SPARK PROJECT

Parallelizing High-level Synthesis

Rajesh K. Gupta, Sumit Gupta
University of California, San Diego

MESL . UCSD . EDU

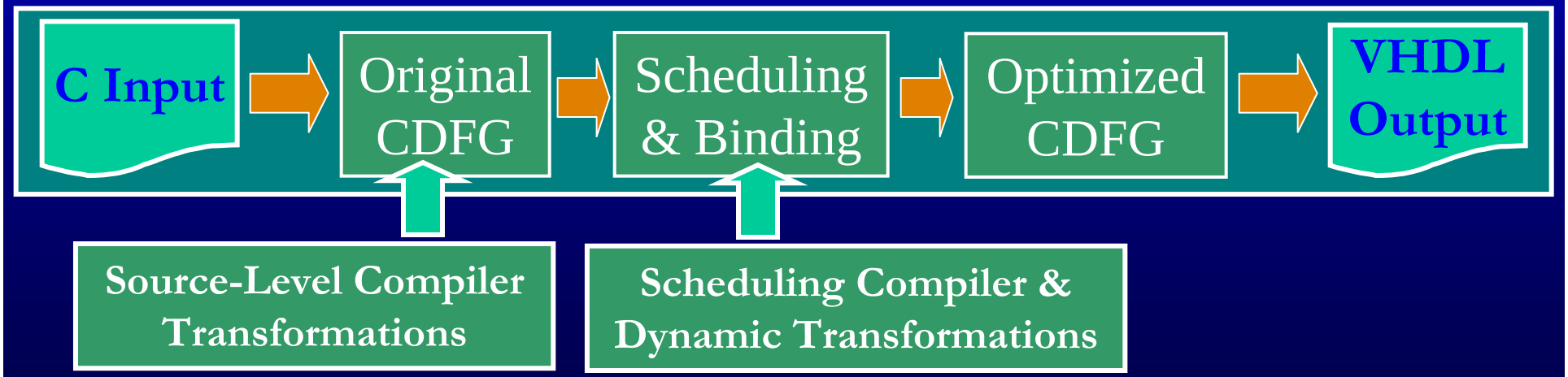
Contributors: Nick Savoiu, Sudipta Kundu
Collaborators: Nik Dutt, Alex Nicolau, UC Irvine

SPARK Vision & Strategy

- **Vision**
  - Aggressive & global code motions in an attempt to get past the QOR issue in HLS
- **Strategy**
  - Identify *really useful* parallelizing transformations
  - Apply coarse and fine grain HL & compiler optimizations
    - » target control flow transformations
    - » "Fine grain" loop optimization techniques for multiple and nested loops
    - » Mixed IR suitable for fine and coarse grain compiler transformations (similar to other systems such as SUIF)
  - Make it accessible through C, SystemC
- **Advantages**
  - Improve quality and controllability of HLS results.

Our Approach to Parallelizing HLS (PHLS)

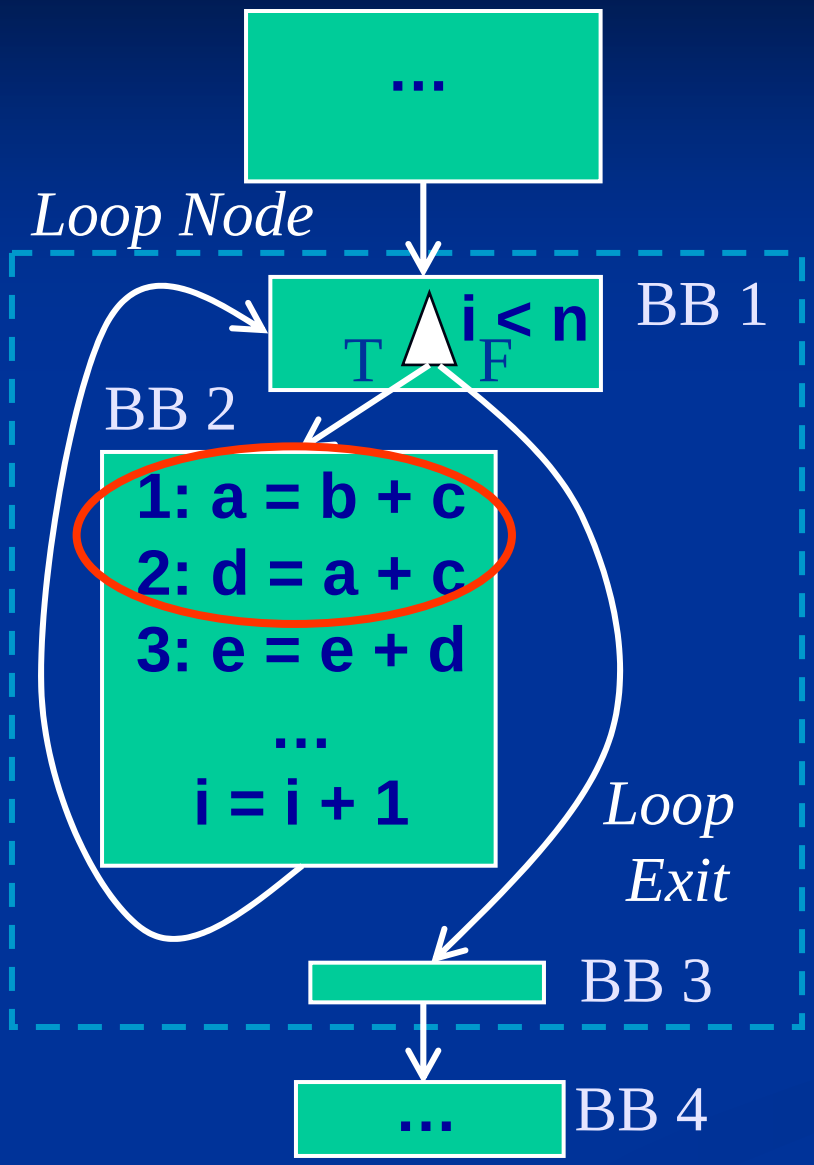
- **Transformations applied at the source level and during scheduling**
  - Source-level code refinement using Pre-synthesis transformations
  - Code Restructuring by Speculative Code Motions
  - Operation replication to improve concurrency
  - Dynamic transformations: exploit new opportunities during scheduling
- **Use heuristics to balance parallelism against hardware costs.**

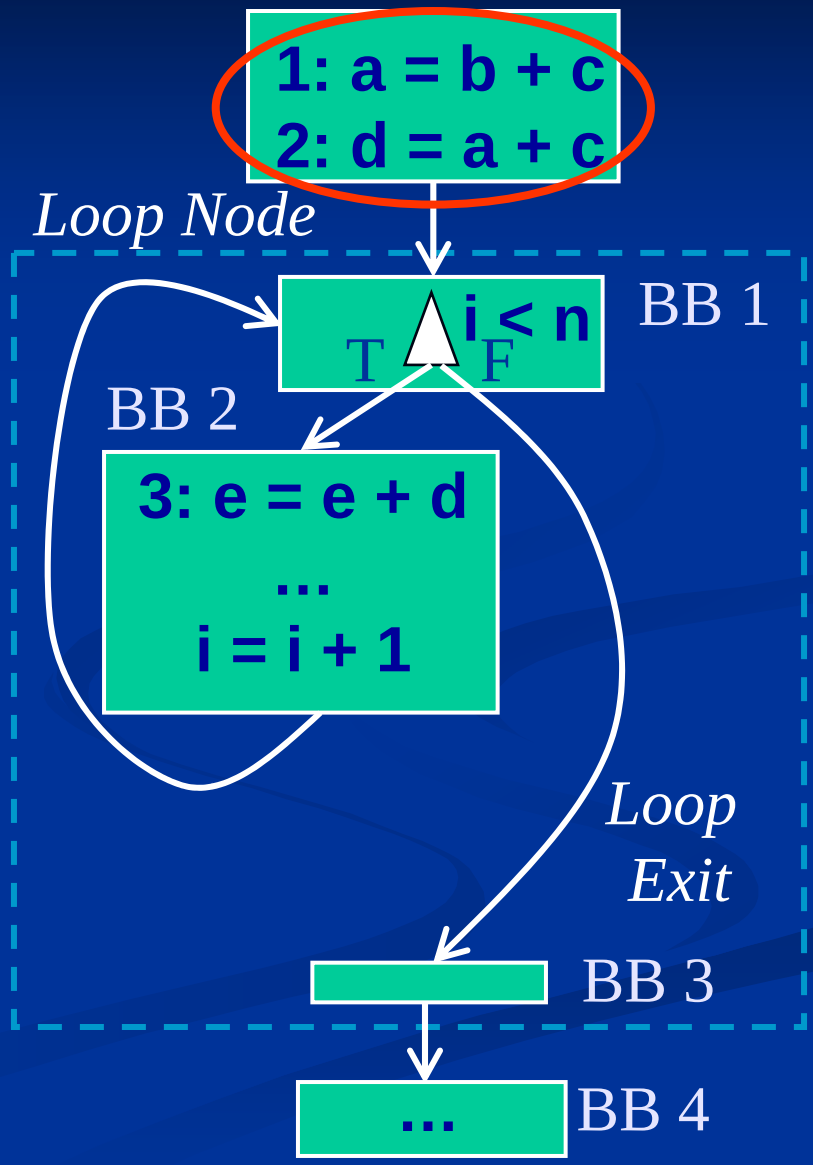


PHLS Transformations Organized into Four Groups

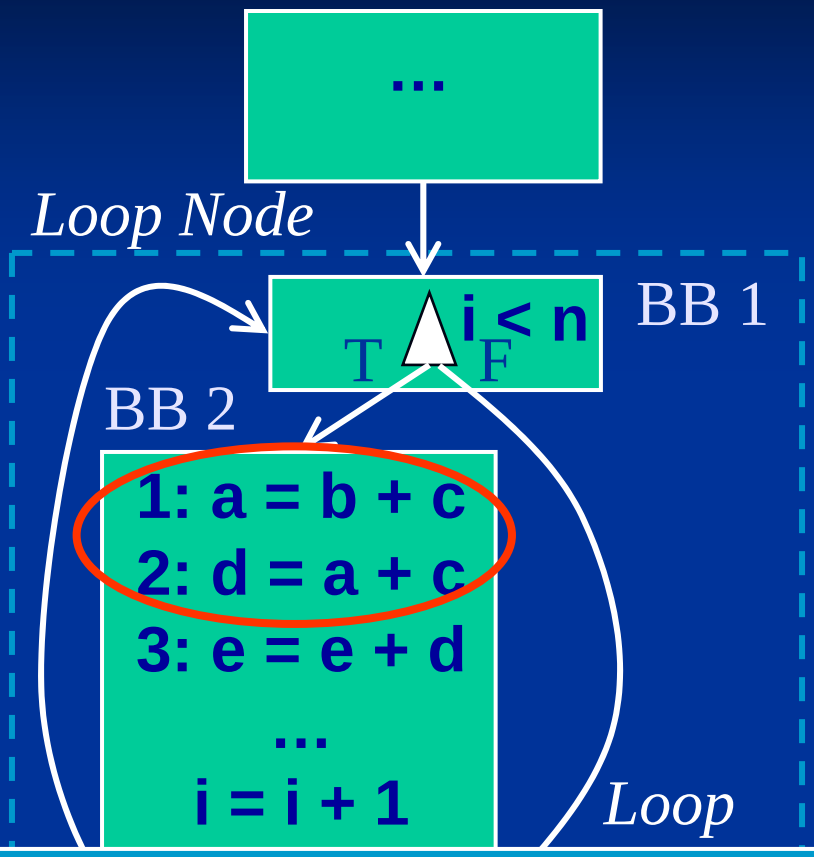
1. **Pre-synthesis:**
   - Loop-invariant code motions, Loop unrolling, CSE
2. **Scheduling:**
   - Speculative Code Motions, Multi-cycling, Operation Chaining, Loop Pipelining
3. **Dynamic:**
   - Transformations applied dynamically during scheduling: Dynamic CSE, Dynamic Copy Propagation, Dynamic Branch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

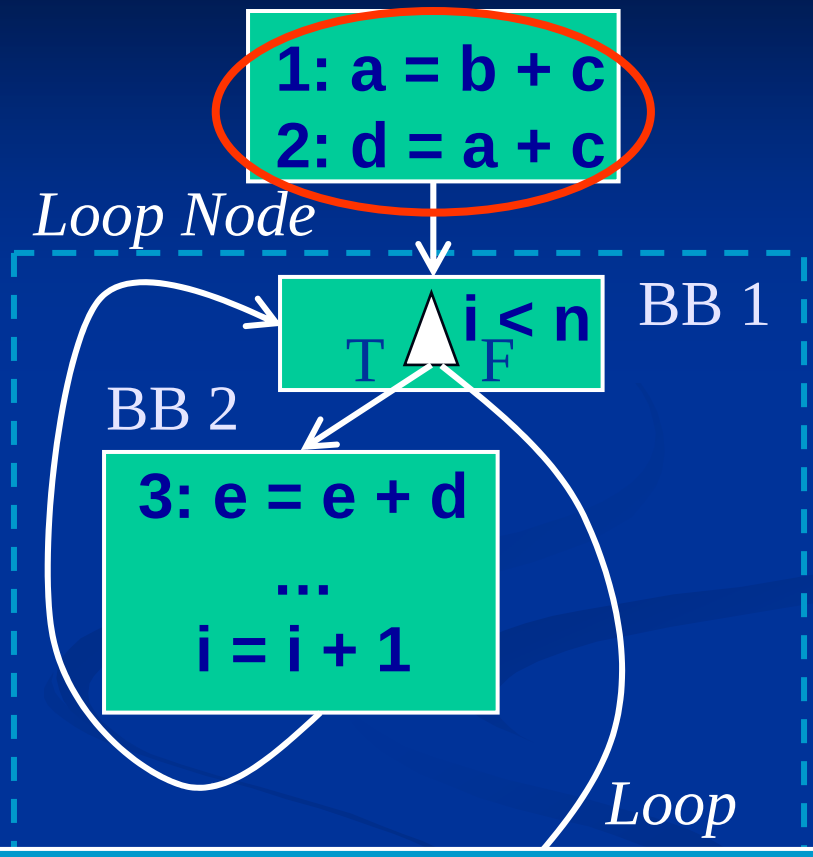
1. Pre-synthesis: Loop Invariant CM





1. Pre-synthesis: Loop Invariant CM



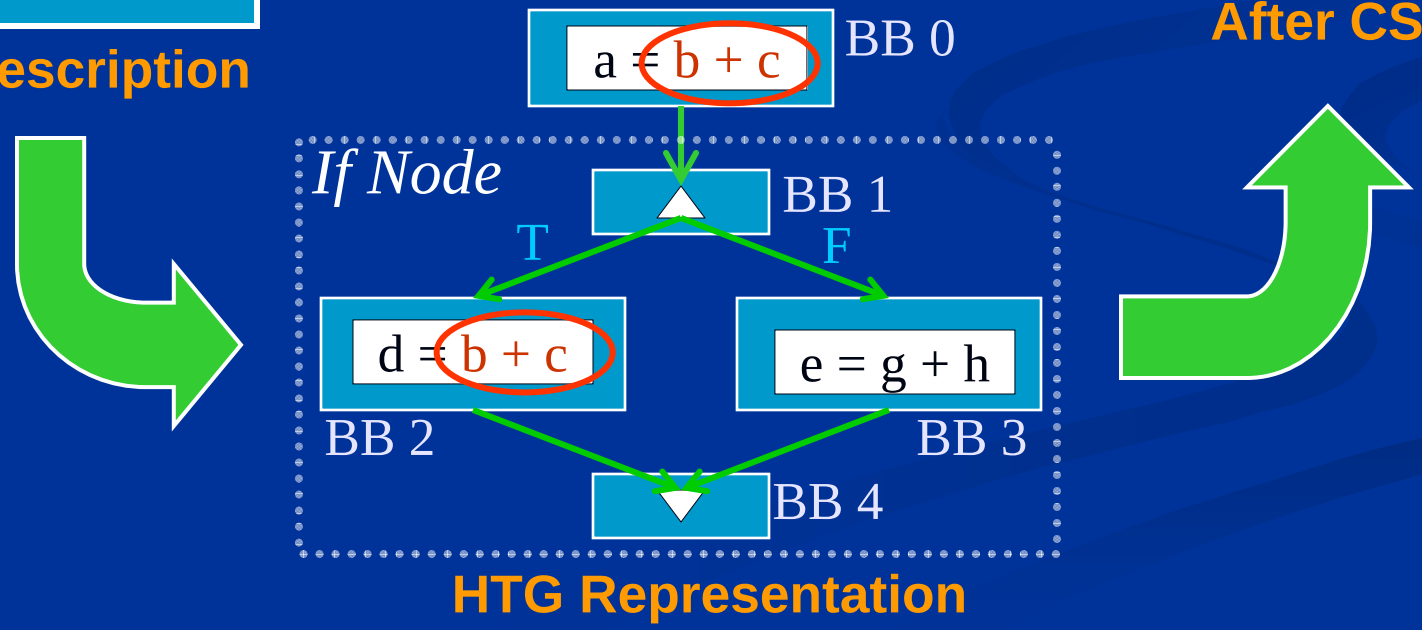


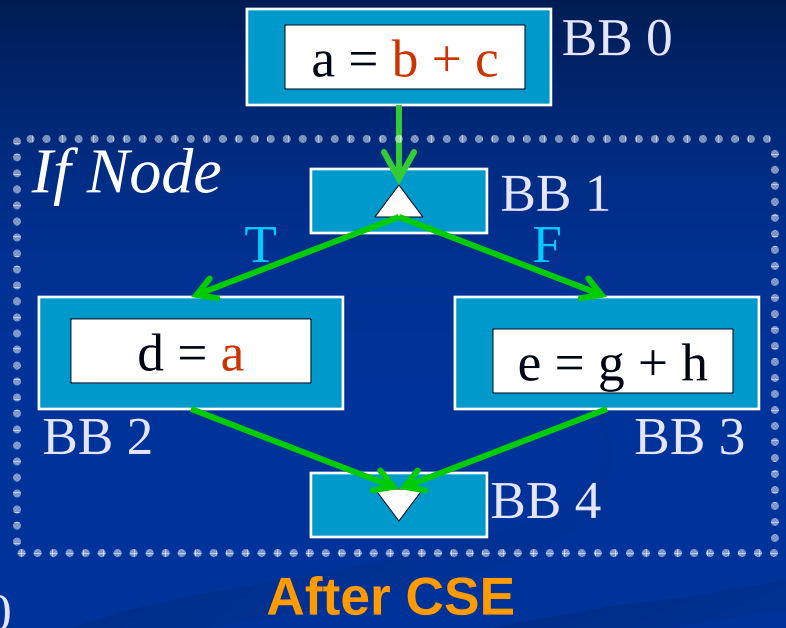
- Reduce number of operations that execute in loops
- Putting code inside loops is a **programming convenience**
  - Common situation in media applications

1. Common Sub-Expression Elimination

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

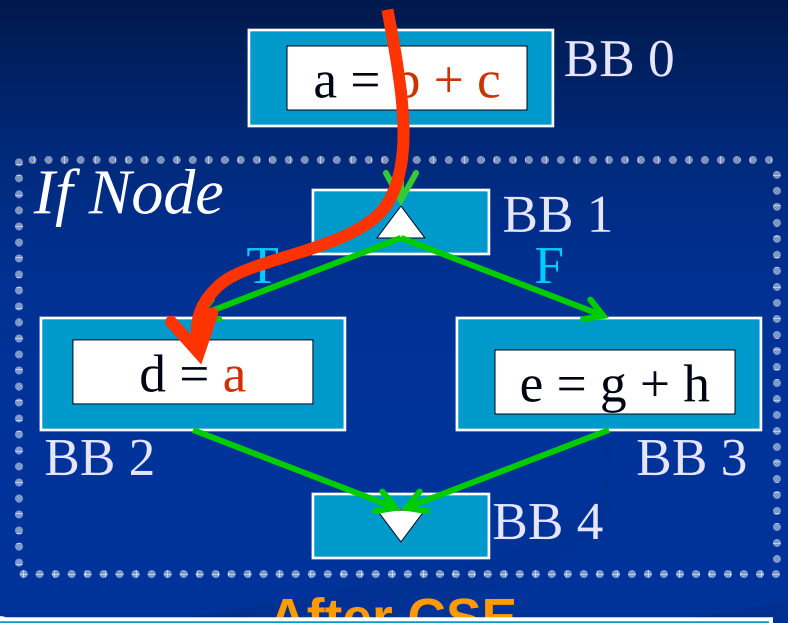
C Description





1. Common Sub-Expression Elimination

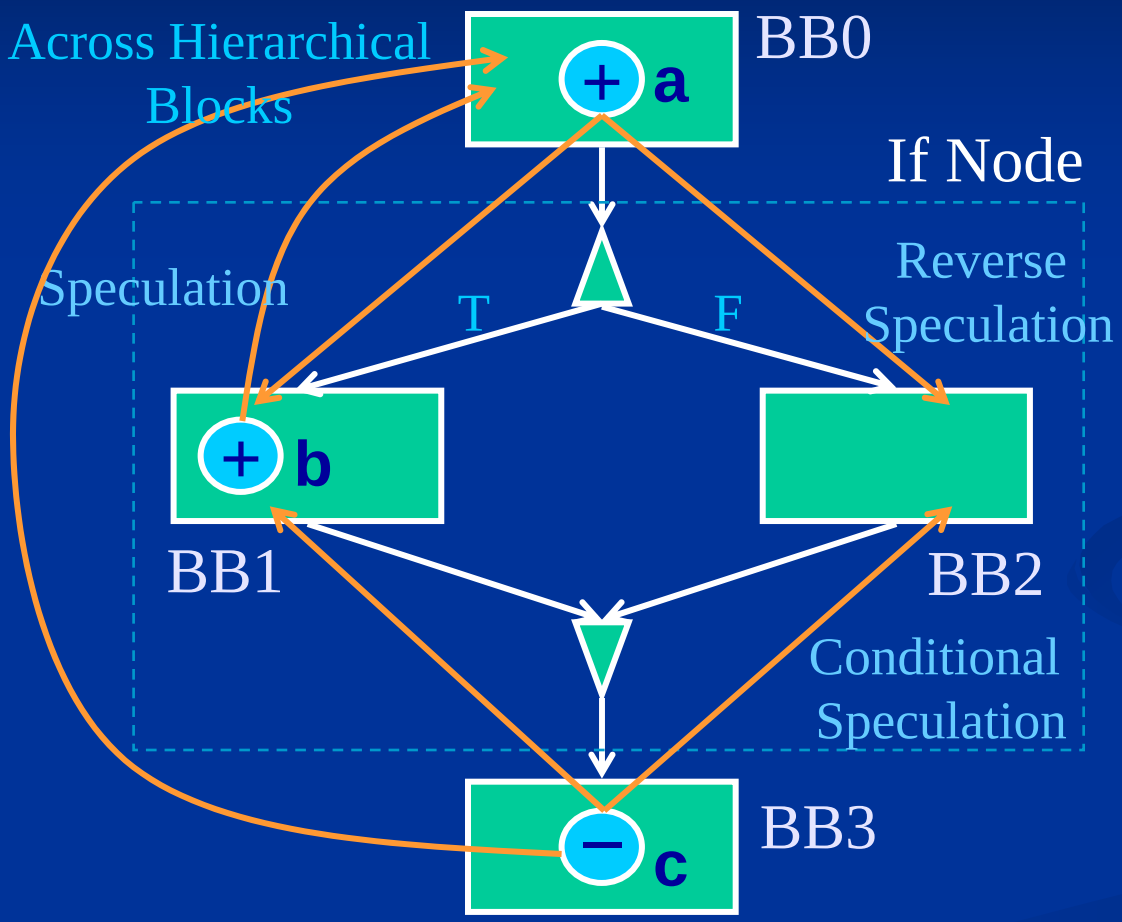
```
a = b + c;
c = b < c;
if (c)
  d = b + c;
else
  e = g + h;
```



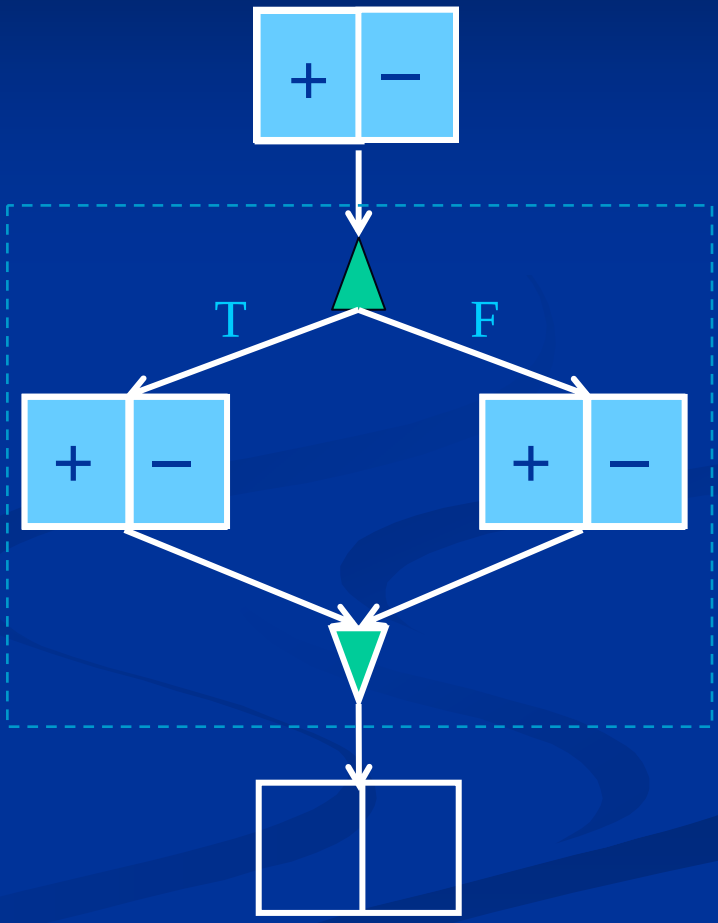
- We use notion of **Dominance** of Basic Blocks
  - A basic block **BBi** dominates another basic block **BBj** if all control paths from the **initial** basic block of the design graph leading to **BBj** goes through **BBi**
- We can eliminate an operation **opj** in **BBj** using common expression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

2. Scheduling Transformations:
Speculative Code Motions

Hierarchical Task Graph Representation



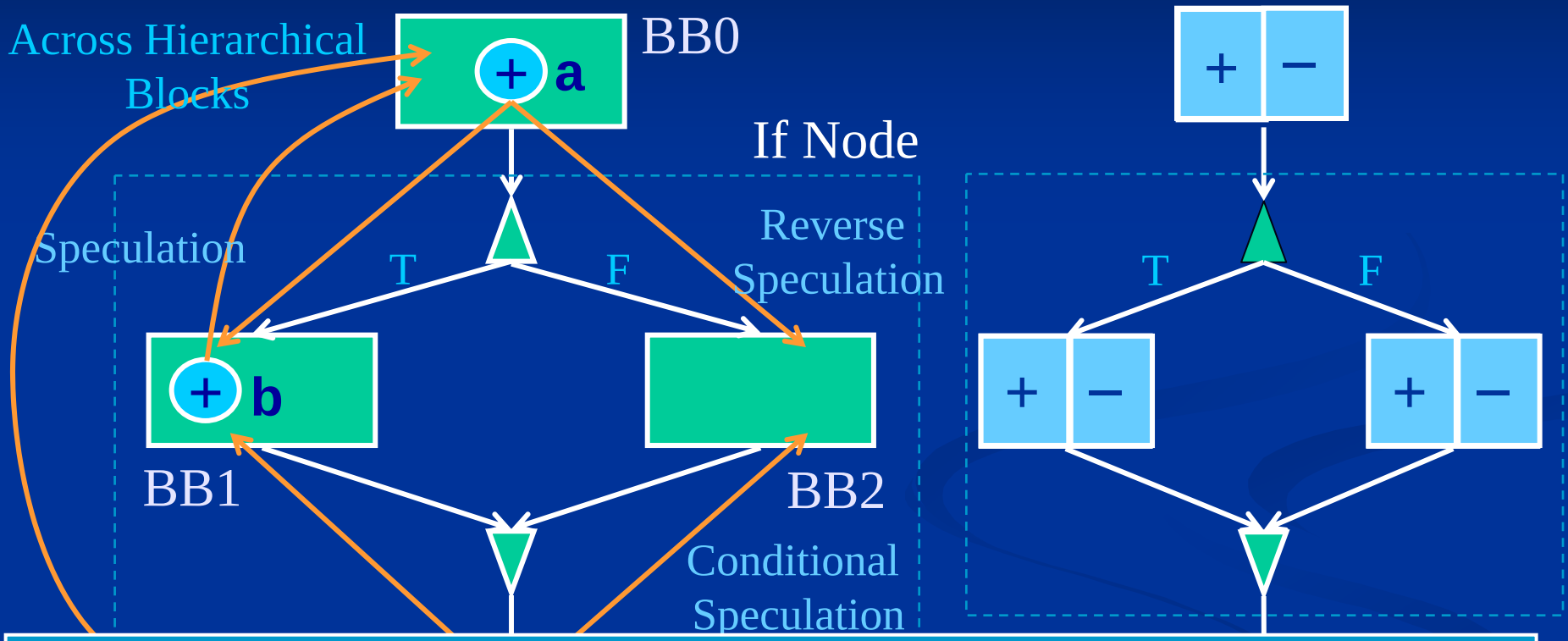
Resource Utilization



Schedule under Resource Constraints + –

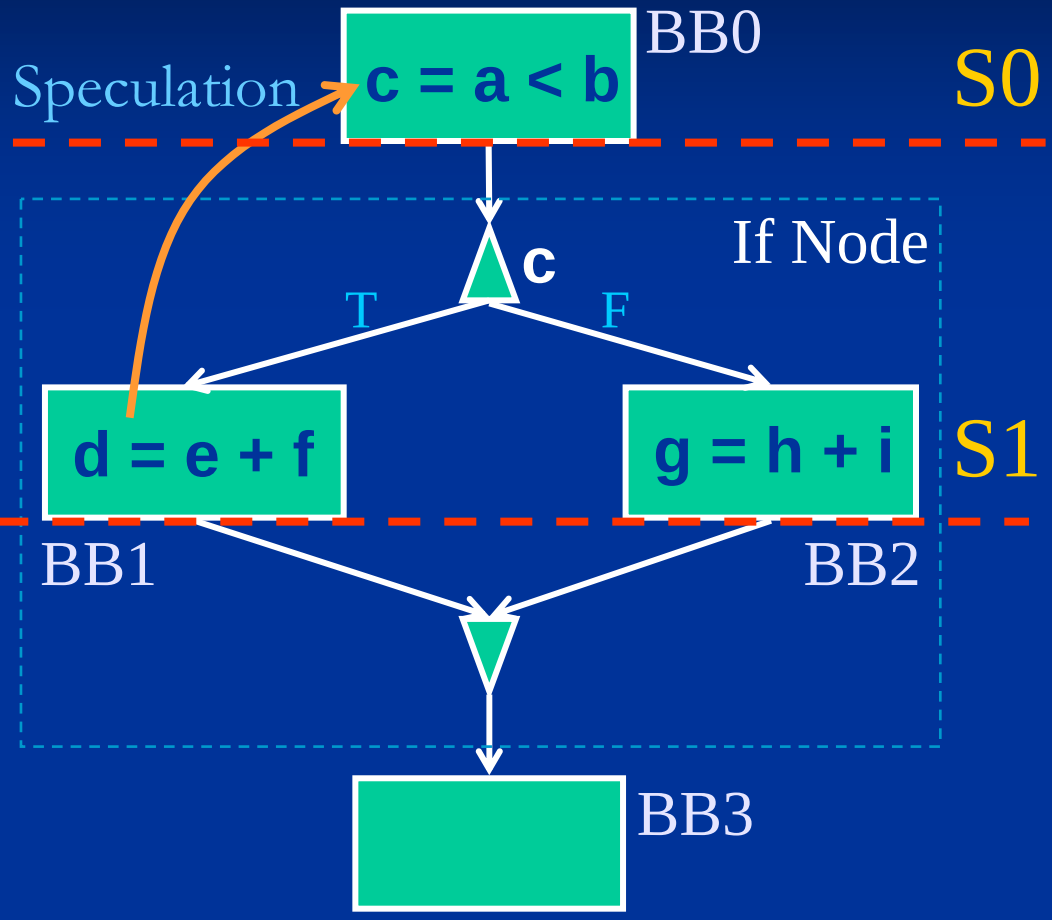
2. Scheduling Transformations:
Speculative Code Motions

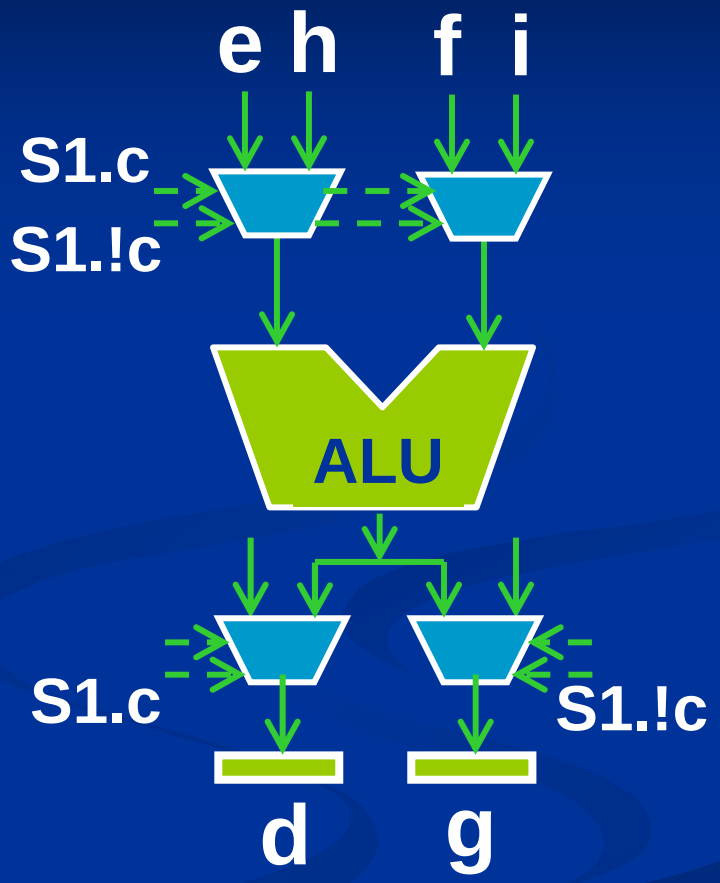
Hierarchical Task Graph Representation **Resource Utilization**



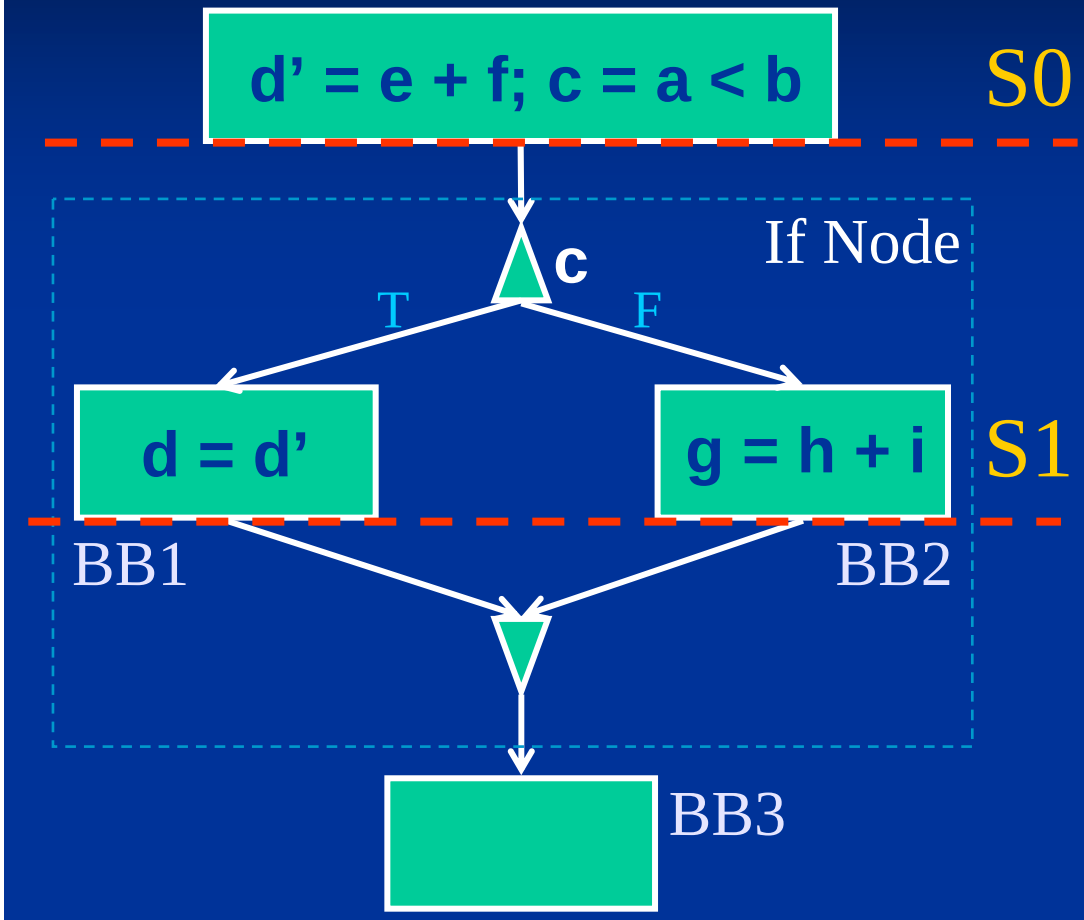
- Lead to Shorter Schedule Lengths by utilizing resources that are "idle" in earlier cycles
- Reduce the impact of programming style (operation placement) on quality of HLS results

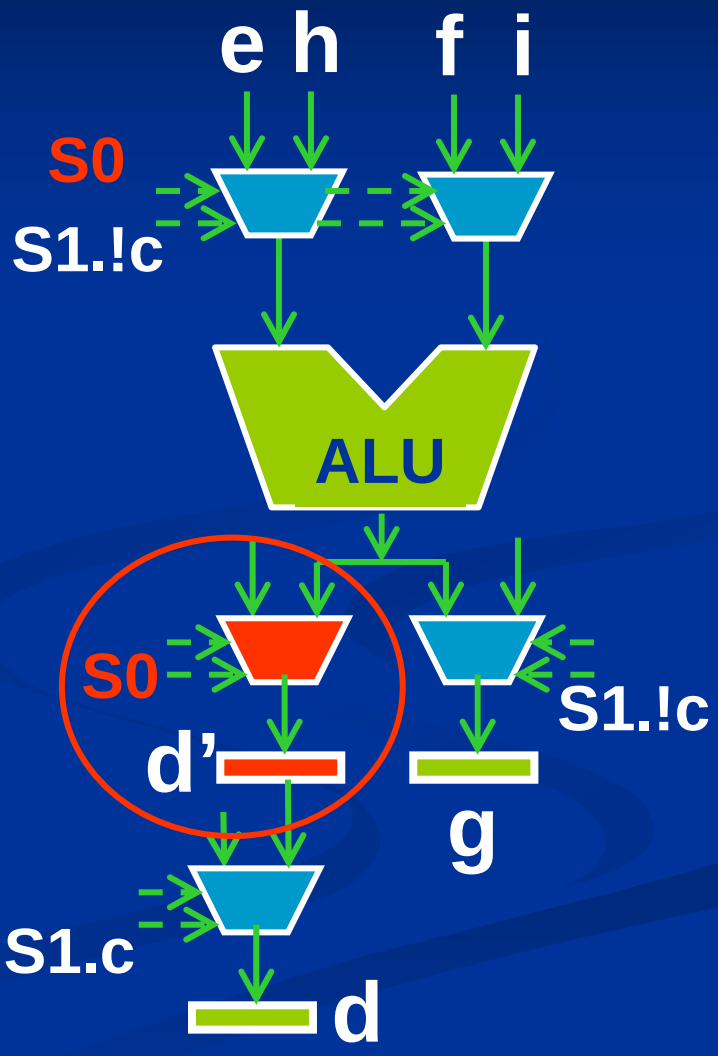
Hardware Costs of Speculative Code Motions: Speculation



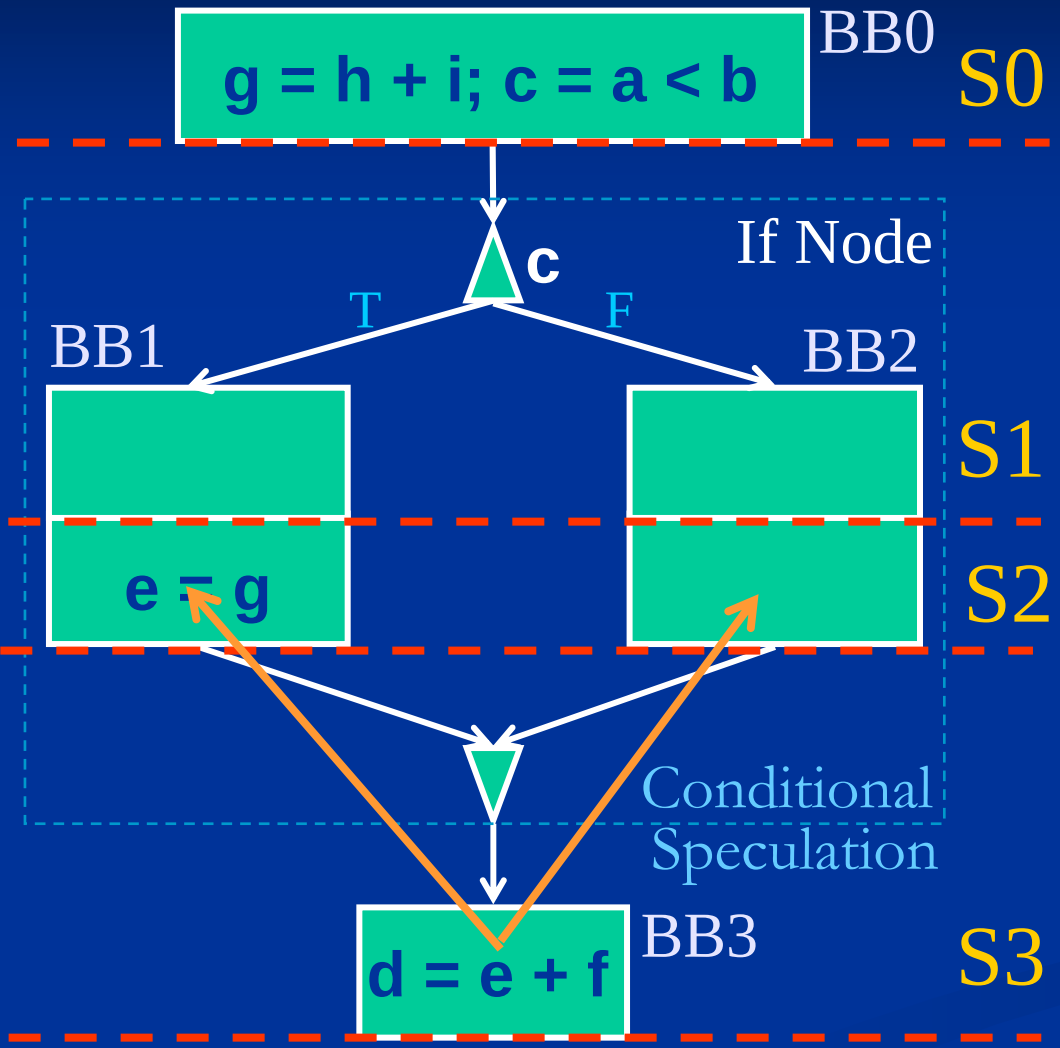


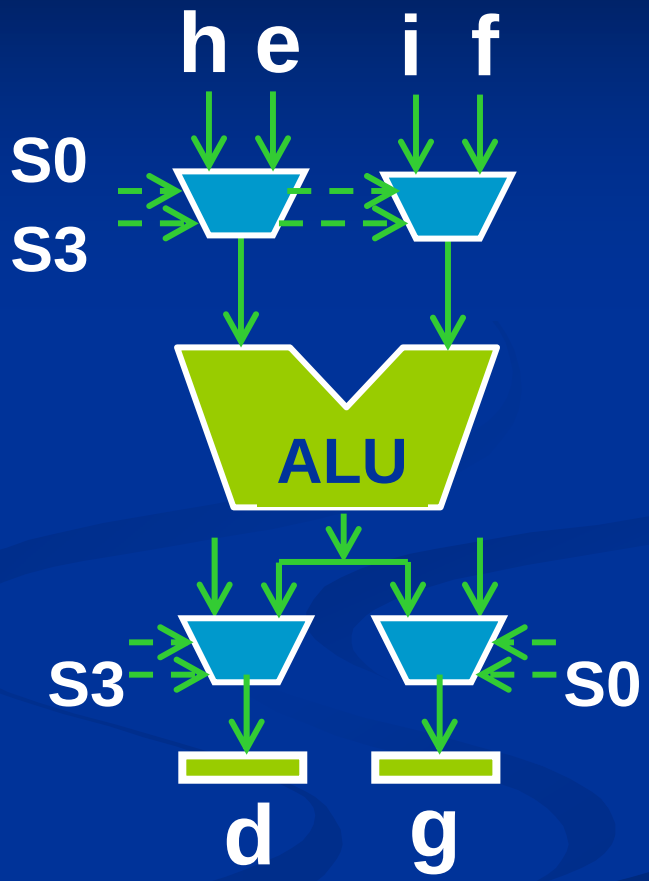
Hardware Costs of Speculative Code Motions: Speculation



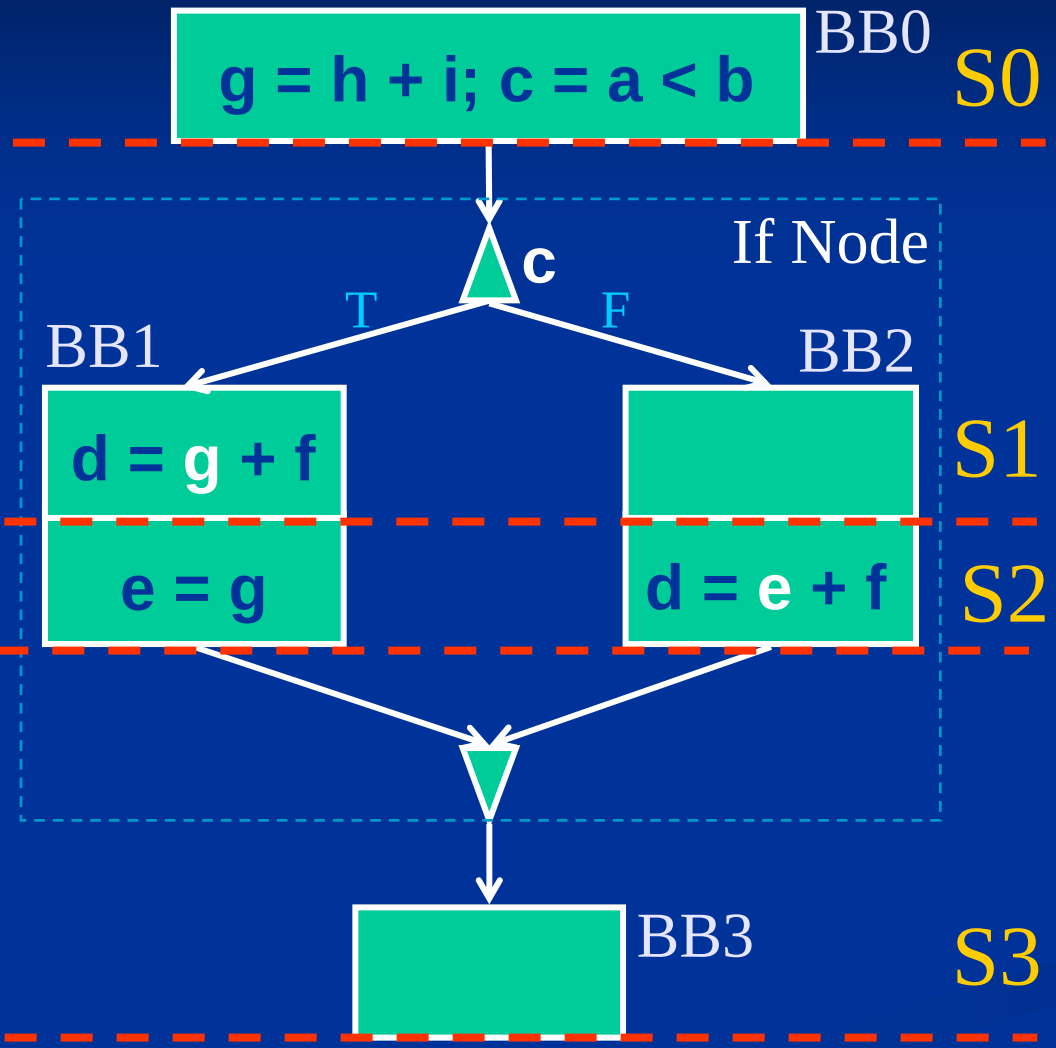


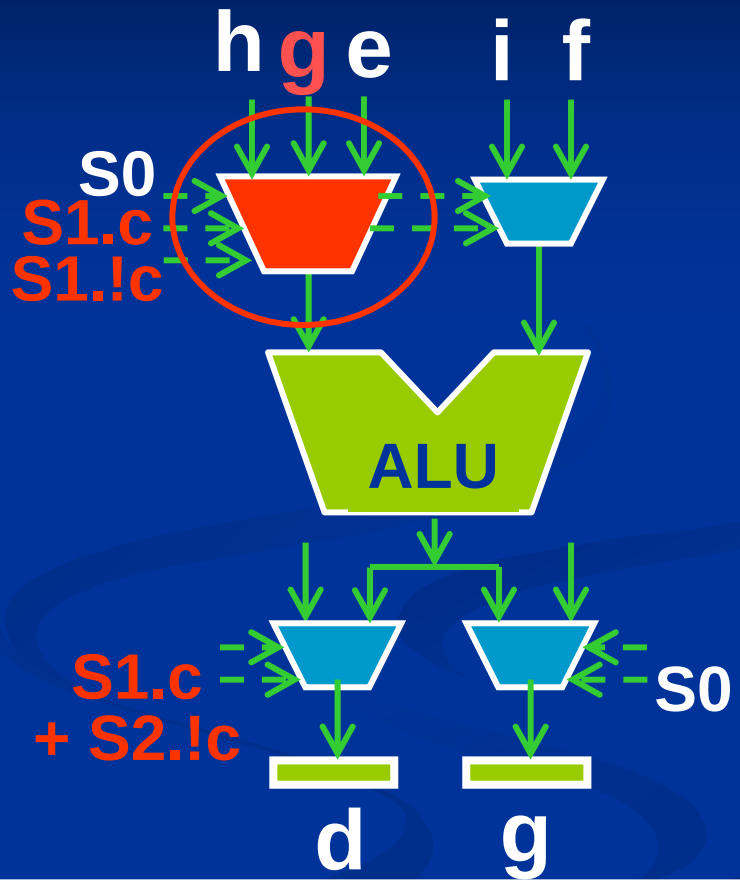
Hardware Costs of Speculative Code Motions: Conditional Speculation





Hardware Costs of Speculative Code Motions: Conditional Speculation

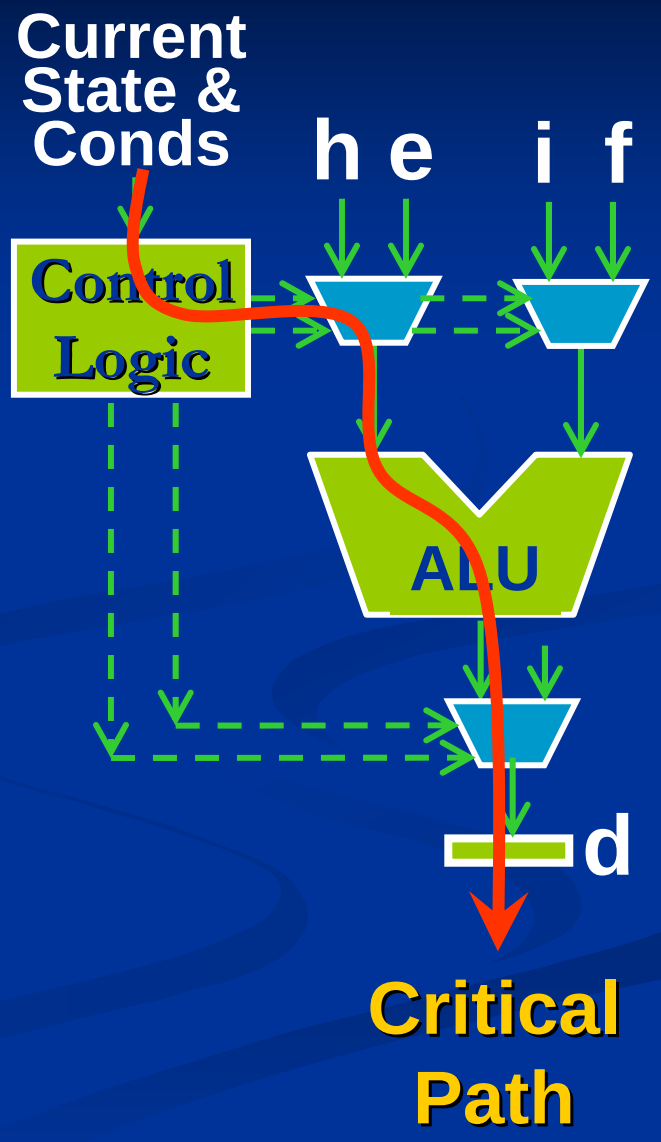




- Fewer Cycles but more complex Control and Multiplexing

Hardware Costs of Speculative Code Motions

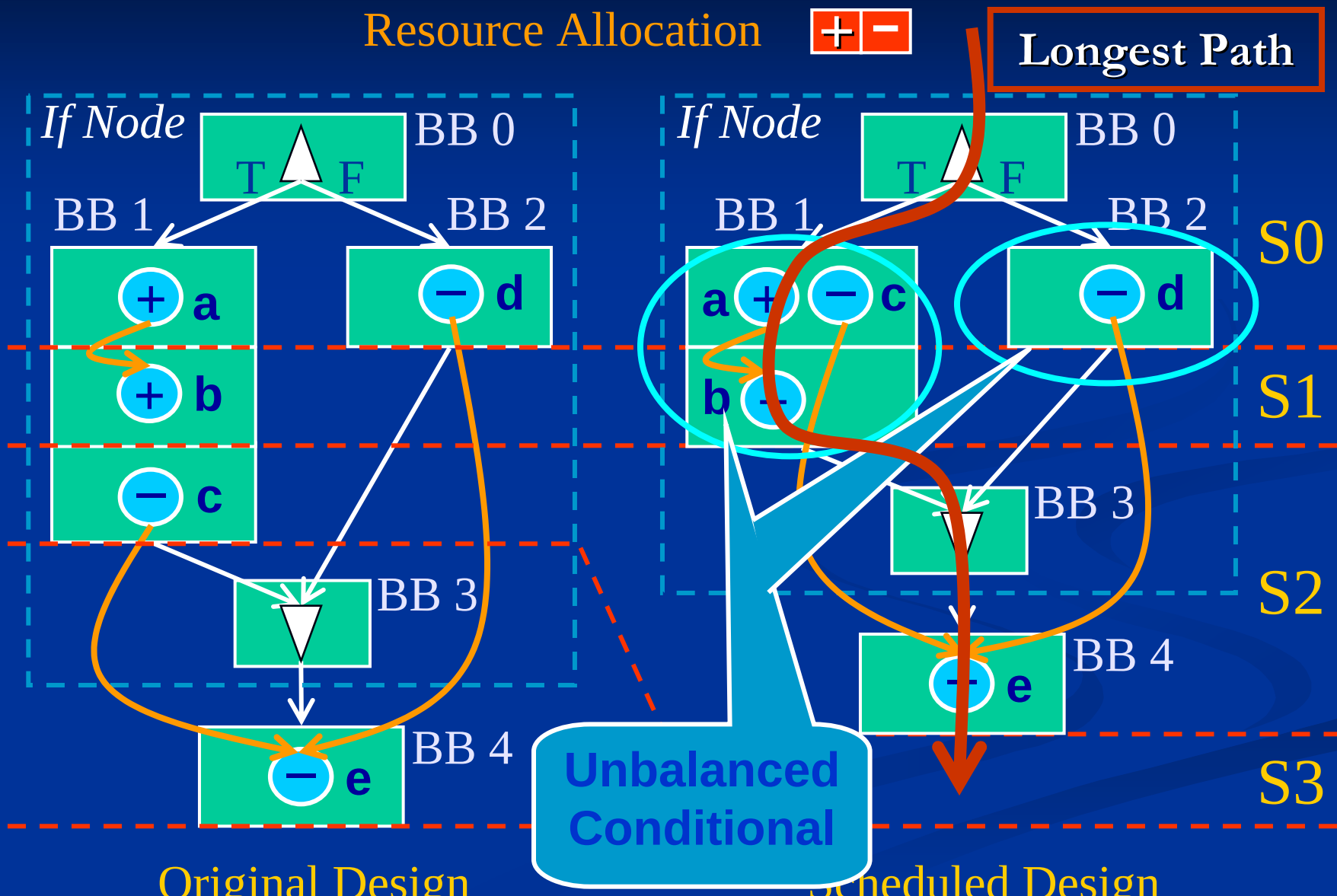
- Speculative code motions lead to two opposite effects on the control, multiplexing and area costs
  1. Shorter Schedule lengths
     - Leads to Smaller Controllers (fewer states in State Machine)
     - This leads to smaller area
  2. More multiplexing and control costs to steer the data
     - Particularly when operations are conditionally speculated
     - Leads to longer critical paths



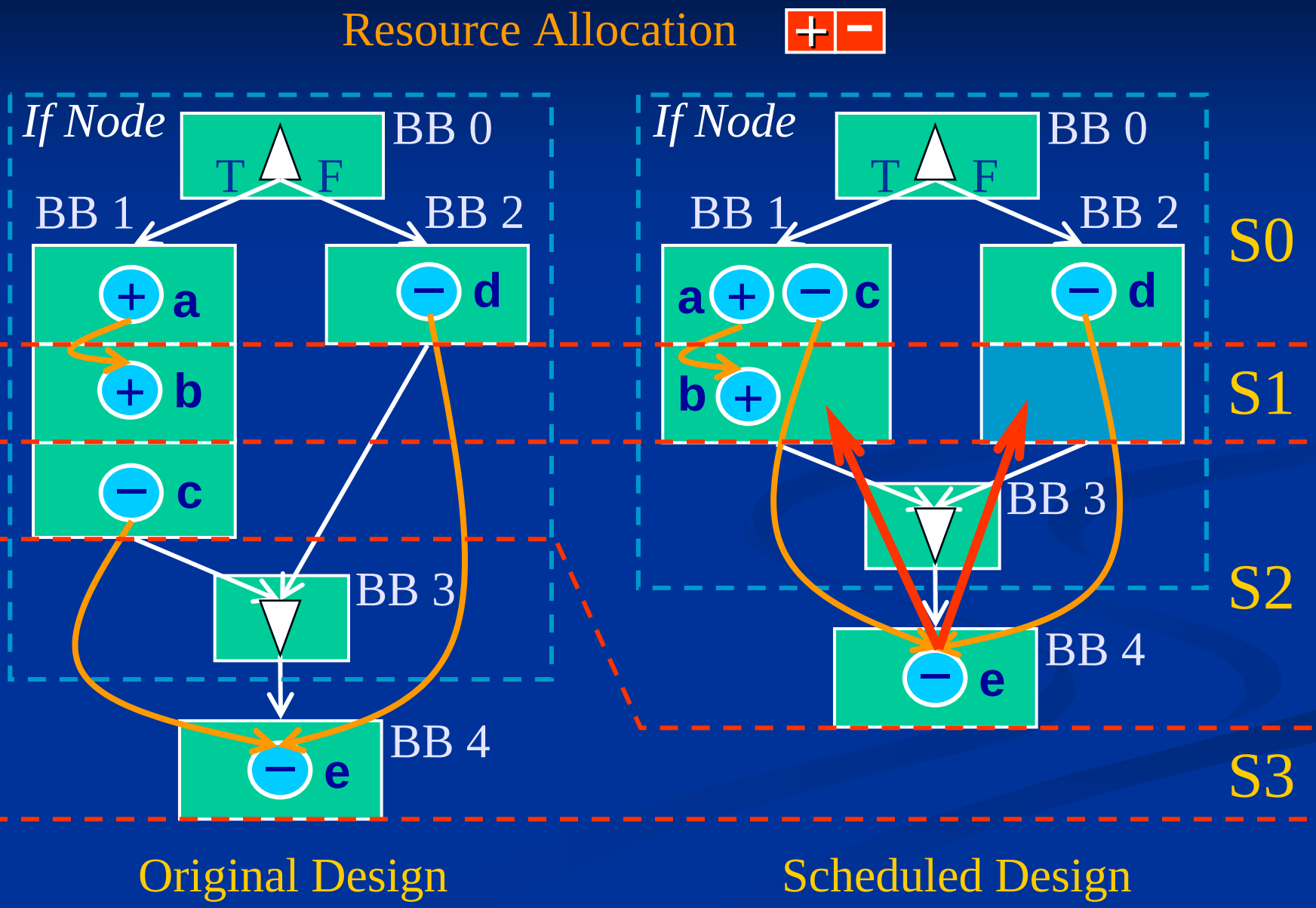
3. Dynamic Transformations

- 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

3. Dynamic Branch Balancing

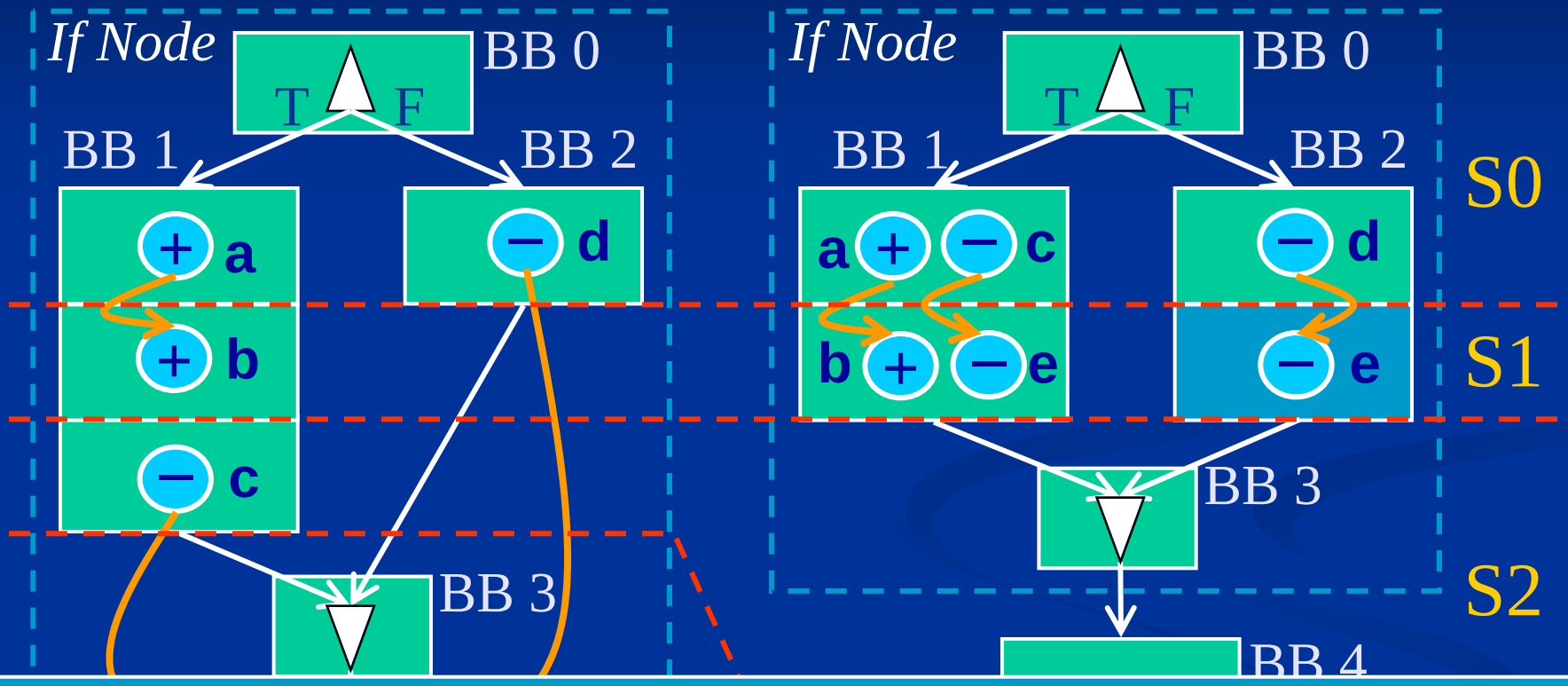


Insert New Scheduling Step in Shorter Branch



Insert New Scheduling Step in Shorter Branch

Resource Allocation 



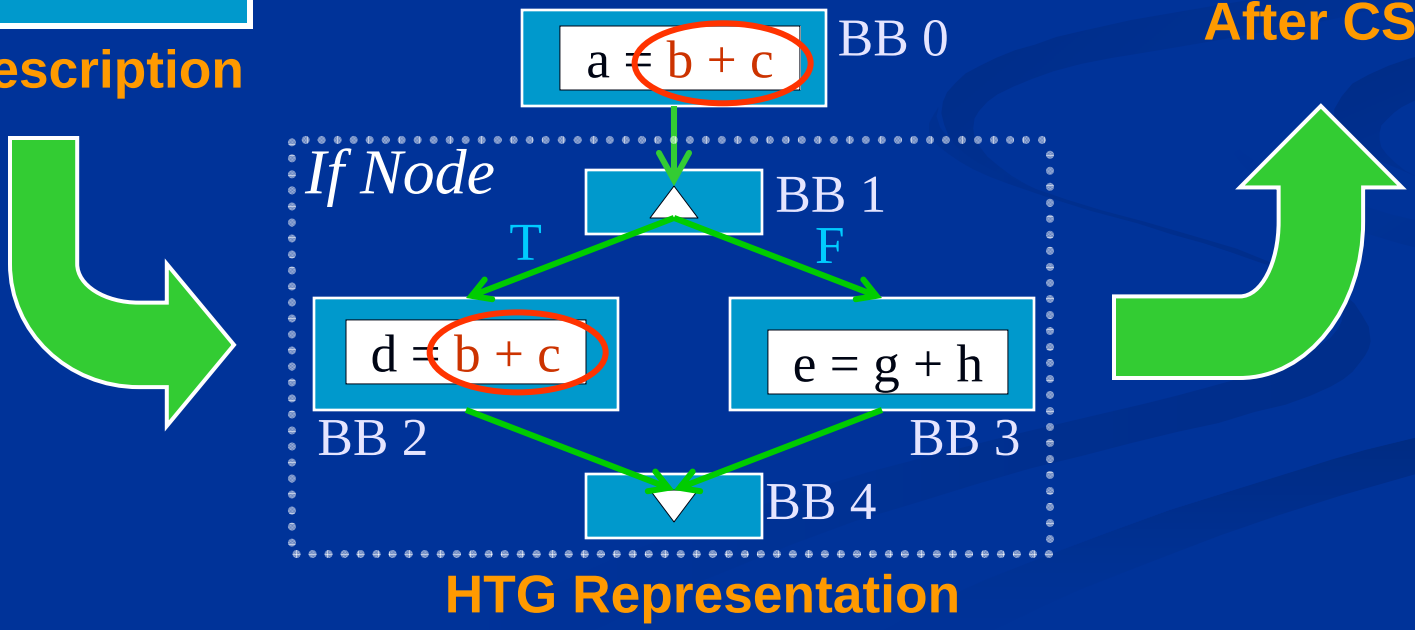
Dynamic Branch Balancing is done

1. **While Traversing the design**
2. **And if it enables Conditional Speculation**

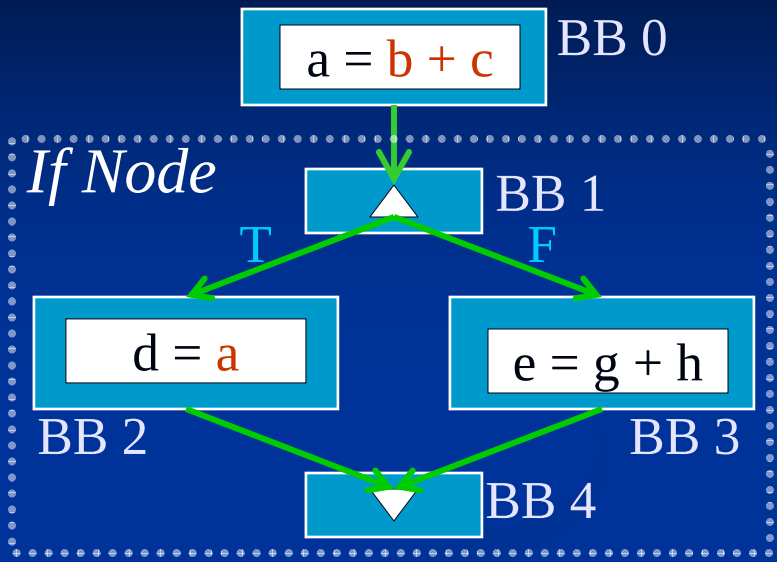
3. Dynamic CSE: Going beyond Traditional CSE

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

C Description

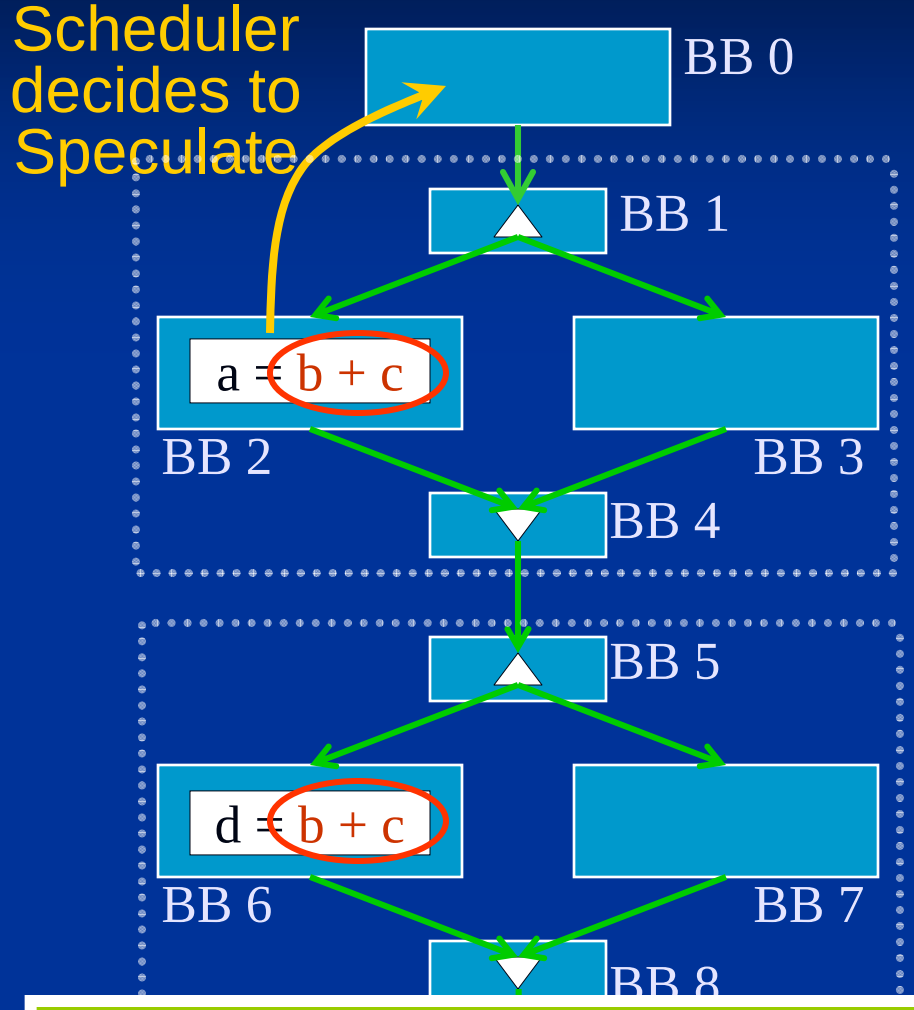


HTG Representation

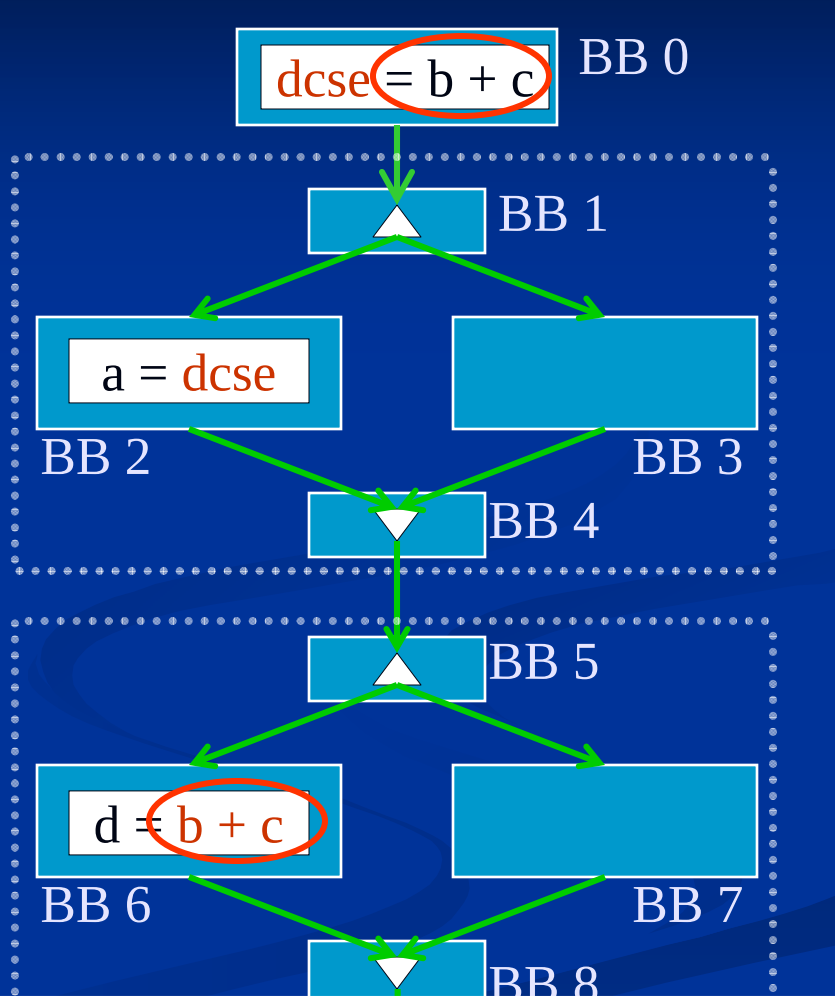


After CSE

New Opportunities for "Dynamic" CSE Due to Code Motions

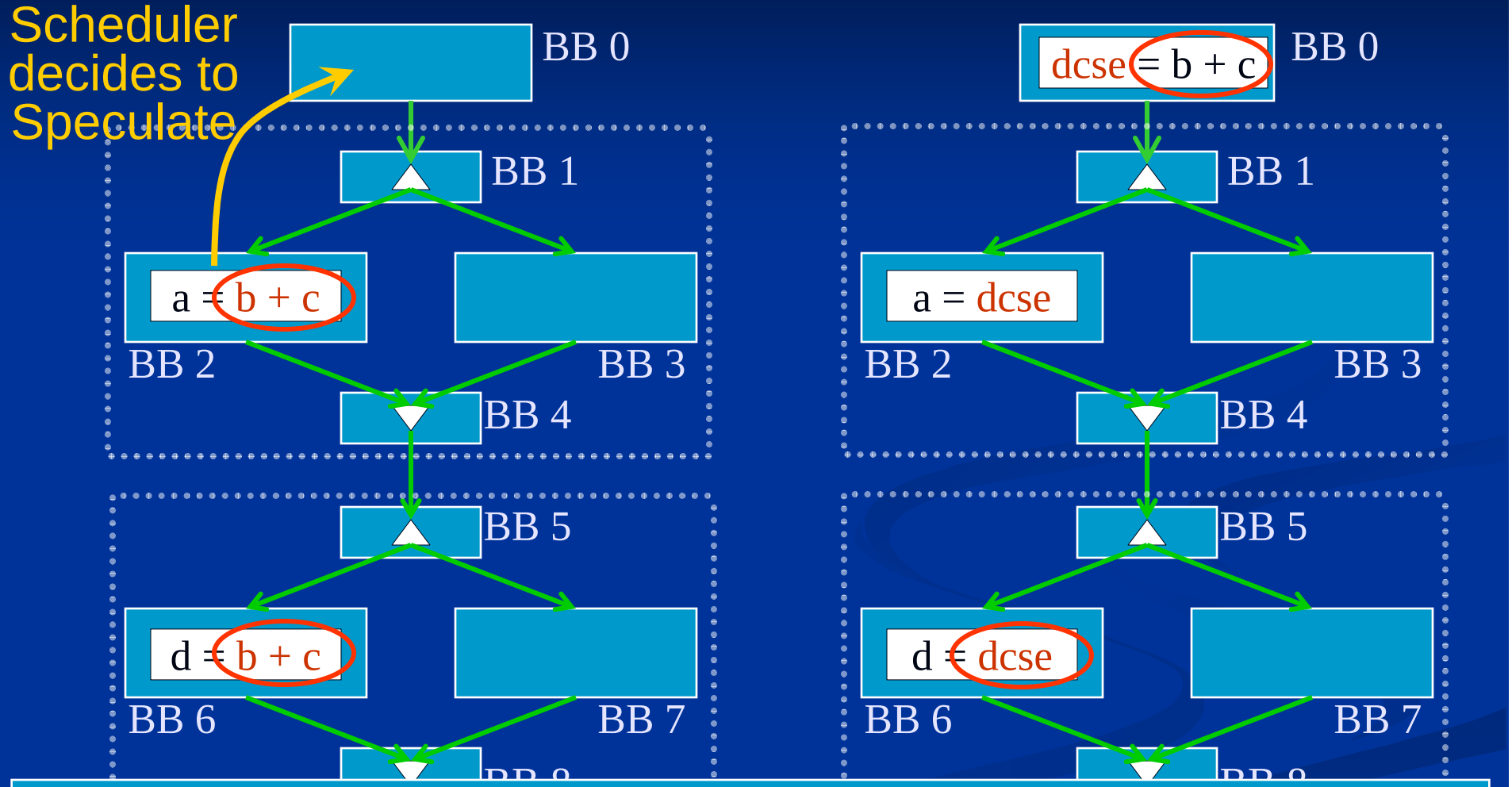


CSE not possible since BB2 does not dominate BB6



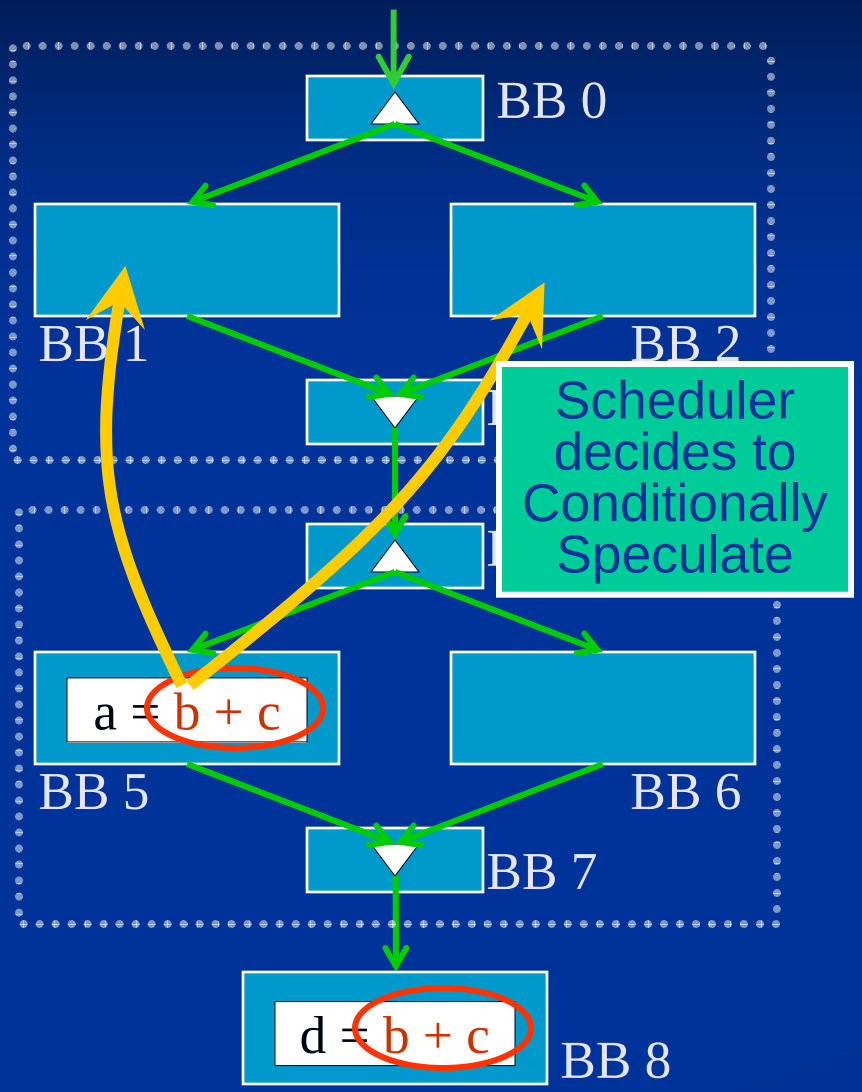
CSE possible now since BB0 does dominate BB6

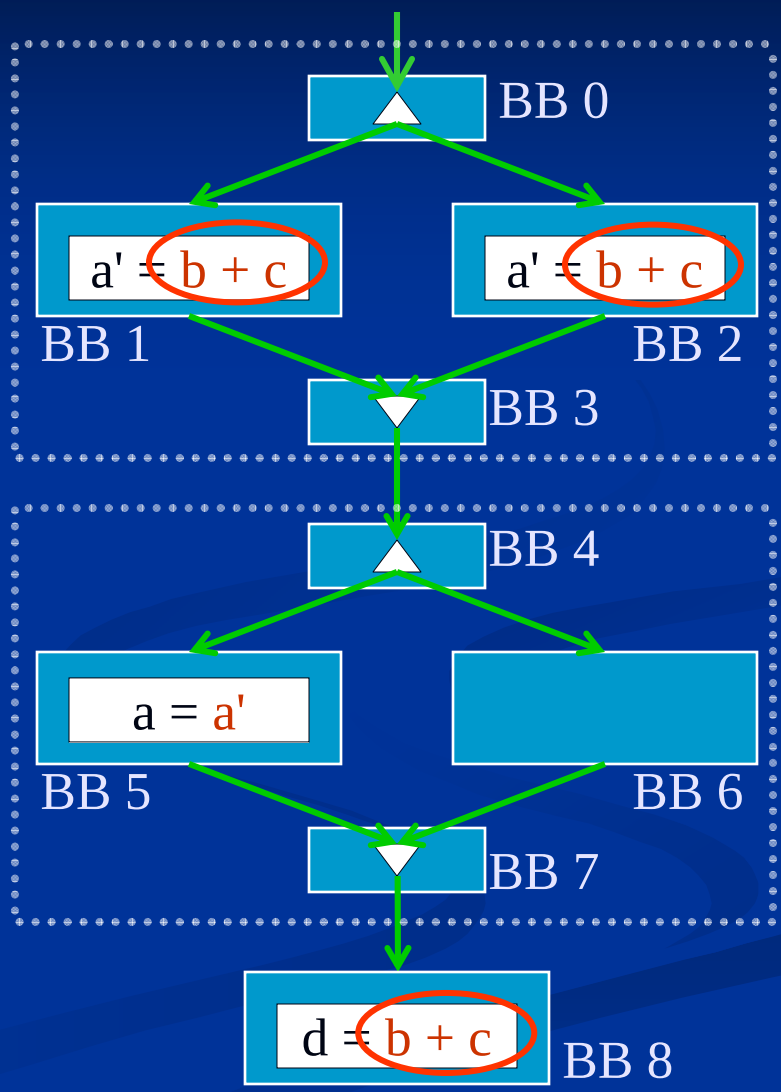
New Opportunities for "Dynamic" CSE Due to Code Motions



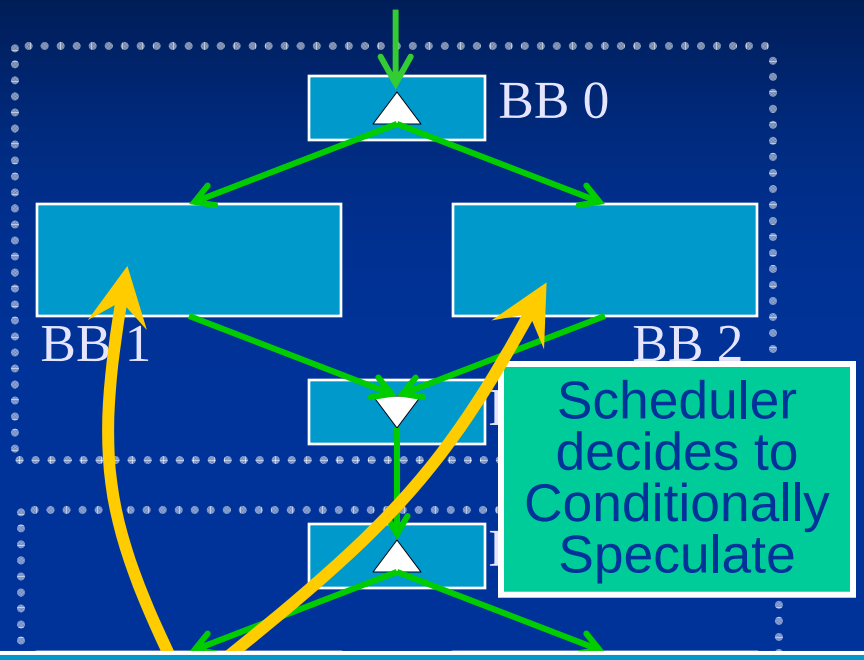
If scheduler moves or duplicates an operation op, apply CSE on remaining operations using op

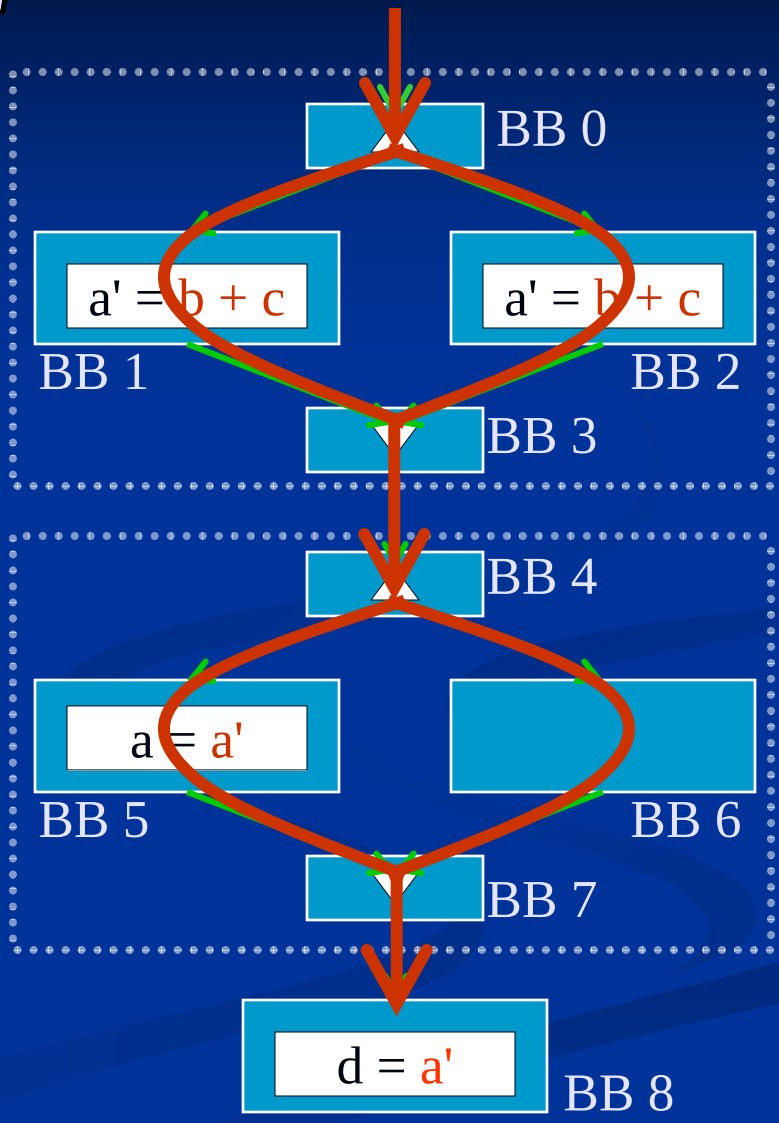
Condition Speculation & Dynamic CSE





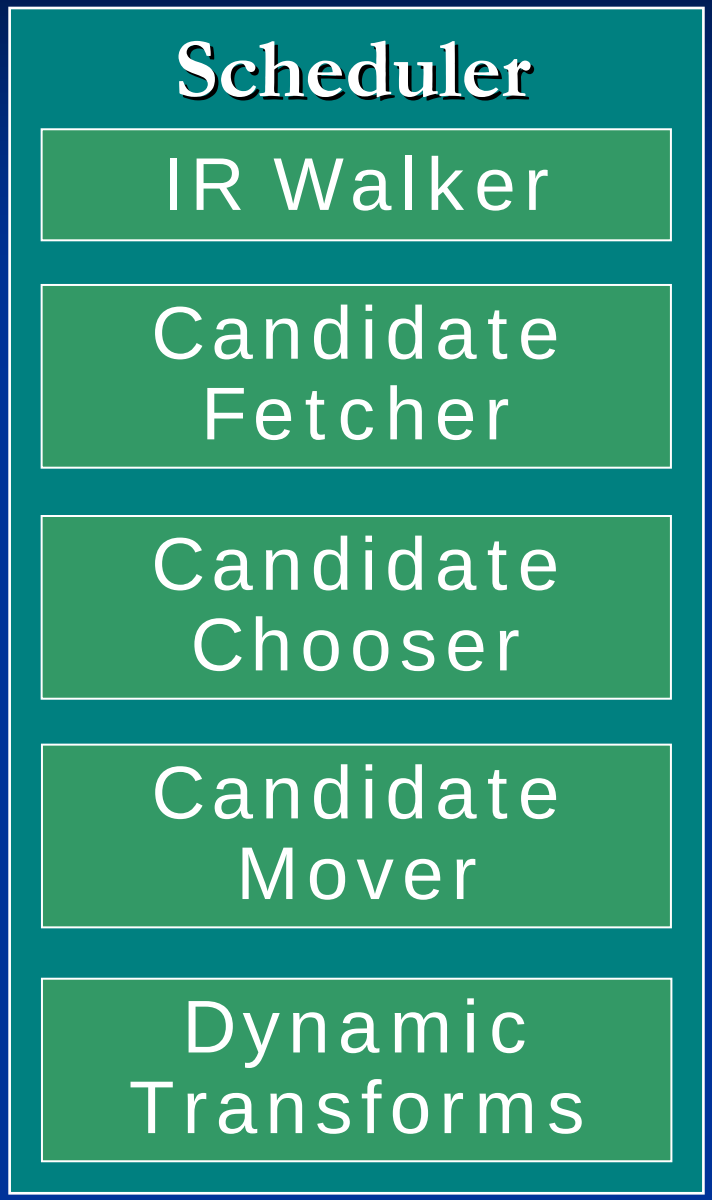
Condition Speculation & Dynamic CSE





- **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**

Architecture of the PHLS Scheduler



Traverses Design to find next basic block to schedule

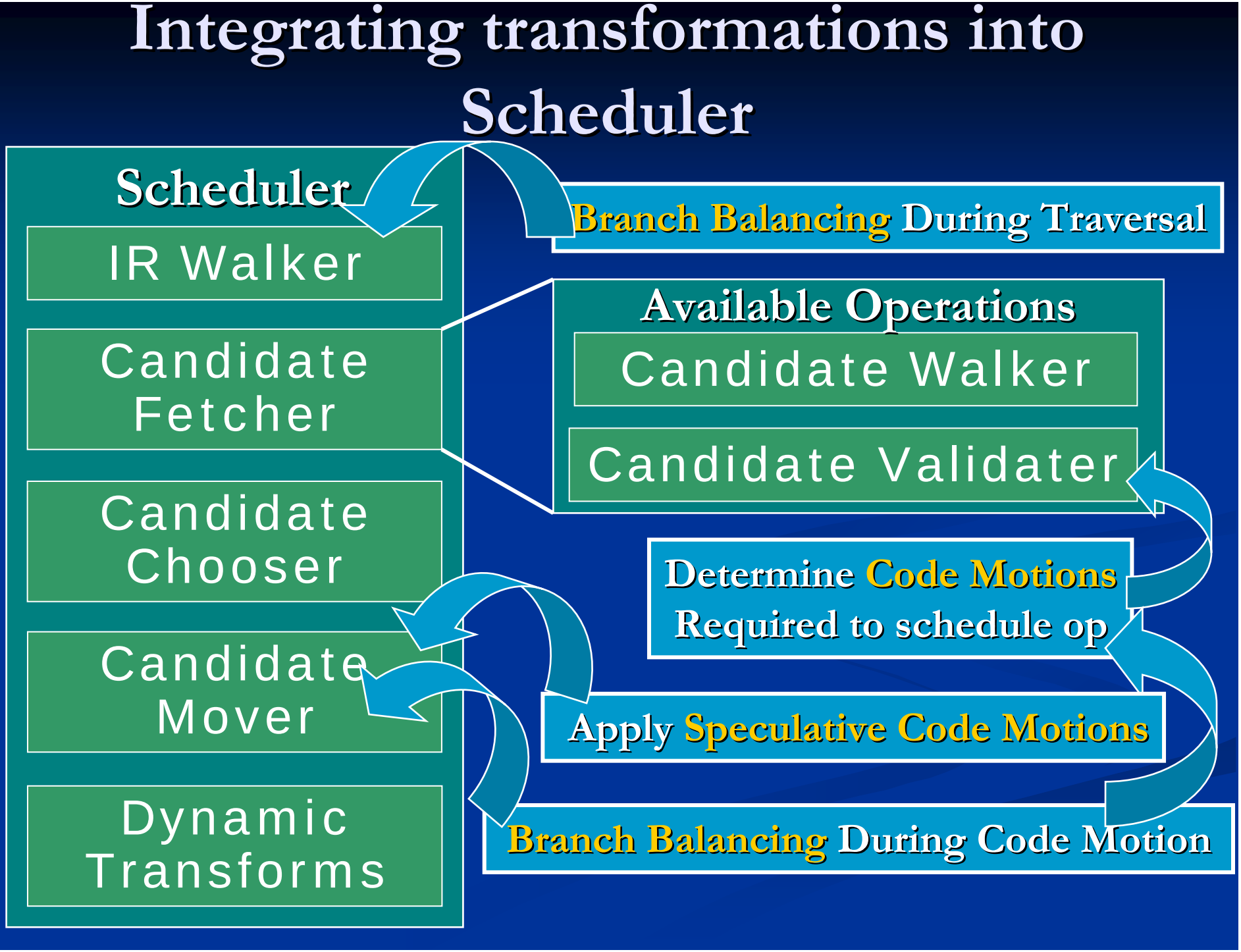
Traverses Design to find Candidate Operations to schedule

Calculates Cost of Operations and chooses Operation with lowest cost for scheduling

Moves, duplicates and schedules chosen Operation

Dynamically apply transformations such as CSE on remaining Candidate Operations using scheduled operation

Integrating transformations into Scheduler

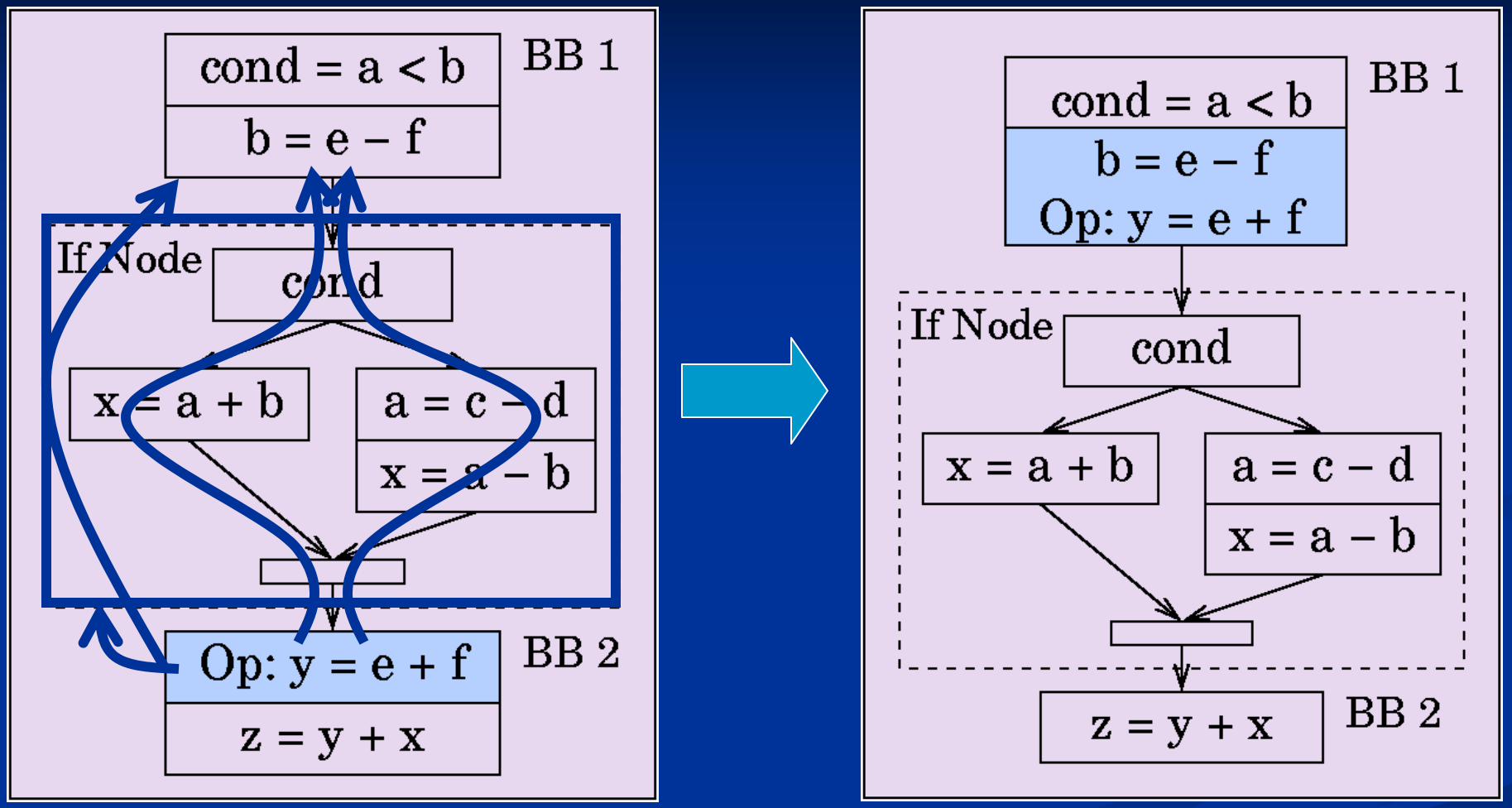


The Intermediate Representation

- Hierarchical Task Graph (HTG) is main structure in the intermediate representation (IR)

- Maintains information on:
  - Code structure (IFs, LOOPs)
  - Loop bounds, type (FOR, WHILE)
  - Array accesses are not lowered to address calculation followed by memory access
- Is complete
  - Can regenerate input C code

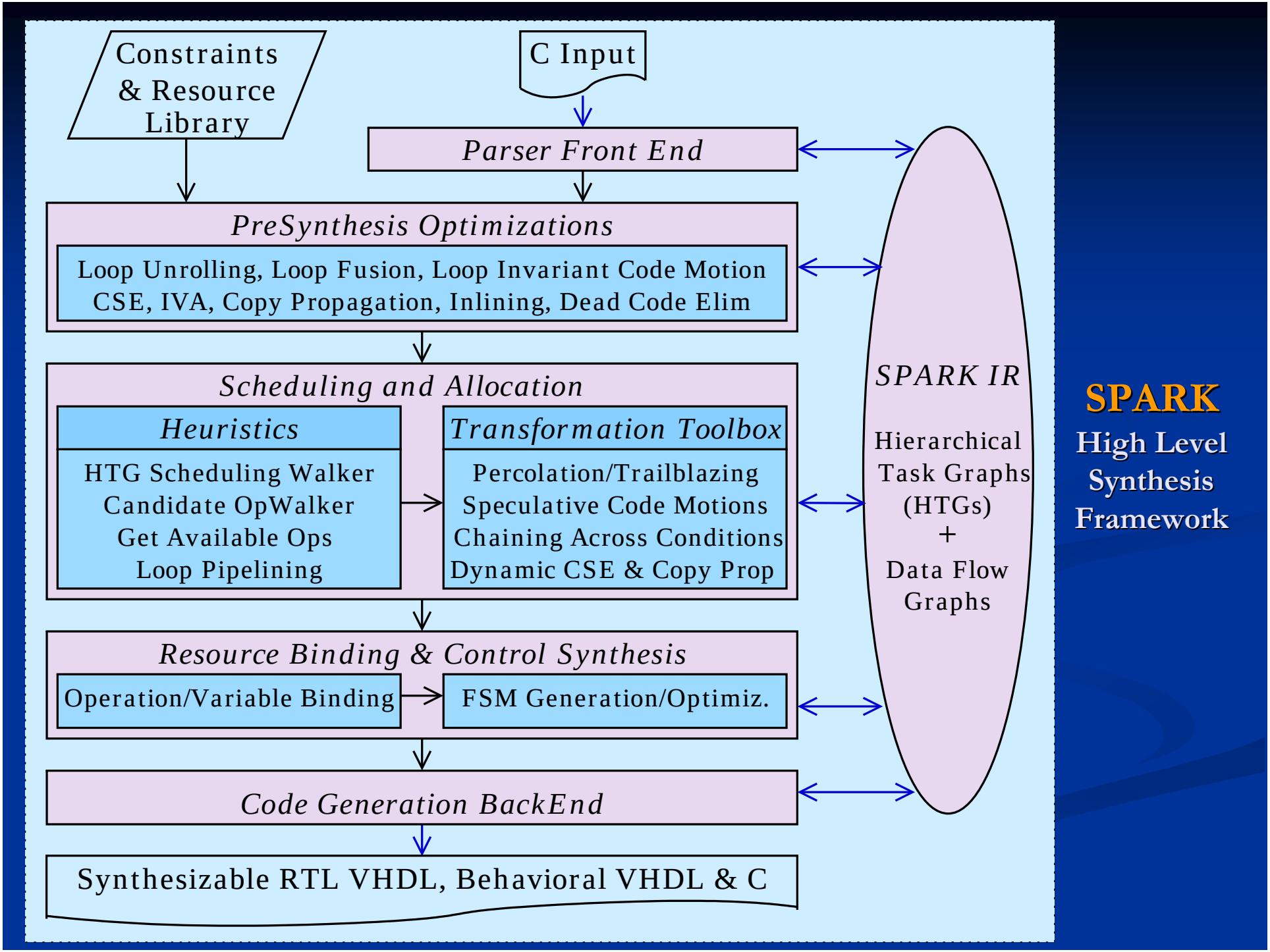
Use HTG for Hierarchical Code Motions



- Can move operations across large pieces of code without visiting each node in between

SPARK Parallelizing HLS Framework

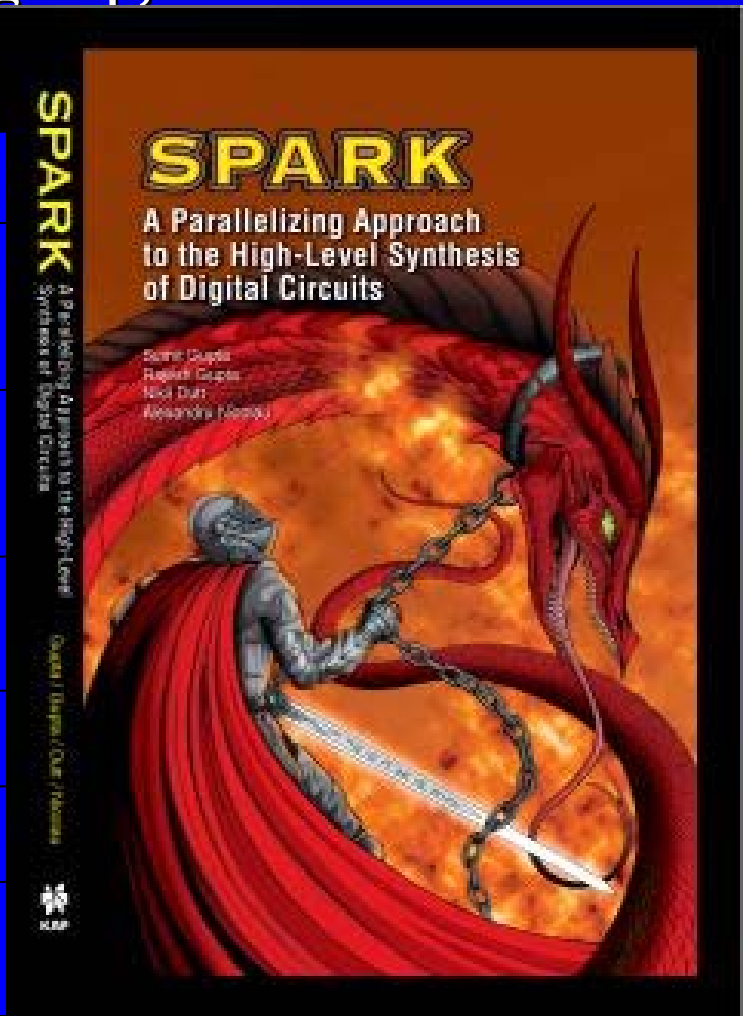
- C input and Synthesizable RTL VHDL output
- Range of compiler, parallelizing compiler and HLS transformations applied during Pre-synthesis and Scheduling phases
- Tool-box of Transformations and Heuristics
  - Each of these can be developed independently of the other
- Complete HLS tool: Does Binding, Control Synthesis and Backend VHDL generation
  - Interconnect Minimizing Resource Binding
- Enables Graphical Visualization of Design description and intermediate results
- About 100,000 + lines of C++ code


Constraints & Resource Library
C Input
Parser Front End
PreSynthesis Optimizations
Loop Unrolling, Loop Fusion, Loop Invariant Code Motion
CSE, IVA, Copy Propagation, Inlining, Dead Code Elim
Scheduling and Allocation
Heuristics
HTG Scheduling Walker
Candidate OpWalker
Get Available Ops
Loop Pipelining
Transformation Toolbox
Percolation/Trailblazing
Speculative Code Motions
Chaining Across Conditions
Dynamic CSE & Copy Prop
Resource Binding & Control Synthesis
Operation/Variable Binding
FSM Generation/Optimiz.
Code Generation BackEnd
Synthesizable RTL VHDL, Behavioral VHDL & C
SPARK IR
Hierarchical Task Graphs (HTGs)
+
Data Flow Graphs
SPARK
High Level Synthesis Framework

Status

- **SPARK released in September 2003**
  - Over 4000 downloads, Spark users group, one licensee

Release	Ver	Download	Change Log
05-20-05	1.3	Linux (Built on Gentoo) Windows (built on XP)	Version 1.2 to 1.3
2-5-04	1.2	Linux (Redhat 9.0) Solaris 5.7 Windows (built on XP)	Version 1.1 to 1.2
1-12-04	1.1	Linux (Redhat 9.0) Solaris 5.7 Windows XP	Version 1.0 to 1.1
12-29-03	1.0	Linux(Redhat 9.0) Solaris 5.7	Version 0.9 to 1.0
11-10-03	0.9	Linux(Redhat 9.0) Solaris 5.7	Version 0.8 to 0.9
09-19-03 10-09-03	0.8	Linux(Redhat 9.0) Solaris 5.7	None Fixed some file name problems



Experiments

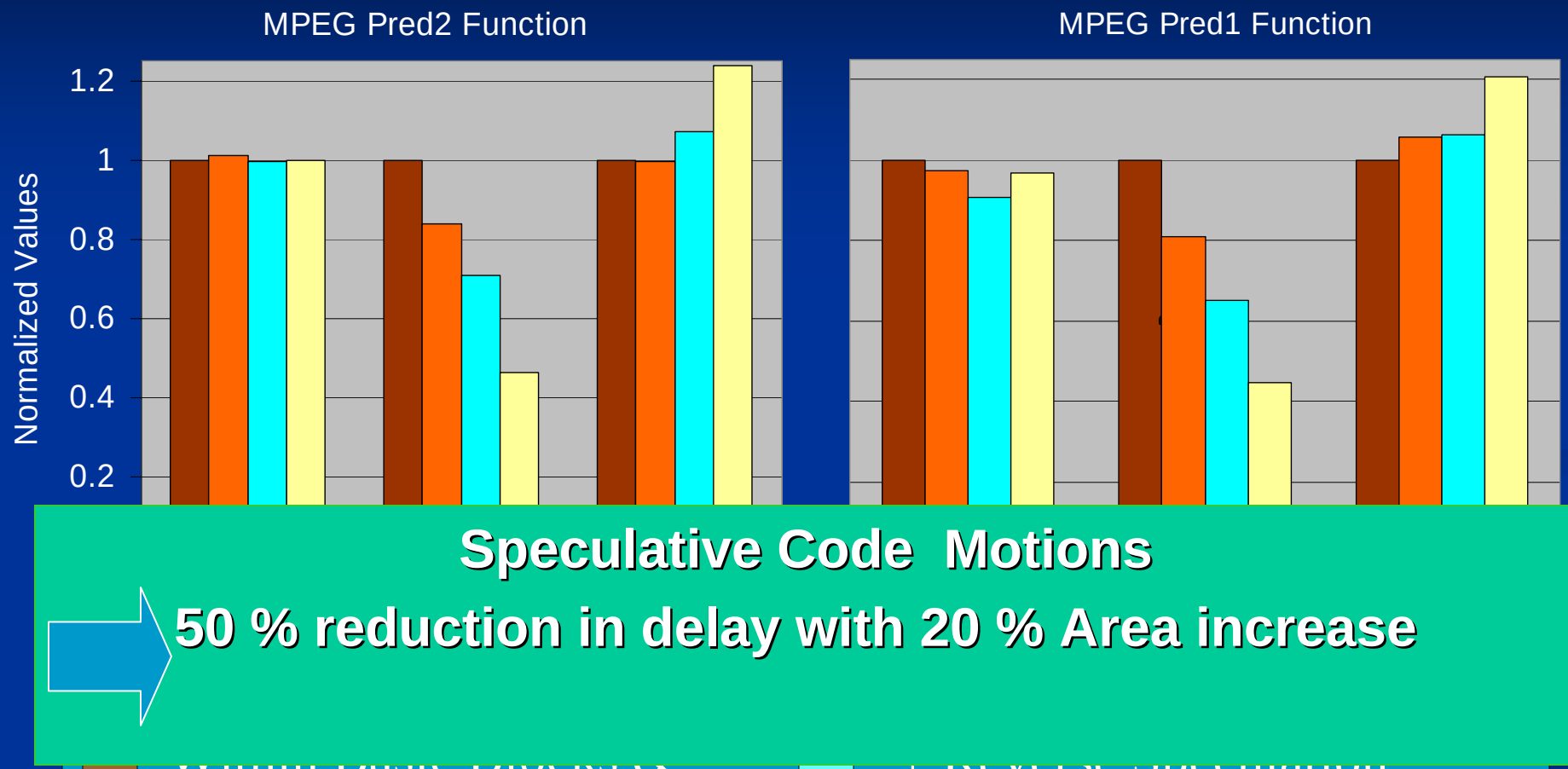
- Experiments for several transformations
  - Pre-synthesis transformations: loop invariant code motions, CSE
  - Speculative Code Motions
  - Dynamic CSE
- We have used Spark to synthesize designs derived from several industrial designs
  - MPEG-1, MPEG-2, GIMP Image Processing software
  - Case Study: 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

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

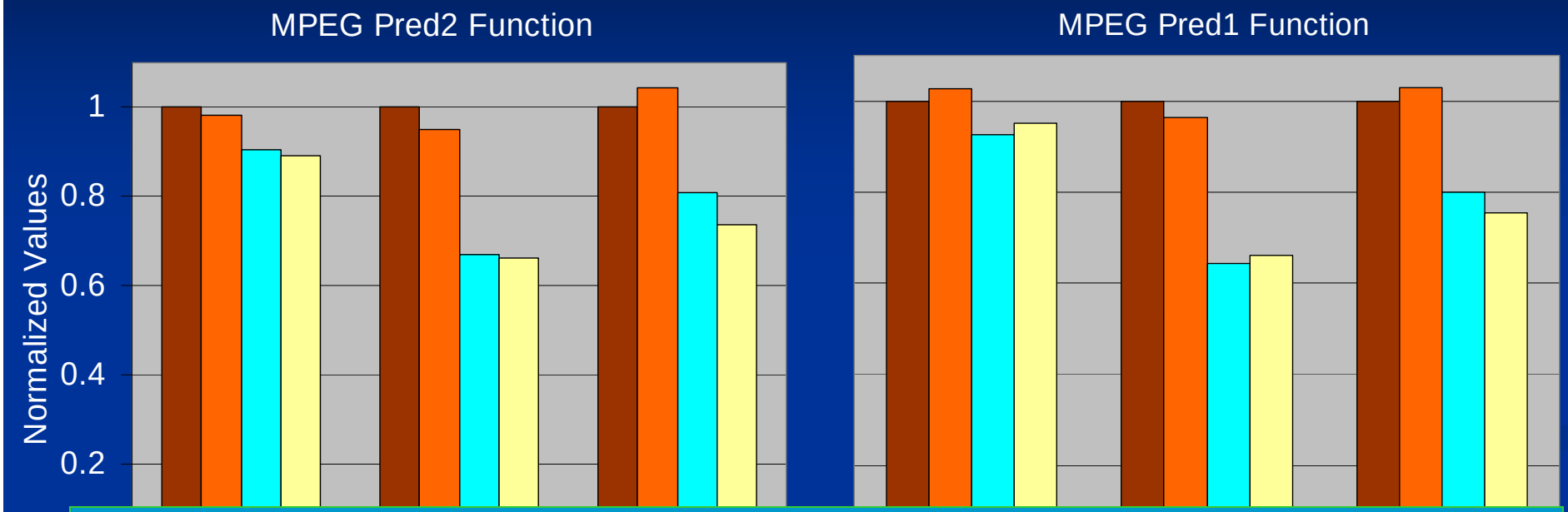
Code Motions: Logic Synthesis Results



Speculative Code Motions

50 % reduction in delay with 20 % Area increase

CSE/Dynamic CSE Results



Speculative Code Motions + Dynamic CSE

75 % reduction in delay with No Area increase

30 % reduction in delay, 25 % reduction in Area

Feedback

---------- Forwarded message ----------

SPARK is readily used here. Rather all the applications we test/demonstrate, invariably needs SPARK. Following is the description of how it fits into with *** Tool chain. *** Tools partition the hot spots of the application and place them in a generated hardware accelerator unit. The hardware accelerator unit has components like a slave bus system, DMA controllers, Controller Units etc, but it was missing the actual compute unit. The designers here were manually writing it. Now SPARK is being used for this task. We feed the C hot loops into the SPARK and get VHDL and Verilog RTL for the compute unit. SPARK has to be changed quite a bit to use it the way we want. To give you an estimate: One engineer modified it to make it work and use-able for us. One engineer worked to get Verilog code out of it. And I am generating the hardware interface. You can say that 3 resources are working on it since last 8 months at least. ---------