CSC 230 - Summer 2018

Branching and SREG

Bill Bird

Department of Computer Science
University of Victoria

June 11, 2018

Counting to Ten (1)

C

```c
unsigned int i = 0;
unsigned int sum = 0;
unsigned int max = 10;
while(i <= max){
  sum += i;
  i++;
}
//Done
```

Consider the C code snippet above, which uses a loop to compute the sum $1 + 2 + \ldots +$ `max`.

Notice that in C (and most other languages), the loop will continue as long as the condition '`i <= max`' is true.

Counting to Ten (2)

C

```c
unsigned int i = 0;
unsigned int sum = 0;
unsigned int max = 10;
while(1){
  if (max < i)
    break;
  sum += i;
  i++;
}
//Done
```

Now consider the alternative version above, which uses a notionally infinite loop, but with a conditional `break` statement to end the loop when the converse condition '`max < i`' is true.

(Note that in C, the value 0 is considered to be `false`, and all non-zero values are considered to be `true`, so the statement '`while (1)`' is equivalent to '`while (true)`' in Java)

Counting to Ten (3)

C

```
unsigned int i = 0;
unsigned int sum = 0;
unsigned int max = 10;
LOOP:
if (max < i)
  goto END;
sum += i;
i++;
goto LOOP;
END:
//Done
```

Finally, consider the version above, which discards the structured `while` statement in favour of unstructured `GOTO` statements[a].

This is a better reflection of the machine code that would be generated by a C compiler when a `while` or `for` loop is compiled.

[a] Advice: Never use `GOTO` statements in actual C code.

Counting to Ten (4)

C

```
unsigned int i = 0;
unsigned int sum = 0;
unsigned int max = 10;
LOOP:
if (max < i)
  goto END;
sum += i;
i++;
goto LOOP;
END:
//Done
```

At the beginning of each iteration of the loop, the loop condition is checked, and if it is `false` (that is, if the converse condition is true), the loop ends by jumping to the first statement after the loop body.

Counting to Ten (5)

C

```c
unsigned int i = 0;
unsigned int sum = 0;
unsigned int max = 10;
LOOP:
if (max < i)
  goto END;
sum += i;
i++;
goto LOOP;
END:
//Done
```

Otherwise, the loop body executes and, at the end of the loop body, control jumps back to the top of the loop to test the condition again.

Counting to Ten (6)

AVR Assembly

```
clr r0
clr r1
ldi r16, 10
loop:
  cp r16, r0
  brlo end
  add r1, r0
  inc r0
  rjmp loop
end:
rjmp end
```

The same pattern can be used to write loops and other control flow (e.g. if-statements) directly in AVR assembly. In the program above, `r0`, `r1` and `r16` are used for `i`, `sum` and `max`, respectively.

Counting to Ten (7)

AVR Assembly

```
clr r0
clr r1
ldi r16, 10
loop:
  cp r16, r0
  brlo end
  add r1, r0
  inc r0
  rjmp loop
end:
rjmp end
```

The JMP, RJMP and BRLO instructions are all **branch instructions**. The JMP and RJMP instructions are **unconditional branches** (which always jump to the specified location)

Counting to Ten (8)

AVR Assembly

```
clr r0
clr r1
ldi r16, 10
loop:
  cp r16, r0
  brlo end
  add r1, r0
  inc r0
  rjmp loop
end:
rjmp end
```

The BRLO instruction (and the rest of the BR family) is a **conditional branch**. A conditional branch will either jump to the specified location (if a condition is met) or do nothing (and proceed to the next instruction).

Counting to Ten (9)

AVR Assembly

```
clr r0
clr r1
ldi r16, 10
loop:
  cp r16, r0
  brlo end
  add r1, r0
  inc r0
  rjmp loop
end:
rjmp end
```

The various conditional branch instructions are defined based on conditions involving the flags in SREG (including the C, N, Z and V flags above). Therefore, conditional branch instructions are generally used in close proximity to instructions that set flags.

Counting to Ten (10)

AVR Assembly

```
clr r0
clr r1
ldi r16, 10
loop:
  cp r16, r0
  brlo end
  add r1, r0
  inc r0
  rjmp loop
end:
rjmp end
```

The CP (Compare) instruction takes two operand registers and sets the flags in SREG based on their difference. We will cover the exact mechanics of this soon, but we should first discuss the intuitive meaning of comparisons when used with branch instructions.

Counting to Ten (11)

AVR Assembly

```
clr r0
clr r1
ldi r16, 10
loop:
  cp r16, r0
  brlo end
  add r1, r0
  inc r0
  rjmp loop
end:
rjmp end
```

After an instruction of the form '`cp A, B`' (for two registers `A` and `B`), the `BRLO` ('Branch if lower') instruction will follow the branch if `A` $<$ `B`, assuming that both `A` and `B` contain unsigned values.

Counting to Ten (12)

AVR Assembly

```
clr r0
clr r1
ldi r16, 10
loop:
  cp r16, r0
  brlo end
  add r1, r0
  inc r0
  rjmp loop
end:
rjmp end
```

To branch on the condition `A` $<$ `B` when `A` and `B` are signed, the `BRLT` ('Branch if less than') instruction can be used instead.

Counting to Ten (13)

AVR Assembly

```
clr r0
clr r1
ldi r16, 10
loop:
  cp r16, r0
  brlo end
  add r1, r0
  inc r0
  rjmp loop
end:
rjmp end
```

Question: Why is it necessary to use separate logic to test the condition `A` $<$ `B` for signed and unsigned values?

Counting to Ten (14)

AVR Assembly

```
clr r0
clr r1
ldi r16, 10
loop:
  cp r16, r0
  brlo end
  add r1, r0
  inc r0
  rjmp loop
end:
rjmp end
```

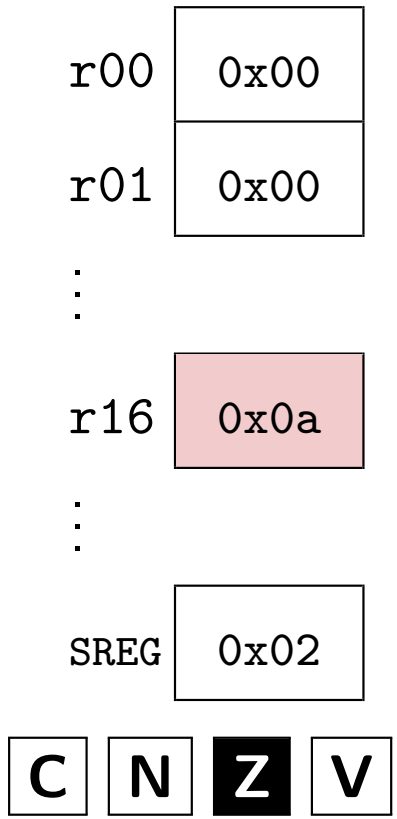
Consider the values A = 0x06 and B = 0xe6. As unsigned values, $A = (6)_{10}$ and $B = (230)_{10}$, so A < B. As two's complement signed values, $A = (6)_{10}$ and $B = (-26)_{10}$, so A > B.

Counting to Ten (15)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

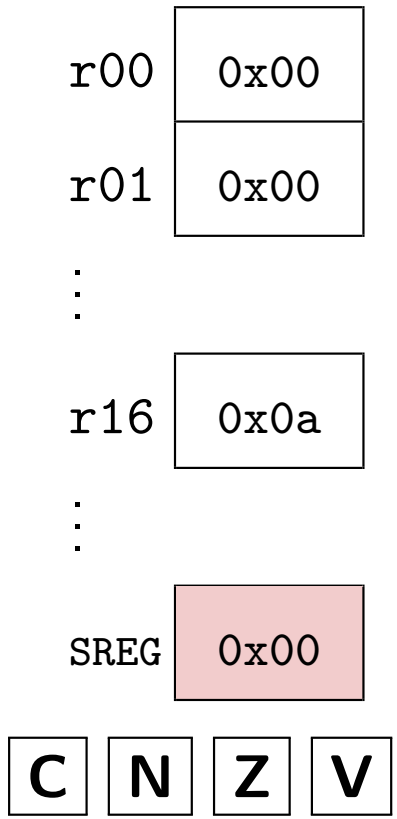
When the code reaches the CP instruction for the first time, r0 equals 0x00.

Counting to Ten (16)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

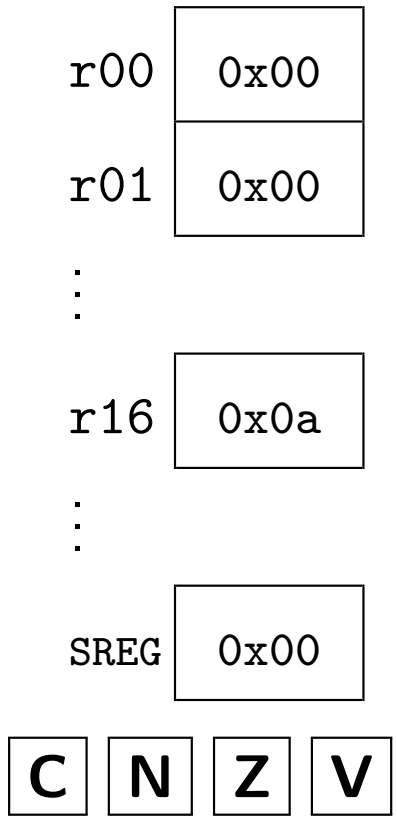
The CP instruction computes the value r16 − r0 = 0xa and sets the flags of SREG accordingly (in this case, no flags are set at all).

Counting to Ten (17)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

The body of the loop then executes.

Counting to Ten (18)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers

r00	0x00
r01	0x00
⋮	
r16	0x0a
⋮	
SREG	0x02

C N **Z** V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

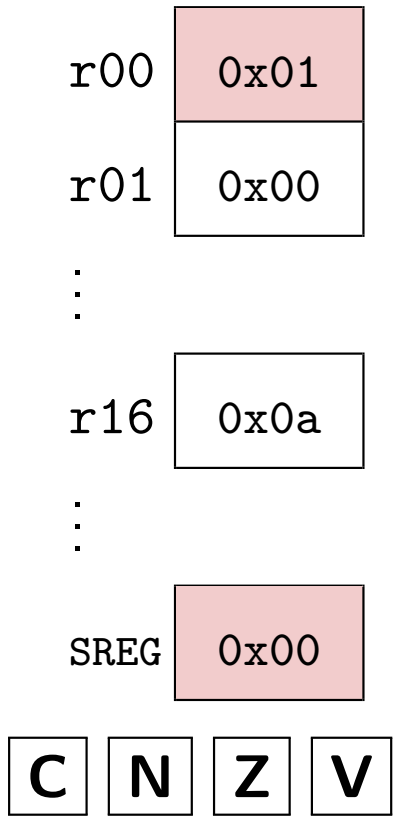
The body of the loop then executes.

Counting to Ten (19)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

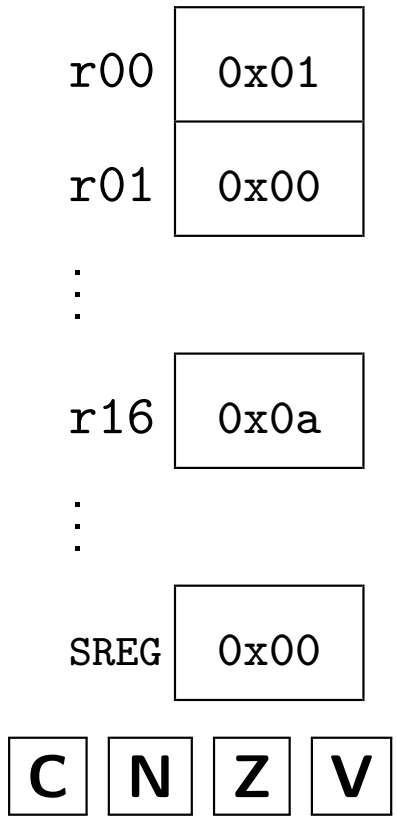
The body of the loop then executes.

Counting to Ten (20)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

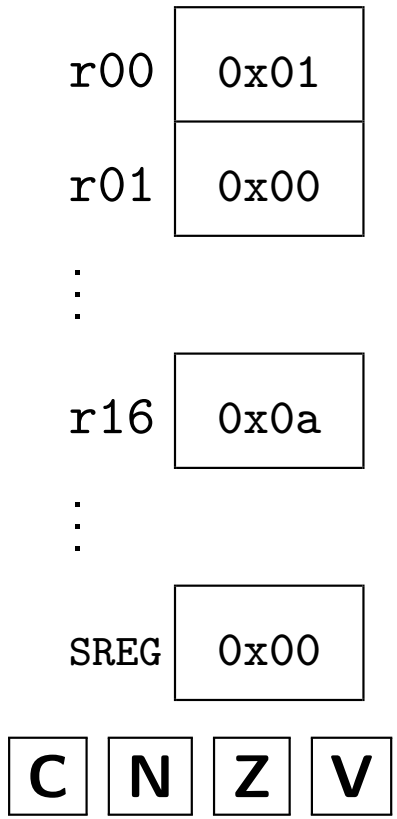
The body of the loop then executes.

Counting to Ten (21)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

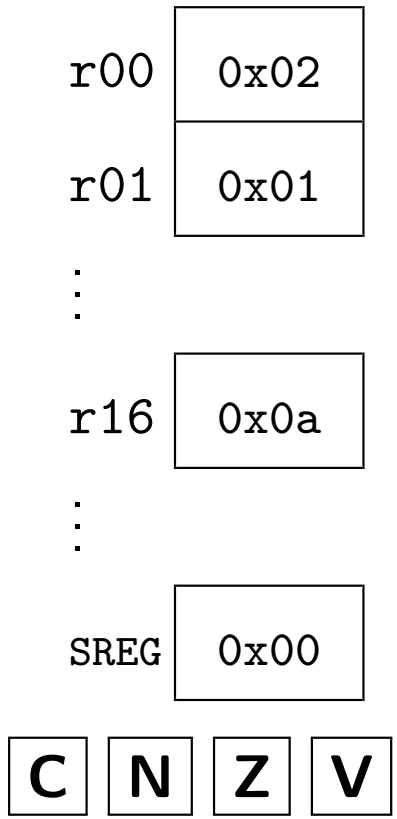
When the BRLO instruction is reached for the second time, the value of r0 is 0x01, so the branch will still not be followed.

Counting to Ten (22)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

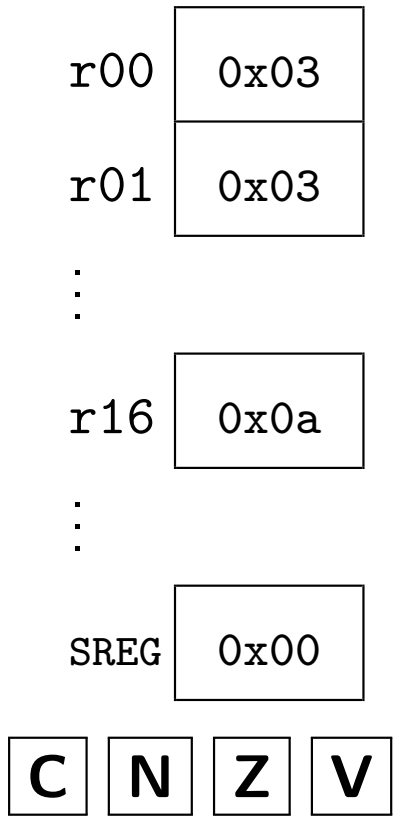
This continues as r0 iterates through 0x02 to 0x09.

Counting to Ten (23)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

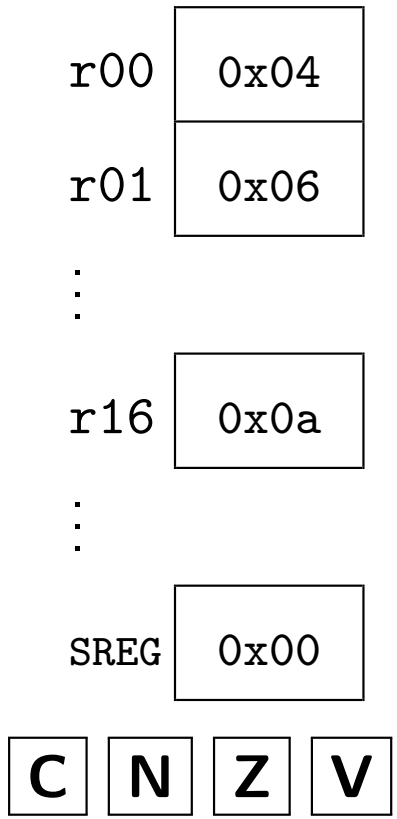
This continues as r0 iterates through 0x02 to 0x09.

Counting to Ten (24)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

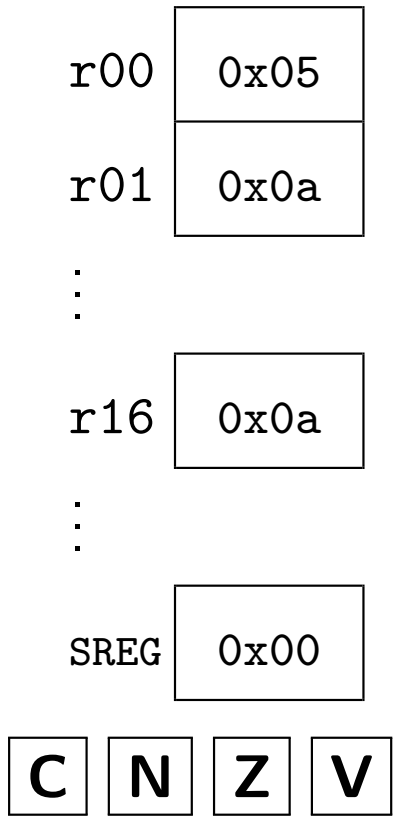
This continues as r0 iterates through 0x02 to 0x09.

Counting to Ten (25)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

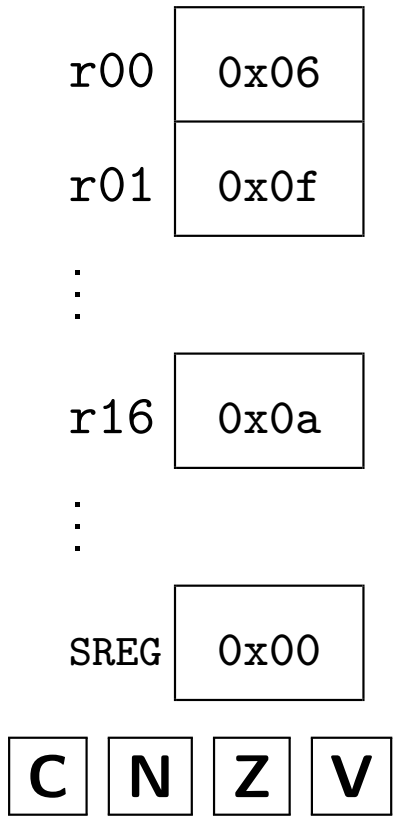
This continues as r0 iterates through 0x02 to 0x09.

Counting to Ten (26)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

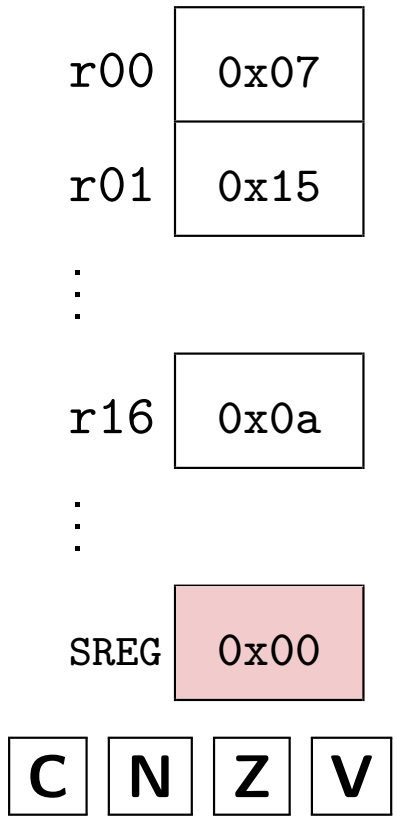
This continues as r0 iterates through 0x02 to 0x09.

Counting to Ten (27)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

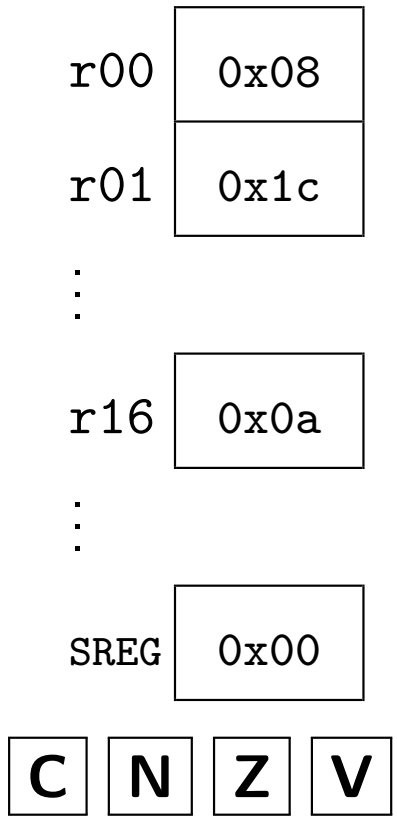
This continues as r0 iterates through 0x02 to 0x09.

Counting to Ten (28)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

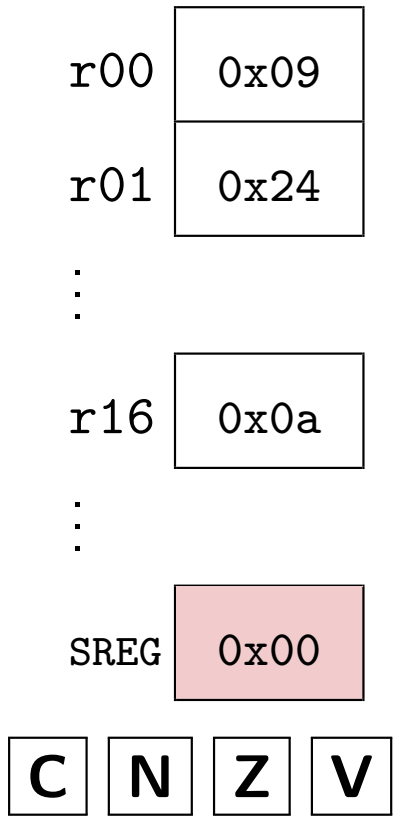
This continues as r0 iterates through 0x02 to 0x09.

Counting to Ten (29)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

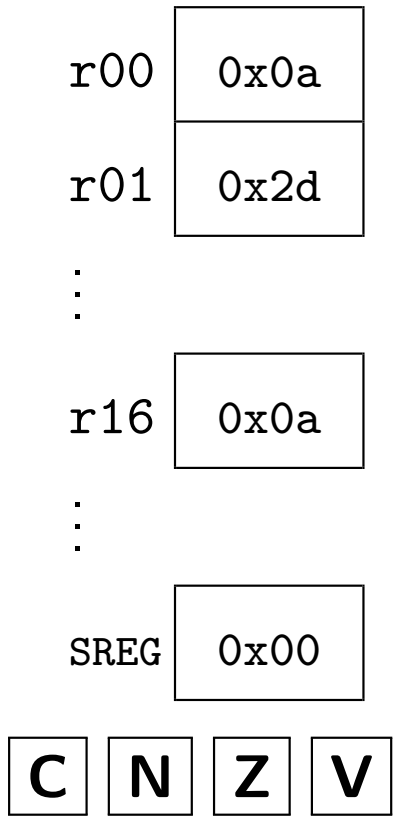
This continues as r0 iterates through 0x02 to 0x09.

Counting to Ten (30)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

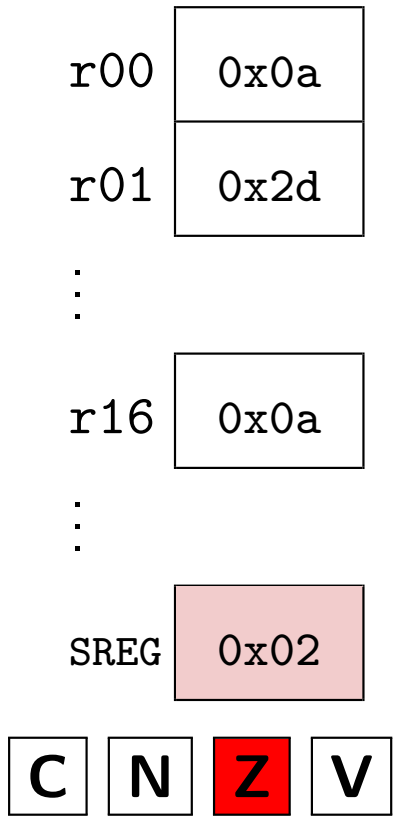
At the beginning of the last iteration, `r0` equals `0x0a`.

Counting to Ten (31)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

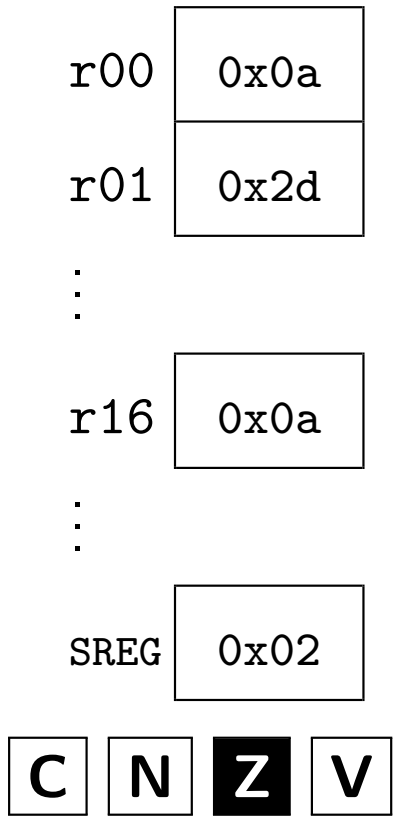
Notice that the CP instruction sets the Z flag, since r16 and r0 are now equal. However, r16 is still not less than r0, so the BRLO still does not branch.

Counting to Ten (32)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

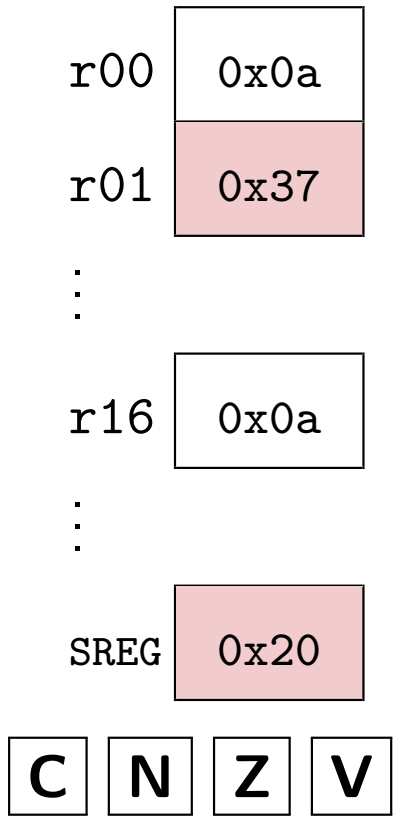
The last iteration of the loop body then executes, resulting in `r0` equalling `0xb`.

Counting to Ten (33)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

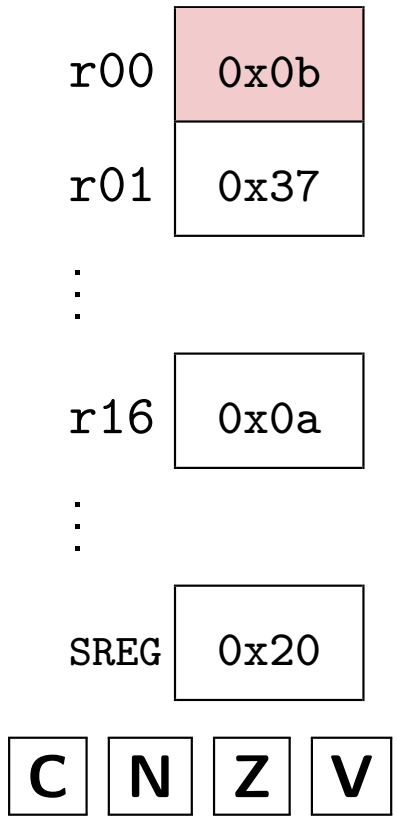
The last iteration of the loop body then executes, resulting in `r0` equalling `0xb`.

Counting to Ten (34)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

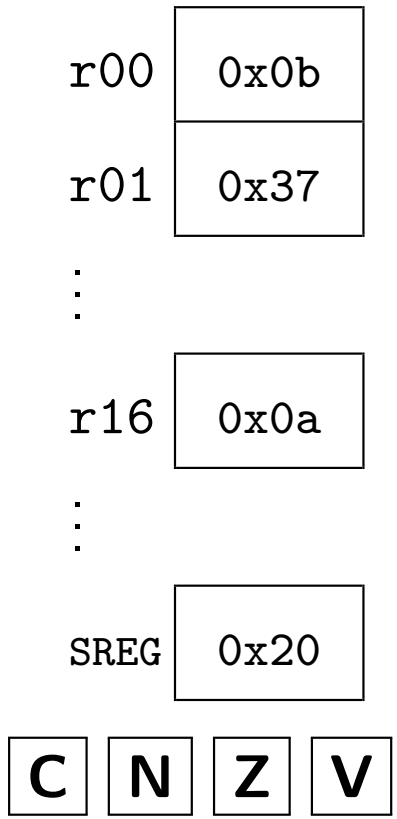
The last iteration of the loop body then executes, resulting in `r0` equalling `0xb`.

Counting to Ten (35)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

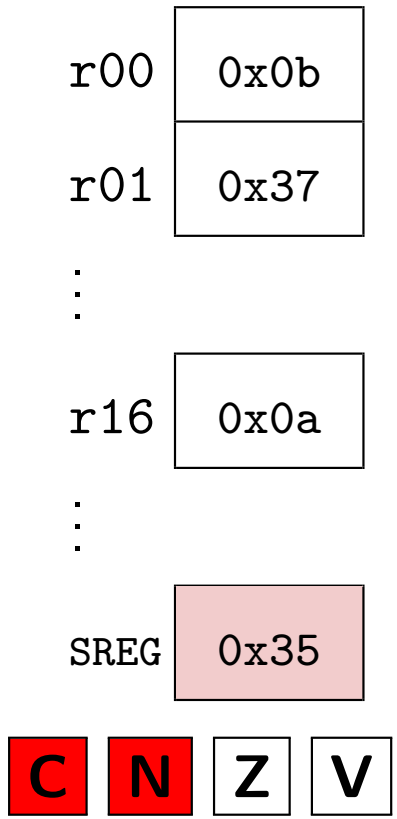
Now the CP instruction computes the value r16 − r0, which results in an unsigned overflow (setting the carry flag), since 0x0a is less than 0x0b.

Counting to Ten (36)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers



Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

When the branch instruction is executed, it will now follow the branch. As it happens, the criteria for BRLO to branch is $C = 1$, since the operation A − B will set the carry bit whenever A is smaller than B (using unsigned arithmetic.

Counting to Ten (37)

Program Memory

```
(0x0000)    clr   r0
(0x0001)    clr   r1
(0x0002)    ldi   r16, 10
        loop:
(0x0003)    cp    r16, r0
(0x0004)    brlo  end
(0x0005)    add   r1, r0
(0x0006)    inc   r0
(0x0007)    rjmp  loop
        end:
(0x0008)    rjmp  end
```

Registers

r00	0x0b
r01	0x37
⋮	
r16	0x0a
⋮	
SREG	0x35

C **N** Z V

Data Memory

Address	Contents
0x0200	0x00
0x0201	0x00
0x0202	0x00
0x0203	0x00
0x0204	0x00
0x0205	0x00

Control then jumps to the `end` label. Notice that the sum in `r1` is 0x37, which is $(55)_{10} = 1 + 2 + \ldots + 10$.

Branch Instructions

After a compare (`CP`, `CPC` or `CPI`) operation of the form `CP A, B`, the basic numerical relationships can be implemented with the branch instructions below.

Instruction	Relationship
`BRLO`	`A` $<$ `B` (unsigned)
`BRLT`	`A` $<$ `B` (signed)
`BREQ`	`A` $=$ `B`
`BRNE`	`A` $\neq$ `B`
`BRSH`	`A` $\geq$ `B` (unsigned)
`BRGE`	`A` $\geq$ `B` (signed)

Question: How can we implement comparisons like $A \leq B$ or $A > B$ using the operations above?

Status Register Flags (1)

		1	1	1	0	0	1	1	0	0xe6
−		1	1	1	0	0	0	0	1	0xe1
	0	0	0	0	0	0	1	0	1	0x05

Carry

Flags: **C** **N** **Z** **V** **H**

The `CP` family of instructions sets six of the flags in SREG based on the result of a subtraction operation. Other instructions (such as `ADD` and `SUB`) also set the flags; in fact, the flags set by a `CP` instruction are identical to those set by the corresponding `SUB` instruction (although the result of the subtraction is discarded).

Status Register Flags (2)

		1	1	1	0	0	1	1	0	0xe6
−		1	1	1	0	0	0	0	1	0xe1
	0	0	0	0	0	0	1	0	1	0x05

Carry

Flags: **C** **N** **Z** **V** **H**

All of the BR instructions act only on the values of the flags in the SREG status register. There is no requirement that a CP operation be used before a conditional branch (since the flags can be set by numerous other instructions).

Status Register Flags (3)

SREG is an 8-bit register comprising 8 flag bits (and since the flags are independent of each other, SREG is not normally viewed as a single numerical value, but as 8 independent values sharing a byte of memory).

The flags are

- Bit 0: **Carry (C)**
- Bit 1: **Zero (Z)**
- Bit 2: **Negative (N)**
- Bit 3: **Two's Complement Overflow (V)**
- Bit 4: **Signed Test Flag (S)**
- Bit 5: **Half-byte Carry (H)**
- Bit 6: **Transfer Bit (T)**
- Bit 7: **Interrupt Enable (I)**

Status Register Flags (4)

There are conditional branch instructions defined which will branch/not branch based on the value of any single flag. For example, `BRHC` and `BRHS` will branch if the H bit is clear (`BRHC`) or set (`BRHS`).

For most numerical comparisons, the only relevant bits are C, Z, N and V. The H bit has a specialized use, and the S bit is generally set based on the values of N and V. The T and I bits are not set by arithmetic instructions.

Status Register Flags (5)

		1	1	1	1	1	1	1	0	0xfe
+		0	0	0	0	0	0	1	1	0x03
	1	0	0	0	0	0	0	0	1	0x01
	Carry									

Flags: **C** **N** **Z** **V** **H**

The **Carry (C)** flag is set whenever an arithmetic operation produces a carry from the highest bit. It is also set by shift instructions to contain the bit that is shifted off the end of the byte by the shift.

Status Register Flags (6)

		1	1	1	1	1	1	1	0	0xfe
+		0	0	0	0	0	0	1	1	0x03
	1	0	0	0	0	0	0	0	1	0x01

Carry

Flags: **C** **N** **Z** **V** **H**

In the addition above, the unsigned calculation `0xfe` + `0x03` = $(254)_{10} + (3)_{10}$ overflows (producing the result `0x01`), so the carry bit is set.

Status Register Flags (7)

		0	0	0	0	0	0	0	1	0x01
–		0	0	0	0	0	0	1	1	0x03
	1	1	1	1	1	1	1	1	0	0xfe

Carry

Flags: **C** **N** **Z** **V** **H**

The carry bit will also be set by subtraction operations when the second operand is larger than the first operand. In the example above, the subtraction `0x01` – `0x03` evaluates to `0xfe`.

Status Register Flags (8)

		0	0	0	0	0	0	0	1	0x01
–		0	0	0	0	0	0	1	1	0x03
	1	1	1	1	1	1	1	1	0	0xfe

Carry

Flags: **C** **N** **Z** **V** **H**

Note that the 'overflow' caused by both subtraction and addition is irrelevant when two's complement is being used, but can be an indication of incorrect results with unsigned arithmetic.

Status Register Flags (9)

		1	1	1	0	0	1	1	0	0xe6
–		1	1	1	0	0	1	1	0	0xe6
	0	0	0	0	0	0	0	0	0	0x00

Carry

Flags: **C** **N** **Z** **V** **H**

The **Zero (Z)** flag is set whenever the 8-bit result of an operation is exactly zero. The Z flag is set by almost every arithmetic or bitwise instruction.

Status Register Flags (10)

		0	1	1	1	1	1	1	1	0x7f
+		1	0	0	0	0	0	0	1	0x81
	1	0	0	0	0	0	0	0	0	0x00

Carry

Flags: **C** **N** **Z** **V** **H**

Note that the Z flag might be set in conjunction with other flags, as in the addition above.

Status Register Flags (11)

		0	0	0	0	0	0	0	1	0x01
−		0	0	0	0	0	0	1	1	0x03
	1	1	1	1	1	1	1	1	0	0xfe

Carry

Flags: **C** **N** **Z** **V** **H**

The **Negative (N)** flag is set whenever the result of an operation is negative under a two's complement interpretation (that is, if bit 7 is set to 1). In the case above, if signed arithmetic is assumed, then the result `0xfe` $= (-2)_{10}$ is a negative value.

Status Register Flags (12)

		0	1	1	1	1	1	1	1	0x7f
+		0	0	0	0	0	0	1	1	0x03
	0	1	0	0	0	0	0	1	0	0x82

Carry

Flags: C **N** Z **V** **H**

Note that the negative flag is irrelevant if unsigned arithmetic is being used. The operation `0x7f` + `0x03` is equivalent to $127 + 3 = 130$ in unsigned arithmetic, but the N flag is set since the value would be negative in a signed context.

Status Register Flags (13)

		0	0	0	0	0	0	0	1	0x01
−		0	0	0	0	0	0	1	1	0x03
	1	1	1	1	1	1	1	1	0	0xfe

Carry

Flags: **C** **N** **Z** **V** **H**

Recall that the overflow from `0xff` to `0x00` handled by the C bit is not an 'overflow' in two's complement, since `0xff` $= (-1)_{10}$ and `0x00` $= (0)_{10}$. However, an 'overflow' can still occur: if two large positive numbers are added, the result might overflow and become negative.

Status Register Flags (14)

		0	1	1	1	0	0	0	0	0x70
+		0	1	1	1	0	0	0	0	0x70
	0	1	1	1	0	0	0	0	0	0xe0
	Carry									

Flags: **C** **N** **Z** **V** **H**

The **Two's Complement Overflow (V)** flag is set whenever an operation results in a two's complement overflow (e.g. 'Positive + Positive = Negative' or 'Negative - Positive = Positive').

Status Register Flags (15)

		0	1	1	1	0	0	0	0	0x70
+		0	1	1	1	0	0	0	0	0x70
	0	1	1	1	0	0	0	0	0	0xe0
	Carry									

Flags: **C** **N** **Z** **V** **H**

In the example above, $\texttt{0x70} + \texttt{0x70} = (112)_{10} + (112)_{10}$ gives the result 0xe0, which is -32 in two's complement. Therefore, the V flag is set to indicate that the range of 8-bit two's complement has been exceeded.

Status Register Flags (16)

		1	0	0	1	1	1	0	0	0x9c
−		0	0	0	1	1	1	1	0	0x1e
	0	0	1	1	1	1	1	1	0	0x7e

Carry

Flags: **C** **N** **Z** **V** **H**

Similarly, the operation $(-100)_{10} - (30)_{10} = \texttt{0x9c} - \texttt{0x1e}$ gives the result `0x7e`, which is $+126$ in two's complement, so the V flag is set.

Status Register Flags (17)

		0	1	1	1	0	0	0	0	0x70
+		0	0	0	1	0	1	0	1	0x15
	0	1	0	0	0	0	1	0	1	0x85

Carry

Flags: **C** **N** **Z** **V** **H**

In general, the V flag is set whenever the result of an operation (like $A + B$ or $A - B$) is out of range for 8-bit two's complement. For example $112 + 21$ (shown above) should equal 133, but since 133 is larger than the upper limit (127) of 8-bit two's complement, the V flag is set.

Status Register Flags (18)

		0	0	0	1	0	1	1	1	0x17
+		0	1	1	1	1	0	0	0	0x78
	0	1	0	0	0	1	1	1	1	0x8f

Carry

Flags: **C** **N** **Z** **V** **H**

The **Half-byte Carry (H)** flag is set whenever a carry occurs from bit 3 to bit 4. For the addition above, the H flag is not set.

Status Register Flags (19)

		0	0	0	1	1	0	0	1	0x19
+		0	0	0	0	1	0	0	0	0x08
	0	0	0	1	0	0	0	0	1	0x21
	Carry									

Flags: **C** **N** **Z** **V** **H**

The addition above sets the H flag. The H flag behaves similarly to the C flag (including as a borrow indicator for subtraction and comparison), but for the lower 4 bits of the value instead of all 8 bits.

Status Register Flags (20)

		1	0	0	0	0	0	0	0
+		1	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	0

0x80 + 0x80 = 0x00

Carry

Flags: **C** **N** **Z** **V** **H**

The unusual case above sets three of the four primary flags, since 0x80 is negative, but $0x80 + 0x80 = 0$ due to an unsigned overflow.

Status Register Flags (21)

		1	1	1	0	0	0	0	1	0xe1
−		1	1	1	0	0	1	1	0	0xe6
	1	1	1	1	1	1	0	1	1	0xfb

Carry

Flags: **C** **N** **Z** **V** **H**

Question: Is `0xe1` less than `0xe6`?

Status Register Flags (22)

		1	1	1	0	0	0	0	1	0xe1
−		1	1	1	0	0	1	1	0	0xe6
	1	1	1	1	1	1	0	1	1	0xfb

Carry

Flags: **C** **N** **Z** **V** **H**

Answer: It depends. In unsigned arithmetic `0xe1` $= (225)_{10}$ and `0xe6` $= (230)_{10}$, but as two's complement values, `0xe1` $= (-31)_{10}$ and `0xe6` $= (-26)_{10}$.

Status Register Flags (23)

		1	1	1	0	0	0	0	1	0xe1
−		1	1	1	0	0	1	1	0	0xe6
	1	1	1	1	1	1	0	1	1	0xfb

Carry

Flags: **C** **N** **Z** **V** **H**

In unsigned arithmetic, the carry bit will be set by the calculation `A` − `B` if and only if `A` < `B`. As a result, the `BRLO` (branch if lower) instruction will branch if $C = 1$, while the converse instruction `BRSH` (branch if same or higher) will branch if $C = 0$.

Status Register Flags (24)

		1	1	1	0	0	0	0	1	0xe1
−		1	1	1	0	0	1	1	0	0xe6
	1	1	1	1	1	1	0	1	1	0xfb

Carry

Flags: **C** **N** **Z** **V** **H**

For signed arithmetic, a more nuanced test is needed. In the case above, the value `0xe1` − `0xe6` = `0xfb`, which is −5 in two's complement.

Status Register Flags (25)

		1	1	1	0	0	0	0	1	0xe1
−		1	1	1	0	0	1	1	0	0xe6
	1	1	1	1	1	1	0	1	1	0xfb

Carry

Flags: **C** **N** **Z** **V** **H**

Idea: If the operation `A` − `B` sets the N flag, then $A < B$ (since the result will only be negative if $A - B < 0$).

Status Register Flags (26)

		1	1	1	0	0	0	0	1	0xe1
−		1	1	1	0	0	1	1	0	0xe6
	1	1	1	1	1	1	0	1	1	0xfb

Carry

Flags: **C** **N** **Z** **V** **H**

For the example above, the N flag idea seems to hold.

Status Register Flags (27)

		0	1	1	1	0	0	0	0	0x70
−		1	1	1	0	1	1	1	1	0xef
	1	1	0	0	0	0	0	0	1	0x81

Carry

Flags: **C** **N** **Z** **V** **H**

However, consider the case above, which compares `0x70` (112) and `0xef` (-17). The N flag is set, since the result $112-(-17) = 112+17$ results in a two's complement overflow into negative numbers, but 112 is not less than -17.

Status Register Flags (28)

		1	1	1	0	1	1	1	1	0xef
−		0	1	1	1	0	0	0	0	0x70
	0	0	1	1	1	1	1	1	1	0x7f

Carry

Flags: **C** **N** **Z** **V** **H**

Similarly, consider the case above, which compares the same two values in reverse. Clearly, $-17 < 112$, but the negative flag is not set by the operation (although again the V flag is set since $-17 - 112 = -129$ is out of range of 8-bit two's complement).

Status Register Flags (29)

		1	1	1	0	1	1	1	1	0xef
–		0	1	1	1	0	0	0	0	0x70
	0	0	1	1	1	1	1	1	1	0x7f

Carry

Flags: **C** **N** **Z** **V** **H**

Clearly the N flag by itself does not determine the relationship between the two values. The V flag also does not determine the relationship (since in the previous examples there were cases where both flags were set for either the 'less than' or 'greater than' relationship).

Status Register Flags (30)

		1	1	1	0	1	1	1	1	0xef
–		0	1	1	1	0	0	0	0	0x70
	0	0	1	1	1	1	1	1	1	0x7f

Carry

Flags: **C** **N** **Z** **V** **H**

However, the result of the subtraction `A` – `B` does dictate the relationship between `A` and `B` if both the N and V flags are used.

Status Register Flags (31)

If $\texttt{A} < \texttt{B}$, then clearly the (theoretical) value of $\texttt{A} - \texttt{B}$ will be negative.

- If $\texttt{A} - \texttt{B}$ is within the range of representable negative values ($-128 to -1$), then the N flag will be set and the V flag will be cleared (since no two's complement overflow occurred if the value was within range).
- If $\texttt{A} - \texttt{B}$ is less than -128, then it will wrap around to positive values, so N will be cleared and V will be set.

Status Register Flags (32)

If $\texttt{A} > \texttt{B}$, then the (theoretical) value of $\texttt{A} - \texttt{B}$ will be positive or zero.

- If $\texttt{A} - \texttt{B}$ is within the range of representable non-negative values ($0 to 127$), then both the N and V flags will be clear.
- If $\texttt{A} - \texttt{B}$ is larger than 127, the value will wrap around to negative values, resulting in both the N and V flags being set.

Status Register Flags (33)

		1	1	1	0	1	1	1	1	0xef
−		0	1	1	1	0	0	0	0	0x70
	0	0	1	1	1	1	1	1	1	0x7f
	Carry									

Flags: **C** **N** **Z** **V** **H**

Therefore, `A` $<$ `B` if exactly one of the N and V flags is set, and `A` $\geq$ `B` if neither or both of the flags are set. The `BRLT` (branch if less than) instruction branches when $N \oplus V = 1$ (where $\oplus$ is the exclusive-OR operation) and the converse `BRGE` (branch if greater or equal) instruction branches if $N \oplus V = 0$.

Exercises

Exercise: Explain why there are no two values A and B such that $A - B$ will set both of the carry and zero flags to 1.

Exercise: For each of the flag configurations below, find two 8-bit values A and B such that the computation $A - B$ sets each flag as listed.

- $C = 1, N = 0, Z = 0, V = 0$
- $C = 1, N = 1, Z = 0, V = 0$
- $C = 1, N = 1, Z = 0, V = 1$

Exercise: For each of the flag configurations below, find two 8-bit values A and B such that the computation $A + B$ sets each flag as listed.

- $C = 0, N = 1, Z = 0, V = 1$
- $C = 1, N = 0, Z = 1, V = 0$
- $C = 0, N = 1, Z = 0, V = 1$