Computer Architecture Exercises

Qiyang Hu, Givi Meishvili, Adrian Wälchli

March 13, 2018

Today

- Review of Series 1
  - Theoretical Part
  - Programming Part
- Introduction to Series 2
  - Theoretical Part
  - Programming Part

Review of Series 1

Exercise 1

...	’c’	’o’	’m’	’p’	’u’	’t’	’e’	’r’	’ ’
’a’	’r’	’c’	’h’	’i’	’t’	’e’	’c’	’t’	’u’
’r’	’e’	0	...						

$\Rightarrow$ 21 bytes for the letters plus one byte for the null terminator = 22 bytes

Exercise 2

```
double getAt(double *a, int i) {
    return *(a+i);
}

int getAt(int *a, int i) {
    return *(a+i);
}

short getAt(short *a, int i) {
    return *(a+i);
}
```

We can get out of the array range!!!

⇒ Pointer arithmetic

Exercise 2

```
#define N 10

double a[N];

double getAt(int i) {
        if(i < N){
                return a[i];
        }
        printf("index␣out␣of␣bound\n");
        return -1;
}
```

⇒ Pointer arithmetic

Exercise 3

```
long a = 1234567890;  /* Hex: 499602d2 */
long b = 1000000000;   /* Hex: 3b9aca00 */
```

Exercise 3

```
long a = 1234567890;   /* Hex: 499602d2 */
long b = 1000000000;    /* Hex: 3b9aca00 */
```

	Address	Content (Hex)
&b	bffff844	00
	bffff845	ca
	bffff846	9a
	bffff847	3b
&a	bffff848	d2
	bffff849	02
	bffff84a	96
	bffff84b	49

- Little Endian
- Least significant byte?
- GNU C: `sizeof(void)`= 1

Exercise 3

	Address	Content (Hex)
&b	bffff844	00
	bffff845	ca
	bffff846	9a
	bffff847	3b
&a	bffff848	d2
	bffff849	02
	bffff84a	96
	bffff84b	49

- Little Endian
- Least significant byte?
- GNU C: sizeof(void)= 1

```
void * p = &b;
printf("%x\n", p);                         /* bffff844 */
printf("%x\n", *(long*)p++);               /* 3b9aca00 */
printf("%x\n", *(char*)p++);               /* ffffffca */
printf("%x\n", *(unsigned char*)p++);/* 9a */
printf("%x\n", p);                         /* bffff847*/
```

Exercise 4

```
int main () {
  int i, j;
  i = 103;
  j = increment(&i);
}

/* Version 1 */
int increment(int *x) {
  return ++(*x);
}
```

- i = 104
- j = 104

Exercise 4

```
int main () {
  int i, j;
  i = 103;
  j = increment(&i);
}

/* Version 2 */
int increment(int *x) {
  return (*x)++;
}
```

- i = 104
- j = 103

Exercise 5

```
short x[3] = {3, 2, 1};
short *px = x;
printf("%i␣␣%i\n", *x, *px);
px++;
printf("%i␣␣%i\n", *x, *px);
```

Output

```
3 3
3 2
```

Exercise 5

```
short x = 3;
short *px = &x;
*(px--) = 20;
*px = 21;
printf("%i␣␣%i\n", x, *px);
```

(Possible) Output

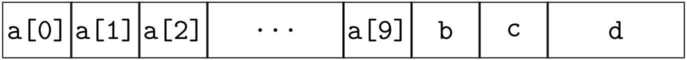
```
20 21
```

- Access to “unsafe” memory

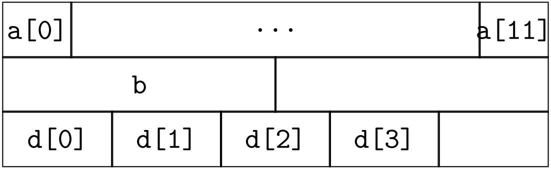
Exercise 6

```
struct {
  char a[10];
  char b;
  char c;
  short int d;
} myStruct;
```

```
union {
  char a[12];
  int b;
  short int d[4];
} myUnion;
```



a[0] a[1] a[2] ··· a[9] b c d



a[0] ··· a[11]

b

d[0] d[1] d[2] d[3]

sizeof(myStruct)=14

sizeof(myUnion)=12

Exercise 7

```
#define callA callB(13)

void callB(int a) {
    printf(”%i\n”, a+2);
}

int main() {
  callA;
  return EXIT_SUCCESS;
}
```

- `define` means “Search & Replace”

Exercise 7

```
void callB(int a) {
    printf("%i\n", a+2);
}

int main() {
  callB(13);
  return EXIT_SUCCESS;
}
```

- Output: 15

Programming Part: InstructionTypeI

```
typedef struct {
    unsigned immediate : 16;
    unsigned rt : 5;
    unsigned rs : 5;
    unsigned opcode : 6;
} InstructionTypeI;
```

Programming Part: Instruction and InstructionType

```c
typedef union {
    InstructionTypeI i;
    InstructionTypeJ j;
    InstructionTypeR r;
} Instruction;

typedef enum {iType, jType, rType, specialType
   } InstructionType;
```

Programming Part: Operation and Function

```
typedef struct {
    char name[OP_NAME_LENGTH+1];
    InstructionType type;
    void (*operation)(Instruction*);
} Operation;

typedef struct {
    char name[FUNC_NAME_LENGTH+1];
    void (*function)(Instruction*);
} Function;
```

Programming Part: `printInstruction`

```
void printInstruction(Instruction *i) {
    Operation o = operations[i->i.opcode];
    Function f = functions[i->r.funct];
    switch (o.type) {
        case iType:
            printf("%-4s␣%02i,␣%02i,␣0x%04x\n"
               , o.name, i->i.rt, i->i.rs, i->
               i.immediate );
            break;
        case jType:
        /* ...and so on... */
```

Introduction to Series 2

Pointers

```
int* a;
int *b;
int* c, d;
int *e, f;
```

lvalues vs. rvalues

lvalues *locator* values

rvalues *readable* values

```
int x = 3;      /* x is lvalue, 3 is rvalue  */
int *px = &x;   /* &x is rvalue              */
(*px)++;        /* *px is lvalue             */
3 + x;          /* addition has two rvalues
                 * as argument, returns rvalue
                 * each lvalue is an rvalue  */
3++;            /* error: lvalue required as
                 * increment operand         */
(x++)++;        /* error: lvalue required as
                 * increment operand         */
                /* increment returns  rvalue*/
```

Big vs. Little Endian

$$\underbrace{\texttt{1234}}_{\text{MSB}}\ \texttt{ABCD 5678}\ \underbrace{\texttt{EF90}}_{\text{LSB}}$$

LSB	least significant byte
MSB	most significant byte
Big Endian	Word address = address of MSB
Little Endian	Word address = address of LSB

C Datatypes

`int` signed, at least 16 bit long

`unsigned int` only positive integers, at least 16 bit long

`intN_t` signed, *exactly* `N` bits long

`uintN_t` unsigned, *exactly* `N` bits long

Listing 1: mips.h

```c
typedef uint32_t word;
typedef uint16_t halfword;
typedef uint8_t byte;
```

General Remarks

- You are not allowed to remove or modify the tests we provide.
- A failing test is a good indication that there is a problem with your implementation.
- When in doubt, ask in our forum on ILIAS.
- Specifications of the MIPS operators can be found in Patterson-Hennessy. See also ILIAS folder “Literatur”.