Project #3
RISC-V Assembly Programming

4190.308 Computer Architecture (Fall 2020)

Sangjun, Son

1 Introduction

RISC-V Assembly Programming을 이용해 프로젝트 #1에서 구현한 Compressing Data with Huffman Coding의 반대인 Decoder를 구현합니다. 구현 함수는 `decode()` 이며 함수 명세는 아래와 같습니다. 이번 프로젝트는 C 언어가 아닌 어셈블리어로 코딩을 합니다.

```
int decode(const char *inp, int inbytes, char *outp, int outbytes);
```

Simplified Huffman Decoding을 위해 rank table을 구하고 encoded data가 해당하는 부분이 어떤 글자로 인코딩이 되어있는지 역으로 계산하여 Output pointer에 해당하는 메모리에 저장하게 됩니다.

2 Implementation

Environment

32-bit RISC-V processor에서 RV32I based assembly code를 작성할 것입니다. Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz에서 구현을 하였습니다.

Control Flow

프로그램에서 동작하는 알고리즘은 다음과 같습니다.

1. `inbytes`가 0 인 경우 바로 0 을 반환합니다.
2. 입력된 모든 매개변수 (`inp`, `inbytes`, `outp`, `outbytes`)를 stack memory에 저장합니다. 이 때 `inbytes`와 `outbytes`를 Nibble (4 bits) 형태로 바꾸어 2배 한 값을 저장합니다.
3. 0 ∼ 7 번째에 해당하는 Rank Table (첫 4 bytes)를 Little Endian 형식에서 Big Endian 형식으로 바꾸어 줍니다.
4. 8 ∼ 15 번째에 해당하는 Rank Table을 `freq` 배열 (해당 글자가 있는지 없는지 저장)을 사용하여 구합니다.
5. Padding을 구하기 위해 다음 4 bytes 중 첫 4 bits를 추출합니다.
6. 남은 encoding data를 decode하기 위해서 남은 body에 있는 데이터를 2 bytes (4 nibbles) 씩 `remainder`에 추가시킵니다.
7. `remainder`에 있는 가장 첫 번째 bit가 0 인지 보고 0 이라면 000 ∼ 011 까지 매핑을 하고 아니라면 1000 ∼ 1011 까지 매핑을 한다. 2 번째 MSB까지 1이라면 11000 ∼ 11111 까지 매핑한다.
8. Step 7에서 매핑한 value를 rank table에서 찾아 원래 값으로 변환시킨다. 이후 Step 6 ∼ 8을 반복한다.
9. 각각의 과정에서 decoding 된 글자가 있으면 레지스터에 저장하고 레지스터가 다 찼을 경우 `store` 연산을 한다. 이 때 조건에 따라 -1 을 반환한다.

Codes written in C language

```
union Byte {
  struct {
    unsigned int second: 4;
    unsigned int first: 4;
  } pair;
  unsigned int value: 8;
};

union Boolean {
    unsigned int value: 1;
};


union Byte findHuffman(union Byte r, int *len, char *outp, int *outb, int *rank,
                                                int *outbytes) {
    if (*outb < 0) return r;

    int l = *len-1, x = 0, y = 0;
    if (l < 0) return r;
    if (r.value & (1<<l)) {
        l--;
        if (l < 0) return r;
        if (r.value & (1<<l)) {
            l -= 3;
            if (l < 0) return r;
            x = 8;
            y = ((r.value & (7<<l))>>l);
            *len -= 5;
        }
        else {
            l -= 2;
            if (l < 0) return r;
            x = 4;
            y = ((r.value & (3<<l))>>l);
            *len -= 4;
        }
    }
    else {
        l -= 2;
        if (l < 0) return r;
        x = 0;
        y = ((r.value & (3<<l))>>l);
        *len -= 3;
    }

    int ind = (*outb)>>1;
    if(ind+1 > *outbytes) {
        *outb = -1;
        return r;
    }
    if(((*outb)++) % 2)
        outp[ind] <<= 4;
    outp[ind] += rank[x+y];
    return findHuffman(r, len, outp, outb, rank, outbytes);
}

int min(int x, int y) {
    if (x>y) return y;
```

```
    return x;
}


int decode(const unsigned char *inp, int inbytes, char *outp, int outbytes) {
    if (inbytes == 0) return 0;

    int rank[16];
    union Boolean freq[16] = {0};
    int k = 0;
    for (; k < 8; k += 2) {
        union Byte b;
        b.value = inp[k >> 1];
        rank[k] = b.pair.first;
        rank[k+1] = b.pair.second;
        freq[b.pair.first].value = 1;
        freq[b.pair.second].value = 1;
    }
    for (int i = 0; i < 16; i++) {
        if (freq[i].value == 0) {
            rank[k++] = i;
        }
    }

    union Byte b;
    b.value = inp[4];
    unsigned int padding = b.pair.first;

    union Byte remainder;
    remainder.value = b.pair.second;
    int outb = 0, len = 4;
    remainder = findHuffman(remainder, &len, outp, &outb, rank, &outbytes);

    for (int i = 5; i < inbytes; i++) {
        union Byte b;
        b.value = inp[i];

        if (i+1 != inbytes || padding <= 4) {
            len += 4;
            remainder.value <<= 4;
            remainder.value += b.pair.first;
            remainder = findHuffman(remainder, &len, outp, &outb, rank, &outbytes);
        }
        else {
            len += (8-padding);
            remainder.value <<= (8-padding);
            remainder.value += (b.pair.first >> (padding-4));
            remainder = findHuffman(remainder, &len, outp, &outb, rank, &outbytes);
        }

        if (i+1 != inbytes) {
            len += 4;
            remainder.value <<= 4;
            remainder.value += b.pair.second;
            remainder = findHuffman(remainder, &len, outp, &outb, rank, &outbytes);
        }
        else if (padding < 4) {
            len += (4-padding);
            remainder.value <<= (4-padding);
            remainder.value += (b.pair.second >> padding);
```

```
            remainder = findHuffman(remainder, &len, outp, &outb, rank, &outbytes);
        }
    }
    return outb;
}
```

3 Conclusion

`int decode(const char *inp, int inbytes, char *outp, int outbytes);` 함수를 먼저 C 코드를 구현함으로써 함수의 알고리즘을 대략적으로 구상할 수 있었습니다. 그리고 이렇게 생성된 `decode.c` 파일을 pyrisc의 다음과 같은 코드로 disassemble을 하려했습니다.

- `riscv32-unknown-elf-gcc -Ofast -c decode.c`
- `riscv32-unknown-elf-objdump -d decode.o`

생성된 파일은 `zero (x0), sp, ra, a0  a5` 외에는 다른 레지스터를 쓰지 말라는 문제 조건에 맞지 않는 RISC-V Assembly Code를 생성하였습니다. 최대한 레지스터 사용을 최소화하기 위해 Stack memory의 효율적인 사용을 하려 했으나 disassembled code 라인 수가 무려 500 줄 가까이 되어 이는 무리라고 판단되어 빠른 포기를 하였습니다.
Assembly code로 위의 `decode` 함수를 새로 구현하기 시작하였습니다. 강의시간에 언급되었던 Operation 시간이 오래걸리는 Branch operation과 Data transfer operation 보다는 ALU operation를 활용하기 위해 Stack memory 사용을 최소화 했으며, 사용이 불가피한 경우 loop 바깥에서 사용하였습니다.
모든 테스트케이스에 대해 통과를 하여 Performance metric을 높이기 위해 코드를 수정하여 재채점을 해본 결과 metric을 산정할때 단순히 cycle 수로만 판단하는 사실을 알게 되었습니다. 결국 DTO 개수가 ALU OP 보다 더 많더라도 총 수행되는 instruction 개수가 같다면 같은 점수가 되는 것이었으며 앞서 했던 최적화가 소용없게 되었고 Metric 최적화에 시간과 열정이 충분하다면 아래와 같은 개선점에 대해 생각해 보았습니다.

Improvements

- 컴파일러를 통해 Disassemble된 코드를 레지스터 개수를 줄이면서 옮겨쓴다.
- Loop에서 load/store operation을 금기시 하지 않고 instruction 개수만 줄일 수 있도록 한다.
- Branch operation 보다는 inline statement을 통해 PC instruction 개수를 줄인다.
- 들어오는 input batch size를 7 nibbles 까지 키워 작업을 수행한다.