HackSim Design Document

HackSim is a deep hardware simulator of my implementation of the Hack computer from Nand2Tetris, an open-source educational project by Shimon Schocken and Noam Nisan. In the Nand2Tetris course, students learn computer organization by designing logic gates, adders, memory, and more using only three primitive components: `Nand`, `DFF` (D flip-flop), and a clock. Students supply their implementations of hardware components in an HDL-like language, and Nand2Tetris provides an open-source Java-based hardware simulator that simulates and tests the circuits implemented in HDL.

Why Build Another Hardware Simulator?

The advantage of Nand2Tetris's hardware simulator is that it can accept any valid hardware configuration as input. The trade-off is that such a simulator is hard to optimize and is overwhelmed by the thousands of components in, say, a 4K RAM module. The authors of the Nand2Tetris hardware simulator subvert this problem by providing built-in simulations for large components like RAM modules. This modular design allows the hardware designer, while enjoying a significant simulation speedup, to test components in isolation.

Indeed, this is a practical solution, but a part of me really wanted to see all of the cogs I designed come together to create a living computer. I wondered if it was possible perform a deep simulation of *all* of my hardware, all the way down to the NAND gates and flip-flops. It turns out that, with the help of a single optimization[1], it was possible (see

Optimizations). This bizarre obsession of mine culminated in this C++ project, entitled *HackSim*.

HackSim Hardware Simulator

This section describes the design of the Hack hardware simulator.

Combinational Circuits

In the scope of this project, a combinational circuit is a circuit whose outputs are the result of applying a Boolean function to the inputs. Formally, $\text{out}_t = f(\text{in}_t)$, where t is the current time in clock cycles, and f is a Boolean function mapping inputs to outputs. This definition implies that outputs are computed "instantaneously" from inputs. In reality, this is not necessarily the case, as evidenced by circuits that take advantage of delay-based electrical properties of the underlying components. These types of circuits (such as latches) often do this by using outputs as inputs, creating a feedback loop. Logic gates may even have feedback loops in their internal implementation. In this project, circuits that contain these types of loops are not considered to be pure combinational circuits. The `Nand` gate is treated as a "black box" in this regard.

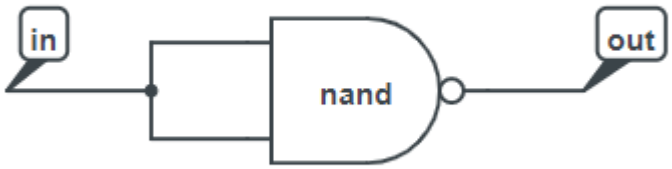
Implementation of Combinational Circuits

Each combinational circuit in `Chips.h` is represented by a class that has a member function called `computeOutput()`, which sets the circuit's outputs depending upon the current values of the circuit's inputs. All combinational circuits (except the two most primitive: `Nand` and `Nand16`) model their natural design in hardware by containing other combinational circuits as submodules. The `computeOutput()` member function of each submodule is invoked as necessary.

There are two combinational circuits whose `computeOutput()` functions call no other `computeOutput()` functions. These are the `Nand` and `Nand16`[2] circuits. In the spirit of the Nand2Tetris course, these circuits are to be understood as "primitive" or "given". At the lowest level, all combinational circuits rely on the outputs of `Nand` and `Nand16` circuits.

Example: `Not` Implementation

The `Not` circuit has one input pin named `in` and one output pin named `out`. Below is the implementation of the `Not` circuit using a `Nand` gate.



To implement this diagram, the `Not` class contains a member function `set_in(bool)` that allows the caller to set the input pin. There is no public function to set the output directly; rather, the `computeOutput()` function does this as a side-effect. Here is the implementation of the `computeOutput()` function:

```
void Not::computeOutput() {
  m_nand.set_a(in());
  m_nand.set_b(in());
  m_nand.computeOutput();
  set_out(m_nand.out());
}
```

The `Not` class contains a `Nand`-type member variable `m_nand`, which contains two input pins (`a` and `b`) and one output pin (`out`). Calling `m_nand.computeOutput()` has the effect of setting `m_nand`'s output pin, accessible via `m_nand.out()`. The `Not` class exposes its output via the `bool Not::out()` member function. Even the most complicated combinational circuits (e.g., `ALU`) follow this fundamental convention. That is `computeOutput()` sets the values of all output pins based on the values of the input pins before the call.

Sequential Circuits

Consistent with Nand2Tetris, this project defines sequential circuits as circuits that have at least one output $\text{out}_t[i]$ that depends on in_{t-1}. In other words, at least one output depends on the value of one or more inputs from the previous time-step. A circuit that uses at least one sequential circuit internally is considered to be implicitly sequential.

To avoid dealing with the intricacies of electronics, the Nand2Tetris course accepts the `DFF` (D flip-flop) circuit as the only primitive sequential circuit. All sequential circuits eventually make use of a `DFF`.

`tick()` and `tock()`

An output that depends on an input from the previous time-step is considered to be *clocked*. An input upon which a clocked output depends is likewise considered to be clocked. In a sequential circuit, it is not necessary for all outputs and inputs to be clocked. For example, of the outputs in the `CPU` circuit, only `addressM[15]` and `pc[15]` are clocked; the `outM[16]` and `writeM` outputs are combinational. Because of the heterogeneous nature of output types in clocked circuits, a single `computeOutput()` function is insufficient to handle clocked circuits.

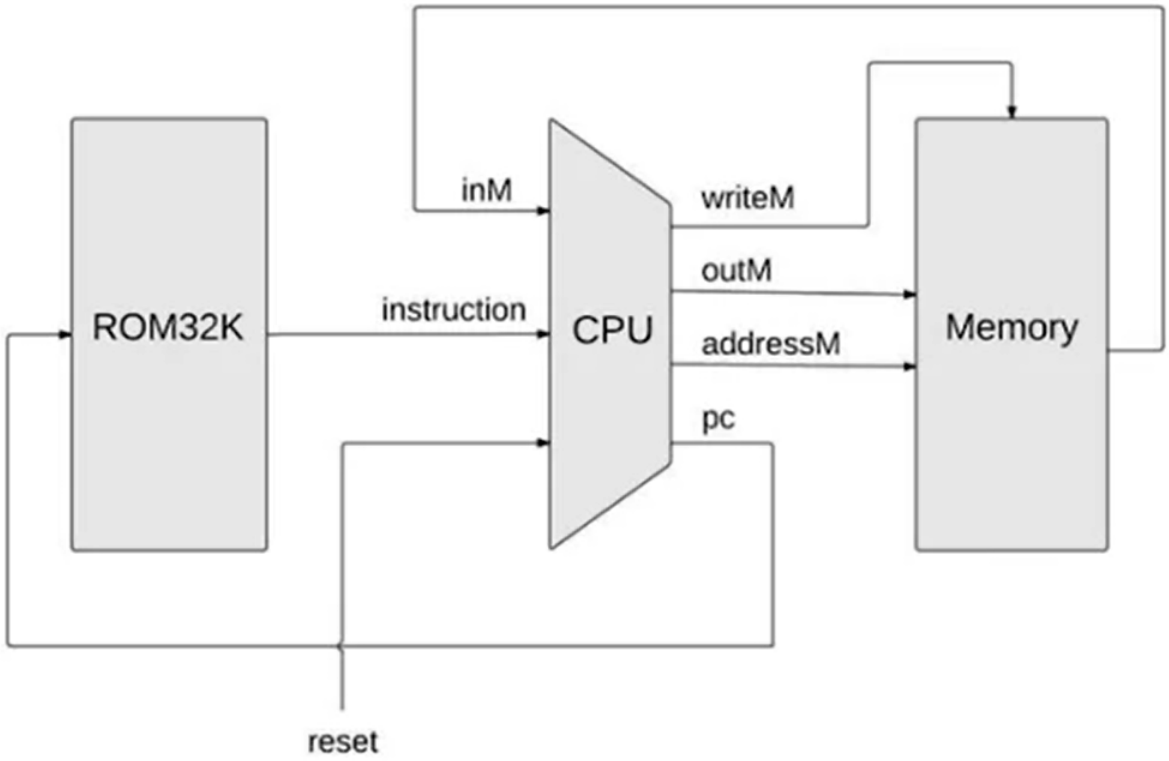
This project solves this problem by splitting each clock cycle into a `tick` phase followed by a `tock` phase. The `tick` phase of the clock cycle represents the period of time when the clock is low, allowing voltages in the system to settle into their intended values. The sequential circuits effect this behavior by implementing a `tick()` function, which allows a sequential circuit to update its internal combinational circuits, set the input pins of (and call `tick()` on) its internal sequential circuits, and set the values of its unclocked outputs (if any). Like the `computeOutput()` function of combinational circuits, the `tick()` function of sequential circuits is *idempotent*.

The `tock` phase of each clock cycle represents the moment when the clock rises from low to high. At this time, all sequential circuits in the system update their clocked outputs simultaneously. The changes in these outputs will be observed in the upcoming `tick` phase by the components that make use of them.

Example: `Computer` Implementation

The necessity of the separation of `tick` and `tock` is most evident in the `Computer` circuit. Below is the Nand2Tetris hardware diagram of the `Computer` circuit, followed by the implementation of `Computer::tick()`.

Hack Computer implementation




```
void Computer::tick() {
  m_rom.set_address(m_cpu.pc());

  m_mem.set_address(m_cpu.addressM());
  m_mem.tick();

  uint16_t memOut = m_mem.out();

  // m_rom is a shallow chip, so it is not necessary to call
  // m_rom.computeOutput() before observing its output instruction
  m_cpu.set_instruction(m_rom.instruction());
  m_cpu.set_reset(reset());
  m_cpu.set_inM(m_mem.out());
  m_cpu.tick();

  // outM and writeM are updated by the CPU tick()
  m_mem.set_in(m_cpu.outM());
  m_mem.set_load(m_cpu.writeM());
  m_mem.tick();

  assert(m_mem.out() == memOut);
  // If this assertion failed, it would mean that we would have to
  // `tick` the CPU again, entering a feedback loop. If the Memory
```

```
  // unit is implemented correctly, then m_mem.out() will be
  // different only if m_mem.set_address() caused the selected
  // memory register to be different before the tick.
}
```

Suppose it is time t. The clocked output `addressM` of the `CPU` was updated at time $t-1$; therefore, it is necessary to set the `address` input of the `Memory` circuit and then call `Memory::tick()`. The output of the `Memory` object is immediately updated with the contents of the selected memory register as a result of calling `Memory::tick()`. If `Memory::tick()` were to be called again at this point, there would be no further changes to the state of the `Memory` chip, as all `tick()` functions are idempotent.

Next, because the `CPU` uses the output of the `Memory` chip as input, we must set the `CPU`'s `inM` input (along with a few other inputs) and then invoke `CPU::tick()`. But because `outM` and `writeM` are unclocked outputs of the `CPU` chip, these changes must be observed by the `Memory` chip, and so `Memory::tick()` is called again!

One might wonder whether `CPU`'s input needs to be updated and `tick`ed again. It turns out that, if `Memory::tick()` is implemented correctly, calling `Memory::tick()` can only potentially update the `Memory`'s output pins if the `address` input has been updated since the last `tick`. But because only the `load` and `in` inputs to the `Memory` are changed between the calls to `Memory::tick()`, it is not necessary to continue simulating the feedback loop. Of course, if `Memory::tick()` does not obey this rule, then there is a bug.

Once the `Computer` has been `tick`ed, it can then be `tock`ed to complete the execution of the current instruction. The implementation of `Computer::tock()` is quite straightforward:

```
void Computer::tock() {
  m_mem.tock();
  m_cpu.tock();
}
```

In theory, the order of the `tock`s is irrelevant. Calling `CPU::tock()` before `Memory::tock()` would have the same effect.

Optimizations

The "spirit" of this project is to simulate as many components as possible to see if the Hack implementation works. That said, there is only one optimization hardcoded into the simulator, but it is a crucial optimization. It turns out that simulating the entire RAM is

pretty demanding, not in terms of space necessarily, but computation. The `RAM16K` module contains 4 `RAM4K` modules. The other RAM modules each contain 8 smaller RAM modules. The result is, of course, 16K registers at the bottom, each with 16 flip-flops and some logical components. That's a lot of function calls for each `tick()` and `tock()`, most of which have no effect on the `Memory`'s output pins.

Knowing the heirarchical structure of memory ahead of time, one can determine that the only RAM modules that need to be simulated are the RAM modules selected by the address bus. This optimization drastically improves performance and reduces the simulation of a RAM with N registers from $O(N)$ to $O(\log N)$. It does feel a bit "cheaty", though. The issue is that, if my hardware implementation of the `load` bit was incorrect such that, say, all RAM registers would be updated regardless of the value of the `load` bit, then this optimization would hide that bug, because unselected RAM modules are not simulated under the optimization. I decided to include this optimization anyway, because the performance improvement is massive, and even under this optimization, RAM is still simulated deeply, down to the flip-flops.

Even with this optimization, though, HackSim would not be able to run programs like `Pong.hack` at a speed that most users would desire. Because it was not too difficult to add, I decided to include an efficient, "shallow" Hack computer simulator whose purpose is to execute Hack programs as quickly as possible. This efficient implementation, which uses high-level simulations of the `Memory` and `CPU` units, can be enabled by setting the optimizations in the `hacksim.conf` file to `true`.

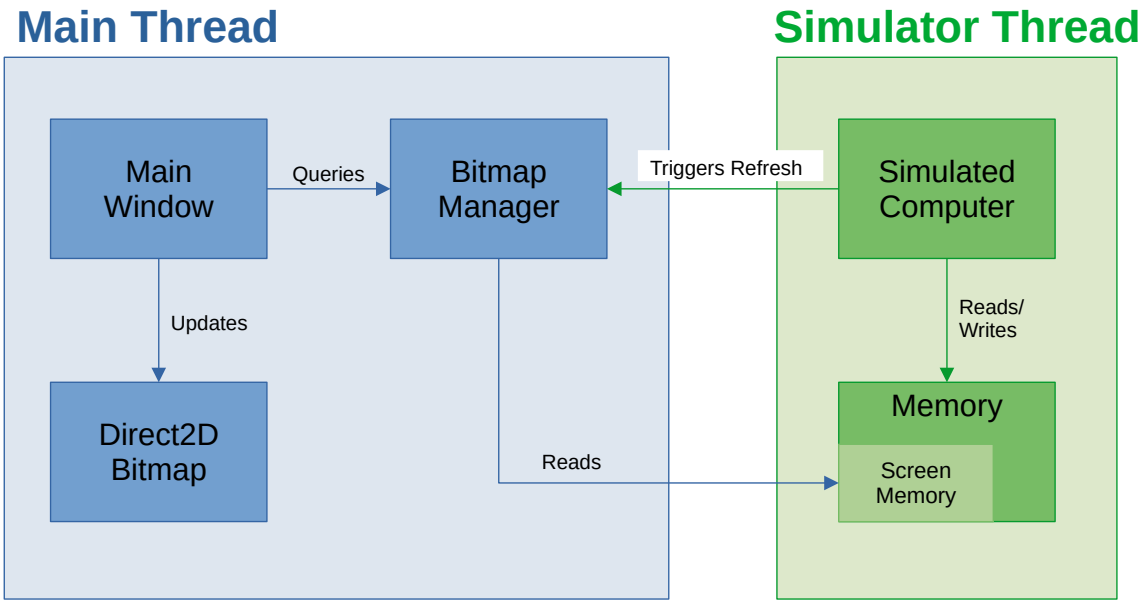
Win32 GUI Wrapper

The simulator library is accompanied by a Win32 application that displays the contents of the simulated computer's screen memory map and allows the user to supply keyboard input to the computer.

Architecture

The GUI application consists of a Main thread that renders the UI and a Simulator thread that runs the hardware simulator. The interaction between the two threads is shown in the diagram below.

HackSim GUI Design



As the Simulator thread runs, it keeps track of the amount of time that has passed since the last frame was rendered. After about 16 ms (which translates to about 60 frames per second), the Simulator thread triggers a repaint by calling `InvalidateRect()` and setting a flag on the `BitmapManager` object owned by the Main thread. When the Main thread receives the repaint message, it asks the `BitmapManager` to fill a buffer of 32-bit BGRA-formatted pixel values based on the current state of the simulated computer's screen memory map. Then, the Main thread updates the `ID2D1Bitmap` object and makes a Direct2D call to render the updated bitmap.

Future Work

I am happy with the state of the project at this point, insofar as it runs without issues on my laptop. There are some opportunities to make the project better, though.

- Currently, the entire Hack screen is copied to the bitmap buffer every frame. Surprisingly, the animation is still very smooth on my laptop with only integrated graphics. A smarter algorithm could keep track of which parts of the screen have changed to take advantage of more efficient bitmap rendering.
- One might allow opening `.jack` and `.asm` files in addition to the currently recognized `.hack` files by building in a Jack compiler and VM translator.

- HackSim could use a menu bar for toggling optimizations instead of using a `.conf` file. The menu bar could also provide mechanisms for swapping Hack programs, resetting the emulator, and capping the clock speed.
- It would be nice to be able to resize the window to get a larger, scaled up view of the Hack screen.
- It would be cool to see diagnostic info (e.g., clock speed, FPS) while the simulator is running.

Endnotes

---

1. I chose optimizations very carefully based on whether I thought each optimization was "in the spirit of" the project. Though I decided to also include an efficient Hack computer simulator for fun (see Optimizations), that is not "in the spirit of" this experiment. There is great satisfaction in seeing the machine come to life, knowing that this is due to all of its thousands of cogs being simulated.↩
2. The Nand2Tetris course does not assume `Nand16` as primitive, but HackSim uses this circuit to achieve significant 16-bit optimizations. It is not hard to imagine how such a circuit would be designed, and so I think accepting this circuit as primitive is still in the spirit of Nand2Tetris.↩