論文 88-25-3-6

머신 독립적인 고급 마이크로프로그래밍 언어의 설계

(The Design of a Machine Independent High Level Microprogramming Language)

李 相 靜*, 林 寅 七*

(Sang Jeong Lee and In Chil Lim)

要 約

본 논문에서는 C언어와 유사한 구조를 갖는 머신 독립적인 고급 마이크로프로그래밍 언어 HLML-C (high level microprogramming language C)를 제안한다. 즉, 다양한 머신의 특성을 고려하여 일반화시킨 추상머신(abstract machine) 상에서 언어의 오퍼레이션을 정의하고, 머신 독립적으로 추상머신의 중간언어를 생성시킨다. 중간언어 생성시 효율적인 마이크로코드의 생성을 위하여 머신 독립성을 유지하면서 대상머신(target machine)에 종속적인 특수한 오퍼레이션의 확장 적용과 각 하드웨어 자원의 효율적인 이용이 가능한 고급 마이크로프로그래밍 언어를 제안한다.

제안된 HLML-C 언어의 컴파일러는 VAX 11/750 (4.3BSD) 상에서 yacc와 C언어로써 실현하고, 다양한 예의 마이크로프로그램에 대하여 적용시키고 검토 분석한다.

Abstract

In this paper, HLML-C (High Level Microprogramming Language C) is proposed, which is independent of target machines and has similar structure to C language. The HLML-C operations are defined for a abstract machine which contains characteristics of various microarchitectures, and can extend to define a target machine's special operations for efficient microcode generation. A microprogram written in this language is translated into a machine independent intermediate language on abstract machine with the information of a target machine's resource usage and then microoperations of a target machine.

The HLML-C compiler is implemented with yacc and C language on VAX-11/750 (4.3 BSD) computer. Through the various test microprogram applied to HLML-C compiler, their results are analyzed.

I. 서 론

최근 컴퓨터 시스템의 요구가 다양해지고 반도체 기술이 발전하여 시스템이 복잡해짐에 따라 시스템을 마이크로코드(microcode)로써 제어하는 프로세서(processor)가 보편화되고 있는 추세이다. 또한 시스템의 효율(performance), 신뢰도(reliability), 함수성(functionality) 및 기밀유지(security) 향상을 위하여 지금까지 소프트웨어로 실현하였던 시스템이 자주 사용하는 함수들의 마이크로프로그램으로의 이

*正會員, 漢陽大學校 電子工學科
(Dept. of Elec. Eng., Hanyang Univ.)

接受日字 : 1987年 11月 24日

식이 증대되고 있다. 또 사용자가 직접 마이크로프로그램을 작성하여 시스템을 제어하는 마이크로프로그래머블 프로세서(microprogrammable processor)의 등장으로 마이크로프로그램이 널리 사용되고 있다.

마이크로프로그램은 프로세서 내부의 하드웨어 동작을 직접 제어하는 낮은 수준의 프로그램이기 때문에 프로그램의 작성과 유지가 일반 프로그램 보다 어렵고, 특히 수평 마이크로명령어 형식(horizontal microinstruction format)을 갖는 프로세서가 보편화되면서 기존의 마이크로어셈블리 언어(micro-assembly language)나 마이크로코드로써 마이크로프로그램의 직접 작성이 더욱 어렵게 되었다.

따라서 마이크로프로그램을 일반 프로그램의 고급 언어와 같이 고급 마이크로프로그래밍 언어(high level microprogramming language, HLML)를 개발, 사용함으로써 마이크로프로그래머가 프로세서 내부동작 및 하드웨어 구조를 자세히 모르더라도 적은 비용과 노력으로 마이크로프로그램을 작성함으로써 프로그래머의 생산성(productivity)을 높이고, 프로그램의 신뢰도(reliability)를 향상 시키면서 마이크로프로그램의 유지(maintenance) 및 디버깅(debugging)을 용이하게 하고자하는 마이크로코드 자동생성 툴에 관한 연구가 활발히 진행 중 이다.[1~11]

이러한 HLML에 관한 연구는 HLML의 오퍼레이션이나 데이타 구조를 특정 대상머신(target machine)의 하드웨어 자원(resource)에 직접 대응(mapping) 하도록 언어를 정의한 머신 종속적인 HLML (machine dependent HLML)과[1~4] HLML의 오퍼레이션이나 데이타 구조 등이 특정 머신에 귀속됨이 없이 머신 독립적으로 정의하고, 컴파일러가 코드생성(code generation) 과정에서 대상머신에 결합시키는 머신 독립적인 HLML(machine independent HLML)의 두분야로 나누어 진행되고 있다.[5~11]

머신 종속적인 HLML은 코드 생성 과정이 간단하고 HLML 오퍼레이션을 어느 특정 머신에 귀속하여 정의함으로써 효율적인 마이크로코드를 생성할 수 있다는 장점이 있으나 이용 범위가 특정 머신에 국한됨으로써 그 개발 노력과 비용에 비하여 효용 가치가 적으므로 현재는 머신 독립적인 HLML 개발에 연구가 집중되고 있는 실정이다. 머신 독립적인 HLML은 컴파일러가 어느 중간 단계까지만 특정 머신에 관계 없이 공통적으로 수행하고 마이크로코드 생성 과정에서 대상머신에 맞도록 재구성함으로써 다양한 머신에 적용 가능 하도록 시도하고 있다. 그러나 머신 독립적인 HLML은 컴파일러 자체가 복잡하고, 효율적인 마이크로코드 생성이 어렵다는 문제점이 있다. 이러한 머신 독립적인 HLML에 관한 연구로는 Ramamoorthy et. al.의 SIMPL,[1] DeWitt의 EMPL,[5] Malik의 VMPL,[6] Dasgupta의 S*,[1] Davidson의 MABLE,[3] Marwedel의 MIMOLA,[10] Hopkins의 Micro-C[11] 등이 있으나 아직까지는 완전히 머신 독립적이면서 효율적인 마이크로코드를 생성하는 HLML은 개발되지 못하고 있는 실정이다.

본 논문에서는 일반 C언어 형태와 유사한 구조를 갖는 머신 독립적인 고급 마이크로프로그래밍 언어인 HLML-C를 개발, 제안한다. 즉, 다양한 머신의 특성을 고려하여 일반화시킨 추상머신(abstract machine)을 설정하고, 이 추상머신 상에서 HLML 오퍼레이션을 정의 한다. 또 HLML-C로 쓰여진 소스 마이크로프로그램을 머신 독립적으로 추상머신의 중간언어로 변환한다. 중간언어 생성 시 효율적인 마이크로코드 생성을 위하여 머신 독립성을 유지하면서 대상머신에 종속적인 특수한 오퍼레이션의 확장 적용이 가능하게 하고, 각 머신의 하드웨어 자원 특성을 고려하여 중간언어를 생성하는 머신 독립적인 고급 마이크로프로그래밍 언어 HLML-C를 제안한다.

또한 VAX 11/750(4.3 BSD) 상의 컴파일러 자동생성기인 yacc와 C언어로써 HLML-C 컴파일러를 실현하고 다양한 마이크로프로그램의 예에 적용시켜 비교 검토하여 봄으로써 제안된 컴파일러의 타당성을 입증한다.

Ⅱ. 시스템 구성

HLML-C 시스템은 모든 마이크로아키텍처에 일반적이고 공통적으로 수행되는 하드웨어 특징을 갖는 추상머신을 대상으로 하여 머신 독립적으로 중간언어를 생성함으로써 가능한 한 다양한 마이크로아키텍처에 머신 독립적으로 적용 가능하도록 설계하였다. 추상머신은 다양한 마이크로아키텍처[14,15]의 다음과 같은 하드웨어 기능들을 고려하였다.

- △ ALU 함수 기능
- △ 쉬프트/로테이션(shift/rotation) 기능
- △ 메모리 억세스 기능
- △ 하드웨어 스택(stack) 억세스 기능
- △ 분기 조건코드의 세팅(branch condition code setting) 기능

HLML-C 오퍼레이션은 위와 같은 일반적인 하드웨어 특징과 함께 특정 대상머신이 갖는 고유 기능 및 I/O 마이크로오퍼레이션을 수행함으로써 효율적인 마이크로코드를 생성하도록 대상머신의 특정 레지스터 지정과 특수 기능의 마이크로오퍼레이션으로 확장 적용이 가능하도록 설계하였다. 즉, 대상머신이

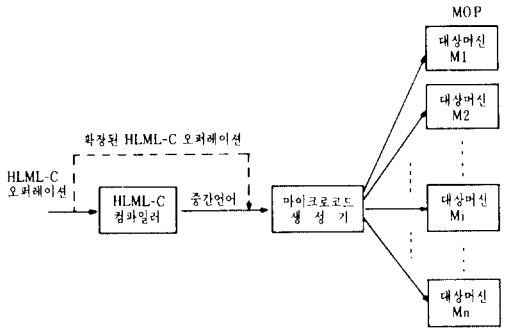

그림 1. HLML-C 오퍼레이션 변환 모델
Fig. 1. HLML-C Operation translation model.

갖는 고유의 특성을 HLML-C로 작성 가능하게 함으로써 머신 독립적으로 효율적인 마이크로코드 생성을 시도하였다. 그림 1은 HLML-C의 오퍼레이션이 다양한 대상머신들의 마이크로오퍼레이션으로 변환되는 모델을 나타낸 그림이다.

HLML-C 시스템은 크게 두 부분으로 나눌 수 있다(그림 2). 즉, HLML-C언어로 작성된 소스 마이크로프로그램이 컴파일러를 통하여 어휘(lexical) 및 구문(syntax) 분석 된 후 머신 독립적인 중간 언어(machine independent intermediate language, MIIL)를 생성하는 과정과 이 중간언어로부터 각 대상머신의 마이크로코드를 생성하는 과정으로 나눌 수 있다. HLML-C로 작성된 소스 마이크로프로그램 알고리즘의 정확성 검증을 위해 소스 마이크로프로그램의 선언문의 일부 수정과 입출력문을 삽입한 후 기존의 C 컴파일러를 이용하여 검증한다.

MIIL은 다양한 머신의 특성을 고려하여 일반화시킨 추상머신을 대상으로 하여 머신 독립적으로 생성된다. 생성된 MIIL의 각 기호 변수(symbolic variable)들은 대상머신의 레지스터에 할당(allocation), 지정(assignment)되어 머신 종속적인 중간언어(machine dependent intermediate language, MDIL)를 생성한다. 생성된 MDIL은 시뮬레이션과 디버깅을 통해 확인 검증된 후 변환표 구동방식(table drive method)에 의하여 마이크로오퍼레이션(microoperation, MOP)들로 변환되고, 이러한 MOP들은 각 머신의 병렬수행성을 고려하여 컴팩션시킴으로써 최종적인 마이크로명령어(microinstruction, MI)로 구성된 대상머신상에서 수행 가능한 마이크로코드를 생성한다.

Ⅲ. HLML-C 언어

1. 선언문

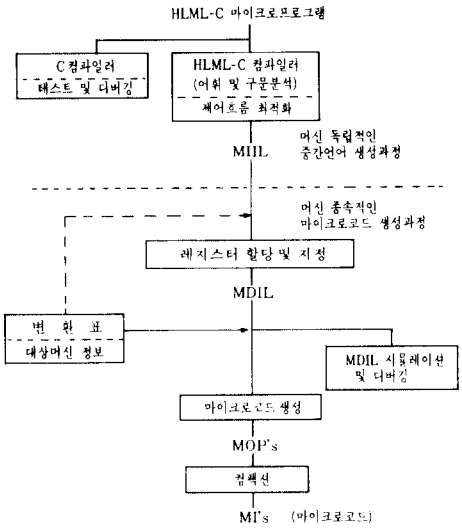

그림 2. HLML-C 시스템 구성도
Fig. 2. HLML-C system configuration.

1) 변수 선언문

HLML-C에서 사용되는 모든 변수의 데이타 형태는 대상머신의 하드웨어 자원 내용을 나타내기 위한 정수 값을 갖는다. HLML-C에서는 각 변수를 하드웨어 자원의 형태에 따라 메모리, 레지스터, 스택 변수로 구분되고 이러한 변수들은 사용하기 전에 비트 크기와 함께 반드시 선언되어야 한다.

주 메모리를 나타내기 위한 메모리 선언문은 다음과 같이 2가지 형식이 있다.

〔형식〕

memory bit_size symbol_variable〔[array_limit] …〕: address, …;

mainmem bit_size symbol_variable〔word_size〕

memory 선언문은 외부 입력등으로 메모리의 주소가 이미 고정된 경우 직접 메모리의 주소를 변수(또는 배열변수)로 선언하는 문이다. 배열변수로 선언된 경우 명시된 주소는 배열이 할당될 첫번째 주소이다. mainmem 선언문은 마이크로프로그램 내에서 직접 메모리의 주소를 지정하여 메모리 변수를 선언한다. mainmem으로 선언되는 변수는 오직 하나로 1차원의 배열변수이다.

레지스터를 나타내는 변수에 대한 선언은 일반 범용 레지스터 변수에 대한 선언과 대상머신의 특정 레지스터를 직접 기호변수로 지정하여 선언하는 문이 있다.

〔형식〕

register bit_size variable_list ;

spreg bit_size symbol-variable : register, … ;

register 선언문은 일반 범용 레지스터 변수를 선언한다. register 선언문으로 선언된 변수들은 마이크로코드 생성 시 레지스터 할당 및 지정과정에서 대상머신의 범용 레지스터에 결합된다. spreg 선언문은 특정 레지스터를 직접 지정함으로써 특정 오퍼레이션에 대한 효율적인 마이크로코드를 생성한다.

스택선언문은 대상머신에 하드웨어 스택이 있는 경우 선언되고 이 변수에 대한 할당(assignment) 오퍼레이션은 중간언어 생성 시 PUSH/POP 오퍼레이션 코드를 생성한다.

〔형식〕

stack bit_size symbol_variable, … ;

또 상수를 선언하기 위한 define 선언문은 자주 사용하는 상수를 기호로 표시함으로써 마이크로프로그램의 기술 및 이해를 쉽게한다.

〔형식〕

#define symbol constant

2) 오퍼레이션 확장 선언문

대상머신에 종속적인 특수한 오퍼레이션의 확장선언을 위해 extension 문으로 대상머신에 종속적인 특수한 오퍼레이션을 선언한다.

〔형식〕

extension OP_code_list ;

대상머신에 종속적인 특수한 오퍼레이션은 직접 중간언어(MIIL) 형태의 OP 코드로 선언하고, 선언된 OP 코드를 이용하여 직접 중간언어 형태로 특수한 오퍼레이션을 기술함으로써 리던던트한 마이크로코드의 부가없이 효율적으로 마이크로코드를 생성한다.

〔예〕 범용 레지스터 변수 a, b, c에 대해 일반 HLML-C 오퍼레이션으로써 a=~b&c를 수행하면 다음과 같은 중간언어가 생성된다.

```
CMP  b
AND  a  b  c
```

즉, b를 complement 시킨 후 b와 c의 논리곱의 결과를 a에 할당한다. 그러나 Am2901의 ALU는 ~ R & S를 수행하는 기능이 있어서 complement와 논리곱을 하나의 마이크로오퍼레이션으로 수행한다. 이와 같이 대상머신 종속적인 a=~b & c는 다음과 같은 extension 문과 중간언어로 직접 작성할 수 있다.

```
extension CAND ;
microprogram Am2901 ;
{register a, b, c ;
        :
if(a  ! =0)
  CAND  a  b  c ;
        :
}
```

즉, 직접 대상머신 종속적인 특수 기능의 마이크로오퍼레이션을 확장 선언하고 특수 기능의 중간언어(MIIL)를 직접 작성함으로써 변환표 구동방식에 의하여 마이크로코드 생성 시 리던던트한 부가적인 코드를 없앨 수 있다.

2. 연산자

HLML-C에서 사용되는 연산자는 대상머신의 primitive 오퍼레이션을 정의한 것으로써 대상머신의 프로세서 내부 동작 및 ALU 함수 기능에 따라 다양하기 때문에 HLML-C에서는 다양한 대상머신에서 기본적으로 서술될 수 있는 연산자를 가능한 한 광범위하게 정의하였다. HLML-C에서 정의되지 않는 대상머신 종속적인 특수한 연산은 확장 선언문에 의해 선언된 특수 연산으로 OP 코드을 사용하여 직접 중간언어 형태로 작성한다. HLML-C에서 정의된 연산자는 아래와 같은 부류로 나눌 수 있다.

△산술 연산자(arithmetic operator)
△논리 연산자(logical operator)
△비교 연산자(relational operator)
△쉬프트 연산자(shift operator)
△기타 연산자

표 1은 HLML-C에서 정의된 연산자를 나타내고, 표 2는 각 연산자의 오퍼레이션 우선순위를 나타내는 표이다.

3. 제어문

HLML-C에서는 일반 고급언어와 같은 기능을 유지하기 위하여 프로그램의 각 동작의 실행 순서를규규정하여 제어흐름(control flow)을 결정하는 아래와 같은 제어문들이 있다. 각 제어문의 사용 형식은 일반 C언어와 유사한 형식을 취한다.

△조건 분기문(conditional branch statement) :
if… (**else**)…
△무조건 분기문(unconditional branch statement):
goto
△다중 분기문(multiway branch statement) :
switch(with case, default, break)

표 1. HLML-C 연산자
Table 1. HLML-C operators.

종 류	연 산 자	의 미
산 술 연산자	+, -, *, / +¦, -¦ ++, --	add., sub., mult, div. addition, subtraction with carry increment, decrement
논 리 연산자	~, &, ¦, ^ ~&, ~¦, ~^	not. and, or, exclusive or not and, not or, equlivalence
비 교 연산자	OF, UF, PS NG, ZR, CR ==, !(=) >(=), <(=) &&, ¦¦	overflow, underflow, positive conditon negative, zero, carry conditon equal, not(equal) greater than(equal), less than (equal) and, or
쉬프트 연산자	≫(¦) ≪(¦) ⋙, ⋘	shift right filled by 0(1) shift left filled by 0(1) right rotation, left rotation
기 타	@ // 확장 연산자 복합 연산자	memory address concatenation extension operation '산술, 논리, 쉬프트 연산자'+'='

표 2. HLML-C 연산자 오퍼레이션 우선순위
Table 2. The precedence of HLML-C operators.

우선순위	연 산 자
1	( ), [ ]
2	++, --, ~, !, @
3	≫, ≫¦, ≪, ≪¦, ⋙, ⋘, //
4	&, ¦, ^, ~&, ~¦, ~^
5	*, /
6	+, -, +¦, -¦
7	OF, UF, PS, NG, ZR, CR
8	<, <=, >, <=
9	==, !=
10	&&, ¦¦
11	=, 확장 연산자, 복합 연산자

△반복 구조문(loop construct statement) :
while,
for,
do··· while

다음 예는 4비트 슬라이스인 Am2901 ALU[15] 를 4개 cascade로 연결시킨 가상의 대상머신 상에서 8비트 곱셈의 예이다.

〔예〕 곱셈 마이크로프로그램의 예

```
/* 8bit multiplication of sign magnitude form */
#define SIGN 0X8000 /* for sign extraction */
#define DIGIT 0X7FFF /* for digit extraction */
memory 16 A : 100, /* multiplicand */
          B : 101, /* multiplier */
          C : 102; /* result */
microprogram multiplication;
{register 16 mier, mcand, result;
 mier=B; /* get multiplier in the register */
 result=(mier & SIGN+A) & SIGN; /* get sign result*/
 mier &=DIGIT; /* get digit of multiplier */
 mcand=A & DIGIT; /* get digit of multiuplicand */
 while(mier != 0) {/* iterate until mier is zero */
  if((mier & 1)==1) /* if LSB of mier is one */
    result+=mcand; /* add mcand to result */
  mier≫=1; /* shift right one mier */
  mcand≪=1; /* shift left one mcand */}
  C=result; /* store result in memory */}
```

Ⅳ. 중간언어

1. MIIL 형식

HLML-C로 작성된 마이크로프로그램은 가능한 한 모든 오퍼레이션을 수용하는 추상머신을 모델로 하여 머신 독립적인 중간언어(MIIL)를 생성한다. 생성된 MIIL로부터 각 대상머신에서 수행 가능한 마이크로코드의 생성을 위해 변환표 구동방식에 의하여 각 대상머신 종속적인 마이크로코드를 생성하는 계층적 구조(hierachical structure)로 구성함으로써 마이크로프로그램의 이식성(portability) 및 대상머신 독립성(independence)을 보장하고 컴파일러의 설계를 용이하게 하였다. HLML-C의 MIIL은 추상머신의 어셈블리 언어 형식으로 3주소 코드(3 address code) 형태로 중간언어를 생성한다.

〔형식〕

OP_code destination source1 source2

모든 MIIL은 문번호로 시작되는 고정된 형식(fixed format)을 가지며, OP코드에 따라 오퍼랜드의 종류 및 수가 결정된다. HLML-C의 MIIL은 다음과 같은 종류로 나눌 수 있다.

△할당문
△산술 논리문
△쉬프트문
△제어문
△기타

표 3은 HLML-C MIIL의 각 OP 코드를 나타내는 표이다.

2. 변수의 분류

HLML-C 컴파일러는 중간언어 생성 시 대상머신

표 3. MIIL의 OP 코드
Table 3. The OP code of MIIL.

종 류	OP 코 드	의 미
할 당 OP 코드	MOVE	move register to register
	RMOVE, WMOVE	read, write memory with specified address
	LOAD, STORE	load, store memory with prefixed address
	POP, PUSH	pop, push hardware stack
산 술 논 리 OP 코드	ADD, SUB, MPY, DIV	add., sub., mult., div.
	ADDC, SUBC	addition, subtraction with carry
	AND, OR, XOR	and, or, exclusive or
	NAND, NOR, NXOR	not and, not or, equivalence
	INC, DEC, CMP	increment, decrement, complement
쉬프트 OP 코드	SHR0, SHR1, SHL0, SHL1	shift right, left (filled by 0, 1)
	ROTR, ROTL	rotation right, left
제 어 OP 코드	JUMP	unconditional jump
	CJMPGE, CJMPGT	comditional jump with greater equal/
	CJMPLE, CJMPLT	than, less equal/than,
	CJMPEQ, CJMPNE	equal, not equal
	CJMP flag code	flag code : OVERFLOW, UNDERFLOW, POSITIVE NEGATIVE, ZERO, CARRY
	JUMPSUB	jump to procedure
	POPSUB, PUSHSUB	pop, push return address
pseudo OP 코드	ENTRY, EXIT	specify microprogram beginning, end
	PROCEDURE	specify procedu re beginning
	END-PROCEDURE	specify procedure end

의 범용 레지스터가 무한한 것으로 가정하고 레지스터 할당 및 지정 과정에서 대상머신의 가용 레지스터(avaiable register) 수 보다 변수의 수가 더 많은 경우 초과한 만큼의 변수를 할당된 메모리 위치로 STORE 하고, 이 변수를 사용하는 경우 메모리로 부터 레지스터에 LOAD 하여 사용한다. 일반적으로 고급언어로 작성된 마이크로프로그램에서 변수의 수는 대상머신의 가용 레지스터의 수보다 훨씬 크기 때문에 효율적인 마이크로코드 생성을 위해서는 대상머신의 각 레지스터의 효율적인 이용이 필수적이다. 따라서 효율적인 레지스터 이용과 리던던트한 부가적인 코드를 줄임으로써 효율적인 마이크로코드를 생성하고자 HLML-C 에서는 중간언어 생성 시 각 변수를 다음과 같이 4 가지로 분류한다.

△기호변수 범용 레지스터 변수
특수 레지스터 변수
△임시변수 ALU 레지스터 변수
메모리 레지스터 변수

1) 기호변수

기호변수는 HLML-C 에서 레지스터로 선언된 변수를 나타낸다. 범용 레지스터 변수는 register 선언문으로 선언된 변수로써 코드 생성 과정 중 레지스터 할당 과정에서 대상머신의 범용 레지스터로 할당되는 변수이다. 특수 레지스터 변수는 효율적인 코드 생성을 위하여 마이크로프로그래머가 HLML-C 의 spreg 선언문으로 직접 대상머신의 레지스터를 지정하고자 할 때 사용되는 변수로 중간언어 생성시 각 변수 앞에 * 가 붙는다.

2) 임시변수

임시변수는 HLML-C 의 표현식의 수행을 위해 일시적으로 데이타의 값을 저장하기 위해 컴파일러에 의해 생성되는 변수이다.

(1) ALU 레지스터 변수

일반적으로 대부분의 아키텍처가 ALU연산 시 데이타의 값을 일시 저장하기 위해 ALU 입출력 레지스터를 제공하고 있다. 따라서 HLML-C 의 표현식 중 ALU연산을 위하여 생성된 임시변수를 직접 ALU 입출력 레지스터에 할당함으로써 범용 레지스터에 리던던트하게 억세스하는 코드를 생성하지 않고, 또 가용 레지스터의 수를 늘림으로써 효율적인 레지스터 할당을 할 수 있다. ALU 레지스터 변수는 중간언어 생성 시 $T 로써 시작되는 변수이다.

(2) 메모리 레지스터 변수

대부분의 아키텍처가 메모리 억세스 시 억세스할 데이타를 메모리 데이타 레지스터(또는 메모리 buffer 레지스터)에 억세스 할 데이타를 일시 저장한 후 메모리를 억세스한다. 따라서 HLML-C 표현식 중 메모리 억세스를 위하여 생성된 임시변수를 직접 메모리 데이타 레지스터에 할당하여 효율적인 마이크로코드를 생성하기 위하여 중간언어 생성 시 $M으로써 메모리 레지스터 변수를 구분한다.

3. 제어흐름 최적화

HLML-C 컴파일러는 HLML-C 언어의 개개의 문(statement) 단위로 조사하여 중간언어(MIIL)를 생성하기 때문에 분기문에서 다음 예와 같은 리던던트한 명령어들이 생성될 수 있다.

```
1) 10  JUMP   11
   11  SHRO   dest   src 1   src 2
2) 20  CJMPGE  src 1  src 2  22
   21  JUMP   18
   22  ADD    dest   src 1   src 2
```

1)의 예의 경우는 문번호 10의 분기문이 다른 명령어의 타겟이 되는 라벨(label) 문이 아니라면 제거할 수 있다. 2)의 예에서는 문번호 21의 분기문이 라벨문이 아니라면 문번호 20의 분기문의 조건을 역으로 하고 타겟주소를 분기문의 타겟주소로 변환한 후 문번호 21의 리던던트한 분기문을 다음과 같이 제거할 수 있다.

```
2)' 20  CJMPLT  src 1  src 2  18
    22  ADD     dest   src 1  src 2
```

〔예〕 곱셈 마이크로프로그램에 대한 MIIL 생성 예

```
0   ENTRY    multiplication
1   LOAD     *SP     &TOP
2   ADD      *TOP    *SP     3
3   LOAD     $M0     101
4   MOVE     mier    $M0
5   AND      $T0     mier    0X8000
6   LOAD     $M0     100
7   ADD      $T0     $T0     $M0
8   AND      $T0     $T0     0X8000
9   MOVE     result  $T0
10  AND      mier    mier    0X7FFF
11  LOAD     $M0     100
12  AND      $T0     $M0     0X7FFF
13  MOVE     mcand   $T0
14  CJMPEQ   mier    0       23
16  AND      $T0     mier    1
17  CJMPNE   $T0     1       20
19  ADD      result  result  mcand
20  SHR0     mier    mier    1
21  SHL0     mcand   mcand   1
22  JUMP                     14
23  STORE    result  102
24  EXIT     multiplication
```

예는 가용한 범용 레지스터의 수를 2, ALU, 메모리 레지스터가 각각 1개로 가정했을 때 곱셈 마이크로프로그램에 대해 레지스터 할당 후의 MDIL 생성 예이다. 여기서 %, #은 ALU, 메모리 레지스터를, $는 범용 레지스터를 나타내고, STORE#, LOAD#은 레지스터 할당 후에 삽입된 명령이다,

〔예〕 *곱셈 마이크로프로그램에 대한 MDIL 생성 예

```
0   ENTRY    multiplication
1   LOAD     *SP     &TOP
2   ADD      *TOP    *SP     3
3   LOAD     #1      101
4   MOVE     $1      #1
5   AND      %1      $1      0X8000
6   LOAD     #1      100
7   ADD      %1      %1      #1
8   AND      %1      %1      0X8000
9   MOVE     $2      %1
10  STORE#   $2      2
11  AND      $1      %1      0X7FFF
12  LOAD     #1      100
13  AND      %1      #1      0X7FFF
14  MOVE     $2      %1
15  CJMPEQ   $1      0       26
16  AND      %1      $1      1
17  STORE#   $1      0
18  CJMPNE   %1      1       22
19  LOAD#    $1      2
20  ADD      $1      $1      $2
21  STORE#   $1      2
22  LOAD#    $1      0
23  SHRO     $1      $1      1
24  SHLO     $2      $2      1
25  JUMP                     15
26  LOAD#    $1      2
27  STORE    $1      102
28  EXIT     multiplication
```

V. 언어비교 및 적용결과 분석

1. 언어비교

표 4는 현재까지 개발된 대표적인 머신 독립적인 언어와 HLML-C 와의 비교표이다. 표 4에 나타난 바와 같이 HLML-C 언어는 오퍼레이션 확장 기능과 다양한 변수의 표현 능력을 갖고 있으면서 레지스터 할당 및 컴팩션을 자동적으로 수행할 수 있음을 나타낸다. 몇 개의 예에 대하여 본 논문에서 적용된 레지스터 할당 알고리즘의 적용한 결과, VMPL에서 적용된 Ma의 알고리즘[7] 보다는 78%, IBM의 Kim과 Tan의 알고리즘[12] 보다는 16% 정도의 개선을 보였다.[20]

2. 적용결과 분석

4비트 비트슬라이스(bit slice) 인 Am2901 ALU[15]을 4개 연결시킨 가상의 16비트 대상머신 상에서 다음과 같이 6개의 테스트 마이크로프로그램을 작성하여 HLML-C 언어로 작성하여 HLML-C 컴파일러 및 레지스터 할당 후에 생성되는 MIIL, MDIL 중간언어를 분석한다.

△prime : 소수 계산 마이크로프로그램
△fbnc : Fibonacci 급수 계산 마이크로프로그램
△bubble : bubble sort 마이크로프로그램
△cvrt : 밀집십진문자열(packed decimal string)을 이진수로 변환하는 마이크로프로그램
△fmpy : 32비트 부동소수(가수 24 비트, 지수 8 비트)에 대한 곱셈 마이크로프로그램
△fadd : 32 비트 부동소수(가수 24 비트, 지수 8 비트)에 대한 덧셈 마이크로프로그램

표 5는 HLML-C로 작성한 각 테스트 마이크로프로그램이 중간언어인 MIIL, MDIL로 확장 생성된 결과를 보여준다. (1)의 HLML-C 실행문의 수는 마이크로프로그래머의 프로그램 스타일에 따라 조금씩 가변적이다. (2)는 register 선언문으로 선언된 범용 레지스터 변수의 수이고, (3),(4)는 제어흐름 최적화 전후에 각각 생성된 MIIL 중간언어문의 수이다. HLML-C 실행문에서 MIIL 중간언어로의 확장비율((7)) 은 수행된 연산의 복잡도를 나타내는 수치로써 대략 1.5~3.0 정도의 값을 갖는다. 이는 HLML-C 언어의 함축성 및 간결성을 간접적으로 나타낸다. (9)는 제어흐름 최적화 전후의 MIIL 중간언어의 비교치로써 제어흐름 최적화 후에 약 10% 정도로 MIIL 문의 수가 감소 되었다. (5),(6)은 대상머신의 ALU, 메모리 레지스터가 각각 0,0과 1,1이고, 가용 범용 레지스터 수를 각각 7,5,3으로 가정했을 때 레지스터 할당[19,20] 후에 생성된 MDIL 중간언어문의 수이다. (8)은 HLML-C 실행문과 MDIL 중간언어와의 확장비율을 나타내는 수치이고, (10)은 생성된 MIIL 과 MDIL 의 비교치로써 범용 레지스터 변수의 수 ((2)) 가 대상머신의 가용 범용 레지스터를 초과한 경우 레지스터 할당 후에 약 26% 정도의 부가 MDIL 문이 삽입되었다. (11)은MIIL 중간언어 상에서 ALU, 메모리 임시변수의 설정이 레지스터 할당 후의 MDIL 에 미친 영향을 보여준다. 여기서 대상머신의 가용 범용 레지스터 수가 범용 레지스터 변수의 수 보다 적은 경우(R<(2)) 에 R=5인 fadd 마이크로프로그램의 경우를 제외하고는 MIIL에서의 임시 변수 설정이 전체적으로 약 4% 정도로 MDIL에서의 부가명령이 제거되었다. 이 수치는 처음 기대한 것 보다는 적었으나 MDIL 이 최종적으로 마이

표 4. 고급 마이크로프로그래밍 언어비교
Table 4. The comparison of high level microprogramming languages.

	EMPL	S*	VMPL	Microcode C	MIMOLA	Micro-C	HLML-C
표 현 식	단 일 식	다 중 식	단 일 식	다 중 식	다 중 식	다 중 식	다 중 식
오퍼레이션 확장	가 능	가 능	불 가 능	불 가 능	가 능	불 가 능	가 능
변 수	레지스터	레지스터, 메 모 리, 스 택	레지스터 메 모 리, 스 택	레지스터	레지스터, 메 모 리	레지스터 메 모 리	레지스터, 메 모 리, 스 택
레지스터 할당	자 동	메 뉴 얼	자 동	매 뉴 얼	매 뉴 얼	매 뉴 얼	자동/매뉴얼
컴팩 팩 션	자 동	매 뉴 얼	자 동	자 동	자동/매뉴얼	자 동	자 동
개발기관, 연도	Univ. of Michigan, 1976.	Simon Fraser Univ., 1978.	Oregon State Univ. 1979.	Univ. of Waterloo, 1983.	Kiel Univ., 1984.	Burroughs Company 1985.	

표 5. 테스트 마이크로프로그램에 대한 대한 적용 결과

Table 5. The experimental results of the test microprograms.

결과 \ 마이크로프로그램		prime	fbnc	bubble	cvrt	fmpy	fadd
(1) HLML-C 실행문 수		16	13	12	21	54	89
(2) 범용레지스터변수 수		5	5	4	7	12	13
(3)** MIIL1문 수		49	23	25	38	107	187
(4)** MIIL2문 수		46	22	22	34	94	169
(5)***MDIL1문 수	*R=7	46	22	22	36	114	204
	R=5	50	22	22	45	129	212
	R=3	58	29	25	58	163	237
(6)***MDIL2문 수	R=7	46	22	22	34	112	185
	R=5	46	22	22	42	122	217
	R=3	56	29	25	51	161	233
(7) MIIL2/HLML-C((4)/(1))		2.89	1.69	1.83	1.62	1.74	1.90
(8) MDIL2/HLML-C ((6)/(1))	R=7	2.89	1.69	1.83	1.62	2.07	2.08
	R=5	2.89	1.69	1.83	2.00	2.26	2.44
	R=3	3.50	2.23	2.08	2.43	2.98	2.62
(9) MIIL1/MIIL2 ((3)/(4))		1.07	1.05	1.13	1.12	1.14	1.11
(10) MDIL2/MIIL2 ((6)/(4))	R=7	1.00	1.00	1.00	1.00	1.19	1.09
	R=5	1.00	1.00	1.00	1.24	1.30	1.28
	R=3	1.22	1.32	1.14	1.50	1.71	1.38
(11) MDIL1/MDIL2 ((5)/(6))	R=7	1.00	1.00	1.00	1.06	1.02	1.10
	R=5	1.09	1.00	1.00	1.07	1.06	0.98
	R=3	1.04	1.00	1.00	1.14	1.01	1.02

* R : 가용 범용 레지스터 수
** MIIL1 : 제어흐름 최적화 전의 MIIL,
MIIL2 : 제어흐름 최적화 후의 MIIL
***MDII 1 : ALU 레지스터, 메모리 레지스터가 각각 0인 경우를 가정한 MDIL,
MDIL2 : ALU 레지스터, 메모리 레지스터가 각각 1인 경우를 가정한 MDIL

크로오퍼레이션으로 더욱 확장될 것을 감안하면 무시할 수는 없다.

Ⅵ. 결 론

본 논문에서는 마이크로프로그래머가 프로세서 내부동작 및 하드웨어 구조를 자세히 모르더라도 적은 비용과 노력으로 마이크로프로그램을 작성할 수 있도록 일반 C 언어 형태와 유사한 구조를 갖는 머신 독립적인 고급 마이크로프로그래밍 언어 HLML-C를 개발, 제안하였다. 즉, 다양한 머신의 특성을 고려하여 일반화시킨 추상머신 상에서 HLML-C 오퍼레이션을 정의하였고 머신 독립적으로 추상머신의 중간언어로 변환하였다. 중간언어 생성 시 효율적인 마이크로코드 생성을 위하여 머신 독립성을 유지하면서 대상머신에 종속적인 특수한 오퍼레이션의 확장 적용이 가능하게 설계하였고 각 머신의 하드웨어 자원 특성을 고려하여 중간언어를 생성하였다. 또한 다양한 예의 테스트 마이크로프로그램에 대해 적용 분석하여 제안된 언어 및 컴파일러의 타당성을 입증하였다.

본 시스템은 현재 VAX-11/750(4.3 BSD) 컴퓨터 상에서 컴파일러, 레지스터 할당 과정 및 MDIL 시뮬레이터가 yacc와 C 언어를 이용하여 실현되었다. 컴파일러와 MDIL 시뮬레이터는 각각 2300, 900 스텝의 yacc와 C 언어로, 레지스터 할당 과정은 5000 스텝의 C 언어로 작성되었다.

부 록

1. HLML-C 구문

```
program        → defs
defs           → def
                 | defs def
def            → func_def
                 | global_dcl;
                 | reg_dcl;
                 | mainmem num id[num]
                 | #define id num
func_def       → microprogram id; program_body
                 | id(optional_id_list) reg_dcls program_body
global_dcl     → memory num mem_id_list
                 | spreg num spreg_id_list
                 | stack num id_list
                 | extension id_list
reg_dcl        → register num id_list
reg_dcls       → reg_dcl;
                 | reg_dcls reg_dcl;
optional_id_list → id_list
                 | ∈
id_list        → id
                 | id_list, id
mem_id_list    → mem_id: num
                 | mem_id_list, mem_id: num
mem_id         → id
                 | id ixlist
ixlist         → [num]
                 | ixlist[num]
```

```
spreg_id_list   → id : id
                  | spreg_id_list, id : id
program_body    → {reg_dcls stmt_list}
stmt_list       → stmt
                  | stmt_list stmt
stmt            → basic_stmt
                  | if (cond) stmt
                  | if (cond) restrict stmt else stmt
                  | while (cond) stmt
                  | for (for_expr ; cond ; for_expr) stmt
                  | do stmt while (cond) ;
                  | id : stmt
restrict_stmt   → basic_stmt
                  | if (cond) restrict_stmt else restrict_stmt
                  | while (cond) restrict_stmt
                  | for (assmt ; cond ; assmt) restrict_stmt
                  | do stmt while (cond) ;
for_expr        → assmt
                  | ∈
assmt           → id ass_op add_expr
                  | id sh_ass_op num ;
                  | unary_op id ;
                  | id add_expr add_expr add_expr ;
cond            → cond || cond
                  | cond && cond
                  | add_expr rel_op add_expr
                  | (cond)
                  | flag
                  | ! flag
basic_stmt      → compound_stmt
                  | assmt ;
                  | goto id ;
                  | switch(add_expr) {sw_stmt_list}
                  | id (optional_id_list) ;
                  | return ;
                  | ;
compound_stmt   → {stmt_list}
sw_stmt_list    → sw_stmt_code default_stmt
sw_stmt_code    → sw_stmt
                  | sw_stmt_code sw_stmt
default_stmt    → default : stmt
                  | ∈
sw_stmt         → case num : sw_code
sw_code         → stmt_list
                  | stmt_list break ;
add_expr        → mul_expr
                  | add_expr add_op mul_expr
mul_expr        → logic_expr
                  | mul_expr mul_op logic expr
logic_expr      → shift_expr
                  | logic_expr logic_op shift_expr
shift_expr      → term
                  | shift_expr shift_op num
                  | shift_expr cat_op term
term            → factor
                  | unary_op id
factor          → (add_expr)
                  | num
                  | id
                  | @ id
                  | memory_id
memory_id       → id[add_expr]
                  | memory_id [add_expr]
flag            → OF | UF | PS | NG | ZR | CR
```

2. 어휘분석 규칙

1) 코멘트 표시는 '/*' 와 '*/' 로표기된다.

2) 명령어(keyword)는 심볼테이블에 미리 예약(reserve) 되어 있다.

3) id와 num은 각각 다음과 같이 정의 된다.

```
letter      → 'A' | 'B' | ··· 'Z' | 'a' | ··· | 'z'
digit       → '0' | '1' | '2' | ··· | '9'
hex_prefix  → '0' 'x'
              | '0' 'x'
hex_digit   → digit
              | 'A' | 'B' | ··· | 'F'
              | 'a' | 'b' | ··· | 'f'
num         → digit digit*
              | hex_prefix hex_digit hex_digit*
id          → letter (letter | digit)*
```

4) 각 오퍼레이터의 정의는 다음과 같다.

```
rel_op      →  >= | > | <= | < | == | !=
add_op      →  + | +¦ | - | -¦
mul_op      →  * | /
logic_op    →  & | ¦ | ^ | ~& | ~¦ | ~^
shift_op    →  >> | >>¦ | << | <<¦ | >>> | <<<
cat_op      →  //
unary_op    →  ++ | -- | ~
ass_op      →  = | += | -= | +¦= | -¦=
               | *= | /=
               | &= | ¦= | ^= | ~&= | ~¦= | ~^=
shift_ass_op → >>= | >>¦= | <<= | <<¦= | >>>= | <<<=
```

參 考 文 献

[1] S. Dasdupta, "Some aspects of high-level microprogramming," Computing Surveys, vol. 12, no. 3, pp. 295-234, Sept. 1980.

[2] M. Sint, "A survey of high level microprogramming languages," *IEEE 13th Annual Microprogramming Workshop*, pp. 141-153. Nov. 1980.

[3] M. Davidson, "Progress in high-level microprogramming," *IEEE Software*, pp. 18-26, July 1986.

[4] R. Gurd, "Experience developing microcode using a high-level language," *IEEE 16th Annual Microprogramming Workshop*, pp. 179-184, Oct. 1983.

[5] D.J. DeWitt, "A machine independent approach to the production of horizontal microcode," Ph. D. thesis, Univ. of Michigan, 1976.

[6] K. Malik, "Designing a high level microprogramming language," Ph. D. thesis, Oregon State Univ., 1979.

[7] P.Y. Ma, "Optimizing the microcode produced by a high level microprograming language," Ph. D thesis, Oregon State Univ., 1979.

[8] S.R. Vegdahl, "Local code generation and compaction in optimizing microcode complieers," Ph. D. thesis, Carnegie-Mellon Univ., 1982.

[9] R.J. Sheraga and J.L. Gieser, "Experiments in automatic microcode generation," *IEEE Trans. on Computer* vol. C-32, pp. 557-569, June 1983.

[10] P. Marwedel, "A retargetable compiler for a high-level microgramming language," *IEEE 17th Annual Microprogramming Work shop*, pp. 267-274, Nov. 1984.

[11] W. Hopkins et. al., "Target-independent high-level microprogramming," *IEEE 18th Annual Microprogramming Workshop*, pp. 137-144, Dec. 1985.

[12] J. Kim, C.J. Tan, "Register assignment algrithms for optimizing micro-code compilers," IBM Research Report RC 7639 (# 33035), May 1979.

[13] R.A. Mueller et. al., "A survey of resource allocation methods in optimizing microcode compilers," *IEEE 17th Annual Microprogramming Workshop*, pp. 285-295, Nov. 1984.

[14] A.K. Agrawala, T.G. Rauscher, "*Foundations of Microprogramming,*" Academic Press, 1976.

[15] G.J. Myers, "*Digital System Design with LSI Bit-Slice Logic*", John Wiley & Sons, 1980.

[16] A.V. Aho, R. Sethi, J.D. Ullman, "*Compilers, Principles, Techniques, and Tools,*" Addison-Wesely, 1986.

[17] 이상정, 박종득, 조영일, 임인칠, "Total Execution Time 및 Contrd Memory Space의 감소를 위한 Microprogram의 광역적 최적화 기법," 한국정보과학회논문지, 제11권, 제 3 호, pp. 210-222, 1984. 8.

[18] Young-Il Cho, Sang-Jeong Lee, Jong-Deuk Park, In-Chil Lim, "A technique for global microcode compaction," *IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1051-1054, June 1985.

[19] 이상정, 임인칠, "마이크로프로그램의 레지스터 할당을 위한 변수결합 알고리즘," 대한전자공학회논문지, 제24권 제2호, pp48-55, 1987. 3.

[20] 이상정, 임인칠, "마이크로프로그램의 레지스터 할당 알고리즘," 대한전자공학회 전자계산연구회 학술발표회논문집, pp. 54-58, 1987. 9.

[21] 이상정, 조영일, 임인칠, "고급 마이크로프로그래밍 언어의 개발," 한국정보과학회 학술발표논문집, 제14권 1호, pp. 161-164, 1987. 4.