CSM 61C Advanced RISC-V and Intro to Caching

Spring 2019 Mentoring 4: February 10, 2019

1 Number Representation

1.1 What is the range of integers represented by a n-bit binary number? Your answers should include expressions that use 2^n.

(a) Unsigned:

$[0, 2^n - 1]$

(b) Two's Complement:

$[-2^{n-1}, 2^{n-1} - 1]$

(c) One's Complement:

$[-(2^{n-1} - 1), 2^{n-1} - 1]$

(d) Bias (with bias b):

$[-b, 2^n - 1 - b]$

1.2 How many unique integers can be represented in each case?

(a) Unsigned:

2^n

(b) Two's Complement:

2^n

(c) One's Complement:

$2^n - 1$

(d) Bias (with bias b):

2^n

For both unsigned and twos complement, each bit string corresponds to a different integer, so we have 2^n unique integers.

Ones complement is an exception: we have two zeroes. 2^n treats these two zeroes separately, but really the two are indistinguishable. Therefore we need to subtract one possibility to get $2^n - 1$.

Bias is just a shifted version of unsigned and so it can represent the same number of integers.

2 C Memory Locations

2.1 Consider the following C program:

```
int a = 5;
int main()
{
    int b = 0;
    char* s1 = cs61c;
    char s2[] = cs61c;
    char* c = malloc(sizeof(char) * 100);
    return 0;
};
```

For each of the following values, state the location in the memory layout where they are stored. Answer with code, static, heap, or stack.

(a) `s1`

stack

(b) `s2`

stack

(c) `s1[0]`

static

(d) `s2[0]`

static

(e) `c[0]`

heap

(f) `a`

static

2.2 Consider the C code here, and assume the malloc call succeeds. Rank the following values from 1 to 5, with 1 being the least, right before bar returns. Use the memory layout from class; Treat all addresses as unsigned numbers.

```
#include <stdlib.h>

int FIVE = 5;

int bar(int x) {
    return x * x;
}

int main(int argc, char *argv[]) {
    int *foo = malloc(sizeof(int));
    if (foo) free(foo);
    bar(10); //  snapshot just before it returns
    return 0;
}
```

```
foo:   _______
&foo:  _______
FIVE:  _______
&FIVE: _______
&x:    _______
```

```
foo:   3
&foo:  5
FIVE:  1
&FIVE: 2
&x:    4
```

Going in order numerically, `FIVE` itself contains the value 5, `&FIVE` contains the address of `FIVE` and since it is a global variable, it is stored statically, which means that it will be stored in the data segment. Since `foo` is a pointer, it contains the address of whatever it was assigned to, which in this case, is `malloc(sizeof(int))`. As a result, `foo` is stored on the heap, so it is above the data segment and so the value it contains will be larger than the address of `FIVE`, since the heap is above the data segment. `&foo` itself lives in on the stack, since the space that it takes to store the pointer to the data that foo holds is allocated on the stack. This is above the heap. `x` is a local variable, so it also gets allocated on the stack so its address is also greater than the value stored in `foo`; the reason `&x` is smaller than `&foo` is simply because the stack grows downwards, and during the execution of the program, the space for `foo` is allocated before the space for `x`, so foo lives in higher memory than `x`.

3 Linked Lists

3.1 Fill out the declaration of a singly linked linked-list node below.

```
typedef struct node {
    int value;
    ____________ next; // pointer to the next element
} sll_node;
```

```
struct node* next;
```

Remember the pointer to the next node in a linked list is one pointing to another node, so the type of next is a pointer to the same type as the first linked list node.

3.2 Let's convert the linked list to an array. Fill in the missing code.

```
int* to_array(sll_node *sll, int size) {
    int i = 0;
    int *arr = _________________;
    while (sll) {
        arr[i] = ___________;
        sll = ___________;
        _______;
    }
    return arr;
}
```

```
int* to_array(sll_node *sll, int size) {
    int i = 0;
    int *arr = malloc(size * sizeof(int));
    while (sll) {
        arr[i] = sll->value;
        sll = sll->next;
        i++;
    }
    return arr;
}
```

Converting the linked list to an array requires traversing the linked list. But first, you must allocate enough space to store `size` number of integers. Then, you can go ahead and iterate over the linked list. Assign to the array each corresponding linked list value. Move the pointer of the linked list and increment the array counter after each assignment.

3.3 Finally, complete the function `delete_even()` that will delete every second element of the list. For example, given the lists below:

```
Before: Node 1  Node 2  Node 3  Node 4
After: Node 1  Node 3
```

Calling `delete_even()` on the list labeled "Before" will change it into the list labeled "After". All list nodes were created via dynamic memory allocation.

```
void delete_even(sll_node *s11) {
    sll_node *temp;
    if (!sll || !sll->next) {
        return;
    }
    temp = ________;
    sll->next = ________;
    free(________);
    delete_even(________);
}
```

```
void delete_even(sll_node *s11) {
    sll_node *temp;
    if (!sll || !sll->next) {
        return;
    }
    temp = sll->next;
    sll->next = temp->next (or sll->next->next);
    free(temp);
    delete_even(sll->next);
}
```

4 RISC-V to C

4.1 Assume we have two arrays input and result. They are initialized as follows:

```
int *input = malloc(8*sizeof(int));
int *result = calloc(8, sizeof(int));
for (int i = 0; i < 8; i++) {
    input[i] = i;
}
```

You are given the following RISC-V code. Assume register x10 holds the address of input and register x12 holds the address of result.

```
    add x8, x0, x0
    addi x5, x0, 8
    addi x11, x0, 0
    addi x12, x0, 0
Loop:
    beq x5, x11, Done
    lw x6, 0(x10)
    add x8, x8, x6
    slli x7, x5, 2
    add x7, x7, x12
    sw x8, 0(x7)
    addi x5, x5, -1
    addi x10, x10, 4
    j Loop
Done:
    // exit
    .
```

```
// sizeof(int) == 4
int sum = 0;
```

for (int i = 0; i ¡ 8; i++) sum += a[i]; c[i] = sum;

4.2 What is the end array stored starting at register x12?

$[0, 1, 3, 6, 10, 15, 21, 28]$

Meta: This is a challenging question for students since they are all new to RISC-V. Make sure to walk through and write out each single RISC-V instruction functionality first. Since the lecture will not cover the detailed name for each register, it will be good just going along with x0 - x31. Drawing all the detailed memory diagram will be helpful for this question since it involves a lot of load and store instructions.

5 Advanced RISC-V

5.1 You are given the following RISC-V code:

```
Loop:   andi t2 t1 1
        srli t3 t1 1
        bltu t1 a0 Loop
        jalr s0 s1 MAX_POS_IMM
```

(a) What is the value of the byte offset that would be stored in the immediate field of the bltu instruction?

-8

(b) What is the binary encoding of the bltu instruction? Please use hexadecimal to represent your answer.

0xFEA36CE3

5.2 As a curious 61C student, you question why there are so many possible `opcode`s, but only 47 instructions. Thus, your propose a revision to the standard 32-bit RISC-V instruction formats where each instruction has a unique `opcode` (which still is 7 bits). You believe this justifies taking out the `funct3` field from the `R`, `I`, `S`, and `SB` instructions, allowing you to allocate bits to other instruction fields except the `opcode` field.

(a) What is the largest number of registers that can now be supported in hardware?

64

(b) With the new register size, how far can a `jal` instruction jump to (in halfwords)?

$[-2^{18}, 2^{18-1}]$

(c) Assume register `s0 = 0x1000 0000`, `s1 = 0x4000 0000`, `PC = 0xA000 0000`. Lets analyze the instruction `jalr s0, s1, MAX_POS_IMM` where `MAX_POS_IMM` is the maximum possible positive immediate for `jalr`. Using the register sizes defined above, what are the values in registers `s0`, `s1`, and `pc` after the instruction executes?

```
s0 = 0xA000 0004
s1 = 0x4000 0000
pc = 0x4000 0FFF
```