CS 2110 Homework 07

Kyle Murray, Austin Adams, and Cem Gökmen

Fall 2017

Contents

1 Assignment

1.1 BitVector in LC-3 – 50 Points

In this first part of the homework, you'll implement a few functions from BitVector back in Homework 2: `clear()`, `set()`, `isset()`, and `toggle()`. Both functions take the bits on which to operate as their first argument, the index of the bit to set/clear as the second argument, and return the modified bits. The index is in the range 0 to 15 (inclusive) and bit 0 is the least significant bit. Assume all arguments are nonnegative.

1.1.1 `leftshift(number, n)` – 10 Points

LC-3 lacks an instruction for a left bit shift, so to help in writing `clear()`, `set()`, and `isset()` later, you'll have to take a (painfully) inefficient approach for left shifting: implementing a function which shifts a nonnegative integer number left by n bits. That is, it returns `number << n`. [1]

Here's the pseudocode:

```
// Left shifts number(assumed to be unsigned) by n bits
// leftshift(4, 1) => 8
// leftshift(0xFFFF, 3) => 0xFFF8
int leftshift(number, n) {
    while (n > 0) {
        n--;
        number = number + number; // a single left shift
    }
    return number;
}
```

1.1.2 `clear(bits, index)` – 10 Points

`clear()` behaves like `clear()` from BitVector in homework 2 except that you pass in `bits` as the first argument and it returns the result. That is, it clears bit `index` of `bits` and returns the result. (Recall that `index` begins at bit 0, the least significant bit of bits.) Assume $0 \leq \text{index} \leq 15$.

Here's the pseudocode:

```
// clear(0xFFFF, 0) => 0xFFFE
// clear(0x8000, 15) => 0
// clear(0xFF00, 0) => 0xFF00
int clear(bits, index) {
    int mask = leftshift(1, index);
    return bits & ~mask;
}
```

[1] Using C syntax here is a little misleading, because when $n \geq 16$, your `leftshift()` should return 0, but this is not necessarily how C would behave on the LC-3. Indeed, an `int` on the LC-3 would probably be 16 bits and according to C99, "if the value of the right operand [of `<<` or `>>`] is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined."

1.1.3 `set(bits, index)` – 10 Points

`set()` behaves like `set()` from BitVector in Homework 2 except that you pass in `bits` as the first argument and it returns the result. That is, it sets bit `index` of `bits` and returns the result. (Recall that `index` begins at bit 0, the least significant bit of bits.) Assume $0 \leq \text{index} \leq 15$.

Here's the pseudocode:

```
// set(0, 15) => 0x8000
// set(1, 0) => 1
// set(0xF000, 3) => 0xF008
int set(bits, index) {
    int mask = leftshift(1, index);
    int result = ~mask & ~bits;
    return ~result;
}
```

1.1.4 `isset(bits, index)` – 10 Points

`isset()` behaves like `isSet()` from BitVector in Homework 2 except that you pass in `bits` as the first argument and it returns 1 if the given bit is set and 0 otherwise. (Recall that `index` begins at bit 0, the least significant bit of bits.) Assume $0 \leq \text{index} \leq 15$.

Here's the pseudocode:

```
int isset(bits, index) {
    int mask = leftshift(1, index);
    if (bits & mask == 0) {
        return 0;
    } else {
        return 1;
    }
}
```

(We'll skip implementing `isclear()` because it's equivalent to `!isset()`.)

1.1.5 `toggle(bits, index)` – 10 Points

`toggle()` uses every bitvector function you've written so far to toggle the bit `index` of the integer `bits`. That is, if the bit at `index` is 0, set it to 1 (and vice versa) and return the result. (Recall that `index` begins at bit 0, the least significant bit of `bits`.) Assume $0 \leq \text{index} \leq 15$.

Here's the pseudocode:

```
int toggle(bits, index) {
    if (isset(bits, index) == 0) {
        return set(bits, index);
    } else {
        return clear(bits, index);
    }
}
```

Warning: while you *could* implement `toggle()` with an XOR like you did in Homework 2, please do not or you will lose points. Our testers look for calls to `isset()`, `set()`, and `clear()` because we want you to practice calling subroutines with the LC-3 calling convention. Follow the pseudocode.

1.2 GCD – 20 Points

Now that you have a decent grasp of how function calls work and how stacks are necessary for them, you will be implementing a simple recursive function that finds the Greatest Common Divisor of two values.

Here's the pseudocode:

```
// GCD(50, 15) => 5
// GCD(13, 12) => 1
// Assume a is nonnegative and b is positive
GCD(a, b) {
    R0 = a;        // a must be put in R0
    R1 = b;        // b must be put in R1
    UDIV();        // UDIV is a trap that divides and mods
    // R0 = a / b
    // R1 = a % b

    if(R1 == 0) {     // if((a % b) == 0)
        return b;
    } else {
        return GCD(b, R1);
    }
}
```

This function uses the trap called `UDIV` included in Complx. Unlike other traps you've used (such as `GETC`, `OUT`, `PUTS`, and `HALT`), this trap is not built into LC3 but is an external plugin you must include. The provided `gcd.asm` file already includes the plugin at the top using the `@plugin` comment. **Do not modify this line.** The way `UDIV` works is that it accepts two arguments through registers R0 and R1 and returns the division and modulo of those two in the same registers, as shown in the comment in the pseudocode. R2-R6 are not modified, R7 is of course overwritten for the return address.

1.3 Binary Search – 30 Points

Sometimes the best approach to a problem is a recursive one. Binary search is one such example. When given a sorted array which may include negative elements, you want to return the index of the element you are searching for, or if it is not in the array return the value -1. Start by looking at the middle index between index start (inclusive) and index end (exclusive). If your target value is less than the value at that index, continue your search (via recursion) in the left half of the array.

Here's the pseudocode:

```
// int[] arr = {-3,0,2,3,5,9};
// binSearch(arr, 5, 0, arr.length) => 4
// binSearch(arr, 1, 0, arr.length) => -1

int binSearch(int[] arr, int target, int start, int end) {
    if (start >= end) {
        return -1;
    }
    int middle = (start + end) / 2; // Use UDIV for division here
    int val = arr[middle];
    if (val == target) {
        return middle;
    } else if (val > target) {
        return binSearch(arr, target, start, middle);
    } else {
        return binSearch(arr, target, middle+1, end);
    }
}
```

val - target
BRz EQUAL
BRp
GREATER
BRn LESS

1.4 Tips and Tricks

A few pointers that will help you out:

1. Comment your code! It is very easy to get lost in assembly.
2. First thing you should do when writing any function is to setup the frame at the top of the function and teardown the frame at the bottom. I recommend a common "RETURN" at the bottom, this way multiple ways to return would just branch there.
3. Complx is a very powerful debugger, offering the ability to step forward through code, step backwards through code, and setting breakpoints. When you run your code and it doesn't work, step through it line by line and ensure all your registers and the stack are correct at each step.
4. In this class, you should use callee save for registers R0-R4, meaning functions should back up any of R0-R4 they modify to local variables and then restore them before they return. This way, registers R0 through R4 keep their values in the caller even after calling a function. (Note: this is kind of a late change to this PDF, so if you already implemented caller save, where registers are backed up in the caller instead, don't worry. Our tests allow either. However, in the future you should follow the calling convention for this class by doing callee save.)
5. In the `bitvector.asm` file, you are implementing several functions in the same file. Remember that when the assembler looks at your file, it first reads all labels (symbols) declared. This means you cannot use duplicate label names. If you follow Tip #2, you will likely want a `RETURN` label in each function. A recommended approach is to prepend the initials of the function to all your labels, so `isset()` would have `ISSET_RETURN`, `set()` would have `SET_RETURN`, etc.
6. Inspecting the stack in Complx can be done in a separate view: hit Ctrl+V to open a new memory view then hit Ctrl+G to jump to address xF000 for the stack.
7. For testing your code, we've provided the following XML files:
   (a) `bitvector_test.xml`, for `bitvector.asm`
   (b) `gcd_test.xml`, for `gcd.asm`
   (c) `bin_search_test.xml`, for `bin_search.asm`
   (d) Run these tests in the terminal with `lc3test` like: `lc3test X_test.xml X.asm`
   (e) While these **tests are not comprehensive**, if you pass them that means there is a high likelihood of the function being correct. We will be using our own more comprehensive tests for grading. It is recommended you modify the XML files to create more tests. You may share these tests on Piazza as well.
8. **Debugging Test Failures**: Don't panic if your code doesn't pass the tests on the first try — our solutions didn't either. To debug a test (with breakpoints etc.), load your code in Complx and go to Debug → Simulate Subroutine Call, choose your subroutine, enter the same arguments as the test uses (comma-separated), and then press "Ok." This will setup the PC and stack as needed to call your subroutine, so you can (for example) hit "Run" to execute the function. After it returns, you can peek at the stack using the instructions in Tip 6 above.

1.5 Deliverables

Remember to put your name at the top of each file you submit. The files should be named exactly as given.

1. `bitvector.asm`
2. `gcd.asm`
3. `bin_search.asm`

2 LC-3 Assembly Programming Requirements

2.1 Overview

1. Your code must assemble with **NO WARNINGS OR ERRORS**. To assemble your program, open the file with Complx. It will complain if there are any issues. **If your code does not assemble you WILL get a zero for that file.**

2. **Comment your code!** This is especially important in assembly, because it's much harder to interpret what is happening later, and you'll be glad you left yourself notes on what certain instructions are contributing to the code. Comment things like what registers are being used for and what less intuitive lines of code are actually doing. To comment code in LC-3 assembly just type a semicolon (;), and the rest of that line will be a comment.

3. Avoid stating the obvious in your comments, it doesn't help in understanding what the code is doing.

   Good Comment

   ```
   ADD R3, R3, -1          ; counter--
   BRp LOOP                ; if counter == 0 don't loop again
   ```

   Bad Comment

   ```
   ADD R3, R3, -1          ; Decrement R3
   BRp LOOP                ; Branch to LOOP if positive
   ```

4. **DO NOT assume that ANYTHING in the LC-3 is already zero.** Treat the machine as if your program was loaded into a machine with random values stored in the memory and register file.

5. Following from 3. You can randomize the memory and load your program by doing File - Randomize and Load.

6. Use the LC-3 calling convention. This means that all local variables, frame pointer, etc... must be pushed onto the stack. Our autograder will be checking for correct stack setup.

7. Start the stack at xF000. **The stack pointer always points to the last used stack location.** This means you will allocate space **first**, then store onto the stack pointer.

8. Do NOT execute any data as if it were an instruction (meaning you should put .fills after **HALT** or RET).

9. Do not add any comments beginning with @plugin or change any comments of this kind.

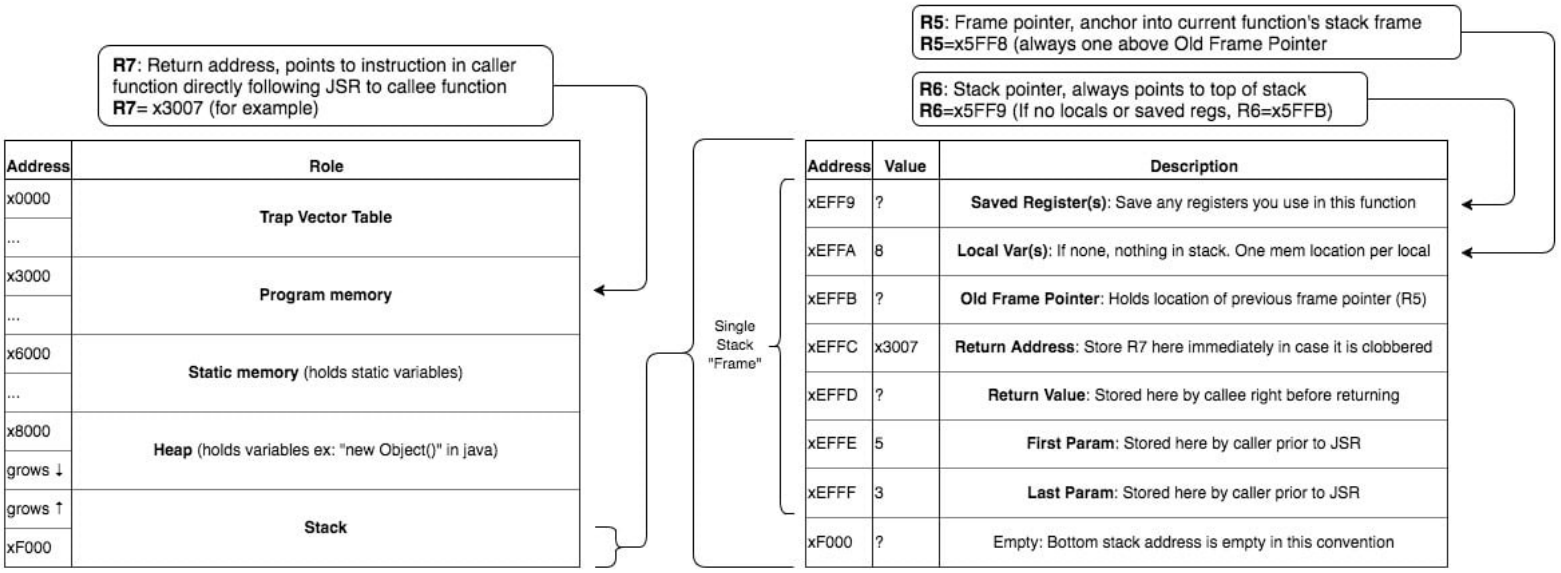
10. **Test your assembly.** Don't just assume it works and turn it in.

2.2 In-Depth Guide to the LC-3 Calling Convention

Kyle's LC-3 calling convention guide follows:

Overview

On the left is a grand-scale layout of memory in a computer. On the right is a zoomed in view of the stack. The state of the stack is a snapshot during the execution of some function (we will call it "foo"). Since the stack begins at xF000, we know that we are currently executing the first function called by main (main itself does not have its own stack frame). At this state in time, the main function is the "caller", and the currently executing function, foo, is the "callee". Appropriate example addresses and values are included throughout this guide, but note that these values are not necessarily the standard.



Walkthrough

In the following walkthrough, function "foo" will call function "bar", and you will be shown snapshots of the stack at different stages of the calling convention. Note that "caller/foo" and "callee/bar" will be used interchangeably. For transitions 2, 3, and 4 (not 1, 2.1, or 3.0), the instructions can be EXACTLY the between different functions, a "cookie cutter" set of instructions. The following pseudocode describes the two functions (they don't do anything important, they are just examples of functions you might have to recreate in assembly):

```
int foo (int a, int b) {
    int local_var = a + b;
    answer = bar(a, local_var);
    return answer;
}
```

```
int bar (int c, int d) {
    int var1 = c + d;
    int var2 = c - d;
    answer = var1 / var2;
    return answer;
}
```

State 0 - During Foo's Execution

```
int foo (int a, int b) {
    int local_var = a + b;
    answer = bar(a, local_var);   <-- PC
    return answer;
}
```

The function has stored local_var but not yet called bar. Foo is about to call bar so we will now refer to foo as the "caller" and bar as the "callee". Remember the PC points to the *planned-to-be-next instruction.*

Address	Value	Description	
xEFF9	?	**Foo's Saved Register(s)**: Foo saved all registers it used before use	R6
xEFFA	8	**Foo's Local Var(s)**: stores value of local_var	R5
xEFFB	?	**Foo's Old Frame Pointer**: Foo's caller is main so this is garbage	
xEFFC	x3007	**Foo's Return Address**: Points to instruction in main to return to	
xEFFD	?	**Foo's Return Value**: currently unknown	
xEFFE	5	**Foo's First Param**: a	
xEFFF	3	**Foo's Last Param**: b	
xF000	?	**Empty** (main's frame)	

Caller Setup (Transition 1)

Before calling JSR BAR, foo must initiate the creation of bar's stack frame by giving bar its params and moving the stack pointer (R6) accordingly. That's it.

Tasks

Tasks directly translate to assembly instructions

Move stack pointer up enough spaces to make room for params.
Store params on stack.
JSR to BAR.

State 1 - Foo to Bar handoff

```
int foo (int a, int b) {
        int local_var = a + b;
        answer = bar(a, local_var);
        return answer;
}
int bar (int c, int d) {
        int var1 = c + d;
        int var2 = c - d;
        answer = var1 / var2;
        return answer;
}
```

PC before JSR
R7 after JSR

PC after JSR

This is the snapshot of the stack right as JSR is called. Notice the callee's (bar's) frame has been started but is incomplete. Think of this stage as the handoff between caller and callee. Immediately after JSR is called, R7 has the value of the return address and our program is now executing in bar.

	Address	Value	Description	
Callee's (bar's) stack frame	xEFF7	5	**Callee's First param:** c	← R6
	xEFF8	8	**Callee's Last param:** d	
Caller's (foo's) stack frame	xEFF9	?	**Caller's Saved Register(s):** Foo saved any registers before use	
	xEFFA	8	**Caller's Local Var(s)**	← R5
	xEFFB	?	**Caller's Old Frame Pointer**: Foo's caller is main so this is garbage	
	xEFFC	x3007	**Caller's Return Address**: Points to main	
	xEFFD	?	**Caller's Return Value**: currently unknown	
	xEFFE	5	**Caller's First Param**: a	
	xEFFF	3	**Caller's Last Param**: b	
	xF000	?	**Empty** (main's frame)	

Callee Setup (Transition 2)

As soon as our new function (bar) is called, we have to finish creating the stack frame and store our important values: return address (R7) and old frame pointer (R5). You should use R6 as your reference point.

Tasks

Move stack pointer up to make room for everything.
Store current value of R5 (caller's frame pointer) onto the stack.
Store the return address (currently stored in R7) onto the stack.
Update R5 to point to one above the position of the old frame pointer in the current frame.

Callee Setup cont (Transition 2.1)

We must now make room for local variables and store the values of any registers you plan on using on the stack. They will be restored later, immediately prior to returning to the caller (foo).

Tasks

Move stack pointer up enough to make room for locals and saved registers.
Store registers you plan to use onto stack.
Store value of local variables whenever they are calculated (not part of this transition)

State 2 - During Bar's Execution

```
int foo (int a, int b) {
        int local_var = a + b;
        answer = bar(a, local_var);
        return answer;
}

int bar (int c, int d) {
        int var1 = c + d;
        int var2 = c - d;
        answer = var1 / var2;
        return answer;
}
```

Return address

PC

This is the snapshot of the stack once bar has finished setting up its stack frame. It may now execute all of its logic freely, making sure to only use registers R0-R4 if they have been saved on the stack (in this example just R0 and R1). Notice that this state is similar to State 0, except it is for bar instead of foo. Also note that if bar decided to call a function (eg "baz"), bar would become the caller in that relationship and baz would be the callee.

	Address	Value	Description	
Callee's (bar's) stack frame	xEFF0	ex: 6	**Bar's Saved Register:** R0	R6
	xEFF1	ex: 7	**Bar's Saved Register:** R1	
	xEFF2	13	**Bar's Last Local Var**: stores value of var2	
	xEFF3	-3	**Bar's First Local Var**: stores value of var1	R5
	xEFF4	**x5FFA**	**Bar's Old Frame Pointer**: points to Foo's old FP/R5	
	xEFF5	**See Left**	**Bar's Return Address**: Points to instruction in foo to return to	
	xEFF6	?	**Bar's Return Value**: currently unknown	
	xEFF7	5	**Bar's First Param**: c	
	xEFF8	8	**Bar's Last Param**: d	
Caller's (foo's) stack frame	xEFF9	?	**Caller's Saved Register(s):** Foo saved any registers before use	
	xEFFA	8	**Caller's Local Var(s)**: int local_var	
	xEFFB	?	**Caller's Old Frame Pointer**: Foo's caller is main so this is garbage	
	xEFFC	x3007	**Caller's Return Address**: Points to main	
	...	...	...	

Callee Teardown (Transition 3.0)

Once we have our return value and are ready to return to the caller, we must first store the return value on the stack, then restore any registers we used. You should use R5 as your reference point

Tasks

Store return value onto stack
Restore any registers (R0-R4) you used from stack. In bar's case we restore R0 and R1

Callee Teardown (Transition 3)

Now for the cookie cutter instructions: Restore R5 (caller's frame pointer), restore R7 (our return address into the caller), pop down R6 to point to the return value (making it easy for the caller to find it), and return.

Tasks

Restore R5 (frame pointer) from the stack
Restore R7 (return address) from the stack
Pop down R6 (stack pointer) to point to the return value
Return

State 3 - Bar hands back to Foo

```
int foo (int a, int b) {
        int local_var = a + b;
        answer = bar(a, local_var);
        return answer;
}
int bar (int c, int d) {
        int var1 = c + d;
        int var2 = c - d;
        answer = var1 / var2;
        return answer;
}
```

R7 / PC after RET

PC during RET

Once bar is finished running and properly torn its stack down to the return value, we are ready to return. Upon calling "RET", the stack looks as it does on the right, the PC is updated with the return value (R7), and the caller, foo, picks it up from there. Bar points R6 at the return value so that foo can load the value immediately using R6.

	Address	Value	Description	
Callee's (bar's) stack frame	xEFF6	-4	**Callee's Return Value**	← R6
	xEFF7	5	**Callee's First param:** c	
	xEFF8	8	**Callee's Last param:** d	
Caller's (foo's) stack frame	xEFF9	?	**Caller's Saved Register(s):** Foo saved any registers before use	
	xEFFA	8	**Caller's Local Var(s)**	← R5
	xEFFB	?	**Caller's Old Frame Pointer**: Foo's caller is main so this is garbage	
	xEFFC	x3007	**Caller's Return Address**: Points to main	
	xEFFD	?	**Caller's Return Value**: currently unknown	
	xEFFE	5	**Caller's First Param**: a	
	xEFFF	3	**Caller's Last Param**: b	
	xF000	?	**Empty** (main's frame)	

Caller Accepts Return (Transition 4)

All we have to do is load the return value into a register so we can work with it, and then pop the stack pointer back down to the top of our stack.

Tasks

Load return value into a register we want it in
Pop down R6 to top of caller's frame

State 4/0 - During Foo's Execution

```
int foo (int a, int b) {
        int local_var = a + b;
        answer = bar(a, local_var);
        return answer;
}
```

PC

We are now back to where we started! Woo, what a lot of work. Just remember that once you figure out the instructions for transitions 2-4, they can be used as cookie cutters for every function call you ever make. Transitions 1, 2.1, and 3.0 will vary depending on the function (number of params, locals, and regs used). Note: If we wanted to continue, returning out of foo and back to main would look just like returning from bar to foo. Just treat foo as the callee of main.

Address	Value	Description	
xEFF9	?	**Foo's Saved Register(s)**: Foo saved all registers it used before use	← R6
xEFFA	8	**Foo's Local Var(s)**: stores value of local_var	← R5
xEFFB	?	**Foo's Old Frame Pointer**: Foo's caller is main so this is garbage	
xEFFC	x3007	**Foo's Return Address**: Points to instruction in main to return to	
xEFFD	?	**Foo's Return Value**: currently unknown	
xEFFE	5	**Foo's First Param**: a	
xEFFF	3	**Foo's Last Param**: b	
xF000	?	**Empty** (main's frame)	

3 Rules and Regulations

3.1 General Rules

1. Starting with the assembly homeworks, Any code you write (if any) must be clearly commented and the comments must be meaningful. You should comment your code in terms of the algorithm you are implementing we all know what the line of code does.

2. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. This means that (in the case of demos) you should come prepared to explain to the TA how any piece of code you submitted works, even if you copied it from the book or read about it on the internet.

3. Please read the assignment in its entirety before asking questions.

4. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.

5. If you find any problems with the assignment it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

3.2 Submission Conventions

1. All files you submit for assignments in this course should have your name at the top of the file as a comment for any source code file, and somewhere in the file, near the top, for other files unless otherwise noted.

2. When preparing your submission you may either submit the files individually to T-Square or you may submit an archive (zip or tar.gz only please) of the files (preferred). You can create an archive by right clicking on files and selecting the appropriate compress option on your system.

3. If you choose to submit an archive please don't zip up a folder with the files, only submit an archive of the files we want (see Deliverables).

4. Do not submit compiled files that is .class files for Java code and .o files for C code. Only submit the files we ask for in the assignment.

5. Do not submit links to files. We will not grade assignments submitted this way as it is easy to change the files after the submission period ends.

3.3 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.

2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via T-Square. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over T-Square.

3. There is a 6-hour grace period added to all assignments. You may submit your assignment without penalty up until 11:55PM, or with 25% penalty up until 5:55AM. So what you should take from this is not to start assignments on the last day and plan to submit right at 11:54AM. You alone are responsible for submitting your homework before the grace period begins or ends; neither T-Square, nor your flaky internet are to blame if you are unable to submit because you banked on your computer working up until 11:54PM. The penalty for submitting during the grace period (25%) or after (no credit) is non-negotiable.

3.4 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work.

Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to find evidence of unauthorized collaboration.

What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment. Submissions that are essentially identical will receive a zero and will be sent to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply e-mailing it to them so they can look at it. If you supply an electronic copy of your homework to another student and they are charged with copying, you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code such as but not limited to public repositories (Github), pastebin, etc. If you would like to use version control, use github.gatech.edu

3.5 Is collaboration allowed?

Collaboration is allowed on a high level, meaning that you may discuss design points and concepts relevant to the homework with your peers, as well as help each other debug code. What you shouldn't be doing, however, is paired programming where you collaborate with each other on a low level. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, and it is often the case that the recipient will simply modify the code and submit it as their own.

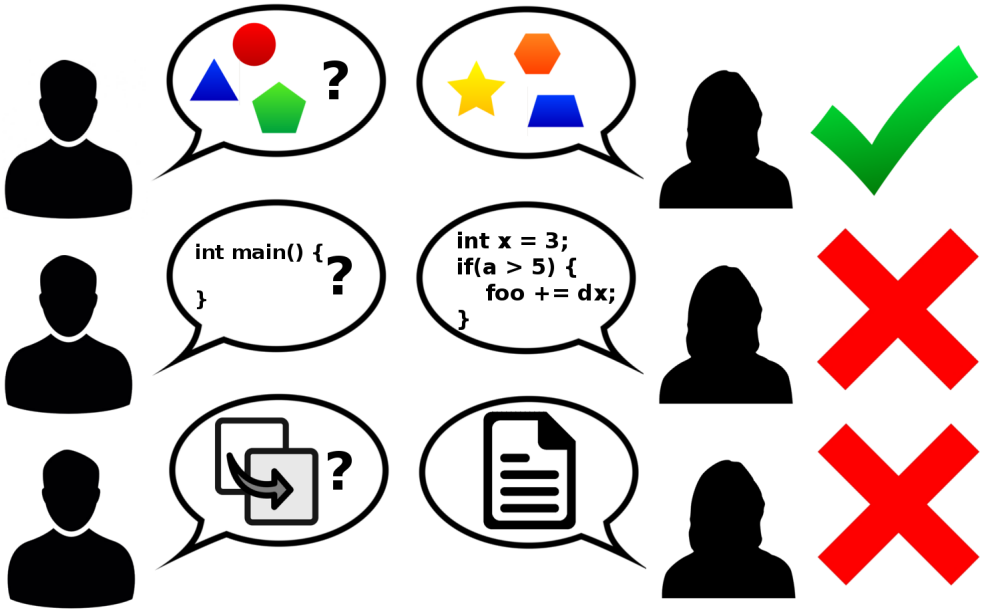


Figure 1: Collaboration rules, explained