GNU 컴파일러를 이용한 ES-C2340 DSP2용 C 교차 컴파일러의 개발

이 시 영[†] · 권 육 춘[†] · 유 하 영[††] · 한 기 천[††] · 김 승 호[†]

요 약

본 논문에서는 GNU 컴파일러를 이용하여 ES-C2340 DSP2 프로세서를 위한 C 교차 컴파일러를 개발한다. 신속하고 효율적인 컴파일러의 개발을 위해 언어 의존적인 프론트 앤드(front-end)의 일부는 GNU 컴파일러를 사용하고, 프로세서 의존적인 백 앤드(back-end) 부분은 새로이 작성하여 결합하는 접근 방법을 사용한다. 이러한 접근 방법은 첫째, 프론트 앤드 부분에서 잘 검증된 GNU 컴파일러의 뛰어난 최적화 기법과 다중 언어 지원성을 사용하므로 컴파일러의 효율성과 범용성이 보장되고, 둘째, 하드웨어 의존적인 부분의 구현에만 집중함으로써 개발 기간이 단축되며, 셋째 개발 시간의 단축으로 인해 프로세서의 개발시에 논리 검증 도구로 고급 언어를 사용할 수 있게 한다. 그리고 본 논문에서는 교차 컴파일러를 지원하기 위하여 텍스트 수준의 선링커(pre-linker)도 구현한다.

The Implementation of C Cross-Compiler for ES-C2340 DSP2 by Using the GNU Compiler

Si Young Lee[†] · Yoock Choon Kwon[†] · Hah Young Yoo[††] · Ki Cheon Han[††] · Sung Ho Kim[†]

ABSTRACT

In this paper, we describe the implementation of the C cross-compiler for the ES-C2340 DSP2 processor by using the GNU compiler. For the rapid and efficient developing of the compiler, we use the approach that combines some parts of the language-dependent front-end of the GNU compiler and other parts like the processor-dependent back-end which is implemented newly to build the compiler. This approach has several advantages. First, as we use GNU compiler's well-proved excellent optimization method and multi-language support capability, we can improve the efficiency and generality of the compiler. Second, as we concentrate on the developing of processor-dependent part, it shorten the compiler developing time, third, with that, we can use high-level language as logic approving tool in processor developing process. And to support the cross-compiler, we also implement a text-level pre-linker.

1. 서 론

컴퓨터 산업에서 소프트웨어에 관련된 비용은 점차 증가하고 있다. 이는 소프트웨어의 중요성이 점차 부각됨에 따라 발생하는 현상이지만, 매년 40%에 이르는 하드웨어 성능 향상에 비해 3~8%에 그치는 소프트웨어 생산성에도 기인하고 있다[1].

† 정 회 원:경북대학교 컴퓨터 공학과
†† 정 회 원:한국전자통신연구소
논문접수:1995년 9월 23일, 심사완료:1996년 1월 12일

따라서 소프트웨어에 관련된 비용을 줄이기 위하여 개발 단계에서 water-fall model의 사용, McIlroy가 제안한 소프트웨어 재사용(software reuse) 개념[1, 2], 고급 언어 컴파일러의 제공에 따른 소프트웨어 이식[3] 등이 사용되고 있다. 이 중 소프트웨어의 이식은 기존의 소프트웨어를 목적 하드웨어 의존적인 부분만을 수정하기 때문에, 컴파일러의 개발이 선행된다면 소프트웨어의 재작성에 따른 많은 개발 시간과 비용을 절감할 수 있다.

그러나 목적 프로세서용의 고급 언어 컴파일러를 만드는 것은 많은 노력과 시간을 요구하는 작업이다. 비록 정의된 언어에 대해 스캐너와 파서의 작성을 도와주는 Lex와 Yacc같은 컴파일러 제작용 도구들이 있기는 하지만[4], 컴파일의 프론트 앤드(front-end), 즉 스캐너와 파서는 언어에 의존적인 부분일 뿐 프로세서에는 독립적인 부분임을 감안할 때, 기존 컴파일러의 모든 부분들을 새로이 작성한다는 것은 바람직하지 못하다. 더군다나 VLSI 설계 기술의 발달로 인하여 DSP(Digital Signal Processor)같은 특수 목적의 프로세서 개발이 가속화되는 현 시점에서는 새로운 프로세서를 위한 고급 언어 컴파일러를 신속하게 개발할 수 있는 방법이 필요하다.

따라서 본 논문에서는 FSF(Free Software Foundation)에서 제공하고 있는 GNU 컴파일러를 이용하여 ES-C2340 DSP2 C 교차 컴파일러를 개발한다. 컴파일러의 개발은 프론트 앤드 중에서 언어 의존적인 스캐너와 파서는 이미 잘 검증되어 있는 GNU 컴파일러를 사용하고, GNU 컴파일러의 중간 코드 생성기(intermediate code generator)를 목적 프로세서에 맞도록 수정한 후 새로이 작성한 백 앤드(back-end)와 결합하는 접근 방법을 사용한다. 이러한 접근 방법은 중간 코드로 사용되는 RTL(Register Transfer Language)의 생성 단계에서 GNU 컴파일러의 뛰어난 최적화 기법을 사용할 수 있고, 백 앤드 부분과 같이 하드웨어에 의존적인 부분의 구현에만 집중할 수 있어 단기간에 효율적인 컴파일러를 쉽게 구성할 수 있게 한다. 그리고 GNU 컴파일러의 프론트 앤드를 사용하므로 GNU에서 지원하고 있는 FORTRAN이나 C++ 같은 언어들의 범용 컴파일러로의 사용도 가능하다.

또, 본 논문에서는 ES-C2340 DSP2 프로세서의 라이브러리 지원을 위해 어셈블리어 원시 프로그램 수준에서 링크를 수행하는 선링커도 개발한다. 선링커는 교차 컴파일 결과를 어셈블하기 전에 링크를 수행하기에 운영 체제의 지원이 필요없고, 텍스트 수준의 라이브러리 사용으로 라이브러리의 추가와 확장이 용이하다.

본 논문은 다음과 같이 구성되어 있다. 제 2장에서는 구현에 사용된 GNU 컴파일러의 특징과 목적 프로세서인 ES-C2340 DSP2의 구조에 대해 기술하고, 제 3장에서는 구현된 교차 컴파일러와 링커에 대해 설명한다. 제 4장에서는 구현된 컴파일러의 사용법과 성능을 분석하고, 마지막으로 제 5장에서는 결론과 향후 개선 사항에 대해 기술한다.

2. 개 요

최근 활발히 진행되고 있는 다양한 특수 목적 프로세서들의 개발이 성공적으로 이루어지기 위해서는 개발 단계, 그리고 활용 단계에서 이들의 전반적인 성능 측정과 논리적 검증 및 활용에 사용될 수 있는 범용 고급 언어 컴파일러의 개발이 필요하다[5]. 이는 단순히 컴파일러의 개발뿐만 아니라 컴파일러를 지원하기 위한 링커와 라이브러리까지 포함해야 할 것이다. 그러나 국내에서는 TDX 교환기용 ETRI-Chill 컴파일러를 제외하고는 범용 컴파일러의 개발 같은 기초적인 기술에 대한 투자가 거의 이루어지지 않고 있고, 일부의 경우에도 대부분 프로토타입에 그쳐 실제 응용하기에는 미비한 실정이다[6].

컴파일러 개발의 일반적인 접근 방법은 분석-종합(analysis-synthesis) 모델에 따라 언어에 의존적인 프론트 앤드와 하드웨어에 의존적인 백 앤드의 각 모듈들을 순서적으로 작성하는 것이다. 이렇게 함으로써 각 모듈간에 밀집된 구성을 가지는 컴파일러를 만들 수 있다.

그러나 이러한 접근 방법은 새로이 정의된 언어에 대해서는 적절한 선택이 될 수 있으나, 이미 잘 검증된 스캐너나 파서를 가지고 있는 C 같은 언어들은 언어에 의존적인 프론트 앤드 부분을 중복하여 작성하는 결과가 된다. 비록 Lex나 Yacc 같은 컴파일러 제작용 도구들이 지원되고는 있으나 많은 시간과 노력이 중복되는 것은 피할 수 없다.

따라서 본 논문에서는 위와 같은 중복을 피하고,

ES-C2340 DSP2 [7] C 교차 컴파일러를 신속하고 효율적으로 개발하기 위하여 FSF에서 제공하고 있는 GNU 컴파일러[3, 6]를 사용하는 접근 방법을 선택한다.

본장에서는 컴파일러의 구현 설명에 앞서 구현에 사용된 GNU 컴파일러의 특징과 목적 프로세서인 ES-C2340 DSP2의 구조에 대해 설명한다.

2.1 GNU 컴파일러

본 절에서는 GNU 컴파일러의 특징, 기능별 구성과 처리 과정 그리고 중간 코드인 RTL에 대해서 간단하게 기술한다.

2.1.1 특 징

GNU 컴파일러는 FSF에서 제공하는 원시 프로그램이 공개된 컴파일러로 GPL(General Public License)을 따르고 있어 원시 프로그램의 수정, 사용 및 배포에 제약이 없는 컴파일러이다. 그리고 잘 정의된 컴파일 과정의 각 구성 단계와 공개된 원시 프로그램으로 인해 이미 많은 시스템에 이식되어 사용되고 있으며, 최적화 및 디버깅 기능이 우수하다[3, 6].

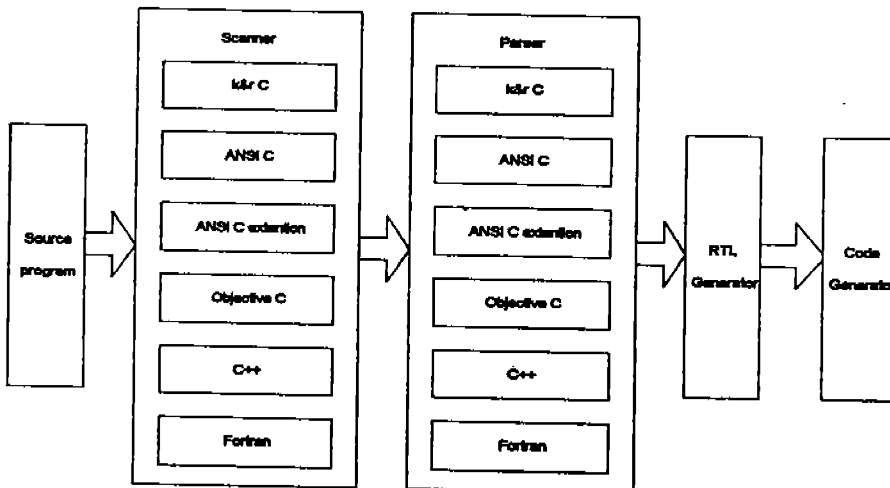
그리고 k&r C, ANSI C, ANSI C extension, Objective C, C++, FORTRAN에 대해 별도의 스캐너와 파서를 지원하지만 공통적으로 RTL을 중간 코드로 사용하므로 그림 1과 같은 범용 컴파일러로의 개발이 가능하다. 그러나 라이브러리의 지원 측면에서는 표준 헤더 파일의 지원이 되지 않기에 프로세서마다 새로 정의하거나 기존의 라이브러리를 수정해야 하는 단점이 있다.

2.1.2 구 성

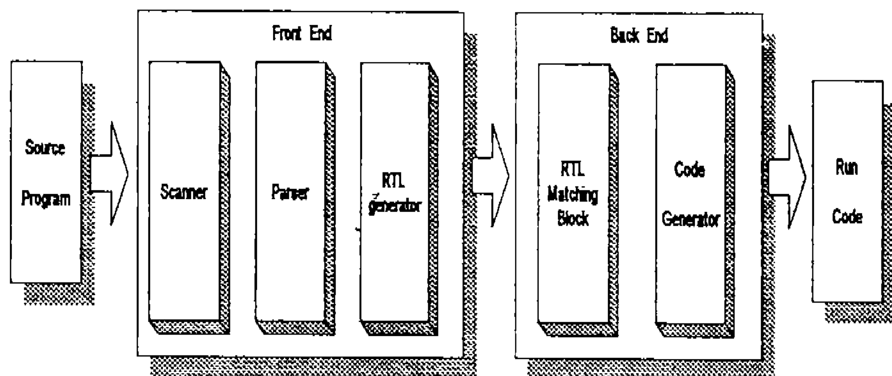
GNU 컴파일러는 그림 2와 같이 컴파일 과정의 부분 작업들을 수행하는 스캐너, 파서, RTL 생성기, 최적화기, 그리고 후위처리기로 이루어져 있다. 그리고 목적 프로세서에 의존적인 부분들은 상당 부분이 매크로로 처리되어 컴파일러의 구성 중에 기계 정의 파일(machine definition file)과 매크로 정의 파일(macro definition file), 그리고 기계 기술 파일(machine description file)에서 참조하도록 되어 있다.

2.1.3 RTL

GNU 컴파일러는 컴파일 과정에서 중간 코드를 생성하기 위해 RTL이라는 언어를 사용한다. RTL은 출력될 명령어가 가상의 프로세서(virtual processor) 상에서 수행해야 하는 일을 지시하는 대수학적 형태(algebraic form)로 기술된다. 내부 데이터는 바이트 단위로 자료를 처리하며, 바이트의 크기는 기계 정의 파일 안에 매크로의 형태로 프로세서마다 정의하도록 되어 있다.



(그림 1) GNU 컴파일러의 다중 언어 지원
(Fig. 1) Multi-language support capability of the GNU compiler



(그림 2) GNU 컴파일러의 구성
(Fig. 2) Structure of the GNU compiler

RTL의 표현은 다른 구조(structure)를 가리키는 구조들로 이루어진 내부 형태(internal form)와 기계의 기술이나 디버깅 정보의 출력시에 사용되는 문서 형태(textual form) 두 가지로 이루어져 있으며, 문서 형태는 내부 형태의 포인터를 지시하는 중첩된 괄호를 사용한다.

RTL을 이루는 구성 요소로는 정수, 확장 정수(wide integer), 문자열, 표현식을 가리키는 임의의 포인터 벡터, 그리고 이들로 형성되어 있는 RTX(RTL expression)가 있다. 실제 컴파일의 중간 결과로 생성되는 프로그램 그래프는 RTL을 이용한 RTX의 이중 연결 리스트로 구성된다.

2.2 ES-C2340 DSP2

본 절에서는 목적 프로세서인 ES-C2340 DSP2의 구조와 특징에 대해 기술한다[7].

2.2.1 기능 블록의 구조

ES-C2340 DSP2는 X와 Y, 2개의 양방향 버스를 가지고 있다. 주소 지정이 가능한 메모리는 16비트의 폭을 지니며, 내부 연산 중에는 4비트의 보조 레지스터를 더한 36비트 누산기와 16비트로 분할할 수 있는 32비트의 레지스터를 사용한다.

프로세서의 내부는 그림 3처럼 프로그램 제어, 연산, 메모리 관리, 그리고 입출력 관리 블록 등으로 구성되어 있다. 이 중에서 프로그램 제어(program control)블록은 프로그램 카운터(program counter)와 명령 레지스터(instruction register)를 중심으로 구성된다. 이 곳에서는 프로그램 카운터에 관련된 명령어들로 분기, 호출, 반복, 복귀 등의 명령을 처리한다.

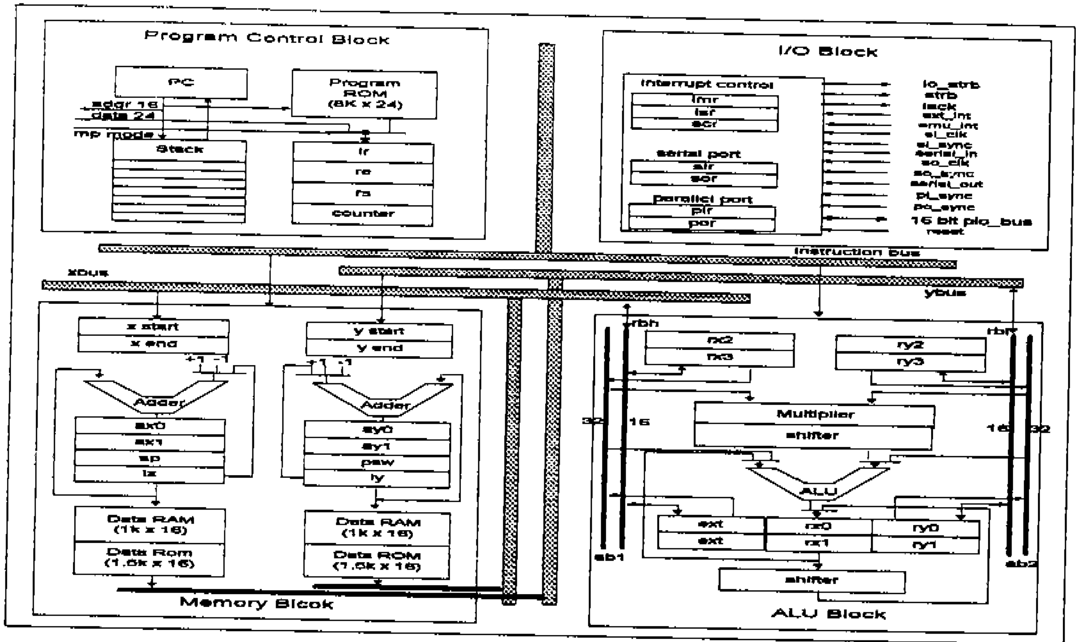
메모리 관리 블록은 직접 번지 지정 방식(direct addressing mode)과 간접 번지 지정 방식(indirect addressing mode)을 사용하여 주기억장치에 데이터를 입출력하는 기능을 수행한다. 또한, 스택을 다루는 push와 pop 명령어도 처리하며, 두 개의 16비트 데이터를 동시에 처리할 수 있다.

연산(ALU) 블록은 산술 연산과 논리 연산을 수행한다. 이 블록은 주로 즉치 번지 지정 방식(immediate addressing mode)이나 직접 번지 지정 방식을 사용하며, 두 쌍의 36비트 누산기와 X, Y 뱅크에 각각 두 쌍의 범용 레지스터를 가지고 있다.

입출력 관리 블록은 16비트 병렬 포트와 8비트 또는 16비트로 동작하는 직렬 포트로 구성된다. 또한, 인터럽트의 처리를 위해 인터럽트 설정 레지스터와 인터럽트 해제 레지스터를 포함하고 있다. 입출력 포트들은 입력과 출력을 동시에 수행할 수 있도록 두 개의 독립된 레지스터들로 구성되고, 입출력 제어를 위한 제어 레지스터도 입출력 관리 블록에 포함된다.

2.2.2 명령어 집합

ES-C2340 DSP2 프로세서의 명령어는 실수 연산을 제공하지 않으며, 모든 명령어를 단일 사이클에 실행시키는 축소 명령 집합을 제공한다. 이들 명령어 집합은 그 기능에 따라 표 1과 같이 데이터 이동형, 산



(그림 3) ES-C2340 DSP2 프로세서의 구조
(Fig. 3) Structure of the ES-C2340 DSP2 processor

<표 1> ES-C2340 DSP2 프로세서의 명령어 집합
<Table 1> Instruction set of ES-C2340 DSP2 processor

	000	001	010	011	100	101	110	111
0000	NOP	IDLE	IACK	SetPSW	ROUND			
0001	LOAD-IMMEDIATE							
0010	LOAD-DIRECT							
0011	LOAD-INDIRECT							
0100	PUSH	POP						
0101	RMOV							
0110	STORE-DIRECT							
0111	STORE-INDIRECT							
1000		para ADD	para SUB	para MUL	para UMUL	para MAC		
1001		para DADD	para DSUB		para DADDC	para DSUBB		
1010		ADD	SUB	MUL	UMUL	MAC		
1011		DADD	DSUB	SUBC	DADDC	DSUBB		
1100		OR	XOR	SHIFTL	SHIFTR	COMPL	AND	
1101		REPEAT	RETINT	RETSUB	DOR	DXOR	DAND	
1110	BR	BRZ	BRGT	BRGE	BRC	BRov	BRLT	BRLE
1111	CALL	CALLZ	CALLGT	CALLGE	CALLC	CALLov	CALLLT	CALLLE

술/논리 연산형, 분기/호출형, 특수형, 그리고 병렬형 명령어로 분류할 수 있다.

2.2.3 레지스터 정의

ES-C2340 DSP2 프로세서의 레지스터는 표 2와 같이 구성되어 있다. 36비트의 폭을 가진 두개의 누산기(R0,R1) 레지스터는 각각 4비트의 확장 레지스터와 Rx계열, Ry계열 레지스터들을 합쳐서 사용한다. PSW 레지스터의 비트 구성과 의미는 표 3과 같다.

〈표 2〉 ES-C2340 DSP2 프로세서의 레지스터
〈Table 2〉 Registers of ES-C2340 DSP2 processor

Register Group	X Reg.	Y Reg.	Double Register	Description
Data Register Group	RX0	RY0	R0=4Bit+RX0+RY0	Accumulator 0
	RX1	RY1	R1=4Bit+RX1+RY1	Accumulator 1
	RX2	RY2	R2=RX2+RY2	Operand Data Register
	RX3	RY3	R3=RX3+RY3	Operand Data Register
Address Register Group	AX0	AY0	R4=AX0+AY0	Address Resister 0
	AX1	AY1	R5=AX1+AY1	Address Resister 1
	SP	PSW	R6=SP+PSW	Stack Pointer(SP) Program Status Word(PSW)
	IX	IY	R7=IX+IY	Index Register

〈표 3〉 PSW 내용
〈Table 3〉 Contents of PSW

7	6	5	4	3	2	1	0
Y Cir	X Cir	SHL	SHR	OF	SIGN	ZERO	C

Y Cir - Y data에 대해 Circular Addressing Mode를 설정
X Cir - X data에 대해 Circular Addressing Mode를 설정
OF - Indicates Overflow
SIGN - Set when negative
ZERO - Set when zero
C - Carry
SHL,R - 00 : No shift, 01 : 2bit Shift Right, 10 : 2bit Shift Left

3. C 교차 컴파일러의 구현

본 장에서는 ES-C2340 DSP2 C 교차 컴파일러와 링커의 구현에 대해 기술한다. 먼저 3. 1절에서는 교차 컴파일러의 구현에 필요한 매크로와 이를 이용한 RTL 생성기, 백 앤드의 구현에 대해 설명하고, 3. 2절에서는 텍스트 수준에서 링크를 수행하는 선연결 링커에 대해 설명한다.

3.1 C 교차 컴파일러의 구현

이 절에서는 ES-C2340 DSP2의 C 교차 컴파일러 실제 구현에 대해 기술한다. 먼저 ES-C2340 DSP2의 특성에 맞도록 정의한 기본적인 매크로들의 종류와 용도, 그리고 정의한 내용에 대해 설명을 하고, 정의한 매크로들을 통합하여 기계 정의 파일, 매크로 정의 파일, 그리고 기계 기술 파일들을 중심으로 RTL 생성기와 백 앤드의 구현에 대해 설명한다.

3.1.1 레지스터와 레지스터 클래스

ES-C2340 DSP2에서 사용하는 레지스터는 32비트 레지스터 8개와 이들을 상위와 하위로 나눈 16비트 레지스터 16개를 합쳐 모두 24개이다. 그러나 실제로는 두 개의 16비트 레지스터가 하나의 32비트 레지스터를 이루게 되므로 16비트 레지스터들을 기준으로 중복이 없게 15개로 정의한다. PSW 레지스터는 레지스터 할당에 사용할 수 없기 때문에 레지스터 정의에서 제외한다. 출력되는 레지스터의 이름이 32비트로 처리될 때에는 별도의 레지스터 이름을 사용한다.

```
#define REG_R0 1
#define REG_RY0 2
        ...
#define REG_IY 15
```

ES-C2340 DSP2에서 일반적인 용도로 사용할 수 있는 레지스터의 수는 제한적이다. 이러한 구조는 프로세서 구조를 간단히 하여 컴파일러를 구성하기 쉽게 하지만, 컴파일할 때 값의 로딩이나, 임시 계산값의 저장 등의 일반적인 목적으로 사용할 수 있는 레지스터의 수를 제한하기에 스택을 사용한 레지스터의 저장과 복귀가 필요하다. 그렇기 때문에 컴파일러의 구현에서는 컴파일 과정에서 임의로 사용할 수 있는 가상 레지스터 그룹을 설정하여 사용하고, 실제 어셈블리어로 출력될 때는 REG_IY 이하의 레지스터로 값을 옮겨 출력하는 방법으로 제한된 레지스터 수에 따른 문제를 해결한다.

REG_BASExx 레지스터들은 메모리 레지스터로서 실제 레지스터가 아니라 0h번지에서 1Fh번지까지를 레지스터처럼 사용하고 있는 것으로 함수의 호출이나 과다한 변수의 사용이 일어날 때 임시적으로 레지

스터의 값을 저장하는 데 사용된다.

```
#define REG_TMP 16
#define REG_BASE0 26
        ...
#define REG_BASE31 57
```

레지스터 클래스는 변수가 레지스터로 적재될 때에 사용할 수 있는 레지스터들의 집합을 나타낸다. 이것은 레지스터에 저장된 변수의 내용이 사용되는 용도나 연산에 따라 달라지게 되고, 이에 따라서 변수를 레지스터에 적재할 때에 할당할 수 있는 레지스터들이 제한된다.

아래는 구현에 사용된 레지스터 클래스의 종류와 심벌, 그리고 그 의미에 대해 설명을 한다. 레지스터 클래스 RX0_REG 부터 IY_REG까지는 QImode (quarter integer mode), 또는 QFmode(quarter floating mode)의 값을 저장할 수 있는 16비트 크기의 레지스터 클래스들이다.

j: RX0_REG
k: RY0_REG
q: RX1_REG
h: ACCU_HIGH_REGS
u: RY1_REG
b: ACCU_LOW_REGS
y: RX2_REG
z: RY2_REG
v: RX3_REG
w: RY3_REG
e: ADDR_HALF_REGS

레지스터 클래스 e는 간접 번지 지정에 사용할 수 있는 어드레스 레지스터 AX0, AX1, AY0, AY1을 나타낸다. 그러나 변수들은 모두 X 뱅크 메모리에 저장되기 때문에 변수에 대한 간접 번지 지정이 가능한 레지스터는 AX0, AX1 뿐이고, AY0과 AY1을 간접 번지 지정에 사용하려면 별도의 처리가 필요하다. 이에 대한 것은 레지스터의 기능별 분류에서 더 자세히 설명한다.

x: IX_REG

ACCU_REGS 부터는 HImode(half integer mode), 또는 HFmode(half floating mode)의 값을 저장할 수 있는 32비트 크기의 레지스터 클래스들이다.

A: ACCU_REGS
f: R2_REG
t: R3_REG
Z: R2_R3_REGS
d: ACCU_R2_R3_REGS

3.1.2 제약(constraints)

제약은 기계 기술 파일의 define_insn이나 define_expand 등의 RTX에서 적용할 수 있는 모드의 제한을 나타낸다. 제약도 역시 부분적으로는 특정한 레지스터들을 포함하고 있지만 레지스터 클래스처럼 세부적이지 않고, 또한 직접 지정해 줄 필요도 없다.

m: 메모리
o: 메모리에서 오프셋으로 사용이 가능한 레지스터로 로딩
V: 메모리에서 오프셋으로 사용이 불가능한 레지스터로 로딩
<: 주소 자동 증가 레지스터
>: 주소 자동 감소 레지스터
r: 범용 레지스터
i: 즉치 형식의 정수 값을 저장할 수 있는 레지스터
=: 연산 중 읽기 전용으로 사용되는 레지스터
+: 연산 중 읽기/쓰기 둘 다 허용되는 레지스터
&: 연산이 끝나기 전에 자신의 값이 훼손되는 레지스터
%: Commutative operands

3.1.3 프린트 오퍼랜드(print operand)

프린트 오퍼랜드 부분은 실제의 어셈블리어 코드 출력에 관련된 부분으로, 기계 기술 파일의 define_insn에 나타난 제약들의 선택(alternative)에서 출력의 대상이 되는 레지스터를 지정한다.

(1) User Print Operands
%H: Lower 16bits of Constraints
%U: Upper 16bits of Constraints

%h : Decimal Constraints
%w : Lower half of register
%u : Upper half of register
%b : High of accumulator
%V : Double register
%m : HI mode register
(2) General Output Operands
%C : Constant value of immediate operand
%n : Negated constant value of immediate operand
%a : Memory reference operand
%l : Label reference into jump instruction

3.1.4 기능별 레지스터 분류

이 부분은 구현에 사용된 레지스터의 기능별 분류이다. 기계 정의 파일에서 레지스터들의 용도를 정의하기 위한 부분으로, 수행하는 일에 따라 사용이 가능한 레지스터들을 정의하고 있다. 레지스터의 수가 실제 필요한 수보다 적기 때문에 IX와 IY같은 레지스터들은 용도가 중복되어 있다.

(1) Data arithmetic unit
:RX0, RY0, RX1, RY1, RX2, RY2, RX3, RY3, IX
(2) Address arithmetic unit
:AX0, AY0, AX1, AY1, IY

여기서 AY1 레지스터는 프레임 포인터(frame pointer)로도 사용이 되기 때문에 아무 처리도 하지 않고 간접 번지 계산에 사용하면 프로시쥬어의 프레임 포인터 값을 잃어버리게 된다. 따라서 어셈블리어 코드 출력시 번지 연산에 AY1 레지스터를 사용하기 위해서는 저장하고 있는 값을 스택에 저장, 복귀하는 부분이 필요하다. 그리고 AY0, AY1 레지스터들은 ES-C2340 DSP2의 메모리 특성상 X 뱅크 메모리의 간접 엑세스가 불가능하여 X 뱅크 메모리에 저장되어 있는 변수들의 엑세스가 한 번에 2개로 제한되는 문제가 발생한다. 따라서 AY0, AY1 레지스터들이 X 뱅크 메모리 엑세스에 사용될 때는, 먼저 AX0, AX1 레지스터 중의 하나를 스택에 저장하고, 저장한 레지스터로 값을 이동시켜 사용할 수 있게 하는 어셈블리어 처리 루틴을 추가하여 문제를 해결한다.

(3) Return value from function
:R0
(4) Return value from function for structure
:AX0
(5) Stack pointer register
:SP
(6) Frame pointer register
:AY1
(7) Return address register
:IY
(8) Static chain register
:IX

3.1.5 백 앤드의 구현

컴파일러를 구성할 때 매크로 중에서 프로세서에 관한 하드웨어적인 정의는 기계 정의 파일에서, 백 앤드 구성 시에 필요한 백 앤드 내부 함수의 정의는 매크로 정의 파일에서, 그리고 어셈블리어 코드에 관한 정보는 기계 기술 파일에서 참조한다. 출력되는 어셈블리어 코드와 밀접한 관계가 있는 기계 정의 파일은 목적 코드의 명령어 집합을 기술하는 매크로 형식으로 정의한다.

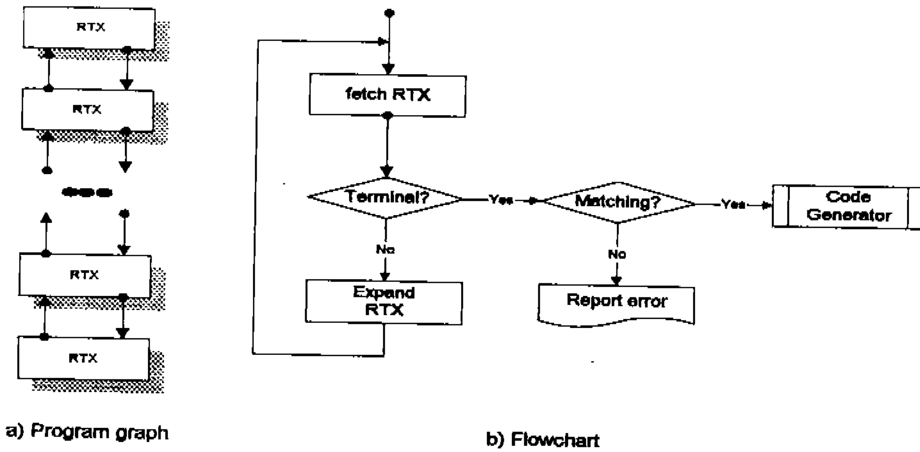
백 앤드는 기능 면에서 RTL 정합 블록과 코드 생성기로 나누어진다. RTL 정합 블록은 그림 4와 같이 기계 정의 파일에서 추출된 정보를 가지고 프론트 앤드에서 생성된 프로그램 그래프를 순회하면서 적절한 RTX 패턴을 찾는다. 이렇게 찾아진 RTX 패턴은 코드 생성기를 통하여 필요한 어셈블리어 코드로 출력이 된다.

이 절에서는 컴파일러의 RTL 생성기 및 백 앤드를 구성하기 위하여 구현한 기계 정의 파일, 매크로 기술 파일, 기계 기술 파일들에 대해 기술하고 구현된 내용에 대해 설명한다.

3.1.5.1 기계 정의 파일

기계 정의 파일은 ES-C2340 DSP2의 하드웨어적 구조를 정의하는 부분으로 컴파일러의 구성시 가장 기본적인 역할을 하는 파일이다. 따라서 이 파일에는 3.1.1절부터 3.1.4절까지 설명한 부분과 같이 각 단위 모드(mode)당 비트수, 스택 구성에 관련된 사항, 포인터의 처리 및 크기, 레지스터의 할당, 컴파일 옵션의 처리 등과 같은 정보들이 매크로로 정의된다. 기계 정의 파일에 정의되어 있는 부분은 크게 다음과 같다.

(1) Internal Structure



a) Program graph

b) Flowchart

(그림 4) RTL 정합 블록
(Fig. 4) RTL matching block

실제로 출력되는 목적 코드와는 관계가 없으나, 교차 컴파일러의 백 앤드 구성에 필요한 내부 자료구조와 내부 함수 등을 정의한다.

(2) Storage Layout

ES-C2340 DSP2의 내부 처리 자료의 크기, 메모리의 구조, 자료의 메모리 저장 방법, 그리고 포인터에 관련된 내용들을 정의한다.

(3) Register Usage

ES-C2340 DSP2의 레지스터에 관련된 부분들을 정의한다. 즉, 레지스터의 정의, 용도, 어셈블리어 출력시 가상 레지스터(pseudo register)를 실제 레지스터(hard register)에 할당하는 규칙, 그리고 레지스터 클래스의 정의와 포함되는 레지스터들을 정의한다. 또한 어셈블리어 코드 출력에 사용되는 16비트와 32비트 레지스터들의 이름도 정의한다.

(4) Stack Layout

프로세서에서 사용하고 있는 스택의 동작, 성장 방향을 정의한다.

(5) Calling convention

함수의 호출과 복귀시 수행되어야 하는 프롤로그(prologue), 에필로그(epilogue)에 대한 함수 선언, 내부 함수 호출 루틴 등을 정의한다.

(6) Addressing mode

ES-C2340 DSP2에서 사용이 가능한 번지 지정 방식의 정의와 특정한 번지 지정 방식을 사용할 수 있는 어드레스 레지스터들을 정의한다. ES-C2340 DSP2에서는 즉치 번지 지정 방식, 직접 번지 지정 방식, 간접 번지 지정 방식 그리고 원형 번지 지정 방식을 사용할 수 있지만 구현된 컴파일러에서는 원형 번지 지정 방식(circular addressing mode)은 지원하지 않는다.

3.1.5.2 매크로 정의 파일

매크로 정의 파일은 기계 정의 파일이나 기계 기술 파일에서 사용하는 내부 함수나 자료구조에 대한 정의를 가지고 있다. 컴파일러를 구성할 때 프로세서 의존적인 자료의 처리나 작업이 필요할 경우 그 일을 수행하는 함수들을 작성하여 매크로 정의 파일에 두고, 기계 정의 파일이나 기계 기술 파일에서 사용하게 된다. 아래에는 구현된 주요 내부 함수와 그에 대한 설명을 한다.

```
int hard_regno_mode_ok (regno, mode) int regno;
enum machine_mode mode;
```

hard_regno_mode_ok 함수는 각 모드에 대해 사용이 가능한 레지스터들을 결정한다. 즉, 인수로 넘어온 레지스터의 번호가 인수로 넘어온 모드에서 사용이 가능하면 1, 가능하지 않으면 0을 돌려줌으로서 레지스터의 할당 시에 기계 정의 파일에 정의된 레지스터의 할당 순서대로 레지스터를 검사하면서 특정 모드에서 사용 가능한 첫 번째 비어있는 레지스터를 검색

할 수 있게 한다.

int regno_reg_class(regno) int regno;

regno_reg_class 함수는 레지스터 번호가 인수 regno인 레지스터가 포함되어 있는 가장 작은 레지스터 클래스를 돌려주는 함수이다. 레지스터 클래스의 크기 순서는 기계 정의 파일의 레지스터 클래스 정의에서 나타난 순서에 따른다.

enum reg_class limit_reload_class (mode, class) enum machine_mode mode;enum reg_class class;

limit_reload_class 함수는 가상 레지스터가 하드 레지스터로 할당될 때 사용할 수 있는 레지스터 클래스를 결정한다.

long compute_frame_size (size) int size;

compute_frame_size 함수는 함수의 호출이 일어날 경우 호출 당한 함수의 프레임 크기를 결정하는 함수이다. 이 함수는 프로그램 내에서 정의된 사용자 정의 함수의 컴파일 시에만 적용되는 부분이기에 내부 함수나 외부함수를 지원하는 경우에는 라이브러리의 작성자가 따로 계산을 하여 라이브러리 파일 안에 포함해야 한다.

void function_prologue (file, size) FILE *file;int size;
void function_epilogue (file, size) FILE *file;int size;

function_prologue 함수는 사용자 정의 함수 루틴 첫머리에 호출 준비에 필요한 어셈블리어 루틴을 출력하는 함수로서 함수의 프레임 크기 계산, 레지스터 저장, 스택 포인터 증가, 프레임 포인터 변경 등을 담당하는 함수이다. 사용자 정의 함수의 끝 부분에는 function_epilogue 함수가 호출되어 function_prologue 함수와는 반대의 역할을 수행한다. ES-C2340 DSP2 C 교차 컴파일러에서는 0h번지부터 32개의 가상 메모리 레지스터들을 사용하기에 _main 함수의 프롤로그에는 스택 포인터의 초기값을 20h로 설정하는 부분이 추가되어 있다.

void double_reg_from_memory (operands) rtx operands[];
void double_reg_to_memory (operands) rtx operands[];

ES-C2340 DSP2는 메모리로의 접근이 스택일 때만 32비트를 허용하고, 나머지는 16비트를 기본으로 일어난다. 따라서 32비트 값인 HImode나, HFmode를 일반 변수의 값으로 메모리에 저장하거나 메모리에서 적재할 때는 별도의 루틴이 필요하게 된다. 위의 두 함수는 32비트 레지스터를 X 뱅크, Y 뱅크 메모리에 저장하고, 뱅크로 부터 적재할 때 처리를 담당한다.

void print_operand(file, op, letter) FILE *file;rtx op;
int letter;

print_operand 함수는 어셈블리어 코드 출력에 관계하는 함수로 비교 연산자나 레지스터에 해당하는 문자를 출력한다.

void ay0_reg_from_memory (operands) rtx operands[];
void ay0_reg_to_memory (operands) rtx operands[];

ES-C2340 DSP2에서는 X, Y 뱅크 메모리에 간접번지를 지정할 수 있는 어드레스 레지스터를 각각 두 개씩 제공하고 있으나, 뱅크당 두 개로는 부족하므로 Y 뱅크 메모리 어드레스 레지스터 중에서 프레임 레지스터인 AY1을 제외한 AY0를 X 뱅크 메모리 간접번지 지정에 사용할 수 있도록 어셈블리어 코드를 출력해 주는 함수이다.

void asm_output_float (file, fp_const) FILE *file;
double fp_const;

asm_output_float 함수는 부동 소수점 값을 출력하는 함수로 실제의 출력은 32비트형의 정수값으로 이루어진다.

struct rtx_def *dsp_function_arg (args_so_far, mode, type, named)
CUMULATIVE_ARGS args_so_far;
enum machine_mode mode;tree type;int named;
void dsp_function_arg_advance (cum, mode, type, named)
CUMULATIVE_ARGS *cum;enum machine_mode mode;tree type;int named;

위의 두 함수는 함수 호출시 인수의 전달에 관한 내용을 담고 있다. 함수의 호출시 인수의 전달은 레지스터를 통해 전달하는 방법과 스택을 사용하여 전달하는 두 가지의 방법이 있다. 이 중에서 레지스터

를 통해 인수를 전달하는 방법은 인수의 빠른 참조를 보장하기에 함수의 호출과 복귀시 수행이 빠르지만, 충분한 수의 레지스터가 하드웨어로 구현되어 있어야 되기 때문에 구현의 비용이 커진다. 이에 비해 스택을 이용한 인수의 전달은 기존의 메모리를 사용하기에 별도의 비용이 필요하지는 않지만 인수의 메모리 엑세스 때문에 함수의 호출과 복귀의 속도가 느려지게 된다.

하지만 실제의 함수 호출시 인수의 수는 보통 두 개를 넘지 않기 때문에 본 컴파일러에서는 인수의 수가 두 개보다 적을 때는 레지스터를 사용하고, 세 개보다 많을 때는 레지스터와 스택을 병용하는 기법을 사용하여 별도의 비용 증가 없이 빠른 함수의 수행과 복귀를 보장한다.

```
char * output_block_move (operands) rtx operands[ ];
```

ES-C2340 DSP2의 어셈블러어는 쉬프트(shift)나 로테이트(rotate) 연산에서 즉치값 만을 지원한다. 하지만 컴파일의 결과로 형성되는 쉬프트나 로테이트 연산은 레지스터에 저장된 값을 사용하는 연산이 많으므로, ouput_block_move 함수를 사용하여 어셈블리어 코드 출력시에 레지스터를 사용한 쉬프트나 로테이트 연산을 즉치값을 사용한 블록 루프문의 형태로 출력한다.

3.1.5.3 기계 기술 파일

기계 기술 파일은 백 앤드의 핵심이 되는 부분으로 RTX를 해당 어셈블리어로 바꾸어 출력할 때 필요한 명령어 패턴을 제공한다. 어셈블리어 코드의 출력은 먼저 define_expand 표현식을 사용하여 프론트 앤드에서 생성된 RTX를 확장한 후, define_insn 표현식을 사용하여 확장된 표현식 중 패턴이 일치하는 부분을 아래의 예와 같이 어셈블리어 코드로 출력한다. 아래에 보인 예는 두 개의 QImoide 덧셈 연산의 명령어 패턴을 정의하는 부분이다.

```
(define_expand "addqi3"
  [(parallel [(set (match_operand:QI 0 "register_
  operand" "")
  (plus:QI (match_operand:QI 1 "register_operand" "")
  (match_operand:QI 2 "nonmemory_operand" "")))
   (clobber (match_scratch:QI 3 ""))])]
 ""
 "
 {
 if (reload_in_progress)
   { … 생략 … }
 }")
(define_insn "match_addqi3"
  [(set (match_operand:QI 0 "register_operand"
  "=!a, !a, k, u, !k, !u, h, !a")
      (plus:QI (match_operand:QI 1 "register_operand
      "0, 0, uk, uk, uk, uk, h, 0")
          (match_operand:QI 2 "nonmemory_operand"
          "W, N, wzJ, wzJ, uk, uk, J, n")))
      (clobber (match_scratch:QI 3
      "=X, X, j, q, j, q, X, W"))]
  ""
  "*
{switch (which_alternative)
  {
    case 0:
      return \"m[sp]=r0;\;ry0=%2;\;rx0=%0;\;r0
      =rx0+ry0;\;%0=ry0;\;r0=m[sp];\";
    case 1:
      switch (INTVAL (operands[2]))
        {
        case -1:
          return \"m[sp]= r0;\;ry0= -h1;\;rx0
          =%0;\;r0=rx0-ry0;\;%0=ry0;\;r0
          =m[sp];\";
          … 생략 …
        }
      case 7:
        return \"m[sp]=r0;\;ry0=%2;\;rx0
        =%0;\;r0=rx0+ry0;\;%0=ry0;\;r0
        =m[sp];\";
    }
}")
```

ES-C2340 DSP2 C 교차 컴파일러에서는 아래와 같은 순서로 위에서 보인 예처럼 명령어 패턴들을 정의

한다.

(1)명령어 속성 정의

이 부분은 명령어의 종류, 길이, 수행 시간 등의 속성을 정의해 주는 부분이다. 컴파일러는 수행 중에 이 부분에서 정의한 자료를 토대로 하여 해당 속성의 어셈블리어 코드를 출력해 낸다.

(2)제로 비교 명령어 패턴 정의

이 부분은 특정 레지스터의 값이 0인가를 검사하는 테스트 명령어들의 패턴들을 정의한다. 실제로 ES-C2340 DSP2에서는 테스트 명령어들을 제공하지 않기 때문에 뺄셈 연산을 이용하여 정의한다.

(3)비트 테스트 패턴 정의

이 부분은 비트 단위 논리 명령어 패턴을 정의한다.

(4)비교 명령어 패턴 정의

비교 명령어 정의 부분은 두 개의 레지스터 값을 비교하는 명령어의 패턴들을 정의한다. ES-C2340 DSP2에서는 비교 명령어 구문이 없다. 따라서 비교 명령어 패턴에 정합(match)하는 어셈블리어 코드는 블록의 형태로 비교 명령을 수행할 수 있는 다수의 어셈블리어 코드로 정의한다.

(5)사칙 연산 명령어 패턴 정의

이 부분에서는 각 모드별 사칙연산 명령어 패턴들을 정의한다.

(6)논리 연산 명령어 패턴 정의

이 부분에서는 논리 연산과 관련된 명령어 패턴들을 정의한다.

(7)이동 명령어 패턴 정의

이동 명령어는 레지스터간의 이동뿐만 아니라, 레지스터에서 메모리, 메모리에서 레지스터로의 이동도 포함하여 정의한다.

(8)형변환 명령어 패턴 정의

이 부분에서는 연산중 형변환에 사용될 수 있는 모드 간의 변환 규칙을 정의한다.

(9)쉬프트 명령어 패턴 정의

ES-C2340 DSP2의 어셈블리어에서는 즉치값만이 허용되므로 대부분의 쉬프트 명령어는 여러 개의 즉치값의 쉬프트 명령어 그룹으로 변경되도록 정의한다.

(10)분기 명령어 패턴 정의

이 부분에서 정의된 분기 명령어들은 ES-C2340 DSP2 어셈블러에서 지연 분기를 처리해 주기 때문에 모두 일반 분기 명령어들로 정의한다.

(11)호출 명령어 패턴 정의

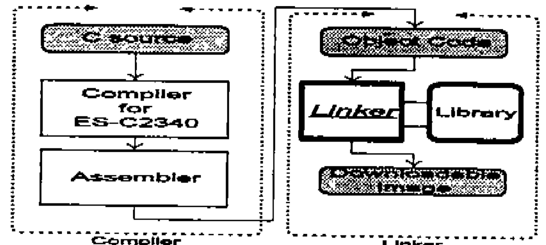
이 부분에서는 호출 명령어를 정의한다. 분기 명령어와 마찬가지로 지연 호출이 아닌 일반 호출 명령어로서 지연 호출에 대한 부분은 ES-C2340 DSP2 어셈블러를 고려하여 생략한다.

3.2 링커의 구현

본 절에서는 구현된 컴파일러를 지원하기 위해 작성된 링커에 대해 기술한다. ES-C2340 DSP2를 지원하기 위해 구현된 링커는 일반적인 링커와는 달리 어셈블리어 코드 수준에서 링크를 수행하는 선링커(pre-linker)이다.

3.2.1 개 요

일반적인 고급 언어는 컴파일 후 목적 코드 수준에서 그림 5와 같이 링크를 수행한다. 이러한 링킹 기법은 동일한 프로세서의 시스템들에 공통으로 사용할 수 있는 라이브러리를 제공함으로써, 컴파일러의 범용성을 높이고, 소프트웨어의 이식성을 높여 준다. 그러나 이러한 방식은 범용의 컴파일러에서는 적합한 방식이라고 할 수 있으나, ES-C2340 DSP2 같이 적용 함수의 수가 적고, 그 운영 체제나 기타 소프트웨어 도구들이 지원되지 않는 환경에는 적합하지 않다. 따라서 본 논문에서는 다음 절과 같은 선연결 링커를 개발한다.

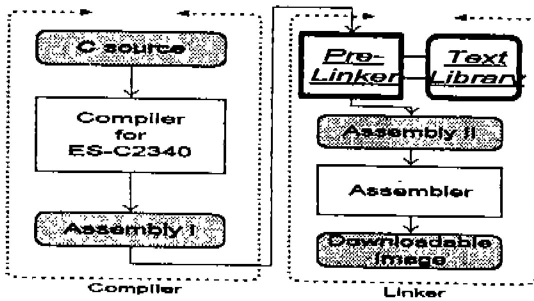


(그림 5) 일반적인 이진 링커
(Fig. 5) General binary linker

3.2.2 선연결 링커

ES-C2340 DSP2 링커의 개발에서는 그림 6과 같이 미리 어셈블되어 있는 라이브러리 코드를 어셈블 단계 이전에 병합해주는 방식을 사용한다. 이러한 접근

방법은 사용 함수가 많고, 큰 응용 프로그램을 작성하는 데는 적합하지 않지만, DSP에서 사용하는 응용 프로그램이 대부분 8KB 이하이고 사용 함수의 수가 적다는 것을 고려할 때는 효율적이라 할 수 있다[7]. 또한 라이브러리를 어셈블리어 원시 프로그램 수준에서 지원함으로써, 용도에 따른 함수 변경이 용이하고, 추가 및 확장이 쉽게 이루어진다.



(그림 6) 선링커
(Fig. 6) Pre-linker

4. 운용 및 분석

이 장에서는 구현된 ES-C2340 DSP2 C 교차 컴파일러를 이용하여 C 원시 프로그램을 컴파일한 예와 컴파일러의 성능 분석에 대해 논한다. GNU 컴파일러는 GUN 2.7.0 개정판을 사용하고, 컴파일러의 성능 분석은 리눅스(Linux) 운영 체제를 사용하는 486PC에서 같은 DSP 프로세서인 AT&T DSP1601의 C 교차 컴파일러를 사용하여 비교한다. 비교 내용은 동일한 조건에서 각각 사용자 정의 함수, 내부 함수 그리고 외부 함수의 호출을 가진 C 언어 원시 파일의 컴파일 결과로 나타낸다.

4.1 운용

구현된 교차 컴파일러를 이용하여 C 언어 원시 파일을 컴파일 하는 데는 출력되는 결과에 따라 두 가지의 옵션이 제공된다. 컴파일의 중간 과정 생성물인 RTL 표현식으로 컴파일하기 위한 -v -dr 옵션과, 어셈블리어 코드를 얻기 위한 -S 옵션이다.

xgcc -v -dr source_c_file : source_c_file에 확장자 .rtl이 추가된 파일이 새로 생성

xgcc -S source_c_file : .c의 확장자가 .s로 바뀐 파일이 새로 생성

교차 컴파일러는 어셈블리어 코드 생성 단계까지만 컴파일을 수행하기 때문에 내부 함수나 외부 함수는 링커를 통하여 원시 파일 루틴을 추가하여야 한다. 링커를 사용하는 방법은 아래와 같다.

- lndsp filename.s

filename.s파일에 내부 함수와 외부 함수의 어셈블리어 원시 파일 루틴을 추가

새로운 라이브러리의 추가는 제공되는 dsplib 디렉토리에 지원함수에 해당하는 원시 프로그램 파일을 작성한 후, 같은 디렉토리에 있는 mk_table 실행 파일을 수행하면 된다. 그러면 mk_table 프로그램이 추가된 함수 이름을 링크 시에 사용하는 표에 자동으로 등록하여 준다.

- mk_table

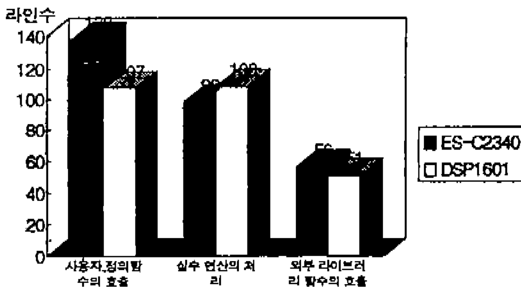
4.2 분 석

구현된 교차 컴파일러의 성능 평가를 위해 기존의 교차 컴파일러 중의 하나인 AT&T의 DSP1601 C 교차 컴파일러를 사용하였다. DSP1601 프로세서는 ES-C2340 DSP2 프로세서와 같은 단일 사이클 축소 명령 집합을 사용하고, 실수 연산을 지원하지 않기 때문에 교차 컴파일러에서 내부 함수의 호출 형태로 이를 처리해야 하는 공통점을 가지고 있다.

성능 분석에 사용한 C 언어 원시 프로그램들은 DSP1601에 비해 1/3 정도의 레지스터 수를 가지는 ES-C2340 DSP2의 특성을 고려하여 과다한 레지스터의 사용이 일어나지 않는 범위 내에서 작성되었다. 사용된 예제 프로그램들은 프로그램 내의 함수 호출에 대한 스택과 프레임의 처리 부분을 비교하기 위해 한 개의 사용자 정의 함수와 이에 대한 호출을 가지고 있는 프로그램, 시스템 내부 함수의 호출로 처리되는 실수 연산의 처리 부분을 비교하기 위해 사칙연산에 해당하는 네 개의 실수 연산과 한 개의 형변환 연산을 포함하고 있는 프로그램, 그리고 외부 라이브

러리의 호출로 처리되는 외부 함수의 처리 부분을 비교하기 위해 두 개의 외부 함수의 호출을 가지고 있는 프로그램들을 사용하였고, 교차 컴파일러의 최적화율과 효율성을 비교하기 위해 각 부분에 대한 어셈블리어 출력 결과를 비교하였다.

각 예제 프로그램의 교차 컴파일 결과의 출력 어셈블리어 라인 수는 그림 7에 나타내었다. 비록 DSP1601 프로세서가 명령어의 포맷은 다르지만 같은 DSP로서 비슷한 수준의 명령어를 제공하고 있는 것을 감안할 때, 구현된 교차 컴파일러는 어셈블리어 코드의 출력 결과에서 실수 연산의 처리와 외부 함수의 호출 처리 부분에 있어서는 DSP1601 C 교차 컴파일러와 거의 유사한 성능을 보임을 알 수 있었다. 사용자 정의 함수와 이에 대한 호출을 포함한 프로그램에서는 27% 정도 더 많은 어셈블리어 코드 출력을 보이고는 있으나 이는 교차 컴파일러의 효율성 문제가 아니라 함수의 호출과 복귀의 처리시에 많은 수의 레지스터 사용이 일어나 상대적으로 적은 수의 레지스터를 가지는 ES-C2340 DSP2에서는 스택에 레지스터의 값을 저장, 복구하는 과정이 어셈블리어로 출력되어야 하기 때문이었다.



(그림 7) 컴파일 결과 비교
(Fig. 7) Comparing of the compile results

5. 결론 및 향후 개선 사항

본 논문에서는 FSF의 GNU 컴파일러를 이용하여 ES-C2340 DSP2 C 교차 컴파일러를 개발하였다. 개발된 컴파일러는 원시 프로그램이 공개되어 있는 GNU 컴파일러의 프론트 앤드 부분 중에서 언어 의존적인 부분을 사용하고, 프로세서 의존적인 부분들은 새로이 작성하는 접근 방법을 선택하여 신속한 교차 컴파일러의 구성을 시도하였다.

이와 같은 접근은 첫째 스캐너나 파서 같이 이미 잘 검증되어 있는 프론트 앤드 부분에 대한 중복된 작업이 없어 개발 시간과 비용이 절약되고, 둘째 이에 따라 프로세서의 개발시 논리 검증의 도구로 C 같은 고급 언어를 사용할 수 있으며, 세째 명확한 컴파일 단계의 모듈화로 프로세서의 구조 변경에도 유연하게 대처할 수 있다. 또한 교차 컴파일러이므로 프로세서를 위한 별도의 운영 체제의 제공이 힘든 상황에서 유닉스(Unix)나 리눅스의 환경에서 컴파일만 수행하여 실행 코드를 얻을 수 있다.

그리고 컴파일러를 지원하기 위해 개발한 링커는 별도의 운영 체제가 없으므로 일반적인 링크 방법 대신, 어셈블리어 코드 차원에서 텍스트 링크를 수행하도록 개발하였다. 이는 운영체제가 없는 작은 시스템에 적용하기 좋고, 다양한 라이브러리가 필요하지 않는 특수 목적의 프로세서에서 필요한 라이브러리의 수정, 추가, 삭제가 용이하다.

그러나 본 논문에서 개발한 컴파일러는 하나의 메모리 뱅크를 지원하는 GNU 컴파일러의 프론트 앤드를 사용하기 때문에 ES-C2340 DSP2의 두 개의 메모리 뱅크를 균형적으로 활용하기가 힘들고, ES-C2340 DSP2의 특징인 병렬 명령어에 대한 처리가 힘들다. 따라서 앞으로 컴파일러의 성능 향상을 위하여 균등한 X, Y 메모리 뱅크 사용과 병렬 명령어의 지원에 대한 연구가 계속되어야 할 것이다.

참 고 문 헌

[1] 홍의경, 송영기, 최완, "소프트웨어 재사용을 위한 라이브러리 관리 시스템," 한국정보과학회 논문지, Vol. 20, No. 4, pp. 538~549, 1993.

[2] E. Horowitz and J. B. Munson, "An Expansive View of Reusable Software," IEEE Trans. Software Engineering, Vol SE-10, No. 5, pp. 477~487, Sept. 1984.

[3] Richard M. stallman, "Using and Porting GNU CC for version 2.7," pp. 223~438, Free Software Foundation, 1996.

[4] Axel T. Scheiner, "USING C with CURSES,

LEX and YACC," pp. 25~90, Prentice Hall, 1990.

[5] 한국전자통신연구소 반도체연구단, "MPEG-4 비디오 부호화 알고리듬 개발 연구," pp. iii~vi, 한국전자통신연구소, 1995.

[6] 권 육춘, "GNU 컴파일러를 이용한 DSP용 C 컴파일러의 설계 및 구현," 석사 학위논문, 경북대학교 대학원, pp. 3~7, 1994.

[7] 한국전자통신연구소 반도체연구단, "ES-C2340 DSP2 Digital Signal Processor User's Guide," 한국전자통신연구소, 1995.

[8] K. Leary, "Numerical C and DSP," Dr. Dobb's Journal, Vol. 19, Iss. 8, pp. 18~24, Aug. 1994.

[9] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "Compilers, Principles, Techniques, and Tools," pp. 463~584, Addison-Wesley, 1988.

[10] 한 기천, "Vocoder DSP Instruction Format," 한국전자통신연구소, 1993.

이 시 영

1995년 경북대학교 컴퓨터공학과 졸업(학사)

1995년~현재 경북대학교 대학원 컴퓨터공학과 석사과정

관심분야: 이미지 프로세싱, 컴파일러, 소프트웨어공학

권 육 춘

1990년 경북대학교 컴퓨터공학과 졸업(학사)

1995년 경북대학교 대학원 컴퓨터공학과 졸업(공학석사)

1995년~현재 경북대학교 대학원 컴퓨터공학과 박사과정

관심분야: 멀티미디어, 이미지 프로세싱, 컴파일러

유 하 영

1982년 서울대학교 전자공학과 졸업(학사)

1985년 Iowa state Univ.(미) 대학원 전기공학과 졸업(공학석사)

1988년 Iowa state Univ.(미) 대학원 전기공학과 졸업(공학박사)

1989년~현재 한국전자통신 연구소 VLSI 구조 연구실 책임연구원

관심분야: VLSI 설계, CAD, Digital Signal Processing

한 기 천

1986년 광운대학교 전자계산학과 졸업(학사)

1988년 광운대학교 대학원 전자계산학과 졸업(공학석사)

1988년~현재 한국전자통신 연구소 VLSI 구조 연구실 선임연구원

관심분야: Digital Signal Processing, Real time system, 운영체제

김 승 호

1981년 경북대학교 전자공학과 졸업(학사)

1983년 한국과학기술원 전산학과 졸업(공학석사)

1994년 한국과학기술원 전산학과 졸업(공학박사)

1985년~현재 경북대학교 컴퓨터공학과 부교수

관심분야: 알고리즘, 멀티미디어