To begin

At first, we take our CPP files and create assembly versions of them using https://godbolt.org/ and use the RISC-V 32bit gcc (Specifically RISC-V (32-bits) gcc 13.2.0). I attempted to find libraries that could do this for me but I had very messy and not readable results. Therefore, I decided to use assembly converter to obtain the assembly versions of each of the advp.cpp and myp.cpp. I would copy the code taken in the two cpp files and copy the output from the website converter and put them into txt files called myp.txt and advp.txt

In [ ]:
```
from tabulate import tabulate
###### need this to help table our results in the python notebook #######
```

In [ ]:
```
# get each line for the text file and turn it into a string array
with open('myp.txt', 'r') as file :
    myp_array = [line.strip() for line in file]

# get each line for the text file and turn it into a string array
with open('advp.txt', 'r') as file :
    advp_array = [line.strip() for line in file]
```

Idea

I used the idea of three address code to help me approach the problem in generating basic blocks in our assmebly language. Since three address code mainly deals with assembly langauge or some form of pesudocode of assembly language, it was very easy to use their algorithms to help find basic blocks especially due to control flow insturctions being specified by starting with characters J and B. The code below helps develop the basic blocks in any basic assembly. Since the txt file also has main: and .LX in the file, we can also use these markers to help see where basic blocks would form since they end with a colon or ":"

In [ ]:
```
def parse_basic_blocks(assembly_code):
    basic_blocks = []
    current_block = []
    line_numbers = []

    for i, instruction in enumerate(assembly_code, start=1):
        if instruction.endswith(':'): # only main, .LX have a : at the end of
            if current_block:
                basic_blocks.append((current_block, line_numbers))
                current_block = []
                line_numbers = []
            current_block.append(instruction)
            line_numbers.append(i)
        else:
            current_block.append(instruction)
            line_numbers.append(i)
            if instruction.startswith('j') or instruction.startswith('b'):
                # control flow instruction are only jumps and branches and they
                basic_blocks.append((current_block, line_numbers))
                current_block = []
                line_numbers = []

    if current_block:
        basic_blocks.append((current_block, line_numbers))

    return basic_blocks
```

Idea

The main idea behind this function is just to help generate the values for the table for each block and generate the lines that are used within each block and the entry and exit points of each basic block

In [ ]:
```
def generate_basic_block_info(basic_blocks):
    block_info = []
    for i, (block, line_numbers) in enumerate(basic_blocks, start=1):
        entry_point = line_numbers[0]
        exit_point = line_numbers[-1]
        lines = ', '.join(map(str, line_numbers))
        block_info.append({
            'block': str(i),
            'lines': lines,
            'entry point': str(entry_point),
            'exit point': str(exit_point)
        })
    return block_info
```

In [ ]:
```
bb1 = parse_basic_blocks(myp_array)
block_table = generate_basic_block_info(bb1)
print(tabulate(block_table, headers="keys", tablefmt="fancy_grid"))
```

```
╒═════════╤═════════════════════════════════╤═══════════════╤══════════════╕
│   block │ lines                           │   entry point │   exit point │
╞═════════╪═════════════════════════════════╪═══════════════╪══════════════╡
│       1 │ 1, 2, 3, 4, 5, 6, 7, 8, 9       │             1 │            9 │
├─────────┼─────────────────────────────────┼───────────────┼──────────────┤
│       2 │ 10, 11, 12                      │            10 │           12 │
├─────────┼─────────────────────────────────┼───────────────┼──────────────┤
│       3 │ 13, 14, 15                      │            13 │           15 │
├─────────┼─────────────────────────────────┼───────────────┼──────────────┤
│       4 │ 16, 17, 18, 19, 20, 21          │            16 │           21 │
├─────────┼─────────────────────────────────┼───────────────┼──────────────┤
│       5 │ 22, 23, 24, 25, 26, 27, 28, 29  │            22 │           29 │
├─────────┼─────────────────────────────────┼───────────────┼──────────────┤
│       6 │ 30, 31, 32, 33                  │            30 │           33 │
├─────────┼─────────────────────────────────┼───────────────┼──────────────┤
│       7 │ 34, 35, 36, 37, 38              │            34 │           38 │
╘═════════╧═════════════════════════════════╧═══════════════╧══════════════╛
```

In [ ]:
```
bb2 = parse_basic_blocks(advp_array)
block_table1 = generate_basic_block_info(bb2)
print(tabulate(block_table1, headers="keys", tablefmt="fancy_grid"))
```

block	lines	entry point	exit point
1	1, 2, 3, 4, 5, 6, 7, 8, 9	1	9
2	10, 11, 12	10	12
3	13, 14, 15	13	15
4	16, 17, 18, 19, 20, 21	16	21
5	22, 23, 24, 25, 26, 27, 28, 29	22	29
6	30, 31, 32, 33	30	33
7	34, 35, 36, 37, 38, 39	34	39
8	40, 41, 42, 43	40	43
9	44, 45, 46, 47	44	47
10	48, 49, 50	48	50
11	51, 52, 53, 54	51	54
12	55, 56, 57, 58	55	58
13	59, 60, 61	59	61
14	62, 63, 64	62	64
15	65	65	65
16	66, 67, 68, 69, 70	66	70
17	71, 72, 73, 74, 75	71	75
18	76, 77, 78, 79, 80	76	80
19	81, 82, 83	81	83
20	84, 85, 86, 87	84	87
21	88, 89, 90	88	90
22	91, 92	91	92
23	93, 94, 95, 96	93	96
24	97, 98, 99, 100, 101, 102	97	102