

# Agenda

- Settle in
- SPV overview
- Setting up a project
- First steps – driving signals
- Using the debugger
- More driving (and collecting) signals
- SpvBitVec and friends

# Agenda (cont)

- Generation
- Coverage
- Project Exercise
- Optional
- Summary

# What is SPV Test Bench Studio?

- Verification library
  - Simulation interaction (connectivity)
    - Write/Read signals
    - React to simulation events (processes)
  - Generation
  - Coverage
  - Basic helper classes (bit vector and friends)
  - Companion libraries
- Simulator Add-on (PLI, FLI, VHPI, etc.)
- Stand-alone SPVSim (Modeling)

# What is SPV? (cont)

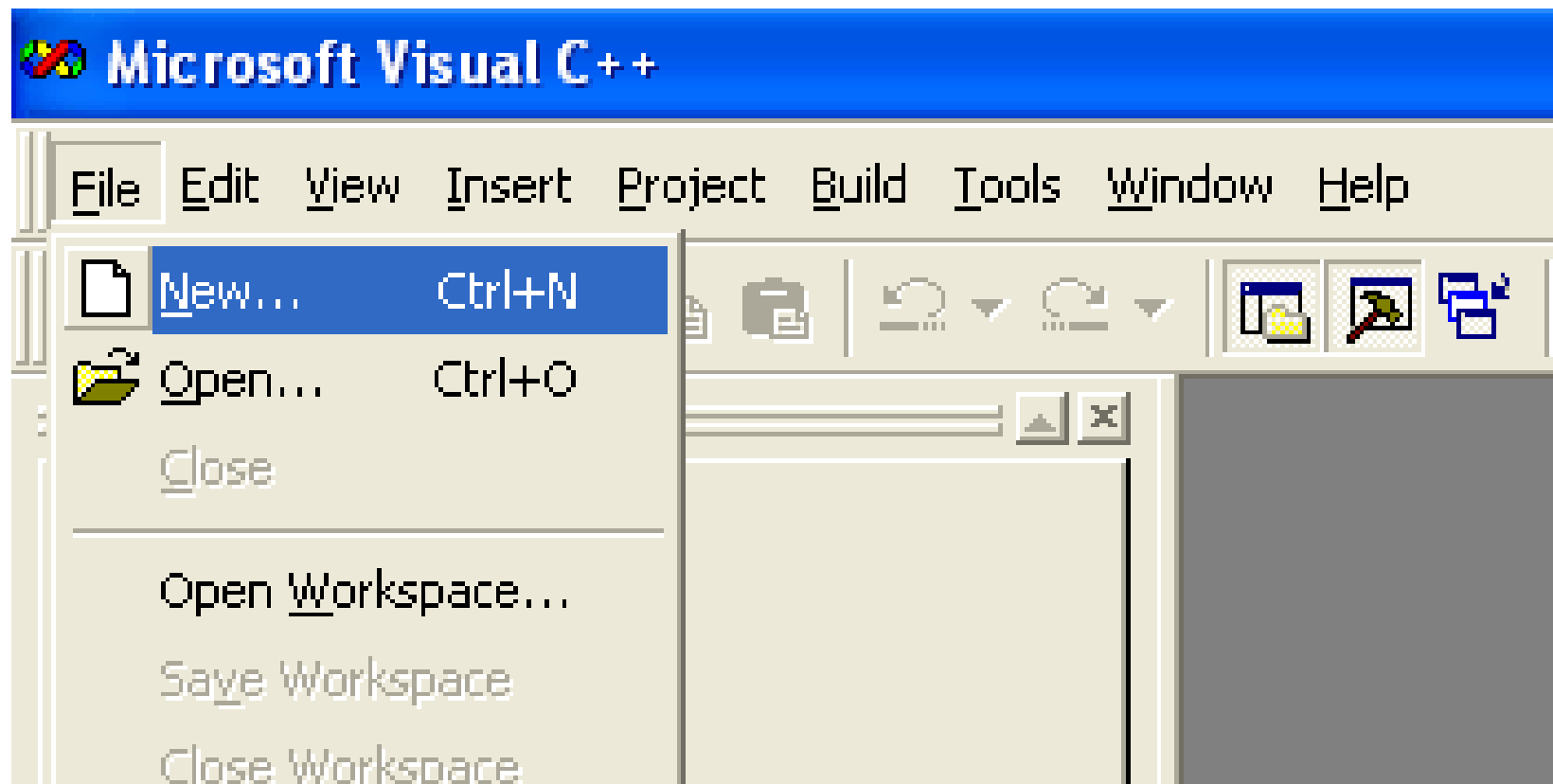
- Cross platform
  - Windows
  - Linux
  - Others
- Compilers
  - VS 6 (Windows) (now adding support for later VS)
  - GCC 2.96 (Linux/Unix)
  - GCC 3.2 and higher
    - Linux/Unix
    - Windows (MinGW)

# Setting Up

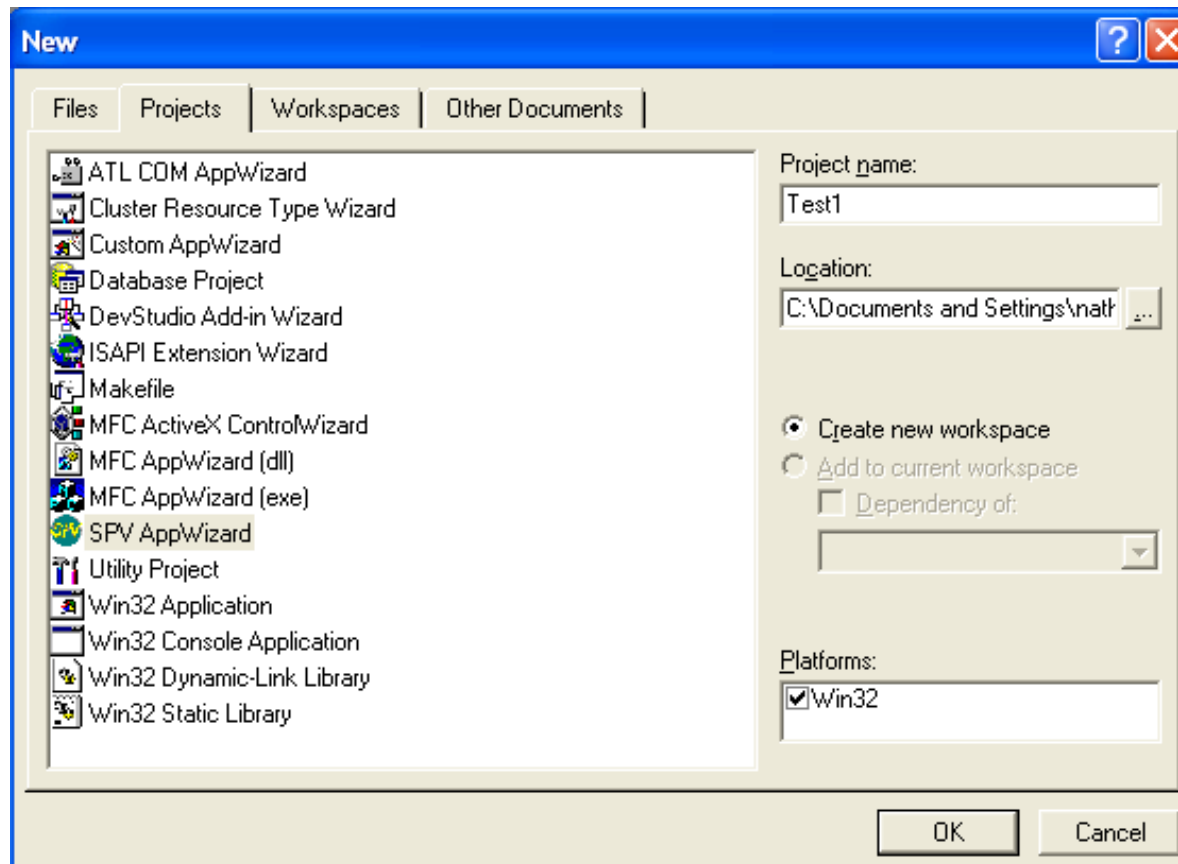
## **Requirements:**

- ✓ Simulator (e.g. Active HDL)
- ✓ Visual Studio 6 with SP 6
- ✓ SPV Test Bench Studio

# SPV AppWizard – Step 1



# SPV AppWizard – Step 2



# SPV AppWizard – Step 3

SPV AppWizard - Step 1 of 2

HDL Language

- Verilog
- VHDL

Simulator

- Active-HDL
- ModelSim

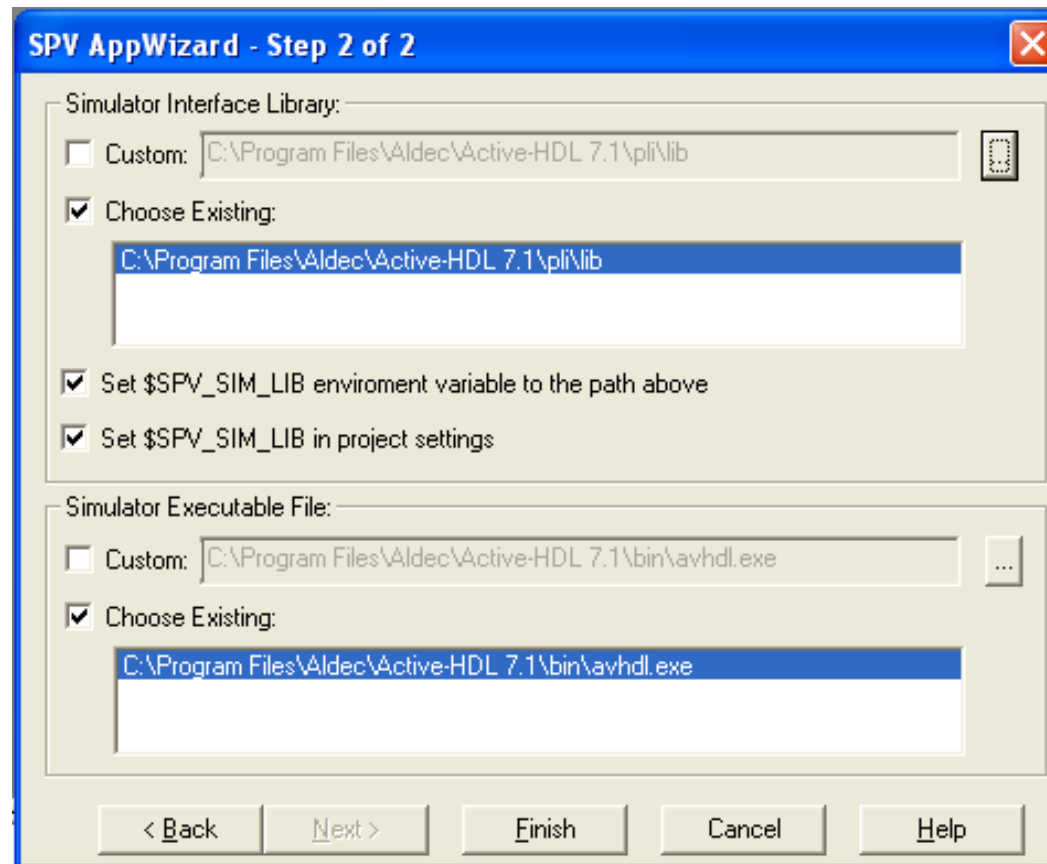
Project Type

- Empty
- Driver - Collector ?
- Skeleton
- Example

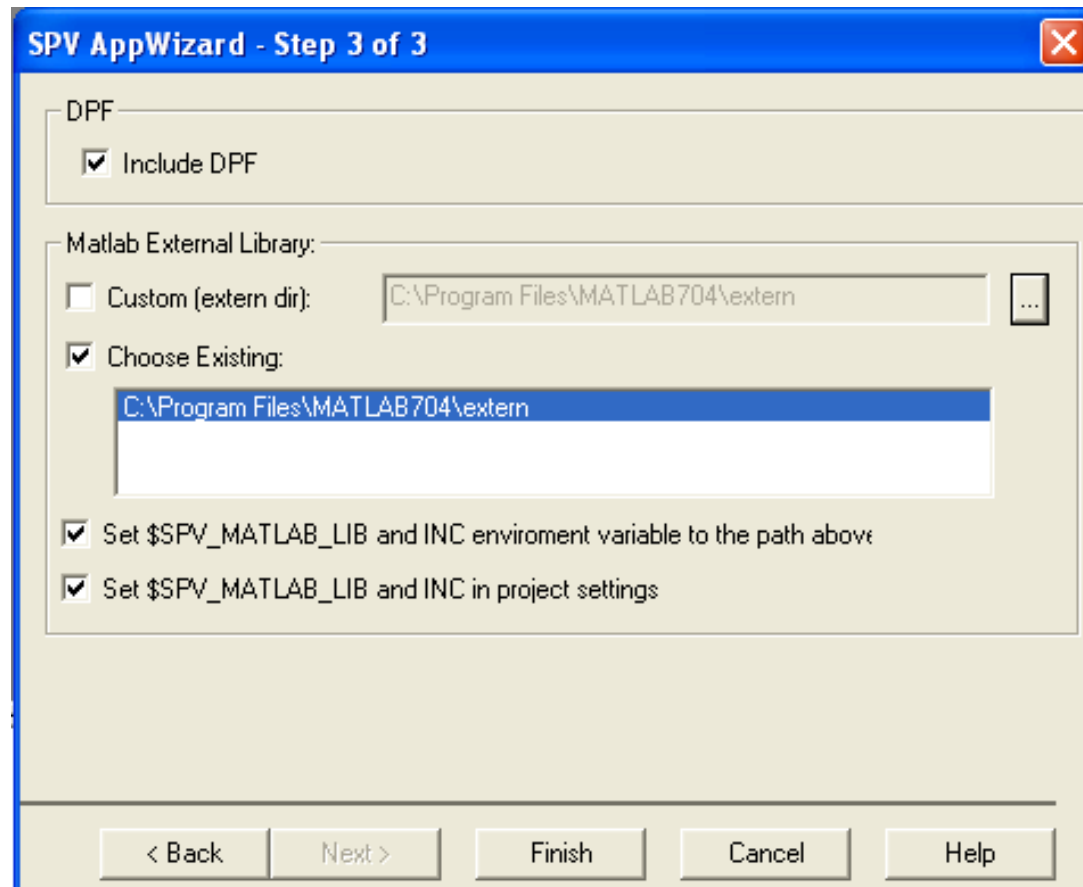
< Back   Next >   Finish   Cancel   Help



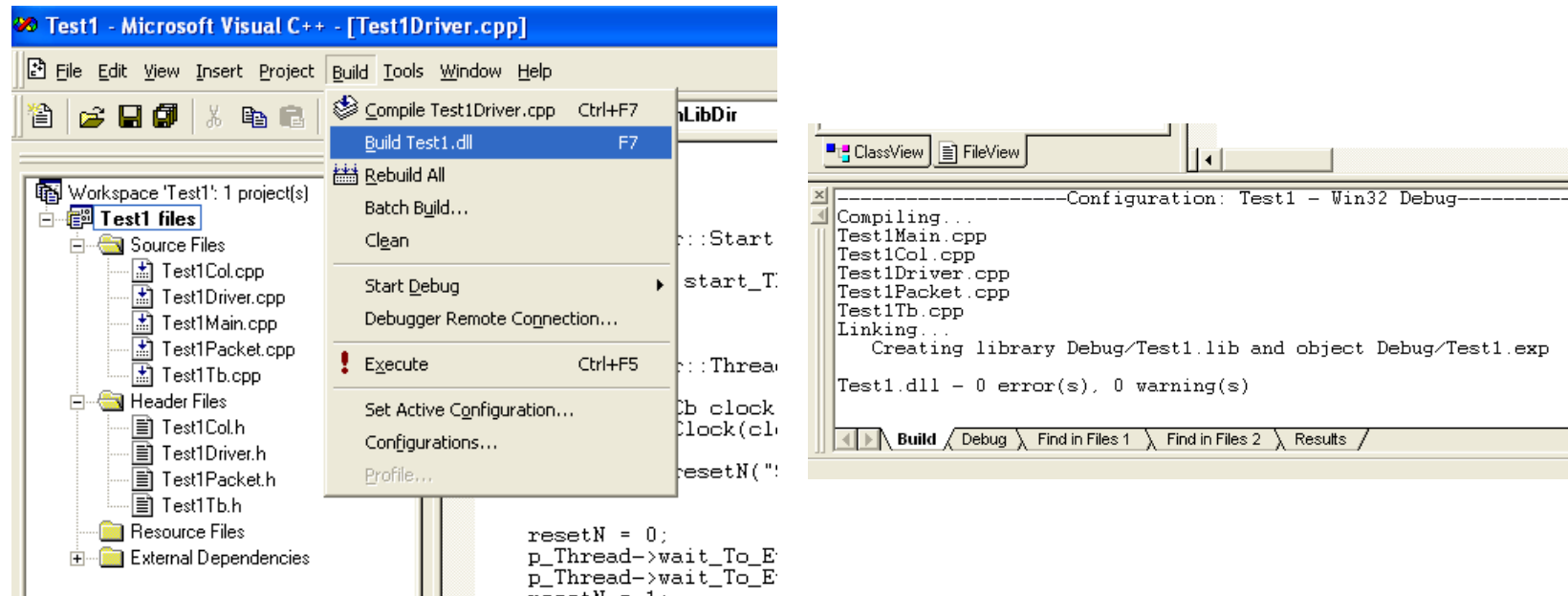
# SPV AppWizard – Step 4



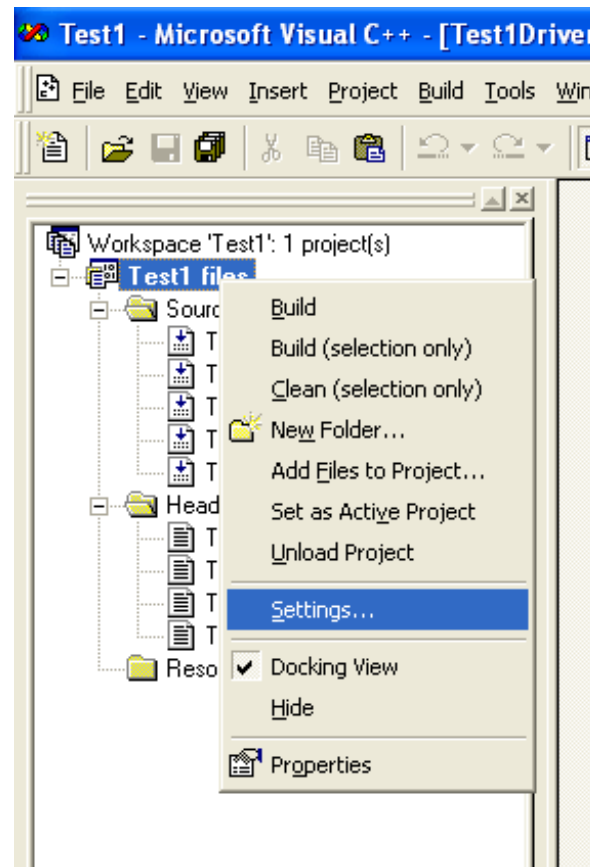
# SPV AppWizard – Step 5



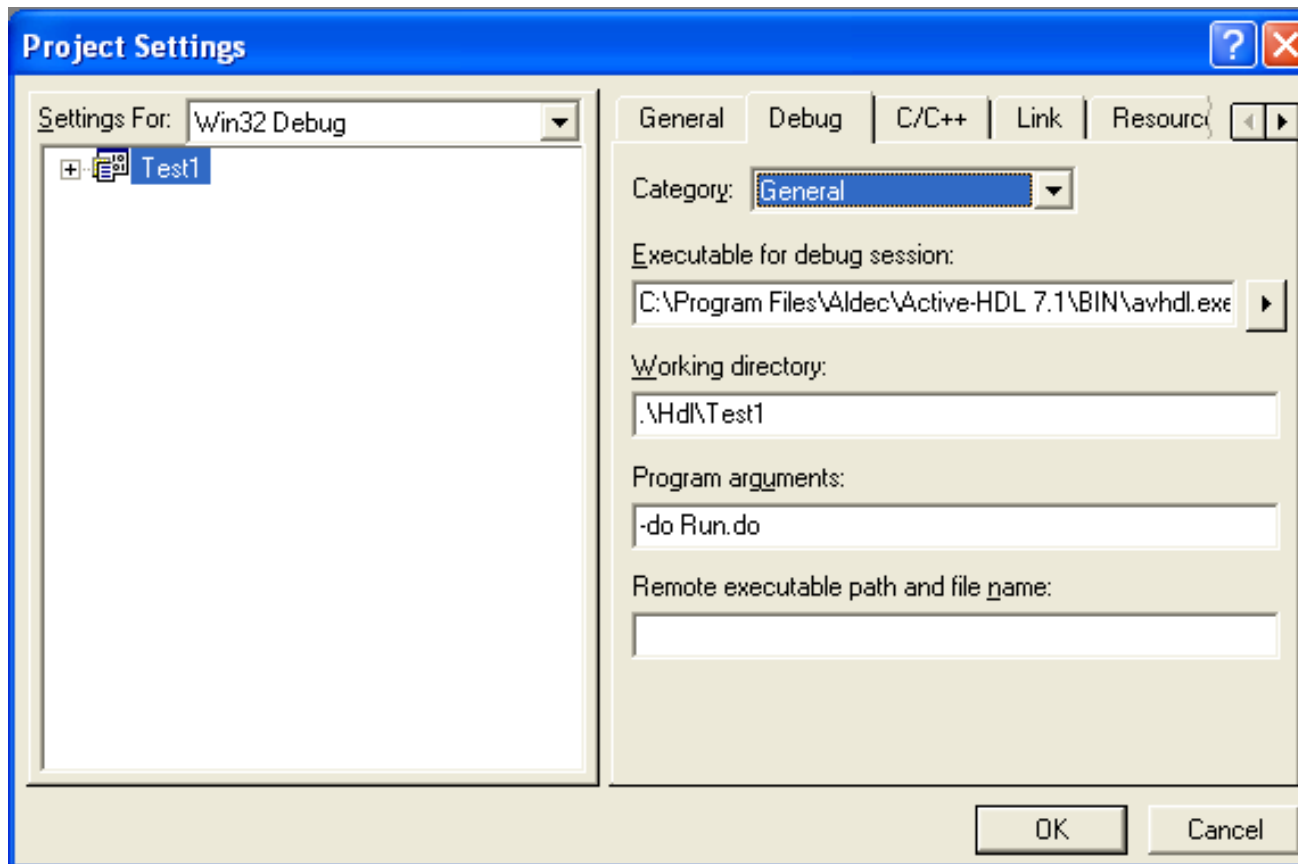
# Build



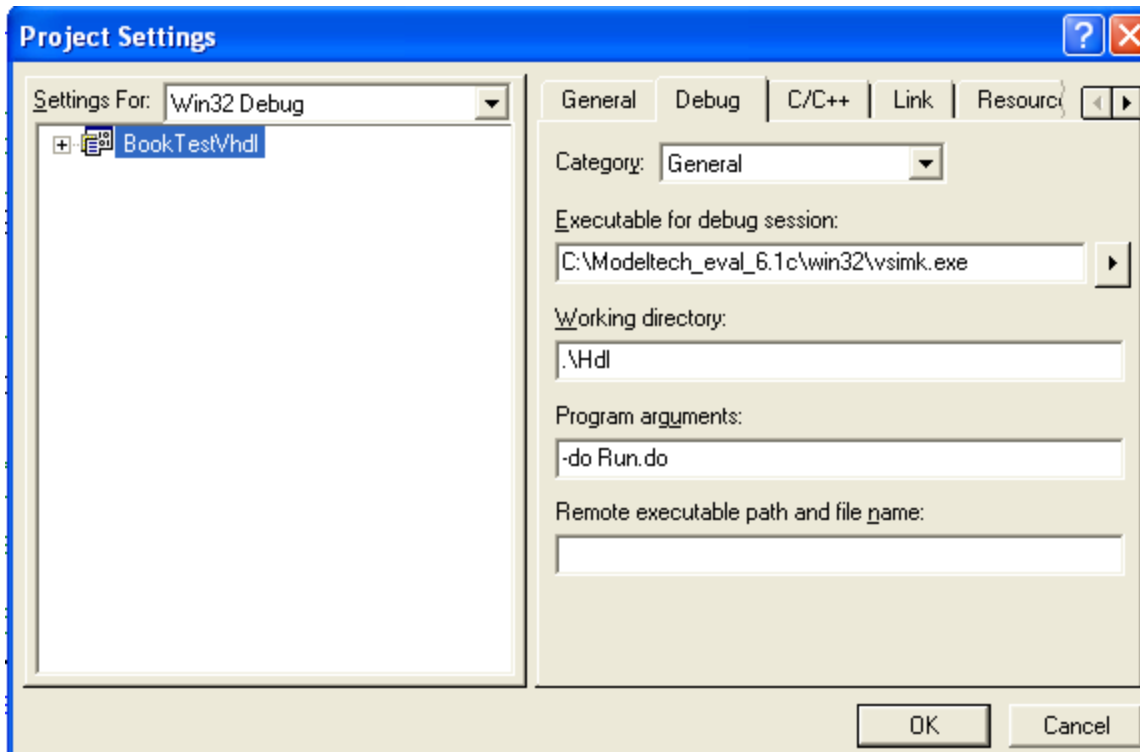
# Debugger Setup – Step 1



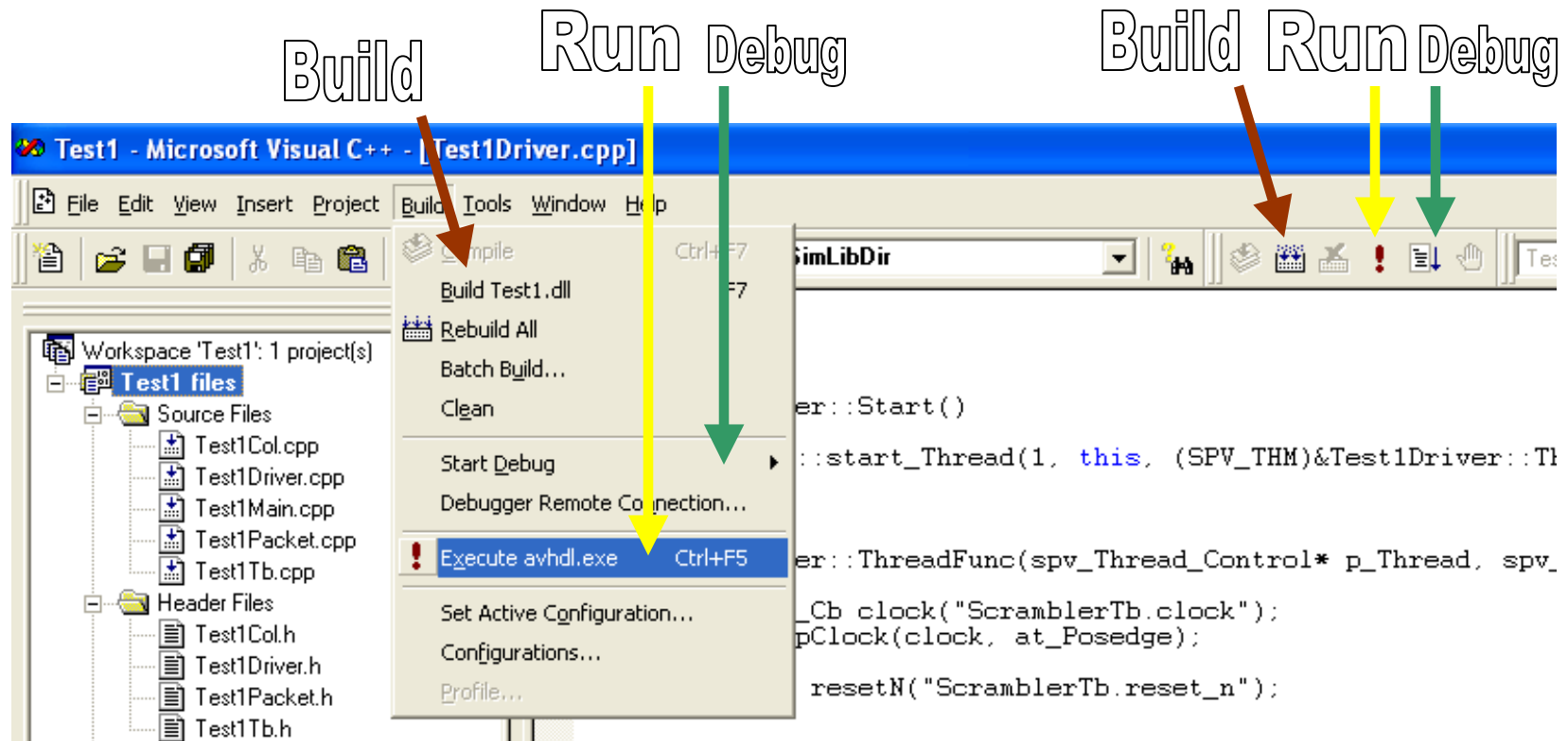
# Debugger Setup – Step 2 (ActiveHDL)



# Debugger Setup – Step 2 (ModelSim)

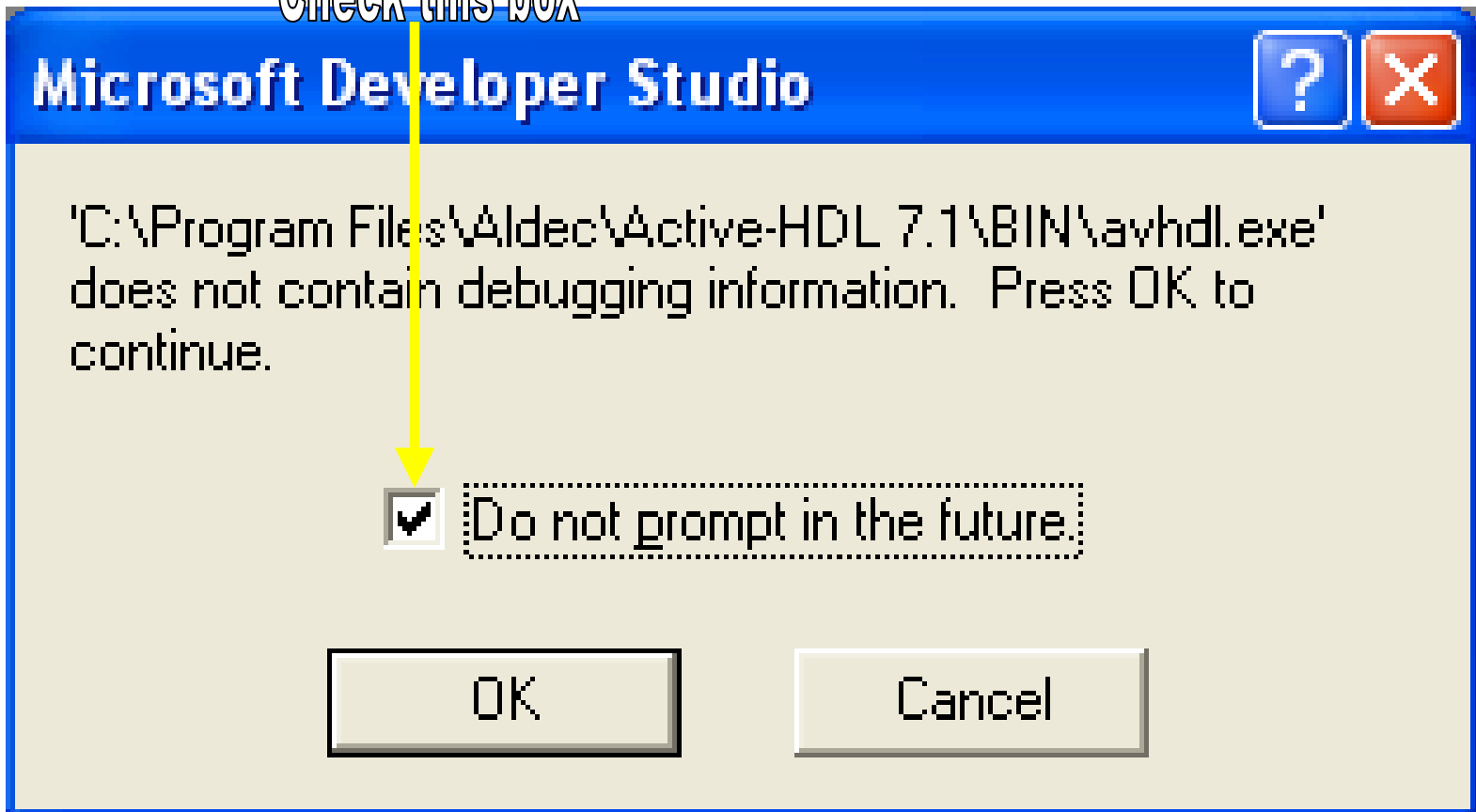


# Run/Debug



# Run/Debug 2

Check this box





# Driving Signals

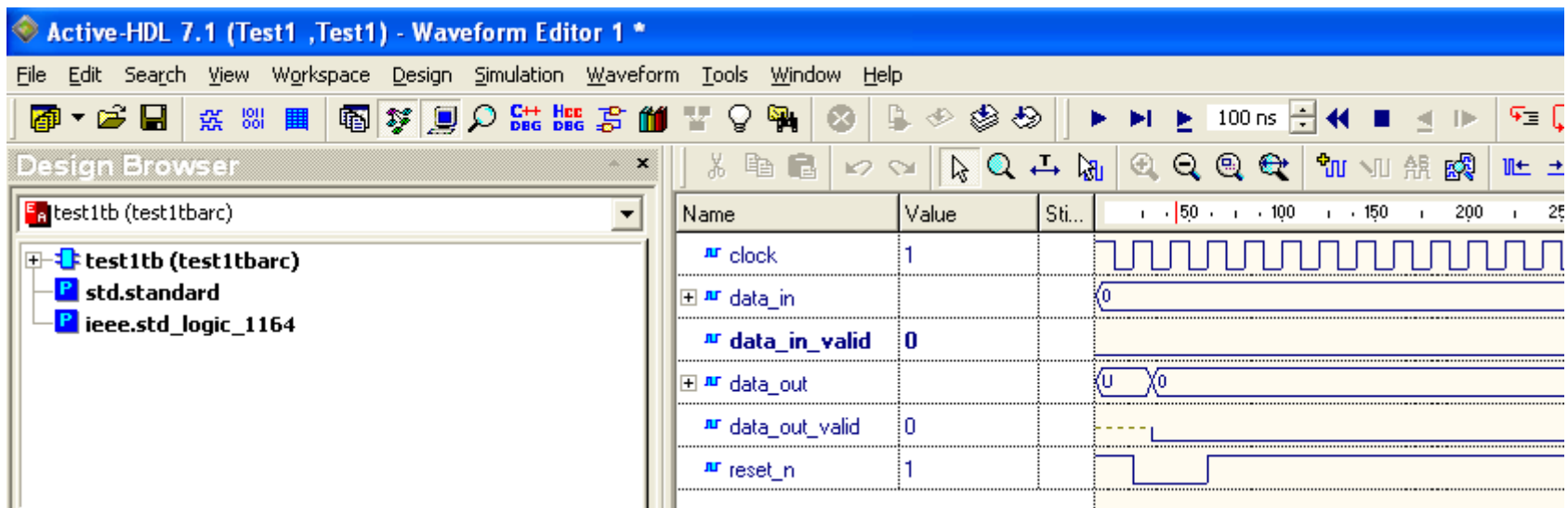
```
void Test1Driver::ThreadFunc()
{
    SpvEvent pClock("Test1Tb.clock", AtPos);

    SpvSig resetN("Test1Tb.reset_n");
    SpvSig dataIn("Test1Tb.data_in");
    SpvSig dataEn("Test1Tb.data_in_valid");
    //TODO: Instantiate other signals here

    // Waiting 2 Clocks, Resetting for 2 clocks and continue
    Wait(pClock);
    resetN = 0;
    Wait(pClock,2);
    resetN = 1;

    //TODO: Drive data here
}
```

# Driving Signals - Waveform



# Exercise 1

See what happens when the signal name is incorrect

- Drive the `data_in` and `data_in_enable` signals
  - Experiment with different values, including those larger than the signal width.
  - Change signal values over time
- Drive signals in a loop – one iteration per clock
- Add a “ready” signal to the design. Have the DUT raise and lower the ready. Have the driver check the ready signal before driving.
- Use the rise of the ready signal asynchronously. (Hint: Create a new event)

# Driving Signals – X and Z

```
Wait(pClock);
resetN = 0;
Wait(pClock,2);
resetN = 1;
dataIn(ZVal) = 0xF;
dataEn       = 0;

Wait(pClock);
dataEn       = 1;
unsigned i;
for(i = 0; i < 100; i++)
{
    dataIn = i;
    Wait(pClock);
}
dataEn     = 0;
dataIn(ZVal) = 0xF;
```

# Driving Signals – Slices

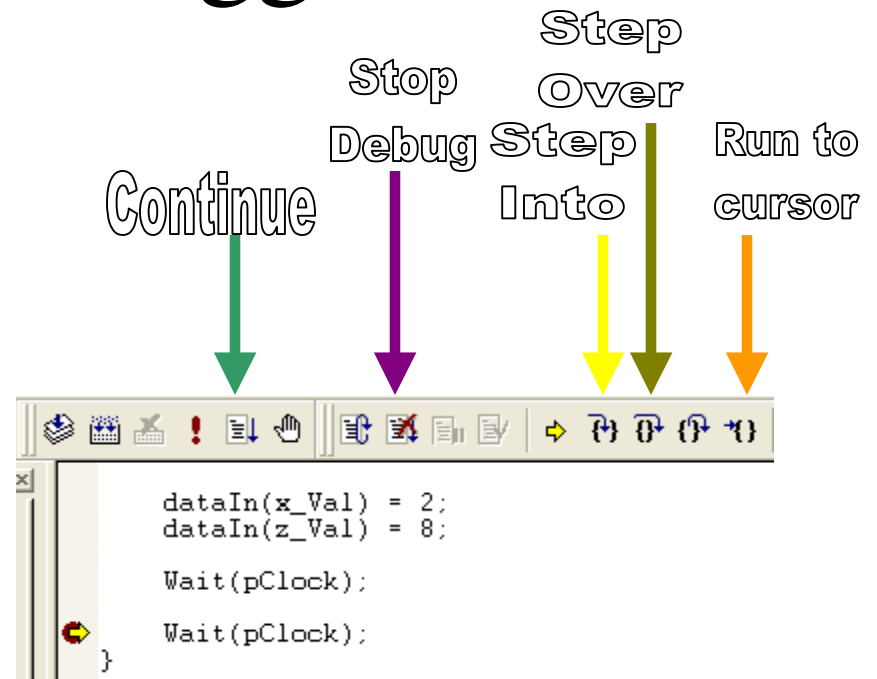
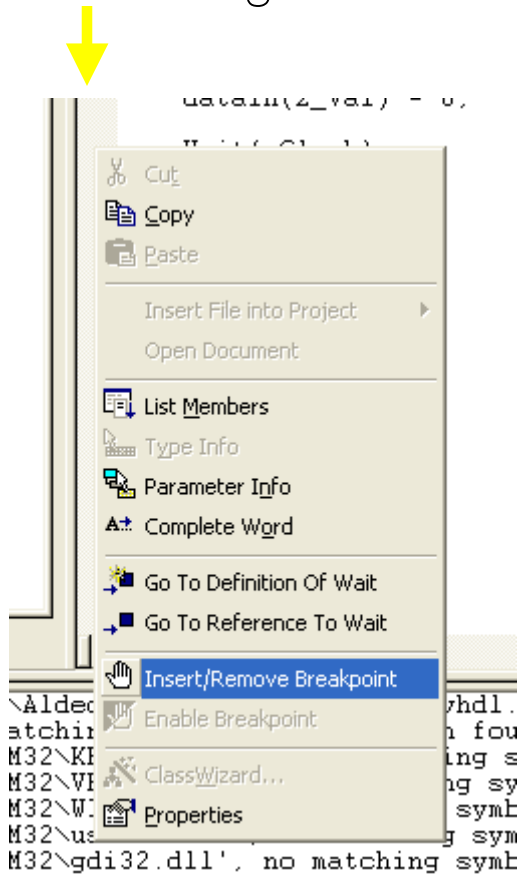
```
Wait(pClock);
dataEn      = 1;
unsigned i;
for(i = 0; i < 100; i++)
{
    dataIn[0](ZVal)      = 1;          //Drive z to bit 0
    dataIn[1]            = !dataIn[1]; //Toggle bit 1
    dataIn(2,3)(XVal)    = 3;          //Drive x to bits 2 through 3
    dataIn(3, dataIn.Size() - 1)(XVal) = i; //Drive x to bits 4 through highest with i
    Wait(pClock);
}
dataEn      = 0;
dataIn(ZVal) = 0xF;
```

# Using the Debugger

- Breakpoints
- Step Into
- Step Over
- Run to Cursor
- Set execution position

# Using the Debugger 2

Right-Click in margin

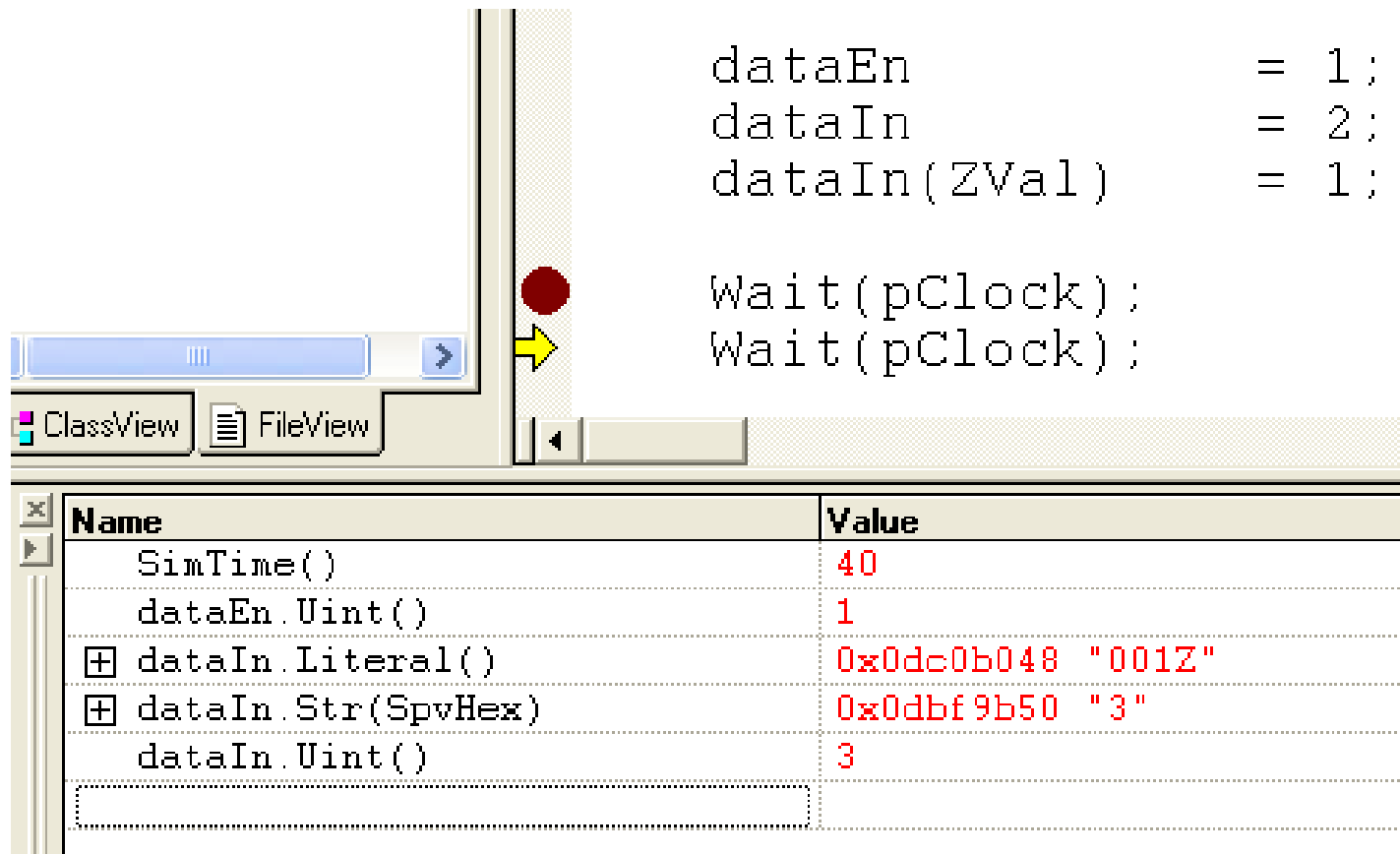


# Using the Debugger – SPV helper functions

- File should include `SpvHFile.h`
- `SimTime()` will return the simulation time as a 64 bit unsigned integer
- `sig.Uint64()` will return the first 64 bits of a signal
- `sig.Str(SpvDec/SpvHex/SpvBin)` returns a signal's numerical string representation (of any length)
- `XVal` and `ZVal` with `sig.Literal()`



# Using the Debugger - Screenshot



The screenshot shows a debugger interface. On the left, there are tabs for 'ClassView' and 'FileView'. A yellow arrow points to a red circular breakpoint marker on the left side of the code window. The code window displays the following code:

```
dataEn          = 1;  
dataIn          = 2;  
dataIn(ZVal)    = 1;  
  
Wait(pClock);  
Wait(pClock);
```

Below the code window is a variable watch window with the following data:

Name	Value
SimTime()	40
dataEn.Uint()	1
dataIn.Literal()	0x0dc0b048 "001Z"
dataIn.Str(SpvHex)	0x0dbf9b50 "3"
dataIn.Uint()	3

# Processes Creation

- (Class must inherit from SpvBase)
- Add process function to class header
- Add process function to class body
- Call `StartTProc(this, (SPVPM)&MyClass::MyFunc);`
- Optionally, save the returned `SpvTProcCtrl` pointer
- **REMEMBER:** Thread processes are not cyclical like HDL processes!

## Exercise 2

- Create a new process just for driving reset.
  - Move the current resetting code out of ThreadProc and into the new process.
  - While you are at it, use Wait() on time, instead of Wait() on the clock, for the reset duration.
- Create a third process that waits on reset and stops and restarts ThreadProc when that happens. (hint: Use the Restart() function of ThreadProc's SpvTProcCtrl pointer)

# SpvBitVec and Friends

- SpvBitVec helps us deal with bit oriented data.
  - Includes index and slice operations
  - Interoperable with C *unsigned* data type
  - Interoperable with SpvSig
- SpvBitVecCollector accumulates chunks of data into one bit vector.
- SpvBitVecIterator separates a bit vector into data chunks.
- Both SpvBitVecCollector and SpvBitVecIterator are declared in SpvBitVecIterator.h

# SpvBitVec

- Default initial size is 32 bits, but this (as well as an initial value) can be set in the construction. Upper limit is define by system memory.
- Most operations on SpvBitVec (and SpvSig) and unsigned are transparent. Some require the Uint()/Uint64() function.
- Assignments from signals/bit vectors of different size will not change the vector size. Use the Copy() function to force a resize.
  - This is not true in bit vector **construction!**
  - Manual resize can be accomplished with the Resize() function.
- Debug functions exist – similar to SpvSig.
- Convenience functions exist for:
  - Zeroing out all bits (Zero() function)
  - Setting all bits to one (One() function)

# SpvBitVec (cont)

- Randomized content (Gen() function)
- Bit vector concatenation (Pack() function)
- Parity (Parity() function – calculates **odd** parity)
- Counts the number of ones in the vector (Sum1() function)
- Bit vector comparison (FindUnmatchedBit() function)
- Str() and SliceStr functions return vector/slice contents in string form. Also, ostream (e.g. cout) is supported.
- Most arithmetic and bitwise operators, including the shift operators, work for bit vectors.
- Bit vectors should be used where the bit orientation and/or unlimited length is an advantage. Otherwise, stay with the native data types.

# SpvBitVecCollector

- Initialized (and cleared) with `Init()`
  - Sets maximum bit vector size
  - Sets default chunk size
  - Optionally, the chunks can be saved in reverse order
- `SetNext()` pushes chunks. Optionally, the chunk size can be specified for each call.
- `BitsCollected()` returns the number of bits accumulated. `IsEmpty()` returns true when `BitsCollected()` returns zero.
- Either assignment to a bit vector or the `Collected()` function return the current contents.

# SpvBitVecIter

- Initialized (and cleared) with `Init()`
  - Sets initial contents
  - Sets default chunk size
  - Optionally, the chunks can be ladled out in reverse order
- `Next()` pops chunks. Optionally, the chunk size can be specified for each call.
- `BitsRemaining()` returns the number of bits left (not popped). `IsEmpty()` returns true when `BitsRemaining()` returns zero.
- Assignment to a bit vector or the `LastIteration ()` function return the last popped chunk.



# STL Alternatives

- If the bit orientation is not important, then there are better alternatives to the bit vector collector/iterator types
- deque – double ended queue, preferable when pushing/popping at the front or back.
- vector – contiguous array, preferable when accessing the middle of the array.
- Declaration example:
  - `deque< unsigned > d;`
  - `vector< unsigned > v;`
- Both have `push_back()`, `push_front()`, `front()`, `back()`, `pop_back()`, `size()`, and index operations.
- Deque also has a `pop_front()` function.

# STL Alternatives - example

```
#include <deque>
using namespace std;          //So we won't have to use the full name, std::deque

....

deque< unsigned > d;

//push 5 elements
d.push_back(1);
d.push_back(2);
d.push_back(4);
d.push_back(3);
d.push_back(7);

d[2] = 10; //Overwrite 4 with 10

unsigned front = d.front(); //front will be 1
d.pop_front();
front = d.front();         //front will be 2

unsigned back = d.back(); //back will be 7
d.pop_back();
back = d.back();          //back will be 3
```

# Exercise 3

- Use the SpvBitVec class to write functions for:
  - Bit reversal
  - Little endian to big endian conversion. (Arbitrary byte size, but must be a factor of the total size)
- Functions should take a bit vector as a parameter and return the new, converted, vector.
- Do two implementations:
  - Using the STL deque.
  - Using the SpvBitVecCollect/Iter classes.

# Collection

- Collector is similar to Driver, but has opposite purpose. Here we accumulate data from the DUT.
- We could collect in the driver (and will sometimes do this), but generally we want to keep the functionality in separate classes.

# Exercise 4

- Add the code to the collector class to collect packets at the output.
  - First, modify the driver to drive binary (not x or z) data when enable is high.
  - Reset is not driven in the collector process, so we have to handle reset in a similar fashion as in exercise 2. (Separate processes)
  - Use the `SPV_OUT` macro and the bit vector string functions to output the packet to both the screen and log file (spv.log).  
E.g. `cout<<"PrintMe"<<endl;`  
→ `SPV_OUT(<<"PrintMe"<<endl);`

# Exercise 5

- Add another process to collect the packets at the DUT input.
  - Use the `std::deque` class to collect the `SpvTProcCtrl` for each thread and loop over them in the Reset Listener thread to restart the threads at reset, instead of the current implementation.
  - If the new thread looks quite similar to the last one, don't worry, but start thinking about code reuse...
- As the next step, now compare the packet collected by each process. Identify the first wrong bit. Try inserting a bug into the DUT and see if it gets caught.
- Now change the DUT to cause a very large delay. Have the driver send multiple small, non-identical, packets. The check will probably fail. What's the problem with the check? How will you fix it?

# Transfer Function

- DUT may manipulate the input (output  $\neq$  input)
- Prior to comparison we must model the transfer function of the DUT
- The collected input will be passed through a function that computes the expected output. This will be the basis for comparison.

# Exercise 6

- Change the DUT to perform some operation on the data before output. (e.g. bitwise negation, or a mapping function)
- In the C++, write a Transfer() function and call it before the comparison, using the result of Transfer() instead of the original input.



# Generation

- Definition – Stimulus fabrication
- Pseudorandom – deterministic with the same seed
  - Range
  - Generate from list of values
  - Weighted
  - Coverage based, random choice
- Non-Random
  - Constant
  - Sequence List
  - Coverage based, sequential

# Generation – SPV classes, partial list

- GenUnsigned() – range generation **function**  
(Defined in SpvGlobal.h)
- SpvGenConst()
- SpvGenInRange()
- SpvGenInRangeList()
- SpvGenNotInRangeList()
- SpvGenInRangeListOrder()
- SpvGenNextStep()

# Generation - Use

- Instantiate class and initialize
  - First parameter is generally the bit size of the result. (Retrieve with Size())
  - List generators can be initialized with either a string, or a vector of values.
- Call Gen() function – returns *unsigned*.
- Some generators can be Reset()
- Use SetName() to attach an ID to a generator instance. Use Name() to retrieve it. (Useful for debugging, amongst other things...)

# Generation - Example

```
SpvGenInRangeList  dataGen(dataIn.Size(), "0, 3 7, 9");//0, 3 through 7, 9
SpvGenInRange      lengthGen(32, 1, 100);
SpvGenInRangeListOrder idleGen(32, "5, 10, 3");

while(1)
{
    Wait(pClock);
    dataEn      = 1;
    unsigned len  = lengthGen.Gen();
    unsigned j;
    for(j = 0; j < len; j++)
    {
        Wait(pClock);
        dataIn = dataGen.Gen();
    }
    dataEn      = 0;
    Wait(pClock, idleGen.Gen());
}
```

# Generation – Vector Init

```
vector<SpvRange> v;
v.push_back((0, 0));
v.push_back((3, 7));
v.push_back((9, 9));

SpvGenInRangeList      dataGen(dataIn.Size(), v);
SpvGenInRange          lengthGen(32, 1, 100);
SpvGenInRangeListOrder idleGen(32, "5, 10, 3");

while(1)
{
    Wait(pClock);
    dataEn      = 1;
    unsigned len = lengthGen.Gen();
    unsigned j;
    for(j = 0; j < len; j++)
    {
        Wait(pClock);
        dataIn = dataGen.Gen();
    }
    dataEn      = 0;
    Wait(pClock, idleGen.Gen());
}
```

# Exercise 7

- Add randomized generation to the driver
  - Experiment with the different classes
  - Create a range generator for the length. Have its min and max be set based on another generator.

# Weighted Generation

```
SpvGenConst  dataGen1(dataIn.Size(), 0);
SpvGenInRange dataGen2(dataIn.Size(), 1, 3);

SpvGenWeighted dataGen(dataIn.Size());
dataGen.AddGenElem(dataGen1, 90);           //90 weight
dataGen.AddGenElem(dataGen2, 10);          //10 weight

SpvGenInRange  lengthGen(32, 1, 100);
SpvGenInRangeListOrder idleGen(32, "5, 10, 3");

while(1)
{
    Wait(pClock);
    dataEn = 1;
    unsigned len = lengthGen.Gen();
    unsigned j;
    for(j = 0; j < len; j++)
    {
        Wait(pClock);
        dataIn = dataGen.Gen();
    }
    dataEn = 0;
    Wait(pClock, idleGen.Gen());
}
```

# Exercise 8

- Add weighted generation of the packet length to the driver.
  - 25% generate from 1 to 3
  - 25% generate from 20 to 22 or 40 to 42
  - 25% generate one of 50, 60, 70
  - 25% generate (in order) 80, 81, 82
- Experiment with the weights. What happens if the sum of the weights is greater than 100?



# Generation By File

- Text file defines named generators
- Generators can be retrieved at runtime by name
- Allows end-user test configuration without compilation
- INCLUDE directive allows preexisting definitions to be added to a definition file.
- Redefinition of generator is possible (last definition is conclusive)
- Redefinition together with INCLUDE allows for defaults to be defined and overridden per test.

# File Syntax

- `//` - comment to end of line
- `START:`
- `STOP:`
- `DEFINE: {name} [val]`
- `INCLUDE: "{FileName}"`
- `EXT_ENUM_LIST: {name} "{en1, en2, ...}" //enumeration en1=0, en2=1, ...`
- `NUMBER: {name} {value}`
- `GC: {name} {bitsize} {val} //const`
- `GR: {name} {bitsize} {min} {max} //range`
- `GS: {name} {bitsize} {stepsize} {from} {to} //step`
- `GRANGE: {name} {bitsize} {"rangelist"} {truefalse} //range list (incl.lexc.)`
- `GRANGE: {name} {bitsize} {"rangelist"} //sequential range list`
- `GPERCENT: {name} {bitsize} {"genlist"} {percentlist} //weighted`

# File Syntax - Examples

START:

```
//NOTE: WHEN ENTRY IS DUPLICATED, LAST DEFINITION IS  
//CONCLUSIVE. HERE ALL GENERATORS ARE NAMED  
//DataGen, BUT ONLY THE LAST ONE IS EFFECTIVE
```

```
//Const (here, always 3)  
GC: DataGen 32 3
```

```
//Range (here, 0 to 3)  
GR: DataGen 32 0 3
```

```
//Step up/down (by last param). Step size is second param.  
//Starting value is penultimate param. (here, 0, 3, 6, 9, 12, 15, 2, 5, etc)  
GS: DataGen 32 3 0 15 0 true
```

```
//Range list/not in list (by last param) (here, 0 to 4, and 15)  
GRANGE: DataGen 32 "0-4,15" true
```

```
//Range list order up/down (by last param)  
GRANGE0: DataGen 32 "0-4,15" true
```

```
//Weighted generation  
GC: DataGen1 32 3  
GC: DataGen2 32 7  
GPERCENT: DataGen 32 "DataGen1,DataGen2" "10,90"
```

STOP:

# Generation By File – Code

```
//File contents:  
//GR: DataGen 32 0 3  
//GC: IdleGen 32 2  
//GRANGE0: LenGen 32 "0-4,15" true
```

```
Wait(pClock);
```

```
SpvGen* dataGen  = NULL;  
SpvGen* idleGen  = NULL;  
SpvGen* lenGen   = NULL;  
SD::GetGen(dataGen, "DataGen");  
SD::GetGen(idleGen, "IdleGen");  
SD::GetGen(lenGen, "LenGen");
```

```
while(1)
```

```
{  
    unsigned len = lenGen->Gen();           //Generate packet length. Note use of -> because of pointer type  
    dataEn = 1;  
    unsigned i;  
    for(i = 0; i < len; i++)  
    {  
        dataIn = dataGen->Gen();           //Generate data nibble  
        Wait(pClock);  
    }  
    dataEn = 0;  
  
    Wait(pClock, idleGen->Gen());          //Time between packets  
}
```

# Exercise 9 – Generation By File

- Change your code to use file based generators
  - Define the file generators to the same definitions as currently coded.
  - Change the definitions, keeping the same generator names, and rerun the simulation.

# Dirty Words – A touch of OOD concepts

- Base Class – Class that provides some basic definitions and optionally, basic functionality.
- Child Class – Class that **inherits** from some base class. Generally, it has at least the same definitions as its base, but can extend or replace the implementations (“hook function”), as well as add unrelated functionality.
- Overrideable functions are defined with the *virtual* keyword.
- In SPV, all generators are child classes of SpvGen. SpvGen defines the Gen() function, but not its implementation. Each generator class implements the Gen() function differently.
- Through an SpvGen pointer we can call Gen() on the generator returned by SD::GetGen() without knowing or caring which class it actually is.

# Coverage

- Shows what situations we have reached in our simulations.
- Let's us determine the efficiency of the generation. Can also be considered a self check on the verification coding and test configuration.
- Helps show how much of the test plan has been accomplished.

# Coverage (cont)

- Coverage is composed as:
  - Sampling trigger
  - Values to record
- Most commonly, the sampling trigger will be a signal edge and the recorded values will come from signals as well.
- Sometimes, things are more complex
  - History – coverage includes non-current signal values
  - C++ values – recorded values don't exist in the simulator
  - Sample trigger is a complex combination and/or chain of events



# SpvCover

- SpvCover is the main coverage class in SPV
  - Add signals to record with calls to AddItem()
  - Determine the sampling trigger with Start()
  - Suspend sensitivity to trigger with Stop()
  - Resume sensitivity with Restart()
- Extension for more complex cases is possible with derivation and overrides.
- For the case of a complex trigger event, it is often easier to define a derivative signal in the HDL realm than to derive & override.
- Start() has option to load coverage from previous run

# Coverage Example

```
void Test1Tb::Init()
{
    m_Driver->Start();
    m_Col->Start();

    m_Cov->AddItem("Test1Tb.data_in");

    //enable_and_clock is an HDL signal that I have added. It is defined as (clock & data_en)
    m_Cov->Start("DataCov", SpvEvent("Test1Tb.enable_and_clock", AtPos));
}
```

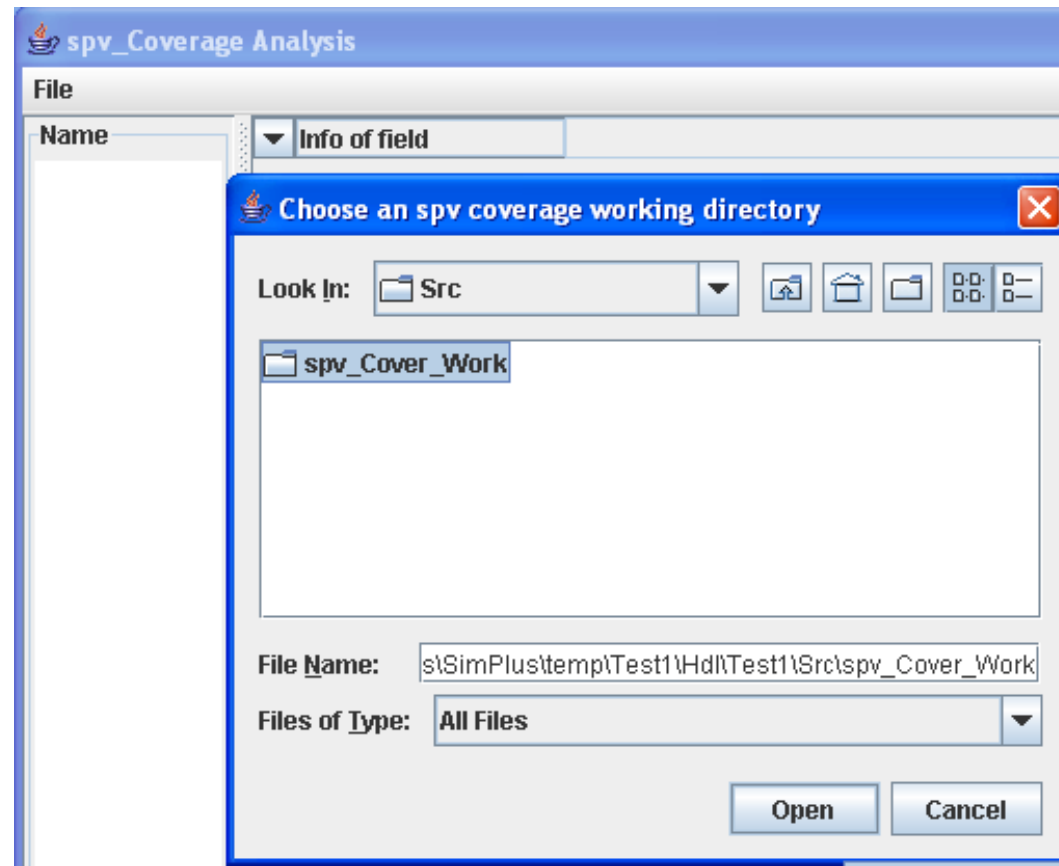
# Cross Coverage

- Cross coverage is the vector (cross) product of multiple coverage values.
- Simple coverage can be thought of as a degenerate instance of cross coverage.
- Cross coverage in SPV is simply additional calls to `AddItem()` before calling `Start()`

# Exercise 10

- Add a mode signal
  - Add to DUT input
  - Add mode to driver – drive it once each packet
  - Cover the mode input with SpvCov. The trigger event should be the mode signal itself.

# Coverage Display – Open File



# Coverage Display 2

