

Slide 1



Slide 2



So today, we're going to talk about MicroBlaze 4.0 and exactly what it is. We'll also be talking about some of the new features available from Xilinx Platform Studio, also known -- we'll refer to that as "XPS 7.1" recently shipped.

We'll also learn how to develop a virtual platform for MicroBlaze in just a couple of minutes. We'll do this in about 45 minutes. As a matter of fact, you could probably do this at your desk in about 15 minutes; it's actually quite a simple system. And we'll go through the various steps to do that. We'll be using XPS to actually define and build the system, which is Xilinx Platform Studio.

The results of a secondary tool, an Eclipse-Based Platform Studio SDK. We'll be using that actually to do a lot of our compiling and debugging and analyzing our software.


And at the end, we'll actually use that tool to compare the performance of a couple of different implementations of our

design, where we trade off some of the opportunities to add additional hardware to design and improve the performance of our software.

Slide 3

Custom Processing On FPGA

- Xilinx Embedded Solutions — Get the exact processor you want
 - 8 bit or 32 bit
 - Hard or soft
 - PicoBlaze (8-bit), MicroBlaze (32-bit), PowerPC 405 (32-bit)
 - Use only the peripherals and configurations you want
 - Avoid obsolescence
- MicroBlaze — Custom Processing Made Easy
 - Ready to use, prebuilt configuration options
 - Performance when you need it — match applications' compute needs
- Introducing MicroBlaze v4.00
 - Answering the demand for floating-point support
 - Ease of use — common development tasks improved
 - More compute performance for growing application sizes



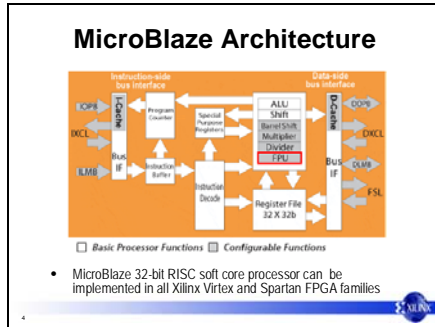
There are number of processors available from Xilinx. We'll be talking about MicroBlaze today, which is a 32-Bit Soft Core Processor. However, there are a number of other processors as well. We have a Power PC Hard Core, PowerPC 405 available in our Virtex families. There's also an 8-bit Microcontroller "PicoBlaze" available; the point here is that you could choose a processor tailored to the type of application you might be running.

The difference between a soft processor and a hard processor, we might want to point out for those not familiar with our FPGAs, is a hard processor is actually implemented in the FPGA at a transistor level. It's just like something you would buy off the shelf at your local electronics or store. However, a soft core is actually an IP block, a real design in an HDL that you would implement in the available resources of an FPGA. Our soft cores consist of MicroBlaze and PicoBlaze. PowerPC is our hard

core.

MicroBlaze, is a ready-to-use, out-of-the-box soft core implementation. One of the advantages of soft-core designs is that they are very configurable. In other words, you do not have to incorporate every feature that's available. So if you're not using a particular feature, you can actually disable it, and it won't actually be included in your design; therefore it will not utilize any of the resources in your device. So you can actually scale the implementation to your application or custom-tailor your processor.

MicroBlaze 4.0 actually builds on the three previous versions, all based on the same instruction set. And we'll be talking a little bit about some of the new floating-point features that have been added to this device to add capability for customers utilizing floating-point units. It also gives you the advantage of being able to actually implement a reference design that might have been a floating-point reference, include that, and actually then do a fixed-point implementation and actually be able to compare the results of your floating-point reference and your fixed-point running simultaneously on the same processor.




So we want to take a look at the MicroBlaze architecture, you'll see it's a traditional hardware architecture. We do have a number of interfaces available. This is one of the very flexible features of MicroBlaze. You do have what's called an on-chip, peripheral bus, or an OPB bus. This is a core-connect IBM standard bus. This gives us the capability to connect a wide variety of available IP blocks and peripherals out there on the market and also available from Xilinx to that bus. It also gives us the capability of connecting directly to a PowerPC. Some of our devices have PowerPC in them. You could then also then slave MicroBlaze off of an OPB bus, for example, and have your PowerPC maybe being a master on that system.

Some additional capabilities available: You'll see a Fast Simplex link (FSL) on the right side. This is a new interface. It's a FIFO-base interface, and it is there specifically to allow users to define custom pieces of hardware that might assist in a particular application. We have reference designs you might have seen previously, or even look at on our Web site, the IDCT or FFT applications using this interface. Any kind of DSP, bit-extraction, packet-processing types of algorithms that might run efficiently on a standard processor, you can actually define some custom hardware and bolt it directly to the processor via the FSL port. It is

Slide 5

MicroBlaze v4.00 Highlights

- Backward compatible with MicroBlaze v3.00
- New features
 - Floating-point unit (FPU)
 - IEEE754-compatible, single-precision floating point
 - FP instructions part of ISA
 - Fully supported by compiler, tools, and instruction-set simulator (ISS)
 - Pattern-compare instructions for optimized comparisons
 - Three new compare instructions
 - Augments the rich set of branch and compare instructions
 - Configurable hardware multiplier
 - Cost-minded customers can disable inclusion to save gates
 - Enhanced debug logic for faster download
 - Higher maximum clock frequency
 - Fewer cycles per instruction (CPI)



a very low-latency, high-speed interface.

Another interface that's now available is our Xilinx Cache Link (XCL) interface, both for data and instruction. As we'll learn, if you take a look again at the Web site, this is a new way of connecting a high-speed memory interface. It's an efficient port for DMA support.

So, again, the point being here is there's quite a number of opportunities to connect to this device. It just depends on what kind of a system you might be architecting and what kind of performance you might need.

Let's talk about some of the new features for MicroBlaze. One of the newest features for MicroBlaze is the floating-point unit. It is an IEEE754-compatible, single-precision floating point unit. We also added to the instruction set the necessary instructions to utilize that floating-point unit.

There is also a new pattern-compare instruction which, basically, augments some of the existing pattern-matching OP codes that are available. We've also made the hardware multiplier, that has been available, a configurable option; so include, or disable that from your design if you do not need a hardware multiplier -- again, saving FPGA resources.


There has been an enhancement to the debug interface module, which now allows us to actually have complete access to the core pipeline. We've also done some

floor-planning and brought up the overall maximum frequency and we've also added some new instructions that actually, reduced the number of cycles per instruction.

Slide 6

MicroBlaze Array of Configurable Features

- Floating-Point Unit
 - IEEE-754 compatible
 - Single precision
- Hardware Exception Support
 - Unaligned access
 - Illegal instruction
 - Data bus error
 - Instruction bus error
 - Divide-by-zero
 - Floating point exceptions
- Fast Simplex Link Co-Processor Interface
 - Direct access to the general purpose registers for hardware acceleration
 - Up to 8 dedicated 32-bit input ports
 - Up to 8 dedicated 32-bit output ports
- Instruction and Data Caches
 - Uses on-chip block RAM primitives
 - Configurable size: 2Kb - 64Kb
 - Configurable cacheable range
 - Direct mapped write-through operation
 - Caches over OPB (1 word cache line) or CachotLink (4 word cache line)



If you take a look at some of the existing features, you can see there is a broad range of configurable options. Not only is a floating-point unit something we can include or not include, but everything from hardware-exception support and also individual types of exceptions. Whether you want to trap illegal instructions or a bus error, a divide-by-zero error, these can be individually included, or not included, in your design.

You have new Fast Simplex Link. There are up to 8-bi-directional ports available there, so if you're architecting some hardware, you can have eight additional instruction units added to your processor and a fast high-/low-latency interface. These FSL link can also operate as uni-directional links as well. So if you happen to be sampling data or just outputting something to another processor or device on the board, they can operate independently at 16 individual channels.


We've got a configurable cache size -- again, up to 64-

kilobytes. Now you can configure your cache sizes, whether they're mapped directly or as a Write through; again, a wide variety of opportunities, again, to configure this core.

Slide 7

MicroBlaze Configurable Features Continued

- System Interface
 - Different combinations of OPB, LMB, XCL, and FSL for flexible system design
- Barrel Shifter
 - 2 cycle operation
- Hardware Integer Divide
 - Hardware Multiply
 - 3 cycle operation
- Debug Logic
 - JTAG control via a debug support core
 - Up to 8 hardware break points
 - Up to 4 read address watch points
 - Up to 4 write address watch points
- Instruction Set Extensions
 - Pattern Compare Instructions
 - Machine Status Register Set and Clear
- Interrupt Signaling
 - Edge or level
 - Active high or low




From the system interface, we've talked some about some of the buses that are available, basically four: local-memory bus, OPB bus, the XCL bus and FSL. There is a 2-cycle barrel shifter available. There is also a hardware divide capability included -- all these being optional.

Again, we can read down the list here. There are quite a few configurable features available.

Slide 8

Xilinx Platform Studio (XPS) 7.1i

- XPS enables the creation of the hardware for our CPU and peripherals as well as the software that run on the system
- New support for the generation of virtual system models using the Virtual Platform Generator
- Xilinx Platform Studio SDK now includes a new graphical profiling capability
- Support for the new Virtex-4 FPGA family.
- Introduces a new peripheral import wizard



If you want to develop for a Xilinx microprocessor, you will need Xilinx Platform Studio. Xilinx Platform Studio actually is a union of a number of different tools and we'll break that down in a moment.

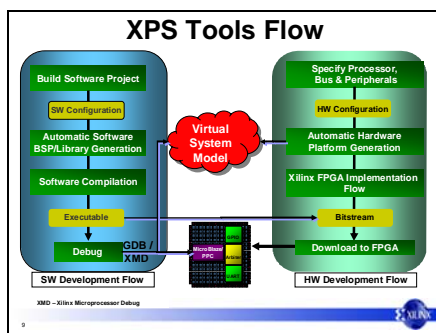
Some of the new features of Xilinx Platform Studio is the capability to generate a virtual platform, with the Virtual Platform Generator. So as you architect a system, you're no longer have to prototype on a hardware development board somewhere at your desk, or even have to go through the flow

of developing that hardware. With Virtual Platform Generator, you can now generate a software model of your system and then compile, profile, execute software, on that model.

The Xilinx Platform Studio SDK, or Software Developer Kit, also has some new features for graphical profiling that we'll take a look at as well.

7.1 also adds support for the latest generation of the Virtex family, the Virtex-4 family, and also introduces some new peripheral import wizards. These are, actually, quite remarkable in that you can now define your own hardware and using an import wizard, create wrappers around that hardware. That would then create the bus interface for the variety of different buses we have. We won't actually be running that today but I encourage you to take a look and download the software. It's a very useful feature.

Slide 9



So lets take a look at what Xilinx Platform Studio can do for you: It's actually running two different traditional flows. Unlike your discrete processor, that you buy off the shelf, you can choose the peripherals you want in your system. We're actually creating a custom processor here and we're implementing it in an FPGA.

So the first thing we're going to have to do here is define what features of the processor we'd like to implement and, also attach the various peripherals to the processor

buses that we would like in our system.

XPS has to help us define a system architecture. You also have, on the right-hand side of this diagram, a full suite of tools for implementing an FPGA. Running HDL synthesis, mapping to an architecture, place-and-route tools -- all the traditional tools for a hardware-implementation system are also encapsulated under Xilinx Platform Studio.

On the left-hand side of this diagram, you can see the traditional software flow. So, obviously, if we're developing software, we need a compiler and linker to generate an executable and we need debug tools. And Xilinx Platform Studio will encapsulate that as well.


Xilinx Platform Studio Eclipse-Based SDK will do that as well, and we'll be looking at the latter in the presentation.

So let's walk through the process of creating a MicroBlaze System with XPS.

Slide 10

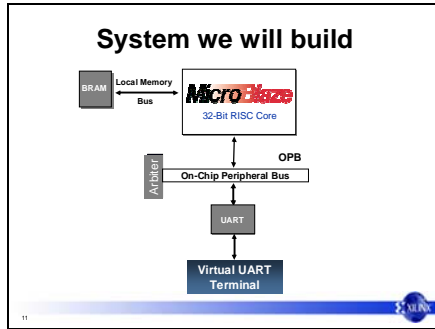
**Learn How to Build a
MicroBlaze System**

- Use XPS base system build wizard to build a custom virtual hardware platform.
- Use the Eclipse based Platform Studio SDK tools to compile, debug, and profile a software application
- Compare performance of a floating point FIR filter application as we change the available features in the MicroBlaze processor



10

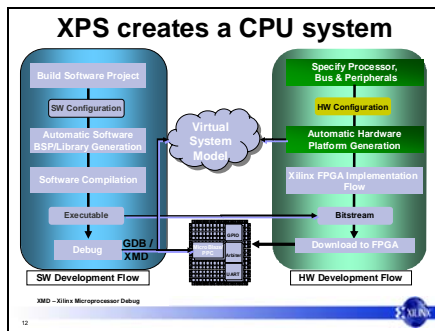
Slide 11



Now, if we want to build a system, we have to decide what we're going to build. For, today, we've got a limited amount of time, so we're going to build a very simple system. We're going to instantiate MicroBlaze. We're going to give it a local memory, and we're also going to instantiate a UART or an RS232 or giving us a terminal to view printf() results out of our software program.

And, in this case, what might have been running on hardware is a UART connected to a hyper-terminal on our Windows system through a UART cable. We are going to connect directly to a UART virtual terminal. One of the nice things in the tool system is they will recognize when you have an I/O module within your device and give you access, via a virtual model, to those inputs/outputs and terminals.

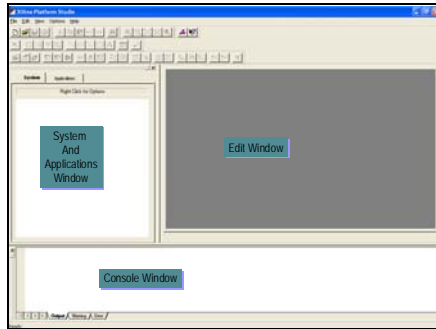
Slide 12



So if we go back to our diagram, we're going to implement the first three steps on the right-hand panel.

We're going to specify our processor and interfaces, and we're going to configure the hardware peripherals. Then and we're going to generate virtual models for all the necessary pieces.

Slide 13



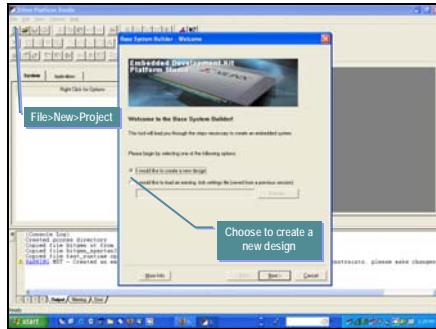
The screen you're looking at is Xilinx Platform Studio. On the right side, you have an edit window for the various text files that you'll be viewing. It's HTML-capable and it does color-context highlighting for various file formats.

You have a console window along the bottom. This will allow you to view all of the various tools being launched behind the scenes and the results from those tools -- and all your status and warning messages would be visible there.

On the left-hand panel, we've actually got two tabs. These are our "system" and "application window." When a system tab is highlighted, you're actually looking at the hardware architecture that you've created, and all the various peripherals and the processor. When the application tab is selected, you're looking at your C-source code and more of a software view of the world.

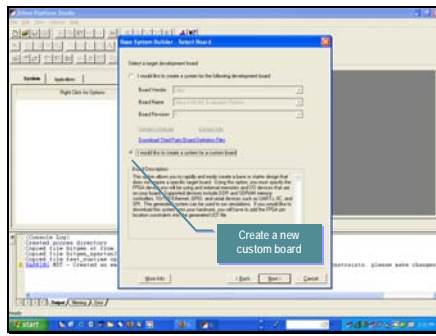
Today, we'll actually be building a system in this Tool; however, when we get to the applications side, we're going to move to the Xilinx Platform Studio SDK Tool System.

Slide 14



So, from this tool, all we need to do is start building a project. And Step 1 would be to go to the File Menu and just choose "File>New>Project" and we get an immediate window that will ask us, do we want to create a brand new design. We'll click that radio button for this option and select "Next"

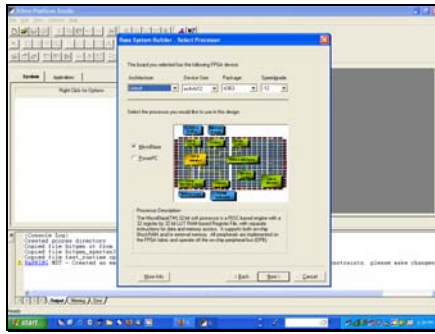
Slide 15



In the next window we have an opportunity here to target an existing development board. And one of the nice features of the System Builder is that it is completely aware of all of the available Xilinx development board vendors, all of our partners, and the list of boards available will show up in the upper tabs of this menu. So I can choose a particular board, it would understand all of the I/O capabilities, all of the functions supported on that and, also, all of the device pin assignments. It, basically, allows you to build a complete board. For any of you that have ever had to bring up a board, it's not a trivial matter. This basically makes it a couple of click-box options and you have a working board. We're actually going to choose the second option today. We're going to create something custom. Since we're actually building a virtual model, there is no need to have hardware involved. So we're just going to bypass that step and move

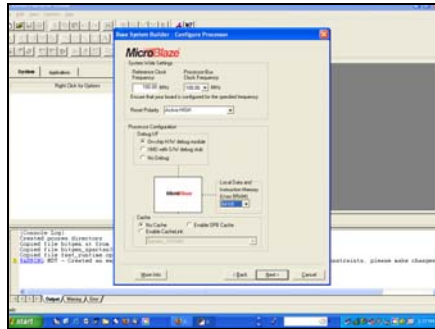
forward.
Choose Create new custom board and select "Next"

Slide 16



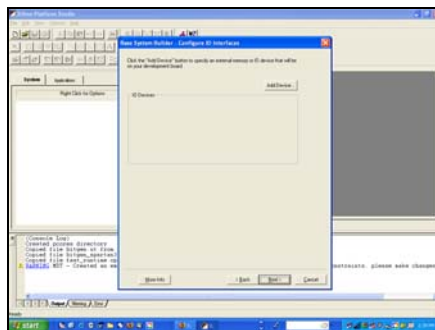
The next tab brings up an option to choose a processor and also a device and family architecture. In this case, the architecture doesn't particularly matter since we're going to do a virtual model of this system. However, we do have a Virtex-4 targeted. You'll notice, with Virtex-4 being targeted, we have the option of two different processors. There is a hard-core PowerPC in these devices -- sometimes, more than one. We can also implement MicroBlaze as well in this designs. We could easily have a multi-processor system here, but this gives us a starting point with MicroBlaze. So we choose MicroBlaze for this system. Select "Next"

Slide 17



Our next window is going to ask us to configure our processor. At this point, if we're building a board, we would input some information about our frequency and how we'd like our resets implemented -- some of the hardware details of a real board. And, again, these would come up with default settings for you. If you were targeting a particular board, then you would have, then, the option to change those options. The important thing here is, we're actually going to use an on-chip debugging module, and we're also going to change our default local memory size up to 64K, so we can run just about any of our test applications here. And, again, all we have to go through is "Next" on our menu system.

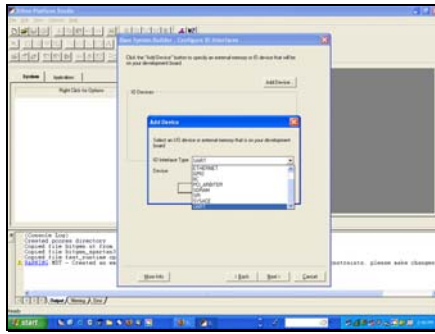
Slide 18



Then we get to a tab that allows us to add some additional devices to our system. By clicking the button in the upper right-hand side, you'd be able to add a device to your system. And, in this case, we want to add a UART, so we choose from our drop-down list -- you can see there's a variety of things available on the list here. Simply selecting these, in turn, adds them to your system, automatically generates memory maps and wires up all the necessary system components to make these things active in your system. Again, you don't have to get into editing any individual wires; the system takes care of that for you. So we choose our UART

and move forward.

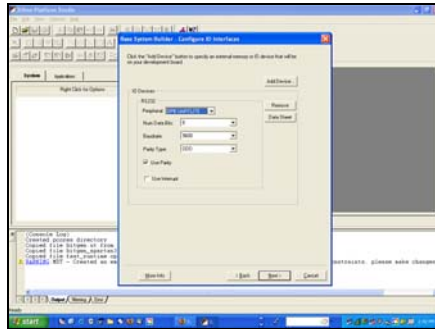
Slide 19



Then we get to a tab that allows us to add some additional devices to our system. By clicking the button in the upper right-hand side, you'd be able to add a device to your system. And, in this case, we want to add a UART, so we choose from our drop-down list -- you can see there are a variety of things available on the list here. Simply selecting these adds them to your system, automatically generating memory maps and wiring up all the necessary system components to make these peripheral active in your system. Again, you don't have to get into editing any individual wires; the system takes care of that for you.

So we choose our UART and move forward.

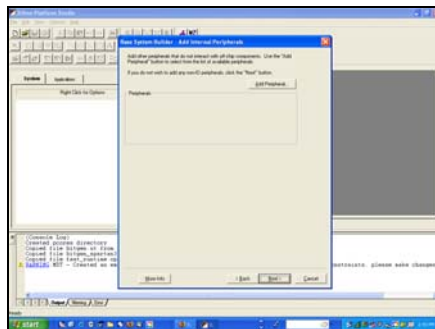
Slide 20



We, then, get an opportunity to configure the UART; again, parity, baud-rate and, again, we'll accept the default here; we can choose any range of baud rates up to 115K. You can opt to use interrupts, if your system needs them.

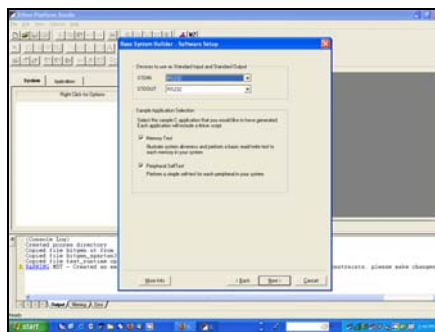
We won't be using interrupts so we leave this option unchecked. Let's move on by selecting "Next"

Slide 21



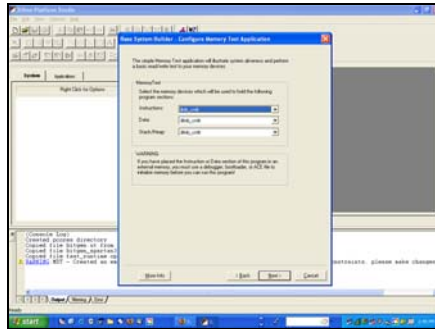
We then come to another tab that asks us if we want to add an additional peripheral to the system. This is the only peripheral we're adding, so we're going to move forward. Select "Next"

Slide 22



We, then, get asked, where we want the standard-in and standard-out information from our C-application to go. The main reason we added a UART is so that we had a terminal here to view printf() output in our system. So we're going to ask that the standard-in/standard-out be sent to the RS232 port here. And, again, once that's configured, we move forward.

Slide 23

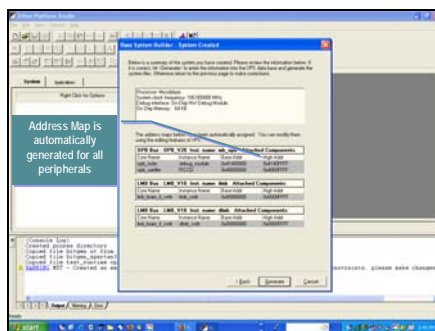


We are next asked where would we actually like our instructions and our data stored, how do we want to organize our memory. Since we're only utilizing a local memory, at this point, we'll have our instructions in the instruction side, local memory and data in the data-side local memory; and, again, our stack would also be in the data side local memory.

If we add an external memory interface to our system, we would be given an option to place our program sections in external memory.

We chose "next" to continue.

Slide 24



On the next panel, you'll see the system has been created and you get a list of the peripherals and also their memory-mapped addresses.

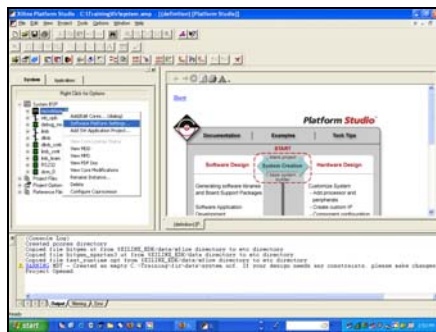
So you can see here, we have a couple of local memory interfaces. We also then have our UART, and you can see the addresses at which these peripherals are located. These address maps are automatically generated; however, they are editable, and over-writeable, once we go back to the tool system. So if you don't like those particular settings, you can change them. You also have the options to lock some address assignments and have the tools automatically generate the addresses for those peripherals that remained unlocked.

We chose generate to build our system.

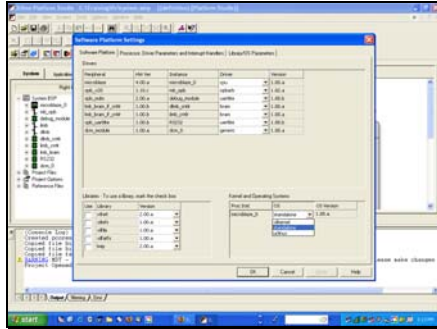
links to the Document System. So if you have any questions on where to go from here, you can look at these various steps and click on those links and it will take you to the appropriate Documentation Sections that will describe how to process those sections. So, again, a nice time-saving feature.

Now, some of the options we have here is we can actually go in and modify some of our software settings. So simply selecting "MicroBlaze" with a right click would bring up a menu that would allow quite a number of different things. We can actually view the documentation specifically for that core. So, again, right-clicking on any of our peripheral objects allows us to go straight into the documentation system.

Slide 27



In this case, we're actually going to take a look at the software platform settings. So we right click on the MicroBlaze CPU instance and choose "Software Platform Settings"



There are quite a number of options available here again. We're going to accept most of the defaults, but if you take a look at the upper half of the panel that comes up, you can see all of our peripherals and the drivers associated with each.

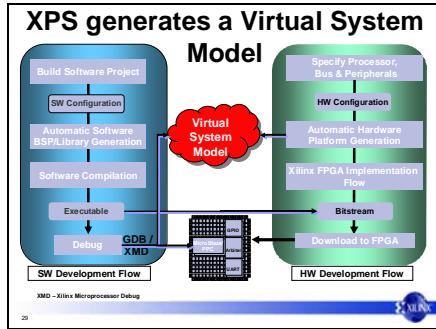
At any point, if we would like to change the drivers or add our own, we can go into these menus and point the peripherals to "New Drivers" or "Customized Drivers" or just different versions of drivers, if we happen to want to rev. backwards for a particular reason.

On the bottom half of the panel, you'll also see there are quite a number of libraries available. Everything from a networking support to a light-weight IP block, fat file systems - - all of these things are available, simply at a click of a button. They would then be linked into your system so that as the compilers are building you application, you would have access to all these features.

On the right-hand, bottom side, you can also see that we can even target a stand-alone OS, a simple Xilinx kernel, or even something as complex [sic] as a uC-Linux implementation are available at a click of a button.

We'll leave these settings as they are and select "Cancel"

Slide 29



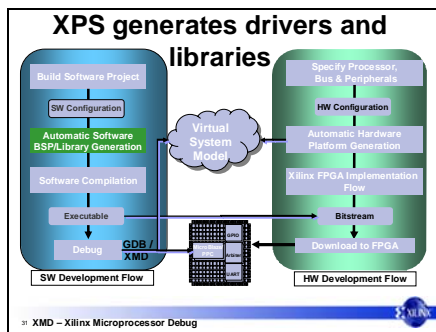
So now we've actually defined a system architecture. What we need to do now is actually generate a virtual platform software model and also generate the libraries for that system. So let see how we do that.

Slide 30



And, again, to do that, we select from the Tools Menu and we choose the "Generate Virtual Platform" option and that will, then, kick off the Virtual Platform Generator that would then take a look at all the various components in our system and generate C-models for them.

Slide 31



Now once we have the model for our system implementation, there is one more step. We have generate all the necessary drivers for the peripherals that we installed and we also have to generate the libraries that we've selected.

Slide 32

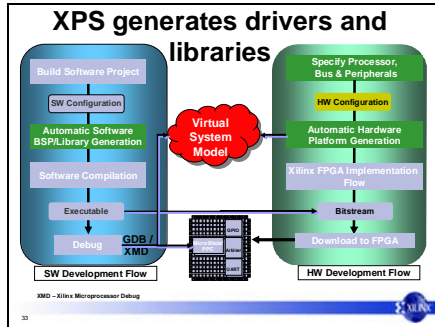


And to do that, again, we go back to the Platform Studio Tool and we, again, choose from the Tools Menu, and our first option is a "generate libraries and BSPs."

One nice thing about this tool is that it is completely make-file-driven. This entire flow is encapsulated in a make-file, so, at any given point, you can simply select the output you need and XPS will launch only those tools needed to complete your request.

For example, if I actually want to generate a hardware bitstream and download it to an FPGA, I can simply select the "Download" option and the tools will, via the make-file, understand that all the various sources may not have been processed yet, then go back and, actually, fire up all the required tools to generate all of the individual components -- all the way down to generating and downloading a bitstream. So, at any point, if we added or changed things, the make-file system automatically regenerates the necessary files, so we don't have to go back and build everything from scratch at each, each time we change things. This is a very, very nice time-saving feature.

Slide 33



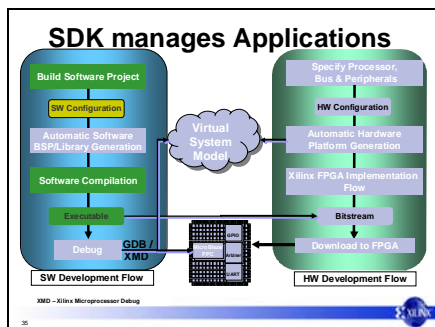
OK, at this point, we now have the libraries and the virtual model generated.

Slide 34



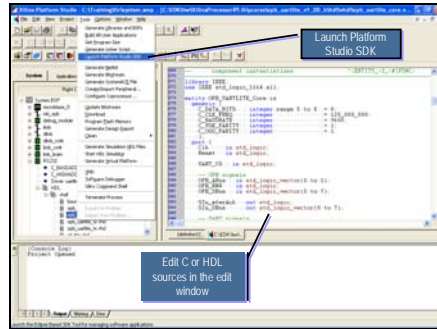
So now we move on to the next step. We now have specified our hardware processor system. We've got a virtual model and we've got the software libraries generated. So now we're going to move off into the Eclipse-based Xilinx Platform Studio SDK Tools where we're going to start setting up a project and start compiling and profiling this application.

Slide 35



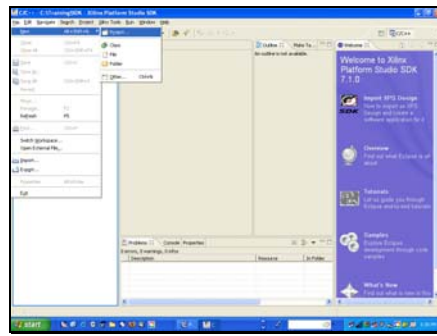
So if we go back again to our diagram, the various tasks we're going to be performing are here on the left-hand side, mostly the software-related tasks. So we're going to generate a project, set up our software and that would then create a tool chain and, then, we'll generate an executable from that.

Slide 36



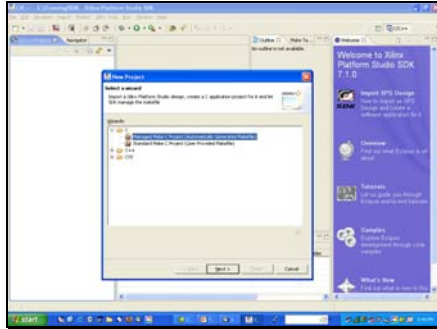
We can launch Xilinx Platform Studio SDK from within XPS. Simply select the Tools -> Launch Platform Studio SDK. We also have an example here, on the right-hand side, of our Edit window. Whether you're editing C or HDL, again, you can see that everything's color-highlighted for you.

Slide 37



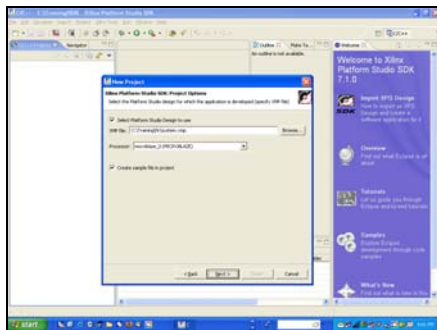
Now, we're looking at Eclipse. And to begin in Eclipse, we have to create a software project. Eclipse is a very unique tool and powerful tool in that it has the capability of multiple perspectives. So, the tool can take on a number of different "looks", if you will. You have a perspective for managing C Projects, you have a perspective for profiling, you can also have perspectives for various runs and also a debugger view. So, the tool can take on the look and feel of familiar applications, and it allows us to instantiate the Xilinx tool system underneath this nice interface. So, to create a project, we're going to go: File -> New Project

Slide 38



We are going to choose to have what's called a "Managed Makefile System." You have the option if you're very familiar with make-files and you have your own preferences for how those are constructed. You can manage those manually; that would be the second option. For those of us like myself who are more of a hardware-oriented person -- I don't necessarily understand make-files tremendously well -- we'll have those things managed for us. So the Tool will create the makefile and manage it for us as we go through the project changes. Select "Next"

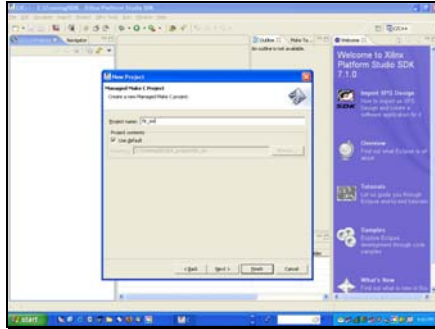
Slide 39



To start a project, we have to point back to the platform we just created. We have an underlying hardware architecture and software models that we've just created; so we have to point our project back to that platform, so that everything can be built on that platform.

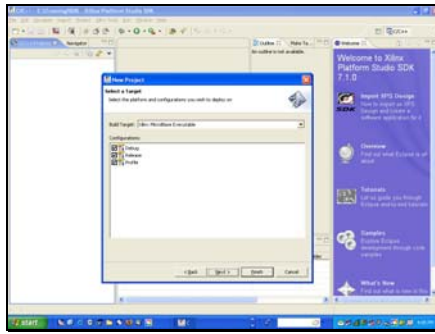
The way we do that is through our "New Project" Menu here where we simply just give it a path back to the system.xmp file which, is created by Xilinx Platform Studio, the Xilinx Microprocessor Description File. And, once you choose that file the tools know which processor you selected, and how your processor system is configured. Select the "Next" button?

Slide 40



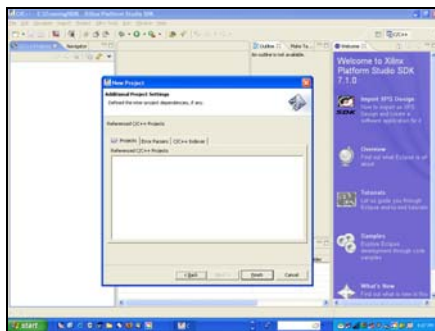
The next thing we do is give the software project a name. In this case, we're going to give it the name "fir_SW"
We select "Next"

Slide 41



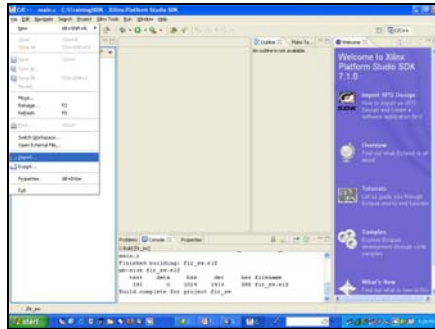
By default, you get three different configurations options available when you launch, a debug configuration, a release configuration and a profile configuration. Configurations, you might think of as just a recipe for building your executable, so these may be the three different ways that we build an executable. We'll see a little bit more about how we might use that feature later.
Select "Next"

Slide 42



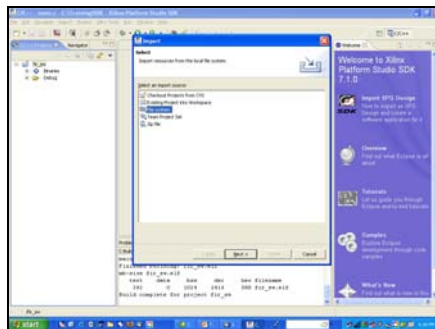
Now we get an opportunity to add some additional features which we're not going to do at this point, and we will go directly to our application.
Select "Finish"

Slide 43



Now we want to add some C source code to our project., so we're going to import into this project now.
To do that, we choose "File -> Import."

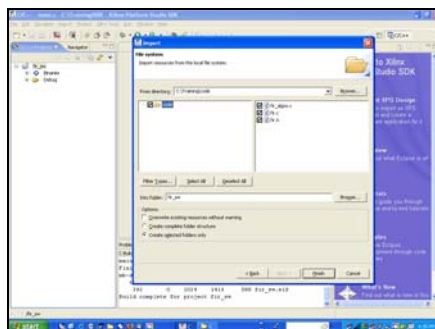
Slide 44



This brings up an import menu, and we can choose to import from a variety of places. Eclipse happens to be a completely CVS-capable tool set, so we could pull things out of a working CVS system. We could pull it from a ZIP file.

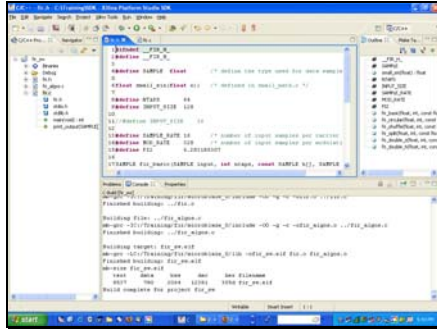
In our case, we're actually just going to pull it from the hard drive, so we're going to choose "File System."
Select "Next"

Slide 45



And once we have that done, we just supply a path to where our source code is and the tools will automatically recognize any file that happened to be, you know, source files. By simply check-boxing the file that we would like included, we, then, import those into the project.
Check of the three source files and select "Finish"

Slide 46



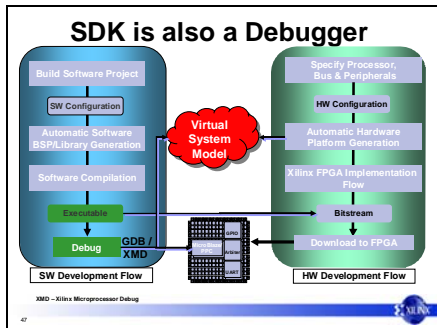
So now we have a C-Project in Eclipse, or a Xilinx Platform Studio SDK. On the left-hand panel, you can see the C++ Project perspective; the tab is highlighted at the top. You also have a navigator tab on the left; that's another perspective for that window; it allows you to do file management.

This left panel is your C-Project. And you'll notice that there are folders, one called "Debug," one for each of our various configurations. Debug happens to be the only active configuration at this point; so that's where we would find our executable.

In the center panel, we have our Edit window and C-program with color-coded keywords and, on the right-hand panel, we have all of our function calls that we link to with a simple click. And, at the bottom, we have our console window.

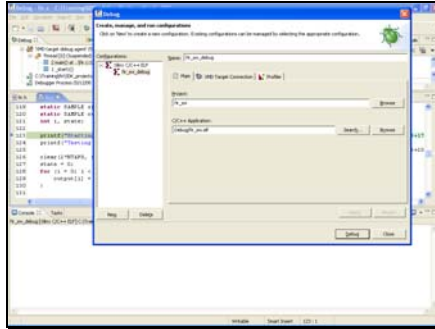
You can see that the project was automatically compiled and an ELF file was created.

Slide 47



OK. So now we have a software project built. The next thing we want to be able to do is -- we've just generated an executable -- we'd actually like to be able to run that executable. We can either do a run or we can actually do debug session. In this case, we're just going to step through, into the debugger and take a look at what that might look like.

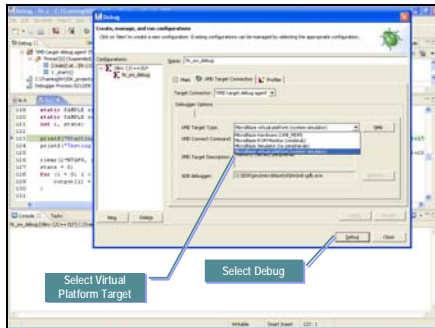
Slide 48



The first thing we need to do is set up a new configuration for launching the debug session. Select Run > Debug from the main menu to bring up the Create, Manage, and Run configuration window.

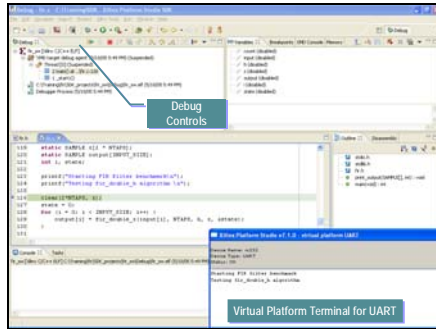
In this window, we have an option of selecting "New" to create a new configuration, and we can have multiple debug configurations, each targeting different boards or even targeting our virtual model. I'm basically telling the tool where my executable is, and that's done through the C application box -- you can see I'm pointing to the ELF file for this project -- in the debug folder.

Slide 49



If I click the center tab, the XMD target connection -- "XMD" happens to be the Xilinx Microprocessor Debugger interface -- and you can see that I can connect to a number of different targets.

I can actually connect to a debug board that I might have sitting next to my PC. I can also connect to this virtual platform model, we are going to connect to a virtual platform target; so I choose that as my target option. This, then, will launch the debugger and have it connect via a socket to a software model. To begin our debug session select "Debug" and move forward.

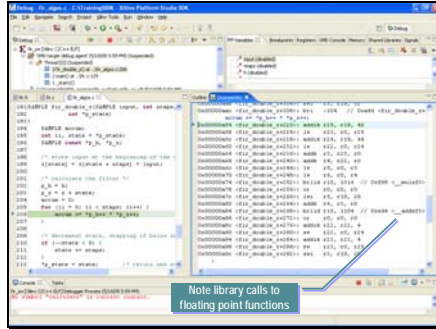


This is the perspective for the debugger. Again, we mentioned that the perspective changes depending on what your task is within the tool. We happen to be in the debugging perspective and in the upper-left window, you'll see above that window a variety of different debug options. I can run, I can stop, I can single-step, I can step into and out of functions -- again, all of the normal things I would expect from a debugger. These are standard features within Eclipse.

On the right-hand panel, we can see all of our variables, we can change tabs to breakpoints, we can look at memory contents here. Since we don't have a live tool, we won't be stepping through all of these options. What we wanted to make available was that the debug perspective is something you can use and you can single-step through and implement your code.

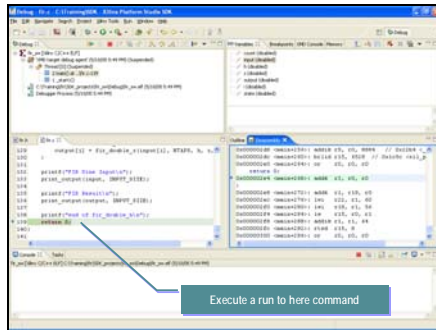
In the bottom part of the panel, you can see actually the Xilinx Platform Studio Virtual UART terminal popped up and since we're single-stepping through this code here, as we get to each individual line, you can see the `printf()` statements that have been dumped to our terminal window.

Slide 51



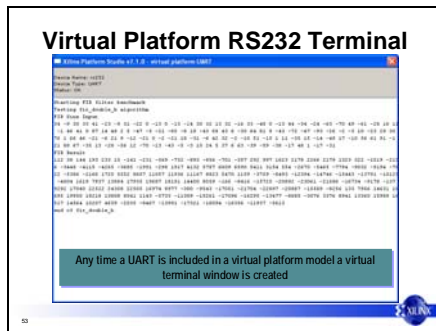
Selecting a line in the Disassembly window automatically highlights the corresponding line in the C source file and visa versa. Highlighted in the left panel we can see the core arithmetic loop function that implements our We can see that this small loop of code make calls to floating point library functions muldf3 and addsf3. These functions are large and consume a large part of the CPU time in the FIR function call as we will see in the following section on profiling.

Slide 52



By Selecting the “return 0;” line in the left C source window we can then right click our mouse and choose to run the program to this line. This will run our program to the end so we can see our results.

Slide 53

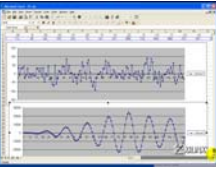


We can see the virtual terminal now displays the input and output data for our filter.

Slide 54

Does the Data Look Correct?

- We paste the data in MS Excel and graph.
- Input is a sign wave with noise superimposed.
- Output looks like a clean sign wave.
- Looks like this filter is functioning correctly!



54

Cutting and pasting this data from the terminal window into an Excel spreadsheet lets us quickly see our results graphically. Indeed, everything looks correct.

Slide 55

Learn How to Build a MicroBlaze System

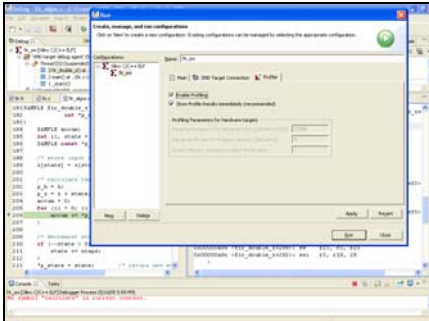
- Use XPS base system build wizard to build a custom virtual hardware platform.
- Use the Eclipse based Platform Studio SDK tools to compile, debug, and profile a software application
- Compare performance of a floating point FIR filter application as we change the available features in the MicroBlaze processor



55

Next lets look at how we might improve the performance of our FIR filter by enabling some extra features in our MicroBlaze CPU core.

Slide 56



56

We begin by creating a new run configuration by selecting the run option from the 'Run' menu. (Run > Run)

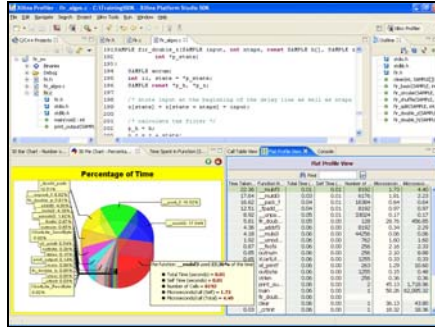
Clicking on the new button we can give this new configuration a new name 'fir_sw'

Our project name will still be fir_sw and our application target is Debug/fir_sw.elf

Select the XMD Target Connection tab and enable profiling by check the two boxes found there.

Select 'Run'

Slide 57

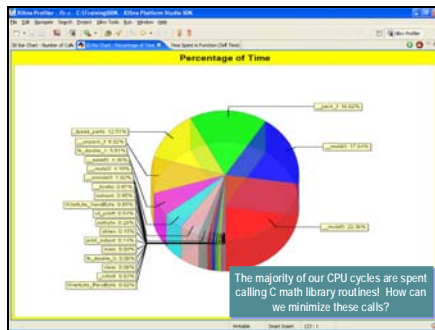


As the application runs profiling data is accumulated and displayed upon completion of the run.

The profiling data is displayed in two panels. A graphical panel is on the left and a spreadsheet version is on the right. Each panel has multiple tabs so we can view the data in a hierarchical fashion or a flat format.

Note the “time taken” column in the Flat Profile View tab on the right panel. Over 50% of the CPU time is spent calling floating point libraries from the standard C library set. These libraries are not small. We can also see this graphically on the left panel.

Slide 58

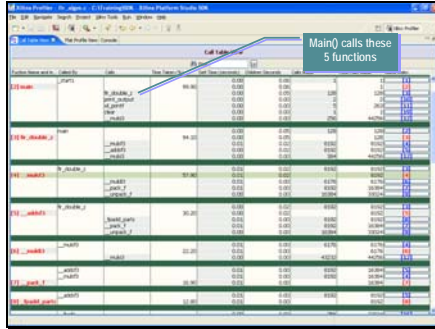


The important thing to see here are the functions taking the majority of the time for the CPU are these function calls to our floating-point library. How can avoid calling floating point libraries?

If our processor had a floating point unit it would be able to do these computations natively, thus avoiding library calls.

We can easily see here the 5 most time consuming functions are all library calls.

Slide 59

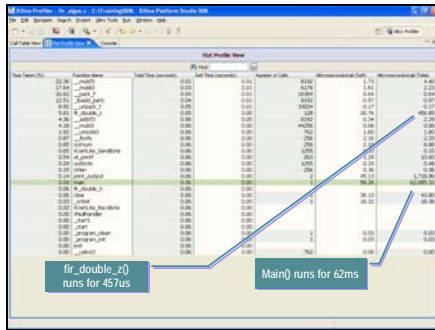


If we look at the hierarchical view of the spreadsheet panel we can see that each function is listed with function it calls grouped together.

Here we can see the `fir_double_z()` function takes 94.10 percent of our CPU time and `__mulsf3()` consumes 57 percent of the time in `fir_double_z()`.

A nice feature of this view is that the far right column contains hyperlinks to sections of the document pertaining to each function. This helps up quickly find the data that matters to us.

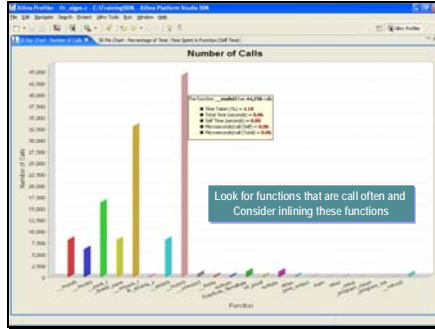
Slide 60



Note `Main()` takes 62 ms to run here and `fir_double_z` takes 457us. This is without a FPU unit.

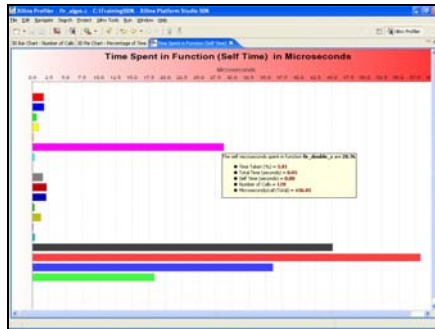
We'll compare these numbers to our new profile after making some modifications to the CPU.

Slide 61



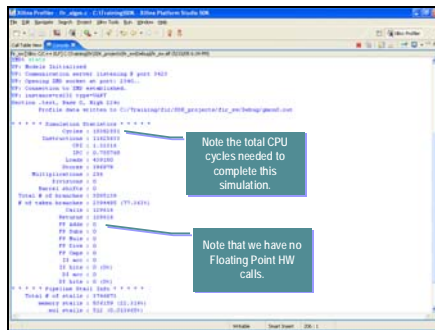
Functions that are called most often are candidates for inlining. While they may not account for a large number of CPU cycles, they may cause unneeded stack overhead. The Call Graph tab gives us this data instantly.

Slide 62



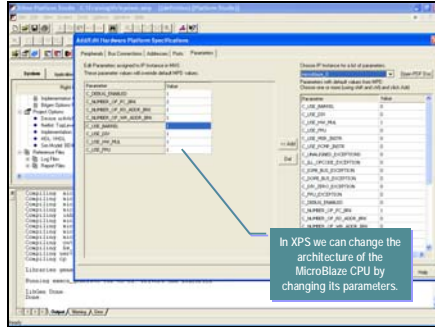
We can also see the total time spent in a function in microseconds reported as well. This would help us decide which function to optimize first. These could be a candidate for a hardware assist engine attached to the processor. This would offload this function to hardware freeing up needed CPU cycles.

Slide 63



If we type 'stats' in the console window we get the statistics from our profile. In this case, it's taking about 15 million cycles, and you'll also note below that the processor is not making any calls to the floating point unit. This confirms that we don't have a floating-point unit included in our processor.

Slide 64



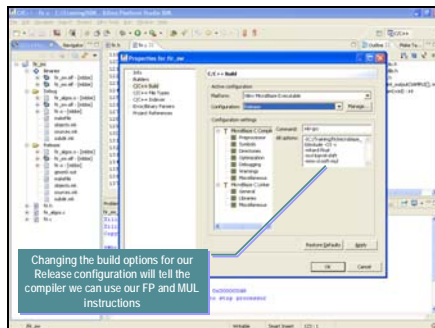
Now, at this point, I've got two options to add a floating point unit to this design.

Number one, I can go back to the Xilinx Platform Studio System and simply bring up the Add/Edit Core dialog as seen here.

I can change the parameters on MicroBlaze. And you can see in this window here, there's a number of options on the right-hand side and if we look at the left-hand side, we can see the actual parameters that we might be overriding or changing. So we can accept the defaults or we can change them. By default the C_USE_FPU parameter was '0'. We'll can change this to a '1' and recompile the design which would include the FPU in the next CPU build process.

In this case, also added support for the barrel shifter, divider, hardware multiplier and a floating-point unit.

Slide 65

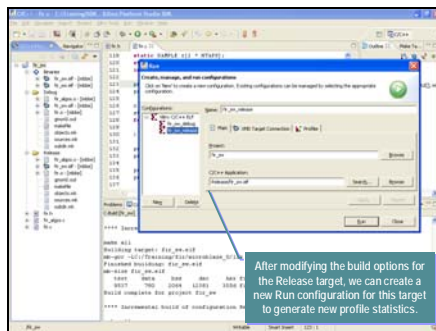


Now, in our instance, we can take advantage of the fact that we generated a software model of this system and we can realize that the ISS will model the FPU, Barrel Shifter, and Multiplier by simply supplying a compiler directive.

So what we're going to do is we're going to go and change the properties for our build and, in this case, we can change the configuration we've targeted. Previously, we were targeting the debug configuration, now we are targeting the "Release" build configuration as you can see in the upper half of this menu here. For the "Release" configuration

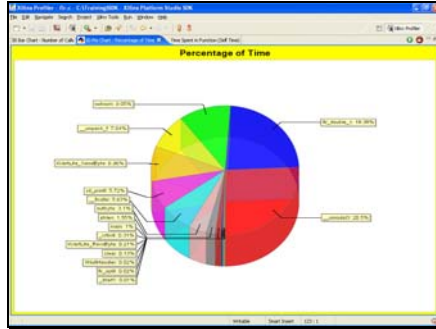
we add some command-line options for the compiler, so that it knows that it can generate those hardware instructions specific to the multiplier and the divider and also the instructions specific to the floating-point unit. We don't actually have to go back and rebuild anything. This methodology allows us to quickly create a number of different build configurations with differing architectural features included or cache size changes and simultaneously run profiles on many different processor configurations which to fine tune our processor for a given application.

Slide 66



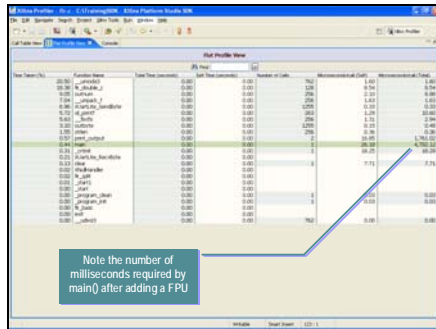
We can now create a new run configuration like we did previously. This time we will label the configuration `fir_sw_release` and our target will be `Release/fir_sw.elf`. We also ensure our profiling option is checked in the XMD Target Connection tab. We then select `Run` to build a new profile.

Slide 67



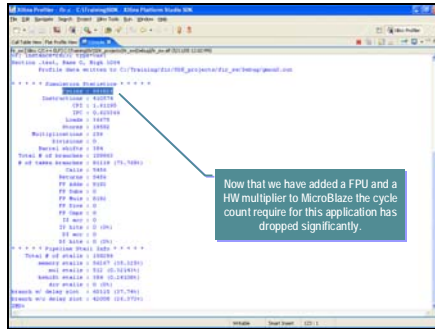
if we, again, launch this run and compare it to the last run, we'll see that the overall time spent in the various functions has changed dramatically. We also now see that `fir_double_z()` is a bigger portion, not the library call in that design. So we've actually substantially improved the performance of this design, simply by just adding a floating-point unit.

Slide 68



Now, if we look at that in the flat-profile view and compare those results to our previous result, we can see that we're down to about 4.7 milliseconds for this application from 62 milliseconds previously. This is about a 13X speedup for `Main()`. Not `fir_double_z()` is actually 53x faster than our previous run. We could further investigate how to optimize out the calls for `__umodsi3` for more potential improvement. So we substantially improved the performance of this algorithm, simply by changing the processor architecture ever so slightly.

Slide 69



If we go ahead and look at the statistics from our run, we can then see that the overall cycles taken before our simulation was down to about 660,000 versus 15 million.

The other thing to note here is also, down a little lower in the data, is that the floating-point adds and multiplies are now listed as 8192. So you can see that we do indeed have calls going to our hardware floating-point unit.

Slide 70

Performance Improvement

- Adding a Floating Point Unit, Barrel Shifter, and HW Multiplier to this design reduced the number of CPU cycles required by this application
 - Main() results in a 13x improvement
 - Fir_double_z() results in a 53x improvement
- Printf() accounts for a larger percentage of the CPU cycles in Main() now that the fir_double_z() function cycle count has been reduced


So if we contrast those numbers and take a look at things, we can see that Main sped up substantially.

The overall application saw about a 13x improvement but the fir sub-function, itself, actually saw an improvement about 53x. You know, Main also includes a number of their other things like print-outs and things like that that probably would not be included in our final embedded system. But, again, we can see substantial improvement, simply by changing the hardware available to better suit your software algorithm.

Slide 71

Hardware Emulation

- Compiling our custom processor system into an available development board is still an option and would provide a much faster simulation platform for large software simulations.
- This design was originally compiled for a Virtex-4 device on a ML401 development board




At this point, we do have a couple of other options available. We can actually go back to hardware emulation. This design originally was running on an ML 401 development board, which is a Virtex-4-based system and a lot of this data was originally captured simply via a UART terminal back to our PC.

Slide 72

Summary

- In 45 min. we have demonstrated:
 - Defined and built a custom CPU system
 - Generated and address map for the system
 - Created a C compiler tool chain specific to our system
 - Created all necessary drivers, libraries, and include files
 - Generated a custom C model for this new system
 - Setup a C project and configured multiple debug and run sessions.
 - Profiled a floating point FIR filter application on two different configurations of MicroBlaze.
 - Learned what functions are consuming most of our CPU cycles
 - Improved the performance of our function 53 times



So, in the last 45 minutes or so, we've defined a custom system, building a CPU system from scratch. If I needed 5 UARTs, I could have 5 UARTs; all of that could have been specified in the base system builder.

I generated an address map for my system. All the linker scripts that had to put things in various memory locations were automatically created.


I created a custom compiler tool chain, all my /lib and /include-directories were automatically generated for me. All the drivers to the various I/O blocks that I have specified would have been included. We only had a couple in this case.

We set up a C Project. We ran two different runs, profiled those runs and, again, improved the performance of an algorithm, about 53x. And, again, in a very short period of time.

Slide 73

Summary

- In reality, this takes 10 minutes at your PC!!
- Utilizing Xilinx Platform Studio we can generate cycle accurate software models and profile the performance of software running on a fully custom virtual system.
- We can quickly tune the performance of our processor architecture and system architecture to achieve an optimal balance of performance vs. hardware resources.

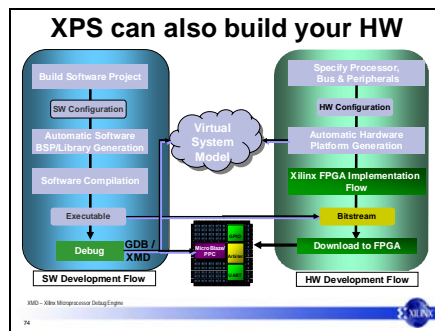


73

Now, in reality, if we had been doing this at the PC, this is about a 10-15 minutes process. I mean, simply to run these and set these things up, the first time is a very, very trivial thing to do. The learning cycle was fairly short. Eclipse is a very intuitive tool as well. It sets up most of the system automatically for us. And the point here is we can quickly tune the performance of a processor to a particular application and no longer are we constrained in having to architect our software for a particular CPU platform.

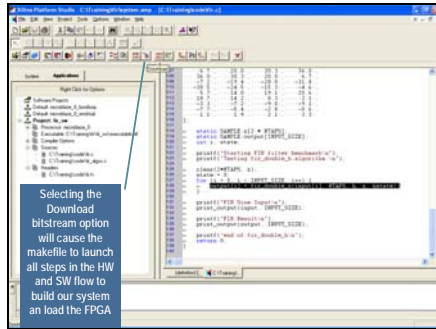
We can also go ahead and tune the processor to an application. Again, that's a new degree of freedom that we've not had before.

Slide 74



In the end you are creating a design in an FPGA. You have all the freedom of custom logic in your hands as well as a fully capable CPU. You choose the peripherals and tune the system to your application.

Slide 75



The last step here would be to actually run emulation. All I've really shown here is a screen shot that would show the FPGA implementation tools. All I have to do is go back to Xilinx Platform Studio and click a second button which would then launch the entire FPGA synthesis and bitstream generation tool system. And, then, I can actually download that to a hardware platform and go back and do Eclipse and target a new hardware platform instead of targeting a virtual platform and the same tool flow would have been used, just a different target!

Slide 76



In this example we were ultimately targeting a Xilinx ML 401 board, which can do everything from run Linux to serving a web page. It's quite a capable board. More information can be learned about this on the Xilinx Web site.

Slide 77



For those of you who are might be interesting in the Spartan family, this is a soft core so you can target a Spartan board as well. As we mentioned, soft cores can target any of the devices available from Xilinx -- either the Virtex or the Spartan family. So this would be a low cost, about \$100 Starter Kit board that you could use for experiments.

Slide 78

MicroBlaze 4.00 Summary

- MicroBlaze Increased performance
 - Up to 200 MHz operation with Virtex-4
 - Extends Configuration Flexibility
 - Configurable hardware multiplier
 - Floating point is a selectable option
- MicroBlaze v4.00 available now
 - Shipped with Xilinx Embedded Development Kit (EDK) 7.1i
 - Most complete design environment for customers flexibility
 - Optimized HW and SW partitioning
 - EDK7.1i — \$495 includes:
 - Platform Studio 7.1i development tools
 - Hardware and software IP support for MicroBlaze and PPC 405
 - No additional licensing fee or royalties for MicroBlaze

So if we take a look at MicroBlaze 4.0.

You'll see we have upped the performance of this processor, up to 200 MHz of operation with the Virtex-4 family. That's the highest performance family available from Xilinx.

We've extended our configuration, flexibility; we can configure hardware much more easily. A Floating Point Unit is available and hardware multipliers are now optional.

MicroBlaze 4.0 is available now in the 7.1 EDK tool system. The MicroBlaze core is actually part of the development system.

When you buy a development system, you do get a license for MicroBlaze.

That's a one time cost of \$495 and there are no additional royalties for using this soft processor.

Next Steps

- Get started
 - Buy the Embedded Development Kit (EDK) available at www.xilinx.com/edk
 - Select a Development board that suits your application.
 - Learn more about the Xilinx MicroBlaze 32-bit RISC processor at www.xilinx.com/microblaze

