A Sophomore Course in Codesign

Madsen, Jan; Steensgaard-Madsen, Jørgen; Christensen, Lars Munk

Published in:
Computer

Link to article, DOI:
10.1109/MC.2002.1046983

Publication date:
2002

Document Version
Publisher's PDF, also known as Version of record

A Sophomore Course in Codesign

Jan Madsen, Jørgen Steensgaard-Madsen, and Lars M. Christensen, Technical University of Denmark

We teach a hardware and software codesign course to second-year students who have expressed an interest in either electronics or informatics (computer science). The course emphasizes concepts and methods that are useful to both hardware and software developers and in particular to developers of embedded systems who must consider both disciplines as well as their interaction. We consider the course to be part of a search for better development methods and hope to increase the number of professional developers.

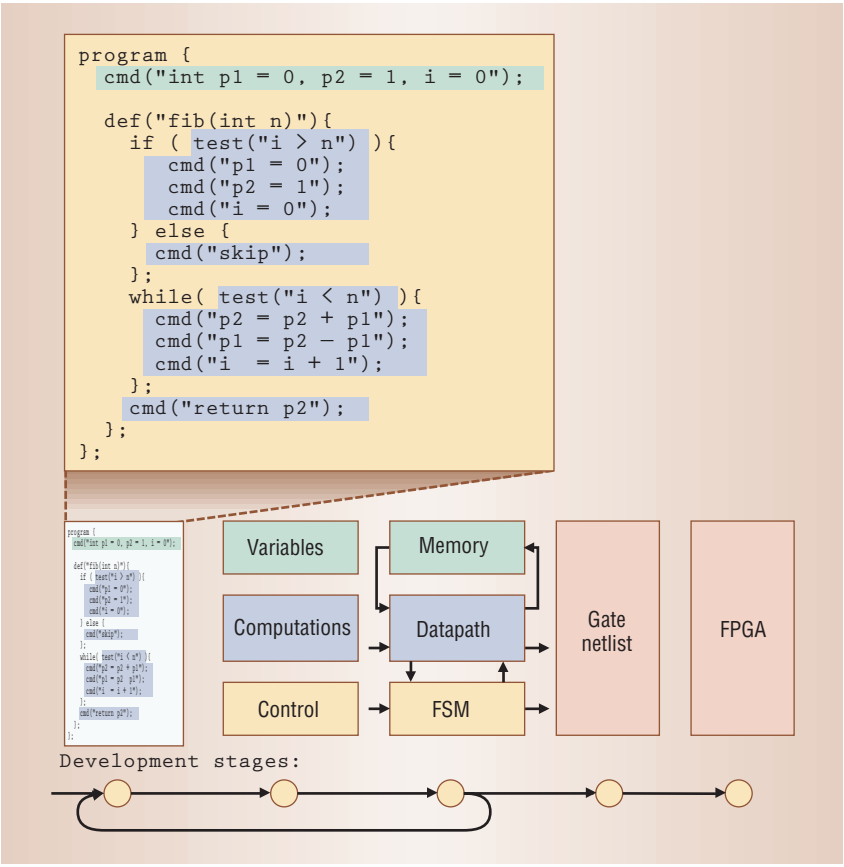

Figure 1. Algorithm for calculating the nth Fibonacci number and development stages for developing it from C code to implementation in a field programmable gate array.

WHY A SOPHOMORE COURSE?

As others, we do have a course in hardware/software codesign at the graduate level. However, we believe that introducing codesign at the sophomore level has three advantages:

- Students are exposed to codesign before they choose to specialize in either field.
- It motivates students to take both hardware and software courses to better meet the challenges of embedded systems design.
- Illustrating concepts in these two disciplines emphasizes the relation between abstract and concrete.

The philosophy underlying the course is that a function can be implemented in either software or hardware. The choice between the two is based on system requirements and measurable properties of the implementation. Hence, a central part of the course is that students not only must assess their design according to its functionality, they also must quantify properties of their design.

An important aspect of the course is that it uses examples such as the following from both hardware and software:

- software multiword addition versus a hardware *n*-bit ripple-carry adder,
- software pipe connections versus hardware signal connections, and
- software translation into assembly code versus hardware netlist synthesis from a model.

These examples help to illustrate concepts from these apparently different disciplines.

FROM HARDWARE TO SOFTWARE

The main approach is top-down. In one assignment, for example, we ask students to design hardware using an algorithm expressed in C.

To illustrate the process, let's consider the example of a simple, well-

known algorithm for calculating the *n*th Fibonacci number. The top part of Figure 1 lists the abstract program. The `cmd` and `test` tokens classify a string as either a nonbranching command or a test, respectively.

This program contains computation, control flow, and storage requirements. Students learn to convert such code stepwise into a hardware model. The ultimate goal is to implement the model with a field-programmable gate array. Students use SystemC, an open-source industry standard for system-level design (http://www.systemc.org), that spans hardware and software design from concept to implementation.

FSM WITH DATA PATH

Students know finite state machines and Boolean algebra from mathematics, but not the notion of a data path—that is, an architecture containing registers, simple operations, and their interconnections using buses or multiplexers. We introduce the general model of an FSM with data path (FSMD) early in the course and support it with a simple tool that translates an abstract program into an FSMD description.

Figure 2 shows partial results for the FSMD translation of the abstract program from Figure 1. Students can execute and test a corresponding concrete program version of the algorithm. The interpretation displays the FSM literally and the data path as labeled sections of code (for `A_S7` in Figure 2).

In this way, we introduce the students to the key concepts of basic blocks and control structure. The assembler code that a C compiler generates is another illustration. We use both concepts to prescribe systematic software test and to justify methods to prove program correctness.

The transition `S0 => S1` in Figure 2 lets us discuss the initialization that must take place before the component can react to external signals. The precise coupling of memory to the data path—in particular, the choice between registers and memory—is left for later development steps.

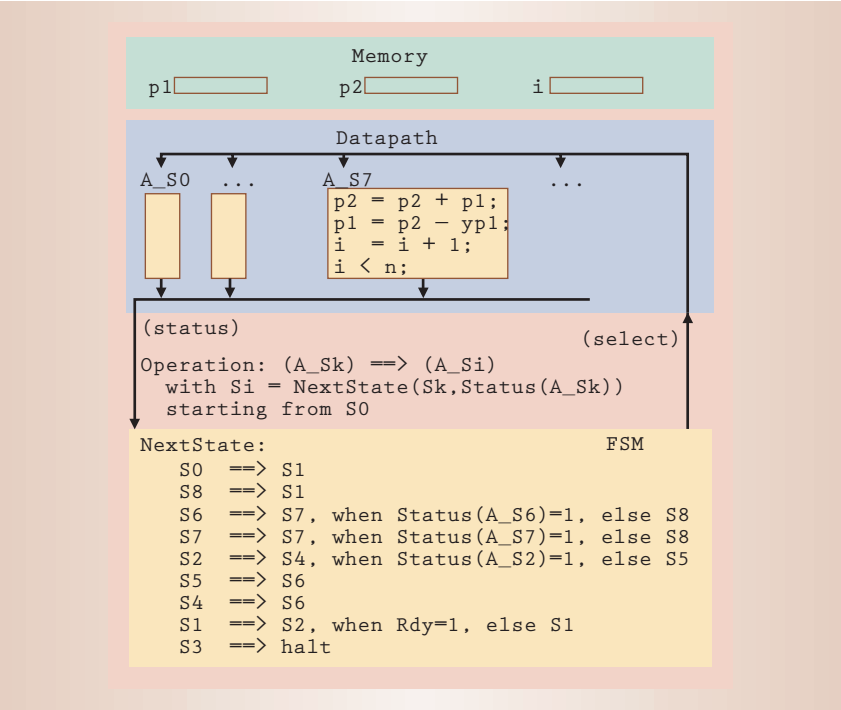

Figure 2. The Fibonacci code "compiled" into an FSM with data path. A simple tool translates the computation into basic blocks (in the data path). A finite state machine represents the control structure.

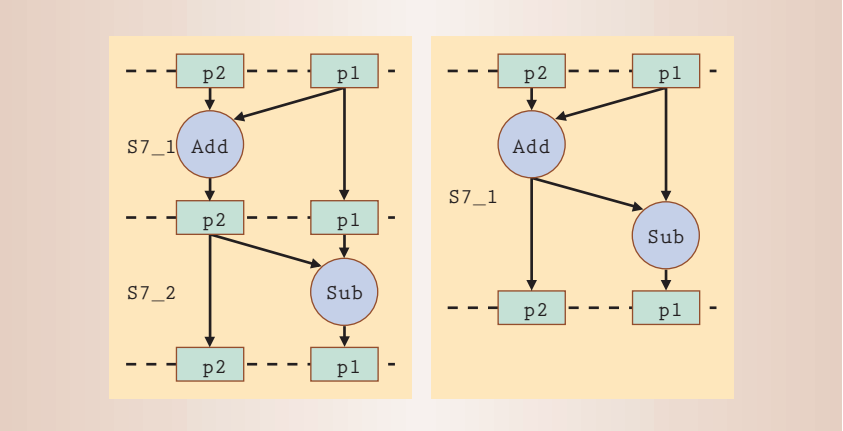

Figure 3. Optional scheduling operations in the data path for basic block A_S7 in Figure 2. The schedule on the left requires two cycles but only one arithmetic logic unit; the schedule on the right computes in one cycle but requires two ALUs.

Students use SystemC to write behavioral models for a circuit having data path operations identical to the basic blocks. We introduce various techniques for refining the models. After each refinement, the students simulate the design to validate it.

In the last step, the students refine the data path to a register transfer level, making design decisions that influence quantifiable properties. For example, Figure 3 shows two possible schedules of the computation of `p1` and `p2` in basic block `A_S7` in Figure 2. The

schedule on the left requires two cycles but only one arithmetic logic unit (ALU) because it executes the two operations (`Add` and `Sub`) in different cycles, allowing them to share the same resource—a single ALU. The schedule on the right computes in one cycle but requires two ALUs.

When they complete the design, students measure its speed, size, and power consumption and compare it with measures of the original pure software implementation.

During this course, students are faced with problems and facts that challenge their prejudices. The lessons they learn include the following:

- Running software on a Pentium processor can be faster than using dedicated hardware.
- Dedicated hardware is more power-efficient than general-purpose hardware.
- It is difficult to write assembler programs that are better than compiled C-code.
- Memory and time efficiencies are not always in conflict.

These experiences emphasize the value of using a scientific approach.

> Measure what is measurable, and make measurable what is not so.
>
> —Galileo Galilei

Jan Madsen is a full professor with primary interest in hardware-software codesign. Contact him at jan@imm.dtu.dk.

Jørgen Steensgaard-Madsen is an associate professor with primary interest in software. Contact him at jsm@imm.dtu.dk.

Lars M. Christensen is a research scientist with primary interest in software. Contact him at lmc@imm.dtu.dk.