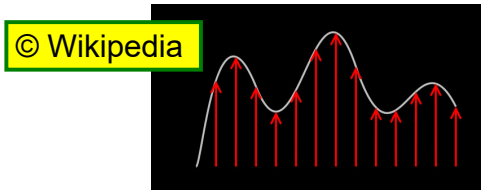
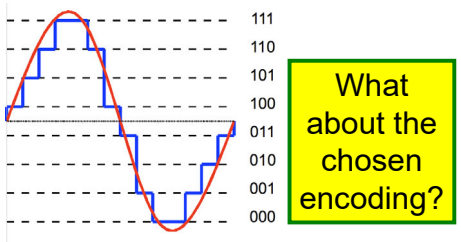
OUTLINE

- DSP applications
- DSP platforms
- The synthesis problem
- Models of computation

DIGITAL VS. ANALOG SIGNAL PROCESSING

- *Digital signal processing* (DSP) characterized by:
  - *Time-discrete* representation of signals: signals sampled at regular time intervals.
  - *Quantized* representation of signals: signal level is given by a finite number of bits.





APPLICATIONS OF DIGITAL SIGNAL PROCESSING

- Embedded digital signal processing is everywhere!
- Examples:
  - Speech
  - Audio
  - Video
  - Radio/wireless
  - Radar
  - AI
  - Any application that processes signals in the digital domain.

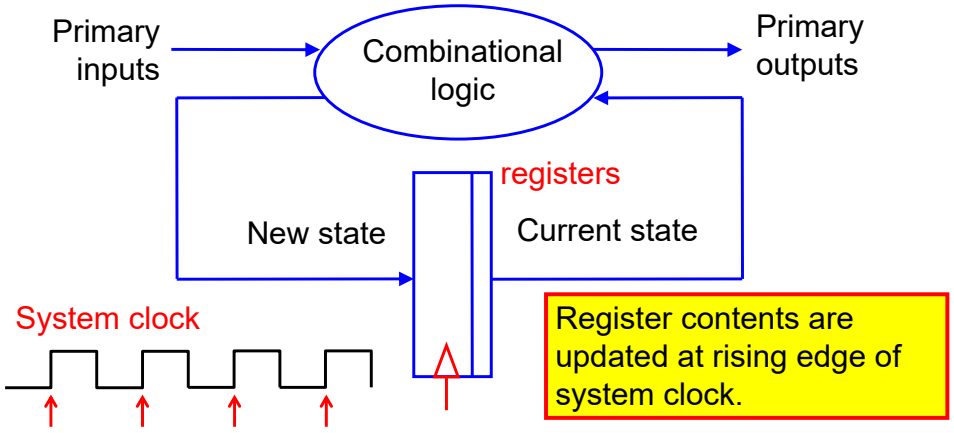
TYPICAL ALGORITHMS

- Filtering: FIR, IIR, with fixed coefficients or adaptive
- Convolution on images
- Encoding/decoding
- Compression/decompression
- Frequency-domain processing
- Downconversion: shifting carrier frequency in communication
- Etc.

TYPICAL NUMBERS

- Speech: 8 kHz, 12-16 bits
- Audio: 44 kHz, 16-24 bits, two channels (stereo)
- Video, various formats, e.g.:
  - HDTV approx. 2000 by 1000 pixels at 50 frames per second resulting in data rates of 100 MHz, 3 colors of 8-12 bits each

REGISTER-TRANSFER (RT) VIEW OF HARDWARE



SAMPLE FREQUENCY VS. SYSTEM CLOCK FREQUENCY

- The ratio between the system clock frequency and the sample frequency determines the necessity for parallel processing.
- A single processor clocked at, say, 100 MHz may handle all audio processing on its own: it has thousands of clock cycles available per signal sample.
- Video processing may on the other hand require multiple processors and/or dedicated hardware.

STREAMING VS. BLOCK-BASED

- *Streaming* data:
  - Data samples are processed as they arrive
  - Requires little local storage
  - Time-domain processing
- *Block-based* processing:
  - Stores incoming data until some block size is filled
  - Processes entire block
  - Think e.g. of an FFT (Fast Fourier Transform) or DCT (Discrete Cosine Transform)

IMPLEMENTATION PLATFORMS (1)

- *General-purpose processor* (GPP), such as a Pentium
- *Digital signal processor* (DSP):
  - Much better suited (parallel arithmetic in data path, support for "multiply-accumulate" operation, Harvard architecture for parallel access to data and program memory, etc.)
- *Multicore GPPs or DSPs* (trend!)
- *Very large instruction word* (VLIW) processor:
  - Many parallel arithmetic units in data path, each controlled by appropriate bits in instruction word
- *Graphics processing unit* (GPU):
  - *General purpose computation on GPUs* (GPGPU)
- *Neural processing unit* (NPU)

IMPLEMENTATION PLATFORMS (2)

- Processor arrays:
  - Think of *Montium* processor tile as developed in the CAES group (starting from the early years 2000, continued by spin-off *Recore Systems*, now *Technolution*).
  - Often interconnected by a *network on chip* (NoC), an interconnection structure somewhat comparable to data networks connecting computers (may be circuit switched or packet switched).
- User-defined architectures:
  - ASIPs (application-specific instruction processors)
- Dedicated logic:
  - ASICs (application-specific integrated circuits)
  - FPGAs (field-programmable gate arrays)

MAPPING PROBLEM

- How do we get the most efficient implementations of DSP algorithms on our platforms?
- Optimization criteria:
  - Fastest
  - Smallest
  - Minimal energy
  - Shortest design time
- In general, *flexibility* comes at the expense of *efficiency*:
  - In view of the costs of manufacturing ASICs, programmable hardware is often very desirable.

HIERARCHY AND OPTIMIZATION

- Design choices at higher hierarchical levels have the most impact:
  - Modifying your algorithm (e.g. getting rid of some computation in the inner loop) is often better than modifying your architecture (e.g. adding more arithmetic units).
  - Modifying your architecture (e.g. distributed memory instead of central memory) can be better than logic-level modifications (replacing ripple adders by carry look-ahead adders).
  - There is still place for dedicated logic for signal processing (e.g. phasor rotation).

AUTOMATED MAPPING

- Already familiar with *register-transfer level (RTL) synthesis* (clock-cycle true descriptions in VHDL mapped on cells from standard-cell library, see e.g. System-on-Chip Design course)
- *Architectural synthesis* (also called *high-level synthesis*, HLS) will automatically decide about the mapping of computations across clock cycles and architectural primitives.
  - Requires a formal representation of computations
  - And a formal representation of architectures

SILICON COMPILATION

- RTL synthesis is a mature technology:
  - All current-day digital IC design and FPGA design makes use of it.
  - Most RTL is hand-coded (in VHDL, Verilog)
  - HLS delivers machine-generated RTL.
- HLS is being talked about since the 1980s, tools exist, but they are not widely adopted.
  - To be covered later on in this course.
- HLS was also called *silicon compilation* (no longer popular):
  - A *software (SW) compiler* hides assembly-level details from the user.
  - A *silicon compiler* should hide RT-level details from the hardware designer.
- Next: SW compilation for DSP.

SW COMPILATION PROBLEM (1)

- When mapping on given programmable hardware, one talks of *compilation* rather than synthesis.
- Commercial processors often come with their own compilers.
- Designing an ASIP requires both:
  - The design of the hardware, and
  - The design of a compiler to map user programs onto the hardware.
- Compiling for DSPs, VLIW processors, etc. is more difficult than compiling for CPUs:
  - The challenge is how to optimally use the available parallel hardware,
  - Especially when the source code is sequential.

SW COMPILATION PROBLEM (2)

- Approaches:
  - *Leave all to the compiler.* This means that it is left to the compiler to discover the available parallelism in sequential code like C.
  - *Language extensions.* Extend a language like C with constructs (pragmas etc.) that explicitly describe parallelism. Use the information to optimally exploit parallelism in target hardware.
  - *Extensions with APIs (application programming interface)*. Have a library of routines that optimally exploit the parallel hardware and force user to use these APIs.

MULTICORE PROGRAMMING

- Often based on *threads*, sequential pieces of code that run on a single processor.
- Parallel computing amounts to distributing threads across the available processors.
- Communication and synchronization is based on:
  - Shared memory
  - Message passing

OPENMP

- OpenMP (open multi-processing):
  - Language extension with annotations for C/C++/Fortran
  - Supported by GCC
- Example: *for loop* will be split in multiple threads executing on multiple cores

```
int main(int argc, char *argv[]) {
  const int N = 100000;
  int i, a[N];
  #pragma omp parallel for
  for (i = 0; i < N; i++)
    a[i] = 2 * i;
  return 0;
}
```

Question: why is the code easy to be executed in parallel?

GOALS OF MODELING

- *Verification by simulation:*
  - mostly executed on one CPU;
  - should provide the relevant degree of accuracy.
- Models are also used for *formal verification.*
- *Synthesis;* maps model on a realization consisting of:
  - a single processor (general purpose/digital signal processor);
  - multiple processors;
  - dedicated hardware;
  - a mixture of dedicated hardware and processors.

MODELING OF TIME

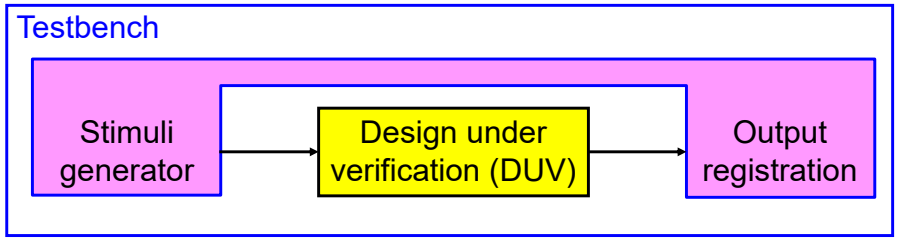
- Continuous time:
  - solve differential equations for analog simulation.
- Discrete time:
  - delay from input to output of hardware blocks;
  - clock signals may be involved (register-transfer level, RTL);
  - event-driven simulation may be used.
- Untimed:
  - no delay inside hardware blocks;
  - timing controlled by external signals and flow-control blocks such as FIFO (first-in first-out) buffers.

MODELING OF SIGNALS

- Analog values:
  - voltages, currents;
  - floating-point data types.
- Digital values:
  - bits and bit vectors;
  - bit vectors need an interpretation: e.g. unsigned, 2's complement signed, fixed-point or floating-point numbers.
- More complex data types: e.g. records.

"CLASSICAL" SIMULATION

- Based on simple generation of stimuli and designer inspection of waveforms or text output for determination of correctness.
- It is quite common to base stimuli generation and output registration on data streams read from and written to a file.



SHORTCOMINGS OF CLASSICAL SIMULATION

- There is only one design, the "implementation". The "reference" is in designer's and verification engineer's mind.
  - Good idea to have separate verification engineer, for a "second opinion" on the interpretation of specification.
- DUV is at RT level and becomes available in a late stage of the design:
  - Software development cannot start easily in time; verification with software will delay the tape-out.
  - RTL code is slow to simulate; it is only feasible to simulate small software programs.

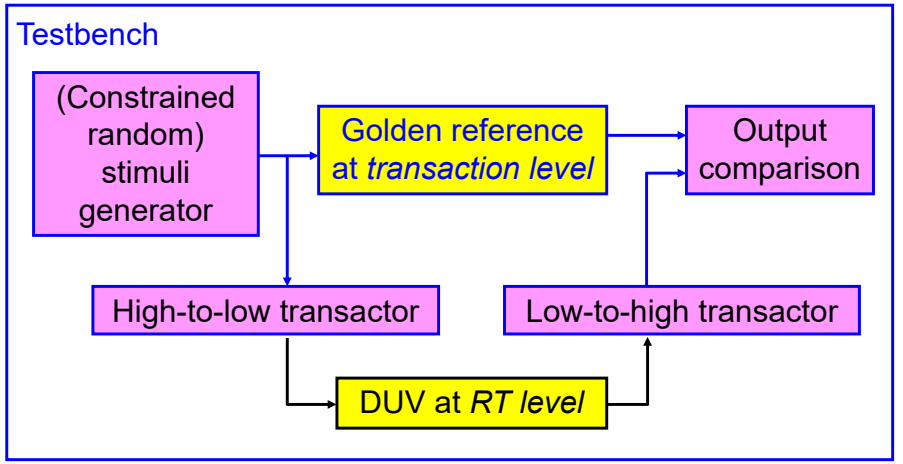
TRANSACTION-LEVEL MODELING

- Abstract way of looking at hardware:
  - I/O signals not at the bit level, but as abstract data structures
  - Behavior specified in terms of transactions
  - In general, not clock-cycle accurate
- Example:
  - "Write to memory" is a transaction; its implementation will involve preparing data, address and control signals with the required *timing* relations.
- *Transactors* translate transactions to bit-level signal changes and back.

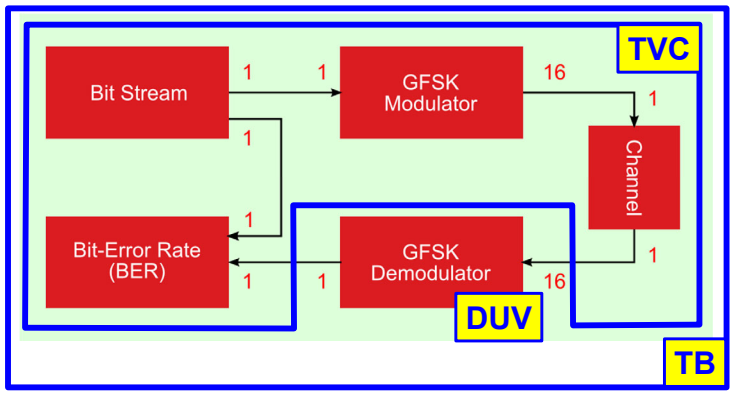
FEATURES OF ADVANCED SIMULATION

- *Self-checking* testbenches: waveform inspection only for debugging.
- Transaction-level "*golden* reference design" built into testbench.
- Golden reference design, being not clock-cycle accurate, executes much faster and can be used for software verification at an early stage.
- Stimuli generation makes use of *constrained random pattern generation* to increase code coverage.
- Transactors evolve together with RT-level implementation.
- Assertions are extensively exploited.

ADVANCED TESTBENCH STRUCTURE



SNEAK PREVIEW: GFSK PROJECT



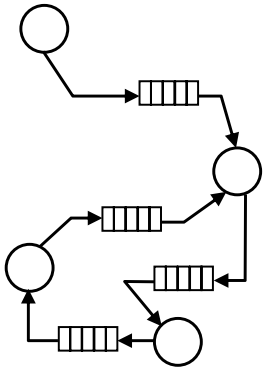
DUV = design under verification
TVC = test-vector controller; TB = testbench

COMPUTATION AND COMMUNICATION

- The issue is the modeling of *parallelism* present in hardware. A system consists of:
  - entities computing output signals from input signals.
  - a structure interconnecting the entities.
- Interconnection may be *direct* or *buffered*.

KAHN PROCESS NETWORK (KPN)

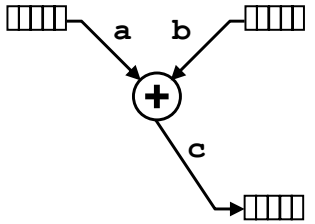
- Network of entities (*nodes*) interconnected by FIFO buffers.
  - *Reads* are *blocking*, i.e. a computation waits until there is data available to read.
  - *Writes* are *non-blocking*, i.e. writes are always allowed implying that the FIFO buffers have unbounded depths.
- The behavior of the nodes can be given in a traditional sequential programming language.



Gilles Kahn, FR, 1974

EXAMPLE OF A KPN ADDER NODE

```
read(a);
read(b);
c = a + b;
write(c);
```

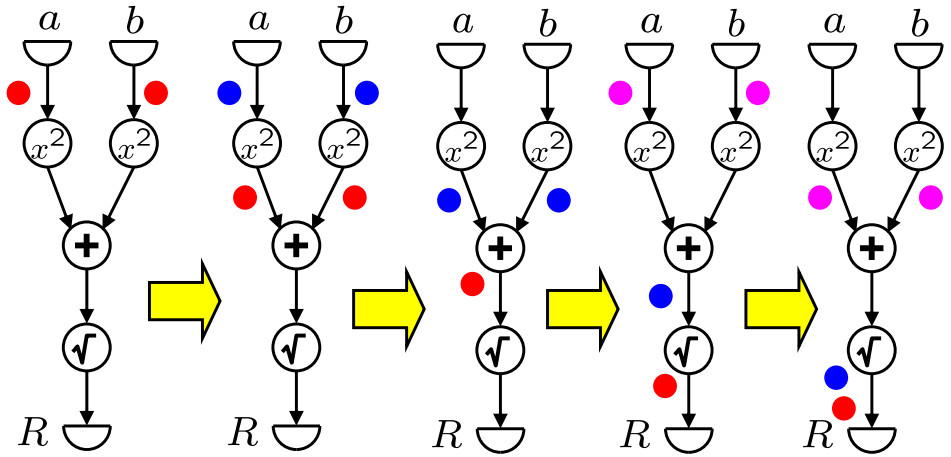


The addition can only be executed when input data are available; otherwise, the operation waits.

DATA-FLOW BASICS

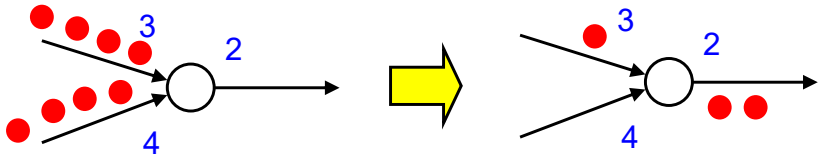
- A *data-flow graph* (DFG) consists of nodes (vertices) and edges.
- In its most general form, a DFG is equivalent to a KPN.
- Nodes perform computations.
- Edges indicate *precedence* relations and behave as FIFOs.
- Data flow is best understood in terms of *tokens*, carriers of data.
- A node will *fire* when a sufficient number of tokens is available on all its inputs.
- The result of firing is that tokens are *consumed* at the inputs and tokens are *produced* at the outputs.

TOKEN FLOW EXAMPLE: $R = \sqrt{a^2 + b^2}$



SYNCHRONOUS DATA FLOW (SDF)

- Characterized by fixed consumption and production numbers for each node invocation.
- Suitable for the specification of *multi-rate* DSP algorithms.



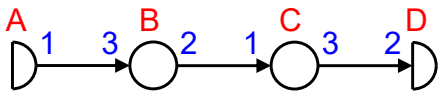
Lee, E.A. and D.G. Messerschmitt, "Synchronous Data Flow", Proceedings of the IEEE, Vol. 75(9), pp 1235–1245, (September 1987).

CONSISTENCY IN SDF

- It is relatively easy to check whether:
  - No deadlock occurs;
  - Number of tokens on an edge does not grow indefinitely;
  - There are sufficient initial tokens to keep loops going.
- A consistent graph:
  - Has a *repetitions vector* indicating how often a node needs to be invoked for one computation of the graph;
  - Can be scheduled statically, without the need to implement FIFO buffers for the edges.

SOFTWARE SYNTHESIS

- Example graph:

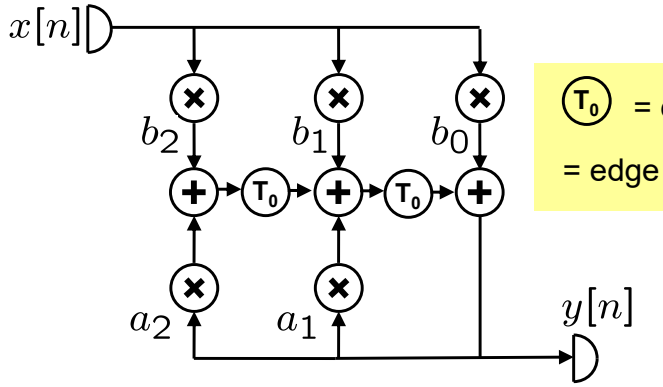


Rep. vector: (3) (1) (2) (3)

- Possible *single-processor* schedule: (3A)B(2C)(3D)

Question: if 3 tokens are "streamed into" the system at A, how many tokens are then "streamed out" at D?

DATA-FLOW GRAPH EXAMPLE

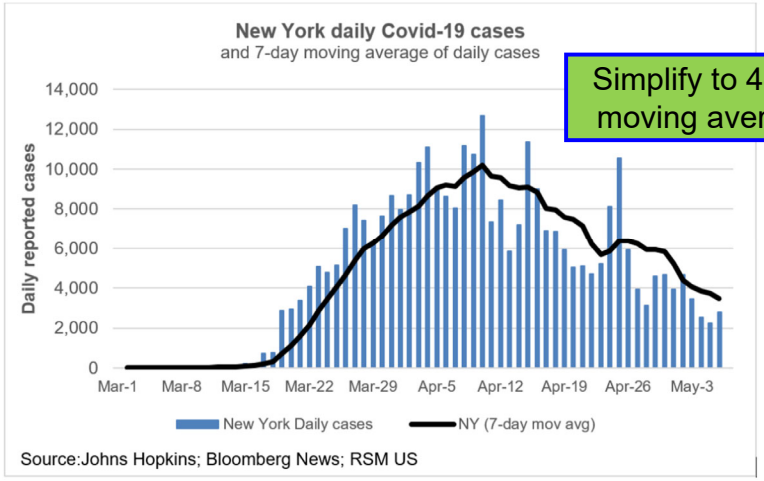


T_0 = delay element

= edge with one initial token

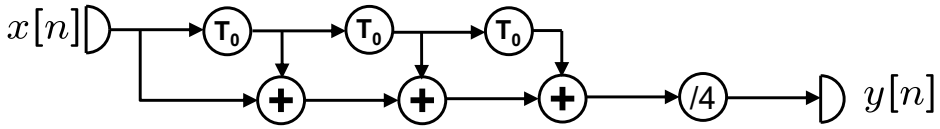
Second-order IIR filter

DIY: DFG FOR MOVING AVERAGING

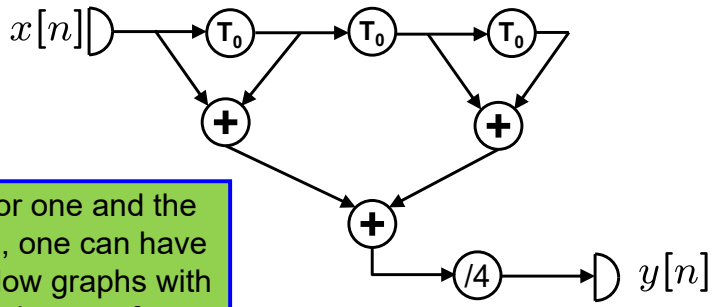


Simplify to 4-day moving average

DFG FOR MOVING AVERAGING (1)



DFG FOR MOVING AVERAGING (2)

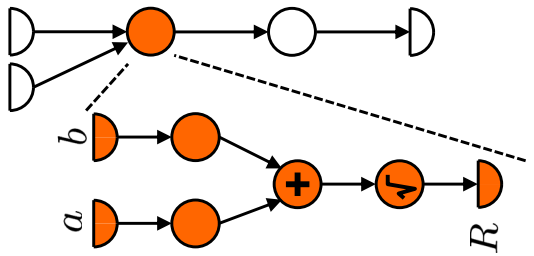


Conclusion: for one and the same problem, one can have multiple data-flow graphs with a varying degree of parallelism.

Is there a more efficient alternative?

HIERARCHICAL DFGS

- Nodes in a DFG do not need to be *atomic* (indivisible computations) but could be expanded into DFGs themselves.
- In this way, one gets *hierarchical DFGs.*
- Nodes that do not have subgraphs are called *primitive.*
- Example: *Simulink.*



GRAPHICAL VS. TEXTUAL FORMATS

- It is obvious that DFGs are very suitable as an *internal representation* format of a synthesis tool.
- DFGs are, however, not always the most suitable format for a designer to specify a computation; one does not want to draw separate addition nodes for each addition and interconnect these nodes.
- The solution is to start with a textual representation and convert it to a DFG by means of *data-flow extraction*.
- Graphical-entry tools are mainly useful for specifying complex computations with hierarchy; primitive nodes (that do not have subnodes) are normally specified in a textual format.