Compiler Report

Jonathan Zheng

This is my report on the compiler I made to translate c into ARM assembly. The bulk of the report will be walkthroughs of example code to show my design decisions and the compiler's behaviour, including an experiment that involves computing pi at the end. This introduction will be an overview of my grammar and how my code is organised.

My grammar can be viewed broadly as:
program->functions->instructions->declarations/assignments/function calls/statements

All the grammar rules use left recursion, as this is the suggested direction on the bison web guide. I also made the decision not to have any grammar rules that can lead to empty. This was to reduce the chance of shift reduce conflicts while writing the grammar, as well as making the whole thing easier to debug and follow through. If you read through it you'll notice that I tried to separate all recursive sections into 'symbol_list : symbol_list symbol | symbol' so as to be able to pinpoint where the grammar can parse out of control.

The parser's code is all contained within the 'Parser' class, where functions such as lexer.YYText() and lexer.lineno() can be called. It also means that variables can work kind of like global variables to store information to help with printing out the ARM code. It can be viewed into a number of sections inside ('StoringInformation.cpp', 'ArmInstructions.cpp', 'ConvertingToArm.cpp') :

Storing information

stack track	keeps track of the stack in the ARM code
scope	keeps track of the variables in the c code
types	keeps track of types in the c code
storing functions	keeps track of any declared functions

Converting to ARM

arm instructions	converts ARM instructions into strings (in a separate file due to its large size
arm memory	creates the labels and directives that go at the end of the ARM code
calling functions	creates the group of instructions to call functions with
function code	creates the group of instructions to start and end a function definition
operators	converts operators in c into their ARM equivalent
assignments	converts the operator '=' in c into the ARM equivalent

If viewing the code on gedit, a tab width of 4 should be used so that the lines of code line up properly. The ARM code is written by a filestream into a file called "test.s", so that 'cout' can be used separately to return errors. Any error messages will be printed directly onto the command line output.

The example programs in the following pages are in the following order:
Basic compiler structure, Loops, Functions, Arrays & Pointers, Error Checking, Computing Pi

Simple program

```c
#include <stdio.h>

int main(){
    int a = 1, b = a + 7;

    printf("%d\n", b);
}
```

```
    .text
.global main

main:
    PUSH {r4-r12, lr}

    MOV r4, #1
    LDR r12, addr_a1
    STR r4, [r12]
    ADD r11, r4, #7
    MOV r5, r11
    LDR r11, addr_b1
    STR r5, [r11]

    LDR r0, print_args1
    MOV r1, r5
    BL printf
    POP {r4-r12, pc}

print_args1: .word print_args_str1
addr_a1: .word a1
addr_b1: .word b1

    .data
print_args_str1: .asciz "%d\n"
a1: .skip 4
b1: .skip 4
```

I think that the best way to explain how my compiler works is to start off by walking step by step through a simple program.

The parser starts off by looking for a function, by looking for an ID (regex [a-zA-Z_]+) as the return type, followed by another ID, as the function name. It then looks for a pair of round brackets filled with pairs of ID's separated by commas to act as parameter declarations for the function (in this case empty).

The compiler then prints the name of the function as a label (in this case "main") and pushes registers r4 to r12 onto the stack. Every time it sees a PUSH instruction, a structure called "**stack_track**" is created that contains an array of 13 booleans (one for each register r0-r12 that we are allowed to change) and a pointer, "stack_above", that points to another stack_track structure, making it a linked list. This is so that when it sees a POP instruction, the compiler can just 'hop' back up the chain of stacks to keep track of which registers are free.

When it sees a curly bracket that signals the beginning of the function code, a structure called "**scope**" is created that contains a map containing information on any variables declared, along with a pointer, "scope_above" to another scope structure, making it also a linked list. This was created for the same reason as the other linked list, but keeps track of the state of the c code, whereas stack_track is used to keep track of the state of the ARM code. A more detailed explanation of these linked lists and how they function will be placed later on in this report.

After a function is seen, the grammar looks for instructions, the first of which in this program is a declaration. It looks for an ID (as the type) then looks for either another ID (as the variable name) or an assignment (the variable name along with its value), separated by commas.

For each variable declared, the compiler stores the type, register stored in, array size, and whether or not it's a pointer (in a structure called "variable_info"). When it sees a variable's name, it allocates a register by going through the current stack_track, and looking for a free register starting from r4 (because r0-r3 are kept free for function calls), setting the boolean to true and recording the register used.

If there are square brackets after the variable name, the array size (int array_size) is set to the number inside the brackets, 0 otherwise to signal not an array. If there's an asterisk before the name, whether it's a pointer (bool pointer) is set to true, false otherwise. All of this information is stored inside a map that uses the variable's name as a key, and variable_info as the data. This allows for quick error checking, as all the information on a variable can be gotten by using its name as the index of the map e.g. checking if the name is reused, or whether it can be dereferenced.

With the register for a variable allocated (in this case a is in r4, b is in r5), the assignments can be created. It looks at the left hand side and creates a MOV into the variable's register, and the expression on the right hand side is evaluated into a temporary register unless it's a literal value (in this case a = 1 doesn't need one).

For b = a + 7, the compiler sees the '+' operator in the right hand side, and for every operator the result is stored into a temporary. This creates ADD r11, r4, #7 (where r11 is the temporary) and then r11 is MOVed into r5. This can be optimised into ADDing straight into r5, but because my compiler evaluates one operator at a time, it cannot do this immediately.

At the end of each assignment, the code then has to STR the value into the variable's location in memory. To do this, the address (addr_+*variable name*) is LDRed into a temporary register to get the location in the .data section of the ARM code. The variable's register is then STRed using the temporary's register surrounded by square brackets, storing the value into its location in memory. This leads to a MOV, LDR, STR pattern at the end of each assignment.

Because the same variable name can be used inside different scopes in c, they have to be differentiated somehow inside the ARM memory section. To do this, my compiler assigns a number to each scope node in the linked list (int scope_no) and affixes this number to the end of the variable's labels at the end of the program.

After each declaration instruction, the labels for each variable are created. The size, in bytes, of each type are stored inside another map, and this is used to decide the value used after .skip for each label.

Phew, that's one instruction done. The other instruction inside this sample program is printf, a function call. Luckily, ARM has its own implementation of printf that uses registers r0-r3, so it's not too much work. The string is created by using .asciz "*text*" inside the .data section, which makes the label "print_args_str" a pointer to a constant string allocated somewhere in memory (the same thing that c does when it sees two inverted commas). To implement multiple printfs, the compiler affixes an incrementing int to the label each time.

Procedure calls in ARM have to abide by a few rules. The parameters are stored (in order or appearance in the call parameters) from r0-r3, and the return value is placed in r0 after the procedure is run. That means that, to call printf, the string goes into r0, and the parameters used in

the string e.g. %d are placed into r1-r3.[1] In this case the register allocated to b, r5, is MOVed into r1 and the pointer to the string is placed into r0. All that's left is to BL (branch and link) to the label printf and the function gets called. The values inside r0-r3 aren't guaranteed to be preserved so that's why variables shouldn't be stored there.

When the closing curly bracket is seen, the program hops back up the scope linked list to the scope above, deleting the map inside that node of the list. This method is also nice because c allows you to access variables inside any scope above the current one, so if it doesn't find a variable's name inside the current variable map, it just needs to hop to the next scope up and look there. This also means that it'll find the variable name in the closest scope to the current, making using the right variables come together nicely.

[1]This means that printf cannot take more than 4 arguments. To fix this, the printf in c would have to be split into multiple calls of the printf in ARM. I decided not to do this in the end but it is a possible improvement.

Loops

test.c test.s

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    char a = 'a';
    int i = 0;

    do{
        printf("%c", a + i);
        i++;
    }
    while(i < 26);

    printf("\n");
}
```

```
Save modified buffer (ANSWERING "No" WI

pi@raspberrypi:~$ gcc test.s -o test
pi@raspberrypi:~$ ./test
abcdefghijklmnopqrstuvwxyz
pi@raspberrypi:~$
```

test.c *test.s

```
    .text
.global main

main:
    PUSH {r4-r12, lr}
    LDR r12, addr_char1
    LDR r12, [r12]
    MOV r4, r12
    LDR r11, addr_a1
    STR r4, [r11]
    MOV r5, #0
    LDR r12, addr_i1
    STR r5, [r12]
label1:
    ADD r12, r4, r5
    LDR r0, print_args1
    MOV r1, r12
    BL printf
    ADD r5, r5, #1
    LDR r12, addr_i1
    STR r5, [r12]
    CMP r5, #26
    MOVLT r12, #1
    MOVGE r12, #0
    CMP r12, #0
    BNE label1
    LDR r0, print_args2
    BL printf
    POP {r4-r12, pc}

addr_char1: .word char1
print_args1: .word print_args_str1
print_args2: .word print_args_str2
addr_a1: .word a1
addr_i1: .word i1

    .data
char1: .byte 'a'
print_args_str1: .asciz "%c"
print_args_str2: .asciz "\n"
a1: .skip 1
i1: .skip 4
```

This c program is designed to show how loops and chars work inside my compiler. Luckily, chars can be operated on like numbers so incrementing one in a for loop can show successive ASCI values. Loops and ifs are put into a non-terminal called "**statement**" that can be derived from "instruction". In the lex file, "if", "for", "while" etc. are terminals, so as to make sure that those IDs are reserved.

A dowhile loop creates the least ARM instructions, so I chose to use it for this example. The compiler finds "do" and then finds an opening curly bracket, after which any other list of instructions can exist before the closing bracket. Since statements are a part of "instruction" this allows for nested loops.[2] After a list of one or more instructions, it then sees the word "while", and looks for (*expression*), where *expression* is evaluated and compared with 0. The comparison operators (e.g. '==', '>', '<=') are put together with the other mathematical operators, like plus and minus, so they actually

[2] Unfortunately I couldn't get ifelse to work recursively this way without shift/reduce conflicts, so "else" isn't supported by my compiler.

evaluate into a value (in this case true evaluates to 1 and false to 0). This allows all of these operators to be used together at the same time, as in c.[3] Unfortunately, due the nature of my grammar, allowing assignments inside these brackets would have been quite messy so my compiler doesn't allow it.

The ARM instructions that implement this *expression* start from "CMP r5, #26". The next two MOVs are conditional according to the operator used (in this case '<' evaluates to 1 if true using the 'LT' condition code). The second CMP compares the result to 0, and branches if not equal, causing the loop to branch until the condition is met. The labels branched to are disambiguated by again affixing an incrementing int to each successive one allocated.

This program also shows how chars and multiple printfs work nicely. When a character (regex '(.|"\n"|"\t")', giving any single character inside two inverted commas) needs to be assigned to a char, the literal value can't just be MOVed into the register, as no ARM instruction supports this. This means that it has to be converted into a loadable form using the .byte directive (.byte 'a' in the sample code). This can then be loaded directly into a register using two LDRs (the first two instructions after the PUSH in main).

Multiple printfs work in a similar way. Every time the function is called, the string in the first argument is converted using the .asciz directive.[4] All of these label types have an incrementing int affixed to the end of them. This could have been replaced with just one incrementing int and a generic name for all labels (to make the compiler more efficient), but this way makes the code a lot more readable (which helps especially for debugging).

[3] In hindsight, the assignment operator should also have been grouped together with the other operators using '=', along with things like commas and square brackets as well. This would have encapsulated anything to do with operators in the grammar and made it a lot more organised. It was too late to change after I realised this but if I were to start again I would make sure to do it.

[4] I would have done this for all strings including those outside of printf, but I decided to spend my time on other aspects of the language.

Functions

*test.c

```c
#include <stdio.h>
#include <stdlib.h>

int myFactorial(int a);

int main(){
    int a = 10;
    a = myFactorial(a);

    printf("%d\n", a);
}

int myFactorial(int a){
    if(a == 0) return 1;

    return myFactorial(a-1) * a;
}
```

```
pi@raspberrypi:~$ gcc test.s -o test
pi@raspberrypi:~$ ./test
3628800
pi@raspberrypi:~$
```

*test.c test.s

```
    .text
.global main

main:
    PUSH {r4-r12, lr}
    MOV r4, #10
    LDR r12, addr_a1
    STR r4, [r12]
    MOV r0, r4
    BL myFactorial
    MOV r12, r0
    MOV r4, r12
    LDR r11, addr_a1
    STR r4, [r11]
    LDR r0, print_args1
    MOV r1, r4
    BL printf
    POP {r4-r12, pc}

myFactorial:
    PUSH {r4-r12, lr}
    MOV r4, r0
    LDR r12, addr_a2
    STR r4, [r12]
    CMP r4, #0
    MOVEQ r12, #1
    MOVNE r12, #0
    CMP r12, #0
    BEQ label1
    MOV r0, #1
    POP {r4-r12, pc}
label1:
    SUB r12, r4, #1
    MOV r0, r12
    BL myFactorial
    MOV r12, r0
    MUL r11, r12, r4
    MOV r0, r11
    POP {r4-r12, pc}
    POP {r4-r12, pc}
```

The next port of call is functions, and how better to show how my compiler deals with functions than a horribly inefficient implementation of a factorial function. The first thing the compiler sees is a function prototype, which actually contains enough information to branch to the corresponding procedure in arm. All the compiler needs is the name and how many parameters a function has to branch to it.

When the compiler sees a function call, it puts the parameters, in order of appearance, from registers r0 and up. The above function only has 1 parameter, so it moves r4 (variable a's register) into r0. Having everything inside the same registers before every function call means that the function doesn't have to worry about where variables are; it just looks at the order in which they appear in the definition. Furthermore, the return value of a function is always located in register 0, which is why, after the "BL myFactorial" instruction, r0 is put into r12 (a temporary) then moved into r4 (variable a).

If a function has more than 4 parameters, it can't just shove all the values into r0 and up, because once you try to use r4+ you'll end up overwriting the registers used for variables. To fix this, I push any registers needed past r4 onto the stack before the function gets called, and then put the variables into the registers just pushed. This can also cause data hazards, but this is fixed by moving them all to temporaries before putting them into the correct registers for the function call.[5]

The function code itself always begins by setting up the state of the arm machine correctly by placing all passed parameters into the correct registers and storing them in the correct place in memory. The procedure call starts with a PUSH instruction to keep the program from overwriting the state of the procedure called from. Then, for each parameter, there'll be a MOV, LDR, STR sequence to move the variable from r0+ to r4+ (in case of recursion, like in the example), then to load the address of the procedure's allocated memory space and store it there.

The rest of the code should be translatable as if it were anywhere else. The if statement contains the base case, in this case returning 1 by placing #1 in r0 and executing a POP, thereby not reaching the branch and link that is further down in the function. If the base case isn't reached, the passed parameter is decremented and placed in r0 just like it is originally done in main. Each time the function is returned, it's multiplied by the non-decremented variable and placed in r0 before the POP is executed.

[5] Nested function calls e.g. myFunction(a, myFunction(a, b)) are not supported by my compiler. The problem is that I use the same variable to store which parameters are seen by a function call, so the parameters in the nested call end up overwriting the ones in the outer call. I could fix this by keeping track of each function call's parameters separately, but decided to spend time on other features instead.

Pointers & Arrays

Compiler.y | Parser.cpp

```
#include <stdio.h>
#include <stdlib.h>

typedef int *int_pointer;

int main(){
    int_pointer a[10];

    int i;
    for(i = 0; i < 10; i++){
        a[i] = malloc(sizeof(int));
        *a[i] = i * 2;
        printf("%d ", *a[i]);
    }
}
```

```
pi@raspberrypi:~$ gcc test.s -o test
pi@raspberrypi:~$ ./test
0 2 4 6 8 10 12 14 16 18 pi@raspberrypi:~$
```

```
    .text
.global main

main:
    PUSH {r4-r12, lr}
    LDR r4, addr_a1
    MOV r5, #0
    LDR r12, addr_i1
    STR r5, [r12]
label1:
    MOV r0, #4
    BL malloc
    MOV r12, r0
    STR r12, [r4, r5, LSL #2]

    MOV r11, #2
    MUL r12, r5, r11
    LDR r11, [r4, r5, LSL #2]
    STR r12, [r11]

    LDR r12, [r4, r5, LSL #2]
    LDR r12, [r12]
    LDR r0, print_args1
    MOV r1, r12
    BL printf

    ADD r5, r5, #1
    LDR r11, addr_i1
    STR r5, [r11]
    CMP r5, #10
    MOVLT r12, #1
    MOVGE r12, #0
    CMP r12, #0
    BNE label1
    POP {r4-r12, pc}

print_args1: .word print_args_str1
addr_a1: .word a1
addr_i1: .word i1

    .data
print_args_str1: .asciz "%d "
.balign 4
a1: .skip 40
i1: .skip 4
```

This last sample program is designed to show how my compiler deals with arrays and pointers. The program doesn't do much; it just calculates a sequence of even numbers. However, the even numbers get stored in memory through an array of pointers, which showcases how my program deals with this sort of stuff well.

The first new thing in this example is typedef. I already had a system in place for type checking that could deal with typedef quite well. Another map, "**type_map**" is used, in the same vein as variables and functions, to store information on defined types that contains the type's name, size (in bytes), and whether it's a pointer. If it's a pointer, the size is automatically set to 4, as addresses in ARM are that long. When a variable is declared using a type that is a pointer, the 'pointer' field inside the variable map is also set to true, and it can be used just as if it were declared as 'int *a'.

One thing to note about my compiler is that only one asterisk before a variable is supported i.e. pointers to pointers aren't allowed so 'int **a' would give a syntax error. I decided not to add a recursive component to the '*' operator, because I figured that one would be difficult enough. The same thing applies to arrays, only one-dimensional arrays are supported by the compiler. If I were to start again, I would also place these with the other mathematical operators like '+' or '-'. Because of this, using the above program as an example, I made my compiler give a non-syntax error if 'int_pointer *a' is seen, as it's equivalent to 'int **a'.

When a pointer is declared, it acts just like any other 4-byte variable inside ARM. The difference is that it can be dereferenced in c. When it's used on the right hand side of an assignment and dereferenced, instead of using, for example, 'MOV r4, r5', it needs to be 'LDR r4, [r5]'. [6]

When an array is declared, the difference is that more memory has to be allocated than the amount for a normal variable. This can be seen at the bottom of the sample ARM code under the .data section. Before the label for the array, a1, the directive .balign 4 is used to tell the processor how long a word is (int this case 4 for a pointer). The number after .skip is also now 40, because a total of 40 bytes are needed for 10 4 byte values. The address of the first value is then LDRed into the register allocated for the array, in this case 'LDR r4, addr_a1'.

The next 3 instructions are just for initialising i to 0 using the MOV, LDR, STR pattern to start the for loop. Malloc is actually quite easy to implement in arm, because there's a native function that does it for you. All you have to do is pass the number of bytes you want to allocate and it returns a pointer to that memory in r0. You can see that the next three instructions place #4 into r0 and the returned pointer into r12.[7]

The next instruction is quite interesting. Because 'a' is an array, it's assigned using STR instead of MOV. The value inside the square brackets for the STR is then created as follows. The first register, r4, contains the address of the first value in the array. Because the index is another variable, 'i', that variable's register gets used as an offset by placing it after r4 to give '[r4, r5'. This isn't enough though, because the size of an integer needs to be added to the offset each time, so r5 gets 'LSL #2' because log2(sizeof(int)) equals 2. This all gives '[r4, r5, LSL #2]', which is equivalent to a[i] in c.

The next 4 instructions deal with *a[i] = i * 2. The MOV and MUL instructions place the right hand side of the assignment into r12, the LDR and STR get the address that needs to be changed in the same way the index was created before, except this time it's used to store into memory.

The rest of the ARM code works just like previous examples. The next 5 instructions deal with printf and the rest of it is just moving things around to implement the for loop properly.

---

[6] Assignments are more complicated when a dereference is used on the left hand side. The problem is that if you change a variable at a memory location, you also have to change the value inside the variable's register, but by the time you're assigning it, the information on what variable needs to be changed has been forgotten. This means that another data structure needs to be in place to keep track of what pointers are pointing to what variables.
To do this, yet another map, "**pointer_map**" is used to keep track of information in the c code, which maps registers that contain pointers, to the registers that need to be changed if the dereferenced pointer is assigned. This gets quite messy for arrays of pointers, which requires the index to be stored too.

[7] Sizeof is calculated at compile time by placing it under the same grammar rule that numbers are, and just passing the evaluation as if it were a regular number up the parse tree.

ERROR CHECKING

```
#include <mistake.h>

int main(){
    itn index;
    myFunction(variable);
    int a;
    *a = 4;
    a[34] = b;
    printf("hello\n");
}

void myFunction(notatype name){
    return 4;
}
```

This program is designed to be full of bugs to show how my compiler does error checking. The compiler saves all the errors it collects across a run of the program, and saves them into a vector of strings to print out at the end; it prints errors directly into the console so that it doesn't get mixed up with the output file.

One problem, as you can see in the console, is that sometimes the same error message gets printed multiple times. This is because the function that looks for an error (e.g. above where the compiler can't find a variable used in an expression) is sometimes needed multiple times in the same c instruction.

The other problem is that syntax errors can't really be helpful, because it's not possible to know how the grammar hasn't been followed if you don't include incorrect expansions inside the grammar, which I haven't done. It is possible to print the line number at which the parser fails, so the compiler at least tells the user where the error is, even if it can't deal with specifics.

Computing pi

```c
#include <stdio.h>

int myDiv(int top, int bottom){
  int i = 0;
  while(top >= bottom){
    top = top - bottom;
    i++;
  }
  return i;
}

int main(){
  int pi = 0;
  int million = 100 * 100 * 100;

  int i, j = 4;
  for(i = 1; i < million; i++){
    pi = pi + j*myDiv(100*million, (2*i-1));
    j = 0-j;
  }

  printf("%d\n", pi);

}
```

```
[jxz12@eews304b-022 c programs]$ gcc test.c -o test
[jxz12@eews304b-022 c programs]$ ./test
314159160
[jxz12@eews304b-022 c programs]$
```

```
pi@raspberrypi:~$ ./test
314159160
pi@raspberrypi:~$
```

I thought it would be fun to do a little experiment to compare the efficiency of ARM compiler against gcc by having a race to calculate pi. It's also quite good for showing off the limitations of my compiler.

The program implements the series 1 - 1/3 + 1/5 - 1/7… but you'll notice that the c code has a few strange features because of my compiler. The first is that the divide operator, '/' isn't supported because just using the SDIV instruction gives errors to do with processors so I decided to leave it out. Because of this, I had to implement a function by hand in the c code that does it (very inefficiently). It just subtracts the divisor until it can't subtract anymore using a while loop, and returns how many subtractions it took.

There is also no way of using a decimal point inside my compiler because doubles and floats require different treatment from integers, and I decided to concentrate on other features first. Because of this, pi will not be calculated as 3.14… but will have the first 3 as the most significant digit of the result. Effectively, everything is multiplied by a hundred million.

Another limitation is the size of literals. For example 'MOV r1, #*number*' only supports *number*s up to 255 due to the design of immediates inside ARM assembly. Because of this, to load a million, the c

code has to multiply 100*100*100 because that ends up splitting up the calculation so that the MOVs only every have to load 100 in at a time.

I also forgot to add unary operators e.g. 'j = -j', so the code has to subtract j from 0, which isn't too bad but gives an indication of just how much there is inside c.

With all of these changes made, my compiler came up with this giant piece of code

```
myDiv:
    PUSH {r4-r12, lr}
    MOV r5, r0
    LDR r12, addr_top1
    STR r5, [r12]
    MOV r4, r1
    LDR r12, addr_bottom1
    STR r4, [r12]
    MOV r6, #0
    LDR r12, addr_i1
    STR r6, [r12]
    CMP r5, r4
    MOVGE r12, #1
    MOVLT r12, #0
    CMP r12, #0
    BEQ label1
label2:
    SUB r11, r5, r4
    MOV r5, r11
    LDR r10, addr_top1
    STR r5, [r10]
    ADD r6, r6, #1
    LDR r12, addr_i1
    STR r6, [r12]
    CMP r5, r4
    MOVGE r12, #1
    MOVLT r12, #0
    CMP r12, #0
    BNE label2
label1:
    MOV r0, r6
    POP {r4-r12, pc}
    POP {r4-r12, pc}

main:
    PUSH {r4-r12, lr}
    MOV r4, #0
    LDR r12, addr_pi3
    STR r4, [r12]
    MOV r12, #100
    MOV r10, #100
    MUL r11, r12, r10
    MOV r10, #100
    MUL r12, r11, r10
    MUL r12, r11, r10
    MOV r5, r12
    LDR r11, addr_million3
    STR r5, [r11]
    MOV r7, #4
    LDR r12, addr_j3
    STR r7, [r12]
    MOV r6, #1
    LDR r12, addr_i3
    STR r6, [r12]
label3:
    MOV r11, #100
    MUL r12, r5, r11
    MOV r11, #2
    MUL r10, r11, r6
    SUB r11, r10, #1
    MOV r0, r12
    MOV r1, r11
    BL myDiv
    MOV r12, r0
    MUL r11, r7, r12
    ADD r12, r4, r11
    MOV r4, r12
    LDR r11, addr_pi3
    STR r4, [r11]
    MOV r12, #0
    SUB r11, r12, r7
    MOV r7, r11
    LDR r12, addr_j3
    STR r7, [r12]
    ADD r6, r6, #1
    LDR r11, addr_i3
    STR r6, [r11]
    CMP r6, r5
    MOVLT r12, #1
    MOVGE r12, #0
    CMP r12, #0
    BNE label3
    LDR r0, print_args1
    MOV r1, r4
    BL printf
    POP {r4-r12, pc}
```

Amazingly, this code actually works, and gave the same result as the gcc compiler! I got out a stopwatch to compare the speeds of them both (due to there being no timing functions implemented in ARM) and gcc clocked in at **1.94s,** while my arm got **20.73s**. It's a lot slower than gcc but I'm pretty happy with a 10x difference (especially on a raspberry pi vs a desktop PC), and even happier that it actually works and doesn't crash.