BSC~ Microsoft Research Centre

High Level Synthesis

VLSI Design

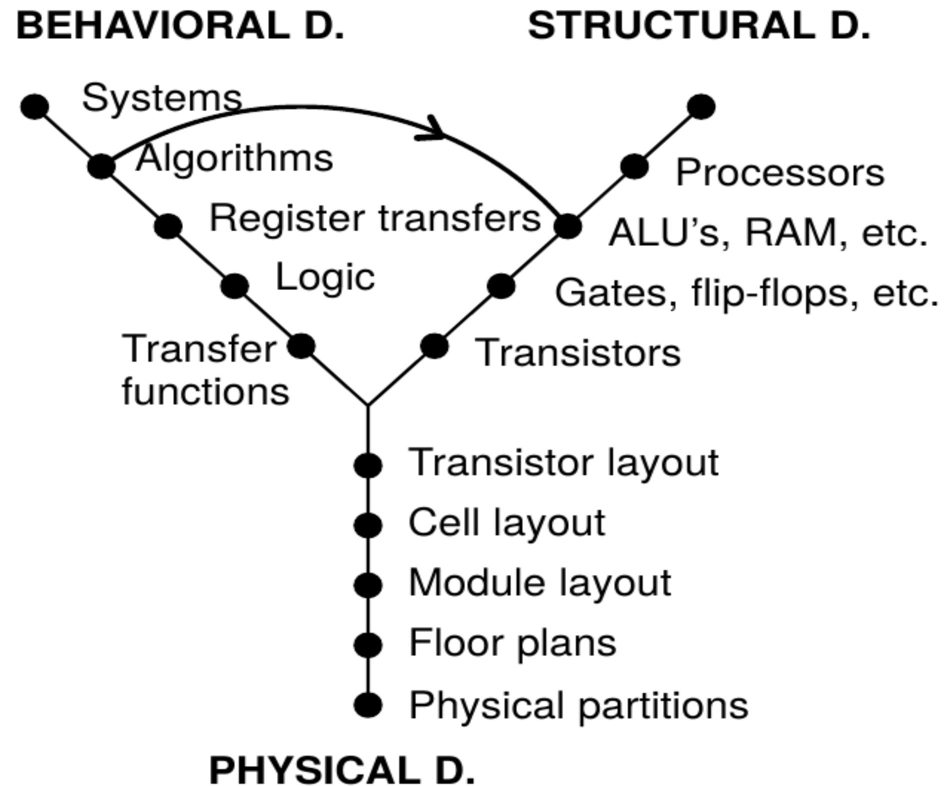
Tassadaq Hussain Cheema

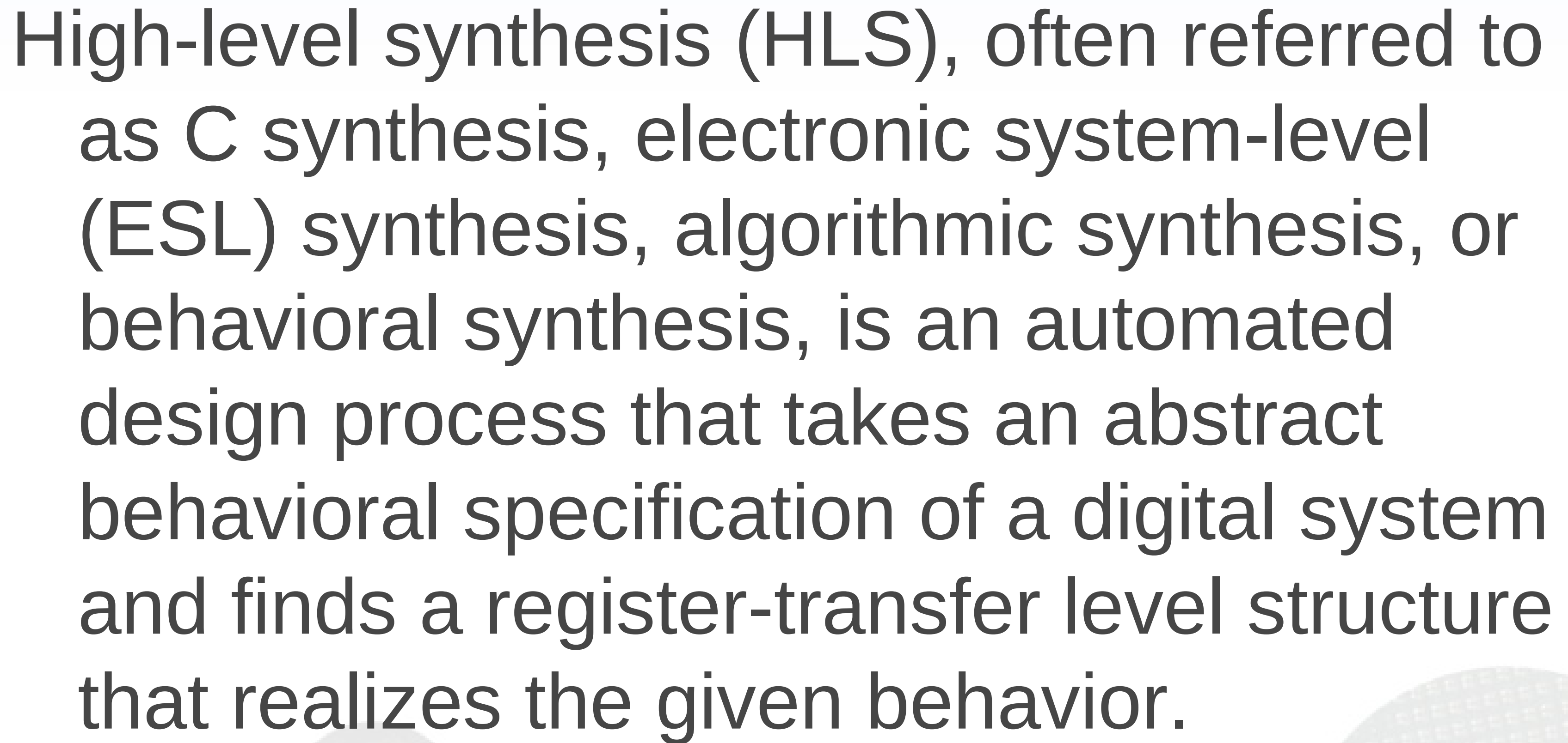
Professor EE Department

PakASIC.com

HIGH-LEVEL SYNTHESIS

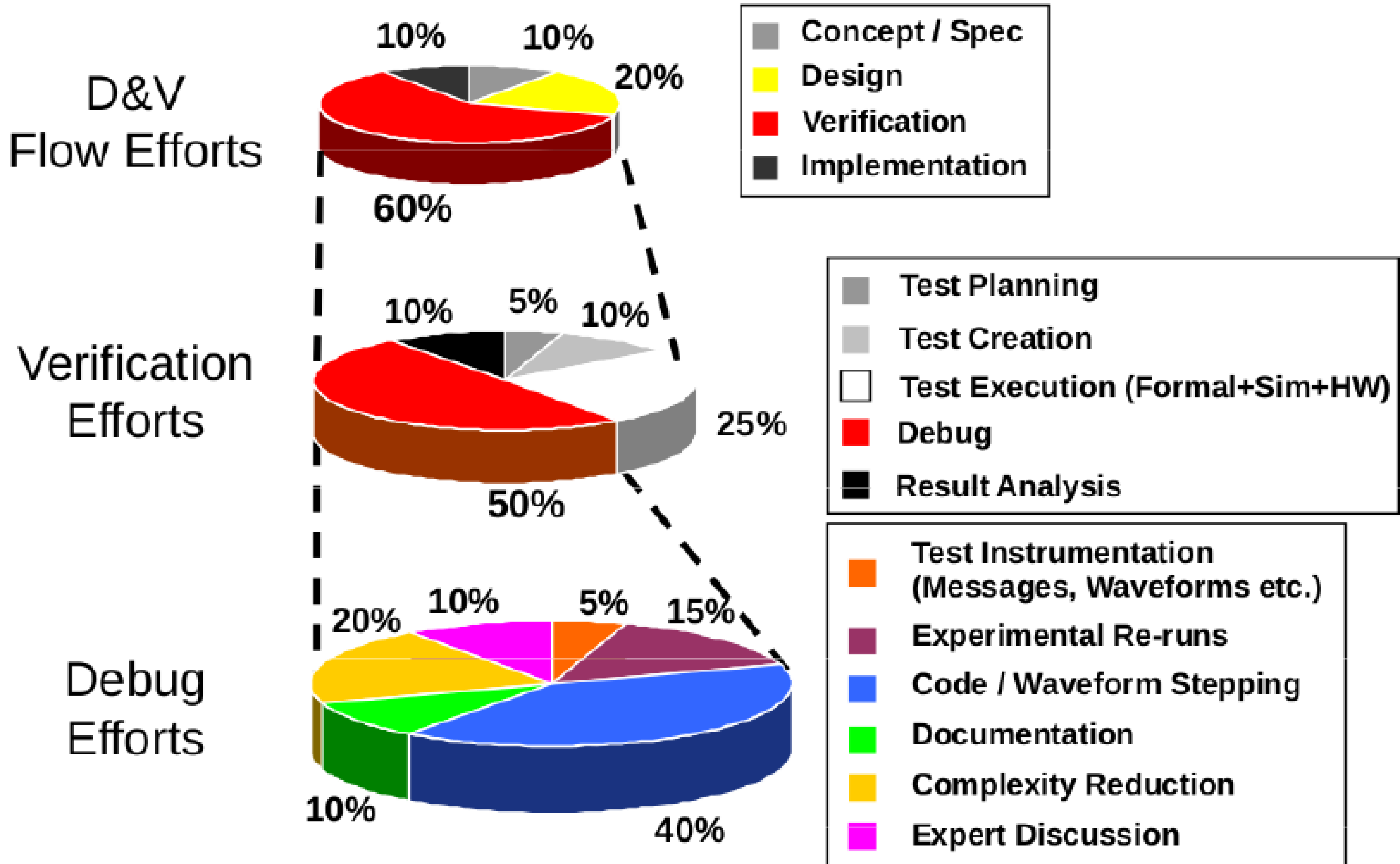
High-level synthesis: the automatic addition of structural information to a design described by an algorithm.

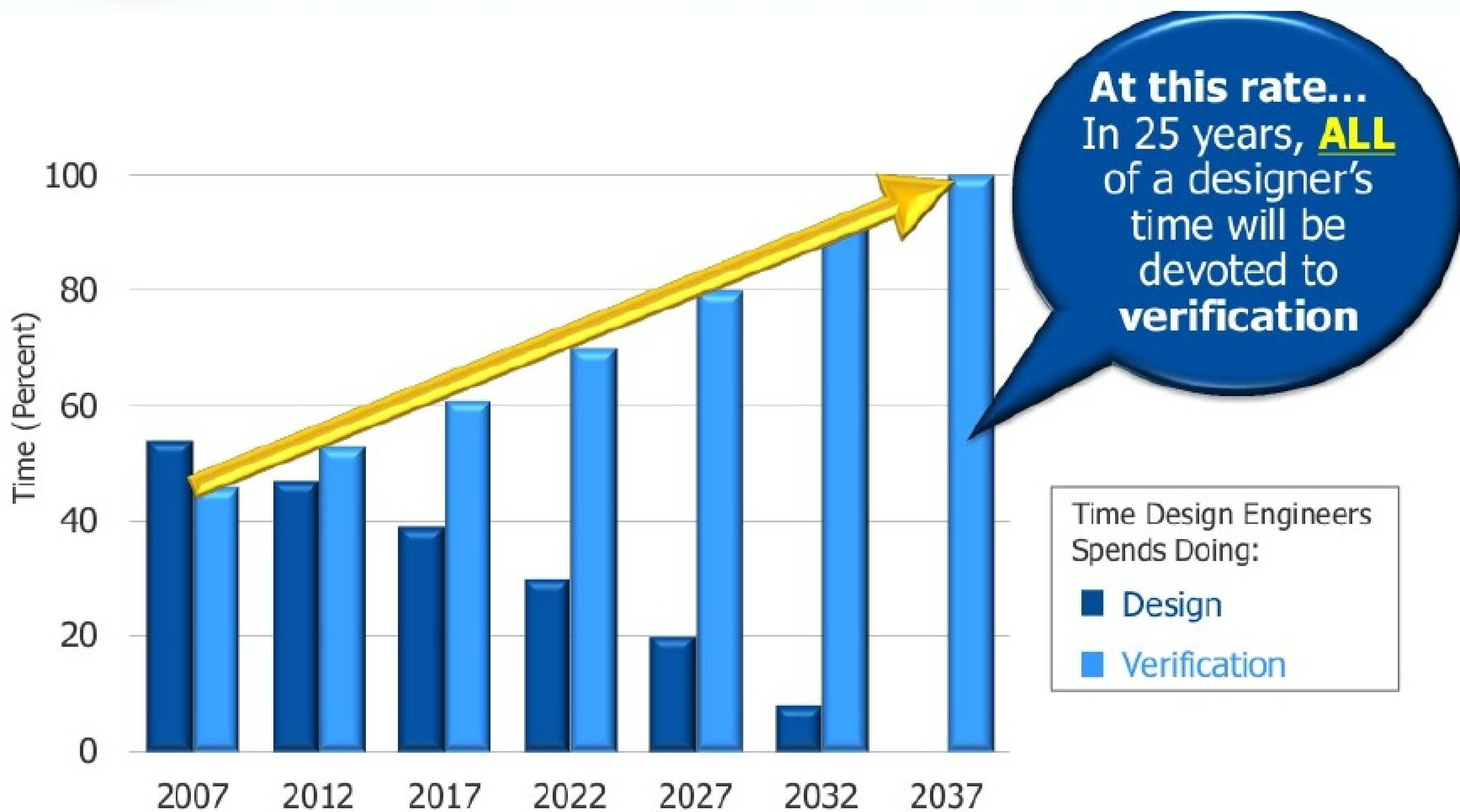




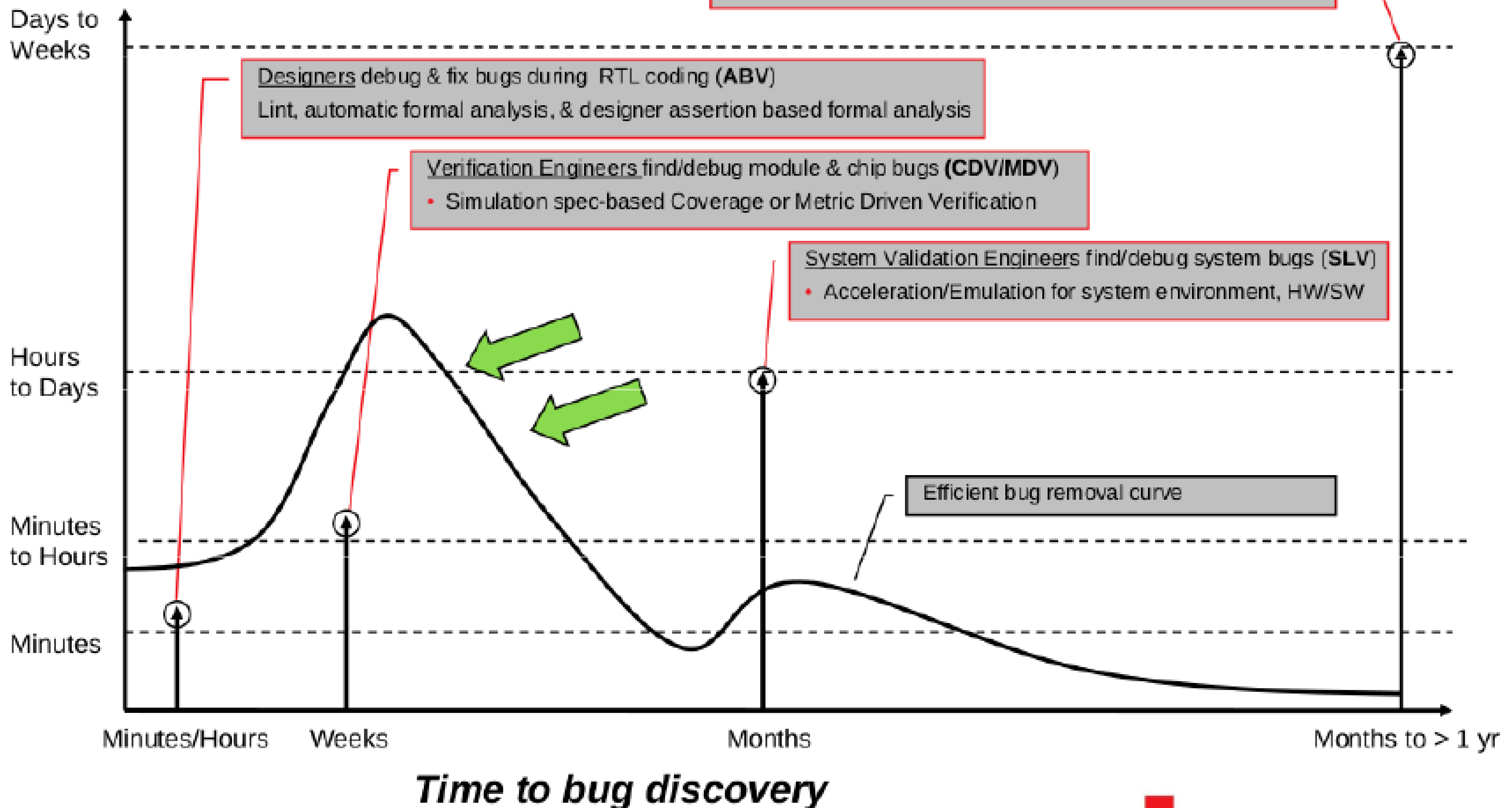
High-level synthesis (HLS), often referred to as C synthesis, electronic system-level (ESL) synthesis, algorithmic synthesis, or behavioral synthesis, is an automated design process that takes an abstract behavioral specification of a digital system and finds a register-transfer level structure that realizes the given behavior.

Why HLS

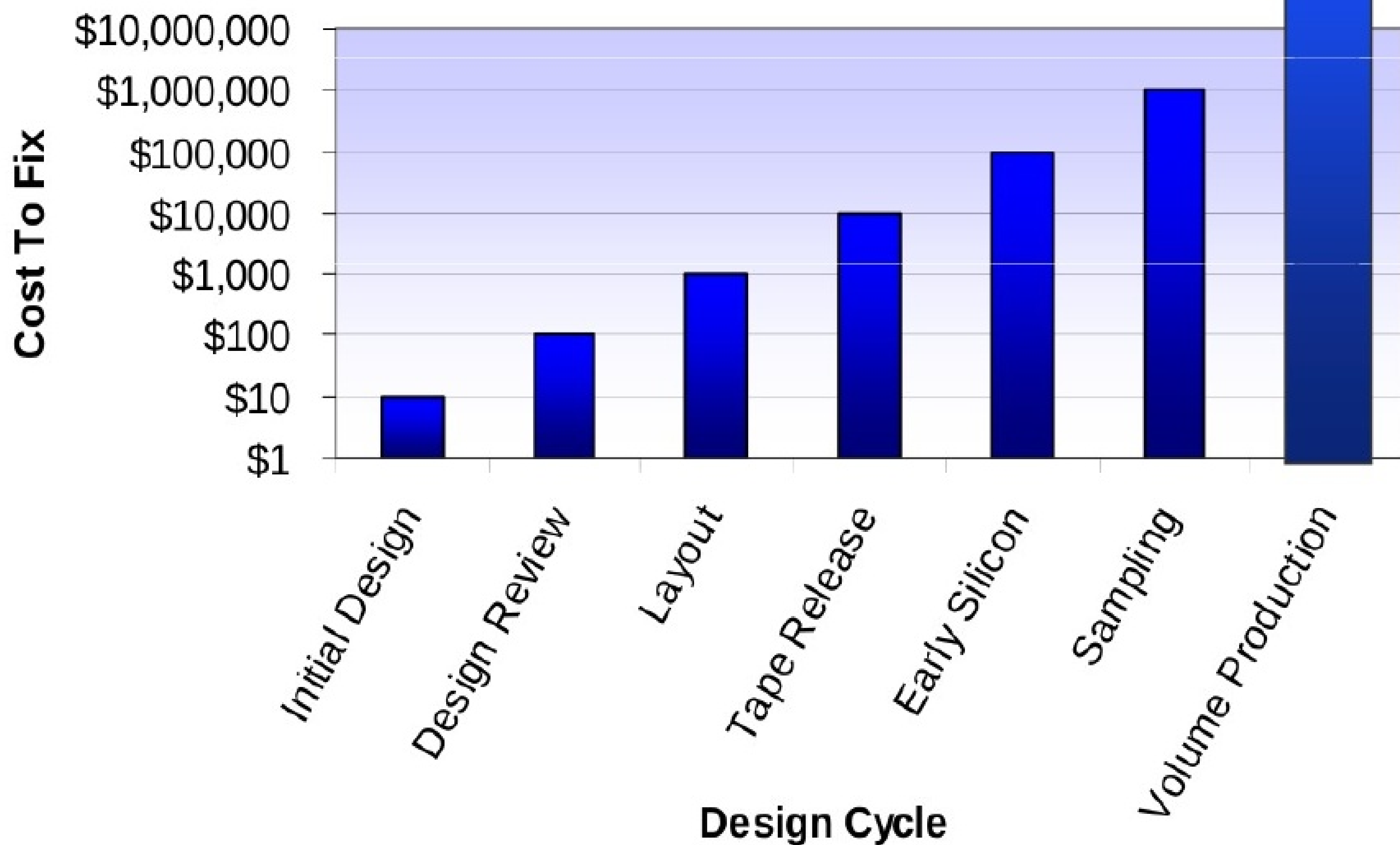



At this rate…
In 25 years, ALL of a designer's time will be devoted to verification
100
80
60
40
20
0
Time (Percent)
2007
2012
2017
2022
2027
2032
2037
Time Design Engineers Spends Doing:
Design
Verification

Late-Stage Bugs Are More Costly



The Relative Cost of Finding Bugs



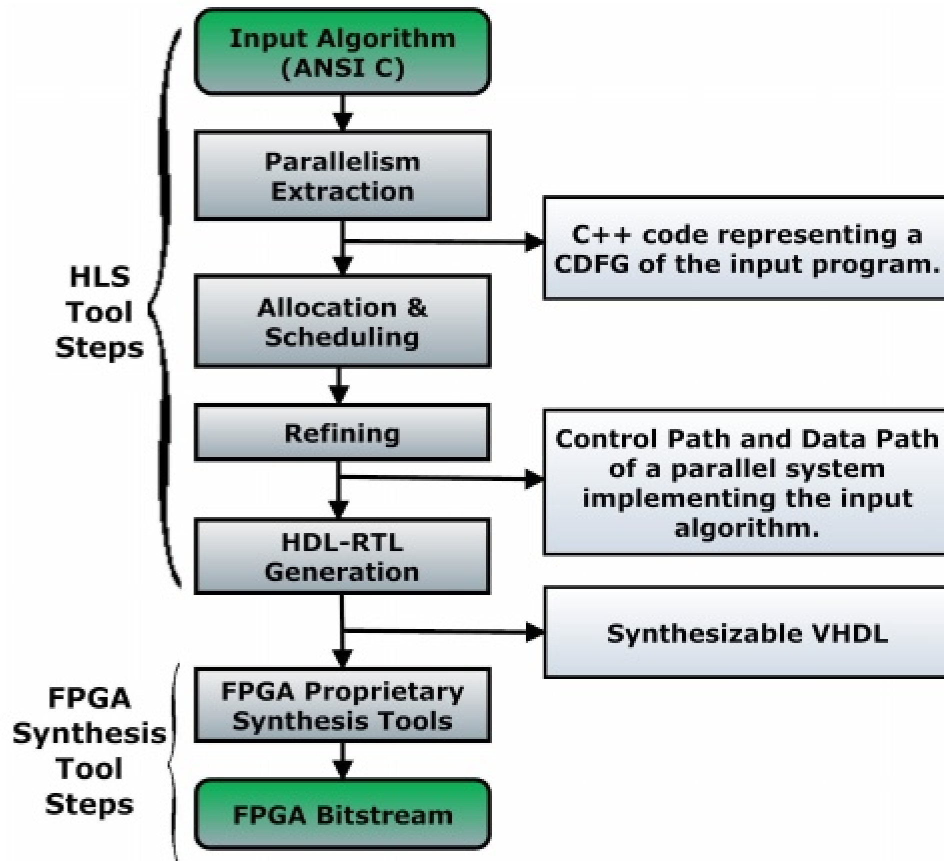
Functional Verification

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.

HLS Tools

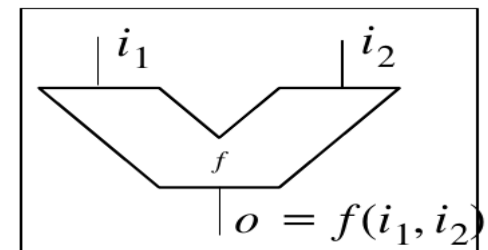
Stratus HLS
AUGH
eXCite
Bambu
Bluespec
QCC
CHC
CoDeveloper
HDL Coder
CyberWorkbench
Catapult
DWARV
GAUT
Hastlayer
Instant SoC
Intel High Level Synthesis Compiler
LegUp HLS
LegUp
MaxCompiler
ROCCC
Symphony C
VivadoHLS
Kiwi
CHiMPS
gcc2verilog
HercuLeS
Shang

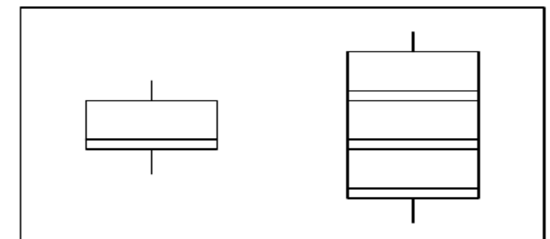
HLS Synthesis Flow

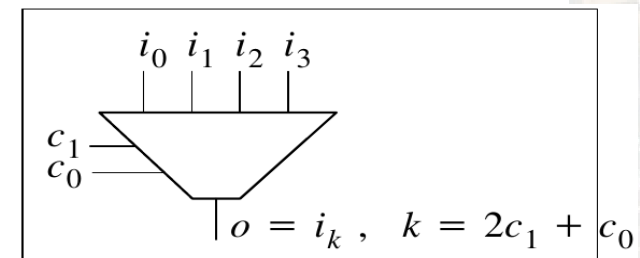


HARDWARE MODELS FOR HIGH-LEVEL SYNTHESIS

- 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

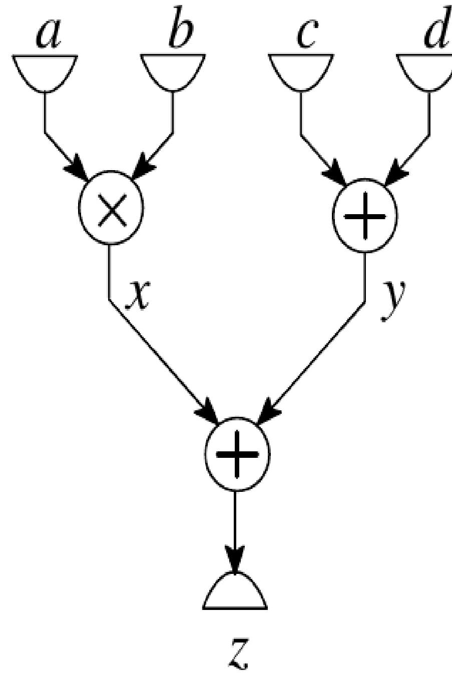
Data Path + Control Structure

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

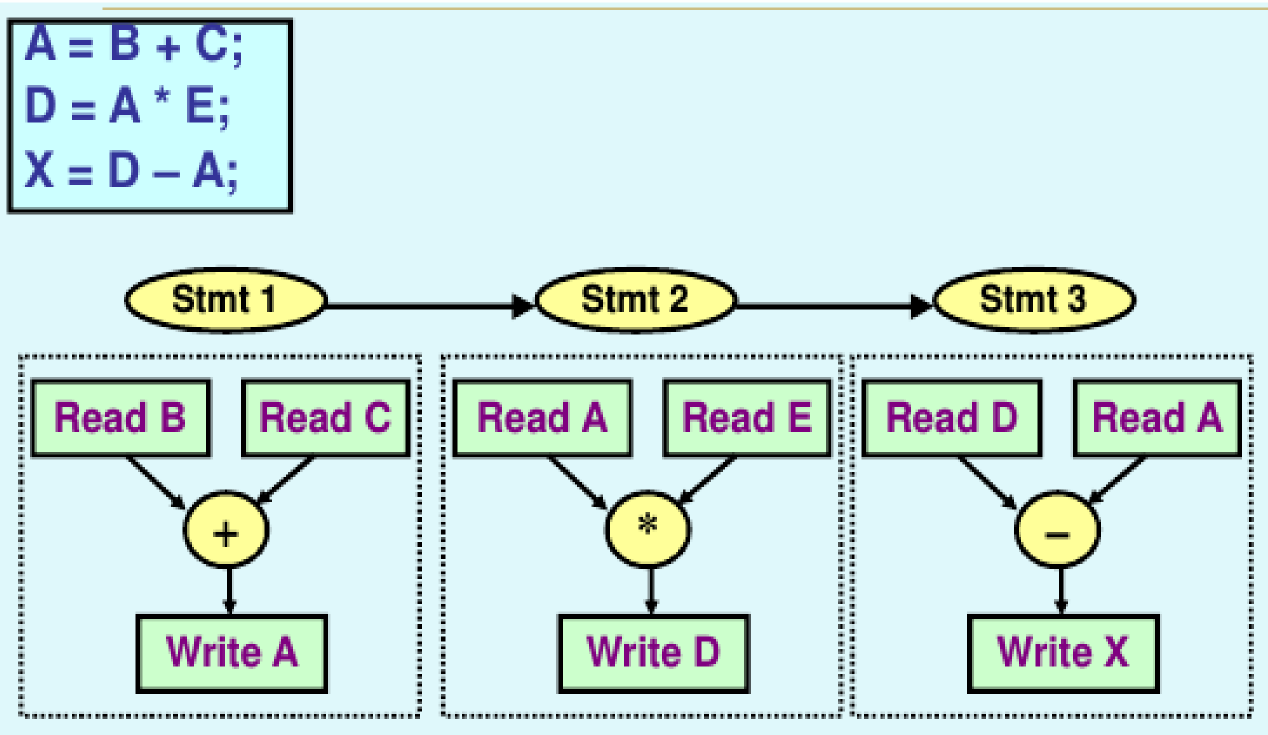
Control: the part of the hardware that takes care of having the data present at the right place at a specific time, of presenting the right instructions to a programmable unit, etc.

Data-flow Graph

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

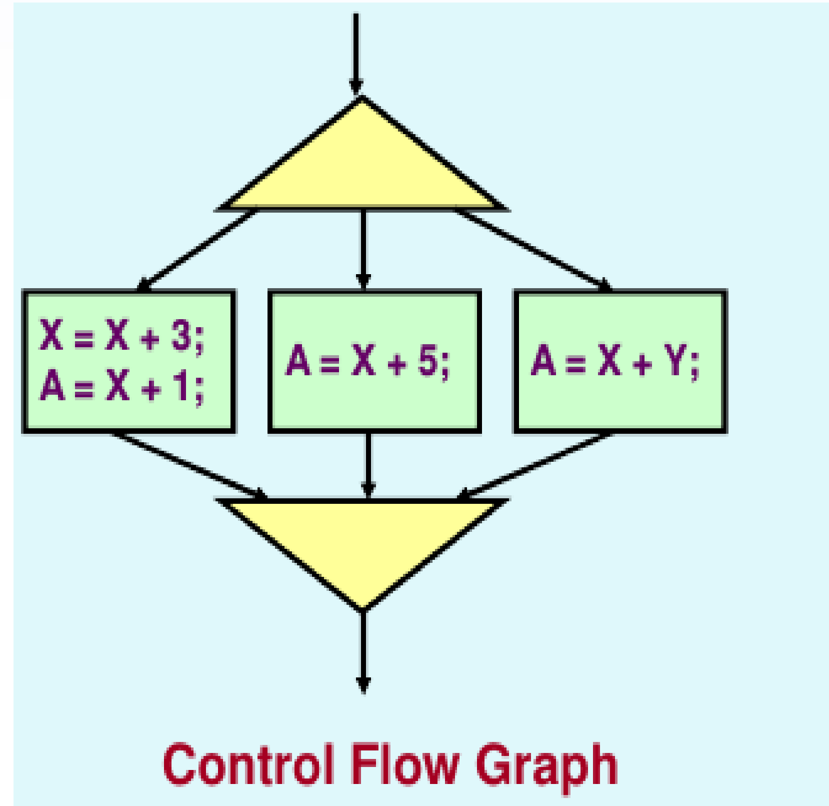


Data-flow Graphs



Control-flow Graph

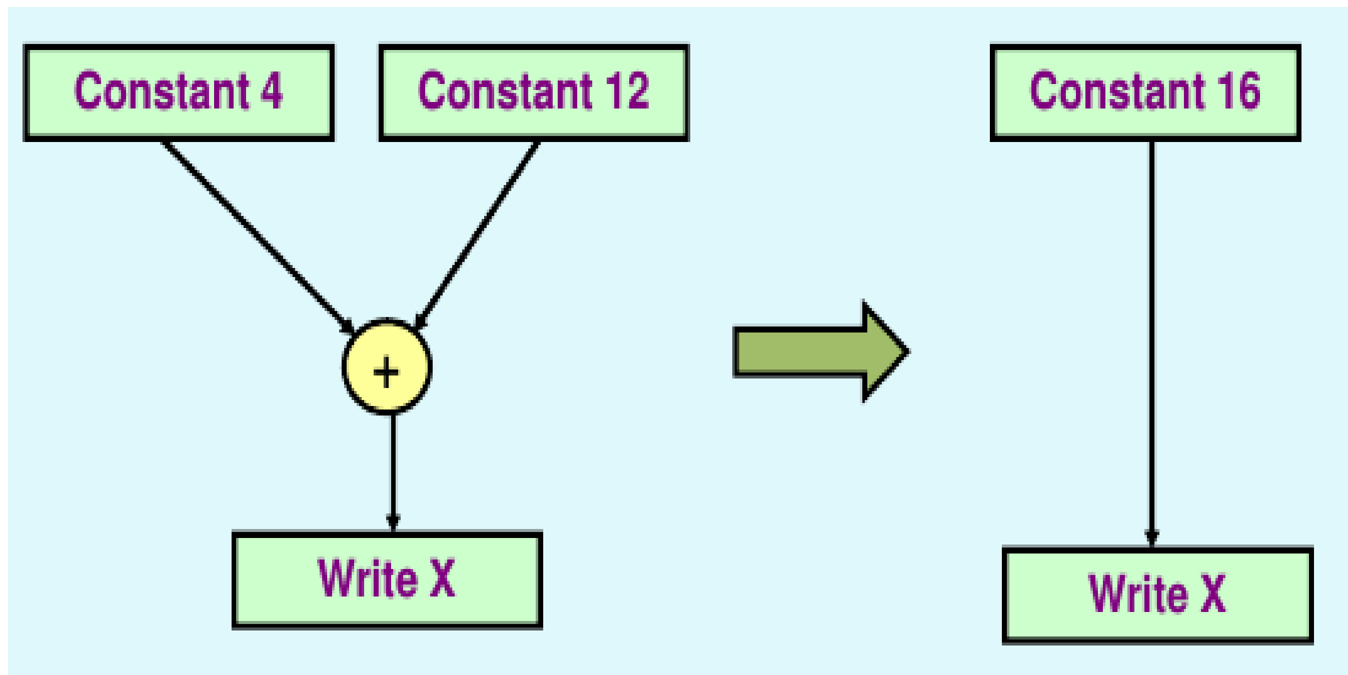
```
case (C)
1:
begin
X = X + 3;
A = X + 1;
end
2:
A = X + 5;
default:
A = X + Y;
endcase
```



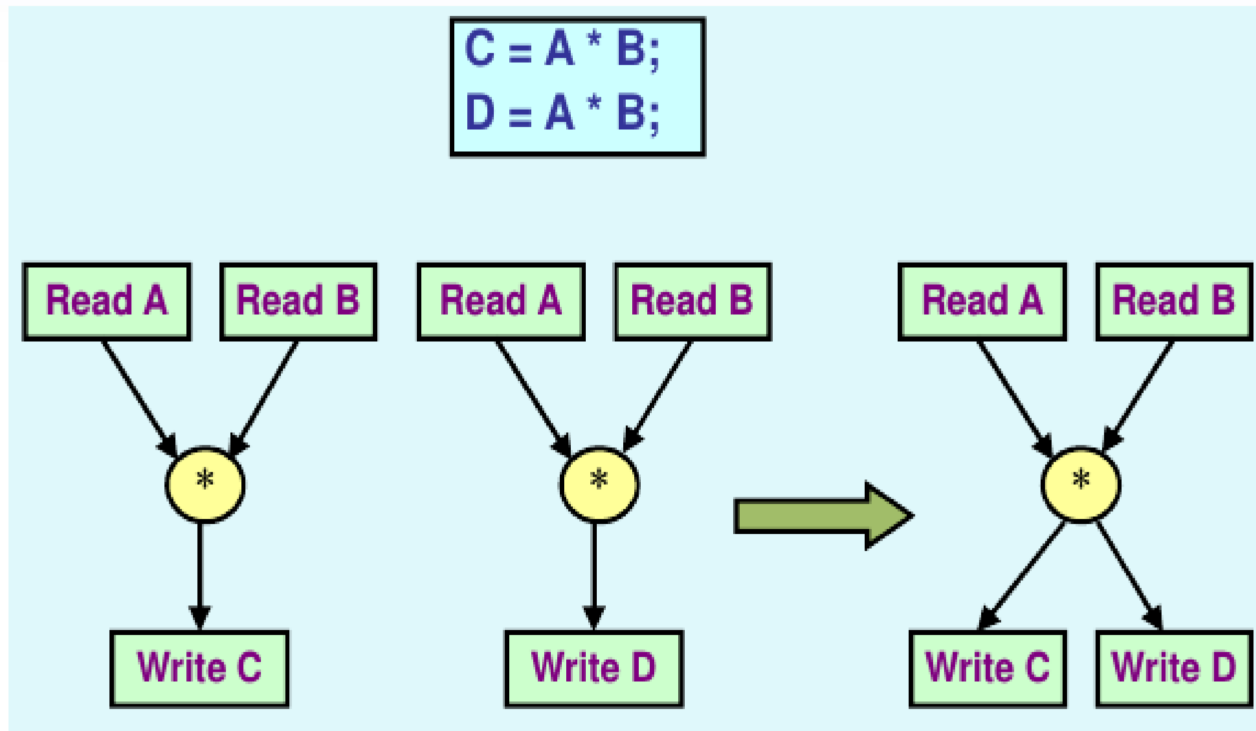
HLS Compiler Transformation

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

Constant Folding

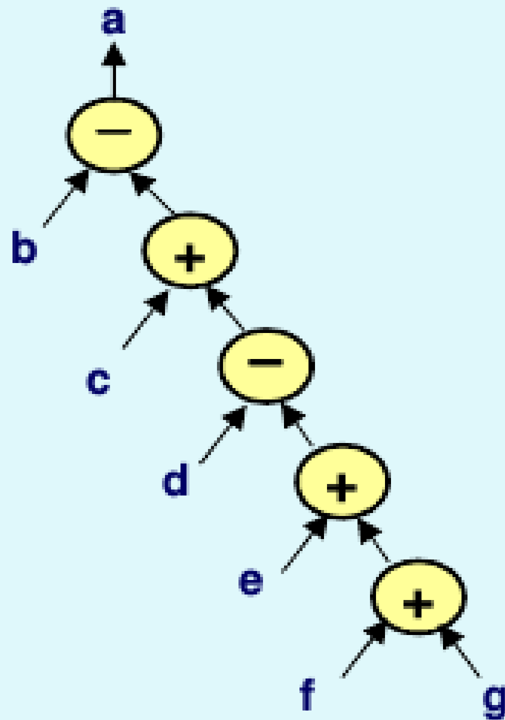


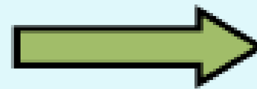
Redundant operator elimination

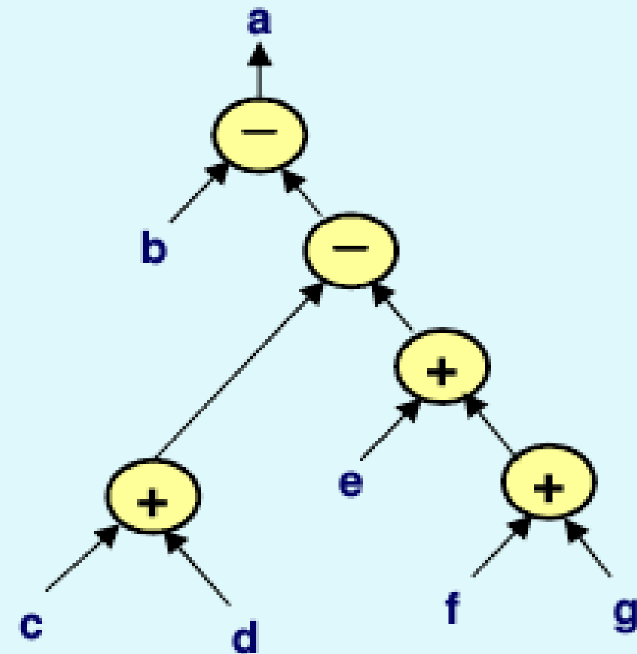


Tree height transformation

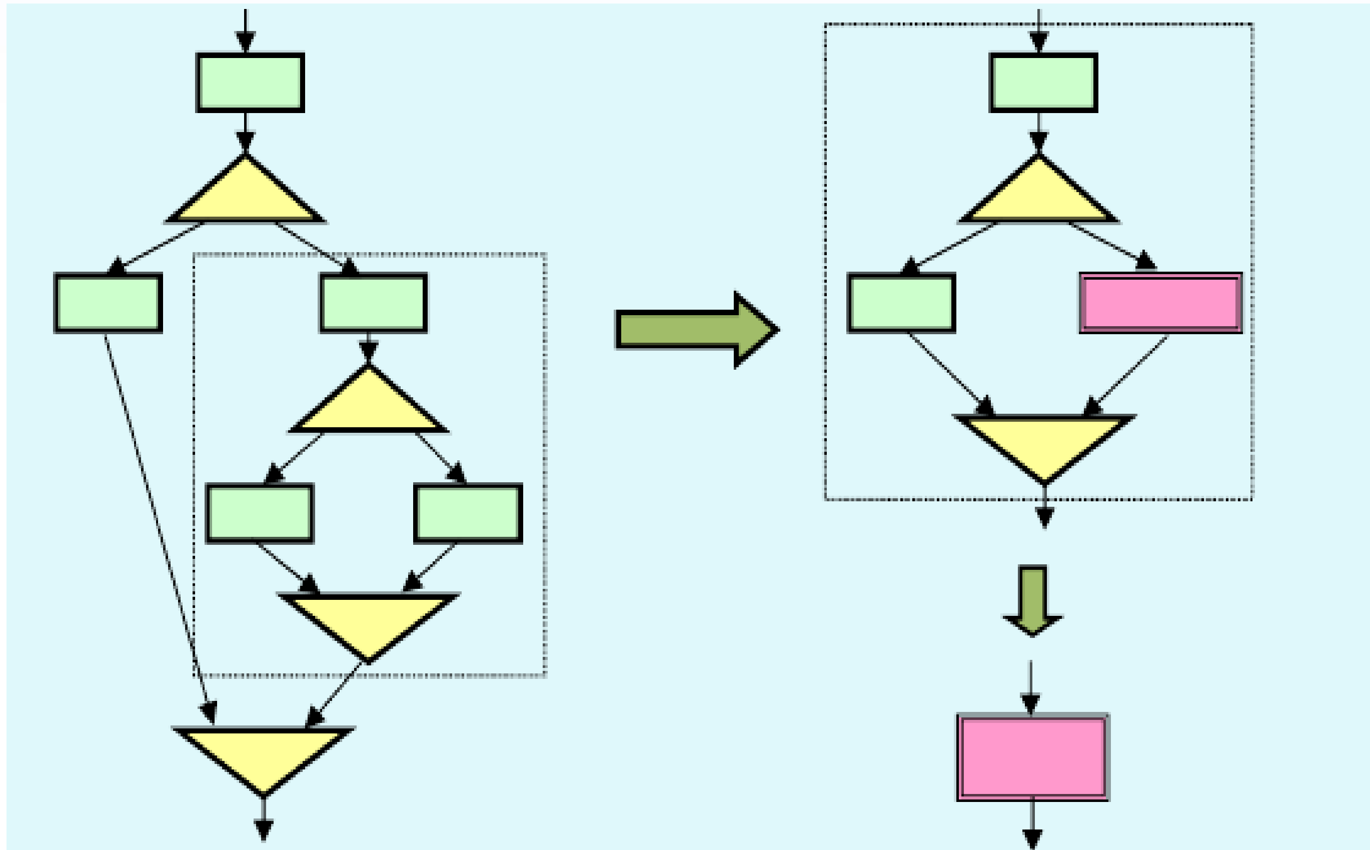
$a = b - c + d - e + f + g$



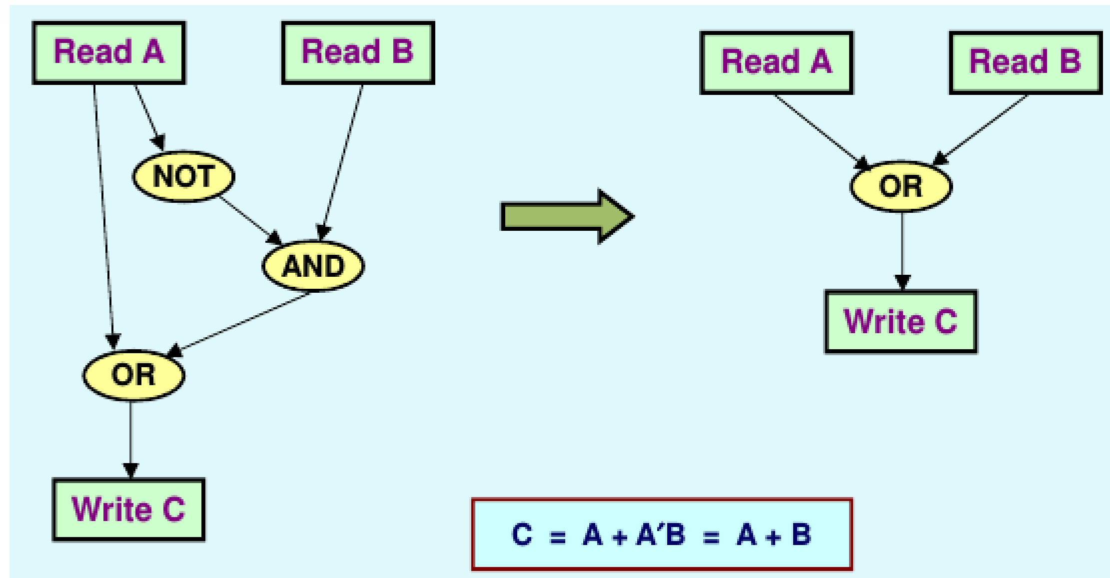




Control flattening

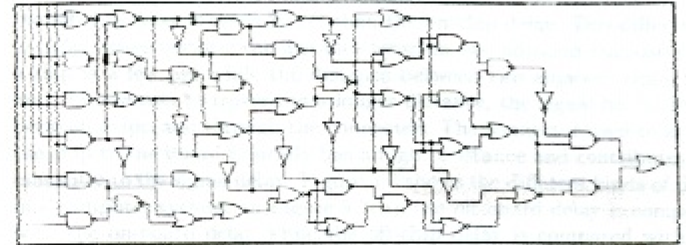


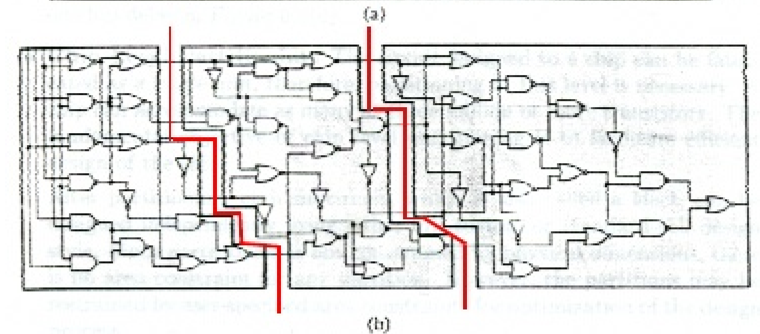
Logic Level Transformation



Partitioning

- Scheduling
- Allocation
- Unit selection



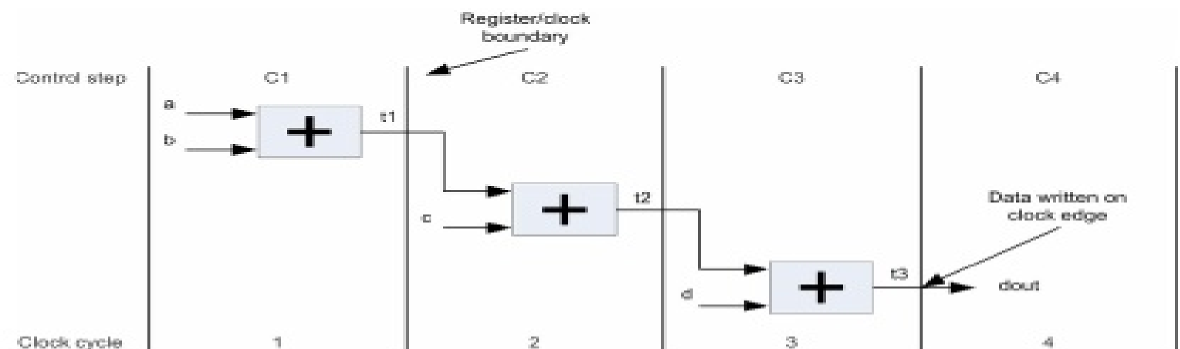


Loop Unrolling

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:

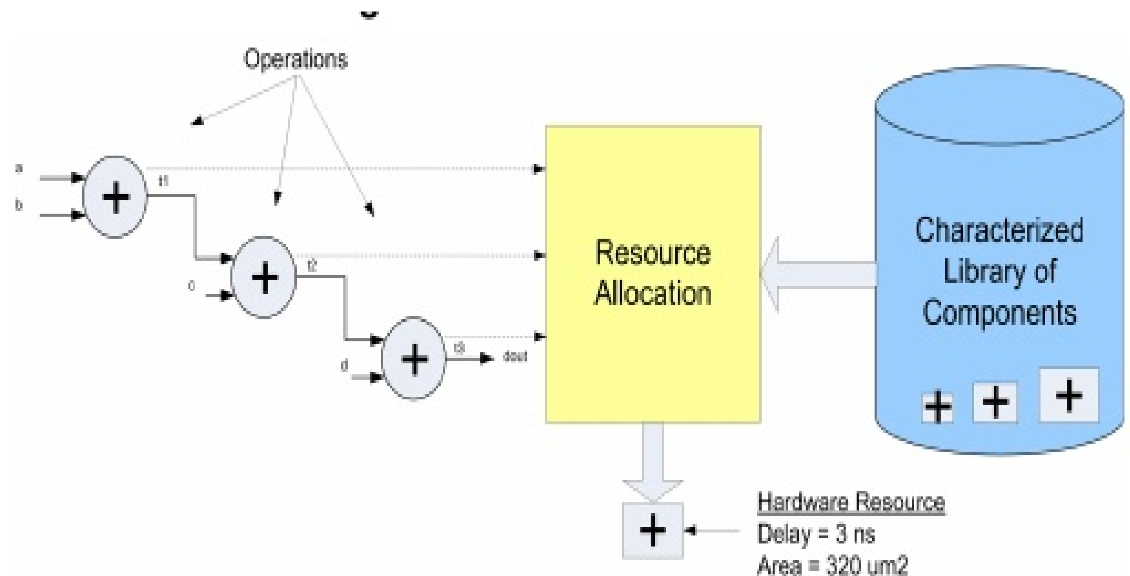
– As Soon As Possible (ASAP)

– As Late As Possible (ALAP)

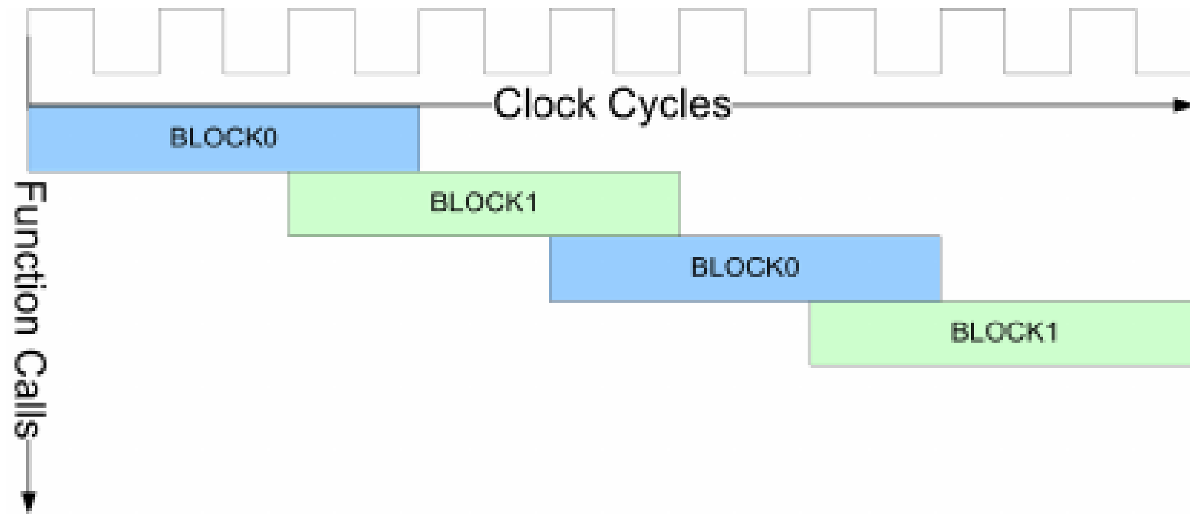
– Resource Constrained (List scheduling)

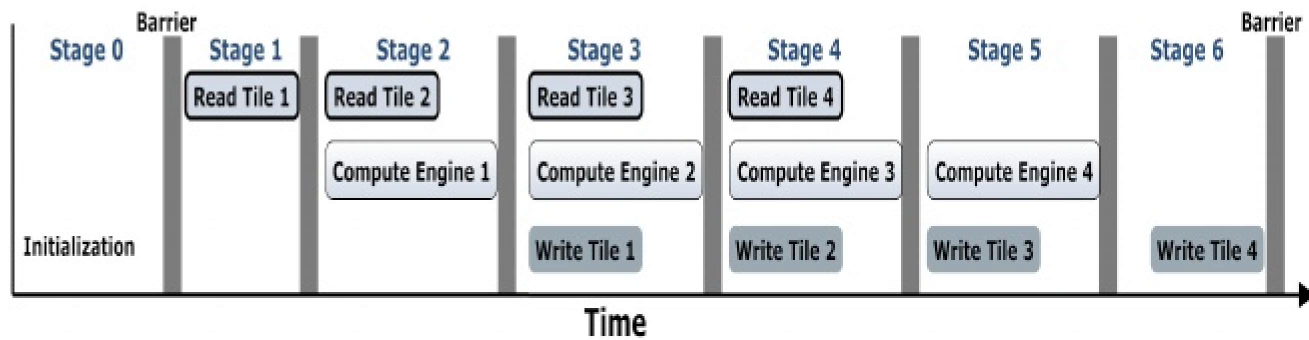
Resource Allocation

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.

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 ;
  }

}
```

Vivado HLS Example

```
// original, non-optimized version of FIR
#define SIZE 128
#define N 10
void fir(int x[SIZE], int y[SIZE]) {
// FIR coefficients
int coeff[N] = {13, -2, 9, 11, 26, 18, 95, -43, 6, 74};
// exact translation from FIR formula above
for (int n = 0; n < SIZE; n++) {
int acc = 0;
for (int i = 0; i < N; i++ ) {
if (n - i >= 0)
acc += coeff[i] * x[n - i];
}
y[n] = acc;
}
}
```

Matrix Multiplication

```c
#define N 3  // Size of the matrix (N x N)

// Function to multiply two matrices
void mat_mul(const int A[N][N], const int B[N][N], int C[N][N]) {

    // Matrix multiplication
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            int sum = 0;
            for (int k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}
```

Sobel Filter

```
void image_filter(int img_in[512][512], int img_out[512][512]) {
    int Gx[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};
    int Gy[3][3] = {{-1, -2, -1}, {0, 0, 0}, {1, 2, 1}};

    #pragma HLS ARRAY_PARTITION variable=Gx complete dim=0
    #pragma HLS ARRAY_PARTITION variable=Gy complete dim=0

    for (int i = 1; i < 511; i++) {
        for (int j = 1; j < 511; j++) {
            int sumX = 0, sumY = 0;
            for (int k = -1; k <= 1; k++) {
                for (int l = -1; l <= 1; l++) {
                    sumX += img_in[i+k][j+l] * Gx[k+1][l+1];
                    sumY += img_in[i+k][j+l] * Gy[k+1][l+1];
                }
            }
            img_out[i][j] = sqrt(sumX*sumX + sumY*sumY);
        }
    }
}
```

I2C Communication

```
#include <hls_stream.h>
#include <ap_int.h>

#define SDA_HIGH 1
#define SDA_LOW  0
#define SCL_HIGH 1
#define SCL_LOW  0
void i2c_main(
volatile bool *sda,     // Data line
volatile bool *scl,     // Clock line
unsigned char address,  // I2C slave address
unsigned char data,     // Data to send
volatile bool *done,    // Operation done flag
bool start              // Start signal
) {
  #pragma HLS INTERFACE ap_none port=sda
    #pragma HLS INTERFACE ap_none port=scl
    #pragma HLS INTERFACE ap_none port=address
    #pragma HLS INTERFACE ap_none port=data
    #pragma HLS INTERFACE ap_none port=done
    #pragma HLS INTERFACE ap_none port=start
    #pragma HLS INTERFACE ap_ctrl_none
     port=return
```

```
    static enum {IDLE, START, ADDR,
    DATA, STOP} state = IDLE;

    static unsigned char bit_counter =
    0;

    static unsigned char shift_reg = 0;
    switch(state) {
        case IDLE:
            *scl = SCL_HIGH;
            *sda = SDA_HIGH;
            if (start && !(*done)) {
                state = START;
            }
            break;
        case START:
*sda = SDA_LOW;  // Start condition:
    SDA goes low while SCL is high
            state = ADDR;
            shift_reg = address <<
    1; // Shift left to make room for
    R/W bit
            bit_counter = 0;
            break;
```

```
case ADDR:
            if (bit_counter < 8) {
                *sda = (shift_reg &
     0x80) ? SDA_HIGH : SDA_LOW;
                shift_reg <<= 1;
                *scl = SCL_LOW;
                *scl = SCL_HIGH;
                bit_counter++;
            }
else {
                state = DATA;
                shift_reg = data;
                bit_counter = 0;
            }
            break;
        case DATA:
            if (bit_counter < 8) {
                *sda = (shift_reg &
     0x80) ? SDA_HIGH : SDA_LOW;
                shift_reg <<= 1;
                *scl = SCL_LOW;
                *scl = SCL_HIGH;
                bit_counter++;
            } else {
                state = STOP;
            }
            break;
        case STOP:
            *sda = SDA_LOW;
            *scl = SCL_HIGH;
            *sda = SDA_HIGH;  // Stop
     condition: SDA goes high while
     SCL is high
            *done = true;
            state = IDLE;
            break;
     } }
```

Results

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	5.32	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1	1	1	1	none

Detail

- Instance
- Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	15
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	141
Register	-	-	30	-
Total	0	0	30	156
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_none	i2c_main	return value
ap_rst	in	1	ap_ctrl_none	i2c_main	return value
sda	out	1	ap_none	sda	pointer
scl	out	1	ap_none	scl	pointer
address	in	8	ap_none	address	scalar
data	in	8	ap_none	data	scalar
done_i	in	1	ap_none	done	pointer
done_o	out	1	ap_none	done	pointer
start	in	1	ap_none	start	scalar

Export the report(.html) using the Export Wizard

Open Analysis Perspective Analysis Perspective

Tasks

What is the difference between two approaches of System Design

a) RTL Sequential Code Architecture

b) HLS Architecture

Install Xilinx Vitis and Vivado HLS

Write few examples and check their behaviours