Praktikumsaufgabe: 8-Bit-Assembler

Prof. Dr. Ralf Gerlich
Hochschule Furtwangen
Fakultät Informatik
Robert-Gerwig-Platz 1
D-78120 Furtwangen
ralf.gerlich@hs-furtwangen.de

8. August 2024

Zusammenfassung

Implementieren Sie 16-, 24- und 32-Bit-Operationen in Assembler auf dem 8-Bit-Mikrocontroller des Arduino.

Inhaltsverzeichnis

1 Schaltungsaufbau

Für diese Aufgabe müssen Sie lediglich den Arduino per USB-Schnittstelle mit Ihrem Host-Rechner verbinden. Ein gesonderter Schaltungsaufbau ist nicht erforderlich.

2 Automatische Tests

Die Projektvorlage enthält integrierten Test-Code, mit denen Sie die Korrektheit Ihrer Implementierungen prüfen können. Zur Ausführung der implementierten Unit-Tests (Datei `test/test.c`) gehen Sie wie folgt vor:

1. Schließen Sie den Arduino an Ihren Rechner an.
2. Führen Sie die Tests aus:

VSCode Die Tests werden ausgeführt, wenn Sie das „Test“-Kommando aus der Gruppe „Advanced“ ausführen.

CLion Hier scheint es keinen direkten Support zu geben. Öffnen Sie mit einem Rechtsklick auf den Projektordner das Projektkontextmenü und wählen Sie unter „Open In...“ den Eintrag „Terminal“. Im sich öffnenden Terminalfenster tippen Sie das Kommando `pio test`, um die Tests auszuführen.

Andere Hier können Sie ähnlich wie bei CLion verfahren.

Die Ergebnisse der Tests werden Ihnen dann auf der Konsole ausgegeben.

3 Rechnen mit 8 Bit

Der AVR ist eine 8-Bit-Architektur, d.h. alle Register haben nur 8 Bit. Werte mit mehr als 8 Bit müssen auf mehrere Register aufgeteilt werden.

Wir nutzen zur Darstellung der Registerbelegung die Form `rh:rl`. Es bedeutet, dass das niederwertige Byte des Werts in `rl` und das höchstwertige Byte in `rh` gespeichert ist.

Zum Beispiel kann ein 32-Bit-Wert auf die Register `r25:r22` aufgeteilt werden. Das höchstwertige Byte liegt dann in `r25`, das niederwertige Byte in `r22`:

r25	r24	r23	r22

4 Aufgabenstellung

In dieser Aufgabe sollen Sie einige grundlegenden arithmetischen Operationen in AVR-Assembler implementieren.

Eine Übersicht der wichtigsten AVR-Assemblerbefehle finden Sie im Anhang zu diesem Aufgabenblatt (Abschnitt A). Genauere Beschreibungen finden Sie in der AVR Instruction Set Reference im Projekttemplate.

In der Datei `lib/asm_arith/src/arith.S` sind Vorlagen für die zu implementierenden Funktionen vorhanden. Ersetzen Sie die mit `TODO` markierten Stellen durch Ihren Code.

4.1 16-Bit-Addition (2 Punkte)

Implementieren Sie in der Funktion `add16` die Addition von zwei vorzeichenbehafteten 16-Bit-Zahlen, wobei das Ergebnis als 16-Bit-Zahl zurückgegeben werden soll. Die beiden Parameter der Funktion werden in den Registern `r25:r24` und `r23:r22` übergeben. Das Ergebnis soll in `r25:r24` abgelegt werden.

Der AVR ist eine 8-Bit-Architektur, d.h. einzelne Befehle (darunter auch die Additionsbefehle) implementieren nur Operationen mit 8-Bit-Operanden.

Für eine 16-Bit-Addition sind also mehrere Operationen erforderlich. Betrachten Sie analog die schriftliche Addition von Zahlen im Dezimalsystem mit mehreren Ziffern, z.B. $19 + 21$.

4.2 16-Bit Shift Left (2 Punkte)

Implementieren Sie in der Funktion `shl16` eine Bitverschiebung nach links um ein Bit (Left Shift). Der Parameter der Funktion wird in den Registern `r25:r24` übergeben. Das Ergebnis soll in denselben Registern abgelegt werden.

Überlegen Sie sich, was jeweils mit dem höchstwertigen Bit der beiden Bytes passieren soll.

Hinweis: Der AVR bietet zwei Befehle zur Linksverschiebung, `lsl` und `rol`.

4.3 8-Bit-Multiplikation (4 Punkte)

Implementieren Sie in der Funktion `mult8_8` eine Multiplikation zweier vorzeichenbehafteter 8-Bit-Zahlen. Die Parameter der Funktion werden in den Registern `r24` bzw. `r22` übergeben. Das Ergebnis soll in den Registern `r25:r24` abgelegt werden.

Überlegen Sie, warum für das Ergebnis zwei 8-Bit-Register erforderlich sind!

Beachten Sie, dass die Register `r0` und `r1` durch die Multiplikationsbefehle überschrieben werden! Sofern erforderlich, sichern Sie sie am Anfang der Funktion und stellen Sie vor Ende der Funktion wieder her. Ob die Sicherung erforderlich ist, können Sie der Übersicht im Anhang A entnehmen.

4.4 8-zu-16-Bit-Multiplikation (8 Punkte)

Implementieren Sie in der Funktion `multu_8_16` eine Multiplikation einer vorzeichenlosen 8-Bit-Zahl mit einer vorzeichenlosen 16-Bit-Zahl. Die Parameter der Funktion werden in den Registern `r24` bzw. `r23:r22` übergeben. Da kein 24-Bit-Typ existiert, soll das Ergebnis als 32-Bit-Wert in den Registern `r25:r22` gespeichert werden.

Beachten Sie, dass Quell- und Zielregister der Operation sich überlappen. Sie müssen also zunächst die Inhalte der Quellregister „retten“. Dies können Sie tun, indem Sie den Inhalt von `r24` mit dem `mov`-Befehl in `r20` und den Inhalt von `r23:r22` mit `movw` in `r19:r18` kopieren. Diese Register dürfen Sie laut Aufrufkonvention benutzen, ohne ihren Inhalt vorher sichern zu müssen.

Sie müssen die Multiplikation in mehrere Teile aufteilen. Sei a die 8-Bit-Zahl und b die 16-Bit-Zahl. Letztere lässt sich durch ihren höherwertigen Anteil b_h und ihren niederwertigen Anteil b_l darstellen:

$$b = b_h 2^8 + b_l$$

Dabei liegt die Kopie von a in `r20`, die Kopie von b_h liegt in `r19` und die Kopie von b_l liegt in `r18`.

Die Multiplikation ab kann nun also wie folgt dargestellt werden:

$$ab = a\left(b_h 2^8 + b_l\right) = ab_h 2^8 + ab_l$$

Die folgende Tabelle illustriert, wie sich das Ergebnis zusammensetzt:

Register	r25	r24	r23	r22
	0	0	r20 * r18	
+	0	r20 * r19		0
=	ab			

Um ein Register `rX` auf Null zu setzen, können Sie den Befehl `eor rX, rX` verwenden. Überlegen Sie, warum das Ergebnis dieser Operation immer Null ist!

4.5 16-Bit-Multiplikation (12 Punkte)

Implementieren Sie in der Funktion `multu_16_16` eine Multiplikation zweier vorzeichenloser 16-Bit-Zahlen. Die Parameter der Funktion werden in den Registern `r25:r24` bzw. `r23:r22` übergeben. Das Ergebnis soll in den Registern `r25:r22` gespeichert werden.

Sichern Sie den ersten Parameter (`r25:r24`) in die Register `r21:r20` und den zweiten Parameter (`r23:r22`) in die Register `r19:r18`. Sie können außerdem die Register `r26` und `r27` als Hilfsregister oder für Zwischenergebnisse nutzen.

Auch hier muss die Multiplikation in mehrere Teile aufgeteilt werden. Sei a der erste Parameter und b der zweite Parameter. Beide lassen sich in ihren höher- und ihren niederwertigen Teil aufteilen:

$$\begin{aligned} a &= a_h 2^8 + a_l \\ b &= b_h 2^8 + b_l \end{aligned}$$

Dabei liegt a_h in `r21`, a_l in `r20`, b_h in `r19` und b_l in `r18`.

Die Multiplikation ab kann nun also wie folgt dargestellt werden:

$$ab = \left(a_h 2^8 + a_l\right)\left(b_h 2^8 + b_l\right) = a_h b_h 2^{16} + \left(a_l b_h + a_h b_l\right) 2^8 + a_l b_l$$

Die folgende Tabelle illustriert, wie sich das Ergebnis zusammensetzt:

Register	r25	r24	r23	r22
	r21 * r19		0	0
+	0	0	r20 * r18	
+	r21 * r18			0
+	r20 * r19			0
=	ab			

Beachten Sie, dass es sich bei den letzten beiden Additionen in der Tabelle jeweils um Additionen eines 16-Bit-Werts auf einen 24-Bit-Wert handelt!

5 Aufrufkonventionen

Bestimmte Register dürfen von der aufgerufenen Funktion ohne weiteres überschrieben werden. Alle anderen Register müssen von der aufgerufenen Funktion vor dem Überschreiben mit `push` auf dem Stack gespeichert und vor der Rückkehr aus der Funktion mit `pop` wiederhergestellt werden (sog. call-saved registers). Der AVR-Kurzreferenz im Anhang an dieses Aufgabenblatt können Sie entnehmen, welcher der beiden Kategorien die Register jeweils unterliegen.

Achtung: Beachten Sie, dass es sich beim Stack um eine Last-In-First-Out-Datenstruktur handelt! Die Register müssen also in umgekehrter Reihenfolge wieder vom Stack geholt werden, als sie auf den Stack gelegt wurden.

A Übersicht AVR-Assemblerbefehle

AVR Kurzreferenz

Konventionen

Symbol	Bedeutung
Rd, Rr	Register (R0-R31)
Rd+1:Rd	Register-Paar ($d \in \{24, 26, 28, 30\}$)
X|Y|Z	Eines der 16-Bit-Index-Register (X, Y oder Z)
imm8	8-Bit-Konstante
imm5	5-Bit-Konstante
addr16	16-Bit Adresskonstante (ggf. Label)
addr22	22-Bit Adresskonstante (ggf. Label)
raddr	Relative 7-Bit-Adresse (Zweierkomplement)
MEM[a]	Byte an Adresse a
STACK	Stack
PC	Program Counter

Die Index-Register sind die Registerpaare R27:R26 (X), R29:R28 (Y) und R31:R30 (Z).

Arithmetikbefehle

Instruktion	Operation	Flags	Beschreibung
add Rd, Rr	Rd←Rd+ Rr	Z,C,N,V,S	Add without Carry
adc Rd, Rr	Rd←Rd+ Rr+	Z,C,N,V,S	Add with Carry
adiw Rd, imm5	Rd+1:Rd←Rd+1:Rd+ imm5	Z,C,N,V,S	Add Immediate to Word
sub Rd, Rr	Rd←Rd- Rr	Z,C,N,V,S	Subtract without Carry
sbc Rd, Rr	Rd←Rd- Rr- C	Z,C,N,V,S	Subtract with Carry
sbiw Rd, imm5	Rd+1:Rd←Rd+1:Rd- imm5	Z,C,N,V,S	Subtract Immedate from Word
subi Rd, imm8	Rd←Rd- imm8	Z,C,N,V,S	Subtract Immedate
sbci Rd, imm8	Rd←Rd- imm8- C	Z,C,N,V,S	Subtract Immedate with Carry
neg Rd	Rd←- Rd	Z,C,N,V,S	Negate
mul Rd, Rr	R1:R0 ←Rd × Rr	Z,C	Multiply Unsigned
muls Rd, Rr	R1:R0 ←Rd × Rr	Z,C	Multiply Signed ($16 \leq d, r$)
mulsu Rd, Rr	R1:R0 ←Rd × Rr	Z,C	Multiply Signed with Unsigned ($16 \leq d, r \leq 23$)
inc Rd	Rd←Rd+ 1	Z,N,V,S	Increment
dec Rd	Rd←Rd- 1	Z,N,V,S	Decrement

Ganzzahlvergleiche

Instruktion	Operation	Flags	Beschreibung
cp Rd, Rr	Rd- Rr	Z,C,N,S,V	Compare
cpc Rd, Rr	Rd- Rr- C	Z,C,N,S,V	Compare with Carry
cpi Rd, imm8	Rd- imm8	Z,C,N,S,V	Compare with Immediate

Bitweise Arithmetik

Instruktion	Operation	Flags	Beschreibung
and Rd, Rr	Rd←Rd ∧ Rr	Z,N,V=0,S	Bitwise AND
andi Rd, imm8	Rd←Rd ∧ imm8	Z,N,V=0,S	Bitwise AND immediate
or Rd, Rr	Rd←Rd ∨ Rr	Z,N,V=0,S	Bitwise OR
ori Rd, imm8	Rd←Rd ∨ imm8	Z,N,V=0,S	Bitwise OR immediate
eor Rd, Rr	Rd←Rd ⊕ Rr	Z,N,V=0,S	Bitwise XOR
com Rd	Rd←$\overline{\text{Rd}}$	Z,C=1,N,V=0,S	Bitwise NOT
sbr Rd, imm8	Rd←Rd ∨ imm8	Z,N,V=0,S	Set bits in register
cbr Rd, imm8	Rd←Rd ∧ $\overline{\text{imm8}}$	Z,N,V=0,S	Clear bits in register
lsl Rd	C:Rd←Rd << 1	Z,C,N,V	Logical Shift Left
lsr Rd	Rd:C ←Rd	Z,C,N,V	Logical Shift Right
asr Rd	Rd:C ←Rd	Z,C,N,V	Arithmetic Shift Right
rol Rd	C:Rd←Rd:C	Z,C,N,V	Rotate Left through Carry
ror Rd	Rd:C ←C:Rd	Z,C,N,V	Rotate Right through Carry

Sprünge

Instruktion	Operation	Beschreibung
breq raddr	if Z=1 then PC←PC + 1 + raddr	Branch if equal
brne raddr	if Z=0 then PC←PC + 1 + raddr	Branch if not equal
brcs raddr	if C=1 then PC←PC + 1 + raddr	Branch if carry set
brcc raddr	if C=0 then PC←PC + 1 + raddr	Branch if carry clear
brlo raddr	if C=1 then PC←PC + 1 + raddr	Branch if lower
brsh raddr	if C=0 then PC←PC + 1 + raddr	Branch if same or higher
brmi raddr	if N=1 then PC←PC + 1 + raddr	Branch if minus
brpl raddr	if N=0 then PC←PC + 1 + raddr	Branch if plus
brlt raddr	if S=1 then PC←PC + 1 + raddr	Branch if less than
brgt raddr	if S=0 then PC←PC + 1 + raddr	Branch if greater or equal
brvs raddr	if V=1 then PC←PC + 1 + raddr	Branch if overflow set
brvc raddr	if V=0 then PC←PC + 1 + raddr	Branch if overflow clear
jmp addr22	PC←addr22	Jump
rjmp raddr	PC←PC + 1 + raddr	Relative Jump
ijmp	PC←Z	Indirect Jump to Z
call addr22	STACK←PC, PC←addr22	Call to Subroutine
rcall raddr	STACK←PC, PC←PC + 1raddr	Relative Call to Subroutine
icall	STACK←PC, PC←Z	Call to Z
ret	PC←STACK	Return from Subroutine
reti	PC←STACK, I=1	Return from Interrupt

Hinweis: Bis auf reti verändern die Sprungoptionen das Flags-Register nicht.

Erstellt von: Prof. Dr. Ralf Gerlich, ralf.gerlich@hs-furtwangen.de

Fassung vom 26. Januar 2024

Transferbefehle

Instruktion	Operation	Beschreibung
`mov Rd, Rr`	Rd←Rr	Copy Register
`movw Rd, Rr`	Rd+1:Rd←Rr+1:Rr	Copy Register Pair (d, r even)
`ldi Rd, imm8`	Rd←imm8	Load Immediate
`lds Rd, addr16`	Rd←MEM[addr16]	Load from Data Space
`ld Rd, X|Y|Z`	Rd←MEM[X|Y|Z]	Load indirect
`ld Rd, X|Y|Z+`	Rd←MEM[X|Y|Z], X|Y|Z←X|Y|Z+ 1	Load ind. w. post-increment
`ld Rd, -X|Y|Z`	X|Y|Z←X|Y|Z- 1, Rd←MEM[X|Y|Z]	Load ind. w. pre-decrement
`ldd Rd, X|Y|Z+imm5`	Rd←MEM[X|Y|Z + imm5]	Load ind. w. displacement
`sts addr16, Rr`	MEM[addr16] ←Rr	Store to Data Space
`st X|Y|Z, Rr`	MEM[X|Y|Z] ←Rr	Store indirect
`st X|Y|Z+, Rr`	MEM[X|Y|Z] ←Rr, X|Y|Z←X|Y|Z+ 1	Store ind. w. post-increment
`st -X|Y|Z, Rr`	X|Y|Z←X|Y|Z- 1, MEM[X|Y|Z] ←Rr	Store ind. w. pre-decrement
`std X|Y|Z+imm5, Rr`	MEM[X|Y|Z + imm5] ←Rr	Store ind. w. displacement
`push Rr`	STACK←Rr	Push Register on Stack
`pop Rd`	Rr←STACK	Pop Register from Stack
`lpm Rd, Z`	Rd←MEM[Z]	Load from Program Memory
`lpm Rd, Z+1`	Rd←MEM[Z], Z←Z+ 1	`lpm` with post-increment

Flag-Manipulation

Instruktion	Operation	Beschreibung
`sec`	C ←1	Set carry flag
`clc`	C ←0	Clear carry flag
`sen`	N ←1	Set negative flag
`cln`	N ←0	Clear negative flag
`sez`	Z ←1	Set zero flag
`clz`	Z ←0	Clear zero flag
`ses`	S ←1	Set signed flag
`cls`	S ←0	Clear signed flag
`sev`	V ←1	Set overflow flag
`clv`	V ←0	Clear overflow flag
`sei`	I ←1	Set global interrupt flag
`cli`	I ←0	Clear global interrupt flag

`avr-gcc` Aufrufkonventionen

Register	Call-saved?	Hinweis
`R0`	nein	Register für temporäre Ergebnisse
`R1`	ja	Wird als Zero-Register verwendet (immer 0)
`R2-R7`	ja	
`R8-R17`	ja	u.a. Parameter
`R18-R25`	nein	u.a. Parameter und Rückgabewert
`R26-R27`	nein	X-Index-Register
`R28-R29`	ja	Y-Index-Register, wird von gcc als Frame-Pointer verwendet
`R30-R31`	nein	Z-Index-Register

- Rückgabewerte von bis zu 8 Byte Größe werden in Registern zurückgegeben.
- Für Rückgabewerte größer als 8 Byte reserviert der Aufrufer Platz auf dem Stack. Die Adresse dieses Bereichs wird als impliziter erster Parameter übergeben.
- Parameter werden in den Registern `R8` bis `R25` übergeben. Der folgende Algorithmus entscheidet, in welchen Registern die Parameter liegen:
  1. Setze `n` = 26.
  2. Für jeden Parameter `p`, beginnend beim ersten:
     (a) Sei `s` die Größe des Parameters in Bytes, aufgerundet auf die nächste gerade Zahl.
     (b) Setze `n=n-s`.
     (c) Wenn `n>=8`: Speichere das niederwertigste Byte von `p` in `Rn` und weitere Bytes in den darauffolgenden Registern.
     (d) Wenn `n<8`: Übergebe den Parameter auf dem Stack und beende die Schleife hier. Alle weiteren Parameter werden ebenfalls auf dem Stack übergeben.
- Wird der Rückgabewert in einem Register zurückgegeben, werden dieselben Register verwendet, die für einen ersten Parameter derselben Größe verwendet würden. Beispiel: Ein 8-Bit-Wert wird im Register `R24` zurückgegeben, ein 32-Bit-Wert in den Registern `R25:R22`.