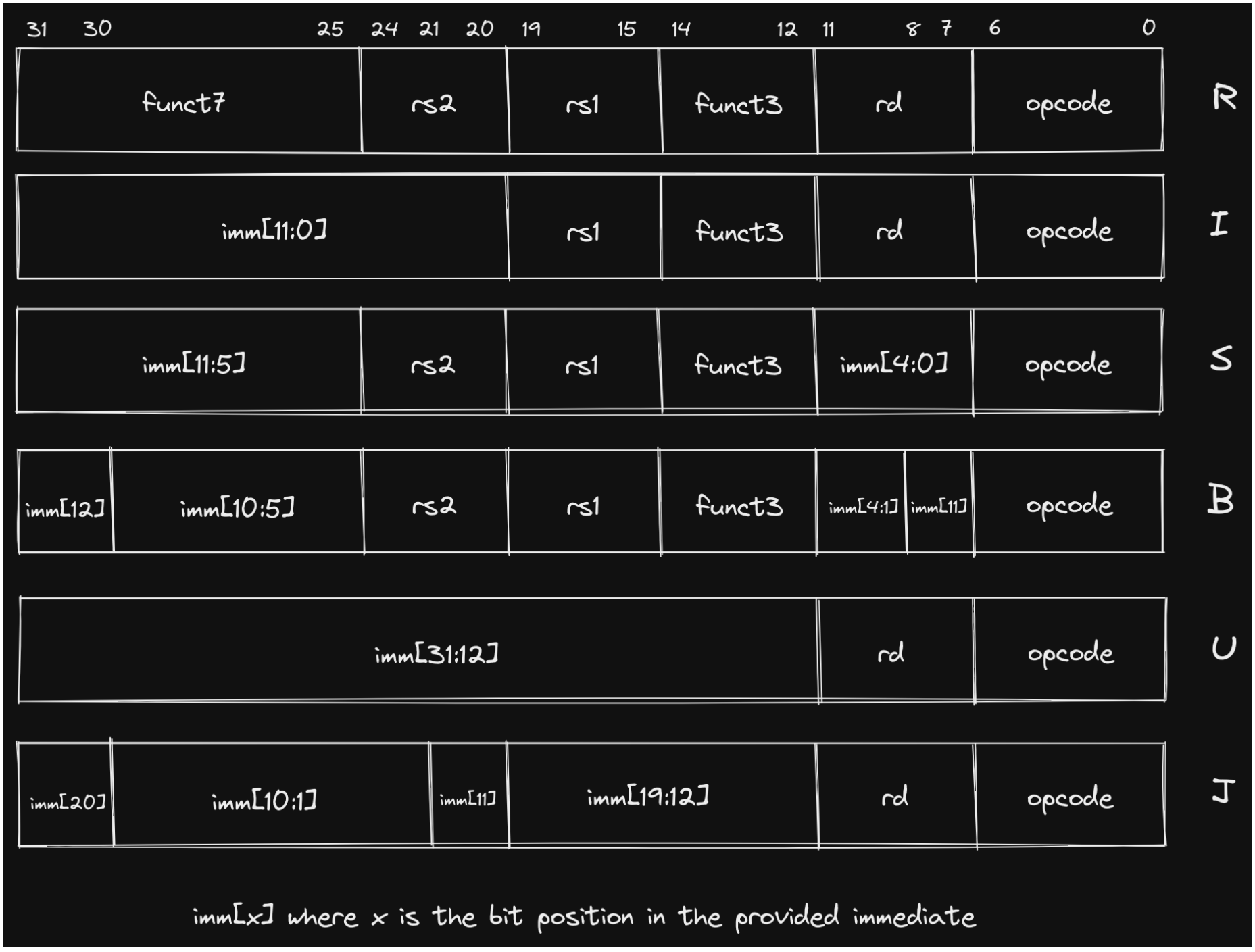
Resources for Doing Assignment

- Online RISCV simulator for assembly language code development, execution and debugging
  - https://venus.cs61c.org/

- Microsoft VS-CODE based locally installed RISCV simulator for assembly language code development, execution and debugging (Windows, Linux, macOS).
  - VS-Code must be installed first.
  - Navigate to “Extensions” button after launching VS-Code and download RISC-V Support and RISC-V Venus Simulator from menu
  - Howto: https://marketplace.visualstudio.com/items?itemName=hm.riscv-venus

- Quick start to RISC V assembly language programming:
  - https://www.cp.eng.chula.ac.th/~prabhas/teaching/comparch/2022/Programming-RISC-V-assembly.htm

- Detailed RISCV assembly language manual and RISC-V ISA specs have been uploaded on Moodle and MS-Teams group (Files Section)


31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 rs2 rs1 funct3 rd opcode R
imm[11:0] rs1 funct3 rd opcode I
imm[11:5] rs2 rs1 funct3 imm[4:0] opcode S
imm[12] imm[10:5] rs2 rs1 funct3 imm[4:1] imm[11] opcode B
imm[31:12] rd opcode U
imm[20] imm[10:1] imm[11] imm[19:12] rd opcode J
imm[x] where x is the bit position in the provided immediate

ISA Design

- RISC-V has 32 integer registers and can have 32 floating-point registers
  - Register number 0 is a constant 0
  - Register number 1 is the return address (link register)
- The memory is addressed by 8-bit bytes
- The instructions must be aligned to 32-bit addresses
- Like many RISC designs, it is a "load-store" machine
  - The only instructions that access main memory are loads and stores
  - All arithmetic and logic operations occur between registers
- RISC-V can load and store 8 and 16-bit items, but it lacks 8 and 16-bit arithmetic, including comparison-and-branch instructions
- The 64-bit instruction set includes 32-bit arithmetic

Inst[4:2] / Inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom- 1*	AM0	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv 128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custoni,.3/rv 128*	>=80b

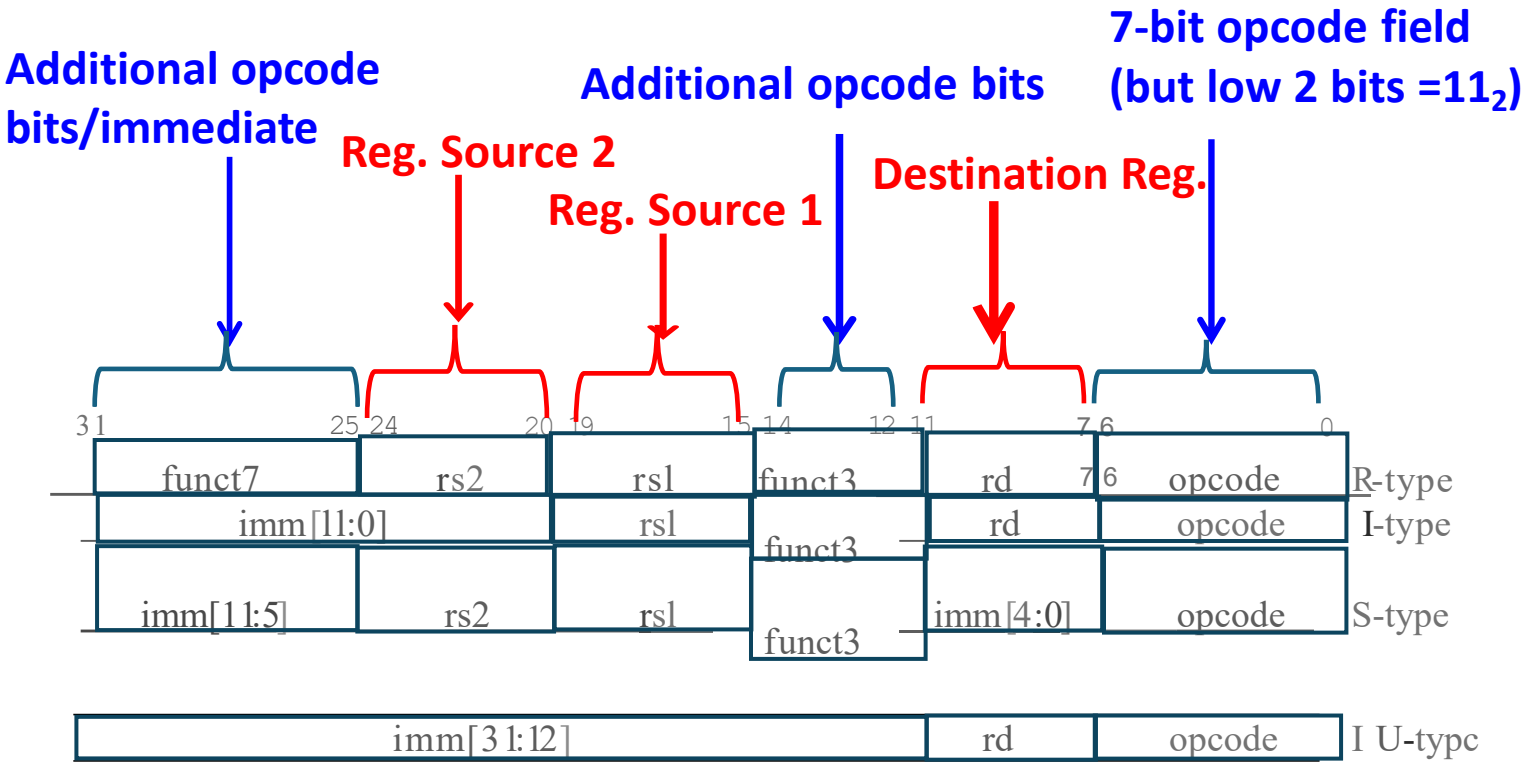
Fact Sheet of RISC V Instructions

- 32 bits ([31:0]) wide.
- First 7 bits ([6:0]) for the opcode.
- If two formats support same operands, that operand is always in the same location in the instruction
- At least one register operand (rd / rs1/ rs2).
- All register operands are the same number of bits (5).
- While two types of instructions that fall under the same format may seem unrelated, they are implemented using the same underlying operations in the ALU.
    - R: register-register ALU instructions
    - I: ALU immediate instructions, load instructions
    - S / B: store instructions, comparison and branch instructions
    - U / J: jump instructions, jump and link instructions

Examples

- addi: I Format
- sd: S Format
- lui: U Format
- addw: R Format

RISC-V Instruction Formats



- Aligned on four-byte boundary in memory.
- Sign bit of immediates always on bit 31 of instruction.
- Register field positions fixed.

Data Formats and Memory Addresses

Data formats:

8-b Bytes, 16-b Half words, 32-b words and 64-b double words

Some issues

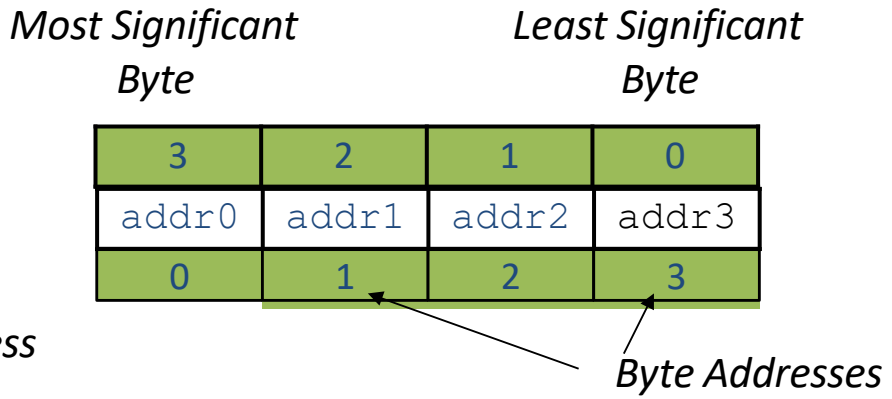
- *Byte ordering*

Little Endian

MSB of value goes to lowest address(RISC-V is little endian)

Big Endian

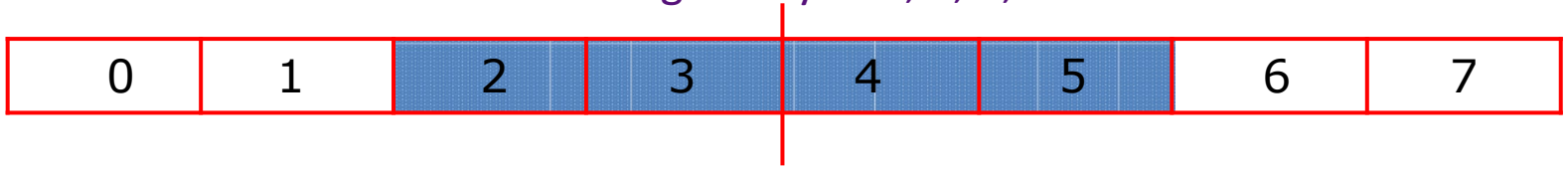
MSB of value goes to highest address



- *Word alignment*

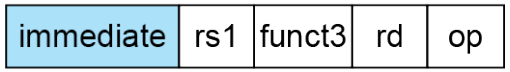
Suppose the memory is organized in 32-bit words.

Can a word address begin only at 0, 4, 8, ... ? Yes for Risc V

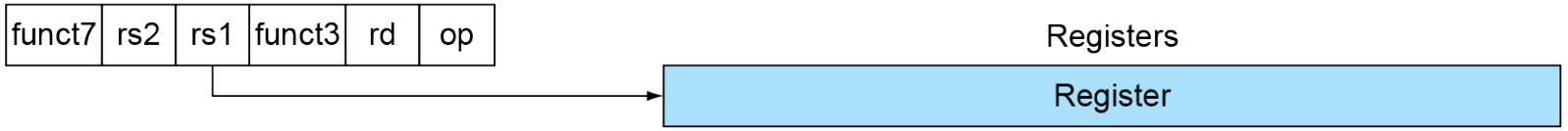


RISC-V Addressing Summary

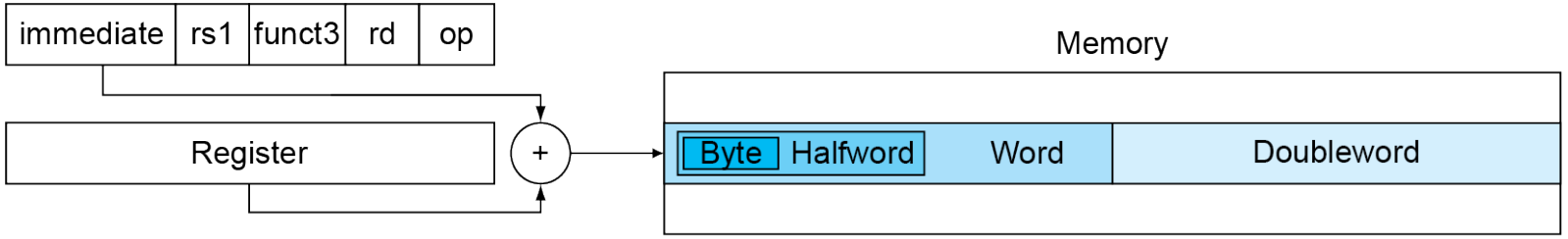
1. Immediate addressing



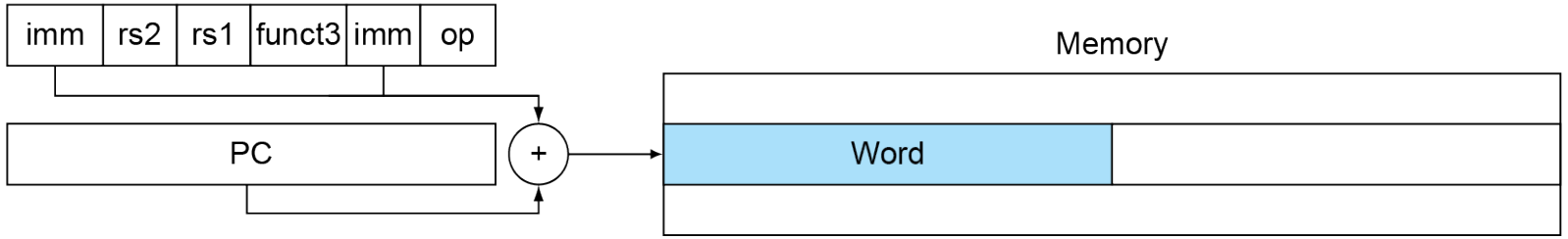
2. Register addressing



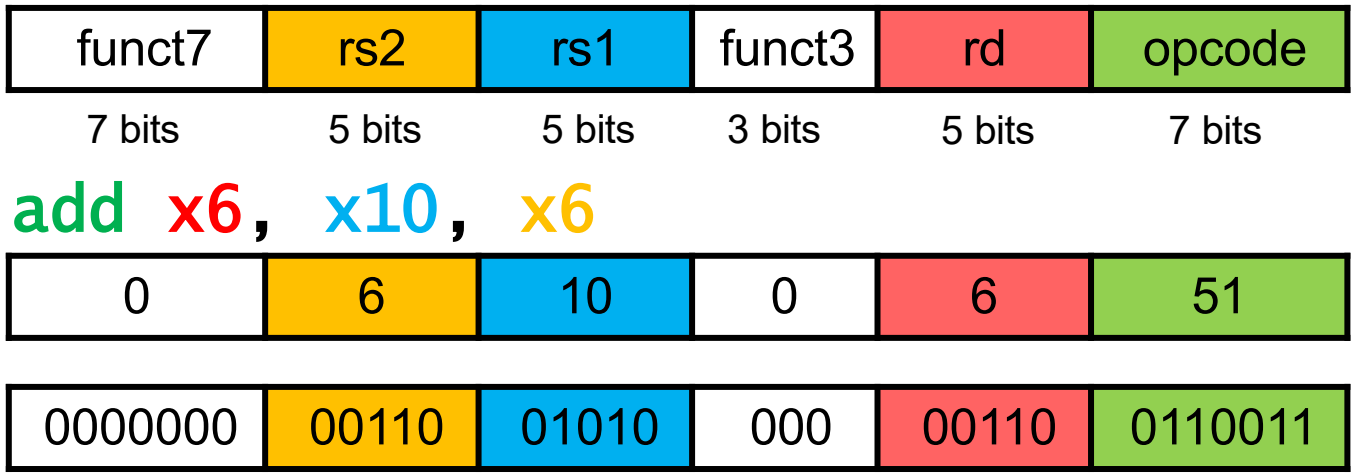
3. Base addressing, i.e., displacement addressing



4. PC-relative addressing



R-Format Encoding Example

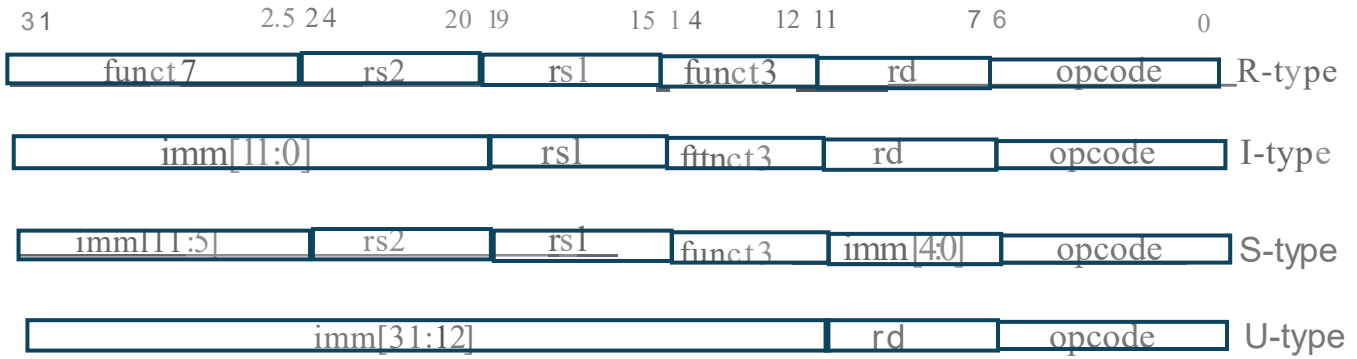


0000 0000 0110 0101 0000 0011 0011 0011b =

0x00650333

ISA Design for Performance (Instruction Decoding)

- Features to increase a computer's speed, while reducing its cost and power usage
  - placing most-significant bits at a fixed location to speed sign-extension, and a bit-arrangement designed to reduce the number of multiplexers/decoders in a CPU



I-immediate[11:0]	src	ADDI/SLTI [U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

ALU Instructions

Example instruction	Instruction name	Meaning
add x1,x2,x3	Add	Regs[x1] ← Regs[x2] + Regs[x3]
addi x1,x2,3	Add immediate unsigned	Regs[x1] ← Regs[x2] + 3
lui x1,42	Load upper immediate	Regs[x1] ← 0^{32} ## 42 ## 0^{12}
sll x1,x2,5	Shift left logical	Regs[x1] ← Regs[x2] << 5
slt x1,x2,x3	Set less than	if (Regs[x2] < Regs[x3]) Regs[x1] ← 1 else Regs[x1] ← 0

Bit 31 – 12 of opcode

Basic ALU instructions in RISC-V are available both with register register operands and with one immediate operand. LUI uses the U-format which uses the rs1 field as part of the immediate, yielding a 20-bit immediate.

Load/Store Instructions

Example instruction	Instruction name	Meaning
ld x1,80(x2)	Load doubleword (64b)	Regs[x1] ← Mem[80 + Regs[x2]
lw x1,60(x2)	Load word (32b)	Regs[x1] ← $_{64}$(Mem[60 +Regs[x2]$_0$) ^{32} ## Mem[60+Regs[x2]]
lwu x1,60(x2)	Load word unsigned	Regs[x1] ← $_{64}$ 0^{32} ## Mem[60+Regs[x2]]
lb x1,40(x3)	Load byte (8b)	Regs[x1] ← $_{64}$(Mem[40 + Regs[x3]]$_0$) ^{56} ## Mem[40+Regs[x3]]
lbu x1,40(x3)	Load byte unsigned	Regs[x1] ← $_{64}$ 0^{56} ## Mem[40+Regs[x3]]
lh x1,40(x3)	Load half word (16b)	Regs[x1] ← $_{64}$ (Mem[40 + Regs[x3]]$_0$) ^{48} ## Mem[40 +Regs[x3]]
flw f0,50(x3)	Load FP single (32b)	Regs[f0] ← $_{64}$ Mem[50+Regs[x3]] ## 0^{32}
fld f0,50(x2)	Load FP double (64b)	Regs[f0] ← $_{64}$ Mem[50+Regs[x2]]
sd x2,400(x3)	Store double (64b)	Mem[400 + Regs[x3]] ← $_{64}$Regs[x2]
sw x3,500(x4)	Store word (32b)	Mem[500 + Regs[x4]] ← $_{32}$Regs[x3]$_{32..63}$
fsw f0,40(x3)	Store FP single (32b)	Mem[40 +Regs[x3]] ← $_{32}$Regs[f0]$_{0..31}$
fsd f0, 40(x3)	Store FP double (64b)	Mem[40+Regs[x3]] ← $_{64}$Regs[f0]
sh x3,502(x2)	Store half (16b)	Mem[502 + Regs[x2]] ← $_{16}$Regs[x3]$_{48..63}$
sb x2, 41(x3)	Store byte (8b)	Mem[41 +Regs[x3]] ← $_{8}$Regs[x2]$_{56..63}$

Sign extension

Zero padding

Load and store instructions in RISC-V. Loads shorter than 64 bits are available in both sign extended and zero-extended forms. All memory references use a single addressing mode. Both loads and stores are available for all the data types shown. Since RV64G supports double precision floating point, all single precision floating point loads must be aligned in the FP register, which are 64-bits wide.

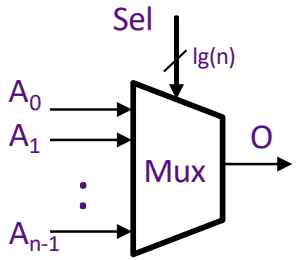
Control Flow Instructions

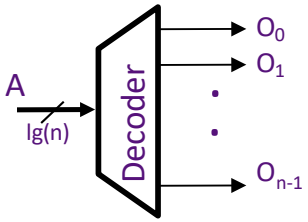
Example instruction		Instruction name	Meaning
jal	x1,offset	Jump and link	Regs[x1] ← PC+4; PC ← PC + (offset << 1)
jalr	x1,x2,offset	Jump and link register	Regs[x1] ← PC+4; PC ← Regs[x2]+offset
beq	x3, x4, offset	Branch equal zero	if (Regs[x3]==Regs[x4]) PC ← PC + (offset<<1)
bgt	x3, x4, name	Branch not equal zero (>)	if (Regs[x3] > Regs[x4]) PC ← PC + (offset << 1)

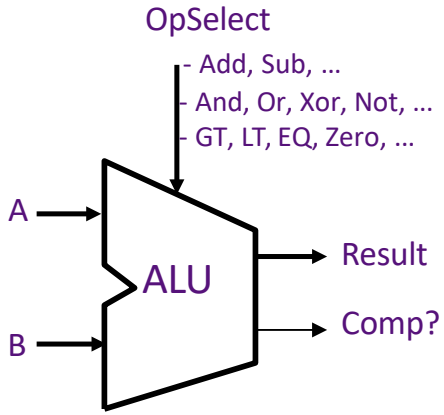
Typical control flow instructions in RISC-V. All control instructions, except jumps to an address in a register, are PC-relative.

Hardware Elements of CPU

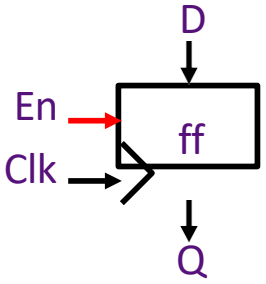
- Combinational circuits
  - Mux, Decoder, ALU, ...

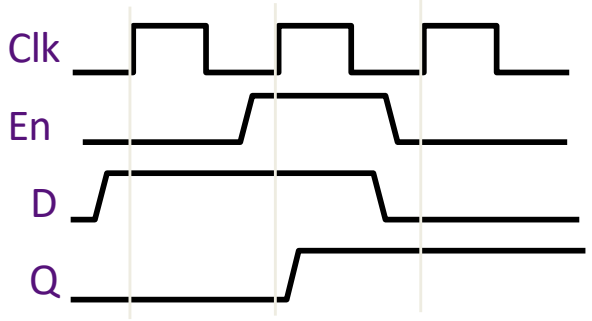






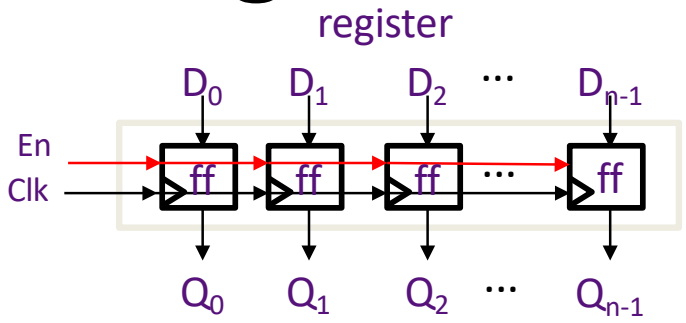
- Synchronous state elements
  - Flipflop, Register, Register file, SRAM, DRAM



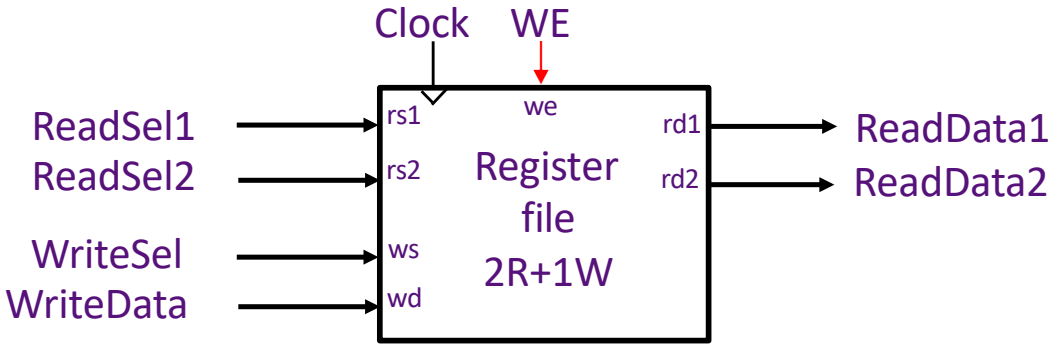


Edge-triggered: Data is sampled at the rising edge

Register Files

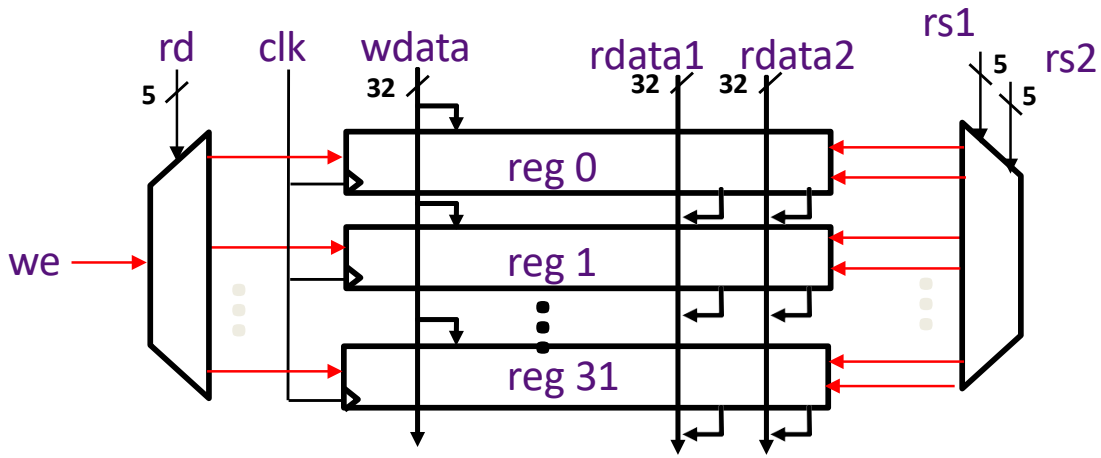


- Reads are combinational
  - Can read in any cycle and for multiple reads
  - 2 register source operands needed

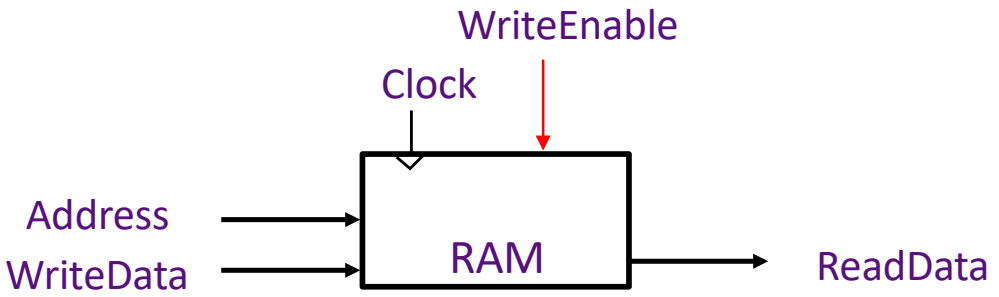


Register File Implementation

- RISC-V integer instructions have at most 2 register source operands



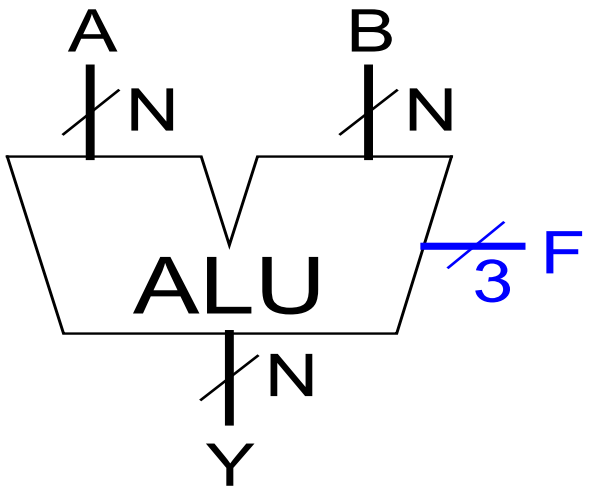
A Simple Memory Model



- Reads and writes are always completed in one cycle
- Read can be done any time (i.e. combinational)
- Write is performed at the rising clock edge
  - if it is enabled

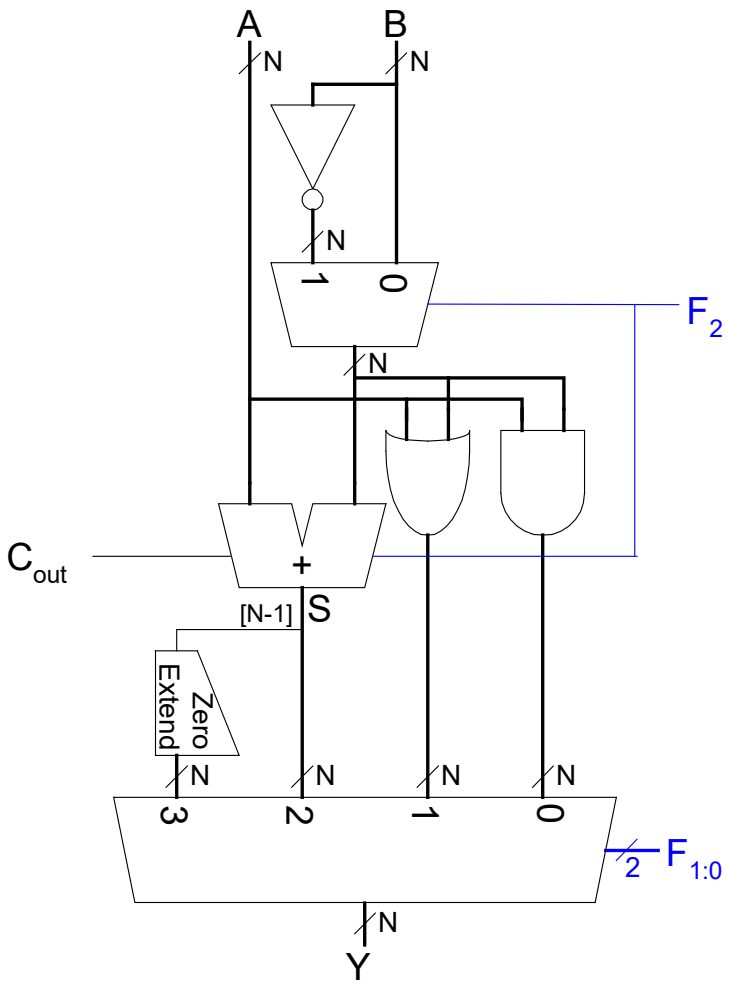
Arithmetic Circuits

Arithmetic Logic Unit (ALU)



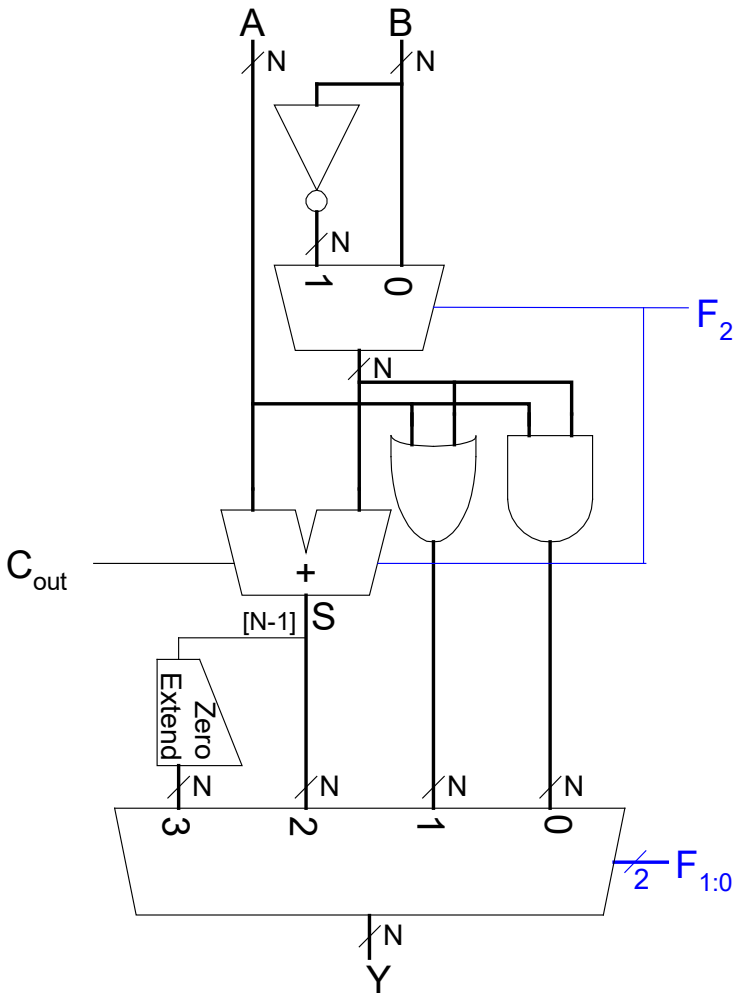
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

ALU Design



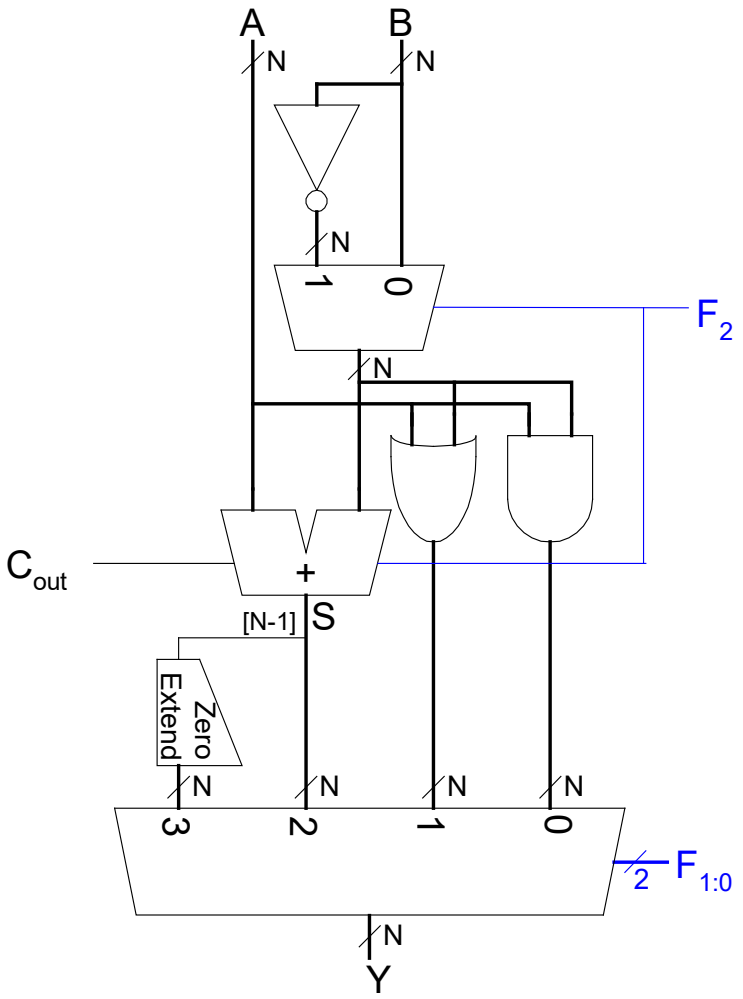
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Set Less Than (SLT) Example



- Configure a 32-bit ALU for the set if less than (SLT) operation. Suppose $A = 25$ and $B = 32$.

Set Less Than (SLT) Example

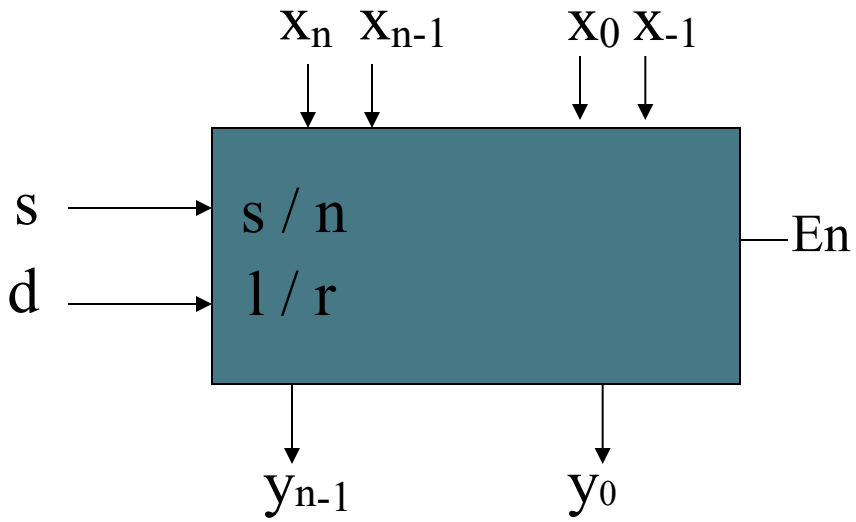


- Configure a 32-bit ALU for the set if less than (SLT) operation. Suppose $A = 25$ and $B = 32$.
  - A is less than B, so we expect Y to be the 32-bit representation of 1 (0x00000001).
  - For SLT, $F_{2:0} = 111$.
  - $F_2 = 1$ configures the adder unit as a subtracter. So 25 - 32 = -7.
  - The two's complement representation of -7 has a 1 in the most significant bit, so $S_{31} = 1$.
  - With $F_{1:0} = 11$, the final multiplexer selects $Y = S_{31}$ (zero extended) = 0x00000001.

Shifters

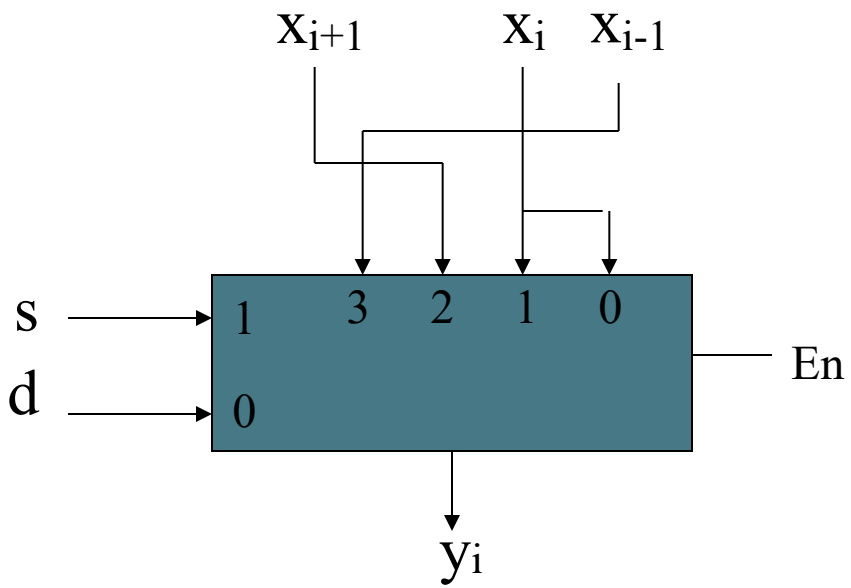
- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
  - Ex: 11001 >> 2 = 00110
  - Ex: 11001 << 2 = 00100
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
  - Ex: 11001 >>> 2 = **11**110
  - Ex: 11001 <<< 2 = 00100
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: 11001 ROR 2 = 01110
  - Ex: 11001 ROL 2 = 00111

Shifter

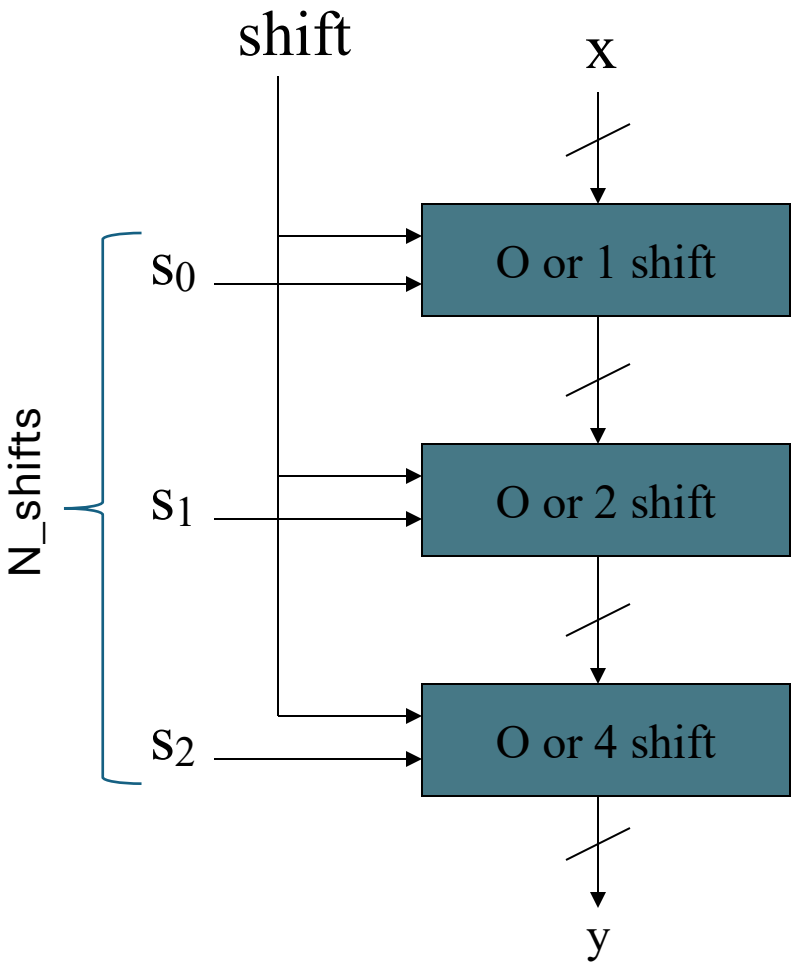


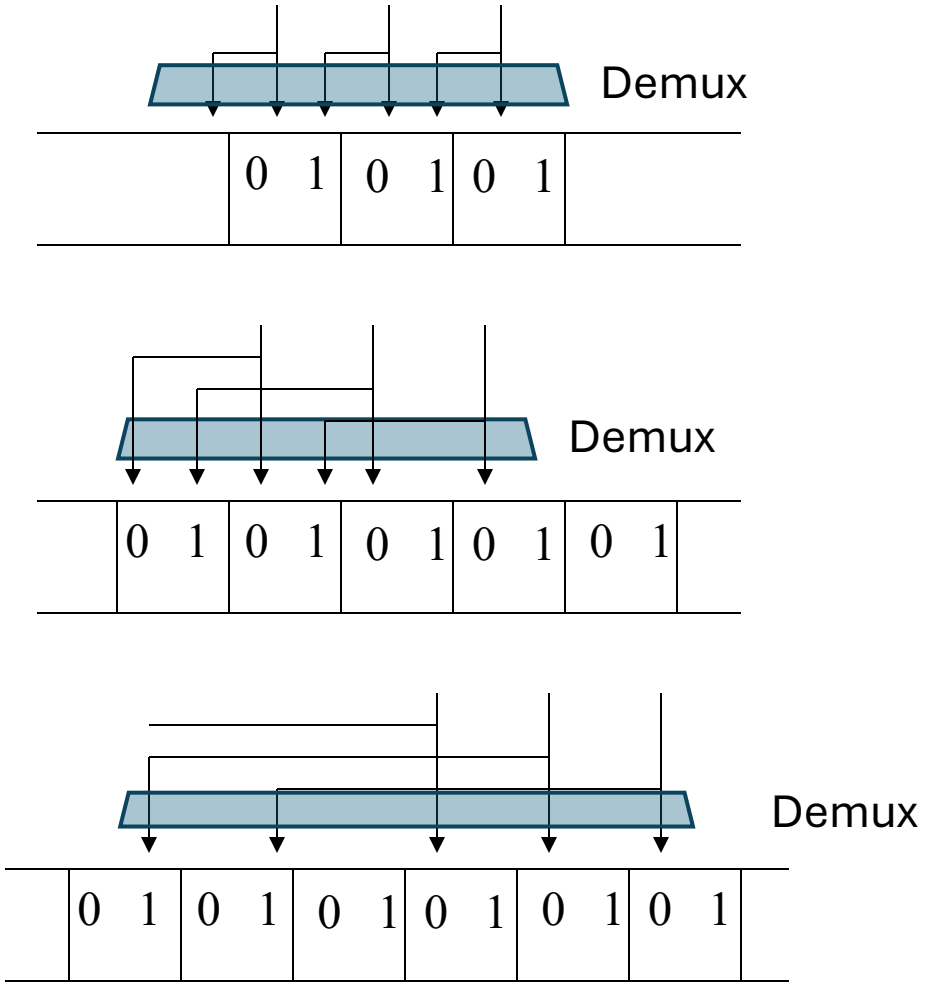
$y_i = x_{i-1}$ if En = 1, s = 1, and d = L (ie.1)
$= x_{i+1}$ if En = 1, s = 1, and d = R (ie.0)
$= x_i$ if En = 1, s = 0, and d = ϕ
$= 0$ if En = 0

Each o/p bit can be implemented with a mux

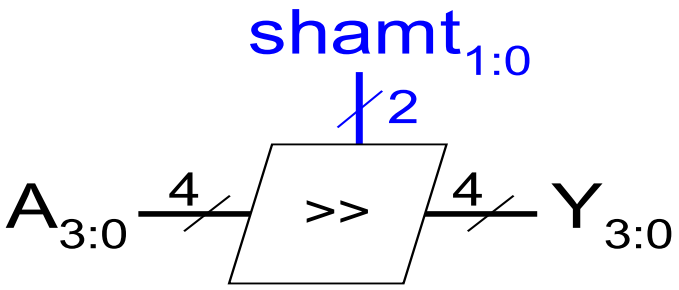


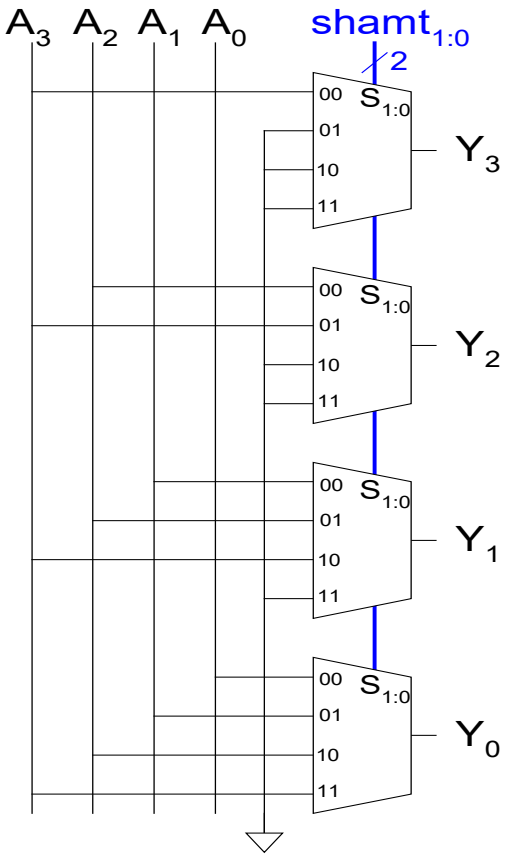
Barrel Shifter





Shifter Design





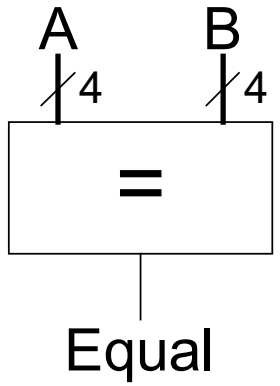
Shifters as Multipliers and Dividers

- A left shift by *N* bits multiplies a number by 2^N
  - Ex: 00001 << 2 = 00100 ($1 \times 2^2 = 4$)
  - Ex: 11101 << 2 = 10100 ($-3 \times 2^2 = -12$)

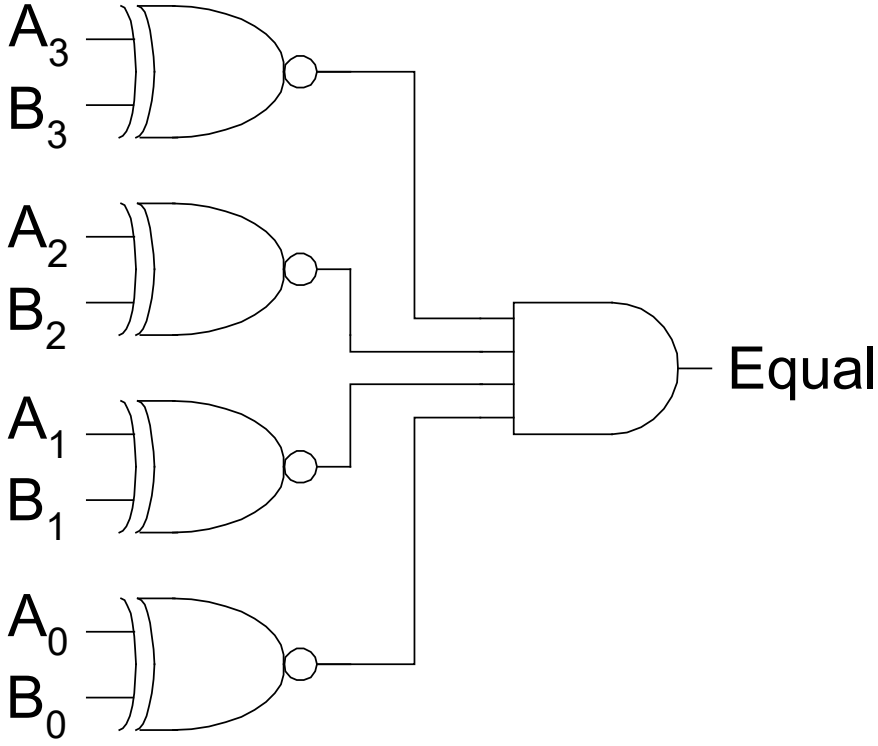
- The arithmetic right shift by *N* divides a number by 2^N
  - Ex: 01000 >>> 2 = 00010 ($8 \div 2^2 = 2$)
  - Ex: 10000 >>> 2 = 11100 ($-16 \div 2^2 = -4$)

Comparator: Equality

Symbol

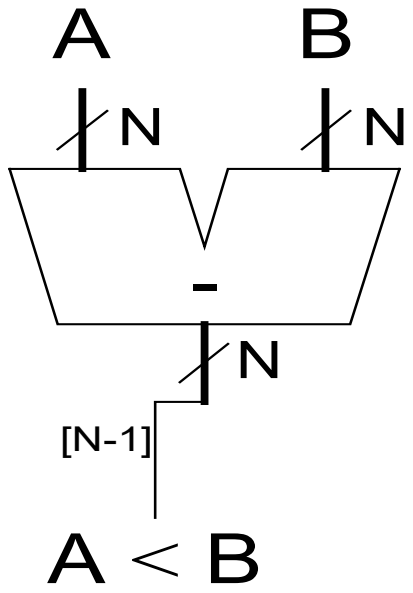


Implementation

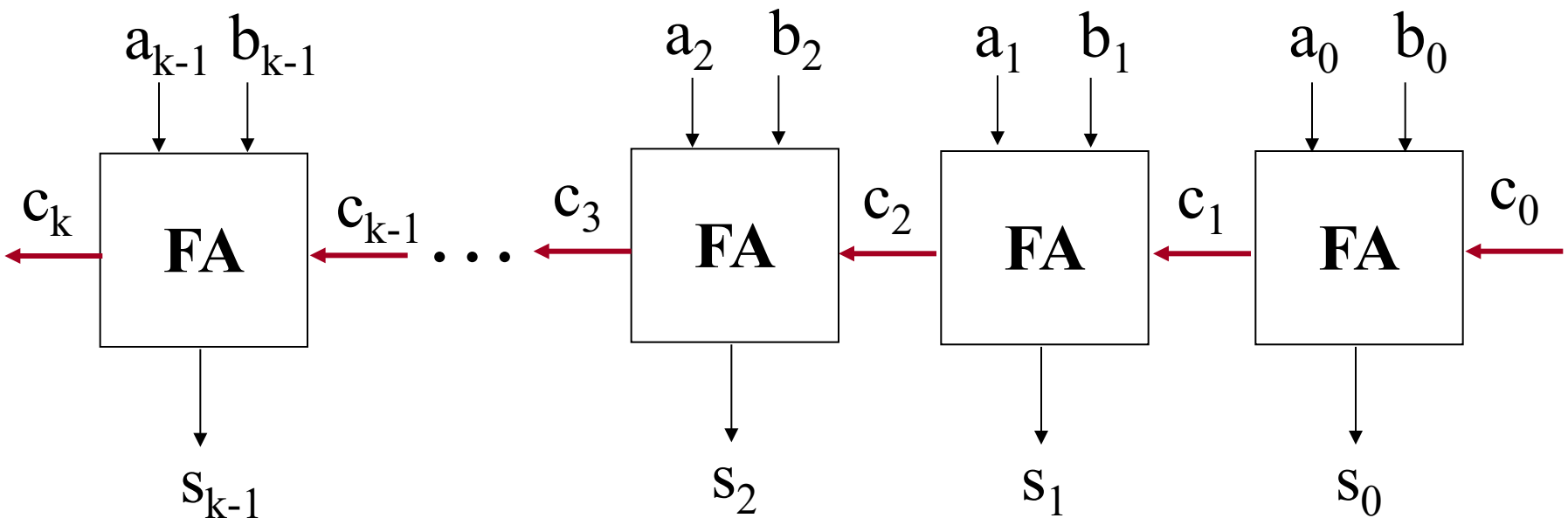


Comparator: Less Than

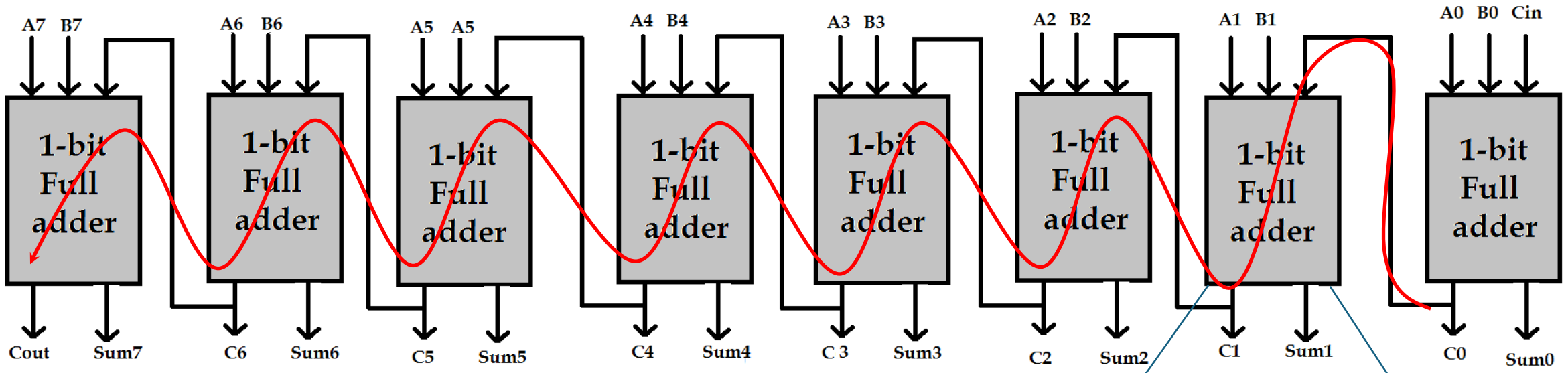
- For unsigned numbers



Ripple-Carry Carry Propagate Adder (CPA)



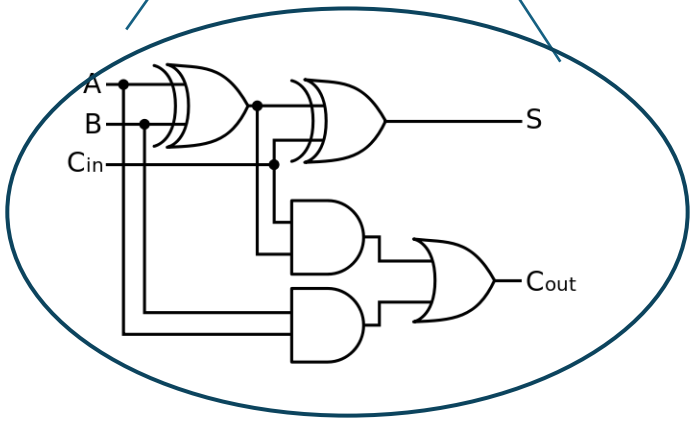
Naïve 8-bit adder (Ripple carry adder)



Slow

Carry Propagation path determines speed

2 gate delays in carry path in each 1 bit adder



Carry-Lookahead Adder

- Reduce no. of gates C_{out} propagates through
- **Some definitions:**
  - Generate (G_i) and propagate (P_i) signals for each column:
    - A column will generate a carry out if A_i AND B_i are both 1.

$$G_i = A_i B_i$$

    - A column will propagate a carry in to the carry out if A_i OR B_i is 1. – *Carry in generates carry out*

$$P_i = A_i + B_i$$

    - The carry out of a column (C_i) is:

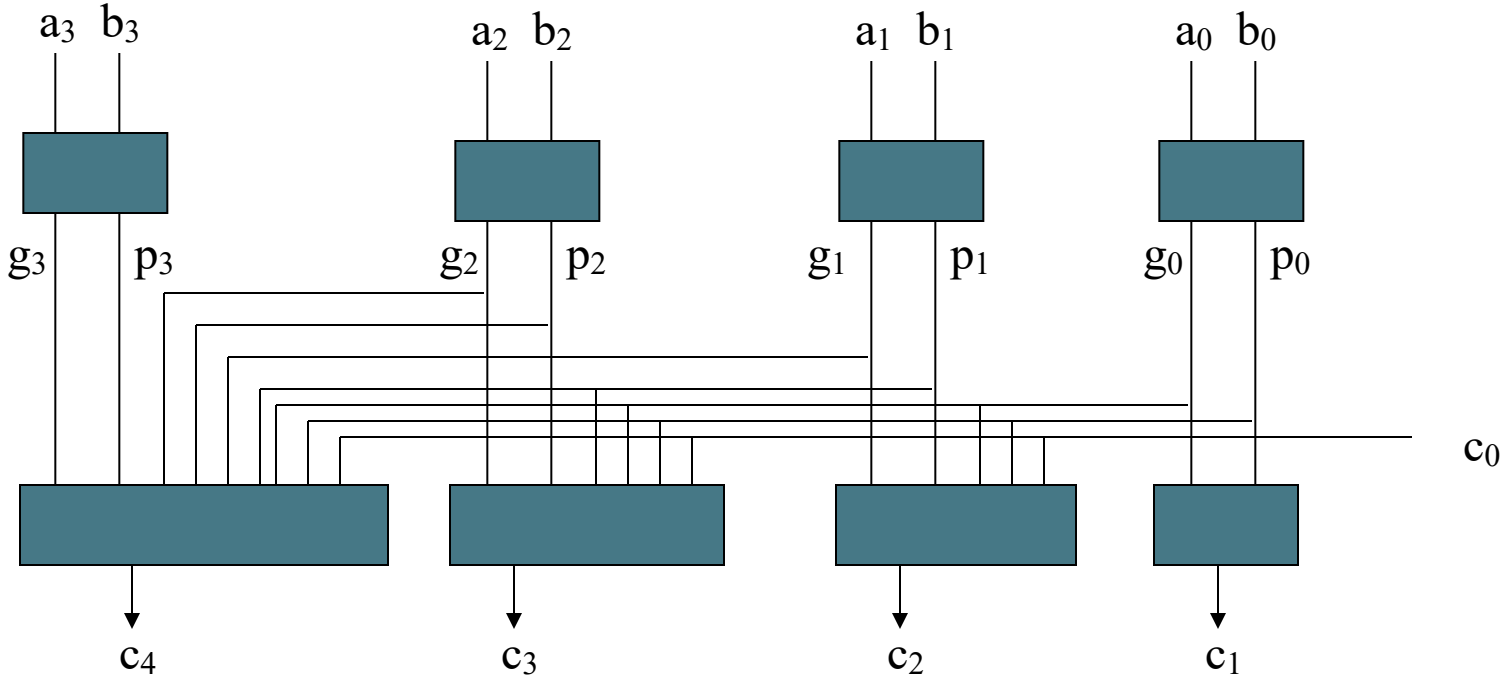
$$C_{i+1} = A_i B_i + (A_i + B_i) C_i = G_i + P_i C_i$$

Inputs			Outputs	
A	B	C_{in}	Sum	Carry
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

Carry Look Ahead Adder

$$c_1 = a_0b_0 + (a_0+b_0)c_0 = g_0 + p_0c_0$$
$$c_2 = a_1b_1 + (a_1+b_1)c_1 = g_1 + p_1c_1 = g_1 + p_1g_0 + p_1p_0c_0$$
$$c_3 = a_2b_2 + (a_2+b_2)c_2 = g_2 + p_2c_2 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$$
$$c_4 = a_3b_3 + (a_3+b_3)c_3 = g_3 + p_3c_3 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$$

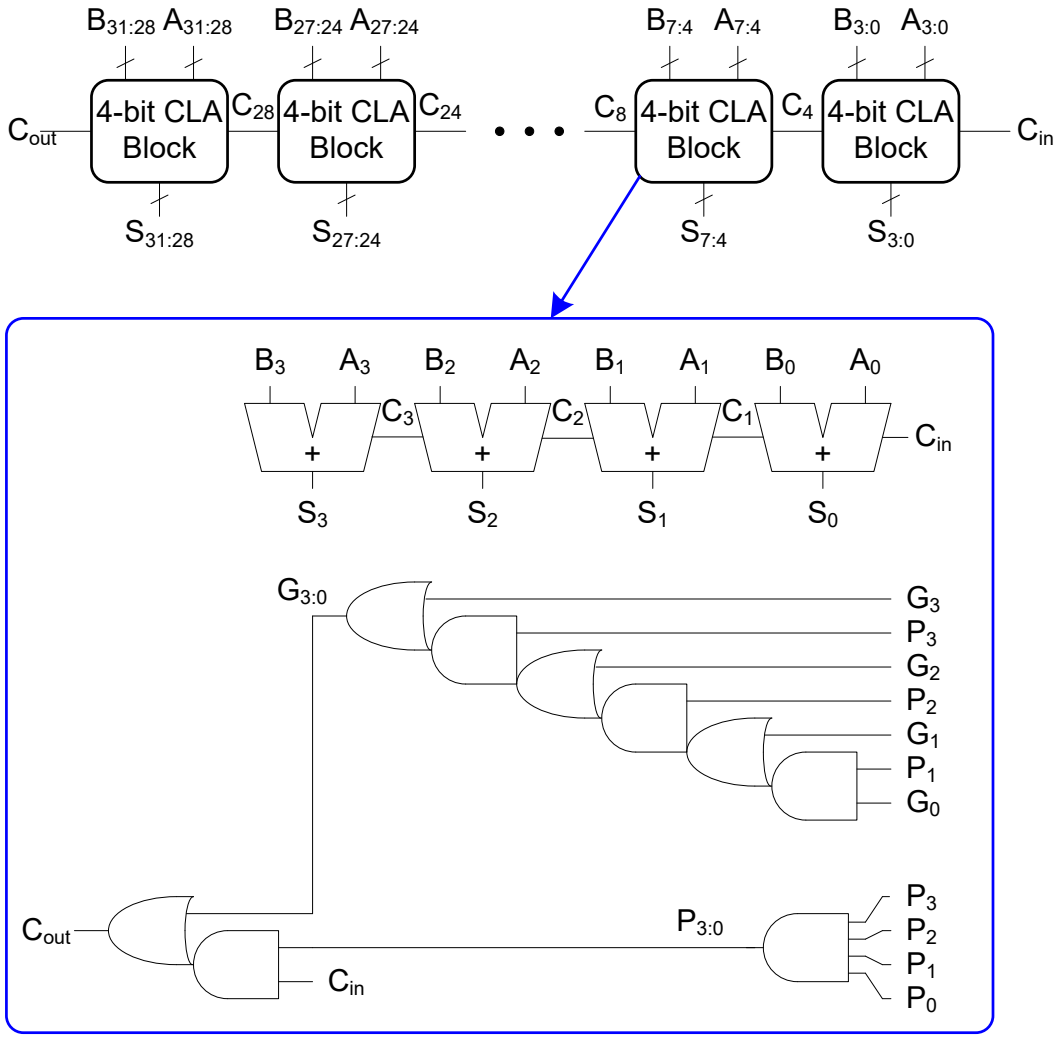
$g_i = a_ib_i$ $\quad$ $p_i = a_i + b_i$

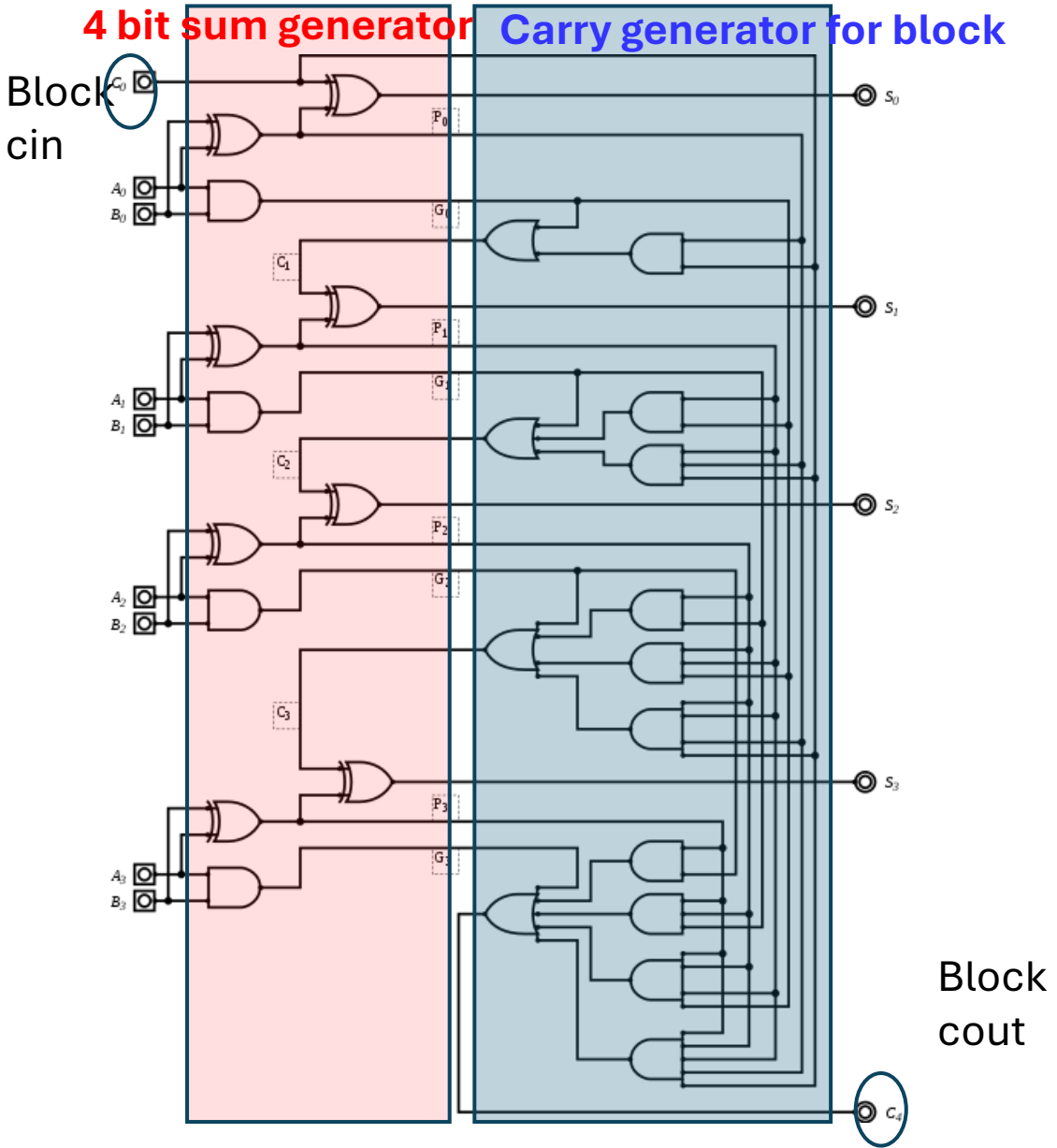


Carry-Lookahead Addition

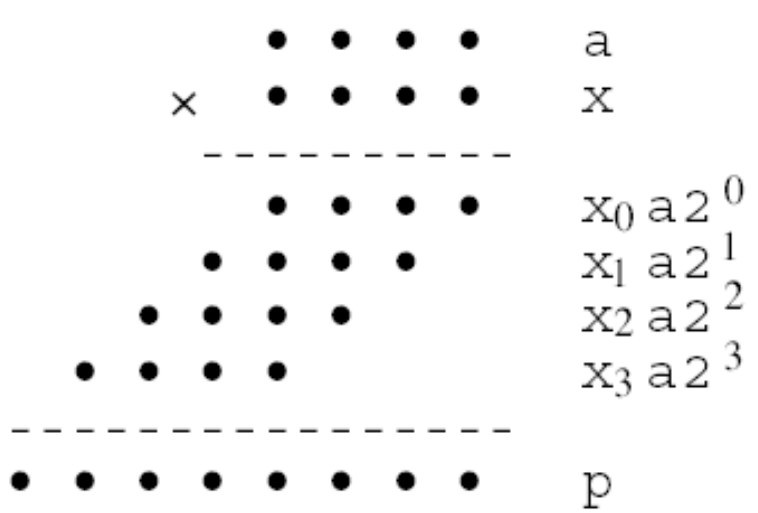
- Step 1: compute *generate* (G) and *propagate* (P) signals each bit
- Step 2: compute G and P for k-bit blocks
- Step 3: C_{in} propagates through each k-bit propagate/generate block

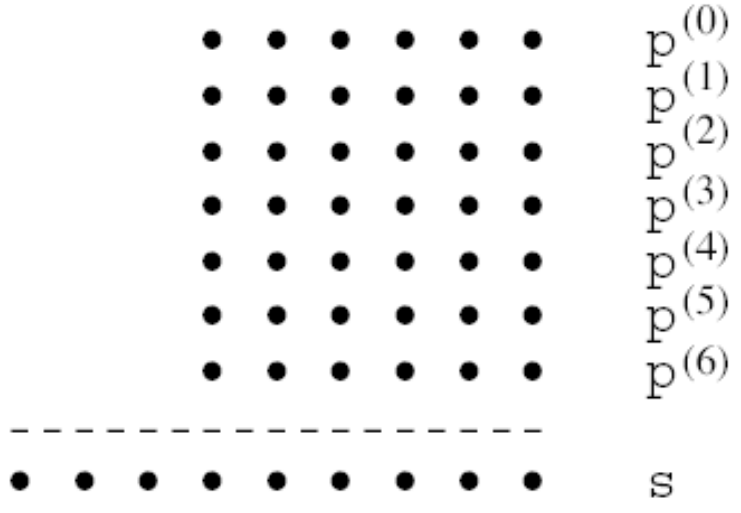
32-bit CLA with 4-bit blocks





Multioperand addition





Multiplication

$p = a \cdot x$

Inner product, Convolution

$$s = \sum_{i=0}^{n-1} x^{(i)} y^{(i)} = \sum_{i=0}^{n-1} p^{(i)}$$

Multipoperand addition
Number of bits of the result

$$S = \sum_{i=0}^{n-1} x^{(i)} \qquad x^{(i)} \in [0..2^k\text{-}1]$$

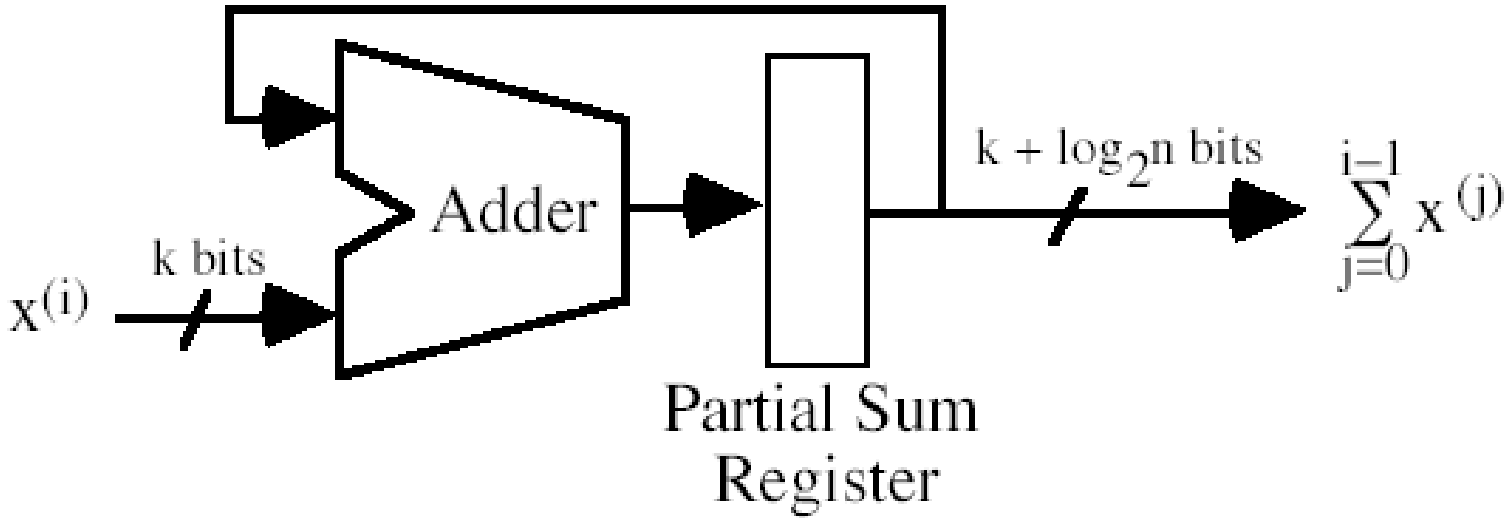
$$S_{min} = 0 \qquad S_{max} = n\,(2^k\text{-}1)$$

$$\#\text{ of bits of } S = \left\lceil \log_2 (S_{max} + 1) \right\rceil =$$

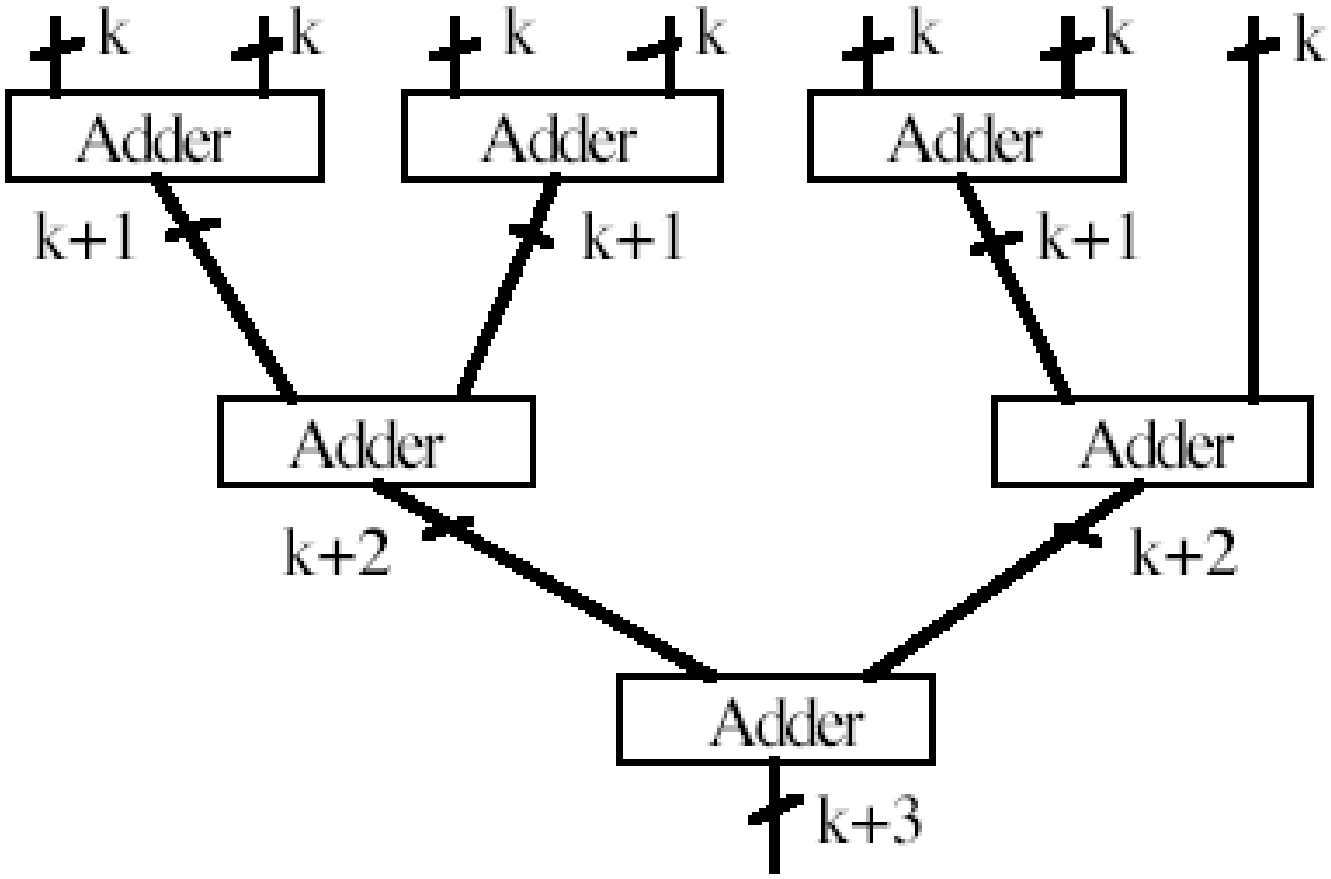
$$= \left\lceil \log_2 (n\,(2^k\text{-}1) + 1) \right\rceil \leq \left\lceil \log_2 n\,2^k \right\rceil =$$

$$= k + \left\lceil \log_2 n \right\rceil$$

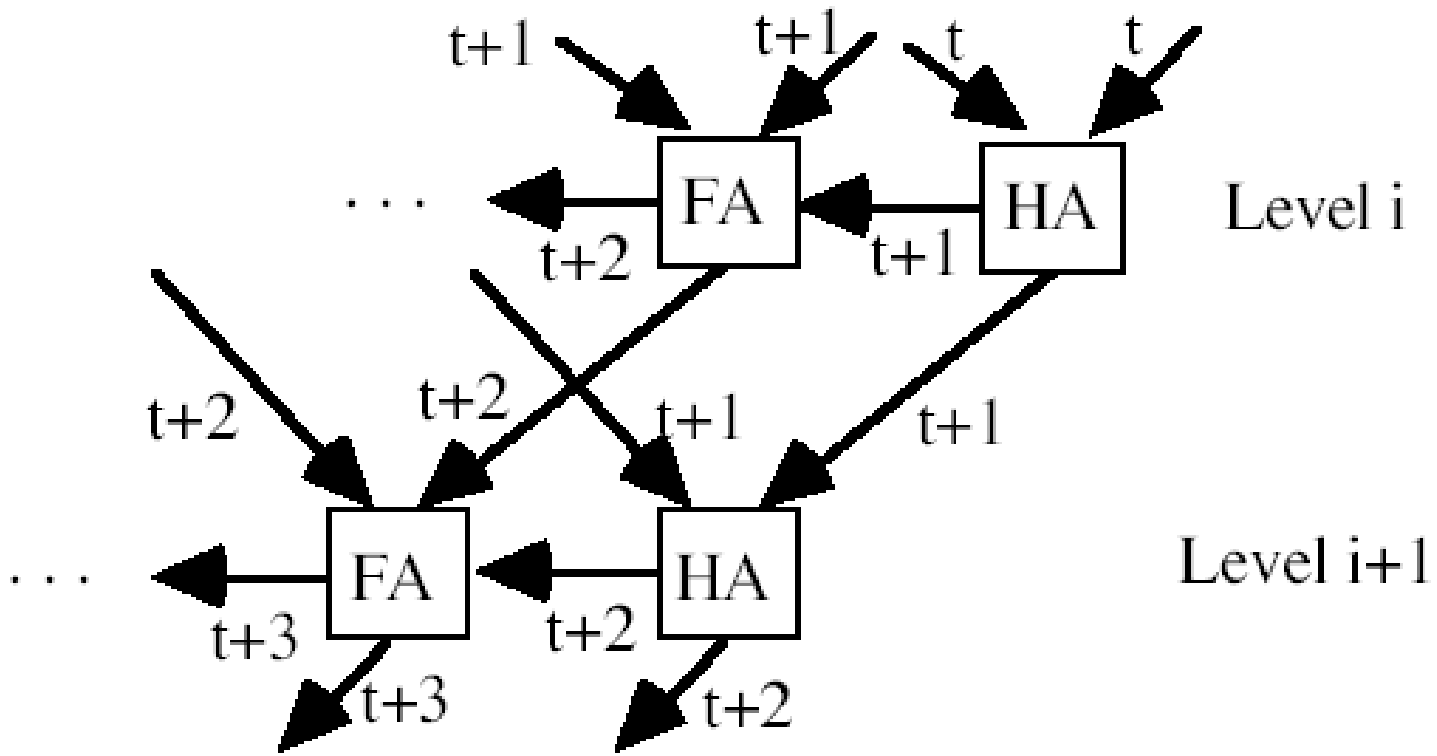
Serial implementation of multioperand addition



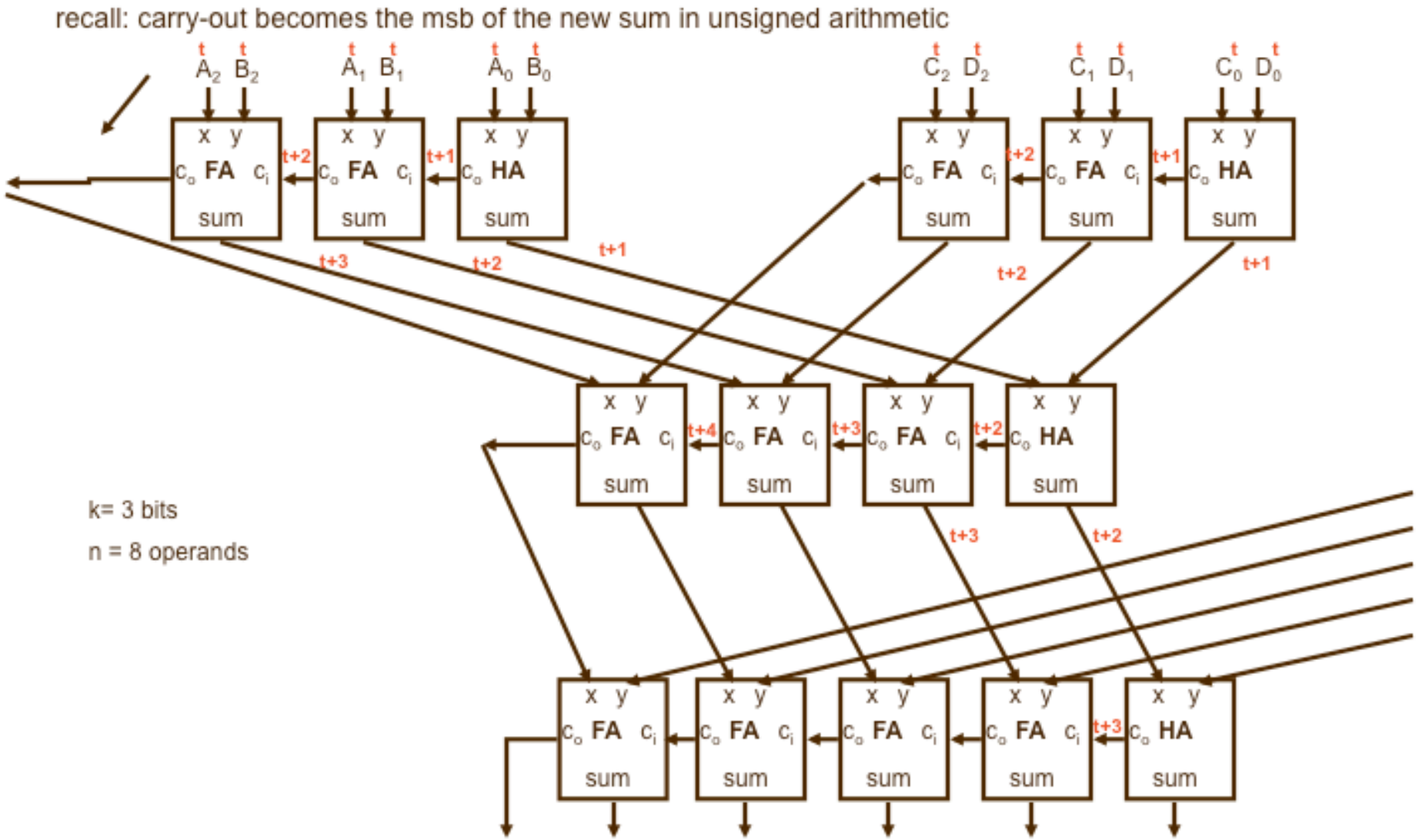
Adding 7 numbers in the binary tree of adders



Ripple-carry adders at levels i and i+1



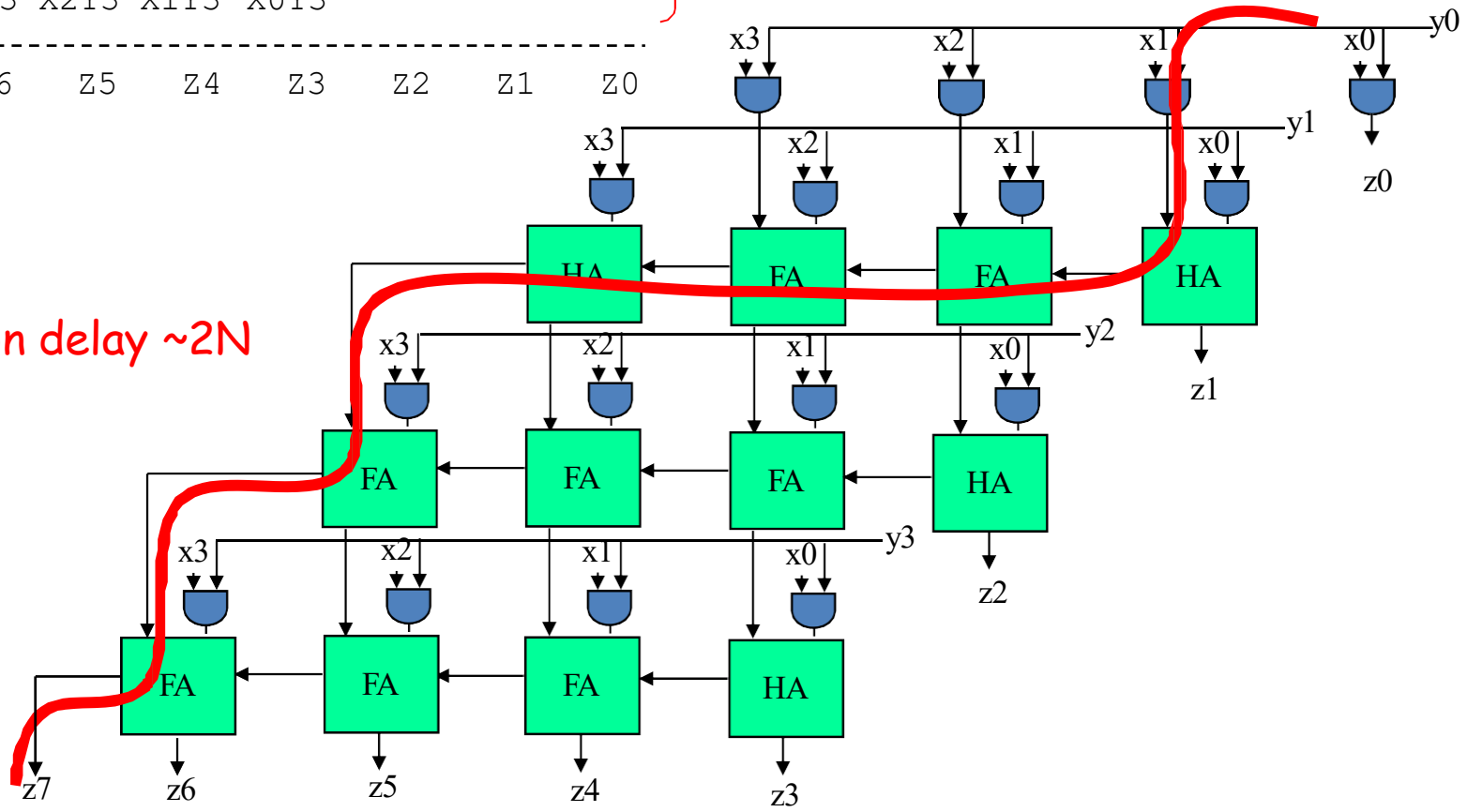
Example: Adding 8 3-bit numbers



Combinational Multiplier (unsigned)

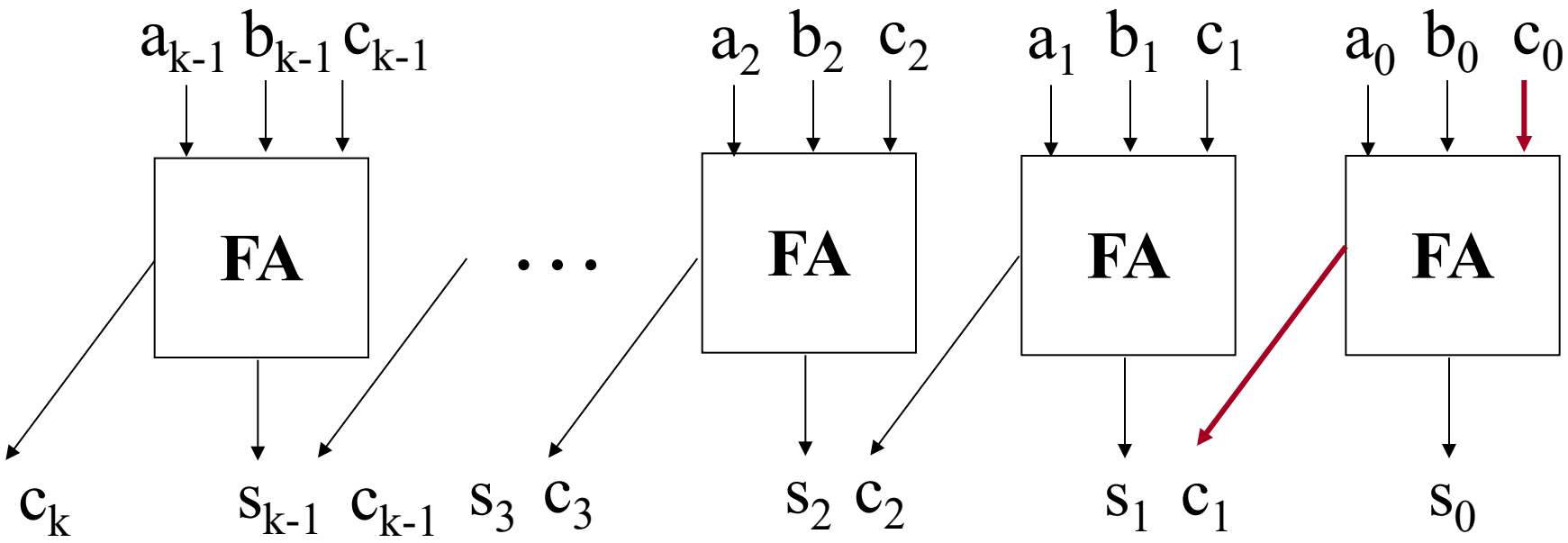
```
                          X3    X2    X1    X0   <- multiplicand
                      *   Y3    Y2    Y1    Y0   <- multiplier
                      ------------------------
                         X3Y0  X2Y0  X1Y0  X0Y0
+                  X3Y1  X2Y1  X1Y1  X0Y1
+            X3Y2  X2Y2  X1Y2  X0Y2
+      X3Y3  X2Y3  X1Y3  X0Y3
-----------------------------------------------
    Z7    Z6    Z5    Z4    Z3    Z2    Z1    Z0
```

Partial products, one for each bit in multiplier (each bit needs just one AND gate)

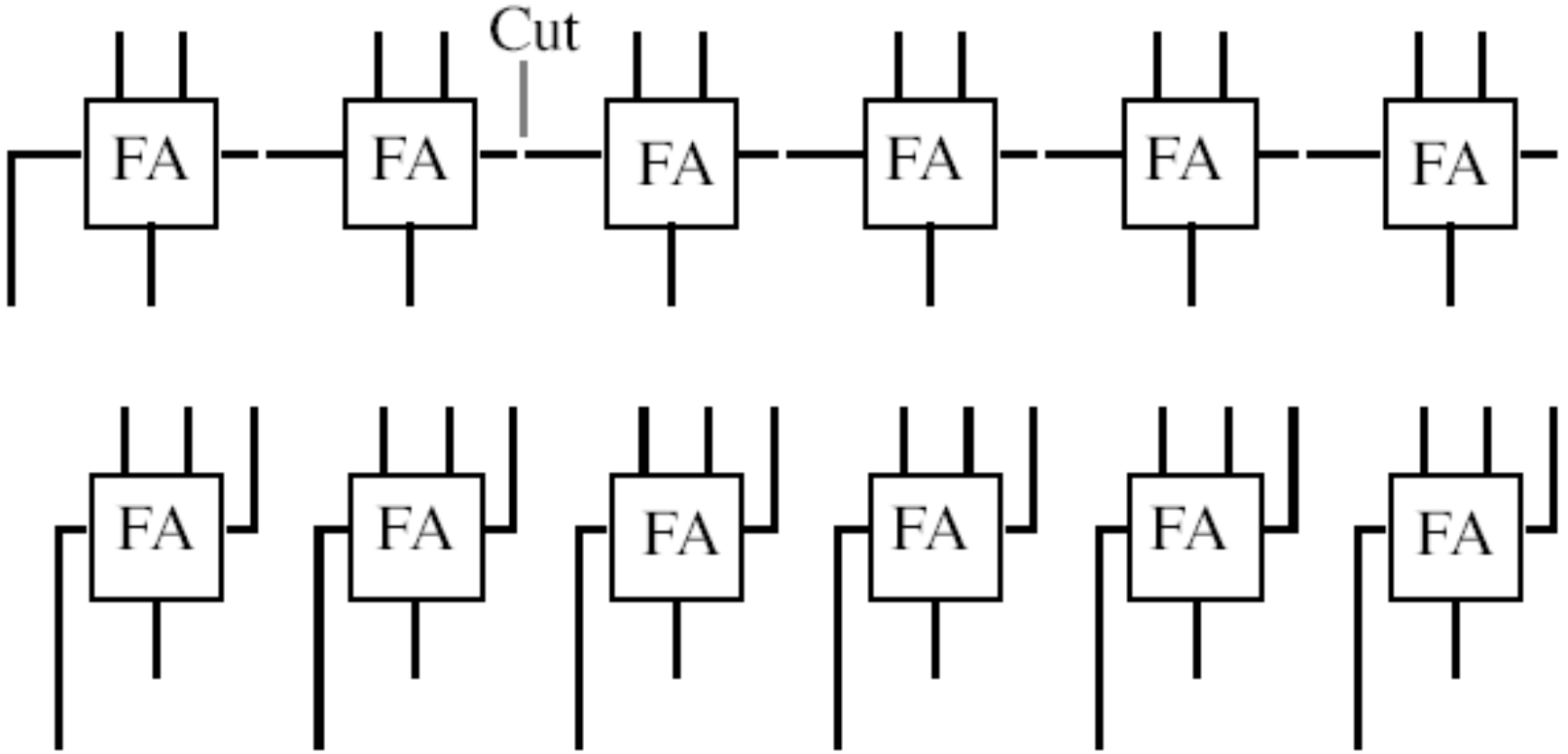


- Propagation delay ~2N

Carry Save Adder (CSA)

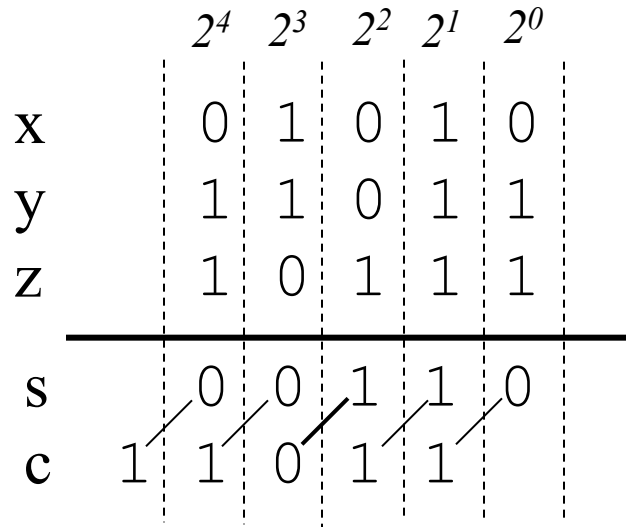


A Ripple-Carry vs. Carry-Save Adder



Operation of a Carry Save Adder (CSA)

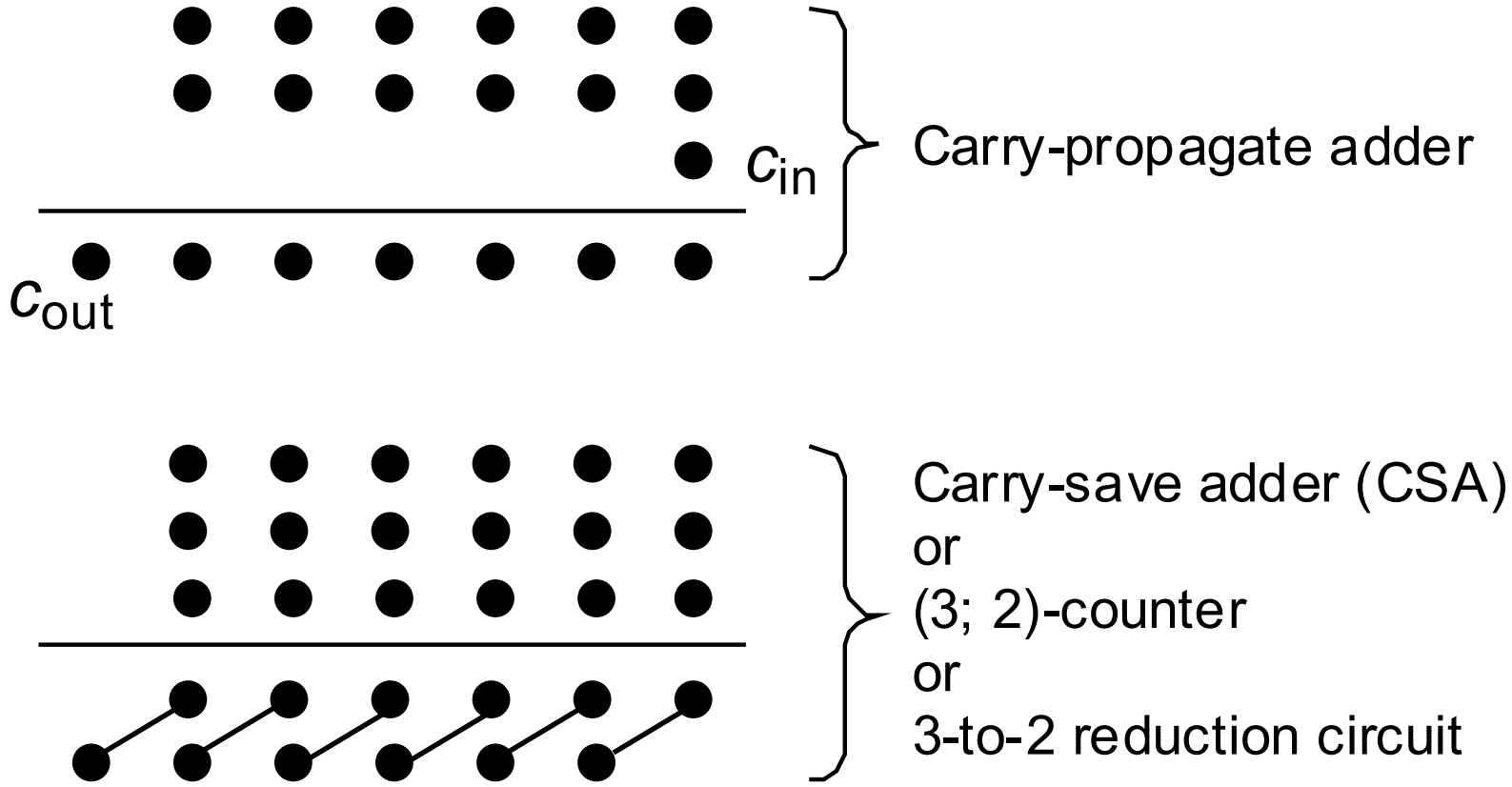
Example



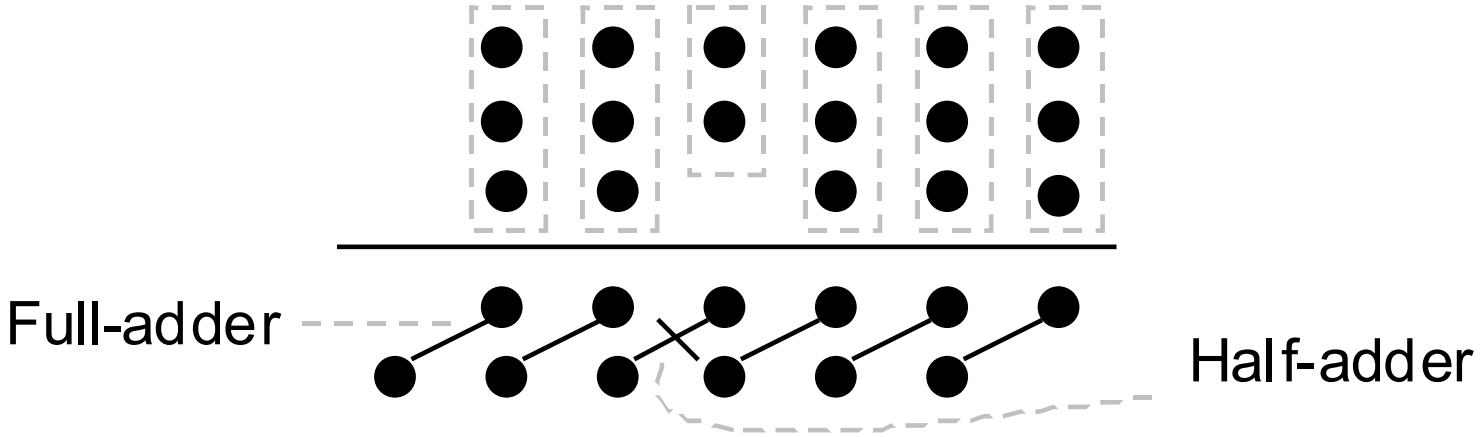
$$x+y+z = s + c$$

$$s_i = x_i \oplus y_i \oplus z_i$$
$$c_i = x_i y_i + y_i z_i + z_i x_i$$

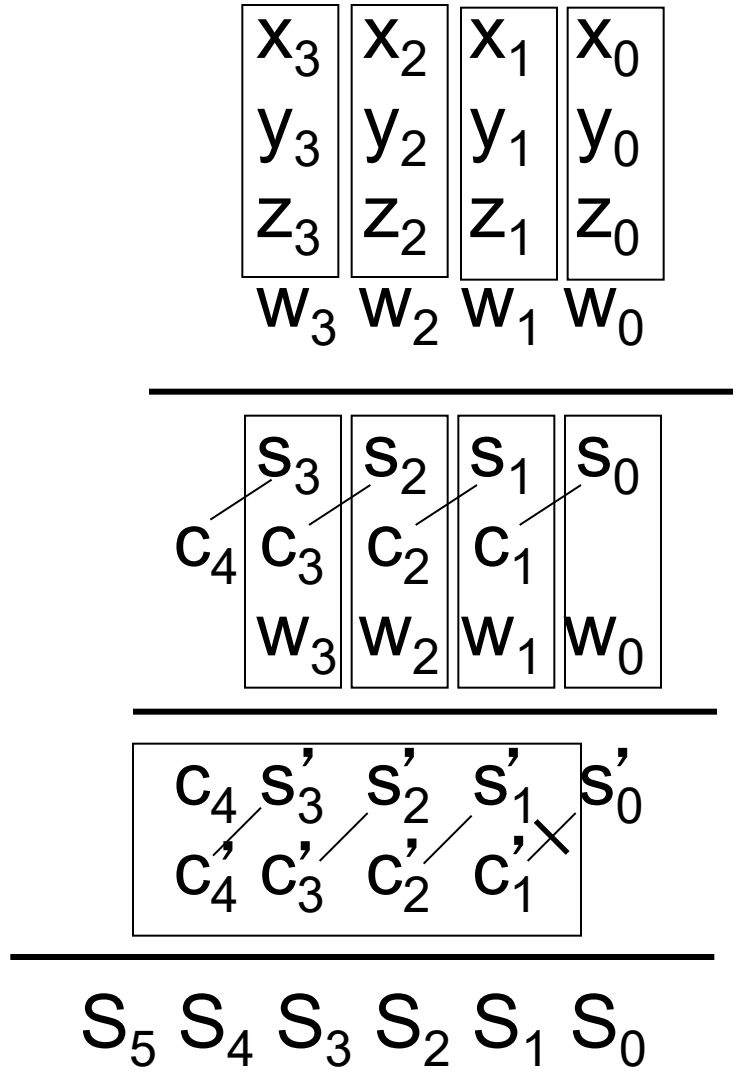
Carry propagate and carry-save adders in dot notation



Specifying full- and half-adder blocks in dot notation



Carry-save adder for four operands



Carry-save adder for four operands