Rechnerorganisation

Sommersemester 2023

Prof. Stefan Roth, Ph.D.

TECHNISCHE UNIVERSITÄT DARMSTADT

8. Aufgabenblatt mit Lösungsvorschlag

19.6.2023
Mikroarchitekturen

Aufgabe 1: Theoriefragen

a) Was ist der Unterschied zwischen einem Eintakt-Prozessor, einem Mehrtakt-Prozessor und einem Pipelined-Prozessor?

Lösungsvorschlag:

Beim Eintaktprozessor wird jede Instruktion in einem Takt ausgeführt. Beim Mehrtaktprozessor hingegen kann dies mehrere Takte dauern. Jede Instruktion wird also in Teilschritte zerlegt. Beim Pipelined-Prozessor werden die Instruktionen ebenfalls in Teilschritte zerlegt und die Teilschritte mehrerer Instruktionen (ggf.) gleichzeitig ausgeführt.

b) Warum benötigt das Registerfeld in dem vorgestellten Prozessor ein Write Enable Signal?

Lösungsvorschlag:

Da das Registerfeld nicht bei jedem Befehl überschrieben werden darf.

c) Erläutern Sie anhand des Datenpfads wie bei einem ldr/str Befehl die Adresse berechnet wird.

Lösungsvorschlag:

Zur Berechnung wird eine Addition mit der ALU ausgeführt. Am Eingang SrcA liegt der Wert des Registers, das die Basisadresse enthält, an. An SrcB liegt der erweiterte Immediate-Teil des Befehls.

d) Worin unterscheidet sich eine Von-Neumann-Architektur von einer Harvard-Architektur?

Lösungsvorschlag:

- Von-Neumann-Architektur: gemeinsamer Speicher für Maschinenbefehle und Daten
- Harvard-Architektur: Befehlsspeicher und Datenspeicher sind getrennt

Aufgabe 2: Opcodes und Bitfeldbelegung von Maschinenbefehlen

Befehle werden durch sogenannte Opcodes (Operationcodes) definiert. Der ARM®-Prozessor kennt außerdem einige Besonderheiten, wie z. B. die konditionale Befehlsausführung. Bei der Entwicklung des Eintakt-Prozessors wurde bereits die Bitfeldbelegung verschiedener Befehle vorgestellt. Diese Bitfeldbelegung erlaubt neben der Konstruktion des Datenpfads auch ein weiteres vertieftes Verständnis der Befehlskodierung, welches beim Disassemblieren hilfreich ist.

Betrachtet wird das folgende, einfache Assemblerprogramm.

```
/* -- analysis.s */
/* Kommentar  */
.global main /* Einsprungpunkt Hauptprogramm */

main:           /* Hauptprogramm */
    mov r1, #42 /* Schreibe eine 42 in das Register r1 */
    mov r2, #5 /* Schreibe eine 5 in das Register r2 */
    add r0,r1,r2 /* Addiere die Register r1 und r2 */
    bx lr       /* Springe zurueck zum aufrufenden Programm */
```

Assemblieren und Linken Sie das Programm. Schauen Sie sich nun den Object Dump an. Suchen Sie den Additionsbefehl und analysieren Sie die Belegung des Bitfeldes. Nutzen Sie dazu das ARM® Instruction Set[1] Handbuch. Insbesondere das Studium von Abschnitt 4.2 und 4.5 sollte hilfreich sein.

Lösungsvorschlag:

Object Dump:

```
00010408 <main>:
10408: e3a0102a mov r1, #42 ; 0x2a
1040c: e3a02005 mov r2, #5
10410: e0810002 add r0, r1, r2
10414: e12fff1e bx lr
```

Der Additionsbefehl ist hexadezimal codiert 0x e0810002. Die Umrechnung in das Binärsystem führt zu folgender Bitfolge: `1110 00 0 0100 0 0001 0000 0000 0000 0010`

Mittels Abschnitt 4.2 und 4.5 des ARM® Instruction Set Handbuchs lassen sich die Bitfelder analysieren.

Bit(s)	Wert	Kommentar
31:28	1110	CC always
27:26	00	Opcode Data Processing
25	0	0 = operand 2 is a register
24:21	0100	Operation Code (für add)
20	0	0 = do not alter condition codes
19:16	0001	Quellregister (r1)
15:12	0000	Zielregister (r0)
11:4	0000 0000	Shift-Feld
3:0	0010	2nd operand register (r2)

Aufgabe 3: Ägyptisches Multiplizieren

Das Produkt m zweier ganzer, vorzeichenbehafteter Zahlen a und b lässt sich leicht durch das sogenannte ägyptische Multiplikationsverfahren berechnen.

Der Multiplikand wird ständig verdoppelt, der Multiplikator (unter Wegwerfen des Restes) ständig halbiert; aufaddiert werden sogleich oder schlussendlich diejenigen Vielfachen, bei denen in der Multiplikatorhalbierung ein Rest weggeworfen wurde. Bedenken Sie zusätzlich Sonderfälle wie $b < 0$ und $a == 0$.

Beispiel: a = 11, b = 5

11	5
(22)	2
44	1
55	

[1] https://moodle.tu-darmstadt.de/pluginfile.php/1809291/mod_folder/content/0/Material/arm-instructionset.pdf

Die mit „(..)“gekennzeichnete Zahl wird für die Produktbildung nicht aufaddiert.

1. Implementieren Sie den Algorithmus in C. Die Benutzung der Multiplikation und Division (*, / und %) ist dabei nicht zulässig. Benutzen Sie stattdessen die Shiftoperationen.
2. Implementieren Sie den Algorithmus in ARM-Assembler. Dabei sollen 32-Bit Integer, also vorzeichenbehaftete Zahlen in 2K-Darstellung, verwendet werden. Beachten Sie dabei:
   - Die Multiplikationsbefehle sowie die Divisionsbefehle dürfen nicht verwendet werden.
   - Die Werte *a* und *b* können Sie fest im *.data*-Bereich unterbringen sowie andere Werte die gegebenenfalls benötigt werden.
   - Kommentieren Sie Ihre Lösung.
3. Testen Sie Ihr Programm mit den Werten

a	b
11	5
-99	99
13	-50
-72	-32
0	56

Lösungsvorschlag:

1. C-Programm zum ägyptischen Multiplizieren

```
#include <stdio.h>

int main() {

    int a, b, result;

    printf("\nBitte␣geben␣Sie␣den␣Wert␣von␣a␣ein:␣");
    scanf("%d", &a);
    printf("\nBitte␣geben␣Sie␣den␣Wert␣von␣b␣ein:␣");
        scanf("%d", &b);

    if(b < 0) {
        a = -a;
        b = -b;
    }
    while(b > 0) {
        if(b << 31 != 0)    //ist b ungerade
             result += a;   //dann addiere a auf das Ergebnis
        b >>= 1;        //Dividiere b mit 2
        a <<= 1;        //Multipliziere a mit 2
    }

    printf("\nDie␣Aegyptische␣Multiplikation␣von␣a␣und␣b␣ergibt␣%d\n", result);

    return 0;
}
```

2. Assemblerprogramm

```
.data
a:      .word 11
b:      .word 5
message: .asciz "The␣result␣is:␣%d\n"
return: .word 0
```

```
.text
.global main

main:
mov r0, #0      /* init the result of the computation with 0)       */
ldr r1, adr_a       /* address of a                      */
ldr r1, [r1]        /* value stored at address               */
ldr r2, adr_b       /* address of b                      */
ldr r2, [r2]        /* value stored at address               */
mov r3, #0      /* a has no sign                     */
mov r4, #0      /* b has no sign                     */

getsignofa:
cmp r1, #0      /* compare a with 0                  */
bge getsignofb      /* a >= 0 ?                      */
neg r1, r1      /* a = -a                        */
mov r3, #1      /* a has a sign                      */

getsignofb:
cmp r2, #0      /* compare b with 0                  */
bge while       /* b >= 0 ?                      */
neg r2, r2      /* b = -b                        */
mov r4, #1      /* b has a sign                      */

while:
cmp r2, #0      /* compare r2 with 0                     */
ble sign        /* r2 <= 0 ?                         */
lsl r12, r2, #31    /* shift b 31bits to the left with a temporary register */
cmp r12, #0     /* compare r12 with 0                    */
beq comp        /* if b is even prepare for the next iteration?     */
add r0, r0, r1      /* if b is odd: result += a              */

comp:
lsl r1, r1, #1      /* a = a * 2                         */
lsr r2, r2, #1      /* b = b / 2                         */
b while         /* jump to while                     */

sign:
eor r4, r3, r4      /* xor of both signs gives the resulting sign       */
cmp r4, #0      /* if there is no sign in the result then ...       */
beq print       /* jump to print                     */
neg r0, r0      /* otherwise negate the answer of the computation   */

print:
ldr r1, adr_return  /* address of return                     */
str lr, [r1]        /* store the value into the return address      */
mov r1, r0      /* move the answer of the computation into r1       */
ldr r0, adr_message /* address of message                    */
bl printf       /* call printf method                    */
ldr lr, adr_return  /* address of return                     */
ldr lr, [lr]        /* load the value into lr                */
bx lr           /* jump back to the calling programm             */

adr_a: .word a
adr_b: .word b
adr_message: .word message
adr_return: .word return
```