Gadgets

To construct register machines which perform complex operations, we introduce *gadgets* which are components used to perform specific operations. A gadget is defined informally as a partial register-machine graph that has a designated initial label and one or more designated exit labels (which contain no instructions). The gadget operates on registers specified in the gadget's name, and are used for input and output — we call these the input/output registers. The gadget may use other registers for temporary storage — we call these *scratch registers*. The gadget assumes the scratch registers are initially set to 0, and *must* ensure that they are set back to 0 when the gadget exits. Ensuring that the scratch registers are reset to 0 is important so that the gadget may be safely used within loops.

Slide 1

Gadgets

A **gadget** is a partial register-machine graph.

It has one entry wire, and one or more exit wires.

The gadget operates on input and output registers specified in the gadget's name.

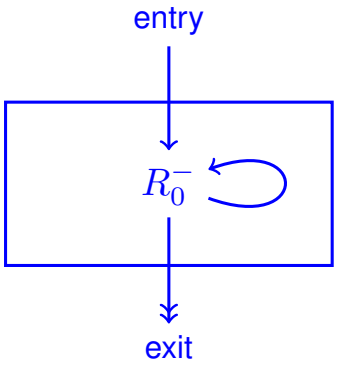
The gadget may use other registers, called scratch registers, for temporary storage.

The gadget assumes the scratch registers are initially set to 0, and **must** ensure that they are set back to 0 when the gadget exits.

Slide 2

Gadget: "zero R_0"

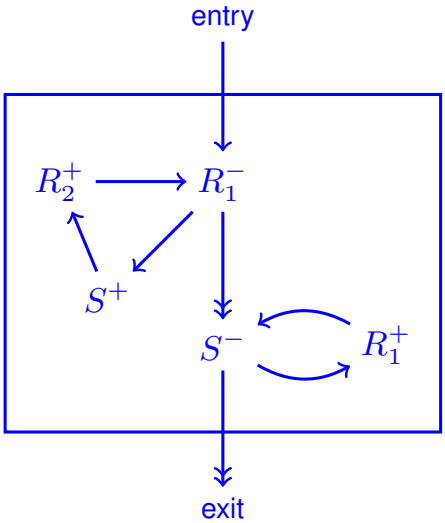
The gadget "zero R_0" sets register R_0 to be zero, whatever its initial value:



Slide 3

Gadget: "add R_1 to R_2"

The gadget "add R_1 to R_2" adds the initial value of R_1 to register R_2, storing the result in R_2 but restoring R_1 to its initial value.

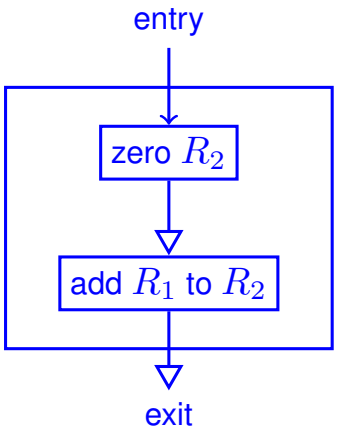


We can compose gadgets, constructing bigger gadgets and eventually complete register machines. To construct such bigger gadgets, we rename the registers used by each gadget: all of its scratch registers are renamed to things that do not occur in the rest of the machine, and its input/output registers are renamed to whichever registers the program requires. We then 'wire up' the gadgets to make bigger gadgets, by joining (possibly many) exit wires to the unique entry wire. For example, consider the gadget "copy R_1 to R_2" defined on slide 10. We will use this gadget to construct the universal register machine. The gadget "copy R_1 to R_2" copies the value of register R_1 into register R_2, leaving R_1 with its initial value. It does this by joining together the "zero" and "add" gadgets: it first sets register R_2 to zero, then adds the value of R_1 to R_2 and leaves the value of R_1 the same, using some scratch register inside the "add" gadget.

Slide 4

Gadget: "copy R_1 to R_2"

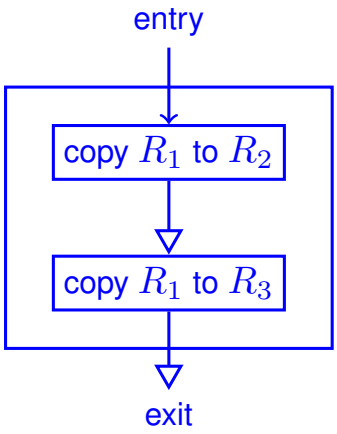
The gadget "copy R_1 to R_2" copies the value of register R_1 into register R_2, leaving R_1 with its initial value:



When we use gadgets to make other gadgets or register machines, we may need to rename the registers. For example, to construct the gadget instance "copy R_1 to R_3", R_1 (the original target in the definition of the gadget) gets replaced by R_3. Any scratch registers used in the gadget are renamed to be different from any other registers used elsewhere.

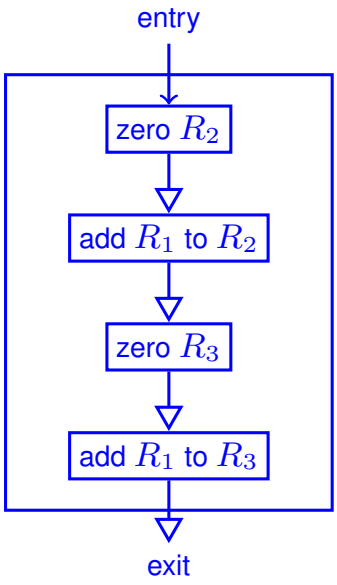
Slide 5

Gadget : "copy R_1 to R_2 and R_3"



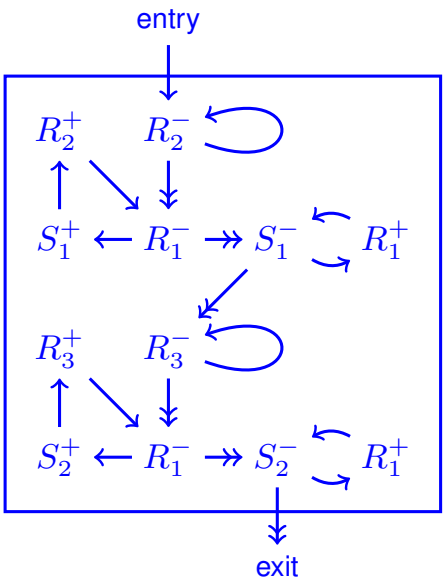
Slide 6

Gadget: "copy R_1 to R_2 and R_3"



Slide 7

Gadget: "copy R_1 to R_2 and R_3"

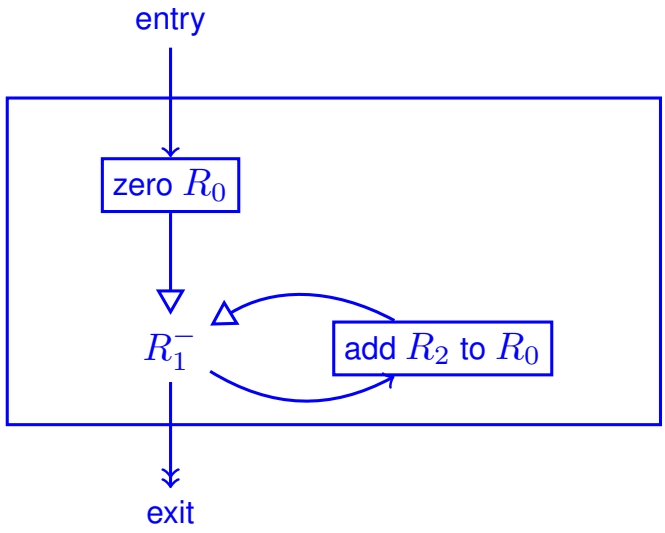


Recall the register machine for multiplication given earlier in the course, which multiplies R_1 by R_2 and stores the result in R_0, possibly overwriting the initial values of R_1 and R_2. We can construct this register machine using the "zero" and "add" gadgets.

Slide 8

Gadgets: "multiply R_1 by R_2 to R_0"

We can implement "multiply R_1 by R_2 to R_0" by repeated addition:

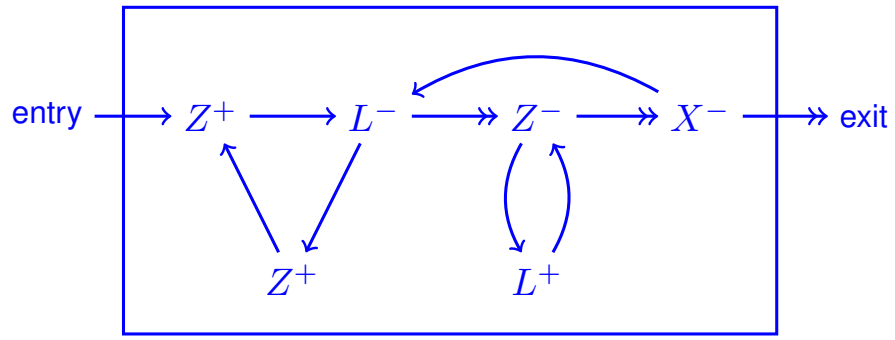


As well as the "copy" gadget, we require two more gadgets to define the universal register machine: "push X to L" and "pop L to X". Given input values $X = x$ and $L = l$, the gadget "push X to L" returns the value $X = 0$ and $L = 2^x(2\ell + 1)$. Given input value $L = \langle\!\langle x, \ell \rangle\!\rangle$ and $X = y$, the gadget "pop L to X" returns $X = x$ and $L = l$. Given input $L = 0$, the gadget returns $X = 0$ and $L = 0$.

Slide 9

Gadget: "push X to L"

The gadget "push X to L":



Given input values $X = x$, $L = \ell$ and $Z = 0$, it returns the output values $X = 0$, $L = \langle\!\langle x, \ell \rangle\!\rangle = 2^x(2\ell + 1)$ and $Z = 0$:

Analysing Register Machines

We could define a push gadget in terms of a simpler "multiply L by 2" gadget. Instead, we defined one directly in Slide 9. How do we know that the gadget does what we want? And in general, how can we tell what a register machine does?

Unfortunately, there is no foolproof method to tell us what a register machine does in the general case. In fact (spoiler alert!) we will see that there is no algorithm that tells us if a register machine halts, in general. However, we will only ever ask you to work out what reasonably simple register machines do, so there are a some methods that can help, even if they won't give you an answer in the general case.

A first useful approach to determining what a register machine does is to test it on various inputs. If you see a pattern emerging, that should help you to guess what function is being computed. For instance, with the push gadget we can compute the following resulting values for register L (X and Z always end up at 0) for different inputs:

		ℓ			
		0	1	2	3
	0	1	3	5	7
x	1	2	6	10	14
	2	4	12	20	28

The values in the first row correspond to $2\ell + 1$, and each subsequent row is double the previous one. This suggests that the gadget computes $2^x(2\ell + 1)$. However, it is not proof that this is the case: the above results would also fit if it computed $\frac{1}{2}(x^2 + x + 2)(2\ell + 1)$.

Another approach is to break up the execution of the register machine into bits we understand. For instance, we could reason about push as follows:

Suppose we start with $X = x$, $L = \ell$ and $Z = 0$. When we first hit the L^- instruction, $Z = 1$. If L is not already 0, we will increment Z by two and return to L^-. This increment loop will happen ℓ times (since that is the initial value of L) before L hits 0. At that point, we will have $Z = 1 + 2\ell$.

Next L is incremented Z times — we move Z to L. When we hit the X^- instruction, therefore, $L = 2\ell + 1$ and $Z = 0$. If X is not already 0, we loop back and do it all again, except for the initial increment of Z. Each time we loop from X^- back to itself, the value of L is doubled. This loop happens x times (the initial value of X), so when the loop exits the value of L will be $2^x(2\ell + 1)$.

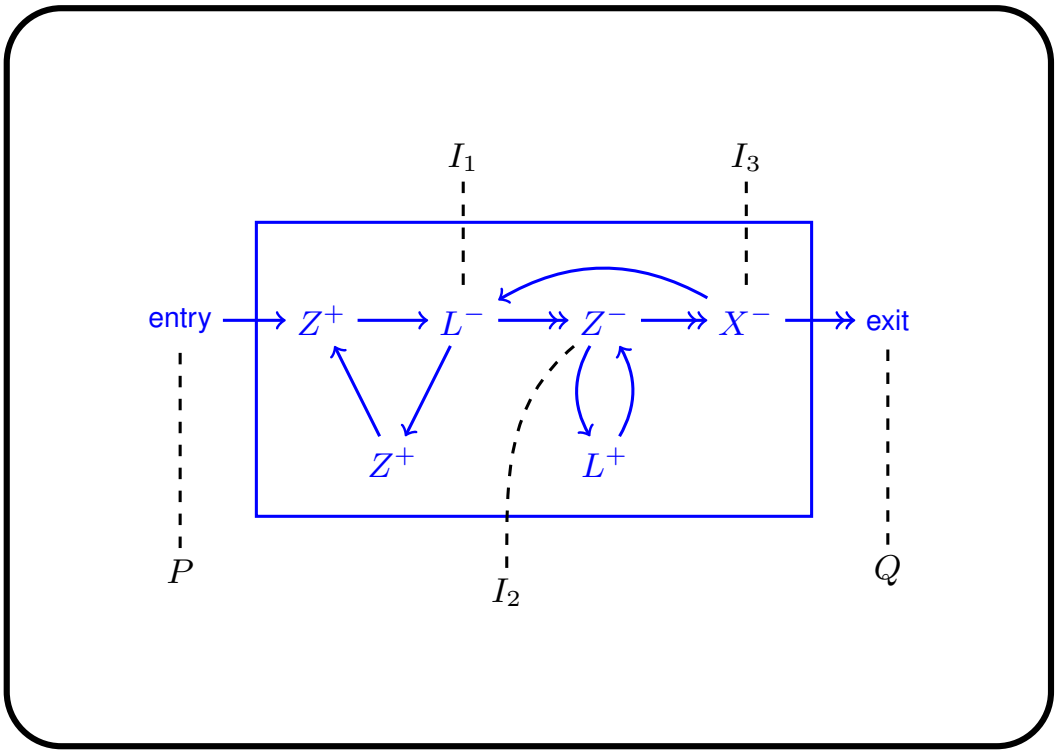
This is probably the most efficient way of convincing yourself of what a register machine does, but we can be a bit more formal about it. One way of doing this is with *invariants*: logical assertions that describe the state of the machine whenever it reaches particular labels, in terms of the initial inputs. The postcondition, which is the invariant at the halting instruction, will tell us about the result of the computation.

For the invariants to be correct, every execution path (for which the precondition invariant held on the initial state) establishes each invariant whenever it reaches the associated label. To check this, it is enough that, whenever one invariant holds and we execute the machine until we reach another label with an invariant, that invariant will hold. (To see this, suppose that we have an execution that eventually breaks an invariant. The path from the last invariant to hold to the invariant that doesn't hold would give a counterexample to the

proposed condition.)

If we want to analyse a register machine, we could label every instruction with an invariant. Usually, however, it is enough to have one invariant for each loop. Making sure each loop has an invariant guarantees that we will only have to consider paths from one invariant to the next that have a bounded length. For the push gadget, we want to find invariants at the points labelled in Slide 10.

Slide 10



As our precondition, we choose $P \equiv (X = x \wedge L = \ell \wedge Z = 0)$. By considering the paths between invariants, we construct a set of constraints on our invariants (called *verification conditions*). For instance, on the path from P to I_1, Z is incremented, while every other register is unchanged. Consequently, we must have that $P[Z - 1/Z] \implies I_1$. (If P held for the old value of Z then $P[Z - 1/Z]$ will hold for the new value, because the old value is one less than the new value.) The other verification conditions

generated in this way are:

$$\begin{array}{rcl@{\qquad}rcl} I_1[L+1/L, Z-2/Z] & \implies & I_1 & I_1 \wedge L = 0 & \implies & I_2 \\ I_2[Z+1/Z, L-1/L] & \implies & I_2 & I_2 \wedge Z = 0 & \implies & I_3 \\ I_3[X+1/X] & \implies & I_1 & I_3 \wedge X = 0 & \implies & Q \end{array}$$

We could just try to guess invariants that satisfy these properties, but we can be a bit more methodical. We will start by guessing all of the invariants (except P) to be $\bot$, and weaken them (make them more general) as necessary to meet the constraints.

The first constraint tells us that $(X = x \wedge L = \ell \wedge Z = 1) \implies I_1$. This doesn't hold for $\bot$, so let's weaken the invariant to

$$I_1 \equiv (X = x \wedge L = \ell \wedge Z = 1)$$

Now the first constraint holds, but, for this I_1 the second constraint requires that $(X = x \wedge L + 1 = \ell \wedge Z - 2 = 1) \implies I_1$. We can weaken the invariant:

$$I_1 \equiv (X = x \wedge L = \ell \wedge Z = 1) \vee (X = x \wedge L + 1 = \ell \wedge Z - 2 = 1)$$

But this still doesn't satisfy the second constraint. If we continue this process, we get:

$$\begin{aligned} I_1 \equiv\ & (X = x \wedge L = \ell \wedge Z = 1) \\ & \vee (X = x \wedge L + 1 = \ell \wedge Z - 2 = 1) \\ & \vee (X = x \wedge L + 2 = \ell \wedge Z - 4 = 1) \end{aligned}$$

Clearly, we could go on adding disjuncts forever without meeting the constraint. What we need to do is *abstract*: come up with a formula that describes all these disjuncts. Fortunately, we can spot a pattern to help us: L goes down one and Z goes up two each time. We can therefore weaken to the following abstracted invariant:

$$I_1 \equiv (X = x \wedge Z + 2L = 2\ell + 1)$$

Now $(X = x \wedge (Z - 2) + 2(L + 1) = 2\ell + 1) \implies (X = x \wedge Z + 2L = 2\ell + 1)$, so this really does satisfy the second condition.

To satisfy the third constraint, we can now pick:

$$I_2 \equiv (X = x \wedge Z = 2\ell + 1 \wedge L = 0)$$

However, for the fourth condition we want to abstract again:

$$\begin{aligned} I_2 \equiv (X = x \wedge Z = 2\ell + 1 \wedge L = 0) \\ \vee (X = x \wedge Z + 1 = 2\ell + 1 \wedge L - 1 = 0) \\ \vee (X = x \wedge Z + 2 = 2\ell + 1 \wedge L - 2 = 0) \vee \ldots \end{aligned}$$

Giving:

$$I_2 \equiv (X = x \wedge Z + L = 2\ell + 1)$$

For the fifth condition we get:

$$I_3 \equiv (X = x \wedge L = 2\ell + 1 \wedge Z = 0)$$

The sixth condition takes us back to I_1:

$$I_1 \equiv (X = x \wedge Z + 2L = 2\ell + 1) \vee (X + 1 = x \wedge L = 2\ell + 1 \wedge Z = 0)$$

It's not very clear how to abstract this yet, so let's apply the second condition a few times.

$$\begin{aligned} I_1 \equiv (X = x \wedge Z + 2L = 2\ell + 1) \\ \vee (X + 1 = x \wedge L = 2\ell + 1 \wedge Z = 0) \\ \vee (X + 1 = x \wedge L + 1 = 2\ell + 1 \wedge Z - 2 = 0) \\ \vee (X + 1 = x \wedge L + 2 = 2\ell + 1 \wedge Z - 4 = 0) \end{aligned}$$

We can now see how to abstract everything but the first disjunct:

$$I_1 \equiv (X = x \wedge Z + 2L = 2\ell + 1) \vee (X + 1 = x \wedge Z + 2L = 2(2\ell + 1))$$

This now meets the second condition, so we could continue round the big loop until we have to abstract again. However, let's use a bit of intuition and guess that we're going to double the $Z + 2L$ part each time round that loop:

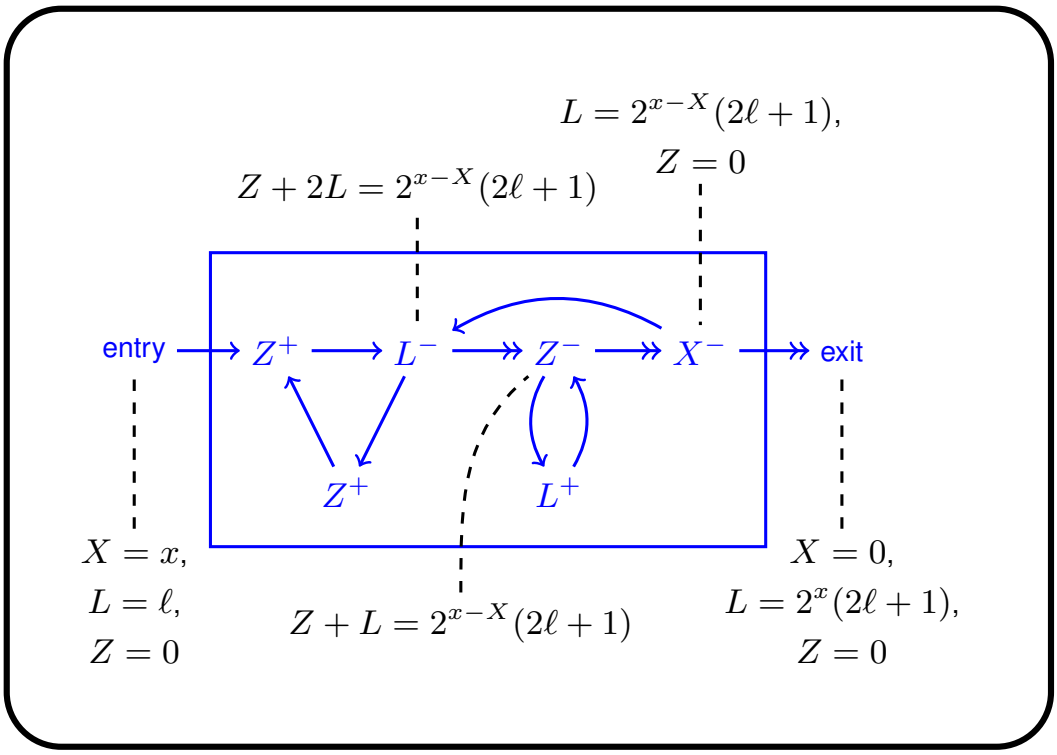
$$I_1 \equiv (Z + 2L = 2^{x-X}(2\ell + 1))$$

This meets the second condition. Continuing, we get

$$\begin{aligned} I_2 &\equiv (Z + L = 2^{x-X}(2\ell + 1)) \\ I_3 &\equiv (L = 2^{x-X}(2\ell + 1) \wedge Z = 0) \\ Q &\equiv (X = 0 \wedge L = 2^x(2\ell + 1) \wedge Z = 0) \end{aligned}$$

These meet all of the conditions, so we know that when the gadget exits $L = 2^x(2\ell + 1)$, exactly as we want.

Slide 11

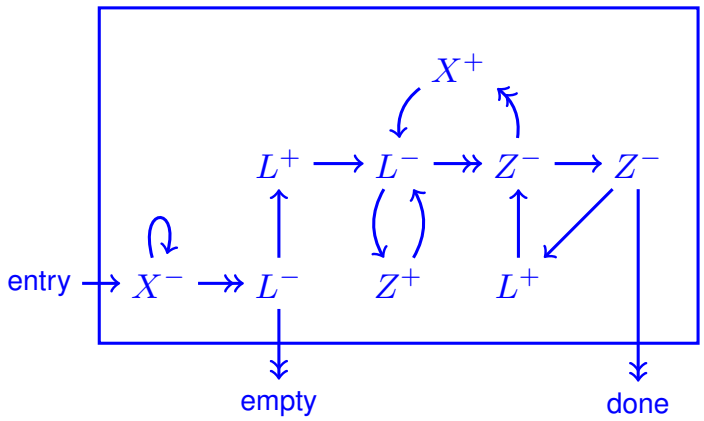


We haven't actually shown that the gadget will eventually exit, however. It could just run forever. To show that it does terminate, we should find some variant that decreases each time an invariant is visited. For I_1, we can use (X, L) (with lexicographic ordering). Each time it is revisited, X will have decreased, or X will have stayed the same but L decreased. For I_2, we can use (X, Z), and for I_3 we can use X.

Slide 12

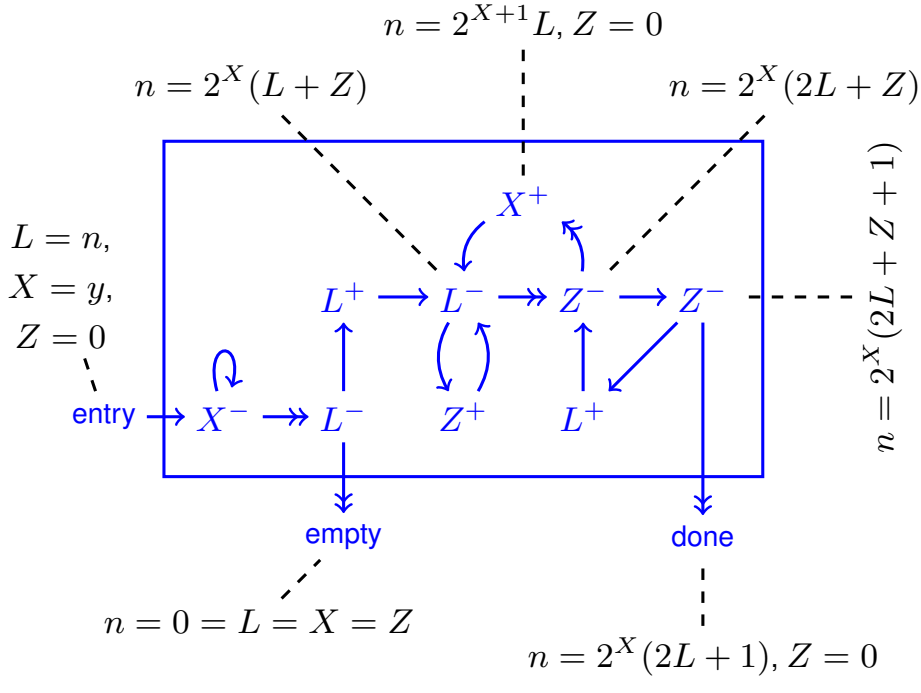
Gadget: "pop L to X"

The gadget "pop L to X":

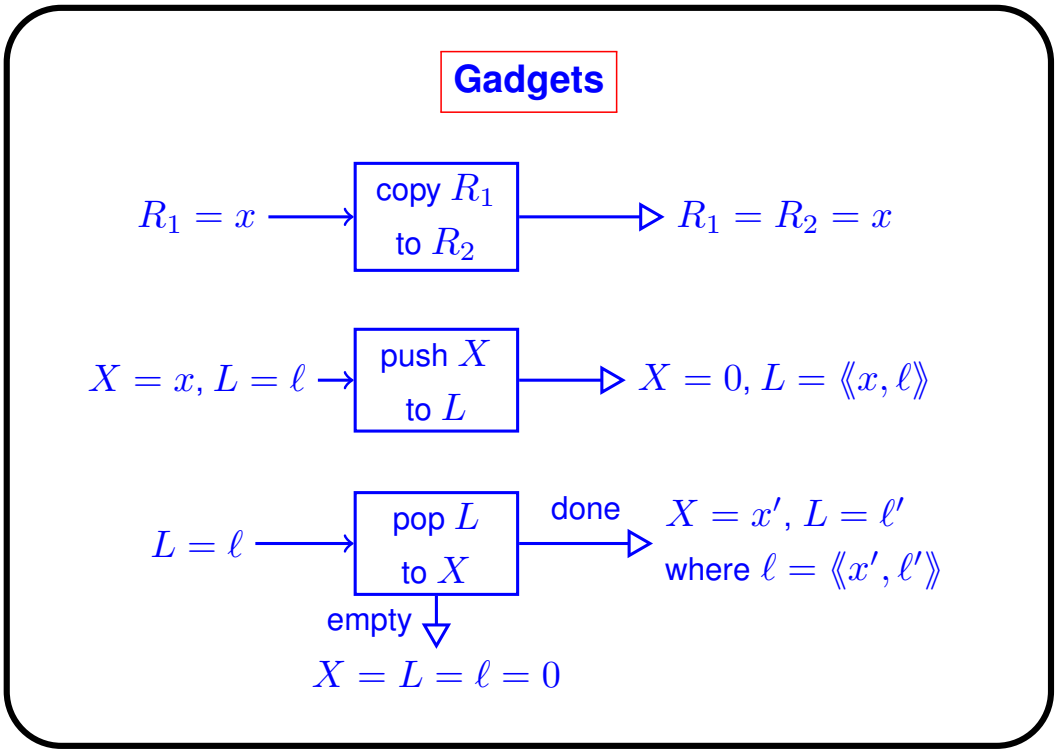


If $L = 0$ then return $X = 0$ and go to "empty". If $L = \langle\langle x, \ell \rangle\rangle = n$ then return $X = x$ and $L = \ell$, and go to "done".

Slide 13



Slide 14



The Universal Register machine

A universal register machine (URM) is a register machine that can simulate an arbitrary register machine on arbitrary input. It achieves this by reading the code (unique description) of the machine to be simulated and the code (unique description) of the input.

Slide 15

The Universal Register Machine

The *universal register machine* carries out the following computation, starting with $R_0 = 0$, $R_1 = e$ (code of a program), $R_2 = a$ (code of a list of arguments) and all other registers zeroed:

- decode e as a RM program P
- decode a as a list of register values $a_1, \ldots, a_n$
- carry out the computation of the RM program P starting with $R_0 = 0, R_1 = a_1, \ldots, R_n = a_n$ (and any other registers occurring in P set to 0).

Slide 16

Mnemonics for the registers of U and the role they play in its program:

R_0 result of the simulated RM computation (if any).

$R_1 \equiv P$ Program code of the RM to be simulated

$R_2 \equiv A$ list of RM Arguments (or register contents) of the simulated machine

$R_3 \equiv PC$ Program Counter—label number of the current instruction

$R_4 \equiv N$ label number(s) of the Next instruction(s)—also used to hold code of current instruction

$R_5 \equiv C$ code of the Current instruction body

$R_6 \equiv R$ value of the Register to be used by current instruction

$R_7 \equiv S$ and $R_8 \equiv T$ are auxiliary registers.

R_9**...** other scratch registers.

Slide 17

Overall structure of the URM

1 copy PCth item of list in P to N (halting if $PC >$ length of list); goto **2**

2 if $N = 0$ then halt, else decode N as $\langle\langle y, z \rangle\rangle$; $C ::= y$; $N ::= z$; goto **3**

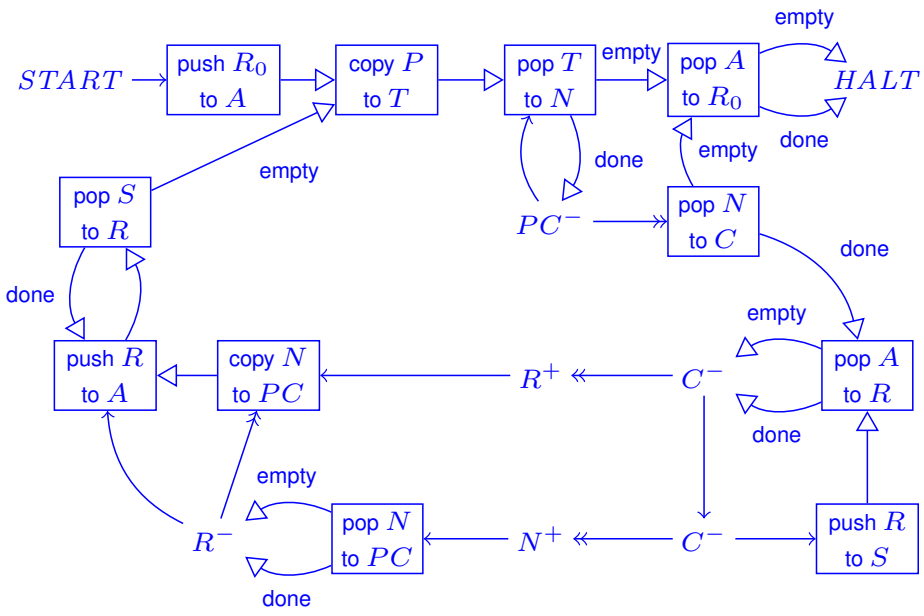
{at this point either $C = 2i$ is even and current instruction is $R_i^+ \to L_z$, or $C = 2i + 1$ is odd and current instruction is $R_i^- \to L_j, L_k$ where $z = \langle j, k \rangle$}

3 copy ith item of list in A to R; goto **4**

4 execute current instruction on R; update PC to next label; restore register values to A; goto **1**

Slide 18

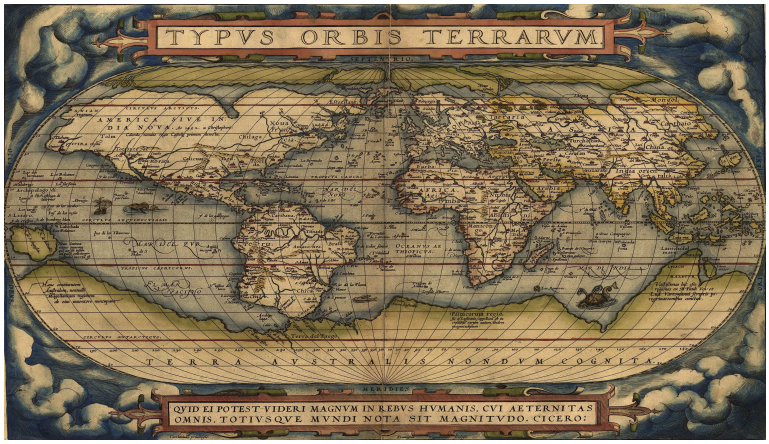
The Universal Register Machine



Slide 19

Universal Register Machines

Ivan Korec: *Small Universal Register Machines*. Theoretical Computer Science, Volume 168 (1996), pp267–301.



Ortelius: *Typvs Orbis Terrarvm* 1570. Wikimedia Commons