１１．コンパイラ

１．本実験の目的

一般的なユーザーが計算機を使用する場合、通常はCやFortran等の高級言語を用いてプログラミングを行うことがふつうである。しかし、計算機そのもの、すなわち中央処理装置(CPU)が実際に解釈実行しているのは0/1のビットパターンで表される機械語である。すなわち、高級言語プログラムを計算機がそのまま実行することは不可能であり、何らかの処理を行って機械語に変換する必要がある。その方法には大きく分けてインタプリタとコンパイラの二種類がある。前者は高級言語プログラムを解釈しながら機械語サブルーチンを呼び出して実行を行う方法である。後者は高級言語プログラム全体を機械語プログラムに変換し実行する方法である。

本実験では、高級言語プログラムを機械語に変換するプログラムであるコンパイラについて、その中でもっとも基本となる式の機械語への変換アルゴリズムを学び、また、式のコンパイラによって生成された機械語のプログラムを実行することを通して、実際の計算機上で高級言語を実行するためにどのような処理が行われているかについての理解を深めることを目的としている。

２．計算機アーキテクチャと機械語の基礎

2.1　計算機の基本的な構成と動作

現在一般に使用されているプログラム内蔵方式の計算機の内部構成を図2.1に示す。中央処理装置(CPU)は記憶装置に格納された機械語命令に従ってプログラムの実行の制御および演算を行う。記憶装置（メモリ）には機械語命令のほか、プログラムが処理するデータも格納される。記憶装置にはデータの記憶単位ごとに番地(アドレス）が付けられており、データやプログラムを格納する場所を指定するために用いられる。入出力装置は外部との入出力を行う。

次に、中央処理装置の内部構成を図2.2に示す。中央処理装置の内部は大別して制御装置、演算装置、一時記憶装置などから構成される。

制御装置は、次に取り出す命令が格納されているアドレスを指すプログラムカウンタ(PC)、取り出された命令を格納する命令レジスタ、命令レジスタに格納された命令を解読するためのデコーダ、デコーダの出力に従って各種の制御信号を発生する制御回路等から構成されている。

演算装置は制御装置から発生される信号に従って各種の数値演算や論理演算を行う装置で、ALU(Arithmetic Logic Unit)と呼ばれることもある。

一時記憶装置は演算装置が処理するデータを一時的に保持しておくための場所で、アキュムレータまたはレジスタと呼ばれる。これらに記憶できるデータの量、すなわちレジスタの数は計算機の構成法によって大きく異なり、少ないものでは数個しかないものもあれば、百数十個のレジスタを持つものもある。

このような計算機の論理的な内部構成の方式のことを、一般に計算機アーキテクチャと呼ぶ。計算機の処理する対象に応じて、現在では多種多様な計算機アーキテクチャが用いられている。

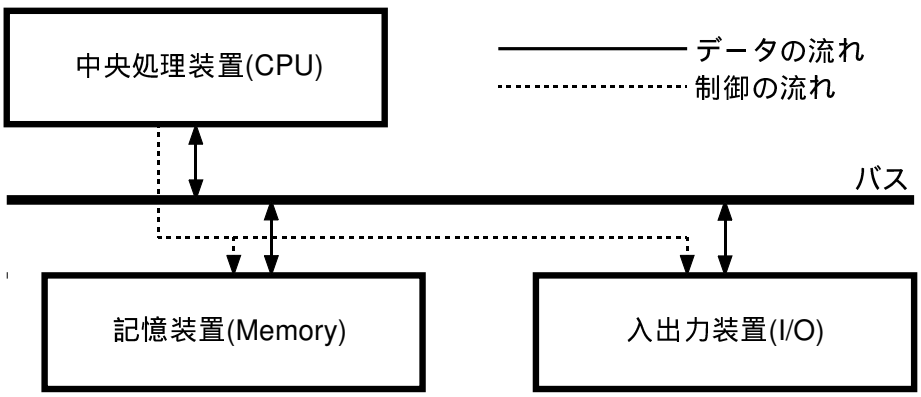


図2.1　計算機の構成

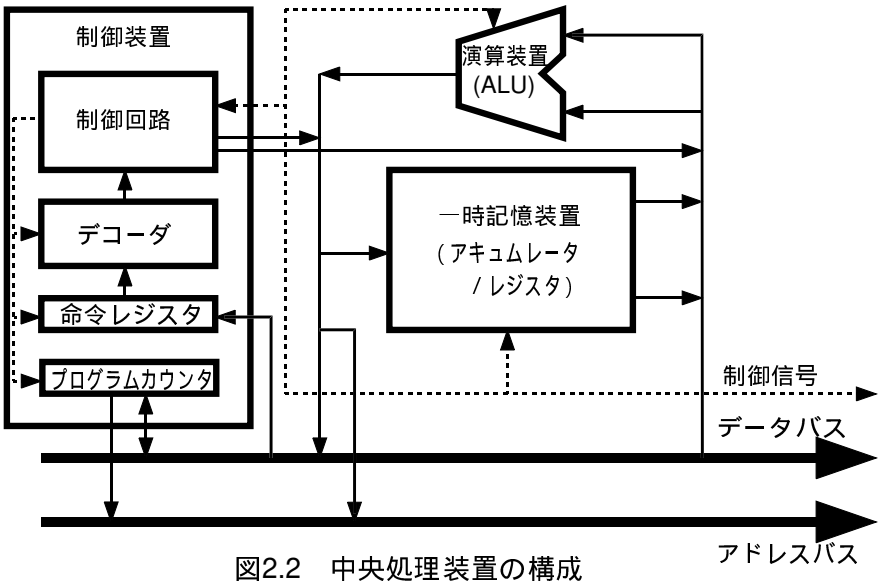


図2.2　中央処理装置の構成

2.2　機械語とアセンブリ言語

計算機の機械語命令を分類すると、いくつかの種類に大きく分けることができる。

一つはデータを移動する命令である。これにはメモリからレジスタにデータを読み込むロード命令、逆にレジスタからメモリに書き込むストア命令、複数のレジスタを持つ場合にレジスタ間でデータを転送する命令などが含まれる。

二つ目は演算命令である。これには加減乗除や論理演算、シフト†などの命令が含まれる。演算対象となるデータの格納される場所の指定は、レジスタ間のみの演算だけが可能なものや、メモリに格納されているデータも直接演算可能なものなど、アーキテクチャによって異なる。

三つ目は分岐命令である。分岐命令にはあるアドレスに無条件に分岐する無条件分岐命令、演算等の結果によって分岐を実行するかどうかを決める条件分岐命令、あるいはサブルーチンを呼んだり、サブルーチンから戻ったりするための命令などが含まれる。

このほかにも、なにも有効な処理を行わないノーオペレーション命令や、計算機の実行を停止する命令などもある。

計算機が実行する実際の機械語命令は、0/1のビットパターンの組み合わせより構成されているため、人間が直接機械語でプログラムを作成することは非常に困難である。そこでもう少し簡単に機械語プログラムを作成するための手段として、アセンブリ言語が考案された。

アセンブリ言語とは、機械語命令にそれぞれ対応する名前を付け、またアドレスを指定する場合もその番地に名前（ラベル）を付けられるようにするなどして、機械語プログラムを作成しやすくしたものである。機械語命令に対応する名前としては、その機械語の操作に対応した英単語の省略形が用いられることが多い。たとえば加算(Addition)ならADD,ロード(Load)ならLDなどが用いられる。これらはニーモニック(mnemonic)と呼ばれる。アセンブリ言語で記述された機械語プログラムは、アセンブラと呼ばれる変換プログラムにより、実際の機械語プログラムに変換され、実行される。本実験におけるコンパイラでは、直接機械語を出力するのではなく、アセンブリ言語のプログラムを出力する。

2.3　実験用仮想計算機アーキテクチャ

2.3.1　実験用仮想計算機について

実際に使用されている計算機を、本実験において使用するコンパイラのための対象アーキテクチャとして使用するのは、必要以上に複雑であったりして問題を困難にするので、ここでは式のコンパイルに必要なもののみに限定した仮想計算機アーキテクチャを用いることにする。

---

† データのビットを左右に移動すること。

本実験で使用する仮想計算機には、アキュムレータアーキテクチャ、スタックアーキテクチャ、CISCアーキテクチャ、RISCアーキテクチャの４種類がある。まずはじめに、各アーキテクチャ間で共通な仕様を述べ、その後各アーキテクチャとその機械語について説明する。

2.3.2　共通仕様

ここでは４種のアーキテクチャに共通した仕様について述べる。

本実験で使用する仮想計算機は、基本的に16ビットからなる１語（ワード）を単位として動作する16ビットマシンである。記憶装置（メモリ）は１ワードごとにアドレスが付けられており、0 ~ 65535までの64Kワードの記憶容量を持つ。

演算装置も16ビットの整数演算が可能で、２の補数形式で数値が表現されている。すなわち、表現できる整数の範囲は-32768 ~ 32767までの範囲となる。なお、浮動小数点演算は行わない。

また、実行できる演算は加算、減算、乗算、除算、剰余、論理和、論理積、排他的論理和、論理否定、符号反転の10通りである。４種の論理演算はビットごとの論理演算を行う。また、演算において桁あふれが生じても無視されるが、０による除算が生じた場合には実行時エラーにより実行が停止する。

ロード・ストア命令およびメモリに対する演算命令において指定できるメモリアドレスは直接番地指定のみ可能である。

なお本仮想計算機では、対象を式のコンパイルに限定してあるため、分岐命令はいっさい含まれていないことに注意する必要がある。

最後に、すべての仮想計算機に共通する機械語命令を２つ説明する。一つはなにも実行しない（ノーオペレーション）命令である。この命令を実行すると、意味のある操作は行われず、次の命令に実行が移る。アセンブリ言語ではNOPと記述する。もう一つは計算機を停止させる命令である。この命令を実行すると計算機の実行が停止される。アセンブリ言語ではHLTと記述する。

2.3.3　アキュムレータアーキテクチャ

アキュムレータアーキテクチャの構成図を図2.3に示す。

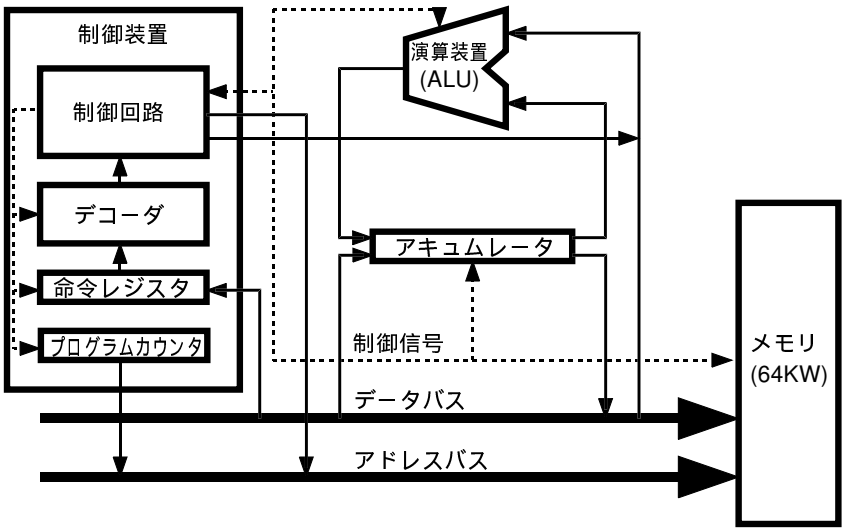


図2.3　アキュムレータアーキテクチャの構成

図中に示されているとおり、演算を行う際の片方の被演算数と演算結果の書き込み先は常にアキュムレータに固定されている。もう一方の被演算数はメモリまたは即値(Immediate)†のどちらかとなる。また、データ転送命令にはアキュムレータ・メモリ間の転送命令と、即値をアキュムレータに格納する命令の二種類がある。

以下に、各機械語命令のアセンブリ言語における記法とその意味を示す。それぞれ命令の種類、アセンブリ言語の記法、意味の順になっている。なお、意味の欄で、mはメモリアドレス、M[m]はメモリのm番地、iは即値、Accはアキュムレータを指す。

---

† 機械語の命令語中に含まれている数値のこと。

・データ転送命令

ロード命令：　LD m　M[m]->Acc　ストア命令：　ST m　Acc->M[m]

即値ロード命令：　LDI i　i->Acc

これらの命令は、アキュムレータとメモリ間のデータの転送、および即値をアキュムレータに格納する命令である。

・演算命令

加算：　ADD m　Acc+M[m]->Acc　減算：　SUB m　Acc-M[m]->Acc

乗算：　MUL m　Acc*M[m]->Acc　除算：　DIV m　Acc/M[m]->Acc

剰余：　MOD m　Acc mod M[m]->Acc

これらはアキュムレータとメモリの内容の間で整数演算を行い、結果をアキュムレータに格納する。

論理和：　AND m　Acc and M[m]->Acc　論理積：　OR m　Acc or M[m]->Acc

排他的論理和：XOR m　Acc xor M[m]->Acc

これらはアキュムレータとメモリの内容の間でビットごとの論理演算を行い、結果をアキュムレータに格納する。

符号反転：　NEG　-(Acc)->Acc　論理否定：　NOT　not(Acc)->Acc

これらはアキュムレータの内容にそれぞれの演算を行い、結果をアキュムレータに格納する。

・即値演算命令

即値加算：　ADDI i　Acc+i->Acc　即値減算：　SUBI i　Acc-i->Acc

即値乗算：　MULI i　Acc*i->Acc　即値除算：　DIVI i　Acc/i->Acc

即値剰余：　MODI i　Acc mod i->Acc

これらはアキュムレータと即値との間で整数演算を行い、結果をアキュムレータに格納する。

即値論理和：　ANDI i　Acc and i->Acc　即値論理積：　ORI i　Acc or i->Acc

即値排他的論理和：XORI i　Acc xor i->Acc

これらはアキュムレータと即値との間でビットごとの論理演算を行い、結果をアキュムレータに格納する。

2.3.4　スタックアーキテクチャ

スタックアーキテクチャの構成図を図2.4に示す。

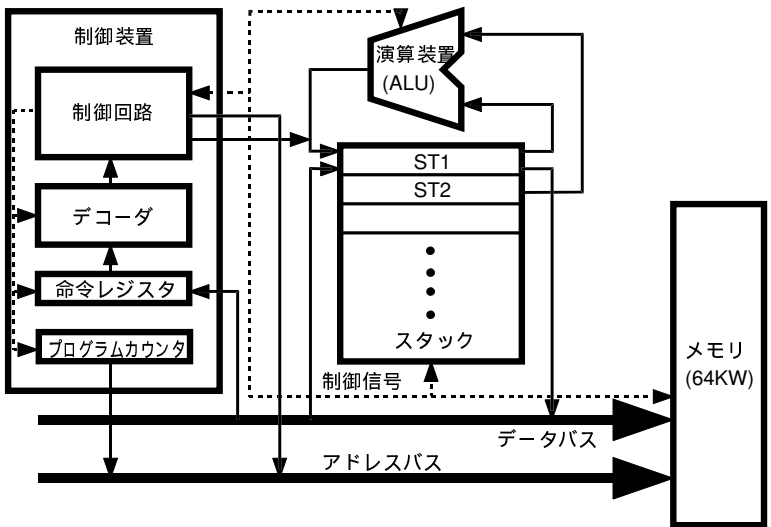


図2.4　スタックアーキテクチャの構成

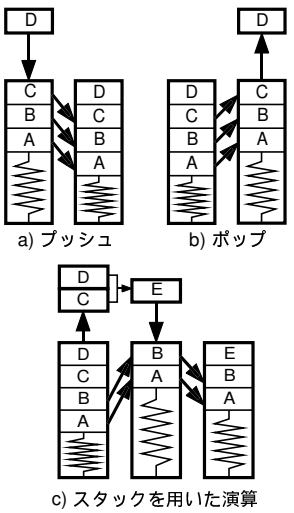


図2.5　スタックの概念図

本アーキテクチャの特徴は一時記憶装置としてスタックを使用している点である。スタックとはFILO†形式のデータ構造で、スタックに入れたデータはその逆順でのみ取り出せるようになっている。その概念図を図2.5に示す。なお、スタックにデータを格納することをプッシュ(Push),データをスタックから取り出すことをポッ

† First In, Last Out の略。先入れ後出し方式。

プ(Pop)と呼ぶ。

本アーキテクチャでは、演算はスタックの最上位からポップされた 2 つの値の間で行われ、その結果は再びスタックにプッシュされる。

また、データ転送命令にはスタックにメモリの内容をプッシュする命令、スタックからポップした値をメモリに格納する命令、即値をスタックにプッシュする命令の三種類がある。

以下に、各機械語命令のアセンブリ言語における記法とその意味を示す。それぞれ命令の種類、アセンブリ言語の記法、意味の順になっている。なお、意味の欄で、mはメモリアドレス、M[m]はメモリのm番地、iは即値、ST1はスタックの最上位、ST2はスタックの上から二番目を示す。

・データ転送命令

プッシュ命令： PUSH m M[m]->ST1 ポップ命令： POP m ST1->M[m]

即値プッシュ命令：PUSHI i i->ST1

これらの命令は、スタックとメモリ間のデータの転送、および即値をスタックにプッシュする命令である。

・演算命令

加算：	ADD	ST2+ST1->ST1	減算：	SUB	ST2-ST1->ST1
乗算：	MUL	ST2*ST1->ST1	除算：	DIV	ST2/ST1->ST1
剰余：	MOD	ST2 mod ST1->ST1			

これらはスタックから 2 つの値をポップし、その間で整数演算を行い、結果を再びスタックにプッシュする。

論理和：	AND	ST2 and ST1->ST1	論理積：	OR	ST2 or ST1->ST1
排他的論理和：	XOR	ST2 xor ST1->ST1			

これらはスタックから 2 つの値をポップし、その間でビットごとの論理演算を行い、結果を再びスタックにプッシュする。

符号反転：	NEG	-(ST1)->ST1	論理否定：	NOT	not(ST1)->ST1

これらはスタックから 1 つの値をポップし、それに対して演算を行い、結果を再びスタックにプッシュする。

2.3.5 CISCアーキテクチャ

CISC†アーキテクチャの構成図を図2.6に示す。

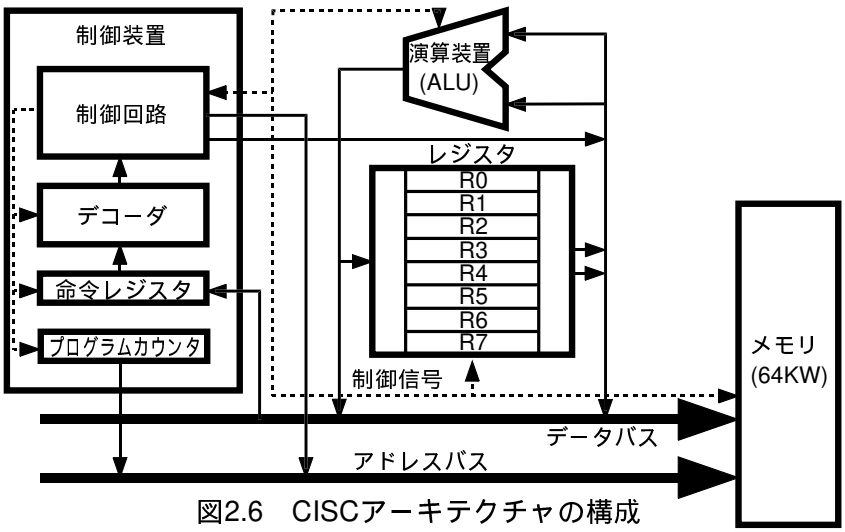


図2.6 CISCアーキテクチャの構成

このアーキテクチャでは、演算を行う際の被演算数としてはメモリの内容、レジスタ、即値の 3 つを自由に選択できるようになっている。演算結果の書き込み先はメモリまたはレジスタのどちらかを選択できる。

また、データ転送命令においても、転送元はメモリ、レジスタ、即値の 3 つから、転送先はメモリとレジスタの 2 つから自由に選択可能である。

なお、使用可能なレジスタはR0 ~ R7までの 8 つである。

† Complex Instruction Set Computer,複雑な命令セットを持つ計算機。

以下に、各機械語命令のアセンブリ言語における記法とその意味を示す。それぞれ命令の種類、アセンブリ言語の記法、意味の順になっている。なお、意味の欄で、mはメモリアドレス、M[m]はメモリのm番地、iは即値、Rxはレジスタ番号xのレジスタを指す。

・データ転送命令

ロード命令：　LD src,Rx　src->Rx,src=m/i

ストア命令：　ST src,m　src->M[m],src=Rx/i

データ移動命令：　MOV m1,m2　M[m1]->M[m2]　MOV Rx1,Rx2　Rx1->Rx2

ロード命令は、メモリの内容もしくは即値をレジスタに格納する命令である。ストア命令は、レジスタの内容もしくは即値をメモリに格納する命令である。データ移動命令は、メモリ間またはレジスタ間のデータ移動を実行する命令である。

・演算命令

加算：　ADD src1,src2,dst　src1+src2->dst　減算：　SUB src1,src2,dst　src1-src2->dst

乗算：　MUL src1,src2,dst　src1*src2->dst　除算：　DIV src1,src2,dst　src1/src2->dst

剰余：　MOD src1,src2,dst　src1 mod src2->dst

これらはsrc1とsrc2の間で整数演算を行い、結果をdstに格納する。src1,src2はそれぞれメモリ、レジスタ、即値のどれかであり、dstはメモリかレジスタのどちらかである。

論理和：　AND src1,src2,dst　src1&src2->dst　論理積：　OR src1,src2,dst　src1|src2->dst

排他的論理和：XOR src1,src2,dst　src1 xor src2->dst

これらはsrc1とsrc2の間でビットごとの論理演算を行い、結果をdstに格納する。src1,src2はそれぞれメモリ、レジスタ、即値のどれかであり、dstはメモリかレジスタのどちらかである。

符号反転：　NEG src,dst　-(src)->dst　論理否定：　NOT src,dst　not(src)->dst

これらはsrcの内容にそれぞれの演算を行い、結果をdstに格納する。srcはメモリ、レジスタ、即値のどれかであり、dstはメモリかレジスタのどちらかである。

なおこのCISCアーキテクチャについてのみ、アセンブリ言語において即値を表す場合には、たとえば#100のように先頭に#を付ける。また、レジスタを指定するにはRx(x=0 ~ 7)と記述する。それ以外はすべてメモリアドレスとみなされる。

2.3.6　RISCアーキテクチャ

RISC†アーキテクチャの構成図を図2.7に示す。

このアーキテクチャでは、演算命令の被演算数として、レジスタ間の演算かもしくはレジスタの内容と即値

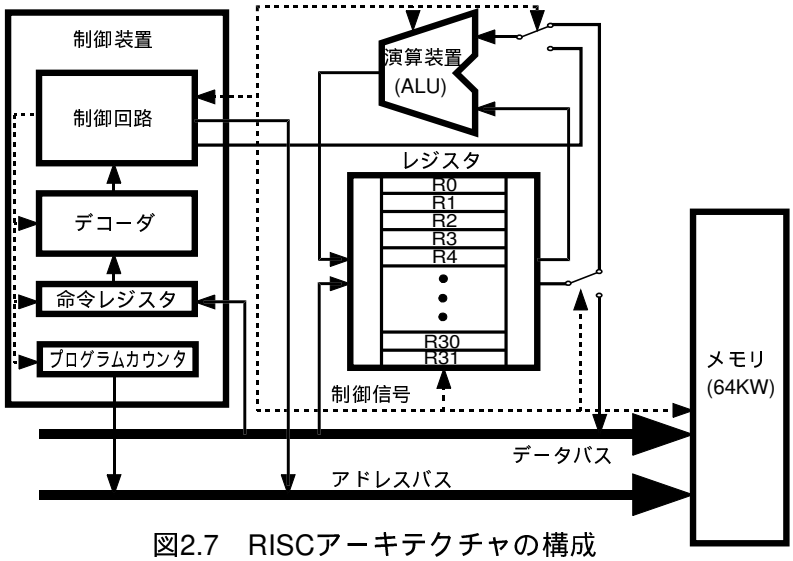


図2.7　RISCアーキテクチャの構成

† Reduced Instruction Set Computer,単純化した命令セットを持つ計算機。

の間の演算かのどちらかを選択できるようになっている。ただし演算結果の書き込み先はレジスタに固定されている。

また、データ転送命令はメモリ・レジスタ間のロード・ストア命令と即値をレジスタに格納する即値ロード命令の３種類である。

なお、使用可能なレジスタはR0 ~ R31までの32個である。

以下に、各機械語命令のアセンブリ言語における記法とその意味を示す。それぞれ命令の種類、アセンブリ言語の記法、意味の順になっている。なお、意味の欄で、mはメモリアドレス、M[m]はメモリのm番地、iは即値、Rxはレジスタ番号x(x=0 ~ 31)のレジスタを指す。

・データ転送命令

ロード命令：　LD m,Rx　M[m]->Rx　ストア命令：　ST m,Rx　Rx->M[m]

即値ロード命令：　LDI i,Rx　i->Rx

これらの命令は、レジスタとメモリ間のデータの転送、および即値をレジスタに格納する命令である。

・演算命令

加算：　ADD Rx,Rx',Rx"　Rx+Rx'->Rx"　減算：　SUB Rx,Rx',Rx"　Rx-Rx'->Rx"

乗算：　MUL Rx,Rx',Rx"　Rx*Rx'->Rx"　除算：　DIV Rx,Rx',Rx"　Rx/Rx'->Rx"

剰余：　MOD Rx,Rx',Rx"　Rx mod Rx'->Rx"

これらはRxおよびRx'で指定されるレジスタの内容の間で整数演算を行い、結果をRx"で指定されるレジスタに格納する。

論理和：　AND Rx,Rx',Rx"　Rx&Rx'->Rx"　論理積：　OR Rx,Rx',Rx"　Rx|Rx'->Rx"

排他的論理和：XOR Rx,Rx',Rx"　Rx xor Rx'->Rx"

これらはRxおよびRx'で指定されるレジスタの内容の間でビットごとの論理演算を行い、結果をRx"で指定されるレジスタに格納する。

符号反転：　NEG Rx,Rx'　-(Rx)->Rx'　論理否定：　NOT Rx,Rx'　not(Rx)->Rx'

これらはRxで指定されるレジスタの内容にそれぞれの演算を行い、結果をRx'で指定されるレジスタに格納する。

・即値演算命令

即値加算：　ADD Rx,i,Rx'　Rx+i->Rx'　即値減算：　SUB Rx,i,Rx'　Rx-i->Rx'

即値乗算：　MUL Rx,i,Rx'　Rx*i->Rx'　即値除算：　DIV Rx,i,Rx'　Rx/i->Rx'

即値剰余：　MOD Rx,i,Rx'　Rx mod i->Rx'

これらはRxで指定されるレジスタの内容と即値iの間で整数演算を行い、結果をRx'で指定されるレジスタに格納する。

即値論理和：　AND Rx,i,Rx'　Rx and i->Rx'　即値論理積：　OR Rx,i,Rx'　Rx or i->Rx'

即値排他的論理和：　XOR Rx,i,Rx'　Rx xor i->Rx'

これらはRxで指定されるレジスタの内容と即値iの間でビットごとの論理演算を行い、結果をRx'で指定されるレジスタに格納する。

即値符号反転：　NEG i,Rx　-(i)->Rx　即値論理否定：　NOT i,Rx　not(i)->Rx

これらは即値iにそれぞれの演算を行い、結果をRxで指定されるレジスタに格納する。

2.4　アセンブリ言語の疑似命令

アセンブリ言語では、これまでに述べてきた各アーキテクチャにおける機械語命令のほかに、以下のような疑似命令を使用できる。

開始番地の指定：　ORG m

機械語プログラムの格納されるメモリ上の番地を指定できる。上記の例の場合、この疑似命令の直後に続く命令から、m番地以降に格納されることになる。通常はプログラムの先頭に置かれる。省略された場合、0番地が仮定される。

メモリ上の領域確保：　DS n

現在の番地からメモリ上にnワード分記憶領域が確保される。これは、たとえば式のコンパイルにおいて変数の記憶領域を確保する場合に用いる。

メモリ上に定数確保：　DW i[,i...]

メモリ上に1ワード分記憶領域が確保され、さらに整数値iがそこに書き込まれる。

ラベルに定数代入：　　ラベル名　EQU i

アセンブリ言語では、アドレスや各種の定数を表すのにラベルを用いるが、ラベルに整数定数を代入するために用いられるのがこの疑似命令である。

アセンブリ言語における整数定数としては、以下のようなものが使用できる。まず通常の10進数は当然使用可能である。16進数は123Hのように最後にHを付加して表す。16進数で最初の桁がA～Fになる場合は、ラベルと区別できるように0ABCHのように先頭に必ず0を付加する必要がある。2進数も使用可能で、010010Bのように最後にBを付けて表記する。

ラベル名としては、一般的な変数名同様に英文字で始まる英数字の並びが使用可能である。ただし機械語命令もしくは疑似命令で使用されている名前は使用できない。また、アンダーライン '_' も通常の英文字同様に使用できる。

整数式は、整数定数、ラベル、および整定数またはラベルを +, -, *, /, &, |, ^, ~ の演算子で結合したものである。演算子の意味はC言語と同様で、( )も使える。

現在アドレスにあるラベルを付けること、すなわちラベルへの現在のアドレスの代入は、行の先頭にラベル名を書き、その直後に :を付けることにより行える。

なお、アセンブリ言語では大文字小文字の区別はいっさい行われない。また、行の途中に ;を書くとそこから行末までがコメントとして扱われる。

最後に、アセンブリ言語のプログラム例を図2.8に示す。

```
; 式 A=B*C の RISC architectureでのコンパイル例
;
        org     0           ;プログラム開始番地は0

        ld      var_B,R1    ;変数Bの値をレジスタR1にロード
        ld      var_C,R2    ;変数Cの値をレジスタR2にロード
        mul     R1,R2,R3    ;R1*R2->R3
        st      var_A,R3    ;レジスタR3の値を変数Aにストア
        hlt                 ;停止
;
;       変数領域の確保
;
var_A:  ds      1       ;変数Aの領域を1word確保、ラベルを付ける
var_B:  ds      1       ;変数Bの領域を1word確保、ラベルを付ける
var_C:  ds      1       ;変数Cの領域を1word確保、ラベルを付ける
```

図2.8　機械語プログラムの例

３．　式のコンパイラ

3.1　コンパイラの構成

一般的なコンパイラの構成を、図3.1に示す。コンパイルする元となる高級言語プログラム(これをソースプログラムという) は、まず字句解析部によりある単位ごとに分割・分類される。たとえば、数値、記号、変数名などである。これらは次に、構文解析部に送られ、ソースプログラムの文法に従って解析された後にいったん中間的な表現に変換される。式の場合では、逆ポーランド記法や木構造などがある。最後に、コード生成部が構文解析部の出力結果に基づいて、実行可能な機械語プログラム (これをオブジェクトプログラムという) を出力する。計算機のアーキテクチャが異なれば機械語も異なってくるため、各アーキテクチャに応じてコー

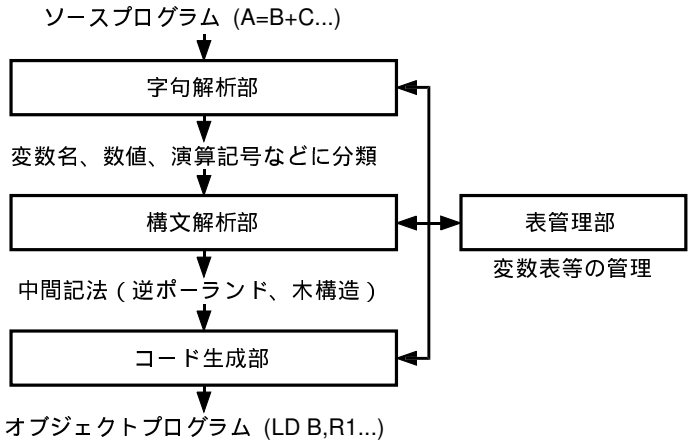


図3.1　コンパイラの構成

ド生成部は変更する必要がある。

これらのほかに、各部分から参照されるものとして、表管理部がある。表管理部では、変数、関数、ラベル等の表を作成し、字句解析部において新しい変数名等が出現した場合にはその名前や形などをそれぞれの表に登録する。ここで登録された情報は、コード生成部において必要に応じて参照され、機械語生成のために使用される。

3.2　ソースプログラム (式) の仕様

本実験で使用するコンパイラは、本節で定義する式のコンパイルを行いアセンブリ言語のオブジェクトプログラムを生成する。

文法の定義をする場合によく用いられる記述法に、バッカス記法(BNF†)と呼ばれるものがある。BNFとは文法を再帰的に記述する方法で、以下のように名前を定義していくことにより文法の定義を行う。

具体的には、「$a_1, a_2, \ldots, a_n$ のことを b という」ということの定義は、以下のように記述される。

< b > ::= a_1 | a_2 | ... | a_n

b は $a_1, a_2, \ldots, a_n$ に対して付けられた名前であり、このような名前は < > で囲むことにより表される。

まず、BNFにより < 符号なし整数 > 、 < 変数名 > の定義は以下のようになる。なお、変数名では大文字と小文字は区別せずすべて大文字として扱う。

< 英字 > ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

< 数字 > ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

< 変数名 > ::= < 英字 > | < 変数名 > < 英字 > | < 変数名 > < 数字 >

< 符号なし整数 > ::= < 数字 > | < 符号なし整数 > < 数字 >

この定義により、数字の並びは符号なしの整数値になり、英字で始まる英数字の並びは変数名になることが定義される。例として、123およびAB01を上記の定義に従って分析した例を図3.2に示す。

次に、これらを用いて式の定義を行う。ここでいう式は一般的な数学の式の中で、四則演算と括弧のみに制限したものである。また、単項演算子である + および - は、式の一番先頭もしくは (の直後にのみ使用できるものと限定する。

< 式 > ::= < 項 > | < 加減演算子 > < 項 > | < 式 > < 加減演算子 > < 項 >

< 項 > ::= < 1 次子 > | < 項 > < 乗除演算子 > < 1 次子 >

< 1 次子 > ::= < 符号なし整数 > | < 変数名 > | (< 式 >)

< 乗除演算子 > ::= * | /

< 加減演算子 > ::= + | -

さらに、この式の定義を用いて代入文およびプログラムの定義をすると以下のようになる。ただし、EOFはソースプログラムの最後、CRは改行を表すものとする。

---

† Backus Naur Form または Backus Normal Form の略。

＜代入文＞::=＜変数名＞=＜式＞CR

＜プログラム＞::=EOF|＜代入文＞＜プログラム＞

これらの定義により、プログラムは右辺の式の値を左辺の変数に代入する代入文の並びと定義される。代入文は必ず改行で終わる1行の中に記述する必要がある。なお、上記の定義では内容のない空のプログラムも認められる。

これらの定義を用いたあるプログラムの分析例を図3.3に示す。

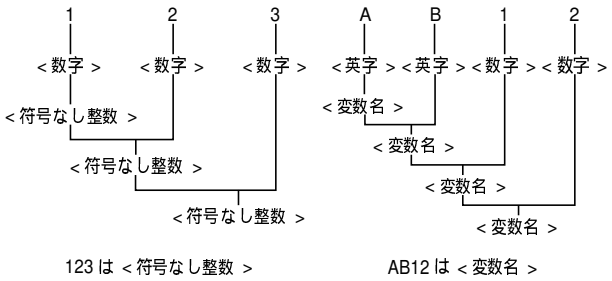


図3.2　BNFの例（その１）

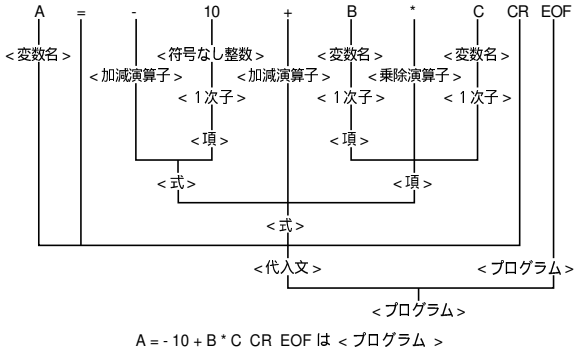


図3.3　BNFの例（その２）

3.3　字句解析部

コンパイラの字句解析部では、入力されたソースプログラムを変数名、整数定数、演算子などの記号といった単位に分割し、それらの属性を調べて次段の構文解析部に送る必要がある。

本実験の式のコンパイラでは、演算子等の記号はすべて1文字であるので、その1文字が出現した時点で記号であるという分類と、その記号の種類を返すことができる。ここで検出すべき記号としては、+, -, *, / の四則演算記号、(,) の左および右括弧、= の代入記号、そして改行およびソースファイルの最後を表すEOFがある。

読み込んだ1文字目がアルファベットであった場合、これは変数名の始まりであると考えられるので、読み込んだ文字がアルファベットか数字である限り変数名として読み込む。読み込んだ文字がアルファベットもしくは数字以外になった場合、変数名の終わりであるので、余分に読んだその文字を入力にいったん戻してから、変数であるという情報と読み込んだ変数名を返す。このとき、表管理部を呼び出して変数表への名前の登録も行う必要がある。

整数定数の場合もほぼ変数名と同様で、異なるのは数字で始まり、数字が続く限り整数定数として取り込むことだけである。整数定数の場合は、数字であるという情報と、数字の並びから変換した実際の数値を返す必要がある。

なお、区切りを表す空白もしくはタブは読み飛ばされる。

3.4　構文解析部

字句解析部によって分けられたそれぞれの単語は、構文解析部によって式の文法に従って解析される。式の形そのままではコード生成を行うことは困難なため、解析の結果によって式の構文はコード生成に用いられる中間的な表現に変換される。式の構文の表現法にはいくつかの種類があるが、本実験ではその中で代表的なものとして逆ポーランド記法と木構造の2種類を用いる。以下で、元の式からそれぞれの表現への変換のためのアルゴリズムについて説明する。

3.4.1　逆ポーランド記法

式の逆ポーランド記法について、まず次に示す式を例にして説明する。

A=B*C-(D+E)/(E-F)

この式において、演算は基本的に左から右に実行され、また演算子の優先順位は*, / が一番高く、次に+, - で、一番低いのが= である。当然括弧がある場合には括弧の中が優先される。この優先順位に従って、上の式で行われるそれぞれの演算を、それぞれの演算子を名前として持つ二つの引数を持つ関数のように表すと以下のよ

うになる。

```
=(A,-(*(B,C),/(+(D,E),-(E,F))))
```

ここで、関数名に相当する演算子を括弧の後方に移動すると、次のようになる。

```
(A,((B,C)*,((D,E)+,(E,F)-)/)-)=
```

この式から( ) , をすべて取り除くと、

```
ABC*DE+EF-/-=
```

となる。これが、最初の式を逆ポーランド記法で表した式である。逆ポーランド記法で表すことにより、括弧や優先順位を考えることなく機械的に演算を行うことが可能になり、従ってコード生成もより容易に行うことができる†。

通常の式の記法から逆ポーランド記法への変換は、以下のようなアルゴリズムにより行うことができる。

まずスタックを用意し、元の式の左から右へと調べていくことにする。

(1) 変数もしくは定数の場合、それを出力する。

(2) 演算子αの場合、

(i) スタックが空、または、スタックの最上位が (の時、αをスタックに入れる

(ii) スタックの最上位が演算子βの時、αとβの優先順位を比較して

(イ) αの方が低いか等しければ、βをスタックから取り出して出力し、(2)へもどる。

(ロ) αの方が高ければ、αをスタックに入れる。

(3) (なら、それをスタックに入れる。

(4)) または CR なら、

(i) スタックの最上位が (のとき、(をスタックから取り出す。

(ii) スタックの最上位が演算子βの時、βをスタックから取り出して出力し、(4)に戻る。ただし、βが = のとき終わり。

このアルゴリズムにより、式**A=B*C-(D+E)/(E-F)**を逆ポーランド記法に変換した例を表3.1に示す。

3.4.2 木構造

ここでは、式を木構造を用いて表現する方法について、逆ポーランド記法のときと同じ次に示す式を例にして説明する。

```
A=B*C-(D+E)/(E-F)
```

逆ポーランド記法への変換のときと同様に、上の式で行われるそれぞれの演算をそれぞれの演算子を名前として持つ二つの引数を持つ関数のように表すと以下のようになる。

```
=(A,-(*(B,C),/(+(D,E),-(E,F))))
```

ここで、図3.4(a)に示すように、「演算子(α , β)」の形の部分を、演算子を節点としαおよびβを葉とする木構造に置き換える操作を順次繰り返していくと、上記の式は図3.4(b)に示すような木構造で表現することができる。このような木構造で表現することによっても、括弧や優先順位を考えることなく機械的に演算を行うことが可能になり、従ってコード生成もより容易に行うことができる。

通常の式の記法から木構造への変換は、以下のようなアルゴリズムにより行うことができる。

まず二つのスタックS_1,S_2を用意し、元の式の左から右へと調べていくことにする。

(1) 変数もしくは定数の場合、それをスタックS_1に入れる。

(2) 演算子αの場合、

(i) スタックS_2が空、または、スタックS_2の最上位が (の時、αをスタックS_2に入れる

(ii) スタックS_2の最上位が演算子βの時、αとβの優先順位を比較して

(イ) αの方が低いか等しければ、スタックS_1の最上位から二つを取り出して葉とし、スタックS_2からβを取り出して節点とした部分木(subtree)を出力する。そして、その部分木を指すポインタT_i

† 式だけでなく言語全体が逆ポーランド式の言語にFORTHなどがある。

(i=1,2,...)をスタックS_1に入れて、(2)へもどる。

(ロ) αの方が高ければ、αをスタックS_2に入れる。

(3) (なら、それをスタックS_2に入れる。

(4)) または CR なら、

(i) スタックS_2の最上位が (のとき、(をスタックS_2から取り出す。

(ii) スタックS_2の最上位が演算子βの時、スタックS_1の最上位から二つを取り出して葉とし、スタックS_2からβを取り出して節点とした部分木を出力する。そしてその部分木を指すポインタT_iをスタックS_1に入れて、(4)にもどる。ただし、βが = のとき終わり。

このアルゴリズムにより、先に示した例**A=B*C−(D+E)/(E−F)**を木構造に変換した例を表3.2に示す。

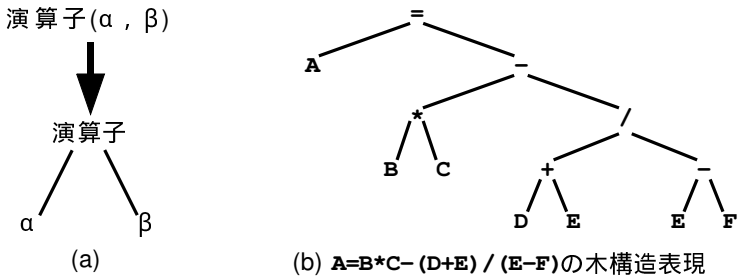


図3.4　木構造による式の表現

表3.1　逆ポーランド記法への変換

x:式の単語
A=B*C−(D+E)/(E−F)
を入力

x	スタックの内容	出力
A		A
=	=	
B		B
*	* =	
C		C
−	=	*
	− =	
(	(− =	
D		D
+	+ (− =	
E		E
)	(− =	+
	− =	
/	/ − =	
(	(/ − =	
E		E
−	− (/ − =	
F		F
)	(/ − =	−
	/ − =	
CR	− =	/
	=	−
		=

スタックの先頭　　出力は **ABC*DE+EF−/−=**

表3.2　木構造への変換

x	スタックS_1の内容	スタックS_2の内容	出力
A	A		
=		=	
B	B A		
*		* =	
C	C B A		
−	T_1 A	=	T_1=(B*C)
		− =	
(		(− =	
D	D T_1 A		
+		+ (− =	
E	E D T_1 A		
)	T_2 T_1 A	(− =	T_2=(D+E)
		− =	
/		/ − =	
(		(/ − =	
E	E T_2 T_1 A		
−		− (/ − =	
F	F E T_2 T_1 A		
)	T_3 T_2 T_1 A	(/ − =	T_3=(E−F)
		/ − =	
CR	T_4 T_1	− =	T_4=(T_2 /T_3)
	T_5 A	=	T_5=(T_1 −T_4)
	T_6		T_6=(A=T_5)

スタックの先頭　　スタックの先頭

3.5　コード生成部

前節で作成した中間表現から、実際に演算を行う機械語命令を生成する部分がこのコード生成部になる。コード生成部は中間表現の違いによって大きく分けられ、さらに対象となるアーキテクチャによっても異なってくる。以下で、それぞれの組み合わせについて順次説明する。

3.5.1　逆ポーランド記法からのコード生成

3.5.1.1　スタックアーキテクチャのためのコード生成

対象となるアーキテクチャがスタックアーキテクチャの場合、元々逆ポーランド記法の演算を行いやすいように設計されているアーキテクチャのため、逆ポーランド記法からのコード生成はたやすく行うことができる。そのアルゴリズムを以下に示す。

逆ポーランド記法の左から右へと調べていくことにする。

(1)　先頭の要素は代入先であるはずなので、変数であればその名前を記録しておく。変数以外の場合は代入できないのでエラーとなる。

(2)　現在見ているのが演算子αの場合、αに対応する機械語コードを出力する。

(3)　現在見ているのが変数の場合、PUSH命令を生成する。この場合、変数のアドレスにはラベルを用いる。

(4)　現在見ているのが定数の場合、PUSHI命令を生成する。プッシュする即値はその定数の値である。

(5)　現在見ているのが = (代入記号)の場合、(1)で記録しておいた代入先の変数に値を格納するためにPOP命令を生成し終わり。

このアルゴリズムにより、逆ポーランド式**ABC*DE+EF-/-=**からコード生成を行った例を表3.3に示す。なお、ここでは**A**～**F**はすべて変数であるものとしている。定数の場合はPUSHのかわりにPUSHI命令を使用すればよい。

3.5.1.2　スタック以外のアーキテクチャのためのコード生成

スタック以外のアーキテクチャの場合には、スタックを用意して以下のようなアルゴリズムに従いコード生成を行う。

逆ポーランド記法の左から右へと調べていくことにする。

(1)　現在見ているのが変数もしくは定数の場合、スタックにプッシュする。

(2)　現在見ているのが演算子αの場合、スタックから被演算子(変数もしくは定数)を必要なだけポップし、それらに対してαの演算を行う機械語コードを出力する。ただし、演算結果は一時記憶領域を確保しそこに入れる(この一時記憶領域をTとする)。つぎに、このTをスタックにプッシュする。

(3)　現在見ているのが = (代入記号)の場合、代入先が変数かどうかチェックし、変数であれば値を格納するためのST命令を生成し終わり。そうでなければエラーである。

上記を繰り返すことにより、コード生成を行うことができる。

上記のアルゴリズムにおいて、アーキテクチャにより異なってくるのが(2)の演算機械語コードの生成、および一時記憶領域の確保の部分である。

アキュムレータアーキテクチャの場合、一時記憶領域はすべてメモリ上に確保しなければならない。また、演算はアキュムレータに対して行われるので、以下のような段階によって演算を行う必要がある

(1) アキュムレータにロード(LD もしくは LDI)

(2) アキュムレータに対して演算(即値の場合は即値演算)

(3) アキュムレータからメモリへストア(ST)

CISCアーキテクチャの場合は、一時記憶領域としてレジスタが8個使用可能であるので、レジスタが使用可能である間はレジスタを使用し、レジスタが不足した場合のみメモリを使用するようにする。CISCアーキテクチャの場合、演算対象(メモリ、レジスタ、即値)を自由に選択できるので、アキュムレータの場合のように数段階に渡って演算を行う必要はない。

RISCアーキテクチャの場合は、一時記憶領域としてレジスタが30個使用できる。残り2つのレジスタのうち一つは、アキュムレータの代わりとして使用し、もう一つは、メモリにあるデータを演算に使うときに一時的に数値を入れておくためのレジスタとする。たとえばそれぞれR0,R1を使用すればよい。

このアルゴリズムにより、逆ポーランド式**ABC*DE+EF-/-=**からコード生成を行った例を表3.4, 3.5, 3.6に示す。なお、ここでは**A**～**F**はすべて変数であるものとしている。また、T_i(i=1,2,...)はメモリ上の一時記憶領域、Ri(i=0,1,...)はレジスタとしている。

なお、RISCアーキテクチャのコード生成例においては、アキュムレータ役のレジスタはR0、メモリ演算用一時レジスタはR1とした。

表3.3　逆ポーランド記法からのコード生成
(スタックアーキテクチャ)

x:逆ポーランド記法
ABC*DE+EF-/-=
を入力(A-Fは変数)

x	機械語命令
A	(変数名Aを記憶)
B	PUSH Var_B
C	PUSH Var_C
*	MUL
D	PUSH Var_D
E	PUSH Var_E
+	ADD
E	PUSH Var_E
F	PUSH Var_F
-	SUB
/	DIV
-	SUB
=	POP Var_A(記憶した変数名)

表3.4　逆ポーランド記法からのコード生成
(アキュムレータアーキテクチャ)

x	スタック	機械語命令
A	A	
B	B A	
C	C B A	
*	T_1 A	LD Var_B;MUL Var_C;ST T_1
D	D T_1 A	
E	E D T_1 A	
+	T_2 T_1 A	LD Var_D;ADD Var_E;ST T_2
E	E T_2 T_1 A	
F	F E T_2 T_1 A	
-	T_3 T_2 T_1 A	LD Var_E;SUB Var_F; ST T_3
/	T_4 T_1 A	LD T_2 ; DIV T_3 ; ST T_4
-	T_5 A	LD T_1 ; SUB T_4 ; ST T_5
=		LD T_5 ; ST Var_A

スタックの先頭

表3.5　逆ポーランド記法からのコード生成
(CISCアーキテクチャ)

x	スタック	機械語命令
A	A	
B	B A	
C	C B A	
*	R0 A	MUL Var_B,Var_C,R0
D	D R0 A	
E	E D R0 A	
+	R1 R0 A	ADD Var_D,Var_E,R1
E	E R1 R0 A	
F	F E R1 R0 A	
-	R2 R1 R0 A	SUB Var_E,Var_F,R2
/	R3 R0 A	DIV R1,R2,R3
-	R4 A	SUB R0,R3,R4
=		ST R4,Var_A

スタックの先頭

表3.6　逆ポーランド記法からのコード生成
(RISCアーキテクチャ)

x	スタック	機械語命令
A	A	
B	B A	LD Var_B,R0
C	C B A	LD Var_C,R1
*	R2 A	MUL R0,R1,R2
D	D R2 A	LD Var_D,R0
E	E D R2 A	LD Var_E,R1
+	R3 R2 A	ADD R0,R1,R3
E	E R3 R2 A	LD Var_E,R0
F	F E R3 R2 A	LD Var_F,R1
-	R4 R3 R2 A	SUB R0,R1,R4
/	R5 R2 A	DIV R3,R4,R5
-	R6 A	SUB R2,R5,R6
=		ST Var_A,R6

スタックの先頭

3.5.2　木構造からのコード生成

中間表現として木構造を用いた場合、基本的なコード生成のアルゴリズムは再帰呼び出しを用いることにより実現できる。以下で、各アーキテクチャのためのコード生成アルゴリズムについて、順を追って説明する。

3.5.2.1　アキュムレータアーキテクチャのコード生成

木構造からアキュムレータアーキテクチャのためのコード生成を行うアルゴリズムは、以下のようになる。まず、木構造のルートから調べ始める。

(1) 現在見ているノードが演算子で、左右の子ノードが両方ともリーフ (定数あるいは変数) の場合、そのあいだで演算を行うコードを生成する。演算結果はアキュムレータに格納する。

(2) 現在見ているノードが演算子で、左右の子ノードが両方とも演算子 (すなわちさらに下位のノードを持つ) の場合は以下のようにする。

(i) まず、右側の子ノードをルートとみなし、再帰呼び出しによりコード生成を行う。結果はアキュムレータに格納されているので、一時記憶領域を確保しそこに結果を保存する。

(ii) 次に、左側の子ノードをルートとみなし、再帰呼び出しによりコード生成を行う。結果はアキュムレータに格納されている。

(iii)左側の子ノードの演算結果と、右側の子ノードの演算結果の間で、現ノードの演算を行うコードを出力する。具体的には、左側の子ノードの演算結果はすでにアキュムレータに入っているので、アキュムレータと(i)で保存した右側の演算結果との間の演算命令を出力すればよい。結果はアキュムレータに格納する。

(3) 現在見ているノードが演算子で、右の子ノードが演算子（すなわちさらに下位のノードを持つ）、左の子ノードがリーフの場合は以下のようにする。

(i) まず、右側の子ノードをルートとみなし、再帰呼び出しによりコード生成を行う。結果はアキュムレータに格納されているので、一時記憶領域を確保しそこに結果を保存する。

(ii) 次に、左側の子ノードの値をアキュムレータにロードするコードを出力する。

(iii)アキュムレータと右側の子ノードの演算結果の間で、現ノードの演算を行うコードを出力する。具体的には、アキュムレータと(i)で保存した右側の演算結果との間の演算命令を出力すればよい。結果はアキュムレータに格納する。

(4) 現在見ているノードが演算子で、左の子ノードが演算子（すなわちさらに下位のノードを持つ）、右の子ノードがリーフの場合は以下のようにする。

(i) まず、左側の子ノードをルートとみなし、再帰呼び出しによりコード生成を行う。結果はアキュムレータに格納される。

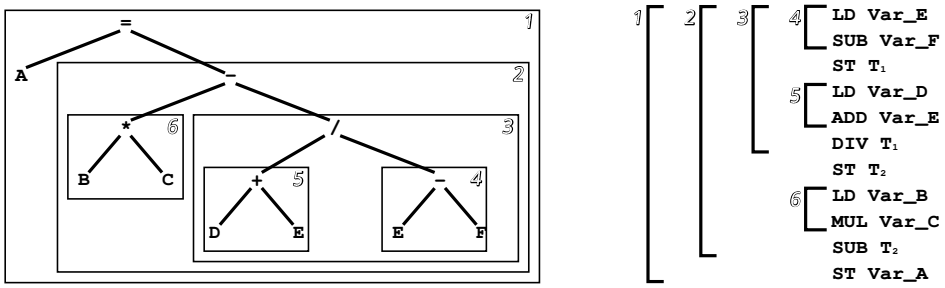
(ii) アキュムレータと右側の子ノードの値との間で、現ノードの演算を行うコードを出力する。右側の子ノードが変数か定数かによって、演算命令もしくは即値演算命令を用いる。結果はアキュムレータに格納する。

(5) 現在見ているノードが =(代入記号)かつ左の子ノードがリーフでしかも変数ならば、以下の処理を行い、コード生成を終える。左の子ノードが変数でない場合はエラーである。

(i) 右側の子ノードがリーフの場合、その値をアキュムレータにロードし、その後左の子ノードの変数にストアするコードを出力する。

(ii) 右側の子ノードが演算子の場合、右側の子ノードをルートとみなし、再帰呼び出しによりコード生成を行う。結果はアキュムレータに格納されるので、その後で左の子ノードの変数にストアするコードを出力すればよい。

このアルゴリズムによる、式**A=B*C−(D+E)/(E−F)**の木構造表現に対するコード生成の例を図3.5に示す。なお、T_i(i=1,2,...)は一時記憶領域を意味する。図中の白抜き数字は、再帰呼び出しされる順番、および生成されたコードと木構造の関係を表している。



A=B*C−(D+E)/(E−F)の木構造表現　　木構造から生成された機械語コード

図3.5　木構造からのコード生成(アキュムレータアーキテクチャ)

3.5.2.2　スタックアーキテクチャのためのコード生成

木構造からスタックアーキテクチャのためのコード生成を行うアルゴリズムは、以下のようになる。

まず、木構造のルートから調べ始める。

(1) 現在見ているノードがリーフ（定数あるいは変数）の場合、その値をプッシュするコードを出力する。

(2) 現在見ているノードが演算子の場合は以下のようにする。

(i) まず、左側の子ノードをルートとみなし、再帰呼び出しによりコード生成を行う。

(ii) 次に、右側の子ノードをルートとみなし、再帰呼び出しによりコード生成を行う。

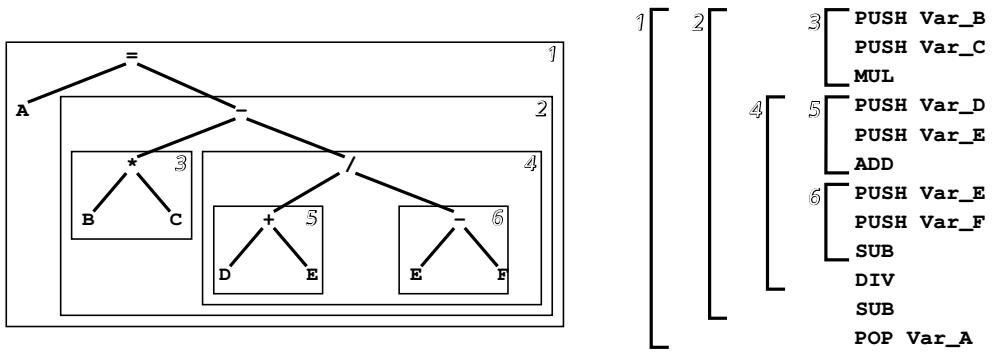
(iii)現ノードの演算を行う機械語命令を出力する。

(5) 現在見ているノードが = (代入記号)かつ左の子ノードがリーフでしかも変数ならば、以下の処理を行い、コード生成を終了する。左の子ノードが変数でない場合はエラーである。

(i) 右側の子ノードをルートとみなし、再帰呼び出しによりコード生成を行う。

(ii) その後で左の子ノードの変数にポップするコードを出力する。

このアルゴリズムによる、式**A=B*C-(D+E)/(E-F)**の木構造表現に対するコード生成の例を図3.6に示す。図中の白抜き数字は、再帰呼び出しされる順番、および生成されたコードと木構造の関係を表している。



A=B*C-(D+E)/(E-F)の木構造表現　　木構造から生成された機械語コード

図3.6　木構造からのコード生成(スタックアーキテクチャ)

3.5.2.3　CISCおよびRISCアーキテクチャのためのコード生成

CISCおよびRISCアーキテクチャの場合、木構造からコード生成を行うアルゴリズムは、基本的に同じで、以下のようになる。

まず、木構造のルートから調べ始める。

(1) 現在見ているノードが演算子の場合は、以下の処理を行う。

(i) もし左の子ノードが演算子(すなわちさらに下位のノードを持つ)ならば、左側の子ノードをルートとみなし、再帰呼び出しによりコード生成を行う。リーフであれば次に進む。

(ii) 同様に、もし右の子ノードが演算子(すなわちさらに下位のノードを持つ)ならば、右側の子ノードをルートとみなし、再帰呼び出しによりコード生成を行う。リーフであれば次に進む。

(iii) (i),(ii)の結果、左右の子ノードは必ずリーフ(変数、定数、または一時記憶領域)になるので、そのあいだで演算を行う演算命令を出力する。結果は一時記憶領域に格納し、現在のノードをその一時記憶領域を示すものに変更する。

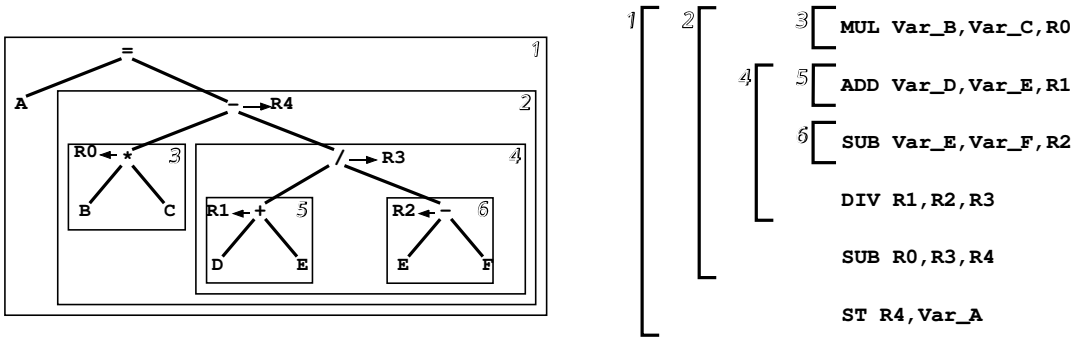
(2) 現在見ているノードが = (代入記号)かつ左の子ノードがリーフでしかも変数ならば、以下の処理を行いコード生成を終了する。左の子ノードが変数でない場合はエラーである。

(i) もし右の子ノードが演算子(すなわちさらに下位のノードを持つ)ならば、右側の子ノードをルートとみなし、再帰呼び出しによりコード生成を行う。リーフであれば次に進む。

(ii) (i)の結果、右側の子ノードは必ずリーフになるので、その値を左の子ノードの変数にストアするコードを出力する。

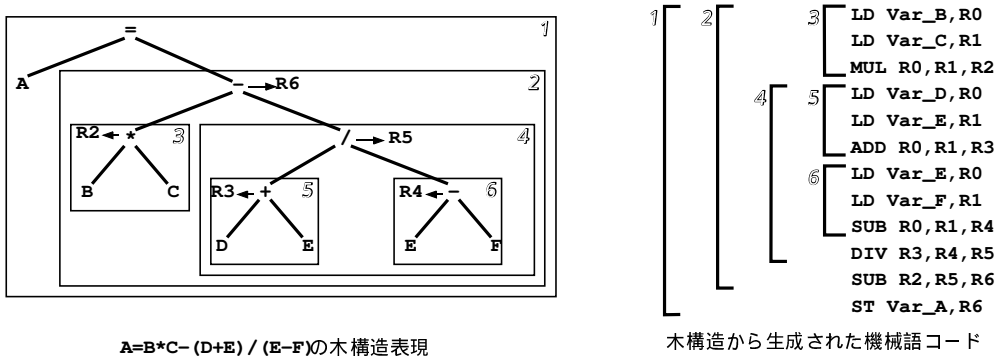
一時記憶領域としては、逆ポーランド記法の場合と同様にできるだけレジスタを使用し、レジスタが不足したときのみメモリを使用するようにする。また、RISCの場合は被演算数や格納先に応じて、演算を行う機械語コードを変化させる必要があるが、これは逆ポーランド記法の場合を参考にすること。従って、アキュムレータとメモリ演算用のレジスタを確保しておく必要があることも同じである。

このアルゴリズムによる、式**A=B*C-(D+E)/(E-F)**の木構造表現に対するコード生成の例を図3.7、3.8に示す。なお、Ri(i=0,1,...)は一時記憶領域としてのレジスタを意味する。図中の白抜き数字は、再帰呼び出しされる順番、および生成されたコードと木構造の関係を表している。



A=B*C-(D+E)/(E-F)の木構造表現　　木構造から生成された機械語コード

図3.7　木構造からのコード生成(CISCアーキテクチャ)



A=B*C-(D+E)/(E-F)の木構造表現　　木構造から生成された機械語コード

図3.8　木構造からのコード生成(RISCアーキテクチャ)

４．　実験方法

本実験では、UNIXワークステーションを使用し実験を行う。ワークステーションのOSであるUNIXや、ウインドウシステム、エディタ等の基本的使用法に関しては、すでに知っているはずなのでここでは省略する。

実験の順序としては、まず式の変換アルゴリズムについてよく理解した後、各アーキテクチャ用のコンパイラ（それぞれ2種類の中間記法を用いている）、アセンブラおよびシミュレータを用いて、実際にコンパイルと機械語プログラムの実行を行い、実際の計算機における高級言語プログラム実行の基本を実習する。

4.1　式の作成

まず、以降の実験においてソースプログラムとして使用する式のプログラムをエディタを使用して作成する。その形式は以下のようにすること。はじめに定数への代入を行い、最後に目的の式を書く。

```
A=10
B=20
C=30
D=40
E=(A+B)*(C-D)/A
```

このとき、最後の式の直後には必ず改行を入れること。また、最後の式より先に空行を入れると、コンパイラが正常動作しないので注意する。

4.2　式のコンパイルの実行

本実験では、すでに説明した2つの中間表現と4つのアーキテクチャの組み合わせにより、8種のコンパイラを使用して、式のコンパイルを行うことができる。

コンパイラは標準入力から式を入力して、標準出力にコンパイル結果のアセンブリ言語プログラムを出力するようになっているので、以下のようにしてコンパイルを実行する。

```
% r-risc < expr.src > r-risc.asm [Ret]
```

ここで、r-riscはコンパイラのコマンド名、expr.srcは式のプログラム、r-risc.asmはコンパイル結果のアセンブリ言語プログラム名である。アセンブリ言語プログラム名は上記のようにあとで区別できるような名前を付けておくこと。

4.3　アセンブリ言語プログラムの実行

コンパイラの出力したアセンブリ言語は、アセンブラにより機械語プログラムに変換され、シミュレータを用いて実行することができる。

まず、アセンブラを用いてアセンブリ言語プログラムを実行可能な機械語プログラムに変換(アセンブル)する。アセンブラは各アーキテクチャによって、4種用意してあるので、対象としたアーキテクチャ用のアセンブラを使用する。

アセンブラの使用法は、たとえばRISCアーキテクチャで、コンパイラの出力したアセンブリ言語プログラムファイルの名前をr-risc.asm(拡張子は.asmとする)の場合、

```
% rasm r-risc.asm [Ret]
```

のように入力することにより、r-risc.asmがアセンブルされて、エラーがなければ機械語オブジェクトプログラムファイルr-risc.objおよびリスティングファイルr-risc.lstが出力される。リスティングファイルとは、アセンブリ言語と変換された機械語プログラム、および格納番地を対応させて出力したファイルである。アセンブリ言語プログラムと、それをアセンブルした結果のリスティングファイルの例を図4.1(a),(b)に示す。

```
; compiled code for A=B*C
; (RISC architecture)
;
        org     0
        ld      Var_B,R0
        ld      Var_C,R1
        mul     R0,R1,R2
        st      Var_A,R2
        hlt
;
Var_A:  ds1
Var_B:  ds1
Var_C:  ds1
```

格納番地(16進数)　機械語コード(16進数)

```
0000
0000  0000 0160
0002  0000 0181
0004  5000 0022
0006  0840 0140

0008  C800 0000

000A  0000
000B  0000
000C  0000
```

```
; compiled code for A=B*C
; (RISC architecture)
;
        org 0

        ld      Var_B,R0
        ld      Var_C,R1
        mul     R0,R1,R2
        st      Var_A,R2

        hlt
;
Var_A:  ds      1
Var_B:  ds      1
Var_C:  ds      1
```

(a) アセンブリ言語プログラム　　(b) リスティングファイル

図4.1　アセンブリ言語プログラムとリスティングファイル

機械語ファイルが出力されたら、シミュレータを用いてその機械語ファイルを実行する。シミュレータは各アーキテクチャによって、４種用意してあるので、対象としたアーキテクチャ用のシミュレータを使用する。

シミュレータの使用法は、たとえばRISCアーキテクチャで機械語プログラムファイルの名前をr-risc.obj(拡張子は.objとする)の場合、

```
% rsim r-risc.obj [Ret]
```

のように入力すると、シミュレータの実行が開始され、コマンド待ちの状態になる。シミュレータの画面構成を図4.2に示す。上部から、コマンド入力部、実行ステップ数と現在のPCの表示、現在実行中の命令、メモリの内容、そして一番下がレジスタ / アキュムレータ / スタックの内容を表示している。

主なコマンドには以下のようなものがある。なお、実行時エラーやHLT命令により、シミュレータは停止しコマンド待ちになる。

(1) run:　実行を開始する。パラメータとして実行開始番地を渡せる。デフォルトでステップ実行モードになっているので、リターンキーを押すごとに一命令づつ実行される。q[Ret]またはctrl-C [Ret]により実行を一時中断できる。

(2) cont: 一時中断した実行を再開する。
(3) init: シミュレータのイニシャライズを行う。
(4) dmem: メモリの内容表示を行う。jおよびkで1行、nおよびpで1画面スクロールする。qでコマンド待ちに戻る。
(5) dreg: (RISCシミュレータのみ)レジスタの内容表示を行う。jおよびkで1行、nおよびpで1画面スクロールする。qでコマンド待ちに戻る。
(6) load: パラメータとしてファイル名を指定することにより、別な機械語プログラムをシミュレータにロードする。
(7) quit: シミュレータを終了する。

シミュレータの画面表示において、数値は基本的に16進数で表示されている。( )の中に表示されているのが10進数である。コマンドのパラメータで数値を与える場合、C言語の書式を用いること(16進数なら0xを頭に付ける)。

実行の際には、ウインドウを 2 つ使用し、lessコマンド等を用いて片方でアセンブラの出力のリスティングファイルを参照しながら、もう一方のウインドウでシミュレータを実行する。このとき、runコマンドで実行を開始する前に、dmemコマンドを用いて変数領域のメモリの内容を表示しておくようにすること。変数領域のアドレスは、リスティングファイルの最後の方を参照すれば調べられる。

runコマンドにより実行を開始した後は、リスティングファイルの対応する部分と、メモリやレジスタ等の内容を比較対照しながら、実行の様子を確認する。

```
COMMAND:
Command error.
Commands : run init trace cont dreg dmem bp clearbp wait step status load
dumpmem quit
cycle        0 : PC = 0000(     0)

Current instruction : LD        0064H,R0

Memories :
Mem[012C(   300)]:0000(     0)          Mem[012D(   301)]:0000(     0)
Mem[012E(   302)]:0000(     0)          Mem[012F(   303)]:0000(     0)
Mem[0130(   304)]:0000(     0)          Mem[0131(   305)]:0000(     0)
Mem[0132(   306)]:0000(     0)          Mem[0133(   307)]:0000(     0)
Mem[0134(   308)]:0000(     0)          Mem[0135(   309)]:0000(     0)
Mem[0136(   310)]:0000(     0)          Mem[0137(   311)]:0000(     0)
Mem[0138(   312)]:0000(     0)          Mem[0139(   313)]:0000(     0)
Mem[013A(   314)]:0000(     0)          Mem[013B(   315)]:0000(     0)
Mem[013C(   316)]:0000(     0)          Mem[013D(   317)]:0000(     0)
Mem[013E(   318)]:0000(     0)          Mem[013F(   319)]:0000(     0)

Registers:
R00:0000(     0)                        R01:0000(     0)
R02:0000(     0)                        R03:0000(     0)
R04:0000(     0)                        R05:0000(     0)
R06:0000(     0)                        R07:0000(     0)
R08:0000(     0)                        R09:0000(     0)
R10:0000(     0)                        R11:0000(     0)
R12:0000(     0)                        R13:0000(     0)
R14:0000(     0)                        R15:0000(     0)
R16:0000(     0)                        R17:0000(     0)
R18:0000(     0)                        R19:0000(     0)
```

図4.2　シミュレータ(rsim)の画面構成

4.4　補足事項

(a)　実験で使用するコンパイラ、アセンブラ、シミュレータのコマンド名は、各ターゲットアーキテクチャと中間記法に対応して図4.3のようになっているので、間違わないようにすること。異なったアーキテクチャのアセンブラやシミュレータを使用すると、エラーを生じることがある。

(b)　コンパイラがアセンブリ言語ファイルを出力する際、式に対応した中間表現を、アセンブリ言語プログラム中のコメント分の形式で出力する。逆ポーランド記法の場合には上から下へただ並んでいるだけであるが、木構造の場合には木のレベルと左右の枝を表すR/Lを組み合わせて、図4.4のような形式で出力される。

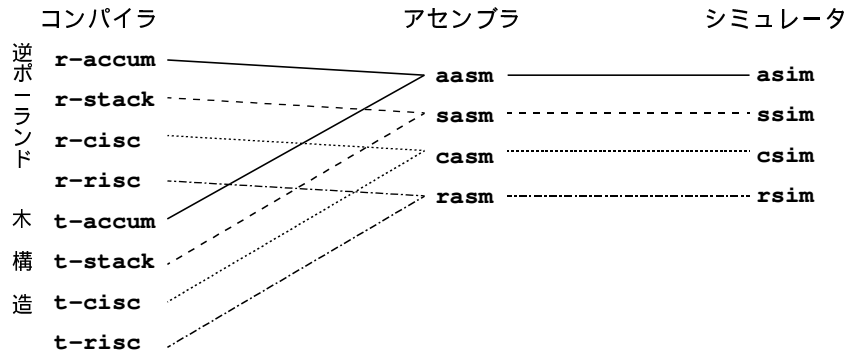


図4.3　コンパイラ、アセンブラ、シミュレータのコマンド名

式：　**A=(B+C)*(D-E/F)**

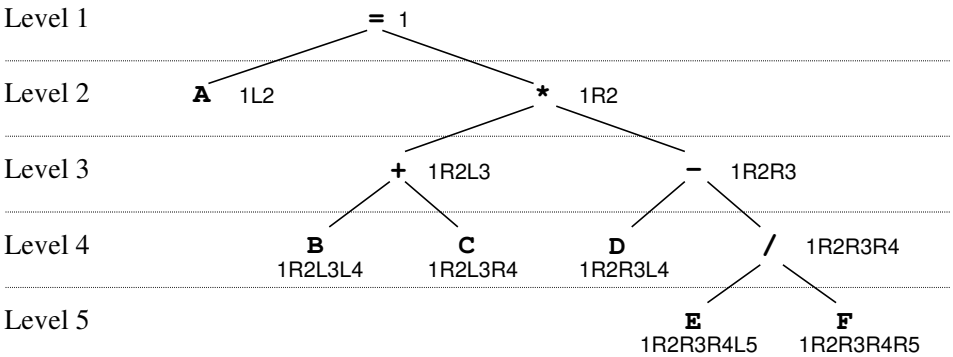


図4.4　リスティングファイルにおける木構造の表現例

5．考察課題

(1)　本実験で使用した各アーキテクチャの実例について調べよ。また、現在使用されている各種の計算機のCPUの機械語や、コンパイラで処理されている言語についても調べてみよ。

(2)　それぞれの中間記法、およびアーキテクチャについて、適当な式の例を用いて、コンパイルの手順を順を追って確認せよ。簡単な式ではなく、できるだけ各種の演算子や、かっこの入った式を例に用いること。

(3)　実験の結果から、CISCアーキテクチャの方がRISCアーキテクチャよりも少ない機械語命令ステップ数で実行できることがわかるが、一般にはRISCアーキテクチャの方がより高速な実行が可能であることが知られている。RISCアーキテクチャが、命令ステップ数が多いにも関わらず高速な実行が可能となっている理由について調べよ。

(4)　本実験で説明したアルゴリズムでコード生成を行うと、効率の良い機械語コードが生成できない場合がある。実際のコンパイラでは、効率をよくするために最適化という処理が行われている。効率をよくするための最適化処理としてはどのようなものが考えられるか。

以上は必修の課題である。余力がある場合には以下の課題についても考察せよ。

(5)　今回は式のみのコンパイラであったが、実際のコンパイラではC言語におけるif文やfor文のような条件分岐および繰り返し文の処理も必要になる。このような処理を行うためには現在の仮想計算機の命令に加えてどのような命令が必要になるか。また、これらの文に対する機械語コードはどのようになるか。自分で新たに必要となる機械語命令を定義し、適当なソースプログラムを用いてコンパイルされた機械語コードを例示せよ。

(6)　最近のＣＰＵは同時に実行可能な複数の命令を並行して実行することにより高速化を図るスーパースカラという処理方式を採用しているものが多い。本実験で使用したRISCアーキテクチャの機械語プログ

ラムについて、もしスーパースカラ方式を採用した場合にどのように実行されるか考察せよ。

(7) 本実験で使用したCISC,RISCアーキテクチャの機械語のエンコーディングがどうなっているか、実験で使用した式のコンパイル・アセンブル結果を利用して解析を試みよ。

６．参考文献

本資料の作成において、以下の文献を参考にした。

萩原　他：電子計算機　ソフトウエアの基礎（４章）オーム社
中西正和、大野義夫：やさしいコンパイラの作り方（４章）共立出版

さらにコンパイラについて詳しく知りたい人は、上記の文献の他、たとえば以下のような文献を参照するとよい。

中田育男：コンパイラ、産業図書
島内　他：コンパイラのうちとそと、共立出版
エイホ、セシィ、ウルマン：コンパイラI,II、サイエンス社
ピットマン＆ピーターズ：コンパイラ設計技法　理論と実践、トッパン

また、RISCアーキテクチャを中心とした計算機アーキテクチャ一般についての参考書としては、たとえば以下のような文献があげられる。

ヘネシー＆パターソン：コンピュータ・アーキテクチャ　設計・実現・評価の定量的アプローチ、日経BP
パターソン＆ヘネシー：コンピュータの構成と設計（上）（下）日経BP

最近の新しいCPUのアーキテクチャに関する情報については、たとえば以下のような文献がある。

日経エレクトロニクス編：スーパーチップ最前線、日経BP
日経バイト：最新マイクロプロセサテクノロジ、日経BP

「機械知能工学実験−コンパイラ」
1993年10月　第 1 版
1994年10月　第 2 版
1995年 7月　改訂第 1 版
1996年 8月　改訂第 2 版
1997年 8月　改訂第 3 版
東北大学工学部機械知能工学科　計算機システム工学講座
東北大学大学院情報科学研究科ソフトウエア科学講座アーキテクチャ学分野
片平昌幸
Copyright © 1993,1994,1995,1996,1997 by Masayuki Katahira