Project Part 5

In the final part of your project, you will write simple *device drivers* in assembly that your main assembly program can use through syscalls. This mimics real world I/O, which is handled by the kernel/OS in many situations. Your assignment is due on midnight before the day of the final Interview, which will take place during the final exam period - 2pm-5pm on Tuesday Dec 12. All bonus components to the project must be completed when submitting.

1 Syscall

We will write a very simplistic kernel with device drivers to interact with I/O devices. Specifically, 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, and must be the first file assembled. So your assembler should be called like:

./assemble kernel.asm myprojectfile.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 $PC + 4$ 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 $k0, k1$. 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.

1.1 CPU Changes from Project part 4

None.

1.2 Kernel Syscalls

1.2.1 Initialization: Syscall 0

If the value of $v0$ is 0, 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.

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. 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 function.

1.2.2 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 logisim project, just loop forever.

1.2.3 Print Character: Syscall 11

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

1.2.4 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.

1.2.5 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.

As mentioned in class, 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.

1.2.6 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 assume that the integer is positive.

1.2.7 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 assume that the integer is positive and smaller than 32 bytes.

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

2 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 bonus points, write a simple calculator that will read arithmetic like "34 + 91" or "1234−103" or "_/4", where "_" is a special character that means the result of the previous calculation. You may assume that all expressions only have two terms. Hint: As always, 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 project with a different one to earn 5-50 bonus points, depending on the difficulty of the idea.

3 Bonus

There will be about 300+ bonus points available on this project in total. The maximum bonus point grade bump you can get is 10% of your *total class grade*, not just your project grade. The maximum 10% bump is earned at 200 total bonus points above the mandatory $50(n-1)$ points required for working in a group of size n.

Feel free to come up with your own ideas for bonus points, and let me know so I can decide how many points you would get *before* doing that task.

One way to get bonus points is to do a more interesting program INSTEAD of the basic calculator. Feel free to come up with any program idea at all. You would earn 5-50 points from this task.

Program ideas:

- A board game like tic tac toe or connect four, where the game board is displayed on the monitor, and moves are entered on the keyboard, and the computer prints out a message when a player has won. (25 points)
- A more complex python-like calculator that will also read arithmetic like "34 + (91*(4 -2))" or "$(1234 - 103) * (34 - 23)$" or "$(\_ - 4) * \_$", 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. (35 points)
- 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. (25 points)

Other bonus points involve the assembler / CPU / Kernel. For any of these bonus tasks, you MUST also write a small assembly program where you test it. Submit your bonus point tests with your final project.

- You get 1 point each for implementing the following MIPS pseuodinstructions by translating them into the above "real" instructions first: mov, li
- You get 2 points each for implementing the following MIPS pseuodinstructions by translating them into the above "real" instructions first: sge, sgt, sle, seq, sne bge, bgt, ble, blt, abs. (Hint: adding new labels is a pain. Try to figure out a way that doesn't require new labels).
- You get 5 points for handling immediates that are larger than 16 bits, but smaller than 32 bits.

  The above psuedoinstructions do NOT have a MIPS encoding. You must translate them into one or more of the "standard" MIPS instructions first, potentially making use of the $at register as needed.
- You get up to 10 points for implementing support for strings.
  - 10 points for static memory strings. These appear with the tag .asciiz.
    For example,

```
.data
  .asciiz "Hello, world."
```

Although a real world character string needs only 1 byte per character, you should use 4 bytes per character in your output with the rest of the bits being 0.

Simply storing a string correctly in static memory is 10 points.

- 10 points for implementing syscall 4 to print out a string
- 10 points 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).

- You get 1 point each for implementing the following real MIPS instructions in your assembler:

  and,or,nor,xor,andi,ori,xori,lui

  You can get an additional 2 points each for adding these instructions functionality to your ALU and control.

- You get 1 point each for implementing clo (count leading ones) and clz (count leading zeroes) in your assembler. You get an additional 6 points each for implementing them in your ALU and control. Since these functions are not built in to Logisim, you will have to design custom circuits with appropriate use of gates/multiplexors/etc. to implement these two functions.

- You get 15 points *each* for attaching sound, joysticks, LED panels, or any other widget to the bus with an appropriate controller (part 5) and appropriate assembly device driver (part 5). This will affect both parts 4 and 5 of your project. In part 4, you write the controller to attach the device (10 points), and in part 5, you write a device driver in assembly to communicate with that device (5 points).

- You get a variable number of points (up to 100) 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 noop 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 bonus points. Your assembler would also have to detect these hazards.

  - Base: 25 points for implementing just the pipelined processor where your solution to hazards is to have the assembler add noop instructions between hazards.
  - +25 additional points if you add data forwarding out of the ALU and MEM
  - +25 additional points if you "flush" the pipeline only on a bad branch guess as opposed to using noops to deal with branches.
  - +25 additional points if you add any kind of non-trivial branch prediction.

- You get 20 points 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 10 points 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 50 points for replacing the RGBController with a "graphics card" that can print 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. A 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. See me for details.

- You get 25 points 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 50 points 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 50 points for adding a syscall that will let you deallocate memory. The programmer should provide a pointer to an object together with a 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 actually 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 50 points for implementing a 64 bit integer class with your 32 bit CPU. See more for details.

- You get a variable number of points for doing anything else cool. Just clear it with me first, so we can figure out how many points it is worth in advance.

Project Part 5

Submit all source files to blackboard. Only one person per team needs to submit the source files.

0. (0 points) (NOT OPTIONAL) Who did you work with on this project? What sources did you consult? How long did this assignment take?

> **Solution:** I worked with Tenzin on this project. I consulted Dr. Dollak and Dr. Bhakta for advice and the notes on blackboard. I used online sources to figure out ascii code.

1. (0 points) (NOT OPTIONAL) How did you divide the work?

> **Solution:** We did not do a definite divide-and-conquer approach. We worked on each part of our project separately and submitted whichever one yielded the best result, sometimes that would be a combination of both of our projects.

2. (0 points) (NOT OPTIONAL) Which of the extra credit options did you implement? Provide a list here.

> **Solution:**
>
> Total Bonus = 10 + 105 + 25 + 35 + _ = 175+ points
>
> - asciiz (storing static memory strings): 10
> - Widgets (each with its own controller and device driver): 7 X 15 = 105
>   - LED: turns red and stays red until given instruction to turn off, incorporates register
>   - RGBLED: blinks red, green, or blue when instructed to, does not incorporate register but makes use of the keyboard (in the test code) as the controller decides the color of the blinking light when instructed to using ASCII values of 'R', 'G' and 'B'
>   - Joystick: given specific instruction stores value of x or y coordinate of the joystick
>   - Button: the controller detects if the button is pressed or not
>   - Switch: the controller has a mechanism to provide input and output for the switch.
>   - Buzzer: the controller rings the buzzer based on the frequency, volume and duty cycle values (all stored in register)
>   - Slider: returns the value of slider when instructed to. (In the test code, it is connected to the buzzer to change its volume)
> - Complex calculator: 35
>
>   Calculates results of complex equations like "34 + (91*(4 -2))" or "(1234−103)∗(34−23)" or "(_ − 4) ∗ _", where "_" is a special character that means the result of the previous calculation. The default value of "_" is 0. It is important to make sure that there is no additional bracket than required for the calculation and to end the expression by pressing enter.

- 5 Stage Pipeline (Most basic version): 25

  Its part includes:

  - Circuit: Consists of additional registers from regular CPU to enable 5-stage calculation. Certain changes in the Control Unit and the CPU to synchronize information parsing. It is not integrated with the rest of the system and is a stand-alone project.
  - Assembler (project1.cpp, project1.h): The assembler adds additional noops wherever appropriate (for instance two noops after beq and bne). It calculates number of dependent registers and if they were changed recently to add another necessary amount of noops. The assembler also modifies the labels to jump for beq and bne to fit the new system (2 additional instruction lines to consider.
  - test.asm: The assembler that tests almost all of the instructions to ensure proper functioning of the code whilst maintaining hazard control.

- Miscellaneous: To Be Determined

  - Clear Screen: Extra functionality on Terminal to clear screen. Incorporated in kernel as syscall 13.
  - Print Negative Number: Syscall 1 can print negative integers too.