Computation Structures

October 2015

Assignement 1

Binary Trees and β-Assembly

- **Deadline:** 20/10/2015 23:59.
- Late submissions will result in a penalty.
- Questions will no longer be answered 24h before the deadline.
- English is strongly encouraged.
- Contact: `dtaralla@ulg.ac.be`, Office 2.96 (B28)

---

1 Introduction

The goal of this assignment is to make you more familiar with the β-assembly language by writing a simple yet complete program in β-assembly. You should be familiar with the algorithm, and you probably already have implemented it in at least two high-level languages: it is the construction of an ordered binary tree.

2 Assignment

2.1 Goal

A node in the tree contains an integer value and pointers to its two children (left and right), one or both of which could be non existing. Pointers to non-existing children are set to the value `NULL` (i.e., 0). For a node with value v, all nodes to its left have values less than or equal to v, and all nodes to its right have values strictly greater than v. Figure 2 shows an example of such a tree. A program in C for building an ordered binary tree by inserting values into it is shown on Figure 1.

Your goal is to implement this tree building algorithm in β-assembly. To help you experiment with and test your code, a simulator (`bsim.jar`) is provided. You will insert your code in the provided `tree.uasm` file at the intended locations (see the comments in that file).

The file `config.uasm` contains initialization code, some useful constants, as well as data available to your program. **You can not modify this file**. It contains the following global variables:

- `next_mem` — The address of the next free byte in memory.
- `tree_loc` — The address of the tree, i.e. of its root node.
- `list` — The array of values to insert in the tree. Its length is `ARGV_NVAL`.

```c
typedef struct Node_t Node;
struct Node_t {
    int value;
    Node* left;
    Node* right;
}

void add_node(Node* n, int value); // add a node of value v to tree n
Node* create_node(int value);      // allocate and initialize a node; not given

int main(int argc, char** argv) {
    int* valuesToInsert;
    int n;
    // ...

    // valuesToInsert contains the values to insert in the tree
    // n contains the number of values in valuesToInsert

    Node* tree = create_node(valuesToInsert[0]);

    for (int i = 1; i < n; i += 1)
        add_node(tree, valuesToInsert[i]);

    return 0;
}

void add_node(Node* n, int v) {
    if (v <= n->value) {
        if (n->left == NULL)
            n->left = create_node(v);
        else
            add_node(n->left, v);
    }
    else {
        if (n->right == NULL)
            n->right = create_node(v);
        else
            add_node(n->right, v);
    }
}
```

Figure 1: C program building an ordered binary tree of integers.

For instance, to load into `R1` the address of the next free byte in memory, you can use `LD(R31, next_mem, R1)`. To load into `R1` the `Reg[R2]`-th integer of `list` (starting with 0), you can use `MULC(R2, 4, R1)` followed by `LD(R1, list, R1)`.

2.2 Tree Representation

Your tree, that is its root node, will be located in main memory (RAM) at the address `tree_loc`, which is in fact the last valid address of the simulator's memory (`0x0003FFFC`). The tree is going to grow towards **decreasing** addresses. Each node will occupy three 32-bit words representing, in the given order:

1. the value of the node;
2. the address of its left child;
3. the address of its right child.

Take for instance the root node, whose address is `tree_loc` = `0x0003FFFC`. It means that `tree_loc - 12` is the address of the node that was allocated just after the root. The three elements of the rood node of the tree can thus be found at the following addresses (with `tree_loc` = `0x0003FFFC`):

- the value is located at `tree_loc + NODE_VALUE` = `tree_loc - 0`.
- the left child is located at `tree_loc + NODE_LEFT` = `tree_loc - 4`.
- the right child is located at `tree_loc + NODE_RIGHT` = `tree_loc - 8`.

For example, for the tree in Figure 2, the corresponding memory content is shown in Table 1.

2.3 Creating a Node

To create a node, you will use the global variable `next_mem`. This global variable has the value `NULL` at the start of the program; it will be your job to update it so that it to always points to the next free node location. For example, in your `main` code, after creating the root node at `tree_loc`, you will set the value of `next_mem` to `tree_loc + NODE_OFFSET` = `tree_loc - 12`. To create a node with value v, you execute

1. `RAM[RAM[next_mem] + NODE_VALUE]` $\leftarrow v$
2. `RAM[RAM[next_mem] + NODE_LEFT]` $\leftarrow 0$
3. `RAM[RAM[next_mem] + NODE_RIGHT]` $\leftarrow 0$
4. `RAM[next_mem]` $\leftarrow$ `RAM[next_mem] + NODE_OFFSET`

2.4 Working Example

As stated above, at the beginning of your program you are provided with a predefined list of numbers to insert in a tree. This list is $(5, 3, 6, 1, 8, 4)$, in this order, which means that the resulting tree should be the one shown in Figure 2. According to the tree representation we have chosen, the memory should look like Table 1. Thus, when we run your code, we expect that it will leave the upper part of the memory in this exact state.[1]

[1] Do not try to store directly the values in memory without actually implementing the tree building algorithm. You have been warned...

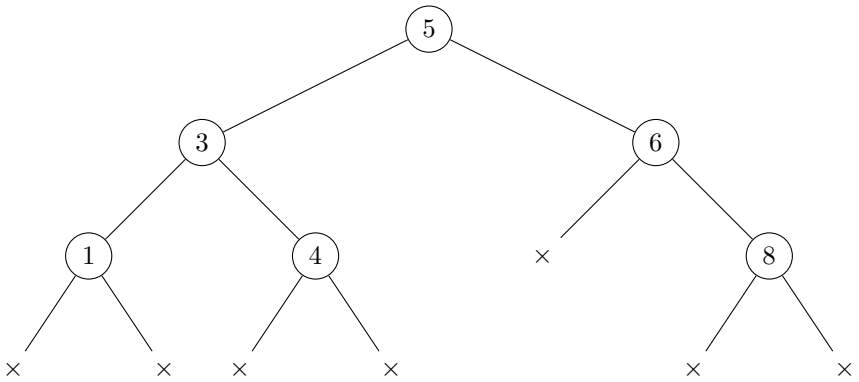

Figure 2: The tree obtained by inserting the values $5, 3, 6, 1, 8, 4$.

⋮	⋮
0x3FFB8	00000000
0x3FFBC	00000000
0x3FFC0	00000004
0x3FFC4	00000000
0x3FFC8	00000000
0x3FFCC	00000008
0x3FFD0	00000000
0x3FFD4	00000000
0x3FFD8	00000001
0x3FFDC	0003FFCC
0x3FFE0	00000000
0x3FFE4	00000006
0x3FFE8	0003FFC0
0x3FFEC	0003FFD8
0x3FFF0	00000003
0x3FFF4	0003FFE4
0x3FFF8	0003FFF0
0x3FFFC	00000005

Table 1: The memory configuration corresponding to the tree obtained from $(5, 3, 6, 1, 8, 4)$.

3 Additional Guidelines

3.1 Practical Organization

In order to learn β-assembly effectively, **this assignment will be done individually**. A report of **maximum two pages** *can* be provided if you want to explain things that are not easy to understand by just looking at the code and comments. Providing a report does not necessarily mean that you will earn a better grade; it should be provided only if it brings something that is not mentioned clearly elsewhere.

Plagiarism is of course not allowed and severely punished. Any detected attempt will result in the grade 0/20 for all who have participated in this practice.

You will include your completed `tree.uasm` and your (optional) report (PDF only) in a ZIP archive named `sXXXXXX_NAME.zip` where `sXXXXXX` is your student ID and `NAME` your family name in uppercase. Insert your `tree.uasm` in a ZIP archive even if you do not provide a report.

Submit your archive to the **Montefiore Submission Platform**[2], after having created an account if necessary. If you encounter any problem with the platform, let me know. However problems that unexpectedly and mysteriously appear five minutes before the deadline will not be considered. **Do not send your work by E-mail; it will not be read**.

3.2 Code Clarity and Efficiency

Choose a coding style and *stick to it*. You are advised to use the coding style used in `tree.uasm`. The goal here is not to write code which is as compact and efficient as possible, but to learn the concepts of β-assembly. However, your code should not be unreasonably long and inefficient: minimize the number of registers you use in your recursive procedures, as it impacts the growth rate of your stack.

3.3 Documentation

One of the challenges when writing assembly code is to write a program which is relatively easy to understand. Thus, the second most important element taken into account for your grade (after correctness) will be your code's readability. Use comments extensively (your comments can be larger than your code), but don't be *verbose* : explain the non obvious, not the immediately apparent. Moreover, follow the following conventions for documenting branches and procedures.

3.3.1 Documenting Procedures

Pre- and post- conditions should appear clearly: the arguments have to be properly defined and any return value as well as any modifications of global data documented. Register `R0` is generally used to store return values.

3.3.2 Documenting Branches

Branches are blocs of code that are executed between jump and branch instructions. All branches have to be preceded by some documentation containing the following information:

1. **Parent procedure:** Which procedure(s) this branch is a part of.
2. **Precondition:** For each register that will be read in this branch, a brief description of what the branch expects to see in it.

[2] `http://submit.run.montefiore.ulg.ac.be/`

3. **Postcondition:** A description of what happens in this branch, i.e. of what it does.
4. **Modifications:** For each register whose value was modified in this branch, a brief description of what it contains after its execution.

Example You will find below an example that shows an acceptable documentation style.

```
| inverse_truth(value): Complement the value given in argument and put the result in R0
inverse_truth:
    ...                       | initialize inverse_truth; load argument in R1
    BEQ(R1, make_true, R2)
    BNE(R1, make_false, R2)
    ...                       | return from function

| make_true [Part of "inverse_truth"]
| PRE:
|     R2 contains the address to return to once this branch is over
|
| POST: Put ‘‘true’’ in R0.
|     R0 contains 1
|
| MOD: N/A
make_true:
    CMOVE(1, R0)
    JMP(R2)

| make_false [Part of "inverse_truth"]
| PRE:
|     R2 contains the address to return to once this branch is over
|
| POST: Put ‘‘false’’ in R0.
|     R0 contains 0
|
| MOD: N/A
make_false:
    MOVE(R31, R0)
    JMP(R2)
```