COMP 442 / 6421 Compiler Design

Tutorial 6

Instructor:	Dr. Joey Paquet	paquet@cse.concordia.ca
TAs:	Haotao Lai	h_lai@encs.concordia.ca
	Jashanjot Singh	s_jashan@cs.concordia.ca

Content

- Important parts in code generation
- How to use MOON
- Example

Attention

There are two approaches to do the code generation:

- tag-based approach: cannot achieve all required functionalities, but it is simple
- stack-based approach: can achieve all requirement, but complicated

If you decide to achieve most of the functionalities you need to choose stack-based approach but it will require a lot of work.

Code Generation

tag-based approach

Tag-based Approach

The way to do it is straightforward and simple, for each variable you allocate a memory for it and associate it with a unique tag which is stored in the symbol table.

Next time, when you want to access this variable (in-memory location) you can just get its address by using that predefined tag in your table.

Code Generation

stack-based approach

The key of code generation → offset

Recall

- How can you know whether a variable has been declared or not when you try to use it ?
- How many column you have in your symbol table ? What do they use for ?

Offset

- It represent how far a variable away from a base address;
- For example, a member variable of a class, offset of the variable means how far this variable’s first address away from the first address of the class;

In order to achieve code generation:

- Add a new column to your symbol table → offset
- Calculate the offset of each data type when you add that entry into your table

Offset Example → the forth column

```
class MyClass {
    int x[3][8];
    int addNum() {
        int x;
    };
};

program {
    int x;
    int y;
    MyClass myClass[4][5];
    MyClass myClass1;

};
```

Table Name: MyClass table, Parent Table Name: global table

name	kind	type	offset	link
addNum	Function	Int	96	addNum table
x	Variable	Int[3][8]	0	null

Table Name: program table, Parent Table Name: global table

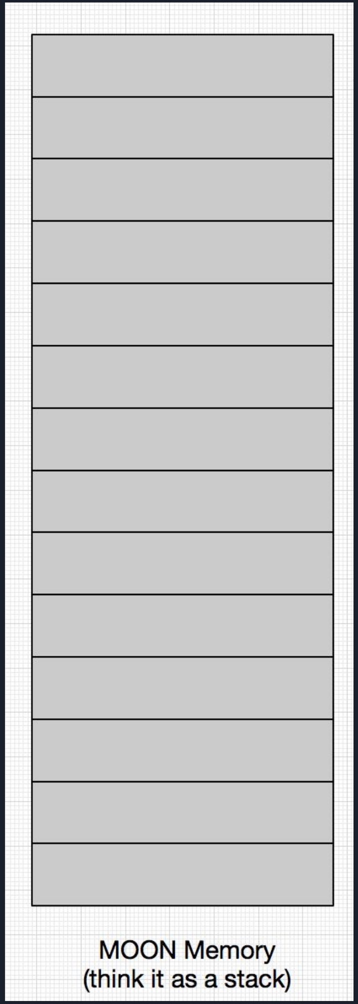
name	kind	type	offset	link
myClass1	Variable	MyClass	1928	MyClass
x	Variable	Int	0	null
myClass	Variable	MyClass[4][5]	8	null
y	Variable	Int	4	null

Stack Mechanism

What is the stack?

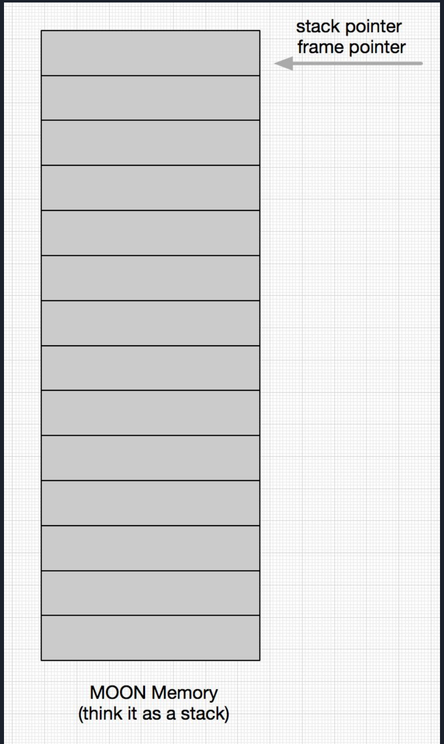
The stack we talk about here is not the real “data structure stack”. It is a function invocation stack. When a function being called, its frame will be pushed into the stack and when the function return the corresponding frame will be popped out.

In our case, we treat the MOON’s memory as a stack.



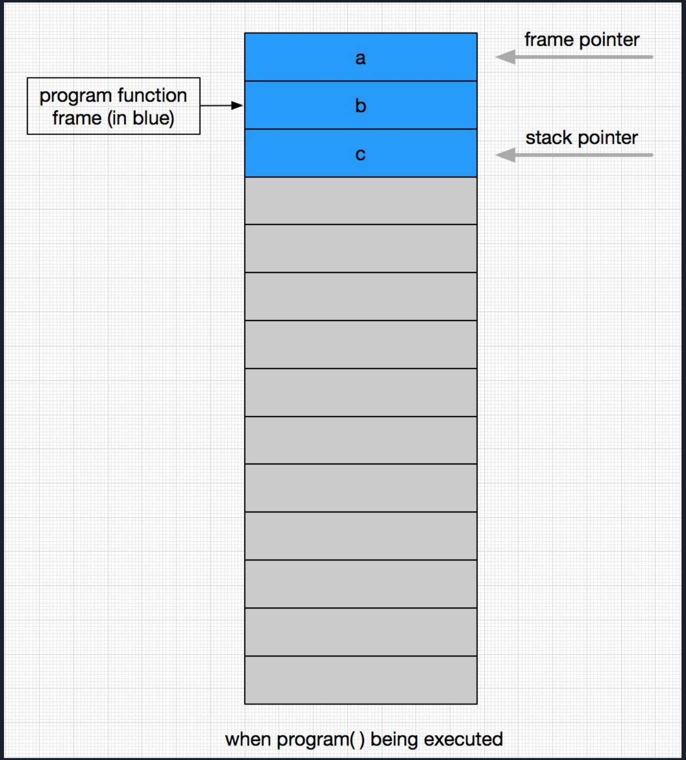
Stack-based Function Call Mechanism

```
int add(int a, int b) {
    return a + b;
}

program {
    int a;
    int b;
    int c;
    a = 1;
    b = 2;
    c = add(a, b);
    put c;
}
```

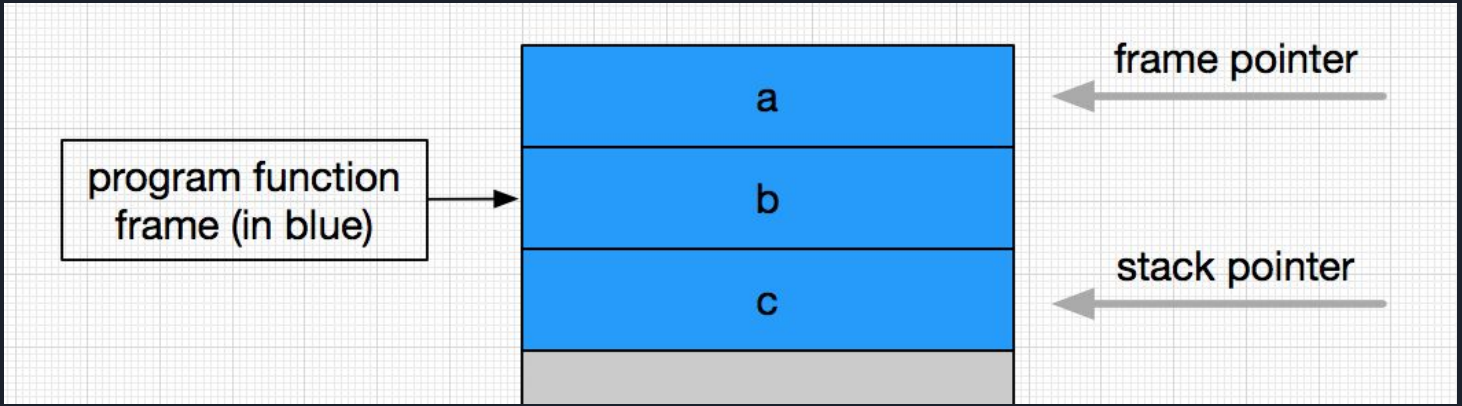


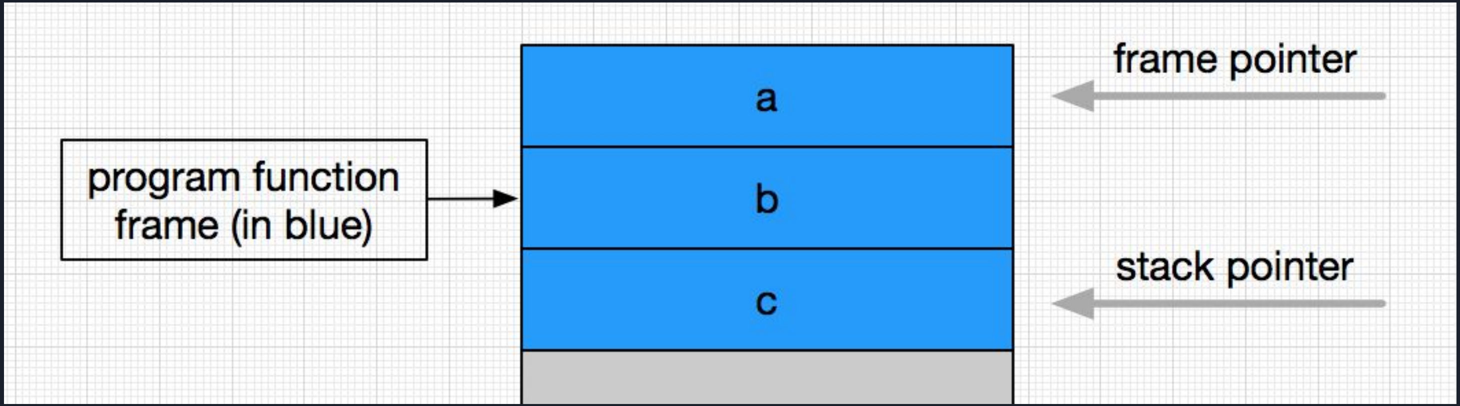
Stack-based Function Call Mechanism

```
int add(int a, int b) {
    return a + b;
}

program {
    int a;
    int b;
    int c;
    a = 1;
    b = 2;
    c = add(a, b);
    put c;
}

```



How you can know where you should put a, b, c and how to locate them?





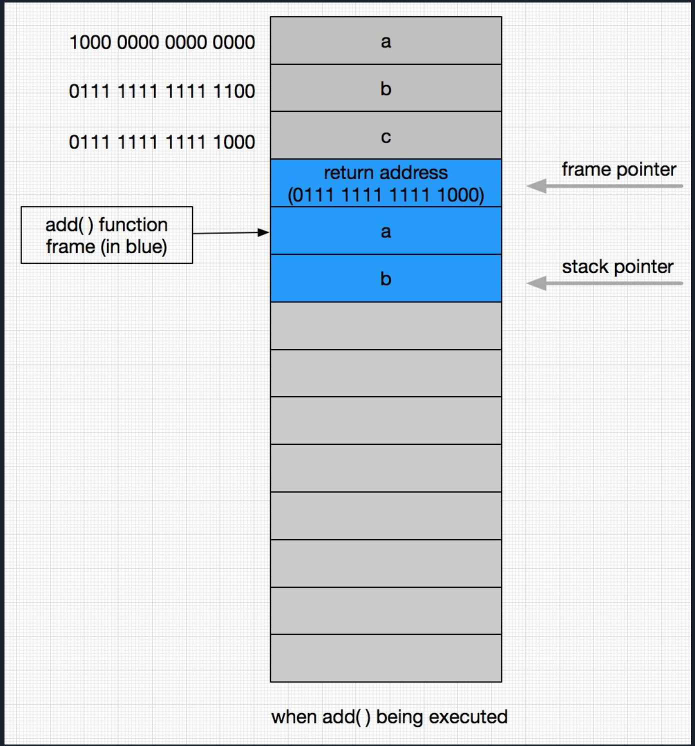
Remember we have offset!

offset → the distance from the variable cell to the frame pointer (current function's base address).

stack pointer → where the new function frame should be put.

Stack-based Function Call Mechanism

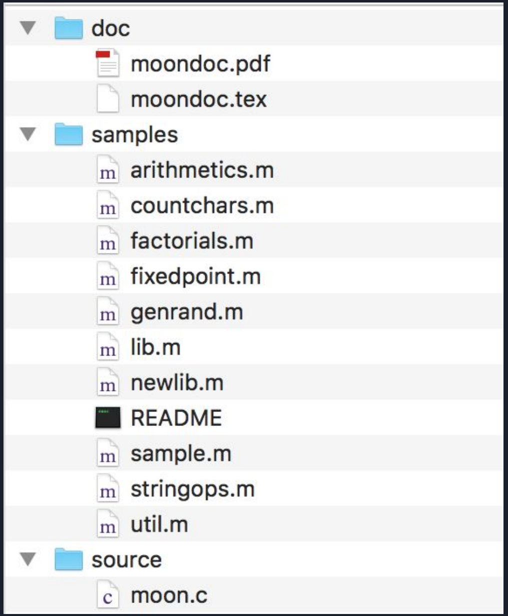
```
int add(int a, int b) {
    return a + b;
}

program {
    int a;
    int b;
    int c;
    a = 1;
    b = 2;
    c = add(a, b);
    put c;
}

```



MOON Processor

Background

- The MOON processor is wrote by Dr. Peter Grogono, the last modification is on 30 January 1995;
- It is a kind of “virtual machine” we used to run our generated code (assembly language)
- You can get the source code of Moon in the bottom of the course website
- You need to have the very basic idea of assembly language



How to compile MOON?

1. You need to have a C compiler (eg. gcc)
2. Download the source code and unzip it
3. Open Terminal, change your working directory to where you put the source code
4. Compile it using the very basic compile command

For example, if you are using **gcc**, just type the following command in the terminal:

```
gcc [-o executable_file_name] moon.c
```

If you don't specify the name, the executable will be named "a" in Unix, Linux or macOS .

Note: there is a PDF file accompanying with the source code, you are strongly suggested to read that file before you ask any question.

Important Parameters of MOON

- All instructions of MOON occupy one word
- There are total 16 registers from R0 to R15, R0 always contains zero
- Program counter is 32-bit and contains the address of next instruction to be executed
- Memory address in the range of [0, 2^31], the usable memory is less than that

How to use MOON?

There are 4 types of instruction:

1. Data access instructions
2. Arithmetic instructions
3. Input and output instructions
4. Control instructions

Terminology

- M_8[K]: it denotes the **byte** stored at address K;
- M_{32}[K]: it denotes the **word** stored at address K, K + 1, K + 2 and K + 3;
- An address is **aligned** if it is a multiple of 4;
- An address is **legal** if the address byte exists;
- The name PC denotes the **program counter**;
- The name R0, R1, … denotes the **registers**;
- The symbol ← denotes data transfer;

Note: the slide cannot show all instructions provided by MOON, please consult the documentation for more detailed !

Data Access Instructions

	Function	Operation	Effect
must aligned	Load word	lw $Ri, K(Rj)$	$\mathcal{R}(i) \overset{32}{\longleftarrow} \mathcal{M}_{32}[\mathcal{R}(j) + K]$
	Load byte	lb $Ri, K(Rj)$	$\mathcal{R}_{24..31}(i) \overset{8}{\longleftarrow} \mathcal{M}_{8}[\mathcal{R}(j) + K]$
must aligned	Store word	sw $K(Rj), Ri$	$\mathcal{M}_{32}[\mathcal{R}(j) + K] \overset{32}{\longleftarrow} \mathcal{R}(i)$
	Store byte	sb $K(Rj), Ri$	$\mathcal{M}_{8}[\mathcal{R}(j) + K] \overset{8}{\longleftarrow} \mathcal{R}_{24..31}(i)$

Take load word as an example:

$R(i) \overset{32}{\longleftarrow} M_{32}[R(j) + K]$ means take one word data stored in the address (R(j) + K) and put it into register R(i)

where K in the range of [-16384, 16384)

Arithmetic Instructions

There are two types of arithmetic instructions:

1. R (i) ← R (j) + R (k), sum up the second and third register's value and put the result into the first register;
2. R (i) ← R (j) + k, sum up the second register's value and the third value then put the result into the first register;

We call all productions like the second one shown above "instruction with immediate operand".

Arithmetic Instructions

Function	Operation		Effect
Add	add	Ri, Rj, Rk	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) + \mathcal{R}(k)$
Subtract	sub	Ri, Rj, Rk	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) - \mathcal{R}(k)$
Multiply	mul	Ri, Rj, Rk	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \times \mathcal{R}(k)$
Divide	div	Ri, Rj, Rk	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \div \mathcal{R}(k)$
Modulus	mod	Ri, Rj, Rk	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \bmod \mathcal{R}(k)$
And	and	Ri, Rj, Rk	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \wedge \mathcal{R}(k)$
Or	or	Ri, Rj, Rk	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \vee \mathcal{R}(k)$
Not	not	Ri, Rj	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \neg\mathcal{R}(j)$
Equal	ceq	Ri, Rj, Rk	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) = \mathcal{R}(k)$
Not equal	cne	Ri, Rj, Rk	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \neq \mathcal{R}(k)$
Less	clt	Ri, Rj, Rk	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) < \mathcal{R}(k)$
Less or equal	cle	Ri, Rj, Rk	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \leq \mathcal{R}(k)$
Greater	cgt	Ri, Rj, Rk	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) > \mathcal{R}(k)$
Greater or equal	cge	Ri, Rj, Rk	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \geq \mathcal{R}(k)$

Function	Operation		Effect
Add immediate	addi	Ri, Rj, K	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) + K$
Subtract immediate	subi	Ri, Rj, K	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) - K$
Multiply immediate	muli	Ri, Rj, K	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \times K$
Divide immediate	divi	Ri, Rj, K	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \div K$
Modulus immediate	modi	Ri, Rj, K	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \bmod K$
And immediate	andi	Ri, Rj, K	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \wedge K$
Or immediate	ori	Ri, Rj, K	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \vee K$
Equal immediate	ceqi	Ri, Rj, K	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) = K$
Not equal immediate	cnei	Ri, Rj, K	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \neq K$
Less immediate	clti	Ri, Rj, K	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) < K$
Less or equal immediate	clei	Ri, Rj, K	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \leq K$
Greater immediate	cgti	Ri, Rj, K	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) > K$
Greater or equal immediate	cgei	Ri, Rj, K	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(j) \geq K$
Shift left	sl	Ri, K	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(i) \ll K$
Shift right	sr	Ri, K	$\mathcal{R}(i) \stackrel{32}{\longleftarrow} \mathcal{R}(i) \gg K$

- the logical operation operate on each bit of the word
- the comparison operator store result either “1” (true) or “0” (false)
- in the right side table, the operand K is a signed 16-bit quantity, negative numbers like -1 is interpreted as -1 not 65535

Input and Output Instructions

Function	Operation	Effect
Get character	getc Ri	$\mathcal{R}_{24..31}(i) \stackrel{8}{\longleftarrow}$ Stdin
Put character	putc Ri	Stdout $\stackrel{8}{\longleftarrow} \mathcal{R}_{24..31}(i)$

This two instructions are useful when you try to out the result of your program to show it really worked during the final demo.

Control Instructions

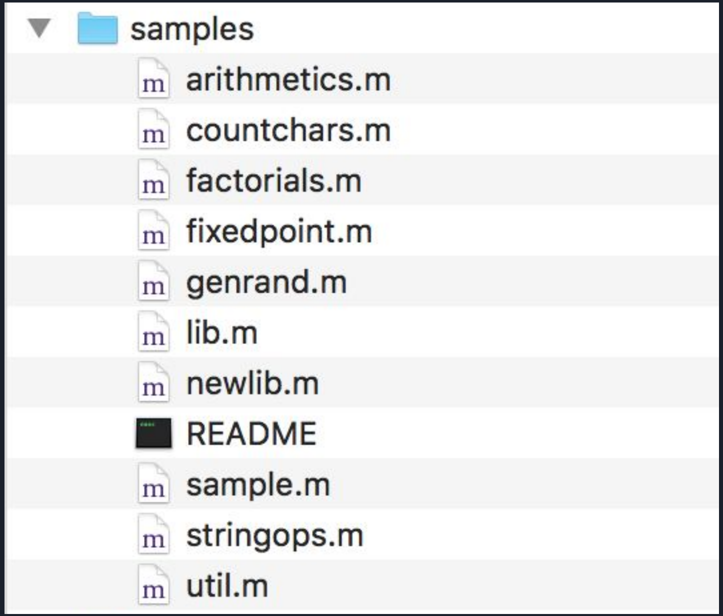
Function	Operation		Effect
Branch if zero	bz	Ri, K	if $\mathcal{R}(i) = 0$ then $PC \overset{16}{\longleftarrow} PC + K$
Branch if non-zero	bnz	Ri, K	if $\mathcal{R}(i) \neq 0$ then $PC \overset{16}{\longleftarrow} PC + K$
Jump	j	K	$PC \overset{16}{\longleftarrow} PC + K$
Jump (register)	jr	Ri	$PC \overset{32}{\longleftarrow} \mathcal{R}(i)$
Jump and link	jl	Ri, K	$\mathcal{R}(i) \overset{32}{\longleftarrow} PC + 4; PC \overset{16}{\longleftarrow} PC + K$
Jump and link (register)	jlr	Ri, Rj	$\mathcal{R}(i) \overset{32}{\longleftarrow} PC + 4; PC \overset{16}{\longleftarrow} \mathcal{R}(j)$
No-op	nop		Do nothing
Halt	hlt		Halt the processor

- when you use branch, remember to set the PC (program counter) correctly
- jump instruction will be useful when you generate function code, you need to store the return address properly

MOON Example

Refer to the sample folder

the most simple one is the sample.m, I strongly recommend you begin with this example in order to get familiar with MOON.



sample.m

```
        org   103
message db    "Hello, world!", 13, 10, 0
        org   217
        align
        entry                   % Start here
        add   r2,r0,r0
pri     lb    r3,message(r2)    % Get next char
        ceqi  r4,r3,0
        bnz    r4,pr2           % Finished if zero
        putc   r3
        addi   r2,r2,1
        j      pri              % Go for next char
pr2     addi   r2,r0,name       % Go and get reply
        jl     r15,getname
        hlt                     % All done!
```

sample.m

```
% Subroutine to read a string
name     res     59                  % Name buffer
         align
getname  getc    r3                  % Read from keyboard
         ceqi    r4,r3,10
         bnz     r4,endget           % Finished if CR
         sb      0(r2),r3            % Store char in buffer
         addi    r2,r2,1
         j       getname
endget   sb      0(r2),r0            % Store terminator
         jr      r15                 % Return

data     dw      1000, -35
```

Final Example

source code → assembly code

```
program {
    int x;
    int y;
    int z;
    x = 2;
    y = 34;
    z = x + y * x;
    put (z);
};
```

```
entry   % ======program entry======
align   % following instruction align
addi    R1, R0, topaddr % initialize the stack pointer
addi    R2, R0, topaddr % initialize the frame pointer
subi    R1, R1, 12  % set the stack pointer to the top position of the stack
addi    R14, R0, 2  %
sw  -12(R2), R14    %
addi    R8, R0, 34  %
sw  -8(R2), R8  %
lw  R6, -12(R2) %
lw  R9, -8(R2)  %
lw  R11, -12(R2)    %
mul R9, R9, R11 %
add R6, R6, R9  %
sw  -4(R2), R6  %
lw  R10, -4(R2) %
putc    R10 %
hlt % ======end of program======
```

```
ERIC_LAI  ~/Downloads/moon  ./moon ../OnlyProgram.m
Loading ../OnlyProgram.m.
F
221 cycles.
```

2+2*34=70 —> ascii code F

```
program {
    int x;
    x = 65;
    if (x == 1) then {
        x = 65;
    } else {
        x = 66;
    };
    put (x);
};
```

```
        entry   % ======program entry======
        align   % following instruction align
        addi    R1, R0, topaddr % initialize the stack pointer
        addi    R2, R0, topaddr % initialize the frame pointer
        subi    R1, R1, 4   % set the stack pointer to the top position of the stack
        addi    R14, R0, 65 %
        sw  -4(R2), R14 %
        lw  R8, -4(R2)  %
        ceqi    R8, R8, 1   %
        bz  R8, else_1  % if statement
        addi    R6, R0, 65  %
        sw  -4(R2), R6  %
        j   endif_1 % jump out of the else block
    else_1  addi    R9, R0, 66  %
        sw  -4(R2), R9  %
    endif_1 nop % end of the if statement
        lw  R11, -4(R2) %
        putc    R11 %
        hlt % ======end of program======
```

```
ERIC_LAI  ~/Downloads/moon  ./moon ../IfStatement.m
Loading ../IfStatement.m.
B
162 cycles.
```

Thanks!

Good Luck for your project . . .