16-bit CPU용 C 컴파일러 개발

정삼진*
*백석대학교 정보통신학부
e-mail:sjjeong@bu.ac.kr

Development of C Compiler for 16-bit CPU

Sam-Jin Jeong*
*Dept of Information and Communication, Baekseok University

요 약

본 연구는 16 비트 CPU를 위한 새로운 C 컴파일러를 개발하고자 한다. 새로운 ASIC 프로세서가 특정 용도로 설계되었을 때 그 CPU를 위한 새로운 컴파일러의 개발이 필요하다.

공개 소프트웨어인 GNU C 컴파일러를 사용하여 기계 의존 원시 파일들을 수정함으로서 새로운 컴파일러를 개발할 수 있다. 개발된 컴파일러는 단지 기계어에 의해 처리될 수 있는 기능들만 지원할 수 있기 때문에 곱셈, 나눗셈, 부동소수점 처리등과 같은 기능들을 지원하기 위해서는 더 많은 연구가 필요하다.

완전한 컴파일러가 개발된 후에는 새로운 CPU에서 실행될 수 있는 응용 프로그램의 개발이 필요하다. 본 연구에 의해서 앞으로 개발될 여러 가지 다른 용도의 CPU를 위한 컴파일러들이 쉽게 개발될 수 있을 것이다.

1. 서론

지금까지는 소수의 유명 컴퓨터 회사에서 개발한 컴퓨터를 잘 사용하는 것에 만족하였으나, 각종 전자 제품, 통신기기등과 같은 특수한 목적에 범용 프로세서(processor)를 사용할 경우 상당히 비효율적이기 때문에, 사용 목적에 따라 그 목적에 적합한 프로세서를 제작하고자 하는 주문형 프로세서(ASIC)에 관한 연구가 매우 활발하게 진행되고 있어, 앞으로 다양한 용도의 주문형 프로세서의 출현될 것으로 예상된다.

새로운 프로세서가 개발되면 기존의 프로세서와는 다른 새로운 명령어(기계어)를 인식하므로 이에 적절한 어셈블러(assembler)가 필요하게 되고, 고급 언어의 사용을 위해 새로운 컴파일러(compiler)가 필요하게 된다. 새로운 프로세서를 위한 어셈블리어와 컴파일러 개발 작업은 프로그램 개발 환경이 잘 갖추어져 있는 기존의 범용 컴퓨터 시스템에서 이루어진다.

새로운 프로세서를 위한 컴파일러를 개발하는 방법에는 직접 설계 및 프로그램으로 작성하는 방법과 public domain에 공개된 컴파일러를 이용하여 관련된 부분을 수정하여 개발하는 방법이 있다.

본 연구는 개발 기간이 단축이 되고, 안정성이 있고, 또한 향후 새로운 프로세서를 위한 컴파일러 개발의 장래성을 위해 두 번째 방법을 택하여 UNIX 환경 하에서 사용할 수 있는 GNU C 컴파일러(간단히 GCC라고 부른다)를 이용하여 개발하였다. GCC는 Free Software Foundation에서 개발한 것으로서, 저작권을 주장하지 않고서 누구나 제약 없이 사용할 수 있도록 공개되어 있다[5]. Host 컴퓨터는 586PC이고, 운영체제는 공개된 UNIX인 LINUX을 바탕으로 gcc를 사용하여 개발하였다[6]. 일반적으로 컴파일러는 대상이 되는 프로세서와 무관한 전반부(front end)와 프로세서와 밀접한 관계가 있는 후반부(back end)로 크게 두 부분으로 나누어진다[1]. GCC는 대단히 방대한 분량의 소프트웨어이지만, 후반부는 비

교적 간단하여 여러 가지 대상 프로세서를 위한 컴파일러를 적은 노력으로 개발할 수가 있다[5].

본고의 2장에서는 GCC의 설치과정과 GCC에서 이루어지는 작업 단계를 살펴보고, 3장에서는 본 연구의 대상이 되는 CPU의 구조 및 특성과 이와 관련하여 실질적인 C 컴파일러의 설계 및 구현에 대해서 상세히 설명하고, 4장은 새로 개발된 C 컴파일러에 의해서 생성된 어셈블리 코드를 보여 준다.

2. GCC의 구조

GCC는 여러 단계를 거쳐 C 언어로 작성된 원시 프로그램(source program)을 컴파일한다.

처음의 전처리(preprocessing) 단계를 수행하는 프로그램은 “cpp″이며, 두 번째 단계부터 마지막 단계까지를 수행하는 프로그램은 “cc1″이다. 그리고 cpp와 cc1 프로그램을 순서에 따라 수행하도록 제어하는 프로그램이 있는데, 이것이 “gcc″이다.

GCC는 cpp나 cc1을 UNIX 시스템 함수 “vfork″를 수행하여 자녀 프로세서(child process)를 생성하여 차례로 cpp와 cc1을 수행하도록 제어한다. 이때 각 단계의 수행에 필요한 스위치(switch)들을 파라메타(parameter)로 넘겨준다. GCC를 수행할 때 컴파일하고자 하는 파일만 지정해 주면 충분하지만, 필요에 따라서 컴파일러 수행 방법을 여러 가지 스위치를 이용하여 지정해 줄 수도 있다. GCC는 프로그램 내에 스위치가 지정되지 않았을 때 디폴트(default)로 적용되는 스위치들을 가지고 있다가 gcc를 수행할 때 지정된 스위치에 대해서만 디폴트기능을 대치하여 cpp와 cc1에 해당 스위치들을 넘겨주어, 지정된 스위치에 따라 컴파일하도록 한다.

전처리 단계에서 생성된 새로운 원시 프로그램은 /tmp 디렉토리(directory)에 임시 파일로 저장한 후 다음 단계인 어휘 분석 단계에서 사용된다.

어휘 분석(scanning) 단계는 원시 프로그램을 구성하는 문자열을, 문법적으로 의미를 갖는 구문적 요소인 토큰(token)들로 구별해 낸다.

원시 프로그램은 어휘 분석 단계를 거치면 일련의 토큰들로 바뀌게 되고, 이것이 다음 단계인 파싱(parsing) 단계로 넘어간다.

일반적으로 파싱단계는 입력된 원시 프로그램을 구문적으로 분석하여 문법적 오류가 발생하면 오류 메시지를 생성하고, 오류가 발견되지 않으면 원시 프로그램의 문법적 구조를 정확하게 표현할 수 있고, 다음 단계에서 편리하게 처리할 수 있는 형태인 구문 트리를 생성하는 단계이다. GCC에서는 Free Software Foundation에서 개발하여 공개한 파서 생성기인 GNU bison을 이용하여 생성된 파서를 사용한다[2][3]. GCC의 파싱단계는 어휘 분석 단계에서 생성된 토큰 열을 받아서 프로그램을 문장 단위로 처리하여 각각의 문장을 구문 트리로 변환한다.

RTL은 Register Transfer Language의 약자로서, gcc가 구문 트리를 중간 코드로 변환하는데 중간코드로 사용되는 언어이다. RTL의 형태는 lisp 언어의 표현 방법인 리스트(list)이다. 따라서 lisp 언어의 특징인 패턴 매칭(pattern matching)을 용이하게 처리할 수가 있다. 이 단계를 거쳐 구문 트리가 RTL로 변환되면, 이후에 이루어지는 컴파일러의 모든 작업은 RTL을 대상으로 이루어지게 된다.

코드 최적화 단계에서는 RTL을 대상으로 여러 가지 최적화 작업을 하게 된다. 최적화단계에서는 loop 최적화, 공통수식 제거, 레지스터(register) 할당 등과 같은 작업을 통하여 최종적으로 효율이 좋은 코드를 생성하기 위한 단계이다. RTL을 대상으로 작업을 하여 효율이 좋은 RTL을 생성해 낸다.

마지막 단계는 코드 생성 단계로서, RTL로부터 대상 프로세서를 위한 어셈블리코드를 생성한다.

3. C 컴파일러의 설계 및 구현

본 장은 대상 머신에 맞는 Word의 정의, 머신 모드의 설정, 레지스터 사용의 정의, 중간 코드 및 출력 코드 생성을 위한 머신 표현파일(machine description file)의 구현에 대해서 설명하고자 한다.

3.1. Word의 정의

GCC의 대상 머신은 32-bit 머신이고, 레지스터들의 크기 또한 32-bit이기 때문에 GCC의 표준 Word의 크기는 32-bit이지만, 본 연구의 대상 머신은 16-bit 머신이고, 일반 레지스터(general register)의 크기 또한 16-bit이기 때문에, 본 연구의 컴파일러의 표준 Word의 크기는 16-bit로 하며, Integer 또한 16-bit로 정의한다[4].

3.2 머신 모드 의 설정.

머신 모드는 특정한 머신 레벨(level)에서의 데이터의 크기와 포멧에 관해서 상세히 설명하며, 각각의 RTL 표현은 머신 모드를 가진다. GNU C 컴파일러에서는 32-bit가 기준이 되어 Integer과 Word를 Single Integer(SI)로 선언하지만, 본 컴파일러에서는 16-bit가 기준이 되어 Integer과 Word를 Half Integer(HI)로 수정하며, GNU C 컴파일러에서 정의한 전체적인 머신 모드 골격을 유지하고자 한다.

3.3 레지스터 사용의 정의

3.3.1 Hard 레지스터의 구성

일반 레지스터는 16-bit 8개(R0-R7)로 구성되어 있으며, 특별 레지스터(special register)는 인덱스 레지스터인 Offset0(16-bit), Offset1(16-bit)와 주소 레지스터인 BP0(20-bit), BP1(20-bit)와 Stack Pointer(20-bit)와 Program Counter인 PC(20-bit)로 구성되어 있다.

3.3.2 레지스터의 정의

실제적인 Hard 레지스터의 수는 14개이나, 사용자가 프로그램용으로 사용할 수 없는 Program Counter(PC)와 Status 레지스터를 제외하면 12개이다. 데이터 레지스터 번호는 0 - 7 이며, R0 - R7으로 표현하고, 인덱스 레지스터 번호는 8 - 9, I0 - I1으로 표현하고, 주소 레지스터 번호는 10 - 11, A0 - A1으로 표현하고, Stack Pointer는 12번 이고, SP으로 표현한다. 대상 머신에서 Floating Point 레지스터는 없다.

3.4 머신 표현 파일 구현

본 연구에서는 C 언어의 문법에 따른 컴파일러의 작업 내용은 GNU C 컴파일러의 내용을 그대로 사용하고 대상 머신에 의하여 컴파일러의 후반부인 중간 코드 생성 부분 이후의 과정을 설계 및 구현한다.

중간 코드 생성을 위해서는 필요한 중간 코드의 형태를 정의하고, 컴파일러가 그들 중 각 문법 트리에 적합한 것을 선택하여 중간 코드를 생성하도록 하여야 한다. 머신 표현 파일에서 명령어 기술시 에 임의의 이름이 주어진 명령어 형태가 이러한 목적으로 사용된다.

이러한 중간 코드의 형태는 대상 머신의 명령어 집합에 따라, 제공되는 중간 코드 형태의 개수라든지 각 중간 코드의 형태에서 operand가 될 수 있는 요건들이 달라진다.

본 머신 표현 파일은 16-bit integer만 처리할 수 있는, 즉 머신 모드가 HI인 경우만 구현되어 있기 때문에 Double Integer이나 Floating-point number 등은 처리할 수 없다.

대상 머신에 대한 각 명령어의 기능 및 각 명령어에 따라 insn의 구현 방법의 소개, 구현된 각각의 define_insn 코드, 어셈블리 코드 생성 루틴에 대한 상세 설명은 생략한다.

4. 목적 프로그램 생성

본 장은 샘플 프로그램을 통하여 C 프로그램이 중간 코드인 RTL로 변환되고, 또한 RTL 코드가 목적 프로그램인 어셈블리어로 변환되어 지는 과정을 상세히 기술한다.

본 샘플 프로그램은 FOR문, IF문, 할당문들이 CMP 명령어 및 JUMP 명령어, GOTO 명령어 등으로 어떻게 변환되어지는 가를 보여준다.

4.1 C 원시 프로그램

```
main()
{
 int i, j, k, l ;
 j = 0 ;
 k = 0 ;
 l = 0 ;
 for (i=1 ; i<=100 ; i++)
   {  if (i <= 50)
        j = j + i ;
      else
        k = k + i;
      l = l + i;
    }
}
```

4.2 어셈블리 코드(Assembly Code)

```
;; Pseudo Code
#NO_APP
gcc2_compiled.:
___gnu_compiled_c:
.text
        .even
.globl _main

;; Main Procedure
_main:
        PUSH M(SP) = A1                 ;;  PUSH memory address
at SP
        CALL ___main                    ;;  call _main
        LOAD R0 = 0
        STORE *(A1-4) = R0              ;;  j = 0
        LOAD R0 = 0
        STORE *(A1-6) = R0              ;;  k = 0
        LOAD R0 = 0
        STORE *(A1-8) = R0              ;;  l = 0
```

```
        LOAD R0 = 1
        STORE *(A1-2) = R0              ;;  i = 1
L2:
        LOAD R1 = *(A1-2)
        CMP R1 100                      ;;  compare i & 100
        if(above) goto L3               ;;  if i > 100 goto L3(END)
        LOAD R1 = *(A1-2)
        CMP R1 50                       ;;  compare i & 50
    if(above) goto L5                   ;;  if i > 50 goto L5
        LOAD R0 = *(A1-4)
        ADD R0 +=  *(A1-2)
        STORE *(A1-4) = R0              ;;  j = j + i
        goto L6                         ;;  goto L6
L5:
        LOAD R0 = *(A1-6)
        ADD R0 +=  *(A1-2)
        STORE *(A1-6) = R0              ;;  k = k + i
L6:
        LOAD R0 = *(A1-8)
        ADD R0 +=  *(A1-2)
        STORE *(A1-8) = R0              ;;  l = l + i
L4:
        LOAD R0 = *(A1-2)
        ADD R0 += 1
        STORE *(A1-2) = R0              ;;  i = i + 1
        goto L2                         ;;  goto L2 (DO again)
L3:
L1:
        POP A1 = M(SP)                  ;;  POP Stack Point
        RET
```

5. 결론

GNU C 컴파일러를 이용하여 새로운 프로세서를 위한 컴파일러를 개발하는 작업은 대상 프로세서의 특성에 관계되는 부분이 무척 적기 때문에 비교적 용이하게 컴파일러를 개발 할 수가 있는 장점이 있다. 그러나 컴파일러 개발에 앞서 전제되어야 하는 사항은 GCC가 중간 코드로 사용하는 RTL의 정확한 사용법과 대상 프로세서의 특성, 그리고 이들과의 관계를 확실히 이해하고 있어야 한다. 일단 이런 것들에 대한 완벽한 이해가 이루어지면, 원래의 GCC와 동일한 수준으로 정확하게 동작하는 컴파일러를 얻을 수 있다.

본 연구를 수행하는 과정에서 나타난 가장 근본적인 문제점은, GCC가 기본적으로 32-bit 프로세서에서 동작하도록 개발되어 있으나, 본 과제에서 대상으로 하고 있는 CPU는 16-bit인 것이다. 따라서 표준 데이터 크기를 16-bit로 바꾸어서 생각해야 한다. 그리고 주소를 지정하는 주소 레지스터(address register)의 크기가 20-bit로서 8의 배수가 아닌 점이다. 원래 GCC의 모든 데이터는 그 크기가 8의 배수가 되도록 되어있다. 지금은 주소를 24-bit로 처리하고 있으나, 아직까지는 별 문제가 나타나고 있지 않다.

향후 연구과제로는, 먼저 CPU가 하드웨어로 지원하지 못하는 기능들을 지원하는 라이브러리 함수를 개발하는 과제가 반드시 뒤를 이어야 하겠다. 아울러서 어셈블러와 링크에디터, 그리고 헤더 파일등 도 별도로 개발되어야 한다.

본 연구가 계속되어서 하나의 완전한 컴파일러의 개발이 완료되면, 이 과정에서 축적된 기술을 적용하여, 앞으로 예측되는 다양한 형태의 주문형 프로세서에 대해서 별 어려움 없이 짧은 시간 내에 새로운 컴파일러를 개발할 수 있으리라 기대된다.

참고문헌

[1] Alfred V.Aho,Ravi Sethi and Jeffrey D. Ullman, "Compilers: Principles, Techniques, and Tools", Addison-Wesley, 1986.

[2] Charles Donnelly and Richard Stallman, "Bison: The YACC-compatible Parser Generator",Free Software Foundation, Dec., 1992.

[3] MasBr90 Tony Mason and Doug Brown, "lex & yacc", O'Reilly & Associates, Inc., 1990.

[4] Richard M. Stallman, "The C Preprocessor", Free Sofeware Foundation, Jul., 1992

[5] Richard M. Stallman, "Using and Porting GNU CC", Free Software Foundation, Oct., 1993.

[6] Matt Welsh, "Linux Installation and Getting Started", Free Software Foundation, 1994.