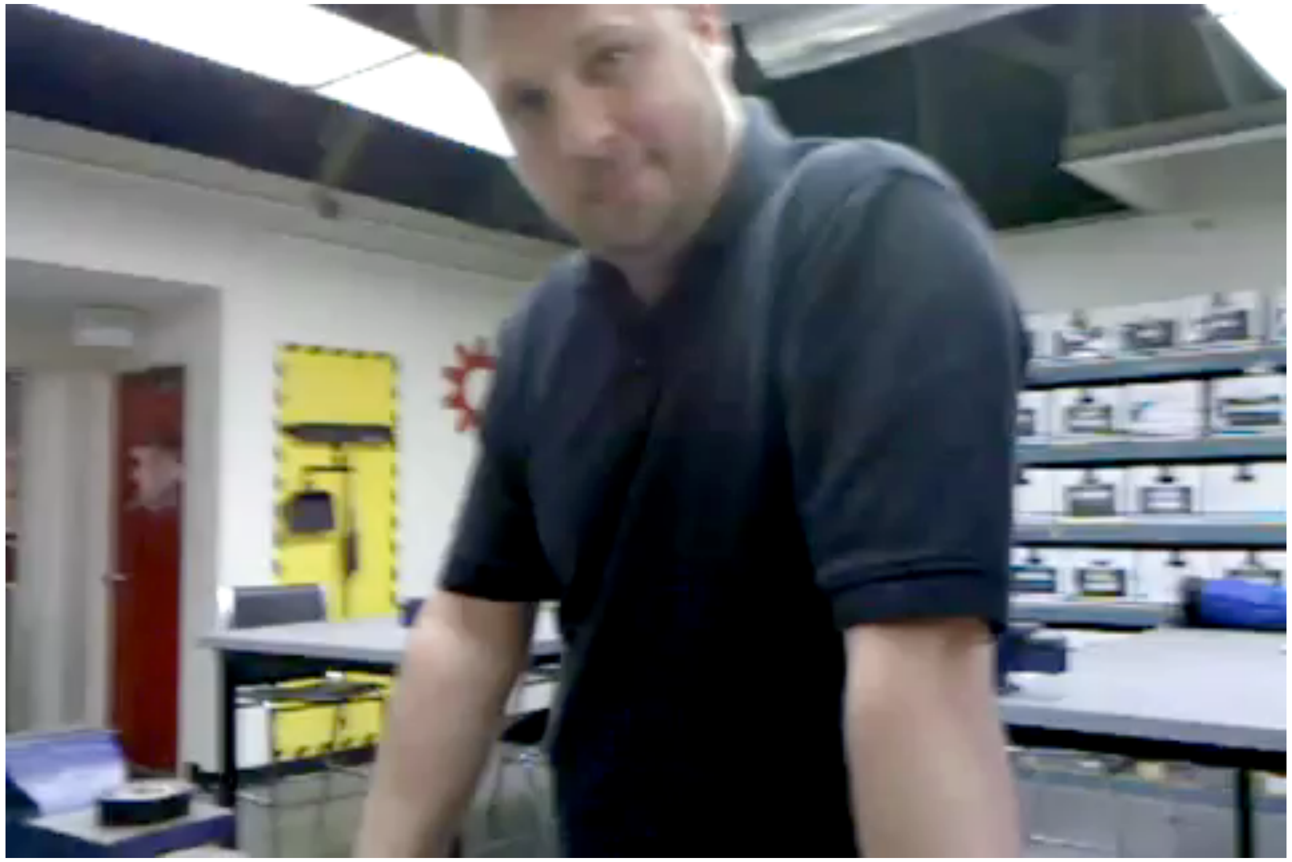
http://www.instructables.com/id/Android-G1-Serial-To-Arduino/?ALLSTEPS

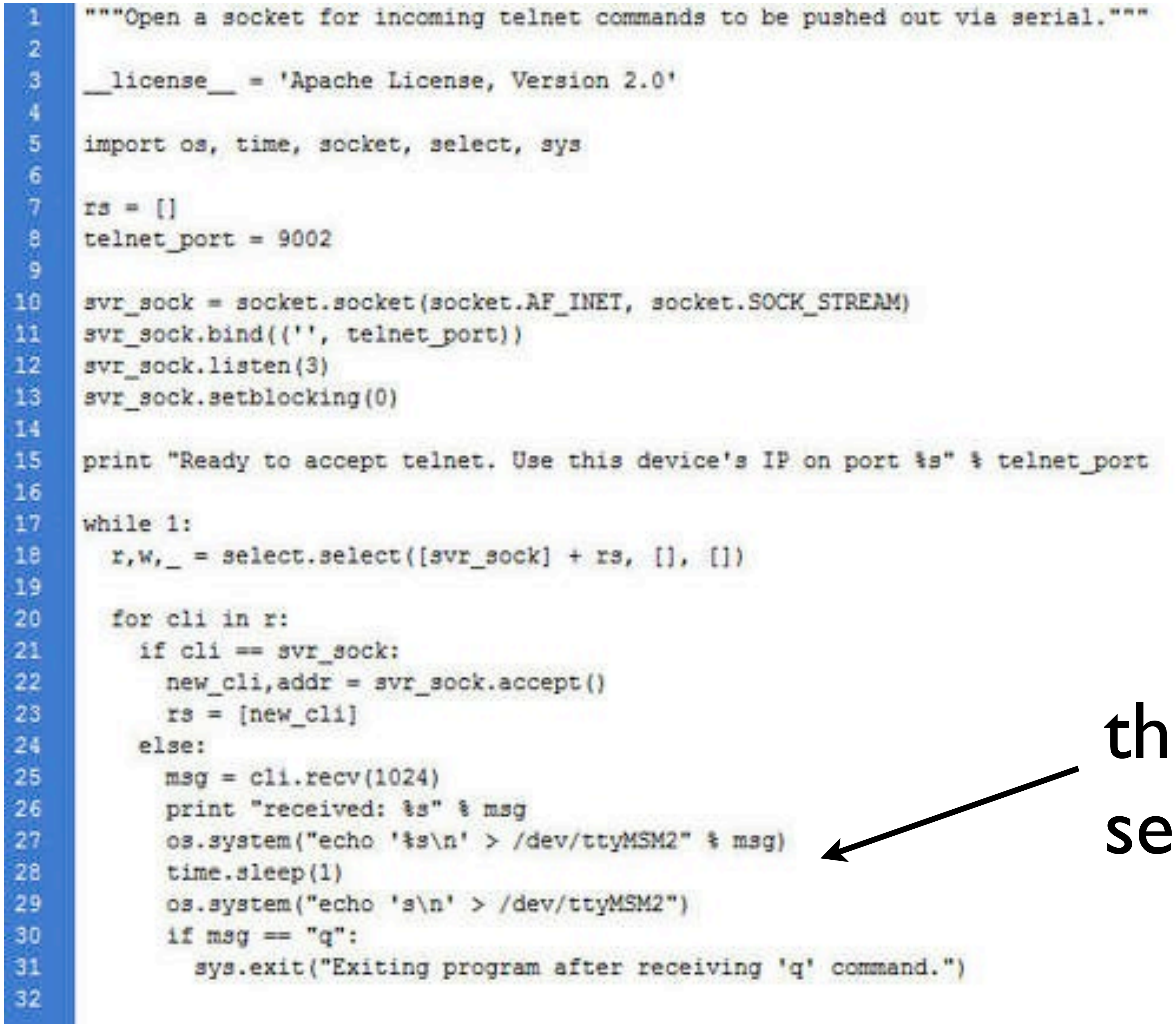


f,r,l,b,s,fr

procedure:

1. send command to phone
2. phone sends to uC via serial
3. uC decodes command
4. starts/stops appropriate wheel(s)

send to phone



```python
"""Open a socket for incoming telnet commands to be pushed out via serial."""

__license__ = 'Apache License, Version 2.0'

import os, time, socket, select, sys

rs = []
telnet_port = 9002

svr_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
svr_sock.bind(('', telnet_port))
svr_sock.listen(3)
svr_sock.setblocking(0)

print "Ready to accept telnet. Use this device's IP on port %s" % telnet_port

while 1:
  r,w,_ = select.select([svr_sock] + rs, [], [])

  for cli in r:
    if cli == svr_sock:
      new_cli,addr = svr_sock.accept()
      rs = [new_cli]
    else:
      msg = cli.recv(1024)
      print "received: %s" % msg
      os.system("echo '%s\n' > /dev/ttyMSM2" % msg)
      time.sleep(1)
      os.system("echo 's\n' > /dev/ttyMSM2")
      if msg == "q":
        sys.exit("Exiting program after receiving 'q' command.")
```

this sends commands via serial port

on uC: 1. read serial
2. parse command
3. perform command

```
void loop()
{
  readSerialInput();
  checkIfStopBot();
}
```

```
// Reads serial input if available and parses command when full command has been sent.
void readSerialInput() {
  while(Serial.available() && serialIndex < BUFFERSIZE) {
    //Store into buffer.
    inBytes[serialIndex] = Serial.read();

    //Check for command end.
    if (inBytes[serialIndex] == '\n' || inBytes[serialIndex] == ';' || inBytes[serialIndex] == '>') { //Use ; when using Serial Monitor
       inBytes[serialIndex] = '\0'; //end of string char
       parseCommand(inBytes);
       serialIndex = 0;
    }
    else{
      serialIndex++;
    }
  }

  if(serialIndex >= BUFFERSIZE){
    //buffer overflow, reset the buffer and do nothing
    //TODO: perhaps some sort of feedback to the user?
    for(int j=0; j < BUFFERSIZE; j++){
      inBytes[j] = 0;
      serialIndex = 0;
    }
  }
}
```

```
// Cleans and parses the command
void parseCommand(char* com) {
  if (com[0] == '\0') { return; } //bit of error checking
  int start = 0;
  //get start of command
  while (com[start] != '<'){
    start++;
    if (com[start] == '\0') {
      //its not there. Must be old version
      start = -1;
      break;
    }
  }
  start++;
  performCommand(com);
}
```

commands: f,r,l,b,s,fr

```
void performCommand(char* com) {
  if (strcmp(com, "f") == 0) { // Forward
    stopTime = driveWheels(speedMultiplier * 10, speedMultiplier * 10);
    servosActive = true;
  } else if (strcmp(com, "r") == 0) { // Right
    stopTime = driveWheels(speedMultiplier * 10, speedMultiplier * -10);
    servosActive = true;
  } else if (strcmp(com, "l") == 0) { // Left
    stopTime = driveWheels(speedMultiplier * -10, speedMultiplier * 10);
    servosActive = true;
  } else if (strcmp(com, "b") == 0) { // Backward
    stopTime = driveWheels(speedMultiplier * -10, speedMultiplier * -10);
    servosActive = true;
  } else if (strcmp(com, "s") == 0) { // Stop
    stopBot();
    servosActive = false;
  } else if (strcmp(com, "fr") == 0 || strcmp(com, "fz") == 0 || strcmp(com, "x") == 0) { // Read
distance sensor
    dist = getDistanceSensor(rangePinForward);
    itoa(dist, msg, 10); // Turn the dist int into a char
    serialReply("x", msg); // Send the distance out the serial line
  } else if (strcmp(com, "z") == 0) { // Read and print ground facing distance sensor
    dist = getDistanceSensor(rangePinForwardGround);
    itoa(dist, msg, 10); // Turn the dist int into a char
    serialReply("z", msg); // Send the distance out the serial line
  } else if (strcmp(com, "h") == 0) { // Help mode - debugging toggle
    // Print out some basic instructions when first turning on debugging
    if (not DEBUGGING) {
      Serial.println("Ready to listen to commands! Try ome of these:");
      Serial.println("F (forward), B (backward), L (left), R (right), S (stop), D (demo).");
      Serial.println("Also use numbers 1-9 to adjust speed (0=slow, 9=fast).");
    }
```

Assembly VIII

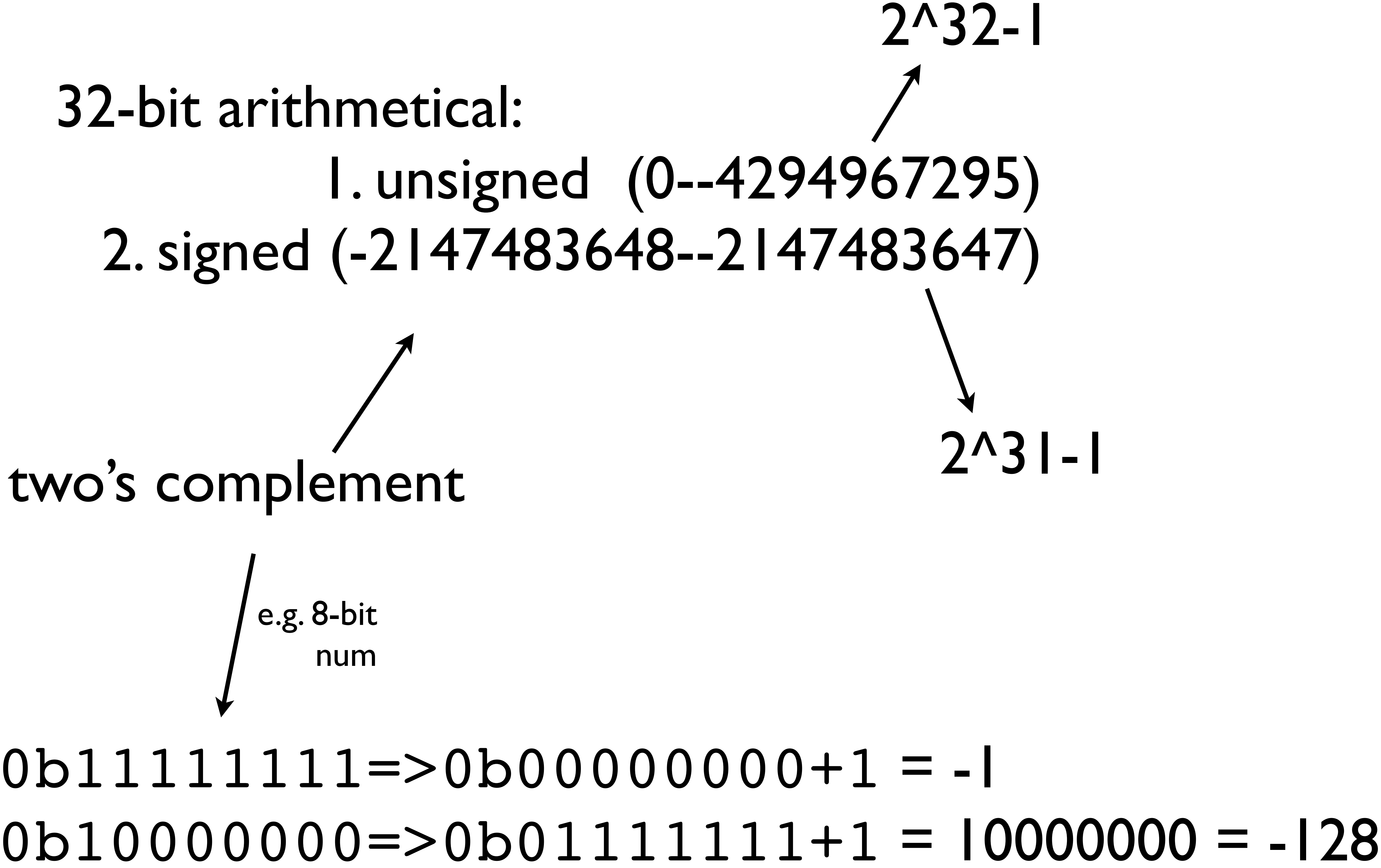
ECE 3710

I intend to live forever,
or die trying.

- Groucho Marx

instructions:

1. shifts
2. arithmetical



2^32-1

32-bit arithmetical:

1. unsigned (0--4294967295)

2. signed (-2147483648--2147483647)

2^31-1

two’s complement

e.g. 8-bit num

0b11111111=>0b00000000+1 = -1

0b10000000=>0b01111111+1 = 10000000 = -128

(+) → unsigned instructions

```
add{s} R0,#0x3d       ;R0 = R0+0x3d
adc{s} R0,R1,#0x3d    ;R0 = R1+0x3d+C
sub{s} R0,R1,R2       ;R0 = R1-R2
sbc{s} R0,#0x3d       ;R0 = R0-0x3d-(~C)
mul{s} R0,R1,R2       ;R0=R1*R2 (first 32-bits)
umull{s} R0,R1,R2,R3 ;[R1(32--63) R0(0--31)]
                      ; = R2*R3 (64-bits)
udiv R0,R1            ;R0 = R0/R1
```

syntax:

1. + & -: `<op> {Rd,} Rn, <op2>`
2. * & /: `<op> {Rd,} Rm, Rs`
3. *-long: `umull{s} RdLo, RdHi, Rm, Rs`

arith_unsigned

(+) →

unsigned instructions

```
add{s} R0,#0x3d       ;R0 = R0+0x3d
adc{s} R0,R1,#0x3d ;R0 = R1+0x3d+C
sub{s} R0,R1,R2       ;R0 = R1-R2
sbc{s} R0,#0x3d       ;R0 = R0-0x3d-(~C)
mul{s} R0,R1,R2       ;R0=R1*R2 (first 32-bits)
umull{s} R0,R1,R2,R3 ;[R1(32--63) R0(0--31)]
                      ; = R2*R3 (64-bits)
udiv R0,R1            ;R0 = R0/R1
```

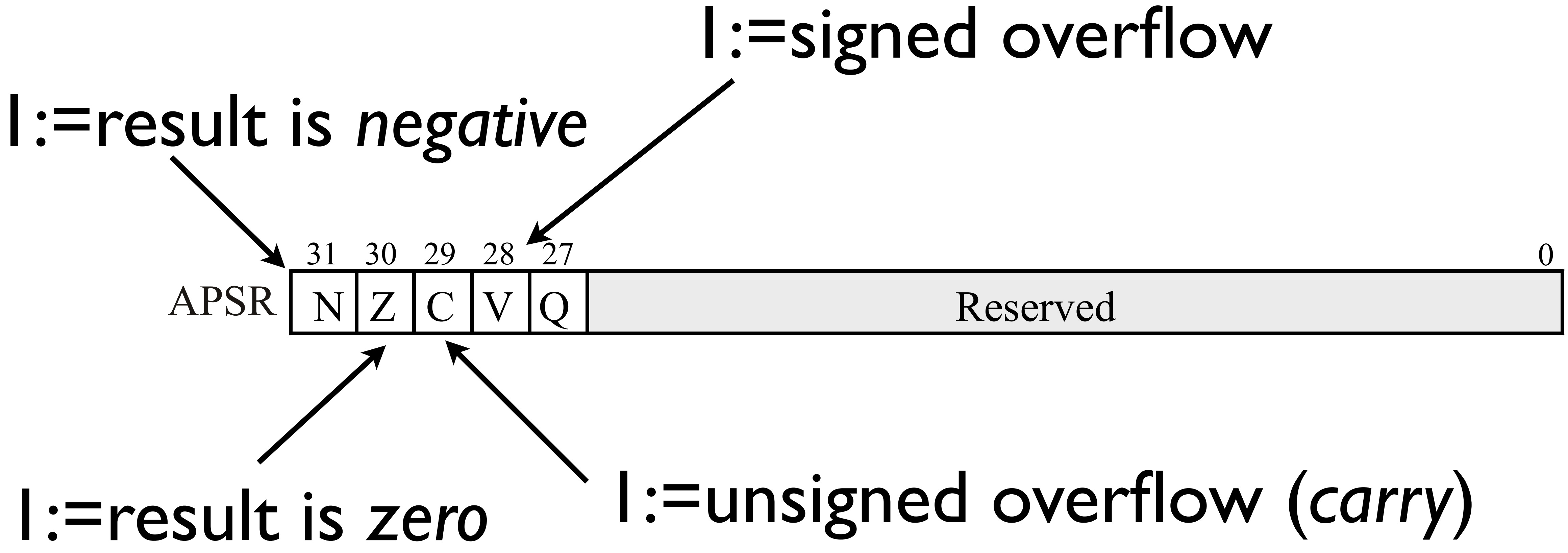
Q:

1. why use C?

2. what if something goes wrong?

APSR

last operation:



1:=signed overflow

1:=result is *negative*

1:=result is *zero*

1:=unsigned overflow (*carry*)

‘condition codes’ ⟶ these are context dependence

how possible errors are noted: NZCV

like :

don't do this:
1. set dest. to zero
2. halt uC

... NZCV ← instructions must alter these

(append s)

adc: add num > 32-bit

```
   0x75d451baa6d30ad3
 + 0x72689958c14f48ff:  00000001 01000000
                        7514517a a6d30ad3
                      + 72689952 c145f41a
                       =E77CEACD 6818FEED
```

add word-by-word:

```
ldr R0,=#0xa6d30ad3  ;lower first
ldr R1,=#0xc145f41a
adds R3,R0,R1        ;lower to r3
ldr R0,=#0x7514517a  ;upper next
ldr R1,=#0x72689952
adc R4,R0,R1        ;use carry
                     ;upper in r4
```

must tell uC to note carry

```
result = [R4 R3]
```

add64

note:

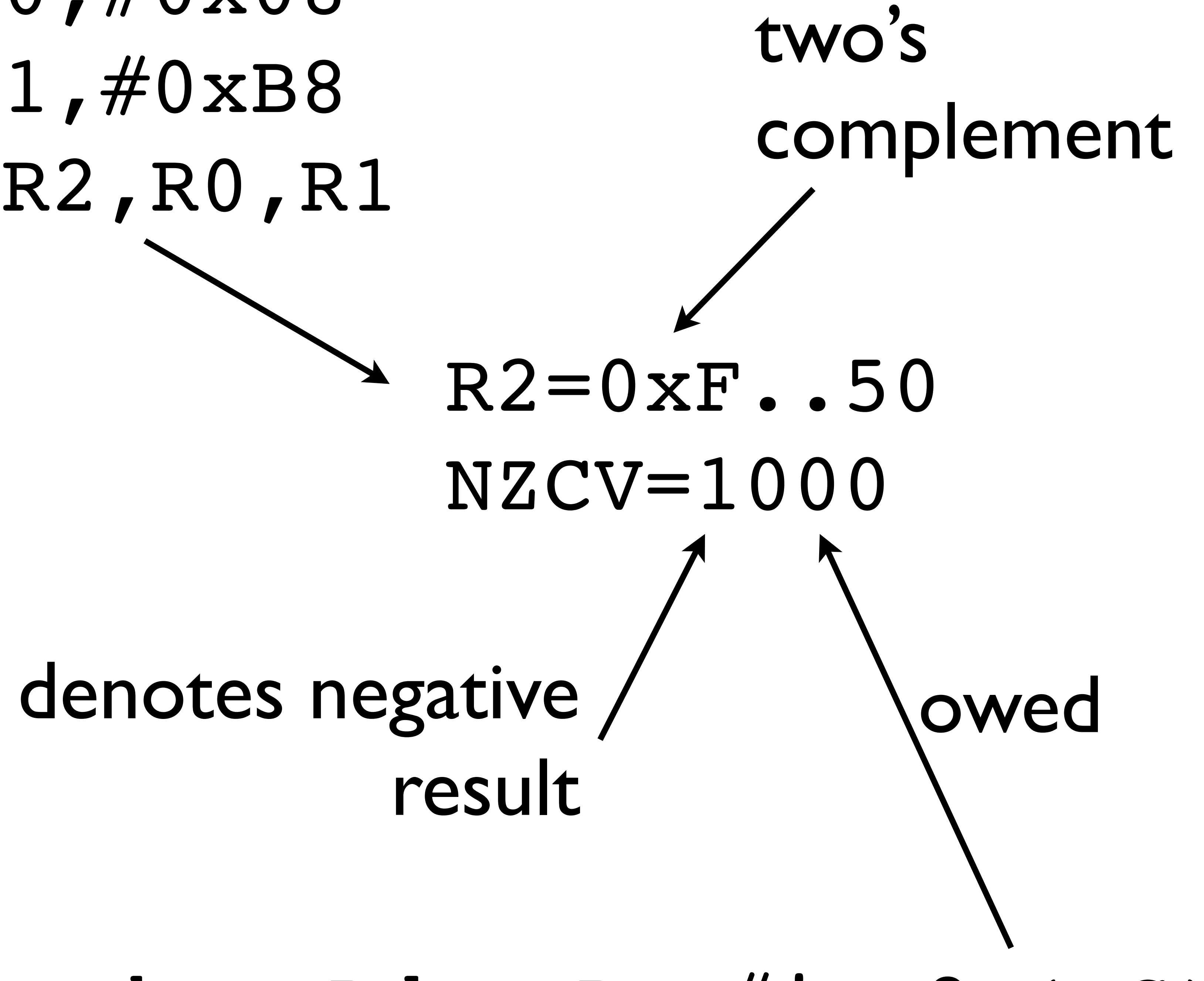
```
    adc R4,R0,R1          ;use carry
```

does not clear C flag

(op{s} or cmp will overwrite, though)

sub resulting in negative

```
   08    mov R0,#0x08
  -B8    mov R1,#0xB8
=-B0     subs R2,R0,R1
```



R2=0xF..50

NZCV=1000

two's complement

denotes negative result

owed

`sbc: Rd = Rn-#imm8-(~C)`

R2=0x50 when N=1 results from `sub`

```
0xF...50:
 ~1...01010000
 =0...10101111
            +1
 =0...10110000
      =(-)0xB0
```

→

```
mov R0,#0x08
mov R1,#0xB8
subs R2,R0,R1
mvn R2,R2
add R2,#1
```

from N=1

make the uC do it

C is useful for:

1. multi-byte addition/subtraction
2. denoting neg. results after subtraction

(C = 0)

signed numbers:

1. `add,sub,mul,smull,sdiv`
2. two’s complement

assembler allows
this:

```
mov R0,#-5
add R0,#-2
```

←

puts numbers
in two’s complement
(lucky you)

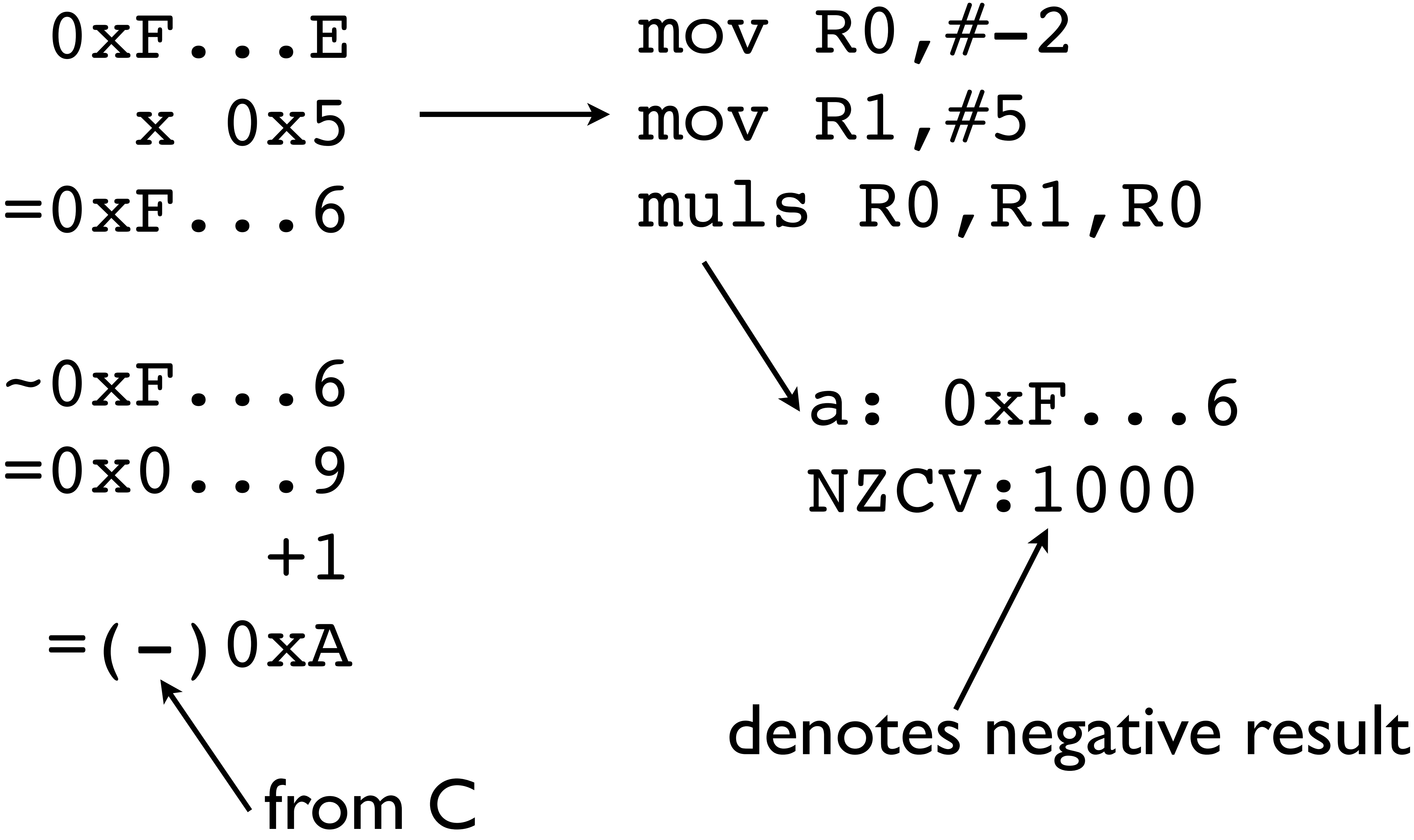
-2+-5

```
mov R0,#0xfe
add R0,#0xfb
```

```
mov R0,#2 ;get -2
mvn R0
add R0,#1
mov R1,#5  ;get -5
mvn R1
add R1,#1
add R2,R0,R1
mvn R2        ;just check mag
add R2,#1
```

just in case in you need to convert

`mul`: + times -



```
 0xF...E
   x 0x5
=0xF...6

~0xF...6
=0x0...9
      +1
 =(-)0xA
```

from C

```
mov R0,#-2
mov R1,#5
muls R0,R1,R0
```

```
a: 0xF...6
NZCV:1000
```

denotes negative result

div vs. sdiv

(do not set NZCV)

udiv: -15/5

```
mov R0,#-15
mov R1,#5
udiv R2,R0,R1
```

```
R2= 0x3..0
```

-15=0xF...1= 4294967281

/5

= 858993456

=0x3...0

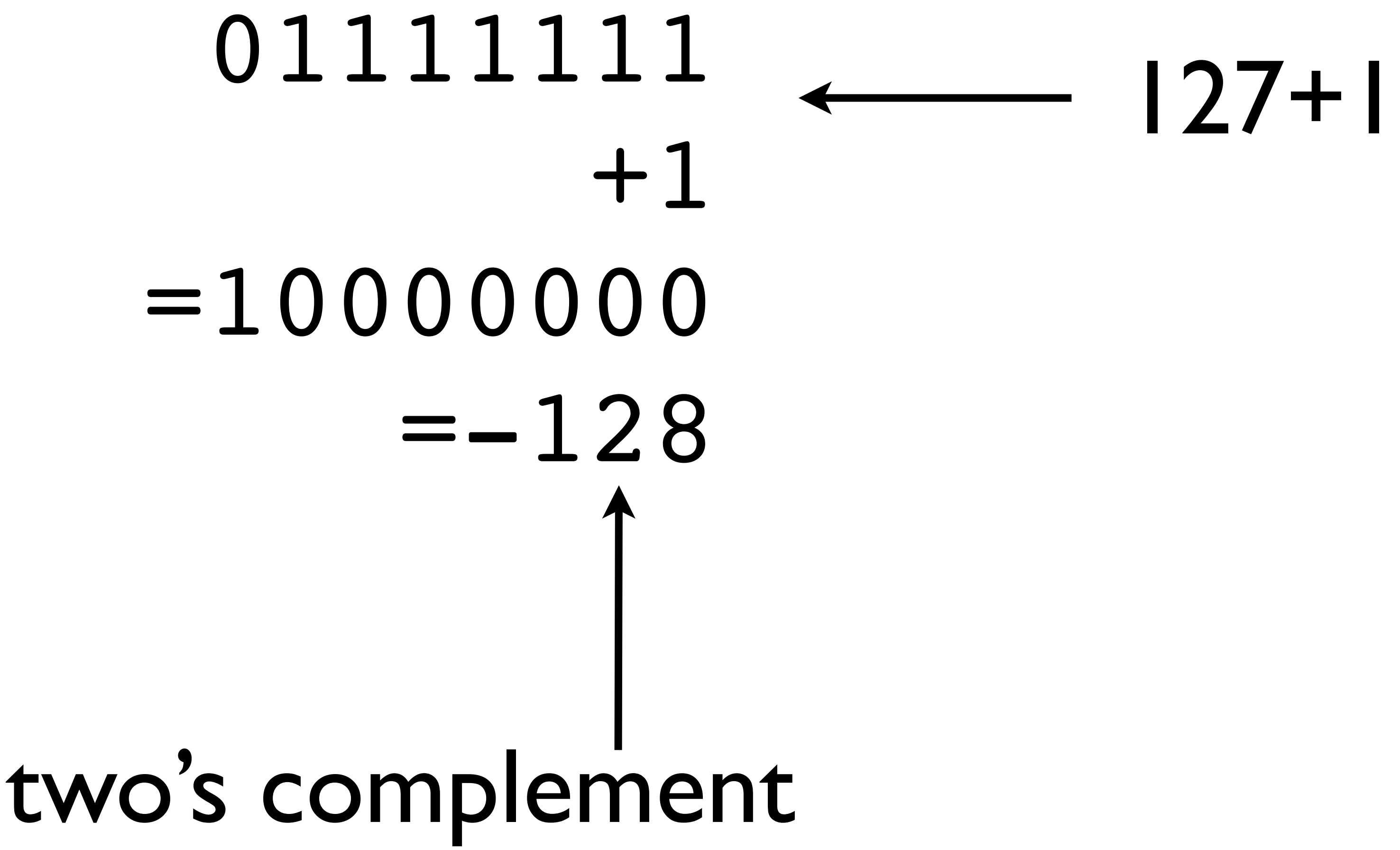
sdiv: -15/5

```
mov R0,#-15
mov R1,#5
sdiv R2,R0,R1
```

```
R2=0xF...D
```

beware the overflow: 127+1 = -128?

(8-bit example)



```
 01111111
       +1
=10000000
     =-128
```

127+1

two’s complement

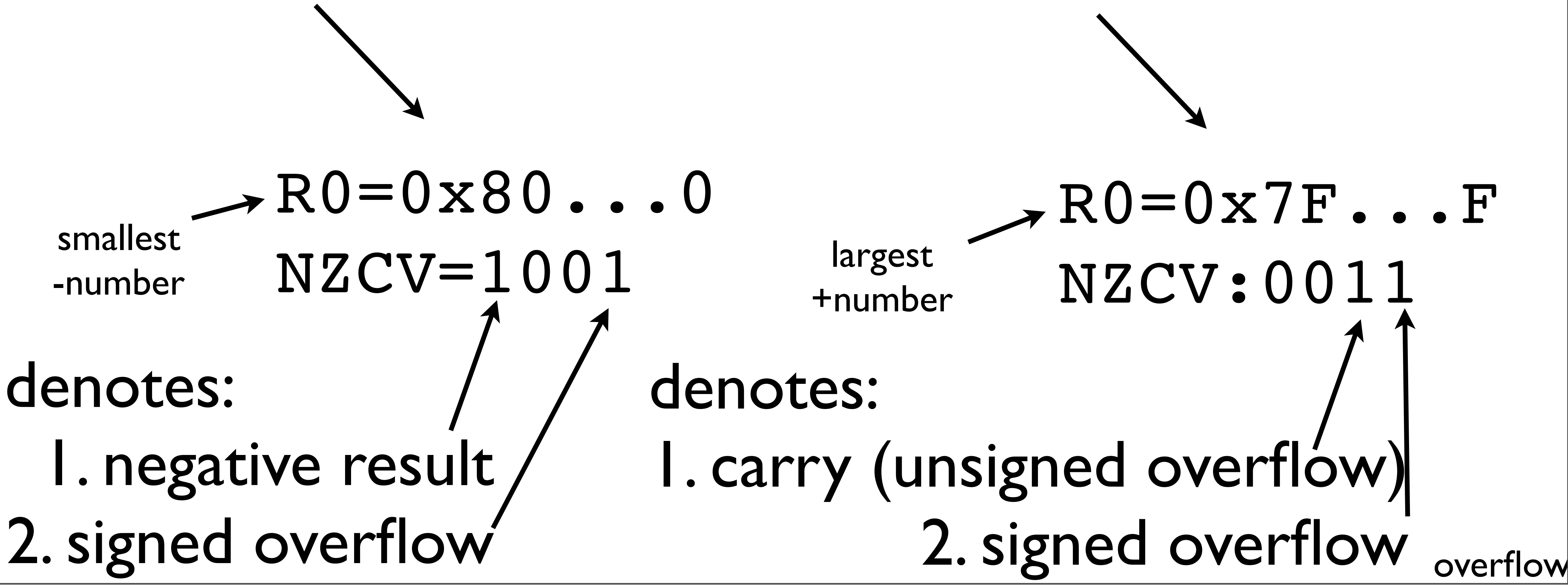
+ to - and - to +

the uC will let you know:

(so long as you tell it to and check)

```
;(largest +number)
mov R0,#0x7F...F
adds R0,#1
```

```
;(largest -number)
mov R0,#0xF...F
adds R0,#0x80...0
```



```
R0=0x80...0
NZCV=1001
```

smallest -number

denotes:

1. negative result
2. signed overflow

```
R0=0x7F...F
NZCV:0011
```

largest +number

denotes:

1. carry (unsigned overflow)
2. signed overflow

overflow

remember:

uC doesn’t tell you if you’ve exceeded register limits for multiplication

`muls Rd,Rn,Rd` → NZ

`(Rd=Rn*Rd)`

sets only

must have this form if {s} used

because we’re engineers:

the appropriate response
to x86 asm...

...that's your head at the end