THE MESCAL ARCHITECTURE
DEVELOPMENT SYSTEM (TIPI) TUTORIAL

by

Matthias Gries, Scott Weber and Christopher Brooks

Memorandum No. UCB/ERL M03/40

15 October 2003
THE MESCAL ARCHITECTURE
DEVELOPMENT SYSTEM (TIPI) TUTORIAL

by

Matthias Gries, Scott Weber and Christopher Brooks

Memorandum No. UCB/ERL M03/40
15 October 2003

ELECTRONICS RESEARCH LABORATORY
College of Engineering
University of California, Berkeley
94720
The Mescal Architecture Development System (Tipi) Tutorial

Matthias Gries, Scott Weber, Christopher Brooks
Electronics Research Laboratory
University of California at Berkeley

{gries|sjweber|cxh}@eecs.berkeley.edu
Contents

1. Introduction ........................................................................................................... 3
2. Installation ............................................................................................................. 5
3. Using Tipi ............................................................................................................. 9
4. Defining a customized instruction set ................................................................. 19
5. Developing Actors ............................................................................................... 25
6. Cycle-accurate simulation .................................................................................... 27
7. Hardware generation ............................................................................................ 31
8. Compiler View ..................................................................................................... 34
9. Conclusion ........................................................................................................... 35
10. References .......................................................................................................... 36

Revision History

<table>
<thead>
<tr>
<th>Oct 15th, 2003</th>
<th>Initial Release</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Introduction

The aim of the MESCAL framework is to introduce a disciplined approach to developing reusable architectural platforms that can be easily programmed to meet the requirements for a variety of applications. Our goal is to provide an architecture development system to specify, evaluate, and explore fully-programmable (micro-) architectures using a correct-by-construction method to ease the definition of customized instruction sets. The micro-architecture design tool called 'Tipi', which is the main focus of this tutorial, specializes on the design of application-specific instruction-set processors (ASIPs).

Compared to existing approaches, Tipi improves design productivity by following two design principles:

- **Multiple views approach**: Tipi maintains a single, simple, formal description of the architecture with a minimal set of restrictions on the design space. This allows a very broad class of architectures to be described. A view of this model is an abstraction that encapsulates the information required by a facet of the architecture. Each view is implemented with a synonymous tool in the Tipi framework for interacting with the architecture using the corresponding abstraction. Views do not introduce semantics not present in the model, and thus are consistent with the model by construction. Views described later in this tutorial include, for instance, representations of the micro-architecture, the instruction set, and generated hardware in Verilog.

- **Correct-by-construction operation-level design**: The design flow in Tipi encourages the designer to think of the data-path micro-architecture first. He/she lays out the elements of the data path without connecting control ports. For unconnected control ports, operations are automatically extracted which are supported by the data path. These operations form the basis for more complex, multi-cycle instructions that are composed of several primitive operations. Traditional design flows define the instruction set architecture (ISA) first and then implement a corresponding micro-architecture. In this situation, the problem arises that the actual architecture must be checked against the original ISA specification to see whether it implements the ISA. This verification step is avoided in Tipi, since all instructions are automatically derived from the micro-architecture.

1.1. Features

Features of the Tipi architecture development system include:

- The editors for the micro-architecture employ actor-based design as implemented by the Ptolemy software design framework on which the Tipi editors are based. This means in particular, actors, representing primitive data path elements or whole subsystems, can be interconnected and combined hierarchically. These actors can be defined in a data-type polymorphic way so that a logic or state element definition, such as an adder, can be used for different integer resolutions. Actors can reveal parameters of the underlying data path element to the designer. The graphical environment allows easy and intuitive assembly of the architecture by dragging and dropping components from predefined libraries. User defined components are also supported.

- The construction of complex, multi-cycle instructions is supported by defining
temporal and spatial constraints, i.e., automatically extracted operations from the data path can be combined in parallel and in sequence as long as resources for these operations do not conflict with each other, which is checked by Tipi. A dedicated editor supports the designer by highlighting all data path elements that are affected by combining operations on a cycle-by-cycle basis.

- A cycle-accurate simulator is generated from the micro-architecture description and the defined instruction set. For efficiency reasons, the simulator is generated in C++ and uses pre-decoded instructions to accelerate simulation speed. Case studies suggest that the obtainable simulation speed is only one order of magnitude lower than the execution speed of the simulation host, making the simulator a superior choice compared to existing, less accurate instruction set-level simulators.

- Constructive estimation is supported by generating a synthesizable, Verilog-based hardware description of the architecture. Different options for implementing a controller can be explored.

- An assembler is generated from the description of the instruction set to ease the deployment of the ASIP.

- The main design system is developed in Java, whereas speed-critical parts, such as the automatic extraction of operations and the generated cycle-accurate simulator, are implemented in C++.

- In order to ease debugging of the architecture, simulation probes can be used to output the inner state of parts of the micro-architecture. Due to the graphical composition of the micro-architecture and the automatic generation of operations and their semantics, the designer is able to quickly alter the design interactively.

1.2. Requirements

Tipi is being developed and tested under Linux (kernel version 2.4 and higher) and cygwin (http://www.cygwin.com/), which provides a Unix environment and GNU tools under Windows. Tipi should work under Solaris as well. In order to install and run Tipi, make sure to meet the following requirements:

- Java 2 platform, release version 1.4.2.
- Recent C++ compiler, e.g. gcc version 3.2 or higher installed.
- Main memory of at least 256MB, 512 MB recommended.
- Main CPU speed > 1GHz.
- 64 MB free hard disk space.

The tutorial is structured as follows. The next section describes the installation procedure for Tipi. The following section introduces the graphical environment, which is based on Ptolemy II, and the first steps towards modeling a data path of an application-specific instruction set processor. In Section 4, we then show how to define a custom instruction set from automatically extracted operations. In Section 5, we describe how custom actors can be developed. Section 6 continues with showing the options for generating a cycle-accurate simulator. Section 7 reveals the functionality for generating synthesizable Verilog output. Finally, Section 8 explains how we export the semantics of our design.
2. **Installation**

If you do not have jdk1.4.2, get it from:

http://java.sun.com/

2.1. **Cygwin on Windows**

If you do not have cygwin on windows, get it from:

http://sources.redhat.com/cygwin/setup.exe

or

http://ptolemy.eecs.berkeley.edu/ptolemyII/ptIIlatest/cygwin.htm

which has detailed installation instructions.

Note that mescal/views/simulation/model/UINT.h includes stdint.h, so Cygwin must provide stdint.h. The configure step below checks for stdint.h, but if you have not updated Cygwin since September, 2003, you might want to do so now. Usually, stdint.h is found as c:/cygwin/usr/include/stdint.h If that file is present, then there is probably no need to update Cygwin. If that file is missing, then you should definitely update Cygwin.

2.2. **Environment Variables**

On windows set the following environment vairables in your System Environment Variables. This is in Control Panel -> System -> Advanced -> Environment Variables. On Linux, set the following environment variables.

Set PTII to the directory where you checked it out to.

(Optional) Set PTJAVA_DIR to the directory where jdk1.4.2 lives.

Add $PTII/bin to your path. Create a new variable called PTJAVA_DIR and a new variable called PTII. Edit the PATH variable to include $PTII/bin and $PTII/bin/mescal.
2.3. Install GMP

Under Cygwin, install gcc 3.2 (gcc-3.3.1 had problems with gmp and c++)

The source for GMP can be found at http://www.swox.com/gmp/.

We are using gmp-4.1.2, but most any version of gmp should work.

I. Start up a Cygwin bash shell.

II. The Mescal configure will look in /usr/local and
$PTII/mescal/include and $PTII/mescal/lib for gmp, so
gmp should be configured with:

./configure --enable-cxx --prefix=$PTII/mescal

The installation instructions (texinfo file says)

--start--

Microsoft Windows
On systems `*-*-cygwin*', `*-*-mingw*' and `*-*-pw32*' by
default GMP builds only a static library, but a DLL can be
built instead using

./configure --disable-static --enable-shared

Static and DLL libraries can't both be built, since certain
export directives in `gmp.h' must be different. `--enable-
cxx' cannot be used when building a DLL, since libtool
doesn't currently support C++ DLLs. This might change in
the future.

--end--

III. Make gmp: make >& make.out

IV. Run make check and identify problems now with g++ that will bite you
later:

make check

Under Solaris, if make check fails with

FAIL: t-gmpmax

ld.so.1: fatal:
/usr/local/lib/libgcc_s.so.1: wrong ELF
class: ELFCLASS32

you might want to set the ABI environment variable to 32 so as to avoid a
64 bit ABI problem.

setenv ABI 32

./configure --enable-cxx --prefix=$PTII/mescal/include

and then re run make and make check.

V. Finally, run make install: make install >& install.out
2.4. Install BOOST
Boost is also needed. Download Boost from http://www.boost.org.

Install it with:

    mkdir $PTII/mescal/include
    cd boost-1.30.2
    cp -r boost $PTII/mescal/include

2.5. GNU make
GNU make is required to handle GNU make conditionals in relsat/makefile and kirika/makefile.

If you are running under Cygwin or Linux, then GNU make is probably already installed.

To test for GNU make, run make -v, you should see something like:

    cooley 6% make -v
GNU Make version 3.79.1, by Richard Stallman and Roland McGrath.
Built for sparc-sun-solaris2.5.1
Copyright (C) 1988, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 2000
  Free Software Foundation, Inc.
  This is free software; see the source for copying conditions.
  There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
  PARTICULAR PURPOSE.
  Report bugs to <bug-make@gnu.org>.
    cooley 7%

Any recent version of GNU make should work.

2.6. Building
1. Mescal uses libraries that get installed in $PTII/mescal/bin. Under Linux or Solaris, add $PTII/mescal/bin to LD_LIBRARY_PATH. If you are running csh under Linux or Solaris:

    setenv LD_LIBRARY_PATH $PTII/mescal/bin:$LD_LIBRARY_PATH

Under Cygwin, add $PTII/mescal/bin to your PATH.

If you are running bash under Cygwin:

    PATH=$PTII/mescal/bin:$PATH
    export PATH

Mescal uses libraries that get installed in $PTII/mescal/bin. Under Linux or Solaris.

II. Configure Mescal and Ptolemy II

    cd $PTII
    ./configure

III. Make

    make fast >& make.out &
2.7. Simple Example
Below is how to run a start up the Convolution Coding Processor example

I. Change to appropriate directory:
   cd $PTII/mescal/domains/mescalPE/demo/cc

II. See the index.htm file in that directory.

2.8. Building under Eclipse
See
for instructions on setting up Eclipse.

2.9. Installing via CVS
If you have CVS access to the mescal and Ptolemy II trees

1. Change to the directory above where you want to install Ptolemy.

Installing ptII ...

2. > cvs co ptII
3. > cd $PTII
4. > ./configure
5. > make fast

Installing mescal ...

6. Check to see if there is a $PTII/mescal directory present.
   If there is then goto 10.
7. > cvs co ptII_mescal
8. > mv ptII_mescal mescal
9. > cd mescal
10. > cvs update -dP
11. > make fast

To run mescal (assuming that your path is setup correctly) ...

12. > mescal

In future updating, always use cvs update -dP in case you did not know.

2.10. Hardware design flow
The suggested design flow for hardware output in this tutorial is based on the design
compiler by Synopsys and the verilog desktop by Cadence for simulation. The supported
scripts could easily be retargeted to other tool flows.

2.11. Documentation
Further documentation, including this tutorial, can be found at $PTII/mescal/doc.
3. Using Tipi

Simply enter `mescal` in a shell to see Mescal's starting screen as seen in Figure 1.

![Mescal Starting Screen](image)

**Figure 1:** The Mescal starting screen.

Since the graphical editors are based on Ptolemy classes, having a look at the Ptolemy design guide [1] in addition to this tutorial could also help to familiarize you with the usage of the graphical environment.

In order to start our design, we would like to define the 'black box' view of our ASIP. Under the File menu, choose `New -> Graph Editor` to open a new editor window. Your screen should look like Figure 2.

![Empty Black Box Editor](image)

**Figure 2:** Empty black box editor.

On top you find a row of editor buttons. The first five buttons from the left deal with the
display of the working area, i.e. they are used to zoom in, zoom out, or recenter the working space. The following three red buttons are unused in Tipi¹. The following four buttons are used to define input and output ports to higher levels in the hierarchy. The black-filled buttons are used to define single connections whereas the white ones are able to aggregate several connections. Since we are currently on the highest level of abstraction, these four buttons do not have a useful meaning here. It is also important to note that only one relation (connection) can ever be connected to hierarchical ports, whereas on the lowest level of abstraction multi-ports can be used on actor boundaries to connect several relations to the actor.

On the left, you see two separate display areas. The upper area provides a directory listing of available library elements for designing models. The lower area is a small version of the editor working area that allows you to quickly navigate in the working area and refocus the display.

In order to begin with our design, we have to define the black box of our ASIP. Double-click on the actor library directory entry in the library display on the left. We are going to design our micro-architecture using Mescal’s processing element model of computation, which is why you should double-click on the mescalPE subdirectory. Then drag-and-drop the PE entry onto the working area on the right. Your window should now look as in Figure 3.

![Figure 3: Initial black box.](image)

In this way, you just defined the initial black box of your ASIP. Recognize that one input has already been defined for you. You can move the mouse over the input port symbol to see a small description text. The input port is called instruction and will provide instructions to the underlying (to be defined) data path on a cycle-by-cycle basis.

Move your mouse pointer over the PE element and right-click to open a pop-up menu. You can choose Customize Name to rename your ASIP black box. We will now

---

¹ Within Ptolemy, these buttons control the visual simulation of the model of computation in the editor window, defined by the corresponding director.
define a small data path for our ASIP. Choose Look Inside from the pop-up menu to open an editor for the data path. The new editor window should look like in Figure 4.

![Figure 4: Empty editor for processing element domain.](image)

Note that a Ptolemy director has been placed on the working area for the processing element domain of Tipi. It defines the kind of coordination between elements of the data path and affects internal data representations. As a user of Tipi, you do not have to worry about the meaning of this block. It will always be there for Tipi architecture editors.

You also see an icon representing the instruction input port of our black box in the upper left corner. We now build a small data path that will provide us with a functional unit that performs addition and buffers inputs and outputs. Again, we look in the actor library and focus on the components and memories subdirectories. In the memory directory, you find different kinds of state elements, such as registers, flip-flops, and RAMs. In the components subdirectory, there are combinatorial logic elements, such as decrement, addition, and comparison. The components directory also contains elements for routing connections in the data path. See [3] for a detailed description of how actors are described.
We start with a simple example. In Figure 5, you see a data path consisting of two input flip-flops, an adder element, one output flip-flop and three corresponding ports. Drag-and-drop the corresponding elements from the library and add the ports using the buttons from the button row on top. In order to draw connections (‘relations’ in Ptolemy terms) between components, click on one port and hold the mouse button, drag the mouse to the destination port and release the mouse button. Rename the components. Have a look at the window showing the black box view of our ASIP. As you can see, the new ports have been added to this representation as well, i.e. all the displays are kept consistent. We just designed a simple data path that could be used as follows: with every cycle, the input values are latched into the two flip-flops. In the following cycle, the addition takes place and the result is stored into the output flip-flop. In yet another cycle the result could be read out through the output port. This functionality could of course also be pipelined to keep the adder busy. But how would this mode of operation work exactly? Obviously, we have not yet dealt with the instruction input so far. Also, in the current display, all ports seem to be connected. So, where does the programmability come into play?
The answer is that there are actually unconnected ports. Have a look into one of the flip-flops. You see, as in Figure 6, that the flip-flops from the library are hierarchical components containing further components. Indeed, the ports defining the read or write state are unconnected. The read and write addresses are also unconnected since the flip-flop can only store one element. We also see connected data input and data output ports.

This configuration underpins the main idea of the design flow in Tipi: You can leave control inputs unconnected in the design and Tipi will generate operations automatically so that you are able to program this functionality. For the current design, the program would be fed cycle by cycle from external stimuli through the instruction input port to initiate computations in the data path. Note that the instruction port is intentionally left unconnected. The assignment from operations to control signals is implicit. For you as a designer this means that you do not have to care about control encodings at all.

You may be wondering what the precision of the implemented adder is. The answer is that the precision has not been defined yet. Go back to the architecture view showing the data path (Figure 5). We are going to determine the resolution of the data path by fixing the type of the input ports and propagating this type information through the path. Select one of the input ports, e.g. input A. Choose Configure from the pop-up menu. You see that one design parameter called type is exposed to you. Type ‘32’ into the text field and commit your input. You just defined the resolution of the port to correspond to 32 bits. Do the same with input B. This information is enough to propagate the type information through our data path. In order to do this, choose Resolve Types from the Tipi menu.

In the next step, we are now going to automatically extract all possible cycle-by-cycle operations of the data path. In order to do this, choose Operation View from the Tipi menu of one of the open mescal PE domain windows. Another window should appear and look like in Figure 7. In the upper part, you will recognize the display of the data path from the architecture view. However, it is for debugging purposes only and cannot be altered in this view. In the lower part, you find all the functionality required to

Figure 6: Flip flop library element.
handle automatically extracted cycle-by-cycle operations. Operations can be found with unresolved types. For our design, five primitive operations have been extracted from the data path description in the architecture view. Select one of the operations in the list on the left. As soon as an operation is selected, the active parts in the data path for that operation are highlighted. In the text boxes on the right, additional information is displayed, such as the input and output ports used and the semantics of the operation. In addition, the data flow through the components is displayed in textual form. The button row on top of the text boxes can be used to customize the text displays. Currently, all paths to components are displayed using paths relative to the currently displayed part of the data path. This can be altered to use the full paths or the names of the components only. Further, the display of the semantics and the data flow can be switched off. In order to improve the clarity of the displays, multiple tabs can be enabled to simplify the display if several operations are displayed (selected) together. Finally, syntax highlighting can be switched on\(^2\).

![Diagram](image)

**Figure 7: The operations view.**

With the buttons on the left, operations can be customized. Operations can be renamed. If certain operations are not needed, their use can be restricted by unchecking the use field in the corresponding row of the list. Unused operations can be hidden from the list by pressing the Hide button and brought back by clicking Show All.

For our design, operations have been created to latch data from the input ports, perform the addition, and read out the result through the output port. The very first operation is a

---

\(^2\) The options for multiple tabs and using colored syntax highlighting are memory-intensive. If several operation view windows are open and you recognize considerable delay, increasing the maximum heap size of the java virtual machine to at list 256MByte is recommended to decrease the penalty of garbage collection.
no-operation without functionality. After enabling some display options, the operation view and our set of operations could look like in Figure 8.

![Figure 8: Operation view with renamed operations.](image)

With our basic operation set defined, we are close to actually being able to program the data path. What is still missing is a program memory and some means to step through operations cycle-by-cycle, which would be a program counter in traditional architectures. In order to store the program, change to the architecture view (showing the data path, Figure 5) again and add a MescalPETypedCompositeActor from the mescalPE actor library that will be used as a container for the PC control unit. Look inside this actor and add an instruction ROM memory element from the memory subdirectory of the actor library. Note that there is a special element called InstructionROM for this purpose. It contains a ROM called InstructionState, which is mandatory for Tipi to associate an assembly program with this memory. Add a register, a constant generator, a distributor element, a multiplexer, an increment element, and an output port to the working area. Interconnect these elements following Figure 9. You will recognize that some of the actors have ports which are not filled but have an outline only. These ports represent multi-ports, i.e. they can be connected to several connections. Also note that control inputs to the multiplexer and the distributor (select inputs) remain unconnected. You can always quickly determine the name of a port by placing the mouse pointer over it. Rename the register and the hierarchical output port to some more meaningful names. Note that you cannot rename ports of actors on the lowest level in the hierarchy since the semantics of the actor must match the names of the ports. You just created the PC control logic which allows you to reset the PC, increment the PC, and of course read out the instructions memory. You probably should set the size of InstructionROM memory to something larger than one. This can be done by right
clicking on the actor and choose the configure menu item. You can then change the size parameter.

![Figure 9: PC control and instruction ROM.](image)

Go to the architecture view showing the data path. Add an output port and connect the output of the PC control actor with the output port. Rename the output port. This view should then look like in Figure 10.

![Figure 10: Data path with PC control.](image)

Finally, the nextInstruction output must be fed back to the instruction input. Change to the black box view of the ASIP and interconnect these two ports accordingly, as shown in Figure 11. Issue a type resolution from one of the open architecture views. Note that type resolution will automatically be invoked for all the other views. If the operation view is still open, you will recognize that you have to press the update button to
generate the new instructions for the PC control path. Otherwise, open an operation view from the Tipi menu of one of the architecture views.

![Image of operation view](image)

**Figure 11:** Black box with instructions fed back.

In the operation view, look into the PC control actor using the pop-up menu of this actor. We see that three new operations have been generated to reset the PC, to increment the PC, and to read out the instruction ROM. Rename the operations accordingly. The operation view should look like in Figure 12.

![Image of operation view](image)

**Figure 12:** Operation view for PC control.

With this operation set, we could already program our data path. However, we would
have to manually combine all potential concurrent operations for each cycle, such as operations to control the PC and operations for the data path. In the following section, we will have a look at defining more complex, e.g. multi-cycle, instructions. We will define a more complex instruction set by combining primitive operations that do not conflict on hardware resources of the data path. Now is a good time to save your design. Choose save as from one of the open views. Note that Tipi opens a new black box view using your new design name. You can safely close all other windows.
4. Defining a customized instruction set

In the following section, we are going to combine primitive, automatically generated cycle-by-cycle operations to form more complex, multi-cycle operations that can also exploit the concurrency of operations if there are no resource conflicts in the data path. From one of the open views, choose Constraints View from the Tipi menu. The name has its origins in the fact that by defining more complex instructions we constrain possible execution patterns of primitive operations and thus also potentially affect the complexity of the automatically generated instructions decoder. The constraints view should look like in Figure 13.

![Figure 13: Constraints view.](image)

The list on the left shows all primitive operations and later on also more complex operations ('macro-operations') defined by several primitive operations. The mode column allows you to provide hints to the instruction decoder. The 'HW' mode indicates the operation or operations that create a macro-operation always occur in the manner indicated in the constraint. The 'ASM Macro' indicates that the constraint should be expanded before it is analyzed by the instruction decoder generator. In the future, fuzzier definitions on constraints could be developed to aid in decoder generation.

On the right side you see two editor/display areas which are used to define macro-operations. The upper display allows you to define a constraint in text form, whereas the lower part uses a table representation in order to better structure the input. If the table grows in the horizontal direction, more primitive operations can be executed concurrently. This is why columns are associated with issue slots. If the table grows vertically, it means that more clock cycles are needed to execute the macro-operation and primitive operations have been combined in sequence.

In order to define constraints, either the text editor or the table can be used. However, as soon as you decide on one option, the other option will be disabled. After having defined a constraint, it must be checked against conflicts on resources in the data path. On top of the editor/display areas, you find buttons for creating and checking constraints.

Let us create a simple macro-operation by combining the primitive operations for reading in the input ports A and B. Push the New button and choose a name for the new macro-
operation. Click into the text editor field and specify the constraint by typing “latch_A || latch_B”. With the “||” symbol you want the primitive operations to be executed concurrently since the associated data flow does not collide in the data path. Push the Check Constraint button to verify whether there are any conflicts or typos in the definition of the new constraint. If everything goes well, the table will be updated and the constraint view should look like Figure 14.

![Figure 14: Defining macro-operations.](image1)

You could now use latch_inputs in an assembler program to replace the composition of latch_A and latch_B. This macro-operation is not useful yet. It makes sense to also update the PC control state concurrently with latching the inputs. We are going to do this change to the current constraint using the editor table. Push the add slots button and add two further slots. You will notice that the text editor has been disabled since we chose to use the table for changing the constraint. Enter incPC and nextInstr into the two new row entries. Finish your input by hitting enter. Finally, check the constraint. The constraint view should look like Figure 15.

![Figure 15: PC control and data flow combined.](image2)

We can continue the definition by adding additional cycles. Have a look at the example in Figure 16. Two further clock cycles have been added to perform the addition and to read out the data. The symbol “;” denotes sequential execution. This is why it makes sense to increment the PC and read out the next instruction in every cycle. Do not forget to push the Check Constraint button to verify whether your definition contains a typo or resource conflicts between the operations. Note that constraints that are highlighted yellow have not been checked.
Figure 16: A temporal constraint.

If one of the operation views is still open, you will see that in addition to the list of operations there is also a list of valid macro-operations that appears below. If multi-tabs is enabled, you can step through the different cycles of the macro-operation and watch the affected parts of the architecture, as displayed in Figure 17.

Figure 17: Operation view with macro-operation.

The \texttt{do addition} constraint defines a sequence of three clock cycles to perform one addition together with the required I/O. This is an inefficient use of hardware resources. Figure 18 shows a counter example, where a single macro-operation allows one to use the data path in a fully pipelined way. The latch, add, and read-out phases are combined concurrently, not sequentially. Note that the increment PC and next instruction primitive operations have been left out intentionally. In this example, the \texttt{pipe-addition}
macro-operation would be sufficient to describe an endless loop of adding input values in a pipelined fashion. The behavior would be close to an ASIC and the assembler program would only consist of this single macro-operation. These simple corner examples reveal how easy it is for the designer to explore trade-offs concerning flexibility, program memory requirements, and the complexity of the instruction decoder by iteratively defining and hiding different sets of constraints.

![Figure 18: Pipelined addition.](image)

In the last example in this chapter, we are going to extend one of the operations to use parameters. So far, we have used operations in a streaming computation style where data is processed in a fixed manner. In order to introduce an intermediate parameter to the addition operation, go back to the architecture view and add an immediate actor element with a corresponding multiplexer in front of the adder for input B as shown in Figure 19.

![Figure 19: Extension for add immediate operation.](image)

Update the display in the operation view. We see an additionally generated operation that

---

3 In order to assess the complexity of the instruction decoder, the hardware view must be used to generate Verilog output, as it will be shown in a later section.
uses the immediate input as a parameter, see Figure 20. In the figure, the operation is already renamed accordingly. How would you use the newly generated operation that requires one parameter?

Figure 20: Add immediate operation view.

Have a look at the constraints view again. The new ADDI operation uses a parameter list with one parameter, where $a$ is a placeholder for this particular parameter, see Figure 21.

Figure 21: Add immediate using one parameter.

For instance, a valid usage of ADDI could be ADDI(2), ADDI(5), etc. This instruction will use a 32-bit field in the microcode to encode the parameter. In the same way, more complex operations can be defined in the constraints view which could use one or several parameters. Several parameters are separated by commas. As an example, the sequence
(ADDI(a) || nextInstr() || incPC());
(ADDI(b) || nextInstr() || incPC())

could be defined as a constraint ADDSeq with two parameters, i.e. ADDSeq(12, 3)
would be a valid use of ADDSeq and its signature would be ADDSeq(a, b).
5. Developing Actors

So far we have only utilized existing library actors in our designs. Although this may be all that is needed for most designs, we do provide the mechanism to design new actors. To create a new actor, drag either an AnalyzableActor, a ProbeActor, or a SimulationOnlyActor from the library menu. See [3] to understand the difference between these three types of actors. After dragging the actor, right click on it to select the Configure Ports item. This will allow you to add, delete, and modify ports on the actor. After adding ports, you must create a semantics file. This can be done in an editor of your choice. See [3] to understand how to describe the semantics of an actor. When writing the semantics, make sure that the port names in the semantics file match the port names in the architecture view. After you have completed describing the semantics of the actor, right click on the actor, and select the Configure item. Set the semantics parameter to the location of the semantics file that you have just created. You can also create environment variables for this actor here.

For actors that contain memory elements, there is a special FileParameter called:

\[ \text{init <memory name>} \]

To create this parameter, click on Add and you will get a Parameter box, see Figure 22. Make sure that you specify the class to be

\[ \text{ptolemy.kernel.data.expr.FileParameter} \]

![Figure 22: Parameter Box](image)

In order to give you an idea how an actor description looks like, we will have a look inside our adder element. In the architecture view (Figure 19), choose Look Inside from the pop-up menu of the Add element. A text display with two display areas appears. The defined display area shows the original definition of the actor and should look like:
Add(out, inputA, inputB) {

    output out;
    input inputA;
    input inputB;

    or(
        fire(out.p.1, inputA.p.1, inputB.p.1) {
            out = inputA + inputB;
        }
        no_fire(out.p.0, inputA.p.0, inputB.p.0) {
    }
    }

The signature of the actor with its input and output ports is defined at the beginning. The functionality of the actor is described using firing rules, which are enabled if certain signals are present. The last firing rule is the default rule that is valid if no signal is present. In the contextual display area, you can see the definition of the actor under the current context of the design. In our case, this means the type of the ports has been resolved to be 32 bit. The definition should look like

    input <32> inputA;
    input <32> inputB;
    output <32> out;

    or(
        fire_0(out.p.1, inputA.p.1, inputB.p.1) {
            out = (inputA + inputB);
        }
        no_fire_1(out.p.0, inputA.p.0, inputB.p.0) {
    }
    }

For further information concerning the description language and the composition of actors, please have a look at the reference manual [3].
6. Cycle-accurate simulation

In order to use the simulator that we are going to generate, we have to write an assembly program first. With the instruction set defined in Section 4 we have different options for implementing an addition. Make a directory for all files related to simulation in your current working directory, e.g. my_ASIP_sim. In an editor of your choice, create a text file called addition.asm with the following content, which defines the program for a non-pipelined version of our addition:

{nextrInstr(), incPC(), latch_A(), latch_B()}
{nextrInstr(), incPC(), ADD()}
{nextrInstr(), resetPC(), read_result()}
{nextrInstr(), incPC()}

This code uses primitive operations only. An instruction issued to the processor in one cycle is determined by curly brackets, whereas operations within an instruction are separated by commas. The first line latches the inputs, the second line performs the addition, the third line reads out the result and the final line jumps back to the beginning of the program. One iteration therefore takes four cycles.

An alternative program could make use of one of our custom instructions. Create a second file addition2.asm with the following content:

{do_addition()}
{nextrInstr(), resetPC()}
{nextrInstr(), incPC()}

This program uses our custom instruction do_addition() to shorten the assembler program. One iteration takes five cycles, since resetPC() needs an explicit extra cycle because we did not include it in do_addition().

Finally, our third example implements a pipelined version of the addition and simply consists of a two lines of code. Create a third file addition3.asm with the following content:

{pipe_addition()}

In order to generate object files and the simulator, open the assembler and simulation views (in the Tipi menu) from one of the present views. The two windows in Figure 23 and Figure 24 should appear. Specify the path and the file name of the assembler file (addition.asm) in the assembler program line in the assembler view. Specify the directory for the generated files in the corresponding line of the simulation view. Restrict the number of cycles of the simulation to 20 cycles. You should select either a compiled code or an interpretive simulator. Choose the interpretive simulator for now since we do not want to rebuild the simulator for each program. You can now press the Build Simulator button to generate a simulator and a testbench skeleton for the simulator in C++.
In your specified simulation directory, three files should have been generated. Assuming that you chose sim as simulator name, sim.cc represents the main simulator file. sim_Testbench.H is used to generate stimuli for the design under test. sim_Instruction.H contains representations for instructions used by the main simulator. In order to perform simulations, you only have to deal with the testbench file. Open sim_Testbench.H in a text editor. You should recognize the automatically generated functions that deal with reading and writing ports on the periphery of the PE. They follow <relation to port>_read() and <relation to port>_write() naming scheme. In our example, the input ports input_A and input_B and the output port final_output are on the periphery. We use the corresponding function bodies in the testbench file to feed in and read out data. These functions are the interface to our simulator. For the inputs, simply change the value 0 in the expression:

```
    return UINT<32>::T(0)
```

to a 32-bit unsigned integer of your choose. In the body of the output function add the following line:

```
    cout << "value on " << cycle << " = " << value << endl;
```

This will echo the value that is placed on the final_output, and will indicate on
which cycle it was placed. Of course, any code can be placed in these blocks to interface to the processing element.

Compile the simulator with:

```bash
g++ -O3 -Wall -lgmp -lgmpxx sim.cc -o sim
```

Now return to the assembler view and create object files for the three programs that you wrote. To do this change the assembly file and press the Build Simulator Object button. This will create an object file with the same name as your assembly file, but the asm extension will be replaced by a obj extension. We can now run the three programs by invoking the following:

```bash
./sim addition.obj
./sim addition2.obj
./sim addition3.obj
```

You should see the results of your additions. Note that the results for each run will produce results at different times.

Instead of creating an interpretive simulator, we could have created a compiled code simulator for each program. To do this, select Compiled as the Simulation Type in the simulation view. You need to modify the testbench again and recompile the application. However, now the program (assuming you called it sim) is invoked simply by calling:

```
./sim
```

This will run whatever program was specified in the assembly view. Remember, that compiled code simulators are faster than the interpretive simulator, but do require that you produce a new one for each assembly program.

Please note that you could perform a simulation of your design without having specified any program control logic at all. For instance, let us assume that we stopped our design at the level of Figure 8, i.e. providing us with a data path only. There would be no instruction ROM and no connection to the instruction input. However, we could already write an assembler program, more precisely a trace of execution of the data path. For instance, the program:

```c
{latch_A(), latch_B()}
{ADD()}
{read_result()}
```

is a valid execution trace for one iteration of the addition and can be specified in the simulation view as assembler file. The generated simulator would execute this trace for the data path accordingly.

As a last step in this chapter, we are going to introduce a probe element into the design. Probe elements allow us to monitor the inner state of the design for debugging purposes. Go back to the architecture view and add a distributor and a probe element (simulation subdirectory) to the design to monitor the output, as sketched in Figure 25.
Go to the operation view and update the display. An additional operation has been recognized that can be used to execute the probe, see Figure 26.

That means, with this additional operation, that can be used concurrently with any other operation, we have a very fine grained control over the amount and times when we actually want to generate debug information in our program without disturbing the main execution of the program. The probe simply writes its input to standard out.
7. Hardware generation

We also provide a path to hardware in the form of synthesizable Verilog. By invoking the hardware view from the Tipi menu, you will see the window shown in Figure 27. Specify the directory where you want to generate the Verilog in the hardware directory item. The hardware type section allows you to create hardware for "simulation only" or for "simulation/synthesis". "Simulation only" hardware includes control for probe actors in the microcode, but since probe actors cannot be synthesized we call this hardware "simulation only". "Simulation/synthesis" hardware does not contain control for probe actors. When running "simulation only" Verilog, the simulation will produce results that are exact with the C++ simulation. When running "simulation/synthesis" Verilog, the simulation will execute probe actors on each cycle and hence the simulation results will include probes at unwanted times, but the simulation is still equivalent on the cycles that the probe should be executed as according to the assembly program. If a "simulation/synthesis" hardware type is selected, then the user can specify the stages of synthesis that should occur, the libraries that are needed for synthesis, and a dc_shell script template. After building the hardware, a dc_shell script is produced that is parameterized with these settings. The hardware name section allows the user to specify a unique name for the hardware. The end simulation allows the user to set a limit on the number of cycles to run the hardware in a simulation (it has no bearing for synthesis). The assembly program section reflects the currently specified assembly file.

Figure 27: Hardware view window.
After configuring the hardware view, we press the Build Hardware button to produce the Verilog. The following five files are created assuming the hardware name is set to design and the assembly file is addition.asm:

    design_data.v
    design_control.v
    design_top.v
    design_testbench.v
    design_addition.v

If the hardware type is "simulation/synthesis" the following dc_shell script is also generated:

    design.tipi.scr

The file `design_data.v` contains the data path of the design. The file `design_control.v` contains the controller for the design. The control that is generated is currently always a horizontal microcode controller. The file `design_top.v` connects the data path and controller. Assuming that the hardware was built as a "simulation/synthesis" hardware, these three files are all that is needed for synthesis. To invoke synthesis run:

    dc_shell -f design.tipi.scr

We can successfully analyze and elaborate designs. We do get some warnings about feed-throughs and shorts, but from a code generation standpoint these are sensible warnings. We are open to comments on the Verilog that gets generated and better synthesis strategies.

The file `design_testbench.v` provides a testbench for simulation. Finally, the file `design_addition.v` is used to initialize the contents of the instruction memory. Verilog simulation requires all five Verilog files. To run the Verilog simulation do:

    verilog   design_data.v design_control.v
    design_top.v design_testbench.v design_addition.v

If you want to run a different program, all you need to do is open the assembly view and change the assembly program and click on the Build Hardware Object button. This will create one Verilog file that has the same name as the assembly program file, but asm is replaced with v. For simulation, simply change the assembly Verilog file on the command line.
The diagram in Figure 28 demonstrates the controller strategy that is being explored. We currently automatically generate the Program Store, Horizontal Microcode Control, and the Data Path components. In the future we will be using a more sophisticated strategy to encode and decode the control bits in order to compress the memory size. The encoder is a software module. The decoder is a hardware module. We will utilize information from the operation and constraints views as well as from traces, to automatically generate/parameterize the encoder and decoder. Currently, our encoder simply places the code object in the Program Store, and the Decoder simply moves data from the Program Store to the Horizontal Microcode Control Buffer.
8. Compiler View

We provide a simple utility, which is called the compiler view, to export the design so that a compiler and other tools can be generated. This utility currently produces an xml file. The file contains all the operations, the assembly format for each operation, the microcode signature for each operation, the semantics of each operation, information about memories, information about I/O, and information about constraints between operations. We are currently using this information, coupled with information from the constraints view, to generate compilers and hardware decoders. The generation of the compiler and decoder is not included in this release.

Figure 29: Compiler view window.

The compiler view allows you to specify the location of the xml file to be exported. Pressing the Export button creates the xml file. Pressing the Import button simply reads and echoes the xml file. The xml parser that gets invoked by the Import button can be modified by other tools to retrieve information about the design.
9. Conclusion

We have presented a tutorial design in this document to show the facilities of the Architecture Development System called Tipi, which is part of the Mescal project at UC Berkeley. The example underpins the major design concepts and capabilities of Tipi, i.e.

- A correct by construction, bottom-up approach automatically generates all primitive operations supported by the modeled data path.

- A customized instruction set can be defined by combining primitive operations sequentially and concurrently, trading off code size vs. speed vs. instruction decoder complexity.

- A fast, cycle-accurate simulator and an assembler can be generated from the architecture description.

- A path to hardware output (Verilog) is provided.

- The design flow is supported by graphical editors.

The bottom-up approach is in particular feasible for the development of application-specific instruction set processors, where new application domains require a frequent adaptation of the instruction set to the characteristics of the application domain. The correct-by-construction method thus avoids the expensive verification of the instruction set together with the implemented architecture.

9.1. Acknowledgment

The Mescal project and the Tipi framework are supported, in part, by Infineon Technologies and the Microelectronics Advanced Research Consortium (MARCO), and are part of the efforts of the Gigascale Systems Research Center (GSRC).
10. References

