Tassadaq Hussain

VLSI Design

High Level Synthesis

Outline

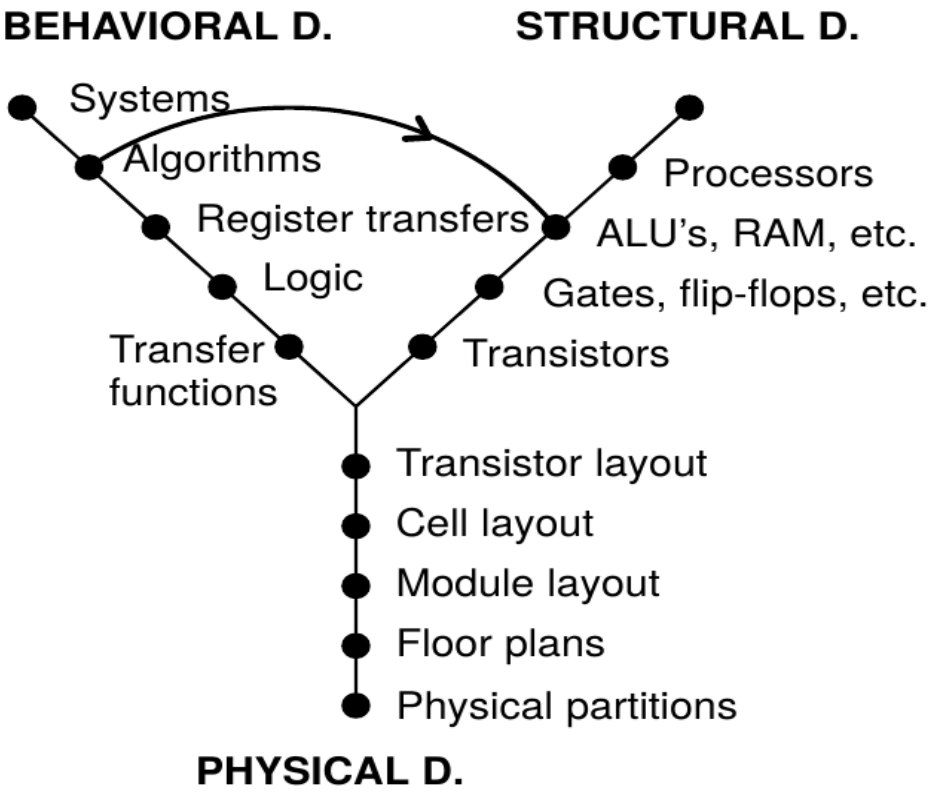
ROCCC

LegUP

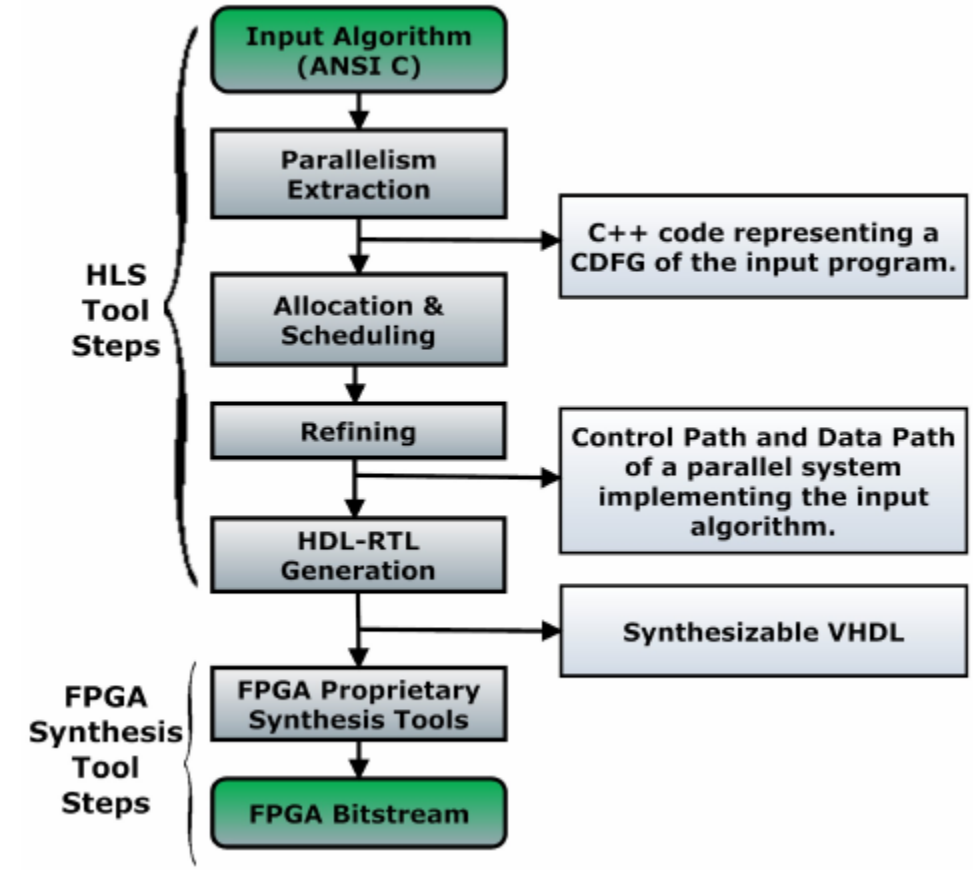
Modelsim

HIGH-LEVEL SYNTHESIS

High-level synthesis: the automatic addition of structural information to a design described by an algorithm.

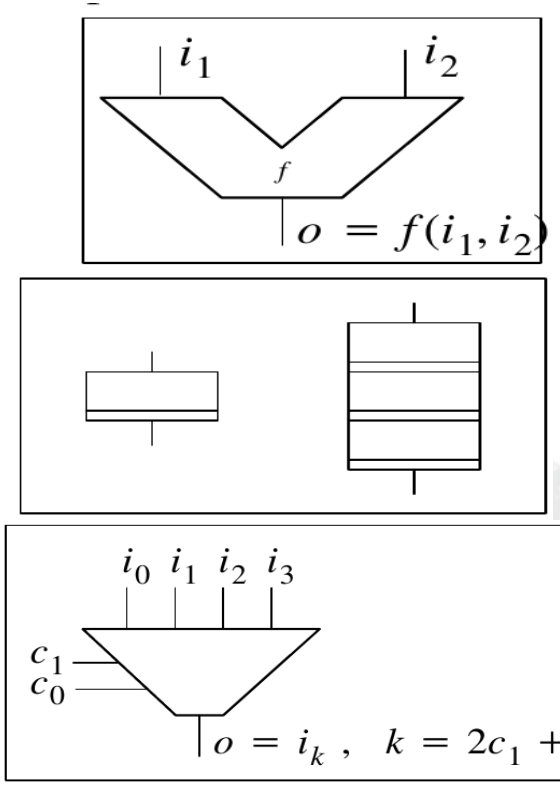


HLS Synthesis Flow



HARDWARE MODELS FOR HIGH-LEVEL SYNTHESIS

- All HLS systems need to restrict the target hard-ware.
- All synthesis systems have their own peculiarities; but most systems generate synchronous hardware and build it with the following parts:
  - ALU
  - Registers
  - MUX
  - Buses
  - Three State Driver (Controller)



HLS Hardware Concepts

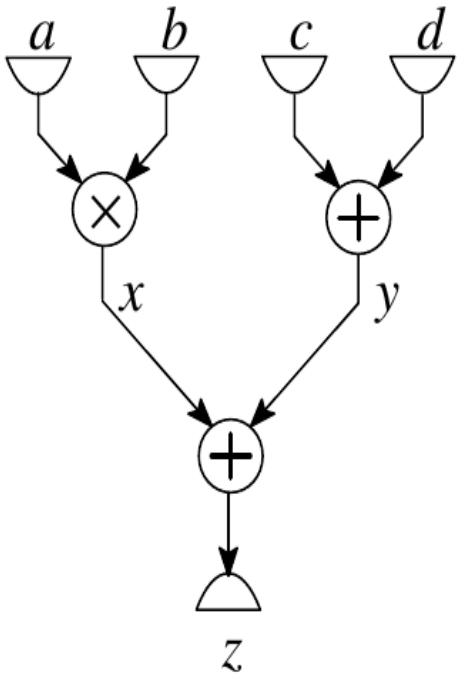
Data Path + Control Structure

The data path: a network of functional units, registers, multiplexers and buses. The actual ‘‘computation’’ takes place in the data path.

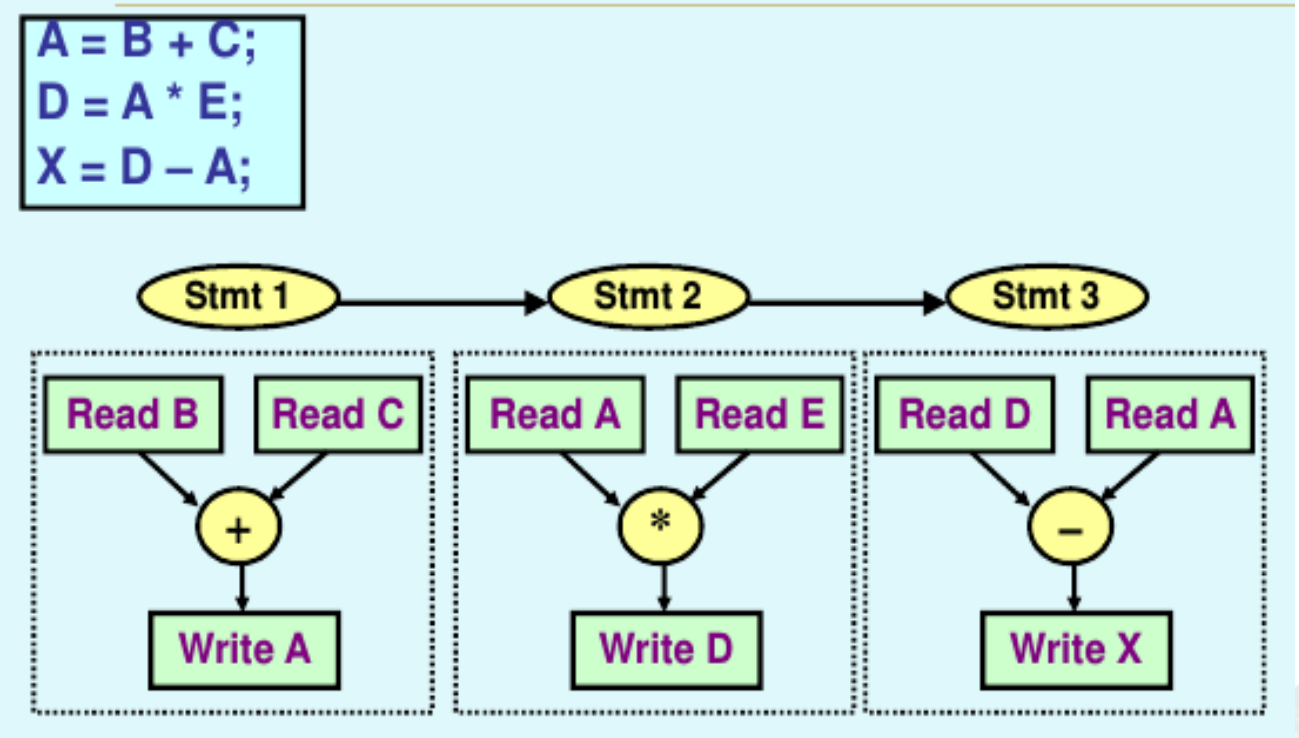
Control: the part of the hardware that takes care of having the data present at the right place at a specific time, of presenting the right instructions to a programmable unit, etc.

Data-flow Graph

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

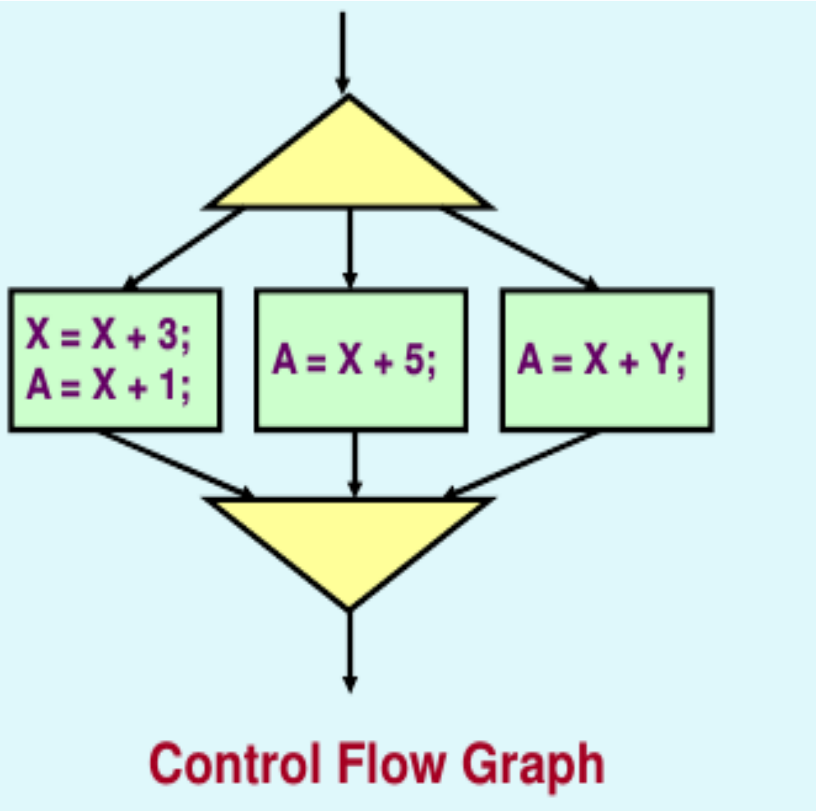


Data-flow Graphs



Control-flow Graph

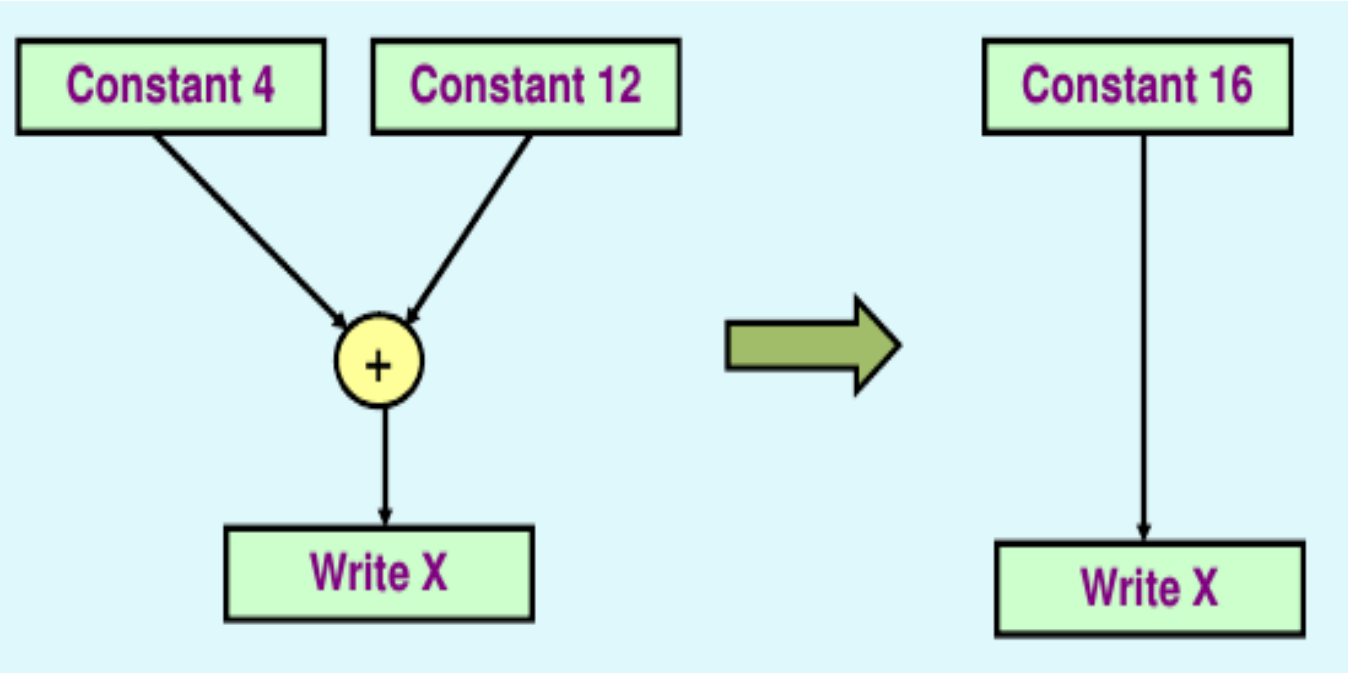
```
case (C)
1:
begin
X = X + 3;
A = X + 1;
end
2:
A = X + 5;
default:
A = X + Y;
endcase
```



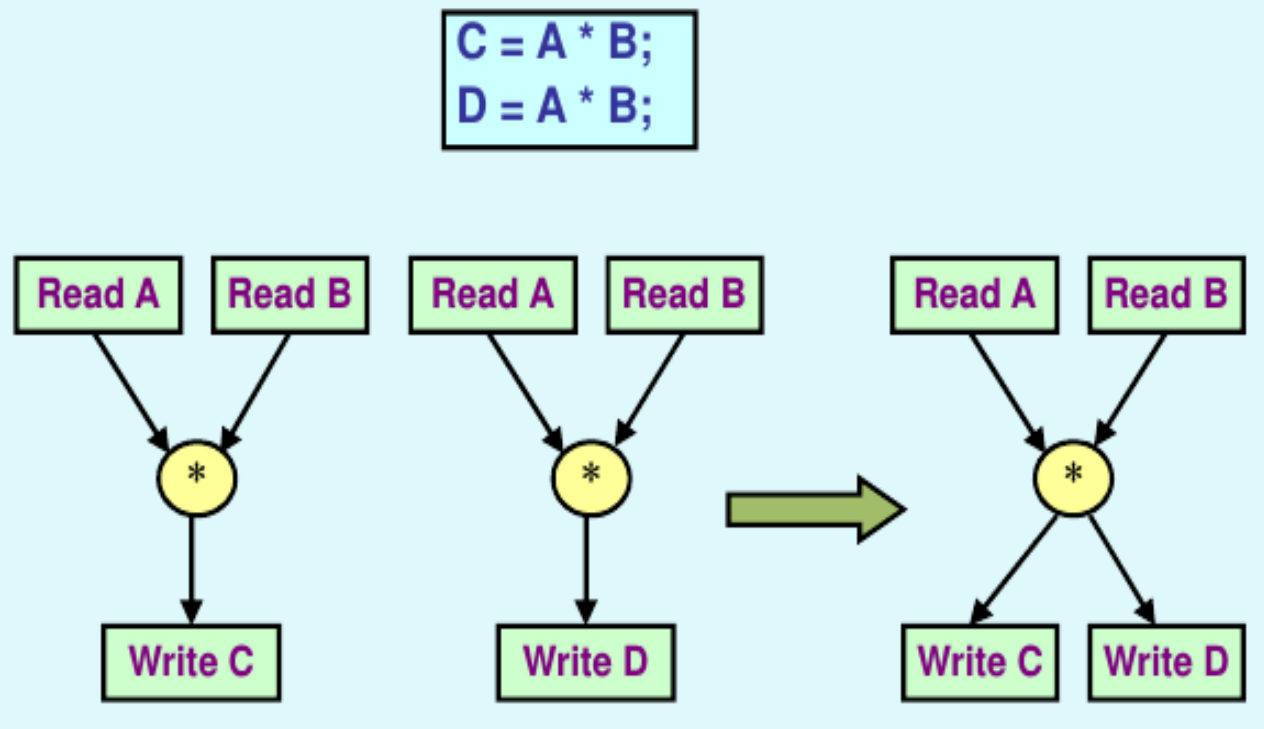
HLS Compiler Transformation

- Constant folding
- Redundant operator elimination
- Tree height transformation
- Control flattening
- Logic level transformation
- Register-Transfer level transformation

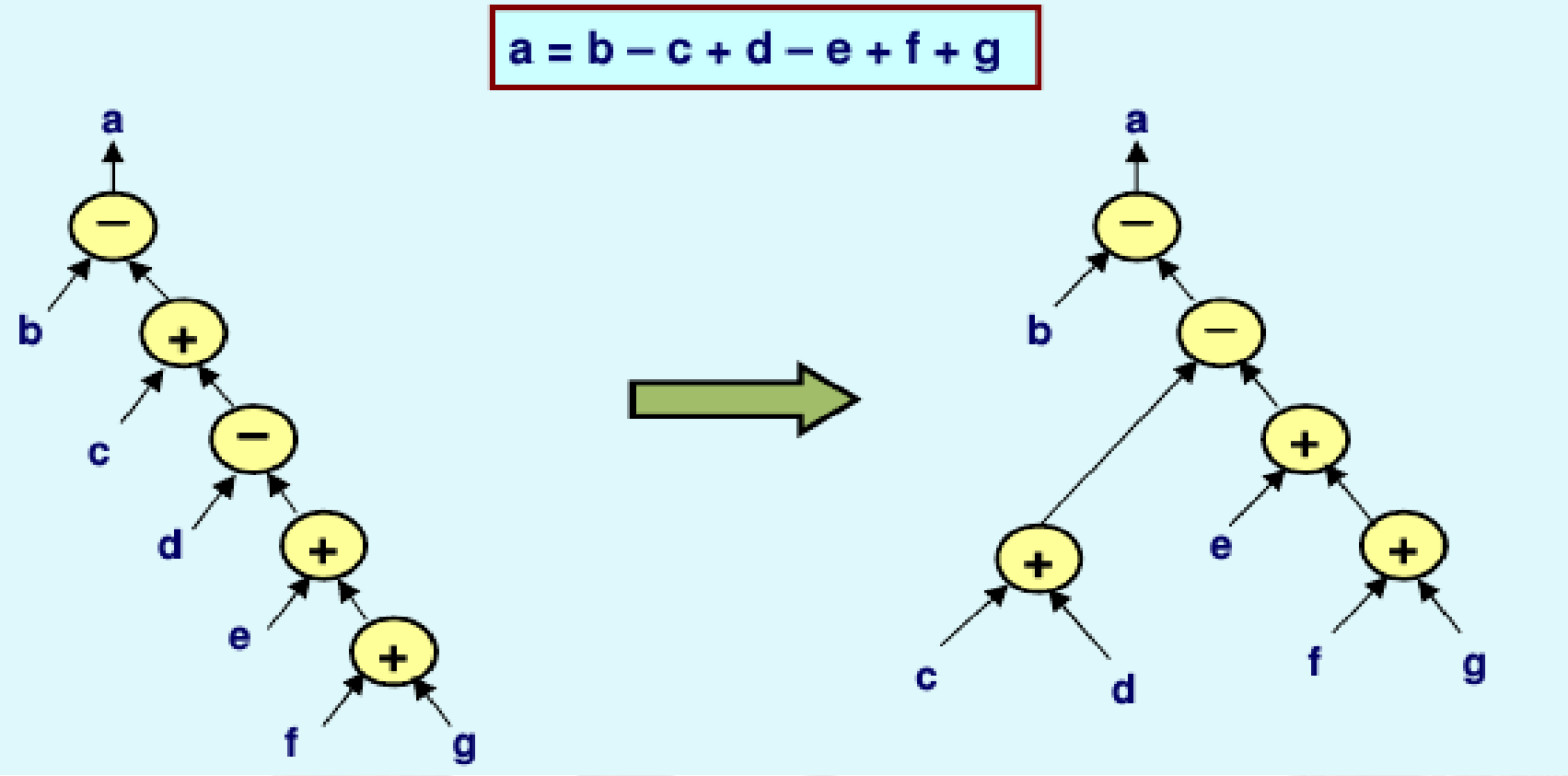
Constant Folding



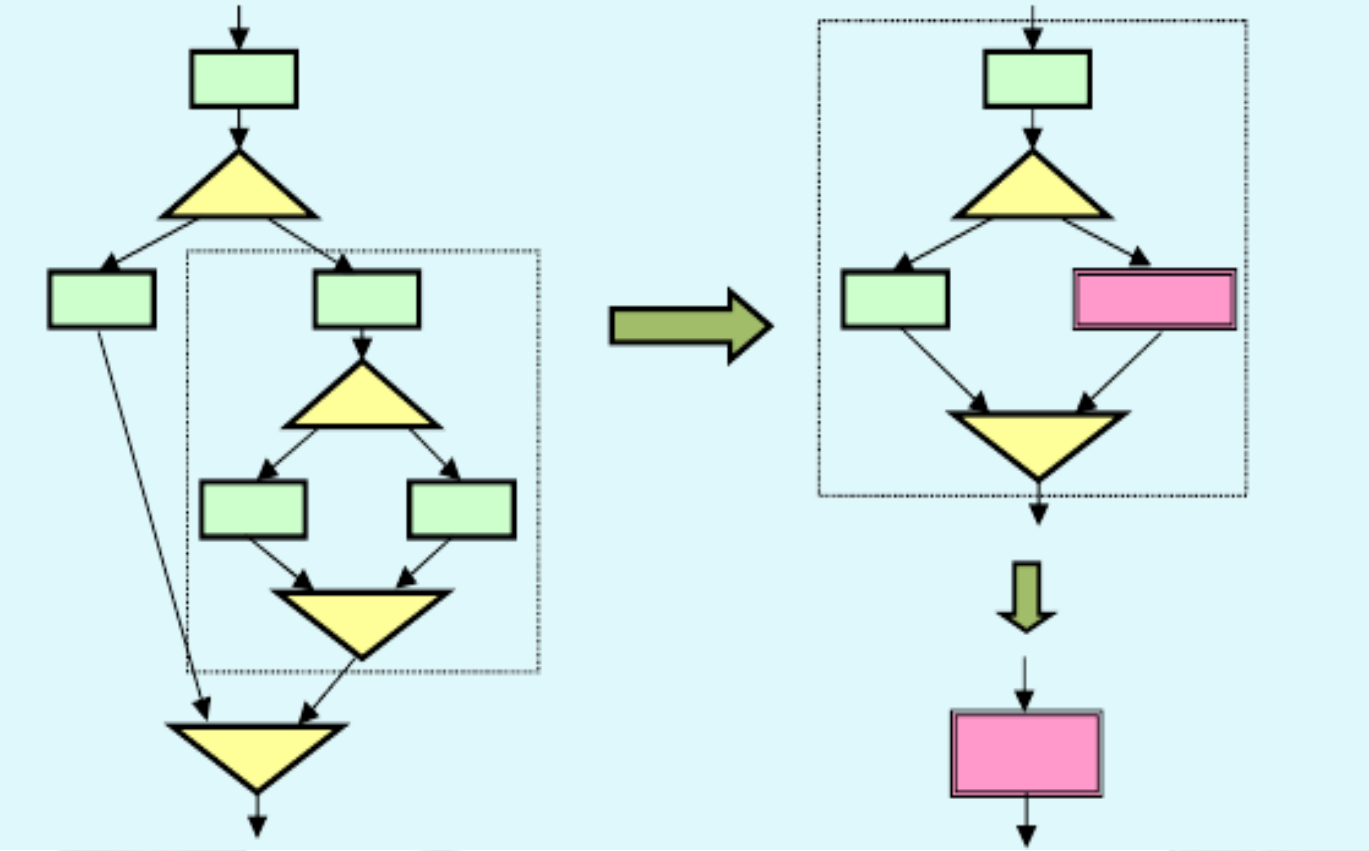
Redundant operator elimination



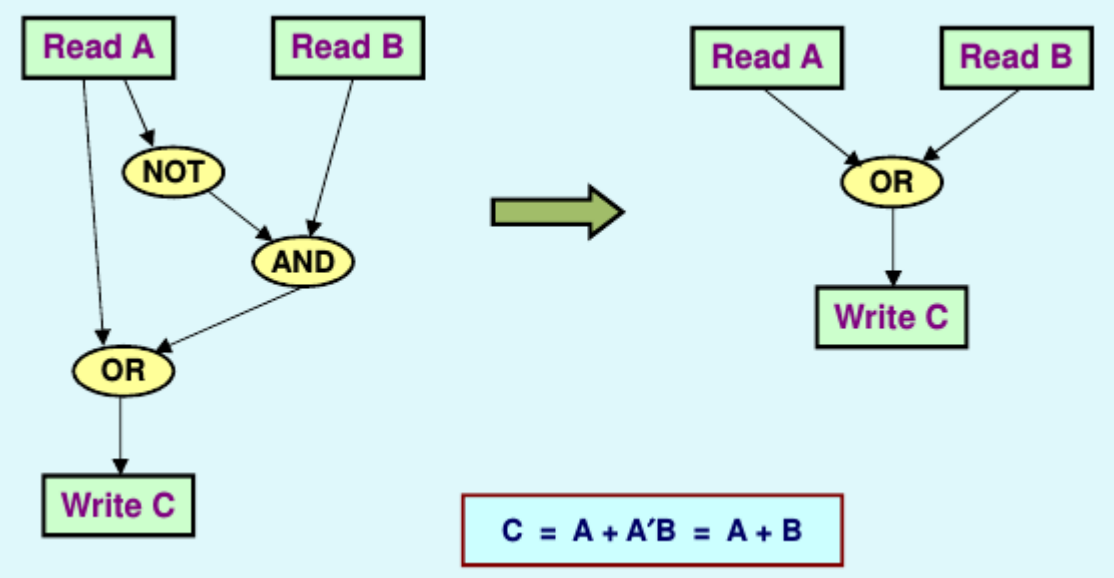
Tree height transformation



Control flattening

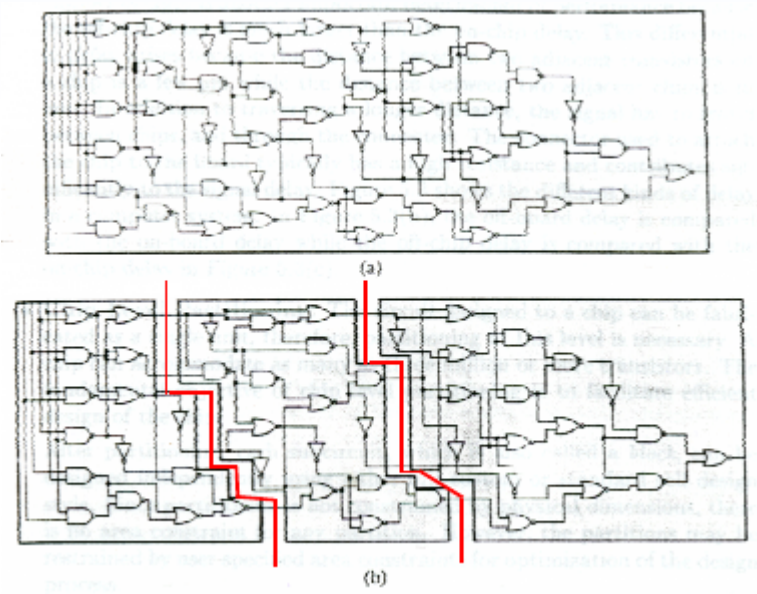


Logic Level Transformation



Partitioning

- Scheduling
- Allocation
- Unit selection

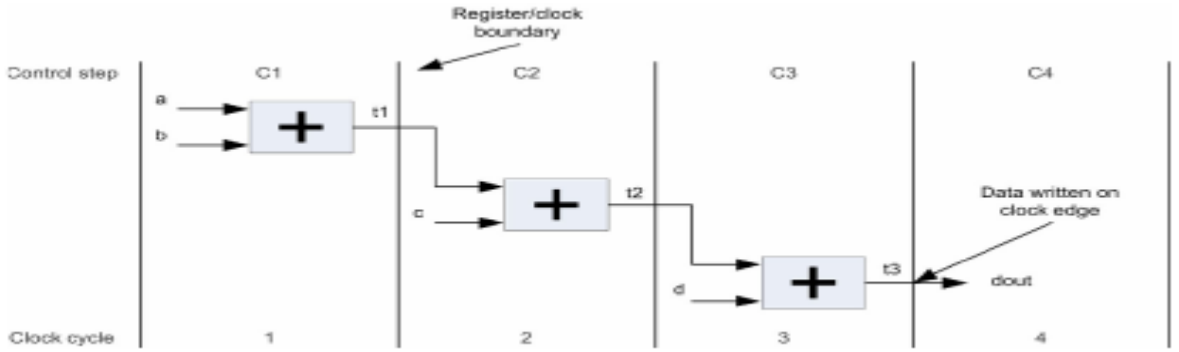


Loop Unrolling

Loop unrolling is the primary mechanism to add parallelism into a design. This is done by automatically scheduling multiple loop iterations in parallel, when possible.

SCHEDULING

- Task of assigning behavioral operators to control steps.
  - Input:
    - ✓ Control and Data Flow Graph (CDFG)
  - Output:
    - ✓ Temporal ordering of individual operations (FSM states)
  - Basic Objective:
    - ✓ Obtain the fastest design within constraints (exploit parallelism).



Scheduling Algorithms

Three popular algorithms:

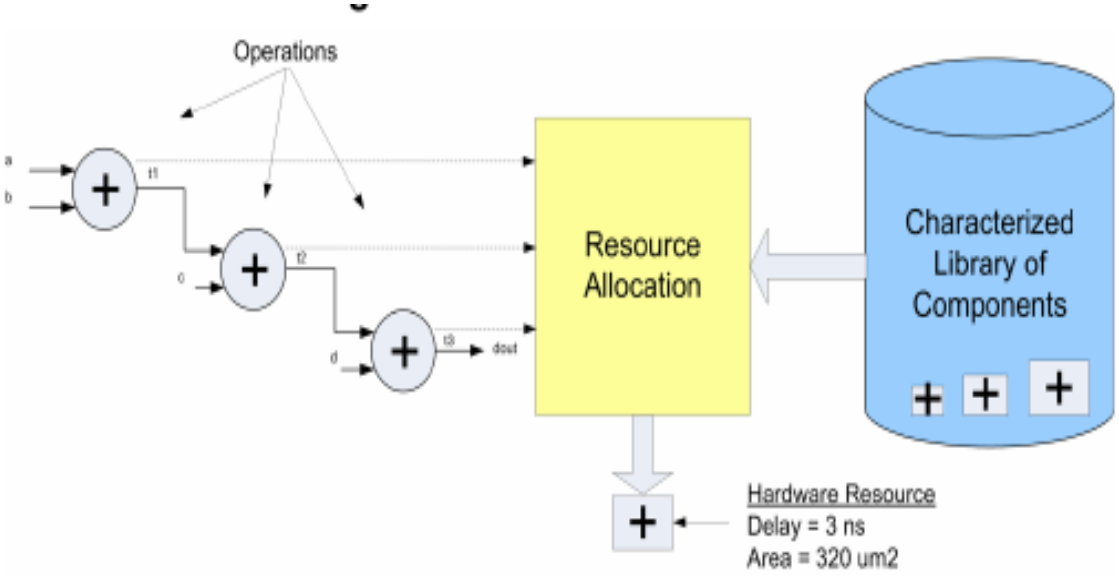
– As Soon As Possible (ASAP)

– As Late As Possible (ALAP)

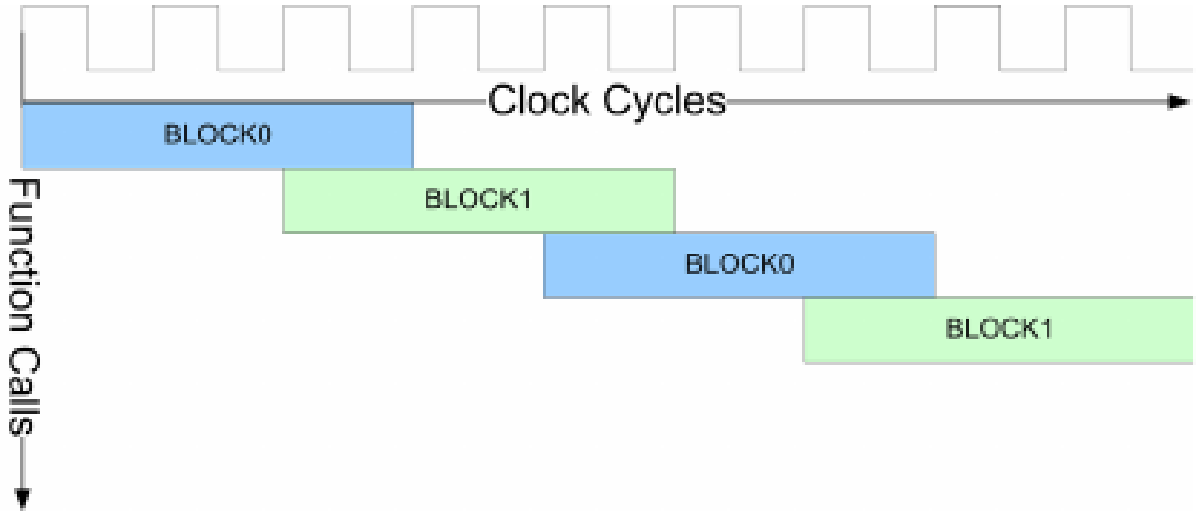
– Resource Constrained (List scheduling)

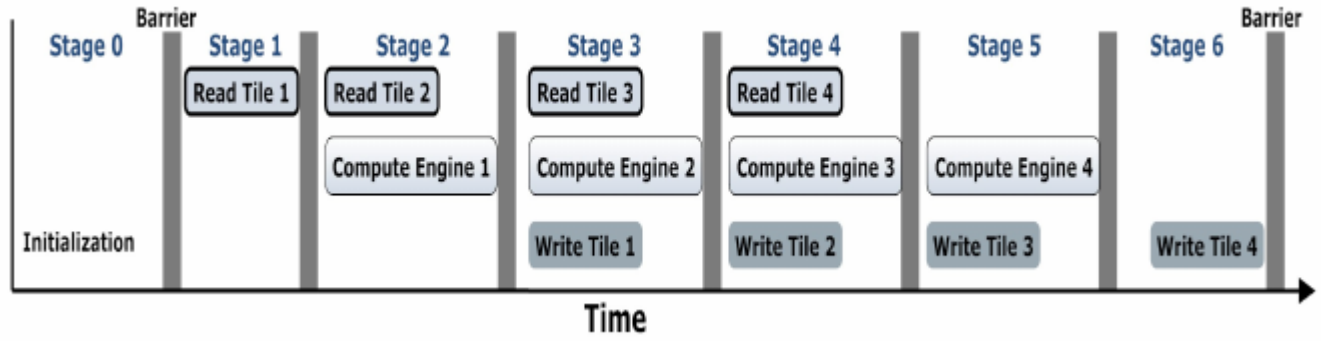
Resource Allocation

Once the DFG has been assembled, each operation is mapped onto a hardware resource which is then used during scheduling.



Pipeline and Overlap





Memories

Smart Buffers:

To manage on-chip data for computationally intensive window (loop) operations, the HLS use smart buffers in accelerators. The smart buffer helps to minimize the accesses to he Main Memory for programs that operate on static data structures and to perform loop operations over arrays. The smart buffer is a part of ASHA and uses FPGA resources.

Scratch-pad Memory:

The Scratch-pad is a fast directly addressed software managed SRAM memory. The Scratch-pad has better real-time guarantees than caches and by its significantly lower overheads it is better in access time, energy consumption and area. Recent advances have made much progress in compiling static structures into scratch-pad memory that enable several performance enhancements. Instead of using traditional load/store instructions the scratch-pad uses direct memory-memory operations using DMA. The Scratch-pad memory access uses source and destination address registers, each of which holds a starting address of the memory.

Functional Verification

High-level functional verification provides substantial decrease in the test generation time, test application time. By utilizing debug/verify facility it increases the fault coverage and decrease area/delay overheads.

Benefits of HLS

- ✓ Reducing design and verification efforts
- ✓ More effective reuse
- ✓ Investing R&D resources where it really matters
- ✓ Testing and verifications

Example

```
void MaxFilter(int A0, int A1, int A2, int& max)
{
  int tmp ;
  if (A0 > A1)
  {
    tmp = A0 ;
  }
  else
  {
    tmp = A1 ;
  }
  if (tmp > A2)
  {
    max = tmp ;
  }
  else
  {
    max = A2 ;
  } }
```

FIR Filter

$$y[n] = \sum_{k=0}^{N} h_k x[n-k]$$

```
/*  A five-tap FIR filter.*/
typedef struct
{  // Inputs
  int A0_in ;
  int A1_in ;
  int A2_in ;
  int A3_in ;
  int A4_in ;
  // Outputs
  int result_out ;
} FIR_t ;
```

```
FIR_t FIR(FIR_t f)
{  // Should be propagated
  const int T[5] = { 3, 5, 7, 9, 11 } ;
  f.result_out = f.A0_in * T[0] +
    f.A1_in * T[1] +
    f.A2_in * T[2] +
    f.A3_in * T[3] +
    f.A4_in * T[4] ;
  return f ;
}
```

Digital Filter

```
#include "roccc-library.h"
void firSystem()
{
  int A[100] ;
  int B[100] ;
  int i ;
  int myTmp ;
  for(i = 0 ; i < 100 ; ++i)
  {
    // The mapping of the signals must match the order in which they appear
    //  in the exported struct.  Hence, the switching of the i+2 and i+3
    //  elements.
    FIR(A[i], A[i+1], A[i+3], A[i+2], A[i+4], myTmp) ;
    B[i] = myTmp ;
  }

}
```