RA FS 18 Serie 4

Qiyang Hu, Givi Meishvili, Adrian Wälchli

Die vierte Serie ist bis Dienstag, den 1. Mai 2018 um 15:00 Uhr zu lösen und auf ILIAS hochzuladen. Für Fragen steht im ILIAS jederzeit ein Forum zur Verfügung. Allfällige unlösbare Probleme sind uns so früh wie möglich mitzuteilen, wir werden gerne helfen. Viel Spass!

Theorieteil

Gesamtpunktzahl: 12 Punkte

1 Sign Extension (1 Punkt)

Erläutern Sie, weshalb die Sign-Extension gerade bei Branch-, Load- und Store-Befehlen gebraucht wird.

2 Logische und bitweise Operationen (1 Punkt)

Bestimmen Sie die jeweilige Ausgabe:

Eingabe	Ausgabe
`0xA3 & 0x3A`	
`0x0 && 0xEF`	
`0xA3 | 0x3A`	
`0x0 || 0xEF`	
`~0xFE`	
`!0xFE`	

3 Hardwarezugriff (1 Punkt)

Wie könnte ein Prozessor auf verbundene Hardware zugreifen?
Benutzen Sie die MIPS Emulation als Inspiration.

4 `bne` statt `beq` (2 Punkte)

Welche Änderungen sind bei der in der Vorlesung vorgestellten MIPS-Implementation (siehe "Basic MIPS Architecture Review") notwendig, um `bne` statt `beq` zu implementieren.

(a) Für die Singlecycle-Implementation

(b) Für die Multicycle-Implementation

5 Pipeline Registers (1 Punkt)

Wozu werden die Register (siehe Folie 6, "Basic MIPS Pipelining Review") zwischen den einzelnen Berechnungsstufen benötigt?

6 Pipelining Hazard (2 Punkte)

Erklären Sie den Unterschied zwischen Control-, Data- und Structural-Hazards.

7 Stall (2 Punkte)

Erläutern Sie, warum es auf Folie 15, im Gegensatz zur Folie 19, genügt, nur zwei Taktzyklen zu warten. (Die Seitenangaben beziehen sich auf das Kapitel "Basic MIPS Pipelining Review")

8 Data Hazard (2 Punkte)

Geben Sie sämtliche Data Hazards im folgenden Code an. Bei welchen Abhängigkeiten handelt es sich um Data Hazards, die mittels Forwarding behoben werden können? Bei welchen Abhängigkeiten handelt es sich um Data Hazards, die zu einem *stall* führen?

```
1 add $t0, $t5, $t4
2 lw $s2, 0($t0)
3 sub $s3, $t0 ,$s2
4 sw $t4, 4($s3)
```

Programmierteil

Die Programmieraufgaben mit dem Raspberry Pi sind in Zweier- oder Dreiergruppen zu lösen. Sie und Ihre Gruppenmitglieder arbeiten gemeinsam am Code oder teilen sich Unteraufgaben auf. Um sicherzustellen, dass jeder Programmierer den gesamten Code verstehen und erklären kann, verlangen wir von jedem Mitglied eine leicht unterschiedliche Abgabe. Wie unten beschrieben underscheiden sich die Versionen nur um eine spezielle Funktionalität für die jeder Teilnehmer selbst verantwortlich ist.

Vorbereitung

(a) Stellen Sie zuerst sicher, dass Ihr Raspberry Pi korrekt funktioniert und dass Sie in der Lage sind, Assembler-Code zu kompilieren und auszuführen. Eine entsprechende Anleitung befindet sich auf Ilias.

Stellen Sie einen Defekt beim Raspberry Pi oder den Zusatzkomponenten fest, melden Sie sich bitte umgehend bei den Assistenten.

(b) Lesen Sie das Tutorial zum Raspberry Pi und der ARM Assemblersprache durch. Testen Sie das Demo Programm `blink.s`, welches die LED auf dem Erweiterungsboard zum Blinken bringt. Bevor Sie mit den Aufgaben beginnen empfiehlt es sich, den Demo Code sorgfältig durchzulesen und zu verstehen.

(c) Laden Sie sich das Programmiergerüst zu dieser Aufgabenserie von ILIAS herunter. Die optionalen Aufgaben können Ihnen helfen, den vorhandenen Code zu verstehen.

Lauflicht

Ihre Aufgabe ist es, das gegebene Programmgerüst wie folgt zu vervollständigen:

(a) Tragen Sie Ihren Namen sowie den Namen einer allfälligen Übungspartnerin oder eines allfälligen Übungspartners an der vorgesehenen Stelle in der Datei `knightRider.s` ein.

(b) Implementieren Sie ein Lauflicht auf den acht LED Segmenten, das bei `LED0` beginnt, nach `LED7` wandert, dann wieder zurück nach `LED0` und so weiter. Im Anhang finden Sie zusätzliche Informationen, wie Sie die nötigen elektronischen Komponenten ansprechen können.

> **Student 1:** Sie geben die beschriebene Standardversion des Lauflichts ab.
>
> **Student 2:** Sie müssen Sie die Anzeige des Lauflichts negieren. Das heisst, alle Segmente bis auf eines sind eingeschaltet.

(c) Erweitern Sie Ihr Program, so dass das Drücken der Taste `BUTTON1` die Laufgeschwindigkeit des Lichts erhöht. Die Taste `BUTTON2` soll die Geschwindikeit reduzieren.

> **Student 3:** In Ihrer Version soll `BUTTON2` die Geschwindigkeit nicht reduzieren, sondern auf den Anfangswert zurücksetzen (ohne Neustart des Programms).

(d) **Optional:** Fügen Sie ein weiteres Lauflicht hinzu welches sich in die Gegenrichtung bewegt, so dass sich beide Lichter in der Mitte kreuzen.

Einreichung

(a) Stellen Sie sicher, dass Ihr Assemblerprogramm *ausführlich und sinnvoll* kommentiert ist. Als Richtwert gilt, dass jede Zeile kommentiert werden soll. Sie können zwei Zeilen zusammenfassen, falls dies Sinn macht. Dies ist eine notwendige Voraussetzung, damit der Programmierteil als erfüllt gilt.

(b) Stellen Sie ausserdem sicher, dass Ihre Implementation ohne Fehler und Warnungen kompilierbar ist. Dies ist eine notwendige Voraussetzung, damit der Programmierteil als erfüllt gilt.

(c) Erstellen Sie aus Ihrer Lösung eine Zip-Datei namens `<nachname>.zip`, wobei `<nachname>` durch Ihren Nachnamen zu ersetzen ist.

(d) Geben Sie die Zip-Datei elektronisch durch Hochladen in ILIAS ab.

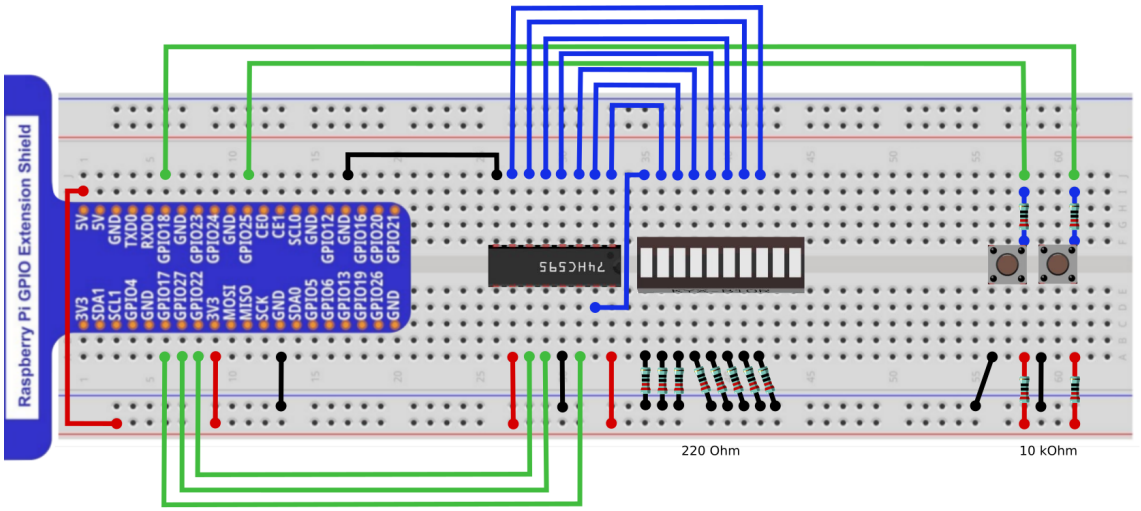

Abbildung 1: Bauplan für die LED Segmentanzeige mit Seriell- zu Parallel Konverter (74HC595) und zwei Tasten (rechts). Rot: Stromverbindung zu positiver Elektrode (+). Schwarz: Stromverbindung zu negativer Elektrode (−). Grün: Datenleitungen für Eingangs- und Ausgangssignale. Blau: Sonstige Datenleitungen.

A Zugriff auf LED Anzeige

In der Einführungsdemo zum Raspberry Pi haben Sie bereits gelernt, wie Sie auf einzelne Pins des GPIO zugreifen können. Im Prinzip ist es möglich, die LEDs der in Abbildung 1 gezeigten LED Schiene einzeln anzusprechen, das heisst, jede LED wird über einen Pin am GPIO gesteuert. Das macht es aber etwas umständlich, die in dieser Serie verlangte Animation zu implementieren. Deshalb verwenden wir einen Umwandler, der ein serielles Signal liest, zwischenspeichert und parallel ausgibt.

Nehmen wir die Sequenz 00100000 als Beispiel. Wir möchten gerne, dass diese Serie von Einsen und Nullen an den Umwandler-Chip (74HC595 genannt) gesendet wird, so dass dieser das Signal auf die einzelnen Pins der LED Anzeige aufspalten kann. Für diese Beispielsequenz soll also die dritte LED (von links) aufleuchten. Dazu werden drei Pins benötigt (anstelle acht):

- **LATCH PIN** Der Zustand dieses Pins gibt an, ob ein serielles Signal gelesen wird (LOW) oder nicht (HIGH).
- **DATA PIN** Über diesen Pin wird die Sequenz Bit für Bit gesendet.
- **CLOCK PIN** Gibt den Takt bei der Übertragung einer Sequenz an, indem das Ausganssignal abwechslungsweise zwischen HIGH und LOW bzw. 0 und 1 umgeschaltet wird.

Diese drei Pins entsprechen den drei unteren grünen Leitungen in Abbildung 1. Um Ihnen die Datenübertragung zu vereinfachen, stellen wir Ihnen die Subroutine

```
void shiftOut(datapin, clockpin, firstbit, data)
```

zur Verfügung. Die beiden ersten Argumente sind die Nummern für Takt- und Datenpin. Der Parameter `data` ist ganz einfach eine Zahl (die Daten). In unserem Beispiel ist das $00100000_{(2)} = 2^5 = 32$. Der Parameter `firstbit` gibt lediglich die Bitwertigkeit an (0: Bit mit tiefstem Stellenwert wird zuerst gelesen, 1: höchstwertigstes Bit wird zuerstgelesen). Wichtig: Bevor `shiftOut` verwendet werden kann, muss der Latch-Pin auf LOW gesetzt werden um eine Datenübertragung zu signalisieren, und nach verwenden von `shiftOut` muss dieser wieder auf HIGH zurückgestellt werden. In unserem Beispiel ist der Ablauf also folgender:

```
digitalWrite(27, 0)
shiftOut(17, 22, 0, 32)
digitalWrite(27, 1)
```

Teil Ihrer Aufgabe ist es also, diese drei Subroutinenaufrufe in Assemblercode umzusetzen. Tipp: Die nötigen Pinnummern sind im Programmgerüst bereits als Konstanten definiert.

B Zugriff auf Drucktasten

Die in Abbildung 1 gezeigten Tasten sind an die Pins 18 (`BUTTON1`) und 25 (`BUTTON2`) angeschlossen. Um das Eingangsignal an diesen Pins zu lesen wird die vordefinierte Subroutine `digitalRead(pin)` verwendet. Der Rückgabewert dieser Funktion ist 1 (Taste oben) oder 0 (Taste unten). Im Falle der Aufgabenstellung dieser Serie möchten wir nur dann eine Aktion durchführen, wenn das Signal von 1 auf 0 wechselt (fallende Signalflanke), denn das bedeutet, dass die Taste nach unten gedrückt wird.

Zur Detektion dieses Signalwechsels stellen wir Ihnen die Subroutine `waitForButton` zur Verfügung, mit der Sie auf eine fallende Signalflanke warten können.

```
[pressed, state] waitForButton(pin, timeout, state)
```

Das Argument `timeout` ist die Wartezeit in Millisekunden, und `state` ist der (vorherige) Zustand des Input-Pins (0 oder 1). Wird innerhalb des definierten Zeitfensters eine fallende Flanke detektiert, dann wird `pressed` auf 1 gesetzt, und sonst auf 0.

Im Aufgabenteil (c) müssen Sie nun diese Subroutine mit den korrekten Argumenten aufrufen und basierend auf dem Rückgabewert die entsprechende Aktion ausführen. Tipp: In Assembler werden die beiden Rückgabewerte in den Registern `R0` und `R1` abgelegt.