1 Analyzer Developer's Guide

User-written analyzers are a key feature of AMD's SimNow™ simulator software. Analyzers allow the user to insert arbitrary pieces of C code into various points of execution of the processor model. There are many reasons to write analyzers; for example, you could write an analyzer to:

- Create workload traces
- Gather information to help you debug software
- Modify architectural semantics
- Collect information to feed to architectural studies
- Implement tiny devices

An analyzer must implement the thin C *Analyzer* interface. This interface includes four required functions and four optional functions. The simulated processor model calls into your analyzer through this interface.

The simulated processor model provides an *AweSim* interface to the analyzers. An analyzer can call through this interface to get processor model state, access simulated memory, break simulation, and more.

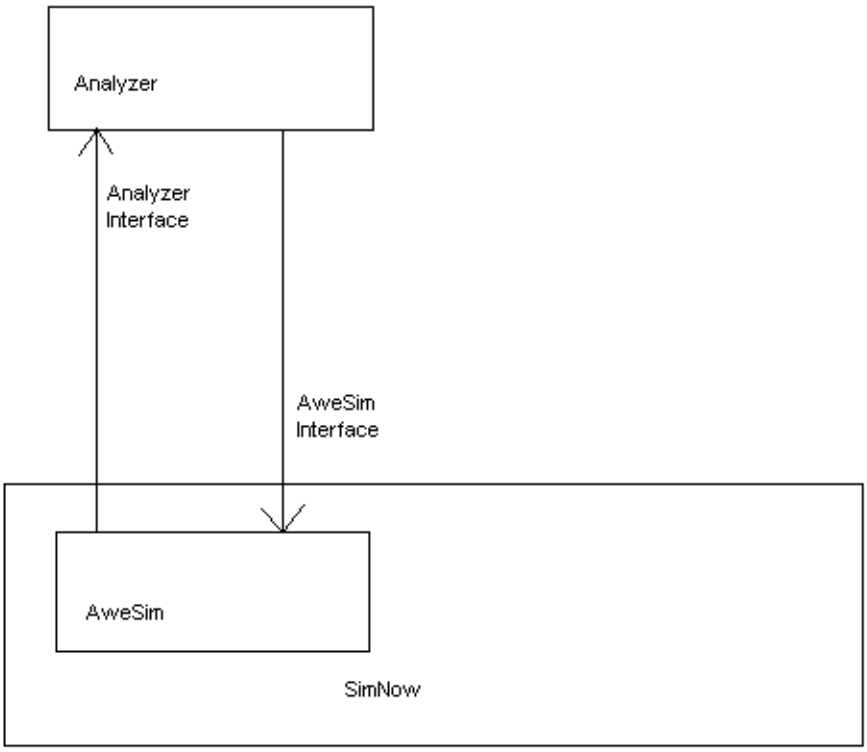

Figure 1-1

There are a dozen available intercepts. Each analyzer broadcasts which of these intercepts he wants to hook. The analyzer is notified when an event occurs that is hooked by the analyzer.

To develop analyzers, the developer must use an AMD64 system running either 64-bit Windows® or 64-bit Linux operating system. If you are running 64-bit Windows, you will need Visual Studio 2005 installed. Otherwise, you will need the gcc toolchain installed.

AMD provides a sample analyzer that implements CPU logging. You may base your analyzer off this sample. The sample will give you a known working starting point, which will make implementing your analyzer easier than attempting to build it from just this documentation. The sample analyzer logs execution in a human-readable form, and implements all of the optional parts of the analyzer interface. The sample code is included in the *./analyzers/devel/cpulogger* directory of your SimNow installation. The header files required to implement an analyzer are included in the *./analyzers/devel* directory of your SimNow installation. No other header files or libraries are required to develop an analyzer.

1.1 TagSample source files

This section contains the source to a simple analyzer that implements a breakpoint when a CPUID instruction is executed. This analyzer is very simple, but is a useful reference while you are reading this developer's guide.

TagSample.cpp

```
// TagSample.cpp : Defines the entry point for the DLL
application.
//
//This sample illustrates how to tag specific instructions for
analysis, and
//  also how to generate a breakpoint on execution of a
specific instruction.

#include "typedefs.h"
#include "commoncpu.h"
#include "TagSample.h"
#include "Analyzers.h"

#include <string.h>

#ifdef WIN32
//Windows standard DLL entry point

BOOL APIENTRY DllMain( HANDLE hModule, DWORD
ul_reason_for_call, LPVOID lpReserved)
{
    switch (ul_reason_for_call)
  {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
```

```
                break;
        }
        return TRUE;
}
#endif

//AnalyzerLoaded - This analyzer has been loaded into memory

DLLEXPORT bool AnalyzerLoaded(CAweSim *pAweSim, const char
*pszArguments, void **ppVoid)
{
  MYDATA *pMyData = new MYDATA;
  pMyData->nInstructionCount = 0;
  pMyData->nInstructionTag = 0;
  pMyData->pAweSimModule = pAweSim;
  *ppVoid = (void *) pMyData;
  return(true);
}

//AnalyzerUnloaded - This analyzer is being unloaded from
memory

DLLEXPORT void AnalyzerUnloaded(void *pVoid)
{
}

//GetAnalyzerVersion - Get the version number of the interface
this Analyzer is using

DLLEXPORT UINT32 GetAnalyzerVersion()
{
  return(ANALYZER_CURRENT_VERSION);
}

//GetAnalyzerFlags - Return various flag bits. See the
ANALYZER_xxxx bits for definitions

DLLEXPORT UINT32 GetAnalyzerFlags()
{
  return(ANALYZER_IN_TRANSLATE | ANALYZER_IN_EXECUTETAG);
}

//GetAnalyzerDescription - Returns a short description of this
analyzer

DLLEXPORT bool GetAnalyzerDescription(char *pszDest, int
nDestSize)
{
  if((unsigned)nDestSize <= strlen("TagSample Analyzer"))
        return(false);
  strcpy(pszDest, "TagSample Analyzer");
  return(true);
}

//AnalyzerNotify - Called when one of the events requested by
GetAnalyzerFlags() happens
```

```
DLLEXPORT void AnalyzerNotify(void *pVoid, UINT32 nNotifyCode,
void *pParam1, void *pParam2)
{
  switch(nNotifyCode)
  {
        case ANALYZER_IN_TRANSLATE:
              HandleTranslate((INTRANSLATESTRUCT *) pParam1,
(MYDATA *) pVoid);
              break;
        case ANALYZER_IN_EXECUTETAG:
              HandleExecuteTag((UINT32) pParam1, (MYDATA *)
pVoid);
              break;
  }
}

//HandleTranslate - Called during translation of every
instruction.
//
//Note: This will be called only when the instruction is
translated, not when it
//                  is executed. Because of speculative
translation, it may never be executed.
//

void HandleTranslate(INTRANSLATESTRUCT *pInstruction, MYDATA
*pData)
{

  //    Check for CPUID.
  if((pInstruction->nOpcodeCount >= 2) && (*(pInstruction-
>pOpcodeBuffer+pInstruction->nOpcodeOffset) == 0x0F) &&
(*(pInstruction->pOpcodeBuffer+pInstruction->nOpcodeOffset+1)
== 0xA2))
  {
        if(pData->nInstructionTag == 0)
              pData->nInstructionTag = pData->pAweSimModule-
>AllocateInstructionTag();

        //    This is a CPUID instruction. Tag it so we get
notifications of its execution

        pInstruction->nInstructionTag = pData-
>nInstructionTag;
  }
}

//HandleExecuteTag - This routine is called when a tagged
instruction is executed
//

void HandleExecuteTag(UINT32 nTag, MYDATA *pData)
{
  UINT64 tCount = pData->pAweSimModule->InstructionCount();
  // Is this our CPUID instruction?
```

```
  if((nTag == pData->nInstructionTag) && (tCount != pData-
>nInstructionCount))
  {
        pData->nInstructionCount = tCount;
        pData->pAweSimModule-
>ControlSimulation(Control_Break);
  }
}
```

TagSample.h

```
#ifdef WIN32
#define DLLEXPORT __declspec(dllexport)
#else
#define DLLEXPORT
#endif

struct INTRANSLATESTRUCT;

class CAweSim;

extern "C" {
  DLLEXPORT UINT32 GetAnalyzerVersion();
  DLLEXPORT UINT32 GetAnalyzerFlags();
  DLLEXPORT bool AnalyzerLoaded(CAweSim *pAweSim, const char
*pszArguments, void **ppVoid);
  DLLEXPORT void AnalyzerUnloaded(void *pVoid);
  DLLEXPORT void AnalyzerNotify(void *pVoid, UINT32
nNotifyCode, void *pParam1, void *pParam2);
  DLLEXPORT bool GetAnalyzerDescription(char *pszDest, int
nDestSize);
}

//Variables used to track changes to the CPU state as
instructions execute

struct MYDATA
{
  UINT64 nInstructionCount;
  // Instruction count at our last breakpoint
  UINT32 nInstructionTag;
  // Tag we will use to tag CPUID instructions
  CAweSim *pAweSimModule;
  // Pointer to AweSim helper module
};

//Internal routines, specific to this sample

void HandleTranslate(INTRANSLATESTRUCT *pInstruction, MYDATA
*pData);
void HandleExecuteTag(UINT32 nTag, MYDATA *pData);
```

1.2 Building an Analyzer – Linux

Building your analyzer on Linux is a simple process. Use the makefile that is included in ~/analyzers/devel/cpulogger/makefile.

makefile

```
##
## Copyright 1995 - 2005 ADVANCED MICRO DEVICES, INC.  All Rights
Reserved.
##

include ../SDKmakefile.template
```

Then, create a file called 'makefile.local' in your analyzer's directory. You must add two lines to your 'makefile.local' file. Add one line that sets the variable 'OBJS' to a list of the object files that are a part of your analyzer. Add another line that sets the variable 'LIB' to the name of your analyzer's shared library.

makefile.local

```
##
## Copyright 1995 - 2004 ADVANCED MICRO DEVICES, INC.  All Rights
Reserved.
##

OBJS=TagSample.o
LIB=TagSample.so
```

1.3 Building an Analyzer - Windows®

To build your analyzer on Windows, you will need to create a solution file and configure it to build a Win64 DLL.

- First open Visual Studio 2005
- Add a new project (go to File - New - Project)
- Select Win32 project (not Win32 console) from the "New Project" dialog box.
- Select "DLL" as application type
- Open the configuration manager (go to Build – Configuration Manager)
- In "active solution platform", choose "new"
- Select "Win64 (AMD64)" as the new configuration type, and click OK
- Open the configuration manager again (go to Build – Configuration Manager)
- In "active solution platform", choose "edit"
- Select the Win32 platform and click "remove"

1.4 Analyzer Interface

The simulated processor model calls into your analyzer via the analyzer interface. The analyzer interface consists of four required functions and four optional functions. Each function takes a *CCaller* pCaller* parameter. Each instance of an analyzer has a unique *pCaller* parameter associated with it. This parameter allows you to manage multiple instances of your analyzer. Please refer to the provided sample cpulogger analyzer to see an example implementation that manages multiple instances.

1.4.1 Required Functions

All analyzers are required to implement these functions.

GetAnalyzerVersion

Implement *GetAnalyzerVersion* to return the version number of the interface that this Analyzer is using. Typically, you can simply return ANALYZER_CURRENT_VERSION.

Parameters

pCaller is a reference to the caller of this function. It should be unique for each instantiation of your analyzer. If the analyzer will be instantiated multiple times simultaneously, the analyzer can use this variable to determine which instance is being called.

AnalyzerLoaded

Your analyzer must implement *AnalyzerLoaded* to receive a request from SimNow to load your analyzer.

Parameters

pAweSim

A pointer to the AweSim Interface (see section , "AweSim Interface", on page 11 for more information). Typically an analyzer will save away the CAweSim pointer so that it can be used later.

pszArguments

A pointer to the arguments that the analyzer was loaded with. It may be NULL or an empty string.

ppVoid

ppVoid may have memory allocated to it in your implementation of AnalyzerLoaded; a pointer to this memory will then be passed to you in the functions AnalyzerUnloaded and AnalyzerNotify. You may use this for internal storage for your analyzer.

pCaller

A reference to the caller of this function. It should be unique for each instantiation of your analyzer. If your analyzer will be instantiated multiple times simultaneously, you can use this variable to determine which instance of your analyzer is being called.

pAr

A pointer to a *CArchive* object. If your analyzer supports BSD Load/Save, you will pass *pAr* to *CaweSim::LoadBSD()* to load the internal state of the analyzer if the *Carchive* pointer is not NULL.

GetAnalyzerFlags

Implement *GetAnalyzerFlags* to return which intercepts your analyzer hooks. If your analyzer hooks multiple intercepts, simply return the OR of all the intercepts you hook. See Section , "Intercepts", on page 20 for more information.

Parameter

pCaller is a reference to the caller of this function. It should be unique for each instantiation of your analyzer. If the analyzer will be instantiated multiple times simultaneously, the analyzer can use this variable to determine which instance is being called.

AnalyzerNotify

AnalyzerNotify is called whenever an event that your analyzer has hooked occurs.

Parameters

pVoid

A pointer to the data you may have allocated from *ppVoid* in *AnalyzerLoaded*.

nNotifyCode

An unsigned integer that indicates which intercept event you are being notified of.

pParam1, pParam2

Pointers to either variables or structures. The format is different for each intercept.

pCaller

A reference to the caller of this function. It should be unique for each instantiation of your analyzer. If your analyzer will be instantiated multiple times simultaneously, you can use this variable to determine which instance of your analyzer is being called.

1.4.2 Optional Functions

These optional functions are not necessary to write a functional analyzer. However, a user-friendly full-featured analyzer will override all of these functions.

GetAnalyzerDescription

GetAnalyzerDescription is called to get a human-readable ASCII description of your analyzer.

Parameters

pszDest

A pointer to a string where the analyzer description will be saved in.

nDestSize

An integer that contains the amount of memory in Bytes allocated for *pszDest*. If your description length is greater than this size, you must return *false* and not copy your description to *pszDest*

pCaller

A reference to the caller of this function. It should be unique for each instantiation of your analyzer. If your analyzer will be instantiated multiple times simultaneously, you can use this variable to determine which instance of your analyzer is being called.

Return Value

You must return *true* if your description length is less than *nDestSize*, and you copied your description successfully to *pszDest*. Otherwise, return *false*.

EnableAnalyzer

EnableAnalyzer is called both to enable and disable your analyzer. Note that your analyzer may be disabled and later re-enabled.

Parameters

bEnable

True if the analyzer is being enabled, and *false* if it is being disabled.

pCaller

A reference to the caller of this function. It should be unique for each instantiation of your analyzer. If your analyzer will be instantiated multiple times simultaneously, you can use this variable to determine which instance of your analyzer is being called.

AnalyzerUnloaded

AnalyzerUnloaded is called to unload your analyzer. If your analyzer has any cleanup to do (for example, deleting storage that you allocated), it must be done here.

Parameters

pVoid

A pointer to the data you may have allocated from *ppVoid* in *AnalyzerLoaded*.

pCaller

A reference to the caller of this function. It should be unique for each

instantiation of your analyzer. If your analyzer will be instantiated multiple times simultaneously, you can use this variable to determine which instance of your analyzer is being called.

SaveAnalyzer

SaveAnalyzer is called to save internal state from your analyzer into a SimNow BSD. Your analyzer should implement this function only if it supports BSD Load/Save.

Parameters

pAr

A pointer to a *CArchive* object. If your analyzer supports BSD Load/Save, you will pass *pAr* to *CaweSim::SaveBSD()* to save the internal state of the analyzer if the *Carchive* pointer is not *NULL*.

pCaller

A reference to the caller of this function. It should be unique for each instantiation of your analyzer. If your analyzer will be instantiated multiple times simultaneously, you can use this variable to determine which instance of your analyzer is being called.

1.5 AweSim Interface

The AweSim interface is composed of a number of utility functions that are available for your analyzer to call. Your analyzer calls into the simulated processor model via this AweSim interface.

LogPrintf

LogPrintf outputs the given string to the SimNow message log output. Usage of this function is similar to *printf*.

Parameter

pFormat

A pointer to a string that contains the string which needs to be logged to the message log output.

Return Value

The return value is the number of characters written, or negative if an error occurred.

ErrPrintf

ErrPrintf outputs the given string to the SimNow error log output. Usage of this function is similar to *printf*.

Parameter

pFormat

A pointer to a string that contains the string which needs to be logged to the error log output.

Return Value

The return value is the number of characters written, or negative if an error occurred.

LoadBSD

LoadBSD is used to restore internal state of your analyzer when a BSD is loaded into SimNow. Your analyzer will typically call *LoadBSD* when *AnalyzerLoaded* is called.

Parameters

pAr

A pointer to the *CArchive* object that you were provided in *AnalyzerLoaded*.

pBuffer

A pointer to the memory location that *LoadBSD* will restore state to.

Length

The number of bytes that will be read from *pAr* into *pBuffer*

SaveBSD

SaveBSD is used to save the internal state of your analyzer when SimNow saves a BSD. Your analyzer will typically call *SaveBSD* when *SaveAnalyzer* is called.

Parameters

pAr
A pointer to the *Carchive* object that you were provided in *SaveAnalyzer*

pBuffer
A pointer to the memory location that *SaveBSD* will save state from.

Length
The number of bytes that will be stored to *pAr* from *pBuffer*

GetX8664SMMState

GetX8664SMMState copies CPU state in the X86-64 SMM save-state area format to the supplied *X8664SMMSTATE* pointer.

Parameter

pState
A pointer to the *X8664SMMSTATE* structure.

SetX8664SMMState

SetX8664SMMState sets CPU state in the X86-64 SMM save-state area format from the supplied *X8664SMMSTATE* pointer.

Parameter

pState
A pointer to the *X8664SMMSTATE* structure.

GetFXSAVEState

GetFXSAVEState copies CPU state in the 64-bit FXSAVE image format to the supplied *FXSAVEAREA* pointer.

Parameter

pState
A pointer to the *FXSAVEAREA* structure.

SetFXSAVEState

SetX8664SMMState sets CPU state in the 64-bit FXSAVE image format from the supplied *FXSAVEAREA* pointer.

Parameter

pState

A pointer to the *FXSAVEAREA* structure.

AllocateInstructionTag

AllocateInstructionTag allocates a unique instruction tag. These instruction tags are used by the *ANALYZER_IN_EXECUTETAG* intercept.

Return Value

AllocateInstructionTag returns the unique instruction tag

InstructionCount

InstructionCount returns the number of instructions executed by the simulated processor model.

ReadPhysicalMemory

ReadPhysicalMemory is used to read simulated memory based on physical address.

Parameters

PhysAddress

Beginning address of memory request.

pBuffer

A pointer to where the data will be read in to.

nCount

An unsigned integer that contains the number of bytes to read.

pBytesDone

A pointer to an unsgned 32-bit integer that stores the number of bytes read.

Return Value

ReadPhysicalMemory returns *true* if the memory request was successful. Otherwise it returns *false*.

WritePhysicalMemory

WritePhysicalMemory is used to write simulated memory based on physical address.

Parameters

PhysAddress
Beginning address of memory request.

pBuffer
A pointer to where the data will be written from.

nCount
An unsigned integer that contains the number of bytes to write.

pBytesDone
A pointer to an unsgned 32-bit integer that stores the written number of bytes.

Return Value

WritePhysicalMemory returns *true* if the memory request was successful. Otherwise it returns *false*.

ReadLinearMemory

ReadLinearMemory is used to read simulated memory based on linear address.

Parameters

LinAddress
Beginning address of memory request.

pBuffer
A pointer to where the data will be read from memory into.

nCount
An unsigned integer that contains the number of bytes to read.

pBytesDone
A pointer to an unsgned 32-bit integer that stores the number of bytes read.

Return Value

ReadLinearMemory returns *true* if the memory request was successful. Otherwise it returns *false*.

WriteLinearMemory

WriteLinearMemory is used to write simulated memory based on linear address.

Parameters

PhysAddress
Beginning address of memory request.

pBuffer

A pointer to where the data will be written to memory from.

nCount

An unsigned integer that contains the number of bytes to write.

pBytesDone

A pointer to an unsgned 32-bit integer that stores the number of bytes written.

Return Value

WriteLinearMemory returns *true* if the memory request was successful. Otherwise it returns *false*.

ReadIOPort

ReadIOPort is used to read from simulated IO ports.

Parameters

Port

Beginning port of IO request.

pBuffer

A pointer where the data will be read from the IO port to.

nCount

An unsigned integer that contains the number of bytes to read. Must be one, two or four.

Return Value

ReadIOPort returns *true* if the IO access was successful. Otherwise it returns *false*.

WriteIOPort

WriteIOPort is used to write to simulated IO ports.

Parameters

Port

Beginning port of IO request.

pBuffer

A pointer from where the data will be written to the IO port.

nCount

An unsigned integer that contains the number of bytes to write. Must be one, two or four.

Return Value

WriteIOPort returns *true* if the IO access was successful. Otherwise it returns *false*.

ControlSimulation

ControlSimulation is used to breakpoint, save BSD, or exit SimNow.

Parameter

NewControl

Indicates whether to stop simulation via a breakpoint, save the BSD, or exit SimNow. If you ask to save the BSD, it will get saved as "simnow-synchsave.bsd".

Return Value

ControlSimulation does not return if the control command succeeded. If the control command fails, it returns *false*.

Remarks

The simulation cannot be controlled from *ANALYZER_IN_EVENT, ANALYZER_IN_INTERCEPTIO* and *ANALYZER_IN_INTERCEPTMEM*, or if a control is requested in the middle of a memory request generated by the analyzer itself.

GetA20Mask

GetA20Mask returns the address mask in effect for the simulated processor.

Return Value

If gate A20 is not pulled, *GetA20Mask* will return ~(ADDRTYPE)0. If A20 is pulled, *GetA20Mask* will return ~(ADDRTYPE(1<<20)).

GetPhysAddress

GetPhysAddress returns the corresponding physical address for a given virtual address.

Parameter

LinAddresss

The virtual address that you want the phyiscal address for.

Return Value

GetPhysAddress returns the physical address for the virtual address *LinAddress*.

GetMemType

GetMemType returns the corresponding memory type for a given virtual address.

Parameter

LinAddresss

The virtual address that you want the memory type for.

Return Value

GetMemType returns the memory type for the virtual address *LinAddress.*

ReadMSR

ReadMSR is used to read the simulated processor's model specific registers.

Parameters

MSRNumber

The address of the MSR that is being read

pResult

The buffer that the MSR data will be read to

Return Value

ReadMSR returns *true* if the MSR was found and read into *pResult*. Otherwise it returns *false*.

WriteMSR

WriteMSR is used to write the simulated processor's model specific registers.

Parameters

MSRNumber

The address of the MSR that is being written

MSRValue

The value written to the MSR

Return Value

WriteMSR returns *true* if the MSR was found and read written. Otherwise it returns *false*.

Disassemble

Disassemble disassembles the instruction buffer into human-readable form.

Parameters

pBuffer

A pointer to the instruction buffer that you want disassembled

Length

A reference where the length of the instruction disassembled will be stored

Return Value

Disassemble returns a pointer to the human-readable string representing the instruction that was disassembled. The length parameter is updated to reflect the number of instruction bytes for this instruction.

ForceRebuildAnalyzerFlags

ForceRebuildAnalyzerFlags is used to force the simulated processor to re-sync itself with its loaded analyzers. This needs to be used before your analyzer, while in the process of being disabled or unloaded, generates a memory request. More commonly, this will need to be used if your analyzer dynamically changes the value it returns from *GetAnalyzerFlags.*

IsRunning

IsRunning is used to determine if the simulated machine is running. An analyzer may be called when the simulation machine is not running, for example, if an analyzer is intercepting memory and the user initiates a memory access through the SimNow debugger.

Return Value

true if the simulated machine is running, *false* otherwise.

NotifyLater

NotifyLater is used to indicate to the CPU model that your analyzer needs to be called back after he has executed a given number of instructions. Your analyzer will be called back with the ANALYZER_IN_EVENT intercept.

Parameter

Instructions

The number of instructions that the processor will execute before calling back.

GetPageBit

GetPageBit returns the corresponding number of bits of the page size for a given virtual address.

Parameter

LinAddresss

The virtual address that you want the page size for.

Return Value

GetPageBit returns the number of bits of the page size (unsigned int) for the virtual address *LinAddress.*

SetClkIpc

SetClkIpc modifies the simulated processor's fixed IPC.

Parameter

Instr
The numerator of the IPC fraction
Cycle
The denominator of the IPC fraction

GetCoreFreq

GetCoreFreq returns the current operating frequency of the simulated processor.

Return Value

GetCoreFreq returns the current operating frequency of the simulated processor.

1.6 Intercepts

Analyzers broadcast which intercepts they hook via their implementation of the *GetAnalyzerFlags* function. The analyzer is notified via *AnalyzerNotify* when an event occurs that the analyzer hooks.

Please note the difference between *execution-time* and *translation-time* in our processor model. The SimNow processor model translates instructions, caches them, and may then execute them multiple times. The SimNow processor model may translate instructions that never execute. The intercepts ANALYZER_BB_TRANSLATE and ANALYZER_IN_TRANSLATE fire during *translation-time*.

Intercepts	Description
ANALYZER_BB_TRANSLATE	Generated when the processor model begins to translate a basic block. No other information is passed in *pParam1* or *pParam2*
ANALYZER_IN_TRANSLATE	Generated when the processor model begins to translate an instruction. An *INTRANSLATESTRUCT* pointer is passed in pParam1.
ANALYZER_IN_EXECUTEALL	Generated at the start of every instruction execution. No other information is passed in *pParam1* or *pParam2*.
ANALYZER_IN_SWI	Generated when a software interrupt is being executed. The vector of the interrupt is passed in pParam1
ANALYZER_IN_HWI	Generated when a hardware interrupt is being taken. The vector of the interrupt is passed in *pParam1*.
ANALYZER_IN_EXC	Generated when a processor exception is being taken. An *INEXCSTRUCT* pointer is passed in *pParam1*.
ANALYZER_IN_EXECUTETAG	Generated when a tagged instruction is

	executed. The unique instruction tag is passed in pParam1.
ANALYZER_IN_MONITORIO	Generated when an I/O instruction is being executed. An *INMONITORIOSTRUCT* pointer is passed in *pParam1*.
ANALYZER_IN_INTERCEPTIO	Generated when an I/O instruction is being executed. This intercept allows the analyzer to override the result of I/O instruction. An *ININTERCEPTIOSTRUCT* pointer is passed in pParam1. If your analyzer aborts the IO intercept; i.e., your handler does not handle the I/O, you must set the *bAbort* field of the *ININTERCEPTIOSTRUCT* pointer to true. This indicates to the simulator that your analyzer aborted its implementation of the I/O access.
ANALYZER_IN_MONITORMEM	Generated when a memory access is occurring. An *INMEMORYIOSTRUCT* pointer is passed in *pParam1*.
ANALYZER_IN_INTERCEPTMEM	Generated when a memory access is occurring. This intercept allows the analyzer to override the result of the memory access. An *ININTERCEPTMEMSTRUCT* pointer is passed in pParam1. If your analyzer aborts the memory intercept; i.e., your handler does not handle the memory access, you must set the *bAbort* field of the *ININTERCEPTMEMSTRUCT* pointer to true. This indicates to the simulator that your analyzer aborted its implementation of the memory access.
ANALYZER_IN_RESET	Generated when the processor model is being reset. This includes warm reset, cold reset, and the user resetting the BSD.
ANALYZER_IN_EVENT	Generated when an event initialized by your analyzer by calling CAweSim::NotifyLater() fires.
ANALYZER_IN_POWER	Generated when a low-power state or ACPI sleep state is entered.

Table 1-1: Intercepts