RTeasy
An Algorithmic Design Environment on Register Transfer Level

Hagen Schendel, Carsten Albrecht, and Erik Maehle
Institute of Computer Engineering
University of Lübeck
{schendel, albrecht, maehle}@iti.uni-luebeck.de

Abstract

Current developer tools and HDLs for system design are powerful instruments and support a variety of abstraction levels but they are too complex for didactic purposes. This paper describes the RTeasy IDE, an algorithmic design environment on register transfer level that has been developed to provide a simple system-design tool for didactic purposes to be used e.g. in introductory courses in computer engineering and digital design. The RTeasy tool suite includes an HDL, a simulator and further design features. As an example, it is applied to the design flow of a shift-multiplier.

1 Introduction

Nowadays, the design of complex systems and the implementation of new architectures demands high-level tool support. Different abstraction levels such as the gate, the register, and the processor levels are used to define and describe the structure and behavior of new designs. Especially the gate level is well-supported. Current hardware description languages (HDL) such as VHDL [2], Verilog, and ABEL [3] are utilized. Their integrated development environments (IDE) provide programming support, gate-level simulation, and download opportunities to suitable devices. Unfortunately, these languages and tools require a high-level knowledge and experience of system design. For educational purpose and due to the tremendous number of functionalities most of the common tools are too difficult for beginners. In the introductory course on computer engineering [4] and its following lab course at our university, the register transfer notation (RTN) of John P. Hayes [1] is applied. Here, second-year students of computer science are taught the principles of digital systems and system design. Hayes' RTN allows them to create own hardware modules based on an algorithmic description. Especially in a didactic view this way of description is well chosen because of the similarity of high-level programming and the RTN. The IDE RTeasy backs the design flow with editor, parser and simulator for a variant of Hayes' RTN.

In the following the RTeasy tool suite is presented. Section 2 introduces the RTeasy HDL. Then, the RTeasy IDE is described in Section 3 and applied in a small design example in Section 4. Section 5 concludes the presented work.

2 RTeasy HDL

The RTeasy HDL is a register transfer language based on RTN. In the following section the basic modeling constructs of the RTeasy HDL are introduced. Their usage is demonstrated by arbitrary chosen parts of RTeasy HDL code. An RTeasy program consists of two parts: declarations of components and the program body containing a description of the algorithm. Registers, busses and memories are the provided components and can be declared as follows:

Registers and Busses are declared by an identifier in uppercase letters and the number and order of bits in brackets. In the brackets the left number of the colon stands for the most significant bit (MSB) and the right one stands for the least significant bit (LSB). The numbers represent the indices used in the program.

Example:

declare register A(7:0), STATUS(1:5), RDY
declare bus INTERNAL_BUS(7:0)

A is an eight-bit register where the MSB has index 7 and the LSB has index 0, STATUS is a five-bit register with MSB index 1 and LSB index 5, and RDY is a single-bit register which does not need any indices. INTERNAL_BUS is an eight-bit bus similar to register A. In contrast to registers, busses hold their data for only one clock cycle followed by a reset to 0. In hardware they are e.g. realized as signal lines with tristate drivers.

Memories are declared by an identifier followed by address and data registers enclosed in brackets. These registers must be declared beforehand.

Example:

declare memory MEM(AR,DR)

Here, AR is the address register and DR is the data register. The memory dimension depends on the size of the address register determining its address space and the size of the data register determining the memory-word width. Data transfer between data register and memory is triggered by the commands **write** MEM and **read** MEM.

The program body generally describes a finite state machine where each state includes some concurrent hardware operations given by data transfers between registers or combined registers performed directly or via busses. A simple timing model is used with each statement executed in exactly one clock cycle.

Each state or concurrent-command sequence is separated by a semicolon and has the following form:

[*label* :] *concurrent operations* ;

Note that a label can be left out. The semicolon can be interpreted as a state transition and indicates the end of a clock cycle. In general, the next state is the one after the semicolon. For the modification of the succeeding state an absolute branch command is provided:

goto *label*

It can be combined with the if-statement described in the following to get a conditional branch.

The concurrently executed RT operations, see below, per state are given by a comma-separated sequence.

Example:

BUS ← A + B, RESULT ← BUS

Note that only busses instantly take new values that are written on it in the *same* clock cycle, registers do not take their new values until the *end* of the clock cycle. The example above shows concurrently executed RT operations and the use of busses. The first RT operation writes the sum of A and B on the bus BUS. The second RT operation switches the bus signals to the input ports of the register RESULT. A trace of the input values of RESULT shows after a half cycle that first all bits are zero, then some hazards follow, and finally the result of A + B appears. In fact the observed values are the same as the output of the adder circuit belonging to the expression A + B, delayed by the bus delay.

In the following all statements which can build up a concurrently executed statement sequence are shown:

Conditional Statements have the form:

if *expression* **then** *RT operations*
[**else** *RT operations*] **fi**

The conditions are evaluated one half cycle ahead of the RT operations they account for. So, in general the evaluation bases on the global state of all registers and busses at the end of the preceded clock cycle.

The **if**-statements can be used wherever an RT operation may occur. They may be nested to any arbitrary level as this is only a conjunction of conditions easily realizable by AND gates.

Expressions are used to model computations in a similar way to high-level programming languages. Here, the operands are registers, busses and bit-word constants instead of variables and constants. Parts of registers or buses and single bits can be combined to *combined bit-words* using the dot operator.

Example:

A ← A(6:0).A(7)

The example shows a left rotation by one bit of the 8-bit register A. Combined bit-words are handled like normal registers or busses because they are only arbitrary combinations of signals in hardware. Bit-word constants are either positive decimal, binary led by % or hexadecimal led by $.
The set of operators includes the binary operators $+, -, <, <=, >, >=, =, <>$, and, nand, or, nor, xor and unary operators $-$ (sign) and not with usual precedences. The arithmetic operators + and − generate a carry bit, thus, the result is one bit wider than the widest operand. All other operators deliver a result being as wide as the widest operand. Missing bits of the smallest operand are extended by leading zeros. For arithmetic operations registers, busses and combined bit-words are processed right-aligned. Logical operators operate bitwise on bit-words.

Example:

```
A <> B # 1 if A and B not equal
%1 + 7  # %1000 or 8
not B   # B is bitwise negated
```

Comments can be inserted wherever it is necessary by # as a line comment.

Register Transfer Operations are written as

combined bit-word ← *expression*

It is not allowed to transfer data from bus to bus so that a bus can only be used on the left or on the right side in an RT operation. This is due to the fact that busses are realized as signal lines. The triggering of an RT operation results in signal propagation from the circuit belonging to the expression to the input ports of the combined bit-words elements. When the triggering edge of the operation units occurs the registers read their input signals and save them for the next cycle. When an RT operation writes on a bus, the signal would be propagated through other circuits representing expressions using the bus.

On many points in the execution of algorithms further computations depend on values evaluated shortly before, i.e. in the same clock cycle.[1] The application of RTeasy constructs defined above does not allow to use the results of any RT operation in the conditional expression of a conditional statement in the same concurrent statement sequence. The conditional expressions are evaluated utilizing the global state of the *preceding* clock cycle. So, there is no chance to realize a conditional branch by a goto-statement embedded in an if-statement if the conditions need the values of the *same* clock cycle. The general solution is a bulky conditional branch in the next state where the first operation of the branch is included. In the example below, the first program code consumes two clock cycles because the conditional branch is performed in a separate cycle although it would be possible to avoid it. This problem occurs frequently so that the introduction of a handy notation for these conditional branches is worthwhile. That is why RTeasy provides an additional separator: the pipe symbol. Now, a state can be of the form:

[*label* :] *concurrent operations* | *conditional branch* ;.

The *conditional branch* which is an **if**-statement only including **goto**-statements may use the results of the RT operations on the left side of the pipe operator symbol. So, the pipe operator saves one clock cycle per each step of the loop, see the program example. For internal processing or hardware implementations, the pipe symbol is expanded to equal but bulky statements.

Example:

The RTeasy program

```
LOOP: COUNTER ← COUNTER + 1;
      if COUNTER < 20 then
        goto LOOP
      else
        # do something
      fi;
      # go on
```

can be refactored by

```
LOOP: COUNTER ← COUNTER + 1
      | if COUNTER < 20 then goto LOOP;
      # do something
      # go on
```

The command **nop** must be used to indicate that no RT operations should be triggered. It can be used to describe a state with no signal output.

[1] This has also been taken into account during the design of the C programming language regarding the `++i` construct.

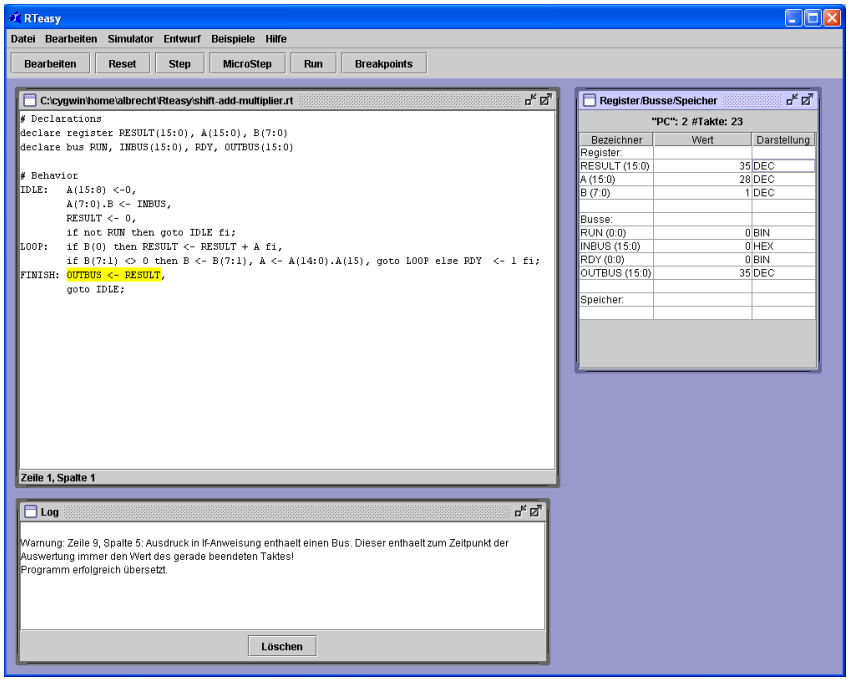


Figure 1: Stepwise Simulation in RTeasy

3 RTeasy IDE & Simulator

The design concept of the IDE of RTeasy resembles an assembler or embedded systems IDE. Actually, the only thing still missing is an opportunity to download developed designs to target devices. It provides a text editor with usual functionalities and a basic help system. The IDE has two working modes, editing and simulation. After launching RTeasy, the system is always set in editing mode where programs can be written, loaded, or saved. The simulation mode is entered by clicking the 'Simulate' button. Then the system performs syntactic and semantic analysis on the contents of the editor window which is the upper window on the left side in Figure 1. In case of a successful analysis, the simulation status window which is the upper one on the right side in Figure 1, pops up. It contains all declared registers, busses and memories in the sequence of their declaration. The full memory contents can be shown in a separate window. Each register and bus is depicted and attributed with its dimensions and current value. The contents of registers, busses, and memory cells may be shown in one of five modes: binary, decimal, signed two's complement decimal, hexadecimal and signed two's complement hexadecimal. All values of registers, busses, or memory cells can be interactively changed by the user. The simple concept avoids complex layout problems as they would occur by graphical representations such as RT-level block diagrams and nevertheless displays all relevant information.

Simulation Capabilities

The user controls the simulator by some buttons well known from other simulation environments. There are 'Reset', 'Step', and 'Run/Stop' with the expected functionality and the 'MicroStep' button explained below.

One 'Step' begins at each simulated clock cycle with the triggering edge of the control unit. Note that our RT designs can be split up into two units, the control unit representing the algorithmic behavior and the operational unit including all hardware components such as registers, busses, memories and arithmetic and logic units. The simulated state is marked in the editor window, the upper left one in Figure 1, by a colored background and the contents of the status window shows the values the registers, busses and memories have taken at the end of this clock cycle. The values shown on

busses are reset at the end of each clock cycle because the control unit emits other control signals at the beginning of the next cycle.
The 'Run' button launches a continuous execution of the 'Step' simulation and changes its caption to 'Stop' so that the next click aborts the infinite simulation. Simulation may also end when the last state is executed without a **goto**-statement or the program quits because of the **goto end**-statement. 'MicroStep' provides a detailed view to the concurrent execution of RT operations. Although they are executed concurrently this feature allows the traversal through the states on an operation-by-operation base. During the traversal the currently executed RT operations are marked yellow whereas conditions not met are marked with magenta background. The contents of the status window is consecutively updated as well ignoring their concurrency. If the operations would be executed in the order of appearance in the program 'MicroStep' and 'Step' simulation would differ and even the first one would not hold because busses might not reach their final values. Thus, all bus writes are simulated first.
In addition the IDE provides breakpoints which are useful for debugging purposes.

Design Tools

Beyond modeling and simulating features RTeasy IDE provides design tools for further system development. These features are gathered in the 'Design' menu. They include extraction of control and conditional signals and model expansion. The extraction of control and conditional signals defines the input and output of the control unit. RT operations are simply assigned with numbers representing their triggering control signal lines. Equal RT operations will be assigned to the same number. Conditional signals are extracted from **if**-statements. Boolean expressions that contain expressions of other types such as arithmetic ones are splitted, nested **if**-statements are flattened by combining the boolean expressions. This flattening is necessary to build up the state transition table of the control unit. Furthermore, conditional signals that only occur together in state transition tables can be merged by optimizations. Model expansion unfolds the right side of each pipe symbol and merges nested **if**-statements to approach the description of a synthesizable finite state machine.

Currently, the opportunities of the 'Design' menu will be enlarged by student research project. The goal of this work is a function that generates VHDL code for control and operation unit. The exported code can be used and simulated by other tools such as Mentor Graphics' FPGA Advantage [5].

4 RTeasy Example

The example shown in Figure 2 is a simple shift/add multiplier. Its interface consists of two 16-bit busses, an incoming (INBUS) and an outgoing one (OUTBUS), and two signals (1-bit busses): RUN (in) and RDY (out). Furthermore, the model makes use of two 16-bit registers A, and RESULT and one 8-bit register B. The underlying algorithm is quite simple:

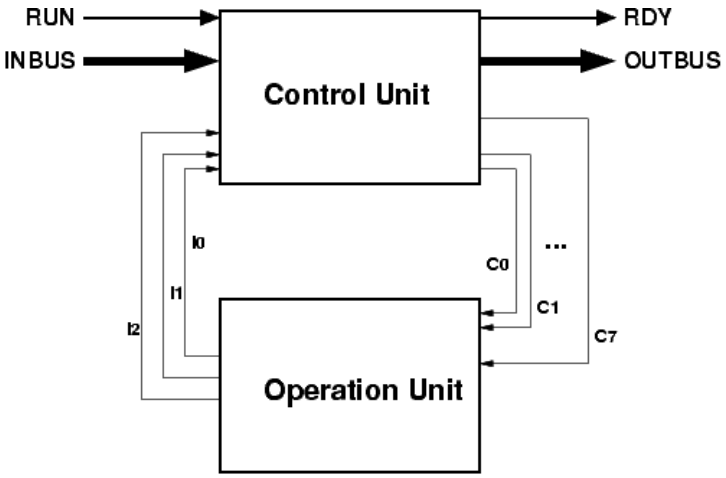
1. As long as RUN is not set, read both factors from INBUS. The first factor, which takes the higher 8 bits, is transferred to the lower part of register A. The second one, stored in the lower 8 bits, is put to register B. The higher 8 bits of A and RESULT are initialized by zero.

2. For each bit k in B, beginning at the LSB with index 0, add $2^k \cdot$A to RESULT if $B_k = 1$.

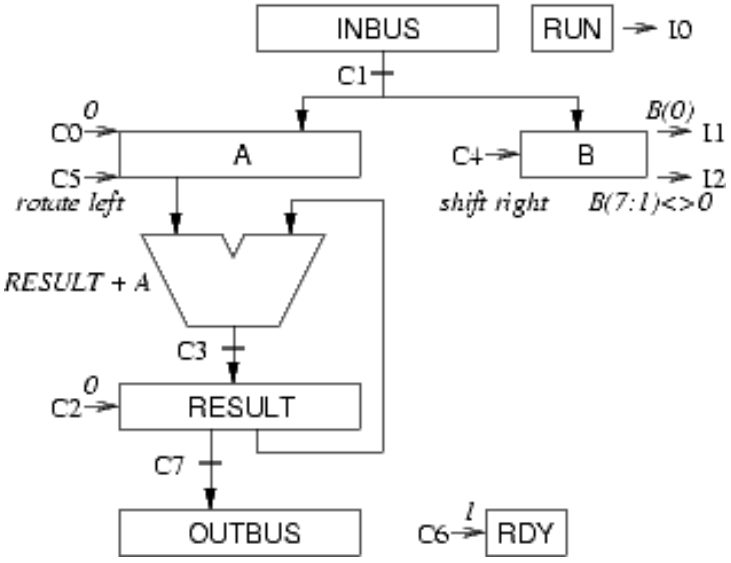
3. During the last iteration RDY is set to 1 and in the next cycle the contents of RESULT is written on OUTBUS.

The addition of $2^k \cdot$A is realized by left-shifting of A. The test $B_k = 1$ is realized by right-shifting B and testing the LSB. The loop is aborted if the remaining part of B does not contain any bit with value 1.
The screenshot in Figure 1 shows that RTeasy identifies 7 unique RT operations and 3 conditional expressions. The conditional expressions are mapped to input signals for the control unit and the RT operations to output signals to affect the behavior of the operation unit. The block diagram of Figure 2(b) and the listing of Figure 2(c) are attributed with these signals. Figures 2(a) shows the composition and interconnection of the two units.

This example design contains three kinds of RT operations with different effects shown in the block diagram. Simple data transfer operations such as C1 and C7 that only transfer data from one entity to another. Then there are operations on registers such as shift, rotate, and set/reset operations (C0,C2,C4,C5,C6). The RT operation triggered



(a) System diagram.



(b) Block diagram of the operation unit.

 # Declarations

declare register RESULT(15:0), A(15:0), B(7:0)
declare bus RUN, INBUS(15:0), RDY, OUTBUS(15:0)

 # Behavior

IDLE:	A(15:8) ← 0, A(7:0).B ← INBUS, RESULT ← 0,		C0,C1,C2
	if not RUN **then goto** IDLE **fi**;	I0	
LOOP:	**if** B(0) **then** RESULT ← RESULT + A **fi**,	I1	C3
	if B(7:1) <> 0 **then** B ← B(7:1), A ← A(14:0).A(15), **goto** LOOP	I2	C4,C5
	else RDY ← 1 **fi**;	I2	C6
FINISH:	OUTBUS ← RESULT, **goto** IDLE;		C7

(c) RTeasy program code describing algorithmic behavior and usage of components.

Figure 2: Design example of a shift/add multiplier.

by C3 is a special one. It involves an adder circuit which is represented by the V-shaped symbol in the block diagram.

The extracted information is used to generate a Moore or Mealy state machine for the control unit. These state machines can be minimized by well-known techniques and implemented utilizing their optimized switching functions taken from the state transition table, one-shot circuits, or a modulo sequencer.

5 Conclusion

In this paper, an algorithmic design environment on register transfer level is presented. The tool bases on a modified version of the RTN introduced by John P. Hayes [1]. The IDE is implemented using Java and works fine on different platforms such as Solaris, Linux and Windows 2000/XP. In the last winter term, its usability was tested and proven by the application in the introductory course of computer engineering. Second-year students of computer science quickly accepted the tool and easily applied the IDE at home and at the university to solve exercises and to deepen their acquired knowledge of system design. Moreover, they have supported the development of RTeasy with critical remarks and useful proposals for improvement.

In contrast to previous paper designs it was know possible for them to test and debug their algorithms with the simulator. Furthermore, the simulation proved to be very helpful in introducing basic RT algorithms in the lectures replacing "hand simulations" on the blackboard.

Beside the generation of VHDL code, the future work includes the extension of simulation features as well as more design tools. It is planned to include the generation and viewing of VCD (Value Change Dump) traces of register and bus values into the IDE. Additionally, a capability of unit tests should be implemented to support the proofreading of student exercises and automatic testing of complex designs.

References

[1] John P. Hayes. *Computer Architecture and Organization*. McGraw-Hill, 3rd Edition, 1998.

[2] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, 1995.

[3] Lattice Semiconductor Corporation. *ABEL-HDL Reference Manual*. Reference Manual, DSNEXP-ABL-RM Rev 8.0.2, 2003

[4] Erik Maehle. *Technische Grundlagen der Informatik*. Course script, Institute of Computer Engineering, University of Lübeck, Germany, 2003.

[5] Mentor Graphics Corporation. *FPGA-Advantage*. Datasheet, 2003