Wordle On the Switch_v1

Start-up:
1) Compile the wordle.p4 with

p4c --target bmv2 --arch v1model --std p4-16 wordle.p4

2) Load wordle.json file onto the switch

sudo simple_switch -i 0@eth0 wordle.json

3) Run wordle_sender.py

sudo python3 wordle_sender.py

4) Run wordle_receiver.py

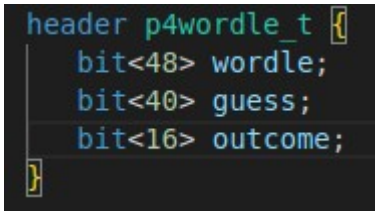
sudo python3 wordle_receiver.py

Documentation:

wordle.p4

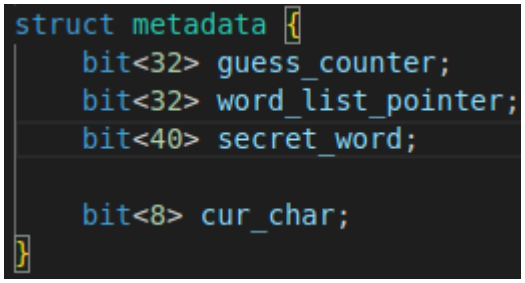
Right Now, this is the world's worse game of wordle.

This is our Packet Format.



p4wordle_t.wordle : This is just the word "WORDLE". It is used a check in the wordle_receiver.py to make sure that we are looking at the correct script.
p4wordle_t.guess: This represents our guess. wordle_sender.py rescripts the guess to a 5 character Uppercase word.
p4wordle_t.outcome: This is represents the whether our guess was correct. wordle_receiver.py will process this value. Note: We are only using 10 bits to check our word. However, we have to send our packets in groups of 8 bits, hence we are sending 16 bits for p4wordle_t.outcome here.

I have created some values in metadata.



These are some recurring values that I will be using in my functions.

To remember the word list counter values between consecutive packets, we use a bunch of registers.

```
/*Here are the registers*/
register <bit<40>>(8) word_list;
register <bit<32>>(1) guess_counter_state;
register <bit<32>>(1) word_list_pointer_state;
```

Here is the word list.

```
word_list.write(0,0x524f555445);/*ROUTE*/
word_list.write(1,0x5441424c45);/*TABLE*/
word_list.write(2,0x4D41474943);/*MAGIC*/
word_list.write(3,0x47414D4553);/*GAMES*/
word_list.write(4,0x4D4F555345); /*MOUSE*/
word_list.write(5,0x4241434F4E); /*BACON*/
word_list.write(6,0x43554C5453); /*CULTS*/
word_list.write(7,0x535441494E); /*STAIN*/
```

There are only 8 words at the moment, but it is relatively trivial to add more.

Whenever a valid packet is sent to the switch. We load up the correct values from the register into our metadata.

```
/* State operations */
guess_counter_state.read(meta.guess_counter, 0);
word_list_pointer_state.read(meta.word_list_pointer, 0);
word_list.read(meta.secret_word, meta.word_list_pointer);
```

The word comparison operation is done via two nested functions. This is just to make the code look neater.

```
action operation_check_char(bit<3> i){
    if (i==1){
        meta.cur_char = hdr.p4wordle.guess[7:0];
        compare_char(meta.cur_char,1,0x4000,0xc000);

    }

    if (i==2){
        meta.cur_char = hdr.p4wordle.guess[15:8];
        compare_char(meta.cur_char,2,0x1000,0x3000);
    }

    if (i==3){
        meta.cur_char = hdr.p4wordle.guess[23:16];
        compare_char(meta.cur_char,3,0x0400,0x0c00);

    }

    if (i==4){
        meta.cur_char = hdr.p4wordle.guess[31:24];
        compare_char(meta.cur_char,4,0x0100,0x0300);

    }

    if (i==5){
        meta.cur_char = hdr.p4wordle.guess[39:32];
        compare_char(meta.cur_char,5,0x0040,0x00c0);
    }
}
```

This function select the character for comparison and and defines the output binary values to be applied to p4calc_t.output.

```
action compare_char(bit<8> cur_char, bit<3> i, bit<16> exist, bit<16> right_place){
    if (cur_char == meta.secret_word[7:0]){
        if (i==1){
            state_machine(right_place);
        }
        else{
            state_machine(exist);
        }
    }
    if (cur_char == meta.secret_word[15:8]){
        if (i==2){
            state_machine(right_place);
        }
        else{
            state_machine(exist);
        }
    }
    if (cur_char == meta.secret_word[23:16]){
        if (i==3){
            state_machine(right_place);
        }
        else{
            state_machine(exist);
        }
    }
    if (cur_char == meta.secret_word[31:24]){
        if (i==4){
            state_machine(right_place);
        }
        else{
            state_machine(exist);
        }
    }
    if (cur_char == meta.secret_word[39:32]){
        if (i==5){
            state_machine(right_place);
        }
        else{
            state_machine(exist);
        }
    }
}
```

The second function decides whether the character exist and if it is in the right place.
We have two flags:
exists = 0
right_place = 0

If a char is found anywhere in the word, exists = 1
(Note: This is not strictly speaking how wordle works, but the limited computational ability of the switch makes storing information of each char count difficult)

If a char is in the correct place, right place = 1

We then load these flag into p4wordle_t.outcome
right_exists right_exists right_exists right_exists right_exists Unused

place 1st Char of Guess	place 2nd Char of Guess	place 3rd Char of Guess	place 4th Char of Guess	place 5th Char of Guess	Unused

So, if you think about the possible states of the system.

If the char does not exist, we get 00
If the char exists but not at the right place, we get 01
If the char exists and is at the right place, we get 11

We have hard coded for this comparison to done on each char.

```
operation_check_char(1);
operation_check_char(2);
operation_check_char(3);
operation_check_char(4);
operation_check_char(5);
```

Finally we increment our guess_counter and pointer.

```
if (hdr.p4wordle.outcome == 0xffc0){
    meta.guess_counter = 0;
    meta.word_list_pointer = meta.word_list_pointer+1;
    if (meta.word_list_pointer == 8){
        meta.word_list_pointer = 0;
    }
}
else{
    meta.guess_counter = meta.guess_counter + 1;
    if (meta.guess_counter == 6){
        meta.guess_counter = 0;
        meta.word_list_pointer = meta.word_list_pointer + 1;
        if (meta.word_list_pointer == 8){
            meta.word_list_pointer = 0;
        }
    }
}

guess_counter_state.write(0,meta.guess_counter);
word_list_pointer_state.write(0, meta.word_list_pointer);
```

wordle_sender.py

The structure of this piece of code is very similar to calc_sender.py.

```
class P4wordle(Packet):
    name = "P4wordle"
    fields_desc = [ StrFixedLenField("wordle", "WORDLE", length=6
                    StrFixedLenField("guess", "GUESS", length=5),
                    IntField("outcome", 0x0000)]
```

This is the format of the packet we expect to receive.

```
def word_parser(s, i, ts):
    pattern = "\s*^[A-Z]{5}\s*"
    match = re.match(pattern,s[i:])
    if match:
        ts.append(match.group())
        return i + match.end(), ts
    raise WordParseError("Expected An Upper Case 5 Letter Word")
```

This is our custom parser that seeks to process the packet it receives.

wordle_receiver.py

The structure of this piece of code is very similar to calc_receiver.py

```
def outcome(ans, x):
    os.system('color')

    #For some reason the int of the outcome is bitshifted to the left by 16 more spaces than expected
    #Note: Recieved bit stream is in the opposite direction of how the bits are defined in p4
    for i in range(5):
        exist = False
        right_place = False
        if ((x&(1<<(23+2*i-1)))!=0): exist = True
        if ((x&(1<<(23+2*i)))!=0):right_place = True
        if (right_place):
            print(colored(chr(ans[i]), 'green'),end="")
        elif(exist):
            print(colored(chr(ans[i]), 'yellow'),end="")
        else:
            print(colored(chr(ans[i]), 'white'),end="")
    print()
```

Essentially we want to print our outcome based on the classic color scheme of Wordle.

Note: Due to some weird parsing of the received packet, we see that we get a 32 bit outcome int.

We will be only dealing with the first 16 bits.

Furthermore, due to some incompatibility of how the outcome string is defined. The received outcome bitstream looks like this.

right_place	exist	right_place	exist	right_place	exist	right_place	exist	right_place	exist
5th Char of Guess		4th Char of Guess		3rd Char of Guess		2nd Char of Guess		1st Char of Guess	