Rechnerorganisation

Sommersemester 2023

Prof. Stefan Roth, Ph.D.

TECHNISCHE UNIVERSITÄT DARMSTADT

11. Aufgabenblatt mit Lösungsvorschlag

10.7.2023
Mikroarchitekturen, Pipeline-Prozessor, Hazards, Dateien

Aufgabe 1: Theoriefragen

a) In welche Phasen wird die Ausführung einer Instruktion bei dem vorgestellten Pipelined-Prozessor unterteilt?

Lösungsvorschlag:

Fetch-, Decode-, Execute-, Memory- und Writeback-Phase.

b) Was versteht man unter einem Hazard?

Lösungsvorschlag:

Hazards sind Abhängigkeiten zwischen Pipeline-Stufen. Man unterscheidet zwischen Data- und Control-Hazards. Data-Hazards treten auf, wenn Daten gelesen werden, die jedoch von einer der vorherigen Instruktionen noch nicht vollständig bearbeitet worden sind. Da erst relativ spät in der Pipeline entschieden wird, ob ein Sprung vorliegt oder nicht, können die folgenden ausgeführten Instruktionen unter Umständen nicht die gewünschten sein. Dies wird als Control Hazard bezeichnet.

c) Bei der Entwicklung von modernen Prozessoren werden viele Ressourcen in die Verbesserung der Sprungvorhersage gesteckt. Warum wirkt sich ein falsch vorhergesagter Sprung negativ auf die Performance aus?

Lösungsvorschlag:

Falls ein Sprung falsch vorhergesagt worden ist, müssen die falschen Instruktionen verworfen werden. Wäre der Sprung korrekt vorhergesagt worden, hätten die Takte für die Berechnung der falschen Instruktionen genutzt werden können.

d) Bei Prozessoren unterscheidet man zwischen in-order oder out-of-order Ausführung von Instruktionen. Welcher Ansatz ist schneller? Können Sie sich auch Einsatzszenarien für den anderen Ansatz vorstellen?

Lösungsvorschlag:

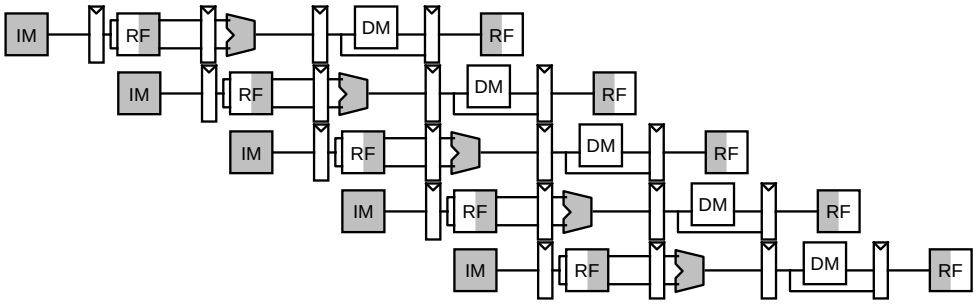
Die out-of-order Ausführung ist schneller, da durch die Umsortierung der Befehle Stalls vermieden werden können und die Pipeline besser ausgelastet werden kann. Die Einheiten zur Selektion der nächsten Instruktion belegen jedoch Siliziumfläche. Dies treibt die Herstellungskosten und die Leistungsaufnahme in die Höhe. Deswegen wird bei manchen günstigen oder sparsamen Prozessoren auf in-order Ausführung gesetzt.

Aufgabe 2: Auflösen von Hazards

Zeichnen Sie die Hazards der angegebenen ARM-Programme in das Pipeline-Diagramm ein. Modifizieren Sie anschließend das Programm, sodass es korrekt ausgeführt wird. Sie dürfen nur *nop*s einfügen oder die Zeilenreihenfolge ändern, nicht jedoch die Instruktionen verändern. Versuchen Sie die minimale Lösung zu finden. Gehen Sie davon aus, dass der eingesetzte Prozessor keine Hazard Unit besitzt. Register `r11` ist initial mit 0 belegt.

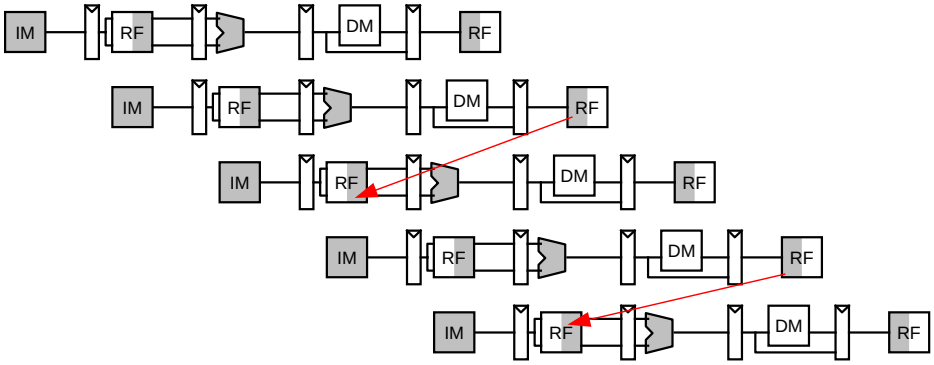
a) Programm 1

```
add r2, r11, #3
add r0, r11, #5
lsl r0, r0, #2
ldr r1, [r11, #40]
and r1, r1, #1
```



Lösungsvorschlag:

Abhängigkeiten im Diagramm:

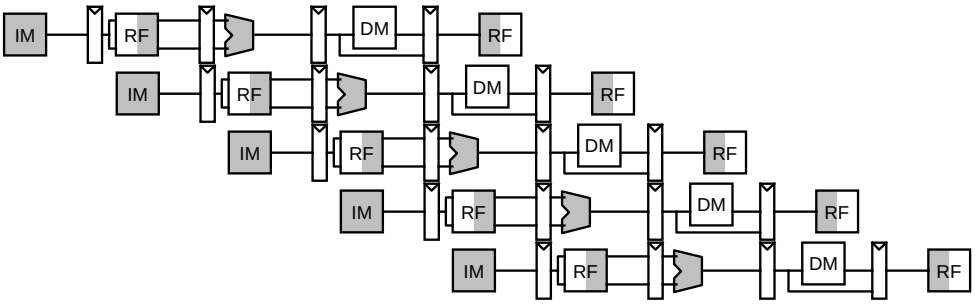


Eine mögliche Modifikation wäre:

```
add r0, r11,  5
ldr r1, [r11, 40]
add r2, r11, 3
lsl r0, r0, 2
and r1, r1, 1
```

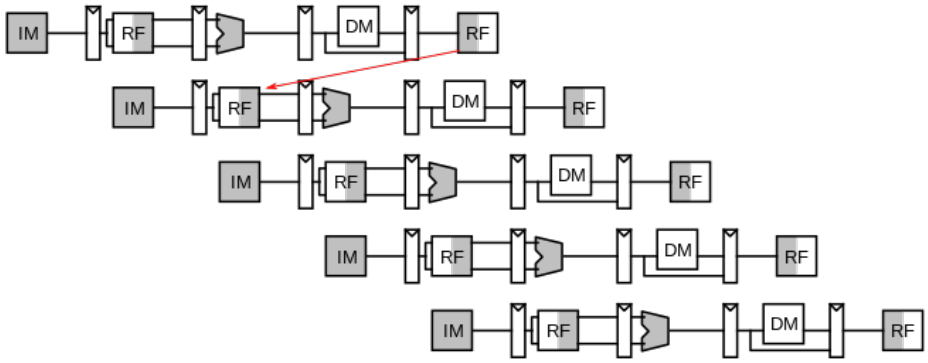
b) Programm 2

```
sub r3, r2, r1
add r1, r3, r2
ldr r0, [r5, #4]
orr r0, r3, r4
add r3, r1, #5
```



Lösungsvorschlag:

Abhängigkeiten im Diagramm:



Eine mögliche Modifikation wäre:

```
sub r3, r2, r1
ldr r0, [r5, 4]
nop
add r1, r3, r2
orr r0, r3, r4
nop
add r3, r1, 5
```

Aufgabe 3: Dateien

Diese Aufgabe wurde von Florian Piana im Rahmen eines Praktikums in der Lehre entwickelt.

Dateien sind essentielle Bestandteile der Nutzer-Maschinen-Interaktion und ein wesentliches Konzept in der Datenmanipulation des alltäglichen Gebrauchs. Diese Aufgabe soll Ihnen die Verwendung von Dateien mit Hilfe des ARM-Befehlssatzes nahebringen und verständlich machen.

Die Grundlage für einen Dateizugriff bietet ein sogenannter „File-Descriptor “ (Oftmals abgekürzt als „FD“). Jedes Mal, wenn Sie einen Dateizugriff anfordern, wird ein solcher File-Descriptor vom Betriebssystem festgelegt und dem Programmierer (in diesem Falle Sie) übergeben. Der File-Descriptor dient nun als eine Art Identifikationsnummer, mit der Sie verschiedene Funktionalitäten an der Datei aufrufen können.

In Niedersprachen sind Software-Interrupts besonders dafür geeignet. Wir werden uns zunächst mit Software-Interrupts auseinandersetzen, um diese später mit bereitgestellten Funktionen zu vergleichen.

Die

```
SWI{cond} <immediate_24_bit>
```

Instruktion wird genutzt, um einen Software-Interrupt auszuführen. Dies sorgt für eine Statusänderung der Prozessor-Einheit in einen besonderen Modus (die Details werden hier nicht weiter behandelt), sodass das Betriebssystem die Kontrolle übernehmen und eine durch den Immediate-Wert bestimmte Aktion ausführen kann. Jedoch wird aus Performancegründen inzwischen immer ein Interrupt mit Immediate-Wert 0 (Null) ausgeführt und die Aktion als Zahl im Register 7 (R7) abgelegt.

Man nennt solch eine Interaktion mit dem Betriebssystem manchmal auch „syscall“.

Die Werte, die in R7 abgelegt werden müssen, sind systemspezifisch. Wir gehen hier davon aus, dass Sie Dateizugriff auf ein ARM-Linux mit EABI-Standard (also zum Beispiel dem Client-SSH) nutzen wollen. Hier eine Zusammenfassung einiger Funktionalitäten, die durch Software-Interrupts genutzt werden können:

R7-Wert	Beschreibung	R0-Wert	R1-Wert	R2-Wert	Rückgabe
3	Lesen	File-Descriptor	Zeiger auf Puffer, in dem die gelesenen Bytes gespeichert werden sollen	Anzahl der zu lesenden Bytes	Anzahl der tatsächlich gelesenen Bytes
4	Schreiben	File-Descriptor	Zeiger auf Puffer, in dem die zu schreibenden Bytes liegen	Anzahl der zu schreibenden Bytes	Anzahl der tatsächlich geschriebenen Bytes
5	Öffnen	Zeiger auf Datei/Pfad (als String)	Flags (gibt an, welche Zugriffsrechte angefordert werden sollen)	Modus (dies kann für unsere Zwecke zunächst 0666_8 bleiben)	Negativ (-1), falls Datei nicht geöffnet werden konnte, sonst File-Descriptor
6	Schließen	File-Descriptor	-	-	Bei Fehler -1, sonst 0
19	Lese-/Schreib-Index verschieben	File-Descriptor	Relative Verschiebung	Bezugspunkt	Der geänderte Lese-/Schreib-Index

a) Die Konsole als „Datei“

Viele Betriebssysteme – einschließlich Linux-Systeme – nutzen das Prinzip der File-Descriptors sogar um die Konsole zu repräsentieren. Dadurch kann Text auf der Konsole auch ausgegeben werden, indem man sie als (bereits geöffnete) „Datei“ behandelt. Diese Konsolen-Schnittstelle nennt man häufig „stdout“ (gesprochen: Standard-Out). STDOUT hat standardmäßig den File-Descriptor 1.

Hello World

Das Vorgehen ist beim Schreiben in die Konsole dann wie folgt:

1. Vorbereiten der Daten und Argumente
   - In R0 den Wert 1 ablegen (da 1 der File-Descriptor für die Ausgabe auf der Konsole ist)
   - In R1 einen Zeiger auf Zeichen, die in die Ausgabe geschrieben werden sollen, ablegen.
   - In R2 die Anzahl der Zeichen, die in STDOUT geschrieben werden sollen.
   - In R7 den Wert 4 ablegen (da wir in die Konsole schreiben möchten)
2. SWI 0 aufrufen

Aufgabe: Schreiben Sie mit Hilfe des ARM-Instruktionssatzes ein Programm, in dem der String „Hello World“ ausgegeben wird.

Lösungsvorschlag:

```
.data
// We do not need the extra NULL-Byte at the end, because we use the character-count directly
output: .ascii "Hello␣World"

.text
.global main
main:
  push {lr}       // syscalls do not guarantee the integrity of lr
  mov r0, #1      // STDOUT has the file-descriptor 1
  ldr r1, =output // Pointer to the string
  mov r2, #11     // Character-count of the string, we want to print
  mov r7, #4      // We want to write, which has the syscall-number 4
```

```
  swi 0           // Let the Operating System do the printing
  pop {lr}
  bx lr
```

Man beachte: Im Gegensatz zur Verwendung von bl printf muss hier nicht unbedingt .asciz benutzt werden, da wir die länge des Strings manuell angeben. ABER dafür müssen wir die Anzahl der auszugebenden Zeichen schon zur Compile-Zeit kennen. (Dies lässt sich durch cleveres Programmieren jedoch lösen.)

Eingabe

Aufgabe: Versuchen Sie nun, eine Eingabe des Nutzers in einem Puffer zu speichern, indem Sie die Konsole als „Datei“ betrachten. Geben Sie die Nutzer*inneneingabe anschließend aus und hängen Sie an das Ende der Nutzer-Eingabe ein „!“ an. Was fällt ihnen auf, wenn der Nutzer mehr Zeichen eingibt, als Sie in den Argumenten des „Lesen “ -Syscall angegeben haben? Hinweis: Der ASCII-Code für „!“ ist 33_{10} (als dezimal) bzw. 21_{16} (als hexadezimal).

Lösungsvorschlag:

```
.data
// We allocate 50 Bytes for user-input, 1 Byte for appending the line break
buffer: .skip 51

.text
.global main
main:
  push {lr}
  mov r0, #1      // 1 is the file-descriptor for STDOUT
  ldr r1, =buffer // Our buffer is the destination
  mov r2, #50     // We read 50 bytes of user-input into the buffer
  mov r7, #3      // Read-Syscall has the number 3
  push {r0, r1}   // Push register values, so we do not have to repeat
                  // the instructions above
  swi 0           // Instantiate the syscall

  pop {r0, r1}    // Retrive saved register values
  mov r3, #'!'    // We can write #'!' toget the ascii-value directly
  str r3, [r1, #50] // Append !  to the end

  mov r2, #51     // Include the appended character in printing
  mov r7, #4      // Syscall-number for writing is 4
  swi 0

  pop {lr}
  bx lr
```

Wenn der Nutzer mehr Zeichen eingibt, als durch die Lese-Anzahl angegeben wurde, werden alle Zeichen, die über diese Anzahl hinausragen, nicht akzeptiert und erst bei der nächsten Eingabe übernommen.