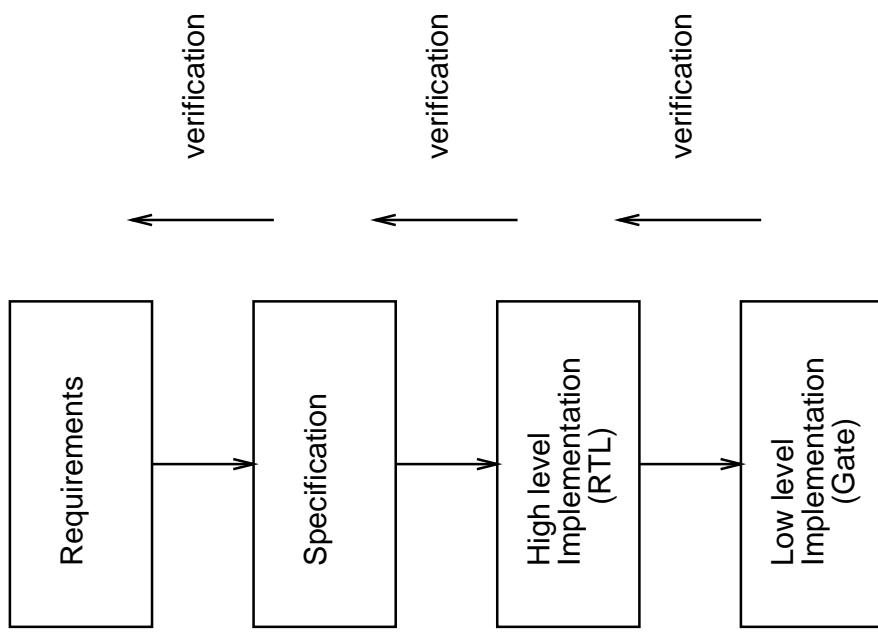


# ECE 741/841

1 October 2002

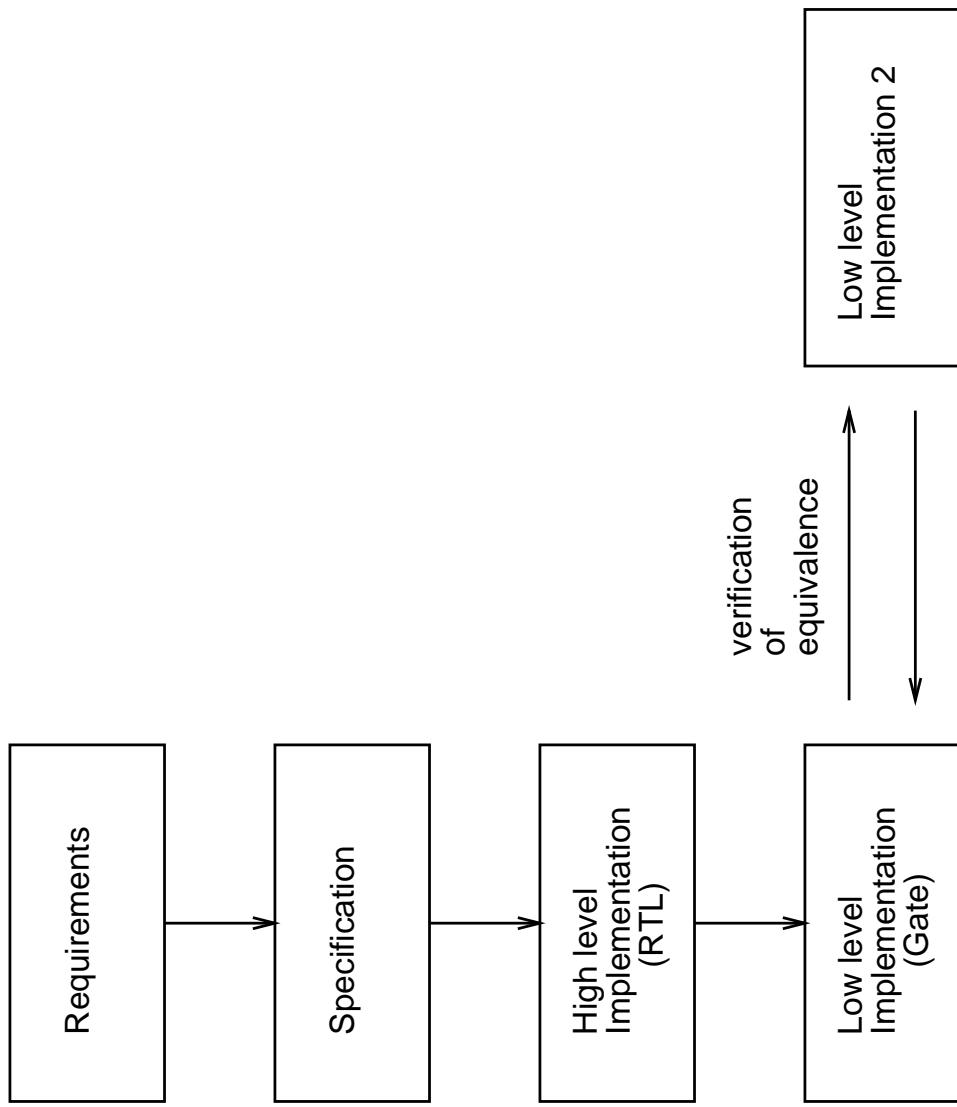
## Verification

Verification is usually performed **between levels of abstraction**.



# Verification

Verification can also be used to show that two models are equivalent or that other properties hold.



## Verification Example 1

In the first verification example, we are going to design a digital adder. We will have:

- A specification.
- A high level implementation.
- A gate level implementation.

## Verification Example 1, cont.

After we have these definitions, we will attempt a verification by proving the high level implementation from the gate level implementation and the specification from the high level implementation.

gate level implementation  $\vdash$  high level implementation

high level implementation  $\vdash$  specification

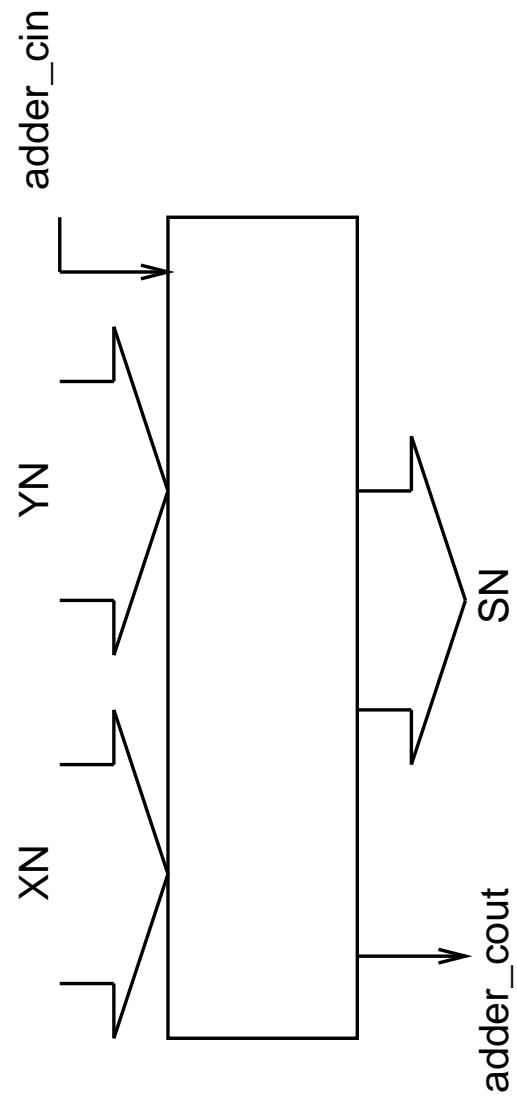
or equivalently

$\vdash$  gate level implementation  $\Rightarrow$  high level implementation

$\vdash$  high level implementation  $\Rightarrow$  specification

## Specification

The adder is to take two numbers  $X_N, Y_N \in \{0..2^N - 1\}$  and a carry in  $\text{adder\_cin} \in \{0, 1\}$ . It is to add  $X_N, Y_N$  and  $\text{adder\_cin}$  modulo  $2^N$  and output the result on  $S_N \in \{0..2^N - 1\}$ . If  $X_N + Y_N + \text{cin}$  is greater or equal to  $2^N$ , then the carry out  $\text{adder\_cout}$  is set to 1, else it is set to 0.



## Specification in PVS

```
adder_spec : theory
begin

N : nat

importing digits

XN,YN : VAR below[exp2(N)] % {n:nat | n < exp2(N)}
adder_cin : VAR below[2] % {n:nat | n < 2}

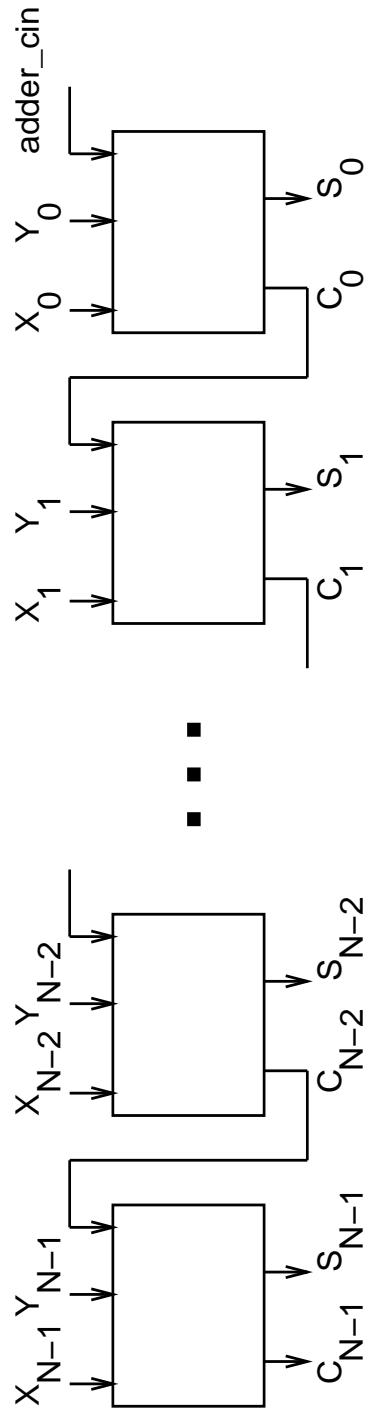
adder_S(XN,YN,adder_cin) : below[exp2(N)] =
if XN+YN+adder_cin >= exp2(N) then XN+YN+adder_cin - exp2(N)
else XN+YN+adder_cin
endif
```

## Specification in PVS, cont.

```
adder_cout(XN,YN,adder_cin) : below[2] =  
if XN+YN+adder_cin >= exp2(N) then 1  
else 0  
endif  
  
end adder_spec
```

## High Level Implementation

Our implementation will be a ripple carry adder.



## Definition of Bit and Bit Vector

Before we define the high level implementation, we need to define what a bit vector is.

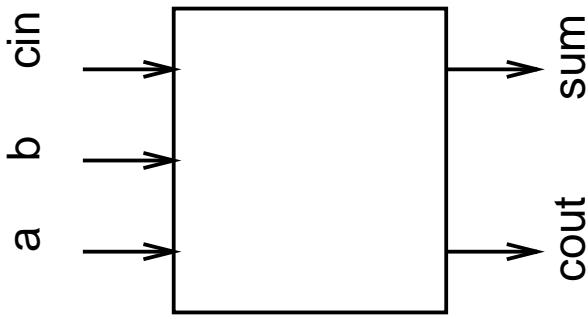
A bit is defined as 0 or 1

A bit vector is defined as a function that takes as argument a natural number  $i$  and returns the  $i^{th}$  element of the vector.

```
bit : type = below[2] % {0,1}
```

```
bvec : type = [below[N] -> bit]
```

## Specification of One Bit Adder

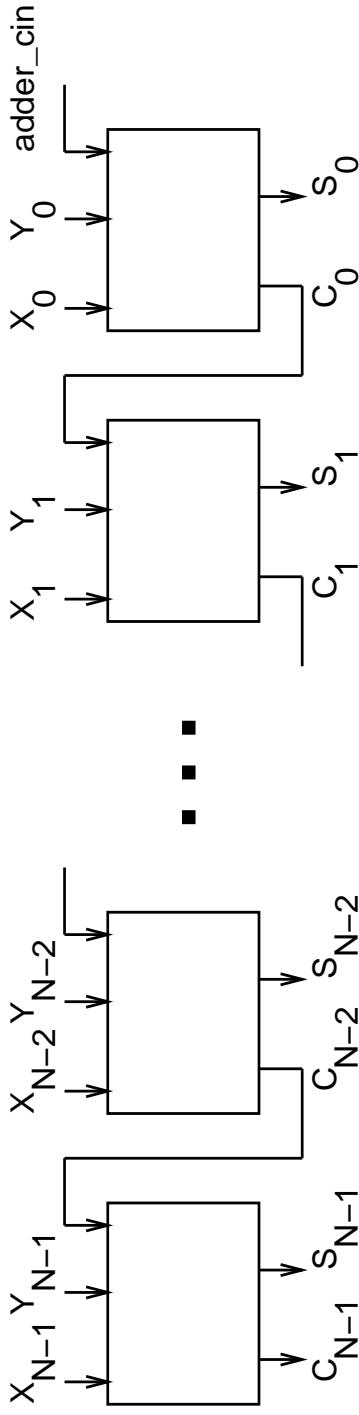


a,b,cin : VAR bit

```
OBA_sum(a,b,cin) : bit =  
if a+b+cin = 0 or a+b+cin = 2 then 0 else 1 endif
```

```
OBA_cout(a,b,cin) : bit = if a+b+cin >= 2 then 1 else 0 endif
```

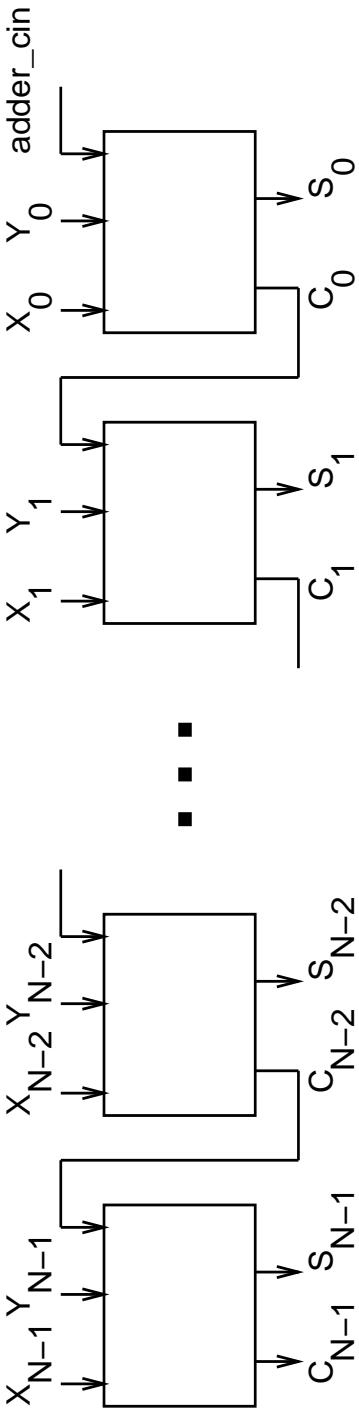
## High Level Implementation



```
n : VAR below[N]
X,Y : VAR bvec
adder_cin : VAR bit
```

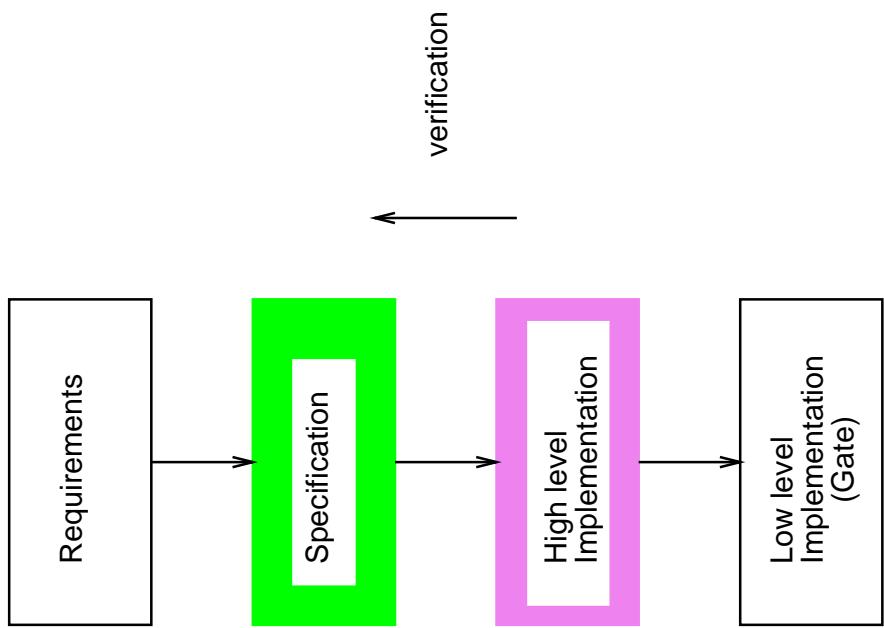
```
n_cout(n,X,Y,adder_cin) : recursive bit =
if n=0 then OBA_cout(X(0),Y(0),adder_cin)
else OBA_cout(X(n),Y(n),n_cout(n-1,X,Y,adder_cin))
endif
measure n
```

## High Level Implementation, cont.



```
adder_sum(X,Y,adder_cin) : bvec =  
LAMBDA n :  
  if n=0 then OBA_sum(X(0),Y(0),adder_cin)  
  else  OBA_sum(X(n),Y(n),n_cout(n-1,X,Y,adder_cin))  
endif
```

## Verification



## Correctness of One Bit Adder

If the specification of the one bit adder is correct, then the following property must hold.

OBA\_correct : LEMMA

$$a+b+c_{in} = 2*OBA\_cout(a,b,c_{in}) + OBA\_sum(a,b,c_{in})$$

## Correctness of High Level Implementation

A notion of correctness must be established for the verification. A first idea is to show that given the same inputs, the specification and the high level implementation produce the same results.

```
adder_correct : THEOREM  
adder_sum(X,Y,adder_cin) = adder_S(XN,YN,adder_cin)
```

```
cout_correct : THEOREM  
n_cout(N-1,X,Y,adder_cin) = adder_cout(XN,YN,adder_cin)
```

This notion of correctness has a big problem. We are talking here about two different levels of abstraction.

## Interpretation (Semantics) of Bit Vectors

Note that a bit vector can represent a floating point number or a integer in two's complement encoding, or have other meanings.

For our example, the bit vectors represent natural numbers.

The formal interpretation is a function that takes a bit vector and returns a natural number:

$$bv2nat(bv) = bv(N-1)2^{N-1} + bv(N-2)2^{N-2} + \dots + bv(1)2^1 + bv(0)$$

$$bv2nat(bv) = \sum_{i=0}^{N-1} bv(i)2^i$$

## Bit Vector Function Definition

We now create a function in the PVS language to represent this summation.

```
bv : VAR bvec  
n : VAR below[N]
```

```
bv2nat(bv) : below[exp2(N)] = bv2nat_rec(N-1,bv)  
  
bv2nat_rec(n, bv) : recursive nat =  
  if n=0 then bv(0)  
  else bv(n)*exp2(n) + bv2nat_rec(n-1,bv)  
  endif  
measure n
```

## Correctness

Using the interpretation of bit vectors, we can now compare the specification with the high level implementation.

```
adder_correct : THEOREM
forall X,Y,adder_cin :
bv2nat (adder_sum(X,Y,adder_cin)) =
adder_S(bv2nat(X),bv2nat(Y),adder_cin)
```

```
cout_correct : THEOREM
n_cout(N-1,X,Y,adder_cin) =
adder_out(bv2nat(X),bv2nat(Y),adder_cin)
```

## Auxiliary Theorem

```
adder_correct_rec : LEMMA
FORALL (n:nat) :
bv2nat_rec(n,adder_sum(X,Y,adder_cin)) =
adder_S(bv2nat_rec(n,X),bv2nat_rec(n,Y),adder_cin)
```