Generating Compilers for Generated Datapaths *

M. Held, M. Glesner
Darmstadt University of Technology
Institute for Microelectronic Systems
Karlstr. 15
64283 Darmstadt, FRG
Phone (+49)6151/16–5136, Fax (+49)6151/16–4936
email: michael@microelectronic.e-technik.th-darmstadt.de

Abstract

Modern CAD systems allow the designers to come up with powerful programmable datapaths in a very short time. The time to develop compilers for this datapaths is much longer. This paper presents a new approach to compiler generation. We show how a VHDL description of a programmable datapath can be analyzed to extract several informations for compiler generation. The analysis finds computing and storage resources, classifies signals as control or data, and extracts all the possible micro operations for this datapath.

1 Introduction

Synthesis of synchronous datapaths has been one of the hot research topics for the last years. Several algorithms for scheduling, allocation, module binding, high level transformations, etc. have been presented and discussed in the literature. Special synthesis scripts for dedicated target architectures have been developed. All of those systems transform a given algorithm down to silicon. The generated datapath and its associated controller are more or less well suited for the execution of the given algorithm. While high level synthesis systems are still under development, the lower level or logic synthesis and optimization systems gain more and more acceptance in industry. VHDL with its different levels of abstraction is widely used in all stages of a design from specification down to gate level netlists of optimized random logic.

1.1 Rapid Prototyping

However, designing a special purpose datapath and controller still needs lot of time and man power. To prevent long and costly fabrication runs rapid prototyping systems based on field programmable gate arrays (FPGAs) became popular. But even those rapid prototyping systems or "ASIC emulators" result in a fixed architecture which is suitable for the given algorithm only. If the algorithm has to be changed, one has to redesign the datapath and/or the controller.

*The published work is supported by Deutsche Forschungsgemeinschaft under contract SFB 214 "IMES" (Integrated Mechano Electronic Systems)

1.2 ASPPs

The architecture can be made more versatile if the datapath is programmable. This leads to ASPPs (Application Specific Programmable Processors). Such datapaths may consist of several ALUs, registers, I/O ports, barrel shifters, or special purpose functional units. Other possible components may be memory structures like FIFOs, ring buffers, register files, or dual port RAMs. There even may be controller components like stacks, stack pointers, program counters, loop register/counters. All of those components can be connected by busses or multiplexors. The control signals of all the components (selection signals of multiplexors, enable signals of tristate drivers, function selection signals of ALUs, load signals of registers, etc.) may be concatenated to form one big control word. This leads to a VLIW like structure which has the potential parallelism and versatility to fulfill the requirements of computation intensive tasks. Different algorithms can be executed on such a structure by applying sequences of control words. The generation of this microcode is a very strenuous task and should be left to compilers.

1.3 Microcode Generation

The micro programmed controllers of the first micro processors have been the reason for microcode generation and microcode programming. The first microcode was written and optimized by hand and resident in a microcode ROM. With the advent of writable microprogram storage devices and the possibility for user specific microcode there was a need for microcode generation and compaction.

The atomic operations of a microprogram are called *micro operations*. By combining several micro operations into *micro instructions* horizontal microcode is generated. Several algorithms for microcode compaction have been published. See e.g. [2] for local microcode compaction algorithms. More sophisticated global compaction algorithms are mostly based on Fishers Trace Scheduling [4]. The improvements concentrate on the reduction of computation time and better trace selection (e.g. [9], [17],[18], [6] and [13]). A good overview on global microcode compaction can be found in [16].

The most promising optimizations are on frequently executed loops. This may have been the main drawback of Fishers trace scheduling. Fisher worked with *DAGs* (Directed Acyclic Graphs) to represent the micro operations

and their dependencies. Therefore he could not model loops. He proposed to compact loops without their loop-back edge and treat the resulting compacted part as one node *(loop representative)* in a hierarchical DAG. Many algorithms especially for loop optimization have been proposed. In terms of hardware synthesis they are based on *loop unrolling*. The software (or microcode) people talk of *software pipelining*. See e.g. [17] (URCR – UnRoll, Compact, Reroll), [19] and [12] (URPR – UnRoll, Pipeline, Reroll), [3] (pipeline scheduling), [18] (GURPR – Global URPR), [10] (based on URPR) and [20] (GURPR*). A good overview is given by [7] and [8].

1.4 Retargetable Code Generation

The reasons mentioned above require compilers which can be easily adapted to new target architectures. The first step to achieve this goal has been the splitting of compilers into a target independent frontend and target dependent backend. The frontend generates code for a virtual machine, the backend interprets this code and generates code for the target architecture. This *interpretative code generation* requires a new backend for each new target architecture. The *pattern matched code generation* uses a graph or tree structured machine description in the backend. The code is generated by pattern matching on this machine description (e.g. in MIMOLA [11] [14] [15]). However there is still a target independent backend which is inefficient for different architectures (think of RISC, CISC, vector, super scalar, VLIW architectures). Modern code generators therefore generate code in several phases (like register allocation, storage allocation, loop optimizations, instruction selection, delayed branch optimization). Each part of the compiler corresponding to the different phases is generated by a code-generator-generator. The input is a machine description file which not only contains informations on the possible instructions but also on storage layout, functional units (pipelines, parallelism), etc. For an overview see [5].

2 Our Approach to Compiler Generation

We try to set up a system which generates a C-compiler for a given VHDL description of a datapath. The output of this compiler will be microcode for this datapath. This microcode will be interpreted by a dedicated controller which will be generated too. This paper focuses on the first step of the compiler generation. The basic information needed for a compiler generator are the possible micro operations which can be executed on the datapath. We also need to know which control signals have to be applied to perform a given operation. Last not least we are interested in the necessary resources. This information will make a later micro code compaction possible.

2.1 Modeling the Datapath

Our approach to generate a compiler starts with a given VHDL description of a programmable datapath. As an example consider fig. 1. There are three functional units and three registers connected by several busses. The control word shown on the right consists of six control bits for the multiplexors (two bits each), three enable signals for the registers, and one control bit for the ALU.

The components of this datapath are modeled in behavioral VHDL. The “ALU” (which can only add or subtract) for example can be described like in fig. 2. Note that the level of abstraction is very well suited for modern design

```
architecture Behave of ALU is
begin
  process (a, b, mode)
  begin
    case mode is
      when '1' =>    o <= a + b;
      when '0' =>    o <= a - b;
    end case;
  end process;
end;
```

Figure 2: Behavioural VHDL Description – ALU

systems. On one hand it is simple enough to be handled by synthesis systems like e.g. Synopsys Design Compiler. On the other hand our tool is able to extract possible micro operations (the addition and the subtraction) and control signals (signal ‘mode’) from it. Registers are inferred by either wait statements or clock sensitive processes. In fig. 3 for example a positive edge triggered register will be inferred.

```
architecture Behave of REG is
begin
  process (clk)
  begin
    if (clk = '1') then
      o <= i;
    end if;
  end process;
end;
```

Figure 3: Behavioral VHDL Description – Register

2.2 Extracting the CDFG of the Datapath

The next step in our approach is to extract the control/data flow graph (CDFG) of the given VHDL description. We make use of the VTIP (VHDL Tool Integration Platform) toolbox from COMPASS for this purpose.

VTIP provides tools to transform VHDL descriptions to an internal data structure and a C-language interface *SPI (Software Procedural Interface)* to work on it. VTIP currently provides two possible views of a VHDL description:

- The *VHDL View* is an attributed syntax tree of the VHDL description. It is possible to do transformations on this tree (optimizations) and convert it to a VHDL description again.
- The *Synthesis View* is composed of more abstract objects like State Machines, Logic Networks and Black Boxes. The control and data flow graph are explicitly available in this representation.

For our purpose we use the Synthesis View only. All the objects in VTIP are realized as C-structures. We use graphical representations to show how our example is transformed to the synthesis view. In fig. 4 you see the top level object of our small example. Note that this a typical representation of a structural VHDL description. Each component

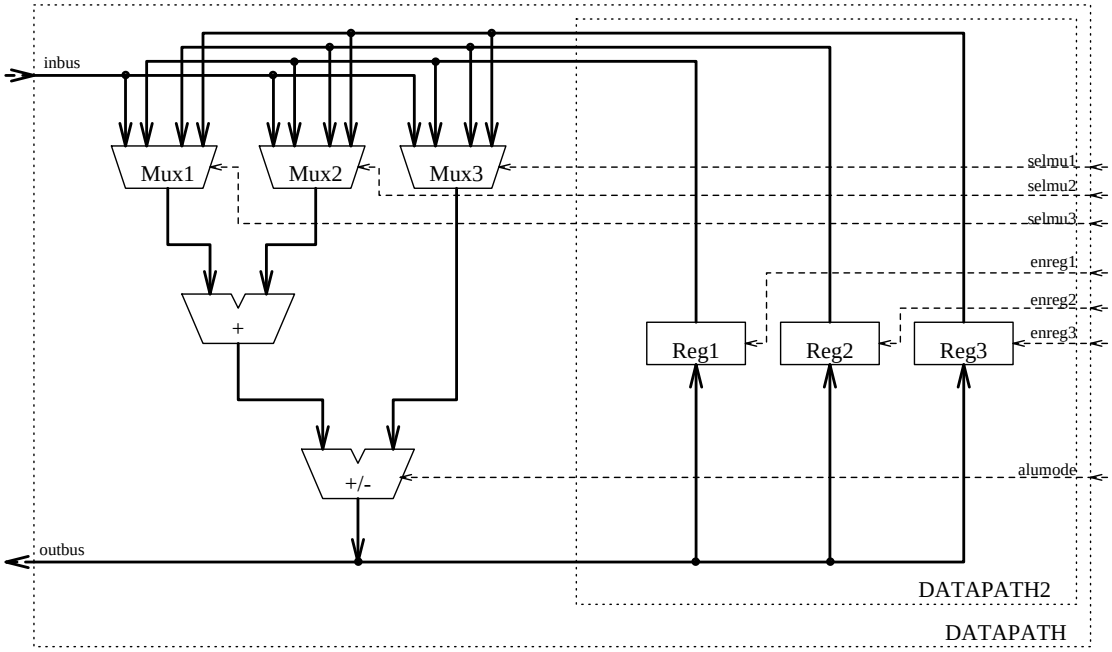

Figure 1: Programmable Datapath

instantiation results in a BlackBox. The BlackBoxes represent hierarchy: the BlackBox `datapath2` contains more BlackBoxes – the three registers. Fig. 5 shows the ALU in the Synthesis View representation. There are special fork and join nodes which represent the control flow. Only one of the two conditional logic blocks is executed depending on the input of the fork op. We can identify control signals as input to control nodes. By backtracking them to the top entity it is possible to define a control word. All of the nodes in this representation are atomic, i.e. there is no more level of hierarchy. By looking at the operations on the datapath we can extract the micro operations Add and Sub. Finally fig. 6 shows the register representation. The important node is the TimingOp. Associated with the TimingOp are attributes which give the signal name and the condition for "firing" this operation. In this case the attributes help us to find the clock signal (clk).

2.3 Extracting Micro Operations

Our tool traverses the synthesis view and associates several attributes to the different nodes:

1. Signals are categorized in different groups:
   (a) Control signals are determined as the inputs to the known control flow operators. E.g. `case` and `if` statements result in fork and join operators. Inputs to this operators are classified as control signals. All the control signals are concatenated to a single control word.
   (b) Clock signals are determined by the attributes of the TimingOps.
   (c) All the other signals are classified as data signals.
   (d) Signals with more than one driver are classified as busses.
   (e) Signals with exactly one driver and one sink are classified as "transient" registers. This signals can appear as a sink to micro operations. However the stored result has to be used by another micro operation in the same micro instruction.
2. Storage units (registers and the transient registers) are marked as such.
3. From every input of a storage unit we trace back the data signals to possible sources. This is equivalent to searching all possible paths from a storage units input to any storage units output. The functional units passed on this path give the micro operation. To perform this micro operation it is necessary to set the following bits in the control word:
   - The enable signals of the involved registers.
   - The selection signals of multiplexors on the path.
   - The control signals of ALUs on the path.
4. For every micro operation we note
   - The name of the operation. Up to now we can identify the following: `move, and, not, nand, nor, xor, equal, notequal, less, notless, greater, notgreater, add, sub, neg, abs,`

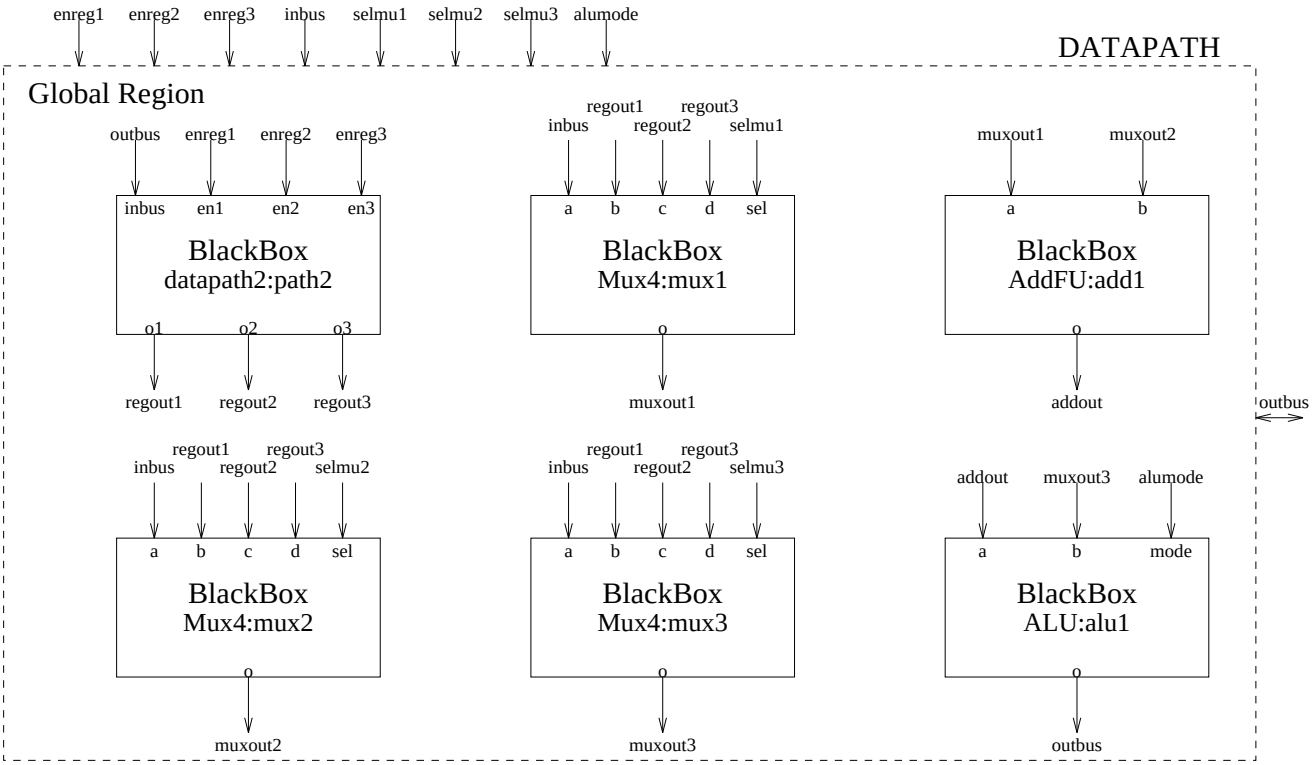

Figure 4: Synthesis View: top level entity

`mul`, `div`, `remain`, `modulo`, `shl`, `asr`, `shr`, `rol`.

- A list of input registers.
- The output register.
- A list of necessary resources (ALUs, multiplexors, busses).
- The – incompletely specified – control word.

2.4 Results

The result of our example is given in fig. 7. The lines marked with a colon show the classified control lines and how they make up the control word. E.g. bits 3 and 4 of the control word are the control lines for multiplexor 1 (selmu1). The lines with the greater sign show the classified I/O ports of the whole circuit. The rest of the lines are the possible micro operations.

3 Conclusion and Future Work

We have shown how a VHDL description can be analyzed to extract possible micro operations on this hardware. The prototype system shows the feasibility of this approach. However there are some modifications necessary:

- The type (and bit width) of the data signals is not taken into account. However the information on this is already in the synthesis view.
- Status words are not taken into account. E.g. an adders carry output or a comparators output should be concatenated to a status word.
- The classifying algorithms are weak in some cases and often require a certain VHDL style.

The next step will be the transformation of this description to a machine description suitable for a compiler generator. Some more informations will be necessary including informations concerning the controller.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] Scott Davidson, David Landskov, Bruce D. Shriver, and Patrick W. Mallett. Some Experiments in Local Microcode Compaction for Horizontal Machines. *IEEE Transactions on Computers*, C-30(7):460–477, July 1981.

[3] Kemal Ebcioğlu. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In *Proc. 20th Annual Workshop on Microprogramming (MICRO–20)*, pages 69–79, Colorado Springs, CO, December 1–4 1987.

[4] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[5] Mahadevan Ganapathi, Charles N. Fisher, and John L. Hennessy. Retargetable Compiler Code Generation. *Computing Surveys*, 14(4):573–592, December 1982.

```
: 0 0 enreg1
: 1 1 enreg2
: 2 2 enreg3
: 3 4 selmu1
: 5 6 selmu2
: 7 8 selmu3
: 9 9 alumode
> I inbus
> IO outbus
> I enreg1
> I enreg2
> I enreg3
> I selmu1
> I selmu2
> I selmu3
> I alumode
(sub,(rreg-behav_7_2,taddout),toutbus,(alu-behav_14,mux4-behav_11),011XXXXXXX)
(sub,(rreg-behav_6_2,taddout),toutbus,(alu-behav_14,mux4-behav_11),010XXXXXXX)
(sub,(rreg-behav_4_2,taddout),toutbus,(alu-behav_14,mux4-behav_11),001XXXXXXX)
(sub,(tinbus,taddout),toutbus,(alu-behav_14,mux4-behav_11),000XXXXXXX)
(add,(rreg-behav_7_2,taddout),toutbus,(alu-behav_14,mux4-behav_11),111XXXXXXX)
(add,(rreg-behav_6_2,taddout),toutbus,(alu-behav_14,mux4-behav_11),110XXXXXXX)
(add,(rreg-behav_4_2,taddout),toutbus,(alu-behav_14,mux4-behav_11),101XXXXXXX)
(add,(tinbus,taddout),toutbus,(alu-behav_14,mux4-behav_11),100XXXXXXX)
(mov,toutbus,rreg-behav_4_2,,XXXXXXXX1)
(mov,toutbus,rreg-behav_6_2,,XXXXXXX1X)
(mov,toutbus,rreg-behav_7_2,,XXXXXX1XX)
(add,(rreg-behav_7_2,rreg-behav_7_2),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX1111XXX)
(add,(rreg-behav_7_2,rreg-behav_6_2),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX1110XXX)
(add,(rreg-behav_7_2,rreg-behav_4_2),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX1101XXX)
(add,(rreg-behav_7_2,tinbus),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX1100XXX)
(add,(rreg-behav_6_2,rreg-behav_7_2),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX1011XXX)
(add,(rreg-behav_6_2,rreg-behav_6_2),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX1010XXX)
(add,(rreg-behav_6_2,rreg-behav_4_2),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX1001XXX)
(add,(rreg-behav_6_2,tinbus),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX1000XXX)
(add,(rreg-behav_4_2,rreg-behav_7_2),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX0111XXX)
(add,(rreg-behav_4_2,rreg-behav_6_2),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX0110XXX)
(add,(rreg-behav_4_2,rreg-behav_4_2),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX0101XXX)
(add,(rreg-behav_4_2,tinbus),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX0100XXX)
(add,(tinbus,rreg-behav_7_2),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX0011XXX)
(add,(tinbus,rreg-behav_6_2),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX0010XXX)
(add,(tinbus,rreg-behav_4_2),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX0001XXX)
(add,(tinbus,tinbus),taddout,(addfu-behav_12,mux4-behav_10,mux4-behav_8),XXX0000XXX)
```

Figure 7: Result

[6] Michael A. Howland, Robert M. Mueller, and Philip H. Sweany. Trace Scheduling Optimization in a Retragetable Microcode Compiler. In *Proc. 20th Annual Workshop on Microprogramming (MICRO–20)*, pages 106–114, Colorado Springs, CO, December 1–4 1987.

[7] Reese B. jones and Vicki H. Allan. Software Pipelining: A Comparison and Improvement. In *Proc. 23rd Annual Workshop and Symposium MICRO 23 Microprogramming and Microarchitecture*, pages 46–56, Orlando, Florida, November 27–29 1990.

[8] Reese B. Jones and Vicki H. Allan. Software Pipelining: An Evaluation of Enhanced Pipelining. In *Proc. 24th Annual International Symposium on Microarchitecture (MICRO–24)*, pages 82–92, Albuquerque, New Mexico, November 18–20 1991.

[9] Joseph L. Linn. SRDAG Compaction – A Generalization of Trace Scheduling to Increase the Use of Global Context Information. In *Proc. 16th Annual Microprogramming Workshop (MICRO–16)*, pages 11–22, Downingtown, PA, October 11–14 1983.

[10] Joseph L. Linn and Cy D. Ardoin. An Example of Using Pseudofields to Eliminate Version Shuffling in Horizontal Code Compaction. In *Proc. 22nd Annual International Workshop on Microprogramming and Microarchitecture (MICRO–22)*, pages 172–180, Dublin, Ireland, August 14–16 1989.

[11] Peter Marwedel. A Retargetable Compiler for a High-Level Microprogramming Language. In *Proc. 17th Annual Microprogramming Workshop (MICRO–17)*, pages 267–274, New Orleans, Louisiana, October 30 – November 2 1984.

[12] Robert A. Mueller, Bogong Su, Michael R. Duda, and Brian L. Plomondon. A Case Study in Signal Processing Microprogramming using the URPR Software Pipelining Technique. In *Proc. 19th Annual Workshop on Microprogramming (MICRO–19)*, pages 104–115, New York, NY, October 15–17 1986.

[13] Alexandru Nicolau. Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies. *IEEE Transaction on Computers*, 38(5):663–678, May 1989.

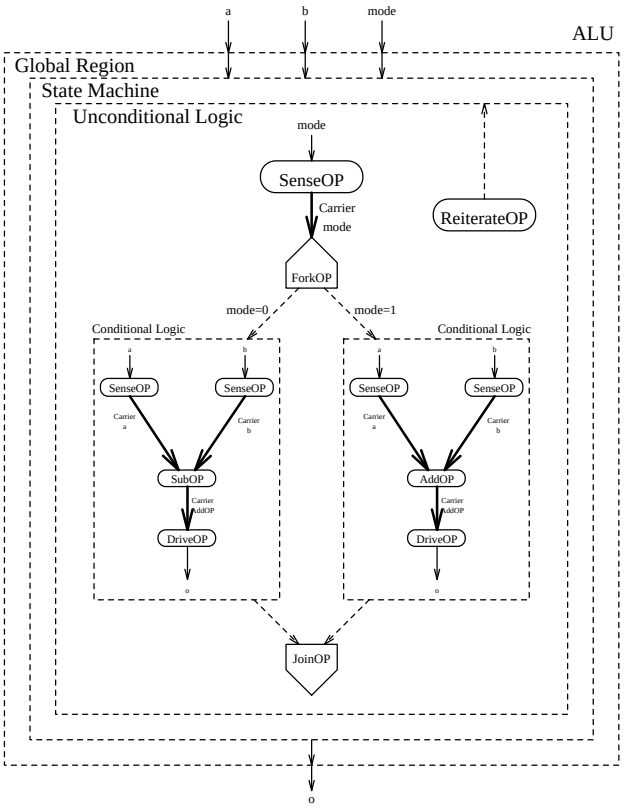

Figure 5: Synthesis View: ALU

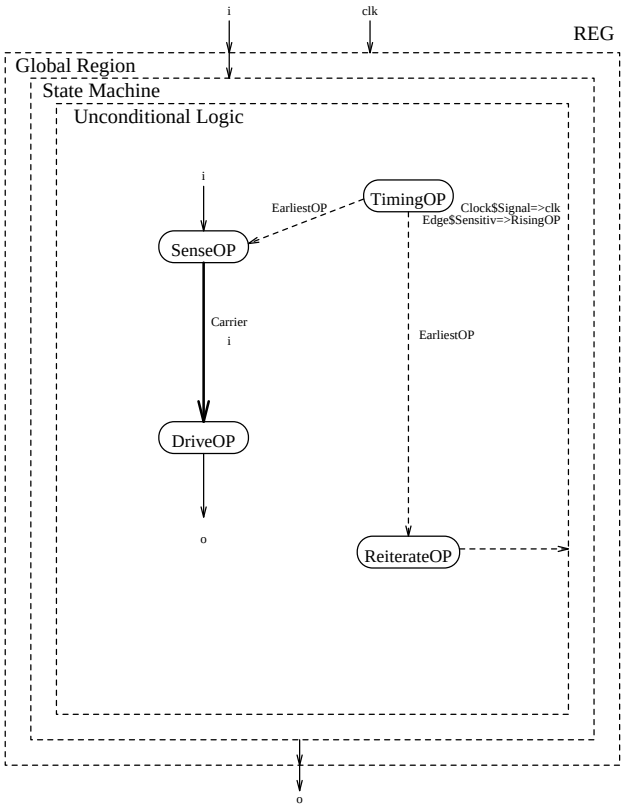

Figure 6: Synthesis View: Register

[14] Lothar Nowak. Graph Based Retargetable Microcode Compilation in the MIMOLA Design System. In *Proc. 20th Annual Workshop on Microprogramming (MICRO–20)*, pages 126–132, Colorado Springs, CO, December 1–4 1987.

[15] Lothar Nowak and Peter Marwedel. Verification of Hardware Descriptions by Retargetable Code Generation. In *Proc. 26th ACM/IEEE Design Automation Conference*, pages 441–447, 1989.

[16] Bogong Su and Shiyuan Ding. Some Experiments in Global Microcode Compaction. In *Proc. 18th Annual Workshop on Microprogramming (MICRO–18)*, pages 175–180, Pacific Grove, CA, December 3–6 1985.

[17] Bogong Su, Shiyuan Ding, and Lan Jin. An Improvement of Trace Scheduling for Global Microcode Scheduling. In *Proc. 17th Annual Microprogramming Workshop (MICRO–17)*, pages 78–85, New Orleans, Louisiana, October 30 – November 2 1984.

[18] Bogong Su, Shiyuan Ding, Jian Wang, and Jinshi Xia. GURPR – A Mthod for Global Software Pipelining. In *Proc. 20th Annual Workshop on Microprogramming (MICRO–20)*, pages 88–96, Colorado Springs, CO, December 1–4 1987.

[19] Bogong Su, Shiyuan Ding, and Jinshi Xia. URPR – An Extension of URCR for Software Pipelining. In *Proc. 19th Annual Workshop on Microprogramming (MICRO–19)*, pages 94–103, New York, NY, October 15–17 1986.

[20] Bogong Su and Jian Wang. GURPR: A New Global Software Pipelining Algorithm. In *Proc. 24th Annual International Symposium on Microarchitecture (MICRO–24)*, pages 212–216, Albuquerque, New Mexico, November 18–20 1991.