High Performance Arithmetic

John Kerl
Lockheed Martin Management & Data Systems
Intelligence, Surveillance, and Reconnaissance Systems
Litchfield Park, Arizona
October 18, 2004

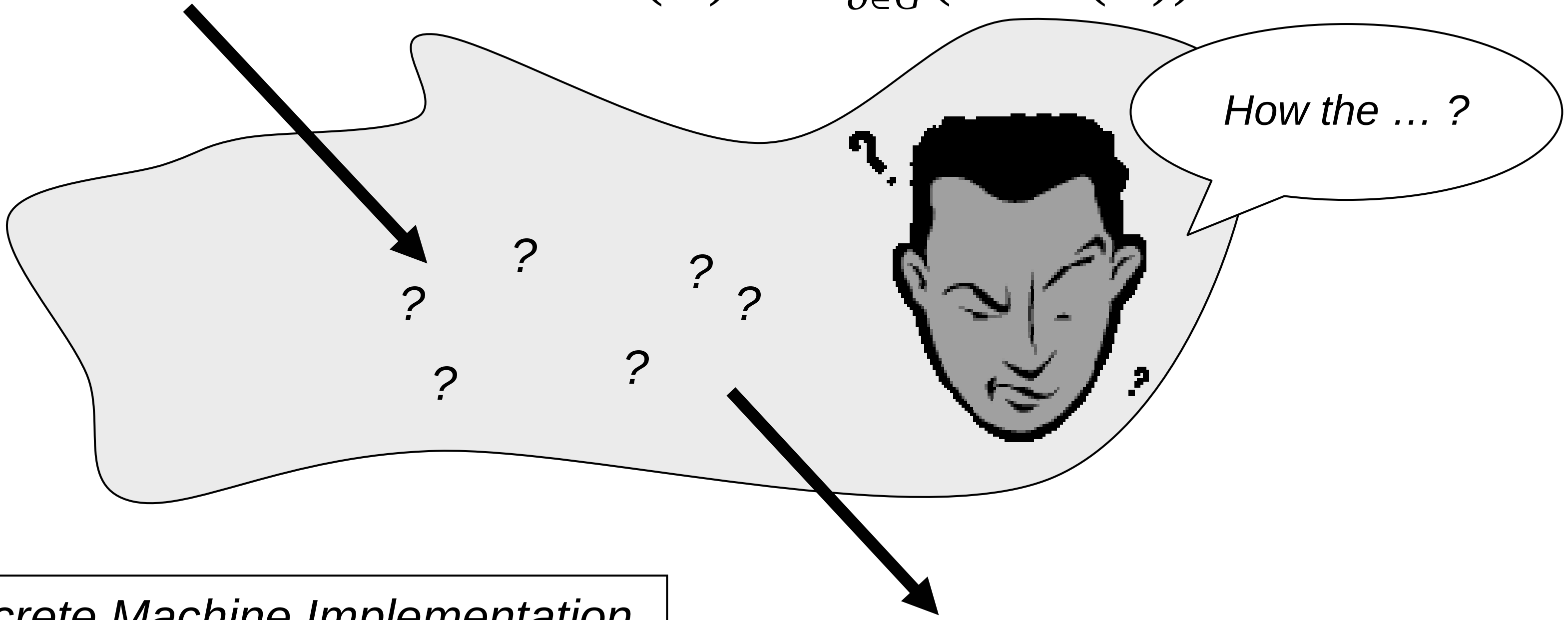
`john dot r dot kerl at lmco dot com`
`kerl at mathpost dot asu dot edu`

Volumes of data require automation

$$F(k) = \int_0^1 e^{-i2\pi kx} f(x)dx$$

Abstract Human Design

$$C(\alpha) = \Pi_{\sigma \in G}(x - \sigma(\alpha))$$



Concrete Machine Implementation

```
0101 0101 1000 1001 1110 0101 1000 1011 0101 0101 0001 0000 0011 0001 1100 0000
0011 1001 1101 0000 0101 0011 1000 1011 0100 1101 0000 1100 1000 1011 0101 1101
0000 1000 1101 1001 1110 1110 0111 1101 0001 1000 1000 1101 0111 0100 0010 0110
0000 0000 1000 1101 1011 1100 0010 0111 1101 1001 0000 0100 1000 0001 1101 1000
```

Isn’t the rest merely implementation details?

- Recent talks in this series have presented some high-level designs for compute-intensive problems
- Implementation details are where engineers spend much of their time, hence much of the company’s resources
- It is important that high-level designers be aware of low-level constraints, and that low-level implementers be aware of the big picture
- Implementation constraints affect design

General-purpose tools don’t always suffice

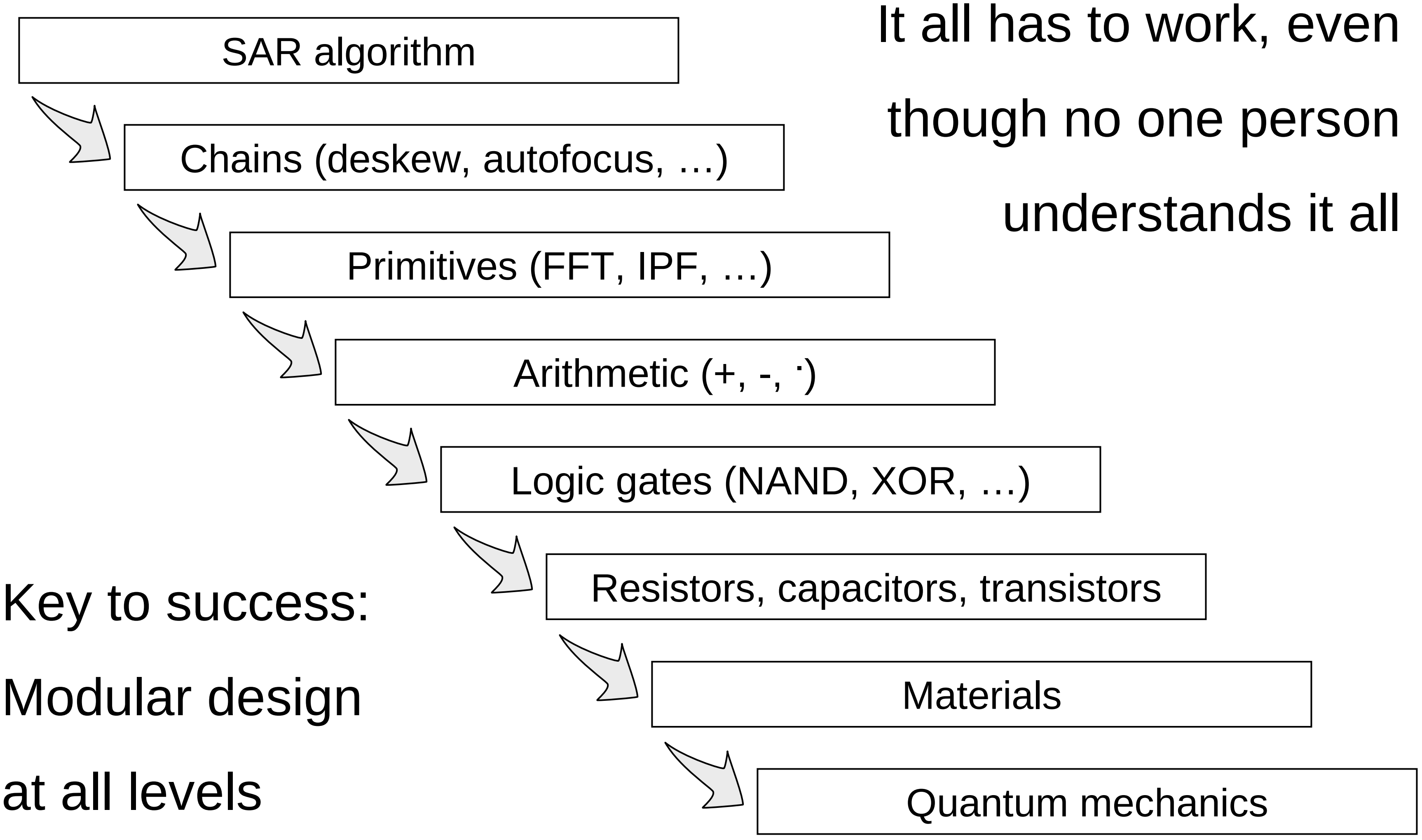
- Computer algebra systems such as MATLAB, Mathematica, etc., provide abstract-looking syntax
- Excellent for prototyping, but don’t provide adequate performance for demanding applications.
- We have competitors, and so do our customers. Everyone wants to process more data, in less time, at more MIPS per watt.
- We use common off-the-shelf (COTS) technology when appropriate
- When standard parts aren’t fast enough, we build our own
- We do what we know, partner for what we don’t
- We re-use past efforts (and design for re-use) to reduce risk and cost

Hardware acceleration is everywhere

HW/SW choices presented here don’t just apply to SAR/DSP:

- Other DSP applications
- Adaptive control
- Telecommunications
- Cryptography: Large-modular (RSA), finite-field (AES), elliptic curves
- Error-control coding
- Anywhere real-time computation is needed

Hierarchy of detail


SAR algorithm
Chains (deskew, autofocus, …)
Primitives (FFT, IPF, …)
Arithmetic (+, -, ·)
Logic gates (NAND, XOR, …)
Resistors, capacitors, transistors
Materials
Quantum mechanics

It all has to work, even though no one person understands it all

Key to success:

Modular design at all levels

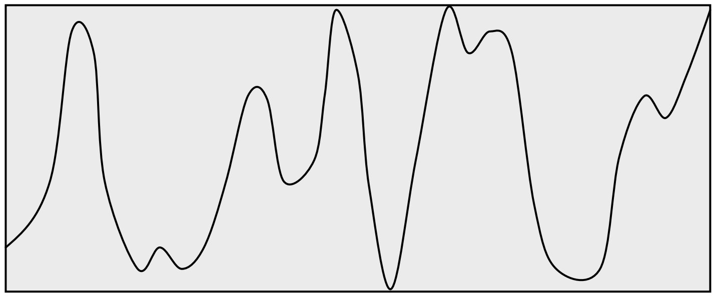
Disciplines

- Systems engineering
- Software engineering
- Electrical engineering
- (Mechanical engineering*)
- (Chemical engineering)
- (Materials-science engineering*)
- Program management: The difference between a good job and a great job; the difference between an also-ran and a winning organization

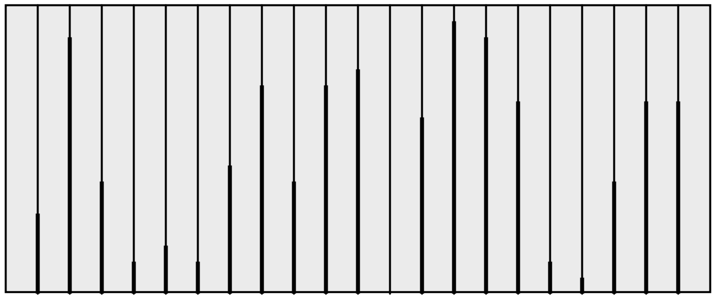
Useful skills for success in industry

- Interdisciplinary education
- Writing and speaking skills are always needed
- Programming skills are vital for almost any technical job. You must learn at least one of C, FORTRAN, MATLAB, Perl, etc.
- Can you perform some basic computational tasks, both on paper and using automation: numerical estimation of a derivative, integration using Simpson's rule, Lagrange interpolation, Taylor-series approximation, making plots, etc.? If not, learn how.
- Undergraduate numerical analysis and computer arithmetic
- Digital design: CSE 330, various EEE courses

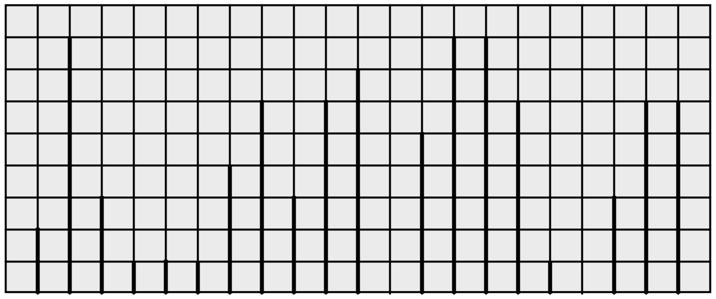
Discretization



Continuous analog waveform …



… sampled in discrete time …



… with discrete amplitudes

Fundamental arithmetic operations for DSP

- Addition, subtraction and multiplication
- Division not so much. Multiply by reciprocals of constants when necessary.
- A common operation is multiply and accumulate (MAC): sum of products
- Number formats: signed or unsigned fixed-point (integers are just a special case); floating point.
- Today we'll discuss addition of unsigned integers.
- In digital logic, high voltage (5.0V, 3.3V, 1.8V, …) represents a one
- Low voltage (0V) represents a zero
- Arithmetic is done in binary (base 2)

Integers and integer addition

- Binary integers: base 2, not 10. E.g. 01011 = 8 + 2 + 1 = 11
- *N* bits: MSB is 2^{N-1}, LSB is $2^0 = 1$

```
  5            0101
+ 3          + 0011
------       -----------
  8            1000
```

- Addition is just like in elementary school
- “1 + 1 is 0, carry the 1 … ”
- Column sums
- Carry-in, carry-out

Digital logic gates

We take these as our starting point (lowest level in the design hierarchy)

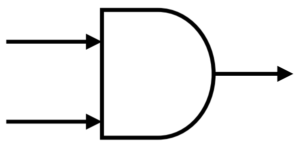
Name

AND:

Truth table

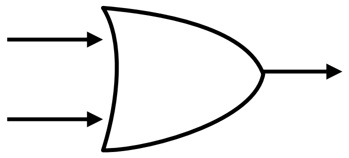
	0	1
0	0	0
1	0	1

Schematic symbol



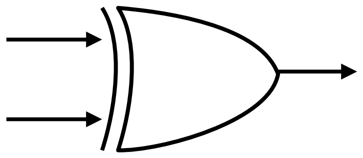
OR:

	0	1
0	0	1
1	1	1



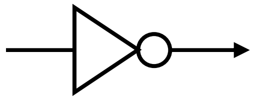
XOR:

	0	1
0	0	1
1	1	0

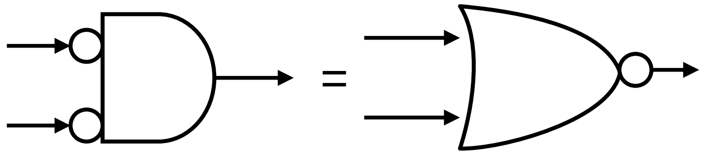


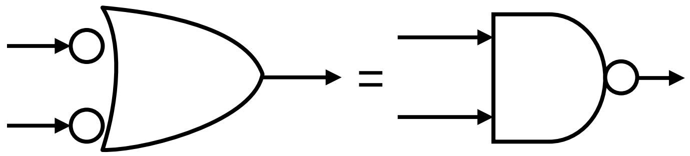
NOT:

0	1
1	0



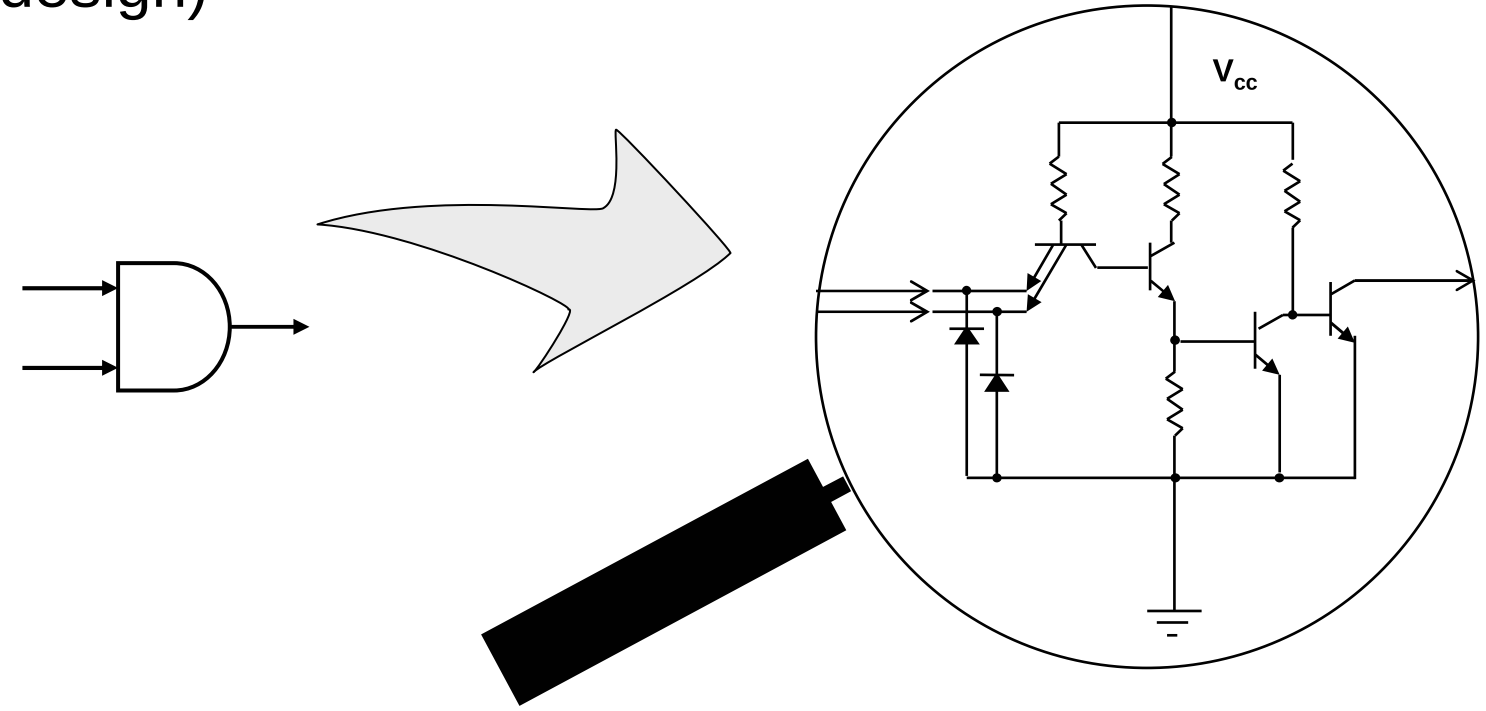
DeMorgan’s Laws:





Digital logic gates (cont'd)

- Each of these is composed of resistors, capacitors, diodes, transistors and wires, each of which is built to have a simple mathematical model
- Put it in a box and label it with a schematic symbol (modular design)



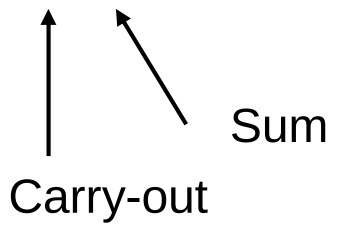
Digital logic gates (cont'd)

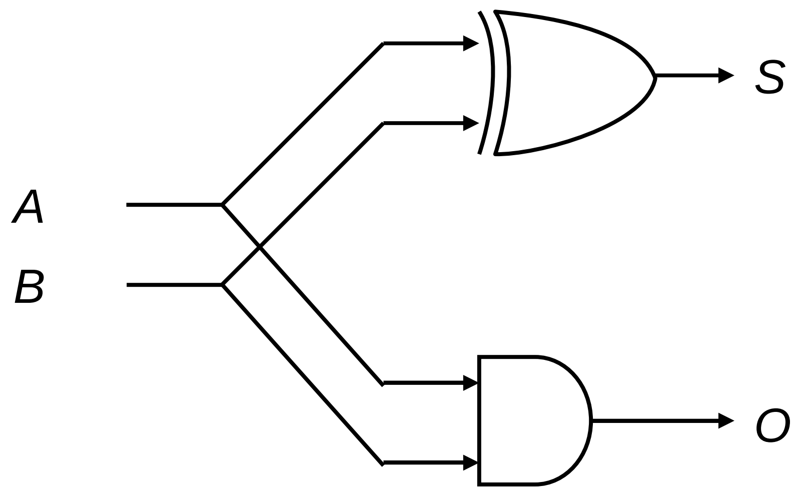
- Conductors have overlapping outer bands; outer electrons are free to flow
- Electron charges are quantized, but (at fabrication scales in use today!) we can still model them as a fluid
- Current flows, but in digital logic we think of *voltage* as carrying information
- Power-plane voltage is high (1); ground-plane voltage is low (0)
- A NOT gate drives out a low voltage when input voltage is high, and vice versa. Similarly for the other gates.

Integer addition using logic circuits

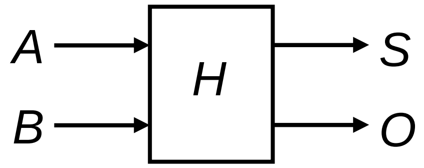
- 1-bit half adder:

```
  0        0        1        1
+ 0      + 1      + 0      + 1
------   ------   ------   ------
0 0      0 1      0 1      1 0
```





Hide the details in a box:

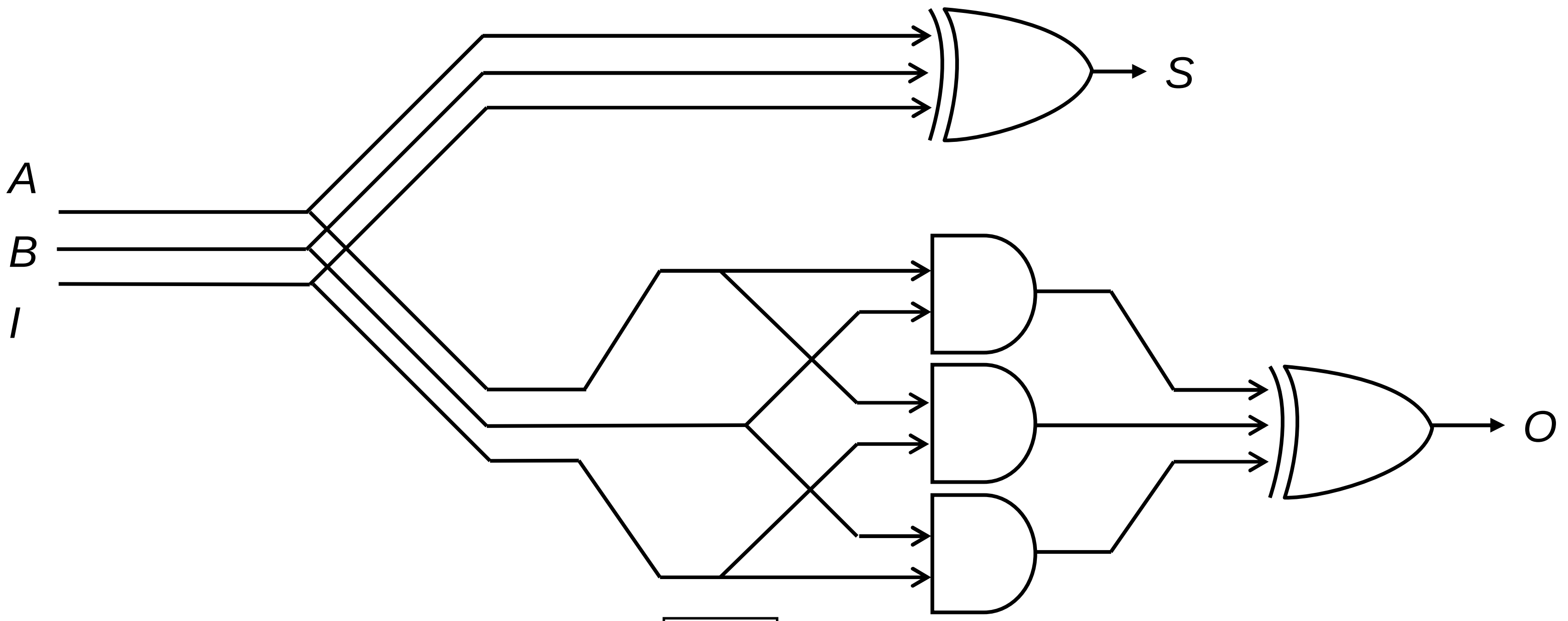


Notice:

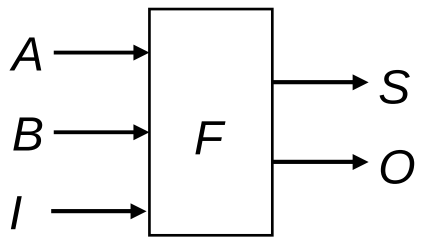
- Column sum is XOR of inputs (sum mod 2)
- Carry-out is 1 if both inputs are 1 (AND)

Integer addition using logic circuits (cont'd)

- 1-bit full adder:
- *A* + *B* + carry-in gives column sum and carry-out

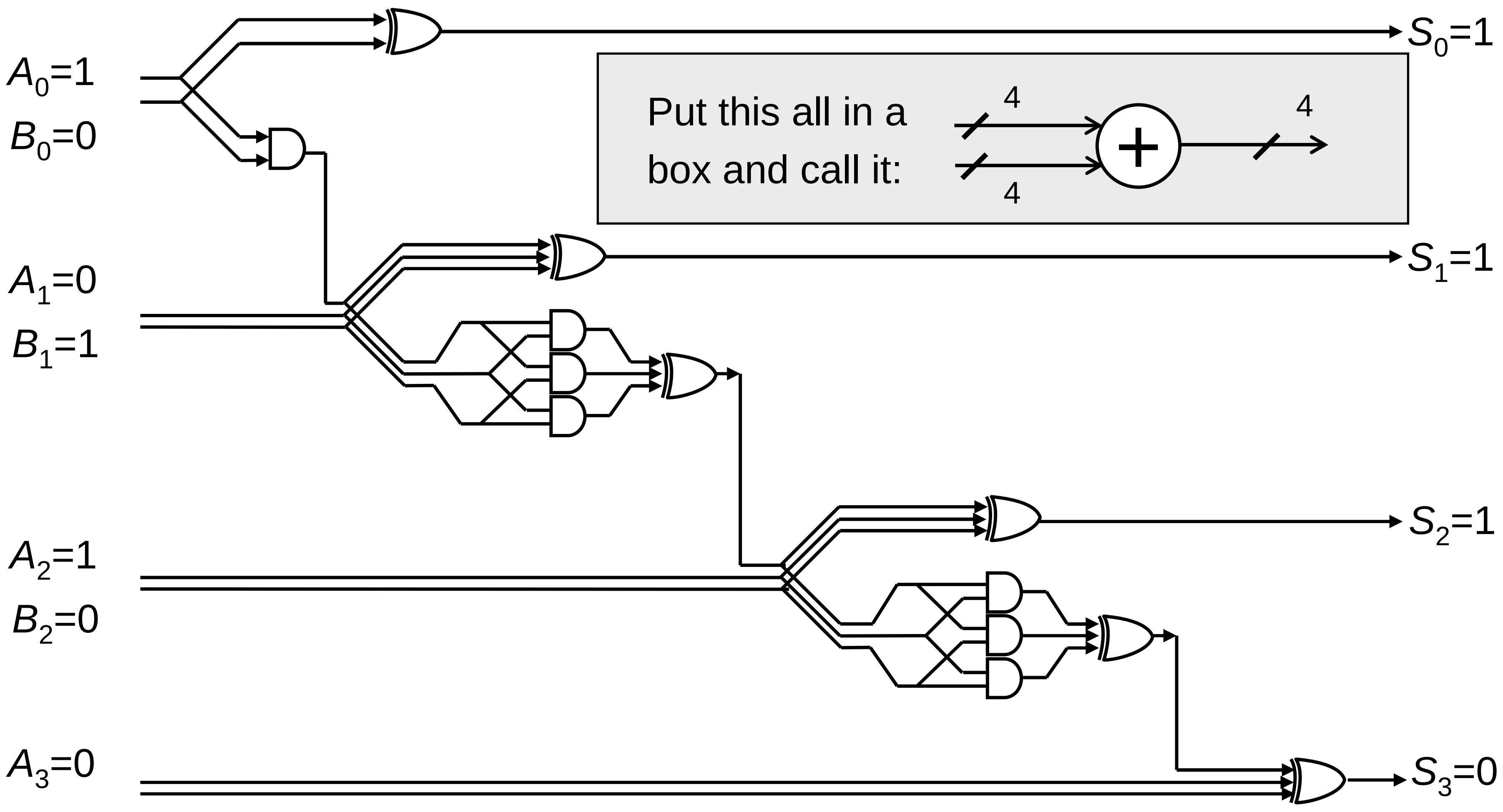


Hide the details in a box:



N-bit full adder (4-bit example)

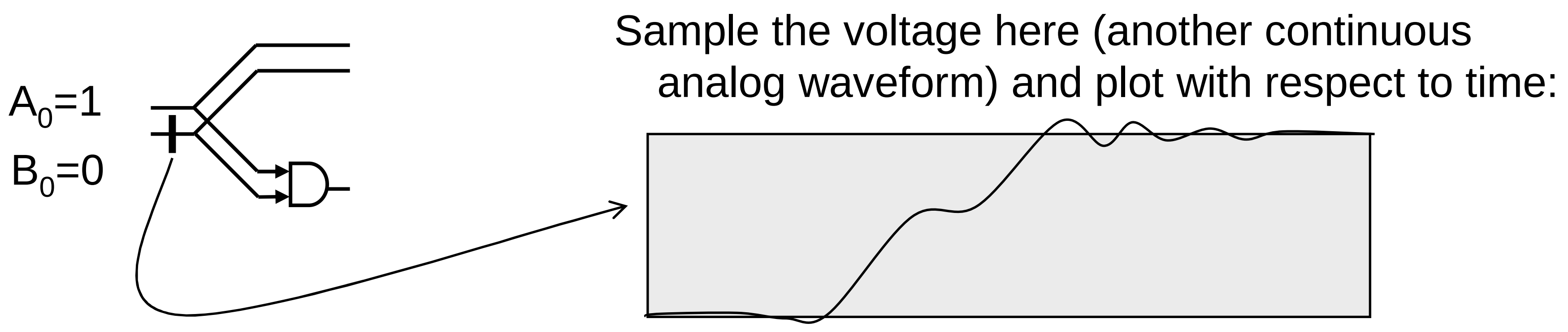
0101 + 0010 = 0111 i.e. 5 + 2 = 7 (1's here are marked in red)



(Remember: this is nothing more than the elementary-school algorithm.)

Timing (the heart of digital design)

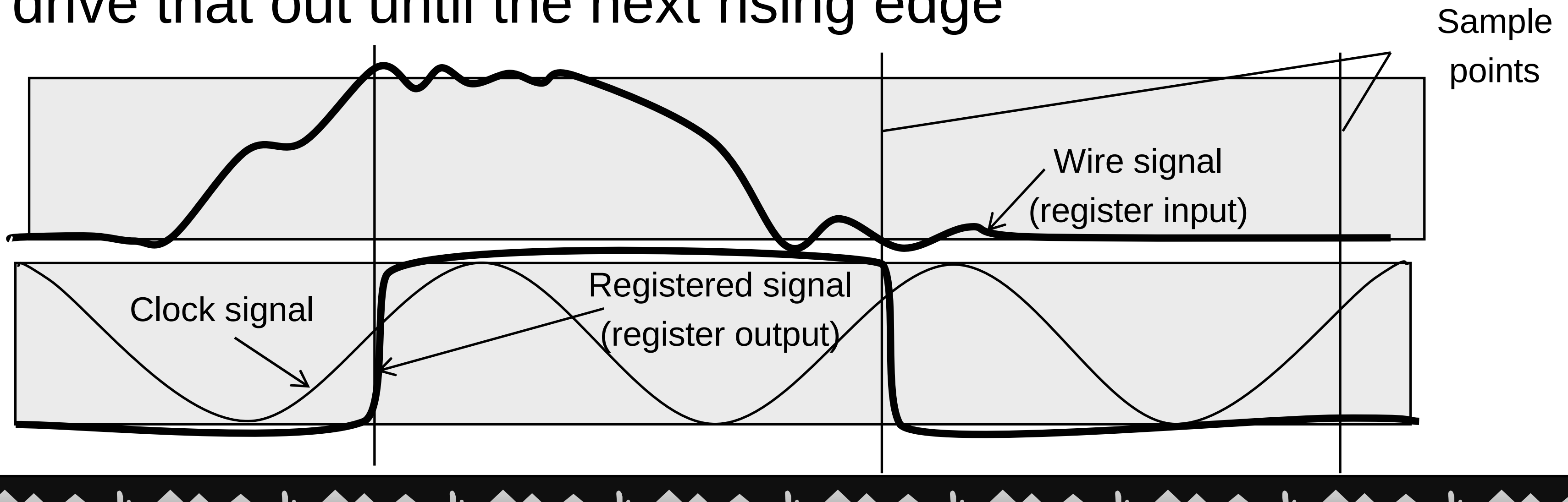
- Everything up to now was static
- Now let bit B_0 change from 0 (low voltage) to 1 (high voltage)
- The low-to-high wave front has its own rise time:


A_0=1
B_0=0
Sample the voltage here (another continuous analog waveform) and plot with respect to time:

- Furthermore, it takes some *propagation time* for the wave fronts to travel from the B_0 input to the S_0-S_3 outputs (all of which change in this example), then stabilize (remember forced damped oscillator from ODEs?) to their new values
- Values during that time are not mathematically correct

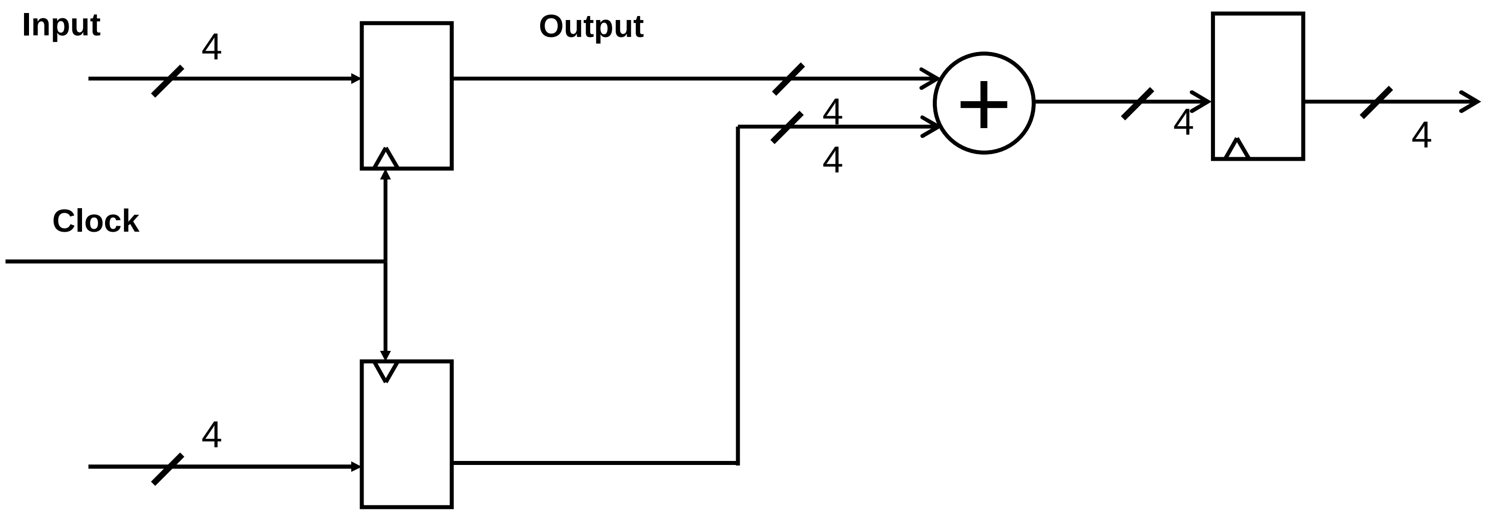
Clocking

- Just as with the signal under analysis (for which these circuits are built), we sample the voltages at discrete times, with discrete amplitudes (but only two levels here: high and low)
- There is an oscillating (sinusoidal or square) signal called the c*lock* fed throughout the chip. Clock frequency in MHz or GHz.
- Electronic devices (made of logic gates) called *registers* retain whatever value is present at, say, the rising clock edge, and drive that out until the next rising edge


Sample points
Wire signal (register input)
Clock signal
Registered signal (register output)

Registers

- The amount of *combinational logic* between registers determines the *pipeline depth*.
- Maximum depth constrains clock speed, or vice versa.
- In order to meet timing, sometimes logic must be split across registers, decreasing depth but increasing latency (e.g. 1 clock for an add, 3 for a multiply).



Registers and wires

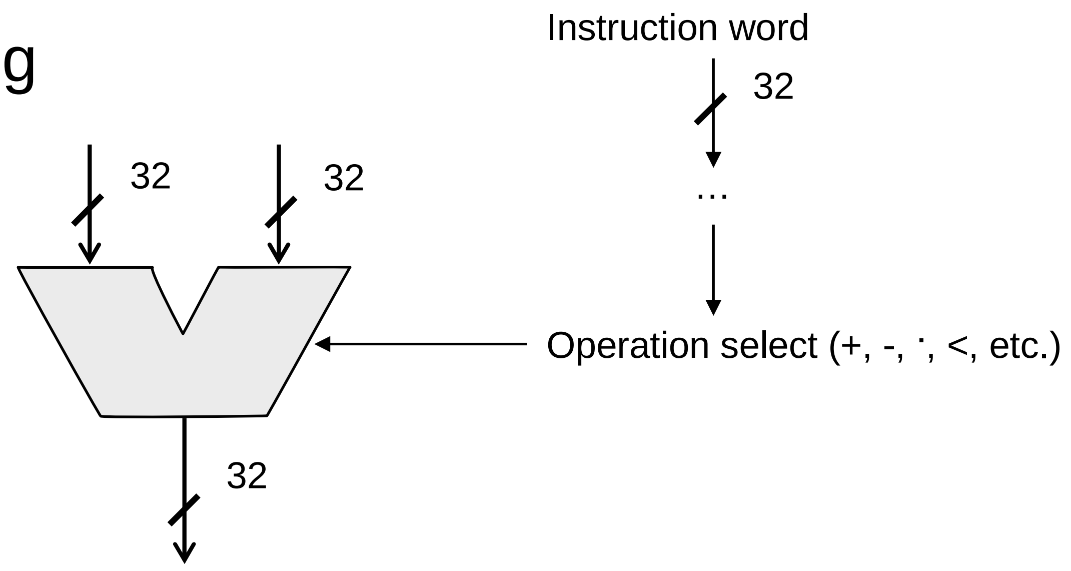
- To a first approximation, digital logic consists of:
  - The clock (distributed throughout a chip)
  - Registers, where voltages can change only at e.g. rising clock edge
  - Wires (“combinational logic”), where voltages can change at any time
- The clock signal must be clean (no spurious edges)
- Register inputs must not be near half-value at sampling time
- The deepest logic in the circuit limits the clock speed
- Clock frequency can’t be too high (and/or logic too deep), else wire signals will be sampled before they are stabilized to their new values
- This is why engineers have to work so hard to increase clock frequency

Faster, faster, faster

- Increase the clock frequency, i.e. shorten the clock period
- Requires shortening path length
- Requires finer fabrication techniques (130 nm, 90 nm, …)
- Keeps electrical and materials-science engineers employed

Sequential processors

- Machine instructions are just integers stored in memory
- Stored-program concept: instructions are data
- Various bits in an instruction word specify arithmetic and/or I/O operations
- Arithmetic and logic unit (ALU) has various arithmetic blocks
- Only one result is done at a time
- Sequential processing



Sequential processors (cont’d)

- Everyone knows about Pentiums
- Embedded processors: PowerPC, ARM, etc.
- Programmable via an instruction set
- Higher-level languages (C/C++, FORTRAN, MATLAB, ...), largely portable
- Compilers are highly non-trivial (keeping computer scientists employed)
- Many MB (GB?) of RAM, plus GB of disk, permit quite large instruction space, stack space, deep recursion, many function arguments, etc.
- The programmer has a lot of freedom

Sequential processors (cont'd)

- Hardware design is fixed
- Mercifully, you don't need to muck with the hardware in order to write programs
- Intel et al. invest time and resources into making a reliable, functionally correct processor
- Customers don't need to be convinced that such chips function correctly
- Approximately one instruction per clock cycle
- *Key point: Quicker to write, slower to run*

Custom parallel processing

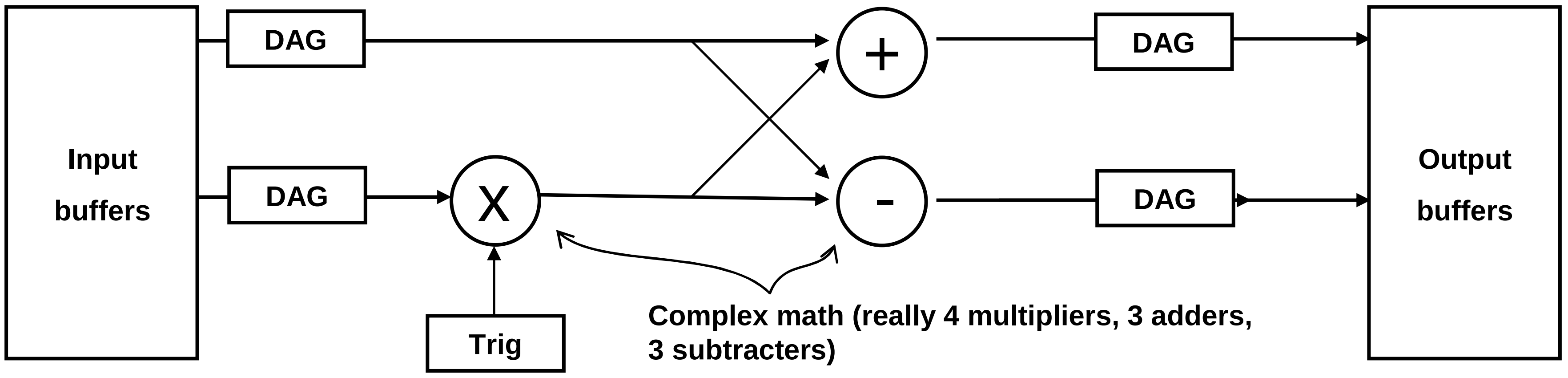
- We want to do more than one thing at a time
- The hardware design is our own, so we can do what we want
- This takes time and resources to implement
- VHDL/Verilog are fundamentally different from C/FORTRAN
- But we don’t want to make everything custom:
- CPUs are highly non-trivial
- Expense of design and verification
- Customer might doubt the result will be bug-free (“risk reduction”)
- Focus on our core competencies
- CPUs are still nice for setup and control
- *Key point: Slower to write, quicker to run.*

Custom parallel processing (cont'd)

- Find those steps in the algorithm most in need of acceleration, and most amenable to it. Create custom circuitry for those things only: *hardware-software co-design.*
- How much programmability should we implement?
  - At least, vector lengths and coefficients
  - Microcode?
  - Simple instruction set?
  - Include a third-party CPU core (e.g. ARM)?

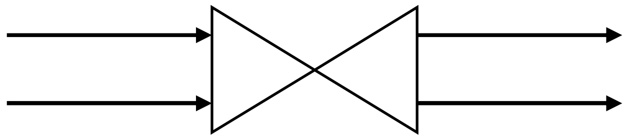
Custom parallel processing (cont'd)

- Signal processing primitive: FFT radix-2 butterfly. $A \pm w \cdot B$, with A, B, and w complex numbers, w on the unit circle ($e^{i2\pi k/N}$)
- $e^{i2\pi k/N}$ might be computed/interpolated using custom circuitry
- Depending on the amount of parallelism, maybe several output samples per clock
- Logic depth and clock determine number of registers (latency)
- The result can far outperform a comparably clocked sequential processor

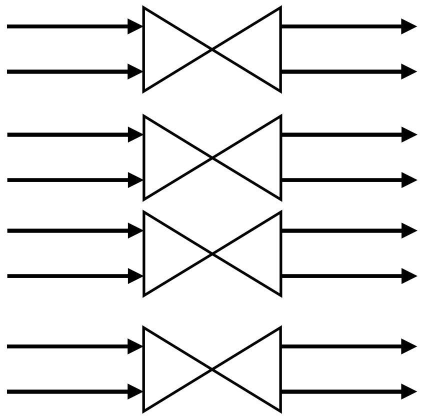

Input buffers
DAG
DAG
X
Trig
+
-
DAG
DAG
Output buffers
Complex math (really 4 multipliers, 3 adders, 3 subtracters)

Custom parallel processing (cont’d)

- Put this in a box and call it:



- Throughput is two output samples per clock, with 4 multiplies, 3 adds and 3 subtracts.
- But it requires that input data can be provided at 2 samples per clock.
- These can be stacked up to increase throughput even more (parallelism):
- Implementing these arithmetic circuits requires more space on the chip
- Keeping these arithmetic circuits busy requires that I/O be done at the same rate



References

- Feynman, *Feynman Lectures on Computation*
- Hennessey and Patterson, *Computer Organization and Design*
- Horowitz and Hill, *The Art of Electronics*
- Knuth, *The Art of Computer Programming: Seminumerical Algorithms (vol. 2)*
- Press et al., *Numerical Recipes*

Thanks for attending!