Introduction to Electronic Design Automation

Jie-Hong Roland Jiang
江介宏

Department of Electrical Engineering
National Taiwan University

Spring 2012

Model of Computation

Model of Computation

- In system design, intended system behavior is translated into physical implementation
  - The physical implementation can be in hardware or software, in silicon or non-silicon (e.g., living cells)
  - How a system behaves or interacts with its environmental stimuli must be specified *formally*
- Model of computation (MoC) can be seen as the subject of devising/selecting effective "data structures" in describing system behaviors precisely and concisely
- MoC gives a formal way of describing system behaviors
  - It is useful in the specification, synthesis and verification of systems

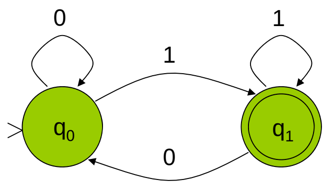
Model of Computation

- Outline
  - State transition systems
    - Finite automata / finite state machines
  - Real-time systems
    - Timed automata
  - Hybrid systems
    - Hybrid automata for hybrid systems, which exhibits both discrete and continuous dynamic behavior
  - Asynchronous systems
    - Petri nets for asynchronous handshaking
  - Signal processing systems
    - Dataflow process network for signal processing applications

(See Wikipedia for more detailed introduction)

Modeling State Transition

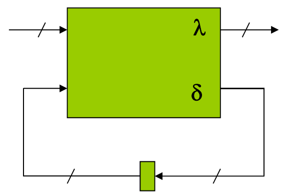
- Finite automata A = (Q, q_0, F, Σ, δ)
  - Q: states; q_0: initial state; F: accepting states; Σ: input alphabet; δ: Σ×Q→Q transition
  - Can be alternatively represented in *state diagram*
- Finite automata are used as the recognizer of *regular language*
  - Example

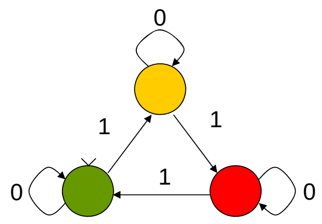


- The finite automaton accepts all binary strings ended in a "1", i.e., which form the language: (0*1*)*1 or {0,1}*1

Modeling State Transition (cont'd)

- Finite state machine (FSM) M = (Q, I, Σ, Ω, δ, λ)
  - Q: states; I: initial states; Σ: input alphabet; Ω : output alphabet; δ: Σ×Q→Q transition function; λ: Σ×Q→Ω (respectively λ: Q→Ω) output function for Mealy (respectively Moore) FSM
  - Can be alternatively represented in *state transition graph* (STG) or *state transition table* (STG)
  - E.g., vending machine, traffic light controller, elevator controller, Rubik's cube!, etc.





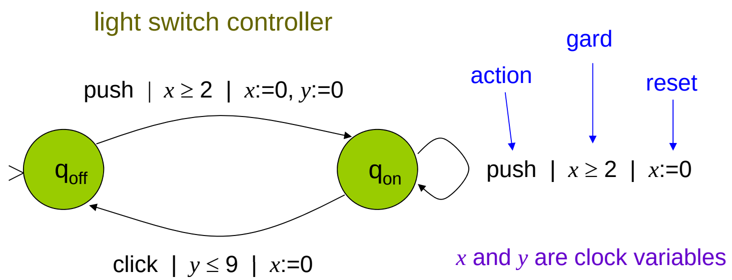
Σ	Q	Q'	Ω
0	G	G	go
0	Y	Y	yield
0	R	R	stop
1	G	Y	go
1	Y	R	yield
1	R	G	stop

Modeling State Transition (cont'd)

- FSMs are often used as controllers in digital systems
  - E.g. data flow controller, ALU (arithmetic logic unit) controller, etc.
- Variants of FSM
  - Hierarchical FSM
  - Communicating FSM
  - …

Modeling Real-Time Systems

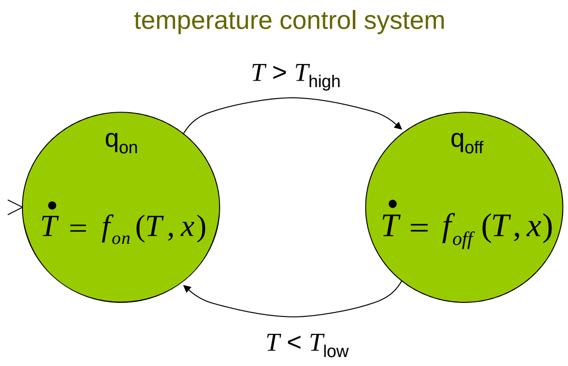
- Timed automata
  - Example



- Switch may be turned on whenever at least 2 time units has elapsed since last turn off
- Light switches off automatically after 9 time units

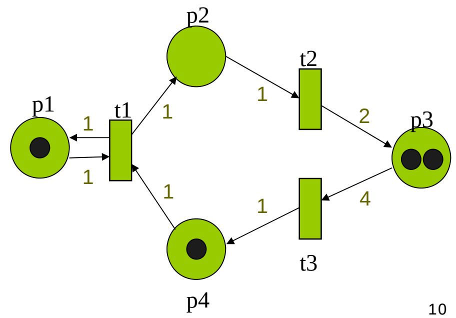
Modeling Hybrid Systems

- Hybrid automata
  - Example



Modeling Asynchronous Systems

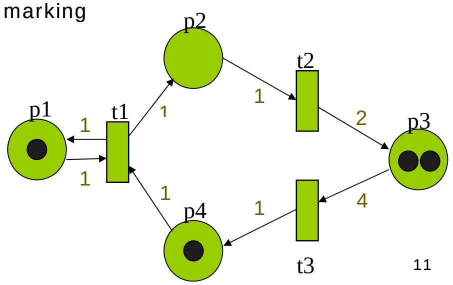
- Petri net P = (G, M_0)
  - Petri net graph G is a bipartite weighted directed graph:
    - Two types of nodes: *places* in circles and *transitions* in boxes
    - Arcs: arrows labeled with weights indicating how many tokens are consumed or produced
    - Tokens: black dots in places
  - Initial marking M_0
    - Initial token positions



ref: EE249 lecture notes, UC Berkeley

Modeling Asynchronous Systems (cont'd)

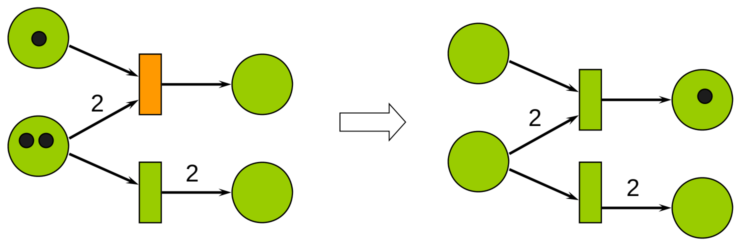
- In a Petri net graph G,
  - places: represent distributed state by holding tokens
    - marking (state) M is an vector $(m_1, m_2, \ldots, m_n)$, where m_i is the non-negative number of tokens in place p_i
    - initial marking M_0 is initial state
  - transitions: represent actions/events
    - enabled transition: enough tokens in predecessors
    - firing transition: modifies marking



ref: EE249 lecture notes, UC Berkeley

Modeling Asynchronous Systems (cont'd)

- A marking is changed according to the following rules:
  - A transition is enabled if there are enough tokens in each input place
  - An enabled transition may or may not fire (i.e. non-deterministic)
  - The firing of a transition modifies marking by consuming tokens from the input places and producing tokens in the output places

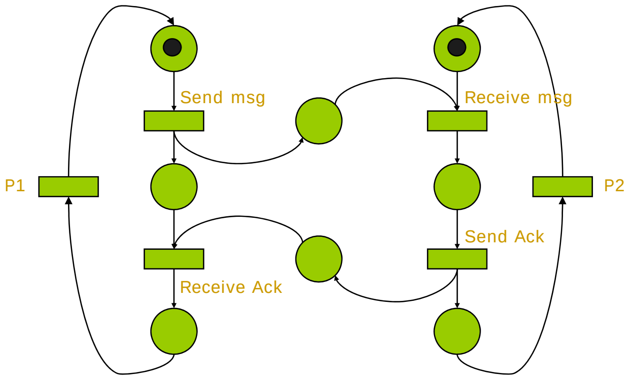


ref: EE249 lecture notes, UC Berkeley

Modeling Asynchronous Systems (cont'd)

- Example

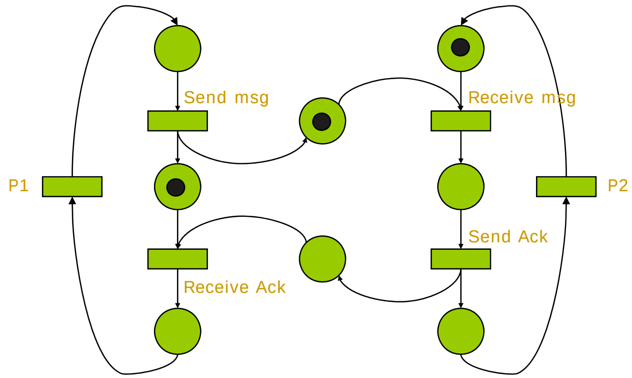
communication protocol



Modeling Asynchronous Systems (cont'd)

- Example

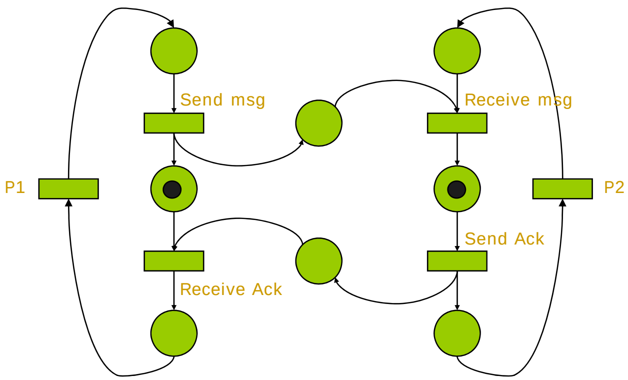
communication protocol



Modeling Asynchronous Systems (cont'd)

- Example

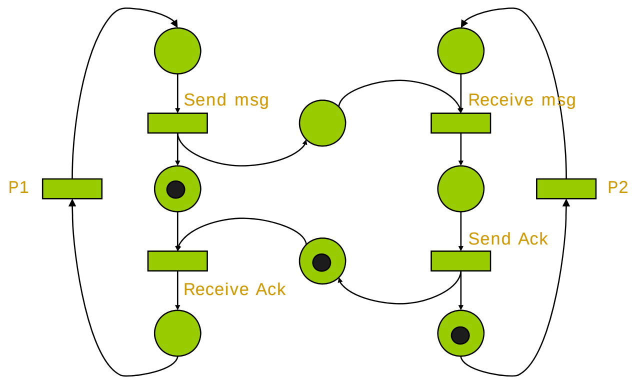
communication protocol



Modeling Asynchronous Systems (cont'd)

- Example

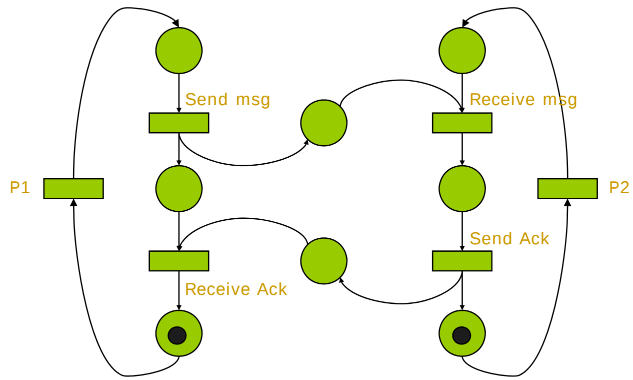
communication protocol



Modeling Asynchronous Systems (cont'd)

- Example

communication protocol

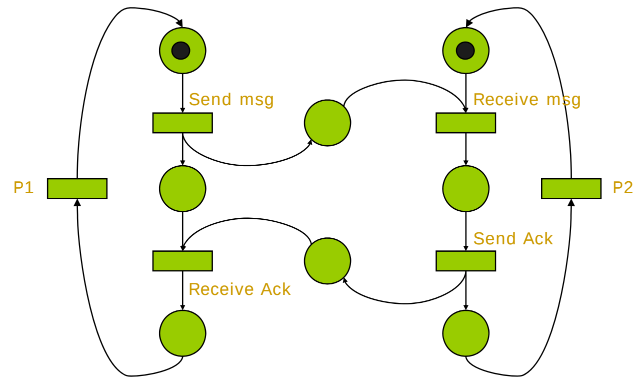


source: EE249 lecture notes, UC Berkeley

Modeling Asynchronous Systems (cont'd)

- Example

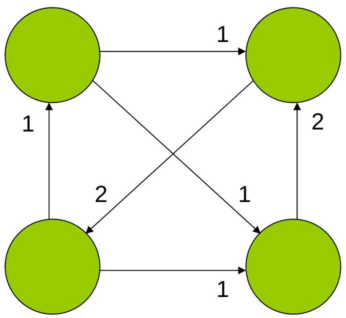
communication protocol



source: EE249 lecture notes, UC Berkeley

Modeling Signal Processing

- Data-flow process network
  - Nodes represent actors; arcs represent FIFO queues
    - Firing rules are specified on arcs
    - Actors respect firing rules that specify how many tokens must be available on every input for an actor to fire. When an actor fires, it consumes a finite number of tokens and produces also a finite number of output tokens.

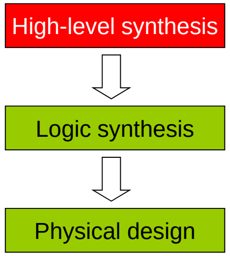


ref: http://www.create.ucsb.edu/~xavier/Thesis/html/node38.html

MoC in System Construction

- There are many other models of computation tailored for specific applications
  - Can you devise a new computation model in some domain?
- Hierarchical modeling combined with several different models of computation is often necessary
- By using a proper MoC, a system can be specified formally, and further synthesized and verified
  - In the sequel of this course, we will be focusing on FSMs mainly

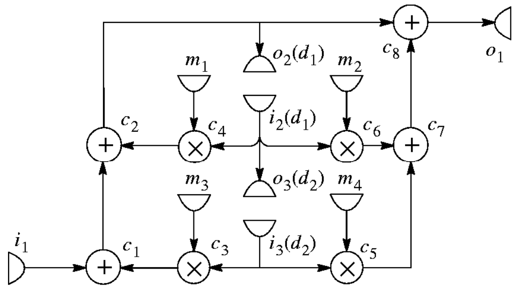
High Level Synthesis



Slides are by Courtesy of Prof. Y.-W. Chang

High Level Synthesis

- Course contents
  - Hardware modeling
  - Data flow
  - Scheduling/allocation/assignment
- Reading
  - Chapter 5

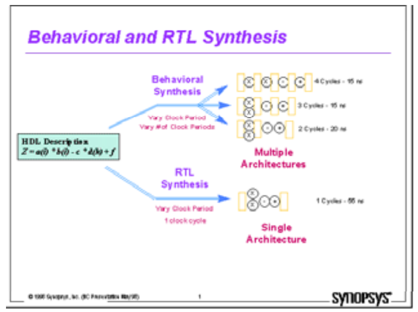


High Level Synthesis

- Hardware-description language (HDL) synthesis
  - Starts from a register-transfer level (RTL) description; circuit behavior in each clock cycle is fixed
  - Uses logic synthesis techniques to optimize the design
  - Generates a netlist
- High-level synthesis (HLS), also called architectural or behavioral synthesis
  - Starts from an abstract behavioral description
  - Generates an RTL description
  - It normally has to perform the trade-off between the number of cycles and the hardware resources to fulfill a task

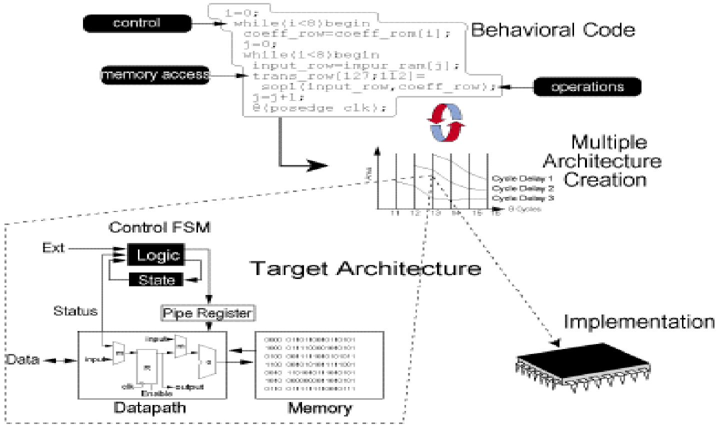
HL Synthesis vs. RTL Synthesis

- RTL synthesis implements all functionality within a single clock cycle
- HL synthesis automatically allocates the functionality across multiple clock cycles



Output of High Level Synthesis

- Behavioral Compiler creates a design that consists of a datapath, memory I/O and a control FSM



Benefits of High Level Synthesis (1)

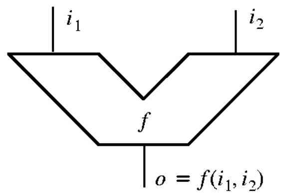
- Quick specification and verification
  - Specify behavioral HDL easily, since it's intuitive and natural to write
  - Save time -- behavioral HDL code is up to 10 times shorter than equivalent RTL
  - Simulate orders of magnitude faster because of the higher level of abstraction
  - Reuse designs more readily by starting with a more abstract description
- Reduce design time
  - Model hardware and software components of system concurrently
  - Easily implement algorithms in behavioral HDL and generate RTL code with a behavioral compiler
  - Verify hardware in system context at various levels of abstraction

Benefits of High Level Synthesis (2)

- Explore architectural trade-offs
  - Create multiple architectures from a single specification
  - Trade-off throughput and latency using high-level constraints
  - Analyze various combinations of technology-specific datapath and memory resources
  - Evaluate cost/performance of various implementations rapidly
- Automatically infer memory and generate FSM
  - Specify memory reads and writes
  - Schedule memory I/O, resolve conflicts by building control FSM
  - Trade-off single-ported (separate registers) vs. multi-ported memories (register files)
  - Generate a new FSM

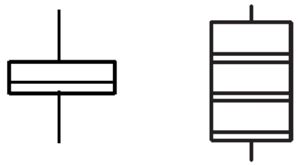
Hardware Models for HL Synthesis

- All HLS systems need to restrict the target hardware
  - Otherwise search space is too large
- All synthesis systems have their own peculiarities, but most systems generate synchronous hardware and build it with functional units:
  - A functional unit can perform one or more computations, e.g., addition, multiplication, comparison, ALU

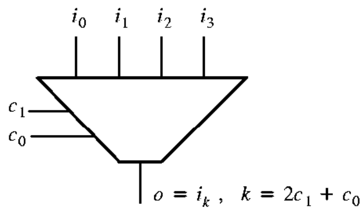


Hardware Models

- **Registers:** they store inputs, intermediate results and outputs; sometimes several registers are taken together to form a register file

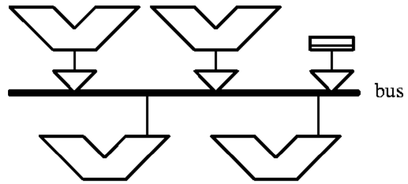


- **Multiplexers:** from several inputs, one is passed to the output

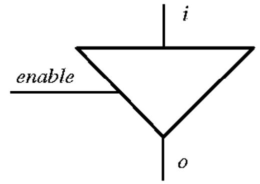


Hardware Models (cont'd)

- **Buses:** a connection shared between several hardware elements, such that only one element can write data at a specific time



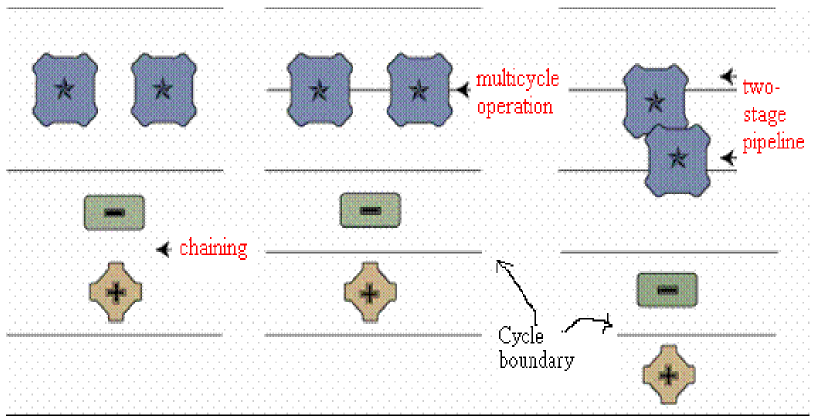
- **Tri-state drivers:** control the exclusive writing on the bus



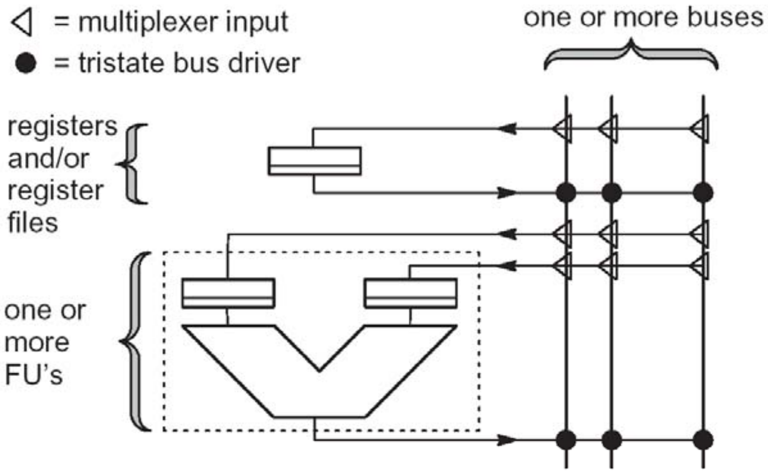
Hardware Models (cont'd)

- Parameters defining the hardware model for the synthesis problem:
  - **Clocking strategy:** e.g. single or multiple phase clocks
  - **Interconnect:** e.g. allowing or disallowing buses
  - **Clocking of functional units:** allowing or disallowing
    - multicycle operations
    - operation chaining (multiple operations in one cycle)
    - pipelined units

Chaining, Multicycle Operation, Pipelining



Example of a HLS Hardware Model



Hardware Concepts: Data Path + Control

- Hardware is normally partitioned into two parts:
  - **Data path:** a network of functional units, registers, multiplexers and buses
    - The actual "computation" takes place in the data path
  - **Control:** the part of the hardware that takes care of having the data present at the right place at a specific time, of presenting the right instructions to a programmable unit, etc.
- High-level synthesis often concentrates on data-path synthesis
  - The control part is then realized as a finite state machine or in microcode

Steps of High Level Synthesis

- **Preprocess** the design with high-level optimization
  - Code motion
  - Common subexpression elimination
  - Loop unrolling
  - Constant propagation
  - Modifications taking advantage of associativity and distributivity, etc.
- **Transform** the optimized design into intermediate format (internal representation) which reveals more structural characteristics of the design
- **Optimize** the intermediate format
  - Tree height reduction
  - Behavior retiming
- **Allocate** the required resources to implement the design
  - Also called module selection
- **Schedule** each operation to be performed at certain time such that no precedence constraint is violated
- **Assign (bind)** each operation to a specific functional unit and each variable to a register

HLS Optimization Criteria

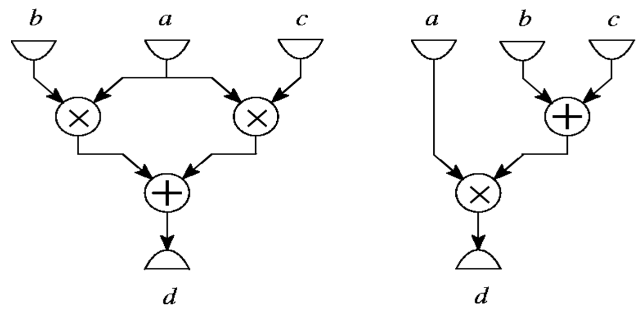
- Typically, in terms of speed, area, and power consumption
- Optimization is often constrained
  - Optimize area when the minimum speed is given ⇒ time-constrained synthesis
  - Optimize speed when a maximum for each resource type is given ⇒ resource-constrained synthesis
    - E.g. power-constrained synthesis
  - Minimize power dissipation for a given speed and area requirement ⇒ time- and area- constrained synthesis

Input Format

- The algorithm, which is the input to a high-level synthesis system, is often provided in textual form either
  - in a conventional programming language, such as C, C++, SystemC, or
  - in a hardware description language (HDL), which is more suitable to express the parallelism present in hardware.
- The description has to be parsed and transformed into an internal representation and thus conventional compiler techniques can be used.

Example of HL Optimization

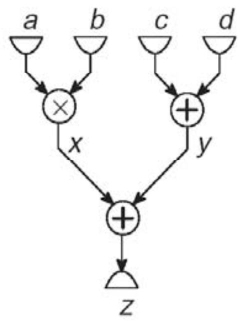
- Applying the distributive law to reduce resource requirement



Internal Representation

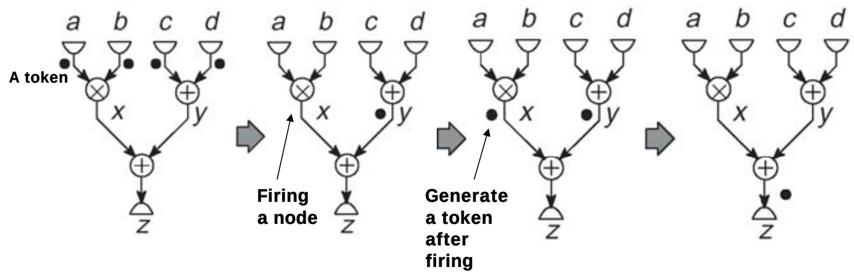
- Most systems use some form of a data-flow graph (DFG)
  - A DFG may or may not contain information on control flow
- A data-flow graph is built from
  - vertices (nodes): representing computation, and
  - edges: representing precedence relations

```
x := a * b; y := c + d;
z := x + y;
```



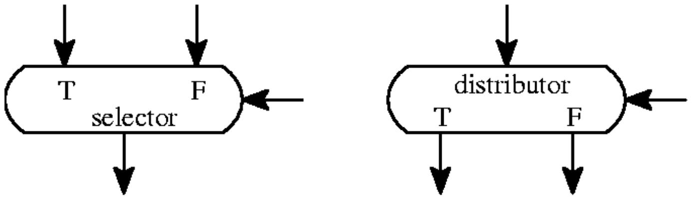
Token Flow in a DFG

- A node in a DFG fires when all tokens are present at its inputs
- The input tokens are consumed and an output token is produced (like in Petri nets)



Conditional Data Flow

- Conditional data flow by means of two special nodes:

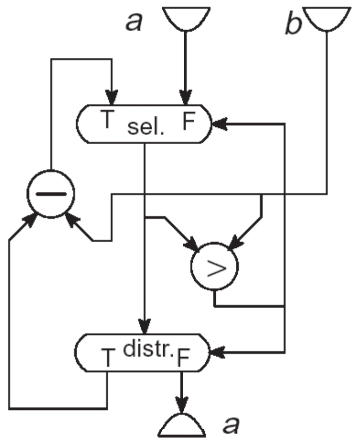


Explicit Iterative Data Flow

- Selector and distributor nodes can be used to describe iteration
  - Example

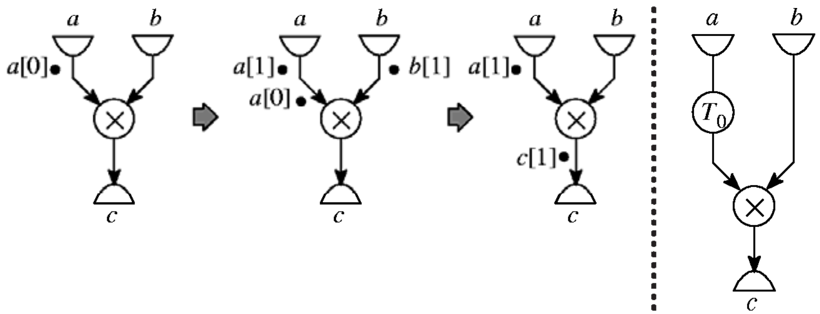
```
while (a > b)
    a ← a - b;
```

- Loops require careful placement of initial tokens on edges



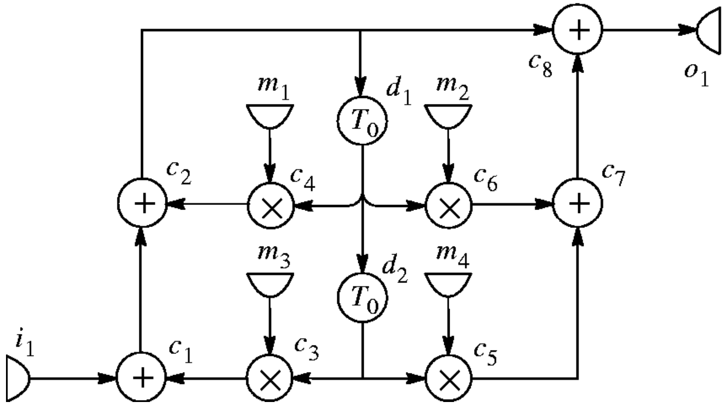
Implicit Iterative Data Flow

- Iteration implied by regular input stream of tokens
- Initial tokens act as buffers
- Delay elements instead of initial tokens



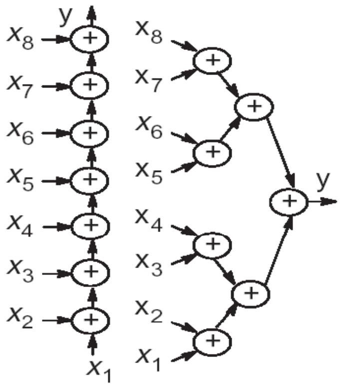
Iterative DFG Example

a second-order filter section



Optimization of Internal Representation

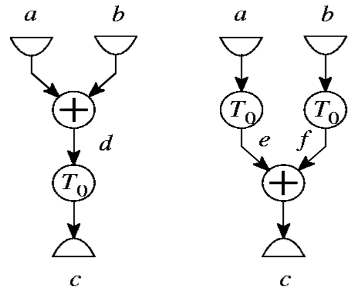
- Restructuring data and control flow graphs prior to the actual mapping onto hardware
  - Examples:
    - Replacing chain of adders by a tree
    - Behavior retiming

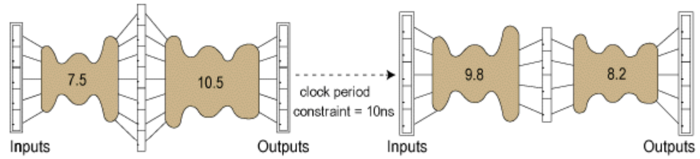


Tree height reduction

Behavior Retiming (BRT)

- By moving registers through logic and hierarchical boundaries, BRT reduces the clock period with minimum area impact





Effectiveness of Behavior Retiming

Synopsys exp:

	RTL Design		Behavioral Retiming		
Design Type	Speed (Clock Period)	Gates	Speed (Clock Period)	Gates	Summary
Control	44 ns	10,913 gates	30.6 ns	11,313 gates	30% faster, 4% more area
Control	23.1 ns	3,598 gates	19.6 ns	4,575 gates	15% faster, 27% more area
Control	28.6 ns	3,585 gates	28.6 ns	3,359 gates	same speed, 6% less area
Dataflow & Control	17 ns	28,900 gates	12.5 ns	30,100 gates	26% faster, 4% more area
Dataflow & Control	16 ns	7,620 gates	13 ns	8,019 gates	20% faster, 5% more area
Dataflow	22 ns	4,990 gates	18.5 ns	5,109 gates	16% faster, 2% more area
Dataflow	28 ns	31,226 gates	26 ns	32,032 gates	8% faster, 2% more area
Dataflow	26.2 ns	14,351 gates	23.6 ns	13,847 gates	10% faster, 4% less area
Dataflow	25.9 ns	16,798 gates	20.8 ns	15,550 gates	20% faster, 7% less area
Dataflow	45 ns	28,705 gates	26 ns	30,987 gates	42% faster, 8% more area

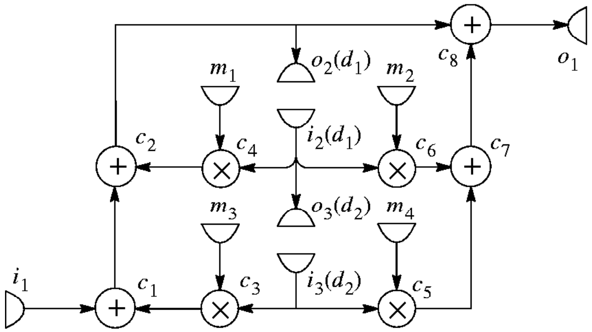
- RTL designs have a single clock net and were synthesized into gates using Synopsys Design Compiler
- Design type: dataflow implies significant number of operators; control implies state machine dominated

HLS Subtasks: Allocation, Scheduling, Assignment

- Subtasks in high-level synthesis
  - **Allocation (Module selection):** specify the hardware resources that will be necessary
  - **Scheduling:** determine for each operation the time at which it should be performed such that no precedence constraint is violated
  - **Assignment (Binding):** map each operation to a specific functional unit and each variable to a register
- Remarks:
  - Though the subproblems are strongly interrelated, they are often solved separately. However, to attain a better solution, an iterative process executing these three subtasks must be performed.
  - Most scheduling problems are NP-complete $\Rightarrow$ heuristics are used

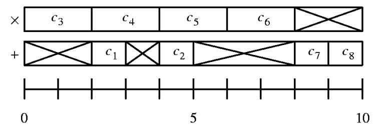
Example of High Level Synthesis

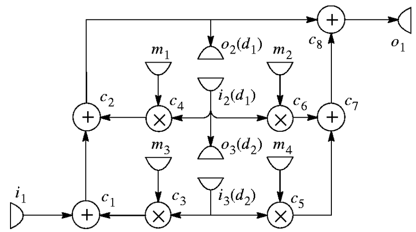
- The following couple of slides shows an example of scheduling and binding of a design with given resource allocation
- Given the second-order filter which is first made acyclic:



Example of Scheduling

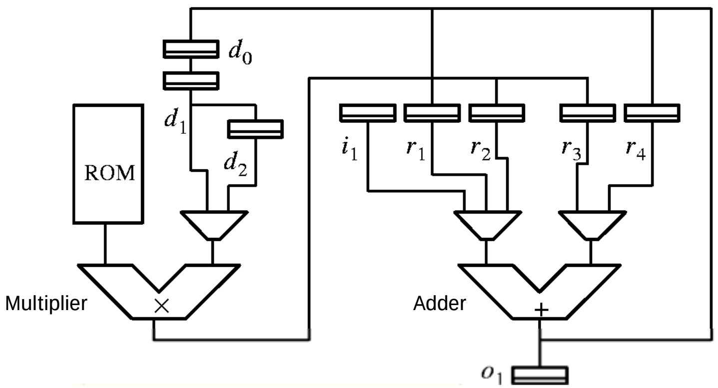
- The schedule and operation assignment with an allocation of one adder and one multiplier:





Binding for Data Path Generation

- The resulting data path after register assignment
  - The specification of a controller would complete the design



Resource Allocation Problem

- This problem is relatively simple. It simply decides the kinds of hardware resources (hardware implementation for certain functional units such as adder, multiplier, etc.) and the quantity of these resources.
  - For example two adders, one multiplier, 4 32-bit registers, etc. for a certain application
- The decision made in this step has a profound influence on the scheduling which under the given resource constraints decides the time when an operation should be executed by a functional unit
- This step set an upper bound on the attainable performance.

Problem Formulation of Scheduling

- Input consists of a DFG $G(V, E)$ and a library $\mathfrak{R}$ of resource types
- There is a fixed mapping from each $v \in V$ to some $r \in \mathfrak{R}$; the execution delay $\delta(v)$ for each operation is therefore known
- The problem is time-constrained; the available execution times are in the set

$$\mathcal{T} = \{0, 1, \ldots, T_0 - 1\}.$$

- A schedule $\sigma: V \rightarrow T$ maps each operation to its starting time; for each edge $(v_i, v_j) \in E$, a schedule should respect: $\sigma(v_j) \geq \sigma(v_i) + \delta(v_i)$.
- Given the resource type cost $\omega(r)$ and the requirement function $N_r(\sigma)$, the cost of a schedule σ is given by:

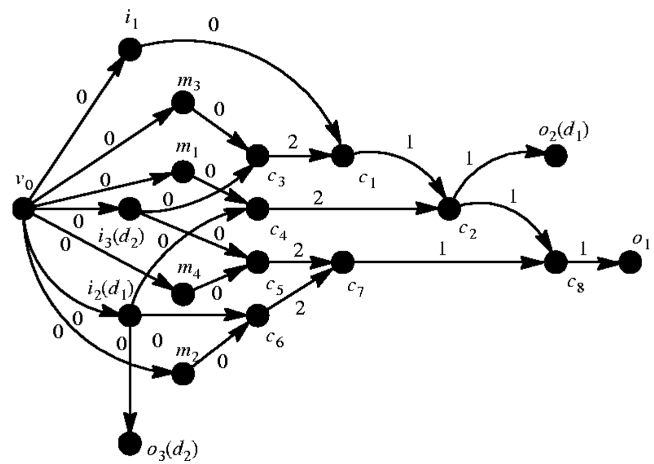
$$\sum_{r \in \mathfrak{R}} \omega(r) N_r(\sigma).$$

ASAP Scheduling

- **As soon as possible (ASAP) scheduling** maps an operation to the earliest possible starting time not violating the precedence constraints

- Properties:
  - It is easy to compute by finding the longest paths in a directed acyclic graph
  - It does not make any attempt to optimize the resource cost

Graph for ASAP Scheduling



Mobility Based Scheduling

- Compute both the ASAP and ALAP (as late as possible) schedules σ_S and σ_L

- For each $v \in V$, determine the scheduling range $[\sigma_S(v)\ ,\ \sigma_L(v)]$

- $\sigma_L(v) - \sigma_S(v)$ is called the mobility of v

- Mobility-based scheduling tries to find the best position within its scheduling range for each operation

Simple Mobility Based Scheduling

- A partial schedule $\tilde{\sigma} : V \rightarrow [\mathcal{T}, \mathcal{T}]$ assigns a scheduling range to each $v \in V$,

$$\tilde{\sigma}(v) = \left[\tilde{\sigma}_{\min}(v), \tilde{\sigma}_{\max}(v)\right]$$

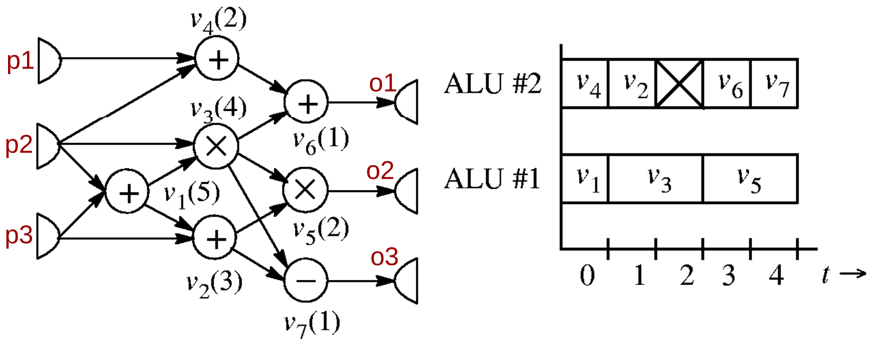
- Finding a schedule can be seen as the generation of a sequence of partial schedules $\tilde{\sigma}^{(0)}, \ldots, \tilde{\sigma}^{(n)}$.

```
"determine σ̃(0) by computing σS and σL";
k ← 0;
while ("there are unscheduled operations") {
    v ← "one of the nodes with lowest mobility";
    "schedule v at some time that optimizes the current resource utilization";
    "determine σ̃(k+1) by updating the scheduling ranges
     of the unscheduled nodes";
    k ← k + 1
}
```


List Scheduling

- A resource-constrained scheduling method
- Start at time zero and increase time until all operations have been scheduled
  - Consider the precedence constraint
- The ready list L_t contains all operations that can start their execution at time *t* or later
- If more operations are ready than there are resources available, use some priority function to choose, e.g. the longest-path to the output node $\Rightarrow$ critical-path list scheduling

List Scheduling Example

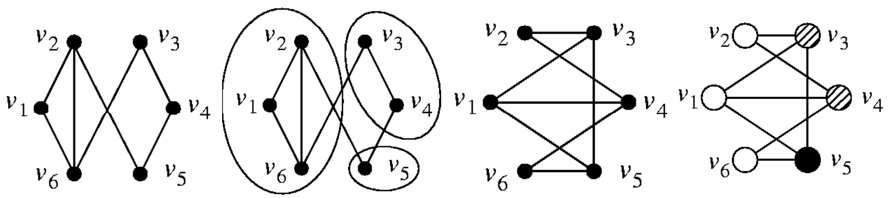


Assignment Problem

- Subtasks in assignment:
  - operation-to-FU assignment
  - value grouping
  - value-to-register assignment
  - transfer-to-wire assignment
  - wire to FU-port assignment
- In general: task-to-agent assignment

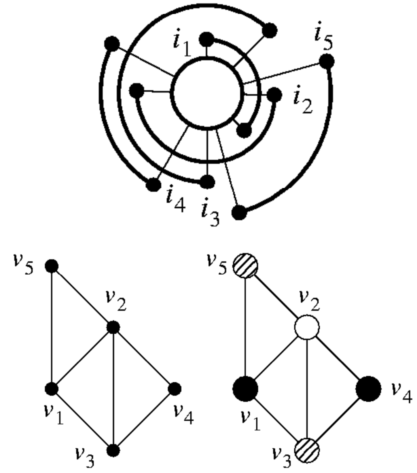
Compatibility and Conflict Graphs

- Clique partitioning gives an assignment in a compatibility graph
- Graph coloring gives an assignment in the complementary conflict graph



Assignment Problem

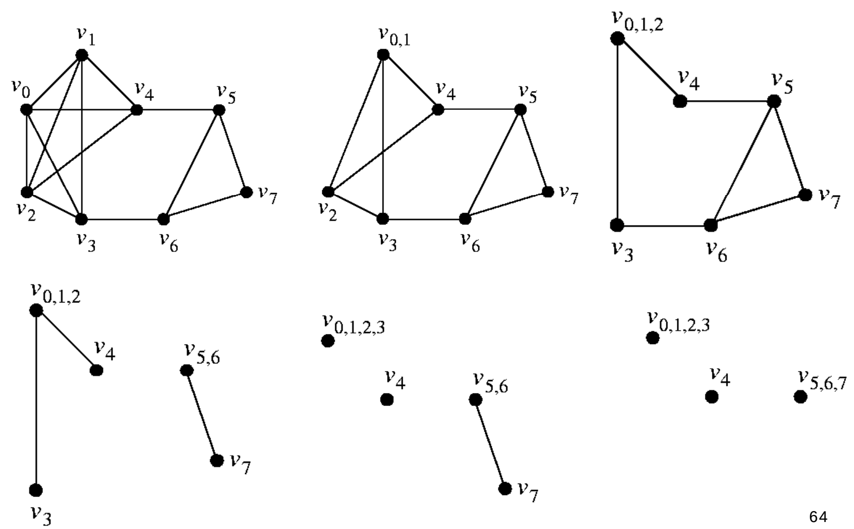
- Assumption: assignment follows scheduling.
- The claim of a task on an agent is an interval ⇒ minimum resource utilization can be found by left-edge algorithm.
- In case of iterative algorithm, interval graph becomes circular-arc graph ⇒ optimization is NP-complete.



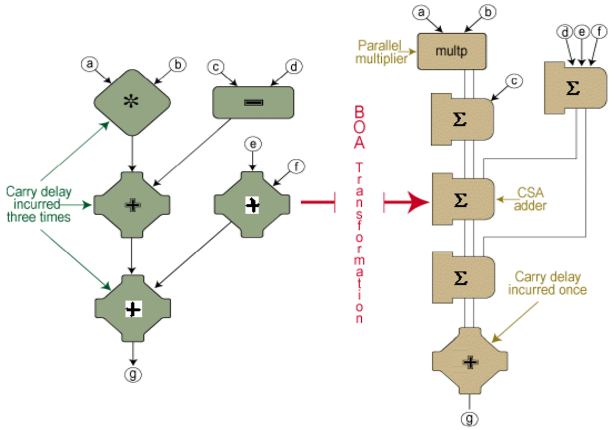
Tseng and Sieworek's Algorithm

$k \leftarrow 0$;
$G_c^k(V_c^k, E_c^k) \leftarrow G_c(V_c, E_c)$;
while $(E_c^k \neq \emptyset)$ {
  "find $(v_i, v_j) \in E_c^k$ with largest set of common neighbors";
  $N \leftarrow$ "set of common neighbors of v_i and v_j";
  $s \leftarrow i \cup j$;
  $V_c^{k+1} \leftarrow V_c^k \cup \{v_s\} \setminus \{v_i, v_j\}$;
  $E_c^{k+1} \leftarrow \emptyset$;
  for each $(v_m, v_n) \in E_c^k$
    if $(v_m \neq v_i \wedge v_m \neq v_j \wedge v_n \neq v_i \wedge v_n \neq v_j)$
      $E_c^{k+1} \leftarrow E_c^{k+1} \cup \{(v_m, v_n)\}$;
  for each $v_n \in N$
    $E_c^{k+1} \leftarrow E_c^{k+1} \cup \{(v_n, v_s)\}$;
  $k \leftarrow k + 1$;
}

Clique-Partitioning Example



Example of Behavior Optimization



Behavior Optimization of Arithmetic Circuit (BOA)

Effectiveness of BOA

Synopsys example

Design Type	RTL Design	BOA	Summary
Motion estimation	23.6 ns 12,793 gates	20.2 ns 12,215 gates	14% faster, 5% less area
Graphics interpolation	19.2 ns 3,507 gates	17.5 ns 2,952 gates	9% faster, 16% less area
Color space conversion and scaling	16 ns 35,866 gates (manual CSA implementation)	14.9 ns 34,397 gates	7% faster, 4% less area
Sum of 9 operands	7.7 ns 1,418 gates	5.3 ns 1,307 gates	31% faster, 8% less area
a * b + 1	11.6 ns 2,577 gates	9.3 ns 2,524 gates	20% faster, 2% less area
a * 4104 (0100000100000100)	4.4 ns 759 gates	3.1 ns 449 gates	30% faster, 40% less area
a * 3E3E (0011111000111110)	5.7 ns 927 gates	4.6 ns 709 gates	19% faster, 23% less area
a * b + c	11.0 ns 2,707 gates	10.0 ns 2,689 gates	9% faster, same area
a * b + c * d + e * f	14.2 ns 7,435 gates	12.8 ns 7,110 gates	10% faster, 4% less area
Sum of 16 operands	8.1 ns 2,836 gates	6.7 ns 2,123 gates	17% faster, 25% less area