마이크로프로세서 설계 무작정 따라하기 part-II (4)

부제: 명령어 시뮬레이터, 어셈블러, 컴파일러의 개발

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

이번 강좌에서는 Machine description 을 작성하는 방법에 대하여 알아보도록 한다. Target description macro 의 작성이 끝나면, 이를 바탕으로 machine description 작성을 시작한다. Machine description 은 기계에서 제공되는 명령어들의 종류와 동작에 관련된 정보를 제공하는 역할을 한다.

1. 데이터 전송 패턴의 기술

Machine description 의 작성 과정에서 먼저 처리해야 할 작업은 전송패턴인 '`mov`*m*' 계열의 패턴들을 정의하는 것이다. 가장 기본이 되는 `SImode` 의 전송패턴을 살펴보자.

```
(define_expand "movsi"
 [(set (match_operand:SI 0 "nonimmediate_operand"  "")
       (match_operand:SI 1 "general_operand"      ""))]
 ""
 "
 {
    if ((push_operand(operands[0], SImode) &&
         immediate_operand(operands[1], SImode))
      || (push_operand(operands[0], SImode) &&
         SimpleCore_indirect_disp_operand(operands[1], SImode))
      || (memory_operand(operands[0], SImode) &&
         immediate_operand(operands[1], SImode))
      || (SimpleCore_indirect_disp_operand(operands[0], SImode) &&
          memory_operand(operands[1], SImode))
      || (memory_operand(operands[0], SImode) &&
           SimpleCore_indirect_disp_operand(operands[1], SImode)))
    {
      rtx temp = force_reg (SImode, operands[1]);
      emit_move_insn (operands[0], temp);
      DONE;
    }
 }
 "
)
```

전송 명령어들은 매우 다양할 뿐만 아니라 여러 가지 제한 사항들을 가지기 때문에 대부분의 기계에서 `mov`*m* 패턴은 `define_expand` 를 이용하여 정의한다. SimpleCore 의 경우도 마찬가지이다. 위의 `SImode` 의 전송패턴에서 출력 오퍼랜드로는 `nonimmediate_operand` 를, 입력은 `general_operand` 를 받아들인다. `nonimmediate_operand` 는 `general_operand` 에서 `immediate_operand` 에 해당하는 오퍼랜드를 제거한 것이다. 그 이유는 `immediate_operand` 에 어떤 값이 저장될 수 없기 때문이다. `general_operand` 는 모든 오퍼랜드를 다 받아들인다. 이런 형태의 오퍼랜드가 주어졌을 때, 그 가능은 모든 조합의 오퍼랜드 쌍들을 SimpleCore 가 다 지원하지는 못하기 때문에, 오퍼랜드들을 적절한 변형을 통해 지원할 수 있는 명령어들의 형태로 만들어야 한다.

위의 패턴을 보면, 다섯 가지의 오퍼랜드 조합에 대해 변형을 가하고 있다. 각 경우 모두, 입력 오퍼랜드를 레지스터로 바꾸면, SimpleCore 에서 제공하는 명령어와 일치하는 형태를 지니게 된다. 임시 레지스

터 temp 에 1 번 오퍼랜드를 reload 하고, 그 결과를 0 번 오퍼랜드에 저장하는 전송명령을 출력한다. 한편, 이 다섯 가지에 포함되지 않는 오퍼랜드 조합을 가진 insn 들을 그대로 출력된다.
define_expand 로부터 출력된 전송 명령어들은 이제 다른 패턴에 의해 매칭된다. 앞서 언급했듯이 define_expand 를 통해 생성된 insn 과 매칭되는 패턴이 반드시 머신 설명상에 존재해야 한다. 이와 관련하여 아래와 같이 모두 다섯 개의 패턴들을 machine description 에 정의한다:

```
(define_insn "push_si"
  [(set (match_operand:SI 0 "push_operand"        "")
        (match_operand:SI 1 "nonimmediate_operand"  "d,a,T"))]
  ""
  "push  %1"
  []
)

(define_insn "pop_si"
  [(set (match_operand:SI 0 "SimpleCore_gam_operand"  "")
        (match_operand:SI 1 "push_operand"  ""))]
  ""
  "pop  %1"
  []
)
```

먼저, push 명령과 pop 명령에 매칭되는 패턴을 정의한다. 앞서 movsi 패턴에 의해서 push_operand 와 다른 오퍼랜드로 이루어진 전송 insn 이 생성될 수 있으며, 이 insn 은 push 또는 pop 동작을 뜻하므로 이 패턴들에 의해 push 또는 pop 명령어가 출력된다. 그러나 SimpleCore 에는 push 나 pop 명령어가 없기 때문에 synthetic 명령어가 사용된다. Synthetic 명령어란 machine 에서는 지원되지 않는 명령어 이지만, 여러 개의 머신 명령어를 사용하여 구현할 수 있는 명령어이다. 따라서 일종의 매크로와 같은 역할을 하는 것으로 볼 수 있다. 이러한 synthetic 명령어들은 어셈블러에 의하여 머신 명령어로 변환되어 어셈블된다. push 명령어는 stack pointer 를 감소시키고 stack pointer 가 가리키는 메모리 위치에 오퍼랜드의 값을 저장하는 명령어로 치환되어 어셈블된다.

```
(define_insn "SimpleCore_ldi_si"
  [(set (match_operand:SI 0 "register_operand" "=a,a,d,d")
        (match_operand:SI 1 "immediate_operand" "N,S,N,S"))]
  ""
  "mov   %0, %1"
  []
)
```

1 번 오퍼랜드가 immediate 오퍼랜드이면, 이 패턴에 매칭된다. SimpleCore 에서 immediate 값을 로드하는 명령어 mov 가 출력된다.

```
(define_insn "SimpleCore_movsi"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=U,U, a,d, a,a,a, d,d,d, T,T,T, c,c,
a,d, a,r, e,e")
        (match_operand:SI  1 "nonimmediate_operand" "a,d,  U,U,  a,d,T,  a,d,T,  a,d,T,  a,d,
c,c,  e,e,  a,r"))]
  ""
  "mov   %0, %1"
  []
)
```

앞의 세 패턴과 매칭되지 않는 전송 insn 들은 위의 패턴에 매칭되어 mov 명령어가 출력된다. DImode, 즉 32 비트 모드에 대해서도 비슷한 방식으로 전송패턴을 정의한다. 여기서 달라지는 것은 pattern matching 과정에서의 출력 템플리트(template) 작성 방식이다. 예를 들어, 바로 위에 패턴에 해당되는 DImode 용 패턴은 다음과 같다.

```
(define_insn "SimpleCore_movdi"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=U,U, a,d, a,a,a, d,d,d, T,T,T, c,c, a,d")
        (match_operand:DI 1 "nonimmediate_operand" "a,d, U,U, a,d,T, a,d,T, a,d,T, a,d, c,c"))]
  ""
  "mov   %0, %1\;mov   %d0, %d1"
  []
)
```

앞선 패턴에서는 mov 명령어 하나만 출력했으나, 여기서는 레지스터 두개를 하나의 32 비트 레지스터로 간주하게 되기 때문에 상위 레지스터와 하위 레지스터에 대해 별도로 mov 명령어를 출력해야 한다. 특히 두번째 mov 명령어 출력과 같이 'd' 지시자를 통해서 연속된 레지스터의 하위 레지스터를 표현하는 방법을 잘 익혀두자.

SimpleCore 의 경우 기본적으로 고정소수점 프로세서이기 때문에, 부동소수점 처리에는 다소 복잡한 방법들이 사용된다. 이에 대해서는 소스 코드를 참조하기 바란다.

2. 산술 연산 패턴의 기술

가장 대표적인 산술 연산 패턴인 addm3 패턴은 덧셈 명령 뿐만 아니라 메모리 주소 계산에도 이용되기 때문에 조심스럽게 잘 정의해야 한다. SimpleCore 의 경우 다음과 같이 정의한다:

```
(define_expand "addsi3"
  [(set (match_operand:SI 0 "register_operand" "")
     (plus:SI (match_operand:SI 1 "register_operand" "0")
         (match_operand:SI 2 "SimpleCore_gam_immediate_operand"  "")))]
  ""
  "
  {
    if (!reload_in_progress && !reload_completed) {
    if (immediate_operand(operands[2], SImode)){
        if (!register_operand(operands[0], SImode)
          ||!register_operand(operands[0],SImode)|| INTVAL(operands[2])>0x03f) {
            rtx temp = force_reg (SImode, operands[2]);
            emit_insn (gen_addsi3 (operands[0], operands[1], temp));
            DONE;
        } else if (INTVAL(operands[2]) < 0) {
            rtx temp=force_reg(SImode,gen_rtx(CONST_INT,VOIDmode,-
                                        INTVAL(operands[2])));
            emit_insn (gen_subsi3 (operands[0], operands[1], temp));
            DONE;
          }
      }
    }
  }
  "
)

(define_insn ""
[(set (match_operand:SI 0 "SimpleCore_gam_operand"    "=a,a,a, d, T")
       (plus:SI (match_operand:SI 1 "SimpleCore_gam_operand"  "0,0,0, 0, 0")
        (match_operand:SI 2 "SimpleCore_gam_operand"  "a,d,T, a, a")))]
```

```
  ""
  "add   %0, %2"
  []
)

(define_insn "SimpleCore_addsi"
  [(set (match_operand:SI 0 "register_operand"          "=a,d")
        (plus:SI (match_operand:SI 1 "register_operand"   "0,0")
                 (match_operand:SI 2 "immediate_operand"  "")))]
  ""
  "add  %0, %2"
  []
)
```

reload 단계에서 재귀적으로 계속 호출이 일어날 염려가 있기 때문에, `reload_in_progress` 와 `reload_completed` 를 이용하여 reload 단계에서는 명령어 확장 작업이 일어나지 않도록 막고 있다. 여기서는 2 번 오퍼랜드에 따라서 다른 처리를 하고 있다. 2 번 오퍼랜드가 7 비트를 초과하는 immediate 이면 SimpleCore 의 `add` 명령으로는 처리할 수 없다. 따라서 이 값을 일단 레지스터로 reload 하고 `gen_addsi3` 를 호출하면 다시 이 패턴이 호출된다. 하지만 이때는 이미 레지스터로 reload 된 상태이므로 그대로 통과하게 된다. 한편, immediate 가 음수인 경우는 그것을 양수로 바꾼 다음, `gen_subsi3` 를 호출하여 `subsi3` 에 해당되는 패턴을 생성한다.

`addsi3` 패턴을 통해 생성된 패턴은 `SimpleCore_addsi` 패턴에 의해 매칭되어 해당 어셈블리 코드가 출력된다.

32 비트의 덧셈, 즉 SimpleCore 에서 long 형의 덧셈은 해당 명령어가 없기 때문에 라이브러리 함수 호출을 이용해야 한다.

```
(define_expand "adddi3"
  [(set (match_operand:DI 0 "register_operand" "")
        (plus:DI (match_operand:DI 1 "register_operand" "")
                 (match_operand:DI 2 "register_operand" "")))]
  ""
  "
  {
    if (!SimpleCore_adddi3_libcall)
      SimpleCore_adddi3_libcall = gen_rtx (SYMBOL_REF, Pmode, ADDDI3_LIBCALL);
    emit_library_call (SimpleCore_adddi3_libcall, 1, DImode, 2,
                       operands[1], DImode,
                       operands[2], DImode);
    emit_move_insn (operands[0], hard_libcall_value(DImode));
    DONE;
  }
  "
)
```

모든 오퍼랜드를 `register_operand` 로 받고 있다. 이 패턴의 처리 방식은 라이브러리 호출에 있어서 전형적으로 사용되므로 잘 기억해두길 바란다. 기타 다른 연산 패턴들도 위와 같은 방식으로써 작성할 수 있을 것이다.

3. 비교 및 분기 명령어 패턴의 기술

이어서, 비교 및 분기 명령어 패턴을 정의하는 방법을 살펴보자. SimpleCore 는 조건코드가 GCC 에서 표준적으로 요구하는 것보다 적기 때문에 이를 처리하게 위해서 몇 가지 트릭을 사용한다.

`cmpsi` 패턴은 다음과 같다:

```
(define_expand "cmpsi"
  [(set (cc0)
     (compare (match_operand:SI 0 "register_operand"            "")
         (match_operand:SI 1 "SimpleCore_gam_immediate_operand" "")))]
  ""
  "
  {
    if (operands[0])           /* avoid unused code message */
      {
        branch_cmp[0] = operands[0];
        branch_cmp[1] = operands[1];
        DONE;
      }
  }
  "
)
```

여기서는 단순히 오퍼랜드들을 미리 변수 branch_cmp 에 저장만 하고 실제 insn 은 생성하지 않고 끝난다. 그리고 다음과 같이 b*cond* 패턴에서 define_expand 를 이용, 비교 insn 을 출력한다.

```
(define_expand "beq"
  [(set (cc0) (compare (match_dup 1) (match_dup 2)))
   (set (pc)
       (if_then_else (eq (cc0) (const_int 0))
                     (label_ref (match_operand 0 "" ""))
                     (pc)))]
  ""
  "
  {
    operands[1] = branch_cmp[0];
    operands[2] = branch_cmp[1];
  }
  "
)
```

여기서 생성되는 첫번째 insn 은 다음의 패턴에 매칭되어 cmp 명령이 출력된다:

```
(define_insn ""
  [(set (cc0)
     (compare (match_operand:SI 0 "register_operand"            "a,a,a,d,a")
       (match_operand:SI 1 "SimpleCore_gam_immediate_operand" "a,d,T,a,N")))]
  ""
  "@
  cmp   %0, %1
  cmp   %0, %1
  cmp   %0, %1
  cmp   %0, %1
  cmp  %0, %1"
  []
)
```

그리고, 두번째 insn 은 다음의 분기 패턴에 매칭되어 br 명령이 출력된다:

```
(define_insn "jump"
  [(set (pc) (label_ref (match_operand 0 "" "")))]
  ""
  "br    %0"
  []
)
```

SimpleCore 에서는 조건 코드 중에는 `bge` 에 사용할 수 있는 코드가 없다. 따라서 비교 오퍼랜드를 서로 바꾸어 주거나 SimpleCore 에서 지원하는 조건 코드를 사용하여 비교하는 명령어들을 출력하여야 한다. 비교할 오퍼랜드들이 모두 레지스터면 그냥 바꾸기만 하면 되지만, 한쪽이 immediate 인 경우는 해당하는 어셈블리 명령어가 없기 때문에, 이 오퍼랜드는 레지스터로 reload 하고 나서 서로 교환하는 것을 볼 수 있다. `ble`, `bgeu`, `bleu` 역시 모두 비슷한 방식으로 처리를 하면 된다.

4. 함수 호출 패턴의 기술

`call` 패턴 및 `call_value` 패턴은 아래와 같이 작성한다. 주소는 `symbol_reg` 표현식이나 `label_ref` 표현식만 허용된다. SimpleCore 에는 call 명령어가 없기 때문에 이 명령어 역시 synthetic 명령어이다. 이 명령어는 stack pointer 를 변경하는 명령어와 호출되는 함수의 위치로 분기하는 branch 명령어로 구성된다. 여기에 설명되지는 않았지만 return 패턴 역시 "`ret`"라고 하는 synthetic 명령어를 출력하도록 기술되어야 한다.

```
(define_insn "call"
  [(call (match_operand:SI 0 "SimpleCore_call_address_operand" "")
         (match_operand:SI 1 "immediate_operand" "i"))]
  ""
  "call  %0"
  []
)

(define_insn "call_value"
  [(set (match_operand:SI 0 "register_operand" "=a,d")
    (call (match_operand:SI 1 "SimpleCore_call_address_operand" "")
          (match_operand:SI 2 "general_operand" "g,g")))]
  ""
  "call  %1"
  []
)

int
SimpleCore_call_address_operand (rtx op, enum macdine_mode mode)
{
  if (GET_CODE(op) != MEM)
    return 0;

  switch (GET_CODE (XEXP(op, 0)))
    {
    case SYMBOL_REF:
    case LABEL_REF:
      return 1;

    default:
      return 0;
    }
}
```

[참고문헌]

- V. Aho, M. R. Setdi, 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

* 이번 회를 마지막으로 compiler 포팅에 관한 연재를 마칩니다. 이어지는 회에서는 어셈블러의 구현에 관하여 설명됩니다.