50003 - Models Of Computation - (Prof Wiklicky) Lecture 2

Oliver Killane

31/03/22

Lecture Recording

Lecture recording is available here

Gadgets

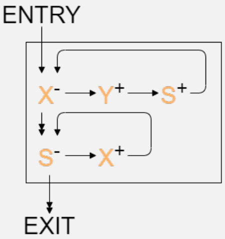
Definition: Register Machine Gadget

A gadget is a partial register machine graph, used as components in more complex programs, that can be composed into larger register machines or gadgets.

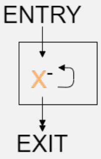
- Has a single *ENTRY* (much like *START*).
- Can have many *EXIT* (much like **HALT**).
- Operates on registers specified in the name of the gadget (e.g *"Add R_1 to R_2"*).
- Can use scratch registers (assumed to be zero prior to gadget and set to zero by the gadget before it exits - allows usage in loops)
- We can rename the registers used in gadgets (simply change the registers used in the name (*push R_0 to R_1* → *push X to Y*), and have all scratch registers renamed to registers unused by other parts of the program)

For example we can create several gadgets in terms of registers that we can rename.

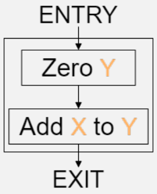
Add X to Y



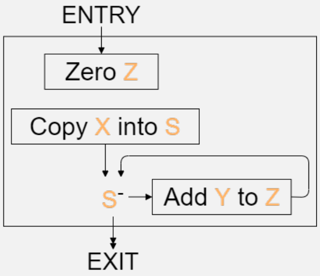
Zero X



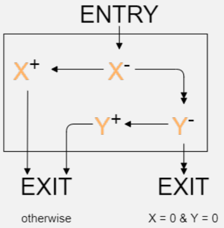
Copy X into Y



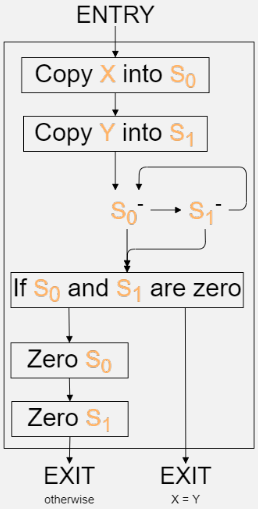
Multiply X by Y to Z



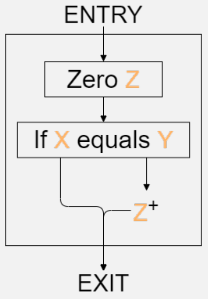
If X and Y are zero



If X equals Y



If X equals Y set Z



And then can use these to create larger programs.

Analysing Register Machines

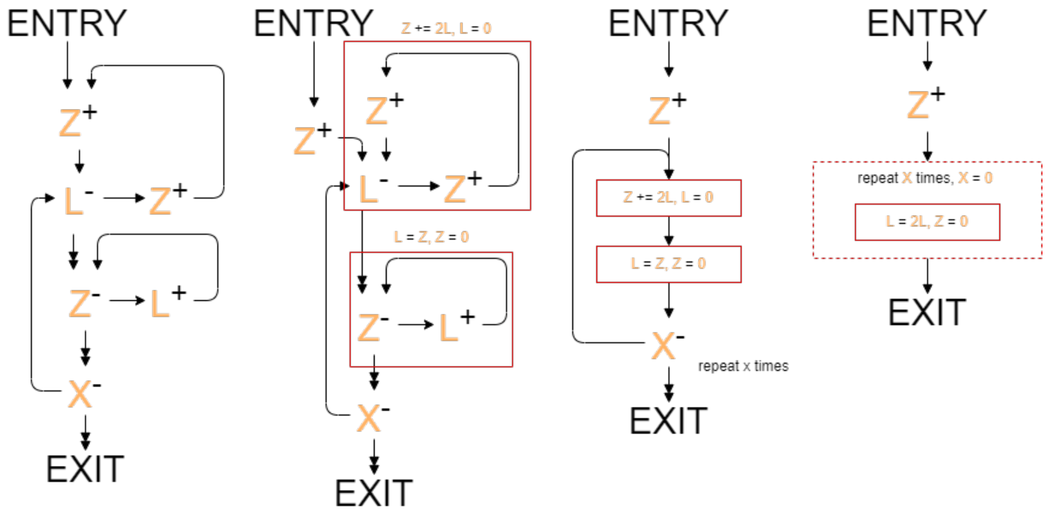
There is no general algorithm for determining the operations of a register machine (i.e halting problem)

However there are several useful strategies one can use:

- **Experimentation**
  Can create a table of input values against outputs to attempt to fetermine the relation - however the values could match many different relations.

- **Creating Gadgets**
  We can group instructions together into gadgets to identify simple behaviours, and continue to merge to develop an understanding of the entire machine.

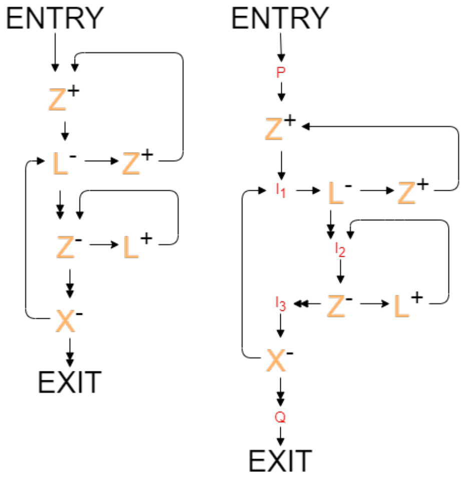
  For example below, we can deduce the result as $L = 2^X(2L+1)$



- **Invariants**
  We can use logical assertions on the register machine state at certain instructions, both to get the result of the register machine, and to prove the result.

  If correct, every execution path to a given instruction's invariant, establishes that invariant.

  We could attach invariants to every instruction, however it is usually only necessary to use them at the start, end and for loops (preconditions/postconditions).



Our first invariant (P) can be defined as:

$$P \equiv (X = x \wedge L = l \wedge Z = 0)$$

Next we can use the instructions between invariant to find the states under which the invariants must hold.

1.	$P[Z-1/Z] \Rightarrow I_1$	After incrementing Z needs to go to the start of the first loop.
2.	$I_1[L+1/L, Z-2/Z] \Rightarrow I_1$	The loop decrements L and increases Z by two. After each loop iteration, I_1 must still hold.
3.	$I_1 \wedge L = 0 \Rightarrow I_2$	If $L = 0$ the loop is escaped, and we move to I_2.
4.	$I_2[Z+1/Z, L-1/L] \Rightarrow I_2$	Loop increments L and decrements Z on each iteration, after this, I_2 must still hold.
5.	$I_2 \wedge Z = 0 \Rightarrow I_3$	Loop ends when $Z = 0$, moves to I_3.
6.	$I_3[X+1/X] \Rightarrow I_1$	Large loop decrements X on each iteration, invariant must hold with the new/decremented X.
7.	$I_3 \wedge X = 0 \Rightarrow Q$	When the main X-decrementing loop is escaped, we move to exit, so Q must hold.

We can now use these constraints (also called **verification conditions**) to determine an invariant.

For each constraint we do:

1. Get the basic for (potentially one already derived) for the invariant in question.
2. If there is iteration, iterate to build up a disjunction.
3. Find the pattern, and then re-form the invariant based on it.

Constraint 1.

Hence we can deduce I_1 as:

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

(Take P and increment Z)

Constraint 2.

We can iterate to get the disjunction:

$I_1 \equiv (X = x \wedge L = l \wedge Z = 1) \vee (X = x \wedge L + 1 = l \wedge Z - 2 = 1) \vee (X = x \wedge L + 2 = l \wedge Z - 4 = 1) \vee \dots$

Hence we can determine the pattern for each disjunct as:

$$Z + 2L = 2l + 1$$

Hence we create our invariant:

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

Constraint 3.

Hence as $L = 0$ we can determine that $Z = 2l + 1$.

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

Constraint 4.

We iterate to get the disjunction:

$I_2 = (X = x \wedge Z = 2l + 1 \wedge L = 0) \vee (X = x \wedge Z = 2l + 0 \wedge L = 1) \vee (X = x \wedge Z = 2l - 1 \wedge L = 2) \vee \dots$

Hence we notice the pattern:

$$Z + L = 2l + 1$$

So can deduce the invariant:

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

Constraint 5.

We can derive an invariant I_3 using $Z = 0$.

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

Constraint 6.

We can use the constraint, and the currently derived I_1 to get a disjunction:

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

We can apply constraint **2.** on the first part of this disjunction, iterating to get the disjunction:

$$I_1 = (X = x \wedge Z + 2L = 2l + 1) \vee \begin{pmatrix} (X = x - 1 \wedge L = 2l + 1 \wedge Z = 0)\vee \\ (X = x - 1 \wedge L = 2l + 0 \wedge Z = 2)\vee \\ (X = x - 1 \wedge L = 2l - 1 \wedge Z = 4)\vee \\ (X = x - 1 \wedge L = 2l - 2 \wedge Z = 8) \vee \dots \end{pmatrix}$$

Hence for the second group of disjuncts we have the relation:

$$Z + 2L = 2(2l + 1)$$

Hence we have:

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

Hence when we repeat on the larger loop, we will double again, iterating we get:

$$I_1 = (X = x \wedge Z + 2L = 2l + 1) \vee (X = x - 1 \wedge Z + 2L = 2(2l + 1)) \vee (X = x - 2 \wedge Z + 2L = 4(2l + 1)) \vee \dots$$

Hence we have the relation:

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

We can apply this doubling to L_2 also as it forms part of the larger loop:

$$I_2 = (Z + L = 2^{X-x}(2l + 1))$$

And to I_3:

$$I_3 = (L = 2^{X-x}(2l + 1) \wedge Z = 0)$$

Constraint 7.

Hence we can now derive Q as:

$$Q = (L = 2^{x}(2l + 1) \wedge Z = 0)$$

Termination

We also need to show that each of our loops eventually terminate, we can do this by showing that sme variant (e.g register, or combination of) decreases every time the invariant is reached/visited.

For I_1 we can use the lexicographical ordering (X, L) as in each inner loop L decreases, but for the larger loop while L is reset/does not increase, X does.

For I_2 we can similarly use the lexicographical ordering (X, Z)

For I_3 we can just use X.

Universal Register Machine

A register machine that simulates a register machine.

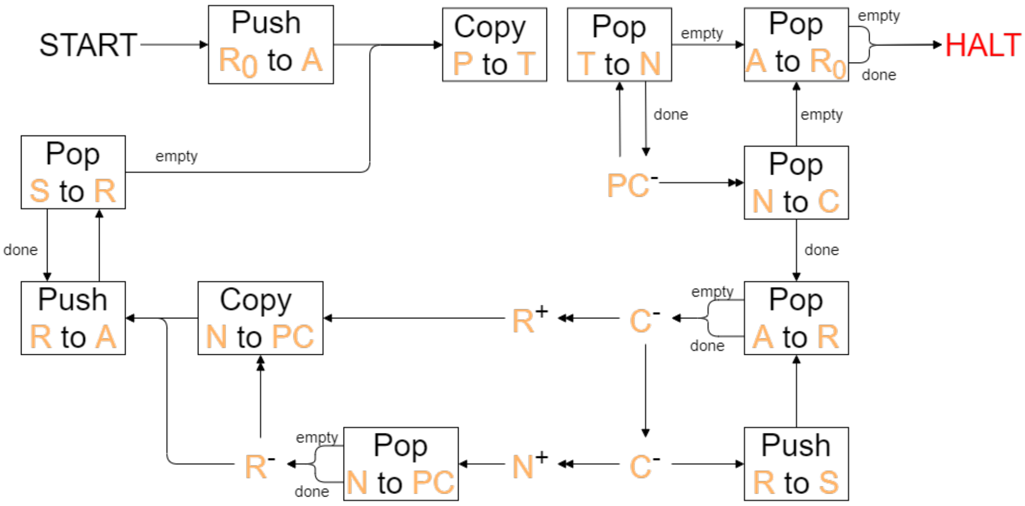
It takes the arguments:

- R_0 = 0
- R_1 = the program encoded as a number
- R_2 = the argument list encoded as a number
- **All other registers zeroed**

The registers used are:

R_1	P	Program code of the register machine being simulated/emulated.
R_2	A	Arguments provided to the simulated register machine.
R_3	PC	Program Counter - The current register machine instruction.
R_4	N	Next label num,ber/next instruction to go to. Is also used to store the current instruction
R_5	C	The current instruction.
R_6	R	The value of the register used by the current instruction.
R_7	S	Auxiliary Register
R_8	T	Auxiliary Register
$R_9 \ldots$		Scratch Registers

```
while true:
    if PC >= length P:
        HALT!

    N = P[PC]

    if N == 0:
        HALT!

    (curr, next) = decode(N)
    C = curr
    N = next

    # either C = 2i (R+) or C = 2i + 1 (R-)
    R = A[C // 2]

    # Execute C on data R, get next label and write back to registers
    (PC, R_new) = Execute(C, R)
    A[C//2] = R_new
```



START
Push R_0 to A
Copy P to T
Pop T to N
empty
Pop A to R_0
empty
HALT
done
done
empty
PC⁻
Pop N to C
Pop S to R
empty
done
done
Push R to A
Copy N to PC
R⁺
C⁻
empty
done
Pop A to R
R⁻
empty
done
Pop N to PC
N⁺
C⁻
Push R to S