15-213

"The course that gives CMU its Zip!"

Verifying Programs with BDDs
Sept. 21, 2004

Topics

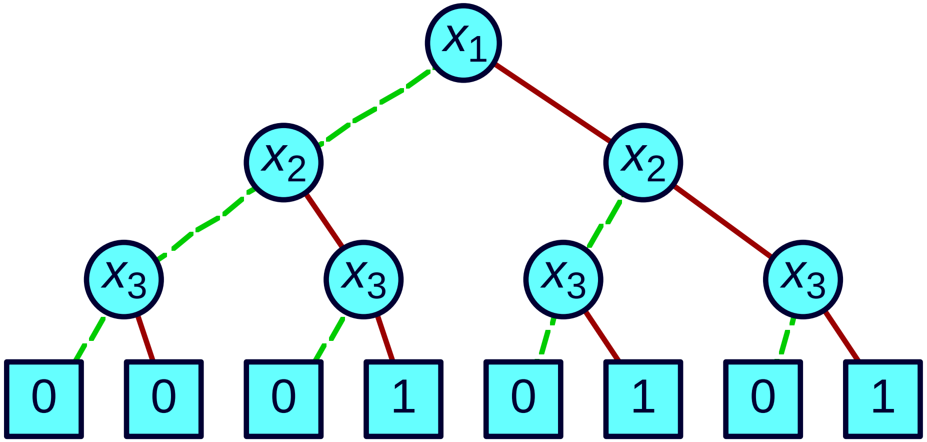
- Representing Boolean functions with Binary Decision Diagrams
- Application to program verification

Decision Structures

Truth Table

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

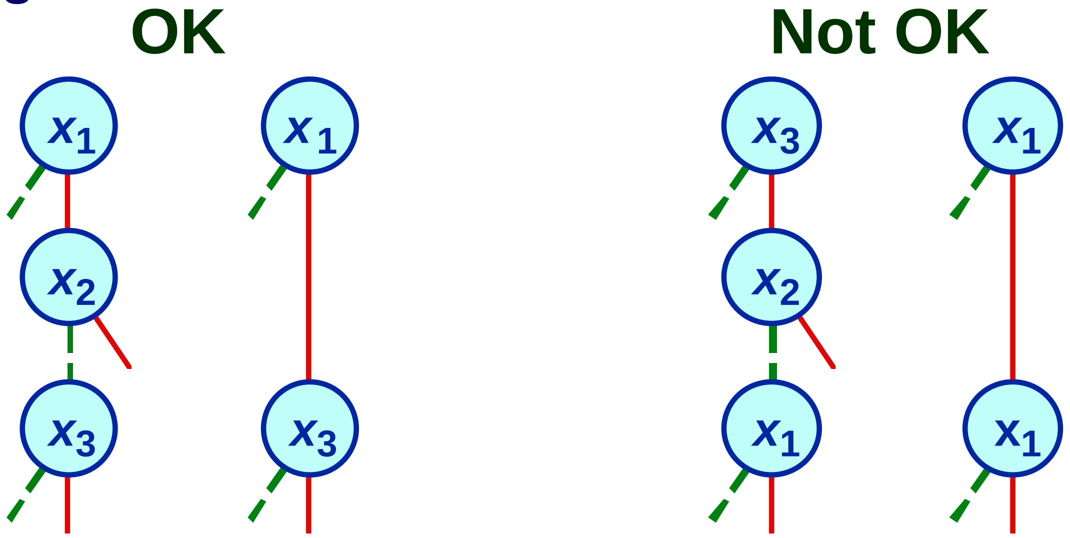
Decision Tree



- Vertex represents decision
- Follow green (dashed) line for value 0
- Follow red (solid) line for value 1
- Function value determined by leaf value.

Variable Ordering

- **Assign arbitrary total ordering to variables**
  - e.g., $x_1 < x_2 < x_3$
- **Variables must appear in ascending order along all paths**

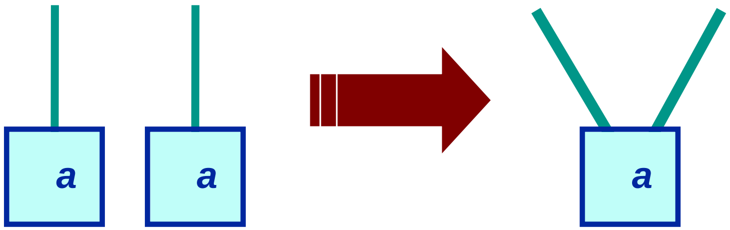


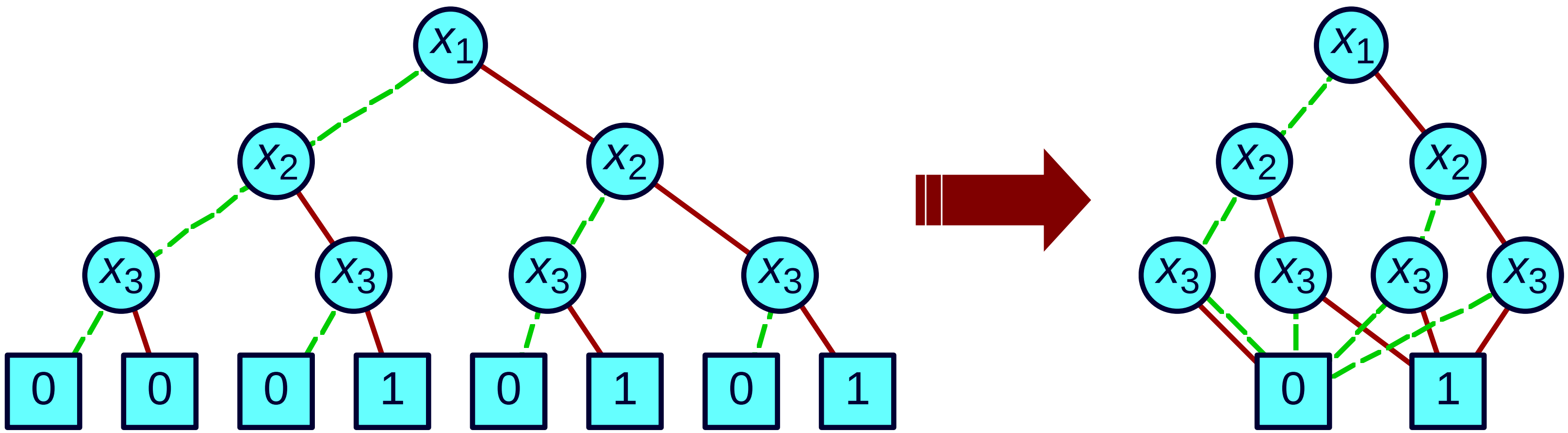
Properties

- **No conflicting variable assignments along path**
- **Simplifies manipulation**

Reduction Rule #1

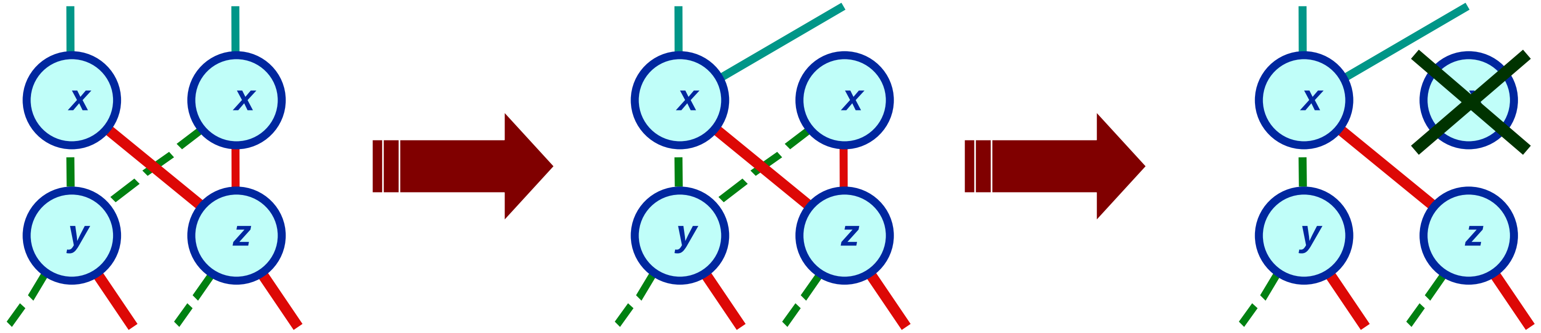
Merge equivalent leaves

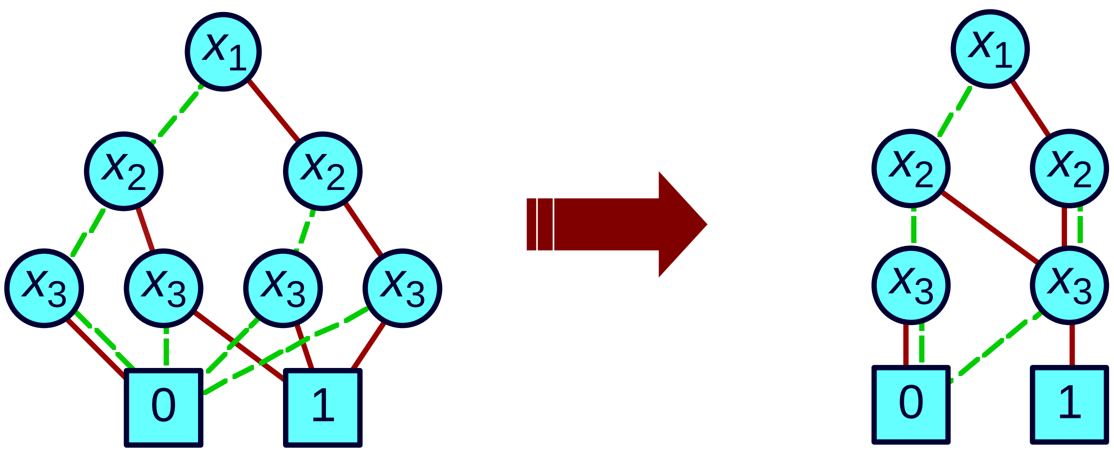




Reduction Rule #2

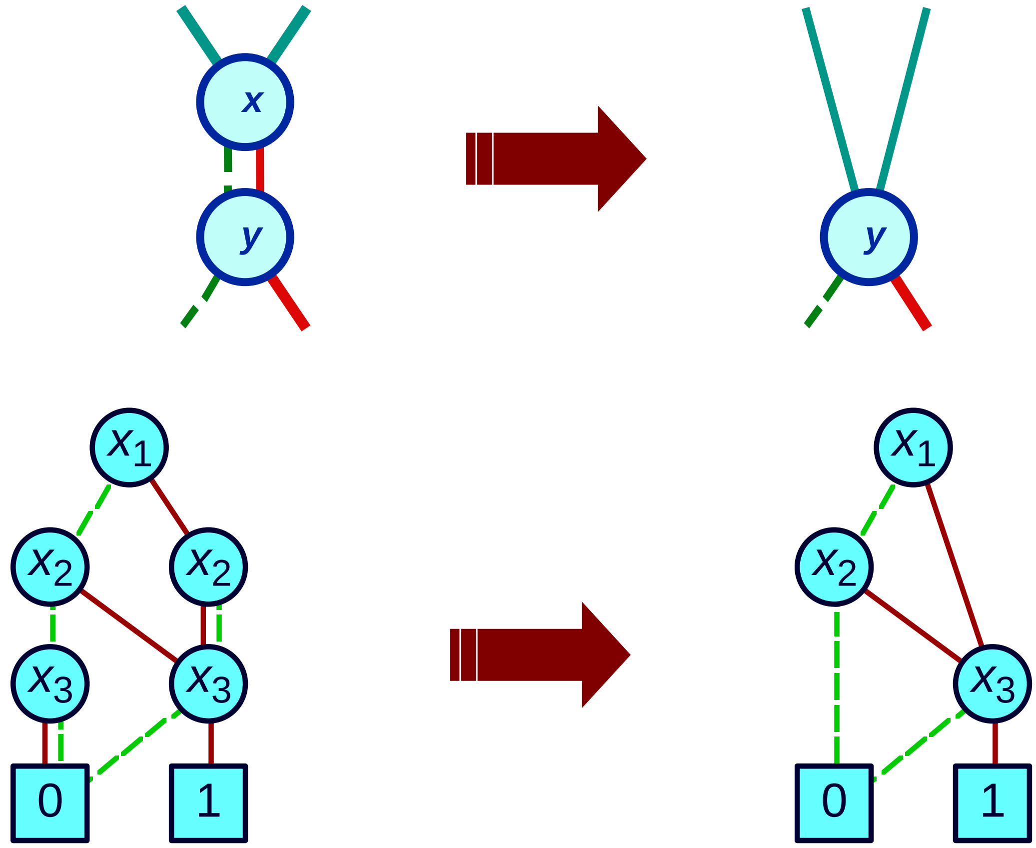
Merge isomorphic nodes





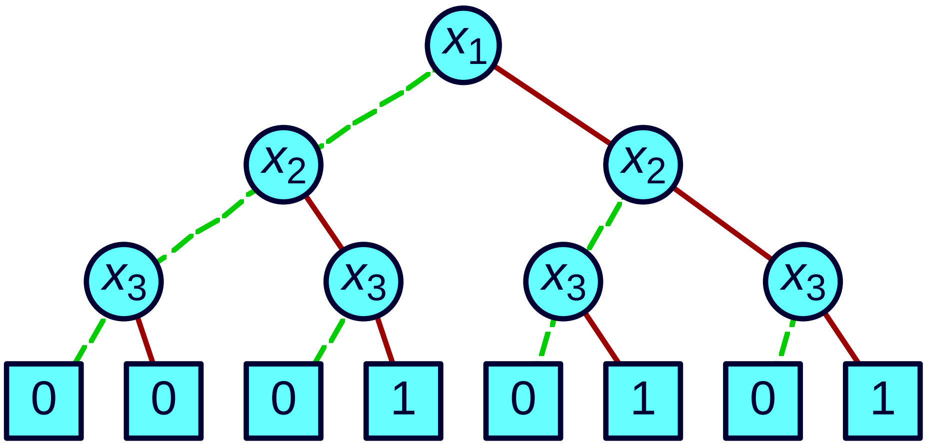
Reduction Rule #3

Eliminate Redundant Tests

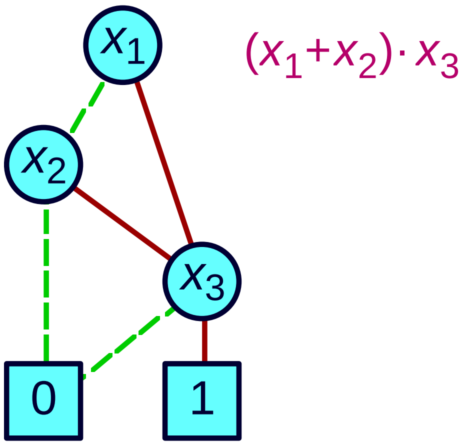


Example OBDD

Initial Graph



Reduced Graph



Canonical representation of Boolean function

- **For given variable ordering**
- **Two functions equivalent if and only if graphs isomorphic**
  - Can be tested in linear time
- **Desirable property: *simplest form is canonical*.**

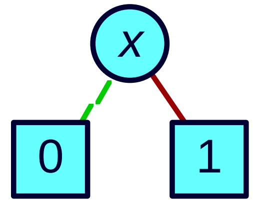
Example Functions

Constants

Unique unsatisfiable function

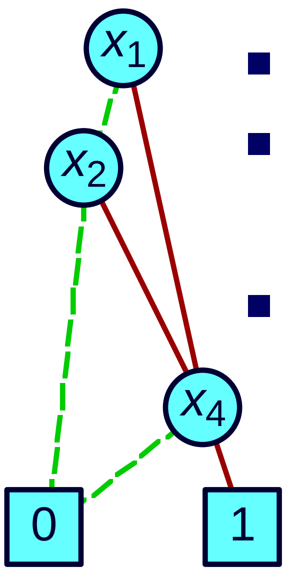
Unique tautology

Variable



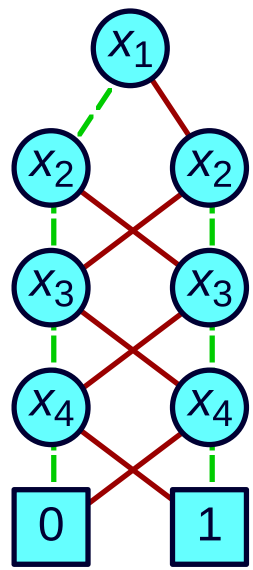
Treat variable as function

Typical Function



- $(x_1 \vee x_2) \wedge x_4$
- **No vertex labeled** x_3
  - independent of x_3
- **Many subgraphs shared**

Odd Parity



Linear representation

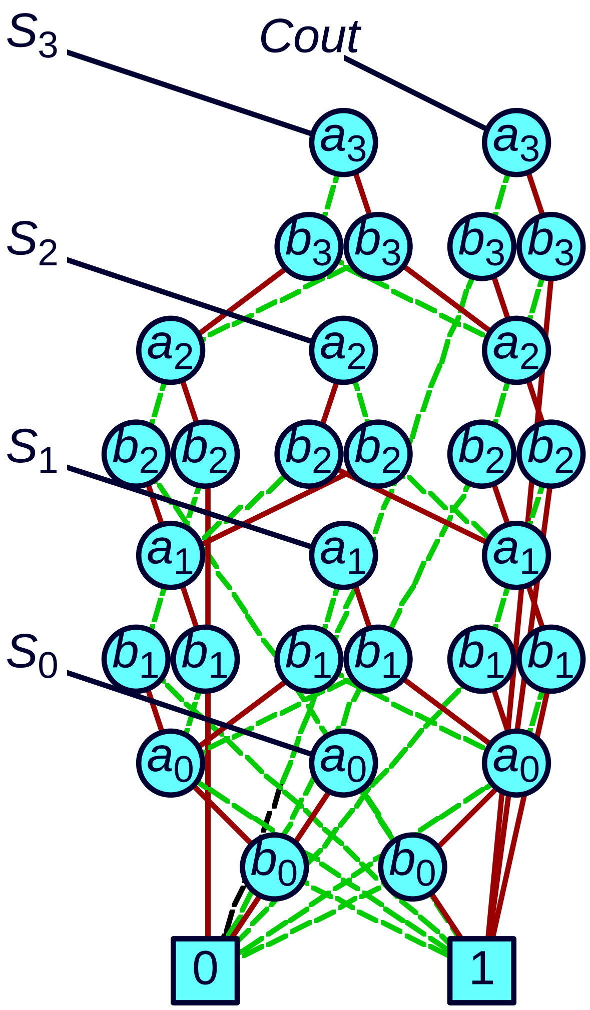
More Complex Functions

Functions

- Add 4-bit words a and b
- Get 4-bit sum s
- Carry output bit Cout

Shared Representation

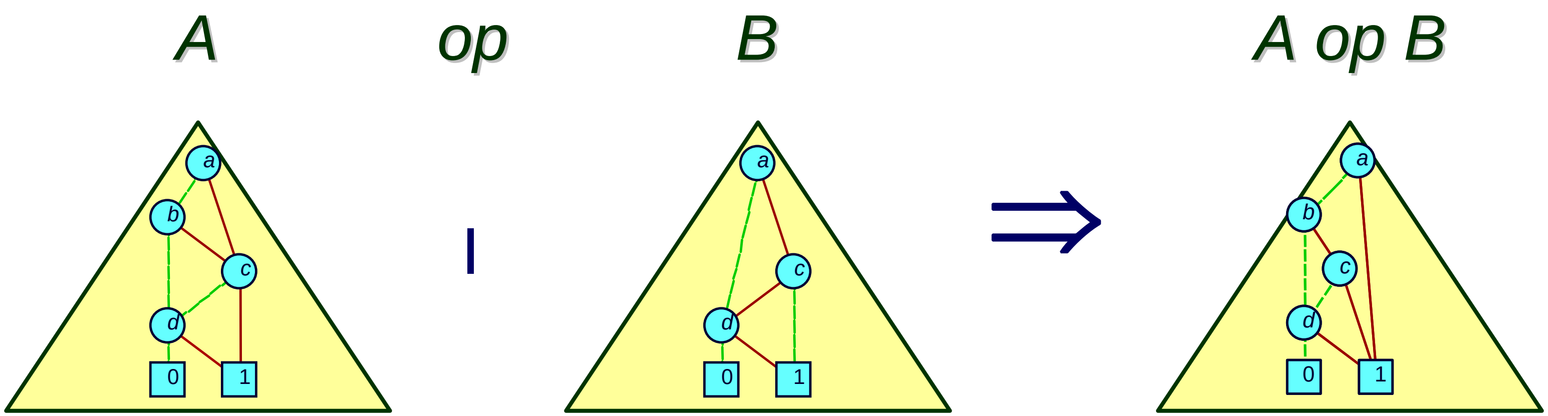
- Graph with multiple roots
- 31 nodes for 4-bit adder
- 571 nodes for 64-bit adder
- Linear growth!



Apply Operation

Concept

- **Basic technique for building OBDD from Boolean formula.**



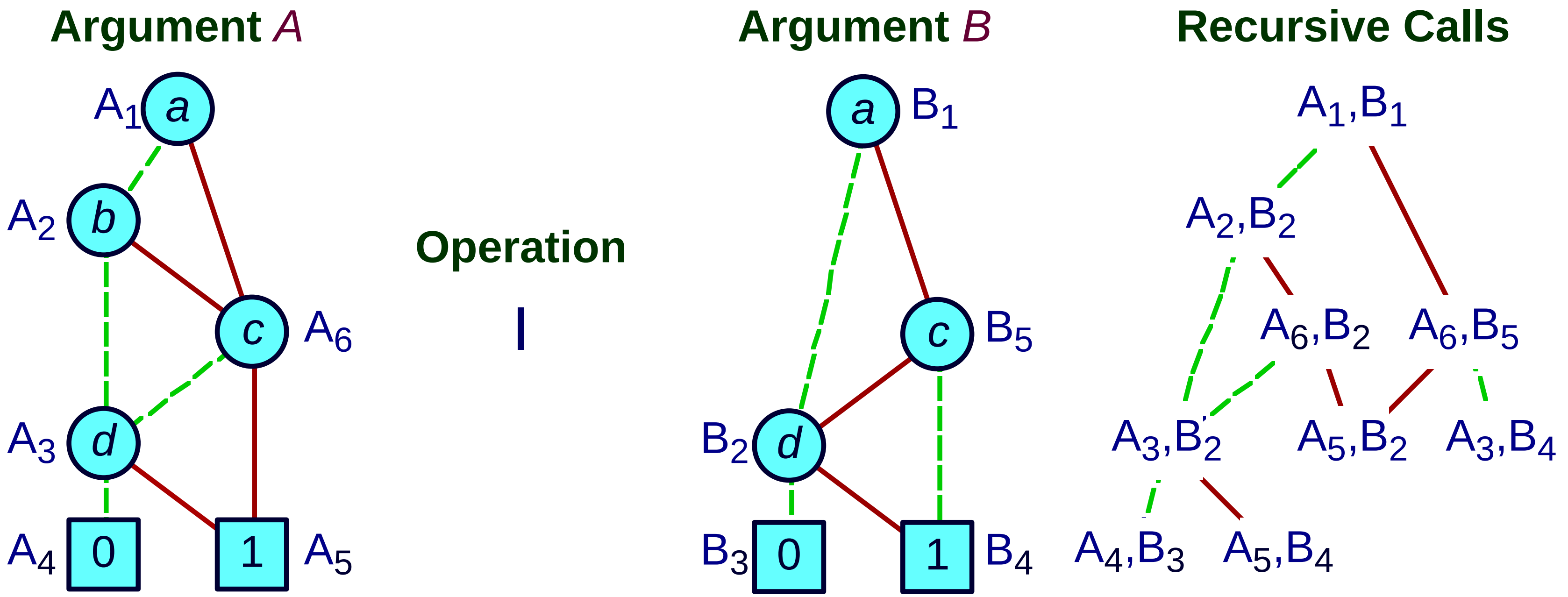
Arguments *A*, *B*, *op*

- ***A* and *B*: Boolean Functions**
  - **Represented as OBDDs**
- **op: Boolean Operation (e.g., ^, &, |)**

Result

- **OBDD representing composite function**
- *A op B*

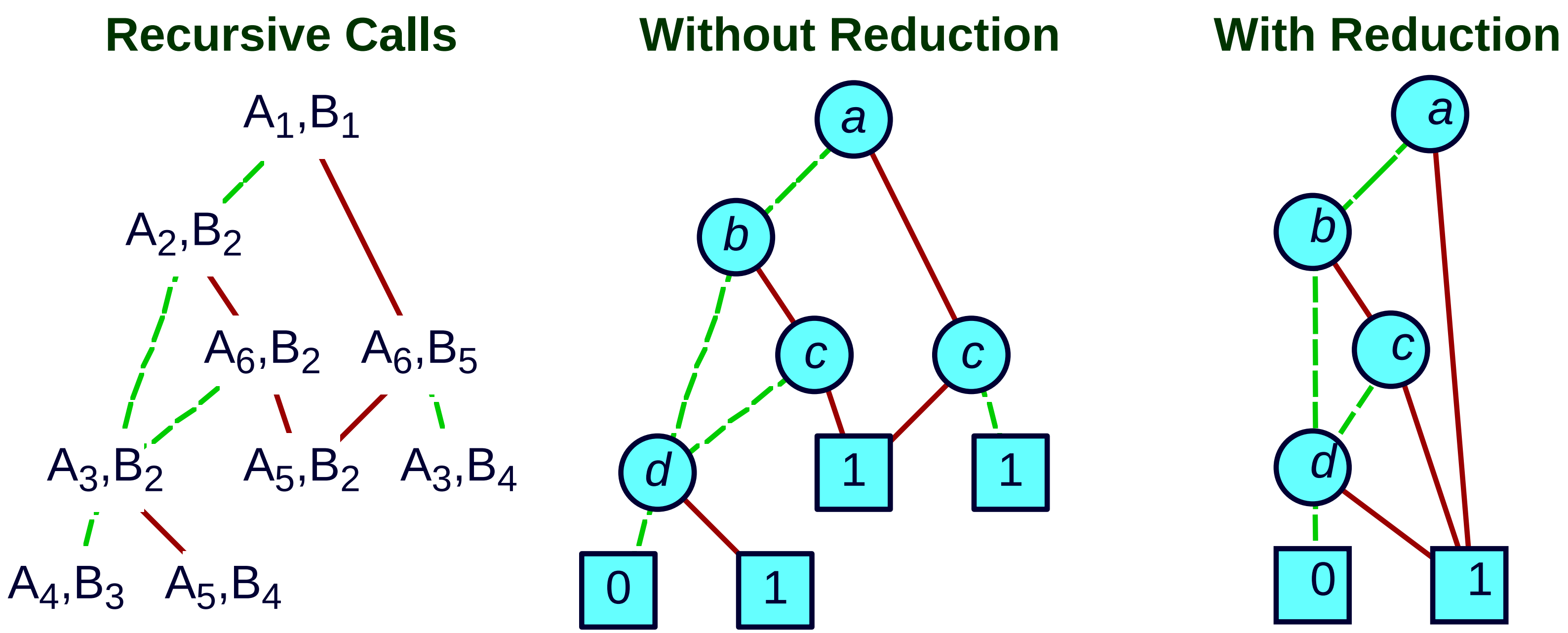
Apply Execution Example



Optimizations

- **Dynamic programming**
- **Early termination rules**

Apply Result Generation



- Recursive calling structure implicitly defines unreduced BDD
- Apply reduction rules bottom-up as return from recursive calls

Program Verification

```
int bitOr(int x, int y)
{
  return ~(~x & ~y);
}
```

```
int test_bitOr(int x, int y)
{
  return x | y;
}
```

Do these functions produce identical results?

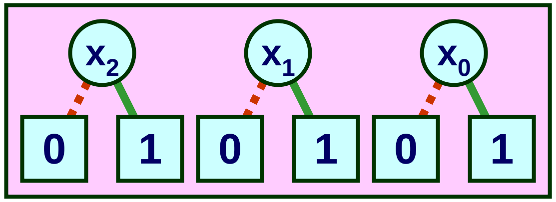
Straight-Line Evaluation

`x`
`y`
`v1 =  ~x`
`v2 =  ~y`
`v3 =  v1 & v2`
`v4 =  ~v3`
`v5 =  x | y`
`t  =  v4 == v5`

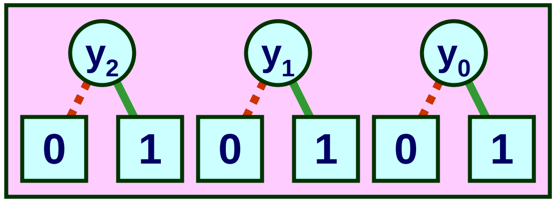
Symbolic Execution

(3-bit word size)

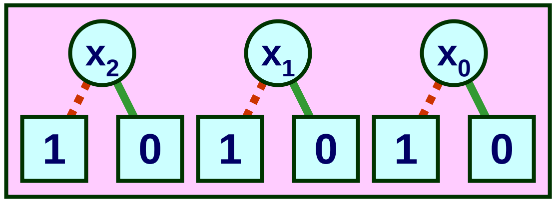
```
x
```



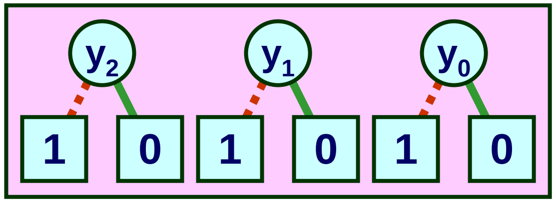

```
y
```




```
v1 =  ~x
```

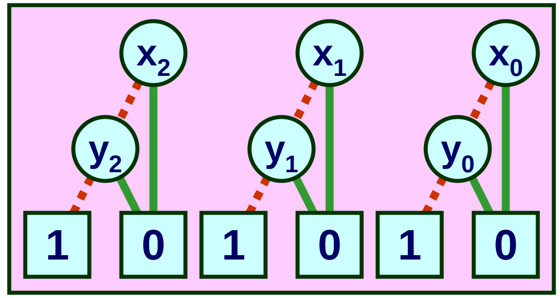



```
v2 =  ~y
```

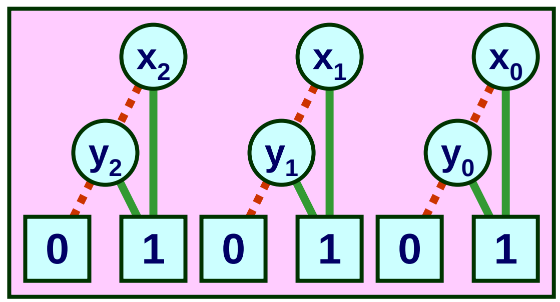


Symbolic Execution (cont.)

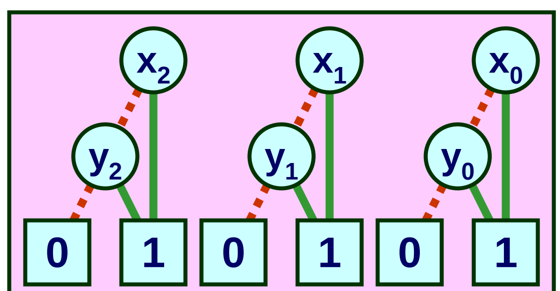
```
v3 =  v1 & v2
```



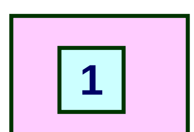

```
v4 =  ~v3
```




```
v5 =  x | y
```




```
t  =  v4 == v5
```



Counterexample Generation

```
int bitOr(int x, int y)
{
  return ~(~x & ~y);
}
```

```
int bitXor(int x, int y)
{
  return x ^ y;
}
```

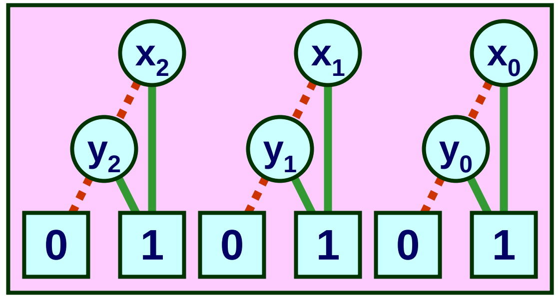
Find values of x & y for which these programs produce different results

Straight-Line Evaluation

```
x
y
v1 =  ~x
v2 =  ~y
v3 =  v1 & v2
v4 =  ~v3
v5 =  x ^ y
t  =  v4 == v5
```

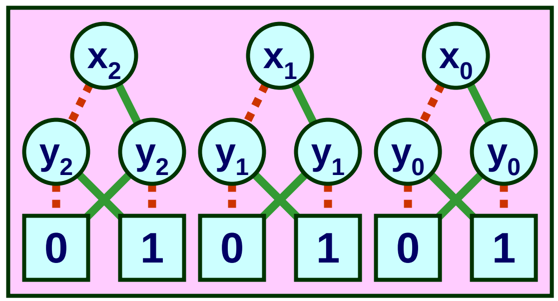
Symbolic Execution

```
v4 =  ~v3
```

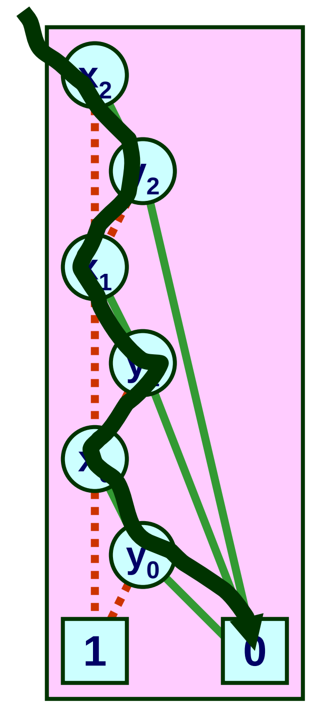



```
t  =  v4 == v5
```

```
v5 =  x ^ y
```



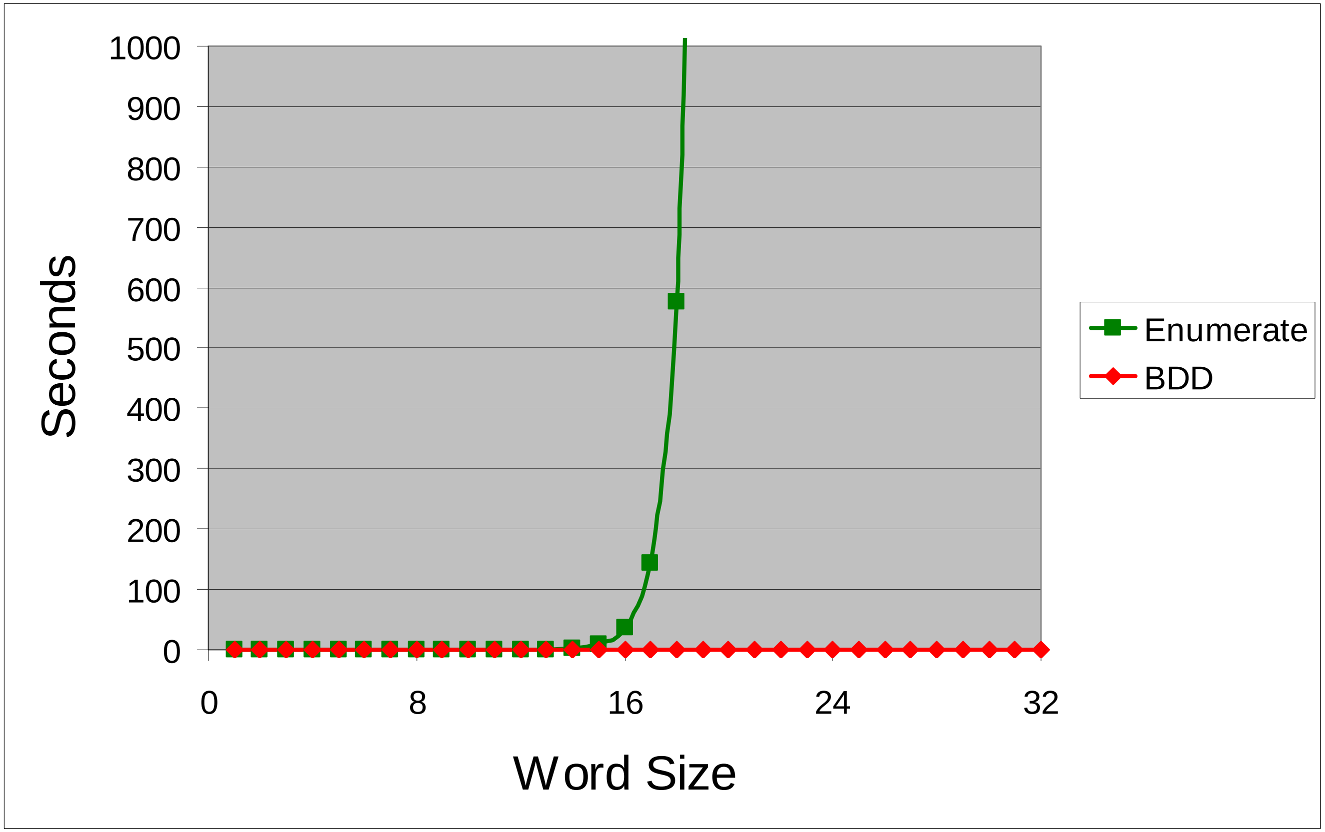

```
x = 111
y = 001
```



Performance: Good

```
int addXY(int x, int y)
{
  return x+y;
}
```

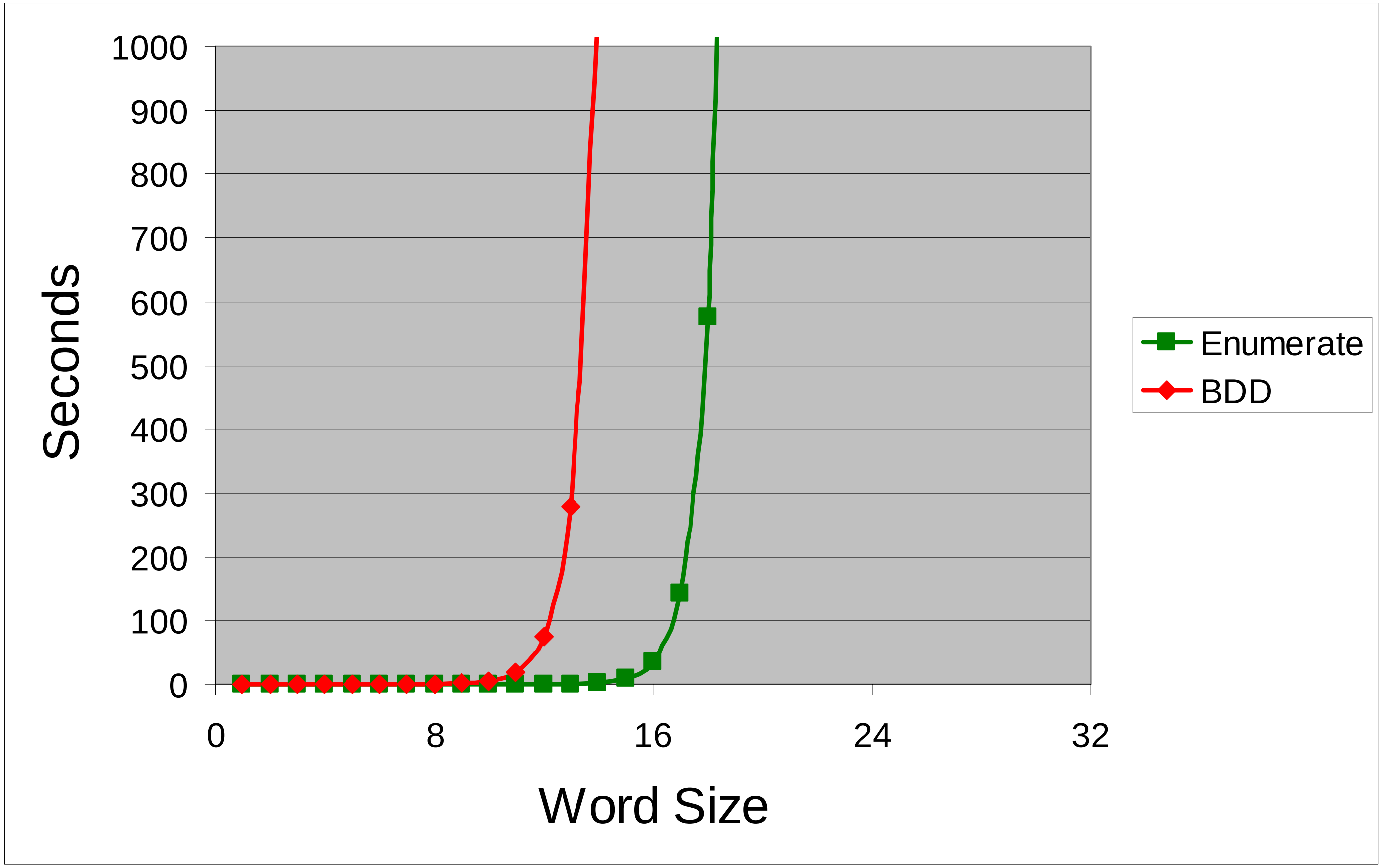
```
int addYX(int x, int y)
{
  return y+x;
}
```



Performance: Bad

```
int mulXY(int x, int y)
{
  return x*y;
}
```

```
int mulYX(int x, int y)
{
  return y*x;
}
```



What if Multiplication were Easy?

```
int factorK(int x, int y)
{
  int K = XXXX...X;
  int rangeOK =
     1 < x && x <= y;
  int factorOK =
     x*y == K;
  return
    !(rangeOK && factorOK);
}
```

```
int one(int x, int y)
{
  return 1;
}
```

Evaluation

Strengths

- Provides 100% guarantee of correctness
- Performance very good for Datalab functions

Weaknesses

- Important integer functions have exponential blowup
- Not practical for programs that build and operate on large data structures

Some History

Origins

- **Lee 1959, Akers 1976**
  - Idea of representing Boolean function as BDD
- **Hopcroft, Fortune, Schmidt 1978**
  - Recognized that ordered BDDs were like finite state machines
  - Polynomial algorithm for equivalence
- **Bryant 1986**
  - Proposed as useful data structure + efficient algorithms
- **McMillan 1993**
  - Developed symbolic model checking
  - Method for verifying complex sequential systems
- **Bryant 1991**
  - Proved that multiplication has exponential BDD
  - No matter how variables are ordered