Introduction to Boole algebra

Binary algebra

Boole algebra

- George Boole’s book released in 1847
- We have only two digits: true and false
- We have NOT, AND, OR, XOR etc operations
- We have axioms and theorems
- In computers represented by bits 1 and 0
- In circuits LOW and HIGH voltages
- Operations implemented in hardware as gates or more precisely as certain configuration of transistors

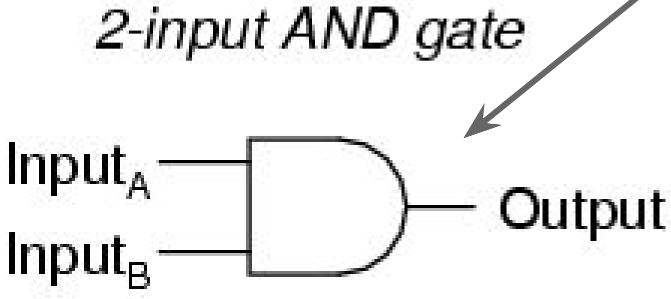
Boole operations

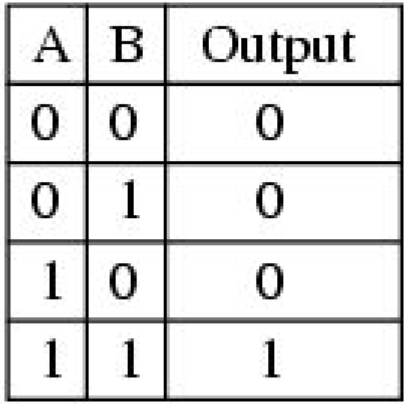
AND, OR, XOR etc

Conjunction (AND)

- True if both operands are true
- Scientific: A ∧ B
- Alternatively: A . B
- C/Java: A && B
- Python: A and B
- Bitwise: A & B

ANSI symbol

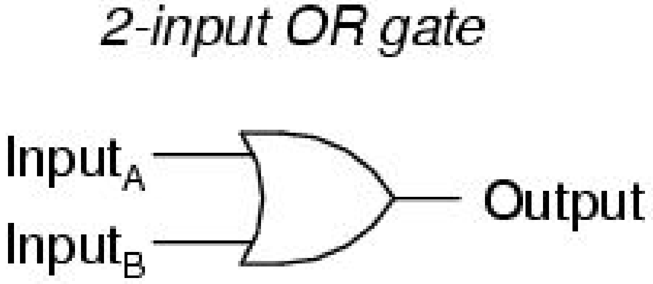


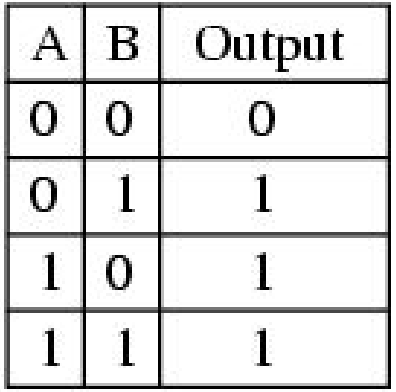


A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Disjunction (OR)

- True if at least one of the operands is true
- Scientific: A v B
- Alternatively: A + B
- C/Java: A || B
- Python: A or B
- Bitwise: A | B

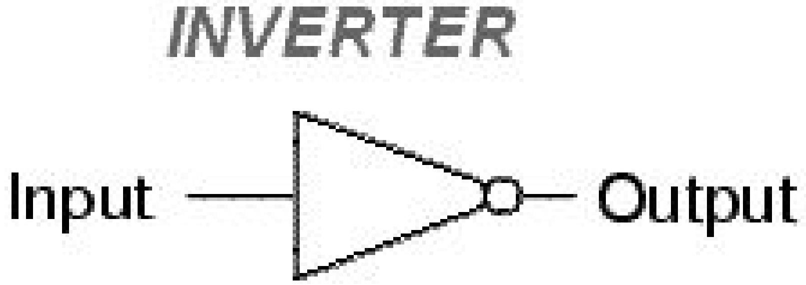


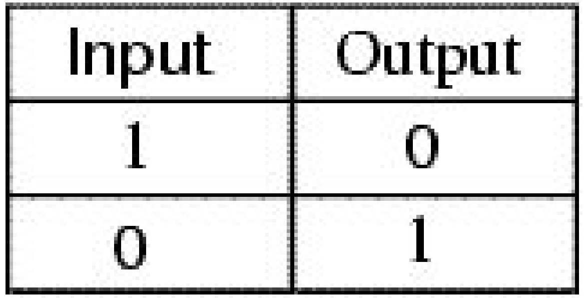


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Negation (NOT)

- True if and only if A is false
- Scientific: ¬A
- C/Java: !A
- Python: not A
- Bitwise: ~A

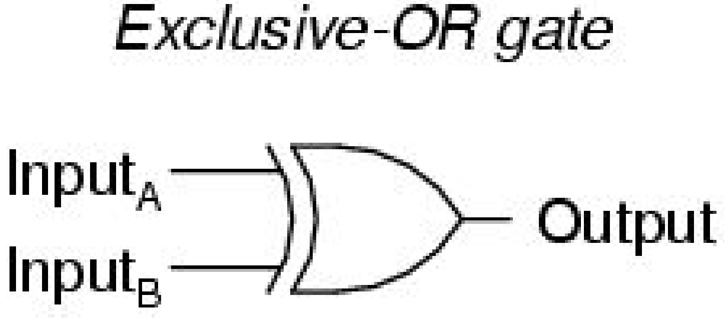


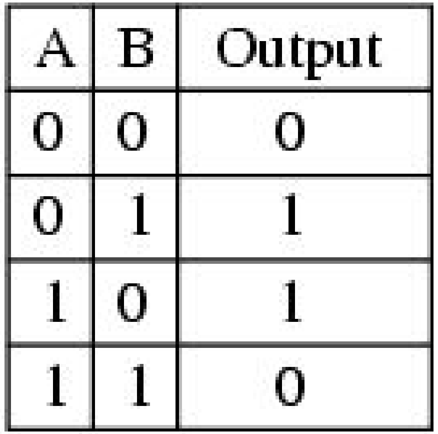


Input	Output
1	0
0	1

Exclusive OR (XOR)

- True if and only if A and B are unequal
- Scientific: $A \veebar B$
- Alternatively: $A \oplus B$
- Bitwise XOR in Python/C/Java: A ^ B





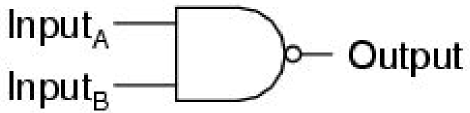
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Negated AND (NAND)

- Output of AND gate is negated
- Commonly written as:

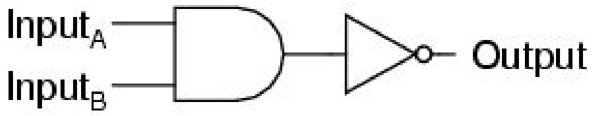
$$\overline{A \cdot B} \text{ or } A \uparrow B$$

2-input NAND gate



A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Equivalent gate circuit

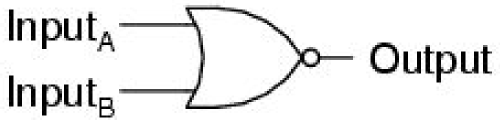


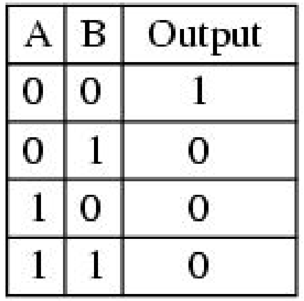
Negated OR (NOR)

- Output of OR gate is negated
- Commonly written as:

$$\overline{A+B} \text{ or } A - B$$

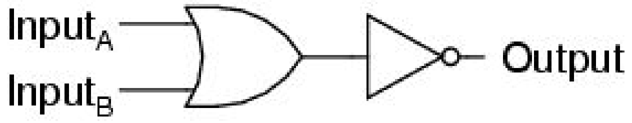
2-input NOR gate





A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

Equivalent gate circuit



Axioms and theorems

- Identity:
  - X OR 0 = X
  - X AND 1 = X
- Null:
  - X OR 1 = 1
  - X AND 0 = 0
- Idempotency:
  - X AND X = X
  - X OR X = X

Axioms and theorems

- Involution: NOT NOT X = X
- Complementarity:
  - X OR NOT X = 1
  - X AND NOT X = 0
- Commutativity:
  - X OR Y = Y OR X
  - X AND Y = Y AND X

Axioms and theorems

- Distributivity:
  - X AND (Y OR Z) = X AND Y OR X AND Z
  - X OR (Y AND Z) = X AND Y OR X AND Z
- Uniting:
  - X AND Y OR X AND NOT Y = X
  - (X OR Y) AND (X OR NOT Y) = X
- And so forth

Disjunctive normal form (DNF)

- Disjunction of clauses, where a clause is a conjunction of literals
- Simply put it’s an OR of AND-s:
  (NOT A AND B) OR (A AND B) OR (...) OR (...)
- Karnaugh’s map output is DNF
- Further optimizations/substitutions can be performed on DNF

Gates in hardware

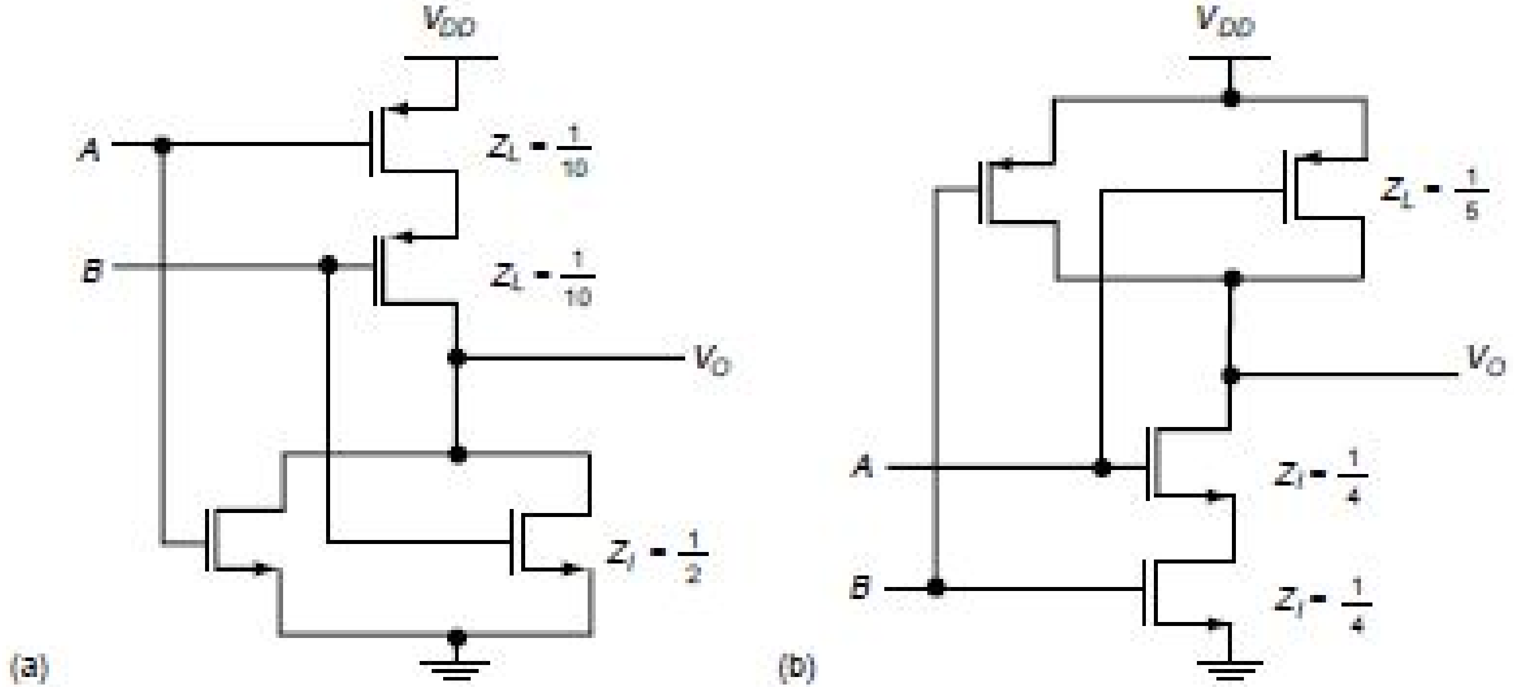
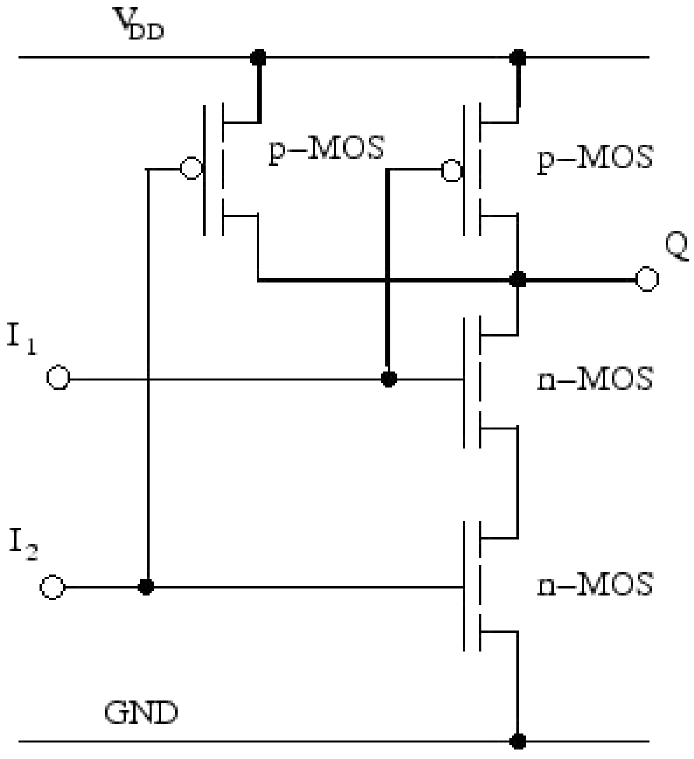
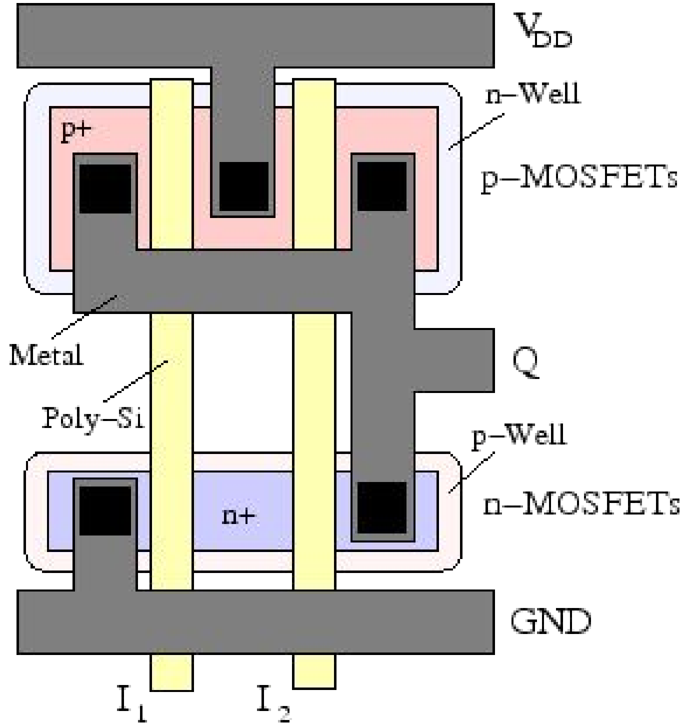


FIGURE 8.18 CMOS gates: (a) CMOS NOR gate, (b) CMOS NAND gate.

NAND with CMOS





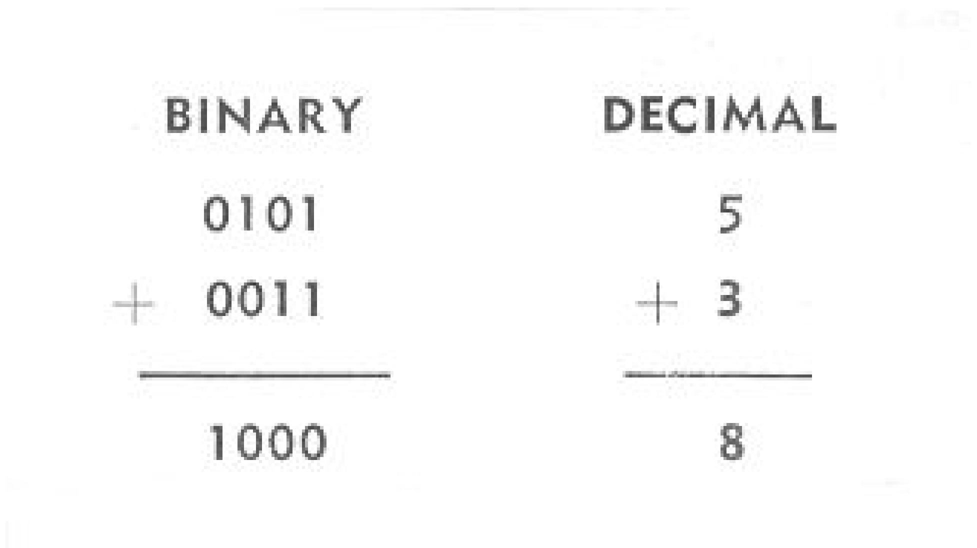
Latency of approximately 4ps

Adding in hardware

Half adder, full adder, carry ripple adder

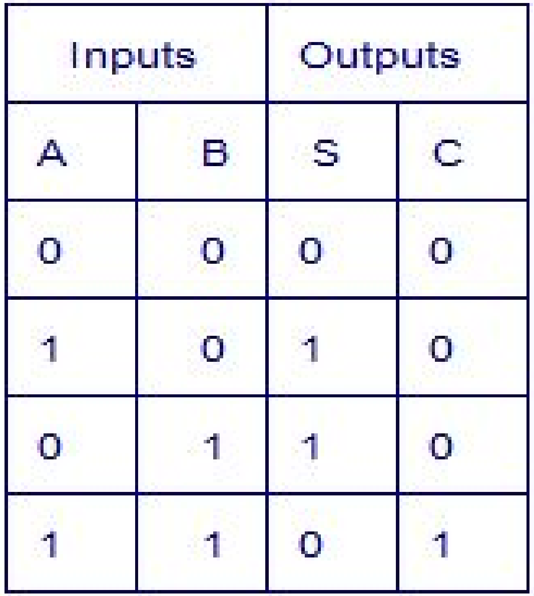
Binary addition

- Essentially same as in decimal
- Bit is carried with 1+1
- How can we implement this in Boole algebra?



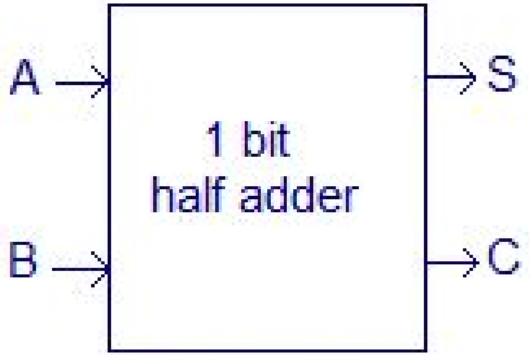
BINARY	DECIMAL
0101	5
+ 0011	+ 3
1000	8

Half adder

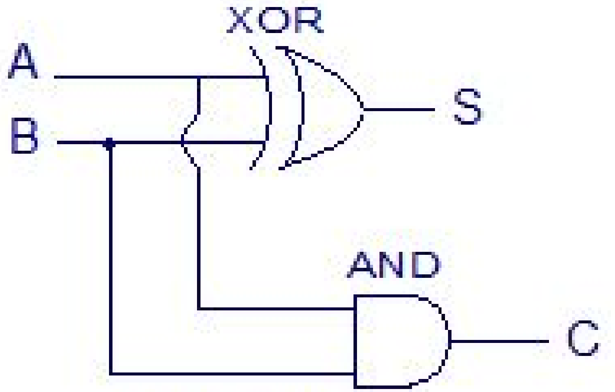


Inputs		Outputs	
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Truth table



Schematic



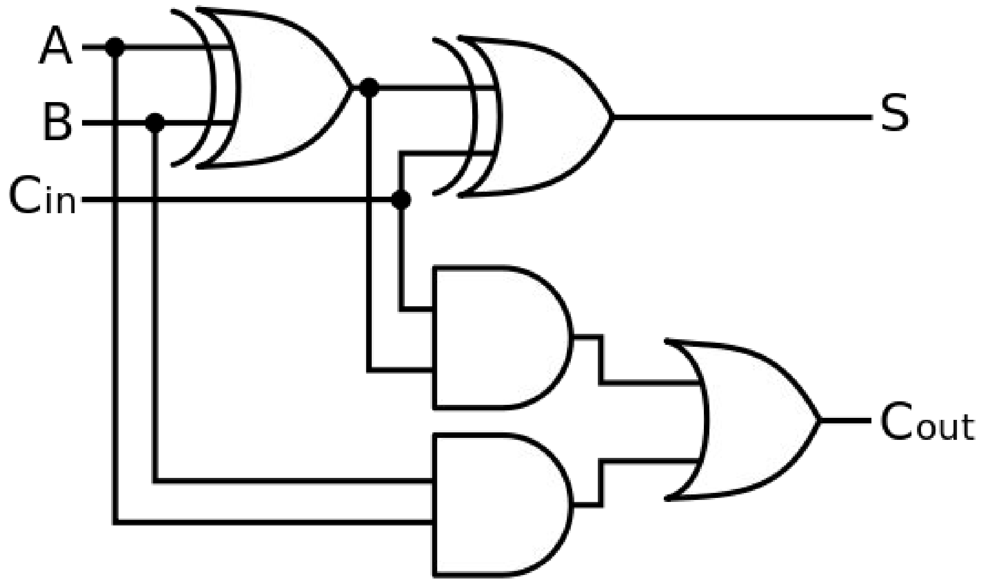
Realization

Sum = A XOR B
Carry = A AND B

Read: odd number of 1-s on inputs
Read: both inputs are 1

Full adder

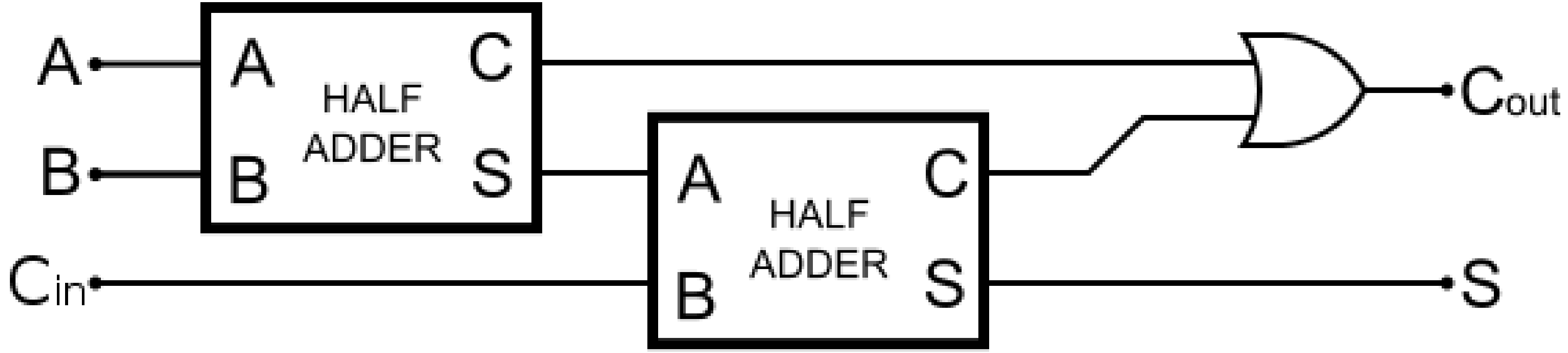
Input			Output	
A	B	Cin	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



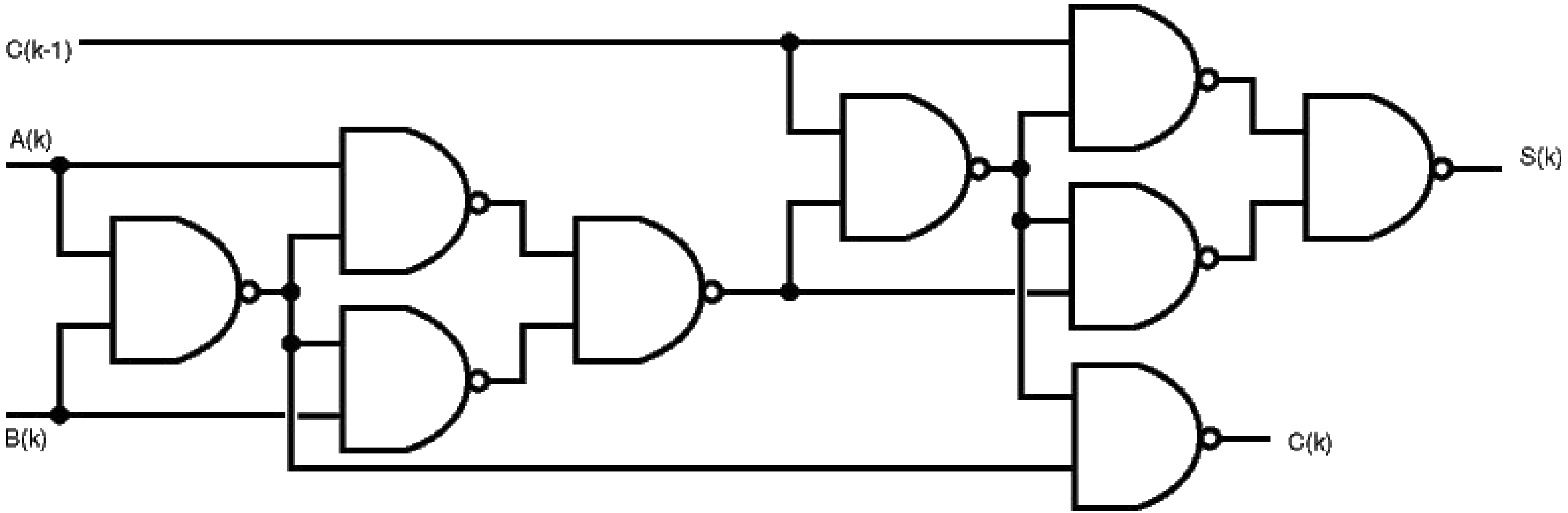
Sum = A XOR B XOR Cin
Carry = (A XOR B) AND Cin OR (A AND B)

Read: odd number of 1-s on inputs
Read: two or more 1-s on inputs

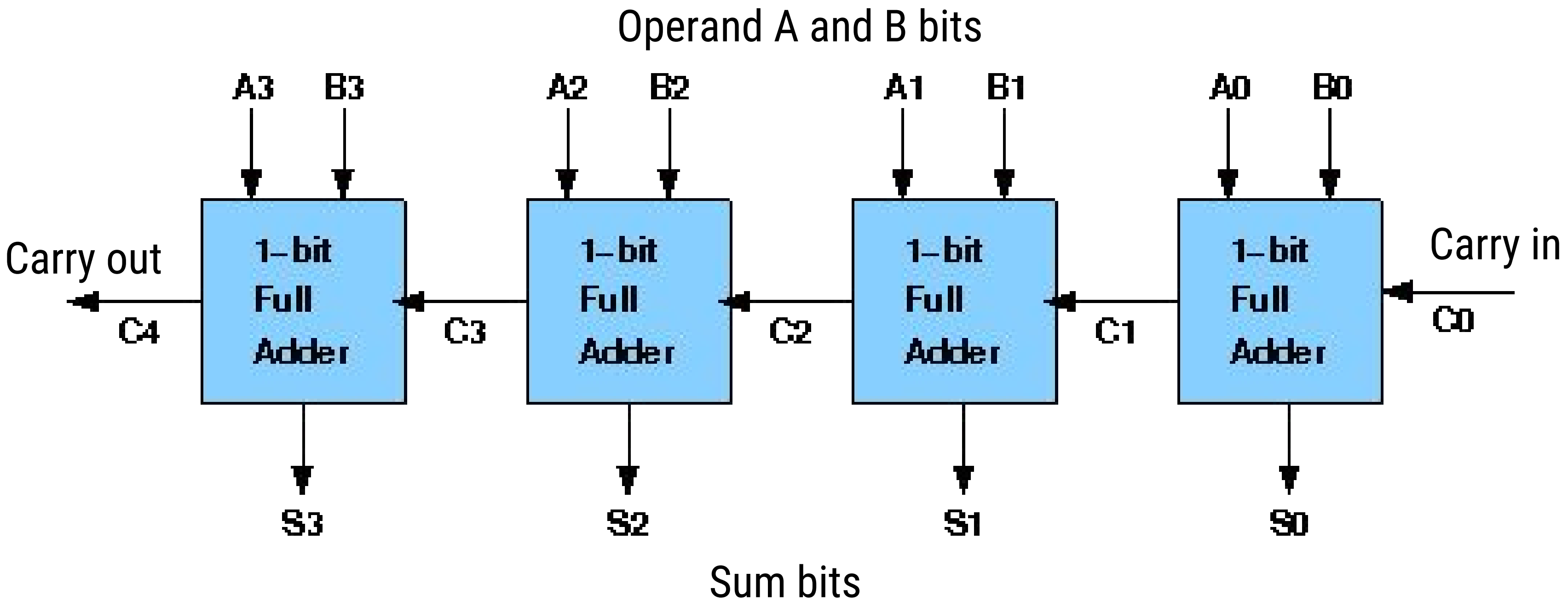
Full adder



Full adder with NAND gates



Carry ripple adder



Hitting the ceiling

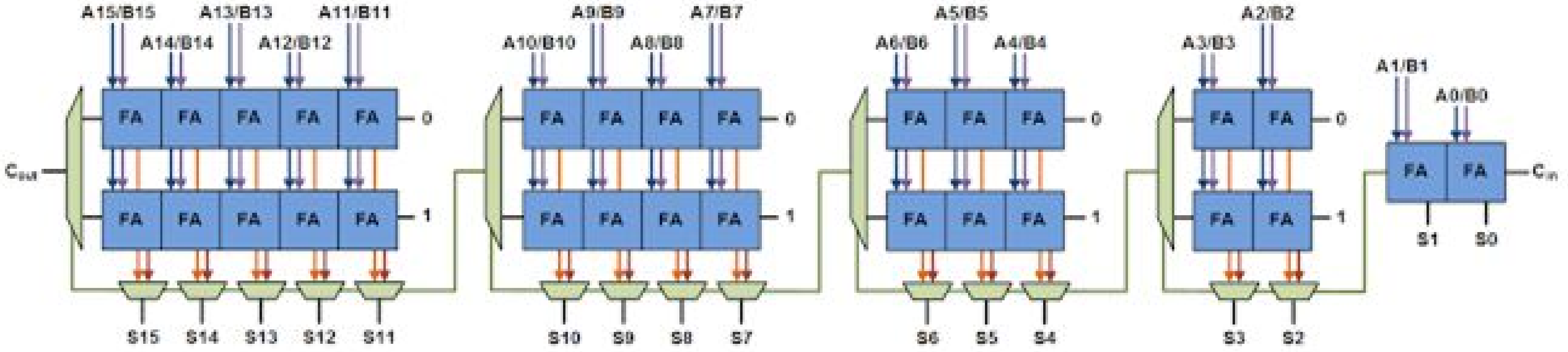
- Consider clock frequency of 2GHz (500ps)
- Within one clock period we could chain up to 500ps / 4ps ≃ 125 NAND gates in series
- Full adder carry in-carry out path contains 2 NAND gates 500ps / 8ps ≃ 62 full adders chained
- So 64 bit *carry-ripple adder* at 2GHz is problematic if not even impossible

Solutions

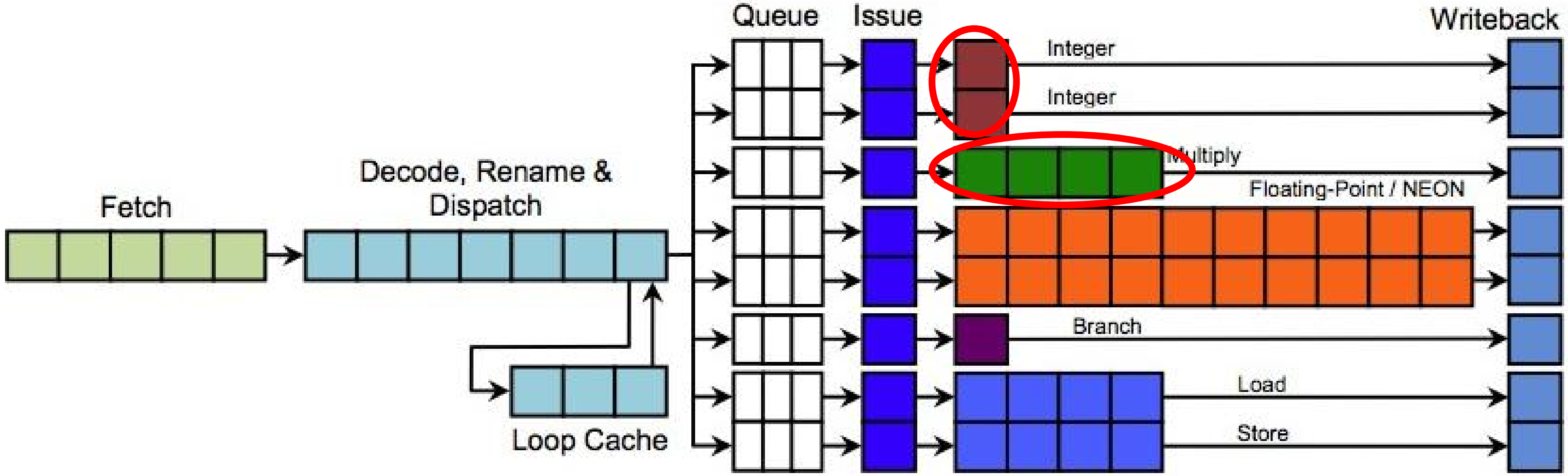
- Overclocking: Increase voltage to reduce gate delay, problem is increased thermal dissipation
- Better circuit design with less gates in critical path
- Span the operation over several processor cycles
- Law of diminishing returns can be observed

Carry select adder

- Critical path is shorter in terms of gates
- 30 FA-s instead of 16 FA-s -> more resources used
- Mux logic



Spanning operations

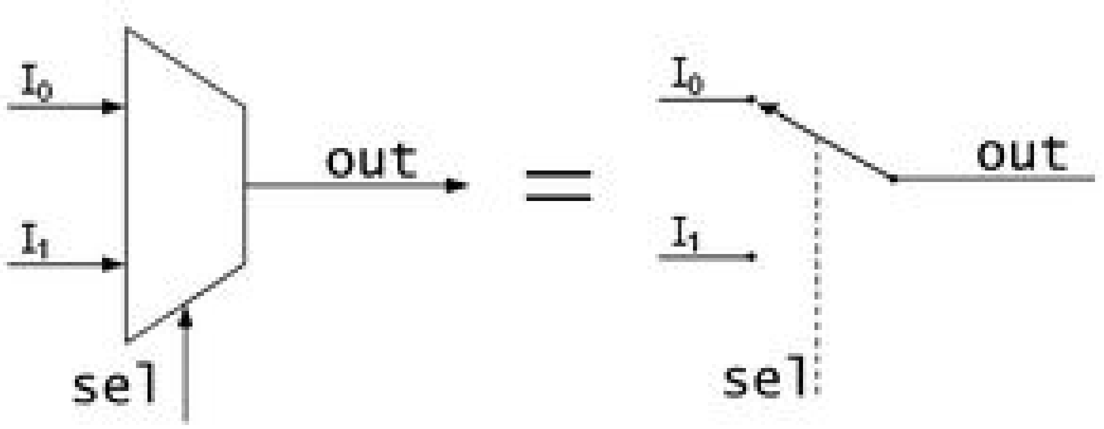


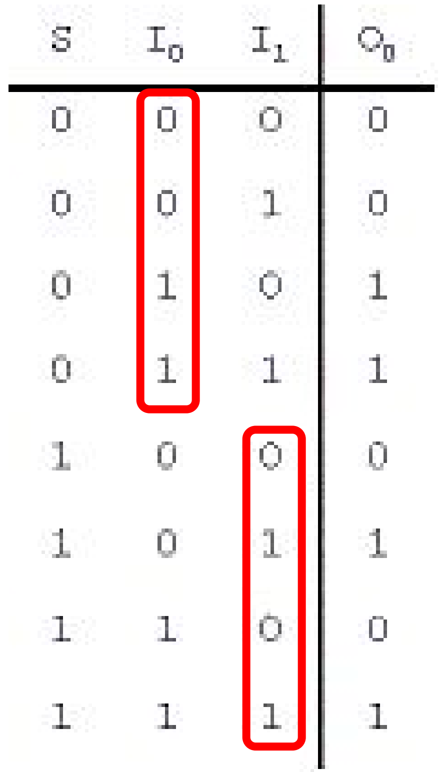
Multiplexer

Selecting signals

2:1 multiplexer

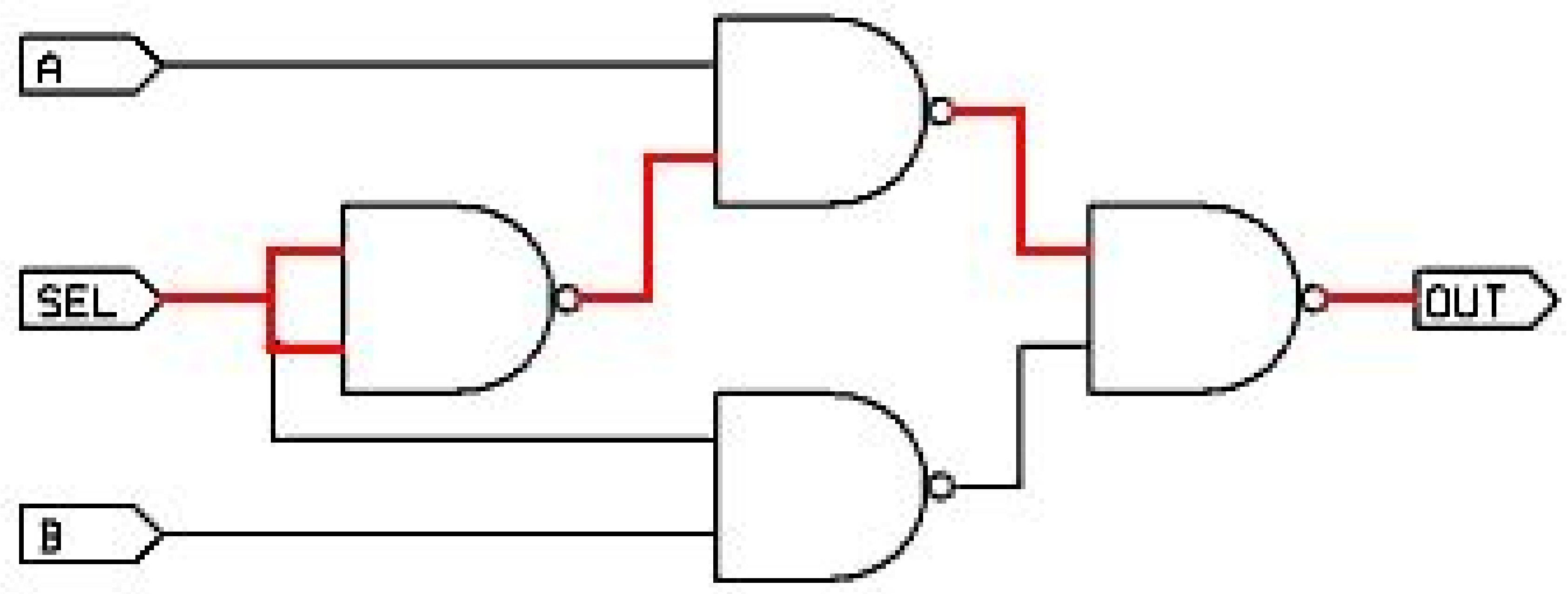
- Selects between inputs





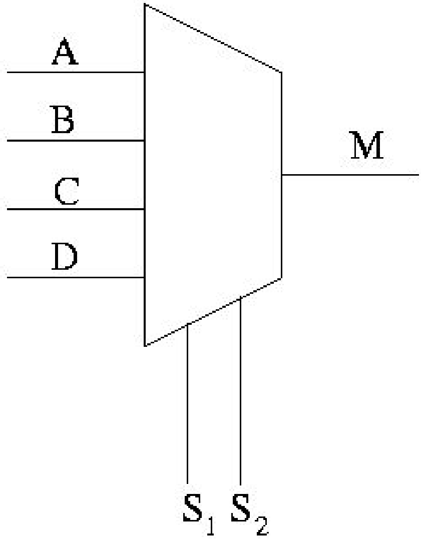
S	I_0	I_1	O_0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

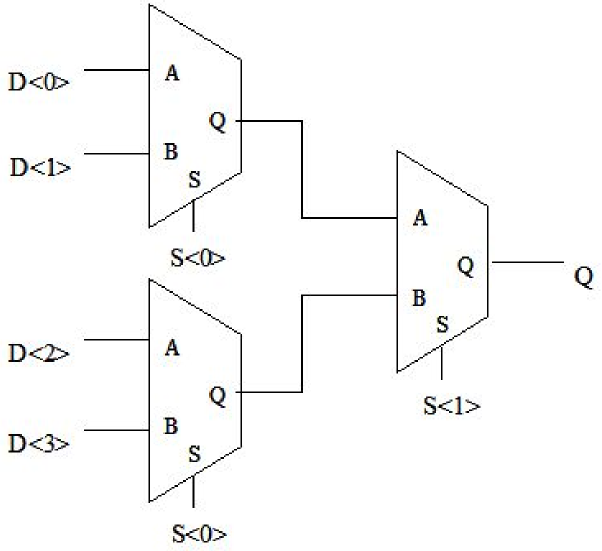
2:1 mux with NAND gates



4:1 mux

- Use 2 selector inputs to select between 4 data inputs





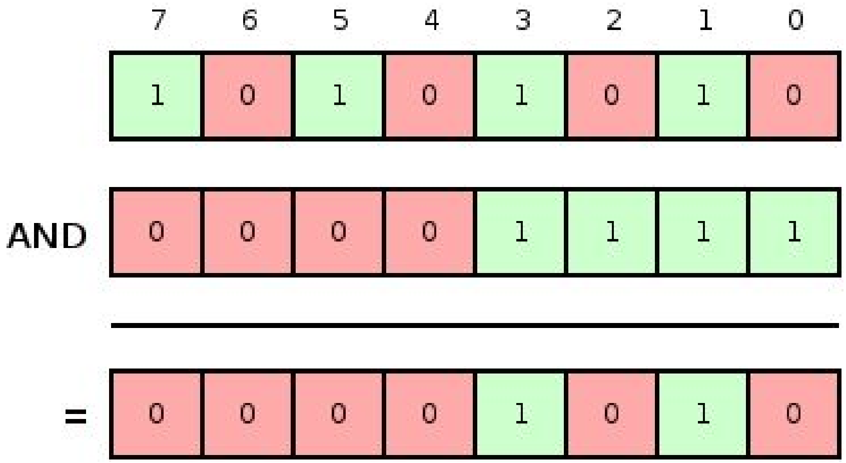
Multiplexer uses

- Selecting input value from registers
- Selecting result within ALU for output value
- Converting parallel data to serial
- Implementing programmable logic
    - Use input pins as programming bits
    - Use selector bits as logic inputs
    - That’s basically how FPGA-s work

Bitwise operations

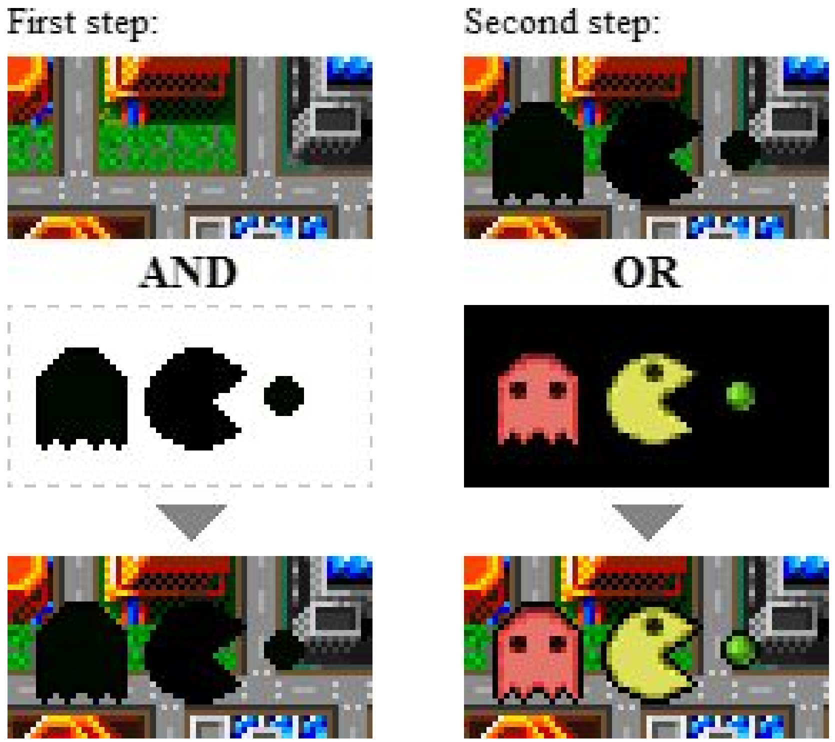
Bitwise operations

- Perform AND, OR, XOR, etc logic operation on registers
- In C/Java/Python:
  - AND: a & b
  - OR: a | b
  - XOR: a ^ b



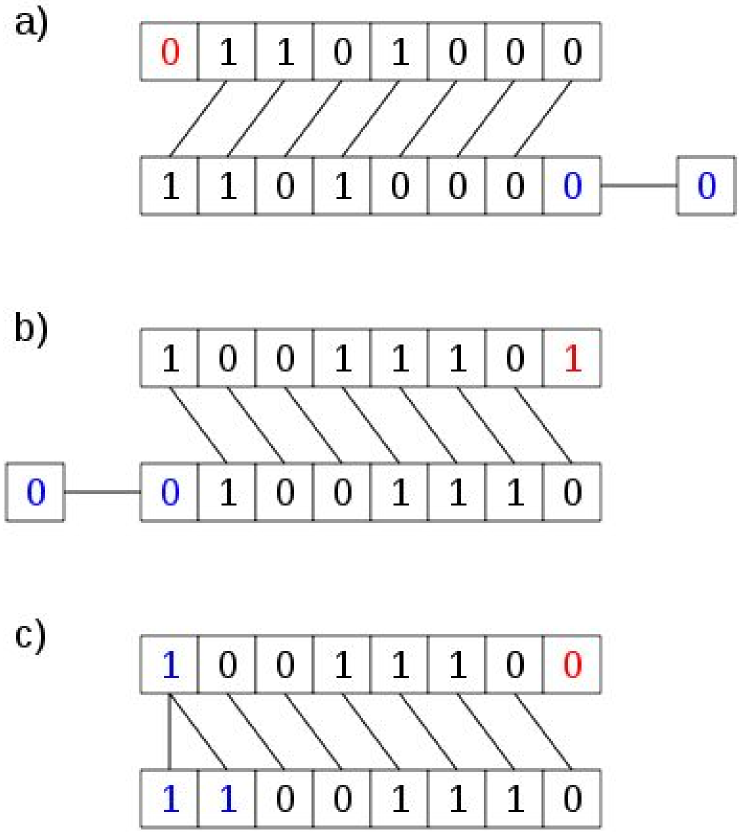
Bitwise operation uses

- Overlaying sprites
- These are images or animations integrated into larger scene



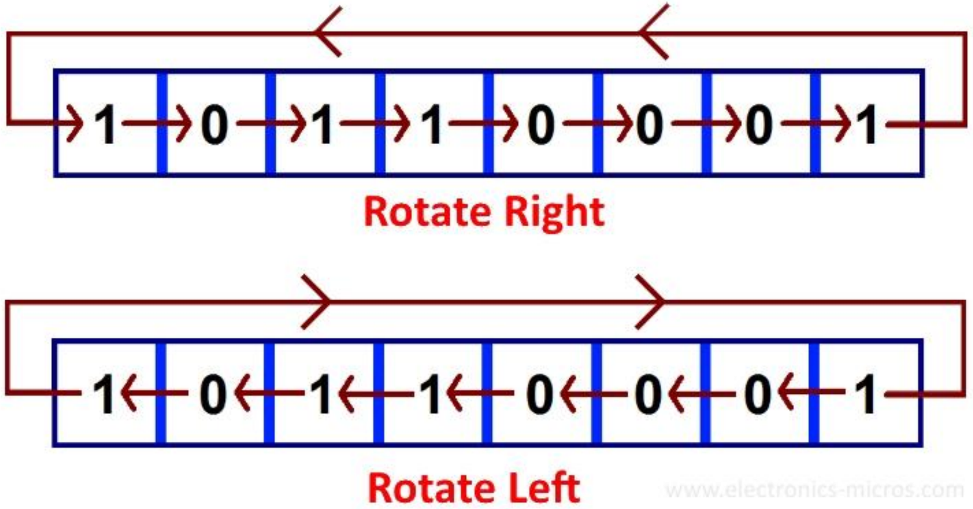
Bit shifting

- Perform quick integer mult/div with 2 and it’s powers
- In C/Java/Python
  - x << y
  - x >> y
- Applied in 3D graphics



Circular shift (bit rotate)

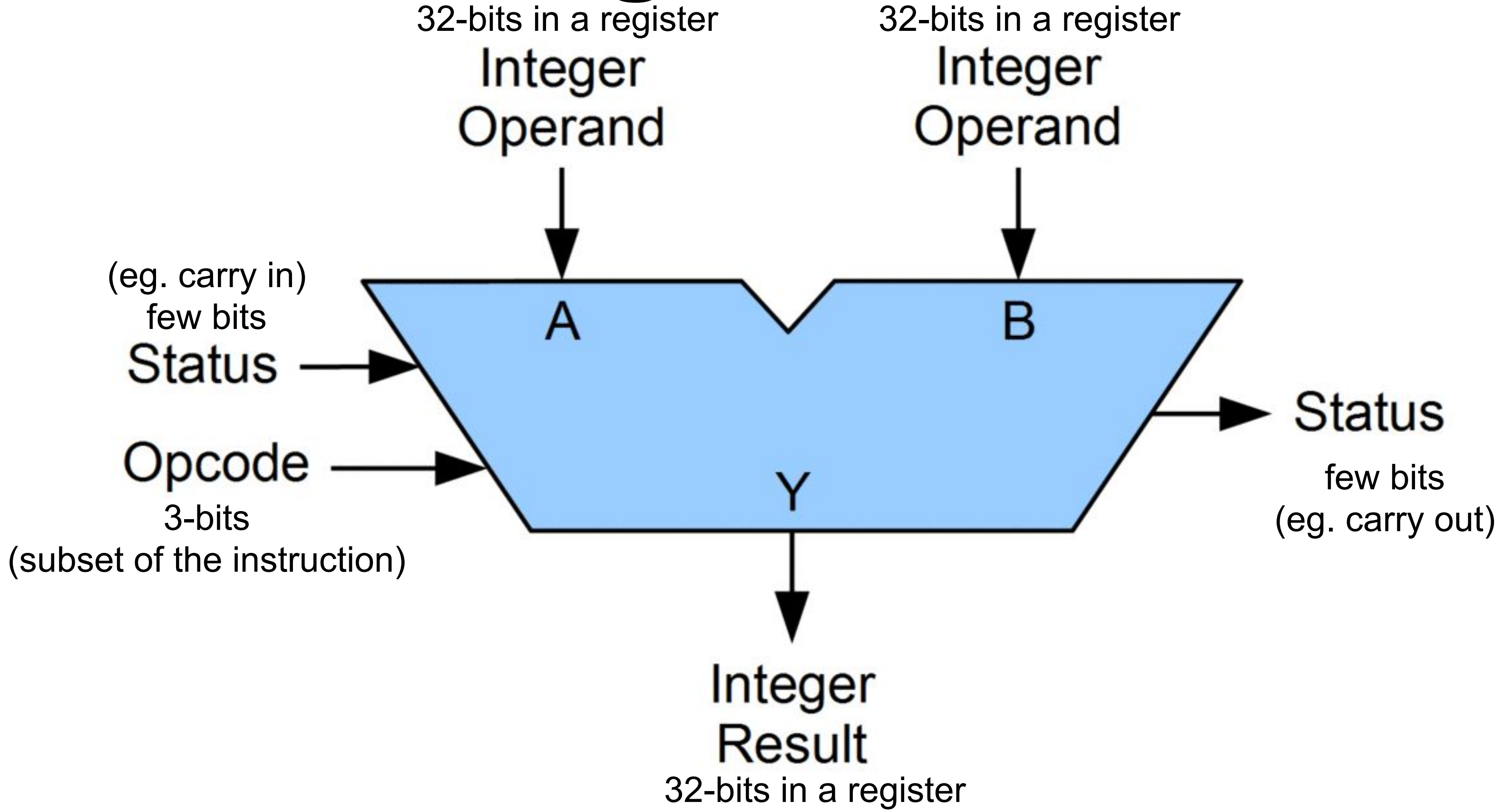
- Applied in cryptography to permutate bit sequences
- Not exposed in most programming languages, can be invoked via inlined assembly



ALU

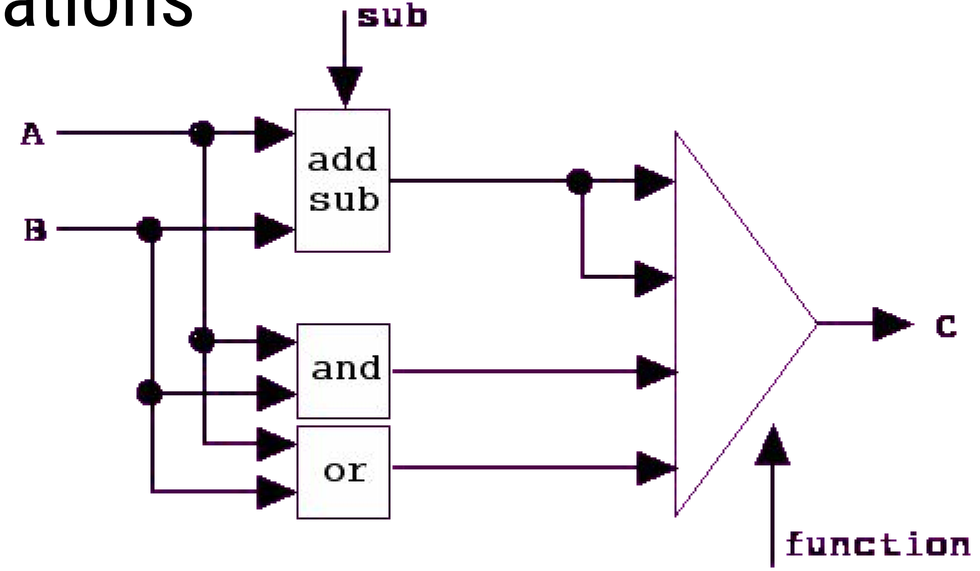
Arithmetic logic unit

Arithmetic-logic unit

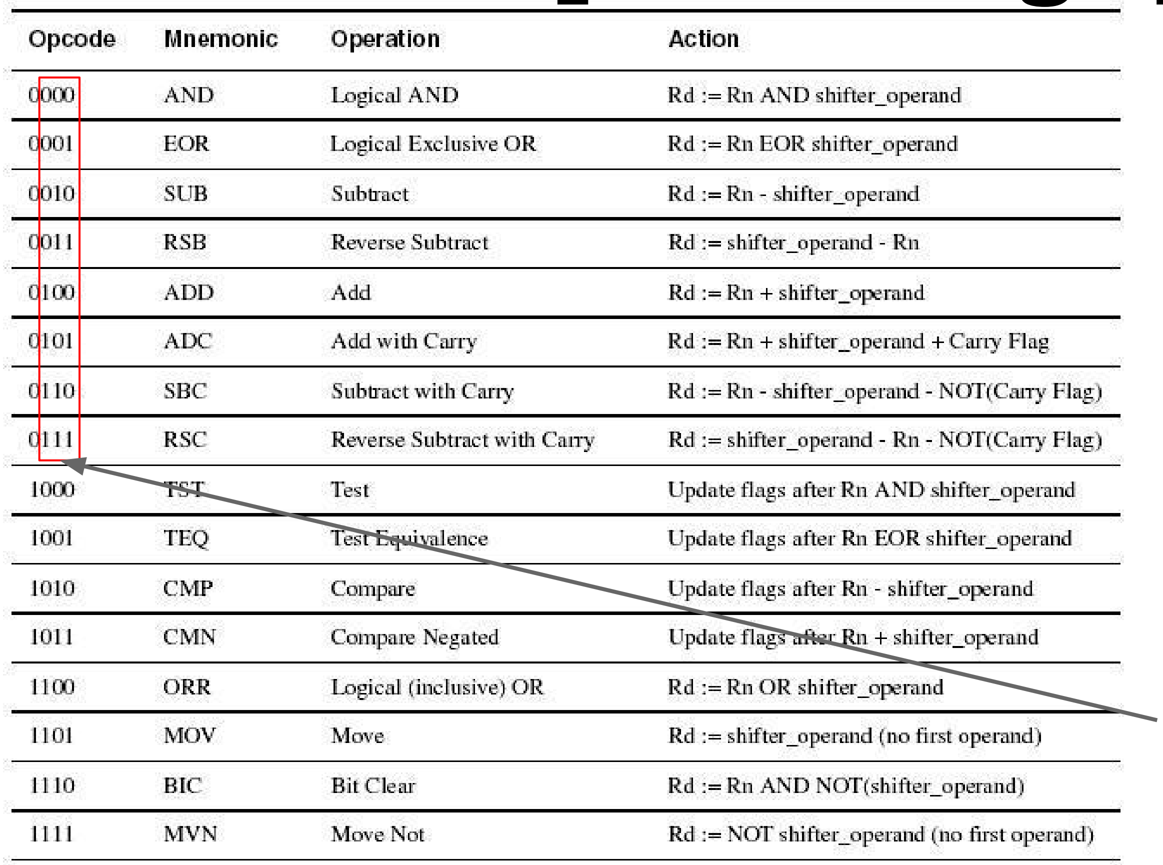


Arithmetic logic unit inside

- Perform different operations
- Calculate all possible outputs
- Use multiplexer to select correct one



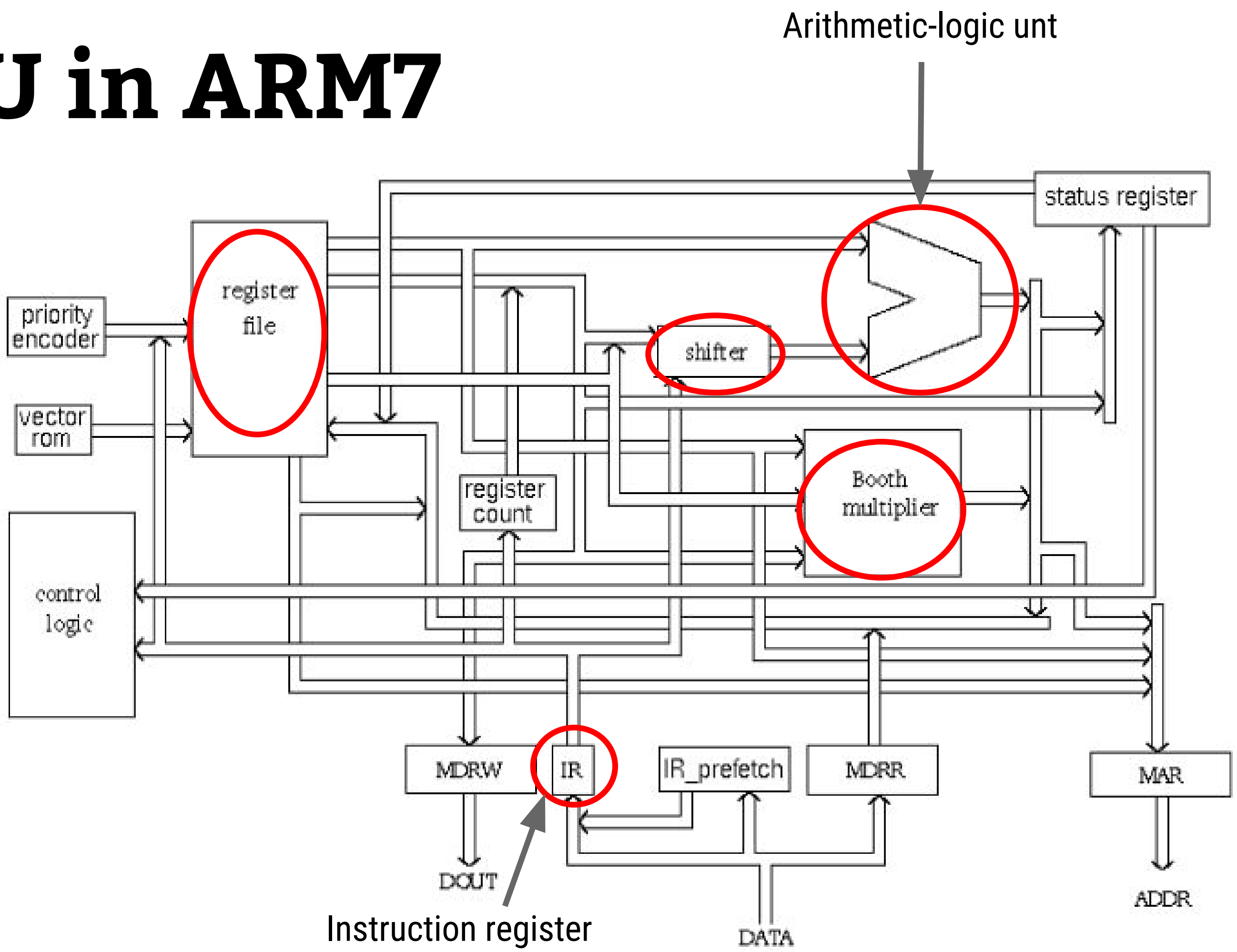
ARM7 data processing opcodes



Opcode	Mnemonic	Operation	Action
0000	AND	Logical AND	Rd := Rn AND shifter_operand
0001	EOR	Logical Exclusive OR	Rd := Rn EOR shifter_operand
0010	SUB	Subtract	Rd := Rn - shifter_operand
0011	RSB	Reverse Subtract	Rd := shifter_operand - Rn
0100	ADD	Add	Rd := Rn + shifter_operand
0101	ADC	Add with Carry	Rd := Rn + shifter_operand + Carry Flag
0110	SBC	Subtract with Carry	Rd := Rn - shifter_operand - NOT(Carry Flag)
0111	RSC	Reverse Subtract with Carry	Rd := shifter_operand - Rn - NOT(Carry Flag)
1000	TST	Test	Update flags after Rn AND shifter_operand
1001	TEQ	Test Equivalence	Update flags after Rn EOR shifter_operand
1010	CMP	Compare	Update flags after Rn - shifter_operand
1011	CMN	Compare Negated	Update flags after Rn + shifter_operand
1100	ORR	Logical (inclusive) OR	Rd := Rn OR shifter_operand
1101	MOV	Move	Rd := shifter_operand (no first operand)
1110	BIC	Bit Clear	Rd := Rn AND NOT(shifter_operand)
1111	MVN	Move Not	Rd := NOT shifter_operand (no first operand)

These 3 bits look like directly mappable ALU opcodes

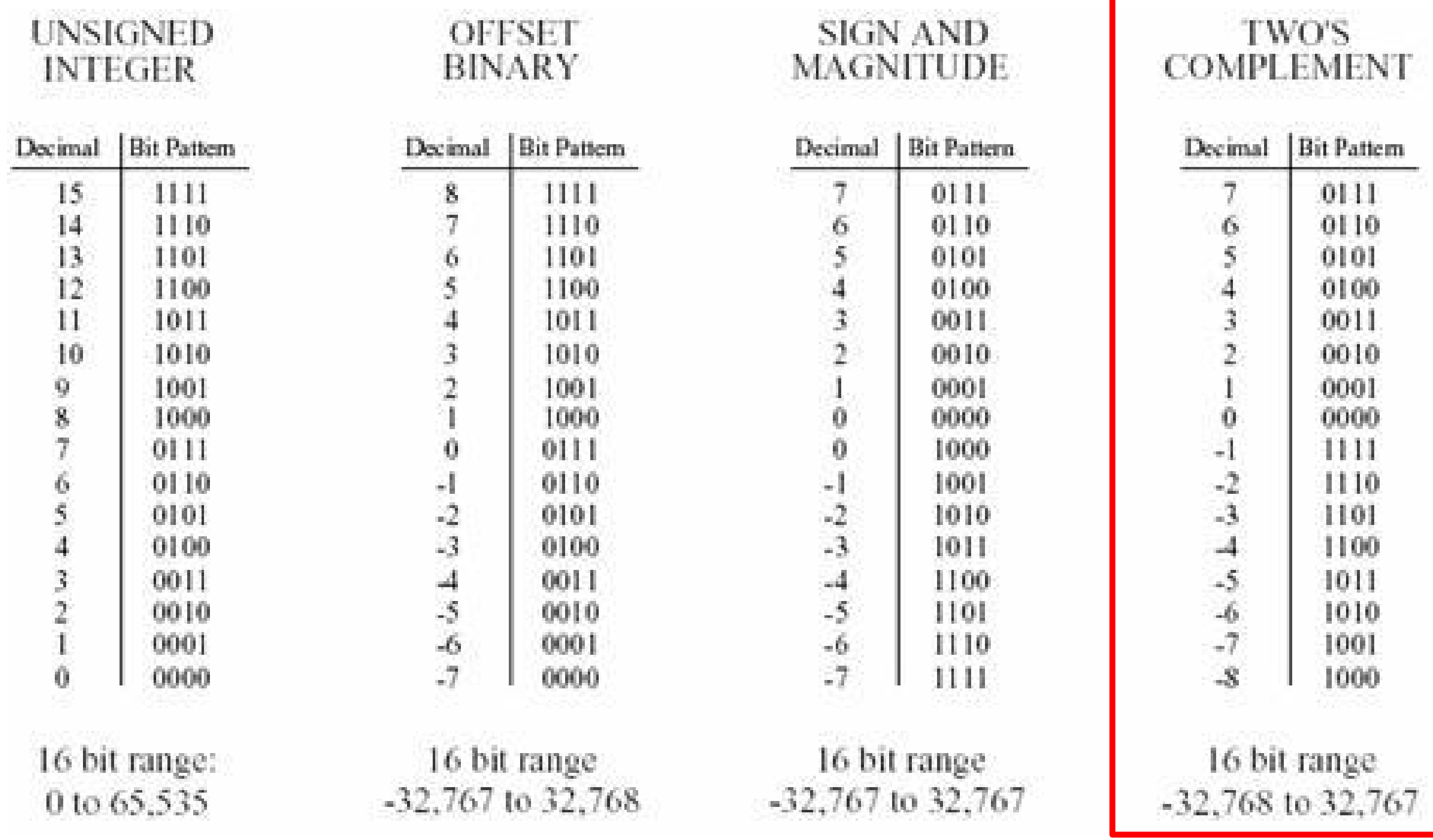
ALU in ARM7


Arithmetic-logic unt
status register
register file
priority encoder
vector rom
shifter
register count
Booth multiplier
control logic
MDRW
IR
IR_prefetch
MDRR
MAR
DOUT
DATA
ADDR
Instruction register

Subtraction

Two’s complement

Negative numbers in binary



UNSIGNED INTEGER

Decimal	Bit Pattern
15	1111
14	1110
13	1101
12	1100
11	1011
10	1010
9	1001
8	1000
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000

16 bit range:
0 to 65,535

OFFSET BINARY

Decimal	Bit Pattern
8	1111
7	1110
6	1101
5	1100
4	1011
3	1010
2	1001
1	1000
0	0111
-1	0110
-2	0101
-3	0100
-4	0011
-5	0010
-6	0001
-7	0000

16 bit range
-32,767 to 32,768

SIGN AND MAGNITUDE

Decimal	Bit Pattern
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

16 bit range
-32,767 to 32,767

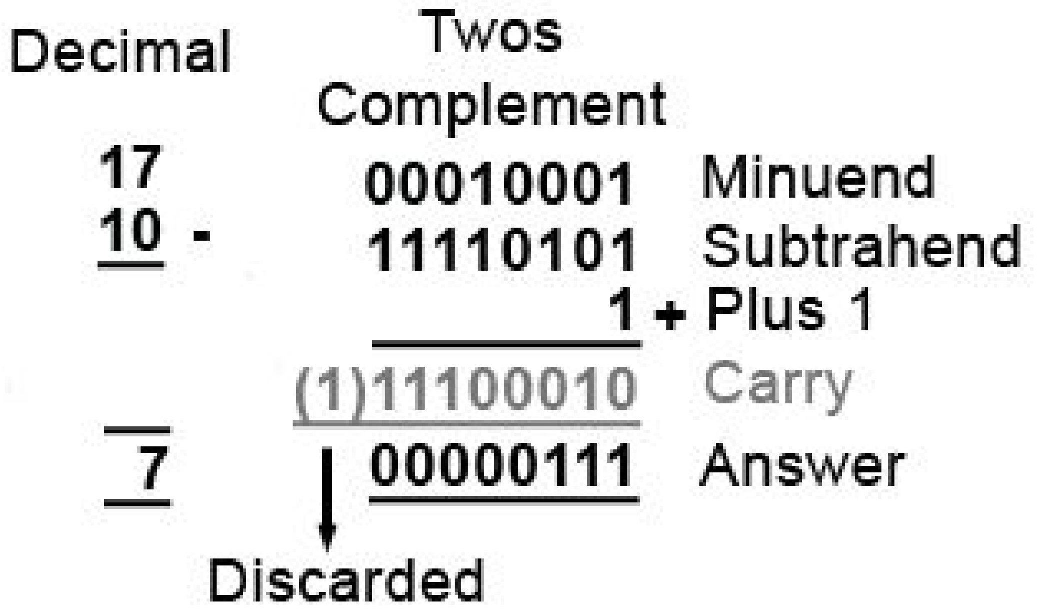
TWO'S COMPLEMENT

Decimal	Bit Pattern
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

16 bit range
-32,768 to 32,767

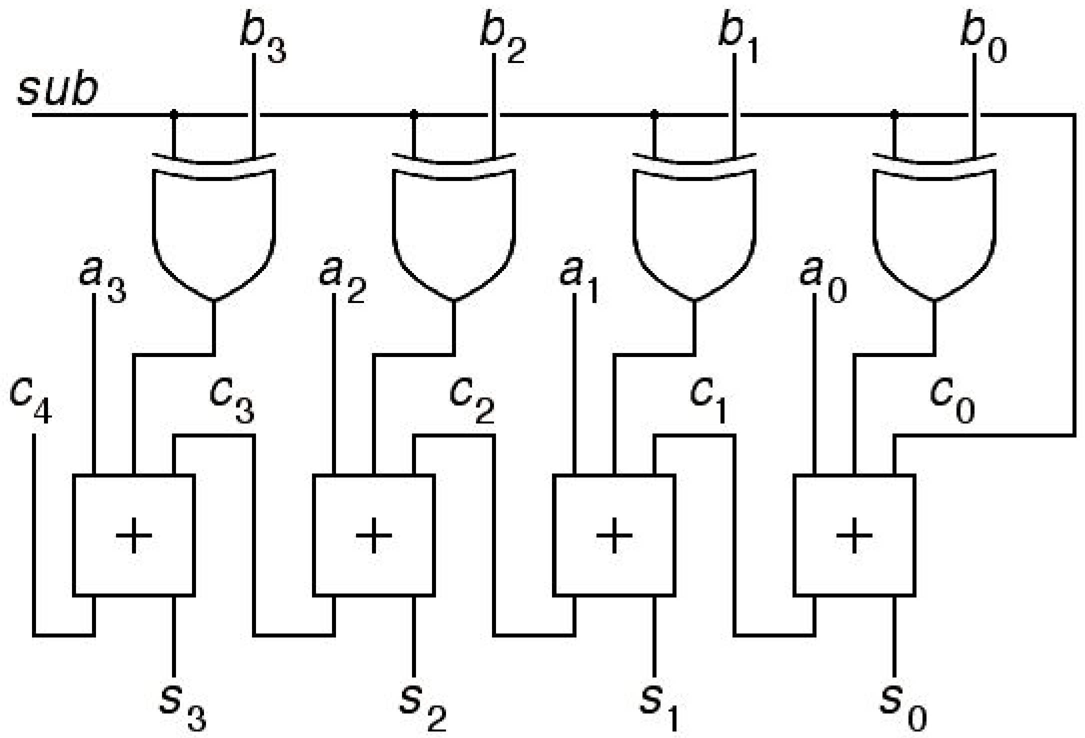
Binary subtraction

- Addition of negative number
- Minuend is added to inverted bits of subtrahend and 1 (due to offset)



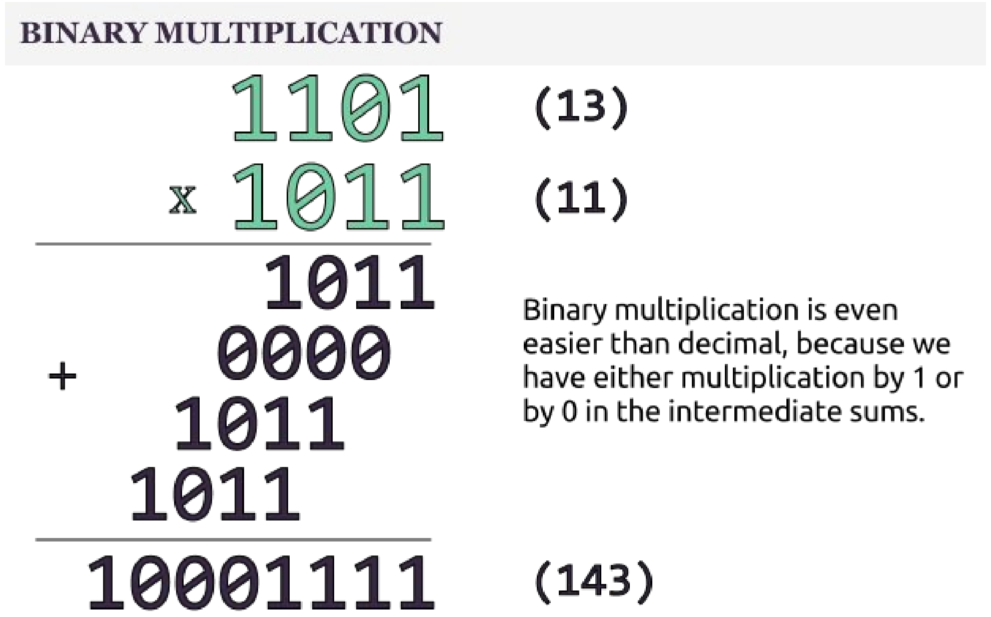
Subtraction in hardware

subtraction flag from the opcode inverts bits of second operand and sets carry in to 1



Binary multiplication

- Implemented with
  - AND operation
  - Binary addition
- Booth's multiplier
  - Group bits
  - Precalculate addends


BINARY MULTIPLICATION

```
     1101     (13)
   x 1011     (11)
   -------
     1011
+   0000
   1011
  1011
  --------
 10001111     (143)
```

Binary multiplication is even easier than decimal, because we have either multiplication by 1 or by 0 in the intermediate sums.

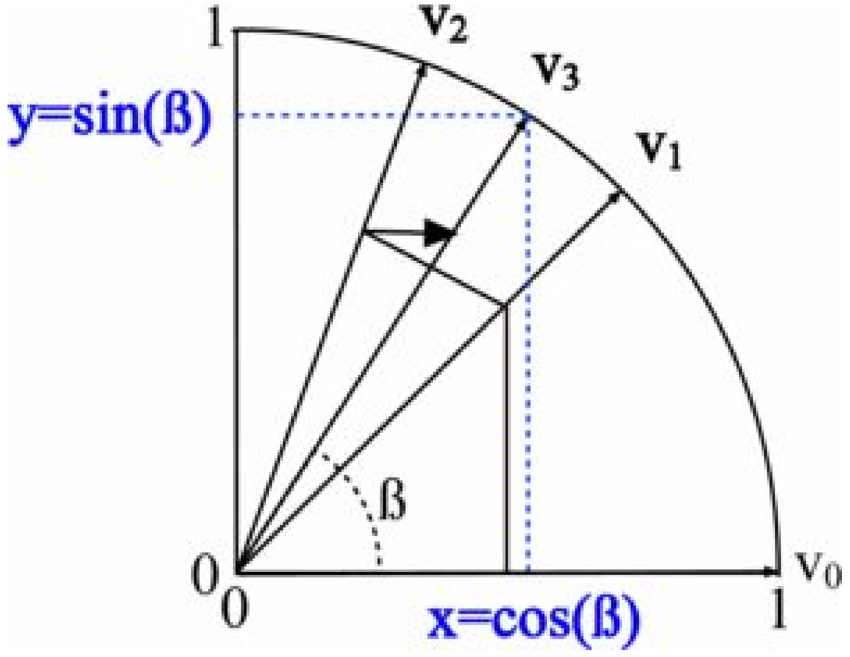
Beyond multiplication

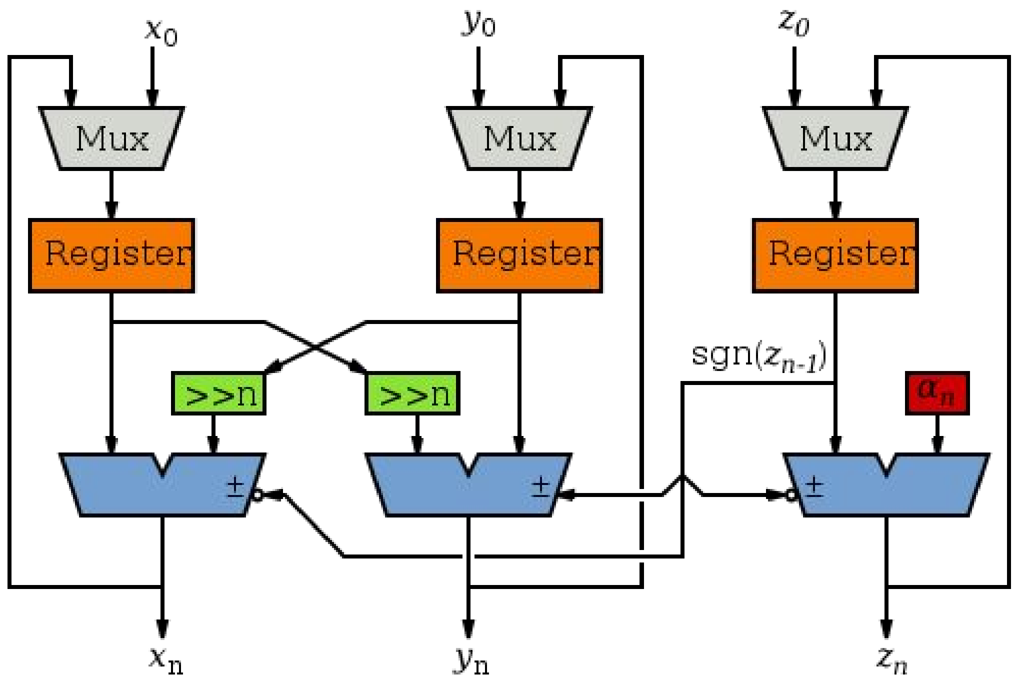
- There aren’t many interesting operations that can be performed on integers
- Most mathematical functions (div, sqrt, exp, sin, cos, tan) are approximations
- Possible implementations
  - Lookup table
  - Polynomial approximation
  - CORDIC

Old school hacks to get stuff faster

- Precalculate results of certain slow operations when the program is started
- Place the results in a lookup table (array)
- When looking up value find nearest two results
- Interpolate the result from two nearest results
- For sine, cosine use single lookup table for first 90 degrees only

COordinate Rotation DIgital Computer (CORDIC)





Fast $\frac{1}{\sqrt{x}}$ in Quake 3

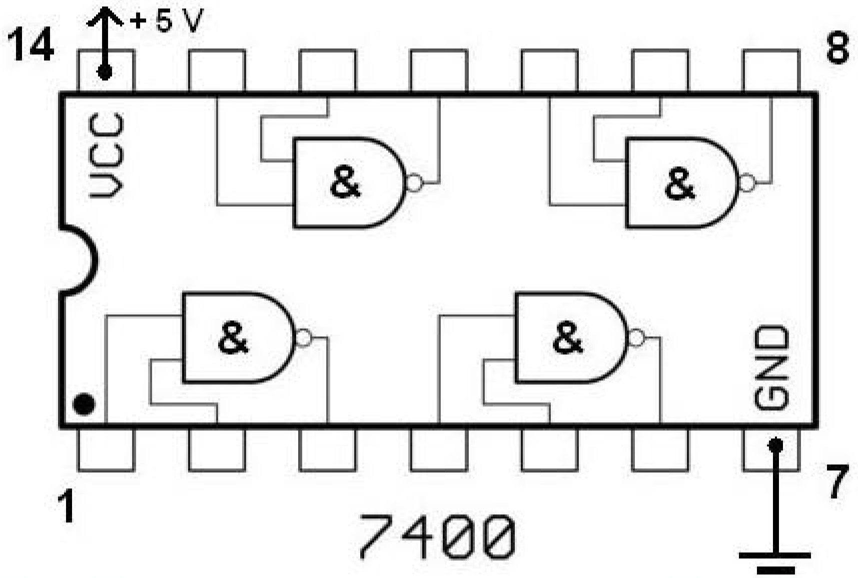
```c
float Q_rsqrt( float number ) {
     long i;
     float x2, y;
     const float threehalfs = 1.5F;
     x2 = number * 0.5F;
     y  = number;
     i  = * ( long * ) &y;           // evil floating point bit level hacking
     i  = 0x5f3759df - ( i >> 1 );    // what the fuck?
     y  = * ( float * ) &i;
     y  = y * ( threehalfs - ( x2 * y * y ) );
     return y;
}
```

Assignment

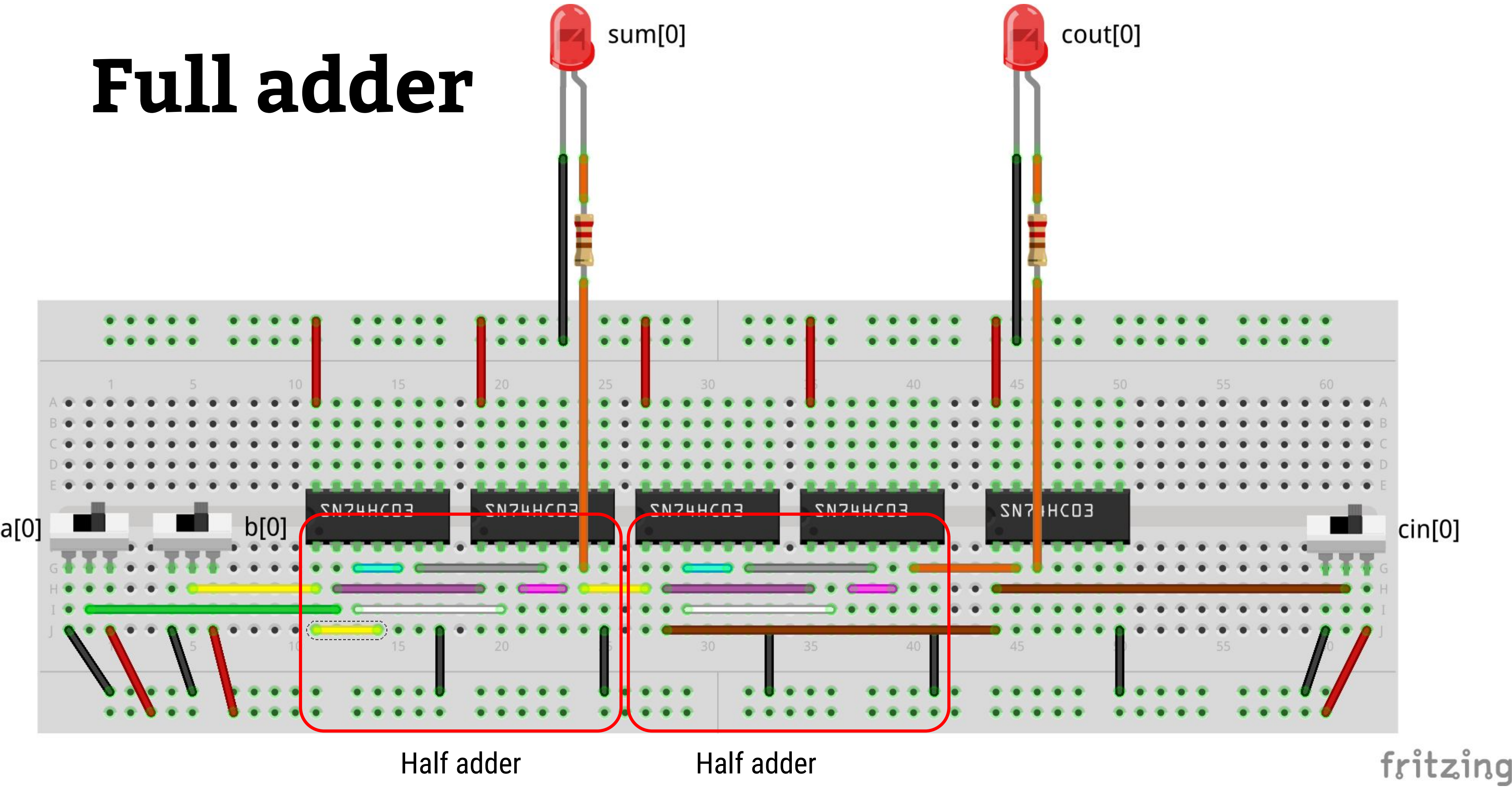
Adder/ALU

Assignment

- Use 7400 chips to implement carry ripple adder
- Get up to 4 points depending on how many FA-s you can wire
- 2 extra points for adding toggleable subtracting



Full adder

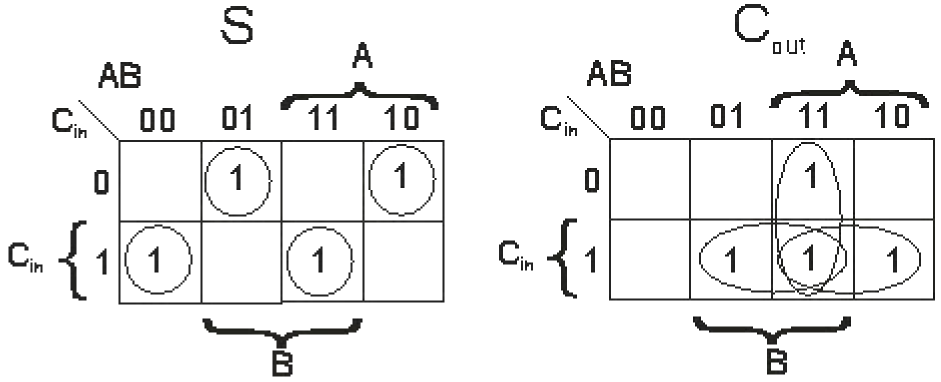

sum[0]
cout[0]
a[0]
b[0]
cin[0]
SN74HC03
SN74HC03
SN74HC03
SN74HC03
SN74HC03
Half adder
Half adder
fritzing

Some tips

- One breadboard (5 chips) should be enough to implement 2-bit adder
- Another board (+4 chips) is required to implement 4-bit adder
- Add another board (+4 chips) to implement subtractor logic
- Use Arduino to run a testbench against the circuit

Karnaugh map for full adder

- Determine inputs
- Determine outputs
- Derive truth table for each output bit
- Use Karnaugh map to derive Boole formula



- Sum = (!A AND !B AND C) OR (!A AND B AND !C) ...
- Cout = (B AND Cin) OR (A AND Cin) OR (A AND B)