(*Lec 4*) Binary Decision Diagrams: Manipulation

What you know

- Basic BDD data structure
  - DAG representation
  - How variable ordering + reduction = canonical
- A few immediate applications
  - eg, trivial tautology checking

What you don't know

- Algorithms to build one efficiently
- Most useful implementation tricks for efficiency

  - (Thanks to Randy Bryant & Karl Brace for nice BDD pics+slides)

Copyright Notice

Where Are We?

- More BDDs -- now how to actually implement them

	M	T	W	Th	F	
Aug	27	28	29	30	31	1
Sep	3	4	5	6	7	2
	10	11	12	13	14	3
	17	18	19	20	21	4
	24	25	26	27	28	5
Oct	1	2	3	4	5	6
	8	9	10	11	12	7
	15	16	17	18	19	8
	22	23	24	25	26	9
	29	30	31	1	2	10
Nov	5	6	7	8	9	11
	12	13	14	15	16	12
Thnxgive	19	20	21	22	23	13
	26	27	28	29	30	14
Dec	3	4	5	6	7	15
	10	11	12	13	14	16

Introduction
Advanced Boolean algebra
JAVA Review
Formal verification
2-Level logic synthesis
Multi-level logic synthesis
Technology mapping
Placement
Routing
Static timing analysis
Electrical timing analysis
Geometric data structs & apps

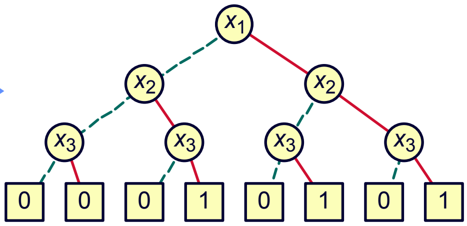
Handouts

- Physical
  - **Lecture 04 -- BDD Manipulation**
- Electronic
  - **HW2 is out on the web site.**
  - **(Also, note all TA/Prof office hours are on web site, in "About Class")**
- Assignments
  - **Proj3 out late today or tomorrow: Building a simple BDD package in JAVA, then using it for some example verification tasks**

BDDs: Creation

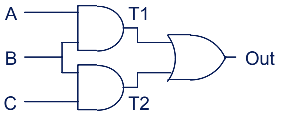
- First goal
  - Input: a gate-level logic network
  - Output: a BDD that represents Boolean function of the network
- Question: how?
  - Cannot afford to do it the way we developed the BDD idea…
  - …cannot build the full decision diagram from the truth table and then do the reduction to canonical form

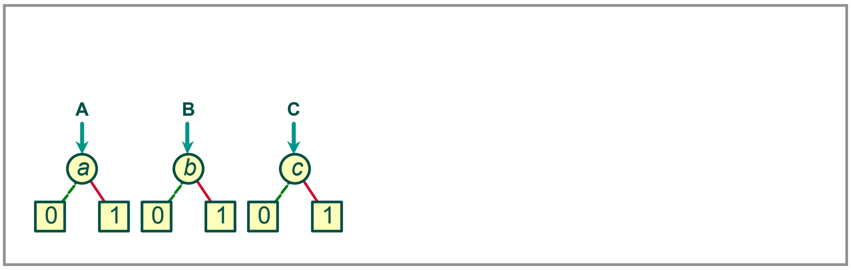
x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



BDDs: Creation

- So, how do we do it?
  - Build up BDD *incrementally*, by walking the gate network
  - Each input is a BDD
  - Each gate becomes an operator that produces a new BDD





BDDs: Creation

- Strategy
  - Represent data as *set* of OBDDs, all with *identical* variable orderings
  - Express solution method as a *sequence* of symbolic operations
  - Implement each operation by OBDD manipulation
- What exactly is a *symbolic* operation?
  - Think of it like a C language subroutine that works on BDDs...

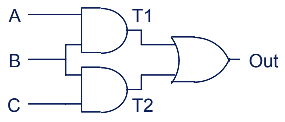
```
bdd *OPERATOR( bdd *bddInput1,   bdd *bddInput2)
```

  - ... ie, it's a routine that takes 2 pointers to BDD structures, builds a new BDD structure, and returns you a pointer to it
- Algorithmic properties
  - Arguments are OBDDs with identical variable orderings.
  - Result is OBDD with same ordering.
  - Called the "Closure Property":   OBDDs go in, OBDDs come out

BDDs: Creation

- How do we do it?
- Option 1 (The obvious approach)
  - Build BDD operators for all the things you can do directly on logic gate networks, or Boolean functions themselves
    - Simple operators: NOT(a), AND(a,b) OR(a,b), EXOR(a,b), etc
    - Complex operators: COFACTOR(F,x), CONSENSUS(F,x), etc

To build the BDD for Out...



might want something like this:

BDDs: Creation

- What's wrong with this?
  - Nothing, actually. Perfectly workable...
  - ...but *not* the most elegant way of doing it
  - And not the way people actually do it these days
- Option 2: (Subtle approach)
  - Build a few, critical, core operators on BDDs
  - Build up ALL the other operators you like using *ONLY* this small set of building-block operators
- Advantage
  - Actually less programming work, since you only have to build a few 'complicated' routines
  - Makes it much easier to add new operators later

BDDs: RESTRICT Operator

- So, what are the core operators we need?
  - Surprisingly, there are only 2: RESTRICT, ITE
- RESTRICT(function F, variable v, constant k)
  - This is just the Shannon cofactor of F wrt variable v
  - Example: RESTRICT(F, v, 0) == $F_{v'}$
    RESTRICT(F, v, 1) == F_v
  - Remember: F is represented as a BDD,
    result of RESTRICT is another BDD
  - ie, we are really implementing something like this:

```
bdd *RESTRICT(bdd *F, variable v, int k)
```

BDDs: ITE If_Then_Else Operator

ITE(function F, function G, function H)

- Called the IF-THEN-ELSE operator
- ITE(F,G, H) is itself another Boolean function
- Couple of different ways to describe just what ITE is...

In pseudo-C...

```
ITE(F,G,H)(x1,...,xn) {
   if ( F(x1, ..., xn)==1 )
   then return ( G(x1,..., xn) )
   else return ( H(x1,..., xn) )
}
```

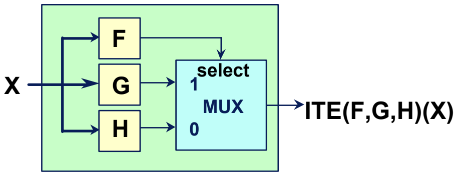
In Boolean algebra...

- Let X = x1,x2,...xn, just for convenience of notation

$$\text{ITE}(F,G,H)(X) = F(X)\bullet G(X) + \overline{F(X)}\bullet H(X)$$

BDDs: ITE

As hardware...



- This one is also pretty easy to remember
- ITE is like an IF-THEN-ELSE in software, or a MUX in hardware

Critical things to remember are:

- ITE(F,G,H) IS A NEW FUNCTION
- F,G,H are inputs to ITE operator, represented as BDDs
- ITE(F,G,H) creates (is) a new BDD

BDDs: ITE

Conventions

- Usually see the input functions to ITE written like this:

ITE(I, T, E)

- ...just for convenience in remembering that the first is the IF function, the second is the THEN function, last is ELSE function

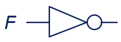
Back to BDD core operators

- Sort of easy to see why you need RESTRICT -
  - Lots of useful things you can build out of cofactors
- What good is ITE? What can you do with it?
  - Actually, almost everything...

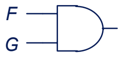
BDDs: Uses of ITE

Can make any basic 2-input gate out of ITE

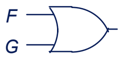
Not



And

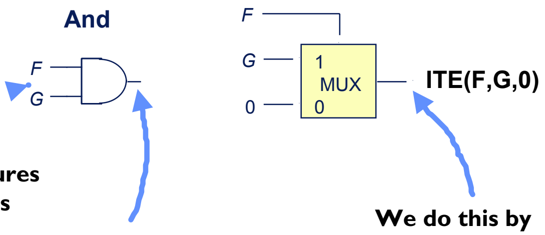


Or



BDDs: Uses of ITE

- Let's be *precise* abut how we intend to use ITE



Have BDD data structures for functions F(X), G(X)

We want to create the BDD for the function (F•G)(X)

We do this by calling the ITE routine on the BDD data structures for function F, for function G, and for constant BDD== '0'

ITE returns as value a pointer to a new BDD, which == (F•G)

BDDs: Creation

- If you can implement RESTRICT, ITE…
  - **You can build up basically anything useful you need to manipulate Boolean functions, or build BDDs of actual gate networks**
- New question
  - **How do we actually *implement* RESTRICT and ITE as algorithms?**
- Answers
  - **Shannon cofactoring to the rescue, again!**
  - **Recursive divide and conquer, again!**
  - **Need a couple of special data structures to make it efficienct**
  - **Start by looking at ITE**

BDDs: Implementing ITE

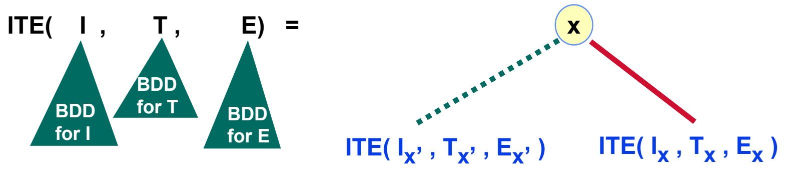
Assumptions

- There is a global variable ordering: x1 < x2 < x3 < ... < xn
  - If I give you a few variables, you can tell me which is “smallest”
  - Example: Input: x2, x7 smaller(x2, x7) == x2
- We will use *multi-rooted* DAGs (MRDAGs) for the BDDs
  - Amazingly enough, it’s *simpler* to implement
  - Newly created BDDs always try to share nodes in existing BDDs
- Example: what is MRDAG BDD for functions F(a,b)=a+b, G(a,b)=b ?

BDDs: Implementing ITE

Key implementation idea: Use Shannon expansion

- Suppose X=x1, x2, ... x, ... xn
- Suppose we have 3 functions I(X), T(X), E(X), then...
- ...ie, ITE can itself be decomposed recursively!
- And, if you want to think BDDs...


ITE(I , T , E) =
BDD for I
BDD for T
BDD for E
x
ITE($I_{x'}$, $T_{x'}$, $E_{x'}$)
ITE(I_x , T_x , E_x)

BDDs: Implementing ITE

So, what do we need to do recursive ITE?

- **Termination conditions:** when can we recognize answer and quit?
- **Splitting criterion:** which variable (x?) do you pick to split on?
- **Cofactoring:** how hard is it to get I_x, T_x, E_x, *etc.* from their BDDs?
- **Efficiency concerns:** how fast is this, how big is the recursion?
- **Reduction:** how do we guarantee the BDD we produce is reduced?

Solutions

- Turns these thing are actually all *interdependent*
- Let’s start walking thru these issues and see what we get…

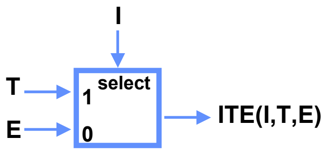
BDDs: Implementing ITE

General algorithm

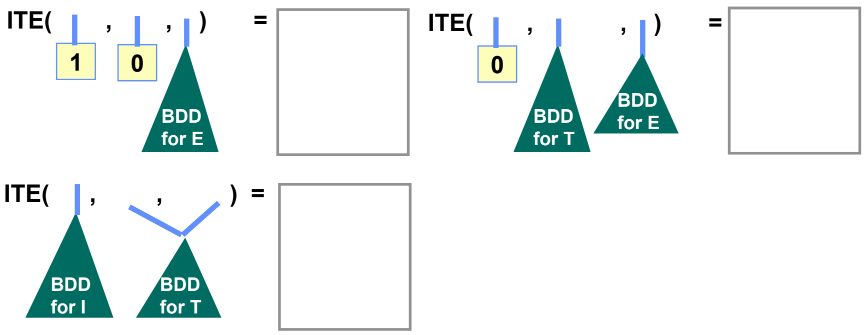
```
ITE( bdd I, bdd T, bdd E ) {
    if (terminal case) {
        return computed result ;
    }
    else {
        let x be splitting variable;
        PosFactor = ITE( $I_x$ , $T_x$ , $E_x$ ) ;
        NegFactor = ITE( $I_{x'}$ , $T_{x'}$ , $E_{x'}$ ) ;
        R = new node for var x ;
        R.loson = NegFactor ;
        R.hison = Posfactor ;
        Do reductions;
        return( R );
    }
}
```

BDDs: ITE Implementation

- ITE(I, T, E) terminal cases
  - If I = 1
  - if I = 0
  - If T = 1 && E = 0
  - If T = E

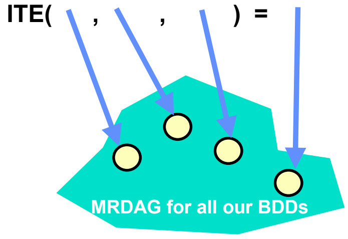


- Examples



BDDs: ITE Implementation

- Some subtlety about termination
  - A concern raised earlier was: how do we keep answer reduced?
  - At least for the termination conditions, easy in a multi-rooted DAG
  - When you terminate, you just return a pointer to something that *already exists* in the MRDAG
  - Since the MRDAG is a BDD, it's already reduced
    - ie, No extra work
- Note: so, when ITE is called and terminates, it really looks like this...



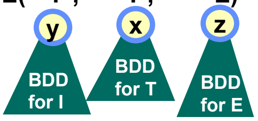
Each of the inputs – I, T, E are really represented by a ptr to some root node in this big MRDAG. Result ITE(I,T,E) ultimately ends up another node in here. For termination conditions, the result is always an existing node.

BDDs: ITE Implementation

Splitting variable selection

- There's actually only one right choice
- Split on smallest var among roots of I, T, E

ITE(I , T , E) => if x < y < z, pick x to split on



Why?

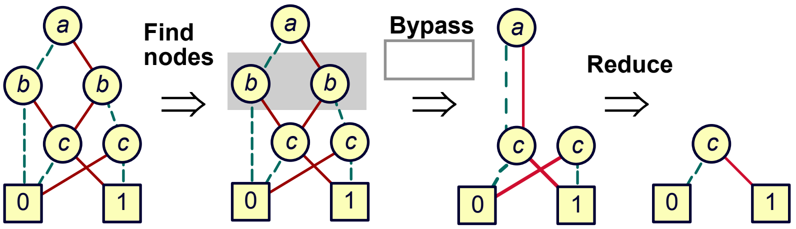
- Because it makes the cofactoring trivial to implement!
- Remember, we now have to compute I_x , T_x , E_x , $I_{x'}$, $T_{x'}$, $E_{x'}$

BDDs: ITE Implementation

RESTRICT(F,v,k)

- Remember, this is the operator that does the cofactor on BDDs
- In general, it's a bit complicated
- Trouble is when you cofactor wrt a var "down deep" inside the BDD
- Example: Restrict variable *b* to 1

```
Restrict (bdd F,  var x,  const k ):
   Bypass any nodes
    for variable x  /* set to const */
      – Choose Hi  child for k = 1
      – Choose Lo child for k = 0
   Reduce result
```


Find nodes ⇒

Bypass ⇒

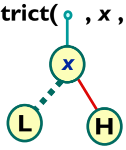
Reduce ⇒

BDDs: ITE Implementation

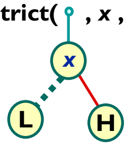
- ..but there's some useful special cases here
  - If we always pick splitting var as: smallest(roots of I, T, E)...
  - ...then we are always doing one of these 2 cases

Case 1: Restrict on root node variable

Restrict(, *x* , 1)



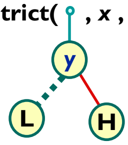
Restrict(, *x* , 0)



Case 2: Restrict on variable less than root node variable

– E.g., *x* < *y*

Restrict(, *x* , 1)



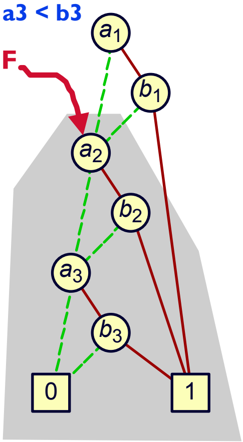
BDDs: ITE Implementation

- OK -- that *last* one was subtle...
  - Let's look at a concrete example to see "the idea" here.

var order is: a1 < b1 < a2 < b2 < a3 < b3

Function F

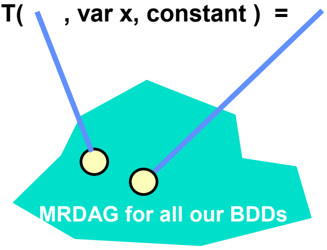
Restrict(F , b1, 1) == ?



BDDs: ITE Implementation

- Why that works
  - Always pick a variable at or above the root of I, T, E
  - Never then have to cofactor “down deep” in BDD
  - Result is always a pointer to some node that already exists in the multi-rooted DAG of our BDDs
  - So, again, the reduction step takes care of itself here
    - ie, the MRDAG is reduced at all times, we just return a pointer to a subDAG inside it, which is also reduced

RESTRICT(, var x, constant) =

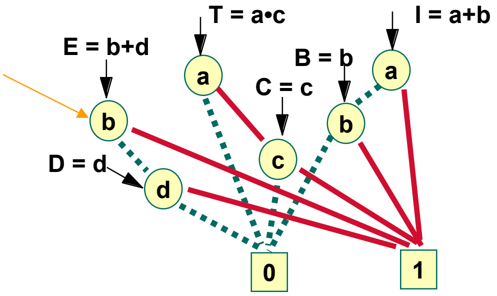


In a multirooted DAG, these special cases of RESTRICT will always return a pointer to some node that is already in DAG

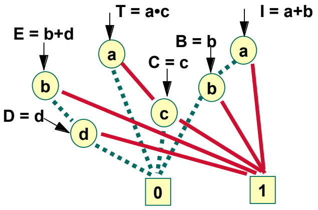
BDDs: ITE Example

- Let’s try a little ITE example
  - Capital letters = functions, small letters = variables
  - Var order is a < b < c < d
  - We want to compute ITE (I(a,b,c,d) , T(a,b,c,d) , E(a,b,c,d))
  - Assume it’s a MRDAG, we use function names to ID nodes...

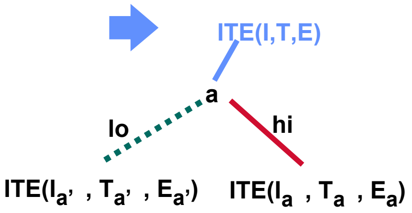
notation is “E” for this node, ie, we will label the Boolean function for each node

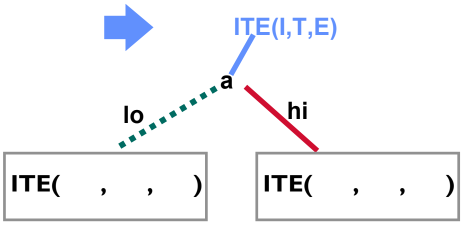


BDD: ITE Example

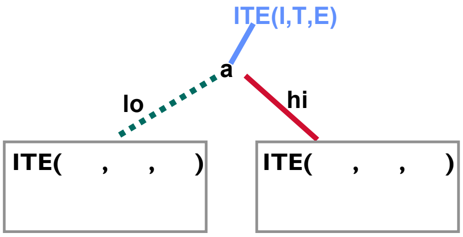


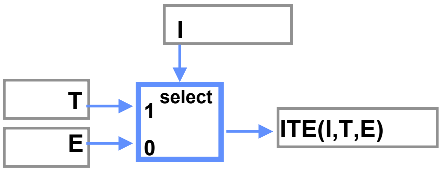
ITE(I, T, E) = ?
smallest var among roots is a

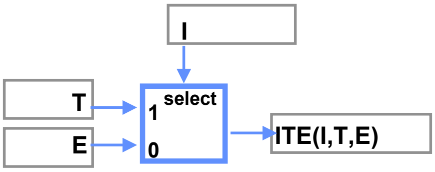

ITE(I,T,E)
a
lo
hi
ITE($I_{a'}$, $T_{a'}$, $E_{a'}$)
ITE(I_a , T_a , E_a)



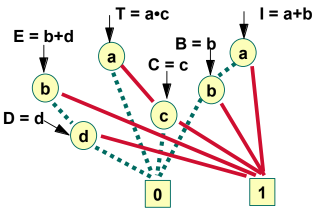
- Can we terminate these new ITEs right away?
  - …ie, will we recurse more, or do we know the answers now?
  - Let's remember the special cases for ITE termination

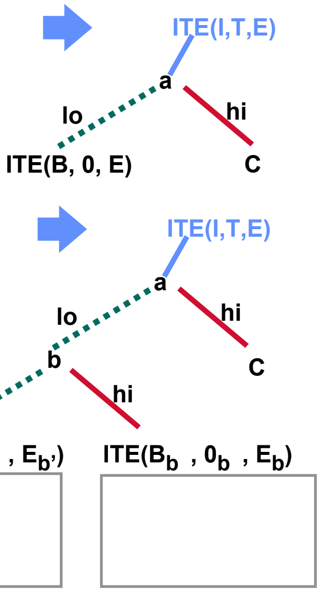




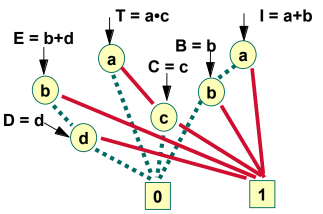


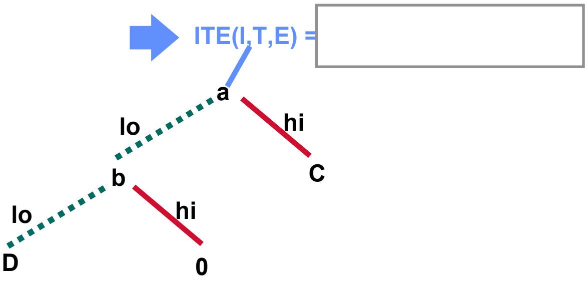
BDD: ITE Example




ITE(I,T,E)
a
lo
hi
ITE(B, 0, E)
C
ITE(I,T,E)
a
lo
hi
b
C
lo
hi
ITE(B$_{b'}$, 0$_{b'}$, E$_{b'}$)
ITE(B$_b$, 0$_b$, E$_b$)

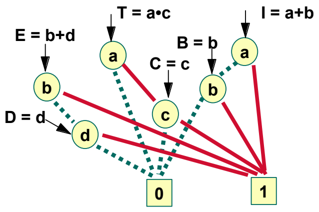
BDD: ITE Example

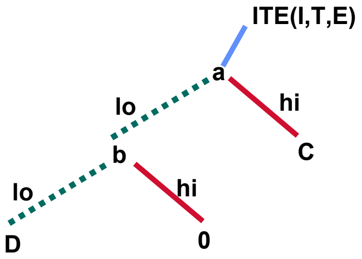




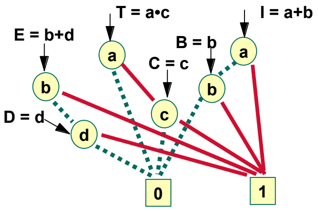
- And we are done!
  - Recursion tree naturally traces out the resulting BDD for ITE(I,T,E)
  - We set up termination conditions so that leaves are always nodes that already exist in the original multi-rooted DAG for
  - So, we can just "plug" our result back into our MRDAG...

BDD: ITE Example



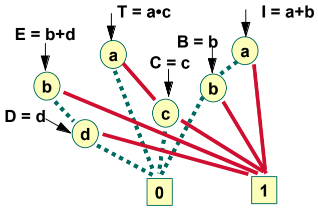


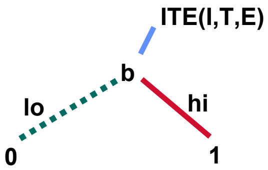
Actual resulting
MRDAG
that we get



BDD: ITE Implementation

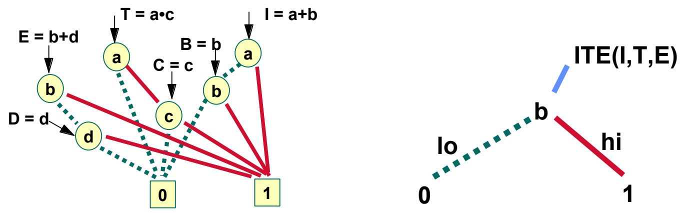
- Reduction, revisited
  - In our little example, the resulting function ITE was new, not already existing in the MRDAG, and all the leaf nodes were already there.
  - What happens if the resulting function is NOT new, if it already exists in the MRDAG?
- Example
  - Suppose that, for whatever reason, ITE returns the "B = b" function

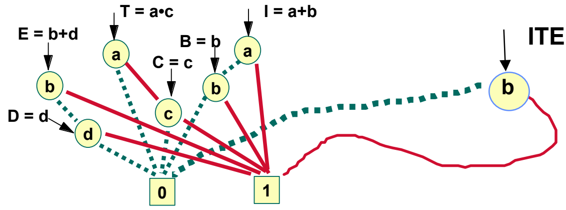




BDDs: ITE Implementation

- Following prior examples, we build MRDAG as…

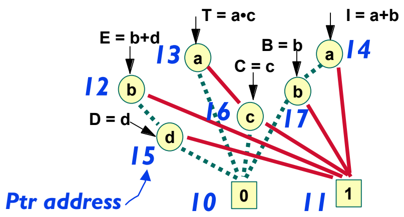




TROUBLE!!
It's not reduced anymore!!

BDDs: ITE Implementation

- Easy to create a new BDD that's *not reduced*
  - In our 'B=b' example, we made a redundant 'b' var node
  - How to fix?
- Clever trick: The "Unique" Table
  - New notation: make key for each node as (var, lo, hi)
  - Make a hash table (called the Unique table) that maps (var, lo, hi) for each node into actual pointer address of the node in the MRDAG



Unique Table

Hash Index	Address
(a,17,11)	14

BDDs: ITE Implementation

Trick

- Never, ever just create a node, like a ‘B=b’ node, without checking to see if the node you want to create already exists in the Unique table
- This is the “big idea” for how we avoid redundancy

Need a new function, called *FindOrCreateNode*

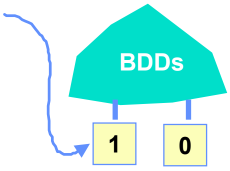
- FindOrCreateNode checks to see if the node you want to make already exists in the MRDAG.
- If so, it just returns a pointer to that node, instead of making a new node
- FindOrCreateNode also can check to make sure you don’t do something stupid to make an unreduced DAG
- Helpful to see a picture of what FindOrCreateNode does...

BDDs: ITE Reduction

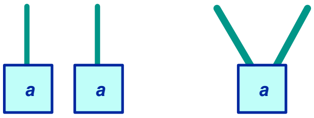
Case 1: Trying to make a constant node

- If you invoke FindOrCreate(constant, null, null) and the MRDAG already has that constant, you return a pointer to it

Ex: FindOrCreateNode(1, null, null) =



- Note: this handles the first reduction rule we gave...

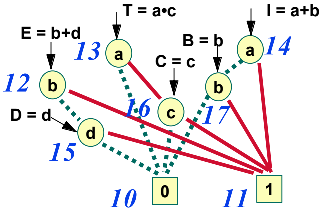


BDDs: ITE Reduction

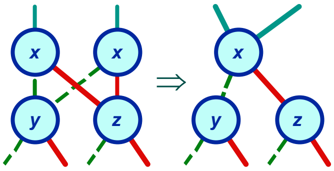
Case 2: Trying to duplicate an existing var node

- If you invoke FindOrCreate(var, loson, hison) and the MRDAG already has this node, you just return a pointer to it

Ex: FindOrCreateNode(a, 10, 16) = □



- Note: this handles the second reduction rule we gave...

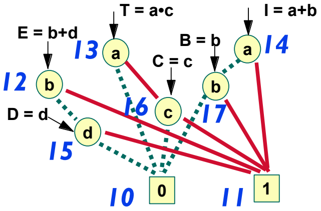


BDDs: ITE Reduction

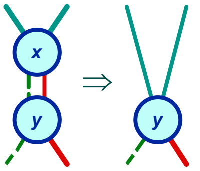
Case 3: Trying to make an unnecessary test node

- If you invoke FindOrCreate(var, loson, hison) and loson = = hison, then you just return loson, and don't even try to look in DAG

Ex: FindOrCreateNode(b, 15, 15) = □



- Note: this handles the last reduction rule we gave...

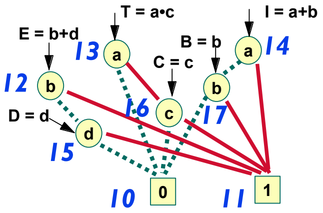


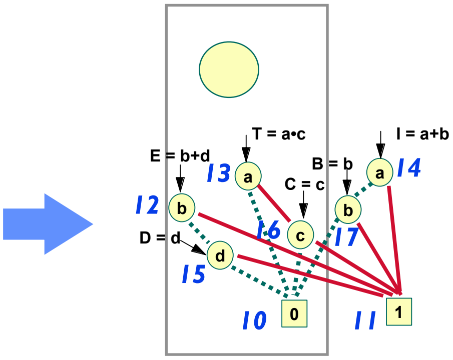
BDDs: ITE Reduction

Case 4: None of the above

- Well, you can't find it in the existing **MRDAG**, and it's not stupid to create it, so go ahead and **create** the node, and then remember to **insert** it in the unique table

Ex: FindOrCreateNode(b, 15, 16) = ?





Unique Table

Hash index	Address
•	
•	
•	
(b, 15, 16)	20

ITE: FindOrCreateNode

Algorithm looks like this

```
FindOrCreateNode(var v,   bdd loson,    bdd hison) {
   if (v is actually a constant ) {
      if( this constant NOT already in Unique table )
         install this constant in UNIQUE table;
      return ( pointer to constant );
   }
   else if (loson == hison)
      return (loson) ;
   else if ( (v, loson, hison) already in Unique table )
      return ( pointer to (v, loson, hison) from Unique table);
   else {
      create new node = (v, loson, hison);
      install this node in Unique table;
      return (pointer to (v, loson, hison) );
   }
}
```

BDDs: Reduction

- What's really going on here?
  - Instead of building a nonreduced BDD, then going back to reduce it…
  - …instead, we check each node as we try to create it during recursive divide&conquer…
  - …if it would create a redundant node, we don't allow it to happen.
  - Instead, we return an existing node of the BDD whenever we can.

- This is actually how ALL reductions get done on BDDs
  - All algorithms that manipulate or create BDDs call FindOrCreateNode, to ensure things always stay reduced
  - *Never, ever create an unreduced DAG*

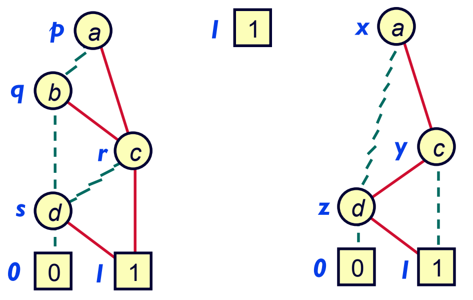
ITE: Implementation

- One more essential trick for efficiency
  - Turns out that on bigger problems, it's very easy for the recursion to try to recompute the same intermediate result – ITE(•, •, •) called on intermediate functions – many times
  - In fact, can be exponential number of calls to ITE in the recursion, but not all of them are unique computations
  - Turns out there is a really easy fix…
  - But let's convince ourselves that this is a real problem, first…

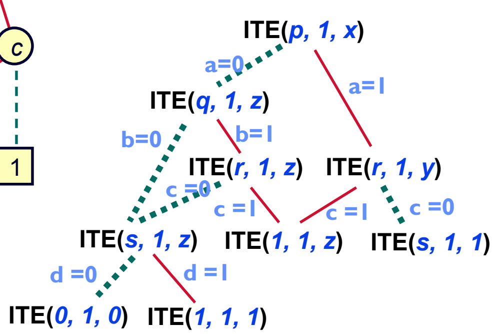
ITE: Recursive Calls

- Compute ITE(I, T, E) for BDDs below (alphabet var order)

Argument I Argument T Argument E



ITE Recursive Calls


ITE(*p, 1, x*)

a=0 a=1

ITE(*q, 1, z*)

b=0 b=1

ITE(*r, 1, z*) ITE(*r, 1, y*)

c =0 c =1 c =1 c =0

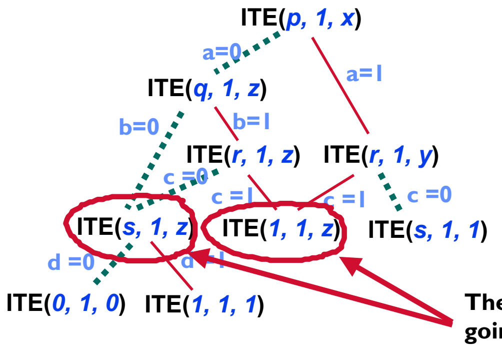
ITE(*s, 1, z*) ITE(*1, 1, z*) ITE(*s, 1, 1*)

d =0 d =1

ITE(*0, 1, 0*) ITE(*1, 1, 1*)

BDDs: ITE Recursive Calls

- Recursive call tree tells us a lot…


ITE(*p, 1, x*)

a=0 a=1

ITE(*q, 1, z*)

b=0 b=1

ITE(*r, 1, z*) ITE(*r, 1, y*)

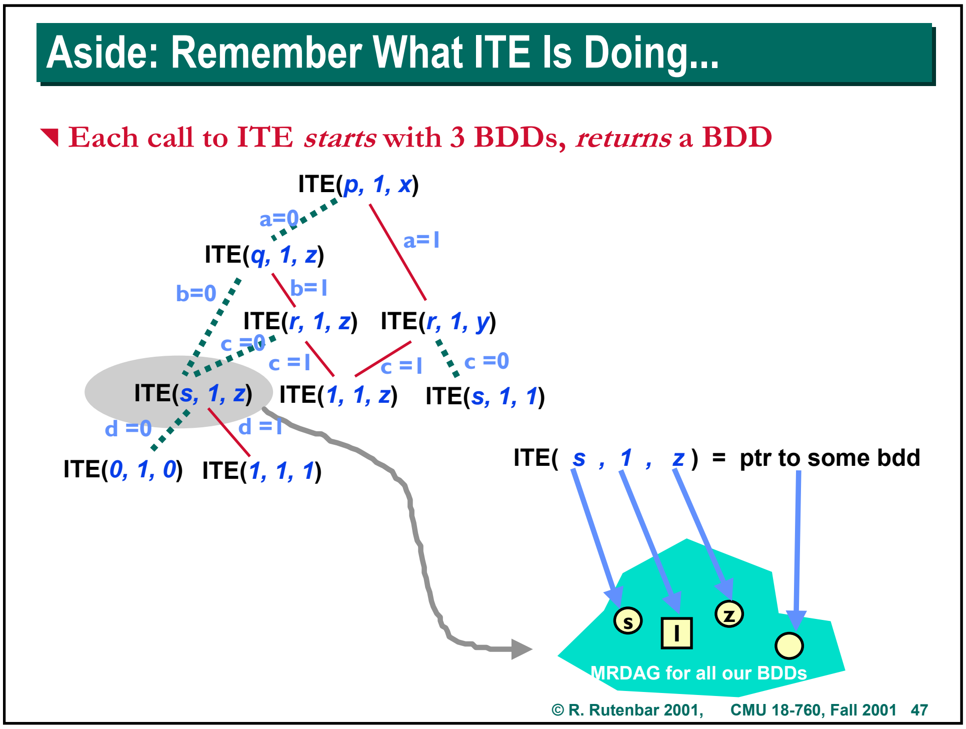
c =0 c =1 c =1 c =0

ITE(*s, 1, z*) ITE(*1, 1, z*) ITE(*s, 1, 1*)

d =0 d =1

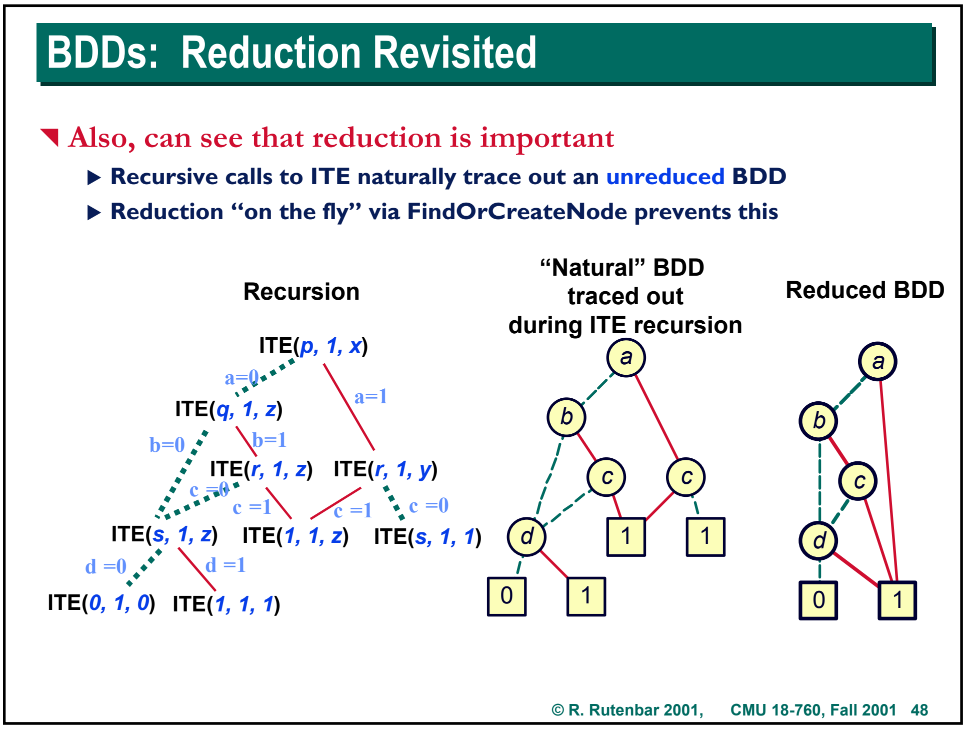
ITE(*0, 1, 0*) ITE(*1, 1, 1*)

These have 2 arrows going INTO them, so these ITE results will get recomputed twice during recursion



Aside: Remember What ITE Is Doing...

- Each call to ITE *starts* with 3 BDDs, *returns* a BDD



BDDs: Reduction Revisited

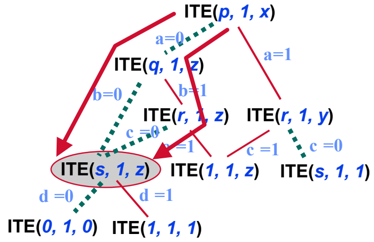
- Also, can see that reduction is important
  - Recursive calls to ITE naturally trace out an unreduced BDD
  - Reduction "on the fly" via FindOrCreateNode prevents this

BDDs: ITE Operation Table

How to prevent *recomputation* during ITE recursion?

- Every time you return from ITE, you store your computed answer in a table, called the **Operation Table**. Key is (I,T,E), value is address of computed node in BDD returned
- Every time you enter ITE, ***you look in the table first*** to see if the answer has already been computed earlier in recursion
- Nice, simple solution, trades some memory (table) for time

Recursion



Operation Table

Hash Value	Address
(p,1,r)	bddNodeI
(q,1,d)	bddNodeJ
(s,1,z)	bddNodeK
•	
•	
•	
•	

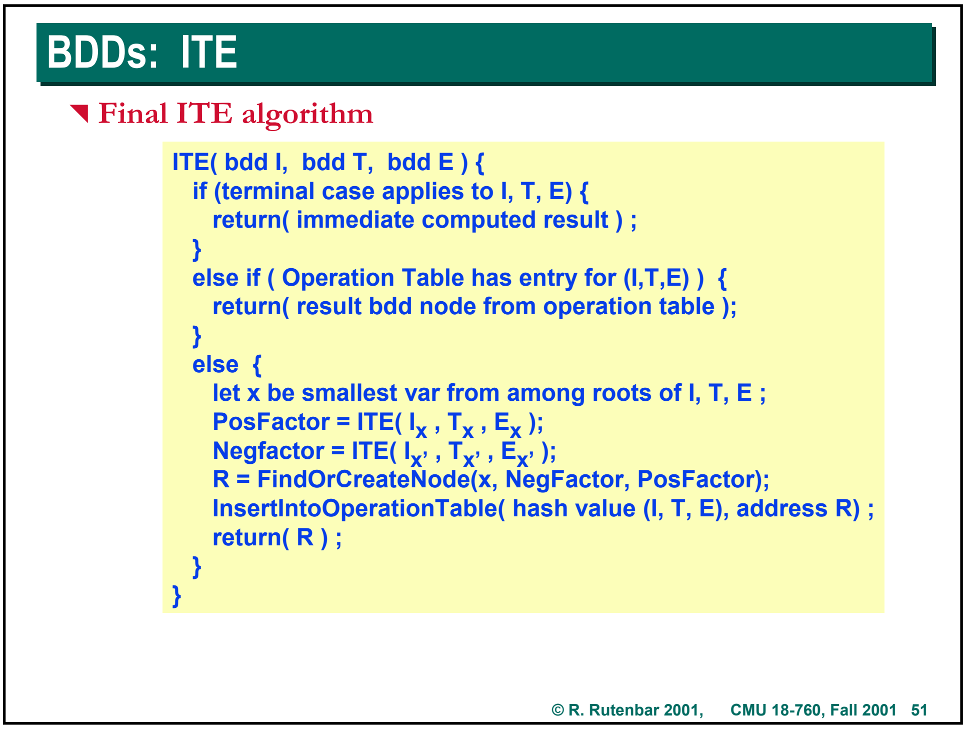
BDDs: ITE Complexity

Efficiency

- Without Operation Table
  - Exponential Complexity
  - Effectively expand out decision trees
- With Operation Table
  - Worst case = product of graph sizes for *I*, *T*, and *E*
  - At worst, will fill table with all possible keys

Interaction with multi-root DAG assumption

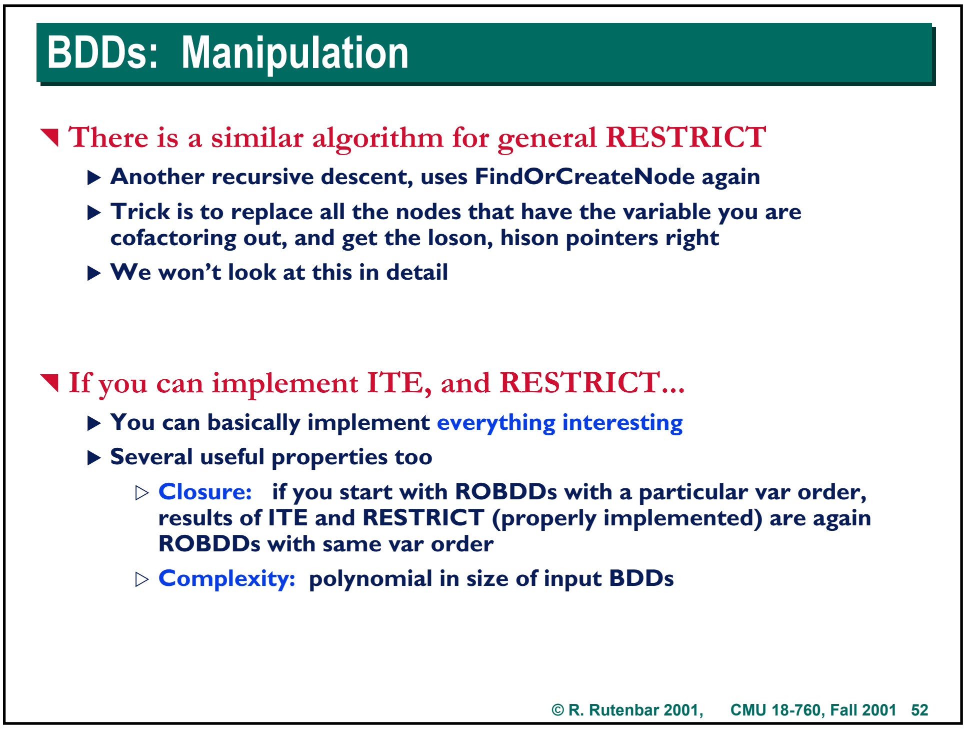
- If all BDDs use shared nodes...
  - Maintain **global** Operation Table, across **ALL** BDDs
  - More possibilities for quick termination



BDDs: ITE

- Final ITE algorithm

```
ITE( bdd I,  bdd T,  bdd E ) {
  if (terminal case applies to I, T, E) {
    return( immediate computed result ) ;
  }
  else if ( Operation Table has entry for (I,T,E) )  {
    return( result bdd node from operation table );
  }
  else  {
    let x be smallest var from among roots of I, T, E ;
    PosFactor = ITE( $I_x$ , $T_x$ , $E_x$ );
    Negfactor = ITE( $I_{x'}$ , $T_{x'}$ , $E_{x'}$ );
    R = FindOrCreateNode(x, NegFactor, PosFactor);
    InsertIntoOperationTable( hash value (I, T, E), address R) ;
    return( R ) ;
  }
}
```



BDDs: Manipulation

- There is a similar algorithm for general RESTRICT
  - Another recursive descent, uses FindOrCreateNode again
  - Trick is to replace all the nodes that have the variable you are cofactoring out, and get the loson, hison pointers right
  - We won’t look at this in detail
- If you can implement ITE, and RESTRICT…
  - You can basically implement everything interesting
  - Several useful properties too
    - Closure: if you start with ROBDDs with a particular var order, results of ITE and RESTRICT (properly implemented) are again ROBDDs with same var order
    - Complexity: polynomial in size of input BDDs

BDDs: Manipulation

Summary

- Implement 2 core operators on BDD data structures
  - RESTRICT(bdd F, var x, constant k) = Shannon cofactor
  - ITE(bdd I, bdd T, bdd E) = IF-THEN-ELSE operator
  - Each of these takes BDDs as inputs, returns a new BDD
- From ITE & RESTRICT,you can implement lots of stuff
  - Cofactoring, quantification, derivatives
  - All the basic gate types: NOT, AND, OR, EXOR, etc
  - Other more exotic stuff (see homework)
- Given an efficient BDD software package, can do lots of nice practical engineering applications
  - Satisfiability: trace BDD from root to ‘1’ node to find inputs that make the function == 1
  - Equivalence: if 2 functions are same over ALL inputs, their BDDs end up itentical, ie, 2 pointers to same node in the MRDAG