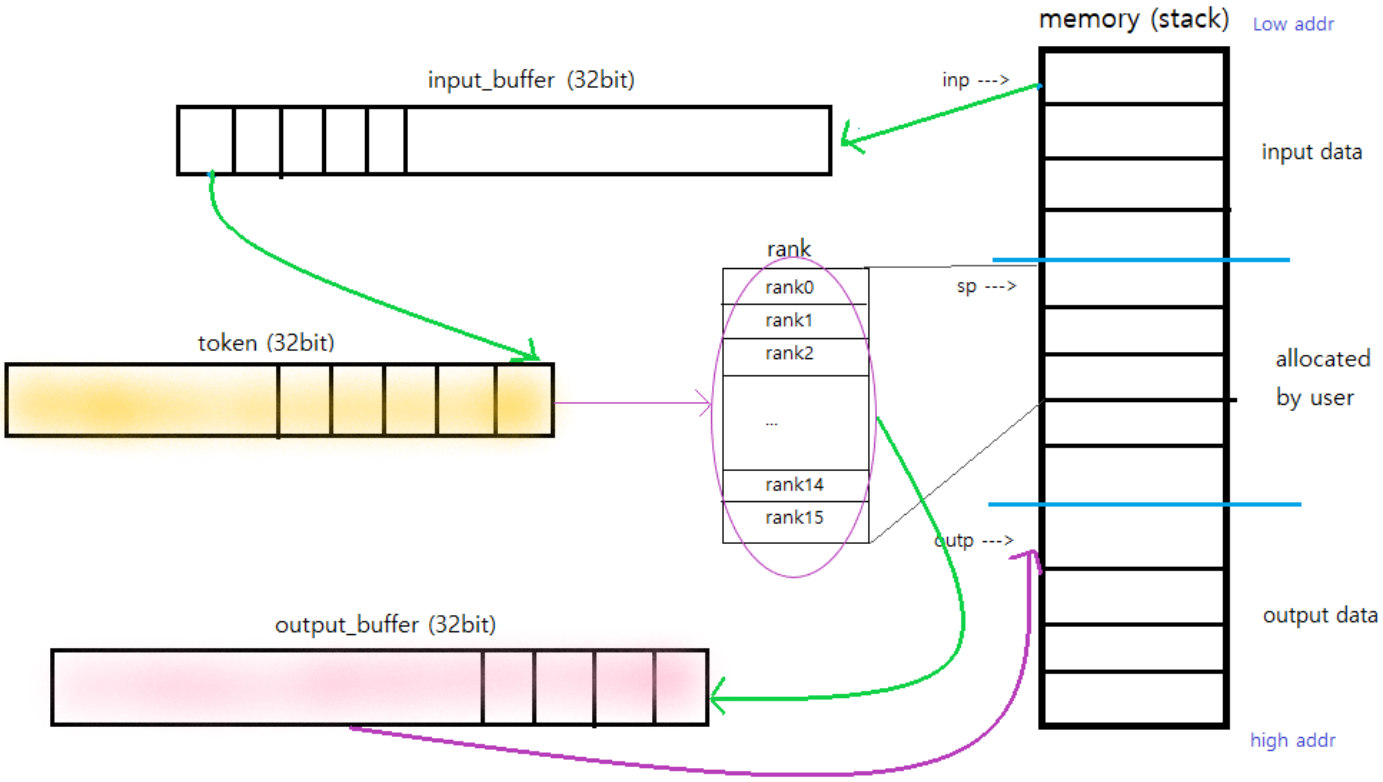
Computer Architecture Project Assignment 3

- decoding by pyrisc

자유전공학부 2012-13311 안 효 지

0. 개요



Input data가 일렬로 연속해서 주어지지 않고 메모리에 4byte 단위로 저장되어 있다. 한 번에 전체 데이터를 불러와 일렬로 정렬시켜놓고 디코딩을 하면 편하겠지만 가용 레지스터가 정해져 있기에 load word -> decoding -> store word를 반복하는 방식으로 알고리즘을 짰다. 디코딩 알고리즘의 큰 구조는 위와 같다.

1) inp가 가리키는 주소에 담긴 data를 input buffer(a3 reg)로 불러온다. Inp는 +4를 하여 그 다음 input 주소를 가리키도록 한다. 불러온 후 little endian을 big endian으로 convert한다.

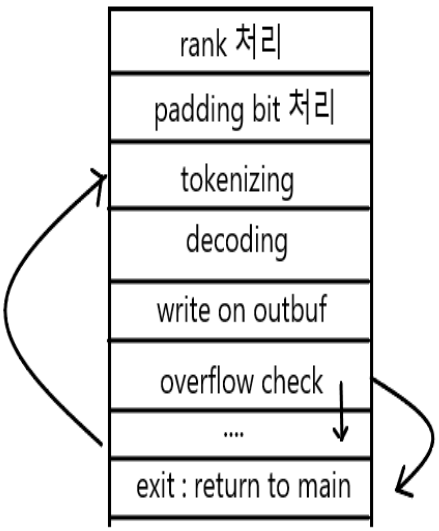
2) (padding info제외) msb를 읽어 token reg로 보낸다.

3) token reg에 정해진 수의 bit가 들어오면 stack에 들어있는 rank와 비교하여 해당하는 값을 output buffer에 넣는다.

4) output buffer가 다 차면 endian convert를 하여 outp가 가리키는 주소에 값을 저장한다. 저장 후 outp를 업데이트 해준다.

이 과정을 input data가 다 없어질 때까지 반복한다.

끝나는 시점, input buffer를 채우는 시점, output buffer를 비우는 시점을 판단하기 위해 각각 'bits_should_be_read', 'bits in inputbuf', 'empty bits in outbuf'라는 변수를 사용한다.



1. Ranking 저장하기

프로그램이 시작되면 바로 sp-128을 하여 공간을 할당해준다. 맨 처음 inp가 가리키는 주소의 4byte data는 rank이다. 따라서, 이 값을 버퍼로 읽어와 big endian으로 convert한 후, rank0 ~ rank15까지 stack에 저장한다. 처음엔 자주 쓰이는 rank0~7를 하나의 레지스터에 저장해 놓는다면 token의 값을 ranking을 찾을 때 memory access를 하지 않고 bitwise연산만으로 가능할 것이라 생각했었다. 하지만 본 과제에서는 가용 레지스터의 수가 제한되어 있기에, 이 과정에서 레지스터를 이용하려면 다른 값을 메모리에 저장해야 하는 조삼모사가 발생할 것이라 판단하여 모든 rank를 각각 0(sp) ~ 60(sp)에 저장하는 방법을 사용하였다.

2. Padding info 처리하기

Ranking 다음 input은 4bit의 padding info와 28bit의 code이다. Inbyte에 *8을 하여 들어오는 인풋의 길이를 bit로 만들어놓은 'bits_should_be_read'에서, 앞서 처리한 rank 32bit, padding info 4bit, padding bit의 값을 빼준다. 이로써 본격적인 decoding을 시작하기 위한 처리가 다 되었다.

앞으로 진행될 과정에서 꼭 지켜야 할 규칙은 세 가지가 있다. 1) data를 읽는 도중 input buffer가 비면 즉시 lw를 해서 채워줄 것. 2) outbuf가 꽉 차면(빈 자리가 없으면) sw를 해서 비워줄 것. 3) outbuf가 다 차지 않았어도 모든 input data를 처리한 경우에는 outbuf에 있는 값을 메모리에 sw할 것이다.

본격적인 Decoding 시작

3. Tokenizing

Inbuf에 남아있는 비트를 기준으로 판단했다. 1) 5bit 이상 남아있으면 msb 0이면 3개를 읽어서 token에 넣고, msb1인 경우엔 그 다음 비트를 보고 0이면 4bit를, 1이면 5bit를 읽어서 token에 넣었다. 2) 4bit 남아있으면 msb 0 이면 3bit를 읽고, inbuf의 값이 0b1100~~ 보다 작으면 4bit를 읽었다. 큰 경우에는 일단 4bit를 읽어서 token에 넣어놓은 다음, load를 하고 msb 1bit를 token에 넣어주었다. 3) 3bit가 남아있으면 일단 3bit를 token에 받아놓고 크기비교를 통해 로드 후 1bit를 더 받을지, 2bit를 더 받을지 결정한다. 4) 2bit가 남아있으면 일단 2bit를 받아놓고 마찬가지로 크기비교를 통해 최종적으로 3, 4, 5bit 중 얼마나 받을지를 결정한다. 5) 1bit가 남아있는 경우도 마찬가지로 한다.

4. Decoding (match with rank)

Token의 값을 통해 몇 번째 rank의 값을 가져야하는지를 판단한다. token의 값이 0, 1, 2, 3, 8, 9, 10, 11..... 이렇게 정해져 있기 때문에 token ==0 -> lw rd, 0(sp)처럼 하드코딩을 해서 decoding의 결과를 받는다.

5. Write on outbuf

받은 결과값을 outbuf에 넣는다. 한 번 이 작업이 수행될 때마다 outbuf에는 4bit씩 들어오므로 empty bit을 계산하기도 편하다. 값을 받기 전에 <<4를 해주어야 한다. 또한, 수행횟수마다 outlen++를 해주면

4bit당 길이가 1이 늘어나므로 나중에 >>2 를 하여 실제 결과값의 byte수를 return value에 담을 수 있다.

6. Overflow check

Outlen을 계산한 직후, outlen이 outbytes보다 크면 바로 -1을 리턴해주도록 한다(여기가 80점에서 100점으로 올라가는 곳이었다.). 원래는 맨 마지막에만 이것을 위치시켰으나, main함수를 살펴보니 output을 쓸 수 있는 공간이 outbytes만큼만 할당되어 있는 것을 확인하였다. outlen이 outbytes를 초과한 상태에서 메모리에 output을 쓰는 것은 할당되지 않은 공간에 접근하는 것을 의미한다. 따라서 오류가 생길 것이라고 판단, 수정했다.

7. 모든 input을 읽었는지 확인

하나의 token을 읽어 처리한 후, Outbuf가 다 찼는지, input buf가 비었는지 확인하기 전에 모든 input을 읽었는지를 먼저 확인하면 불필요한 작업을 줄일 수 있다. 이 경우엔 현재 outbuf에 담겨있는 값을 메모리에 마저 쓰고, outlen을 리턴하면 된다.

그 후에 outbuf가 다 찼으면 메모리에 써주고, 관련 변수를 초기화 시킨다. Input buf가 비어있는 경우에는 다시 채워주고 3번의 tokenize 작업으로 돌아가 반복한다.

8. Check Last buf

들어오는 input data의 종류를 크게 세 가지로 나누어보았다.

1)[rank] [end info + codes (<= 4byte)]

2) [rank] [end info + codes] [codes (< 4byte)]

3) [rank] [end info + codes] [codes] …… [codes] [codes (<=4byte]]

지금까지 작성한 루프를 한 번 반복한 경우에는 우리의 testcase에 있는 ca2020과 같은 1)의 경우는 이미 완성이 된다. 또한, 3) [rank] [end info+codes] [codes] [codes]….. 인 것들도 앞으로 루프를 돌면서 커버가 될 것이다. 하지만 2)와 같이 마지막으로 buffer에 들어온 값 중에 byte align을 위한 쓰레기값이 들어있는 경우에는 그 값들은 제외하고 decoding을 시키는 처리를 해주어야 한다. 이를 위해 'bits_in_inputbuf'를 'bits_should_be_read'의 값으로 바꾸어주어야 하는데, 'bits_in_inputbuf'는 lw할 때마다 32로 자동으로 초기화를 시켜놓았기 때문이다. 실제 남은 bit를 기준으로 종료시점을 잡아야 오류가 안 생긴다.

9. Exit

main으로 return하기 전에 stack dealloc을 하고, 초반에 저장해놓았던 ra를 다시 ra register에 담아준다.

10. Register 사용법

	a0	a1	a2	a3	a4	a5
loop	token	shoulberead	bits inbuf	-		bigendian
seq read	token	〃	〃	-		〃
load	token	〃	〃	load받은거	inp addr	〃
convert	token	〃	〃	load받은거		'
tokenizing	token	〃	〃	-		〃
decoding	token	〃	〃	outbuf		〃
write outbuf	outlen	〃	〃	outbuf	emptybit_outbuf	〃
check etc	outlen	〃	〃	outbuf		〃

각 단계마다 필요한 변수들을 나열한 후, 최대한 동일 register에 머물도록 하였다. 가령 a0에는 decoding까지는 token값이 들어있다가, 그 후 token이 필요 없어지면 작성한 코드 말미까지는 outlen을 넣어놓고, 다시 loop의 처음으로 돌아가 decoding을 해야 하면 outlen은 mem에 넣어놓고 token을 초기화해서 사용한다. a1과 a2에는 처음부터 끝까지 각각 'bits should be read'와 'bits in inputbuf'가 들어있다. convert에서는 a3에 들어온 값을 a5로 내보내게 설계해놓았기 때문에 convert가 필요한 lw도 a3으로만 받았고, 나중에 convert를 거친 후 outp 주소에 쓰여져야 할 outbuf도 a3을 사용하도록 했다.

대략적인 register 배치는 위 표와 같다. 세부적인 것은 코딩하면서 조절하였다.