Dokumentation
Makefile
Linkerscript
Startup

1. Makefile 2
1.1 Compilerpräfix 2
1.2 Compilerflags 2
1.3 Quellen und Ausgabe 3
1.4 make Regeln 3
2. Speicherhierarchie 4
2.1 Exceptionvektoren 4
2.2 Sections 4
2.3 Speicherlayout 5
3. Linkerscript 6
3.1 Exceptionvektoren 7
3.2 ROM Bereich 8
3.3 RAM Bereich 8
4. Startup 10

1. Makefile

Als erstes das Makefile als Ganzes und danach die einzelnen Abschnittsbeschreibungen.

```
#Compilerprefix and Flags
PRC = mips-elf-

#standard flags for compiling
CFLAGS = -g -O2 -Wall -Wextra -march=mips1
#the div0 check uses alway branch delay slots
CFLAGS += -mno-check-zero-division
#there is nothing but emptyness around our code
CFLAGS += -ffreestanding
CFLAGS += -msoft-float
CFLAGS += -nostartfiles
CFLAGS += -T linkerscript.lds

#Binaryname and Sourcefiles
BIN = test
OBJS = start.o div_u.o mult.o multu.o sp2io.o main.o

#compile Objects and Link Objects, generate map File
#Objectdump, create Binary, measure Codesize
all:  $(OBJS)
      $(PRC)gcc $(CFLAGS) $(OBJS) -o $(BIN) -lm -lc -specs=nosys.specs
            -Wl,-Map -Wl,$(BIN)_map.map
      $(PRC)objdump -D $(BIN) > $(BIN)_dissassembly.s
      $(PRC)objcopy -S -O binary $(BIN) $(BIN).bin
      $(PRC)size $(BIN)

%.o:  %.S
      $(PRC)gcc $(CFLAGS) -c   $< -o $@

%.o:  %.s
      $(PRC)gcc $(CFLAGS) -c   $< -o $@

%.o:  %.c
      $(PRC)gcc $(CFLAGS) -c  $< -o $@

clean:
      rm -f $(OBJS)
      rm -f $(BIN)
      rm -f $(BIN).bin
      rm -f $(BIN)_dissassembly.s
      rm -f $(BIN)_map.map
```

1.1 Compilerpräfix

Als erstes der Compilerpräfix, dieser wird extra gespeichert um später z.B. auf einen anderen MIPS Compiler wechseln zu können. Aus diesem Präfix werden die Kommandozeilenaufrufe des Compilers generiert. Also Beispielhaft mips-elf-gcc oder mips-elf-objdump.

1.2 Compilerflags

Zu den Compilerflags gehören erstmal die Standardmäßigen, wie -02 für die Optimierungsstufe sowie –Wall und –Wextra um möglichst viele Fehlermeldungen zu erhalten, zuletzt in der ersten Zeile noch die Angabe, dass er für den MIPS1 Befehlssatz compilieren soll. Das –g sorgt für Debug Informationen im Compilat, somit liefert der Objectdump lesbarere Ergebnisse.

Weiterhin soll die Division nicht gegen div0 getestet werden, da dafür eine Systemexception bereitstehen müsste und zudem fängt das bereits die Punktrechnung Software Emulation ab.
Weiterhin ist das Programm freestanding, läuft also nicht auf einem Betriebsystem. Eine FPU ist nicht vorhanden, also muss soft-float genutzt werden. Weiterhin soll kein C Compiler startup Code mit eingebunden werden, das wird manuell in der start.s erledigt.
Schlussendlich wird noch das Linkerscript bekannt gegeben.

1.3 Quellen und Ausgabe

Hinter OBJS müssen alle Quelldateien angegeben werden, egal ob mit der Endung .s/.S/.c es muss mit .o hingeschrieben werden. Der Name nach BIN gibt an wie die Ausgabedateien heißen sollen.

1.4 make Regeln

Der Rest des Makefiles besteht aus den make Regeln. Wird „make all“ eingetippt so wird die Regel all: ausgeführt. Dort werden zuerst alle Quelldaten (OBJS) compiliert und als Objectfiles abgelegt (.o) dabei sind die Regeln %.o behilflich, welche weitere Flags für den Vorgang angeben können.
Danach werden alle Objectfiles gelinkt in der Zeile mit gcc. Mit der Angabe weiterer zu linkenden Objekten wie der lib math und der libc (lm und lc). Das nosys.specs dient der Angabe, dass kein Betriebssystem vorhanden ist und somit auch keine Systemcalls vorhanden. So würde z.B. ein malloc() das Betriebsystem nach Speicher fragen, hier beim Spaceage2 müssen dies eingebettete libs übernehmen, die mit nosys.specs eingebunden werden. Weiterhin wird in dieser Zeile auch das map File erzeugt um zu überprüfen ob auch alles wie im Linkerscript angegeben gelinkt wurde.
Danach wird der Objectdump erstellt um überprüfen zu können was der Compiler erzeugt hat. Danach wird durch Objcopy das Binary erzeugt, welches auf den Spaceage2 hochgeladen werden kann, um die Größe des Programms zu wissen wird noch size aufgerufen.

2. Speicherhierarchie

Bevor das Linkerscript erklärt werden kann, muss erst einmal definiert werden wie das Speicherlayout vom Spaceage2 aussieht. Dieses gilt nur für Programme die direkt also ohne Betriebsystem auf dem Spaceage2 laufen.
Der Spaceage2 hat 2 Speicherbereiche:
- 1MB Batteriegepufferter SRAM von 0x00000000 bis 0x000FFFFC, als ROM bezeichnet
- 4MB SRAM von 0x00400000 bis 0x007FFFFC, als RAM bezeichnet

2.1 Exceptionvektoren

Am Anfang des ROM Bereichs befindet sich hardgecodete Vektoren des Prozessorkerns für Exception, Interrupts und dem Reset.

Diese sind wie folgt definiert:

Exception	**Vektor**
Reset	0x00000000
Divison Unsigned Emulation	0x00000080
Division Emulation	0x000000A0
Multiplikation Unsigned Emulation	0x000000C0
Multiplikation Emulation	0x000000E0
Interrupt	0x00000100
Integer Overflow	0x00000120
Undefined Instruction	0x00000140
Syscall	0x00000160
Break	0x00000180

2.2 Sections

Weiterhin erzeugt der Compiler für verschiedene Speicherbereiche Code und Daten, den sogenannten Sections.

.text
Enthält den eigentlichen Programmcode.

.rodata
Die Section für read only Daten wie Konstanten und Strings.

.data
Hier befinden sich mit ungleich Null vorinitialisierte globale oder static Variablen.

.bss
Hier befinden sich auch globale oder static Variablen wieder, aber diejenigen die mit Null initialisiert sind.

.common
Schlussendlich die globalen oder static Variablen die deklariert, aber nicht initialisiert sind.
Diese werden beim Linkerscript zu .bss gehören und daher auch mit Null initialisiert sein zur Laufzeit.

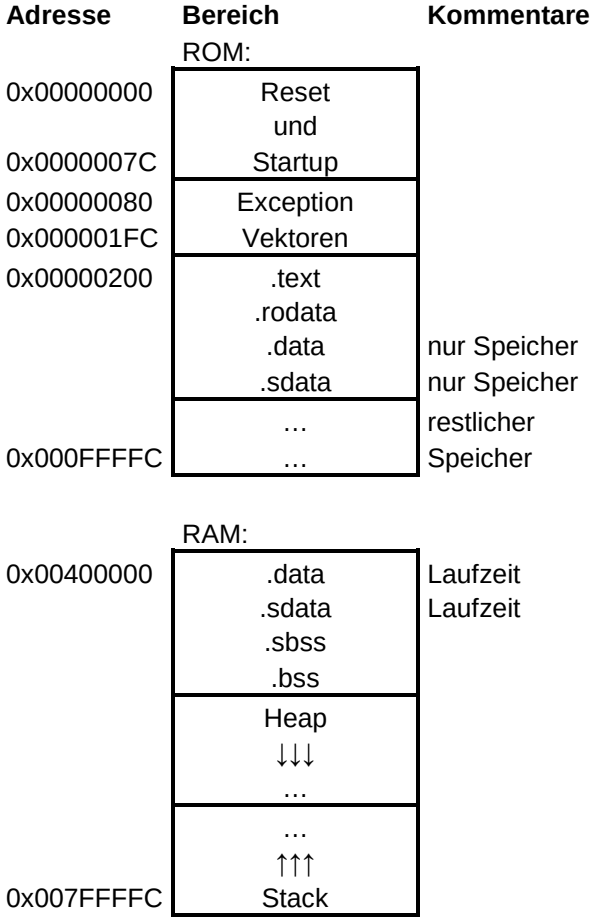
Weiterhin besitzt der MIPS noch small data und small bss auf diese mit Hilfe des Global Pointer Register zugegriffen wird. Hier finden Variablen mit jeweils unter 8Byte Größe ihren Platz für den schnelleren Zugriff. Diese Sections heißen .sdata und .bss

Weiterhin wird noch ein Heap benötigt für dynamische Daten (malloc) sowie ein Stack zum Register sichern bei dem Aufruf von Unterfunktionen.

2.3 Speicherlayout

Die Exceptionvektoren, die Sections sowie Heap und Stack müssen nun in den beiden Speicherbereichen untergebracht werden.

Damit ergibt sich folgendes Speicherlayout:



Im ROM Bereich werden alle Read Only Sections vorgehalten, hier befinden sich aber auch .data / .sdata
Diese Sections werden allerdings für den RAM Bereich gelinkt, aber im ROM Bereich gespeichert und beim Startup kopiert.
Dies hat den Hintergrund, dass in .data zwar bereits die Initialisierungen gespeichert sind, aber zur Programmlaufzeit ändern diese sich mit Sicherheit. Bei einem Neustart / Reset würde das Programm also nicht mehr die vorgesehenen Startbedingungen vorfinden. Nach dem kopieren von dem ROM in den RAM Bereich nach dem Startup findet das Programm immer seine definierten Startbedingungen vor.

3. Linkerscript

Damit der Linker des Compilers auch das gewünschte Binary für das Speicherlayout erzeugt, muss ihm das per Linkerscript mitgeteilt werden.

Dieses Script besteht aus 2 Hauptkomponenten, ENTRY und SECTIONS. Die ENTRY verweist nur darauf wo der Startpunkt des Programms ist, in unserem Falle 0x00000000, also der Reset Vektor und dort ist dann auch das ENTRY Symbol _start in der start.s deklariert.
Der SECTIONS Teil legt die Compilersections fest. Wie die Vektoren und die Datenbereiche.

3.1 Exceptionvektoren

Die Exceptionvektoren müssen an bestimmten Speicheradressen liegen, daher muss jeder einzelner Vektor an seine Adresse gelegt werden und bekommt seine eigene .text Untersection wie z.B. .text.reset. Diese kann in der start.s angegeben werden und so landet der Codeblock an dieser Adresse.

```
SECTIONS
{
	/*resetvector*/
	. = 0x00000000;
	reset : {
		*(.text.reset)
	}
	/*div unsigned*/
	. = 0x00000080;
	divu : {
		*(.text.divu)
	}
	/*div signed*/
	. = 0x000000A0;
	div : {
		*(.text.div)
	}
	/*mul unsigned*/
	. = 0x000000C0;
	multu : {
		*(.text.multu)
	}
	/*mult signed*/
	. = 0x000000E0;
	mult : {
		*(.text.mult)
	}
	/*ext interrupt*/
	. = 0x00000100;
	eirq : {
		*(.text.eirq)
	}
	/*overflow*/
	. = 0x00000120;
	ovw : {
		*(.text.ovw)
	}
	/*undefined instruction*/
	. = 0x00000140;
	undef : {
		*(.text.undef)
	}
	/*syscall exception*/
	. = 0x00000160;
	sysc : {
		*(.text.sysc)
	}
	/*break exception*/
	. = 0x00000180;
	break : {
		*(.text.break)
	}
}
```

3.2 ROM Bereich

Ab der Adresse 0x00000200 liegt der Programmbereich. Dabei sind .text und .rodata noch verständlich und offensichtlich angelegt. Die Sections .data und .sdata sind jedoch nur als Labels zu sehen aber einer Adresse hinter .rodata. Wobei .sdata hinter .data liegt (erkennbar an dem + _data_size). Ab hier beginnt die Linkerscript Magic.
Durch einen , in dem Unterkapitel RAM beriech erklärten, Linkerscriptbefehl werden zwar diese Sections im RAM Bereich gelinkt, aber im Binary an die Labels _rom_data_start und _rom_sdata_start gespeichert.

```
SECTIONS
{
	/*main*/
	. = 0x00000200;
	.text : {
		*(.text.startup)
		main.o (.text)
		*(.text)
		*(.text.phandling)
	}

	.rodata : {
		*(.rodata)
		*(.rodata*)
		*(.MIPS.abiflags)
		*(.eh_frame)
	}

	/*Hier liegen data und sdata zum Kopieren in den RAM*/
	_rom_data_start = .;
	_rom_sdata_start = . + _data_size;
}
```

3.3 RAM Bereich

Zuerst wird die Adresse des Linkers in diesem Bereich auf 0x00400000 gesetzt, denn ab jetzt sollen alle Sections ab dort anfangen.
Im RAM Bereich werden die Sections von .data und .sdata angelegt und mit dem AT(Label) wird dem Linker mitgeteilt, dass diese Sections zwar für die Adressen des RAM gelinkt werden sollen. Also das Programm wird auf diese Adressen zugreifen, aber der Inhalt liegt erstmal im ROM Bereich des Binarys. Weiterhin werden Labels erstellt, welche Start, Ende und Größe der Sections markieren, damit die start.s darauf zugreifen kann.

Nach dem .sdata wird der Global Pointer festegelegt, dieser befindet sich zwischen .sdata und .sbss, damit der signed Offset des Pointers (MIPS Ladebefehl) maximal ausgenutzt werden kann.
Die .sbss und .bss Section muss beim Startup mit Null überschrieben werden, daher auch hier wieder Labels für Start, Ende und Größe damit die start.s darauf zugreifen kann.
Schlussendlich wird noch das Label „end“ gesetzt, ab hier beginnt dann der Heap.

```
SECTIONS
{
	/*Ab hier liegt alles im RAM
	initialisierte Daten werden kopiert
	damit der Initwert beim Neustart erhalten bleibt.
	bss wird genullt zum startup*/
	. = 0x00400000;

	.data : AT( _rom_data_start ) {
		_data_start = .;
		*(.data)
		*(.data*)
		_data_end = .;
	}
	_data_size = _data_end - _data_start;

	.sdata : AT( _rom_sdata_start ) {
		_sdata_start = .;
		*(.sdata)
		*(.sdata*)
		_sdata_end = .;
		_sdata_size = _sdata_end - _sdata_start;
	}

	/*.sbss und .sdata benutzt das Global Pointer Register
	daher passendes Symbol in die Mitte setzen fuer start.s*/
	_gp = .;

	_sbss_start = .;
	.sbss : {
		*(.sbss)
		*(scommon)
		*(.scommon)
	}
	_sbss_end = .;
	_sbss_size = _sbss_end - _sbss_start;

	_bss_start = .;
	.bss : {
		*(.bss)
		*(common)
		*(.common)
	}
	_bss_end = .;
	_bss_size = _bss_end - _bss_start;

	/*Nach dem Programm und seinen statischen Daten
	kommt der Heap mit den dynamischen Daten*/
	end = .;
}
```

4. Startup

In der start.s befindet sich der Startup Bereich ab dem Reset Vektor und später die Verteilung der Exception Vektoren.

In dem Startup werden zuerst Stack und Global Pointer gesetzt. Danach werden die .sbss und .bss Section mit Nullen überschrieben. Anschließend werden mit Hilfe der Labels aus dem Linkerscript die .data und .sdata aus dem ROM in den RAM kopiert. Dafür werden die Labels in die Argument Register geladen und dann eine Funktion der libc aufgerufen. Schlussendlich wird in das Hauptprogramm gesprungen.

```
.section .text.reset
.extern main
_start:
	la $sp, 0x800000 /*Stack an 8MB -> Ende von den 4MB RAM*/
	la $gp, _gp

	/* data vom ROM in den RAM kopieren*/
	la $a0, _data_start
	la $a1, _rom_data_start
	li $a2, _data_size
	jal memcpy

	/* sdata vom ROM in den RAM kopieren*/
	la $a0, _sdata_start
	la $a1, _rom_sdata_start
	li $a2, _sdata_size
	jal memcpy

	/*.sbss Section mit 0 initilisieren*/
	la $a0, _sbss_start
	li $a1, 0
	li $a2, _sbss_size
	jal memset

	/*.bss Section mit 0 initilisieren*/
	la $a0, _bss_start
	li $a1, 0
	li $a2, _bss_size
	jal memset


	/*endlich gehts zum Hauptprogramm*/
	j main
```

Bei den Exceptionvektoren der Punktrechnungsemulation wird dann die entsprechnde Emulationsroutine angesprungen. Alle anderen Exceptionvektoren soringen bisher nur ins Hauptprogramm zurück, können später dann auch zu den passenden Handlern springen.

```
.section .text.div
div_vec:
      j divsim_sign

.section .text.divu
divu_vec:
      j divsim_unsign

.section .text.mult
mult_vec:
      j mulsim_signed

.section .text.multu
multu_vec:
      j mulsim_unsigned

.section .text.eirq
eirq_veq:
      rfe

.section .text.ovw
ovw_veq:
      rfe

.section .text.undef
undef_vec:
      rfe

.section .text.sysc
sysc_veq:
      rfe

.section .text.break
break_veq:
      rfe
```