Hochsprachen-Compiler für den FHOP

Dipl.-Ing. (FH) Oliver Bischoff,
Prof. Dr.-Ing. Dirk Jansen, ASIC-Design-Center
Fachhochschule Offenburg, Badstr. 24, 77652 Offenburg
Tel. 0781/205-267, Fax 0781/205-242,
E-Mail: d.jansen@fh-offenburg.de

An der Fachhochschule Offenburg wurde ein C-Compiler entwickelt. Es handelt sich dabei um einen Hochsprachen-Compiler für den an der Fachhochschule entwickelten Prozessor FHOP (First Homemade Operational Prozessor), der ANSI-C-Code in Maschinencode umsetzt.
Um den Maschinencode zu erzeugen, wird die Hilfe des bereits existierenden Crash - Assemblers in Anspruch genommen. Der C-Compiler Assemblercode erzeugt, welcher dann durch den Crash-Assembler in Maschinencode umgesetzt wird.
Der Compiler soll als Tool bei der Entwicklung neuer Software für den FHOP dienen. Hierbei wird vor allem an die Entwicklungsgeschwindigkeit gedacht, die für C-Code wesentlich höher ist, als für Assemblercode, der die selbe Funktionalität besitzt.

1 Einleitung

Die Programmiersprache „C" erfreut sich großer Beliebtheit und weiter Verbreitung. Dies beruht auf der Tatsache, daß C eine sehr übersichtliche und einfach zu erlernende Programmiersprache ist. Aufgrund der Vielzahl an Prozessoren stehen auch eine Menge an Compiler zur Verfügung; ein Teil ist „Retargetable", d. h. es besteht die Möglichkeit, mit diesen Compilern Maschinencode für Prozessoren zu erzeugen, die nicht vom selben Prozessortyp sind, wie der, auf dem der Sourcecode compiliert wurde. Diese Compiler, z. B. GNU, erfordern eine 32-Bit-Architektur des Prozessors, so daß eine Anpassung dieser Compiler für den FHOP nicht möglich war. Deshalb entschloß ich mich, mit Hilfe von Compilerbau-Tools, einen Compiler selbst zu schreiben. Dieser Compiler sollt sich so nah wie möglich an dem ANSI-C-Standard halten, damit es später möglich ist, bestehende C-Bibliotheken einzubinden.

2 Hochsprache! - Warum?

Das Programmieren von Prozessoren ist eine angenehme Arbeit, sofern die Tools zum Programmieren des Prozessors vorhanden sind. Maschinencode direkt einzugeben, ist sicher keine sehr angenehme Arbeit, zumal die Verwaltung des Speichers und sämtlicher Adressen in den Händen des Programmierers liegt. Variablen existieren auf dieser Programmierebene nicht. Der Code entbehrt jeglicher Übersicht. Eine Erleichterung bietet der Assembler, der das Erzeugen von Maschinencode komfortabler gestaltet. Er übernimmt die Adressenverwaltung und erlaubt das Anlegen von Variablen. Jedoch bleiben Zuverlässigkeit und Übersichtlichkeit weiterhin ein Problem des Assemblerprogramms. Diese Nachteile weißt ein Hochsprachen-Compiler nicht mehr auf. Sowohl die Übersichtlichkeit, als auch die Zuverlässigkeit des Codes sind weitaus besser, als bei einem Assemblercode. Dabei besticht vor allem die Lesbarkeit. Durch Unterteilung in einzelne Funktionen, erhält man ein übersichtlichen und lesbaren Code. Die Zuverlässigkeit wird durch erweiterte Syntaxprüfungen erhöht. Bei einem Assemblerprogramm kann das Vergessen eines Push- oder Pop-Befehls den Prozessor zum Abstürzen veranlassen, während man in einer Hochsprache dazu schon mehr braucht.
Ein weiteres Argument, welches die Vorteile eines C - Compilers unterstreicht, ist die Entwicklungszeit. Ein Vergleich (Abbildung 1) von zwei Programmausschnitten wird dies gut darstellen.

C - Code

```
main() {
    initialize();
    for(i=0;i<=3;i++) {
      tast_in();
      if(taste == kode[i])
        state++;
      else
        state=0;        }
  if(state==4)
    Tuer_out();
  else {
    state=0;
  Fehler_out();         }
}
```

Assembler-Code

```
main: CAL initialize
      LDI i_global
      PSH A
      LDI #0000
      POP B
      STA M
_FOR_1:
      LDA i_global
      PSH A
      LDI #0003
      POP B
      CAL KLGL_INT
      CMI 0
```

Abbildung 1 Vergleich C - / Assembler - Code

Dabei hat der unterstrichene C-Code links genau die gleiche Funktionalität wie der Assemblercode rechts.

Man sieht schon hier einen deutlichen Unterschied im Umfang.
Außerdem läßt sich der linke Code in Abbildung 1 wesentlich besser lesen. Durch die Anlehnung des C-Befehlssatzes an das Englische Vokabular kann sehr schneller verstanden werden, was ein bestimmter Programmteil ausführt. (z. B. if-Anweisungen). So läßt sich auch nach größeren Zeiträumen schneller nachvollziehen, welche Funktionalität der C - Code erfüllen soll. Ein Nachteil der Hochsprache der Umfang des erzeugten Codes. Bei Assembler-programmierung läßt sich der Code geringer halten. Jedoch ist der Speicher des FHOP so groß ausgelegt, so daß speicherplatzkritische Anwendungen bis jetzt nicht aufgetreten sind. Deshalb wird dieser Nachteil gerne in Kauf genommen, da die oben an - gesprochenen Vorteile überwiegen.

3 Wie wird ein Compiler erstellt?

Um die Vorteile der verwendeten Tools zu erkennen, möchte ich hier kurz auf die Funktionsweise eines Compilers eingehen.
Zuerst wird der Source-Code einer Überprüfung unterzogen. Wird diese erfolgreich abgeschlossen, d.h. der Source-Code ist fehlerfrei, wird versucht, den Source-Code durch eine Baumstruktur darzustellen. Abbildung 2 zeigt eine solche Baumstruktur.

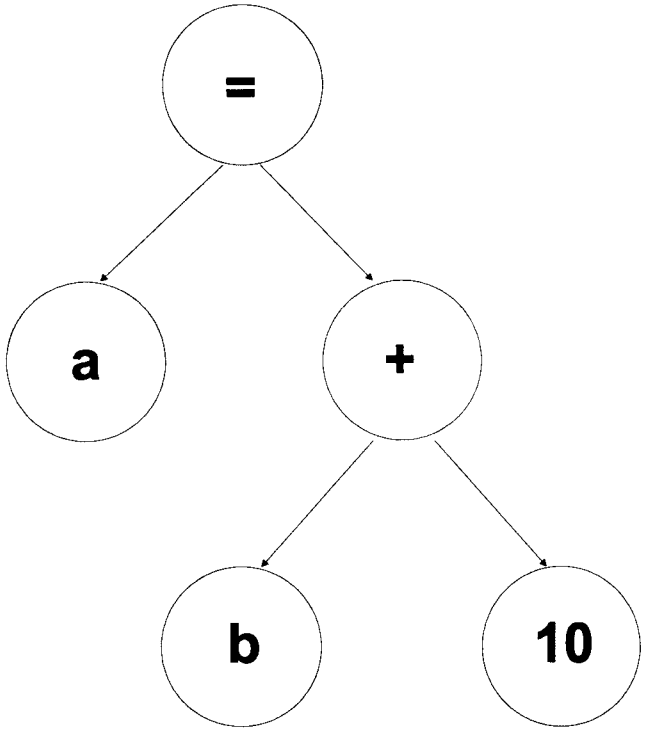

Abbildung 2 Baumstruktur

Die abgebildete Baumstruktur ergab die C-Anweisung „a = b + 10". Nachdem nun der gesamte Source-Code durch solche Baumstrukturen abgebildet wurde, versucht der Compiler diese Strukturen mit Assemblerbefehlen zu überdecken, d.h. in Assemblercode umzusetzen. Für die Baumerzeugung und die Überdeckung wurden nun Compilerbau-Tools verwendet. In Abbildung 3 wird der Ablauf „Source - Code → Maschinencode" schemenhaft dargestellt.

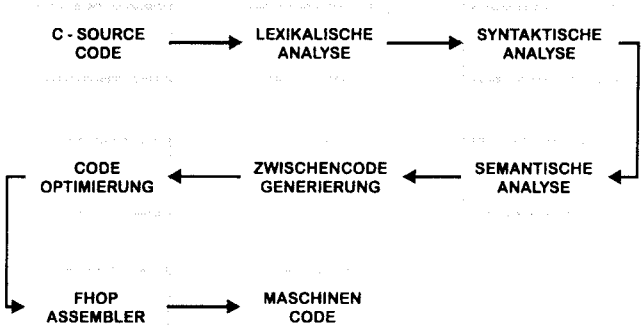

Abbildung 3 - Stufen der Codeerzeugung

Anhand dieses Schemas lassen sich die einzelnen Stufen der Umsetzung vom Source-Code in Assemblercode gut nachvollziehen. Das C-Source-File ist der Ausgangspunkt. Diese wird zuerst auf Richtigkeit überprüft und dann in 2 Schritten in Maschinencode umgesetzt. Schritt 1 ist die Erzeugung von Assemblercode. Dieser wird dann durch den Crash-Assembler in Maschinencode gewandelt. Der erste Schritt wird mit Hilfe von Compilerbau-Tools durchgeführt. Diese Tools helfen dem Anwender bei der Umsetzung der einzelnen Compilerfunktionen. Folgende Tools kamen dabei zum Einsatz:

3.1 flex: Scanner - Tool

Der Compiler erhält das vollständige File. Dieses muß zuerst auf Fehler durchsucht werden. Um diese Überprüfung durchführen zu können, muß dem Compiler bekannt sein, welche Ausdrücke zulässig sind und welche nicht .
Das Absuchen des C-Sourcefiles „von Hand" wäre sehr umständlich. Diese Arbeit übernimmt das Tool flex. flex ist die GNU-Variante des Standard-UNIX-Tools lex, welches bei allen UNIX-Betriebsystemen mit ausgeliefert wird. flex hat den Vorteil, daß es ein Freeware-Tool ist.
Diesem Tool werden nun die erlaubten Stringfolgen bzw. Befehle in einem Definitionsfile bekannt gemacht. Aus diesem File erzeugt flex dann ein C-File, welches dann compiliert die Funktion des Scanners übernimmt, der das gesamte File nach Fehlern in den Stringketten absucht. Diese Fehlersuche entspricht der lexikalischen Analyse, dargestellt als 2 Block in Abbildung 3.

3.2 bison: Parser - Tool

Die darauf folgende syntaktische Analyse erfolgt mit Hilfe des Tools bison. Bison ist ebenso wie flex eine GNU-Variante eines Standard-UNIX-Tools.
Wie auch bei lex, wird hier mit Hilfe eines Definitionsfiles dem Tool bekanntgemacht, welche Ausdrücke zulässig sind und welche nicht.
Bison wandelt das Definitionsfile in C-Code um, der, nachdem er compiliert wurde, die Funktion eines Parsers übernimmt.
Beim Parsen des Files wird nun nicht mehr auf die Richtigkeit einzelner Zeichenketten geachtet, sondern auf den sinnvollen Zusammenhang der Zeichenketten, die vom Scanner erkannt wurden.
So müssen bei bestimmten Operationen gewisse Bedingungen erfüllt sein. Bei einer mathematischen Operation, wie z.B. der Addition, müssen zwei Operanden zur Verfügung stehen. Dies heißt nun für den Parser, das beim Auffinden eines Additionszeichens der Scanner genau zwei Operanden liefern muß. Diese können direkt Zahlenwerte, Variablen oder auch beliebig komplexe Ausdrücke sein. Jedoch muß die Bedingung der zwei Operanden erfüllt sein. Ist dies nicht der Fall, so wird das Parsen mit einer Fehlermeldung abgebrochen.

3.3 Ox: Scanner & Parser - Tool

Ox ist ein Tool, welches die beiden vorherigen Tools, flex und bison unterstützt.
Beim Parsen und Scannen ist es notwendig, mehr Informationen als nur die vorgefundene Zeichenkette weiterzugeben. Dies läßt relativ gut an einem Beispiel zeigen.
Der Ausdruck „int a", definiert eine Variable „a" vom Typ „int". Trifft man nun im späteren Verlauf wieder auf diese Variable „a", z.B. bei „a = 0", so erkennt man nun zwar den Variablennamen „a", der Typ wird jedoch nicht mehr mit angegeben. Diese Information muß also irgendwo gespeichert werden. Nun wäre es möglich bei flex und bison eine Information auszutauschen. Jedoch ist es nicht möglich mehrere solcher Informationen untereinander auszutauschen. Und hier schaltet sich Ox ein. Ox ermöglich diesen Datenaustausch. Und dies auf eine sehr einfache und elegante Weise, so daß beliebig viele Informationen ausgetauscht werden können.

3.4 iburg: Code - Generating Tool

Nachdem nun mit Hilfe der ersten drei Tools der C-Code in eine Baumstruktur umgesetzt werden konnte, steht nun das Problem an, wie setzte ich diese Struktur wieder in ausführbaren Code um, im Falle des FHOP-C-Compilers in Crash-Assemblercode.
Zur Lösung dieses Problems gib es nun das Tool iburg. iburg hilft bei der Umsetzung der Baumstruktur in Assemblercode. Wie bei flex und bison wird diesem Tool durch ein Definitionsfile mitgeteilt, welche Assemblerbefehle zulässig sind. Aus diesem Definitionsfile erzeugt iburg ein C-File. Diesem File, nachdem es compiliert wurde, wird die erzeugte Baumstruktur zugeführt. Iburg erzeugt dann aus der zugeführten Baumstruktur und den Assemblerbefehlen ein lauffähiges Assemblerfile.

3.5 Alle Tools zusammen -> Compiler

Nimmt man nun alle C-Codefiles, die von den Compilerbau-Tools erzeugt wurden, und linkt diese zusammen, erhält man einen Compiler. Dieser Compiler setzt dann, die ihm zugeführte Hochsprache, im Falle des FHOP-C-Compilers die Sprache C, in Maschinencode um.
Diese Art des Compilerbaus ist sehr komfortabel, weil die standardisierten Abläufe bei der Compilierung eines Sourcecodes durch die Tools übernommen werden.
Mit diesen Tools lassen sich auch Compiler für andere Hochsprachen erzeugen, sofern man die Befehle und Grammatik für den Sprache besitzt.
Aber auch die Generierung von Assemblern mit diesen Tools ist möglich, da bei Assembler keine komplizierten Grammatiken vorliegen, so daß man mit diesen Tools sehr schnell zu einem Ergebnis gelangen kann.

4 Was kann der Compiler?

Der Compiler wurde als Kommandozeilen - Compiler konzipiert. Dabei soll in der Befehlszeile außer dem Aufruf verschiedene Dinge angegeben werden können. So z.B. den Beginn des Data-Segmentes, das Data-Limit, der Beginn des Stackbereichs, u. a.. Da zunächst die Tools zum Compilerbau nur auf UNIX zur Verfügung standen, wurde begonnen auf UNIX zu entwickeln. Nachdem jedoch die Tools erfolgreich auf DOS-Ebene portiert wurden, konnte der Compiler ebenfalls auf der DOS-Plattform generiert werden.
Auf der DOS-Ebene sind bis jetzt folgende Befehle implementiert:

- Variablen:
  - int - Variablendeklaration
  - const int - Variablendeklaration
  - Eindimensionale int - Arrays
- Mathematische Operationen:
  - +, -, *, /, Modulo-Fkt
  - Incrementieren & Decrementieren
  - Klammersetzung

- Bitoperationen:
  - AND
  - OR
  - XOR
- Logische Operationen
  - AND
  - OR
- Schiebeoperationen
  - >> (Schiebe nach rechts)
  - << (Schiebe nach links)
- Vergleichsoperationen
  - == (Gleich)
  - != (Ungleich)
  - < (Kleiner als)
  - > (Größer als)
  - >= (Größer gleich)
  - <= (Kleiner gleich)
- Kommentarzeilen
- Inline - Befehle
  - 1 zu 1 - Übernahme der angeführten Assemblerbefehle
- if - Abfrage
  - einfache if - Abfrage
  - if - else - Abfrage
  - if - else if - Abfrage
  - if - else if - else Abfrage
- while - Schleifen
- for - Schleifen

Dieser Befehlsumfang, so hat sich gezeigt, genügt in erster Linie den Ansprüchen der Problematik der Assemblerprogrammierung. Deshalb wird dieser Befehlsumfang zuerst in einigen Projekten tiefergehend getestet. Dabei soll die Tauglichkeit und Zuverlässigkeit des Compilers geprüft werden. Spätere Erweiterungen des Befehlsumfanges sind angedacht und jederzeit möglich, so daß einer Erweiterung des Befehlsumfanges nach erfolgreichen Tests nichts im Wege steht.

Zum Schluß möchte ich noch kurz zwei Screen-Shots zeigen: die Help-Seite und eine erfolgreiche Compilierung:

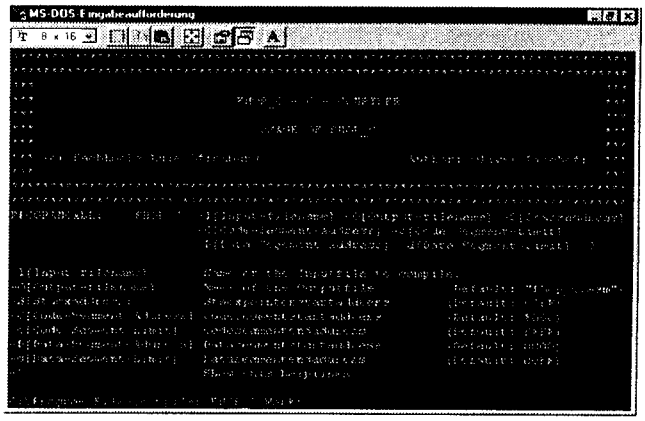

Abbildung 4 Die Help-Seite

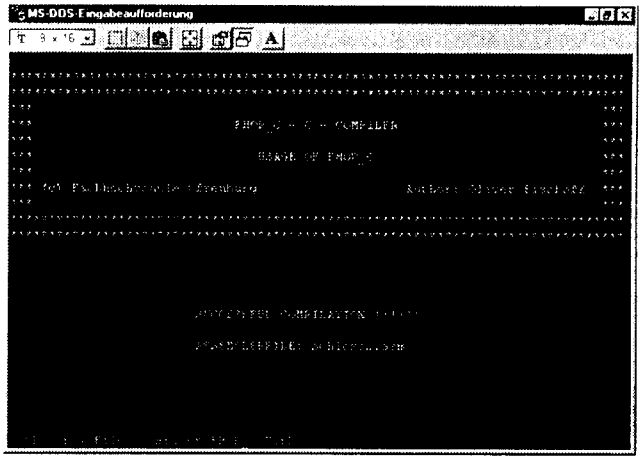

Abbildung 5 Eine erfolgreich Compilierung

5 Literatur

[1] HP-UX Reference Volume 1
SECTION 1: User Commands
Release 8.05

[2] Kurt M. Bischoff
Ox, Tutorial Introduction
November 1993

[3] Christopher W. Fraser
Engineering a Simple, Efficient Code Generator Generator

[4] Brian W. Kernighan
Dennis M. Richtie
The C - Programming Language 2nd Ed. Prentice Hall, 1988

Hochsprachen-Compiler für den FHOP

Dipl.-Ing. (FH) Oliver Bischoff,
Prof. Dr.-Ing. Dirk Jansen, ASIC-Design-Center
Fachhochschule Offenburg, Badstr. 24, 77652 Offenburg
Tel. 0781/205-267, Fax 0781/205-242,
E-Mail: d.jansen@fh-offenburg.de

An der Fachhochschule Offenburg wurde ein C-Compiler entwickelt. Es handelt sich dabei um einen Hochsprachen-Compiler für den an der Fachhochschule entwickelten Prozessor FHOP (First Homemade Operational Prozessor), der ANSI-C-Code in Maschinencode umsetzt.
Um den Maschinencode zu erzeugen, wird die Hilfe des bereits existierenden Crash - Assemblers in Anspruch genommen. Der C-Compiler Assemblercode erzeugt, welcher dann durch den Crash-Assembler in Maschinencode umgesetzt wird.
Der Compiler soll als Tool bei der Entwicklung neuer Software für den FHOP dienen. Hierbei wird vor allem an die Entwicklungsgeschwindigkeit gedacht, die für C-Code wesentlich höher ist, als für Assemblercode, der die selbe Funktionalität besitzt.

1 Einleitung

Die Programmiersprache „C" erfreut sich großer Beliebtheit und weiter Verbreitung. Dies beruht auf der Tatsache, daß C eine sehr übersichtliche und einfach zu erlernende Programmiersprache ist. Aufgrund der Vielzahl an Prozessoren stehen auch eine Menge an Compiler zur Verfügung; ein Teil ist „Retargetable", d. h. es besteht die Möglichkeit, mit diesen Compilern Maschinencode für Prozessoren zu erzeugen, die nicht vom selben Prozessortyp sind, wie der, auf dem der Sourcecode compiliert wurde. Diese Compiler, z. B. GNU, erfordern eine 32-Bit-Architektur des Prozessors, so daß eine Anpassung dieser Compiler für den FHOP nicht möglich war. Deshalb entschloß ich mich, mit Hilfe von Compilerbau-Tools, einen Compiler selbst zu schreiben. Dieser Compiler sollt sich so nah wie möglich an dem ANSI-C-Standard halten, damit es später möglich ist, bestehende C-Bibliotheken einzubinden.

2 Hochsprache! - Warum?

Das Programmieren von Prozessoren ist eine angenehme Arbeit, sofern die Tools zum Programmieren des Prozessors vorhanden sind. Maschinencode direkt einzugeben, ist sicher keine sehr angenehme Arbeit, zumal die Verwaltung des Speichers und sämtlicher Adressen in den Händen des Programmierers liegt. Variablen existieren auf dieser Programmierebene nicht. Der Code entbehrt jeglicher Übersicht. Eine Erleichterung bietet der Assembler, der das Erzeugen von Maschinencode komfortabler gestaltet. Er übernimmt die Adressenverwaltung und erlaubt das Anlegen von Variablen. Jedoch bleiben Zuverlässigkeit und Übersichtlichkeit weiterhin ein Problem des Assemblerprogramms. Diese Nachteile weißt ein Hochsprachen-Compiler nicht mehr auf. Sowohl die Übersichtlichkeit, als auch die Zuverlässigkeit des Codes sind weitaus besser, als bei einem Assemblercode. Dabei besticht vor allem die Lesbarkeit. Durch Unterteilung in einzelne Funktionen, erhält man ein übersichtlichen und lesbaren Code. Die Zuverlässigkeit wird durch erweiterte Syntaxprüfungen erhöht. Bei einem Assemblerprogramm kann das Vergessen eines Push- oder Pop-Befehls den Prozessor zum Abstürzen veranlassen, während man in einer Hochsprache dazu schon mehr braucht.
Ein weiteres Argument, welches die Vorteile eines C - Compilers unterstreicht, ist die Entwicklungszeit. Ein Vergleich (Abbildung 1) von zwei Programmausschnitten wird dies gut darstellen.

C - Code

```
main() {
    initialize();
    for(i=0;i<=3;i++) {
      tast_in();
      if(taste == kode[i])
        state++;
      else
        state=0;        }
  if(state==4)
    Tuer_out();
  else {
    state=0;
  Fehler_out();         }
}
```

Assembler-Code

```
main: CAL initialize
      LDI i_global
      PSH A
      LDI #0000
      POP B
      STA M
_FOR_1:
      LDA i_global
      PSH A
      LDI #0003
      POP B
      CAL KLGL_INT
      CMI 0
```

Abbildung 1 Vergleich C - / Assembler - Code

Dabei hat der unterstrichene C-Code links genau die gleiche Funktionalität wie der Assemblercode rechts.

Man sieht schon hier einen deutlichen Unterschied im Umfang.
Außerdem läßt sich der linke Code in Abbildung 1 wesentlich besser lesen. Durch die Anlehnung des C-Befehlssatzes an das Englische Vokabular kann sehr schneller verstanden werden, was ein bestimmter Programmteil ausführt. (z. B. if-Anweisungen). So läßt sich auch nach größeren Zeiträumen schneller nachvollziehen, welche Funktionalität der C - Code erfüllen soll. Ein Nachteil der Hochsprache der Umfang des erzeugten Codes. Bei Assembler-programmierung läßt sich der Code geringer halten. Jedoch ist der Speicher des FHOP so groß ausgelegt, so daß speicherplatzkritische Anwendungen bis jetzt nicht aufgetreten sind. Deshalb wird dieser Nachteil gerne in Kauf genommen, da die oben an - gesprochenen Vorteile überwiegen.

3 Wie wird ein Compiler erstellt?

Um die Vorteile der verwendeten Tools zu erkennen, möchte ich hier kurz auf die Funktionsweise eines Compilers eingehen.
Zuerst wird der Source-Code einer Überprüfung unterzogen. Wird diese erfolgreich abgeschlossen, d.h. der Source-Code ist fehlerfrei, wird versucht, den Source-Code durch eine Baumstruktur darzustellen. Abbildung 2 zeigt eine solche Baumstruktur.

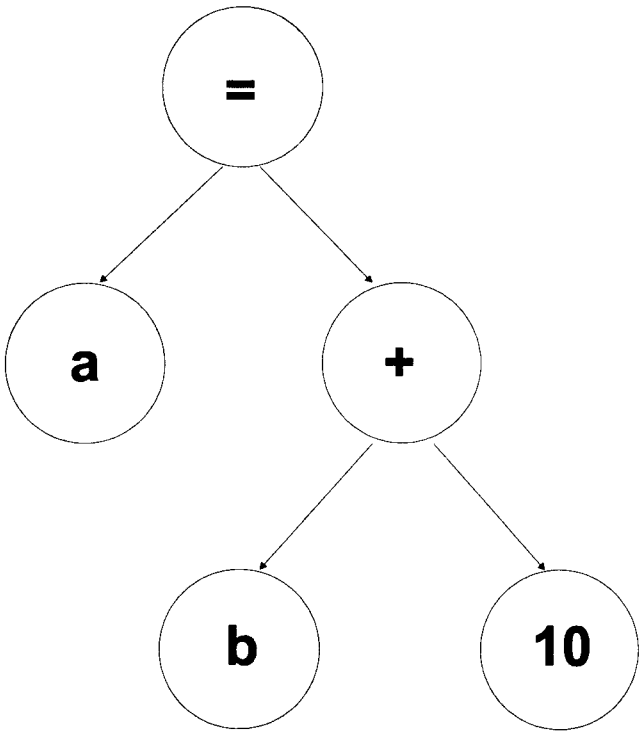

Abbildung 2 Baumstruktur

Die abgebildete Baumstruktur ergab die C-Anweisung „a = b + 10". Nachdem nun der gesamte Source-Code durch solche Baumstrukturen abgebildet wurde, versucht der Compiler diese Strukturen mit Assemblerbefehlen zu überdecken, d.h. in Assemblercode umzusetzen. Für die Baumerzeugung und die Überdeckung wurden nun Compilerbau-Tools verwendet. In Abbildung 3 wird der Ablauf „Source - Code → Maschinencode" schemenhaft dargestellt.

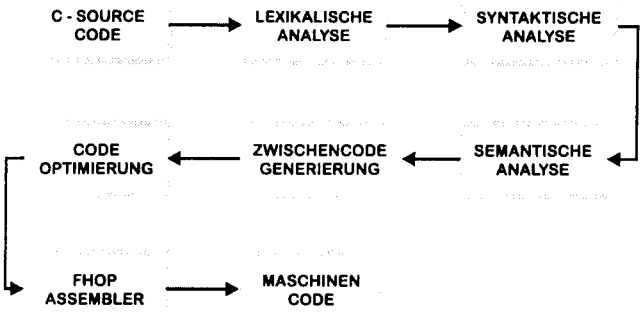

Abbildung 3 - Stufen der Codeerzeugung

Anhand dieses Schemas lassen sich die einzelnen Stufen der Umsetzung vom Source-Code in Assemblercode gut nachvollziehen. Das C-Source-File ist der Ausgangspunkt. Diese wird zuerst auf Richtigkeit überprüft und dann in 2 Schritten in Maschinencode umgesetzt. Schritt 1 ist die Erzeugung von Assemblercode. Dieser wird dann durch den Crash-Assembler in Maschinencode gewandelt. Der erste Schritt wird mit Hilfe von Compilerbau-Tools durchgeführt. Diese Tools helfen dem Anwender bei der Umsetzung der einzelnen Compilerfunktionen. Folgende Tools kamen dabei zum Einsatz:

3.1 flex: Scanner - Tool

Der Compiler erhält das vollständige File. Dieses muß zuerst auf Fehler durchsucht werden. Um diese Überprüfung durchführen zu können, muß dem Compiler bekannt sein, welche Ausdrücke zulässig sind und welche nicht .
Das Absuchen des C-Sourcefiles „von Hand" wäre sehr umständlich. Diese Arbeit übernimmt das Tool flex. flex ist die GNU-Variante des Standard-UNIX-Tools lex, welches bei allen UNIX-Betriebsystemen mit ausgeliefert wird. flex hat den Vorteil, daß es ein Freeware-Tool ist.
Diesem Tool werden nun die erlaubten Stringfolgen bzw. Befehle in einem Definitionsfile bekannt gemacht. Aus diesem File erzeugt flex dann ein C-File, welches dann compiliert die Funktion des Scanners übernimmt, der das gesamte File nach Fehlern in den Stringketten absucht. Diese Fehlersuche entspricht der lexikalischen Analyse, dargestellt als 2 Block in Abbildung 3.

3.2 bison: Parser - Tool

Die darauf folgende syntaktische Analyse erfolgt mit Hilfe des Tools bison. Bison ist ebenso wie flex eine GNU-Variante eines Standard-UNIX-Tools.
Wie auch bei lex, wird hier mit Hilfe eines Definitionsfiles dem Tool bekanntgemacht, welche Ausdrücke zulässig sind und welche nicht.
Bison wandelt das Definitionsfile in C-Code um, der, nachdem er compiliert wurde, die Funktion eines Parsers übernimmt.
Beim Parsen des Files wird nun nicht mehr auf die Richtigkeit einzelner Zeichenketten geachtet, sondern auf den sinnvollen Zusammenhang der Zeichenketten, die vom Scanner erkannt wurden.
So müssen bei bestimmten Operationen gewisse Bedingungen erfüllt sein. Bei einer mathematischen Operation, wie z.B. der Addition, müssen zwei Operanden zur Verfügung stehen. Dies heißt nun für den Parser, das beim Auffinden eines Additionszeichens der Scanner genau zwei Operanden liefern muß. Diese können direkt Zahlenwerte, Variablen oder auch beliebig komplexe Ausdrücke sein. Jedoch muß die Bedingung der zwei Operanden erfüllt sein. Ist dies nicht der Fall, so wird das Parsen mit einer Fehlermeldung abgebrochen.

3.3 Ox: Scanner & Parser - Tool

Ox ist ein Tool, welches die beiden vorherigen Tools, flex und bison unterstützt.
Beim Parsen und Scannen ist es notwendig, mehr Informationen als nur die vorgefundene Zeichenkette weiterzugeben. Dies läßt relativ gut an einem Beispiel zeigen.
Der Ausdruck „int a", definiert eine Variable „a" vom Typ „int". Trifft man nun im späteren Verlauf wieder auf diese Variable „a", z.B. bei „a = 0", so erkennt man nun zwar den Variablennamen „a", der Typ wird jedoch nicht mehr mit angegeben. Diese Information muß also irgendwo gespeichert werden. Nun wäre es möglich bei flex und bison eine Information auszutauschen. Jedoch ist es nicht möglich mehrere solcher Informationen untereinander auszutauschen. Und hier schaltet sich Ox ein. Ox ermöglich diesen Datenaustausch. Und dies auf eine sehr einfache und elegante Weise, so daß beliebig viele Informationen ausgetauscht werden können.

3.4 iburg: Code - Generating Tool

Nachdem nun mit Hilfe der ersten drei Tools der C-Code in eine Baumstruktur umgesetzt werden konnte, steht nun das Problem an, wie setzte ich diese Struktur wieder in ausführbaren Code um, im Falle des FHOP-C-Compilers in Crash-Assemblercode.
Zur Lösung dieses Problems gib es nun das Tool iburg. iburg hilft bei der Umsetzung der Baumstruktur in Assemblercode. Wie bei flex und bison wird diesem Tool durch ein Definitionsfile mitgeteilt, welche Assemblerbefehle zulässig sind. Aus diesem Definitionsfile erzeugt iburg ein C-File. Diesem File, nachdem es compiliert wurde, wird die erzeugte Baumstruktur zugeführt. Iburg erzeugt dann aus der zugeführten Baumstruktur und den Assemblerbefehlen ein lauffähiges Assemblerfile.

3.5 Alle Tools zusammen -> Compiler

Nimmt man nun alle C-Codefiles, die von den Compilerbau-Tools erzeugt wurden, und linkt diese zusammen, erhält man einen Compiler. Dieser Compiler setzt dann, die ihm zugeführte Hochsprache, im Falle des FHOP-C-Compilers die Sprache C, in Maschinencode um.
Diese Art des Compilerbaus ist sehr komfortabel, weil die standardisierten Abläufe bei der Compilierung eines Sourcecodes durch die Tools übernommen werden.
Mit diesen Tools lassen sich auch Compiler für andere Hochsprachen erzeugen, sofern man die Befehle und Grammatik für den Sprache besitzt.
Aber auch die Generierung von Assemblern mit diesen Tools ist möglich, da bei Assembler keine komplizierten Grammatiken vorliegen, so daß man mit diesen Tools sehr schnell zu einem Ergebnis gelangen kann.

4 Was kann der Compiler?

Der Compiler wurde als Kommandozeilen - Compiler konzipiert. Dabei soll in der Befehlszeile außer dem Aufruf verschiedene Dinge angegeben werden können. So z.B. den Beginn des Data-Segmentes, das Data-Limit, der Beginn des Stackbereichs, u. a.. Da zunächst die Tools zum Compilerbau nur auf UNIX zur Verfügung standen, wurde begonnen auf UNIX zu entwickeln. Nachdem jedoch die Tools erfolgreich auf DOS-Ebene portiert wurden, konnte der Compiler ebenfalls auf der DOS-Plattform generiert werden.
Auf der DOS-Ebene sind bis jetzt folgende Befehle implementiert:

- Variablen:
  - int - Variablendeklaration
  - const int - Variablendeklaration
  - Eindimensionale int - Arrays
- Mathematische Operationen:
  - +, -, *, /, Modulo-Fkt
  - Incrementieren & Decrementieren
  - Klammersetzung

- Bitoperationen:
  - AND
  - OR
  - XOR
- Logische Operationen
  - AND
  - OR
- Schiebeoperationen
  - >> (Schiebe nach rechts)
  - << (Schiebe nach links)
- Vergleichsoperationen
  - == (Gleich)
  - != (Ungleich)
  - < (Kleiner als)
  - > (Größer als)
  - >= (Größer gleich)
  - <= (Kleiner gleich)
- Kommentarzeilen
- Inline - Befehle
  - 1 zu 1 - Übernahme der angeführten Assemblerbefehle
- if - Abfrage
  - einfache if - Abfrage
  - if - else - Abfrage
  - if - else if - Abfrage
  - if - else if - else Abfrage
- while - Schleifen
- for - Schleifen

Dieser Befehlsumfang, so hat sich gezeigt, genügt in erster Linie den Ansprüchen der Problematik der Assemblerprogrammierung. Deshalb wird dieser Befehlsumfang zuerst in einigen Projekten tiefergehend getestet. Dabei soll die Tauglichkeit und Zuverlässigkeit des Compilers geprüft werden. Spätere Erweiterungen des Befehlsumfanges sind angedacht und jederzeit möglich, so daß einer Erweiterung des Befehlsumfanges nach erfolgreichen Tests nichts im Wege steht.

Zum Schluß möchte ich noch kurz zwei Screen-Shots zeigen: die Help-Seite und eine erfolgreiche Compilierung:

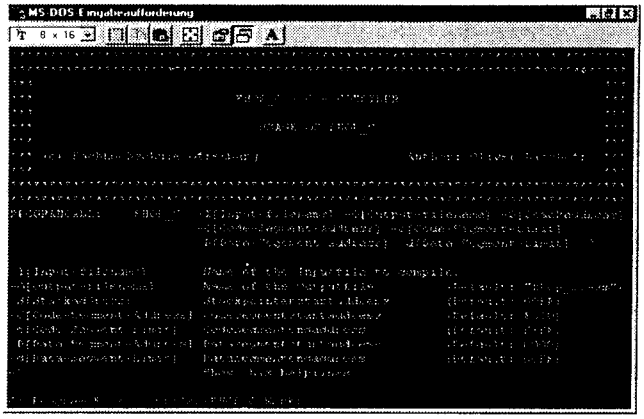

Abbildung 4 Die Help-Seite

Abbildung 5 Eine erfolgreich Compilierung

5 Literatur

[1] HP-UX Reference Volume 1
SECTION 1: User Commands
Release 8.05

[2] Kurt M. Bischoff
Ox, Tutorial Introduction
November 1993

[3] Christopher W. Fraser
Engineering a Simple, Efficient Code Generator Generator

[4] Brian W. Kernighan
Dennis M. Richtie
The C - Programming
Language 2nd Ed. Prentice Hall, 1988

Einfache Lösungen für Digital-Analog-Wandler und für Analog-Digital-Wandler

Martin Rieger
Fachhochschule Albstadt-Sigmaringen
Johannesstraße 3, 72458 Albstadt,
Tel. 07431/579-124, email: rieger@fh-albsig.de

Kurzfassung

Es werden Wege aufgezeigt, wie in rein auf Digitalschaltungen ausgerichteten Technologien, Digital-Analog-Wandler und Analog-Digital-Wandler implementiert werden können. In unserem Labor wurden verschiedene Entwürfe zu diesem Themenbereich in PLDs realisiert. Die Tests zeigen, daß die Entwürfe funktionsfähig sind. Sigma-Delta-Wandler für Videosignale, sind wesentlich kompakter als vergleichbare Parallelwandler. Ein experimenteller Sigma-Delta-Analog-Digital-Wandler, der in einer Silizium-Technologie mit f_T = 20 GHz realisiert wurde, zeigt die Leistungsfähigkeit dieses Konzepts.

1. Einleitung

Seit einigen Jahren ist in der Halbleitertechnik der Trend hin zur Digitaltechnik und zu CMOS-Technologien klar erkennbar [1].
Insgesamt gesehen verlieren analoge Komponenten gegenüber digitalen Komponenten an Bedeutung. Daher wird bei der Entwicklung neuer Halbleitertechnologien mehr Gewicht auf die Optimierung digitaler Komponenten gelegt, die Entwicklungsgeschwindigkeit analoger Komponenten ist deutlich geringer.
Aus dieser Beobachtung heraus entsteht der Wunsch, Schaltungen, die bisher aus analogen Komponenten aufgebaut waren, soweit wie möglich aus digitalen Komponenten aufzubauen. Im Rahmen dieser Arbeit werden mehrere Ansätze für Digital-Analog-Wandler (DAC) und Analog-Digital-Wandler (ADC) gezeigt, bei denen die Anforderungen hinsichtlich der analogen Komponenten stark reduziert sind.
Ein weiteres Ziel der Arbeiten mit DACs und ADCs ist es, für einschlägige Vorlesungen und Praktika interessante und anschauliche Beispiele zu schaffen.

2. Digital-Analog-Wandler

2.1. Digital-Analog-Wandler mit Widerstandsnetzwerk

Grundlegende Ansätze für CMOS-DACs sind seit langem bekannt [2].
Eine einfache Lösung ist in Bild 1 dargestellt.

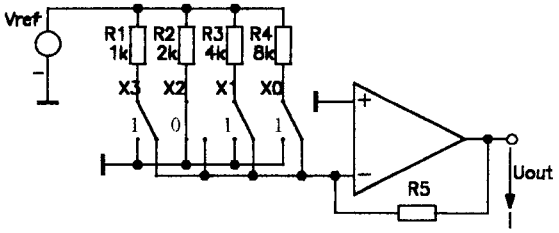

Bild 1: DAC mit binär gewichteten Widerständen und Operationsverstärker

Die binär gewichteten Widerstände verursachen entsprechend binär gewichtete Strömen, die je nach Schalterstellung X_i zur Masse oder zum Summationspunkt am Operationsverstärker fließen. Die binären Werte bestimmten die Schalterstellungen X_i, wobei die höchstwertige Stelle X_3 den höchsten Strom schaltet. Damit erhält man für V_{out} in Bild 1

$$V_{out} = -V_{ref} \cdot \left(\frac{1}{2} \cdot X_3 + \frac{1}{4} \cdot X_2 + \frac{1}{8} \cdot X_1 + \frac{1}{16} \cdot X_0 \right)$$

$$= -V_{ref} \cdot \frac{X}{X_{max} + 1},$$

wobei X der zu wandelnde binäre Wert und X_{max} der größte binäre Wert ist.

Für den in Bild 1 dargestellten DAC werden die Komponenten

Binär gestuftes Widerstandsnetzwerk,
Um- oder Einschalter und
Verstärker

benötigt. In rein digital ausgerichteten Technologien stehen diese Komponenten so nicht zur Verfügung und es erhebt sich die Frage, wie Ersatz beschafft werden kann.

Für die Widerstände kann in einfacher Weise kein direkter Ersatz gefunden werde, so daß diese Komponente außerhalb des ICs als externe Komponente verwendet werden müssen.

Anstelle der Schalter können Transfergates eingesetzt werden.

Der Verstärker kann durch einen Inverter ersetzt werden, wie dies in Bild 2 dargestellt ist. Allerdings sind Einschränkungen gegenüber

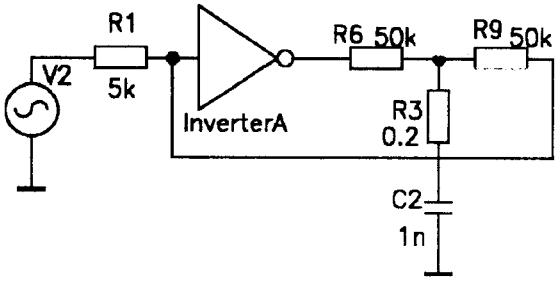

Bild 2: CMOS-Inverter als Operationsverstärker

einem üblichen Operationsverstärker zu beachten. Der Inverter erfordert aus Stabilitätsgründen eine Kompensation. Zudem kann am Eingang des Inverters das Bezugspotential nicht frei gewählt werden, vielmehr liegt das Bezugspotential fest bei der Entscheiderschwelle des Inverters.

Bild 3 zeigt eine Realisierung, wie sie in unserem Labor erprobt wurde. Die Spannungsquellen V_1 bis V_4 repräsentieren den zu wandelnden binären Wert. Die gestrichelt gezeichneten Einheiten sind in einem PLD integriert.

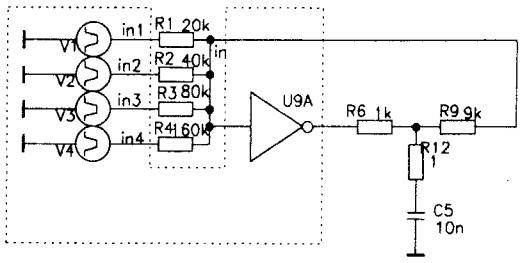

Bild 3:DAC mit CMOS-Inverter und binär gestuftem Widerstandsnetzwerk

Es wurden noch weitere DACs mit Widerstandsnetzwerk untersucht wobei R-2R-Netzwerke Verwendung fanden. Diese Netzwerke haben gegenüber binär gestuften Widerstandsnetzwerken den Vorteil von gleichartigen und somit besser matchenden Widerständen.

2.2. Digital-Analog-Wandler mit Pulsweitenmodulator

In rein digital ausgerichteten Technologien können Widerstandsnetzwerke nicht integriert werden, so daß sie als externe Komponenten hinzugefügt werden müssen. Eine rein digitale Lösung ohne externe Komponenten wäre hier vorzuziehen. Einen Ansatz hierzu bildet der Pulsweitenmodulator (Bild 4a).

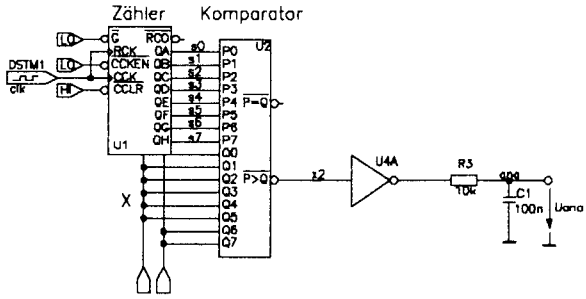

Bild 4a: Pulsweitenmodulator als DAC

Ein digitaler Komparator vergleicht den zu wandelnden binären Wert X mit dem Zählerstand s. Der Zähler inkrementiert seinen Stand mit jedem Clockimpuls, bis der maximale Zählerstand erreicht ist, dann springt der Zählerstand auf 0 und das Inkrementieren wird

fortgesetzt. Der Komparator-Ausgang x_2 ist high, solange s > X ist. Sobald s < X wird springt der Pegel bei x_2 auf low (Bild 4b).

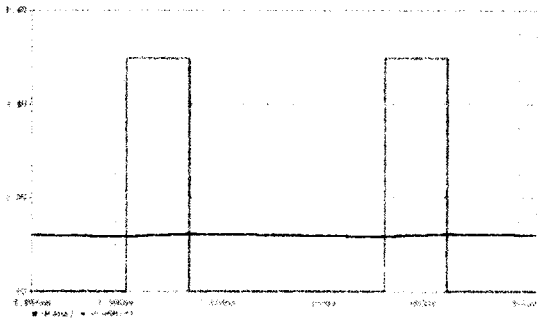

Bild 4b: Simulationsergebnis zu Bild 4a: V(U4) (Pulse), V(ana) (Gleichwert)

Natürlich ist diese Kurvenform an x_2 noch nicht brauchbar. Ein nachfolgendes Tiefpassfilter unterdrückt die höheren Frequenzanteile weitgehend, so daß V_{ana} nahezu frei von Störanteilen ist.
Den niederfrequenten Spannungsanteil erhält man zu

$$V_{ana} = V_{ref} \cdot (1 - \frac{X}{2^n - 1}).$$

Damit ist es tatsächlich gelungen, den DAC mit hoher Auflösung fast vollständig zu integrieren, lediglich das Tiefpassfilter ist extern.
Ein Nachteil des Pulsweitenmodulators liegt allerdings in der begrenzten Geschwindigkeit. Einfache Überlegungen zeigen, daß die Grenzfrequenz f_g des Tiefpasses

$$f_g \leq \frac{f_{clk}}{2^n}$$

sein muß, wobei f_{clk} die Frequenz des Clocks und n die Auflösung des DACs sind.
Für eine akzeptable Einschwingzeit des DACs sind die Anforderungen an die Clockfrequenz und somit an die Schaltgeschwindigkeit der Komponenten recht hoch.
Ein DAC ähnlich zu Bild 4a wurde in unserem Labor mit einem PLD realisiert. Die Programmierung des PLDs wurde in VHDL vorgenommen. Bei einer Auflösung von 8 Bit und einer Taktfrequenz von 40 MHz wurde die Grenzfrequenz auf 1 kHz gelegt, so daß die Wandelgeschwindigkeit maximal 1kHz beträgt.

Eine wesentliche Verbesserung hinsichtlich der Geschwindigkeit brächte die Anwendung eines Sigma-Delta-Wandlers [3]. Dabei wird, anschaulich ausgedrückt, die high- und low-Zeit in Bild 4b feiner verteilt, so daß die Grenzfrequenz fg wesentlich höher gewählt werden darf. Überlegungen zu einem DAC nach dem Sigma-Delta-Prinzip sind bei uns im Gange.

3. Analog-Digital-Wandler

3.1. Analog-Digital-Wandler nach dem Wägeverfahren

Das wohl naheliegendste Verfahren zur Analog-Digital-Wandlung ist das Parallelverfahren [2] (Bild 5).

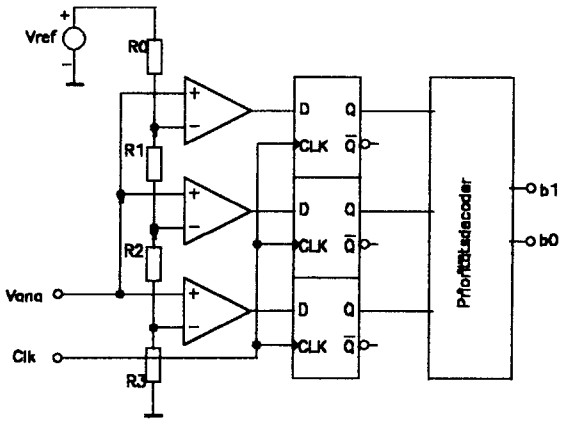

Bild 5: Parallelwandler

Dabei wird das zu digitalisierende Signal V_{ana} in einer Kette von analogen Komparatoren mit Bruchteilen der Referenzspannung V_{ref} verglichen. Die Ergebnisse des Vergleichs werden in Flipflops gespeichert und im Prioritätsdecoder in binäres Format umgewandelt.
Parallelwandler haben zwar den Vorteil sehr hoher Wandelgeschwindigkeit, dem stehen aber die Nachteile hoher Leistungs- und Flächenverbrauch, nur mittlere Auflösung und schlechte Integrierbarkeit in digitale Technologien gegenüber.
Interessanter erscheinen Verfahren, die nach dem Wägeprinzip arbeiten [2] (Bild 6). Hier wird mittels eines Wägeverfahrens eine analoge Spannung U_z so nachgeführt, daß sie der

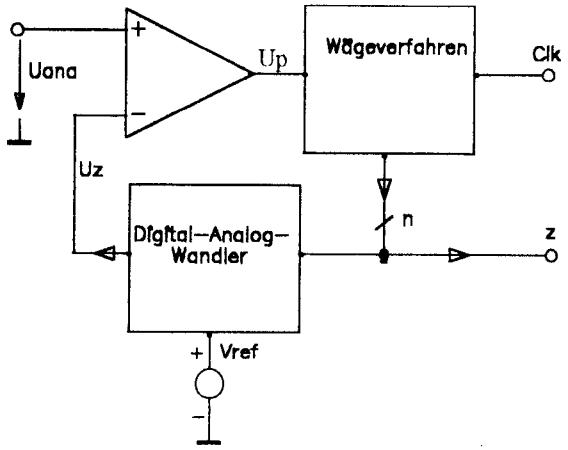

Bild 6: ADC mit Wägeverfahren

zu wandelnden Spannung U_{ana} möglichst ähnlich wird. Je nach verwendetem Wägeverfahren und je nach Typ des Digital-Analog-Wandlers erreicht man mit ADCs nach dem Wägeverfahren eine mäßige bis hohe Umsetzgeschwindigkeit. Der Aufwand ist eher gering und bei konsequenter Ausnutzung aller schaltungstechnischen Möglichkeiten ist dieser Wandlertyp auch in rein digitalen Technologien gut integrierbar.
Es sind verschiedene Wägeverfahren möglich, ein recht leistungsfähiges Verfahren arbeitet mit sukzessiver Approximation. Das Verfahren ist in Bild 7a am Beispiel eines 3 Bit Wandlers dargestellt. Es wird, beginnend mit dem höchstwertigen Bit und dann absteigend in Richtung niederwertigstem Bit, eine Bitstelle nach der anderen probeweise gesetzt. Ist $U_{ana} > U_z$, d.h. Up=1, so bleibt das geprüfte Bit gesetzt, ansonsten wird das geprüfte Bit rückgesetzt.

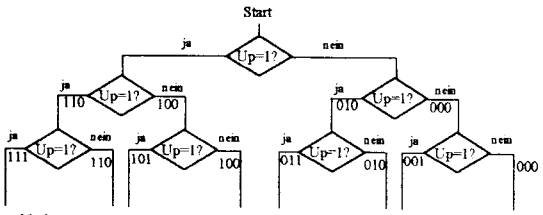

Bild 7a: Wägeverfahren mit sukzessiver Approximation

In unserem Labor wurde ein ADC nach dem Verfahren der sukzessiven Approximation in VHDL entworfen und in ein PLD implementiert. Der DAC in der Rückführung des ADCs arbeitet mir Pulsweitenmodulation, wie in Bild 4a erläutert. Dabei ist das Tiefpassfilter des DAC, wie in Bild 7b gezeigt, mit der Komparatorschaltung verknüpft. Der Inverter übernimmt, zusammen mit den Widerständen R_l und R_g die Funktion des Komparators. Die rückgewandelte Spannung V_z stellt sich so ein, daß

$$V_z = -V_{ana}\frac{R_1}{R_g} - V_{th}\frac{R_1 + R_g}{R_g}$$

wird, wobei V_{th} die Entscheiderschwelle des Inverters ist, d.h. V_z stellt ein Abbild von V_{ana} dar.
Somit konnte der gesamte ADC, mit Ausnahme von R_l, R_g und C_g, in eine rein digital ausgerichtete Technologie integriert werden.

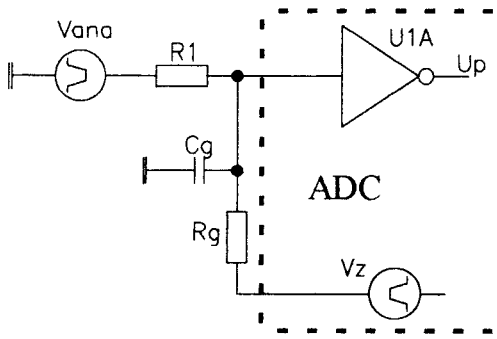

Bild 7b Realisierter ADC

3.2. Analog-Digital-Wandler mit Sigma-Delta-Modulator

ADCs mit Sigma-Delta-Modulator sind in der Verarbeitung von Audiosignalen schon lange im Einsatz, während für Videoanwendungen bisher Parallelwandler nach Bild 5 verwendet werden. Wie schon geschildert, weisen diese Parallelwandler aber einige Nachteile auf, so daß auch hier Wandler mit Sigma-Delta-Wandler eine interessante Alternative darstellen.
Bild 8a zeigt einen Sigma-Delta-Wandler 2. Ordnung. Der digitale Datenstrom d(t) stellt sich so ein, daß er den analogen Momentanwert x(t) möglichst gut nachbildet.

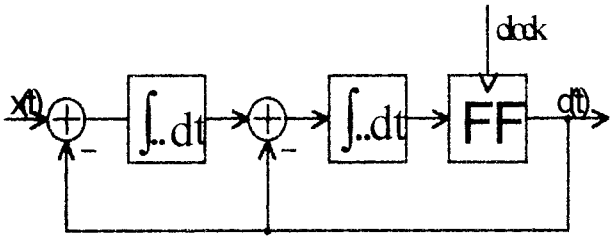

Bild 8a: Sigma-Delta-Wandler 2. Ordnung

Von der Differenz x(t) - d(t) der äusseren Schleife wird das Integral gebildet. Vom Ergebnis der Integration wird in der inneren Schleife d(t) subtrahiert und diese Differenz wird ebenfalls integriert. Ein Flipflop dient zur Speicherung und als 1-Bit-ADC. Dieses Flipflop wir mit einer sehr hohen Taktfrequenz f_{clk} betrieben, die sehr viel größer als die höchste im analogen Signal vorkommende Frequenz f_s ist. Der digitale 1-Bit Datenstrom d(t) wird von einem folgenden Dezimationsfilter in ein paralleles Format gewandelt.

Ein Sigma-Delta-Wandler 1. Ordnung kann ein Signal-Rausch-Verhältnis

$$SNR = 20 \cdot \log\left(\frac{\sqrt{6}}{\pi} \cdot \left(\frac{f_{clk}}{2 \cdot f_s}\right)^{3/2}\right)$$

erzielen, während ein Sigma-Delta-Wandler 2. Ordnung

$$SNR = 20 \cdot \log\left(\frac{\sqrt{60}}{\pi^2} \cdot \left(\frac{f_{clk}}{2 \cdot f_s}\right)^{5/2}\right)$$

erreichen kann [3]. Eine Abschätzung zeigt, daß mit einem Oversampling-Verhältnis

$$\frac{f_{clk}}{2 \cdot f_s} = 64$$

ein Signal-Rausch-Verhältnis besser als 60 dB erreichbar sein sollte.

In Zusammenarbeit mit der Firma Thomson-Multimedia wurde ein experimenteller Sigma-Delta-Wandler entworfen und hergestellt [4], [5], [6].

Zur Realisierung stand eine 20 GHz Silizium Bipolartechnologie zur Verfügung. Schon in der Simulation zeigte sich, daß die Transistoren dieser Technologie nicht so schnell schalten, wie dies idealerweise für einen Sigma-Delta-Wandler 2. Ordnung wünschenswert ist. Wegen der mangelhaften Schaltgeschwindigkeit arbeitet die innere Schleife des Wandler nur unzureichend. Das erreichbare Signal-Rausch-Verhältnis liegt somit näher bei dem für einen Sigma-Delta-Wandler 1. Ordnung erwarteten Wert, als bei dem Wert, der einen Sigma-Delta-Wandler 2. Ordnung erhofft werden kann.

Bild 9a zeigt den in ein 20-Pin LCC-Gehäuse eingebauten Die. Die Gesamtfläche des Chips beträgt 6 mm^2, aber die aktive Fläche des Sigma-Delta-Wandlers nimmt davon nur 0.45 mm^2 in Anspruch. Der größte Teil der Fläche ist für Testschaltungen, ECL-Treiber und für Abblockkondensatoren verwendet. Der Modulator-Kern verbraucht eine Leistung von 250 mW an 5V. Die nominale Taktfrequenz beträgt 1.28 GHz.

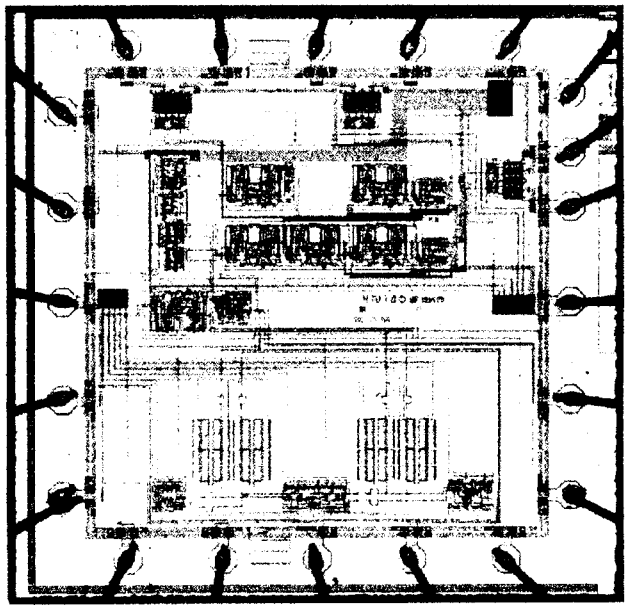

Bild 9a: Chip-Photo des Sigma-Delta-Wandlers

In Bild 9b ist das gemessene Signal-Rausch-Verhältnis in Abhängigkeit von der Clockfrequenz dargestellt. Die Messungen zeigen gute Übereinstimmung mit Spice-Simulationen. Wie aus der Simulation zu erwarten, wird für das Signal-Rausch-Verhältnis etwa 45 dB erreicht. Diese Performanz ist für viele Anwendungen, wie z. B. digitales Fernsehen, ausreichend.

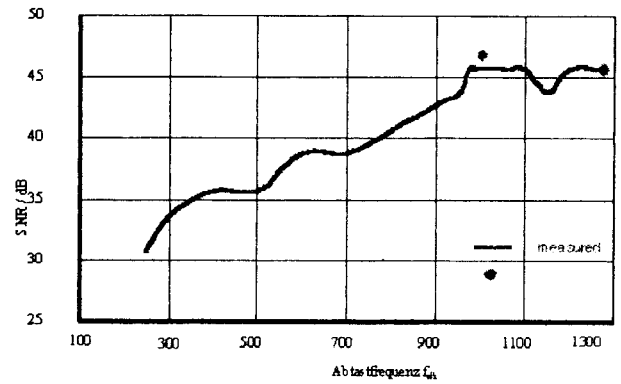

Bild 9b: Signal-Rausch-Verhältnis des Sigma-Delta-Wandlers

4. Zusammenfassung

Es konnte gezeigt werden, wie in rein auf Digitalschaltungen ausgerichteten Technologien, in denen keine Widerstände und Analog-Komparatoren zur Verfügung stehen, Digital-Analog-Wandler und Analog-Digital-Wandler implementiert werden können. Die Entwürfe wurden zusammen mit Studenten ausgeführt, wobei die Erfahrung gewonnen wurde, daß sich diese Art von DACs und ADCs gut für Entwurfsarbeiten im Rahmen von Praktika eignet.

In unserem Labor wurden verschiedene Entwürfe von DACs und ADCs in PLDs realisiert. Die Tests zeigen, daß die Entwürfe funktionsfähig sind.

Eine Unzulänglichkeit in unserer Simulationsumgebung besteht zur Zeit noch darin, daß Schaltungen mit VHDL-Komponenten und mit Analog-Komponenten nur getrennt nach Analog- und Digitalteil simuliert werden können, was für die ADC-Simulation recht störend ist. Sobald dieser Mangel behoben ist und somit zuverlässige Simulationen möglich sind, werden die DAC und ADC-Entwürfe in einem ASIC realisiert.

In Zusammenarbeit mit Thomson-Multimedia entstand ein experimenteller Sigma-Delta-Wandler für Videosignale. Dieser Ansatz ermöglicht ADCs, die wesentlich kompakter sind als die bei der Videoverarbeitung bisher verwendeten Parallelwandler. Die Schaltung, die in einer Silizium-Technologie mit f_T = 20 GHz realisiert. erwies sich als funktionsfähig. Allerdings wäre eine noch schnellere Technologie wünschenswert, um einen Sigma-Delta-Wandler 2. Ordnung mit hohen Taktfrequenz von 1.2 GHz ideal zu realisieren.

Literaturverzeichnis

[1] The National Technology Roadmap for Semiconductors: Semiconductor Industry Association (SIA) 1997 edition report published by SEMA-TECH, Inc.

[2] A. B. Grebene: "Bipolar and MOS Analog Integrated Circuit Design" John Wiley & Sons 1983.

[3] J.C. Candy, G. C. Temes: "Oversampling Methods for A/D and D/A Conversion". Oversampling delta-sigma data converters: theory, design and simulation, IEEE Press 1992

[4] M. Erbar, M. Rieger, H. Schemmann: A 1.28-GHz Sigma-Delta Modulator for Video A/D Conversion, ICCE 1996 digest of technical papers S. 78,79 ISSN 0-7803-3029-3

[5] M. Erbar, M. Rieger, H. Schemmann: The Fastest Sigma-Delta A/D Converter on Si, Nikkei Electronics 1996.10.7 (no. 672) S. 123 - 130 ISSN 0385-1680

[6] M. Erbar, M. Rieger, H. Schemmann: A 1.28-GHz Sigma-Delta Modulator for Video A/D Conversion, IEEE Transactions on Consumer Electronics, Aug. 1996, vol. 42, No. 3 S. 357 - 361 ISSN 0098-3063