

Cedric Aebi 17-103-235

Nicolas Müller 17-122-094

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	
$\sim 0xFE$	
$\neg 0xFE$	

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 unterscheiden 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 Geschwindigkeit 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.

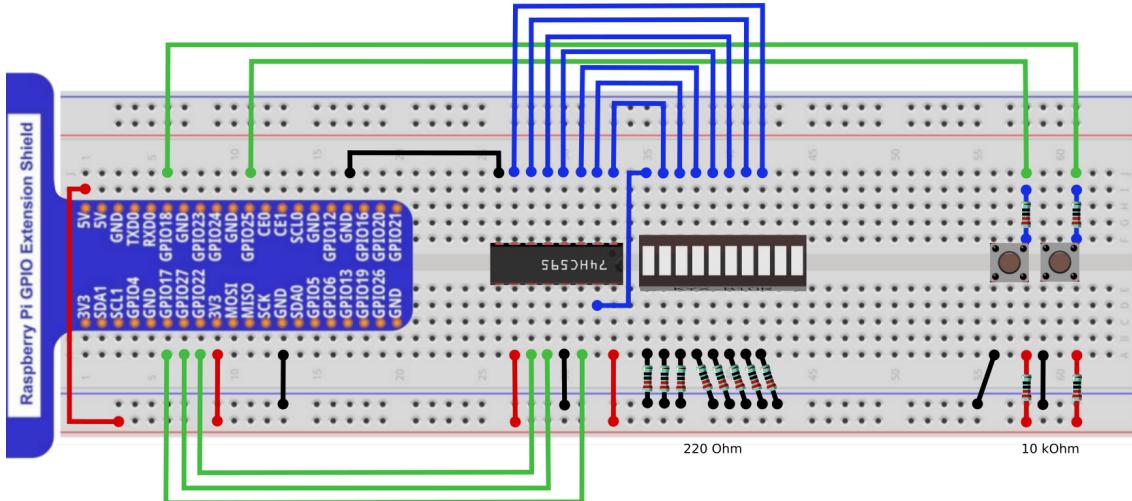


Abbildung 1: Bauplan für die LED Segmentanzeige mit Seriell- zu Parallel Konverter (74HC595) und zwei Tasten (rechts). Rot: Stromverbindung zu positiver Elektrode (+). Schwarz: Stromverbindung zu negativer Elektrode (-). Grün: Datenleitungen für Eingangs- und Ausgangssignale. Blau: Sonstige Datenleitungen.

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 Beispieldsequenz 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 Ausgangssignal abwechselungsweise 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 zuerst gelesen). 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 Eingangssignal 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.

1 Sign Extension (1 Punkt)

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

Weil der PC nach dem Befehl immer 4 bytes (32 bits) weitergeht. Somit muss der Befehl immer 32 bits lang sein. Branch, Load und Store Funktionen müssen somit mit bits nachgeführt werden und benötigen somit Sign-Extend.

Zudem rechnen ALUs nur mit 32 bits.

2 Logische und bitweise Operationen (1 Punkt)

Bestimmen Sie die jeweilige Ausgabe:

	Eingabe	Ausgabe
1	0xA3 & 0x3A	0x22
2	0x0 && 0xEF	0x0
3	0xA3 0x3A	0xB3
4	0x0 0xEF	0xEF
5	~0xFE	0x1
6	!0xFE	0x0



$$\begin{array}{r} 10100011 \\ \& \underline{00111010} \\ \hline 00100010 = 22 \end{array}$$

2) && is logical. It returns 0x0 because 0x0 is false

$$\begin{array}{r} 10100011 \\ | \underline{00111010} \\ \hline 10111011 = 0xB3 \end{array}$$

$$4) \sim 0xFE \\ 11111110 = 00000001$$

5) 0xFE is true. So 0x0

3 Hardwarezugriff (1 Punkt)

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

Man könnte die Adresse der Hardware im Memory abspeichern und über diese die entsprechende Hardware zugreifen. Die Adressen können zum Beispiel in einer Hashtabelle abgespeichert werden.



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.

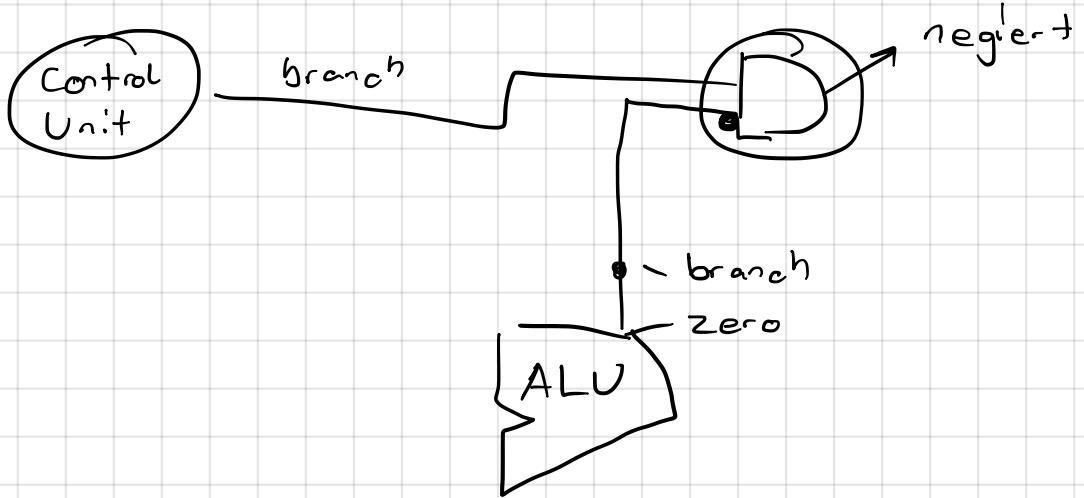
- (a) Für die Singlecycle-Implementation
- (b) Für die Multicycle-Implementation

a) Die beq-Operation hat die 4 Schritte:

1. fetch PC+4
2. T1 and T2 read from Register (BEQ +1, +2, L)
3. ALU performs SUB / PC+4 added to sign-extended offset
4. Zero output from ALU used to set PC.

Man könnte den Output vom ALU zur Branch-Control-Unit negieren. Somit wäre beq = bne und umgekehrt.





b) Auch hier muss man bei Step 3: branch anstelle von $\text{if}(A == B) \quad PC \leftarrow \text{AluOut};$ den Ausgang negieren. Das heißt $\text{if } !(A == B) \quad PC \leftarrow \text{AluOut},$

5 Pipeline Registers (1 Punkt)

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

Sie werden dazu gebraucht, um die einzelnen Parts zu isolieren in der Pipeline. Nur wenn alle Register bereit sind, kann der nächste Schritt ausgeführt werden.



6 Pipelining Hazard (2 Punkte)

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

Structural Hazard: Wenn der Prozessor nicht die benötigte Hardware besitzt, um alle Befehle auszuführen. Vor allem bei Register wenn gleichzeitig gelesen und geschrieben werden soll.

Data Hazard: Entsteht vor allem im Register

Wenn ein Zyklus auf Daten des vorangehenden baut, kann es zu Hazards kommen, wenn der vorangehende noch nicht fertig ist.

Control Hazard:

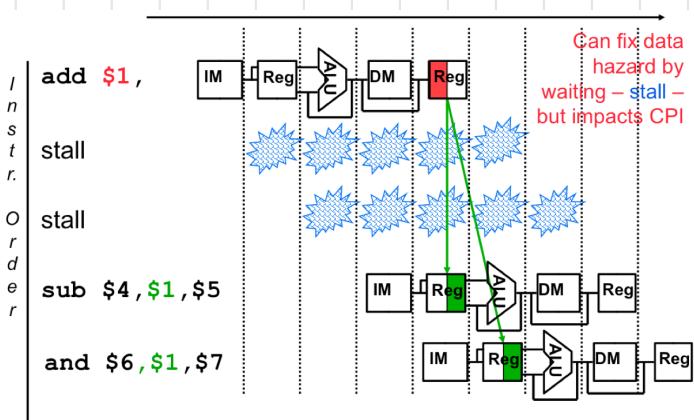
Diese kommen von Operationen, die den PC ändern. Eine solche verändert den PC und wenn nun eine andere Operation diesen braucht, liegt er falsch. Bei einer Branch-Operation muss gewartet werden, bis der Zyklus fertig ist, erst dann wird weitergeföhrt.

- **structural hazards**: attempt to use the same resource by two different instructions at the same time
- **data hazards**: attempt to use data before it is ready
 - An instruction source operand(s) is produced by a prior instruction still in the pipeline
- **control hazards**: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
 - branch instructions



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")

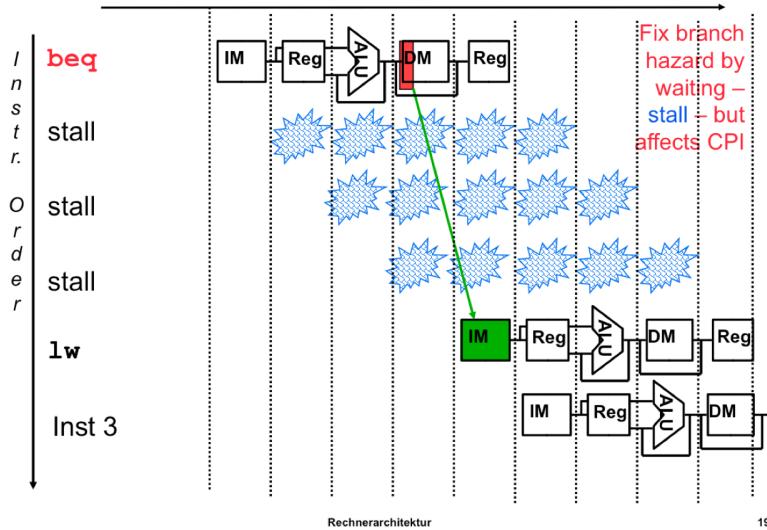


Hier betrifft der Data

Hazard das Register-Memory.

Hier kann gleichzeitig gelesen (2te Hälfte) und geschrieben (1te Hälfte) werden. So mit genügen nur 2 stalls,

weil man Reg gleichzeitig nutzen kann.



Hier wird nun die "Hardware genutzt". Sobald Loads, oder auch Branch Operation betroffen werden, muss abgewartet werden, bis DM fertig gerechnet hat.

LW benötigt ihn und BEQ verändert ihn. Somit kann LW erst benutzt werden, wenn BEQ fertig ist.
→ Deshalb 3 Schritte.

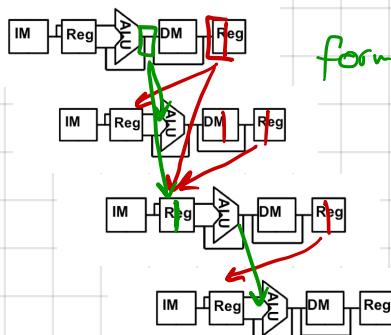


8 Data Hazard (2 Punkte)

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)

add \$t0, \$t5, \$t4



lw \$s2, 0(\$t0)

sub \$s3, \$t0 ,\$s2

sw \$t4, 4(\$s3)

