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)

Binary, 2's complement

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

Binary

- Individual digits (bits)

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

Simply 100010
if the digits are known

- Digit string (representation)

$$\langle a_{n-1} a_{n-2} \dots a_2 a_1 a_0 \rangle$$

Numbers

We usually care for three types of numbers:

- **Integers** (signed and unsigned)

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

- **Fixed Point**

$$0\underline{.12}, 3\underline{.14}, 1073741823\underline{.75}$$

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

- **Floating Point**

$$3.14E3, -2.5E1, 1.0E0, 4.2E-2, -1.5E-3$$

Unsigned Integers

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

- Definition:

If R = 2, binary

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

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:

$$+100, -2345$$

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

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

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

- Definition:

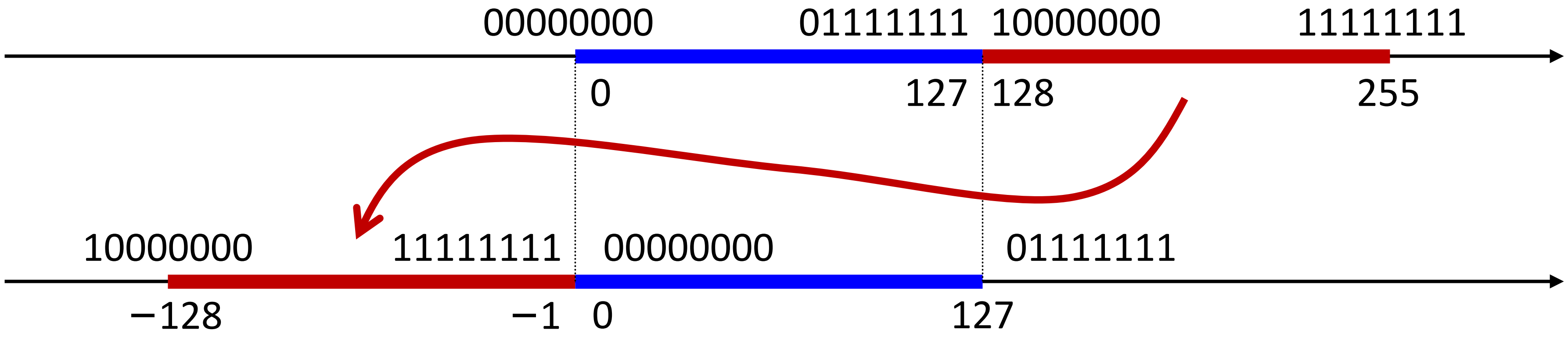
If R = 2, binary

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

0 or 1

Radix's Complement

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



R = 2
n = 8

- Property when R = 2:

$$A = \langle a_{n-1}a_{n-2}\ldots a_2a_1a_0\rangle = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

Radix's Complement

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

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

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

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

2's Complement from Subtraction

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

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

2’s Complement from Subtraction

- Consider a “normal” **paper-and-pencil subtraction**

	-1	-1	-1				-1		
	0	0	0	0	1	0	1	0_2	10_{10}
−	0	0	0	1	0	0	0	1_2	17_{10}
…	…	1	1	1	1	0	0	1_2	
					↓				
	$\boxed{-1}$	1	1	1	1	0	0	1_2	
	$\boxed{-2^7}$	$+2^6$	$+2^5$	$+2^4$	$+2^3$			$+2^0$	-7_{10}

Stop and “accept” the −1…

A sign bit

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

				1	1	0	1_2	4 bits
				$\boxed{-2^3}$	$+2^2$		$+2^0$	-3_{10}
=								
1	1	1	1	1	1	0	1_2	8 bits
-2^7	$+2^6$	$+2^5$	$+2^4$	$+2^3$	$+2^2$		$+2^0$	-3_{10}

Truncation is legal only if you remove copies of the final sign bit

Instructions for Signed Numbers

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

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

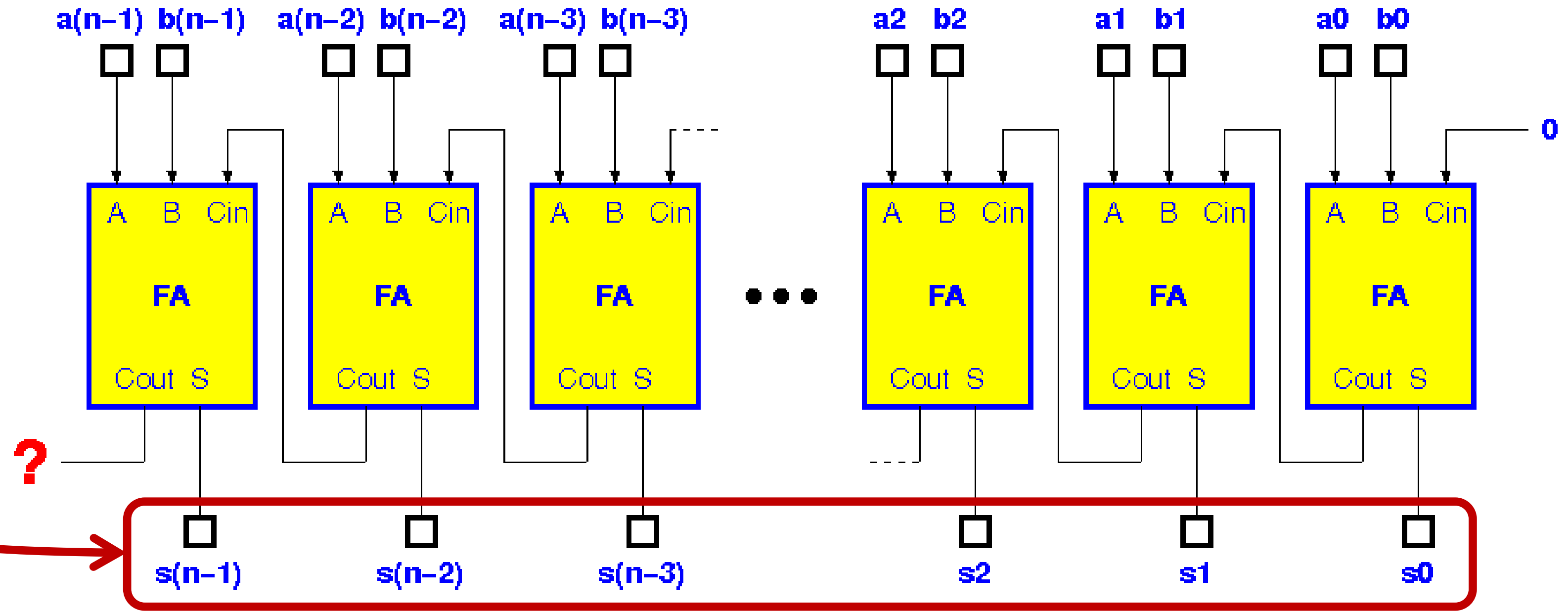
$1110_2/2 = 0111_2$
but
$1110_{2c}/2 = 1111_{2c}$

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

Overflows in 2's Complement Addition

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

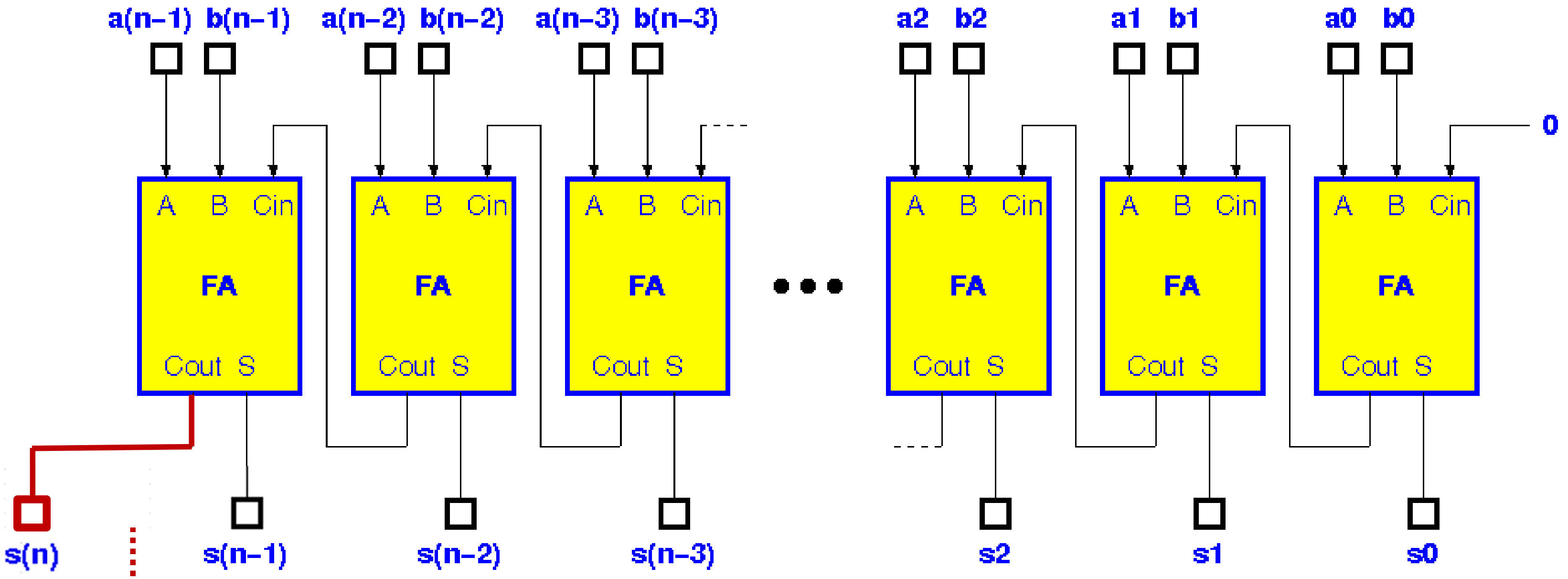
These bits are all identical to unsigned addition…



…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

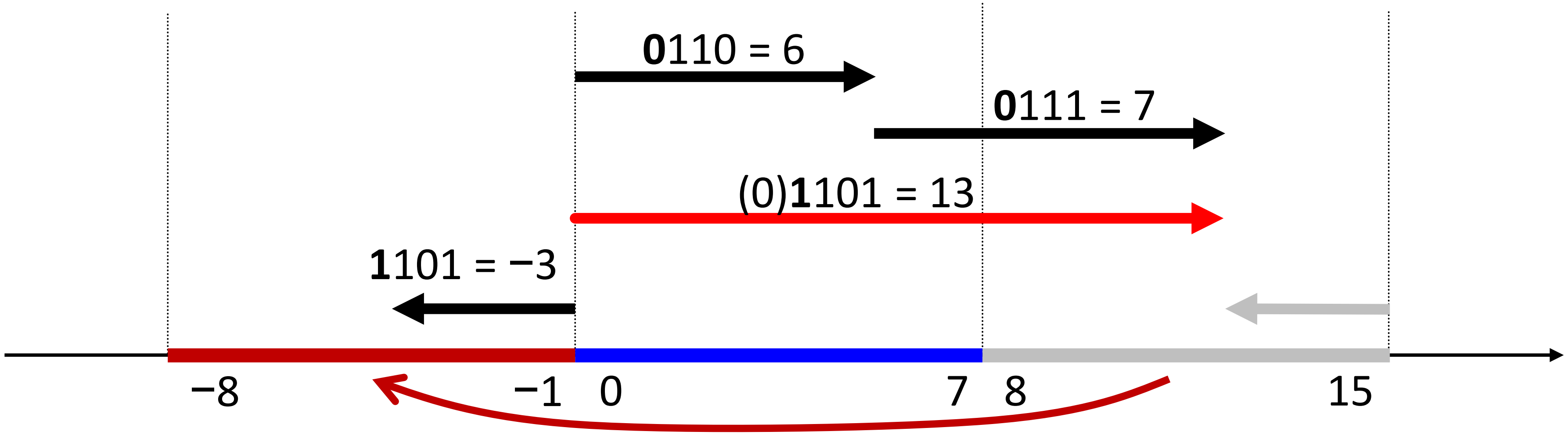
No 1 0

In 2’s complement, the carry bit must be **equal to the next bit**

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

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

- A "strange" but **very useful property**

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

- Not too hard to prove

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

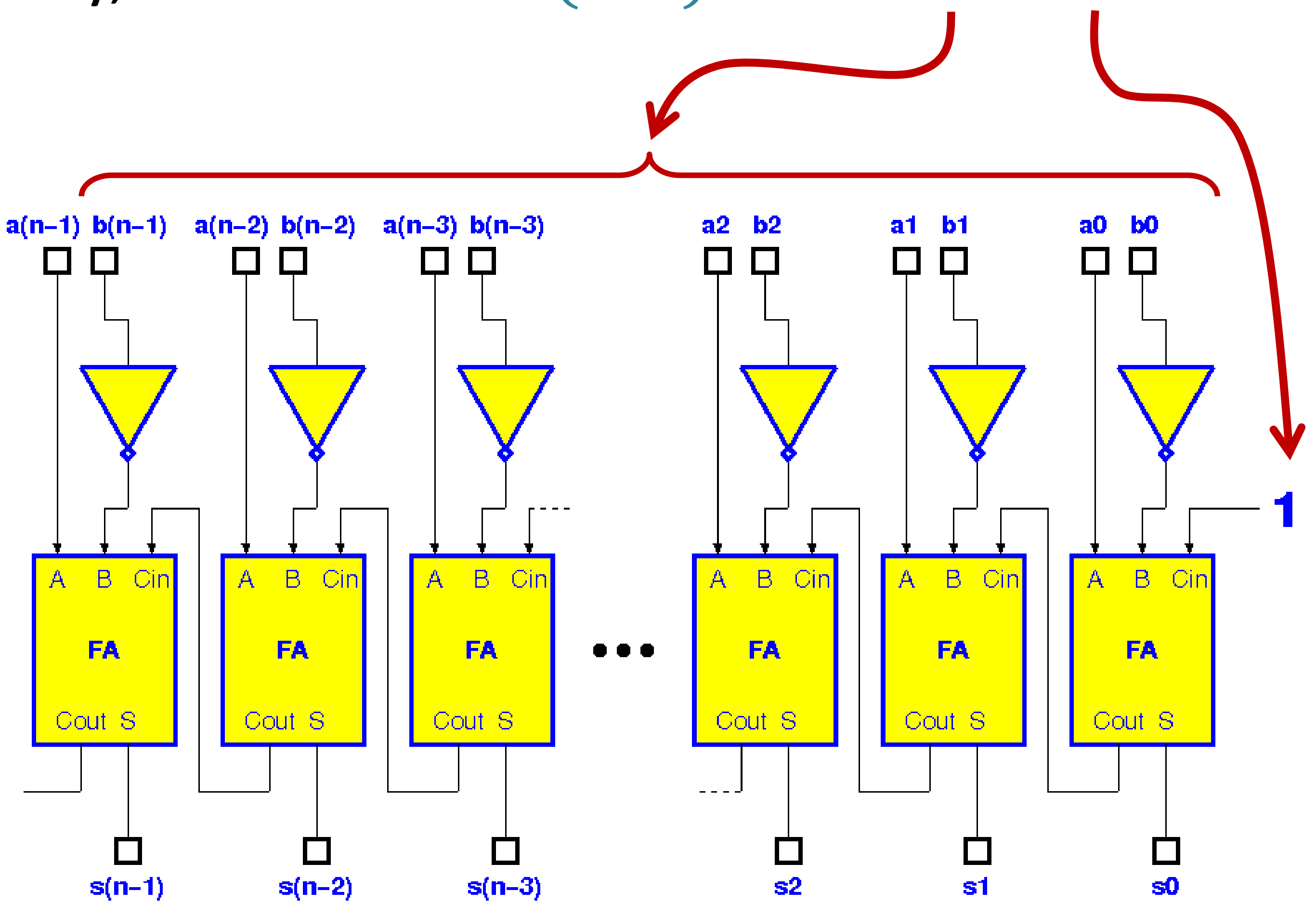
$$= -(a_{n-1} + \overline{a_{n-1}}) \cdot 2^{n-1} + \sum_{i=0}^{n-2} (a_i + \overline{a_i}) \cdot 2^i = -2^{n-1} + \sum_{i=0}^{n-2} 2^i = -1$$

- Also somehow intuitive

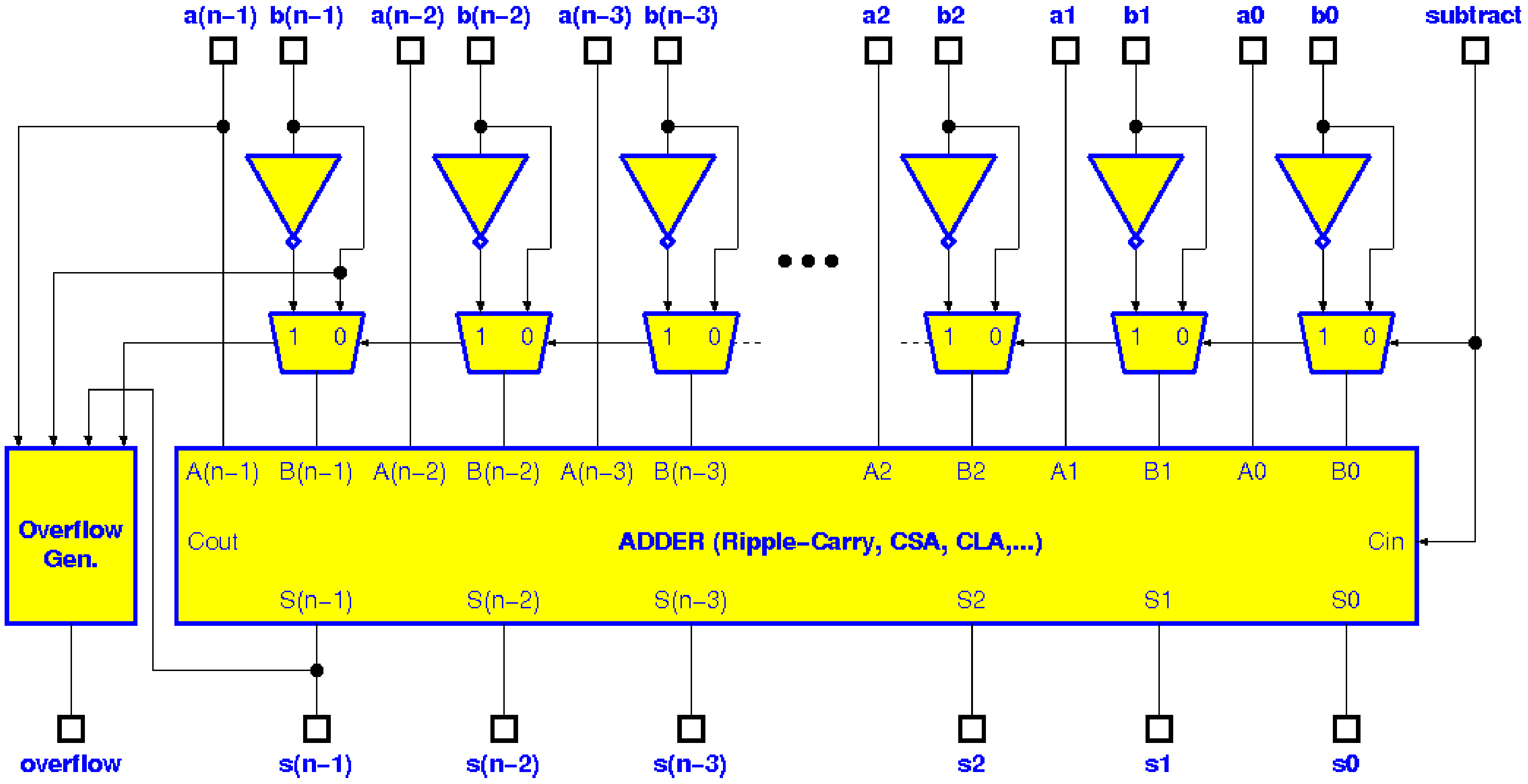
$$\begin{array}{lccccccccc} A & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & + \\ \bar{A} & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & = \\ \hline -1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \end{array}$$

Two's Complement Subtractor

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



Two's Complement Add/Subtract Units



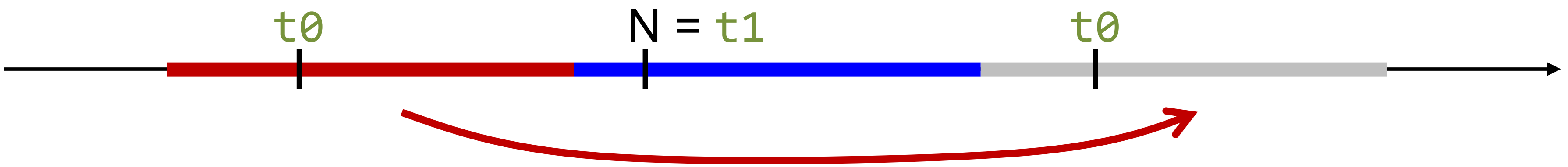
Fun Stuff: Bounds Check

- 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

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

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** (<u>1.</u>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

References

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