High-Level Synthesis: from theory to practice

Philippe COUSSY

philippe.coussy@univ-ubs.fr

2019, May 21

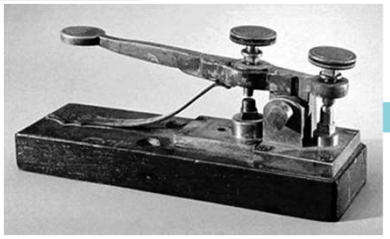
Université Bretagne Sud

ARCHI 2019 Lorient

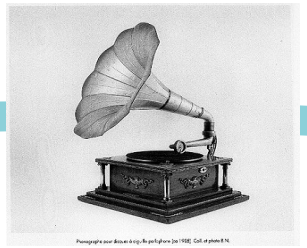
Lab-STICC

cnrs

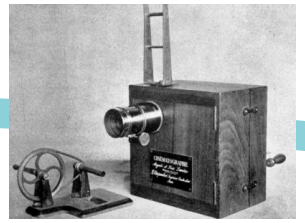
Evolution of ICT



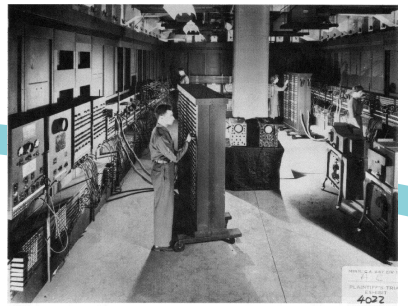
1840



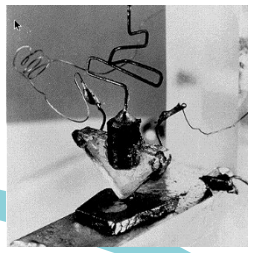
1877



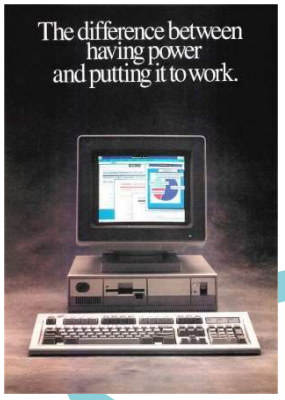
1895



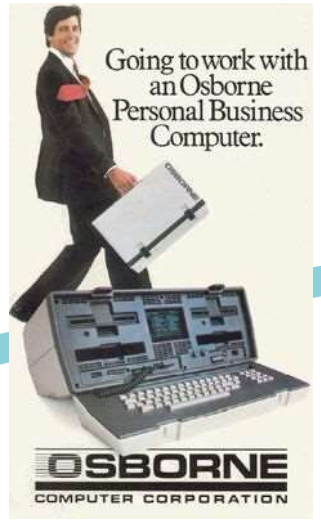
1946



1947



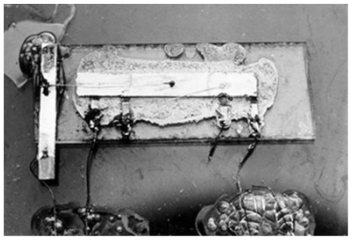
1990



1981

1973

1966



1958

1990

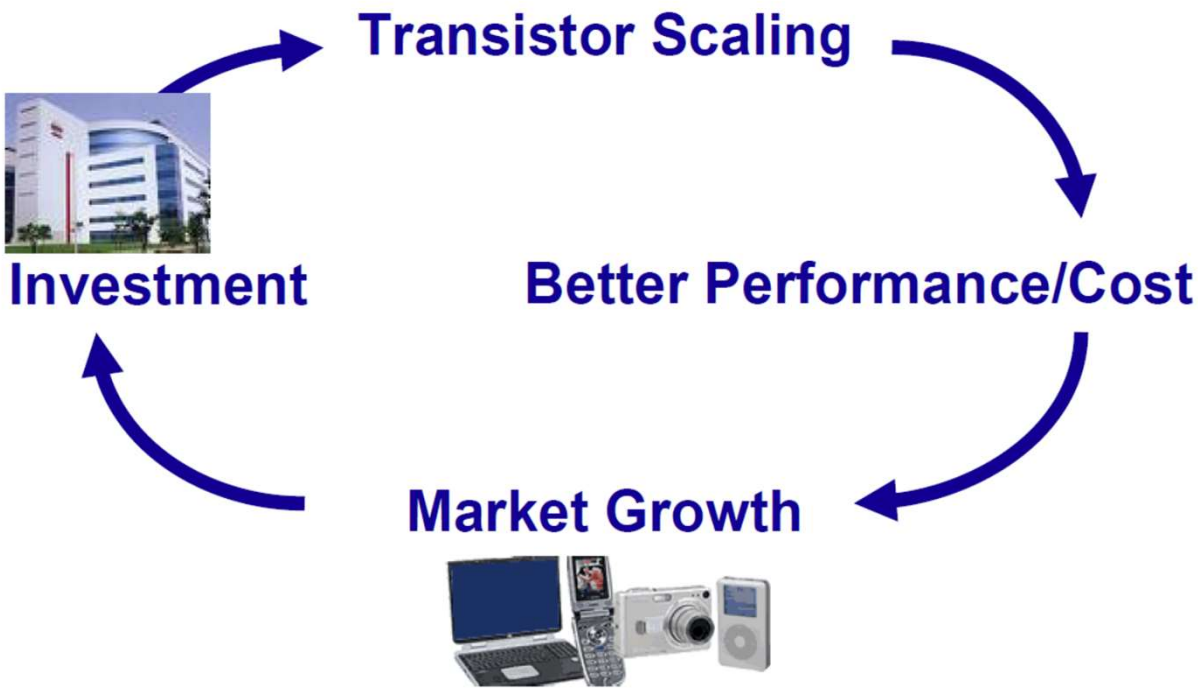
1990

2000

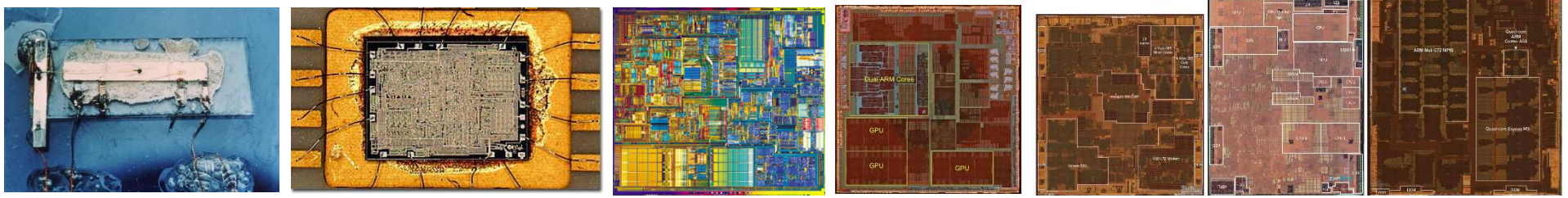
2010

2

The virtuous circle



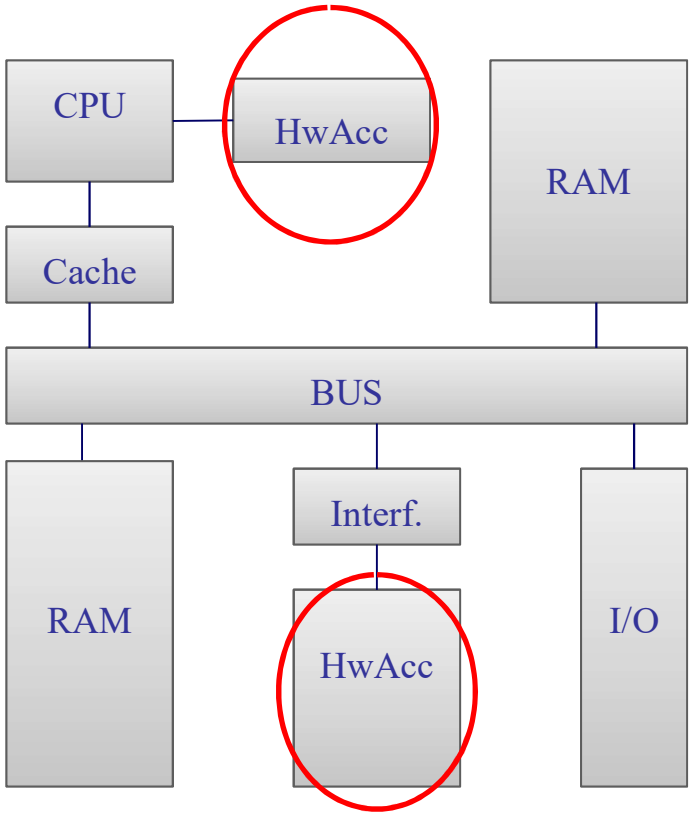
The virtuous circle of the semiconductor industry (source: ITRS)



IC Trends

- Complexity ↑
- Performance ↑
- Flexibility ↑
- Reliability ↑
- Security ↑
- NRJ and Power Consumption ↓
- Cost ↓
- Time to market ↓
- …

Typical SoC architecture



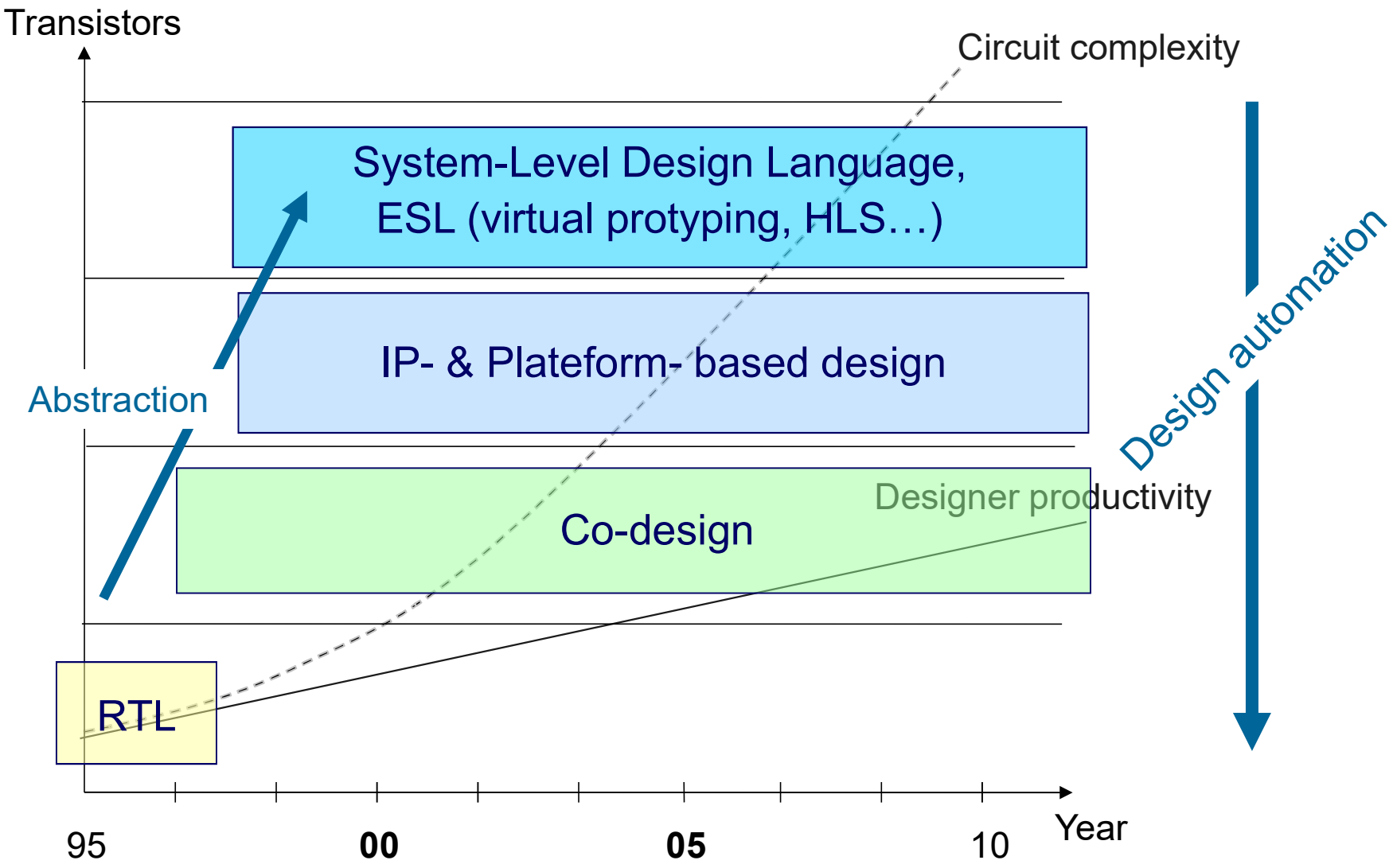
CPU: Central Processing Unit
RAM: Random Access Memory
HwAcc: Hardware Accelerator
I/O: Input / Output

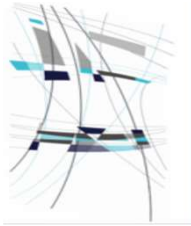
Hardware Accelerators are
- non programmable processing units

- still needed for
  + Timing performances
  + Power/NRJ consumption

- tightly or loosely coupled

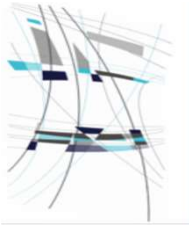
Electronic System Level Design (ESLD)





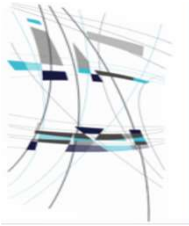
Design methodologies

- Synthesis and verification automation has always been key factors in the evolution of the design process
  - Allow to explore the design space efficiently and rapidly
  - Correct by construction design



Design methodologies

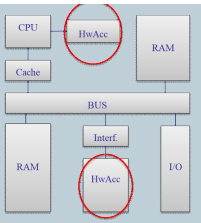
- Software domain
  - Machine code (binary sequence)
  - 1950s: concept of assembly language (and assembler)
    - based on mnemonics
    - Maurice V. Wilkes de l'université de Cambridge
  - Later: High-level languages and compilers
    - 1951: First compiler
      - (A-0 system) par Grace Hopper
    - Fortran 1954-1957: First high-level language
      - FORmula TRANslator
    - Cobol 1959, Basic 1964, C 1972, C++ 1983, Java 1995…



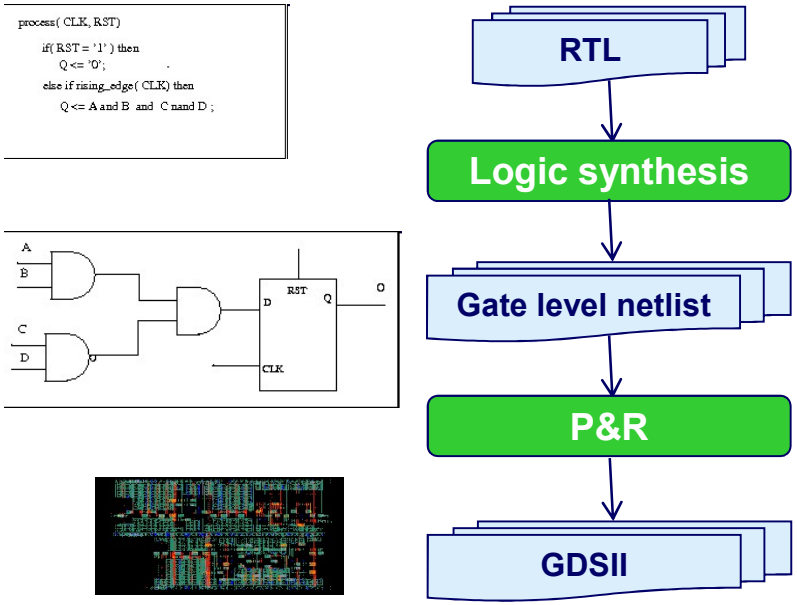
Design methodologies (1/2)

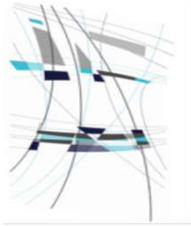
- Hardware domain
  - 1960: “hand-made” era
    - design, optimization, layout
  - 1970: Gate-level simulation
  - end of 70: Cycle-based simulation
  - 1980: Wide automation
    - place & route, schematic circuit capture, formal verification and static timing analysis
  - Mid 1980: Hardware description language
    - 1986 Verilog, 1987 VHDL
  - 1990: logic synthesis
    - VHDL and Verilog synthesizable subsets

Typical HW design flow



- From **Register Transfer Level** (RTL) down to IC layout

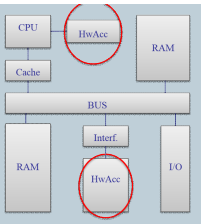




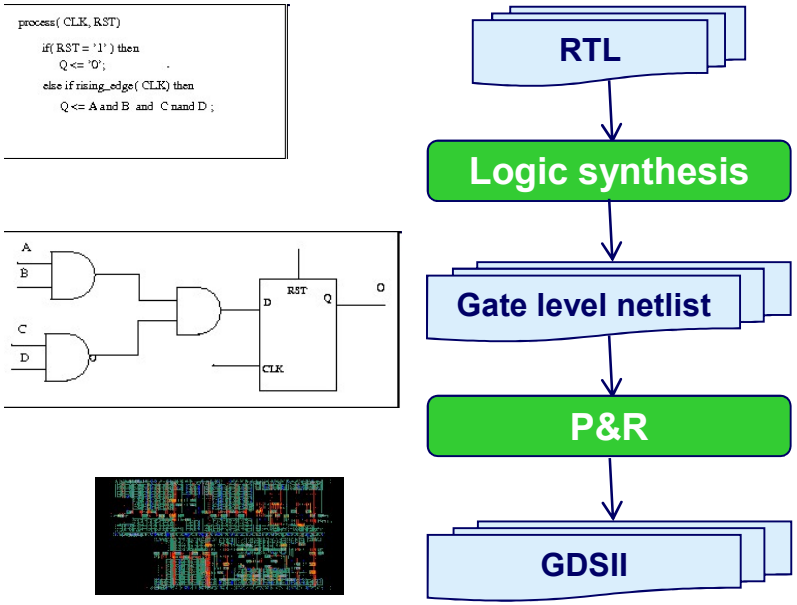
Design methodologies (2/2)

- Mid 80s:
  - High-level synthesis (First gen.)
    - Mainly research domain
- Mid 1990:
  - High-level synthesis (Second gen.)
    - e.g. Behavioural Compiler (Syn.), Monet (Mentor.)…
  - Co-design, IP-core reuse…
- 2000 : Electronic System Level ESL
  - System level language
    - SystemC, SystemVerilog…,
  - Virtual prototyping, Transaction Level Modelling TLM
  - High-level Synthesis (Third gen.)
  - …

Typical HW design flow

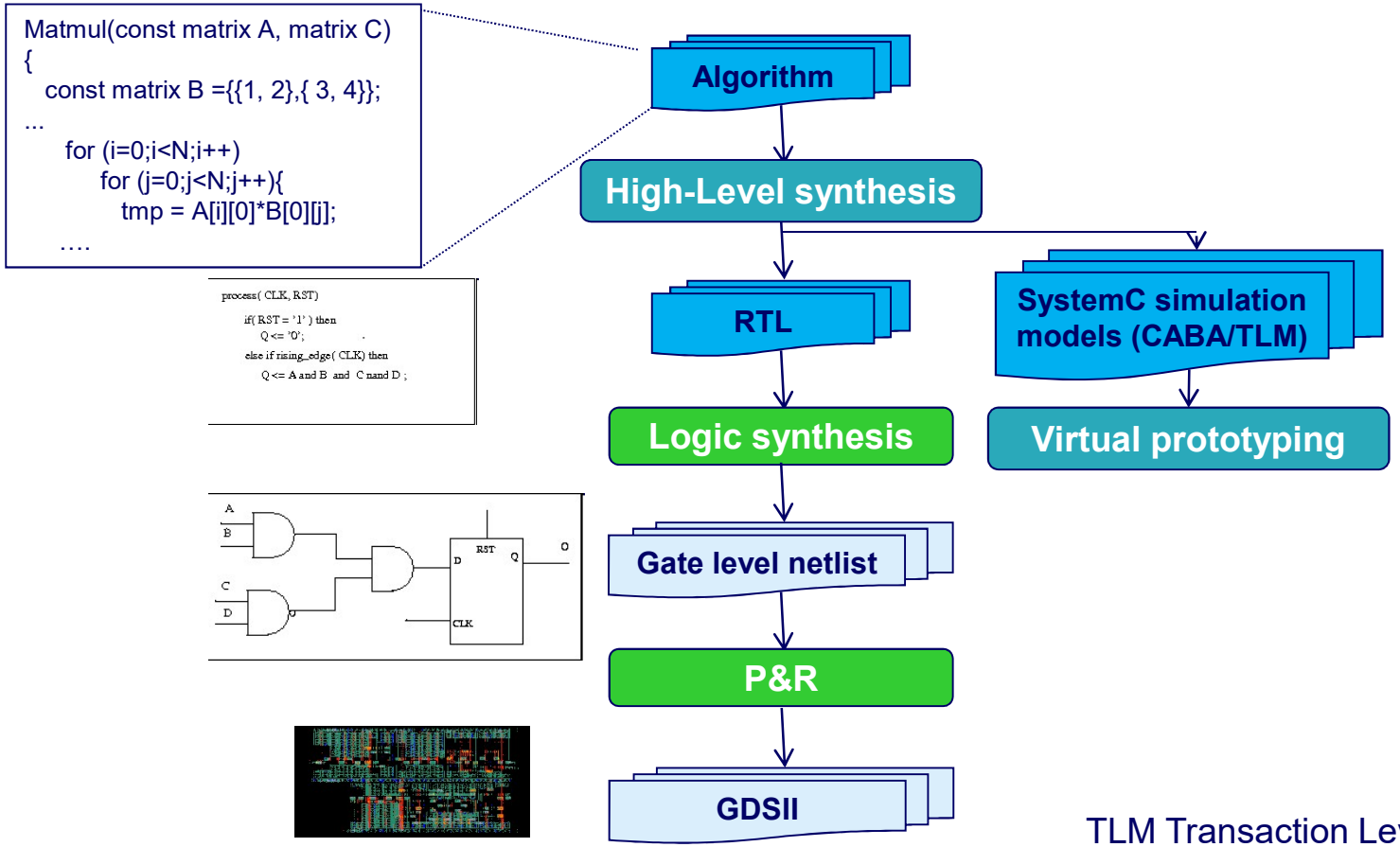


- From **Register Transfer Level** (RTL) down to IC layout

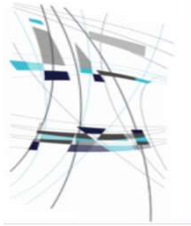


HLS-based HW design flow

- From **Algorithm** down to **TLM, CA** and **RTL** models



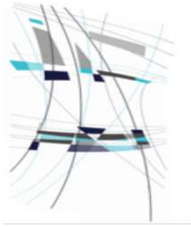
TLM Transaction Level Model
CABA Cycle Accurate Bit Accurate



Outline

- Context
- HLS basics
  - Modeling
  - Scheduling
  - Binding
- Conclusion

- The GAUT tool
  - Overview
  - Practice (rooms 107/108)

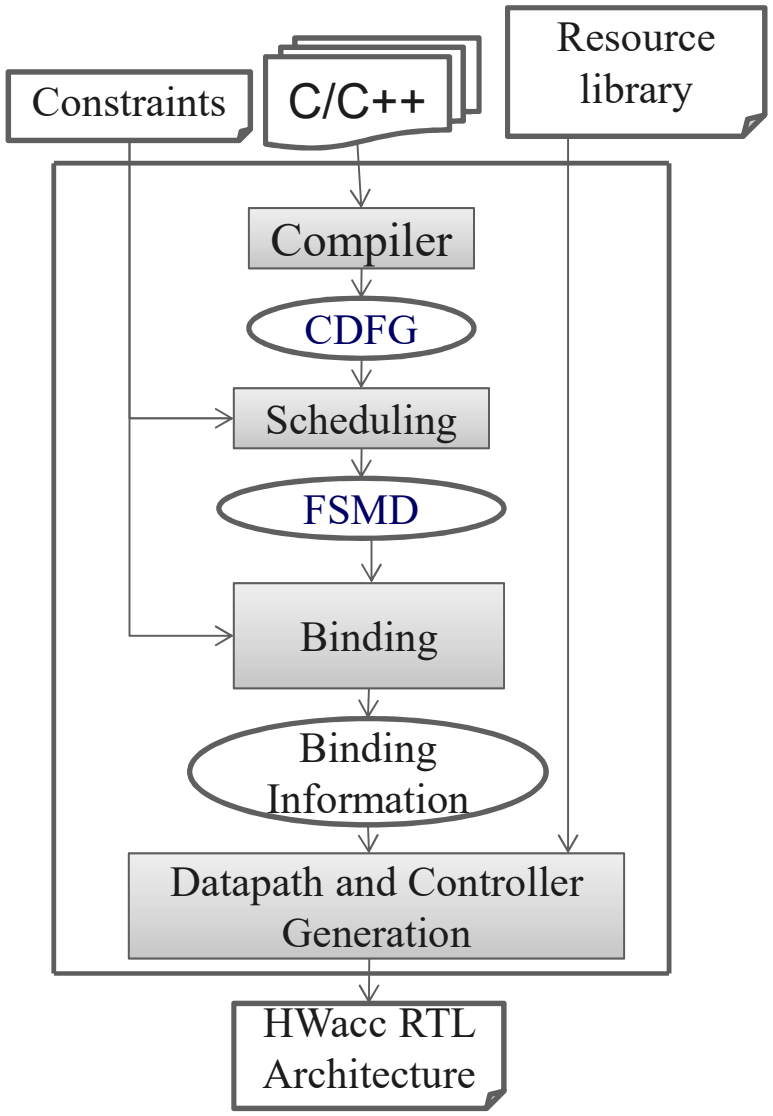


High-Level Synthesis (HLS)

- Starting from a functional description, automatically generate an RTL architecture

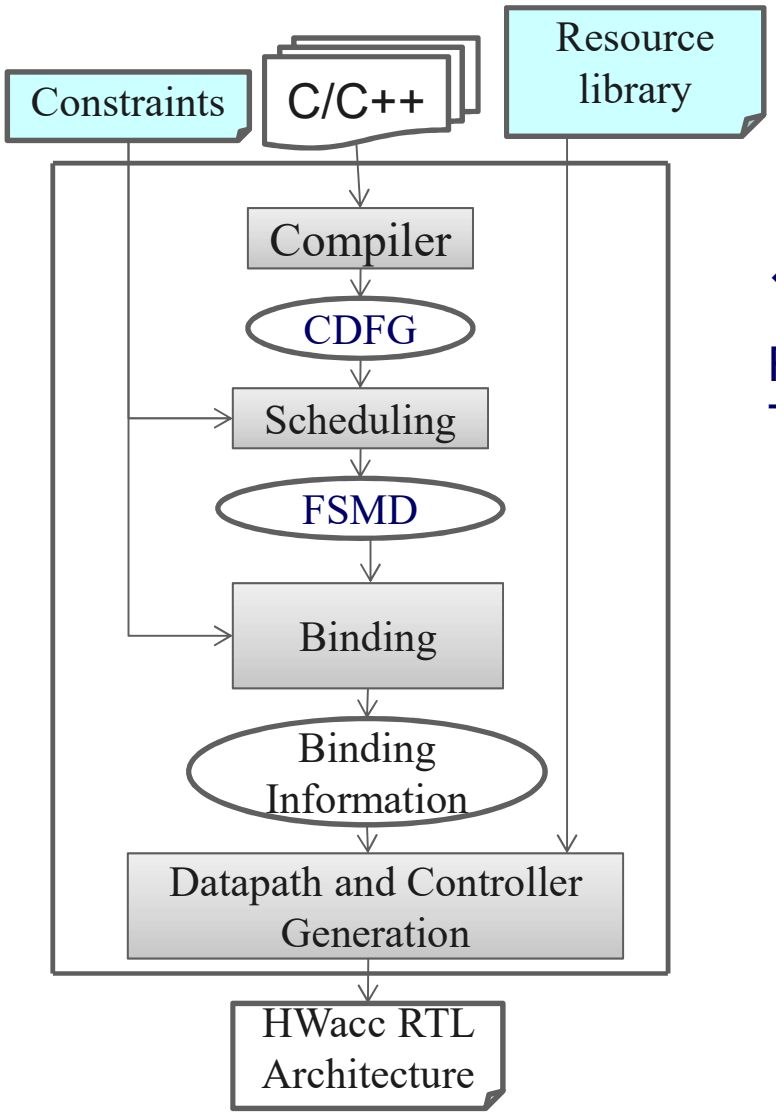
- Constraints
  - Timing constraints: latency and/or throughput
  - Resource constraints: #Operators and/or #Registers and/or #Memory, #Slices...
  - …
- Objectives
  - Minimization: area i.e. resources, latency, power consumption…
  - Maximization: throughput, clock frequency…
  - …

Typical HLS flow



- **Compilation** generates a formal modeling of the algorithmic input description
- **Scheduling** defines the execution time of each operation
- **Binding** defines which operator executes a given operation and which memory element stores a given data
- **Allocation** defines the number and the type of each resource
- **Architecture generation** writes out the RTL source code…

HLS inputs (1/2)



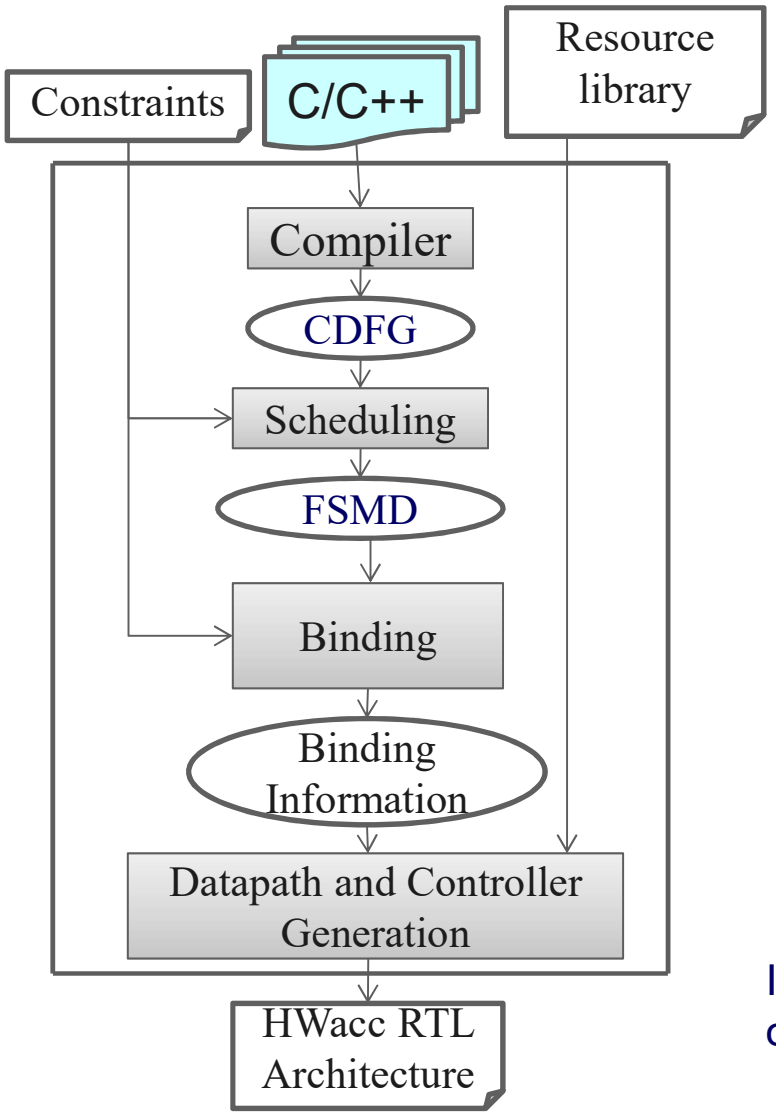
❖ **Constraints (resp. objectives):**

Resources: 1 adder, 1 multiplier, 1 memory bank
Timing : throughput, latency, clock period…

❖ **Resource library:**

The set of characterized functional units/ operators/components used to design the architecture (≠techno, ≠target, ≠architecture…)

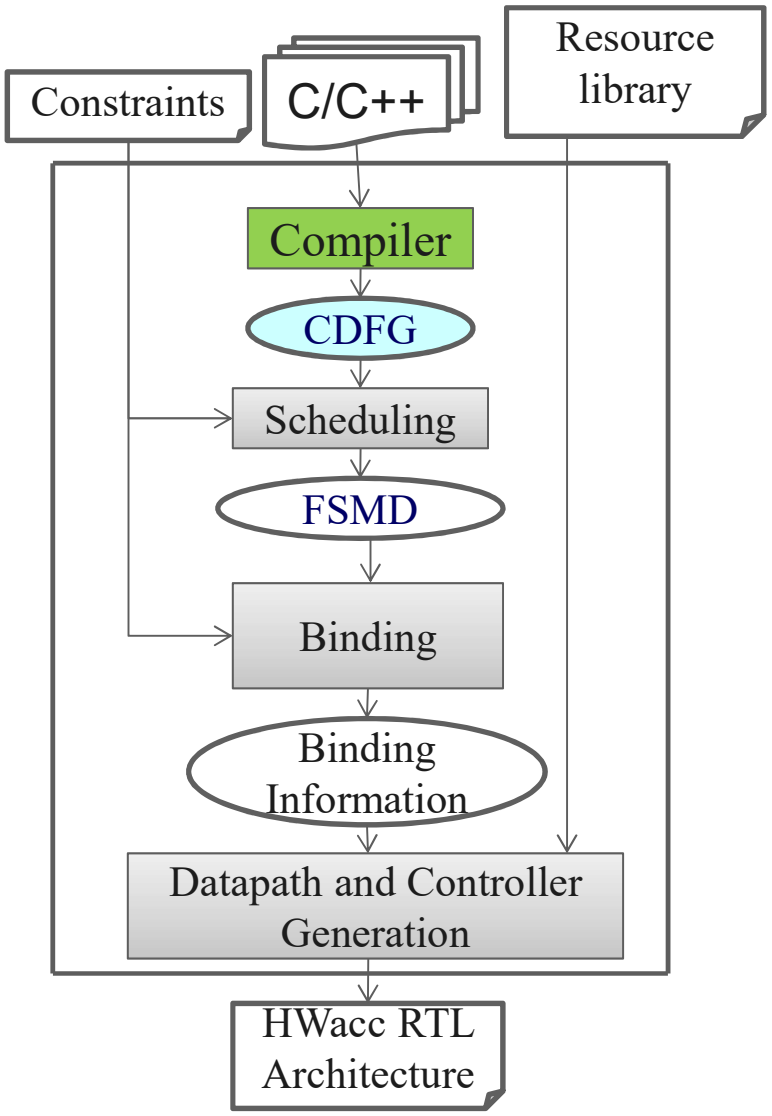
HLS inputs (2/2)




```
Void Filter (int N, int C[N], int X[N], int Y[N]){
   (1)  int i,j;
   (2) for (j =0; j<N; j++){ // loop1
   (3)      Y[j] = 0;
   (4)     for (i=0; i<N, i++){// loop2
   (5)          Y[j]= Y[j] + C[i]*X[N-1-i];
   (6)       }
   (7)    }
}
```

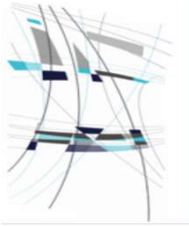
Input descriptions can also use bit-accurate data types or specify timing references (wait statements…)

Application modeling (1/3)



Compilation realizes optimizations and generates a formal modeling of the algorithmic input description

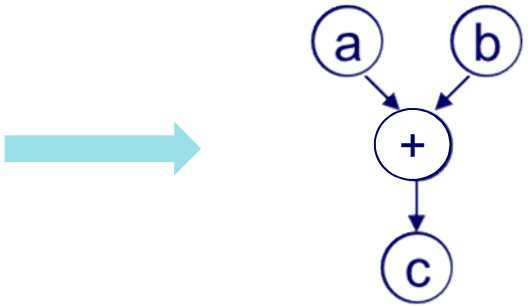
- **Dataflow Graph (DFG)**
- **Control Flow Graph (CFG)**
- **Control and Data Flow Graph (CDFG)**
- **Signal Flow Graph (SFG)**
- **Abstract Syntax Tree (AST)**
- **Petri Net**
- **Polyhedral model**
- **…**

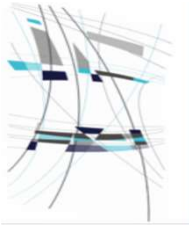


DataFlow Graph (1/2)

- Exhibits the parallelism between operations
  - Through data dependencies
    - Two types of nodes: Variable & Operation

c = a + b;



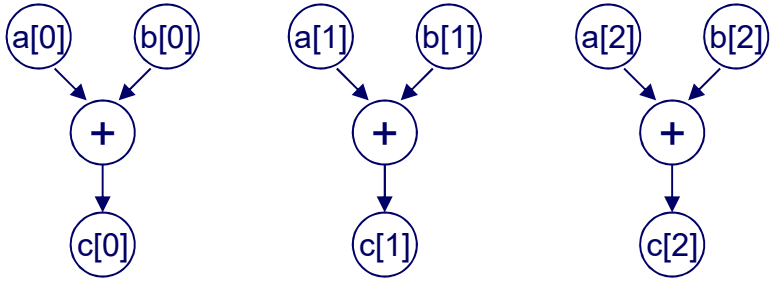


DataFlow Graph (DFG) (1/2)

- Exhibits the parallelism between operations
  - Through data dependencies
    - Two types of nodes: Variable & Operation
- Control structures are not supported
  - Loops are completely unrolled

```
for i : 0 → 2
    c[i] = a[i] + b[i]
```

```
c[0] = a[0] + b[0]
c[1] = a[1] + b[1]
c[2] = a[2] + b[2]
```

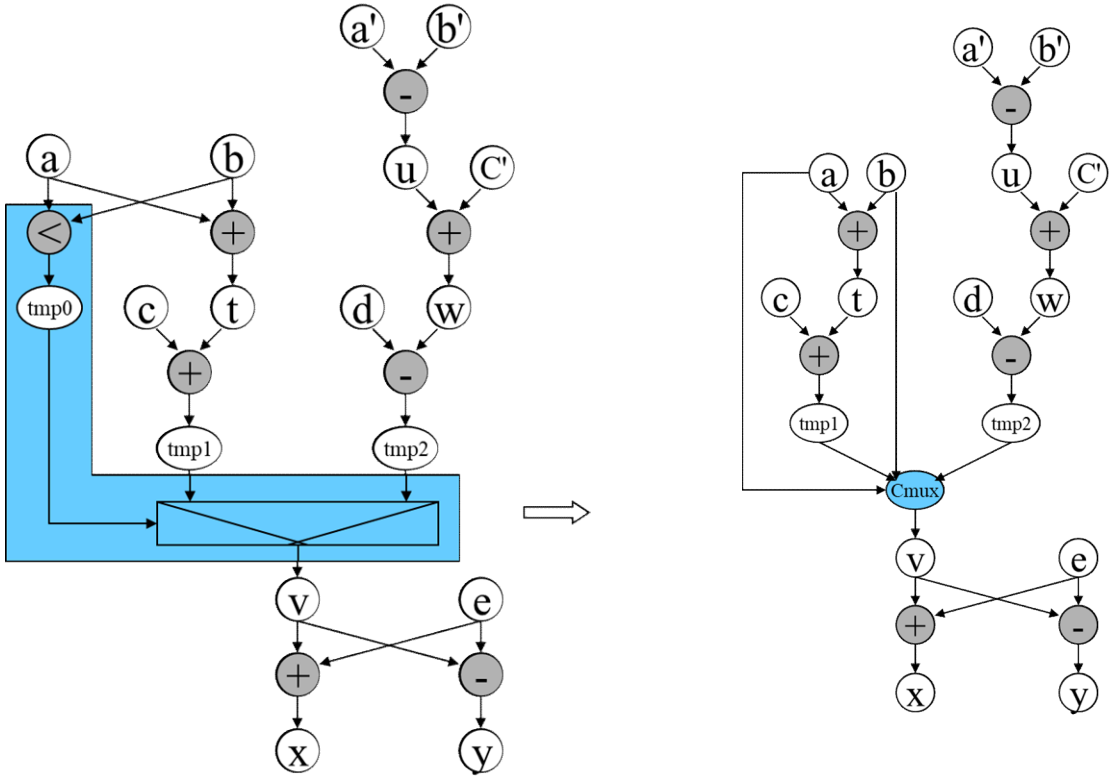


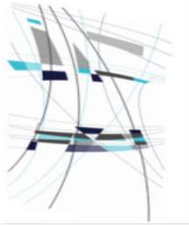
Data Flow Graph (DFG) (2/2)

- Conditional assignments are transformed
  - i.e. *if/switch* constructs, are resolved by creating multiplexed values

```
1: t = a+b;
2: u = a'-b';
3: if (a<b)
4:     v = t+c;
else
{
5:     w = u+c';
6:     v = w+d;
}
7: x = v+e;
8: y = v-e;
```

Source code



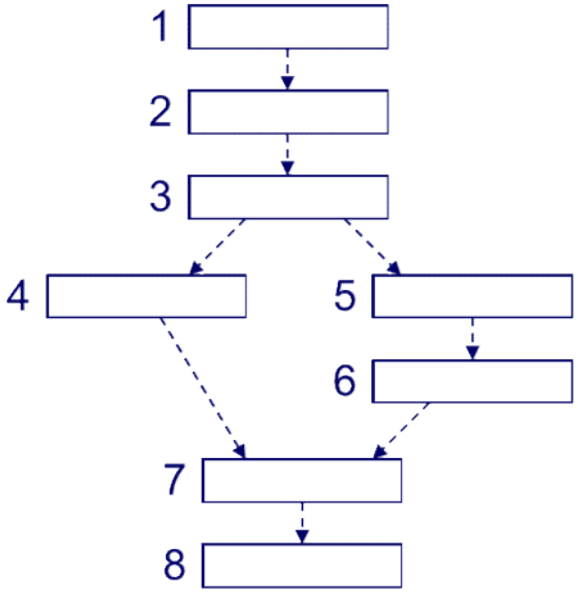


Control Flow Graph (CFG)

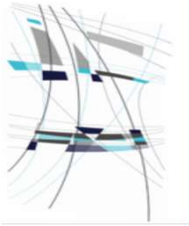
- Exhibits operation sequences
  - through control dependencies
- The sequence of operations comes directly from the source code and is kept unchanged

```
1: t = a+b;
2: u = a'-b';
3: if (a<b)
4:     v = t+c;
else
{
5:     w = u+c';
6:     v = w+d;
}
7: x = v+e;
8: y = v-e;
```

Source code

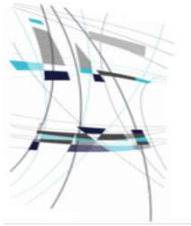


Graphical representation



Control & Data Flow Graph (CDFG)

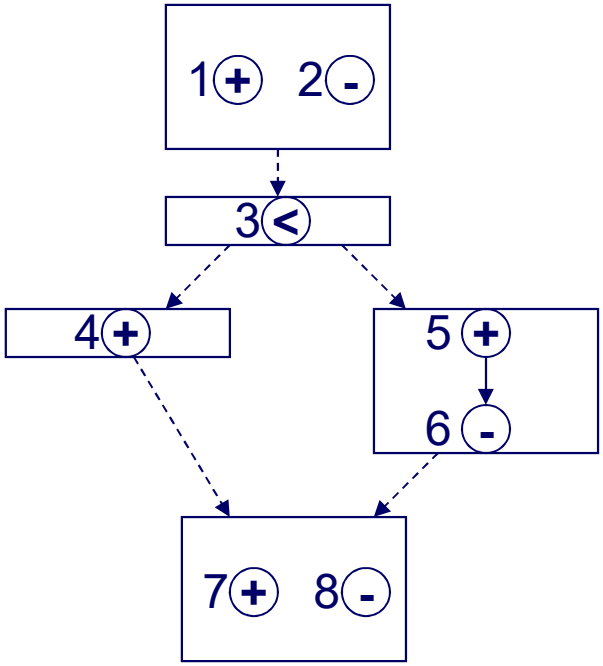
- Combination of DFG and CFG
- CFG is used to exhibit the sequence of operations
  - between « basic blocs » *BB*
- « Basic Bloc » (BB)
  - A linear sequence of instruction without any control operation (for, if, while…)
- DFG is used to exhibit the parallelism between operations
  - in the « basic blocs » *BB*



Control & Data Flow Graph (CDFG)

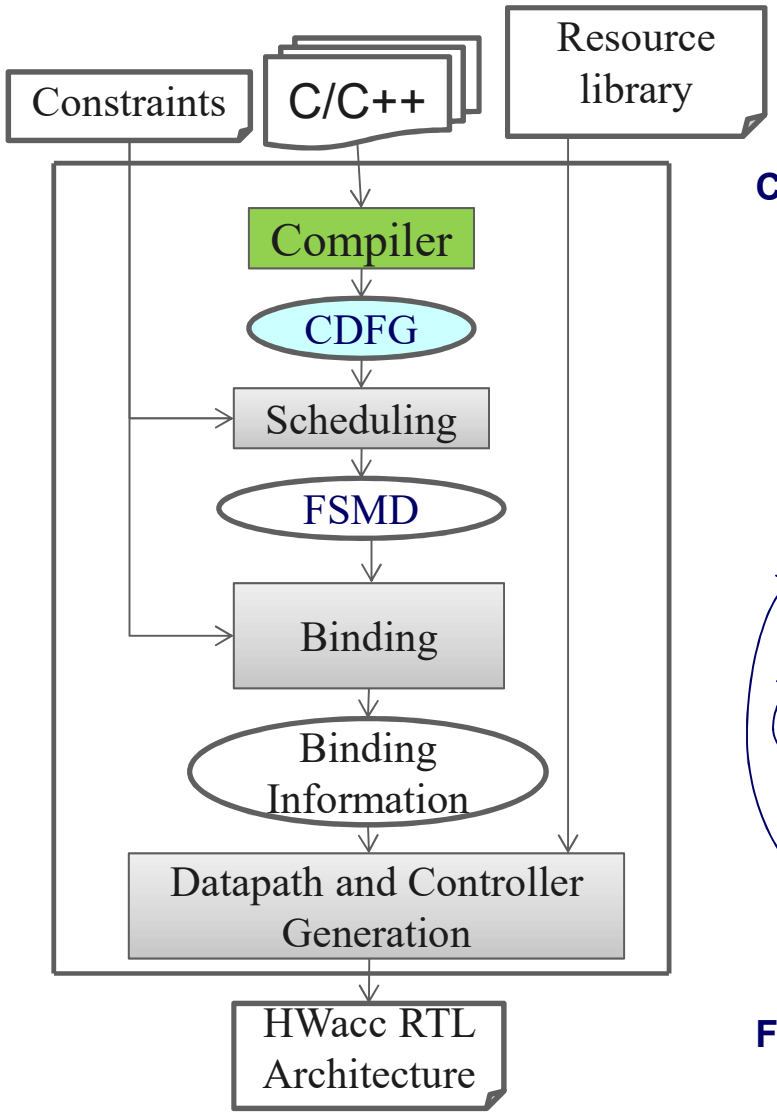
```
1: t = a+b;
2: u = a'-b';
3: if (a<b)
4:     v = t+c;
else
{
5:     w = u+c';
6:     v = w-d;
}
7: x = v+e;
8: y = v-e;
```

Source code



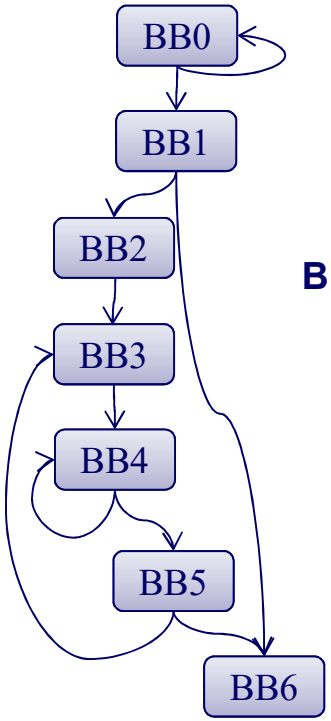
Graphical representation

Application modeling (2/3): CDFG



Compilation generates a formal modeling of the algorithmic input description

CFG: Control Flow Graph

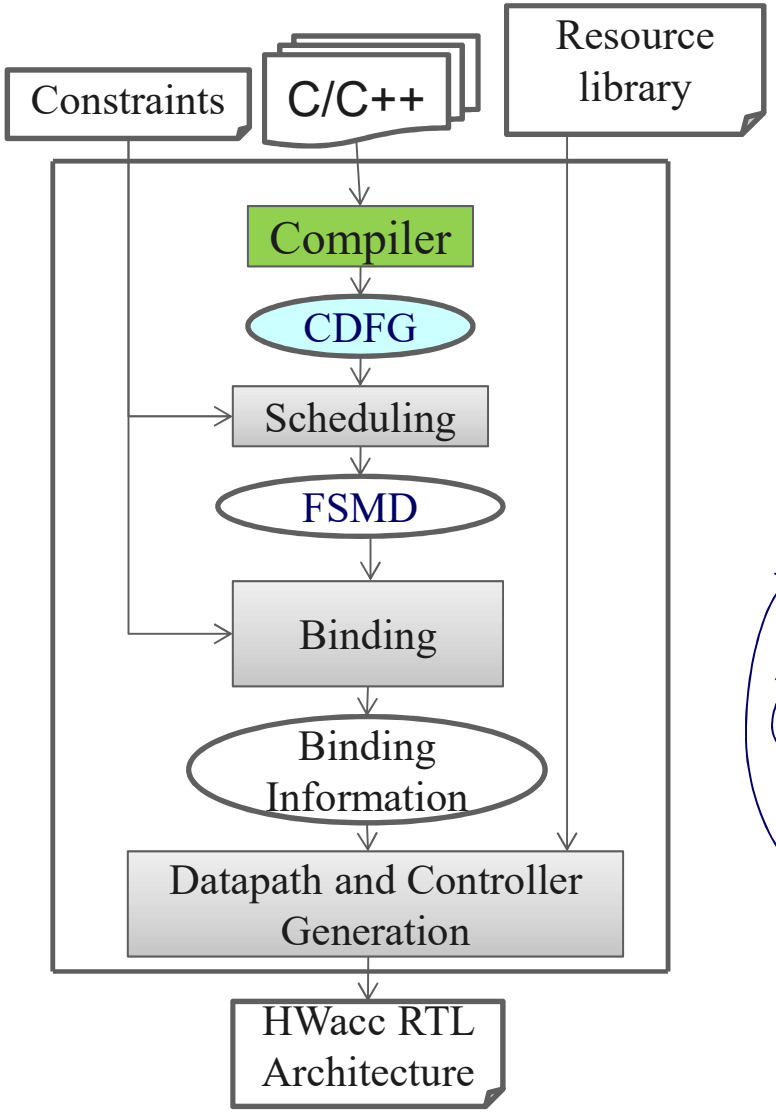


BB: Basic Block

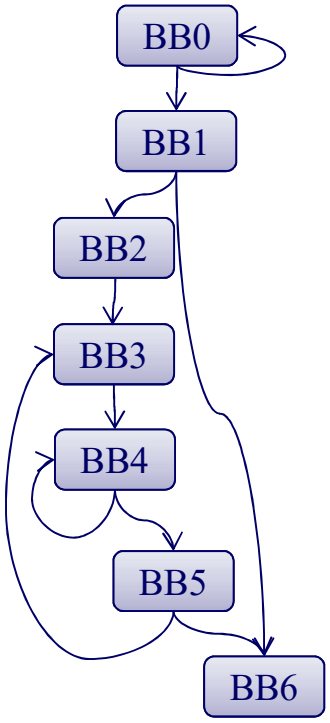
Front-end : GCC or LLVM / CDFG: Control Data Flow Graph

Application modeling (3/3)

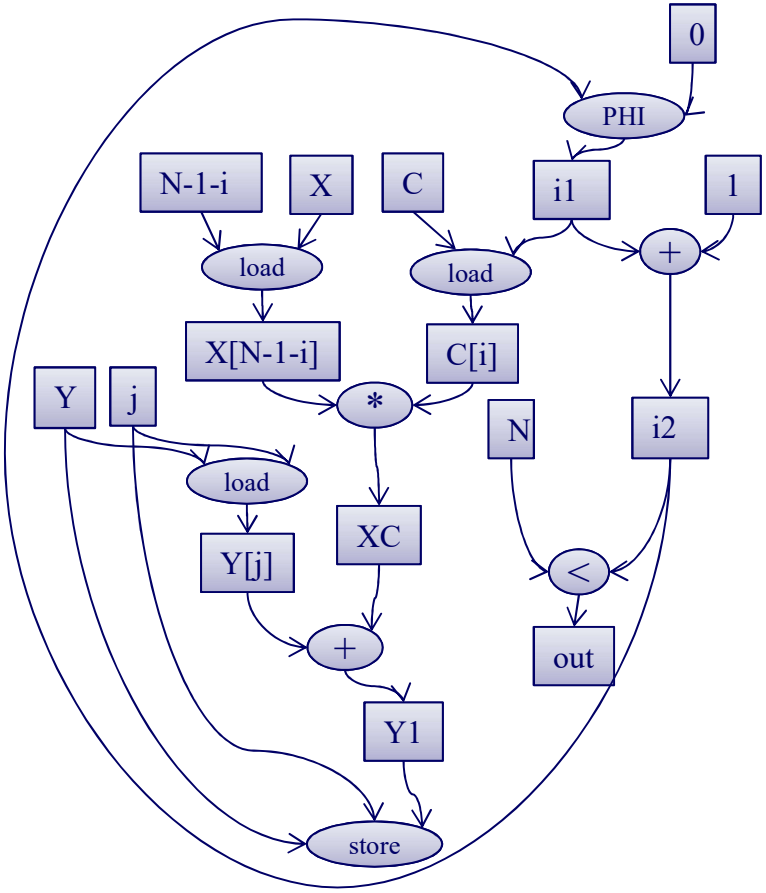
```
Void Filter (int N, int C[N], int X[N], int Y[N]){
   (1)  int i,j;
   (2) for (j =0; j<N; j++){ // loop1
   (3)      Y[j] = 0;
   (4)     for (i=0; i<N, i++){// loop2
   (5)          Y[j]= Y[j] + C[i]*X[N-1-i];
   (6)       }
   (7)    }
}
```



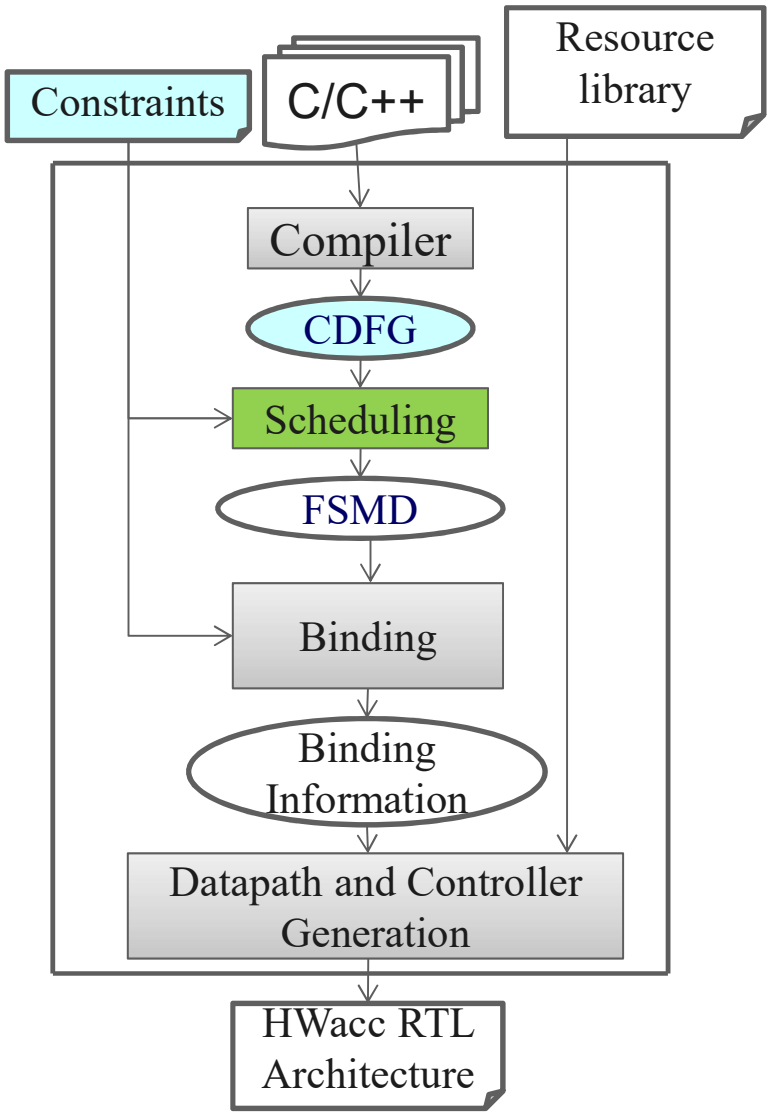
CFG



DFG: Data Flow Graph

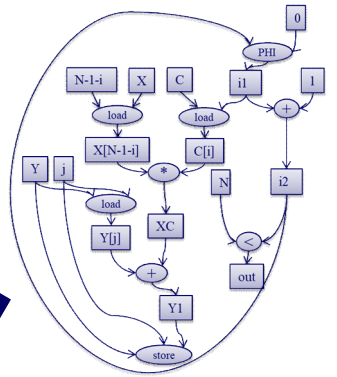


Scheduling step



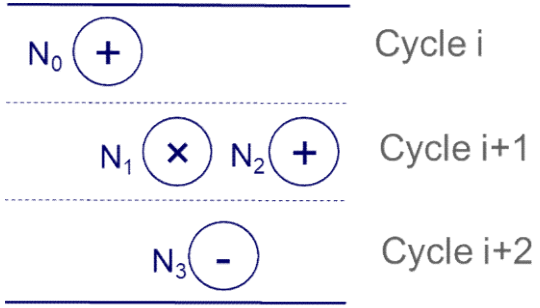
Scheduling defines the execution time of each operation

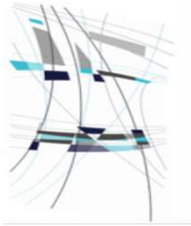
DFG: Data Flow Graph



Constraints / Objectives

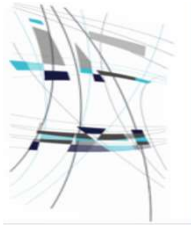
Scheduling





Scheduling algorithms

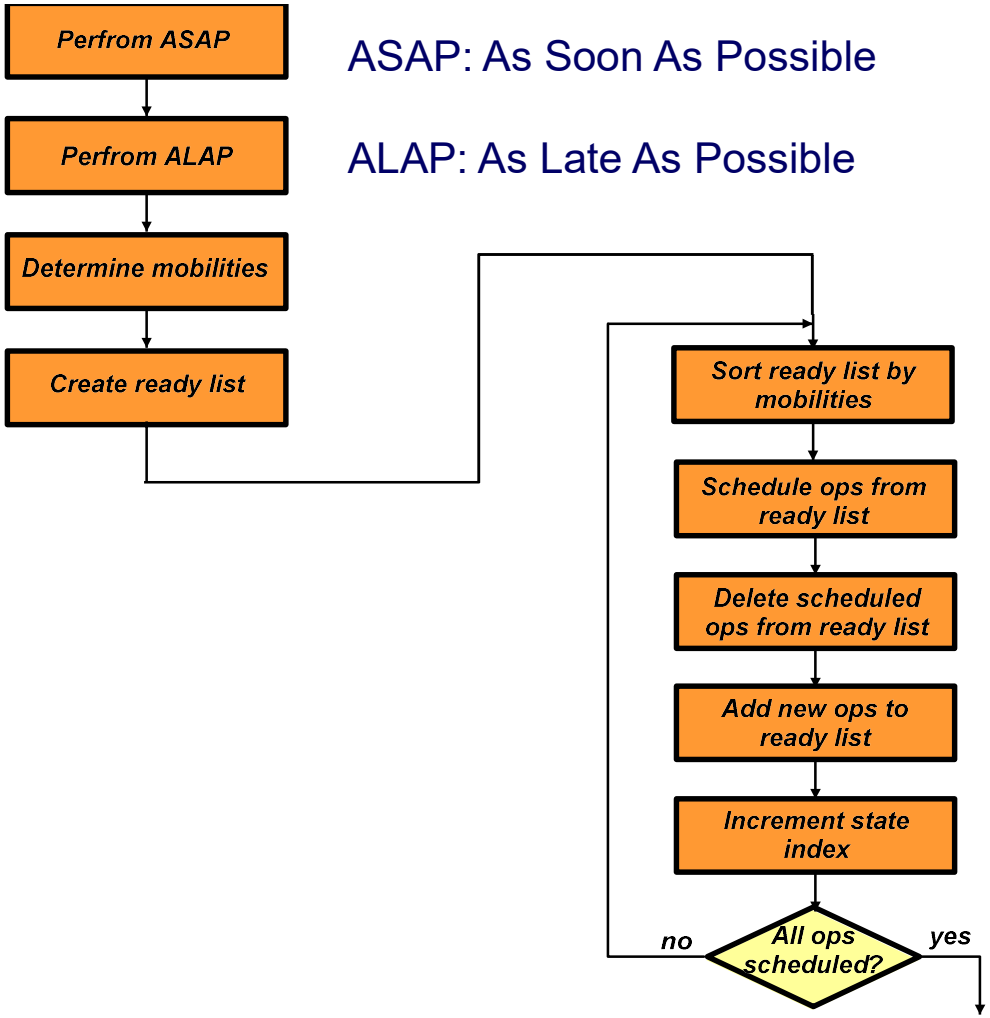
- Not constrained
  - ASAP, ALAP
- Resource constrained
  - Minimize latency
    - List-Scheduling…
  - Maximize throughput
    - Modulo scheduling (IMS, SMS…)
- Time constrained
  - Minimize resources
    - Force-directed scheduling…
- Miscellaneous
  - ILP, SDC, SA, GA, ACO, CP…



Scheduling algorithms

- Not constrained
  - ASAP, ALAP
- Resource constrained
  - Minimize latency
    - List-Scheduling…
  - Maximize throughput
    - Modulo scheduling (IMS, SMS…)
- Time constrained
  - Minimize resources
    - Force-directed scheduling…
- Miscellaneous
  - ILP, SDC, SA, GA, ACO, CP …

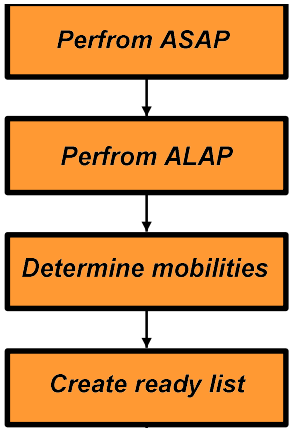
List-Scheduling Algorithm



ASAP: As Soon As Possible

ALAP: As Late As Possible

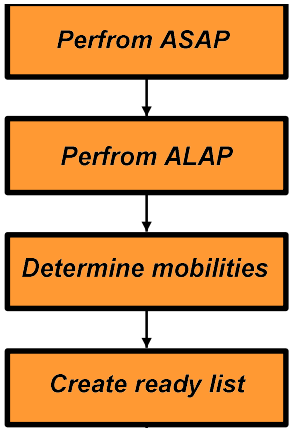
List-Scheduling Algorithm



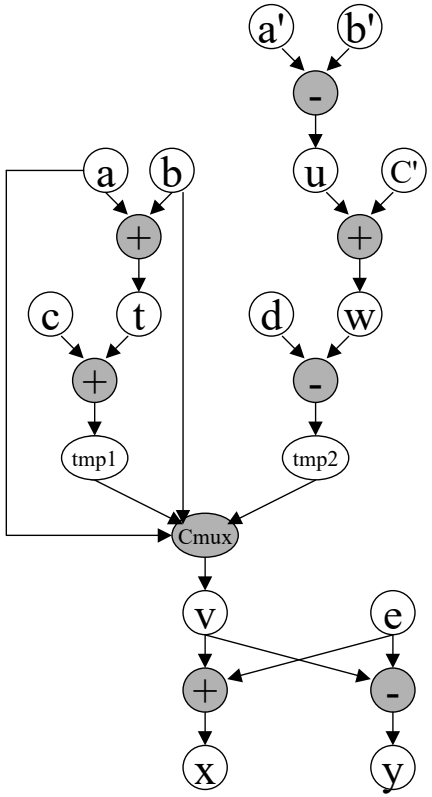
ASAP: As Soon As Possible

ALAP: As Late As Possible

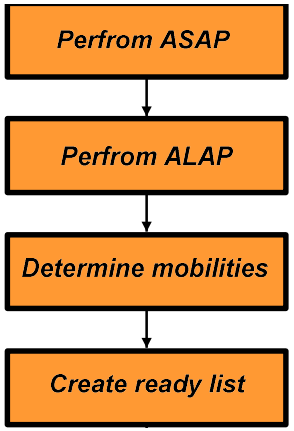
List-Scheduling Algorithm



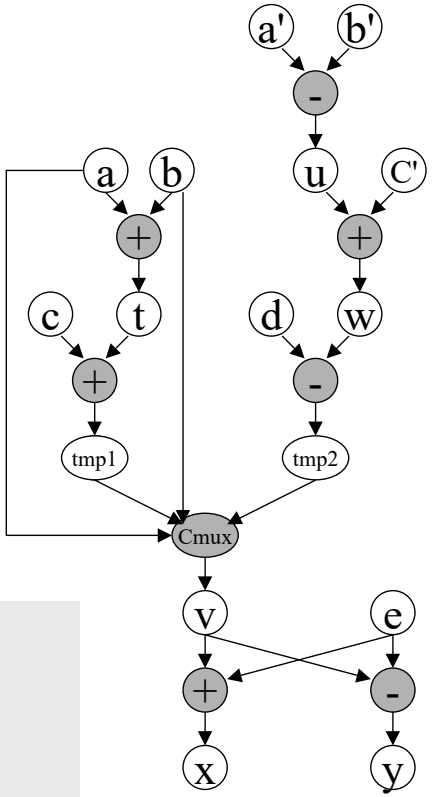
Data Flow Graph (DFG)



List-Scheduling Algorithm

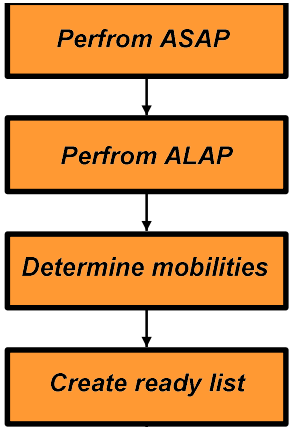


Data Flow Graph (DFG)

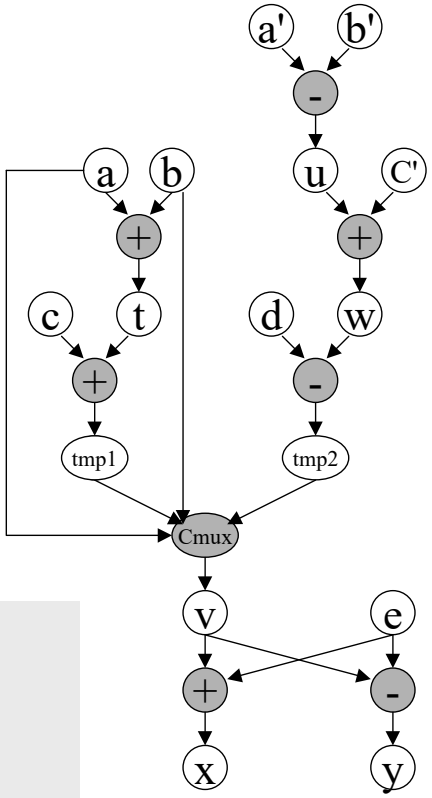


- Constraints
  - 1 adder (1 cycle)
  - 1 subtractor (1 cycle)
  - 1 comp/select (1 cycle)

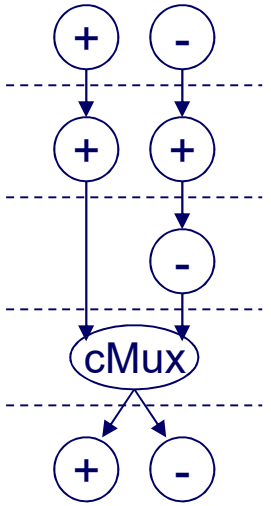
List-Scheduling Algorithm



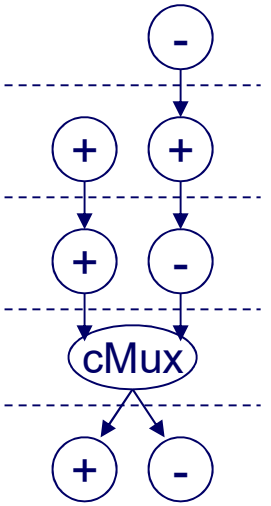
Data Flow Graph (DFG)



ASAP

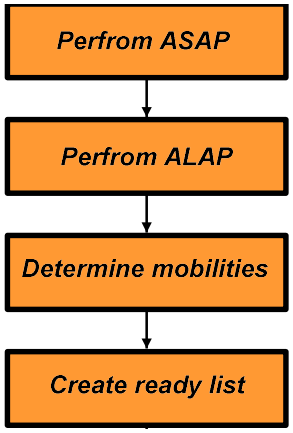


ALAP

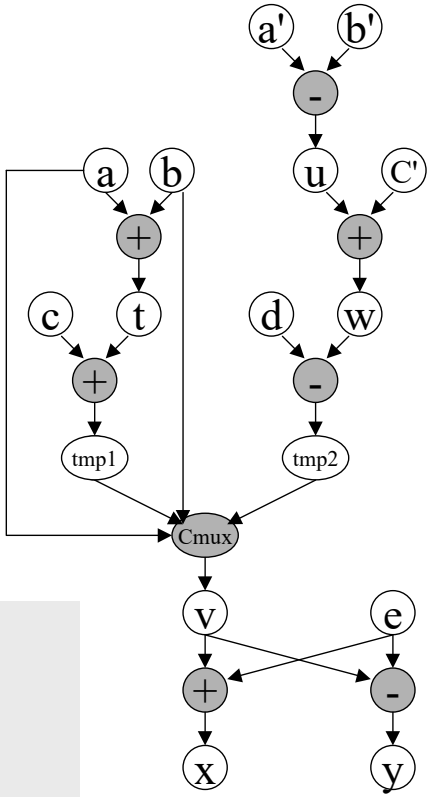


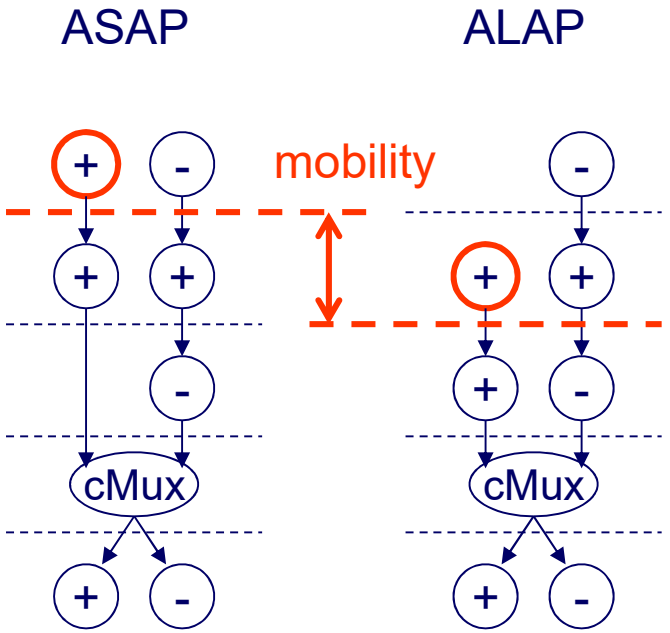
- Constraints
  - 1 adder (1 cycle)
  - 1 subtractor (1 cycle)
  - 1 comp/select (1 cycle)

List-Scheduling Algorithm



Data Flow Graph (DFG)

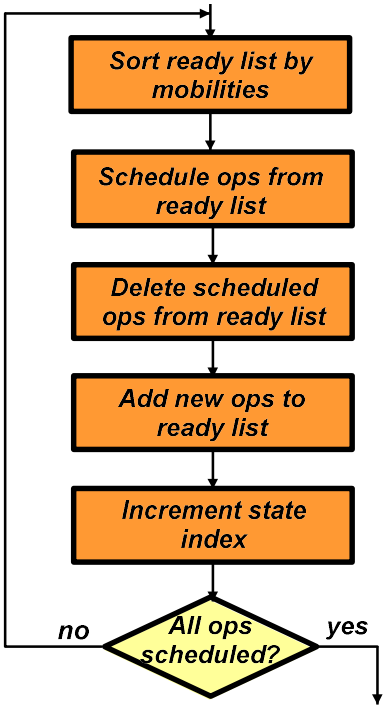


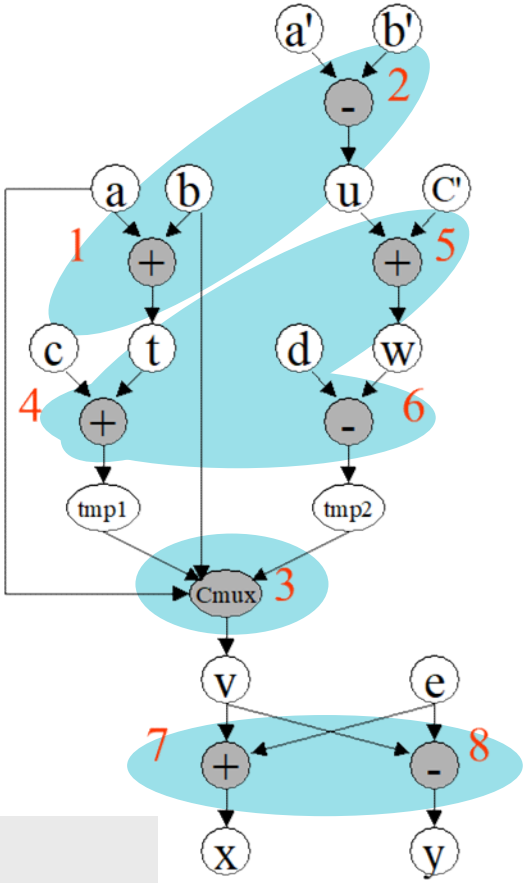


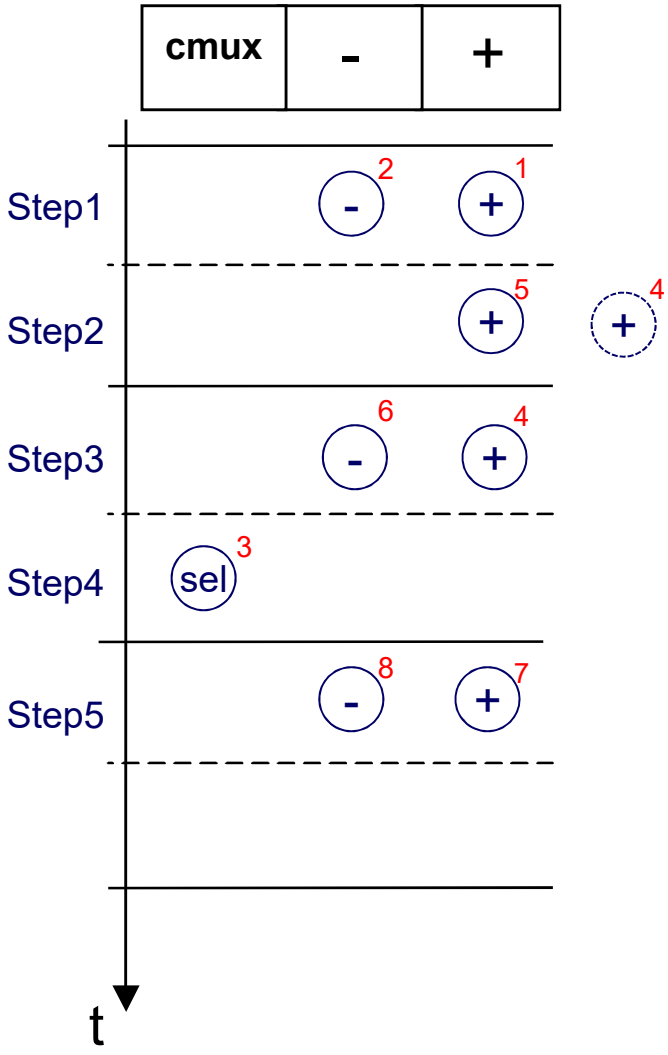
Priority = 1/Mobility

- Constraints
  - 1 adder (1 cycle)
  - 1 subtractor (1 cycle)
  - 1 comp/select (1 cycle)

List-Scheduling Algorithm

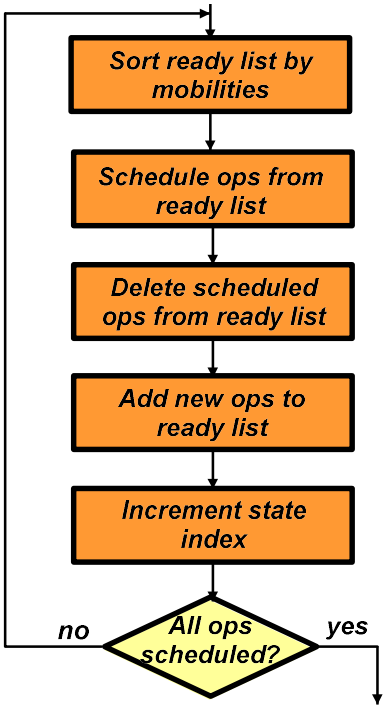


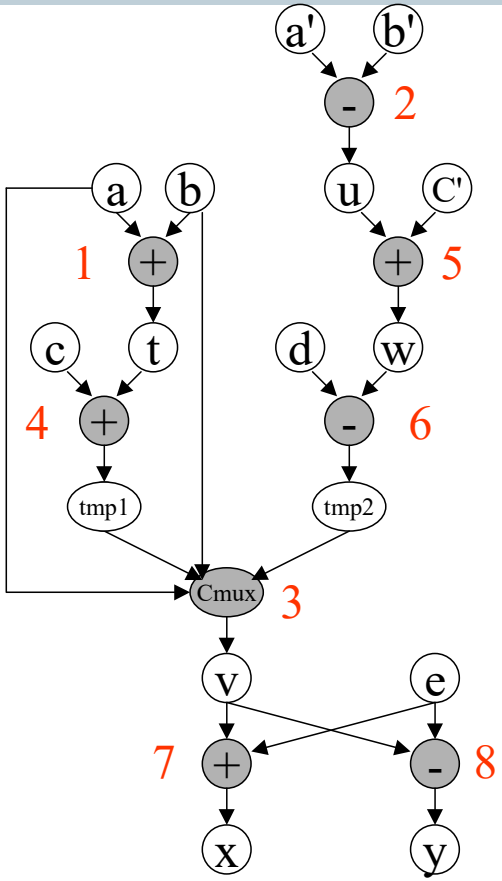


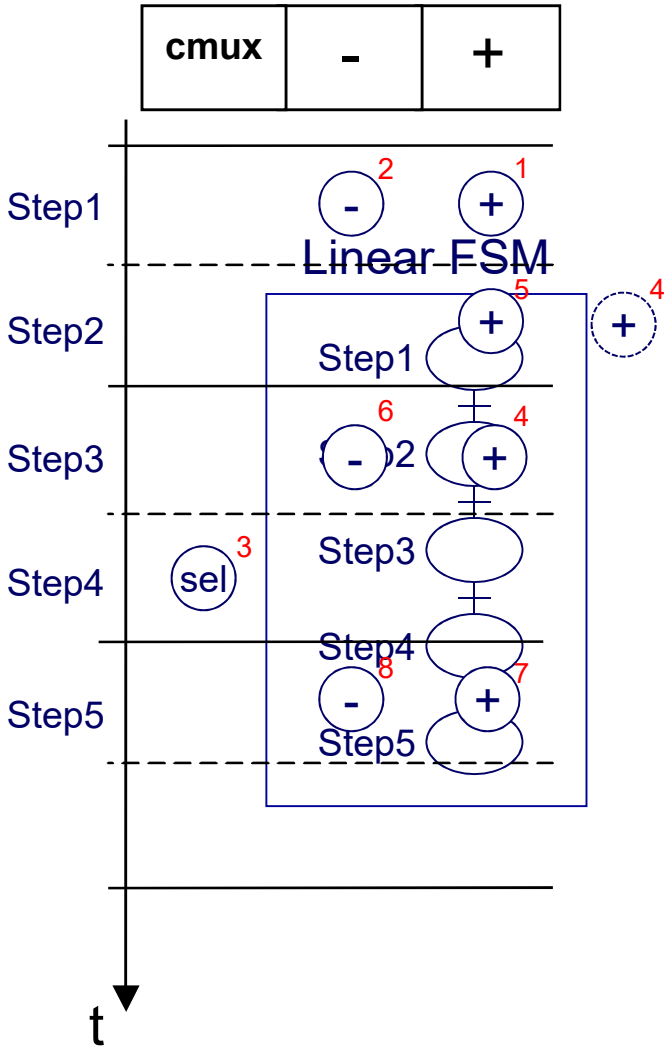


- Constraints
  - 1 adder (1 cycle)
  - 1 subtractor (1 cycle)
  - 1 comp/select (1 cycle)

List-Scheduling Algorithm

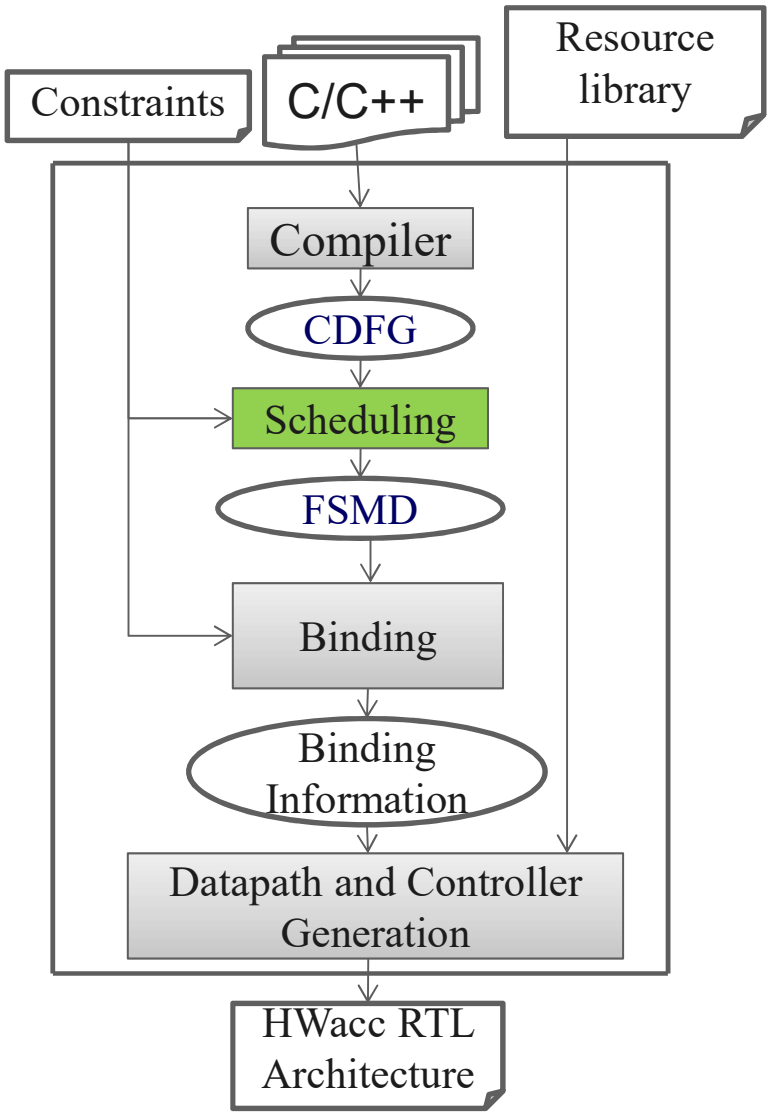




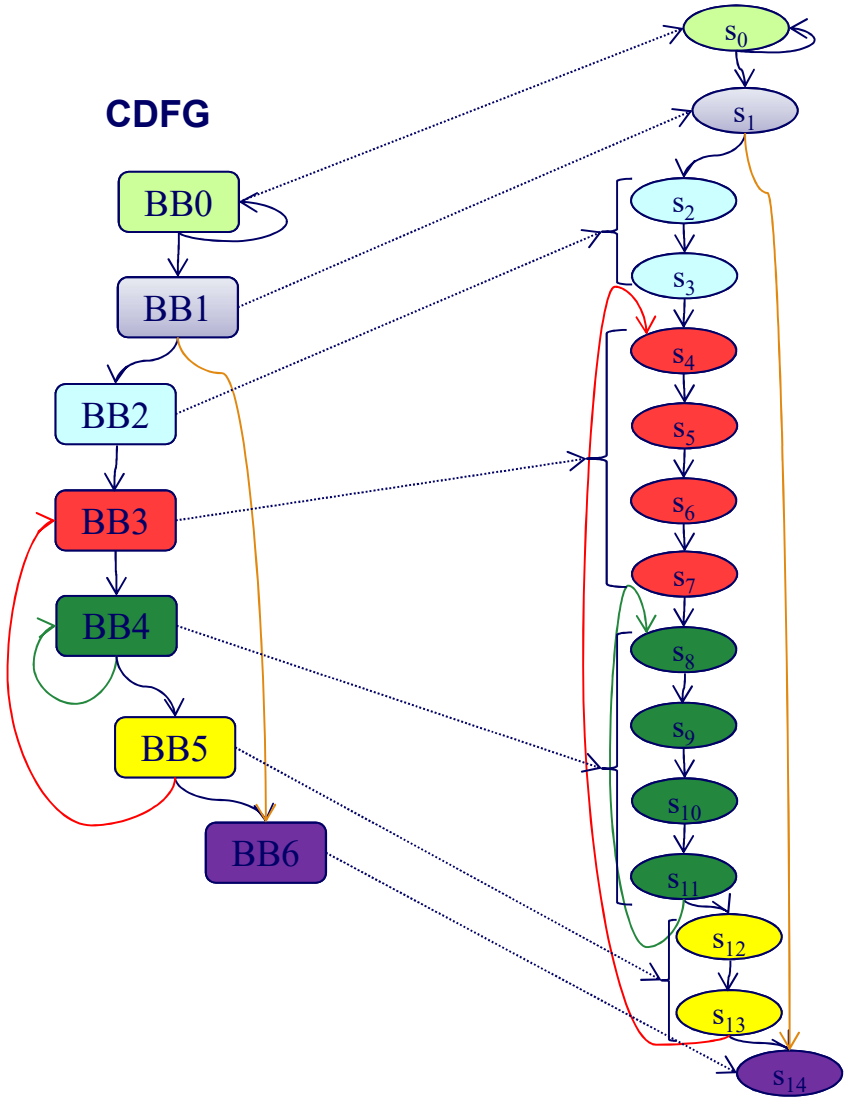


- Constraints
  - 1 adder (1 cycle)
  - 1 subtractor (1 cycle)
  - 1 comparing component (1 cycle)

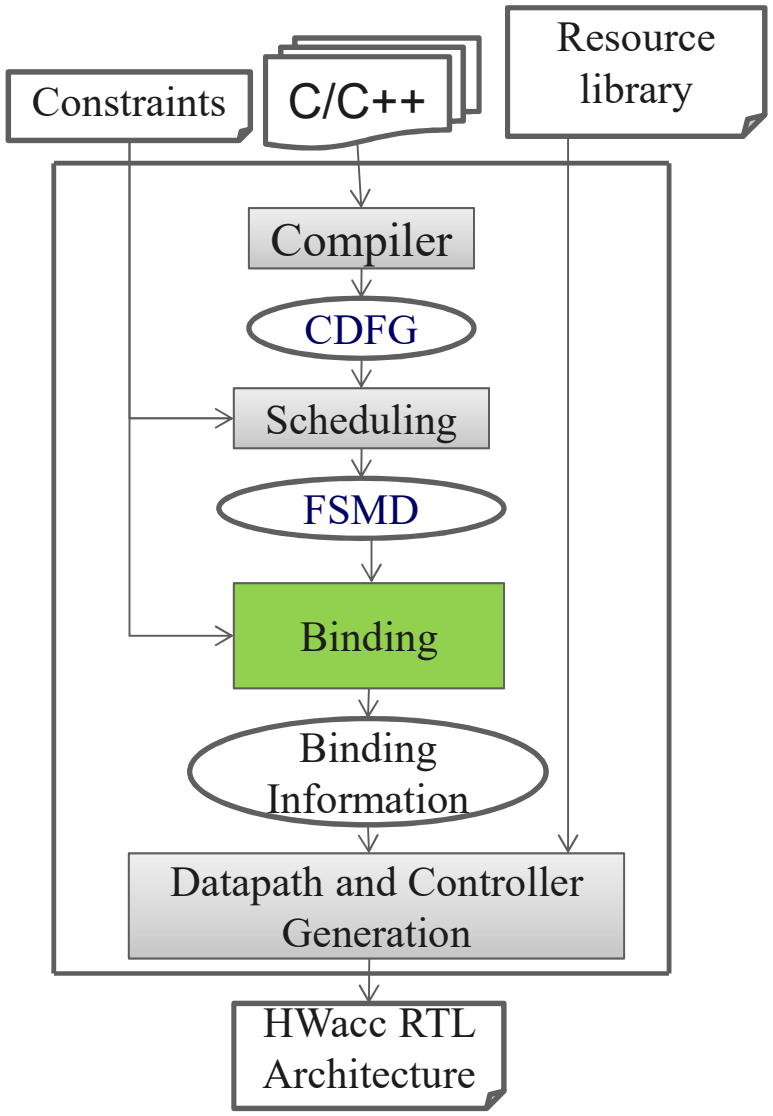
Scheduling results



FSMD: Finite State Machine with Data-Path



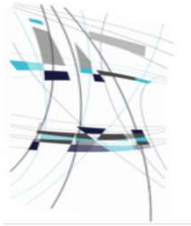
Binding step



Binding defines which operator executes a given operation and which memory element stores a given data

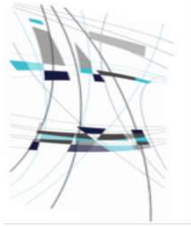
1 adder, 1 multiplier and 1 bank memory

Operation/Variable	Operator #instance
load	Load #0
store	Store #0
+	ADD #0
*	MUL #0
X	REG #6
Y	REG #1
C	REG #7
N-1-i	REG #4
i	REG #2
j	REG #0



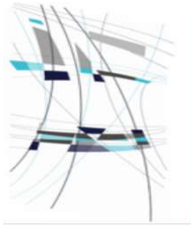
Binding step

- Complex problems to solve
- Operation and data binding are interdependent
- Resource-constrained or time-constrained
- Impact many architecture characteristics
  - Area, clock frequency, power/NRJ consumption, testability, temperature…
  - Can thus have different objectives
- Resource sharing (general case)
  - Assignment of a resource to more than one operation/data
- Binding strongly depends on scheduling results



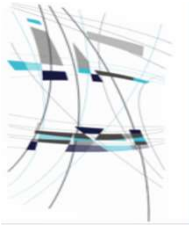
Binding approaches

- Based on the scheduling results, extract and model timing information
  - Compatibility graph
  - Conflict graph
  - Comparability graph
  - Interval Graph
  - Bipartite Graph
  - …
- Associated methods
  - Clique partitioning, coloring, heuristics, ILP, SA, CP, MWBM…



Binding approaches

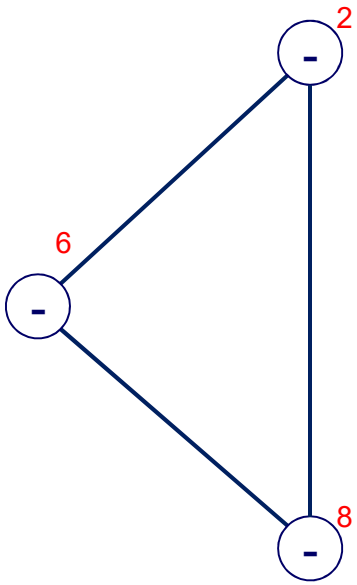
- Based on the scheduling results, extract and model timing information
  - Compatibility graph
  - Conflict graph
  - Comparability graph
  - Interval Graph
  - Bipartite Graph
  - …
- Associated methods
  - Clique partitioning, coloring, heuristics, ILP, SA, CP, MWBM…



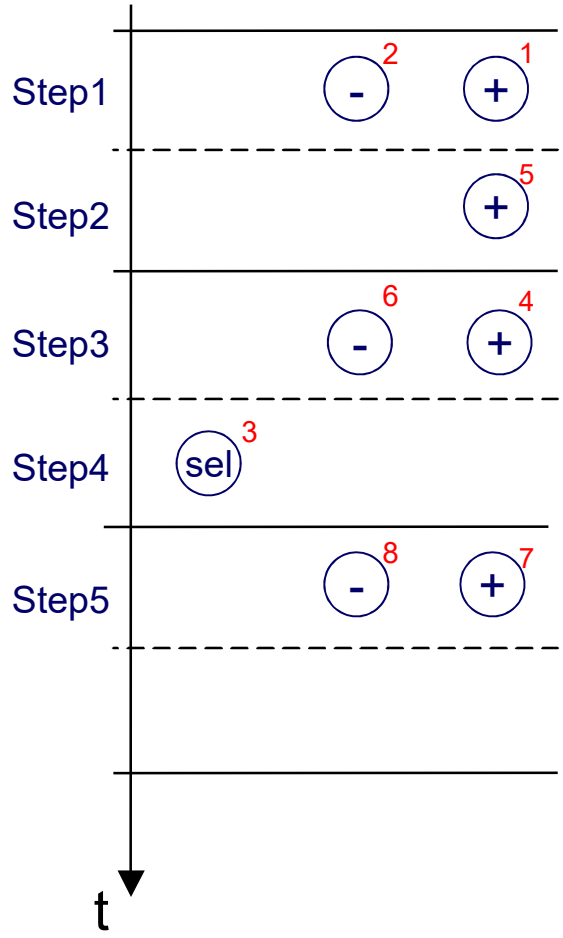
Compatibility Graph

- Resource Compatibility Graph *G (V,E)*
  - *V* represents operations
  - *E* represents *compatible* operation pairs
- Compatible operations
  - are not concurrent (i.e. belongs to ≠ control step/clock cycles)
  - can share the same type of operators
- Objective
  - Partition the graph in a minimum number of cliques
    - Clique cover number / minimum clique cover

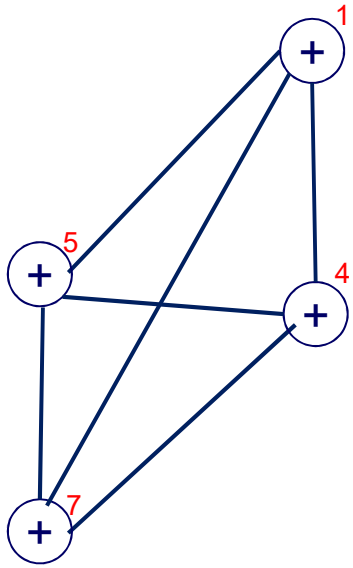
Compatibility Graph (ex. #1)



SUB

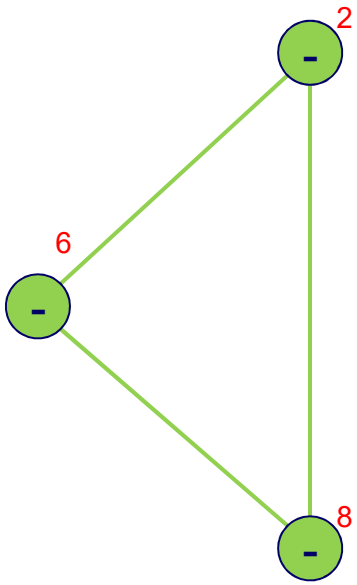


Scheduling result

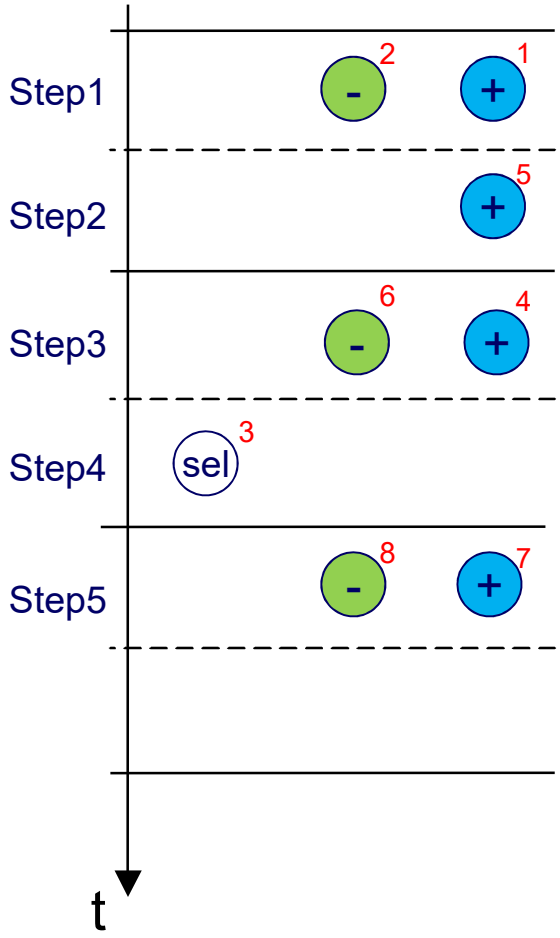


ADD

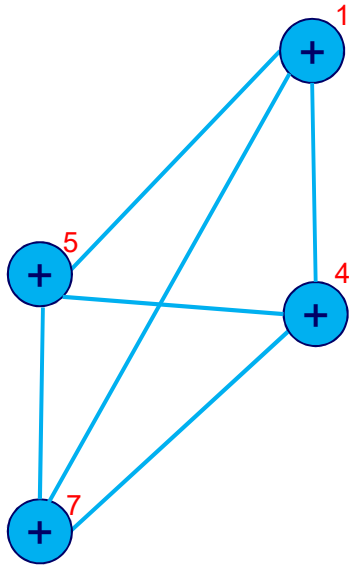
Compatibility Graph (ex. #1)



SUB

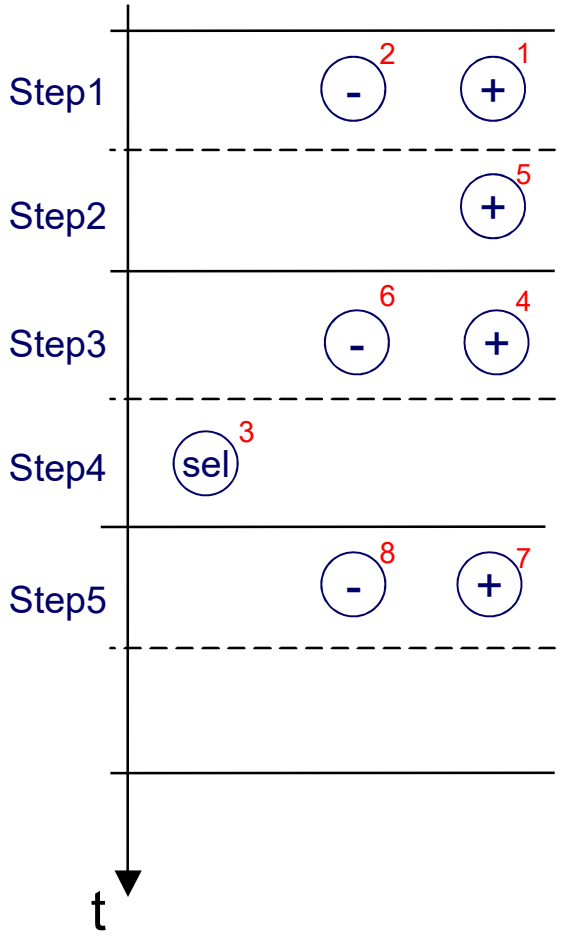


Scheduling result

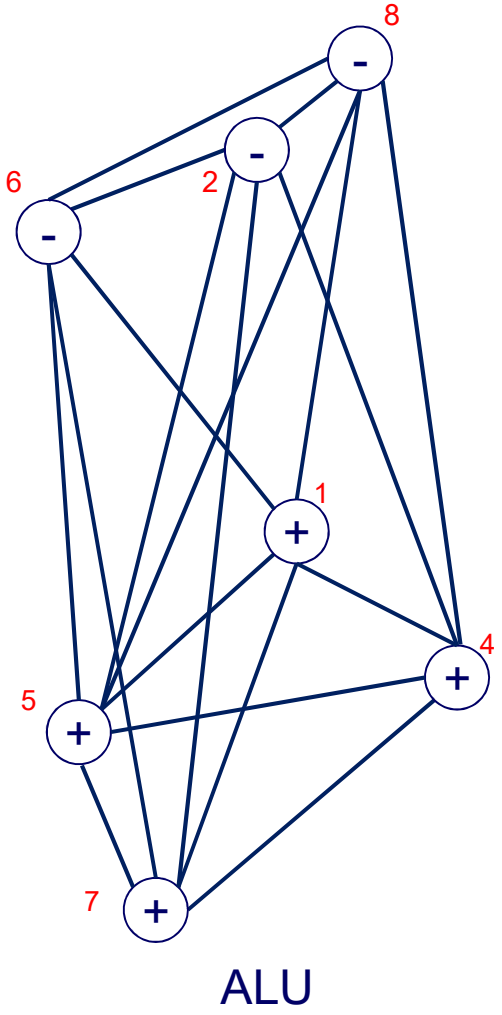


ADD

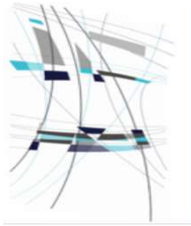
Compatibility Graph (ex. #2)



Scheduling result



ALU

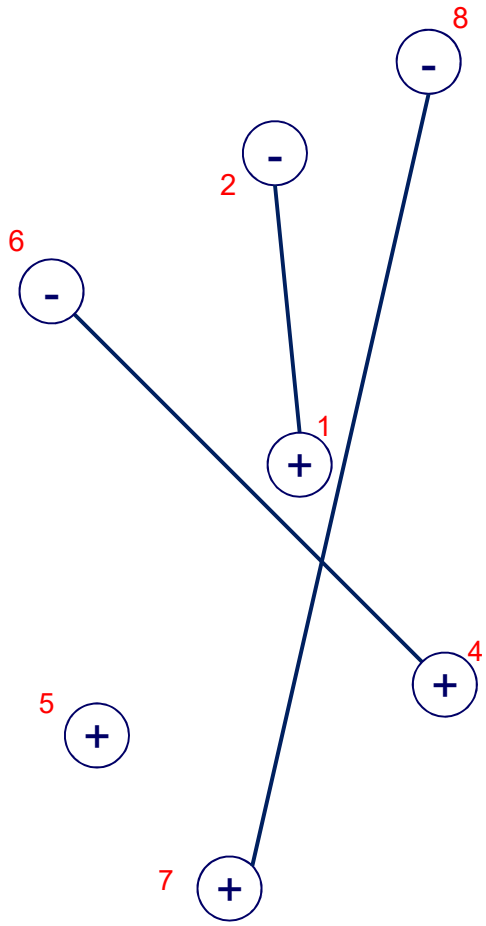


Conflict graph

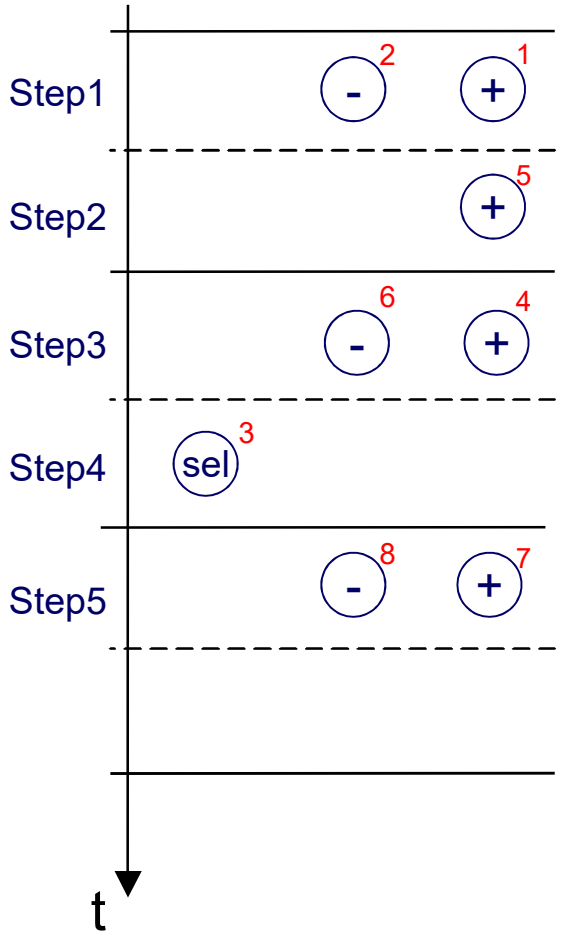
- Complementary to compatibility graph

- Resource Conflict Graph *G(V,E)*
  - *V* represents operations
  - E represents conflicting operation pairs
- Conflicting operations
  - Two operations are conflicting if they are *not* compatible
- Find independent set of *G(V,E)*
  - A set of mutually compatible operations
  - Coloring with minimum number of colors
    - Chromatic number

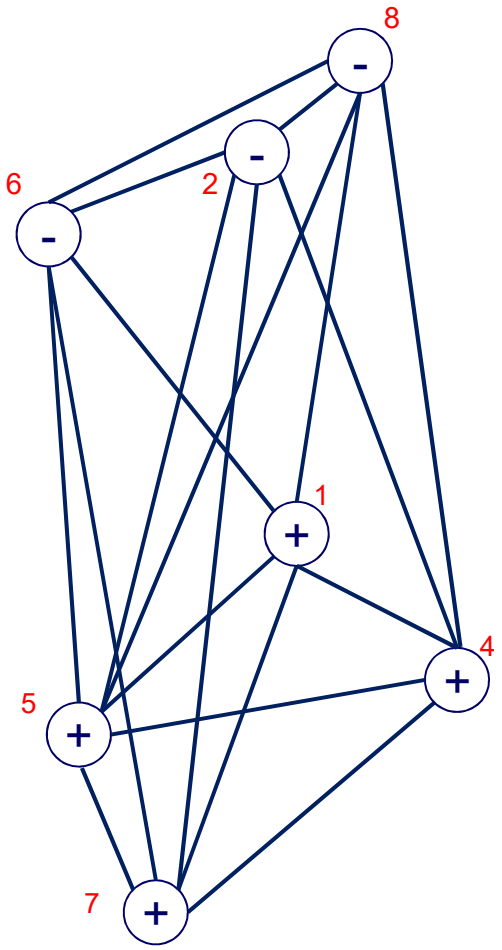
Conflict graph (ex. #2)



Conflict graph

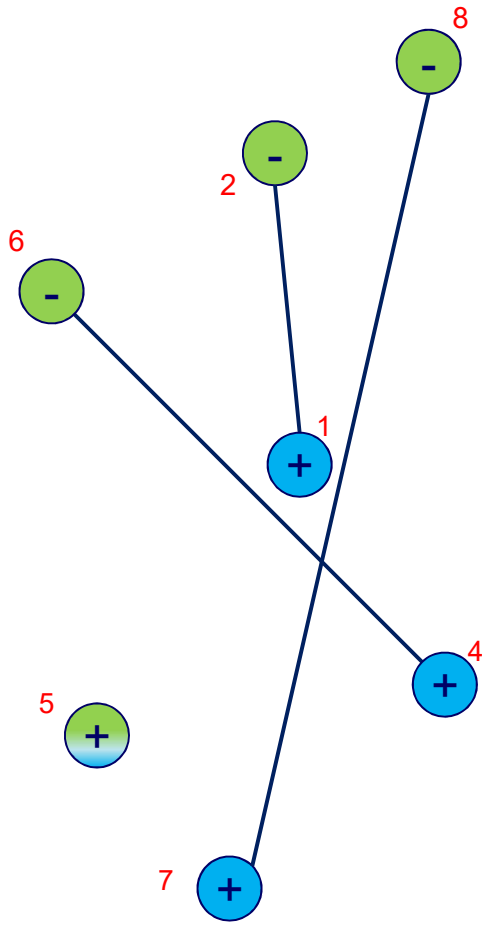


Scheduling result

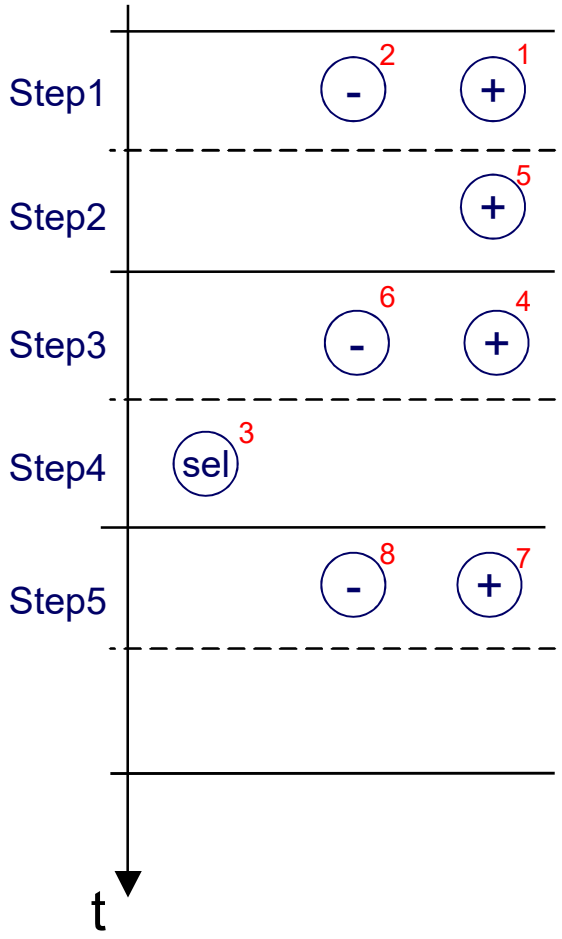


Compatibility graph

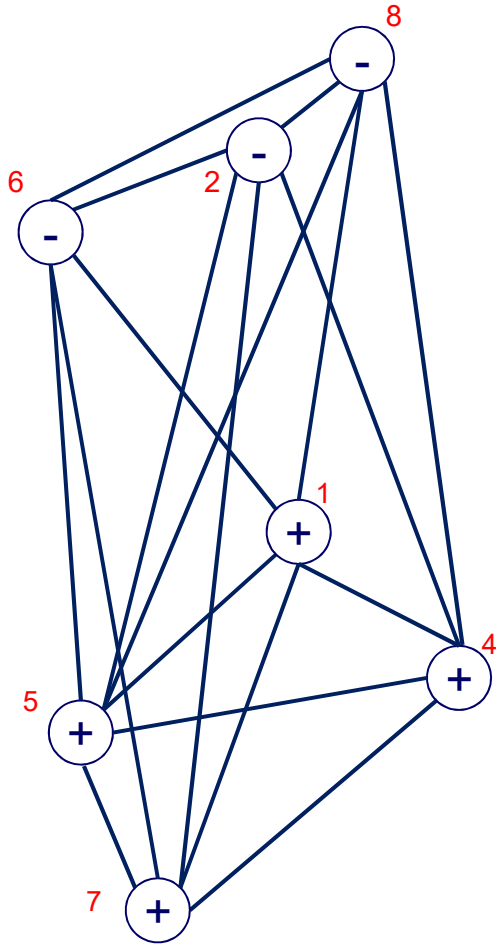
Conflict graph (ex. #2)



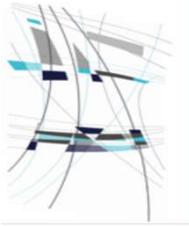
Conflict graph



Scheduling result



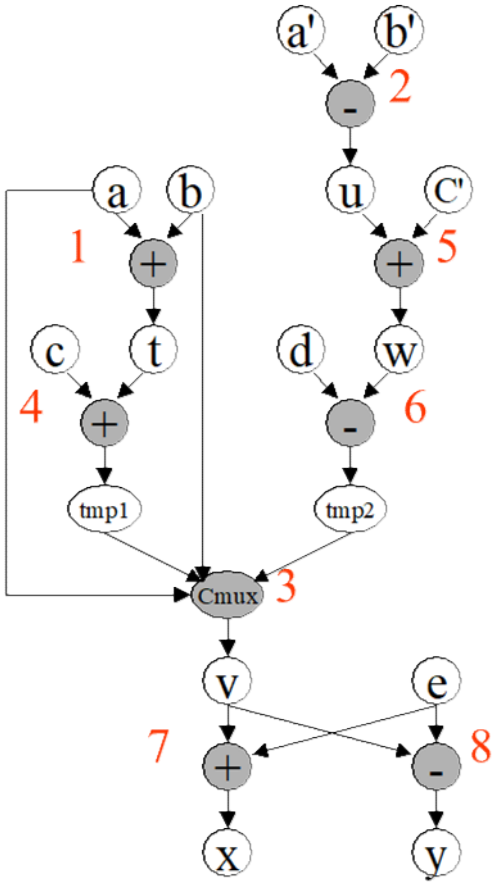
Compatibility graph



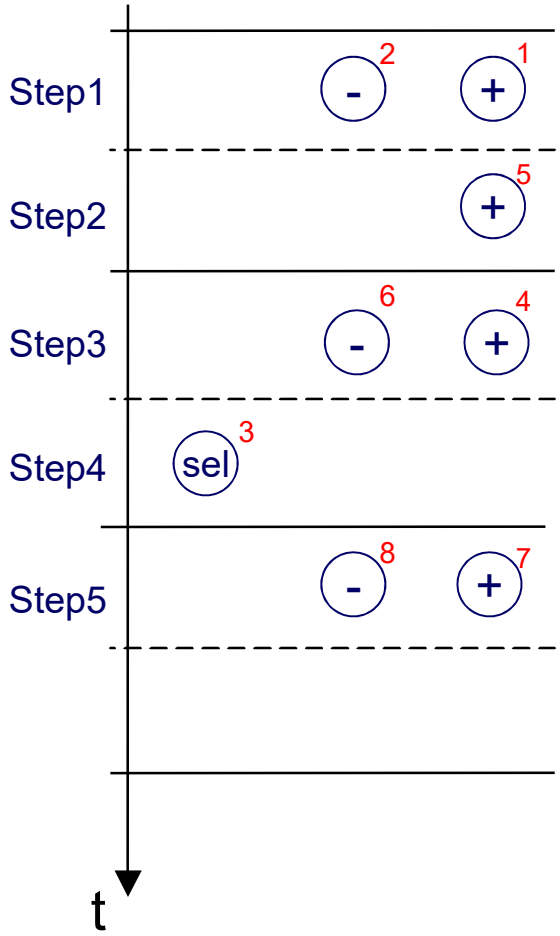
Comparability graph

- Compatibility graph
- Conflict graph
- Comparability graph
  - Graph has an orientation with transitive property
    - Edges are oriented => arcs

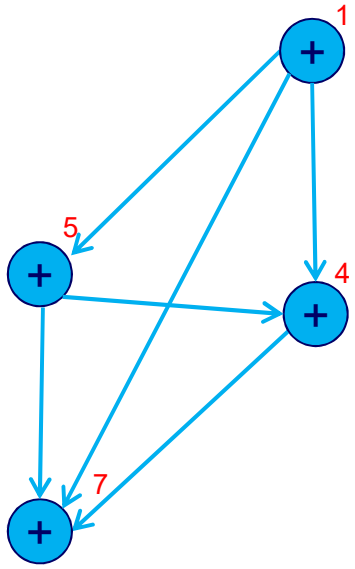
Comparability graph (ex. #1)



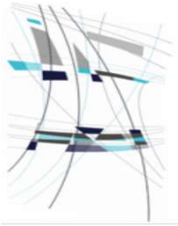
Data Flow Graph (DFG)



Scheduling result



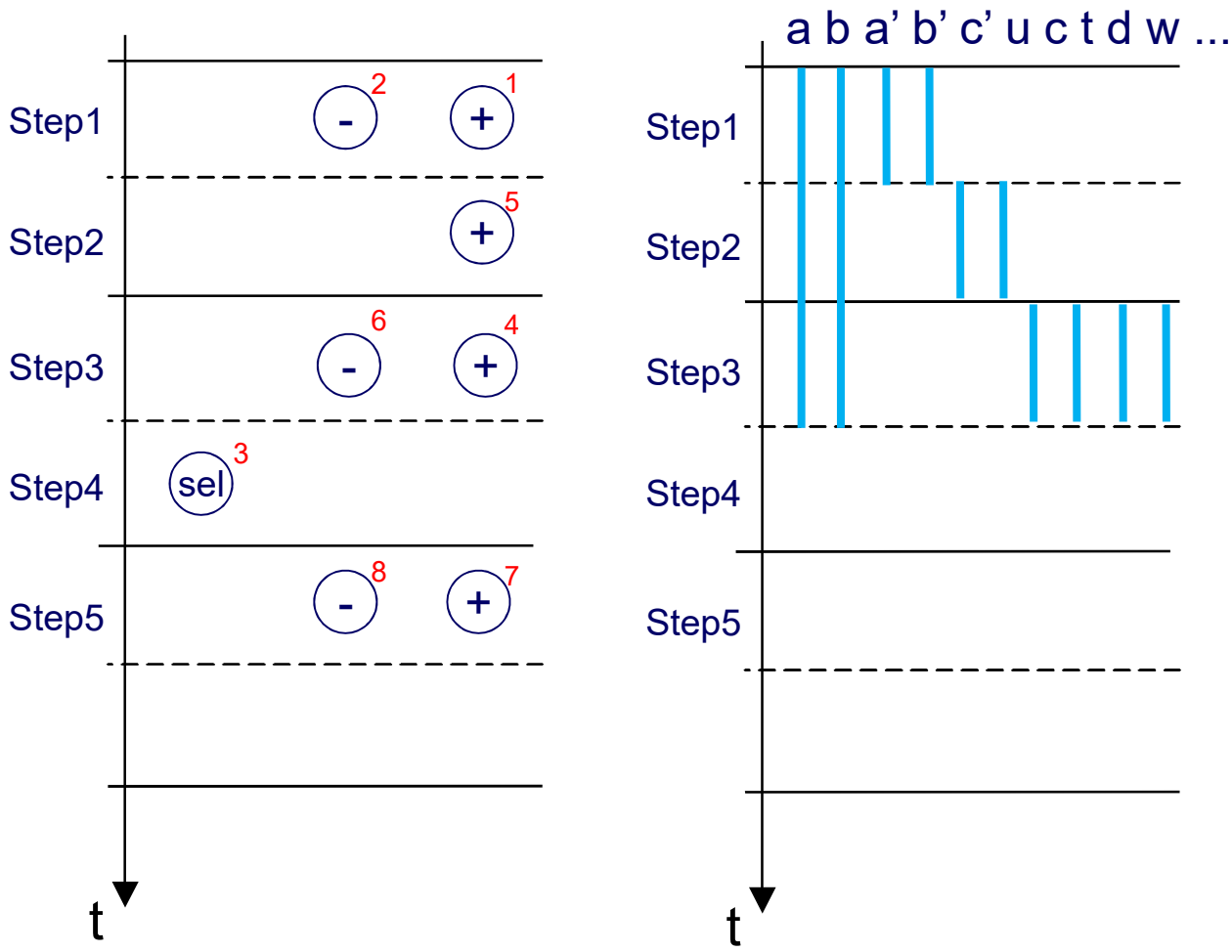
Comparability graph



Interval Graph

- Compatibility graph
- Conflict graph
- Comparability graph
- Interval graph
  - Vertices correspond to *intervals*
  - Edges correspond to interval *intersections*

Timing intervals



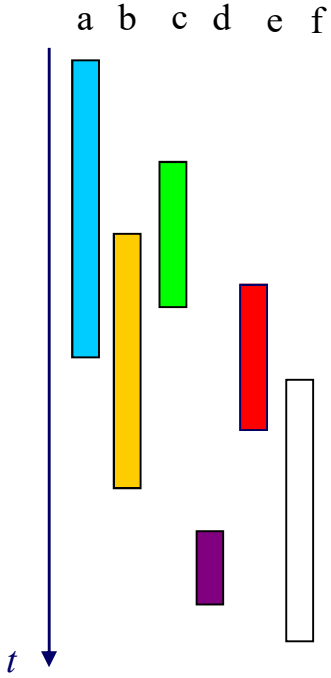
Scheduling result

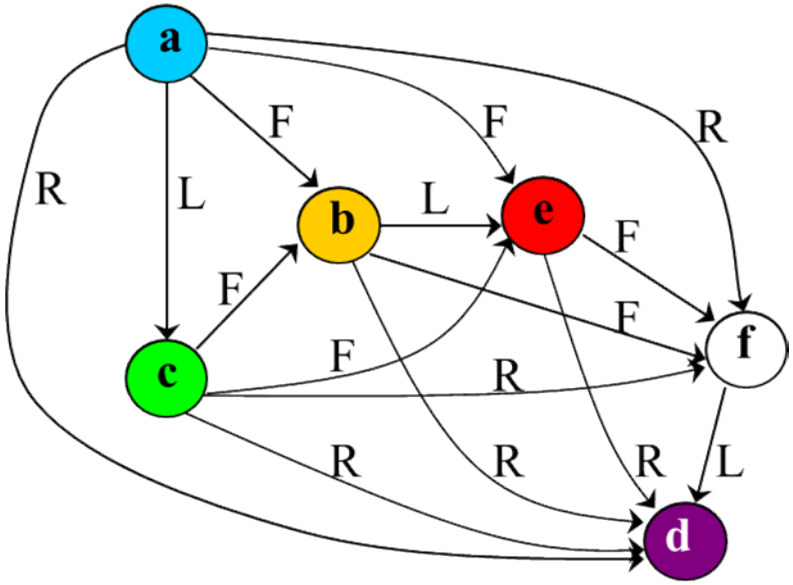
Data lifetime

Interval graph (ex. #3)

- Compatibility graph
- Conflict graph
- Comparability graph
- Interval graph

F: FIFO
L: LIFO
R: Register

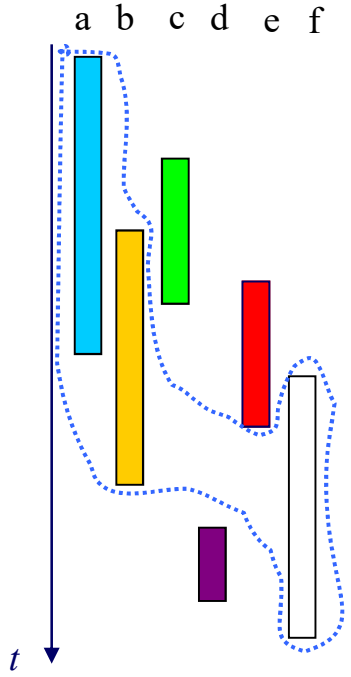


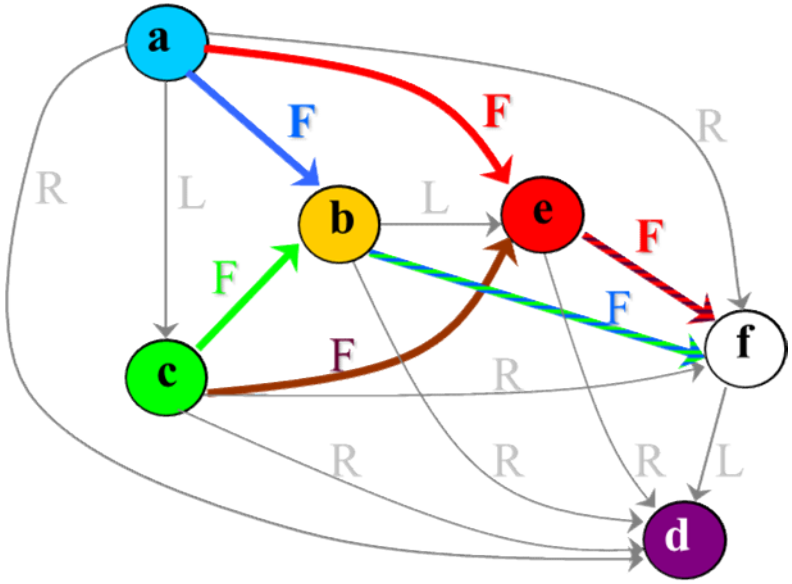


Interval graph (ex. #3)

- Compatibility graph
- Conflict graph
- Comparability graph
- Interval graph

F: FIFO
L: LIFO
R: Register

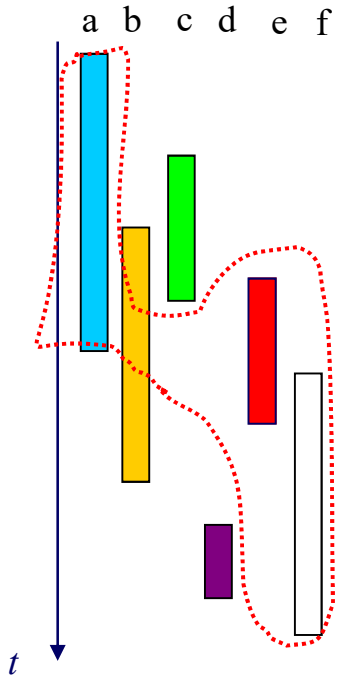


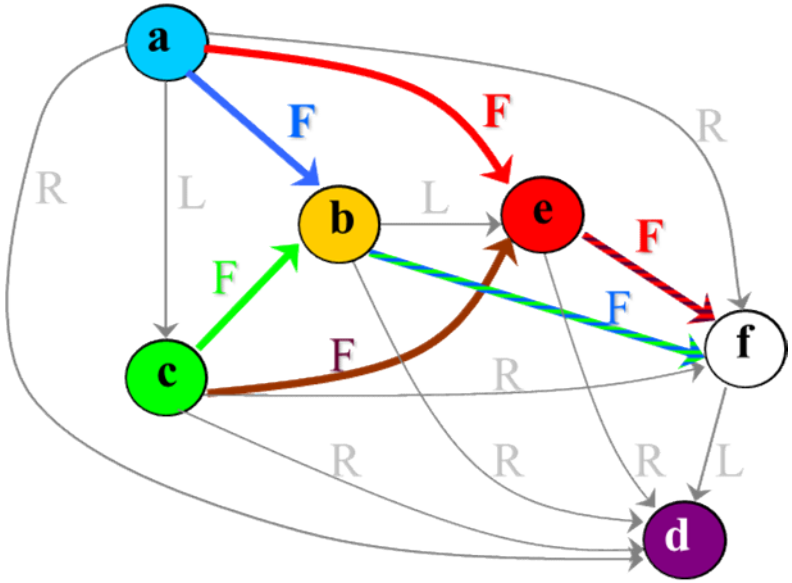


Interval graph (ex. #3)

- Compatibility graph
- Conflict graph
- Comparability graph
- Interval graph

F: FIFO
L: LIFO
R: Register

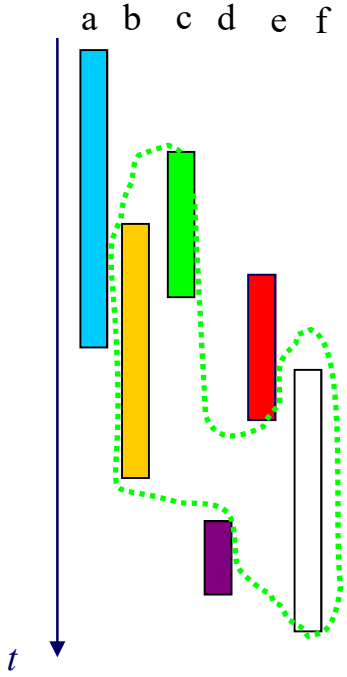


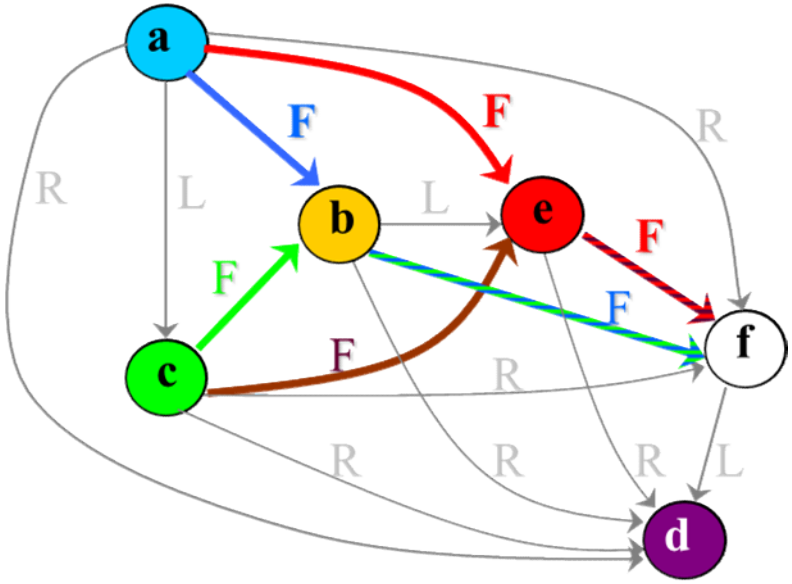


Interval graph (ex. #3)

- Compatibility graph
- Conflict graph
- Comparability graph
- Interval graph

F: FIFO
L: LIFO
R: Register

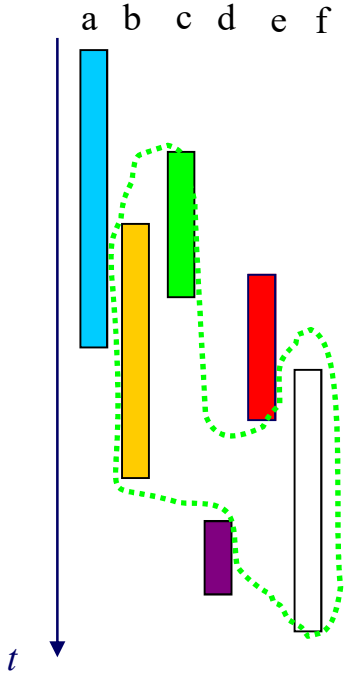


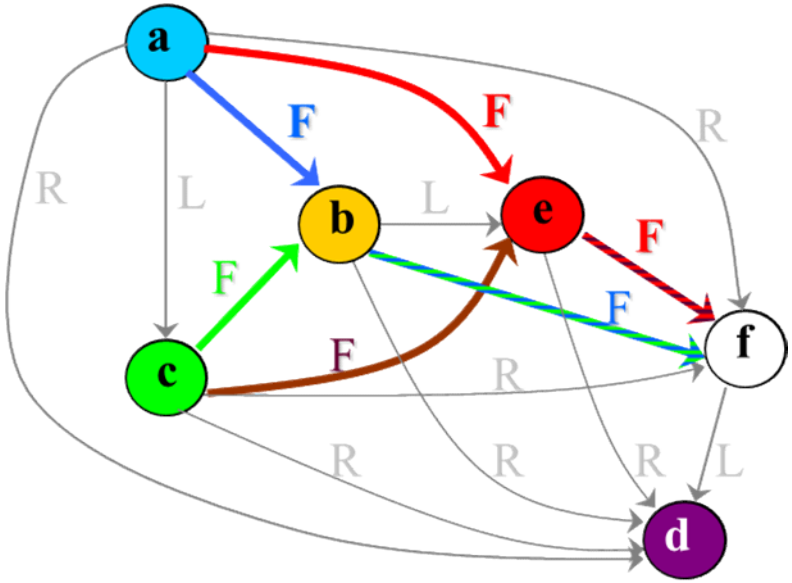


Interval graph (ex. #3)

- Compatibility graph
- Conflict graph
- Comparability graph
- Interval graph

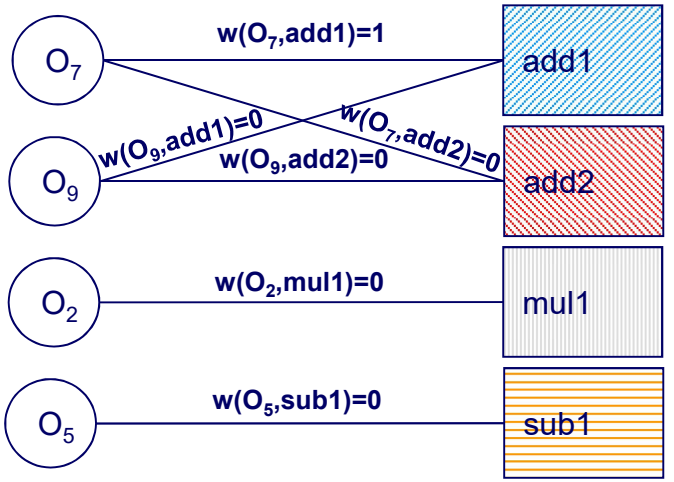
F: FIFO
L: LIFO
R: Register

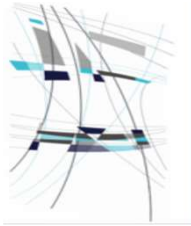




Bipartite Graph

- Compatibility graph
- Conflict graph
- Comparability graph
- Interval graph
- Bipartite graph
  - Vertices correspond to operations (data) and operators (storage elem.)
  - Edges connect operations to operators
  - Weights quantify the costs/interests of the binding
  - Cycle basis
  - Maximum Weight Bipartite Matching MWBM Algorithm

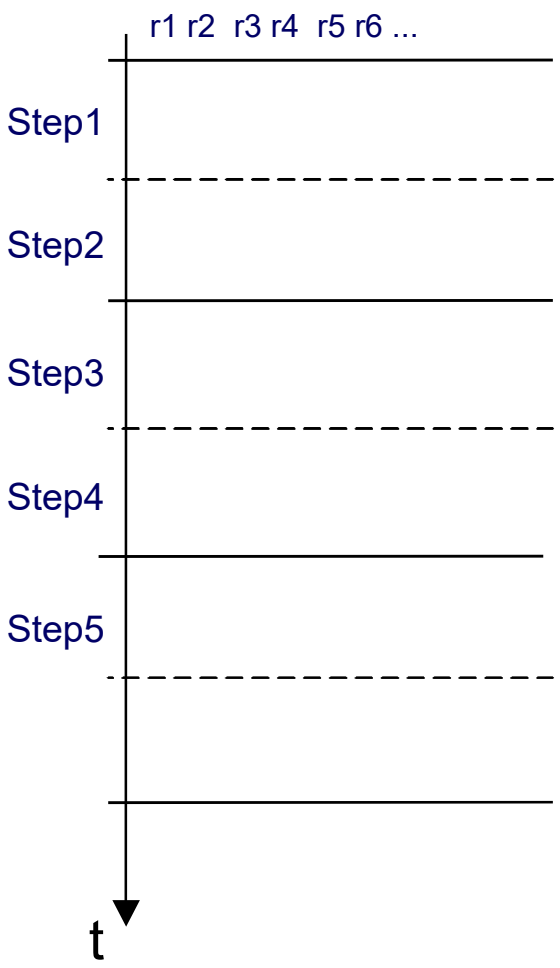




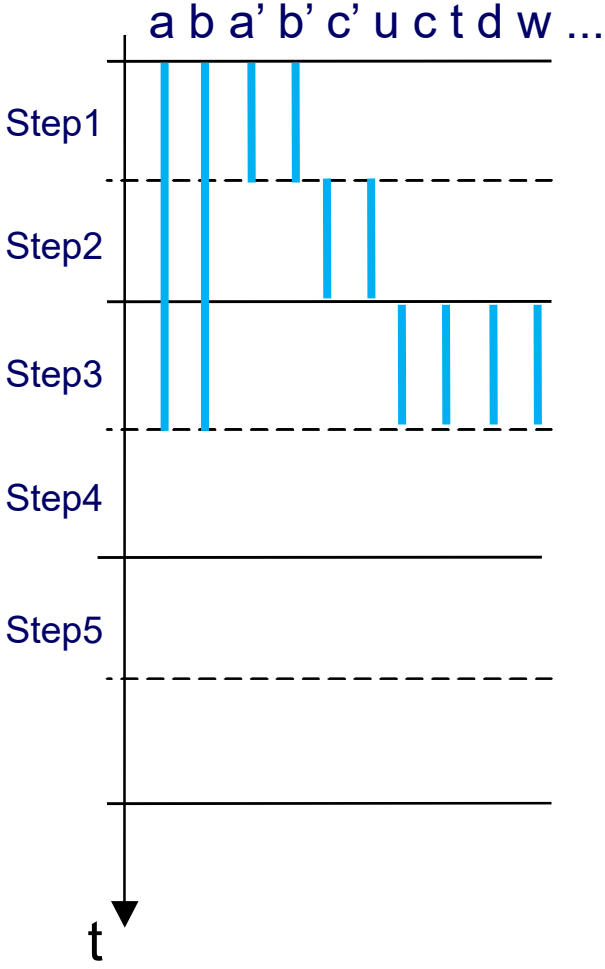
Left-Edge Algorithm (LEA)

- LEA
  - Sorts intervals by their *left* edge coordinates
  - Assigns non-overlapping intervals to first track (leftmost) using the sorted list
  - When possible intervals are exhausted, increase track counter and repeat.
- Properties
  - Simple
  - Polynomial time algorithm
    - Thanks to the timing information

Register binding (ex. cont.)

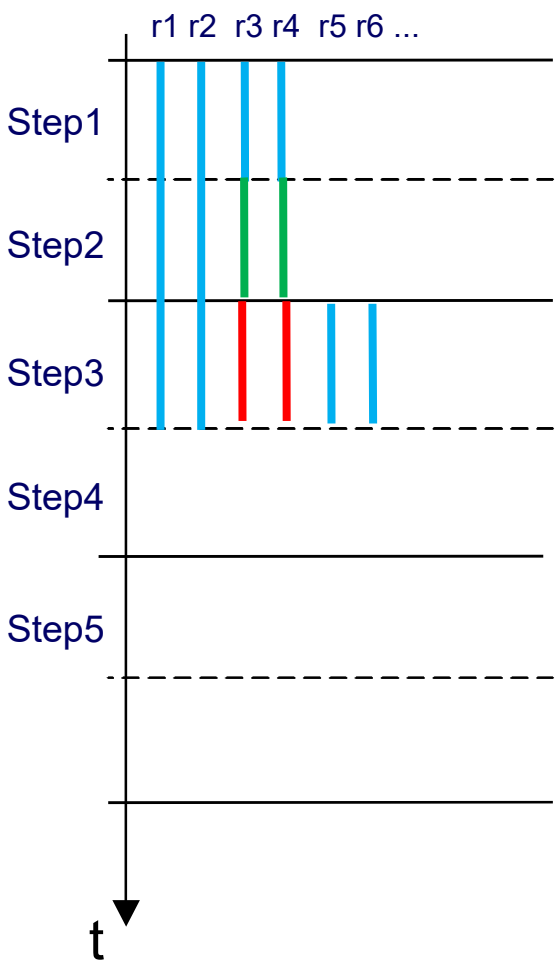


LEA results

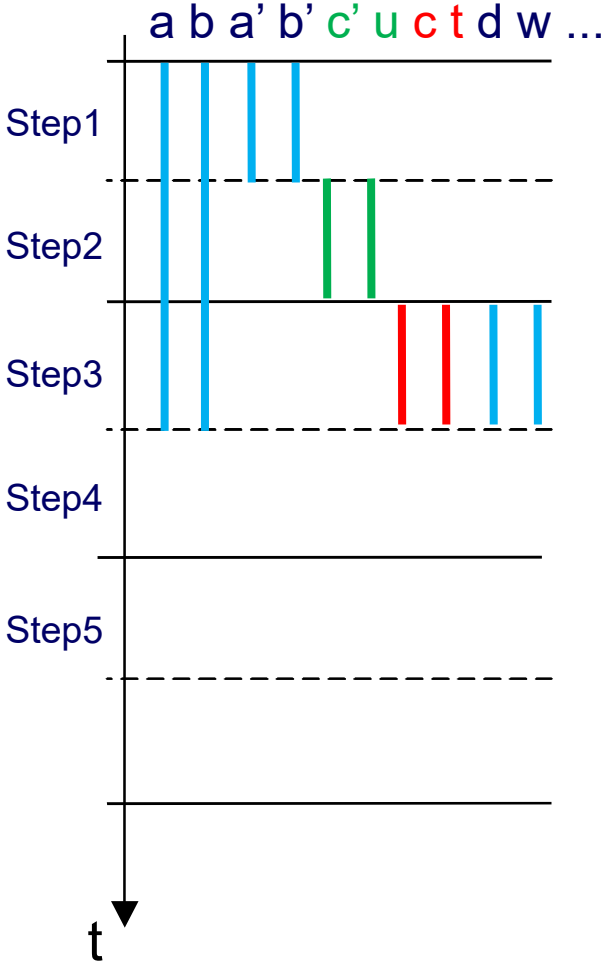


Data lifetime

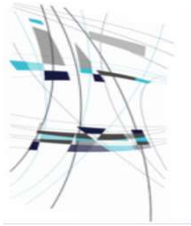
Register binding (ex. cont.)



Lea results



Data lifetime

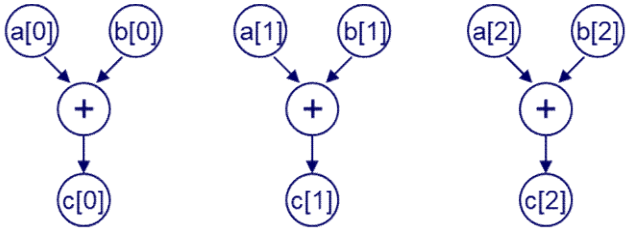


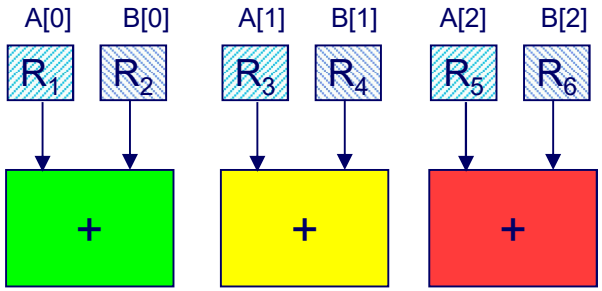
Binding problem

- Resource sharing often requires multiplexors
- Register and operator binding are interdependent w.r.t. specific objectives like area or clock frequency

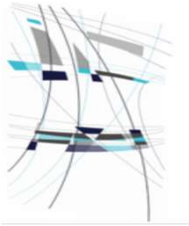
for i : 0 → 2
  c[i] = a[i] + b[i]

c[0] = a[0] + b[0]
c[1] = a[1] + b[1]
c[2] = a[2] + b[2]





Solution #1,
no sharing,
no mux

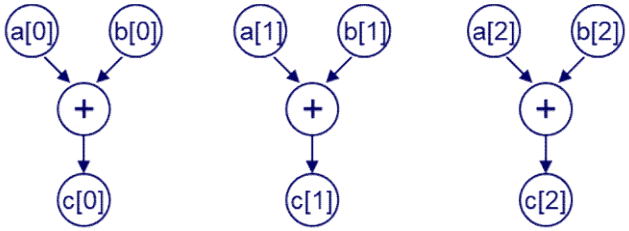


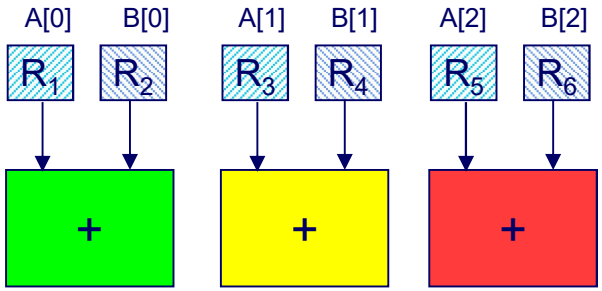
Binding problem

- Resource sharing often requires multiplexors
- Register and operator binding are interdependent w.r.t. specific objectives like area or clock frequency

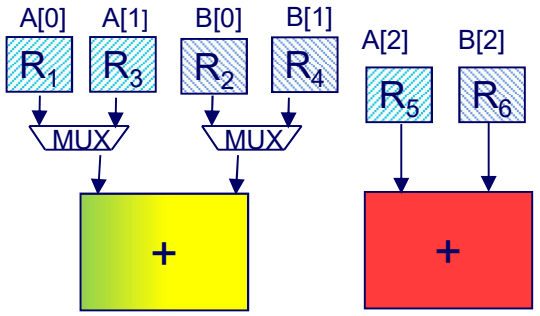
for i : 0 → 2
  c[i] = a[i] + b[i]

c[0] = a[0] + b[0]
c[1] = a[1] + b[1]
c[2] = a[2] + b[2]





Solution #1,
no sharing,
no mux



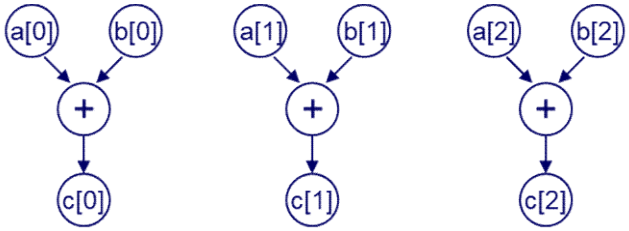
Solution #2,
Partial sharing,
Additional mux

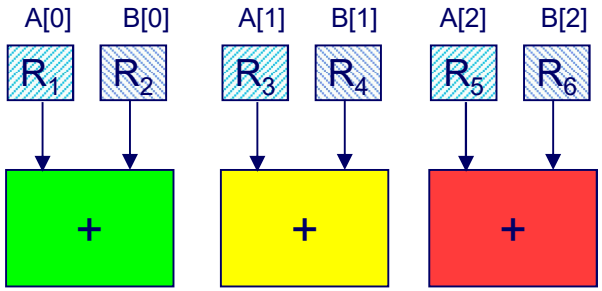
Binding problem

- Resource sharing often requires multiplexors
- Register and operator binding are interdependent w.r.t. specific objectives like area or clock frequency

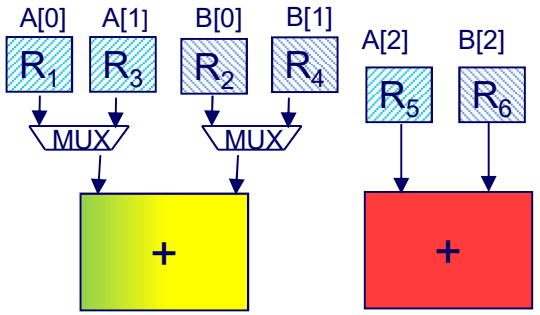
for i : 0 → 2
  c[i] = a[i] + b[i]

c[0] = a[0] + b[0]
c[1] = a[1] + b[1]
c[2] = a[2] + b[2]



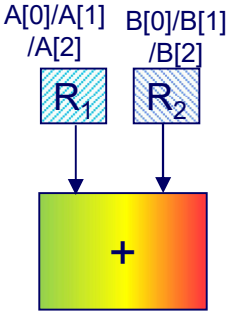


Solution #1,
no sharing,
no mux



Solution #2,
Partial sharing,
Additional mux

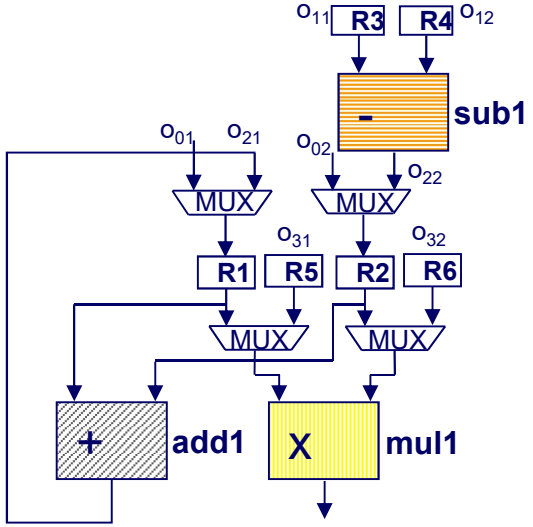
...



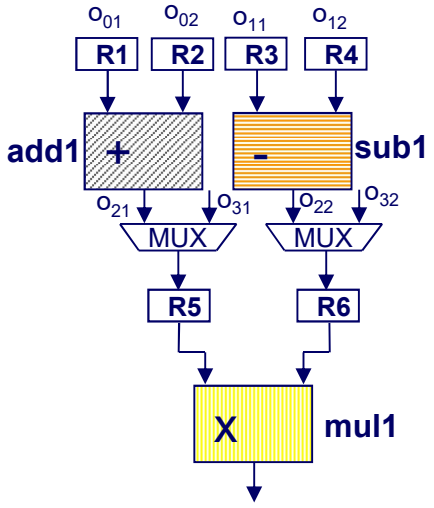
Solution #N,
Max sharing,
no mux

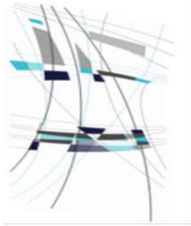
Binding problem

- Resource sharing often requires multiplexors
- Register and operator binding are interdependent w.r.t. specific objectives like area or clock frequency
- For a given number of registers/operators, the number of multiplexors may widely vary
- Minimizing the number of registers or operators does not imply minimizing area or maximizing clock frequency



V.S.

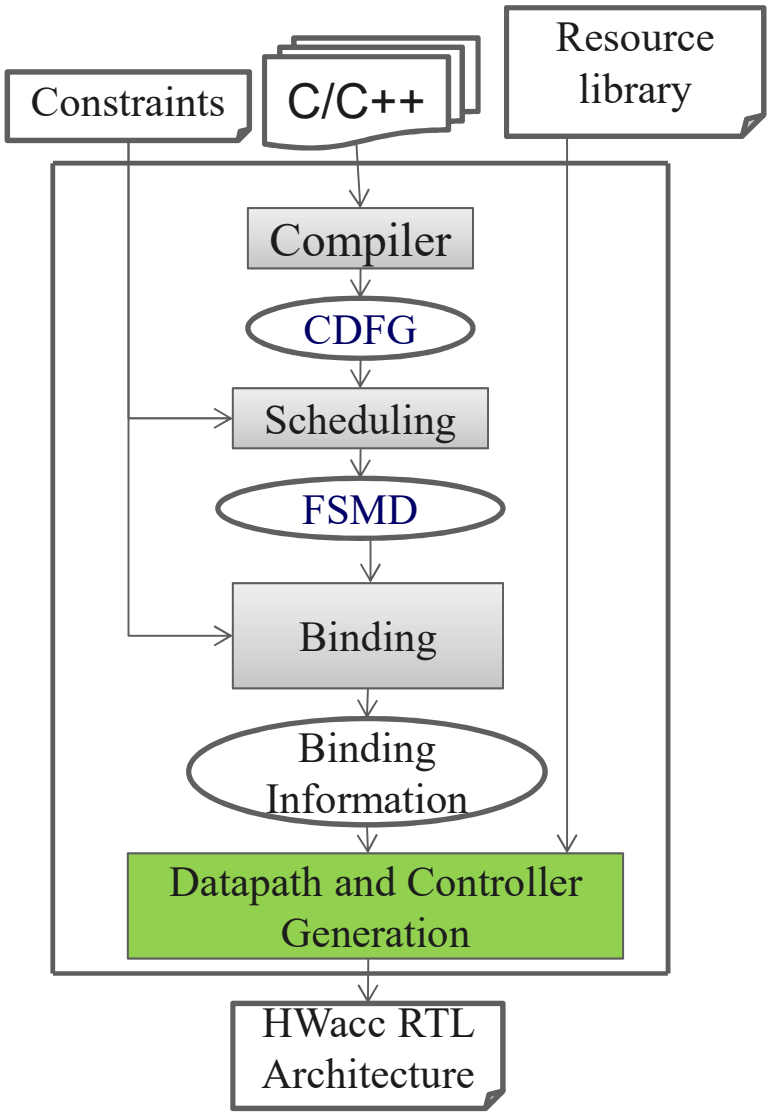




One missing step!

- Allocation
  - Defines the number and the type of HW resources
  - Resource constrained
    - Allocation → Scheduling →Binding
  - Time constrained
    - Scheduling → Allocation & Binding

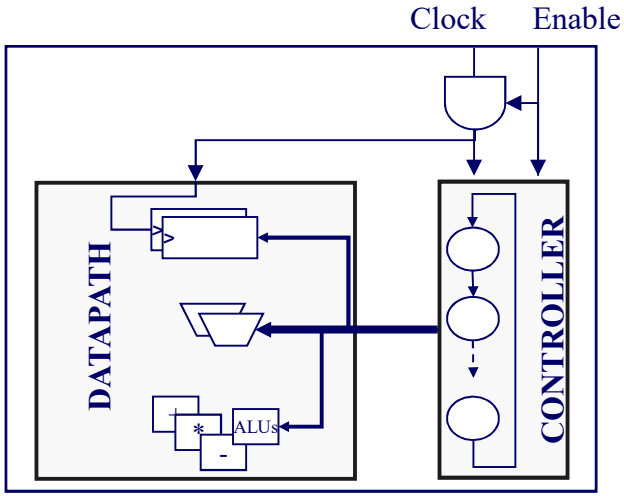
High-Level Synthesis flow



Architecture generation writes out the RTL source code (e.g. in VHDL) and writes out the CA or TLM description (e.g. in SystemC)

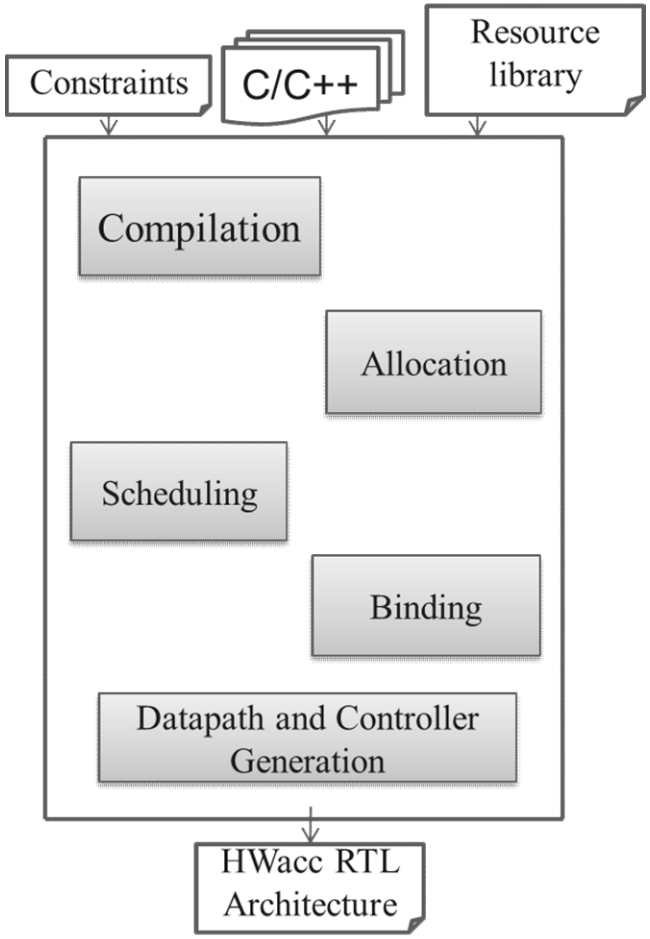
HWacc RTL Architecture

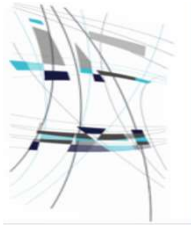
FIR_Fun.vhdl	***Top Level***
FIR_FSM.vhdl	***Controller***
FIR_DP.vhdl	***DataPath***



Many possible design flows

- No unique design flow i.e. many orders for the synthesis steps
  - Allocation → Scheduling
  - Scheduling → Allocation
  - Scheduling → Binding
  - Binding → Scheduling
  - Scheduling & Binding
  - ...
- Complex problems & different approaches
  - Models & algorithms
    - Exact approaches
    - Metaheuristics
    - Heuristics

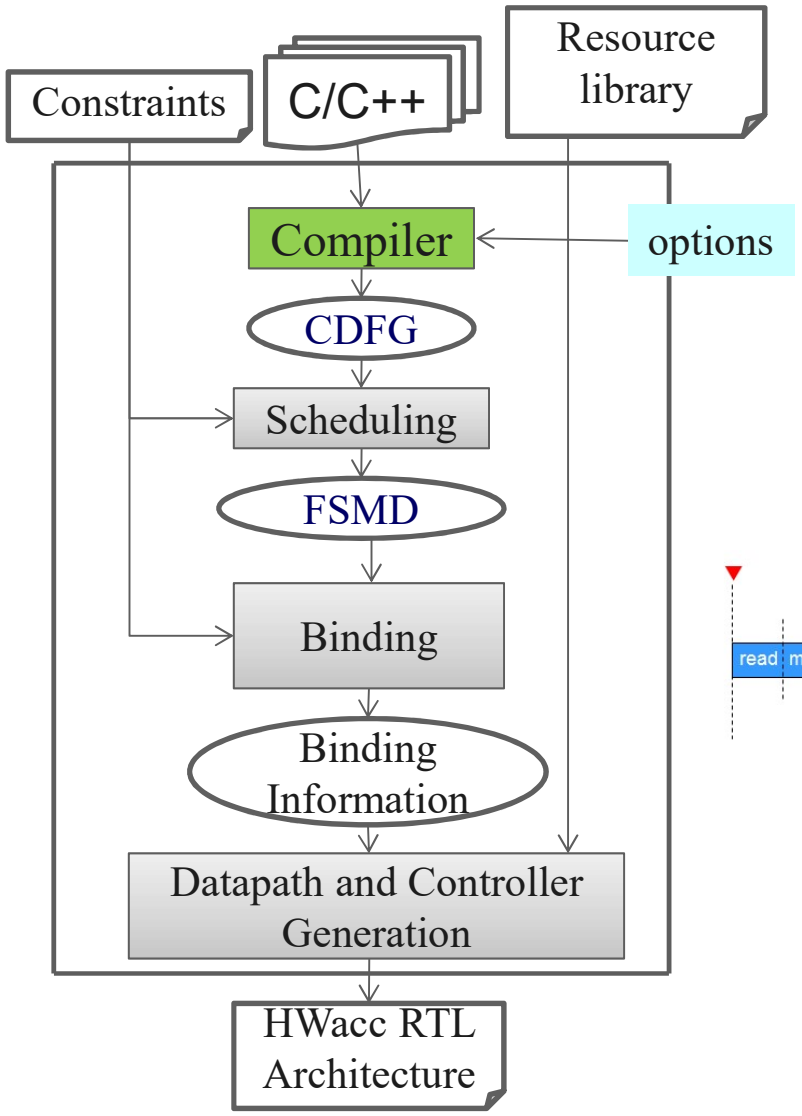




And a lot of other problems...

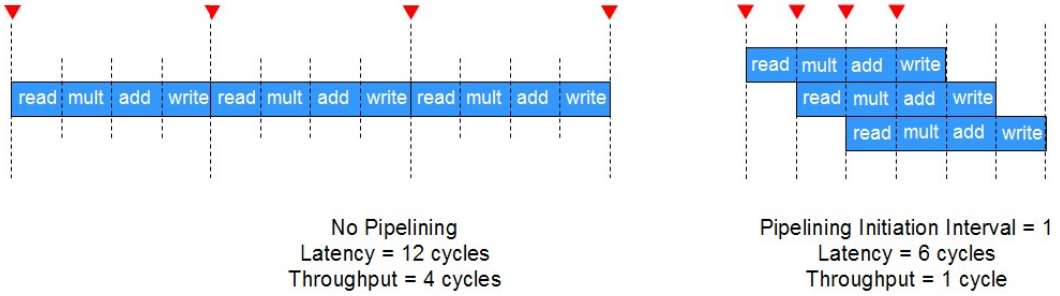
- Connection merging (Busses)
- Register merging (Register files...)
- Chaining
  - Several sequential operations in a clock cycle
- Multi-cycling
  - One operation takes more than one clock cycle to execute
- Pipelining
  - pipelined operator, pipelined datapath, pipelined controller
- Power/NRJ, Thermal aspects, LS-aware, P&R-aware...
- Bitwidth-aware, multimode…
- Control-dominated, data-dominated…
- …

High-level transformations



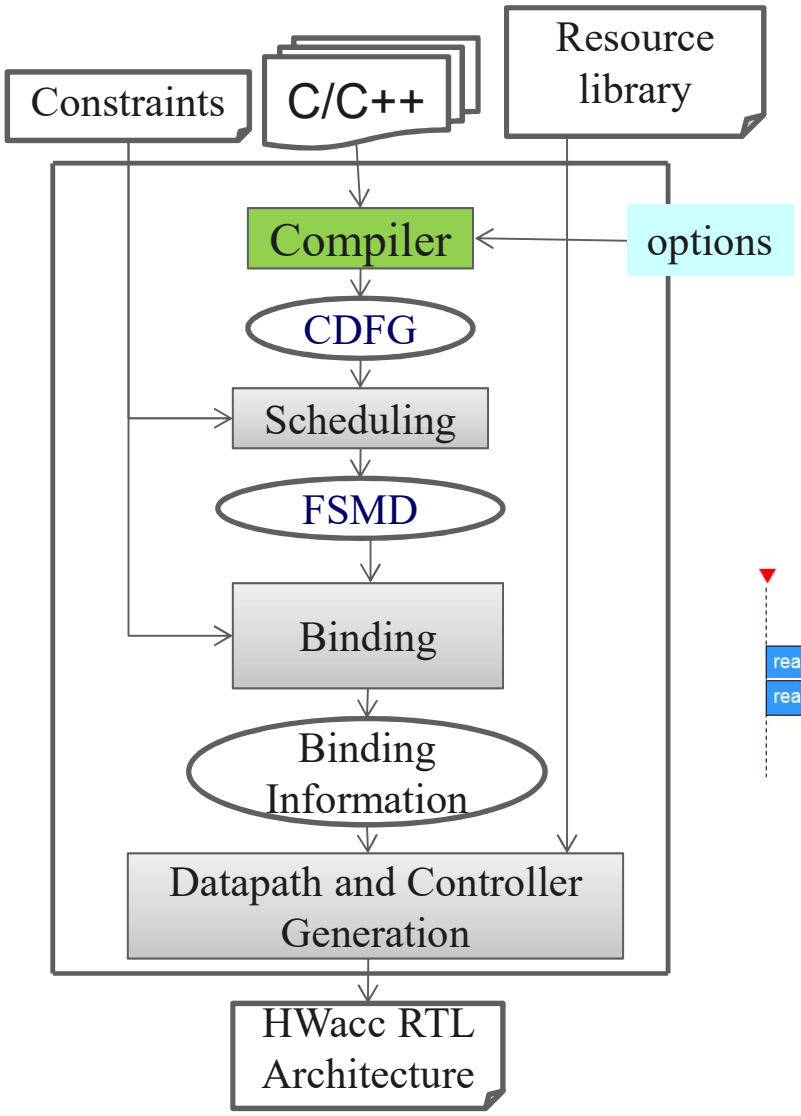
- Loops
  - Loop pipelining
  - loop unrolling
    - None, partially, completely
  - Loop merging
  - Loop tiling
  - …

```
for (i = 0; i<3; i++)
{
   out[i] = (in[i] * coef1) + coef2;
}
```



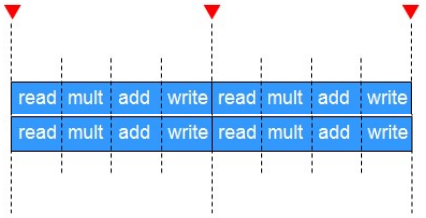
Pipelining runs several loop "stages" at the same time

High-level transformations

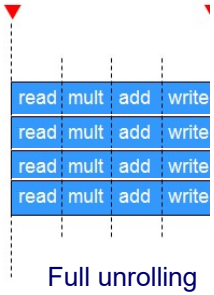


- Loops
  - Loop pipelining
  - loop unrolling
    - None, partially, completely
  - Loop merging
  - Loop tiling
  - …

```
for (i=0; i<4; i++)
   out[i] = (in[i] * corf1) + coef2;
```

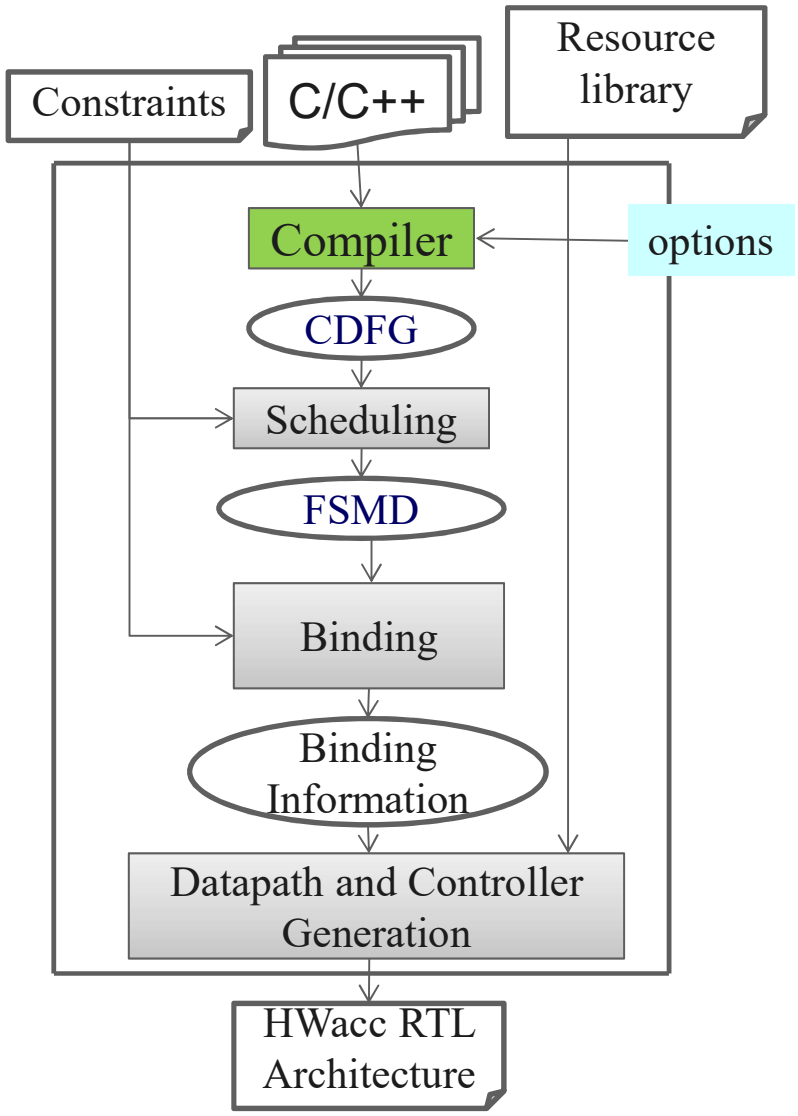


Partial unrolling (fact. 2)
Latency = 8 cycles



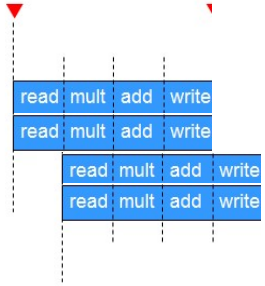
Full unrolling
Latency = 4 cycles

High-level transformations

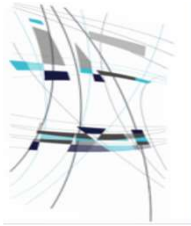


- Loops
  - Loop pipelining
  - loop unrolling
    - None, partially, completely
  - Loop merging
  - Loop tiling
  - …

```
for (i=0; i<4; i++)
   out[i] = (in[i] * corf1) + coef2;
```



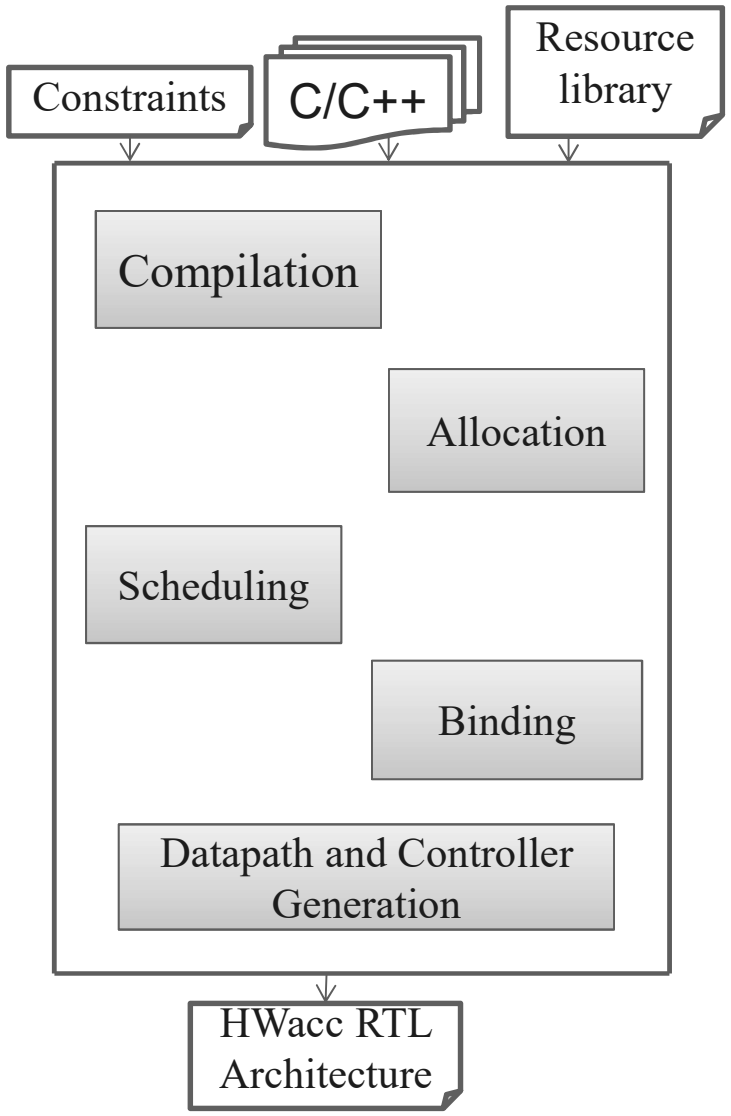
Partial unrolling (fact. 2) + Pipelining
Latency = 5 cycles
Throughput = 1 cycle



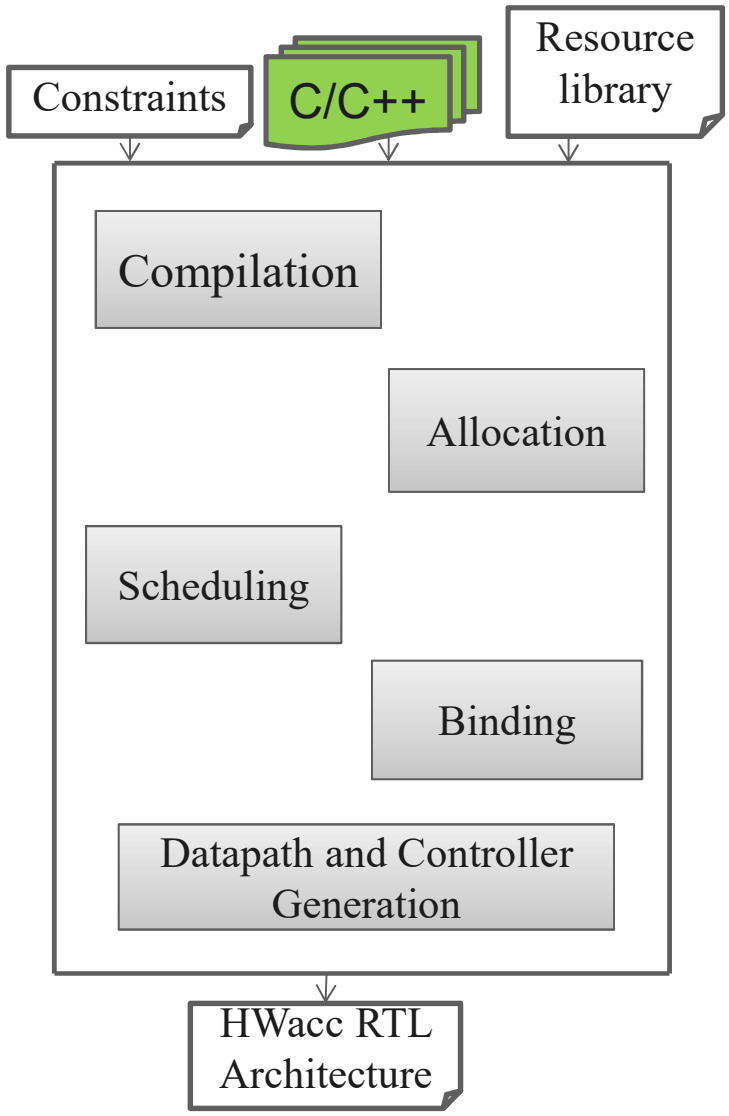
High-level transformations

- Loops
  - Loop pipelining
  - Loop unrolling
    - Partially or completely
  - Loop merging
  - Loop tiling
  - …
- Arrays
  - Constant arrays can be synthesized as logic
  - Scalars can be gathered into arrays
  - Arrays can be splitted
  - Arrays can be mapped on memory banks or synthesized as registers
  - …
- Functions
  - Function calls can be in-lined
  - Function is synthesized as an operator
    - Sequential, pipelined, functional unit…
  - Single function instantiation
  - …

HLS in brief and accelerated



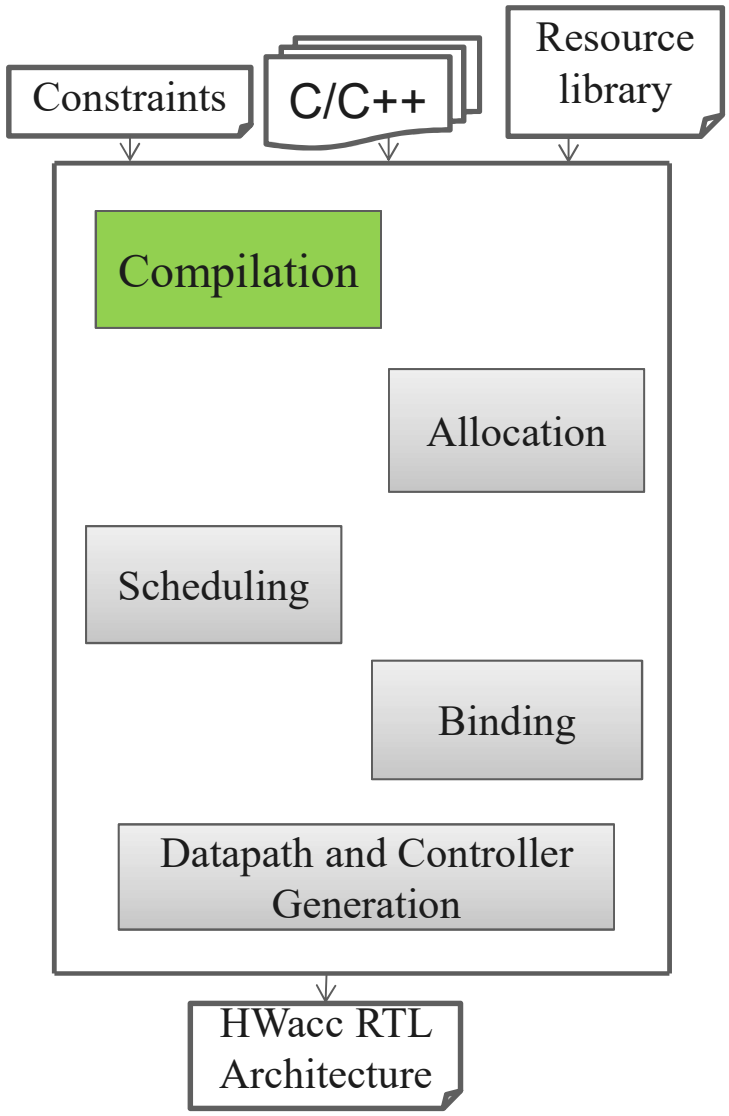
HLS in brief and accelerated

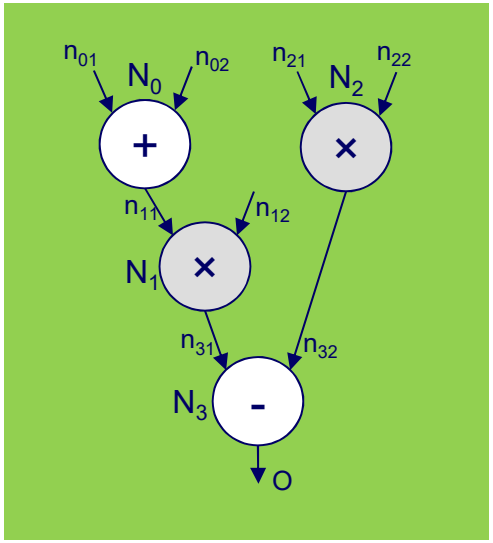


Input description

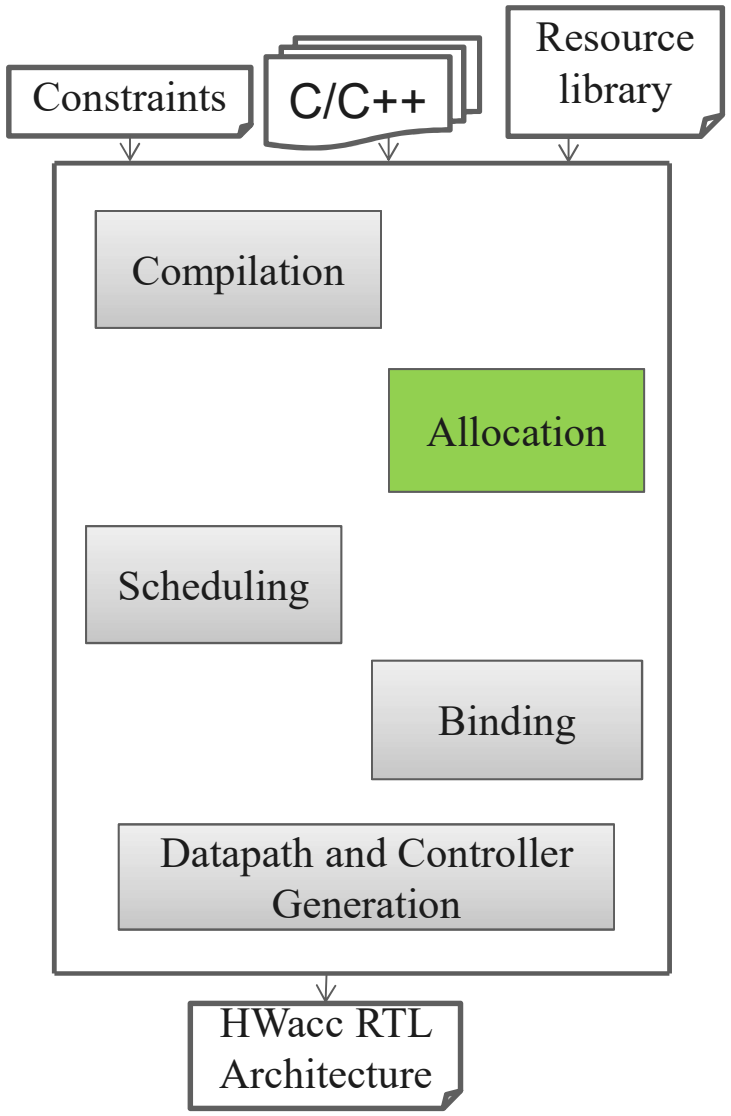
$O = ((n_{01}+n_{02}) * n_{12}) - (n_{21} * n_{22})$

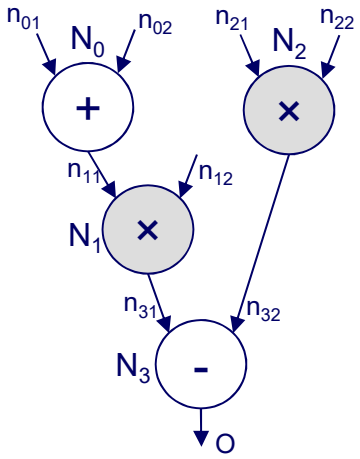
HLS in brief and accelerated

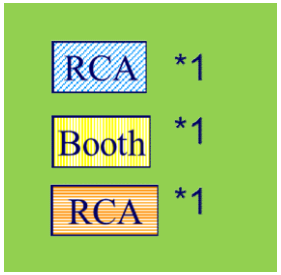




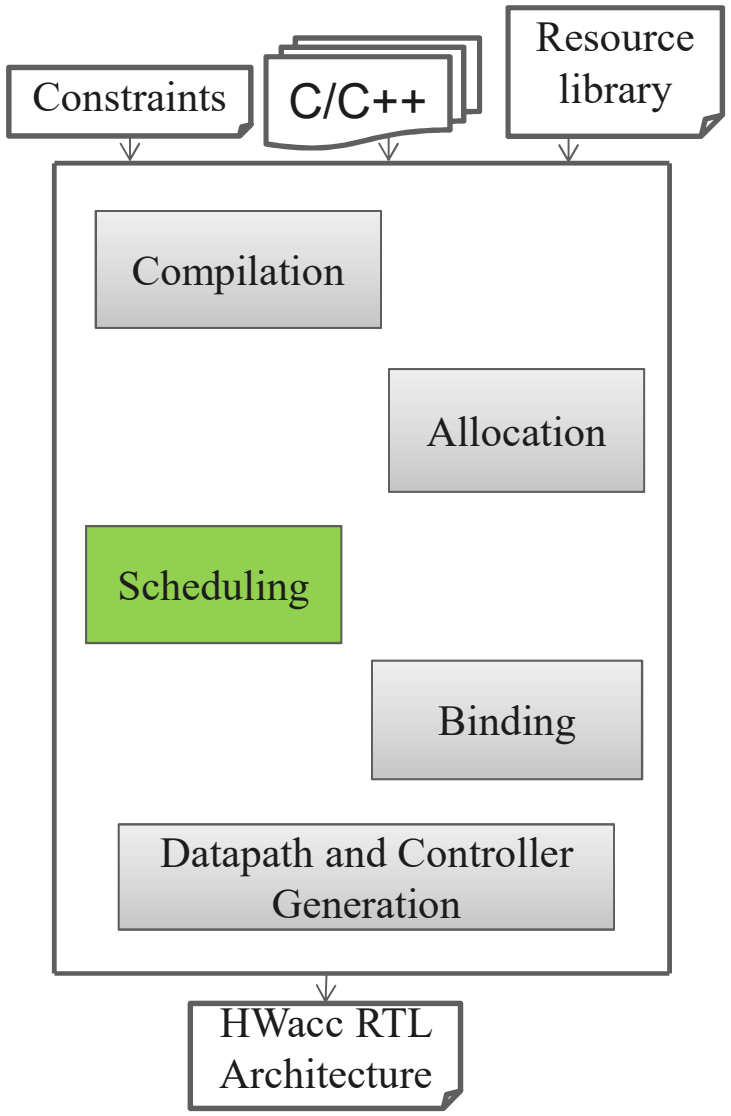
HLS in brief and accelerated

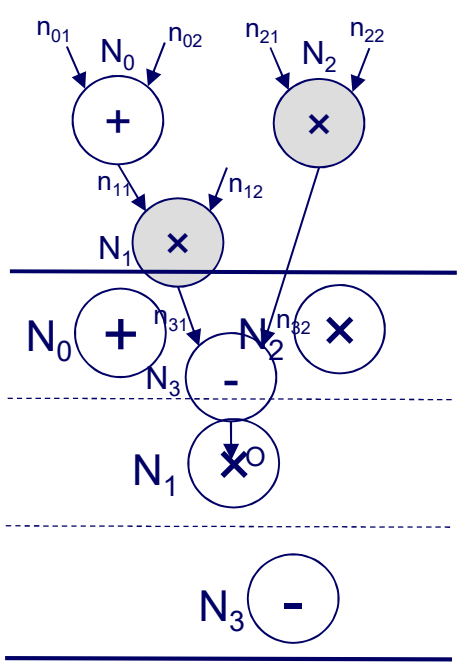




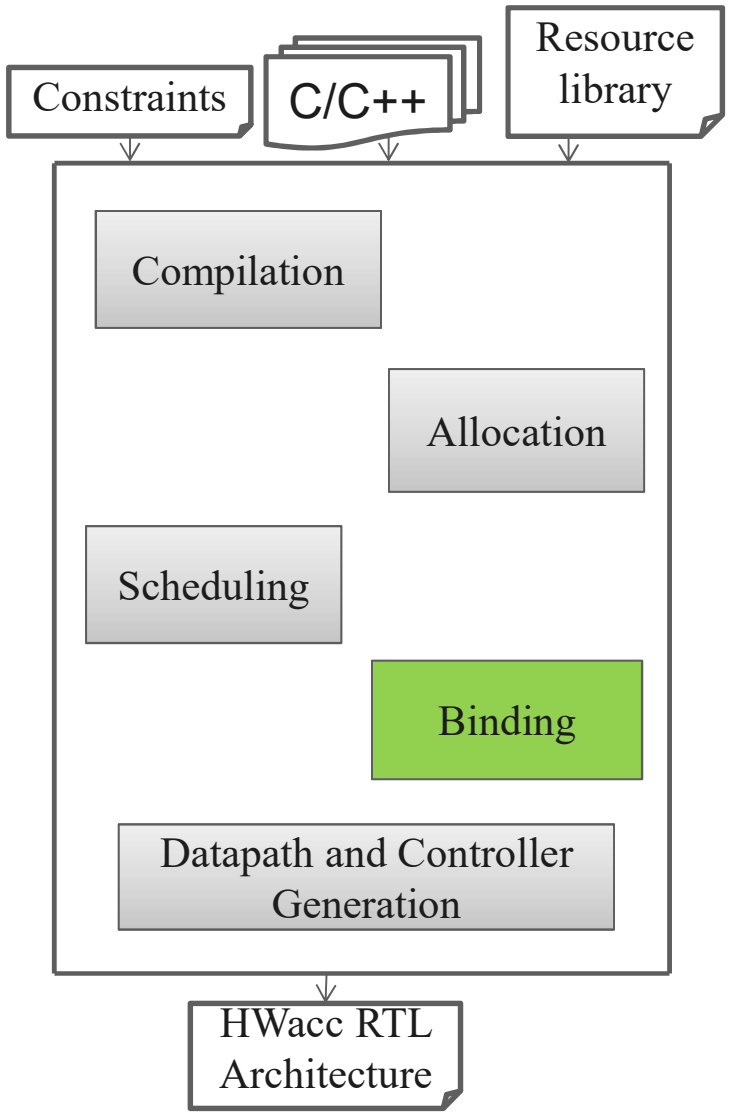


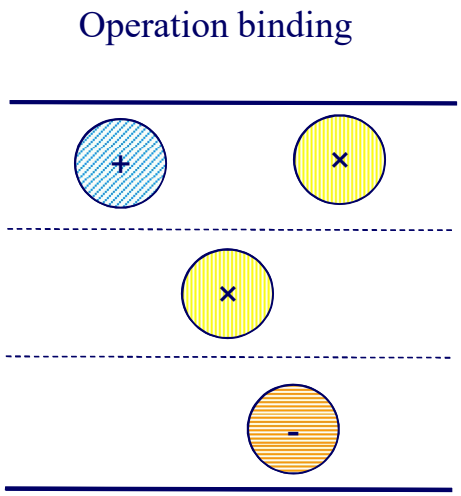
HLS in brief and accelerated





HLS in brief and accelerated





Data Binding

$n_{01} \rightarrow R_1$

$n_{02} \rightarrow R_2$

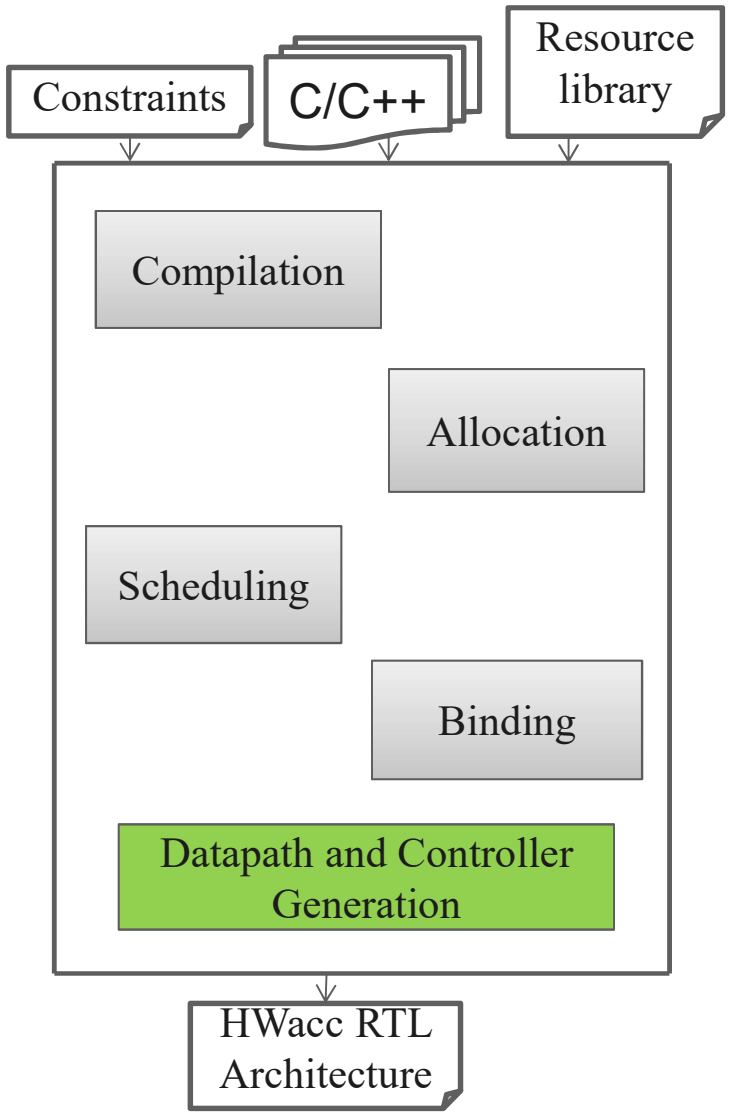
$n_{21}, n_{11} \rightarrow R_3$

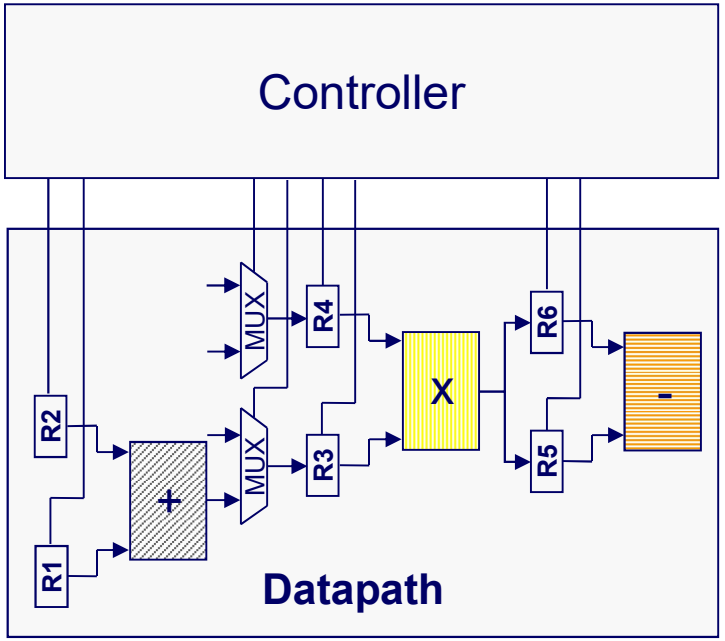
$n_{22}, n_{12} \rightarrow R_4$

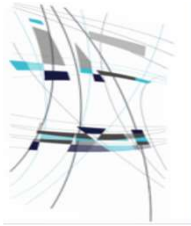
$n_{31} \rightarrow R_5$

$n_{32} \rightarrow R_6$

Wrap up

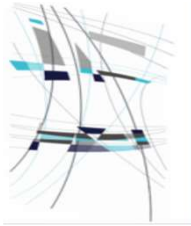

Constraints
C/C++
Resource library
Compilation
Allocation
Scheduling
Binding
Datapath and Controller Generation
HWacc RTL Architecture


Controller
MUX
R4
R6
X
-
R2
MUX
R3
R5
+
R1
Datapath



Conclusion (1/2)

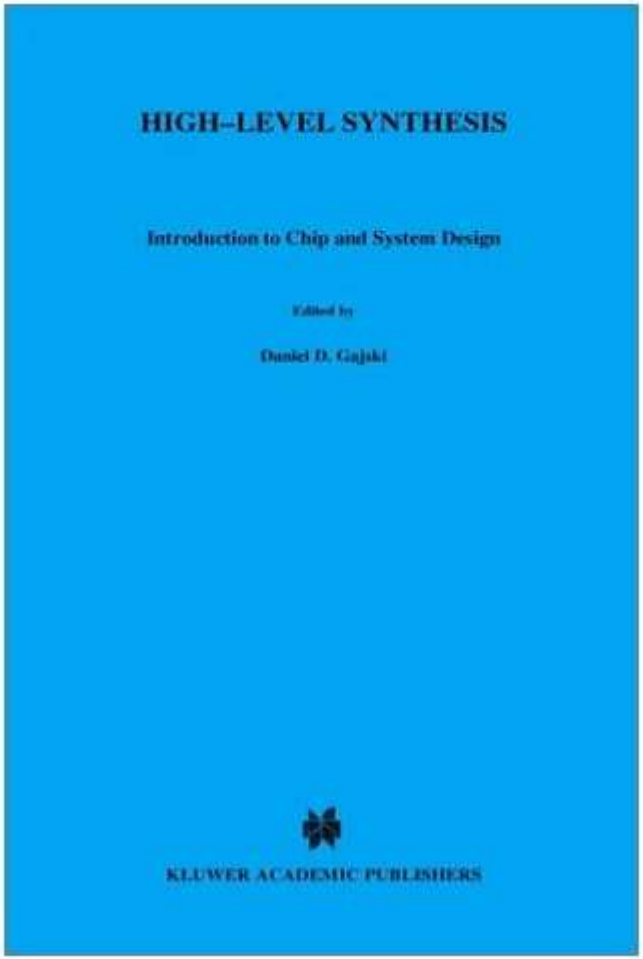
- HLS allows to automatically generate RTL architectures from an algorithmic description
- Different constraints and objectives
  - Time, area, power…
- Different input languages
  - C, C++, SystemC, Java, Ruby, Python…
- Different targets
  - ASIC, FPGA, SoG
- Different uses
  - Design, prototyping, HW-in-the-loop, DSE…
- Complex process which may vary and use different approaches coming with their parameters



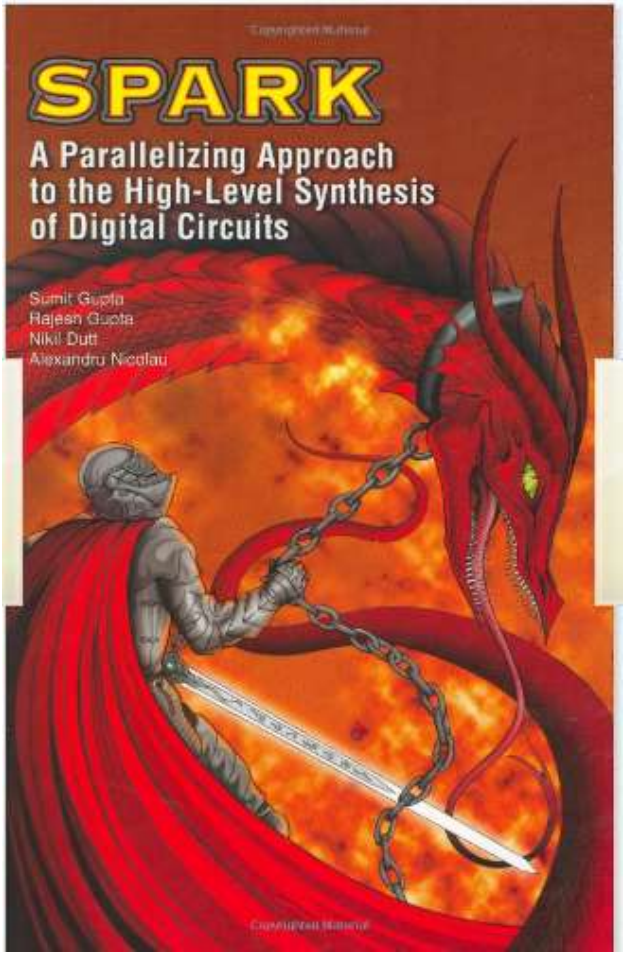
Conclusion (2/2)

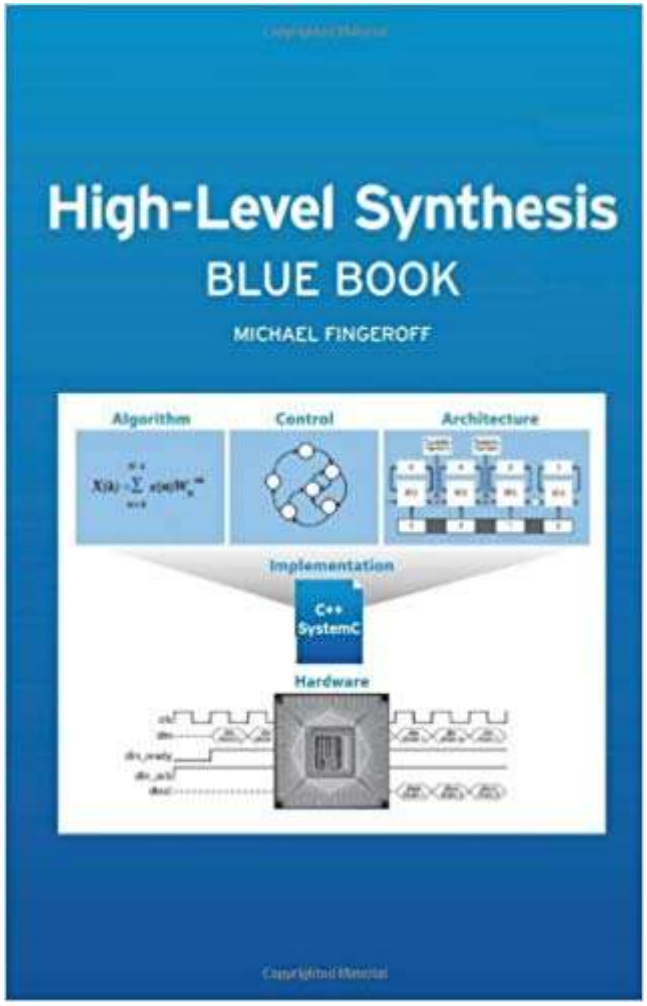
- From on input description, many RTL architectures can be designed
  - Design Space Exploration
- Commercial tools
  - Vivado HLS (Xilinx, ex. AutoESL), CatapultC (Siemens, ex Mentor Graphics), CWB (NEC), Stratus (Cadence, ex. Forte Design), Synphony (Synopsys, Ex. Synfora)…
- Academic tool
  - GAUT, Leg-up, Bambu, UGH…
- Many remaining open challenges
- Active research field
- HLS is still alive!

References (1/3)

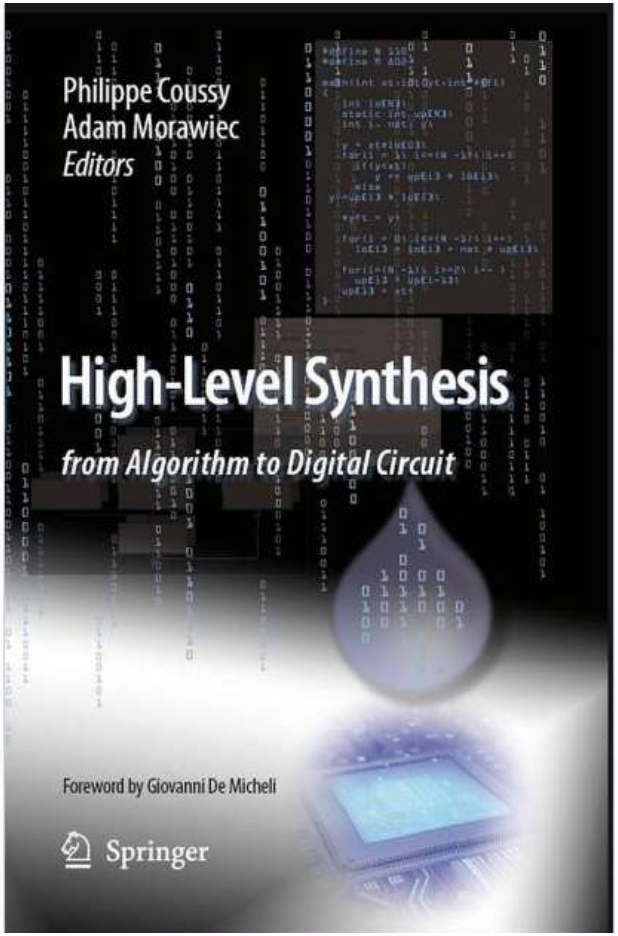


References (2/3)





References (3/3)



High-Level Synthesis: from theory to practice

Philippe COUSSY

philippe.coussy@univ-ubs.fr

2019, May 21

Université Bretagne Sud

ARCHI 2019 Lorient

Lab-STICC

cnrs