**Simulation of Physical RISC Architectures Using a Modified Turing Machine and SRAM**

**Abstract**

Model provides a compromise between turing machines and physical architectures. It caps the circuit complexity and includes fast array accesses.

* Functional model for risc architectures
* Add RAM with access time O(1) to a turing machine in the most minimal way possible
* Path indexing and return matrix relate this to any risc, including the log aspects
* Pay attention to the circuit complexity -> using transistors instead of logic gates and formal definition because this more directly corresponds to the silicon on chips
* Goal is to provide a system with minimal circuit, memory, and time complexity to simulate RISC.
  + If time and space complexity are one and the same, then an increase in time or memory complexity on this system to execute complex operations like floating point multiplication on this system will correspond to any other system

We introduce a physical RISC system to examine, then introduce a modified turing machine which is weaker than the physical RISC system. The power with which this tape machine can execute different algorithms is proven. This modified tape machine allows analysis of fast array accesses; input arrays can be processed in O(log(n)) time and space as a function of input size instead of O(n^k) (where k is constant) time and space. Using a novel algorithm, the time complexity for arbitrary data movements on a stream of size n is O(log^2(n)), the instruction complexity is O(log(n)), the RAM word complexity is O(n), and the number of transistors excluding RAM/ROM is O(log(n)) (because of register sizes). O(n) RAM is okay because the RAM used for the array is already O(n). Using another algorithm, function returns on the base system (where n is the number of function calls; we assume that the number of function calls is significantly lower than the total number of words/instructions in ROM and RAM) require O(log(n)) time, O(n) instructions, O(1) RAM words, and O(1) transistors. O(n) instructions is okay because executing the function already requires O(n) instructions. Furthermore, unlike a Turing machine, the circuit complexity of this tape machine with respect to input stream size is capped; because it is simpler than the baseline RISC architecture, it requires at most that many transistors.

**Physical Computer Architecture**

This architecture will provide an upper bound on the circuit complexity of the turing machine setup described next.

**Assumptions about Transistors**

Work perfectly and produce a perfectly sharp output with a constant time delay. There is no limit for the number of transistors that can be stacked or fed from a single transistor. The speed of communication between transistors is considered to be instantaneous. Transistors are magically supplied with the perfect amount of power, and no noise exists.

**Details about SRAM**

*Theorem 1:*

The propagation delay in memory (RAM or ROM) independent of the total pointer size. This is possible with at least one multiplexer design using the transistor assumptions laid out in the previous section.

*Proof:*

Consider an architecture where the pointer is passed through a series of parallel not gates. This gives separate lines with the values for each bit of (pointer) and ~(pointer). ~ is the bitwise inversion operator as used in C.

We assume that the access time for a lonely memory bit is constant regardless of the presence of other bits. This does not refer to reading a value at a specific location through a multiplexor, just accessing the value directly.

Each address in memory corresponds to an arrangement of bits taken from (pointer) and ~(pointer) such that, if and only if (pointer) is equal to the address, all of the bits taken from (pointer) and ~(pointer) will be 1’s. Use the set S to represent the set of all bits that will be 1’s if and only if (pointer) = address. Use P to represent (pointer) and Q to represent ~(pointer). Use A to represent the value of the address of the memory bit. Pi refers to the value of the “i”th bit in P starting indexing where 0 is the rightmost bit. Qi and Ai are analogous. Let n be the number of bits in the pointer.

For all Pi equals P0 through Pn, Pi is a member of S if and only if Ai equals one. For all Qi equals Q0 through Qn, Qi is a member of S if and only if Ai equals zero. The members of S are all one when P = A (note that P = ~Q). Because S contains bits that must change if any bit in P changes, any deviation of P away from equalling A will result in a bit ceasing to be one; the only alternative is zero, so it will change to zero. It is for this reason that each bit also has a unique S composed of bits from P and Q.

The bits in S for each memory address and the value at that memory address are all and-ed together. The results of the and-ing for every address are then or-ed together, which produces the final output of the multiplexor for memory.

The bits in each S for each bit in memory update with the max delay of a single not gate, which does not increase in time or memory complexity with the pointer or array size. The physical transistor complexity increases linearly with pointer size. Then, the propagation delay for the and and or gates is also independent of the number of bits; all transistors can update at once, and the number of gates chained together end-to-end is only two. Again, for these gates, the number of transistors increases linearly with pointer size. The number of transistors in the multiplexer as a whole increases as O(log(n)) where n is the number of total words stored in memory. The number of transistors in memory increases linearly with array size.

**Instruction Syntax**

Uses an op code only. The next and previous locations are explicitly stated. The addresses are counted forward and backward (don’t need Gray code). While using excessive space in ROM, this is fine because it only increases the ROM complexity by a constant factor and thus doesn’t affect big O analysis.

**General Structure**

Detailed diagram (basically use the map25.2 without a general register; it can be more powerful than the turing machine, just make it VERY simple)

*Figure 1: Functional diagram of the MIN*

**Diagram Description**

All registers are constantly outputting and accepting input at the same time. There is a latch enable for each register which decides whether its value tracks the input or is held constant. ROM is constantly outputting and accepting address changes. This is why it has a separate “z-pass” which is essentially a tri-state buffer.

**System Operation**

The general shape is to reset, then have a fetch and decode process. Each op code corresponds to moving a single bit, either reading it into the program address or writing to the address register or read/write from ram (which has single bit words).

Note that the curr register is two bits smaller than the nxt register, and it feeds into the top of the nxt register. This is because the last two bits are reserved to store the state of if-else statements. Typical branching is not possible because each instruction can only go either forward or backward.

The instructions for the computer are read-only from the perspective of the program. They must be edited before turning the computer on by an external entity.

Note that the address register is write-only from the perspective of the program. Only the ram multiplexer reads the address register.

This setup means that the entire computer only has a few methods of buffering data to use later: either spit it into RAM, load it into the program state using an “if” statement, or m ove the position of the machine in ROM.

The design doesn’t need an ALU because it has a single bit to begin with, and operations on a single bit can be simulated simply by using nested “if” statements.

**Reset Procedure**

Reset must go to the next register due to timing. Discuss good reset behavior.

**Handshakes and clearing the Buses**

Requires a four-cycle latch.

2 main handshake categories:

1. Need to ensure that multiple gates don’t output to the same portion of the same bus.
2. Prevent “racing” and/or shoot through

**Definition in Terms of a Turing Machine**

**Detailed Definition**

Qualitatively describe the structure and what the system actually does.

Let s be the number of states on the Tape MIN and a be the number of possible RAM index values. The number of possible values in each RAM cell is ra. Let the number of possible values in each ROM cell be ro. The number of possible operations per tick is always 3 because the machine can write to ram, move the ram address head, and write to the ram address (excludes moving the ROM head and changing state) . Let S be the state of the system. Let Ro be the position of the ROM head. Let Ra be the position of the RAM address head.

The ROM tape is finite but large in the right direction and has a start point in the left direction. The ROM head starts at index zero. Let Ro(0) be the value of the ROM tape at the leftmost location, and Ro(n) be the value at a location that is a displacement of n to the right.

Assume that the RAM tape starts initialized with all zeros. Let Ra(0) be the value of the ROM tape at the RIGHTMOST location, and Ra(n) be the value at a location that is a displacement of n to the left. It is finite in the left direction and has a total size which is just enough to index all values in RAM; the size is ceiling(ln(RAM size)/ln(a)). A is the value of the tape as a whole (the rightmost bit is the lowest value; the value of RAM is simply the ram values written out directly as a binary number). M is the value at that address of RAM.

The following are definitions for instructions. Let I(Ro(n), Ra(A), S, ADDR) be the address write for the corresponding values of ROM, RAM, and state at position n on the ROM tape. Let I(Ro(n), Ra(A), S, RAM) be the same thing for the write RAM value. Let I(Ro(n), Ra(A), S, MOVA) be the same thing for direction to move the RAM address head. Let I(Ro(n), Ra(A), S, MOVR) be the same thing for the direction to move the ROM head. Let I(Ro(n), Ra(A), S, STATE) be the same thing for the new state.

The above can be used to define the “logic table” of the system, the size of the system, and the values within the system. They can be dynamically edited to represent the execution of the system.

See the wikipedia turing machine page and mirror that formal definition.

**Command Unit Truth Table**

\*Order inside cells is (write RAM addr value, write ram value, move RAM addr head, move ROM head, next state). They execute in the order written from left to right.

| (RAM, ROM); State | A | B | C | D | E | F |
| --- | --- | --- | --- | --- | --- | --- |
| (0, 0) | 0\*\*RA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | 0\*\*RC |
| (0, 1) | 1\*\*RA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | 1\*\*RC |
| (0, 2) | \*0\*RA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | \*0\*RC |
| (0, 3) | \*1\*RA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | \*1\*RC |
| (0, 4) | \*\*RRA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | \*\*RRC |
| (0, 5) | \*\*LRA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | \*\*LRC |
| (0, 6) | \*\*\*RA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | \*\*\*RC |
| (0, 7) | \*\*\*RA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | \*\*\*RC |
| (0, 8) | \*\*\*RF | \*\*\*RF | \*\*\*LB | \*\*\*LB | \*\*\*LB | \*\*\*LB |
| (1, 0) | 0\*\*RA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | 0\*\*RC |
| (1, 1) | 1\*\*RA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | 1\*\*RC |
| (1, 2) | \*0\*RA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | \*0\*RC |
| (1, 3) | \*1\*RA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | \*1\*RC |
| (1, 4) | \*\*RRA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | \*\*RRC |
| (1, 5) | \*\*LRA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | \*\*LRC |
| (1, 6) | \*\*\*RA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | \*\*\*RD |
| (1, 7) | \*\*\*RA | \*\*\*LB | \*\*\*RD | \*\*\*RE | \*\*\*RF | \*\*\*RE |
| (1, 8) | \*\*\*RF | \*\*\*RF | \*\*\*LB | \*\*\*LB | \*\*\*LB | \*\*\*LB |

**Modified Turing Machine Power**

The physical machine is at least as powerful as the turing machine. This is because, as structured, the turing machine has less functionality in terms of movement (only forward and backward versus anywhere at any time). The physical computer is able to buffer up to O(log(n)) “if” statements in the program state where n is the length of ROM, while the turing machine can only buffer up to 2 “if” statements in the program state. The physical machine can edit any bit of the RAM address at once, while the turing machine can only move up or down in which bit it is editing.

The fact that the turing machine doesn’t have to have no-ops might make it seem more powerful. However, insert logic.

**Steps to Simulate Tape MIN on Register MIN**

Each physical position in the ROM corresponds to a position on the ROM tape from the Tape MIN. The general structure is to have sections of size (# possible states)\*(# possible ram values)\*(# operations possible for each turing machine tick) for each section in ROM. The value of the ROM tape in this section is simply used to determine what instructions to write because executing different instructions is the only result of the Tape MIN reading its ROM tape anyways.

“ROM position buffering” is defined as follows. Consider a series of instructions K[0] through K[m] that constitute any operations you want. There is a value, which is either 1 or 0, and the goal is to buffer that value during the execution of all instructions in K. In order to do this, write instructions into ROM, writing K[0] twice, then K[1] twice, etc until K[m]. If the contents of ROM are dumped sequentially, they should be as follows: K[0], K[0], K[1], K[1], K[2], K[2], K[3], K[3], … K[m], K[m]. This means that, if you index through instructions by twos, then there are two different threads of K. Before entering this sequence, simply use an if statement to see if the value you want to buffer is a 1 or a 0. If it is a 0, jump to the first thread of K; otherwise, jump to the second thread of K. After the threads are done executing, each can jump to a different location which dumps either a 1 or a 0 into RAM or wherever you want it. You can also modify each version of K to do different things based on the value buffered. This “ROM position buffering” is the fundamental building block for simulating instructions of the Tape MIN on the Register MIN.

**Simulation Complexity**

*Theorem:*

Every instruction on the turing machine can be simulated by the physical machine in

*Proof:*

Insert here

Therefore, the physical machine can simulate the operation of the turing machine by simply simulating each of its instructions in turn. Therefore, every proof below regarding the power of the Turing machine extrapolates to a physical machine of at most the size of the MIN.

**Further Simplification**

Can we compress some of the operations to make better use of the states C, D, and E? This might lower the number of required symbols

**Virtual Machine**

Can’t simulate the transistors themselves because we assumed that the speed of light is infinite, so the electrical signal will transmit through the transistor quickly. This would require looping back through every transistor in the chain to see if it is connected to a power rail or is just pulled up to 5 volts.

Note that the VM will grow faster than linearly with respect to word size when simulating a full-fledged ALU. However, this is okay.

**Importance of Simulation**

Touching on the simulation and complexity for a general set of functions touches on whether np = p.

Touch on the difference between finite and infinite space.

**Modeling F in Terms of Unit Functions (eg simulating 2-input nand gates)**

*Theorem 2:*

Because F can be constructed as a physical computer, its computation can be modeled as a series of logic gates. Need to rigorously translate to f(a, b) = f(a, c) = c and data indexes being simulated on the S(<sufficiently large>, n) to simulate with only linear slowdown.

**Complexity of Simulation for a Single Clock on the Virtual Machine**

The Tape MIN is designed to simulate logic gates in succession. It can rewind and get stuck in an infinite loop; however, this loop can be at most one layer deep because it has no differentiation for other depths. Theoretically, RAM could store the loop depth and help determine which loops to begin/end. However, this would destroy the value of the address tape; this tape is write only, so it can’t be buffered. Moreover, it requires actually having a program pointer, which is nearly impossible to simulate without one natively available.

Because there are no loops inside the tick simulation, and “if” statements can only go 2 deep, time and ROM space complexity for simulating the virtual machine are the same. Furthermore, because many of the expressions for time and ROM space complexity vary as a function of the number of nand gates in the virtual machine, this is kind of a way to translate circuit complexity into time and ROM complexity at the cost of efficiency.

**Array Accesses**

Despite not having a

**Path Indexing**

This is used to pull a pointer from RAM and use that pointer to access and retrieve another value in RAM. It can also be used to pull a pointer from RAM, pull a value from RAM, and dump that value into the address pointed to by the pointer. Prove that this allows for all possible array functions.

Assume that there is only 1 RAM access per tick.

1. Simulate the 2-input nand gates of the virtual machine
2. After that is done, the outputs of several nand gates can also be used to access a place in RAM.
   1. It doesn’t matter how long it takes to transition from these gates to the path itself; at worst, it will require O(log(w)) time to switch from the gates to the path (where w is the number of gates in the base computer), which is already exponentially faster than simulating the virtual computer’s logic gates.
   2. The output of each gate will simply be referred to as bit 0, bit 1, etc for simplicity sake. Bit 0 is the least significant digit of the array pointer.
3. Bit 0 is buffered using an “if” statement.
4. The upper bits for the path are then written by placing a constant in the upper values of the RAM tape; simply move the RAM head up, then start placing values, then move it back down.
   1. This has complexity in time and ROM space of O(log(n)) where n is the number of bits in the RAM array (this is because you have to move less upwards if the RAM address needs to be smaller, and the upper bit signifying that one has entered the path index can simply be a single zero). At worst, it adds to the O(log(w)) complexity described before for w being the number of transistors on the virtual machine.
5. Place the value of bit 0 in the beginning of the path.
6. Go back to and buffer the value of bit 1 using an “if” statement.
7. Go to the beginning of the path and buffer the value there using an “if” statement.
8. Place a bit higher on the address to indicate a new portion of the path, place the value buffered from the “if” statement executed just now (put it in the lower most address), then place the value of bit 1 in the RAM.
9. Repeat steps 5 through 8 for bit 2, bit 3, bit 4, etc until the end. Note that there will be more chaining for each step. Note that all of this is explicitly written out; no loops.

The complexity to simulate one tick is (wlogw + log^2r). W is the number of logic gates in the virtual machine and r is the number of values in the RAM array. Note that w should also be written as a function of r because it could change based on that. Having linear complexity with w is okay because we assume that w stays constant, and we only care about changing word size for processing; in other words, just pick a sufficiently powerful virtual machine and then you don’t have to fester with w at all as you do more complex things with the system. What about adding and more complex operations that don’t scale the number of logic gates as the log of word size?

You have to bit bash instead of using a loop because it requires 3 “if” statements to do the type of looping required, but the machine only has enough states for 3 “if” statements. Proof:

1. You have to buffer whether to return or keep going on the path (eg the flag values for looping because you have to mark, on the path, how far you are going because the program doesn’t keep track of it).
2. While doing so, you also have to buffer the value of the next value to write to the path and the next value to transfer into the address.

An actual use case for gotos!

**Return Matrix**

Not strictly necessary because the virtual machine can just have an additional register to buffer returns.

Rigorously define what a return lattice is.

**Compromises**

Basically, take a huge coefficient in order to have linear or the like end behavior.

**More Complicated Operations**

Note that, while this system does not have an “and” operation, one can be constructed programmatically. Insert the program here.

The complexity increase is linear in time with respect to input size, and this is

This is just a proof that it can generate complex programs easily. We already know about array accesses and arbitrary simulation happen within polynomial time of any other computer, specifically O(n^3logn) in time and ROM and RAM and O(n) in transistors \*\*THIS IS WRONG. This is just a demonstration of extrapolation and programming for practical use.

Example: binary search can be conducted in O(log(n)) time, but it would take exponentially longer on a turing machine.

**Conclusion**

A Turing machine with one read-only head/tape and one write-only head/tape that connects to single-bit-access RAM is capable of simulating array data transfers in O(log(n)) time without the need for any helper registers. This machine can simulate every function of complexity O(n^k) on a sufficiently powerful computer in complexity O(n^(6k)) and at least some non-trivial functions of complexity O(log(n)) on a sufficiently powerful computer in at most O(log^12(n)).

An upper bound for the universal Tape MIN’s number of states and values is 6 states, 9 values on the rom tape, 2 values on the RAM tape, and a RAM tape of size log(# bits in RAM)

**Open Questions**

Are the four fundamental functions really enough to measure the power of any computer when taking into account fast array access?

**References**

<http://homepage.cs.uiowa.edu/~jones/arch/risc/>

<https://dl.acm.org/doi/10.1145/36206.36197>

Time hierarchy theorem proof

2,3 state machine proof

Proof that turing machine can simulate any algorithm in O(n^6)

<http://theory.cs.princeton.edu/complexity/circuitschap.pdf>

<http://theory.cs.princeton.edu/complexity/circuitschap.pdf>