High-Level Synthesis

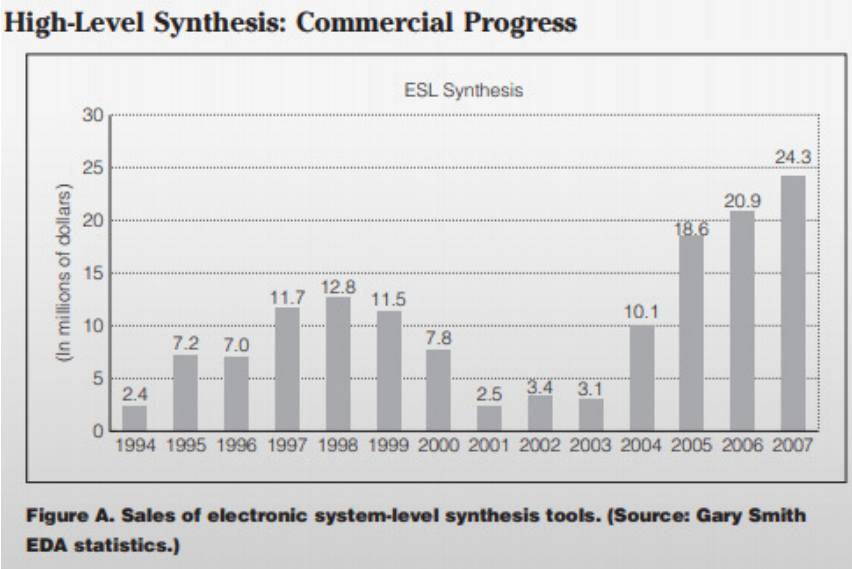
An easy *(or not so easy)* path to hardware programming

What is High-Level Synthesis (HLS)?

- High-Level Synthesis came to existance in the early 1980s. Although more prominent as a research topic, it tried to find its way to commercial use.
- At the time of its introduction to market, RTL was benefiting from revolutionizing features, such as automatic place and route technology.
- As such, HLS, still in its infancy, was an inferior design tool. It required the use of obscure languages (e.g. Silage) and implementation results were far from optimal.

What is High-Level Synthesis (HLS)?

- After years of trying to find its market share, HLS finally succeeded in the early 2000s, catapulted by the adoption of standard programming languages and algorithm improvements.



So, what is HLS like nowadays?

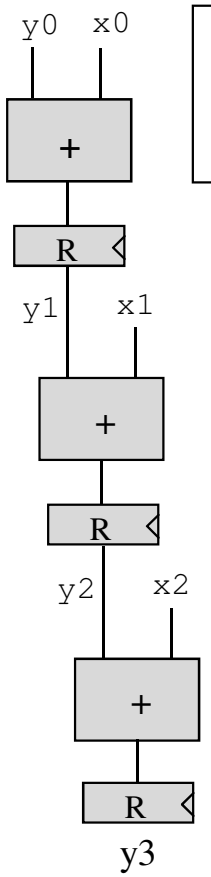
What is High-Level Synthesis (HLS)?

- High-Level synthesis is a method of translating a purely functional description of a task into RTL.
- HLS is typically coded in high-level programming languages, such as C/C++, systemC, OpenCL and even Matlab.
- The user is able to control certain design specification through the use of language specific directives.
- HLS is an iterative process.
    1. Hardware is synthesized.
    2. Tool generates reports that must be analyzed by the developer.
    3. Hardware directives and constraints are changed. Back to step 1

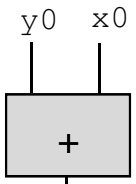
What is High-Level Synthesis (HLS)?

- RTL describes hardware as it is. HLS synthesis requires an RTL interpretation of an algorithm
  - Unlike in RTL, a clock is somewhat of an abstract concept in HLS implementation.
  - Hard to differentiate sequencial from combinational logic.
  - HLS hardware development is a lot faster. Time to market decreases significantly.
  - RTL design is a much slower process, but fully customizable.
  - RTL development requires a much deeper knowledge of hardware design (timing constraints, resources, strategies, ...).
- HLS relies on implementation goals set by the user
  - Is area an important metric? (number of FF, LUT, etc)
  - What is the desired speed of operation? (pipelining, clock frequency, parallelism,...)
  - And power consumption? (to DSP or not DSP, clock frequecy, ...)

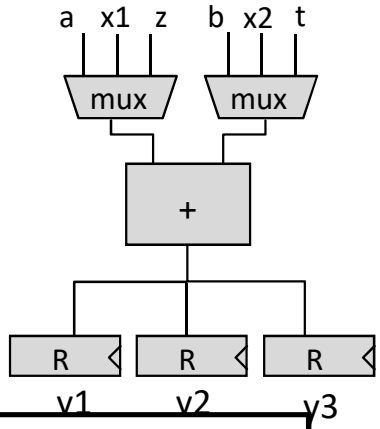
HLS vs RTL synthesis



Requires Control
↑ Area
↑ Latency
↑ Clock Frequency

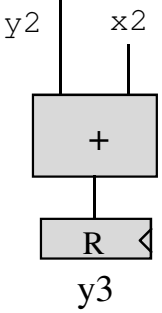


Requires Control
↑ Latency
↓ Area

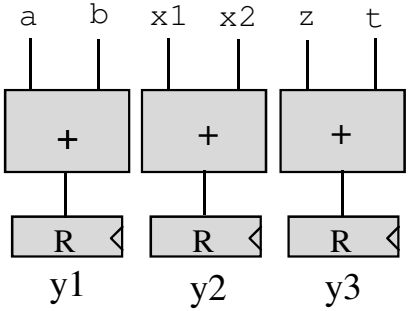


Which solution is better?

No Control Required
↓ Area
↓ Latency
↓ Clock Frequency



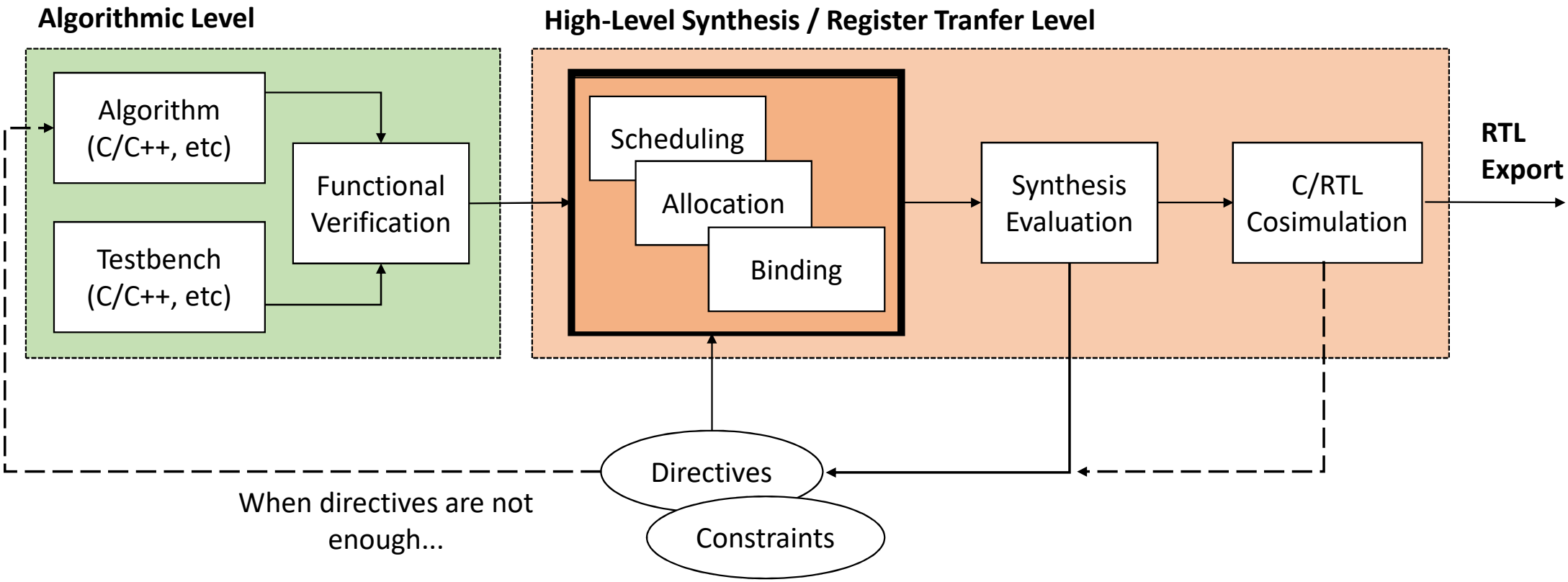
No Control Required
↓ Latency
↑ Area



When should we use HLS?

- **When time to market is importante**
  - Decreases migration time from a higher abstraction language to hardware.
  - Debug time is significantly reduced.
    - Code readability is increased.
    - Testbenching is easier to perform.
  - Easier and more abstract workflow - **Less prone to errors!**
- **When constraints are less stringent**
  - HLS automatically uses IPs from vendors.
  - Developers are less respondible for optimization and performance.

Workflow in HLS



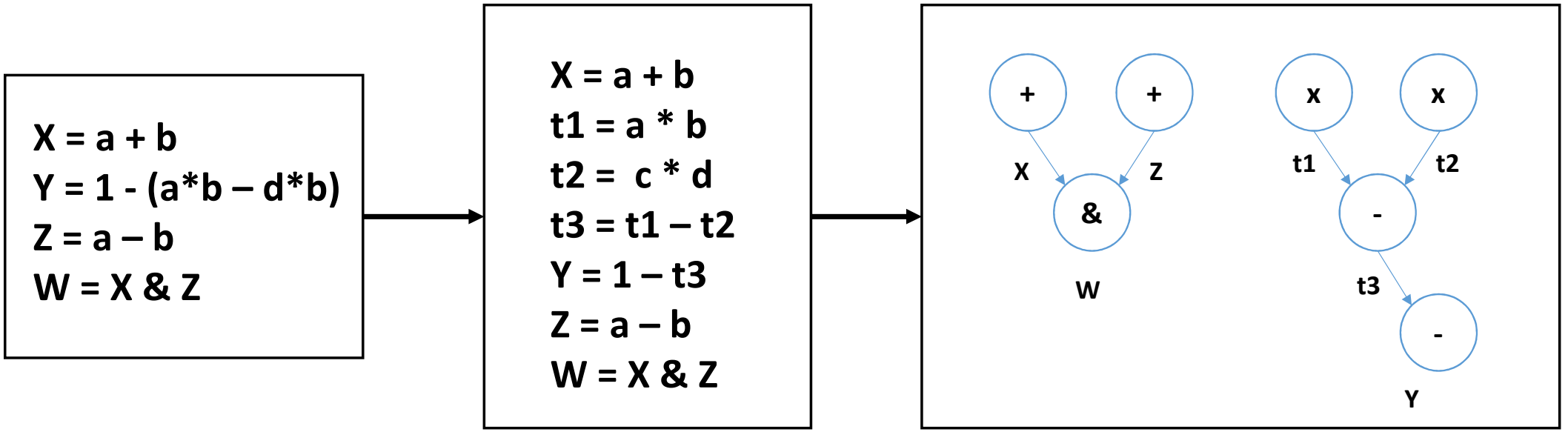
Scheduling in HLS

- Tasks are translated into a **graph problem**
- Determines which operations occur during each clock cycle based on:
  - Clock frequency
  - Time it takes for the operation to complete, as defined by the target device
  - User-specified optimization directives
- Extracts dependencies (task pipelining vs. parallelism)

Scheduling is impacted by the device. FPGAs with a better logic fabric can perform more tasks per clock cycle than a slower device.

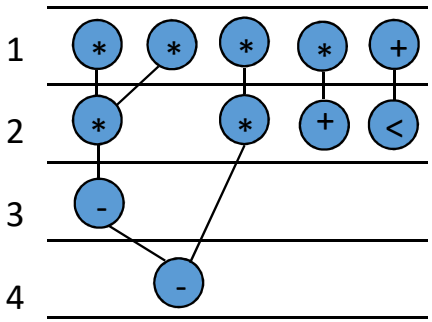
Scheduling in HLS – Data Flow Graphs

- Data flow graphs represent how data flows through a computation
- Breaks down algorithm into simple operations. This is performed by a compiler. **Now we can figure out dependencies!**



Simple scheduling strategies (an example)

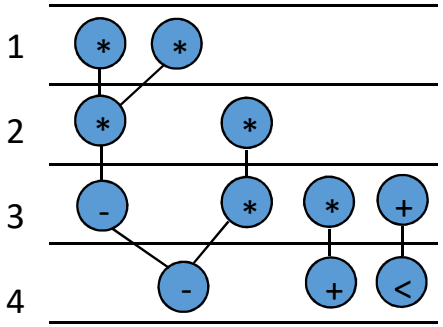
ASAP - As Soon As Possible



4 multipliers
1 subtractor
1 adder

6 functional units

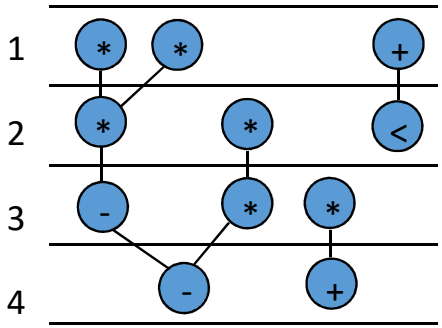
ALAP - As Late As Possible



2 multipliers
2 subtractors
1 adder

5 functional units

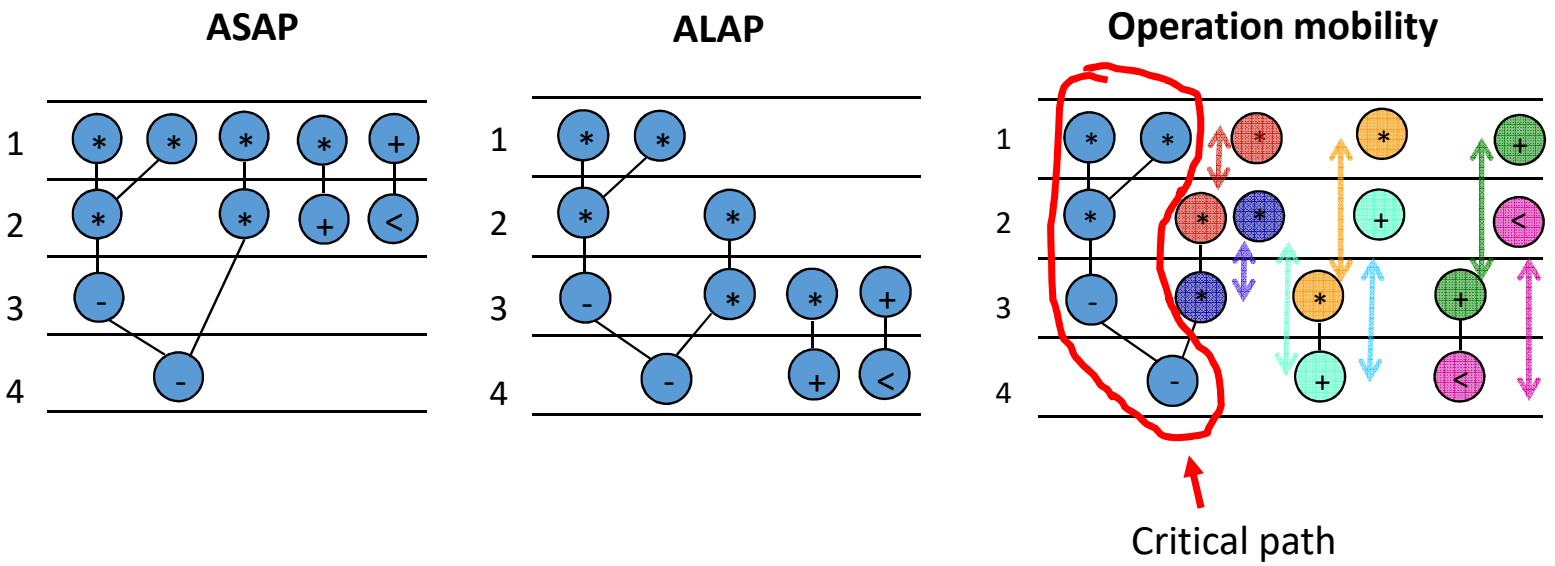
Resource constrained ASAP



2 multipliers
1 subtractor
1 adder

4 functional units

Simple scheduling strategies



Other scheduling techniques

Heuristic-based approaches are commonly used to solve a very complex problema such as scheduling.

- Neural Network scheduling
- Simulated Annealing
- Genetic algorithms
- Swarm inteligence

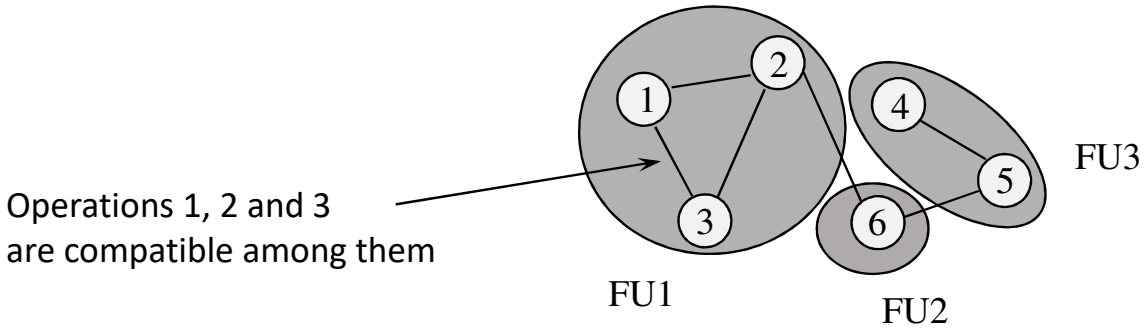
Allocation

- Allocation of functional units
  - The selected set must be capable of performing all the operations
  - The scheduling dictates the minimum set of FUs
  - FUs can be single cycle, multicycle, pipelined, iterative along time…
    - Different speed – area tradeoffs
  - FUs may perform different operations in non-overlapping times (eg. ALU)

- Allocation of storage units (registers, RAM, ROM, DRAM)
  - Storage is needed to hold the variables handled by the algorithm
  - Physical storage locations can be re-used along the lifetime of the algorithm
  - Storage is not constrained to a single large memory as in conventional CPUs
    - Small RAM memories may be allocated to diferent matrices
    - Registers can be shared by diferent sized variables
    - Large data sets can be mapped to off-chip dynamic memory (DRAM)

Binding

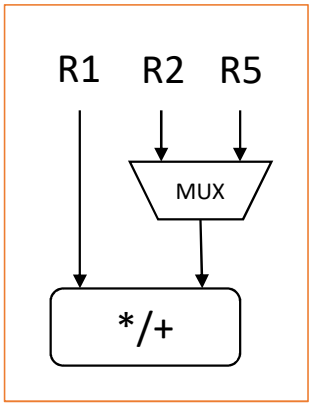
- Formulation as a graph problem
  - Nodes represent the operations;
  - Create undirected arcs between node **i** and **j** if
    - these operations can be performed by the same functional unit.
    - These operations are not executed in the same cycle (according to the scheduling)
  - Solution is to find the minimum number of complete sub-graphs
    - A complete sub-graph contains operations that can share one FU.

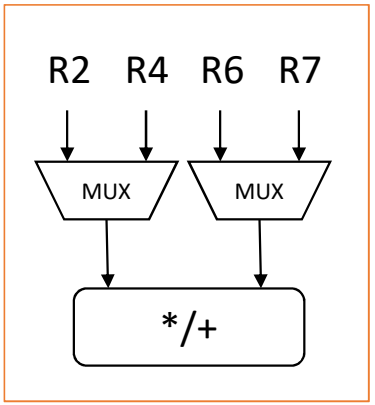


GC1

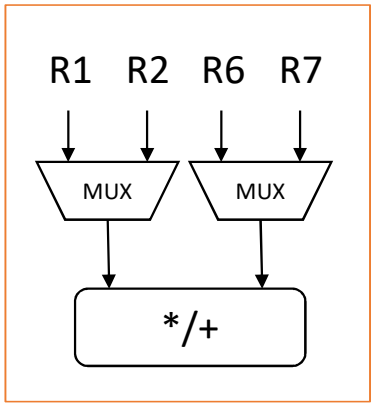
A simple problem

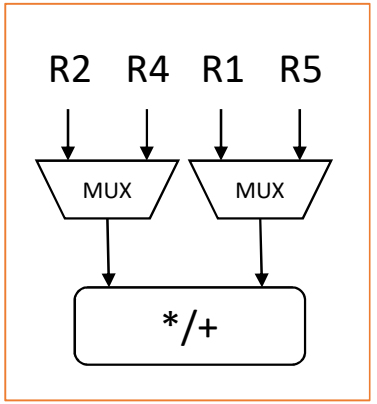
ALU1	ALU2
u1 = R1 + R2	**u2 = R2 + R4**
u3 = R1 * R5	u4 = R6 * R7





ALU1	ALU2
u1 = R1 + R2	**u2 = R2 + R4**
u4 = R6 * R7	u3 = R1 * R5





Slide 16

GC1 Guilherme Carvalho; 02/12/2020

Scheduling, Allocation & Binding

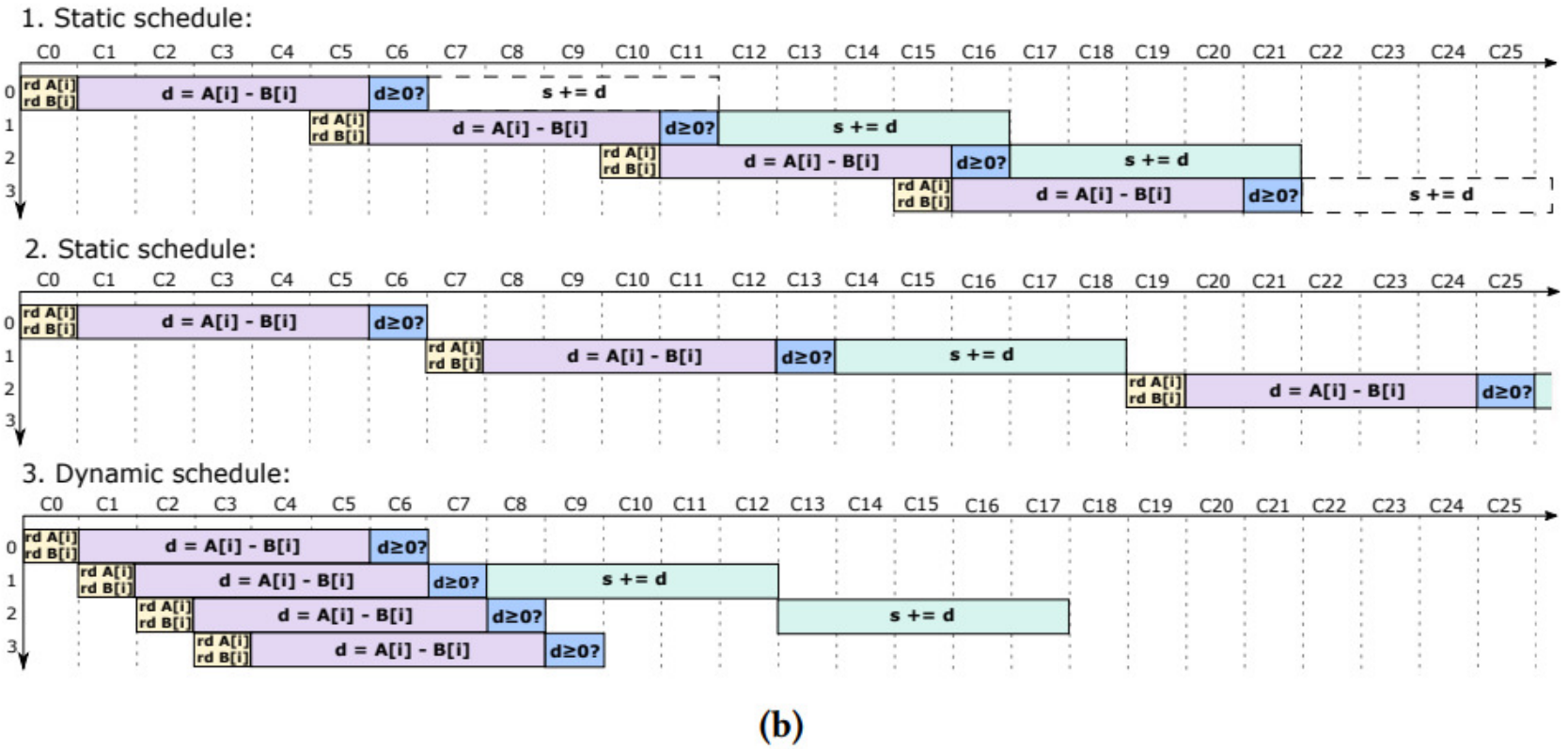
Two main types

- Static
  - Performed at compile time.
  - Very hard to determine dependencies.
  - Performance limited by high latency.
- Dynamic
  - Performed *on the fly.*
  - Better solution achievable.
  - Dependencies do not need to be determined at compile time.
  - Problem is a lot more complex.

Dynamic vs Static Scheduling

```
float d, s = 0.0;
int i;
for (int i=0; i<100; i++){
      d = A[i] - B[i];
      if (d >= 0)
            s += d;
}
 A[0]=1.0; B[0]=3.0;
 A[1]=4.0; B[1]=3.0;
 A[2]=2.0; B[2]=2.0;
 A[3]=4.0; B[3]=5.0;
          ⋮
```

(a)



(b)

Source : Dynamically Scheduled High-level Synthesis. Josipovic, 2018

Vivado HLS

- Vivado HLS is one of the many popular high-level synthesis tools available on the market
- Developed and maintained by Xilinx, Inc. Supports devices starting from the 7-series
- Programming in C/C++

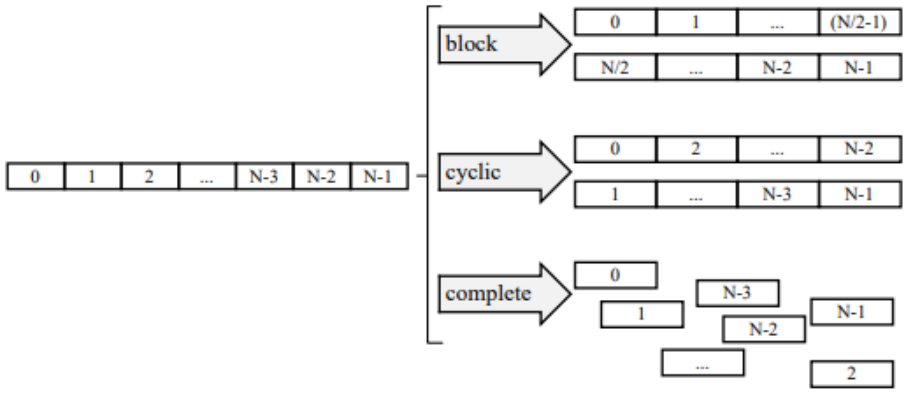
Vivado HLS - Directives

- **Interface**
  - Allows developer to specify I/O protocols and bandwidth
  - Extremely importante when using data streaming
- **Unroll**
  - Default behavior when using nested loops
  - Simplifies hardware by combining loops
- **Pipeline**
  - Reduces the Initiation Interval (II) of certain tasks
  - Tasks are scheduled in such way that data dependencies minimize throughput
- **Resource**
  - Defines which resource should be specifically allocated

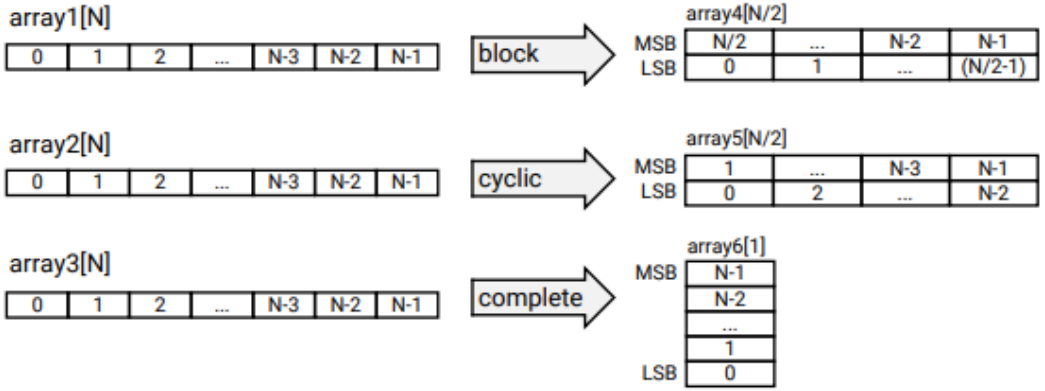
Vivado HLS - Directives

- Array Partitioning and Reshaping
  - Can solve problems of simultaneous access to the block RAM – Remember BRAMs have a limited number of ports!
  - Significantly improves pipelining

Array Partitioning



Array Reshaping



Source : UG902 (v2019.1) July 12, 2019