Diss. ETH No 7195

Code Generation and the Lilith Architecture

Dissertation

submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Technical Sciences

presented by

Christian Jacobi

Dipl. Math. of the Swiss Federal Institute of Technology
born August 10, 1951
Citizen of Zürich (Canton of Zürich)

Accepted on the recommendation of
Prof. Dr. N. Wirth
Dr. P. Schulthess

1982

Abstract

The Lilith computer is particularly suited to execute Modula-2 programs. The instruction set is chosen to reflect the needs of the compiler. Compilations, therefore, are easy and produce dense and fast code. There are special purpose high-level instructions as well as quite primitive operations.

The Lilith computer uses a stack for the evaluation of expressions. These evaluations are done with the use of a small hardware *expression* stack. The hardware expression stack permits arithmetic operations without memory access. It allows to combine the simplicity of a stack computer with the efficiency of a computer with general registers. The compiler guarantees that the expression stack never overflows. Lilith is also a stack computer in another sense; it allocates data segments of procedures on a stack located in main memory.

The memory is subdivided into several areas. Local data, global data, and code are accessed relative to specific registers. These registers are updated on execution of a call or return instruction. Data of external modules are accessed indirectly through pointers in the module table. Relative addressing allows usage of short offsets. Over 95% of all *load* instructions need an offset of less than 12. For these instructions the offset is directly encoded in the instruction byte.

Some language constructs suggest the provision of corresponding machine instructions. The CASE and the FOR statements suggest corresponding *case* and *for* instructions. Most instructions do not correspond to Modula structures in such a simple manner. However, the influence of Modula in defining the overall behaviour of otherwise general instructions is pervasive.

In structured programming languages short-distance jumps occur much more frequently than long-distance jumps. The compiler performs a jump optimization. For most jumps a short-address instruction is generated.

Defining our own instruction set allowed simple code generation. In spite of this, programming the code generation part of the compiler required more work than usual, since a lot of thought went into defining the instructions. As a result, simple code generation and efficient object code was achieved. The code of the Lilith computer (M-code) is more than twice as dense as the code of the well-known PDP-11.

Zusammenfassung

Mit dem Projekt, einen eigenen Rechner zu entwerfen, haben wir drei Hauptziele verfolgt. Wir wollten forschen, uns selber weiterbilden und als Resultat auch noch ein gutes Werkzeug zur Softwareherstellung erhalten. Wir sind konsequent nach einem Top-Down Verfahren vorgegangen. Zuerst wurde die Programmiersprache Modula-2 entworfen. Dies ist die einzige Programmiersprache, welche wir auf dem Lilith Rechner verwenden. Die Architektur des Rechners ist derart festgelegt, dass der Compiler einfach wird, aber trotzdem sehr effizienten Code erzeugt. Schliesslich wurde die Hardware darauf ausgelegt, dass diese Architektur effizient microcodiert werden konnte.

Stack Rechner erlauben einfache Code Generierung, gelten sonst aber nicht als effizient. Der Lilith Rechner hat einen kleinen Hardware *Expression*-Stack. Stackoperationen sind dadurch in einem einzigen Maschinenzyklus möglich und benötigen keine Speicherzugriffe. Der Expression-Stack erlaubt, die Einfachheit einer Stack Maschine mit der Effizienz einer Register Maschine zu kombinieren. Der Compiler stellt sicher, dass der Expression-Stack nicht überläuft.

Auf den Speicher wird auf strukturierte Weise zugegriffen. Lokale Daten, globale Daten eines Moduls und Instruktionen werden relativ zu entsprechenden Registern adressiert. Diese Register werden bei Prozeduraufrufen oder Rücksprüngen automatisch nachgeführt. Die Daten externer Module können erreicht werden, indem über die Modul-Tabelle zugegriffen wird. Die Adressierung relativ zu den entsprechenden Registern erlaubt kurze Adressen (*Offsets*). Auszählungen haben ergeben, dass über 95% aller *Load*-Instruktionen einen Offset kleiner als 12 benötigen. Für diese Instruktionen ist der Offset im Instruktionsbyte codiert.

Für gewisse Sprachelemente lohnt es sich, entsprechende Maschineninstruktionen zu definieren. Die *CASE* Anweisung wird in die entsprechende *Case*-Instruktion compiliert. Analoge Spezialinstruktion sind für die *FOR* Anweisung reserviert. Meistens sind aber die M-Code Instruktionen nicht so offensichtlich auf Modula-2 zugeschnitten. Die Eigenschaften der Sprache Modula-2 wiederspiegeln sich mehr in der Gesamtstruktur des Instruktionssatzes als in den einzelnen Instruktionen.

Kurze Sprünge treten wesentlich häufiger auf als Sprünge über weite Strecken. Die Länge eines Programmes kann wesentlich reduziert werden, wenn für kurze Sprünge auch kurze Instruktionen generiert werden. Der Modula Compiler führt eine Sprungoptimierung durch. Diese Sprungoptimierung ist wesentlich einfacher als entsprechende in der Literatur erwähnte Optimierungen. Sie basiert auf den Tatsachen, dass Sprunginstruktionen relativ addressiert werden, und dass für strukturierte Anweisungen sowohl das Sprungziel wie auch die Absprungstelle Bestandteil derselben Anweisung sind.

Die Definition eines geeigneten Instruktionssatzes hat zu einer relativ einfachen Codegenerierung geführt. Trotzdem steckt dadurch im Codegenerierungsteil des Compilers mehr Arbeit als üblich. Als Resultat entsteht eine einfache Codegenerierung, die dennoch äusserst effizienten Code erzeugt. Der Code für den Lilith Rechner (M-Code) braucht weniger als die Hälfte des Speicherplatzes des entsprechenden PDP-11 Codes.