Device entsprechend konfigurieren. Wie aktiviert man die Interrupts der I/O-Devices? Welche Hardware Interrupt-Levels werden dabei verwendet? Schreiben Sie ein Assembler Programm, das Interrupts für das Eingabe-Terminal aktiviert.

b) Implementieren Sie nun den Interrupt Handler als Assembler Programm. Im Interrupt Handler soll das gerade eingegebene Zeichen im Ausgabe-Terminal angezeigt werden. Verwenden Sie für die Ausgabe Polling (wie in der letzten Aufgabe). Benutzen Sie das folgende Template:

```
.ktext 0x80000180 # forces interrupt routine to be located at 0x80000180
interrupt:
.set noat         # tell assembler to stop using assembly temporary ($at)
sw $at, save_at   # now we can safely use it
.set at           # give back $at to the assembler

# ***** YOUR CODE GOES HERE *****
# save registers (use words in kernel data section for this purpose)
# implement interrupt handler
# restore registers

.set noat
lw $at,save_at
.set at
eret              # exception return

.kdata
save_at: .word 0
save_t0: .word 0
save_t1: .word 0
save_t2: .word 0
save_t3: .word 0
```

c) Verwenden Sie die Assembler Programme aus a) und b), um das zur Verfügung gestellte Template `interrupt.s` zu vervollständigen. Damit Sie die Unterbrechung des Hauptprogramms durch den Interrupt Handler verfolgen können, gibt das Hauptprogramm kontinuierlich Punkte aus.

   Verwenden Sie bei der Simulation mit SPIM folgende Kommandozeilenoptionen, um memory-mapped I/O zu aktivieren und den standardmässigen Interrupt/Exception Handler durch Ihr eigenes Programm zu ersetzen: `xspim -noexception -mapped_io interrupt.s`