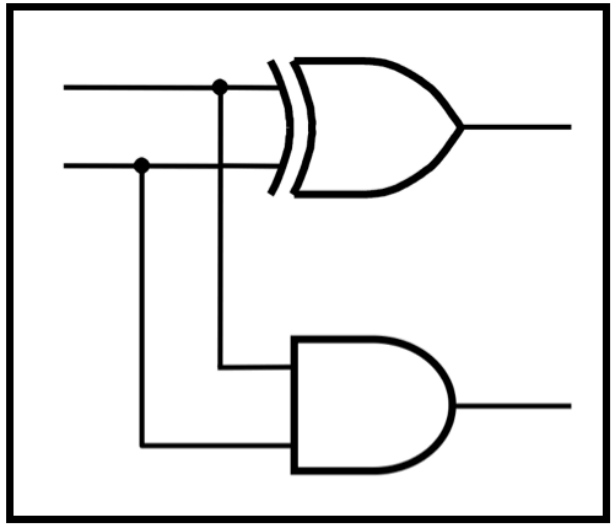


# CprE 281: Digital Logic

**Instructor: Alexander Stoytchev**

**http://www.ece.iastate.edu/~alexs/classes/**

# Floating Point Numbers

*CprE 281: Digital Logic*
*Iowa State University, Ames, IA*

# Administrative Stuff

- **HW 6 is out**
- **It is due on Monday Oct 10 @ 10pm.**

# More Details on Comparison Circuits

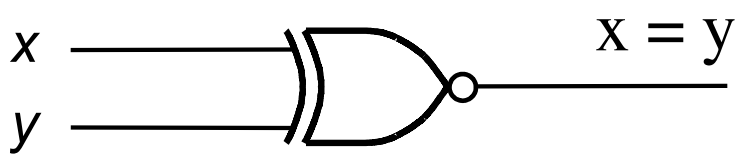
# Comparison of 1-bit numbers

# Equal

| x | y | x = y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

# Equal

| x | y | x = y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

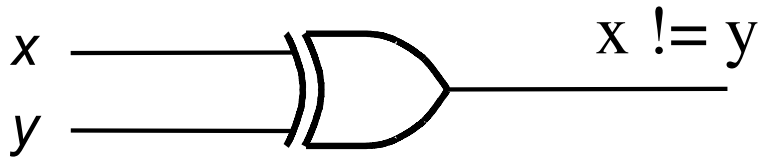


# Not Equal

| x | y | x != y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# Not Equal

| x | y | x != y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



# Less

| x | y | x < y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

# Less or Equal

| x | y | x <= y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

# Less or Equal

| x | y | x < y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

or

| x | y | x = y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

=

| x | y | x <= y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

# Greater

| x | y | x > y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# Greater or Equal

| x | y | x >= y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

# Greater or Equal

| x | y | x > y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

or

| x | y | x = y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

=

| x | y | x >= y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

# Some Interesting Dualities

# Equal

| x | y | x = y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

# Not Equal

| x | y | x != y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

## Greater

| x | y | x > y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

## Less or Equal

| x | y | x <= y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

## Greater or Equal

| x | y | x >= y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

## Less

| x | y | x < y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

# Some Interesting Dualities

- $\overline{\text{Equal}} = \text{Not Equal}$
- $\overline{\text{Greater}} = \text{Less or Equal}$
- $\overline{\text{Less}} = \text{Greater or Equal}$

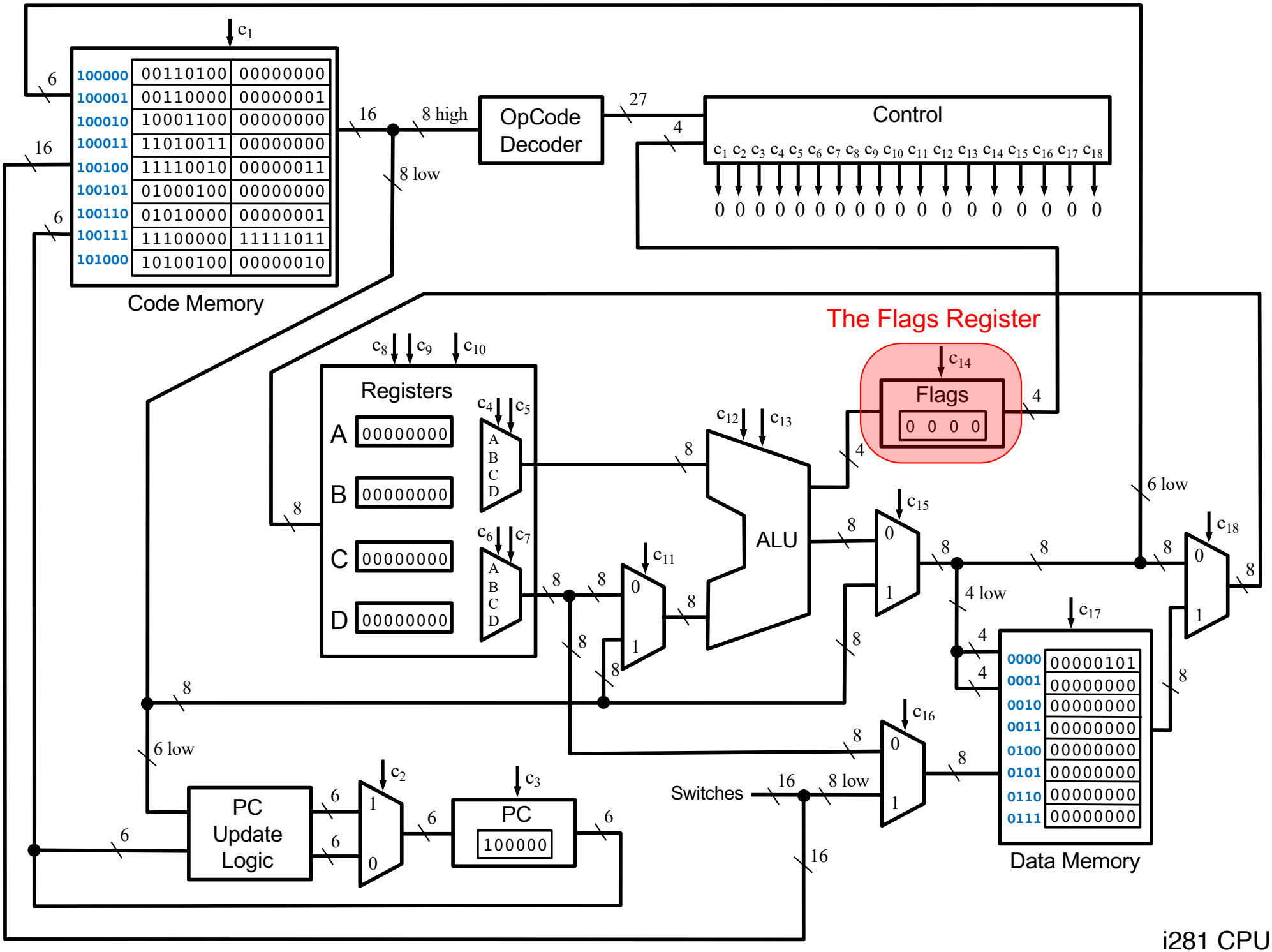
# Comparison of n-bit signed numbers
## (stored in 2's complement representation)

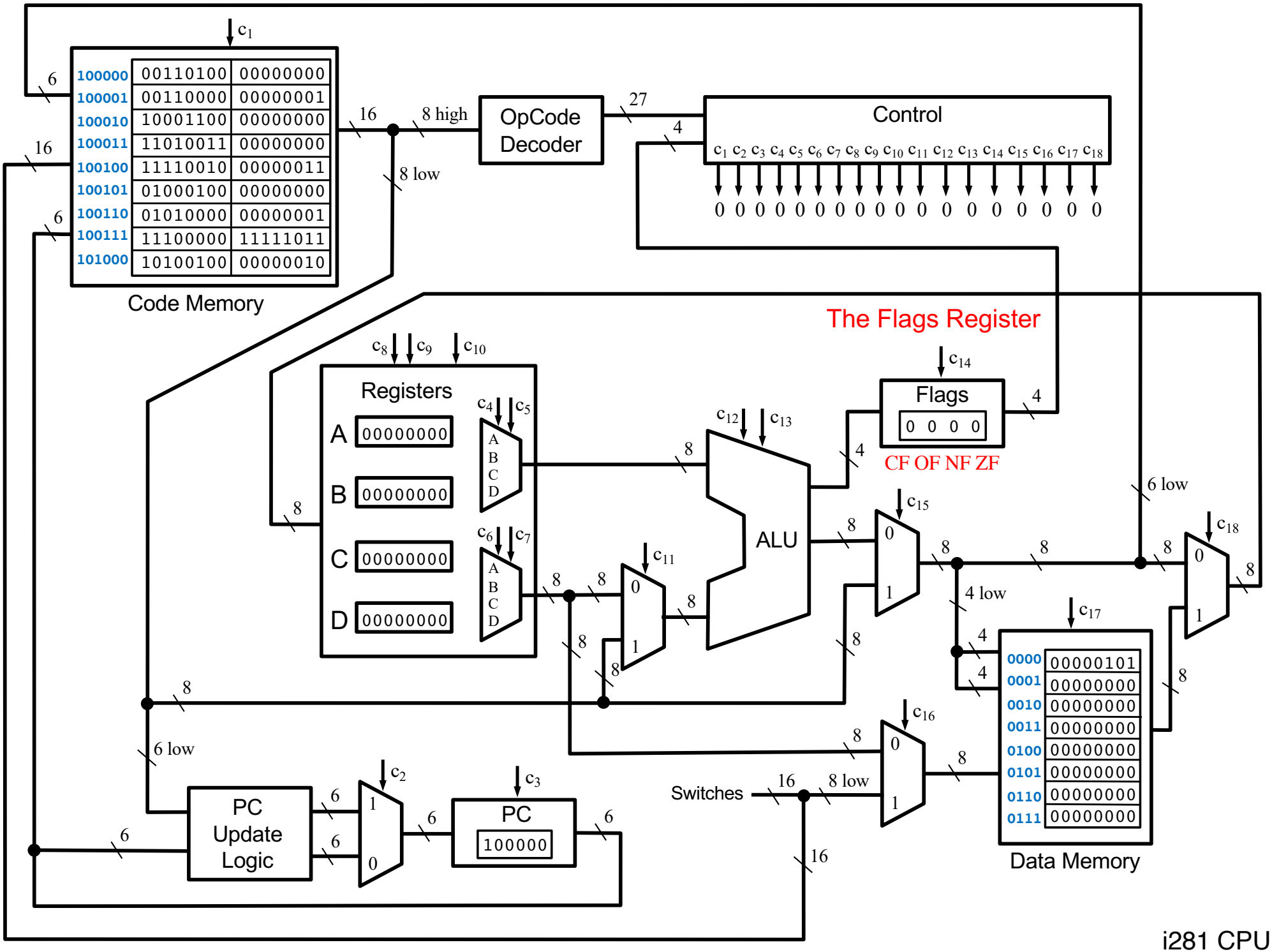
# Comparison of n-bit signed numbers
## (stored in 2's complement representation)

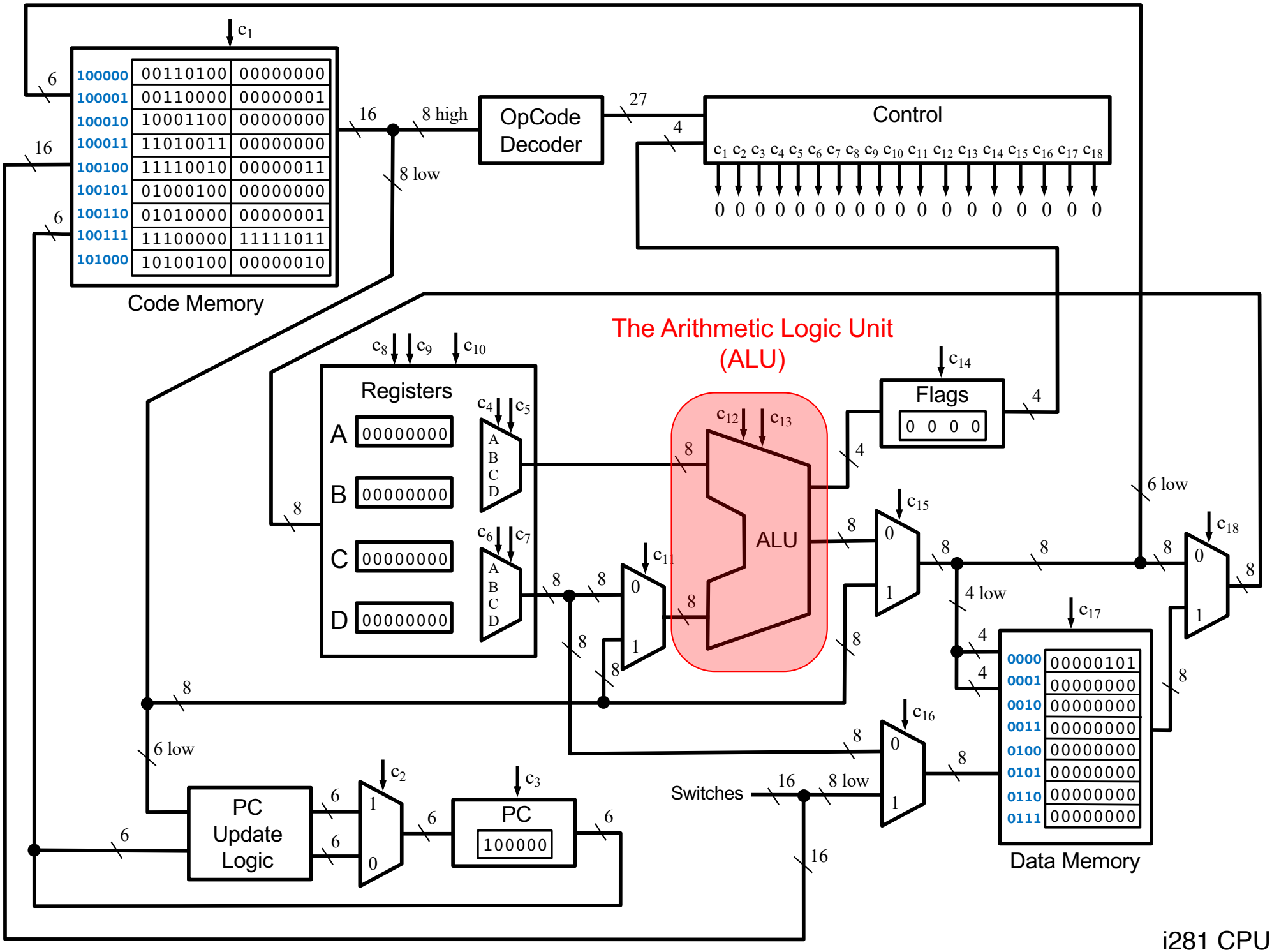
The comparison is done with subtraction and then looking at the flags.

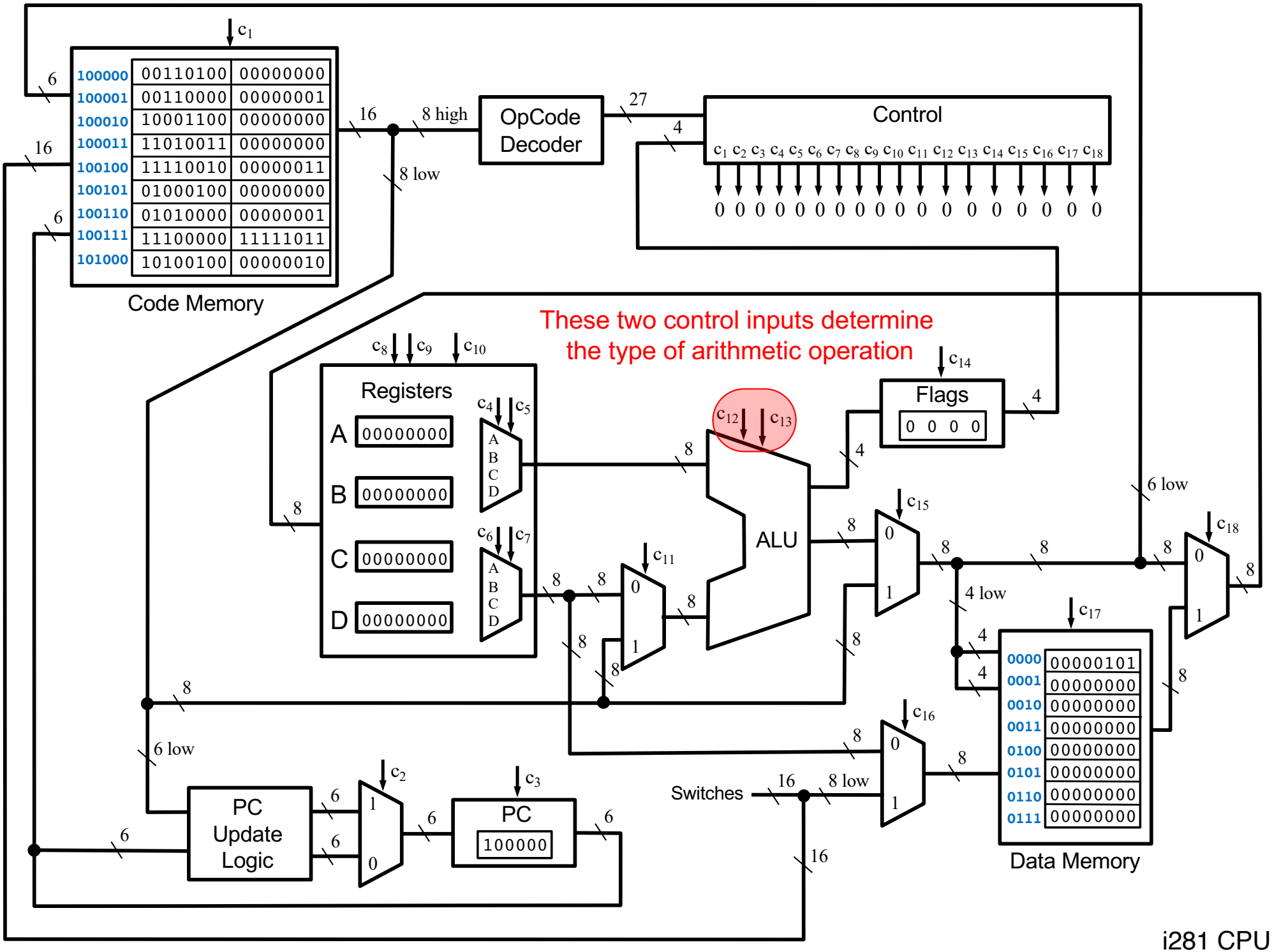
# Abbreviations for the Flags

- **Carry Flag (CF)**
- **Overflow Flag (OF)**
- **Negative Flag (NF)**
- **Zero Flag (ZF)**


c1
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
The Flags Register
c14
Flags
0 0 0 0
4
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
8
c12 c13
ALU
c11
c15
c16
c17
c18
6 low
4 low
Switches
8 low
16
PC Update Logic
c2
c3
PC
100000
6
Data Memory
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
i281 CPU


c1
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
The Flags Register
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c14
Flags
0 0 0 0
CF OF NF ZF
c12 c13
ALU
c11
c15
c16
c18
6 low
4 low
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
Switches
8 low
c2
c3
PC Update Logic
PC
100000
i281 CPU


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
8 high
8 low
OpCode Decoder
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
The Arithmetic Logic Unit (ALU)
Registers
A 00000000
B 00000000
C 00000000
D 00000000
ALU
Flags
0 0 0 0
PC Update Logic
PC
100000
Switches
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
i281 CPU


c1
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
These two control inputs determine
the type of arithmetic operation
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c12 c13
ALU
c14
Flags
0 0 0 0
c11
c15
c16
c18
6 low
4 low
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
Switches
8 low
c2
c3
PC Update Logic
PC
100000
i281 CPU

# This ALU Can Perform 4 Operations

| ALU_SELECT1 | ALU_SELECT0 | Operation |
|---|---|---|
| 0 | 0 | SHIFTL |
| 0 | 1 | SHIFTR |
| 1 | 0 | ADD |
| 1 | 1 | SUB/CMP |

# This ALU Can Perform 4 Operations

Names of these control lines

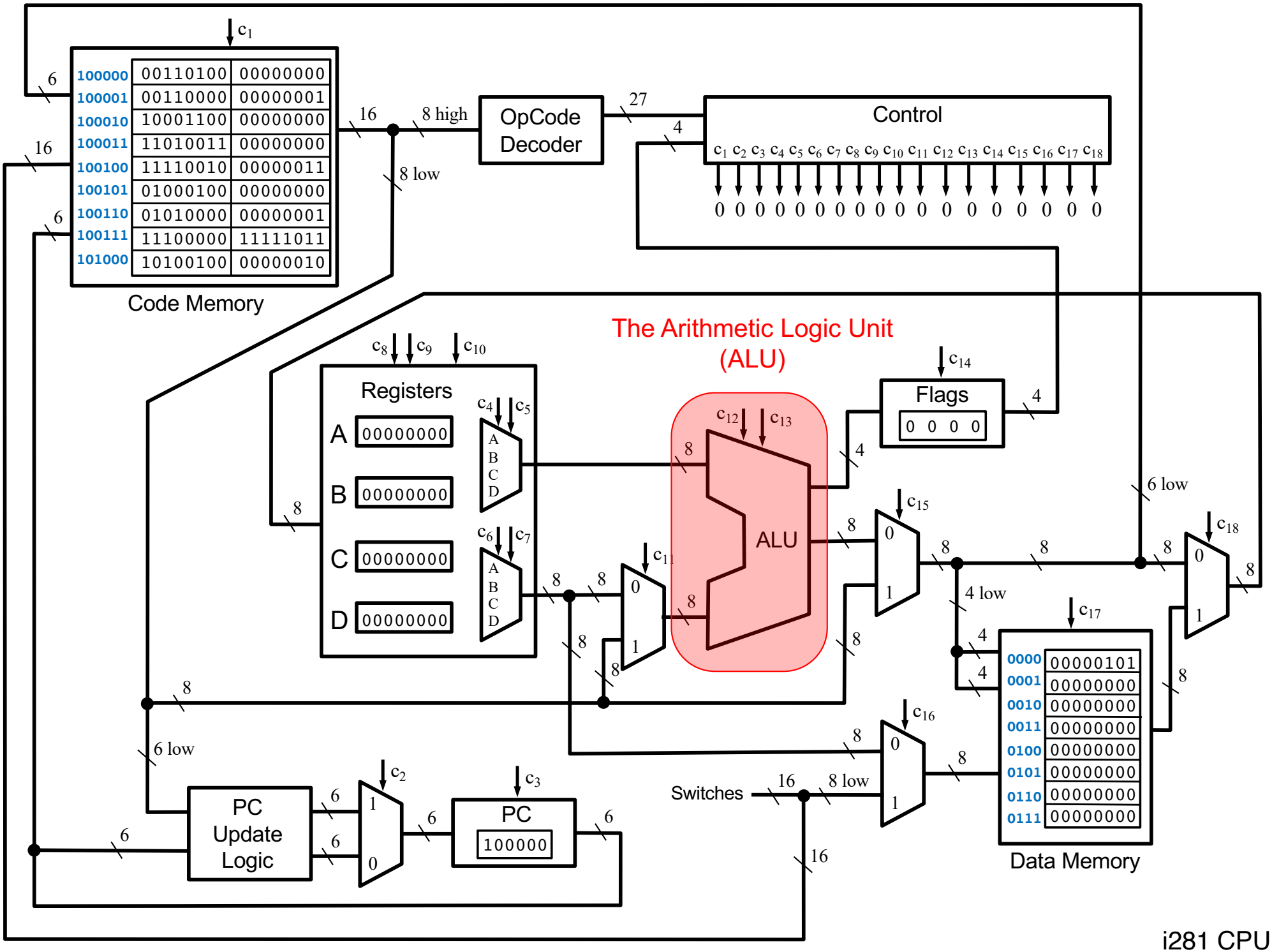
| $C_{12}$ | $C_{13}$ | |
|---|---|---|
| ALU_SELECT1 | ALU_SELECT0 | Operation |
| 0 | 0 | SHIFTL |
| 0 | 1 | SHIFTR |
| 1 | 0 | ADD |
| 1 | 1 | SUB/CMP |

# This ALU Can Perform 4 Operations

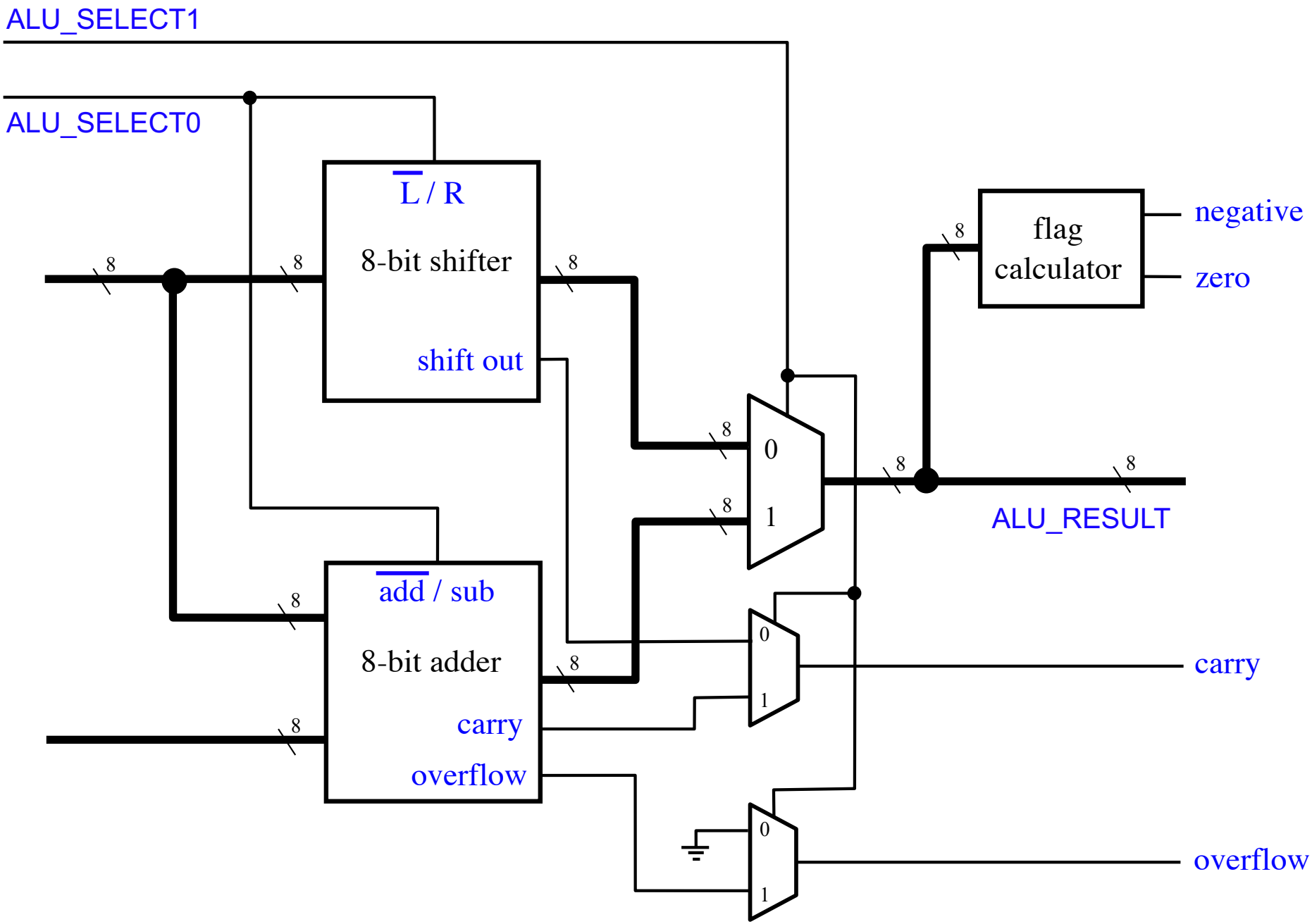
| $C_{12}$ | $C_{13}$ | |
|---|---|---|
| ALU_SELECT1 | ALU_SELECT0 | Operation |
| 0 | 0 | SHIFTL |
| 0 | 1 | SHIFTR |
| 1 | 0 | ADD |
| 1 | 1 | SUB/CMP |

Both SUB and CMP are implemented as subtraction. They both set the flags.

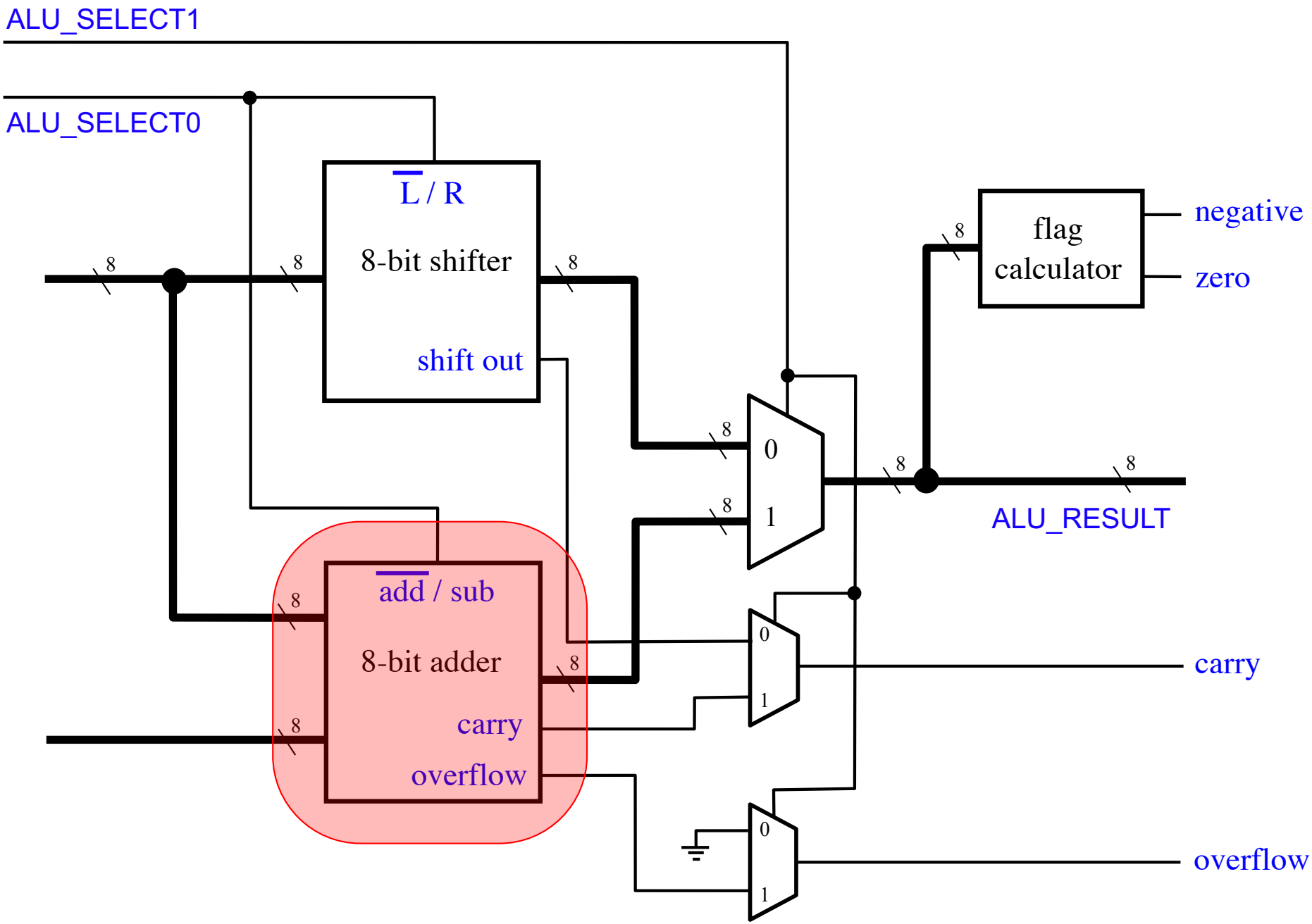
The difference is that CMP does not write back the result of the subtraction to the registers. Only the side effect through the flags remains.


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
8 high
8 low
OpCode Decoder
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
The Arithmetic Logic Unit (ALU)
Registers
A 00000000
B 00000000
C 00000000
D 00000000
ALU
Flags
0 0 0 0
PC Update Logic
PC
100000
Switches
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
i281 CPU

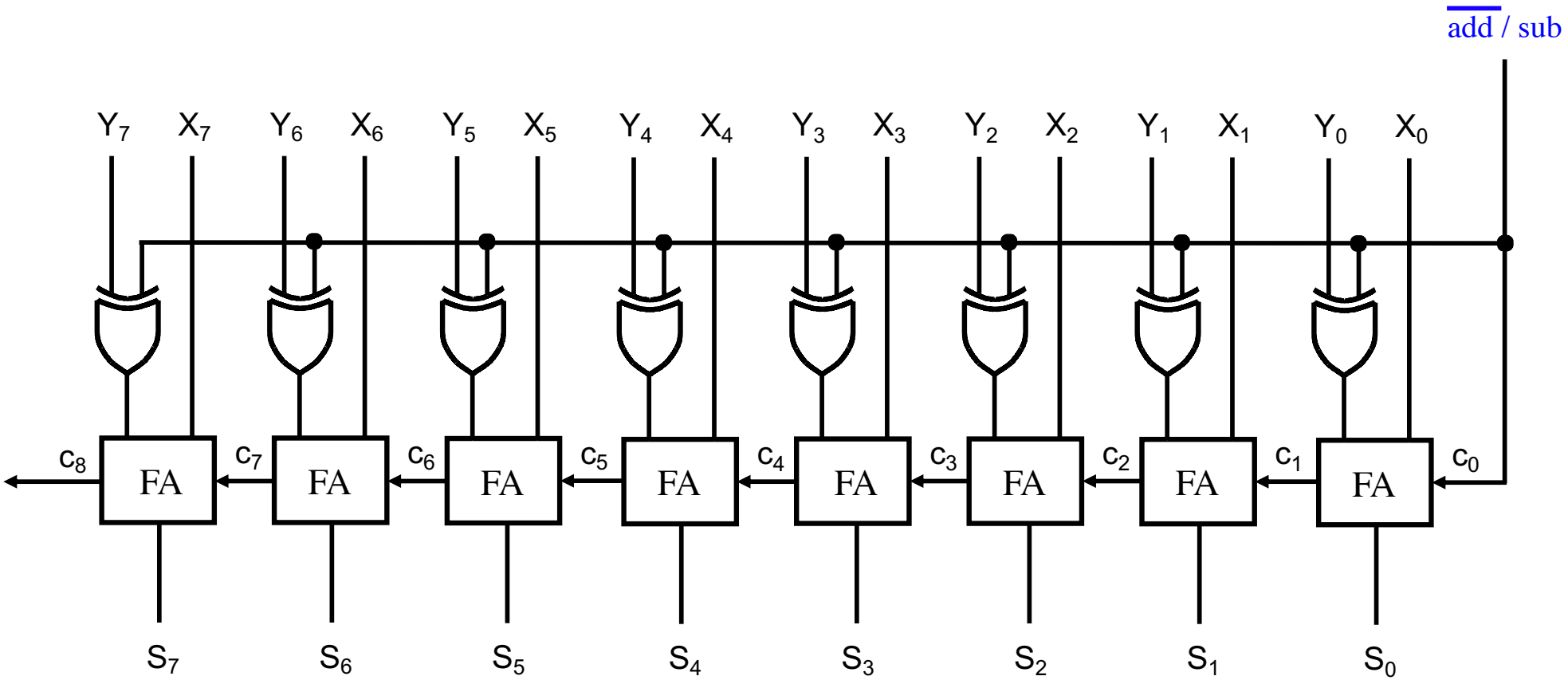
# The ALU


ALU_SELECT1
ALU_SELECT0
L / R
8-bit shifter
shift out
add / sub
8-bit adder
carry
overflow
flag calculator
negative
zero
ALU_RESULT
carry
overflow
0
1
8

# The Adder / Subtractor

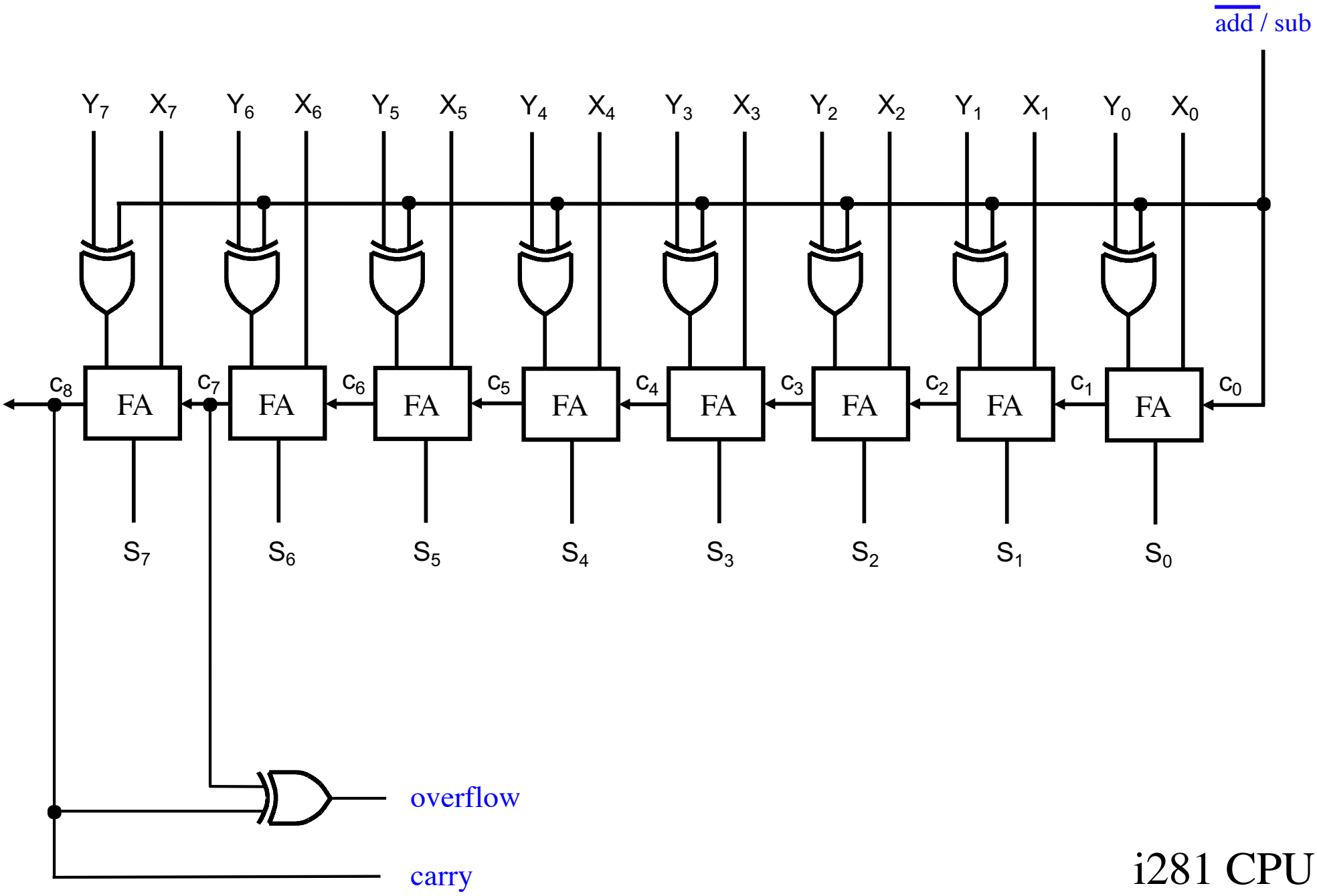


# The Adder / Subtractor

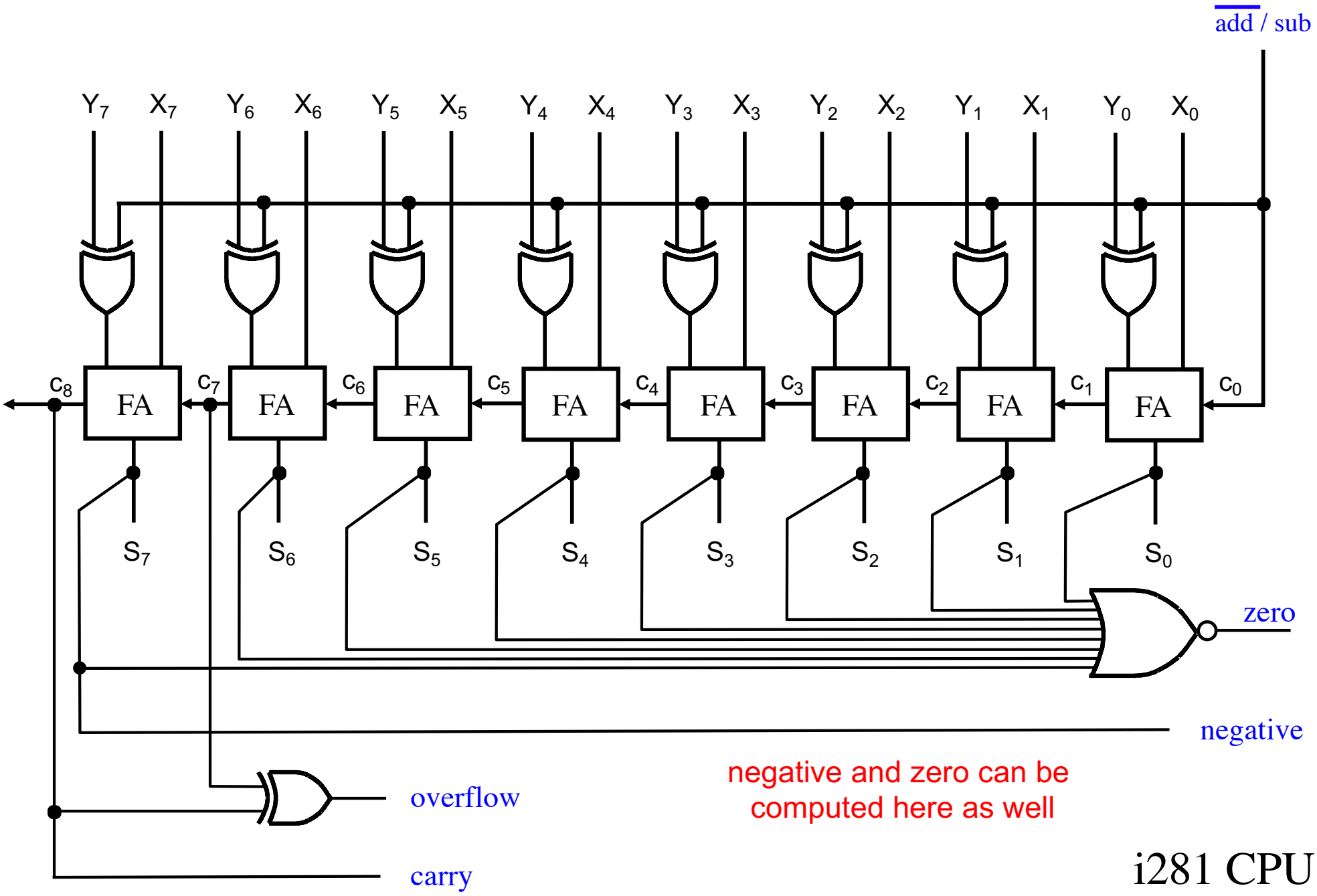



This is an 8-bit ripple-carry adder. Note that the X and Y lines are swapped.

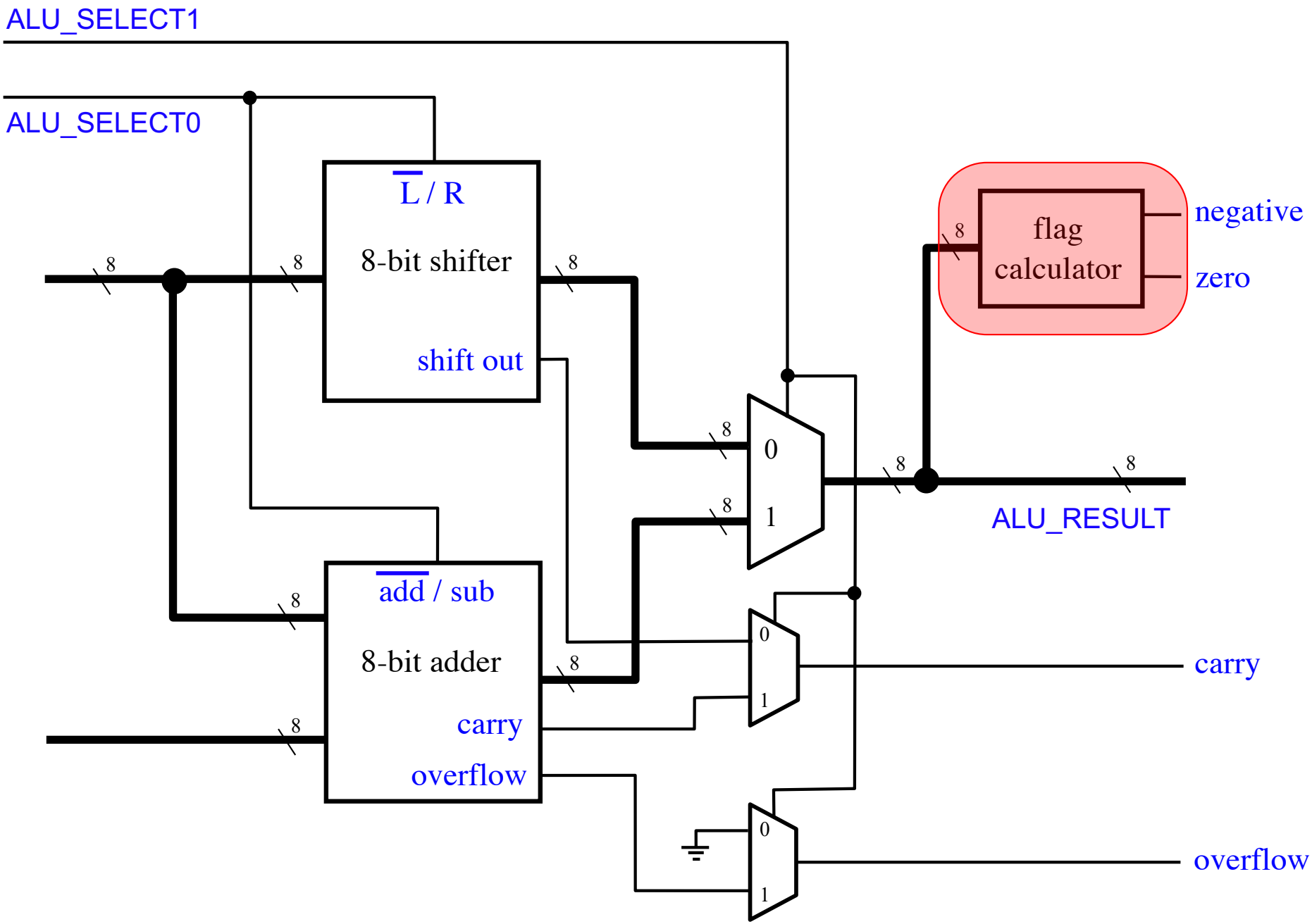
# The Adder / Subtractor



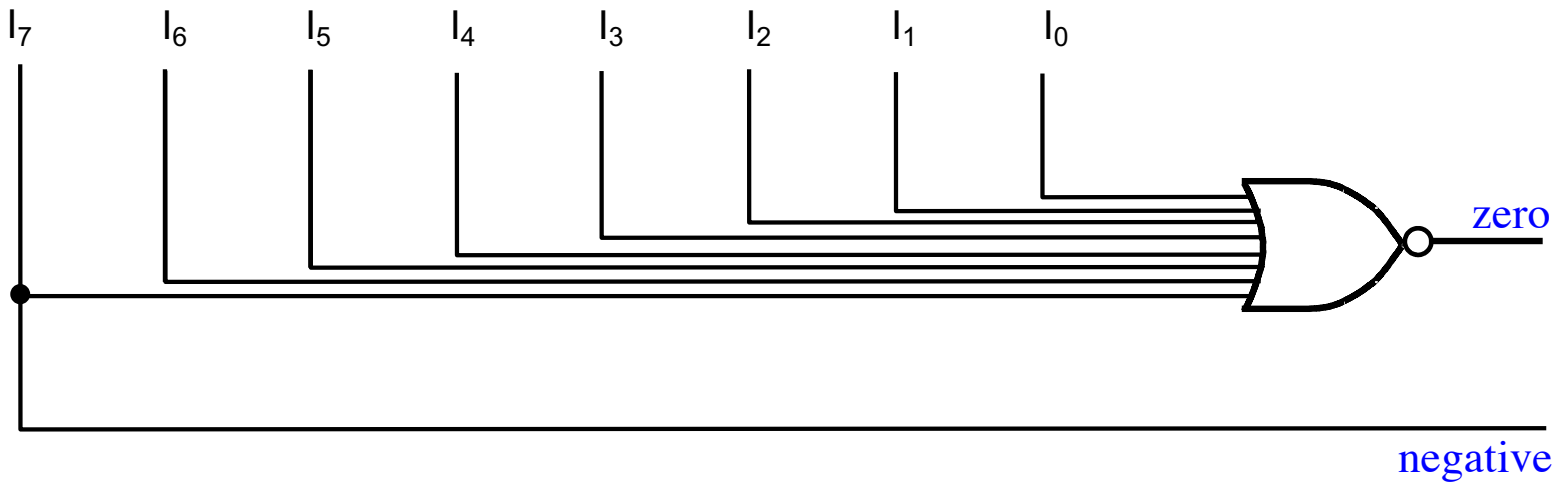
# The Adder / Subtractor



# The ALU Flag Calculator



# The ALU Flag Calculator



# Abbreviations for the Flags

- **Carry Flag (CF)**
- **Overflow Flag (OF)**
- **Negative Flag (NF)**
- **Zero Flag (ZF)**

# Abbreviations for the Flags

- **Carry Flag (CF)**
- **Overflow Flag (OF)**
- **Negative Flag (NF)**
- **Zero Flag (ZF)**

In some CPU architectures the carry flag means borrow. And it could be inverted relative to the previous diagram.

# Comparison of Signed Numbers

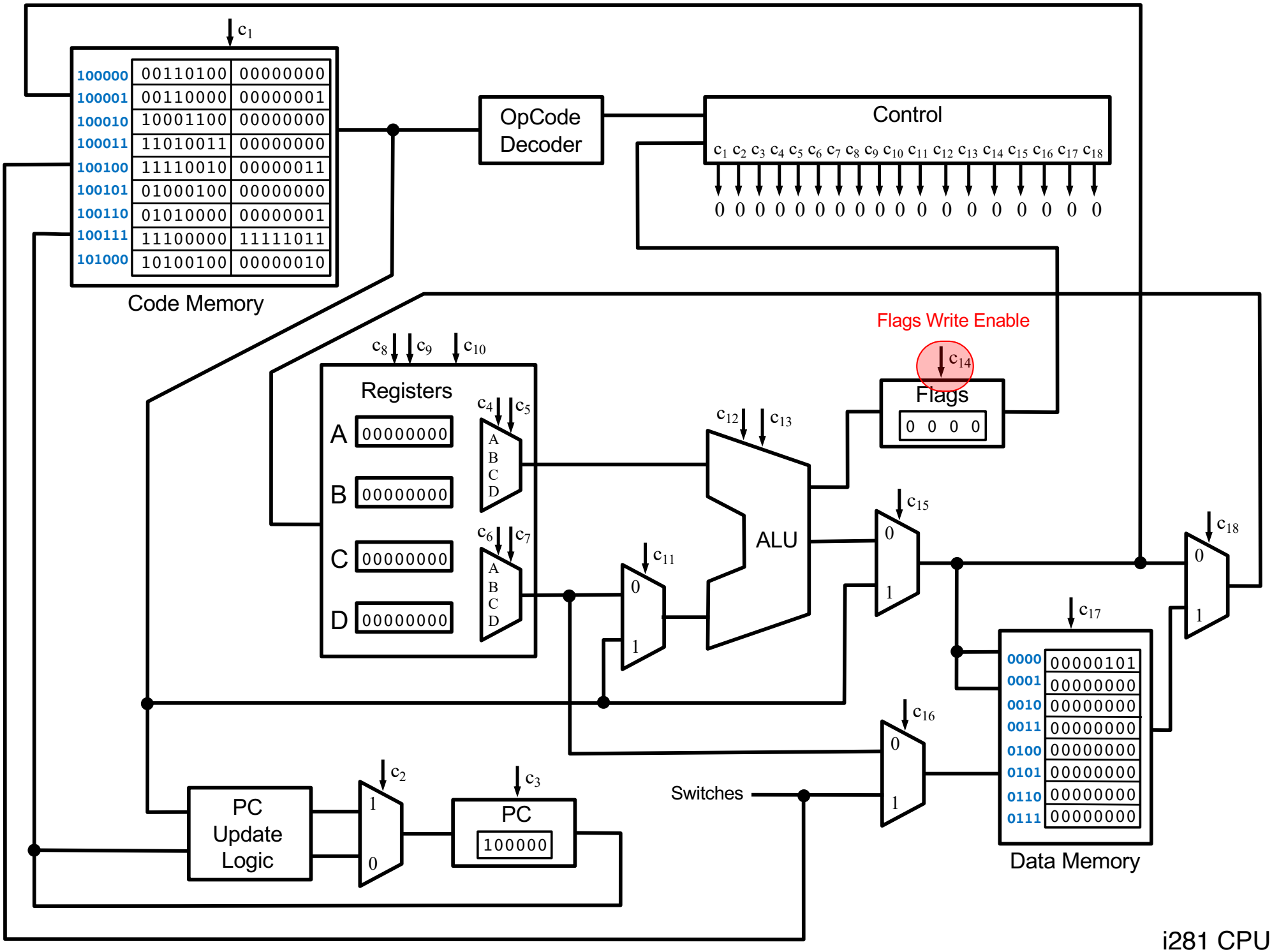
- **Equal** — **ZF = 1**
- **Not equal** — **ZF = 0**
- **Greater** — **ZF = 0 and NF = OF**
- **Greater or Equal** — **NF = OF**
- **Less** — **NF != OF**
- **Less or Equal** — **ZF = 1 or NF != OF**

# Comparison of Signed Numbers

- **Equal** — **ZF**
- **Not equal** — $\overline{ZF}$
- **Greater** — $\overline{ZF} \bullet XNOR( NF, OF )$
- **Greater or Equal** — **XNOR( NF, OF )**
- **Less** — **XOR( NF, OF )**
- **Less or Equal** — **ZF + XOR( NF, OF )**

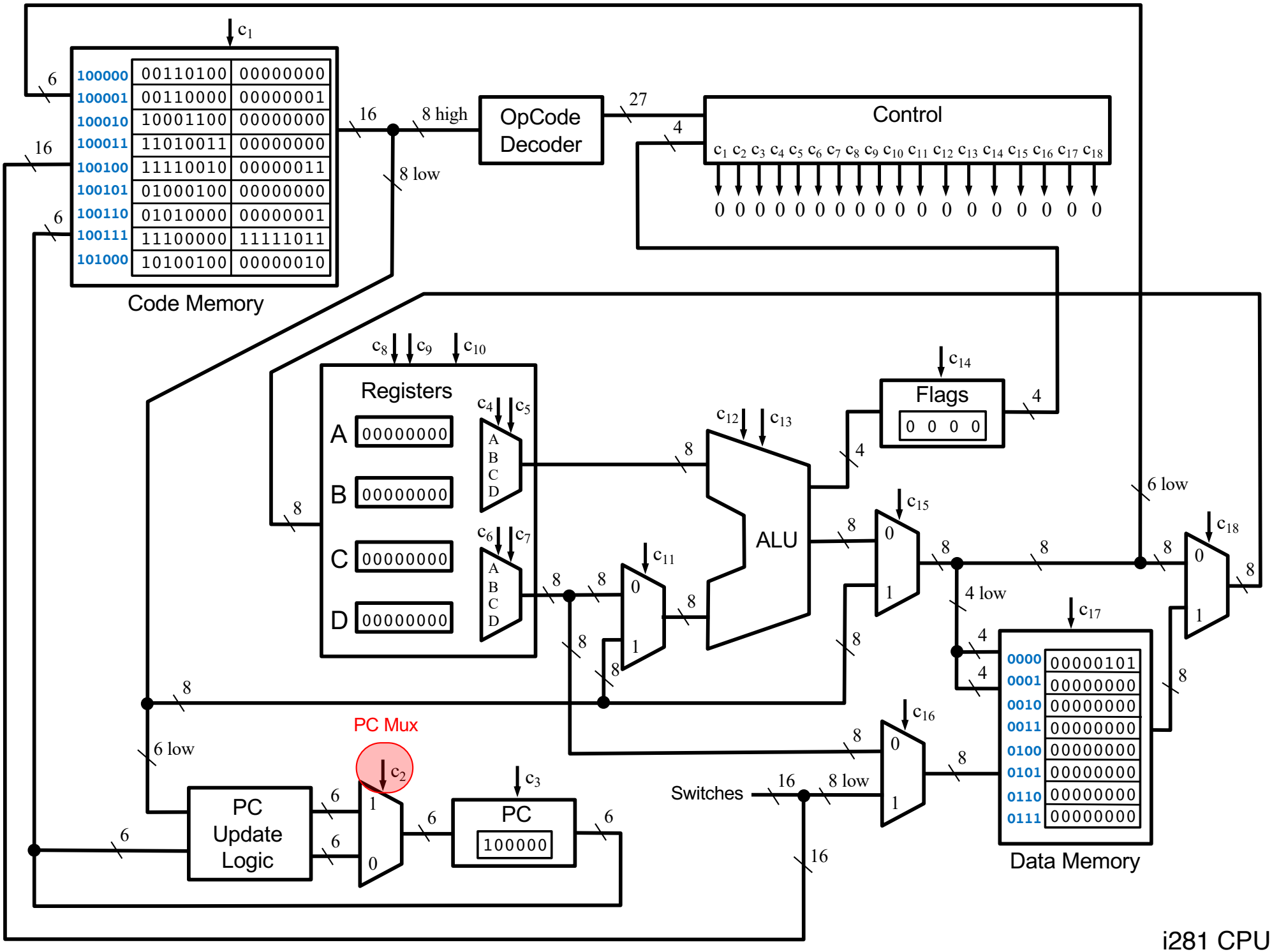
# These 7 Instructions Update the Flags

| | |
|---|---|
| NOOP | NO OPeration |
| INPUTC | INPUT into Code memory |
| INPUTCF | INPUT into Code memory with oFfset |
| INPUTD | INPUT into Data memory |
| INPUTDF | INPUT into Data memory with oFfset |
| MOVE | MOVE the contents of one register into another |
| LOADI | LOAD Immediate value |
| LOADP | LOAD Pointer address |
| ADD | ADD two registers |
| ADDI | ADD an Immediate value to a register |
| SUB | SUBtract two registers |
| SUBI | SUBtract an Immediate value from a register |
| LOAD | LOAD from a data memory address into a register |
| LOADF | LOAD with an oFfset specified by another register |
| STORE | STORE a register into a data memory address |
| STOREF | STORE with an oFfset specified by another register |
| SHIFTL | SHIFT Left all bits in a register |
| SHIFTR | SHIFT Right all bits in a register |
| CMP | CoMPare the values in two registers |
| JUMP | JUMP unconditionally to a specified address |
| BRE | BRanch if Equal |
| BRZ | BRanch if Zero |
| BRNE | BRanch if Not Equal |
| BRNZ | BRanch if Not Zero |
| BRG | BRanch if Greater |
| BRGE | BRanch if Greater than or Equal |

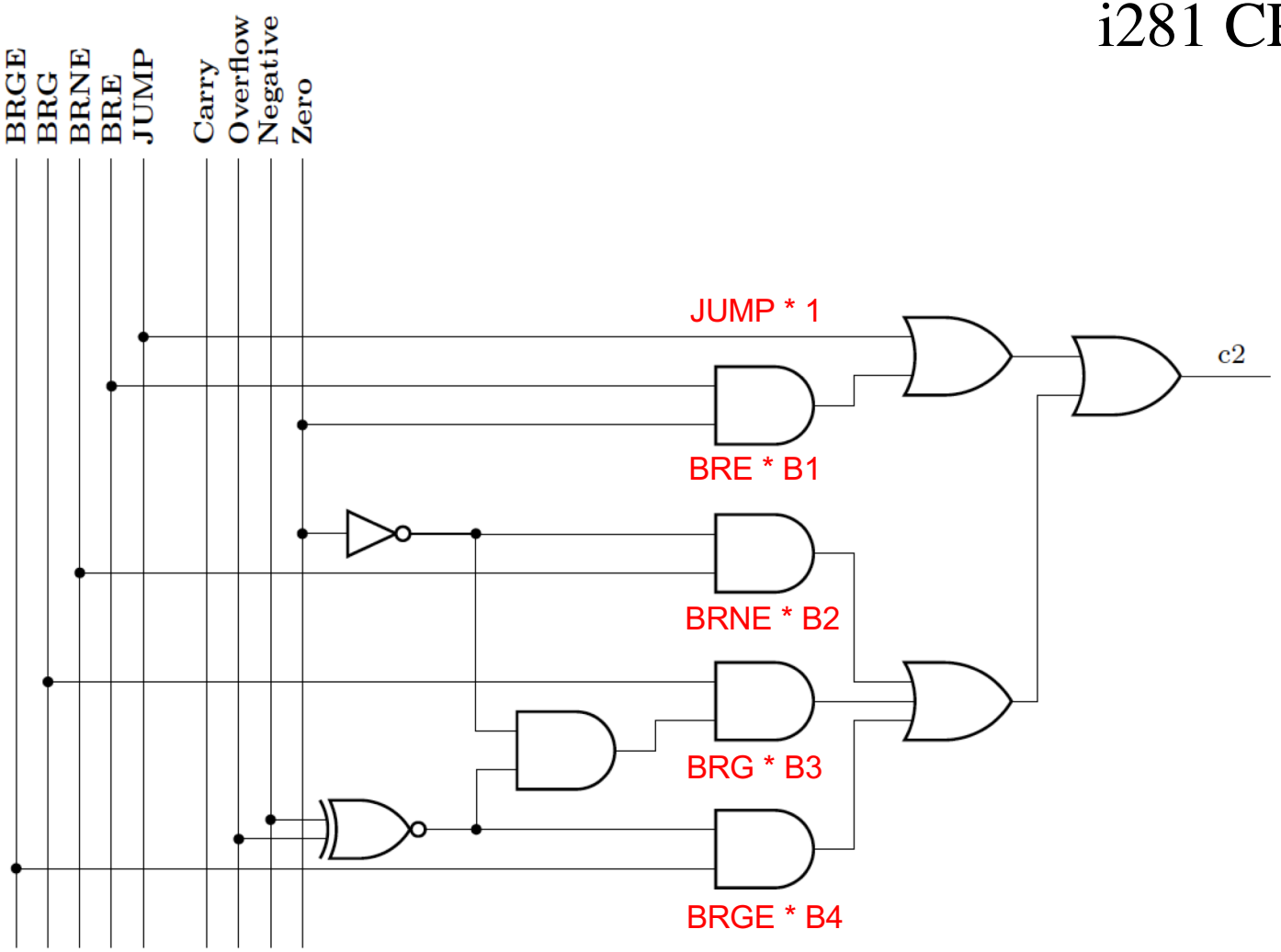

c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
OpCode Decoder
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Flags Write Enable
c14
Flags
0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c12 c13
ALU
c11
c15
c18
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
c16
Switches
c2
c3
PC Update Logic
PC
100000
i281 CPU

# These 7 Instructions Use the Flags

| | |
|---|---|
| NOOP | NO OPeration |
| INPUTC | INPUT into Code memory |
| INPUTCF | INPUT into Code memory with oFfset |
| INPUTD | INPUT into Data memory |
| INPUTDF | INPUT into Data memory with oFfset |
| MOVE | MOVE the contents of one register into another |
| LOADI | LOAD Immediate value |
| LOADP | LOAD Pointer address |
| ADD | ADD two registers |
| ADDI | ADD an Immediate value to a register |
| SUB | SUBtract two registers |
| SUBI | SUBtract an Immediate value from a register |
| LOAD | LOAD from a data memory address into a register |
| LOADF | LOAD with an oFfset specified by another register |
| STORE | STORE a register into a data memory address |
| STOREF | STORE with an oFfset specified by another register |
| SHIFTL | SHIFT Left all bits in a register |
| SHIFTR | SHIFT Right all bits in a register |
| CMP | CoMPare the values in two registers |
| JUMP | JUMP unconditionally to a specified address |
| BRE | BRanch if Equal |
| BRZ | BRanch if Zero |
| BRNE | BRanch if Not Equal |
| BRNZ | BRanch if Not Zero |
| BRG | BRanch if Greater |
| BRGE | BRanch if Greater than or Equal |


c1
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
8
c12 c13
ALU
c14
Flags
0 0 0 0
4
c11
c15
c18
6 low
4 low
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
c16
Switches
PC Mux
c2
c3
PC
100000
PC Update Logic
i281 CPU

# i281 CPU




| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| JUMP | | 1 | 1 | | | | | | | | | | | | | | | |
| BRE/BRZ | | B1 | 1 | | | | | | | | | | | | | | | |
| BRNE/BRNZ | | B2 | 1 | | | | | | | | | | | | | | | |
| BRG | | B3 | 1 | | | | | | | | | | | | | | | |
| BRGE | | B4 | 1 | | | | | | | | | | | | | | | |

$C_2$ is the OR of these five times the OPCODE

```
B1=  ZF
B2= ~ZF
B3= AND(~ZF, XNOR(NF, OF))
B4= XNOR(NF, OF)
```

Zero Flag (ZF)
Negative Flag (NF)
Overflow Flag (OF)

# Comparison of Signed Numbers

- **Equal** — **ZF**
- **Not equal** — $\overline{ZF}$
- **Greater** — $\overline{ZF} \bullet XNOR( NF, OF )$
- **Greater or Equal** — **XNOR( NF, OF )**
- **Less** — **XOR( NF, OF )**
- **Less or Equal** — **ZF + XOR( NF, OF )**

# Comparison of Signed Numbers

| | | |
|---|---|---|
| • Equal | ZF | B1 |
| • Not equal | $\overline{ZF}$ | B2 |
| • Greater | $\overline{ZF}$ • XNOR( NF, OF ) | B3 |
| • Greater or Equal | XNOR( NF, OF ) | B4 |
| • Less | XOR( NF, OF ) | |
| • Less or Equal | ZF + XOR( NF, OF ) | |

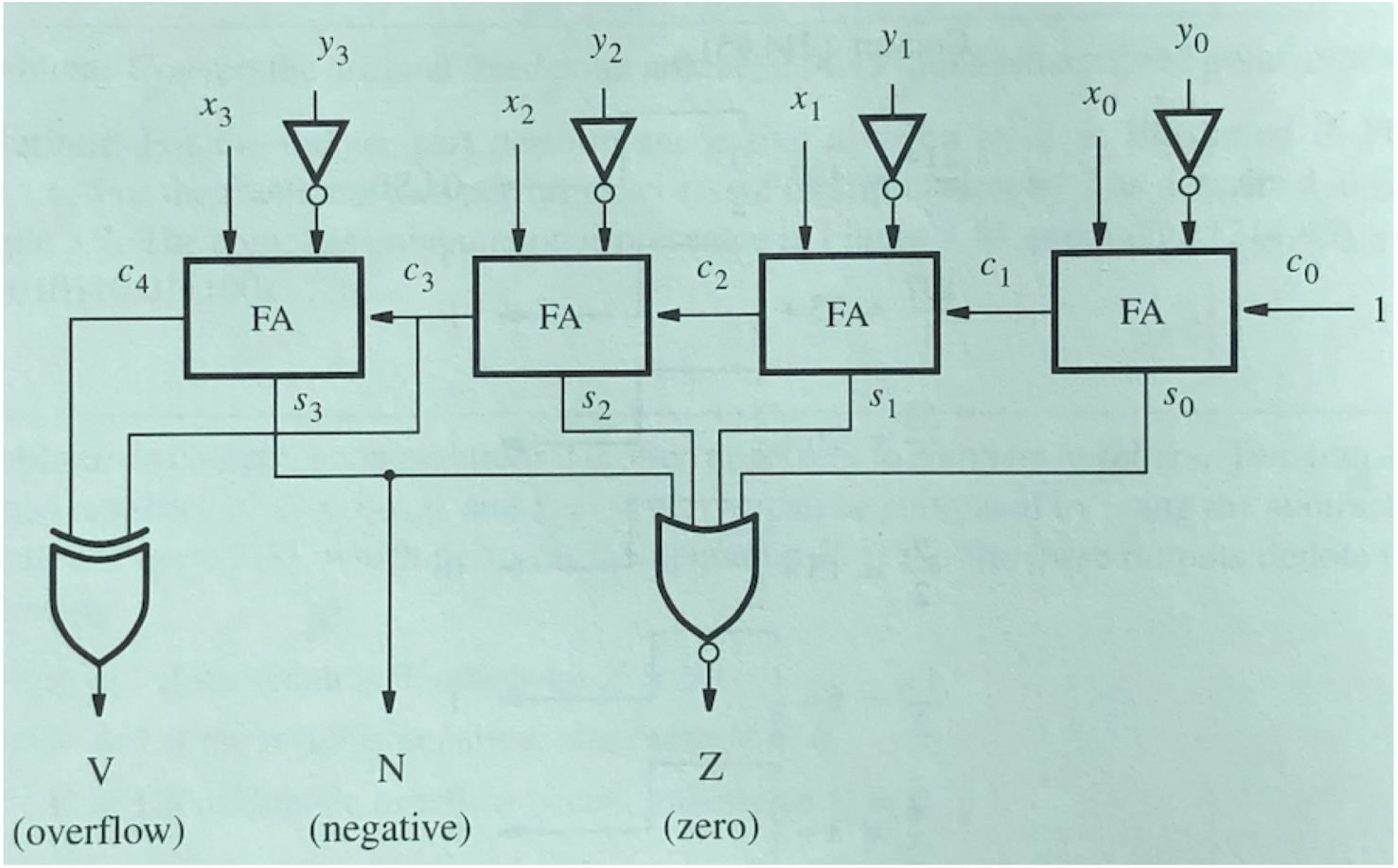
# Comparison of Signed Numbers

- **Equal** — **ZF**
- **Not equal** — $\overline{ZF}$
- **Greater** — $\overline{ZF} \bullet XNOR( NF, OF )$
- **Greater or Equal** — **XNOR( NF, OF )**
- **Less** — **XOR( NF, OF )**
- **Less or Equal** — **ZF + XOR( NF, OF )**

The i281 CPU does not implement these checks. Instead, it relies on the programmer/compiler to rearrange the code to use the other checks.

# Comparison examples with 4-bit signed numbers

# A four-bit comparator circuit

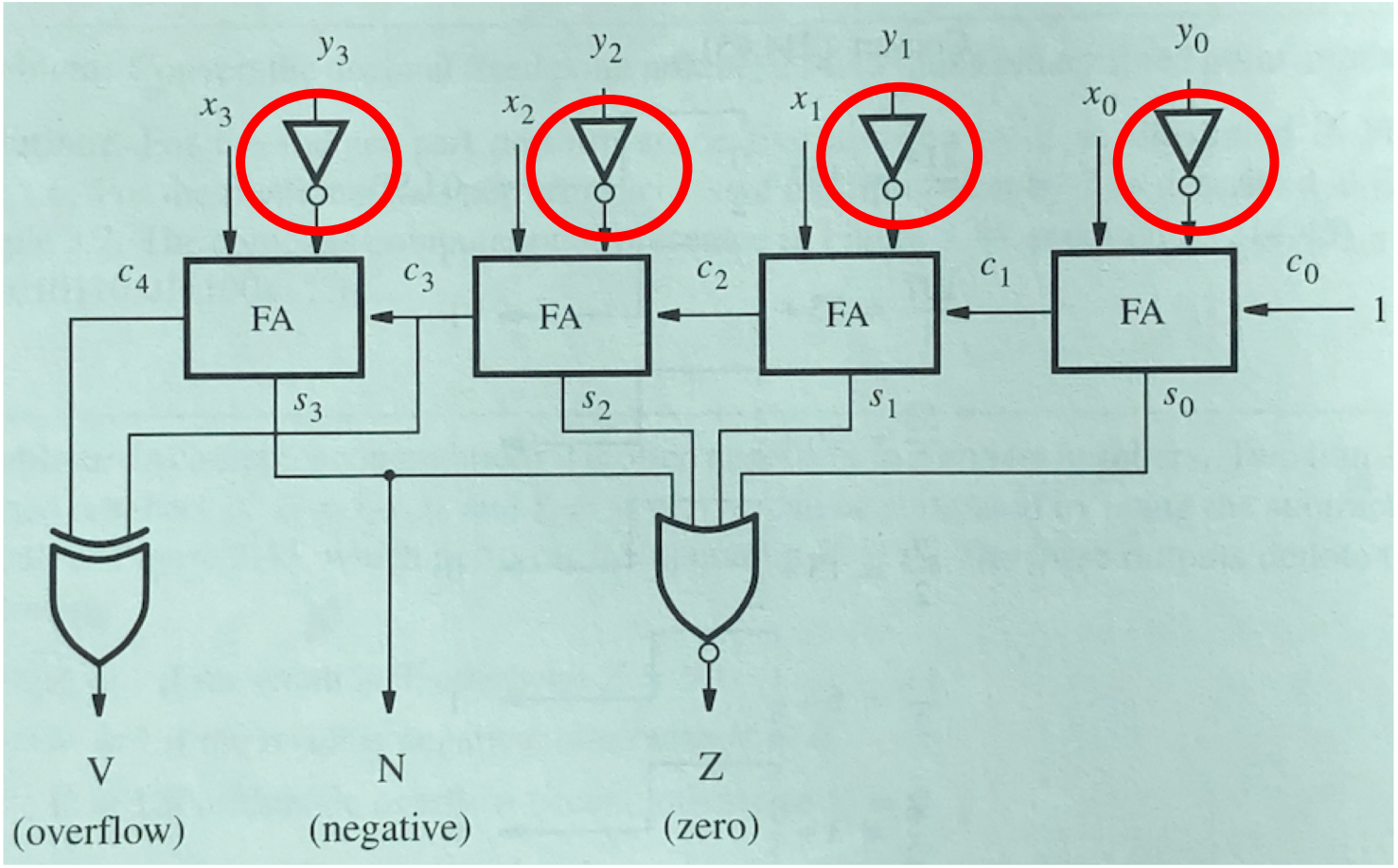



[ Figure 3.45 from the textbook ]

# A four-bit comparator circuit

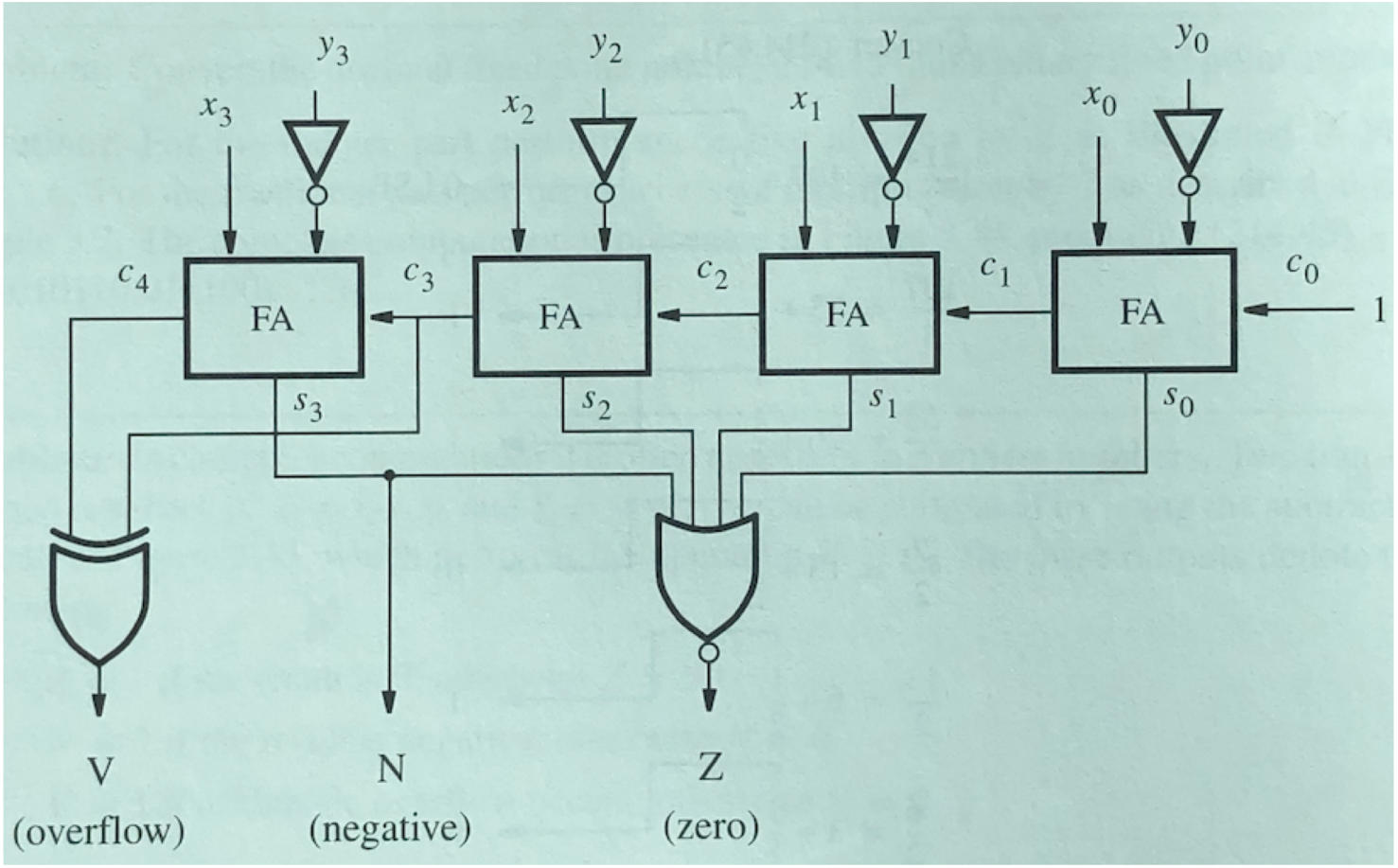
This circuit only subtracts.

That is why these are NOTs instead of XORs.




[ Figure 3.45 from the textbook ]

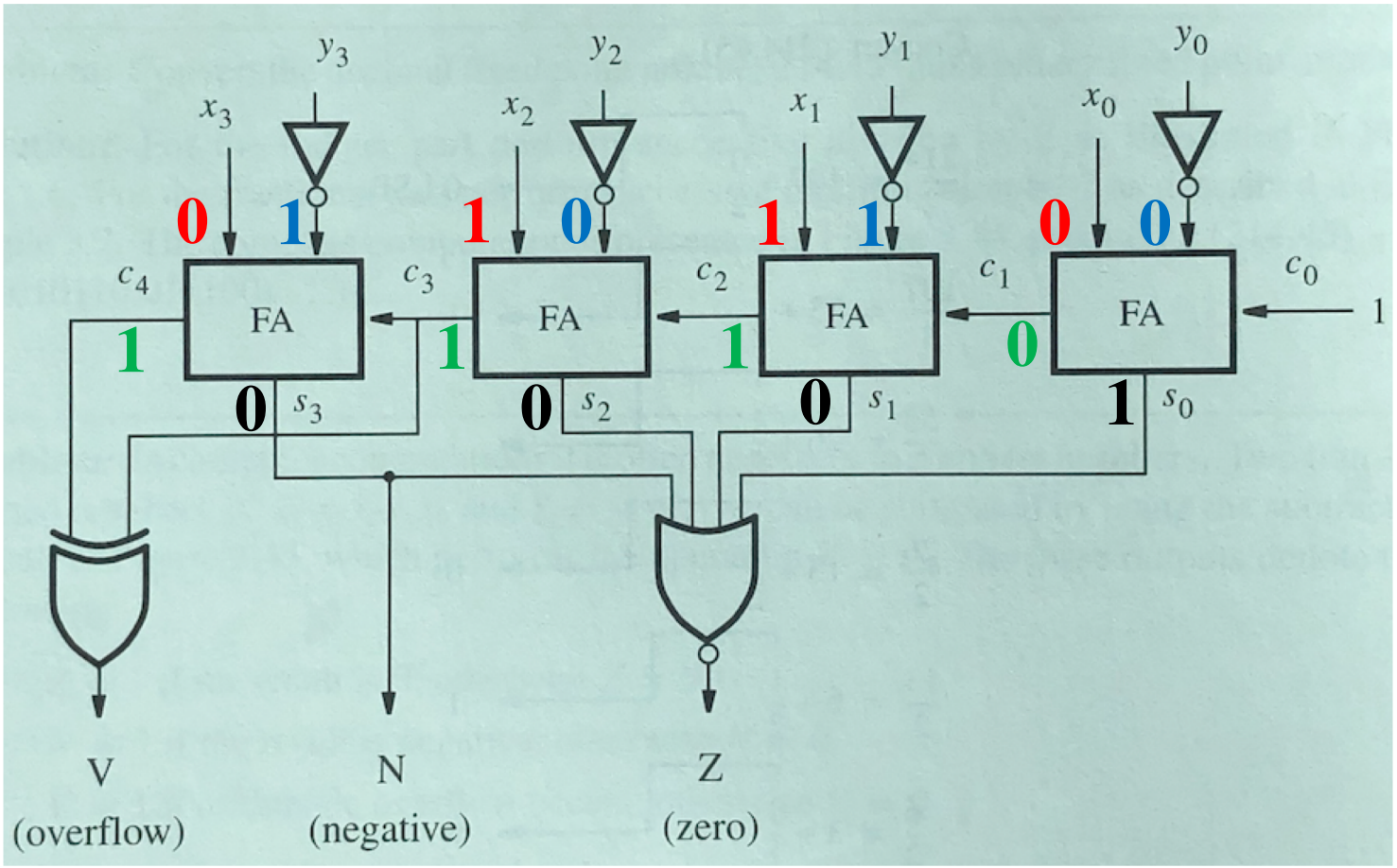
# A four-bit comparator circuit



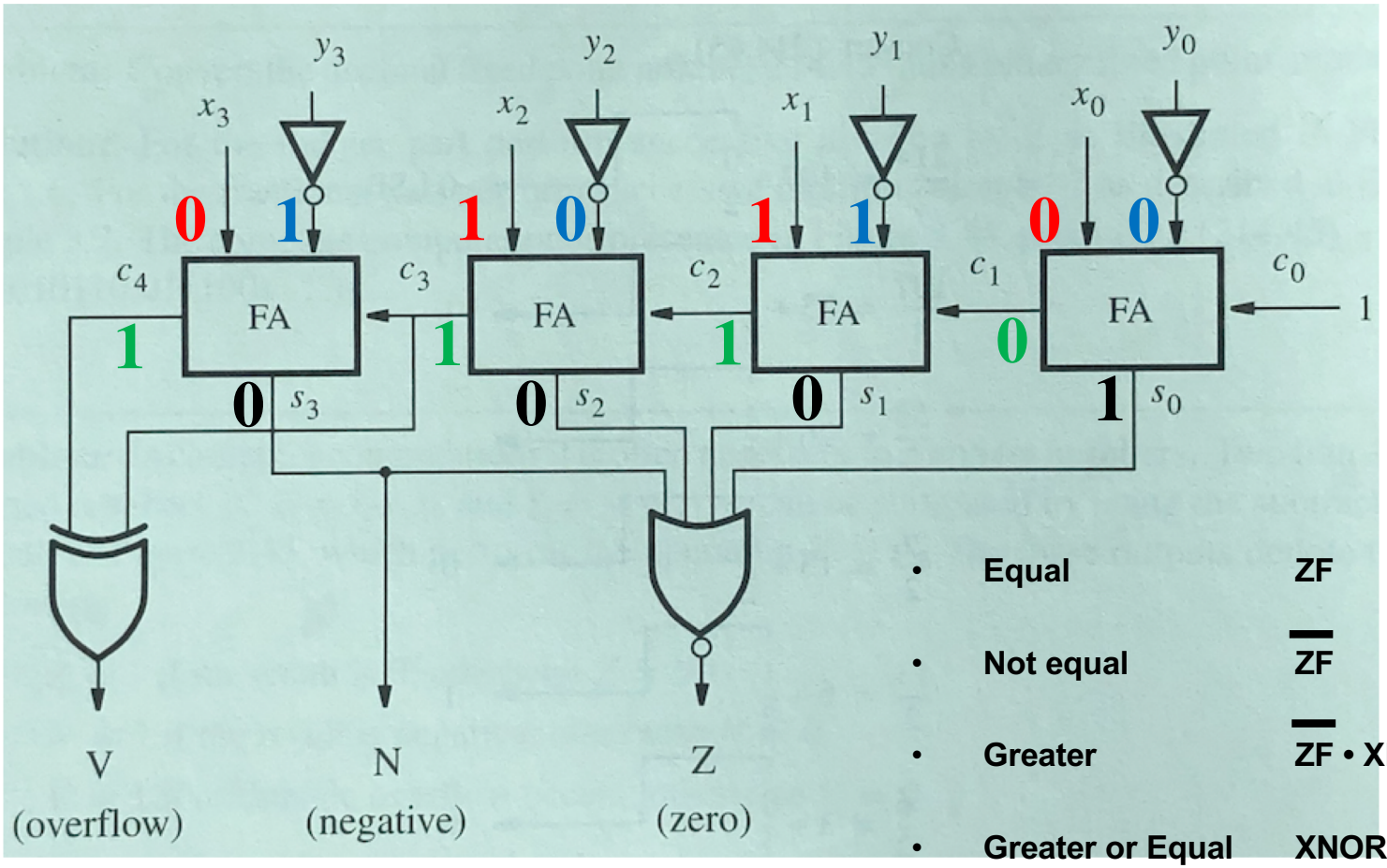

OF NF ZF

alternative names for the flags

# Compare 6 with 5: (+6) - (+5)

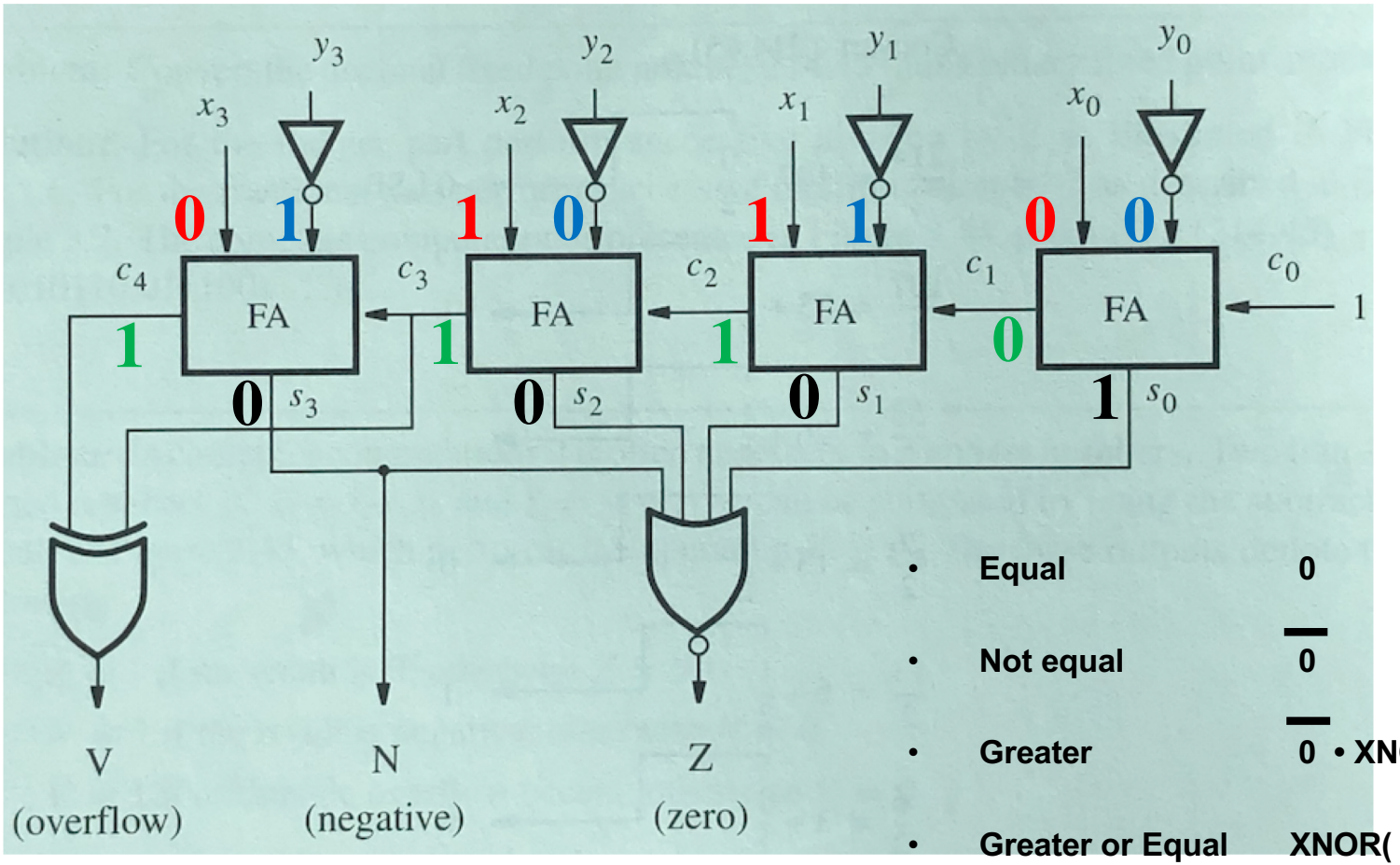


# Compare 6 with 5: (+6) - (+5)



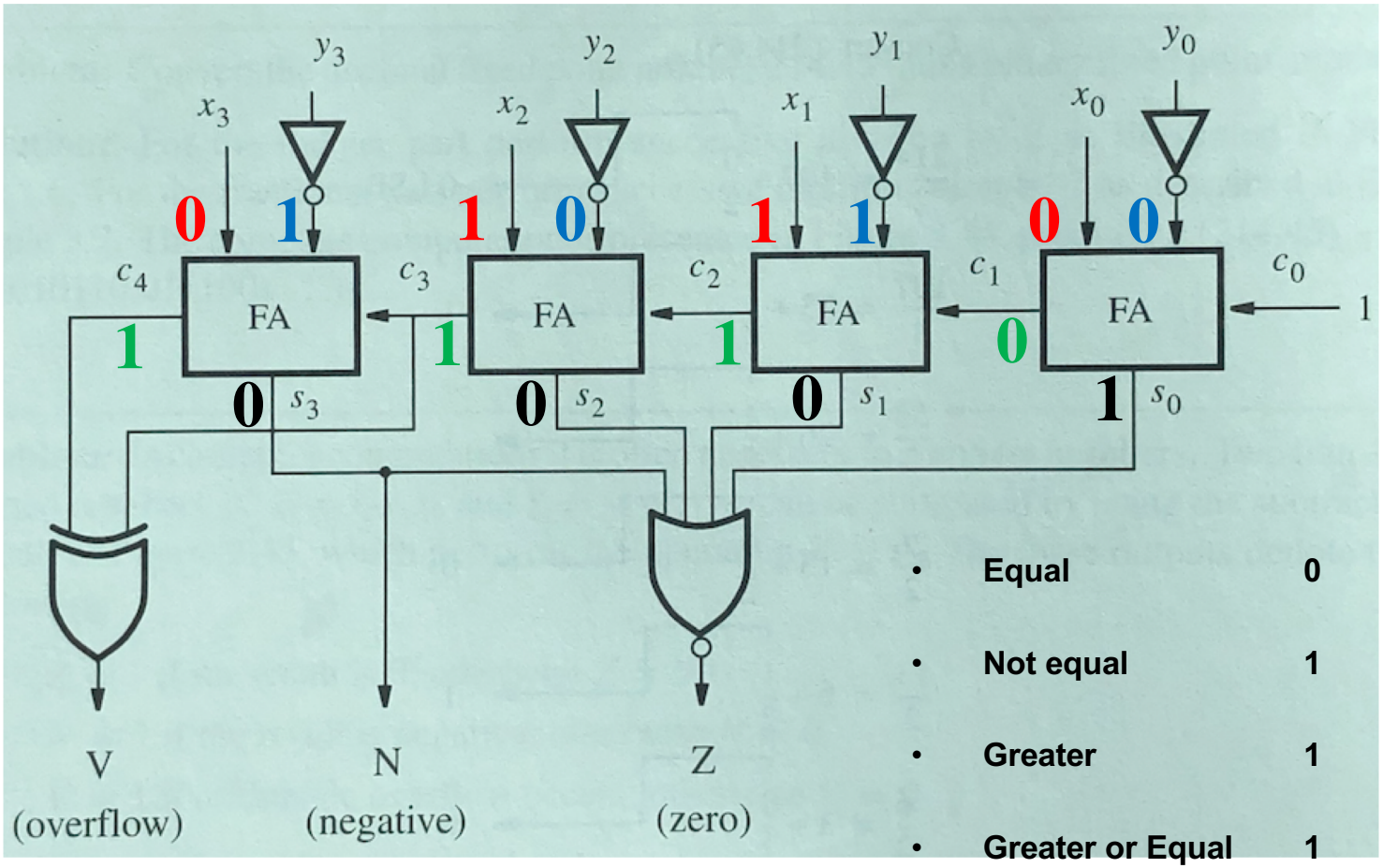

- Equal ZF
- Not equal $\overline{ZF}$
- Greater $\overline{ZF \bullet XNOR(NF, OF)}$
- Greater or Equal XNOR( NF, OF )
- Less XOR( NF, OF )
- Less or Equal ZF + XOR( NF, OF )

# Compare 6 with 5: (+6) - (+5)



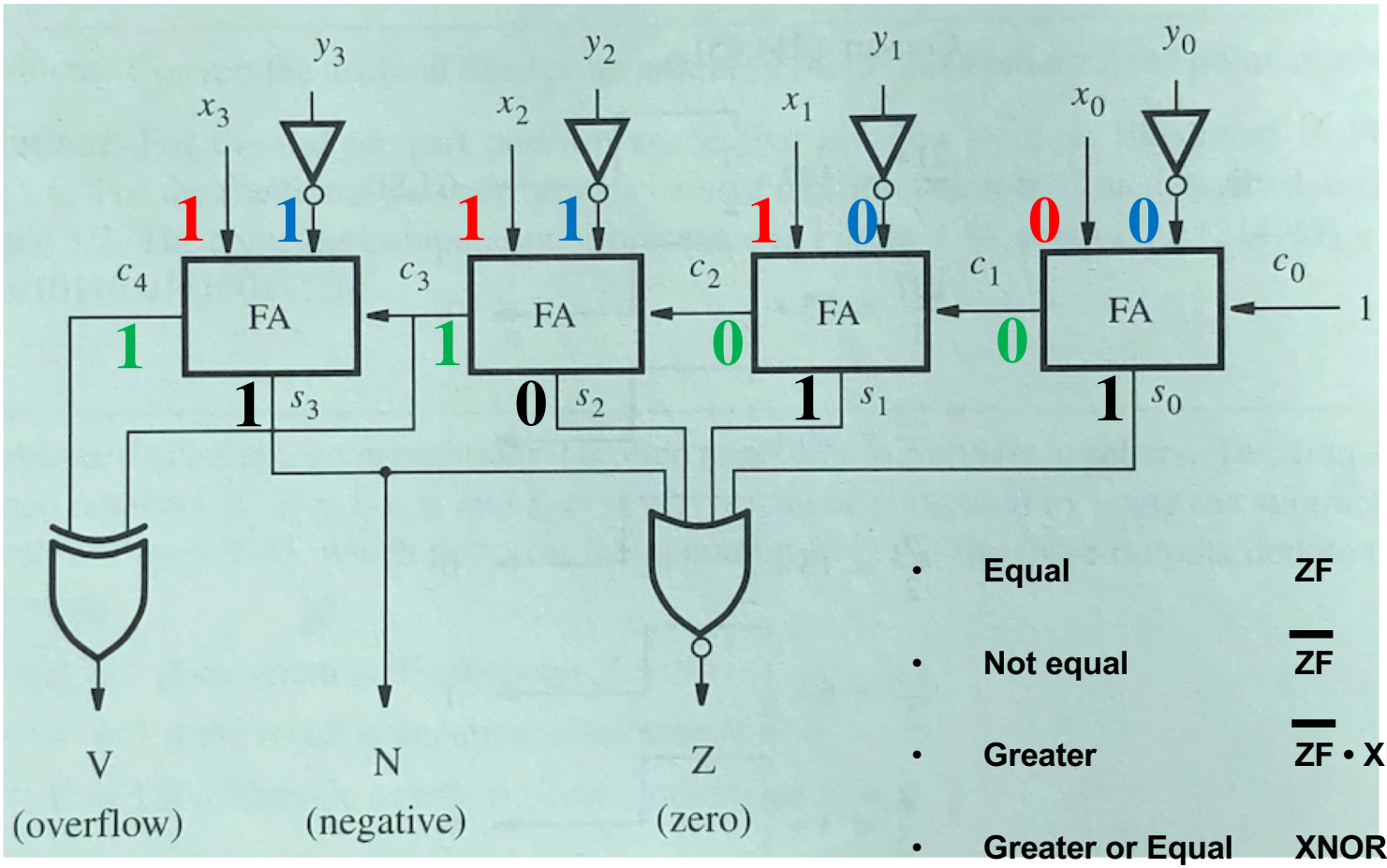

- Equal 0
- Not equal $\overline{0}$
- Greater $\overline{0}$ • XNOR( 0, 0 )
- Greater or Equal XNOR( 0, 0 )
- Less XOR( 0, 0 )
- Less or Equal 0 + XOR( 0, 0 )

# Compare 6 with 5: (+6) - (+5)



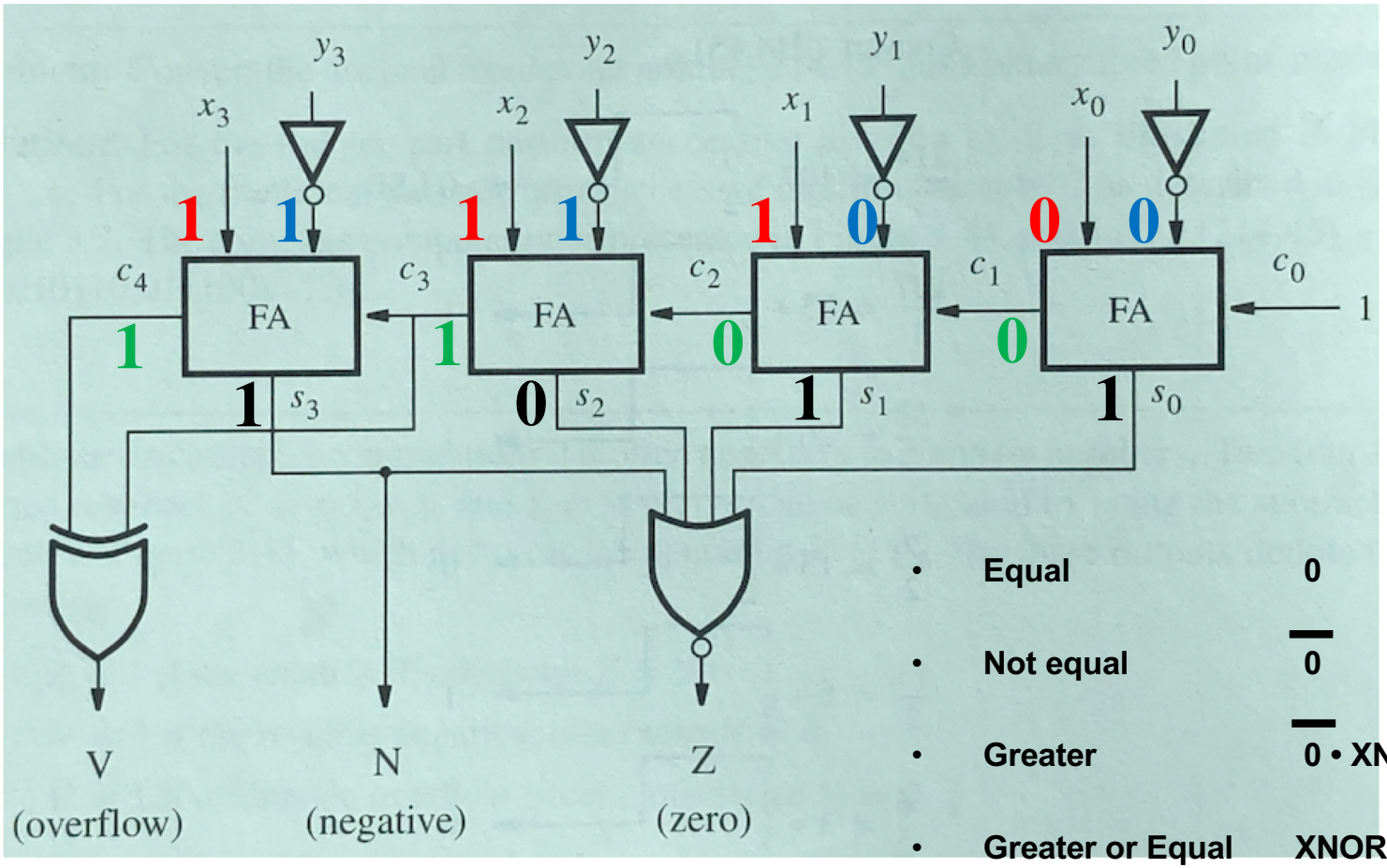

- Equal 0
- Not equal 1
- Greater 1
- Greater or Equal 1
- Less 0
- Less or Equal 0

# Compare negative 2 with 3: (-2) - (+3)



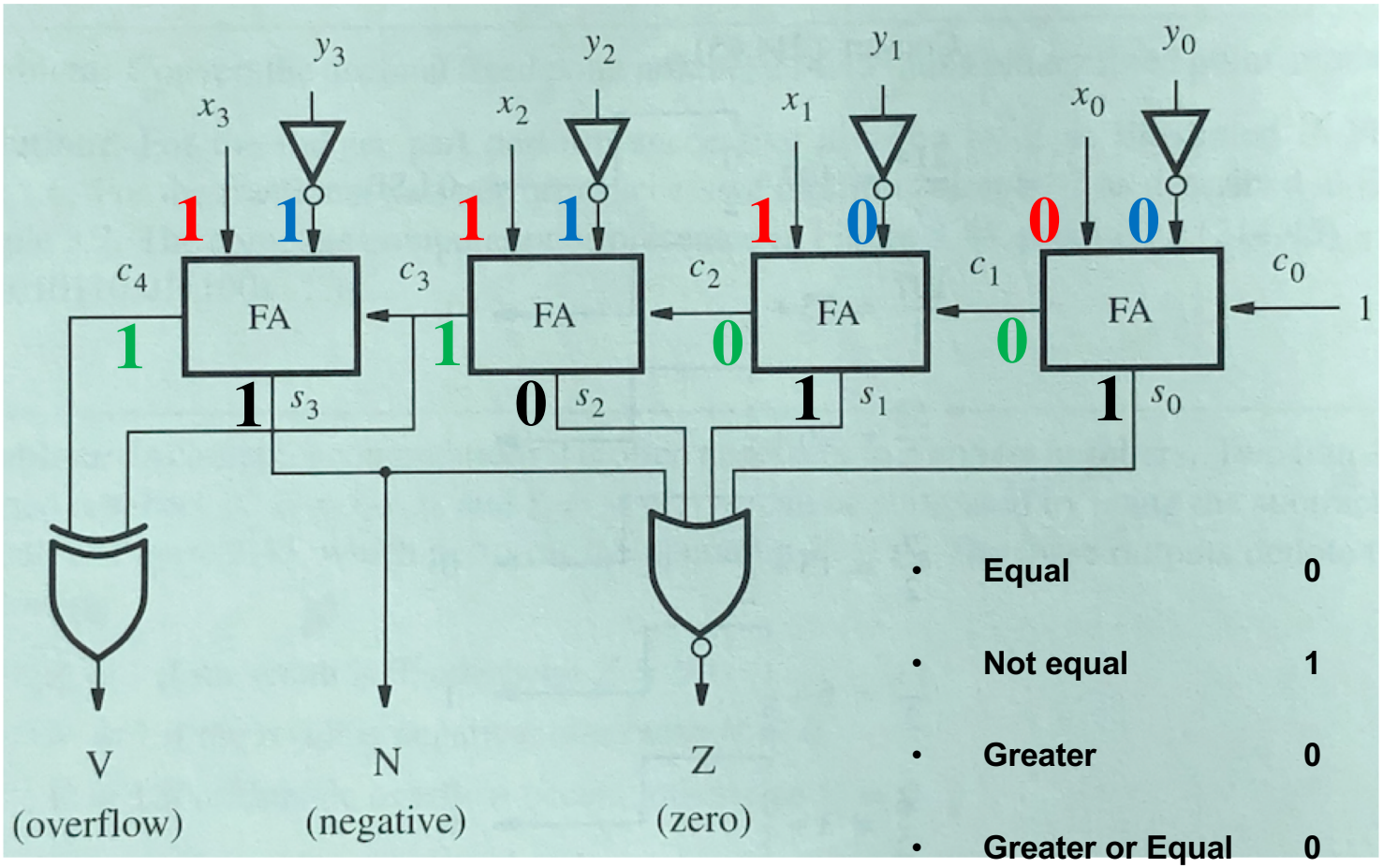

- Equal ZF
- Not equal $\overline{ZF}$
- Greater $\overline{ZF}$ • XNOR( NF, OF )
- Greater or Equal XNOR( NF, OF )
- Less XOR( NF, OF )
- Less or Equal ZF + XOR( NF, OF )

# Compare negative 2 with 3: (-2) - (+3)



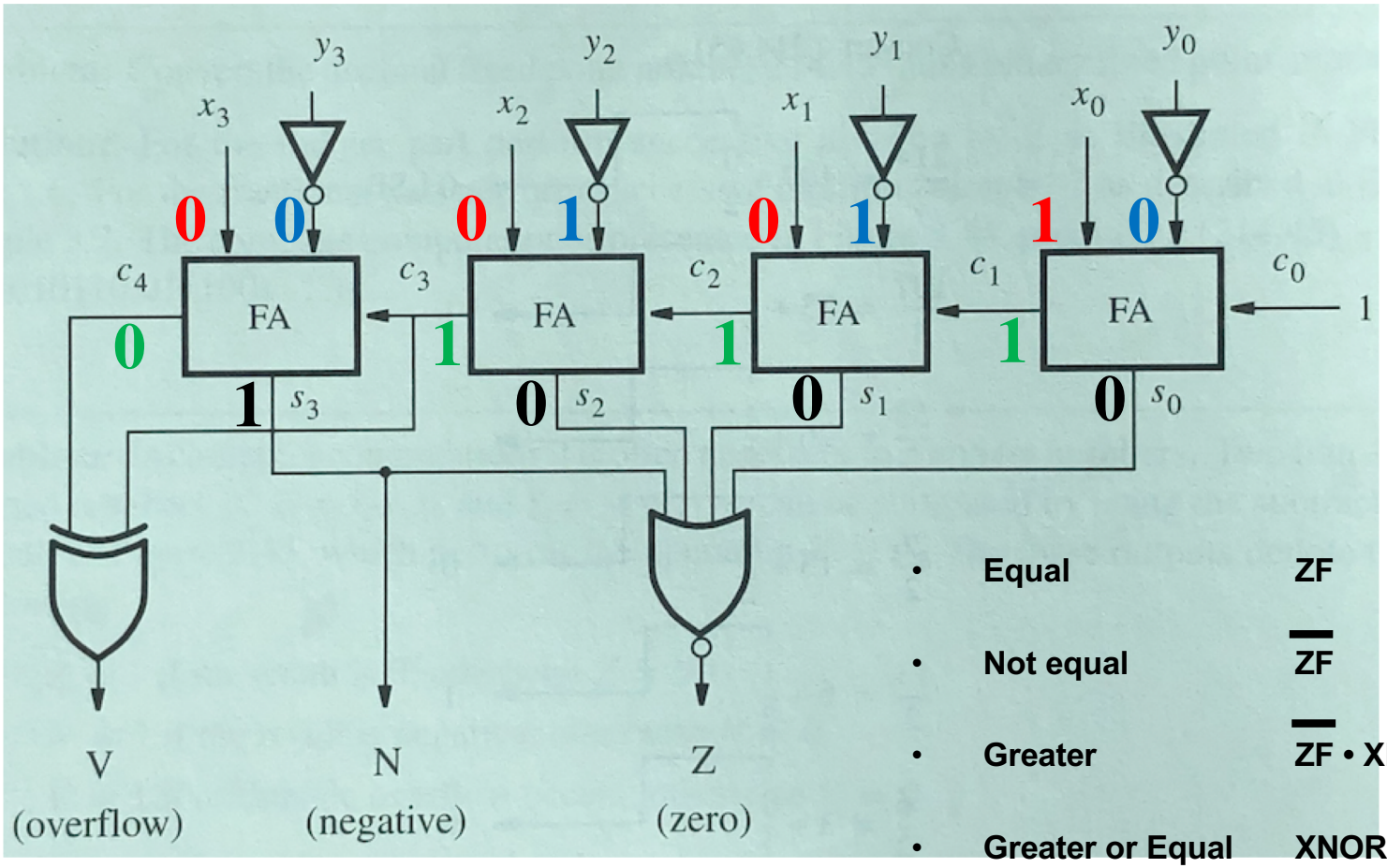

- Equal 0
- Not equal $\overline{0}$
- Greater $\overline{0}$ • XNOR( 1, 0 )
- Greater or Equal XNOR( 1, 0 )
- Less XOR( 1, 0 )
- Less or Equal 0 + XOR( 1, 0 )

# Compare negative 2 with 3: (-2) - (+3)



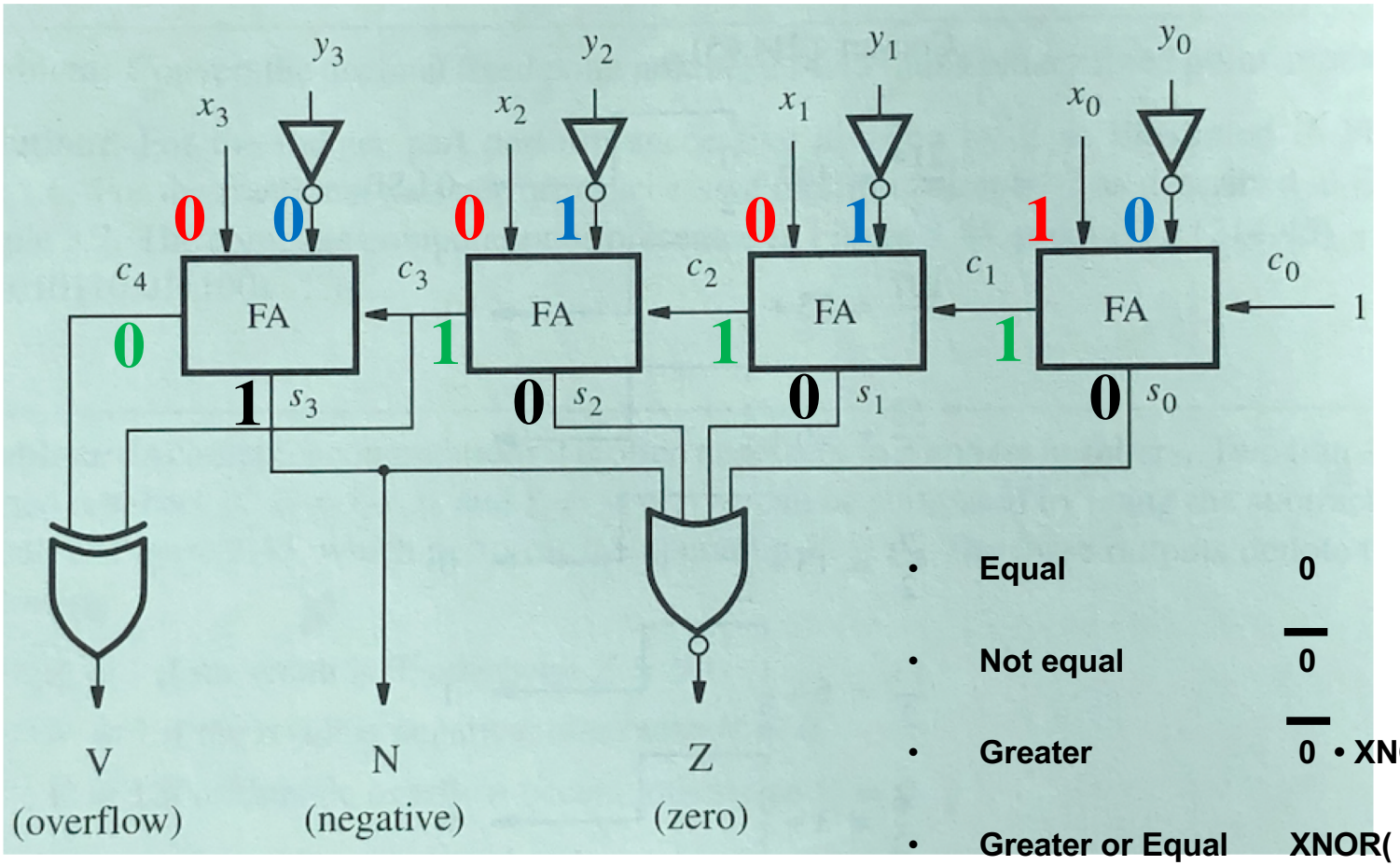

- Equal 0
- Not equal 1
- Greater 0
- Greater or Equal 0
- Less 1
- Less or Equal 1

# Compare 1 with negative 7: (+1) - (-7)



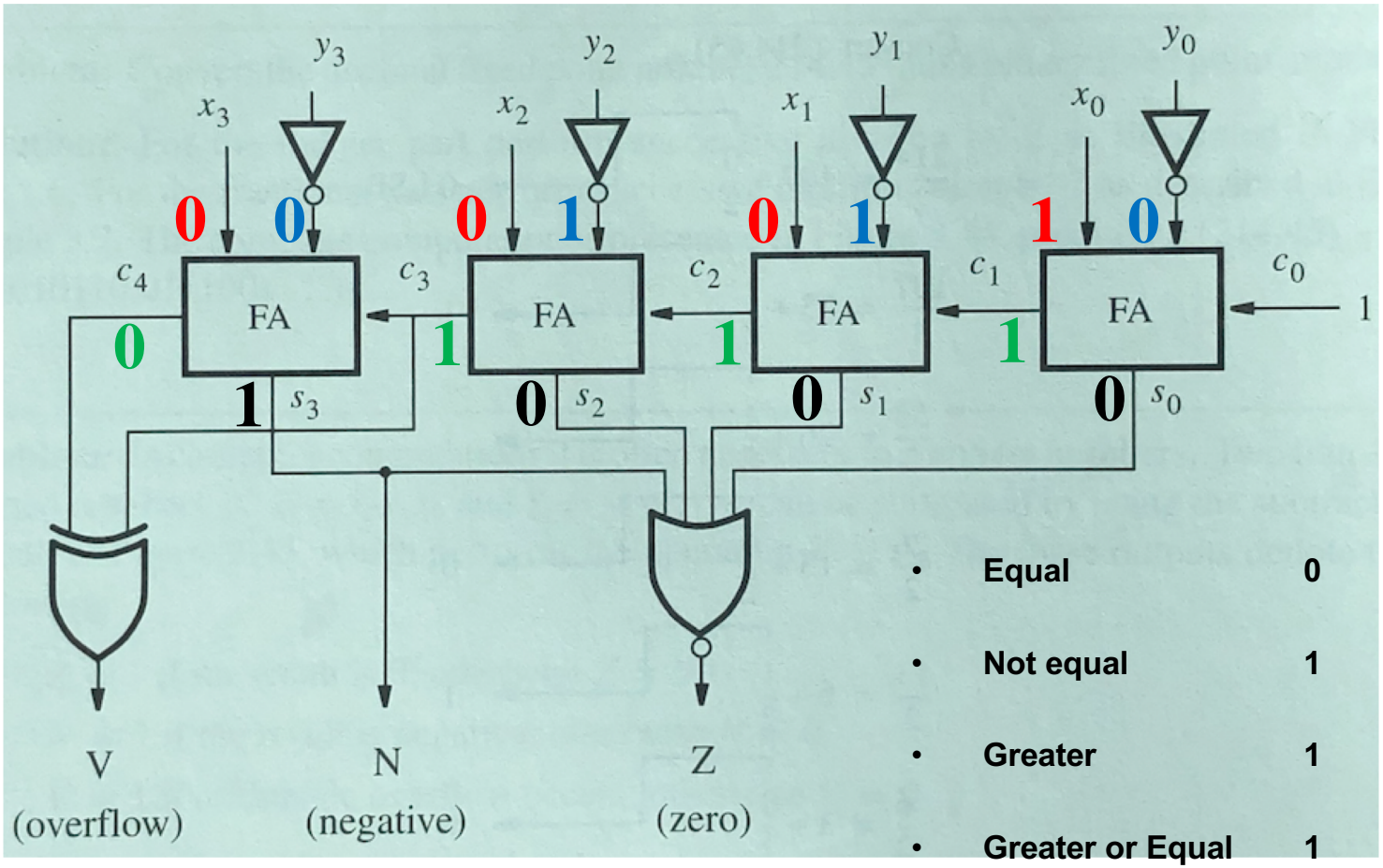

- Equal ZF
- Not equal $\overline{ZF}$
- Greater $\overline{ZF}$ • XNOR( NF, OF )
- Greater or Equal XNOR( NF, OF )
- Less XOR( NF, OF )
- Less or Equal ZF + XOR( NF, OF )

# Compare 1 with negative 7: (+1) - (-7)



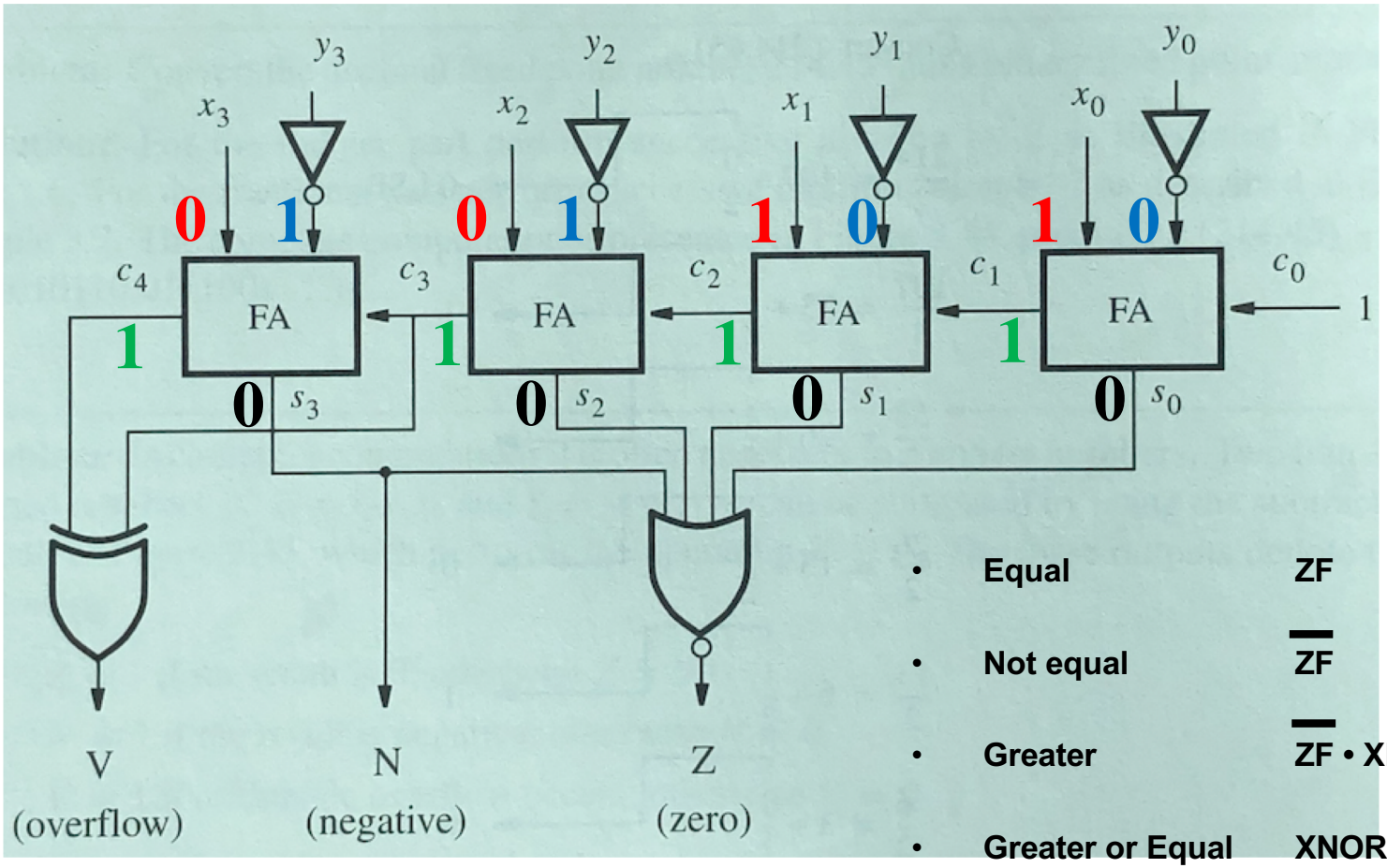

- Equal 0
- Not equal $\overline{0}$
- Greater $\overline{0}$ • XNOR( 1, 1 )
- Greater or Equal XNOR( 1, 1 )
- Less XOR( 1, 1 )
- Less or Equal 0 + XOR( 1, 1 )

# Compare 1 with negative 7: (+1) - (-7)



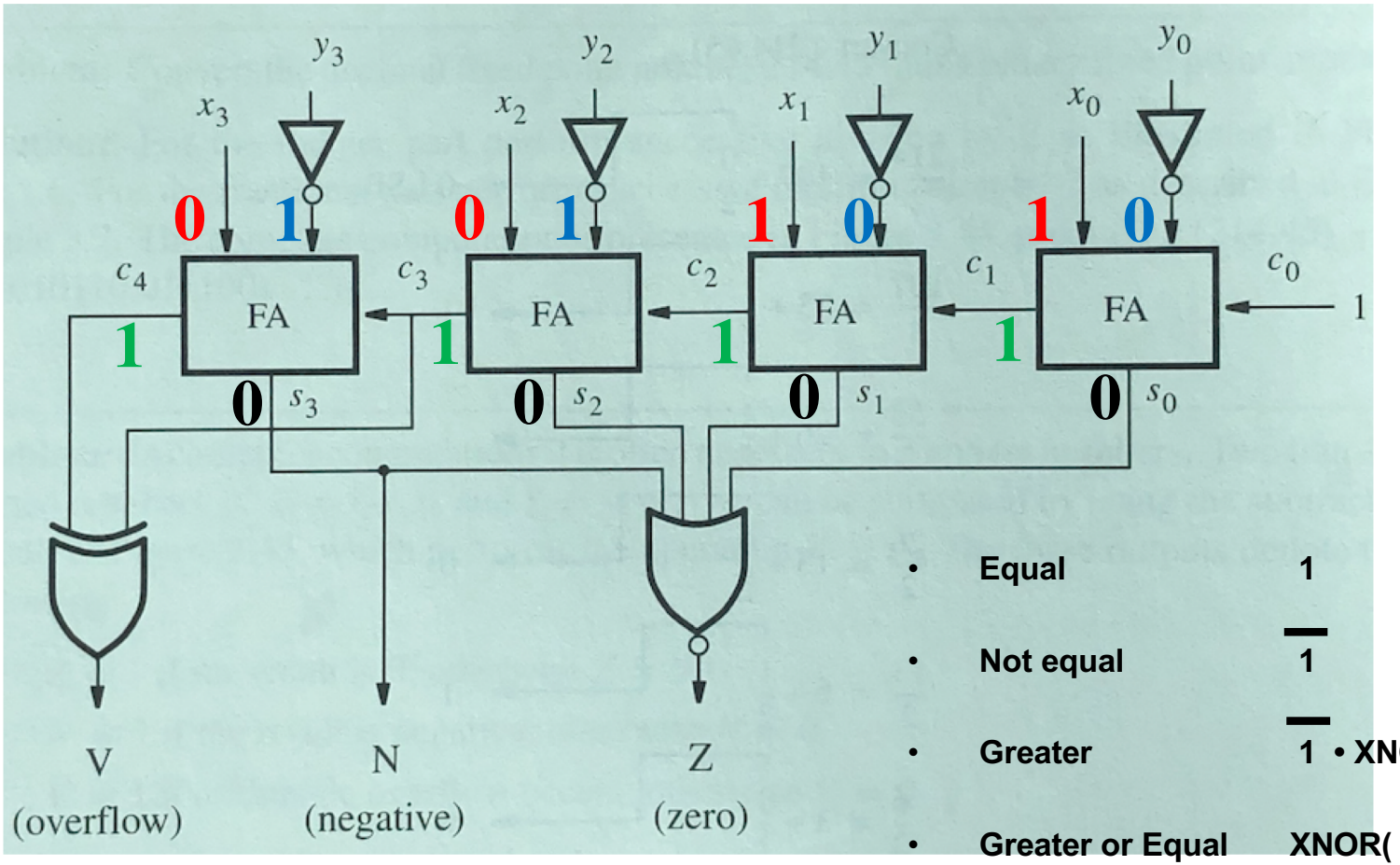

- Equal 0
- Not equal 1
- Greater 1
- Greater or Equal 1
- Less 0
- Less or Equal 0

# Compare 3 with 3: (+3) - (+3)



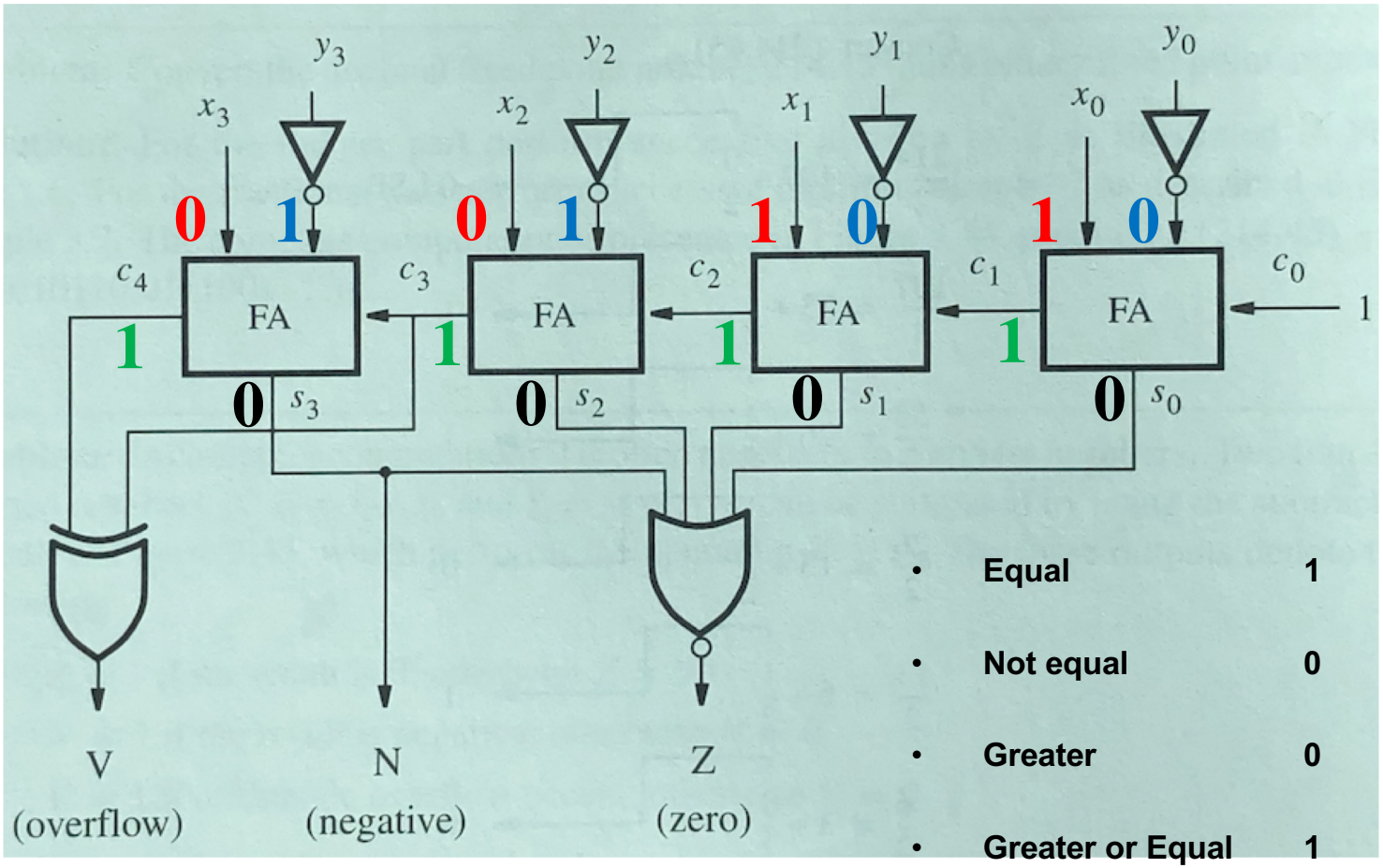

- Equal ZF
- Not equal $\overline{ZF}$
- Greater $\overline{ZF}$ • XNOR( NF, OF )
- Greater or Equal XNOR( NF, OF )
- Less XOR( NF, OF )
- Less or Equal ZF + XOR( NF, OF )

# Compare 3 with 3: (+3) - (+3)




- Equal 1
- Not equal $\overline{1}$
- Greater $\overline{1}$ • XNOR( 0, 0 )
- Greater or Equal XNOR( 0, 0 )
- Less XOR( 0, 0 )
- Less or Equal 1 + XOR( 0, 0 )

# Compare 3 with 3: (+3) - (+3)



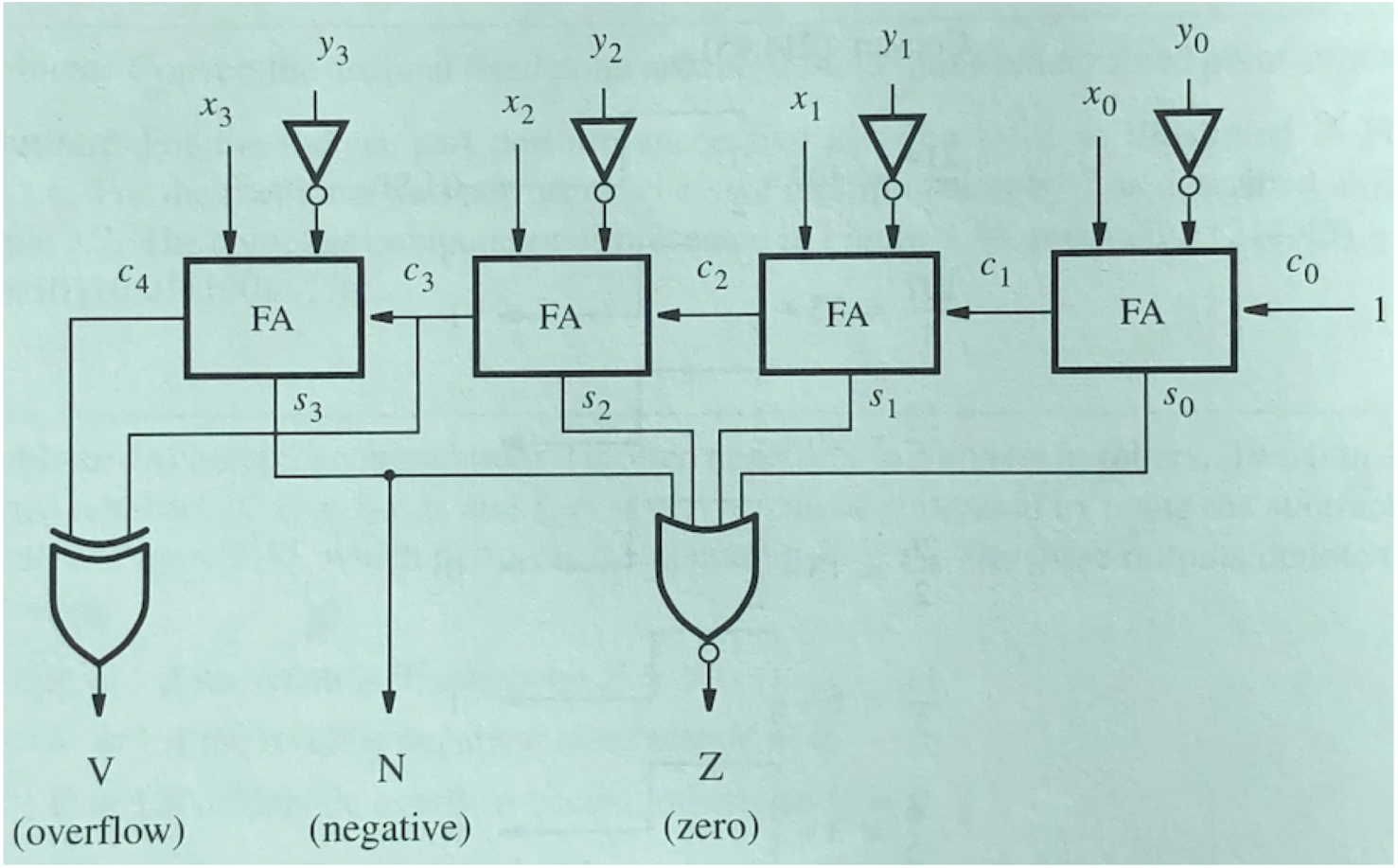

- Equal 1
- Not equal 0
- Greater 0
- Greater or Equal 1
- Less 0
- Less or Equal 1

# Comparison of Unsigned Numbers (not supported by the i281 CPU)

# Comparison of Unsigned Numbers

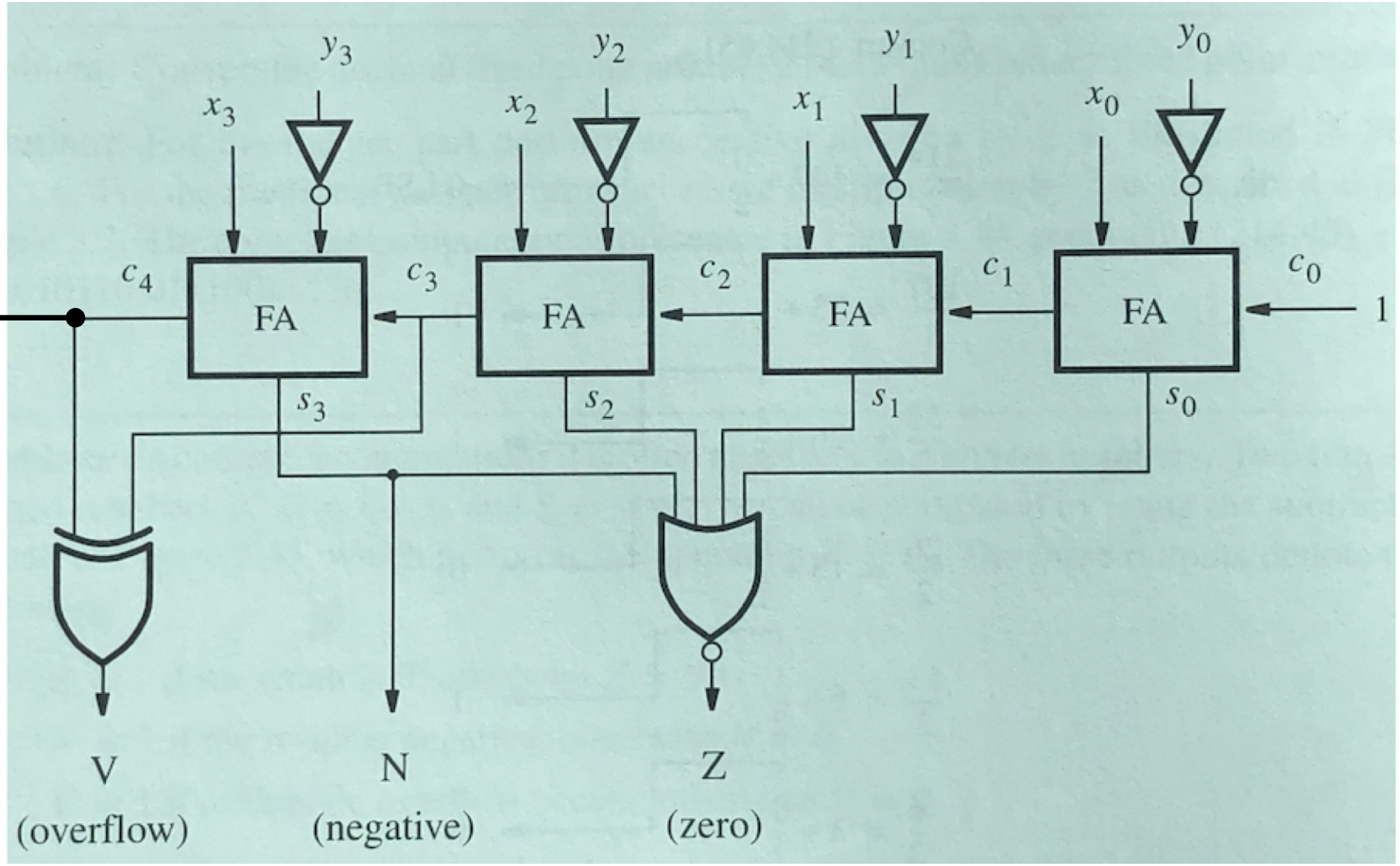
Also done with subtraction and then looking at the flags,
but the logic expressions are now different.

# A four-bit comparator circuit

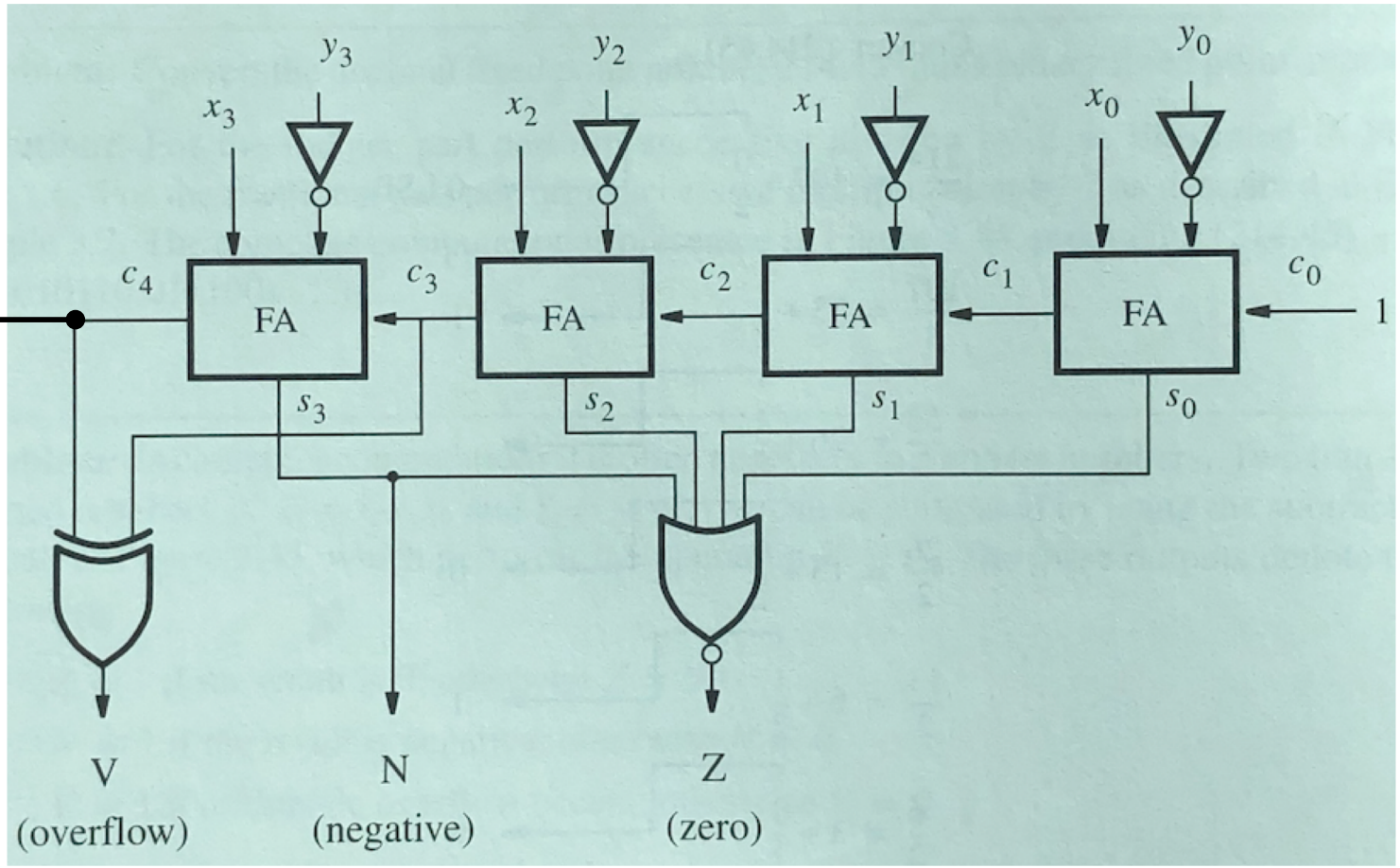



[ Figure 3.45 from the textbook ]

# A four-bit comparator circuit



# A four-bit comparator circuit




CF OF NF ZF

alternative names for the flags

# Comparison of Unsigned Numbers

- Equal
- Not equal
- Greater
- Greater or Equal
- Less
- Less or Equal

# Comparison of Unsigned Numbers

- Equal
- Not equal
- Greater / Above
- Greater or Equal  / Above or Equal
- Less / Below
- Less or Equal / Below or Equal

# Comparison of Unsigned Numbers

- Equal — ZF = 1
- Not equal — ZF = 0
- Greater — ZF = 0 and CF = 1
- Greater or Equal — CF = 1
- Less — CF = 0
- Less or Equal — ZF = 1 or CF = 0

# Comparison of Unsigned Numbers

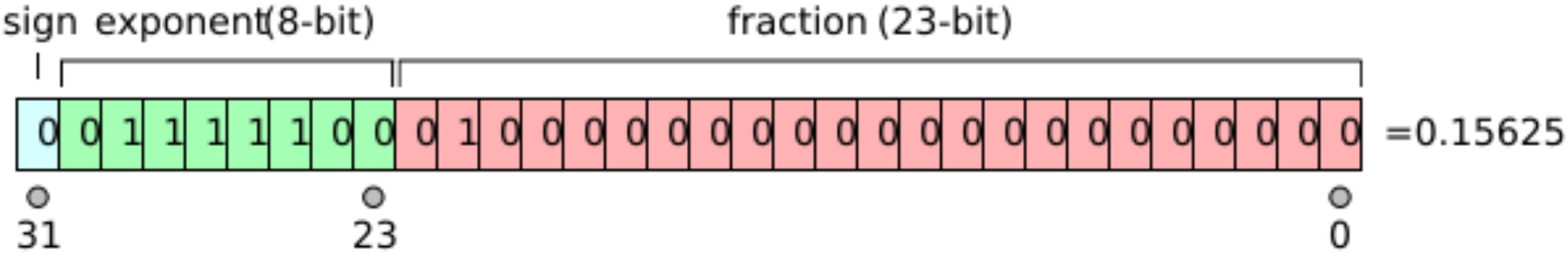
- Equal — $ZF$
- Not equal — $\overline{ZF}$
- Greater — $\overline{ZF} \cdot CF$
- Greater or Equal — $CF$
- Less — $\overline{CF}$
- Less or Equal — $ZF + \overline{CF}$

# Comparison of Unsigned Numbers

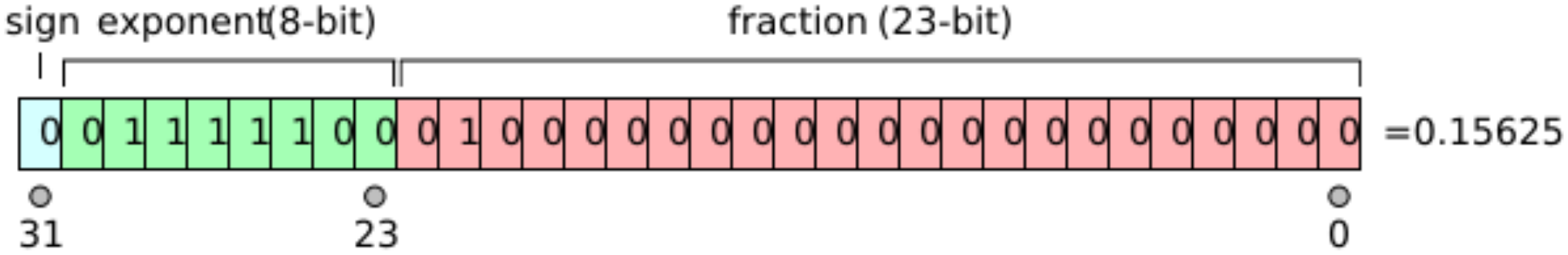
- Equal — $ZF$
- Not equal — $\overline{ZF}$
- Above — $\overline{ZF} \cdot CF$
- Above or Equal — $CF$
- Below — $\overline{CF}$
- Below or Equal — $ZF + \overline{CF}$

# Floating Point Numbers

# The story with floats is more complicated
# IEEE 754-1985 Standard




[http://en.wikipedia.org/wiki/IEEE_754]




$$v = (-1)^{sign} \times 2^{exponent - exponent\ bias} \times 1.fraction$$

s = +1 (positive numbers and +0) when the sign bit is 0

s = −1 (negative numbers and −0) when the sign bit is 1

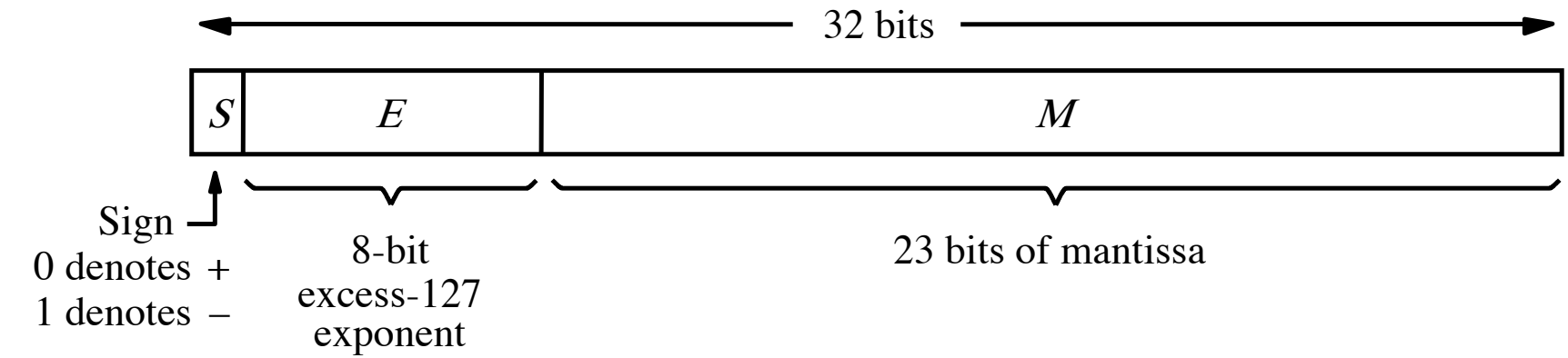
e = *exponent* − 127 (in other words the exponent is stored with 127 added to it, also called "biased with 127")

**In the example shown above, the *sign* bit is zero, the *exponent* is 124, and the significand is 1.01 (in binary, which is 1.25 in decimal). The represented number is**

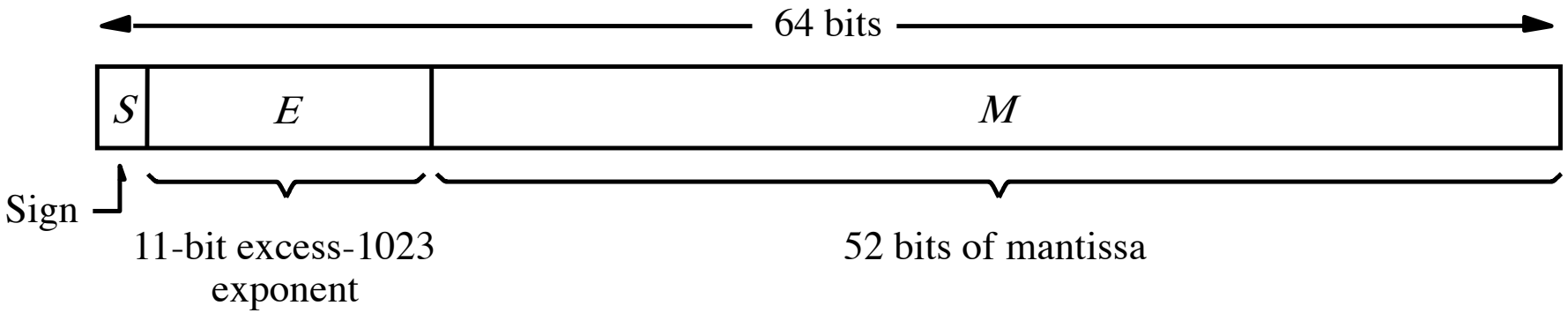
$$(-1)^0 \times 2^{(124 - 127)} \times 1.25 = +0.15625.$$

[http://en.wikipedia.org/wiki/IEEE_754]

# Float (32-bit) vs. Double (64-bit)




(a) Single precision




(b) Double precision

[Figure 3.37 from the textbook]

# On-line IEEE 754 Converter

- https://www.h-schmidt.net/FloatConverter/IEEE754.html

# Representing 2.0

sign=+1 exp=1 mantisse=1.0

| | |
|---|---|
| Binary representation | 01000000000000000000000000000000 |
| Hexadecimal representation | 40000000 |
| Decimal representation | 2.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 2.0

sign=+1 exp=1 mantisse=1.0

| | |
|---|---|
| Binary representation | 01000000000000000000000000000000 |
| Hexadecimal representation | 40000000 |
| Decimal representation | 2.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 2.0

sign=+1 exp=1 mantisse=1.0


| | |
|---|---|
| Binary representation | 01000000000000000000000000000000 |
| Hexadecimal representation | 40000000 |
| Decimal representation | 2.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 2.0

sign=+1 exp=1 mantisse=1.0

$(-1)^{0}$ $2^{(128-127)}$ $(1 + 0)$

| | |
|---|---|
| Binary representation | 01000000000000000000000000000000 |
| Hexadecimal representation | 40000000 |
| Decimal representation | 2.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 2.0

sign=+1 exp=1 mantisse=1.0

$(-1)^{0} \times 2^{(128-127)} \times (1 + 0) = 2.0$

| | |
|---|---|
| Binary representation | 01000000000000000000000000000000 |
| Hexadecimal representation | 40000000 |
| Decimal representation | 2.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 2.0

sign=+1 exp=1 mantisse=1.0

$1 \times 2^1 \times 1.0 = 2.0$

| | |
|---|---|
| Binary representation | 01000000000000000000000000000000 |
| Hexadecimal representation | 40000000 |
| Decimal representation | 2.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 4.0

sign=+1 exp=2 mantisse=1.0

| | |
|---|---|
| Binary representation | 01000000100000000000000000000000 |
| Hexadecimal representation | 40800000 |
| Decimal representation | 4.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 4.0

sign=+1 exp=2 mantisse=1.0

| | |
|---|---|
| Binary representation | 01000000100000000000000000000000 |
| Hexadecimal representation | 40800000 |
| Decimal representation | 4.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 4.0


[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 4.0

sign=+1 exp=2 mantisse=1.0

$(-1)^{0} \times 2^{(129-127)} \times (1 + 0) = 4.0$

| | |
|---|---|
| Binary representation | 01000000100000000000000000000000 |
| Hexadecimal representation | 40800000 |
| Decimal representation | 4.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 4.0

sign=+1 exp=2 mantisse=1.0

$1 \times 2^2 \times 1.0 = 4.0$

| | |
|---|---|
| Binary representation | 01000000100000000000000000000000 |
| Hexadecimal representation | 40800000 |
| Decimal representation | 4.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 8.0

sign=+1 exp=3 mantisse=1.0

| | |
|---|---|
| Binary representation | 01000001000000000000000000000000 |
| Hexadecimal representation | 41000000 |
| Decimal representation | 8.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 16.0

sign=+1 exp=4 mantisse=1.0

| | |
|---|---|
| Binary representation | 01000001100000000000000000000000 |
| Hexadecimal representation | 41800000 |
| Decimal representation | 16.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing -16.0

sign=-1 exp=4 mantisse=1.0

| | |
|---|---|
| Binary representation | 11000001100000000000000000000000 |
| Hexadecimal representation | C1800000 |
| Decimal representation | -16.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 1.0

sign=+1 exp=0 mantisse=1.0

| | |
|---|---|
| Binary representation | 00111111100000000000000000000000 |
| Hexadecimal representation | 3F800000 |
| Decimal representation | 1.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 3.0

sign=+1 exp=1 mantisse=1.5

| | |
|---|---|
| Binary representation | 01000000010000000000000000000000 |
| Hexadecimal representation | 40400000 |
| Decimal representation | 3.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 3.0

sign=+1 exp=1 mantisse=1.5

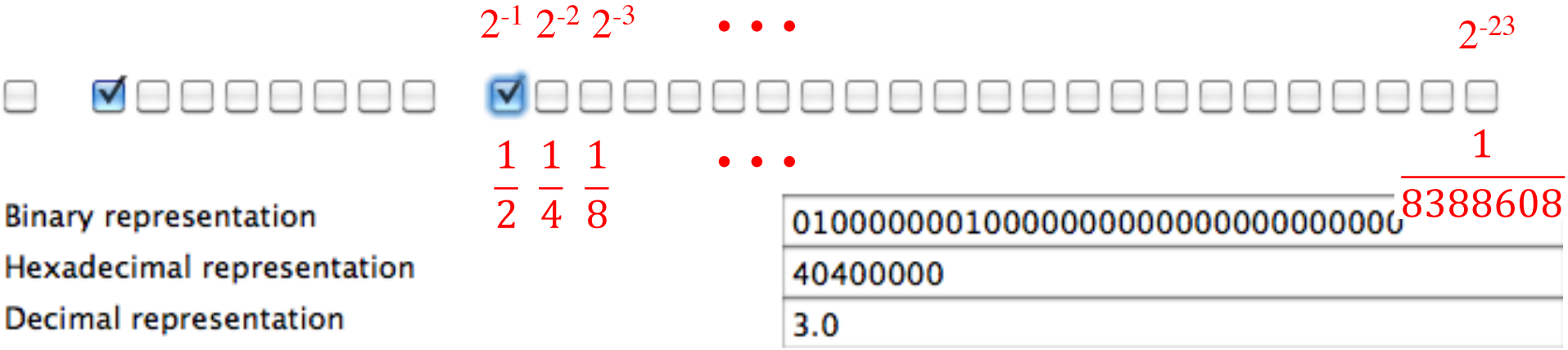
$2^{-1}$ $2^{-2}$ $2^{-3}$ • • • $2^{-23}$

| | |
|---|---|
| Binary representation | 01000000010000000000000000000000 |
| Hexadecimal representation | 40400000 |
| Decimal representation | 3.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

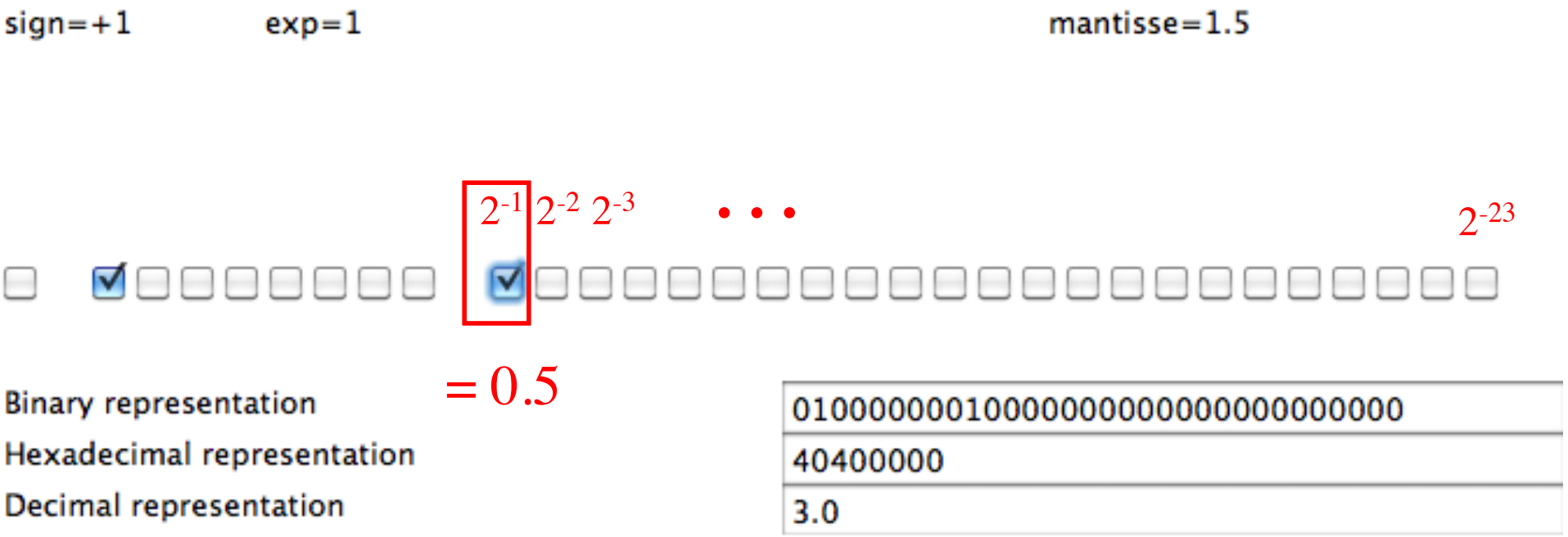
# Representing 3.0

sign=+1 exp=1 mantisse=1.5




[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 3.0




[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 3.0


[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 3.0

sign=+1 exp=1 mantisse=1.5

$(-1)^{0} \times 2^{(128-127)} \times (1 + 0.5) = 3.0$

| | |
|---|---|
| Binary representation | 01000000010000000000000000000000 |
| Hexadecimal representation | 40400000 |
| Decimal representation | 3.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 3.0

This 1 is given to you for free.
It is part of the formula/standard.

sign=+1 exp=1 mantisse=1.5

$(-1)^{0} \times 2^{(128-127)} \times (1 + 0.5) = 3.0$

| | |
|---|---|
| Binary representation | 01000000100000000000000000000000 |
| Hexadecimal representation | 40400000 |
| Decimal representation | 3.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 3.0

sign=+1 exp=1 mantisse=1.5

$1 \times 2^1 \times 1.5 = 3.0$

| | |
|---|---|
| Binary representation | 01000000010000000000000000000000 |
| Hexadecimal representation | 40400000 |
| Decimal representation | 3.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 3.5

sign=+1 exp=1 mantisse=1.75

| | |
|---|---|
| Binary representation | 01000000011000000000000000000000 |
| Hexadecimal representation | 40600000 |
| Decimal representation | 3.5 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 3.5

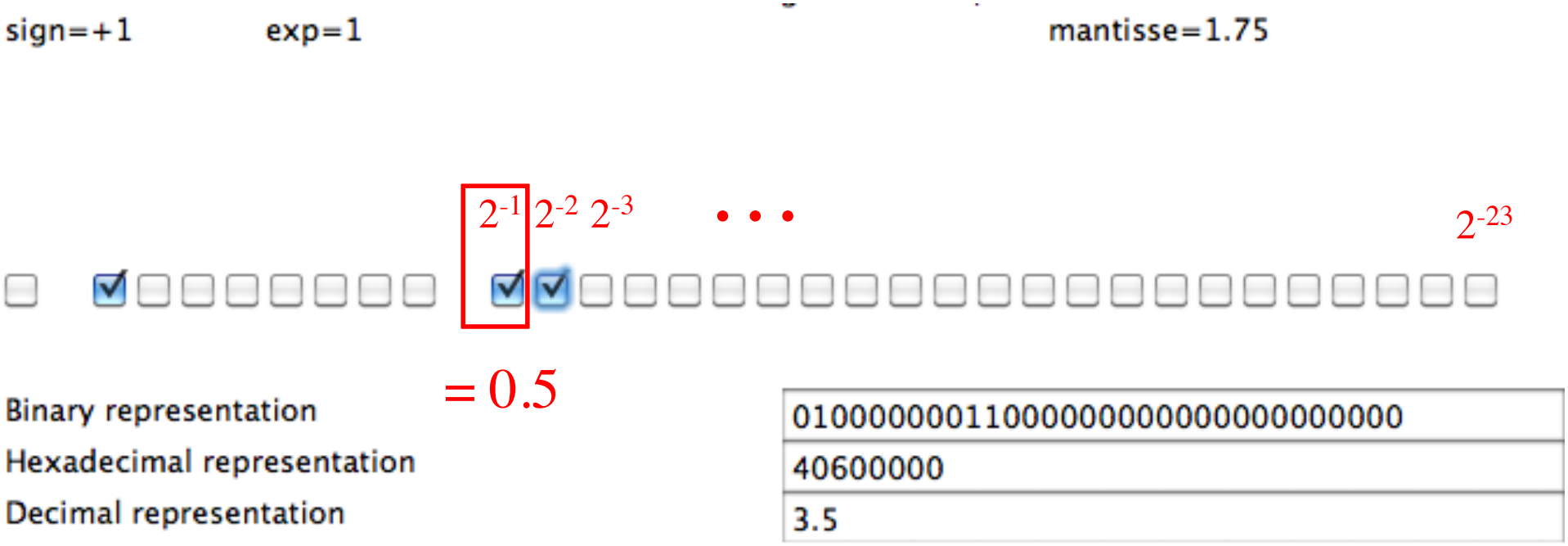
sign=+1 exp=1 mantisse=1.75

$2^{-1}$ $2^{-2}$ $2^{-3}$ • • • $2^{-23}$

| | |
|---|---|
| Binary representation | 01000000011000000000000000000000 |
| Hexadecimal representation | 40600000 |
| Decimal representation | 3.5 |

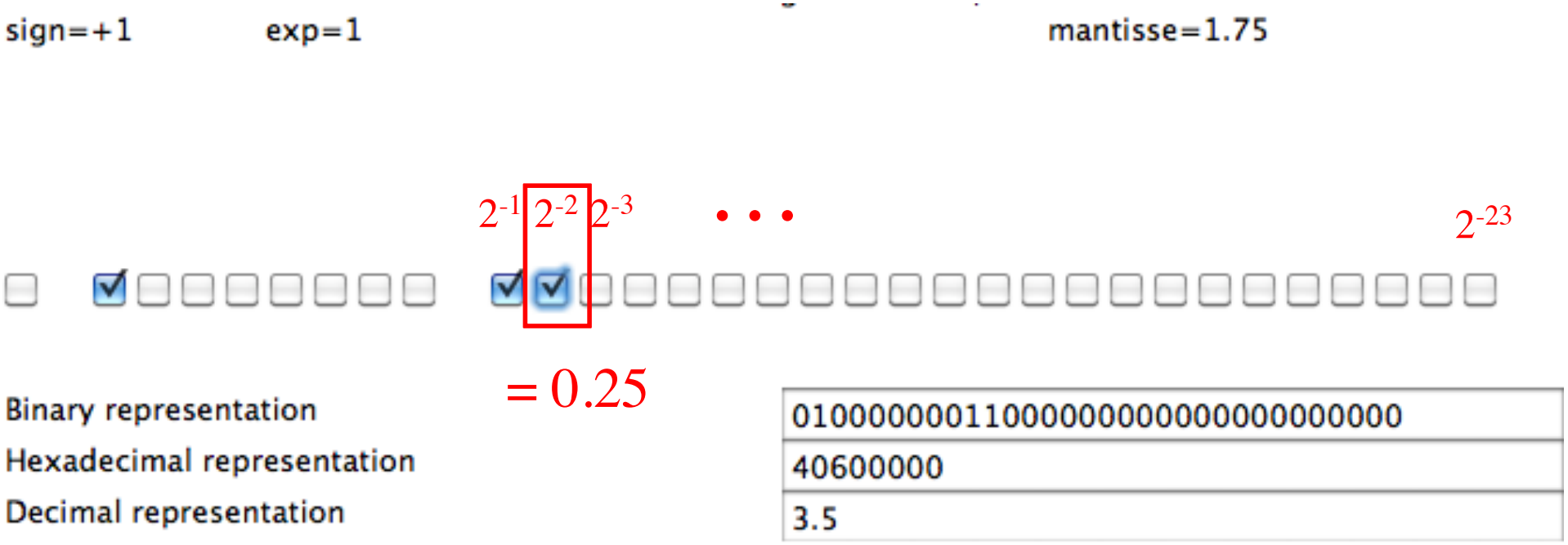
[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 3.5




[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 3.5




[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 3.5

sign=+1 exp=1 mantisse=1.75


| | |
|---|---|
| Binary representation | 01000000011000000000000000000000 |
| Hexadecimal representation | 40600000 |
| Decimal representation | 3.5 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 3.5

sign=+1 exp=1 mantisse=1.75

$(-1)^{0} \times 2^{(128-127)} \times (1 + 0.5 + 0.25) = 3.5$

| | |
|---|---|
| Binary representation | 01000000011000000000000000000000 |
| Hexadecimal representation | 40600000 |
| Decimal representation | 3.5 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 3.5

This 1 is given to you for free.
It is part of the formula/standard.

sign=+1 exp=1 mantisse=1.75

$(-1)^{0} \times 2^{(128-127)} \times (1 + 0.5 + 0.25) = 3.5$

| | |
|---|---|
| Binary representation | 01000000011000000000000000000000 |
| Hexadecimal representation | 40600000 |
| Decimal representation | 3.5 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 3.5

sign=+1 exp=1 mantisse=1.75

$1 \times 2^1 \times 1.75 = 3.5$

| | |
|---|---|
| Binary representation | 01000000011000000000000000000000 |
| Hexadecimal representation | 40600000 |
| Decimal representation | 3.5 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 5.0

sign=+1 exp=2 mantisse=1.25

| | |
|---|---|
| Binary representation | 01000000101000000000000000000000 |
| Hexadecimal representation | 40A00000 |
| Decimal representation | 5.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 6.0

sign=+1 exp=2 mantisse=1.5

| | |
|---|---|
| Binary representation | 01000000110000000000000000000000 |
| Hexadecimal representation | 40C00000 |
| Decimal representation | 6.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing -7.0

sign=-1 exp=2 mantisse=1.75

| | |
|---|---|
| Binary representation | 11000000111000000000000000000000 |
| Hexadecimal representation | C0E00000 |
| Decimal representation | -7.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 0.8

sign=+1 exp=-1 mantisse=1.6

| | |
|---|---|
| Binary representation | 00111111010011001100110011001101 |
| Hexadecimal representation | 3F4CCCCD |
| Decimal representation | 0.8 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing 0.8

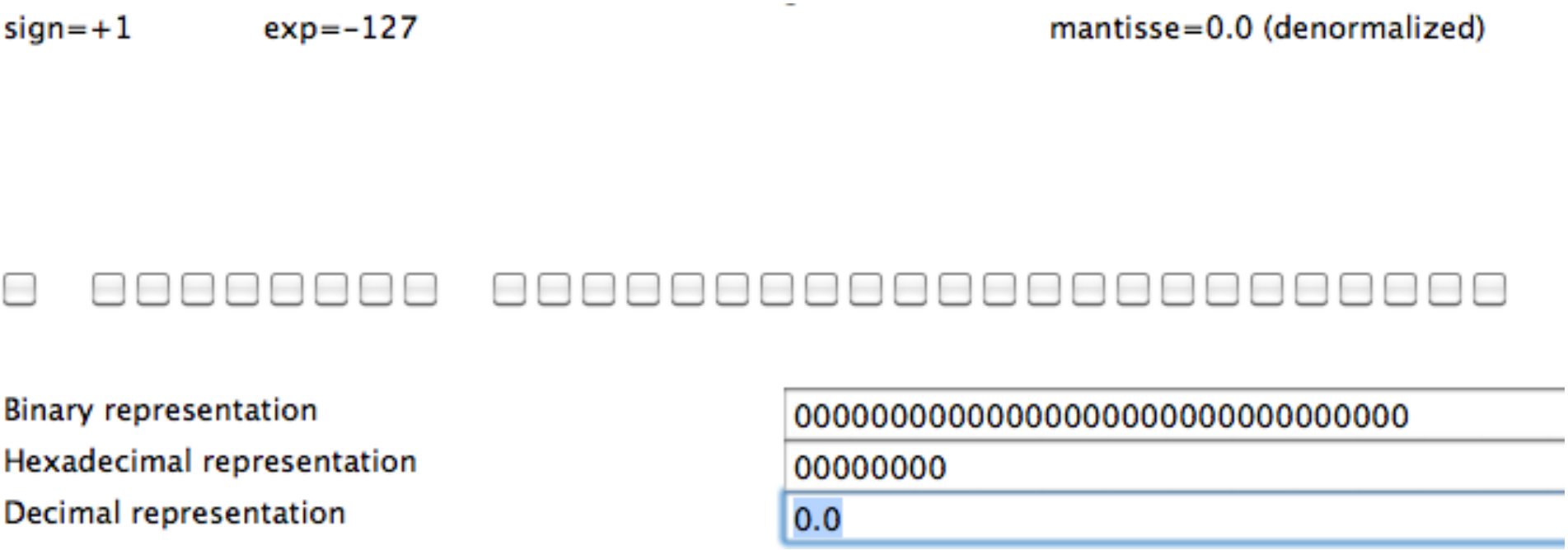
sign=+1 exp=-1 mantisse=1.6

| | |
|---|---|
| Binary representation | 00111111010011001100110011001101 |
| Hexadecimal representation | 3F4CCCCD |
| Decimal representation | 0.8 |

This decimal number cannot be stored perfectly in this format!
The bits in the mantissa are periodic and will extend to infinity.

Think of storing $1/3 = 0.33333(3)$ with fixed number of decimal digits.
This is similar: $0.8_{10}$ has no finite representation in IEEE 754.

# Representing 0.0




[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing -0.0

sign=-1 exp=-127 mantisse=0.0 (denormalized)

| | |
|---|---|
| Binary representation | 10000000000000000000000000000000 |
| Hexadecimal representation | 80000000 |
| Decimal representation | -0.0 |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing +Infinity

sign=+1 exp=128 mantisse=1.0

| | |
|---|---|
| Binary representation | 01111111100000000000000000000000 |
| Hexadecimal representation | 7F800000 |
| Decimal representation | Infinity |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing -Infinity

sign=-1 exp=128 mantisse=1.0

| | |
|---|---|
| Binary representation | 11111111100000000000000000000000 |
| Hexadecimal representation | FF800000 |
| Decimal representation | -Infinity |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing NaN

sign=+1 exp=128 mantisse=1.5

| | |
|---|---|
| Binary representation | 01111111110000000000000000000000 |
| Hexadecimal representation | 7FC00000 |
| Decimal representation | NaN |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing NaN

sign=+1 exp=128 mantisse=1.9999999

| | |
|---|---|
| Binary representation | 01111111111111111111111111111111 |
| Hexadecimal representation | 7FFFFFFF |
| Decimal representation | NaN |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

# Representing NaN

sign=+1 exp=128 mantisse=1.0000001

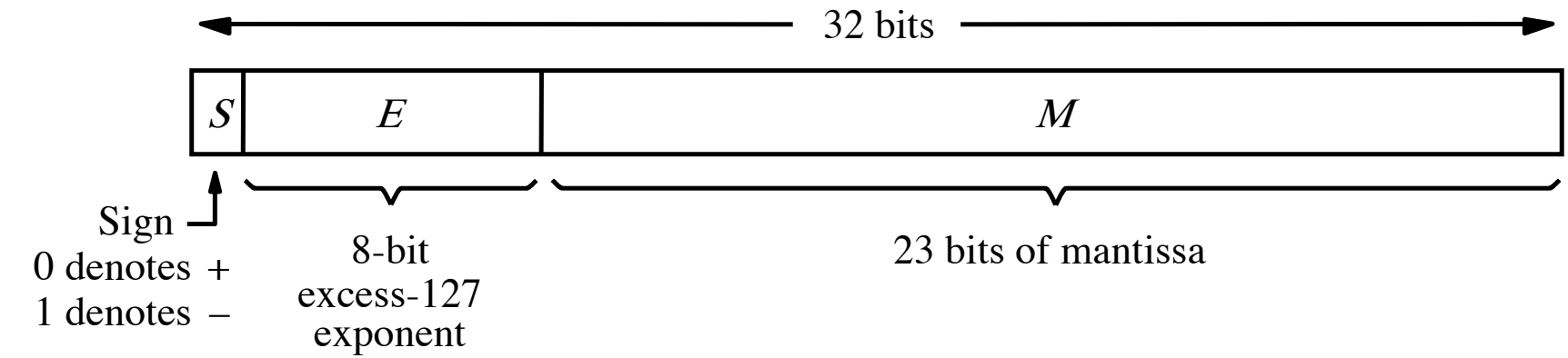
| | |
|---|---|
| Binary representation | 01111111100000000000000000000001 |
| Hexadecimal representation | 7F800001 |
| Decimal representation | NaN |

[https://www.h-schmidt.net/FloatConverter/IEEE754.html]

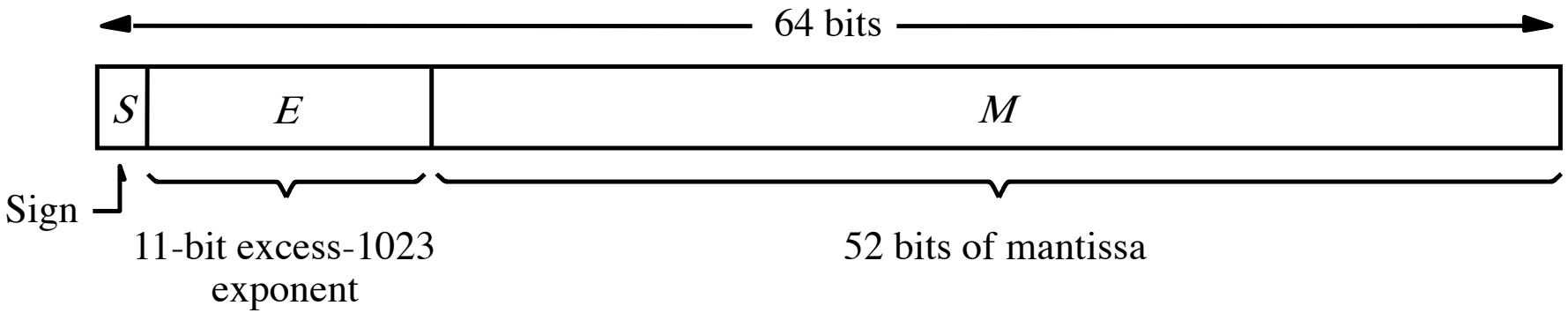
| Range Name | Sign (*s*) 1 [31] | Exponent (*e*) 8 [30-23] | Mantissa (*m*) 23 [22-0] | Hexadecimal Range | Range | Decimal Range § |
|---|---|---|---|---|---|---|
| Quiet -NaN | 1 | 11..11 | 11..11 : 10..01 | FFFFFFFF : FFC00001 | | |
| Indeterminate | 1 | 11..11 | 10..00 | FFC00000 | | |
| Signaling -NaN | 1 | 11..11 | 01..11 : 00..01 | FFBFFFFF : FF800001 | | |
| -Infinity (Negative Overflow) | 1 | 11..11 | 00..00 | FF800000 | $< -(2-2^{-23}) \times 2^{127}$ | $\leq$ -3.4028235677973365E+38 |
| **Negative Normalized** $-1.m \times 2^{(e-127)}$ | 1 | 11..10 : 00..01 | 11..11 : 00..00 | FF7FFFFF : 80800000 | $-(2-2^{-23}) \times 2^{127}$ : $-2^{-126}$ | -3.4028234663852886E+38 : -1.1754943508222875E-38 |
| **Negative Denormalized** $-0.m \times 2^{(-126)}$ | 1 | 00..00 | 11..11 : 00..01 | 807FFFFF : 80000001 | $-(1-2^{-23}) \times 2^{-126}$ : $-2^{-149}$ $(-(1+2^{-52}) \times 2^{-150})$ * | -1.1754942106924411E-38 : -1.4012984643248170E-45 (-7.0064923216240862E-46) * |
| Negative Underflow | 1 | 00..00 | 00..00 | 80000000 | $-2^{-150}$ : < -0 | -7.0064923216240861E-46 : < -0 |
| **-0** | 1 | 00..00 | 00..00 | 80000000 | -0 | -0 |
| **+0** | 0 | 00..00 | 00..00 | 00000000 | 0 | 0 |
| Positive Underflow | 0 | 00..00 | 00..00 | 00000000 | > 0 : $2^{-150}$ | > 0 : 7.0064923216240861E-46 |
| **Positive Denormalized** $0.m \times 2^{(-126)}$ | 0 | 00..00 | 00..01 : 11..11 | 00000001 : 007FFFFF | $((1+2^{-52}) \times 2^{-150})$ * $2^{-149}$ : $(1-2^{-23}) \times 2^{-126}$ | (7.0064923216240862E-46) * 1.4012984643248170E-45 : 1.1754942106924411E-38 |
| **Positive Normalized** $1.m \times 2^{(e-127)}$ | 0 | 00..01 : 11..10 | 00..00 : 11..11 | 00800000 : 7F7FFFFF | $2^{-126}$ : $(2-2^{-23}) \times 2^{127}$ | 1.1754943508222875E-38 : 3.4028234663852886E+38 |
| +Infinity (Positive Overflow) | 0 | 11..11 | 00..00 | 7F800000 | $> (2-2^{-23}) \times 2^{127}$ | $\geq$ 3.4028235677973365E+38 |
| Signaling +NaN | 0 | 11..11 | 00..01 : 01..11 | 7F800001 : 7FBFFFFF | | |
| Quiet +NaN | 0 | 11..11 | 10..00 : 11..11 | 7FC00000 : 7FFFFFFF | | |

[http://babbage.cs.qc.cuny.edu/IEEE-754.old/References.xhtml]

# Float (32-bit) vs. Double (64-bit)




(a) Single precision




(b) Double precision

[Figure 3.37 from the textbook]

# Sample Midterm2 Problem

**(a) Convert $3FA00000_{16}$ (a 32-bit float stored in IEEE 754 format) to decimal:**

0 | 0 1 1 1 1 1 1 1 | 0 1 0 0000 0000 0000 0000 0000

127

$2^{-2}$

$$(-1)^0 \times 2^{127-127} \times \left(1 + \frac{1}{4}\right) = 2^0 \times \frac{5}{4} = 1.25$$

# Sample Midterm2 Problem

**(b) Convert the following 32-bit float number (in IEEE 754 format) to decimal**

1 | 1 0 0 0 0 1 1 0 | 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 → negative

1 0 0 0 0 1 1 0 = 128 + 4 + 2 = 134

$$(-1)^1 \times 2^{134-127} \times \left(1 + \frac{1}{4} + \frac{1}{8}\right) = -2^7 \times \frac{11}{8} = -2^3 \times 2^4 \times \frac{11}{23} = -16 \times 11$$

$$= -176$$

# Sample Midterm2 Problem

**(c) Write down the 32-bit floating point representation (in IEEE 754 format) for $0110_2$**

$0110_2 = 6_{10}$

The highest power of 2 less than 6 is $2^2 = 4$.

$6 / 4 = 1.5$

$6 = (-1)^0 \times \underbrace{2^2}_{2^{129-127}} \times \left(1 + \frac{1}{2}\right)$

$-4$

$20$

$-20$

$0$

0 | 10000001 | 1000.....0

positive

22 zeros

# Sample Midterm2 Problem

**(d) Write down the 32-bit floating point representation (in IEEE 754 format) for $-7_{10}$**

$7/4 = 1.75$

$$(-1)^1 \times \underbrace{2^2}_{2^{129-127}} \times \left(1 + \frac{1}{2} + \frac{1}{4}\right)$$

$-4$ / $30$ / $-28$ / $20$ / $-20$ / $0$

1 | 10000001 | 1100.....0

21 zeros

negative

# Some Issues When Storing Double Numbers

# Memory Analogy

**Address 0**

**Address 1**

**Address 2**

**Address 3**

**Address 4**

**Address 5**

**Address 6**

# Memory Analogy (32 bit architecture)

**Address 0**

**Address 4**

**Address 8**

**Address 12**

**Address 16**

**Address 20**

**Address 24**

# Memory Analogy (32 bit architecture)

Address 0x00

Address 0x04

Address 0x08

Address 0x0C

Address 0x10

Address 0x14

Address 0x18

Hexadecimal

Address 0x0A

Address 0x0D

# Storing a Double

**Address 0x08**

**Address 0x0C**

# Storing 3.14

- **3.14 in binary IEEE-754 double precision (64 bits)**

| sign | exponent | mantissa |
|---|---|---|
| 0 | 10000000000 | 1001000111101011100001010001111010111000010100011111 |

- **In hexadecimal this is (hint: groups of four):**

0100 0000 0000 1001 0001 1110 1011 1000 0101 0001 1110 1011 1000 0101 0001 1111

4 0 0 9 1 E B 8 5 1 E B 8 5 1 F

# Storing 3.14

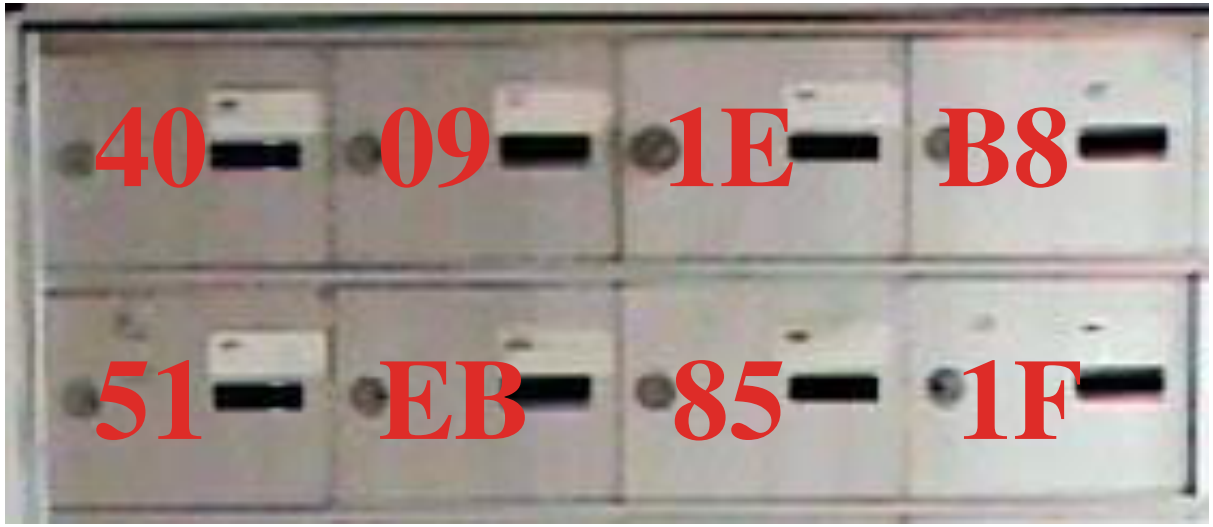
- **So 3.14 in hexadecimal IEEE-754 is 40091EB851EB851F**
- **This is 64 bits.**
- **On a 32 bit architecture there are 2 ways to store this**

| | Big-Endian | Little-Endian |
|---|---|---|
| **Small address:** | **40091EB8** | **51EB851F** |
| **Large address:** | **51EB851F** | **40091EB8** |
| Example CPUs: | Motorola 6800 | Intel x86 |

# Storing 3.14

**Address 0x08**

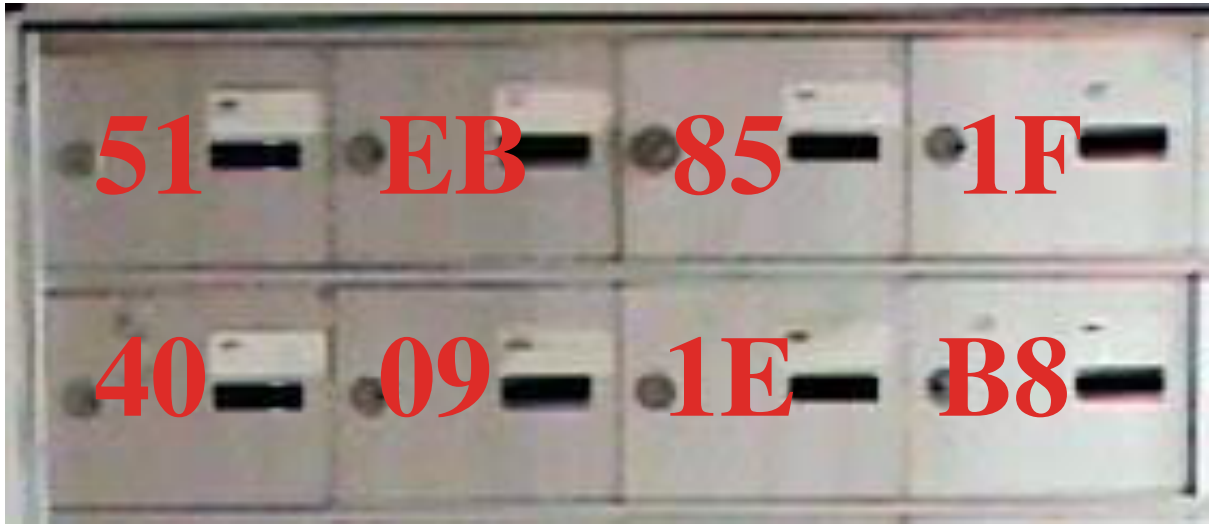
**Address 0x0C**




Big-Endian

**Address 0x08**

**Address 0x0C**




Little-Endian

# Storing 3.14 on a Little-Endian Machine (these are the actual bits that are stored)

Address 0x08

Address 0x0C

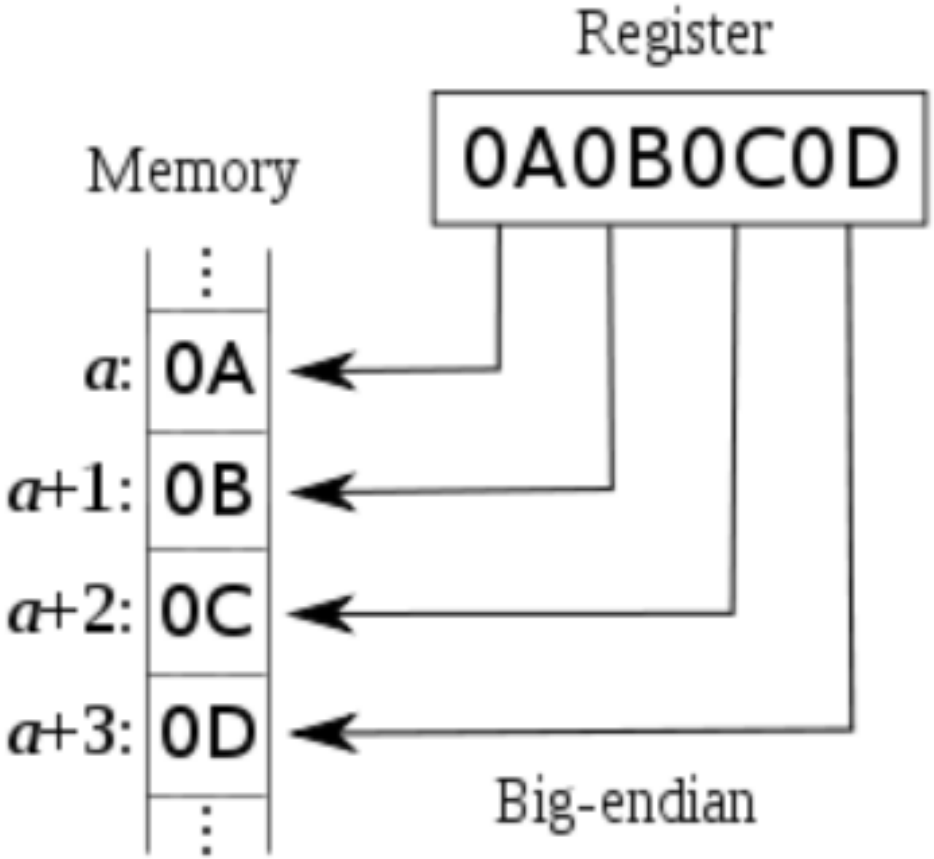

**Once again, 3.14 in IEEE-754 double precision is:**

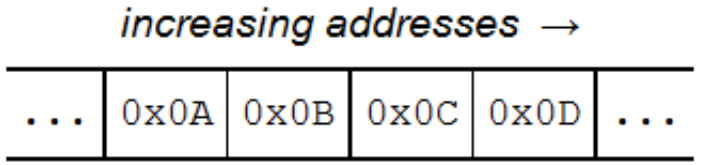
sign exponent mantissa

0 10000000000 1001000111101011100001010001111010111000010100011111

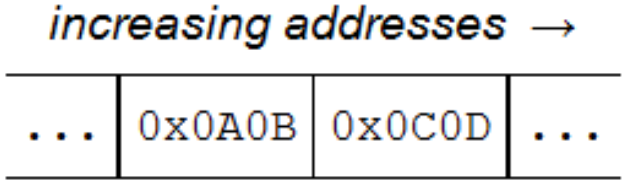
# They are stored in binary
# (the hexadecimals are just for visualization)

# Big-Endian



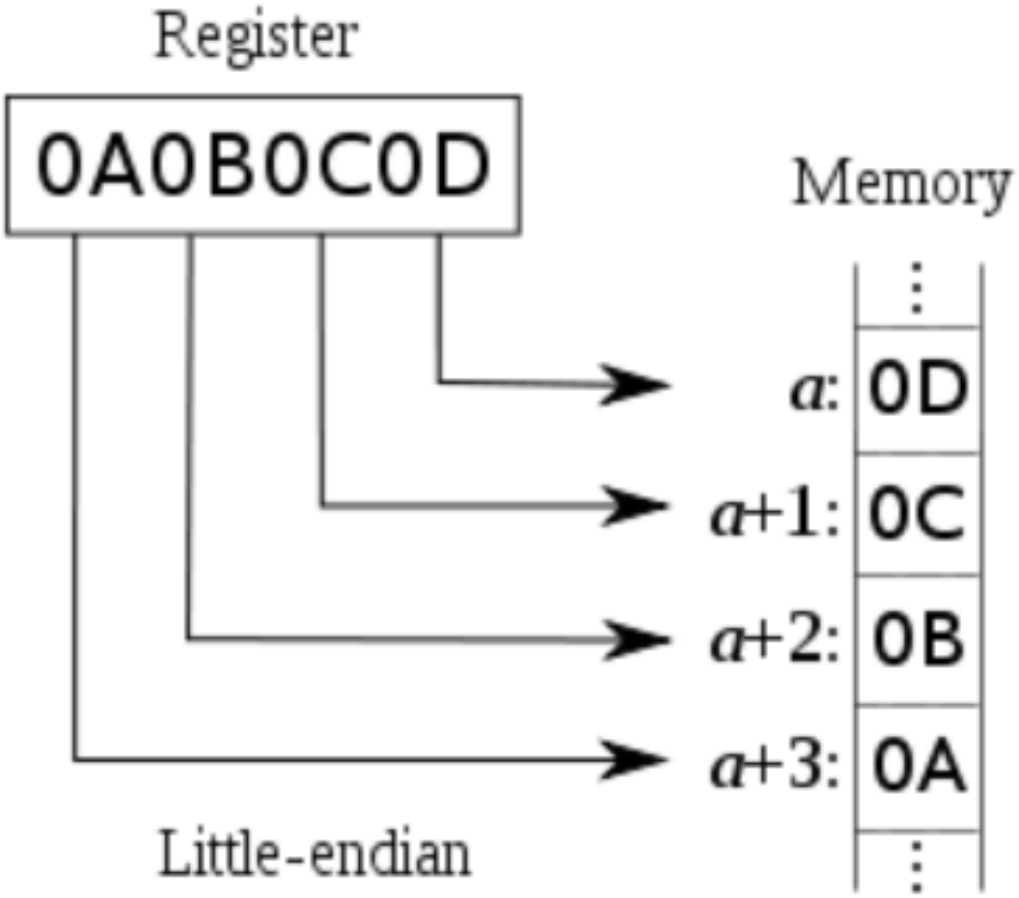

(8-bit architecture)

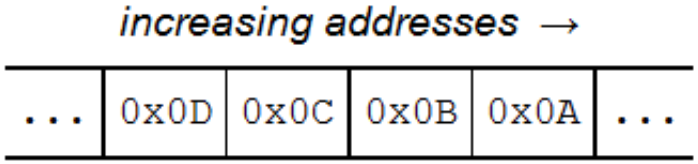



(16-bit architecture)

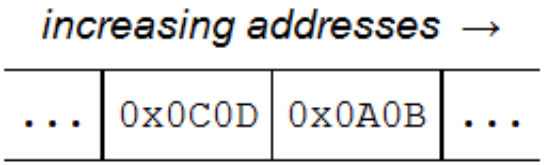
http://en.wikipedia.org/wiki/Endianness

# Little Endian




(8-bit architecture)




(16-bit architecture)

http://en.wikipedia.org/wiki/Endianness

# Big-Endian/Little-Endian analogy


[image fom http://www.simplylockers.co.uk/images/PLowLocker.gif]

# Big-Endian/Little-Endian analogy


[image fom http://www.simplylockers.co.uk/images/PLowLocker.gif]

# Big-Endian/Little-Endian analogy


[image fom http://www.simplylockers.co.uk/images/PLowLocker.gif]

# What would be printed? (don't try this at home)

```
double pi = 3.14;
printf("%d",pi);
```

- **Result: 1374389535**

**Why?**

- **3.14 = 40091EB851EB851F (in double format)**
- **Stored on a little-endian 32-bit architecture**
  - **51EB851F (1374389535 in decimal)**
  - **40091EB8 (1074339512 in decimal)**

# What would be printed? (don't try this at home)

```
double pi = 3.14;
printf("%d %d", pi);
```

- **Result: 1374389535  1074339512**

**Why?**

- **3.14  = 40091EB851EB851F (in double format)**
- **Stored on a little-endian 32-bit architecture**
  - **51EB851F (1374389535 in decimal)**
  - **40091EB8 (1074339512 in decimal)**

  - **The second %d uses the extra bytes of pi that were not printed by the first %d**

# What would be printed? (don't try this at home)

```
double a = 2.0;
printf("%d",a);
```

- **Result: 0**

**Why?**

- **2.0 = 40000000 00000000 (in hex IEEE double format)**
- **Stored on a little-endian 32-bit architecture**
  - **00000000 (0 in decimal)**
  - **40000000 (1073741824 in decimal)**

# What would be printed? (an even more advanced example)

```
int a[2];                    // defines an int array
a[0]=0;
a[1]=0;
scanf("%lf", &a[0]);     // read 64 bits into 32 bits
// The user enters 3.14
printf("%d %d", a[0], a[1]);
```

- **Result: 1374389535 1074339512**

**Why?**

- **3.14 = 40091EB851EB851F (in double format)**
- **Stored on a little-endian 32-bit architecture**
  - **51EB851F (1374389535 in decimal)**
  - **40091EB8 (1074339512 in decimal)**
- **The double 3.14 requires 64 bits which are stored in the two consecutive 32-bit integers named a[0] and a[1]**

# Questions?

# THE END