논문 22-2-01

재겨냥성 어셈블러와 링커의 개발
(Development of Retargetable Assembler and Linker)

김호균 · 정지문 · 이종원 · 박상현 · 윤종희 · 백윤흥
서울대학교 전기컴퓨터공학부
(hkkim · jmjung · jwlee · shpark · jhyoon@optimizer.snu.ac.kr, ypaek@snu.ac.kr)

요 약

오늘날 Consumer Electronics 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 툴들을 개발 시간은 여전히 개선되지 않고 있다. 이러한 점에서 하나의 프로세서를 Architecture Description Language로 기술하여 이 프로세서에 맞는 소프트웨어 툴을 자동으로 생성하는 것은 중요한 일이다. 이 논문에서 우리는 GNU Binutils 툴을 이용하여 각각의 프로세서에 맞는 소프트웨어 툴들을 자동으로 생성하였다. 이 연구를 통하여 우리는 각 프로세서에 특화된 소프트웨어 툴들을 쉽고 빠르게 생성할 수 있음을 확인할 수 있었다.

1. 서 론

Consumer Electronics 기기에서 임베디드 시스템은 매우 중요한 위치를 차지하고 있다. 임베디드 시스템에서 소프트웨어 툴들과 관련된 문제가 있는데 그 중 하나가 바로 time-to-market 개념이다. 최근에 들면서 새로운 프로세서에 대한 개발 주기가 점차 짧아지고 있다. 그에 따라 프로세서에 특화된 Software Development toolkits (SDKs)를 빠르게 개발하는 것이 새로운 프로세서의 개발시간을 단축시키는데 큰 영향을 주고 있다.

다른 문제로는 임베디드 시스템에서 프로세서의 성능을 높이기 위하여 수작업으로 코드를 생성하였지만 점차 시스템이 복잡해짐에 따라 한계에 부딪치게 되었다. 임베디드 시스템에서 Application Specific Instruction set Processor (ASIP)이나 Digital Signal Processor (DSP)와 같은 프로세서들은 중요한 위치를 차지하고 있다. 전통적으로 ASIP이나 DSP 와 같은 프로세서들은 코드의 성능을 높이기 위하여 수작업으로 그 코드를 작성하여 왔다. 그러나 어플리케이션의 복잡성과 사이즈가 증가함에 따라 이러한 방식은 점차 그 한계에 부딪치게 되었다. 결국 임베디드 시스템의 개발자들은 Architecture Description Language (ADL)와 Retargetable Compiler[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, 시뮬레이트와 같은 다른 Retargetable Software toolkits에 대한 연구는 여전히 필요하다.

어셈블러나 링커를 연구하는데 많이 사용되는 툴은 GNU Binutils 패키지이다. GNU Binutils는 오픈 소스이기 때문에 모든 사람들이 자유롭게 이용 할 수 있고 이미 여러 프로세서들에 대하여 개발이 되어져 있다.

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

이러한 점을 개선하기 위하여 이 논문에서는 새로운 프로세서에 대한 Binutils를 포팅하기 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 툴들을 자동으로 생성해 준다.

이 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

예전부터 Retargetable Assembler, Linker에 대한 연구는 계속해서 수행되어져 왔다. 이러한 소프트웨어 툴들을 만들기 위해서는 프로세서를 기술하는 ADL 에 대한 연구가 중요하다. 왜냐하면 이는 소프트웨어 툴을 생성하는데 가장 기본적인 작업이면서도 ADL을 기술하는 특징에 따라 그 소프트웨어 툴들의 성능이 달라질 수 있기 때문이다.

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

자체적으로 구현하는 방법을 사용하여 Retargetable Assembler와 Linker를 생성하면 원하는 기능을 쉽게 추가하거나 수정하기가 쉽다. 그러나 이러한 방법은 구현된 기능만을 사용할 수 있기 때문에 다른 기능을 사용하려면 처음부터 만들어야 한다는 단점이 있다. 이 방법을 사용한 예로는 CHESS[1]와 LISA[2]등이 있다.

이와는 반대로 Binutils를 이용하면 Binutils에서 제공하는 여러 가지 다양한 기능을 사용할 수 있다. Assembler, Linker 뿐만 아니라 Objdump, Objcopy등 이진 파일을 처리할 수 있는 다양한 기능 또한 사용할 수 있다. 그러나 이 방법의 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. 이러한 방법을 이용한 예로는 Abbaspour and Zhu[2002][3] 이 있다.

그러나 Binutils를 이용하려면 그 내부의 동작에 대해서 파악하고 있어야 한다. 왜냐하면 Binutils가 여러 개발자에 의해서 만들어졌기 때문에 그 구조가 상당히 복잡하고 코드 또한 분석하기가 힘들게 되어있다. Binutils의 이러한 단점을 보완하기 위하여 이 논문에서는 ADL을 이용하여 프로세서에 대해 쉽

게 기술한 후 이렇게 기술된 ADL을 가지고 Binutils의 동작에 필요한 여러 입력 파일을 자동으로 생성하는 것에 대해 보여주도록 하겠다.

3. 본 론

3.1. 새로운 프로세서를 위한 ADL의 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 많은 개발자들이 ADL을 사용하고 있다. 특히 ADL을 특정 포맷에 맞게 기술한다면 소프트웨어 툴을 직업 변경시키는 것 보다도 개발하는 시간이나 노력이 많이 줄어들 것이다. 이를 위해서 본 장에서는 ADL을 어떻게 기술할 것인지에 대해서 간단하게 소개하도록 하겠다.

먼저 프로세서의 하드웨어 특성을 기술하는 방법에 대해서 알아보도록 하겠다.

가. Storage

Storage는 프로세서의 특성 중 가장 기본적인 부분인데 이 부분을 기술하는 것은 중요하다. Storage는 크게 memory와 register로 나눌 수 있다.

아래의 <표 1>과 <표 2>는 각각 memory와 register에 대해서 기술하는 방법이다. 먼저 <표 1>을 보면 cell_width는 memory에서 address로 지정할 수 있는 최소단위의 width를 말한다. 그리고 memory_name으로 이 메모리의 name을 기술한 후 latency와 시작 주소, 그리고 끝 주소를 기술해 준다.

〈표 1〉 Memory 정보 기술

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

다음으로 register 정보를 기술하는 방법은 아래의 <표 2>와 같다. 아래의 <표 2>에서 register_width는 각 register의 개수를 나타내는 것이다. 그리고 이 register의 name을 기술한다. 마지막으로 register_size는 이 register의 bit 단위 크기를 말하는 것이다.

〈표 2〉 Register 정보 기술

register *register_width register_name*; register *register_width* *register_name*[*register_size*];

나. Conventions

이 부분에서는 크게 하드웨어의 endian 정보와 명령어의 크기, 그리고 register의 기능에 대해서 기술한다.

〈표 3〉 Convention 구조

```
conventions
{

words_endian big_endian;
bytes_endian big_endian;
bits_endian big_endian;

base_insn_bitsize 32;
     //The size of instruction when fetching
word_bitsize 32;
     //The number of bits in a word

//stack (software_stack)
software_stack_location DAT0;
```

```
stack_grows        high2low;
stack_section {
        SS_INCOMING_PARAM,
        SS_FPR,
        SS_CALLER_SAV,
        SS_CALLEE_SAV,
        SS_LOCAL,
        SS_OUTGOING_PARAM,
        SS_RET
};
//special registers like stack pointer
programCounter PC PC;
linkRegister r11 LR;
stackPointer r1 SPR;  //  user stack pointer
framePointer r2 FPR;  //  user frame pointer

return_register 0 r11;
return_register 1 r12;

argument_register 0 r3;
argument_register 1 r4;
argument_register 2 r5;
argument_register 3 r6;
argument_register 4 r7;
argument_register 5 r8;
...
}
```

위의 <표 3>에서처럼 Conventions에서는 Endian, 명령어 size, 그리고 special registers 들을 선언한다. base_insn_bitsize와 word_bitsize는 명령어의 크기와 word 단위의 크기를 나타낸다. 하지만 대부분의 RISC프로세서에서는 이 크기가 동일하기 때문에 같은 크기를 갖는다.

스택의 경우 대부분의 RISC프로세서가 최사위 메모리에서 하위 메모리로 내려오는 구조를 취하기 때문에 stack_grows는 high2low를 사용한다. 또한 stack_section은 다음과 같은 순서대로 잡혀지게 된다. Incomming parameter, frame point register, caller saver register, callee saver register, local 변수, outgoing parameter, return value의 순서로 스택에 들어간다.

그리고 special registers들은 아래의 <표 4>과 같은 구조로 기술한다.

〈표 4〉 Special register 구조

programCounter/linkRegister/stackPointer/statusRegister/framePointer *physical_register_name ID*

위의 <표 4>와 같이 program counter, link register, stack pointer register, status register, frame pointer와 같은 special registers를 기술한 후 실제 프로세서에서 사용되고 있는 register의 번호를 기술한다.

지금까지 기술한 Storage와 Convention의 경우 cpu file의 일부를 자동으로 생성하는데 사용된다. 여기에서 cpu file은 CGEN의 입력으로 들어가게 되는데 CGEN에 대해서는 다음장에서 자세하게 알아보도록 하겠다.

다. Binary Utilities 의 구조

이 부분은 어셈블러와 링커를 생성하기 위해 필요한 부분들을 기술한다. 크게 아래와 같이 char, labels, memory, relocation으로 나눌 수 있다.

(1) char 구조

아래의 <표 5> 는 char의 구조를 보여준다. 여기에서 각각의 문자는 어셈블리 파일에서 사용하는 문자를 나타낸다. comment_chars는 어셈블리 파일에서 주석을 나타낼 때 필요한 문자, 그리고 line_comment_chars와 line_separator_chars은 어셈블리 파일에서 라인별 주석을 나타내기 위해서 필요한 문자, 그리고 exp_chars와 flt_chars는 어셈블리 파일에서 exponential과 floating point를

표기할 때 필요한 문자를 나타낸다.

〈표 5〉 Char 구조

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

(2) labels 구조

어셈블리 파일에서 label은 항상 ':' 기호로 끝을 맺는다. 그러나 특수한 프로세서에서는 label은 아니지만 ':' 기호를 사용하는 instruction이 있을 수 있다. 이와 같은 문제를 해결하기 위하여 모든 ':' 기호를 가진 instruction을 여기에 기술하여 label과 구분을 하여 준다.

(3) memory 구조

아래의 <표 6>은 memory의 구조를 보여준다. 여기에서는 Text start address와 Data start address, 그리고 Max page size를 기술하여 준다.

〈표 6〉 Memory 구조

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

(4) relocation 구조

Relocation은 링커가 Object 파일들을 링킹 할 때 reference를 찾지 못한 symbol들에 대하여 linker를 실행 한 후 정확한 주소를 찾아주기 위한 동작을 말한다. 최신의 프로세서일수록 이와 같은 relocation 동작이 더욱 복잡해지고 있으며 이를 쉽고 효율적으로 기술하는 것이 중요한 일이다.

아래의 <표 7> 은 relocation의 기술한 예제를 보여주고 있다. 여기에서 operandtype은 각 relocation들이 operand로 사용되었을 때 어떻게 사용되는 지에 따라 그 type을 정의한 것이고 rightshift는 이 operand가 실제 동작하기 전에 rightshift를 하는 비트수를 나타낸다. 그리고 size는 이 operand가 사용되는 instruction의 size 이고 src_mask와 dst_mask는 각각 source와 destination instruction의 masking되는 bit의 범위를 나타낸다.

〈표 7〉 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 7>에서 operandtype이 LIMM과 HIMM은 각각 Low Immediate와 High Immediate를 나타낸다. 이러한 operand들은 대부분의 RISC architecture에서 사용되고 있는 것으로 메모리의 특정 번지의 값을 읽으려고 할 때 그 메모리 주소값이 너무 커 하나의 Instruction안에 이 주소값을 표시할 수 없는 경우 메모리 번지를 반으로 나누어 특정 register에 저장할 때 사용된다. 특히 이러한 instruction의 경우 많은 프로세서에서는 pseudo-instruction을 사용 하고 있기 때문에 어셈블러나 링커를 수작업으로 생성할 경우 다른 instruction에 비하여 많은 노력이 든다.

그리고 위의 <표 7>에서 operandtype이 REL인 경우는 Relative address를 나타낸다. 이 operand 역시 RISC architecture에서 자주 사용되고 있는 operand로써 특정 번지로 점프할 경우 특정 주소값에서 기본 주소값을 뺀 주소값을 표현할 때 사용한다.

마지막으로 위의 <표 7>에서는 나타나지 않았지만 절대 주소값을 operand로 사용하고자 하는 경우는 operandtype을 DIR로 기술한다. 이는 direct address를 나타낸다.

라. cpu file 의 구조

cpu 파일은 Binutils의 opcodes부분에 필요한 파일들을 생성하기 위한 파일이다. 이 파일의 일부는 자동으로 생성되지만 나머지 부분은 수동으로 작성을 해야 하기 때문에 cpu 파일의 구조를 아는 것은 중요하다.

cpu 파일은 크게 3부분으로 나누어 진다. 먼저, 프로세서의 하드웨어 특성을 기술한 부분이다. 이 부분은 앞에서 기술한 하드웨어 특징에서 자동으로 생성해 준다. 이 부분에서는 프로세서의 이름, 비트 크기, 레지스트 관련 정보등이 하드웨어 정보들이 기록된다.

〈표 8〉 Hardware 정보

```
(include "simplify.inc")

(define-arch
(name openrisc)
(comment "")
(insn-lsb0? #t)
(machs openrisc_mach)
(isas openrisc_isa)
)

(define-isa
(name openrisc_isa)
(base-insn-bitsize 32)
)

(define-cpu
(name openrisc_cpu)
(comment "")
(endian big)
(word-bitsize 32)
)

(define-hardware
(name h_r) (comment "") (attrs)
(type register WI (32))
(indices keyword ""
        ((r0 0)(r1 1)(r2 2)(r3 3)(r4 4)
        (r5 5)(r6 6)(r7 7)(r8 8)(r9 9)
        (r10 10)(r11 11)(r12 12)(r13 13)
        (r14 14)(r15 15)(r16 16)(r17 17)
        (r18 18)(r19 19)(r20 20)(r21 21)
        (r22 22)(r23 23)(r24 24)(r25 25)
        (r26 26)(r27 27)(r28 28)(r29 29)
        (r30 30)(r31 31)(FPR 2)(LR 11)
        (SPR 1)(zero 0)))
)
```

```
(dnh h_hiimm_16 " " () (immediate (INT 16)) ()
() ())
(dnh h_loimm_16 " " () (immediate (INT 16)) ()
() ())
```

위의 <표 8>처럼 하드웨어 정보들은 위에서 기술한 ADL에 의해서 자동으로 생성된다.

다음으로는 opcode와 operand의 필드 정보들이다. 이 필드 정보는 instruction을 기술할 때 사용하기 때문에 반드시 기술해 주어야 한다. 다시 말하면 각 instruction을 기술할 때 필요한 opcode와 operand의 정보를 제공한다고 보면 된다.

〈표 9〉 Field 정보

```
Register fields
(dnf f_REG_0 "" ()  25        5)
(dnf f_REG_1 "" ()  20        5)
(dnf f_REG_2 "" ()  15        5)
...

;Opcode fields
(dnf f_op_0 "" ()   31        1)
(dnf f_op_1 "" ()   30        5)
(dnf f_op_2 "" ()   9         2)
(dnf f_op_3 "" ()   3         4)
(dnf f_op_4 "" ()   30        1)
...

Reserved fields
(dnf f_res_0 "" ()  10        1)
(dnf f_res_1 "" ()  7         4)
(dnf f_res_2 "" ()  25        10)
(dnf f_res_3 "" ()  10        11)
(dnf f_res_4 "" ()  5         2)
...

Immediate fields
(df f_lo16_0 "" ()  15        16 INT #f #f)
(dnf f_imm16_1 "" ()10        11)
(dnf f_imm16_2 "" ()25        5)
(dnf f_uimm6_3 "" ()5         6)
```

```
...

Enums
(define-normal-insn-enum  insn_op_1  "FIXME"  ()
OP1_ f_op_0 (.map .str (.iota 2)))

(define-normal-insn-enum  insn_op_2  "FIXME"  ()
OP2_ f_op_1 (.map .str (.iota 32)))
...

Instruction Operand
(define-operand
(name lo16_0) (comment "")
(attrs)
(type h-sint)
(index f_lo16_0)
(handlers)
)
```

위의 <표 9>는 각 명령어를 기술할 때 필요한 opcode와 operand의 field 정보를 기술해 준다. 이러한 정보는 자동으로 생성되지 않으므로 직접 기술해 주어야 한다.

마지막으로 instruction 부분이다. 이 부분은 위에서 설명한 바와 같이 위에서 기술한 필드 정보를 이용하여 각 프로세서에서 사용하는 instruction에 대해서 기술하여 준다.

〈표 10〉 명령어 정보

```
Instructions
(dni cmov ""
()
"l.cmov $r_REG_0,$r_REG_1,$r_REG_2"
(+  OP1_1  OP2_24  OP21_0  OP3_0  OP22_0  OP4_14
r_REG_0 r_REG_1 r_REG_2  )
()
()
)

(dni add ""
```

```
()
"l.add $r_REG_0,$r_REG_1,$r_REG_2"
(+ OP1_1 OP2_24 OP21_0 OP3_0 OP22_0 OP4_0
r_REG_0 r_REG_1 r_REG_2 )
()
()
)
...
```

위의 <표 10>은 cpu file의 각 명령어를 기술한 것이다. 이 명령어에서 인코딩 정보는 위에서 기술하였던 field 정보를 이용하여 생성한다. 이 부분역시 직접 기술해야 한다.

3.2. 전체 Binutils 의 구조

이번 장에서는 GNU Binutils[4] 툴에 대하여 설명하도록 하겠다. 먼저 전체적인 Binutils의 구조와 내용에 대해서 알아보고 Binutils중 프로세서에 독립적인 부분과 의존적인 부분을 구분하여 Retargetaeble Binutils가 되기 위해서는 프로세서에 의존적인 부분들이 어떻게 바뀌야 하는 지에 대해서 알아보겠다.

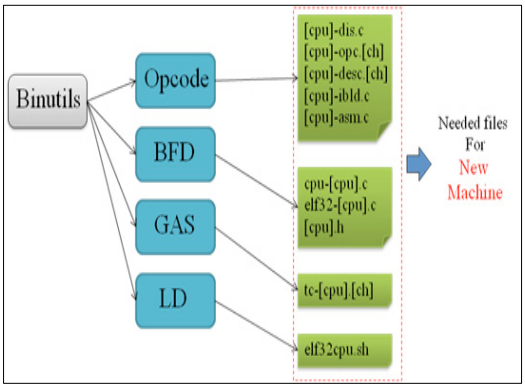
가. Binutils[3] 설명

Binutils는 GNU Binary Utilities의 약자로써 object 파일을 생성 또는 수정하기 위한 여러 가지 프로그래밍 툴들의 모음이라고 할 수 있다. Binutils에 포함된 소프트웨어 툴들은 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 툴들이 제공된다.

아래 [그림 1]은 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.



[그림 1] Binutils 의 전체 구조 및 프로세서 의존적인 부분

(1) Opcode

Opcode는 명령어의 encoding, decoding, simulating 에 필요한 여러 가지 정보를 제공해 준다. 이 부분에서 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 CGEN 이라는 툴을 이용하여 생성한다. CGEN에 대하여는 다음 장에서 설명하도록 하겠다.

(2) Binary File Descriptor library (BFD)

BFD는 여러 가지 object file format과 프로세서 그리고 운영체제에 대하여 지원을 해야 하기 때문에 복잡한 구조를 가진다. 다시 말하면 이 부분이 어셈블러와 링커를 생

성하는데 가장 중요한 부분이라고 할 수 있다. 왜냐하면 여기에서 relocation을 지원하기 때문이다. 여기에서 프로세서에 의존적인 부분은 cpu-[cpu].c, elf32-[cpu].c, [cpu].h 가 있다.

(3) GNU Assembler (GAS)

GAS는 어셈블리 파일을 object 파일로 변환하는 프로그램이다. GAS는 이러한 어셈블리 파일에서 object 파일로 바꾸는 과정에서 주로 3가지 일을 하게 된다. 첫 번째는 어셈블리 파일을 읽어 들여 여기에 맞는 전처리를 해준다. 두 번째는 각 라인별로 명령어를 parsing 한다. 이런 과정에서 symbol의 reference를 찾을 수 없으면 relocation entry를 만들어 준다. 마지막 단계에서는 symbol table 과 함께 최종 object 파일을 만들어 준다. 여기에서 프로세서에 의존적인 부분은 tc-[cpu].cpu 이다.

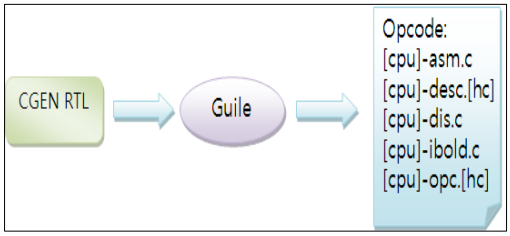
(4) GNU Linker (LD)

LD는 여러 object 파일을 linking 해주고 해당되는 relocation entry에 대하여 처리를 해준다. 그리고 링커가 동작할 때 linker script의 제어를 받는다. 본 논문에서는 여러 가지 실행 파일 format 중 ELF format에 맞게 타겟팅 하였다. 그 이유는 ELF format이 가장 널리 사용되고 있기 때문이다. 여기에서 프로세서에 의존적인 부분은 elf32cpu.sh 파일이다.

나. CGEN[5] 설명

CGEN은 Cpu tools Generator의 약자로써 이 또한 GNU 프로젝트의 일부분이다. CGEN의 목적은 uniform framework을 제공하여 어셈블러, 링커, 시뮬레이터와 같은 프로그램을 쉽게 개발하는 데에 있다. 이와 같은 uniform framework을 실현하기 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 syntax를 참조하고 있다.

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

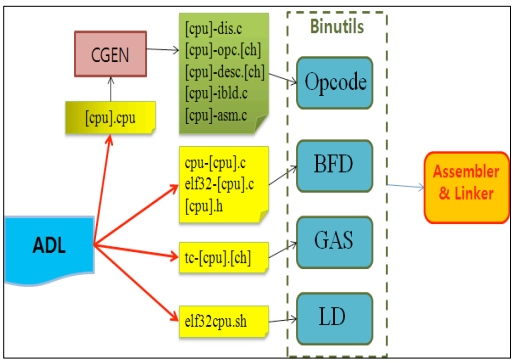


[그림 2] Binutils Opcode 를 위한 CGEN의 지원

위의 [그림 2]에서와 같이 CGEN은 Guile이라는 해석기를 통하여 CGEN RTL로 기술된 파일을 해석하여 Opcode의 프로세서 의존적인 파일들을 생성해 준다. 본 논문에서는 CGEN RTL 부분이 cpu 파일에 해당한다. 따라서 cpu 파일을 입력으로 받아 opcode 부분에 필요한 파일들을 자동으로 생성하여 준다.

3.3. 재겨냥성 Binutils

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



[그림 3] 전체 Binutils 생성 과정

위의 [그림 3]은 Retargetable Binutils를 생성하기 위한 전체 과정에 대하여 보여주고 있다.

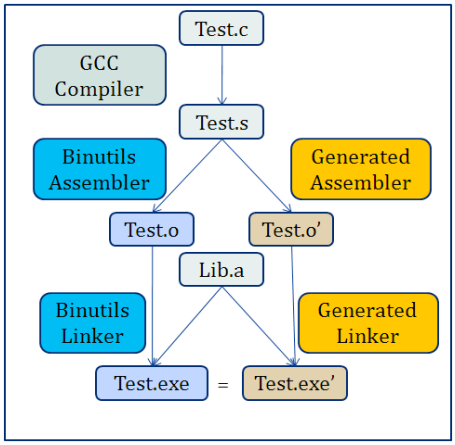
ADL에서 작성한 프로세서의 여러 가지 정보들을 읽어 들인 후 위의 [그림 3]과 같이 프로세서에 의존적인 파일들을 생성한다. 위의 그림에서 특이한 점은 [cpu].cpu 파일인데 이 파일은 다시 CGEN의 입력파일로 들어간다. 다시 말하면 [cpu].cpu 파일이 바로 CGEN RTL 형식의 파일이 되는 것이다. 이 파일을 입력으로 받은 CGEN은 Binutils Opcode에 필요한 파일들을 생성하고 나머지 BFD, GAS, 그리고 LD에 필요한 파일들은 직접 생성이 된다.

4. 구현 및 실험 결과

4.1. 실험 환경

ADL을 이용한 Retargetable 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 ADL로 기술하는 것이 얼마나 유용한지에 대하여 알아보겠다. 프로세서는 Openrisc 1000[6]을 실험하였고 benchmark은 Mediabench인 adpcm, g721, jpeg을 가지고 검증하였다.

Openrisc 1000은 GNU tool에서 지원하는 플랫폼을 설치하여 실험을 하였다.



[그림 4] Openrisc 1000의 실험 환경

실험 환경은 GCC-3.4.4, Binutils-2.17, uClibc-0.9.28, 그리고 Linux-2.6.23이다. 위의 [그림 4]는 Openrisc 1000의 실험 환경을 보여준다. 위의 [그림 4]에서 처럼 GCC compiler를 이용하여 어셈블리 파일을 생성한 후 Binutils에서 지원되는 어셈블러와 링커를 이용한 실행 파일 결과와 SoarDL로 기술된 Retargetable Binutils의 어셈블러와 링커를 이용한 실행 파일의 결과를 비교하였다. 두 개의 실행파일이 동일함을 확인 할 수 있었다. 실험에 이용된 Benchmark에는 adpcm, g721, jpeg 이 사용되었다.

4.2. 실험 결과

이번 장에서는 Openrisc 1000의 실험 결과에 대하여 어떤 효율이 있었는지에 대해서 알아보도록 하겠다.

아래 [그림 5]을 보면 Openrisc 1000을 ADL로 작성했을 때와 Binutils를 손으로 직접

작성하였을 때의 라인수와 파일의 크기를 비교한 것이다. 아래 [그림 5]에서 보듯이 ADL로 기술할 경우 Openrisc 1000은 5%의 라인수, 35%의 파일 크기를 감소하는 효과가 있었다. 다시 말하면, 이는 새로운 프로세서에 대한 어셈블러와 링커를 만드는데 필요한 시간과 노력을 줄일 수 있었다는 것이다.

	Binutils code (Line No./Bytes)					ISA Modeling SoarDL (Line No.)	Reduction Rate (%)
	BFD	Gas	Cpu file	Other	Total		
Openrisc 1000	750/ 23K	645/ 18K	82/ 2K	630/ 19K	2107/ 62K	2000/ 40K	5%/ 35%

[그림 5] Openrisc 1000의 효율성

게다가 Binutils를 직접 수정해야 할 경우 Binutils 전체 코드에 대한 이해가 있어야 한다. 예를 들면 새로운 relocation을 생성 또는 수정할 경우 ADL에 기술하는 것 보다 Binutils를 직접 수정하는 것이 훨씬 더 복잡하다. 이러한 점으로 볼 때 ADL로 기술하는 경우 적은 노력으로도 쉽게 프로세서에 대한 어셈블러나 링커를 생성할 수 있다.

5. 결론 및 향후 연구 과제

5.1. 결론

본 논문에서는 ADL을 기반으로 한 재겨냥성 어셈블러와 링커를 생성하는 방법에 대하여 알아보았다. 기존의 Binutils를 이용하여 직접 새로운 프로세서를 포팅하려면 시간과 노력이 많이 필요하다. 그러나 여기에서 제시한 ADL을 이용하여 Binutils에 필요한 파일을 자동으로 생성한다면 시간과 노력을 줄일 수 있다. 실제로 Openrisc 1000에 대하여 실험을 해 본 결과 적은 노력으로 쉽게 어셈블러와 링커를 생성하는 것을 볼 수 있었다.

5.2. 향후 연구 과제

그러나 여전히 개선해야 할 점은 남아 있다. 먼저 이 논문에서는 cpu 파일 중 일부만을 자동으로 생성해 주고 있기 때문에 cpu 파일을 생성할 때에는 손으로 직접 기술을 해야 하는 불편이 여전히 있다. ADL 기술 중 instruction의인코딩 부분을 추가하여 opcode와 operand의 필드 정보는 물론 instruction까지 자동으로 생성할 수 있도록 만들어야 하겠다.

또한 본 논문의 실험에서 사용된 프로세서가 Openrisc1000 하나이기 때문에 새로운 프로세서에 대하여 실험이 추가되어야 할 것이다.

이러한 점들이 차후에 개선된다면 새로운 프로세서에 대하여 효율적으로 어셈블러와 링커를 생성할 수 있을 것이다.

Acknowledgement

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 국가지정연구실사업(R0A-2008-20110-0), 지식경제부 및 정보통신연구진흥원의 대학 IT연구센터 지원사업(IITA-2008-C1090-0801-0020), 지식경제부 및 정보통신연구진흥원의 IT원천기술개발사업 [과제관리번호: 2006-S-006-02, 과제명: 유비쿼터스 단발용 부품/모듈]의 지원을 받아 수행 되었습니다.

참고문헌

[1] Hartoog, M.R., Rowson, J.A., Reddy, P.D., Desai, S., Dunlop, D.D., Harcourt, E.A., and Khullar, N. 1997. Generation of software tools from processor descriptions for hardware/ software codesign. In Proceedings of the 34th Design Automation Conference. Anaheim, CA, 303-306

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

[3] Maghsoud Abbaspour, Jianwen Zhu "Retargetable Binary Utilities" Design Automation Conference (DAC) 2002, June 10-14, New Orleans, Louisiana, USA

[4] http://www.gnu.org/software/binutils

[5] http://source.redhat.com/cgen

[6] http://opencores.org

김 호 균

hkkim@optimizer.snu.ac.kr

2006년 충북대학교 전자공학과(학사)

2006년~현재 서울대학교 전기컴퓨터공학부 석사과정

관심분야: 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, 어셈블러, 링커.

정 지 문

jmjung@optimizer.snu.ac.kr

2007년 Jilin University Software Engneering(B.S.)

2007년~현재 서울대학교 전기컴퓨터공학부 석사과정

관심분야: 임베디드 소프트웨어, 컴파일러.

이 종 원

jwlee@optimizer.snu.ac.kr

2007년 서울대학교 전기공학부(학사)

2007년~현재 서울대학교 전기컴퓨터공학부 석사과정

관심분야: 임베디드 소프트웨어, 컴파일러.

박 상 현

shpark@optimizer.snu.ac.kr

2004년 서울대학교 전기공학부(학사)

2004년~현재 서울대학교 전기컴퓨터공학부 박사과정

관심분야: 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, 저전력 설계.

윤 종 희

jhyoon@compiler.snu.ac.kr

2005년 KAIST 전기및전자공학과(학사)

2005년~현재 서울대학교 전기컴퓨터공학부 석사과정

관심분야: 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, 재구성 가능 프로세서

백 윤 흥

ypaek@snu.ac.kr

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 부교수

관심분야: 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, MPSoC