Project Checkpoint 5 Specification

*Many thanks to Dr. Prateek Bhakta, who designed the original version of this project. Credit for anything that works belongs to him; blame for any mistakes belongs to Dr. Dollak.*

In the final part of your project, you will write a very simplified kernel with device drivers to interact with I/O devices. Your main assembly program can use these drivers through syscalls. This mimics real world I/O, which is handled by the kernel/OS in many situations.

# Kernel & Syscalls

You will be implementing syscall codes 1,5,9,10,11,12 in a new assembly file called kernel.asm. The code in kernel.asm will need to be included in every program you write that involves a syscall. It must be the first file assembled. So your assembler should be called like:  
./assemble kernel.asm myfile.asm (myprojectfile2.asm ...) static.bin inst.bin

A syscall is essentially a program-thrown interrupt that is used by the program to ask the OS to handle a task that the program can’t do directly. These tasks include I/O, ending the program, allocating memory on the heap, etc. The program specifies which syscall code you want to run by putting the code in register $v0, any arguments in $aX registers, and calling syscall. After the OS handles the interrupt, control reverts back to the program.

Unlike real MIPS, the syscall in your circuit will do the equivalent of a jalr $0, $k0 (it should already by doing this), which means that the value of is stored in register $k0 and the program will jump to line $0. The file kernel.asm will be assembled BEFORE any program that uses syscall, so line $0 will jump execution to the first line of the kernel.asm file.

Your kernel.asm code will have labels like "Syscall0:" and "Syscall10:", etc. that store the code for each syscall subfunction. Starting at line 0, you will have a switch-like statement, where you branch to the appropriate label depending on the value of $v0.

We can end a syscall by performing a jr $k0 instruction to return back to the calling instruction. Note this doesn’t work quite like real world MIPS, as there is a special EPC register that serves the purpose of $k0.

Unlike ordinary function calls, a syscall isn’t allowed to modify any registers other than the two kernel registers . This means that **any registers that you use other than these, even temporary registers, need to be saved on the stack and restored at the end**.

## CPU Changes from Checkpoint 4

None.

## Kernel

### Initialization: “Syscall 0”

If the value of is , which is not a valid syscall, this means that we are in the "boot up" phase of your computer after a reset. This is where we can initialize anything that we need to do before the main part of the computer runs. This will only be run once, at the very start of any program. None of your programs should expressly call syscall 0.

In syscall 0, you need to set the initial value of the stack pointer to 0x03FFF000 / 0xFFFFF000 / -4096. You will also want to set up an initial heap pointer for your syscall 9 implementation (see heap pointer below). Finally, add a label at the end of your syscall.asm file with a name like "\_\_SYSCALL\_EndOfFile\_\_", and jump there at the end of your syscall 0. This will begin the execution of your actual program.

### End Program: Syscall 10

In the real world, the OS would end the process, removing it from the scheduler and freeing any memory both in main and in the cache associated with this process.

For your project, just loop forever.

### Print Character: Syscall 11

Use the appropriate lw/sw to the appropriate addresses to communicate with the terminal registers and print the character that is stored in $a0, then return.

### Read Character: Syscall 12

Use the appropriate lw/sw to the appropriate addresses to communicate with the keyboard registers and read a character, if there is one. If not, loop until there is. Return the character in register $v0.

### Heap Allocation: Syscall 9

Here, the calling program will request a number of bytes in register $a0, and you will provide a block of that many bytes, returning a pointer in $v0. No two separate syscalls should return overlapping memory, and you **do not need to worry about releasing memory**. For us, the number of bytes requested will always be a multiple of 4.

A "heap pointer" solution would work perfectly fine for this class if you don’t need to reallocate memory, but real-world memory allocation is done with the help of more sophisticated data structures, so that memory addresses that are no longer in use can be reclaimed and reused for future objects. Unlike the stack pointer, you should store your heap pointer in memory rather than a register.

The heap pointer should start where static memory ends. To figure out where static memory ends, you will need some help from your assembler. When processing static memory in your assembler, add an extra label to your static memory label map with a special name like "\_END\_OF\_STATIC\_MEMORY\_". Then an instruction like "la $k1, \_END\_OF\_STATIC\_MEMORY\_" inside your kernel will be able to correctly put the address of the end of static memory / beginning of heap memory into a temp register. You can then \*STORE\* this pointer in a special place in memory reserved for the OS - anything between 0x3FFF000 and 0x3FFFFEFC. This is much more address space than you will need, but maybe you have other kernel related challenges you plan to implement.

### Print Integer: Syscall 1

Use the appropriate lw/sw to the appropriate addresses to communicate with the terminal register and print the integer that is stored in $a0, then return. You may NOT assume that the integer is positive.

### Read Integer: Syscall 5

Use the appropriate lw/sw to the appropriate addresses to communicate with the keyboard registers and read an integer that is stored into $v0, then return. You may assume that the integer ends when the next character is a non-digit. (Leave that character on the keyboard queue). You may NOT assume that the integer is positive. You may assume that the integer is smaller than 32 bytes.

You will be given a test file that will test all these syscalls.

# Program

Now that you have a working monitor and keyboard, write a program in assembly that uses these syscalls and does something interesting.

For 0 stars, write a simple calculator that will read arithmetic like "34 + 91" or "" or "", where "\_" is a special character that means the result of the previous calculation. You may assume that all expressions only have two terms. Hint: Figure this out in C++ first, using cin with ints or chars to do the equivalent of syscall 5,11.

As you will see in the next section, you can replace this program with a different one to earn 2-15 stars, depending on the difficulty of the idea.

# Challenges

**Challenge Requirements for Project Checkpoint 5:**

Teams must complete at least 20 stars' worth of challenges for this checkpoint.

**Available Challenges for Project Checkpoint 5:**

You can earn a maximum of 191 stars on this assignment.

For 2-15 stars, submit a more interesting program INSTEAD of the basic calculator. Feel free to come up with any program idea at all and discuss it with me to find out how many stars it is worth. Alternatively, pick one of the following options:

* A board game like tic tac toe or connect four, where the game board is displayed on the monitor or terminal, and moves are entered on the keyboard, and the computer prints out a message when a player has won. (8 stars)
* A more complex python-like calculator that will also read arithmetic like "34 + (91\*(4 -2))" or "" or "", where "\_" is a special character that means the result of the previous calculation. You may assume that all expressions are fully parenthesized so that you don’t have to worry about order of operation rules. (7 stars)
* A game of pong or break out (or other classic video games - verify with me). For a "real time" game, we need a game that only changes a pixel or two per time step because logisim is so much slower than a real CPU. (15 stars)

Handle strings: If you have not already implemented handling .asciiz strings in your assembler, do that; the details are in the Project1 specification, and this is still worth 5 stars. You get another 5 stars if you also implement syscall 4 to print out a string, and another 5 for implementing syscall 8 to read a string from the keyboard (until a newline is pressed). Typically, the characters are stored in a buffer as they are being entered, but you can store them on the stack until a newline is pressed. Then copy the contents of the string into newly allocated heap memory for your string. Remember to end your string with a 0.

Across checkpoint 4 and checkpoint 5, you can earn 5 stars *each* for attaching sound, joysticks, LED panels, or any other widget from the Input/Output menu of Logisim to the bus with an appropriate controller and appropriate assembly device driver. In part 4, you write the controller to attach the device (3 stars), and in part 5, you write a device driver in assembly to communicate with that device (2 stars). You can earn a maximum of 40 stars this way across the two checkpoints (24 stars from checkpoint 4 and 16 here).

You get 5 stars to draw a picture. Extract the RGB values from a 256 x 256 pixel image and store them in a static memory file. Then display them one by one on your monitor. The image will take a while to display.

You get 15 stars for implementing a "kernel" mode, where the OS will block memory access to I/O registers and OS memory unless the program is in "kernel" mode. Because we don’t have a set range of PC addresses that correspond to OS instructions like the real world, you will have to engineer something else. Perhaps you can have a special flip flop somethere that is set when a syscall begins and is reset when a jr $k0 occurs. This should be done entirely in Logisim.

You get 20 stars for replacing the RGBController with a "graphics card" that can print lines or rectangles on command with the CPU. Decide on a protocol / address space for this. A single graphics card syscall should contain the information needed to draw a rectangle. The GPU will then take multiple cycles to display the pixels of the rectangle but will do so much faster than the CPU would. Once set up, the "GPU" should draw one pixel per cycle in the rectangle instead of one pixel per 5-8 cycles like the CPU. The GPU will need 3-5 internal registers, depending on your implementation. The GPU should be able to print rectangles while the CPU does other things - the CPU will have to poll the GPU status register to see if the GPU is ready for a new command. This will be made much easier with the help of a dedicated 16kb VRAM (video memory) that stores all the pixels on the screen, so that the screen and the VRAM can update asynchronously. See me for details.

You get 10 stars for implementing basic exception handling. When a run-time error occurs (divide by 0 / null pointer exception / etc.), your circuit should simulate a syscall with a special error code (change PC to 0, and put a special error value in $v0, ignore $k0). Then write an error handler in your kernel that will print some kind of error message and terminate the program. A null pointer is (in C) a pointer with address 0. Because this is a valid address in our non-standard version of Logisim, you will need to use the 0 address block in your kernel for static memory so that your program would never need to access address 0. See me for details.

You get another 10 stars for implementing more advanced exception handling (try-catch). The OS uses some static memory to store the address of various exception handler functions for various errors. When a try-catch block happens, the program "registers" a new exception handler (the catch block) with the OS at the beginning of the try block, and then "unregisters" that exception handler with the OS at the end of the catch block. When an error occurs, the OS runs the error handler that is currently registered in its static memory. See me for details.

You get 15 stars for adding a syscall that will let you deallocate memory. The programmer should provide a pointer to an object together with the number of bytes to deallocate. Rather than a modern sophisticated solution with complex data structures, a simple solution that would be reasonable to implement in assembly could work like this: In addition to the heap pointer, store ranges of deallocated memory, and then occasionally "defragment" using a sequence of memory copies if the total amount of deallocated memory is ever more than half of all allocated memory. This way, your program is never using more than twice the memory it needs, and you only need to occasionally go through the somewhat expensive defragmenting process. You do not need to verify that the memory that you are deallocating was actually ever allocated. See me for details.

You get 15 stars for implementing a 64-bit integer class with your 32-bit CPU. See me for details.

You get 15 stars for implementing a speaker that plays 8-bit music. You will need to use 3-5 buzzers with different waveforms (square, triangle, etc.) and design a controller for the whole group. Implement a syscall to interface with it. Write an assembly program that plays a song. +5 stars if your song is a classic videogame theme (Super Mario Bros., Zelda, etc.). See me for details.

You get up to 40 stars for implementing the classic 5-stage pipeline. You need to make a change to your processor to add intermediate registers between stages of the pipeline, as well as changes to your assembler to add no-op instructions where needed to your pipeline.

If you do this, you must submit BOTH a single-cycle processor for your main project grade, and a pipelined processor for stars. Your assembler would also have to detect these hazards, so you would need to submit a modified assembler program.

* Base: 10 stars for implementing just the pipelined processor where your solution to hazards is to have the **assembler** add no-op instructions between hazards.
* +10 additional stars if you add data forwarding out of the ALU and MEM
* +10 additional stars if you “flush” the pipeline only a bad branch guess as opposed to using no-ops to deal with branches.
* +10 additional stars if you add any kind of non-trivial branch prediction.

??? stars: An enhancement you invent and discuss with me ahead of time.

# What to Submit

Kernel.asm

Test cases for all of the challenges you implemented.

If you have added any challenges that involve modifying your .circ file, be sure to submit it..

# Deadlines

This checkpoint is due at **11:59 pm on December 4**.

You may continue to add challenge stars for any checkpoint until **11:59 PM on December 8**. If you are trying for the half letter-grade increase in your course grade for completing 100 stars beyond the requirement, submit a table listing the stars you have completed.

# Checkpoint Interview

As always, be sure to schedule an interview to show off your project and testcases. Getting credit for the checkpoint requires showing me (a) that it works, and (b) that you understand how it works.

You may schedule an interview on my calendar as you have for previous checkpoints. This gives you some leeway to revise and reinterview if needed.

The exam period for this class (9 am – 12 pm 12/9 for the morning section of the class, 9 am – 12 pm 12/10 for the afternoon section) is reserved for final project interviews. I will randomly assign time blocks for each team during your section’s exam period. If you have already completed the interview before the exam period, you do not need to attend.

Emergency backup interview: If your “final” interview does not go according to plan, you may sign up for ONE retry interview on Wednesday, 12/11, Thursday, 12/12, or Friday, 12/13. Schedule an in-person appointment using my Calendly, <https://calendly.com/finegan-dollak>; I will have special additional hours for these three days.