TI DSP,MCU및 Xilinux Zynq FPGA 프로그래밍 전문가 과정

이름	문지희
학생 이메일	mjh8127@naver.com
날짜	2018/3/2
수업일수	7일차
담당강사	Innova Lee(이상훈)
강사 이메일	gcccompil3r@gmail.com

목차

1. 함수포인터
    - C 언어로 구사할 수 있는 최고급 기법
    - 함수포인터 응용 문제 풀이
    - 함수포인터를 사용하는 이유
2. C언어 기타 라이브러리들
    - memmov
    - memcpy
    - strlen
    - strcmp
3. 문제
4. 자료구조(스택) 구현하기

1. 함수포인터

예제1)

```
#include<stdio.h>

void aaa(void)
{
printf("aaa called₩n");
}

int number(void)
{
printf("number called₩n");
return 7;
}

void (*bbb(void))(void)
{
printf("bbb called₩n");
return aaa;
}

void ccc(void(*p)(void))
{
printf("ccc:I can call aaa!₩n");
p();
}



int (*ddd(void))(void)
{
printf("ddd : I can call nember₩n");
return number;
}

int main(void)
{
int res;
res= ddd()();
printf("%d₩n",res);
bbb()();
ccc(aaa);
ddd()();
return 0; }
```

프로토타입

> bbb
// void(*)(void) bbb(void)
리턴 : void(*)(void)
함수명 : bbb
인자 : void

> ccc
// void ccc(void(*p)(void))
리턴 : void
함수명 ccc:
인자 : void(*p)(void)

> ddd
// int (*)(void) ddd(void)
리턴 : int (*)(void)
함수명 : ddd
인자 : void

~결과
bbb called
aaa called
ccc:I can call aaa!
aaa called
ddd : I can call nember
number called

main함수에서 int형 변수 res를 선언하고 res에 함수 ddd의 리턴값을 넣는다. 함수 ddd에서는 I can call nember라는 문장을 출력하고 number라는 함수로 리턴하게 되는데 number는 int를 리턴하고 void를 인자로 받는 함수이기 때문에 오류없이 실행된다. number라는 함수에서는 number called라는 문장을 출력하고 7을 출력하게 되고 이 값이 res에 저장되게 되고 res의 값을 출력한다. 이후 함수 bbb가 실행되는데 bbb called라는 문장을 실행하고 함수aaa로 리턴하게 된다. 함수 aaa에서는 aaa called가 출력되고 main함수로 돌아와 함수 aaa를 인자로 받는 함수 ccc가 실행되는데 aaa에서는 aaa called라는 문장을 출력하고 ccc에서는 ccc:I can call aaa!을 출력한다. 이후 또다시 함수ddd가 실행되는데 ddd : I can call nember를 출력하고 number로 가 number called로 출력한다. 위의 함수에서는 res의 값을 보기 위해 main함수의 3줄을 추가했지만 결과 값은 3줄을 뺀 결과 값이다.

예제2)

```
#include<stdio.h>

void aaa(void)
{
printf("aaa called₩n");
}

void (*bbb(void(*p)(void)))(void)
{
p();
printf("bbb called₩n");
return aaa;
}

int main(void)
{
bbb(aaa)();
return 0;
}
```

~결과

```
aaa called
bbb called
aaa called
```

프로토타입

```
> bbb
리턴 : void(*p)(void)
이름 : bbb
인자 : void(*p)(void)
```

main함수에서 함수 bbb가 함수 aaa를 인자로 받게 된다. 함수 aaa는 함수 bbb의 인자형태와 맞으므로 오류없이 진행할 수 있다. 함수 bbb에서 함수 p를 실행하면 인자로 aaa를 받았기 때문에 aaa called가 출력되게 되고 함수 bbb의 bbb called가 출력되며 리턴값이 함수 aaa이므로 다시 aaa called가 출력되게 된다.

예제3)

```
#include<stdio.h>

int (*aaa(void))[2]
{
static int a[2][2]={10,};
printf("aaa called₩n");
return a;
}

int (*(* bbb(void))(void))[2]
{
printf("bbb called₩n");
return aaa;
}

int main(void)
{
int (*ret)[2];
int (*(*(*p[][2])(void))(void))[2]={{bbb,bbb},{bbb,bbb}};
//int(*)[2] (*)(void) (*)(void) p[][2]
int (*(*(*(*p1)[2])(void))(void))[2]=p;
// int (*)[2] (*)(void) (*)(void) (*p1)[2]
ret=((*(*(*(*p1)[2]))())());
printf("%d₩n",*ret[0]);
return 0;
}
```

~결과
bbb called
aaa called
10

프로토타입

> aaa
리턴 : int(*)[2]
이름 : aaa
인자 : void

> bbb
// int(*)[2](*)(void) bbb(void)
리턴 : int(*)[2](*)(void)
이름 : bbb
인자 : void

int형을 2개 저장할 수 있는 ret이라는 배열포인터가 선언되고, p라는 함수포인터에 함수 bbb가 초기화 되었다. p1에 함수포인터 p가 입력되었고 ret에 p1의 값이 입력되었다 *retdml 0번째 요소 값을 출력하면 bbb에서 bbb called라는 문장이 출력되고 aaa로 리턴하여 a라는 전적변수에 10의 값을 입력하고 aaa called를 출력한뒤 a값을 리턴하면 10이 출력되게 된다.

- 함수포인터 응용문제 풀이

```
#include <stdio.h>

int (* aaa(void))[2]
{
static int a[2][2] = {{10, 20}, {30, 40}};
printf("aaa called₩n");
return a;
}

int (*(* bbb(void))(void))[2]
{
printf("bbb called₩n");
return aaa;
}

int (*(* ccc(void))(void))[2]
{
printf("ccc called₩n");
return aaa;
}

int (*(* ddd(void))(void))[2]
{
printf("ddd called₩n");
return aaa;
}

int (*(* eee(void))(void))[2]
{
printf("eee called₩n");
return aaa;
}

int main(void)
{
int (*ret)[2];
int (*(*(*p[][2])(void))(void))[2] = {{bbb, ccc}, {ddd, eee}};
// int(*)[2] (*)(void) (*)(void) p[][2]
int (*(*(*(*p1)[2])(void))(void))[2] = p;
// int (*)[2](*)(void)(*)(void) (*p1)[2]
ret = ((*(*(*(*p1)[3]))())());
// ((*(*(*(*p1)[3]))())());
printf("*ret[0] = %d₩n", *ret[0]);
printf("ret[0][0] = %d₩n", ret[0][0]);
printf("*ret[1] = %d₩n", *ret[1]);
printf("ret[1][1] = %d₩n", ret[1][1]);
return 0;
}
```

~결과
eee called
aaa called
*ret[0] = 10
ret[0][0] = 10
*ret[1] = 30
ret[1][1] = 40

프로토타입
> aaa
리턴 : int(*)[2]
이름 : aaa
인자 : void

> bbb
리턴 : int(*)[2] (*)(void)
이름 : bbb
인자 : void
> ccc
// int (*)[2](*)(void) ccc(void)
리턴 : int(*)[2] (*)(void)
이름 : ccc
인자 : void

> ddd
// int(*)[2](*)(void) ddd(void)
리턴 : int(*)[2](*)(void)
이름 : ddd
인자 : void

> eee
// int (*(* eee(void))(void))[2]
리턴 : int(*)[2](*)(void)
이름 : eee
인자 : void

다중배열 p에 함수 bbb,ccc,ddd,eee 를 초기화시킴. 배열 p1에 배열 p를 입력하고 p1의 3번째 요소를 ret에 입력한다..? >모르겠어요....ㅠㅠㅠ

- 함수 포인터를 사용하는 이유

1. 비동기 처리
2. HW 개발 관점에서 인터럽트
3. 시스템 콜(유일한 SW 인터럽트임)

2번과 3번같은 인터럽트들(SW, HW)은 사실상 모두 비동기 동작에 해당한다. 그러므로 1번인 비동기 처리가 함수 포인터를 사용하는 핵심요소이다.
기본적으로 동기 처리라는 것은 전화 통화와 같이 송신하는쪽과 수신하는쪽이 쌍방 합의하에만 달성된다. 반면에 비동기 처리는 이메일, 카톡등의 메신저에 해당하게 된다.
이메일과 카톡같은 것들은 갑작스럽게 이벤트가 생겨도 지금 당장 처리하는 것이 아니라 시간이 날 때 처리하는 것 들인데 이와 같이 언제 이벤트가 발생할지 모르는 것들을 처리 해야할 때 그 상황에 맞추어 잘 처리하도록 맞추는 것이 함수 포인터이다.

문제1)

```
#include <stdio.h>


float pof_test1(int n1, int n2)
{
return (n1 + n2) * 0.23573;
}


int pof_test2(float n1, double n2, int n3)
{
return (n1 + n2 + n3) / 3.0;
}


// int (*)(float, double, int) pof_test_main(float (*)(int, int))
// int (* pof_test_main(float (*)(int, int)))(float, double, int)
int (* pof_test_main(float (*p)(int, int)))(float, double, int)
{
float res;
res = p(4, 3);
printf("internal ptm res = %f₩n", res);
return pof_test2;
}


int main(void)
{
int res;
res = pof_test_main(pof_test1)(3.7, 2.4, 7);
printf("pof_test_main res = %d₩n", res);
return 0;
}
```

~결과

main함수에서 변수 res가 선언되고 pof_test_main(pof_test1)(3.7, 2.4, 7);의 리턴값이 저장된다. pof_test_main함수에서 p(pof_test1)을 인자로 받아 pof_test1의 리턴값을 res에 저장한다. 7*0.23573이 출력되고 pof_test2로 리턴되어 (3.7+2.4+7)/3.0의 결과값이 리턴되어 출력된다.

문제2)

```
#include <stdio.h>

//int(*)(int, int)
int pof1(int n1, int n2)
{
return n1 + n2;
}


//int(*)(int, int) subpof1(int)
int (* subpof1(int n))(int, int)
{
printf("n = %d₩n", n);
return pof1;
}


//float (*)(int, double)
float pof2(int n1, double n2)
{
return n1 * n2;
}

//int(*)(int, int) (*)(int) pof_test_main(float (*)(int, double))
//int(*)(int, int) (* pof_test_main(float (*)(int, double)))(int)



int(*(* pof_test_main(float (*p)(int, double)))(int))(int, int)
//int(*)(int, int) (*)(int) pof_test_main(float (*p)(int, double))
{
float res;
res = p(3, 7.7);
printf("res = %f₩n", res);
return subpof1;
}


int main(void)
{
int res;
res = pof_test_main(pof2)(3)(7, 3);
printf("res = %d₩n", res);
return 0;
}
```

~결과

int형 변수 res에 pof_test_main(pof2)(3)(7, 3);의 리턴값을 넣는데 각 인자들은 색으로 표시된 것 들이다. float형 res변수가 선언되고 p(pof2)에 인자로 3과 7.7이 입력되고 3과 7.7이 곱해진 값이 res에 저장된 후 출력된다. 23.1을 인자로 받아 출력하고 pof1으로 리턴하게 된다. subpof1에서는 인자로 받은 int n을 출력하게 되어 3을 출력한다. 그리고 pof1으로 리턴하게되고 인자로 받은 7과 3을 더한 10을 리턴하여 main함수로 돌아와 res인 10을 출력하게 된다.

-memmove()
Memory Move의 합성어이다. 메모리의 값을 복사할 때 사용함
memove(목적지, 원본, 길이);로 사용함
memcpy보다 느리지만 안정적이다. memmove(복사할위치, 복사할것, 몇 바이트복사할것인지) memmove보다 memcpy가 더 빠르지만 성능을 중요시 할때에는 memmove사용한다.

예제)
```
#include<stdio.h>
#include<string.h>

int main(void)
{
int i=0;
int src[5]={1,2,3,4,5};
int dst[5];

memmove(dst,src,sizeof(src));

for(i=0;i<5;i++)
printf("dst[%d]=%d₩n",i,dst[i]);
return 0;

}
```

~결과
```
dst[0]=1
dst[1]=2
dst[2]=3
dst[3]=4
dst[4]=5
```

int형 변수와 배열들을 선언한다. 그리고 memmove를 사용하여 src를 복사하여 dst에 입력하는데 for문을 사용해 배열 요소 하나하나의 값을 출력하게 한다.

-memcpy
memory 공간에 겹치는 부분이 없을때 사용한다. 겹치는 부분이 없다면 성능에 좋음.

예제)
```
#include<stdio.h>
#include<string.h>

int main(void)
{
char src[30]="This is amazing";
char *dst=src+3;

printf("before memmove=%s₩n",src);
memcpy(dst,src,3);
printf("after memmove=%s₩n",dst);
return 0;
}
```
~결과
before memmove=This is amazing
after memmove=This amazing

src라는 배열에 This is amazing를 입력하고 포인터 dst에는 src에 +3을 한 것을 입력한다. 그리고 src를 출력하면 before memmove=This is amazing라는 결과가 출력하게 되는데 이후 memcpy를 사용해 src를 dst에 복사하여 출력하면 after memmove=This amazing라는 문장이 출력되게 된다. is가 빠지고 이렇게 된 이유는 위에 char *dst=src+3; 이 문장이기 때문인데 3을 더해서 Thi가 잘리고 그 곳부터 T가 시작되어 This amazing이 출력하게 되었다.

-strlen
예제)

```
#include<stdio.h>
#include<string.h>

int main(int argc,char **argv)
{
char *str="this is the string";
int len =strlen(str);
printf("len=%d₩n",len);
return 0;
}
```

~결과
len=18

main함수에서 argc와 **argc를 인자로 받아온다.
char형 str포인터에 "this is the string" 라는 문장을 저장하고 int형 변수 len에 위의 문장의 길이를 저장하고 이를 출력하면 18의 값이 출력된다.

의문점 > argc와 **argc가 어디에 쓰이는지 모르겠다.
-strcpy, strncpy

예제)

```
#include<stdio.h>
#include<string.h>

int main(int argc,char **argv)
{
char src[20];
char dst[20];
strcpy(src,"pretty");
strcpy(dst,src);
printf("dst=%s₩n",dst);
return 0;
}
```

~결과
dst=pretty

char형 배열 src과 dst를 선언한다. src에 ‘pretty‘라는 문장을 복사하고 dst에 src의 문장을 복사한다. 이를 출력해보았을 때 결과는 dst=pretty가 된다.

-strcmp
문자열이 서로 같은지 비교하고 싶을 때 사용한다.
서로 같은 경우 0을 반환하게됨
strcmp(str1, str2), strncmp(str1, str2, len)처럼 사용

예제)
```
#include<stdion.h>
#include<string.h>
//trcmp(str1, str2), strncmp(str1, str2, len)처럼 사용
int main(int argc,char **argv)
{
char src[20]="Korea";
char dst[20]="korea";

if(!strcmp(src,dst))
printf("src,dst는 서로같음₩n");
else
printf("src,dst는 서로다름₩n");
return 0;
}
```

~결과
src,dst는 서로같음

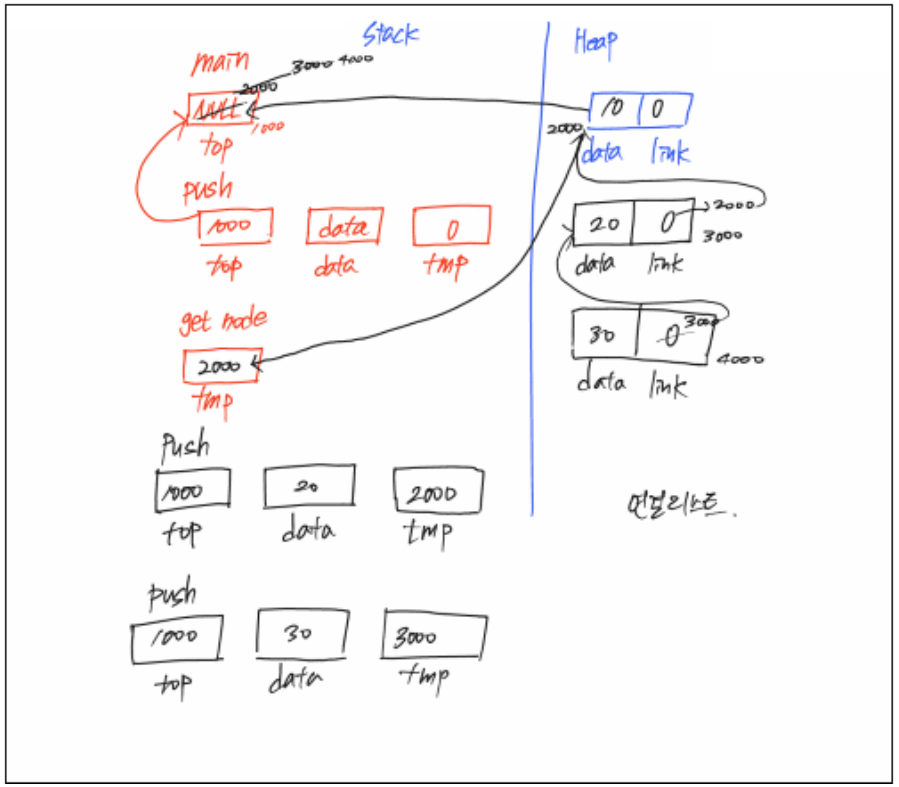
src라는 배열에 Korea라는 문장을 넣고 dst라는 배열에 똑같이 Korea라는 문자를 넣는다. 그리고 if문을 사용하여 src와 dst가 같으면 0을 출력하기 때문에 !를 이용하여 같으면 ‘src,dst는 서로같음‘을 출력하고 다르면 ’src,dst는 서로다름‘을 출력한다. 위의 예제는 src와 dst가 같으므로 ’src,dst는 서로같음’ 가 출력된다.

-stack

예제)

```
#include<stdio.h>
#include<malloc.h>
#include<string.h>
#include<stdlib.h>
#define EMPTY 0

struct node{
int data;
struct node *link;
};
typedef struct node Stack;

Stack *get_node()
{
Stack *tmp;
tmp=(Stack*)malloc(sizeof(Stack));
tmp->link=EMPTY;
return tmp;
}

void push(Stack **top,int data)
{
Stack *tmp;
tmp= *top;
*top=get_node();
(*top)->data =data;
(*top)->link=tmp;
}

int pop(Stack **top)
{
Stack *tmp;
int num;
tmp=*top;
if(*top==EMPTY)
{
printf("Stack is empty!!₩n");
return 0;
}

num=tmp->data;
*top=(*top)->link;
free(tmp);
return num;
}

int main(void)
{
Stack *top=EMPTY;
push(&top,10);
push(&top,20);
push(&top,30);
printf("%d₩n",pop(&top));
printf("%d₩n",pop(&top));
printf("%d₩n",pop(&top));
printf("%d₩n",pop(&top));
return 0;
}
```

~결과
30
20
10
Stack is empty!!
0



Stack *get_node()에서 부터 return 이후를 잘 모르겠다...