Grundlagen der Rechnerarchitektur - Labor

Versuch 3: Zustandsmaschinen

In den ersten beiden Versuchen wurden kombinatorische und sequentielle Digitalschaltungen behandelt. In diesem Versuch wechseln wir wieder eine Ebene nach oben: Zustandsmaschinen. Diese bieten gegenüber den bisherigen Zähler-Dekoder-Steuerungen die Möglichkeit auch auf Eingaben zu reagieren. Technisch wird ebenfalls ein Schritt nach oben vollzogen: es werden sogenannte FPGAs eingeführt, die man mit Logikfunktionen programmieren kann.

Entwurf von Zustandsmaschinen

In der Literatur findet man viele Beschreibungen, wie man Zustandsmaschinen entwirft. Leider auch viele, die im Allgmeinen nicht funktionieren. Im Folgenden wird deshalb eine Beschreibung angegeben, die zuverlässig funktioniert, vollständig alle nötigen Schritte behandelt, allerdings gegenüber vielen anderen Beschreibungen recht lang wirkt. Ich empfehle trotzdem dringend dieser Anleitung zu folgen!

Eine Zustandsmaschine entwirft man, um eine Problemstellung zu lösen. Aus der Problemstellung leitet man die Ein- und Ausgaben für die zu entwerfende Zustandsmaschine ab. Interne Zustände müssen während dem Entwurfsprozess passend gewählt werden.

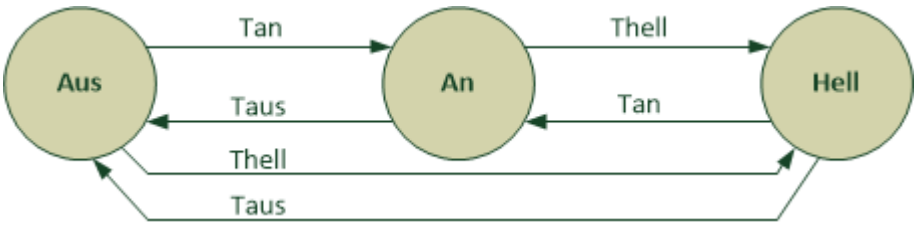
Zur Erklärung wird als Beispiel eine Zustandsmaschine für eine Taschenlampe entwickelt. Diese besitzt drei Tasten "Aus", "An" und "Extrahell". Diese drei Tasten sollen zwei LEDs ansteuern, wobei "An" nur eine LED einschaltet.

Zustandsdiagramm erstellen

Zustandsdiagramme bestehen aus Zuständen, die durch Kreise dargestellt werden, und Zustandsübergängen, Pfeilen zwischen den Kreisen. Die Zustände sollten Anfangs nur sprechende Namen haben (in die Kreise schreiben, binäre Kodierungen werden erst viel später zugewiesen). Die Pfeile für die Zustandsübergänge kann man Anfangs erst mal mit einer gedachten Bedingung belegen: "Wenn Taste 3 gedrückt". Während des Entwurfs werden daraus präzise, boolesche Bedingungen.

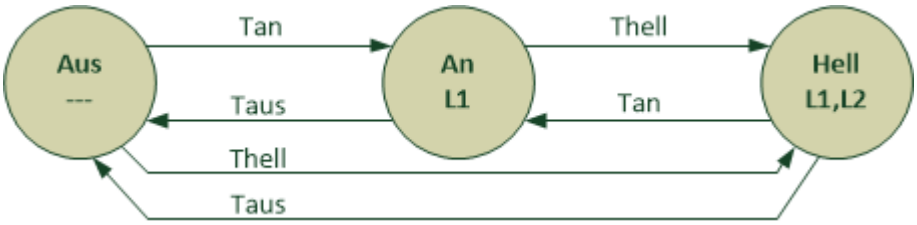
Der erste Schritt ist es, ein Zustandsdiagramm zu erstellen, dessen wichtigste Aufgabe es ist, das vorgegebene Problem anschaulich zu lösen. Man "malt" also ein Zustandsdiagramm, von dem man hofft, dass es das Problem löst, und verfeinert es danach so lange, bis es das wirklich tut. Besprechen Sie Ihre Lösung mit Ihrem Praktikumspartner und fragen Sie Ihren Tutor! Findet man Fehler erst später, oder findet sogar ein grundsätzliches Problem nachträglich, dann hilft nur eines zuverlässig: alles bisherige ignorieren, zurück auf Anfang, mit neuem Zustandsdiagramm von vorne beginnen.

Für das Beispiel mit der Taschenlampe könnte folgender, erster Entwurf eines Zustandsdiagramms entstehen: drei Zustände Aus, An und Hell entsprechen den drei geplanten Betriebszuständen. Die Tasten steuern ihrem Namen nach die Maschine in den entsprechenden Zustand:



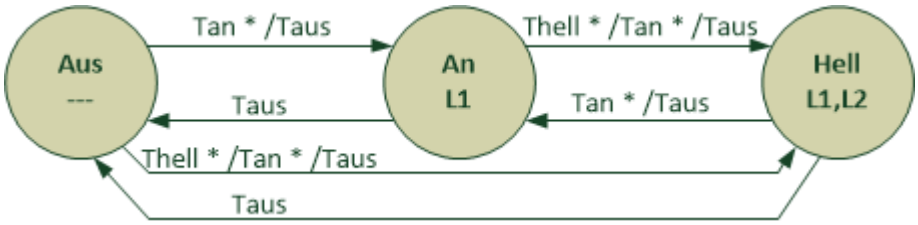
Ausgabewerte schreibt man zu den Zustandsnamen in die Kreise hinein, und zwar nur die Namen solcher Ausgangswerte, die in diesem Zustand log. 1 sein sollen. Die Ausgabewerte im Beispiel sind die

Ansteuersignale für die beiden LEDs L1 und L2. Die Minuszeichen unter "Aus" sollen andeuten, dass hier kein Signal log. 1 sein soll, aber dies Absicht ist. (Leer könnte bedeuten, dass man hier die Ausgaben vergessen hat.)



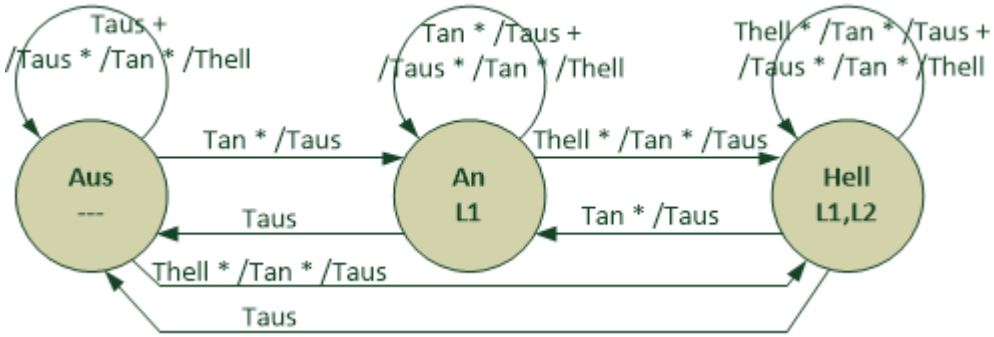
Bis hier darf man relativ frei arbeiten, aber nun muss das Zustandsdiagramm präziser werden:

- Die Bedingungen an den Pfeilen müssen boolesche Terme sein. Dazu müssen alle Eingabewerte eindeutige Namen haben, und die Bedingungen an den Zustandsübergängen dürfen nur noch boolesche Terme dieser Eingabewerte sein. Im Beispiel heißen die Eingabewerte schon präzise "Tan", "Thell" und "Taus". Boolesche Operationen zur Verknüpfung der Eingangssignale sind (bisher) noch nicht vorhanden.
- Die booleschen Bedingungen aller abgehenden Pfeile eines Zustands müssen sich gegenseitig ausschließen. Für jeden Zustand einzeln prüfen!
  Im Beispiel existieren solche Fehler: werden z.B. im Zustand "An" die beiden Tasten Taus und Thell gleichzeitig gedrückt, dann würden zwei Pfeile auf einmal aktiv!
  In der Praxis äußert sich ein derartiger Fehler übrigens nicht brav damit, dass einer der beiden Zuständsübergänge zufällig ausgewählt wird, sondern ein beliebiger Zustand erreicht werden kann. Sogar einer, der gar nicht im Zustandsdiagramm vorgesehen ist! Fehler erzeugen keine logisch rückverfolgbaren Effekte, sondern die ganze Statusmaschine "spinnt" einfach. Sorgfältig arbeiten und den Entwurfspfad kontrollieren sind die einzigen wirksamen Hilfsmittel.
  Damit Taus und Thell sich nicht widersprechen vergeben wir Prioritäten, welche Taste jeweils zu bevorzugen ist (Erweiterung bzw. Präzisierung der Aufgabenstellung): Taus sei wichtiger als Tan sei wichtiger als Thell. Damit ist entschieden, dass, wenn im Zustand "An" Taus und Thell gleichzeitig gedrückt werden, Taus wirken soll. Der Übergang nach "Hell" muss unterdrückt werden, was erreicht wird, indem die Bedingung von (Thell) geändert wird zu (Thell * / Taus). Genauer: weil Tan auch wichtiger ist braucht man: (Thell * /Tan * /Taus). Überarbeitet man alle booleschen Bedingungen entsprechend erhält man:

  Es fällt auf, dass nur drei Bedingungen vorkommen: (Taus), (Tan * /Taus) und (Thell * /Tan * /Taus). Diese entsprechen anschaulich den Prioritäten der Tasten.

- Der nächste Schritt ist, dass die booleschen Bedingungen aller abgehenden Pfeile eines Zustands derart sein müssen, dass immer genau ein Zustandsübergang aktiv wird. Dies beinhaltet die vorige Regel, verlangt jetzt aber zusätzlich, dass *immer* ein Pfeil gefunden wird, dem gefolgt werden soll. Diese Regel wird besonders gerne mit dem Argument verletzt: "soll einfach im alten Zustand bleiben". Das ist in der Entwurfsphase praktisch und sinnvoll, aber eine Zustandsmaschine macht das nicht von alleine.
  Das Beispiel wird deshalb wie folgt ergänzt:



Die Prüfung für den Zustand "Aus": wird immer genau ein Pfeil (Zustandsübergang) gefunden:

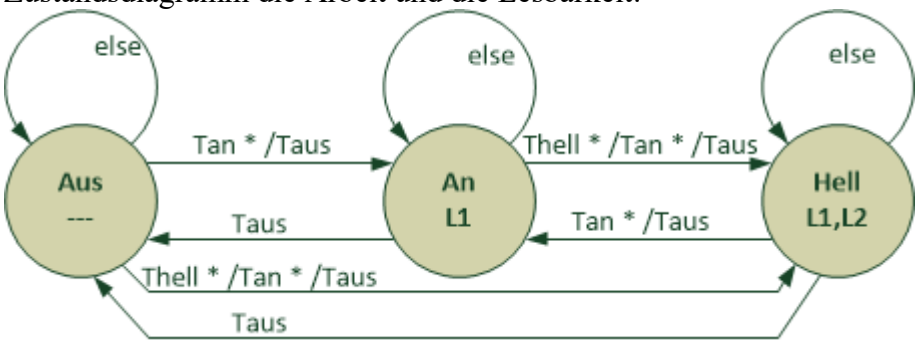
Thell	Tan	Taus	Zustandsübergang nach
1	0	0	HELL
X	1	0	AN
X	X	1	AUS
0	0	0	AUS

Die Wahrheitstabelle muss vollständig sein, d.h., es müssen alle möglichen Eingangskombinationen abgedeckt werden, und genau ein Folgezustand muss jeweils erreicht werden. (Hinweis: ein do-not-care X auf der Eingangsseite bedeutet, dass zwei Zeilen zusammengefasst sind, die sich nur darin unterscheiden, dass das X 0 oder 1 ist. Zwei X bedeuten, dass hier vier Zeilen zusammengefasst sind. Letztlich stehen damit in der Tabelle die für drei Eingangsvariablen zu erwartenden acht Zeilen.)

Was hier ergänzt wurde ist auch wieder anschaulich interpretierbar: ein Zustand muss erhalten bleiben, wenn

- genau die Taste gedrückt ist, die den Zustand herbeigeführt hat
  ODER
- keine Taste mehr gedrückt ist.

- Statt jeweils die fehlende Bedingung auszurechnen darf ein Pfeil pro Zustand die Bezeichnung "else" erhalten. Dieses else wird später in der Wahrheitstabelle expandiert, erleichtert aber im Zustandsdiagramm die Arbeit und die Lesbarkeit:



- Manchmal passiert es, dass die booleschen Bedingungen an den Pfeilen auch Zustandsbits der Zustandsmaschine enthalten. Wenn das passiert und irgendwie beim Denken hilft ist das in der Entwurfsphase akzeptabel, allerdings ist das dann kein richtiges Zustandsdiagramm! Man muß sich deshalb klarmachen, was denn eigentlich die Zustände sind, und letztlich das ganze Umzeichnen: in den Kreisen sind Zustände, an den Übergängen nur boolesche Terme von Eingangsvariablen!

Wahrheitstabelle aufstellen

Ist das Zustandsdiagramm fertig, dann erstellt man die Wahrheitstabelle für den Automaten. Wie immer erhält die Überschrift links die Eingabewerte inklusive dem alten Zustand. Rechts davon folgen neuer Zustand und Ausgabewerte. Die Tabelle selbst füllt man aus, indem man die Werte aus dem Zustandsdiagramm abliest.

Tipps dazu:

- Wahrheitstabellen für Zustandsmaschinen werden meist recht groß: auf einem frischen Blatt oben anfangen!
- Das Prüfen, ob jeder Zustand *immer* einen Zustandsübergang findet, erfolgte oben Anhand einer kleinen Wahrheitstabelle. Dies kann man in den nun folgenden Schritt hineinziehen, indem man beim Aufstellen der großen Wahrheitstabelle jetzt Zustand für Zustand voranschreitet, und dabei jeweils prüft, ob die Wahrheitstabelle für einen Zustand vollständig ist ehe man mit dem nächsten weitermacht.
- Man beachte, dass die Zustände immer noch ihre Namen tragen. Platz lassen für die später hinzukomende binäre Kodierung!

alter Zustand	Platzhalter	Thell	Tan	Taus	neuer Zustand	Platzhalter	L1	L2
AUS		1	0	0	HELL		0	0
		X	1	0	AN		0	0
		X	X	1	AUS		0	0
		0	0	0	AUS		0	0
AN		1	0	0	HELL		1	0
		X	1	0	AN		1	0
		X	X	1	AUS		1	0
		0	0	0	AN		1	0
HELL		1	0	0	HELL		1	1
		X	1	0	AN		1	1
		X	X	1	AUS		1	1
		0	0	0	HELL		1	1

Der alte Zustand ist nur ausgeschrieben, wenn er sich gegenüber der vorigen Zeile ändert. Da man Zustandsweise vorgeht spart dies viel Schreibarbeit und verbessert die Lesbarkeit. Insbesonder kann man Zeilen schneller dem vorhergehenden Zustandsdiagramm zuordnen.

Merkwürdig erscheint in der ersten Zeile, dass auf der Ergebnisseite der neue Zustand "HELL" ist, aber beide LEDs noch aus sind. Beim Entwurf von Moore-Maschinen dürfen Ergebnisse aber nur vom EingangsZUSTAND abhängen, und der ist noch "AUS". (Ansonsten entsteht eine Mealy-Maschine, die wir im Praktikum nicht behandeln wollen.)

Zurück zum Entwurfsverfahren. Als nächstes werden die Zustandskodierungen gewählt. Minimal benötigt man zur Unterscheidung von n Zuständen ld(n) (ld==Logarithmus Dualis, d.h., Logarithmus zur Basis zwei) Flip-Flops, aufgerundet. Bei 3 Zuständen sind das 2 FFs. (Bei bis zu 8 Zuständen 3, bei bis zu 16 dann 4 FFs, usw.) Diese Anzahl ist aber nur das Minimum. Mit Erfahrung "sieht" man manchmal, dass mehr FFs die restliche Logik deutlich vereinfachen. Z.B. das Verfahren "One Hot" nutzt ein Flip-Flop pro Zustand, und die Zustandskodierungen haben alle genau ein log. 1 mit allen anderen Zustandsbits log. 0. Es investiert mehr Flip-Flops zugunsten von hoffentlich weniger kombinatorischer Logik.

Die Auswahl von internen Kodierungen für die Zustände passiert am besten, indem man sich die Wahrheitstabelle anschaut und nach "guten" Kodierungen sucht. Erfahrung hilft, aber es bleibt ein bisschen eine Kunstform. Allerdings gibt es auch einen einfachen Algorithmus, der immer funktioniert: man wähle eine beliebige, zufällige Zuordnung! Schlimmstenfalls fällt die Zustandsmaschine ein bisschen komplexer aus als nötig.

Noch ein Tipp: oft ist es geschickt, wenn man die internen Zustände so wählt, dass die internen Zustandsbits direkt Ausgangswerten entsprechen: so entfällt der kombinatorische Dekoder von den internen Zuständen zu den Ausgangswerten!

In diesem Sinne wählt man beim Taschenlampen-Beispiel als interne Kodierung vorteilhaft die Ausgabewerte der Spalten L1 und L2 passend zur alten Zustandskodierung, also AUS==00, AN==10, HELL==11. Damit ergibt sich die vollständige Wahrheitstabelle wir folgt:

alter Zustand	Z1 Z2	Thell	Tan	Taus	neuer Zustand	Z1 Z2	L1	L2
AUS	0 0	1	0	0	HELL	1 1	0	0
	0 0	X	1	0	AN	1 0	0	0
	0 0	X	X	1	AUS	0 0	0	0
	0 0	0	0	0	AUS	0 0	0	0
AN	1 0	1	0	0	HELL	1 1	1	0
	1 0	X	1	0	AN	1 0	1	0
	1 0	X	X	1	AUS	0 0	1	0
	1 0	0	0	0	AN	1 0	1	0
HELL	1 1	1	0	0	HELL	1 1	1	1
	1 1	X	1	0	AN	1 0	1	1
	1 1	X	X	1	AUS	0 0	1	1
	1 1	0	0	0	HELL	1 1	1	1
HUCH	0 1	X	X	X	AUS	0 0	0	0

Die letzte Zeile ist für die Robustheit. Die Zustandsmaschine kann beim Einschalten oder auch im Betrieb durch einen Störimpuls in den Zustand 01 geraten, der eigentlich nicht vorkommen soll. Es ist gute Praxis, wenn man alle nicht benötigten Zustände erfaßt und ein definiertes Verhalten mit einbaut. In diesem Fall wurde die Entscheidung getroffen, dass die Lampe definiert vom falschen Zustand 01 in den Zustand AUS fallen soll.

Und nun ist man endlich soweit, dass man die Gleichungen ablesen kann. Für jede Ausgabespalte sucht man die Zeilen, in denen eine 1 ausgegeben werden soll. Aus diesen Zeilen liest man die Eingabekombinationen ab, und ver-oder-t diese. Somit entsteht für jedes Ausgabesignal eine Summe von Produkten. Als Syntax ist im Folgenden VHDL gewählt. Die Spaltentreue hilft bei der Fehlersuche, falls dies nötig wird.

```
Z1_neu <= (not Z1 and not Z2 and     Thell and not Tan and not Taus)
       or (not Z1 and not Z2 and                   Tan and not Taus)
       or (    Z1 and not Z2 and     Thell and not Tan and not Taus)
       or (    Z1 and not Z2 and                   Tan and not Taus)
       or (    Z1 and not Z2 and not Thell and not Tan and not Taus)
       or (    Z1 and     Z2 and     Thell and not Tan and not Taus)
       or (    Z1 and     Z2 and                   Tan and not Taus)
       or (    Z1 and     Z2 and not Thell and not Tan and not Taus);

Z2_neu <= (not Z1 and not Z2 and     Thell and not Tan and not Taus)
       or (    Z1 and not Z2 and     Thell and not Tan and not Taus)
       or (    Z1 and     Z2 and     Thell and not Tan and not Taus)
       or (    Z1 and     Z2 and not Thell and not Tan and not Taus);

L1 <= Z1;  -- muss man wohl gar nicht mehr im FPGA ablegen, direkt Z1 benutzen

L2 <= Z2;  -- ditto
```

Es bleibt dem Leser überlasen, wie man Z1, Z1_neu, Z2, ... an Flip-Flops anschließt oder in einer Zustandsmaschine mit 'Process' benutzt.

Die fertigen Gleichungen kann man von Hand optimieren, wie man dies in den ersten Versuchen gelernt hat, oder dies ab jetzt dem VHDL-Compiler überlassen.

Einführung: FPGAs

Im Praktikum wird das Board DE0-Nano des Herstellers Terasic verwendet. Dieses Test- und Entwicklungsboard enthält hauptsächlich ein FPGA vom Typ EP4CE22F17C6N des Herstellers Altera, aus dessen Cyclone IV Serie. Auf der Oberseite des Boards ist das FPGA sichtbar, erkennbar am großen Aufdruck "Altera".

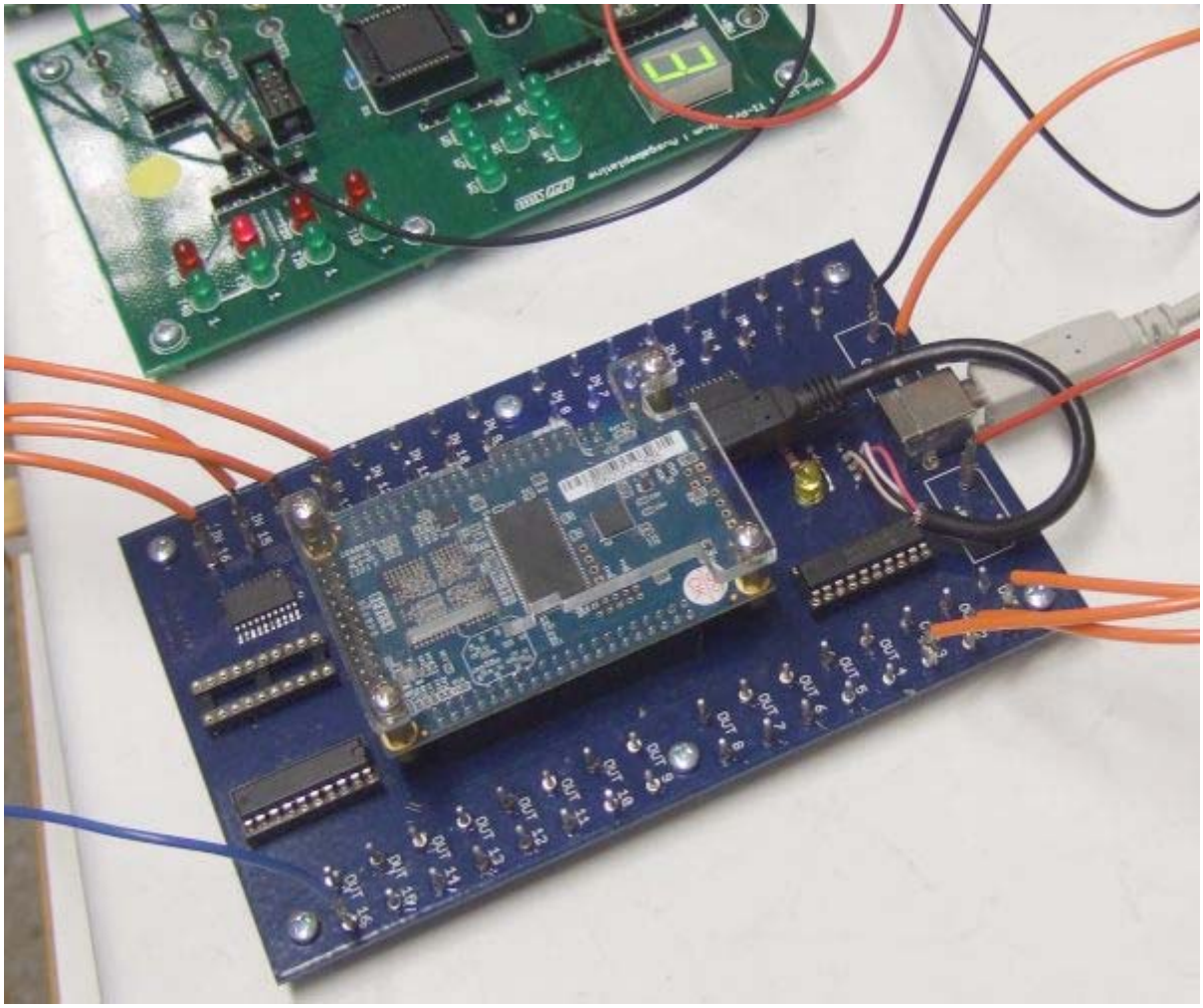
FPGAs sind ICs, die konfiguriert werden können. Typisch kümmert man sich nicht um die Details der Chips, sondern überläßt der Software die Aufgabe VHDL auf den Chip abzubilden. Trotzdem sollte man wissen, wie ein solches FPGA intern aufgebaut ist, schon um abschätzen zu können, was machbar ist.

Die grundlegendsten Elemente vom FPGA im Praktikum sind sogenannte Logik-Elemente, die aus vier booleschen Eingangswerten über eine beliebig konfigurierbare boolesche Funktion einen Ausgangswert erzeugen. Der Ausgangswert kann direkt oder über ein Flip-Flop an eine große, ebenfalls konfigurierbare Verdrahtungsmatrix gegeben werden, die die Logik-Elemente miteinander verknüpft. Das im Praktikum verwendete FPGA enthält 22320 solcher Logik-Elemente, was für (deutlich mehr als nur) einen 32-Bit Prozessor reicht.

Ein kleiner Überblick, was das FPGA noch alles kann, im Schnelldurchgang:

- Ein Logik-Element kann auch zum 2-Bit-Volladdierer werden, was sonst zwei Logik-Elemente benötigen würde. Dies ist geschickt, weil Addierer häufig vorkommen.
- Die Ein- und Ausgabepins des ICs sind an der globalen Verdrahtungsmatrix als besondere Elemente angeschlossen. Jeder Pin kann als Eingang in die Verdrahtungsmatrix eingespeist werden. Ebenso kann fast jeder Pin zum Ausgangspin werden, optional auch mit Flip-Flop. Soll ein Ausgangspin direkt den Wert eines Einganspins wiedergeben, dann kann dies alleine mit der Verdrahtungsmatrix erreicht werden, ohne dass auch nur ein Logik-Element verbraucht wird.
- Andere Elemente, die an der globalen Verdrahtungsmatrix angebunden sind, sind Speicher (Memory) und Multiplizierwerke, die sich nur schlecht mit einzelnen Logik-Elementen nachbilden lassen, aber recht häufig benötigt werden.
- Die Taktverteilung für die Flip-Flops ist im FPGA ebenfalls sehr flexibel konfigurierbar.
- Spezielle Ein-/Ausgabeblöcke unterstützen die Anbindung von immer wiederkehrenden externen ICs oder Protokollen, wie z.B. Speicherbausteine oder der bekannte PCI-Bus.

Im Praktikum sitzt das Entwicklungsboard auf einer Trägerplatine, die eine Anpassung der elektrischen Pegel zum restlichen Praktikum vornimmt, dabei aber auch gleichzeitig das relativ teure Entwicklungsboard vor Fehlern im Praktikum schützt. Das FPGA-Entwicklungsboard sitzt kopfüber gesteckt auf der Trägerplatine, so dass das FPGA nicht sichtbar ist. Gezeigt ist im Foto der Prototyp, d.h., das Serienboard im Praktikum mag leicht anders aussehen:



Am oberen Rand befindet sich der USB-Anschluss zum Programmieren des FPGAs. Hier muss ein USB-Kabel zum Entwicklungsrechner angeschlossen werden. Daneben sind die üblichen 5 Volt und GND Anschlüsse. Ansonsten sind 16 Eingänge IN1 bis IN16 und 16 Ausgänge OUT1 bis OUT16 vorhanden, wie immer im Praktikum mit je zwei Steckstiften.

VHDL und FPGA im Praktikum

Zur Programmierung eines FPGAs müssen nicht etwa einzelne Bits festgelegt werden. Vielmehr wird im Praktikum eine Untermenge der Sprache VHDL benutzt (VHDL==VHSIC Hardware Description Language, mit VHSIC==Very High Speed Integrated Circuits). Ein Compiler übernimmt die Aufgabe VHDL-Quelltext auf das FPGA abzubilden. Im Praktikum wird, bei erfolgreichem Compilieren, das Ergebnis automatisch in das FPGA übertragen.

Im Speziellen beginnt man eine Aufgabe am besten, indem man sich den Beispielquelltext in ein eigenes, neues Verzeichnis kopiert:

```
mkdir <mein_neues_verzeichnis>
cd <mein_neues_verzeichnis>
cp /opt/vhdl/v0.vhdl ./<mein_aufgabenname>.vhdl
```

Die eigene Quelltextdatei kann nun mit einem beliebigen Editor bearbeitet werden, z.B:

```
nano <mein_aufgabenname>.vhdl
```

(Für Aufgabe 0 kommt man auch ohne Editieren aus.)

Der Compiler übersetzt Quelltexte mittels des Kommandos vhdl:

```
vhdl <mein_aufgabenname>.vhdl
```

Nach einer gefühlten Minute, wenn der Quelltext fehlerfrei ist, dann sollte das FPGA sich nun entsprechend dem Quelltext verhalten. Leider klappt das manchmal nicht, insbesondere beim ersten Mal nach dem Einschalten. Dann folgende Schritte durchführen:

1. Der Rechner muß laufen
2. Das FPGA-Board muss Strom haben (Steckernetzteil)
3. Das USB-Kabel darf NICHT eingesteckt sein
4. Auf dem Rechner das Kommando "start_jtagd" ausführen
5. Das USB-Kabel einstecken
   → wenn jetzt der Rechner etwas meldet über ein gefundenes Gerät ist alles ok, sonst ab 3) nochmal
6. Ab jetzt sollte das Kommando "vhdl" wie oben beschrieben einfach funktionieren

Das Kommando "vhdl" ist im Praktikum speziell vorbereitet und nimmt den Namen des Quelltextes mit und ohne die Endung ".vhdl" an. Außerdem erkennt es das Schlüsselwort "clean". Statt clean.vhdl zu compilieren löscht es die vielen Zwischendateien, die der Compiler hinterläßt. Dabei geht es recht ruppig vor. Man sollte es daher wirklich nur in einem speziellen Unterverzeichnis für diesen Versuch benutzen.

Die Sprache VHDL im Praktikum

Im Praktikum wird nur ein Teil der Sprache VHDL verwendet, der hier beschrieben wird.

Die genannte Beispieldatei v0.vhdl dient als Muster, was man alles machen kann. Diese wird nun Abschnittsweise erklärt:

Kommentare sind "--" bis Zeilenende:

```
  -- bei vhdl ist alles von -- bis zeilenende Kommentar
```

Signale stellt man sich am besten als Leitungen/Kabel vor, denen man Namen gibt. Einige Signale sind vorgegeben, weitere kann man selbst deklarieren, einzeln oder als eindimensionales Array:

```
  -- vorbereitete Signale sind p_in(1) bis p_in(16) und p_out(1) bis p_out(16),
  -- passend zur Beschriftung auf der Platine des Praktikums

  signal zustand: std_logic; -- so deklariert man ein eigenes signal
  -- signal my_signal_array: std_logic_vector (5 downto 3); -- oder gleich mehrere
```

Ein Takt wird immer wieder im Praktikum benötigt. Deshalb gibt es das vorgegebene Signal "clock", dessen Geschwindigkeit man selbst wählen kann:

```
  -- vorbereitetes Signal "clock":
  -- auch dann drin lassen, wenn man keinen clock braucht!
  constant speed : integer := 15; -- geschwindigkeit von "clock" hier waehlen
  -- (maximum:) 25 ==> 0.75 Hz, 24 ==> 1.5 Hz, 23 ==> 3 Hz, ..., 1 ==> 25 MHz
  -- Sinnvoll: 25 == 0.75 Hz == langsam zum mitschauen
  --           23 == 3 Hz == alles blinkt flott
  --           15 == 768 Hz == nicht schneller als das im Praktikum !
```

Die wichtigste Zeile ist im Praktikum die Zeile "-- end declarations". Das vorbereitete Kommando "vhdl", das oben erwähnt wurde, mischt nämlich die Quelltextdatei mit einer anderen, vorgegebenen Quelltextdatei zu einem vollständigen vhdl-Programm zusammen, um Ihnen Arbeit zu ersparen. Die Zeile "-- end declarations" dient dem vhdl-Kommando dabei als Schnittpunkt: der erste Teil wird in den Deklarations-Teil des vhdl-Programms eingesetzt, der untere Teil in den Codebereich.

```
  -- die folgende Zeile, obwohl es kommentar ist, unbedingt drin lassen:
-- end declarations
```

Signalen kann man boolesche Terme mittels "<=" zuweisen. Die wichtigsten Operatoren sind hier "not", "and" und "or". "not" bindet höher als die anderen beiden, die gleichberechtigt sind. Um gewünschte Bindungen herzustellen benutzt man runde Klammern.

"nand", "nor", "xor" und "xnor" gibt es auch, jedoch sind diese untauglich, wenn mehr als zwei Operanden verknüpft werden sollen: "a nand b nand c" ergibt nicht ein nand-Gatter mit drei Eingängen, sondern "(a nand b) nand c".

Wichtig gegenüber sequentiellen Programmen: die Reihenfolge der Zeilen im Quelltext spielt keine Rolle, da man hier Verdrahtungen zwischen Gattern angibt, die immer Arbeiten, und gar nichts davon Wissen, dass es eine Reihenfolge im Quelltext gibt.

```
p_out(3) <= p_in(3);     -- pin out3 folgt pin in3
p_out(5) <= not p_in(6); -- pin out5 ist invertiert der eingangswert von in6
p_out(6) <= not p_in(3) or p_in(6); -- out6 := /in3 or in6
```

Wenn man nicht mehr mit kombinatorischen Signalen auskommt, sprich Flip-Flops benötigt, muß man auf das Konstrukt "process" von VHDL ausweichen. Die meisten Zeilen des Musters sollte man blind aus v0.vhdl übernehmen:

```
-- ein Beispiel, wie ein process aussehen kann:
--   das Beispiel schaltet bei jedem "clock" den Ausgang p_out(7) um
process (clock)                          -- immer so lassen
begin                                    -- immer so lassen
  if rising_edge(clock) then             -- immer so lassen
    zustand<='0';             -- hier beginnt der eigene Code
    if zustand='0' then
      zustand<='1';
    end if;
    -- zustand <= not zustand; -- alternative zum if-konstrukt darueber
  end if;                                -- immer so lassen
end process;                             -- immer so lassen

p_out(7) <= zustand;                     -- hier Ausgaben zuweisen
```

Die Zeilen, die man im Praktikum immer so lassen sollte, bewirken, dass eine Zustandsmaschine erzeugt wird (process), die mit clock getaktet wird, und zwar immer nur auf der steigenden Flanke.

Die Zeilen, die das Signal "zustand" beispielhaft benutzen ergeben keinen Sinn, wenn man sie sequentiell liest: das IF wäre immer true.

Vielmehr muß man hier so denken, wie folgendes Java-Programm andeutet:

```
// einige Zeilen ähneln denen im vhdl-code:
z_neu = '0';
if (zustand = '0') {
  z_neu = '1';
}

// eine Zeile kommt dazu:
zustand = z_neu;
```

Der VHDL-Process wird also einmal komplett durchlaufen. Erst danach werden die Signale auf den Endwert gesetzt, der vom Code davor bestimmt wurde. (Danach beginnt der Process wieder oben und wartet auf die nächste Flanke.)

Erreicht wird dieses Verhalten, indem der VHDL-Compiler den Rumpf des process komplett in kombinatorische Logik zerlegt, und die Ergebnisse an Flip-Flops gibt, die getaktet werden. Genau genommen: der Rumpf wird ständig kombinatorisch berechnet, das Ergebnis aber nur bei der passenden Flanke übernommen.

"Zustand" ist dabei immer noch ein Signal, aber eines, das von einem Flip-Flop angesteuert wird.

Das IF darf in VHDL auch mit ELSE und ELSIF erweitert werden. VHDL ignoriert den Unterschied zwischen Gross- und Kleinschreibung. IF und if ist also dasselbe. Ebenso sind Zustand, zustand und ZUSTAND alle identisch.

Noch ein VHDL-Beispiel:

```
  constant speed : integer := 23;
  signal taste : std_logic;
  signal reset : std_logic;
  signal zaehler : std_logic_vector (2 downto 0);
  signal led4 : std_logic; -- an, wenn zaehler==4
  signal led7 : std_logic; -- an, wenn zaehler==7
-- end declarations
  taste <= p_in(1);
  reset <= p_in(2);

  process(clock)
  begin
    if rising_edge(clock) then
      if reset='1' then       -- reset gewinnt: zaehler auf 0
        zaehler <= "000";
      elsif taste='1' then    -- zaehlen, wenn taste
        zaehler <= zaehler + 1;
      else
        zaehler <= zaehler;   -- selbsthaltung
      end if;
      if zaehler = "111" then
        led7 <= '1';
      else
        led7 <= '0';
      end if;
    end if;
  end process;

  -- ausserhalb von process geht nicht: if zaehler="100" then
  -- aber das hier erreicht genau dasselbe:
  led4 <= zaehler(2) and not zaehler(1) and not zaehler(0);

  p_out(1) <= led4;
  p_out(2) <= led7;
  p_out(3) <= not (zaehler(2) or zaehler(1) or zaehler(0));
```

Im Rahmen dieser Anleitung kann leider keine komplette Darstellung von VHDL gegeben werden, und deshalb wird an dieser Stelle auf Google verwiesen, in der Hoffnung, dass die gegebenen Erklärungen schon ausreichen, um das Praktikum vollständig bearbeiten zu können. Ansonsten gilt wie immer: Fragen Sie Ihren Betreuer!

Arbeiten mit Linux bzw. der Kommandozeile

Im Praktikum sind Rechner vorhanden, die Linux booten, und bei denen man sich mit regulären SGI-Accounts einloggen kann (ähnlich wie "xy13"). Die Rechner bieten allerdings nur eine Kommandozeile an, keine graphische Oberfläche. (Nein, der Rechner ist nicht defekt, weil die Maus fehlt.)

Eine kurze Einführung zur Linux-Kommandozeile für absolute Neulinge ist auf der Webseite des Praktikums abrufbar. Diese Einführung ist absichtlich sehr knapp gehalten, und explizit für solche Personen gedacht, die wirklich noch nie mit der Kommandozeile zu tun hatten. Es wird nur das Grundprinzip der Kommandozeile erklärt, damit das Grundverständnis da ist, und es werden einige, wenige Kommandos (ls, cp, logout, ...) erwähnt, damit man das Praktikum durchstehen kann. Google findet natürlich auch Einführungen in die Kommandozeile, die aber leider meist viel zu schnell Details erklären, die ganz am Anfang den Blick auf das Wesentliche versperren.

Aufgabe 0: FPGA in Betrieb nehmen

Nehmen Sie entsprechend der obigen Anleitung das FPGA in Betrieb, d.h., legen Sie ein Projekt an mit der Vorlage v0.vhdl, lesen Sie den Quelltext und leiten daraus eine geeignete Verkabelung mit dem Praktikumsmaterial ab. Probieren Sie aus, ob alle vorgegebene Beispiele funktionieren! Wenn dies klappt, fragen Sie Ihren Betreuer nach einer Aufgabenstellung für Aufgabe 1.

Aufgabe 1: 7-Segmentanzeige

Steuern Sie (wie im 1.Versuch) ein Segment der 7-Segmentanzeige an, nur verwenden Sie diesmal ein FPGA, das Sie natürlich vorher entsprechend programmieren müssen. Das anzusteuernde Segment wird vom Betreuer vorgegeben.

Man braucht für diese Aufgabe kein Statement "process", und deshalb ist es, um Sie vor sich selbst zu schützen, verboten!

Aufgabe 2: eine Zustandsmaschine selbst entwerfen

Vom Betreuer erhalten Sie eine Aufgabenstellung für eine Zustandsmaschine. Entwerfen Sie diese entsprechend der Art, wie sie oben in dieser Versuchsanleitung vorgeführt wird.

Insbesondere bedeutet das, dass Sie "process" nicht in Ihrem Quelltext verwenden dürfen. Da man aber Flip-Flops zur Speicherung des Zustands des Zustandsautomaten benötigt, sind auch 16 D-Flip-Flops vordefiniert. Die Eingänge heissen d_in(1) bis d_in(16) und die Ausgänge d_out(1) bis d_out(16). Der Takt aller dieser Flip-Flops ist fest mit "clock" verbunden. Diese 16 Flip-Flops dürfen aber nur in dieser Teilaufgabe verwendet werden, in allen anderen sind sie verboten.

Hinweis, auch für Aufgabe 3: die Zustandsmaschinen laufen alle darauf hinaus, dass Tasten(-abfolgen) erkannt werden müssen. Dabei sind, neben der Reihenfolge der Tasten immer die Nebenprobleme: ist die Taste noch gedrückt? Ist zwischenzeitlich gar keine Taste gedrückt? Jeder vermeintlich simple Tastendruck muss also registriert werden als Abfolge von wird-gedrückt, ist-noch-gedrückt, nichts-ist-mehr-gedrückt-warten-auf-nächste-Taste-und-dabei-zustand-halten.

Aufgabe 3: eine Zustandsmaschine von VHDL synthetisieren lassen

Vom Betreuer erhalten Sie eine Aufgabenstellung für eine Zustandsmaschine, und zwar eine andere als in Aufgabe 2. Diese entwerfen Sie mittels "process" in vhdl.

Grundlagen der Rechnerarchitektur - Labor

Versuch 4: Assemblerprogrammierung mit MIPS

In diesem Versuch wird Assemblerprogrammierung am Beispiel der MIPS-Architektur betrachtet. Die MIPS-Architektur sollte aus der Vorlesung bekannt sein und wird in dieser Versuchsbeschreibung als bekannt vorausgesetzt.

Als Versuchsumgebung wird der VMIPS-Simulator bereitgestellt, der frei im Internet verfügbar ist (http://vmips.sourceforge.net/). Die Programme des Praktikums werden mit dem GNU-Assembler und -Linker erstellt. Alle diese Tools sind im Praktikum vorinstalliert.

Auf diesen Tools basierend gibt es auch ein komplettes Linux. Dies geht einher damit, das im Praktikum Werkzeuge aus der Praxis genutzt werden sollen.

Umgekehrt werden Lösungen vom MIPS-Simulator MARS nicht akzeptiert, weil er Einstellungen erlaubt, die Realitätsfremd sind. Zur Lösungsfindung und/oder Fehlersuche darf man ihn aber natürlich nutzen. Für Vorlesung und Übungen ist er im Windows-Pool sowieso vorinstalliert.

Überblick bzw. Lernziele

Das wichtigste Lernziel dieses Versuchs ist, dass Sie ein wenig Erfahrung mit Assemblerprogrammierung bekommen sollen.

Daneben sollen die Werkzeuge vorgestellt werden, die benutzt werden, um Code zu erzeugen, d.h., wie man vom Quelltext zum laufenden Programm kommt. Einen Quelltext erstellt man mit einem beliebigen Editor, der dann assembliert, und danach gelinkt werden muss. Für den VMIPS-Simulator folgt danach ein Zwischenschritt, der das fertige Maschinenprogramm in ein besonderes Format, nämlich das rom-Format des VMIPS-Simulators umwandelt. Dann kann das Programm gestartet werden, bei VMIPS im Simulator. Die Details hierzu werden weiter hinten im Abschnitt **Arbeiten mit dem VMIPS Simulator** beschrieben.

Des weiteren soll betrachtet werden, wie Compiler Hochsprachenanweisungen mittels Schablonen in Assembler umsetzen. Hierzu folgt gleich ein Abschnitt **Muster für Kontrollstrukturen**.

Auch wie Compiler mit Prozessorregistern arbeiten soll betrachtet werden. Für Funktions- bzw. Unterprogrammaufrufe gibt es oft Architektur-spezifische Konventionen bezüglich der Datenübergabe und für das Aufrufprozedere; dies ermöglicht das Zusammenspiel zwischen kompiliertem Code verschiedener Compiler/Sprachen auf einer Plattform. Siehe hierzu **Register- und Stackkonventionen bei MIPS** in dieser Anleitung. Dieses Kapitel enthält indirekt auch die Schablone, wie Compiler Funktionsaufrufe in Maschinencode umsetzen.

Zuletzt sollen die Begriffe **Compilezeit** und **Laufzeit** vertieft werden. Gemeint sind damit die Zeiten, zu denen der Compiler läuft bzw. die Zeit in der das fertig erzeugte Programm läuft. Beispiel: Das Programm *input i; print i+3;* hat zur Compilezeit keine Ahnung, welchen Wert *i* hat. Zur Laufzeit hingegen ist es nicht mehr wichtig, dass die Variable *i* heißt; vielmehr muss der erzeugte Code einfach immer die Speicherzelle nutzen, die der Compiler für die Variable vorgesehen hat.

Muster für Kontrollstrukturen

Wie bereits angesprochen, benutzt ein Compiler feste Regeln und Schablonen zur Umsetzung von Hochsprachen-Konzepten wie beispielsweise für Schleifen, bei Array- und Datenstrukturen, und bei Funktions-Aufrufen mit Übergabe von Parametern.

Zum Beispiel könnte das Muster für eine **while**-Schleife

```
WHILE <ausdruck> DO <body> ENDWHILE
```

nach folgender Schablone in MIPS-Assemblercode umgesetzt werden:

```
<NEUES_LABEL_WHILE_AGAIN>:
        <Auswertung von ausdruck, Ergebnis als boolean in t0 erwartet>
        beq     t0,zero,<NEUES_LABEL_ENDWHILE>
        <Code für/von body>
        j       <NEUES_LABEL_WHILE_AGAIN>
<NEUES_LABEL_ENDWHILE>:
```

Aufgabe 0: Muster für andere Strukturen

VOR dem ersten Versuchstermin zu bearbeiten! Überlegen Sie, wie Kontrollstrukturen aussehen müssen für:

- if (...) { ... } else { ... }
- Arrayzugriff, also so etwas wie "f[i]" in "sum = sum + f[i]"
- Zugriff auf die Elemente einer Klassen-Instanz

Aufgabe 1: Einarbeitung in die Thematik

Schreiben Sie ein Programm, das zwei vorzeichenlose, ganze Zahlen einliest, addiert und die Summe ausgibt.

Diese Aufgabe ist trivial und dient dem Kennenlernen der Versuchsumgebung. Beachten Sie, dass es fertige **Unterroutinen zur Ein-/Ausgabe** gibt.

Aufgabe 2: Implementierung eines einfachen Algorithmus

Implementieren Sie folgenden, von Ihrem Betreuer vorgegebenen Algorithmus:

Der Fokus dieser Aufgabe liegt auf der Umsetzung gegebener Anweisungen und Kontrollstrukturen in Assemblercode.

Aufgabe 3: Rekursion

Implementieren Sie folgenden, von Ihrem Betreuer vorgegebenen rekursiven Algorithmus, und erhalten Sie dabei seine Struktur so weit wie möglich (Änderung in iterative Form ist definitiv nicht zulässig):

Schreiben Sie dazu ein kurzes, passendes Rahmenprogramm, welches die Eingabe(n) einliest, den eigentlichen Algorithmus aufruft und dessen Ergebnis anschließend ausgibt.

Benutzen Sie bei Ihrer Lösung auf jeden Fall Unter-/Funktionsaufrufe mit Parameterübergabe auf dem Stack, wobei Sie nicht zwingend die MIPS-Konventionen bezüglich der Stackbelegung einhalten müssen. Diese können allerdings Thema des Kolloquiums sein.

Aufgabe 4: Unterprogrammaufrufe

Setzen Sie ein kurzes C-Programm, welches Ihnen der Betreuer vorgibt, in Assembler-Code um. Die Umsetzung muss dabei derart erfolgen, dass das C-Programm als Kommentar im Assemblerquelltext steht, und zwar so, dass erkennbar ist, welche Assemblerzeilen welches C-Statement implementieren. Kleine Abweichungen von dieser Vorgabe sind unumgänglich, müssen aber alle einzeln (im Kolloquium) begründbar sein.

Benutzen Sie nach Möglichkeit Muster, wie sie in der Einleitung vorgestellt wurden.
Bei dieser Aufgabe ist die Verwendung der **MIPS Register- und Stackkonventionen** bei den Unterprogrammaufrufen **nicht** vorgeschrieben (und auch nicht angeraten). Da keine rekursiven Aufrufe erfolgen, können Sie einfach mit festen Registerbelegungen arbeiten.

Arbeiten mit dem VMIPS-Simulator

In diesem Kapitel wird beschrieben, welche Schritte nötig sind, um von einem Assembler-Quelltext zu einem laufenden Programm zu kommen. Es wird davon ausgegangen, dass ein Quelltext vorliegt, der mit einem beliebigen Editor erstellt wurde.

Ein solcher Quelltext muss zuerst "assembliert" werden, d.h., neben diversen anderen Aufgaben werden die Mnemonics wie "addu" in ihre binären Äquivalente umgesetzt. "addu $1, $2, $3" wird somit zu einem 32-bit Binärwert.

Nach dem Assemblieren muss der Linker gestartet werden. Linken ist der Vorgang, bei dem vor allem Aufrufe zwischen getrennten Übersetzungseinheiten aufgelöst werden. Beispielsweise bei einem Aufruf Ihres Codes an "readline", einer der "Unterroutinen zur Ein-/Ausgabe", kann der Assembler nicht den fertigen Code erzeugen, da ihm nicht bekannt ist, wo diese Routine im Speicher zu finden ist. Der Linker hat die Aufgabe allen Routinen Plätze im Speicher zuzuweisen und dann die Aufrufadressen im Code nachzutragen. Konnte der Linker alle solchen Verweise auflösen, so liegt ein fertiges Programm vor, das auf einem Rechner mit MIPS-Prozessor lauffähig ist.

Der nächste Schritt ist speziell für den VMIPS-Simulator nötig und nicht typisch für Codegenerierung im Allgemeinen. Es wird der fertige Code in ein anderes, VMIPS-spezifisches Format konvertiert, das ROM-File genannt wird.

Zuletzt kann man den Simulator starten und dabei das ROM-File mit dem fertigen Code übergeben.

Noch einmal anders, etwas Praxisnäher: Zuerst erstellt man einen Quelltext in einer Datei mit einem Namen, der bei MIPS die Endung ".S" (großes S) haben sollte. Hat man beispielsweise einen Quelltext in "v4a1.S", dann führen folgende vier Kommandozeilen im Prinzip dazu, dass das Programm im Simulator läuft:

```
assemble v4a1.S   -o v4a1.o
link     v4a1.o   -o v4a1.elf
make-rom v4a1.elf    v4a1.rom
vmips    v4a1.rom
```

Dabei wird immer aus einer Eingabedatei eine Ausgabedatei erzeugt. Aus dem Quelltext v4a1.S wird das "object-file" v4a1.o, daraus das auf einem Linux-Rechner mit MIPS-Prozessor lauffähige Programm v4a1.elf, daraus das Format, das der VMIPS-Simulator benötigt v4a1.rom. Diese letzte Datei wird dem Simulator zur Ausführung übergeben.

Wirklich komplett sind die Kommandos so nicht. Für Details kann man bei Bedarf in das bereitgestellt Makefile (s.u.) hineinschauen.

Vor dem Assembler läuft noch ein Programm, das dafür sorgt, dass der Assembler die Prozessorregister nicht nur als Zahlen, sondern auch als Namen versteht. Man darf deshalb z.B. "fp" statt "$30" schreiben, was Quelltexte deutlich besser lesbar macht.

Dem Linker wird auch eine Steuerdatei übergeben, die ihm sagt, wie er genau arbeiten soll. Diese Steuerdatei wird von vmips vorgegeben, da nur dieser weiss, was er genau braucht. Ausserdem wird dem Linker die Bibliothek mit den von uns vorgegebenen Unterroutinen zur Ein- und Ausgabe übergeben, damit er die davon benutzten Routinen dem fertigen Programm hinzufügen kann.

Das Werkzeug "make" unter Unix ist ein mächtiges Hilfsmittel zur Automatisierung und speziell nützlich im Zusammenhang mit Programmierprojekten. Abläufe lassen sich damit gut automatisieren, und das sogar derart, dass nicht unbedingt alle sondern nur die nötigen Schritte durchgeführt werden, um ein Projekt komplett auf den aktuellen Stand zu bringen. Dies setzt natürlich voraus, dass dem Tool die Abhängigkeiten des Projekts bekannt gemacht werden. Dies geschieht im "Makefile", das parallel zu den Quelltexten des Projektes geführt wird.

Im Praktikum steht ein Makefile zur Verfügung, das wie folgt benutzt werden kann. Zuerst kopiert man (nur einmal nötig) alle Dateien von /opt/mips_versuch in ein eigenes Verzeichnis. Danach kann man einen MIPS-Quelltext, wie beispielsweise die vorgegebene Datei "test.S" (großes S), mittels

```
make test.rom
```

assemblieren, linken und in das rom-Format konvertieren, so dass im Anschluss das Kommando

```
./vmips test.rom
```

das Programm im Simulator zur Ausführung bringt.

Der VMIPS-Simulator kann, wie praktisch jedes Programm unter Unix, jederzeit mit der Tastenkombination Strg-C gestoppt werden. Dies ist z.B. nützlich, wenn das geladene Programm in einer Endlosschleife ist.

Arbeiten mit dem Assembler

Zur Erklärung des Assemblers wird das folgende Programm besprochen. Die Zeilennummern sind nicht Teil des Quelltextes und hier nur vorhanden, damit Zeilen im Text referenziert werden können.

```
 1      #include "asm_regnames.h"
 2              .text                   # los geht's bei "entry"
 3              .globl  entry
 4      entry:
 5              addi    $29,$29,-8      # Ruecksprungadresse sichern
 6              sw      $31,0($29)

 7              .data
 8      text_1:
 9              .asciiz "Eine Taste druecken: "

10              .text
11              lui     $4,%hi(text_1)
12              ori     $4,%lo(text_1)
13              jal     writestring

14              jal     readchar        # Einen Tastendruck einlesen

15              or      a0,v0,$0        # das gelesene Zeichen wieder Ausgeben
16              jal     writechar

17              lw      ra,0(sp)        # zurueck zum Startprogramm/Programmende
18              addi    sp,sp,8
19              jr      ra
```

Assemblerprogramme sind zeilenorientiert mit einer Anweisung pro Zeile. Traditionsgemäß werden die Zeilen spaltentreu benutzt, d.h., IFs und Schleifen werden nicht eingerückt. Die erste Spalte nimmt Labels auf. Die zweite Spalte enthält Mnemonics für Maschinenbefehle oder Assemblerdirektiven, die folgende Spalte die Operanden dazu. Die vierte Spalte ist für Kommentare da.

Ein "Label" ordnet einer Position einen Namen zu, nämlich einen weitgehend frei wählbaren Bezeichner. Bei der Assemblerprogrammierung dienen Labels vielen verschiedenen Zwecken, jeweils mit dem Zweck, dass man nie mit Adressen, also Zahlen hantieren muss, sondern vielmehr der Assembler diese Zahlen jeweils passend verrechnet. Im fertigen Maschinenprogramm sind Labels nicht mehr bekannt. Vielmehr ist es eine der Aufgaben des Assemblers diese aufzulösen. Im Deutschen wird Label gerne mit "Sprungmarke" übersetzt, was aber nur eine der Funktionen von Labels beschreibt. In diesem Skript wird deshalb weiter der Begriff "Label" benutzt.

Das obige Programm beginnt beim Label "entry:" (Zeile 4). Im Praktikum ist fest vorgegeben, dass Programme bei diesem Label beginnen, so wie Java-Programme bei "main" beginnen. Übrigens wird im Simulator ein kleines Startprogramm, bestehend aus einigen MIPS-Befehlen zur Initialisierung des Simulators und für die Praktikumsumgebung ausgeführt, bevor der Sprung zu entry, also zu Ihrem Programm erfolgt.

Die Assemblerdirektive ".globl entry" (Zeile 3) sagt dem Assembler, dass der Bezeichner entry dem Linker mitgeteilt werden muss. Dies ist nötig, weil im Startprogramm ein Sprung "j entry" steht, der Wert dieses Labels aber während dem Assemblieren des Befehls noch gar nicht feststeht. Vielmehr entsteht der Wert für "entry" erst, wenn Sie ein bereits assembliertes Programm dem Linker übergeben. Der Linker weist dem Programm einen Platz zu, und berechnet dann auch alle darin vorkommenden Labels. Dies tut der Linker mit allen übergebenen, bereits assemblierten Programmen. Zuletzt kann er nun den Wert des Labels "entry" in den Code des Aufrufers einarbeiten, so dass der Befehl "j entry" sein Ziel korrekt findet.

Dieser Mechanismus wird auch umgekehrt z.B. beim Befehl "jal readchar" (Zeile 14) benutzt. Das oben gegebene Programm weiß nur, dass es wohl eine Routine mit dem Namen readchar gibt, aber nicht, wo es im Speicher liegt bzw. liegen wird. Der Assembler teilt dem Linker aber automatisch mit, an welcher Stelle ein Sprung zu "readchar" nachgetragen werden muss.

Global gesehen dient der Linker dem Zweck, damit ein Programm nicht immer als ganzes übersetzt werden muss, sondern dass auch Teile einzeln übersetzbar sind. Ein Linker vervollständigt ein Programm bevor und ohne dass es gestartet wird ("early binding"). Dies steht im Gegensatz zum sogenannten "late binding", bei dem ein Programm gestartet wird und sich dynamisch die fehlenden Teile hinzulädt.

Wenn das Programm (nun endlich) bei "entry:" los läuft, wird als erstes die Rücksprungadresse gesichert, die beim Programmende wieder benötigt wird. Hierzu wird (Zeilen 5 und 6) der Stackpointer in Register 29 um 8 verringert um Platz auf dem Stack zu schaffen. Danach kann die Rücksprungadresse dort gespeichert werden mit

```
        sw      $31,0($29)
```

Diese Befehlsfolge passt zur MIPS-Registerkonvention. Man kann diese Befehlsfolge einfach so an den Anfang jeder Routine schreiben oder die Rücksprungadresse auf andere Art sichern (oder auch überhaupt nicht, wenn man selbst keine Aufrufe mehr tätigt). Übernimmt man diese Befehlsfolge "blind", dann muss auch der Abspann, der erst weiter unten erklärt wird, passend dazu übernommen werden.

Danach folgt die Assemblerdirektive ".data" (Zeile 7). "Assemblerdirektiven" sind übrigens Anweisungen an den Assembler, also keine Maschinenbefehle. ".data" teilt dem Assembler mit, dass das Folgende nicht zum Code soll, sondern zu den Daten. Der Assembler verwaltet nämlich (optional auch mehr als) zwei Bereiche, an denen beliebig wechselnd weitergeschrieben werden kann. Die Assemblerdirektive .text (Zeile 10) schaltet zurück, damit am Code weitergeschrieben wird.

Hinter ".data" folgt ein Label (Zeile 8), danach die Direktive ".asciiz" (Zeile 9). Letztere bewirkt, dass der Parameter von ".asciiz", also der angegebene String, ASCII-kodiert abgelegt werden soll (im .data-Segment das vorher angewählt wurde). ".asciiz" legt hinter dem String noch ein Byte mit dem Wert Null ab. (Null = engl. Zero, daher das Z am Ende von asciiz.) Das Label "text_1:" ordnet dem Namen "text_1" die Anfangsadresse des abgelegten Strings zu. D.h., während dem Assemblieren steht "text_1" für die Adresse des ersten Zeichens des Strings.

Nachdem mit ".text" wieder umgeschaltet wurde, damit am Code weitergeschrieben wird, folgen drei Maschinenbefehle (Zeile 11 .. 13):

```
        lui     $4,%hi(text_1)
        ori     $4,%lo(text_1)
        jal     writestring
```

Mit "lui" und "ori" wird ein 32-bit großes Literal in das Register $4 geladen. Es sollte aus der Vorlesung bekannt sein, dass bei MIPS kein 32-bit Literal in einen Befehl passt und deshalb eine solche Kombination nötig ist. Der Assembler unterstützt dies hier, indem er mit den Funktionen %hi() und %lo() die passenden Anteile von einer Adresse, hier text_1, freischneiden kann. Letztlich dient der Wert in $4 dann als Parameter für den Aufruf an writestring. Writestring gibt einen asciiz-String auf dem Bildschirm aus. Zur Laufzeit wird der Text "Eine Taste druecken: " erscheinen.

Die folgende Zeile "jal readchar" (Zeile 14) ruft eine weitere, vorgegebene Unterroutine auf. Sie wartet auf einen Tastendruck und liefert dann ein ASCII-Zeichen, entsprechend der gedrückten Taste in Register $2 zurück.

Danach wird das gerade eingegebene Zeichen wieder ausgegeben (Zeilen 15 und 16), was die Routine "writechar" erledigt. Allerdings erwartet diese Routine das auszugebende ASCII-Zeichen in Register $4. Zum Transfer des Zeichens vom Register $2 ist deshalb der folgende Befehl eingefügt:

```
        or      a0,v0,$0
```

Neu ist hier, dass die Operanden bei dem Befehl nicht "$4,$2,$0" lauten, vielmehr wurden hier die Namen nach der Registerkonvention von MIPS benutzt. Beispielsweise steht a0 dabei für Register $4, da es üblicherweise das erste (genauer: nullte) Argument bei Aufrufen ist.

Damit diese symbolischen Namen funktionieren, läuft vor dem eigentlichen Assemblieren der C-Preprocessor "cpp". Präprozessoren dienen hauptsächlich textuellen Ersetzungen vor einem Compiler- oder Assemblerlauf. Im vorliegenden Beispielprogramm werden zwei Spielarten davon genutzt. In Zeile 1 steht eine Anweisung für cpp:

```
#include "asm_regnames.h"
```

Diese Anweisung bewirkt, dass die include-Anweisung selbst durch den Inhalt der angegebenen Datei ersetzt wird.

In der Datei "asm_regnames.h" finden sich Zeilen wie die folgende:

```
#define a0 $4
```

Diese Anweisung bewirkt, dass ab sofort jeder Bezeichner "a0" im Quelltext durch $4 ersetzt wird, also genau die Ersetzung, die beim "or a0,v0,$0" nötig ist. Wichtig: es wird nicht jeder Substring, sondern nur ganze Worte werden ersetzt! Diese Art der textuellen Ersetzung ist trotzdem insofern gefährlich, weil sie unabhängig von Semantik und Kontext erfolgt, was zu unerwarteten Ersetzungen, und damit zu Fehlern führen kann. Mit etwas Selbst-Disziplin kommen diese praktisch aber fast nie vor.

Das Programm endet mit dem Rücksprung an den Aufrufer, also das vorgegebene Startprogramm. Hierzu muss die am Anfang des Programms auf dem Stack gesicherte Rücksprungadresse wieder in einem Register verfügbar gemacht werden, hier mit

```
lw   ra,0(sp)
```

(Zeile 17). Danach müssen die 8 Bytes auf dem Stack wieder freigegeben werden (Zeile 18), bevor der eigentliche Rücksprung erfolgen kann (Zeile 19).

Es ist wichtig, dass jedes Programm "sauber" beendet wird. Kümmert man sich nicht um ein Programmende, so stört das weder die MIPS-Simulation noch irgendeinen Prozessor der Welt. Vielmehr wird der Programmzähler weiter erhöht und zufällige Werte aus dem Speicher werden als Maschinenbefehle zur Ausführung gebracht.

Im obigen Beispielprogramm werden Literale immer nur dezimal angegeben. Im Folgenden drei Zeilen, die den gleichen Code erzeugen, aber das Literal dem Assembler unterschiedlich mitteilen:

```
        addiu   $2,$4,48        # Literal ist dezimal
        addiu   $2,$4,0x30      # Literal ist hexadezimal
        addiu   $2,$4,'0        # Literal ist als ASCII-Zeichen angegeben
```

Assembler Direktiven

Es folgt eine Zusammenstellung der Assembler-Direktiven, die man im Praktikum kennen sollte.

- **.text und .data**
  Diese beiden Direktiven schalten zwischen dem Code-Segment und dem Daten-Segment um. Danach folgende Anweisungen legen Code bzw. Daten dann im jeweils aktiven Segment ab. Es ist so auch durchaus möglich, obwohl meist nicht sinnvoll, z.B. Code im Datensegment abzulegen.
- **.byte**
  Legt die als Parameter der Direktive gegebenen Werte als einzelne Bytes ab.
  Beispiel: .byte 1,2,3,4
- **.word**
  Wie .byte, legt aber 4-Byte Werte als Worte ab.
  Beispiel: .word 1234,0xaffe
- **.ascii und .asciiz**
  Wie .byte, legt aber einen ganzen String byteweise ab. .asciiz fügt zusätzlich noch ein Byte mit dem Wert Null an.
  Beispiel: .ascii "Hello, World"
- **.space <n>**
  Läßt Platz, der nicht vorbesetzt wird, d.h., beim Programmstart nicht mit Nullen gefüllt ist, sondern zufällige Werte enthält.

  ```
  string:
          .space  256    # platz fuer einen 256 byte langen string
  ```

- **.align <n>**
  Sorgt dafür, dass die nächsten Daten bzw. der nächste Code auf der nächsten "geraden" bzw. "ausgerichteten" Adresse abgelegt wird, entsprechend dem Parameter, etwa wie folgt:

  ```
          .byte   123
          .align  4       # ausrichtung auf wort-grenze (4-byte-grenze)
          .word   9999
  ```

  .align ist eine Art bedingtes .space: es werden optional einige Bytes ausgelassen.
- **.globl <identifier>**
  <identifier> soll dem Linker mitgeteilt werden, so dass andere Übersetzungseinheiten diesen Namen referenzieren können. Nötig um Routinen global in einem Programmierprojekt bekannt zu machen. Siehe Beispielsweise die Art und Weise, wie die vorgegebenen Unterroutinen wie "writestring" benutzt werden können, obwohl sie nicht im eigenen Quelltext stehen.

Unterroutinen zur Ein- und Ausgabe

Um die unten aufgeführten Routinen benutzen zu können, muss die Bibliothek dieser Routinen im Link-Step mit angegeben werden. Wie dies geschieht, ist bei der Beschreibung des VMIPS-Simulators mit angegeben.

Die Routinen verändern keine Register außer den Übergabeparametern.

Zeichenweise Ein-/Ausgabe

- **readchar**
  Eingabe: %
  Ausgabe: $2 - Zeichen
  Es wird ein Zeichen von der Tastatur eingelesen und in $2 zurückgegeben. (Es wird auf einen Tastendruck gewartet.)
- **writechar**
  Eingabe: $4 - Zeichen
  Ausgabe: %
  Das Zeichen in $4 wird auf dem Bildschirm ausgegeben.
- **pause**
  Eingabe: %
  Ausgabe: %
  Es wird ein Zeichen von der Tastatur eingelesen und nicht zurückgegeben. Es wird (also nur) auf einen (beliebigen) Tastendruck gewartet.
- **writecrlf**
  Eingabe: %
  Ausgabe: %
  Der Cursor wird an den Anfang der nächsten Zeile verschoben. Dies entspricht der Ausgabe der beiden ASCII-Zeichen CR (Carriage Return, Wagenrücklauf, Cursor an den Zeilenanfang, 0x0d) und LF (Line Feed, Zeilenvorschub, Cursor eine Zeile nach unten, 0x0a).

Stringweise Ein-/Ausgabe

- **readline**
  Eingabe: $4 - Pufferadresse
  Ausgabe: %
  Eine Zeile Text vom Benutzer einlesen und im Puffer ablegen. Es werden max. 80 Zeichen angenommen und gespeichert. Das letzte eingegebene Zeichen (die Return-Taste) wird als Null-Zeichen im Puffer abgelegt. Jedes Zeichen wird als Byte abgelegt. Der Puffer muss eine Größe von 81 Bytes haben. Definiert sind im Puffer nur die Bytes bis zum Null-Zeichen, die restlichen enthalten zufällige Werte.
- **writestring**
  Eingabe: $4 - Adresse
  Ausgabe: %
  Der Text, beginnend bei der Adresse, die in $4 gegeben ist, wird ausgegeben. Als Text wird ein ASCIIZ-String erwartet, also ein Byte pro Zeichen, ein Byte mit dem Wert Null als Kennzeichnung des Stringendes.

Zahlen Ein-/Ausgabe

Diese Routinen benutzen intern einen 81 Byte grossen Puffer.

- **readdec**
  Eingabe: %
  Ausgabe: $2 - Wert, $3 - Anzahl verarbeiteter Zeichen
  Zuerst wird eine Zeile Text in einen internen Puffer eingelesen. Danach wird vom Pufferanfang her eine Dezimalzahl gelesen. Zurückgegeben wird der Wert der Dezimalzahl und die Anzahl der Zeichen, die erfolgreich in diese Zahl eingegangen sind. (Ist bereits das erste Zeichen keine Dezimalziffer, so enthalten $2 und $3 den Wert Null.)
- **readhex**
  Eingabe: %
  Ausgabe: $2 - Wert, $3 - Anzahl verarbeiteter Zeichen
  Wie readdec (s.o.), aber es wird eine Hexadezimalzahl gelesen. Die Ziffern a bis f dürfen klein oder gross geschrieben sein.
- **writedec**
  Eingabe: $4 - Wert
  Ausgabe: %
  Der Wert von $4 wird als vorzeichenlose Dezimalzahl ausgegeben. Außer den nötigen Ziffern wird nichts ausgegeben, d.h., keine führende oder anschließende Leerzeichen.
- **writehex**
  Eingabe: $4 - Wert
  Ausgabe: %
  Der Wert von $4 wird als achtstellige Hexadezimalzahl ausgegeben.

Register- und Stackkonventionen bei MIPS

Da es nützlich ist, wenn Routinen verschiedener Sprachen gemischt werden können, ist es sinnvoll, dass Compiler identische Aufrufkonventionen einhalten. Für die MIPS-Architektur gilt deshalb folgende Konvention (übernommen von http://en.wikipedia.org/wiki/MIPS_architecture, Überschrift "Compiler Register Usage"):

Name	Number	Use	Callee must preserve?
zero	$0	constant 0	N/A
at	$1	assembler temporary	no
v0–v1	$2–$3	Values for function returns and expression evaluation	no
a0–a3	$4–$7	function arguments	no
t0–t7	$8–$15	temporaries	no
s0–s7	$16–$23	saved temporaries	yes
t8–t9	$24–$25	temporaries	no
k0–k1	$26–$27	reserved for OS kernel	no
gp	$28	global pointer	yes
sp	$29	stack pointer	yes
fp	$30	frame pointer	yes
ra	$31	return address	N/A

Die erste Spalte 'Name' vergibt eingängigere Namen für die Register 0 bis 31 bezüglich ihrer jeweiligen Funktion.
Die zweite Spalte 'Number' gibt die Registernummern selbst an.
Die dritte Spalte 'use' gibt an, welchem Zweck das/die Register dienen.
Die letzte Spalte 'Callee must preserve?' gibt an, ob ein Unterprogramm das Register jeweils einfach benutzen darf, oder ob es nach einer Wertänderung bei der Rückkehr zum Aufrufer für eine Wiederherstellung des ursprünglichen Inhalts sorgen muss.

Neben den Register-Konventionen ist noch der Aufbau des Stacks von essentieller Bedeutung. Zur weiteren Erklärung soll deshalb ein C Programm, vom GNU-C Compiler in Assemblercode übersetzt, betrachtet werden.
Zuerst einmal der Quelltext:

```
int globvar;

int f(int p1, int p2, int p3, int p4, int p5) {
  int tmp;
  tmp=p2+p5;
  if (p1==0) {
    return tmp;
  } else {
    return f(p1-1, p2, globvar, 0, p5);
  }
}
```

Die Funktion f() erhält fünf Parameter, wobei der dritte und vierte nur dazu dienen, dass die Routine mehr als vier Parameter hat. Dadurch tritt der Fall auf, dass Parameter auf dem Stack übergeben werden müssen, da die Registerkonvention nur bis zu vier abdeckt.

Der Compiler erzeugt daraus folgenden Code, der hier mit Kommentaren versehen wurde. Außerdem wurden Befehle vertauscht (nur beim Erstellen der Parameter für den rekursiven Aufruf), aber nur so, dass sich am Programm nichts ändert, aber die Befehle, die zu einem logische Schritt gehören, nun beieinander stehen.

```
f:                              ### funktion_beginnen
        addiu   $sp,$sp,-40     # Platz für neues Stackframe schaffen: 40 Bytes
        sw      $31,36($sp)     # Ruecksprungadresse sichern
        sw      $fp,32($sp)     # fp des aufrufers sichern
        move    $fp,$sp         # fp fuer f() setzen
        sw      $4,40($fp)      # p1 bis p4 im stackframe des Aufrufers sichern
        sw      $5,44($fp)
        sw      $6,48($fp)
        sw      $7,52($fp)

        lw      $3,44($fp)      # tmp=p2+p5
        lw      $2,56($fp)
        nop
        addu    $2,$3,$2
        sw      $2,24($fp)

        lw      $2,40($fp)      # if (p1==0) {
        nop
        bne     $2,$0,$L2
        nop

        lw      $2,24($fp)      #   retval=tmp
        nop
        sw      $2,28($fp)
        j       $L1             #   goto Funktion_beenden
        nop

$L2:                            # } else {

        lw      $2,40($fp)      #   p1_neu = p1-1
        nop
        addiu   $3,$2,-1
        move    $4,$3

        lw      $5,44($fp)      #   p2_neu = p2

        lui     $6,%hi(globvar) #   p3_neu = globvar
        lw      $6,%lo(globvar)($6)

        move    $7,$0           #   p4_neu = 0

        lw      $2,56($fp)      #   p5_neu = p5
        nop
        sw      $2,16($sp)

        jal     f               #   rekursiver Aufruf
        nop

        sw      $2,28($fp)      #   retval = retval des rekursiven aufrufs
                                # }

$L1:                            ### Funktion_beenden:
        lw      $2,28($fp)      # retval an Aufrufer zurueckgeben in $2
        move    $sp,$fp         # stack bereinigen
        lw      $31,36($sp)     # ruecksprungadresse vorbereiten
        lw      $fp,32($sp)     # fp das aufrufers restaurieren
        addiu   $sp,$sp,40      # das eigene stackframe wieder freigeben
        j       $31             # ruecksprung!
        nop
```

Zum besseren Verständnis der folgenden Erklärungen noch das Stackframe von f(), wie der Compiler es in obigem Code benutzt:

↑ Abbau des Stacks bei Rücksprung		Vorherige Stackframes
	+56	p5
	+52	p4
	+48	p3
	+44	p2
(vor dem aktuellen Aufruf war hier +0)	+40	p1
	+36	Rücksprungadresse
	+32	gesicherter Framepointer (== fp des Aufrufers)
	+28	Rückgabewert von f
	+24	tmp
	+20	???
	+16	p5 für Aufruf
	+12	p4 für Aufruf
	+8	p3 für Aufruf
	+4	p2 für Aufruf
	+0	p1 für Aufruf
↓ Platz für weitere Aufrufe		frei

Das Stackframe umfasst 40 Bytes, im Bild eingerahmt von etwas dickeren Linien, der Bereich von +0 bis +36. Dazu kommen die Aufrufparameter, die zum Stackframe des Aufrufers gehören, aber immer direkt über dem aktuellen Stackframe anschließen.

Der Code von f() beginnt mit dem Reservieren der 40 Bytes für das Stackframe. Dann werden Rücksprungadresse und Framepointer des Aufrufers gesichert. Danach kann der Framepointer für f() intern passend gesetzt werden. Als letztes sichert der Vorspann die in den Registern übergebenen Parameter in den dafür vorgesehenen Zellen von +40 bis +52 im Stack, genauer, im Stackframe des Aufrufers. Dieser letzte Schritt entfällt fast immer, wenn der Compiler optimiert, was im vorliegenden Beispiel explizit abgeschaltet wurde. Wichtig: Der Aufrufer muss den Platz bereitstellen, da er nicht Wissen kann, ob der Aufgerufene optimiert oder nicht oder den Platz evtl. trotz Optimierung benötigt.

Danach folgt der Code, der dem C-Quelltext entspricht. Es ist jeweils kommentiert welche Maschinenbefehle welchem C-Statement entsprechen. Es lohnt sich dies einmal selbst zu verfolgen. Im Folgenden werden nur noch die Besonderheiten besprochen.

In der Zuweisung *retval = tmp* benennt der Kommentar eine Variable *retval*, die der Compiler intern angelegt hat, d.h., die nicht im Quelltext auftaucht. Sie enthält den Wert, der von f() zurückgegeben werden muss. Dies ist nötig, weil das Statement *return* nicht einfach in einen Rücksprung kodiert werden kann, vielmehr ist eine längliche Codesequenz nötig um eine Routine zu beenden. Diese Sequenz ist am Ende der Routine und wird angesprungen, wenn *return* im Quelltext auftaucht. Deshalb kodiert der Compiler hier statt einem Rücksprung ein *goto Funktion_beenden*.

Im else-Zweig werden zuerst die Parameter für den rekursiven Aufruf bereitgestellt. Man sieht, daß die ersten vier Parameter nur in den Registern bereitgestellt werden, der fünfte nur im Stack.

Interessant ist dabei, dass der Compiler am Anfang von f() genau 40 Bytes auf dem Stack belegt, was nach Abzug der lokalen Variablen für fünf Aufrufparameter reicht. Offensichtlich arbeitet der Compiler so vorausschauend, daß er den Prozedurkopf so anlegt, daß ausreichend Platz für die längste Parameterliste vorhanden ist, die in f() vorkommt. Alternativ könnte vor jedem Funktionsaufruf Platz für die Parameter auf dem Stack geschaffen und danach freigegeben werden. Dies kostet natürlich mehr Befehle und damit auch mehr Zeit, als der Weg, den der Compiler benutzt.

Die Funktion des Befehls "move $sp,$fp # stack bereinigen" ist nicht offensichtlich. Er ist in der Beispielfunktion f() auch überflüssig (und es ist uns nicht gelungen eine Routine zu compilieren, in der der Befehl nicht überflüssig gewesen wäre). Der Befehl ist nötig, falls sp modifiziert wird, z.B. um einen temporären Wert auf den Stack zu legen z.B. mit "addiu $sp,$sp,-4" und "sw $17,0($sp)". Dies kann (zumindest bei anderen Compilern als gcc) vorkommen, wenn eine Ausdrucksauswertung (vgl. Aufgabe 1) so komplex wird, dass die temporären Register des Prozessors nicht ausreichen. Die Stack-artige Verwaltung von Registern wird dann üblicherweise durch den realen Stack ergänzt. Wie auch immer, wenn der Stack derartig modifziert wäre und damit auch das Register sp, und würde dann ein "goto Funktion_beenden" vorkommen, bevor sp bereinigt wurde, dann wäre der am Anfang des Absatzes erwähnte Befehl wichtig, damit die restlichen Befehle des Funktions-Abspanns korrekt funktionieren können.

Mips 32 Instruction Quick Reference

Moving Bits around

LA	Rd, Label	Rd = Address(Label)
LI	Rd, Imm32	Rd = Imm32
LUI	Rd, const16	Rd = const16 << 16
MOVE	Rd, Rs	Rd = Rs
MOVN	Rd, Rs, Rt	IF (Rt ≠ 0) Rd = Rs
MOVZ	Rd, Rs, Rt	IF (Rt = 0) Rd = Rs
MFHI	Rd	Rd = Hi
MFLO	Rd	Rd = Lo
MTHI	Rs	Hi = Rs
MTLO	Rs	Lo = Rs
SEB	Rd, Rs	Rd = ±Rs(7:0)
SEH	Rd, Rs	Rd = ±Rs(15:0)
EXT	Rd, Rs, P, S	Rd = +Rs(P+S-1:P)
INS	Rd, Rs, P, S	Rd(P+S-1:P) = Rs(S-1:0)
WSBH	Rd, Rs	Rd=Rs(23:16)::Rs(31:24)::Rs(7:0)::Rs(15:8)
CLO	Rd, Rs	Rd = CountLeadingOnes(Rs)
CLZ	Rd, Rs	Rd = CountLeadingZeros(Rs)
ROTRV ROTR	Rd, Rs, Rt Rd, Rs, shift5	Rd = Rs((Rt(4:0)-1):0)::Rs(31:Rt(4:0)) Rd = Rs(shift5-1:0) :: Rs(31:shift5)
SLLV SLL	Rd, Rs, Rt Rd, Rs, shift5	Rd = Rs << Rt(4:0) Rd = Rs << shift5
SRAV SRA	Rd, Rs, Rt Rd, Rs, shift5	Rd = ±Rs >> Rt(4:0) Rd = ±Rs >> shift5
SRLV SRL	Rd, Rs, Rt Rd, Rs, shift5	Rd = +Rs >> Rt(4:0) Rd = +Rs >> shift5

Flowcontrol

BEQ	Rs, Rt, off18	IF (Rs = Rt) PC += ±off18
BEQZ	Rs, off18	IF (Rs = 0) PC += ±off18
BNE	Rs, Rt, off18	IF (Rs ≠ Rt) PC += ±off18
BNEZ	Rs, off18	IF (Rs ≠ 0) PC += ±off18
BGEZ	Rs, off18	IF (Rs ≥ 0) PC += ±off18
BGTZ	Rs, off18	IF (Rs > 0) PC += ±off18
BLEZ	Rs, off18	IF (Rs ≤ 0) PC += ±off18
BLTZ	Rs, off18	IF (Rs < 0) PC += ±off18
B	off18	PC += ±off18
J	addr28	PC = PC(31:28) :: +addr28
JR	Rs	PC = Rs
NOP	NO-OP	
BAL	off18	R31 = PC + 8, PC += ±off18
JAL	addr28	R31 = PC + 8; PC = PC(31:28)::+addr28
JALR	Rd, Rs	Rd = PC + 8; PC = Rs
BGEZAL	Rs, off18	R31 = PC + 8; IF (Rs ≥ 0) PC += ±off18
BLTZAL	Rs, off18	R31 = PC + 8; IF (Rs < 0) PC += ±off18

± signed_operation/signed_operand/sign_extended_operand
+ unsigned_operation/unsigned_operand/zero_extended_operand
:: Concatenation of Bit Fields

Arithmetic & Logic

ADDU ADDIU	Rd, Rs, Rt Rd, Rs, const16	Rd = Rs + Rt Rd = Rs + ±const16
ADD ADDI	Rd, Rs, Rt Rd, Rs, const16	Rd = Rs + Rt (Overflow Trap) Rd = Rs + ±const16
SUBU	Rd, Rs, Rt	Rd = Rs - Rt
SUB	Rd, Rs, Rt	Rd = Rs - Rt (Overflow Trap)
NEGU	Rd, Rs	Rd = -Rs
SLT SLTI	Rd, Rs, Rt Rd, Rs, const16	Rd = ±(Rs < Rt) ? 1 : 0 Rd = ±(Rs < ±const16) ? 1 : 0
SLTU SLTIU	Rd, Rs, Rt Rd, Rs, const16	Rd = +(Rs < Rt) ? 1 : 0 Rd = +(Rs < +const16) ? 1 : 0
MUL	Rd, Rs, Rt	Rd = ±(Rs × Rt)
MULT	Rs, Rt	(Hi:Lo) = ±(Rs × Rt)
MULTU	Rs, Rt	(Hi:Lo) = +(Rs × Rt)
DIV	Zero, Rs, Rt	Lo = ±(Rs / Rt); Hi = ±(Rs % Rt)
DIVU	Zero, Rs, Rt	Lo = +(Rs / Rt); Hi = +(Rs % Rt)
MADD	Rs, Rt	(Hi:Lo) += ±(Rs × Rt)
MSUB	Rs, Rt	(Hi:Lo) -= ±(Rs × Rt)
MADDU	Rs, Rt	(Hi:Lo) += +(Rs × Rt)
MSUBU	Rs, Rt	(Hi:Lo) -= +(Rs × Rt)
AND ANDI	Rd, Rs, Rt Rd, Rs, const16	Rd = Rs & Rt Rd = Rs & +const16
OR ORI	Rd, Rs, Rt Rd, Rs, const16	Rd = Rs | Rt Rd = Rs | +const16
NOR	Rd, Rs, Rt	Rd = ~(Rs | Rt)
XOR XORI	Rd, Rs, Rt Rd, Rs, const16	Rd = Rs ⊕ Rt Rd = Rs ⊕ +const16
NOT	Rd, Rs	Rd = ~Rs

Memory

LW	Rd, off16(Rs)	Rd = MEM32[Rs + ±off16]
SW	Rs, off16(Rt)	MEM32[Rt + ±off16] = Rs
LB	Rd, off16(Rs)	Rd = ±MEM8[Rs + ±off16]
LBU	Rd, off16(Rs)	Rd = +MEM8[Rs + ±off16]
SB	Rs, off16(Rt)	MEM8[Rt + ±off16] = RS(7:0)
LH	Rd, off16(Rs)	Rd = ±MEM16[Rs + ±off16]
LHU	Rd, off16(Rs)	Rd = +MEM16[Rs + ±off16]
SH	Rs, off16(Rt)	MEM16[Rt + ±off16] = RS(15:0)
LWL	Rd, off16(Rs)	Rd = LOADWORDLEFT(Rs + ±off16)
LWR	Rd, off16(Rs)	Rd = LOADWORDRIGHT(Rs + ±off16)
SWL	Rs, off16(Rt)	STOREWORDLEFT(Rt + ±off16, Rs)
SWR	Rs, off16(Rt)	STOREWORDRIGHT(Rt + ±off16, Rs)
ULW	Rd, off16(Rs)	Rd = UNALIGNED_MEM32(Rs + ±off16)
USW	Rs, off16(Rt)	UNALIGNED_MEM32(Rt + ±off16) = Rs
LL	Rd, off16(Rs)	Rd = MEM32[Rs + ±off16]; LINK
SC	Rd, off16(Rs)	IF (ATOMIC) MEM32[Rs + ±off16] = Rd Rd = ATOMIC ? 1 : 0