Embedded Systems

Assembler

Praktikum 6

Fachhochschule Bielefeld
Campus Minden
Studiengang Informatik

Beteiligte Personen:

Name	Matrikelnummer
Jan-Hendrik Sünderkamp	1153536
Peter Dick	1050185

---

10. Juni 2018

Inhaltsverzeichnis

1 Vorbereitung

In der Vorlesung wurden ARM-Assembler und Inline-Assembler für den ARM-GCC besprochen. Lesen Sie hierzu für weitere Informationen die Webseite `http://www.ethernut.de/en/documents/arm-inline-asm.html` und `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/CIHJJEIH.html`
(Cortex-M4 Devices Generic User Guide, Kapitel 3 "The Cortex-M4 Instruction Set" im Ilias).

2 Aufgabe 1

Ergänzen Sie das Listing 6.1 mit Assembler-Befehlen in den angegebenen Bereich, so dass Ihr ergänzter Programmcode die Nummer *number* verändert. Die Nummer muss bei jedem Durchlauf verdoppelt werden, bis sie den Wert 128 erreicht hat. Danach muss die Nummer wieder auf den Wert 1 gesetzt werden und die Verdoppelung vorne beginnen.

```
// our working number
volatile uint8_t number = 1;

// baudrate for serial communication
const int baudRate = 9600;

// initialize Serial
void setup() {
  Serial.begin(baudRate);
}

// main loop
void loop() {
  // print number
  Serial.println(number);

  asm volatile(
    "cmp %[n], #128\n\t"
    "ite ne\n\t"
    "addne %[n], %[n]\n\t"
    "moveq %[n], #1\n\t"
    : [n] "=r" (number)
    : "0" (number)
    : "cc", "memory"
  );
}
```

Listing 6.1: Rahmen für Aufgabe 1

3 Aufgabe 2

Ergänzen Sie das Listing 6.2 mit Assembler-Befehlen in den angegebenen Bereich, so dass das Array fibData mit den ersten dreizehn Fibonacci-Zahlen gefüllt wird. Dabei ist die dritte bis dreizehnte Fibonacci-Zahl jeweils aus ihren beiden vorhergehenden zu berechnen. Hinweis: Eine mögliche Wissenslücke bezüglich Fibonacci-Zahlen könnte Wikpedia füllen (`http://de.wikipedia.org/wiki/Fibonacci-Folge`). Beachten Sie, dass nicht alle Register benutzt werden können und verwenden Sie z.B. die Register R4 bis R7.

```
// our working data
const uint8_t lastFiboIndex = 13;
uint8_t fibData[lastFiboIndex];
const int waitTime = 1000;

// baudrate for serial communication
const int baudRate = 9600;

void setup() {
  // initialize Serial
  Serial.begin(baudRate);

  // init first two Fibonacci numbers
  fibData[0] = 1;
  fibData[1] = 2;

  asm volatile(
    "ldr r4, [%[fD], #0]\n\t"
    "add.w r5, r4, %[lFi]\n\t"
    "back: mov r6, r4\n\t"
    "ldr     r5, [%[lFi], #0]\n\t"
    "cmp     r5, %[lFi]\n\t"
    "bhi     exit\n\t"
    "adds r4, #1\n\t"
    "ldrb r7, [r6, #1]\n\t"
    "ldrb r6, [r6, #0]\n\t"
    "cmp r4, r5\n\t"
    "add r7, r6\n\t"
    "strb r7, [r4, #1]\n\t"
    "ldr     r5, [%[lFi], #0]\n\t"
    "add     r5, r5, #1\n\t"
    "str     r5, [%[lFi], #0]\n\t"
    "b       back\n\t"
    "exit:\n\t"
    :: [fD] "r" (fibData[2]), [lFi] "r" (lastFiboIndex -2)
    : "r4" , "r5" , "r6" , "r7" , "cc" , "memory"
```

```
  );
}

// main loop
void loop() {
  // print data
  for (int i = 0; i < lastFiboIndex; i++)
  {
    Serial.println(fibData[i]);
  }
  // delay 1s
  delay(waitTime);
}
```

Listing 6.2: Rahmen für Aufgabe 2

Bestimmen Sie für den Assembler-Code beider Programme die Anzahl der Taktzyklen, die diese benötigen.

Aufgabe 1:
cmp: 1 Taktzyklus
ite: 1 Taktzyklus oder 0 Taktzyklen (wenn am letzten Thumb-Befehl angehängt)
addne: 1 Taktzyklus
moveq: 1 Taktzyklus
Da addne nur dann ausgeführt wird wenn moveq nicht ausgeführt wird, hat Aufgabe 1 nur 3 bzw 2 (Wenn ite 0 Taktzyklen hat) Taktzyklen.

Aufgabe 2:
ldr: 2 Taktzyklen
add.w 1 Taktzyklus

back:
mov: 1 Taktzyklus
ldr: 2 Taktzyklen
cmp: 1 Taktzyklus
bhi: 1 Taktzyklus oder 1 + P Taktzyklen (wenn branch ausgeführt wird)
adds: 1 Taktzyklus
2 mal ldrb: 2 Taktzyklen
cmp: 1 Taktzyklus
add: 1 Taktzyklus
strb + ldr: 2 Taktzyklen
add: 1 Taktzyklus
str: 2 Taktzyklen
b: 1 + P Taktzyklen

Sektion back = 1 + 2 + 1 + 1 + 1 + 2 + 1 + 1 + 2 + 1 + 2 + 1 + P = 16 + P

Zuerst wird ldr und add.w mit 2 + 1 = 3 Taktzyklen ausgeführt. Danach wird 11 mal die Sektion back mit 11 x (16 + P) = 176 + 11 x P Taktzyklen ausgeführt. Am ende wird mov, ldr, cmp und bhi mit 1 + 2 + 1 + 1 + P = 5 + P Taktzyklen ausgeführt. Das ergibt 3 + 176 + 11 x P + 5 + P = 184 + 12 x P Taktzyklen.

Überlegen Sie Sich weiterhin, wie Sie sicherstellen können, dass durch Ihren Programmcode veränderte Register nach der Durchführung ihre ursprüngliche Werte bekommen.

Mithilfe des Ausdrucks ”memory” in der ”clobber list” wird dem Compiler mitgeteilt dass er alle Werte im Cache zwischenspeichern soll und nach der Ausführung des Assembler-Codes alle Werte neu laden soll.