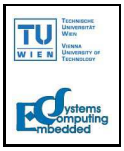

Wolfgang Puffitsch
hausen@gmx.at

Softwaretools für den SPEAR

22. Mai 2007

Inhaltsverzeichnis

Tabellenverzeichnis

Listings

1 Pakete

Die Softwaretools für den SPEAR bestehen aus drei Paketen: cas, spear-binutils und spear32-gcc. Ersteres ist von keinem der anderen Pakete abhängig, spear-binutils ist von cas abhängig, spear32-gcc von spear-binutils.

cas besteht aus dem Programm `cas` und einigen dazugehörigen Include-Dateien. spear-binutils besteht aus den Programmem `as` und `ld`, Dateien die die Library-Funktionen beinhalten, die von spear32-gcc verwendet werden und einigen Header-Dateien, die die Definitionen für die vorhandenen Teile der libc enthalten. spear32-gcc besteht wiederum aus den Programmen `gcc` bzw. `spear32-gcc` und einigen Header-Dateien, die sich auf grundlegende Eigenschaften des Prozessors beziehen.

2 Assembler und Linker

2.1 Beschreibung

Im Verzeichnis `/usr/local/spear/bin` finden sich die Programme `as` und `ld`.

Das erste Programm verwendet intern den Präprozessor des C-Style Assemblers `cas`, um einige Makros, die der Lesbarkeit des Assembler-Codes dienen bzw. das konditionale Linken ermöglichen, aufzulösen. Es verwendet intern einen Assembler, der gegenüber dem originalen, in [Del02] beschriebenen Assembler um einige Assembler-Anweisungen erweitert wurde und verfügt nur über eine sehr eingeschränkte Aufrufsyntax. `as` überprüft anders als herkömmliche Assembler nur die Syntax, die eigentliche Umwandlung in das Format zum Download auf den Prozessor wird erst von `ld` erzeugt.

Das Programm `ld` agiert als Linker und erzeugt entweder eine downloadfähige Datei oder eine Datei im .mif-Format (letzteres durch die Option `-m`). Es funktioniert im Wesentlichen wie `as`, ist aber geringfügig flexibler im Aufruf.

```
as [-o OBJFILE] ASMFILE
```

Abbildung 1: Aufrufsyntax von `as`

```
/usr/local/spear/bin/ld [-m] [-h] [-c CODESIZE] [-d DATASIZE]
[-o OBJFILE] [ASMFILE]
```

Abbildung 2: Aufrufsyntax von `ld`

2.2 Syntax

Der Assembler erlaubt für Konstanten Ausdrücke wie in C, wobei die Funktionen `lo`, `hi`, `3rd` und `4th` dazu verwendet werden können, einzelne Bytes zu extrahieren.

Identifier müssen aus alphanumerischen Zeichen, Unterstrichen (“_”) und Punkten bestehen, wobei Identifier, die mit einem Punkt beginnen, als lokale Symbole interpretiert werden. Labels werden durch einen Identifier, auf den ein Doppelpunkt folgt, deklariert, also z.B. `foo:`. Kommentare - nur Zeilenkommentare sind möglich - beginnen wie bei Assemblern üblich mit einem Strichpunkt (“`;`”).

`.file`	Beginnt eine neue logische Datei. Wird verwendet, um lokale Labels innerhalb einer physikalischen Datei zu unterscheidbar zu machen.
`.text`	Alle nachfolgenden Daten werden in der `.text` Sektion abgelegt.
`.data`	Alle nachfolgenden Daten werden in der `.data` Sektion abgelegt.
`.rodata`	Alle nachfolgenden Daten werden in der `.rodata` Sektion abgelegt.
`.bss`	Alle nachfolgenden Daten werden in der `.bss` Sektion abgelegt.
`.comm` *name*, *zahl*	Für das Symbol *name* wird in der `.bss` Sektion *zahl* Speicherplatz reserviert.
`.lcomm` *name*, *zahl*	wie `.comm`
`.align` *zahl*	Erhöht den Befehlszeiger so, dass die unteren *zahl* bits auf Null gesetzt werden. Im Datenbereich wird der Speicher mit Nullen aufgefüllt, im Befehlsbereich mit `nop` Befehlen.
`.org` *zahl*	Bewegt den Befehlszeiger zu *zahl*. Im Datenbereich wird der Speicher mit Nullen aufgefüllt, im Befehlsbereich mit `nop` Befehlen. Ein Zurückbewegen des Befehlszeigers ist nicht möglich.
`.skip` *zahl*	Erhöht den Befehlszeiger um *zahl*. Im Datenbereich wird der Speicher mit Nullen aufgefüllt, im Befehlsbereich mit `nop` Befehlen. Ein Zurückbewegen des Befehlszeigers ist nicht möglich.

Tabelle 1: Liste der Assembler-Anweisungen

3 GNU Compiler Collection

3.1 Beschreibung

Die GNU Compiler Collection (GCC) besteht aus Frontends für mehrere Sprachen (u.a. C, C++ und Java) und Backends für die meisten gängigen Prozessoren. Die unter `spear32-gcc` bzw. in `/usr/local/spear32/bin` verfügbare Version beinhaltet das C-Frontend und das SPEAR-Backend.

3.2 Aufruf

Der Compiler erlaubt eine Vielzahl von Optionen, die für alle Backends gültig sind. Für eine Beschreibung dieser generischen Optionen und der Eigenheiten des Compilers (z.B. die erweiterte Syntax für Assembler-Befehle in C) sei auf das Manual verwiesen. Die für den SPEAR spezifischen Optionen des Compilers sind in Tabelle 2 auf der nächsten Seite ersichtlich.

3.3 Datentypen

Der Datentyp `char` ist 8 Bit breit, `short` 16 Bit. Die Breite des Datentyps `int` ist als "natürliche Größe die von der Architektur der Laufzeitumgebung vorgegeben wird" festgelegt und daher 32 Bit breit. Der Datentyp `long` ist 32 Bit breit, der Typ `long long` 64 Bit.

3.4 Registerbelegung

Die von `spear32-gcc` verwendete Registerbelegung ist in Tabelle 3 auf Seite 9 ersichtlich.

Wird Assembler-Code in C-Code eingebunden, so kann das Register r13 als `__tmp_reg__` referenziert werden und sein Wert muss nicht erhalten werden.

Wird C- und Inline-Assembler-Code gemischt, so ist keinesfalls die Erhaltung irgendeines Registerwerts über die C-Code-Teile hinweg garantiert; mittels der Aufrufoption `-ffixed-`*reg* kann der Compiler jedoch angewiesen werden, ein bestimmtes Register nicht zu verwenden.

3.5 Schwächen

- "nested functions" werden nicht unterstützt. Aufgrund der strikten Trennung zwischen Code- und Datenspeicher ist eine Implementation nicht möglich.

`-mmif`	Der Linker wird angewiesen, eine Datei im .mif-Format zu erzeugen
`-mhex`	Der Linker wird angewiesen, eine Datei im Intel hex Format zu erzeugen
`-mlib`	Symbole werden markiert, sodass sie nur nach vorheriger Verwendung dazugelinkt werden
`-mcodesize=`*n*	Nimm *n* als Größe des Codespeichers an (0 bis 0x10000, standardmäßig 0x800)
`-mdatasize=`*n*	Nimm *n* als Größe des Datenspeichers an (0 bis 0x10000, standardmäßig 0x800)
`-mregs=`*n*	Verwende nur die obersten *n* Register (16-32, standardmäßig 32)
`-mmcu=`*mcu*	Erzeuge Code für Prozessor *mcu* (`spear`, `needle` oder `lance`, standardmäßig `spear`). Da alle Prozessoren kompatibel sind, verändert diese Option nichts.
`-minit-stack=`*n*	Verwende *n* als anfängliche Stackadresse (0 bis 0x10000, standardmäßig 0x800)
`-mdeb`	Meldungen zum Debuggen des Compilers werden aktiviert
`-masmsize`	Die Größe der Instruktion wird in die Assembler-Datei ausgegeben (nur zum Debuggen des Compilers relevant)
`-masmrtl`	Eine interne Darstellung des Codes wird in die Assembler-Datei ausgegeben (nur zum Debuggen des Compilers relevant)

Tabelle 2: Optionen für `spear32-gcc`

Einige kleinere, selten auftretende Bugs sind ebenfalls bekannt, an ihrer Behebung wird gearbeitet. Sollte bei der Verwendung des Compilers ein Bug auftreten, wird gebeten, eine entsprechende Beschreibung umgehend an `hausen@gmx.at` zu schicken.

Konkret schlagen 166 Testfälle der GCC Testsuite fehl (bei über 36000 Testfällen ist dies eine Fehlerquote von $<0.5\%$). Etliche der gescheiterten Testfälle sind jedoch als harmlos einzustufen, weil sie auf Grund nicht vorhandener Warnungen, fehlender Optimierungen und ähnlicher Dinge, die die Funktionalität des Compilers nicht einschränken, fehlschlagen.

	Verwendung	für Variablen	Aufrufer sichert
r0	Rückgabewert	JA	JA
r1	Argument 1	JA	JA
r2	Argument 2	JA	JA
r3	Argument 3	JA	JA
r4	Argument 4	JA	JA
r5	temporäres Register	JA	JA
...	...	...	...
r8	temporäres Register	JA	JA
r9	gesichertes Register	JA	NEIN
...	...	...	...
r12	gesichertes Register	JA	NEIN
r13	intern	NEIN	-
r14	Rücksprungadresse	NEIN	NEIN[a]
r15	Rücksprungadresse (Exceptions)	NEIN	-
fpw	temporäres Register	JA	JA
fpx	temporäres Register	JA	JA
fpy	Framezeiger	NEIN	-
fpz	Stackzeiger	NEIN	-

Tabelle 3: Registerbelegung

[a]Ein Aufruf einer Unterfunktion wird als Verwendung des Registers interpretiert, was nach sich zieht, dass es beim Eintritt in die Funktion gesichert wird und wiederhergestellt wird, bevor der eigentliche Rücksprung erfolgt.

4 libc

Im Zuge der Compilertests wurde eine eingeschränkte Version der libc implementiert. In Tabelle 4 sind die derzeit vorhandenen Routinen ersichtlich. Eine nähere Beschreibung findet sich z.B. in [fTI02].

abort	memmove	strcmp	putchar[b]
ffs	bzero	strncmp	fprintf[b]
ffsl	memset	strcpy	vfprintf[b]
ffsll	strcat	stpcpy	fputs[b]
isprint[a]	strncat	strncpy	fputc[b]
bcmp	strspn	strlen	fwrite[b]
memcmp	strcspn	calloc	malloc
memcpy	strpbrk	printf[b]	free
mempcpy	strchr	vprintf[b]	sprintf[c]
bcopy	strrchr	puts[b]	vsprintf[c]

Tabelle 4: Vorhandene Routinen der libc

[a] Funktioniert nur für Werte von 0 bis 256 zuverlässig.

[b] Nur Dummy-Funktionen, die immer einen Fehler zurückliefern.

[c] Die Flags `-` und `#`, sowie die Typen `f`, `e`, `E`, `g` und `G` werden nicht unterstützt. Erweiterungen wie die Größenangabe `ll` ("long long") oder die Typen `C` und `S` sind nicht zulässig.

5 Sections und Dateninitialisierung

Werden die Programme `as` bzw. `ld` verwendet, so kann man die verschiedenen Sections mittels der Pseudo-Mnemonics `.text`, `.data` bzw. `.bss` trennen. Die in `.data` bzw. `.bss` befindlichen Initialisierungen werden automatisch in entsprechende Assemblersequenzen übersetzt, sodass bei einem Reset der Speicher initialisiert wird. Es ist zu beachten, dass auf Grund der Anatomie des SPEAR Speicherzellen die ungleich 0 sind im Schnitt drei Befehle benötigen, um initialisiert werden.

6 Interrupts

Um in C Interrupt-Routinen zu implementieren, können den jeweiligen Funktionen die Attribute `signal` oder `interrupt` wie in Listing 1 zugewiesen werden.

Während Funktionen mit dem Attribut `signal` nicht von anderen Interrupts unterbrochen werden dürfen, können bei Funktionen mit dem Attribut `interrupt` andere Interrupts freigeschalten werden. Wichtig dafür ist jedoch, dass das Processor Control Modul mit der BaseAddress -32 eingebunden wird (also die Adressen 0xFFFFFFE0 bis 0xFFFFFFFF darauf gemappt werden).

Auf das Statusregister des Processor Control Module kann manuell wie in Listing 2 zugegriffen werden. In Listing 3 auf der nächsten Seite wird eine Methode `foo` als Interrupt-Routine mit dem Index 5 registriert.

Um das Programmieren zu erleichtern, sind in der Header-Datei `interrupt.h` einige Makros definiert, die in Tabelle 5 auf der nächsten Seite beschrieben sind.

Listing 1: Zuweisung eines Attributs an eine Funktion

```
void isr() __attribute__((signal));
void isr()
{
  ...
}
```

Listing 2: Zugriff auf das Statusregisters des Processor Control Moduls

```
/* Definitionen fuer das Processor Control Modul */
#define PC_BASE          (-8)
#define PC_STATUS        (*(volatile const int *const)(PC_BASE+0))
#define PC_SAVESTATUS    (*(volatile int *const)(PC_BASE+1))

/* Das Register r15 als Variable ansprechbar machen */
register int* r15 asm("r15");

void my_write_pc_status(int value)
{
  int old_status;
  int *old_r15;

  old_status = PC_SAVESTATUS;
  PC_SAVESTATUS = value;

  old_r15 = r15;
  r15 = &&my_write_pc_status_ret;
  asm("rte");   /* hier wird das Statusregister veraendert */
my_write_pc_status_ret:
```

```
  PC_SAVESTATUS = old_status;
  r15 = old_r15;
}
```

Listing 3: Registrierung einer Interrupt-Routine

```
1 asm("stvec %0, %1" : : "r" (&foo), "i" (5));
```

Name	Argumente	Zweck
SEI, sei	()	setzt das Global Interrupt Enable Flag
CLI, cli	()	löscht das Global Interrupt Enable Flag
STVEC	(FUNC, NUM)	registriert FUNC als Routine für Interrupt NUM
PROTI	(NUM)	setzt Bit NUM im Interrupt Protocol Register
UPROTI	(NUM)	löscht Bit NUM im Interrupt Protocol Register
MASKI	(NUM)	setzt Bit NUM im Interrupt Mask Register
UMASKI	(NUM)	löscht Bit NUM im Interrupt Mask Register
ACKI_L	(BADDR)	bestätigt Interrupt für das Modul mit der Basisadresse BADDR
ACKI_G	(NUM)	löscht Bit NUM im Interrupt Protocol Register und bestätigt Interrupt im Processor Control Module
ACKI	(BADDR, NUM)	ACKI_L(BADDR) und ACKI_G(NUM)

Tabelle 5: Makros für Interrupts

7 Beispiele

Die hier zu findenden Beispiele implementieren eine Endlosschleife, in der zuerst mittels Linear Feedback Shift Pseudozufallswerte in ein Array geschrieben werden, das dann mittels Quicksort sortiert wird.

7.1 C

Listing 4: Quicksort in C

```
#define ANZAHL 64

void quicksort(signed int, signed int);
unsigned int lfsr(void);

unsigned int daten [ANZAHL];

int main()
{
  while (1)
    {
      signed int i;
      for (i = 0; i < ANZAHL; i++)
        {
          daten[i] = lfsr();
        }

      quicksort(0, ANZAHL-1);
    }
}

unsigned int lfsr()
{
  static unsigned int randnum = 1;

  randnum = ((randnum << 1) /* emulating the shift */
             | (((randnum >> 15)
                 ^ (randnum >> 4)
                 ^ (randnum >> 2)
                 ^ (randnum >> 1))
                & 1));

  return randnum;
}

void quicksort(signed int links, signed int rechts)
{
  signed int i;
  signed int j;
  signed int hilf;

  i = links;
  j = rechts;
  hilf = daten[(links+rechts) >> 1];

```

```
  do
    {
      while ((daten[i] < hilf) && (i < rechts))
        {
          i++;
        }
      while ((daten[j] > hilf) && (j > links))
        {
          j--;
        }
      if (i <= j)
        {
          signed int tmp;
          tmp = daten[i];
          daten[i] = daten[j];
          daten[j] = tmp;
          i++;
          j--;
        }
    }
  while (i <= j);

  if (links < j)
    {
      quicksort(links, j);
    }

  if (rechts > i)
    {
      quicksort(i, rechts);
    }
}
```

Während `spear32-gcc -o example.bin example.c` direkt eine downloadfähige Datei `example.bin` erzeugen würde, kann mittels `spear32-gcc -o example.mif -mmif example.c` eine Datei `example.mif` im .mif-Format erzeugt werden.

8 FAQ

8.1 Mein Programm tut nicht was es soll, was tun?

Den fehlerhaften Code mit einer möglichst genauen Beschreibung des Fehlers und eventuellen funktionierenden Alternativen an `hausen@gmx.at` schicken.

8.2 Der Download des Programms funktioniert nicht, was tun?

Wird unter Windows gearbeitet, so ist es unbedingt notwendig, das Programm mittels `cat -B foo.bin > COM2` in einer Cygwin-Shell hinunterzuladen. Eine korrekte Konfiguration der Schnittstelle, z.B. mittels des Programms `mode.com` ist natürlich ebenfalls Voraussetzung.

8.3 Der Compiler erzeugt miserablen Code, geht das nicht besser?

Bitte ein Beispiel mit dem C-Code, dem erzeugten Code und einem Verbesserungsvorschlag an `hausen@gmx.at` schicken. Der Compiler wurde vor allem im Hinblick auf Korrektheit entwickelt und getestet, die Qualität des Codes ist also mit ziemlicher Sicherheit verbesserungsfähig.

8.4 Warum lässt sich mein Code nicht linken, wenn ich `-mlib` verwende?

Zuallererst muss die entsprechende Datei mehrfach angegeben werden, um alle Abhängigkeiten aufzulösen. Fruchtet auch das nicht, wurde wahrscheinlich eine `static` Variable innerhalb einer Funktion deklariert. Dies hat zur Folge, dass das entsprechende Symbol vor seiner Verwendung deklariert wird, aber nur innerhalb der Datei gilt. Dies lässt sich aber einfach dadurch umgehen, dass man die Variable außerhalb der Funktion deklariert.

9 HOWTO

9.1 Wie kann ich meinen Code kleiner machen?

- Das Flag `-mlib` verwenden: mit `spear32-gcc -S -mlib -Os foo.c` Assembler-Datei erstellen, mit `spear32-gcc -o foo.bin foo.s foo.s ...` fertig kompilieren; die .s Datei muss dabei im Allgemeinen mehrere Male angegeben werden, weil nur Funktionen inkludiert werden, die bereits verwendet worden sind. Ebenso sind `static` Variablen innerhalb von Funktionen zu vermeiden, da diese dann im Quelltext vor ihrer Verwendung deklariert werden, aber nur lokal innerhalb der Datei gelten.

- Funktionen als `inline` deklarieren, wenn sie nur an wenigen Stellen aufgerufen werden. Insbesondere bei Treiberfunktionen, die oft nur ein einzelnes Register eines Extension-Moduls lesen oder schreiben kann so einiges eingespart werden.

- Auf Gleitkommaarithmetik verzichten, die entsprechenden Libraries benötigen sehr viel Platz und die benötigte Funktionalität lässt sich oft mit Fixkomma-Operationen ebensogut darstellen. Falls sowohl `float` als auch `double` verwendet werden, nur einen der beiden Typen verwenden, da jeder der Typen seine eigene Library benötigt.

9.2 Wie kann ich Interrupts verwenden?

- eine Funktion - sinnvollerweise ohne Argumente und Rückgabewert - wird mit dem Attribut `signal` oder `interrupt` markiert.

- diese Funktion wird während der Initialisierungsphase des Programms mittels `STVEC` als Interrupt-Handler registriert.

- in der Funktion selbst muss folgender Ablauf eingehalten werden:

  1. Lesen der Daten
  2. Bestätigen des Interrupts beim Modul
  3. Löschen des entsprechenden Bits im Interrupt Protocol Register
  4. Bestätigen des Interrupts beim Processor Control Module
  5. gegebenenfalls andere Interrupts freigeben (natürlich nur, wenn die Funktion mit dem Attribut `interrupt` ausgezeichnet wurde)
  6. Verarbeiten der Daten

Die für die Punkte 2-4 kann, wie in Listing 5, das Makro `ACKI` verwendet werden.

Listing 5: Beispiel eine Interrupt-Routine

```
1  void uart_isr() __attribute__((interrupt));
   void uart_isr()
   {
     char msg = UART_MSG;
5    ACKI(UART_BADDR, UART_INTRNUM);
     sei();
     ...
   }
   ...
10 int main(void)
   {
     STVEC(uart_isr, UART_INTRNUM);
     ...
     sei();
15   ...
   }
```

A Literatur

Literatur

[Del02] Martin Delvai. *Handbuch für SPEAR.* Institut für Technische Informatik, TU Wien, 2002.

[fTI02] Institut für Technische Informatik. *Systemnahes Programmieren.* ubooks - Verlag für Print-on Demand und Buchshop, 2002.

[Puf04] Wolfgang Puffitsch. *cas - A C-style Macro Assembler (for cas 0.2).* Institut für Technische Informatik, TU Wien, 2004.

[Sta01] Richard M. Stallman. *Using and Porting the GNU Compiler Collection (for GCC 3.0).* http://gcc.gnu.org/onlinedocs/, 2001.