CS-200
Computer Architecture
—
Part 1e. Instruction Set Architecture Arithmetic

Paolo Ienne
<paolo.ienne@epfl.ch>

Notation

- Number (represented on a specific no. of digits/bits)

$$A = A^{(n)} = A^{(m)}$$

- Number (in binary or decimal)

$$A = A_{10} = A_2 = A_{2c}$$

Binary, 2's complement

Binary

- Individual digits (bits)

$$a_{n-1},\ a_{n-2}, \ldots\ a_2,\ a_1,\ a_0$$

- Digit string (representation)

$$\langle a_{n-1}a_{n-2}\ldots a_2a_1a_0\rangle$$

Simply 100010
if the digits are known

Numbers

We usually care for three types of numbers:

- **Integers** (signed and unsigned)

$$0, 1, 2, 3, 4294967295, -2147483648$$

- **Fixed Point**

$$0.12, 3.14, 1073741823.75$$

  - Essentially integers with **implicit 10^k or 2^k scaling**
  - Extremely important in practice (most signal-processing is fixed point)

- **Floating Point**

$$3.14\text{E}3, -2.5\text{E}1, 1.0\text{E}0, 4.2\text{E}{-2}, -1.5\text{E}{-3}$$

Unsigned Integers

- Weighted (positional)
- Nonredundant
- Fixed-radix (radix-10 or radix-2)
- Canonical

- Definition:

$$A = \langle a_{n-1} a_{n-2} \ldots a_2 a_1 a_0 \rangle = \sum_{i=0}^{n-1} a_i R^i$$

If R = 2, binary

Signed Integers

- **Sign-and-Magnitude**
- **2's Complement** (particular choice of True-and-Complement)
- **Biased**
  - Practically used only in Floating Point numbers (mentioned later)

Sign and Magnitude

- **Human friendly!**
- The first symbol is a sign (+/− for humans, 0/1 for computers)
- The rest is an unsigned number:

$$+100, -2345$$

$$+111_2 = 0111_2^{(4)}$$

$$-111_2 = 1111_2^{(4)}$$

If we use 0/1 for the sign, the number of bits matters

- Definition:

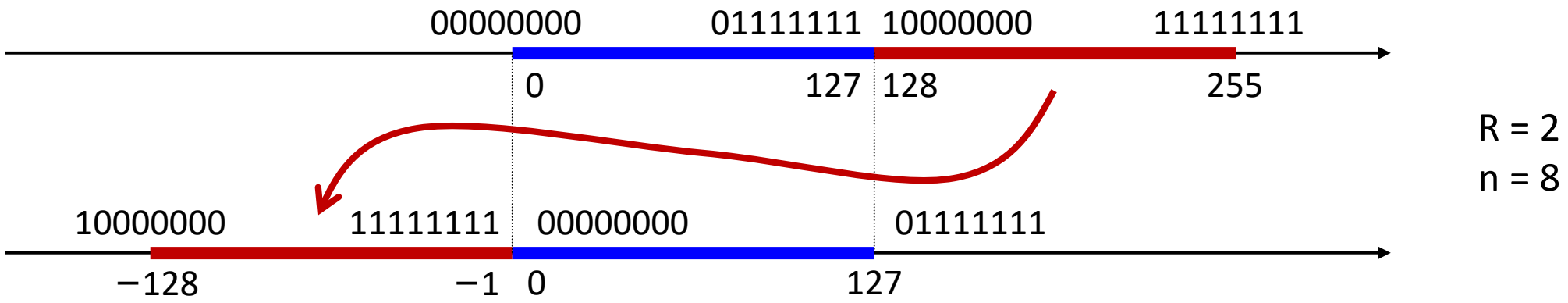
$$A = \langle s a_{n-2} \ldots a_2 a_1 a_0 \rangle = (-1)^s \cdot \sum_{i=0}^{n-2} a_i R^i$$

0 or 1

If R = 2, binary

Radix's Complement

- Special form of **True-and-Complement** with C = R^n



- Property when R = 2:

$$A = \langle a_{n-1} a_{n-2} \ldots a_2 a_1 a_0 \rangle = -a_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

Radix's Complement

- **Not** a **human-friendly** representation
- In **decimal** (10's complement):

$$5{,}678^{(5)}_{10c} = 05{,}678_{10c} = +5{,}678_{10}$$
$$9{,}999{,}999^{(7)}_{10c} = 9{,}999{,}999_{10} - 10^7 = -1_{10}$$
$$8{,}766^{(4)}_{10c} = 8{,}766_{10} - 10^4 = -1{,}234_{10}$$

- In **binary** (2's complement):

$$0100{,}1101{,}0010^{(12)}_{2c} = 100{,}1101{,}0010_2 = +1{,}234_{10}$$
$$1111{,}1111^{(8)}_{2c} = 255_{10} - 2^8 = -1_{10}$$
$$1011{,}0010{,}1110^{(12)}_{2c} = 2862_{10} - 2^{12} = -1234_{10}$$

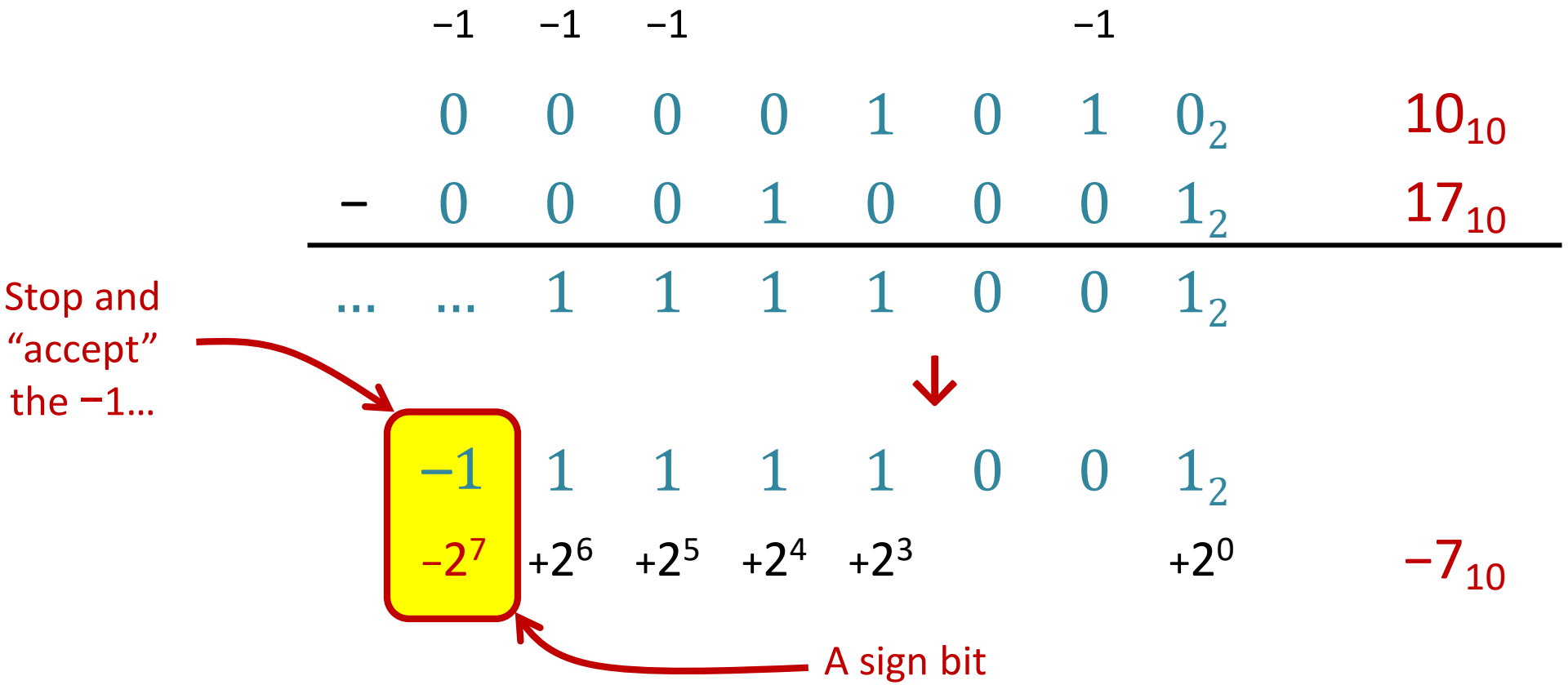
2's Complement from Subtraction

- Consider a "normal" **paper-and-pencil subtraction**

$$\begin{array}{rcccccccc} & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0_2 \quad 10_{10} \\ - & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1_2 \quad 17_{10} \\ \hline \end{array}$$

2's Complement from Subtraction

- Consider a "normal" **paper-and-pencil subtraction**



Addition Is Unchanged from Unsigned

- Only two instructions (with the immediate version; `subi` is a pseudo)

Arithmetic						
add	rd,rs1,rs2	rd ← rs1 + rs2	R	0x00	0x0	0x33
addi	rd,rs1,imm	rd ← rs1 + sext(imm)	I		0x0	0x13
sub	rd,rs1,rs2	rd ← rs1 − rs2	R	0x20	0x0	0x33

- Old architectures (MIPS, notably) had distinct `add` and `addu` but it was essentially a misnomer; **ignore** it and do not be confused!
- Instead, addition of Sign-and-Magnitude numbers is a different problem (see later) → this is why **2's complement is the universal representation** of signed integers today

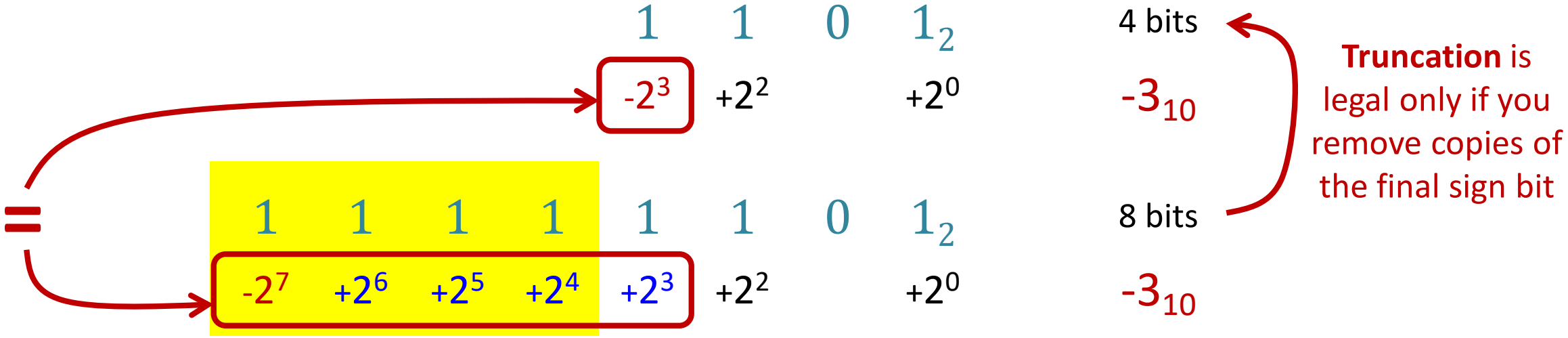
Sign Extension

- **Unsigned numbers** can be though as having infinite 0s in front

$$-1_{10} = -0001_{10}$$

$$1{,}0101_2 = \ 0000{,}0000{,}0001{,}0101_2$$

- Instead, **2's complement numbers** have infinite replicas of the MSB/sign bit in front



Instructions for Signed Numbers

Insert zeroes (**l** = logic → **unsigned**) or sign bits (**a** = arithmetic → **signed**)

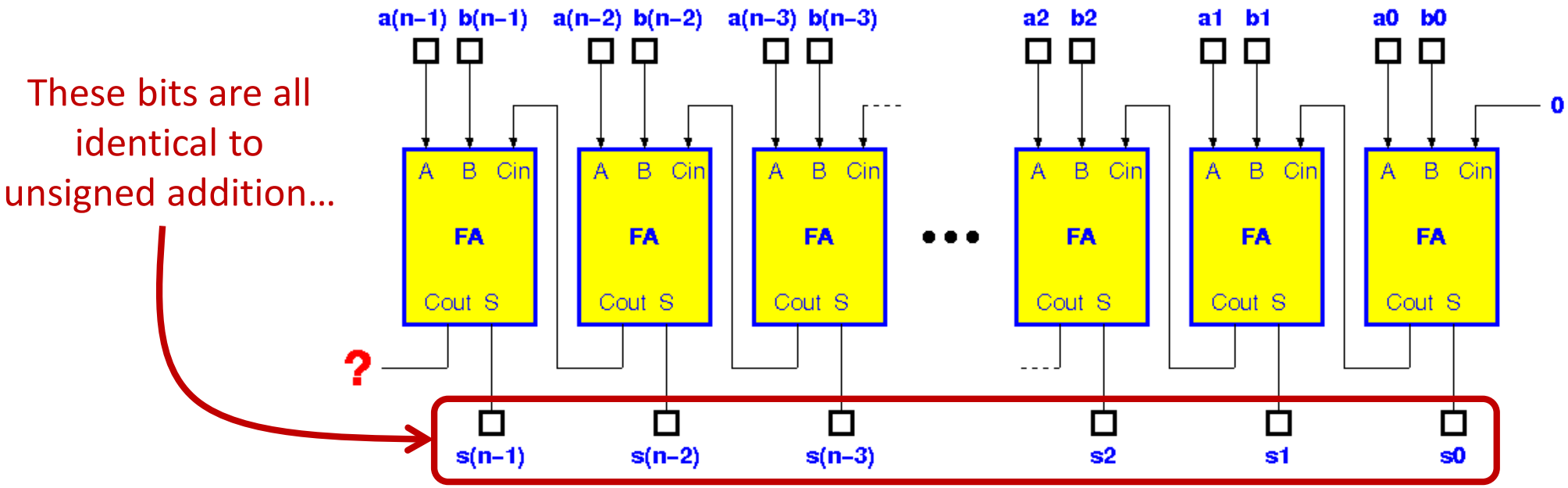
Shift						
srl	rd,rs1,rs2	rd ← rs1 $\gg_u$ rs2	R	0x00	0x5	0x33
srli	rd,rs1,imm	rd ← rs1 $\gg_u$ imm	I	0x00	0x5	0x13
sra	rd,rs1,rs2	rd ← rs1 $\gg_s$ rs2	R	0x20	0x5	0x33
srai	rd,rs1,imm	rd ← rs1 $\gg_s$ imm	I	0x20	0x5	0x13
Compare						
slt	rd,rs1,rs2	rd ← rs1 $<_s$ rs2	R	0x00	0x2	0x33
slti	rd,rs1,imm	rd ← rs1 $<_s$ sext(imm)	I		0x2	0x13
sltu	rd,rs1,rs2	rd ← rs1 $<_u$ rs2	R	0x00	0x3	0x33
sltiu	rd,rs1,imm	rd ← rs1 $<_u$ sext(imm)	I		0x3	0x13
Branch						
blt	rs1,rs2,imm	pc ← pc + sext(imm $\ll$ 1), if rs1 $<_s$ rs2	B		0x4	0x63
bge	rs1,rs2,imm	pc ← pc + sext(imm $\ll$ 1), if rs1 $\geq_s$ rs2	B		0x5	0x63
bltu	rs1,rs2,imm	pc ← pc + sext(imm $\ll$ 1), if rs1 $<_u$ rs2	B		0x6	0x63
bgeu	rs1,rs2,imm	pc ← pc + sext(imm $\ll$ 1), if rs1 $\geq_u$ rs2	B		0x7	0x63
Load						
lb	rd,imm(rs1)	rd ← sext(mem[rs1 + sext(imm)][7 : 0])	I		0x0	0x03
lbu	rd,imm(rs1)	rd ← zext(mem[rs1 + sext(imm)][7 : 0])	I		0x4	0x03
lh	rd,imm(rs1)	rd ← sext(mem[rs1 + sext(imm)][15 : 0])	I		0x1	0x03
lhu	rd,imm(rs1)	rd ← zext(mem[rs1 + sext(imm)][15 : 0])	I		0x5	0x03

$1110_2/2 = 0111_2$
but
$1110_{2c}/2 = 1111_{2c}$

$0000_2 < 1111_2$
but
$0000_{2c} > 1111_{2c}$

Overflows in 2's Complement Addition

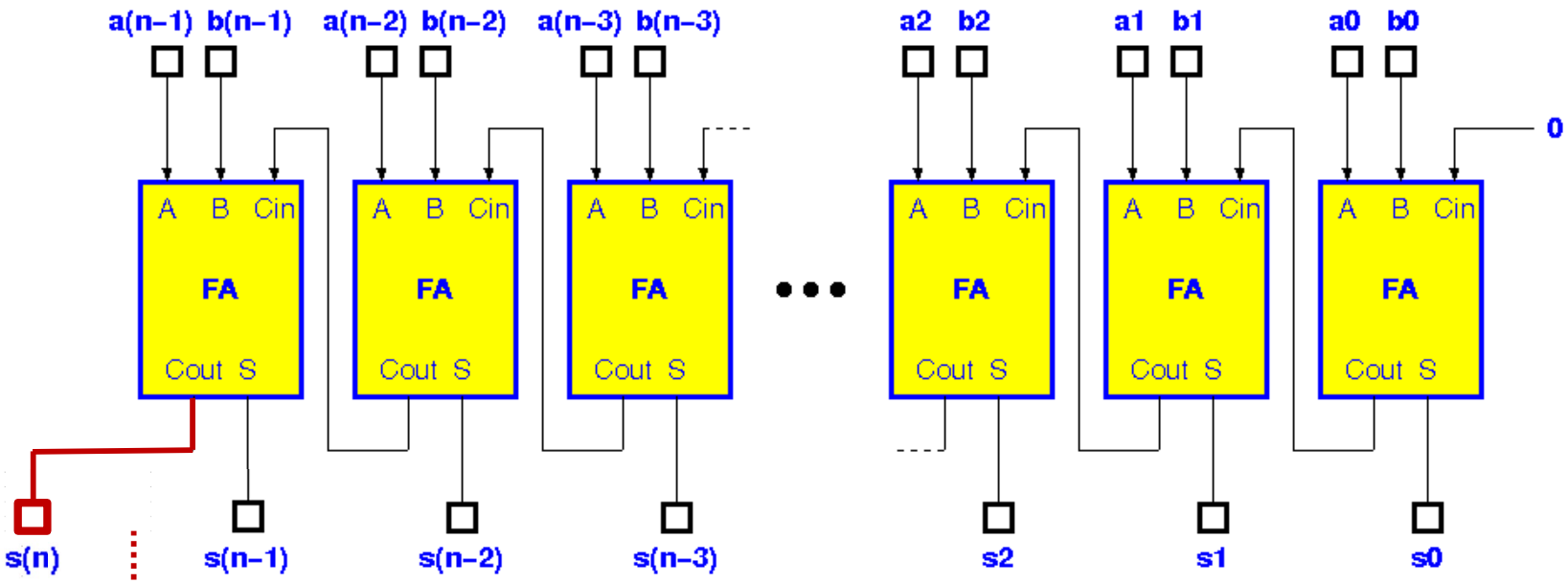
- The **sum** is the same as with **unsigned numbers**:



…but how to assess **overflows**?

Overflows in Hardware

- In hardware, **carry out** is the only missing bit from the **complete** result
- We can think of overflows as a **truncation** problem:

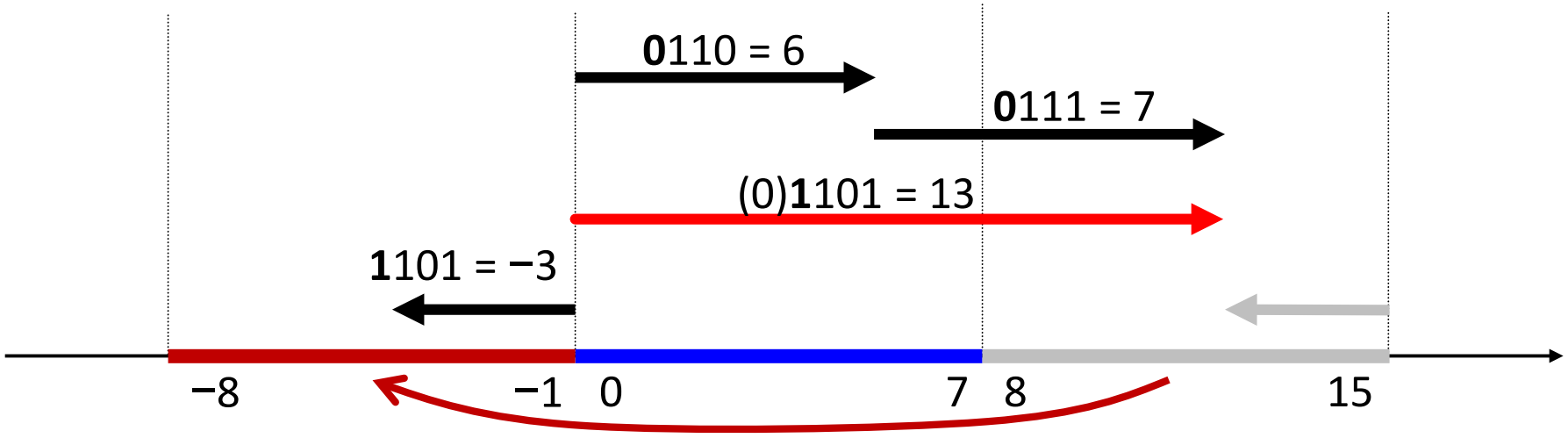


No	1	…	For unsigned numbers, the carry bit must be **zero**
Ok	1	1	In 2's complement, the carry bit must be **equal to the next bit**
No	1	0	

Overflow in Software

- Some architectures (e.g., **x86**) give us the **carry bit** in a special "register" (a **flag**)

  → overflow detection is the same as in hardware
- Other (modern) architectures give us **only the result** of the addition (e.g., **RISC-V**)
- Detection usually based on the following observations:
  - If addition of **opposite sign numbers**, magnitude can only reduce → **no overflow possible**
  - If addition of **same sign numbers**, overflow possible but the sign of the result will appear wrong



Detect Addition Overflow in Software

- Add two 32-bit signed integers **and detect overflow**
  - At call time, `a0` and `a1` contain the two integers
  - On return, `a0` contains the result and `a1` must be nonzero in case of overflow

Detect Addition Overflow in Software

```
add_with_overflow:
    add   t0, a0, a1        # Perform addition of a0 + a1, store result in t0

    xor   t1, a0, a1        # t1<0 if the operands have different signs
    not   t1, t1            # t1<0 if the operands have the same sign
    xor   t2, t0, a1        # t2<0 if the result has different sign from operand
    and   t1, t1, t2        # t1<0 if the same sign ops and different sign result
    srli  a1, t1, 31        # move "sign" to LSB of a1

    mv    a0, t0
    ret                     # Return sum in a0, overflow flag in a1
```

Detect Addition Overflow in Software (Better)

```
add_with_overflow:
    add   t0, a0, a1        # Perform addition of a0 + a1, store result in t0

    slti  t1, a1, 0         # t1 = 1 if one operand is negative
    slt   t2, t0, a0        # t2 = 1 if the result is smaller than the other operand
    xor   a1, t1, t2        # overflow if and only if
                            # - one op negative and result larger than other op
                            # - one op zero/positive and result smaller than other op

    mv    a0, t0
    ret                     # Return sum in a0, overflow flag in a1
```

$A + \bar{A} = -1$

- A “strange” but **very useful property**

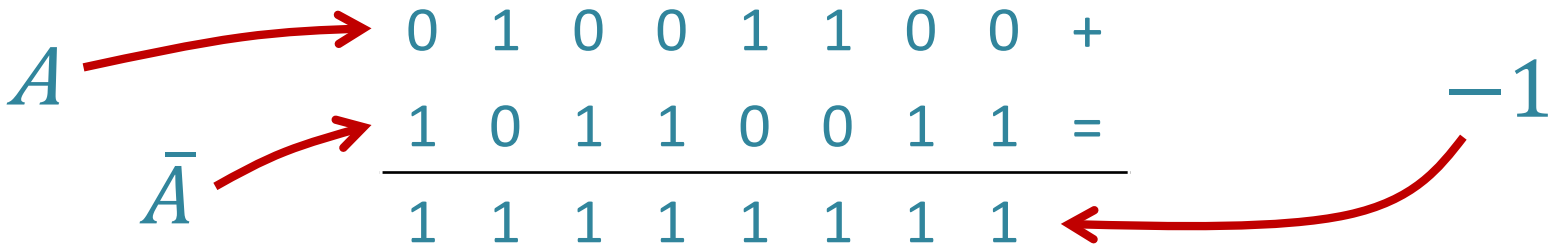
$$A + \bar{A} = -1 \quad \text{or} \quad -A = \bar{A} + 1$$

- Not too hard to prove

$$\left(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i\right) + \left(-\overline{a_{n-1}}2^{n-1} + \sum_{i=0}^{n-2} \overline{a_i} 2^i\right) =$$

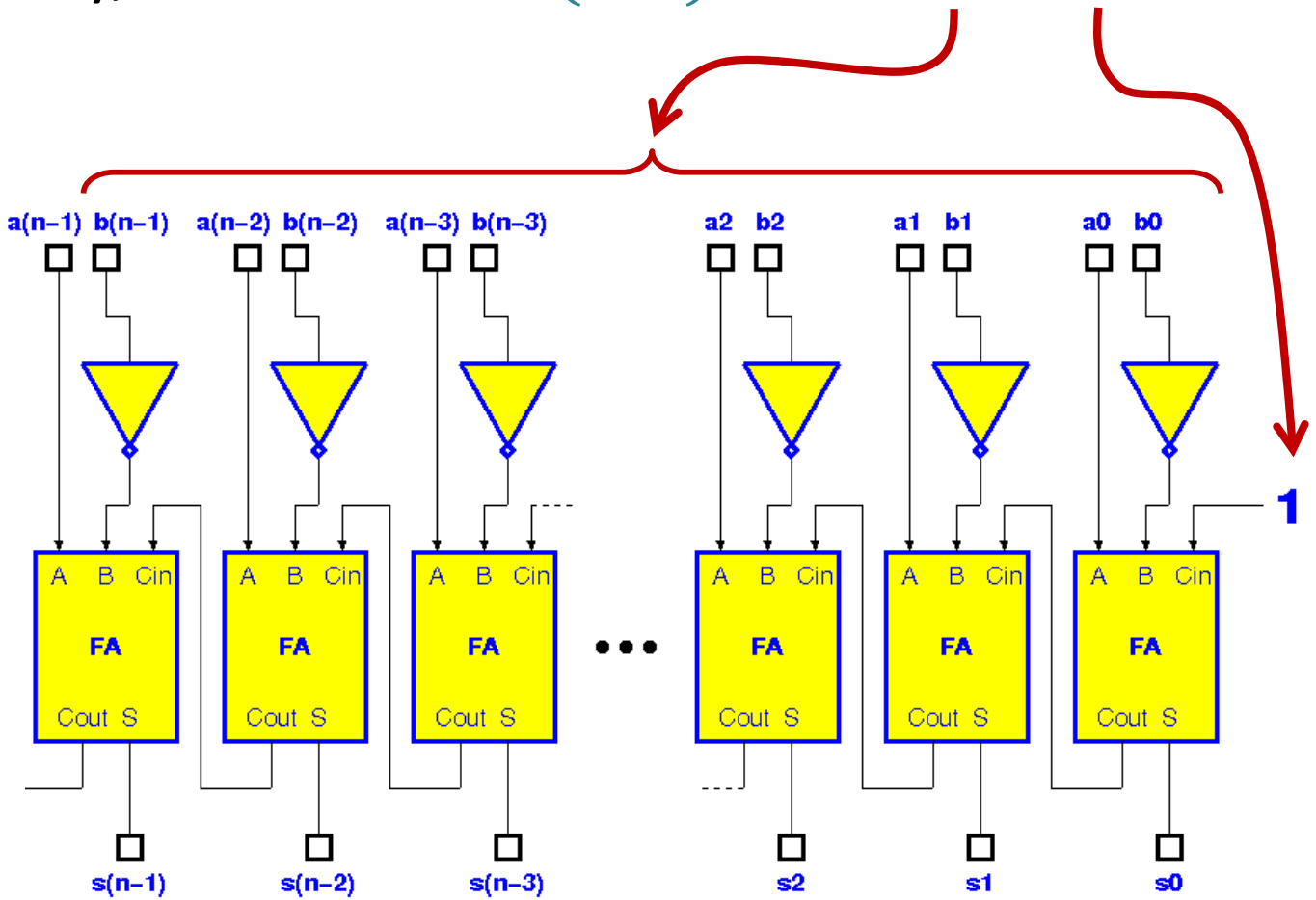
$$= -(a_{n-1} + \overline{a_{n-1}}) \cdot 2^{n-1} + \sum_{i=0}^{n-2} (a_i + \overline{a_i}) \cdot 2^i = -2^{n-1} + \sum_{i=0}^{n-2} 2^i = -1$$

- Also somehow intuitive

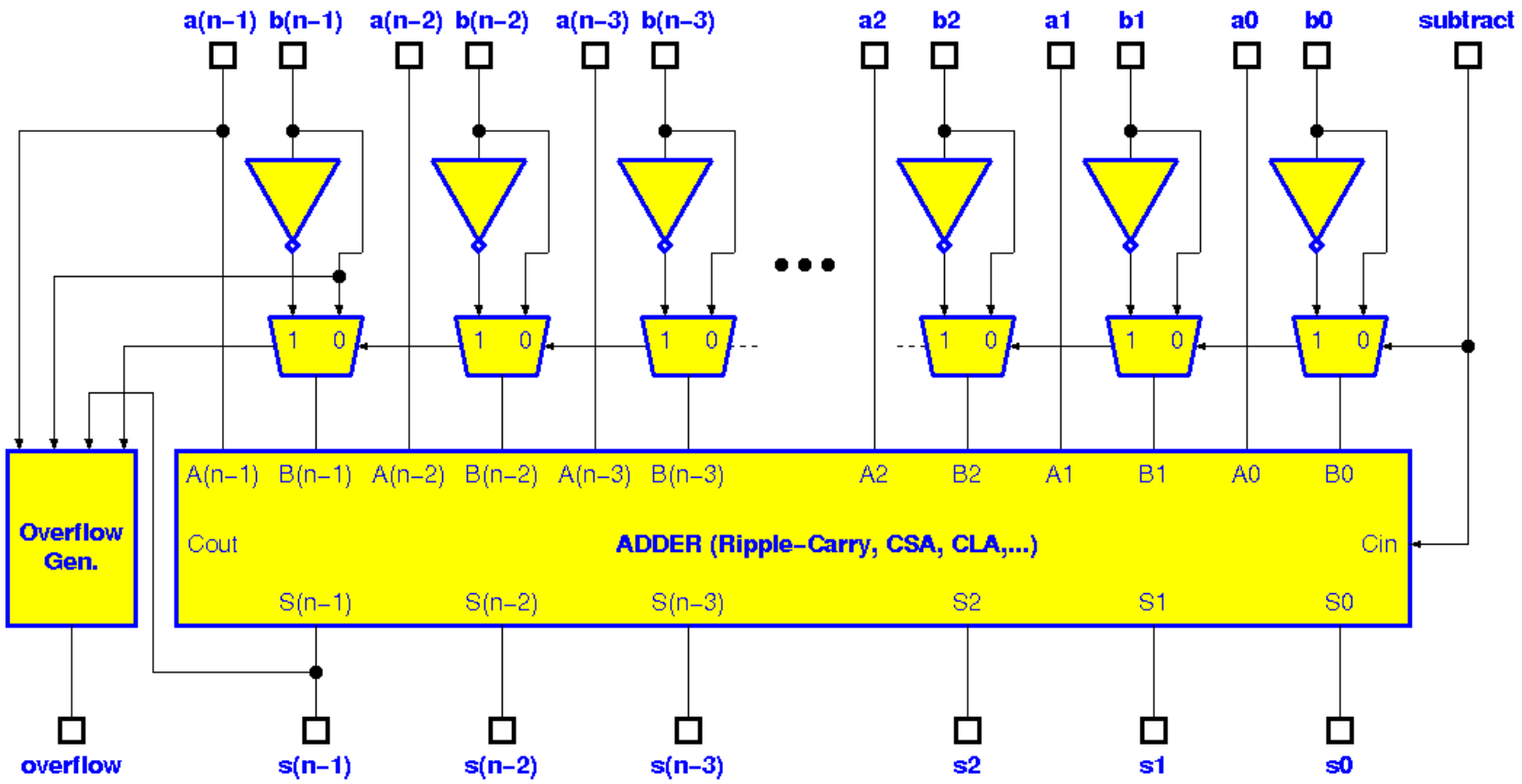


Two's Complement Subtractor

- Using this property, $A - B = A + (-B) = A + \overline{B} + 1$



Two's Complement Add/Subtract Units



Fun Stuff: Bounds Check

- Check for a signed number `t0` (e.g., an array **index**) to be **within the bounds 0..N−1** where N is `t1`

```
bgeu t0, t1, out_of_bound
```

Unsigned!

- **Two checks with a single branch!**
  - If `t0` ≥ 0, `bgeu` is like `bge` and the right behaviour is evident
  - If `t0` < 0, as an unsigned `t0` looks like larger than any signed positive

Floating Point

- Corresponds to our **everyday habits**

Engineering notation

Normalized scientific notation

.18 µm	→	$.18 \cdot 10^{-6}$ m	→	$1.8 \cdot 10^{-7}$ m
75 km	→	$75 \cdot 10^{3}$ m	→	$7.5 \cdot 10^{4}$ m
35 mm	→	$35 \cdot 10^{-3}$ m	→	$3.5 \cdot 10^{-2}$ m
2.5 m	→	$2.5 \cdot 10^{0}$ m	→	$2.5 \cdot 10^{0}$ m

- A significand (or **mantissa**) and an **exponent** of the base, for instance

2's complement exponent

$$X = \langle s a_{n-1} \ldots a_2 a_1 a_0 e_{m-1} \ldots e_1 e_0 \rangle = \underbrace{(-1)^s \cdot \sum_{i=0}^{n-1} a_i 2^i}_{\text{Sign-and-Magnitude significand}} \cdot 2^{-e_{m-1}2^{m-1} + \sum_{j=0}^{m-2} e_j 2^j}$$

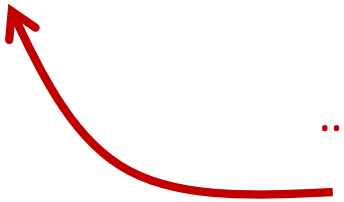
Floating Point

- **Large dynamic range** but **variable accuracy**
- Redundant unless **normalized**
- Not real numbers: **not associative!**
- Often exponent in **biased** signed representation
  - Zero can be represented by 0000…0000
  - Easier for comparisons and hardware implementations
- Often **normalized mantissa** $1 \leq m < 2$ with **hidden bit** (1.xxxxx)
- Today the **IEEE 754 standard** is almost universally adopted
- **x86/x64** supports FP through SSE/AVX extensions (since 1999)
- **RISC-V** supports FP through ISA extensions (not used in CS-200)

Example
Sign-and-Magnitude Addition

- Write a function in RISC-V assembler to sum two **32-bit signed numbers** represented in **sign-and-magnitude (S&M) format** and produce the result also in sign-and-magnitude format
- The two operands are in registers `a0` and `a1` on entry and the result should be placed in register `a0`

- Ignore overflows



…or think about them as an additional exercise

Example
Solution: First Algorithm

- **Do like humans do**: look at the signs, perform an addition or subtraction as required, and decide the final sign
- Basic algorithm

  - If the operands have the **same sign**
    - **Add** the absolute values
    - Attach to the result the **same sign** as the operands
  - If the operands have **different sign**
    - Identify the **largest value** in absolute value
    - **Subtract** the smallest absolute value from the largest one
    - Attach to the result the **sign of the largest value**

- This method is left as an additional exercise

Example
Solution: Second Algorithm

- **Exploit what we have**: implement conversions between the two representations
- Basic algorithm

  - **Convert** the two operands **from sign-and-magnitude to 2's complement**
  - **Add** the two operands
  - **Convert** the result **from 2's complement to sign-and-magnitude**

Example
Solution: Converting S&M to 2's Comp

- Algorithm
  - If the value is positive, no conversion is needed
  - If the value is negative, one should
    - Find the S&M opposite (positive and therefore correctly represented also in 2's comp)
    - Find the 2's comp opposite (negative, as required)

Example
Solution: Checking S&M Signs

- The S&M sign can be checked in many ways
  - Testing bit 31 by right shift

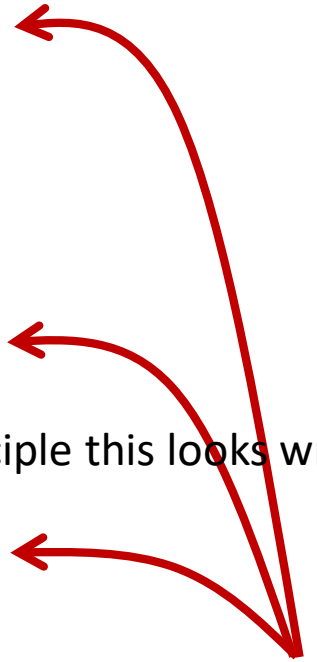
```
srli    t0, a0, 31          # srai would be fine too
beqz    t0, positive
```

  - Testing bit 31 by masking

```
lui     t1, 0x80000
and     t0, a0, t1
beqz    t0, positive
```

  - Comparing to zero (in principle this looks wrong, because **bgez** expects a 2's comp number, but…)

```
bgez    t0, positive
```



We should check that we are treating "minus 0" correctly

Example
Solution: Finding the S&M Opposite

- Once we know that the S&M number is negative, the sign can be changed in a few of ways
  - Inverting bit 31 with a mask (a real sign change)

```
lui     t1, 0x80000
xor     a0, a0, t1
```

  - Clearing bit 31 with a mask (forcing a number to be positive)

```
lui     t1, 0x80000
not     t1, t1
and     a0, a0, t1
```

  - Removing bit 31 by two shifts (forcing a number to be positive)

```
slli    a0, a0, 1
srli    a0, a0, 1               # srai would be wrong!
```

Example
Solution: Finding the 2's Comp Opposite

- Once we have the absolute value, we can find the opposite in 2's comp in three ways
  - Subtracting from zero (which is ok, because `a0` being positive, one can think of it as being in 2's comp)

```
neg     a0, a0          # same as sub a0, zero, a0
```

  - Using the relation $-A = \bar{A} + 1$

```
not     a0, a0
addi    a0, a0, 1
```

  - Using the relation $-A = \overline{A - 1}$

```
addi    a0, a0, -1
not     a0, a0
```

Example
Solution: Converting 2's Comp to S&M

- Algorithm
  - If the value is positive, no conversion is needed
  - If the value is negative, one should:
    - Find the 2's comp opposite (which is positive and therefore correctly represented also in S&M)
    - Find the S&M opposite

- Most steps are similar to those before
  - Checking the sign of a 2's comp is the same as an S&M
  - Finding the 2's comp opposite is exactly as before

Example
Solution: Finding the S&M Opposite

- If we know that a S&M number is positive, the sign can be changed in a few of ways
  - Inverting bit 31 with a mask (a real sign change)

```
lui     t1, 0x80000
xor     a0, a0, t1
```

  - Setting bit 31 with a mask (forcing a number to be positive)

```
lui     t1, 0x80000
or      a0, a0, t1
```

Example
Solution: Complete Program

```
add_sandm:
        lui     t1, 0x80000             # a mask for the sign bit
        and     t0, a0, t1
        beqz    t0, a0_positive
        xor     a0, a0, t1
        neg     a0, a0

a0_positive:                            # now a0 is in 2's complement
        and     t0, a1, t1
        beqz    t0, a1_positive
        xor     a1, a1, t1
        neg     a1, a1

a1_positive:                            # now a1 too is in 2's complement
        add     a0, a0, a1              # perform the addition in 2's complement
        and     t0, a0, t1
        beqz    t0, sum_positive
        neg     a1, a1
        xor     a0, a0, t1

sum_positive:                           # now a0 is in sign-and-magnitude
        ret
```

References

- Patterson & Hennessy, COD – RISC-V Edition
  - **Chapter 2** and, in particular, **Section 2.4**
  - **Chapter 3** and, in particular, **Section 3.2**