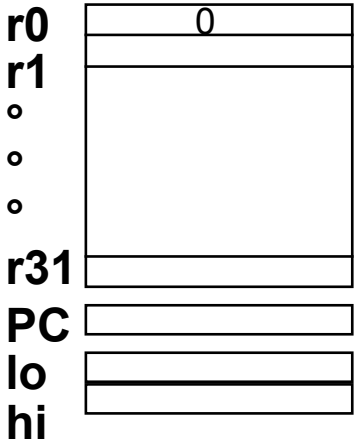
CS61CL Machine Structures

Lec 6 – Number Representation

David Culler
Electrical Engineering and Computer Sciences
University of California, Berkeley

Review: MIPS R3000



Programmable storage

- 2^32 x bytes
- 31 x 32-bit GPRs (R0=0)
- 32 x 32-bit FP regs (paired DP)
- HI, LO, PC

Data types ?

Format ?

Addressing Modes?

Arithmetic logical

Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU,
AddI, AddIU, SLTI, SLTIU, AndI, OrI, XorI, *LUI*
SLL, SRL, SRA, SLLV, SRLV, SRAV

Memory Access

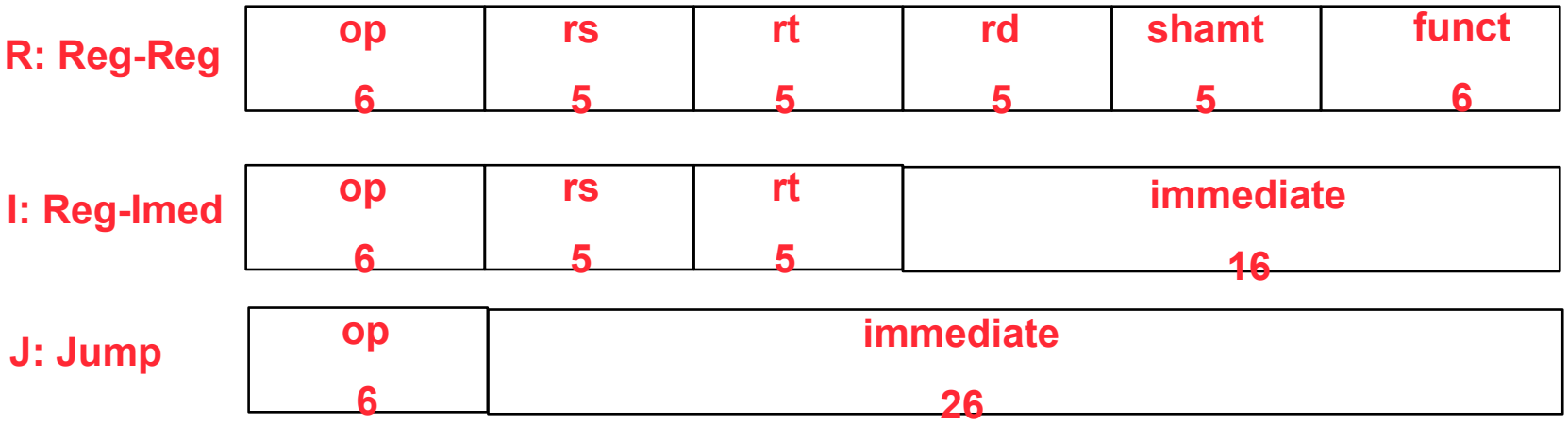
LB, LBU, LH, LHU, LW, LWL,LWR
SB, SH, SW, SWL, SWR

Control

32-bit instructions on word boundary

J, JAL, JR, JALR
BEq, BNE, BLEZ,BGTZ,BLTZ,BGEZ,BLTZAL,BGEZAL

MIPS Instruction Format

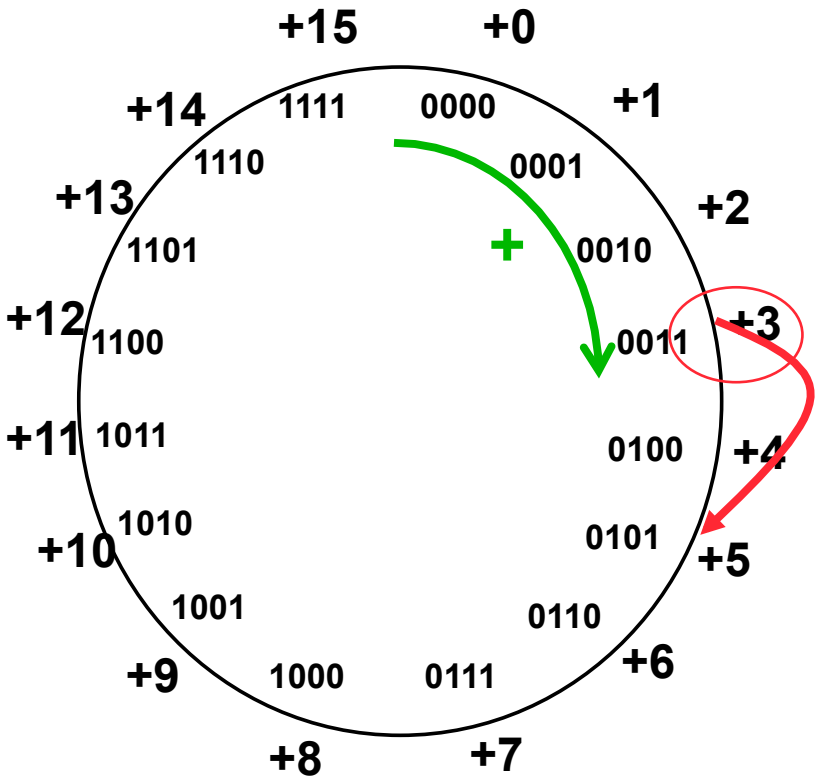


- **Reg-Reg instructions (op == 0)**
  - add, sub, and, or, nor, xor, slt — R[rd] := R[rs] *funct* R[rt]; pc:=pc+4
  - sll. srl, sra — R[rd] := R[rt] *shft* shamt
- **Reg-Immed (op != 0)**
  - addi, andi, ori, xori, lui, — R[rt] := R[rs] op Im16
  - addiu, slti, sltiu
  - lw, lh, lhu, lb, lbu — R[rt] := Mem[R[rs] + signEx(Im16)]*
  - sw, sh, sb — Mem[R[rs] + signEx(Im16)] := R[rt]

Computer Number Systems

- **We all take positional notation for granted**
  - **$D_{k-1}\ D_{k-2}\ \ldots D_0$ represents $D_{k-1}B^{k-1} + D_{k-2}B^{k-2} + \ \ldots + D_0\ B^0$ where $B \in \{ 0, \ldots, B-1 \}$**
- **We all understand how to compare, add, subtract these numbers**
  - **Add each position, write down the position bit and possibly carry to the next position**
- **Computers represent finite number systems**
  - **Generally radix 2**
  - **$B_{k-1}\ B_{k-2}\ \ldots B_0$ represents $B_{k-1}2^{k-1} + B_{-2}2^{k-2} + \ \ldots + B_0\ 2^0$ where $B \in \{ 0,1 \}$**

Unsigned Numbers - Addition



Example: 3 + 2 = 5

Unsigned binary addition

Is just addition, base 2

Add the bits in each position and carry

```
   1
  0 0 1 1
+ 0 0 1 0
---------
  0 1 0 1
```

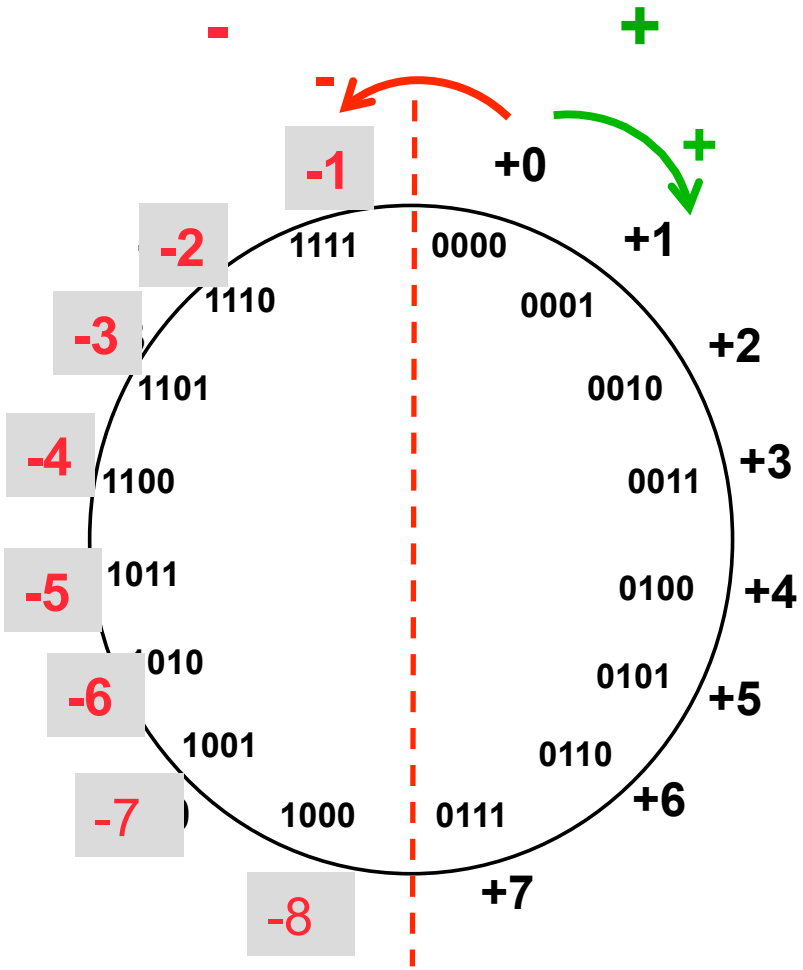
Number Systems

- Desirable properties:
  - Efficient encoding (2^n bit patterns. How many numbers?)
  - Positive and negative
    - » Closure (almost) under addition and subtraction
      - Except when overflow
    - » Representation of positive numbers same in most systems
    - » Major differences are in how negative numbers are represented
  - Efficient operations
    - » Comparison: =, <, >
    - » Addition, Subtraction
    - » Detection of overflow
  - Algebraic properties?
    - » Closure under negation?
    - » A == B iff A – B == 0

Which one did you learn in 2nd grade?

- Three Major schemes:
  - sign and magnitude
  - ones complement
  - twos complement
  - (excess notation)

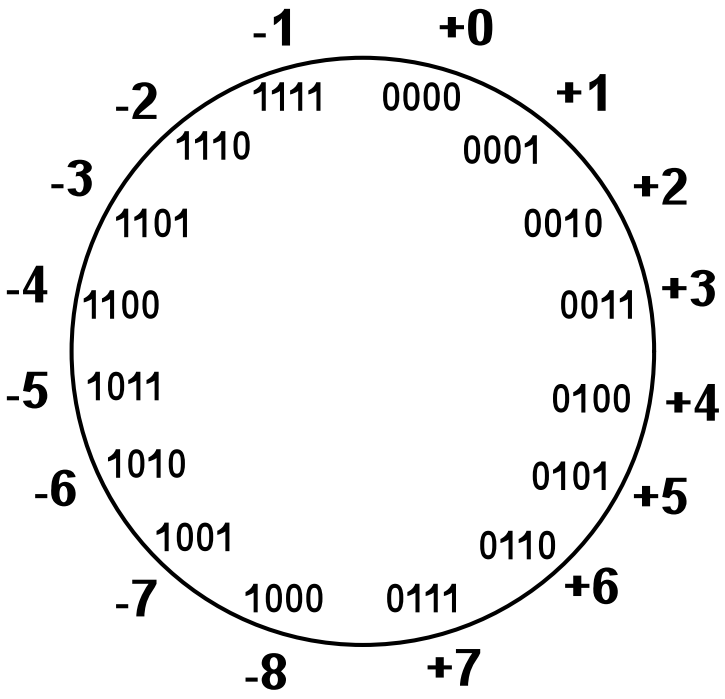
How do we represent signed numbers?

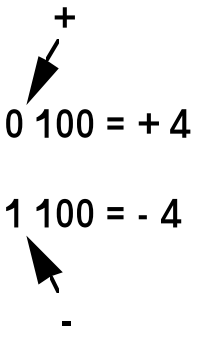


- What algebraic mapping would have this assignment?

Twos Complement

$B_{k-1}\ B_{k-2}\ \ldots B_0$ represents $-B_{k-1}2^{k-1} + B_{-2}2^{k-2} + \ \ldots + B_0\ 2^0$

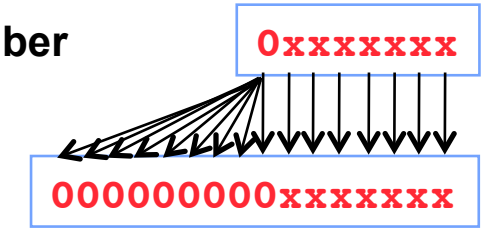




Sign Extension

$$-0*2^7 + B_6 2^6 + B_5 2^5 + B_4 2^4 + B_3 2^3 + B_2 2^2 + B_1 2^1 + B_0 2^0$$

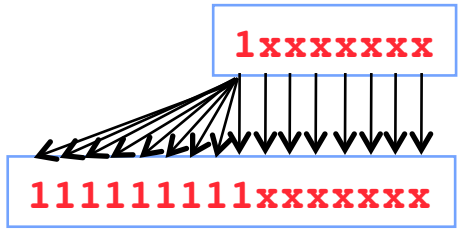
Positive Number



$$-0*2^{15} + \ldots + 0*2^7 + B_6 2^6 + B_5 2^5 + B_4 2^4 + B_3 2^3 + B_2 2^2 + B_1 2^1 + B_0 2^0$$

Negative Number

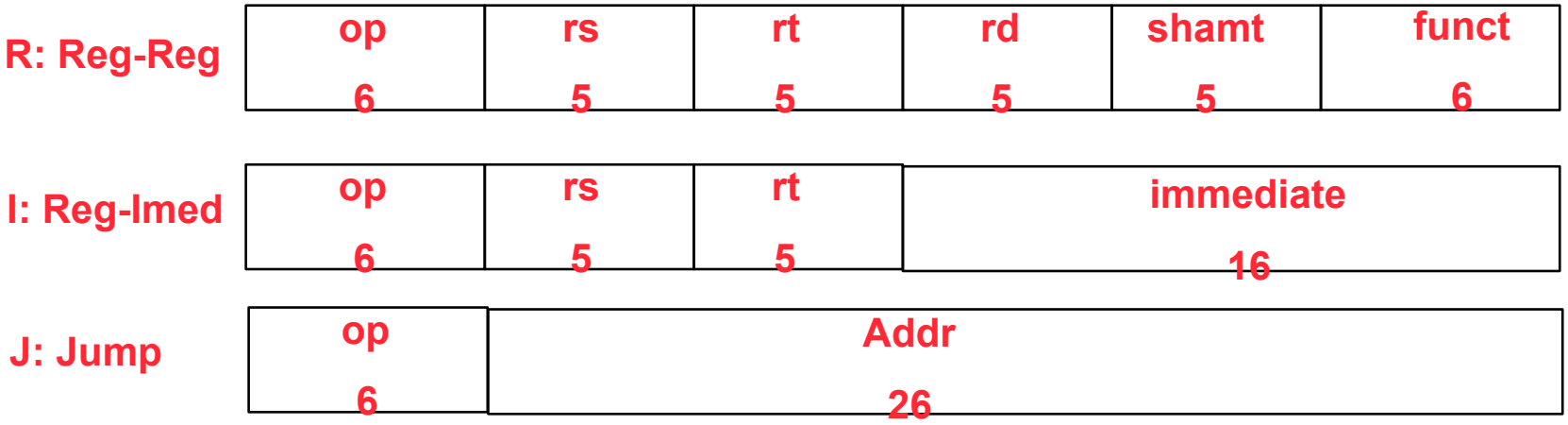
$$-2^7 + B_6 2^6 + B_5 2^5 + B_4 2^4 + B_3 2^3 + B_2 2^2 + B_1 2^1 + B_0 2^0$$



$$-2^{15} + 2^{14} \ldots + 2^7 + B_6 2^6 + B_5 2^5 + B_4 2^4 + B_3 2^3 + B_2 2^2 + B_1 2^1 + B_0 2^0$$

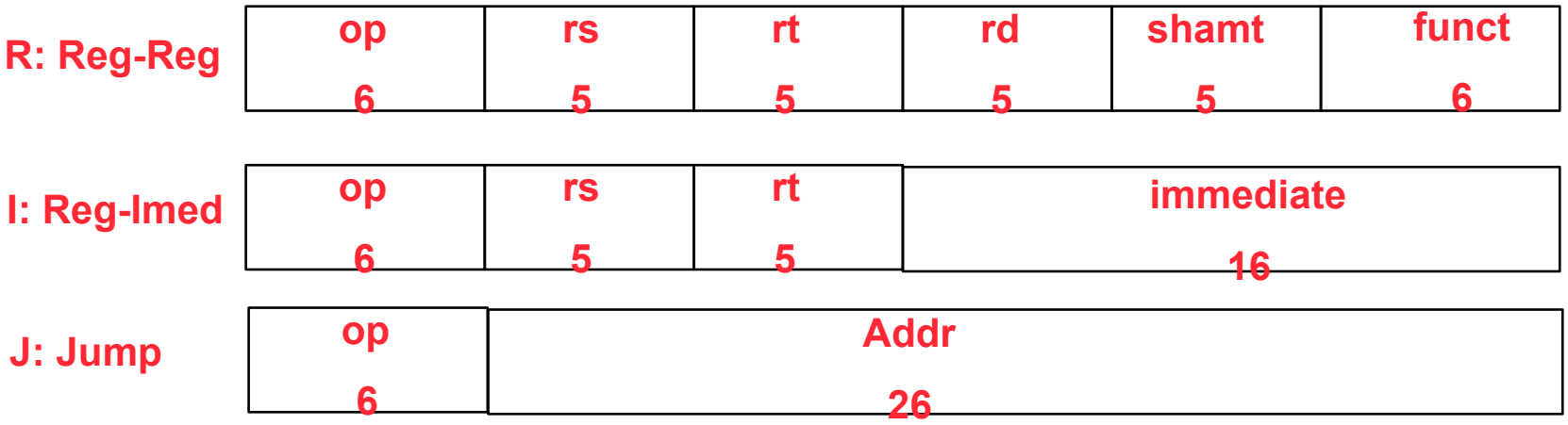
$$-2^7$$

MIPS Instruction Format



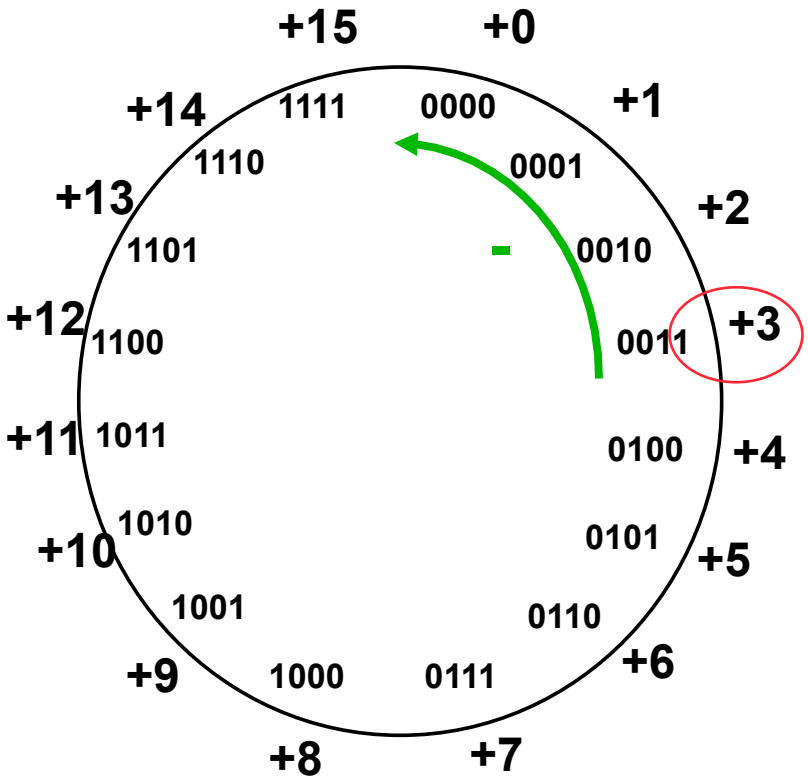
- **Reg-Reg instructions (op == 0)**
- **Reg-Immed (op != 0)**
- **Jumps**
  - **j** PC := $PC_{31..28}$ || addr || 00
  - **jal** PC := $PC_{31..28}$ || addr || 00; R[31] := PC + 4
  - **jr** PC := R[rs]

MIPS Instruction Format



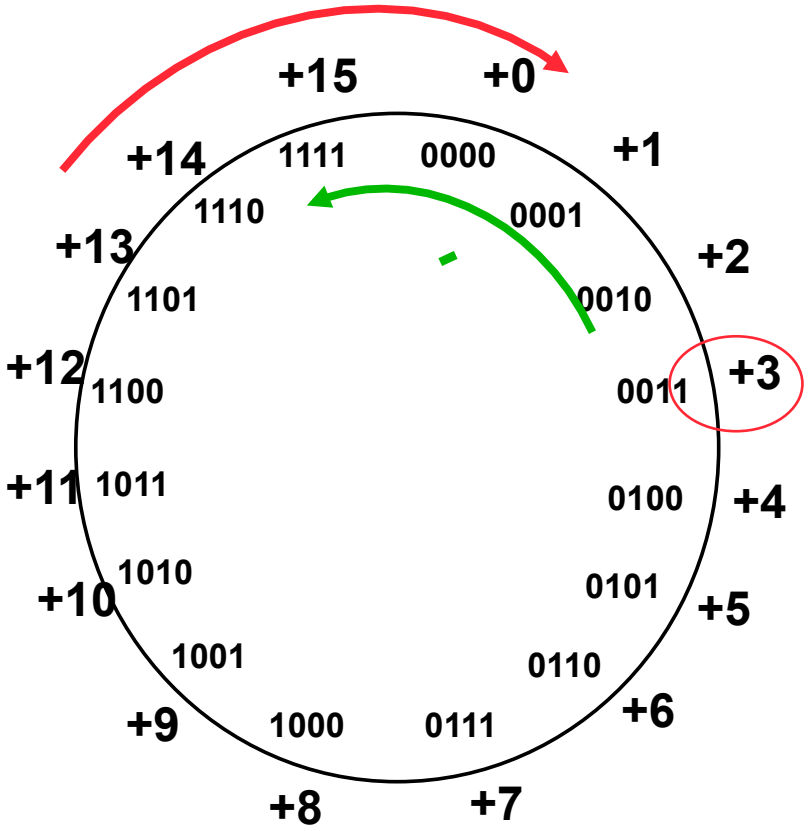
- **Reg-Reg instructions (op == 0)**
- **Reg-Immed (op != 0)**
- **Jumps**
- **Branches**
  - **BEQ, BNE** PC := (R[rs] ?= R[rt]) ? PC + signEx(im16) : PC+4
  - BLT, BGT, BLE, BGTE are pseudo ops
  - Move and LI are pseudo ops too

So what about subtraction?



- Subtraction – or addition of a negative number – should move "backwards" on the number wheel?

Finite representation?



- What happens when $A + B > 2^N - 1$?
  - Overflow
  - Detect?
- What happens when $A - B < 0$?
  - Negative numbers?
  - Borrow out?

Administrative Issues

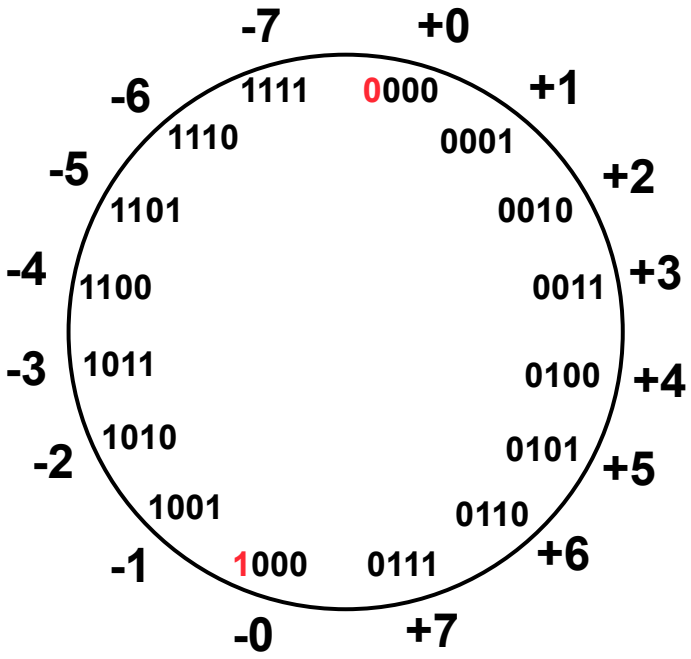
- HW 5 is due tonight before midnight
- Proj 1 is due Friday
  - your submission must build by running make in the directory containing the source files - that is what the autograder does
  - you should confirm that it passes proj1.tests/test.proj1
  - autograder will run additional tests, so generate your own
    - » feel free to share them
    - » can run reference asm16 to generate .o and .syms
- Midterm is wed 10/7 in 10 evans
  - alternate is mon 10/5 in 310 Soda, send email, and do poll
  - review session is Sunday evening (TBA)
  - homework this week is to study previous midterms
- Monday office hours moved to tues 9-11 in QC^3

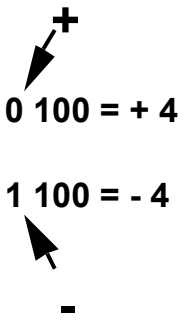
Can't get enough C fun&games…

- **Write a program in C that reproduces itself**
  - **the output is the text of the program**
  - **you can compile the output with gcc and it will produce a program that reproduces itself…**
- **Hint:**
  - **Write the following sentence twice, the second in quotes. "Write the following sentence twice, the second in quotes."**
  - **it is way easier in scheme**

Other choices for negative numbers

Sign and Magnitude





Example: N = 4

High order bit is sign: 0 = positive (or zero), 1 = negative

Remaining low order bits is the magnitude: 0 (000) thru 7 (111)

Number range for n bits = +/- $2^{n-1} - 1$

Representations for 0?

Operations: =, <, >, +, - ???

Ones Complement

Bit manipulation:

simply complement each of the bits

0111 -> 1000

Algebraically …

N is positive number, then $\overline{N}$ is its negative 1's complement

$$\overline{N} = (2^n - 1) - N$$

Example: 1's complement of 7

2^4 10000

-1 - 00001

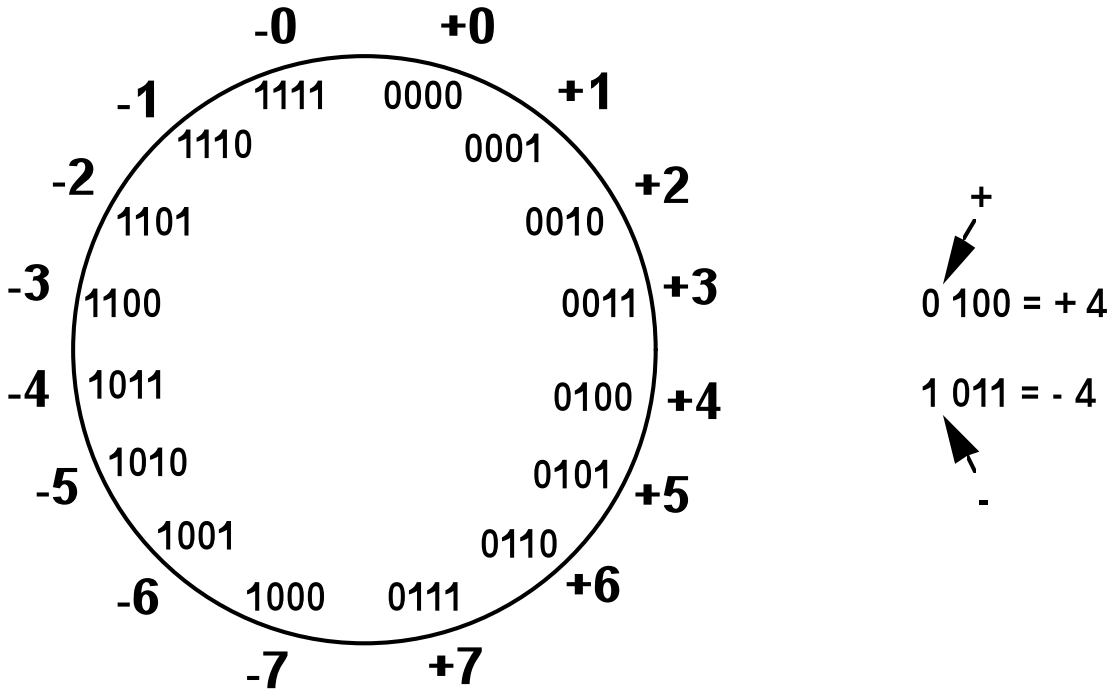
1111

-7 - 0111

1000

-7 in 1's comp.

Ones Complement on the number wheel



Subtraction implemented by addition & 1's complement

Sign is easy to determine

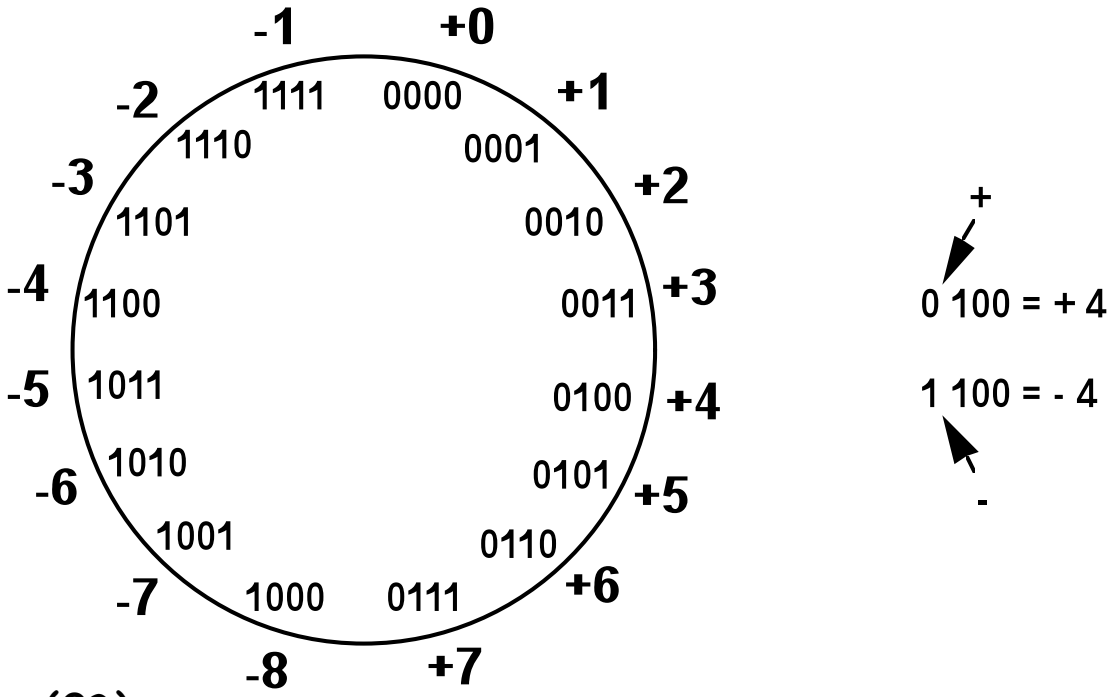
Closure under negation. If A can be represented, so can -A

Still two representations of 0!

If A = B then is A – B == 0 ?

Addition is **almost** clockwise advance, like unsigned

Twos Complement number wheel



Easy to determine sign (0?)

Only one representation for 0

Addition and subtraction just as in unsigned case

Simple comparison: A < B iff A – B < 0

One more negative number than positive number

- one number has no additive inverse

Twos Complement (algebraically)

$N^* = 2^n - N$

Example: Twos complement of 7

$2^4 = 10000$

sub 7 = 0111

1001 = repr. of -7

Example: Twos complement of -7

$2^4 = 10000$

sub -7 = 1001

0111 = repr. of 7

Bit manipulation:

Twos complement: take bitwise complement and add one

0111 -> 1000 + 1 -> 1001 (representation of -7)

1001 -> 0110 + 1 -> 0111 (representation of 7)

How is addition performed in each number system?

- Operands may be positive or negative

Twos Complement Addition

Perform unsigned addition and

Discard the carry out.

4	0100	-4	1100
+ 3	0011	+ (-3)	1101
7	0111	-7	11001

Overflow?

4	0100	-4	1100
- 3	1101	+ 3	0011
1	10001	-1	1111

Simpler addition scheme makes twos complement the most common choice for integer number systems within digital systems

Sign Magnitude Addition

Operand have same sign: unsigned addition of magnitudes

result sign bit is the same as the operands' sign

4	0100	-4	1100
+ 3	0011	+ (-3)	1011
7	0111	-7	1111

Operands have different signs:

subtract smaller from larger and keep sign of the larger

4	0100	-4	1100
- 3	1011	+ 3	0011
1	0001	-1	1001

Ones complement addition

Perform unsigned addition, then add in the end-around carry

4	0100	-4	1011
+ 3	0011	+ (-3)	1100
7	0111	-7	10111
		End around carry	→ 1
			1000

4	0100	-4	1011
- 3	1100	+ 3	0011
1	10000	-1	1110
End around carry	→ 1		
	0001		

2s Comp: ignore the carry out

-M + N when N > M:

$$M^* + N = (2^n - M) + N = 2^n + (N - M)$$

Ignoring carry-out is just like subtracting 2^n

-M + -N where $N + M \leq 2^{n-1}$

$$-M + (-N) = M^* + N^* = (2^n - M) + (2^n - N)$$

$$= 2^n - (M + N) + 2^n$$

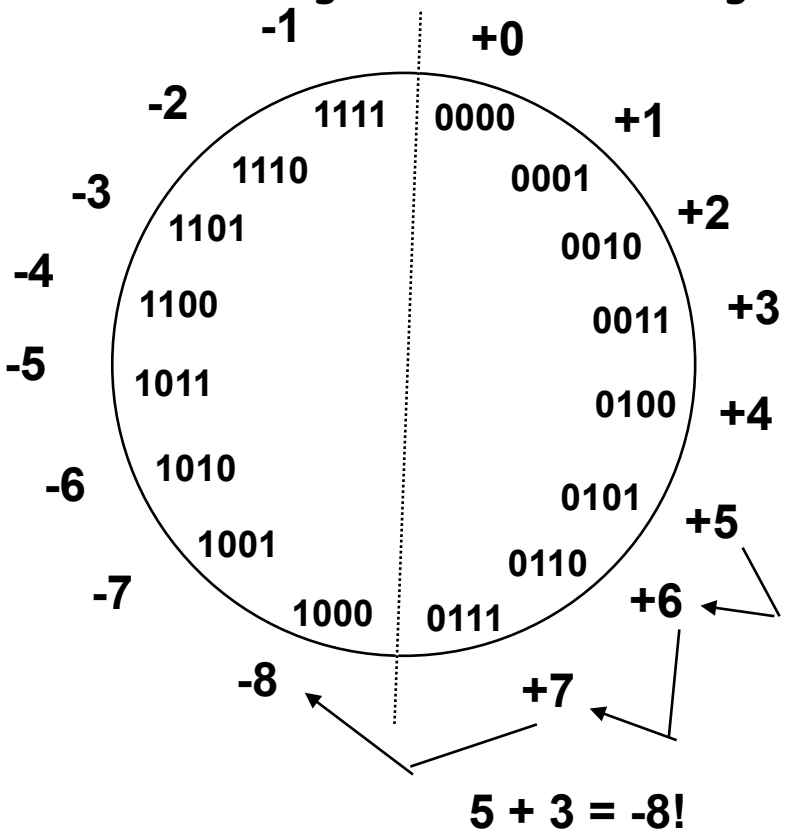
After ignoring the carry, this is just the right twos compl. representation for -(M + N)!

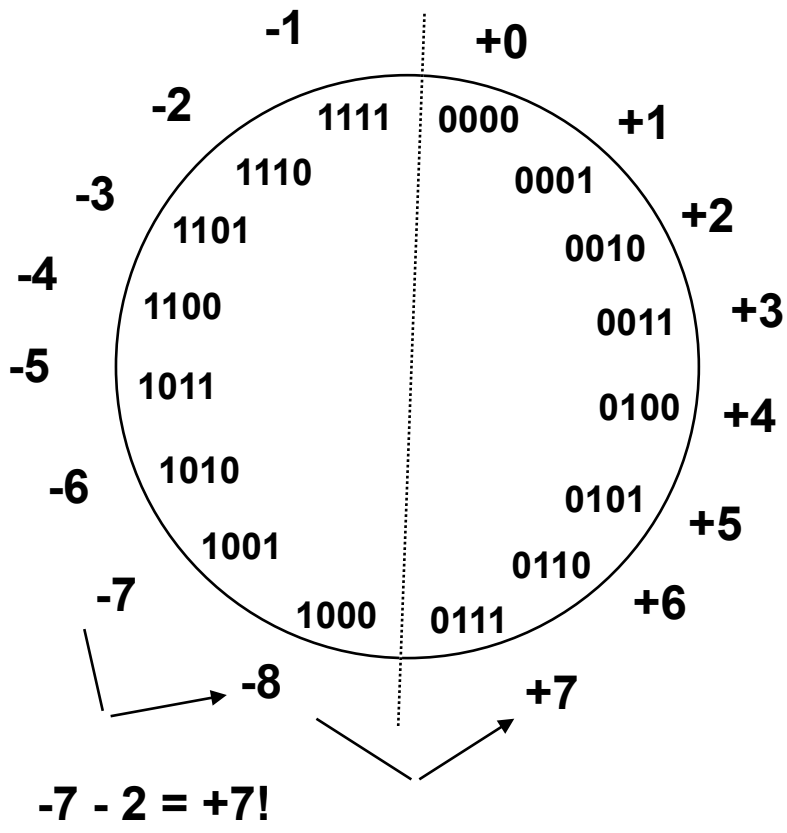
2s Complement Overflow

How can you tell an overflow occurred?

Add two positive numbers to get a negative number

or two negative numbers to get a positive number





2s comp. Overflow Detection

	0 1 1 1	
5	0 1 0 1	
3	0 0 1 1	
-8	1 0 0 0	

Overflow

	1 0 0 0	
-7	1 0 0 1	
-2	1 1 0 0	
7	1 0 1 1 1	

Overflow

	0 0 0 0	
5	0 1 0 1	
2	0 0 1 0	
7	0 1 1 1	

No overflow

	1 1 1 1	
-3	1 1 0 1	
-5	1 0 1 1	
-8	1 1 0 0 0	

No overflow

Overflow occurs when carry in to sign does not equal carry out