High-Level vs. RTL Combinational Equivalence: An Introduction

Alan J. Hu
University of British Columbia

Outline

- Motivation, Problem Statement
- Gate-Level Equivalence Verification
  - Symbolic Simulation
  - Cutpoints
- Symbolic Simulation of a High-Level Model
- Early Cutpoint Insertion
- Future Directions

Why verify?

- Bugs are expensive.

- Bugs are so expensive that:
  - Verification is primary front-end productivity bottleneck.
  - Verification costs swamp design costs.

Why formally verify?

- Simulation speed growing exponentially, with Moore's Law.
- Design size also growing exponentially.
- Therefore, possible behaviors growing **doubly** exponentially!

- Behavior coverage from simulation and testing have become unacceptably low.

What's formal verification?

- Formal verification means **proving** a **property** about a **model** of a design.
  - "**proving**" – as good as mathematical proof.
  - "**property**" – got to specify what is correct
  - "**model**" – the tool runs at some level (layout, schematic, RTL, etc.

- In the past 15-20 years, revolutionary new ideas make formal verification practical in many cases

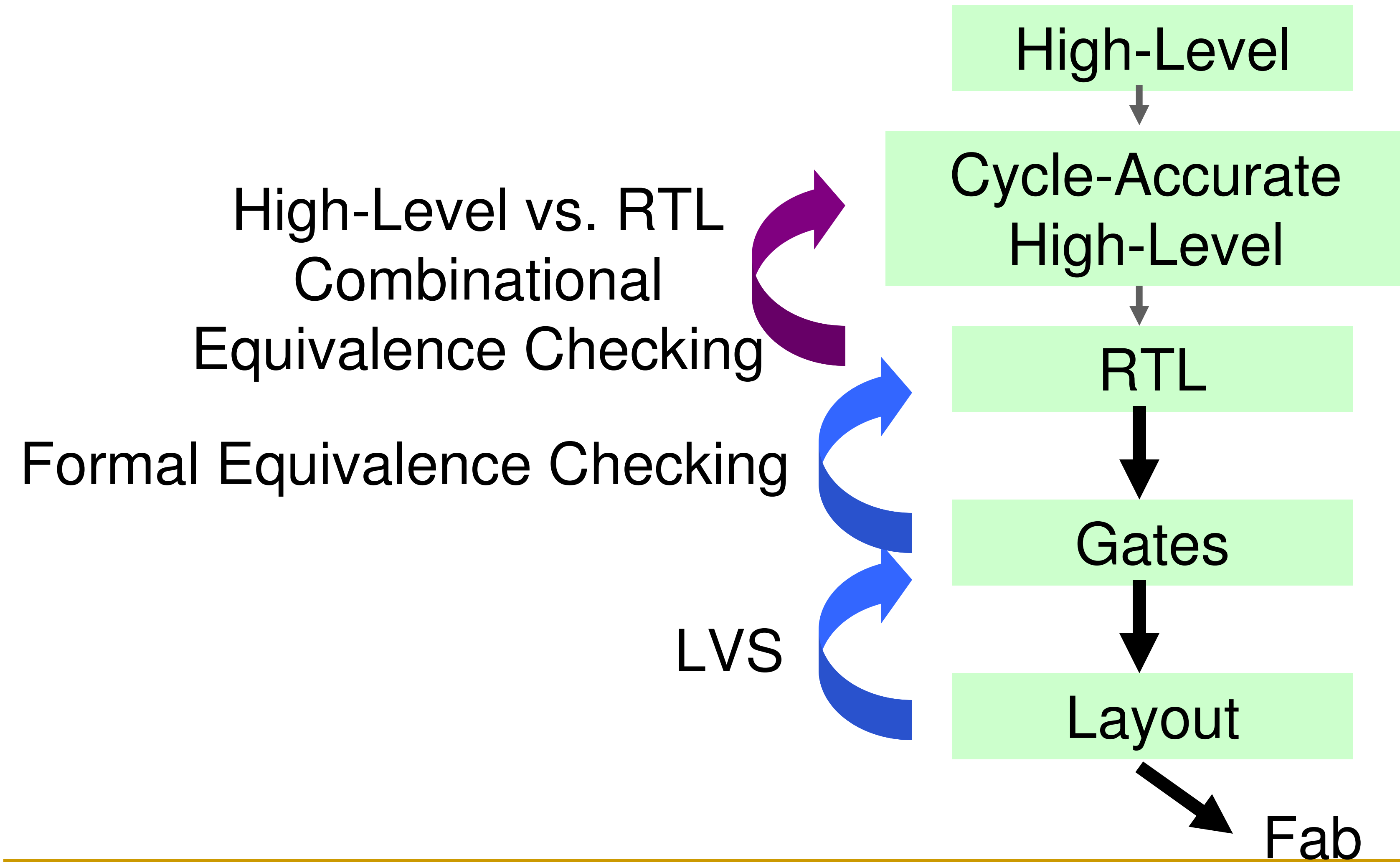
Two Kinds of Verification

- Property Checking (aka Design Verification)
  - Formally specify desired properties (e.g., mutual exclusion, no deadlock)
  - Check that model satisfies property (model checking)
- Equivalence Checking (aka Implementation Verification)
  - Check whether two models are equivalent
  - Biggest success of formal verification to date

Why equivalence checking?

- Theoretically, two kinds of verification are the same.
- In practice, they are different:
  - No separate specification needed
  - Assumption of similarity between two designs

Why high-level vs. RTL?



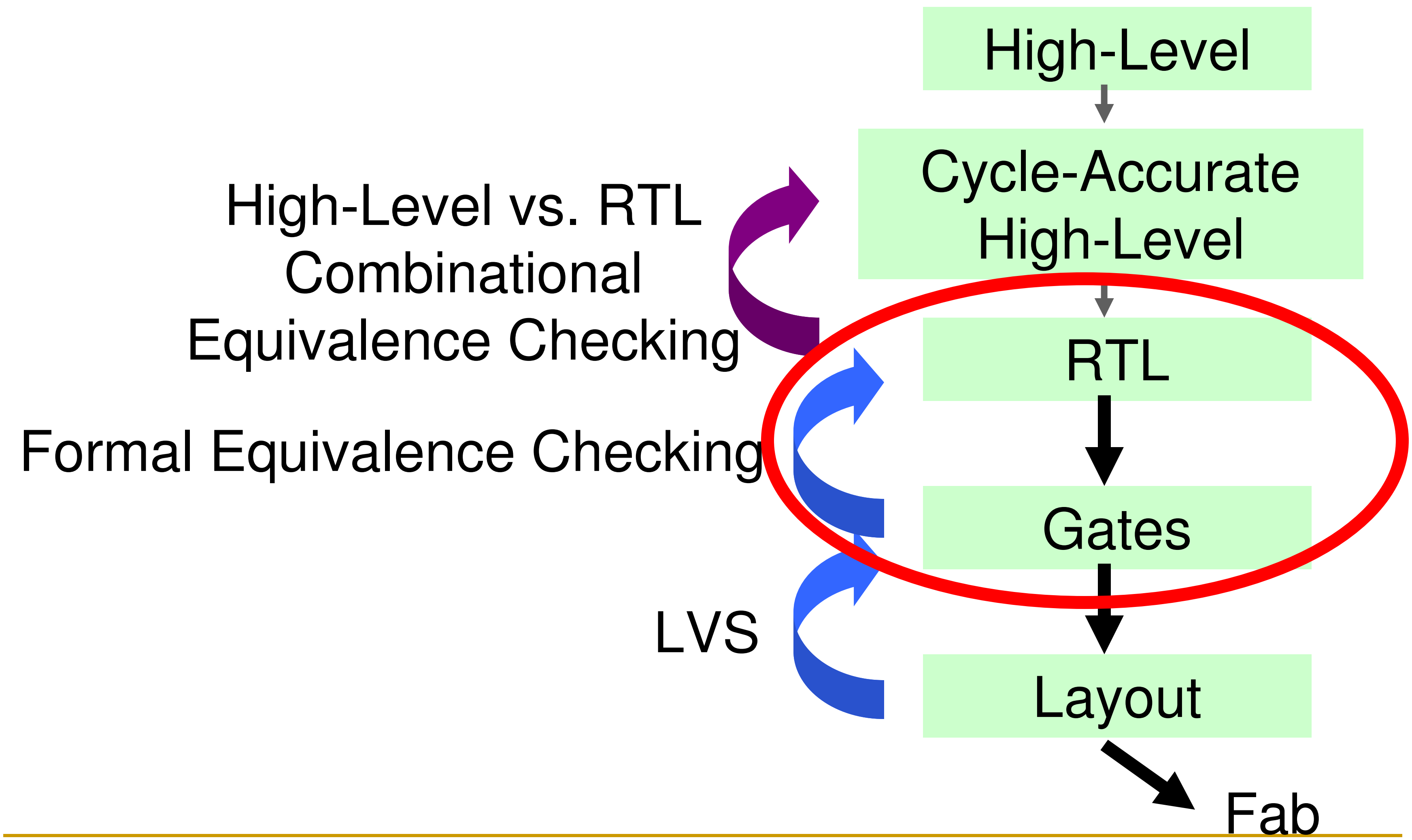
Problem Statement

Given

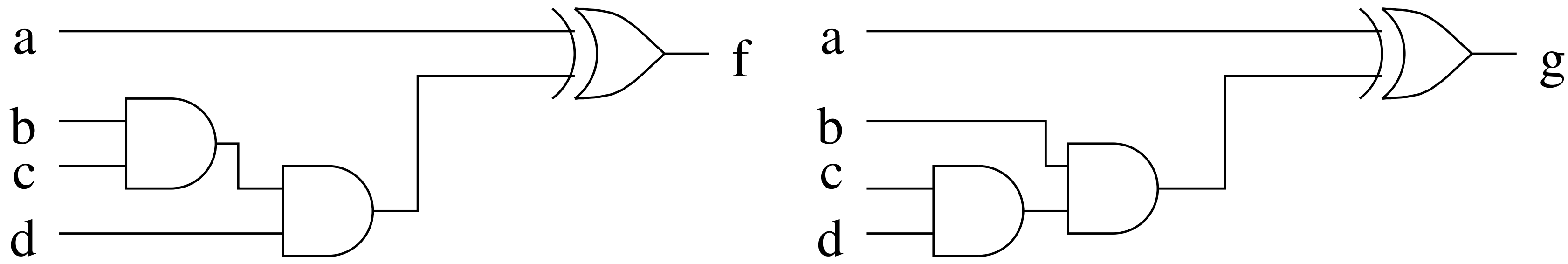
- A high-level software model
  - "Combinational" – output as function of inputs
  - "Non-synthesizable" – too complex for current tools
- A combinational hardware model

Do they have the same functionality?

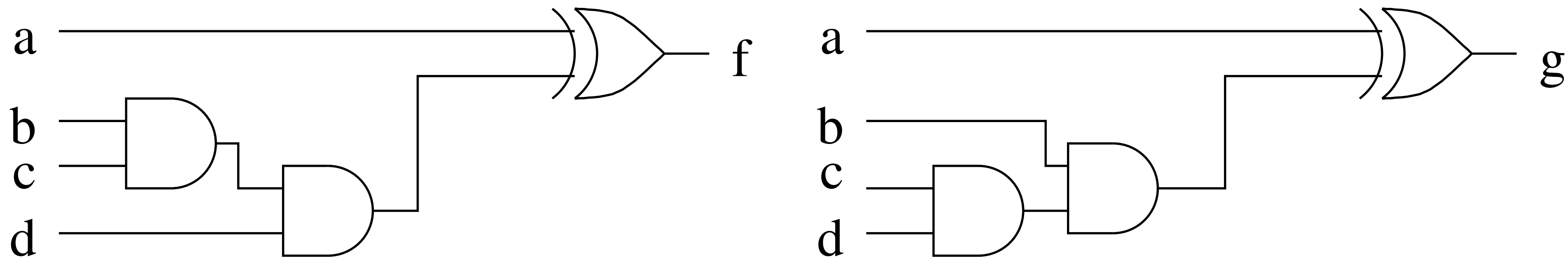
Equivalence Checking



Combinational Equivalence



Combinational Equivalence



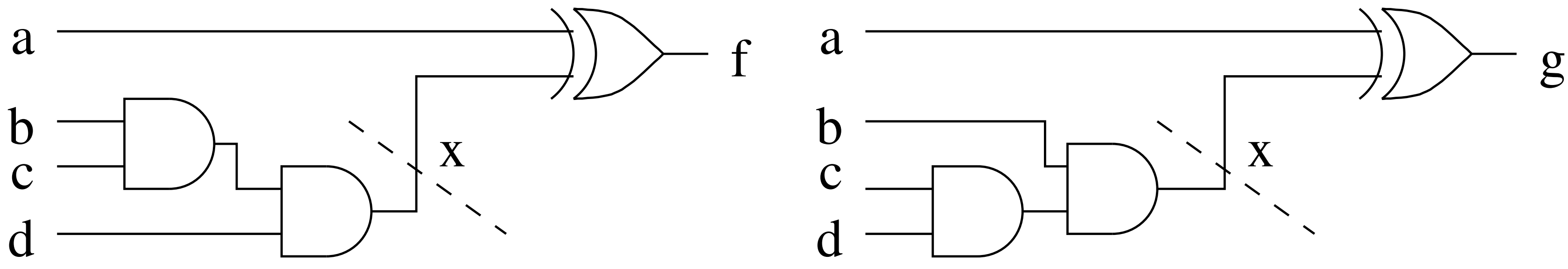
Symbolically simulate both circuits

$f = a \oplus ((b \wedge c) \wedge d) \quad g = a \oplus (b \wedge (c \wedge d))$

Compare results (BDDs, SAT, etc.)

Complexity blow up for industrial circuits.

Cutpoints



Guess cutpoint and prove equivalence:

E.g., the wire x in each circuit

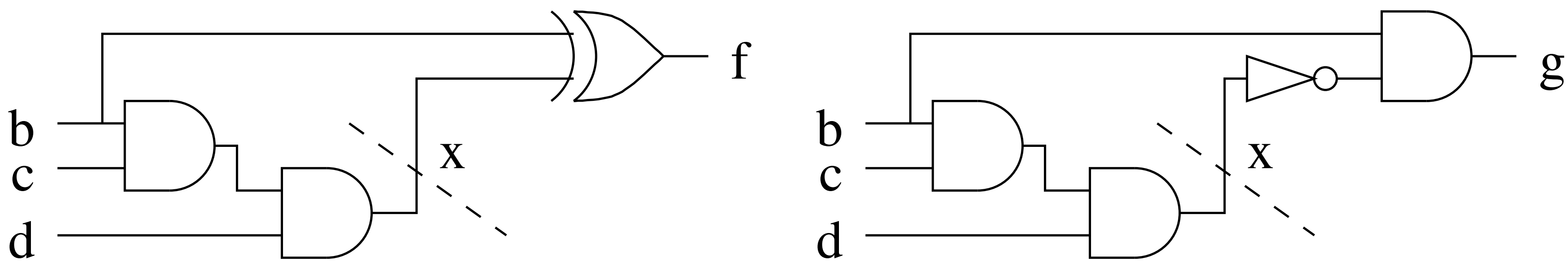
Prove $((b \wedge c) \wedge d) = (b \wedge (c \wedge d))$

Treat cutpoint as new primary input:

Prove $f = a \oplus x$ equivalent to $g = a \oplus x$

Divide and conquer.

False Inequivalence



Guess cutpoint and prove equivalence:

E.g., the wire x in each circuit

Treat cutpoint as new primary input:

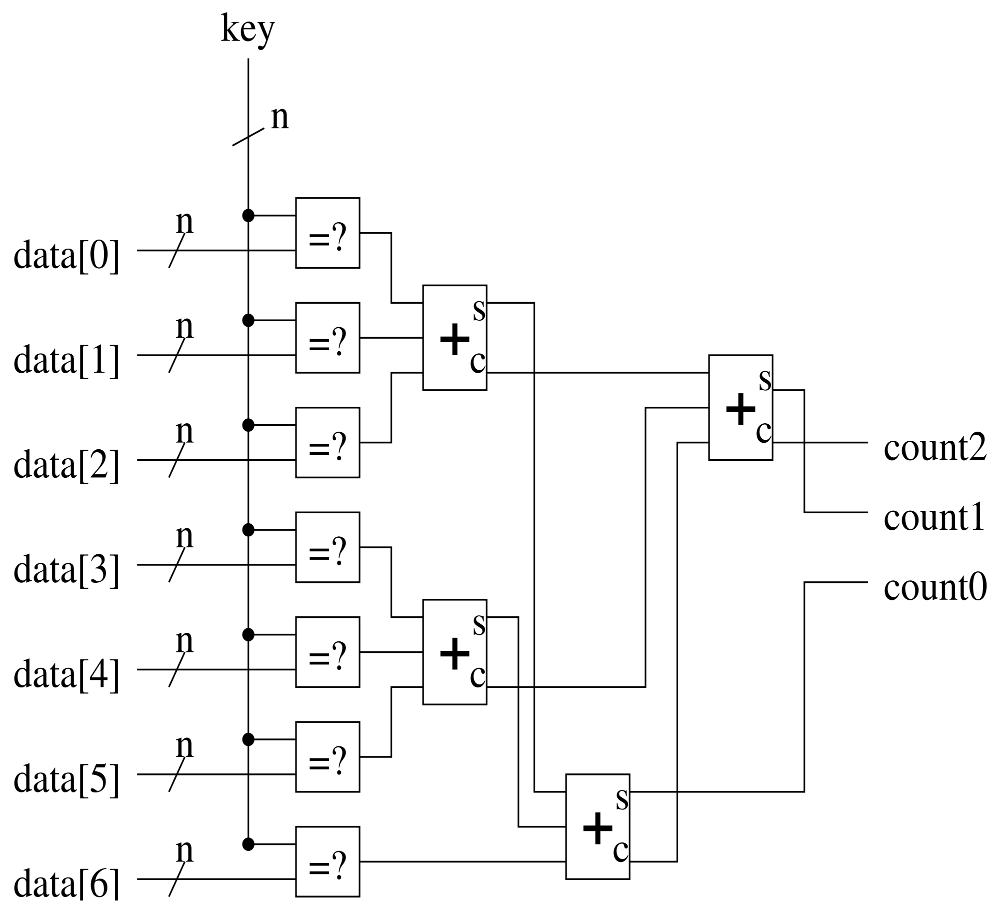
f= b ⊕ x versus g= b ∧ (!x)

Combinational Equivalence: Key Ideas

- Symbolically simulate to compute functionality.
- Use an efficient representation for the symbolic simulation, e.g., BDDs or circuit-like structure for SAT.
- Find equivalent points to use as cutpoints to simplify the problem.

Software vs. Hardware

```
int f(int key, int data[7])
{
    int i, count = 0;
    for (i=0; i<7; i++) {
        if (key==data[i])
            count++;
    }
    return count;
}
```



Combinational Equivalence: Key Ideas

- **Symbolically simulate to compute functionality.**
- Use an efficient representation for the symbolic simulation, e.g., BDDs or circuit-like structure for SAT.
- Find equivalent points to use as cutpoints to simplify the problem.

Software vs. Hardware

```
int f(int key, int data[7])
{
   int i, count = 0;
   for (i=0; i<7; i++) {
         if (key==data[i])
               count++;
   }
   return count;
}
```

- key = orig_key
- data = orig_data

Software vs. Hardware

```
int f(int key, int data[7])
{
   int i, count = 0;
   for (i=0; i<7; i++) {
         if (key==data[i])
               count++;
   }
   return count;
}
```

- key = orig_key
- data = orig_data
- i = ?

Software vs. Hardware

```
int f(int key, int data[7])
{
   int i, count = 0;
   for (i=0; i<7; i++) {
        if (key==data[i])
             count++;
   }
   return count;
}
```

- key = orig_key
- data = orig_data
- i = ?
- count = 0

Software vs. Hardware

```
int f(int key, int data[7])
{
   int i, count = 0;
   for (i=0; i<7; i++) {
         if (key==data[i])
               count++;
   }
   return count;
}
```

- key = orig_key
- data = orig_data
- i = 0
- count = 0

Software vs. Hardware

```
int f(int key, int data[7])
{
   int i, count = 0;
   for (i=0; i<7; i++) {
         if (key==data[i])
                  count++;
   }
   return count;
}
```

- key = orig_key
- data = orig_data
- i = 0
- count = 0

Assume:

orig_key==orig_data[0]

Software vs. Hardware

```
int f(int key, int data[7])
{
   int i, count = 0;
   for (i=0; i<7; i++) {
         if (key==data[i])
               count++;
   }
   return count;
}
```

- key = orig_key
- data = orig_data
- i = 0
- count = 1

Assume:

orig_key==orig_data[0]

Software vs. Hardware

```
int f(int key, int data[7])
{
   int i, count = 0;
   for (i=0; i<7; i++) {
       if (key==data[i])
           count++;
   }
   return count;
}
```

- key = orig_key
- data = orig_data
- i = 1
- count = 2

Assume:

orig_key==orig_data[0]

orig_key==orig_data[1]

Software vs. Hardware

```
int f(int key, int data[7])
{
   int i, count = 0;
   for (i=0; i<7; i++) {
        if (key==data[i])
             count++;
   }
   return count;
}
```

- Different results on every path
- Must track assumptions on each path
- Exponential number of paths!

- Merge paths with conditional expressions?

Path Merging

```
int f(int key, int data[7])
{
   int i, count = 0;
   for (i=0; i<7; i++) {
        if (key==data[i])
              count++;
   }
   return count;
}
```

- i = 0
- count = (orig_key==orig_data[0])?1:0

Path Merging

```
int f(int key, int data[7])
{
   int i, count = 0;
   for (i=0; i<7; i++) {
         if (key==data[i])
                count++;
   }
   return count;
}
```

- i = 1
- count = (orig_key==orig_data[1]) ? ((orig_key==orig_data[0]) ?2:1) : ((orig_key==orig_data[0]) ?1:0)

Path Merging

```
int f(int key, int data[7])
{
    int i, count = 0;
    for (i=0; i<7; i++) {
        if (key==data[i])
            count++;
    }
    return count;
}
```

- i = 2
- count = (orig_key==orig_data[2]) ? ((orig_key==orig_data[1]) ? ((orig_key==orig_data[0])?3:2) : ((orig_key==orig_data[0])?2:1)) : ((orig_key==orig_data[1]) ? ((orig_key==orig_data[0])?2:1) : ((orig_key==orig_data[0])?1:0))

Path Merging

```
int f(int key, int data[7])
{
    int i, count = 0;
    for (i=0; i<7; i++) {
        if (key==data[i])
            count++;
    }
    return count;
}
```

- i = 3
- count = (orig_key==orig_data[3]) ? ((orig_key==orig_data[2]) ? ((orig_key==orig_data[1]) ? ((orig_key==orig_data[0])?3:2) : ((orig_key==orig_data[0])?2:1)) : ((orig_key==orig_data[1]) ? ((orig_key==orig_data[0])?2:1) : ((orig_key==orig_data[0])?1:0))) : ((orig_key==orig_data[2]) ? ((orig_key==orig_data[1]) ? ((orig_key==orig_data[0])?3:2) : ((orig_key==orig_data[0])?2:1)) : ((orig_key==orig_data[1]) ? ((orig_key==orig_data[0])?2:1) : ((orig_key==orig_data[0])?1:0)))

Path Merging

```
int f(int key, int data[7])
{
   int i, count = 0;
   for (i=0; i<7; i++) {
        if (key==data[i])
             count++;
   }
   return count;
}
```

- Exponential growth in expression size!

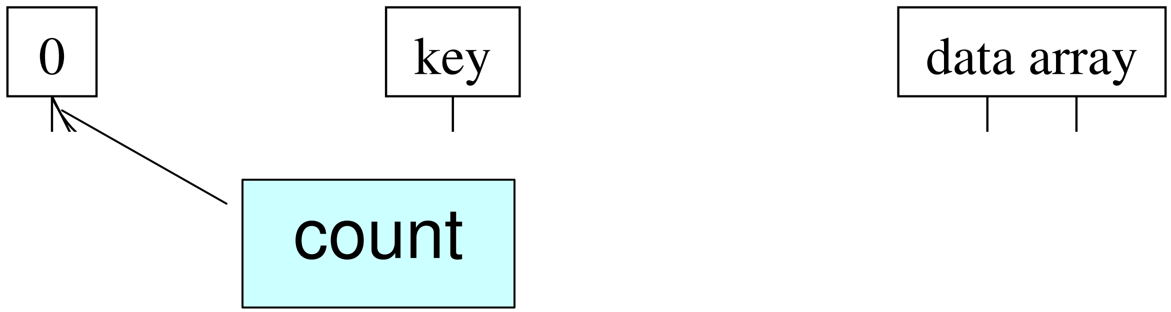
Combinational Equivalence: Key Ideas

- Symbolically simulate to compute functionality.
- **Use an efficient representation for the symbolic simulation, e.g., BDDs or circuit-like structure for SAT.**
- Find equivalent points to use as cutpoints to simplify the problem.

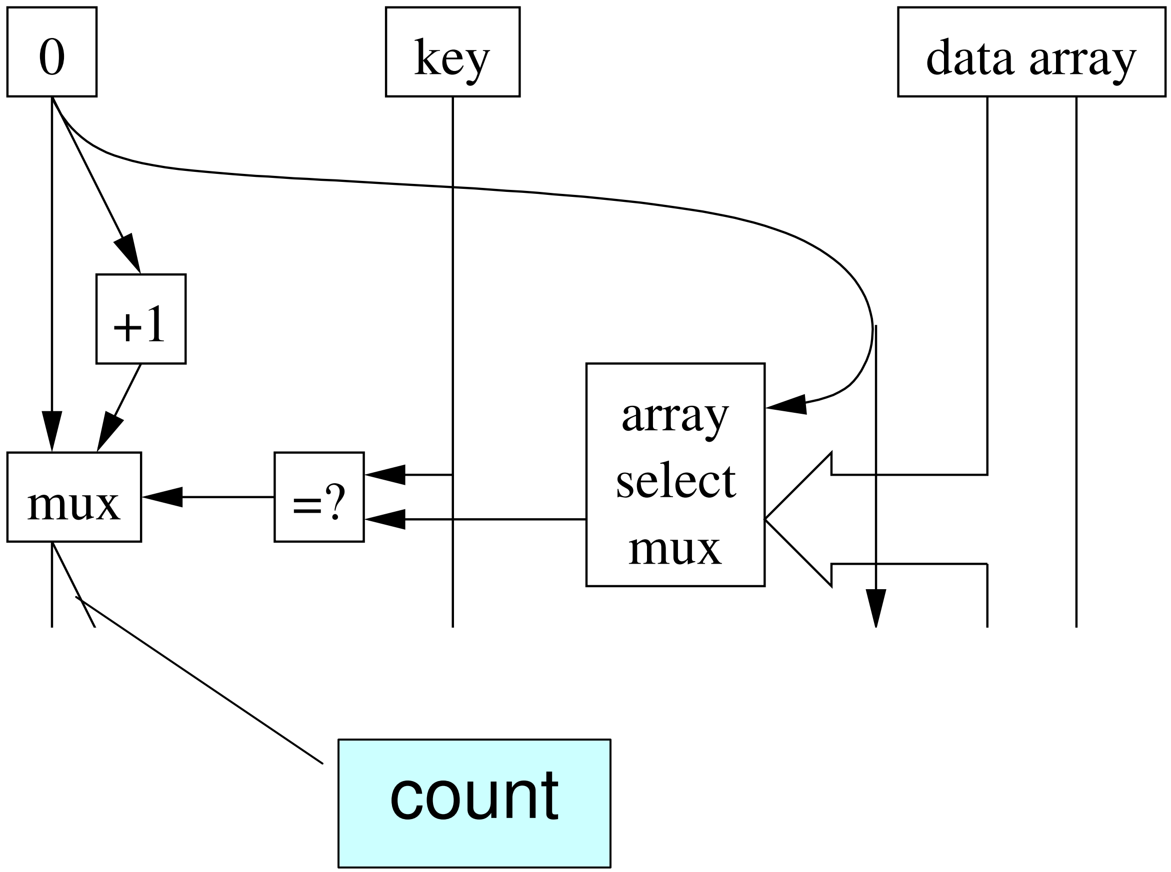
Shared Graph Representation

- Use a maximally shared combinational circuit graph as representation of functionality.

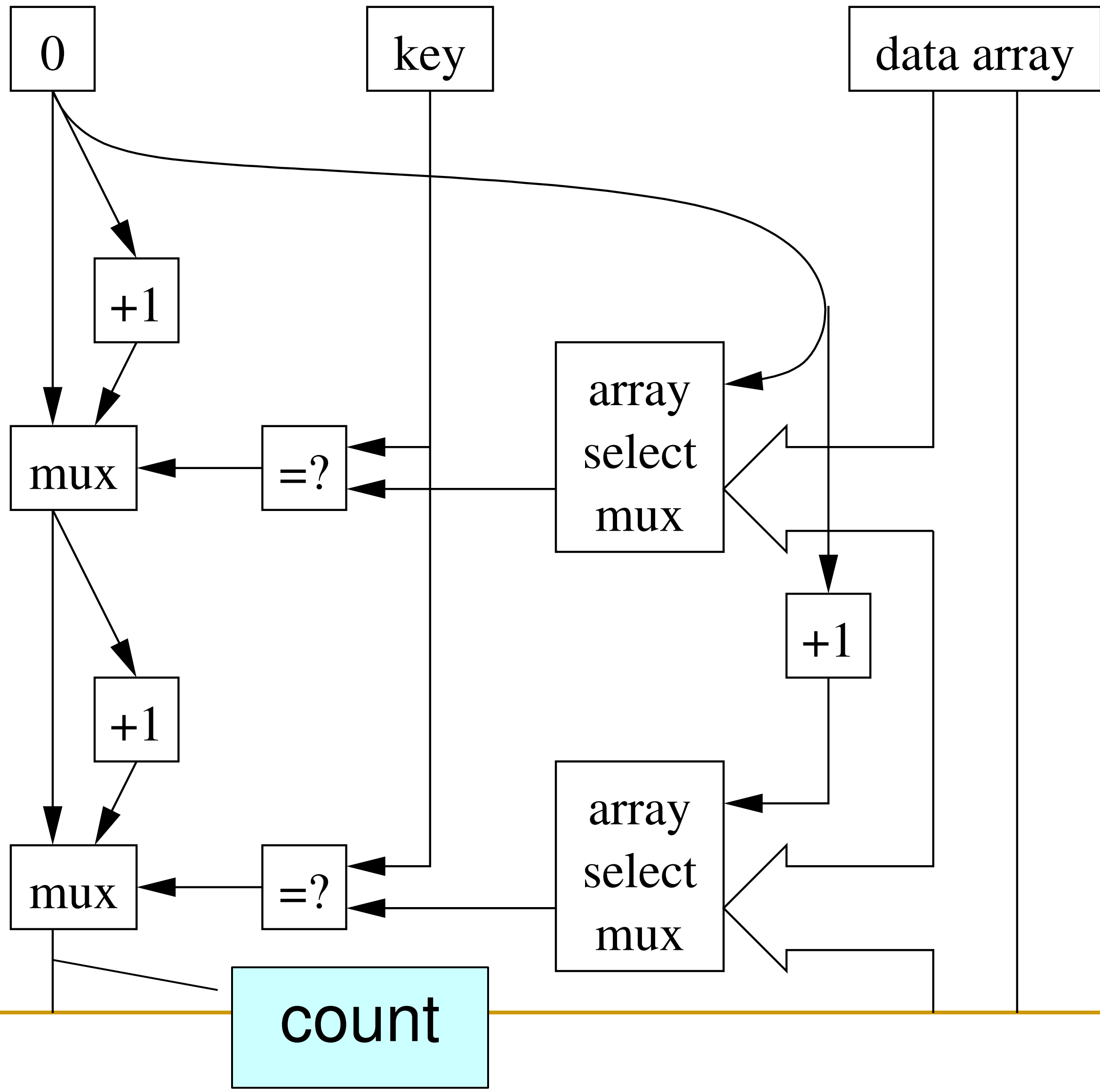
Shared Graph Representation



Shared Graph Representation



Shared Graph Representation



Shared Graph Representation

- Use a maximally shared combinational circuit graph as representation of functionality.
- Graph structure grows linearly (in size of unrolled program).
- Result is essentially a synthesized combinational circuit.

- Still has potential problems for very complex software

Combinational Equivalence: Key Ideas

- Symbolically simulate to compute functionality.
- Use an efficient representation for the symbolic simulation, e.g., BDDs or circuit-like structure for SAT.
- **Find equivalent points to use as cutpoints to simplify the problem.**

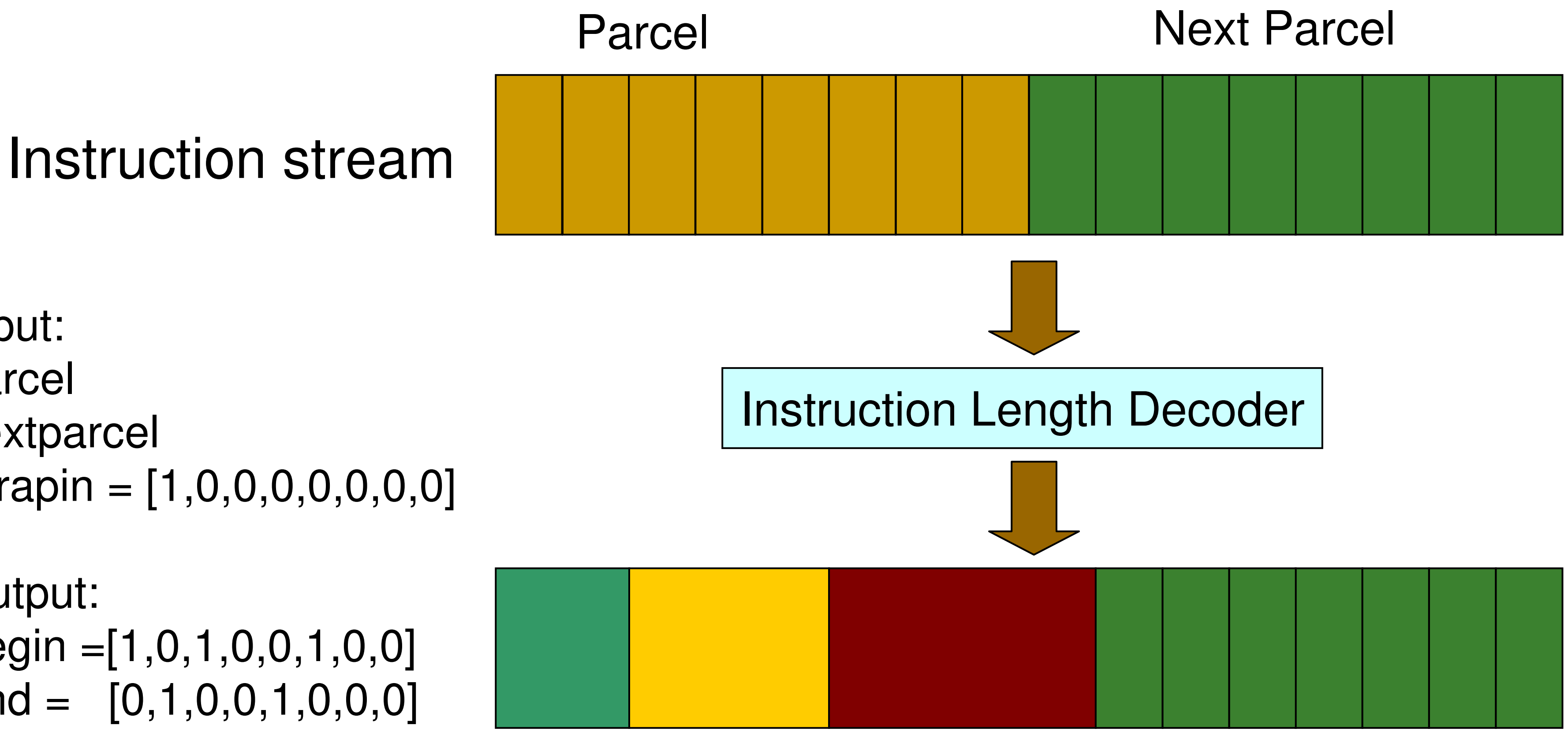
Early Cutpoint Insertion

- Find and insert cutpoint **during** symbolic simulation of software, not after synthesizing an equivalent circuit.
- Reduces blow-up, allows using BDDs to represent path conditions.
- Therefore, can handle much more complex branching and looping conditions.

Case Study: IA-32 Instruction Length Decoder

- Challenge problem suggested by Robert Jones of Intel Corporation.
- IA-32 has very complex instruction encoding:
  - Variable length instructions from 1 to 15+ bytes
  - Prefixes, over-rides of field lengths, etc.
- Instruction length decoder marks instruction boundaries in an instruction buffer, in a single cycle.

IA32 Instruction Length Decoder



Input:
parcel
nextparcel
Wrapin = [1,0,0,0,0,0,0,0]

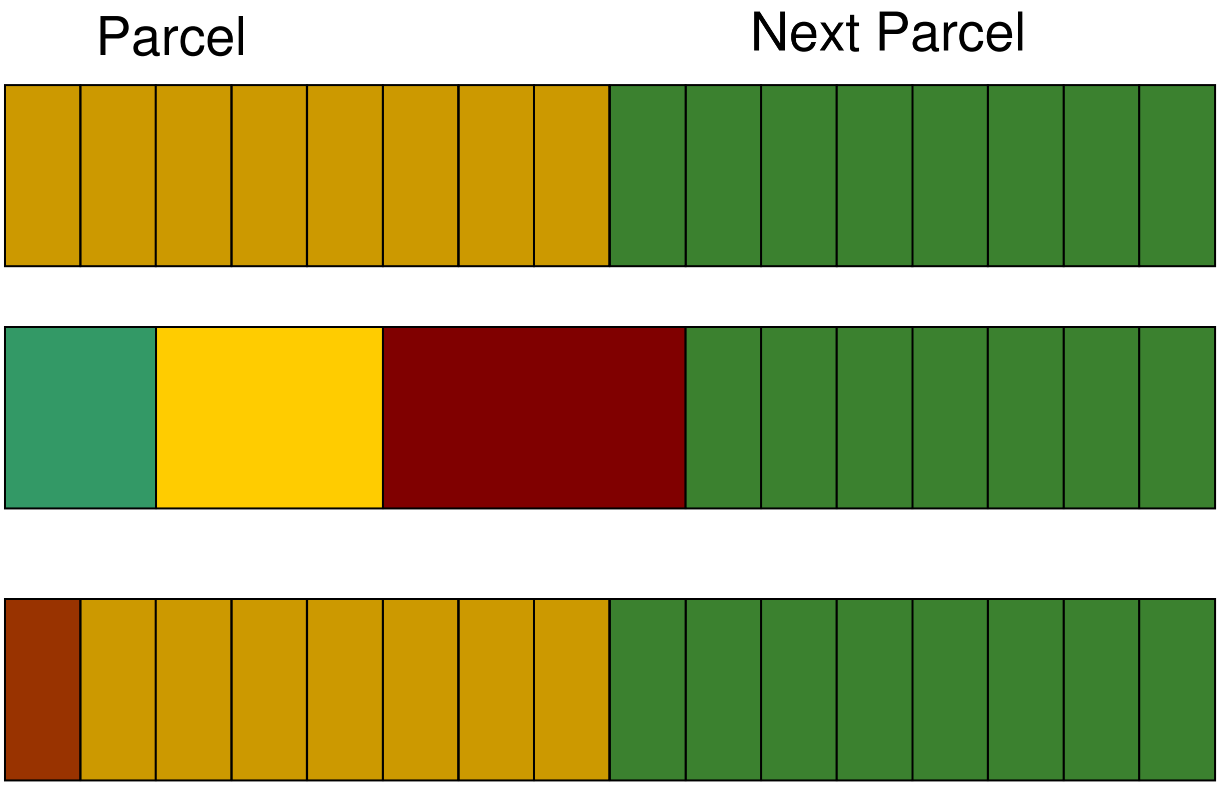
Output:
Begin =[1,0,1,0,0,1,0,0]
End = [0,1,0,0,1,0,0,0]
Wrapout = [0,1,0,0,0,0,0,0]

IA32 Instruction Length Decoder

```
While(1){

    length_decoder();
    parcel = nextparcel;
    wrapin = wrapout;
    read nextparcel;

}
```



Software Model of ILD

```
while (wrap < PARCEL_SIZE) {
  begin[wrap]=1; /* Start of instruction */

  /* Set default sizes. */
  operand_mode = INIT_OPERAND_MODE;
  address_mode = INIT_ADDRESS_MODE;

  get_next_byte();
  ret = handle_prefixes();
  /* If there were any prefixes, get the next byte for opcode. */
  if (ret) get_next_byte();

  if (current_byte != ESCAPE)  handle_one_byte_opcodes();
  else {/* Escape to two-byte opcode */
    get_next_byte(); /* Skip over the escape code. */
    handle_two_byte_opcodes();
  }
}
```

Hardware Model of ILD

- All decoding is in parallel
- A priority-encoding network to decide which blocks of ILD logic are the valid ones:

  valid(P_m) *iff* valid(P_n) $\wedge$ $m = n +$ length_from(n)

- Optimized by using script.rugged (SIS)

Verification Challenges

- Software
  - Easy to describe the functionality
  - Serial
  - Exponential number of paths
  - Very complex control flow
- Hardware
  - Complicated, RTL circuit
  - Highly parallel (one cycle)

Effect of Early Cutpoints

	Linear Building BDD		Early Cutpoints	
Example	Time(s)	Mem(MB)	Time(s)	Mem(MB)
EX20-8	0.28	61	0.11	58
EX20-16	89.01	1746	0.24	60
EX20-32		mem out	0.53	64
EX20-64		mem out	1.35	72
EX97-8	1.46	92	0.51	64
EX97-16	1187.72	1800	1.10	73
EX97-32		mem out	2.35	95
EX97-64		mem out	5.41	136
EX251-12	309.18	1843	0.64	66
EX251-16		mem out	1.09	71
EX251-32		mem out	7.45	170
EX251-64		mem out	16.81	327

Experimental Results
hw-CBMC vs. Early Cutpoints

	hw-CBMC		**Early Cutpoints**	
Example	Time(s)	Mem(MB)	Time(s)	Mem(MB)
TOY-8	6.84	38	0.01	56
TOY-16	502.59	522	0.02	56
TOY-32	time out		0.06	56

Future Directions

- Heuristics for finding cutpoints
- Program analysis and optimization techniques to expose parallelism
- Handling more complex control flow
- Handling dynamic memory
- False inequivalence handling
- Integrating with other techniques to remove cycle-accuracy assumption