

Course Info

- Lab 4 is released, get yourself prepared before going to lab sessions!
- Project 1.1 available, and will be marked in lab sessions. Deadline March 13th.
- HW3 available on piazza, ddl March 18th.
- Discussion on CALL.



信息科学与技术学院

School of Information Science and Technology

CS 110

Computer Architecture

RISC-V M&F Extension

Instructors:

Siting Liu & Chundong Wang

Course website: <https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2023/index.html>

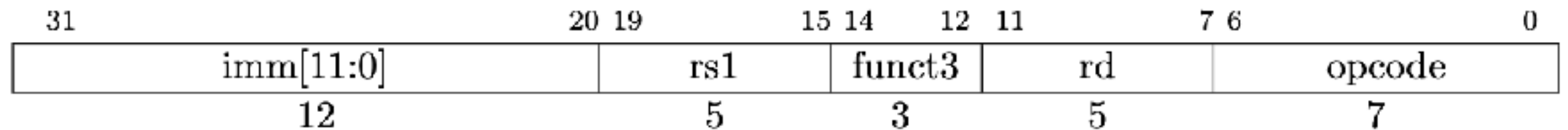
School of Information Science and Technology (SIST)

ShanghaiTech University

2023/2/6

Correction: Return—Jump

- JALR: Jump & Link Register
- Unconditional jump (**I-type**)



`j alr rd imm(rs1)`

1. Jump to (`imm+rs1`), `rs1` can be the return address we just saved to `ra`
2. Save return address (`PC+4`) to `rd`

Base	Version	Status
RVWMO	2.0	Ratified
RV32I	2.1	Ratified
RV64I	2.1	Ratified
<i>RV32E</i>	<i>1.9</i>	<i>Draft</i>
<i>RV128I</i>	<i>1.7</i>	<i>Draft</i>
Extension	Version	Status
M	2.0	Ratified
A	2.1	Ratified
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
C	2.0	Ratified
<i>Counters</i>	<i>2.0</i>	<i>Draft</i>
<i>L</i>	<i>0.0</i>	<i>Draft</i>
<i>B</i>	<i>0.0</i>	<i>Draft</i>
<i>J</i>	<i>0.0</i>	<i>Draft</i>
<i>T</i>	<i>0.0</i>	<i>Draft</i>
<i>P</i>	<i>0.2</i>	<i>Draft</i>
<i>V</i>	<i>0.7</i>	<i>Draft</i>
Zicsr	2.0	Ratified
Zifencei	2.0	Ratified
<i>Zam</i>	<i>0.1</i>	<i>Draft</i>
<i>Ztso</i>	<i>0.1</i>	<i>Frozen</i>

Subset	Name	Implies
Base ISA		
Integer	I	
Reduced Integer	E	
Standard Unprivileged Extensions		
Integer Multiplication and Division	M	
Atomics	A	
Single-Precision Floating-Point	F	Zicsr
Double-Precision Floating-Point	D	F
General	G	IMADZifencei
Quad-Precision Floating-Point	Q	D
Decimal Floating-Point	L	
16-bit Compressed Instructions	C	
Bit Manipulation	B	
Dynamic Languages	J	
Transactional Memory	T	
Packed-SIMD Extensions	P	
Vector Extensions	V	
User-Level Interrupts	N	
Control and Status Register Access	Zicsr	
Instruction-Fetch Fence	Zifencei	A
Misaligned Atomics	Zam	
Total Store Ordering	Ztso	
Standard Supervisor-Level Extensions		
Supervisor-level extension "def"	Sdef	
Standard Hypervisor-Level Extensions		
Hypervisor-level extension "ghi"	Hghi	
Standard Machine-Level Extensions		
Machine-level extension "jkl"	Zxmjkl	
Non-Standard Extensions		
Non-standard extension "mno"	Xmno	

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

Multiplicand	1000	8	t0
Multiplier	<u>x1011</u>	11	t1
	1000		
	1000		
	0000		
	<u>+1000</u>		
	01011000	88	t6

```

li t6,0 #initialize sum
li t2,1 #initialize param.
MULTI: and t3, t1, t2
      bne t2, t3, N_ACCUM
      add t6, t6, t0
N_ACCUM:
      srli t1, t1, 1
      slli t0, t0, 1
      beqz t1, EXIT
      j MULTI
EXIT:  ... ..
    
```

- $m \text{ bits} \times n \text{ bits} = m + n \text{ bit product}$

Integer Multiplication (1/3)

- Paper and pencil example (signed):

$$(a_n a_{n-1} \dots a_1 a_0)_2 = -a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \dots + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

Multiplicand	1000	-8
Multiplier	<u>x1001</u>	-7

Integer Multiplication (2/3)

- In RISC-V, we multiply registers, so:
 - 32-bit value x 32-bit value = 64-bit value
- Multiplication is not part of standard RISC-V because:
 - It requires a more complicated ALU (e.g. `and/or/add(i)/sub` etc. instructions supported by hardware)
 - RV32I is able to perform multiplication by shift-and-add, but sloooow
- Syntax of Multiplication (signed):
`mul rd, rs1, rs2`
`mulh rd, rs1, rs2`
 - Multiplies 32-bit values in those registers and returns either the lower or upper 32b result
 - If you do `mulh/mul` back to back, the architecture can fuse them
 - Also unsigned versions of the above

Integer Multiplication (3/3)

- Example:
 - In C: `int64_t a; int32_t b, c; a = b * c;`
 - These types are defined in C99, in `stdint.h`
- In RISC-V:
 - let `b` be `s2`; let `c` be `s3`; and let `a` be `s0` and `s1` (since it may be up to 64 bits)
 - `mulh s1, s2, s3`
 - `mul s0, s2, s3`

Integer Division (1/2)

- Paper and pencil example (unsigned):

– Quotient = 01001010 / 1000

– Remainder = 01001010 % 1000

Divisor	00001000	$\overline{01001010}$	Quotient
			Dividend

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

Integer Division (2/2)

- Syntax of Division (signed):
 - `div rd, rs1, rs2`
`rem rd, rs1, rs2`
- Divides 32-bit `rs1` by 32-bit `rs2`, returns the quotient (`/`) for `div`, remainder (`%`) for `rem`
- Again, can fuse two adjacent instructions
- Example in C: `a = c/d; b = c%d;`
- RISC-V:
 - `a↔s0; b↔s1; c↔s2; d↔s3`
 - `div s0, s2, s3`
`rem s1, s2, s3`

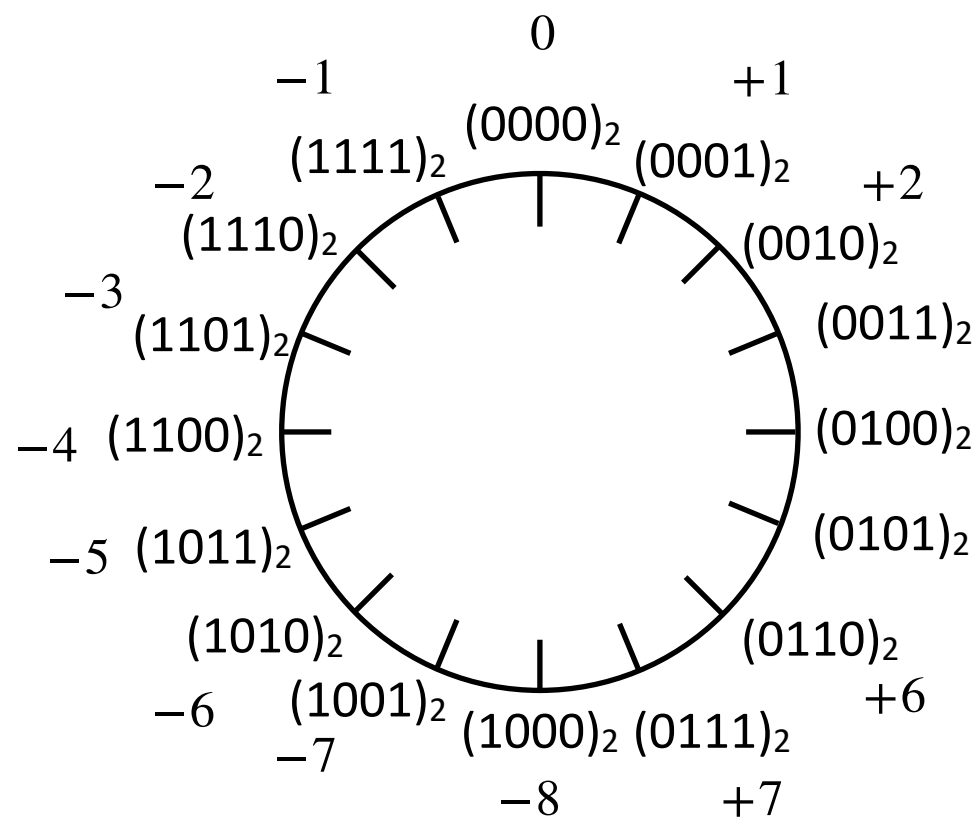
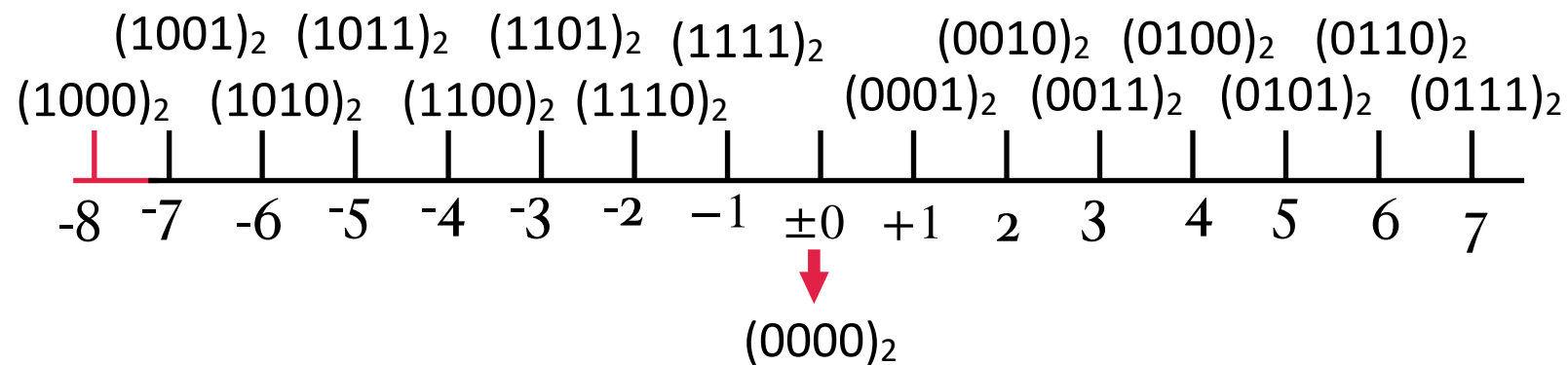
Notes on Optimization...

- A recommended convention
 - `mulh s1 s2 s3`
 `mul s0 s2 s3`
 - `div s0 s2 s3`
 `rem s1 s2 s3`
- Not a requirement but...
 - RISC-V says "if you do it this way, and the microarchitecture supports it, it can fuse the two operations into one"
 - Same logic behind much of the 16b ISA design:
If you follow the convention you can get significant optimizations

For more details, see: <https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf>

RISC-V F-Extension Floating-Point Numbers

Signed & Unsigned Integer



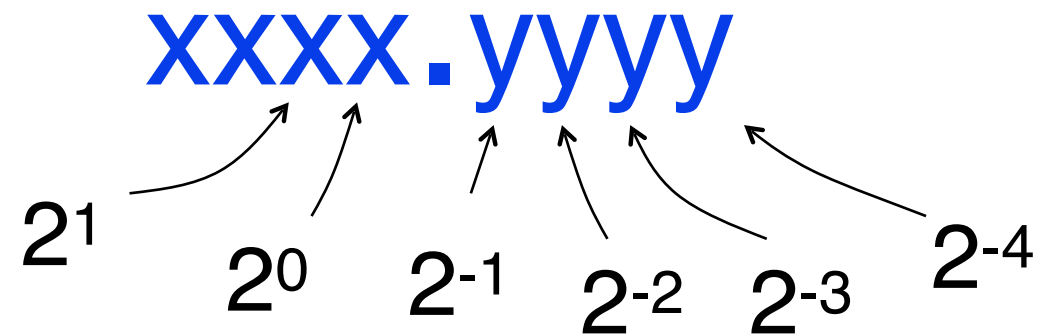
- 2's complement number $(a_n a_{n-1} \dots a_1 a_0)_2$ represents

$$(a_n a_{n-1} \dots a_1 a_0)_2 = -a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \dots + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

- Sign extension
- Arithmetics

Fractional

- “Binary Point” like decimal point signifies boundary between integer and fractional parts:



$$0010.1010_{\text{two}} = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{\text{ten}}$$

- Fixed-point: unsigned/signed number; 1234.5678

Fixed-Point Numbers

Addition is straightforward

$$\begin{array}{r}
 01.100 \\
 + 00.100 \\
 \hline
 10.000
 \end{array}
 \quad
 \begin{array}{r}
 1.5_{\text{ten}} \\
 + 0.5_{\text{ten}} \\
 \hline
 2.0_{\text{ten}}
 \end{array}$$

$$\begin{array}{r}
 01.100 \\
 00.100 \\
 \hline
 00 \ 000 \\
 000 \ 00 \\
 0110 \ 0 \\
 00000 \\
 00000 \\
 \hline
 0000110000
 \end{array}
 \quad
 \begin{array}{r}
 1.5_{\text{ten}} \\
 + 0.5_{\text{ten}} \\
 \hline
 2.0_{\text{ten}}
 \end{array}$$

Multiplication a bit more complex

(Need to remember where point is)

Are there any better methods for fraction arithmetic & represent small/large numbers?

Scientific Notation (in Decimal)

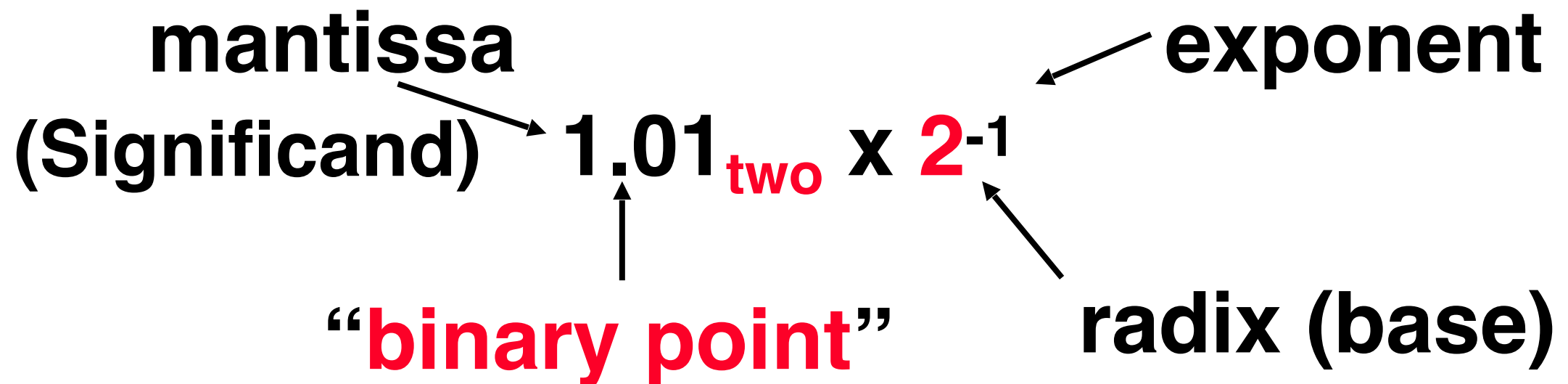
mantissa
(Significand) → 6.02_{ten} x 10²³
↑ decimal point
← exponent
← radix (base)

- Normalized form: no leading 0s
(exactly one digit to left of decimal point)
- Alternatives to representing 1/1,000,000,000

– Normalized: 1.0×10^{-9}

– Not normalized: $0.1 \times 10^{-8}, 10.0 \times 10^{-10}$

Scientific Notation (in Binary)



- Computer arithmetic that supports it called floating point, because it represents numbers where the binary point is not fixed, as it is for integers
 - Declare such variables in C as `float` (32b); `double` for double precision (64b).
 - How to represent in computer? Everything is a number.

Single-Precision 32-bit floating point (IEEE 754)



$$S = (-1)^{\text{Sign}}$$

$$E = \text{Exponent}_2 - 127_{10}$$

$$M = (1.\text{Mantissa})_2$$

$$\text{Value} = S \times M \times 2^E$$

<https://ieeexplore.ieee.org/document/8766229>

- **Biased exponent:** It can represent numbers in $[-127, 128]$, and allows comparing two floating point number easier (bit by bit) than the other representations.

Play with: <https://www.h-schmidt.net/FloatConverter/IEEE754.html>

Conversion

Example

0 01111011 110000000000000000000000

- Step 1: determine the sign

$$S = (-1)^{\text{Sign}} = (-1)^0 = 1 \text{ (positive)}$$

- Step 2: determine the unbiased exponent

$$E = \text{Exponent}_2 - 127_{10} = 01111011_2 - 127_{10} = 123 - 127 = -4$$

- Step 3: determine the Mantissa

$$M = 1.\text{Mantissa}_2 = 1.11_2 = 1.75_{10}$$

- Step 4: determine the converted decimal by using S , E and M

$$\text{decimal} = S \times M \times 2^E = 1 \times 1.75 \times 2^{-4} = 0.109375$$

Conversion

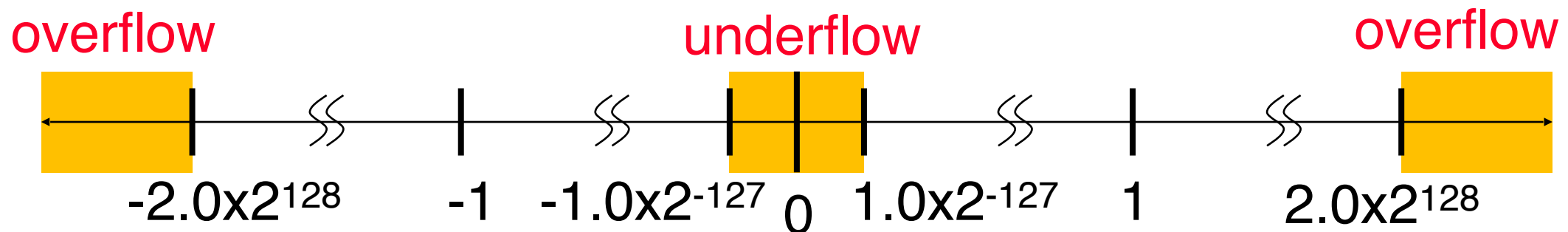
Example: 0.09375 into single precision floating point

- Step 1: determine the sign
 - Positive => Sign bit = 0
- Step 2: Convert the magnitude 0.09375 to binary
 - $0.09375_{10} = 0.00011_2$
- Step 3: Convert to scientific/normalized notation to obtain mantissa and unbiased exponent
 - $0.00011_2 = 1.1_2 \times 2^{-4}$
- Step 4: determine the biased exponent and remove the leading 1 from 1.1
 - Exponent = $-4_{10} + 127_{10} = 123_{10} = 01111011_2$, Mantissa = 1
- Step 5: padding 0s to the end of mantissa to make up to 23 bits/truncate if more than 23 bits

0 01111011 100000000000000000000000

Overflow vs. Underflow

- What if magnitude too large? ($> 2.0 \times 2^{128}$, $< -2.0 \times 2^{128}$)
 - **Overflow!** \Rightarrow Exponent larger than represented in 8-bit Exponent field
- What if result too small? (>0 & $< 1.0 \times 2^{-127}$, <0 & $> -1.0 \times 2^{-127}$)
 - **Underflow!** \Rightarrow Negative exponent larger than represented in 8-bit Exponent field (have method solving it partially later)



- What would help reduce chances of overflow and/or underflow?
- Double: 1 (sign)- 11(exponent, bias 1023)- 52(mantissa) in IEEE 754 standard

Representations for Special Cases

Exponent	Mantissa	Represented value
All ones	All zeros	$\pm \text{Inf}$
All ones	Not all zeros	Not a number (NaN)
All zeros	All zeros	$\pm \text{Zero}$
All zeros	Not all zeros	Sub/denormal

<https://ieeexplore.ieee.org/document/8766229> See sections 3.4, 6&7 for more details.

$$\pm \infty$$

- All exponent = 1s & All Mantissa = 0s
- Sign defined by the sign bit
- Valid Arithmetic
 - (+ - /) with finite numbers
 - Multiplication with finite/infinite non-zero numbers
 - Square root of $+\infty$
 - Conversion
 - `remainder(x, ∞)` for finite normal x
- Can be produced by
 - division ($x/0$, $x \neq 0$), `log(0)`, etc. along with `divideByZero` exception
 - Overflow with overflow exception

NaN (Not-a-Number)

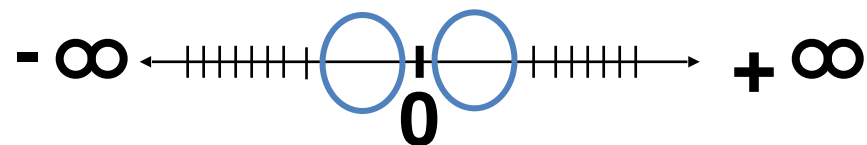
- Resulting from invalid operations (neither overflow nor underflow)
 - e.g. operations with NaN (quiet invalid operations, generally)
 - $0 \times \infty$, $\sqrt{n}(n < 0)$, $0/0$, ∞/∞ , magnitude subtraction of infinities (signaling invalid operation exception)
- Exponent all 1s, mantissa non-zero, sign don't-care
- Why NaN?
 - Represent missing values
 - Find sources of NaNs using signaling NaNs

Denorms.

- Denormalized numbers
- Gap between smallest positive/largest negative FP numbers and 0

Normalization and implicit 1 is to blame!

Gaps!



- We have not used exponent all 0s, mantissa $\neq 0$
- No implicit leading 1, implicit exponent = -126
- Extend smallest pos./largest neg. single-precision FP to $\pm 2^{-149}$ (non-zero)
- Followed by $\pm 2^{-148}$, $\pm 1.5 \times 2^{-148}$, $\pm 2^{-147}$, $\pm 2^{-147}$, $\pm 1.25 \times 2^{-147}$,
- Underflow still exists



Special Cases Summary

Exponent	Mantissa	Represented value
All ones	All zeros	Inf
All ones	Not all zeros	Not-a-Number (NaN)
All zeros	All zeros	Zero
All zeros	Not all zeros	Sub/denormal

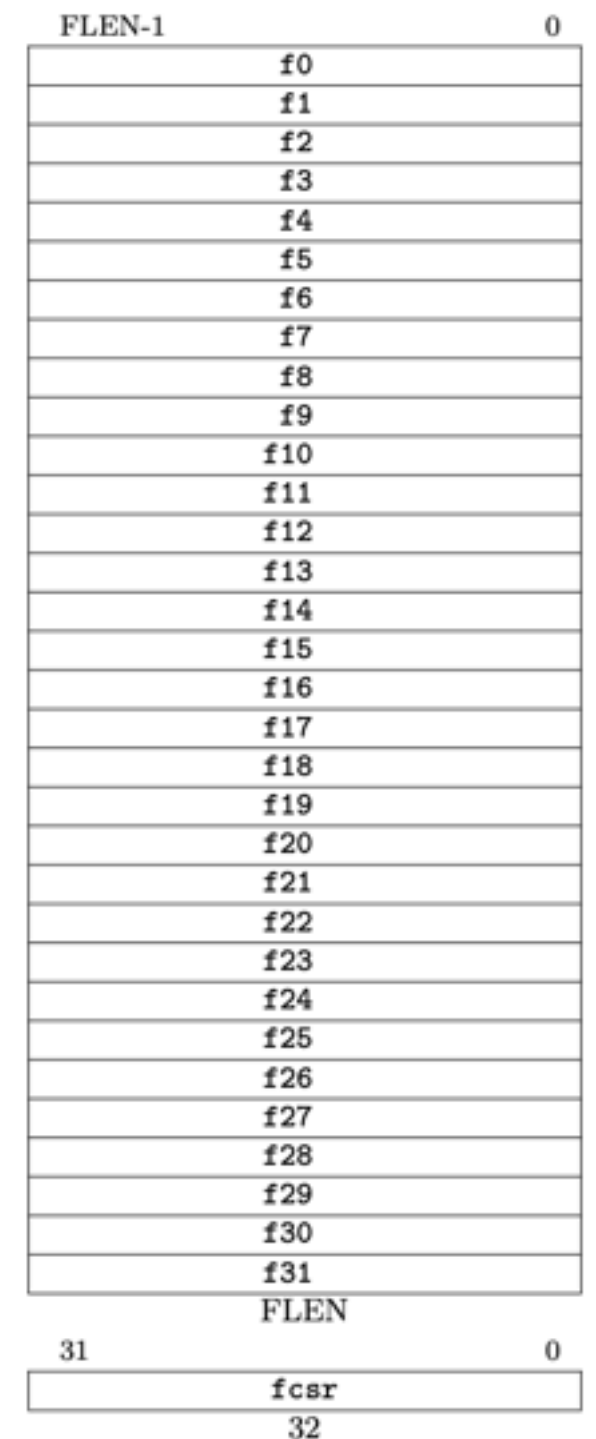
Normal numbers: Exponent 1-254, -126-127 after biasing

Rounding Modes

- Default to “round-to-nearest-ties-to-even” (avoid systematic biases)
- Other modes:
 - Round-to- $+\infty$ (up)
 - Round-to- $-\infty$ (down)
 - Round-towards-0
 - Round-to-nearest-ties-to-max-magnitude (roundTiesToAway, similar to SiSheWuRu)
- Used for FP arithmetics and FP-integer conversions

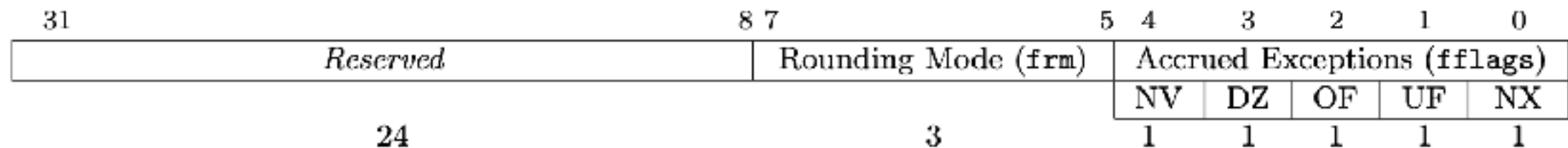
RISC-V F-Extension

- Comply IEEE 754-2008 standard
- Hardware consideration:
 - Add FP register file $f0-f31$, each 31-bit wide
 - Add FP control and status register, `fcsr`
- Add instructions for FP operations:
 - Load/ store – similar to int – e.g.:
 - `flw f1, 0(s1)` # load from address s1 to float reg 1
 - Arithmetic: append .s for “single precision”
 - `fsub.s f2, f3, f1`
 - Fused Multiply Add:
 - `Fmadd.s rd, rs1, rs2, rs3` # $[rd] = [rs1] * [rs2] + [rs3]$
 - Int / float conversions:
 - `fcvt.w.s f4, s4` # convert int in s4 to float in f4



RISC-V `fcsr` Register

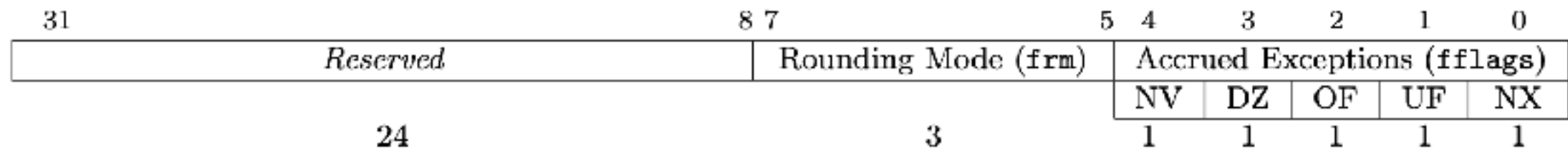
- `fcsr` register has 32 bits/3 fields



frm field	Mnemonic	Meaning
000	RNE	Round-to-nearest-tie-to-even
001	RTZ	Round-to-zero
010	RDN	Round-down
011	RUP	Round-up
100	RMM	Round-to-nearest-tie-to-max-magnitude
101		Reserved for future
110		Reserved for future
111	DYN	In instruction, select dynamic rounding; in RM register, invalid

RISC-V `fcsr` Register

- `fcsr` register has 32 bits/3 fields



Flag Mnemonic	Flag meaning
NV	Invalid operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

For more details, see: <https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf>
Chapter 11. “F” Standard Extension for Single-Precision Floating-Point, Version 2.2



信息科学与技术学院

School of Information Science and Technology

CS 110

Computer Architecture

Digital Circuits

Instructors:

Siting Liu & Chundong Wang

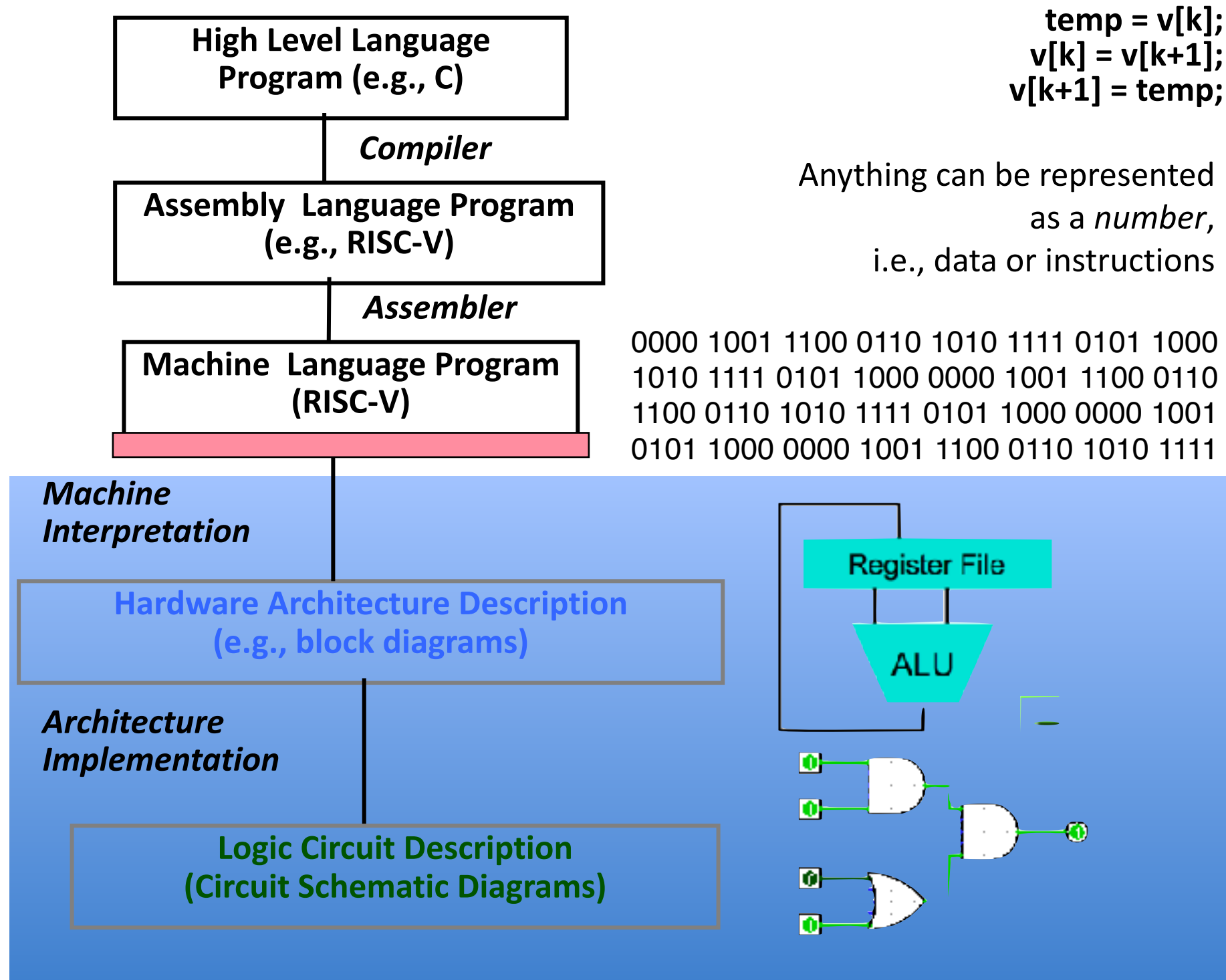
Course website: [https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/
Spring-2023/index.html](https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2023/index.html)

School of Information Science and Technology (SIST)

ShanghaiTech University

2023/3/5

Where are we?



Hardware Design

- Next several weeks: how a modern processor is built, starting with basic elements (transistors) as building blocks
- Why study hardware design?
 - Understand capabilities and limitations of HW in general and processors in particular
 - What processors can do fast and what they can't do fast (avoid slow things if you want your code to run fast!)
 - Background for more in-depth HW courses
 - Hard to know what you'll need for next 30 years
 - There is only so much you can do with standard processors: you may need to design own custom HW for extra performance
 - Even some commercial processors today have customizable hardware!
 - E.g. Google Tensor Processing Unit (TPU)

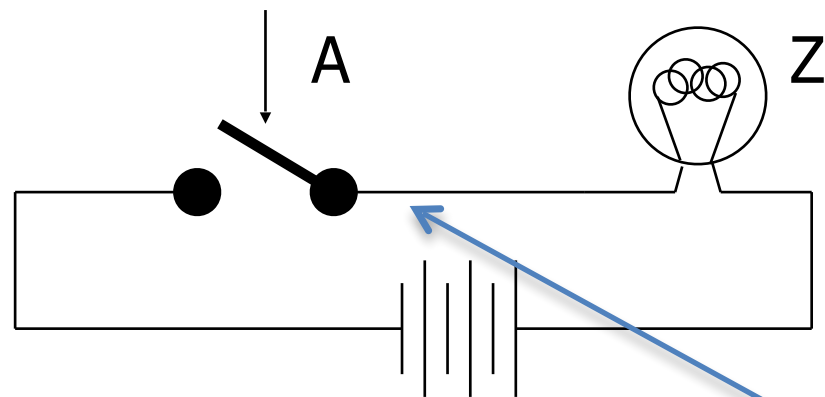
Hardware Revolution

Providers	CPU	GPU	FPGA*	ASIC (DSA)
Alibaba	X86/ARM/RISC-V	Nvidia/AMD	Intel/AMD	AliNPU
AWS (Amazon)	X86/Graviton (ARM)	Nvidia/AMD	AMD	Trainium
Azure (MS)	X86	Nvidia	Intel	N/A
Baidu	X86	Nvidia	AMD	Kunlun
Google	X86	Nvidia	N/A	TPU
Huawei	X86/Kunpeng (ARM)	Nvidia & Ascend	AMD	Ascend
Tencent	X86	Nvidia & Xinghai	AMD	Enflame (燧原)

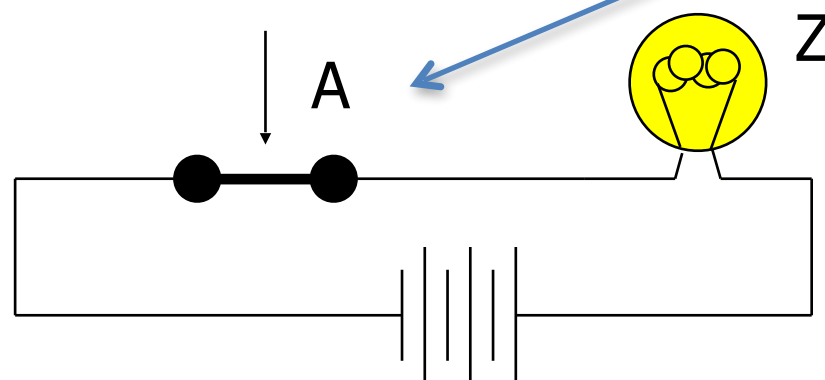
*Intel: formerly a.k.a. Altera
AMD: formerly a.k.a. Xilinx

Switches: Basic Element of Physical Implementations

- Implementing a simple circuit (arrow shows action if wire changes to “1” or is *asserted*):



On-switch (if A is “1” or asserted) turns-on light bulb (Z)

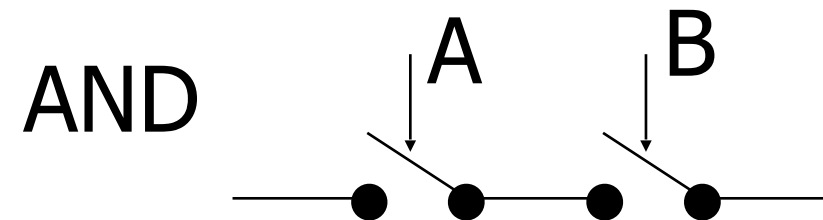


Off-switch (if A is “0” or unasserted) turns-off light bulb (Z)

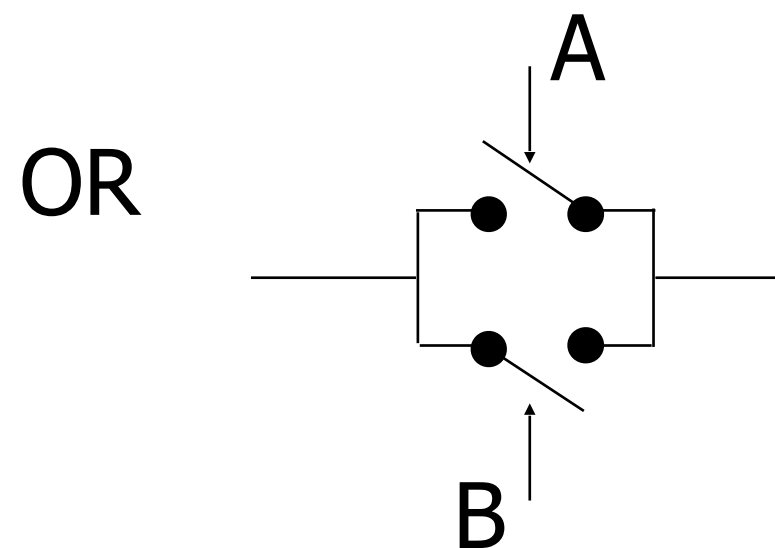
$$Z \equiv A$$

Switches

- Compose switches into more complex ones (Boolean functions):



$$Z \equiv A \text{ and } B$$



$$Z \equiv A \text{ or } B$$

Historical Note

- Early computer designers built ad hoc circuits from switches/relays (controllable switches)
- Began to notice common patterns in their work: ANDs, ORs, ...
- Master's thesis (by Claude Shannon, 1940) made link between work and 19th Century Mathematician George Boole
 - Called it “Boolean” in his honor
- Could apply math to give theory to hardware design, minimization, ...

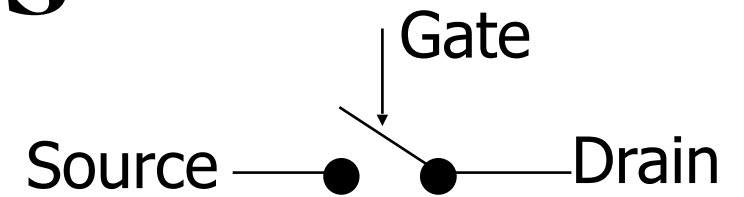


Modern Switches—Transistors

- High voltage (V_{dd}) represents 1, or true
 - In modern microprocessors, $V_{dd} \sim 1.0$ Volt
- Low voltage (0 Volt or Ground) represents 0, or false
- Pick a midpoint voltage (threshold) to decide if a 0 or a 1
 - Voltage greater than midpoint = 1
 - Voltage less than midpoint = 0
 - This removes noise as signals propagate – a big advantage of digital systems over analog systems
- If one switch can control another switch, we can build a computer!
- Our switches: CMOS transistors

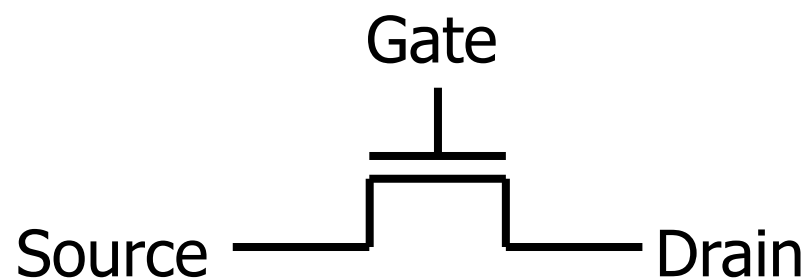
CMOS Transistors

- Three terminals: source, gate, and drain



- Switch action:

if voltage on gate terminal is (some amount) higher/lower than source terminal then conducting path established between drain and source terminals (switch is closed)



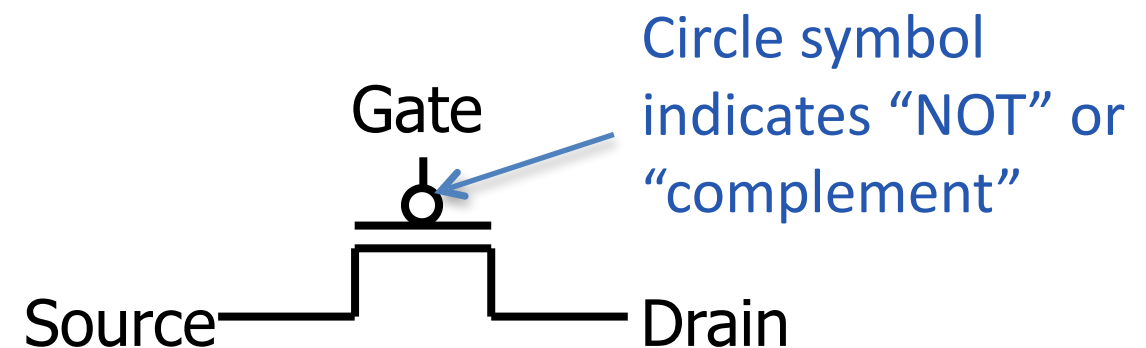
n-channel transistor

off when voltage at Gate is low

on when:

voltage (Gate) > voltage (Threshold)

(**High** resistance when gate voltage **Low**,
Low resistance when gate voltage **High**)



p-channel transistor

on when voltage at Gate is low

off when:

voltage (Gate) > voltage (Threshold)

(**Low** resistance when gate voltage **Low**,
High resistance when gate voltage **High**)

Recall Moore's Law

Moore's Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Our World
in Data

Transistor count

50,000,000,000

10,000,000,000

5,000,000,000

1,000,000,000

500,000,000

100,000,000

50,000,000

10,000,000

5,000,000

1,000,000

500,000

100,000

50,000

10,000

5,000

1,000

1970 1972 1974 1976 1978 1980 1982 1984 1986 1988 1990 1992 1994 1996 1998 2000 2002 2004 2006 2008 2010 2012 2014 2016 2018 2020

Year in which the microchip was first introduced

Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

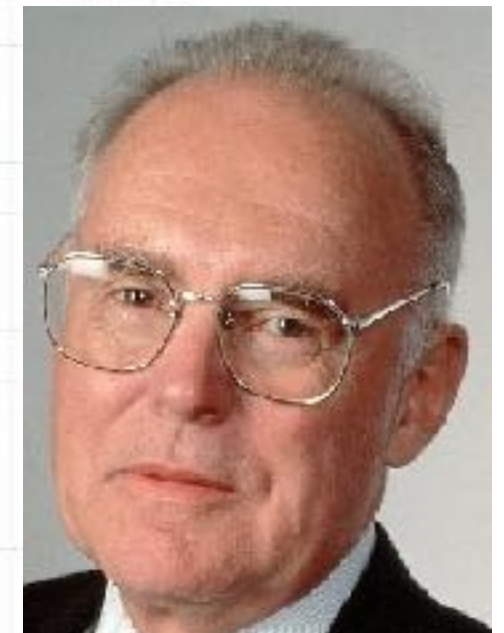
OurWorldinData.org - Research and data to make progress against the world's largest problems.

Licensed under CC-BY by the authors Hannah Fitchie and Max Roser.

40

**Predict based on observation:
2X Transistors / chip
every 2 years (18 months)**

Smaller transistors in old days:
Faster switching speed;
Lower power/energy consumption;
Lower cost per chip

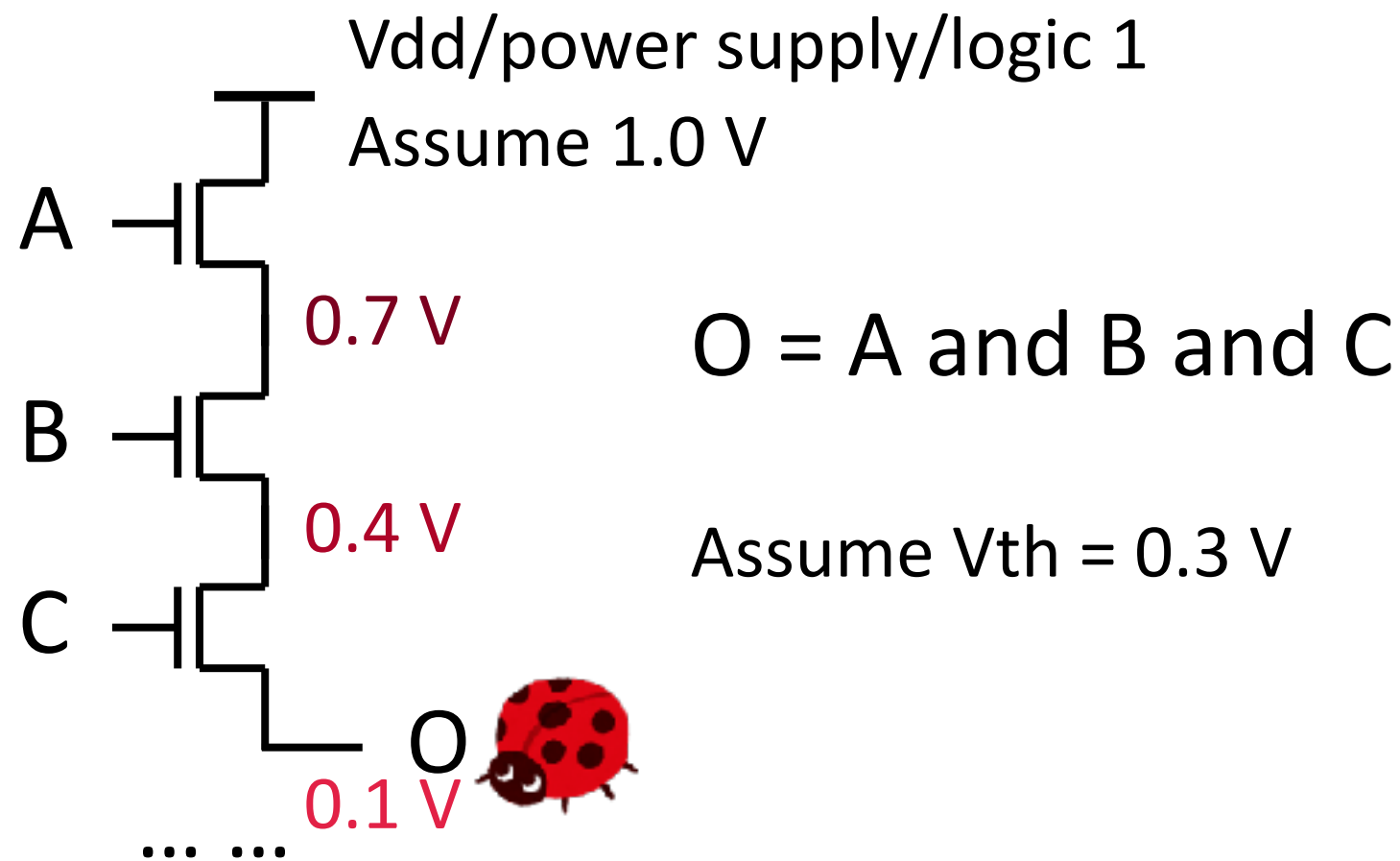


Gordon Moore

Building Logic Gates

CMOS Circuits

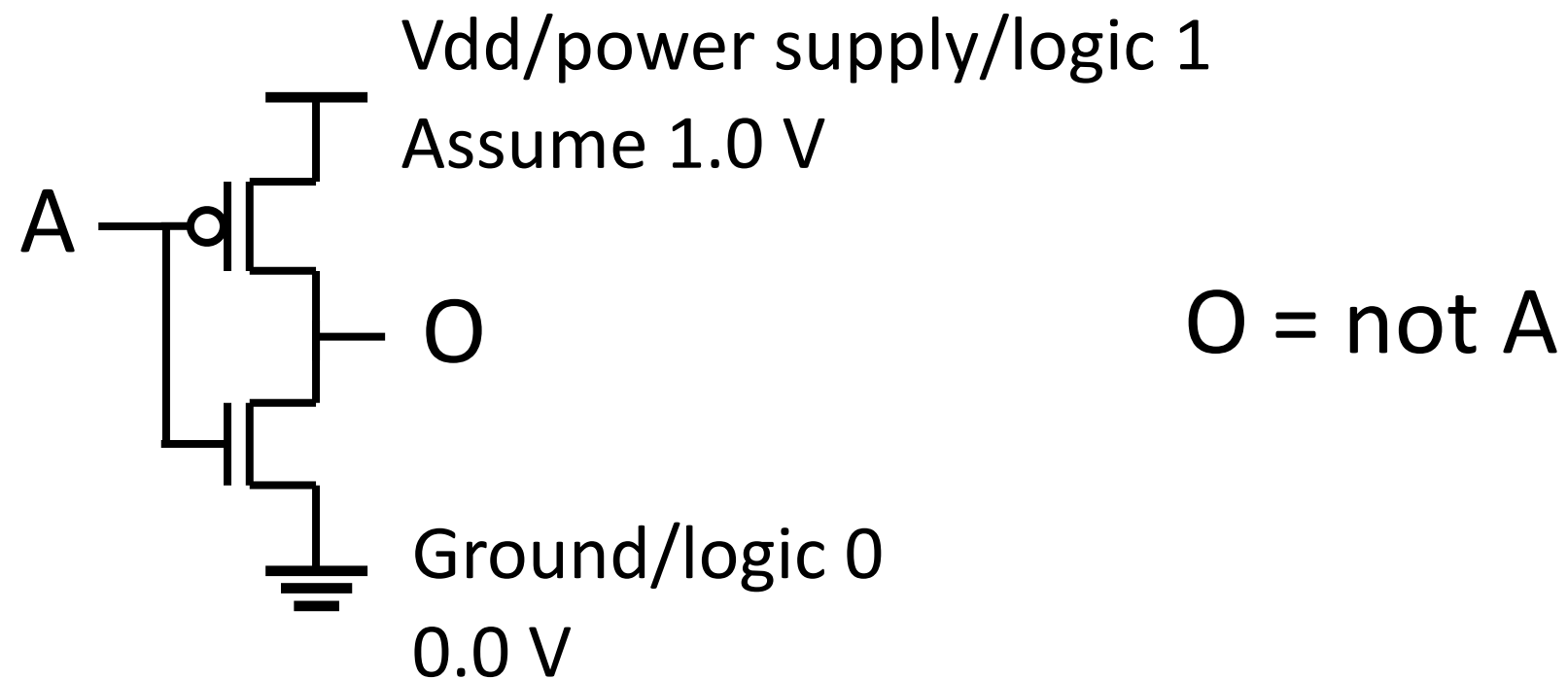
- N-type transistors (NMOS) pass weak 1 ($V_{dd} - V_{th}$) and strong 0
- P-type transistors (PMOS) pass weak 0 (V_{th}) and strong 1



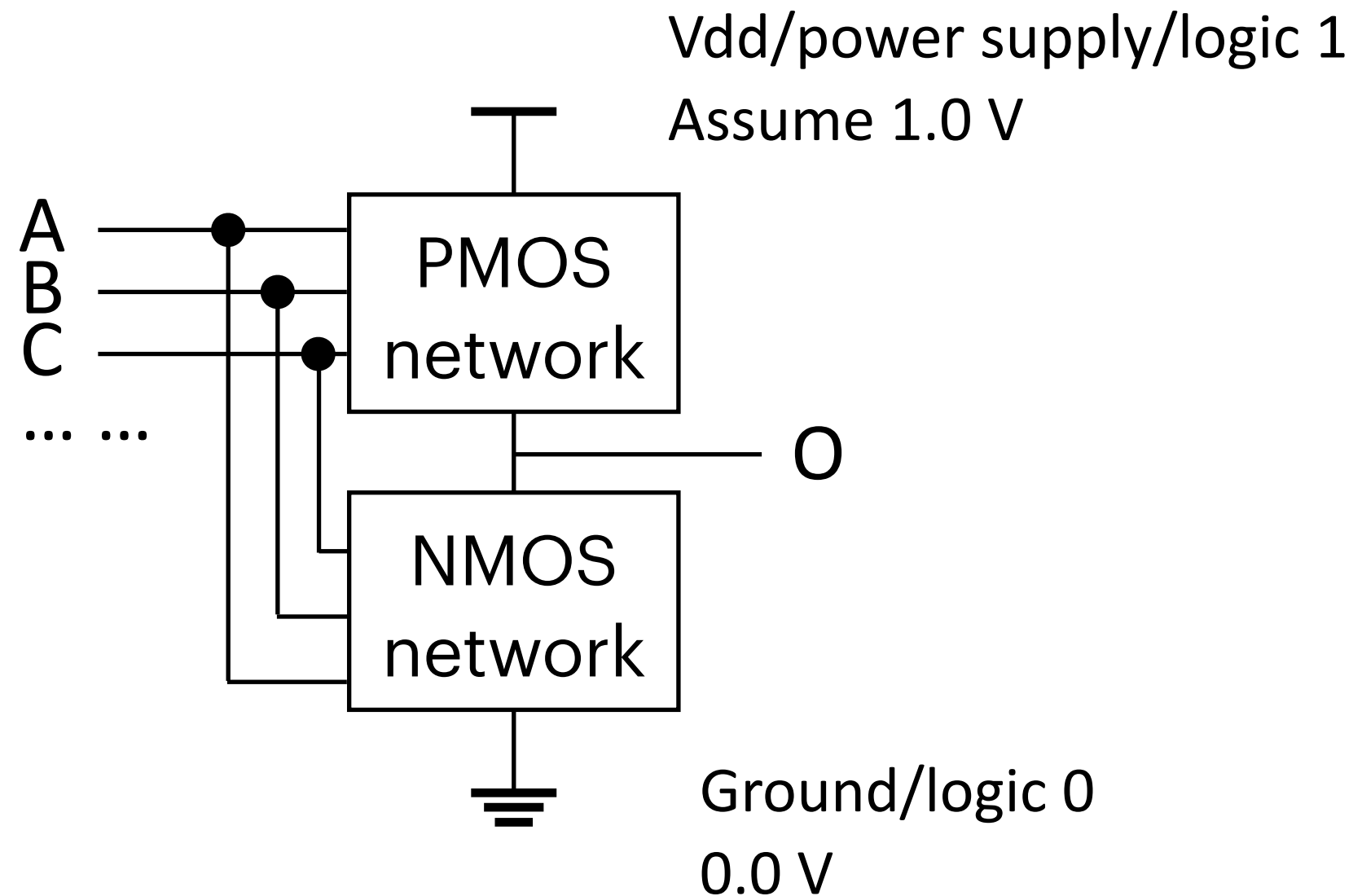
- “C” stands for complementary
- Pairs of N/P-type transistors to pass strong 0 and strong 1

The Simplest CMOS Circuits

- Inverter/Not gate



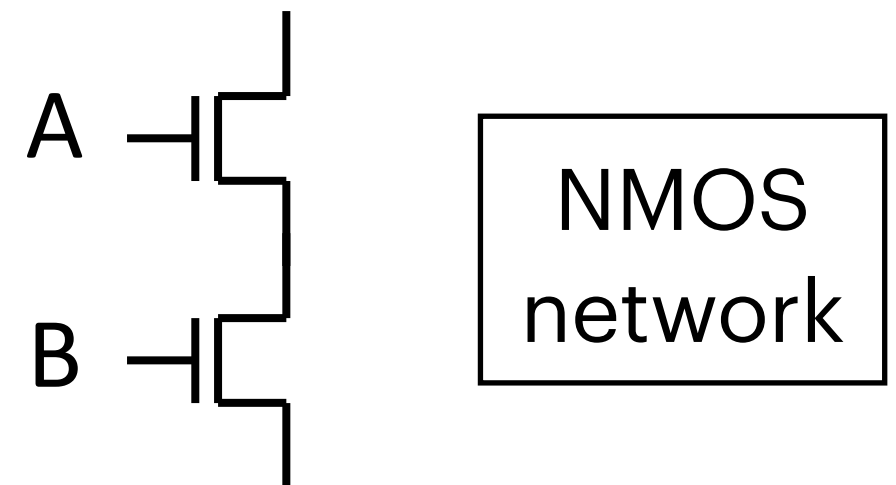
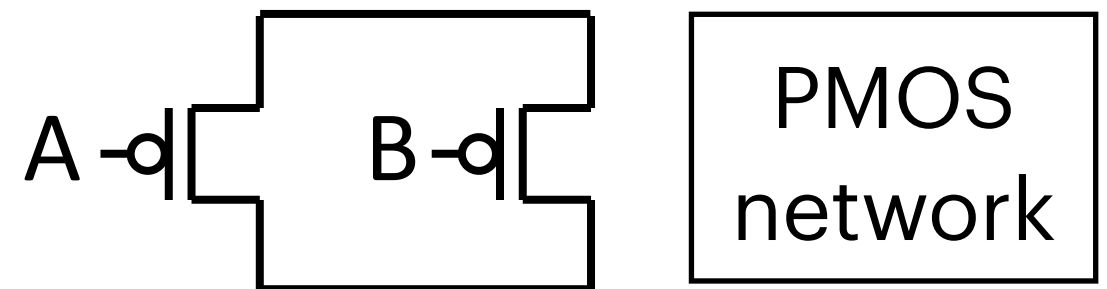
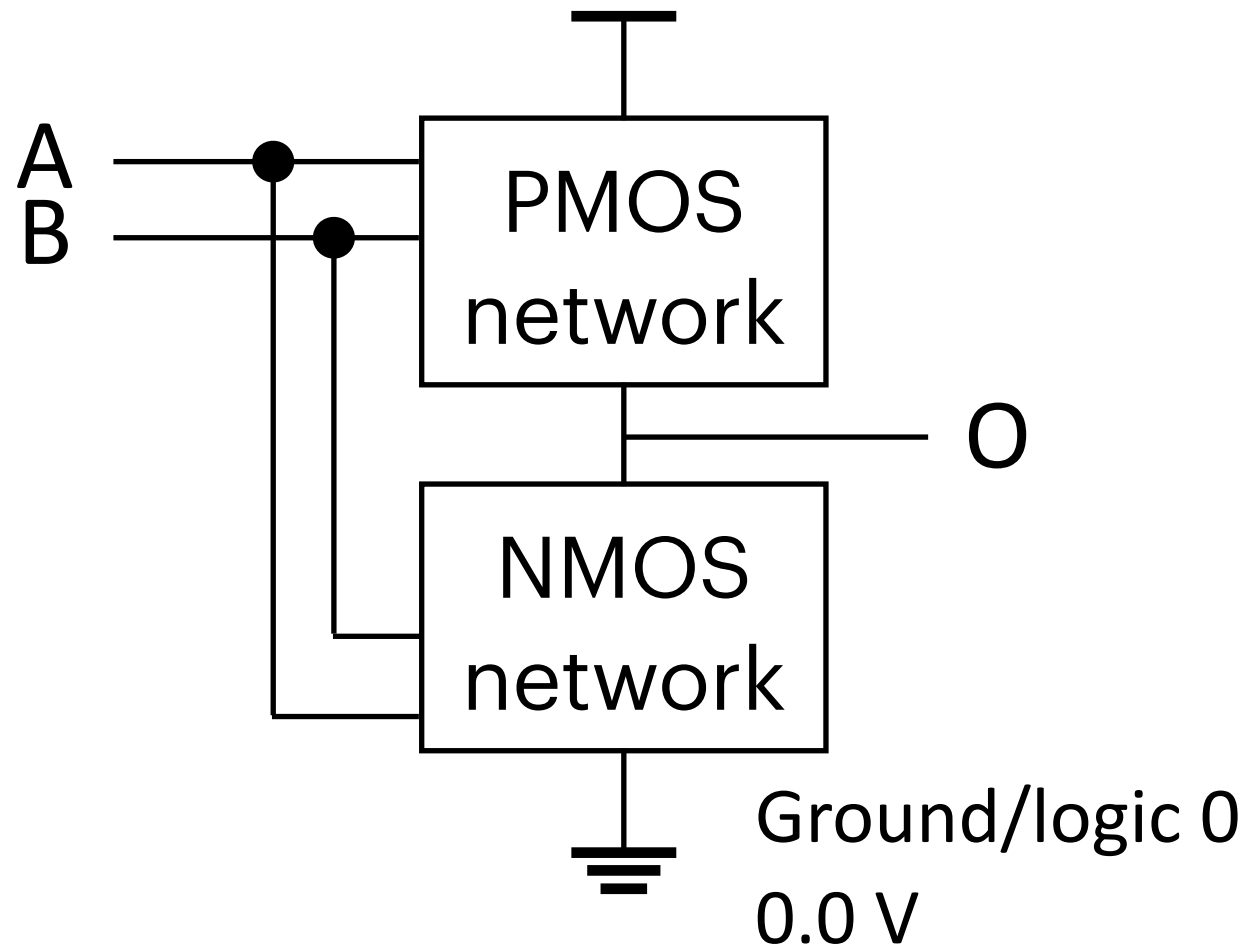
General CMOS Logic Gates



NAND Gate for Example

V_{dd}/power supply/logic 1

Assume 1.0 V



A	B	O
0	0	1
0	1	1
1	0	1
1	1	0

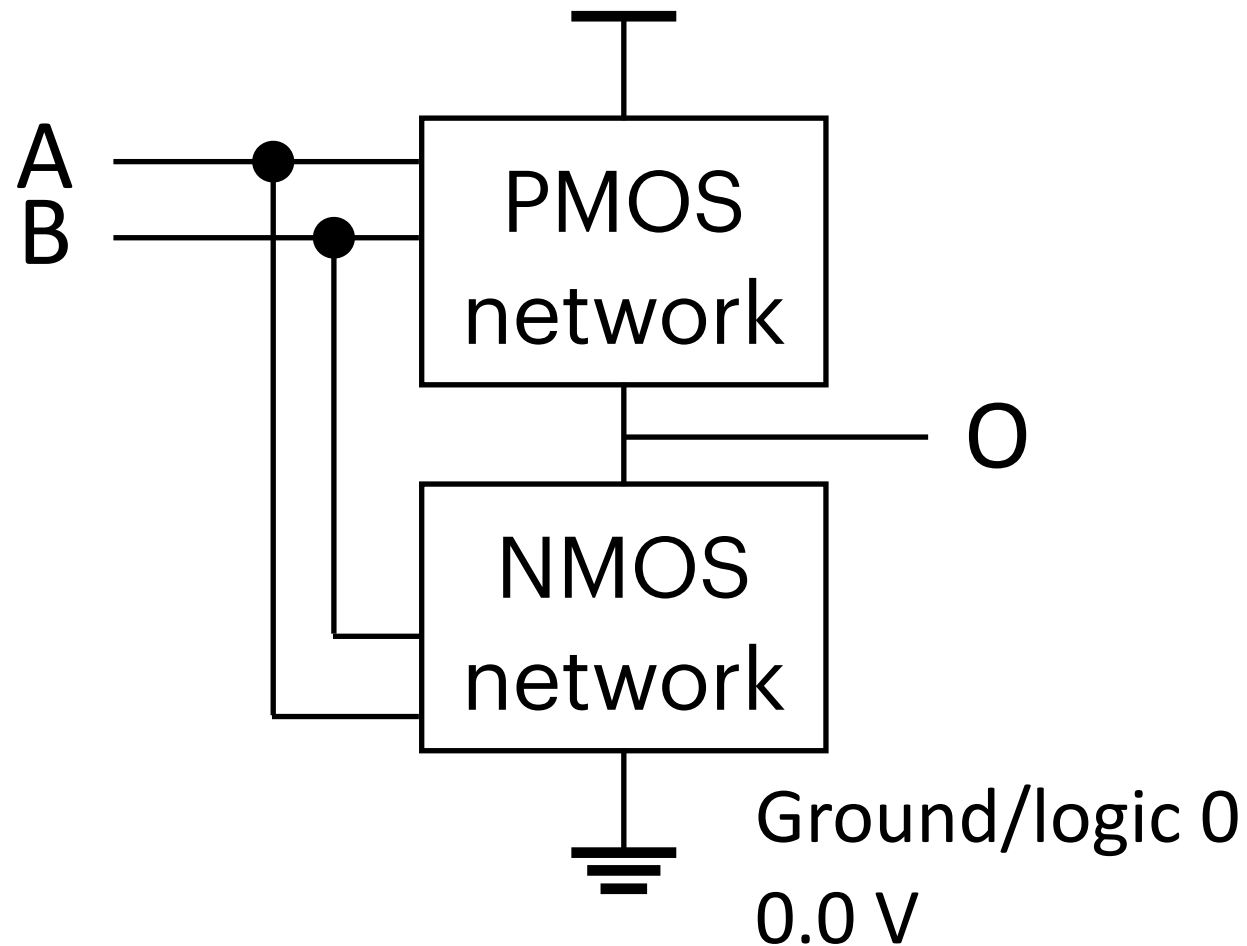
Truth table

What about 3-input NAND?

NOR Gate

V_{dd}/power supply/logic 1

Assume 1.0 V



PMOS
network

NMOS
network

A	B	O
0	0	1
0	1	0
1	0	0
1	1	0

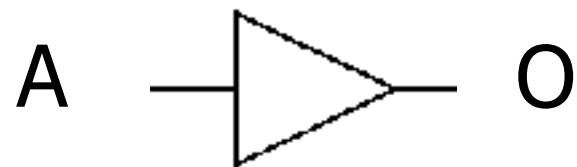
Truth table

What about 3-input NOR?

Basic Symbols

- Standard symbols for logic gates

– Buffer, NOT



– Universal sets

– NOT, AND, OR

Can be combined to implement any logics

– AND, NAND



– NAND

– OR, NOR



– NOR

Through Boolean algebra!

Boolean Algebra

- Use plus “+” for OR
 - “logical sum” $1+0 = 0+1 = 1$ (True); $1+1=2$ (True); $0+0 = 0$ (False)
- Use product for AND ($a \cdot b$ or implied via ab)
 - “logical product” $0*0 = 0*1 = 1*0 = 0$ (False); $1*1 = 1$ (True)
- “Bar” to mean complement (NOT)
- Thus
$$ab + a + \bar{c}$$
$$= a \cdot b + a + \bar{c}$$
$$= (a \text{ AND } b) \text{ OR } a \text{ OR } (\text{NOT } c)$$



Build Combinational Circuits with Basic Logic Gates

- Combinational circuits: the ones that the output of the digital circuits depends solely on its inputs; usually built with logic gates without feedback
- Step 1: Write down truth table of the desired logic

For example build an XOR
with AND/OR/NOT

A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

Build Combinational Circuits with Basic Logic Gates

- Combinational circuits: the ones that the output of the digital circuits depends solely on its inputs; usually built with logic gates without feedback
 - Step 2: Pick the lines with 1 as the output; write them down in *Sum of Minterms (Product)* form;

For example build an XOR
with AND/OR/NOT

A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

Minterms

$\bar{A}\bar{B}$	m_0
$\bar{A}B$	m_1
$A\bar{B}$	m_2
AB	m_3

Build Combinational Circuits with Basic Logic Gates

- Combinational circuits: the ones that the output of the digital circuits depends solely on its inputs; usually built with logic gates without feedback
- Step 3: Simplify using Laws of Boolean algebra;

For example build an XOR
with AND/OR/NOT

A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

Laws of Boolean Algebra

$$X \bar{X} = 0$$

$$X 0 = 0$$

$$X 1 = X$$

$$X X = X$$

$$X Y = Y X$$

$$(X Y) Z = X (Y Z)$$

$$X (Y + Z) = X Y + X Z$$

$$X Y + X = X$$

$$\bar{X} Y + X = X + Y$$

$$\overline{X Y} = \bar{X} + \bar{Y}$$

$$X + \bar{X} = 1$$

$$X + 1 = 1$$

$$X + 0 = X$$

$$X + X = X$$

$$X + Y = Y + X$$

$$(X + Y) + Z = X + (Y + Z)$$

$$X + Y Z = (X + Y) (X + Z)$$

$$(X + Y) X = X$$

$$(\bar{X} + Y) X = X Y$$

$$\overline{X + Y} = \bar{X} \bar{Y}$$

Complementarity

Laws of 0's and 1's

Identities

Idempotent Laws

Commutativity

Associativity

Distribution

Uniting Theorem

Uniting Theorem v. 2

DeMorgan's Law

Your turn!

- Build a half adder:

	Sum	Carry
• $0 + 0 = 0$	0	
• $0 + 1 = 1$	1	0
• $1 + 0 = 1$	1	0
• $1 + 1 = 0$	0	1

- Build a 2-bit adder:

	Sum	Carry
• $00 + 00 = 00$	00	0
• $00 + 01 = 01$	01	0
• $00 + 10 = 10$	10	0
• $00 + 11 = 11$	11	0
• $01 + 00 = 01$	01	0
• $01 + 01 = 10$	10	0
• $01 + 10 = 11$	11	0
• $01 + 11 = 00$	00	1

AB CD

	Sum	Carry
• $10 + 00 = 10$	10	0
• $10 + 01 = 11$	11	0
• $10 + 10 = 00$	00	1
• $10 + 11 = 01$	01	1
• $11 + 00 = 11$	11	0
• $11 + 01 = 00$	00	1
• $11 + 10 = 01$	01	1
• $11 + 11 = 10$	10	1

Another Simplification Method —Karnaugh Map

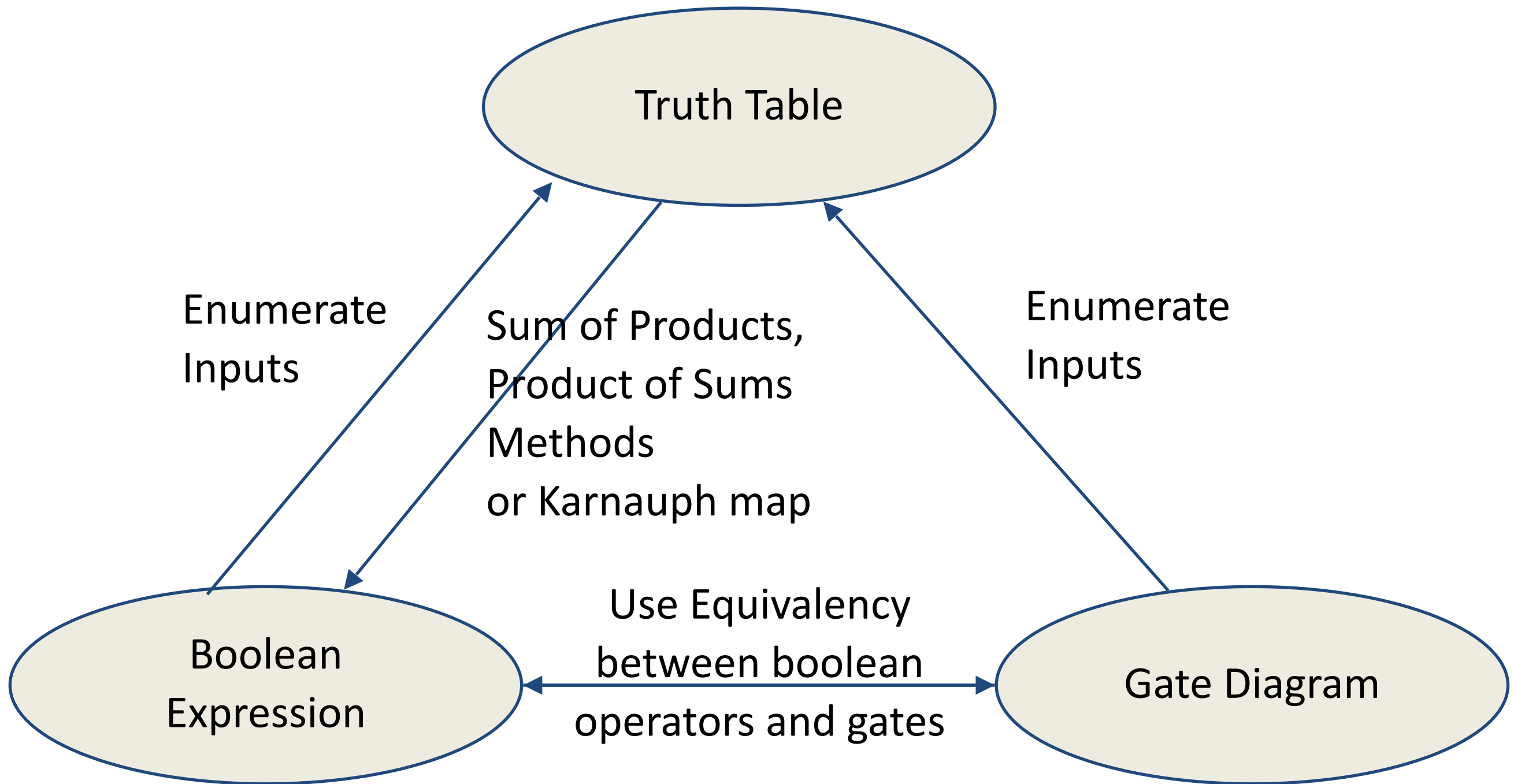
		A B		Gray coded	
		00	01	11	10
CD	00				
	01			1	
	11		1	1	1
	10			1	1

Gray coded

Each cell corresponds to a minterm

Online Karnaugh map solver: <http://www.32x8.com/index.html>

Representations of Combinational Logic

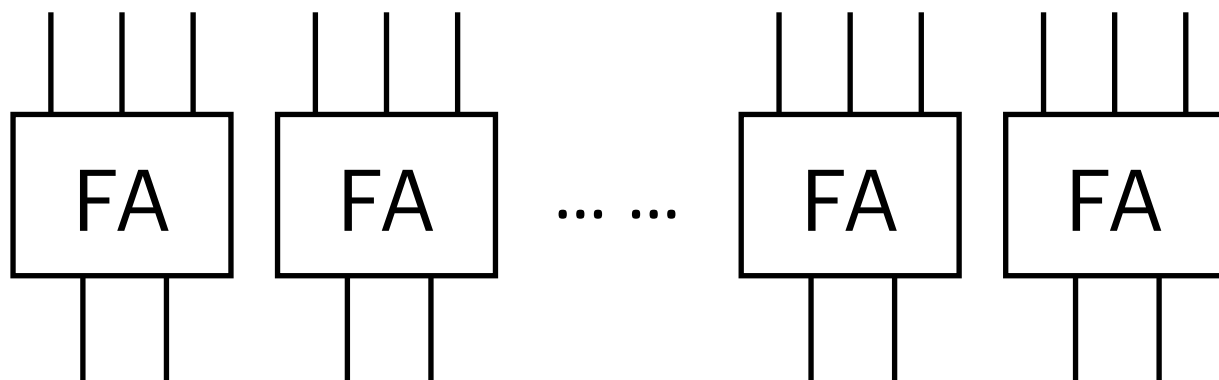


Build Larger Blocks—like LEGO®

$$\begin{array}{r} 01010101 \\ + \underline{01110011} \end{array}$$

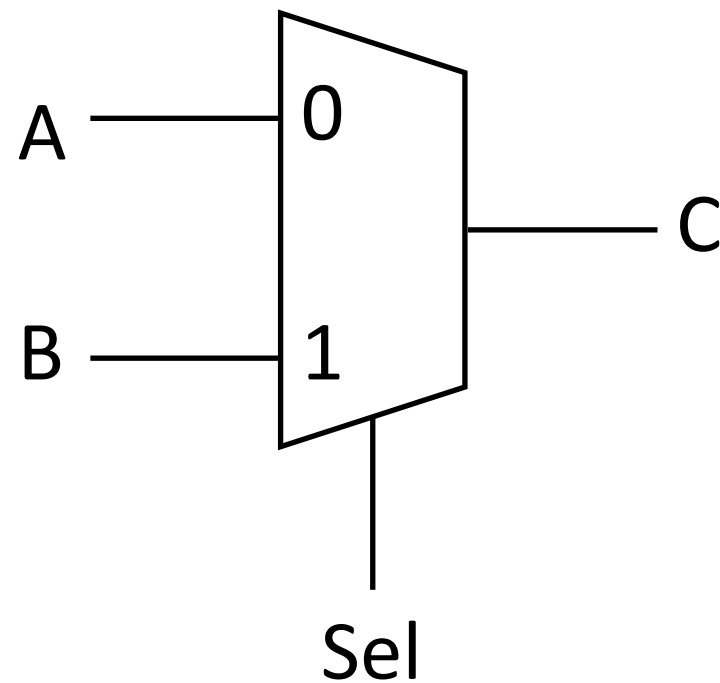
- Build a full adder (FA): truth table

Carry in	A	B	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

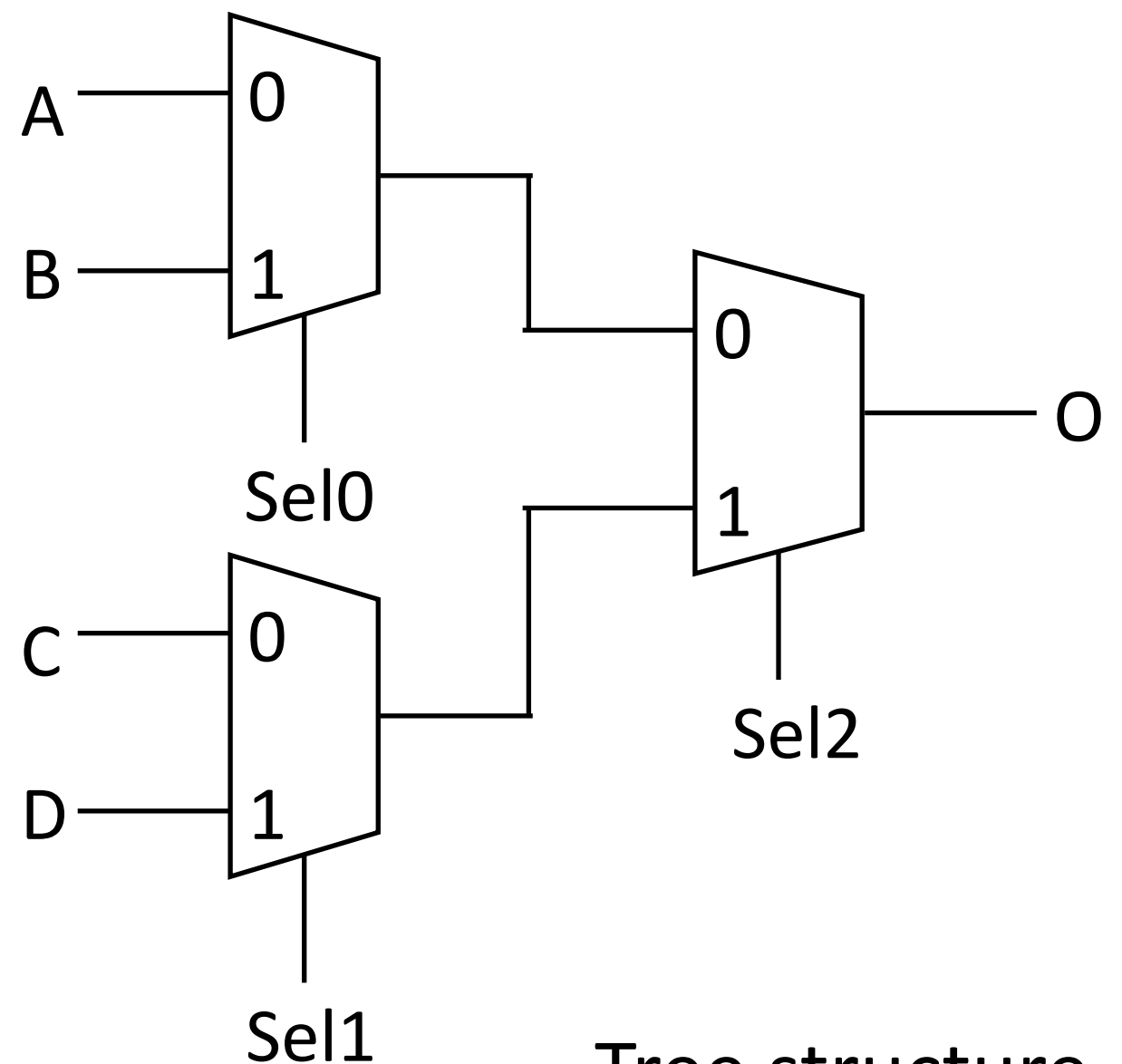


Other Useful Combinational Circuits

- Multiplexer (2-to-1)



- Multiplexer (2^n -to-1)

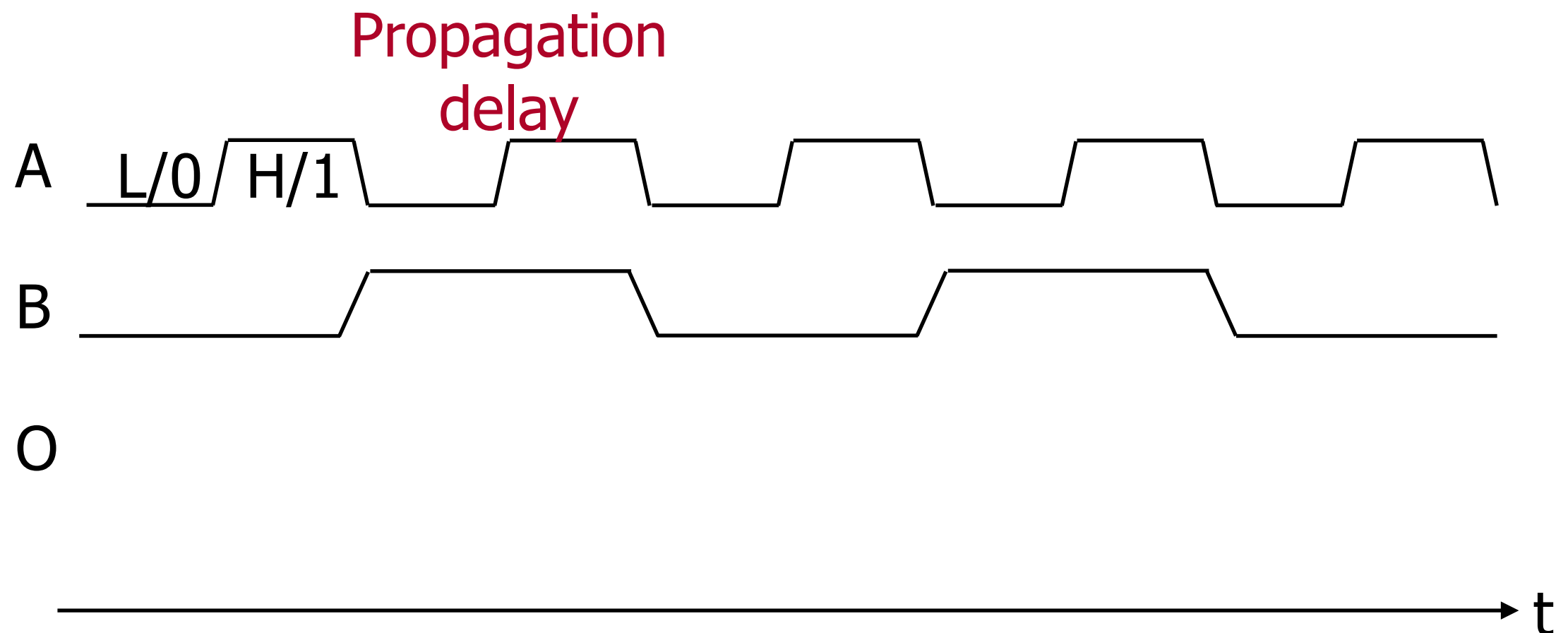


Tree structure

Timing Diagram—Signal & Waveform



Transistors:
non-ideal
switches



Timing Diagram—Signal Grouping



A

a₀

a₁

a₂

b₀

b₁

b₂

O₀

O₁

O₂

O

