

PICSIL

A Data Flow Approach to Silicon Compilation

M W Pearson
P J Lyons
M D Apperley

*Department of Computer Science
Massey University
Palmerston North, New Zealand*

Silicon Compilation is a promising approach to designing today's complex ICs, which have rendered traditional design methods inadequate. Here we describe PICSIL, a graphical input language for a silicon compiler, based on Data Flow Diagrams. A PICSIL diagram is a network representation of a digital system. Several devices have been described in PICSIL. One is included here as a demonstration.

Introduction

Integrated Circuits with over one million transistors are now common, and the current doubling of circuit density every two years seems likely to continue. Consequently, VLSI design has become a major bottleneck in new product development. SSI and MSI design methods are quite inadequate. The design of one 60,000-transistor chip took 15 technical experts three years and cost several million dollars (Johnson & Mazor, 1984). Developing a chip under these constraints in a competitive market is a gamble, and chips with long development times are likely to be outmoded by the time they appear. Developers play safe, choosing conventional, inefficient, architectures rather than risking developing new ones and incurring prohibitive design costs. Small custom runs are rarely worthwhile.

Silicon Compilation (Johannsen, 1981) is the most promising of a number of design techniques (Rubin, 1987) to address these problems. A silicon compiler is a computer system which automatically translates a high level description of a device, with all detail suppressed, into an IC mask layout. The designer is not concerned with the physical details of the layout, and can become productive with far less training. The design time and cost of a moderately complex system can be reduced by orders of magnitude.

A new Hardware Description Language

PICSIL is a Hardware Description Language (HDL) for specifying digital designs to a Silicon Compiler. Currently, the language definition is complete, the graphic editor is nearly complete, and an interface to a lower-level text-based silicon compiler (De Micheli, Ku, Mailhot & Truong's (1990) OLYMPUS system) is about to be started. A number of HDLs designed for the same purpose already exist. Why invent another? To answer this, let us look at programming languages in general, and then consider HDLs.

There is a "semantic gap" between a programmer's mental model of a problem and its ultimate low-level machine code solution. A programming language's data-manipulating statements form an island in the middle, from which bridges extend in both directions. The compiler bridges towards low level machine code, and in the other direction, subroutines and top-down programming techniques together comprise a bridge reaching almost to the programmer's mental model. Inspiration fills the remaining gap. (See figure 1).

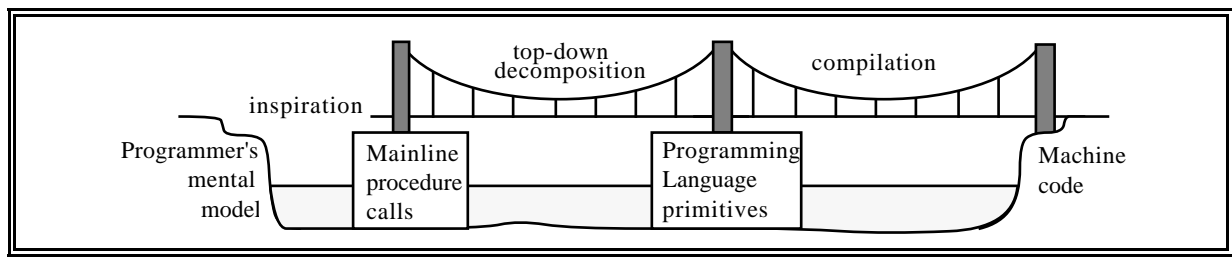


Figure 1. How programming languages bridge the semantic gap

Hardware Definition Languages have a wider gap to bridge, but they are restricted by their reliance on conventional programming language constructs, with only minor extensions to adapt them to describing hardware. The semantic gap left between the designer's mental model and mainline code, to be spanned by unassisted inspiration, is too great.

Personal experience in applying the HDLs to a number of problems showed that the designer always, spontaneously, included a planning stage before writing any text, interpreting his or her mental conception of the device as a diagram of its architecture. (See figure 2). Diagrams were also often used for expressing the two-dimensional structure of the sub-problems. Even with this type of self-help, subsequent translation of the diagrams into a linear textual form was never a trivial task. The idea began to grow that the natural language for expressing the high-level architectural aspects of a hardware design is diagrammatic, and not textual at all.

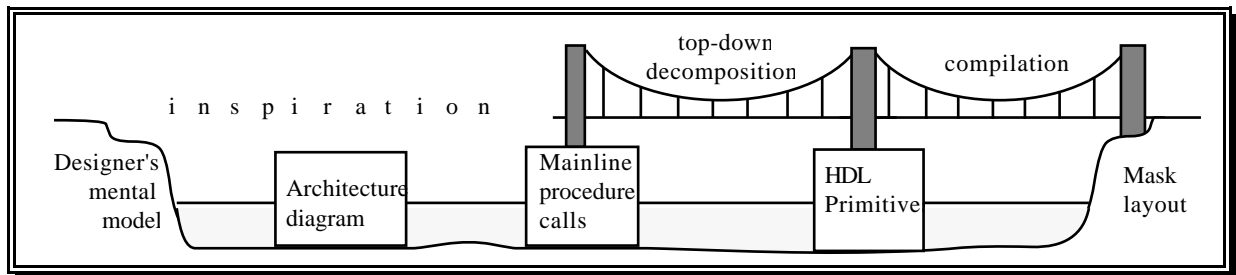


Figure 2. **Conventional HDLs fall far short of bridging the semantic gap**

Such a language would have a set of data-manipulating primitives closer to the designer's mental model, and furthermore, tools such as subroutines would allow the primitives to be built into structures like those in a designer's informal architecture diagrams. Thus both bridges over the semantic gap would stretch, reducing greatly the distance to be spanned by pure inspiration.

So the problem was defined, and the hunt was on for a suitable graphical representation which could be input to a Silicon Compiler. It would enable designers to render a design easily and naturally, using top-down structuring techniques. A diagram at any level would - with appropriately-named components - provide a consistent definition of a solution to a (sub)problem.

Both Structure Diagrams and Finite State Machines were rejected after investigation (Pearson, 1987). Subsequently interest centred on the Data Flow Diagram or DFD (DeMarco, 1978), a purely documentary tool developed to assist software engineers in synthesising software systems with complex structural and behavioural interactions. PICSIL is the result.

Conventional Data Flow Diagrams

A conventional DFD is a design representation for a software system. It has two major components, the *diagram proper*, and a *data dictionary*. The diagram shows a network of data-manipulating *processes* (shown as named icons) linked by *data flows* (represented by labelled arrows). *Data stores* (more, but different, named icons) can also be included. A DFD has a textual data dictionary, which contains a functional definition of some diagram components.

The work described here has developed DFDs into PICSIL, a direct-input language for the definition of digital systems, which can be compiled into a mask layout for a chip. It has not been possible to banish textual representations entirely from the language, but PICSIL's combination of diagram for architecture and text for behaviour bridges the semantic gap far more naturally than do conventional HDLs.

The PICSIL Notation

PICSIL is a language based on conventional DFDs. It has been adapted to describing digital hardware, and has a formal syntax, so it is suitable for direct input and editing on a computer and subsequent compilation into hardware.

Basic Components of PICSIL designs.

Figure 3 shows a screen dump of a contrived PICSIL design for a component of an image analysis system. It inputs two images as serial data, colour-processes each, and superimposes them. An updatable thresholding function may be applied to the resulting image stream. The device is used in the following sections to illustrate the four basic components of a PICSIL diagram. Other components will be introduced later.

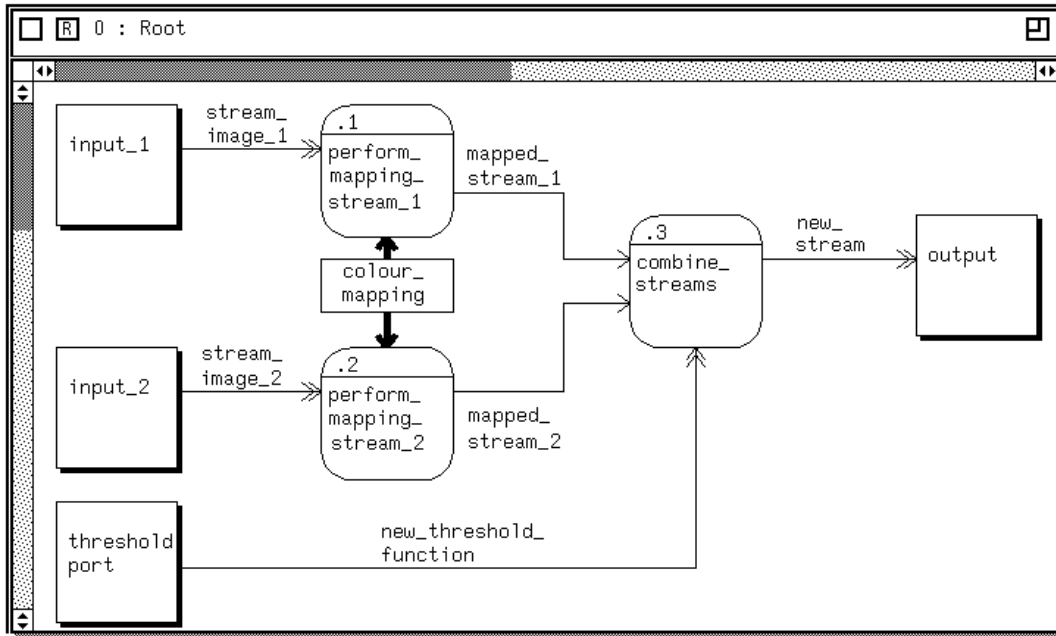


Figure 3. Screen dump of a simple PICSIL design (an image analysis device)

Data flows

A data flow is a pipeline carrying data packets between other elements of the PICSIL design. At the design level, it is a parallel connection, but the compiler may implement it as a serial link. The data dictionary entry for a flow describes the format of the information it carries.

—————> Discrete flows, such as `mapped_stream_1`, transfer data between processes. Data-driven communication protocols are used to ensure interprocess synchronicity. The protocol-handling hardware is provided automatically by the compiler.

—————>> Continuous flows such as `stream_image_1`, and `new_stream` transfer data between processes and external entities (see below). The interprocess synchronicity protocols do not apply to external signals; any synchronising must be provided by the designer.

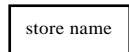
—————> Store flows are unnamed. They are used by a process to access information from a *data store*. As the stores are random access, the processes supply addressing information to locate the information. The information supplied by the store flows in the direction of the arrow, addresses travel in the opposite direction.

External Entities

entity name

 External entities are compiled into the chip's I/O pad drivers. They produce and consume only continuous flows. Hence `stream_image_1`, `stream_image_2` and `new_stream` are asynchronous *continuous flows*.

Data Stores



A data store is a random-access repository for data. In a completed PICSIL design all stores have a data dictionary entry which defines their contents. In the example, the two `perform_mapping` processes use the store `colour_mapping` to transform the input image to improve colour contrast. They access new colour values from the table using the intensity of each input pixel as an index.

Processes



A process transforms incoming flows into outgoing flows. In a completed PICSIL design every process must have a definition. A process defined by a Data Dictionary entry, or *mini-specification* (expressed in a language based closely on HardwareC (Ku and De Micheli, 1990), the HDL used by the OLYMPUS system), is called a *primitive process*; a process defined by a lower-level PICSIL diagram is called a *non-primitive process*.

Process refinement

Figure 4 shows the definition of the non-primitive process `combine_streams`; the input and output flows in this refinement are the same ones interfacing to process `combine_streams` in the top-level diagram. This reflects the macro-based nature of PICSIL's structuring tools.

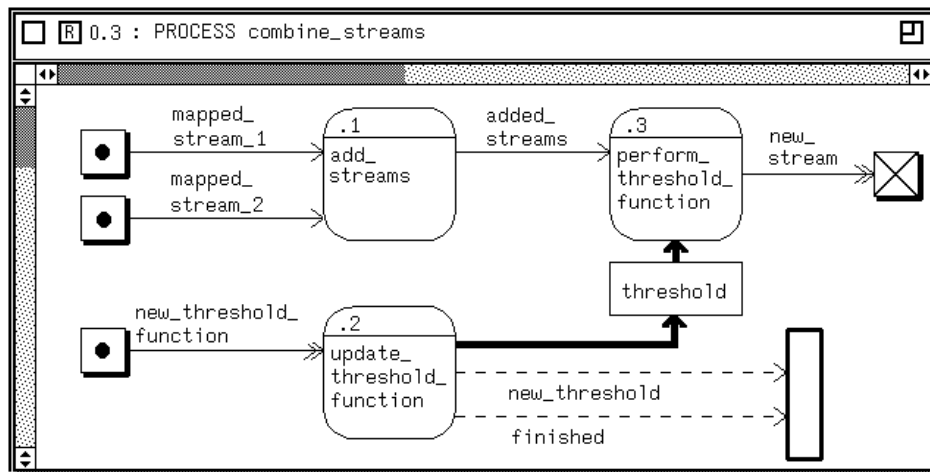


Figure 4. The refinement of process `combine_streams`

Import Links and Export Links



When a designer first opens a refinement window for a process to begin entering its definition, the editor shows all the data flows into and out of the process as import and export links, with the same names as the parent flows, in the refinement. No hardware is generated by the compiler for the links; their function is purely mnemonic. In the example device, the process `Update_threshold_function` accepts a new thresholding function from a flow attached to an import link and puts it into the store `threshold`. Note that the dotted arrows attached to `Update_threshold_function` are discussed below, in the section "The Control Extensions".

Figure 5 shows the data dictionary refinement for the primitive process `perform_threshold_function`, including its output flow, `new_stream`. This definition, together with all the other information implicit in the graphic representations, is combined with the graphical data into a HardwareC description for input to the OLYMPUS Silicon Compilation suite.

```

.3.2 : PROCESS perform_threshold_function
Process perform_threshold_function
{
  struct {
    boolean<0:7> red;
    boolean<0:7> green;
    boolean<0:7> blue;
  } pixel;

  pixel = receive(added_streams);      /*receive next pixel*/
  pixel.red = threshold[pixel.red];   /*threshold red pixel*/
  pixel.green = threshold[pixel.green]; /*threshold green pixel*/
  pixel.blue = threshold[pixel.blue]; /*threshold blue*/
  send(pixel, new_stream);           /*output new pixel value*/
}

```

Figure 5. The data dictionary entry for the primitive process `perform_threshold_function`

The Control Extensions

A complete system specification tool may need to turn on or off groups of processes under the influence events of internal or external to the chip. As conventional DFDs are intended to simplify the design of file manipulation programs, they are generally only a documentary tool, and detailed control specifications are often omitted entirely. However, DFD methodologies (Hatley & Pirbhai, 1987), (Ward, 1986) for some real-time systems include *control extensions* to allow explicit activation and deactivation of individual processes. PICSIL has been similarly extended, with *Event flows* and *Control processes*, to allow process activation to be specified separately from the data manipulation sections.

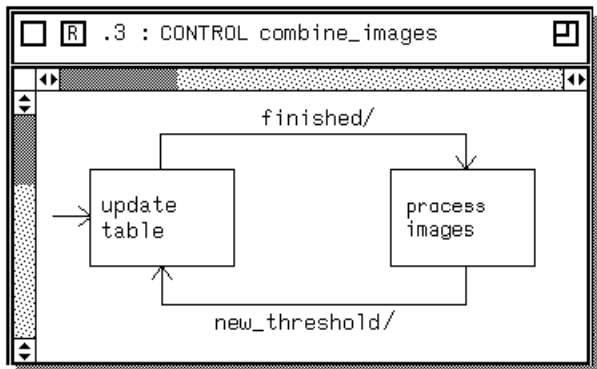


Figure 6. The refinement of a control process

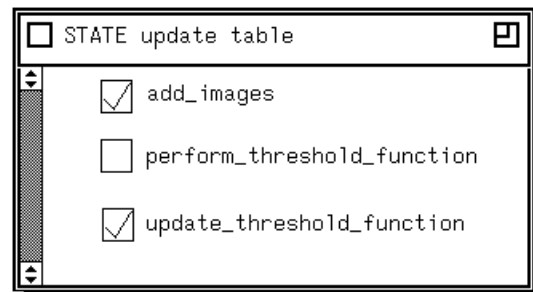


Figure 7. A state activation table

Event Flows

— — — — —> An event flow reports occurrences of an event occurring inside or outside a PICSIL design. Event flows can occur between any pair of: processes, external entities, and control processes.

Control Processes

Control processes explicitly activate and deactivate data processes, and coordinate events internal and external to the chip. The refinement of a control process takes the form of a Finite State Machine diagram (see figure 6). *Events* are represented in the FSM as labels before the slash on the arrows between the states. Events are status information reported to the Control Process along *event flows*. They cause state transitions when they become true and their arrow originates from the current state (states are represented by rectangular boxes in figure 6). *Actions* are represented as labels after the slash

on the state transition arrows. The FSM used for our example specifies no actions. Actions are signals sent by the FSM to processes or external devices when the state transition they are attached to occurs. Each state of the FSM has a *State Activation Table* as its refinement. This specifies a set (empty by default) of processes which are deactivated when the FSM is in that state. Figure 7 shows the deactivation of process `perform_threshold_function` (see also figure 4) during the threshold table update in order to prevent inconsistent thresholding.

Conclusion

PICSIL provides a more natural representation of the architectural aspects of digital systems than conventional, programming-language-oriented, HDLs. The language allows designers to specify a device's architecture in a graphic format based on Data Flow Diagrams and its behaviour textually, in a language based on HardwareC.

An editor (extant) and compiler (planned):

- provide a user interface which supports direct input and modification of a device's architecture diagram and behaviour specification,

- combine these into a HardwareC specification, and will

- supply the HardwareC specification to the OLYMPUS Silicon Compiler system for compilation into an IC mask layout.

A number of example systems have been represented using PICSIL. It has been found that the diagrams are self-documenting, and provide a complete specification of the system at any level of abstraction.

References

De Micheli, G. , Ku, D. Mailhot, F. & Truong, T. (1990): The Olympus Synthesis System, *IEEE Design & Test of Computers Magazine*, 7, 5, 37 - 53.

DeMarco, T. (1978): Structured Analysis and System Specification, Prentice-Hall.

Hatley, D. J. & Pirbhai, I. A. (1987): Strategies for Real-Time System Specification, Dorset House.

Johannsen, D.L. (1981): Silicon Compilation, *Ph.D.Thesis*, California Institute of Technology, Pasadena, California.

Johnson, S. C. & Mazor, S., (1984): Silicon Compiler Lets System Makers Design Their Own Chips, *Electronic Design*, 32, 20, 167 - 181.

Ku, D. & De Micheli, G. (1990): HardwareC - A Language for Hardware Design Version 2.0, *Technical Report CSL-TR-90-419*, Computer Systems Laboratory, Stanford University.

Pearson, M. W. (1987): Silicon Compilation - The shortcut to IC design without tears, *Proceedings Nelcon '87*, Auckland, 23 - 26.

Rubin, S. M. (1987): Computer Aids for VLSI Design, Addison Wesley.

Ward, P. T. (1986): The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing, *IEEE Transactions on Software Engineering*, SE12, 2, 198 -210