DAGAR: AN AUTOMATIC PIPELINED MICROARCHITECTURE SYNTHESIS SYSTEM

Vijay K. Raj
Computer Science Engineering Department
University of Texas at Arlington
P.O.Box 19015, Arlington, Tx-76019

ABSTRACT
This paper presents a Behavioral Synthesis system which can design pipelined microarchitectures.

INTRODUCTION AND PREVIOUS WORK
This paper presents an automated microarchitecture synthesis system called DAGAR [1]. DAGAR takes as input a behavioral description of a digital system and outputs a microinstruction (MI) sequence and a data path. In microarchitectures synthesized by DAGAR, the functional units (FUs) may take one or more clocks (i.e., MIs) to perform an operation. These we call multiclocked FUs. Additionally, if these FUs can accept new data while processing data from a previous MI, we call them pipelined FUs. It will be seen that the use of pipelined FUs typically decreases the number of FUs in the data path without a time penalty. There are many systems which perform the translation from a behavioral description to a microarchitecture. Nearly all these systems [3], [4], [5] assume that each microinstruction is executed in one clock. The Shewa system [7] also synthesizes a pipelined data path. However, their model of a pipeline is a cascading of FUs whereas DAGAR deals with the partitioning of a single FU into stages.

THE MICROARCHITECTURE MODEL
The Model of the Data Path
The model of the data path in DAGAR consists of registers, buses and FUs. Data traverses from a register via a bus to a FU. Then, when the FU has completed its computation, the output data is stored in a register, again via a bus. In the DAGAR system, this process may take one or more clocks. These are detailed in [1] and [8].

The Model of The Functional Units
An FU can perform one or more operations such as "OR", "ADD", or "MPY" (multiply). These operations may each be single clocked, multiclocked or pipelined. The model of these FUs is shown in Figure 1. Each FU has been divided into Stages (shown by dotted lines). One Stage represents one clock period (i.e., one MI). Figure 1 shows three FUs. The first (FU1) can perform 2 operations, "OR" and "AND". These take one clock each and hence this FU is not pipelined. FU2 can perform 2 operations, "ADD" and "SUB" (subtract). These operations can be accomplished in 2 clocks and this unit is also not pipelined. FU3 does a "MPY" (multiply) operation in 3 clocks and is pipelined.

DAGAR treats multiclocked units as pipelined units which cannot accept new data on each clock. Single clock operations units are a one stage pipeline. Hence DAGAR uses the same algorithm irrespective of the type of FUs used in the data path. The number of clocks an operation unit takes and whether it is pipelined are defined by the user of the DAGAR system and this information is entered into DAGAR's data base.

The Model of the Control Unit
DAGAR generates microprogrammed control units. These have been described in [1] and [8] and are almost standardized. Here, we define mainly our extension to the control unit to facilitate multiclocking and pipelining.

Figure 2 shows our basic idea for representing MIs which may be single clock or multiclocked or pipelined. Each row in the table of Figure 2 represents one MI (i.e., one clock cycle). Each rectangle in the table represents a microoperation (MO). We use for this explanation, the three FUs of Figure 1. In MI 1 (i.e., the first row) we start three MOs, namely "OR", "ADD", and "MPY". The "OR" completes execution in MI 1 itself. In MI 2, a new operation "AND" has been started. The "ADD" is still in progress (in the second clock of two clocks) and this FU cannot accept new input. The "MPY" operation is still in progress as indicated by the vertical line. Though it is possible to start
a new "MPY" operation in this MI, we do not. In MI 3, we start a "SUB" operation as the "ADD" has completed (in MI 2) and we also start a new "MPY" operation (which will complete execution in MI 5). However, the first "MPY" operation will output data at the end of MI 3 and hence we must specify its output bus and register in addition to starting a new MO in this FU.

**THE INPUT TO DAGAR**

The input to our system will be a behavioral description. This behavioral description is translated to a data flow graph which is an intermediate representation. This data flow graph is then translated to a microarchitecture. Currently, the language to graph translator is under development and should be completed in a few months. At present we use a hand generated data flow graph as the input to our system. A sample behavioral input into our system is shown in Figure 3 (this is the essential part of the integer restoring division algorithm from [10]).

**THE TRANSLATION PROCESS**

In the first step, behavioral description is translated into a graph. Then the graph is converted into an initial MI sequence which makes the following assumptions: 1) infinite hardware is available in the data path 2) each operation takes one clock. This step uses an "as soon as possible" scheduling algorithm, resulting in the fastest control sequence. We call this control sequence an Abstract Machine, as it does not have a data path associated with it. The "as soon as possible" control sequence requires an unnecessarily wide data path (for example, it may put two shifts in one MI). So, in the next step, the DAGAR system searches the Abstract Machine for MIs where there is more than one MO of the same type. It then attempts to move one of these MOs to another MI. An important point to note is that these routines do not add any new MIs to the control sequence. Hence, DAGAR retains the "as soon as possible" MI sequence but reorganizes the MOs within them to reduce the total number of operations that would be used in the data path when it is designed.

Another optimizing routine estimates the minimal number of FUs required by the data path in order to achieve the "as soon as possible" schedule. This routine proposes to the user the grouping of the operations into FUs in order to have a minimum number of FUs. This routine essentially reduces the number of bits in an MI word and helps merge operations such as add and subtract. The user now has the option of starting from the maximally parallel data path and asking DAGAR to search the design space by specifying fewer and fewer data path components (adders, registers etc.). This is done in the next step where the binding of the MIs to data path components takes place. The two modules which do this are the DPD (Data Path Designer) [9] and the Scheduler. The Scheduler does the control allocation and the DPD does the data path binding.

**The Scheduler**

The Scheduler traverses the abstract machine one MI at a time and passes the MI to the DPD. This consists simply of traversing the MI list. The DPD tries to bind the the MOs of the MI to a data path. If it succeeds, the Scheduler gives the next MI etc. until all MIs are exhausted. If the DPD fails at any MI, it informs the Scheduler which tries to delay an MO to the next MI. This process is repeated until a binding is found or it is not possible to bind even one MO to the data path. In the latter case, the user of DAGAR is informed that he should allow DAGAR to use more hardware in the data path.

The Scheduler gives the DPD one MI at a time. Hence, if the DPD binds an MO to a multicycle or pipelined functional unit, the scheduler stores this information and returns it to the DPD so that it can correctly mark the progress of the MO through the functional units. When the DPD binds an MO to a multicycle unit, the Scheduler may have to move the successors of that MO to another MI. This is because the Abstract Machine assumed that each operation was completed in the same MI. This could result in a successor node in the next MI which will now have to be moved to another MI.

**The Data Path Designer**

The DPD works with a data base of hardware components specified by the user, the MI that it is currently binding and the data path built up to this point (due to the previous MIs). The DPD first binds those MOs which are continued from the previous clock (i.e. multicycle and pipelined operations). It then applies heuristic to bind the remaining MOs to the data path. If it is successful, it passes this information to the Scheduler. If it fails, it tries to pick up the necessary components from the data base. If the data base is exhausted, then the DPD reports failure to the Scheduler whose actions in this context have been described above.
EXPERIMENTS
We ran the input of Figure 3 through DAGAR, varying the following parameters in the data path: (a) number of FUs used in the data path and (b) number of FUs that are (i) multiclocked and (ii) pipelined.

Clock Rates
Our specification (to DAGAR) of the number of clocks taken by each operation were on the assumption that adds, subtract and comparison operations take 150 ns; shifts take 100 ns and all other operations in this example take 50 ns.

Figure 4 shows the control sequence for our example if we assumed a 8 MHz clock. Note that all the operation units used in this example would compute in one clock cycle. The first MI of Figure 4 shows the assignment of constants to registers and the second MI shows the first instruction inside the loop of Figure 3. It shows 4 microoperations (one per line). Each has the format: register, bus = FU#(operation: source1 source2). Here, source is a register, bus pair. Jump MIs have an entry in the JUMP column. The result of a test is stored in a register (in some earlier MI. See MI 6 and 7 of Figure 4) and allowed into the multiplexer of the control unit (CU) in the jump MI. For a more detailed explanation, see [8]. The important point is that there are 11 MIs. Assuming only one iteration through the loop (this is our assumption for all the examples in this paper. If the number of iterations increase, the results would favour DAGAR), we estimate 1375 ns to execute the behavioral specification. Next, we assume a 20 MHz clock with two multiclocked units and we got 20 MIs for a time of 1000 ns. Then, we assumed that the functional units were pipelined and ran the example through DAGAR and got a time of 950 ns.

CONCLUSIONS
We have presented a system that can design microarchitectures with multiclocked and pipelined functional units. Using it, our experiments show (Figure 5) that using single clocked functional units, one can get a limited speed up even with a large number of FUs. However, by using multiclocked functional units one can achieve the maximum speed. Further by using pipelined functional units, one can get the maximum speed even while limiting the number of functional units. This is because each pipelined functional unit can accept new data on each clock. Additionally, this decreases the number of buses in the data path. For our example, we found that the number of input buses used in the pipelined data path was halved because the number of functional units was halved. Hence, pipelined functional units can be used to reduce the size of the data path, while retaining the speed advantage of a horizontal microinstruction sequence.

ACKNOWLEDGEMENTS
Parts of this research as reported in [8] were done at the University of Illinois at Urbana-Champaign.

REFERENCES
repeat
    temp = mask and q ; Rotate by 1 using 
    temp = temp shr 1 ; operation shr
    a = a shr 1
    a = a or temp
    a = a - m
    q = q shr 1
    if (a < 0)
        begin
            q (nml) set 0 ; set bit nml to 0
            a = a + m
        end
    else q (nml) set 1 ; set bit nml to 1
    count = count + 1
until count = n

Figure 3: Behavioral Description