2018.02.26 - 4일차 수업

---

* 디버깅을 하는 이유

디버깅은 컴파일은 성공했으나(즉 문법 오류 없음)

논리적인 오류가 존재하는 경우에 수행하는 것이다.

```
#include <stdio.h>
int main(void)
{
	int number = 1;
	while(1)
	{
		printf("number = %d₩n", number);
		number += number;
		if(number == 100)
			break;
	}
	return 0;
}
```

(위와 같은 케이스에서는 number가 2^n으로 움직여서 무한루프에 빠지게 되는 논리적 오류)

그 외에도 예측치 못한 다양한 문제가 존재할 수 있다.

이런 이상한 동작을 하는 경우 문제점을 파악하기 위해 실행하는 것이 디버깅이다.

---

* 디버깅 방법

gcc -g [파일명.c] 입력

gdb a.out 입력

gdb 쉘이 뜨면 아래와 같이 입력한다.

b main

그리고 r 을 입력하면 프로그램이 실행이 된다.

기계어의 경우에는 si 를 입력해서 분석했었다.

C 레벨에서도 1 줄씩 진행할 수도 있는데

그럴 경우에는 s 나 n 을 입력하면 된다.

s 는 함수가 있다면 함수 내부로 진입하고 없다면 1 줄 진행, n 은 함수가 있던 없던 그냥 1 줄 진행한다.

명령어 l 은 list 의 약자로 전체 C 코드를 볼 수 있다.

명령어 c 는 다음 브레이크 포인트를 만날때까지 계속 작업한다. (continue 의 약자)

---

* continue문

현재 실행중인 반복문 안에서 continue; 이하 부분을 실행하지 않고 되돌아가서 남은 반복문을 계속 진행한다

- continue 예제

```
#include <stdio.h>
int main(void)
{
	int number = 0;
	while(1)
	{
		number++;
		if(number == 5)
			continue;
		printf("%d₩n", number);
		if(number == 10)
			break;
	}
	return 0;
}
```

---

* #define 을 사용하는 이유

예를 들어서 회사 코드에 100 번 루프(반복)

를 돌아야 하는 코드가 777 개 있다.

이때 회사 규모가 커져서 100 번 루프가 아니라

500 번 루프를 돌아야 하는 상황이 되었다고 가정!

그러면 while(i < 100) 했던 부분을

전부다 찾아서 while(i < 500) 으로 변경해야 한다.

또한 while 만 있는것이 아니라

다른 코드들도 100 이라는 숫자에 관계된 코드들이

존재할 수 있다는 것이 문제다.

그러면 숫자 하나 바꾸는것이 모든 프로그램을

뜯어고치는 대 공사가 될 수 있는데

#define TEST 100 으로 선언하고

애초에 while(i < TEST) 로 만들어놨다면

#define TEST 500 으로 1 번 변경해서

모든 변경을 대 공사 없이 수행할 수 있다.

---

* for문

for문은 대표적인 반복문 중 하나이며

for([변수 초기값] ; [반복 조건] ; [변수 변화])

위의 형식으로 사용된다.

만약 [반복 조건]을 비어두면 무한루프가 생성된다.

- Windows 사용자일 경우 주의할점

리눅스는 for(int i = 0; ~ ; ~) 이렇게 for문 안에 자료형 선언이 불가능 하고 미리 변수를 선언해야한다

---

* CPU 파이프라인 관련 goto문의 이점

goto문의 경우 남용 할 경우 이상한 코드를 만들기도 하지만 여러겹의 함수나 반복문 등을 한번에 탈출해야 하는 등의 특별한 경우 사용하면 코드가 간결해지며 성능면에서도 유리하다.

if 문은 기본적으로 mov, cmp, jmp 로 구성된다.

goto 는 jmp 하나로 끝이다.

for 문이 여러개 생기면 if, break 조합의 경우

for 문의 갯수만큼 mov, cmp, jmp 를 해야 한다.

문제는 바로 jmp 명령어다.

call 이나 jmp 를 CPU Instruction(명령어) 레벨에서

분기 명령어라고 하고 이들은

CPU 파이프라인에 매우 치명적인 손실을 가져다준다.

파이프라인은 짧은 것부터 긴 것까지

5 단계 ~ 수십 단계로 구성된다.

(ARM, Intel 등등 다양한 프로세서들 모두 마찬가지)

기본적으로 아주 단순한

CPU 의 파이프라인을 설명하자면

아래와 같은 3 단계로 구성된다.

1. Fetch - 실행해야할 명령어를 가져옴

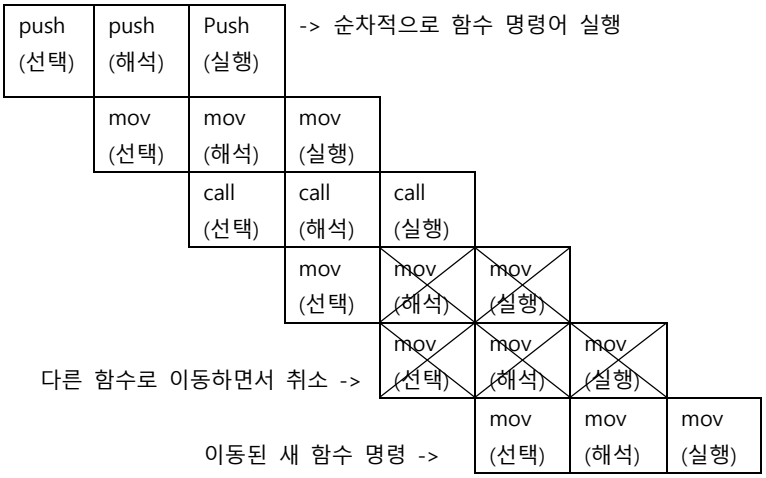
2. Decode - 어떤 명령어인지 해석함

3. Execute - 실제 명령어를 실행시킴

그런데 왜 jmp 나 call 등의

분기 명령어가 문제가 될까 ?

->기본적으로 분기 명령어는 이전에 실행 준비중이던 파이프라인들을 부수고 다른 함수로 넘어간다.



가장 단순한 CPU는 실행까지 3 clock 을 소요하는데

분기 명령어를 실행하는 순간 실행 준비중이던 파이프라인들이 깨져서 3 clock 을 버리게 된다

만약 파이프라인의 단계가 수십 단계라면

분기가 여러번 발생할 경우 파이프라인 단계 x 분기 횟수만큼 CPU clock 을 낭비하게 된다.

따라서 성능면에서도 goto 가 월등히 압도적이다.

(jmp 1 번만 실행된다)

---

* SW 와 HW 동작을 생각할 때 주의할 점

SW 는 멀티 코어 상황이 아니면 어떤 상황이든

한 번에 한 가지 동작만 실행할 수 있다.

한마디로 CPU 한 개는 오로지 한 순간에 한 가지 동작만 할 수 있다

반면 HW 회로는 병렬 회로가 존재하듯이

모든 회로가 동시에 동작할 수 있다.

파이프라인은 CPU 에 구성된 회로이기 때문에

모든 모듈들이 동시에 동작할 수 있는 것이다.

(FPGA 프로그래밍은 병렬 동작이고 CPU 설계는 FPGA 를 이용하여 한다)

---

1 ~ 1000 사이에서 3의 배수의 합을 구하시오

```c
#include <stdio.h>
int main()
{
	int result = 0;
	for (int i = 1; i <= 1000; i++)
	{
		if (i % 3 == 0)
			result += i;
	}
	printf("1~1000 사이 3의 배수의 합: %d₩n", result);
	return 0;
}
```

---

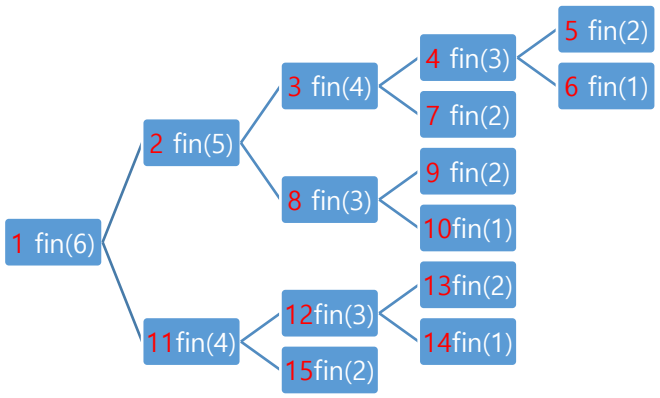
1 ~ 1000 사이에서 4나 6으로 나눠도 나머지가 1인 수의 합을 출력하라.

```c
#include <stdio.h>
int main()
{
	int result = 0;
	for (int i = 1; i <= 1000; i++)
	{
		if (i % 4 == 1 && i % 6 == 1)
			result += i;
	}
	printf("1~1000 사이 4나 6으로 나눠도 나머지가 1인 수의 합: %d₩n", result);
	return 0;
}
```

---

fib 함수 동작 분석(디버깅 및 그림 그리기)

디버깅 분석 결과 그림(빨간색 번호는 함수 처리 순서)



디버깅 내용

```
(gdb) b main
Breakpoint 1 at 0x40064f: file test001.c, line 13.
(gdb) r
Starting program: /home/ds/debug
Breakpoint 1, main () at test001.c:13
13      {
(gdb) n
15          printf("피보나치 항 = ");
(gdb) n
16          scanf("%d", &n);
(gdb) n
피보나치 항 = 6
17          printf("%d₩n", fib(n));
(gdb) s
fib (n=6) at test001.c:4
4           if (n < 1)
(gdb) s
6           else if (n == 1 || n == 2)
(gdb) s
9               return fib(n - 1) + fib(n - 2);
(gdb) s
fib (n=5) at test001.c:4
4           if (n < 1)
(gdb) s
6           else if (n == 1 || n == 2)
(gdb) s
9               return fib(n - 1) + fib(n - 2);
(gdb) s
fib (n=4) at test001.c:4
4           if (n < 1)
(gdb) s
6           else if (n == 1 || n == 2)
(gdb) s
9               return fib(n - 1) + fib(n - 2);
(gdb) s
fib (n=3) at test001.c:4
4           if (n < 1)
(gdb) s
6           else if (n == 1 || n == 2)
(gdb) s
9               return fib(n - 1) + fib(n - 2);
(gdb) s
fib (n=2) at test001.c:4
4           if (n < 1)
(gdb) s
6           else if (n == 1 || n == 2)
(gdb) s
7               return 1;
(gdb) s
10      }
(gdb) s
fib (n=1) at test001.c:4
4           if (n < 1)
(gdb) s
6           else if (n == 1 || n == 2)
(gdb) s
7               return 1;
(gdb) s
10      }
(gdb) s
10      }
(gdb) s
fib (n=2) at test001.c:4
4           if (n < 1)
(gdb) s
6           else if (n == 1 || n == 2)
(gdb) s
7               return 1;
(gdb) s
```

```
10      }
(gdb) s
10      }
(gdb) s
fib (n=3) at test001.c:4
4           if (n < 1)
(gdb) s
6           else if (n == 1 || n == 2)
(gdb) s
9               return fib(n - 1) + fib(n - 2);
(gdb) s
fib (n=2) at test001.c:4
4           if (n < 1)
(gdb) s
6           else if (n == 1 || n == 2)
(gdb) s
7               return 1;
(gdb) s
10      }
(gdb) s
fib (n=1) at test001.c:4
4           if (n < 1)
(gdb) s
6           else if (n == 1 || n == 2)
(gdb) s
7               return 1;
(gdb) s
10      }
(gdb) s
10      }
(gdb) s
10      }
(gdb) s
fib (n=4) at test001.c:4
4           if (n < 1)
(gdb) s
6           else if (n == 1 || n == 2)
(gdb) s
9               return fib(n - 1) + fib(n - 2);
(gdb) s
fib (n=3) at test001.c:4
4           if (n < 1)
(gdb) s
6           else if (n == 1 || n == 2)
(gdb) s
9               return fib(n - 1) + fib(n - 2);
(gdb) s
fib (n=2) at test001.c:4
4           if (n < 1)
(gdb) s
6           else if (n == 1 || n == 2)
(gdb) s
7               return 1;
(gdb) s
10      }
(gdb) s
fib (n=1) at test001.c:4
4           if (n < 1)
(gdb) s
6           else if (n == 1 || n == 2)
(gdb) s
7               return 1;
(gdb) s
10      }
(gdb) s
10      }
(gdb) s
fib (n=2) at test001.c:4
4           if (n < 1)
(gdb) s
6           else if (n == 1 || n == 2)
(gdb) s
7               return 1;
(gdb) s
10      }
(gdb) s
10      }
(gdb) s
10      }
```