Chair for Circuit and System Design

High Level Synthesis

-

HLS

CHEMNITZ UNIVERSITY OF
TECHNOLOGY

Literature

- **High-Level Synthesis Blue Book.** Michael Fingeroff Intermediate

  online: http://www.eet.bme.hu/~timar/data/hls bluebook uv.pdf

- **High-Level-Synthesis - Introduction to Chip and System Design.** D.D. Gajski, Kluwer Publishers, 1992 Academia

- **ASIC - Entwurf und Test.** Herrmann G. and D. Müller. Carl-Hanser-Verlag, 2004 (german)

General issues on Circuit Design

HLS
-
Fundamentals and Placement in the Design Flow

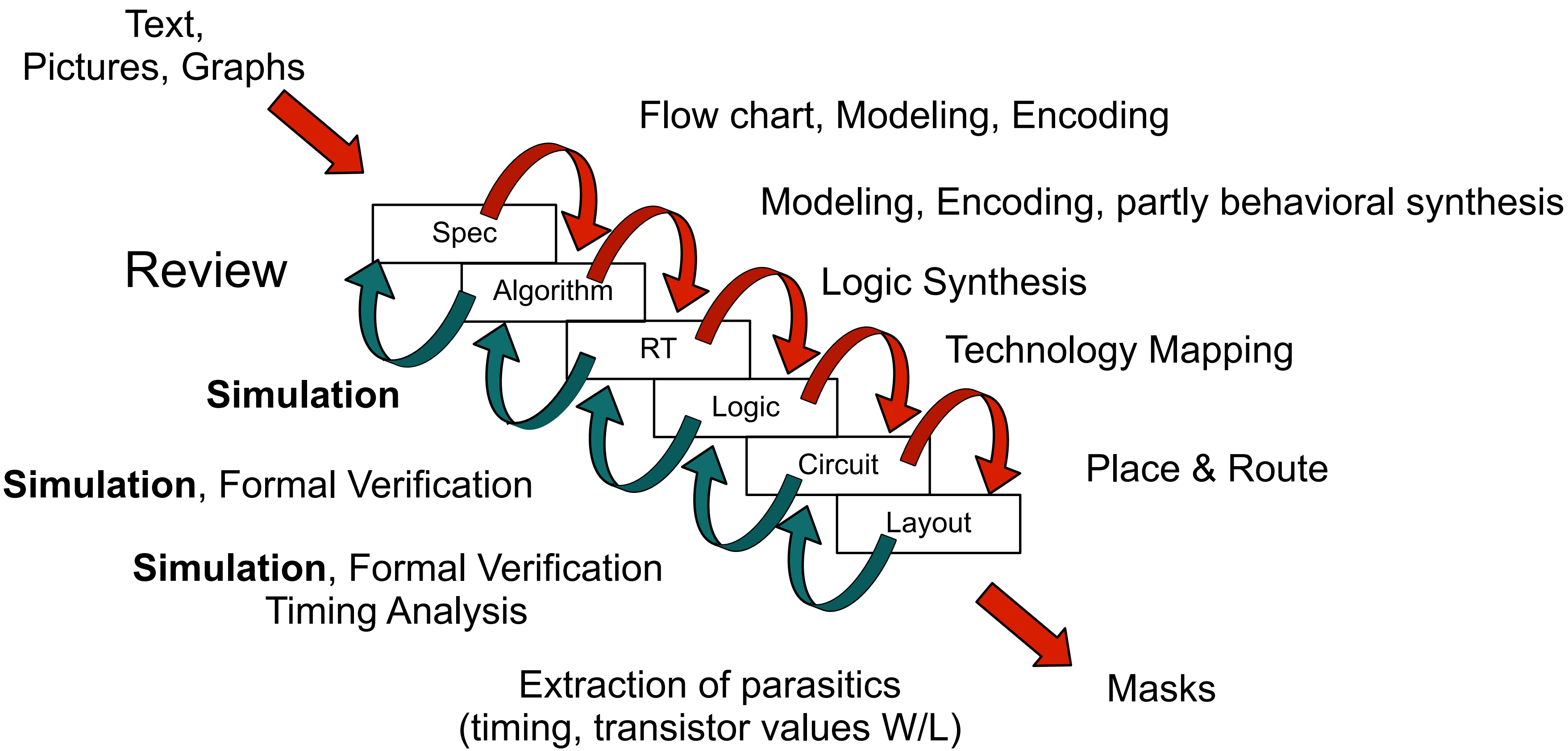
Fundamentals – Concept and Definition

High Level Synthesis:

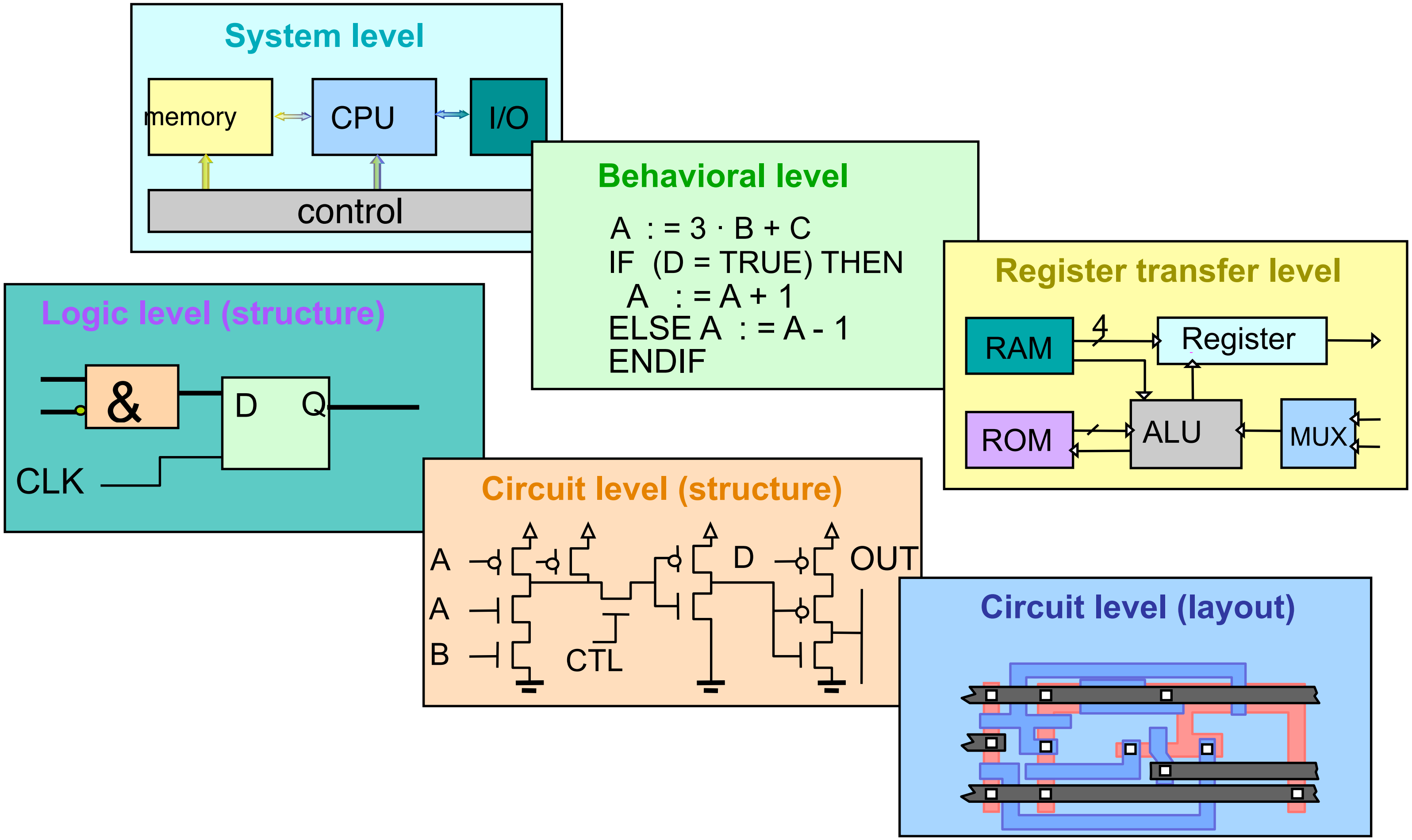
- High Level Synthesis (HLS) is a possible step in the design flow of digital hardware systems. Design descriptions are synthesized from **behavioral algorithmic level** into the **structural RT level.**
- **Logic** and **arithmetic** operations from the algorithmic level are mapped onto **control path** and **data path** of the RT level.

Synonyms: Architecture-Synthesis, Algorithm-Synthesis, Behavioral-Synthesis

Waterfall Model: Prove of Correctness


Text,
Pictures, Graphs
Flow chart, Modeling, Encoding
Modeling, Encoding, partly behavioral synthesis
Logic Synthesis
Technology Mapping
Place & Route
Spec
Algorithm
RT
Logic
Circuit
Layout
Review
Simulation
Simulation, Formal Verification
Simulation, Formal Verification
Timing Analysis
Extraction of parasitics
(timing, transistor values W/L)
Masks

Design Levels



System level

Behavioral level

```
A  : = 3 · B + C
IF  (D = TRUE) THEN
  A   : = A + 1
ELSE A  : = A - 1
ENDIF
```

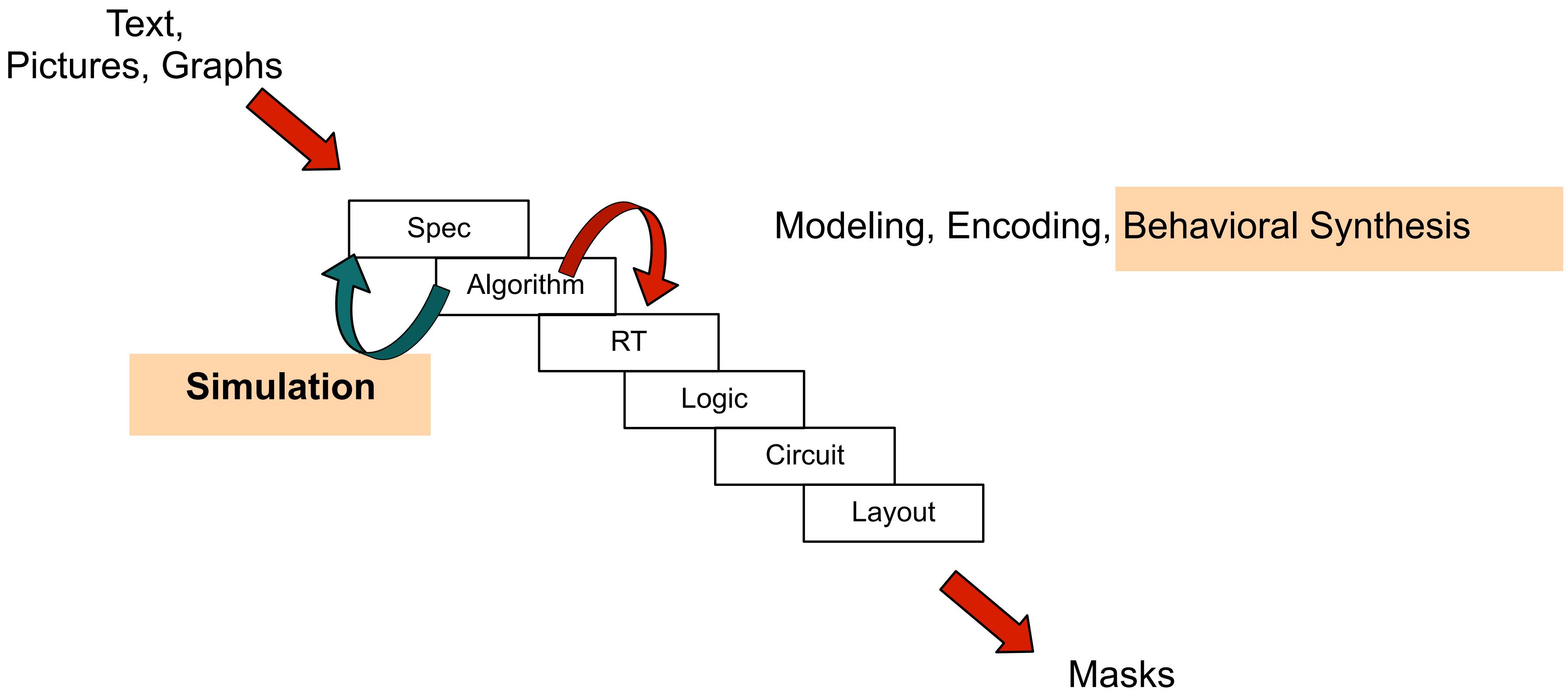
Register transfer level

Logic level (structure)

Circuit level (structure)

Circuit level (layout)

Waterfall Model: HLS-Part



Text,
Pictures, Graphs

Spec

Algorithm

RT

Logic

Circuit

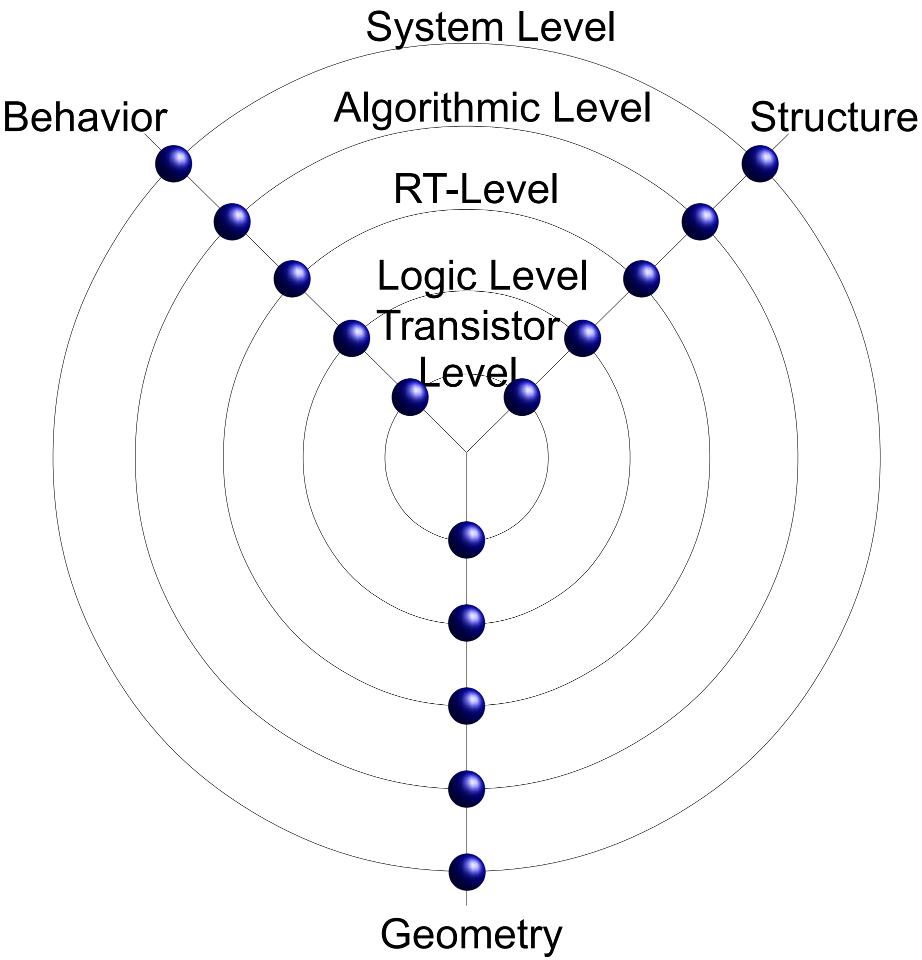
Layout

Masks

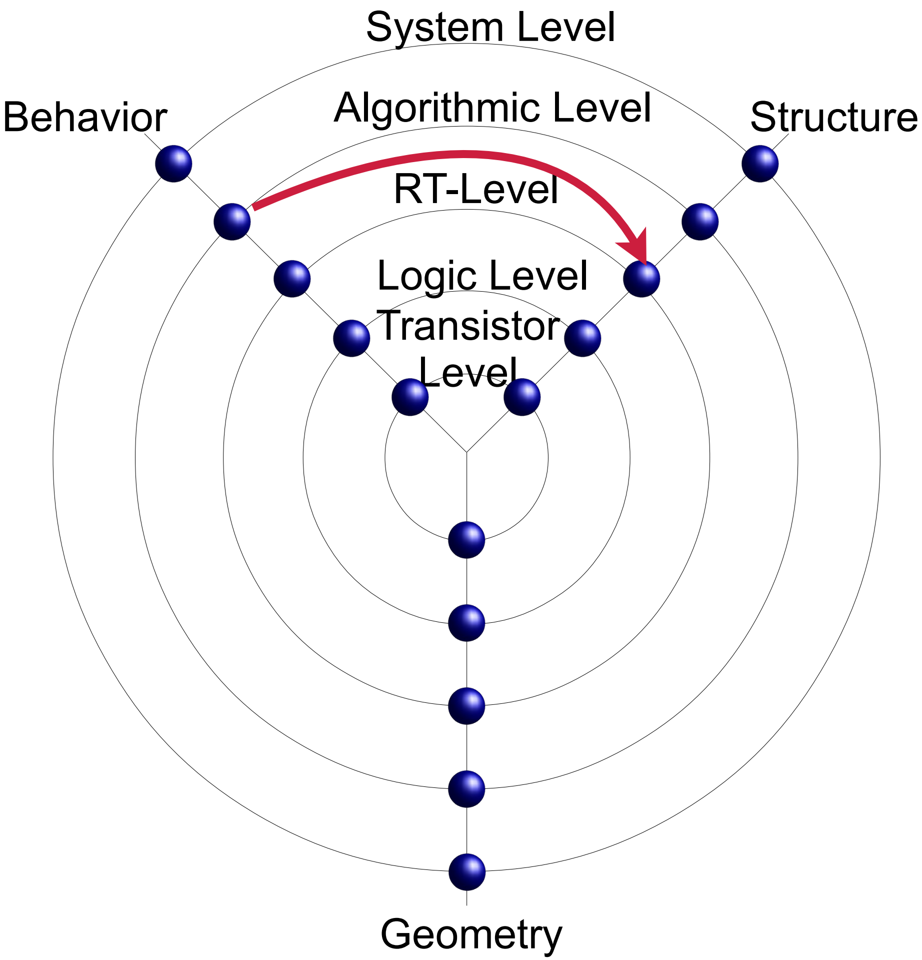
Modeling, Encoding, Behavioral Synthesis

Simulation

Fundamentals – Y-Diagram



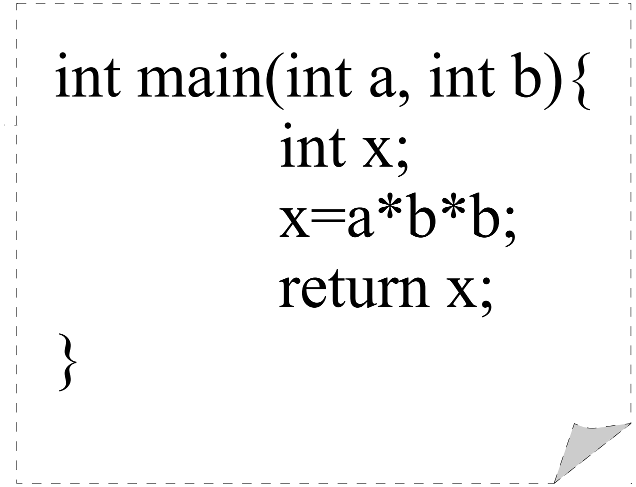
Fundamentals – Y-Diagram



Fundamentals – Algorithmic Level

Describes algorithms and functions of a system as:

- concurrent/parallel portions of an algorithm
- signal based communication



```
int main(int a, int b){
        int x;
        x=a*b*b;
        return x;
}
```

Defined by:

- functions, procedures, processes or threads
- control structures
- signal/data communication

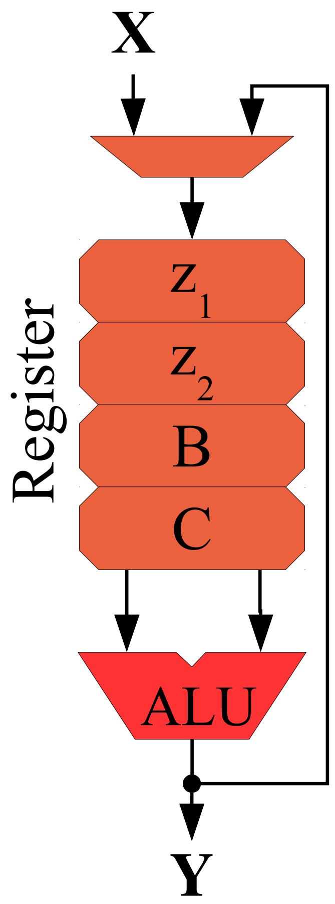
No sense of later implementation or partitioning regarding structure

No sense of timing... only sequence of operations!

Notions of computation

```
int compute(x){
 int y;
 y = x*x + B*x + C;
 return(y);
}
```

Notions of computation



```
int compute(x){
 int y;
 y = x*x + B*x + C;
 return(y);
}
```

$z_1 := X$

$z_1 := z_1 * X$

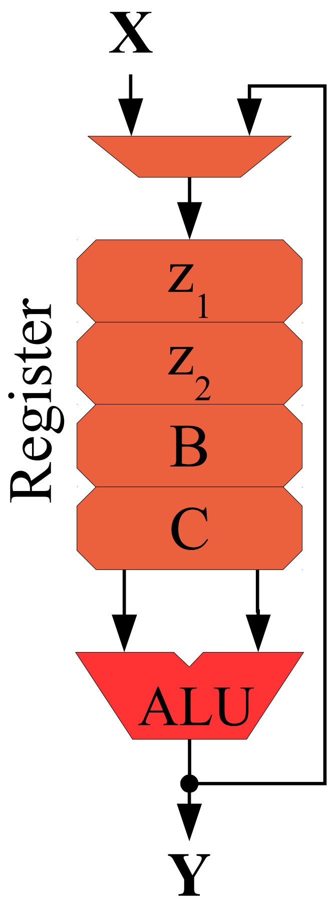
$z_2 := B * X$

$z_2 := z_2 + C$

$Y := z_1 + z_2$

Temporal (Software on a CPU)

Notions of computation



```
int compute(x){
 int y;
 y = x*x + B*x + C;
 return(y);
}
```

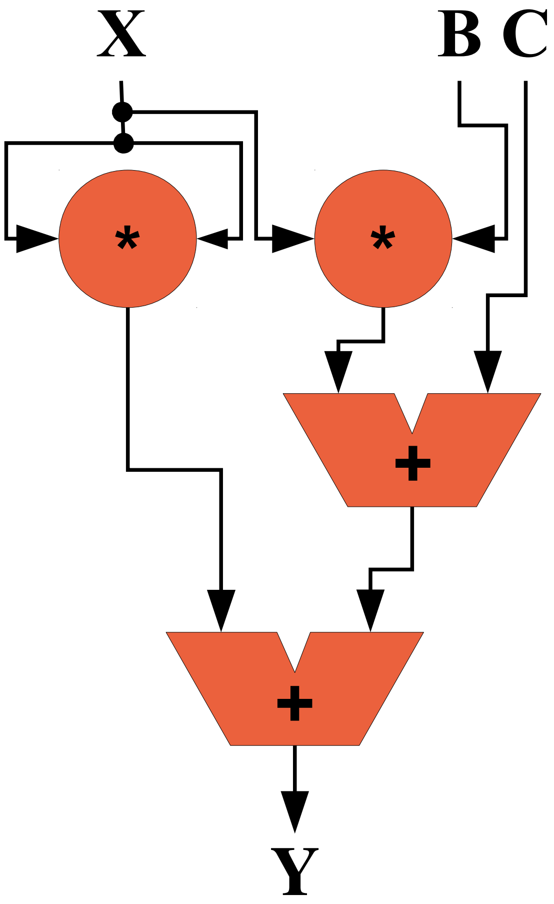
$z_1 := X$

$z_1 := z_1 * X$

$z_2 := B * X$

$z_2 := z_2 + C$

$Y := z_1 + z_2$



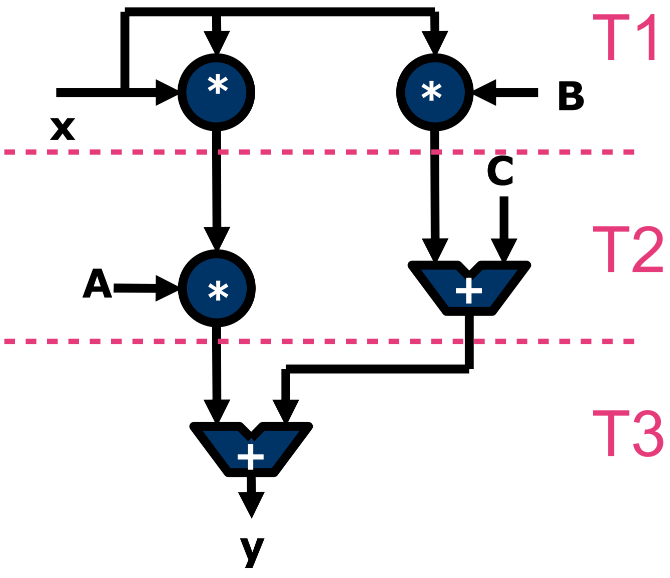
Temporal (Software on a CPU)

Spatial (Hardware in Digital IC)

Fundamentals – Register-Transfer-Level

Properties of a design:

- operations and data transfer are well defined
- clock and reset network defined (synchronous design)
- timing information as of assignments of operations to clock edges

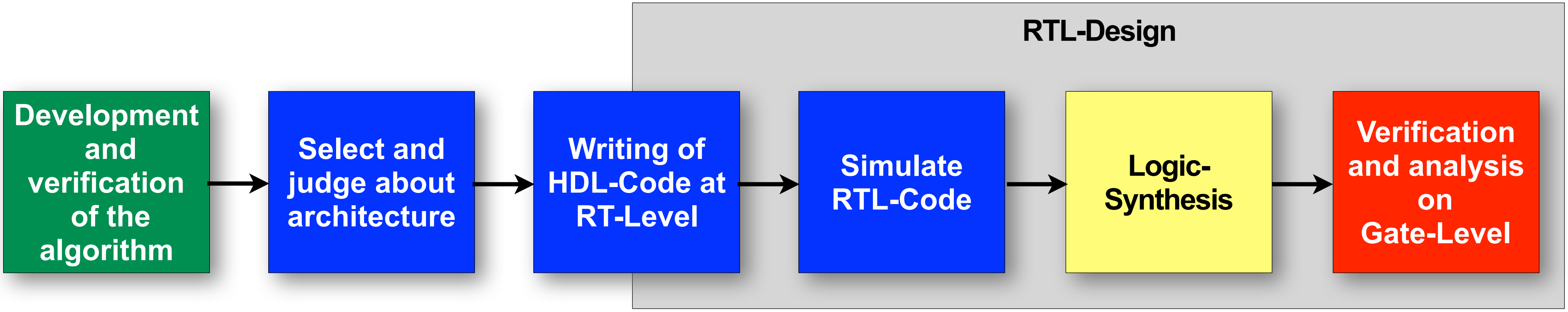


Defined by:

- Blocks (Register, Combinatoric, Multiplier) and signal assignments
- Finite State Machines (FSM)

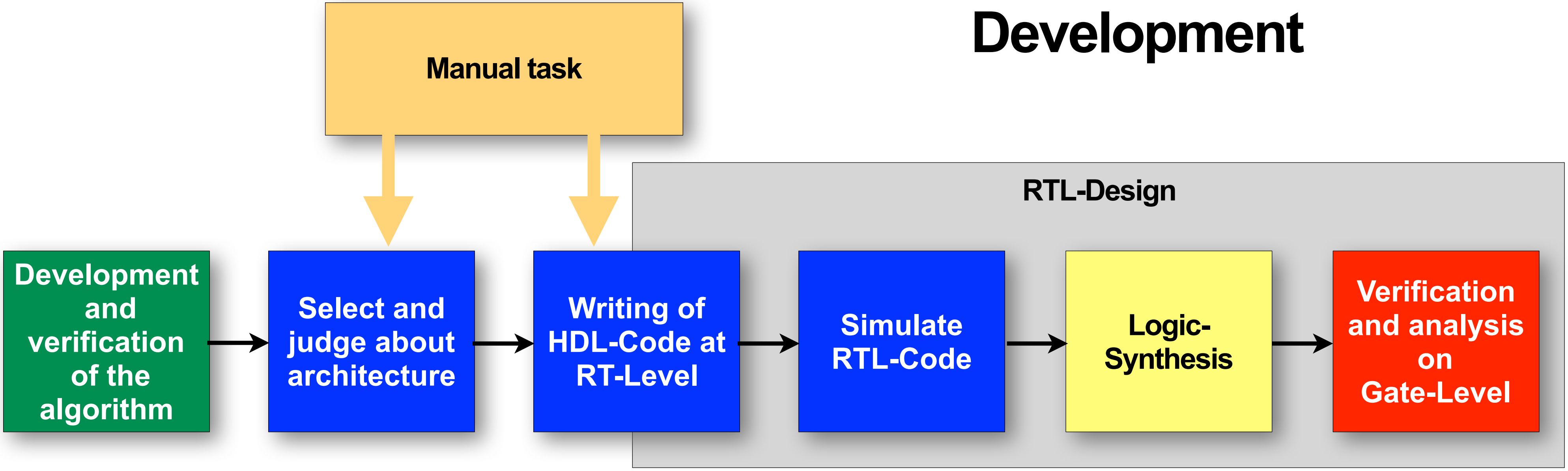
Digital Design Flow - without HLS

Traditional RTL-Development


RTL-Design
Development and verification of the algorithm
Select and judge about architecture
Writing of HDL-Code at RT-Level
Simulate RTL-Code
Logic-Synthesis
Verification and analysis on Gate-Level

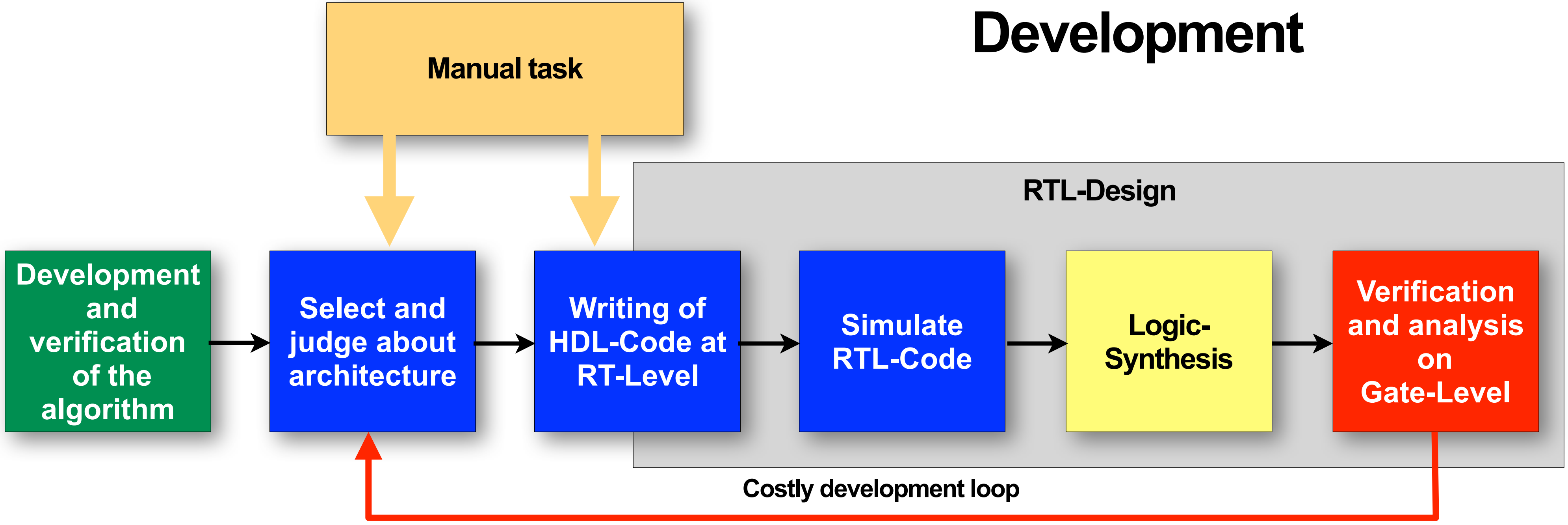
Digital Design Flow - <u>without</u> HLS

Traditional RTL-Development

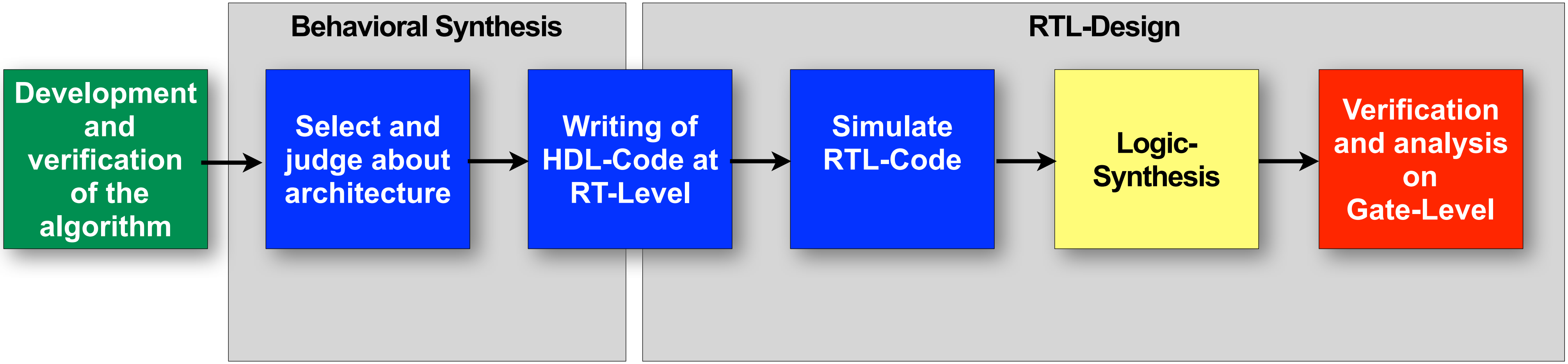

Manual task
RTL-Design
Development and verification of the algorithm
Select and judge about architecture
Writing of HDL-Code at RT-Level
Simulate RTL-Code
Logic-Synthesis
Verification and analysis on Gate-Level

Digital Design Flow - without HLS

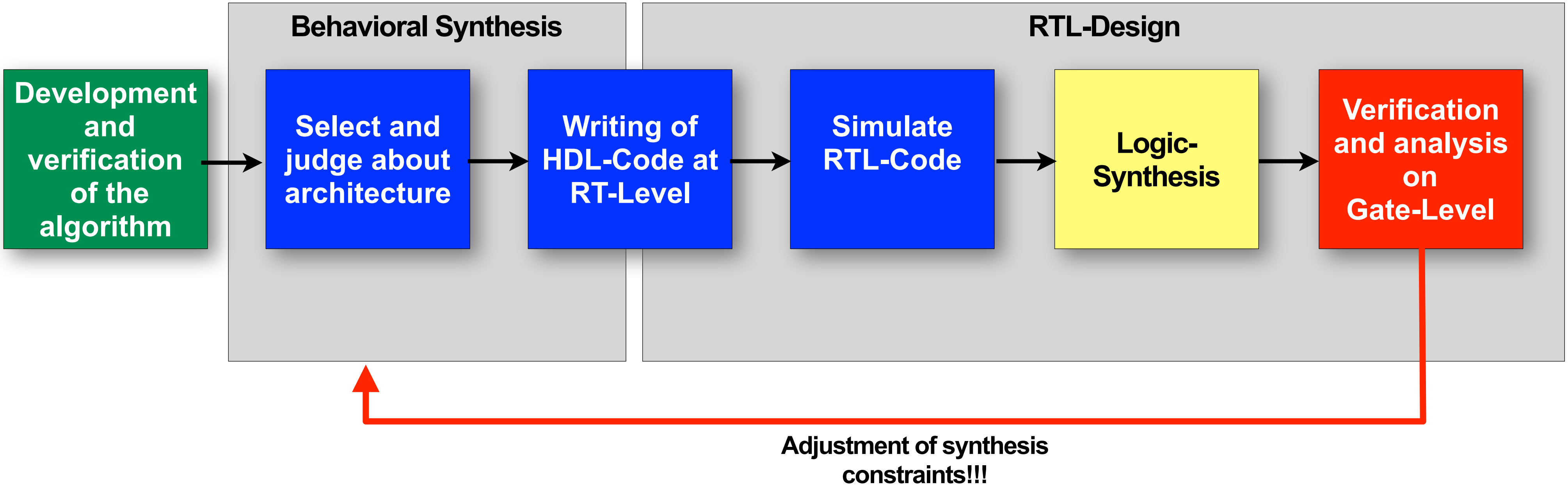
Traditional RTL-Development


Manual task
RTL-Design
Development and verification of the algorithm
Select and judge about architecture
Writing of HDL-Code at RT-Level
Simulate RTL-Code
Logic-Synthesis
Verification and analysis on Gate-Level
Costly development loop

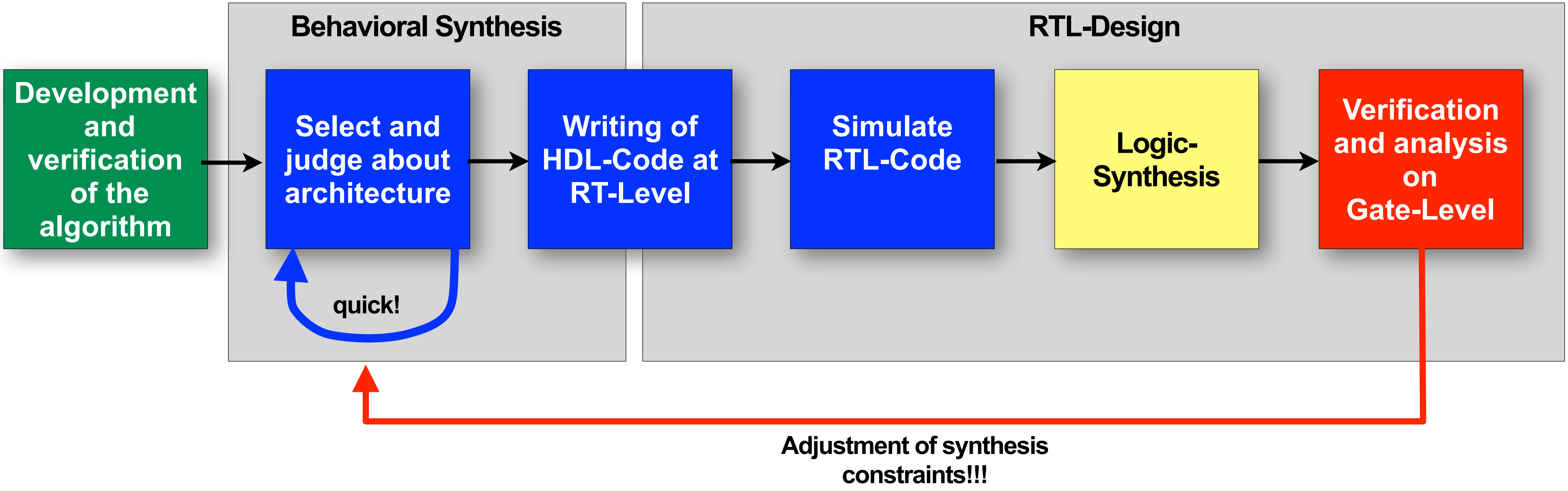
Digital Design Flow - with HLS


Behavioral Synthesis
RTL-Design
Development and verification of the algorithm
Select and judge about architecture
Writing of HDL-Code at RT-Level
Simulate RTL-Code
Logic-Synthesis
Verification and analysis on Gate-Level

Digital Design Flow - with HLS


Behavioral Synthesis
RTL-Design
Development and verification of the algorithm
Select and judge about architecture
Writing of HDL-Code at RT-Level
Simulate RTL-Code
Logic-Synthesis
Verification and analysis on Gate-Level
Adjustment of synthesis constraints!!!

Digital Design Flow - <u>with</u> HLS


Behavioral Synthesis
RTL-Design
Development and verification of the algorithm
Select and judge about architecture
Writing of HDL-Code at RT-Level
Simulate RTL-Code
Logic-Synthesis
Verification and analysis on Gate-Level
quick!
Adjustment of synthesis constraints!!!

Designing on Algorithmic Level

Advantages:

- small comprehensive descriptions (simple to describe and easy to understand)
- very fast simulations
- architecture independent
- early estimations on power, performance and complexity
- quick design space exploration
- faster Time to Market
- design reuse (in terms different architectures and technologies)

Limitations:

- not applicable for asynchronous architectures
- limited control/influence over generated RT Code and architecture
- Optimizations only within design units (functions, procedures...)
- "hand coded RT-Level" leads to much faster and smaller designs...

Synthesis - opportunities on Abstraction Levels

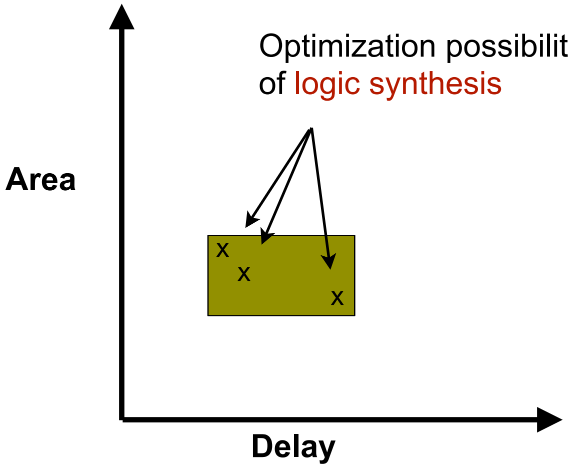
Creating a new digital design means:

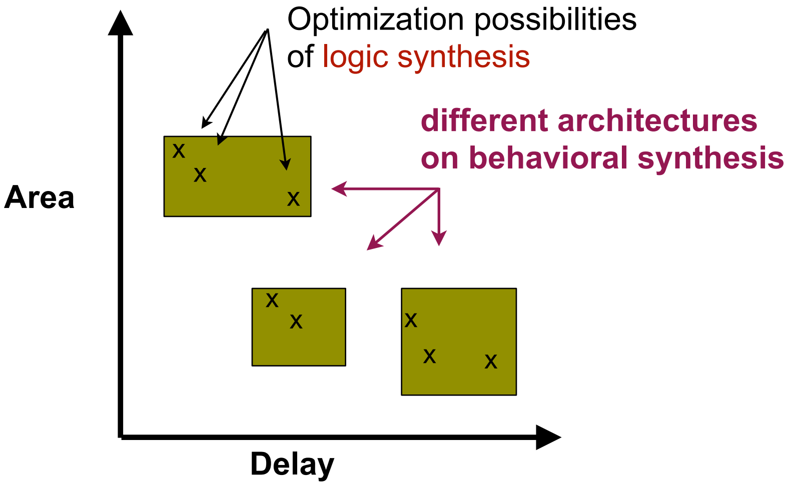
- many architectural choices/solutions (size of thousand lines HDL-Code each)
- each solution has **Pros** and **Cons** regarding area and latency

Problem: Which architecture is the best solution?

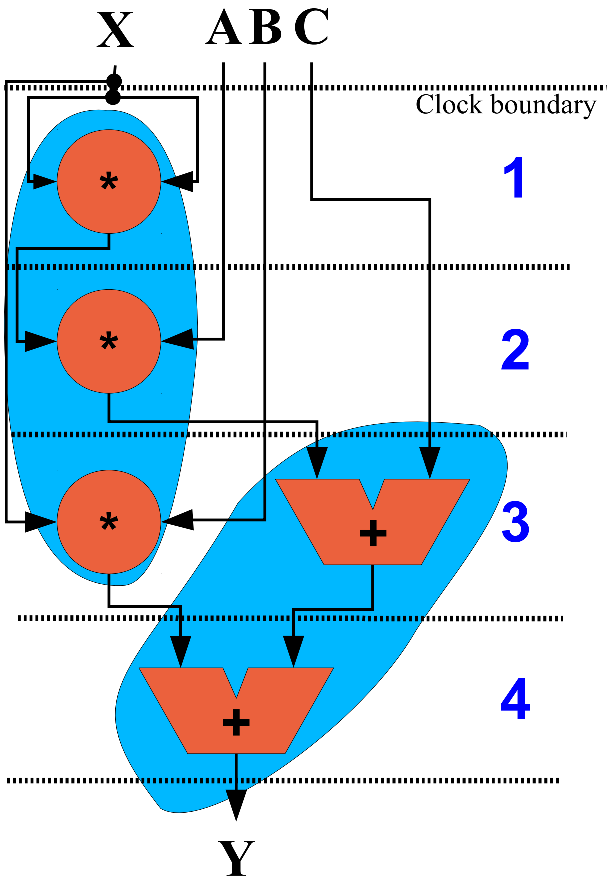
- Even senior design engineers cannot oversee the design space and answer this question!

Conclusion: Let a tool decide! Just describe behavior and let the HLS-Tool find the optimum.

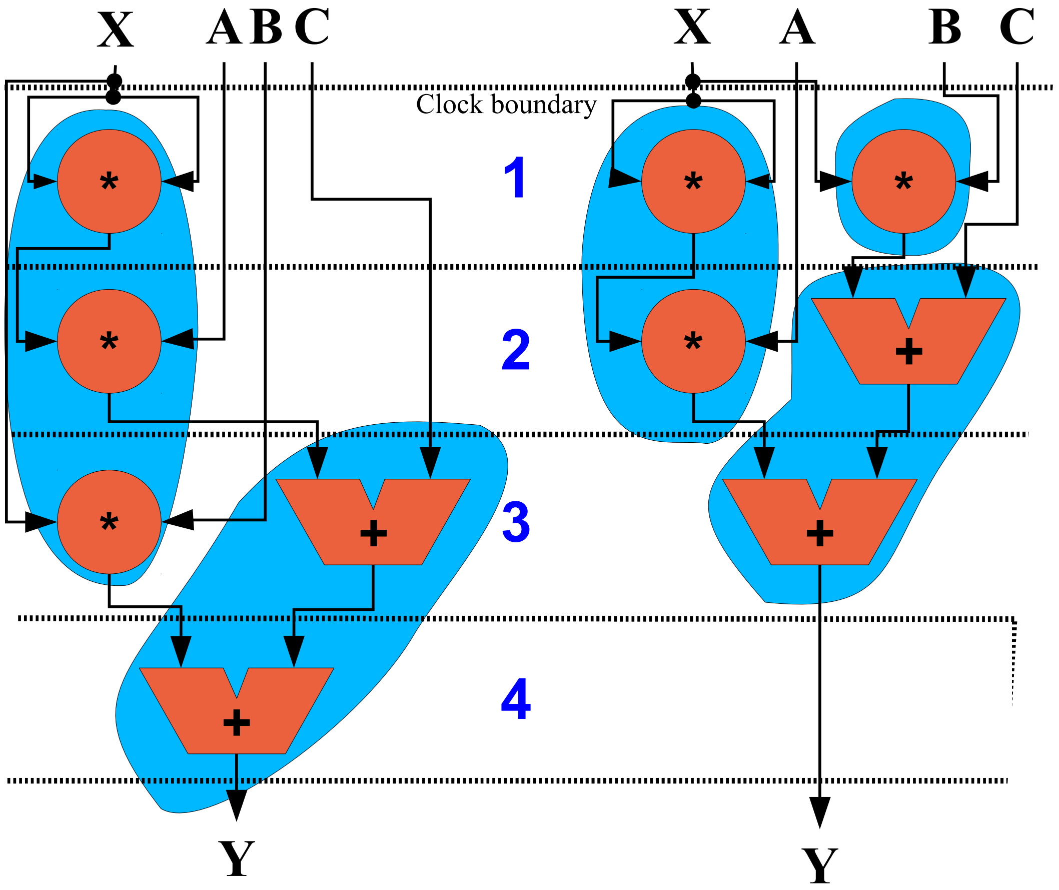




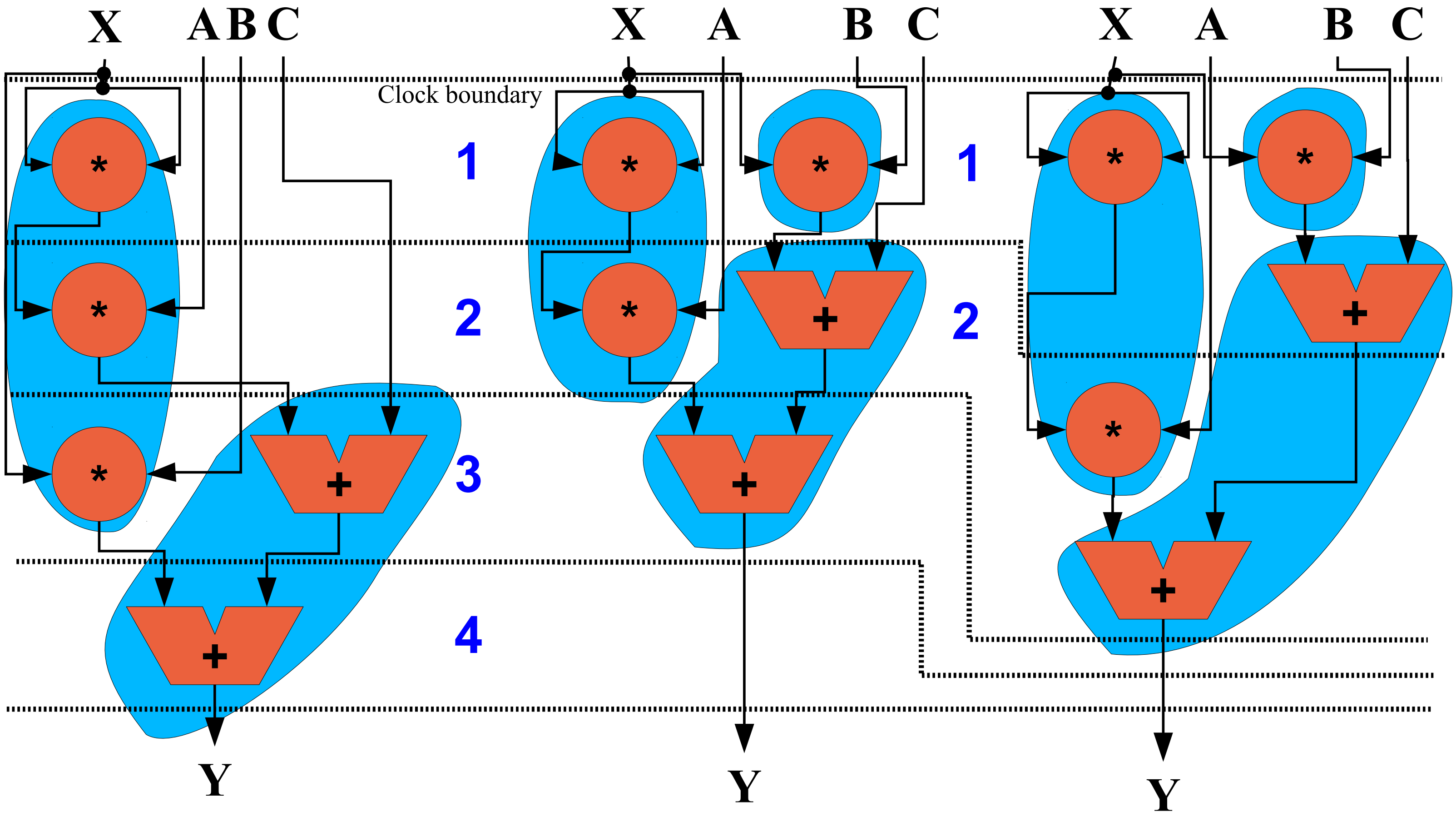
Design Space Example for Ax²+Bx+C (1)



Design Space Example for Ax^2+Bx+C (1)



Design Space Example for Ax^2+Bx+C (1)



Design Space Example for Ax^2+Bx+C (2)

V1:

- 1 Adder, 1 Multiplier
- Latency 4
- Clock period “short”

V2:

- 1 Adder, 2 Multiplier
- Latency 3
- Clock period “short”

V3:

- 1 Adder, 2 Multiplier
- Latency 2
- Clock period “long”

3 dimensional space

Design Space Example for Ax²+Bx+C (2)

V1:

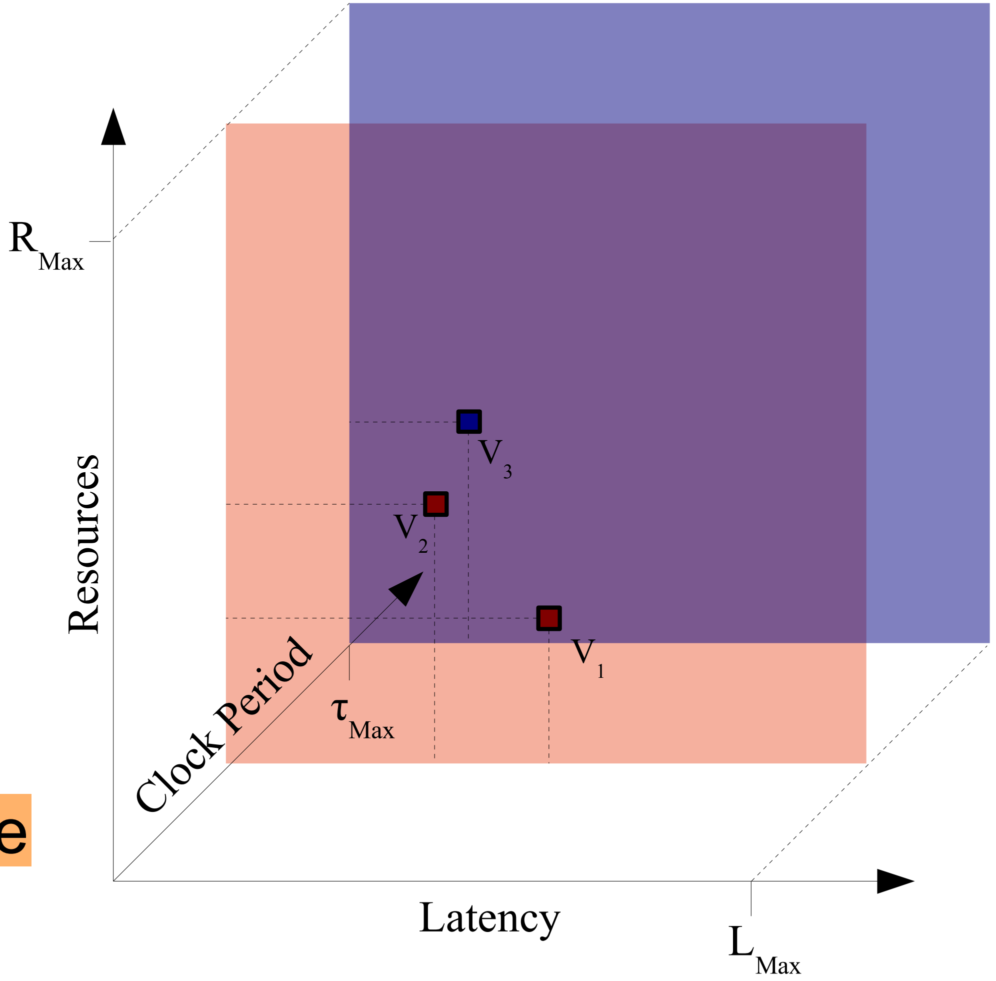
- 1 Adder, 1 Multiplier
- Latency 4
- Clock period “short”

V2:

- 1 Adder, 2 Multiplier
- Latency 3
- Clock period “short”

V3:

- 1 Adder, 2 Multiplier
- Latency 2
- Clock period “long”



3 dimensional space

Design Space Example for Ax^2+Bx+C (2)

V1:

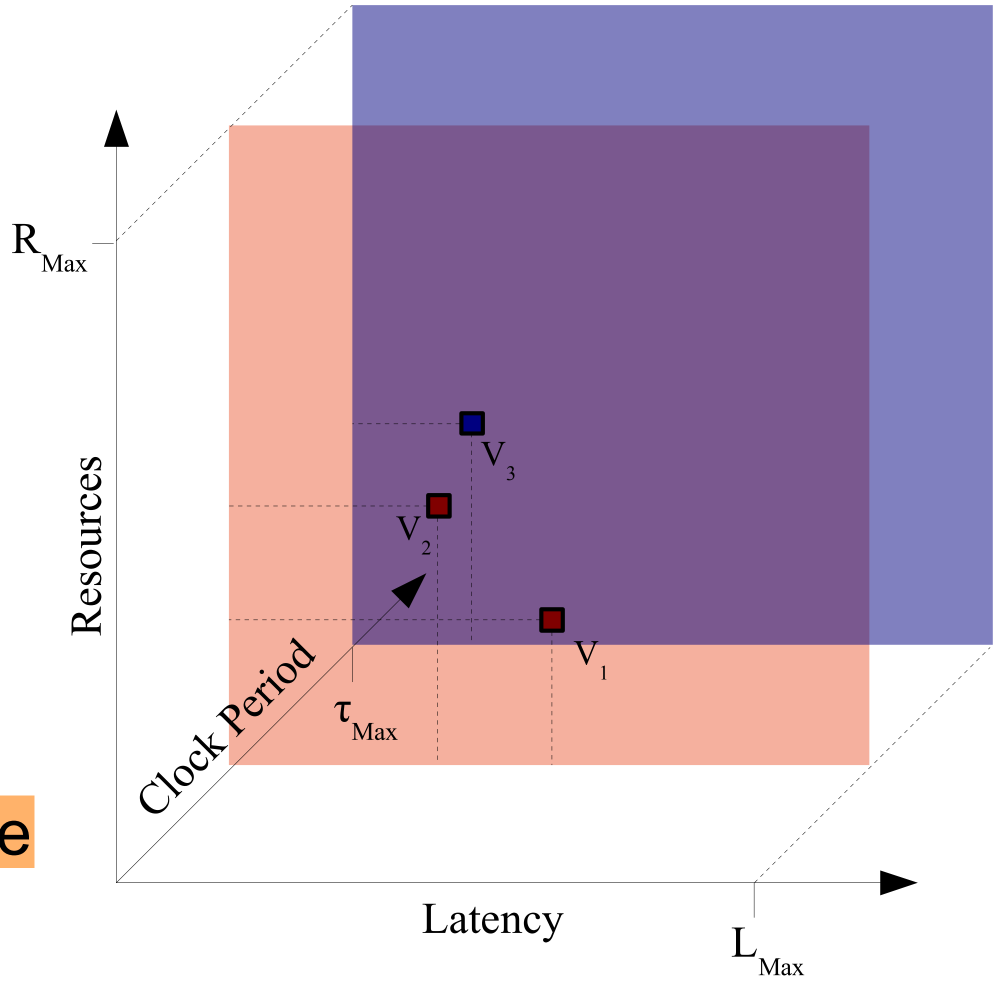
- 1 Adder, 1 Multiplier
- Latency 4
- Clock period “short”

V2:

- 1 Adder, 2 Multiplier
- Latency 3
- Clock period “short”

V3:

- 1 Adder, 2 Multiplier
- Latency 2
- Clock period “long”



3 dimensional space

Best solution?

Design Space Example for Ax^2+Bx+C (2)

V1:

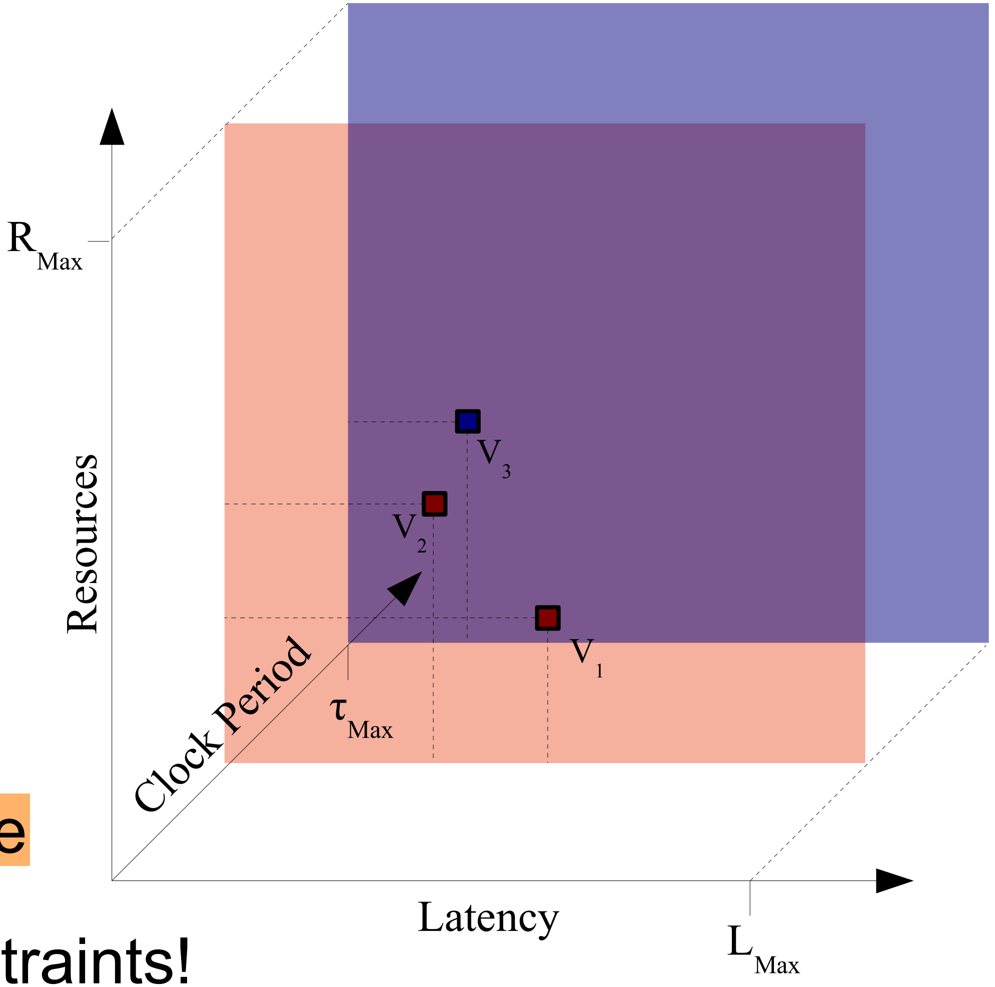
- 1 Adder, 1 Multiplier
- Latency 4
- Clock period “short”

V2:

- 1 Adder, 2 Multiplier
- Latency 3
- Clock period “short”

V3:

- 1 Adder, 2 Multiplier
- Latency 2
- Clock period “long”



3 dimensional space

Best solution? Depends on constraints!

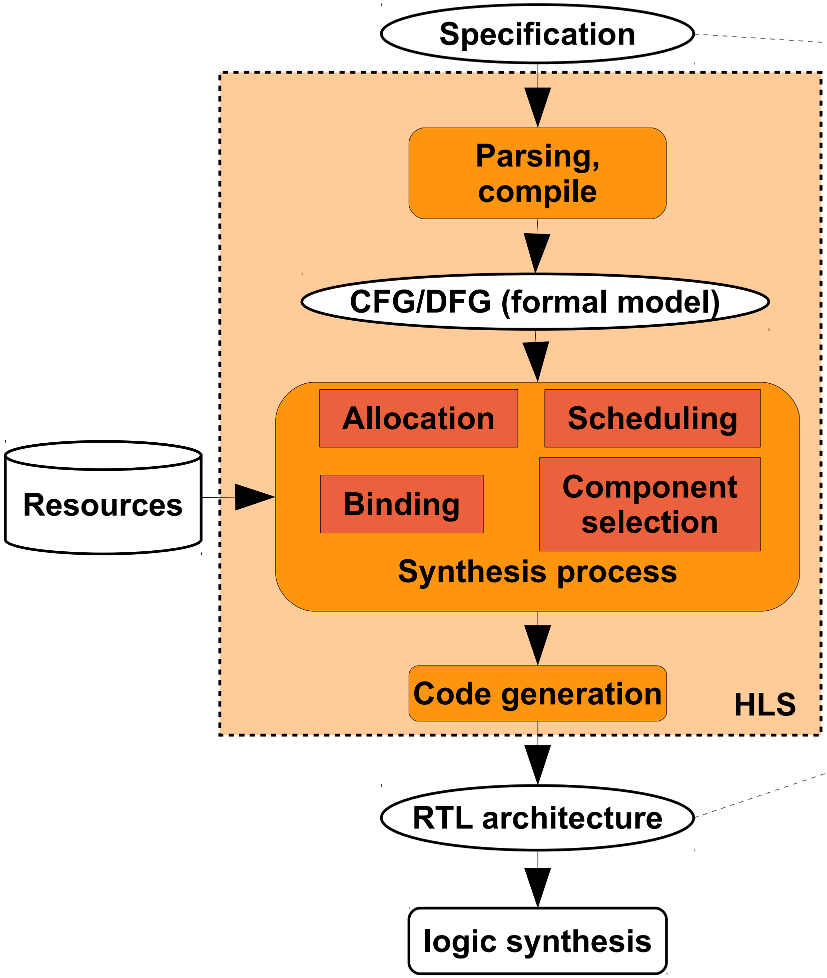
Fundamentals – Summary

High Level Synthesis:

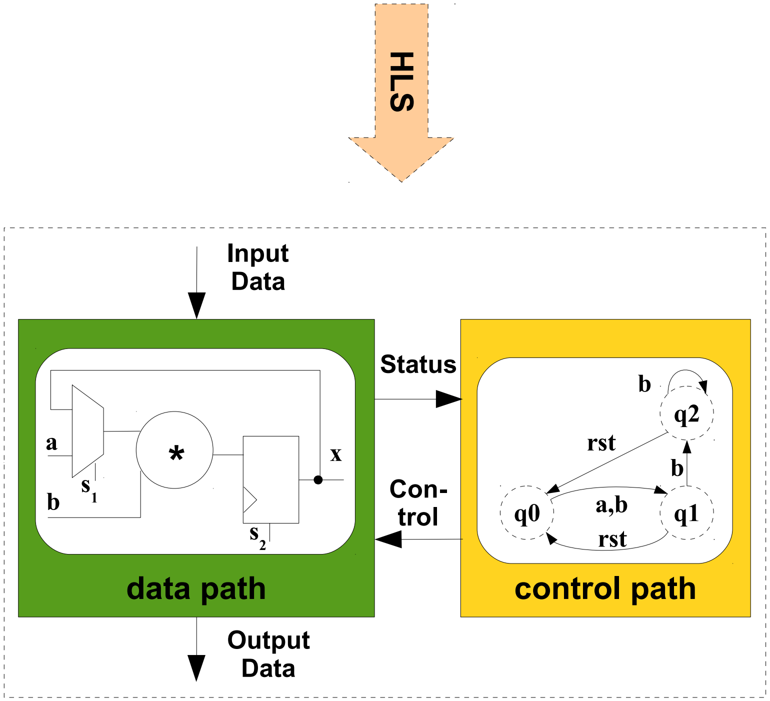
- HLS is a possible step in the design flow of digital hardware systems.
- transforms from **behavioral algorithmic level** into the **structural RT level**
- quick design space exploration
- finds optimal (constrained) solution in terms of:
  - Area, Latency, Timing/Frequency
- "hand coded RT-Level" is much faster and smaller... but takes long time
- RT-code correct by construction (generation)

HLS
-
Synthesis Process

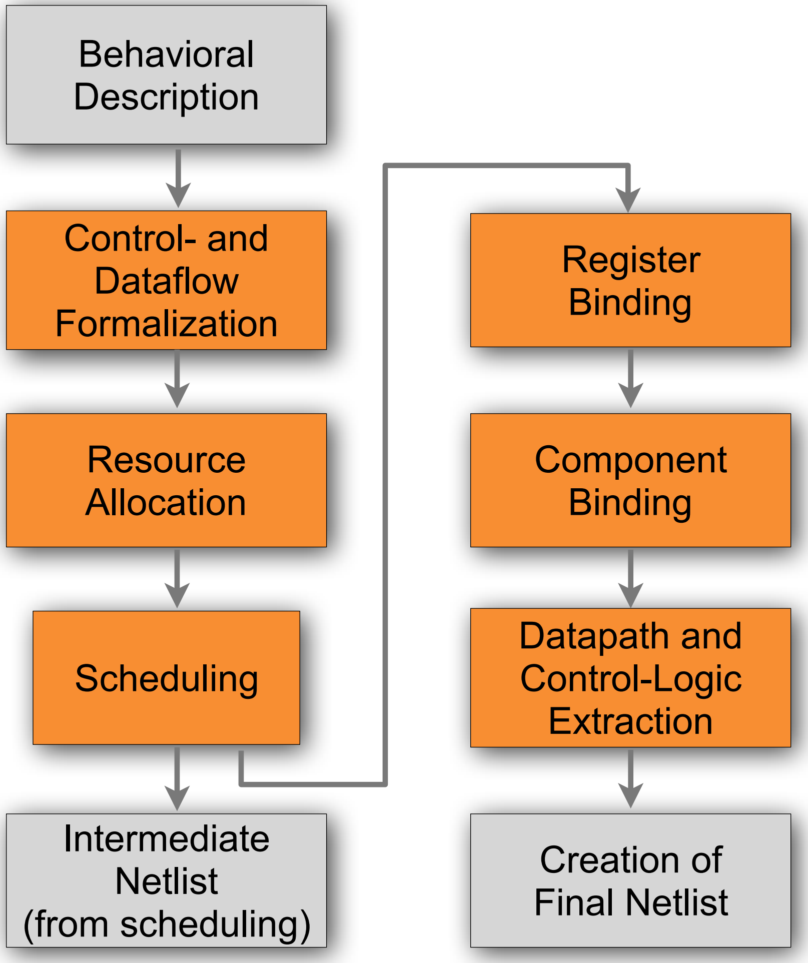
Synthesis process - Overview



```
int main(int a, int b){
          int x;
          x=a*b*b;
          return x;
}
```



Synthesis Process - Overview (simplified)


Behavioral Description
Control- and Dataflow Formalization
Resource Allocation
Scheduling
Intermediate Netlist (from scheduling)
Register Binding
Component Binding
Datapath and Control-Logic Extraction
Creation of Final Netlist

Formalization

- Lexical processing parses the high-level language source code
- Transformation into internal representation (graphs)
- Similar to compilation of conventional high-level programming language
- Identifies inputs, outputs, operations and their dependencies
- Allows optimizations commonly used in parallelizing compilers
  - common subexpression elimination
  - constant propagation ...

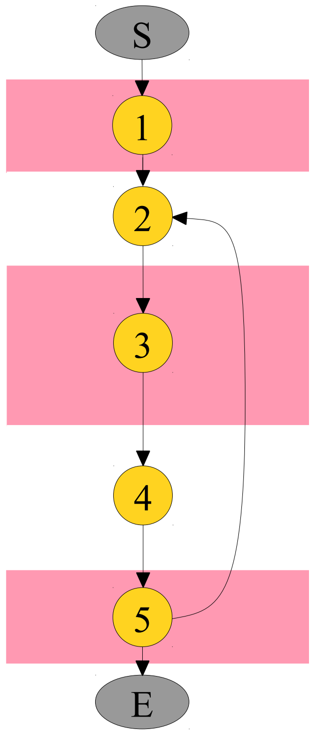
Formal Representation (1) - CFG and DFG

```
compute(){
 int x,y;
 int i;
 for (i=0; i<MAX; i++){
   x = readmem();
   y = A*x*x + B*x + C;
   writemem(y);
 }
}
```

Formal Representation (1) - CFG and DFG

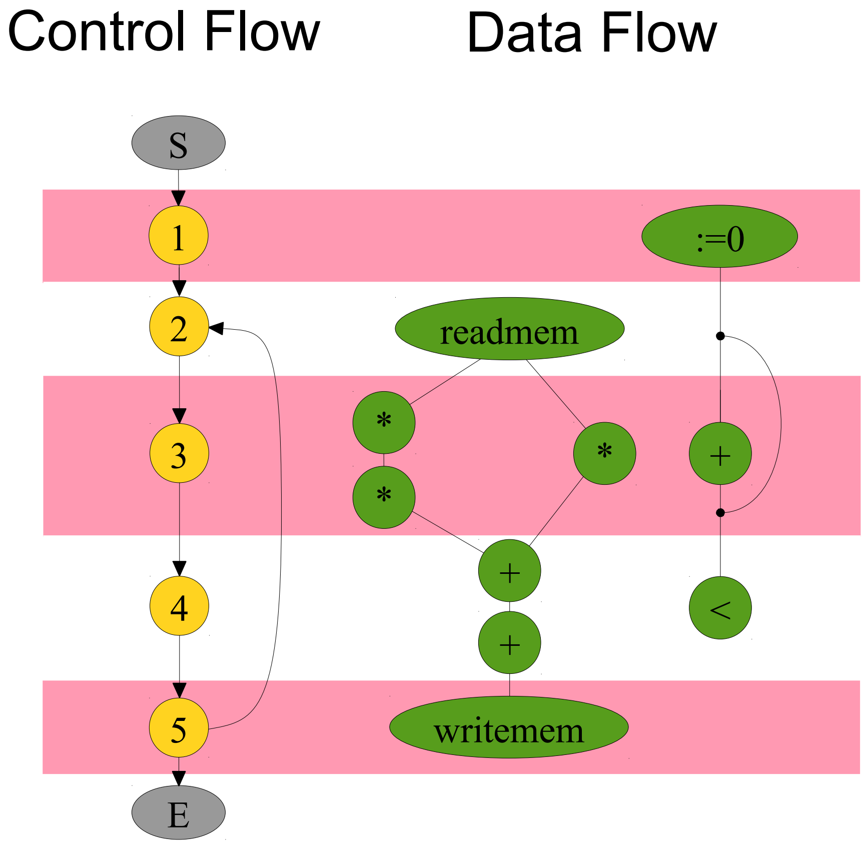
```
compute(){
 int x,y;
 int i;
 for (i=0; i<MAX; i++){
   x = readmem();
   y = A*x*x + B*x + C;
   writemem(y);
 }
}
```

Control Flow



Formal Representation (1) - CFG and DFG

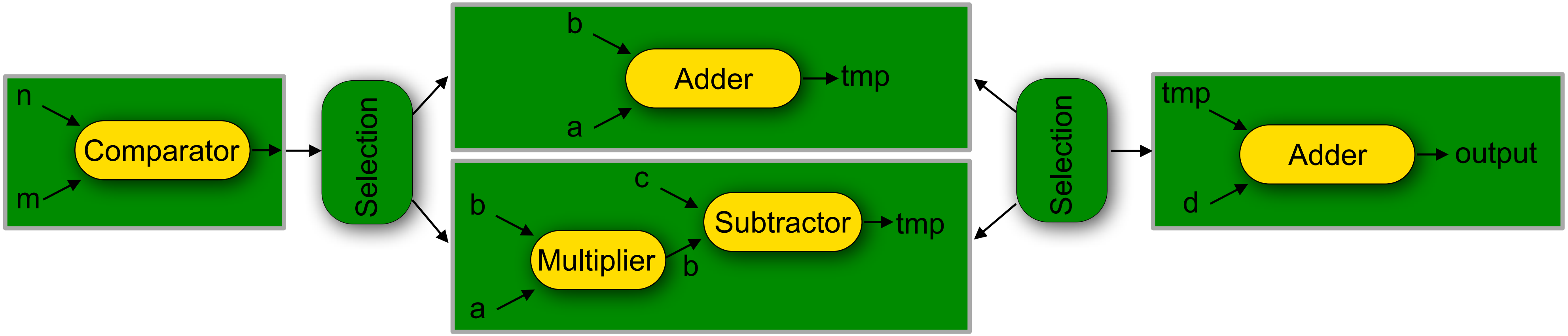
```
compute(){
 int x,y;
 int i;
 for (i=0; i<MAX; i++){
   x = readmem();
   y = A*x*x + B*x + C;
   writemem(y);
 }
}
```



Formal Representation (2) - a CDFG

C/C++ Code

```
if (n > m)
    tmp = a + b;
else
    tmp = a * b - c;
tmp = tmp + e;
```



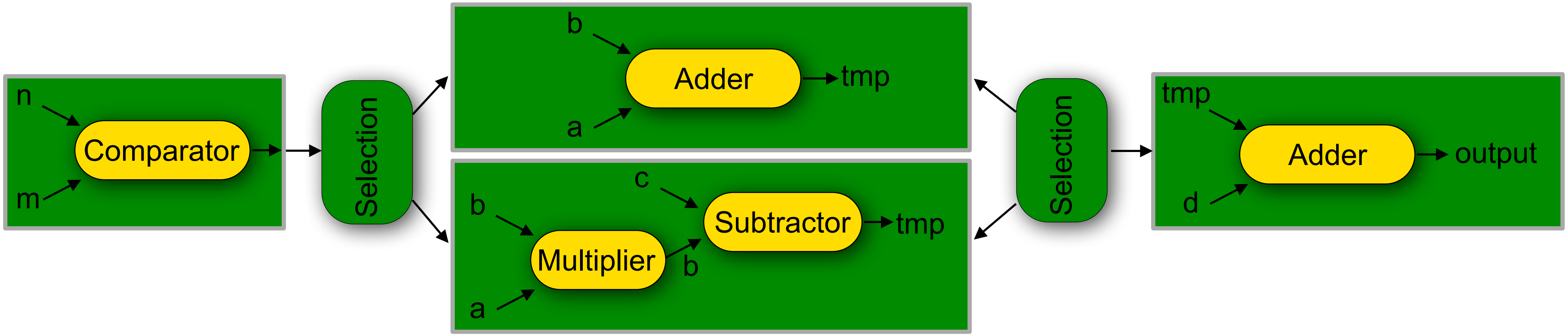
Formal Representation (2) - a CDFG

C/C++ Code

```
if (n > m)
    tmp = a + b;
else
    tmp = a * b - c;
tmp = tmp + e;
```

VHDL (behavioral)

```
IF ( n > m ) THEN
    tmp := a + b;
ELSE
    tmp := a * b - c;
END IF;
tmp := tmp + e;
```

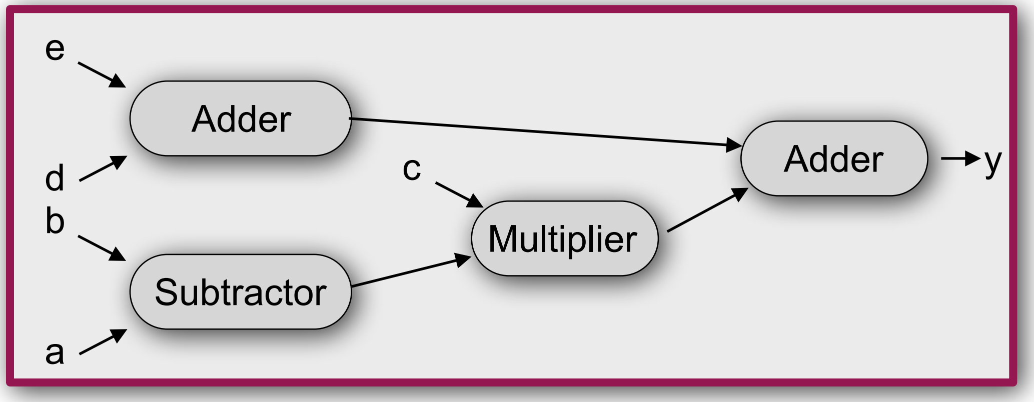


Resource Allocation

- Resource allocation establishes a set of functional units that will be adequate to implement the design
- An initial resource allocation is performed and subsequently modified during scheduling and/or binding
- Determination of type and number of resources required
    - Functional units
    - Storage elements
    - Busses

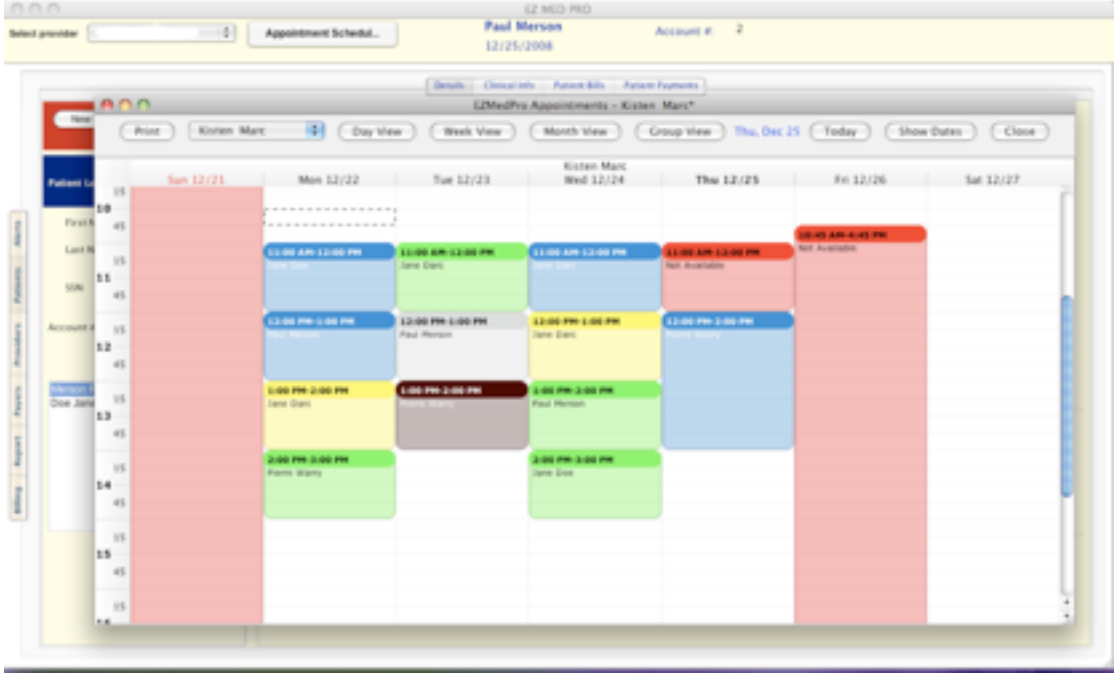
Resource Allocation - Functional units

$$y = (\overbrace{\overbrace{(a-b)}^{\text{8 Bit}} \overbrace{* \; c)}^{\text{8 Bit}}}^{\text{16 Bit}} + (\overbrace{d + e}^{\text{8 Bit}})$$



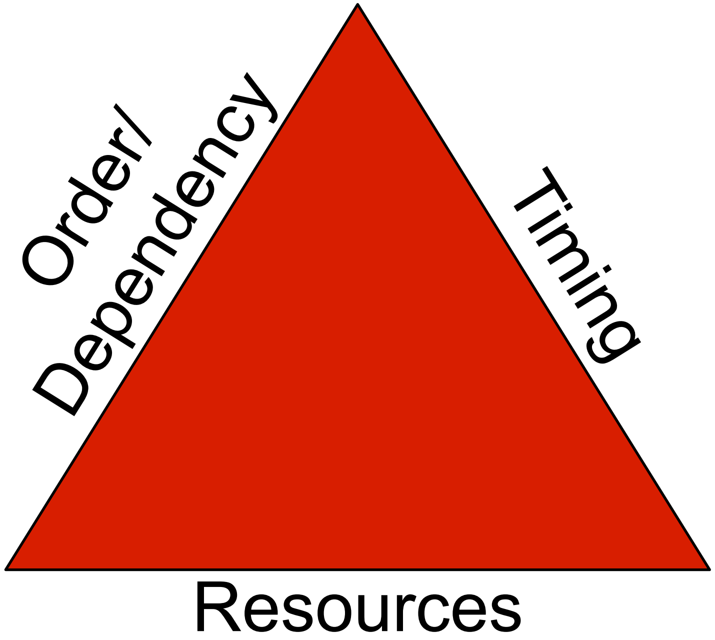
Operation	Component
Subtraction	Ripple-Subtractor Carry-Save-Subtractor Subtraction/Addition Unit
Addition	Subtraction/Addition Unit Curry-Save-Adder 16-Bit-Adder 32-Bit-Adder ...-Bit-Adder Ripple-Adder
Multiplication	Multiplication Unit

Scheduling - Fundamentals



Boss maps tasks to employees. A schedule for everyone results.

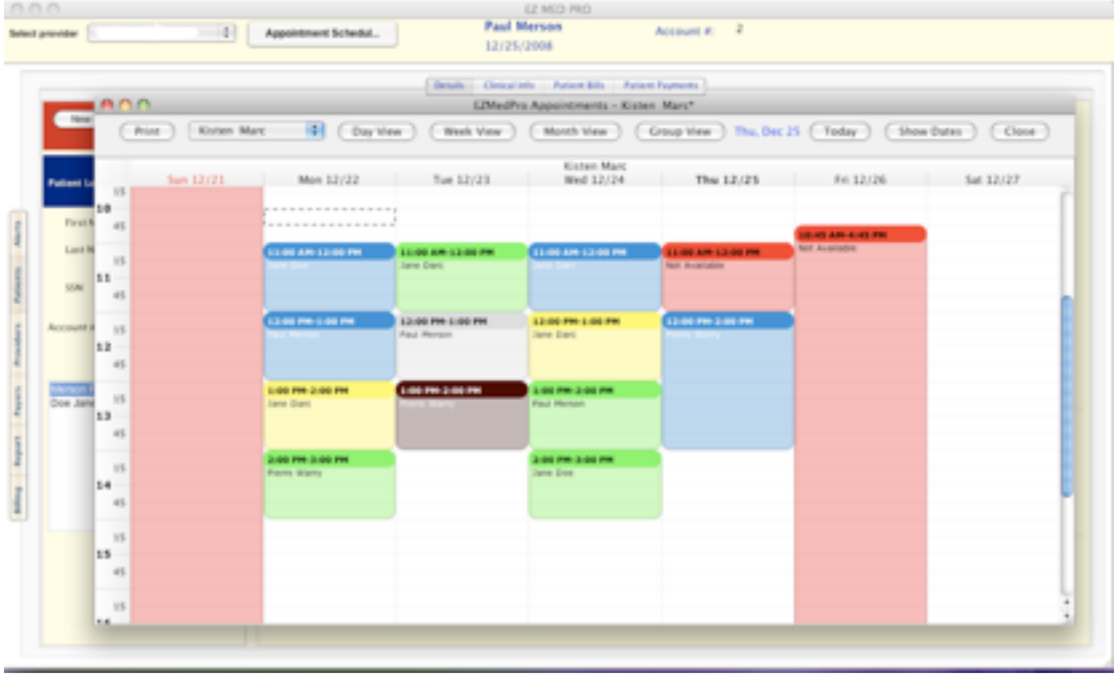
Scheduling - Fundamentals



Order/ Dependency

Timing

Resources



Temporal and Spatial mapping of operations to resources.

Boss maps tasks to employees. A schedule for everyone results.

Scheduling - ASAP

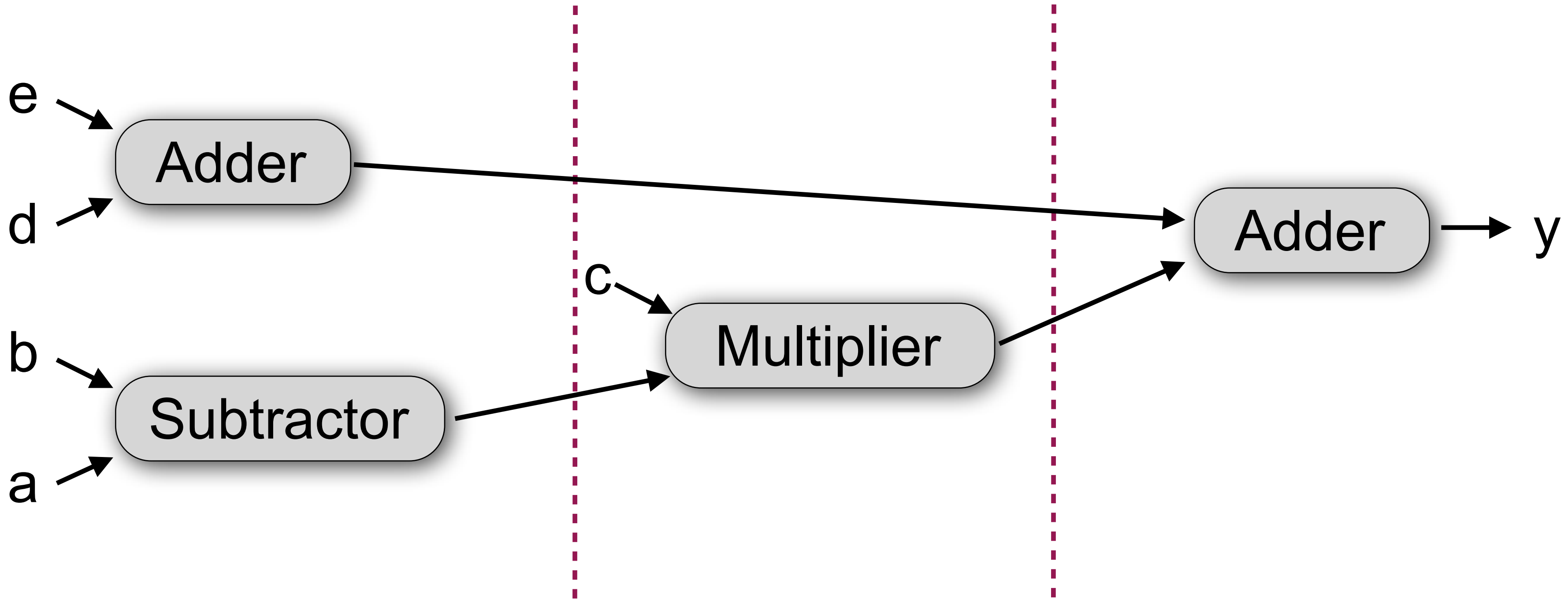
- The simplest type of scheduling occurs when we wish to optimize the overall latency of the computation and do not care about the number of resources required

- This can be achieved by simply starting each operation in a CDFG as soon as its predecessors have completed

- This strategy gives rise to the name ASAP for “As Soon As Possible”

Scheduling - ASAP example

```
y = ( (a - b) * c) + ( d + e )
```

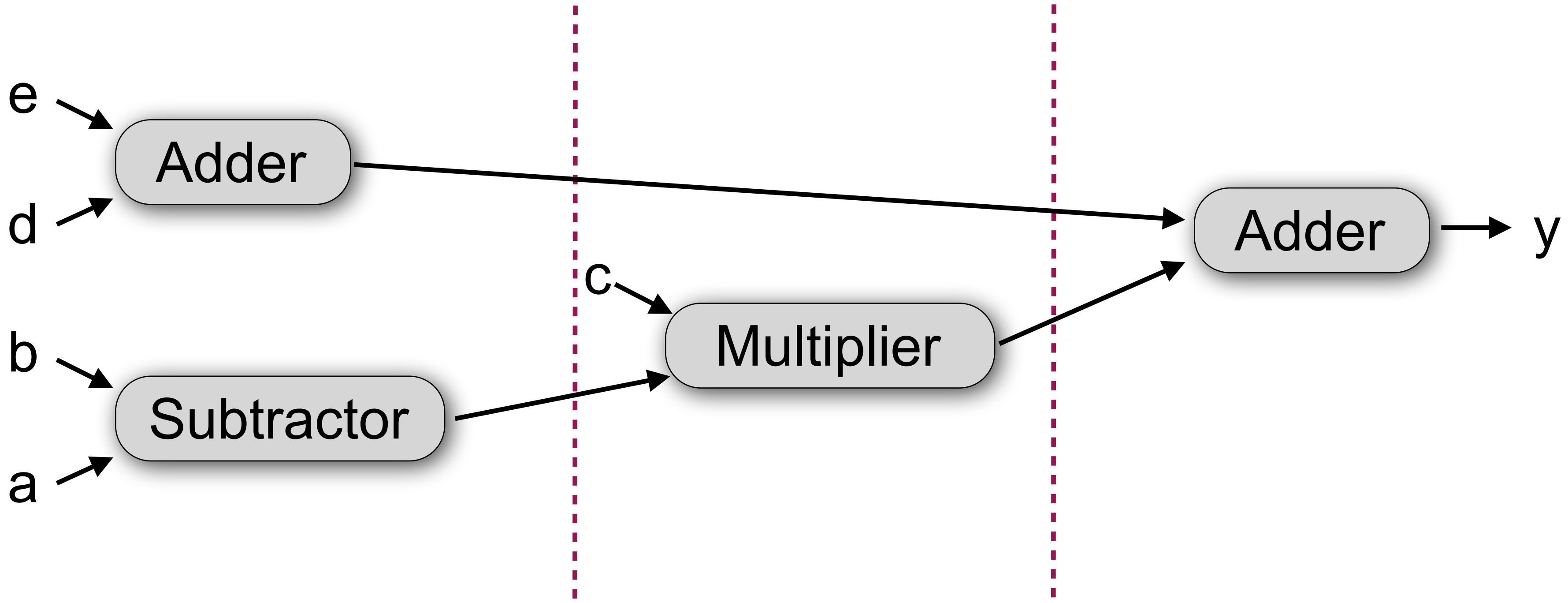


Scheduling - ALAP

- The ASAP algorithm schedules each operation at the earliest opportunity. Given an overall latency constraint, it is equally possible to schedule operations at the latest opportunity.
- This leads to the concept of As-Late-As-Possible (ALAP) scheduling.
- ALAP scheduling can be performed by seeking the longest path between each operation and the end or “sink” node.

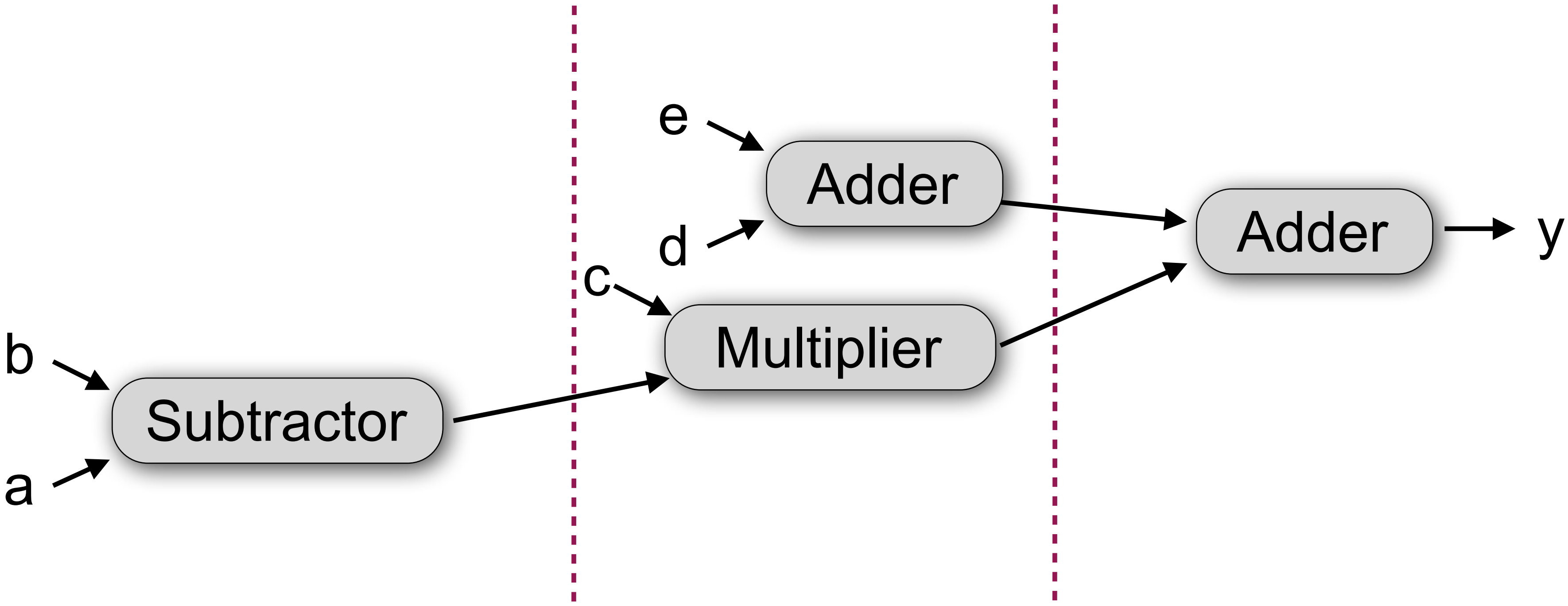
Scheduling example - ASAP versus ALAP

```
y = ( (a - b) * c) + ( d + e )
```



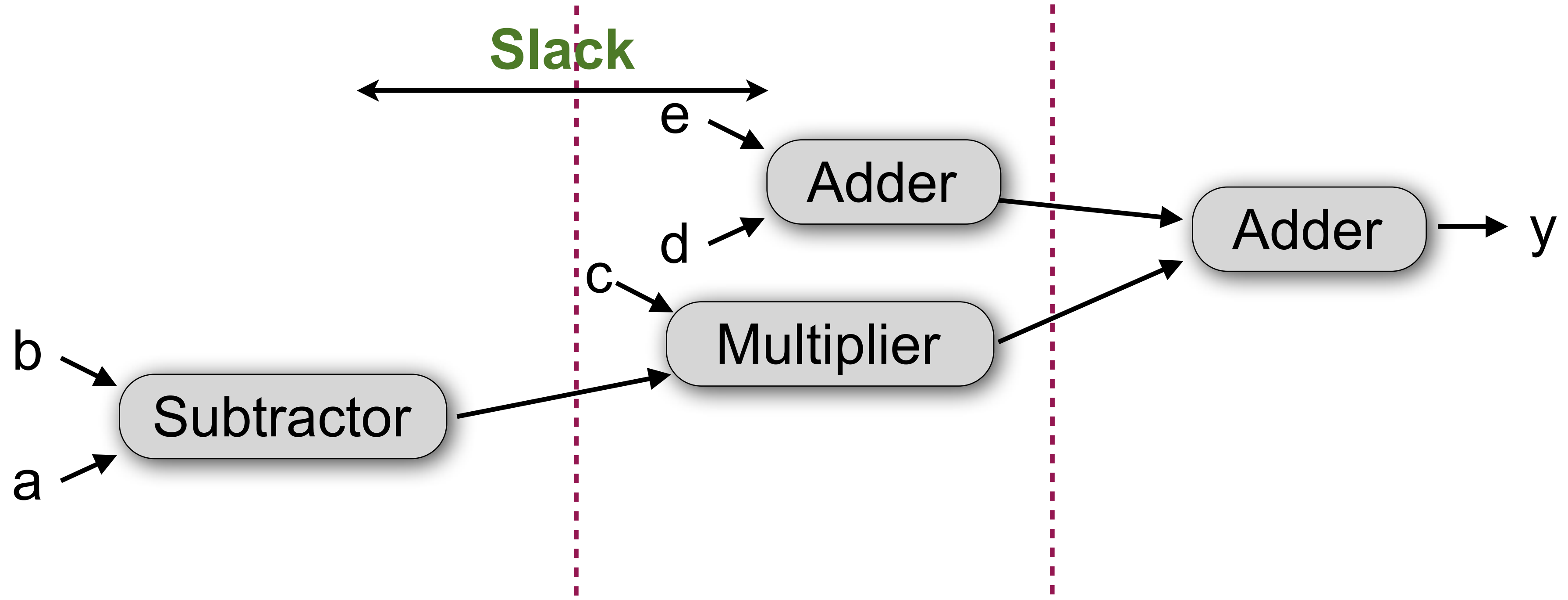
Scheduling example - ASAP versus ALAP

```
y = ( (a - b) * c) + ( d + e )
```



Scheduling example - ASAP versus ALAP

```
y = ( (a - b) * c) + ( d + e )
```



Scheduling - List Scheduling (simple)

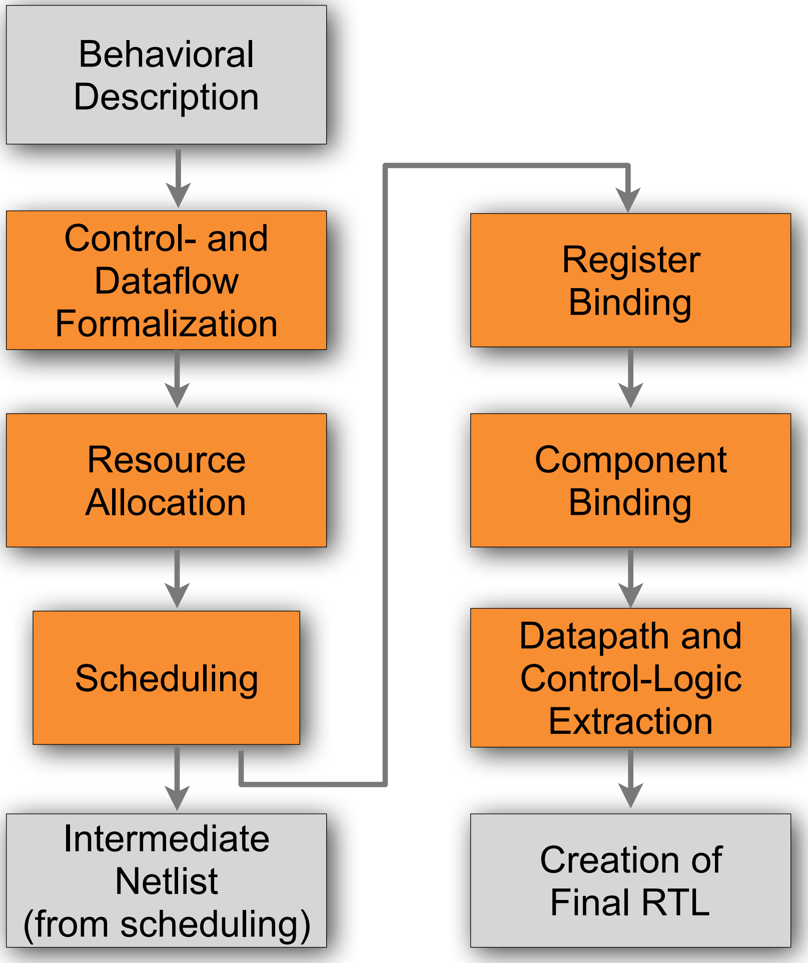
- For each control step, the operations that are available to be scheduled (slack) are kept in a list
- The list is ordered by some priority function:
  1. The length of path from the operation to the end of the block;
  2. Mobility: the number of control steps from the earliest to the latest feasible control step.
- Each operation on the list is scheduled one by one if the resources it needs are free; otherwise it is deferred to the next control step.

Scheduling - Strategy comparison

	ASAP	ALAP	List scheduling (forced directed scheduling/FDS)
Resources	as many as necessary	as many as necessary	constraining possible
Timing	no constraining possible	overall latency constrain	constraining possible
Complexity	simple	simple	medium (NP hard but heuristics are known)

There are many more complex scheduling algorithms that consider the global context

Synthesis Process - Overview (simplified)

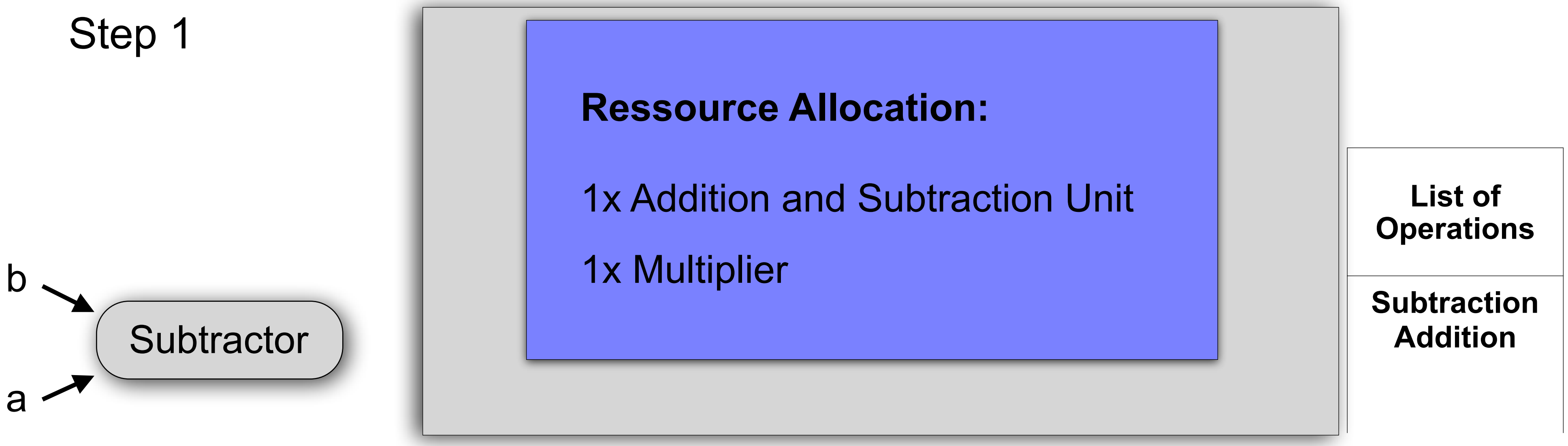

Behavioral Description
Control- and Dataflow Formalization
Resource Allocation
Scheduling
Intermediate Netlist (from scheduling)
Register Binding
Component Binding
Datapath and Control-Logic Extraction
Creation of Final RTL

Planung – Listen (1)

für jeden Zyklus wird eine Liste erstellt

- enthält möglichen Operationen
- ist nach Bedingungen geordnet

Beispiel: Bedingung ist Mobilität



Step 1

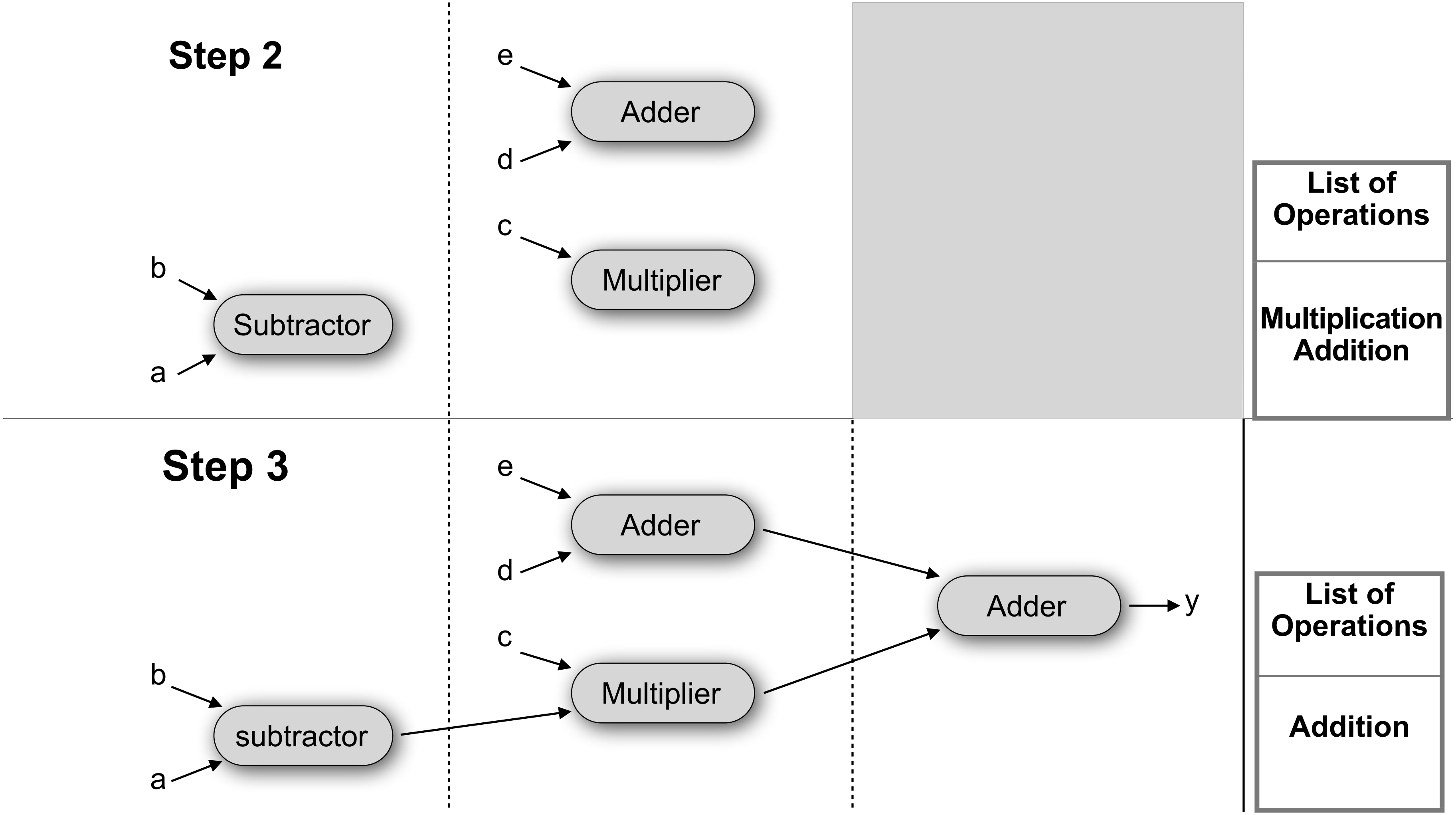
Ressource Allocation:

1x Addition and Subtraction Unit

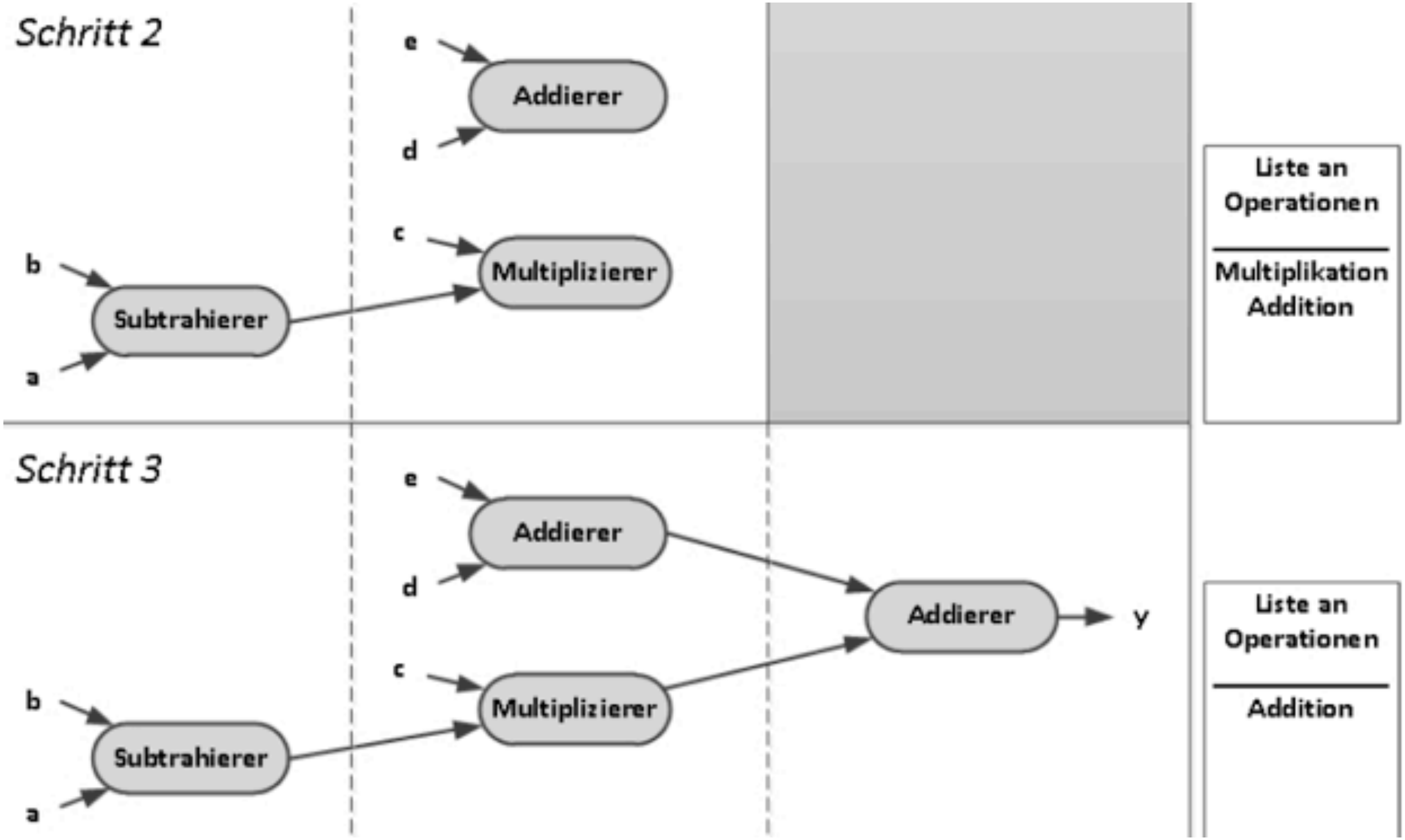
1x Multiplier

List of Operations
Subtraction **Addition**

Planung – Listen (2)



Planung – Listen (2)



Binding - Registers and Components

- Registers store values between computational stages
- Components do the computation

Decisions:

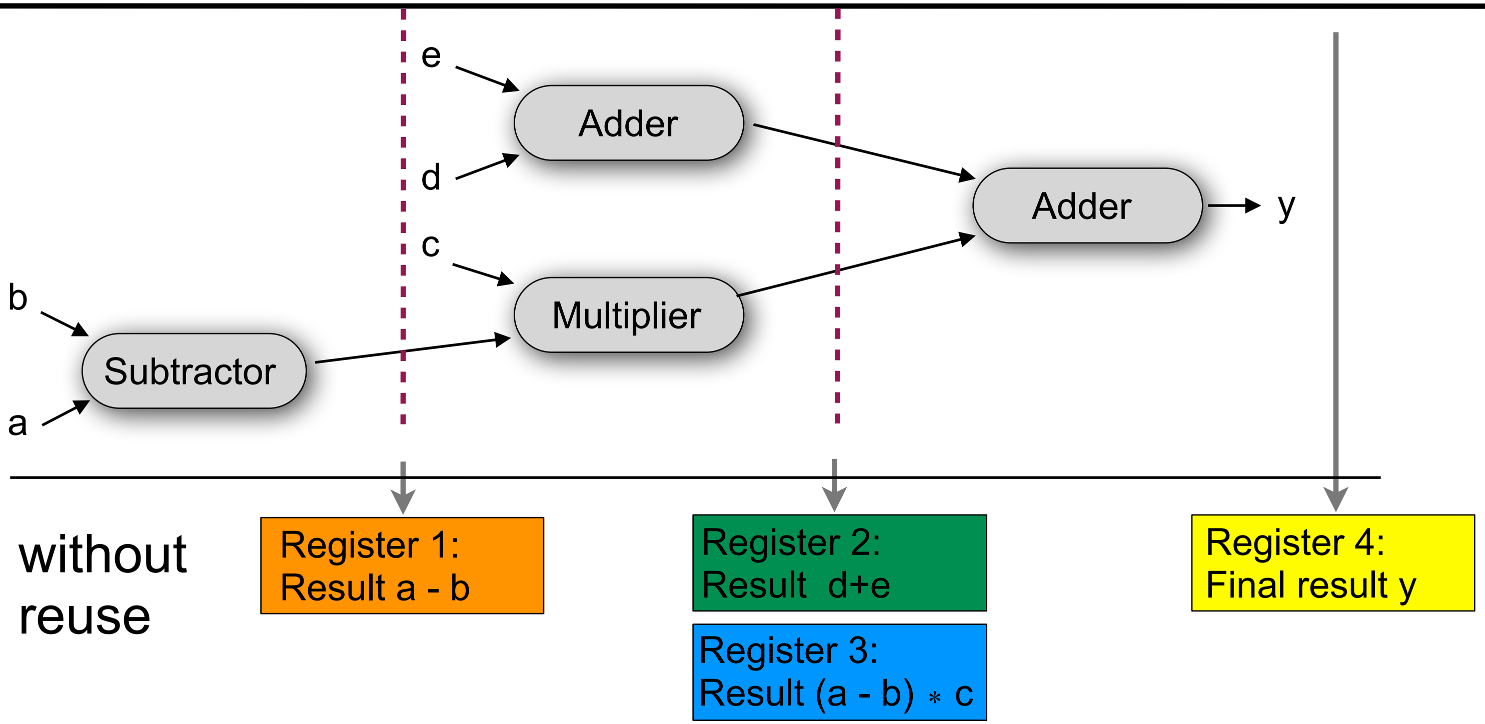
- Which operant/result is stored in which physical register instance
- Which operations of the algorithm are mapped specific instances of functional units.

Reuse:

- Analysis of the lifetime of each data value to use the same physical register to store different values at different times
- Analysis of opportunities to use a component for different operations at different times (stages)

- Heavily influences the size of the design (!)

Register Allocation - Example



without reuse

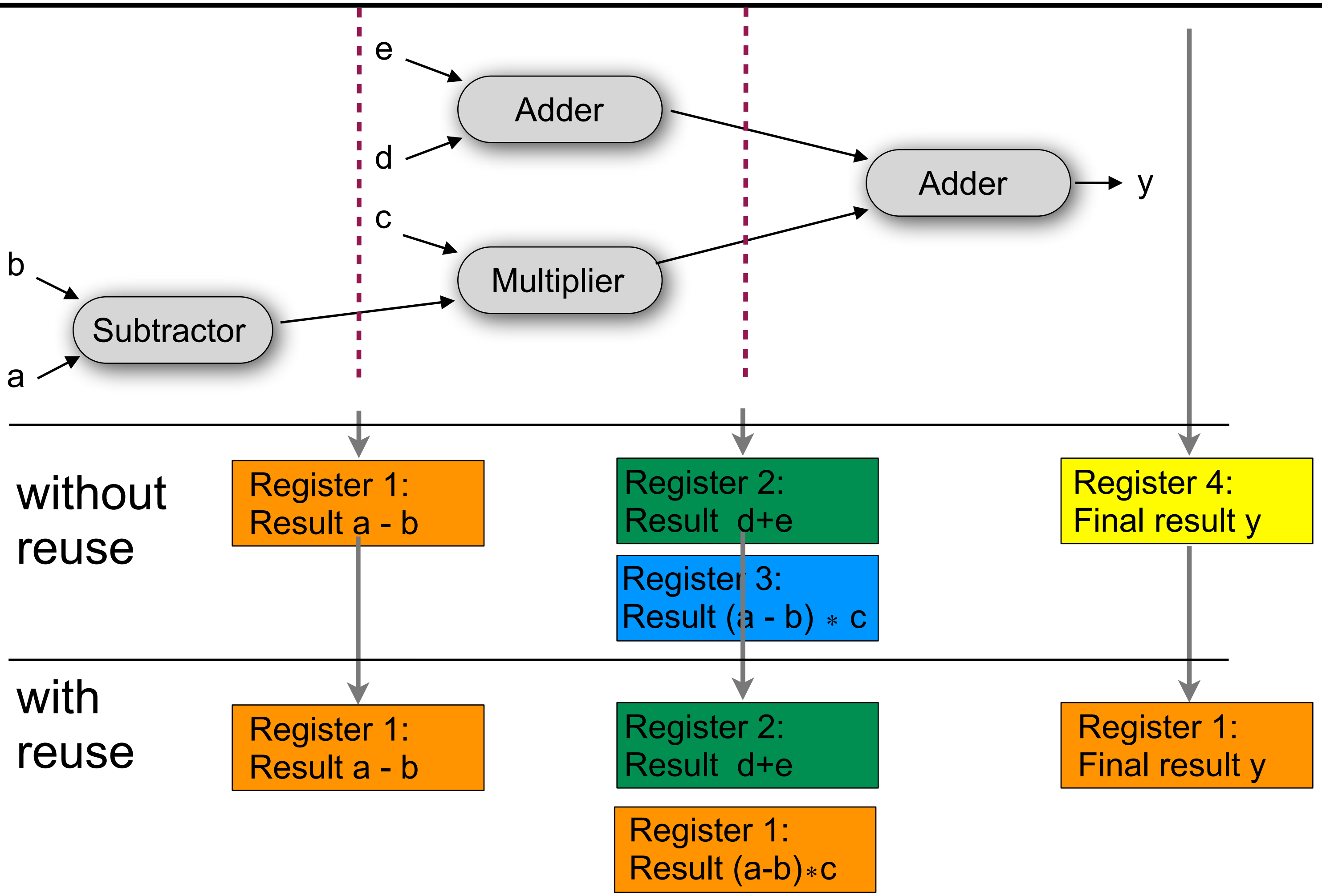
Register 1:
Result a - b

Register 2:
Result d+e

Register 3:
Result (a - b) ∗ c

Register 4:
Final result y

Register Allocation - Example



without reuse

Register 1:
Result a - b

Register 2:
Result d+e

Register 3:
Result (a - b) * c

Register 4:
Final result y

with reuse

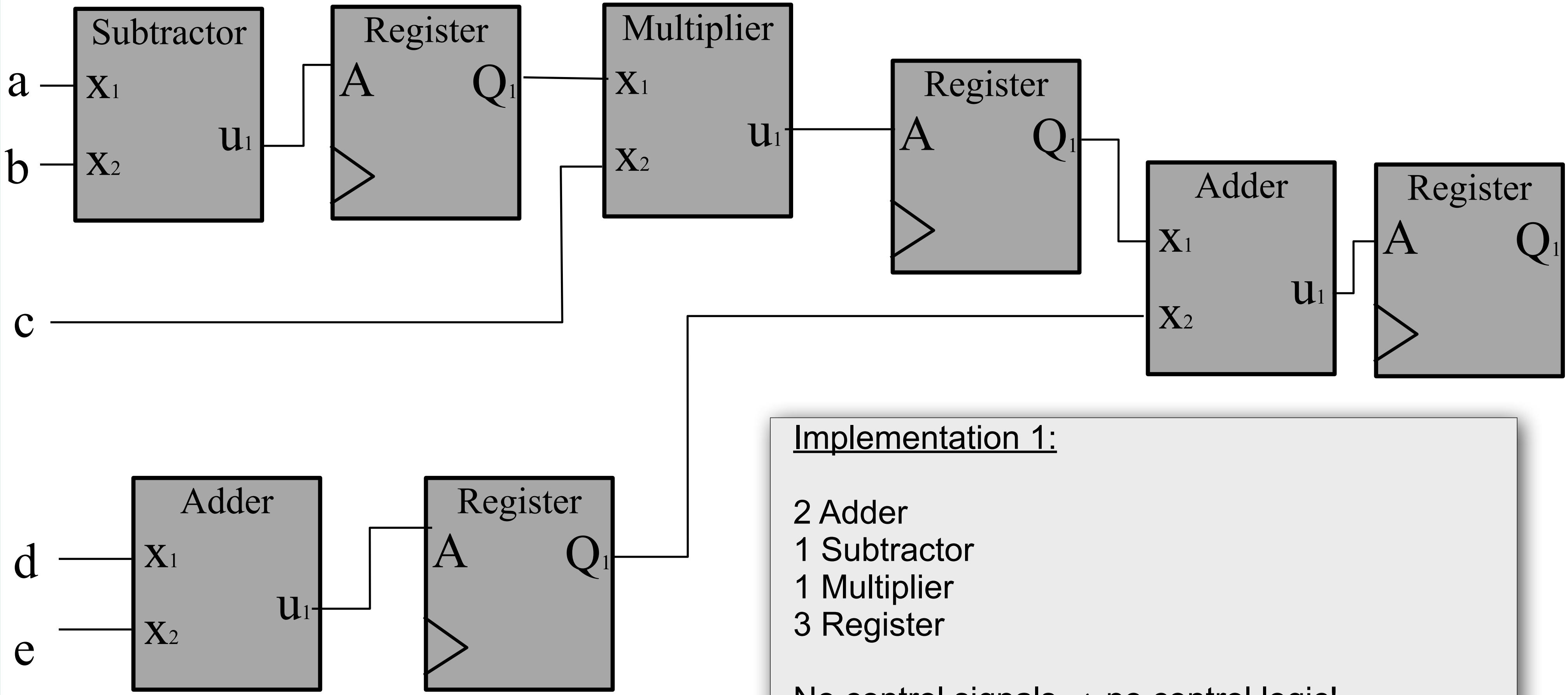
Register 1:
Result a - b

Register 2:
Result d+e

Register 1:
Result (a-b)*c

Register 1:
Final result y

Component Binding - Example (without reuse)

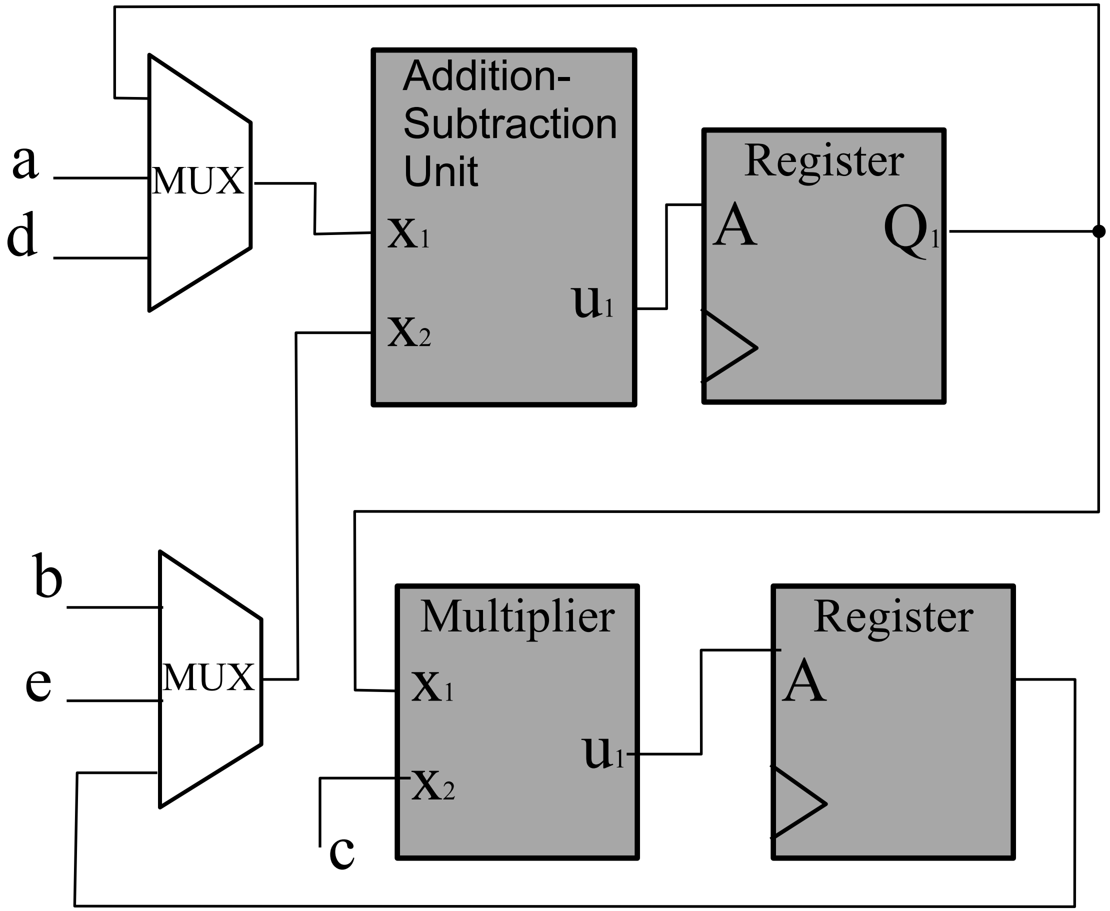


Implementation 1:

2 Adder
1 Subtractor
1 Multiplier
3 Register

No control signals ⇒ no control logic!

Component Binding - Example (with reuse)

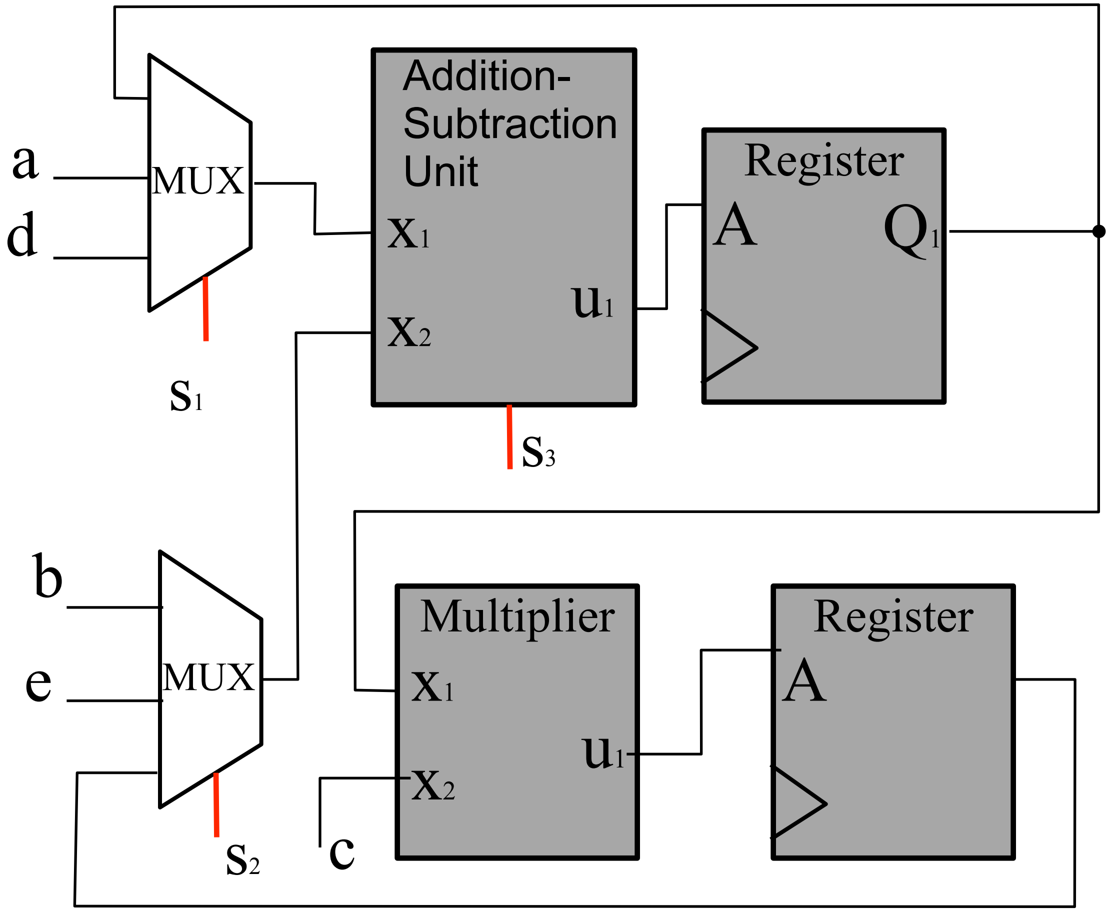


<u>Implementation 2:</u>

2 Multiplexer
1 Multiplier
1 Addition/Subtraction Unit
3 Register

Control signals at multiplexer and ALU (S1-S3)
⇒ Control logic necessary

Component Binding - Example (with reuse)



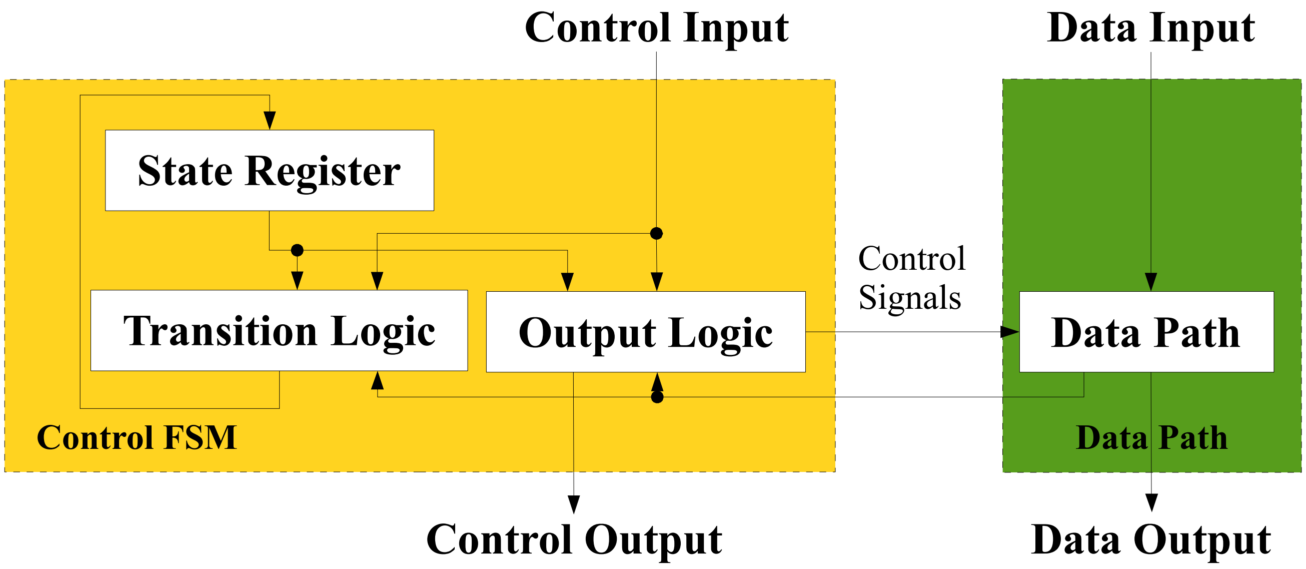
Implementation 2:

2 Multiplexer
1 Multiplier
1 Addition/Subtraction Unit
3 Register

Control signals at multiplexer and ALU (S1-S3)
⇒ Control logic necessary

RTL Code Generation

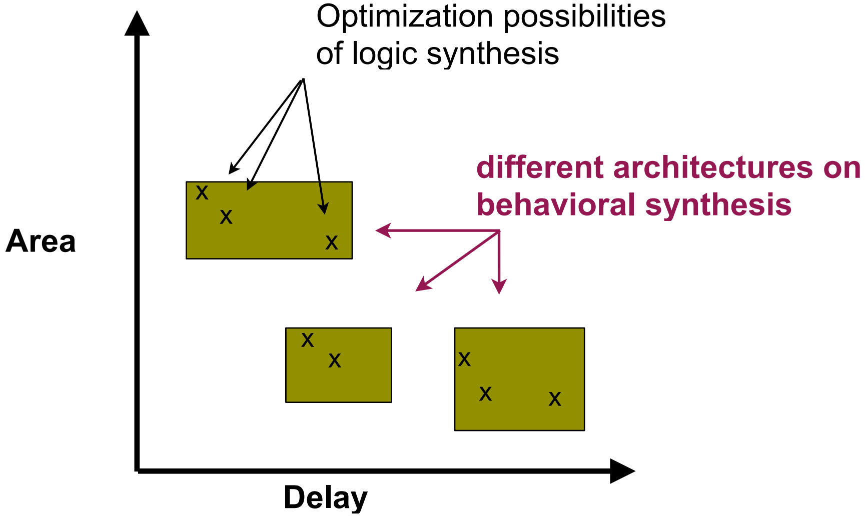
- Extraction of Data path and Control logic
- RTL-Code generation by mapping:
  - Control Logic onto a Finite State Machine(FSM)
  - Data path onto stages of Functional Units/Registers
- Followed by “normal” Logic Synthesis



HLS
-
Loop Optimizations

1. Parallelization

2. Pipelining



Parallelization

- Concurrent execution of loop bodies
- requires data independency between consecutive body executions
- Demands higher resource usage
- Speeds up the execution

original loop

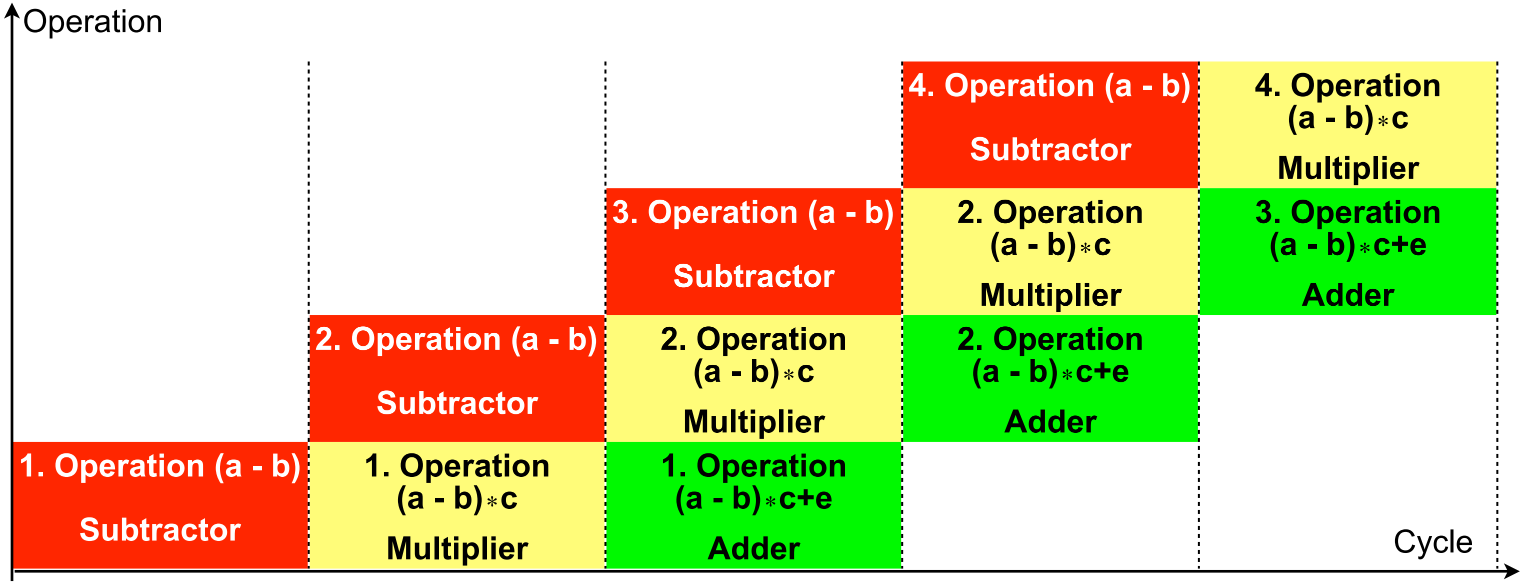
```
for ( i = 0; i < 10; i++ ) {
   d[i] = a[i] * b[i] + c[i];
}
```

fully unrolled loop

```
d[0] = a[0] * b[0] + c[0];
d[1] = a[1] * b[1] + c[1];
d[2] = a[2] * b[2] + c[2];
d[3] = a[3] * b[3] + c[3];
d[4] = a[4] * b[4] + c[4];
d[5] = a[5] * b[5] + c[5];
d[6] = a[6] * b[6] + c[6];
d[7] = a[7] * b[7] + c[7];
d[8] = a[8] * b[8] + c[8];
d[9] = a[9] * b[9] + c[9];
```

Pipelining for (a - b) ∗ c + e

- Loop execution in an “*assembly line*”
- Maximal usage of all Functional Units
- An ideal pipeline produces a new result at every clock cycle
- Demands additional resources for operation unit and stage storage



			4. Operation (a - b) **Subtractor**	**4. Operation** **(a - b)∗c** **Multiplier**
		3. Operation (a - b) **Subtractor**	**2. Operation** **(a - b)∗c** **Multiplier**	**3. Operation** **(a - b)∗c+e** **Adder**
	2. Operation (a - b) **Subtractor**	**2. Operation** **(a - b)∗c** **Multiplier**	**2. Operation** **(a - b)∗c+e** **Adder**	
1. Operation (a - b) **Subtractor**	**1. Operation** **(a - b)∗c** **Multiplier**	**1. Operation** **(a - b)∗c+e** **Adder**		

Pipeline for
(a - b) ∗ c + e

1 Multiplier
1 Adder
1 Subtractor

1. Operation:
(2 - 3) ∗ 4 + 5

2. Operation:
(5 - 4) ∗ 3 + 2

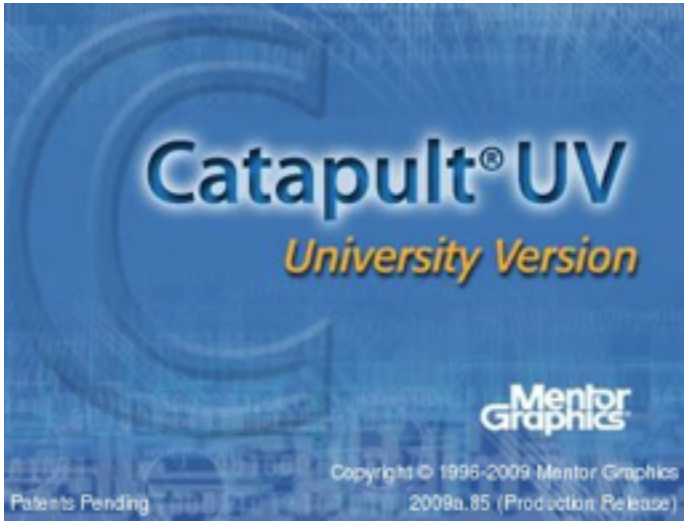
3. Operation:
(4 - 3) ∗ 2 + 5

EDA-Tools Tutorial

CALYPTO®

Design · Optimize · Verify

HLS - with CatapultC


Catapult® UV
University Version
Mentor Graphics
Copyright © 1996-2009 Mentor Graphics
Patents Pending
2009a.85 (Production Release)

CatapultC?

- High-Level-Synthesis Tool
  - since 2004 on the market
  - major role!
- generates RT-Code (VHDL&Verilog) from ANSI C/C++
- Additional output and reports:
  - Scripts to run simulation within Modelsim-Tool on various design levels
  - Schematics of the circuitry
  - Detailed reports on resource usage and timing
  - Integrated environment for test and verification
- Similar tools
  - Vivado from Xilinx (former AutoESL from AutoPilot)
  - Synphony HLS from Synopsys
  - C-to-Silicon from Cadence
  - CoDeveloper from Impulse Inc.

C/C++ in CatapultC

- CatapultC supports C/C++
- CatapultC generates Hardware on RT-Level
- There are restrictions on C/C++
- Partial support of full C/C++ language constructs:
  - Global Variables are not allowed
  - Union type not available
  - No dynamic memory allocation
  - No recursion
  - Limited pointer arithmetics (must resolve at compile time!)

CatapultC: Structures, Datentypes und Variables

- all datatypes and variables are supported (except global variables)
- Structures are subdivided into the member datatypes
- CatapultC defines their meaning in HW according to the “normal” bit widths of the X86 platforms

C++ Code	VHDL	Verilog	Signed
bool MY_Var;	STD_LOGIC My_Var;	reg My_Var;	No

char My_Var; // avoid
signed char MY_Var;
signed char int MY_Var;
	STD_LOGIC VECTOR;
	(7 downto 0) MY_Var;
		reg |7:0]
		My_Var;
			Yes

unsigned char MY_Var;
unsigned char int MY_Var;
	STD_LOGIC VECTOR
	(7 downto 0) MY_Var;
		reg [7:0]
		MY_Var;
			No

short MY_Var;
signed short MY_Var;
signed short int MY_Var;
	STD_LOGIC_VECTOR
	(15 downto 0) MY_Var;
		reg [15:0]
		My_Var;
			Yes

CatapultC: specialized Datentypes

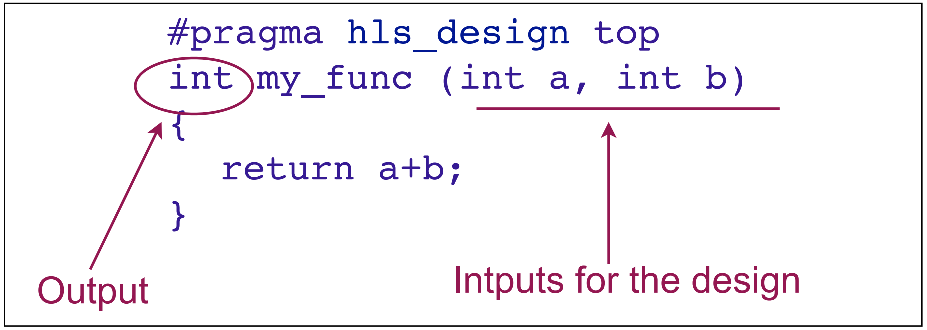
Catapult introduces new data types to explicitly express sign and bit width:

- supports SystemC data types (sc_int<> and friends)
- “Algorithmic C” data types

Data type	Description	Range
ac_int<W,false>	unsigned integer	0 bis $2^{W} - 1$
ac_int<W,true>	signed integer	-2^{W-1} bis $2^{W-1} - 1$
ac_fixed<W,I>	unsigned fixed-point	0 bis $(1 - 2^{-W}) * 2^{I}$
ac_fixed<W,I>	signed fixed-point	$(-0.5) * 2^{I}$ bis $(0.5 - 2^{-W}) * 2^{I}$

CatapultC: Functions, Ports und Interfaces

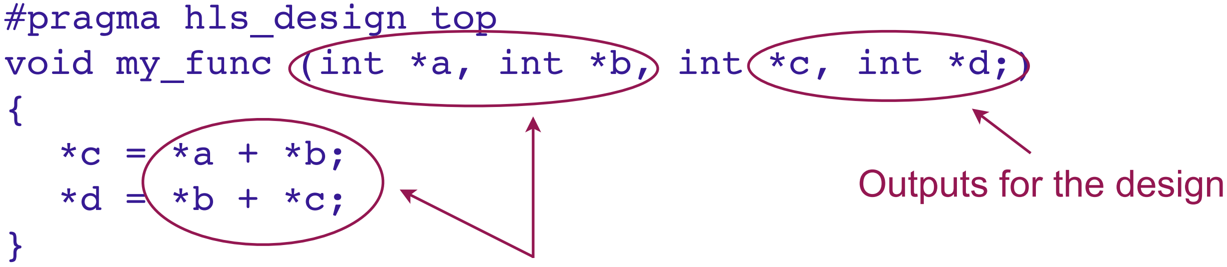
- Parameters and Return value of functions are the input/output ports of the resulting circuitry
- Pointer, Arrays and References are output ports if they are written within the function
- Arrays result in RAM and require a size statement if used as parameter



```
#pragma hls_design top
int my_func (int a, int b)
{
   return a+b;
}
```

Output

Intputs for the design



```
#pragma hls_design top
void my_func (int *a, int *b, int *c, int *d;)
{
   *c = *a + *b;
   *d = *b + *c;
}
```

Outputs for the design

CatapultC: Loops

- Loops are perfectly supported if they have a fixed number of iterations
- Loops with “dynamic” iteration count should have a fixed upper limit:

```
const int MAX = 10;
for (i=0; i<MAX; i++)
{
  if (i==n) break;  //variable termination condition
}
```

- Loops are subject to extensive optimization by the synthesis tool (unrolling and pipelining)

CatapultC: Loops

- Loops are perfectly supported if they have a fixed number of iterations
- Loops with “dynamic” iteration count should have a fixed upper limit:

```
const int MAX = 10;
for (i=0; i<MAX; i++)
{
  if (i==n) break;  //variable termination condition
}
```

- Loops are subject to extensive optimization by the synthesis tool (unrolling and pipelining)

More on all this in:

High-Level Synthesis Blue Book. Michael Fingeroff

online: http://www.eet.bme.hu/~timar/data/hls bluebook uv.pdf

Tasks to exercise your knowledge!

Task 1 for this Tutorial

Given is the following C-Funktion:

```
void calculate(int x[], int y[], int z[])
{
  for (i=0; i<15; i++)
  {
    z[i] = x[i] * i + y[i];
  }
}
```

The FOR-Loop of this function is under investigation!

Multiplication and Addition will take 1 clock cycle to execute. Memory accesses are not considered.

Question A) The loop will be partially unrolled by factor 3

- How many cycles will it take to execute the loop?
- Which type and number of functional resources will be used?

Question B) The loop will be pipelined

- How many cycles will it take to execute the loop?
- Which type and number of functional resources will be used?

Task 2 for this Tutorial

- Complete the following C function as paper work. It is a preparation to the practical course on HLS.
- <u>Questions</u> to the tutor and <u>review</u> of YOUR solution is possible during this lesson/tutorial.

```
#define MAX_DATASIZE 256
#define A 123
#define B 456
#define C 789

void compute( int data[MAX_DATASIZE], int datasize, int &result){

// Compute the equation x*x*A + x*B + C for each integer in
// the input buffer data.
// The fill level of the buffer is given by parameter datasize.
// Return the resulting data in the result reference.

//put your code here!

}
```

How to proceed from here?

- There will be an Tutorial (next week) and a Lab (Practical course 4) on this topic!
- Make sure to attend them!
- Enroll/Subscribe in OPAL!
- Prepare yourself! Read the material for the LAB!
- Recover how to program C/C++ language!
- Do some simple Online-Tutorials on C/C++ programming!