GNU Binutils를 기반으로 한 재겨냥성 이진 유틸리티의 개발

(Development of Retargetable Binary Utilities Based on GNU Binutils)

김 호 균 † (Hogyun Kim) 정 지 문 † (Jimoon Jung) 이 종 원 † (Jongwon Lee)
박 상 현 † (Sanghyun Park) 윤 종 희 † (Jonghee Yoon) 백 윤 흥 †† (Yunheung Paek)

요 약 오늘날 가전 제품 시장에서 임베디드 시스템은 time-to-market 이라는 개념이 점차 중요해지고 있다. 프로세서의 개발 주기가 점차 짧아짐에 따라 소프트웨어의 개발 또한 중요하게 생각되고 있다. 그러나 새로운 프로세서에 특화된 소프트웨어 개발 도구들을 개발하는 시간은 여전히 개선되지 않고 있다. 이러한 점에서 Architecture Description Language(ADL)은 새로운 프로세서에 대한 소프트웨어 개발 도구들을 자동으로 생성하게 함으로써 개발하는 수고를 덜 수 있다. 본 논문에서는 GNU Binutils를 이용하여 각각의 프로세서에 맞는 소프트웨어 개발 도구들을 자동으로 생성하였다. 이 연구를 통하여 우리는 어셈블러나 링커와 같은 소프트웨어 개발 도구들을 쉽고 빠르게 생성할 수 있었다.

키워드 : Architecture Description Language, 재겨냥성 이진 유틸리티, GNU Binutils

Abstract In this days, the concept of time-to-market is important in embedded systems in consumer electronics. According to the short time of development period, it is also important in development of Software Development toolkits (SDKs). However, it is not improved to the development time of SDKs specialized in new processors. In this point, the Architecture Description Language (ADL) is an alternative to relieve the pain of building the SDKs as the required SDKs can be automatically generated from ADL for the processor. In this paper, we automatically generate SDKs specialized in processors using GNU Binutils. Through this research, we can more easier and faster produce the SDKs such as assembler and linker than by using handcrafted code.

Key words : Architecture Description Language, Retargetable Binary Utilities, GNU Binutils

· 본 연구는 교육과학기술부/한국과학재단 우수연구센터육성사업(R11-2008-007-01001-0), 지식경제부 출연금으로 ETRI, SoC 산업진흥센터에서 수행한 ITSoC 핵심설계인력양성사업, 서울시 산학연 협력사업, 2008 년도 정보(교육과학기술부)의 재원으로 한국과학재단의 국가지정연구실 사업(R0A-2008-20110-0), 지식경제부 및 정보통신연구진흥원의 대학 IT 연구센터 지원사업(IITA-2008-C1090-0801-0020), 지식경제부 및 정보통신 연구진흥원의 IT원천기술개발사업[과제관리번호:2006-S-006-04, 과제명: 유비쿼터스 단말용 부품/모듈]의 지원을 받아 수행되었습니다.

† 비 회 원 : 서울대학교 전기컴퓨터공학부
hkkim@optimizer.snu.ac.kr
jmjung@optimizer.snu.ac.kr
jwlee@optimizer.snu.ac.kr
shpark@optimizer.snu.ac.kr
jhyoon@optimizer.snu.ac.kr
(Corresponding author임)

· 이 논문은 2008 프로그래밍언어연구회 추계학술대회에서 '재겨냥성 어셈블러와 링커의 개발'의 제목으로 발표된 논문을 확장한 것임

†† 종신회원 : 서울대학교 전기컴퓨터공학부 교수
ypaek@snu.ac.kr

논문접수 : 2009년 3월 3일
심사완료 : 2009년 6월 29일

정보과학회논문지: 소프트웨어 및 응용 제36권 제9호(2009.9)

1. 서 론

가전 제품 기기에서 임베디드 시스템은 매우 중요한 위치를 차지하고 있다. 임베디드 시스템에서 소프트웨어 개발 도구들과 관련된 문제가 있는데 그 중 하나가 바로 time-to-market 개념이다. 최근에 들면서 새로운 프로세서에 대한 개발 주기가 점차 짧아지고 있다. 그에 따라 특정 프로세서에 특화된 소프트웨어 개발 도구들을 빠르게 개발하는 것이 새로운 프로세서의 개발시간을 단축시키는 데 큰 영향을 주고 있다.

다른 문제로는 임베디드 시스템에서 프로세서의 성능을 높이기 위하여 수작업으로 코드를 생성하였지만 점차 시스템이 복잡해짐에 따라 한계에 부딪치게 되었다. 임베디드 시스템에서 ASIP이나 DSP와 같은 프로세서들은 중요한 위치를 차지하고 있다. 전통적으로 ASIP이나 DSP와 같은 프로세서들은 코드의 성능을 높이기 위하여 수작업으로 코드를 작성하여 왔다. 그러나 어플리케이션의 복잡성과 사이즈가 증가함에 따라 이러한 방식은 점차 그 한계에 부딪치게 되었다. 결국 임베디드 시스템의 개발자들은 ADL과 재겨냥성 컴파일러[1]를 이용하여 코드를 자동으로 생성함으로써 이러한 문제를 해결하려고 하고 있다.

결국 임베디드 시스템에서 ADL을 이용한 효율적인 코드 생성을 위한 컴파일러 연구는 활발하게 진행되고 있다. 그러나 어셈블러나 링커, 시뮬레이터와 같은 다른 재겨냥성 소프트웨어 개발 도구들에 대한 연구는 여전히 필요하다.

어셈블러나 링커를 연구하는데 많이 사용되는 도구는 GNU Binutils 패키지이다. GNU Binutils는 오픈 소스이기 때문에 모든 사람들이 자유롭게 이용할 수 있고 이미 여러 프로세서들에 대하여 개발이 되어 있다.

그러나 GNU Binutils는 새로운 프로세서에 대하여 이식 작업을 하고자 할 때 참고할 수 있는 자료가 부족하고 또한 수작업으로 이식 작업을 하기에는 그 코드의 크기가 커서 상당한 시간과 노력이 들게 된다.

이러한 점을 개선하기 위하여 본 논문에서는 새로운 프로세서에 대한 Binutils를 이식하기 위해서 필요한 작업들을 자동으로 하는 연구를 하였다. 그 결과 Binutils의 필요한 입력 파일을 자동으로 생성할 수 있었으며 더 나아가서는 새로운 프로세서에 대한 어셈블러나 링커와 같은 Binutils에서 제공하는 여러 가지 도구들을 자동으로 생성해 준다.

본 논문의 구조는 다음과 같다. 2장에서는 이 주제와 관련된 다른 연구에 대하여 알아보겠고 3장에서는 구체적으로 어떻게 하였는지를 설명하겠다. 4장에서는 실험 환경과 그 결과에 대하여 설명하겠고 마지막으로 5장에서는 결론을 내리고 앞으로 더 보완되어야 할 부분에 대하여 이야기 하겠다.

2. 관련 연구

이전부터 재겨냥성 어셈블러와 링커에 대한 연구는 계속해서 수행되어 왔다. 이러한 소프트웨어 개발 도구들을 만들기 위해서는 프로세서를 기술하는 ADL에 대한 연구가 중요하다. 왜냐하면 이는 소프트웨어 개발 도구를 생성하는데 가장 기본적인 작업이면서도 ADL을 기술하는 특징에 따라 그 소프트웨어 개발 도구들의 성능이 달라질 수 있기 때문이다.

여러 ADL을 기술하는 방법 중 그 내부 구현 방법에 따라 크게 두 가지로 구분할 수 있다. 내부 구현을 Binutils를 이용하는 방법과 자체적으로 구현하는 방법이 있다.

자체적으로 구현하는 방법을 사용하여 재겨냥성 어셈블러와 링커를 생성하면 원하는 기능을 쉽게 추가하거나 수정하기가 쉽다. 그러나 이러한 방법은 구현된 기능만을 사용할 수 있기 때문에 다른 기능을 사용하려면 처음부터 다시 제작해야 한다는 단점이 있다. 이 방법을 사용한 예로는 CHESS와 LISA[2] 등이 있다.

이와는 반대로 Binutils를 이용하면 Binutils에서 제공하는 여러 가지 다양한 기능을 사용할 수 있다. 어셈블러와 링커 뿐만 아니라 Objdump, Objcopy등 이진 파일을 처리할 수 있는 다양한 기능 또한 사용할 수 있다. 그러나 이것의 단점은 코드를 수정하기가 어렵기 때문에 새로운 기능을 추가하거나 기존의 기능을 수정하기가 어렵다는 점이다. 하지만 가장 최근에 나온 Binutils는 여러 가지 형태의 어셈블리 파일을 처리할 수 있기 때문에 새로운 기능을 추가할 필요는 거의 없을 것이다. 이것을 이용한 예로는 Abbaspour and Zhu[2002][3]와 ArchC[2008][4] 등이 있다.

결국 Binutils를 이용한다면 그 내부 동작 원리나 구현 방법은 비슷할 것이다. 차이점은 프로세서를 기술하는 ADL에서 보일 것인데 ADL을 얼마나 효율적으로 기술하느냐에 따라 그 성능이 결정될 것이다.

먼저 Abbaspour and Zhu[2002]을 보면 ADL을 기술하는 방법은 우리의 방법과 비슷하다. 그러나 이 논문에서는 명령어와 관련된 부분만을 설명하였고 프로세서의 다른 특징을 기술하는 방법에 대해서는 나타나 있지 않다. 또한 실험에서는 SPARC에 대해서만 나오기 때문에 다른 프로세서에 대해서도 재대로 동작하는지에 대해서는 검증이 불가능하다.

다음으로 ArchC[2008]을 보면 ADL을 기술할 때 우리와 가장 큰 차이점은 재배치를 기술하는데 있다. ArchC에서는 재배치를 기술할 때 동작에 대해서 매우 자세하

게 기술하도록 되어 있다. 이 방법은 새로운 재배치를 기술할 때 효과적일 수 있다. 그러나 RISC 프로세서에 대해서 기존에 나와 있는 재배치 방법을 이용하는 프로세서가 대부분이기 때문에 몇 가지 정해진 재배치 형식만을 정의해 준다면 사용자가 쉽게 기술할 수 있다. 이것을 위해서 우리는 Immediate, direct address, relative address 등을 지원하고 있다.

3. 본 론

3.1 새로운 프로세서를 위한 ADL 기술

새로운 프로세서를 개발할 때 그에 따른 새로운 기능을 추가하거나 기존의 기능에 수정을 해야 하는 경우가 발생한다. 이 경우 어셈블러나 링커와 같은 소프트웨어를 직접 변경하게 되면 많은 시간과 노력이 필요할 뿐만 아니라 변경과정에서 많은 에러를 겪을 위험이 있다. 이를 해결하기 위하여 ADL을 특정 포맷에 맞게 기술하는 것은 중요하다고 하겠다.

본 논문에서는 서울대학교 소프트웨어 최적화 연구실에서 개발한 SoarDL[5]을 사용하고 있다. 이번장에는 SoarDL과 관련된 기술에 대하여 설명하도록 하겠다.

3.1.1 저장 장치

아래의 표 1은 Openrisc 1000의 저장장치 구조를 보여준다. 아래의 그림처럼 저장장치에서는 해당 프로세서의 메모리와 레지스터 정보를 기술하여 준다.

표 1 Openrisc 1000 저장 장치의 구조

```
storage odalrisc_storage : odalrisc
{
memory byte prog_mem {
latency 1 @[0x000000..0x7fffff];
};
memory byte data_mem {
latency 1 @[0x800000..0xffffff];
};
register si R[16];
register si CR;// special purpose register
register si IR;// instruction register
register si PC;// program counter
...
}
```

저장장치는 크게 메모리와 레지스터 정보를 기술하는 부분으로 나뉘어지는데 먼저 메모리 정보를 기술하는 방법은 아래의 표 2와 같다. 아래의 표 2에서 cell_width는 메모리에서 주소로 지정할 수 있는 최소단위의 크기를 말한다.

다음으로 레지스터 정보를 기술하는 방법은 아래의 표 3과 같다. 아래의 표 3에서 register_width는 각 레지스터의 갯수이다.

표 2 Memory 정보 기술

```
memory cell_width memory_name { latency 1
@[start_address..end_address]; };
```

표 3 Register 정보 기술

```
register register_width register_name;
register register_width register_name[register_size];
```

3.1.2 컨벤션

아래의 표 4는 Openrisc 1000의 컨벤션 구조를 보여준다.

표 4 Openrisc 1000의 컨벤션 구조

```
conventions odalrisc_conventions : odalrisc
{
// endian
words_endian little_endian;
bytes_endian little_endian;
bits_endian little_endian;
base_insn_bitsize 32;// the same as word size)
word_bitsize 32;// -> primitive word/si
// special registers like stack pointer
programCounter PC PC;
stackPointer R14 SPR;
framePointer R13 FPR;
statusRegister CR SR;
return_register 0 R0;
return_register 1 R1;
argument_register 0 R0;
argument_register 1 R1;
argument_register 2 R2;
argument_register 3 R3;
callee_save_register { R6, R7, R8, R9, R10, R11, R12, R15
};
caller_save_register { R0, R1, R2, R3, R4, R5 };
not_reg_allocatable { IR, N, C, V, Z };
}
```

위의 표 4에서처럼 컨벤션에서는 엔디언, 명령어 크기, 그리고 특수 레지스터들을 선언한다. 그리고 특별 레지스터들은 아래의 표 5와 같은 구조로 기술한다.

표 5 특수 레지스터의 구조

```
programCounter/linkRegister/stackPointer/statusRegister/frame
Pointer physical_register_name ID
```

3.1.3 이진 유틸리티의 구조

본 절은 어셈블러와 링커를 생성하기 위해 필요한 부분들을 기술한다. 크게 아래와 같이 주석, 레이블, 메모리, 재배치로 나눌 수 있다. 그리고 이전 논문과 비교하였을 때 그 개선된 점에 대해 기술하도록 하겠다.

3.1.3.1 주석 구조

아래의 표 6은 주석의 구조를 보여준다. 여기에서 각각의 문자는 어셈블리 파일에서 사용하는 문자를 나타낸다. Comment_chars는 어셈블리 파일에서 주석을 나타낼 때 필요한 기호, 그리고 line_comment_chars와 line_separator_chars은 어셈블리 파일에서 라인별 주석을 나타내기 위해서 필요한 기호, 그리고 exp_chars와 flt_chars는 어셈블리 파일에서 exponential과 floating point를 표기할 때 필요한 기호를 나타낸다.

표 6 Openrisc 1000의 char 구조

```
Chars
{
comment_chars";#";
line_comment_chars";";
line_separator_chars"#";
exp_chars"eE";
flt_chars"rRsSfFdDxXpP";
}
```

3.1.3.2 레이블의 구조

어셈블리 파일에서 레이블은 항상 ':' 기호로 끝을 맺는다. 그러나 특수한 프로세서에서는 레이블은 아니지만 ':' 기호를 사용하는 명령어가 있을 수 있다. 이와 같은 문제를 해결하기 위하여 모든 ':' 기호를 가진 명령어을 여기에 기술하여 레이블과 구분을 하여 준다.

3.1.3.3 메모리 구조

아래의 표 7은 메모리의 구조를 보여준다. 여기에서는 코드 시작 주소와 데이터 시작, 그리고 최대 페이지 크기를 기술한다.

표 7 Openrisc 1000의 memory 구조

```
Memory
{
text_start_addr 0x000000;
data_start_addr 0x800000;
max_page_size 0x400000;
}
```

3.1.3.4 재배치 구조

재배치는 링커가 목적 파일들을 링킹 할 때 참조를 찾지 못한 심볼들에 대하여 링커를 실행 한 후 정확한 주소를 찾아주기 위한 동작을 말한다. 최신의 프로세서 일수록 이와 같은 재배치 동작이 더욱 복잡해지고 있으며 이를 쉽고 효율적으로 기술하는 것이 중요한 일이다.

아래의 표 8은 Openrisc 1000의 재배치 동작을 보여주고 있다. 여기에서 operandtype은 각 재배치들이 오퍼랜드로 사용되었을 때 어떻게 사용되는 지에 따라 그 타입을 정의한 것이고 rightshift는 이 오퍼랜드가 실제 동작하기 전에 오른쪽으로 이동을 하는 비트 수를 나타낸다. 그리고 size는 이 오퍼랜드가 사용되는 명령어의 크기이고 src_mask와 dst_mask는 각각 소스와 목적 명령어의 마스킹되는 비트의 범위를 나타낸다.

표 8 Openrisc 1000의 relocation 구조

```
Relocation
{
   reloc
   {
      operandtype LIMM[0,15];
      rightshift 0;
      size 2; // (0=byte, 1=short, 2=long)
      src_mask 0x0000ffff;
      dst_mask 0x0000ffff;
   }
   reloc
   {
      operandtype HIMM[0,15];
      rightshift 16;
      size 2;
      src_mask 0xffff0000;
      dst_mask 0x0000ffff;
   }
   reloc
   {
      operandtype REL[0,25];
      rightshift 2;
      size 2;
      src_mask 0x00000000;
      dst_mask 0x03ffffff;
   }
}
```

위의 표 8에서 operandtype이 LIMM과 HIMM은 각각 하위 비트와 상위 비트의 크기를 나타낸다. 이러한 오퍼랜드들은 대부분의 RISC 아키텍쳐에서 사용되고 있는 것으로 메모리의 특정 번지의 값을 읽으려고 할 때 그 메모리 주소값이 너무 커 하나의 명령어안에 이 주소값을 표시할 수 없는 경우 메모리 번지를 반으로 나누어 특정 레지스터에 저장할 때 사용된다. 특히 이러한 명령어의 경우 많은 프로세서에서는 슈도 명령어를 사용 하고 있기 때문에 어셈블러나 링커를 수작업으로 생성할 경우 다른 명령어에 비하여 많은 노력이 든다.

그리고 위의 표 8에서 operandtype이 REL인 경우는 상대 주소를 나타낸다. 이 오퍼랜드 역시 RISC 아키텍쳐에서 자주 사용되고 있는 오퍼랜드로써 특정 번지로 점프할 경우 특정 주소값에서 기본 주소값을 뺀 주소값을 표현할 때 사용한다.

마지막으로 위의 표 8에서는 나타나지 않았지만 절대 주소값을 오퍼랜드로 사용하고자 하는 경우는 operandtype

을 DIR로 기술한다. 이는 직접 주소를 나타낸다.

3.1.3.5 Addressmode 구조

AddressMode는 명령어의 오퍼랜드들 중 공통적으로 사용되는 오퍼랜드가 있을 경우 이를 AddressMode에 정의해 줌으로써 프로세서의 명령어를 효율적으로 기술할 수 있게 한다. 실제 ARM 프로세서의 경우 여러 종류의 AddressMode가 정의되어 있으며 이들은 대부분의 명령어에서 반복적으로 사용되고 있다.

AddressMode의 구조 중 서로가 비슷한 구조를 가지는 AddressMode에 대해서는 AddressModeSet으로 묶어서 기술할 수가 있다. 이를 이용하여 프로세서를 기술한다면 더욱 쉽고 빠르게 기술할 수 있을 것이다.

표 9 Openrisc 1000의 AddressModeSet

```
addressModeSet or32AddressModeSet : or32
{
    loadi;
    storei;
    encoding { UNDEF[21];}
}
```

표 10 Openrisc 1000의 AddressMode

```
addressMode loadi : or32AddressModeSet
{
    r rA;
    IMM(16) imm16;
    semantic { efa = plus(rA, imm16); }
    syntax { imm16::"("::rA::")"; }
      encoding { rA::imm16; }
}
```

위의 표 9와 10은 AddressModeSet과 AddressMode를 나타낸다. 위의 표 10에서 semantic은 명령어의 동작을 syntax는 어셈블리 파일에 기술되는 명령어의 형태를 encoding은 이진 파일로 기술될 때 인코딩의 결과를 나타낸다. 위의 표 9와 10에서 보듯이 encoding은 AddressModeSet과 AddressMode에 모두 포함된다. 이는 AddressMode 중 공통되는 encoding이 있을 경우 기술하기 쉽게 하기 위함이다.

3.1.3.6 Instruction 구조

Instruction은 AddressMode와 구조가 동일하다. 차이점은 Instruction에서 실제 프로세서의 명령어들을 기술해 준다.

아래의 표 11과 12는 InstructionSet과 Instruction의 예를 보여주고 있다. InstructionSet과 Instruction의 구조는 AddressModeSet과 AddressMode와 동일하다. 단지 cost 부분만 차이가 있는데 이는 명령어가 실행되는데 걸리는 사이클의 수를 기술해 준다.

표 11 Openrisc 1000의 InstructionSet

```
instructionSet load_instr : or32InstructionSet
{
    lbs;
    lbz;
    lhs;
    lhz;
    lws;
    lwz;
    encoding {
        "100"::UNDEF[29];
    }
}
```

표 12 Openrisc 1000의 Instruction lbs 예제

```
instruction lbs : load_instr
{
    r rD;
    loadi loadi21;
    DAT0 m;
    semantic { rD = zero_extend(m[loadi21, byte]); }
    syntax { "l.lbs "::rD::","::loadi21; }
    encoding { "100"::rD::loadi21;}
    cost(1);
}
```

위의 명령어는 실제 Openrisc 1000의 lbs를 기술한 것이다. InstructionSet과 Instruction 또한 인코딩 정보를 가지고 있는데 이는 비슷한 명령어들끼리 공통되는 인코딩 정보를 가지고 있을 경우 효율적으로 기술할 수 있게 해주기 위해서이다.

3.1.3.7 이전 논문[6]의 개선점

본 절에서는 이전 논문과 비교하였을 때 그 개선된 사항에 대해서 기술하도록 하겠다. 이전 논문에서는 CGEN에서 필요한 파일인 cpu 파일들을 직접 수작업을 통하여 그 내용을 작성하였다. 그러나 본 논문에서는 위에서 설명한 AddressMode, AddressModeSet, Instruction, InstructionSet을 통하여 cpu 파일을 자동으로 생성할 수 있도록 하였다.

Cpu 파일에는 프로세서와 관련된 거의 모든 정보들이 기술되어 있어야 한다. 예를 들면 메모리와 레지스터의 정보는 물론이고 명령어 각각의 정보들 또한 기술되어 있어야 한다. 게다가 명령어 하나를 기술하는 데에는 그에 필요한 오피코드와 오퍼랜더의 정보가 미리 기술되어 있어야 한다. 명령어에 사용되는 오퍼랜더의 경우 레지스터나 직접 주소, 간접 주소 방식이 사용될 수 있다. 이러한 정보들은 필드라는 부분에 기술되고 이를 이용하여 각각의 오퍼랜더를 기술한다. 이러한 부분을 수작업으로 하기에는 상당한 시간이 걸릴 수 있다.

그러나 본 논문에서는 이러한 cpu 파일을 자동으로

생성할 수 있도록 함으로써 그 개발 시간과 노력을 줄일 수 있다. 또한 명령어를 기술하는 부분에서도 앞에서 본 바와 같이 그 명령어의 문법, 형식, 그리고 인코딩 정보만을 기술하면 되기 때문에 cpu 파일을 작성하는 것 보다도 훨씬 쉽게 그 내용을 기술할 수가 있다. 이러한 점에서 이전 논문에 비해서 상당한 부분이 개선되었다고 할 수 있다.

3.2 전체 Binutils[7] 의 구조

이번 장에서는 GNU Binutils 도구에 대하여 설명하도록 하겠다. 먼저 전체적인 Binutils의 구조와 내용에 대해서 알아보고 Binutils 중 프로세서에 독립적인 부분과 의존적인 부분을 구분하여 재겨냥성 Binutils가 되기 위해서는 프로세서에 의존적인 부분들이 어떻게 바꾸어야 하는 지에 대해서 알아보겠다.

3.2.1 전체 Binutils의 구조

Binutils는 GNU Binary Utilities의 약자로써 목적파일을 생성 또는 수정하기 위한 여러 가지 프로그래밍 도구들의 모음이라고 할 수 있다. Binutils에 포함된 소프트웨어 개발 도구들은 다음과 같다.

AS - the GNU assembler.

LD - the GNU linker.

Objdump - Displays information from object files.

Ar - A utility for creating, modifying and extracting from archives.

이 밖에도 object 파일을 다루기 위한 여러 소프트웨어 개발 도구들이 제공된다.

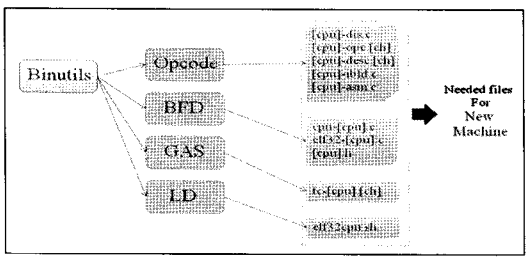

그림 1 Binutils의 전체 구조 및 프로세서 의존적인 부분

위의 그림 1은 Binutils의 전체적인 구조에 대하여 보여준다. Binutils는 크게 Opcode, BFD, GAS, 그리고 LD로 나눌 수 있다. 그리고 각각의 부분마다 프로세서에 의존적인 파일들이 있다. 아래에서 각 부분마다 설명하도록 하겠다.

3.2.1.1 Opcode

Opcode는 명령어의 인코딩, 디코딩, 시뮬레이팅에 필요한 여러 가지 정보를 제공해 준다. 본 절에서 프로세서에 의존적인 파일은 [cpu]-dis.c, [cpu]-opc.[ch], [cpu]-desc.[ch], [cpu]-ibld.c, [cpu]-asm.c가 있다. 여기에서는 프로세서에 의존적인 파일들이 많기 때문에 이를 CGEN이라는 도구를 이용하여 생성한다. CGEN에 대하여는 다음장에서 설명하도록 하겠다.

3.2.1.2 Binary File Descriptor library(BFD)

BFD는 여러 가지 목적 파일의 형식과 프로세서 그리고 운영체제에 대하여 지원을 해야 하기 때문에 복잡한 구조를 가진다. 다시 말하면 본 절이 어셈블러와 링커를 생성하는데 가장 중요한 부분이라고 할 수 있다. 왜냐하면 여기에서 재배치를 지원하기 때문이다. 여기에서 프로세서에 의존적인 부분은 cpu-[cpu].c, elf32-[cpu].c, [cpu].h가 있다.

3.2.1.3 GNU Assembler(GAS)

GAS는 어셈블리 파일을 목적 파일로 변환하는 프로그램이다. GAS는 이러한 어셈블리 파일에서 목적 파일로 바꾸는 과정에서 주로 3가지 일을 하게 된다. 첫 번째는 어셈블리 파일을 읽어 들여 여기에 맞는 전 처리를 해준다. 두 번째는 각 라인 별로 명령어를 파싱한다. 이런 과정에서 심볼의 참조를 찾을 수 없으면 재배치 엔트리를 만들어 준다. 마지막 단계에서는 심볼 테이블과 함께 최종 목적 파일을 만들어 준다. 여기에서 프로세서에 의존적인 부분은 tc-[cpu].cpu이다.

3.2.1.4 GNU Linker(LD)

LD는 여러 목적 파일을 링킹 해주고 해당되는 재배치 엔트리에 대하여 처리를 해준다. 그리고 링커가 동작할 때 링커 스크립트의 제어를 받는다. 본 논문에서는 여러 가지 실행 파일 형식 중 ELF 형식에 맞게 출력하였다. 그 이유는 ELF 형식이 가장 널리 사용되고 있기 때문이다. 여기에서 프로세서에 의존적인 부분은 elf32cpu.sh 파일이다.

3.2.2 CGEN[8] 설명

CGEN은 Cpu tools Generator의 약자로써 이 또한 GNU 프로젝트의 일부분이다. CGEN의 목적은 통일된 구조를 제공하여 어셈블러, 링커, 시뮬레이터와 같은 프로그램을 쉽게 개발하는 데에 있다. 이와 같은 통일된 구조를 구현하기 위하여 CGEN은 CPU 구조를 기술하기 위한 한 가지 언어 CGEN's Register Transfer Language를 제공해 준다. CGEN's RTL은 GCC's RTL을 기반으로 하고 Scheme program language의 문법을 참조하고 있다.

그러나 여전히 개발중인 프로젝트로 현재는 어셈블러와 링커, 시뮬레이터의 부분적인 소스 코드만은 제공해 준다. 어셈블러와 링커를 위하여 CGEN은 Binutils의 오피코드 부분의 프로세서에 의존적인 파일들을 자동으로 생성해 준다.

아래의 그림 2에서와 같이 CGEN은 Guile이라는 해석

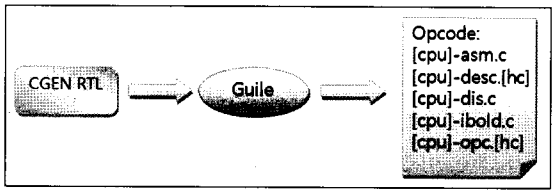

그림 2 Binutils Opcode를 위한 CGEN의 지원

기를 통하여 CGEN RTL 로 기술된 파일을 해석하여 오피코드의 프로세서 의존적인 파일들을 생성해 준다. 본 논문에서는 이를 이용하도록 하겠다.

3.2.3 Retargetable Binutils

이번 장에서는 위에서 설명한 Binutils가 다양한 프로세서에 맞는 재겨냥성 Binutils를 만들기 위한 과정에 대하여 설명하겠다.

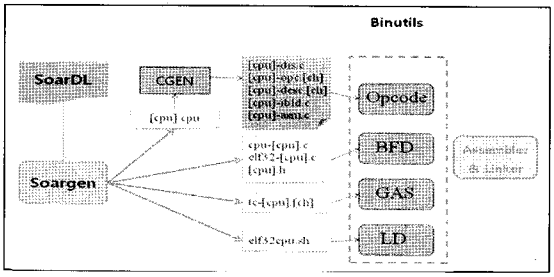

그림 3 재겨냥성 Binutils의 전체 과정

위의 그림 3은 재겨냥성 Binutils를 생성하기 위한 전체 과정에 대하여 보여주고 있다. 위의 그림에서 Soargen은 서울대학교 소프트웨어 최적화 연구실에서 만든 재겨냥성 컴파일러이다. 이 소프트웨어 개발 도구에다 재겨냥성 어셈블러와 링커를 생성하기 위한 코드를 작성하였다.

Soargen은 SoarDL에서 작성한 프로세서의 여러 가지 정보들을 읽어 들인 후 위의 그림 3과 같이 프로세서에 의존적인 파일들을 생성한다. 위의 그림에서 특이한 점은 [cpu].cpu 파일인데 이 파일은 다시 CGEN의 입력파일로 들어간다. 다시 말하면 [cpu].cpu 파일이 바로 CGEN RTL 형식의 파일이 되는 것이다. 이 파일을 입력으로 받은 CGEN은 Binutils 오피코드에 필요한 파일들을 생성하고 나머지 BFD, GAS, 그리고 LD에 필요한 파일들은 Soargen에서 직접 생성한다.

이 외에도 Binutils tool을 실행하기 위해서는 Binutils 폴더에 있는 각각의 메이커 파일들 또한 수정을 해 주어야 한다. 이러한 작업은 그림 3에는 나타나지 않았지만 Soargen을 통하여 이러한 파일들을 수정하여 자동 생성하도록 하였다.

4. 실 험

SoarDL을 이용한 재겨냥성 Binutils를 통한 어셈블러와 링커를 생성한 후 이를 검증하기 위한 검증 환경 및 그 결과에 대하여 본 장에서 이야기 하겠다. 본 실험에서는 생성된 어셈블러와 링커의 정확성에 대하여 검증을 하고 SoarDL로 기술하는 것이 얼마나 유용한지에 대하여 알아보겠다. 프로세서는 Openrisc 1000과 Odalrisc를 실험하였고 벤치 마크는 미디어 벤치 마크인 adpcm, g721, jpeg을 가지고 검증하였다. Openrisc 1000과 Odalrisc는 실험환경이 다르기 때문에 각 실험에 대하여 설명한 후 효율성에 대해서 설명하겠다.

4.1 Openrisc 1000[9]의 설명

Openrisc 1000은 GNU에서 지원하는 플랫폼을 설치하여 실험을 하였다. 실험 환경은 GCC-3.4.4, Binutils-2.17, uClibc-0.9.28, 그리고 Linux-2.6.23이다. 아래의 그림 4는 Openrisc 1000의 실험 환경을 보여준다. 아래의 그림 4에서처럼 GCC 컴파일러를 이용하여 어셈블리 파일을 생성한 후 Binutils에서 지원되는 어셈블러와 링커를 이용한 실행 파일 결과와 SoarDL로 기술된 재겨냥성 Binutils의 어셈블러와 링커를 이용한 실행 파일의 결과를 비교하였다. 두 개의 실행파일이 동일함을 확인할 수 있었다. 실험에 이용된 벤치 마크에는 adpcm, g721, jpeg이 사용되었다.

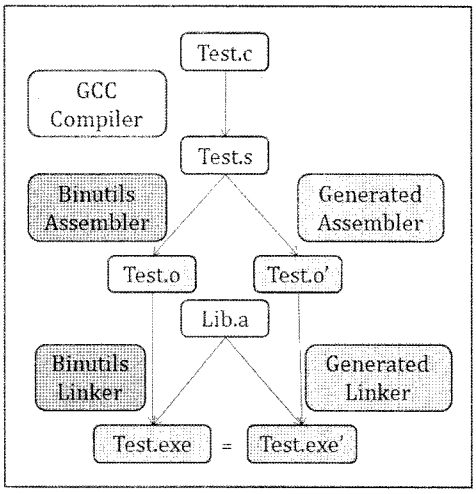

그림 4 Openrisc 1000의 실험 환경

4.2 Odalrisc 설명

Odalrisc는 서울대학교 설계자동화연구실에서 새롭게 개발된 프로세서로 기본적인 RISC 구조를 가진다. Odalrisc는 LISA를 이용하여 개발하였기 때문에 실험 환경 역시 LISA를 이용하여 검정하였다.

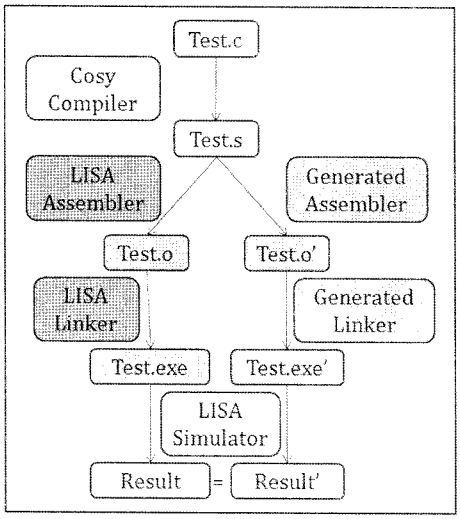

그림 5 Odalrisc의 실험 환경

위의 그림 5에서 보듯이 LISA 에서 제공하는 Cosy 컴파일러를 이용하여 어셈블리 파일을 만든 후 LISA 어셈블러와 링커를 이용하여 실행파일을 만든다. 이 실행파일과 SoarDL로 기술한 재겨냥성 Binutils를 이용하여 생성한 어셈블러와 링커로 만든 실행파일을 비교한다. 비교 환경은 LISA 시뮬레이터를 이용하였다. 본 실험에서는 Jpeg 디코더를 이용하여 검증하였는데 그 결과가 동일하게 나오는 것을 확인하였다.

4.3 효율성 검증

Openrisc 1000과 Odalrisc를 SoarDL로 기술하였을 경우 어떤 효율이 있는지에 대해서 알아보도록 하겠다.

	Binutils code (Line No./Bytes)					ISA Modeling SoarDL (Line No.)	Reduction Rate (%)
	BFD	Gas	Opcodes	Other	Total		
Openrisc 1000	750/ 23K	645/ 18K	1332/ 21K	630/ 19K	3357/ 81K	2000/ 40K	44%/ 51%
Odalrisc	792/ 24K	702/ 20K	900/ 16K	600/ 17K	2994/ 77K	1140/ 18K	62%/ 73%

그림 6 Openrisc 1000과 Odalrisc의 효율성 비교

위의 그림 6을 보면 Openrisc 1000과 Odalrisc를 SoarDL로 작성하였을 때와 Binutils 코드를 직접 고쳤을 때의 라인 수와 워드 크기를 비교한 것이다. 위의 그림 6에서 보듯이 SoarDL로 기술할 경우 Openrisc 1000은 44%, Odalrisc는 62%의 라인 수가 감소하였음을 볼 수 있다. 또한, 워드 크기의 경우 Openrisc 1000과 Odalrisc는 각각 51%와 73%의 워드 크기가 있음을 알 수 있었다.

또한 단순한 라인 수의 감소 뿐만 아니라 Binutils를 직접 수정할 경우 해야 하는 작업이 많다. 예를 들면 새로운 재배치나 명령어를 생성할 경우 SoarDL에 기술하는 것 보다 Binutils를 직접 수정하는 것이 훨씬 더 복잡하다.

5. 결 론

본 논문에서는 SoarDL을 기반으로 한 재겨냥성 어셈블러와 링커를 Binutils로부터 생성하는 것에 대하여 알아보았다. 새로운 프로세서에 대하여 어셈블러와 링커를 생성하려면 많은 시간과 노력이 들지만 SoarDL을 이용한다면 프로세서를 쉽게 기술할 수 있을 뿐만 아니라 자동으로 Binutils의 여러 소프트웨어 개발 도구들이 생성 될 수 있다. 실제로 Odalrisc라는 새로운 프로세서에 대하여 실험을 해 본 결과 효율적으로 생성할 수 있음을 확인하였다.

그러나 여전히 개선해야 할 점은 남아 있다. 먼저는 32 비트 프로세서만을 지원한다는 점이다. 이는 CGEN을 이용하기 때문에 생기는 문제인데 이를 해결하기 위해서는 CGEN에서 생성하는 파일들을 직접 생성하도록 만들어 주어야 한다.

그리고 현재의 SoarDL 기술에서는 메모리를 기술하는데 코드와 데이타의 시작 주소만을 설정할 수 있도록 하고 있다. 그러나 실제 ELF 파일 형식에서는 이외에도 bss, debug, rodata 등의 정보 또한 기록하기 때문에 이러한 영역까지도 시작 주소를 설정할 수 있다면 더욱 효과적일 것이다.

참 고 문 헌

[1] Minwook Ahn, Jooyeon Lee, Yunheung Park, "Optimistic Coalescing for Heterogeneous Register Architectures," *ACM SIGPLAN Notices*, vol.42, Issue 7, pp.93-102, July 2007.

[2] Hoffmann. A, Nohl. A, Braun. G and Meyr. H. 2001, "A survey on modeling issues using the machine description languages LISA," In *Proceeding of the International Conference on Acoustics*, Speech and Signal Processing, vol.2, Salt Lake City, UT, 1137-1140.

[3] Abbaspour. M and Zhu. J, 2002, "Retargetable binary utilities," In *Proceedings of the 39th Design Automation Conference*, New Orleans, LA, 331-336.

[4] Alexandro Baldassin, Paulo Centoducatte, and Asadro Rigo, Daniel Casarotto, Luiz C. V. Santos, Max Schultz and Olinto Furtado, "An Open-Source Binary Utility Generator," ACM Transactions on Design Automation of Electronic Systems, vol.13, no.2, Article 27, April 2008.

[5] Minwook Ahn, Yunheung Paek, "A New ADL-based compiler for Embedded Processor Design," *International SOC Design Conference (ISOCC)*, Seoul, Korea, October, 2005.
[6] 김호균, 정지문, 이종원, 박상현, 윤종희, 백윤흥, "재겨냥성 어셈블러와 링커의 개발", *프로그래밍언어논문지*, 제 22권 제 2호, 2008. 12.
[7] http://www.gnu.org/software/binutils
[8] http://source.redhat.com/cgen
[9] http://opencores.org

김 호 균
2006년 충북대학교 전자공학과(학사). 2006년~현재 서울대학교 전기컴퓨터공학부 석사과정. 관심분야는 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, 어셈블러, 링커

정 지 문
2007년 Jilin University Software Engineering(B.S.). 2007년~현재 서울대학교 전기컴퓨터공학부 석사과정. 관심분야는 임베디드 소프트웨어, 컴파일러

이 종 원
2007년 서울대학교 전기공학부(학사). 2007년~현재 서울대학교 전기컴퓨터공학부 석사과정. 관심분야는 임베디드 소프트웨어, 컴파일러

박 상 현
2004년 서울대학교 전기공학부(학사). 2004년~현재 서울대학교 전기컴퓨터공학부 박사과정. 관심분야는 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, 저전력 설계

윤 종 희
2005년 KAIST 전기및전자공학과(학사) 2005년~현재 서울대학교 전기컴퓨터공학부 석사과정. 관심분야는 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, 재구성 가능 프로세서

백 윤 흥
1988년 서울대학교 컴퓨터공학과(학사) 1990년 서울대학교 컴퓨터공학과(석사) 1997년 UIUC 전산과학(박사). 1997년~1999년 NJIT 조교수. 1999년~2003년 KAIST 전자전산학과 부교수. 2003년~현재 서울대학교 전기컴퓨터공학부 교수 관심분야는 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, MPSoC