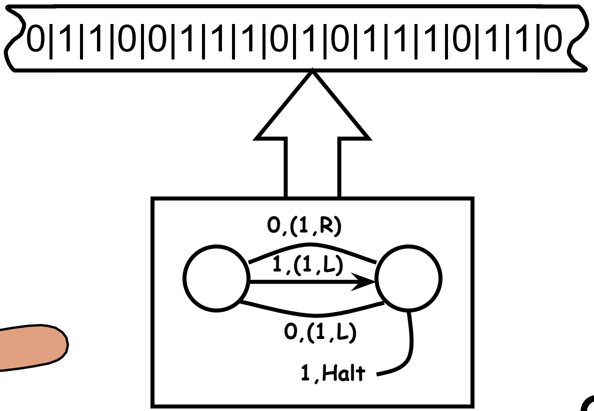
UNBOUNDED-SPACE COMPUTATION

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).

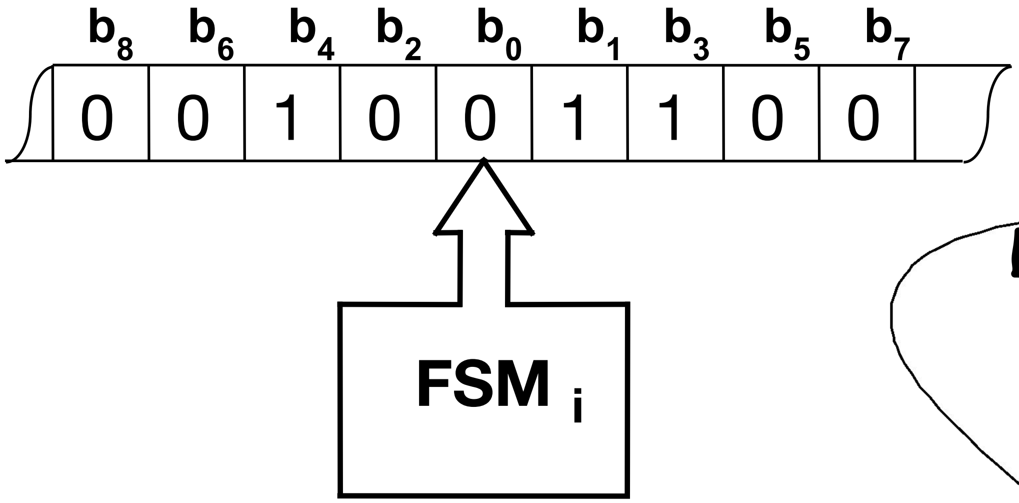


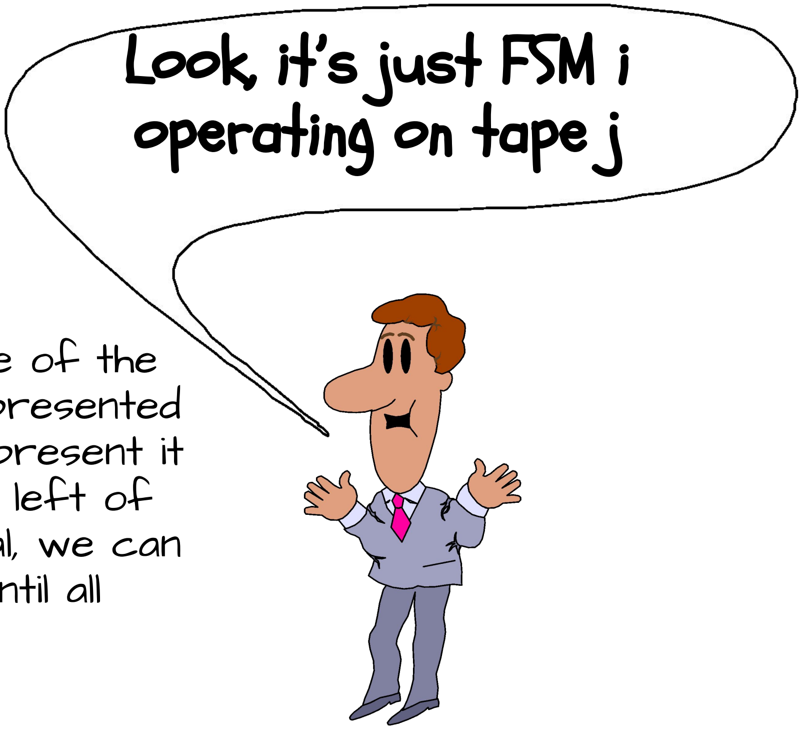
Alan Turing

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:





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

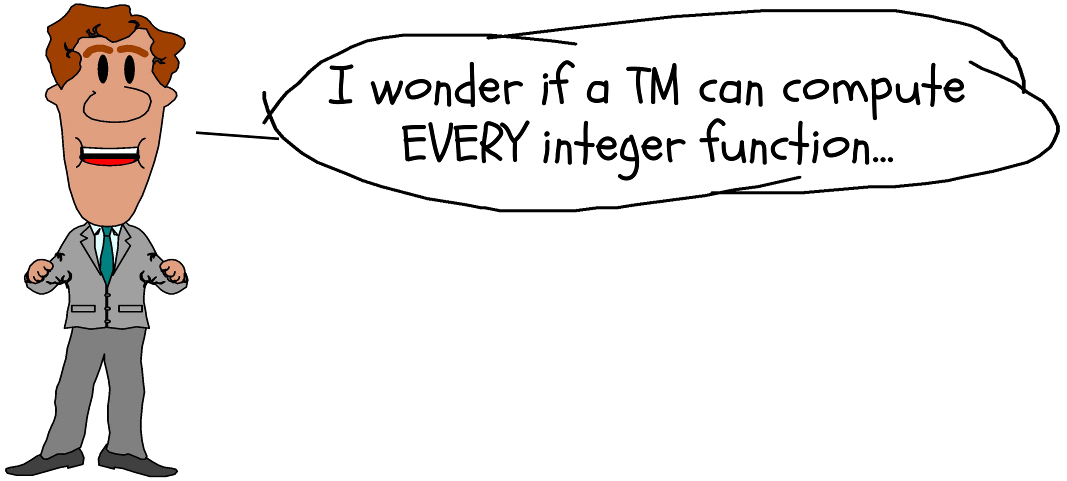
Turing Machine T_i operating on Tape x,

where $x = \ldots b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

$$\boxed{y = T_i[x]}$$

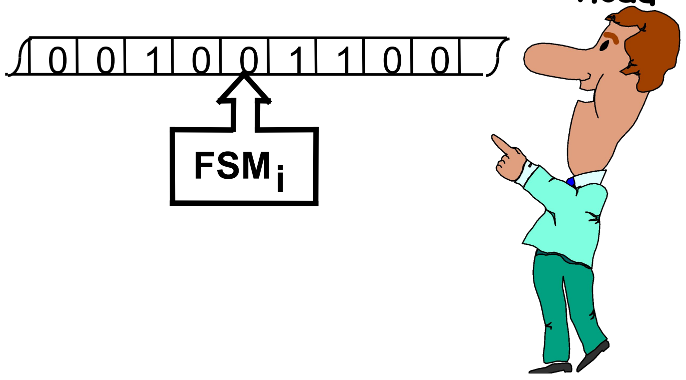
x: input tape configuration

y: output tape when TM *halts*



Alternative Models of Computation

Turing Machines [Turing]



Turing

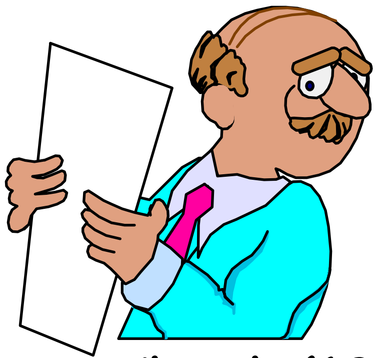
Recursive Functions [Kleene]

$$F(0,x) = x$$
$$F(y,0) = y$$
$$F(y,x) = x + y + F(y-1,x-1)$$

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

Kleene (1909-1994)

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

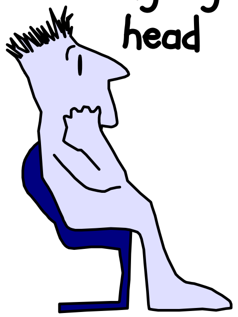


$\lambda x.\lambda y.xxy$

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

Church (1903-1995)
Turing's PhD Advisor

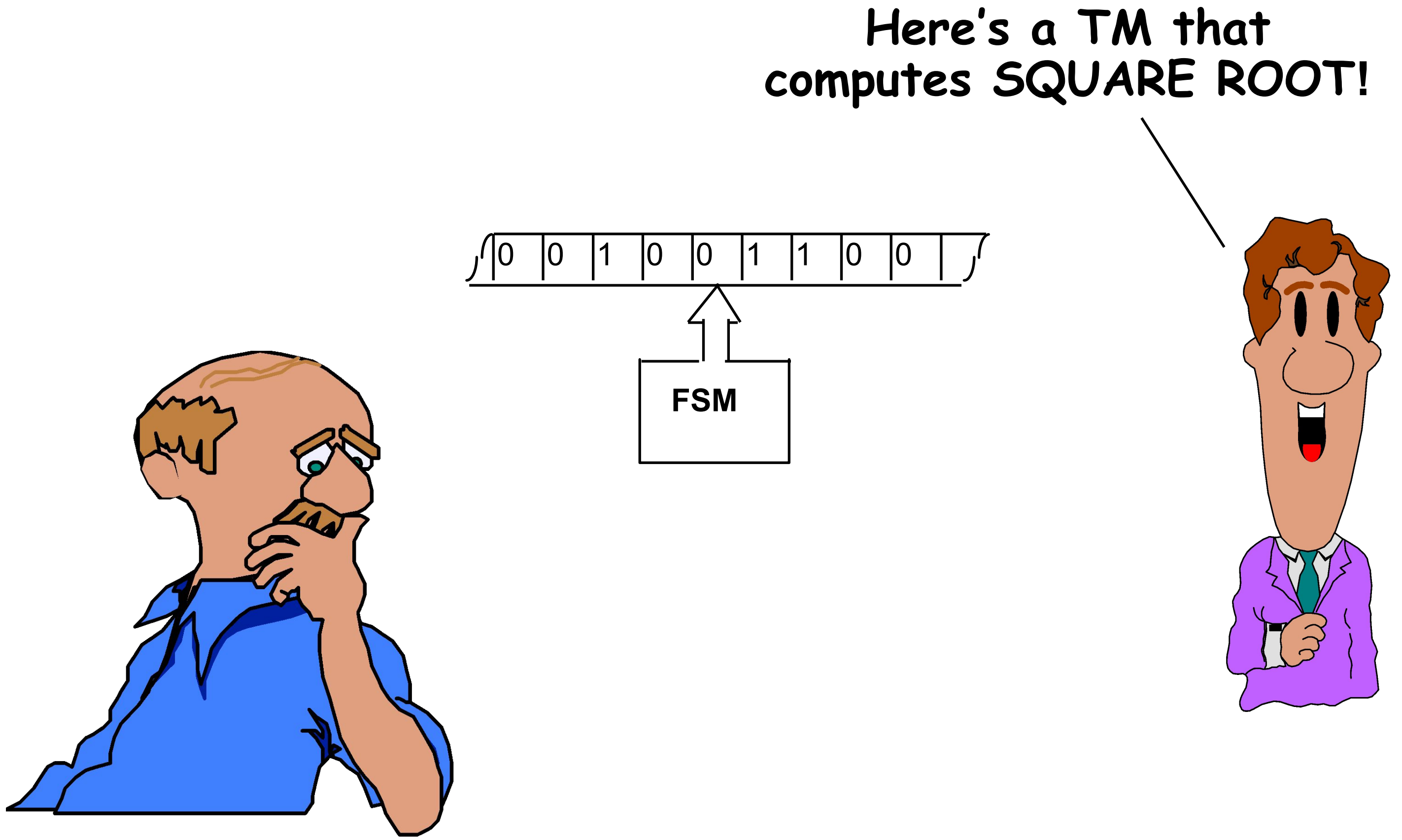
Production Systems [Post, Markov]



$\$_0 \rightarrow []$
$\$ \rightarrow [\$]$
$\$ \rightarrow \$\$$
$\$_i[]\$_j \rightarrow \$_i\$_j$

Post
(1897-1954)

The 1st Computer Industry Shakeout



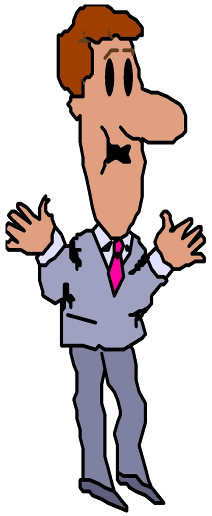
And the Battles Raged

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

`(λ(x) .....)`

... and here's one that computes
the nth root for ANY n!

`(λ(x n) .....)`



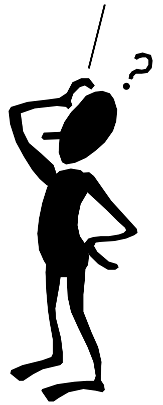
A Fundamental Result

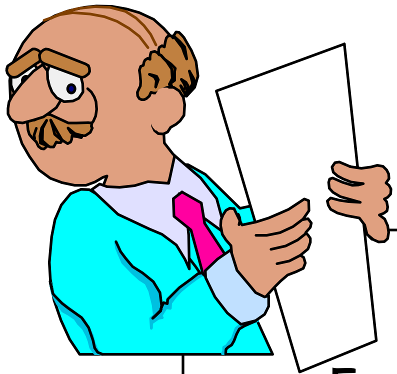
Turing's amazing proof: Each model is capable of computing exactly the same set of integer functions! None is more powerful than the others.

Proof Technique: Constructions that translate between models

BIG IDEA: Computability, independent of computation scheme chosen

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.



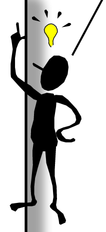


Church's Thesis:

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

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.



f(x) computable <=> for some k, all x:

$$f(x) = T_k[x] \equiv f_k(x)$$

Representation tricks: to compute $f_k(x,y)$ (2 inputs)
<x,y> ≡ integer whose *even* bits come from x,
and whose *odd* bits come from y; whence

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

$f_{12345}(x,y) = x * y$

$f_{23456}(x)$ = 1 iff x is prime, else 0

TMs, 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!

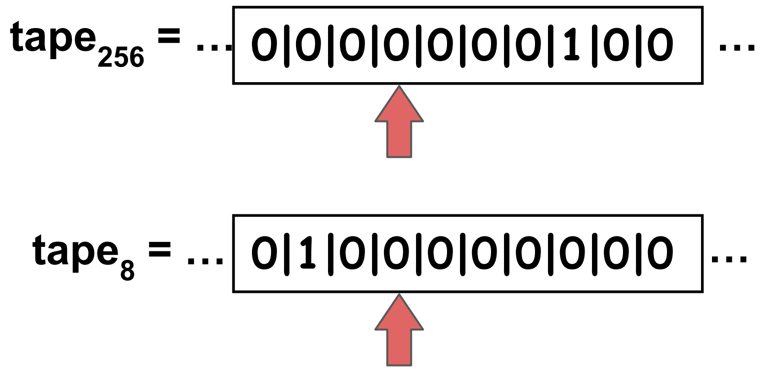
Consider the given TM.

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!

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



$tape_{256}$ = … 0|0|0|0|0|0|0|1|0|0 …

$tape_{8}$ = … 0|1|0|0|0|0|0|0|0|0 …

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.

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?

Turing Machine Tapes ⟶

Turing Machine FSMs ↓

	$f_i(0)$	$f_i(1)$	$f_i(2)$	…	$f_i(j)$	…
f_0	~~37~~ X1	~~23~~ X1	~~*~~ X0	…	…	
f_1	~~42~~ X1	~~*~~ X0	~~666~~ X1	…	…	
…	…	…	…	…	…	
f_k	…	…	…	…	$f_k(j)$	
…						

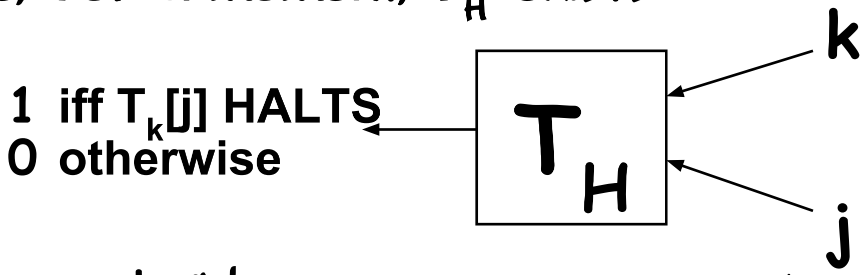
The Halting Problem: Given j, k: Does TM_k Halt with input j?

The Halting Problem

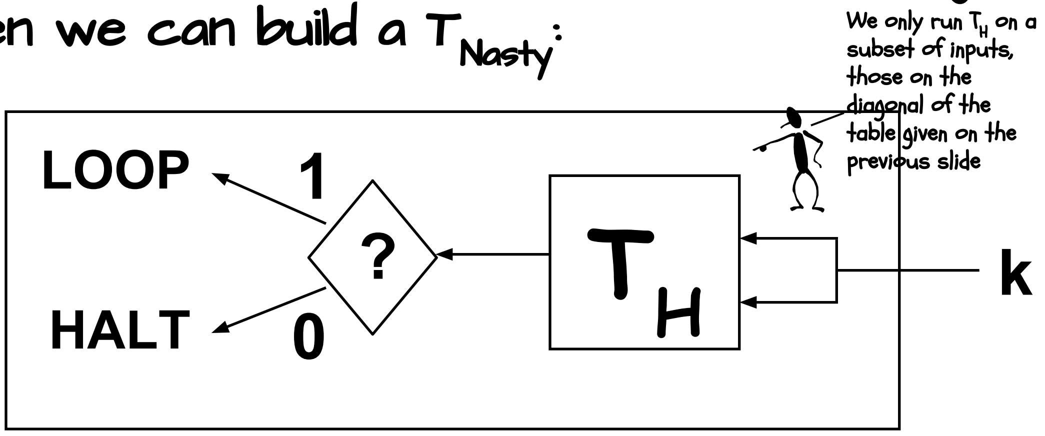
The Halting Function: $T_H[k, j]$ = 1 iff $TM_k[j]$ halts, else 0

Can a Turing machine compute this function?

Suppose, for a moment, T_H exists:



Then we can build a T_{Nasty}:

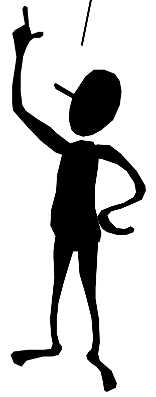


We only run T_H on a subset of inputs, those on the diagonal of the table given on the previous slide
LOOP
HALT
1
0
?
T_H
k

$T_{Nasty}[k]$

LOOP if $T_k[k]$ = 1 (halts)
HALT if $T_k[k]$ = 0 (loops)

If T_H is computable then so is T_{Nasty}



What does T_{Nasty}[Nasty] do?

Answer:

T_{Nasty}[Nasty] loops if T_{Nasty}[Nasty] halts
T_{Nasty}[Nasty] halts if T_{Nasty}[Nasty] loops

That's a contradiction.

Thus, T_H is not computable by a Turing Machine!

<u>**Net Result**</u>: 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

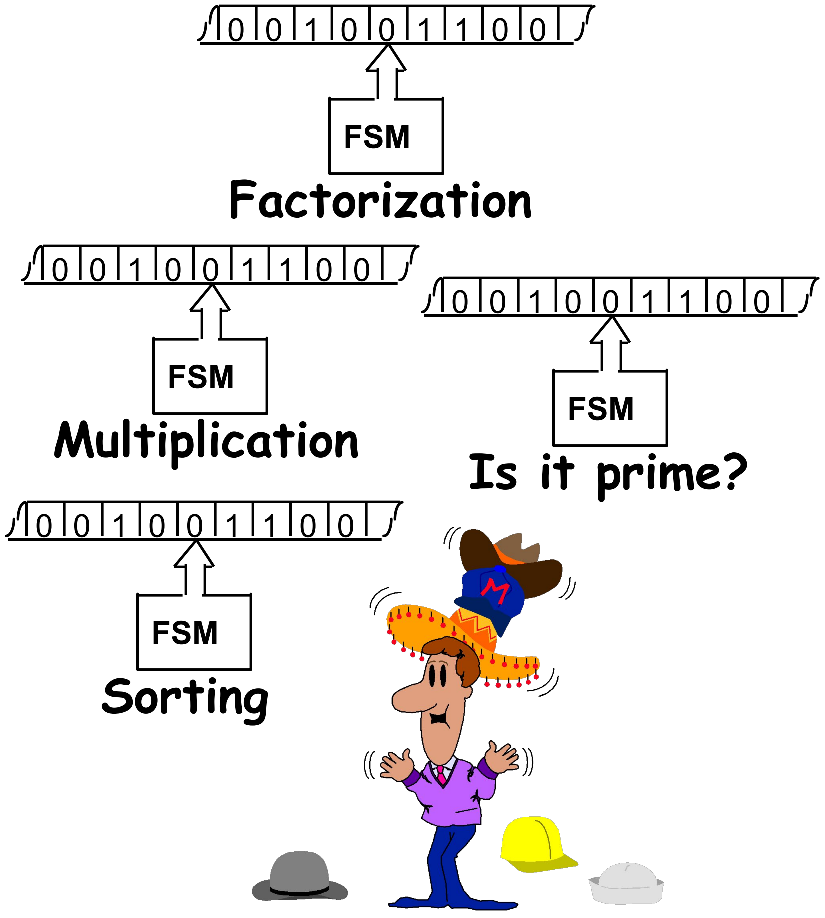
- Fundamental Limits of Computability -
  What is means to compute.
  The existence of uncomputable functions.
- We know of no machine more powerful than a Turing machine in terms of the functions that it can compute.

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)

Computability vs. Programmability



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.

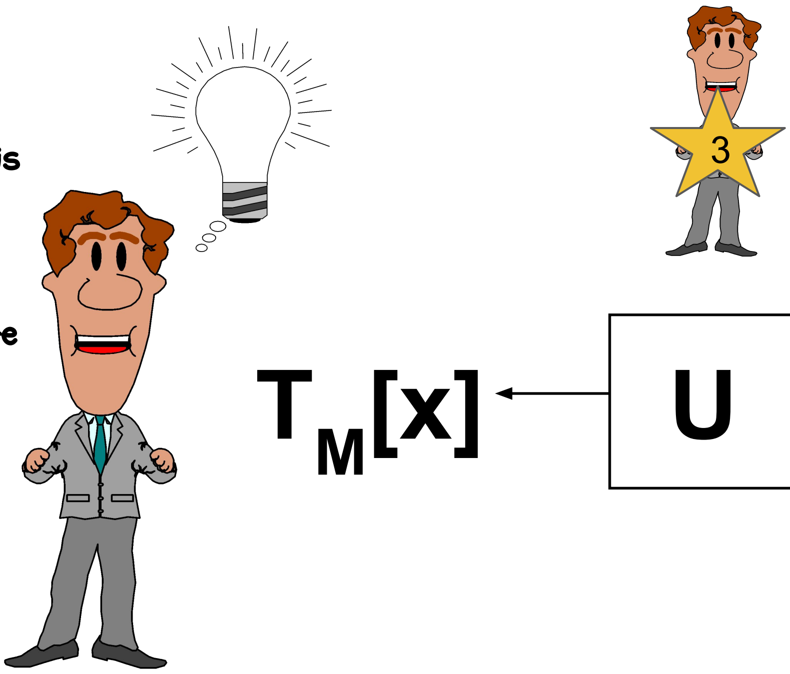
So far, we've been designing **a new FSM for each new Turing machine** that we encounter.

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."
– Turing 1936 (Proc of the London Mathematical Society, Ser. 2, Vol. 42)



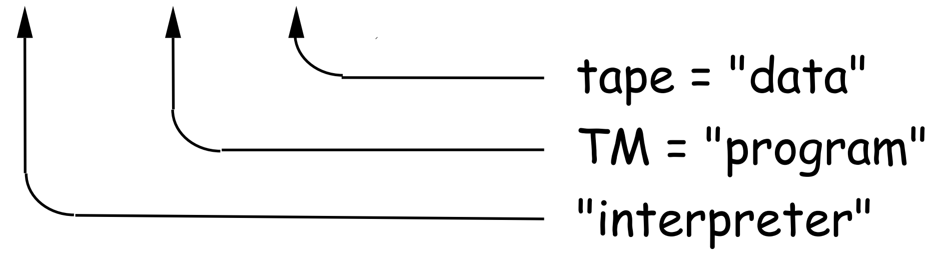
$$\begin{tikzcd} T_M[x] & \boxed{U} \arrow[l] & M \arrow[l] \\ & & x \arrow[ul] \end{tikzcd}$$

Fundamental Result: Universality

Define "Universal Function": $U(x,y) = T_x(y)$ for every x, y …
Surprise! U(x,y) IS COMPUTABLE,
hence $U(x,y) = T_U(\langle x,y\rangle)$ for some U.

Universal Turing Machine (UTM):

$$T_U[\langle y,\ z\rangle] = T_y[z]$$



PARADIGM for General-Purpose Computer!

INFINITELY many UTMs …
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 its universal.

Turing Complete

APPROACH:

1. Find some known universal machine, say T_u.
2. Devise a program, P, to simulate T_u on T_K:
   $T_K[<P,x>] = T_u[x]$ for all x.
3. Since $T_u[<y,z>] = T_y[z]$, it follows that, for all y and z.

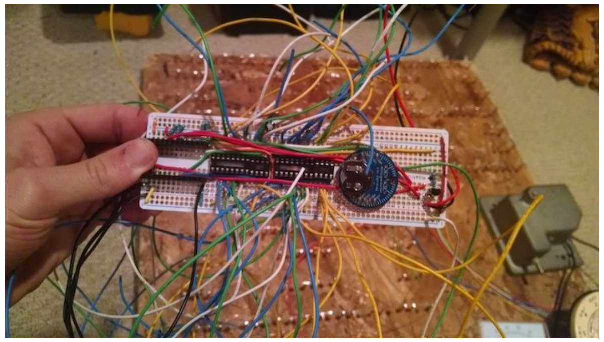
$$\boxed{T_K\,[<P,<y,z>>] \;=\; T_U[<y,z>] \;=\; T_y[z]}$$

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.

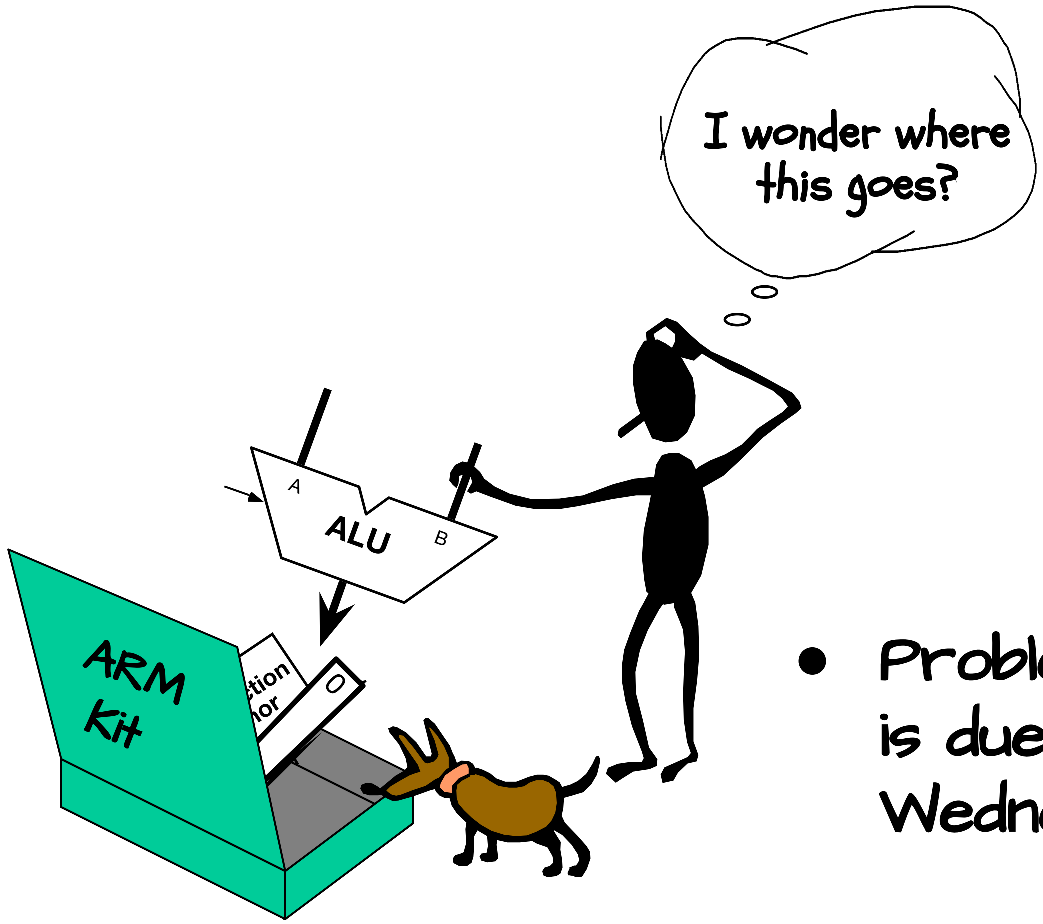
HENCE T_K can compute any function that can be computed by any Turing Machine.

Next Time

Enough theory already, let's build something!



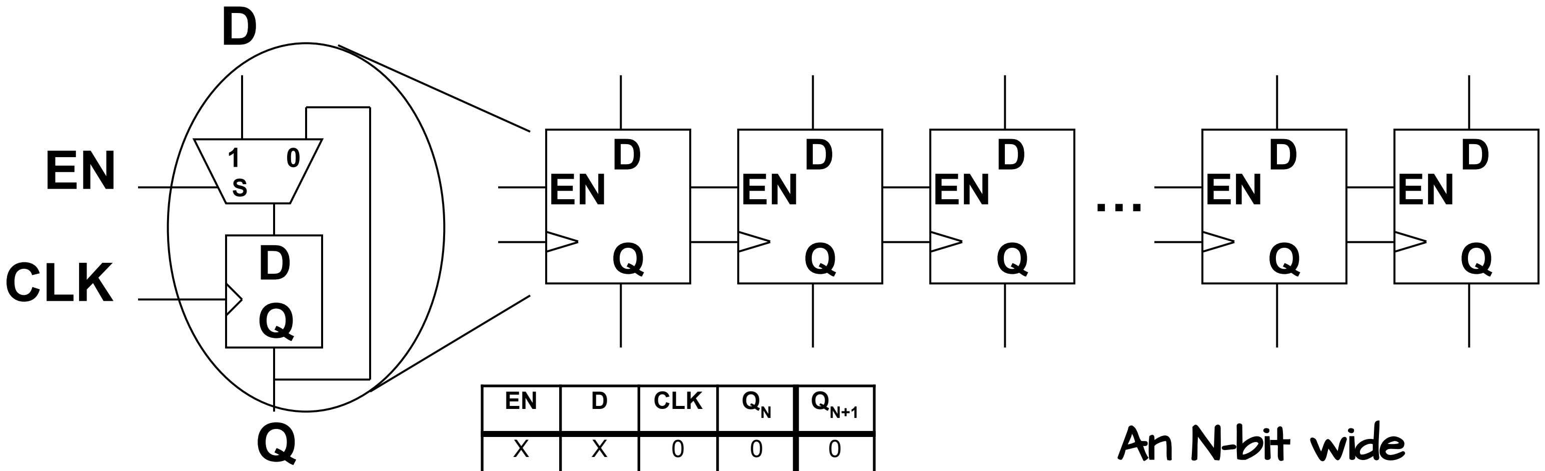
Building a Computer



- Problem Set #3 is due on Wednesday.

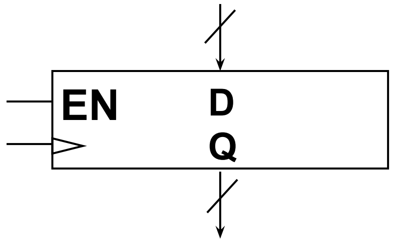
Other Functional Units

We'll need a few more functional units. We begin by adding an "enable" input to a standard flip-flop. With those we will build "wide" registers (i.e. registers with shared clocks and enables).



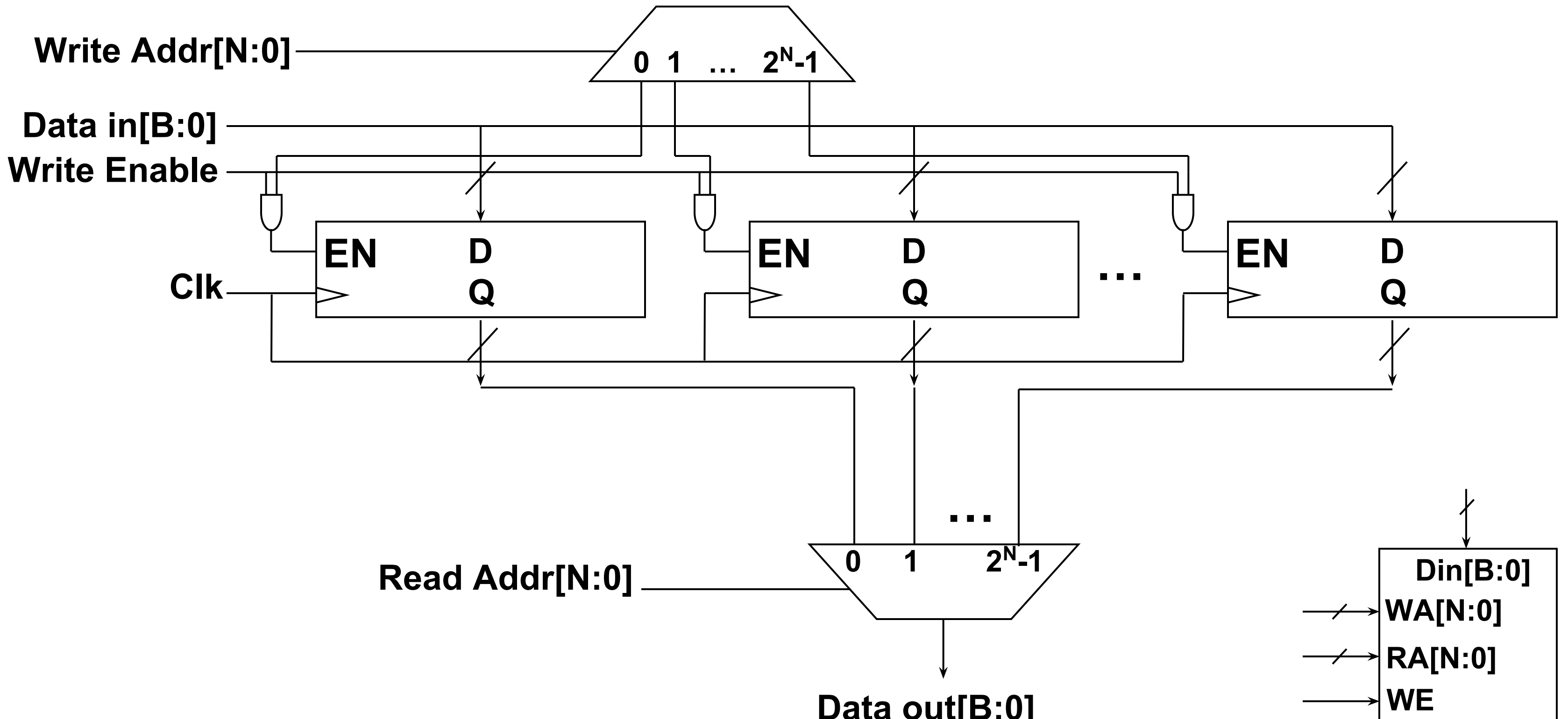
EN	D	CLK	Q_N	Q_{N+1}
X	X	0	0	0
X	X	0	1	1
X	X	1	0	0
X	X	1	1	1
0	X	↑	0	0
0	X	↑	1	1
1	0	↑	X	0
1	1	↑	X	1

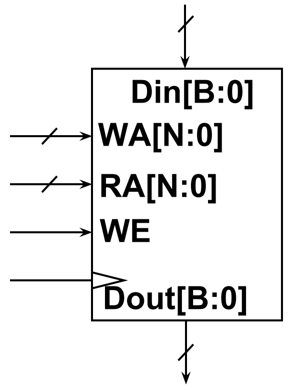
An N-bit wide Register with enable



A Register File

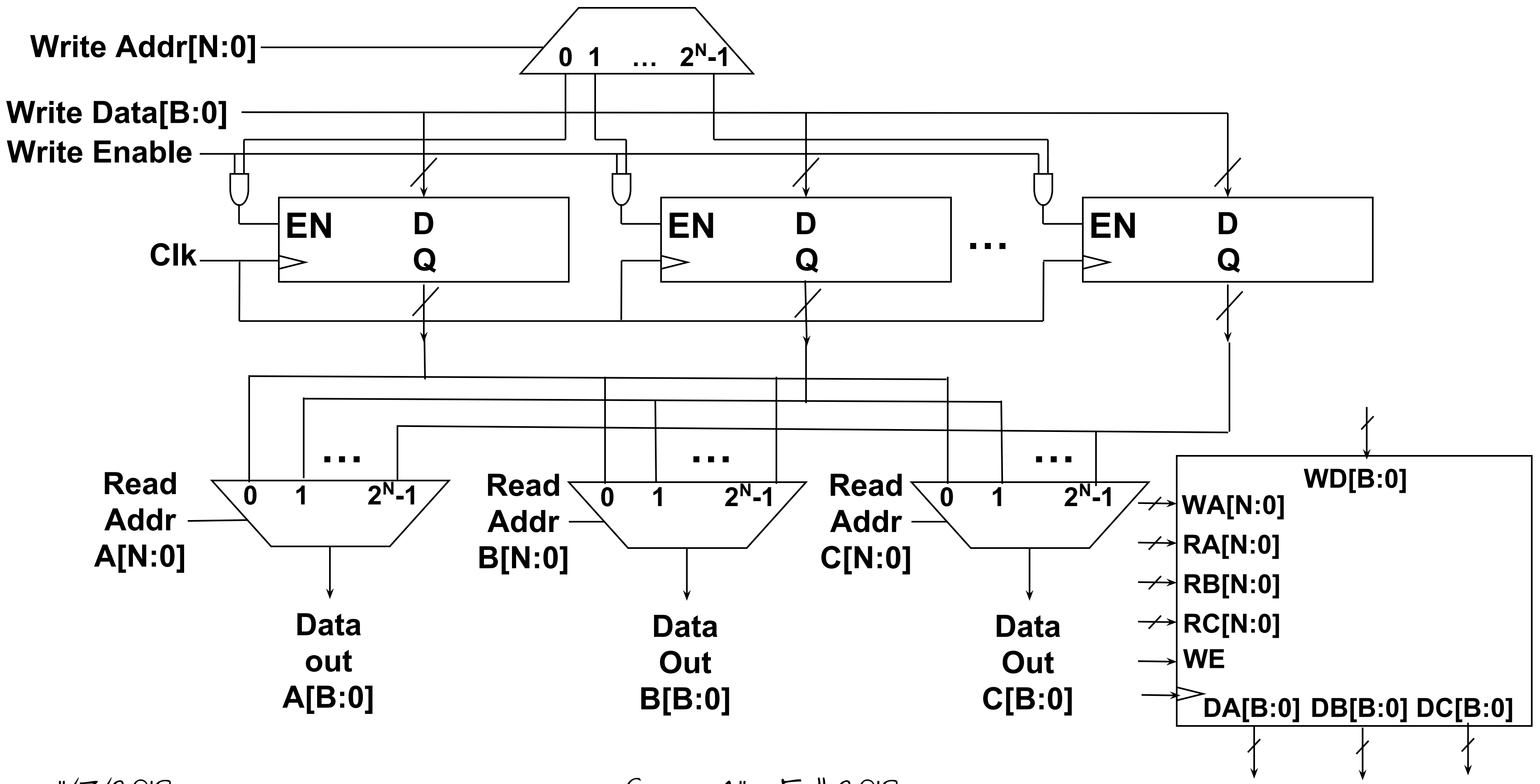
We can also construct an addressable array of registers





A Multi-Ported Register File

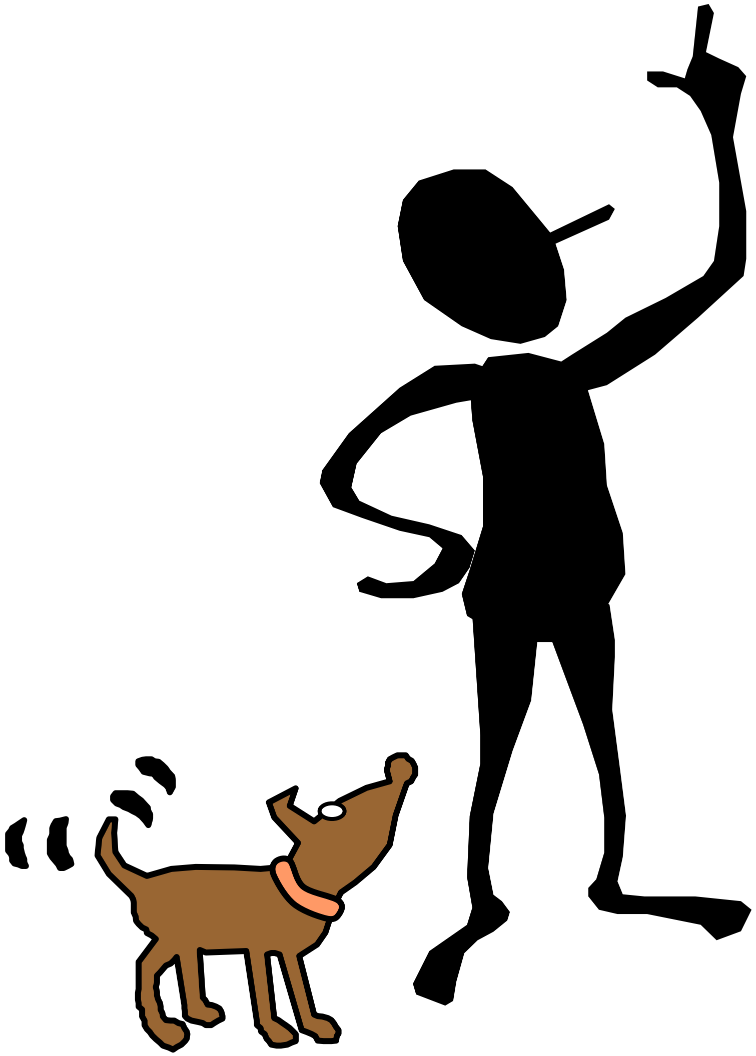
We can add multiple read ports by simply adding more output MUXs



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

	4	3	4	1	4	4	5	2	1	4
R type:	Cond	000	Opcode	S	Rn	Rd	Shift	L A	0	Rm

	4	3	4	1	4	4	4	8
I type:	Cond	001	Opcode	S	Rn	Rd	Shift	Imm

	4	3	5	4	4	12
D type:	Cond	010	AddrMode	Rn	Rd	Imm12

	4	3	5	4	4	5	2	1	4
X type:	Cond	011	AddrMode	Rn	Rd	Shift	L A	0	Rm

	4	3	1	24
B type:	Cond	101	L	Imm24

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

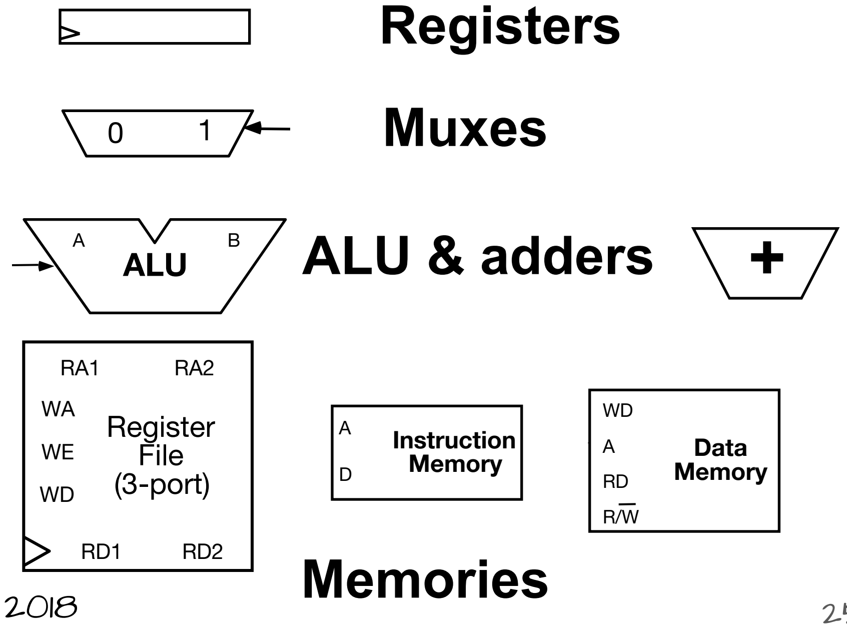
Incremental Featurism:

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:

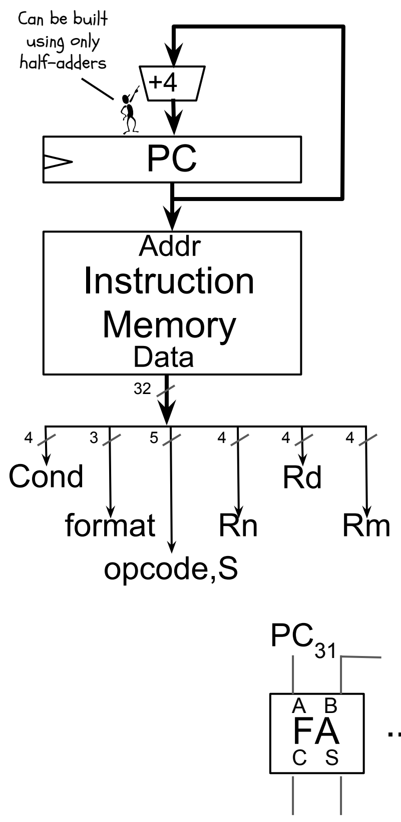
1. 3-Operand ALU instructions
2. ALU w/immediate instructions
2. Load & Store Instructions
3. Branch instructions
4. Leftovers
5. Reset & Exceptions

Our bag of parts:



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

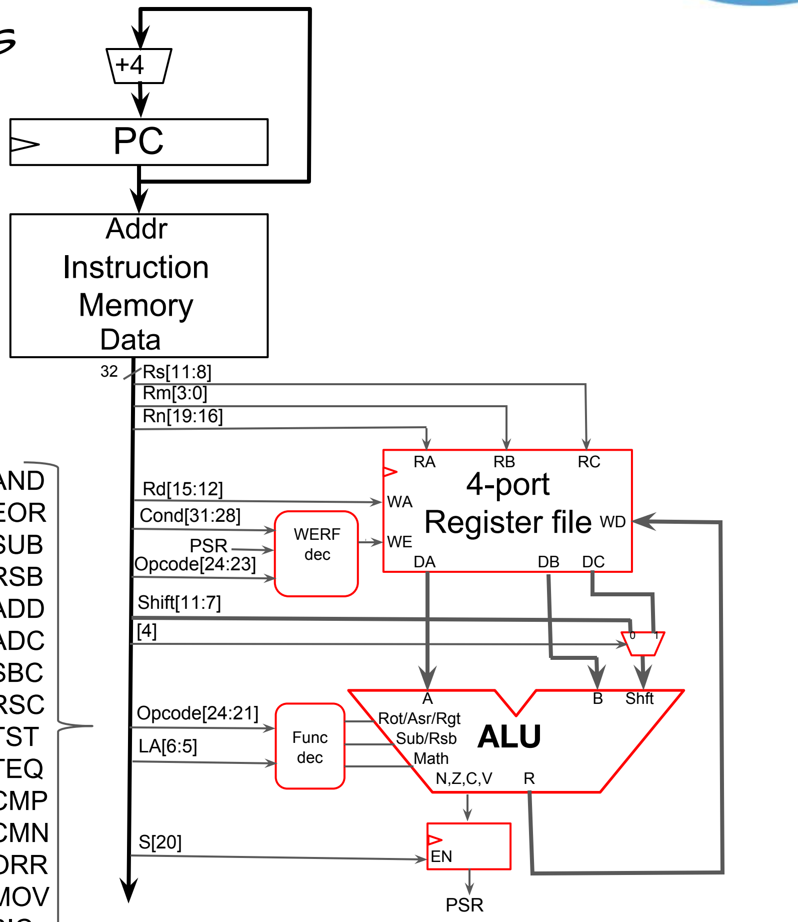
LA - direction and type of shift

S-bit - controls update of PSR

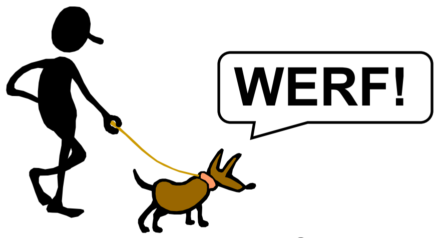
Func decoding from ALU lecture

Register write back controlled by WERF logic

0000	AND
0001	EOR
0010	SUB
0011	RSB
0100	ADD
0101	ADC
0110	SBC
0111	RSC
1000	TST
1001	TEQ
1010	CMP
1011	CMN
1100	ORR
1101	MOV
1110	BIC
1111	MVN

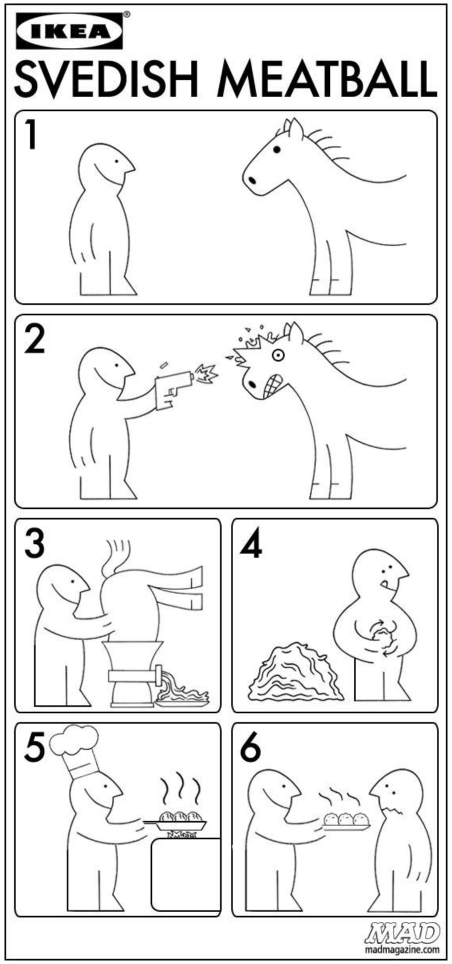


R type:	Cond	000	Opcode	S	Rn	Rd	Shift	LA	0	Rm
	4	3	4	1	4	4	5	2	1	4



Next Time

More instructions…



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..

I_{31}	I_{30}	I_{29}	I_{28}	I_{24}	I_{23}	WE	Notes
X	X	X	X	1	0	0	cmp,cmn,tst,teq
1	1	1	0	0	X	1	Cond = AL
1	1	1	0	X	1	1	Cond = AL
0	0	0	0	0	X	Z	Cond = EQ
0	0	0	0	X	1	Z	Cond = EQ
0	0	0	1	0	X	!Z	Cond = NE
0	0	0	1	X	1	!Z	Cond = NE
...	...	...	...	...	...	...	
1	1	0	0	0	X	!(Z | (N^V))	Cond = GT
1	1	0	0	X	1	!(Z | (N^V))	Cond = GT
1	1	0	1	0	X	Z | (N^V)	Cond = LE
1	1	0	1	X	1	Z | (N^V)	Cond = LE