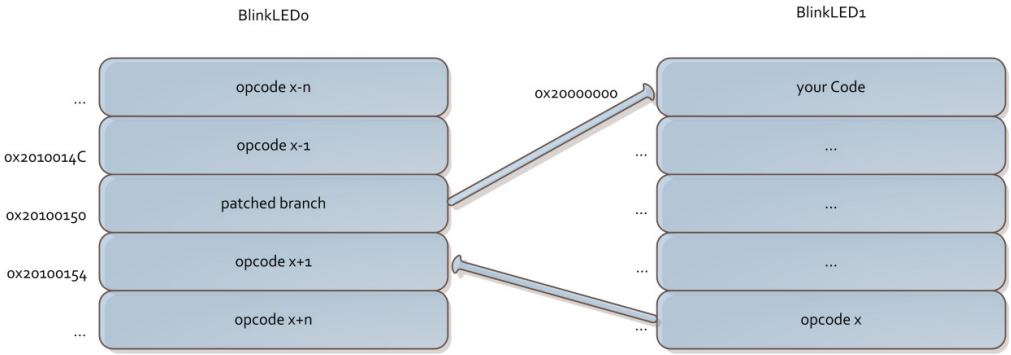
Thema, Ziele: Debugger für Embedded Systems

In diesem Praktikum wird die Thematik eines „on chip“ Debuggers behandelt. Der Unterschied zu einem „normalen“ Debugger über JTAG liegt darin, dass sich ein Teil des Debuggers selbst auf dem Ziel-Controller befindet. Beim OnChip Debugger wird keine zusätzliche Hardware benötigt, was ein grosser Vorteil mit sich bringt. Die Möglichkeiten die sich daraus ergeben sind enorm. So kann z.B. auch ein Gerät „gedebugged“ werden, bei welchem nur die USB Schnittstelle zugänglich ist. Eine grosse Schwierigkeit des OnChip Debuggers ist jedoch, zwei Programme auf einem Controller quasi parallel laufen zu lassen.

Aufgabe 1: Codepatching

Ein Debugger muss bestehenden Code eines zu testenden Programms manipulieren können, ohne das Programm neu zu kompilieren. Dies ist mittels Code-Patching möglich.

Einem im Speicher geschriebenen Code wird an einer gewünschten Stelle der Opcode durch einen eigenen Opcode ersetzt. Wichtig ist dabei, dass der Sprung mittels einer Instruktion aus der ausgeführten, zu patchenden Applikation springt. Dies wird oft durch eine Softwareinterrupt Instruktion oder einen Branch umgesetzt.



Ausgangslage:

Der zu patchende Code wird auf die Adresse 0x20100000 (SDRAM) compiliert. Es handelt sich dabei um das Projekt „BlinkLED0“. Die daraus entstehende Applikation wird als Zielapplikation verwendet.

Der Patch-Code wird auf 0x20000000 (SDRAM) compiliert. Das dazugehörige Projekt heisst „BlinkLED1“. Auf dieser Adresse beginnt gerade die Assembler-Funktion (Label) „patch“.

Ziel:

Die Zielapplikation (zu patchende Applikation) soll durch einen Branch an der Codestelle vor dem „LED_Set(1)“ Funktionsaufruf zum Patch-Code umgeleitet werden. Nach der Ausführung des „Patched Codes“ (Funktion „patchedCode“ soll der überschriebene Opcode wieder an der ursprünglichen Adressstelle eingefügt werden und der Programmcounter dementsprechend gesetzt werden.

a) An welcher Adresse muss der Branch Befehl hineinkopiert werden?

   Tipp: Mittels Debugger kann die Disassemblierung betrachtet werden

b) Wie heisst der Branch Befehl 0xXXXXXXXXX?

   Tipp: Die Branch Instruktion muss von Hand berechnet werden. Die Beschreibung des Befehls kann im „ARM Architecture Reference Manual“ auf Seite A4-10 gefunden werden.

c) Studieren Sie die Assembler Funktionen (Labels) „patch“ und „codeInsertion“. Was ist der Unterschied dieser Routinen und was machen sie genau?

d) Der Einfachheit halber soll die Zielapplikation gerade nach der „codeInsertion“ aus dem „BlinkLED1“ Projekt ausgeführt werden. Schreiben Sie die zugehörige „inline“ Assembleranweisung.

   Tipp: Die Einstiegsadresse der Applikation kann im „map“-File des BlinkLED0 Projekt mit dem Label „_start“ gefunden werden.

Aufgabe 2: Disassembler

Bei einem Programmablauf ohne „branches“ wird der Programmcounter des ARM Controller nach jeder Instruktion um 4 Bytes erhöht.

Falls jedoch „branches“ vorhanden sind, was üblicherweise in den meisten Fällen zutrifft, müssen akausale Entscheidungen getroffen werden, damit beim „Step“-Event analysiert werden kann, wohin der Programmcounter springt, um an dieser Adresse ein Breakpoint zu setzen (Codepatching).

Ziel: Es soll eine ARM Disassembler Anwendung auf dem PC geschrieben werden. Ein vollständiger Disassembler ist sehr umfangreich. Aus diesem Grund sollen im Rahmen dieses Praktikums nur „B“ und „BL“ ARM Assembler Instruktionen analysiert werden. Als Eingabe sollen folgende Test ARM Opcodes verwendet werden:

- 0xEAFBFFF9 (mit PC: 0x20100150, CPSR: 0x00)
- 0xEBFBFFF9 (mit PC: 0x20100150, CPSR: 0x00)
- 0x1AFCFFAC (mit PC: 0x21200150, CPSR: 0x80)
- 0xE3A00000 (mit PC: 0x2120150, CPSR: 0x80)

Welches sind „B“ resp. „BL“ Instruktionen und wohin springt der Programcounter? Finden Sie es mit Hilfe ihres Programms heraus.

Eine Vorlage finden Sie auf dem Skripte Server.

Tipp: Die Assembler Instruktionen sind im „ARM Architecture Reference Manual“ zu finden.

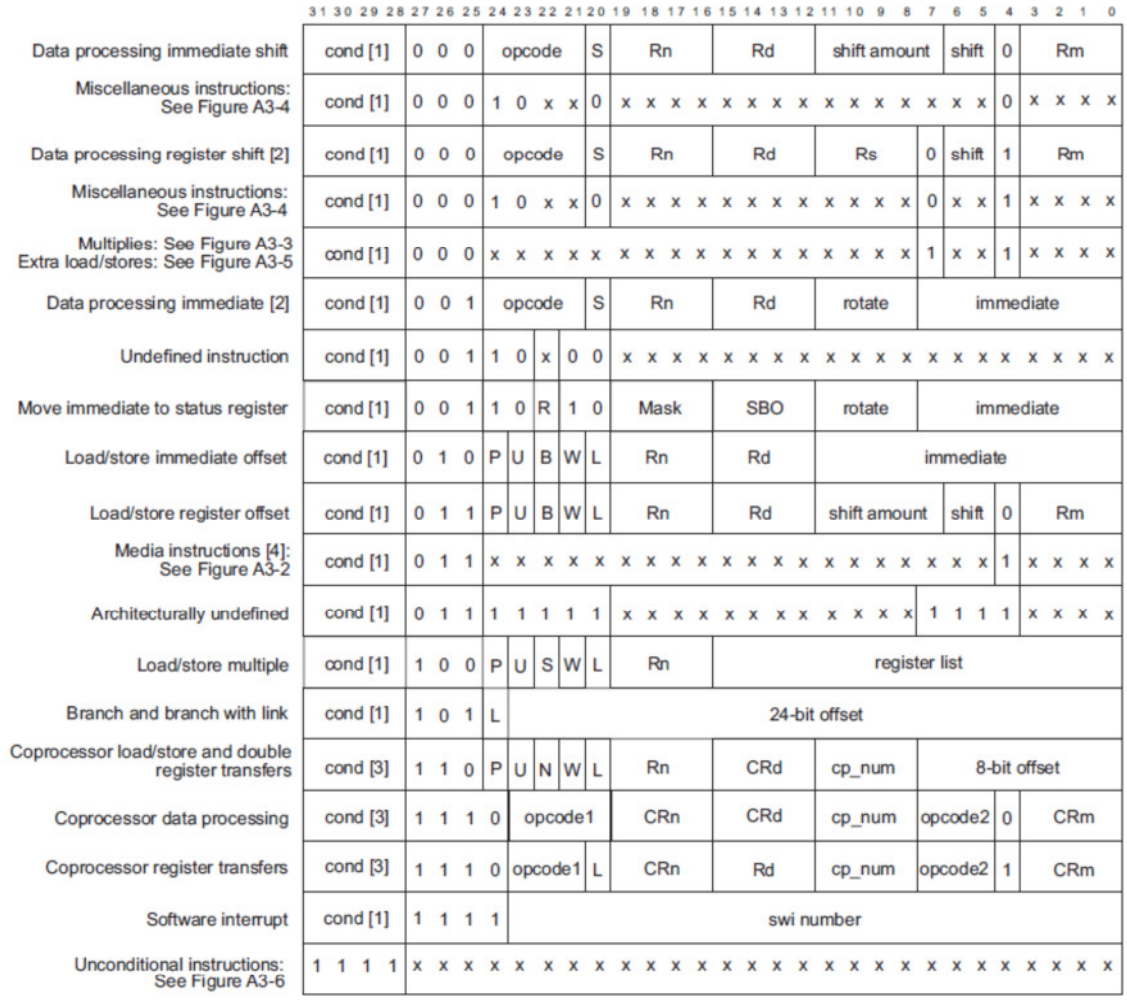
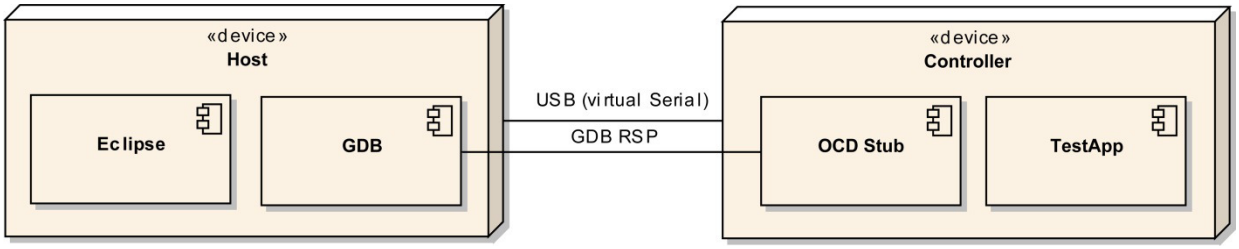


Figure A3-1 ARM instruction set summary

Aufgabe 3: Funktionsweise eines „on chip“ Debuggers

Anhand dieser Aufgabe soll das Zusammenspiel der verwendeten Komponenten aufgezeigt werden, damit das meist im Hintergrund ablaufende Schema eines Debuggers verstanden werden kann.



a) Starten Sie den Debugger „arm-none-eabi-gdb.exe“ in der Konsole „cmd.exe“

b) Folgende Konsoleneingaben sind nun nötig
   a. file Snake.elf
   b. target remote comX (siehe DeviceManager → OCDStub)
   c. load
   d. set debug remote 1
   e. b main (fügt einen Breakpoint an der ersten Zeile in der main Fkt hinzu)
   f. s (für „single step“)

c) Was passiert auf dem Controller zwischen dem Konsolenausgabe „$s#73“ und „S05“?

d) Wie wird das Hinzufügen vom „main“ Breakpoint gehandelt? Geben Sie dazu c (run) in die Konsole ein.

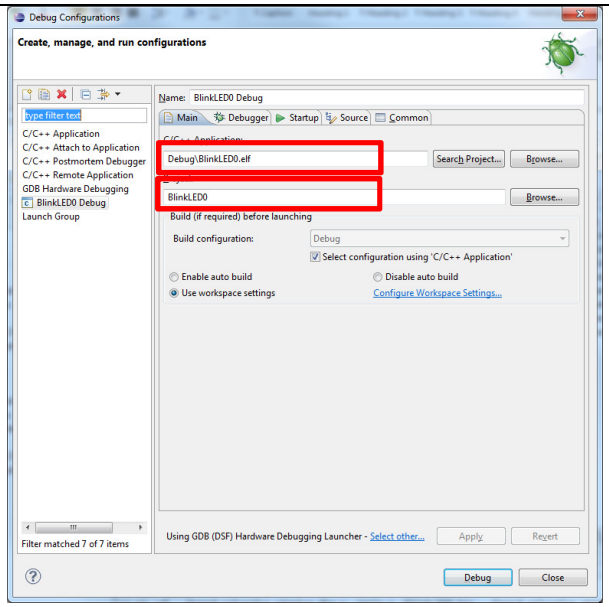
   Tipp: Der Breakpoint wird an der Adresse „0x200004a40“ eingefügt. Mit dem Befehl „$X…“ wird in den Speicher des Controllers geschrieben und mit dem Befehl „$m…“ wird aus dem Speicher des Controllers gelesen.

Das RS Protokoll kann in der Dokumentation „Debugging with GDB.pdf“ gefunden werden.

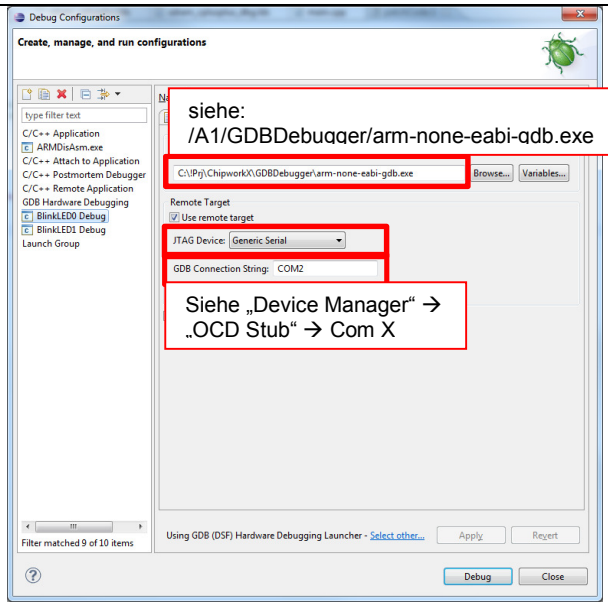
Debug Einstellungen für OCD auf ChipworkX Board

Unter Debug Configurations → GDB Hardware Debugging (Doppelklick):

Tab Main



Tab Debugger



Tab Startup