Computer Architecture Project 3 Report

2019-19222 Ham Taewon

My implementation of decode.s is divided into three parts: 1) reading rank table & end info, 2) decoding while reading input, and 3) decoding after all input is read.

0) Stack structure & Additional functions

address	data
88(sp)	return address
84(sp)	outbytes (argument 3)
80(sp)	'outbytesleft'
76(sp)	outp (argument 2)
72(sp)	inbytes (argument 1), 'bitsleft'
68(sp)	inp (argument 0)
64(sp)	'loadoffset'
60(sp) ~ 0(sp)	rank table - entry r is stored at 4*r(sp)

function	description
toggle_endian	toggles a0's endian mode

1) Read rank table & end info

1-a) Fill in first half of rank table

First word of inp has entries 0 to 7 of rank table, but in little endian. Load it and toggle endian mode, then store it into 0(sp) ~ 28(sp).

1-b) Fill in rest of rank table

Find what numbers aren't already in the rank table, and add them in order. A double loop is used to search each number from 0x0 to 0xf inside entries 0 to 7.

After this, shift all entries left by 28 bits, so that the code is now in bits 28 ~ 31. This is for future convenience.

1-c) Read end info

End info is saved in second word of inp. Simply read it.

2) Decode while reading input

2-a) Introduce buffers

Since reading, decoding, and writing one code at a time is wasteful, we instead introduce buffers named 'inputstream' and 'outputstream.' registers a3 and a4 will be used to make a 64-bit input buffer, and register a2 will be used to make a 32-bit output buffer. This way, we can load an entire word from inp and decode it little by little. We can also store the result of decoding 8 codes into the output buffer and then write the entire 32-bit word to outp.

For example, suppose *(inp + 8) = 0x88888888, *(inp + 12) = 0xffff9999, and we were in the middle of decoding *(inp + 8). The buffers would look like this:

	a3	a4
inputstream	10001000 10001000 10001001 10011001	10011111 11111111 11110000 00000000

	a2
outputstream	00000000 00000000 00000000 00000000

Suppose the code `1000` is decoded into `0011`, and code `1001` is decoded into `1111`. We read each code by looking at the leftmost 5 bits of `inputstream`, determining the code(in this case, `1000`), and shifting `inputstream` to left (in this case, it needs to be shifted 4 bits left). The code is decoded(to `0011`), and stored in `outputstream`(to the correct position in little endian format)

After 8 cycles of decoding, the buffers would look like this:

	a3	a4
inputstream	10011111 11111111 11110000 00000000	00000000 00000000 00000000 00000000

	a2
outputstream	11111111 00111111 00110011 00110011

Now `outputstream` correctly contains the result of decoding 8 codes, and can be written to `outp`. Also, note that (number of empty bits in input buffer) ≥ 32. In this case, we can read another word from `inp`, namely `*(inp + 16)`, and append it to the end of the current input buffer (this is called 'refilling' in the code). After this, we can decode for 8 cycles again.

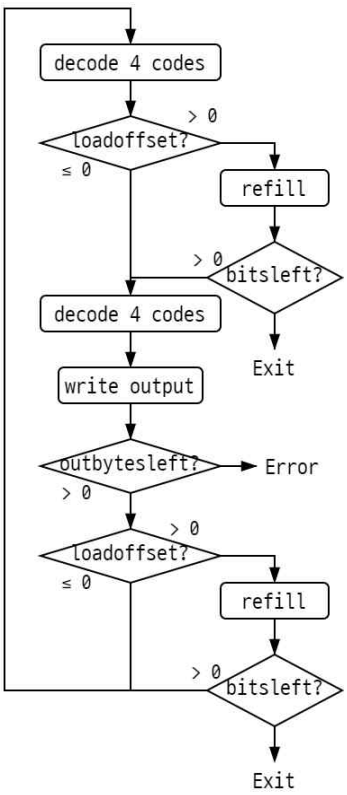
2-b) Things to consider & Complete diagram

To know when to refill the buffer, we store '`loadoffset`' in `64(sp)`, which always contains (number of empty bits in input buffer – 32). Checking loadoffset everytime is wasteful, so it's only checked once every 4 cycles. This is fine because maximum number of bits decoded in 4 cycles is 4*5 = 20 < 32.

Since `a3` and `a4` are 2 separate registers, we need to do additional operations to align them like a single 64-bit register. This aligning is also done once every 4 cycles.

To know if there are more bits to read, we store '`bitsleft`' in `72(sp)`, which always contains (number of bits left to read which aren't in `inputstream` yet). To know if there is more space left to write, we store '`outbytesleft`' in `80(sp)`, which always contains (number of bytes left to write).

The diagram showing the order of these instructions is to the right.



3) Decode after all input is read

After exiting the loop because `bitsleft` ≤ 0, the remaining bits in the buffer have to be read without additional refills until there are no remaining bits. One difference here is that the decoding might be over before decoding 8 cycles, so we have to check actual bits left in every cycle. Since this part is only executed a few times, I didn't pay much attention to optimizing it, and just did the various updating and aligning every cycle.

After the final decoding is done, all we need to do is to calculate how many bytes were actually written. This is calculated by subtracting `outbytesleft` from `outbytes`.