High-Level Design: (Yet) Another Look

Rajesh Gupta

Department of Computer Science and Engineering,

University of California, San Diego

Sudipta Kundu and Sorin Lerner

A Tale of Two Consequence

1. EDA: Raise abstractions
   - Raising abstraction has always been part of the solution strategy to lower design costs.
   - In design modeling, design synthesis, design verification
2. Architecture: Raise programmability
   - Holy Grail: ASIC efficiency with CPU programmability.
   - The tremendous space of architectural innovations between ASIC and FPGA

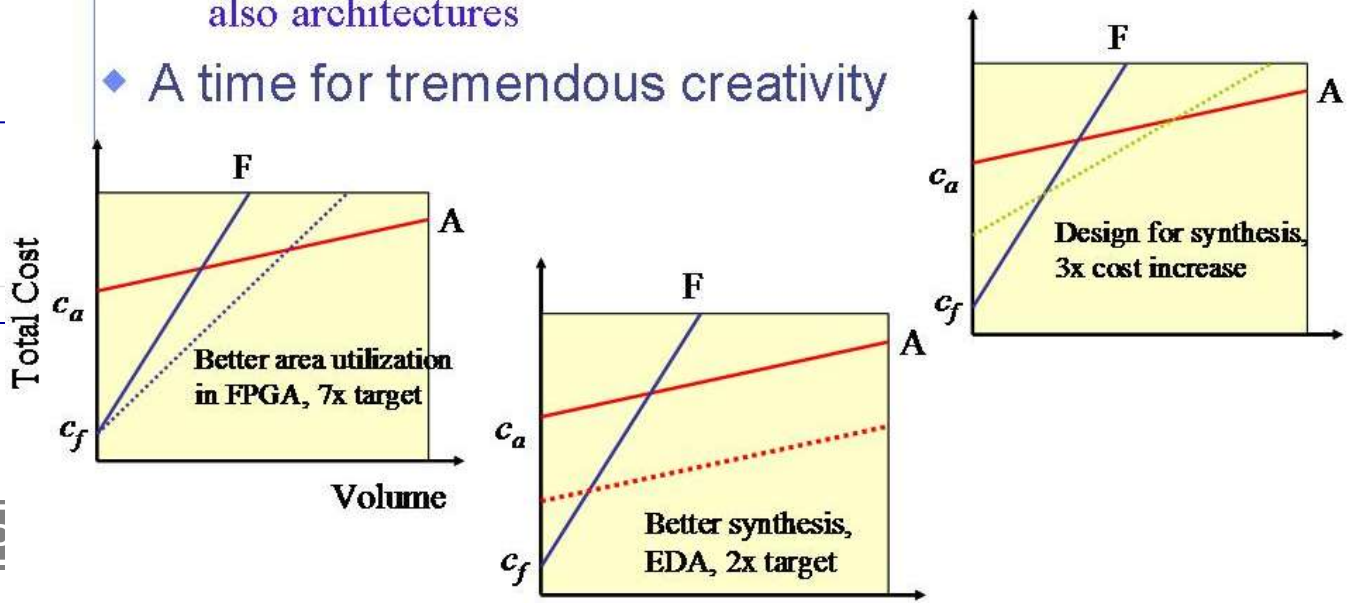
Coming back to EDPS!

Closing Thoughts

- ASIC design cost is the new driver
  - Solution space is expanded to include not only tools but also architectures
- A time for tremendous creativity

Cost of Design

EDPS 2008



Points I want to make today

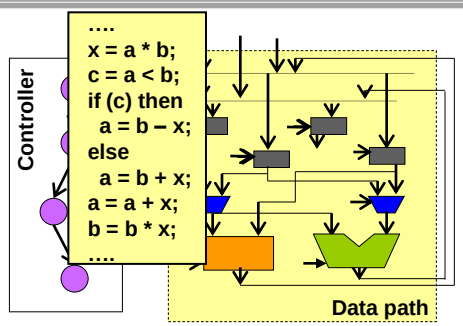
- Inexorable move to the high-level
- Verification must lead the way
- Hardware formals need to be tempered by software pragmatists
- We are making progress in HLV.

High-Level Design

Algorithmic Design

Register Transfer

High-Level Synthesis

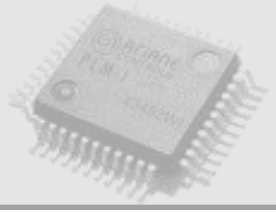


C/C++, SystemC
⟨100 – 10K lines⟩
Verilog, VHDL
⟨1K – 100K lines⟩

- **Challenge:**
  - Cope with the growing size and heterogeneity.

    ⇒ Writing RTL designs more complex, tedious, and error-prone.

Application Specific Integrated Circuit (ASIC)



HLD = HLS + HLV ?

- User accessibility vs. Quality of Results (QOR)
  - Synthesis goal: reduce designer effort within acceptable QOR
  - Verification goal: increased effort acceptable for high QOR
- HLS: compiler?, PL?
  - In reality, it is a verification problem
    - Checking product isn't easy: state explosion in MC, path explosion in stateless MC, specification & solver challenges in implicit/symbolic state checkers (BMC)…
- Step back and look at the problem again…
  - Verify both design process and product together.
  - Pay attention to modularity (even at the cost of QOR).

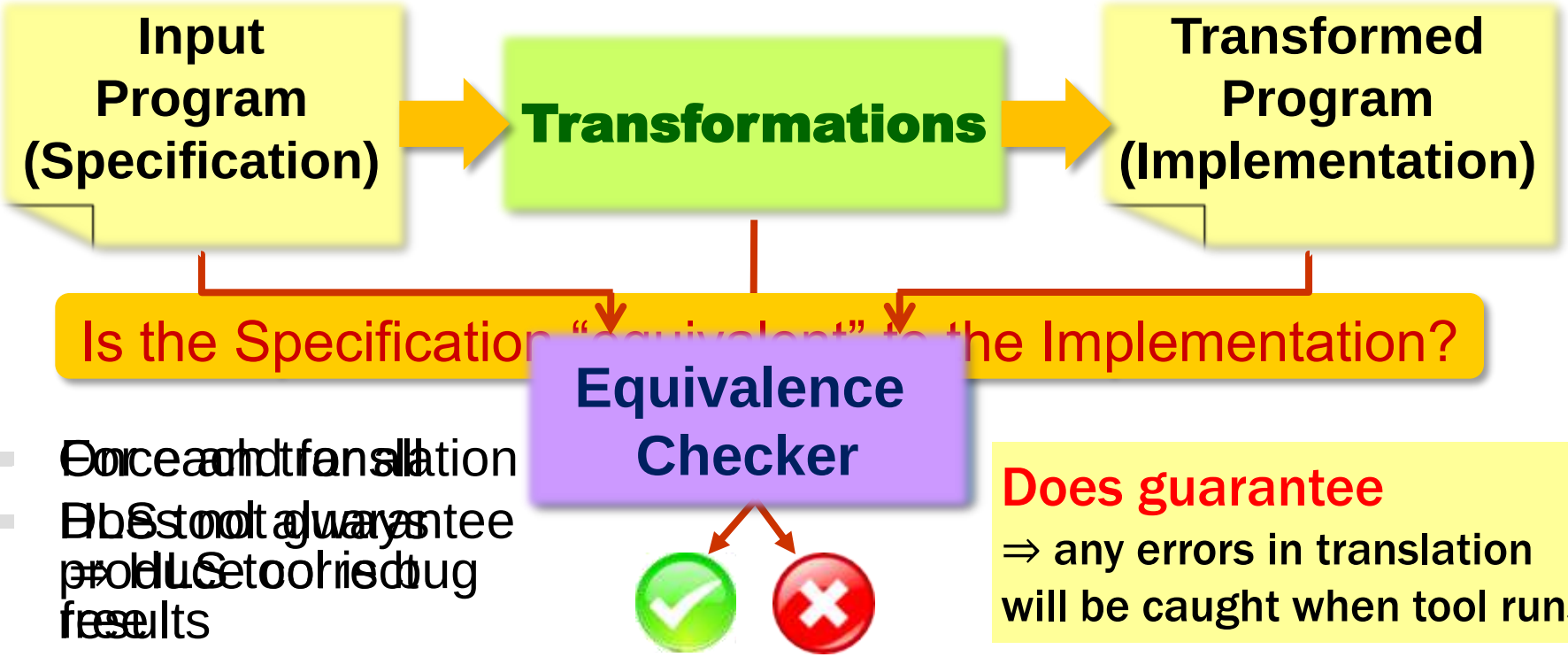
Our recent work in the area: 4 tools

- Explore various scalable and automatic techniques for high-level verification.

- Translation Validation (ICCAD07, CAV08)
- Parameterized Equivalence Checking (PLDI 09)
- Execution-based Model

- **Benefits**
  - Find bugs early in the design phase.
  - Guarantee that the properties checked hold in the RTL design.

Translation Validation (TV)



- For each translation
- Does not guarantee product is bug free

Does guarantee
⇒ any errors in translation will be caught when tool runs

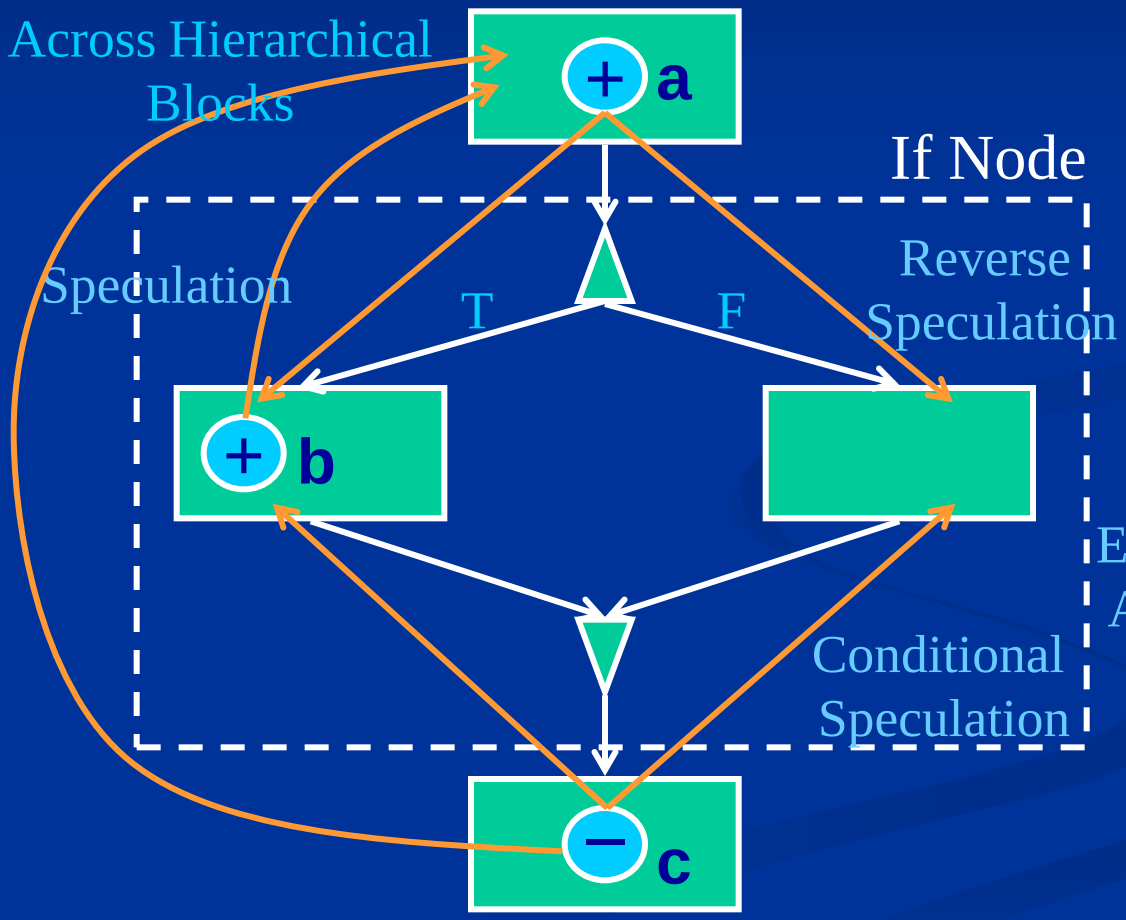
- Check the tool or the product.

Close the loop: TV in HLS

- Parallelizing High-Level Synthesis: SPARK
  - Widely used: 4,000 downloads, over 100 active users.
  - Moderately large software: around 125, 000 LoC.

Speculative Code Motions

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

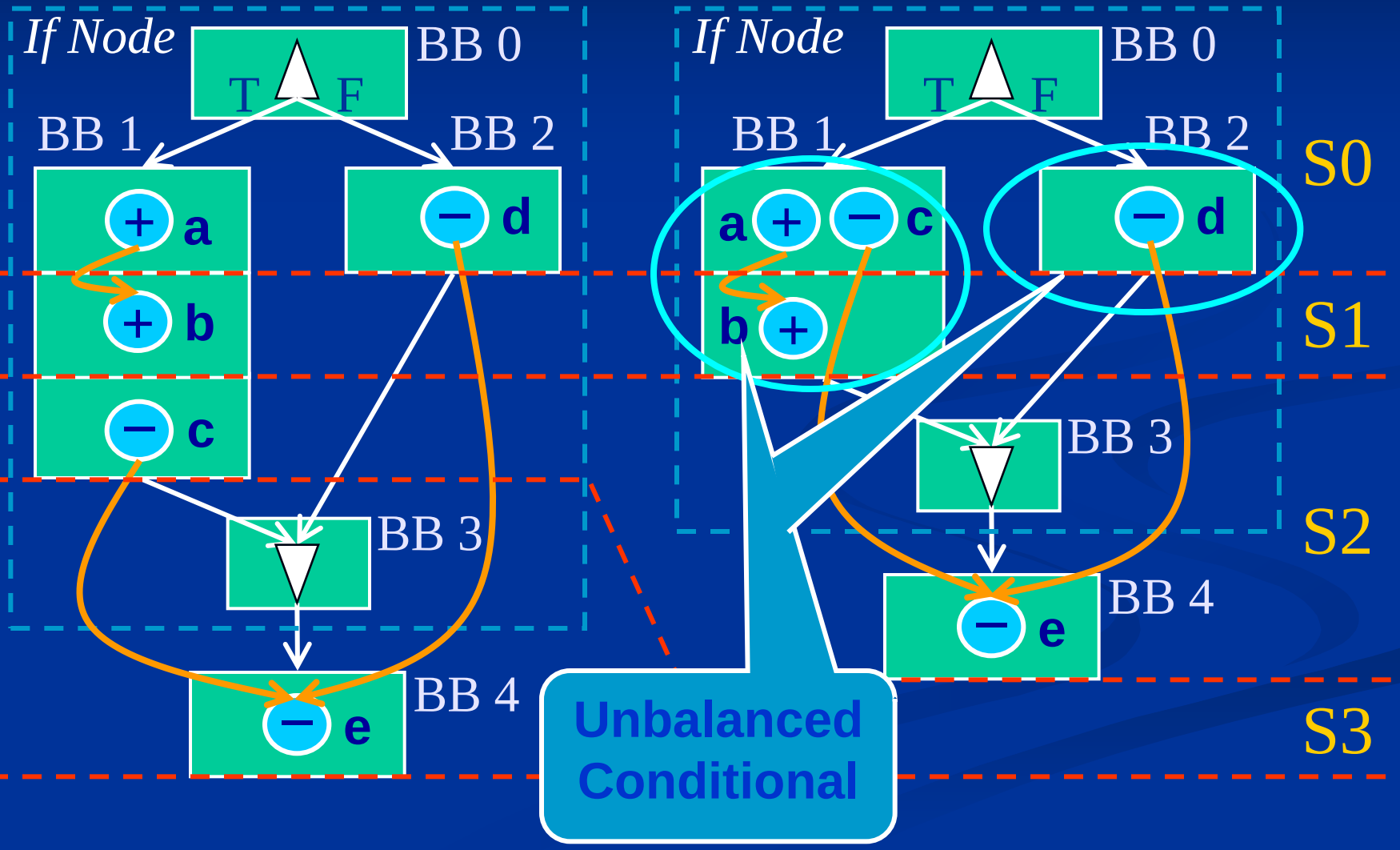


Early Condition Execution

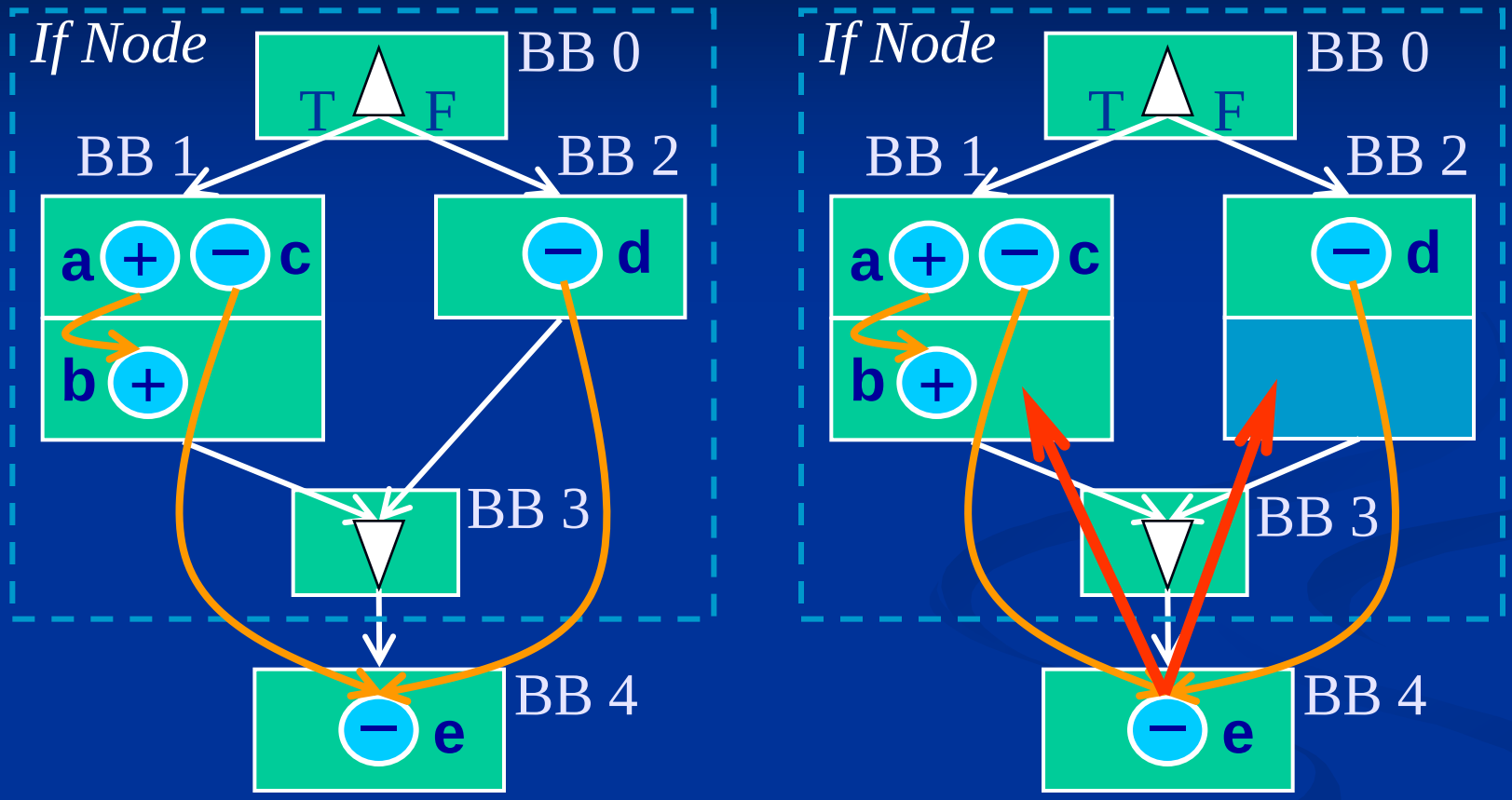
Evaluates conditions As soon as possible

Increasing the scope of Code Motions by Inserting New Scheduling Steps

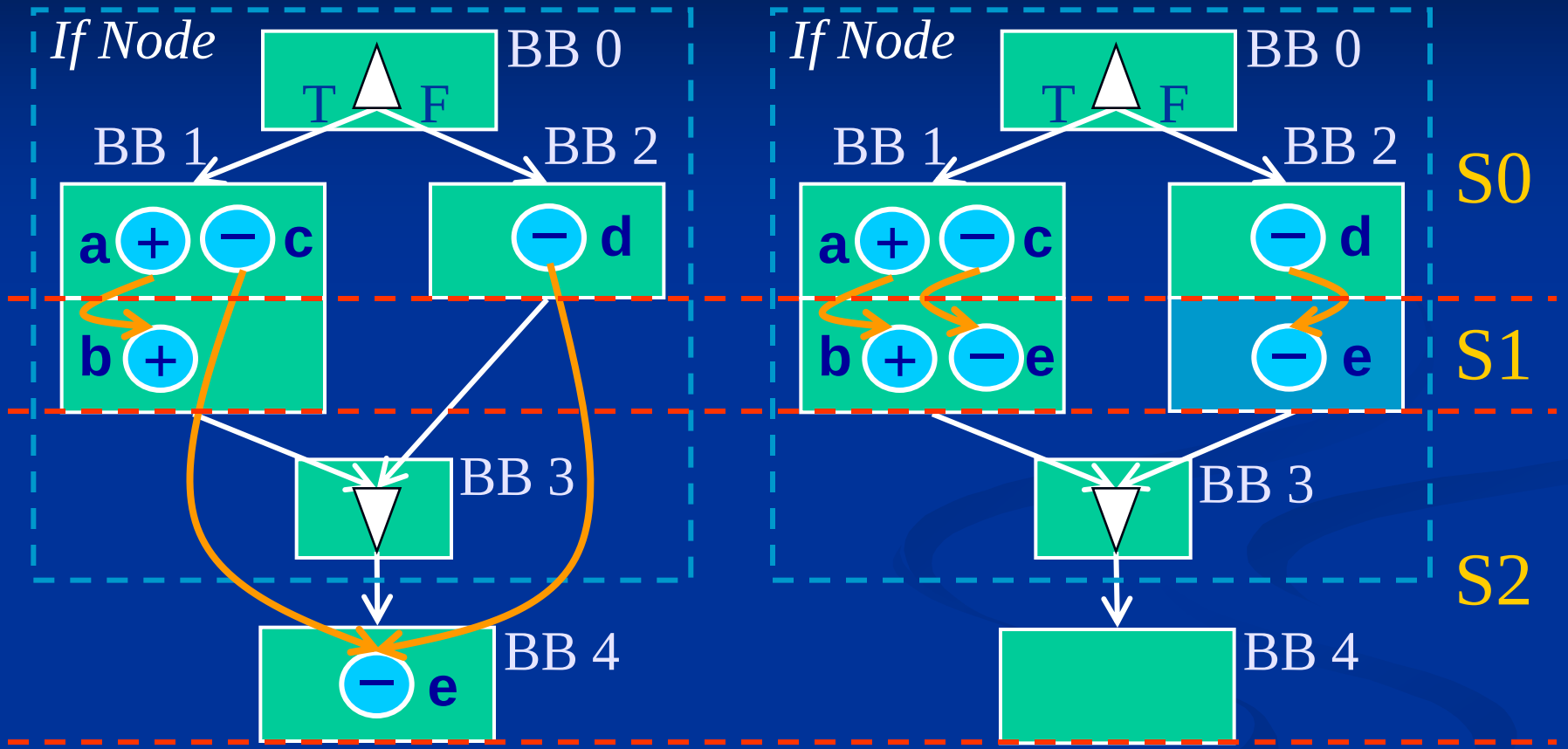
Resource Constraints 



Inserting New Scheduling Steps

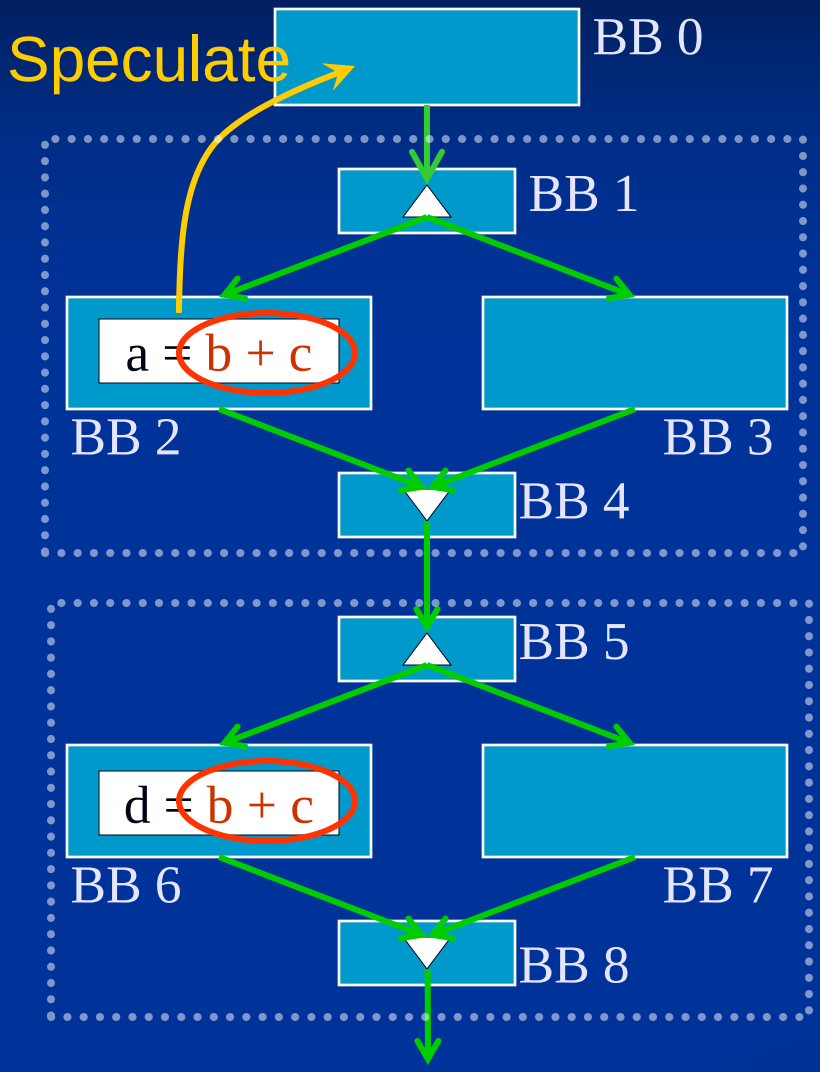


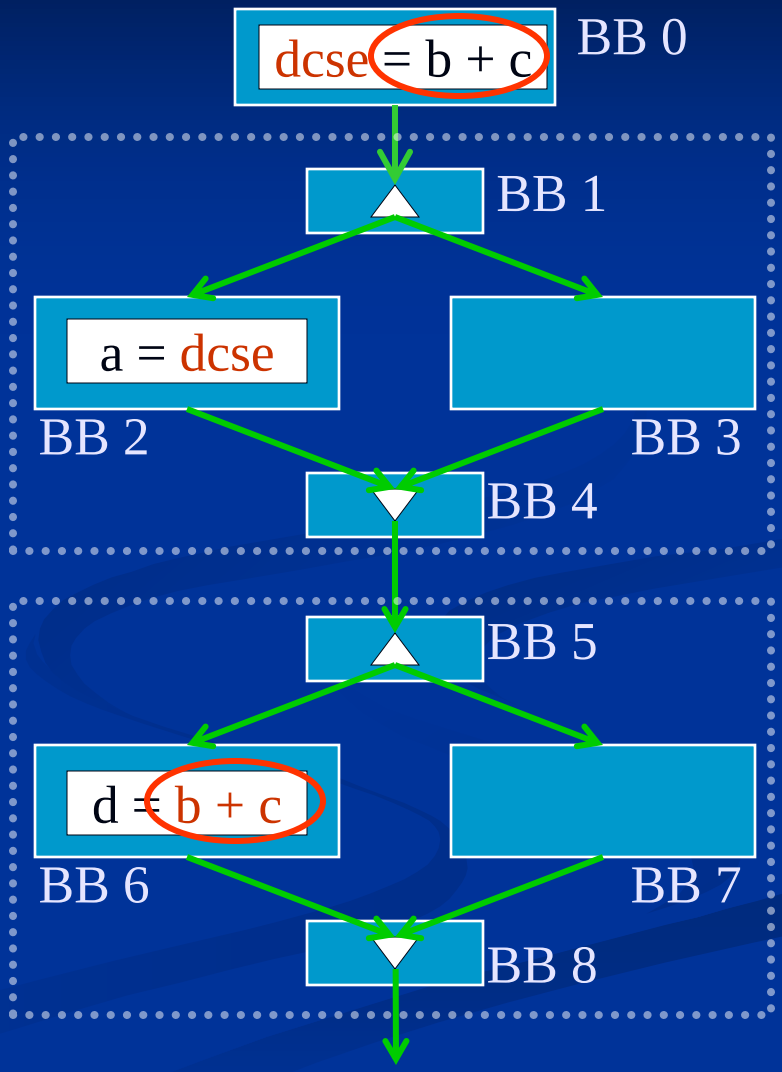
Enables Conditional Speculation



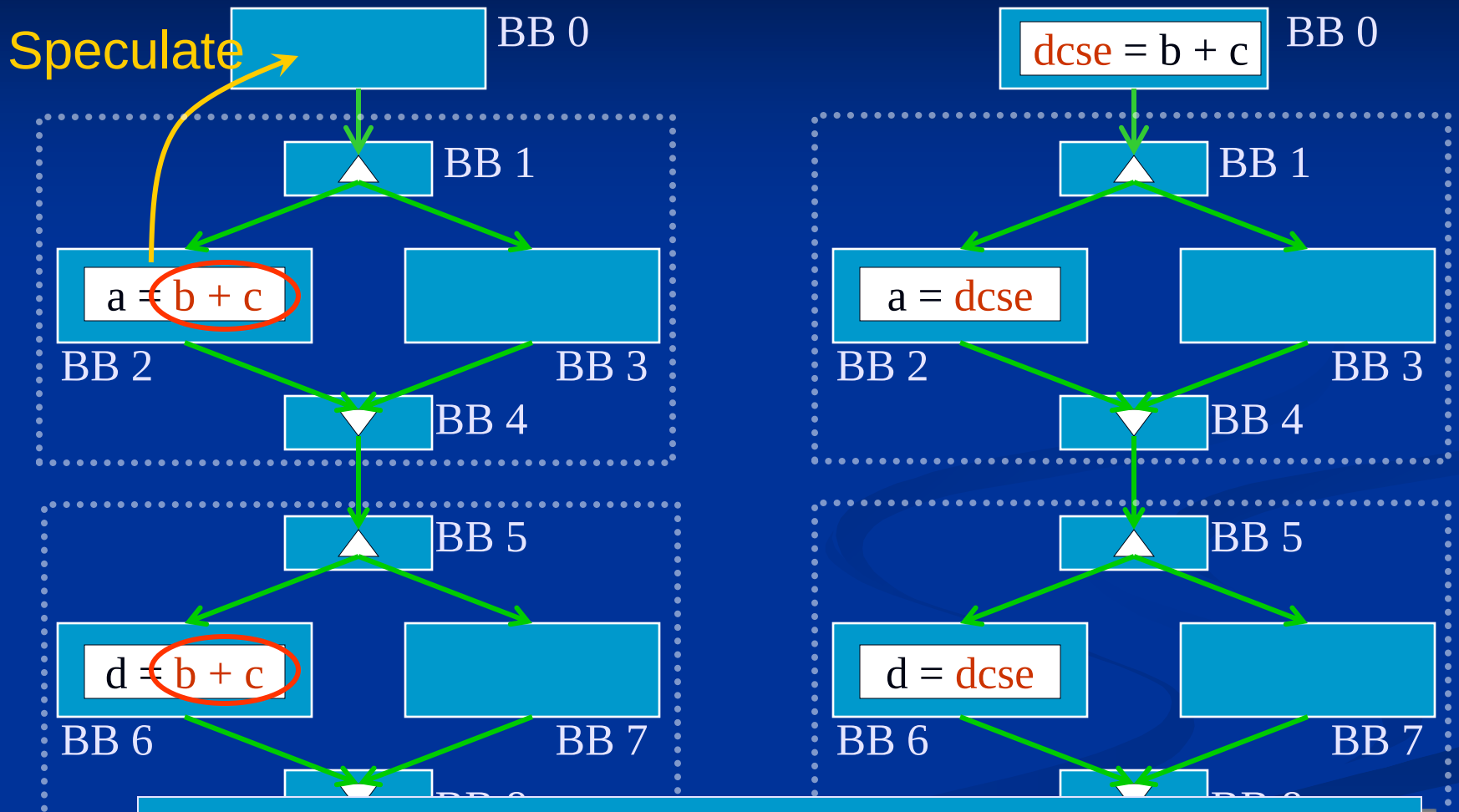
- Insert scheduling steps into **shorter** conditional branch
- Enables further code compaction

New Opportunities for "Dynamic" CSE Due to Code Motions





New Opportunities for "Dynamic" CSE Due to Code Motions

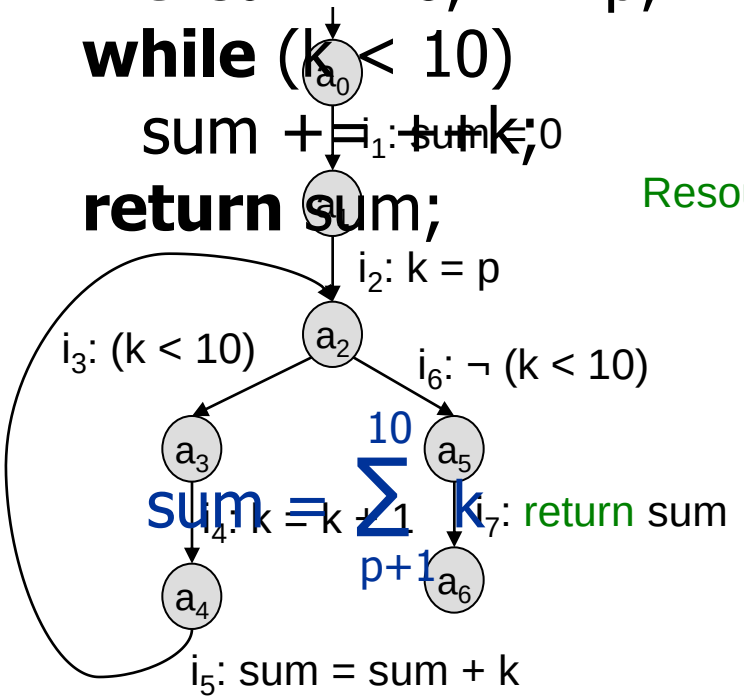


Speculative Code Motions employed during Scheduling enable new "Dynamic" opportunities for CSE

An Example of HLS

Specification:

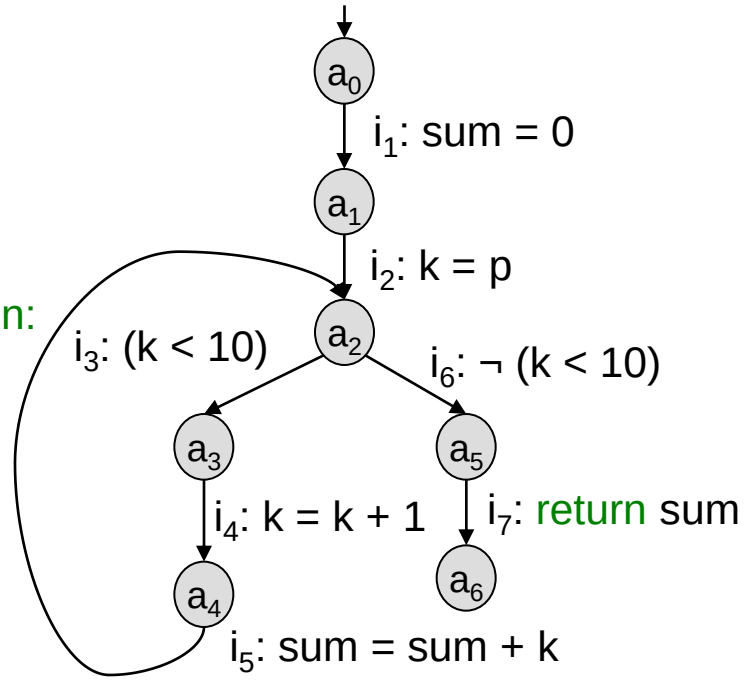
```
int SumTo10 (int p)
  int sum = 0, k = p;
  while (k < 10)
    sum = sum + k;
  return sum;
```



$$sum = \sum_{p+1}^{10} k$$

Resource Allocation:

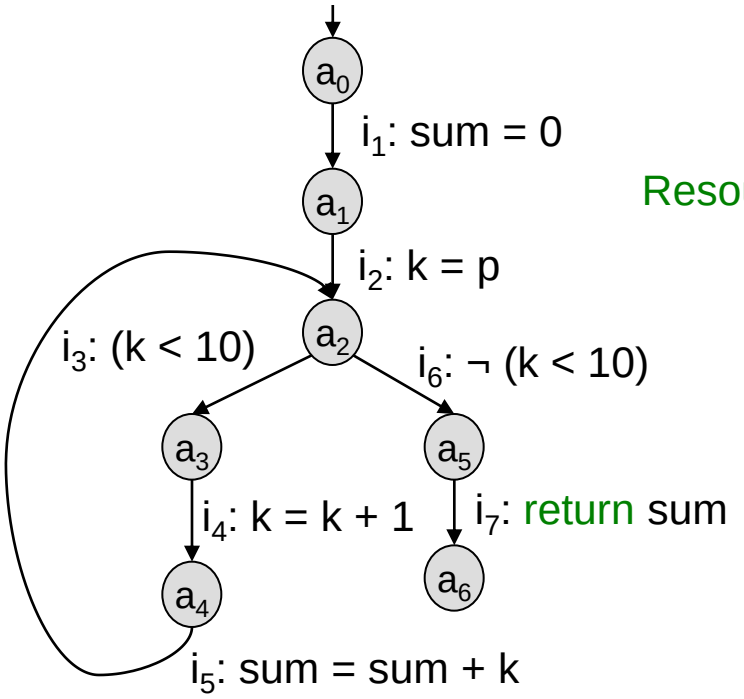
Original Program:



An Example of HLS

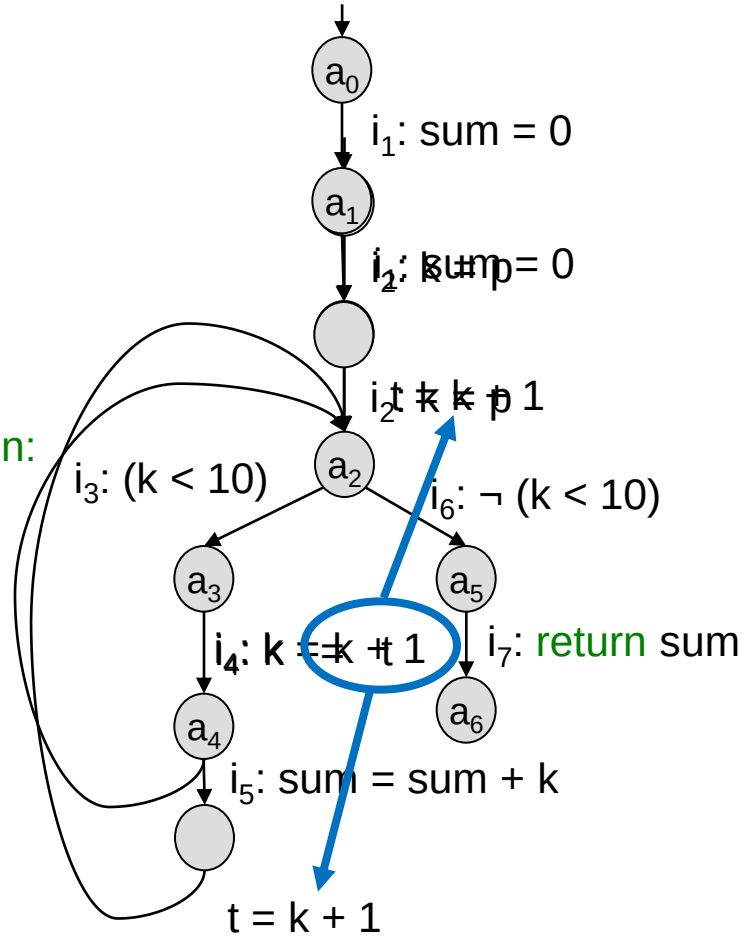
Specification:

```
int SumTo10 (int p)
  int sum = 0, k = p;
  while (k < 10)
    sum += ++k;
  return sum;
```



Resource Allocation:

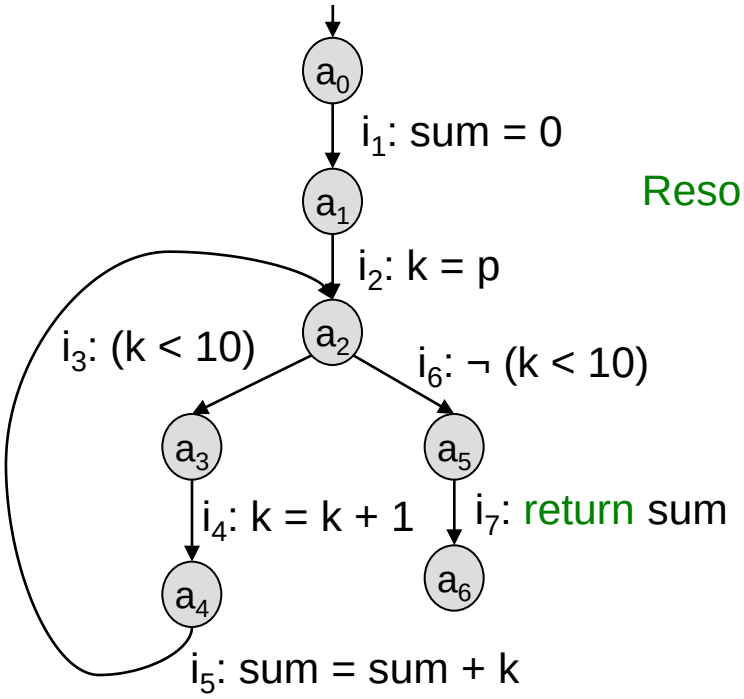
Loop Pipelining:



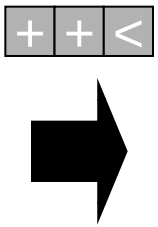
An Example of HLS

Specification:

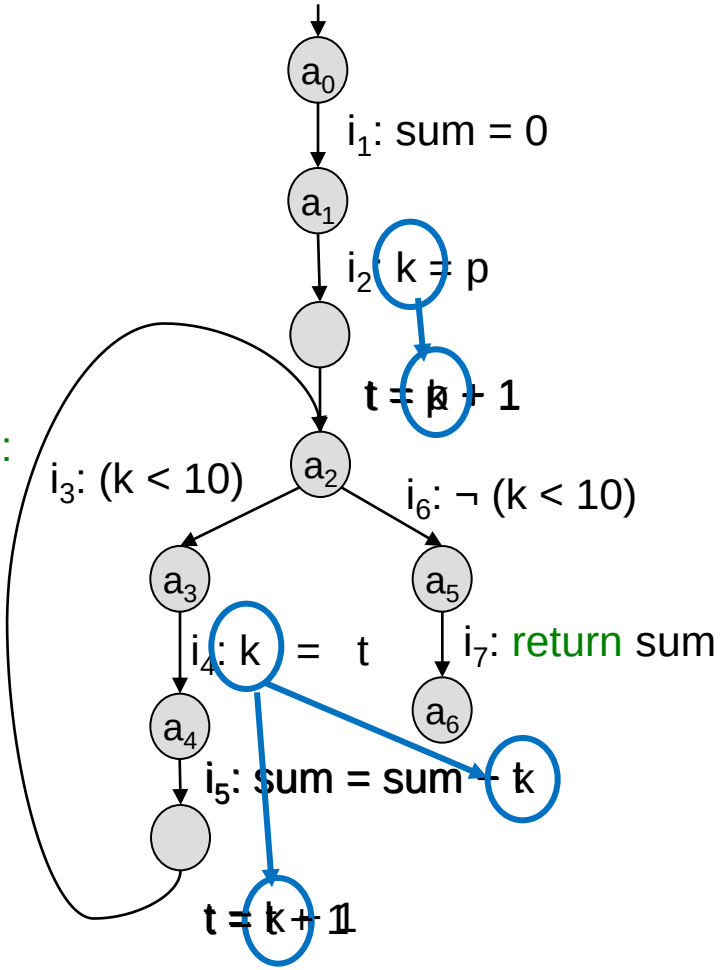
```
int SumTo10 (int p)
  int sum = 0, k = p;
  while (k < 10)
    sum += ++k;
  return sum;
```



Resource Allocation:



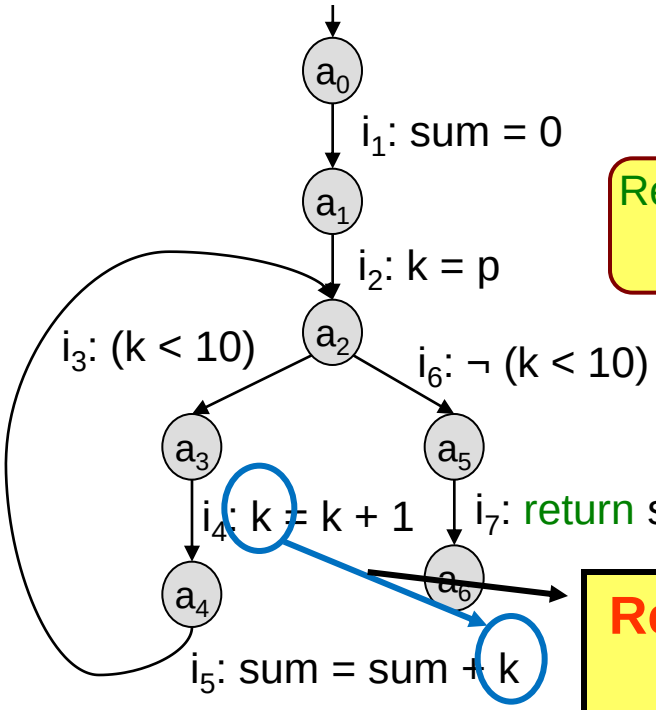
Copy Propagation:



An Example of HLS

Specification:

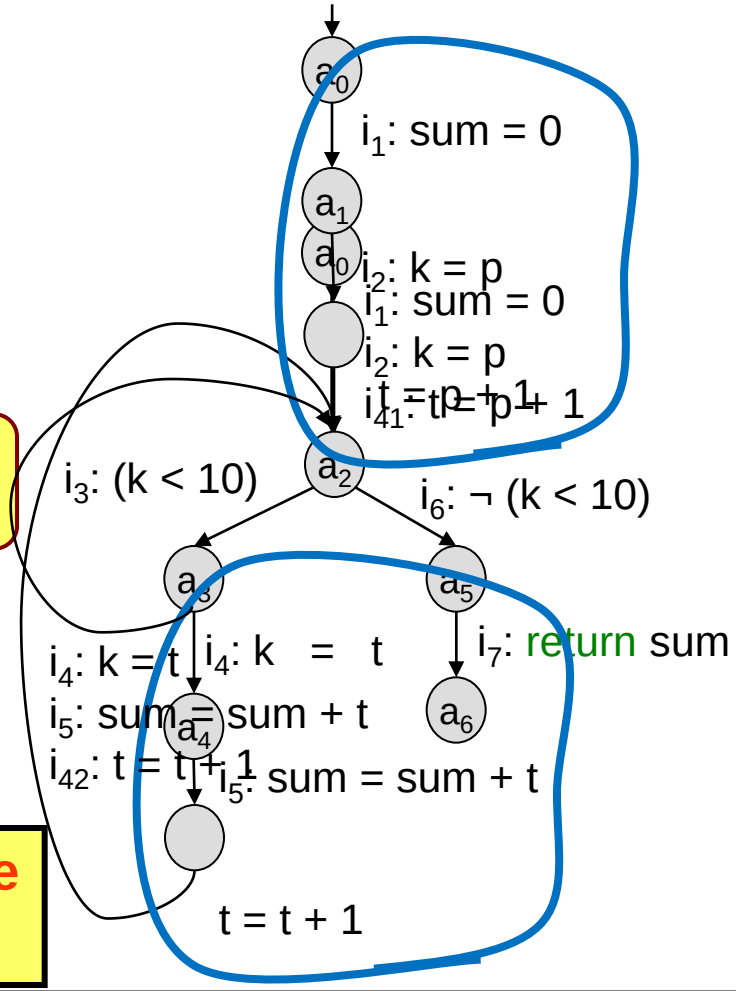
```
int SumTo10 (int p)
  int sum = 0, k = p;
  while (k < 10)
    sum += ++k;
  return sum;
```



Resource Allocation:

Read After Write dependency

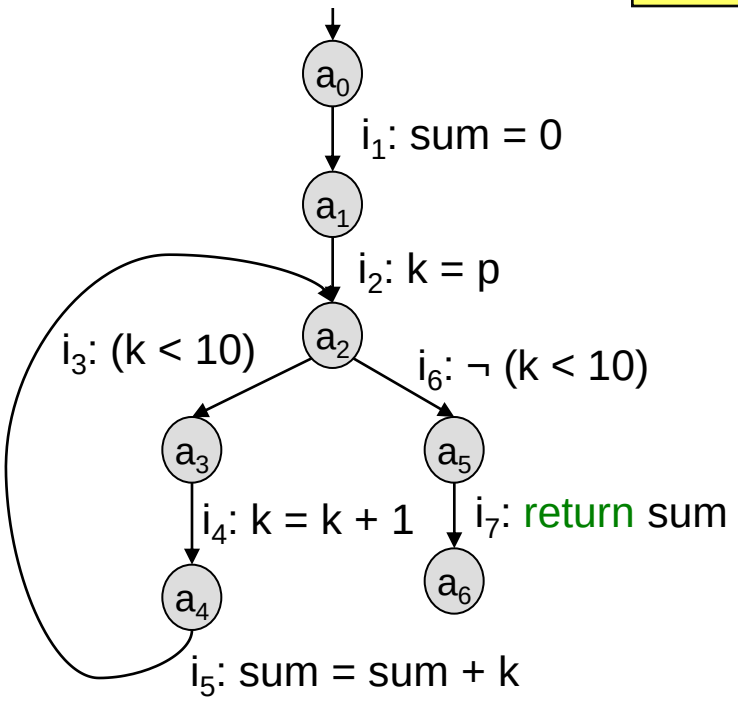
Scheduling:



An Example of HLS

Specification:

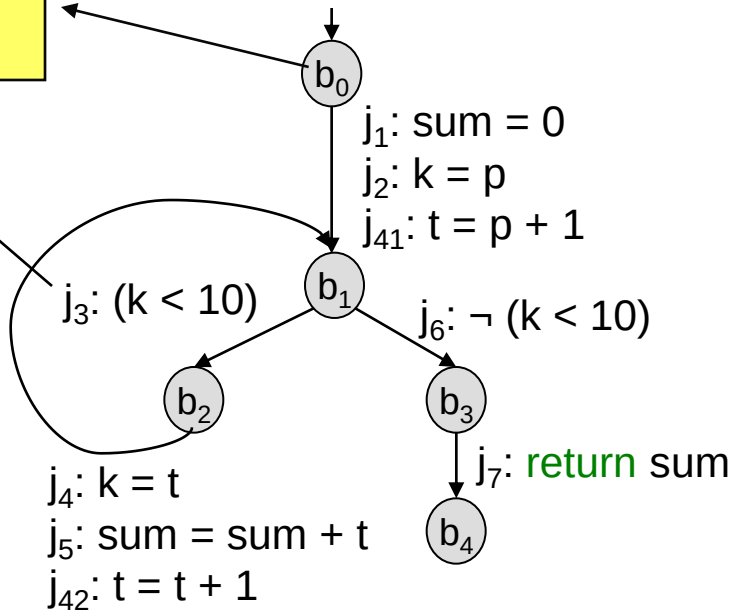
```
int SumTo10 (int p)
  int sum = 0, k = p;
  while (k < 10)
    sum += ++k;
  return sum;
```



Re-labeled

≡

Implementation:



Definition of Equivalence

- Specification ≡ Implementation
  => They have the same set of execution sequences of visible instructions.

- Visible instructions are:
  - Function call and return statements.

- Two function calls are equivalent if the state of globals and the arguments are the same.

- Two returns are equivalent if the state of the globals and the returned values are the same.

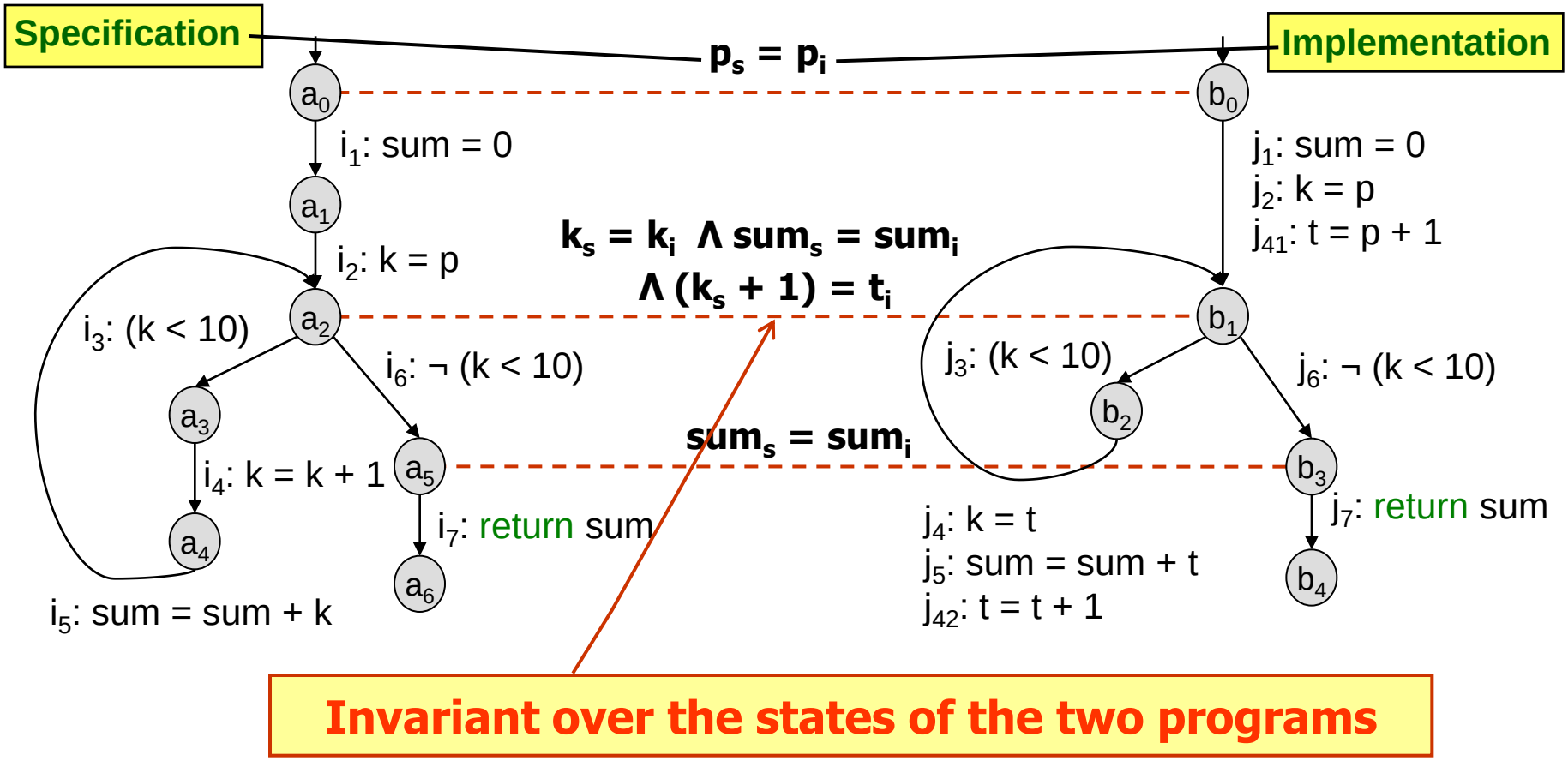
Our Approach

- Split program state space in two parts:
  - control flow state, which is finite.

    ⇒ explored by traversing the CFGs.
  - dataflow state, which may be infinite.

    ⇒ explored using Automated Theorem Prover.

Our Approach



Invariant over the states of the two programs

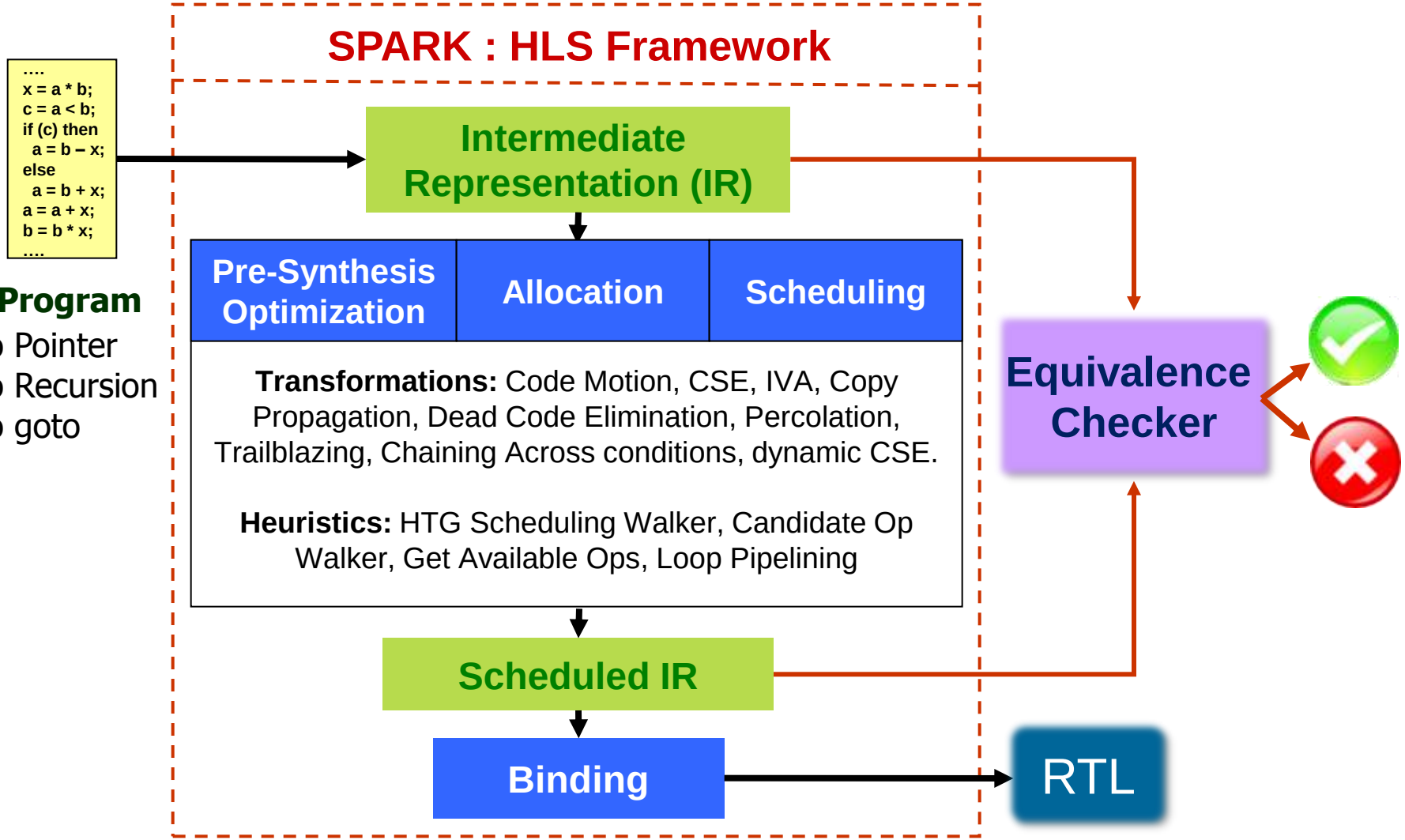
(l_1, l_2)	1st Pass	2nd Pass
1. (a_0, b_0)	$p_s = p_i$	$p_s = p_i$
2. (a_2, b_1)	$k_s = k_i$	$k_s = k_i \ \Lambda \ sum_s = sum_i \ \Lambda \ (k_s + 1) = t_i$
3. (a_5, b_3)	$sum_s = sum_i$	$sum_s = sum_i$

Translation Validation Algorithm

- Two step approach.
  - Generate Constraints: traverses the CFGs simultaneously and generates the constraints required for the visible instructions to be matched.
  - Solve Constraints: solves the constraints using a fixpoint algorithm.

- For loops: iterate to a fixed point.
  - May not terminate in general.
  - However, for all the benchmarks of SPARK that we ran our algorithm terminates.

SPARK: Parallelizing HLS Framework



Results

Benchmarks	No. of simulation relation entries	No. of calls to theorem prover	Time (secs)
1. incrementer	6	9	00.5
2. integer-sum	6	20	00.8
3. array-sum	6	24	00.8
4. diffeq	7	41	01.6
5. waka	11	79	02.6
6. pipelining	12	75	02.3
7. rotor	14	71	02.5
8. parker	26	281	05.2
9. s2r	27	570	26.7
10. findmin8	29	787	14.8

Modular: works on one procedure at a time.
Practical: took on average 6 secs to run per procedure.
Useful: found 2 previously unknown bugs in SPARK.

Bugs Found in SPARK

- Array Copy Propagation

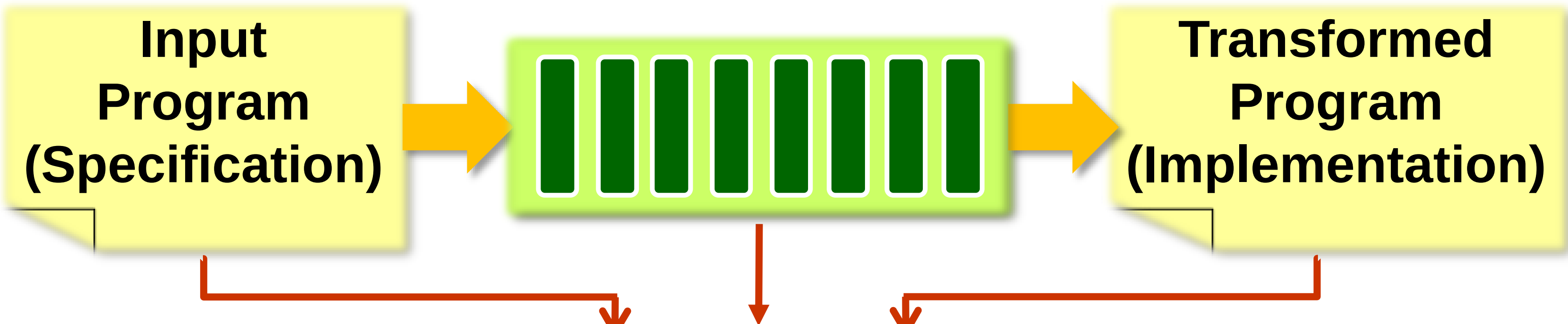
Code fragment		
Before scheduling	After scheduling (Buggy)	After scheduling (Correct)
a[0] := b[1]; c := a[0];	a[0] := b[1]; c := b[0];	a[0] := b[1]; c := b[1];

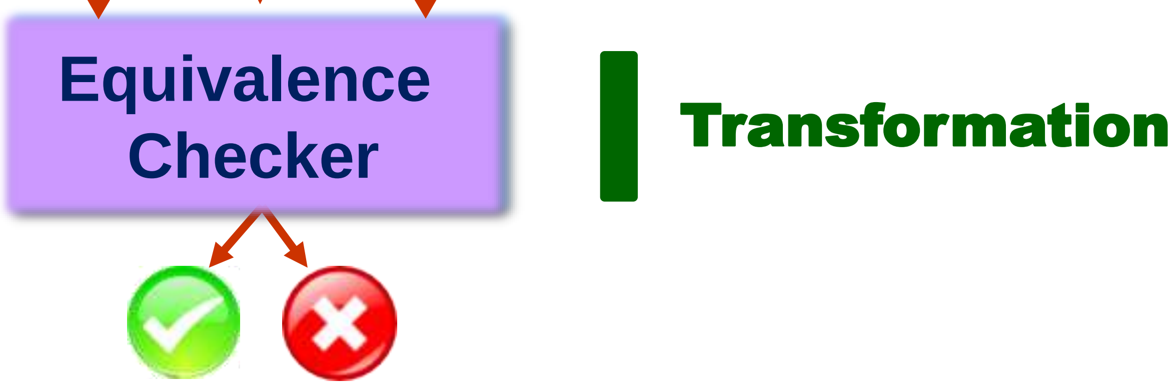
Read after Write dependency | **array index**

- Code Motion

Code fragment		
Before scheduling	After scheduling (Buggy)	After scheduling (Correct)
ret[1] := blk[0] <<3; ret[0] := ret[1];	ret[0] := ret[1]; ret[1] := blk[0] <<3;	ret[1] := blk[0] <<3; ret[0] := ret[1];

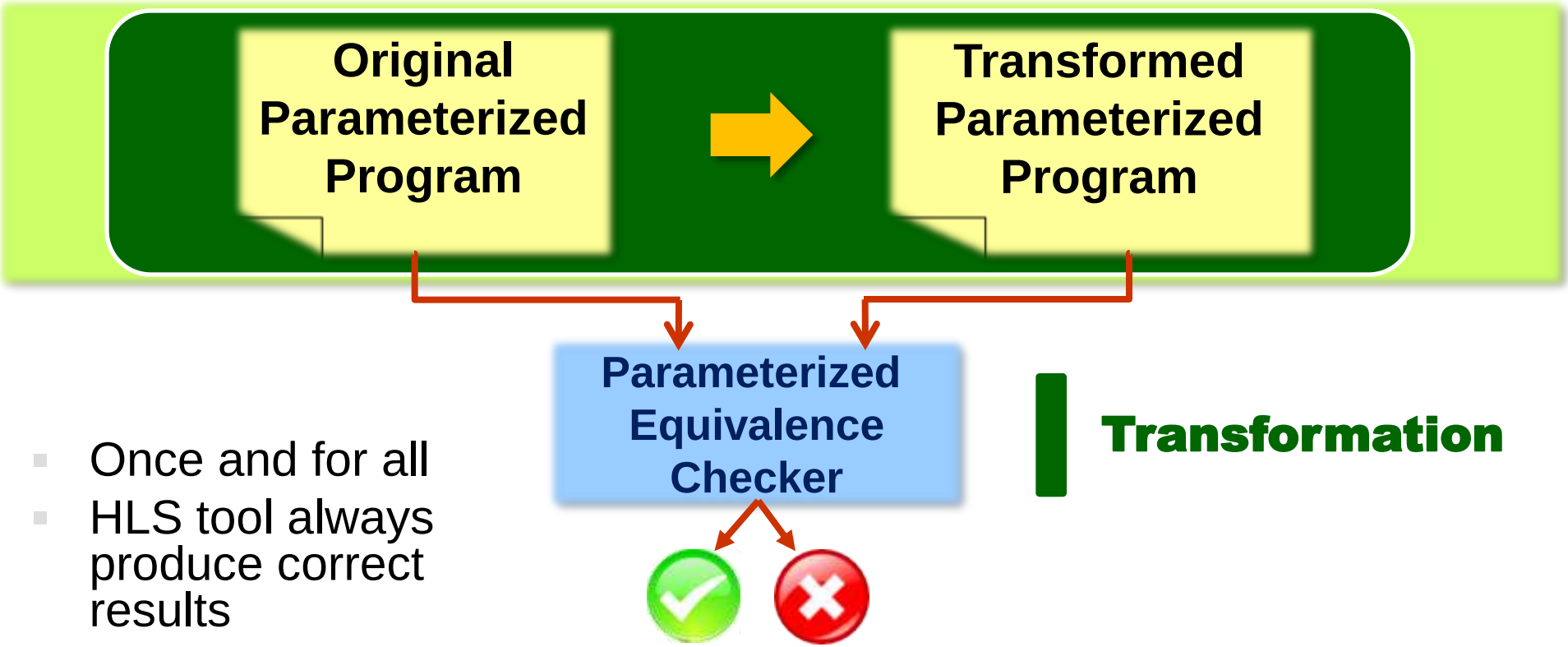
Going Forward: Parameterized Equivalence Checking





- For each translation / Once and for all
- Does not guarantee HLS tool always produce correct results / ⇒ HLS tool is bug free

Parameterized Equivalence Checking (PEC)



- Once and for all
- HLS tool always produce correct results

Experiments and Results

- Expressed and proved correct various transformations.

Transformations	Time (secs)	#ATP Calls
Copy propagation	1	3
Constant propagation	1	3
Common sub-expression elimination	1	3
Partial redundancy elimination	3	13
Loop invariant code hoisting	8	25
Conditional speculation	2	14
Speculation	3	12
Software pipelining	5	19
Loop unswitching	16	94
Loop unrolling	10	45
Loop peeling	6	40
Loop splitting	15	64

Takeaways

- Verification advances at high-level are a precondition to success in HLS
- Moderate expectations: cf. SLS
- Modularity and composition are key to reducing the size of design/verification tasks.

Related Work

- Translation Validation
  - Sequential Programs [Pnueli et al. 98] [Necula 00] [Zuck et al. 05]
  - CSP Programs [Kundu et al. 07]

- HLS Verification
  - Scheduling Step
    - Correctness preserving transformation [Eveking 99]
    - Symbolic Simulation [Ashar 99]
    - Formal assertions [Narasimhan 01]
    - Relational approaches for Equivalence of FSMDs [Kim 04, Karfa 06]