CS 61C:
Great Ideas in Computer Architecture
Everything is a Number

Instructor:

Randy H. Katz

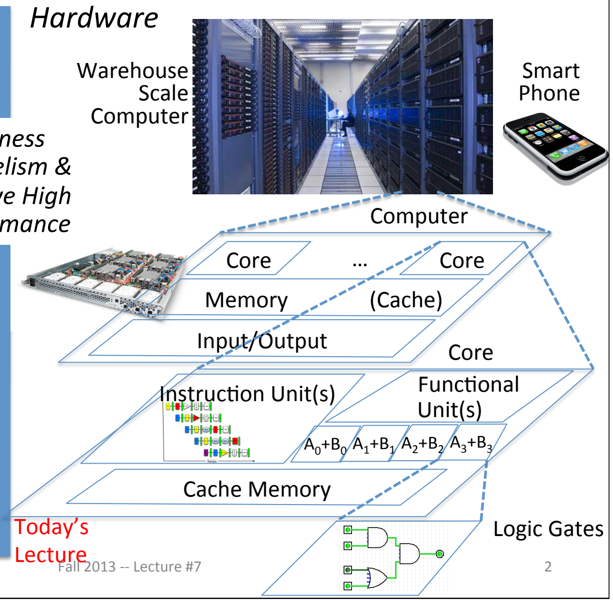
http://inst.eecs.Berkeley.edu/~cs61c/fa13

New-School Machine Structures (It's a bit more complicated!)

Software | *Hardware*

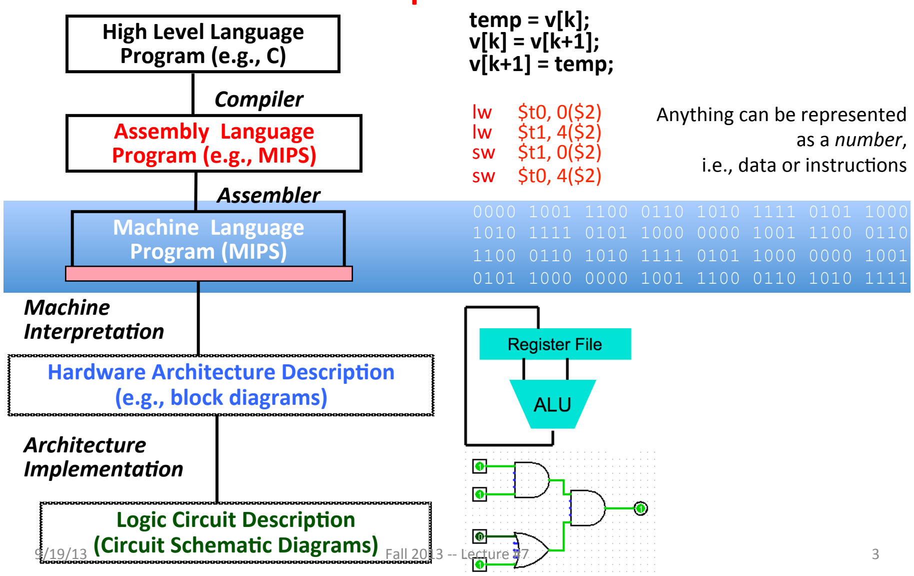
- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Performance



Today's Lecture

Big Idea #1: Levels of Representation/ Interpretation



Review

- C is function oriented; code reuse via functions
  - Jump and link (`jal`) invokes, jump register (`jr $ra`) returns
  - Registers `$a0–$a3` for arguments, `$v0–$v1` for return values
- Stack for spilling registers, nested function calls, C local (automatic) variables

Agenda

- Memory Heap
- Everything is a Number
- Administrivia
- Overflow and Real Numbers
- Technology Break
- Instructions as Numbers
- Assembly Language to Machine Language
- And in Conclusion, …

Agenda

- Memory Heap
- Everything is a Number
- Administrivia
- Overflow and Real Numbers
- Technology Break
- Instructions as Numbers
- Assembly Language to Machine Language
- And in Conclusion, …

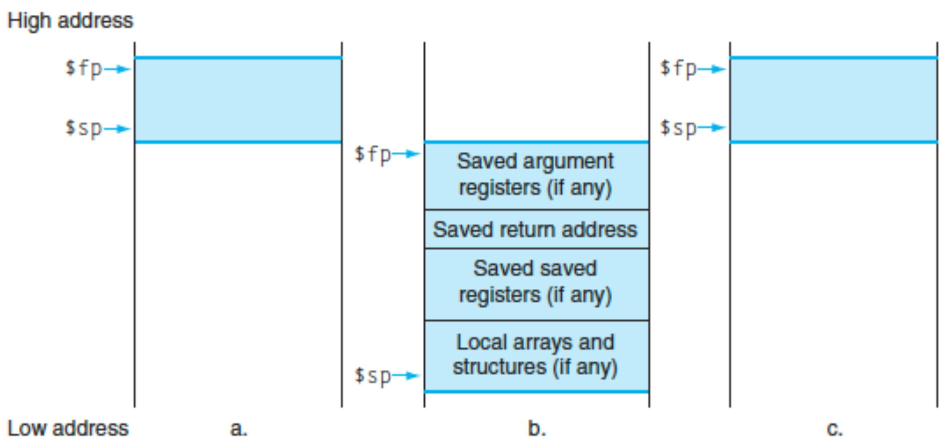
What If a Function Calls a Function? Recursive Function Calls?

- Would clobber values in `$a0` to `$a1` and `$ra`
- What is the solution?

Allocating Space on Stack

- C has two storage classes: automatic and static
  - *Automatic* variables are local to function and discarded when function exits.
  - *Static* variables exist across exits from and entries to procedures
- Use stack for automatic (local) variables that don't fit in registers
- *Procedure frame* or *activation record*: segment of stack with saved registers and local variables
- Some MIPS compilers use a frame pointer (`$fp`) to point to first word of frame
- (29 of 32, 3 left!)

Stack Before, During, After Call



Recursive Function Factorial

```
int fact (int n)
{
  if (n < 1) return (1);
     else return (n * fact(n-1));
}
```

Recursive Function Factorial

```
Fact:
  # adjust stack for 2 items
  addi $sp,$sp,-8
  # save return address
  sw $ra, 4($sp)
  # save argument n
  sw $a0, 0($sp)
  # test for n < 1
  slti $t0,$a0,1
  # if n >= 1, go to L1
  beq $t0,$zero,L1
  # Then part (n==1) return 1
  addi $v0,$zero,1
  # pop 2 items off stack
  addi $sp,$sp,8
  # return to caller
  jr $ra
L1:
  # Else part (n >= 1)
  # arg. gets (n – 1)
  addi $a0,$a0,-1
  # call fact with (n – 1)
  jal fact
  # return from jal: restore n
  lw $a0, 0($sp)
  # restore return address
  lw $ra, 4($sp)
  # adjust sp to pop 2 items
  addi $sp, $sp,8
  # return n * fact (n – 1)
  mul $v0,$a0,$v0
  # return to the caller
  jr $ra
```

mul is a pseudo instruction

Optimized Function Convention

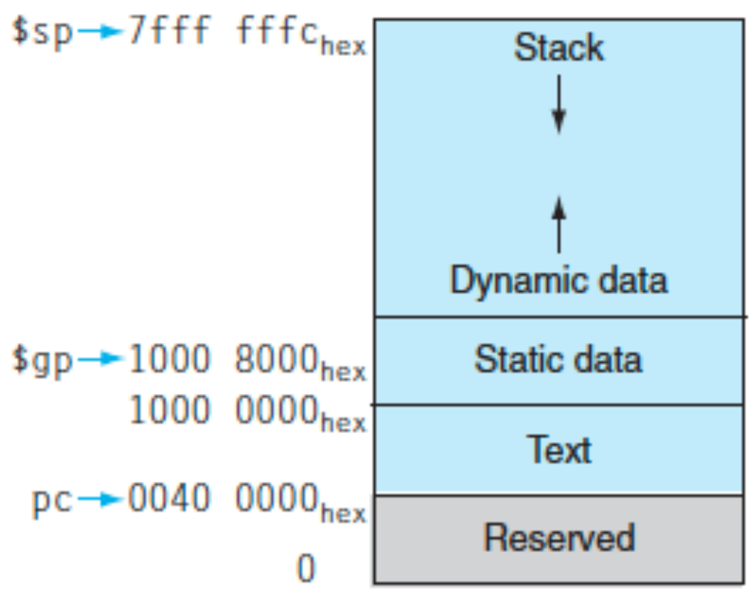
To reduce expensive loads and stores from spilling and restoring registers, MIPS divides registers into two categories:

1. Preserved across function call
   - Caller can rely on values being unchanged
   - `$ra`, `$sp`, `$gp`, `$fp`, "saved registers" `$s0`- `$s7`
2. Not preserved across function call
   - Caller *cannot* rely on values being unchanged
   - Return value registers `$v0`,`$v1`, Argument registers `$a0`-`$a3`, "temporary registers" `$t0`-`$t9`

Where is the Stack in Memory?

- MIPS convention
- Stack starts in high memory and grows down
  - Hexadecimal (base 16) : 7fff fffc_{hex}
- MIPS programs (*text segment*) in low end
  - 0040 0000_{hex}
- *Static data segment* (constants and other static variables) above text for static variables
  - MIPS convention *global pointer* (`$gp`) points to static
  - (30 of 32, 2 left! – will see when talk about OS)
- *Heap* above static for data structures that grow and shrink ; grows up to high addresses

MIPS Memory Allocation



Register Allocation and Numbering

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Which statement is FALSE?

- ☐ MIPS uses `jal` to invoke a function and `jr` to return from a function
- ☐ `jal` saves PC+1 in `$ra`
- ☐ The callee can use temporary registers (`$t`*i*) without saving and restoring them
- ☐ The caller can rely on save registers (`$s`*i*) without fear of callee changing them

Which statement is FALSE?

- ☐ MIPS uses `jal` to invoke a function and `jr` to return from a function
- ☐ `jal` saves PC+1 in `$ra`
- ☐ The callee can use temporary registers (`$ti`) without saving and restoring them
- ☐ The caller can rely on save registers (`$si`) without fear of callee changing them

17

Agenda

- Memory Heap
- Everything is a Number
- Administrivia
- Overflow and Real Numbers
- Technology Break
- Instructions as Numbers
- Assembly Language to Machine Language
- Summary

9/19/13 Fall 2013 -- Lecture #7 18

Key Concepts

- Inside computers, everything is a number
- But everything is of a fixed size
  - 8-bit bytes, 16-bit half words, 32-bit words, 64-bit double words, …
- Integer and floating point operations can lead to results too big to store within their representations: *overflow/underflow*

Number Representation

- Value of i-th digit is $d \times Base^i$ where i starts at 0 and increases from right to left:
- $123_{10} = 1_{10} \times 10_{10}{}^2 + 2_{10} \times 10_{10}{}^1 + 3_{10} \times 10_{10}{}^0$
  $= 1\text{x}100_{10} + 2\text{x}10_{10} + 3\text{x}1_{10}$
  $= 100_{10} + 20_{10} + 3_{10}$
  $= 123_{10}$
- Binary (Base 2), Hexadecimal (Base 16), Decimal (Base 10) different ways to represent an integer
  - We use 1_{two}, 5_{ten}, 10_{hex} to be clearer
    (vs. 1_2, 4_8, 5_{10}, 10_{16})

Number Representation

- Hexadecimal digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- $FFF_{hex} = 15_{ten} x\ 16_{ten}{}^2 + 15_{ten} x\ 16_{ten}{}^1 + 15_{ten} x\ 16_{ten}{}^0$
  $= 3840_{ten} + 240_{ten} + 15_{ten}$
  $= 4095_{ten}$
- $1111\ 1111\ 1111_{two} = FFF_{hex} = 4095_{ten}$
- May put blanks every group of binary, octal, or hexadecimal digits to make it easier to parse, like commas in decimal

Signed and Unsigned Integers

- C, C++, and Java have *signed integers*, e.g., 7, -255:

```
int x, y, z;
```

- C, C++ also have *unsigned integers*, which are used for addresses
- 32-bit word can represent 2^{32} binary numbers
- Unsigned integers in 32 bit word represent 0 to 2^{32}-1 (4,294,967,295)

Unsigned Integers

0000 0000 0000 0000 0000 0000 0000 0000_{two} = 0_{ten}
0000 0000 0000 0000 0000 0000 0000 0001_{two} = 1_{ten}
0000 0000 0000 0000 0000 0000 0000 0010_{two} = 2_{ten}
... ...
0111 1111 1111 1111 1111 1111 1111 1101_{two} = $2{,}147{,}483{,}645_{ten}$
0111 1111 1111 1111 1111 1111 1111 1110_{two} = $2{,}147{,}483{,}646_{ten}$
0111 1111 1111 1111 1111 1111 1111 1111_{two} = $2{,}147{,}483{,}647_{ten}$
1000 0000 0000 0000 0000 0000 0000 0000_{two} = $2{,}147{,}483{,}648_{ten}$
1000 0000 0000 0000 0000 0000 0000 0001_{two} = $2{,}147{,}483{,}649_{ten}$
1000 0000 0000 0000 0000 0000 0000 0010_{two} = $2{,}147{,}483{,}650_{ten}$
... ...
1111 1111 1111 1111 1111 1111 1111 1101_{two} = $4{,}294{,}967{,}293_{ten}$
1111 1111 1111 1111 1111 1111 1111 1110_{two} = $4{,}294{,}967{,}294_{ten}$
1111 1111 1111 1111 1111 1111 1111 1111_{two} = $4{,}294{,}967{,}295_{ten}$

Signed Integers and Two's Complement Representation

- Signed integers in C; want ½ numbers <0, want ½ numbers >0, and want one 0
- *Two's complement* treats 0 as positive, so 32-bit word represents 2^{32} integers from -2^{31} (–2,147,483,648) to $2^{31}-1$ (2,147,483,647)
  – Note: one negative number with no positive version
  – Book lists some other options, all of which are worse
  – Every computers uses two's complement today
- *Most significant bit* (leftmost) is the *sign bit*, since 0 means positive (including 0), 1 means negative
  – Bit 31 is most significant, bit 0 is least significant

Two's Complement Integers

Sign Bit

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
... ...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$
... ...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

Twos Complement Examples

- Assume for simplicity 4 bit width, -8 to +7 represented

3	0011	3	0011	-3	1101
+2	0010	+ (-2)	1110	+ (-2)	1110
5	0101	1	1 0001	-5	1 1011

7	0111	-8	1000
+1	0001	+ (-1)	1111
-8	1000	+7	1 0111
Overflow!		***Underflow!***	

Carry into MSB = Carry Out MSB

Carry into MSB ≠ Carry Out MSB

Suppose we had a 5 bit word. What integers can be represented in two's complement?

- ☐ -32 to +31
- ☐ 0 to +31
- ☐ -16 to +15
- ☐ -15 to +16

27

Suppose we had a 5 bit word. What integers can be represented in two's complement?

- ☐ -32 to +31
- ☐ 0 to +31
- ☐ $\boxed{\text{-16 to +15}}$
- ☐ -15 to +16

28

MIPS Logical Instructions

- Useful to operate on fields of bits within a word
  - e.g., characters within a word (8 bits)
- Operations to pack /unpack bits into words
- Called *logical operations*

Logical operations	C operators	Java operators	MIPS instructions
Bit-by-bit AND	&	&	**and**
Bit-by-bit OR	|	|	**or**
Bit-by-bit NOT	~	~	**nor**
Shift left	<<	<<	**sll**
Shift right	>>	>>>	**srl**

Bit-by-bit Definition

Operation	Input	Input	Output
AND	0	0	0
AND	0	1	0
AND	1	0	0
AND	1	1	1
OR	0	0	0
OR	0	1	1
OR	1	0	1
OR	1	1	1
NOR	0	0	1
NOR	0	1	0
NOR	1	0	0
NOR	1	1	0

Examples

- If register $t2 contains
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- And register $t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of $t0 after:
  and $t0,$t1,$t2 # reg $t0 = reg $t1 & reg $t2

Examples

- If register $t2 contains
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- And register $t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of $t0 after:
  and $t0,$t1,$t2 # reg $t0 = reg $t1 & reg $t2
  0000 0000 0000 0000 0000 1100 0000 0000_{two}

Examples

- If register $t2 contains
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- And register $t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of $t0 after:
  or $t0,$t1,$t2 # reg $t0 = reg $t1 | reg $t2

Examples

- If register $t2 contains
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- And register $t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of $t0 after:
  or $t0,$t1,$t2 # reg $t0 = reg $t1 | reg $t2
  0000 0000 0000 0000 0011 1101 1100 0000_{two}

Examples

- If register \$t2 contains
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- And register \$t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of \$t0 after:
  nor \$t0,\$t1,\$zero # reg \$t0 = ~ (reg \$t1 | 0)

Examples

- If register \$t2 contains
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- And register \$t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of \$t0 after:
  nor \$t0,\$t1,\$zero # reg \$t0 = ~ (reg \$t1 | 0)
  1111 1111 1111 1111 1100 0011 1111 1111_{two}

Shifting

- Shift left logical moves n bits to the left (insert 0s into empty bits)
  - Same as multiplying by 2^n for two's complement number
- For example, if register $s0 contained
  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{two} = 9_{ten}$
- If executed sll $s0, $s0, 4, result is:
  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{two} = 144_{ten}$
- And $9_{ten} \times 2_{ten}^4 = 9_{ten} \times 16_{ten} = 144_{ten}$
- Shift right logical moves n bits to the right (insert 0s into empty bits)
  - NOT same as dividing by 2n (negative numbers fail)

Shifting

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register $s0 contained
  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 1001_{two} = 25_{ten}$
- If executed sra $s0, $s0, 4, result is:

Shifting

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register $s0 contained

  0000 0000 0000 0000 0000 0000 0001 1001$_{two}$= 25$_{ten}$
- If executed sra $s0, $s0, 4, result is:

  0000 0000 0000 0000 0000 0000 0000 0001$_{two}$= 1$_{ten}$

Shifting

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register $s0 contained

  1111 1111 1111 1111 1111 1111 1110 0111$_{two}$= -25$_{ten}$
- If executed sra $s0, $s0, 4, result is:

Shifting

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register $s0 contained

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_{two} = -25_{ten}$
- If executed sra $s0, $s0, 4, result is:

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
- Unfortunately, this is NOT same as dividing by 2n
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 0

Impact of Signed and Unsigned Integers on Instruction Sets

- What (if any) instructions affected?
  - Load word, store word?
  - branch equal, branch not equal?
  - and, or, sll, srl?
  - add, sub, mult, div?
  - slti (set less than immediate)?

C provides two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||) while MIPS doesn't. Why?

- ☐ Logical operations AND and OR do & and | while conditional branches do && and ||
- ☐ The previous statement has it backwards: && and || logical ops, & and | are branches
- ☐ They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C

43

C provides two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||) while MIPS doesn't. Why?

- ☐ Logical operations AND and OR do & and | while conditional branches do && and ||
- ☐ The previous statement has it backwards: && and || logical ops, & and | are branches
- ☐ They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C

44

Peer Instruction Answer

- C provides two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||) while MIPS doesn't. Why?

  Logical operations AND and OR implement & and | while conditional branches implement && and ||

Reason:

e.g., && is logical `and`: true && true is true and everything else is false

& is bitwise `and`: e.g., (1010_{two} & 1000_{two}) = 1000_{two}

Agenda

- Memory Heap
- Everything is a Number
- Administrivia
- Overflow and Real Numbers
- Instructions as Numbers
- Technology Break
- Assembly Language to Machine Language
- And in Conclusion, …

Administrivia

- HW #3 Due Sunday @ 11:59:59
- Project #1 Part 1 Due Sunday @ 11:59:59
- Midterm on the horizon:
  - 10/17 (4 weeks), 6-9 PM
  - It's going to be *completely* multiple choice!

"And in Conclusion, …"

- Program can interpret binary number as unsigned integer, two's complement signed integer, floating point number, ASCII characters, Unicode characters, …
- Integers have largest positive and largest negative numbers, but represent all in between
  - Two's comp. weirdness is one extra negative numInteger and floating point operations can lead to results too big to store within their representations: overflow/underflow
- Floating point is an approximation of reals
- Everything is a (binary) number in a computer
  - Instructions and data; stored program concept