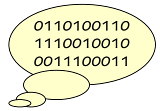
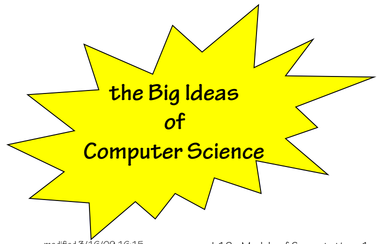
6.004 Computation Structures
Spring 2009

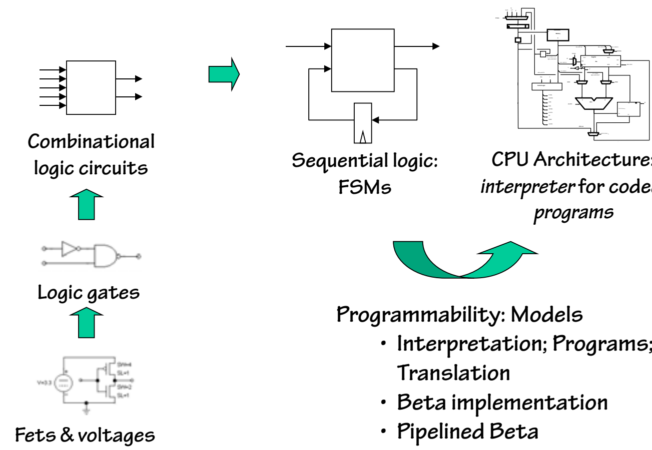
Programmability

from silicon to bits





6.004 Roadmap



Programmability: Models

- Interpretation; Programs; Languages; Translation
- Beta implementation
- Pipelined Beta
- Software conventions
- Memory architectures

FSMs as Programmable Machines

ROM-based FSM sketch:

Given i, s, and o, we need a ROM organized as:

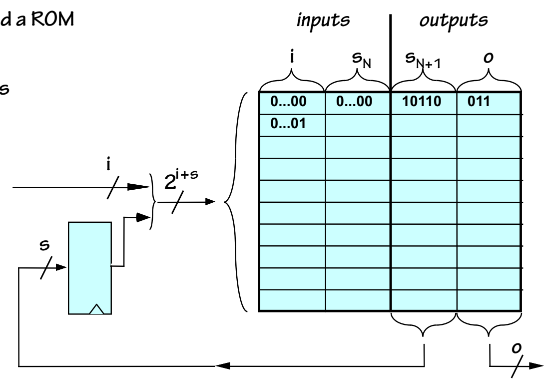
2^{i+s} words x (o+s) bits

So how many possible i-input, o-output, FSMs with s-state bits exist?

$$2^{(o+s)2^{i+s}}$$

(some may be equivalent)

An FSM's behavior is completely determined by its ROM contents.

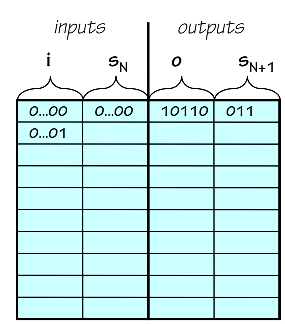


Big Idea #1:

FSM Enumeration

GOAL: List all possible FSMs in some canonical order.

- INFINITE list, but
- Every FSM has an entry and an associated index.



i	s	o	FSM#	Truth Table	
1	1	1	1	00000000	2^8 FSMs
1	1	1	2	00000001	
			...	...	
1	1	1	256	11111111	
2	2	2	257	000000...000000	2^{64}
2	2	2	258	000000...000001	
			...	...	
3	3	3		000000...000000	
			...		
4	4	4		000000...000000	

What if s=2, i=o=1??

Every possible FSM can be associated with a number. We can discuss the i^{th} FSM

Some Perennial Favorites...

FSM_{837}	modulo 3 counter
FSM_{1077}	4-bit counter
FSM_{1537}	lock for 6.004 Lab
FSM_{89143}	Steve's digital watch
$FSM_{22698469884}$	Intel Pentium CPU – rev 1
$FSM_{784362783}$	Intel Pentium CPU – rev 2
$FSM_{72698436563783}$	Intel Pentium II CPU

Reality: The integer indexes of actual FSMs are *much bigger* than the examples above. They must include enough information to constitute a *complete description* of each device's unique structure.

Models of Computation

The roots of computer science stem from the study of many alternative mathematical "models" of computation, and study of the classes of computations they could represent.

An elusive goal was to find an "ultimate" model, capable of representing all practical computations...

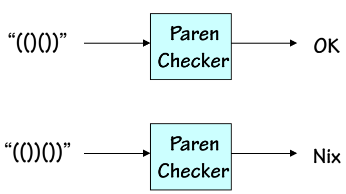
- switches
- gates
- combinational logic
- memories
- FSMs

We've got FSMs ... what else do we need?

Are FSMs the ultimate digital computing device?

FSM Limitations

Despite their usefulness and flexibility, there exist common problems that cannot be computed by FSMs. For instance:



Well-formed Parentheses Checker:

Given any string of coded left & right parens, outputs 1 if it is balanced, else 0.

Simple, easy to describe.

Is this device equivalent to one of our enumerated FSMs???

NO!

PROBLEM: Requires ARBITRARILY many states, depending on input. Must "COUNT" unmatched LEFT parens. An FSM can only keep track of a finite number of unmatched parens: for every FSM, we can find a string it can't check.

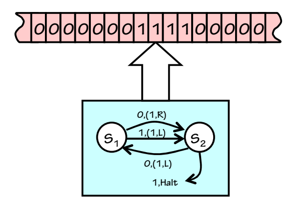
Big Idea #2:

Turing Machines

Alan Turing was one of a group of researchers studying alternative models of computation.

He proposed a conceptual model consisting of an FSM combined with an infinite digital tape that could be read and written at each step.

Turing's model (like others of the time) solves the "FINITE" problem of FSMs.



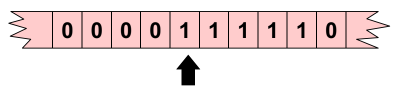
A Turing machine Example

Turing Machine Specification

- Doubly-infinite tape
- Discrete symbol positions
- Finite alphabet – say {0, 1}
- Control FSM
  - INPUTS:
    - Current symbol
  - OUTPUTS:
    - write 0/1
    - move Left/Right
- Initial Starting State {S0}
- Halt State {Halt}

A Turing machine, like an FSM, can be specified with a truth table. The following Turing Machine implements a unary (base 1) incrementer.

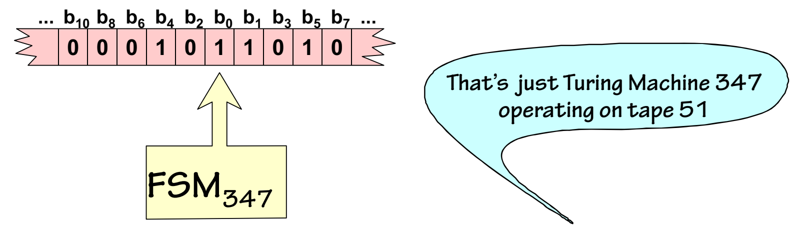
Current State	Input	Next State	Write Tape	Move Tape
S0	1	S0	1	R
S0	0	S1	1	L
S1	1	S1	1	L
S1	0	HALT	0	R



OK, but how about *real* computations... like fact(n)?

Turing Machine Tapes as Integers

Canonical names for bounded tape configurations:



Encoding: starting at current position, build a binary integer taking successively higher-order bits from right and left sides. If nonzero region is bounded, eventually all 1's will be incorporated into the resulting integer representation.

TMs as Integer Functions

Turing Machine T_i operating on Tape x, where $x = \ldots b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

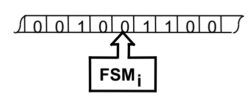
$y = T_i[x]$

x: input tape configuration
y: output tape configuration

Meanwhile, Turing's buddies Were busy too...

Alternative models of computation

Turing Machines [Turing]



Recursive Functions [Kleene]

$F(0,x) \equiv x$

$F(1+y,x) \equiv 1+F(x,y)$

```
(define (fact n)
  (... (fact (- n 1)) ...)
```

Kleene

Turing

Lambda calculus [Church, Curry, Rosser...]

$\lambda x.\lambda y.xxy$

```
(lambda(x)(lambda(y)(x (x y))))
```

Church

Production Systems [Post, Markov]

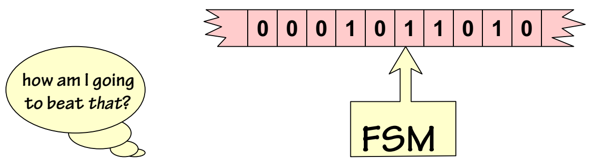
$\alpha \rightarrow \beta$

```
IF pulse=0 THEN
   patient=dead
```

Post

The 1st Computer Industry Shakeout

Here's a TM that computes SQUARE ROOT!



And the battles raged

Here's a Lambda Expression that does the same thing...

```
(λ(x)  .....)
```

... and here's one that computes the n^{th} root for ANY n!

```
(λ(x n)  .....)
```

maybe if I gave away a microwave oven with every Turing Machine...

CONTEST: Which model computes more functions?

RESULT: an N-way TIE!

Big Idea #3:

Computability

FACT: Each model studied is capable of computing <u>exactly</u> the same set of integer functions!

Proof Technique:
Constructions that translate between models

BIG IDEA:
Computability, independent of computation scheme chosen

unproved, but universally accepted...

<u>Church's Thesis:</u>

Every discrete function computable by ANY realizable machine is computable by some Turing machine.

Computable Functions

f(x) *computable* <=> for some k, all x:
$f(x) = T_K[x] \equiv f_K(x)$

Equivalently: f(x) computable on Cray, Pentium, in C, Scheme, Java, ...

Representation Tricks:

- Multi-argument functions? to compute $f_k(x,y)$, use
  <x,y> = integer whose *even* bits come from x, and whose *odd* bits come from y;
  whence
  $f_K(x, y) = T_K[<x, y>]$
- Data types: Can encode characters, strings, floats, ... as integers.
- Alphabet size: use groups of N bits for 2^N symbols

Enumeration of Computable functions

Conceptual table of ALL Turing Machine behaviors…

VERTICAL AXIS: Enumeration of TM's (computable functions)

HORIZONTAL AXIS: Enumeration of input tapes.

ENTRY AT (n, m): Result of applying m^{th} TM to argument n

- INTEGER k: TM halts, leaving k on tape.
- ★ : TM never halts.

f_i	$f_i(0)$	$f_i(1)$	$f_i(2)$	…	$f_i(n)$	…
f_0	37	23	★	…	33	…
f_1	42	★	111	…	12	…
f_2	★	★	★	…	★	…
…	…	…	…	…	…	…
f_m	0	★	831	…	$f_m(n)$	…
…	…	…	…	…	…	…

aren't *all* well-defined integer functions computable?

NO!

there are simply too many integer functions to fit in our enumeration!

Uncomputable Functions

Unfortunately, not every well-defined integer function is computable. The most famous such function is the so-called Halting function, $f_H(k, j)$, defined by:

$f_H(k, j)$ = 1 if $T_k[j]$ halts;
0 otherwise.

$f_H(k, j)$ determines whether the k^{th} TM halts when given a tape containing j.

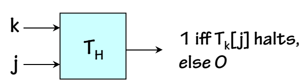
THEOREM: f_H is different from every function in our enumeration of computable functions; hence it cannot be computed by any Turing Machine.

PROOF TECHNIQUE: "Diagonalization" (after Cantor, Gödel)

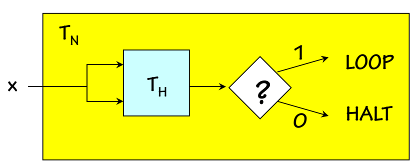
- If f_H is computable, it is equivalent to some TM (say, T_H).
- Using T_H as a component, we can construct another TM whose behavior differs from every entry in our enumeration and hence must not be computable.
- Hence f_H cannot be computable.

Why f_H is uncomputable

If f_H is computable, it is equivalent to some TM (say, T_H):



Then T_N (N for "Nasty"), which must be computable if T_H is:



$T_N[x]$: LOOPS if $T_x[x]$ halts;
HALTS if $T_x[x]$ loops

Finally, consider giving N as an argument to T_N:

$T_N[N]$: LOOPS if $T_N[N]$ halts;
HALTS if $T_N[N]$ loops

Contradiction!

T_N can't be computable, hence T_H can't either!

Footnote: Diagonalization

(clever proof technique used by Cantor, Gödel, Turing)

If T_H exists, we can use it to construct T_N. Hence T_N is computable if T_H is.
(informally we argue by Church's Thesis; but we can show the actual T_N construction, if pressed)

Why T_N can't be computable:

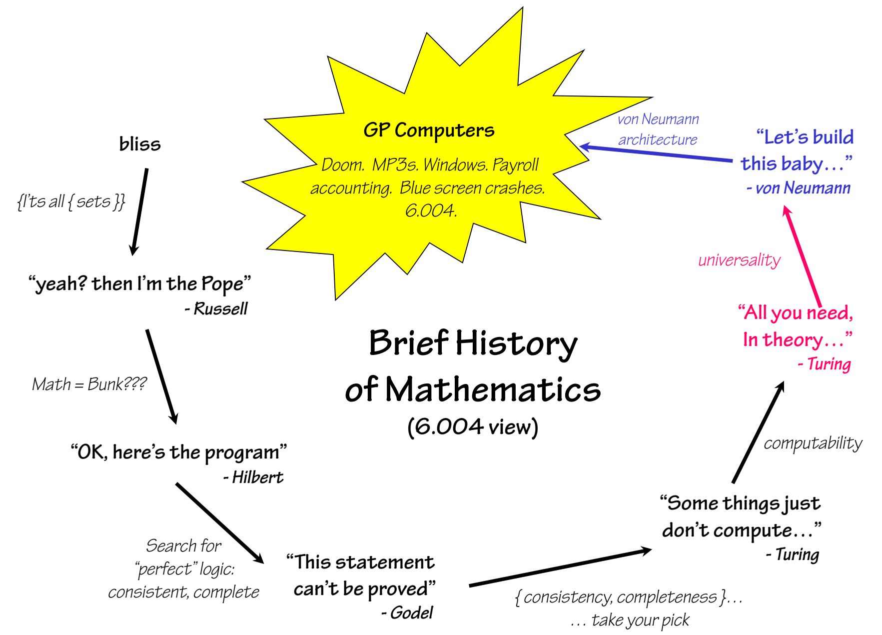
f_i	$f_i(0)$	$f_i(1)$	$f_i(2)$	…	$f_i(n)$	…
f_0	★ 37	23	★	…	33	…
f_1	42	★ 0	111	…	12	…
f_2	★	★	★ 0	…	★	…
…	…	…	…	★ 0	…	…
f_m	0	★	831	…	$f_m(n)$	…
…	…	…	…	…	…	…

T_N differs from every computable function for at least one argument – along the diagonal of our table. Hence T_N can't be among the entries in our table!

Hence no such T_H can be constructed, even in theory.

Computable Functions: A TINY SUBSET of all Integer functions!

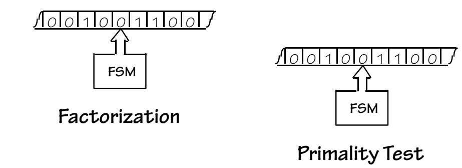
Brief History of Mathematics
(6.004 view)

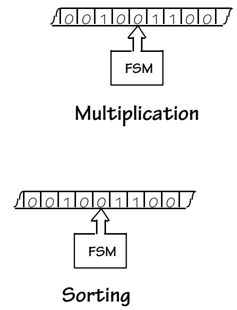


meanwhile…

Turing machines Galore!

"special-purpose" Turing Machines….





Is there an alternative to Infinitely many, ad-hoc Turing Machines?

The Universal Function

OK, so there are uncomputable functions – infinitely many of them, in fact.

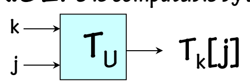
Here's an interesting candidate to explore: the Universal function, U, defined by

$$U(k, j) = T_k[j]$$

Could this be computable???

it sure would be neat to have a single, general-purpose machine…

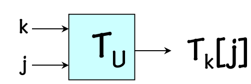
SURPRISE! U is computable by a Turing Machine:



In fact, there are infinitely many such machines. Each is capable of performing *any* computation that can be performed by *any* TM!

Big Idea #4:

Universality



What's going on here?

k encodes a "program" – a description of some arbitrary machine.

j encodes the input data to be used.

T_U *interprets* the program, emulating its processing of the data!

KEY IDEA: Interpretation.
Manipulate *coded representations* of computing machines, rather than the machines themselves.

Turing Universality: The *Universal Turing Machine* is the paradigm for modern general-purpose computers! (cf: earlier *special-purpose* computers)

- Basic threshold test: Is your machine *Turing Universal*? If so, it can emulate every other Turing machine!
- Remarkably low threshold: UTMs with handfuls of states exist.
- Every modern computer is a UTM (given enough memory)
- To show your machine is Universal: demonstrate that it can emulate some known UTM.

Coded Algorithms: Key to CS

data vs hardware

Algorithms as data: enables

COMPILERS: analyze, optimize, transform behavior

$T_{COMPILER\text{-}X\text{-}to\text{-}Y}[P_X] = P_Y$, such that
$T_X[P_X, z] = T_Y[P_Y, z]$

SOFTWARE ENGINEERING:
Composition, iteration, abstraction of coded behavior
$F(x) = g(h(x), p((q(x)))$

LANGUAGE DESIGN: Separate specification from implementation

- C, Java, JSIM, Linux, ... all run on X86, PPC, Sun, ...
- Parallel development paths:
  - Language/Software design
  - Interpreter/Hardware design

Summary

Formal models (computability, Turing Machines, Universality) provide the basis for modern computer science:

- Fundamental limits (what can't be done, even given plenty of memory and time)
- Fundamental equivalence of computation models
- Representation of algorithms as data, rather than machinery
- Programs, Software, Interpreters, Compilers, ...

They leave many practical dimensions to deal with:

- Costs: Memory size, Time Performance
- Programmability

Next step: Design of a practical interpreter!