

## Praktikum Rechnerorganisation

### Versuch 3: Programmierung des Beispielrechners in Assembler

#### Lernziele

- Erstellung eines Assemblerprogramms
- Ausführung von Programmen auf dem Beispielrechner im FPGA

#### Werkzeuge

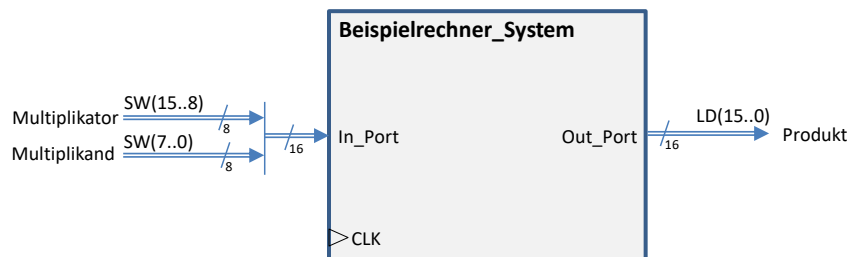
- BASYS3-Board
- Vivado (Version 2019.2)
- Eclipse IDE for C/C++ Developers (Version 2020-12)
- GNU Cross-Compile-Toolchain für MIPS

#### Vorbereitung (Vor dem Praktikum)

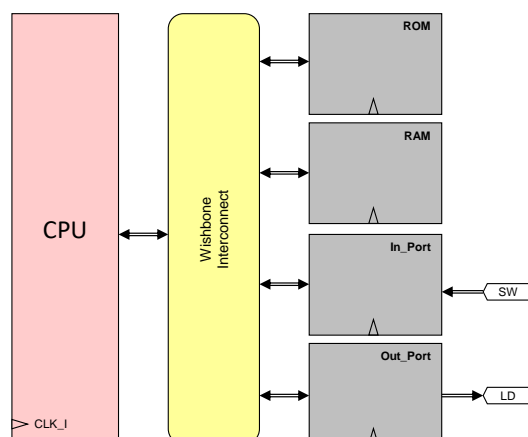
- Wiederholen Sie den Abschnitt der Vorlesung über den Beispielrechner
- Richten Sie die Software-Entwicklungsumgebung ein (Anleitung: [Software-Entwicklungsumgebung.pdf](#))
- Beginnen Sie mit der Erstellung des Assemblercodes für die in den Teilaufgaben geforderten Unterprogramme. Notieren Sie sich dabei auftretende Fragen, damit diese im Praktikum geklärt werden können.

#### Systembeschreibung

In diesem Praktikumsversuch soll mit dem Beispielrechner ein einfaches System zur Multiplikation zweier Zahlen erstellt werden. Die Eingabe der beiden Zahlen erfolgt über die Schiebeschalter (SW) des BASYS3-Boards. Die Ausgabe des Ergebnisses erfolgt über dessen Leuchtdioden (LD).



Die folgende Abbildung zeigt das Blockschaltbild dieses Systems:

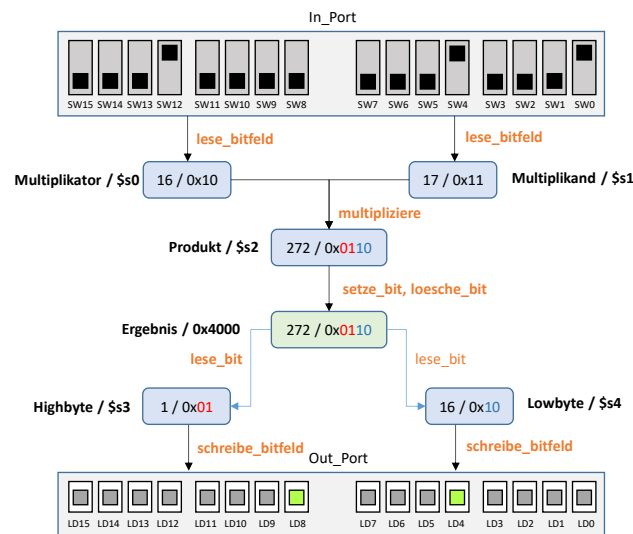


Das Programm wird vor der Ausführung durch den Debugger im ROM abgelegt. Das RAM enthält die globalen Variablen und den Stack. Zur Anbindung der Schiebeschalter wird eine „In\_Port“-Komponente verwendet. Diese stellt in einem Register den von den Schaltern eingelesenen zur Verfügung. Die Ausgabe erfolgt mit einer „Out\_Port“-Komponente. Diese enthält ein rücklesbares Register, dessen Wert auf den Leuchtdioden ausgegeben wird.

Die in diesem Versuch erstellten Unterprogramme werden teilweise in späteren Versuchen wiederverwendet. Um alle Unterprogramme zu testen, wird durch das Hauptprogramm in einer Endlosschleife die Multiplikation in der folgenden Art durchgeführt:

- Der Multiplikator wird mit dem Unterprogramm **lese\_bitfeld** von den Schaltern SW15 - SW8 eingelesen und anschließend in Register **\$s0** abgelegt.
- Der Multiplikand wird mit dem Unterprogramm **lese\_bitfeld** von den Schaltern SW7 - SW0 eingelesen und anschließend in Register **\$s1** abgelegt.
- Die Multiplikation wird mit dem Unterprogramm **multipliziere** durchgeführt. Das Produkt wird anschließend im Register **\$s2** abgelegt.
- Das Produkt wird mit den Unterprogrammen **setze\_bit** und **loesche\_bit** an der Adresse **0x4000** in das RAM geschrieben.
- Das obere Byte des Ergebnisses (Highbyte) wird mit dem Unterprogramm **lese\_bit** aus dem Speicher gelesen und anschließend im Register **\$s3** abgelegt.
- Das untere Byte des Ergebnisses (Lowbyte) wird mit dem Unterprogramm **lese\_bit** aus dem Speicher gelesen und anschließend im Register **\$s4** abgelegt.
- Das Highbyte wird mit dem Unterprogramm **schreibe\_bitfeld** auf den Leuchtdioden LD15 - LD8 ausgegeben.
- Das Lowbyte wird mit dem Unterprogramm **schreibe\_bitfeld** auf den Leuchtdioden LD7 - LD0 ausgegeben.

Sie können die Zwischenergebnisse mit Hilfe des Debuggers durch gezielt gesetzte Haltepunkte (Breakpoints) überprüfen. Nutzen Sie dazu die Ansichten (Views) „Registers“ bzw. „Memory Browser“. Das folgende Diagramm veranschaulicht den vollständigen Ablauf anhand des Beispiels  $16 \times 17 = 272$  (Hexadezimal:  $0x10 \times 0x11 = 0x0110$ ):



## Aufgabe 1: Erstellung und Test des Unterprogramms `lese_bitfeld`

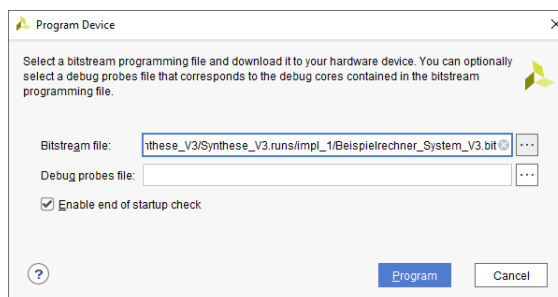
### Aufgabenstellung

Vervollständigen Sie das Unterprogramm `lese_bitfeld`, welches die Aufgabe hat, ein Wort von einer vorgegebenen Speicheradresse zu lesen und daraus ein bestimmtes Bitfeld zurückzugeben. Parameter des Unterprogramms sind die zu lesende Speicheradresse, eine Maske für die zu lesenden Bits sowie die Position des Bitfeldes im Wort. Die Funktion dieses Unterprogramms ist durch folgenden C-Code definiert:

```
int lese_bitfeld(int Adresswert, int Maske, int Shift) {  
    int * Adresse = (int *) Adresswert;  
    return (*Adresse >> Shift) & Maske;  
}
```

### Durchführung

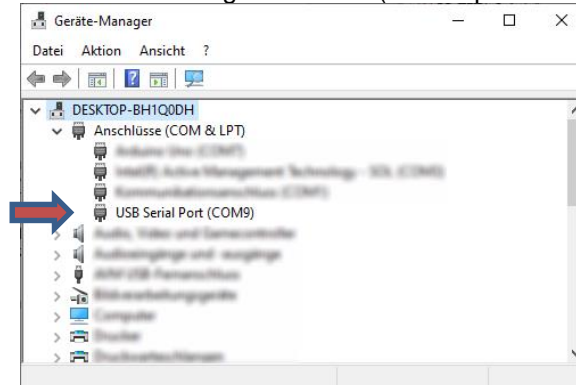
1. Entpacken Sie die bereitgestellte Verzeichnisstruktur in einem Arbeitsverzeichnis. Achten Sie darauf, dass dessen Pfad keine Leer- oder Sonderzeichen wie zum Beispiel Umlaute enthält.
2. Starten Sie Eclipse. Geben Sie als „Workspace“ das Verzeichnis „Software“ in der bereitgestellten Verzeichnisstruktur an.
3. Importieren Sie in Eclipse das Projekt `Multiplikation` in den Workspace: [File] → [Import...] → [General] → [Existing Projects into Workspace] → [Next] → [Browse...] → <...\Software\Multiplikation> → [Ordner auswählen] → <Haken bei Projects: Multiplikation setzen> → [Finish]
4. Ergänzen Sie den Code des Unterprogramms `lese_bitfeld` (Funktion siehe obiger C-Code) an der mit TODO markierten Stelle. Lassen Sie das Programm übersetzen (🔧) und beheben Sie alle Syntaxfehler.
5. Vor Ausführung des Programms muss zuerst das FPGA mit einer Bitstream-Datei (`Hardware\Synthese_V3\Synthese_V3.runs\impl_1\Beispielrechner_System_V3.bit`) programmiert werden. Dazu muss das BASYS3-Board per USB-Kabel mit dem PC verbunden und eingeschaltet werden. Zur Programmierung des FPGAs wird der Hardware-Manager von Vivado verwendet. Rufen Sie dazu in Vivado (ohne Projekt) den Hardware-Manager auf: [Flow] → [Open Hardware Manager]. Nun stellen Sie die Verbindung zum Board her: [Tools] → [Auto Connect]. Starten Sie nun die Übertragung des Bitstreams zum FPGA: [Tools] → [Program Device] → [xc7a35t\_0]. Wählen Sie als „Bitstream File“ die oben genannte Datei an und klicken Sie anschließend auf [Program].



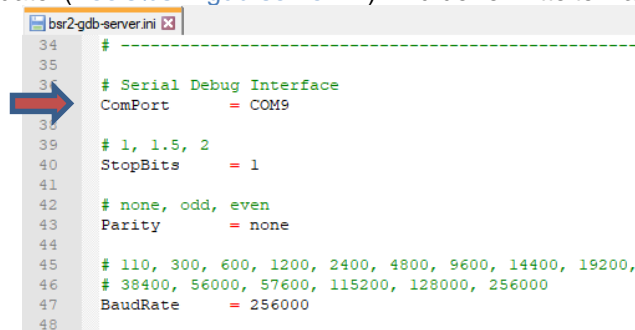
Nun sollte die grüne DONE-LED des BASYS3-Boards leuchten:



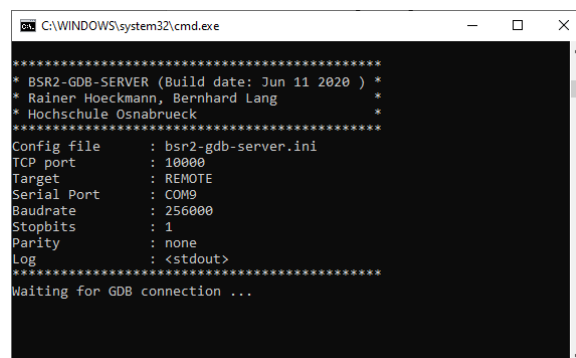
6. Zum Debuggen wird von Eclipse der GNU Debugger (GDB) aufgerufen ([C:\Tools\mips-sourcery\bin\mips-sde-elf-gdb.exe](#)). Dieser kommuniziert per TCP-Protokoll mit einem lokal auf dem PC laufenden Serverprogramm ([Tools\bsr2-gdb-server.exe](#)). Das Serverprogramm verwendet eine (virtuelle) serielle Schnittstelle, um mit dem Beispielrechner im FPGA zu kommunizieren. Die serielle Schnittstelle verwendet das gleiche USB-Kabel, welches auch zum Programmieren des FPGAs verwendet wird. Der PMo-dUSBUART-Adapter wird in diesem Versuch **nicht** benötigt. Sie können den Anschlussnamen des virtuellen COM-Ports im Windows-Gerätemanager ermitteln (im Beispiel COM9):



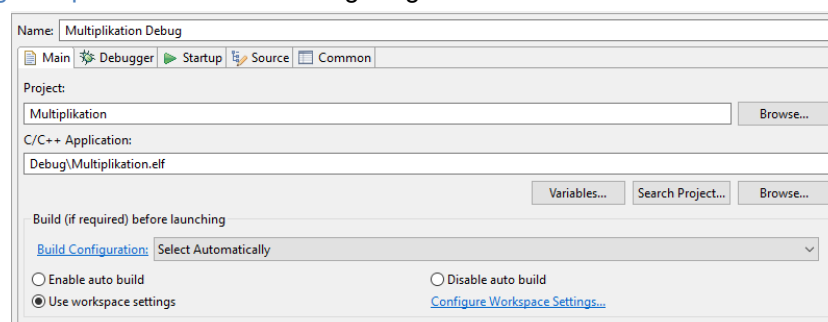
In einer Initialisierungsdatei ([Tools\bsr2-gdb-server.ini](#)) wird der ermittelte Name eingetragen:



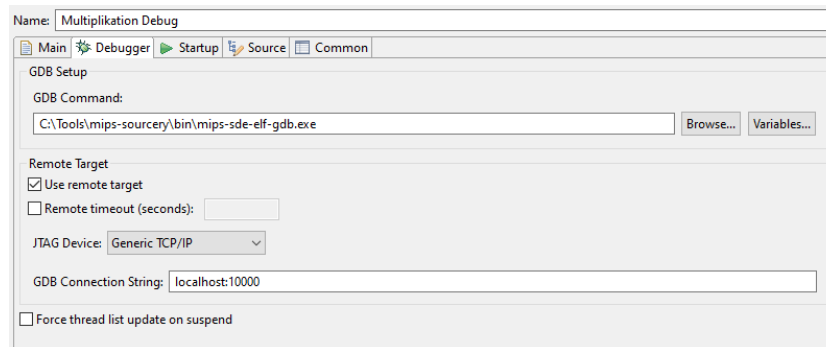
7. Mit einem Batch-File ([Tools/run\\_remote.bat](#)) wird das Serverprogramm gestartet. Die Ausgaben erscheinen in einem Konsolenfenster (eine Warnung von der Windows-Firewall können Sie mit [Abbrechen] beantworten):



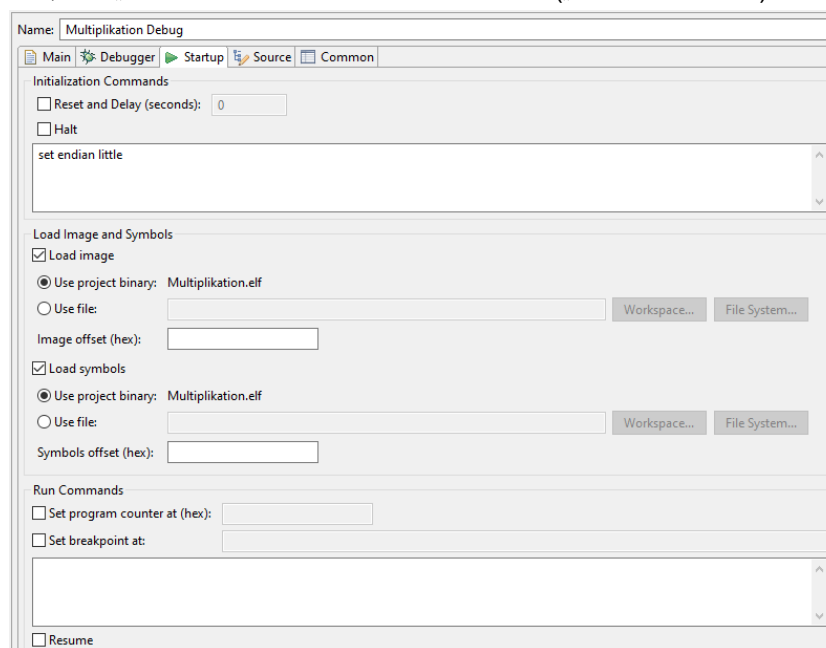
8. In Eclipse muss zum Ausführen eines Applikationsprojektes eine Debug-Konfiguration erstellt werden. Markieren Sie dazu im „Project Explorer“ von Eclipse das Projekt „Multiplikation“ und wählen anschließend die Option [Run] → [Debug Configurations]. Erstellen Sie eine neue Debug-Konfiguration vom Typ „GDB Hardware Debugging“ ( ). Nun müssen in der neuen Debug Konfiguration noch einige Einstellungen vorgenommen werden. Im Tab „Main“ sollte als Project „Multiplikation“ sowie die C/C++ Application „[Debug\Multiplikation.elf](#)“ bereits eingetragen sein:



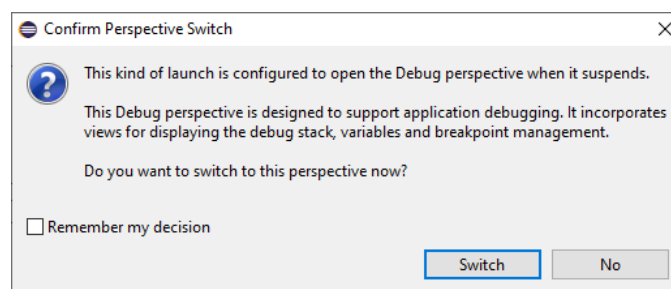
Im Tab „Debugger“ muss der Wert bei „GDB Command“ der Pfad auf den von uns verwendeten Debugger gesetzt werden: (C:\Tools\mips-sourcery\bin\mips-sde-elf-gdb.exe). Der Haken bei „Use remote target“ ist gesetzt. Als „GDB Connection String“ muss „localhost:10000“ eingetragen werden (Auf diesem Port wartet der BSR2-GDB-Server auf Verbindungsanfragen):



Im Tab „Startup“ wird im Feld „Initialization Commands“ eine Anweisung eingetragen, welche den Debugger anweist, das „Little Endian“-Format zu verwenden („set endian little“):



Mit [Debug] speichern Sie einerseits die gemachten Einstellungen und starten gleichzeitig das Debugging. Sie werden gefragt, ob nun auf die Debug-Perspektive umgeschaltet werden soll, dies können Sie mit [Switch] bestätigen:



Nun wird vom Debugger das Programm zum Beispielrechner übertragen:

```
C:\WINDOWS\system32\cmd.exe

*****
* BSR2-GDB-SERVER (Build date: Jun 11 2020 ) *
* Rainer Hoeckmann, Bernhard Lang           *
* Hochschule Osnabrueck                     *
*****
Config file      : bsr2-gdb-server.ini
TCP port        : 10000
Target          : REMOTE
Serial Port     : COM9
Baudrate        : 256000
Stopbits        : 1
Parity          : none
Log             : <stdout>
*****
Waiting for GDB connection ...
GDB connection established
Connecting to target ...
Target connection established
####
```

Die Ausführung des Programms pausiert vor dem ersten Befehl:

```
21 # -----
22 # Einsprungpunkte
23 # -----
24 .section .text
25
26 Reset_Vector:
27 beq $zero, $zero, main
```

9. Die Inhalte der Prozessorregister können in der Ansicht „Registers“ beobachtet werden. Diese kann über [Window] → [Show View] → [Registers] hinzugefügt werden. Sie können diese Ansicht dann mit der Maus in einen anderen Bereich des Fensters ziehen:


Name	Value
General Registers	
zero	0
at	0
v0	0xffffffff (Hex)
v1	0
a0	0x8308 (Hex)
a1	0x1 (Hex)
a2	0
a3	0
t0	0x1 (Hex)
t1	0x2 (Hex)

10. Zum Testen des Unterprogramms `lese_bitfeld` wird im Hauptprogramm der Multiplikator und der Multiplikand von den Schiebeschaltern eingelesen. Stellen Sie auf den 16 Schiebeschaltern des BASYS3-Boards zwei 8-Bit-Zahlen ein (Sie können dazu das oben angegebene Beispiel verwenden). Führen Sie das Programm in Einzelschritten (F5/🔍) aus, bis im Hauptprogramm die beiden Aufrufe des Unterprogramms `lese_bitfeld` abgeschlossen sind:

```
264 # -----
265 # Multiplikation ausfuehren
266 # -----
267 ori $a0, Multiplikator, 0
268 ori $a1, Multiplikand, 0
269 la $t0, multipliziere
270 jalr $ra, $t0
271 ori Produkt, $v0, 0
```

11. Überprüfen Sie in der Ansicht „Register“, ob in den Registern `$s0` und `$s1` die von den Schaltern eingelesenen Werte korrekt abgelegt wurden. Sie können durch Rechtsklicken in der Ansicht „Registers“ das angezeigte Zahlenformat ändern:

0101 s1	0x11 (Hex)	267 ori \$a0, Multiplikator, 0	#	Produkt
0101 s2	272 (Decimal)			
0101 s3	0x0 (Hex)			
0101 s4	0 (Decimal)			
0101 s5	0			
0101 s6	0			
0101 s7	0			

12. Die Schaltfläche  (oder Strg-F2) beendet das Debugging und auch den BSR2-GDB-Server. Dieser muss anschließend neu gestartet werden. Wollen Sie, dass der Server nach Beendigung immer wieder neu gestartet wird, können Sie die Batch-Datei `Tools\run_remote_loop.bat` zum Starten des Servers verwenden.

## Aufgabe 2: Erstellung und Test des Unterprogramms **multipliziere**

### Aufgabenstellung

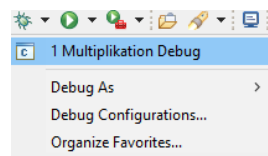
Vervollständigen das Unterprogramm **multipliziere**, welches die Aufgabe hat, zwei **vorzeichenlose** Zahlen zu multiplizieren. Die Funktion dieses Unterprogramms wird durch folgenden C-Code definiert:

```
unsigned int multipliziere (unsigned int a, unsigned int b) {
    unsigned int Produkt = 0;
    for(int bit = 0; bit < 32; bit++) {
        if(bit_von(b, bit) != 0) {
            Produkt = Produkt + a;
        }
        a = a << 1;
    }
    return Produkt;
}
```

Zum Testen des Unterprogramms werden im Hauptprogramm die eingelesenen Zahlen miteinander multipliziert und das Ergebnis im Register **\$s2** abgelegt.

### Durchführung

1. Vervollständigen Sie das Unterprogramm **multipliziere**. Lassen Sie die Applikation übersetzen und beheben Sie alle Syntaxfehler.
2. Um eine existierende Debug-Konfiguration bequem auszuführen, kann man mit den folgenden Einstellungen eine existierende Debug-Konfiguration wiederverwendet werden, Nutzen Sie dazu das schwarze Dreieck neben dem Käfer-Symbol:



Denken Sie daran, dass zur Ausführung ein laufender BSR2-GDB-Server benötigt wird.

3. Setzen Sie einen Haltepunkt (Breakpoint) an die Stelle, an der im Hauptprogramm die Werte eingelesen wurden und die Multiplikation durchgeführt werden soll. Dazu können Sie auf den Rand vor der gewünschten Zeile doppelklicken:

```
264 # -----
265 # Multiplikation ausfuehren
266 # -----
267 ori $a0, Multiplikator, 0      # Produkt = multipliziere(Multiplikator, Multiplikand);
268 ori $a1, Multiplikand, 0
269 la $t0, multipliziere
270 jalr $ra, $t0
271 ori Produkt, $v0, 0
```

Mit der Schaltfläche (F8) kann die Ausführung bis zum Erreichen des nächsten Haltepunktes fortgesetzt werden (Resume). Wird kein Haltepunkt erreicht, läuft das Programm unbegrenzt weiter. Sie können die Ausführung aber mit der Schaltfläche (F5) wieder anhalten (Suspend).

4. Führen Sie das Programm aus, bis im Hauptprogramm der Aufruf des Unterprogramms „multipliziere“ abgeschlossen ist.
5. Überprüfen Sie, ob in Register **\$s2** das korrekte Ergebnis abgelegt wurde.
6. Stoppen Sie das Debugging

## Aufgabe 3: Erstellung und Test des Unterprogramms **loesche\_bit**

### Aufgabenstellung

Vervollständigen Sie das Unterprogramm **loesche\_bit**, welches die Aufgabe hat, an einer vorgegebenen Adresse ein bestimmtes Bit zu löschen. Parameter des Unterprogramms sind die Adresse sowie die Indexnummer des zu löschenden Bits. Die Funktion dieses Unterprogramms ist durch folgenden C-Code definiert:

```
void loesche_bit(int Adresswert, int Bitnummer) {
    int * Adresse = (int *) Adresswert;
    *Adresse = (*Adresse) & ~(1 << Bitnummer);
}
```



Zum Testen des Unterprogramms wird das Ergebnis der Multiplikation (Register **\$s2**) in den Arbeitsspeicher an der Adresse **0x4000** umkopiert. Dabei wird für Null-Bits das Unterprogramm **loesche\_bit** aufgerufen, für Eins-Bits das Unterprogramm **setze\_bit**.

### Durchführung

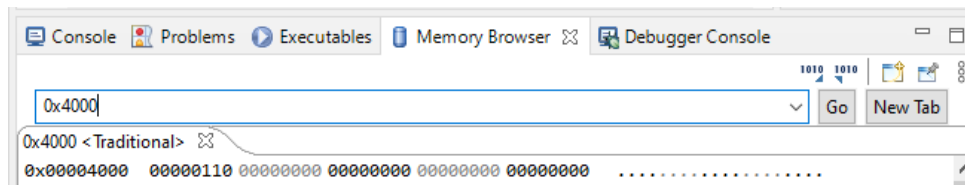
1. Vervollständigen Sie das Unterprogramm **loesche\_bit**. Lassen Sie die Applikation übersetzen und beheben Sie alle Syntaxfehler.
2. Führen Sie das Programm aus, bis im Hauptprogramm das bitweise Umkopieren durch die Schleife abgeschlossen ist:

```

306 # -----
307 # Ergebnis-Highbyte bitweise aus dem Speicher lesen
308 # -----
309 li Highbyte, 0 # Highbyte = 0
310

```

3. Die Inhalte von Speicherstellen können in der Ansicht „Memory Browser“ beobachtet werden. Diese kann über [Window] → [Show View] → [Memory Browser] hinzugefügt werden. Vergleichen Sie das im Speicher abgelegte Wort (Adresse 0x4000) mit dem erwarteten Wert. Bei Abweichungen beenden Sie die Ausführung, korrigieren Sie Ihr Unterprogramm und versuchen Sie es erneut.



### Aufgabe 4: Erstellung und Test des Unterprogramms **schreibe\_bitfeld**

#### Aufgabenstellung

Ergänzen Sie das Unterprogramm **schreibe\_bitfeld**, welches die Aufgabe hat, von einer vorgegebenen Speicheradresse ein Wort zu lesen, bestimmte Bits auf einen definierten Wert zu setzen und das Ergebnis wieder zurückzuschreiben. Parameter des Unterprogramms sind die Speicheradresse, der Wert für die zu setzenden Bits, eine Maske für die zu schreibenden Bits sowie die Position des Bitfeldes im Wort. Die Funktion dieses Unterprogramms ist durch folgenden C-Code definiert:

```

void schreibe_bitfeld(int Adresswert, int Wert, int Maske, int Shift) {
    int * Adresse = (int *) Adresswert;
    *Adresse = (*Adresse & ~(Maske << Shift)) | ((Wert & Maske) << Shift);
}

```

Zum Testen des Unterprogramms werden im Hauptprogramm die beiden Bytes des Ergebnisses mit dem Unterprogramm **lese\_bit** wieder aus dem Speicher gelesen und anschließend mit dem Unterprogramm **schreibe\_bitfeld** in das Register der „Out\_Port“-Komponente geschrieben. Dadurch wird das Ergebnis auf den LEDs des BASYS3-Boards ausgegeben.

### Durchführung

1. Vervollständigen Sie das Unterprogramm **schreibe\_bitfeld**. Lassen Sie die Applikation übersetzen und beheben Sie alle Syntaxfehler.
2. Führen Sie das Programm aus, bis im Hauptprogramm der Aufruf des Unterprogramms **schreibe\_bitfeld** sowie das Schreiben des Ergebnisses in das Register der „Out\_Port“-Komponente abgeschlossen sind. Überprüfen Sie das auf den LEDs ausgegebene Ergebnis.
3. Sie können sich im Memory Browser auch den Inhalt des Registers der Komponente Out\_Port an der Adresse 0x8200 anzeigen lassen. Auch die Änderung des Wertes ist möglich.

**Bitte legen Sie nach Abschluss des Versuchs folgende Datei im OSCA-Dateibereich Ihrer Arbeitsgruppe in einem neuen Ordner „V3“ ab:**

- Multiplikation.S