# 10. Assembly Language, Models of Computation

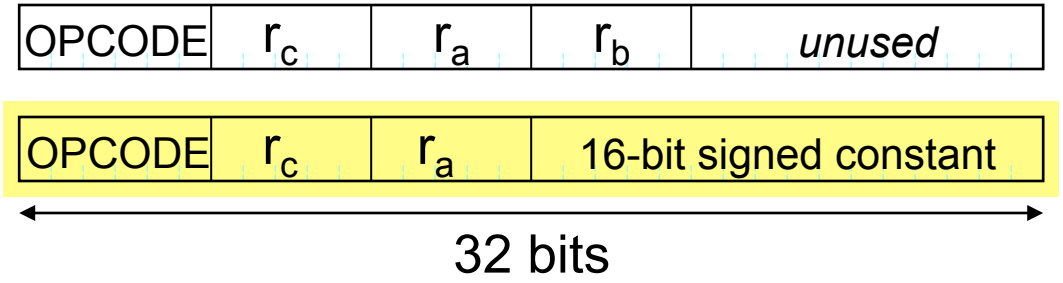
6.004x Computation Structures
Part 2 – Computer Architecture

Copyright © 2015 MIT EECS

# Beta ISA Summary

- Storage:
  - Processor: 32 registers (r31 hardwired to 0) and PC
  - Main memory: Up to 4 GB, 32-bit words, 32-bit byte addresses, 4-byte-aligned accesses

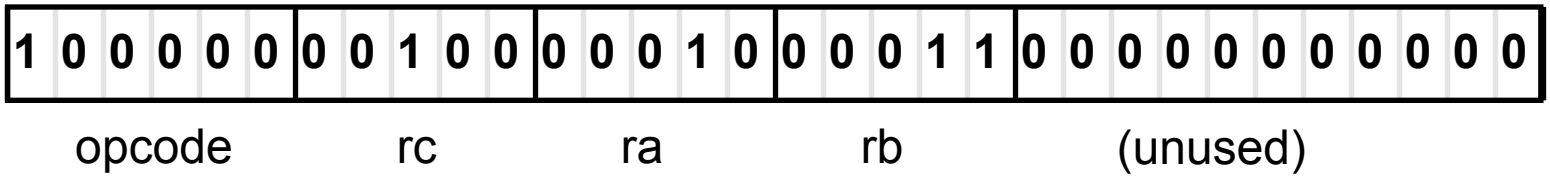
- Instruction formats:




- Instruction classes:
  - ALU: Two input registers, or register and constant
  - Loads and stores: access memory
  - Branches, Jumps: change program counter

# Programming Languages

32-bit (4-byte) ADD instruction:




Means, to the BETA, `Reg[4] ← Reg[2] + Reg[3]`

We'd rather write in *assembly language*:

```
ADD(R2, R3, R4)
```

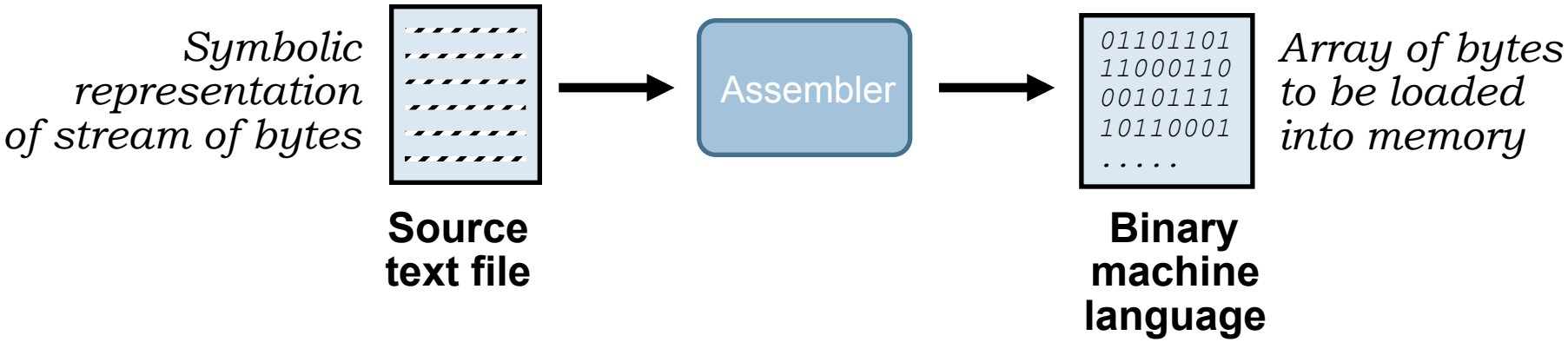
Today

or better yet a *high-level language*:

```
a = b + c;
```

Coming up

# Assembly Language




- Abstracts bit-level representation of instructions and addresses
- We'll learn UASM ("microassembler"), built into BSim
- Main elements:
  - Values
  - Symbols
  - Labels (symbols for addresses)
  - Macros

# Example UASM Source File

```
        N = 12                  // loop index initial value
        ADDC(r31, N, r1)        // r1 = loop index
        ADDC(r31, 1, r0)        // r0 = accumulated product
loop:   MUL(r0, r1, r0)         // r0 = r0 * r1
        SUBC(r1, 1, r1)         /* r1 = r1 - 1 */
        BNE(r1, loop, r31)      // if r1 != 0, NextPC=loop
```

- Comments after //, ignored by assembler (also /*...*/)
- Symbols are symbolic representations of a constant value (they are NOT variables!)
- Labels are symbols for addresses
- Macros expand into sequences of bytes
  - Most frequently, macros are instructions
  - We can use them for other purposes

# How Does It Get Assembled?

**Text input**

```
        N = 12
        ADDC(r31, N, r1)
        ADDC(r31, 1, r0)
loop:   MUL(r0, r1, r0)
        SUBC(r1, 1, r1)
        BNE(r1, loop, r31)
```

**Binary output**

```
110000 00001 11111 00000000 00001100 [0x00]
110000 00000 11111 00000000 00000001 [0x04]
100010 00000 00000 00001 00000000000 [0x08]
                    ...
```

- Load predefined symbols into a symbol table
- Read input line by line
  - Add symbols to symbol table as they are defined
  - Expand macros, translating symbols to values first

**Symbol table**

| Symbol | Value |
|---|---|
| r0 | 0 |
| r1 | 1 |
| ... | |
| r31 | 31 |
| N | 12 |
| loop | 8 |
| | |

# Registers are Predefined Symbols

- r0 = 0, …, r31 = 31
- Treated like normal symbols:

```
ADDC(r31, N, r1)
```

*Substitute symbols with their values*

```
ADDC(31, 12, 1)
```

*Expand macro*

```
110000 00001 11111 00000000 00001100
```

- No "type checking" if you use the wrong opcode…

```
ADDC(r31, r12, r1)
```

```
ADDC(31, 12, 1)

Reg[1] ← Reg[31] + 12
```

```
ADD(r31, N, r1)
```

```
ADD(31, 12, 1)

Reg[1] ← Reg[31] + Reg[12]
```

# Labels and Offsets

**Input file**

```
        N = 12
        ADDC(r31, N, r1)
        ADDC(r31, 1, r0)
loop:   MUL(r0, r1, r0)
        SUBC(r1, 1, r1)
        BNE(r1, loop, r31)
```

**Output file**

```
110000 00001 11111 00000000 00001100 [0x00]
110000 00000 11111 00000000 00000001 [0x04]
100010 00000 00001 00000 00000000000 [0x08]
110001 00001 00001 00000000 00000001 [0x0C]
011101 11111 00001 11111111 11111101 [0x10]
```

```
offset = (label - <addr of BNE/BEQ>)/4 - 1
       = (8 - 16)/4 - 1 = -3
```

- Label value is the address of a memory location
- BEQ/BNE macros compute offset automatically
- Labels hide addresses!

**Symbol table**

| Symbol | Value |
|---|---|
| r0 | 0 |
| r1 | 1 |
| ... | |
| r31 | 31 |
| N | 12 |
| loop | 8 |
| | |

# Mighty Macroinstructions

*Macros* are parameterized abbreviations, or shorthand

```
// Macro to generate 4 consecutive bytes:
.macro consec(n)  n  n+1  n+2  n+3

// Invocation of above macro:
consec(37)
```

Is expanded to

```
⇒ 37 37+1 37+2 37+3 ⇒ 37  38  39  40
```

Here are macros for breaking multi-byte data types into byte-sized chunks

```
// Assemble into bytes, little-endian:
.macro WORD(x) x%256 (x/256)%256
.macro LONG(x) WORD(x) WORD(x >> 16)

. = 0x100
LONG(0xdeadbeef)
```

Has same effect as:

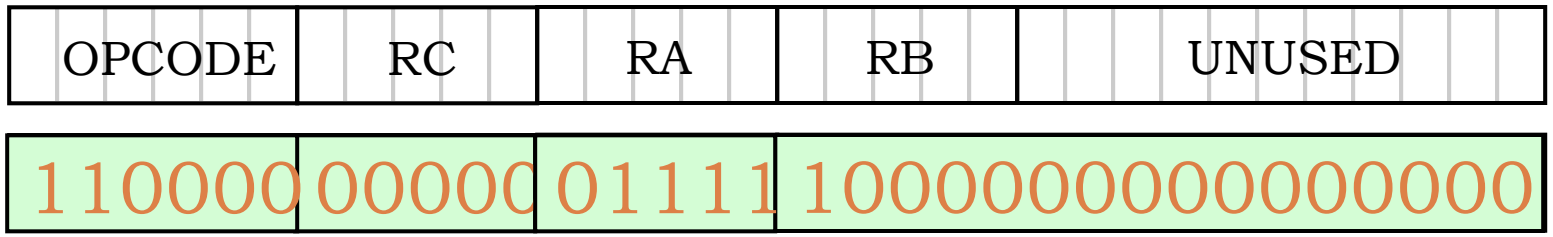
```
      0xef   0xbe   0xad   0xde
Mem: 0x100  0x101 0x102  0x103
```

*Boy, that's hard to read. Maybe, those big-endian types do have a point.*

# Assembly of instructions

-32768 = 1000000000000000

15 = 00000




```
// Assemble Beta op instructions
.macro betaop(OP,RA,RB,RC) {
    .align 4
    LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+((RB%32)<<11))
}

// Assemble Beta opc instructions
.macro betaopc(OP,RA,CC,RC) {
    .align 4
    LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+(CC % 0x10000))
}

// Assemble Beta branch instructions
.macro betabr(OP,RA,RC,LABEL) betaopc(OP,RA,((LABEL-(.+4))>>2),RC)
```

"`.align 4`" ensures instructions will begin on word boundary (i.e., address = 0 mod 4)

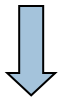
For example:

```
.macro ADDC(RA,C,RC)   betaopc(0x30,RA,C,RC)

ADDC(R15, -32768, R0) --> betaopc(0x30,15,-32768,0)
```

# Example Assembly

```
ADDC(R3,1234,R17)
```



*expand ADDC macro with RA=R3, C=1234, RC=R17*

```
betaopc(0x30,R3,1234,R17)
```

*expand betaopc macro with OP=0x30, RA=R3, CC=1234, RC=R17*

```
.align 4
LONG((0x30<<26)+((R17%32)<<21)+((R3%32)<<16)+(1234 % 0x10000))
```

*expand LONG macro with X=0xC22304D2*

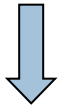
```
WORD(0xC22304D2)    WORD(0xC22304D2 >> 16)
```

*expand first WORD macro with X=0xC22304D2*

```
0xC22304D2%256    (0xC22304D2/256)%256    WORD(0xC223)
```

*evaluate expressions, expand second WORD macro with X=0xC223*

```
0xD2    0x04    0xC223%256    (0xC223/256)%256
```



*evaluate expressions*

```
0xD2    0x04    0x23    0xC2
```

# UASM Macros for Beta Instructions

(defined in beta.uasm)

```
| BETA Instructions:
.macro ADD(RA,RB,RC)   betaop(0x20,RA,RB,RC)
.macro ADDC(RA,C,RC)   betaopc(0x30,RA,C,RC)
.macro AND(RA,RB,RC)   betaop(0x28,RA,RB,RC)
.macro ANDC(RA,C,RC)   betaopc(0x38,RA,C,RC)
.macro MUL(RA,RB,RC)   betaop(0x22,RA,RB,RC)
.macro MULC(RA,C,RC)   betaopc(0x32,RA,C,RC)
   .
   .
   .
.macro LD(RA,CC,RC)    betaopc(0x18,RA,CC,RC)
.macro LD(CC,RC)   betaopc(0x18,R31,CC,RC)
.macro ST(RC,CC,RA)    betaopc(0x19,RA,CC,RC)
.macro ST(RC,CC)   betaopc(0x19,R31,CC,RC)
   .
   .
   .
.macro BEQ(RA,LABEL,RC) betabr(0x1C,RA,RC,LABEL)
.macro BEQ(RA,LABEL)   betabr(0x1C,RA,r31,LABEL)
.macro BNE(RA,LABEL,RC) betabr(0x1D,RA,RC,LABEL)
.macro BNE(RA,LABEL)   betabr(0x1D,RA,r31,LABEL)
```

Convenience macros so we don't have to specify R31…

# Pseudoinstructions

- Convenience macros that expand to one or more real instructions
- Extend set of operations without adding instructions to the ISA

```
// Convenience macros so we don't have to use R31
.macro LD(CC,RC)        LD(R31,CC,RC)
.macro ST(RA,CC)        ST(RA,CC,R31)
.macro BEQ(RA,LABEL)    BEQ(RA,LABEL,R31)
.macro BNE(RA,LABEL)    BNE(RA,LABEL,R31)

.macro MOVE(RA,RC)          ADD(RA,R31,RC)     // Reg[RC] <- Reg[RA]
.macro CMOVE(CC,RC)         ADDC(R31,C,RC)     // Reg[RC] <- C
.macro COM(RA,RC)           XORC(RA,-1,RC)     // Reg[RC] <- ~Reg[RA]
.macro NEG(RB,RC)           SUB(R31,RB,RC)     // Reg[RC] <- -Reg[RB]
.macro NOP()                ADD(R31,R31,R31)   // do nothing

.macro BR(LABEL)            BEQ(R31,LABEL)     // always branch
.macro BR(LABEL,RC)         BEQ(R31,LABEL,RC)  // always branch
.macro CALL(LABEL)          BEQ(R31,LABEL,LP)  // call subroutine
.macro BF(RA,LABEL,RC)      BEQ(RA,LABEL,RC)   // 0 is false
.macro BF(RA,LABEL)         BEQ(RA,LABEL)
.macro BT(RA,LABEL,RC)      BNE(RA,LABEL,RC)   // 1 is true
.macro BT(RA,LABEL)         BNE(RA,LABEL)

// Multi-instruction sequences
.macro PUSH(RA)             ADDC(SP,4,SP)  ST(RA,-4,SP)
.macro POP(RA)              LD(SP,-4,RA)   ADDC(SP,-4,SP)
```

# Factorial with Pseudoinstructions

**Before**

```
       N = 12
       ADDC(r31, N, r1)
       ADDC(r31, 1, r0)
loop:  MUL(r0, r1, r0)
       SUBC(r1, 1, r1)
       BNE(r1, loop, r31)
```

**After**

```
       N = 12
       CMOVE(N, r1)
       CMOVE(1, r0)
loop:  MUL(r0, r1, r0)
       SUBC(r1, 1, r1)
       BNE(r1, loop)
```

# Raw Data

- LONG assembles a 32-bit value
  - Variables
  - Constants > 16 bits

```
N:      LONG(12)
factN:  LONG(0xdeadbeef)
        …
Start:
        LD(N, r1)
        CMOVE(1, r0)
loop:   MUL(r0, r1, r0)
        SUBC(r1, 1, r1)
        BT(r1, loop)
        ST(r0, factN)
```

**Symbol table**

| Symbol | Value |
|---|---|
| … | |
| N | 0 |
| factN | 4 |
| | |

```
LD(r31, N, r1)
```

```
LD(31, 0, 1)
```

```
Reg[1] ← Mem[Reg[31] + 0]
       ← Mem[0]
       ← 12
```

# UASM Expressions and Layout

- Values can be written as expressions
  - Assembler evaluates expressions, they are *not* translated to instructions to compute the value!

```
A = 7 + 3 * 0x0cc41
B = A - 3
```

- The "." (period) symbol means the next byte address to be filled
  - Can read or write to it
  - Useful to control data layout or leave empty space (e.g., for arrays)

```
. = 0x100           // Assemble into 0x100
LONG(0xdeadbeef)
k = .               // Symbol "k" has value 0x104
LONG(0x00dec0de)
. = .+16            // Skip 16 bytes
LONG(0xc0ffeeee)
```

# Summary: Assembly Language

- Low-level language, symbolic representation of sequence of bytes. Abstracts:
  - Bit-level representation of instructions
  - Addresses
- Elements: Values, symbols, labels, macros
- Values can be constants or expressions
- Symbols are symbolic representations of values
- Labels are symbols for addresses
- Macros are expanded to byte sequences:
  - Instructions
  - Pseudoinstructions (translate to 1+ real instructions)
  - Raw data
- Can control where to assemble with "." symbol

# Universality?

- Recall: We say a set of Boolean gates is universal if we can implement any Boolean function using only gates from that set.

- What problems can we solve with a von Neumann computer? (e.g., the Beta)
  - Everything that FSMs can solve?
  - Every problem?
  - Does it depend on the ISA?

- Needed: a mathematical model of computation
  - Prove what can be computed, what can't

# Models of Computation

The roots of computer science stem from the evaluation of many alternative mathematical "models" of computation to determine the classes of computations each could represent.

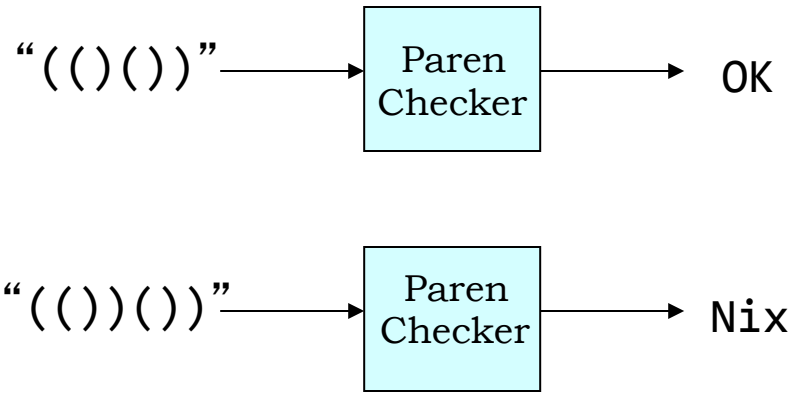
An elusive goal was to find a universal model, capable of representing *all* practical computations…

- switches
- gates
- combinational logic
- memories
- FSMs


Are FSMs the ultimate digital computing device?

# FSM Limitations

Despite their usefulness and flexibility, there are common problems that cannot be solved by any FSM. For instance:



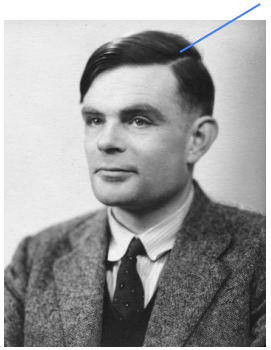

Well-formed Parentheses Checker:

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

Simple, easy to describe.

Can this problem be solved using an FSM??? 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.




Alan Turing

# Turing Machines

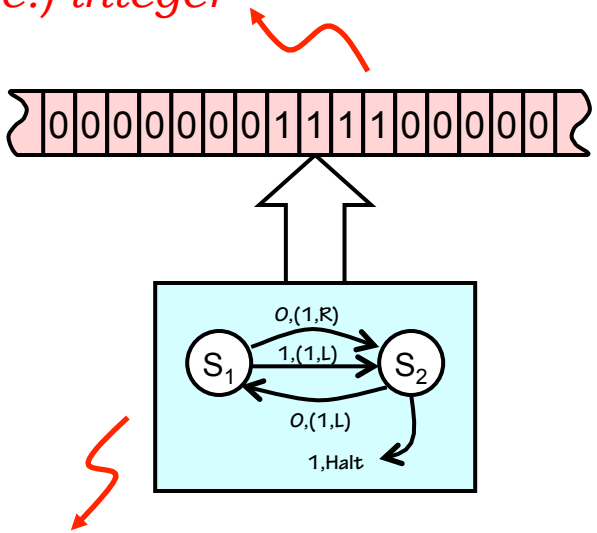
Alan Turing was one of a group of researchers studying alternative models of computation.

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

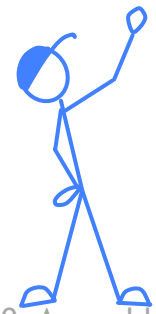
- encode input as symbols on tape
- FSM reads tape/writes symbols/ changes state until it halts
- Answer encoded on tape

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

*Bounded tape configuration can be expressed as a (large!) integer*



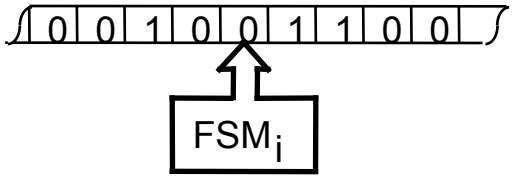

*FSMs can be enumerated and given a (very large) integer index.*

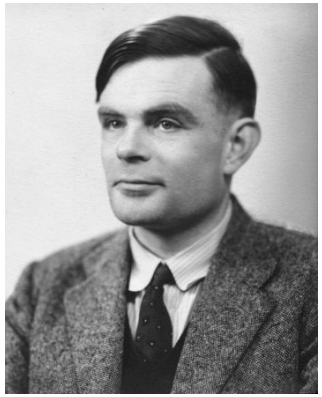


*We can talk about TM 347 running on input 51, producing an answer of 42.*
*TMs as integer functions:*
$y = TM_I[x]$

# Other Models of Computation…

Turing Machines [Turing]



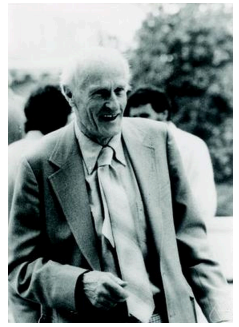
Alan Turing

Recursive Functions [Kleene]

$F(0,x) \equiv x$

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

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



Stephen Kleene

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

Alonzo Church

$\lambda x.\lambda y.xxy$

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

Production Systems [Post, Markov]

Emile Post

$\alpha \rightarrow \beta$

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

# Computability

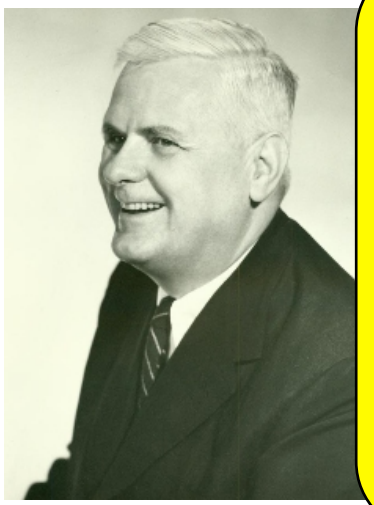
FACT: Each model studied is capable of computing exactly the same set of integer functions!

Proof Technique:

Constructions that translate between models

BIG IDEA:

Computability, independent of computation scheme chosen



## Church's Thesis:

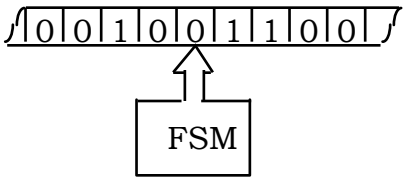
Every discrete function computable by ANY realizable machine is computable by some Turing machine.

f(x) *computable* ⇔ for some k, all x
$f(x) = T_k[x]$

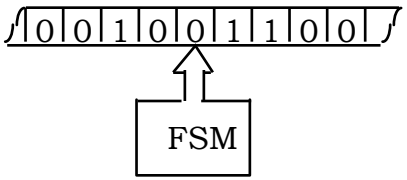
meanwhile...

# Turing machines Galore!

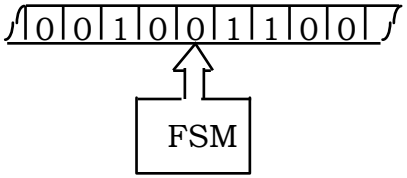
*"special-purpose"* Turing Machines....



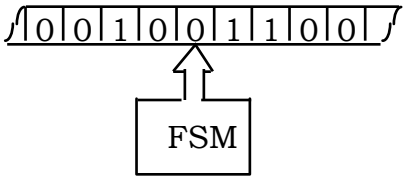

Multiplication



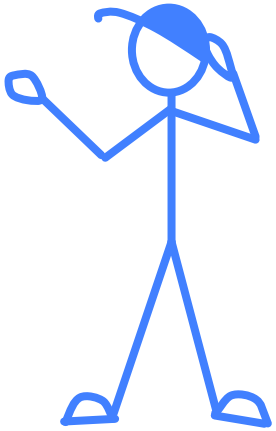

Factorization




Primality Test




Sorting

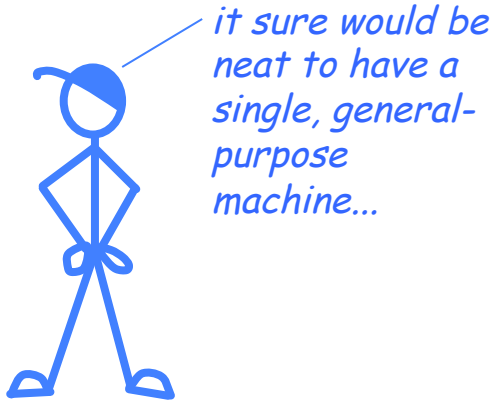


*Is there an alternative to infinitely many ad-hoc Turing Machines?*

# The Universal Function

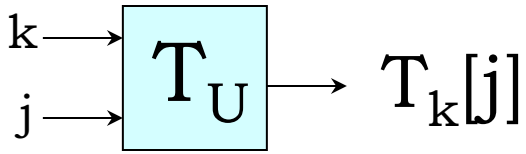
Here's an interesting function to explore: the Universal function, U, defined by

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



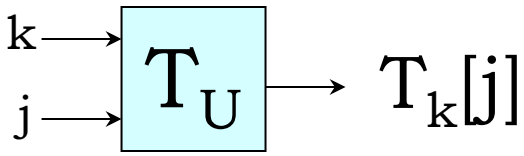

Could this be computable???

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!

# 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!

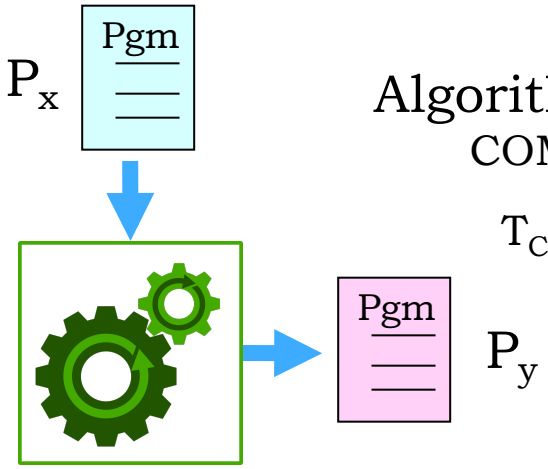
Basic threshold test: Is your computer *Turing Universal*?

- If so, it can emulate every other Turing machine!
- Thus, your computer can compute any computable function

To show your computer is Universal: demonstrate that it can emulate some known UTM.

- Actually given finite memory, can only emulate UTMs + inputs up to a certain size
- This is not a high bar: conditional branches (BEQ) and some simple arithmetic (SUB) are enough.

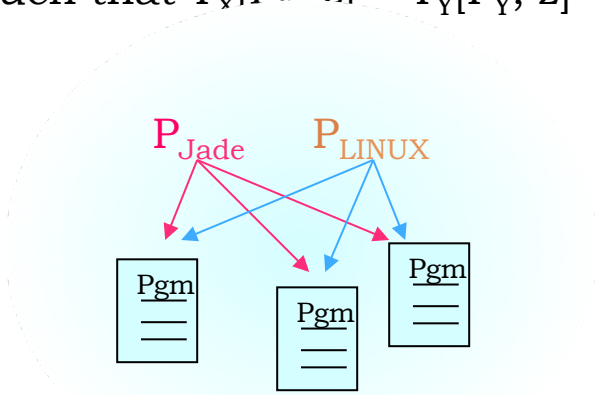
# Coded Algorithms: Key to CS
## data vs hardware

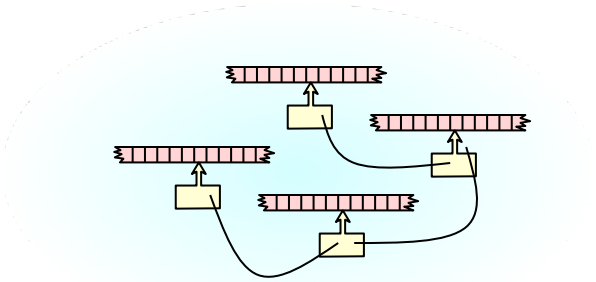



Algorithms as data: enables

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, Sun, ARM, JVM, CLR, …
- Parallel development paths:
  - Language/Software design
  - Interpreter/Hardware design

# Uncomputability (!)

Uncomputable functions: There are well-defined discrete functions that a Turing machine cannot compute

- No algorithm can compute f(x) for arbitrary x in finite number of steps
- Not that we don't know algorithm - can prove no algorithm exists
- Corollary: Finite memory is not the only limiting factor on whether we can solve a problem

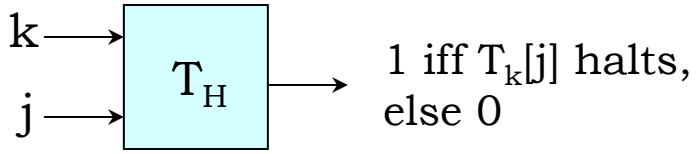
The most famous uncomputable function is the so-called Halting function, $f_H(k, j)$, defined by:

$$f_H(k, j) = \begin{cases} 1 & \text{if } T_k[j] \text{ halts;} \\ 0 & \text{otherwise.} \end{cases}$$

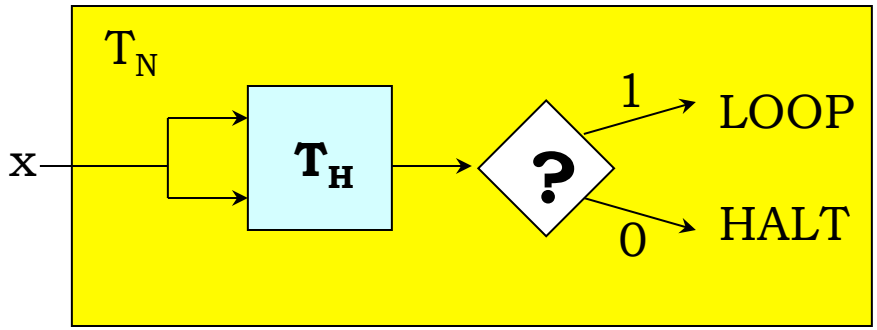
$f_H(k, j)$ determines whether the $k^{th}$ TM halts when given a tape containing j.

# 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!