# Computer Mathematics

## Week 9
## Parallel logic circuits

KUAS 京都先端科学大学
KYOTO UNIVERSITY of ADVANCED SCIENCE
Department of Mechanical and Electrical System Engineering

# last week

the mathematics of logic circuits

- the foundation of all digital design

Boolean logic

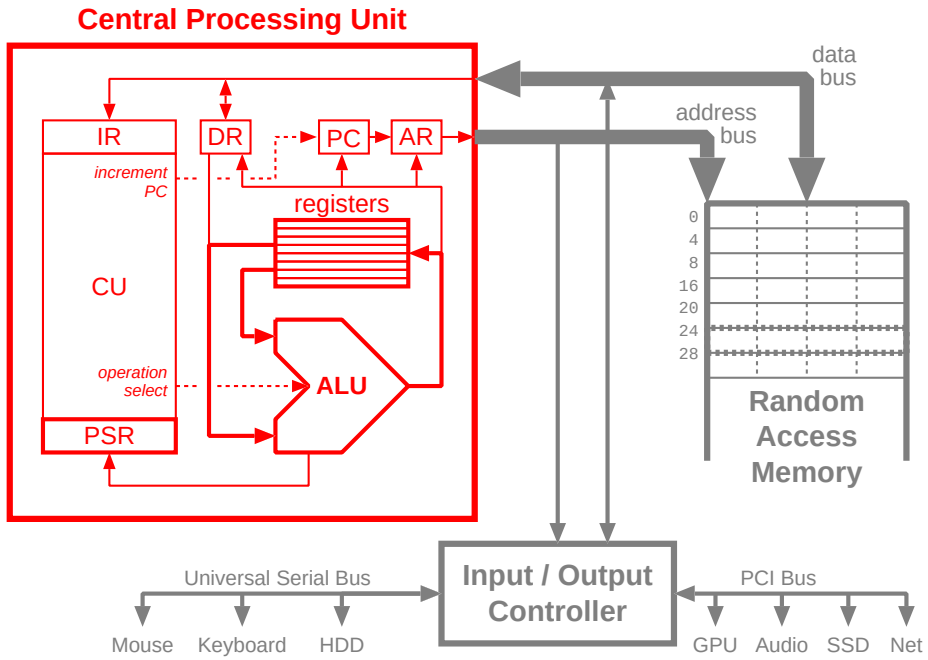
- when `0` and `1` represent true and false

Boolean algebra

- Boolean functions
- canonical forms

simplification of Boolean expressions

- de Morgan's laws



# this week

combinational digital circuits

signals and busses

logic gates

- and, or, not
- nand, nor, xor

gate-level multi-bit logical operations

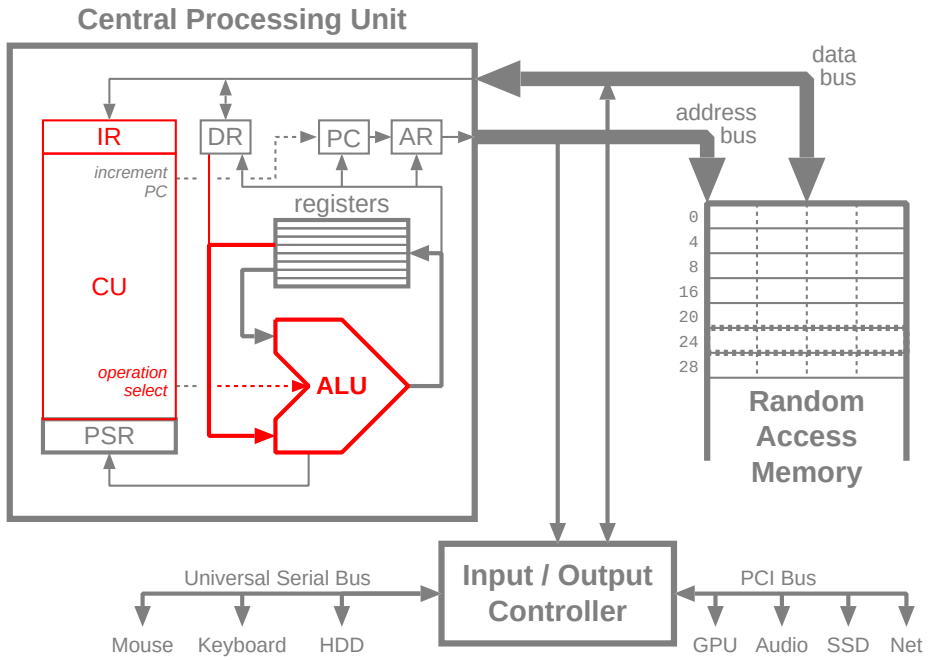
- bitwise: and, or, not

gate-level multi-bit arithmetic operations

- addition, subtraction (unsigned, 2's complement)

1-of-$N$ selection

- multiplexers



# review: addition (single-bit)

| *inputs* | | | *sum* | |
|---|---|---|---|---|
| $c_i$ | $a$ | $b$ | $c_o$ | $s$ |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

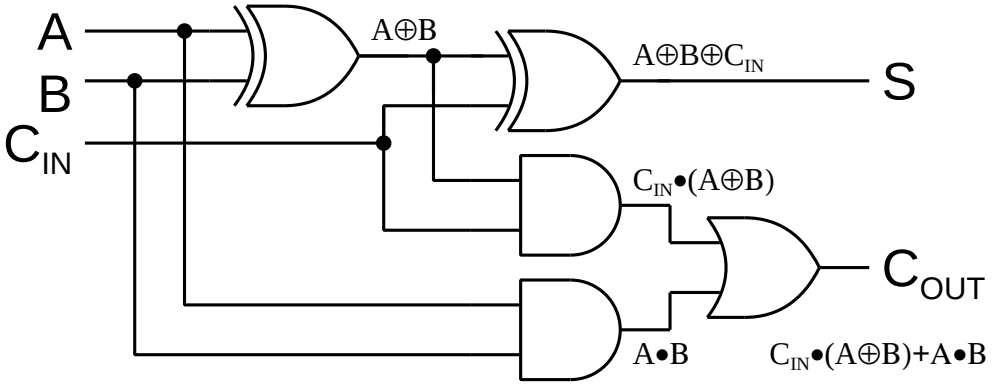
single-bit addition of three inputs

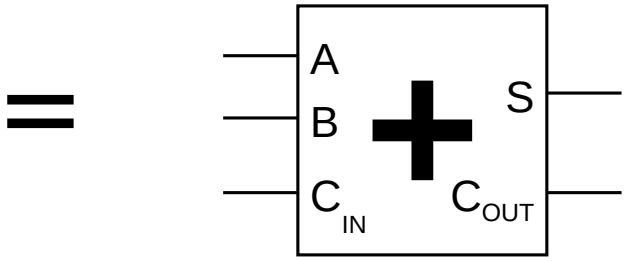
- sum is $1$ if an odd number of inputs are $1$
- carry is $1$ if two or more inputs are $1$

canonical form of each output, simplified, translated into gates

$$\begin{aligned} s &= c_i'ab' + c_i'a'b + c_ia'b' + c_iab \\ &= c_i'(ab' + a'b) + c_i(a'b' + ab) \\ &= c_i'(a \oplus b) + c_i(a \oplus b)' \\ &= c_i \oplus a \oplus b \end{aligned}$$

$$\begin{aligned} c_o &= c_i'ab + c_ia'b + c_iab' + c_iab \\ &= (c_i' + c_i)ab + c_i(a'b + ab') \\ &= ab + c_i(a \oplus b) \end{aligned}$$

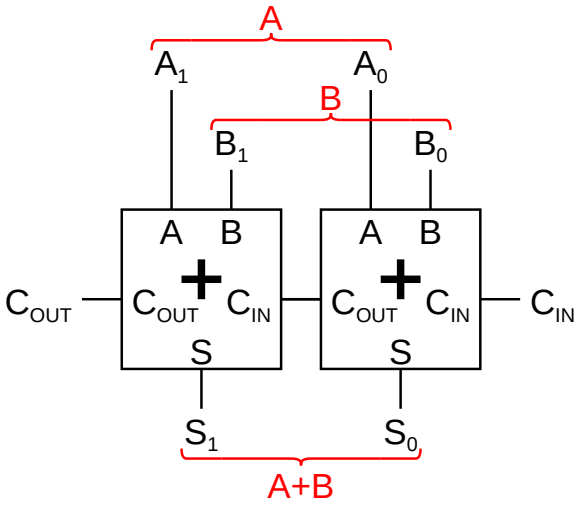


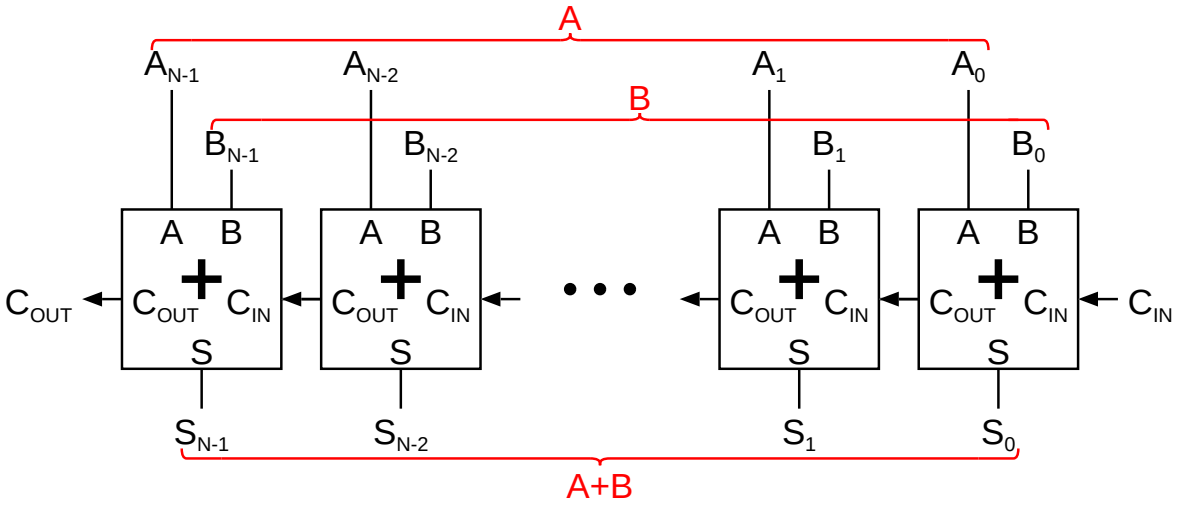

this is a *full adder*

# addition (multi-bit)

two full adders make a two-bit adder




to make a $N$-bit adder, just *daisy chain* $N$ single-bit full adders together




(this is called a *ripple-carry adder* — is there a limit to $N$?)

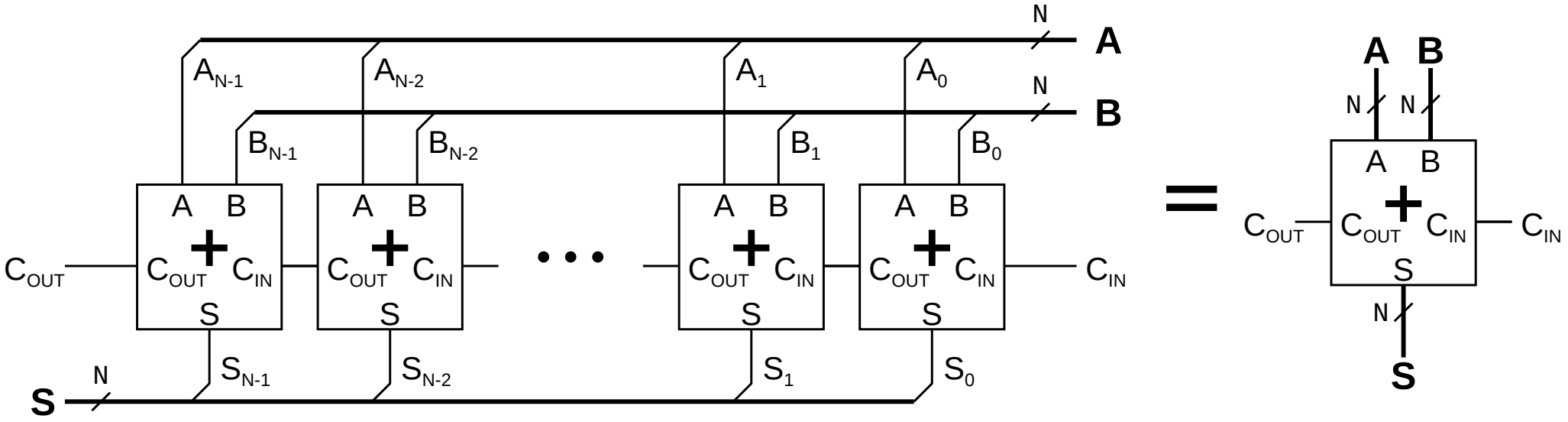
# abstraction — busses

writing the $N$ seperate wires (and adders) for each bit is tedious

- all $N$ wires carrying $A_i$ spend most of their time going to the same places
- similarly for $B$ and $S$

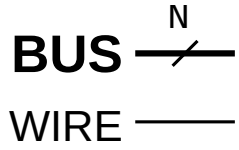
a *bus* carries many individuals in parallel between two points

- collect all the $A_i$ together and put them on a single bus called $A$
- similarly for $B$ and $S$

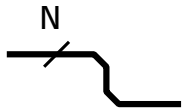


# busses

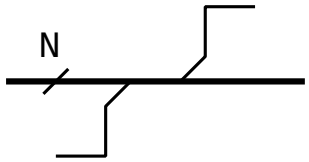
busses are thicker than wires




busses make wide turns

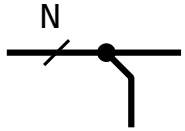



wires joining or leaving a bus do so at an angle




connecting two busses sometimes uses strange symbols to indicate what is happening

- we will just draw a dot, as with wires



# bitwise (multi-bit) logic operations

bitwise logical operations apply a logical operation in parallel to all the bits in a word

e.g., whereas ‘not’ inverts a single logical value

- $0 \rightarrow 1$ and $1 \rightarrow 0$

bitwise ‘not’ inverts all the bits in a word

- it is the ‘one’s *complement*’ operation

$$\begin{array}{rr} \sim & 01101001 \\ \hline & 10010110 \end{array}$$

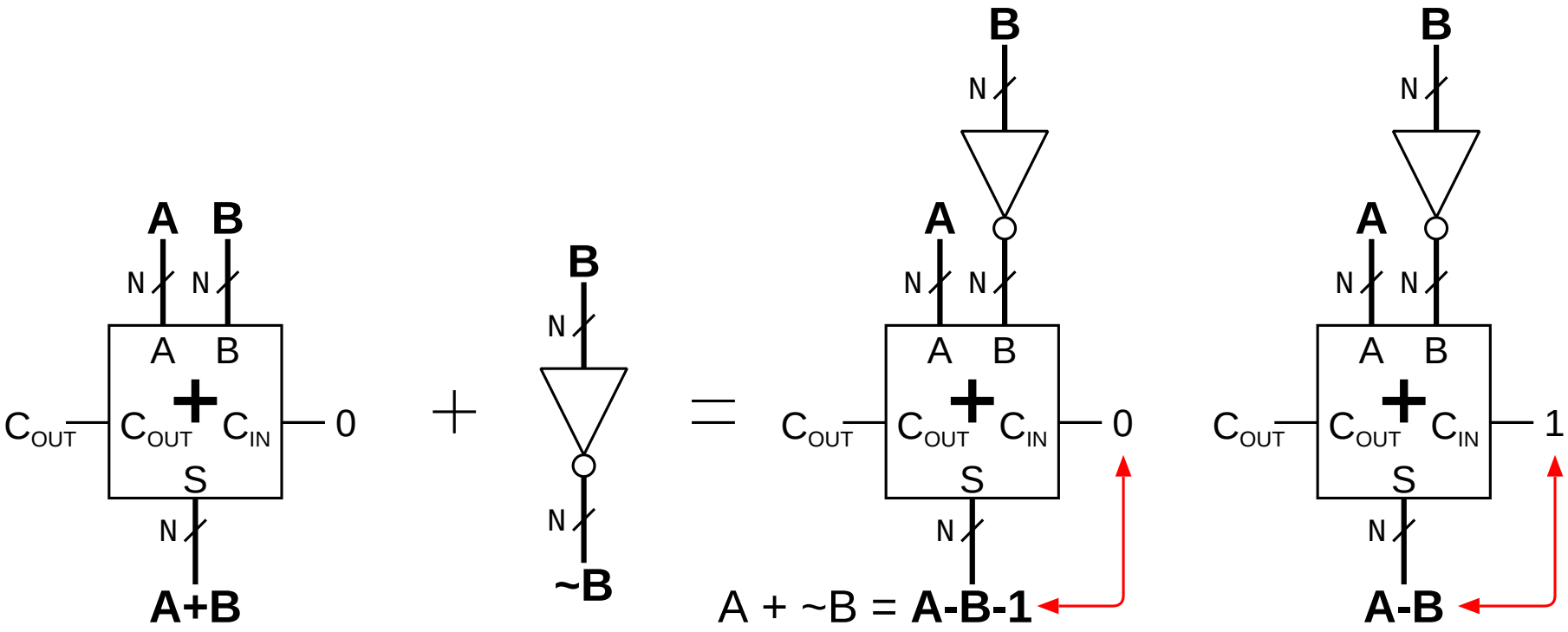
similarly for AND, OR, and XOR

- AND clears (or selects) bits
- OR sets bits
- XOR inverts bits

| AND | OR | XOR |
|---|---|---|
| $\begin{array}{rr} & 11110000 \\ \& & 00110011 \\ \hline & 00110000 \end{array}$ | $\begin{array}{rr} & 11110000 \\ \mid & 00110011 \\ \hline & 11110011 \end{array}$ | $\begin{array}{rr} & 11110000 \\ \oplus & 00110011 \\ \hline & 11000011 \end{array}$ |

# subtraction

combine an adder with a bitwise complement to make a *subtractor*




but for correct results, we need to add the 2's complement of $B$

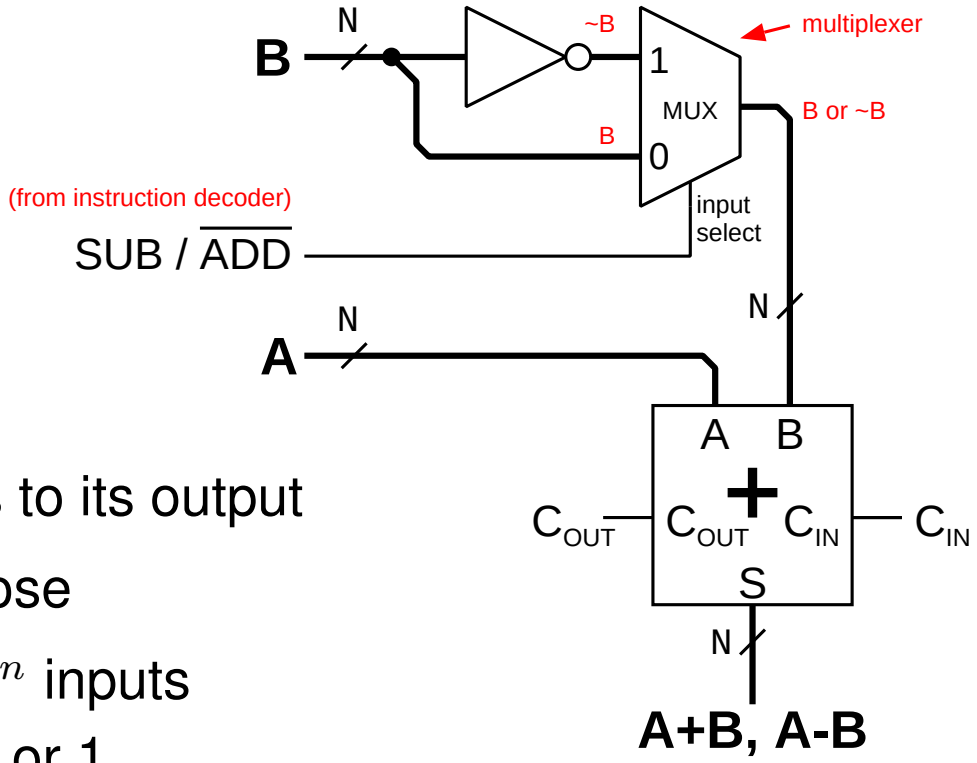
to turn a 1's complement into a 2's complement negation, just add 1

- e.g., by setting $C_{IN} = 1$
- $C_{IN} =$ 'not borrow' when performing subtraction

# selection

are separate adder and subtractor required?

- they are almost identical, except for the complemented $B$ input
- if we could choose between $B$ and $\sim B$, one adder would be enough




a *multiplexer* connects one of several inputs to its output

- a *select* input specifies the input to choose
- $n$-bit select input can choose between $2^n$ inputs
  - 1 select bit chooses between input 0 or 1
  - 2 select bits choose between inputs 0. . . 3, etc.

# multiplexers

2-input multiplexer

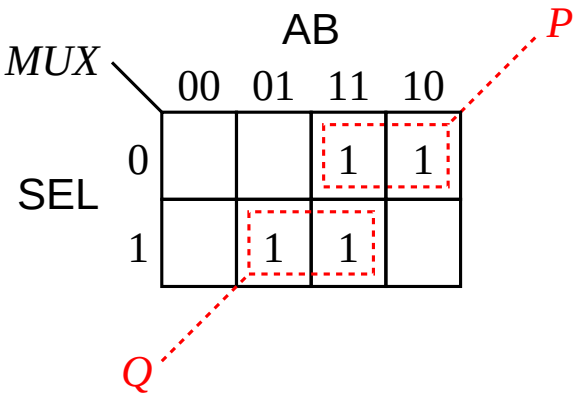
- truth table
- Karnaugh map
- circuit
- functional block

| $SEL$ | $A$ | $B$ | $OUT$ |
|---|---|---|---|
| 0 | $x$ | – | $x$ |
| 1 | – | $y$ | $y$ |

(‘ – ’ = ‘don’t care’)

=

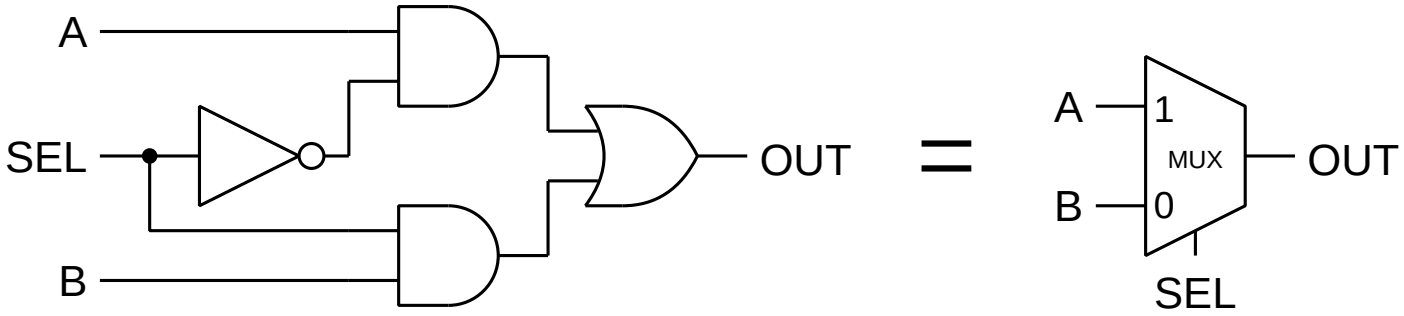
| $SEL$ | $A$ | $B$ | $OUT$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |



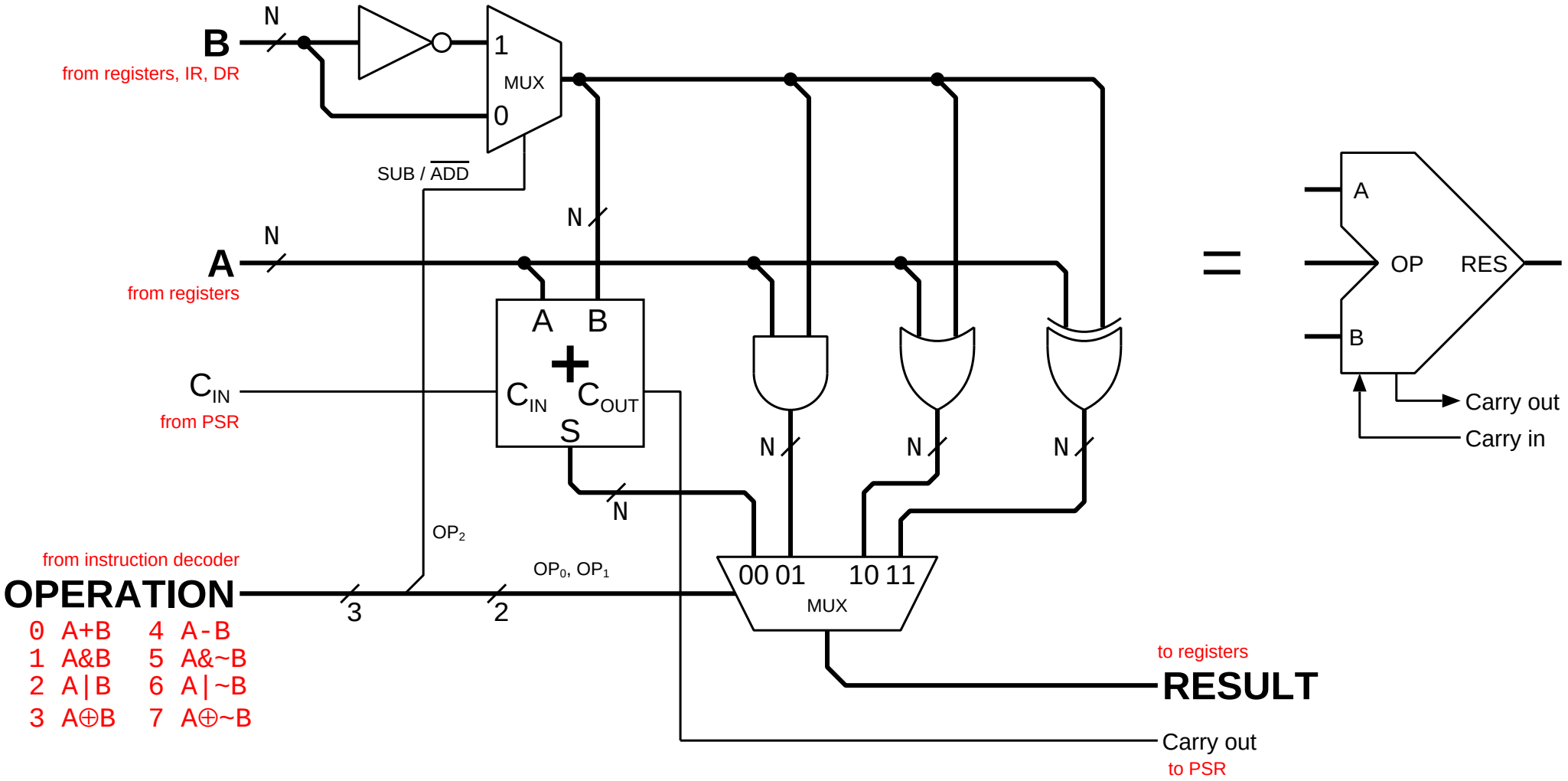

$$\mathrm{P} = \overline{\mathrm{SEL}} \cdot \mathrm{A}$$

$$\mathrm{Q} = \mathrm{SEL} \cdot \mathrm{B}$$

$$\mathrm{OUT} = \overline{\mathrm{SEL}} \cdot \mathrm{A} + \mathrm{SEL} \cdot \mathrm{B}$$



# arithmetic and logic unit



# a typical machine program

search an array to find the index of a particular value, assuming that

- the variable `array` holds the address of the start of the array
- the variable `length` holds the length of the array
- the variable `value` holds the target value that we are searching for
- the index of the element should be returned in register `r0`
- if the target is not found, `-1` should be returned in `r0`

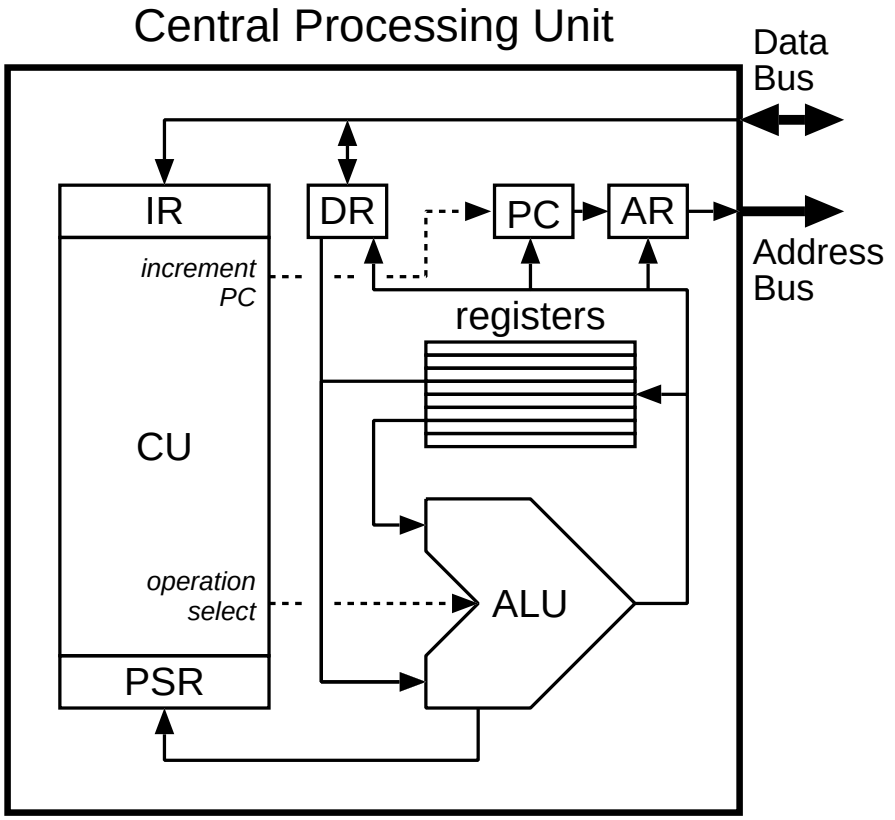
| *label* | *operation* | *operands* | *comment* |
|---|---|---|---|
| | `load` | `r1, array` | load array address into register 1 |
| | `load` | `r2, length` | load array length into register 2 |
| | `load` | `r3, value` | load target value into register 3 |
| | `set` | `r0, 0` | the next index to check is held in `r0` |
| `next:` | `compare` | `r0, r2` | check if we reached the end of the array |
| | `jump_if_equal` | `fail` | if so, target was not found |
| | `load` | `r4, r1[r0]` | fetch next element from array |
| | `compare` | `r4, r3` | compare it to the target value |
| | `jump_if_equal` | `done` | if found, return the corresponding index in `r0` |
| → | **`add`** | **`r0, r0, 1`** | **increment the index** |
| | `jump` | `next` | continue searching at the next index |
| `fail:` | `set` | `r0, -1` | index `-1` means 'target not found' |
| `done:` | `halt` | | `r0` contains index of target element, or `-1` |

# instruction execution

machine ('fetch-execute') cycle:
**repeat**

1. **fetch instruction**
   - copy PC to address bus
   - read from memory into IR
2. **decode instruction**
   - CU inspects the IR
3. **fetch operand(s)**
   - move data from memory into DR, if required
   - present DR and/or registers to ALU
4. **execute instruction**
   - ALU performs operation on data
5. **store result**
   - write ALU output into register or memory
6. **update PC**
   - with the address of the next insutruction

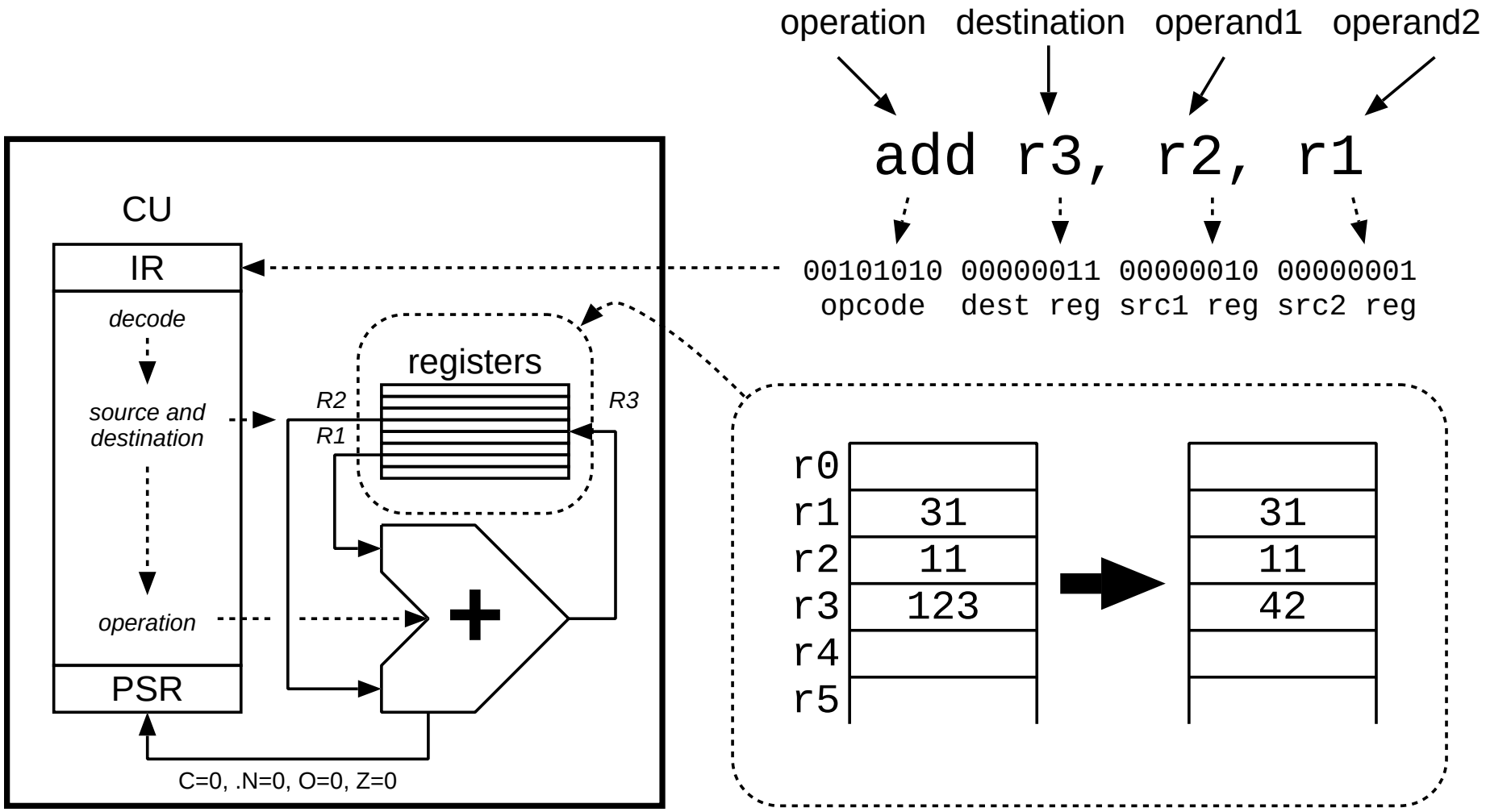
**forever**




DR: data register, holds data read from memory while ALU performs an operation
AR: address register, holds address for memory read/write operation

# a typical machine instruction

add register 1 to register 2, putting the result in register 3

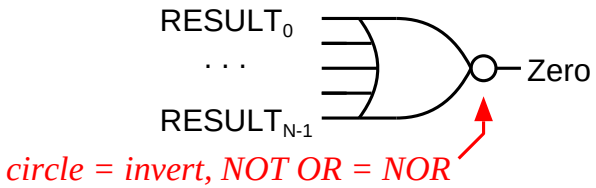


# processor status results

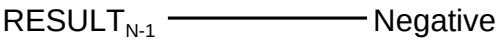
also of interest: '*flags*' for updating the *processor status register*

- was the result zero, or negative?
- did unsigned or signed overflow occur?

the result was *zero* if none of the result bits is 1

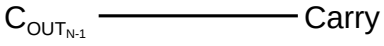



the result was *negative* if the most-significant result bit is 1



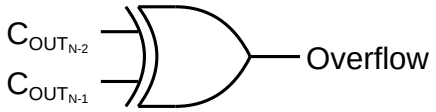

unsigned arithmetic overflow (*carry*) occured in the adder if

- the carry out of the MS bit of the result was 1




signed arithmetic *overflow* occured in the adder if

- the carry in to the MS bit of the result $\neq$ the carry out of it




the condition for signed overflow is maybe a little subtle?

# processor status results — signed overflow

signed arithmetic overflow occured in the adder if

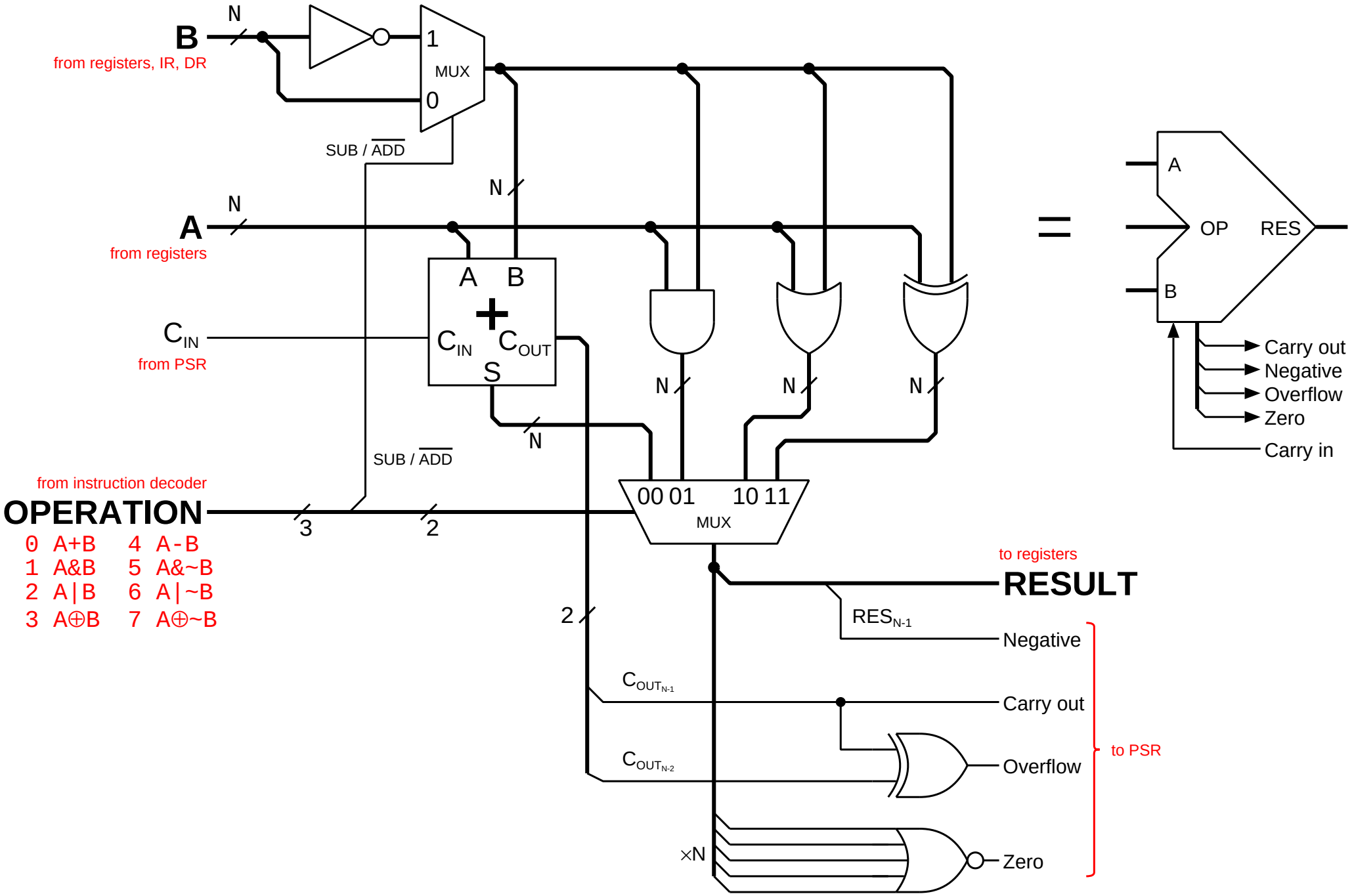
- the carry in to the MS bit of the result $\neq$ the carry out of it

$\Rightarrow$ the sign of the result is wrong (?)

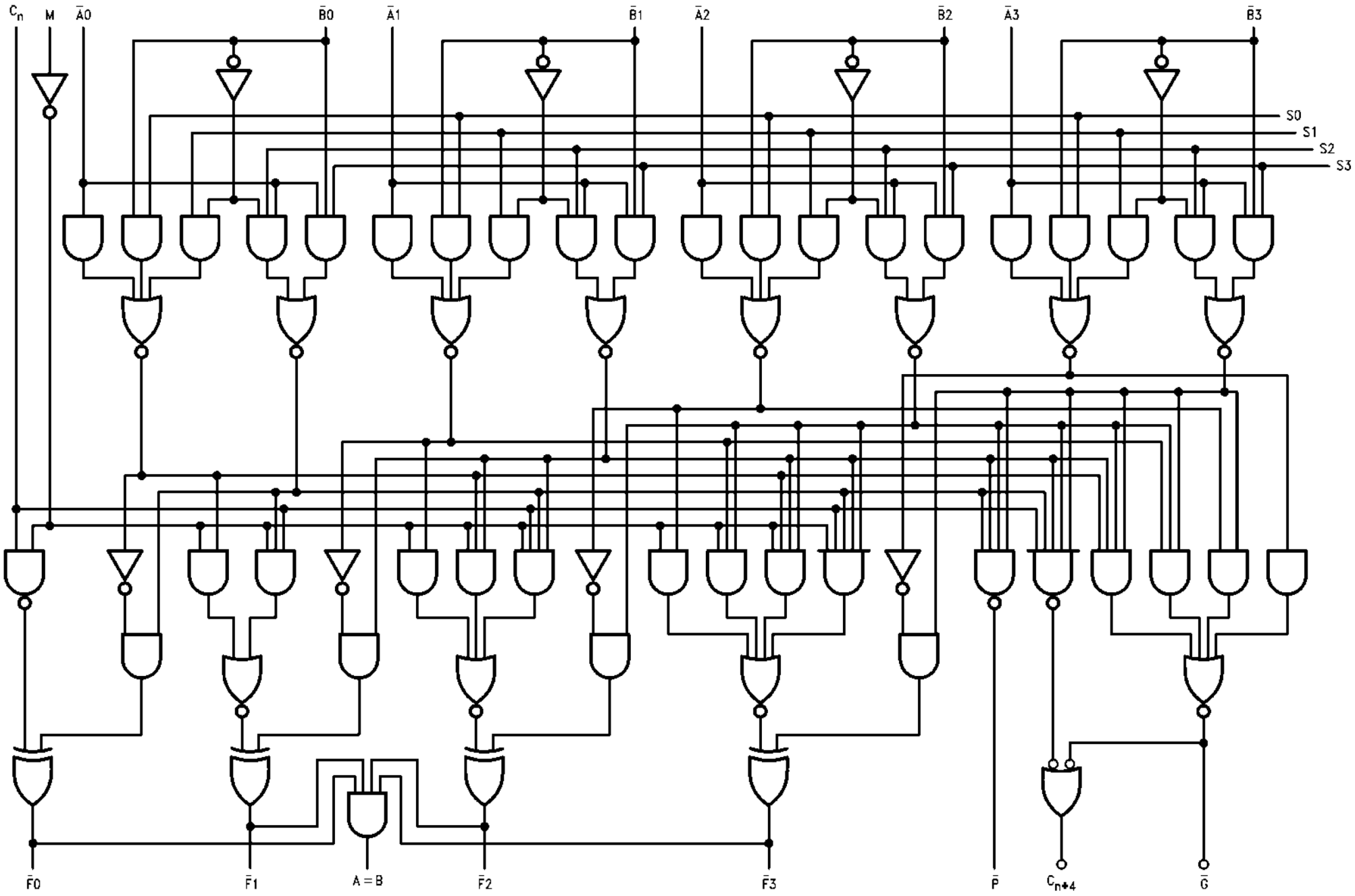
consider a signed, 8-bit, 2's complement addition:

```
   0xxxxxxx    both positive: carry out of MS bit is always 0
 + 0xxxxxxx    if carry in is 1, the sign will change ⇒ overflow
 ---------
0← 0 ← 0 ⟹ ok
0← 1 ← 1 ⟹ overflow

   1xxxxxxx    both negative: carry out of MS bit is always 1
 + 1xxxxxxx    if carry in is 0, the sign will change ⇒ overflow
 ---------
1← 1 ← 1 ⟹ ok
1← 0 ← 0 ⟹ overflow

   0xxxxxxx    if carry in to MS bit is 0, carry out is also 0
 + 1xxxxxxx    if carry in to MS bit is 1, carry out is also 1
 ---------
0← 1 ← 0      (overflow never occurs with numbers of opposite signs)
1← 0 ← 1
```

# ALU with processor status outputs



# optimised 4-bit ALU ‘slice’ with 32 operations



# things we did not consider

shift and rotate instructions

- moving patterns of bits to the left or right, with or without wrap-around
- left as an exercise

multiply instructions

- usually requires synchronous logic (next week)

divide instructions

- always requires synchronous logic (next week)

tri-state logic

- logic that can be 0, 1, or high-impedance
- used for bi-directional signals
- beyond the scope of this introductory course

# next week

sequential digital circuits

stateful logic

- level-triggered devices
- latches

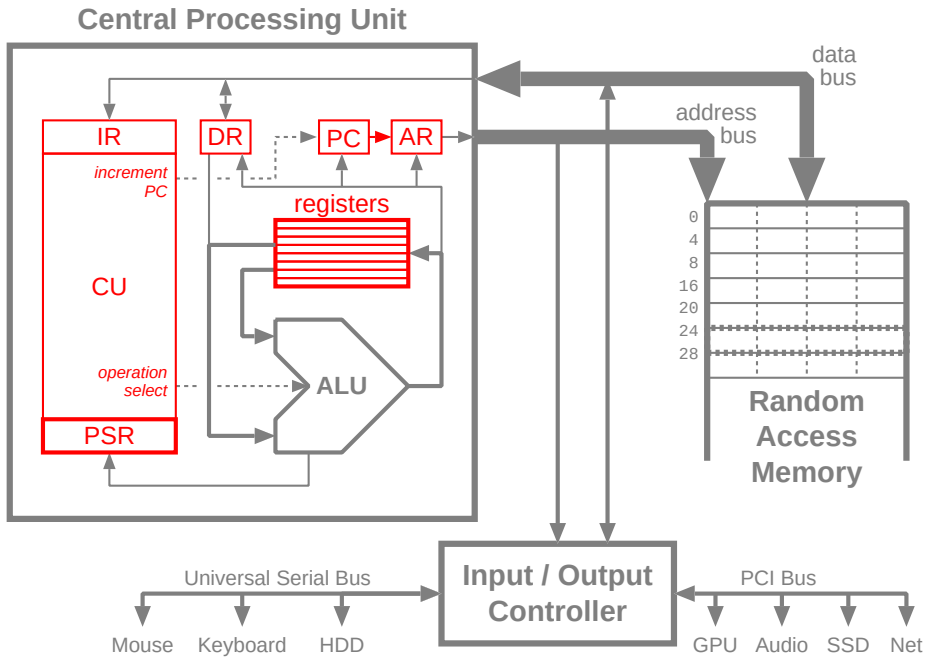
clocks

- edge-triggered devices

synchronous logic

- flip-flops
- registers and memory

CPU operation according to the clock cycle



# homework

**practice** drawing logic circuits for Boolean functions

**consider** some of the gates we did not study in detail

- how many of the logic circuits of this week can you make
  - using only NAND gates?
  - using only NOR gates?

**reinforce your understanding**

- write a Python program to simulate a multi-bit adder
  - consider the logical operations affecting each signal
  - compute the output of each gate based on its input(s)
  - propagate outputs to inputs at every simulation time step

**ask** about anything you do not understand

- from any of the classes so far this semester (or the lecture notes)
- it will be too late for you to try to catch up later!
- I am always happy to explain things differently and practice examples with you

# glossary

**active-low** — a signal that is considered ‘true’ when 0.

**active-high** — a signal that is considered ‘true’ when 1.

**adder** — a logic circuit that implements 2’s complement addition between two words of data.

**bus** — a collection of signals all travelling between the same two points.

**carry** — a processor status bit indicating that the last arithmetic operation generated an unsigned overflow (a carry out of the MS bit).

**complement** — a bitwise logical operation that inverts every bit in a word.

**daisy chain** — connecting two or more devices or circuits in a linear fashion so that some output from one feeds into the input of the next.

**exclusive-or** — an ‘or’ operation that does not allow both inputs to be the same value.

**flag** — a single bit in the processor status register.

**full adder** — an adder that takes three single-bit inputs (two digits and a carry in) and produces two single-bit outputs (a sum and a carry out).

**functional block** — a high-level abstract component in a digitl circuit that represents a reusable pattern of lower-level components or gates.

**gate** — an logic circuit component that implements a fundamental Boolean operation.

**half adder** — an adder that takes two single-digit inputs and produces a sum and carry output.

**multiplexer** — a digital circuit that connectes one of $2^n$ input signals to its output according to a $n$-bit select input.

**negative** — a processor status bit indicating that the result of the last arithmetic operation was negative, assuming the values involved were 2's complement integers.

**overflow** — a processor status bit indicating that the result of the last arithmetic operation generated a signed overflow (the result has the wrong sign), assuming the values involved were 2's complement integers.

**processor status register** — a register in the CPU holding several flags indicating whether the last arithmetic operation produced a zero or negative result, or whether an unsigned or signed overflow occured.

**ripple-carry adder** — an adder constructed by daisy chaining several single-bit full adders together. The name arises because the sum and carry out at each bit position $n$ cannot be computed until the carry out at position $n - 1$ has been computed.

**select** — an input to a multiplexer that selects which of several other inputs should be connected to the output.

**signal** — anything that conveys a logic value from one place to another. In logic circuits, a signal is carried by a wire.

**subtractor** — a logic circuit that implements 2's complement subtraction between two words of data.

**wire** — a connection between several points in a circuit that forces them to all have the same logical value.

**zero** — a processor status bit indicating that the result of the last arithmetic operation generated a zero result (none of the result bits was 1).