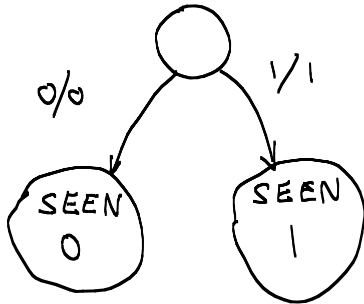
Registers

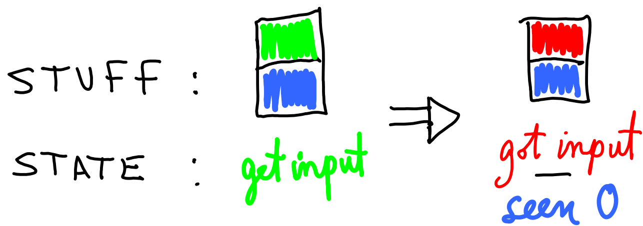


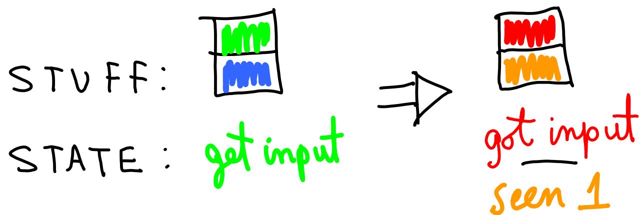
"SEEN 0" registers that we've seen a 0,

"SEEN 1" registers that we've seen a 1.

STATE has two parts:
-- step of operation
-- symbol seen

A real machine has a PHYSICAL state, physical stuff that changes in time.





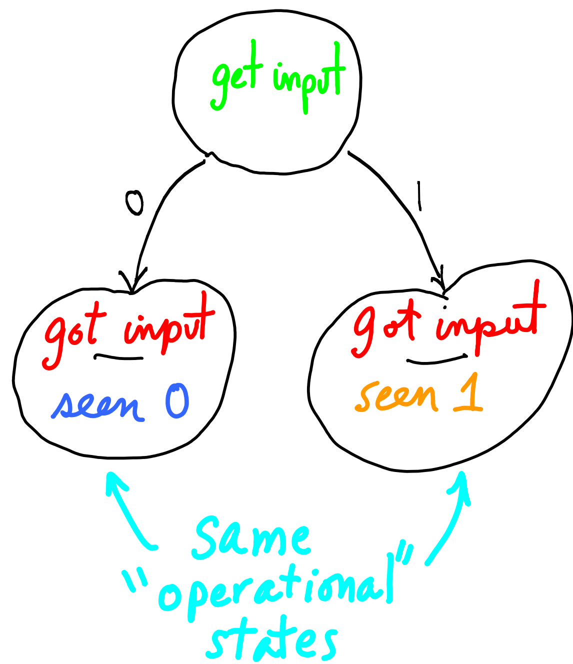
TM's FSM (the CONTROL part of a TM)

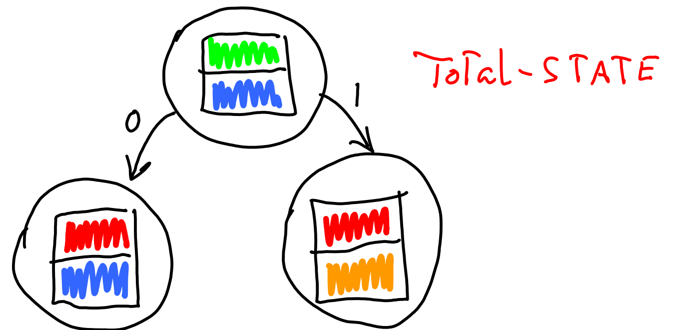
No tape, one input, two outputs:
-- IN-SYMBOL, OUT-SYMBOL, MOVE

For FSM, inputs/outputs are time series:

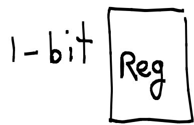
time:	0	1	2	3	4	5	6 ...
input :	0	0	0	1	1	1	0 ...
output:	0	1	1	0	1	0	1 ...
move :	0	1	1	0	1	0	1 ...

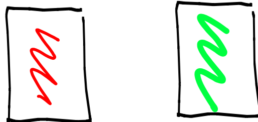
For a physical state machine, we must talk about time explicitly, physical state changes in time.

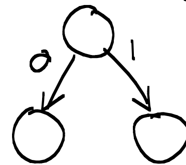


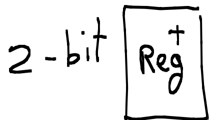


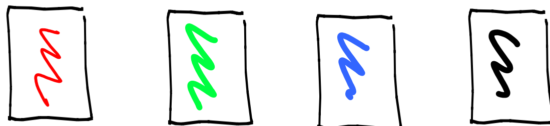
Hardware | possible states | state diagram

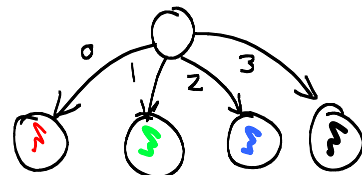






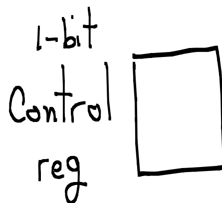


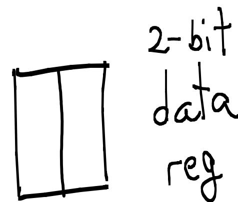




1-bit register ==> 2^1 == 2 states
k-bit register ==> 2^k states

2 1-bit registers ==> 2^(2*1) == 4 states
2 k-bit registers ==> 2^2k states

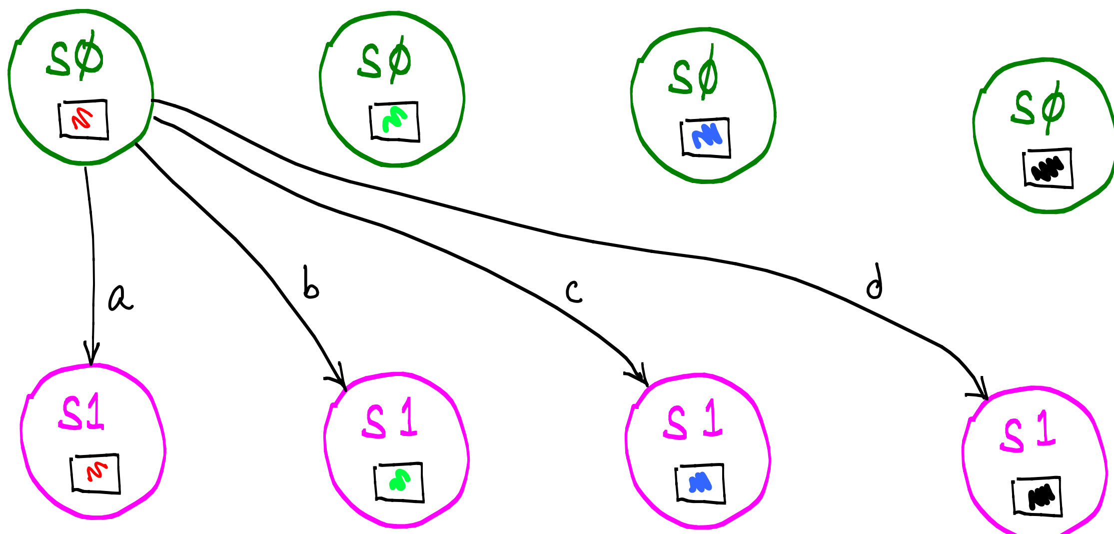




Suppose physical state is composed of one 2-state element and one 4-state element. Altogether, we have a physical system with eight possible states.

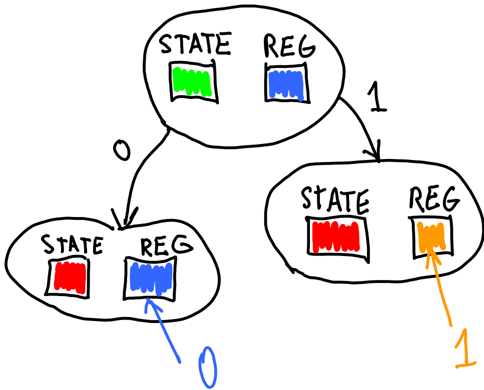
S0: Get data

S1: data registered



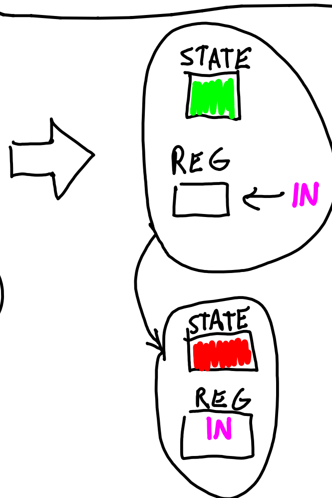
(2-state control) X (4-state data reg) == 8 complete states possible. Before reading input, we don't care which of top 4 states we start in, we know we have not yet registered some data: we are in the "get-data" control state. We are in the "data-registered" control state after. (What if 32-bit symbols ==> 4G branches; something we'd like to hide.)

"Total State"



Complete state:
(1) ready-for-data-and-reg-is-zero,
(2) got-data-and-reg-is-zero
(3) got-data-and-reg-is-one

"Control State" + "Data Register"



Control state:
(1) get-data
(2) got-data

BIG IDEA:

SPLIT TOTAL STATE into two parts:

--- 1. OPERATIONAL STATE
Where we are in doing things

--- 2. DATA REGISTER STATE
What we know at this point

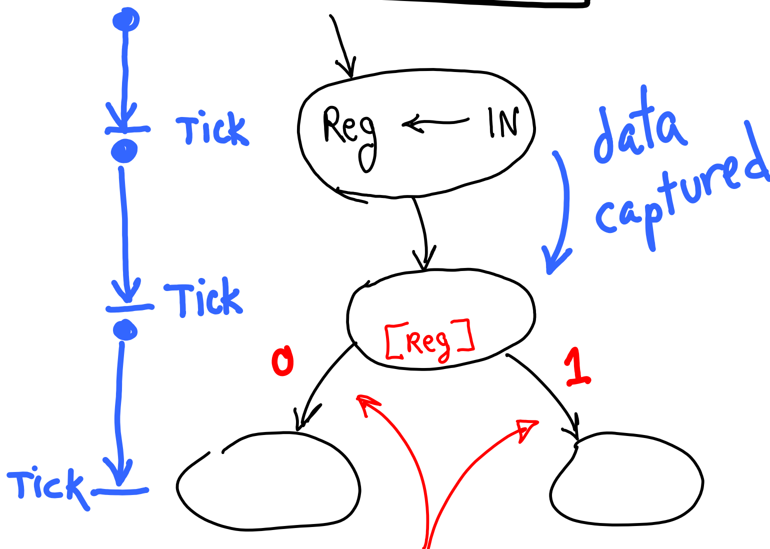
registering a 32-bit input symbol :

-- TOTAL STATE:
(2 Operational states) X (4G data states)

versus

-- 2 Operational states + content of reg.

Control Branching



data used to select next state

Change "operational state" depending register content.

-- States associated with Register Transfer operations (described using RTL).

-- Branches labeled w/ register content (or partial content).

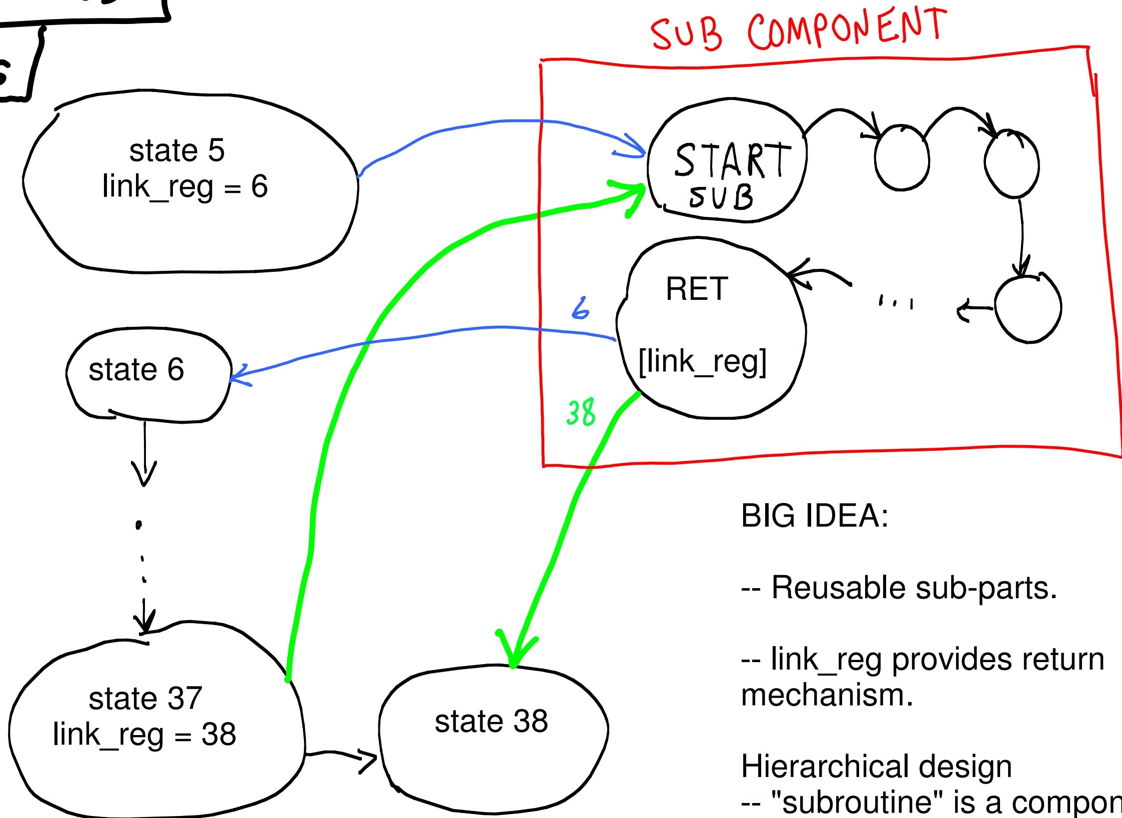
CLOCK causes:

---- register data transfers
---- control state changes

USE REGISTERS for BOTH
---- "STATE" registers
---- "DATA" registers

---- BOTH types change w/ CLOCK

Subroutines
sub-units
HW/SW



BIG IDEA:

-- Reusable sub-parts.

-- link_reg provides return mechanism.

Hierarchical design
-- "subroutine" is a component
-- components connected

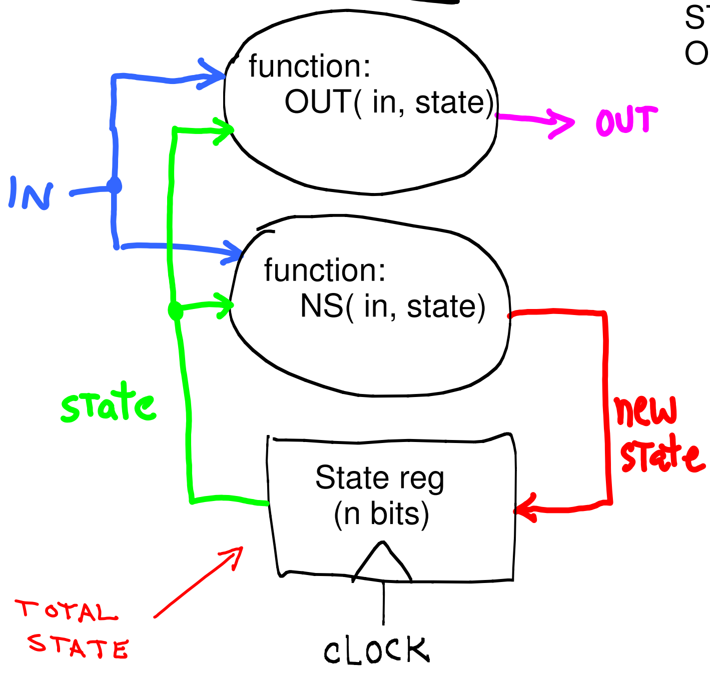
Same as putting one TM into another.

HW = SW
-- Machines or descriptions of machines can have hierarchical design.

- 2 different "entries" into Sub-component's START state
- 2 different "exits" from Sub-component's HALT state

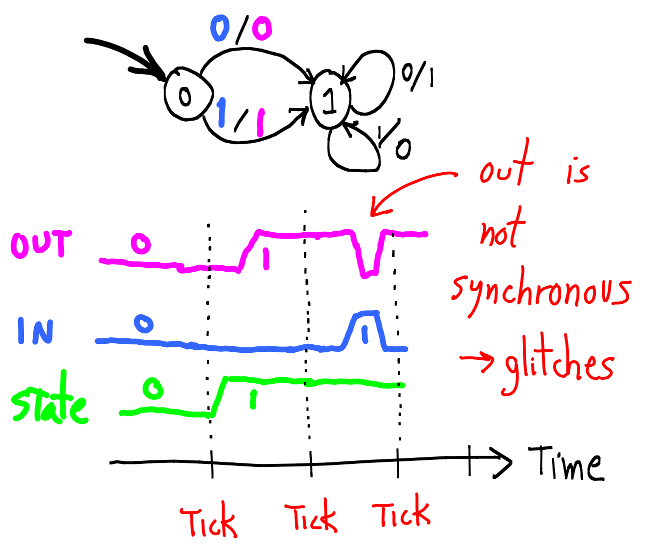
⟹ Reusable sub-"Routine"

Implementation of FSM

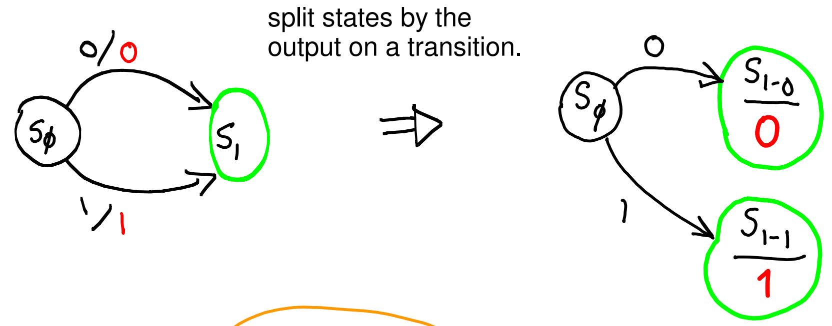


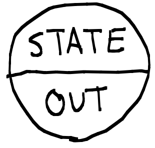
MEALY MACHINE

OUT is continous function of IN+STATE.
STATE changes w/ clock
OUT changes w/ changes in STATE and/or IN

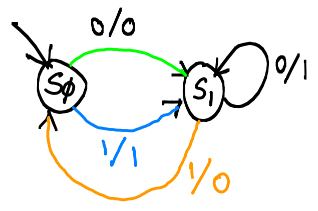


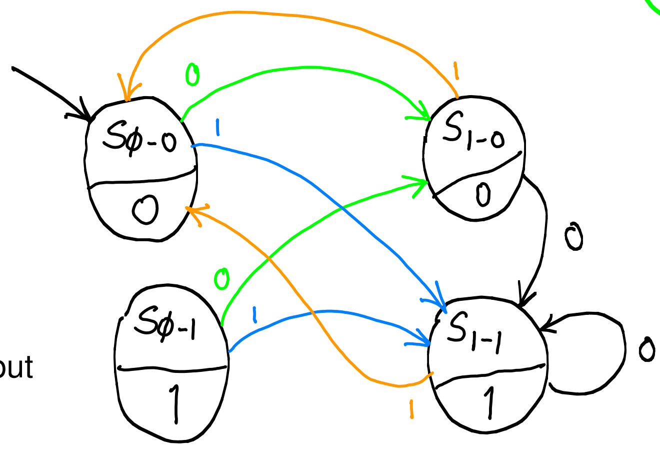
synchronous output → Moore Machine





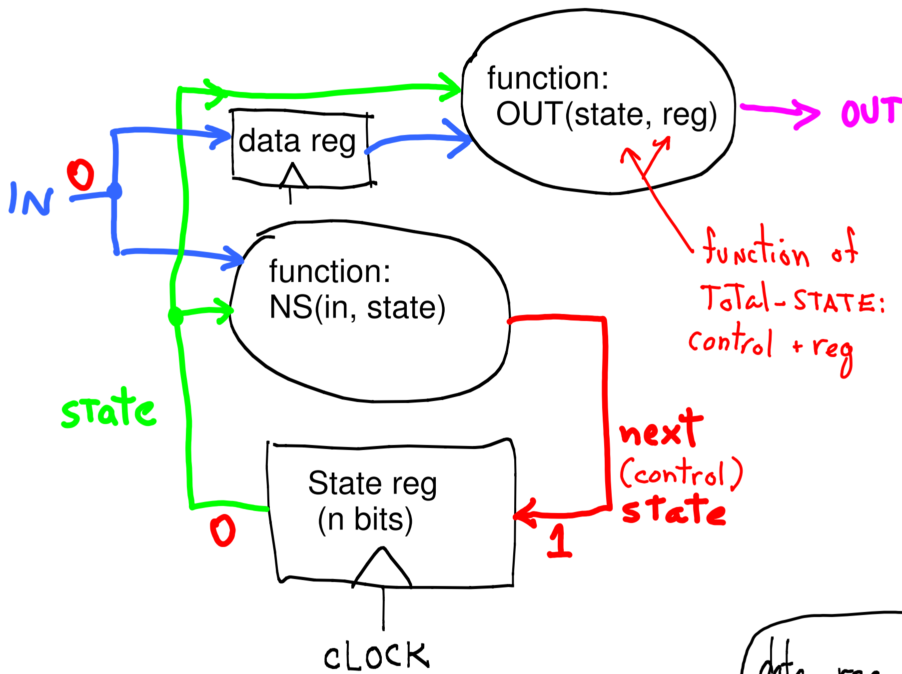
Output while in a specific state is always the same, regardless of input changes: output only depends on current state.





Are they equivalent? Check that same input streams give same output streams.

Moore Machine



Moore machine:

AT CLOCK TICK:

-- data_reg output changes:

data_reg.out == IN

-- State_reg output changes:

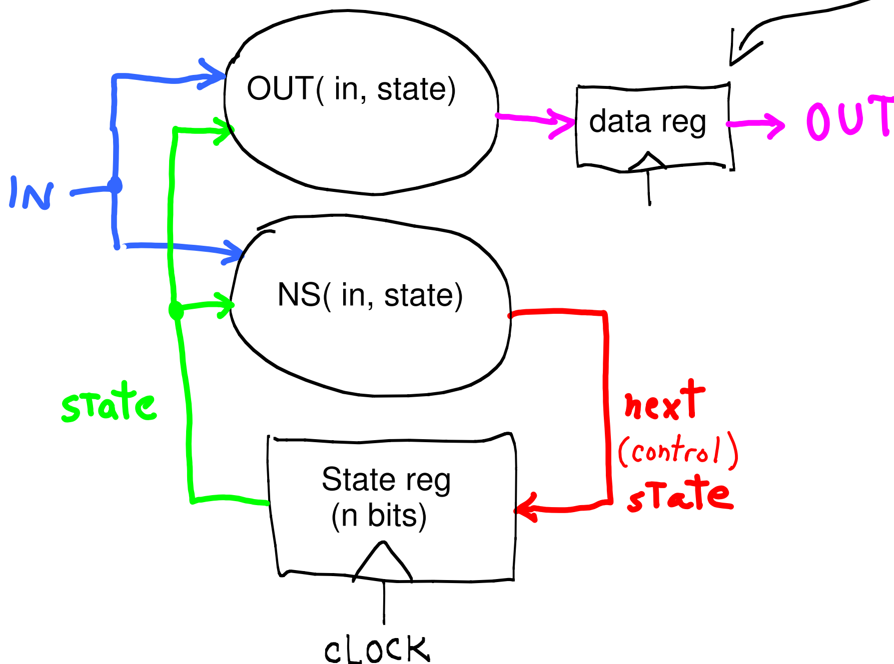
State_reg.out == NS(in, state)

No changes until next tick,
even if input changes.

data reg + state reg = total state

Shortcut Mealy-to-Moore Conversion

Add output reg



A Moore machine:

--- **OUT** changes w/ clock tick, when total STATE changes, and does not change until next tick.

--- **OUT** is a function of total STATE (control and data registers); input does not affect OUT. In this case, it is not dependent on current control state, but previous control state.

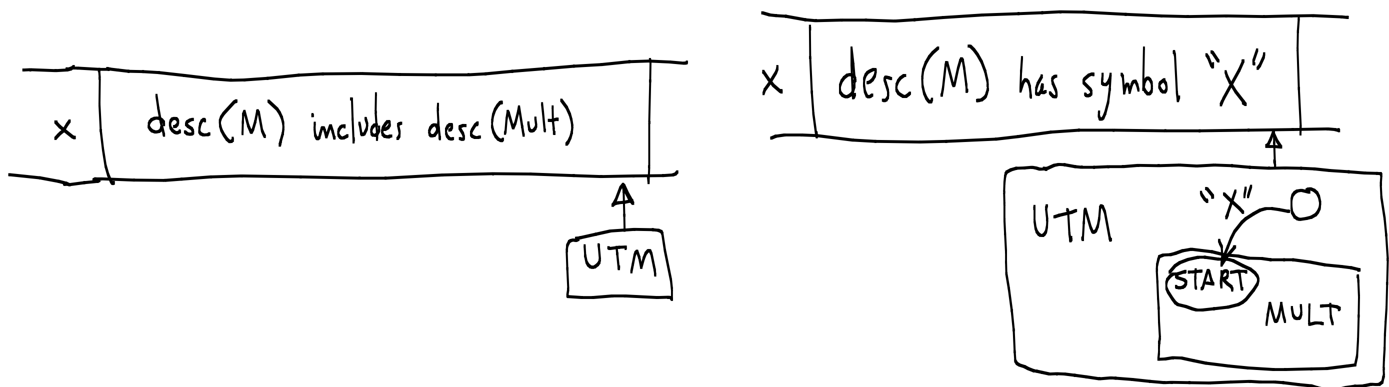
BIG IDEA: Extend simulator with additional hardware (hardware subroutines).

We can add hardware to our computer/simulator:

--- Our description of a machine can have a sub-routine for MULTIPLY,

--- OR, we could add a symbol "X" that causes our simulator/computer to branch to a hardware subroutine: FASTER. Desc(M) is also SMALLER. (*but is computer then slower?)

-- Software == Hardware



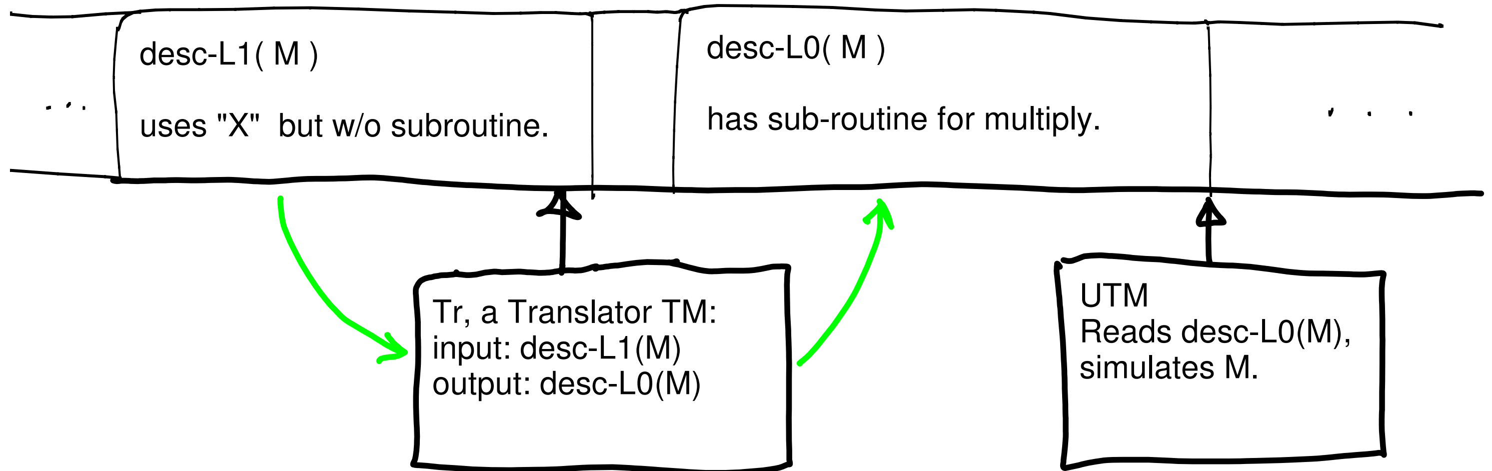
Could we extend the desc(M) in the same way?

Could we have some TM, T, described as part of the description of M?

Perhaps we wouldn't even put desc(T) into desc(M)?

Only put an indicator that desc(T) should be inserted into desc(M)?

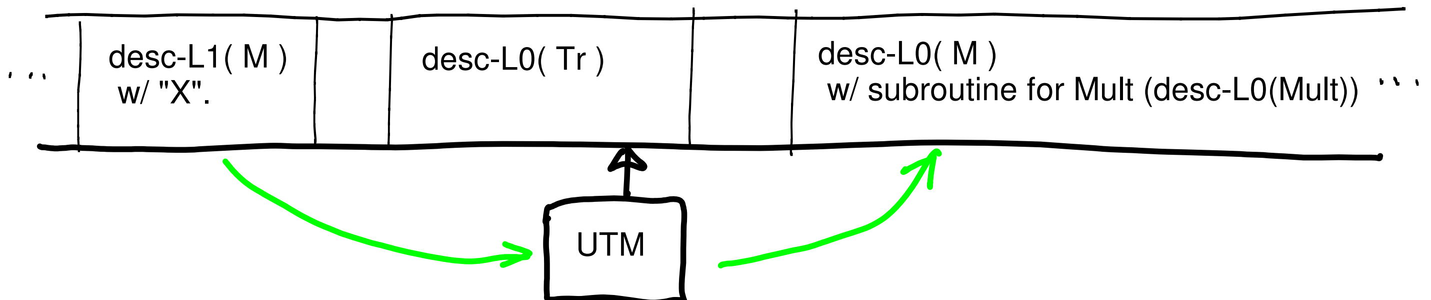
Hierarchical Translation



Tr
1. reads desc-L1(M),
2. sees "X" in rule in desc-L1(M)
   --- writes desc-L0(Mult) as part of
   --- desc-L0(M), fixing up state transistions accordingly

L0: "Instruction Set Architecture", ISA, is language of simulating machine, UTM.

Recall, we only need a description of Tr

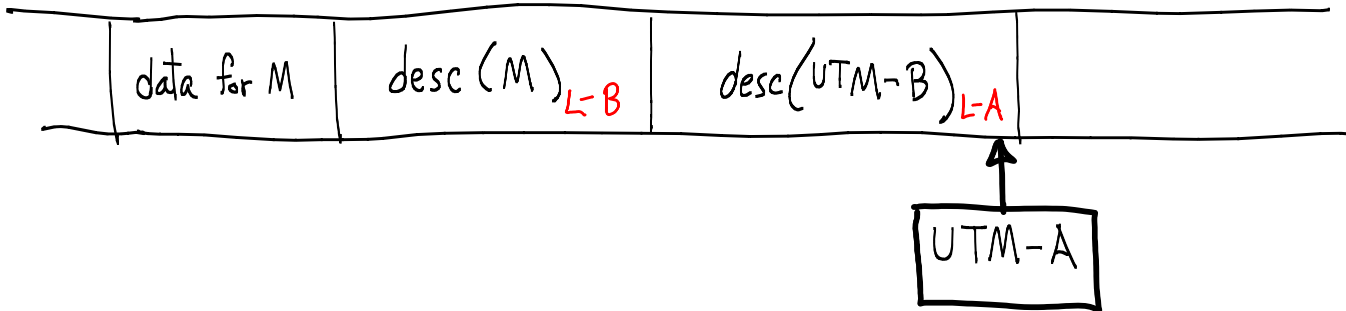


UTM simultates Tr, then simulates M. We can,

--- Add levels: desc-L2(M), and L2-L1 translator.
   --- Eg., scripting language => C++ => C => asm => ISA
--- Migrate subroutines down to lower levels, eventually into UTM's hardware.
--- UTM-0 simulates different UTM-1: desc-L0(UTM-1), UTM-1 has its own ISA.
   ==> interpreted languages, JAVA bytecode.

Interpreter

Simulating a UTM?
Use UTM-A and desc-LA(UTM-B)
==> Simulate UTM-B simulating M.



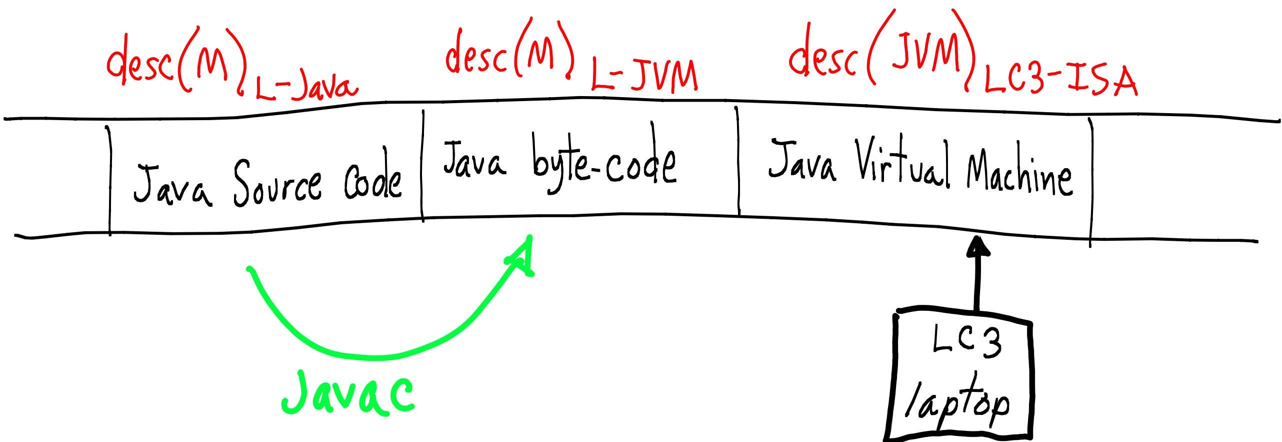
uses language L-A

UTM-A understands descriptions written in L-A.

UTM-B understands descriptions written in L-B.

==> desc(UTM-B) is written in L-A
==> desc(M) is written in L-B

UTM-A simulates UTM-B simulating M.



Let's make things even more exciting!

Have symbol "X" in L-Java,
keep symbol "X" in L-JVM,
have Java-Virtual-Machine jump to
desc-LC3(TM-for-symbol-"X") ===> precompiled, faster than simulating
OR
keep "X" in desc-LC3(TM-for-symbol-"X")
jump to LC3 hardware-sub-TM-for-symbol-"X" (hardware sub-routine)!