Modellierung eines Mikroprozessors in SystemC

Daniel Große[1] Ulrich Kühne[1] Christian Genz[1] Frank Schmiedle[2]
Bernd Becker[2] Rolf Drechsler[1] Paul Molitor[3]

[1]*Institut für Informatik, Universität Bremen, 28359 Bremen*
[2]*Institut für Informatik, Albert-Ludwigs-Universität Freiburg, 79110 Freiburg*
[3]*Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg, 06120 Halle*

Kurzzusammenfassung Um die steigende Komplexität in modernen Systemen handhaben zu können, müssen zunehmend abstraktere Modelle betrachtet werden. Während noch vor Jahren Hardware ausschließlich in Hardware-Beschreibungssprachen (HDL=hardware description language) spezifiziert wurde, werden zunehmend C-artige Beschreibungen verwendet, um schon in frühen Entwurfsphasen ausführbare Spezifikationen zu erhalten.

In dieser Arbeit wird zu einem Prozessor, der in [1] als „Modellprozessor" für die Lehre eingeführt wurde, eine Modellierung in SystemC beschrieben. Es wird exemplarisch ein Assembler-Programm diskutiert und Simulationsergebnisse werden gezeigt.

1. Einleitung

Der Entwurf einer Schaltung erfolgt (in der Regel auf der Register-Transfer-Ebene) mittels einer speziellen Hardwarebeschreibungssprache, wie z. B. VHDL oder Verilog. Da bei heutigen Systemen sowohl die Komplexität als auch der Anteil an Systemfunktionen, die in Software implementiert werden, immer mehr zunimmt, werden in der Praxis auf Systemebene häufig zunächst Referenzmodelle in simulierbaren Beschreibungssprachen, wie C, erzeugt. Um die entstehende Lücke zu VHDL/Verilog zu schließen, wurden verschiedene Erweiterungen von C/C++ vorgeschlagen, die es ermöglichen, Hardware zu modellieren [4]. In diesem Zusammenhang wurde die Systembeschreibungssprache SystemC [8] eingeführt, bei der es sich um eine C++-Klassenbibliothek handelt. SystemC ist frei verfügbar, unterstützt Beschreibungen auf hohen Abstraktionsebenen und besitzt eine hohe Simulationsgeschwindigkeit. Für die Verifikation von Systemen ist ein zyklengenauer Simulator integriert.

In dem Lehrbuch „Technische Informatik — Eine Einführung" von Becker et.al. [1] wurde ein kleiner Ein-Zyklen-Mikroprozessor namens `Zykluno` „gebaut", um den Studierenden den Aufbau einer Maschinensprache nahebringen und deren Interpretation durch Hardware erläutern zu können. Ein vertieftes Verständnis der Funktionsweise dieses „Modellprozessors" können die Studierenden, ähnlich wie die Entwerfer im industriellen Umfeld, aber nur dadurch wirklich bekommen, wenn es ihnen möglich ist, die Abarbeitung kleinerer Assemblerprogramme auf der Hardware zu beobachten und nachzuvollziehen.

So steht den Studierenden zum Beispiel zum Erlernen der Maschinensprache der MIPS-Mikroprozessor-Familie, die ausgiebig in dem Lehrbuch von Hennessy und Patterson [7] behandelt wird, ein in C implementierter Simulator namens SPIM [3] für Lehrzwecke zur Verfügung. SPIM zeigt die Änderungen in den Registern nach jedem Schritt an, illustriert aber nicht, wie die Maschinenbefehle auf der Hardware durch geeignetes Setzen der Steuerleitungen interpretiert werden. Eines der Ziele des oben angesprochenen Lehrbuches [1] ist es, die digitale Ebene und die Maschinensprache eines Prozessors vorzustellen und auf die Interpretation eines Maschinenprogramms auf der digitalen Ebene einzugehen. Deshalb sollte den Studierenden ein Simulator unseres Modellprozessors „Zykluno“ zur Verfügung gestellt werden, der nicht nur die Auswirkungen der Maschinenbefehle auf die Belegung der Register anzeigt, sondern auch eine systemnahe Simulation auf Register-Transfer-Ebene ermöglicht.

Die Arbeit ist wie folgt gegliedert. Nach einer kurzen Vorstellung der SystemC zu Grunde liegenden Konzepte in Abschnitt 2 geht Abschnitt 3 auf die Modellierung des Modellprozessors Zykluno in SystemC ein. Es werden die wichtigsten Eckdaten des Prozessors vorgestellt — für weitergehende Informationen, die die Architektur von Zykluno betreffen, wird aus Platzgründen jedoch auf [1] verwiesen. Exemplarisch wird anschließend die Modellierung des Befehlszählers in SystemC diskutiert. Abschnitt 4 wendet sich der Simulation zu. Es wird zuerst ein Werkzeug zur Übersetzung des Zykluno-Assemblers in Zykluno-Maschinensprache vorgestellt, welches auf dem Parsergenerator PCCTS basiert. Anschließend gehen wir exemplarisch auf die Simulation eines Assembler-Programms zur Berechnung der Fibonacci-Folge ein.

2. Die Systembeschreibungssprache SystemC

SystemC erlaubt die Modellierung von Schaltungen und Systemen auf unterschiedlichen Abstraktionsebenen [2, 8]. Die wichtigsten Konzepte zur Modellierung von Systemen in SystemC werden im Folgenden kurz beschrieben:

1. *Module*: Module sind die „Grundblöcke“ des Systems, d.h. sie kapseln Teile der Schaltung. Ein Modul ist als C++-Klasse realisiert und kann Ports, lokale Daten, Prozesse und Untermodule enthalten, wodurch Hierarchiebildung möglich wird.

2. *Ports*: Ports sind die externen Schnittstellen eines Moduls. Sie ermöglichen die Übertragung von Daten mit anderen Modulen. Ports legen dabei die Richtung des Datenflusses fest, d.h. es gibt Eingangs-, Ausgangs- und bidirektionale Ports.

3. *Prozesse*: Mit Prozessen wird die Funktionalität eines Moduls beschrieben. Sie werden als spezielle Funktionen von Modulen deklariert und können sensitiv auf Eingangs- oder Clock-Signale sein. Durch Prozesse kann wie auch bei anderen HDLs Nebenläufigkeit modelliert werden.

4. *Signale*: Signale repräsentieren physikalische Leitungen und verbinden Module untereinander über ihre Ports, d.h. Signale transportieren die Daten, besitzen aber keine Richtungsinformation. Die Wertzuweisung an ein Signal geschieht wie in anderen HDLs, beispielsweise VHDL, nach dem Prinzip „verzögerte Zuweisung“ (*deferred assignment*).

5. *Clocks*: Clocks sind spezielle Signale, die zur Generierung des Taktes bei der Simulation verwendet werden.

6. *Datentypen*: Neben allen C++-Typen stehen beispielsweise auch vierwertige Logik-Typen, Festkomma-Typen und ganzzahlige Typen unbeschränkter Länge zur Verfügung.

Für Hardware-Entwürfe ist die Beschreibung von Verbindungen mit Signalen zwar ausreichend, aber auf höheren Abstraktionsebenen möchte man beispielsweise verzögerte, gepufferte Verbindungen und Software-Verbindungen (z.B. *remote procedure calls*) beschreiben können. Dafür stellt SystemC *interfaces* und *channels* zur Verfügung. Für detailliertere Informationen sei auf [8, 2, 5] verwiesen. Gegenüber HDLs besitzt SystemC das Merkmal eine ausführbare Spezifikation mit einem C++-Compiler zu erzeugen. Diese simuliert das modellierte System.

3. Modellierung des Mikroprozessors `Zykluno` in SystemC

3.1. Der Mikroprozessor `Zykluno`

`Zykluno` bezeichnet einen Modellprozessor, der in [1] eingesetzt wird, um grundsätzliche Konzepte und Arbeitsweisen eines Mikroprozessors zu vermitteln und zu erörtern. Neben dem Basismodell, dessen Befehlssatz sich auf arithmetische Verknüpfung von Daten, einfache Sprünge und die Kommunikation zwischen Registern, Datenspeicher und Recheneinheit beschränkt, gibt es auch eine erweiterte Version. Diese ist zusätzlich zu den Basisfunktionalitäten in der Lage, Daten mit Ein- und Ausgabegeräten auszutauschen oder sie mittels logischer Operationen zu verknüpfen. Zudem bietet die erweiterte Variante die Möglichkeit zur Unterprogramm- und Interrupt-Behandlung.

Seinen Namen verdankt der Prozessor `Zykluno` der Tatsache, dass es sich um einen Ein-Zyklen-Mikroprozessor handelt. Das bedeutet, dass alle zur Verfügung stehenden Maschinenbefehle vollständig innerhalb eines einzigen Taktzyklus' abgearbeitet werden können.

Weiterhin handelt es sich um einen Vertreter der Harvard-Architektur, Daten und Programme werden also getrennt voneinander in zwei separaten Speichern untergebracht. Beide Speicher arbeiten mit einer Wortbreite von 16 Bit, was u.a. zur Folge hat, dass auch die einzelnen `Zykluno`-Maschinenbefehle 16 Bits umfassen. Während der Datenspeicher ebenfalls mit 16 Adressleitungen ausgestattet ist und somit eine Kapazität von 128kByte aufweist, ist eine Adresse im Programmspeicher lediglich elf Bits lang. Eine Programmspeicher-Adresse kann deshalb komplett als Operand in einem Maschinenbefehl untergebracht werden.

Entsprechend der Wortbreite des Datenspeichers sind auch die Registerspeicher darauf ausgelegt, Datenworte der Breite 16 Bit zu speichern. Insgesamt verfügt `Zykluno` über acht Register, die in einer einzigen Registerbank zusammengefasst sind. Neben einem Schreibport enthält diese Registerbank zwei Leseports, sodass im Bedarfsfall zwei Operanden parallel aus den Registern gelesen werden können. Entsprechend gibt es drei Eingänge der Breite drei Bit zur Adressierung der gewünschten Register, mit denen über die entsprechenden Ports Daten ausgetauscht werden sollen.

Neben den Datenleitungen für Schreibzugriffe (16 Bits) sowie dem Takt- und *write enable*-Signal sind zwei Eingänge H und L vorhanden, durch die Schreibzugriffe auf einzelne Bytes ermöglicht werden.

Die 16-Bit-ALU, die in `Zykluno` integriert ist, erhält die zu verknüpfenden Operanden ausschließlich aus den Datenregistern. Über einen zusätzlichen *select*-Eingang wird eine der zur Verfügung stehenden ALU-Operationen ausgewählt, und nachdem diese auf den/die Operand(en) angewendet wurde, steht das Ergebnis am Ausgang bereit und kann in ein Register geschrieben werden. Insgesamt stellt

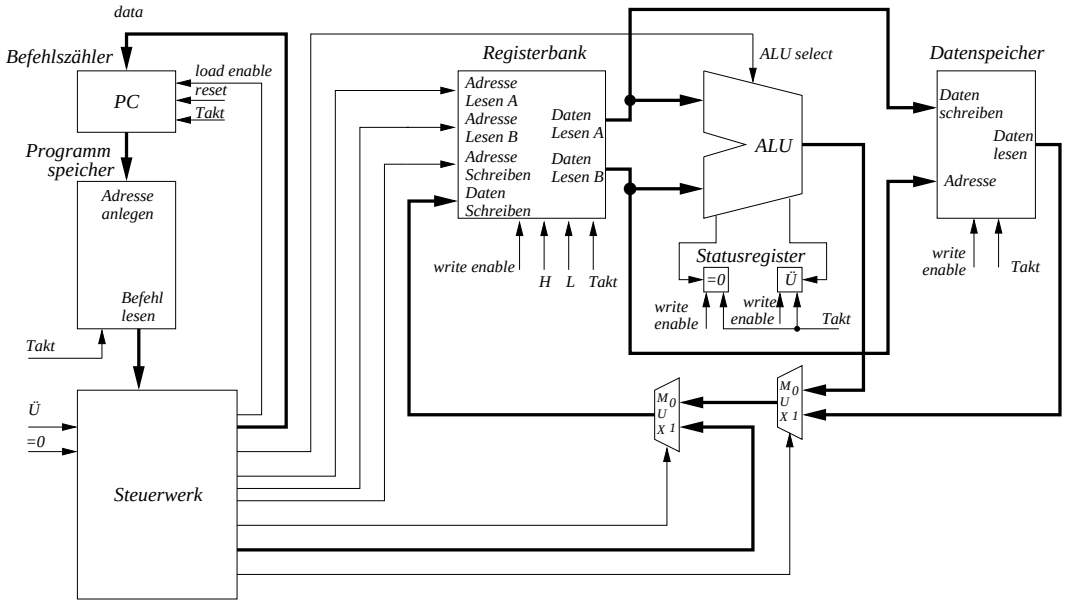

Abbildung 1. Blockschaltbild des Rechners `Zykluno`

die ALU von `Zykluno` jeweils acht arithmetische bzw. logische Operationen bereit, wobei vier der logischen Operationen zum erweiterten Befehlssatz gehören.

Neben den Datenregistern enthält der Prozessor zwei Spezialregister — in der erweiterten Version kommt noch ein drittes hinzu. Da `Zykluno` ein Ein-Zyklen-Prozessor ist, braucht das Befehlswort nicht zwischengespeichert zu werden, weshalb ein Instruktionsregister nicht benötigt wird. Hingegen steht ein, wenn auch nur aus einem *zero*- und einem *carry*-Bit bestehendes Statusregister zur Verfügung, sodass bedingte Sprünge möglich sind. Der Programmzähler steuert in der üblichen Weise den Programmablauf; dieses Register wird in Abschnitt 3.2 noch detaillierter behandelt. Wegen der Unterprogrammbehandlung wird für die erweiterte Version von `Zykluno` ein Stapelspeicher benötigt, und aus diesem Grund umfasst sie mit dem Stapelzeiger auch ein zusätzliches Spezialregister.

Der Gesamtaufbau von `Zykluno` ist in Abbildung 1 skizziert. Man erkennt alle angesprochenen Komponenten sowie ihre Verbindungen untereinander. Datenleitungen sind durch eine etwas dickere Darstellung hervorgehoben, während Verbindungen, die Adress- und Kontrollsignale transportieren, in dünnerer Stärke gezeichnet sind. Zusätzlich sind in der Abbildung zwei Multiplexer und das Steuerwerk eingezeichnet. Durch die Multiplexer kann ausgewählt werden, ob beim Beschreiben eines Registers Daten aus dem Datenspeicher, aus der ALU oder direkt aus dem Steuerwerk in die Registerbank kopiert werden. Diese Selektion wird durch das Steuerwerk vorgenommen, ebenso wie zahlreiche andere Kontrollleitungen vom Steuerwerk gesetzt werden. Beispiele hierfür sind die Adressierung der Registerbank sowie die Auswahl der auszuführenden ALU-Operation.

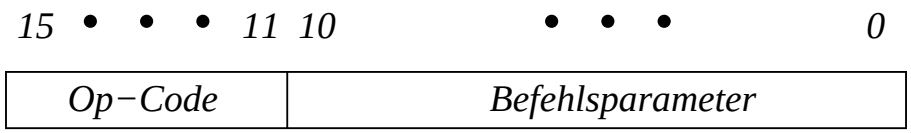

Abbildung 2. Struktur der Maschinenbefehle von `Zykluno`

Wir abstrahieren an dieser Stelle von den Interna des Steuerwerks, da diese für das Weitere eine untergeordnete Rolle spielen, und verweisen für Details auf [1]. Stattdessen gehen wir hier noch etwas näher auf den Befehlssatz von `Zykluno` ein. Wie bereits erwähnt, besteht ein Maschinenbefehl für diesen Prozessor aus 16 Bits. Davon werden 5 Bits für den Op-Code benötigt, folglich stehen für die Repräsentation im Befehl mitgeführter Operanden 11 Bits zur Verfügung (siehe Abbildung 2). Wegen der Breite des Op-Codes können $2^5 = 32$ verschiedene Befehle unterschieden werden. Da die beiden (bzgl. ihrer Funktionalität recht ähnlichen) Befehle zum Rücksprung von Unterprogrammen bzw. von einer Interrupt-Behandlung bei gleichem Op-Code durch das LSB unterschieden werden, umfasst der Befehlssatz der erweiterten `Zykluno`-Variante insgesamt 33 verschiedene Maschinenbefehle, die sich in 5 Befehlsklassen gliedern lassen:

- 6 Lade- und Speicherbefehle (Datentransfer zwischen Registerbank und Datenspeicher oder Ein-/Ausgabegerät, Laden des High- bzw. Low-Byte eines Registers mit einer Konstanten)
- 8 arithmetische Befehle (Addition/Subtraktion mit und ohne *carry*, Rotation und Shift links/rechts)
- 8 logische Befehle (Bitweise Negation, Bitweise Exklusiv-Oder-Verknüpfung, Konjunktion, Disjunktion zweier Operanden sowie Maskierung, Invertierung, Löschen oder Setzen eines einzelnen Bits eines Operanden)
- 5 Sprungbefehle (unbedingter Sprung, bedingter Sprung bei gesetztem/gelöschtem *carry*- oder *zero*-Bit)
- 6 sonstige Befehle (Stapelspeicher-Operationen Push und Pop, Programmhalt, Unterprogrammsprung, Rücksprung von Unterprogramm bzw. Interrupt-Behandlung)

3.2. SystemC Modell

Jede in Abbildung 1 dargestellte Komponente wurde als SystemC Modul mit entsprechender Funktionalität realisiert. Exemplarisch wird nun die Beschreibung für den Befehlszähler diskutiert. Da Adressen des Programmspeichers elf Bits umfassen, ist auch für den Befehlszähler eine Breite von elf Bit ausreichend. Abgesehen von Sprungbefehlen erfolgt der Programmablauf sequentiell, d. h., der Inhalt des Befehlszählers muss außer bei Sprüngen jedes Mal, wenn ein Befehl vollständig abgearbeitet ist, inkrementiert werden. Deshalb ist es sinnvoll, den Befehlszähler nicht in derselben Weise wie die Datenregister zu realisieren, sondern einen elf-Bit-Zähler als Befehlszähler zu verwenden. Dieser muss aber zusätzlich über folgende Funktionalitäten verfügen:

1. Ein *reset*-Eingang, der den Zähler auf den Wert 0 (oder einem anderen ausgezeichneten Wert) zurücksetzt. Er wird benötigt, um z. B. bei einem Systemstart den Befehlszähler mit null initialisieren zu können, damit die Programmabarbeitung an der richtigen Stelle beginnt.
2. Ein *data*-Eingang, über den bei Sprungbefehlen beliebige Werte in den Befehlszähler geladen werden können.
3. Ein *load enable*-Eingang, über den der Ladevorgang des Registers über den zusätzlichen Dateneingang gesteuert wird. Bei der Ausführung von Sprungbefehlen sorgt dieses Signal dafür, dass der neue Wert des Befehlszählers über den Dateneingang geladen wird. Bei allen anderen Befehlen wird der Dateneingang blockiert und der neue Wert ergibt sich durch Inkrementieren des Zählers.

In Abbildung 3 ist die Beschreibung des Befehlszählers in SystemC dargestellt. In den Zeilen 5 bis 7 erfolgt die Deklaration der soeben beschriebenen Eingänge. Die Programmspeicher-Adresse wird in der Variablen `pc` gespeichert (Zeile 11). Dann folgt die Deklaration von zwei Prozessen. Im Konstruktor `SC_CTOR` des SystemC Moduls (Zeile 16-21) werden die Prozesse beim Simulationskernel von SystemC registriert und die entsprechenden Sensitivitätslisten deklariert. Der kombinatorische Prozess `pcout_update()` (Zeile 24-26) schreibt bei einer Wertänderung von `pc` den neuen Wert an den Ausgang `pcout` und ist folglich auf `pc` sensitiv. Der sequentielle Prozess `next_state()` ist für die Berechnung des neuen Wertes des `pc` zuständig. Deshalb ist dieser Prozess auf das Taktsignal `clock` sensitiv. Die Modellierung des eigentlichen Verhaltens des Befehlszählers ist in den Zeilen 28 bis 36 zu sehen. Die drei if-Anweisungen folgen dabei der Beschreibung von oben.

4. Simulation

4.1. Übersetzung des `Zykluno` Assemblers in Maschinenbefehle

Dieser Abschnitt beschreibt das Werkzeug *zykluno_asm*, welches ein Assemblerprogramm für `Zykluno` in die entsprechenden Maschinenbefehle (Bitfolgen der Länge 16) übersetzt. Für den Assembler von `Zykluno` wurde dazu eine Grammatik entwickelt, die als Basis für den Parsergenerator PCCTS (Purdue Compiler Construction Tool Set) [6] dient. Mit Hilfe von PCCTS wird dann ein Parser erzeugt. Aus einem `Zykluno`-Assemblerprogramm entsteht durch das Parsen ein *Abstract Syntax Tree*, bei dessen Traversierung eine semantisch äquivalente Symboltabelle generiert wird. Die Symboltabelle ist sprachunabhängig und repräsentiert sowohl Daten als auch Kontrollflussstruktur des Programms. Die Analysephase, in der die Symboltabelle definiert wird, führt nicht nur semantische Fehlerprüfungen durch. Vielmehr steht in dieser Phase das frühzeitige Erkennen von *forward-declarations* (wie z.B. bei labels, bzw. Sprungzielen) im Vordergrund. Diese müssen unter Umständen schon vor ihrer Deklaration verwendbar und damit bekannt und adressierbar sein. In einer sich anschließenden Phase wird aus der Symboltabelle ein semantisch äquivalenter Maschinencode erzeugt. Zu Simulationszwecken kann der besagte Maschinencode, mittels einer Datei, in den Programmspeicher der ausführbaren `Zykluno`-Spezifikation geladen werden. Auf die selbe Weise kann auch der Datenspeicher initialisiert werden.

Für das einzeilige Assemblerprogramm „`LDD` $R[7]$, $R[0]$ “ zeigt Abbildung 4 den zugehörigen AST. Der AST spiegelt den Aufbau des verwendeten Befehls wider.

4.2. Beispiel

Ein Assembler-Programm zur Berechnung der Fibonacci-Folge $f(n) = f(n-1) + f(n-2)$ mit $f(0) = f(1) = 1$ ist in Abbildung 5 dargestellt. Das hier gegebene Programm berechnet die $n+2$-te Fibonacci-Zahl, wobei n zu Beginn aus dem Datenspeicher von Adresse 0 geladen wird. Der Ablauf wird im Folgenden kurz erläutert:

In Zeile 0 wird das Register $R[7]$ als Zählregister mit dem Wert von n initialisiert. Die Register $R[2]$ und $R[3]$ werden mit den Werten von $f(0)$ bzw. $f(1)$ initialisiert. Das aktuelle Zwischenergebnis ist in Register $R[4]$ zu finden. Die Register $R[0]$ und $R[1]$ sind Spezialregister; $R[0]$ ($R[1]$) enthält stets die Konstante 0 (1).

In der Schleife, die sich über die Zeilen 5 bis 9 erstreckt, findet die eigentliche Berechnung statt: die Summe der Register $R[2]$ und $R[3]$ wird berechnet und in $R[4]$ gespeichert. Anschließend wird

```
1  #include "defs.h"
2
3  SC_MODULE( prog_count ){
4      sc_in_clk               clock; // clock signal
5      sc_in<bool>             reset; // reset signal
6      sc_in<bool>             le;    // load enable
7      sc_in<sc_uint<11> >     din;   // data in
8
9      sc_out<sc_uint<11> >    pcout; // address output
10
11     sc_signal<sc_uint<11> > pc;    // program counter
12
13     void pcout_update();
14     void next_state();
15
16     SC_CTOR( prog_count ){
17         SC_METHOD(pcout_update);
18         sensitive << pc;
19         SC_METHOD(next_state);
20         sensitive << clock.pos();
21     }
22 };
23
24 void prog_count::pcout_update(){
25     pcout = pc.read();
26 }
27
28 void prog_count::next_state(){
29     if (reset.read()){
30         pc = 0;                 // reset to address 0
31     } else if (le.read()){
32         pc = din;               // load address
33     } else {
34         pc = pc.read() + 1;     // increase counter
35     }
36 }
```

Abbildung 3. Beschreibung des Befehlszählers in SystemC

der Wert von $R[3]$ in $R[2]$ und derjenige aus $R[4]$ in $R[3]$ kopiert. Die OR-Befehle in den Zeilen 6 und 7 ersetzen einen im Befehlssatz der Zykluno fehlenden Datentransportbefehl. Eigentlich wird eine bitweise Oder-Verknüpfung mit der Konstanten 0 durchgeführt. Nach Dekrementieren des Zählregisters $R[7]$ wird in Zeile 9 die Abbruchbedingung überprüft: hat der Zähler den Wert 0 erreicht, ist das Programm abgeschlossen, sonst findet ein weiterer Schleifendurchlauf ab Adresse 5 statt.

In Abbildung 6 ist das Ergebnis eines Simulationslaufes des Fibonacci-Programms für $n = 4$ zu

```
1   INSTRUCTION
2   {
3       LDD
4       {
5           REGISTER
6           {
7               ID == "R"
8
9               DECIMALINT == "7"
10          }
11
12          REGISTER
13          {
14              ID == "R"
15
16              OCTALINT == "0"
17          }
18      }
19  }
```

Abbildung 4. AST

Adr.	Maschinenbefehl	Assemblerbefehl		Kommentar
0:	0x5f00	LDD	$R[7]$, $R[0]$;	$R[7]$:= *DSp[*$R[0]$*]*
1:	0x4201	LDL	$R[2]$, #01;	
2:	0x4a00	LDH	$R[2]$, #00;	$R[2] := 1$
3:	0x4301	LDL	$R[3]$, #01;	
4:	0x4b00	LDH	$R[3]$, #00;	$R[3] := 1$
5:	0x3c1a	ADD	$R[4]$, $R[3]$, $R[2]$;	
6:	0x1218	OR	$R[2]$, $R[3]$, $R[0]$;	$R[2] := R[3]$
7:	0x1320	OR	$R[3]$, $R[4]$, $R[0]$;	$R[3] := R[4]$
8:	0x3739	SUB	$R[7]$, $R[7]$, $R[1]$;	
9:	0x6005	JNZ	#0005;	

Abbildung 5. Assembler-Programm zur Berechnung der Fibonacci-Folge

sehen. Die vier Schleifendurchläufe sind durch senkrechte Linien voneinander abgesetzt. Das Ergebnis der Berechnung ist an dem Signal `reg4` abzulesen. In Takt 25 ist schließlich an dem Signal `status->Z` zu erkennen, dass die Abbruchbedingung erfüllt ist.

5. Zusammenfassung

Die Arbeit gibt einen Eindruck, wie der Modellprozessor `Zykluno` mit SystemC von uns modelliert worden ist, und mit welchen Tools die Simulation von `Zykluno`-Assembler erfolgt. Für die Simulation von `Zykluno` wurde das Werkzeug *zykluno_asm* entwickelt, welches ein Assemblerprogramm in

Maschinenbefehle übersetzt. Die Simulation in Abschnitt 4 zeigte die leicht nachvollziehbare und von kommerziellen Tools unabhängige Exploration der Systemspezifikation, am Beispiel der Berechnung einer Fibonacci-Folge. Basierend auf dieser Arbeit ist für Studierende ein vertieftes Verständnis für den Aufbau einer Maschinensprache und deren Interpretation durch Hardware möglich.

Literatur

[1] B. Becker, R. Drechsler, and P. Molitor. *Technische Informatik — Eine Einführung*. Pearson Education Deutschland, 2005.

[2] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[3] J. Larus. *SPIM — A MIPS32 Simulator*. (http://www.cs.wisc.edu/ larus/spim.html, February 2005).

[4] R. G. (moderator). IEEE design and test roundtable on C++-based design. *IEEE Design & Test of Comp.*, pages 115–123, 2001. May-June.

[5] W. Müller, W. Rosenstiel, and J. Ruf, editors. *SystemC Methodologies and Applications*. Kluwer Academic Publishers, 2003.

[6] T. Parr. *Language Translation using PCCTS and C++: A Reference Guide*. Automata Publishing Co., 1997.

[7] D. Patterson and J. Hennessy. *Computer Organization and Design. The Hardware/Software Interface*. Morgan Kaufman Publishers - CA, 1994.

[8] Synopsys Inc., CoWare Inc., and Frontier Design Inc., http://www.systemc.org. *Functional Specification for SystemC 2.0*.

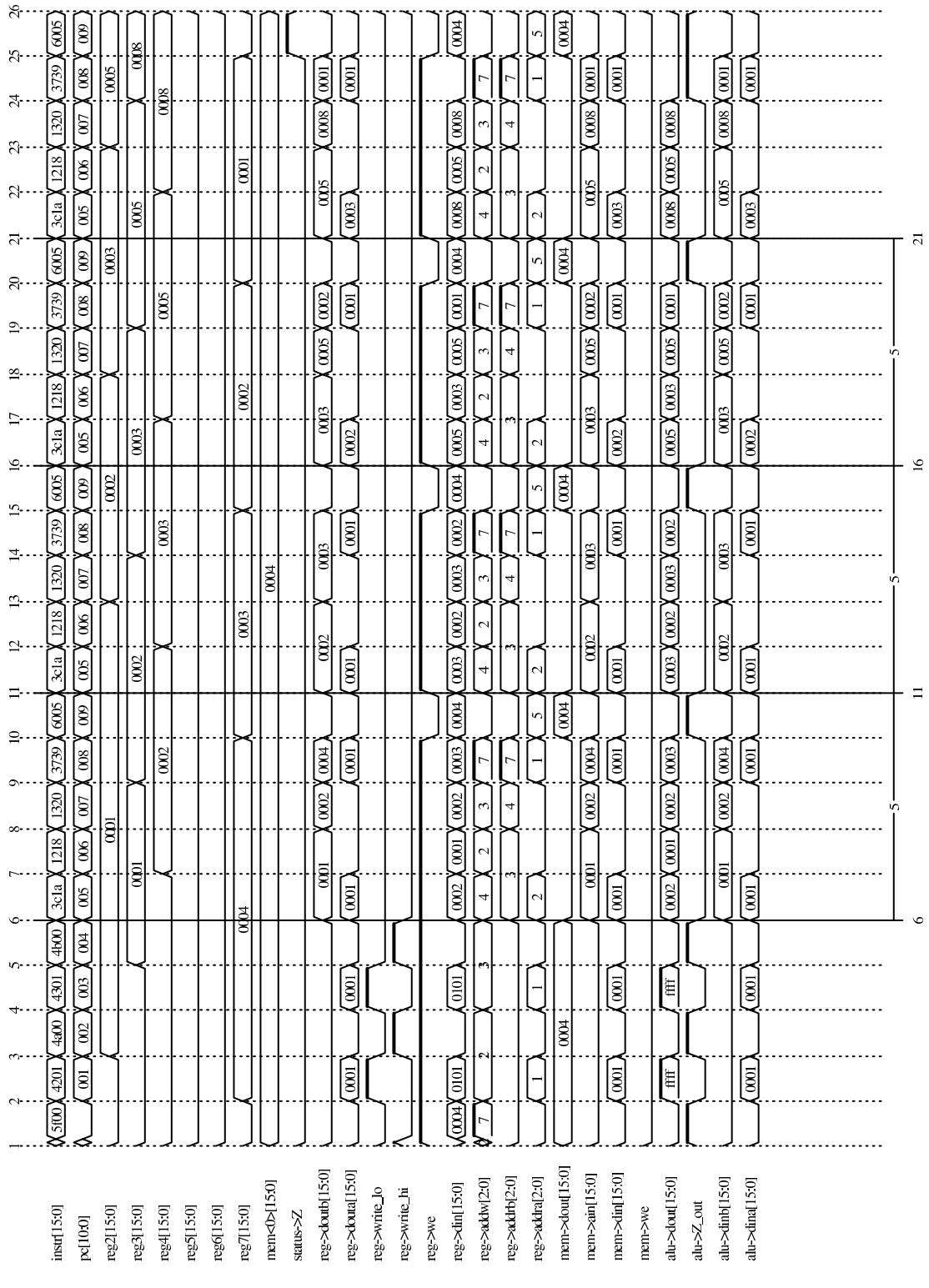

Abbildung 6. Simulationstrace