마이크로프로세서 설계 무작정 따라하기 part-II (3)

부제: 명령어 시뮬레이터, 어셈블러, 컴파일러의 개발

KAIST 전자전산학과 박사과정 배영돈(donny@ics.kaist.ac.kr), 이종열(jylee@ics.kaist.ac.kr)

이번 강좌에서는 Target description macro 를 작성하는 방법에 대하여 알아보도록 한다. Target machine macro 는 target machine 의 특징을 C 언어의 매크로 형태로 기술하는 것이다. 즉 GNU C 컴파일러의 소스 코드로 사용되는 매크로를 작성하는 것이다. Target description macro 에서 주요 내용으로는 저장 영역의 레이아웃(layout), 자료형의 크기 지정, 레지스터의 용도 지정, 스택의 구성, 주소 지정 방식, 조건 코드 관련 매크로, 어셈블러 출력에 관련된 매크로 등이다. 이번 호에서는 가장 대표적인 매크로들의 정의에 대하여 설명할 것이다. 그러나 지면 관계상 모든 코드를 보일 수는 없으므로 보다 자세한 기술을 위해서는 홈페이지의 소스 코드를 참조하기 바란다.

1. 저장 영역의 레이아웃(layout)

가장 먼저 정의되어야 하는 것은 기본적인 하드웨어의 특징이다. Target description macro 에서 정의되어야 하는 하드웨어 특성으로는 엔디안(Endian), 기본 데이터의 단위 등이다.

```
#define BITS_BIG_ENDIAN 1
#define BYTES_BIG_ENDIAN  1
#define WORDS_BIG_ENDIAN  1
```

SimpleCore 에서는 빅엔디안(big-endian)을 가정하고 있으므로 위의 모든 매크로의 값을 1 로 정의하였다. 즉 WORDS_BIG_EBDIAN 이 1 이므로 워드 내에서 가장 왼쪽의 바이트가 최상위 바이트임을 나타내고 있다.

```
#define BITS_PER_UNIT   8
#define BITS_PER_WORD   16
#define UNITS_PER_WORD  2
```

다음으로 유닛과 워드의 기본 크기를 지정한다. 유닛을 바이트로 생각할 수 있기 때문에, 그 값을 8 로 정의하는 것이 일반적이다. 그리고 SimpleCore 는 16 비트 머신이므로 워드의 크기는 16 으로 정의하였고, UNITS_PER_WORD 의 값은 2 가 된다.

다음으로 정의되어야 하는 것은 메모리에 대한 정보이다. 여기서는 포인터의 크기, 데이터의 정렬(alignment) 크기 등이 정의되어야 한다.

```
#define POINTER_SIZE      16
#define PARM_BOUNDARY     16
#define STACK_BOUNDARY    16
#define FUNCTION_BOUNDARY   16
#define BIGGEST_ALIGNMENT   16
#define BIGGEST_FIELD_ALIGNMENT  16
```

위에서 보는 바와 같이 16 비트 워드 머신에서는 메모리의 크기와 alignment 에 관련된 모든 매크로는 워드의 크기로 정의된다.

2. 자료형의 크기 정의

C 언어 자료형의 기본 크기, 즉 int, float, long 과 같은 자료형의 크기를 비트 단위로 지정하여야 한다.

```
#define CHAR_TYPRE_SIZE   8
#define SHOT_TYPE_SIZE    16
#define INT_TYPE_SIZE     16
#define FLOAT_TYPE_SIZE   16
#define DOUBLE_TYPE_SIZE    32
```

SimpleCore 는 16 비트 기계임으로 위의 매크로들을 적당한 크기로 정의하였다.

```
#define SIZE_TYPE  “int”
#define PTRDIFF_TYPE “int”
```

다음으로 SIZE_TYPE 과 PTRDIFF_TYPE 의 크기를 지정한다. 16 비트 기계이므로 위와 같이 정수형으로 정의하였다. 이 매크로들의 기본값은 “long int”이기 때문에 정의되지 않으면, 내부적으로 long int 로 처리된다. 이렇게 되면 주소 계산에서 문제를 유발할 수 있으므로 반드시 int 형으로 정의되어야 한다.
이제까지 설명된 매크로들 외에도 정의되어야 할 매크로들이 더 있지만 이것은 source code 를 참고하기 바란다.

3. 레지스터의 지정

SimpleCore 에는 16 개의 레지스터가 존재한다. 이들이 어떤 형태로 사용되는 가를 여기에서 설명하여야 한다. GNU C 컴파일러에서는 레지스터들을 몇 개의 class 로 구분하여 각 명령어에서 사용되는 레지스터를 지정할 수 있도록 하고 있다. 즉 각 명령어에 대하여 사용 가능한 레지스터의 class 를 지정할 수 있도록 하고 있다.
각 레지스터에 번호를 부여하기 위하여 다음과 같은 열거형 상수들을 정의하여야 한다. simplecore_regs 는 각 레지스터의 번호로 사용되며, reg_class 는 레지스터의 class 를 지정하는 용도로 사용된다.

```
enum simplecore_regs {
SIMPLECORE_R0,  SIMPLECORE_R1,  SIMPLECORE_R2,  SIMPLECORE_R3,  SIMPLECORE_R4,
SIMPLECORE_R5,  SIMPLECORE_R6,  SIMPLECORE_R7,  SIMPLECORE_R8,  SIMPLECORE_R9,
SIMPLECORE_R10, SIMPLECORE_R11, SIMPLECORE_R12, SIMPLECORE_R13, SIMPLECORE_PC,
SIMPLECORE_CC};
```

```
enum reg_class {
NO_REGS, GENERAL_REGS, PC_REGS, CC_REGS, ALL_REGS, LIM_REG_CLASSES};
```

NO_REGS, ALL_REGS, LIM_REG_CLASSES 는 내부적인 필요에 의하여 반드시 정의되어야 하는 class 들이다. GENERAL_REGS 는 general purpose register 를 포함하는 class 이다. 그러므로 SimpleCore 에서는 R0 ~ R15 까지의 레지스터를 포함한다고 볼 수 있다. PC_REGS 와 CC_REGS 는 각각 PC 와 status register(condition code register)를 포함하는 레지스터 class 이다.

```
#define N_REG_CLASSES (int) LIM_REG_CLASSES
#define REG_CLASS_NAMES   \
{“NO_REGS”, “GENERAL_REGS”, “PC_REGS, CC_REGS”, “ALL_REGS”};
#define REG_CLASS_CONTENTS\
```

```
{
{0x0000},   /* NO_REGS */
{0x3fff},   /* GENERAL_REGS */
{0x4000},   /* PC_REGS */
{0x8000},   /* CC_REGS */
{0xffff} /* ALL_REGS */
};
```

N_REG_CLASSES 와 REG_CLASS_NAMES 를 이용하여 레지스터 class 의 수와 각 class 의 이름을 정의한다. REG_CLASS_CONTENTS 를 이용하여 각 class 에 속하는 레지스터들을 지정할 수 있다. 16 개의 레지스터가 존재함으로 16 비트 수를 이용한다. Class 에 레지스터가 속하면 1 이고, 속하지 않으면 0 이다. PC_REGS 는 PC 만을 포함하고 PC 의 번호는 14 번이기 때문에 14 번째 비트만이 1 이고 나머지는 0 이다. 메모리의 주소를 표시에 사용되는 레지스터는 INDEX_REG_CLASS 와 BASE_REG_CLASS 로 정의되어야 한다. 그런데 SimpleCore 에서는 베이스 레지스터를 사용한 어드레싱은 지원되지만 인덱스 레지스터를 이용한 어드레싱은 지원되지 않으므로 다음과 같이 이 두 매크로를 정의한다.

```
#define BASE_REG_CLASS  GENERAL_REGS
#define INDEX_REG_CLASS NO_REGS
```

다음 호에 소개될 Machine description 에서는 각 레지스터 class 를 구분하기 위하여 제약조건 문자를 사용한다. 따라서 이러한 제약조건 문자에 대하여 다음과 같이 정의하여야 한다.

```
#define REG_CLASS_FROM_LETTER(ch)\
((ch) == 'p' ? PC_REGS :     \
 (ch) == 'c' ? CC_REGS :     \
 (ch) == 'd' ? GENERAL_REGS: NO_REGS)
```

REGNO_OK_FOR_BASE_P 를 사용하여 베이스 레지스터를 정의하도록 한다. SimpleCore 에서는 모든 GPR 이 베이스 레지스터로 사용이 가능함으로 다음과 같이 정의하도록 한다.

```
#define REGNO_OK_FOR_BASE_P(REGNO)  \
(SIMPLCORE_R0 <= (REGNO) && SIMPLECORE_R14 >= (REGN0))
#define RENO_OK_FOR_INDEX_P(REGNO)  0
```

GNU C 의 컴파일 과정 중 하나인 reload 를 위하여 다음과 같은 매크로가 정의되어야 한다. 이 매크로들은 컴파일러의 성능에 미치는 영향이 크므로 주의 하여 정의하여야 한다. 이 매크로들의 정의는 상당이 복잡함으로 여기서는 생략하겠다. (홈페이지의 source code 참조)

3. 레지스터의 용도 기술

레지스터들을 class 로 적절히 분류한 후, 각 class 의 용도를 정의하여야 한다. 먼저 머신에 존재하는 레지스터들의 수를 정의하여야 한다.

```
#define FIRST_PSEUDO_REGISTER  SIMPCORE_CC+1
```

다음으로 레지스터 할당(register allocation)에 이용할 수 없는 레지스터들을 정의한다. SimpleCore 에서는 PC 와 status register 를 제외한 모든 레지스터가 GPR 이므로 다음과 같이 정의한다. 고정된 용도로 사용되는 레지스터(fixed register)의 번호에 해당하는 값을 1 로 표시한다.

```
#define FIXED_REGISTERS {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1};
```

함수 호출이 있는 경우 값이 변경되는 레지스터를 미리 지정하면, 그 레지스터는 레지스터 할당에 사용되지 않는다. SimpleCore 에서는 R12 을 함수의 반환값을 저장하는 용도로 사용하고, R14 는 프레임 포인터(frame pointer)로, R13 은 스택 포인터(stack pointer)로 사용하기 때문에 이들은 레지스터 할당에 사용될 수 없다. 그리고 함수 인자의 전달에 사용되는 R11, R10, R9, R8 도 역시 레지스터 할당에 사용될 수 없다. 따라서 다음과 같이 CALL_USED_REGISTERS 를 정의한다. 그리고 FUNCTION_VALUE 매크로에서 정의되는 레지스터에 대해서는 반드시 1 로 정의하여야 한다.

```
#define CALL_USED_REGISTERS {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1};
```

레지스터에 어떤 값을 저장할 때 필요한 레지스터의 수를 정의하기 위하여 HARD_REGNO_NREGS 를 사용한다. 레지스터의 크기가 모두 같은 SimpleCore 와 같은 경우는 다음과 같이 정의된다. 이 매크로는 레지스터 번호 REG_NO 로부터 시작하는 레지스터에 MODE 크기의 값을 저장할 때 필요한 레지스터의 수를 계산한다.

```
#define HARD_REGNO_NREGS(REG_NO, MODE)  \
((GET_MODE_SIZE(MODE) + UNITS_PER_WORD – 1)/UNITS_PER_WORD)
```

그리고 원하는 크기의 값을 레지스터에 저장하는 것 자체가 불가능할 경우가 있으므로 이 것을 HARD_REGNO_MODE_OK 를 통하여 알려 주어야 한다.

4. 상수의 제한조건 기술

상수의 범위에 따라 제약 조건을 마련하기 위하여 사용되는 매크로를 정의하여야 한다. SimpleCore 에서는 명령어에 포함될 수 있는 상수의 크기는 6 비트 immediate 와 12 비트 offset 이다. 따라서 다음과 같이 CONST_OK_FOR_LETTER_P 를 정의한다.

```
#define CONST_OK_FOR_LETTER_P(VALUE, C)      \
( (C) == ‘I’ ? ((VALUE) >= -32 && (VALUE) <= 31)    \
: (C) == ‘J’ ? ((VALUE) >= -2048 && (VALUE) <= 2047)\
: 0)
```

5. 스택의 구성

함수의 호출 및 복귀에는 일반적으로 스택이 사용된다. 스택의 사용 규칙은 기계마다 달라지게 됨으로 상황에 맞게 정의하여야 한다. 스택 구성에 사용되는 레지스터는 다음과 같다. 스택 포인터로는 R13 이 사용되며, 프레임 포인터로는 R14 가 사용된다. 그리고 R12 가 반환값을 가지는 레지스터로 사용된다. SimpleCore 에서 사용하는 스택은 그림 1과 같다. 이 것은 함수가 호출되고 스택에 관한 각종 사항들의 설정이 완료된 상태이다. 프레임 포인터(FP)는 현재 프레임의 시작을 표시하게 된다. 그리고 스택 포인터는 다음 함수의 스택의 시작을 표시한다.

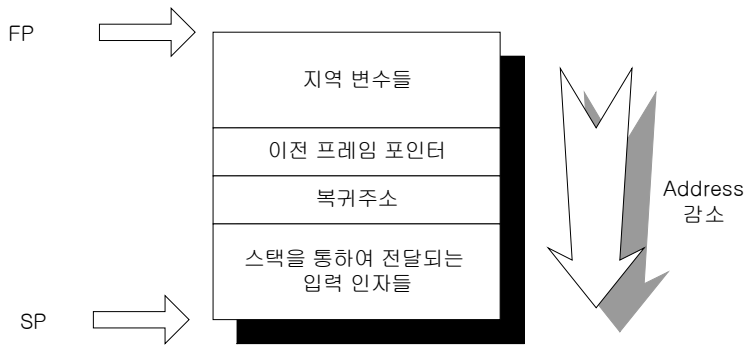

그림 1. SimpleCore 의 스택 구성

스택은 작은 번지로 확장되기 때문에 다음과 같이 정의한다.

```
#define STACK_GROWS_DOWNWARD
#define FRAME_GROWS_DOWNWARD
#define ARGS_GROWS_DOWNWARD
```

위에서 프레임 포인터의 시작과 지역 변수 영역의 시작의 차이가 없고 스택 포인터와 출력인자의 시작 번지가 동일함으로 다음과 같이 정의한다.

```
#define STARTING_FRAME_OFFSET  0
#define STACK_POINTER_OFFSET   0
```

스택의 처리에 필요한 레지스터는 다음과 같은 매크로를 사용하여 정의한다.

```
#define STACK_POINTER_REGNUM SIMPLECORE_R13
#define FRAME_POINTER_REGNUM SIMPLECORE_R14
```

함수가 구조체 값을 반환하는 경우에는 그 구조체를 메모리의 특정 번지에 넣고 그 주소를 R12 를 통하여 전달한다. 따라서 다음과 같이 정의한다.

```
#define STRUCT_VALUE_REGNUM  SIMPLECORE_R12
```

모든 함수는 prologue 와 epilogue 를 가지게 된다. 이들은 함수의 시작과 끝에서 여러 가지 초기화 작업과 종료 작업을 수행하게 된다. 이들은 일반적으로 어셈블리로 작성된다. 이들을 정의하는 매크로는 FUNCTION_PROLOGUE, FUNCTION_EPILOGUE 등이다. 이들에 대해서는 source code 를 참조하기 바란다.

6. 주소 지정 방식

여기서는 SimpleCore 의 주소 지정 방식을 GNU C 컴파일러에 알리는 매크로들을 작성한다. 먼저 주소를 지정하기위해 사용할 수 있는 상수를 정의한다. 일반적으로 다음과 같이 정의하면 무리가 없다.

```
#define CONSTANT_ADDRESS_P(X)  CONSTANT_P(X)
```

다음으로 주소를 나타내는 오퍼랜드에 사용되는 레지스터의 수를 지정하고, 베이스 레지스터로 사용될 수 있는 레지스터의 검사하는 매크로도 정의하여야 한다. SimpleCore 경우에는 간단한 어드레싱 모드만이 지원됨으로 여기에 속하는 매크로들을 쉽게 정의할 수 있다.

7. 어셈블리 출력에 관한 매크로

RTL 코드로부터 어셈블리 코드를 출력하는 매크로들을 작성하여야 한다. 이들 매크로는 상당히 복잡함으로 적당한 함수를 작성하고 이를 매크로를 사용하여 지정하는 방식으로 정의한다. 여기서 정의되어야

하는 매크로들은 레지스터의 이름을 정의하는 매크로 (REGISTER_NAMES), RTL code 로부터 어셈블리 코드를 출력하는 매크로 (PRINT_OPERAND), 주소 오퍼랜드를 어셈블리로 출력하는 매크로 (PRINT_OPERAND_ADDRESS)등이다.

[참고문헌]

- V. Aho, M. R. Sethi, and J. D. Ullman, *Compiler—Principles, Techniques, and Tools*, Addison-Wesly, Reading, MA, 1986.
- R. M. Stallman, *Using and Porting GNU CC for version 2.6*, Free Software Foundation Inc., 1996.
- 황승호, 이대현, 이종열, *컴파일러 개발—GNU C 컴파일러 포팅을 중심으로*, 시그마프레스, 2001

* 다음 회에 계속됩니다. 다음 회에서는 Cross compiler 를 생성하기 위하여 마지막으로 작성되는 Machine description 에 관하여 설명합니다.

본 강좌에서 사용된 예제는 홈페이지(http://www.donny.co.kr/simplecore)를 통해서 제공됩니다.