DURING 1920s & 1930s, much of the "science" part of computer science was being developed (long before actual electronic computers existed). Many different "Models of Computation" were proposed, and the classes of "functions" that each could compute were analyzed.

One of these models was the "TURING MACHINE", named after Alan Turing (1912-1954).

Alan Turing

A Turing Machine is just an FSM which receives its inputs and writes outputs onto an "infinite tape". This simple addition overcomes the FSM's limitation that it can only keep track of a "bounded number of events".
**Turing Machine Tapes as Integers**

Canonical names for bounded tape configurations:

\[
\begin{array}{cccccccc}
  b_8 & b_6 & b_4 & b_2 & b_0 & b_1 & b_3 & b_5 & b_7 \\
  0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
\end{array}
\]

**Look, it's just FSM \( i \) operating on tape \( j \)**

**Note:** The FSM part of a Turing Machine is just one of the FSMs in our enumeration. The tape can also be represented as an integer, but this is trickier. It is natural to represent it as a binary fraction, with a binary point just to the left of the starting position. If the binary number is rational, we can alternate bits from each side of the binary point until all that is left is zeros, then we have an integer.
**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 when TM *halts*

I wonder if a TM can compute EVERY integer function...
**Alternative Models of Computation**

- **Turing Machines** [Turing]
- **Hardware**
- **Recursive Functions** [Kleene]
  
  \[
  F(0,x) = x \\
  F(y,0) = y \\
  F(y,x) = x + y + F(y-1,x-1)
  \]

- **Lambda calculus** [Church, Curry, Rosser...]
  
  \[
  \lambda x. \lambda y. xxy
  \]

- **Math**
  
  \[
  (\text{lambda}(x)(\text{lambda}(y)(x \ (x \ y))))
  \]

- **Production Systems** [Post, Markov]
  
  \[
  s_0 \rightarrow [] \\
  s_i \rightarrow []s_j \\
  s_i \rightarrow s_is_j
  \]

**Turing (1903–1995)**

**Church (1903–1995)**

**Post (1897–1954)**

**Kleene (1909–1994)**
The 1st Computer Industry Shakeout

Here’s a TM that computes SQUARE ROOT!

---

FSM

| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |

---
**AND THE BATTLES RAGED**

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

\[
(\lambda(x) \ldots)
\]

... and here’s one that computes the \(n^{th}\) root for ANY \(n\)!

\[
(\lambda(x \ n) \ldots)
\]
A FUNDAMENTAL RESULT

Turing’s amazing proof: Each model is capable of computing exactly the same set of integer functions! None is more powerful than the others.

**Proof Technique:** Constructions that translate between models

**BIG IDEA:** Computability, independent of computation scheme chosen

Church's Thesis:

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

This means that we know of no mechanisms (including computers) that are more "powerful" than a Turing Machine, in terms of the functions they can compute.
Computable Functions

The “input” to our computable function will be given on the initial tape, and the “output” will be the contents of the tape when the TM halts.

Representation tricks: to compute $f_k(x,y)$ (2 inputs)

$x,y \equiv$ integer whose even bits come from $x$, and whose odd bits come from $y$; whence

$$f_k(x, y) \equiv T_k[<x, y>]$$

$f_{12345}(x,y) = x \times y$

$f_{23456}(x) = 1$ iff $x$ is prime, else 0
TM5, like programs, can misbehave

It is possible that a given Turing Machine may not produce a result for a given input tape. And it may do so by entering an infinite loop!

Consider the given TM.

It scans a tape looking for the first non-zero cell to the right.

What does it do when given a tape that has no 1's to its left?

We say this TM does not halt for that input!

<table>
<thead>
<tr>
<th>Current State</th>
<th>Tape Input</th>
<th>Write Tape</th>
<th>Move</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>1</td>
<td>1</td>
<td>L</td>
<td>Halt S0</td>
</tr>
<tr>
<td>S0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td></td>
</tr>
</tbody>
</table>

$\text{tape}_{256} = \ldots 0|0|0|0|0|0|0|1|0|0 \ldots$

$\text{tape}_{8} = \ldots 0|1|0|0|0|0|0|0|0|0 \ldots$
**Enumeration of Computable Functions**

Conceptual table of TM behaviors...

**VERTICAL AXIS:** Enumeration of TMs.

**HORIZONTAL AXIS:** Enumeration of input tapes.

(j,k) entry = result of TM\(_k[j]\) -- integer, or \(*\) if it never halts.

<table>
<thead>
<tr>
<th>Turing Machine FSMs</th>
<th>Turing Machine Tapes</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(_0)</td>
<td>X1       X1       X0       ...     f(_i(j))  ...</td>
</tr>
<tr>
<td>f(_1)</td>
<td>X1       X0       B00       ...     ...</td>
</tr>
<tr>
<td>...</td>
<td>...     ...      ...       ...   ...</td>
</tr>
<tr>
<td>f(_k)</td>
<td>...       ...       ...       ...     f(_k(j))</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

The Halting Problem: Given j, k: Does TM\(_k \) Halt with input j?

Every computable function is in this table, since everything that we know how to compute can be computed by a TM.

Do there exist well-specified integer functions that a TM can't compute?
The Halting Problem

The Halting Function: $T_H[k, j] = 1$ iff $TM_k[j]$ halts, else 0

Can a Turing machine compute this function?

Suppose, for a moment, $T_H$ exists:

1 iff $T_k[j]$ HALTS
0 otherwise

Then we can build a $T_{Nasty}$:

$T_{Nasty}[k]$ LOOP if $T_k[k] = 1$ (halts)
HALT if $T_k[k] = 0$ (loops)

If $T_H$ is computable then so is $T_{Nasty}$

We only run $T_H$ on a subset of inputs, those on the diagonal of the table given on the previous slide.
What does $T_{\text{Nasty}}[\text{Nasty}]$ do?

Answer:

$T_{\text{Nasty}}[\text{Nasty}]$ loops if $T_{\text{Nasty}}[\text{Nasty}]$ halts

$T_{\text{Nasty}}[\text{Nasty}]$ halts if $T_{\text{Nasty}}[\text{Nasty}]$ loops

That's a contradiction.

Thus, $T_{H}$ is not computable by a Turing Machine!

**Net Result:** There are some integer functions that Turing Machines simply cannot answer. Since, we know of no better model of computation than a Turing machine, this implies that there are some well-specified problems that defy computation.
Limits of Turing Machines

A Turing machine is formal abstraction that addresses

- Fundamental Limits of Computability -
  What is means to compute.
  The existence of uncomputable functions.
- We know of no machine more powerful than a Turing machine in terms of the functions that it can compute.

But they ignore

- Practical coding of programs
- Performance
- Implementability
- Programmability

... these latter issues are the primary focus of contemporary computer science (Remainder of Comp 411)
Recall Church's thesis:

"Any discrete function computable by ANY realizable machine is computable by some Turing Machine"

We've defined what it means to COMPUTE (whatever a TM can compute), but, a Turing machine is nothing more that an FSM that receives inputs from, and outputs onto, an infinite tape.

So far, we've been designing a new FSM for each new Turing machine that we encounter.

Wouldn't it be nice if we could design a more general-purpose Turing machine?
Programs as Data

What if we encoded the description of the FSM on our tape, and then wrote a general purpose FSM to read the tape and EMULATE the behavior of the encoded machine? We could just store the state-transition table for our TM on the tape and then design a new TM that makes reference to it as often as it likes. It seems possible that such a machine could be built.

"It is possible to invent a single machine which can be used to compute any computable sequence. If this machine $U$ is supplied with a tape on the beginning of which is written the S.D ["standard description" of an action table] of some computing machine $M$, then $U$ will compute the same sequence as $M."$

**Fundamental Result: Universality**

Define “Universal Function”: \( U(x,y) = T_x(y) \) for every \( x, y \) ...

Surprise! \( U(x,y) \) is computable, hence \( U(x,y) = T_u(<x,y>) \) for some \( u \).

**Universal Turing Machine (UTM):**

\[
T_U[<y, z>] = T_y[z]
\]

PARADIGM for general-purpose computer!

Any one of them can evaluate any computable function by simulating/emulating/interpreting the actions of Turing machine given to it as an input.

**Universality:**
Basic requirement for a general purpose computer
**DEMONSTRATING UNIVERSALITY**

Suppose you’ve designed Turing Machine $T_K$ and want to show that it’s universal.

**APPROACH:**

1. Find some known universal machine, say $T_U$.
2. Devise a program, $P$, to simulate $T_U$ on $T_K$:
   
   $$T_K[<P,x>] = T_U[x] \text{ for all } x$$

3. Since $T_U[<y,z>] = T_y[z]$, it follows that, for all $y$ and $z$,

   $$T_K[<P,<y,z>>] = T_U[<y,z>] = T_y[z]$$

**CONCLUSION:** Armed with program $P$, machine $T_K$ can mimic the behavior of an arbitrary machine $T_y$ operating on an arbitrary input tape $z$.

**HENCE** $T_K$ can compute any function that can be computed by any Turing Machine.
Next Time

Enough theory already, let’s build something!
I wonder where this goes?
**Another Functional Units**

We'll need functional units fast memories. We begin by building "wide" registers. First, we'll add a control that "enables" the loading of a register.
A Register File

We can also construct an addressable array of registers.
A Multi-Ported Register File

We can add multiple read ports by simply adding more output MUXs.
This is it!

This is where our story actually begins. We are now ready to build a computer.

The ingredients are all in place. It is time to build a legitimate computer. One that executes instructions, much the way any desktop, tablet, smartphone, or other computer does.
**The ARM7 ISA**

<table>
<thead>
<tr>
<th>R type:</th>
<th>Cond</th>
<th>000</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Shift</th>
<th>0</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>I type:</td>
<td>Cond</td>
<td>001</td>
<td>Opcode</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>Shift</td>
<td>Imm</td>
<td></td>
</tr>
<tr>
<td>D type:</td>
<td>Cond</td>
<td>010</td>
<td>AddrMode</td>
<td>Rn</td>
<td>Rd</td>
<td>Imm12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X type:</td>
<td>Cond</td>
<td>011</td>
<td>AddrMode</td>
<td>Rn</td>
<td>Rd</td>
<td>Shift</td>
<td>0</td>
<td>Rm</td>
<td></td>
</tr>
<tr>
<td>B type:</td>
<td>Cond</td>
<td>101</td>
<td>L</td>
<td>Imm24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Five key instruction formats:**  
0) ALU with two register operands  
1) ALU with a register and an immediate operand  
2) Load/Store with an immediate offset  
3) Load/Store with a register offset  
5) Branch
Design Approach

Incremental Featurism:

Each instruction class can be implemented using our component repertoire. We’ll try implementing data paths for each class individually, and merge them as we go (using MUXes, etc).

Steps:
1. 3-Operand ALU instructions
2. ALU w/immediate instructions
3. Load & Store Instructions
4. Branch instructions
5. Leftovers
6. Reset & Exceptions

Our bag of parts:

- Registers
- Muxes
- ALU & adders
- Memories

Diagram of components:

- RA1, RA2
- WA, WE, WD
- Register File (3-port)
- Instruction Memory
- Data Memory
- ALU
- muxes
- data paths
**Instruction Fetch/Decode**

- Fetch an instruction, and decode it

- Use PC as memory address
- Add 4 to current PC, and update PC on the next rising clock
- Fetch instruction from memory
  - We'll use some instruction fields directly (register numbers, constants)
  - Use format, opcode bits, and a few assorted bits to generate controls
R-TYPE DATA PROCESSING

ALU instructions with register operands

Rd - register file write address
Rn, Rm - register source operands
Shift or Rs - Optional shift of Rm
LA - direction and type of shift
S-bit - controls update of PSR
Func decoding from ALU lecture
Register write back controlled by WERF logic
Next Time

More instructions...

1. [Illustration of a bird and a horse]
2. [Illustration of a person feeding a horse]
3. [Illustration of a person cooking a meatball]
4. [Illustration of a person shaping the meatball]
5. [Illustration of a chef serving the meatball]
6. [Illustration of a person eating the meatball]
WERF Logic

Not every instruction updates a destination register.

**CMP, CMN, TST, TEQ** don’t update any register.

Conditional execution is controlled by the WERF logic. WE is set only if the condition is met. Otherwise it is effectively annulled.

<table>
<thead>
<tr>
<th>$I_{31}$</th>
<th>$I_{30}$</th>
<th>$I_{29}$</th>
<th>$I_{28}$</th>
<th>$I_{24}$</th>
<th>$I_{23}$</th>
<th>WE</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>cmp, cmn, tst, teq</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>Cond = AL</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>Cond = AL</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>Z</td>
<td>Cond = EQ</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>Z</td>
<td>Cond = EQ</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Z</td>
<td>Cond = NE</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>!Z</td>
<td>Cond = NE</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>!(Z</td>
<td>(N^V))</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>!(Z</td>
<td>(N^V))</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Z</td>
<td>Cond = LE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>Z</td>
<td>Cond = LE</td>
</tr>
</tbody>
</table>