RUHR-UNIVERSITÄT BOCHUM

RUB

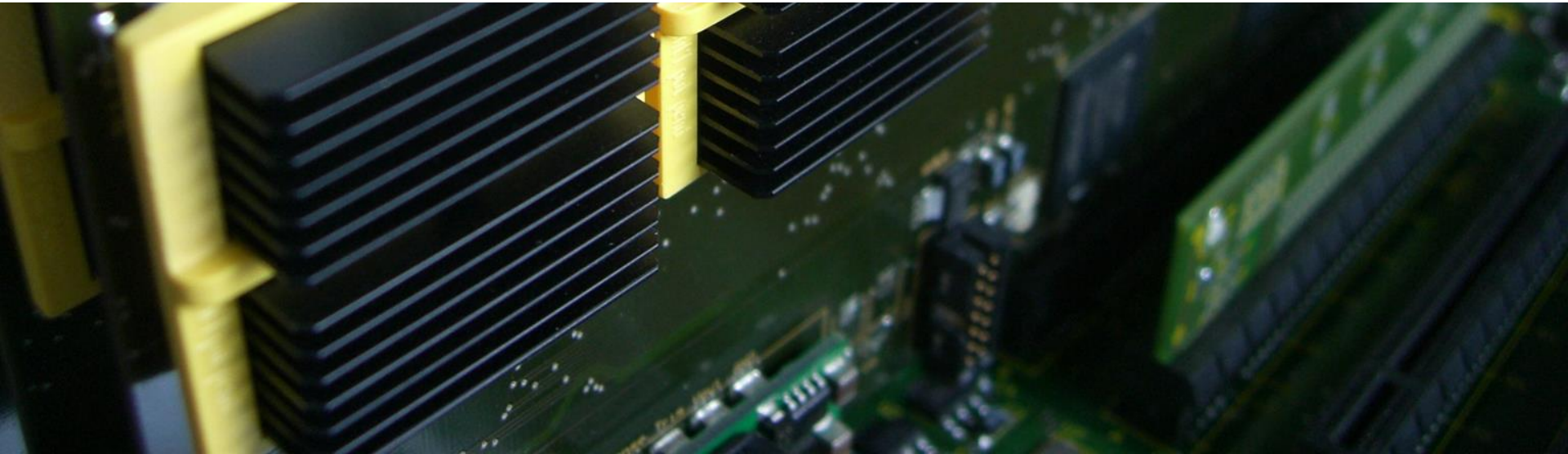
Eingebettete Prozessoren

SS 2014

Übung 6: Programmflusssteuerung

Dipl.-Ing. Thomas Pöppelmann
Arbeitsgruppe Sichere Hardware
Horst Görtz Institut für IT-Sicherheit

29.05.2014



Agenda

1. Organisatorisches
2. (Un-)bedingte Sprünge
3. Makros und Funktionen

1. Organisatorisches

1. Organisatorisches

- Zu dieser Übung findet/fand keine Präsenzübung statt (Himmelfahrt)
- Ingo und Thomas sind vom 02.06-06.06.2014 nicht im Büro. Die Übung am 05.06.2014 wird durch Alexander Wild vertreten.
- Ausführliche Erklärung im Skript unter:
  - „Erweiterte Programmflusssteuerung“
- AVR-Tutorial:
  - https://www.mikrocontroller.net/articles/AVR-Tutorial:_Vergleiche
  - https://www.mikrocontroller.net/articles/AVR-Tutorial:_Stack

3. (Un-)bedingte Sprünge

Unbedingte Sprünge

- Unbedingte Sprünge
  - Verzweigt zu einem beliebigen anderen Befehl
  - Unabhängig vom aktuellen Zustand wird immer an die selbe Stelle gesprungen
  - (R)JMP <Sprungmarke>

```
RJMP  JUMPHERE ; Springe zum Befehl bei Marke JUMPHERE
NOP
...
JUMPHERE: SUB R17 R18
```

Unbedingte Sprünge

- Verwendung von unbedingten Sprüngen
  - Überspringen von Datenbereichen im Code
  - Springen an das Ende eines Codeblocks nachdem ein Pfad eine bedingten (Mehrfach-) Verzweigung abgearbeitet wurde
  - Zurückspringen in den Schleifenkopf am Ende einer Schleife

Bedingte Sprünge

- Bedingte Sprünge werden eingesetzt um die aus C bekannten Kontrollstrukturen wie Schleifen und IF-Abfragen umzusetzen
- Genereller Ablauf
  - Es wird ein Vergleich durchgeführt (z.B. i<5)
  - Abhängig vom Ausgang des Vergleichs wird ein bedingter Sprung durchgeführt
    - Wenn true: Springe zu Programmcode der Do_A ausführt
    - Wenn false: Springe zu Programmcode der Do_B ausführt
  - Durch geschicktes Nutzen der Abfragen/Befehle und Wissen über wahrscheinliche Ereignisse lassen sich in Assembler oft Sprünge einsparen

```
If (i<5) {
   ->Do_A
}
->Do_B
```

Siehe auch:
https://www.mikrocontroller.net/articles/AVR-Tutorial:_Vergleiche

Bedingte Sprünge

- Bedingte Sprünge
  - Verzweigung hängt von einer Bedingung ab
    1. Vergleichen mit CP / CPC / CPI
    2. Abhängig vom Ergebnis verzweigen mit BRxx
- Bsp. if(a == b){…} else {…}

```
CP R16, R17
BREQ gleich
ungleich: … RJMP ende
gleich: …
ende: …
```

Bedingte Sprünge

- Es können auch mehrere Bedingungen hintereinander geprüft werden!
  - Bsp: if(a<2){…} else if(a==2){…} else {…}

```
CPI R16, 2
BREQ gleich
BRLO kleiner
groesser: … RJMP ende
gleich: … RJMP ende
kleiner: …
ende: …
```

Beispiele im Skript unter: Erweiterte Programmflusssteuerung

3. Makros und Funktionen

Makros und Funktionen

- Wiederkehrende Aufgaben sollten als Unterfunktionen oder Makros realisiert werden
  - Einfachere Wartbarkeit des Codes
  - Lesbarkeit wird erhöht
  - Codegröße kann massiv gesenkt werden (Funktionen)
- Makros werden durch den Compiler an die entsprechenden Stellen im Code kopiert
- Funktionen existieren nur 1x im Code

Makros

- Programmfluss wird nicht manipuliert, Codesequenz wird lediglich an entsprechenden Stellen ersetzt
- Parameter werden in Übergebungsreihenfolge mit @0,@1,… angesprochen

```
.MACRO ADD_16 ; Berechnet @1:@0 + @3:@2
ADD @0, @2
ADC @1, @3
.ENDMACRO
; Aufruf im Code
ADD_16 R16, R17, R18, R19
```

Funktionen

- Stapelspeicher (Stack) muss initialisiert sein um Funktionen verwenden zu können!
- Initialisierung muss einmalig zu Beginn des Programms manuell vorgenommen werden
- Stack wird benötigt um Rücksprungadresse zu speichern und Register zu sichern

```
LDI R16, HIGH(RAMEND); oberes Byte von RAMEND in R16 laden
OUT SPH, R16         ; R16 in höherwertigen Teil des SP
LDI R16, LOW(RAMEND) ; niederes Byte von RAMEND in R16 laden
OUT SPL, R16         ; R16 in niederwertigen Teil des SP
```

Funktionen

- „Funktionsdefinition“ einfach über Sprungmarke
- Aufruf mittels RCALL/CALL <Sprungmarke>
- Parameter/Rückgabe werden implizit übergeben

```
ADD_16:
ADD R16, R18
ADC R17, R19
RET

…
RCALL ADD_16
```

Funktionen

- Wenn Register innerhalb einer Funktion überschrieben werden sollten dieser vorher gesichert und am Ende wiederhergestellt werden!

```
SOME_FUNCTION:
PUSH R16
… ; Instruktionen die R16 überschreiben
POP R16
RET
```

Vorteile/Nachteile

- Funktionen reduzieren die Codegröße, da nur einmal vorhanden. Allerdings bedeutet das Einspringen in eine Funktion auch Overhead (rcall/Stack). Bei der Verwendung von Funktionen muss der Stack initialisiert werden.
- Mit Hilfe von Macros lassen sich bestimmte Aufgaben übersichtlich lösen ohne den Overhead einer Funktion zu benötigen.
- Macros können beim Debugging unübersichtlich werden, abhängig von der verwendeten IDE/Debugger.
- Funktionen werden meistens so geschrieben, dass alle Register die in der Funktion verändert werden beim Aufruf auf dem Stack gesichert werden. Damit ist dann ein Einsatz der Funktion an jeder Stelle möglich, ohne das Hauptprogramm zu beeinflussen. Bei Macros muss ebenfalls sicher gestellt werden, dass keine Register unbeabsichtigt beeinfluss werden (schwer zu lokalisierende Fehlerquelle).

Ausrollen (Ausblick)

- Mit Macros können Schleifen einfach und lesbar „ausgerollt“ werden (das C Beispiel muss natürlich in Assembler übertragen werden)
  - 1) Code wird gespart, dafür Overhead durch Schleifenkopf (Abfrage ob *i*<5, inkrementieren von *i*)
  - 2) Kein Overhead, dafür ist der Code von SOME_MACRO fünf mal vorhanden
- Ausrollen nicht möglich, wenn die Anzahl der Schleifendurchläufe nicht bekannt ist (z.B. abhängig von Benutzereingaben)

```
for(i=0; i<5; i++){
  SOME_MACRO
}
```

1)

```
SOME_MACRO
SOME_MACRO
SOME_MACRO
SOME_MACRO
SOME_MACRO
```

2)