1. 숙제풀이

- 03 번 - int res, k=100; (변수 초기화 주의!)
- mv 명령어 설명
  (ex) mv class.c third.c

- shortcut ||에서 연산 횟수를 감소시키기 위해 조건문을 최적화하는 방법
  (ex) i%6==1 || i%4 == 1 => i%4 == 1 || i%6 == 1

- 문제 10 번 소스 gdb debugging

- gdb명령어

  ㄱ. si => 함수내부 진행

  ㄴ. ni => 프로그램 프로시저 진행

2. 버깅의 필요성

```
#include <stdio.h>

void print_rom(void)
{
        int i=2, j=1;

        while(i < 10)
        {

                while(j < 10)
                {

                        printf("%d x %d = %d₩n", i, j, i*j);

                        j++;
                }
                j=1;

                i++;
        }
}

int main(void)
{

        print_rom();
        return 0;
}
```

- 위의 코드는 컴파일은 되지만 논리적으로 오류를 범하게 됨
- number += number 코드는 2 의 승수로 값이 증가
- gdb 명령어(si, ni 사용해 디버깅 수행)

2. scope의 개념

{ => stack frame 생성

} => stack frame 해제

(ex)

```
int main(void)
{


   int global_area = 1;

         {
               int local_area1 = 2;
               printf("global_area = %d₩n", global_area);
               printf("global_area1 = %d₩n", local_area);

         } => 스택프레임 해제

         {

               int local_area2 = 3;
               printf("global_area = %d₩n", global_area);
               printf("global_area1 = %d₩n", local_area2);
         } => 스택프레임 해제

         printf("global_area = %d₩n", global_area);
         printf("local_area1 = %d₩n", local_area1); => 스택프레임 해제로 local_area1 는 존재X
         printf("local_area1 = %d₩n", local_area2); => 스택프레임 해제 local_area1 는 존재X

         return 0;
}
```

- 소스파일의 영역별 저장

   ㄱ. 전역변수, static - data영역저장
   ㄴ. 지역변수 - stack영역저장
   ㄷ. 코드 - text영역저장
   ㄹ. 동적할당 - 힙영역저장

- static 변수

   ㄱ. 지역변수를 static으로 선언시 이 변수를 선언한 함수 내에서만 접근 가능 data영역에 저장

(ex.1)

```
#include <stdio.h>

void count_static_value(void)
{
         static int count = 1;                    //함수 진입시 최초에 1 번만 count=1 등록
         printf("count = %d₩n", count);
         count++;
```

```
}

int main(void)
{
	int i;

	for(i=0; i < 7; i++)
		count_static_value();
}
```

ㄴ. 함수에 static이 붙으면 private 함수 구현가능(다른 외부에서 함수호출 X)

3. Continue문의 필요성

ㄱ. NaN = 숫자/0

ㄴ. Inf = 무한

(ex)

```
int main(void)
{

 int number = 0;

 while(1)
 {
	number++;

	 if(number == 5)
		continue;

	 printf("%d₩n", number);

	 if(number == 10)
		break;

 }

return 0;

}
```

4. do while의 필요성

ㄱ. 1 번은 실행됨

ㄴ. do while의 궁극적 사용이유는 매크로 확장

(ex)

```
int main(void)
{
 int number = 0;

 do
 {
          number++;

           if(number == 5)
                    continue;

           printf("%d₩n", number);


 }while(number < 10)

return 0;

}
```

5. #define의 필요성

ㄱ. 코드에 사용되는 상수를 대체하는데 필요, 코드 revision시 #define만 고치면 되므로 편리함

(ex) #define TEST 500
printf("%d₩n", TEST);

ㄴ. kernel에서 매크로 주로 사용되며 매크로 함수에는 ';'를 하지 않음

6. for문의 필요성

-루프의 간결성을 위해 필요

for(초기화; 조건문; 증감식)
{

}

(ex)

```
for(i=0; i < 10; i++, result++)
{

}
```

(ex)

```
int main(void)
{

	for(i=0, result = 'A'; i < 10; i++, result++)
	{

		printf("%c₩n", result);
	}

	return 0;

}
```

(ex) 무한루프 구현

```
int main(void)
{
	int i, result = 'A';


	for(; ;)
	{
		printf("%c₩n", result);
	}

	return 0;
}
```

*windows 사용자일 경우 주의할 점

리눅스는 for(int i=0; ~~~) 안먹힘

7. goto의 필요성

- system programing에서 자주 사용

- kernel에서 사용하며 buffer를 사용하는 곳에서 주로 사용

(ex) flag를 활용한 루프탈출

```
int main(void)
{
	int i, j, k, flag=0;

	for(i=0; i < 5; i++)
	{

		for(j=0; j < 5; j++)
		{

			for(k=0; k < 5; k++)
			{

				if((i == 2) && (j == 2) && (k == 2))
				{
					printf("Error!!!₩n");
					flag=1;
				}
				else
				{

					printf("Data₩n");
				}

				if(flag)
				{

					break;
				}
			}

			if(flag)
			{
				break;
			}
		}

		if(flag)
		{
			break;
		}
	}


	return 0;
}
```

(ex) goto문

```
#include <stdio.h>

int main(void)
{
	int i, j, k;

	for(i=0; i < 5; i++)
	{

		for(j=0; j < 5; j++)
		{

			for(k=0; k < 5; k++)
			{

				if((i == 2) && (j == 2) && (k == 2))
				{
					printf("Error!!!₩n");
					goto err_handler;
				}
				else
				{

					printf("Data₩n");
				}


			}

		}

	}

	return 0;

err_handler:
 printf("Goto Zzang!₩n");
 return -1;

}
```

- goto의 이점과 cpu 파이프라인

ㄱ. 파이프라인 3 단계

1. Fetch - 실행해야할 명령어를 물어옴
2. Decode - 어떤 명령어인지 해석함
3. Execute - 실제 명령어를 실행시킴

Fetch	Decode	Execute				
ADD(F)	ADD(D)	ADD(E)				
	MOV(F)	MOV(D)	MOV(E)			
		CALL(F)	CALL(D)	CALL(E)		
			MOV(F)	~~MOV(D)~~	~~PUSH(F)~~	
				~~MOV(F)~~	~~MOV(F)~~	

=> call이 실행되면 파이프라인이 깨짐

*sw와 HW 동작을 생각할 때 주의할 점

- sw는 멀티 코어 상황이 아니면 어떤 상황에서도 한 번에 한 가지 동작만 실행
(cpu 한 개는 오직 하나의 프로세스만 한 시점에 실행가능)

- HW 회로는 병렬회로가 존재하듯 모든 회로가 동시 동작 가능
- 파이프라인은 cpu에 구성된 회로로써 모든 모듈의 동시 동작을 가능하게 함
(FPGA는 병렬 동작 가능하며 cpu설계시 사용)

8. 재귀호출

- 재귀호출은 편하지만 재귀호출에 따라 jmp가 발생해 파이프라인을 깨트림

```
#include <stdio.h>

int fib(int num)
{
	int result;

	if(num == 1 || num == 2)
	{
		return 1;
	}
	else
	{
		result = fib(num - 1)+fib(num - 2);

		return result;
	}

int main(void)
{
```

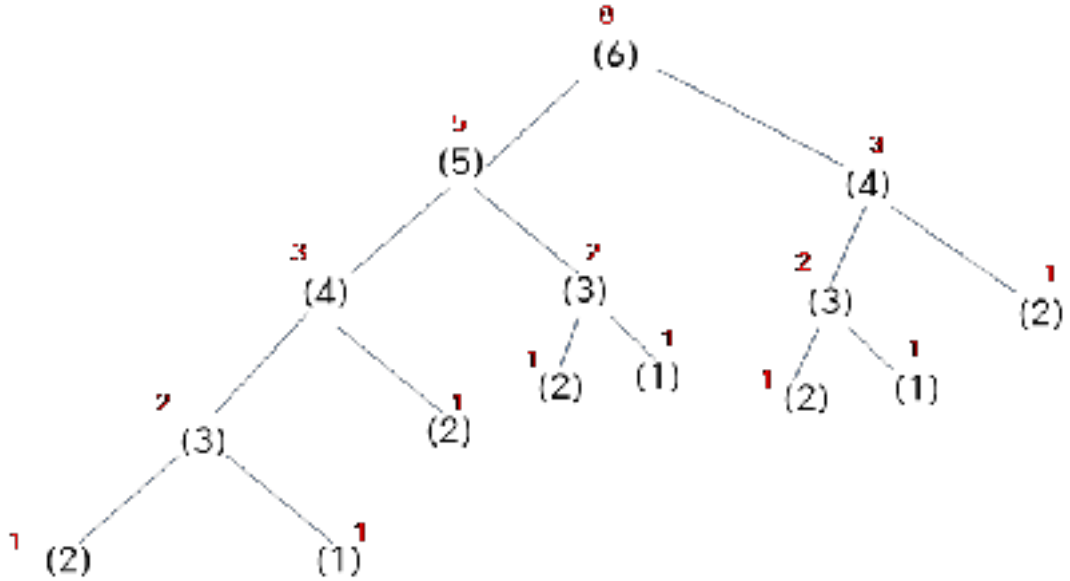
```
   int result, final_val;
   printf("피보나치 수열의 항의 개수를 입력하시오: ");
   scanf("%d", &final_val);
   result = fib(final_val);
   printf("%d번째 항의 수는 = %d₩n", final_val, result);
   return 0;

}
```

- gdb명령어 s,bt, finish 이용해 recursive.c 디버깅

```
(gdb) bt
#0   fib (num=2) at recursive.c:18
#1   0x0000000000400622 in fib (num=3) at recursive.c:14
#2   0x0000000000400622 in fib (num=4) at recursive.c:14
#3   0x0000000000400631 in fib (num=6) at recursive.c:14
#4   0x0000000000400694 in main () at recursive.c:29
(gdb) s
fib (num=1) at recursive.c:8
8                   if(num == 1 || num == 2)
(gdb) s
10                          return 1;
(gdb) s
18      }
(gdb) bt
#0   fib (num=1) at recursive.c:18
#1   0x0000000000400631 in fib (num=3) at recursive.c:14
#2   0x0000000000400622 in fib (num=4) at recursive.c:14
#3   0x0000000000400631 in fib (num=6) at recursive.c:14
#4   0x0000000000400694 in main () at recursive.c:29
(gdb) s
16                          return result;
(gdb) s
18      }
(gdb) bt
#0   fib (num=3) at recursive.c:18
#1   0x0000000000400622 in fib (num=4) at recursive.c:14
#2   0x0000000000400631 in fib (num=6) at recursive.c:14
#3   0x0000000000400694 in main () at recursive.c:29


                                                    ...
```



- rax레지스터에는 함수의 리턴값이 저장됨