JACOBS UNIVERSITY BREMEN

Assignment #9

Student name: *Ivan Kabadzhov*

Course: *Introduction to Computer Science*
Date: *November, 2018*

Problem 9.1: simple cpu machine code

a)

- The first 3 bits represent the op-code and it has according mnemonic representation (such as LOAD, STORE, ADD, etc.).
- The next bit determines if the operand is a constant (1) or a memory address (0). To represent a constant (1) in the assembly code I write down #, otherwise I leave it blank.
- The rest of the bits are for carrying the operand. So, in my machine code I have the corresponding code for a constant or a memory address, represented as a binary number. Whereas in the assembly code, I am converting it to decimal to make it clearer for the reader.

#	Machine code	Assembly code	Description
0	001 1 0001	LOAD #1	Load value 1 into the accumulator.
1	010 0 1111	STORE 15	Store the value of the accumulator in memory location 15.
2	001 1 0000	LOAD #0	Load value 0 into the accumulator.
3	101 1 0100	EQUAL #4	If the value in the accumulator = 4, skip the next step.
4	110 1 0110	JUMP #6	Jump to instruction 6 (set program counter to 6).
5	111 1 0000	HALT #0	Stop execution.
6	001 0 0011	LOAD 3	Load the value in memory location 3 into the accumulator.
7	100 1 0001	SUB #1	Subtract the current value of the accumulator with 1.
8	010 0 0011	STORE 3	Store the value of the accumulator in memory location 3.
9	001 0 1111	LOAD 15	Store the value of the accumulator in memory location 3.
10	011 0 1111	ADD 15	Add the value of memory location 15 to the accumulator.
11	010 0 1111	STORE 15	Store the value of the accumulator in memory location 15.
12	110 1 0010	JUMP #2	Jump to instruction 2 (set program counter to 2).
13	000 0 0000	0	no instruction/ data, initialized to 0
14	000 0 0000	0	no instruction/ data, initialized to 0
15	000 0 0000	0	no instruction/ data, initialized to 0

b) The only values that are going to change by our program are in the accumulator and memory locations 3 and 15.

#	Assembly code	Accumulator	Mem loc 3	Mem loc 15

0	LOAD #1	1	4	0
1	STORE 15	1	4	1
2	LOAD #0	0	4	1
3	EQUAL #4	0 (0!=4)	4	1
4	JUMP #6	0 (jump to instr 6)	4	1
5	HALT #0	jumped	4	1
6	LOAD 3	4 (from mem loc 3)	4	1
7	SUB #1	3 (4-1)	4	1
8	STORE 3	3	3	1
9	LOAD 15	1 (from mem loc 15)	3	1
10	ADD 15	2 (1+1)	3	1
11	STORE 15	2	3	2
12	JUMP #2	2 (jump to instr 2)	3	2
13	0			
14	0			
15	0			

► Cycle 1

#	Assembly code	Accumulator	Mem loc 3	Mem loc 15
2	LOAD #0	0	3	2
3	EQUAL #3	0 (0!=3)	3	2
4	JUMP #6	0 (jump to instr 6)	3	2
5	HALT #0	jumped	3	2
6	LOAD 3	3 (from mem loc 3)	3	2
7	SUB #1	2 (3-1)	3	2
8	STORE 3	2	2	2
9	LOAD 15	2 (from mem loc 15)	2	2
10	ADD 15	4 (2+2)	2	2
11	STORE 15	4	2	4
12	JUMP #2	4 (jump to instr 2)	2	4
13	0			
14	0			
15	2			

► Cycle 2

#	Assembly code	Accumulator	Mem loc 3	Mem loc 15
2	LOAD #0	0	3	4
3	EQUAL #2	0 (0!=2)	2	4
4	JUMP #6	0 (jump to instr 6)	2	4
5	HALT #0	jumped	2	4
6	LOAD 3	2 (from mem loc 3)	2	4
7	SUB #1	1 (2-1)	2	4
8	STORE 3	1	1	4
9	LOAD 15	4 (from mem loc 15)	1	4
10	ADD 15	8 (4+4)	1	4

11	STORE 15	8	1	8
12	JUMP #2	8 (jump to instr 2)	1	8
13	0			
14	0			
15	4			

► Cycle 3

#	Assembly code	Accumulator	Mem loc 3	Mem loc 15
2	LOAD #0	0	1	8
3	EQUAL #1	0 (0!=1)	1	8
4	JUMP #6	0 (jump to instr 6)	1	8
5	HALT #0	jumped	1	8
6	LOAD 3	1 (from mem loc 3)	1	8
7	SUB #1	0 (1-1)	1	8
8	STORE 3	0	0	8
9	LOAD 15	8 (from mem loc 15)	0	8
10	ADD 15	16 (8+8)	0	8
11	STORE 15	16	0	16
12	JUMP #2	16 (jump to instr 2)	0	16
13	0			
14	0			
15	8			

► Cycle 4

#	Assembly code	Accumulator	Mem loc 3	Mem loc 15
2	LOAD #0	0	0	16
3	EQUAL #0	0 (0==0)	0	16
4	JUMP #6	skipped	0	16
5	HALT #0	end	0	16
6	LOAD 3			
7	SUB #1			
8	STORE 3			
9	LOAD 15			
10	ADD 15			
11	STORE 15			
12	JUMP #2			
13	0			
14	0			
15	16			

► Cycle 5

Additionally I am providing a C++ program to show the functionality of the machine code.

```
#include <iostream>
using namespace std;
int main(){ //cycles are defined the way I described them
    cout << "BEGINNING OF CYCLE" << endl;
    int memorylocation15 = 0; //initially set to 0 by default
    memorylocation15 = 1; //LOAD#1, STORE15
    int memorylocation3 = 4;
    cout <<"[Memory_location_3] = " << memorylocation3 << endl;
    cout <<"[Memory_location_15] = " << memorylocation15 << endl;
    cout << "END OF CYCLE" << endl;
    while (memorylocation3 != 0){ //LOAD#0, EQUAL#4
        cout << endl << "BEGINNING OF CYCLE" << endl;
        memorylocation3--; //LOAD3, SUB#1, STORE3
        cout <<"[Memory_location_3] = " << memorylocation3 << endl;
        memorylocation15+=memorylocation15; //LOAD15, ADD15, STORE15
        cout <<"[Memory_location_15] = " << memorylocation15 <<endl;
        cout << "END OF CYCLE" << endl; //JUMP#2
        if (memorylocation3 == 0) //HALT #0
            cout << endl << "STOP EXECUTION" << endl << endl;
    }
    cout <<"final [Memory_location_15] = " <<memorylocation15<<endl;
    return 0;
}
```

Problem 9.2: fold function duality theorems

Consider: ||, && - as associative operations, e - as a neutral element

a) Claim:

`(x || y) || z = x || (y || z)`
`e || x = x` and `x || e = x`
$\overset{?}{\Rightarrow}$ `foldr || e xs = foldl || e xs`

Proof: *Proof by Induction*
Base case (n = 1): only one element represented as `[xs]`
On the one hand, we know that `x || e = x` ⇒ `xs || e = xs`
Thereofre, `foldr (||) e [xs] = xs || e`
On the other hand, `e || x = x` ⇒ `e || xs = xs`
Hence, `foldl (||) e [xs] = e || xs`
⇒ `foldr (||) e [xs] = xs || e = e || xs = foldl (||) e [xs]`
Inductive Hypothesis: `foldr (||) e xs = foldl (||) e xs`
Inductive Step: Applying hypothesis for `(x:xs)`
`foldr (||) e (x:xs) = x || foldr (||) e xs`
`x || foldr (||) e xs = foldl (||) e (x || e) xs`
⇒ `foldl (||) (e || x) xs = foldl (||) e (x:xs)`
⇒ `foldr (||) e (x:xs) = foldl (||) e (x:xs)`
∴ By substituting `x:xs` with `xs`, `foldr || e xs = foldl || e xs`.

■

b) Claim:
`x || (y && z) = (x || y) && z`
`x || e = e && x`
$\overset{?}{\Rightarrow}$ `foldr || e xs = foldl && e xs`

Proof: *Proof by Induction*
Base case (n = 0): empty list `[]`
`foldr (||) e [] = _ || e = e`
`foldl (&&) e [] = e && _ = e` $\Rightarrow$ `foldr (||) e [] = foldl (&&) e [] = e` **Inductive Hypothesis:** `foldr || e xs = foldl && e xs`
Inductive Step: Applying hypothesis for `(x:xs)`
LHS = `foldr (||) e (x:xs) = x || foldr (||) e xs = x || foldl (&&) e xs`
RHS = `foldl (&&) e (x:xs) = foldl (&&) (e && x) xs`
From `x || e = e && x` $\Rightarrow$ `foldl (&&) (x || e) xs = foldl (&&) (x || e) xs`
From `foldl`, `foldr` definitions $\Rightarrow$ `x || foldl (&&) e xs = foldl (&&) (x || e) xs`
From here observe that: `foldr (||) e (x:xs) = x || foldr (||) e xs =`
`= x || foldl (&&) e xs = foldl (&&) (x || e) xs = foldl (&&) (e && x) xs =`
`= foldl (&&) e (x:xs)`
$\Rightarrow$ `foldr (||) e (x:xs) = foldl (&&) e (x:xs)`
$\therefore$ By substituting `x:xs` with `xs`, `foldr || e xs = foldl && e xs`.

■

c) Claim:
`foldr || a xs = foldl ||' a (reverse xs)`
$\overset{?}{\Rightarrow}$ `x ||' y = y || x`
Definitions:
```
flip ::  (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```
Therefore rewrite the problem as:
`foldr || a xs = foldl (flip ||) a (reverse xs)`
$\overset{?}{\Rightarrow}$ `x (flip ||) y = y || x`

Proof: *Proof by Induction*
Base case (n = 0): empty list `[]`
`foldr (||) a [] = []`
`foldl (flip ||) a [] = []`
$\Rightarrow$ `foldr (||) a [] = foldl (flip ||) a []`
Inductive Hypothesis: `foldr || a xs = foldl (flip ||) a (reverse xs)`
Inductive Step: Applying hypothesis for `(x:xs)`
LHS = `foldr || a (x:xs) = foldr (flip ||) a (reverse (x:xs)) =`
`= foldr (flip ||) a ((reverse xs) ++ [x])`

RHS = `foldl (flip ||) a (reverse (x:xs)) = foldl || a (x:xs) =`

```
= foldl (||) (a || x) xs = foldl (||) ((flip ||) x a) xs =
= foldl (||) ((flip ||) x a) xs = foldr (flip ||) (flip ||) x a) (reverse xs)=
= foldr (flip ||) ((flip ||) x (foldr (flip ||) a [])) (reverse xs) =
= foldr (flip ||) (foldr (flip ||) a [x]) (reverse xs) =
=(flip ||) x (foldr (flip ||) (foldr (flip ||) a [x]) (reverse xs) =
= (flip ||) x (foldr (flip ||) a ([x] ++ (reverse xs))) =
= foldr (flip ||) a ((reverse xs) ++ [x])
```

Clearly the LHS and the RHS are equivalent

$\therefore$ By substituting `x:xs` with `xs`, `foldr || a xs = foldl ||' a (reverse xs)`.

$\blacksquare$