15-213

"The course that gives CMU its Zip!"

Verifying Programs with BDDs
Sept. 22, 2006

Topics

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

Verification Example

```
int abs(int x) {
  int mask = x>>31;
  return (x ^ mask) + ~mask + 1;
}
```

```
int test_abs(int x) {
  return (x < 0) ? -x : x;
}
```

Do these functions produce identical results?

How could you find out?

How about exhaustive testing?

More Examples

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

?
=

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

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

?
=

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

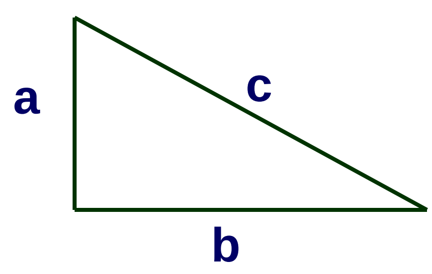
How Can We Verify Programs?

Testing

- Exhaustive testing not generally feasible
- Currently, programs only tested over small fraction of possible cases

Formal Verification

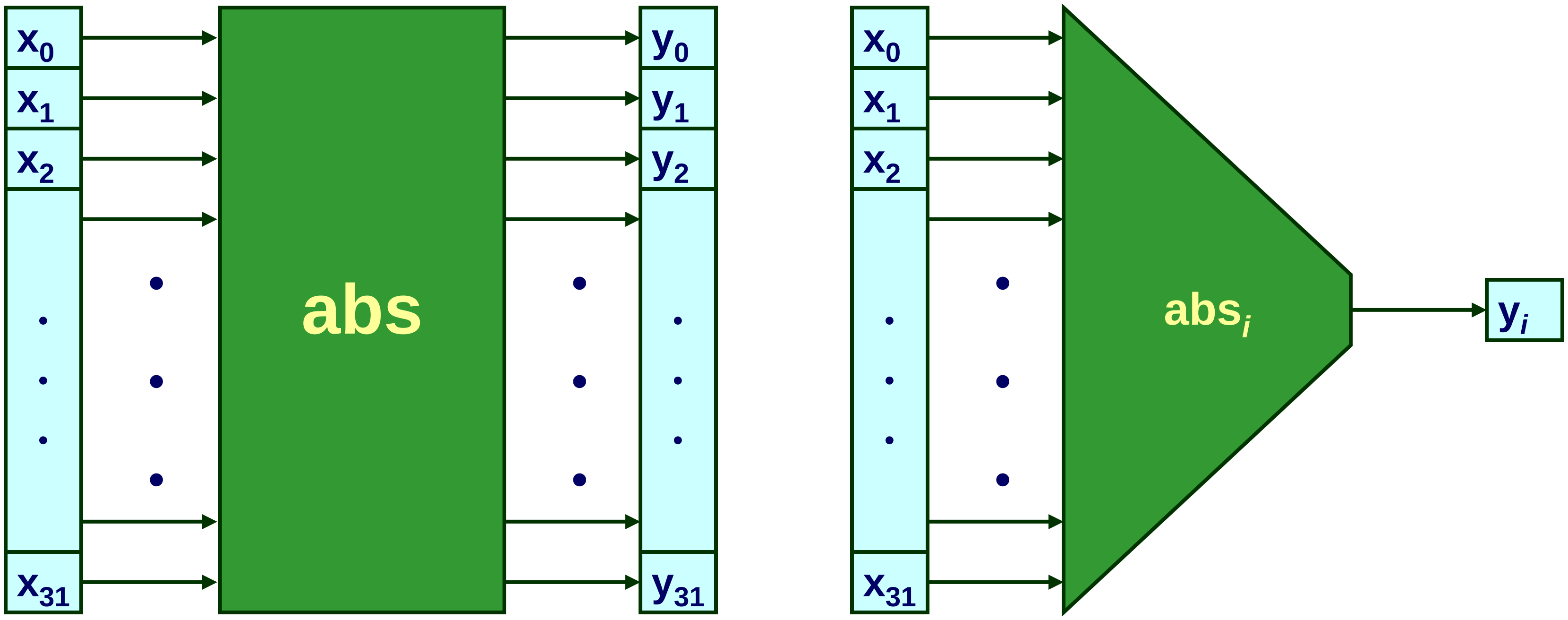
- Mathematical “proof” that code is correct



- Did Pythagoras show that $a^2 + b^2 = c^2$ by testing?

Bit-Level Program Verification

```
int abs(int x) {
  int mask = x>>31;
  return (x ^ mask) + ~mask + 1;
}
```



- View computer word as 32 separate bit values
- Each output becomes Boolean function of inputs

Extracting Boolean Representation

```
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`

Tabular Function Representation

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

- **List every possible function value**

Complexity

- **Function with *n* variables**

Algebraic Function Representation

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

$x_2 \cdot x_3$

$x_1 \cdot x_3$

- $f(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$
- **Boolean Algebra**

Complexity

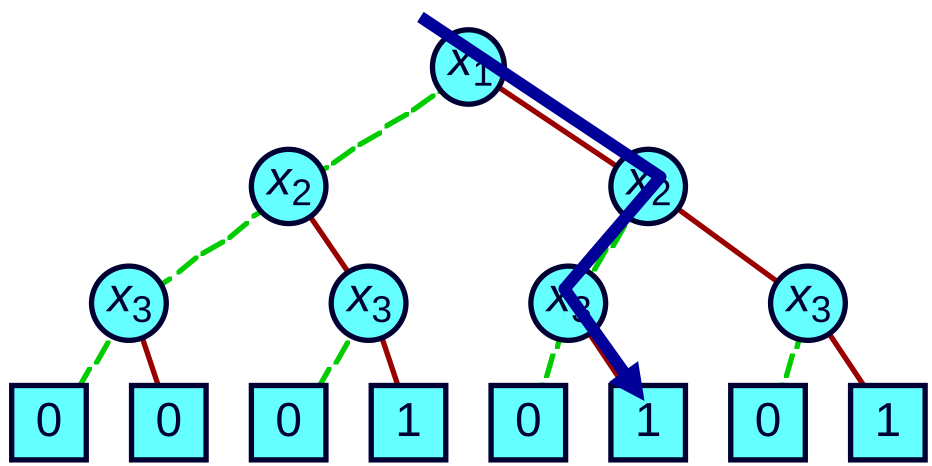
- **Representation**
- **Determining properties of function**
  - E.g., deciding whether two expressions are equivalent

Tree Representation

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

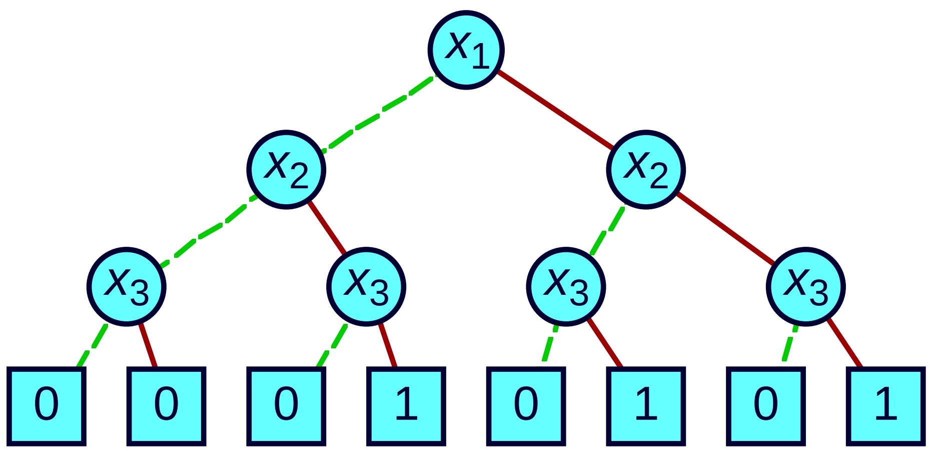


- Vertex represents decision
- Follow green (dashed) line for value 0
- Follow red (solid) line for value 1
- Function value determined by leaf value

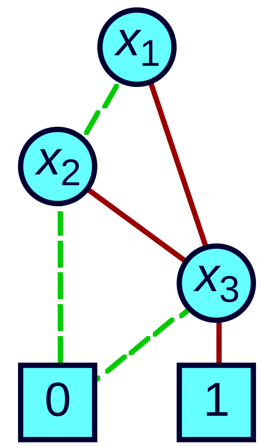
Complexity

Ordered Binary Decision Diagrams

Initial Tree



Reduced Graph



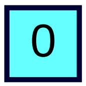
$(x_1 + x_2) \cdot x_3$

Canonical representation of Boolean function

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

Example Functions

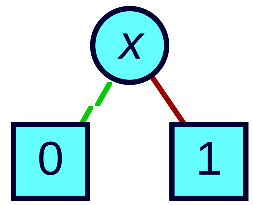
Constants



Unique unsatisfiable function

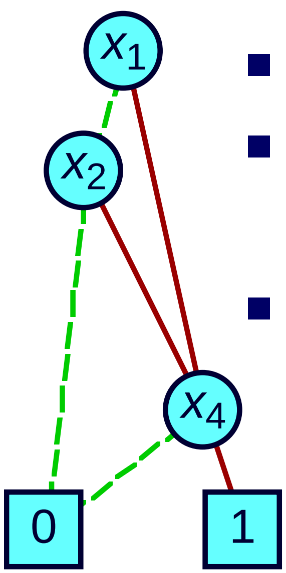
Unique tautology

Variable



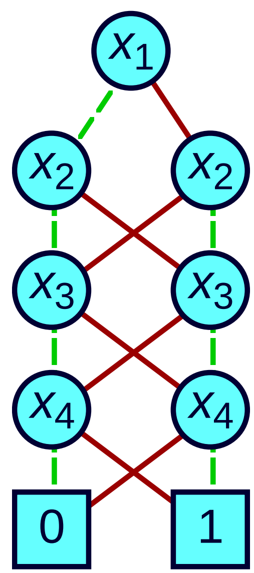
Treat variable as function

Typical Function



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

Odd Parity



Linear representation

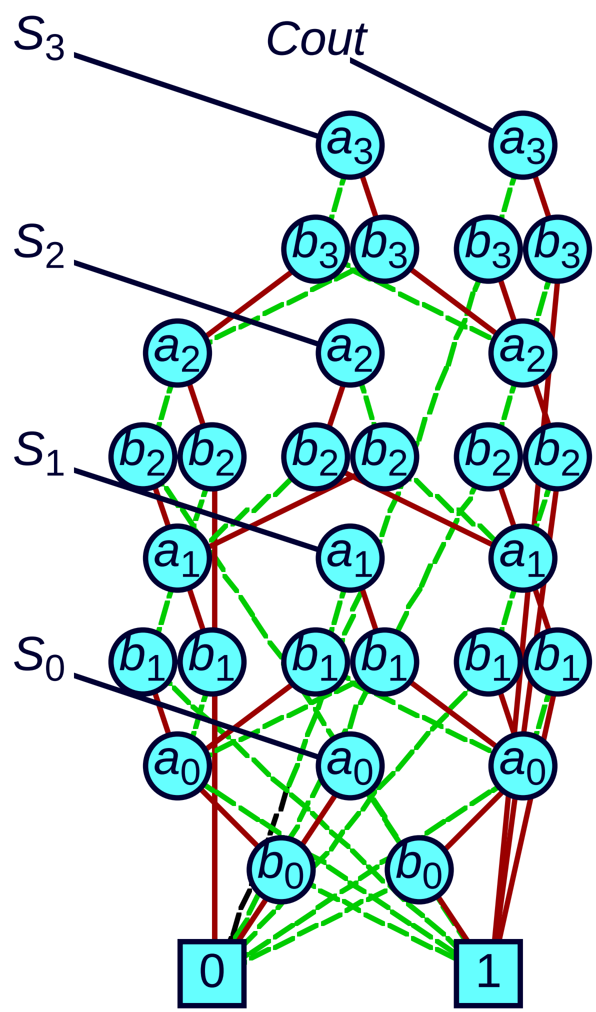
More Complex Functions

Functions

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

Shared Representation

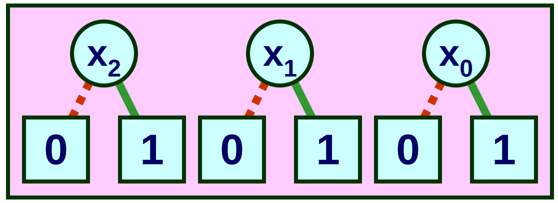
- Graph with multiple roots
- 31 nodes for 4-bit adder
- 571 nodes for 64-bit adder
- Linear growth!



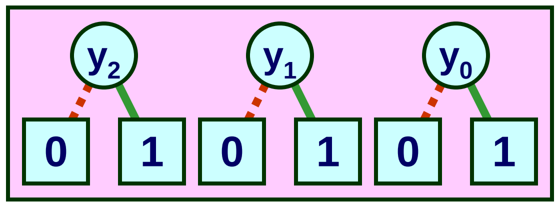
Symbolic Execution

(3-bit word size)

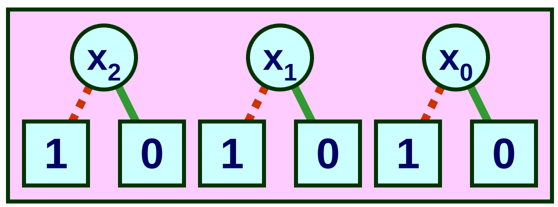
```
x
```



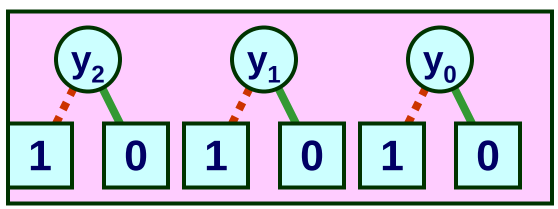

```
y
```




```
v1 =  ~x
```

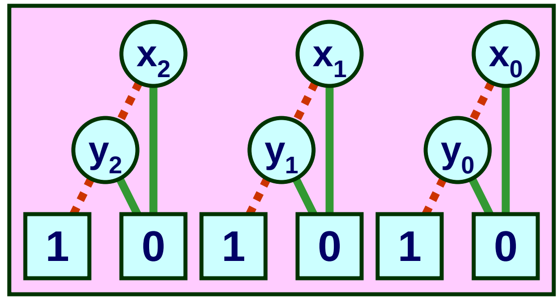



```
v2 =  ~y
```

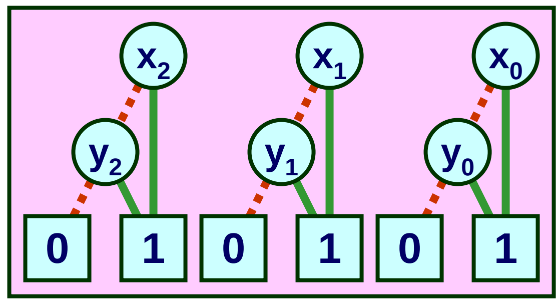


Symbolic Execution (cont.)

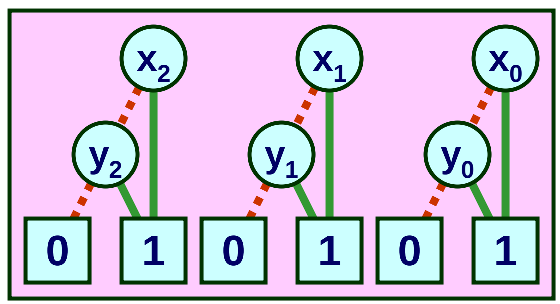
```
v3 =  v1 & v2
```



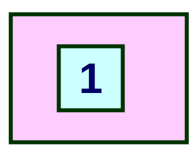

```
v4 =  ~v3
```




```
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;
}
```

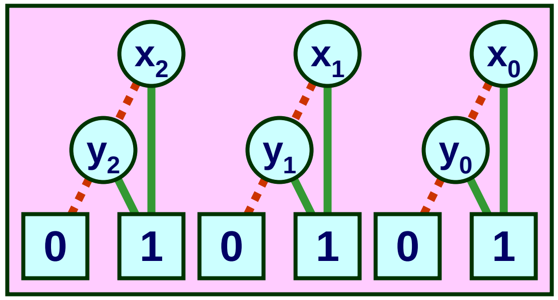
Find values of x & y for which these programs produce different results

Straight-Line Evaluation

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

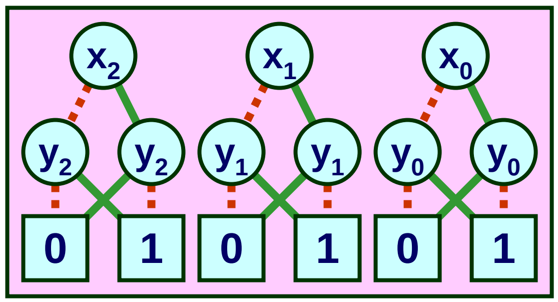
Symbolic Execution

```
v4 =  ~v3
```

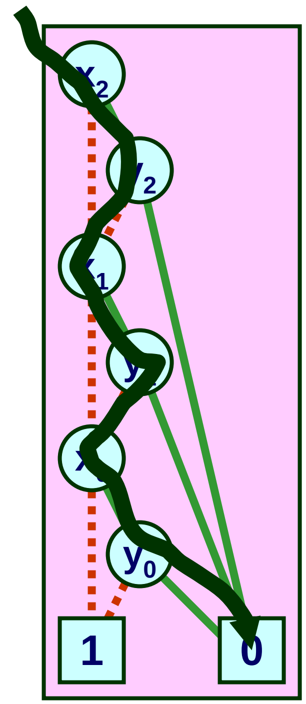



```
t  =  v4 == v5
```

```
v5 =  x ^ y
```



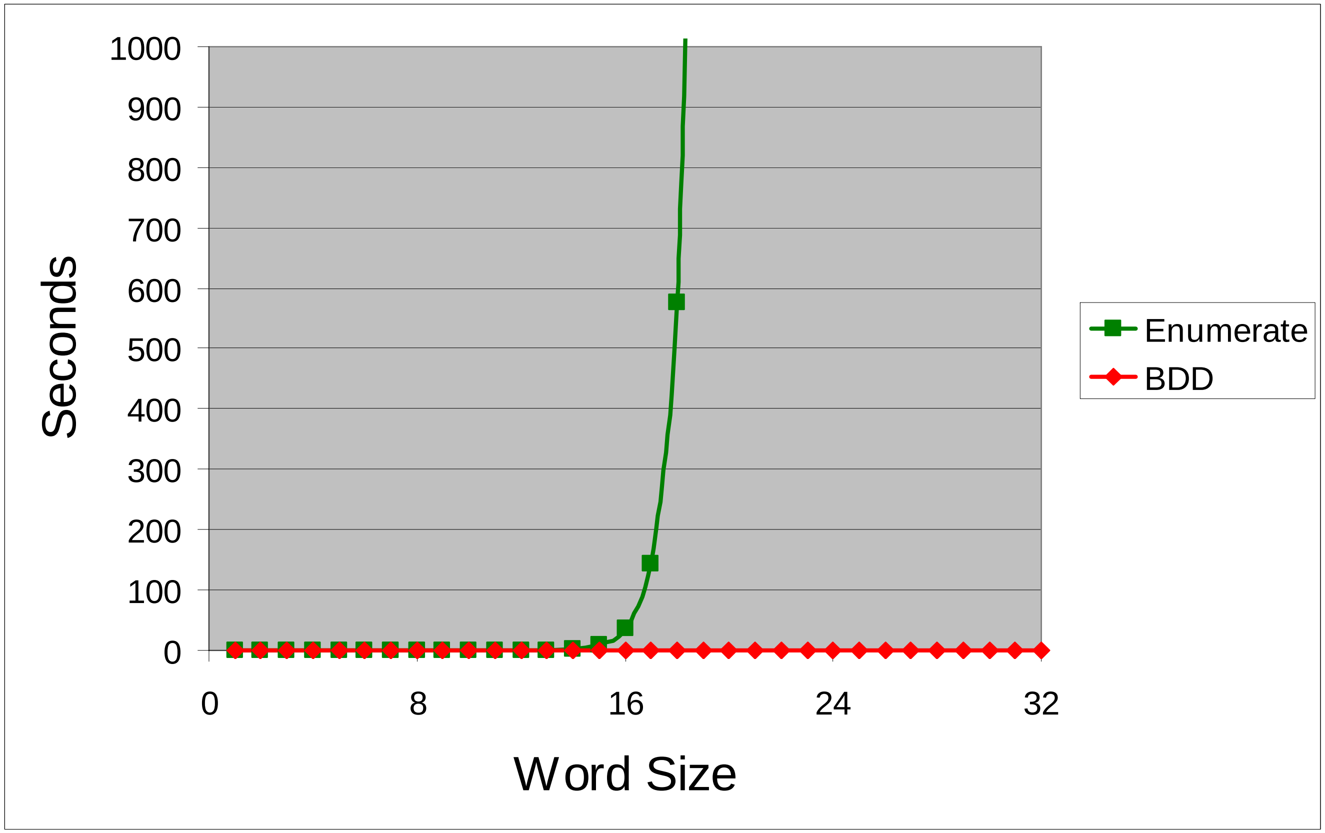

```
x = 111
y = 001
```



Performance: Good

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

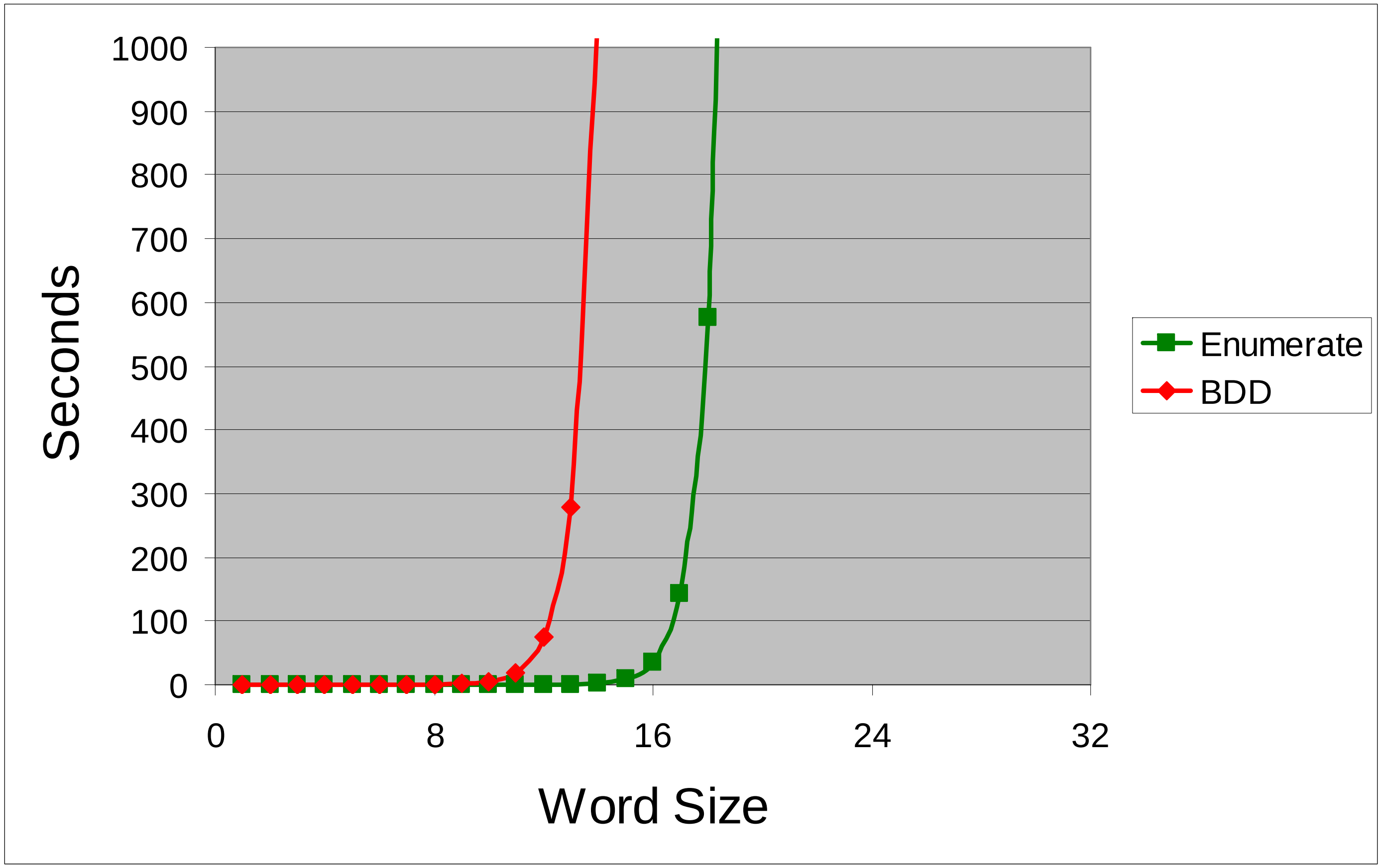
```
int addYX(int x, int y)
{
  return y+x;
}
```



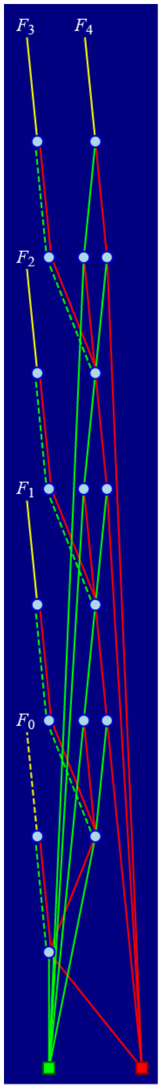
Performance: Bad

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

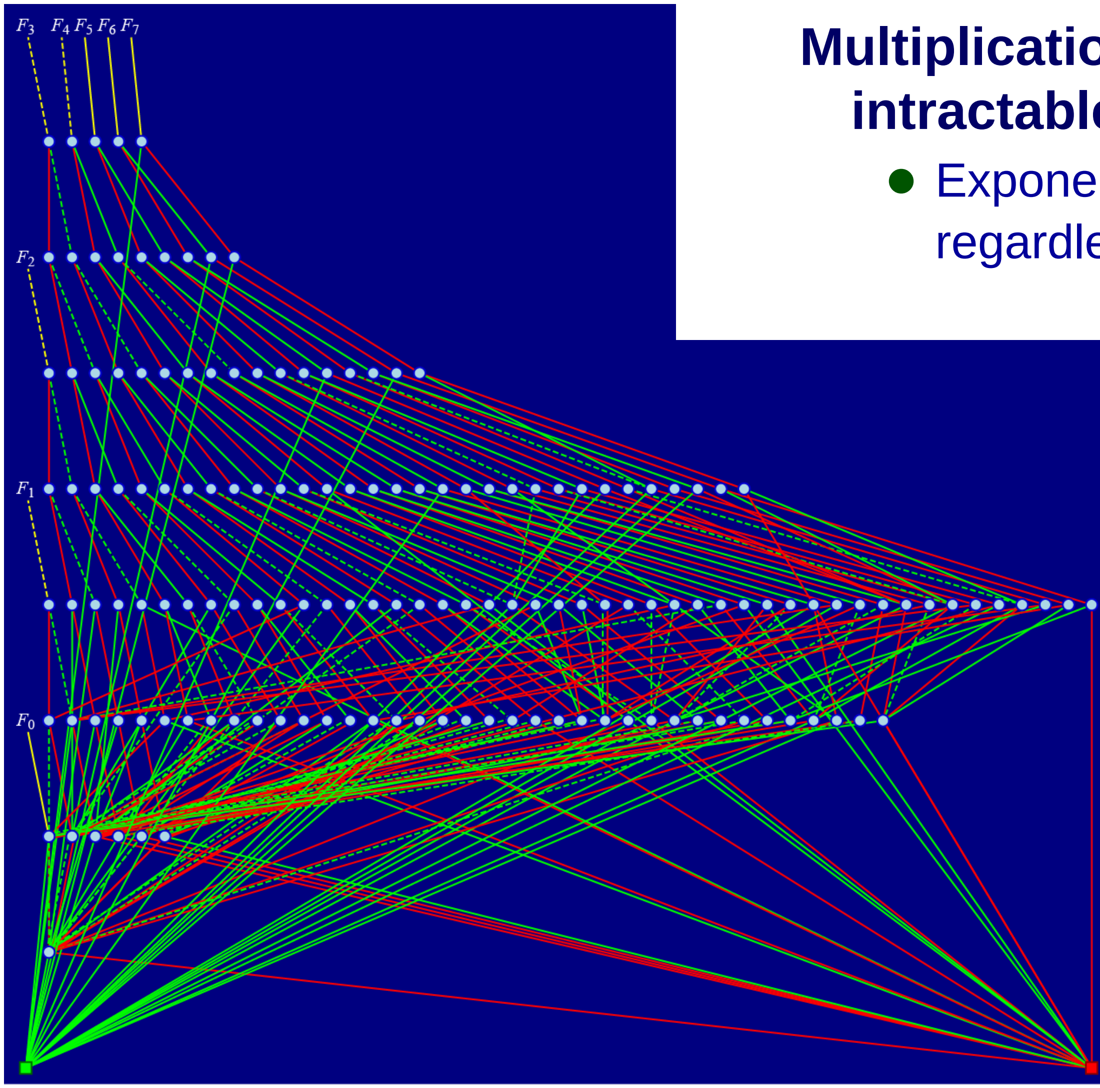
```
int mulYX(int x, int y)
{
  return y*x;
}
```



Why Is Multiplication Slow?



Add-4



Multiplication-4

Multiplication function intractable for BDDs

- Exponential growth, regardless of variable ordering

Node Counts

Bits	Add	Mult
4	21	155
8	41	14560

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;
}
```

Dealing with Conditionals

```
int abs(int x)
{
  int r;
  if (x < 0)
    r = -x;
  else
    r =  x;
  return r;
}
```

	Context	r defined	r value
`x`	`1`	`0`	`0`
`t1 = x<0`	`1`	`0`	`0`
`v1 = -x`	`t1`	`0`	`0`
`r = v1`	`t1`	`t1`	`t1?v1:0`
`r = x`	`!t1`	`1`	`t1?v1:x`
`v2 = r`	`1`	`1`	`t1?v1:x`

During Evaluation, Keep Track of:

- **Current Context: Under what condition would code be evaluated**
- **Definedness (for each variable)**
  - Has it been assigned a value

Dealing with Loops

```
int ilog2(unsigned x)
{
  int r = -1;
  while (x) {
    r++; x >>= 1;
  }
  return r;
}
```

Unroll

- **Turn into bounded sequence of conditionals**
  - Default limit = 33
- **Signal runtime error if don't complete within limit**

Unrolled

```
int ilog2(unsigned x)
{
  int r = 31;
  if (x) {
    r++; x >>= 1;
  } else return r;
  if (x) {
    r++; x >>= 1;
  } else return r;
  . . .
  if (x) {
    r++; x >>= 1;
  } else return r;
  error();
}
```

Evaluation

Strengths

- Provides 100% guarantee of correctness
- Performance very good for simple arithmetic 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 1987**
  - Developed symbolic model checking
  - Method for verifying complex sequential systems
- **Bryant 1991**
  - Proved that multiplication has exponential BDD
  - No matter how variables are ordered