CS118 Lab:1

Saw Thinkar Nay Htoo

March 15, 2016

Change Log
All changes to this lab will be documented in this section.

[0.3.0] - 2016-03-04
Added
- Detailed explanation of the algorithm.
- Detailed explanation of the SML code.
- Detailed explanation of the C code.
- Detailed explanation of the ASM code.

Changed
- labasm file was not compiled due to the usage of "add" error. Now corrected to `add %eax,%ebx`.

[0.3.0] - 2016-03-08
Added
- C to ASM translation of code to illustrate the concept
- 3 functions called

Choosing a car

An algorithm is described to help select the best car based on overall cost related to fuel efficiency.

Inputs : purchase price (in $)and fuel efficiency (in mpg)

Process :

anuual fuel cost = price per gallon times the annual fuel consumed
operating cost = years times annual fuel cost
total cost = purchase price + operating cost

Output : total cost

Analysis

Assume the number of years to use this car be 10.
And let the milage of the car be 15,000 miles per year and gas cost be $3 per gallon.
Purchase price of car is $25,000 and fuel efficiency is 45 miles per gallon.
So by using above information, the total cost is calculated which is the sum of the purchase price and operating cost of the car. [1]

Detailed explanation of the algorithm to calculate the total cost.

Inputs: : purchase price (in $) and fuel efficiency (in mpg)

Process: :

annual fuel cost = 3 times 15000/45

In the above equation, 15,000 miles per year is divided by 45 miles per gallon to give the result of annual fuel usage of 333.33 gallons per year. The annual fuel usage is multiplied with $3 per gallon to give the result of annual fuel cost which is $1000.

operating cost = 10 times annual fuel cost

In the above equation, annual fuel cost of $1000 times 10 years gives the results of the operating cost of $10,000 of the car for 10 years.

total cost = 25000 + operating cost

In the above equation, the calculated operating cost is added with purchasing price of the car which is $25,000 to give the final result of the total cost; $35,000.

Output: total cost

[1] This is the algorithm to formulate the calculation of the total cost.

SML code for the algorithm

Detailed explanation of the SML code.
In this section, I try to break down the SML code into several steps making the algorithm as simple as possible for further coding with different programming languages.

SML

```
fun annual_fuel_consumed(ppg,mpy,mpg) = ppg*mpy div mpg;
val ppg = 3; val mpy = 15000; val mpg = 45;
val fuel = annual_fuel_consumed(ppg,mpy,mpg);
```

-Function "annual_fuel_consumed" is created with three parameters which are ppg, price per gallon; mpy, miles per year; and mpg, miles per gallon.
-Variables ppg, mpy and mpg are declared as 3, 15000 and 45 respectively.
-The result function "annual_fuel_consumed" assigned to variable "fuel", which is 1000.
-Below is the output of the above SML code.

```
> fun annual_fuel_consumed(ppg,mpy,mpg) = ppg*mpy div mpg;
val ppg = 3; val mpy = 15000; val mpg = 45;
val fuel = annual_fuel_consumed(ppg,mpy,mpg);
val annual_fuel_consumed = fn: int * int * int -> int
> val ppg = 3: int
val mpy = 15000: int
val mpg = 45: int
> val fuel = 1000: int
```

SML

```
val years = 10;
fun operating_cost(fc) = years * fc;
val cost = operating_cost(fuel);
```

-Variable "years" is assigned as 10.
-Function "operating_cost" is created.
-Function "operating_cost" is operated using the parameter "fuel", which is calculated previously. Then the result of the function is assigned to the variable "cost" which is 10000.
-Below is the output of the above SML code.

```
> val years = 10;
fun operating_cost(fc) = years * fc;
val cost = operating_cost(fuel);
val years = 10: int
> val operating_cost = fn: int -> int
> val cost = 10000: int
```

SML

```
val pp = 25000;
fun total_cost(oc) = pp + oc;
val total = total_cost(cost);
```

-Variable "pp" is assigned as 25000.
-Function "total_cost" is created.
-The function is operated using the parameter "cost", which is calculated previously. Then the result of the function is assigned to the variable "total", which is the final result, 35000.

-Below is the output of the above SML code.

```
> val pp = 25000;
fun total_cost(oc) = pp + oc;
val total = total_cost(cost);
val pp = 25000: int
> val total_cost = fn: int -> int
> val total = 35000: int
```

Overall output from SML code

```
Poly/ML 5.5.2 Release
val annual_fuel_consumed = fn: int * int * int -> int
val ppg = 3: int
val mpy = 15000: int
val mpg = 45: int
val fuel = 1000: int
val years = 10: int
val operating_cost = fn: int -> int
val cost = 10000: int
val pp = 25000: int
val total_cost = fn: int -> int
val total = 35000: int

Total cost of car: 35000
```

C code implementation of the SML code

Detailed explanation of the C code.
-Global variables are declared which can be accessed by functions starting from third line of following code.

Text written to file lab1.c

```
#include <stdio.h>
int eax; int ebx; int ecx;
int fuel; int cost; int total;
```

-annual_fuel_consumed2 function, operating_cost2 function and total_cost2 function are declared. Variables "years" and "pp",(purchasing price), are defined as 10 and 25000 respectively.

Text appended to file lab1.c

```
//int annual_fuel_consumed(int ppg,int mpy,int mpg)
//{return ppg* mpy/mpg;}
int annual_fuel_consumed2(){return eax = eax * ebx / ecx;}
#define years 10
//int operating_cost(int fc) {return years * fc;}
void operating_cost2(){eax = years * eax;}
#define pp 25000
//int total_cost(int msgoc){return pp + oc;}
int total_cost2(){return pp + eax;}
```

-Variables "ppg", "mpy" and "mpg" are defined as 3, 15000 and 45 respectively.

Text appended to file lab1.c

```
#define ppg 3
#define mpy 15000
#define mpg 45
```

-eax, ebx and ecx are assigned with each variable mentioned in above code.

Text appended to file lab1.c

```
int main()
{
eax = ppg;
ebx = mpy;
ecx = mpg;
```

```
18              eax = ppg;
(gdb) n
19              ebx = mpy;
(gdb) n
20              ecx = mpg;
(gdb) n
22              fuel = annual_fuel_consumed2();
(gdb) p eax
$1 = 3
(gdb) p ebx
$2 = 15000
(gdb) p ecx
$3 = 45
```

-In this case, since this C code is written in the most similar syntax of ASM code, it is assumed that most recently returned value is assigned to eax.
-After the result of the variable "fuel" is produced using function "annual_fuel_consumed2", its value of 1000 is assigned to "eax".

Text appended to file lab1.c

```
//fuel = annual_fuel_consumed(ppg,mpy,mpg);
fuel = annual_fuel_consumed2();
eax = fuel;
```

```
22      fuel = annual_fuel_consumed2();
(gdb) n
23      eax = fuel;
(gdb) n
25      operating_cost2();
(gdb) p eax
$1 = 1000
(gdb) p fuel
$2 = 1000
```

-According to the function, "operating_cost2", declared in the global, the result of the above function, "annual_fuel_consumed2", which is assigned to eax will be times with years of 10, producing the output of 10000.

Text appended to file lab1.c

```
//cost = operating_cost();
operating_cost2();
//assume return value is in eax
//cost = eax;
```

```
25      operating_cost2();
(gdb) n
30      total = total_cost2();
(gdb) p eax
$1 = 10000
```

-Now as we can see the result of eax, 10000 which is calculated previously, will be added to the "pp",(purchase price of 25000 of the car), through the function "total_cost2", consequently giving out the final result of "total" as 35000. Then the result of "total" is printed out after being assigned to the function "total_cost2", followed by "return" function.

Text appended to file lab1.c

```
//total = total_cost(cost);
//eax = cost;
total = total_cost2();
printf("Total cost of car: %i\n",total);
return 0;
}
```

```
30      total = total_cost2();
(gdb) n
31      printf("Total cost of car: %i\n",total);
(gdb) n
Total cost of car: 35000
32      return 0;
(gdb) p total
$10 = 35000
```

Above C output results on the right columns are operated using GDB,the standard debugger for the GNU operating system.

The output when running the C program in the data display debugger, using "ddd" command, is:

Translation of C code to ASM code

C to ASM translation of code to illustrate the concept

- include statements (libraries like stdio)
  There is no need to include declarations of functions like we need to do in C++
  Checklist to skim through.
  Assembly Language Library by David Parket
  The university of California Riverside Standard Library for 80x86 assembly language

- declare variables
  C code, `"int fuel= 0; int cost= 0; int total = 0;"` is translated into ASM as below.

```
.data
  fuel: .long 0
  total: .long 0
  cost: .long 0
```

- define functions
  C code, `"void operating_cost2(){eax = years * eax;}"` is translated as below.

```
  operating_cost:
  #now eax is 1000
  mov $10, %ebx
  mul %ebx
  # eax is 10000
  ret
```

- symbolic constants (in C define)
  e.g. `#define x 0` in C would be `.equ x,0` in asm

```
.equ ppg,3 \\
.equ mpy,15000 \\
.equ mpg,45 \\
```

- assignment statements
  `MOV`
  `mov $10, %ebx`
  See IA32 Cheat Sheet: Left column.

- call assembly language functions

```
call annual_fuel_consumed
call operating_cost
call total_cost
```

- call C functions (printf)
  e.g. the call to printf: printf("Total cost of car: `%i\n`",total); in asm is:

```
push %eax
push $msg
call printf
add $8,%esp
```

- return from main function

```
main:
 call annual_fuel_consumed
 call operating_cost
 call total_cost
 ret \\
```

- arithmetic operations (addition, multiplication, integer division)
  Addition.
  For addition, the value is moved to the register ebx, then the value of ebx is added to eax. C code: `int total_cost2(){return pp + eax;}`

```
mov $25000, %ebx
add %ebx,%eax
```

  Multiplication.
  For multiplication, two value of numbers are assigned to each register then those two registers are operated using "mul" mnemonic. C code: `int annual_fuel_consumed2(){return eax = eax * ebx / ecx;}`

```
.equ ppg,3
.equ mpy,15000
mov $ppg,%eax
mov $mpy,%ebx
mul %ebx
```

  Integer division.
  For division, the same procedure is used. Move the value to the register then register is operated by operation instruction "div". C code: `int annual_fuel_consumed2(){return eax = eax * ebx / ecx;}`

```
.equ mpr,45
mov $mpg, %ecx
div %ecx
```

Detailed explanation of the ASM code.

Text written to file lab1.s

```
.equ ppg,3
.equ mpy,15000
.equ mpg,45
.data
  fuel: .long 0
  total: .long 0
  cost: .long 0
msg: .string "Total cost: %i\n"
```

`.equ ppg,3` in asm is the same as `#define ppg 3` in C code. As we can see, "ppg", "mpy" and "mpg" are assigned with each value. Section .data is used to declare initialized data or constants. Unfortunately, I am not sure why these three variables, fuel, total and cost are declared. "msg: .string" works as "printf".

Text appended to file lab1.s

```
.text
  annual_fuel_consumed:
  mov $ppg,%eax
  mov $mpy,%ebx
  mul %ebx
  # edx:eax = eax * ebx
```

```
37        call annual_fuel_consumed
(gdb) n
11         mov $ppg,%eax
(gdb) n
12         mov $mpy,%ebx
(gdb) n
13         mul %ebx
(gdb) p $eax
$6 = 3
(gdb) p $ebx
$7 = 15000
(gdb) n
15         mov $mpg, %ecx
(gdb) p $eax
$8 = 45000
```

Section .text contains the actual machine instructions. In above function, eax and ebx are registers. "mov" and "mul" are used to represent low-level machine instruction or operation, also known as mnemonic.
In the above code, values of "ppg" and "mpy" is moved into eax and ebx registers respectively. In this case, due to the fact that eax is a default or initial register, "ebx" is multiplied with "eax" of 15000, and result is replaced in "eax", using "mul" operator. As we can see the result on the right column, before the "mul" instruction, value of "eax" is 3. After the multiplication, "eax" becomes 45000.

Text appended to file lab1.s

```
mov $mpg, %ecx
div %ecx
# eax = edx:eax / ecx
ret
```

```
15          mov $mpg, %ecx
(gdb) n
16          div %ecx
(gdb) p $ecx
$9 = 45
(gdb) n
20          mov $10, %ebx
(gdb) p $eax
$10 = 1000
```

Now, "mpg" value of 45 is moved into "ecx". After "eax" is divided by "ecx", a new "eax" result is produced as 1000.

Text appended to file lab1.s

```
operating_cost:
#now eax is 1000
mov $10, %ebx
mul %ebx
# eax is 10000
ret
```

```
20          mov $10, %ebx
(gdb) n
21          mul %ebx
(gdb) n
24          mov $25000, %ebx
(gdb) p $ebx
$11 = 10
(gdb) p $eax
$12 = 10000
```

Value 10 is moved into "ebx". Then "ebx" is multiplied with "eax", giving out the 10000.

Text appended to file lab1.s

```
total_cost:
mov $25000, %ebx
add %ebx,%eax
# eax is 35000
ret
```

```
24          mov $25000, %ebx
(gdb) n
25          add %ebx,%eax
(gdb) n
28          push %eax
(gdb) p $ebx
$13 = 25000
(gdb) p $eax
$14 = 35000
```

Then value of 250000 is moved to "ebx", and using "add" mnemonic value of "ebx" is added to the value of "eax", which at last results "eax" as 35000.

---

3 ASM functions addded.

```
28          push %eax
(gdb) n
29          push $msg
(gdb) n
30          call printf
(gdb) n
0x080482d0 in printf@plt ()
(gdb) p $esp
$13 = (void *) 0xbffff62c
```

Text appended to file lab1.s

```
#below code is to print out
push %eax
push $msg
call printf
add $8,%esp
#below code is for return function
ret
.globl main
main:
call annual_fuel_consumed
call operating_cost
call total_cost
ret
```

I still need to figure out more with `add $8,%esp` and some of the results above. (I got it during lab2 section: to clean up the stack.) According to the book, it seems that it is used to clear the pointer register. Basically, "eax" is pushed on the stack. Then the register "esp" is used to point a specific location on the stack to print out the value of "eax". In the final section of the code, function "annual_fuel_consumed" is returned. Below is the final ASM result.

```
debian@debian:~/labs/lab1$ ./labasm
Total cost: 35000
```

Shell scripts to make processing easier

This shell script is used to make extracting and processing the source code easier. The -g option to the gcc compiler adds debugging symbols so we can refer to variables even though they are not normally stored in the object code.

Text written to file labcode.sh

```
docsml lab1.doc
gcc -Wall -g -o labc lab1.c
gcc -Wall -g -o labasm lab1.s
```

Text written to file labdoc.sh

```
doctex lab1.doc
pptexenv /home/debian/texfot.pl pdflatex lab1.tex
```

Bourne Shell

```
chmod 755 labcode.sh
chmod 755 labdoc.sh
poly < lab1.sml > result.out
./labc > cresult
```