Σχεδίαση Συστημάτων Υλικού-Λογισμικού

Γιάννης Παπαευσταθίου

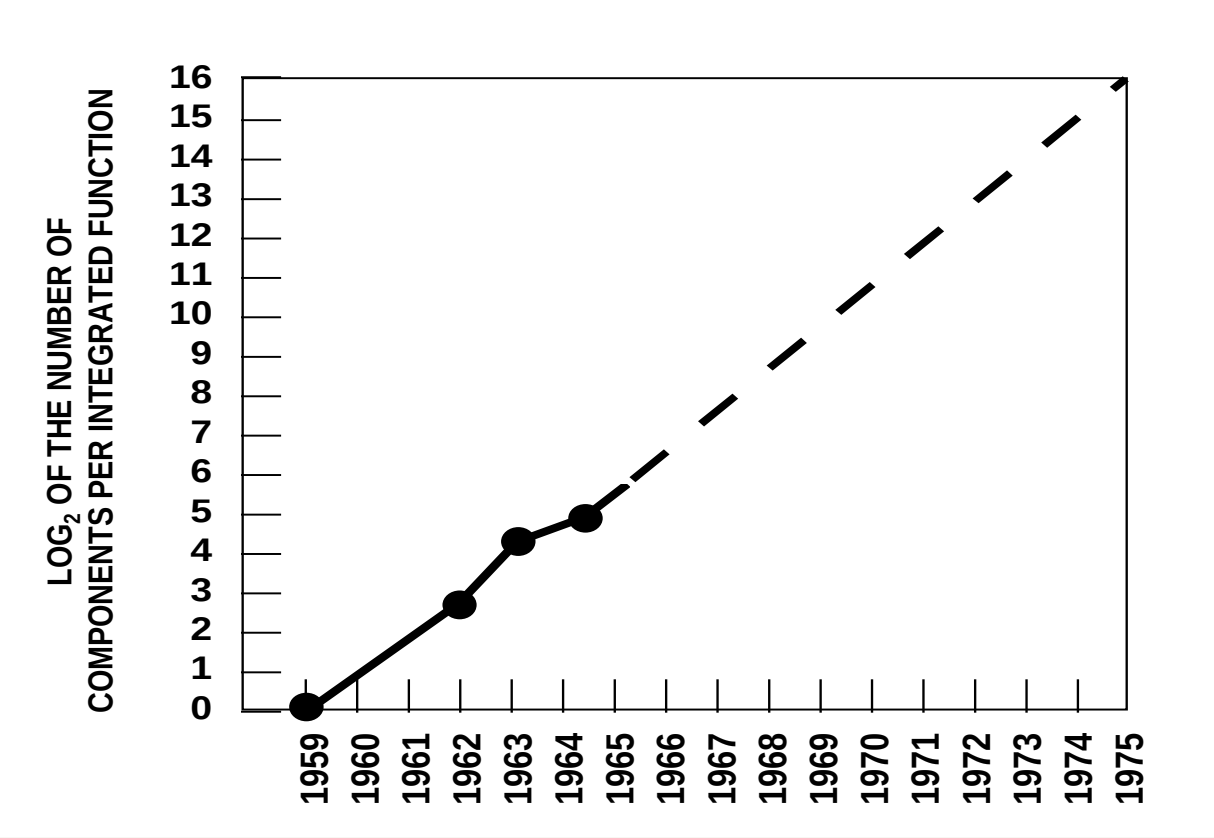
ακ. έτος: 2020-2021

1

Main Motivation for this course (now in week 10 ???????)

- Moore's law used to make transistors faster and cheaper with every generation
- Now this is no longer true, but they still become more numerous…
- … and the variety of application fields keep growing
- Design cost and costs due to late arrival on the market may determine success or failure of a product
- Design automation (modeling, analysis and synthesis) is essential to keep the industry moving:
  - Reduction of design time by raising the level of abstraction
  - Ability to perform HW/SW and architectural trade-offs to optimize product cost and performance

Moore's original law

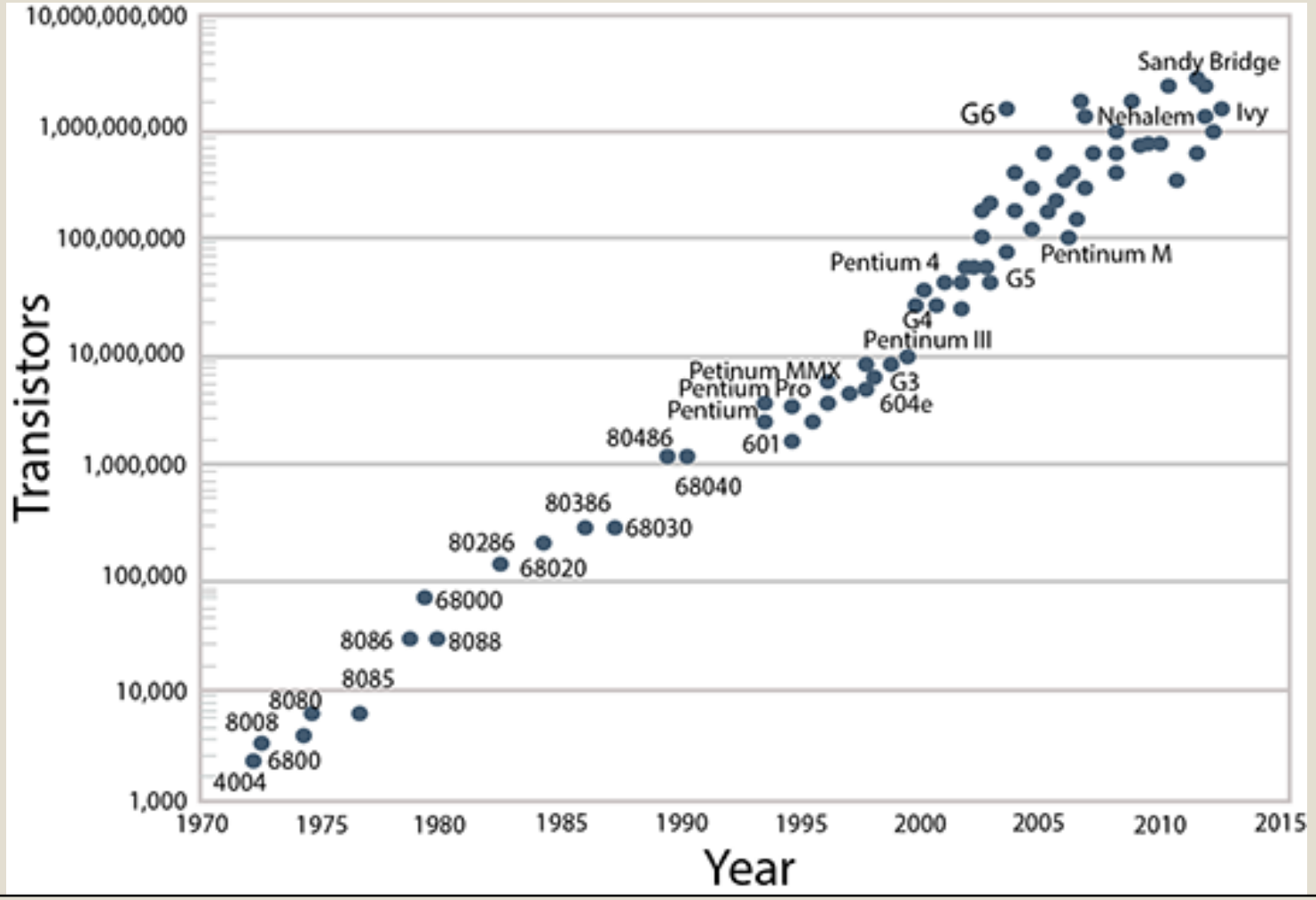


Electronics, April 19, 1965.

Moore's law

- Based on economics, not physics (even though physics determine it in part)
- Needed to be able to forecast what will be the performance of semiconductors when my electronic product will hit the market
  - How many transistors will be available on a chip in 2, 5, 10, 20 years?
  - What will be their cost?
  - At which frequency will the chip work?
  - What will be its power and energy consumption?
- The unprecedented complexity of this forecasting effort required a unique industry-wide effort: the now-defunct International Technology Map for Semiconductors

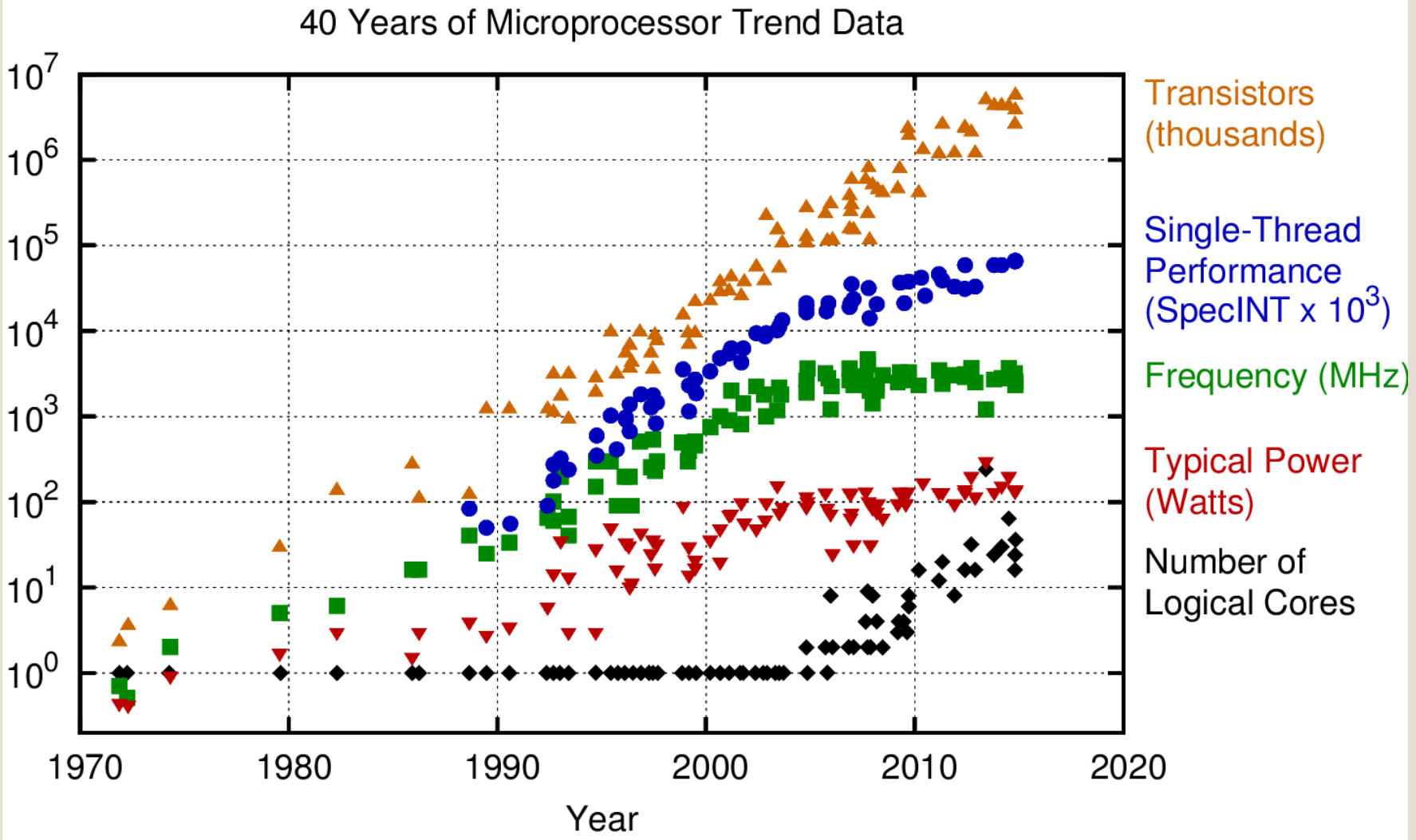
The basis of Moore's law



The number of transistors on a microprocessor doubles every 2 years

Courtesy, University of Wisconsin

The various aspects of



«This is the end, my friend»

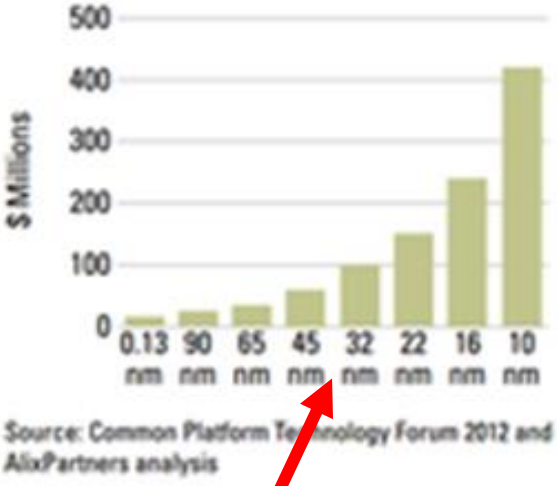
- Transistor costs no longer go down
- It becomes more and more expensive to justify the adoption of new technologies

Technology	Gates/mm² (KU)	Gate utilization (%)	Used gates/mm² (KU)	Parametric yield impact (Δ from D_0 yield)	Actual used gates/mm² (KU)	Gates/ wafer (MU)	Wafer cost ($)	Wafer cost (Δ)	Cost per 100M gate ($)
90nm	637	86	546	97	532	33,831	1,357.62	--	4.01
65nm	1,109	83	919	96	885	56,330	1,585.71	16.8	2.82
45/40nm	2,139	78	1,677	92	1,538	97,842	1,898.83	19.7	1.94
28nm	4,262	77	3,282	87	2,855	181,658	2,361.84	24.4	1.30
20nm	6,992	65	4,524	73	3,293	209,541	2,981.75	26.2	1.42
16/14nm	10,488	64	6,712	67	4,497	286,140	4,081.22	36.9	1.43
10nm	14,957	60	8,974	62	5,564	354,013	5,126.35	25.6	1.45
7nm	17,085	59	10,080	60	6,048	384,813	5,859.28	14.3	1.52

The reason

Source: Common Platform Technology Forum 2012 and AlixPartners analysis

Source: Common Platform Technology Forum 2012 and AlixPartners analysis



Source: Common Platform Technology Forum 2012 and AlixPartners analysis

FIGURE 4. (L) Fab Costs by Node (in US$ billions) **FIGURE 5.** (C) Process Technology Development Costs by Node (US$ billions) **FIGURE 6.** (R) Chip Design Costs by Node (US$ millions)

- Fabrication line, technology development, mask costs grow faster than transistors shrink
- Design costs grow as much as mask costs, since getting working chips back at the first round is essential

Design and production costs

- NRE cost (Non-Recurring Engineering cost)
  - Paid once per design
  - Includes design and mask production costs
  - Independent of the number of manufactured chips N
- Unit production cost (U)
  - Paid once per manufactured chip
  - Includes raw material, plant depreciation (really an NRE but carefully hidden from most of the industry) and labor
  - Proportional to N
- Total per-product cost (C): depends on U, NRE, N
  - C = (NRE / N) + U
- The best amount of design effort depends on N…

16/12/2020 - 9

The cost of flexibility

- SW-like programmability and flexibility has a cost
- About 1 order of magnitude loss in terms of unit cost, performance and energy between levels:
  - General-purpose CPU
    - Most flexible, lowest design cost
  - Application-Specific Instruction set Processor (ASIP)
    - DSP, GPU, vector processor
    - Customized datapath, memory subsystem
  - Field-Programmable Gate Array (FPGA)
    - Custom-like HW without mask costs
    - No more fetch-decode-execute cycle (wasting **energy**)
  - Application-Specific Integrated Circuit (ASIC)
    - Full customization of datapath, control and memory

Programmability versus specificity

- Programmability:
  - Loses efficiency (cost, performance energy, …)
  - Gains flexibility (ease of adaptation, robustness, …)

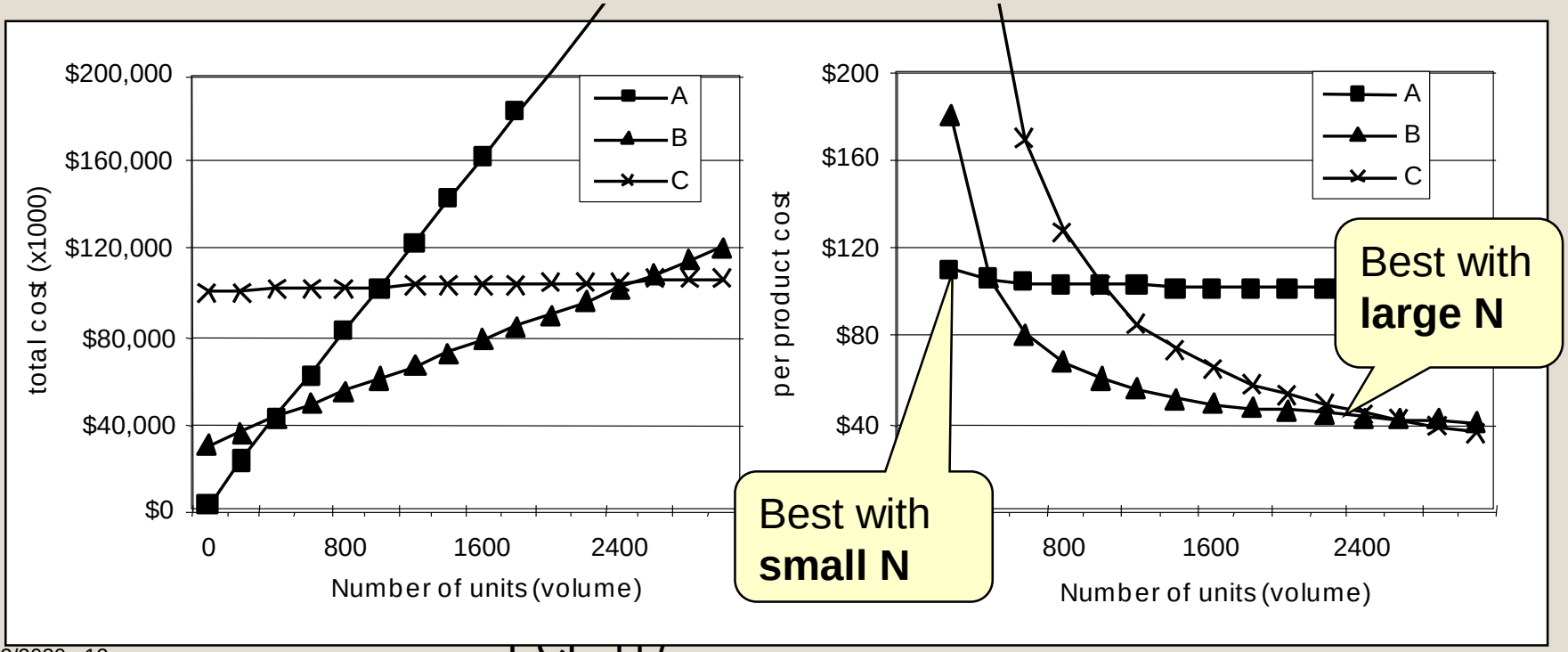
PC, tablet | Smart phone | Video-game console | Telecom switch | Engine control, camera | Washing machine, Elevator

Programmability, flexibility

Efficiency

Various implementation options

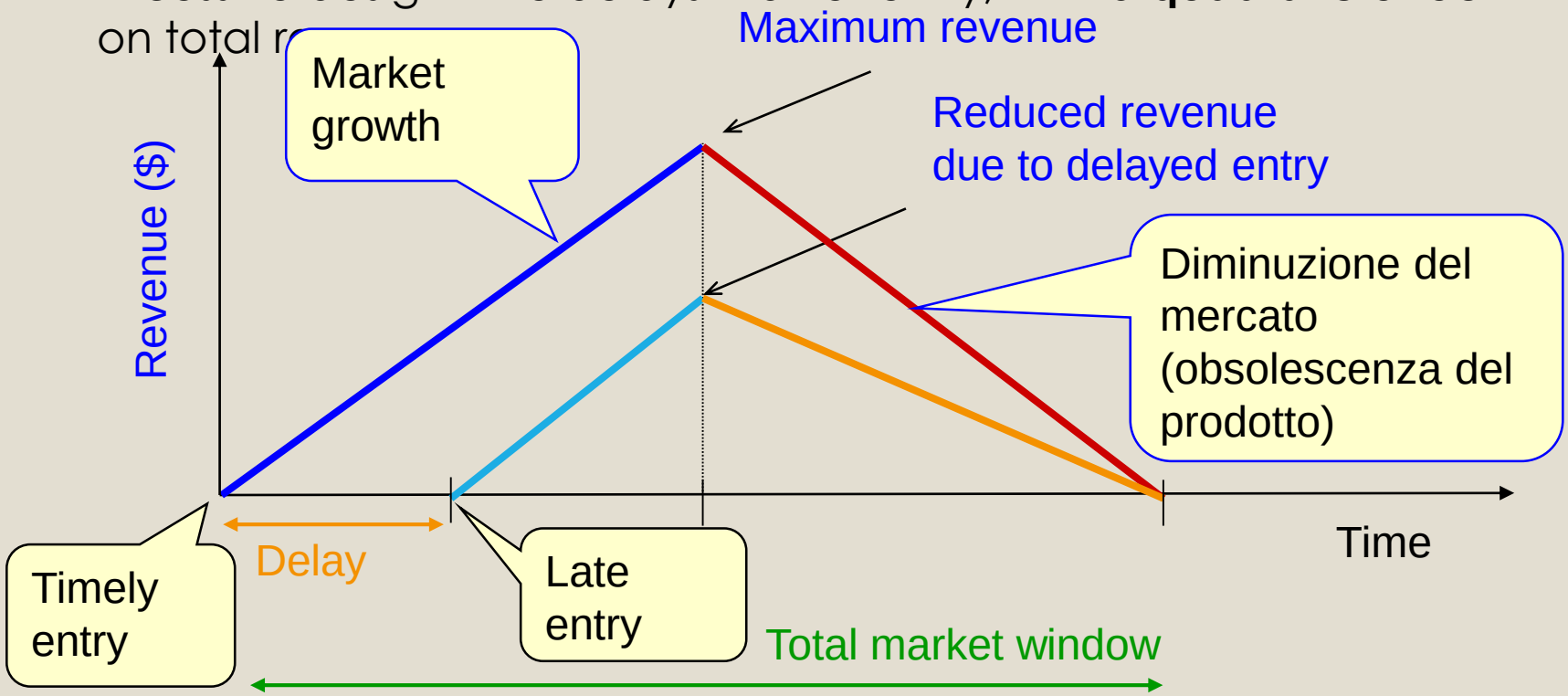
- Mostly SW: NRE = $0.2M U = $100
- Programmable HW: NRE = $10 M U = $30
- Dedicated HW: NRE = $100 M U = $2



16/12/2020 - 12

Time to market

- Excessive design time delays market entry, with a **quadratic** effect on total r



16/12/2020 - 13

Outline

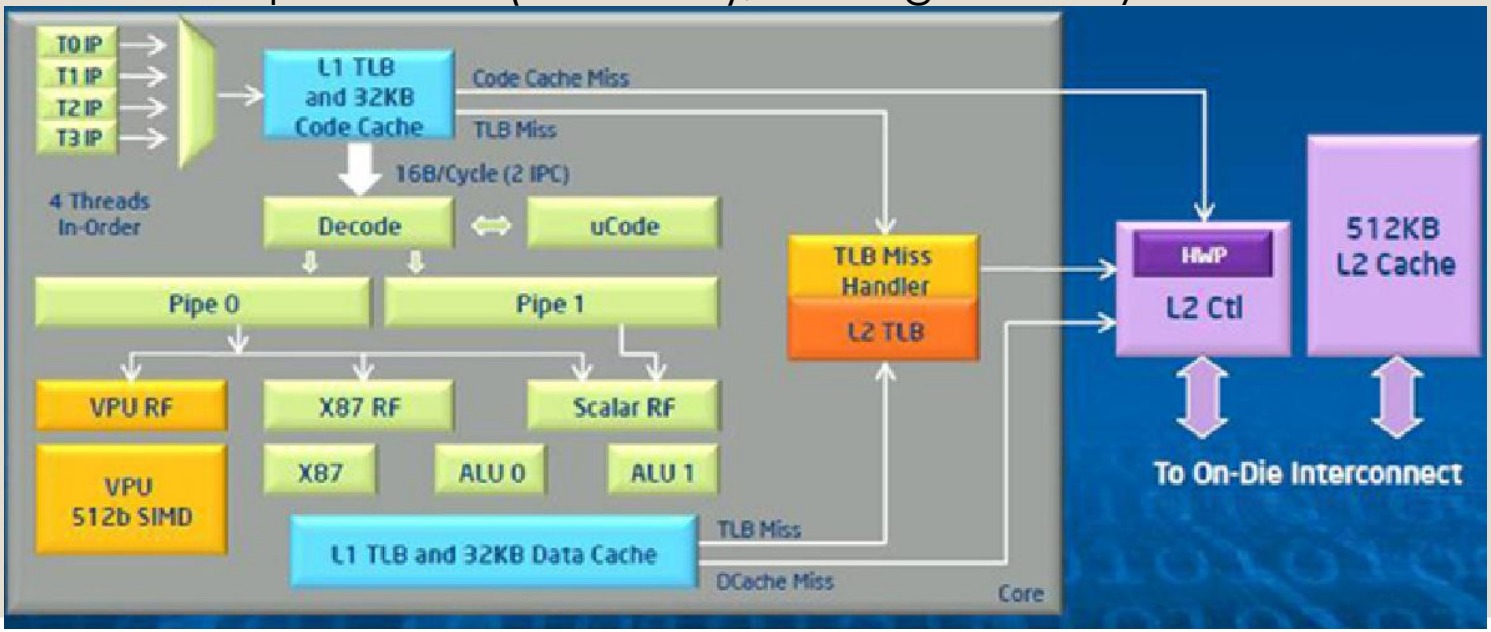
- Motivation:
  - Definition of embedded systems
  - Moore's law
  - The economics of electronic system design
- **Reduction of design costs and performance/cost optimization**
  - Platforms
  - Energy/performance trade-offs
- Modeling languages and expressive power
  - Data-dominated vs control-dominated
  - Turing machines and undecidability

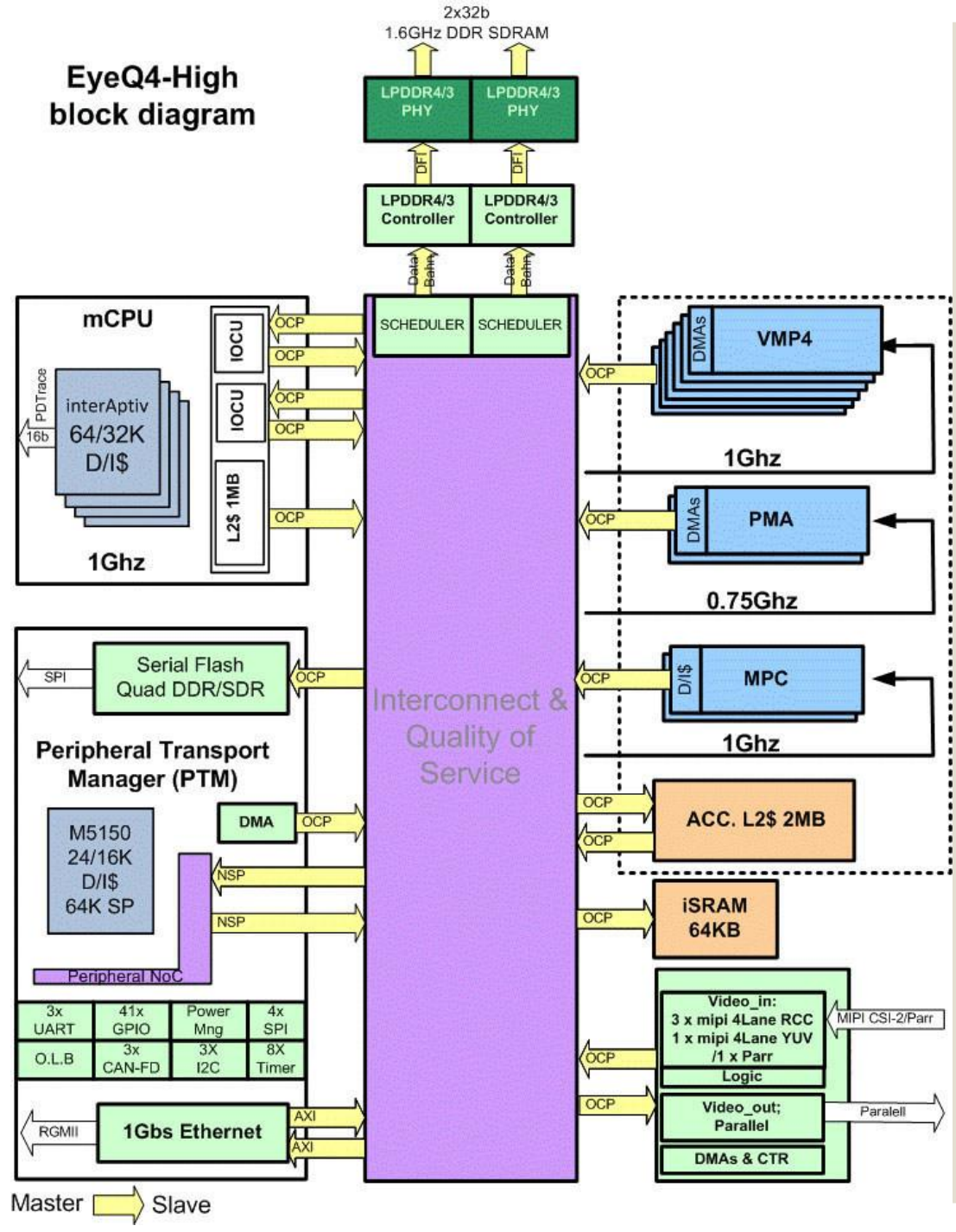
How to reduce design time and NRE costs?

- Most NRE costs come from mask costs:
  - Directly (exponentially growing with technology generation)
  - Indirectly, since avoiding re-spins due to incorrect design requires very long verification times
- Reduce mask costs: **use pre-designed platforms**
  - Systems-on-chip (processors, memory, accelerators)
  - FPGAs with processors
  - Both involve **HW/SW trade-offs**
- **Reduce design time**: main topic of this course
- Reduce verification time: use pre-designed Intellectual Property (IP) blocks
  - Will not be covered in this course, except as full-chip integration verification based on SystemC

Example of CPU platform

- Intel processor-based multi-core platforms (PCs, data centers)
  - Multi-core multi-threaded CPU
  - High-performance memory interface and interconnect
  - Vector processor (Xeon Phi), sharing memory interface with



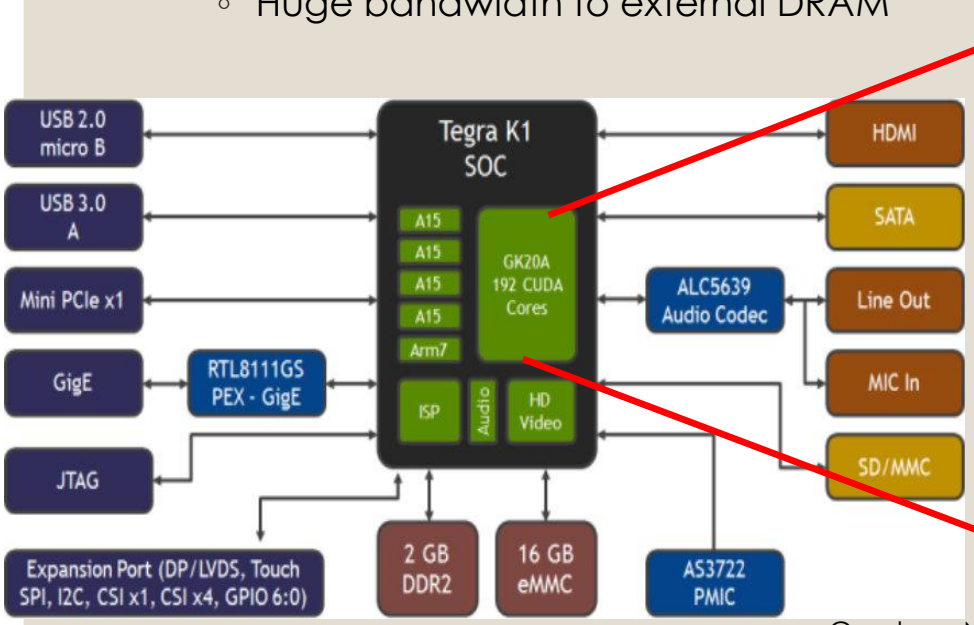


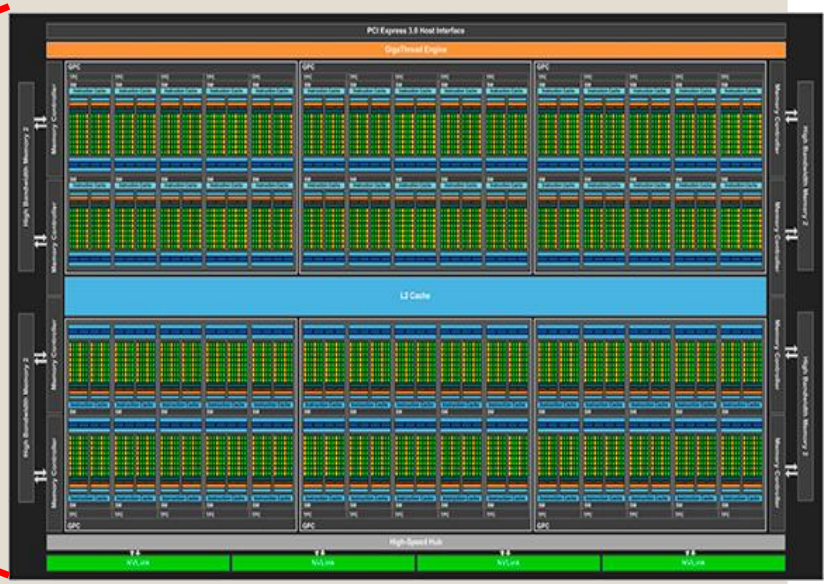
- Example: MobilEye EyeQ
  - Multi-core processor (MIPS-based)
  - SIMD Vector Multi-Processor for image processing
- Similar to, e.g., Kalray MPPA

Courtesy: MobilEye

Example of ASIP platform: GPUs

- Born for PCs, used for both HPC and embedded:
  - High-performance computing, e.g. Nvidia Tesla
  - Embedded applications, e.g. Nvidia Jetson
- Ultimate parallel architecture
  - Huge number of cores (thousands of floating point ALUs)
  - Huge bandwidth to external DRAM



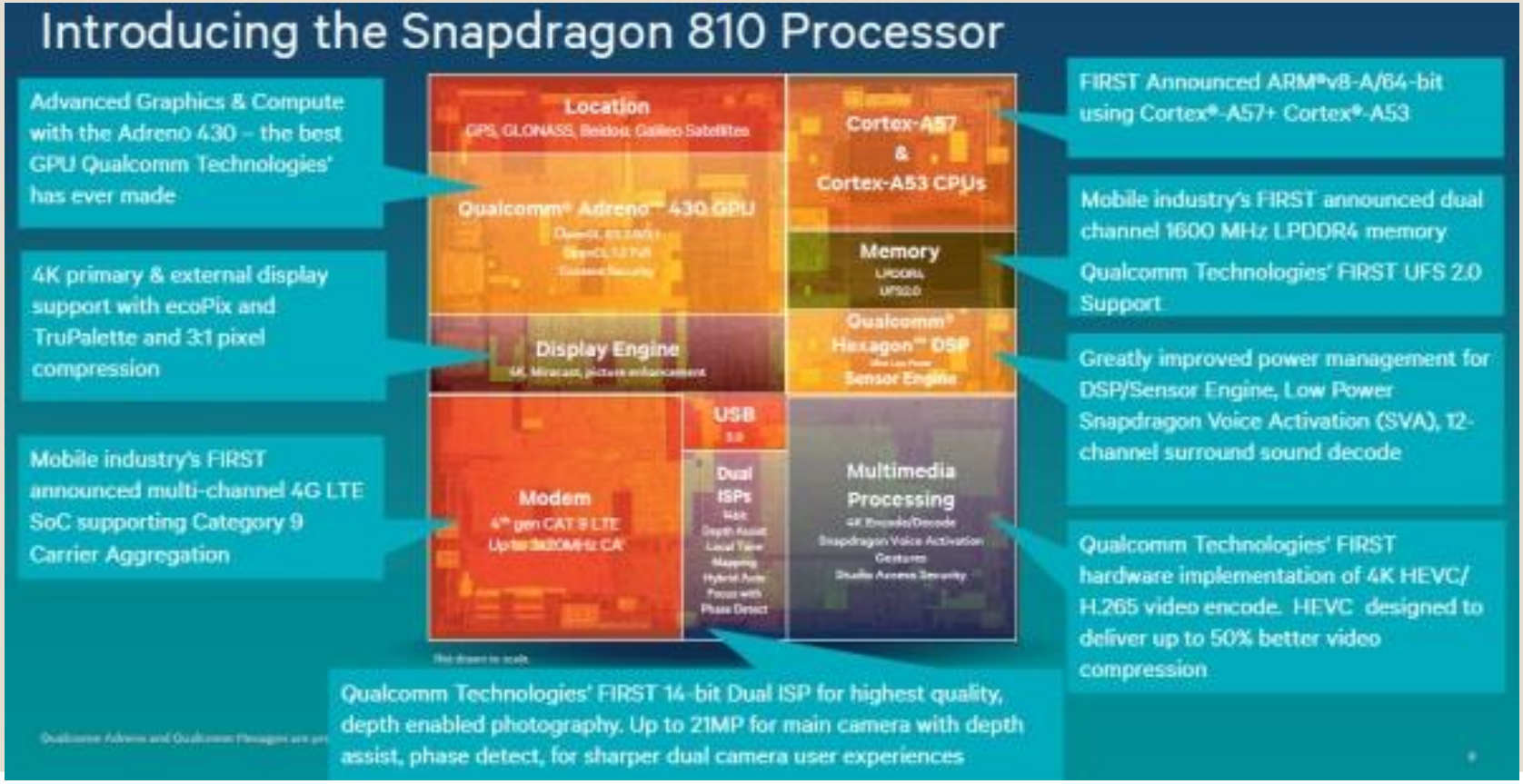


Courtesy: Nvidia

Example of SOC platform

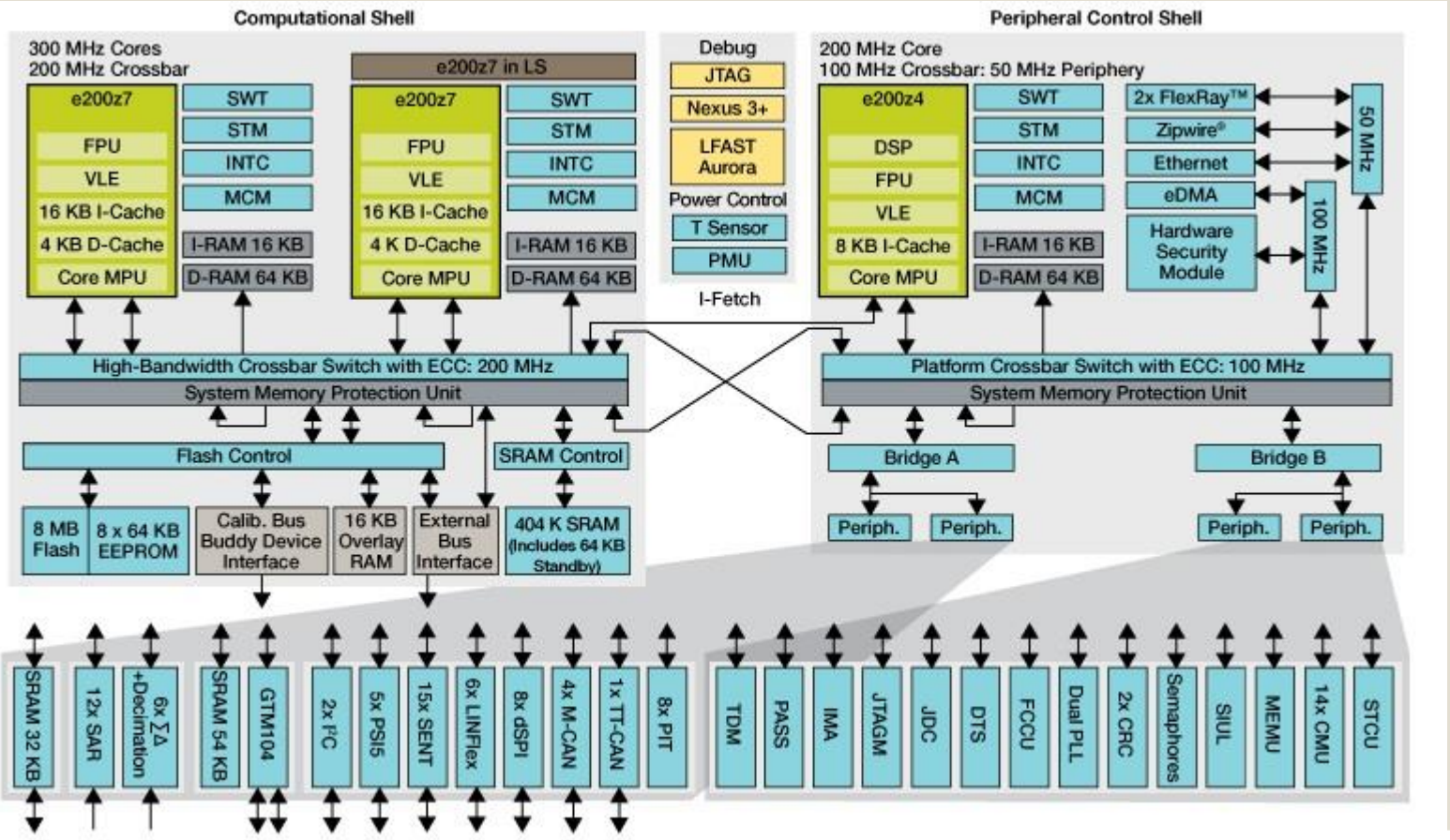
- Example: Snapdragon 810 cell phone SOC
  - Multi-core processor
  - Application-specific accelerators and peripherals

Courtesy: Qualcomm



Example of SOC platform

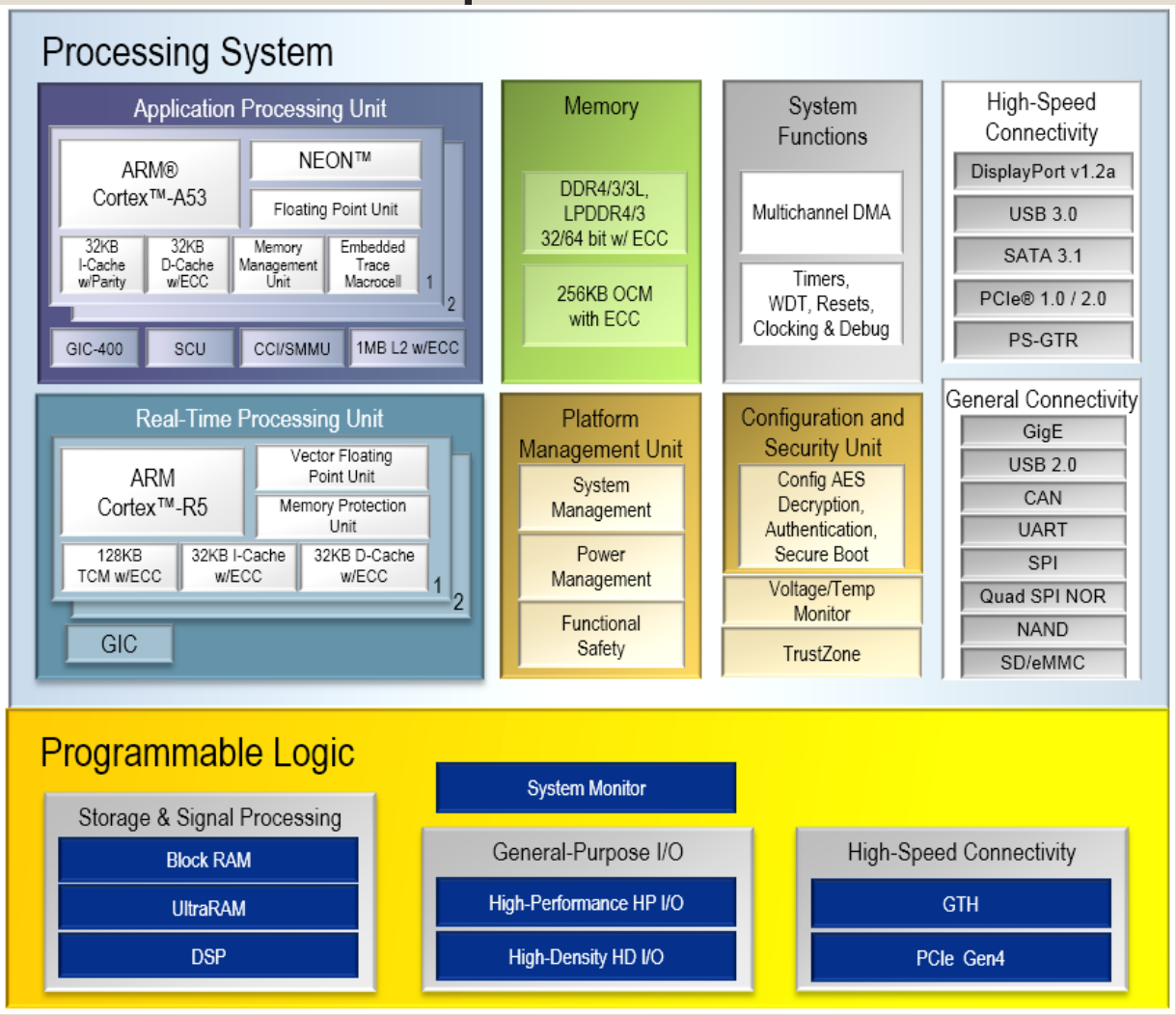
- Example: NXP MPC5777M automotive SOC
  - Multi-core processor (twin for safety)
  - Application-specific peripherals



Courtesy: NXP

Example of FPGA platform

- Xilinx Zynq
  - Multi-core processor
  - Programmable logic
  - Some ASIC-style peripherals
- Similar to, e.g. Altera Arria



Courtesy: Xilinx

Platform analysis

- Several common points:
  - All include one or more single-core or multi-core processors
  - Some processors are highly customized
    - SIMD in GPU
    - Vector processor in EyeQ
  - All include some HW
    - Application-specific HW for efficiency (e.g. modem, video accelerators, peripherals)
    - Reprogrammable HW for flexibility
- Key differentiators are application-specific accelerators and peripherals
- In all cases: HW/SW architecture design (including HW/SW partitioning problem)

Performance and energy efficiency

Type	Device	GFLOPS	Cost (€)	Power(W)	GFLOPS/€	GFLOPS/W
Multi-core	Intel E5-2630v3 8x2.4GHz	600	700	85	0.85	7.05
	Intel E5-2630v3 10x2.3GHz	740	1250	105	0.59	7.04
Many-core	Xeon Phi, knights corner, 16GB	2416	3500	270	0.69	8.94
	Xeon Phi, knights landing, 16GB	**7000**	3500	300	**2.00**	23.3
GPU	Nvidia GeForce Titan X	**7000**	1000	250	**7.00**	28
	Nvidia Tesla V100 (matr. mul)	**120000**	?	300	?	**400**
	Nvidia Tegra X1	512	450	**7**	**19.40**	**73**
	ARM Mali T880 MP16	374	?	**5**	?	**74?**
	Radeon firepro S9150	**5070**	3500	235	1.44	21.5
FPGA	Altera Arria 10	1500	2000	**30**	0.75	**50**
	Altera Stratix 10	**10000**	3000	125?	**3.33**	**80**
	Xilinx Zynq Ultrascale+	**5000**	2000	**40**	**2.30**	**115**

Courtesy: I. Mavroidis, ICS/FORTH

HW/SW co-design basics

- Models and methods to **reduce embedded system-level design time**, where **most of the cost/performance trade-offs occur**
- We will assume an underlying HW+SW platform
  - SW for flexibility and reduced design costs
  - HW for to improve performance, unit cost and energy consumption
- Ideal goal of Model-Based Design:
  - Model once
  - Verify once
  - Synthesize to many targets (SW, FPGA, HW, …)
- Requires verification and synthesis techniques, that will be covered in this course

Outline

- Motivation:
  - Definition of embedded systems
  - Moore's law
  - The economics of electronic system design
- Reduction of design costs and performance/cost optimization
  - Platforms
  - Energy/performance trade-offs
- **Modeling languages and expressive power**
  - Data-dominated vs control-dominated
  - Turing machines and undecidability

Control-dominated versus data-dominated

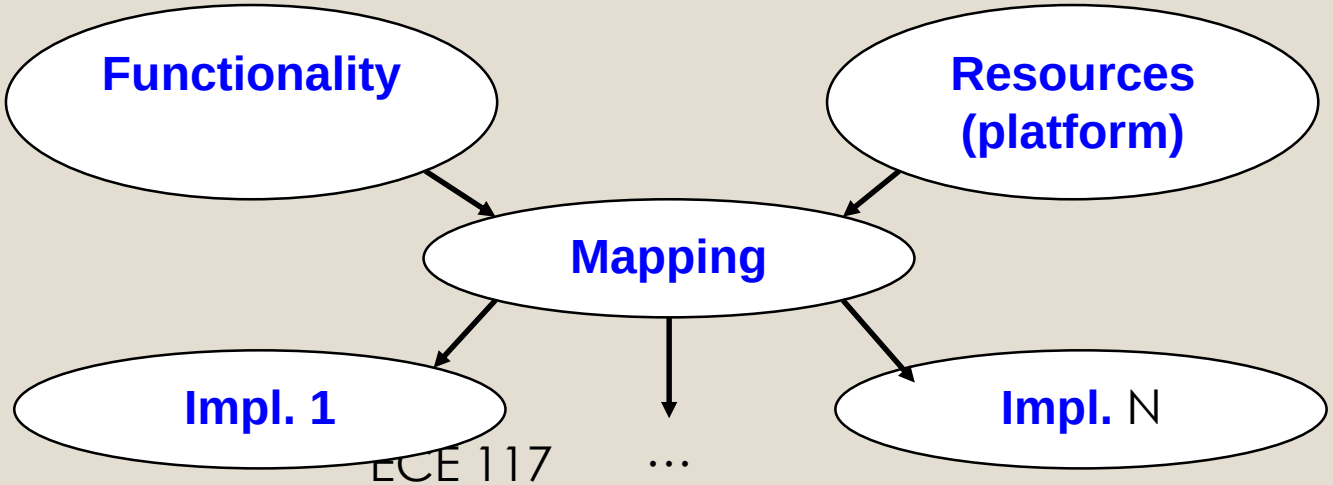
- No single model is suitable for every system
- Control-dominated systems emphasize:
  - Decisions
  - Rapid reaction to stimuli, latency
- Data-dominated systems emphasize:
  - Numerical computations
  - Throughput
- Control-dominated system design is based on extended, concurrent Finite State Machines
- Data-dominated system design is based on Dataflow Networks
- "Real" designs require a mix of both…

Expressive power versus analyzability

- Ideally, a design model should be:
  - **Expressive**, to represent anything a designer would like to design
  - **Compact**, because design time is proportional to the number of designed "objects" (from transistors to FFT stages)
  - **Executable**, to enable simulation-based verification
  - **Analyzable**, so that properties can be easily verified:
    - Correctness ("design verification")
    - Equivalence ("implementation verification")
  - **Synthesizable** and **optimizable**, so that many different implementations can be obtained and explored quickly
- Unfortunately, these properties are incompatible
- Expressiveness implies limited analyzability, which in turn implies limited optimizability

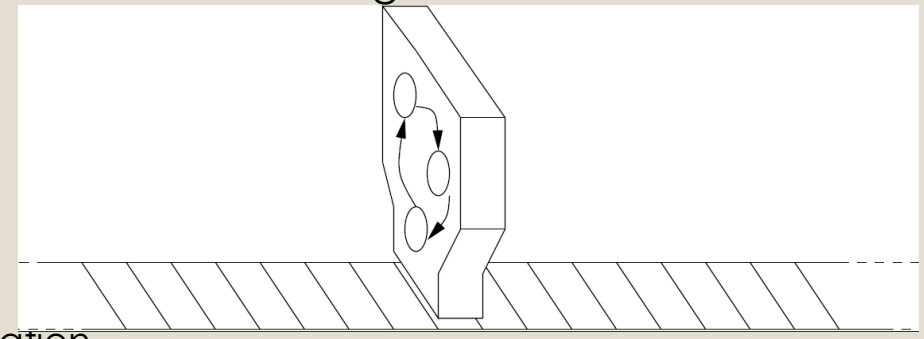
Platform-based design and Y-chart

- A design model should be **orthogonal**, to allow **separation of concerns** between:
  - Functional ("what") aspects
  - Non-functional ("at what cost and performance") aspects
- The implementation ("how") can then be derived by synthesis
  - Or, in the "old world", by manual refinement…



Our "black beast": the Turing Machine

- The simplest and most powerful computational model ever devised
  - A tape with an infinite set of locations, each holding a value from a finite alphabet
  - A "current location" on the tape
  - A Finite-State Machine controller which can at each "step":
    - Read a value from the current location
    - Change state, based on the value read
    - Write to the current location and move one location left or right
  - When the FSM starts, computation inputs are on the tape
  - When the FSM reaches its final state, the computation is "done" and the result is on the tape



Our "black beast": the Turing machine

- According to Church's thesis every "computable function" can be computed by some TM
- In terms of computing power (i.e. expressiveness):
  - It is equivalent to all other "Turing-complete" models
    - General Recursive Functions,
    - λ-calculus
    - (as we will see) Dataflow networks
  - It is strictly more powerful than pushdown automata (from compiler theory) and Finite State Machines
- Unfortunately, there is no way to prove for every TM that it will ever stop and produce a result
- Most non-trivial properties of TMs are undecidable in general

Turing's undecidability theorem

- Theorem: "there is no computable function halts(f) that tells correctly if a given TM f halts or not"
  - Remember that
    - For each computable function g there is at least one TM TM_g that computes it
    - For each TM TM_h there is an equivalent computable function h
- Proof: consider the TM_g that computes:

  g() {if (halts(g)) while (1) {}}

  1. If halts(g) returns true, then g does not halt
  2. If halts(g) returns false, then g halts
- we have a contradiction (by "diagonalization")
- Essential problem with all systems that can "encode themselves" (see Gödel's undecidability theorem)

Consequences of Turing's theorem

- Any computational model that is expressive enough to compute all functions is of limited use for our purposes, because we want to be able to:
  - Analyze our models e.g. to determine that:
    - They will always keep controlling their system
    - Given an input they will provide a response in a bounded amount of time
  - Verify that they are equivalent to another (specification) model
  - Synthesize and optimize them
- C (and C++ and Java) is Turing-complete if we allow:
  - Either dynamic memory allocation
  - Or unbounded recursion

Consequences of Turing's theorem

- Infinite state, and the ability to decide what to do based on it, is essential for undecidability of most "interesting" questions
- This course has already covered :
  - Non-Turing-complete models that can be analyzed, verified, synthesized and optimized:
    - Extended, concurrent FSMs, for control-dominated systems
    - Static Dataflow networks, for data-dominated systems

Useful modeling concepts

- What further properties would we like to have in a modeling language?
- We already mentioned expressivity, compactness, analyzability, synthesizability, executability and orthogonality
- In addition, it should enable us to express:
  - **Deterministic** concurrency, because it is almost impossible to recover from a sequential model, and implementation platforms are **highly concurrent**
  - Abstraction, to enable fast high-level modeling of very large systems
  - Timing properties, because embedded systems almost always must respond in real time
  - Modularity, to divide design among teams and companies

How do we actually use modelling methods to implement those systems ?

- So far, we have a nice mathematical theory, but how do we use it to ***optimize cost and performance of mixed HW/SW implementations of embedded data-dominated applications?***
- We need a ***cost function to optimize***
- We already found one aspect of the cost function: FIFO memory size (on-chip or off-chip RAM)
- From now on, we will assume ***SW execution on a single processor*** (we will look at HW later)
- Code size also matters (on-chip or off-chip RAM, ROM or FLASH)
- Estimating code size require hypotheses on how the code is implemented

Architectural assumptions

- We will first consider SW execution on a single ***Application-Specific Instruction set Processor (ASIP) optimized for data-dominated applications***
- Execution on multiple processors changes little about the code size discussion
- Different processor architectures change the cost function but ***do not change the design space exploration methodology***
  - Balance equations are the key to capture the full set of legal schedules
- Of course, ***multiple resources change performance*** a lot
  - We will discuss this more in the context of scheduling for ***HW***
  - Scheduling on multiple processors is like HW scheduling

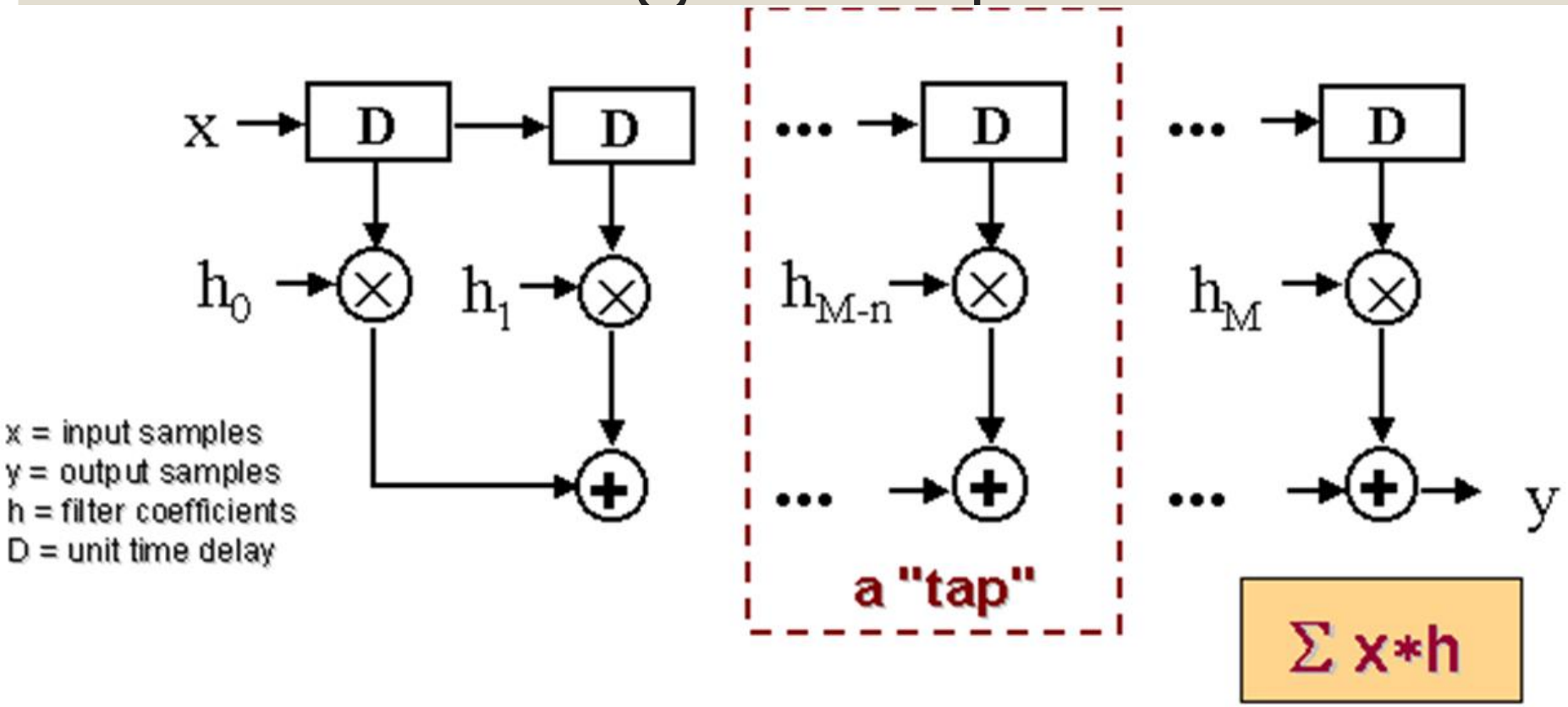
Application-Specific Instruction set Processors

- Combine some advantages of software (low NRE, ease of update) with some of hardware (performance and power)
- Customize a processor's Instruction Set Architecture (ISA) and memory architecture for a ***specific application*** (or application area)
- "Make the common case fast"
- Remove little used instructions
- Examples:
  - encryption processor (good support for shift and exor)
  - Digital Signal Processor

Digital Signal Processor (DSP)

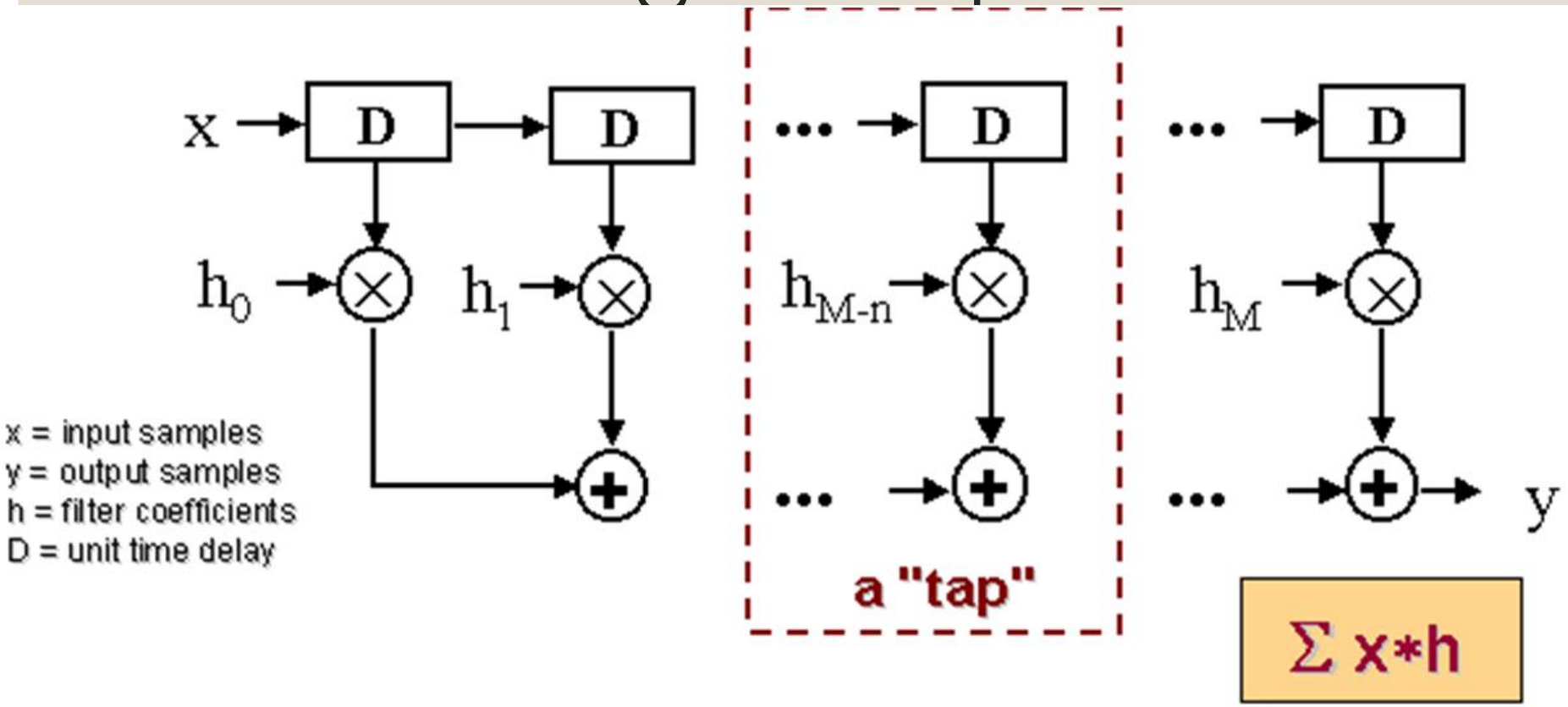
- Fully programmable (like a general-purpose CPU)
- Instruction Set Architecture and memory optimized to execute a particular class of algorithms: repetitive numerical calculations on relatively large vectors or matrices
  - Little decision making (if-then-else, data-dependent loops)
  - Lots of arithmetic operations
  - Fixed iteration count (data-independent) loops
- Examples:
  - Digital filters (Finite Impulse Response, Infinite Impulse Response)
  - Direct and Inverse Fast Fourier Transforms

A motivating example: FIR



- More taps -> higher order filter -> sharper, steeper transfer function -> more computational complexity

A motivating example: FIR



(Multiply and ACcumulate, MAC)

- Fixed iteration loop (number of taps)

Digital Signal Processor (DSP)

- Optimized for vector dot product (basis of FIR, correlation, …)
- Fast multiply and accumulate (MAC)
- Large memory bandwidth (will not discuss further)
- Fixed iteration (or infinite iteration) loops:
  - Single instruction, executed only one when loop starts (Zero-Overhead loops)
  - No pipeline flushing at the end of each iteration!
- No conditionals, i.e. no need for efficient:
  - Subroutine calls
  - Conditional jumps
- Good for real-time applications (predictable execution times)

SW architecture assumptions

- FIFOs mapped to RAM buffers, ***without memory sharing or dynamic allocation*** (fast, predictable code)
- Hardware support for zero-overhead loop
- Process code is ***inlined***, because ***subroutine calls are inefficient***:
  - Pipeline disruption
  - Cost of saving and restoring registers
- Standard code structure: "static looped code"

```
while (1) {
  for (i = 0; i < X_A; i++) { code of process A; }
  for (i = 0; i < X_B; i++) { code of process B; }
  …
}
```

Consequences of architecture assumptions

- FIFO memory size depends on the max size of each FIFO throughout the execution of the schedule
- Code size depends on the ***number of appearances of each process in looped code***
- Example: AAAABACCC

```
while (1) {
  for (i = 0; i < 4; i++) { code of process A; }
  code of process B;
  code of process A;
  for (i = 0; i < 3; i++) { code of process C; }
}
```

- Total size, ignoring cost of loop (1 instruction):
  2 * size(A) + size(B) + size(C)

Consequences of architecture assumptions

- FIFO memory size depends on the max size of each FIFO throughout the execution of the schedule
- Code size depends on the ***number of appearances of each process in looped code***
- Example: AAAABACCC (4ABA3C for short)

```
while (1) {
  for (i = 0; i < 4; i++) { code of process A; }
  code of process B;
  code of process A;
  for (i = 0; i < 3; i++) { code of process C; }
}
```

- Total size, ignoring cost of loop (1 instruction):
  2 * size(A) + size(B) + size(C)

Consequences of architecture assumptions

- Example: AAAAABCCC (5AB3C for short)

```
while (1) {
  for (i = 0; i < 5; i++) { code of process A; }
  code of process B;
  for (i = 0; i < 3; i++) { code of process C; }
}
```

- Total size, ignoring cost of loop (1 instruction):
  size(A) + size(B) + size(C)
- Smaller than before
- ***Single-Appearance Schedules (SAS) have the smallest total code size***
  - Lower bound on FLASH/ROM cost

Example of data and code size analysis

- Let us consider the first example again
- Assume that:
  - Code size(A)=128 bytes
  - Code size(B)=200 bytes
  - Code size(C)=100 bytes
  - Code size(D)=64 bytes
  - Each value in a FIFO has a size of 20 bytes
- Schedule 1: C3B4D2A
  - Total code size: 128+200+100+64=**492** bytes
  - Total data size: 27*20=**540** bytes
- Schedule 2: CABDABDB2D
  - Total code size: 2*128+3*200+100+3*64=**1148** bytes
  - Total data size: 18*20=**360** bytes

Example of data and code size analysis

- How do we compare these two schedules?
  - Schedule 1: C3B4D2A
    - Total code size: 128+200+100+64=**492** bytes
    - Total data size: 27*20=**540** bytes
  - Schedule 2: CABDABDB2D
    - Total code size: 2*128+3*200+100+3*64=**1148** bytes
    - Total data size: 18*20=**360** bytes
- ***It depends*** on the cost of code and data memory, e.g.:
  - Data RAM: 48μm^2/byte, code FLASH: 30μm^2/byte:
    - Schedule 1: 48*540+30*492= **40680** μm^2
    - Schedule 2: 48*360+30*1148=**51720** μm^2
  - Data RAM: 48μm^2/byte, code FLASH: 10μm^2/byte:
    - Schedule 1: 48*540+10*492= **30840** μm^2
    - Schedule 2: 48*360+10*1148=**28760** μm^2

How to compare schedules?

- We cannot compare these two:
  - Schedule 1:
    - Total code size: 128+200+100+64=**492** bytes
    - Total data size: 27*20=**540** bytes
  - Schedule 2:
    - Total code size: 2*128+3*200+100+3*64=**1148** bytes
    - Total data size: 18*20=**360** bytes
- And we cannot compare this hypothetical schedule:
  - Schedule 4:
    - Total code size: 2*128+2*200+100+2*64=**884** bytes
    - Total data size: 20*20=**400** bytes
  - There is no dimension along which it is the best
  - But it is not worse than another schedule in every aspect
  - There exists some RAM and FLASH area for which it ***can be best***

How to compare schedules?

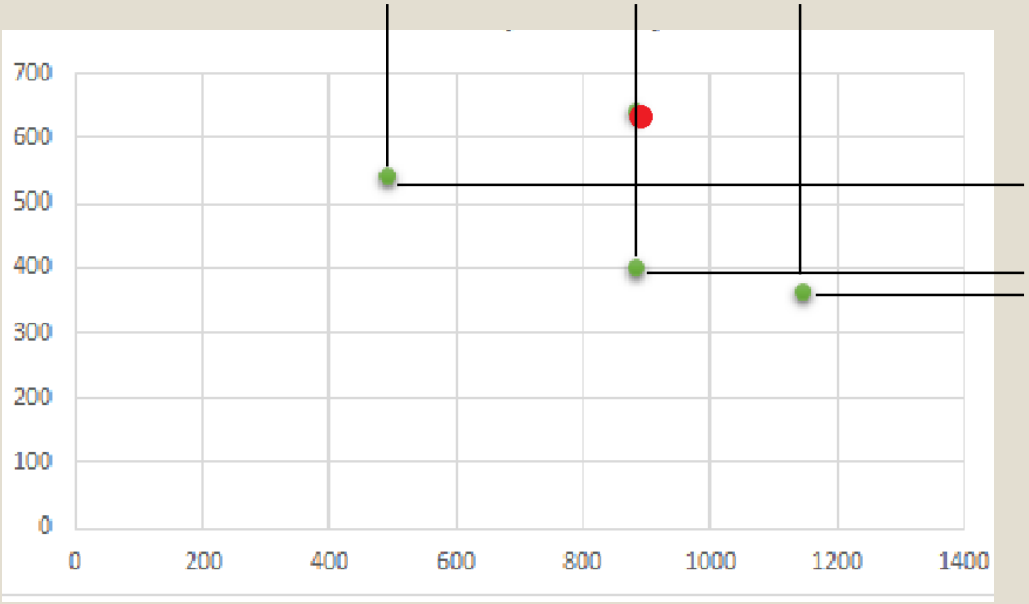
- We cannot compare these two:
  - Schedule 1:
    - Total code size: 128+200+100+64=**492** bytes
    - Total data size: 27*20=**540** bytes
  - Schedule 2:
    - Total code size: 2*128+3*200+100+3*64=**1148** bytes
    - Total data size: 18*20=**360** bytes
- But we can compare this hypothetical schedule:
  - Schedule 5:
    - Total code size: 2*128+2*200+100+2*64=**884** bytes
    - Total data size: 32*20=**640** bytes
  - It is worse than schedule 1 in every aspect
  - There exists **no** RAM and FLASH area for which it can be best

Outline

- **Static Dataflow Networks**
  - Single-resource scheduling (SW)
  - Architectural assumptions: Digital Signal Processors
  - **Pareto optimality**
- Boolean Dataflow Networks
- Multi-resource scheduling (HW and SW)
  - List scheduling
  - Integer Linear Programming scheduling

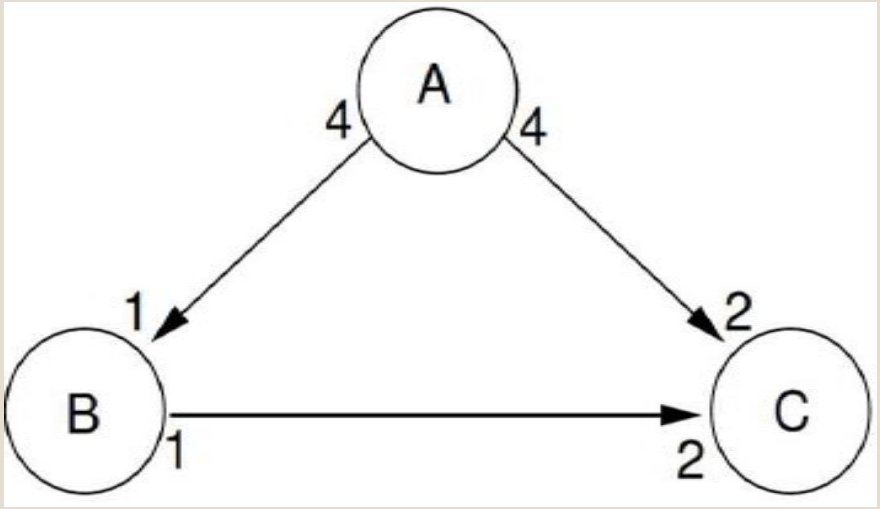
Pareto optimality

- Vilifredo Pareto (economics) found a solution:
  - **Pareto-dominated** solution: it is worse than some other solution in every aspect
  - **Pareto-optimal** solution: not dominated by any other



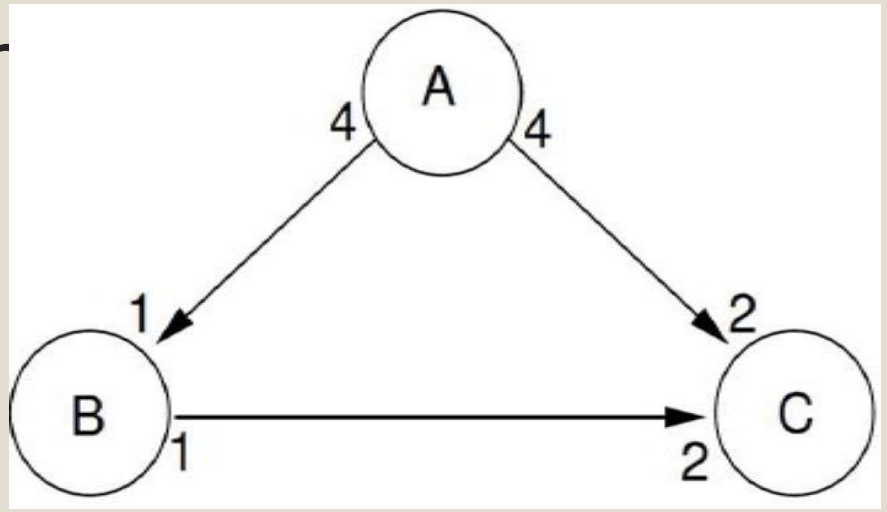
- We need to consider only (and all…) Pareto optimal solutions

Another example



- A is "input process"
- C is "output process"
- Write down the balance equations
- Find any SAS (starting from zero initial tokens, since this graph is acyclic)
- Compute its code memory size, assuming that:
  - Size(A) = 100 bytes
  - Size(B) = 250 bytes
  - Size(C) = 200 bytes
- Compute its data memory size, assuming that each value requires 32 bytes to be stored

Another exam



- Compare the data memory sizes of these two SAS (again assuming 32 bytes per value):
  - A4B2C
  - A2(2BC), with this code structure:

```
while (1) {
   code of process A;
   for (i = 0; i < 2; i++) {
      for (i = 0; i < 2; i++) { code of process B; }
      code of process C;
   }
}
```

How about performance?

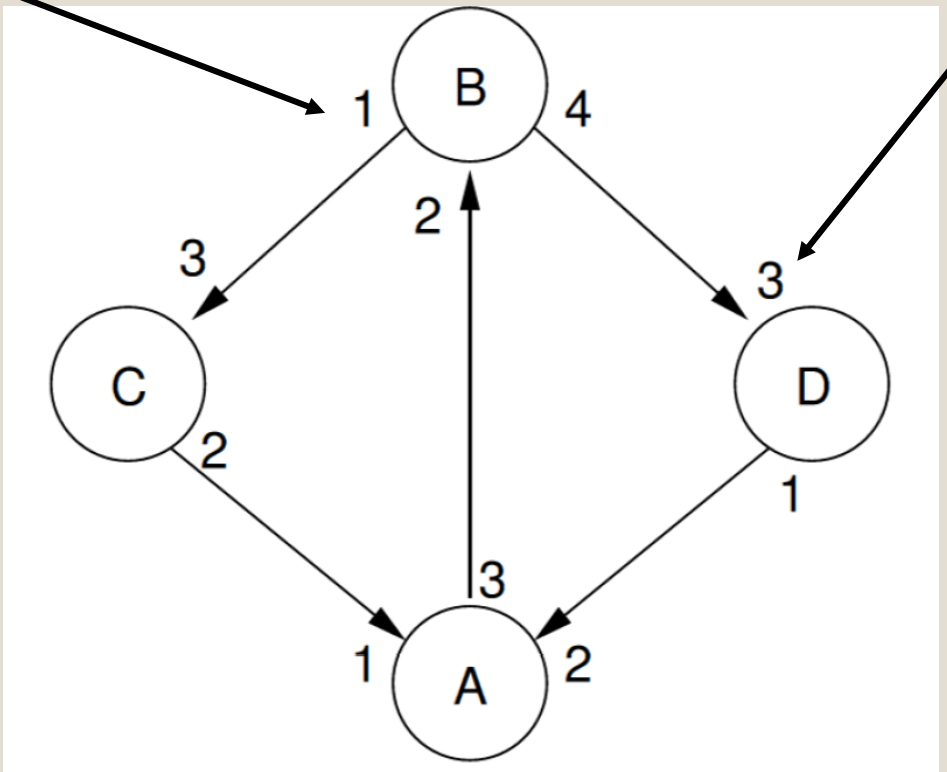
- In DSPs, instructions are pipelined
- In any valid schedule, each process is executed exactly the same number of times for a given amount of work
- Example, consider a minimal schedule (X_C=1) and a non-minimal schedule (X_C=2):
  - C3B4D2A: total time: exec(C)+3*exec(B)+4*exec(D)+2*exec(A)
  - 2C6B8D4A: total time ***for twice the amount of work***: 2*exec(C)+6*exec(B)+8*exec(D)+4*exec(A)
- Hence under our assumptions ***all valid schedules require the same amount of time per unit of work***

How about performance?

- Assume that code is stored in a ***slow large external memory*** and brought into ***fast on-chip memory for execution***
- Assume also that fast memory can contain only code of one process
- Compare the two previous schedules ***for the same amount of work***, i.e. two executions of C, and so on:
  - C3B4D2A: each process is loaded from external slow memory ***twice***
  - 2C6B8D4A: each process is loaded from external slow memory ***once***
- Hence the second schedule is ***faster assuming small on-chip code memory***
- It also requires ***twice as much data memory***

An example of SDF network

- Circles denote processes
- Each input or output shows the number of read or written tokens (FIFOs not shown)



Analyzing SDF

- For now, we focus on ***sequential scheduling***
- We must find a **finite sequence** of process firings (called a **schedule**) that can be **repeated infinitely**
- Key property: when the schedule **ends**, it must leave the SDF with **the same number of tokens** in each queue as there was **at the beginning**
- Then it can be repeated without overflow or underflow
- We will call this a **valid SDF schedule**
  - Each SDF networks can have
    - Either **no valid schedule** (if it is **non-schedulable**)
    - Or an **infinite number of valid schedules**
  - We will later explore that space to find a schedule that **optimizes some cost function**
    - Performance, memory size, power, ….

Analyzing SDF

- The infinite executability in finite memory problem is thus replaced by **schedulability** for SDF
- Let us call
  - x_A the number of times process A fires in such a schedule
  - w_{Ai} the number of tokens that process A writes to FIFO i on each firing
  - r_{Bj} the number of tokens that process B reads from FIFO j on each firing
- Then each FIFO k in the SDF (each edge in the graph) between processes A and B must satisfy the **balance equation:**

$$w_{Ak} * x_A = r_{Bk} * x_B$$

Analyzing SDF

- Since the balance equation must be satisfied for all FIFOs, we get a system of **linear homogeneous** equations that must be satisfied by the number of firings of each process in each valid schedule
- For example:

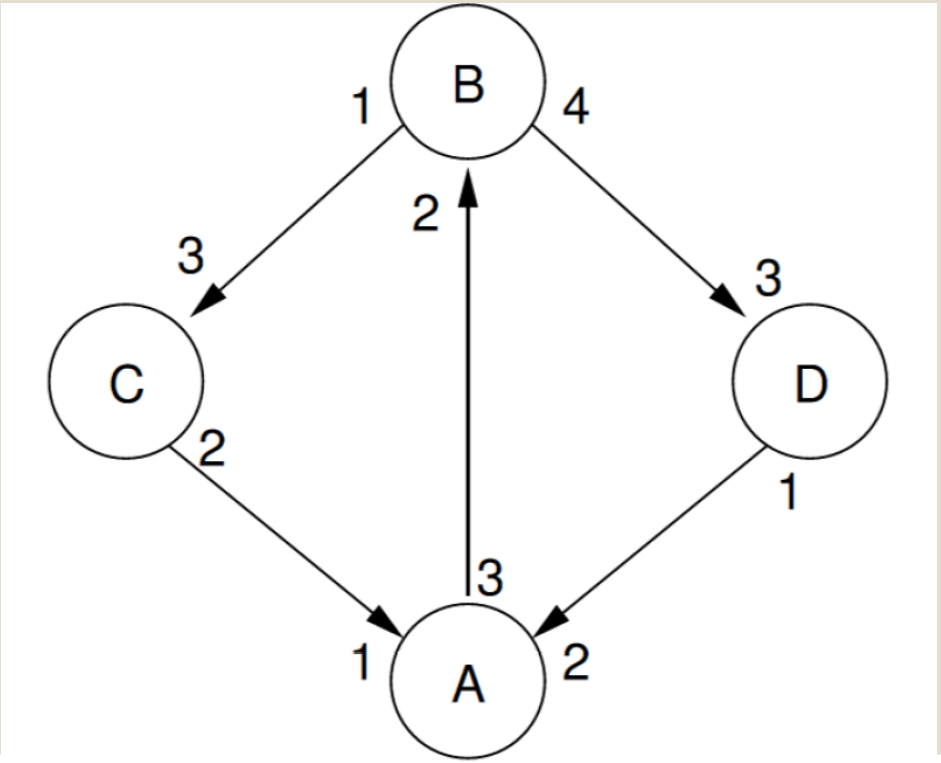
$$1\ x_B = 3\ x_C$$

$$4\ x_B = 3\ x_D$$

$$1\ x_D = 2\ x_A$$

$$3\ x_A = 2\ x_B$$

$$2\ x_C = 1\ x_A$$



EC

Analyzing SDF

- Linear homogeneous equations admit:
  - Always the all-zero solution vector
    - Not interesting: you never overflow if you do not do anything
  - Non-zero solution vectors if the system has fewer linearly independent equations than unknowns
- It can be shown that for a connected SDF network with N processes, there are always either N or N-1 linearly independent equations (rank of the coefficient matrix)
  - N: only the zero vector: **non-schedulable**
  - N-1: infinite family of vectors with 1 generator: **schedulable**

Analyzing SDF

- In our example, we can rewrite the system:

$1\ x_B = 3\ x_C$

$4\ x_B = 3\ x_D$

$1\ x_D = 2\ x_A$

$3\ x_A = 2\ x_B$

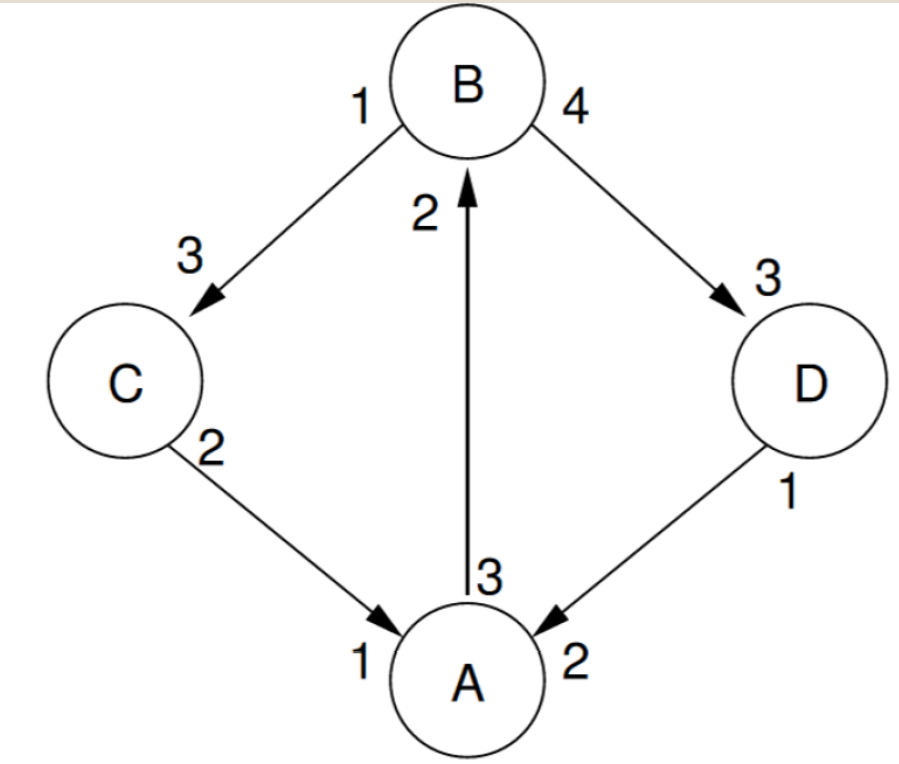
$2\ x_C = 1\ x_A$

as:

$1\ x_B = 3\ x_C$

$1\ x_A = 2\ x_C$

$1\ x_D = 4\ x_C$



Analyzing SDF

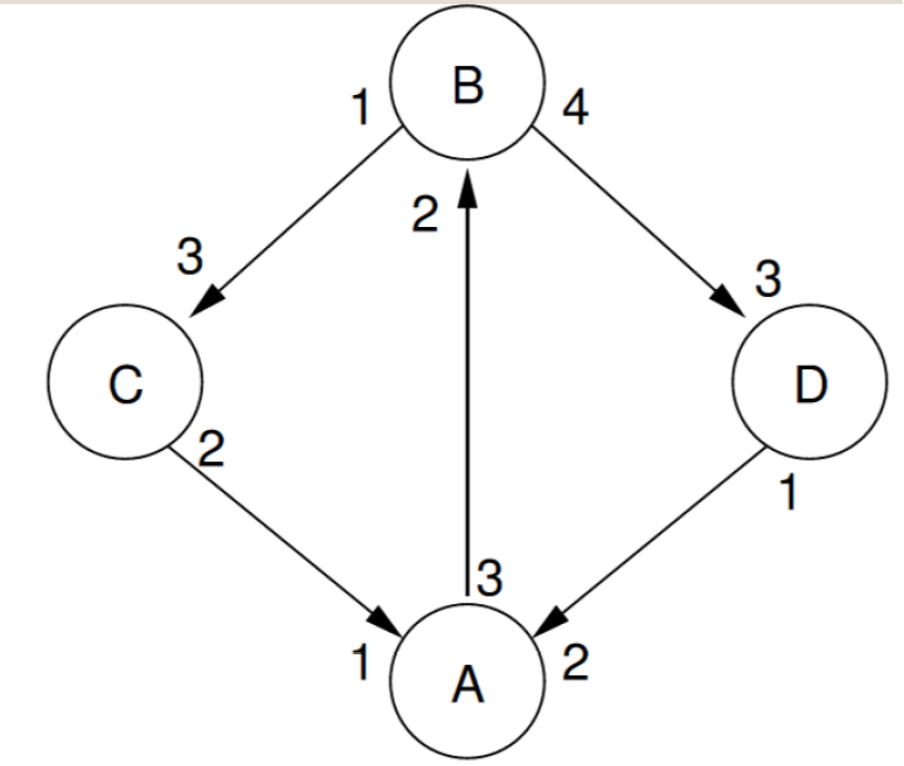
- The system has rank 3, with 4 unknowns: infinite set of non-zero solutions, depending on one parameter

$1\ x_B = 3\ x_C$

$1\ x_A = 2\ x_C$

$1\ x_D = 4\ x_C$

- We are interested in ***non-zero integer solutions*** (we cannot run a process ½ a time...)
- The solutions to this set of equations defines the ***legal solution space***



Analyzing SDF

- For example, let us set $\mathbf{x_C = 1}$

$\mathbf{1\ x_B = 3\ x_C}$

$\mathbf{1\ x_A = 2\ x_C}$

$\mathbf{1\ x_D = 4\ x_C}$

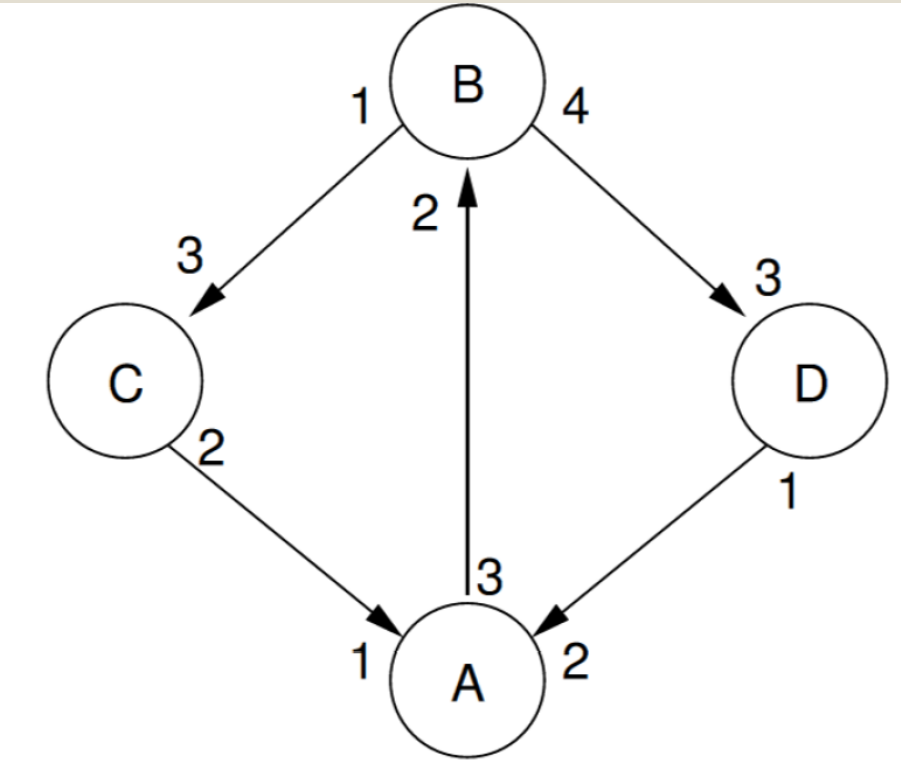
- Minimal non-zero solution:

$\mathbf{x_B = 3}$

$\mathbf{x_A = 2}$

$\mathbf{x_D = 4}$

- Possible executions:
  - CBBBDDDDAA
  - CABDABDBDD
  - BBBCDDDDAA



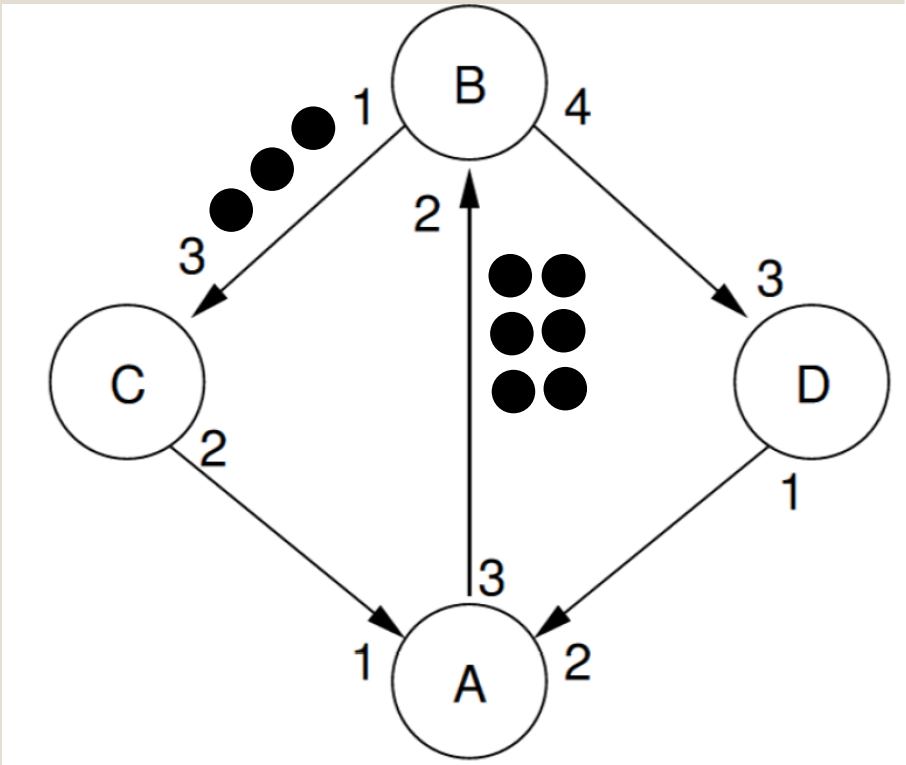
Choosing a schedule

Key theorems (by Lee and Messerschmitt)

- Theorem 1: an SDF network admits one (actually an infinite family of) infinite executions that do ***not overflow and do not deadlock if and only if the set of balance equations has non-zero solutions***
- Theorem 2: the set of schedules is the set of ***executions that satisfy the balance equations***
- Theorem 3: for any schedule that satisfies the balance equations, ***there exists an initial assignment of values to FIFOs*** (also called "initial marking") ***that enables it***

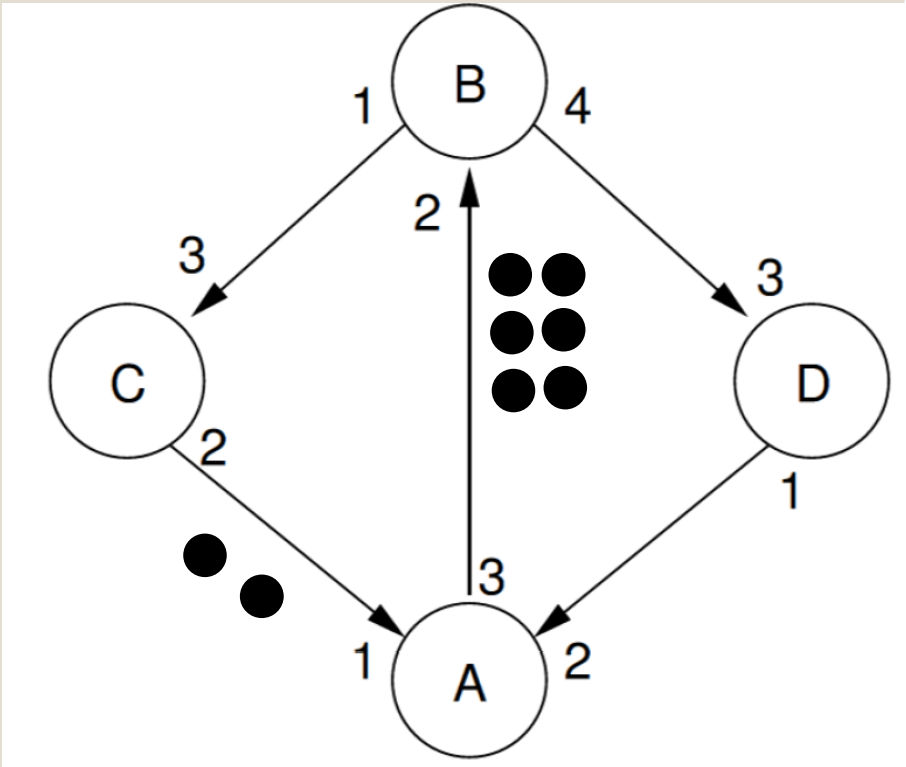
Example: schedule 1

- **C**BBBDDDDAA
- Current max FIFO size: 6+3



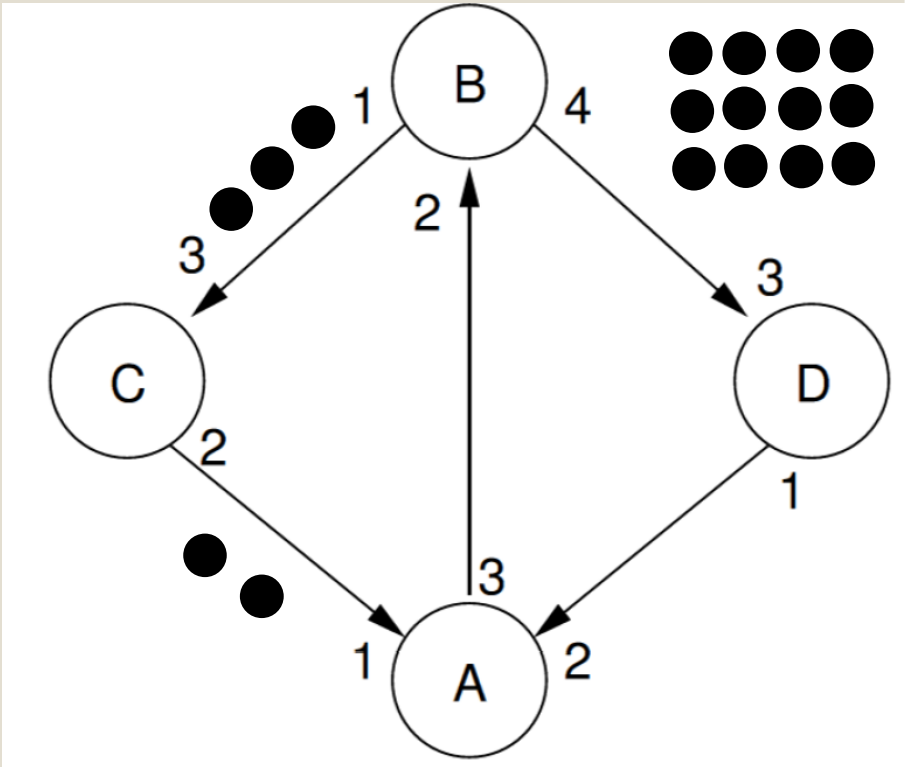
Example: schedule 1

- C**BBB**DDDDAA
- Current max FIFO size: 6+3+2



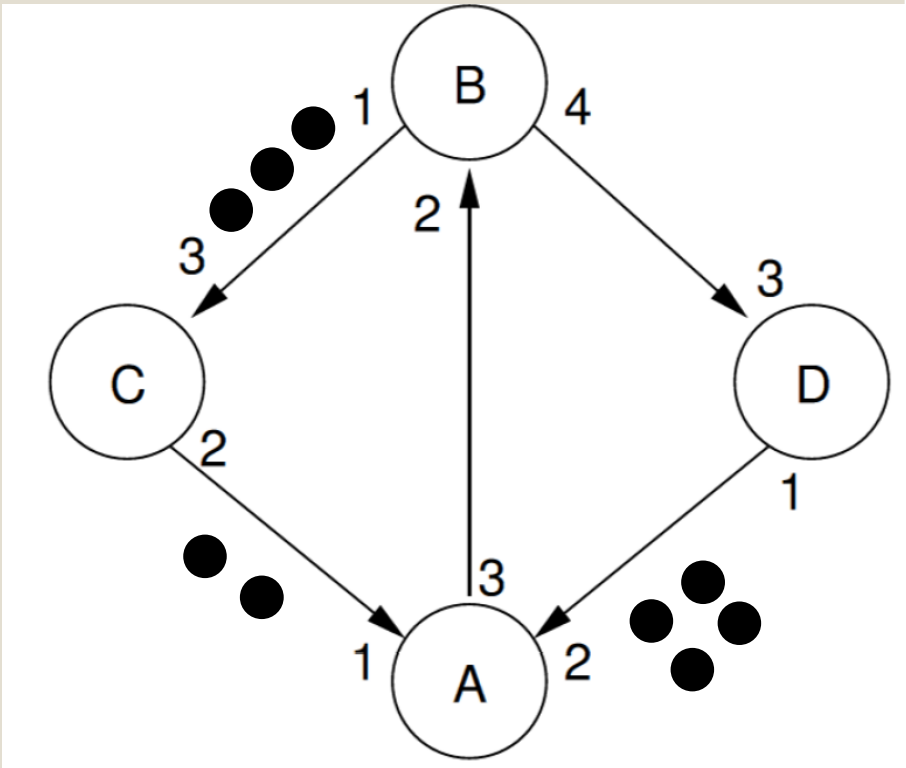
Example: schedule 1

- CBBB**DDDD**AA
- Current max FIFO size: 6+3+2+12



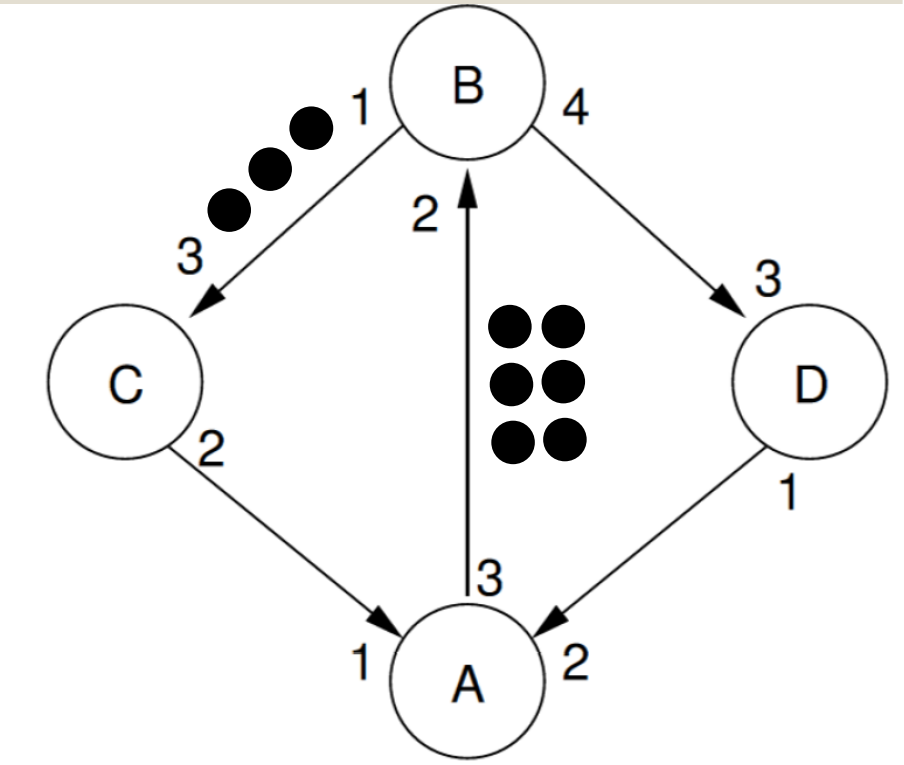
Example: schedule 1

- CBBBDDDD**AA**
- Current max FIFO size: 6+3+2+12+4



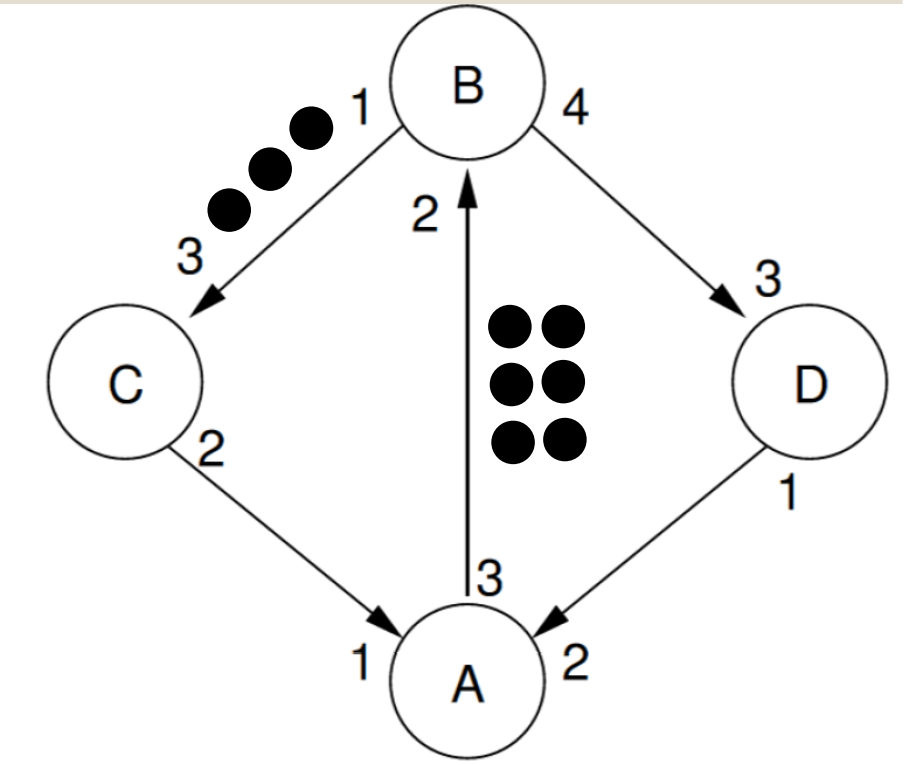
Example: schedule 1

- **C**BBBDDDDAA
- Look, ma!! Back to the initial state ☺ ☺
- Total FIFO size: 6+3+2+12+4=27 va
  (assuming no memory
  sharing among FIFOs)



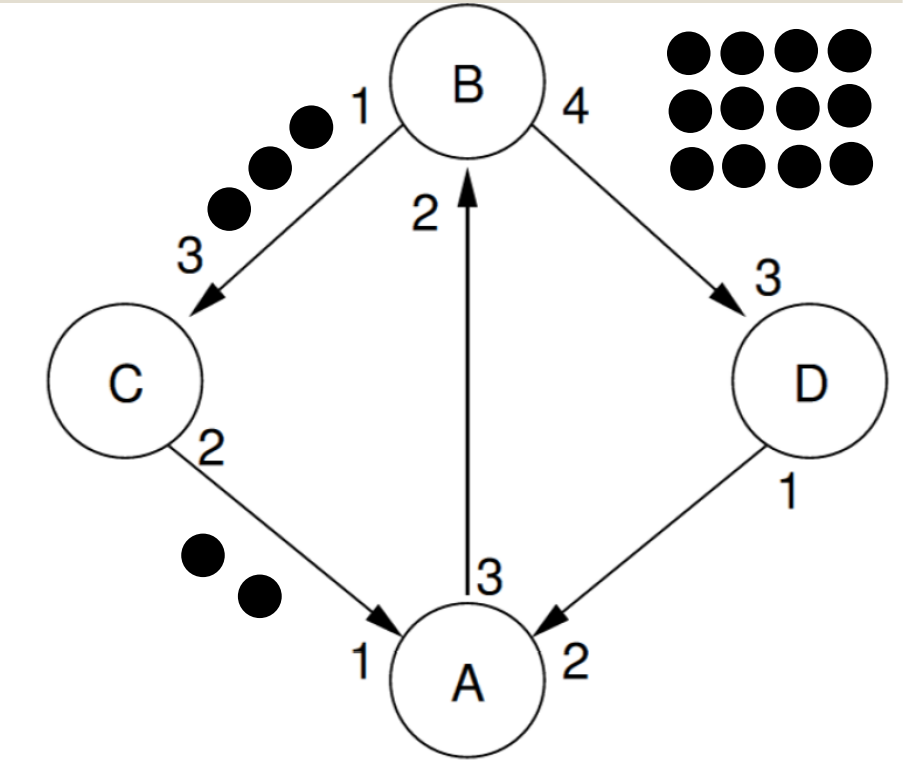
How about concurrent execution?

- **CBBB**DDDDAA
- Nothing really changes with concurrent execution
- **Schedulable networks remain schedulable, and non-schedulable ones remain non-schedulable**



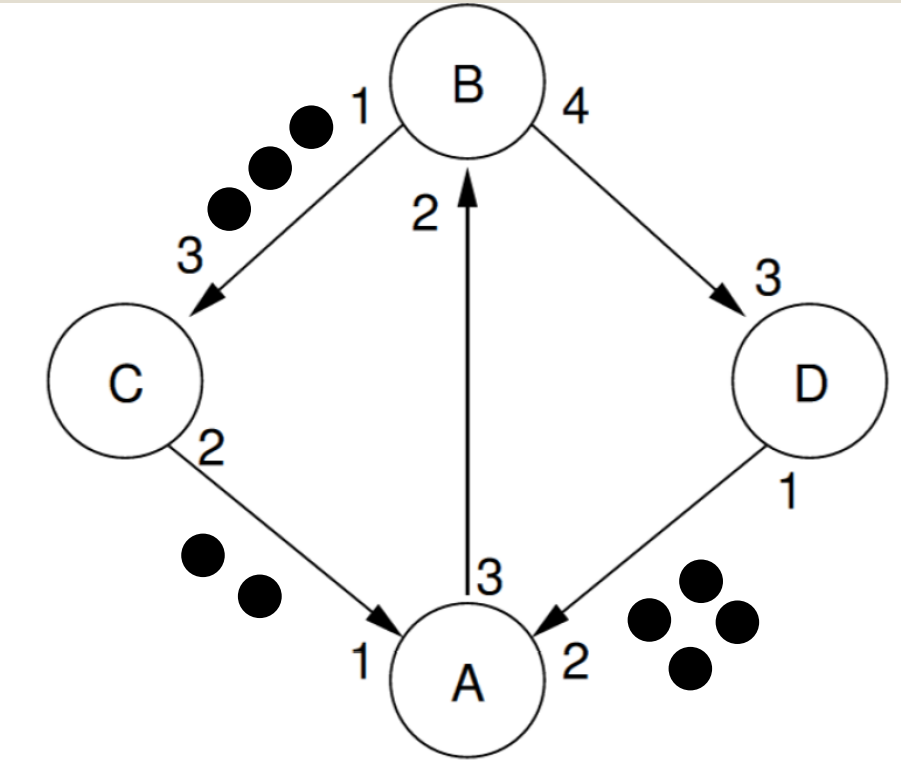
How about concurrent execution?

- CBBB**DDDD**AA
- Note: ***we cannot execute C again yet!!***
- We must reach the end of the sch



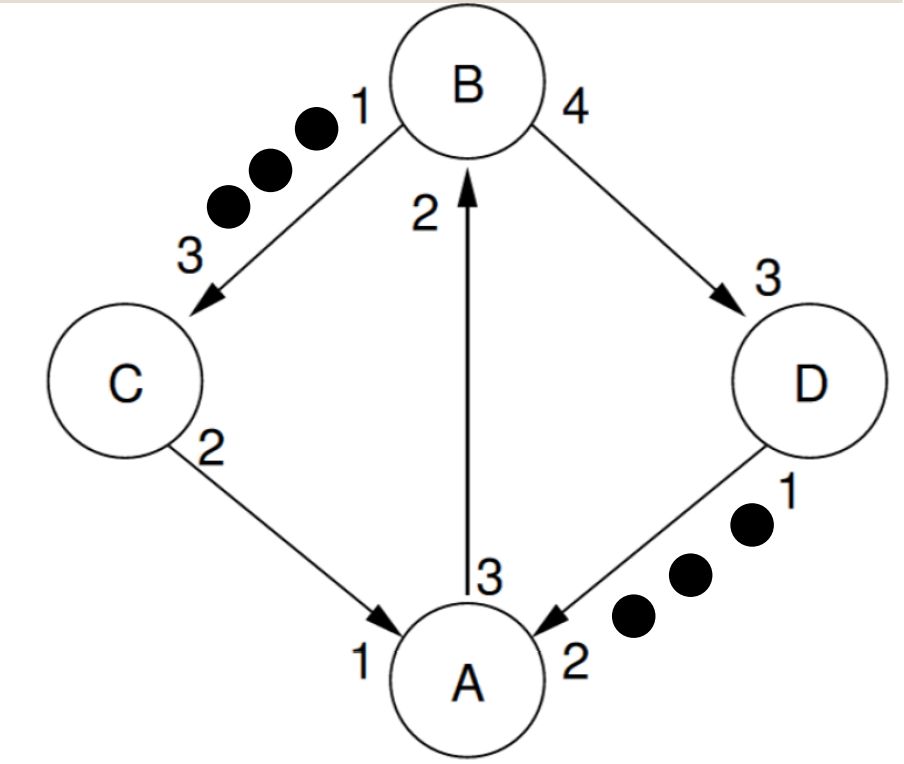
How about concurrent execution?

- CBBBDDDD**AA**
- Same final marking as before…
- We could even alternate betwee
  sequential schedules,
  as long as they all:
  - Satisfy the balance equations
  - They can all be started from the same initial marking



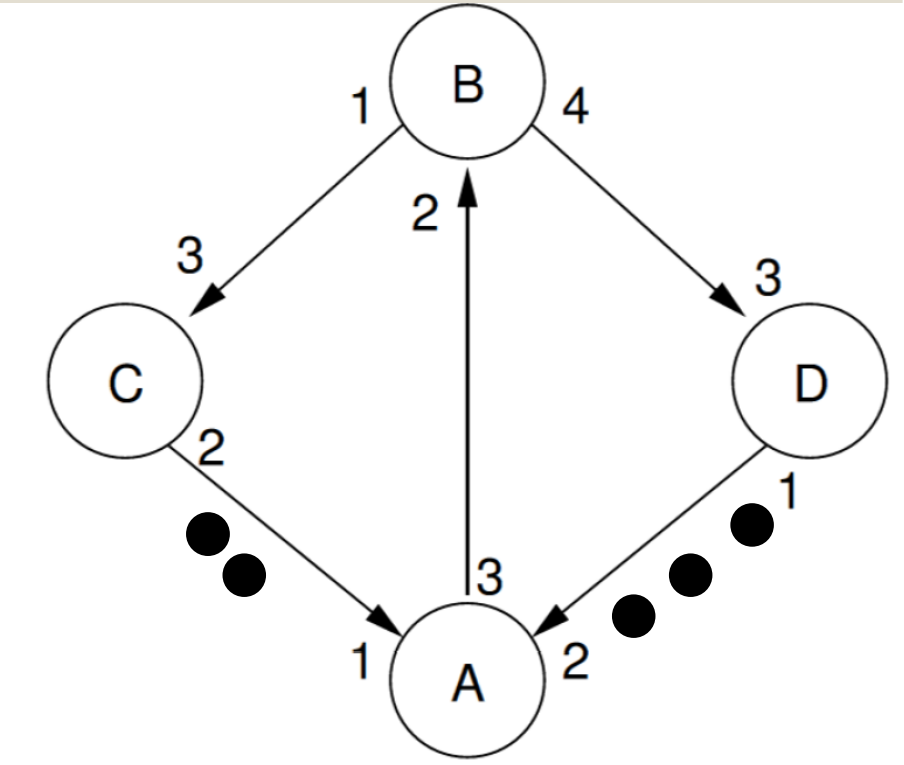
Example: schedule 2

- **C**ABDABDBDD
- Current max FIFO size: 3+3



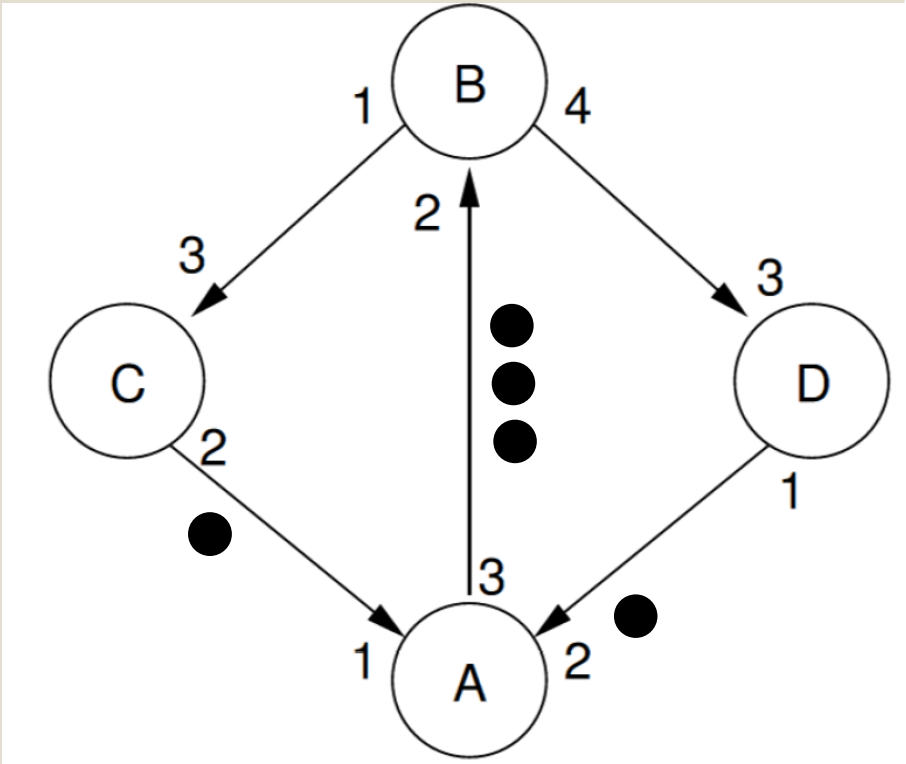
Example: schedule 2

- C**A**BDABDBDD
- Current max FIFO size: 3+3+2



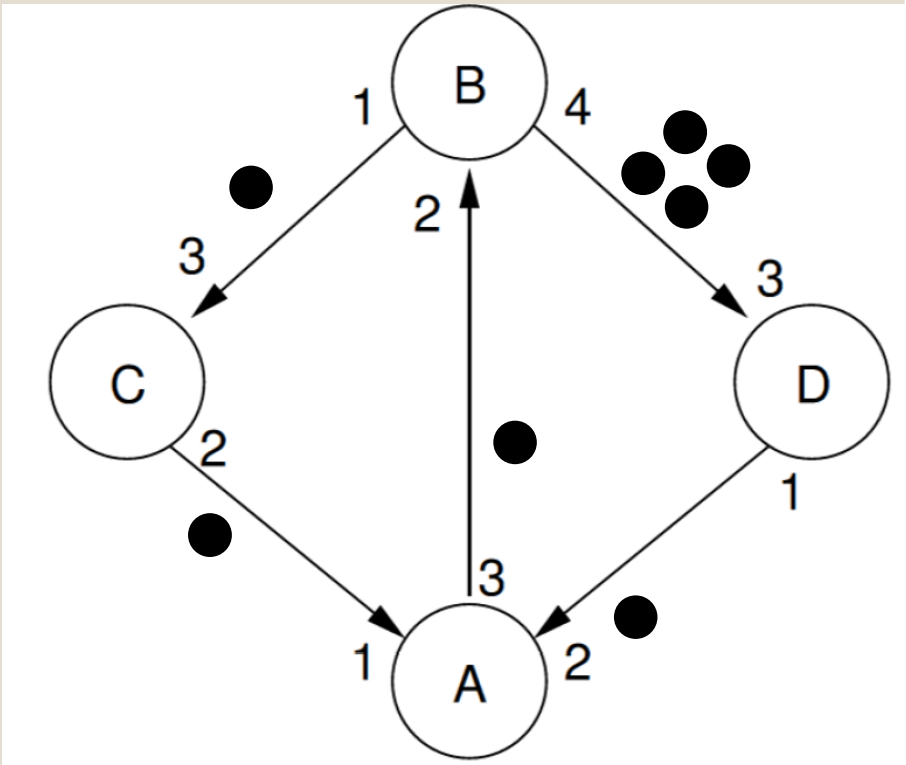
Example: schedule 2

- CA**B**DABDBDD
- Current max FIFO size: 3+3+2+3



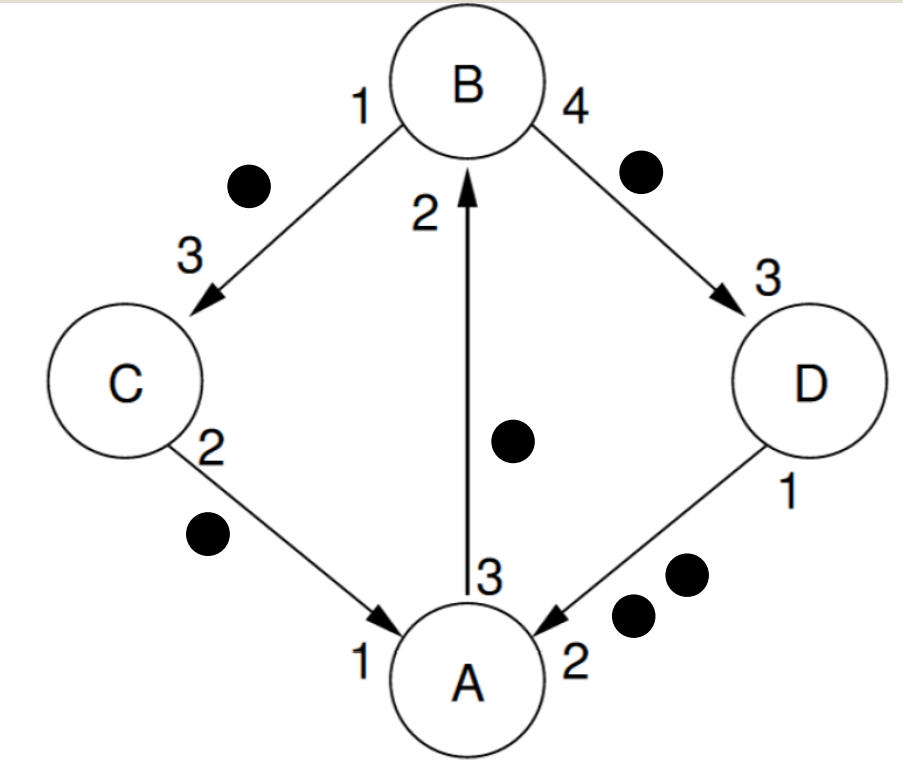
Example: schedule 2

- CAB**D**ABDBDD
- Current max FIFO size: 3+3+2+3+4



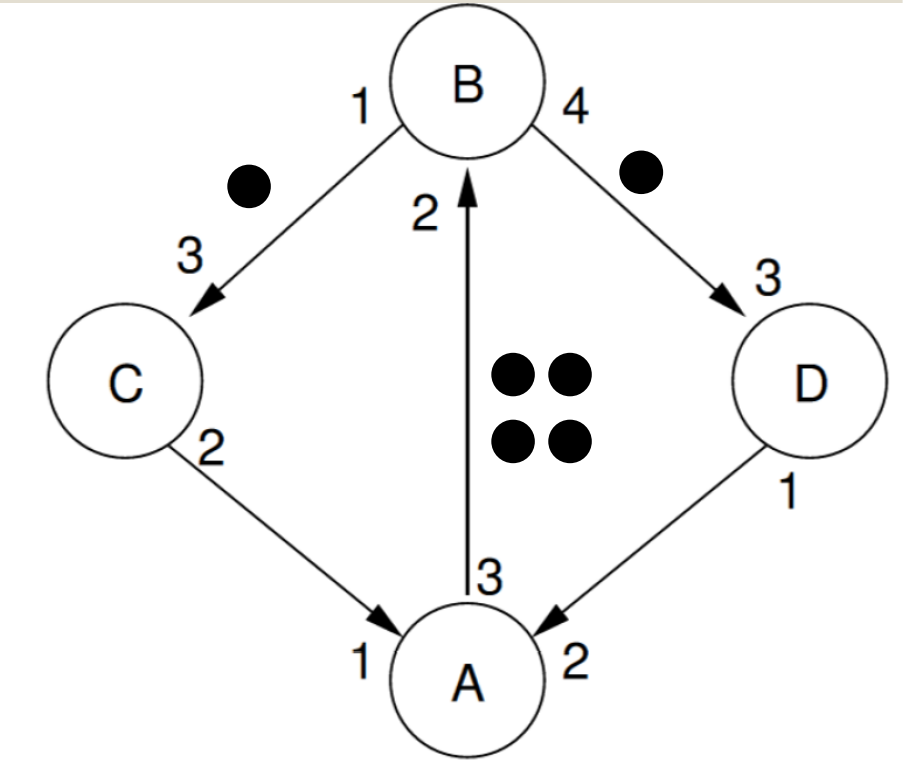
Example: schedule 2

- CABD**A**BDBDD
- Current max FIFO size: 3+3+2+3+4



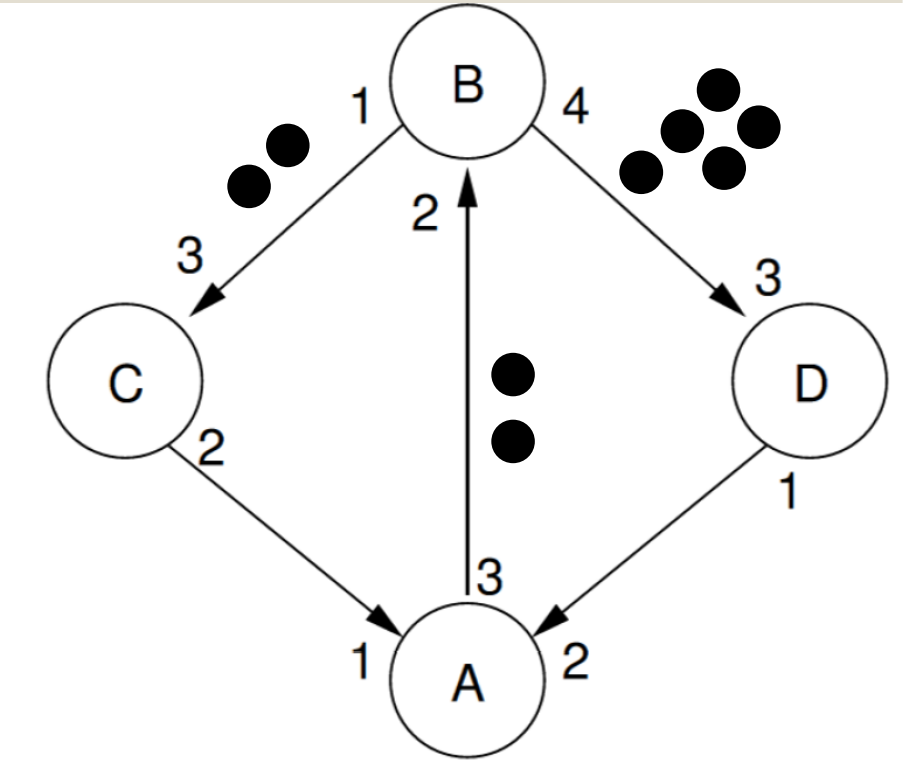
Example: schedule 2

- CABDA**B**DBDD
- Current max FIFO size: 3+3+2+4+4



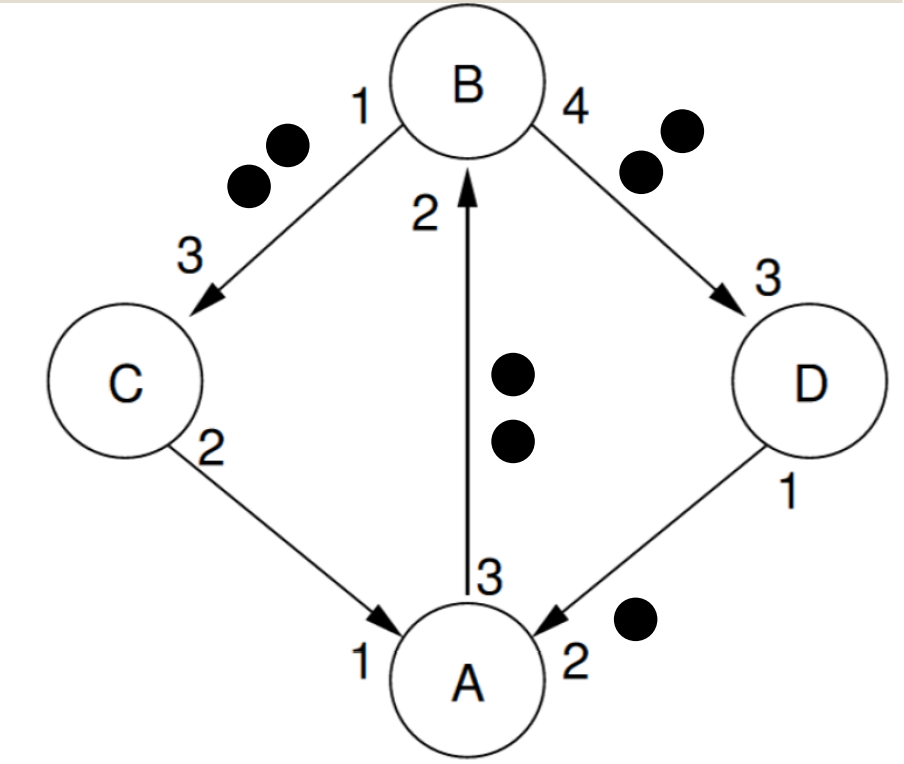
Example: schedule 2

- CABDAB**D**BDD
- Current max FIFO size: 3+3+2+4+5



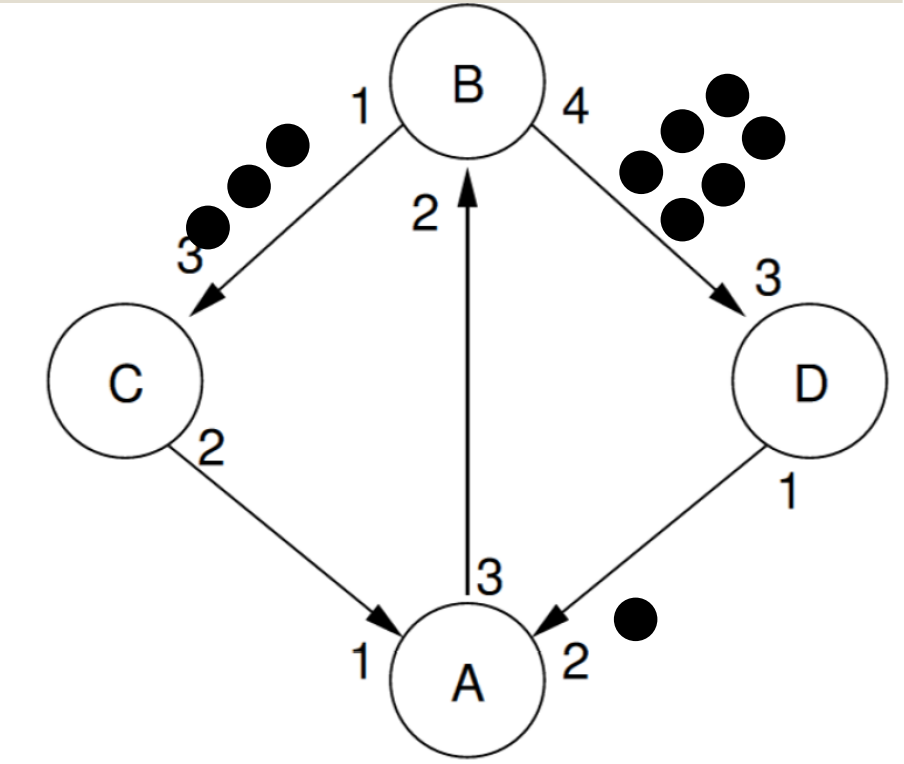
Example: schedule 2

- CABDABD**B**DD
- Current max FIFO size: 3+3+2+4+5



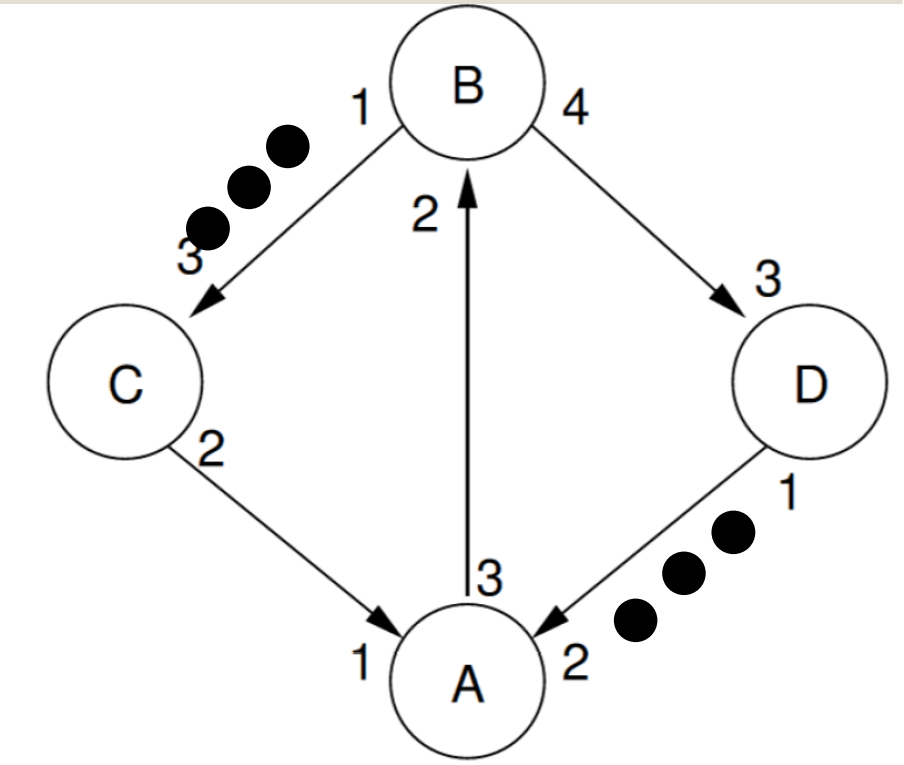
Example: schedule 2

- CABDABDB**DD**
- Current max FIFO size: 3+3+2+4+6



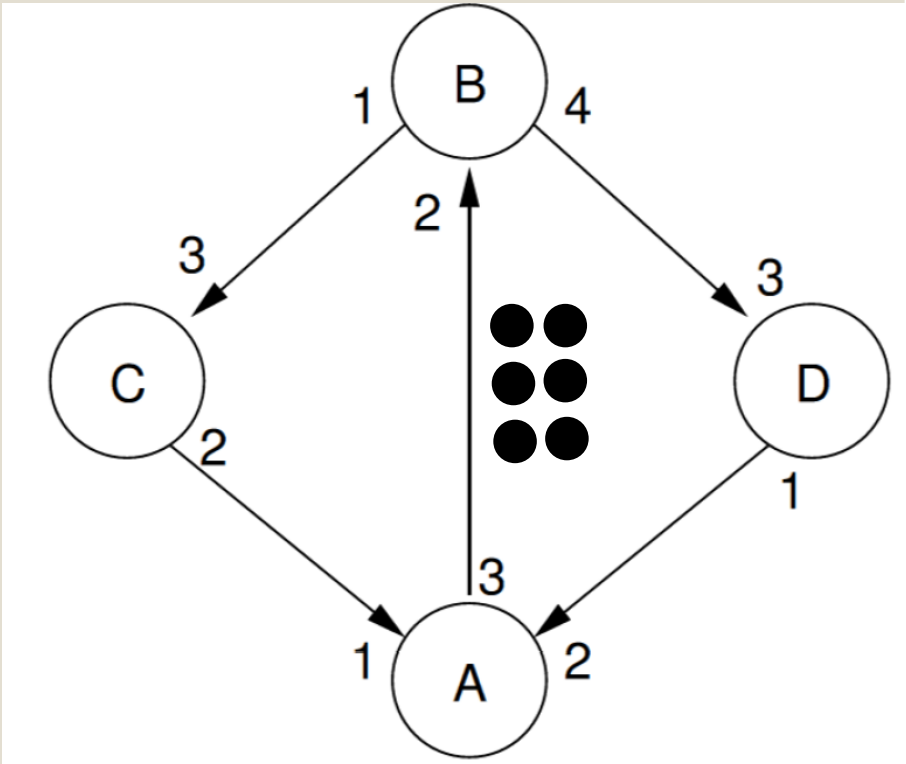
Example: schedule 2

- **C**ABDABDBDD
- Back to the initial marking (not unexpectedly…)
- Total FIFO size: 3+3+2+4+6=**18** valu
- Smaller than before!!
  (it was 27)
- What does this mean?
- **Smaller data storage!**



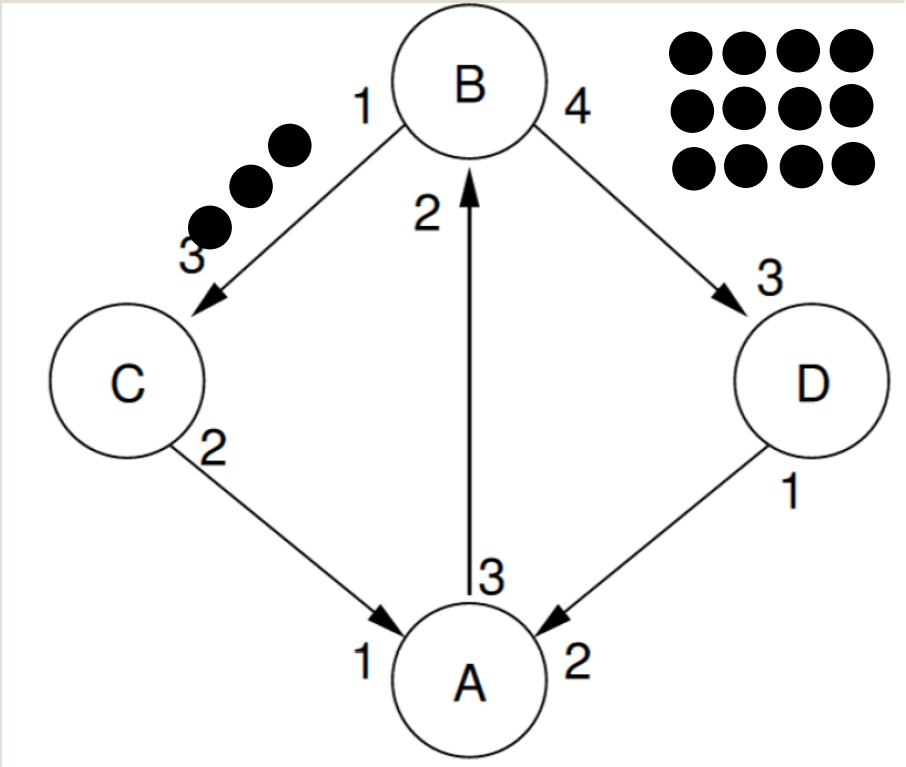
Example: schedule 3

- **BBB**CDDDDAA
- Current max FIFO size: 6



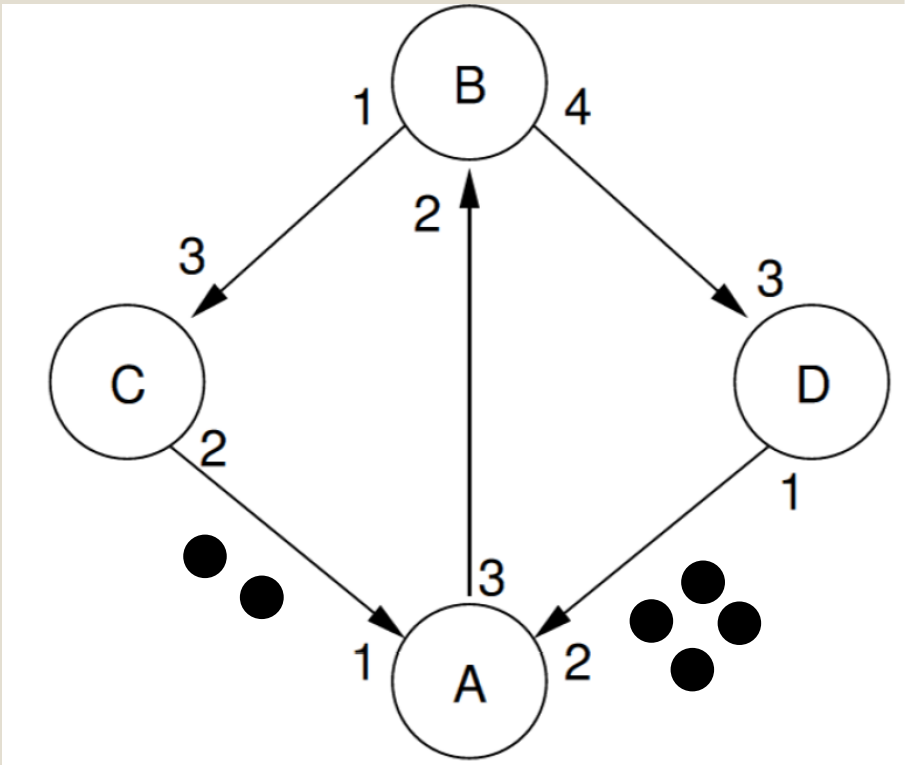
Example: schedule 3

- BBB**CDDDD**AA
- Current max FIFO size: 6+3+12



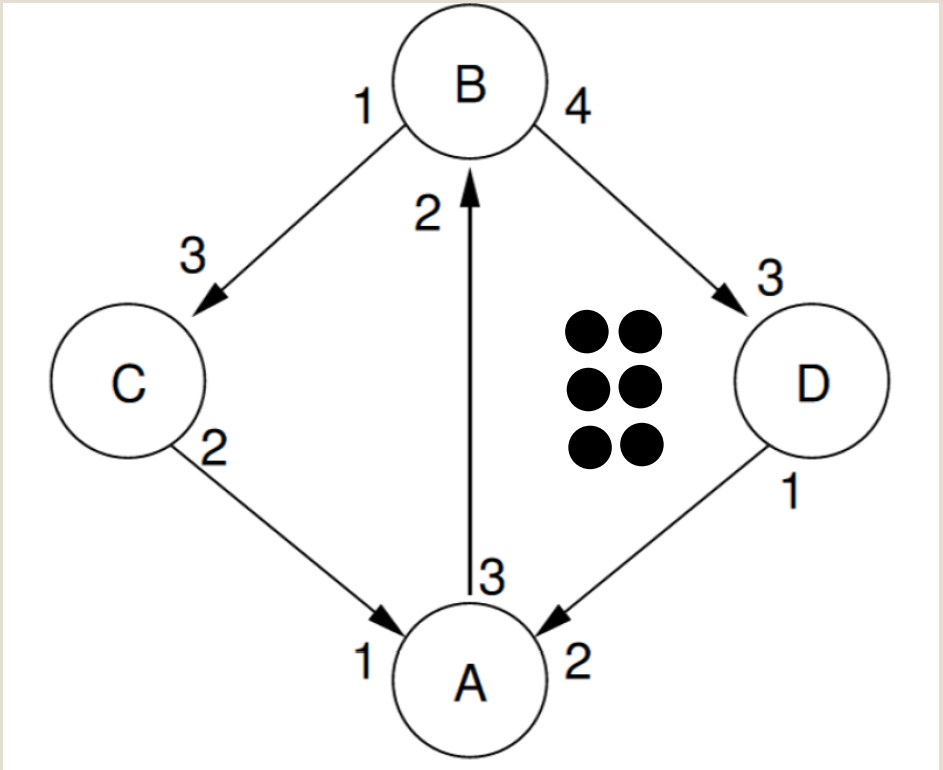
Example: schedule 3

- BBBCDDDD**AA**
- Current max FIFO size: 6+3+12+2+4



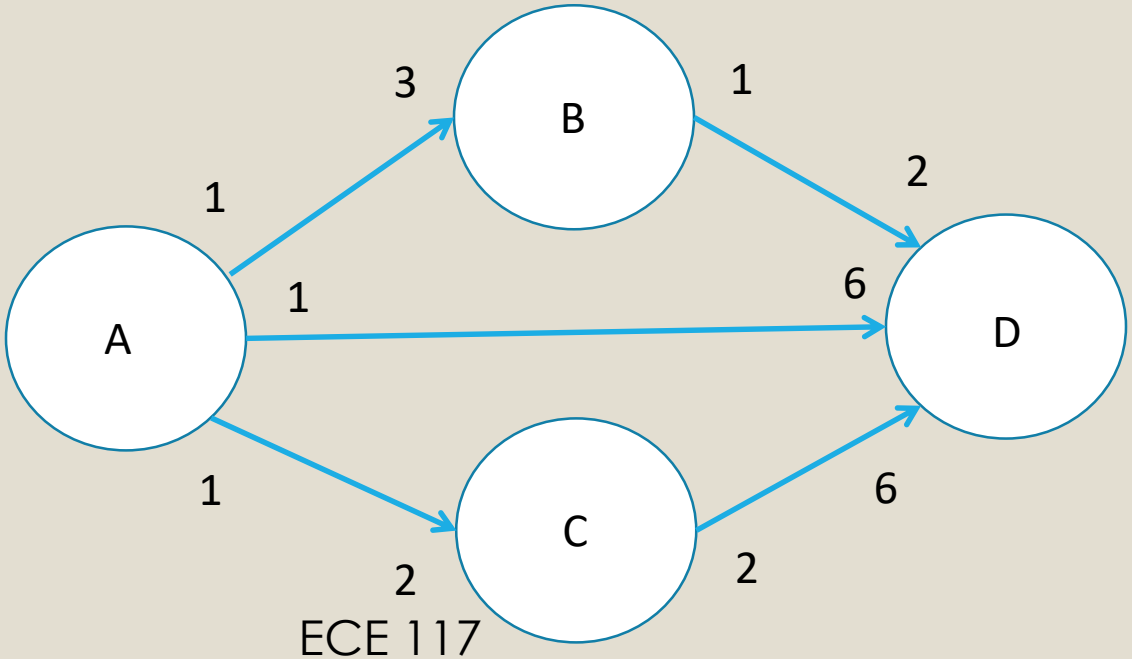
Example: schedule 3

- **BBB**CDDDDAA
- Total FIFO size: 6+3+12+2+4=27 values
- Is it a surprise? ***No!***
- The first schedule was: CBBBDDDDAA
- Both execute each process in a "single burst"
- Both produce the same number of values on each FIFO…



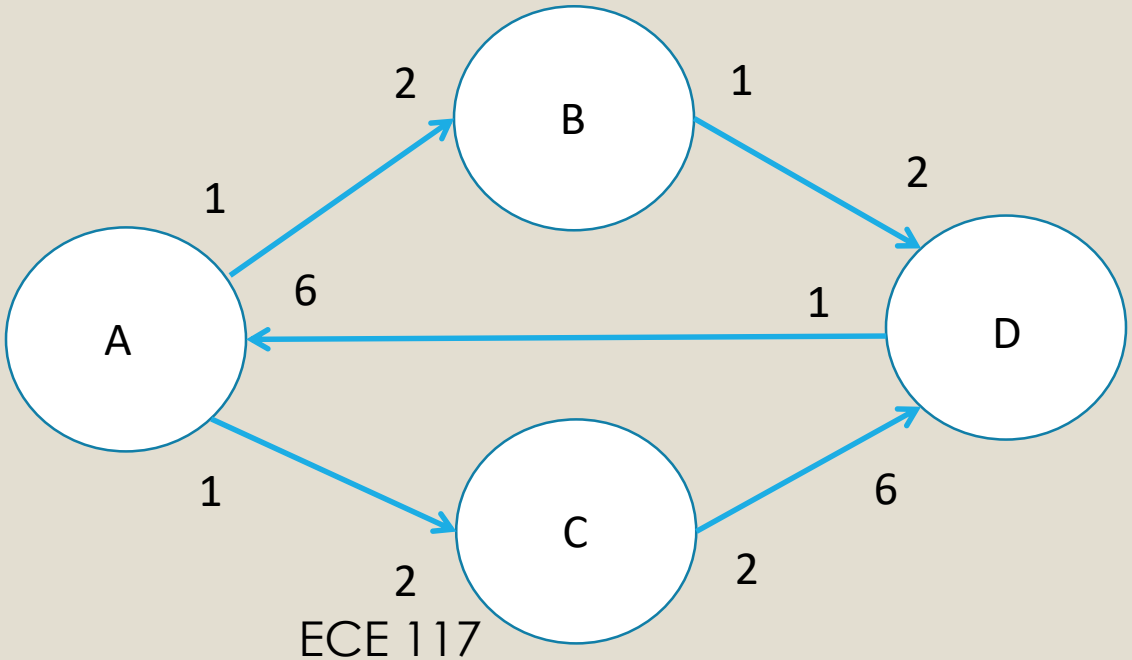
Exercise 1

- Schedule for minimum code size and assign minimum number of initial tokens
- Compute code size and data size assuming that each process is 1000 bytes and each value is 128 bytes



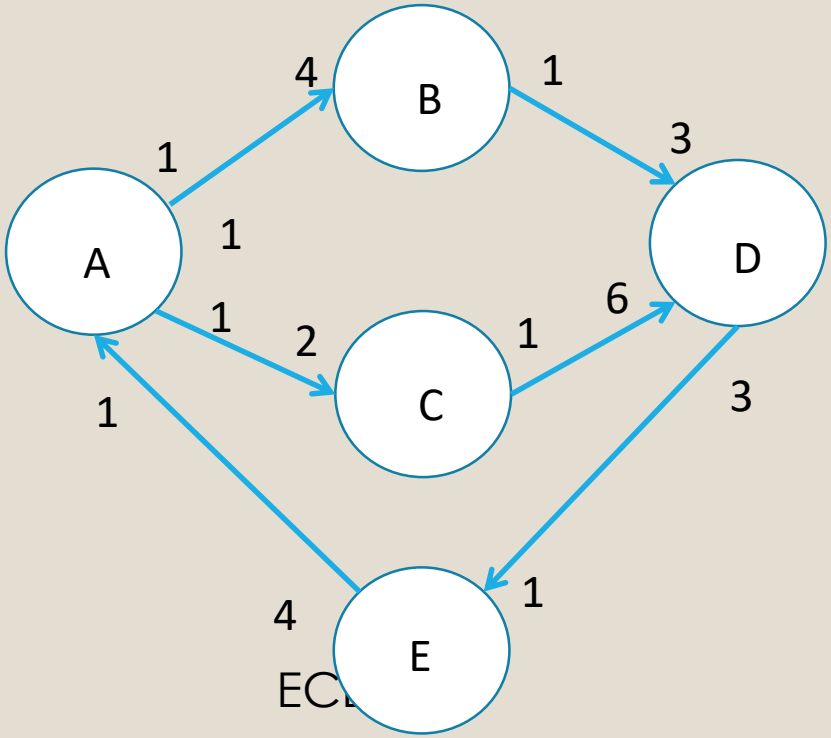
Exercise 2

- Check if this network is schedulable in finite memory
- If not, change 1 or 2 production or consumption rates to make it schedulable.



Exercise 3

- Schedule for minimum code size and assign minimum number of initial tokens
- Compute code size and data size assuming that each process is 1000 bytes and each value is 128 bytes



In summary…

- SDF networks allow us to compute the ***full*** (infinite…) ***set of valid schedules***
- We can compare various schedules with respect to:
  - Code size
  - Data size
  - Performance (actually, slow memory access time)
  - Power (not discussed here)
  - …
- SAS schedules optimize code size
- Schedules that execute processes "as soon as possible" (when they have enough input tokens) help minimize data size

In summary…

- Various algorithms have been devised to explore various solutions that optimize, under various architectural assumptions:
  - Code size
  - Data size
  - Execution time
  - Power and energy
- We need to consider only Pareto-optimal solutions
- Exploring all of them is too expensive ("combinatorial explosion")

Outline

- Static Dataflow Networks
  - Single-resource scheduling (SW)
  - Architectural assumptions: Digital Signal Processors
  - Pareto optimality
- **Boolean Dataflow Networks**
- Multi-resource scheduling (HW and SW)
  - List scheduling
  - Integer Linear Programming scheduling

Boolean Data Flow (BDF)

- SDF is very nice, but… it is sometimes too restrictive: ***no data-dependent value production or consumption***
- Can we expand the expressive power to include:
  - Conditional execution of processes?
  - Data-dependent iteration of processes?
- The answer is "yes and no"
  - SDF is the most expressive model for which schedulability in finite memory is decidable
  - There is a more expressive model (Boolean Data Flow) for which ***schedulability is decidable in many practically interesting cases***

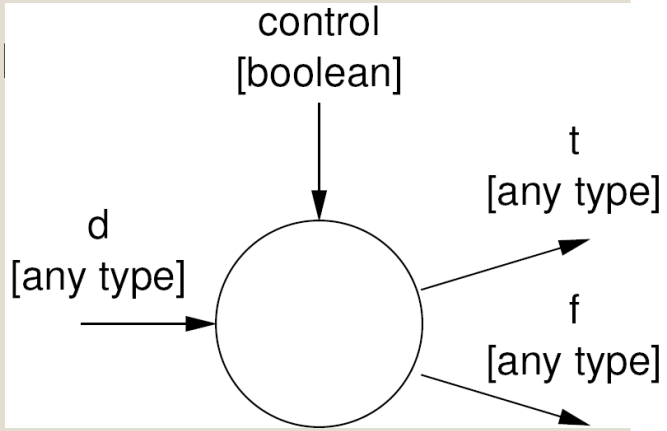
Boolean Data Flow (BDF)

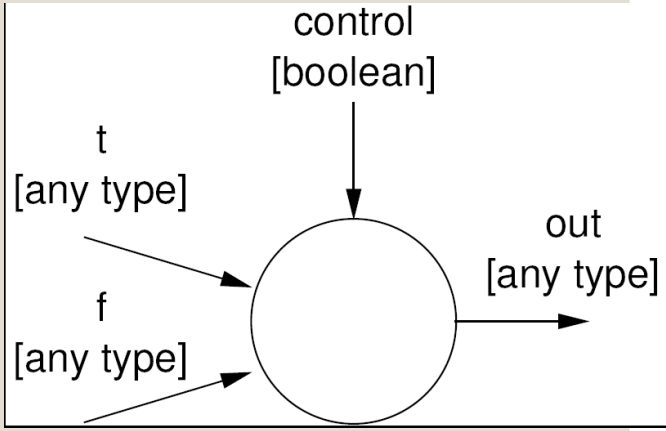
- Minimal extension of SDF with two additional
  - ***Conditional split***:

```
while (1) {
  if (control.read())
       t.write(d.read())
  else f.write(d.read())
}
```

  - ***Conditional merge***:

```
while (1) {
  if (control.read())
       out.write(t.read())
  else out.write(f.read())
}
```



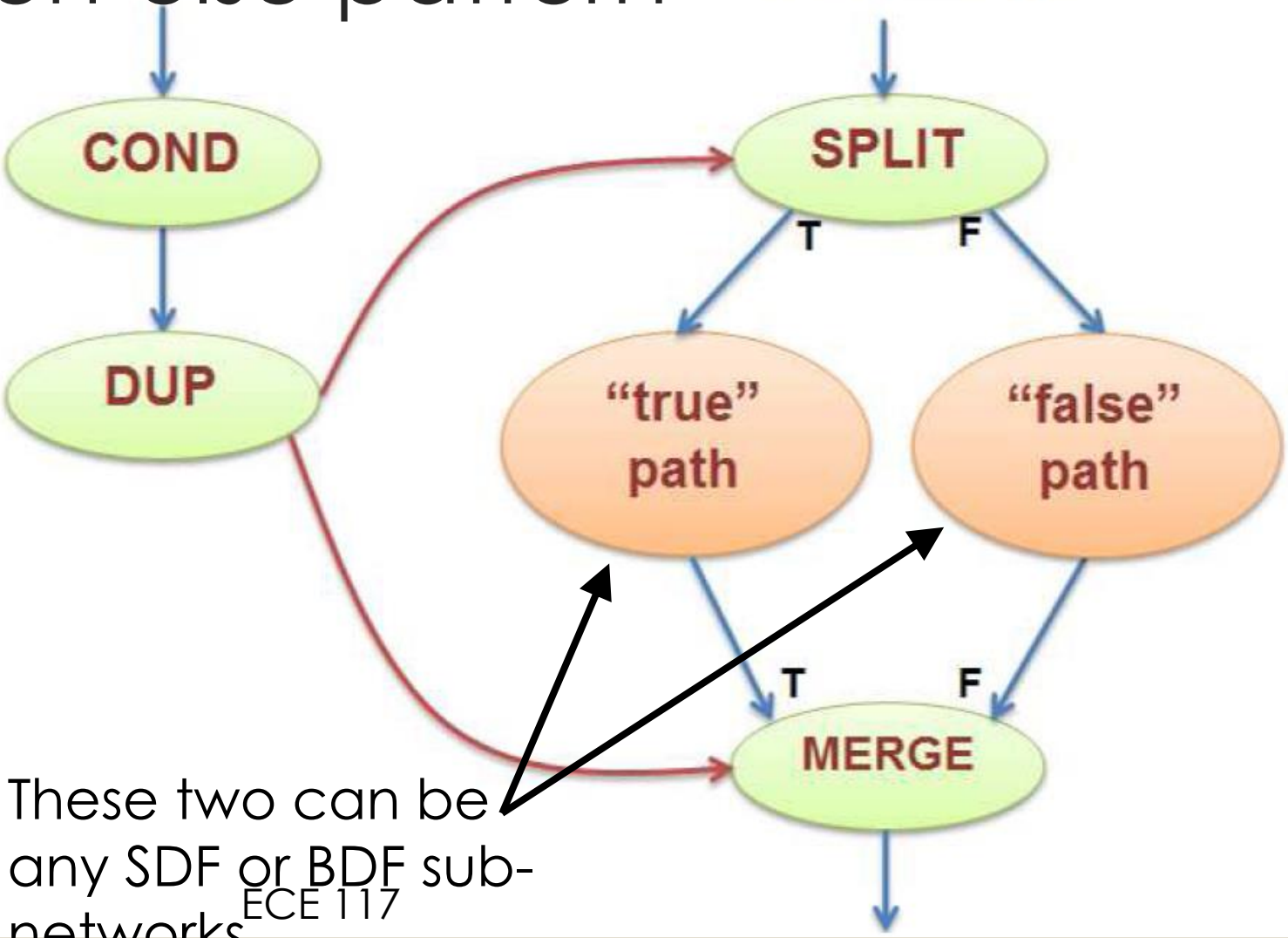


Boolean Data Flow (BDF)

- The bad: BDF is Turing-equivalent, so schedulability in finite memory is undecidable
- The good: there are two ***common and practically useful*** BDF network patterns for which schedulability is decidable:
  - If-then-else
  - Data-dependent iteration loop
- With these two patterns one can ***write any non-recursive algorithm***
- Each pattern behaves like an SDF process
  - constant number of tokens consumed and produced for each execution

If-then-else pattern

The whole pattern behaves like an SDF process

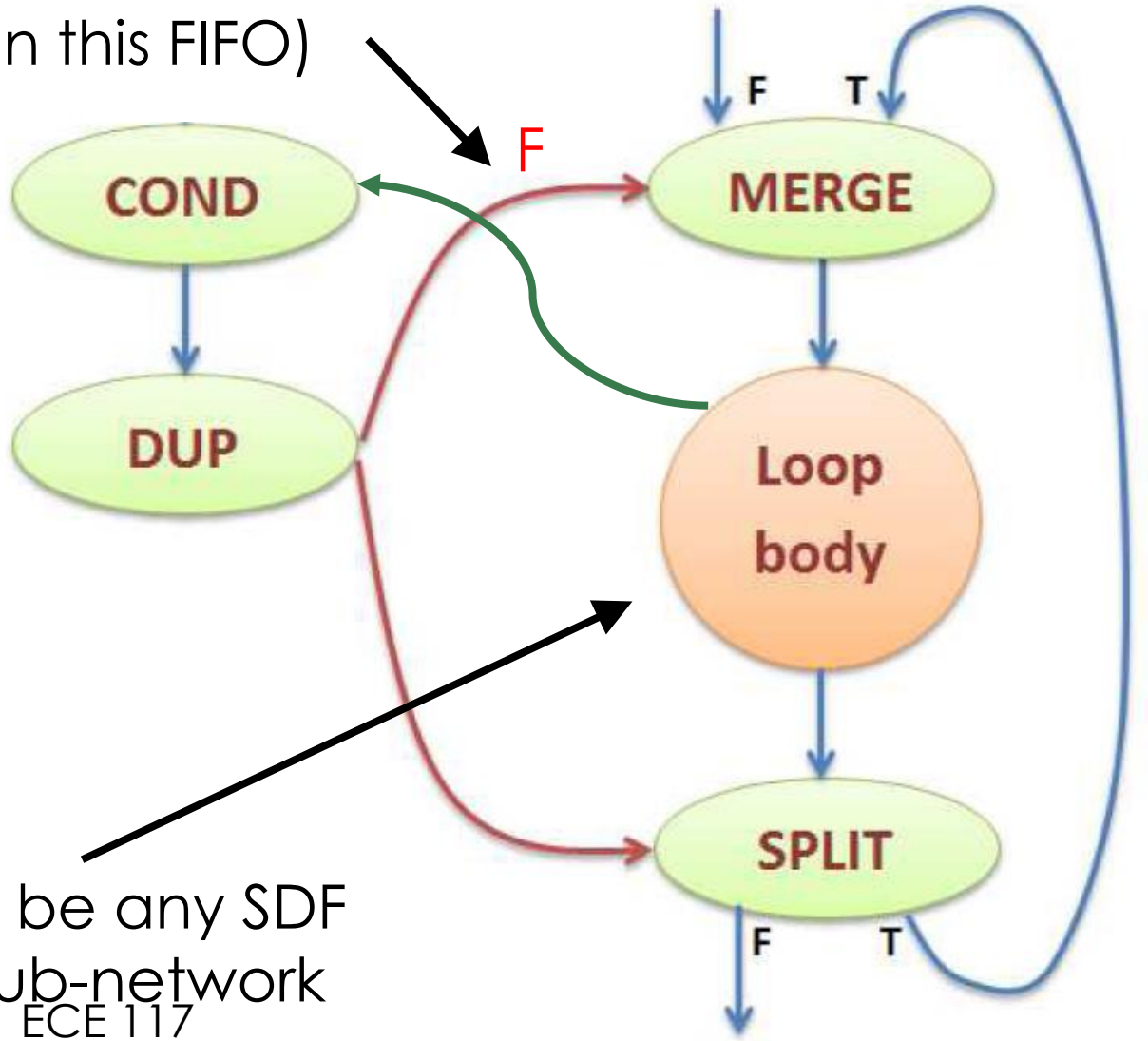


These two can be any SDF or BDF sub-networks

Data-dependent loop pattern

Initial value False
(only on this FIFO)

The whole pattern behaves like an SDF process



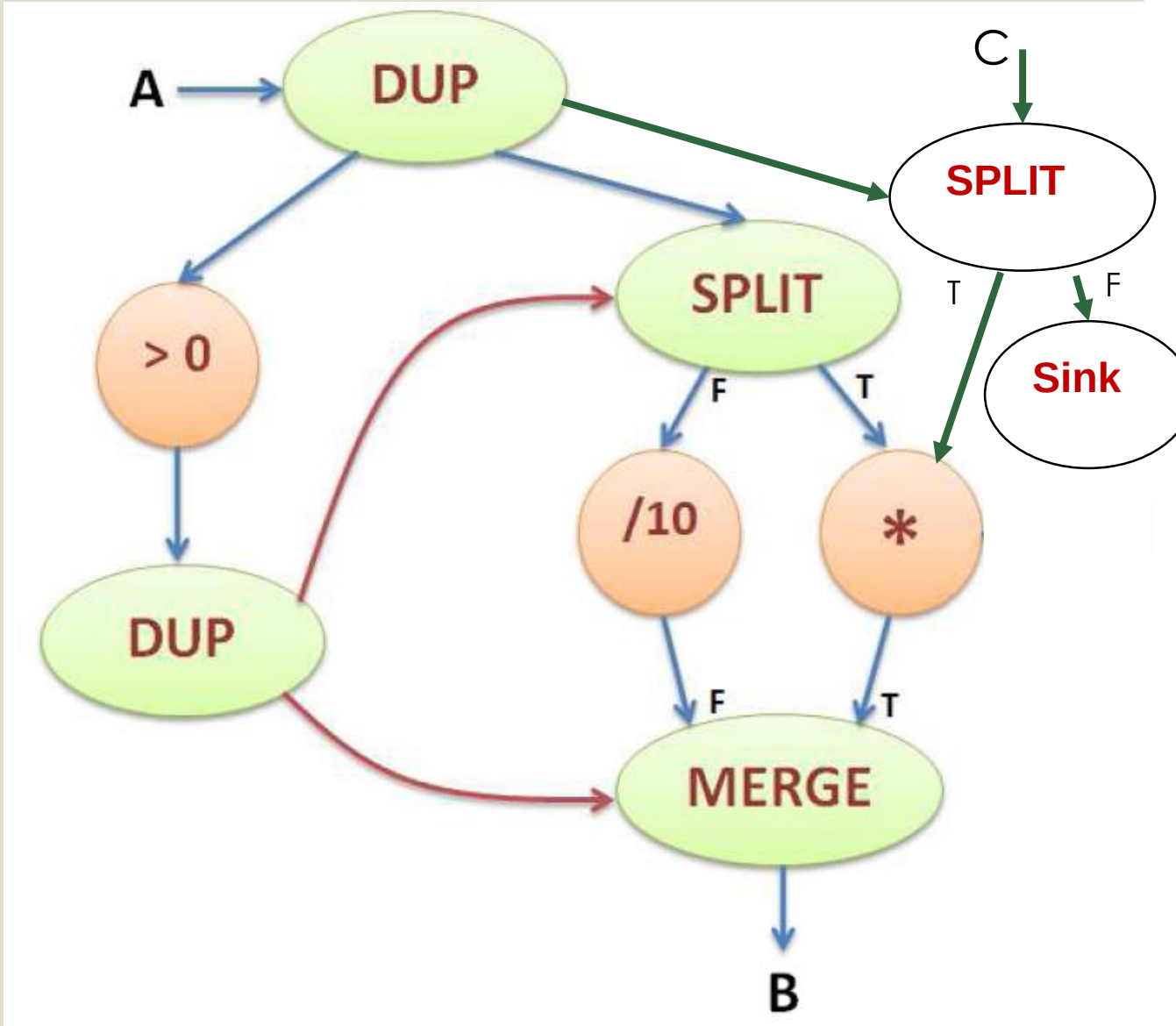
This can be any SDF or BDF sub-network

BDF example

- Corresponding code:

```
if ( A > 0)
   B = A*C;
else
   B = C, A/10;
```

- ***Must consume C even if A<=0!!***

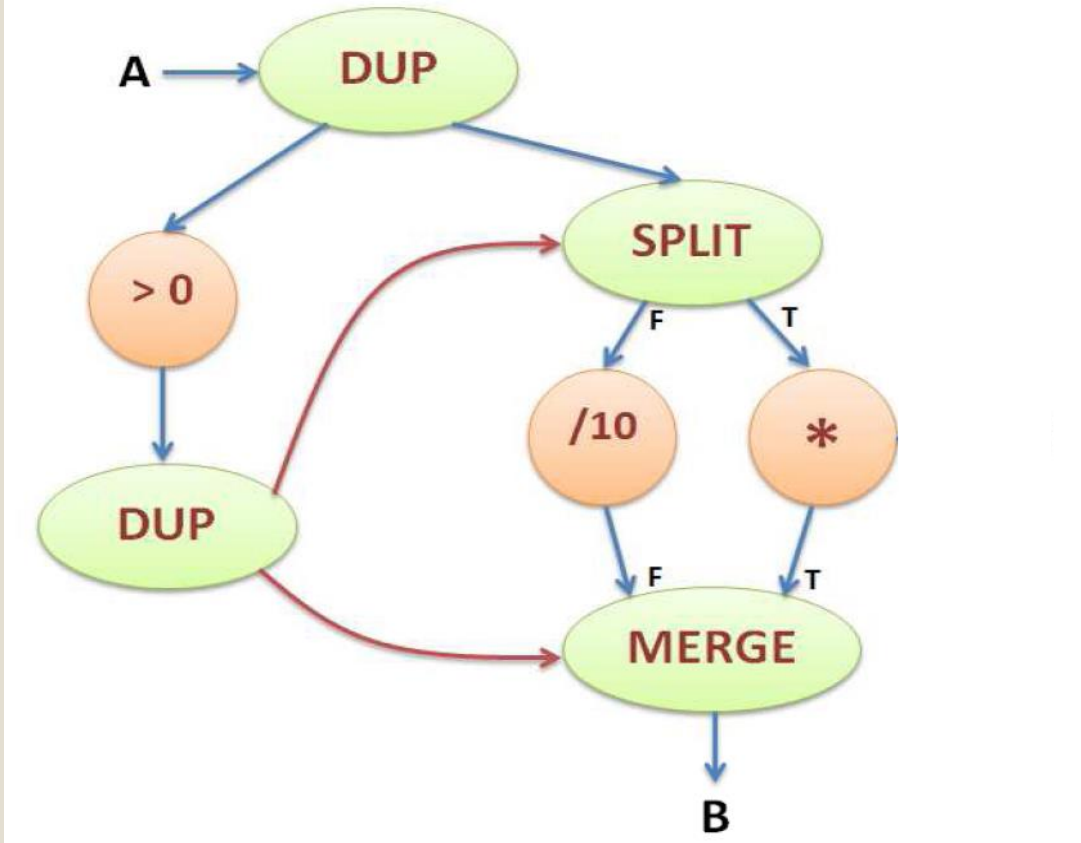


BDF schedulability check

- Intuitive algorithm:
  - Check if any use of split and merge falls into one of the two patterns
  - Collapse each pattern into an SDF process
  - Schedule the network as if it were a (hierarchical) SDF network
- The real algorithm is more complex, and writes the balance equations using additional variables representing the "probability" that each condition is true or false
- Check that the rank of the balance equations is non-full ***for any value of the auxiliary variables***

BDF (simplified) schedulability example

$$
\begin{aligned}
x_{DUP1} &= x_{>0} \\
x_{>0} &= x_{DUP2} \\
x_{DUP2} &= x_{SPLIT} \\
x_{DUP1} &= x_{SPLIT} \\
x_{SPLIT} &= p * x_{/10} \\
x_{SPLIT} &= (1-p) * x_{*} \\
x_{DUP2} &= x_{MERGE} \\
x_{MERGE} &= p * x_{/10} \\
x_{MERGE} &= (1-p) * x_{*}
\end{aligned}
$$



- Total rank = 6 **for any value of p** (between 0 and 1)
- 7 variables -> schedulable!

Outline

- Static Dataflow Networks
  - Single-resource scheduling (SW)
  - Architectural assumptions: Digital Signal Processors
  - Pareto optimality
- Boolean Dataflow Networks
- **Multi-resource scheduling (HW and SW)**
  - List scheduling
  - Integer Linear Programming scheduling

Multi-resource SDF and BDF scheduling

- How about scheduling on multiple resources?
- The balance equations still define the space of valid solutions
- However, now the design space is much broader
- In addition to
  - the number of executions (determined by the balance equation solution) and
  - the execution order (depending on tokens in FIFOs)
- one also needs to consider on which resource (processor, HW unit, …) each process is executed
- This choice (assign processes to resources) is called ***binding***
- Execution time depends ***a lot*** on binding choices

Multi-resource scheduling and binding

- Basic question: should I execute the DF network:
  - Slower on fewer resources
  - Faster on more resources
- We will consider it for the purpose of ***HW implementation of SDF networks***
- With small changes, the same considerations hold also for multi-processor SW implementation of SDF networks
- Implementation of BDF networks (SW or HW) is similar, but must consider mutual exclusion between if-then-else branches
  - They can be executed on a single resource without loss of performance

HW scheduling of SDF networks

- Basic assumptions:
  - Processes are simple operations (add, multiply, load, store, …)
  - There are no splits and merges (SDF only)
  - Each execution of a process consumes 1 value and produces 1 value ("unit rate" SDF)
    - ***The minimal schedule executes each process once***
  - Each operation takes exactly one clock cycle (no "chaining")
  - FIFOs are implemented with shift registers
  - Performance is the ***latency*** (in clock cycles) between the first and the last executed process (no pipelining)
- Each assumption can be relaxed without changing much (we will not look into this)
- HW scheduling of BDF networks is also called ***High-Level Synthesis (HLS)***

High-level synthesis

- Typically divided into several phases:
  - ***Allocation***, i.e. choice of how many resources (adders, subtractors, ALUs, multipliers, …) will be used
  - ***Scheduling*** of the processes (i.e. operations) on the set of selected resources
  - ***Binding*** of operations to resources
- Simultaneous solution of these three problems would lead to better results (closer to the Pareto-optimal set), but it is typically ***too complex***
- Even finding the set of Pareto optimal schedules on multiple resources has exponential complexity in the number of processes and resources

HLS scheduling problems

- Since the Pareto-optimal set has too many solutions, typically people pose and solve two variants:
  - Performance-constrained scheduling: find the schedule using the smallest set of resources and that can be executed within a given latency (number of clock cycles)
  - Resource-constrained scheduling: find the schedule that can be executed within the smallest latency on a given set of resources
- Most "real" problems are performance-constrained
- An algorithm solving one of the two can solve the other
  - E.g. given a resource-constrained scheduling algorithm, try to increase the set of resources until the desired latency is achieved

Exact and heuristic algorithms

- Unfortunately even performance-constrained and resource-constrained scheduling are too complex
- Finding the best solution (smallest set of resources or latency) takes an exponential time, with respect to:
  - Both the number of processes (operations)
  - And the number of resources (ALUs, adders, multipliers, …)
- Since a real HW design easily contains hundreds of thousand of operations and thousands of resources, ***we cannot hope to use an algorithm that finds the best solution***
- In practice we use ***heuristic algorithms*** that do not explore the full design space, and hence can miss the Pareto-optimal solutions

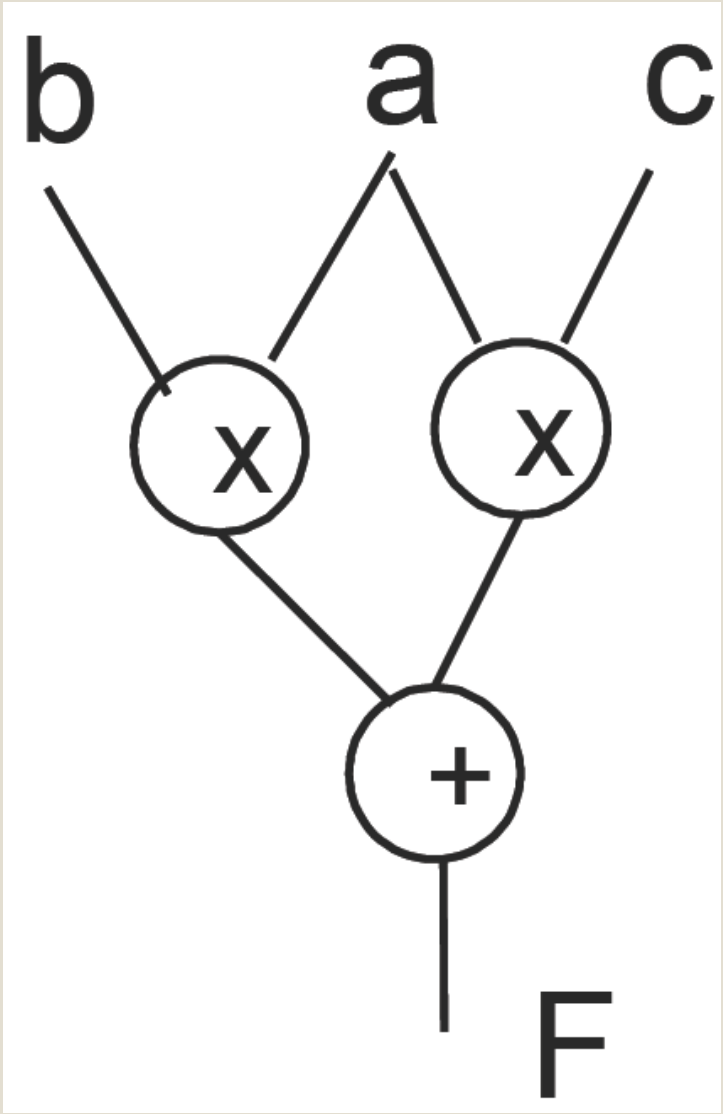
Exact and heuristic algorithms

- "Good" heuristic algorithms should:
  - Have a complexity (i.e. scheduling algorithm execution time) that is linear in the number of processes and resources
    - I.e. in practice we can handle only O(n), or at most O(n log n) algorithms (we can at most sort the processes…)
  - Produce a result that is not too far from the set of Pareto-optimal solutions
    - This set is also called "Pareto surface" for a multi-dimensional optimization problem, like scheduling
- How do we evaluate heuristics?
  - Compare them one against the other on a set of ***standard publicly available benchmark designs***
  - Compare them against exact algorithms for small problems (with a few tens or hundreds of processes)
  - Then hope for the best…

Example

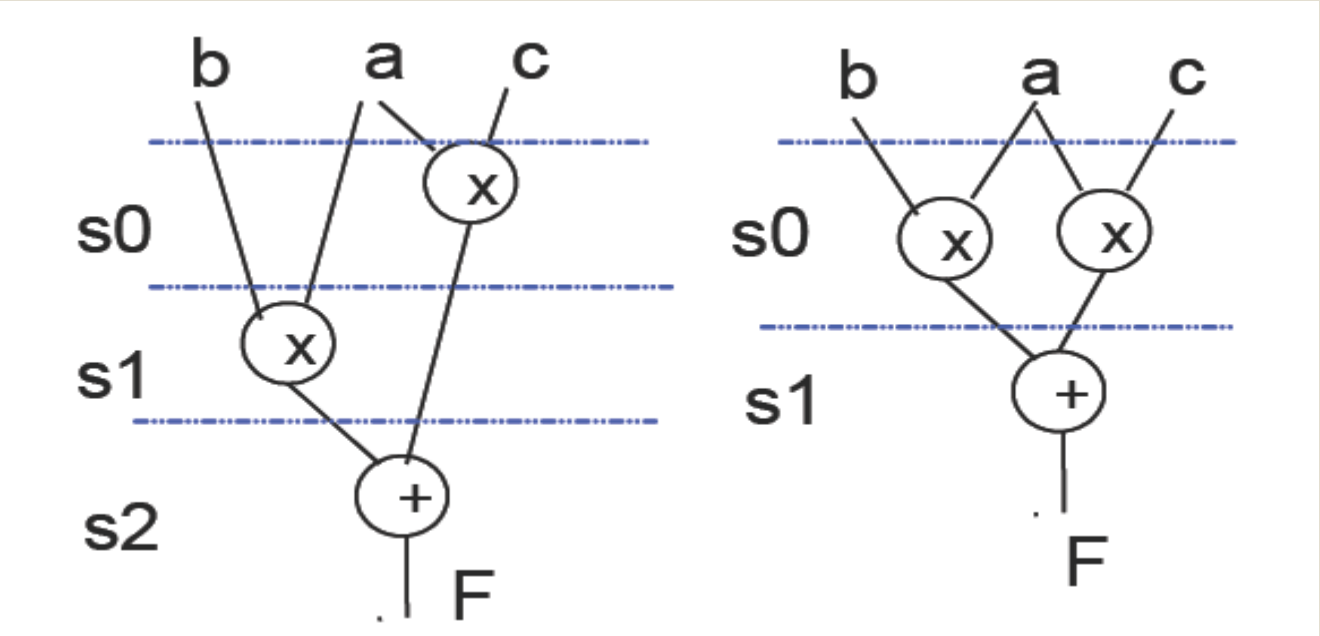
- Consider this process network:

- It can be generated, e.g., if one wants to synthesize in HW this SystemC code:
  F = a * b + a * c;
  (assuming that we cannot simplify it)



Example

- Two possible (both Pareto-optimal) schedules:



Latency = 3 clock cycles Latency = 2 clock cycles

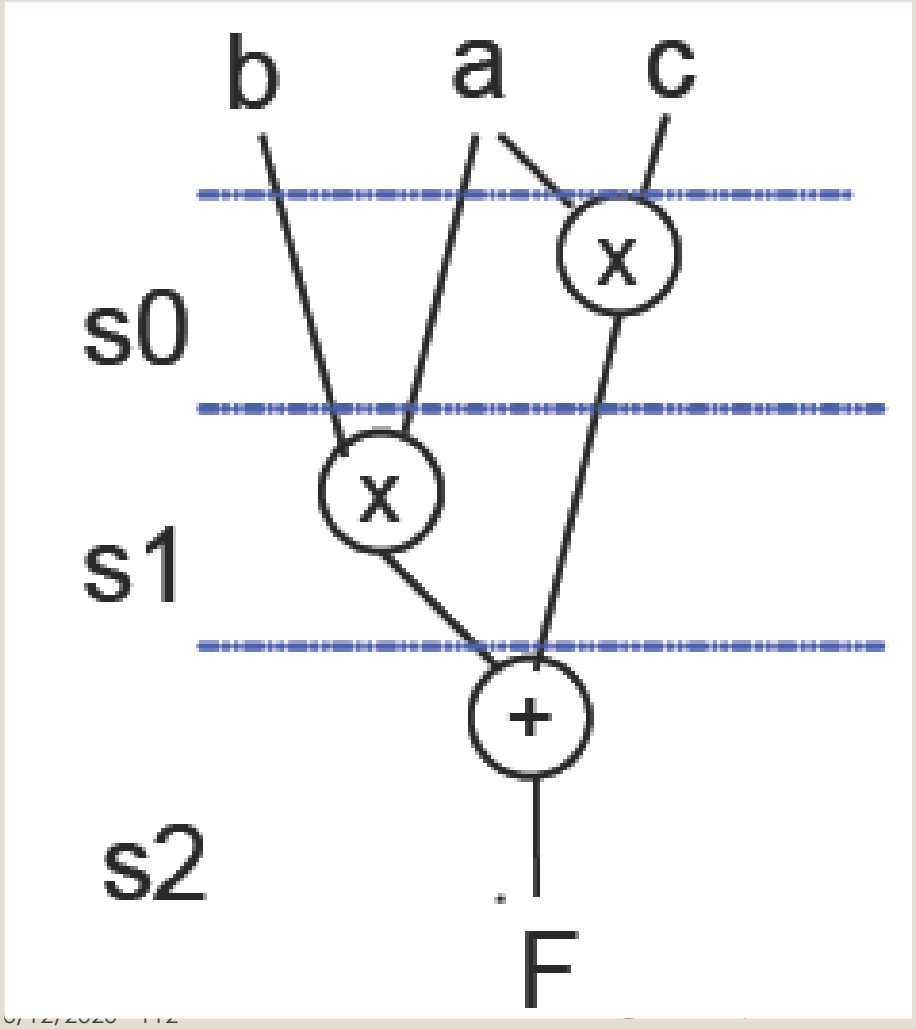
Resources = { X, + } Resources = { X, X, + }

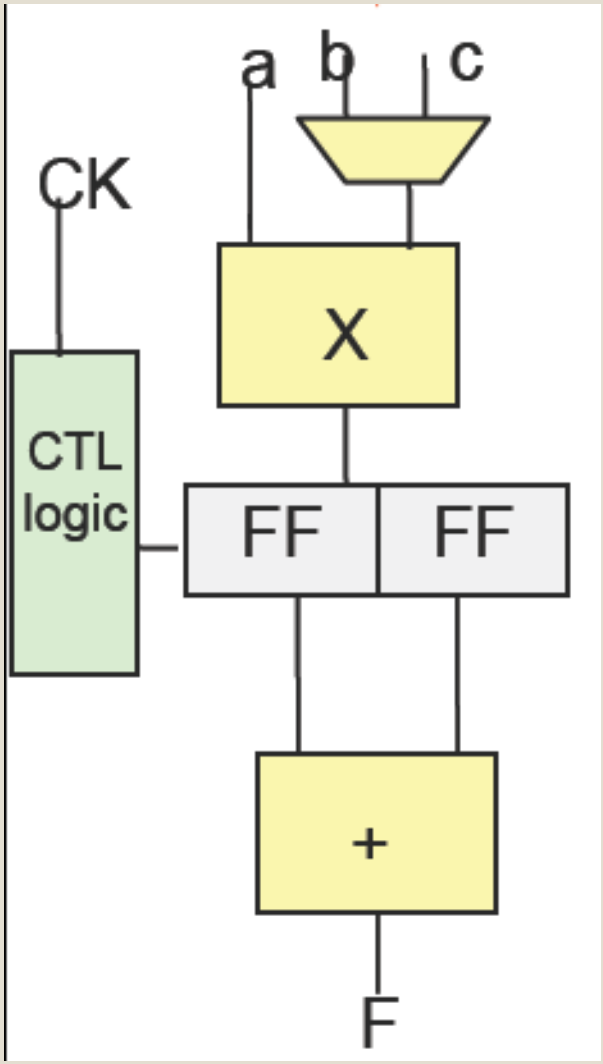
Smaller Faster

From schedule to RTL

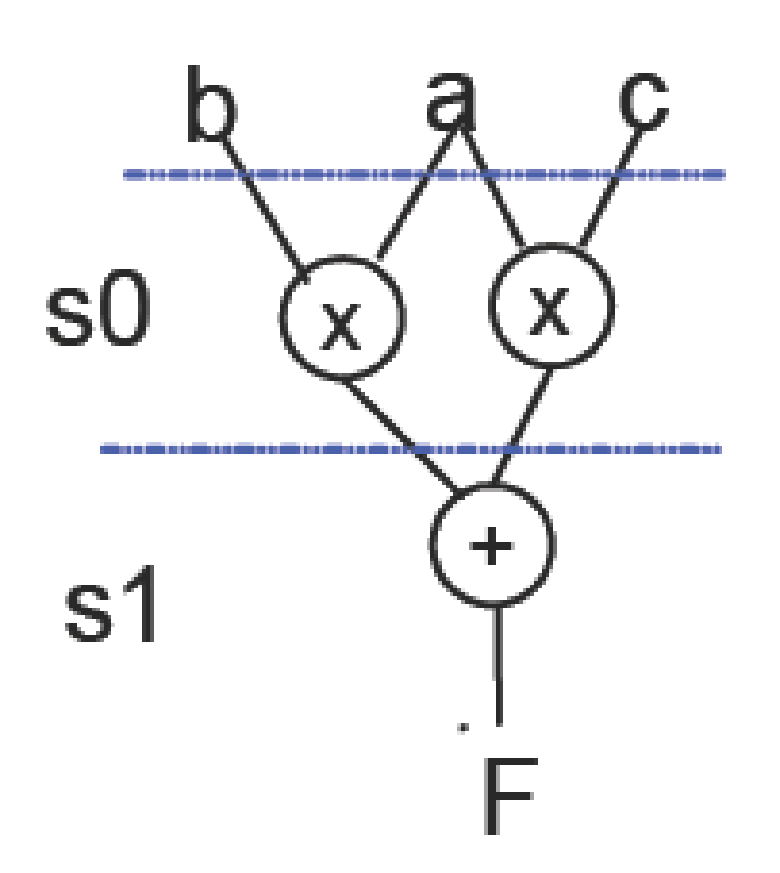
- Given a binding satisfying a given schedule (in turn satisfying a given allocation) for a given DF network one can:
  - Allocate registers (implementations of FIFOs) that store the data between clock cycles and between resources
  - Generate a control FSM that enables registers and muxes to execute correctly the given DF network
- We will not cover these steps in detail
  - Laborious, but conceptually simple
- Let us look at two scheduling algorithms:
  - Fast and good heuristic algorithm
  - Slow exact algorithm

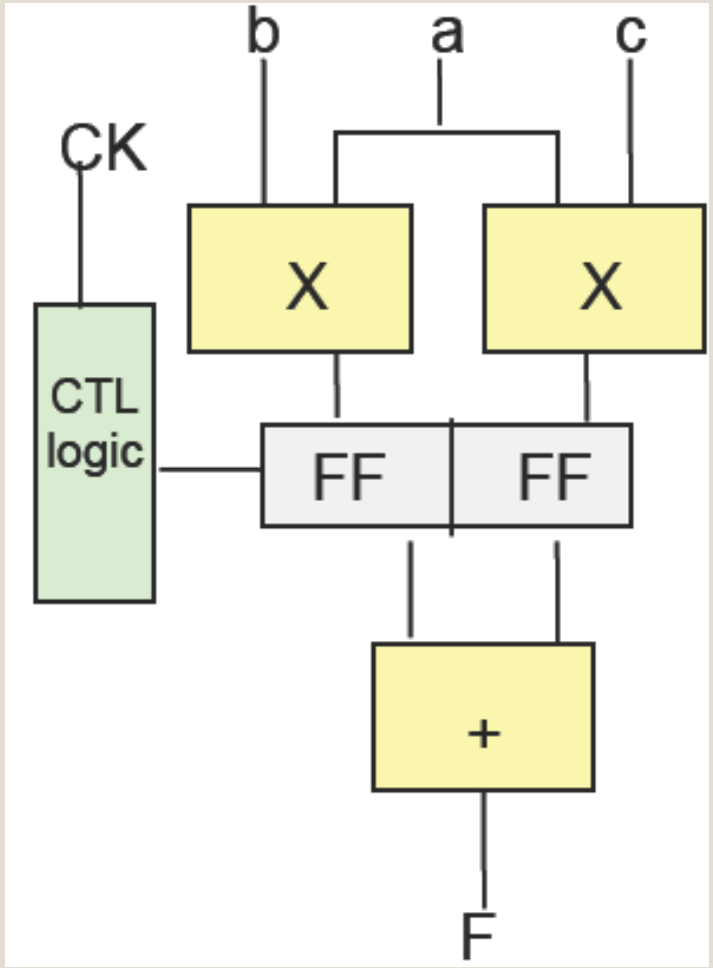
From schedule 1 to RTL





From schedule 2 to RTL





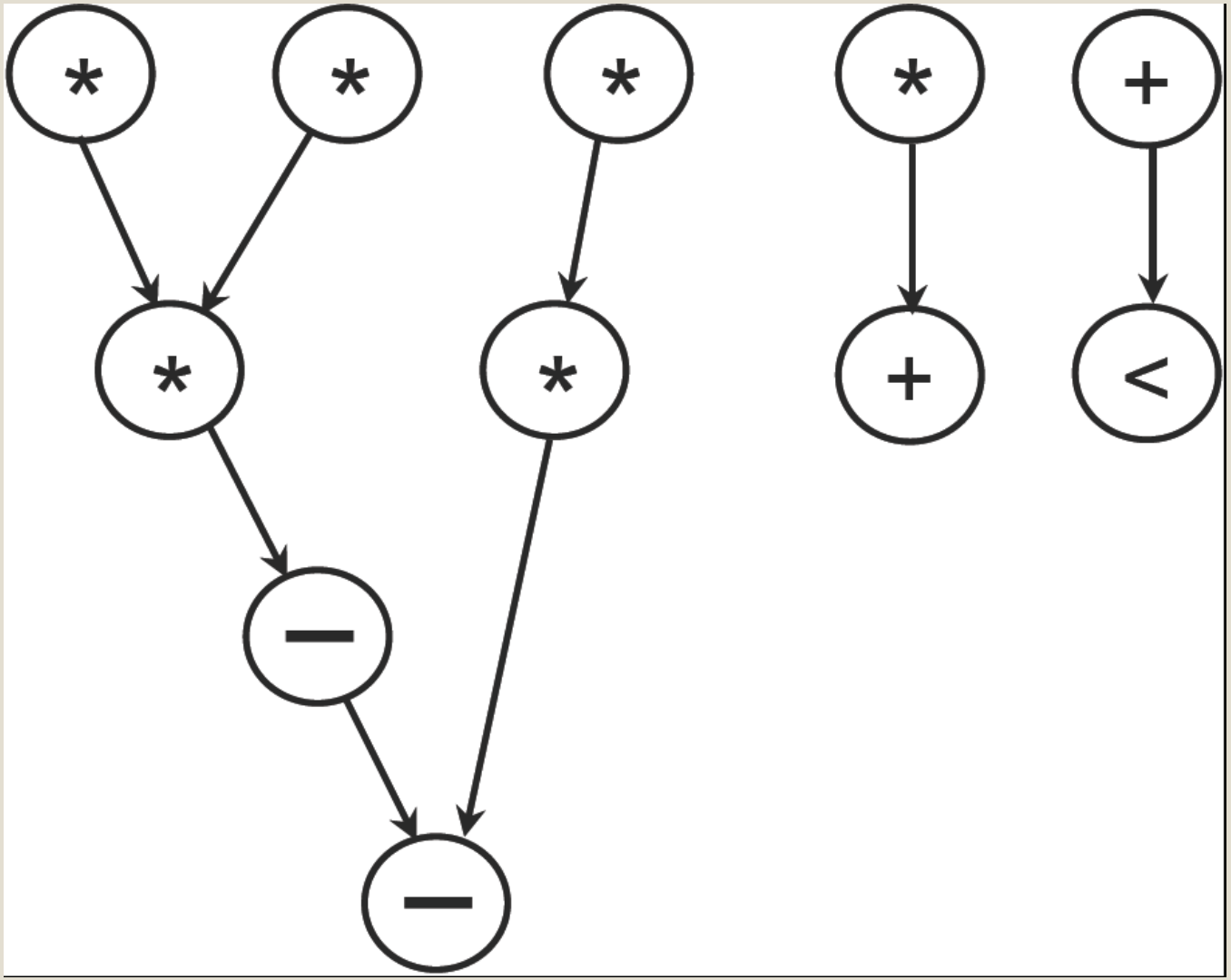
Outline

- Static Dataflow Networks
  - Single-resource scheduling (SW)
  - Architectural assumptions: Digital Signal Processors
  - Pareto optimality
- Boolean Dataflow Networks
- **Multi-resource scheduling (HW and SW)**
  - **List scheduling**
  - Integer Linear Programming scheduling

Heuristic HW scheduling

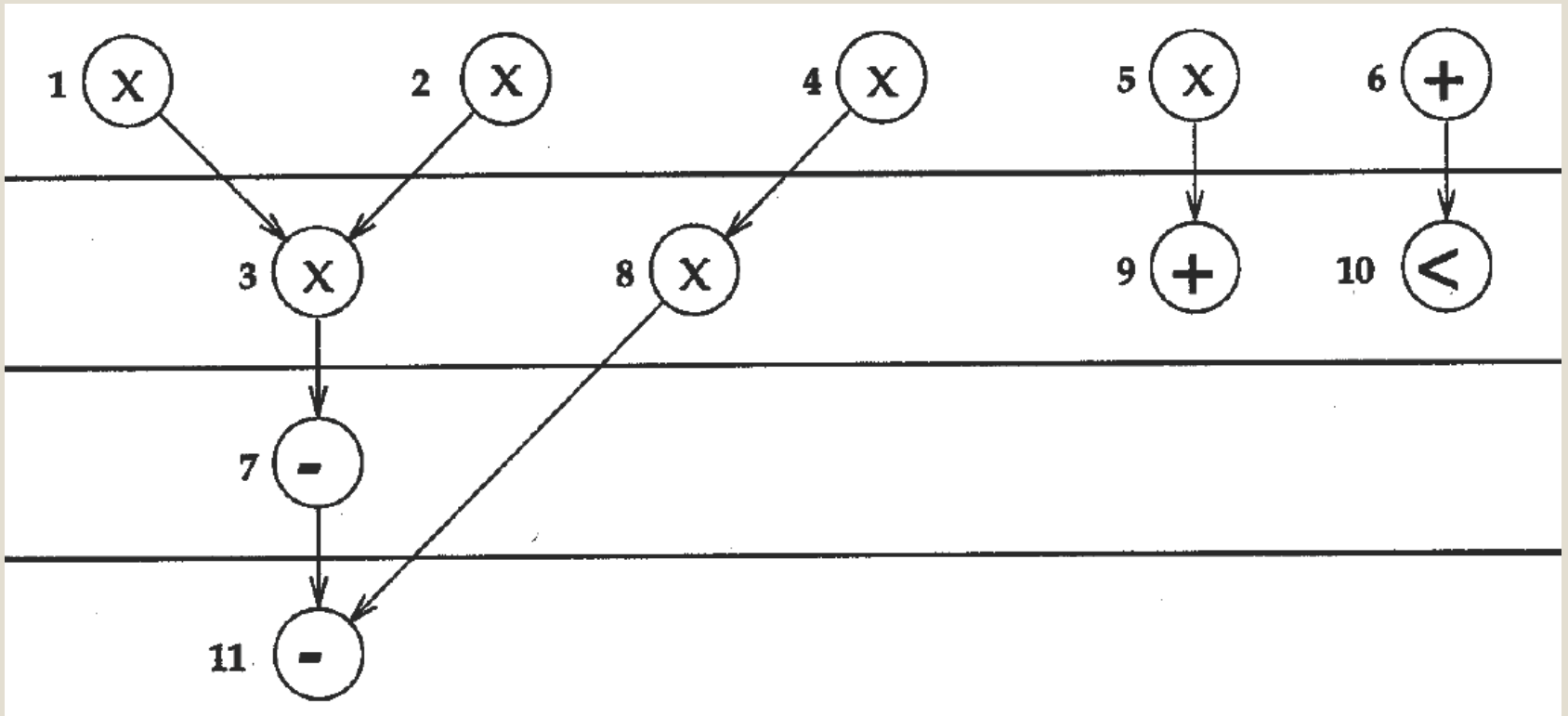
- The heuristic algorithm that we will consider is called
  - List scheduling in High-Level Synthesis for HW
  - Earliest Deadline First in real-time SW scheduling
- It solves resource-constrained scheduling
  - It can also be used for performance-constrained scheduling
- First we will consider two other algorithms that provide ***timing bounds*** that are used by list scheduling:
  - As Soon As Possible (ASAP) provides a ***lower bound on the latency***
  - As Late As Possible (ALAP) provides an ***upper bound on the time by which an operation must be scheduled not to exceed the minimum latency***
  - Neither of them is useful per se, because they assume ***unbounded resources***

Example SDF graph



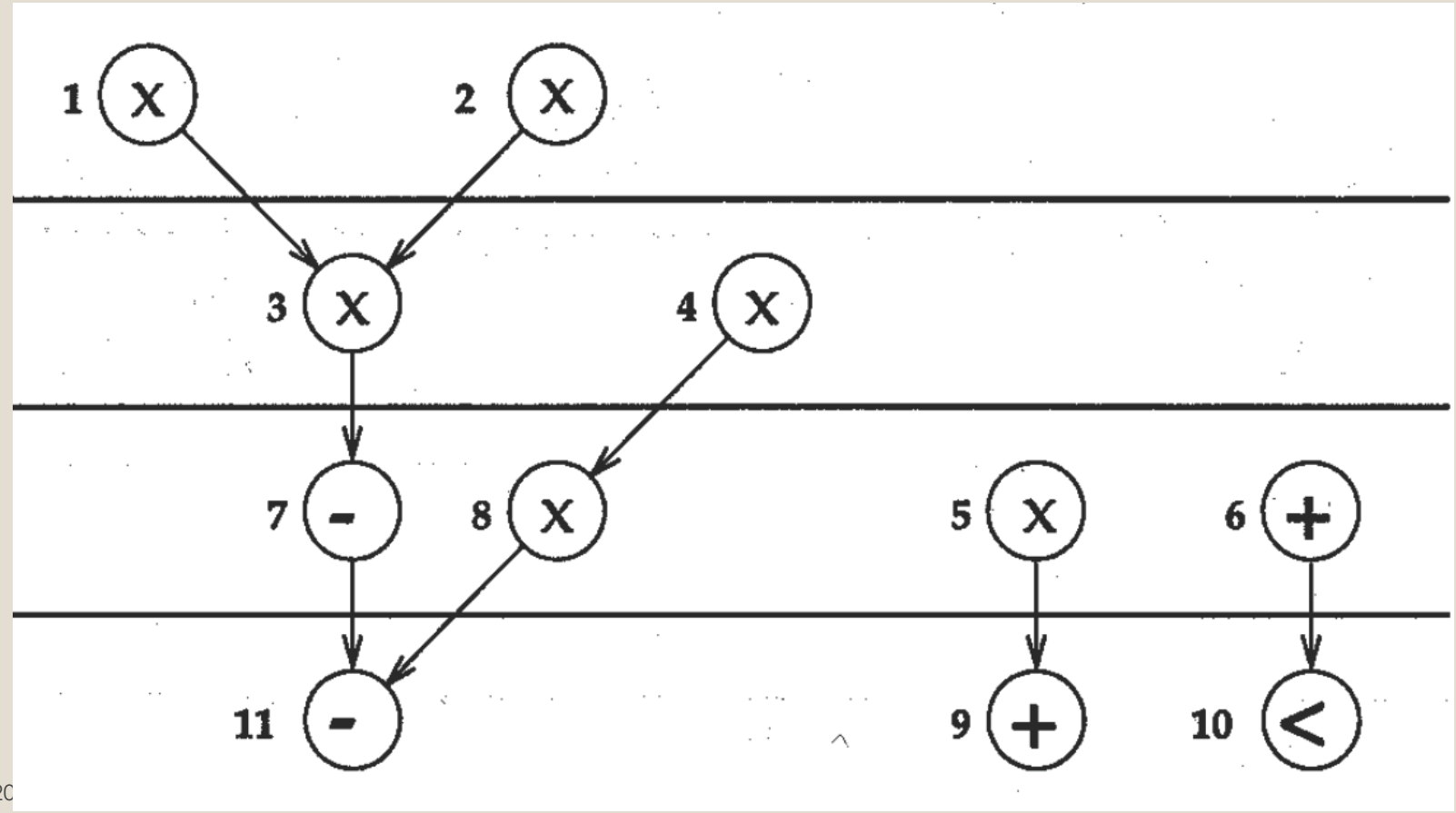
ASAP scheduling

- Schedule each process on its own resource, as soon as its inputs are ready (input FIFOs contain one value)



ALAP scheduling

- Schedule each process on its own resource, as late as possible without increasing the overall latency

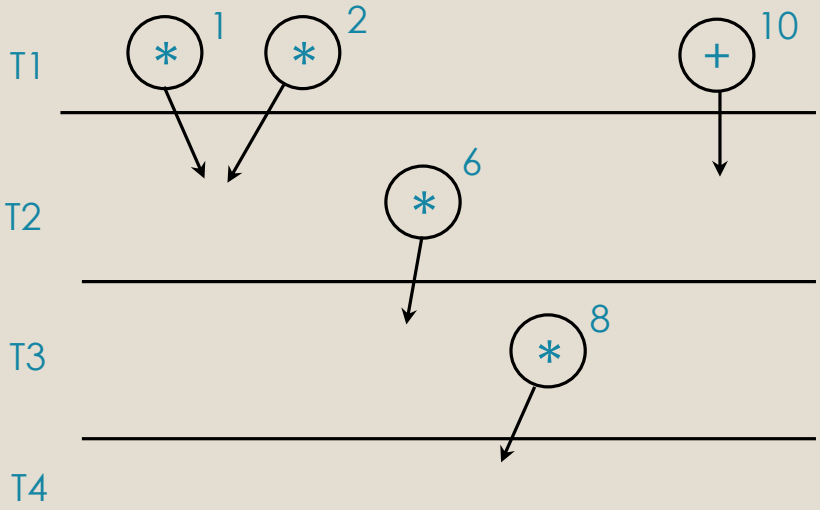


List scheduling

- List scheduling keeps a queue of "ready" operations, sorted by "criticality"
- At each clock cycle, it picks one by one operations to be scheduled on available resources
- When there are no more available resources or ready processes for this clock cycle, scheduling moves to the next clock cycle, when
  - All resources are free again
  - Some more processes can become ready
- The EDF variant of list scheduling defines criticality for each operation as the ALAP cycle for that operation
- Operations which are scheduled earlier in ALAP have higher criticality

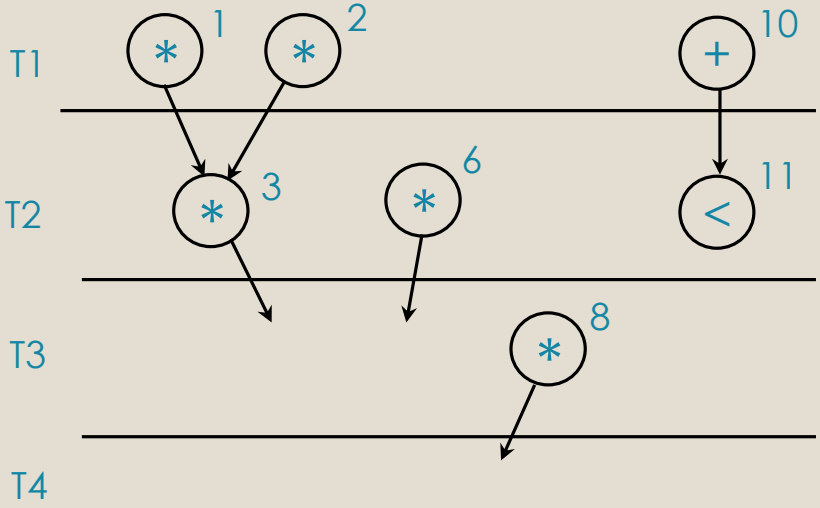
Example of list scheduling

- Resource constraint:
  - 2 MULT (*), 2 ALU (+, <)
- Step 1:
  - $U_{1,1} = \{v_1, v_2, v_6, v_8\}$, select $\{v_1, v_2\}$
  - $U_{1,2} = \{v_{10}\}$, select all



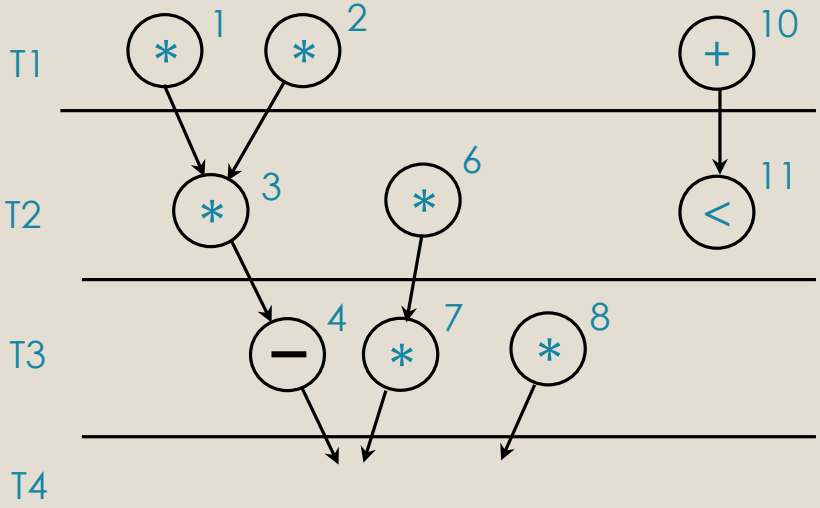
Example of list scheduling

- Resource constraint:
  - 2 MULT (*), 2 ALU (+, <)
- Step 1:
  - $U_{1,1} = \{v_1, v_2, v_6, v_8\}$, select $\{v_1, v_2\}$
  - $U_{1,2} = \{v_{10}\}$, select all
- Step 2:
  - $U_{2,1} = \{v_3, v_6, v_8\}$, select $\{v_3, v_6\}$
  - $U_{2,2} = \{v_{11}\}$, select all



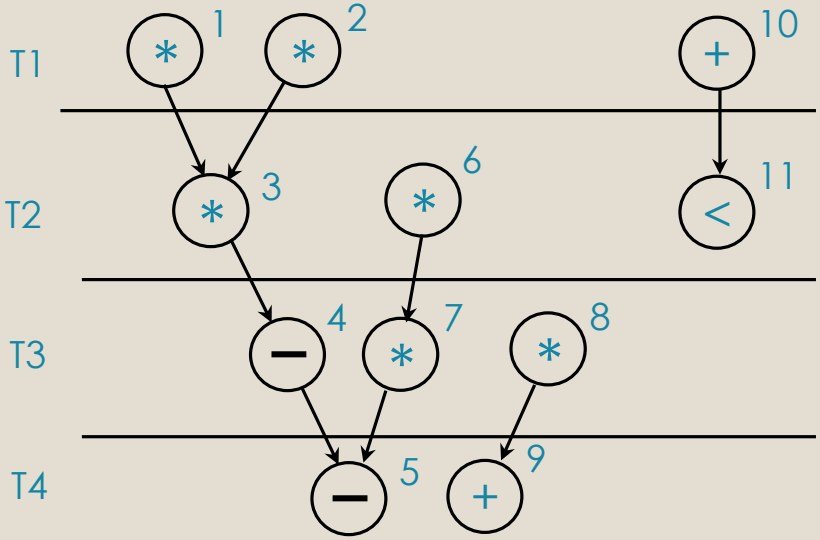
Example of list scheduling

- Resource constraint:
  - 2 MULT (*), 2 ALU (+, <)
- Step 1:
  - $U_{1,1} = \{v_1, v_2, v_6, v_8\}$, select $\{v_1, v_2\}$
  - $U_{1,2} = \{v_{10}\}$, select all
- Step 2:
  - $U_{2,1} = \{v_3, v_6, v_8\}$, select $\{v_3, v_6\}$
  - $U_{2,2} = \{v_{11}\}$, select all
- Step 3:
  - $U_{3,1} = \{v_7, v_8\}$, select all
  - $U_{3,2} = \{v_4\}$, select all



Example of list scheduling

- Resource constraint:
  - 2 MULT (*), 2 ALU (+, <)
- Step 1:
  - $U_{1,1} = \{v_1, v_2, v_6, v_8\}$, select $\{v_1, v_2\}$
  - $U_{1,2} = \{v_{10}\}$, select all
- Step 2:
  - $U_{2,1} = \{v_3, v_6, v_8\}$, select $\{v_3, v_6\}$
  - $U_{2,2} = \{v_{11}\}$, select all
- Step 3:
  - $U_{3,1} = \{v_7, v_8\}$, select all
  - $U_{3,2} = \{v_4\}$, select all
- Step 4:
  - $U_{4,2} = \{v_5, v_9\}$, select all



Example of list scheduling

- In this case list scheduling managed to schedule within the lower bound
  - This schedule is ***optimal***
- In general, this may or may not happen
- Other "criticality" functions:
  - "slack" (ALAP cycle – ASAP cycle)
  - "force" attracting to already scheduled operations
- List scheduling can be easily adapted to use "real" delays of combinational logic, rather than 1 cycle
  - Simply schedule "ready" operations in a clock cycle as long as there are resources and the clock period is not exceeded

Outline

- Static Dataflow Networks
  - Single-resource scheduling (SW)
  - Architectural assumptions: Digital Signal Processors
  - Pareto optimality
- Boolean Dataflow Networks
- **Multi-resource scheduling (HW and SW)**
  - List scheduling
  - **Integer Linear Programming scheduling**

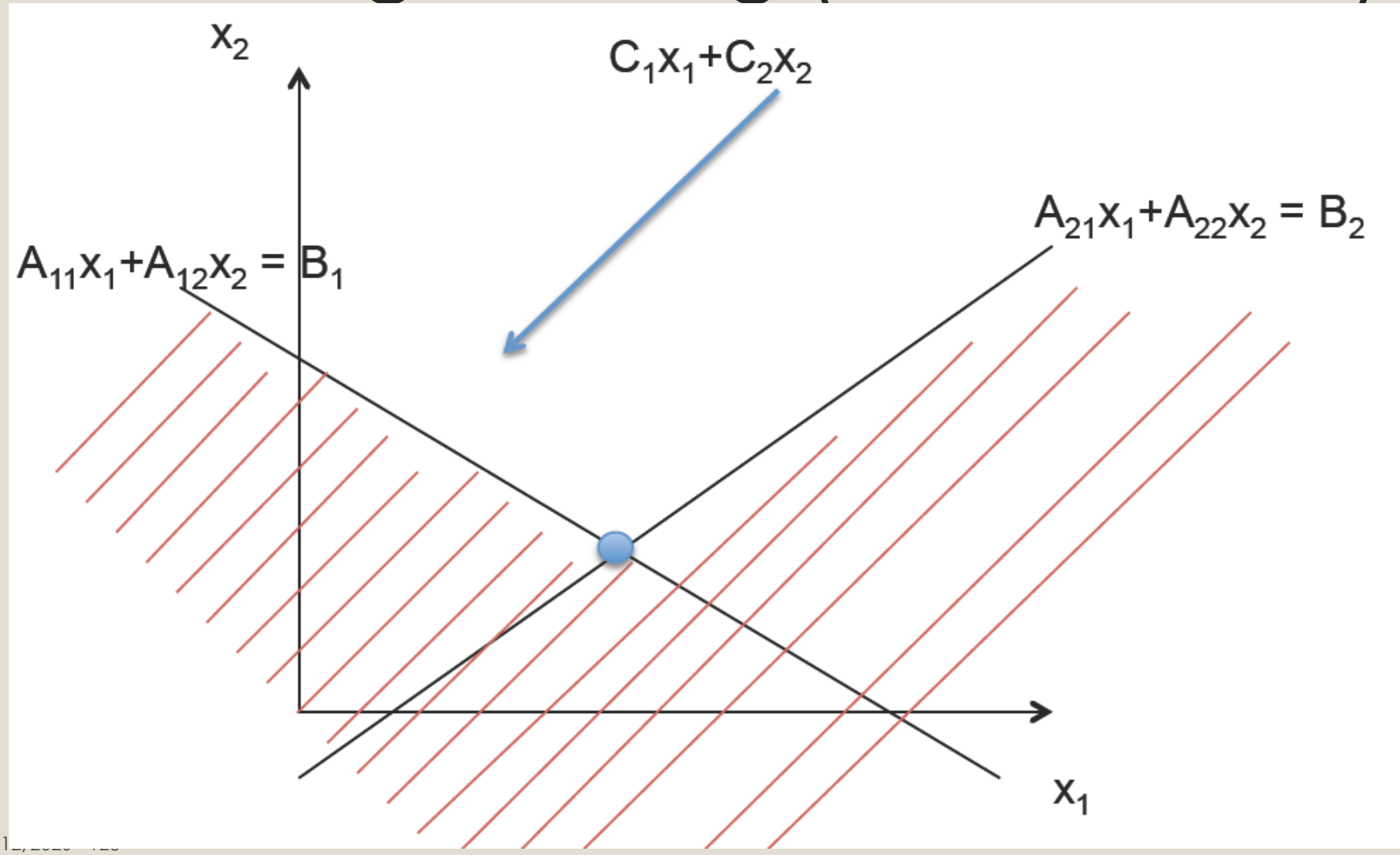
Exact HW scheduling

- A simple exact algorithm ***reduces*** HW scheduling to Integer Linear Programming (ILP)
- ILP is a very flexible and general optimization problem that can be used to solve many problems
  - However, there is no low-complexity algorithm to solve it
- It is a variant of a more widely used optimization problem called ***Linear Programming (LP)***
  - LP can be solved very efficiently (polynomial executime in the input problem size)
- A problem can be "reduced" to another one by providing a ***mapping*** from each instance of the former to an instance of the latter
  - If the latter can be solved (efficiently), the former can also be solved (efficiently)

Linear Programming

- Given:
  - A set of linear inequalities: A X <= B where:
    - A is a matrix of constants
    - X is a vector of variables
    - B is a vector of constants
  - A linear cost function X C where:
    - C is a vector of constants
- Find a set of values for the variables in X that minimizes the value of the cost function
- Intuition: each linear inequality A_i X <= B_i splits the X space into two, with a hyperplane A_i X = B_i
- The solution can be found within the ***polytope*** bounded by all the hyperplanes, by moving along the direction of the cost function

Linear Programming (2 dimensions)



Linear Programming

- The Simplex algorithm to solve an LP problem starts from any feasible solution inside the polytope, then moves along the direction of the cost function, following hyperedges, until it hits the optimum
- While in principle it is exponential in the number of constraints and variables, in practice it completes very quickly
  - A polynomial algorithm exists, but is less efficient in practice
- Problem instances with millions of constraints and hundreds of thousands of variables can be solved relatively quickly

Integer Linear Programming

- Same as LP, but the variables are constrained to be integer
- Finding a real-valued approximation is easy, but…
- We must find the neighboring integer point which satisfies the constraints and minimizes the cost function
- With n variables there are 2^n points to try…
- Unlike LP, ILP has no known efficient algorithm
- However, it is very flexible, and there are good implementation, which make it a popular choice to solve exactly (small instances of) difficult problems

ILP-based HLS: basic encoding

Encoding of resource constrained scheduling using ILP:

- Integer constant T representing the ***max*** latency (e.g. from list scheduling)
- For each operation i and each clock cycle k from 1 to T define a binary variable x_{ik}
  - $x_{ik} = 1$ if operation i is executed in the *k-th* clock cycle
  - $x_{ik} = 0$ otherwise
- Several sets of constraints are defined:

1. for each operation i we force $x_{ik} = 1$ for one time step, to schedule it exactly once:

$$x_{i1} + x_{i2} + \ldots + x_{iT} = 1$$

ILP-based HLS: dependency constraints

2. For each operation i define an auxiliary variable Y_i that represents the clock cycle at which i is scheduled:

   $$x_{i1} + 2\,x_{i2} + \ldots + T\,x_{iT} = Y_i$$

4. For each FIFO between operations i and j, schedule i before j:

   $$Y_i < Y_j$$

5. Define an auxiliary variable L representing the ***actual latency*** of the schedule, and a constraint for each operation i:

   $$Y_i <= L$$

ILP-based HLS

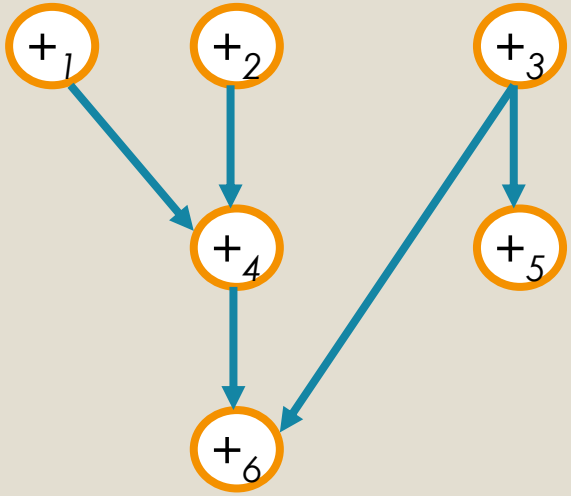
6. For each kind of resource r and operation i and each clock cycle k, the number of operations of that kind scheduled must not be larger than the number of resources n_r of that kind:

$$x_{1k} + x_{2k} + \ldots + x_{nk} \leq n_r$$

7. The cost function to minimize is simply L, i.e. the latency

Example

- T = 3 (from list scheduling)
- Execution constraints:
  - $x_{i1} + x_{i2} + x_{i3} = 1$ for $i = 1, \ldots, 6$
- Dependency constraints:
  e.g. v_4 executes after v_1
  - $x_{11} + 2x_{12} + 3x_{13} = Y_1$
  - $x_{41} + 2x_{42} + 3x_{43} = Y_4$
  - $Y_1 < Y_4$
- Resource constraints (assuming 2 adders):
  - $x_{1k} + x_{2k} + x_{3k} + x_{4k} + x_{5k} + x_{6k} \leq 2$
- Cost function:
  - $Y_i < L$
  - min L



Example

- One solution:

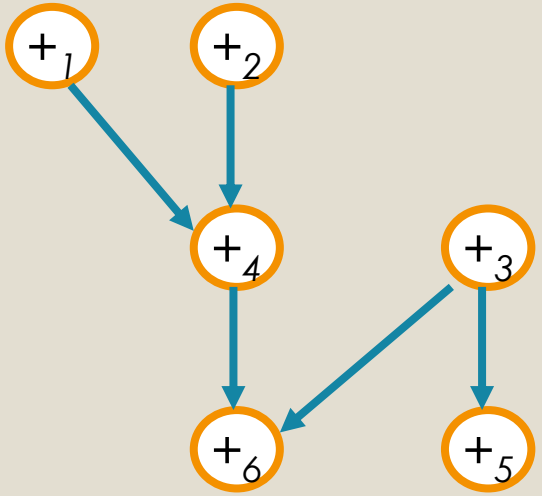
  $m = 2$

  $x_{11} = 1$, $x_{21} = 1$,

  $x_{32} = 1$, $x_{42} = 1$,

  $x_{53} = 1$, $x_{63} = 1$.

  all other $x_{ik} = 0$



A variation of ILP-based scheduling

- Instead of using L binary variable to encode the schedule of each operation, use an integer variable Y_i with value k if operation i is scheduled at clock cycle k
- Cannot represent resource constraints (which require the binary x_{ik}, and which make complexity explode)
- It is a special case of ILP (System of Difference Constraints, SDC) that can be solved very efficiently
- Resources are optimized only during binding
- Works well in practice (used by Vivado HLS, by Xilinx)

Conclusion

- Multi-resource (HW-oriented) scheduling is much more complex
- Exact algorithms, e.g. via ILP formulation:
  - Cannot be used for large DF networks
  - Can be used to provide a "justification" for a heuristic algorithm
- Heuristic algorithms use priority function to schedule processes one by one
- Variations of list scheduling are widely used e.g. by Mentor Catapult or Cadence CtoSilicon
- A variation of the ILP formulation is used by Vivado HLS

Summary

- System-level design offers the best opportunities for optimization of unit cost and efficiency of a product
- Design time and cost must be minimized
- System-level modeling, optimization and implementation on a variety of platforms is essential to achieve both objectives
- Expressiveness and verifiability/optimizability are incompatible
- Need to find the right trade-off between the two