CSE 302: Compilers | Lab 3

Control Structures and x64 Assembly

Starts:	`2020-09-24`
Checkpoint:	`2020-10-01 23:59:59`
Lab due:	**`2020-10-08 23:59:59`**

---

1 INTRODUCTION

In this lab we will extend our source language from `BX0` to `BX1`, which adds support for boolean values, boolean expressions, and control structures. We will also extend our intermediate language, `TAC`, with support for labels and jumps. Finally, in this lab you will build your first complete compiler, targeting `x64` assembly. You will be required to be able to assemble and link your assembly output into executables.

This lab will be assessed. It is worth **15%** of your final grade. You must work in groups of size 2 or 3.

Covid-19

Observe all health guidelines—particularly the interpersonal distancing guidelines—when working in groups. Use a video-conferencing platform to collaborate. If a physical meeting is unavoidable, try to do it outdoors, keep it short, and wear masks for the entire duration.

2 STRUCTURE OF THE LAB

This lab involves a checkpoint at the end of the first week. Every group is required to submit a checkpoint. The checkpoint will be graded for 50% credit only in case you fail to do anything for week 2 of the lab. Keep in mind that we will give partial credit for incomplete solutions, so make sure to submit something for the full lab by the due date regardless of how far you get.

CHECKPOINT DELIVERABLES The checkpoint will consist of a backend *instruction selection* pass that will produce `x64` assembly from `TAC` (extended with labels and jumps). Like in lab 2, we will give you a self-contained `TAC` library, parser, and interpreter (`tac.py`). You will need to design a pass that goes from a `TAC` file `example.tac` to an `x64` assembly file `example.s`, which can then be compiled into `example.exe` using gcc and the `BX` runtime. This task is explained in more detail in section 3.

FINAL DELIVERABLES In the second week of the lab you will write the frontend and middle of the compiler that builds `TAC` from the `BX1` language, which is specified in section 4. In addition to extending your parser for `BX0` to that for `BX1`, you will need to write a syntactic analysis pass that produces an abstract syntax tree (AST), for which you will then write a type-checker. You will then adapt the *maximal munch* algorithms from lab 2 to transform the AST to `TAC` code. You may also implement the *typed maximal munch* variant that makes use of the type information to generate more compact code involving boolean expressions. These tasks are explained in more detail in section 5.

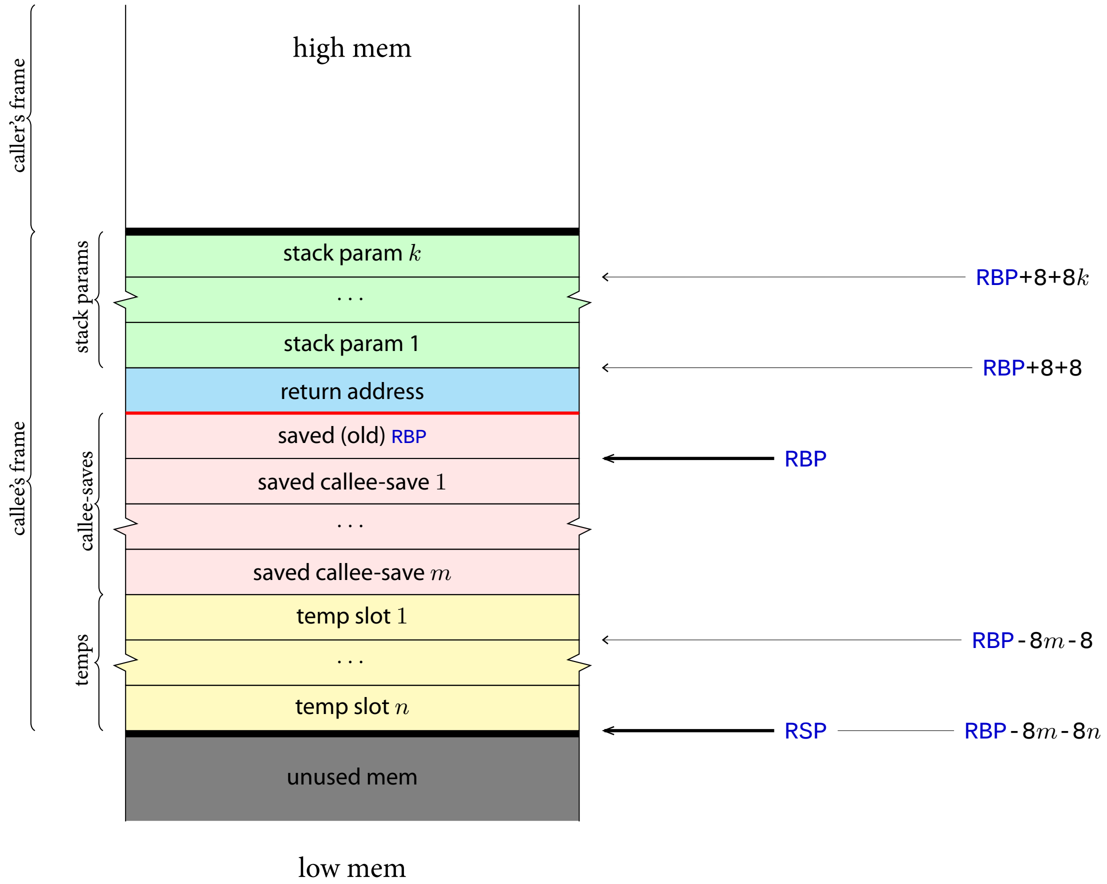

Figure 1: A schematic diagram of a stack frame

3 ASMGEN: FROM TAC TO X64 (WEEK 1 – CHECKPOINT)

3.1 *Mapping* TAC *Temporaries to* x64

Registers and Stack Slots x64 has only 14 *general purpose registers* (GPRs) available for computation. Of these GPRs, a further 5 are *callee-save* registers, and are therefore inadvisable to use at present, since you will not yet have a lot of sophistication in managing the stack. Therefore, the recommendation is to use only the remaining 9 registers: `RAX`, `RCX`, `RDX`, `RSI`, `RDI`, `R8`, `R9`, `R10`, and `R11`.

TAC, on the other hand, can use an arbitrary number of temporaries. Therefore, to compile TAC to x64, you will have to keep these temporaries in main memory, specifically the *stack*. For now, it is useful to think of the stack as being built of *stack slots*. Each temporary that is used in the TAC program should have a dedicated stack slot, which we can identify with a number $\in \{1, 2, \ldots, n\}$ where n is the total number of temporaries. You need to create and manage this mapping in your code.

The Stack Figure 1 contains a schematic diagram of the stack, highlighting a single stack frame. For the purposes of this project, we will only focus our attention on the yellow portion of the figure. (We will explore the rest of the elements of the stack frame in the next lab.)

When the program begins, the `RSP` register points to the *top* of the stack, which (by convention) is the lowest allocated memory location in the stack area of the program. The stack grows downwards from high memory to low memory, so to allocate new stack slots it suffices to decrement `RSP` by the number of slots desired, multiplied by 8 since each stack slot is 8 bytes (64 bits) wide. Therefore, to allocate 42 stack slots, you would need to decrement `RSP` by $42 \times 8 = 336$.

The Frame Pointer, `RBP` At the end of the program, you need to restore the stack pointer, `RSP`, to its initial value; if you don't, your program will most likely crash on exit. To achieve this, a common technique is to use the `RBP` register, known as the *base pointer* or more commonly the *frame pointer*, to store the old value of `RSP`. However, `RBP` itself is a callee-save register, so it too must be restored on exit from a function; therefore, `RBP` is also stored in the stack up front (before allocating the rest of the stack slots for temporaries), and then restored after `RSP` is restored. If you follow this protocol, then the region of memory between `RSP` and `RBP` will be where the stack slots assigned to temporaries are to be found.

Accessing the Contents of the Stack Since we are not using any callee-save registers, the pink region for callee-saves will be limited to just the saved `RBP`; i.e., for us $m = 0$. Therefore, the first slot for temporaries will be at offset `RBP - 8`, and the nth temporary will be located at `RBP - 8`n. Note that memory locations grow upwards, so the first temporary (e.g.) will be laid out in the bytes between `RBP - 8` and `RBP`. Stack slots are always referenced by the location of their first byte.

To get/store the contents of the n slot, we will need to dereference the memory address `RBP - 8`n. In `x64`, this is written conveniently as `-8`n`(%rbp)`; that is, the various slot contents are `-8(%rbp)`, `-16(%rbp)`, `-24(%rbp)`, `-32(%rbp)`, …

Setup To put this together, here is a template you can reuse to build your assembly file for a `BX1` program. The template assumes that it is allocating 7 stack slots for 7 temporaries; you will have to modify this in your compiler

```
    .globl main
    .text
main:
    pushq %rbp           # store old RBP at top of the stack
    movq %rsp, %rbp      # make RBP point to just after stack slots


    # Now we allocate stack slots in units of 8 bytes (= 64 bits)
    # E.g., for 7 slots, i.e., 7 * 8 = 56 bytes
    #     -- MODIFY AS NEEDED --
    subq $56, %rsp

    #
    # The rest of the compiled code from TAC goes here.
    #

    movq %rbp, %rsp      # restore old RSP
    popq %rbp            # restore old RBP
    movq $0, %rax        # set return code to 0
    retq                 # exit
```

3.2 *Instruction Selection*

We recommend that you limit yourself to the following simple subset of the `x64` assembly language. This will minimize complications when trying to convert `TAC` to `x64`. Later, once you have a functional assembly generator, you can experiment with other instructions outside this set. Whenever you try such experiments, make sure to pre-write a regression test case that triggers the modification, and then always check that your experiment yields the same results before and after the modification.

Operand Specifiers In `x64`, instructions can take operands of several different forms, and each form has a unique *operand specifier*. For now we will only use the following specifiers.

kind	example	description
Immediate	`$42`	The value can be in decimal or hexadecimal (using the prefix `0x`). Don't forget the `$` – without it, it will be interpreted as a raw absolute memory address, not an immediate value.
Register	`%rax`	Registers are named with `%` followed by the name of the register in lowercase.
Dereference	`(%rax)`	Gets or sets the value stored at the memory location contained in the given register.
Dereference w/ Offset	`42(%rax)`	Adds the offset to the register value to get the location being dereferenced. Note that the offset can be negative.

In all of the following, the page references are to the document *"AMD64 Architecture Programmer's Manual (vol 3): General Purpose and System Instructions"*, where these instructions are described in the Intel syntax that puts the destination operand first instead of last. We will use the AT&T/GNU syntax that places the destination operand last.

Data Transfer Instructions

instruction	description	page
`movq Src, Dst`	Move `Src` value to `Dst`.	231
`pushq Src`	Decrement `RSP` by 8 and put `Src` into where it points to afterwards	285
`popq Dst`	Load the value pointed to by `RSP` into `Dst`, then increment `RSP` by 8	273

In these and all subsequent instructions, both `Src` and `Dst` cannot be dereferences simultaneously.

Arithmetic Instructions

instruction	description	page
`addq Src, Dst`	Increment `Dst` by the value of `Src`	83
`subq Src, Dst`	Decrement `Dst` by the value of `Src`	342
`imulq Src, Dst`	Multiply `Dst` by the value of `Src`	178
`andq Src, Dst`	Bitwise-and `Dst` with the value of `Src`	87
`orq Src, Dst`	Bitwise-or `Dst` with the value of `Src`	262
`xorq Src, Dst`	Bitwise-xor `Dst` with the value of `Src`	359

instruction	description	page
`notq Dst`	Bitwise-not `Dst` (i.e., flip all its bits)	261
`negq Dst`	Negate `Dst`	258

Arithmetic Instructions with Fixed Operands

instruction	description	page
`sarq Src, Dst`	Arithmetic right-shift `Dst` by the amount `Src`. `Src` cannot be a dereference. If `Src` is a register, it must be `%cl`.	314
`salq Src, Dst`	Arithmetic left-shift `Dst` by the amount `Src`. `Src` cannot be a dereference. If `Src` is a register, it must be `%cl`.	311
`idivq Src`	Signed divide `RDX:RAX` by `Src`, storing quotient in `RAX` and remainder in `RDX`	176
`cqto`	Sign-extend `RAX` into a 128-bit value `RDX:RAX`	140

Conditions and Jumps

instruction	description	page
`cmpq Src1, Src2`	Set the flags register based on the result of computing `Src2 - Src1`. Carefully note the order of the operands of the subtraction!	155
`jmp Lbl`	Unconditionally jump to local label `Lbl`	199
`jcc` `Lbl`	Conditional jump to local label `Lbl`. Here, *`jcc`* is one of the opcodes in the table below, with the interpreted condition with reference to `cmpq` above	194

`jcc`	condition
`je`, `jz`	`Src2 == Src1`
`jne`, `jnz`	`Src2 != Src1`
`jl`, `jnge`	`Src2 < Src1`
`jle`, `jng`	`Src2 <= Src1`
`jg`, `jnle`	`Src2 > Src1`
`jge`, `jnl`	`Src2 >= Src1`

3.3 *Dealing with* `print`

The `print` statement of TAC will be compiled by making a function call from x64 to the BX runtime function `bx_print_int()`. For this lab, the runtime is just the file `bx_runtime.c` shown in figure 2. You have to link it to create the final executable, as explained in section 3.4.

From within x64, calls to `bx_print_int()` will be done as follows: (1) place the argument to the function in `RDI`, then (2) use the instruction: `callq bx_print_int`. For example, here is how you would compile `print %42` assuming `%42` was assigned to stack slot 7.

```
/* This should be in a file such as: bx_runtime.c */

#include <stdio.h>
#include <stdint.h>

/* Note: TAC int == C int64_t
   This is because C int is usually only 32 bits. */

void bx_print_int(int64_t x)
{
  printf("%ld\n", x);
}
```

Figure 2: The BX "runtime"

```
    pushq %rdi                # if you're currently using RDI for anything else
    pushq %rax                # if you're currently using RAX for anything else
    movq -56(%rbp), %rdi      # load stack slot 7 (note: 7 * 8 == 56)
    callq bx_print_int
    popq %rax                 # if you pushed RAX
    popq %rdi                 # if you pushed RDI
```

The saves (`pushq`s) of `RDI` and `RAX`, and their subsequent restores (`popq`s), are optional. They are only needed if you are storing values in these registers that you will need access to after the `print`. These are caller-save registers, so callees such as `bx_print_int()` are allowed to modify them as needed.

3.4 *Building and Debugging Executables*

Once you have produced an assembly file, say `example.s`, you should use `gcc` to link it together with your runtime in one shot. Use the following invocation:

```
$  gcc -g -o example.exe example.s bx_runtime.c
```

The `-g` flag is recommended since it allows you to use the debugger, `gdb`, to step through your assembly code and aid in debugging it. Figure 3 shows an example interaction with `gdb`, with example commands that should be sufficient for all the things you are doing in this lab. You may also need the `gdb` manual.

3.5 *What You Should Submit for the Checkpoint*

Your main program should be called `tacx64.py` (or `tacx64.exe` if you're not using Python). It should at the very least accept a single file in the command line, e.g., `prog.tac`, and it should produce a corresponding x64 assembly file (here, `prog.s`). You don't need to produce `prog.exe` by running `gcc` (but you can if you wish).

```
$ python3 tacx64.py file.tac       # should produce file.s
```

```
$ gdb example.exe
  … several lines of output…
Reading symbols from example.exe...
(gdb) list main                                        (display the assembly code of main())
  … several lines of output…

(gdb) break 5                                          (set breakpoint on line 5)
Breakpoint 1 at 0x1139: file example.s, line 5.

(gdb) run
Starting program: /home/kaustuv/302/lab/03/handout/example.exe

Breakpoint 1, main () at example.s:5
5             cmpq $0, %rcx

(gdb) info register rcx                                (examine a register)
rcx    0xffffffffffffffd6    -42                       (2's complement hex & decimal)

(gdb) info registers                                   (see all registers at once)
  … several lines of output…

(gdb) x/dg $rbp - 8                                    (see stack slot 1)
0x7fffffffe098: 10

(gdb) print ($rbp - $rsp) / 8                          (compute size of stack in #slots)
$1 = 7

(gdb) x/7dg $rsp                                       (see bottom 7 stack slots, printed low-to-high)
0x7fffffffe068: 93824992235925 0                       (low mem, closer to RSP)
0x7fffffffe078: 0              93824992235856
0x7fffffffe088: 93824992235584 140737488347536
0x7fffffffe098: 10                                     (high mem, closer to RBP)

(gdb) set $rcx = -300                                  (change value of register, here RCX)

(gdb) set {long int}($rbp - 56) = -300                 (change value of stack slot, here slot #7)

(gdb) x/dg ($rbp - 56)
0x7fffffffe068: -300

(gdb) next                                             (run to next line)
6             jg .L0

(gdb) quit
A debugging session is active.

        Inferior 1 [process 207327] will be killed.

Quit anyway? (y or n) y

$
```

Figure 3: An example gdb session

4 THE BX1 LANGUAGE

BX1 is a strict superset of BX0, so any BX0 program continues to be a valid BX1 program. The additions of BX1 are as follows:

- A new type, `bool`, of *booleans*. BX1 of course retains the 64-bit signed integer type `int` from BX0. Note that BX1 does not have any *variables* of `bool` type; indeed, all BX1 variables continue to be `int` variables.
- A number of new operators that produce values of `bool` type. This includes the comparison operators (==, !=, <, <=, >, and >=) for comparing two `int` expressions, and the boolean connectives `&&`, `||`, and `!`. There are also two new constants of `bool` type: `true` and `false`.
- Conditional `if` ... `else` ... statements.
- Looping `while` ... statements.
- The two structured jumping statements, `break` and `continue`.

The lexical structure and grammar of BX1 is shown in figure 4. The extended operator precedence table is shown in figure 5. As usuall, the overall BX1 program is represented by the nonterminal ⟨program⟩. In the rest of this section we will specify the semantics of the new features of BX1.

Boolean Relations The six new binary relational operators, {==, !=,<,<=,>,>=}, are used to compare the values of signed 64-bit integers. These operators are *non-associative*, meaning that there is no particular meaning ascribed to expressions such as `x == y == z` or `x <= y < z`. Such expressions would be considered to be parse errors.

Note that the == and != operators are used to compare `int`s alone. Equality of booleans can be expressed in a different way: `b1 == b2` (for boolean expressions `b1` and `b2`) is written `!(b1 ^ b2)` (i.e., the *nxor* of `b1` and `b2`), while `b1 != b2` is just their *xor*, `b1 ^ b2`.

Boolean Connectives and Short-Circuiting The two binary boolean connectives `&&` and `||` and the unary boolean negation `!` have the following truth tables.

b1	b2	b1 && b2	b1 || b2	!b1
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

The binary operators `&&` and `||` are also *short-circuiting*. To compute the value of the expression `b1 && b2`, first `b1` is evaluated; if it is `false`, then the value of `b1 && b2` is taken to be `false` and `b2` is not evaluated. Likewise, the value of `b1 || b2` is taken to be `true` if `b1` evaluates to `true` without evaluating `b2`.

Conditionals The general form of the `if` ... `else` ... statement is shown in figure 6. This form in BX1 is inspired by C. Immediately after the condition, there is a *block* (delimited by {}) that is executed if the condition evaluates to `true`. If the condition evaluates to `false` instead, the control moves to the *optional* remainder of the expression that is separated by means of the `else` keyword. The remainder could contain further conditions to check, or it could be a final fallback for when none of the conditions is `true`. Note that the conditions are evaluated top-to-bottom, and the first conditional that evaluates to `true` causes its corresponding body to be evaluated.

```
⟨program⟩ ::= ⟨stmts⟩

⟨stmts⟩ ::= ϵ | ⟨stmt⟩ ⟨stmts⟩
⟨stmt⟩ ::= ⟨assign⟩ | ⟨print⟩ | ⟨ifelse⟩ | ⟨while⟩ | ⟨jump⟩
⟨assign⟩ ::= IDENT '=' ⟨expr⟩ ';'
⟨print⟩ ::= 'print' '(' ⟨expr⟩ ')' ';'
⟨ifelse⟩ ::= 'if' '(' ⟨expr⟩ ')' ⟨block⟩ ⟨ifrest⟩
⟨ifrest⟩ ::= ϵ | 'else' ⟨ifelse⟩ | 'else' ⟨block⟩
⟨while⟩ ::= 'while' '(' ⟨expr⟩ ')' ⟨block⟩
⟨jump⟩ ::= 'break' ';' | 'continue' ';'
⟨block⟩ ::= '{' ⟨stmts⟩ '}'

⟨expr⟩ ::= IDENT | NUMBER | 'true' | 'false'
         | ⟨expr⟩ ⟨binop⟩ ⟨expr⟩ | ⟨unop⟩ ⟨expr⟩
         | '(' ⟨expr⟩ ')'

⟨binop⟩ ::= '+' | '-' | '*' | '/' | '%' | '&' | '|' | '^' | '<<' | '>>'
          | '==' | '!=' | '<' | '<=' | '>' | '>=' | '&&' | '||'

⟨unop⟩ ::= '-' | '~' | '!'

IDENT ::= [A-Za-z_][A-Za-z0-9_]*                    (excepting keywords)
NUMBER ::= 0|[1-9][0-9]*                 (numerical value ∈ [0, 2^63))
```

Figure 4: The lexical structure and grammar of the BX1 language. To keep things readable, the tokens that are representable as strings have been included inline in the BNF instead of being defined separately; these inlined tokens are indicated as strings such as `'print'`.

operator	description	arity	associativity	precedence
`||`	boolean disjunction (or)	binary	left	3
`&&`	boolean conjunction (and)	binary	left	6
`|`	bitwise or	binary	left	10
`^`	bitwise xor	binary	left	20
`&`	bitwise and	binary	left	30
`==`, `!=`	(dis-)equality	binary	nonassoc	33
`<`,`<=`,`>`,`>=`	inequalities	binary	nonassoc	36
`<<`,`>>`	bitwise shifts	binary	left	40
`+`,`-`	addition, subtraction	binary	left	50
`*`,`/`,`%`	multiplication, division, modulus	binary	left	60
`-`, `!`	integer/boolean negation	unary	–	70
`~`	bitwise complement	unary	–	80

Figure 5: BX1 operator arities and precedence values. A higher precedence value binds tighter.

```
if (cond₁) {
  // body₁
}
else if (cond₂) {
  // body₂
}
else if (cond₃) {
  // body₃
}
⋮
// optional:
else {
  // code that runs if none of the condᵢ is true
}
```

Figure 6: General form of the BX1 conditional.

Loops BX1 has only a single kind of loop, the `while` ... loop. Its syntax is inspired by C and consists of a single condition ⟨expr⟩ that is evaluated for every iteration of the loop. If the condition evaluates to `true`, then the body is evaluated, and and control subsequently returns to the start of the `while` ... loop. If the condition evaluates to `false`, the entire loop is skipped and control moves to the next instruction.

Structured Jumps The two structured jump statements, `break` and `continue`, are allowed to occur in the scope of a `while` ... loop. They are inspired by the identically named constructs from C.

- The `break` statement exits the innermost loop in which the statement occurs. In other words, control jumps to the statement after the innermost `while` ... statement, as if the condition of the statement had evaluated to `false`.
- The `continue` statement immediately jumps to the start of the innermost `while` ... loop. (It turns out that `continue` is not that useful in BX1, but it will be a handy control structure when we add ranged `for`-loops to BX.)

It is a semantic error for these statements to occur outside the body of a loop.

5 IRGEN: FROM BX1 TO TAC (WEEK 2)

Type Information To begin with, build an abstract syntax tree (AST) structure for BX1 with support for type information. Use the features of your chosen programming language to achieve this. In lecture 4 you have seen how to do it in Python using a hierarchy of classes, with each expression subclass having a read-only `.ty` attribute that can be used to access the type of the expression. Place your AST classes in a separate module, say `ast.py`.

Typed Maximal Munch It is your choice whether to use the untyped maximal munch (lecture 3) or typed maximal munch (lecture 4) to handle boolean expressions, but it should be obvious at a glance that the typed variant is shorter and considerably easier to understand. Therefore, it is recommended that you use the typed variant for handling conditions in `if` ... `else` ... and `while` ... statements.

Standalone Front-End Start by ignoring the back-end of the compiler (TAC onwards) and write a standalone BX1 to TAC converter. Call it `bx1tac.py`. Its behavior will be similar to the `bx0tac.py` program you wrote for lab 2: it will take a single `.bx` file as input and convert it to a corresponding `.tac` file. You can then run the provided `tac.py` interpreter on the generated `.tac` file to make sure that your maximal munch implementation is displaying the correct behavior.

Final Deliverable: `bx1cc.py` To put things together, write an overall wrapper program called `bx1cc.py` (`bx1cc.exe` if you are not usng Python) that will chain the phases corresponding to `bx1tac` and `tacx64` together to go from a `.bx` file to a `.s` file.

```
$ python3 bx1cc.py file.bx        # should produce file.s
```

This wrapper is only required to produce a `.s` file. However, you may find it useful to enrich the wrapper with some command-line flags (e.g., `--keep-tac`) that will cause it to also produce the intermediate `.tac` file. You may also want a `--stop-tac` flag that will make the wrapper stop after creating the `.tac` file, so that you can debug the front-end along. Finally, you may also allow the wrapper to accept `.tac` files as input for which you only run the back-end (`tacx64`) phase.

Building Executables As before with `tacx64`, it is not necessary for your compiler to perform the final assembling and linking step to go from a `.s` file to the executable `.exe` file. However, you may still want to call `gcc` directly from `bx1cc.py` because it gets tedious and error-prone to call gcc manually.

In order to run commands from within Python, it is a good idea to make use of the `subprocess` library. Here, for example, is a function that given an assembly file `prog.s` file and the runtime file `bx_runtime.c` will construct the executable `prog.exe`.

```python
import subprocess
def assemble_and_link(asm_file, runtime_file, debug=False):
    """Run gcc on `asm_file' and `runtime_file' to create a suitable
    executable file. If `debug' is True, then also send the `-g' option
    to gcc that will store debugging information useful for gdb."""
    cmd = ["gcc"]
    if debug: cmd.append("-g")
    assert asm_file.endswith('.s')
    exe_file = asm_file[:-1] + 'exe'
    cmd.extend(["-o", exe_file, asm_file, runtime_file])
    result = subprocess.run(cmd)
    return result.returncode  # 0 on success, non-zero on failure
```