Portierung des GDB für die SPEAR 2 Architektur

Ludwig Meier
0526541

25. November 2008

Zusammenfassung

Dokumentation der Portierung des GNU-Debuggers (GDB) für die Spear2-Architektur. Weiters wird die Integration des Spear2-32 Instruction-Level Simulators in den GDB dokumentiert.

Kurze Einführung zur Benutzung der wichtigsten Debugger-Funktionen mit dem integrierten Spear2-32 Simulator als Zielsystem.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Dieses Dokument beschreibt die Portierung des GDB für die Spear2-Architektur und die Integration des Spear2-32 Instruction-Level Simulators in den GDB. Der GDB-Port entstand als Teil eines Praktikums an der TU-Wien. Der Spear2-32 Instruction-Level Simulator existierte bereits und wurde im Rahmen des selben Praktikums in den GDB integriert.

Es werden die Details der GDB-Portierung beschrieben. Außerdem wird der Übersetzungsvorgang der GDB-Quellen dokumentiert.

Dieses Dokument gibt weiters eine kurze Einführung zur Verwendung des GDB mit dem Spear2-Simulator als Zielsystem und dem Data Display Debugger (DDD) als Frontend.

Nicht Teil des Praktikums (und dieses Dokuments) war eine Anbindung des GDB an reale Hardware. Dazu fehlt derzeit noch ein GDB Remote-Stub.

Kapitel 2

Anwender-Dokumentation

In diesem Kapitel wird die Verwendung der grundsätzlichen Funktionen des GDB beschrieben. Als Zielsystem wird der integrierte Spear2-Simulator verwendet.

2.1 Verwendung des DDD

Da meiner Ansicht nach das Debuggen mit dem GDB Kommandozeilen-Interface etwas mühsam ist, wird in diesem Abschnitt nicht das GDB Kommandozeilen-Interface direkt, sondern stattdessen das grafische Frontend Data Display Debugger (DDD), verwendet.

2.1.1 Voraussetzungen

Für diesen Abschnitt wird angenommen, dass der Programmcode mit dem Spear32-gcc in eine ELF-Datei übersetzt wurde. Diesen findet man unter [http://sourceforge.net/projects/spear2-tools/].

2.1.2 Anpassung der VMA

Der Spear32-gcc trägt als VMA der .text Sektion derzeit die Adresse 0x80000000 ein. Der GDB in Kombination mit dem Simulator erwartet hier allerdings die Adresse 0x0. Daher sollte man vor dem Laden der ELF-Datei diese Adresse anpassen.

```
objcopy --change-section-vma .text=0 old.elf new.elf
```

Im GDB lädt man dann die Datei new.elf.

2.1.3 Konfiguration des DDD

Im folgenden wird der DDD zum Debuggen eines Spear2-Programms, das durch den integrierten Simulator ausgeführt wird, konfiguriert.

Start des DDD Damit der DDD den Spear2-GDB verwendet, wird der Pfad zu diesem in der Kommandozeile übergeben.
`ddd --debugger /path-to-spear2-gdb/gdb`.

Programm auswählen Dann mit dem Menüpunkt `File->Open Program...` die ELF-Datei ausgewählt, die den zu untersuchenden Programmcode enthält.

Speicher der Extension-Module Um den Speicherbereich zur Kommunikation mit den, im Simulator integrierten, Extension-Modulen beobachten zu können, wählt man am besten den Menüpunkt `Data->Memory...`. Dort wählt man zur Anzeige des Display-Speicher die Einstellungen:
Examine **2 hex words(4)** from **0xfffffa0**
und bestätigt das Ganze mit der Schaltfläche `Display`.

Für den Sysctrl-Speicherbereich wiederholt man die ganze Prozedur, verwendet aber jetzt die Einstellungen:
Examine **6 hex words(4)** from **0xffffffe0**

Alternativ kann man in der GDB-Eingabeaufforderung (unterster Teil des DDD) folgendes eingeben:

```
graph display ‘x /2xw 0xffffffa0‘
graph display ‘x /6xw 0xffffffe0‘
```

Simulator Als nächstes teilen wir dem GDB mit, dass als Zielsystem der integrierte Spear2-Simulator verwendet werden soll. Das geschieht durch Eingabe von `target sim` in der GDB-Eingabeaufforderung. Der GDB sollte mit `Connected to the simulator.` antworten. Ein anschließendes `load` Kommando überträgt dann das Programm aus der ELF-Datei in den Speicher des Simulators.

Haltepunkte Als nächstes kann man an Programmstellen die interessieren, Haltepunkte platzieren. Beispielsweise am Anfang der Funktion main durch ein `b main` oder an der ersten Instruktion die überhaupt ausgeführt wird mittels `b*0`.

Programmausführung Die Programmausführung selbst wird durch Eingabe von `run` gestartet.

Kapitel 3

Build-Vorgang

3.1 Voraussetzungen

Benötigt wird der Quellcode für GDB Version 6.6 erhältlich unter [ftp://sourceware.org/pub/gdb/releases/gdb-6.6.tar.bz2]

Des weiteren den Patch für die Spear2-Architektur. Dieser hat wahrscheinlich den Namen `gdb-6.6-spear2.patch`.

3.1.1 Übersetzen

```
tar -xvjf gdb-6.6.tar.bz2
cd gdb-6.6
bzcat gdb-6.6-spear2.patch.bz2 | patch -p1
./configure --target=spear2
make
```

Die neue ausführbare Datei liegt in `gdb-6.6/gdb/gdb`.

Kapitel 4

Simulator

Dieses Kapitel beschreibt die Integration des Simulators in den GDB.

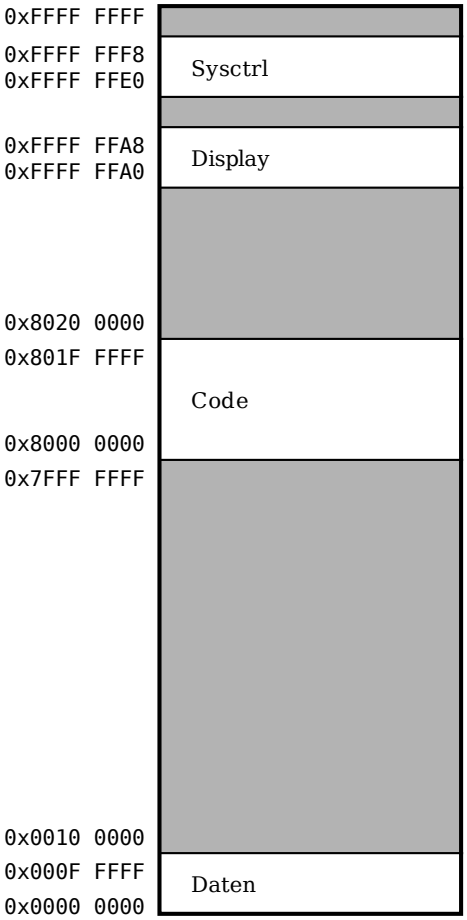
4.1 Integration des Simulators

Der bereits vorhandene Instruction-Level Simulator für die Spear2 32-Bit Architektur wurde in den GDB integriert. Damit der GDB den Simulator als Zielsystem verwendet ist das Kommando `target sim` zu verwenden.

4.2 Adressraum

Der Simulator verwaltet intern Arrays fixer Größe in denen die Daten des Simulierten-Systems abgelegt werden. Die Größe der Bereiche für den Instruktions- und für den Datenspeicher lässt sich zur Übersetzungszeit des GDB durch die Präprozessor-Konstanten `SPEAR2_SIM_CODESIZE` und `SPEAR2_SIM_MEMSIZE` festlegen. Wobei zu beachten ist, dass die Angabe bei `SPEAR2_SIM_CODESIZE` als Anzahl von 16-Bit Instruktionen interpretiert wird. `SPEAR2_SIM_MEMSIZE` wird, wie zu erwarten, in Bytes spezifiziert.

Die Spear2 Architektur verwendet getrennte Adressbereiche für Code und Daten (Beschrieben in [1], Seite 3). Der GDB ist für Systeme mit gemeinsamen Code- und Datenspeicher konzipiert. Um trotzdem Zugriffe auf den Code-Speicher des Simulators zu ermöglichen, wird dieser im Adressraum des GDB ab Adresse 0x8000 0000 abgebildet.


0xFFFF FFFF
0xFFFF FFF8
0xFFFF FFE0
Sysctrl
0xFFFF FFA8
0xFFFF FFA0
Display
0x8020 0000
0x801F FFFF
Code
0x8000 0000
0x7FFF FFFF
0x0010 0000
0x000F FFFF
Daten
0x0000 0000

4.3 Plugins

Der Spear2 Prozessor ist durch Extension-Module erweiterbar. Das Erweiterungs-Konzept ist in [1], S.7ff dokumentiert.

Um Hardware-Erweiterungen im Simulator abbilden zu können (bzw. um potentielle Hardwareerweiterungen vor deren Realisierung testen zu können), enthält dieser ein Plugin-System.

Ein Plugin ist ein shared object, wie sie von der GCC bei Angabe der Option `-shared` erzeugt werden. Zusätzlich muss ein Plugin noch Methoden mit den folgenden Signaturen implementieren:

```
void mapped_to(uint32_t* start, uint32_t* size);
uint8_t get_mem(uint32_t offset);
void set_mem(uint32_t offset, uint8_t value);
void tick();

//Optional part. Required for Interrupt-Support only.
uint8_t* get_status_address();
```

4.3.1 Plugin Schnittstelle

mapped_to Liefert die Sartadresse des Speicherbereichs über den Daten mit diesem Modul ausgetauscht werden. Außerdem wird die Größe dieses Speicherbereichs zurückgegeben.

get_mem Liefert ein Byte aus dem Kommunikations-Speicherbereich. Die übergebene Adresse ist als Offset zum Start des Speicherbereichs zu verstehen.

set_mem Wird aufgerufen um den Wert eines Bytes im Kommunikations-Speicherbereich zu setzen. Die übergebene Adresse ist als Offset zum Start des Speicherbereichs zu verstehen.

tick Diese Funktion wird, für jeden simulierten Taktzyklus des Zielsystems, für jedes Plugin ein mal aufgerufen. Ein Plugin kann in dieser Funktion den Code implementieren, der die Funktionalität des Extension-Moduls nachbildet.

get_status_address (Optional) Diese Methode muss nur implementiert werden, wenn das simulierte Erweiterungsmodul Interrupts auslösen können soll. Diese Methode liefert die Adresse, and der das Plugin-Modul die Informationen des Status-Registers hält. Der Simulator greift auf diese Speicherstelle ausschließlich lesend zu.

Notwendig ist diese spezielle Behandlung des Status-Registers, weil der Simulator in jedem Taktzyklus prüft ob ein Interrupt augelöst wurde (INT-Bit im Status-Register gesetzt). Ein Aufruf von `get_mem(0)` ist in diesem Fall nicht möglich, da manche Module beim Lesen des Status-Registers gleichzeitig noch andere Aktionen durchführen (z.B. das miniUART-Modul löscht das EF-Flag im Status_C-Register wenn das Status-Register gelesen wird).

Interrupts

Plugins der Spear2-Architektur können mit einem Interrupt-Eingang des Prozessor-Kerns verbunden sein. Dieses Zuordnung erfolgt statisch im VHDL-Code.

Damit der Spear2-Simulator eine solche Verbindung für ein Plugin-Modul annimmt, ist eine Datei mit dem Namen des Plugins und der zusätzlichen Erweiterung `.int` zu erstellen. Diese Datei muss die zu verwendende Interrupt-Nummer enthalten (und nur diese, keine Whitespaces, keine Kommentare).

Wenn ein Plugin einen Interrupt auslösen will, setzt es das INT-Bit in seinem Status-Register. Der Spear2-Simulator startet dann die Ausfuhrung der entsprechenden ISR. Wenn der Interrupt durch setzten des INTA-Bits im Config-Register bestätigt wird, hat die Plugin-Implementierung dafür zu sorgen, dass das INTA- und das INT-Bit rückgesetzt werden. Eine detailiertere Beschreibung des Interrupt-Vorgangs findet sich in [6], Kapitel 3.1.

Beispiel Nimmt man also an, dass ein Plugin namens `~/.spear32-sim/libminiUART.so` existiert, so hätte die Entsprechende Interrupt-Mapping Datei den Namen `~/.spear32-sim/libminiUART.so.int` mit dem Inhalt `11`.

Bem Laden des Plugins wird die erstellte Interrupt-Zuordnung ausgegeben.

```
Plugin: /home/user/.spear32-sim/libminiUART.so, 0xFFFFFF80-FFFFFF97,
Interrupt 11
```

Listing 4.1: Beispiel für eine Interrupt-Zuordnung. Aus den Spear2-Quellen zur Laborübung Hardware-Software Codesign im Sommersemester 2007.

```
speari.interruptin <= (others => '0');
speari.interruptin(14) <= syscexto.intreq;
speari.interruptin(13) <= progexto.intreq;
--     speari.interruptin(12) <= protexto.intreq;
speari.interruptin(11) <= miniUARTexto.intreq;
```

4.3.2 Beispiel-Plugin

Es folgt der Code für ein einfaches Beispiel-Plugin. Dieses führt eine Multiplikation mit 2 aus.

Das Beispiel-Plugin hat den Namen `plugin`. Für reale Plugins sollte man jedes Vorkommen von `plugin` in diesem Abschnitt mit einer sprechenderen Bezeichnung ersetzen.

Unbedingt anzupassen ist für jedes Plugin die Präprozessor-Konstante `MEM_START`. Hier ist darauf zu achten, dass niemals 2 Plugins den selben Adressraum belegen. Außerdem darf ein Plugin nicht mit dem Adressraum einer der im Simulator fix eingebauten Erweiterungen kollidieren. Siehe Abbildung 4.1.

plugin.c

```
#include <stdio.h>
#include <stdint.h>

#define MEM_START ((uint32_t)-1024);
static uint8_t mymem[20];

void mapped_to(uint32_t* start, uint32_t* size) {
    *start = MEM_START;
    *size = sizeof(mymem);
}

uint8_t get_mem(uint32_t offset) {
    return mymem[offset];
}

void set_mem(uint32_t offset, uint8_t value) {
    mymem[offset] = value;
}

void tick() {
    *(uint32_t*)&mymem[4] = *(uint32_t*)&mymem[0] * 2;
}
```

Übersetzen

```
gcc plugin.c -shared -fPIC -o libplugin.so -Wl,-soname,libplugin.so
```

Installation

Damit der Simulator ein Plugin findet muss es im Verzeichnis `~/.spear32-sim` abgelegt werden.

```
cp ./libplugin.so ~/.spear32-sim/
```

Verwendung

Das Plugin wird nun automatisch von dem im GDB integrierten Simulator geladen. Die geladenen Plugins werden dabei in der GDB-Kommandozeile ausge-

geben. Das sieht dann beispielsweise so aus:

```
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=spear2"...
(gdb) target sim
PLUGIN: /home/user/.spear32-sim/libplugin.so, 0xFFFFFC00-FFFFFC13
PLUGIN: /home/user/.spear32-sim/libmul2.so, 0xFFFFFFB0-FFFFFFC3
Connected to the simulator.
(gdb)
```

In diesem Beispiel wurden 2 Plugins geladen. `libplugin.so` und `libmul2.so`. Neben jedem Plugin wird der Speicherbereich, über den mit diesem Modul kommuniziert werden kann, ausgegeben.

Zu beachten ist, dass auf den Speicherbereich eines Moduls erst zugegriffen werden kann, nachdem die Programmausführung gestartet worden ist. Sollte man also im GDB Initialisierungen im Plugin-Speicherbereich vornehmen wollen, ist zuvor folgendes einzutippen:

```
b*0
run
```

Kapitel 5

Portierung

Dieses Kapitel beschreibt Details der Portierung des GDB für die Spear2-Architektur.

5.1 Frame

Damit der GDB den Aufrufstapel anzeigen kann, benötigt er Funktionen, die den Inhalt des Stacks analysieren und daraus die Rücksprungadresse zu der aufrufenden Funktion extrahieren. Weiters muss der Teil des Stacks, der der aufrufenden Funktion zuzuordnen ist, identifiziert werden. Diese Funktionen analysieren den Maschinencode und sind daher architekturspezifisch.

5.1.1 Spear2-GCC

Der Frame-Unwind Code im Spear2-GDB ist für Code wie er von der Spear2-GCC erzeugt wird ausgelegt. Wie Code, der von der Spear2-GCC erzeugt wurde, die Register verwendet ist in [2], S. 9 dokumentiert. Es folgt der Ausschnitt der für den Frame-Unwinder relevant ist.

	Verwendung	für Variablen	Aufrufer sichert
r13	intern	NEIN	-
r14	Rücksprungadresse	NEIN	NEIN[a]
r15	Rücksprungadresse (Exceptions)	NEIN	-
fpy	Framezeiger	NEIN	-
fpz	Stackzeiger	NEIN	-

Tabelle 5.1: 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.

Prolog

Der Anfang, soweit er für den Frame-Unwinder relevant ist, eines typischen Funktions-Prologs sieht folgendermaßen aus:

```
147: 0101111111111110; % fu3: stfpz_dec r14, -1 %
148: 0000111100000111; % ldli r7, -16 %
149: 1111000001111101; % ldw r13, r7 %
14a: 0101111111111101; % stfpz_dec r13, -1 %
14b: 0000111101000111; % ldli r7, -12 %
14c: 1111000001111101; % ldw r13, r7 %
14d: 0000111100000111; % ldli r7, -16 %
14e: 1111010101111101; % stw r13, r7 %
14f: 0000111101000111; % ldli r7, -12 %
150: 1111000001111101; % ldw r13, r7 %
151: 1010101111001101; % addi r13, -4 %
152: 1111010101111101; % stw r13, r7 %
```

0x147 Sichert die Rücksprung-Adresse aus Register 14 auf dem Stack. Das wird nur gemacht wenn die Funktion weitere Unterfunktionsaufrufe enthält.

0x148 Lädt den Wert -16 (0xfffffff0) in Register 7. Das ist Offset 0x10 im Speicherbereich des Processor-Control Moduls, Also die Speicherstelle von FPY. Dieses enthält nach Tabelle 5.1 den Framezeiger.

0x149 Lädt den aktuellen Wert von FPY (Framezeiger).

0x14a Sichert FPY (Framezeiger) auf den Stack. Diese Instruktion wird recht häufig am Funktionsanfang ausgeführt. Je nachdem ob die Rücksprung-Adresse gesichert wird als 3. oder 4. Instruktion der Funktion. Daher verwendet der Spear2 Frame-Unwinder diese Instruktion als Ausgangspunkt bei der Suche nach dem Funktionsanfang.

0x14b-0x14e Setzt den Framezeiger (FPY) auf den aktuellen Wert des Stack-Pointers (FPY).

0x14f-0x152 Reserviert Speicherplatz für die lokalen Variablen am Stack.

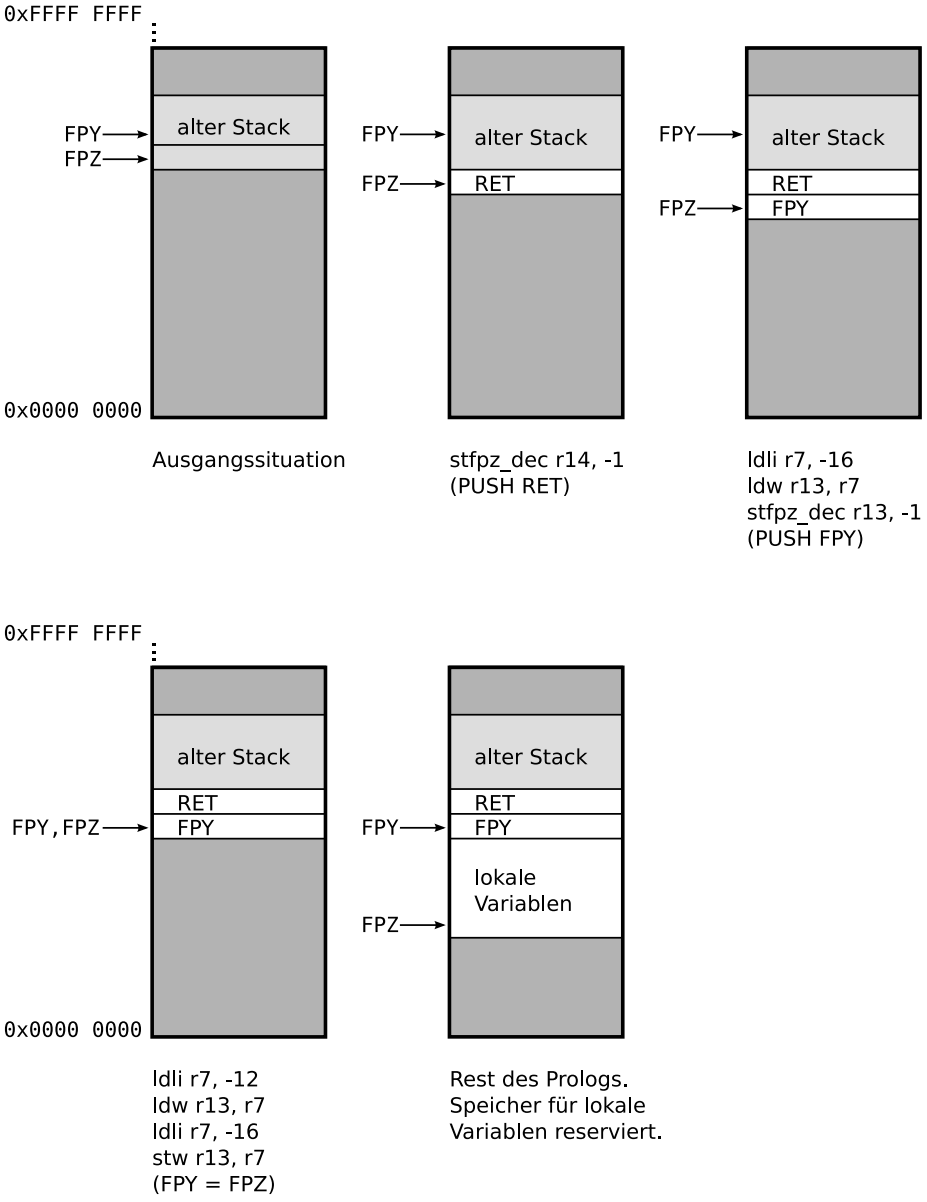


Abbildung 5.1: Stack-Initialisierung im Prolog

Frame-Unwinder

Der Frame-Unwinder untersucht den Maschinencode und den Speicher des Zielsystems, um die Registerwerte, Frame- und Stack-Pointer der aufrufenden Funktion zu ermitteln. Die Kernaufgaben werden von den beiden Funktionen `spear2_frame_prev_register` und `spear2_frame_this_id` erfüllt.

spear2_frame_prev_register Diese Funktion ermittelt den Wert eines Registers in der aufrufenden Funktion. Dazu muss sie herausfinden, wo sich der

Wert dieses Registers, wie ihn die aufrufende Funktion erwartet, befindet. Die wahrscheinlichsten Szenarien sind, dass sich der Wert nach wie vor im Register befindet oder aber, dass der Wert auf den Stack ausgelagert wurde.

Diese Funktion ist derzeit nur für den PC implementiert. Für alle anderen Register wird immer der aktuelle Wert im Prozessorregister geliefert.

spear2_frame_this_id Ermittelt den Wert des Base-Pointers vor dem Funktionsaufruf und die Adresse, an der der Funktionsaufruf erfolgt ist.

5.2 BFD

Die BFD-Bibliothek stellt Funktionen zum Erkennen des Typs von Objektdateien, und zum Laden dieser Dateien zur Verfügung.

Die BFD-Bibliothek [5] wurde um Funktionen zum Einlesen des Binärformats, das die Spear2-GCC erzeugt, erweitert. Diesem Dateiformat gebe ich im folgenden den Namen Spear2bin.

Spear2bin-Dateien beinhalten ausschließlich Daten die in den Instruktionsspeicher des Zielsystems geladen werden müssen. Als Übertragungsmedium vom PC zum Zielsystem kommt derzeit eine serielle Verbindung zum Einsatz. Auf der Zielsystemseite werden sie von einem miniUART genannten Teil entgegengenommen. Dieses ist in [3] dokumentiert. Die Interpretation der Daten erfolgt durch den Code im Boot-Rom der Zielplattform. Dieser ist in [3], Seite 18ff. abgedruckt.

Der Aufbau von Spear2bin-Dateien lässt sich aus dem Boot-Rom Code ableiten. Der grundsätzliche Aufbau ähnelt einem Subset des Intel-Hex Formats [4], mit dem entscheidenden Unterschied, dass die Daten binär und nicht als ASCII-Text abgelegt sind. Außerdem werden der ':' am Zeilenanfang, die Record-Länge und die Checksumme eingespart. Konkret werden die Record-Typen 0x00-'Data Record' und 0x04-'Extended Linear Address Record' verwendet. Weiters ist in Spear2bin-Dateien zu beachten, dass Adressangaben sich immer auf Instruktionen (16-Bit) beziehen.

5.2.1 Spear2bin

Mit Spear2bin bezeichne ich das spezielle Binärformat, das die Spear2-GCC erzeugt.

In den Spear2bin-Dateien sind die Daten auf 16-Bit Blöcke aufgeteilt. Beim Einlesen erzeugt das BFD-Frontend für jeden dieser Blöcke eine eigene BFD-Sektion. BFD-Sektionen beinhalten 2 Adressangaben. Eine 'Virtual Memory Address' (VMA) und eine 'Load Memory Address' (LMA). Die VMA spezifiziert die Adresse, an der die Daten zur Laufzeit des Programms angesprochen werden können. Außerdem wird sie in anderen Objektformaten als Basis für eventuelle Symbol-Relocations herangezogen, nicht jedoch in Spear2bin-Dateien, da diese keine Änderung der Ladeadresse unterstützen. Die LMA wird bei der Übertragung der Daten in den Speicher des Simulators verwendet.

Die beim Einlesen erzeugten Sektionen bekommen als VMA-Adresse die in der Spear2bin-Datei angegebene Adresse (nach einer Multiplikation mit 2, zur Umrechnug von 16-Bit Adressierung auf 8-Bit Adressierung) zugewiesen. Da-

mit beim Übertragen der Daten in den Simulator diese im Instruktionsspeicher landen, wird als LMA = VMA + 0x80000000 gesetzt.

5.2.2 ELF

BFD in der Ausprägung für die Spear2-Architektur unterstützt auch ELF-Objektdateien. Hierfür waren aber nur minimale Konfigurationsanpassungen nötig. Der Code zum Einlesen von ELF-Dateien ist in der BFD-Bibliothek bereits vorhanden und musste nicht geändert werden.

Nach dem Laden der Symbole aus der ELF-Datei, werden die Adressen aller Symbole die zur `.text`-Sektion gehören halbiert. Das ist notwendig, weil der Inhalt der `.text`-Sektion in den Codespeicher des Spear2-Systems geladen wird und dessen Adressierung in 16-Bit Schritten erfolgt.

5.3 libopcodes

Libopcodes beinhaltet die Disassembler-Funktionalität. Deren Sourcecode liegt im Verzeichnis gdb-6.6/opcodes. Der Disassembler für die Spear2-Architektur ist in gdb-6.6/opcodes/spear2-dis.c implementiert.

Kern des Disassemblers ist die Funktion `print_insn_spear2`. Als Parameter erhält sie eine Adresse im Instruktions-Speicher. Auch hier wird der Instruktionsspeicher, wie vom Spear2-PC bekannt nicht Byte weise, sondern auf 16-Bit Basis adressiert.

Bei jedem Aufruf von `print_insn_spear2` wird genau eine Instruktion bearbeitet. Für jeden Befehl werden dessen Assembler-Instruktion und die Werte eventueller Parameter ausgegeben. Zurückgegeben wird von der Funktion nicht wie üblicherweise die Anzahl verarbeiteter Bytes, sondern die Anzahl verarbeiteter 16-Bit Instruktionen, da die Adressierung der Instruktionen ebenfalls in 16-Bit Auflösung erfolgt.

Kapitel 6

Zielsystem Unterstützung

In diesem Abschnitt ist dokumentiert, in welcher Weise die vom GDB benötigten Funktionalitäten im Zielsystem, implementiert sind.

6.1 Übersicht

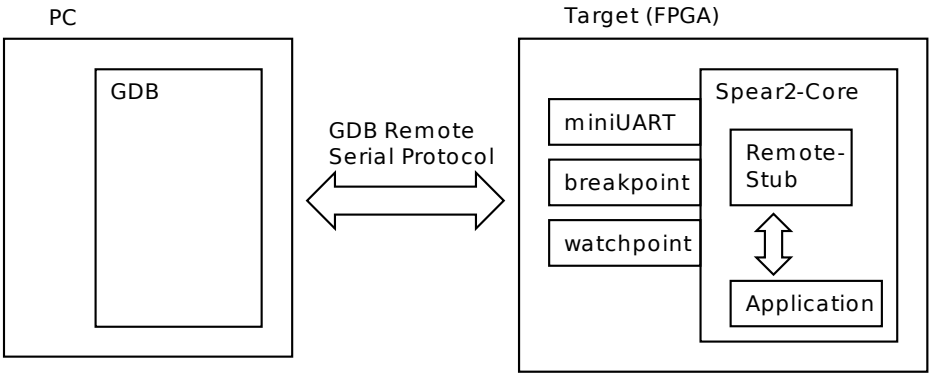


Abbildung 6.1: Übersicht Remote-Debugging

6.2 Hardware

Die zum Debuggen notwendigen Hardware-Funktionen werden durch mehrere Spear2 Extension-Module realisiert. Die beteiligten Module werden in den folgenden Unterabschnitten genauer beschrieben.

Alle zur Debug-Unterstützung notwendigen Extension-Module teilen sich einen Interrupt-Kanal (derzeit Interrupt 11) des Prozessors.

6.2.1 miniUART

Das miniUART-Modul stellt die serielle Schnittstelle bereit, über die die Kommunikation zwischen Debug-Host und Debug-Target erfolgt. Das miniUART Modul ist in [3] dokumentiert.

6.2.2 Breakpoint

Dieses Modul erfüllt zwei verschiedene Funktionen. Anhalten bei Ausführung von Instruktionen auf bestimmten Adressen und Hardware-Unterstützung für Single-Stepping.

Breakpoints auf Code-Adressen

Erstens vergleicht es den aktuellen Program-Counter (PC) mit den Werten einer Liste von Breakpoint-Adressen. Wenn der PC mit einem dieser Werte, wird dem Prozessorkern nicht die im Instruction-RAM liegende Instruktion zur Ausführung übergeben, stattdessen wird eine `TRAP 0` Instruktion generiert. Diese Trap-Instruktion bringt bei einer geeignet initialisierten Exception-Vector-Tabelle Code des GDB-Stubs zur Ausführung.

Eine naheliegende Lösung zur Übergabe der Prozessor-Kontrolle an den GDB-Stub wäre die Verwendung eines der Interrupt-Eingänge des Prozessors. Allerdings werden nach dem Setzten eines Interrupt-Eingangs noch einige Instruktionen ausgeführt, bis die Kontrolle an die ISR übergeben wird. Durch den Eingriff in den Instruktions-Daten Pfad kann die Programmausführung unmittelbar bei der Instruktion an der Breakpoint-Adresse angehalten werden.

Single-Stepping

Ausserdem kann es einen Interrupt nach einer programmierbaren Anzahl von Taktzyklen auslösen. Diese Eigenschaft verwendet der GDB-Stub zur schrittweisen Ausführung von Instruktionen. Dabei wird der Zähler vom GDB-Stub so Programmiert, dass nach dem Verlassen des Stub-Codes noch genau eine Instruktion des Anwendungsprogramms ausgeführt wird. Der nachfolgend generierte Interrupt springt dann erneut in den GDB-Stub.

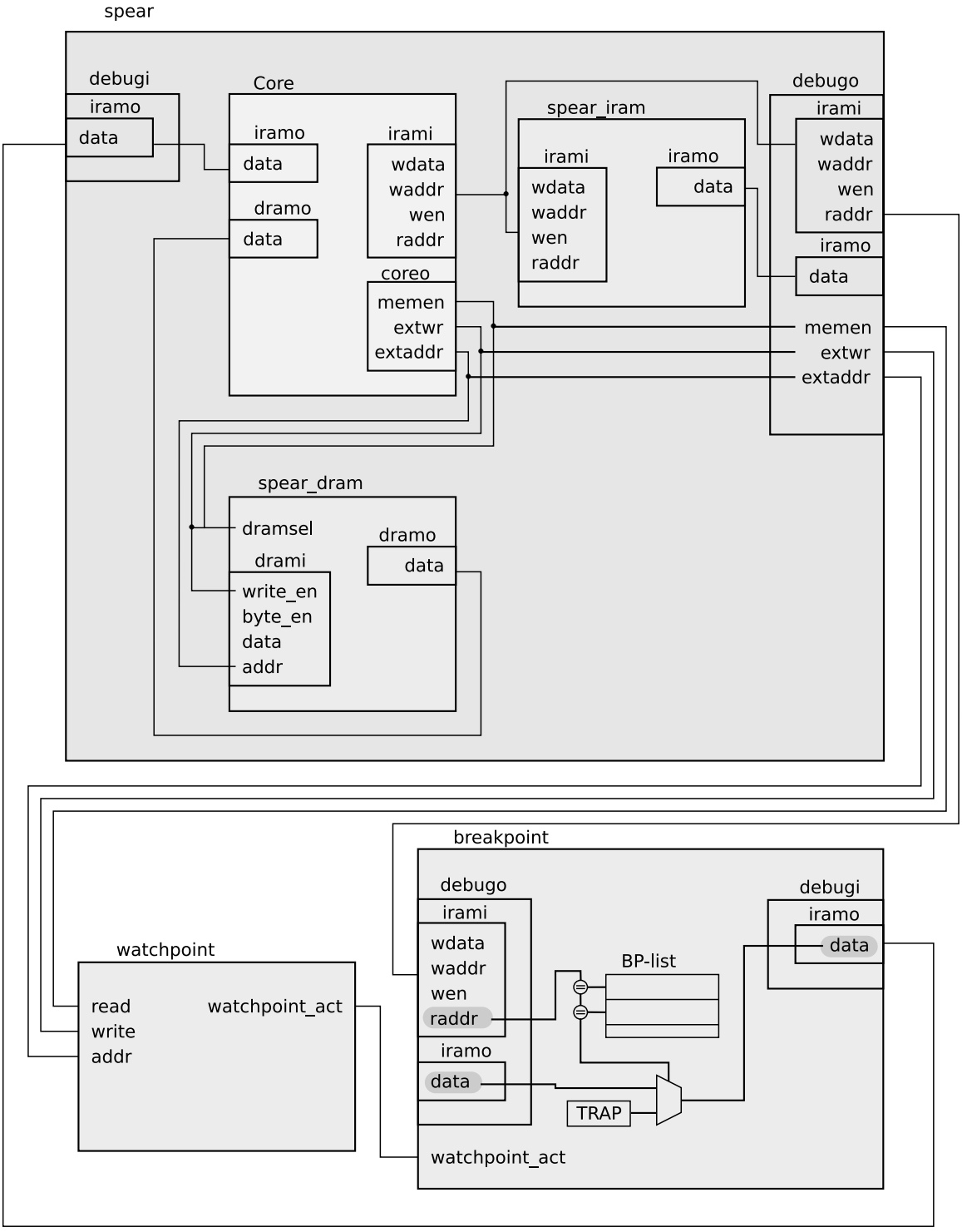


Abbildung 6.2: Integration des breakpoint-Moduls

Register

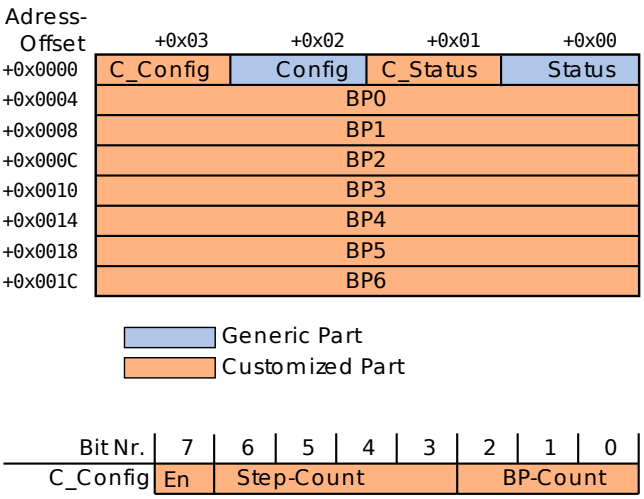


Abbildung 6.3: Interface des breakpoint-Moduls

En: (Reset = 0x0) Bei einem Wert von 0 sind die Register BP0..6 deaktiviert.

BP-Count: (Reset = 0x0) Anzahl gültiger Werte in den Registern BP0 bis BP6. Es sind nur jede BP-Register aktiviert, deren Nummer kleiner ist als BP-Count. Wenn BP-Count 0 ist, sind alle BP-Register deaktiviert.

Step-Count: (Reset = 0x0) Wenn dieses Register einen Wert größer als 0 enthält, wierd dieser bei Ausführung jedes Maschinenbefehls dekrementiert. Beim Erreichen von 0 wird ein Interrupt ausgelöst. Solange Step-Count größer als 0 ist, sind die Register BP0 bis BP6 deaktiviert.

BP0..6: (Reset = 0x0) Nur aktiv wenn En=1. Jedes dieser 7 Register kann eine Breakpoint-Adresse beinhalten. Wenn der Wert des Program-Counter (PC) mit einem der Werte aus den Registern BP0 bis BP6 übereinstimmt wird beim Erreichen dieser Instruktion ein `Trap 0` ausgeführt.

6.2.3 Watchpoint

Durch das Watchpoint-Modul kann man Bereiche im Daten-Speicher auf Lese- und/oder Schreibzugriffe überwachen.

Die zu überwachenden Bereiche werden durch Angabe einer Adresse und einer Maske der zu vergeichenden Bits spezifiziert. Wenn Adressbereiche, die sich nicht direkt in eine entsprechende Adresse/Masken kombination übersetzten lassen, überwacht werden müssen, ist ein entsprechend größerer Speicherbereich zu überwachen und beim Eintreten eines Watchpoint-Ereignisses durch Vergleich mit der in `Access-Adddr` gespeicherten Adresse zu ermitteln, ob dieser Speicherzugriff tasächlich relevant war.

Wenn das Modul einen Speicherzugriff, für den ein Watchpoint definiert ist, erkennt, setzt es seinen Interrupt-Ausgang. In der aktuellen Konfiguration wird die Interrupt-Leitung des Watchpoint-Moduls nicht mit einem Interrupt-Eingang des Prozessors verbunden. Stattdessen wird er mit dem Eingang watchpoint_act des Breakpoint-Moduls verbunden. Das Breakpoint-Modul wiederum generiert beim Aktivierung dieses Signals eine Trap-Instruktion, infolge der die Programm-Kontrolle an den GDB-Stub übergeben wird. Diese Konfiguration ermöglicht es die Programmausführung unmittelbar nach dem interessierenden Speicherzugriff anzuhalten. Wenn man das Watchpoint-Modul direkt mit einem Interrupt-Eingang des Prozessors verbunden würde, würden zwischen dem Speicherzugriff, der das Watchpoint-Modul auslöst und der Übergabe der Programm-Kontrolle an den GDB-Stub noch einige Instruktionen aus dem Anwendugsprogramm ausgeführt werden.

Register

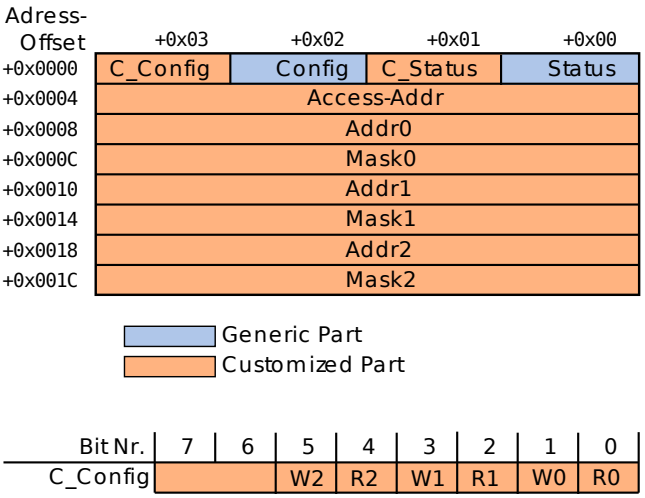


Abbildung 6.4: Interface des watchpoint-Moduls

R0..2: (Reset = 0x0) Aktiviert den entsprechenden Watchpoint-Eintrag bei Lesezugriffen.

W0..2: (Reset = 0x0) Aktiviert den entsprechenden Watchpoint-Eintrag bei Schreibzugriffen.

Access-Addr (Reset = 0x0) Wenn das Watchpoint-Modul einen Daten-Zugriff erkennt, auf den eine der Adresse/Maske Kombinationen zutrifft, wird die Adresse auf die zugegriffen wurde in diesem Register gespeichert. Sie bleibt bis zum nächsten mal ein Watchpoint-Eintrag aktiviert wird erhalten.

Addr0..2: (Reset = 0x0) Adressen die das Watchpoint-Modul auf Zugriffe Überwachen soll. Beim Adressvergleich werden nur jene Bits berücksichtigt, die im ensprechenden Mask-Register auf 0 gesetzt sind.

Mask0..2: (Reset = 0x0) Maske der Bits, die beim Addressvergleich berücksichtigt werden. Wenn ein Bit in der Maske auf 1 gesetzt ist, wird es beim Addressvergleich ignoriert. Ein Eintrag von 0x0 im Mask-Register führt also dazu, dass genau 1 Byte auf Zugriffe überwacht wird. Wenn alle Bits auf 1 gesetzt sind, wird der entsprechende Watchpoint-Eintrag durch jeden Speicherzugriff auf eine beliebiege Adresse aktiviert.

6.3 Remote-Stub

Beim Remote-Stub handelt es sich um eine Software-Komponenete, die das GDB-Remote-Serial Protokoll implementiert. Dieses Protokoll ist in [7], Appendix D, dokumentiert.

Der Remote-Stub wird vom Zielsystem ausgeführt, auf dem auch das zu untersuchende Programm läuft. Damit Programm und Remote-Stub quasi gleichzeitig ausgeführt werden können, ist der Remote-Stub vollständig in Form eines Interrupt-Handlers implementiert. Generiert werden die Interrupts durch die Extension-Module `miniUART`, `breakpoint` und `watchpoint`.

6.3.1 Belegte Ressourcen

In diesem Abschnitt werden die vom Remote-Stub verwendeten Hardware-Resourcen beschrieben. Diese stehen der zu untersuchenden Anwendung nicht mehr zur Verfügung. Unter umständen ist die Anwendung also in manchen Punkten anzupassen um mit dem Debugger untersucht werden zu können.

FPX

Der Remote-Stub verwendet zum Sichern der Registerinhalte das FPX-Register als Zeiger zum Ziel-Speicher. Zur korrekten Funktion ist der Remote-Stub darauf angewiesen, dass der Inhalt des FPX-Registers **nie** verändert wird!

Eintrag 0 der Exception Vektor Tabelle

Der Remote-Stub verwendet den Eintrag 0 der Exception Vektor Tabelle [1], Abshnitt 2.3. Wenn ein Breakpoint bzw. ein Watchpoint ausgelöst wird, wird durch eine vom Breakpoint-Modul generierte `trap 0` Instruktion die Kontrolle an den Remote-Stub übergeben.

Interrup 11

Das zur Seriellen-Datenübertragung verwendete miniUART-Modul und das Breakpoint-Modul benötigen einen Interrupt-Eingang. Derzeit teilen sich diese Module den Interrup-Eingang 11 des Spear-Prozessors. Eine Anwendung darf selbstverständlich den entsprechenden Eintrag in der Exception Vector Tabelle nicht verändern.

6.3.2 Adressraum-Aufteilung

Der Remote-Stub muss sich den vorhanden Adressraum mit der zu untersuchenden Anwendung teilen. Das gilt sowohl für den Instruktions- als auch für den Daten-Speicher.

Instruktions-Speicher

In bisherigen Konfigurationen des SPEAR-Prozessors wurde zwischen dem Boot-ROM und dem Befehlsspeicher umgeschalten. Die Umschaltung erfolgte über `Instr-Src` im Customized-Config Register des Programmer Modules [1], S.13. Wenn man bei der SPEAR Konfiguration die Konstante `GDB_MODE_C` setzt, sind beide Instruktionsspeicher immer ansprechbar und das `Instr-Src` Konfigurationsbit ohne Wirkung.

Jeder der beiden Instruktions-Speicher hatte in der bisherigen Konfiguration, wenn aktiv, den gesammten Adressraum zur Verfügung. Um den GDB-Stub verwenden zu können, ist es notwendig, dass beide Instruktions-Speicherbereiche zur gleichen Zeit ansprechbar sind. Daher wird das Boot-ROM im Instruktions-Adressbereich über dem Befehlsspeicher abgebildet.

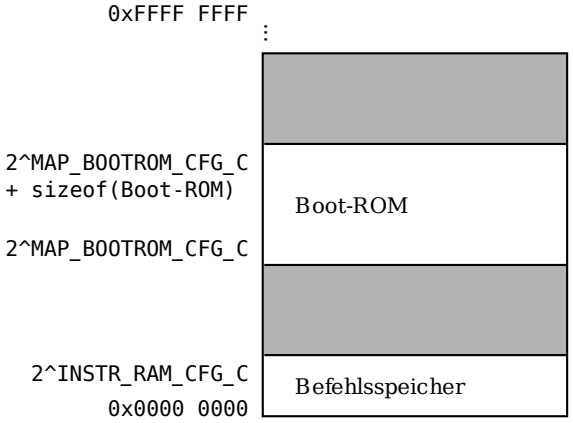


Abbildung 6.5: Aufteilung Instruktions-Speicher

Die Aufteilung des Instruktions-Speicher Adressraums wird bereits im VHDL-Code des SPEAR-Prozessors festgelegt. Der Code des GDB-Stubs liegt im Boot-ROM, dessen Position im Adressraum wird mittels `MAP_BOOTROM_CFG_C` in der SPEAR-Konfiguration festgelegt.

Der Adressbereich im Instruktion-Speicher, an der der GDB Remote-Stubs ausgeführt werden wird, ist bereits bei der Übersetzung des Stubs zu berücksichtigen. Die Startadresse des Stub-Codes ist in `spear_conf.h` mittels einer auf die .text-Sektion bezogenen `.org` Direktive anzugeben. Wenn Beispielsweise der Adressbereich des Boot-ROM's bei 32KB beginnt, sieht der zu erstellende Eintrag folgendermassen aus:
`asm(".text\n.org 032768\n"); //32k`.

Daten-Speicher

Der GDB Remote-Stub benötigt etwas Daten-Speicher um Anfragen des GDB abarbeiten zu können, aber auch um seinen internen Zustand zwischen zwei Anfragen zu speichern. Dafür wird der obere Bereich des Daten-Speichers reserviert.

Der Start des für den Stub reservierten Speicherbereichs wird in `spear_conf.h` festgelegt. In der aktuellen Version benötigt der GDB-Stub 1024 Bytes des

Daten-Speichers. Für eine SPEAR-Konfiguration die insgesammt über 8KB Daten-Speicher verfügt, sollte also ein Eintrag
`asm(".data\n.org 07168\n"); //8K-1K`
existieren.

Der Start des Stub-Speichers stellt klarerweise gleichzeitig das Ende des der Anwendung zur Verfügung stehenden Speicherbereichs dar. Also empfiehlt es sich, die selbe Adresse beim Übersetzten der Anwendung, mittels der Parameter `-mdatasize` und `-minit-stack`, and den Compiler zu übergeben, damit dieser die Beschränkung einhält.

Bsp.: `spear32-gcc main.c -O0 -g -mdatasize=7168 -minit-stack=7168`

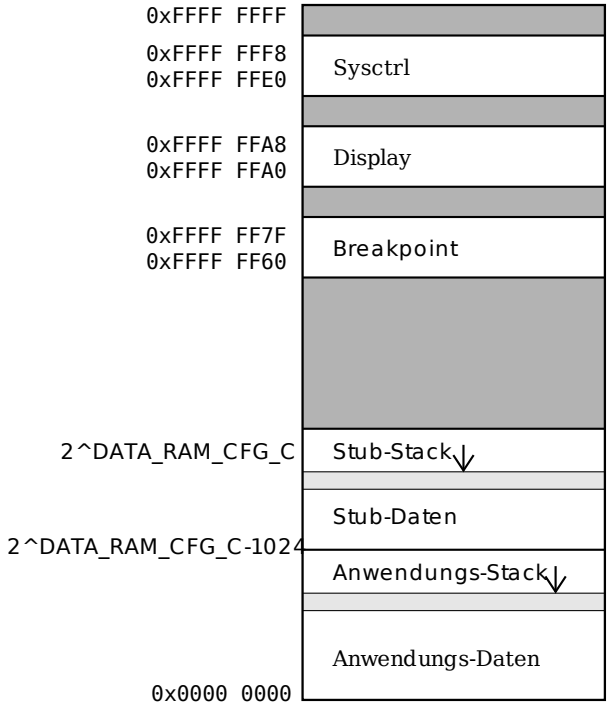


Abbildung 6.6: Aufteilung Daten-Speicher

6.4 Simulation

Dieser Abschnitt beschreibt die Verwendung des Spear2-Simulators zum Debuggen des Remot-Stubs. Ein derartiges Setup ist zur Entwicklung des GDB-Stub Codes sinnvoll, wenn keine Debug-Verbindung zu einem Hardware-Target möglich ist. Dieser Fall tritt insbesondere bei der initialen Entwicklung des Stubs auf, kann aber auch in Zukunft durch Hardware-Änderungen, die dazu führen, dass Anpassungen am Stub-Code notwendig werden, eintreten.

6.4.1 Stub-Debugging

Zum Debuggen des Remot-Stubs kann dieser, statt auf Ziel-Hardware, in einer Simulator-Instanz ausgeführt werden.

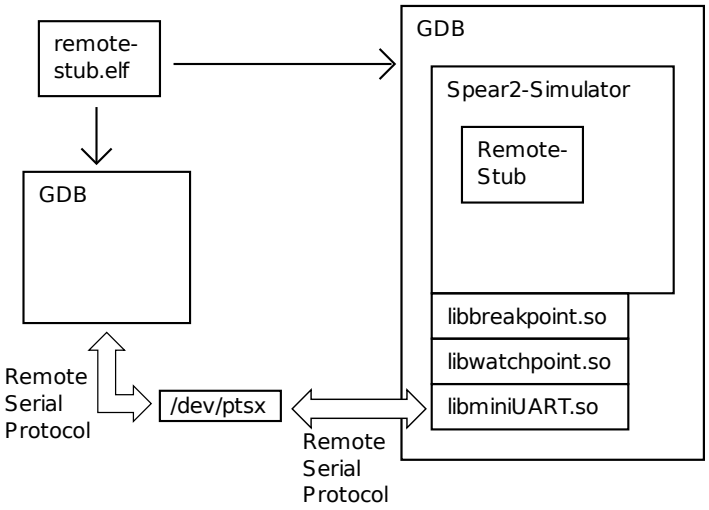


Abbildung 6.7: Übersicht Stub-Debugging

Wie in Abbildung 6.7 ersichtlich, sind an der Simulation zwei GDB-Instanzen beteiligt. Der links dargestellte GDB entspricht dabei dem Debug-Client, der auch bei Verwendung eines Hardware-Targets existieren würde. Rechts findet sich eine weitere GDB-Instanz, die das Zielsystem simuliert.

Die zu untersuchende Anwendung wird mit dem Remote-Stub zu einer einzigen Objekt-Datei gelinkt.

Auf der Target-Seite wird die Remote-Stub Objektdatei vom Simulator geladen und ausgeführt. Der Simulator verwendet libminiUART.so um dem Remote-Stub Code die Kommunikation über eine Pseudo-TTY Schnittstelle zu ermöglichen.

Die selbe Remote-Stub Objektdatei wird vom Client-GDB als Anwendungs-Programm geladen. Hier wird man dann mittels `target remote /dev/ptsx` eine Verbindung zum, vom Simulator der anderen GDB-Instanz ausgeführten, Remote-Stub erstellt. Danach lässt sich hier der der zum GDB-Stub verlinkte Anwendungs-Code debuggen. Der Client GDB schickt dabei Anforderungen mittels des GDB-Remote-Serial Protokolls an den Remote-Stub der vom Simulator des Target-GDB ausgeführt wird. Die Abarbeitung dieser Anfragen vom Remote-Stub kann mit dem Target-GDB untersucht werden.

Kapitel 7

Mögliche Verbesserungen

- Remote-Stub sollte sich mit Compileroptimierung übersetzten lassen (-Os). Derzeit funktioniert der Remote-Stub nur ohen Optimierung (-O0) korrekt. Vorteil: Es sollte möglich sein, die Grösse des Boot-Roms von derzeit 8K-Instruktionen auf 4K-Instruktionen zu verkleinern.
- Portierung des GDB (und des integrierten Simulators) und Remote-Stub auf die Spear16 Architektur. Derzeit funktioniert der GDB nur mit der Spear32 Architektur.
- Den Daten-Speicherbereich des Remote-Stubs vor dem Überschreiben durch die Anwendung schützen. Damit der Remote-Stub funktioniert, muss das oberste Kilobyte des Datenspeichers für ihn Reserviert werden. Wenn eine Anwendung mit falschen `-mdatasize=` bzw. `-minit-stack=` Einstellungen übersetzt wird, überschreibt sie Speicher des Remote-Stubs. Der Remote-Stub funktioniert infolge dessen nicht mehr korrekt. Selbiges gilt natürlich auch wenn der Stub-Speicher durch Stack-Überlauf der Anwendung oder einfach nur durch falsche Pointer überschrieben wird. In genau jenen Fällen wünscht man sich aber einen funktionierenden Debugger um den Grund für die falschen Speicherzugriffe zu finden.
- Wenn man dem Watchpoint-Modul eine Liste mit gültigen Adressbereichen mitgeben würde, könnte dieses ungültige Speicherzugriffe erkennen und das Programm anhalten.
- Wenn ein Programm auf das Zielsystem geladen wird, dass grösser als der Instruktions-Speicher ist, könnte der Remote-Stub das erkennen und einen Fehler melden.
- Zur Beschleunigung des Programm-Downloads könnte der Remote-Stub das Binary-Download Kommando unterstützen.
- Das GDB Remote-Protokoll unterstützt eine einfache Komprimierung wenn mehere gleiche Zeichen hintereinander übertragen werden. Könnte vom Remote-Stub implementiert werden um das Debugging etwas zu beschleunigen.

Literaturverzeichnis

[1] Martin Fletzer (2007), *SPEAR2 Handbuch*, *Scalable Processor for Embedded Applications in Real-Time Environments*, TU-Wien

[2] Wolfgang Puffitsch (2007), *Softwaretools fur den SPEAR*, TU-Wien

[3] Roman Seiger (2007), *miniUART Dokumentation*, *Version 0.9*, TU-Wien

[4] Intel (1988), *Hexadecimal Object File Format Specification*, *Revision A*, Intel Corporation, http://microsym.com/editor/assets/intelhex.pdf

[5] Steve Chamberlain, *libbfd*, *The Binary File Descriptor Librar, teil der GDB Version 6.6 Quellen*

[6] Wolfgang Huber (2002), *Peripherieanbindung an SPEAR: Extension Modules*, *Version 0.90*, Technical report, Embedded Computing Systems Group, TU Wien

[7] Richard Stallman, Roland Pesch, Stan Shebs, et al., *Debugging with GDB*, *Ninth Edition, for GDB version 6.6.50.20070708, 8.7.2007, teil der GDB-6.6 Quellen*

Abbildungsverzeichnis