SPARK: A Parallelizing High-Level Synthesis Framework

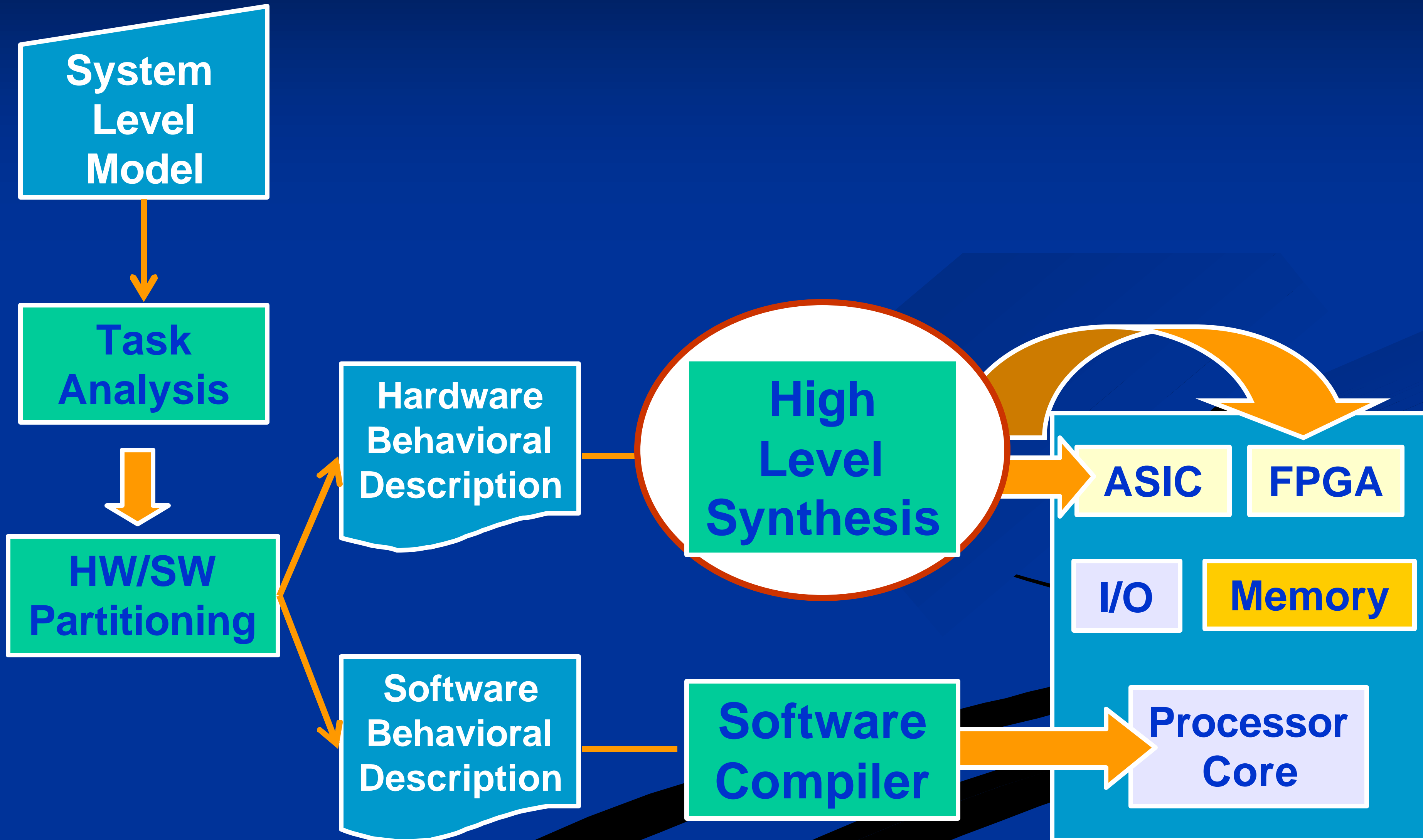
Sumit Gupta

Rajesh Gupta, Nikil Dutt, Alex Nicolau

Center for Embedded Computer Systems
University of California, Irvine and San Diego
http://www.cecs.uci.edu/~spark

Supported by Semiconductor Research Corporation & Intel Inc

System Level Synthesis

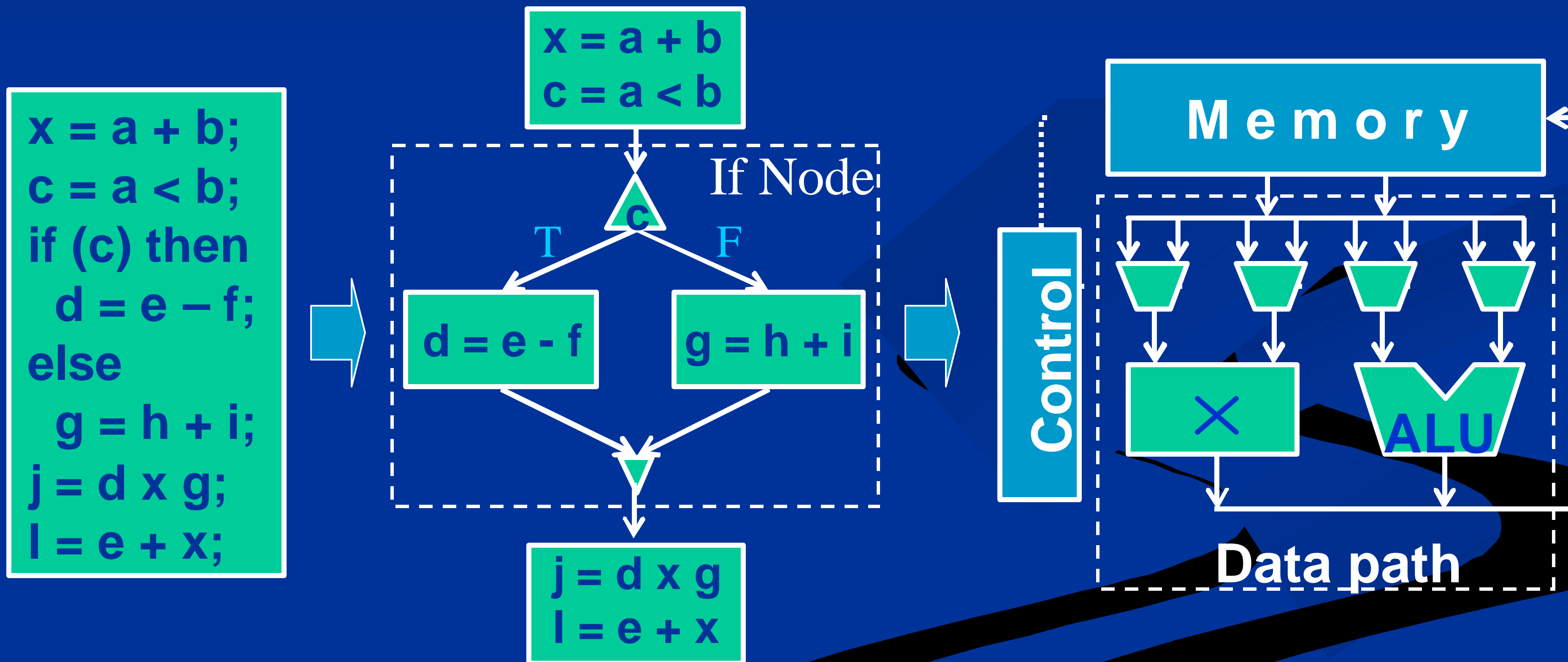


Copyright Sumit Gupta 2003

High Level Synthesis

- Transform behavioral descriptions to RTL/gate level

From C to CDFG to Architecture

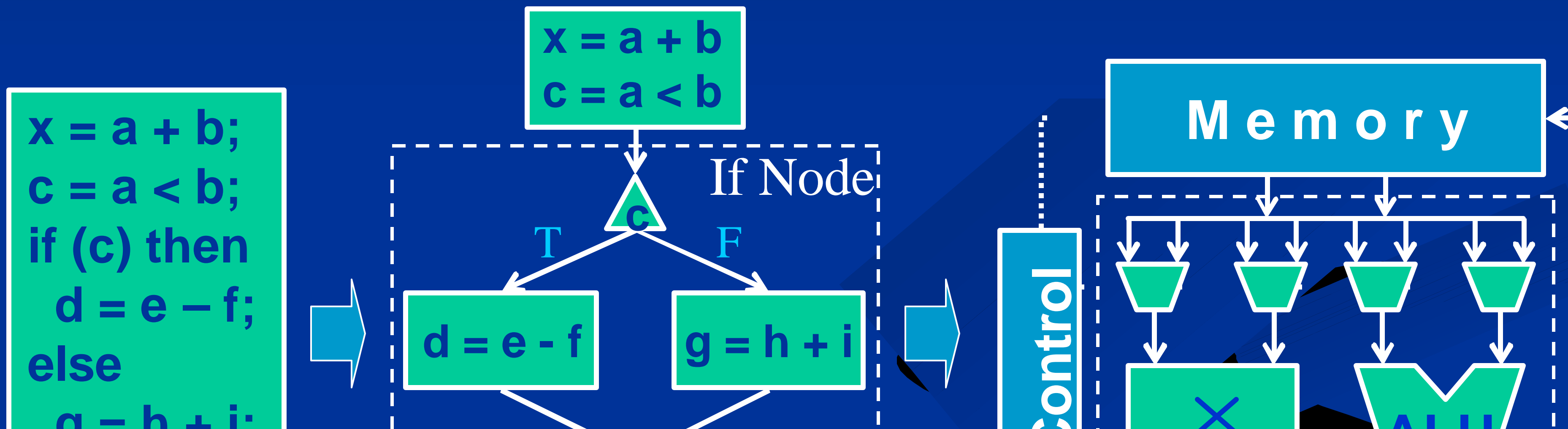


Copyright Sumit Gupta 2003

High Level Synthesis

- Transform behavioral descriptions to RTL/gate level

From C to CDFG to Architecture



Problem # 1 : Poor quality of HLS results beyond straight-line behavioral descriptions

Problem # 2 : Poor/No controllability of the HLS results

Copyright Sumit Gupta 2003

Outline

- Motivation and Background
- Our Approach to *Parallelizing* High-Level Synthesis
- Code Transformations Techniques for PHLS
  - Parallelizing Transformations
  - Dynamic Transformations
- The PHLS Framework and Experimental Results
  - Multimedia and Image Processing Applications
  - Case Study: Intel Instruction Length Decoder
- Conclusions and Future Work

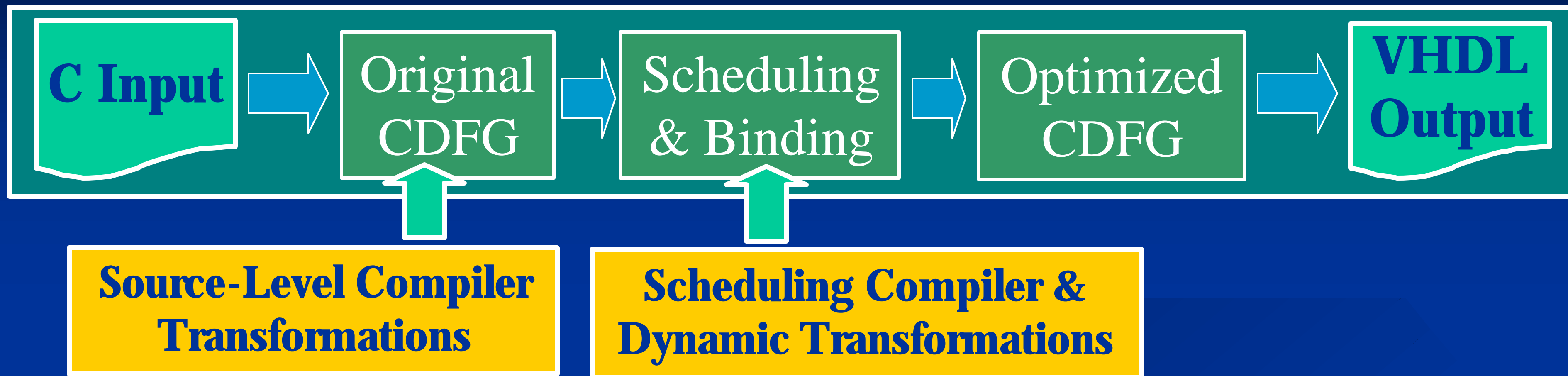
Copyright Sumit Gupta 2003

High-level Synthesis

- Well-researched area: from early 1980's
  - Renewed interest due to new system level design methodologies
- Large number of synthesis optimizations have been proposed
  - Either operation level: algebraic transformations on DSP codes
  - or logic level: Don't Care based control optimizations
  - In contrast, compiler transformations operate at both operation level (fine-grain) and source level (coarse-grain)
- Parallelizing Compiler Transformations
  - Different optimization objectives and cost models than HLS
- Our aim: Develop Synthesis and Parallelizing Compiler Transformations that are "useful" for HLS
  - Beyond scheduling results: in Circuit Area and Delay
  - For large designs with complex control flow (nested conditionals/loops)

Copyright Sumit Gupta 2003

Our Approach: Parallelizing HLS (PHLS)



- Optimizing Compiler and Parallelizing Compiler transformations applied at Source-level (Pre-synthesis) and during Scheduling
  - Source-level code refinement using Pre-synthesis transformations
  - Code Restructuring by Speculative Code Motions
  - Operation replication to improve concurrency
  - Dynamic transformations: exploit new opportunities during scheduling

Copyright Sumit Gupta 2003

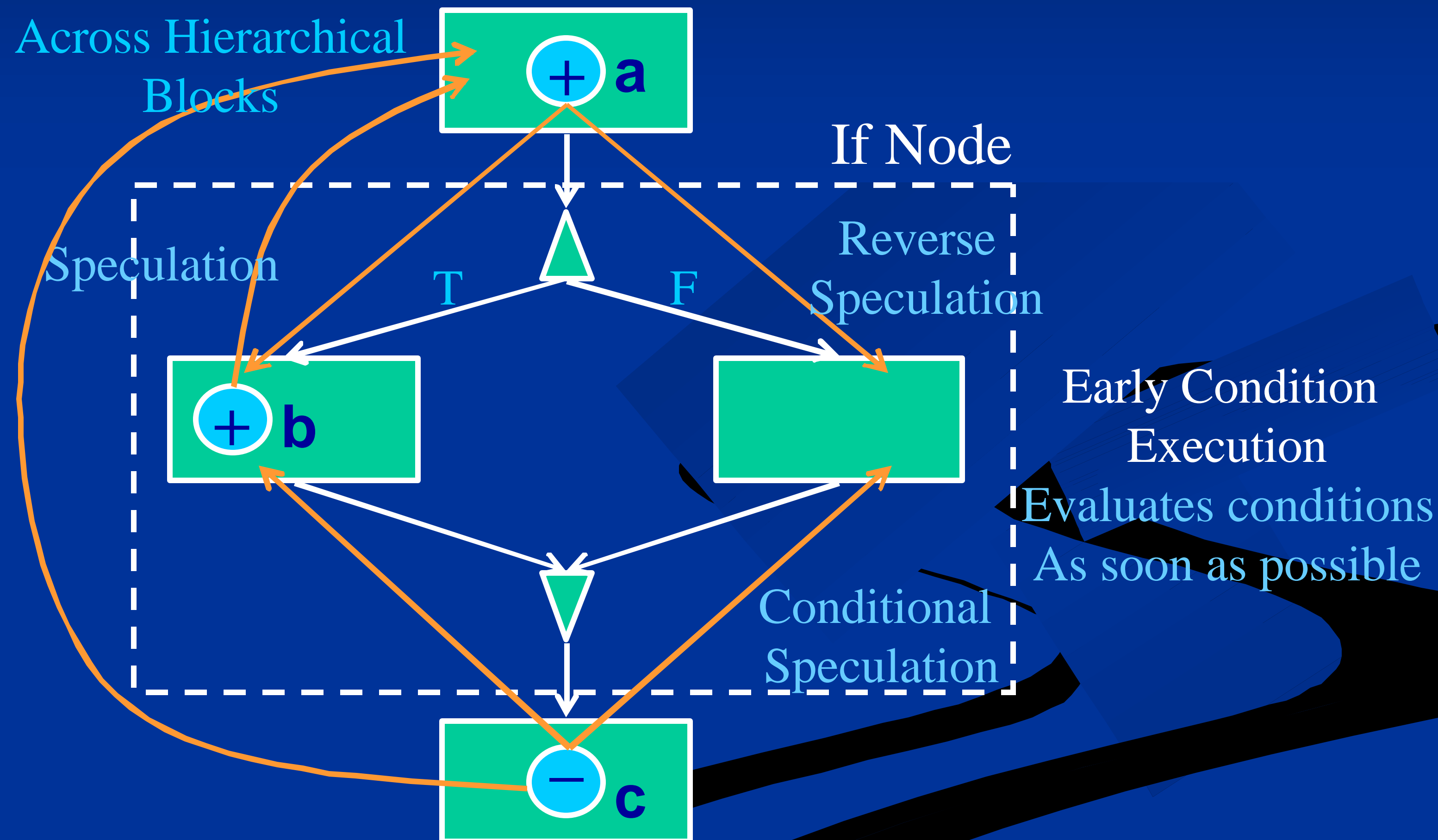
PHLS Transformations Organized into Four Groups

1. Pre-synthesis: Loop-invariant code motions, Loop unrolling, CSE
2. Scheduling: Speculative Code Motions, Multi-cycling, Operation Chaining, Loop Pipelining
3. Dynamic: Transformations applied dynamically during scheduling: Dynamic CSE, Dynamic Copy Propagation, Dynamic Branch Balancing
4. Basic Compiler Transformations: Copy Propagation, Dead Code Elimination

Copyright Sumit Gupta 2003

Speculative Code Motions

Operation Movement to reduce impact of Programming Style on Quality of HLS Results

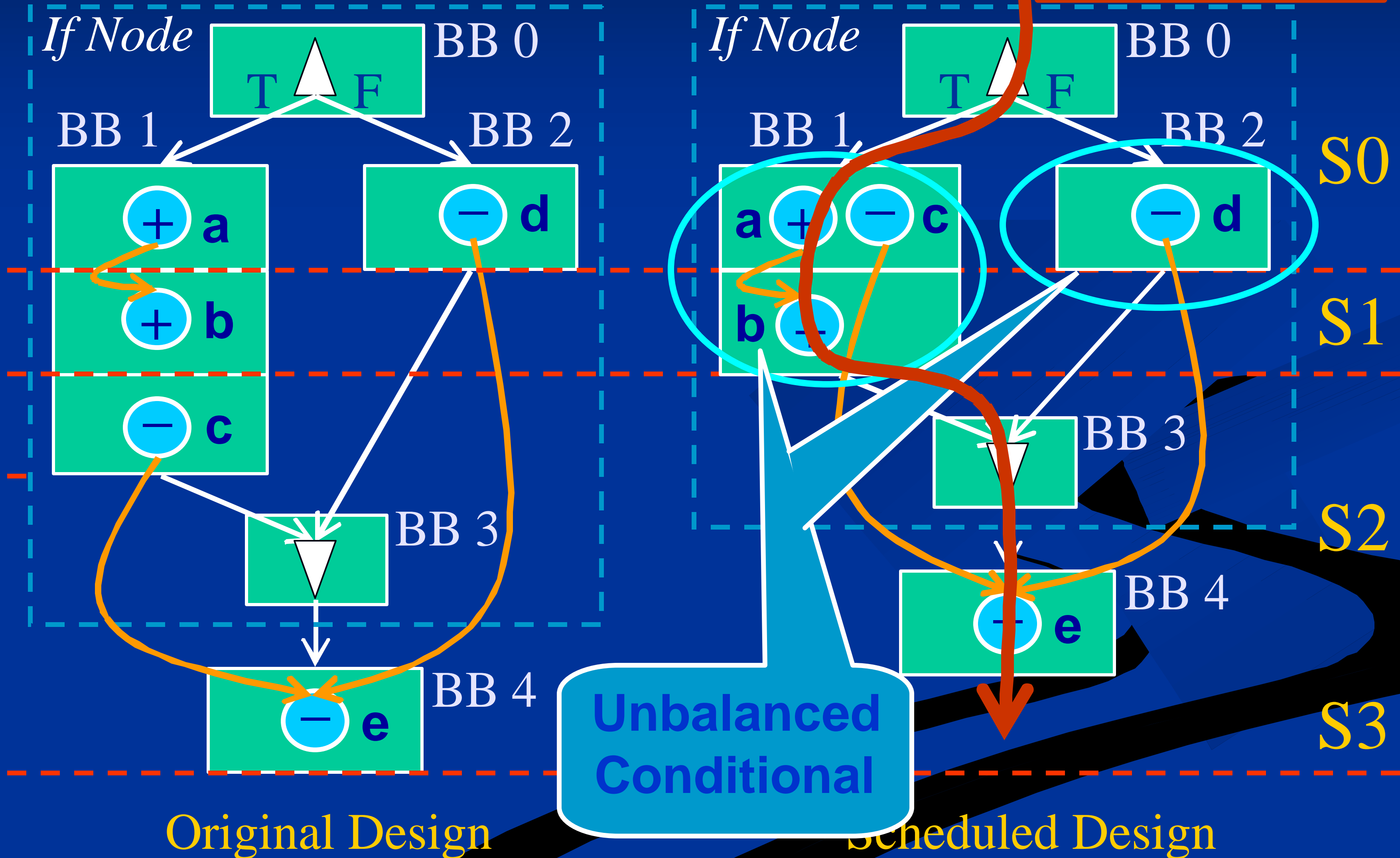


Dynamic Transformations

- Called "dynamic" since they are applied during scheduling (versus a pass before/after scheduling)
- Dynamic Branch Balancing
  - Increase the scope of code motions
  - Reduce impact of programming style on HLS results
- Dynamic CSE and Dynamic Copy Propagation
  - Exploit the Operation movement and duplication due to speculative code motions
    - Create new opportunities to apply these transformations
  - Reduce the number of operations

Copyright Sumit Gupta 2003

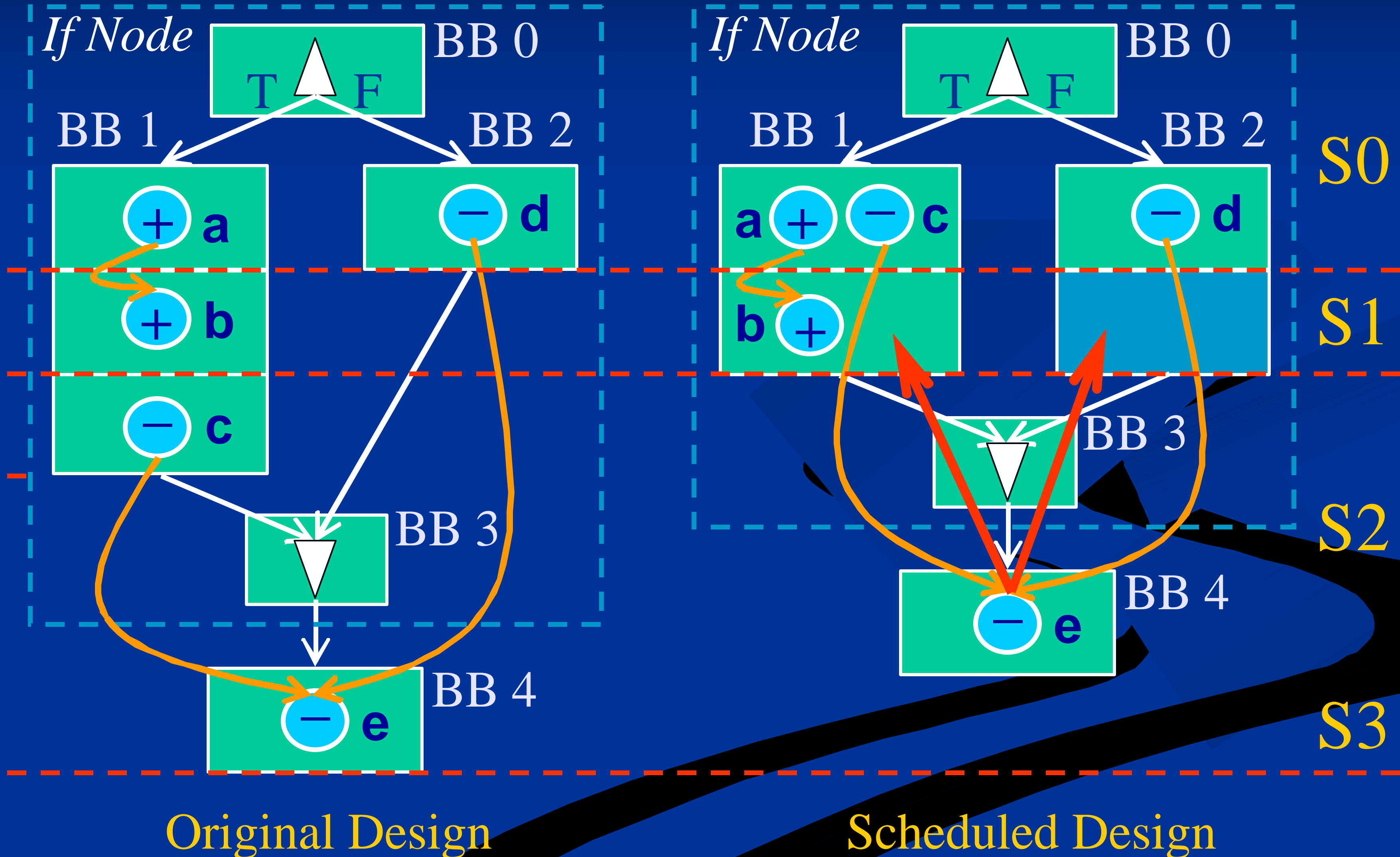
Dynamic Branch Balancing



Copyright Sumit Gupta 2003

Insert New Scheduling Step in Shorter Branch

Resource Allocation



Copyright Sumit Gupta 2003

Insert New Scheduling Step in Shorter Branch

Resource Allocation

Dynamic Branch Balancing inserts new scheduling steps

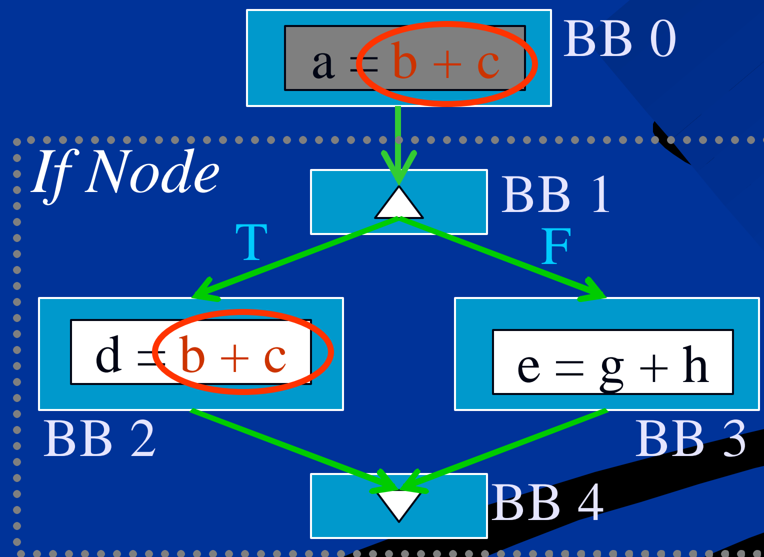
- Enables Conditional Speculation
- Leads to further code compaction

Copyright Sumit Gupta 2003

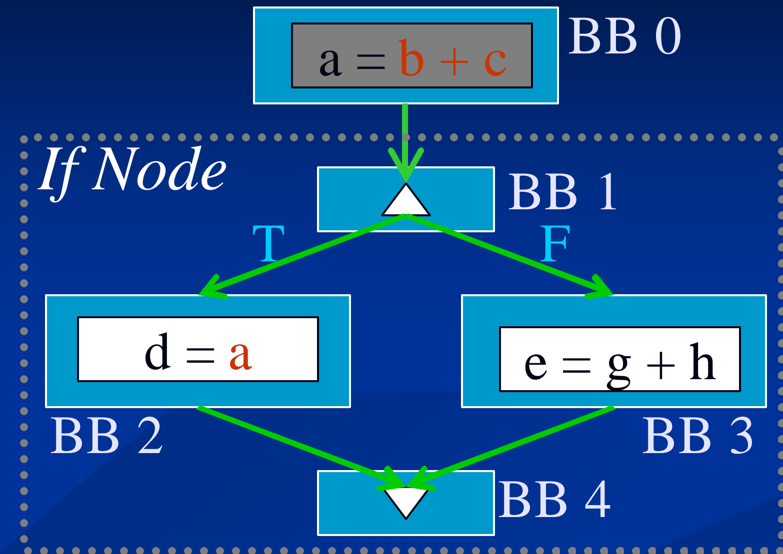
Dynamic CSE: Going beyond Traditional CSE

```
a = b + c;
cd = b < c;
if (cd)
  d = b + c;
else
  e = g + h;
```

C Description



HTG Representation

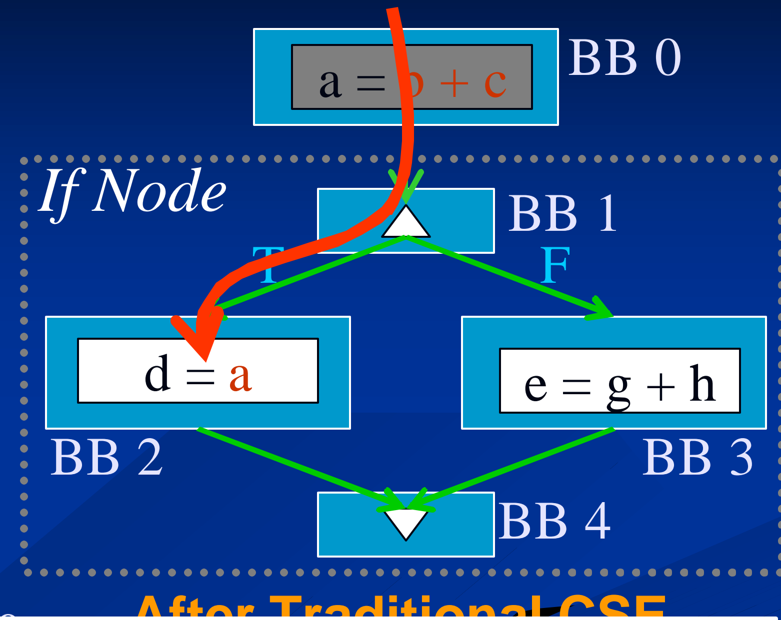


After Traditional CSE

Copyright Sumit Gupta 2003

Dynamic CSE: Going beyond Traditional CSE

```
a = b + c;
cd = b < c;
if (cd)
  d = b + c;
else
  e = g + h;
```

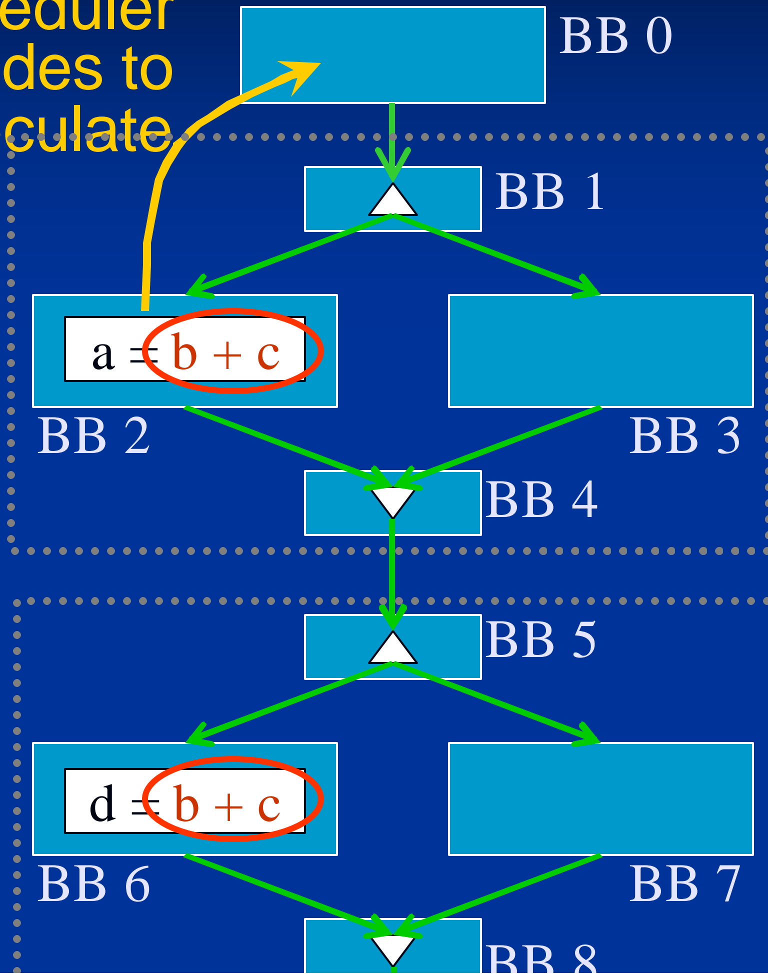


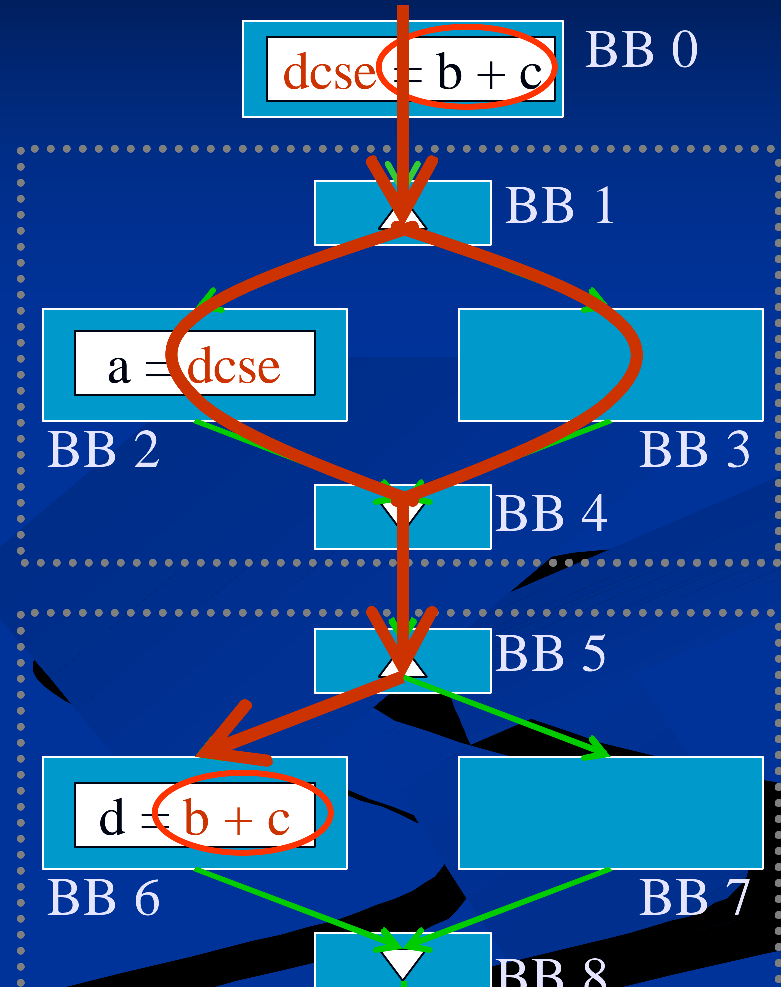
- We use notion of Dominance of Basic Blocks
  - Basic block BBi dominates BBj if all control paths from the initial basic block of the design graph leading to BBj goes through BBi
- We can eliminate an operation opj in BBj using common expression in opi if BBi dominates BBj

Copyright Sumit Gupta 2003

New Opportunities for "Dynamic" CSE Due to Code Motions

Scheduler decides to Speculate

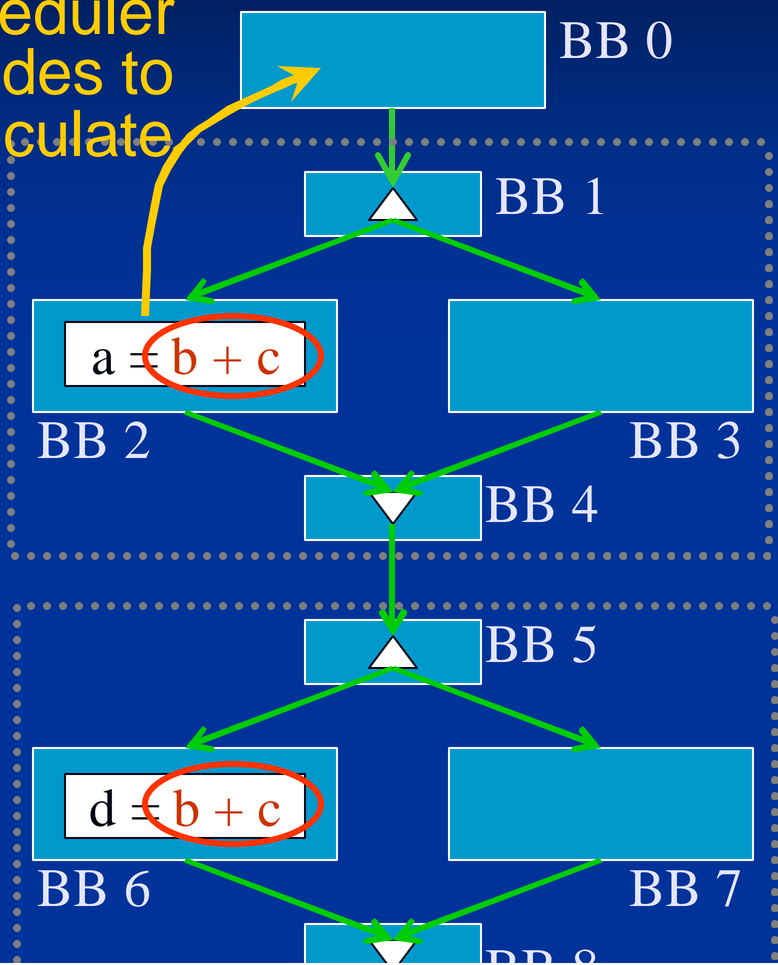


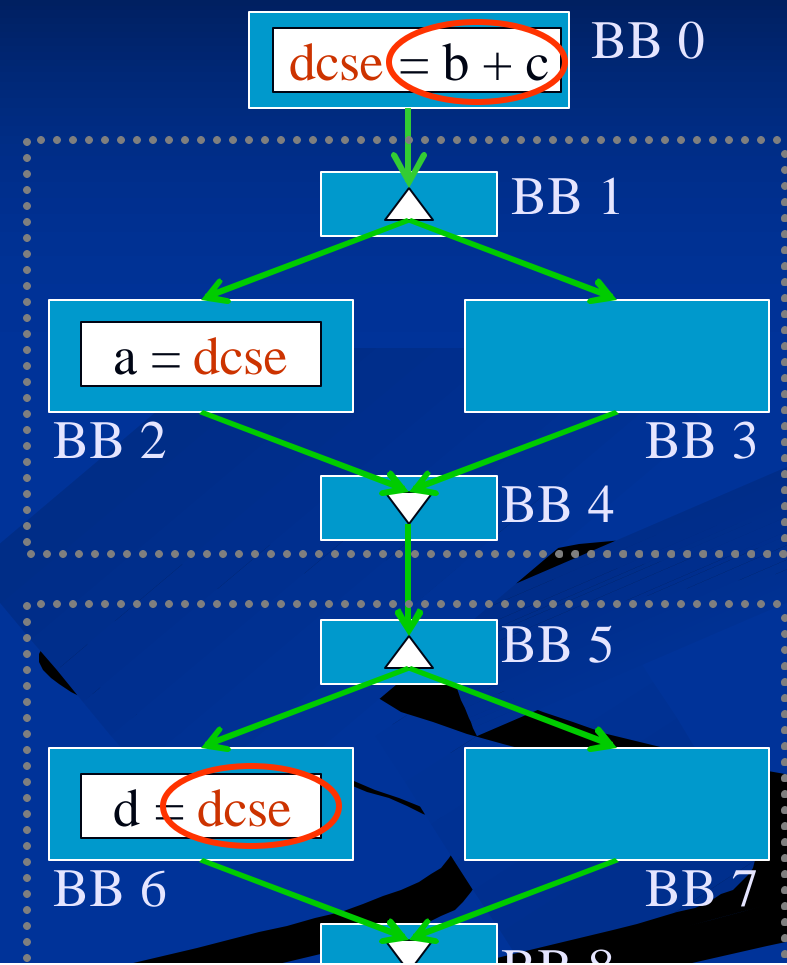


CSE not possible since BB2 does not dominate BB6

CSE possible now since BB0 does not dominate BB6

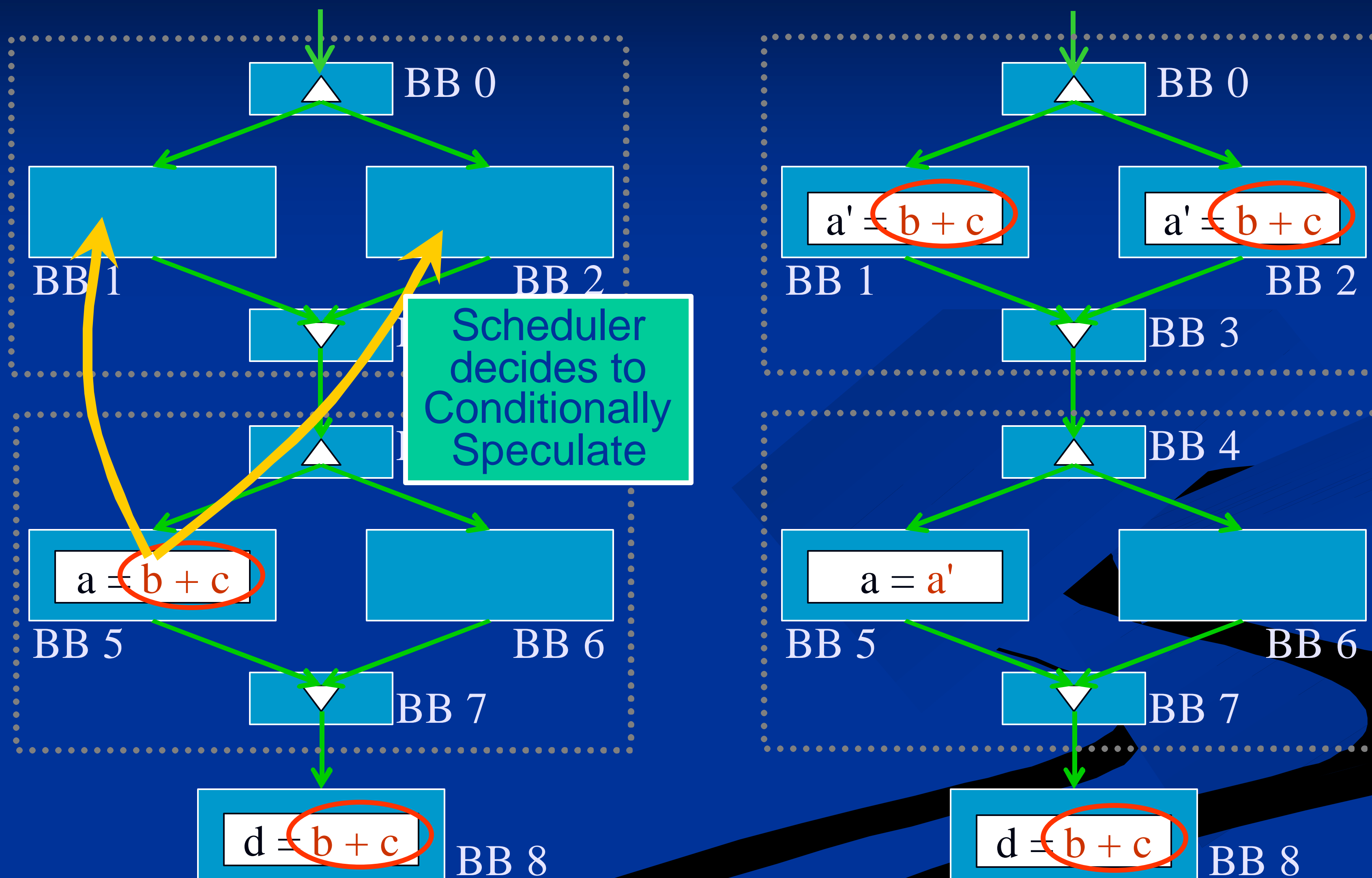
New Opportunities for "Dynamic" CSE Due to Code Motions





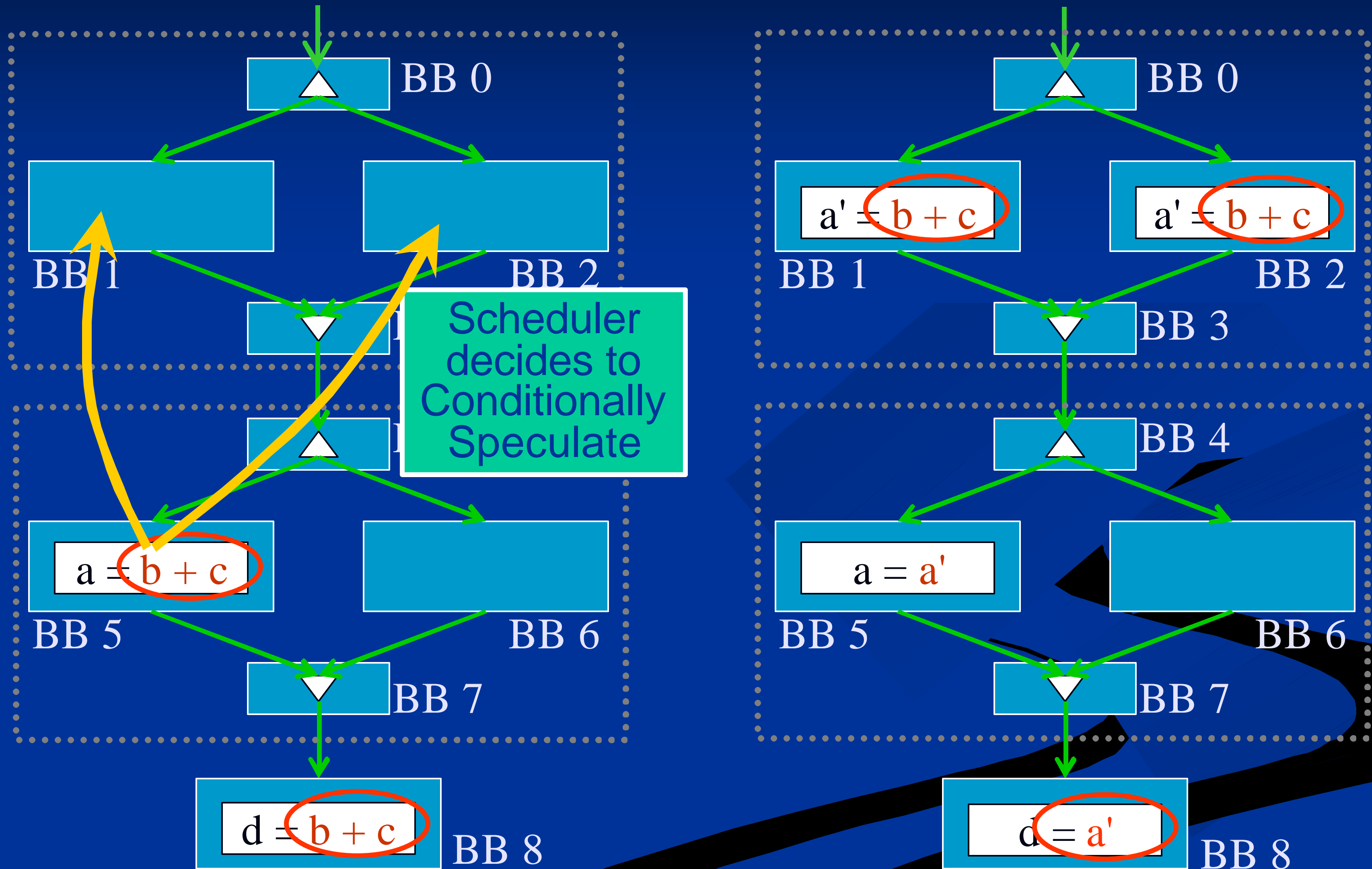
If scheduler moves or duplicates an operation op, apply CSE on remaining operations using op

Condition Speculation & Dynamic CSE



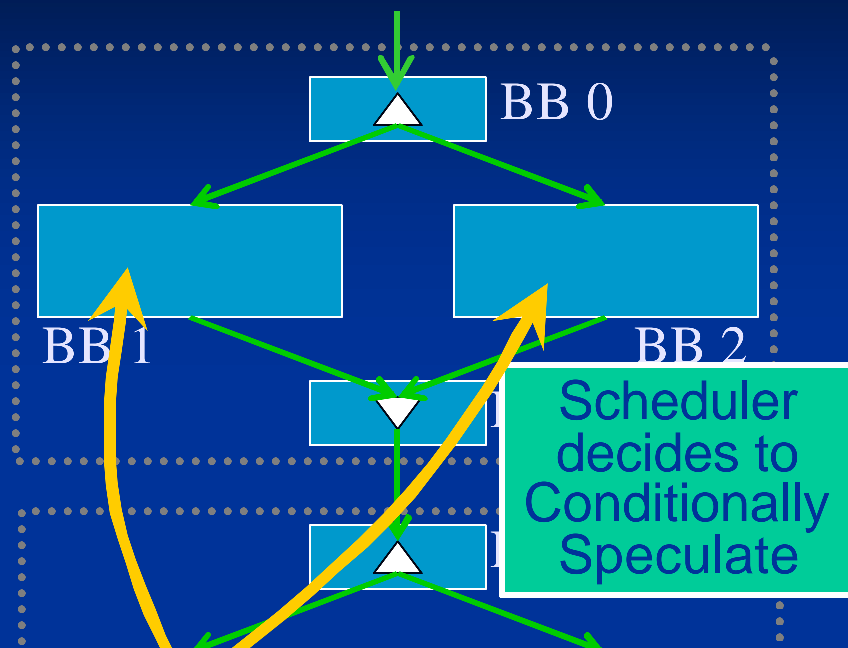
Copyright Sumit Gupta 2003

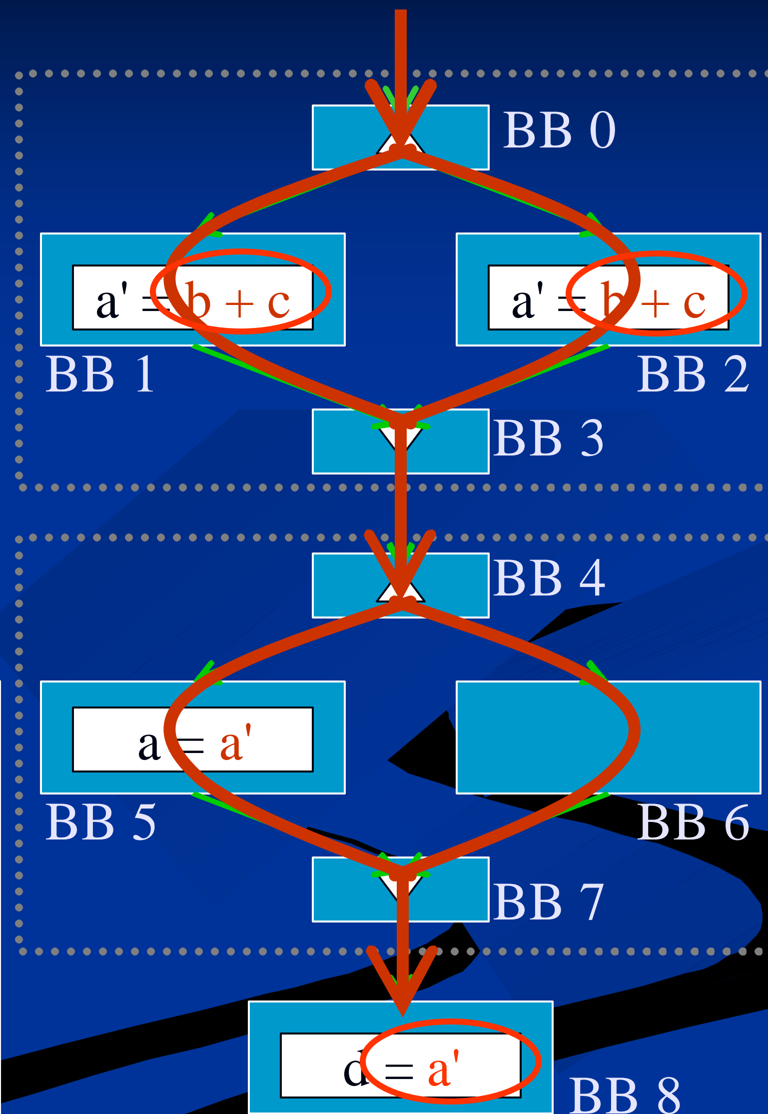
Condition Speculation & Dynamic CSE



Copyright Sumit Gupta 2003

Condition Speculation & Dynamic CSE

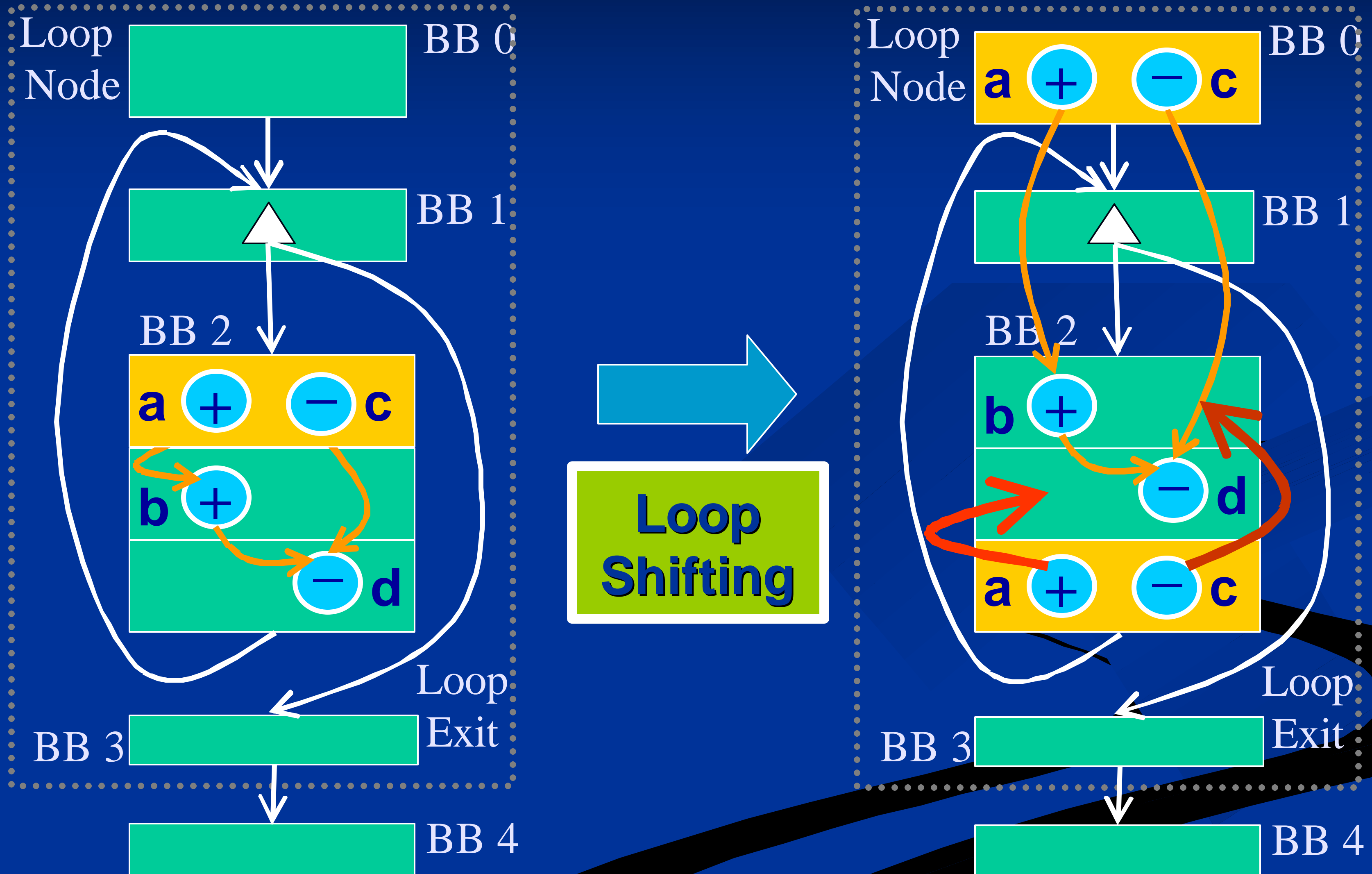




- Use the notion of dominance by groups of basic blocks
  - **All Control Paths leading up to BB8 come from either BB1 or BB2:** => BB1 and BB2 **together dominate** BB8

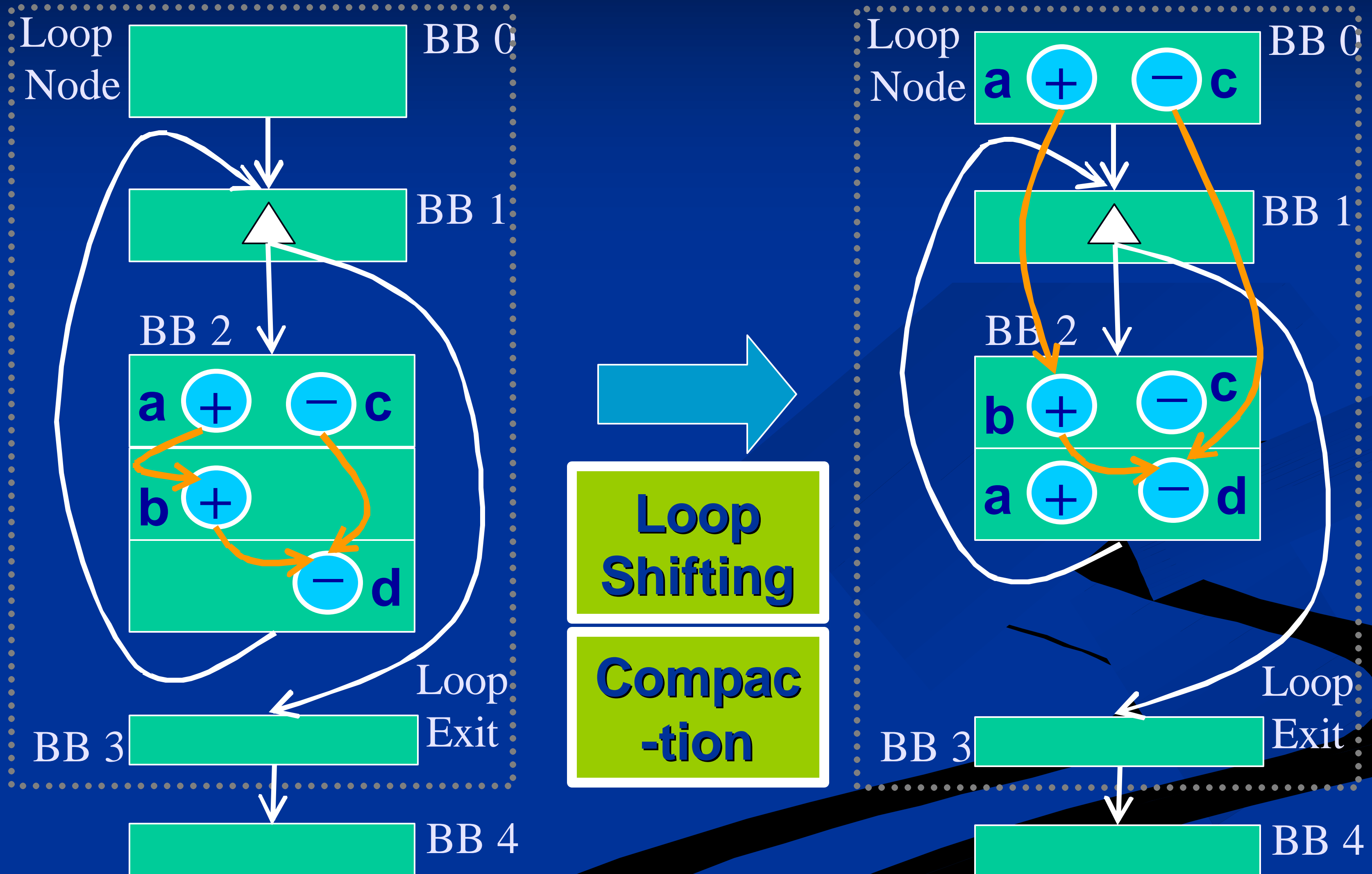
Copyright Sumit Gupta 2003

Loop Shifting: An Incremental Loop Pipelining Technique

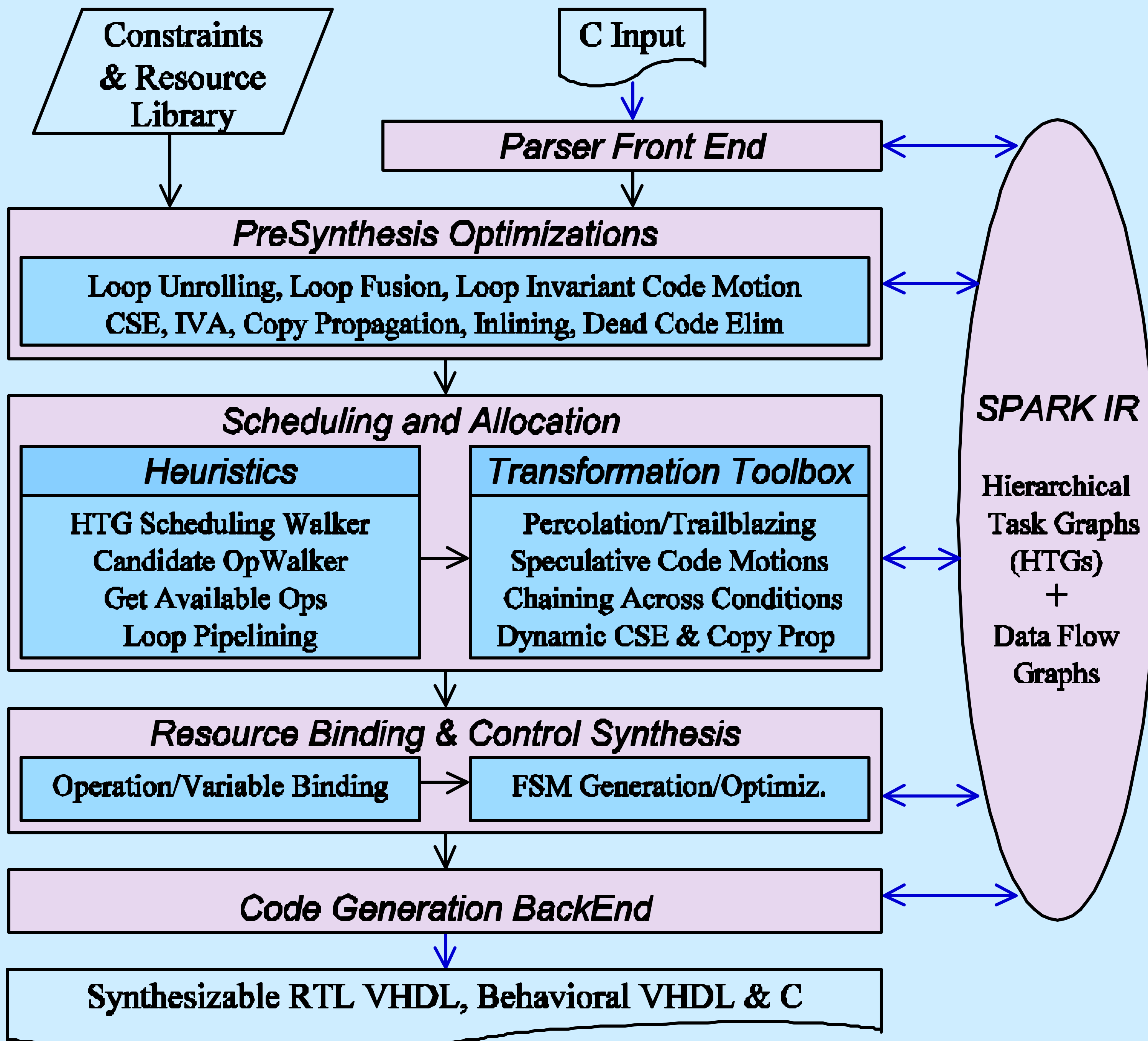


Copyright Sumit Gupta 2003

Loop Shifting: An Incremental Loop Pipelining Technique



Copyright Sumit Gupta 2003



SPARK
High Level Synthesis Framework

SPARK Parallelizing HLS Framework

- C input and Synthesizable RTL VHDL output
- Tool-box of Transformations and Heuristics
  - Each of these can be developed independently of the other
- Script based control over transformations & heuristics
- Hierarchical Intermediate Representation (HTGs)
  - Retains structural information about design (conditional blocks, loops)
  - Enables efficient and structured application of transformations
- Complete HLS tool: Does Binding, Control Synthesis and Backend VHDL generation
  - Interconnect Minimizing Resource Binding
- Enables Graphical Visualization of Design description and intermediate results
- 100,000+ lines of C++ code

Copyright Sumit Gupta 2003

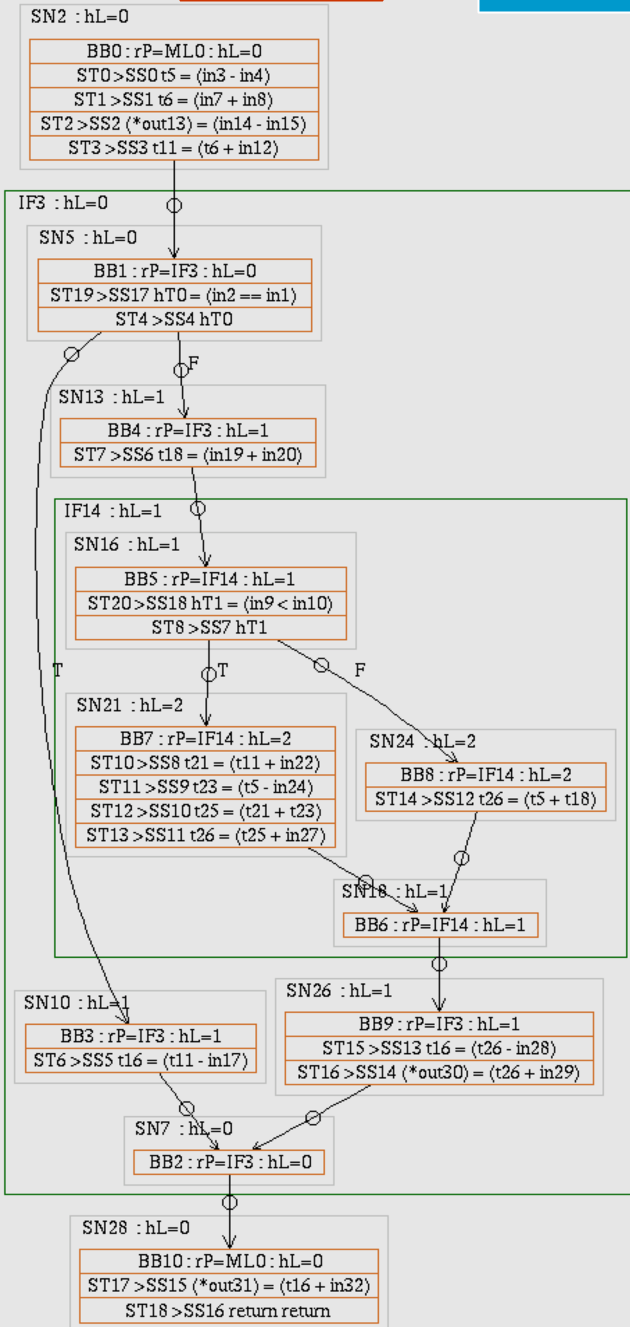
Synthesizable C

- ANSI-C front end from Edison Design Group (EDG)
- Features of C not supported for synthesis
  - Pointers
    - However, Arrays and passing by reference are supported
  - Recursive Function Calls
  - Gotos
- Features for which support has not been implemented
  - Multi-dimensional arrays
  - Structs
  - Continue, Breaks
- Hardware component generated for each function
  - A called function is instantiated as a hardware component in calling function

Copyright Sumit Gupta 2003

Graph Visualization

HTG

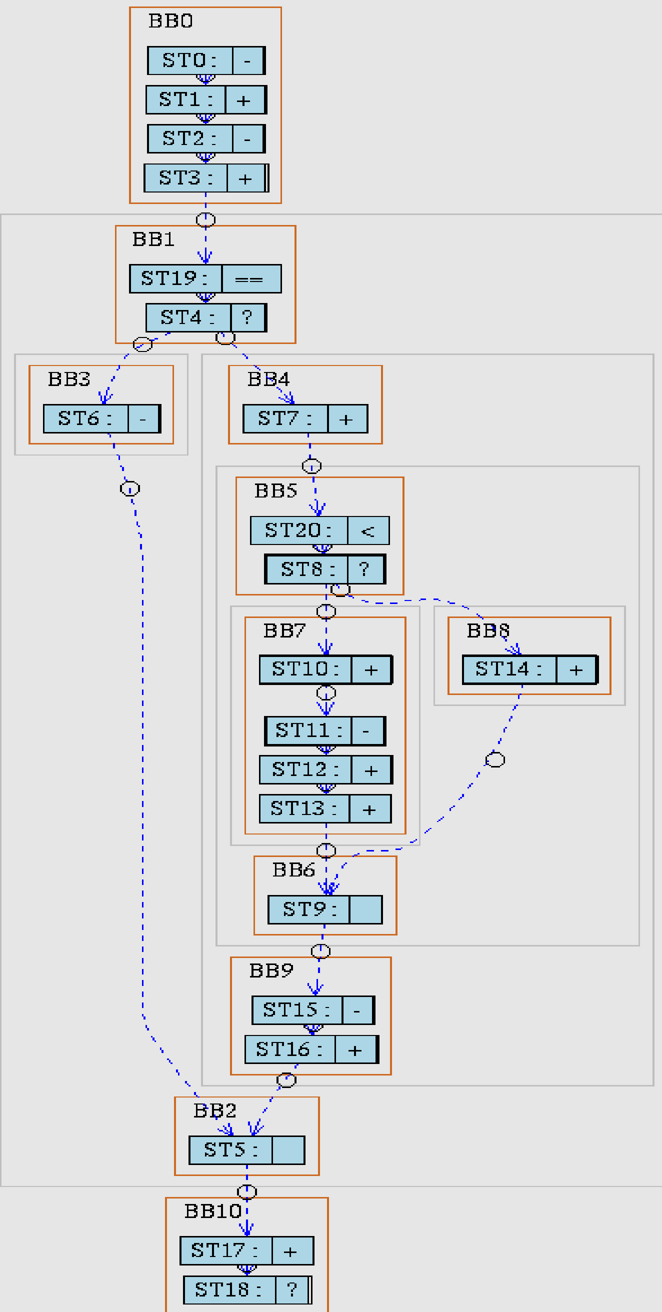


DFG

BB0
SS2 (*out13) = (in14 - in15)
SS1 t6 = (in7 + in8)
SS0 t5 = (in3 - in4)
SS3 t11 = (t6 + in12)
BB1
SS17 hT0 = (in2 == in1)
SS4 hT0
F
BB4
SS6 t18 = (in19 + in20)
BB5
SS18 hT1 = (in9 < in10)
SS7 hT1
F
T
T
BB8
SS12 t26 = (t5 + t18)
BB7
SS8 t21 = (t11 + in22)
SS9 t23 = (t5 - in24)
SS10 t25 = (t21 + t23)
SS11 t26 = (t25 + in27)
BB6
stmt_ST9
BB3
SS5 t16 = (t11 - in17)
BB9
SS14 (*out30) = (t26 + in29)
SS13 t16 = (t26 - in28)
BB2
stmt_ST5
BB10
SS15 (*out31) = (t16 + in32)

Copyright

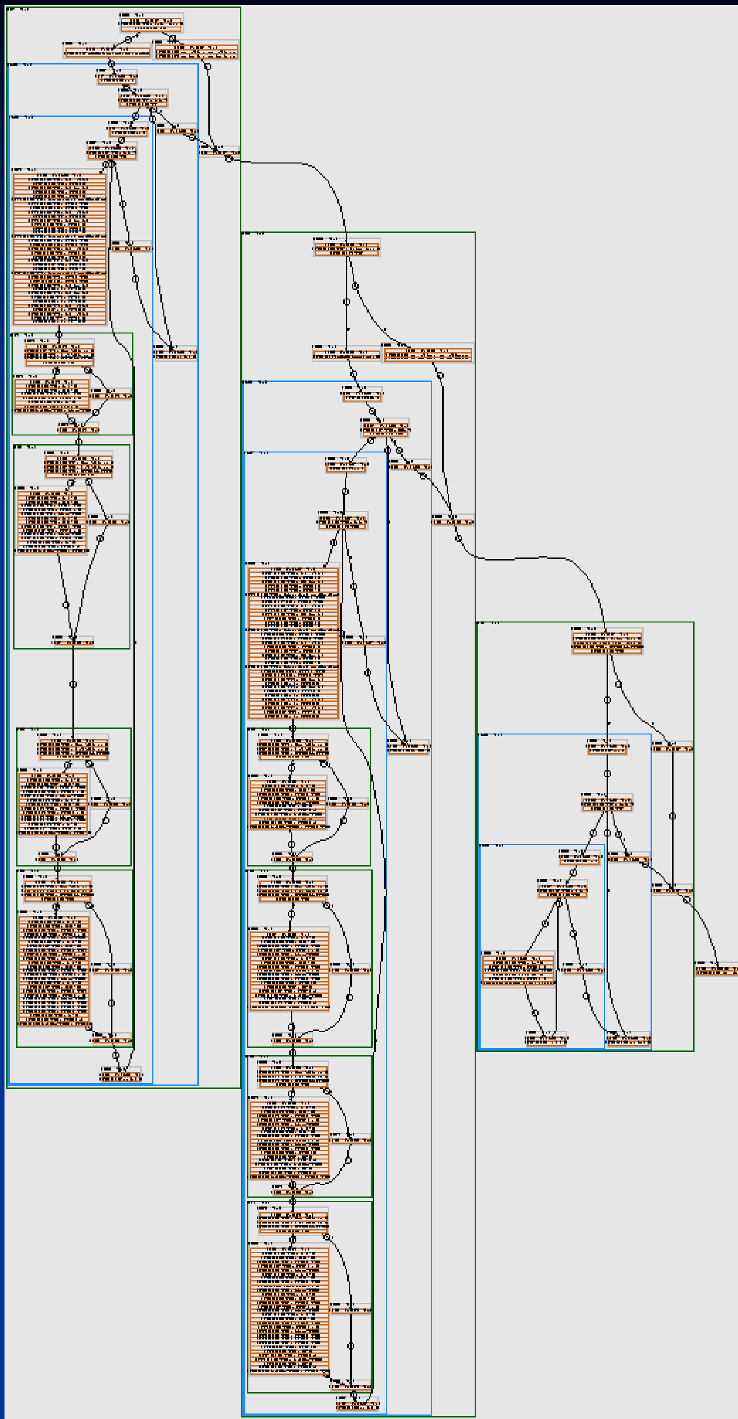
Resource Utilization Graph



Scheduling

BB0
ST0 : + - !=
BB1
ST4 :
BB3
ST6 : + -
ST21 : -
BB4
ST7 : + - !=
BB5
ST8 :
BB7
ST10 : + -
ST12 : +
ST13 : +
BB8
ST14 : + -
ST22 : +
BB6
ST9 :
BB9
ST15 : + -
BB2
ST5 :
BB10
ST17 : +
ST18 :

opyright Sum



Example of Complex HTG

- Example of a real design: MPEG-1 pred2 function
  - Just for demonstration; you are not expected to read the text
- Multiple nested loops and conditionals

yright Sumit Gupta 2003

Experiments

- Results presented here for
  - Pre-synthesis transformations
  - Speculative Code Motions
  - Dynamic CSE
- We used SPARK to synthesize designs derived from several industrial designs
  - MPEG-1, MPEG-2, GIMP Image Processing software
  - Case Study of Intel Instruction Length Decoder

■ Scheduling Results ■ Number of States in FSM ■ Cycles on Longest Path through Design	■ VHDL: Logic Synthesis ■ Critical Path Length (ns) ■ Unit Area

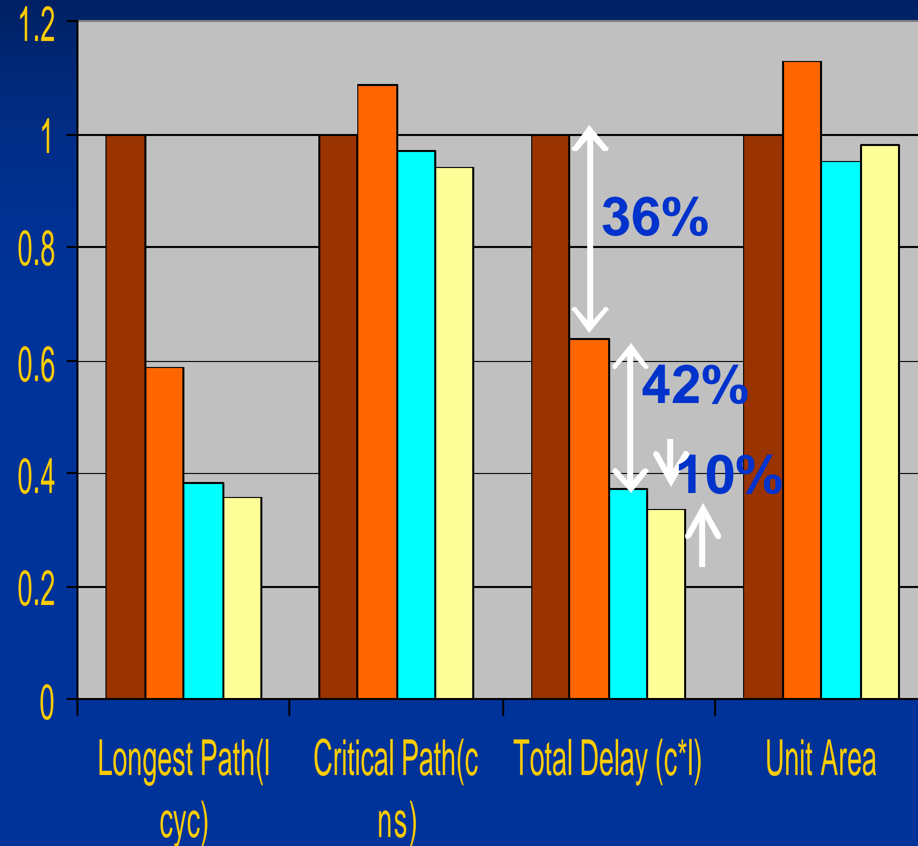
Copyright Sumit Gupta 2003

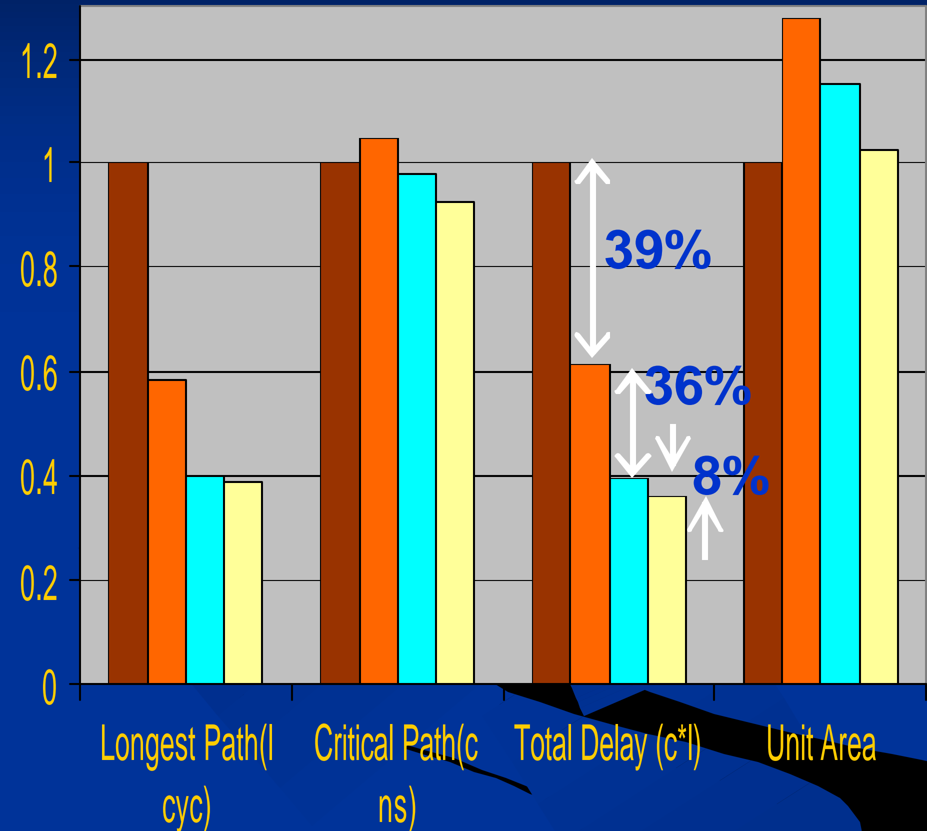
Target Applications

Design	# of Ifs	# of Loops	# Non-Empty Basic Blocks	# of Operations
MPEG-1 pred1	4	2	17	123
MPEG-1 pred2	11	6	45	287
MPEG-2 dp_frame	18	4	61	260
GIMP tiler	11	2	35	150

Copyright Sumit Gupta 2003

Scheduling & Logic Synthesis Results

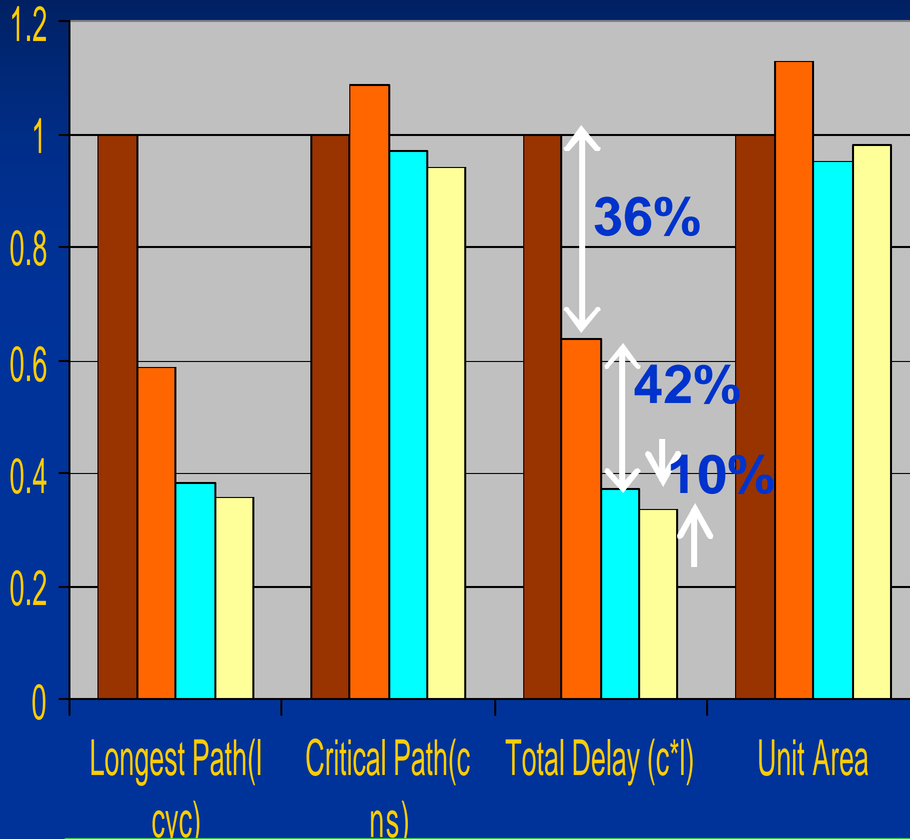


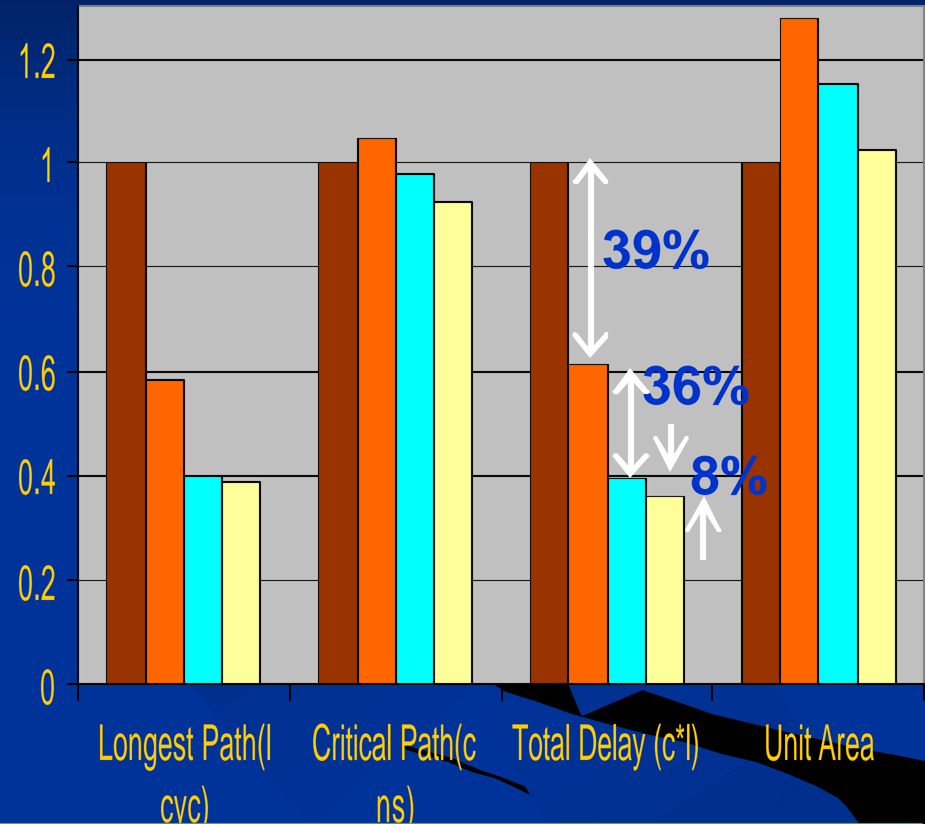


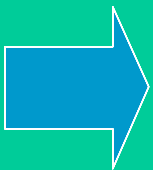
- Non-speculative CMs: Within BBs & Across Hier Blocks
- + Pre-Synthesis Transforms
- + Speculative Code Motions
- + Dynamic CSE

Copyright Sumit Gupta 2003

Scheduling & Logic Synthesis Results





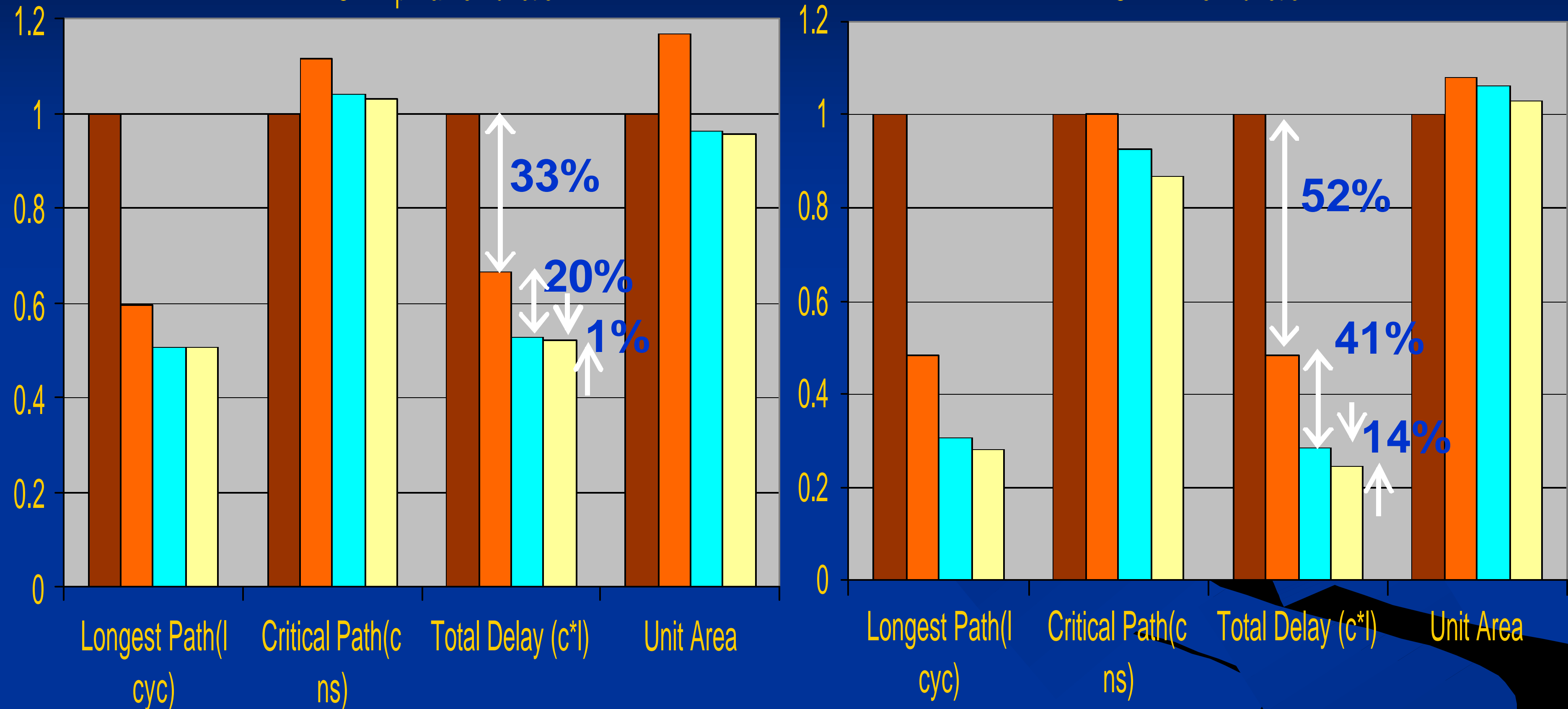


Overall: 63-66 % improvement in Delay

Almost constant Area

Copyright Sumit Gupta 2003

Scheduling & Logic Synthesis Results



Non-speculative CMs: Within BBs & Across Hier Blocks

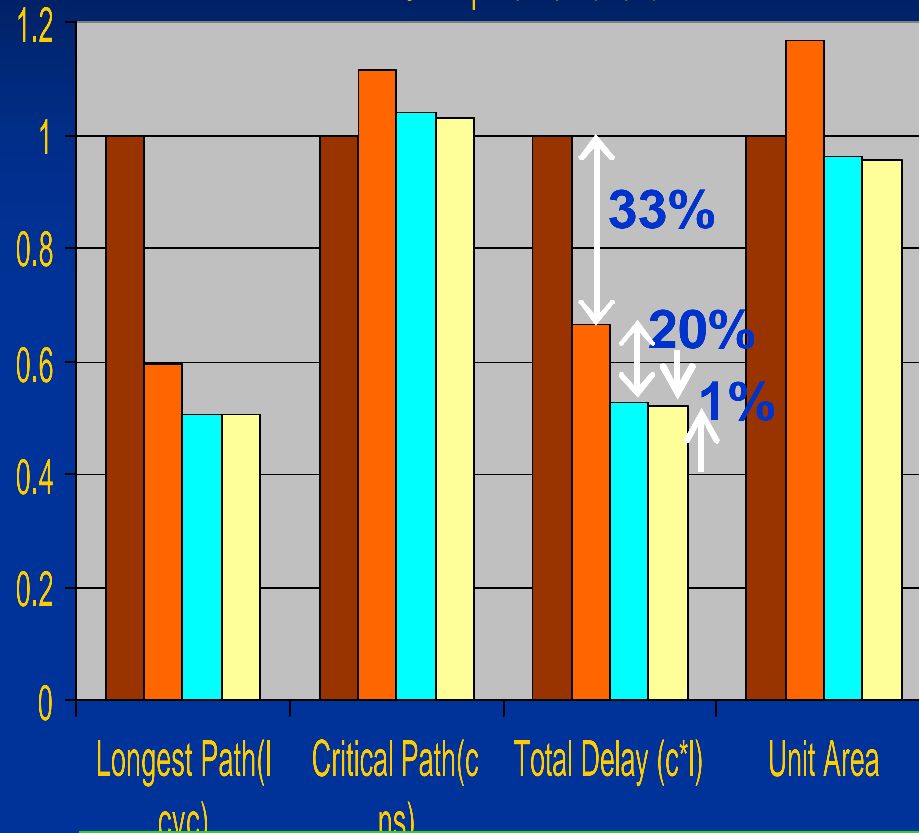
+ Speculative Code Motions

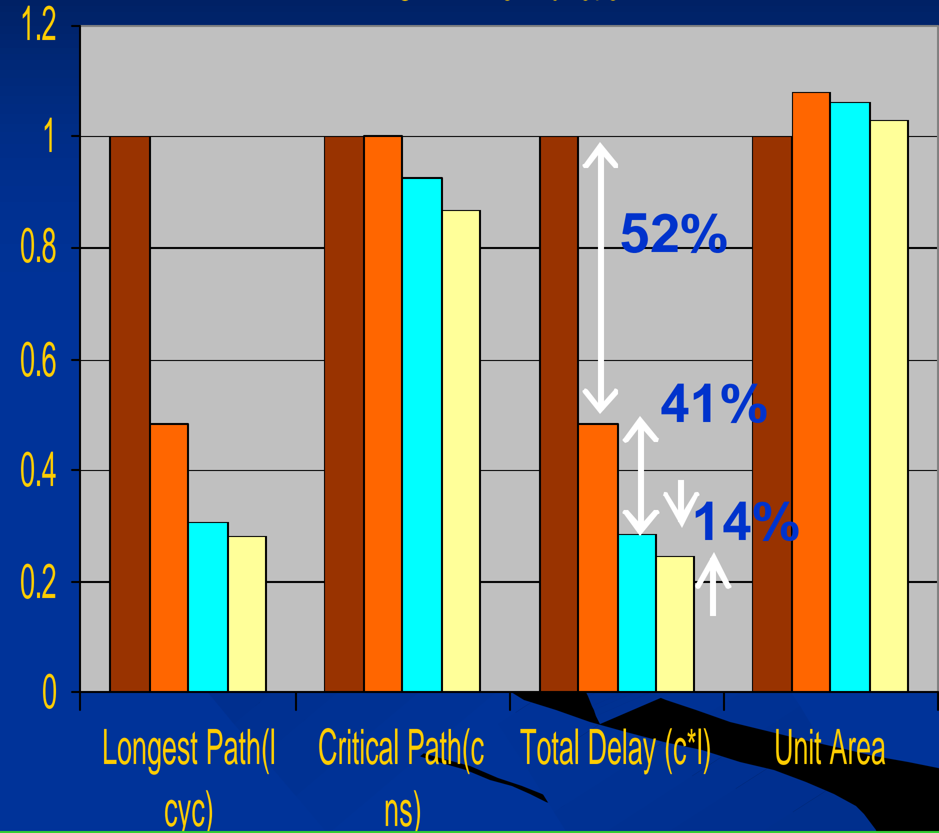
+ Pre-Synthesis Transforms

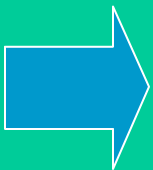
+ Dynamic CSE

Copyright Sumit Gupta 2003

Scheduling & Logic Synthesis Results





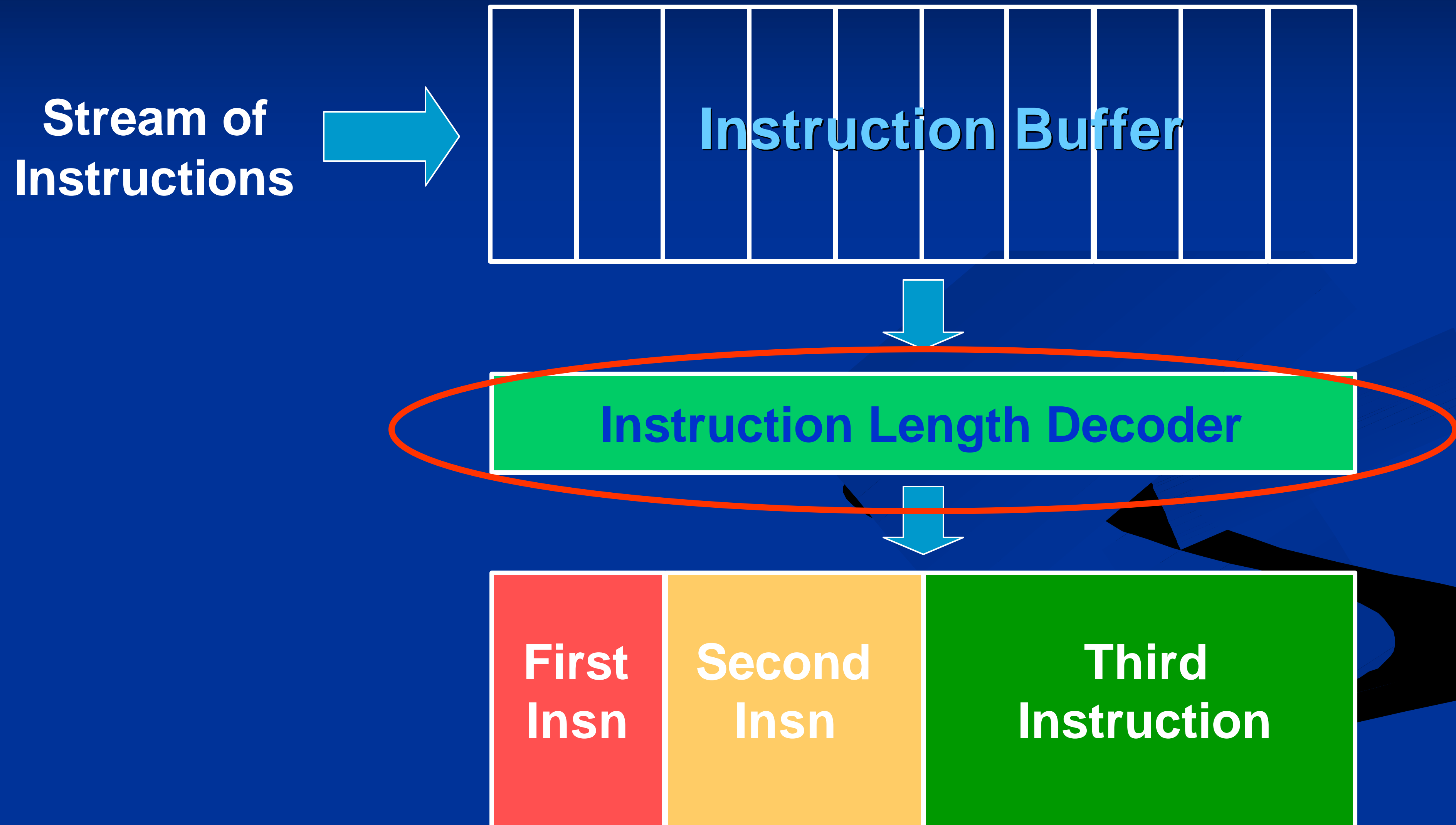


Overall: 48-76 % improvement in Delay

Almost constant Area

Copyright Sumit Gupta 2003

Case Study: Intel Instruction Length Decoder



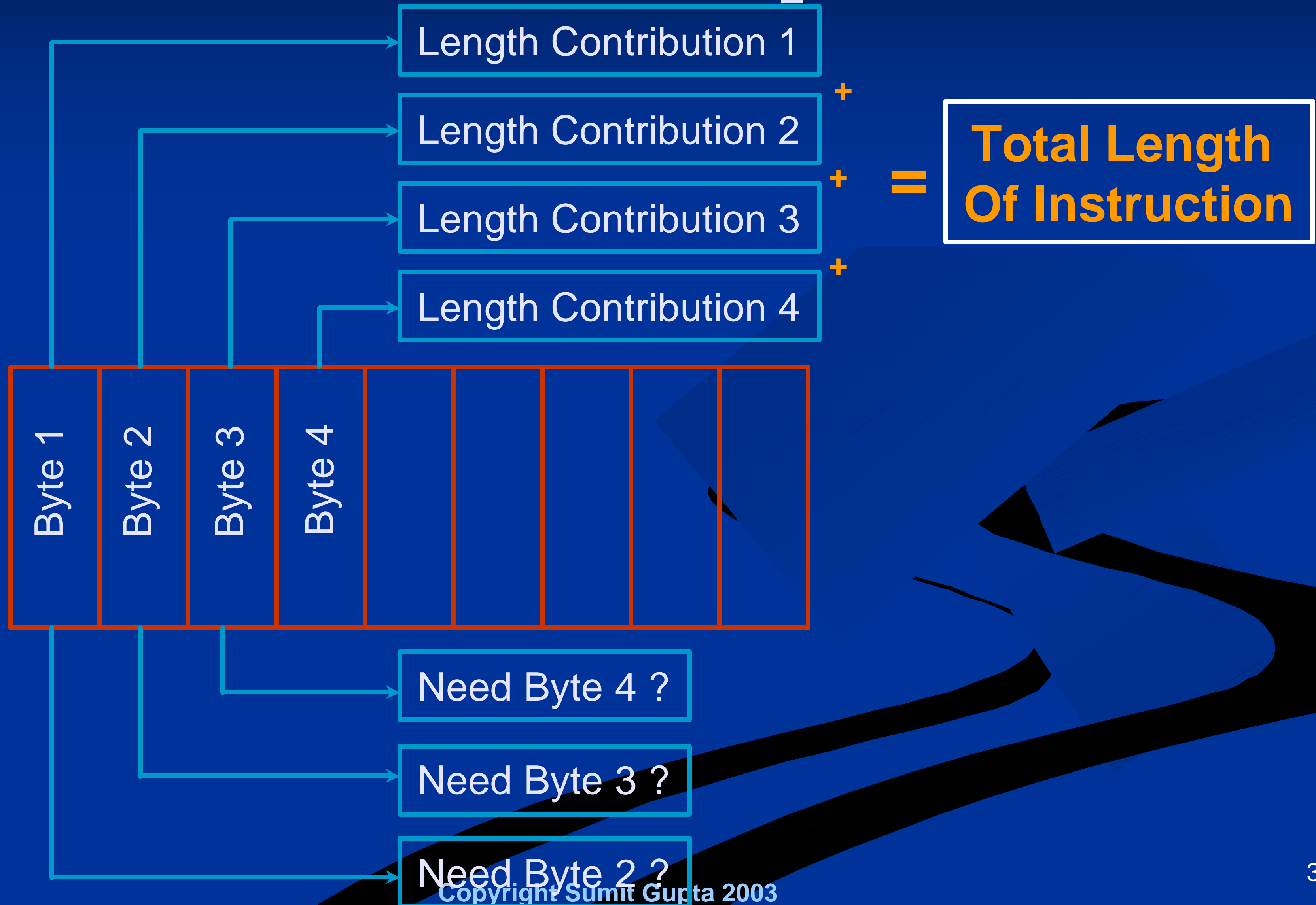
Copyright Sumit Gupta 2003

Example Design: ILD Block from Intel

- Case Study: A design derived from the Instruction Length Decoder of the Intel Pentium® class of processors
  - Decodes length of instructions streaming from memory
    - Has to look at up to 4 bytes at a time
  - Has to execute in one cycle and decode about 64 bytes of instructions
- Characteristics of Microprocessor functional blocks
  - Low Latency: Single or Dual cycle implementation
  - Consist of several small computations
  - Intermix of control and data logic

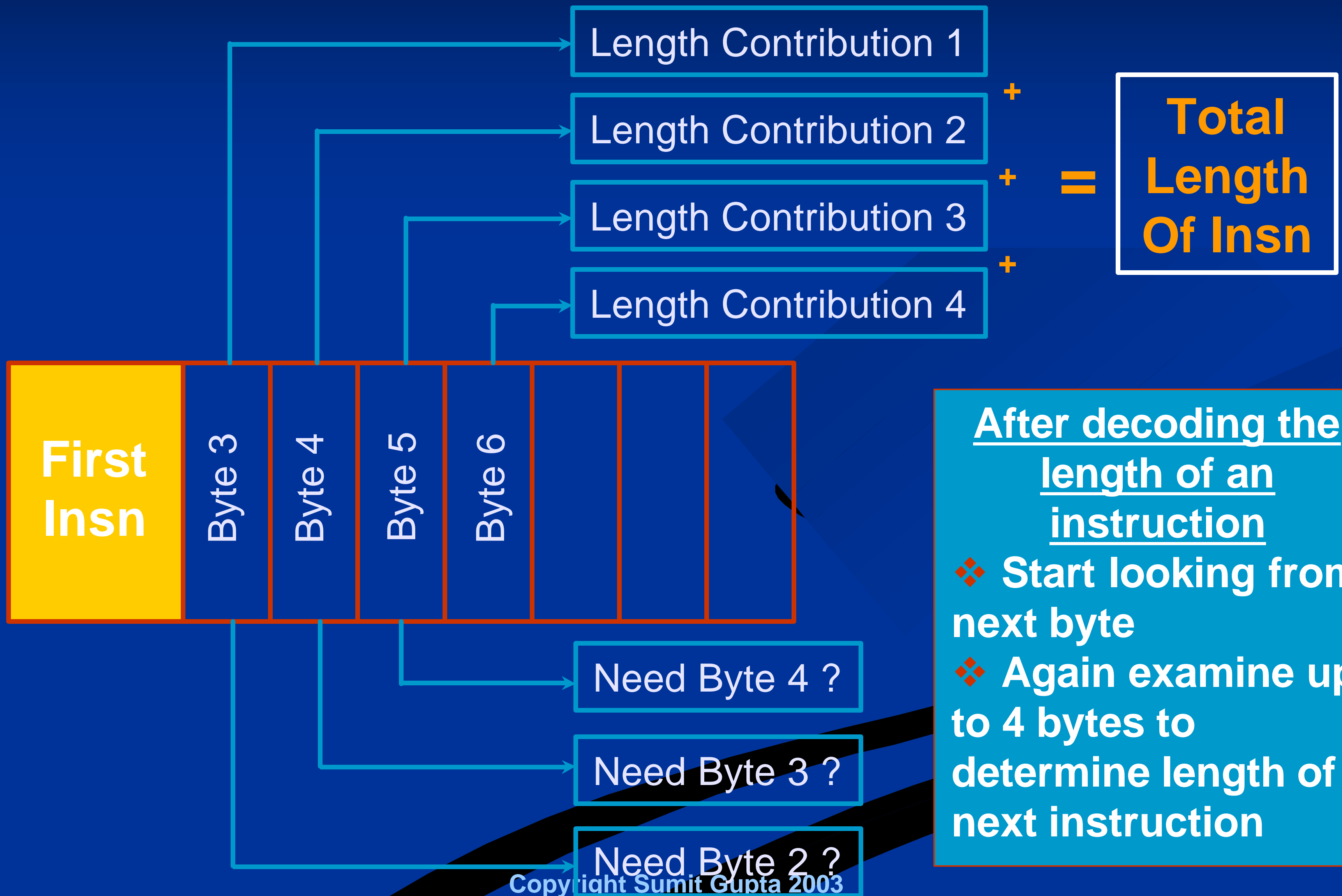
Copyright Sumit Gupta 2003

Basic Instruction Length Decoder: Initial Description



Copyright Sumit Gupta 2003

Instruction Length Decoder: Decoding 2nd Instruction

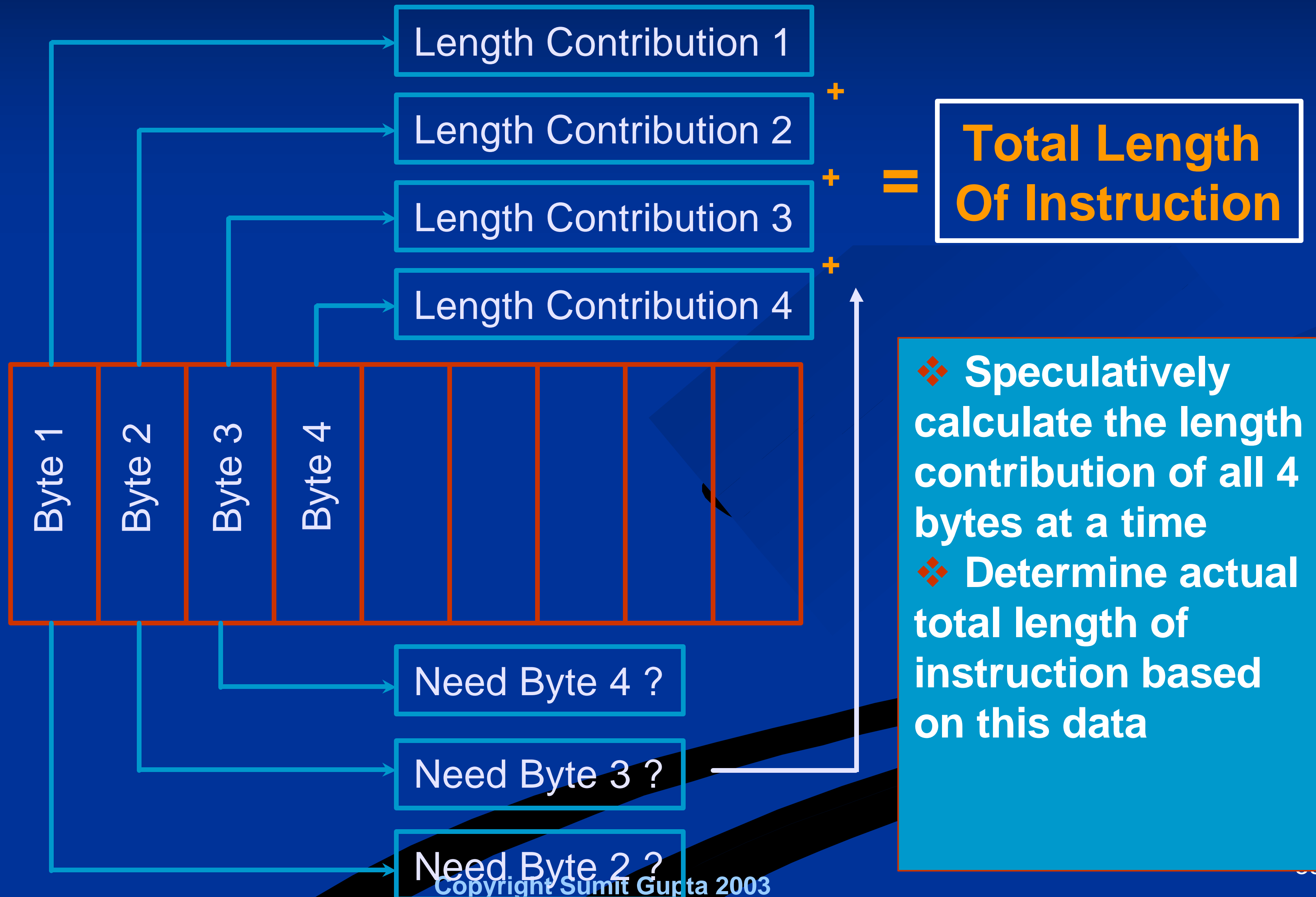


After decoding the length of an instruction

- Start looking from next byte
- Again examine up to 4 bytes to determine length of next instruction

Copyright Sumit Gupta 2003

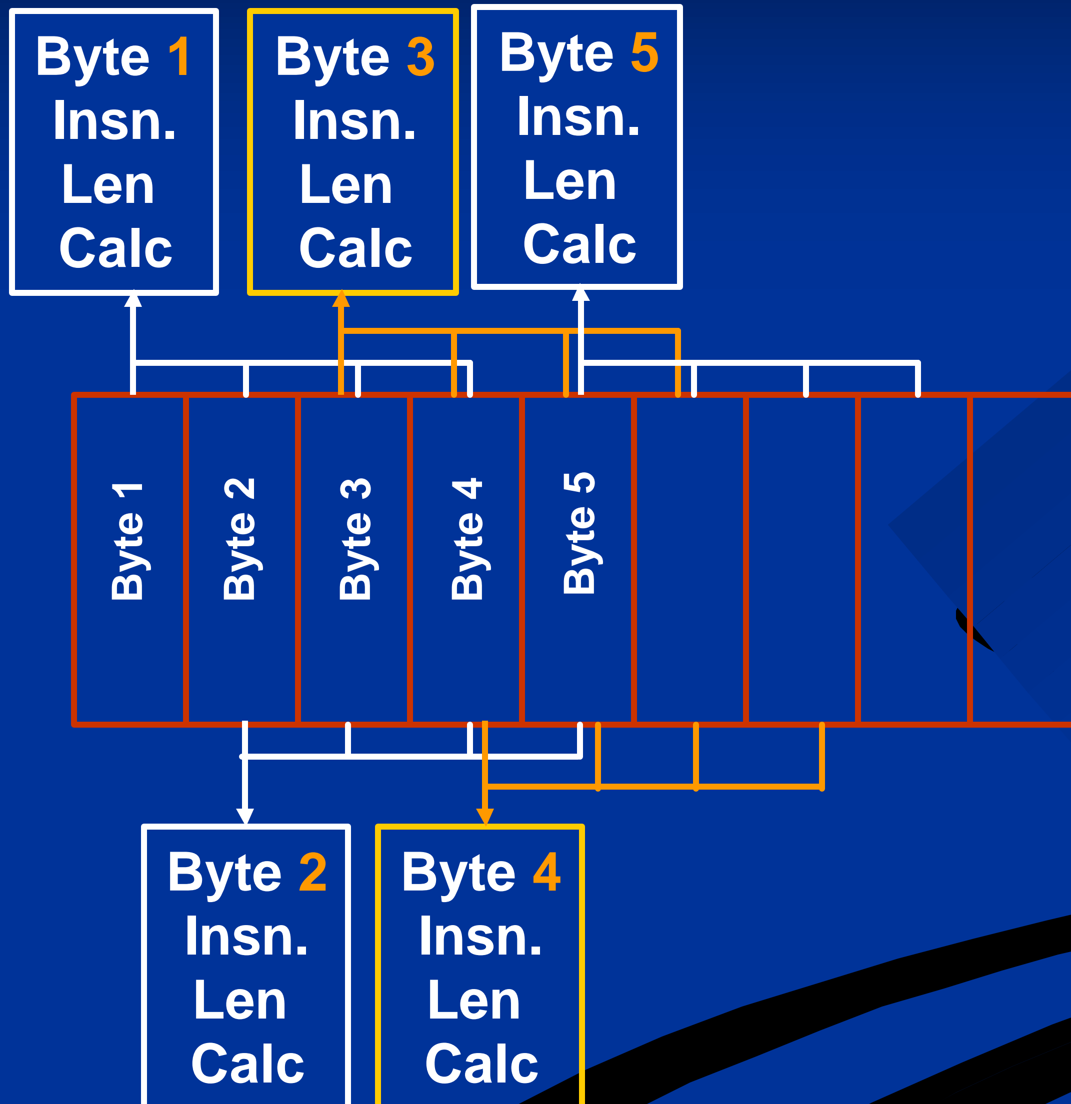
Instruction Length Decoder: Parallelized Description



- Speculatively calculate the length contribution of all 4 bytes at a time
- Determine actual total length of instruction based on this data

Copyright Sumit Gupta 2003

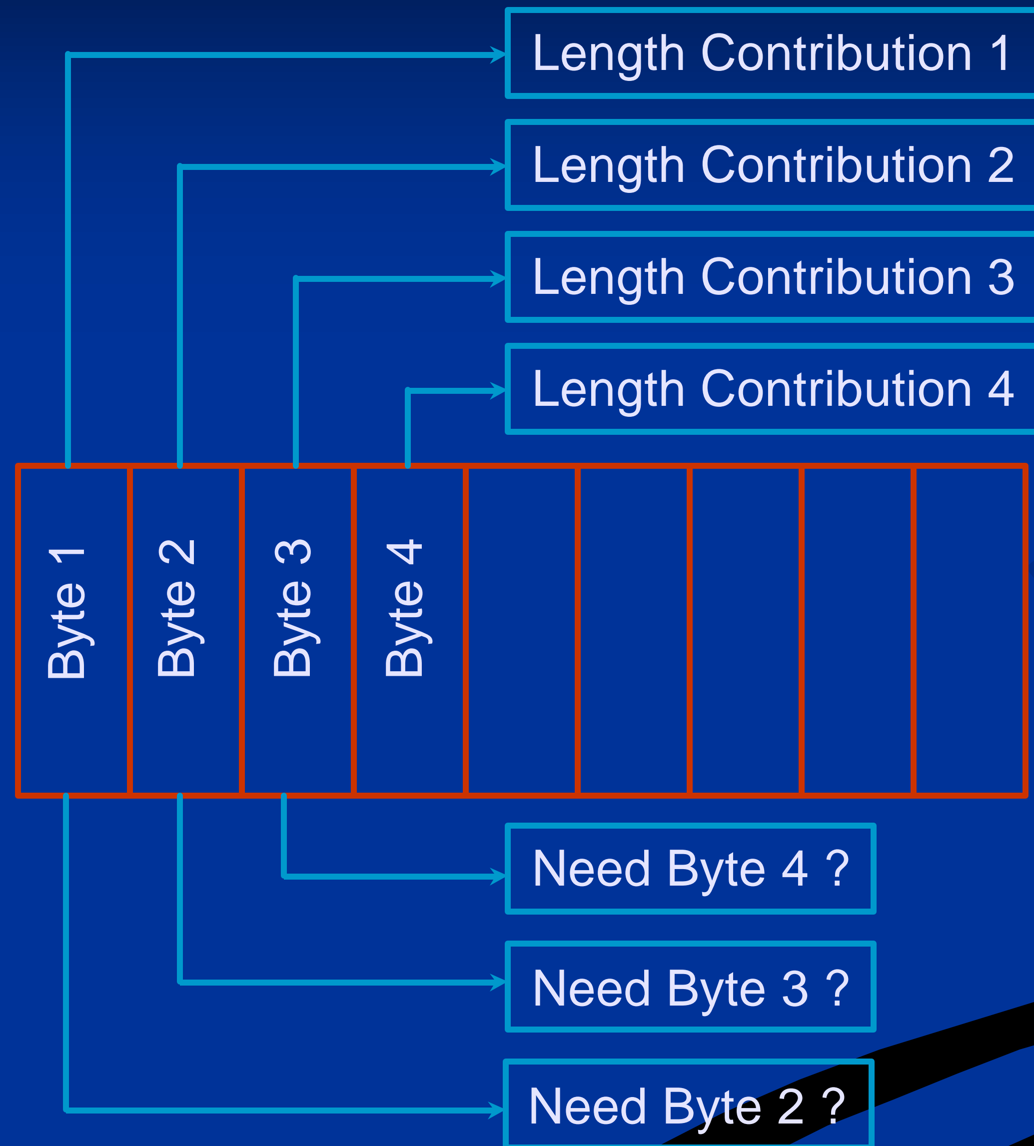
ILD: Extracting Further Parallelism



- Speculatively calculate length of instructions assuming a new instruction starts at each byte
- Do this calculation for all bytes in parallel
- Traverse from 1st byte to last
- Determine length of instructions starting from the 1st till the last
- Discard unused calculations

Copyright Sumit Gupta 2003

Initial: Multi-Cycle Sequential Architecture




```
ResetArray(Mark);
NextStartByte = 0;
for (i=0; i < n; i++) {
  if (i == NextStartByte) {
    lc1 = LengthContribution_1(i);
    if (Need_2nd_Byte(i)) {
      lc2 = LengthContribution_2(i+1);
      if (Need_3rd_Byte(i+1)) {
        lc3 = LengthContribution_3(i+2);
        if (Need_4th_Byte(i+2))  {
          lc4 = LengthContribution(i+3);
          Length = lc1 + lc2 + lc3 + lc4;
        } else
          Length = lc1 + lc2 + lc3;
      } else
        Length = lc1 + lc2;
    } else
      Length = lc1;
  } /* if (i == NextStartByte)  */
  len[i] = Length;
  NextStartByte += len[i];
  Mark[i] = 1;
} /* end of for i loop */
```

Copyright Sumit Gup

ILD Synthesis: Resulting Architecture

```
ResetArray(Mark);
NextStartByte = 0;
for (i=0; i < n; i++) {
  if (i == NextStartByte) {
    lc1 = LengthContribution_1(i);
    if (Need_2nd_Byte(i)) {
      lc2 = LengthContribution_2(i+1);
      if (Need_3rd_Byte(i+1)) {
        lc3 = LengthContribution_3(i+2);
        if (Need_4th_Byte(i+2))  {
          lc4 = LengthContribution(i+3);
          Length = lc1 + lc2 + lc3 + lc4;
        } else
          Length = lc1 + lc2 + lc3;
      } else
        Length = lc1 + lc2;
    } else
      Length = lc1;
  } /* if (i == NextStartByte)  */
  len[i] = Length;
  NextStartByte += len[i];
  Mark[i] = 1;
} /* end of for i loop */
```

Speculate Operations, Fully Unroll Loop, Eliminate Loop Index Variable

```
Results(0) = DataCalculation(0,1,2,3);
Results(1) = DataCalculation(1,2,3,4);
...
Length(0)  = ControlLogic(Results(0));
Length(1)  = ControlLogic(Results(1));
...
if (0 == NextStartByte) {
  NextStartByte += length[0];
  Mark[0] = 1;
}
if (1 == NextStartByte) {
  NextStartByte += length[1];
  Mark[1] = 1;
}
...
```

ight Sumi

ILD Synthesis: Resulting Architecture

```
ResetArray(Mark);
NextStartByte = 0;
for (i=0; i < n; i++) {
  if (i == NextStartByte) {
    lc1 = LengthContribution_1(i);
    if (Need_2nd_Byte(i)) {
```

Speculate Operations, Fully Unroll Loop, Eliminate Loop Index Variable

Multi-cycle Sequential Architecture → Single cycle Parallel Architecture

- Our toolbox approach enables us to develop a script to synthesize applications from different domains
- Final design looks close to the actual implementation done by Intel

Conclusions

- Parallelizing code transformations enable a new range of HLS transformations
  - Provide the needed improvement in quality of HLS results
    - Possible to be competitive against manually designed circuits.
  - Can enable productivity improvements in microelectronic design
- Built a synthesis system with a range of code transformations
  - Platform for applying Coarse and Fine-grain Optimizations
  - Tool-box approach where transformations and heuristics can be developed
    - Enables the designer to find the right synthesis script for different application domains
  - Performance improvements of 60-70 % across a number of designs
  - We have shown its effectiveness on an Intel design

Copyright Sumit Gupta 2003

Acknowledgements

- Advisors
  - Professors Rajesh Gupta, Nikil Dutt, Alex Nicolau
- Contributors to SPARK framework
  - Nick Savoiu, Mehrdad Reshadi, Sunwoo Kim
- Intel Strategic CAD Labs (SCL)
  - Timothy Kam, Mike Kishinevsky
- Supported by Semiconductor Research Corporation and Intel SCL

Copyright Sumit Gupta 2003

Thank You