`mov` is Turing-complete

Stephen Dolan
Computer Laboratory, University of Cambridge
stephen.dolan@cl.cam.ac.uk

Abstract

It is well-known that the `x86` instruction set is baroque, overcomplicated, and redundantly redundant. We show just how much fluff it has by demonstrating that it remains Turing-complete when reduced to just one instruction.

The instruction we choose is `mov`, which can do both loads and stores. We use no unusual addressing modes, self-modifying code, or runtime code generation. Using just this instruction (and a single unconditional branch at the end of the program to make nontermination possible), we demonstrate how an arbitrary Turing machine can be simulated.

1. Introduction

The `mov` instruction on `x86` has quite a few addressing modes. This instruction can be used to do memory loads or stores, as well as loading immediate constants into registers. Powerful as it is, it doesn't do any form of conditional branching or comparison, so it's not obvious that it is Turing-complete on its own.

Of course, on an actual `x86` processor the `mov` instruction can be used to write arbitrary code into memory after the instruction pointer which the processor will then execute, making it in some sense trivially "Turing-complete". We consider this cheating: our simulating of a Turing machine uses no self-modifying code nor runtime code generation, and uses no obscure addressing modes. In fact, the addressing modes used are available as instructions on most RISC architectures, although RISC machines generally don't call them all `mov`.

Executing a finite sequence of `mov` instructions will complete in a finite amount of time. In order to have Turing-completeness, we must allow for nontermination. So, our Turing machine simulator consists of a sequence of `mov` instructions, followed by an unconditional branch back to the start.

2. Machine model

We work with a simple abstract machine model. Our machine has a random access memory composed of words. Each word can hold either a memory address or an offset, where offsets are either 0 or 1 (which are not valid memory addresses). We have n registers $R_1, \ldots, R_n$, which also hold a word each. We assume plenty of registers for now, but later we show how their number can be reduced without losing expressiveness.

We have the following instructions (if you like RISC) or addressing modes (if you like CISC). We use Intel `x86` syntax, where the `mov` instructions have the destination first and the source second, and square brackets indicate memory operands.

Instruction	`x86` **syntax**
Load Immediate	`mov` $R_{\texttt{dest}}$`, c`
Load Indexed	`mov` $R_{\texttt{dest}}$`, [`$R_{\texttt{src}}$ `+` $R_{\texttt{offset}}$`]`
Store Indexed	`mov [`$R_{\texttt{dest}}$ `+` $R_{\texttt{offset}}$`],` $R_{\texttt{src}}$

On `x86`, these are all addressing modes of the same `mov` instruction. However, even on RISC machines these are still single instructions. For instance, on PowerPC these three operations are `li`, `ldx` and `stx`.

It would appear that we are cheating slightly by allowing arithmetic in addresses. However, our "arithmetic" is of a very restricted form: the indexed instructions may only be used when $R_{\texttt{src}}$ (or $R_{\texttt{dest}}$ for stores) is an even-numbered memory addresses. Since offsets are always 0 or 1, then our "arithmetic" can be implemented as bitwise OR, or even bitstring concatenation.

From these three instructions, we have a few simple derived instructions. We can use the load indexed and store indexed instructions with a constant offset by using a temporary register. For instance, the load instruction `mov` $R_{\texttt{dest}}$`, [`$R_{\texttt{src}}$`]` can be implemented as follows, using the register `X` as a temporary:

```
mov X, 0
mov R_dest, [R_src]
```

As it happens, these constant-offset instructions are available as other addressing modes of `mov` on `x86`.

Memory is logically divided into cells, which are pairs of adjacent words which start on even-numbered memory addresses. Our load indexed and store indexed operations can be viewed as load and store to a cell, where the address of the cell is given by one register, and which of the two words to access is specified by another register.

Using just these instructions, we can simulate an arbitrary Turing machine.

3. Representing Turing machines

A Turing machine $\mathcal{M}$ is a tuple

$$\mathcal{M} = (Q, q_0, \Sigma, \sigma_0, \delta)$$

whose components are:

- A finite set of states Q, with a distinguished start state $q_0 \in Q$.
- A finite set of symbols Σ, with a distinguished blank symbol $\sigma_0 \in \Sigma$.
- A transition table δ, which is a partial function $Q \times \Sigma \to \Sigma \times \{L, R\} \times Q$.

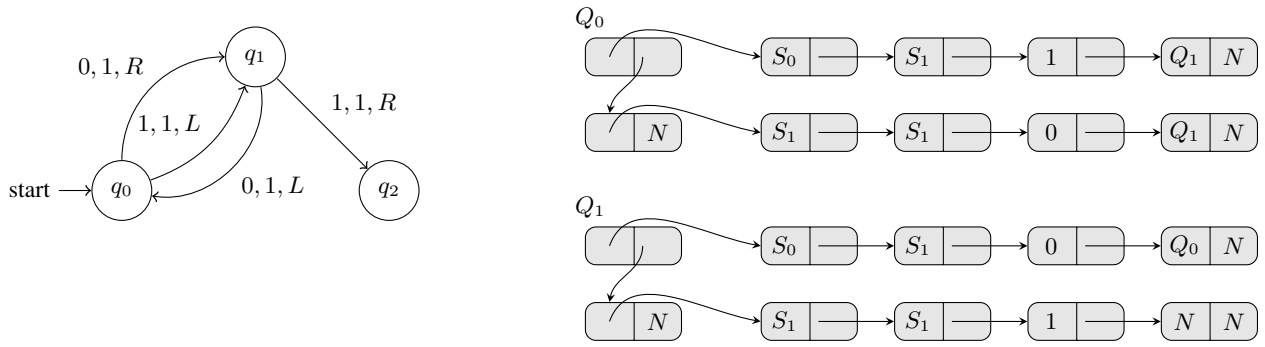

Figure 1. A simple Turing machine and its transition table represented as linked memory cells. Transitions are labelled with their triggering symbol, new symbol, and direction.

The Turing machine has a tape, which consists of a infinite sequence of positions each of which contains a single symbol. The machine has a current state, which is initially q_0, and a current position, which is initially the leftmost position on the tape with the tape extending infinitely to the right. All tape positions initially contain σ_0.

The machine repeatedly consults the transition table δ. If δ is not defined for the current state and the symbol at the current position, then the machine halts. If it is defined as (σ', d', q'), then the current state is set to q', the symbol at the current position is set to σ', and the current position moves leftward one place (if $d' = L$) or rightward one place (if $d' = R$), and the machine continues.

We can represent a Turing machine in memory cells. The symbols are represented as cells at addresses $S_1, \ldots, S_{|\Sigma|}$, where each symbol in Σ corresponds to some S_i and the blank symbol σ corresponds to S_1. The contents of the cells S_i are unspecified.

We represent states and the transition table as lists. Our cells are like Lisp cons cells, and so we may represent lists like Lisp does: a non-empty list is represented by a cell whose first word contains contains the first element of the list, and whose second word points to the rest of the list, represented in the same fashion. The empty list is represented by a cell at address N whose contents are unspecified.

For any state q, we say that its set of outgoing transitions is the set of tuples $(\sigma, \sigma', d', q')$ such that $\delta(q, \sigma) = (\sigma', d', q')$. Each state is represented as a list of its outgoing transitions, and each outgoing transition is represented as a list of four elements: the triggering symbol σ and the new symbol σ' are represented as the addresses of the corresponding cells S_i and S_j, the direction d' is represented as 0 (if $d' = L$) or 1 (if $d' = R$), and the new state q' is represented as the address of its list of outgoing transitions.

In the Turing machine of Figure 1, there are three states q_0, q_1 and q_2. States q_0 and q_1 have two outgoing transitions each, and q_2 has none. The cell Q_0 holds a list representing the outgoing transitions of q_0. Both of these transitions are to the state Q_1, so the fourth cell of the list representing each transition holds the address of Q_1 (this is not shown by an arrow on the diagram to avoid clutter). The state q_3 has no outgoing transitions, so is represented by N, the empty list.

All of the other cells of memory form the tape of the Turing machine. To simulate a Turing machine, we assume the tape is infinite (although on a real machine it will be bounded by address space), and so the cell addresses are given by the sequence $T_1, T_2, \ldots$. These cells are initialised so that the contents of the word at address T_n is the address S_1, and the contents of the word at address $T_n + 1$ is the address T_{n+1}.

In this way, we can think of T_1 as being the start of an infinite list, where each element of the infinite list initially contains S_1.

As is usual when discussing Turing-completeness, we concern ourselves only with computation, ignoring input and output. We assume that the input is a sequence of symbols placed in the first word of the cells T_1 to T_n before our program starts, and that the output of the program is determined by examining the tape afterwards.

Our version of Turing machines doesn't have particular accepting or rejecting states, but this is not a loss of power as they can be emulated by writing an "accept" or "reject" symbol to the tape before halting.

4. Comparisons and conditionals

A fundamental aspect of computation is branching: choosing which action to perform next based on a runtime value. So, our machine's total lack of conditional branches, computed jumps or anything that even smells like a comparision instruction might seem like an impediment.

Yet as it turns out, we can do comparisons using only load and store instructions. If you store a value into address A and a different value into address B, then examining the contents of address A afterwards will tell you whether A and B are equal. Suppose registers R_i and R_j both point to symbols (that is, their values are drawn from $S_1, \ldots, S_{|\Sigma|}$). We can compare them as follows:

```
mov [Ri], 0
mov [Rj], 1
mov Rk, [Ri]
```

This clobbers the values at addresses R_i and R_j, but this doesn't matter since symbol cells have unspecified contents. The effect is that R_k gets the value 1 if $R_i = R_j$, and 0 otherwise.

We can also compare an arbitrary address against N using the following code. In this example, R_i is the address being compared, and we save its value in the scratch register `X` to avoid clobbering. We assume the register `N` contains the address N.

```
mov X, [Ri]
mov [N], 0
mov [Ri], 1
mov Rj, [N]
mov [Ri], X
```

After this sequence, R_j is 1 if R_i is equal to N, and 0 otherwise.

This allows us to do a comparison, resulting in either zero or one. We can then use that result to select between two different values. Suppose we have two registers R_i and R_j, and a register R_k that contains either 0 or 1. We can use R_k to select between R_i and R_j by using R_k to index a two-element lookup table. The address

N is conveniently available in the register `N` and has unspecified contents, so we use that cell to hold our lookup table:

```
mov [N], R_i
mov [N+1], R_j
mov R_l, [N + R_k]
```

This sequence causes R_l to get the value of either R_i or R_j depending on R_k.

With these operations, we are able to simulate an arbitrary Turing machine, as the next section describes.

5. Simulating a Turing machine

We use a register `T` to hold the current transition to be tested, and a register `S` to hold a cell containing the current symbol. The register `L` holds the list representing the part of the tape to the left of the current position, and the register `R` holds the list representing the part to the right.

At program startup, `T` holds the address Q_0, and `S` holds the address T_1. The register `L` holds the address N, and `R` holds the address T_2. The register `L` holds the portion of the tape to the left of the current position in reversed order (which is initially the empty list N). The order is reversed so that the nearest position is always the first element of the list, so moving left can be done without processing the entire list.

The register `R` holds the part of the tape to the right of the current position, which is initially T_2, the list of all but the first position on the tape. The two lists held in `L` and `R` are used as stacks, where moving rightward means pushing the current cell to `L` and popping the next cell from `R`.

Since we use N a lot, we assume the register `N` always contains the address N.

First, we check whether the current transition should fire, by comparing `S` and the symbol on the current transition `T`.

```
mov X, [T]     ;; get transition
mov X, [X]     ;; get trigger symbol
mov Y, [S]     ;; get current symbol
mov [Y], 0     ;; compare symbols
mov [X], 1
mov M, [Y]
```

After this sequence, the register `M` is 1 if the transition matches, and 0 otherwise. Next, we update `S`: if the transition matches, we use the transition's new symbol, otherwise we leave it unchanged.

```
mov X, [T]     ;; get transition
mov X, [X+1]   ;; skip trigger symbol
mov X, [X]     ;; load new symbol
mov Y, [S]     ;; load old symbol
mov [N], Y     ;; select between X and Y
mov [N+1], X
mov Z, [N + M]
mov [S], Z     ;; write the new symbol
```

This updates `S` if the transition matches. Next, if the transition matches, we need to advance the tape in the appropriate direction. We do this in two stages. First, we push the cell `S` to one of the tape stacks, and then we pop a new S from the other tape stack. If the transition does not match, we push and pop S from the same tape stack, which has no effect. To determine whether the transition moves left or right, we use the following sequence:

```
mov D, [T]     ;; get transition
mov D, [D+1]   ;; skip trigger symbol
mov D, [D+1]   ;; skip new symbol
mov D, [D]     ;; load direction
```

After this, the register `D` holds the direction of tape movement: 0 for left, and 1 for right. If we are to move left, then the cell `S` must be added to the tape stack `R`, and vice versa. Adding the cell to a tape stack is done by first writing the tape stack's current top to `[S+1]`, and then modifying the tape stack register to point at `S`.

```
mov [N], R     ;; select new value for [S+1]
mov [N+1], L
mov X, [N + D]
mov [S+1], X
mov [N], L     ;; select new value for L
mov [N+1], S
mov L, [N + D]
mov [N], S     ;; select new value for R
mov [N+1], R
mov R, [N + D]
```

We must ensure that no movement of the tape happens if the transition does not match (that is, if `M` $= 0$). To this end, we flip the value of `D` if the transition does not match, so that we pop the cell we just pushed.

```
mov [N], 1     ;; set X = not D
mov [N+1], 0
mov X, [N + D]
mov [N], X     ;; select between D and X
mov [N+1], D
mov D, [N + M]
```

Next, we pop a cell from a direction indicated by `D`: if `D` $= 0$, we pop a cell from `L`, and if `D` $= 1$ we pop one from `R`.

```
mov [N], L     ;; select new value of S
mov [N+1], R
mov S, [N + D]
mov X, [S + 1] ;; get new start of L or R
mov [N], X     ;; select new value for L
mov [N+1], R
mov L, [N + D]
mov [N], R     ;; select new value for R
mov [N+1], X
mov R, [N + D]
```

So, if the current transition matches, this code writes a symbol to the tape and advances in the appropriate direction. If the transition doesn't match, this code has no effect.

All that remains is to find the next transition to consider. If the current transition matches, then we should look at the next state's list of transitions. Otherwise, we continue to the next transition in the current state.

```
mov X, [T + 1] ;; get next transition of this state
mov Y, [T]     ;; get current transition
mov Y, [Y + 1] ;; skip trigger symbol
mov Y, [Y + 1] ;; skip new symbol
mov Y, [Y + 1] ;; skip direction
mov Y, [Y]     ;; load transition list of next state
mov [N], X     ;; select next transition
mov [N + 1], Y
mov T, [N + M]
```

This finds the next transition our Turing machine should consider. If `T` has the value N, then this means there are no more transitions to consider: either we got to the end of a state's list of transitions with no matches, or we just transitioned to a state that has no outgoing transitions. Either way, the machine should halt in this case. First, we check whether this is the case by setting the register `H` to 1 if `T` is N:

```
mov X, [T]
mov [N], 0
mov [T], 1
mov H, [N]
mov [T], X
```

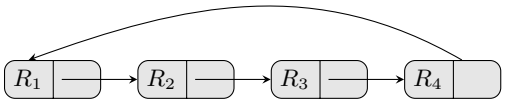

Figure 2. Scratch space as a circular list

If H is 1, we must halt the machine. We do so by reading from the invalid memory address 0:

```
mov [N], 0     ;; select between 0 and N
mov [N+1], N
mov X, [N + H]
mov X, [X]     ;; load from 0 or N
```

If this memory access does not halt the program, then we have successfully found another candidate transition and put a pointer to it in T, and we have the current symbol cell in S. We are therefore in a suitable state to run the program again, so we use our single unconditional jump to loop back to the start:

```
jmp start
```

This simultes an arbitrary Turing machine, using only mov (and a single jmp to loop the program).

6. Register use

While building our Turing machine simulator, we made free use of many temporary registers. The x86 architecture has never been accused of having too many registers, and we seem to already have exceeded the budget of 8 general-purpose registers.

The registers could be allocated more carefully to fit within the limit, but we take a different approach: we show that any program that can be implemented using our restricted instruction set can be translated to an equivalent program using at most four registers.

Suppose the original program uses n registers $R_1, \ldots, R_n$. We represent these in the translated program with n preallocated cells of scratch space. The second word of each cell in scratch space points to the next cell, and the second word of the last cell points to the first, laying them out as a circularly linked list (see Figure 2).

Our four registers are S, which at startup points to the first cell of scratch space, and three other registers A, B and C. We can advance S to the next cell as follows:

```
mov A, 1
mov S, [S + A]
```

If the instruction `mov S, [S + 1]` is available directly, then this can be done without using A. This allows us to load any R_i value into A, B or C: we load 1 into the destination register, then advance S to the correct position, and then perform `mov A, [S]`.

We always know during the translation which scratch cell S points to, so the number of `mov S, [S + A]` instructions required to move S to the right scrach register is easily determined. For instance, to load R_2, R_4 and R_1 into registers A, B and C respectively, we generate the following code (assuming four scratch cells):

```
mov A, 1
mov S, [S + A]
mov A, [S]

mov B, 1
mov S, [S + B]
mov S, [S + B]
mov B, [S]

mov C, 1
mov S, [S + C] ;; wraps back to R1
mov C, [S]
```

Our operations have at most three source operands (for indexed store), so for any operation we can use a sequence like the above to load the operands into the registers A, B and C. We generate code to perform the operation using those registers, and then generate code to store the result back into the scrach space (if there is a result). We assume that the result is in register A, but it is trivial to modify the following for any other result register.

We use B as a scratch register, and cycle S as before:

```
mov B, 1
mov S, [S + A]
```

The `mov S, [S + B]` instruction is repeated as many times as necessary to reach the desired scratch cell, and then the result is stored using `mov [S], A`.

This transformation works for any program that can be implemented in our restricted instruction set, including the Turing machine simulator of the previous section. It is therefore possible to simulate an arbitrary Turing machine using only the mov instruction and four registers.

Thus, while it has been known for quite some time that x86 has far too many instructions, we can now contribute the novel result that it also has far too many registers.

7. Discussion

Finding Turing-completeness in unlikely places has long been a pastime of bored computer scientists. The number of bizarre machines that have been shown Turing-complete is far too great to describe them here, but a few resemble what this paper describes.

That a computer can get by with just one instruction was shown by Farhad Mavaddat and Behrooz Parhami [2]. Their machine uses a single instruction that takes two memory addresses and a jump target, and its operation is "subtract and branch if less than or equal", combining arithmetic, memory load, memory store and conditional branching into one instruction.

A single instruction machine using only MOVE was described by Douglas W. Jones [1], where Turing-completeness is gained by having memory-mapped arithmetic and logic units (so that other operations can be performed by moving data a predefined memory location and collecting the result afterwards). Some machines based on this principle have been built, and are generally known as "move machines" or "transport-triggered architectures".

Raúl Rojas [3] shows that, with self-modifying code, an instruction set with load, store, increment, zero and unconditional branching is Turing-complete. In the same paper, he also shows that a machine without self-modifying code or code generation is Turing-complete with increment and double-indirect loads and stores. The double-indirect memory operations use a register as the address of a memory cell holding the address to access (in pseudo-x86 notation, the load looks like `mov A, [[A]]`).

Removing all but the mov instruction from future iterations of the x86 architecture would have many advantages: the instruction format would be greatly simplified, the expensive decode unit would become much cheaper, and silicon currently used for complex functional units could be repurposed as even more cache. As long as someone else implements the compiler.

References

[1] D. W. Jones. The Ultimate RISC. *ACM SIGARCH Computer Architecture News*, 16(3):48–55, 1988.

[2] F. Mavaddat and B. Parhamt. URISC: The Ultimate Reduced Instruction Set Computer. *International Journal of Electrical Engineering Education*, 25(4):327–334, 1988.

[3] R. Rojas. Conditional Branching is not Necessary for Universal Computation in von Neumann Computers. *Journal of Universal Computer Science*, 2(11):756–768, 1996.