Labor Computerarchitektur

Einführung in Piplining und superskalare Architektur mit dem DLX Simulator am Beispiel eines Multiplikators

Michael Sved 5001977
Linus Andrae 5001800

Juni 2018

Inhaltsverzeichnis

1. Einführung

In diesem Versuch sollen die Themen Pipelining und superskalare Architektur an einem praktischen Beispiel mit dem DLX Simulator bearbeitet werden. Piplining ist eine Möglichkeit parallelität in einen Prozessor einzubringen, in dem der Datenweg mit Latches in kleinere Einheiten geteilt wird (siehe [1, S. 314]). Dies führt dazu, dass man Befehle starten kann, während die vorherigen Befehle noch nicht vollständig abgearbeitet sind, sondern sich in den weiteren Bearbeitungseinheiten befinden. Bei gleichem Takt führt dies zu einem höheren Durchsatz an Befehlen.
Der DLX Simulator bietet eine 5-Stufige Pipeline (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Arbeitsspeicher I/O, WB = Write Back).
Bei Superskalarer Architektur werden teile der Pipeline in Hardware echt paralell angelegt. Da das Starten und Dekodieren im Vergleich zum berechnen schnell geht, kann Beispielsweise die EX Stufe (EX im Simulator) in parallel arbeitende Einheiten zerlegt werden, wobei z.B. eine auf Fließkommaoperationen spezialisierte Einheiten eingefügt werden kann. Die ersten Stufen der Pipeline dekodieren einen Befehl, rufen die Operanden ab und geben dann die Berechnung an eine der mehreren ALUs weiter.

1.1. Thema

Dieser Versuch soll mittels eines Multiplikationsprogramms das Prinzip der Geschwindigkeitsoptimierung von Prozessoren mittels Pipelining am Beispiel des DLX-Simulators praktisch zeigen. Des Weiteren soll superskalare Architektur Theoretisch an diesem Beispiel behandelt werden.

1.2. Hilfsmittel

- DLX-Simulator v1.0-6
- Beispiel example_6_input.s

1.3. Lernziele und Kompetenzen

Bei diesem Versuch sollen sich die Teilnehmenden des Labors mit dem DLX-Simulator auseinandersetzen und an einer praktischen Aufgabe das Prinzip des Pipelinings analysieren.
Dabei soll ein einfaches Multiplikationsprogramm für die MIPS Architektur, welches der Simulator nutzt, geschrieben und dessen Ausführung im Simulator analysiert werden.
Kernkompetenzen die bei diesem Versuch benötigt werden sind:

- Wissen aus der COMPARCH Vorlesungen über den Datenweg, Pipelining und superskalare Architektur anwenden
- Analysieren des Pipelining-Prinzipien mittels eines Simulators
- Anwendung von Assemblerartiger Programmierung des DLX Simulators

- Verstehen des Programmablauf auf Prozessorebene (Datenpfad der Mikroarchitektur)

2. Vorbereitung

Die Laborteilnehmenden sollten sich zunächst in die Dokumentation des DLX-Simulators (EinführungDLX.pdf) einlesen, um Programme für diesen Simulator zu schreiben. Im Tannenbaum und im Vorlesungsskript sind die Kapitel zum Thema Pipelining ebenfalls zu wiederholen. Weiterhin ist es Hilfreich das Dokument „Befehlssatz.pdf“ (ausgedruckt) mitzubringen, um das benötigte Programm für den Simulator zu schreiben. Das Beispiel 6 soll als Grundlage des zu schreibenden Programms dienen, da dieses die wichtigsten Bausteine die zum Ausführen der benötigten Multiplikation sowie Ein- und Ausgabe bereits implementiert, es muss einzig die Anzahl der Multiplikationen erhöht werden. Das Labor eignet sich, sobald das Thema Pipelining und superskalare Architektur in der Vorlesung behandelt wurde.
(Zeitaufwand für die Vorbereitung: 90 Minuten)

3. Aufgaben und Lösungen

In diesem Versuch soll zunächst ein Programm mit dem DLX-Simulator erstellt werden bei dem die Grundlegende Funktion des Simulators und seinem Befehlssatz klar werden soll.
Auf Grundlage dieses Programms soll im Anschluss die Funktionsweise der Pipeline veranschaulicht werden und Vor- und Nachteile ausgearbeitet werden. Am Ende soll in der Theorie die Vorteile und evtl. Probleme von superskalarer Architektur ergründet werden.
Im folgenden werden die Einzelnen werden die einzelnen Aufgaben dargestellt.
Die Aufwandschätzung und Bewertungsvorschlag sind in Klammern angegeben.
Die Gesamtarbeitszeit ist auf 210 Minuten Ausgelegt (exkl. Vorbereitung).
In Kursiv ist die Musterlösung beschrieben
`In Festbreitenschrift ist die erwartete Lösungsdokumentation beschrieben`
Die Lösung soll zum Ende der Laborzeit abgegeben und bewertet werden.

1. Die Laborteilnehmenden sollen zu allererst ein Programm für den DLX-Simulator erstellen. Das Programm soll zunächst 4 Integer-Zahlen von der Standardeingabe lesen, danach sollen diese möglichst performant miteinander multipliziert werden, hierbei sollten die Laborteilnehmenden die Multiplikationen so anordnen, dass zunächst zwei parallel ausführbare Multiplikationen durchgeführt werden und danach die Ergebnisse der Teiloperationen miteinander Multipliziert wird um auf das Endergebnis zu kommen.
   Diese Einsicht ist nicht entscheidend, zeigt jedoch, dass das Konzept hinter superskalarer Architektur verstanden wurde.
   Um nicht zu viel Zeit mit dem Programmieren zu verbringen, sollte sich am Beispiel example_6_input.s aus den DLX-Beispielen orientiert werden.

(Zeitaufwand: 80 Minuten; Bewertung: 40 %)
Antwort: Zum Lösen dieser Aufgabe müssen die bereits im Beispiel enthaltenden Einleseoperationen und die Multiplikation erneut im Programm dupliziert, sowie neue Variablen für die Integerwerte hinzugefügt werden. Es müssen nicht mehr als 20 Zeilen Code hinzugefügt werden. Das Programm muss danach Eingaben entgegennehmen und das Ergebnis einer Multiplikation korrekt anzeigen.
`Der Lauffähige Code ist die erwartete Lösung, wenn die Studierenden mögen, kann Ihnen ab hier auch der Quelltext der Musterlösung (s. Anhang A) gegeben werden.`

2. Zunächst soll nun der das Programm auf dem DLX Simulator ausgeführt werden, dabei soll das Pipelining des Simulators betrachtet und verstanden werden.
   Die Teilnehmenden sollten verstehen, was die einzelnen Stufen tun und warum z.B. Leerstellen also Blockaden entstehen. Dabei sollen Eingabewerte kleiner Zehn benutzt werden, denn dort muss nur ein Zeichen eingelesen werden, dies benötigt weniger Speicher und ist somit schneller. Dies soll in der nächsten Aufgabe herausgearbeitet werden.
   (Zeitaufwand: 20 Minuten; Bewertung: 10 %)
   Antwort: Bei Zahlen < 10 werden im Beispielprogramm 192 Zyklen benötigt, wichtig ist, dass die Anzahl an Zyklen im nächsten Versuch größer ist.
   `Die Fragen sollen in kurzen Sätzen beantwortet werden.`

3. Nun soll mit Zahlen > 10 benutzt werden und der Unterschied in der Ausführungszeit soll mittels des Simulators ergründet werden. Durch das Einlesen von zweistelligen Zahlen benötigt das Programm länger.
   (Zeitaufwand: 20 Minuten; Bewertung 10 %)
   Antwort: Das Einlesen einer Zahlt geschieht auf Zeichenebene, das heißt, sobald mehr als ein Zeichen eingelesen wird, muss die Einleseroutine nochmal durchlaufen werden. Somit erhöht sich die Anzahl der Zyklen im Programmablauf. Dies soll von den Laborteilnehmenden ausgearbeitet und beschrieben werden.
   `Die Fragen sollen in kurzen Sätzen beantwortet werden. Es soll mit der vorherigen Aufgabe verglichen werden.`

4. Nun sollen die Multiplikationen genauer untersucht werden.
   Es soll herausgearbeitet werden, dass das Einführen einer Superskalaren-Architektur mit zwei ALUs sinnvoll ist, da dies die beiden ersten Teiloperationen echt parallel abarbeiten kann.
   (Zeitaufwand: 30 Minuten; Bewertung 15 %)
   Antwort: Eine superskalare Architektur würde die Operationen parallelisieren, dies spricht für die Einführung einer solchen.
   `Die Fragen sollen in kurzen Sätzen beantwortet werden.`

5. In der letzten Aufgabe soll das Verständnis der verschiedenen Einheiten und ihrer Funktion vertieft werden. Durch die Trennung von ALU für Integer und einer auf Floatingpoint Zahlen spezialisierte Einheit, beginnt die Reihenfolge noch wichti-

ger zu sein, da das Ergebnis einer Integer Multiplikation weiterhin ein Integer ist, sobald einmal mit einer Floatingpoint Zahl multipliziert wird, wird die Floatingpoint ALU benötigt. Hierbei wird davon ausgegangen, das die ALU nur Integer, die Floatingpoint unit beides berechnen kann.
(Zeitaufwand: 60 Minuten; Bewertung 25%)
Antwort: Es sollte herausgearbeitet werden, dass die Reihenfolge wichtig ist und das zuerst die Operanden mit dem gleichen Datentypen und danach die Teilergebnisse miteinander multipliziert werden sollten, da somit die beiden Einheiten Ausgelastet werden.
Wird dies nicht beachtet kann es zu Wartezyklen kommen, wenn ausschließlich die Floatingpoint ALU benutzt wird, da zunächst je ein Float mit einem Integer multipliziert wird.
`Dies soll in einigen Sätzen dokumentiert werden.`

A. Anhang

```
; ------------------------
;Loesung fuer Aufgabe 1:
; ------------------------

Prompt:          .asciiz          "Please enter an integer >1 : \n"
PrintfFormat:    .asciiz          "Result: %d.\n"
                                  .align          2
PrintParameter: .word             PrintfFormat
PrintfValue:     .space           4
Int:     .space          4
Int2: .space    4
Int3: .space    4
Int4: .space    4
                        .text
                        .global main
main:
                        ; Set prompt for InputUnsigned function
                        addi    r1,r0,Prompt
                        jal             InputUnsigned ; call of InputUnsigned
                        nop ; required for branch delay slots
                        nop ; required for branch delay slots

      sw Int,r1           ; mov first number to $Int

      addi      r1,r0,Prompt
                        jal             InputUnsigned ; call of InputUnsigned
                        nop ; required for branch delay slots
                        nop ; required for branch delay slots
      sw Int2, r1       ; mov second number to $Int2

      addi r1, r0, Prompt
      jal InputUnsigned
      nop ; required for branch delay slots
      nop ; required for branch delay slots

      sw Int3, r1 ; mov third number to $Int3

      addi r1, r0, Prompt
      jal InputUnsigned
      nop
      nop
```

```
        sw Int4, r1 ; mov forth number to $Int4

        ;Calculate the result
        ;multiply both of the integers
        lw r1, Int
        lw r2, Int2
        mult r3, r1,r2

        lw r1, Int3
        lw r2, Int4
        mult r4, r1, r2

        mult r1, r3,r4
                        ; Output of the calculated result
                        sw        PrintfValue, r1
                        addi    r14, r0, PrintParameter
                        trap    5
                        ; End of program
                        trap    0


;-------------------------------------------------------------------------------
;Subprogram call by symbol "InputUnsigned"
;expect the address of a zero-terminated prompt string in R1
;returns the read value in R1
;changes the contents of registers R1,R13,R14
;-------------------------------------------------------------------------------

                .data
        ; Data for Read-Trap
ReadBuffer:     .space          80
ReadPar:          .word           0,ReadBuffer,80

                ; Data for Printf-Trap
PrintfPar:      .space          4

SaveR2:         .space          4
SaveR3:         .space          4
SaveR4:         .space          4
SaveR5:         .space          4

                .text
                .global         InputUnsigned
```

```
InputUnsigned:
                ; save register contents
                sw              SaveR2,r2
                sw              SaveR3,r3
                sw              SaveR4,r4
                sw              SaveR5,r5

                ; Prompt
                sw              PrintfPar,r1
                addi            r14,r0,PrintfPar
                trap            5

                ; call Trap-3 to read line
                addi            r14,r0,ReadPar
                trap            3

                ; determine value
                addi            r2,r0,ReadBuffer
                addi            r1,r0,0
                addi            r4,r0,10        ;Decimal system

Loop:           ; reads digits to end of line
                lbu             r3,0(r2)
    nop ; required because of data dependency on R3
                seqi            r5,r3,10        ;LF -> Exit
                bnez            r5,Finish
                nop ; required for branch delay slots
                nop ; required for branch delay slots
                subi            r3,r3,48        ; '0'
                multu           r1,r1,r4        ;Shift decimal
                add             r1,r1,r3
                addi            r2,r2,1         ;increment pointer
                j               Loop
                nop ; required for branch delay slots
                nop ; required for branch delay slots

Finish:         ; restore old register contents
                lw              r2,SaveR2
                lw              r3,SaveR3
                lw              r4,SaveR4
                lw              r5,SaveR5
                jr              r31             ; Return
                nop ; required for branch delay slots
                nop ; required for branch delay slots
```

Laboraufgabe: Einführung in Piplining und superskalare Architektur mit dem DLX Simulator am Beispiel eines Multiplikators

Einführung

In diesem Versuch sollen die Themen Pipelining und superskalare Architektur an einem praktischen Beispiel mit dem DLX Simulator bearbeitet werden. Piplining ist eine Möglichkeit parallelität in einen Prozessor einzubringen, in dem der Datenweg mit Latches in kleinere Einheiten geteilt wird (siehe [1, S. 314]). Dies führt dazu, dass man Befehle starten kann, während die vorherigen Befehle noch nicht vollständig abgearbeitet sind, sondern sich in den weiteren Bearbeitungseinheiten befinden. Bei gleichem Takt führt dies zu einem höheren Durchsatz an Befehlen.
Der DLX Simulator bietet eine 5-Stufige Pipeline (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Arbeitsspeicher I/O, WB = Write Back).
Bei Superskalarer Architektur werden teile der Pipeline in Hardware echt paralell angelegt. Da das Starten und Dekodieren im Vergleich zum berechnen schnell geht, kann Beispielsweise die EX Stufe (EX im Simulator) in parallel arbeitende Einheiten zerlegt werden, wobei z.B. eine auf Fließkommaoperationen spezialisierte Einheiten eingefügt werden kann. Die ersten Stufen der Pipeline dekodieren einen Befehl, rufen die Operanden ab und geben dann die Berechnung an eine der mehreren ALUs weiter.

Thema

Dieser Versuch soll mittels eines Multiplikationsprogramms das Prinzip der Geschwindigkeitsoptimierung von Prozessoren mittels Pipelining und Superskalarer-Architektur mit dem DLX-Simulator zeigen.

Hilfsmittel

- Buch von Andrew S. Tannenbaum und Todd Austin „Rechnerarchitektur“ (im Text mit [1] kenntlich gemacht)
- Beispiel example_6_input.s

Vorbereitung

Lesen Sie sich in die Dokumentation des DLX-Simulators (EinführungDLX.pdf) ein. Beachten Sie auch das Konzept von Traps, diese werden für den Versuch gebraucht. Im Tannenbaum sind diese im Kapitel 5.6.4 nachzulesen. Weiterhin ist es Hilfreich das Dokument „Befehlssatz.pdf“ (ausgedruckt) mit zu bringen. Beachten Sie, dass die Angaben für die Standardoptionen
(„use-forwarding = true“ und „Mips-compatibility-Mode = true“, die Branchprediction ist auf None gesetzt)

Aufgaben

In diesem Versuch soll zunächst ein Programm mit dem DLX-Simulator erstellt werden bei dem die grundlegende Funktion des Simmulators und seinem Befehlssatz klar werden soll.
Auf Grundlage dieses Programms sollen die Funktionsweise der Pipeline ausgearbeitet werden. Am Ende soll in der Theorie die Vorteile und Probleme von superskalaren Architekturen ergründet werden. Bei allen Aufgaben sollen die Fragen oder Ergebnisse in kurzen Sätzen beantwortet bzw. Dokumentiert werden. Die Lösung für Aufgabe 1 ist der lauffähige Quellcode für den Simulator.

1. Schreiben Sie ein Programm für den DLX-Simulator. Das Programm soll zunächst vier Integer-Zahlen von der Standardeingabe lesen, danach sollen diese (möglichst performant miteinander) multipliziert werden.
   Das Ergebnis soll wieder auf der Standardausgabe ausgegeben werden.
   Orientieren Sie sich dabei am Beispiel example_6_input.s aus den DLX Beispielen.
   Sie sollten etwa 20 Zeilen Code hinzufügen. Nach dem Lösen dieser Aufgabe können Sie entscheiden ob Sie mit Ihrer oder der Musterlösung weiterarbeiten wollen. Diese erhalten Sie auf Nachfrage beim Dozenten.

2. Simulieren sie nun die Abarbeitung des Programms auf dem DLX Simulator, benutzen sie dabei Zahlen < 10. Vollziehen sie nach, was genau die Pipelining Stufen des Simulators machen und welche Auswirkungen das hat.
   Gibt es Auffälligkeiten?
   Wie viele Zyklen braucht Ihr Programm?

3. Benutzen sie zahlen > 10 und erklären Sie den Unterschied der Ausführungszeit. Dafür können Sie die Theorien mit dem Simulator überprüfen.

4. Springen Sie in ihrem Programm vor die Stelle wo die Multiplikationen ausgeführt werden.
   Legen Sie dar ob und wenn ja, warum bzw. wenn nein, warum nicht, das Einführen einer Superskalaren-Architektur mit zwei ALUs sinnvoll ist. Nehmen Sie dabei an, dass Rechenoperationen (addi, multi, etc.) jeweils 2 Zyklen für jede Berechnung brauchen. Die Restlichen Stufen (außer das Speichern) der Pipeline jeweils nur einen.

5. Würde der DLX-Simulator auch Floatingpointoperationen unterstützen (was er nicht tut) und man würde zwei der Integerwerte mit Floatingpoint Zahlen ersetzten, würde die Reihenfolger der Operationen bei einer superskalaren Architektur (mit einer Integer ALU und einer Floatingpoint ALU, wobei die Floatingpoint ALU auch Integer als auch Floatingpoint Zahlen als Operanden nimmt) einen Unterschied machen?
   Wenn ja, was wäre die Optimale Reihenfolge der Operationen um das Produkt aller vier Faktoren zu bekommen?
   Welche Probleme können bei einer Anderen Reihenfolge auftreten?

Literatur

[1] Andrew S. Tanenbaum und Todd Austin. *Rechnerarchitektur : von der digitalen Logik zum Parallelrechner*. 6., aktualisierte Aufl. Informatik ; 4238. 799 S : Ill., graph. Darst. Hallbergmoos: Pearson, 2014. URL: `https://suche.suub.uni-bremen.de/peid=B77838189`.