An Asynchronous Architecture Model for Behavioral Synthesis

Jordi Cortadella Rosa M. Badia

Dept. of Computer Architecture. Polytechnic University of Catalonia.
Gran Capità, s/n. Mòdul D4. 08034 - Barcelona
e-mail: jordic@ac.upc.es

Abstract

In this paper, an asynchronous architecture model for behavioral synthesis is presented. The basis of the model lies in a distributed control structure consisting of multiple communicating processes. Data processing is performed by self-timed modules. Signal transition graphs (STGs) are used to specify the behavior of the control processes. By using existing synthesis procedures for STGs, circuits based on the presented architecture model are proved to be realizable and hazard-free.

1. Introduction

Behavioral synthesis aims at the automatic generation of structural descriptions from behavioral descriptions [1]. Every synthesis system has a target architecture model where to map high-level objects (i.e. operations, data structures) into hardware objects (i.e. adders, registers, control units).

Until now, only synchronous architectures have been proposed for behavioral synthesis systems. However, their performance is highly reduced, as systems become larger, by the margins required for clock skew and worst-case delay times in the duration of the control step. Asynchronous systems completely avoid these problems.

The effort in the area of asynchronous systems has been mainly focused on the automatic synthesis of asynchronous finite state machines [2][3]. In [4] and [5] STGs are used to describe the behavior and synthesize control circuits. The contributions presented in [6] and [7] aim at the generation of delay-insensitive circuits from high-level specifications by syntax-directed translation according to production rule sets. To our knowledge, the only asynchronous architecture model proposed for behavioral synthesis has been presented in [8], in which the synthesis system compiles ISPS descriptions into a bus-based asynchronous architecture.

In this paper, we present an asynchronous architecture as a target model for behavioral synthesis (section 2). The work is mainly focused on the generation of the distributed control (section 3), based on the specification of an STG for each control process (section 4). Finally, the feasibility of the control circuits is studied by analyzing their unique state coding and hazard-freeness (section 5). Further details of the proposed architecture model can be found in [9].

Work supported by the Ministry of Education of Spain (CICYT TIC89-0300 and TIC91-1036)

2. Asynchronous architecture model

The data-path of our model is based on the utilization of self-timed blocks with two binary handshake signals: *request* (input) and *completion* (output). Both signals follow a four-phase handshake protocol for each operation performed by the block [10].

Three types of blocks are distinguished: computation blocks (adders, multipliers, ALUs, etc), multiplexors, and registers. Registers are implemented as latches and the *request* signal acts as a *load* signal. A latching completion detection mechanism similar to the one used in [6] has been considered. In the case of multiplexors, only the validity of the selected input data is required for a correct operation when the *request* signal goes high [5].

2.1. Distributed control

A centralized control unit would neglect some of the attractiveness of asynchronous systems. Global signals introduce delays that reduce the potential parallelism inherent to asynchronous systems. On the other hand, the number of states of the control unit grows exponentially with the number of control signals [4].

For our architecture, we propose a decentralized approach that lies in implementing the overall control as a set of communicating control processes (CPs), one for each data-path block. Each CP communicates locally with the block under its control through the *request* (*r*) and *completion* (*c*) signals and additional control signals required by the block (i.e. *operation code* for an ALU, *selection* signal for a multiplexor).

CPs communicate to each other through pairs of signals complying a four-phase handshake protocol. Each

0-8186-2645-3/92 $3.00 © 1992 IEEE

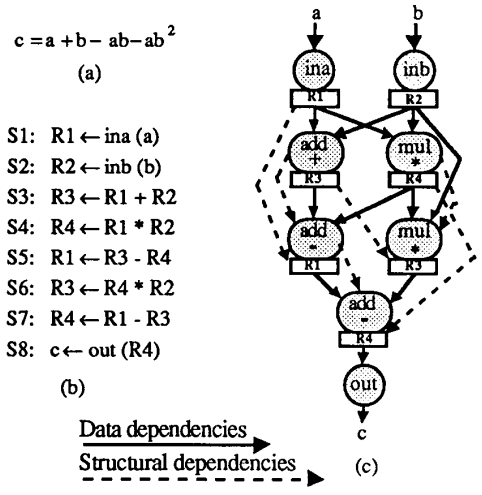

Figure 1. (a) Behavioral description
(b) RTL description
(c) Scheduled Data Flow Graph

pair of signals corresponds to one of the inputs or outputs of the controlled block. For each input to a data-path block, its corresponding CP has a *valid input* signal (*vi*) and a *consumed input* signal (*ci*). The former indicates to the CP when the input data is valid. The latter is generated by the CP indicating that the data-path block no longer requires the input data. Reciprocally, two handshake signals exist for each output data: *valid output* (*vo*) and *consumed output* (*co*).

3. Architecture synthesis

For the synthesis of an asynchronous architecture, we will assume that *scheduling* and *module binding* have already been performed, and a structural description of the data-path has been generated. The essential contribution of this work is the synthesis of the CP for each data-path block. The synthesis procedure receives a *scheduled data flow graph* (SDFG) as input and generates the *signal transition graph* (STG) of each CP.

An SDFG is a data flow graph with the information required for data-path and control synthesis: dependencies, module binding, and scheduling order of the RTL sentences. Figure 1 shows the example used along the paper to illustrate the synthesis of CPs. Scheduling and module binding have been performed by using two computation blocks: an adder/subtracter (*add*) and a multiplier (*mul*). Input and output data blocks (*ina*, *inb*, and *out*) are considered as computation blocks from the point of view of control generation. The dashed arcs introduced in the SDFG indicate structural dependencies and force the scheduling order of all the RTL sentences using the same computation block or register. The execution model assumes for each sentence that operands are always read from a register and the result stored into a register.

3.1. Definitions

An RTL sentence of the type *Ri = Rj op Rk uses* registers *Rj* and *Rk* and *defines* register *Ri*.

For each computation block *B*, its *sentence list SL(B)* is defined as an ordered list of all the RTL sentences executed by the computation block, according to the scheduling order specified in the SDFG. Its *output set OS(B) is defined* as the set of registers defined by sentences belonging to *SL(B)*.

Similarly for registers, *SL(Ri)* is defined as an ordered list of all the RTL sentences that define Ri and *OS(Ri)* as the set of computation block operands that use Ri (B_k will denote the k-th operand of block *B*).

Here we have some examples from Figure 1:
SL (add) = (S3,S5,S7); OS (add) = {R3,R1,R4}
SL (R4) = (S4,S7); OS (R4) = {add_2,mul_1,out_1}

For each register Ri and sentence $S \in$ SL(Ri), *use (Ri, S)* is defined as the set of computation block operands that use the value in Ri defined by sentence *S*. The calculation of *use* is similar to the calculation of *use-def* chains required for code optimization in compilers [11].

In the example, R3 is defined by sentences S3 and S6, but each definition has a different set of uses:
use(R3,S3)={add_1} (R3 is used by add_1 in S5)
use(R3,S6)={add_2} (R3 is used by add_2 in S7)

Given a register Ri and a computation block operand B_k, *first_use (Ri, B_k)* is defined as the set of sentences that use Ri as the k-th operand of block B for the first time after each definition. Similarly, *last_use (Ri, B_k)* is defined for the last use of Ri as the k-th operand of B.

In the example, R1 is defined by S1 and S5. After each definition, the first use of R1 by operand add_1 is produced in S3 and S7 (*first_use*(R1,add_1)={S3,S7}, in this case first_use and last_use coincide). Here we have some more examples:
first_use(R2,mul_2) = {S4}; last_use(R2,mul_2) = {S6}
first_use (R3,add_1) = {S5}; last_use(R3,add_1) = {S5}
first_use (R3,add_2) = {S7}; last_use(R3,add_2) = {S7}

3.2. Interconnectivity between control processes

Input/output signals of a CP fall into two categories:

- Local signals to its own block: *request* and *completion* signals (<r, c>), and control signals required by the block.
- Synchronization signals from/to other CPs: pairs of *valid* and *consumed* data signals.

Each CP has as many pairs of <vi, ci> signals as input data items are received by the block controlled by the CP. It also has as many pairs of <vo, co> signals as blocks receive the output data of its controlled block.

Figure 2 depicts the external interface of the CPs controlling the computation block *add* and its input multiplexors.

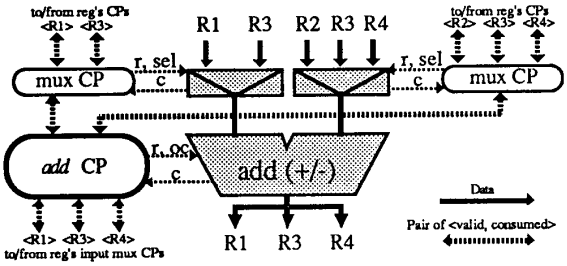

Figure 2. CPs for block *add* and its input muxes.

4. Control synthesis

Control is synthesized by defining the STG [4] corresponding to each CP of the architecture. It is not in the scope of this paper the synthesis of asynchronous circuits from STGs. The reader is referred to existing approaches [4][5][12] that can be used to synthesize the CPs from their STGs.

An STG is structured as a sequence of S-STGs (sentence-STG), each one corresponding to one of the sentences belonging to the *sentence list* of the controlled block (for multiplexors, the *sentence list* corresponds to that of the block at their output). Every S-STG defines the transitions required for its corresponding sentence.

4.1. S-STG for a computation block

The sequence of transitions in the S-STG for a computation block are the following (figure 3.a):

- The CP waits for the inputs and operation code[1] to be valid (vi+ and oc←op).
- Computation is initiated and completed (r+ → c+).
- vo_{Ri} is activated to denote that the input to the destination register Ri is valid.
- co_{Ri}+ indicates the completion of the storage into Ri the computation block must now be reset for another operation).
- The last part completes the handshake for each pair of signals of the CP: <r, c>, <vi, ci>, and <vo_{Ri}, co_{Ri}>.

[1]The operation code (oc) is a vector of signals (oc_1,...,oc_r). Transitions must be generated only for those signals that must change their value. This is denoted by the transition oc←op. The same applies for signal *sel* in the multiplexors (transition sel←@Rk).

4.2. S-STG for a register

The sequence of transitions of the S-STG corresponding to sentence *S* and register Ri is depicted in figure 3.b:

- The register waits for its input to be valid (vi+).
- Latching cannot be initiated until all the operands (j1,..,jm) using the previous value in the register need not it any longer (co-). Then latching is initiated and completed (r+ → c+).
- A four-phase handshake is performed for each of the operands (i1,..,ik) that belong to *use (Ri, S)*: vo+ → co+ → vo- → co-. When none of the computation block operands requires the register's value any longer, the storage of the next value can be initiated.
- In parallel with the previous step, the register is reset (r- → c-) for a new latching operation (the register output value remains unchanged when the load signal (r) is low). Furthermore, input data handshake is completed when data stability at the latch input is not longer required (ci+ → vi- → ci-).

4.3. S-STG for a computation block input multiplexor

Input multiplexors to computation blocks keep track of the lifetime (first and last use) of each register value. Figure 3.c depicts the S-STG for the multiplexor CP corresponding to one of the operands of block B (B_i) when using register Rk in sentence S. The sequence of transitions in the graph are the following:

- The CP waits for the selected input and for the selection signals (sel) to be valid (vi_{Rk}+ and sel←@Rk). Transition vi_{Rk}+ must occur only when the block operand uses Rk's value for the first time since its last definition (S ∈ first_use(Rk,B_i)).
- Selection is initiated and completed (r+ → c+).
- vo is activated to denote that the input to the computation block is valid.
- co+ indicates that the computation block does not require the input data any longer (it occurs after the output data of the computation block being stored into a register).
- The last part of the S-STG completes the handshake for signals <r, c>, and <vo, co>. In case this were the last use of Rk's value (S ∈ last_use(Rk,B_i)) the handshake would be also completed for signals <vi_{Rk}, ci_{Rk}> to indicate the register's CP that the value will not be used any longer by operand B_i.

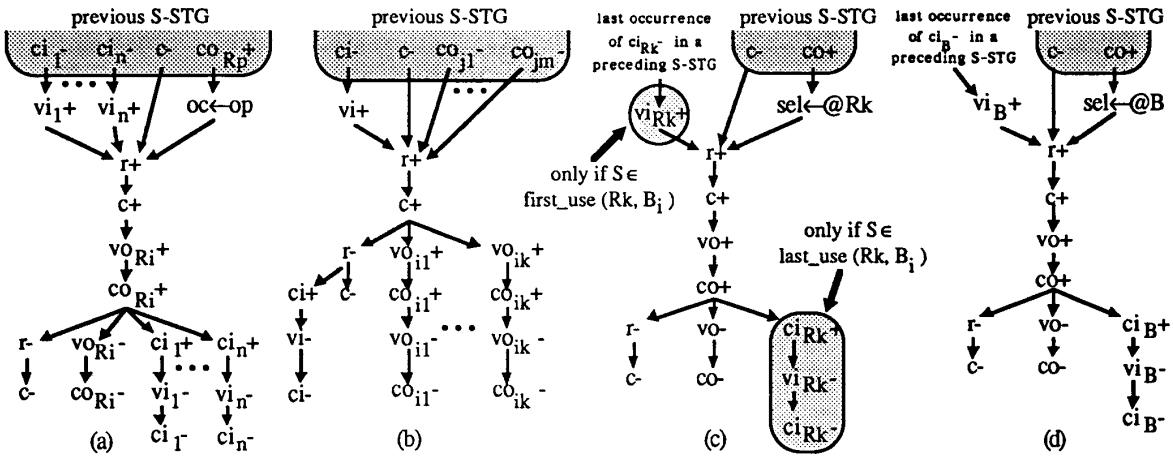

Figure 3. S-STGs for the execution of sentence S (*Ri = Rj op Rk*).
(a) computation block B (with n inputs) (b) Ri. (c) input mux to B_i (reading Rk). (d)input mux to Ri.

4.4. S-STG for a register input multiplexor

The sequence of transitions of the S-STG corresponding to a sentence that uses computation block B is depicted in Figure 3.d:

- The CP waits for the selected input (vi_B+) and for the selection signals (sel← @B) to be valid (the computation block's CP only activates vi_B when the produced data must be stored into the register).
- Selection is initiated and completed (r+ → c+).
- vo is activated to denote that the input to the register is valid.
- co+ indicates the latching completion.
- The last part of the S-STG completes the handshake for signals <r, c>, <vi_B, ci_B>, and <vo, co>.

5. USC and hazard-freeness

Every STG has an underlying state graph (SG) that can be deterministically derived [4]. Each state of the SG corresponds to one of the possible markings of the edges of the STG. In order to realize a circuit from a SG, a unique coding for all the states is required. In [4] the state is defined by using the signals of the STG as state variables.

If an SG has the Unique State Coding property (USC), then a hazard-free asynchronous circuit can be synthesized [12]. The existence of the Unique State Coding property is based on the following theorem [4][12]:

Theorem: An STG S has the USC property if and only if S is *live* and *no complementary set of transitions is feasible* in S.

The reader is referred to [9], where a study on the liveness and feasability of complementary sets of transitions is presented. The STGs generated by the proposed model are also proved to be realizable.

6. An example

Here we present an example of the synthesis of STGs from an SDFG. It corresponds to the generation of S-STGs for the execution of sentence S5 in the example of figure 1. Figure 4 depicts the S-STGs for the following blocks: *add*, R1, input multiplexor for add_2, and the input multiplexor for R1.

Some assumptions have been considered for the control signals of the data-path. The control signal for block *add* (oc) must have the value 0 for addition and 1 for subtraction. The input multiplexor for operand add2 receives two selection signals (sel_1 sel_0) which must have the values "00", "01", and "10" to select registers R2, R3, and R4 respectively. The input multiplexor for R1 receives one selection signal (sel) which must have the values "0" and "1" to select the output of blocks *ina* and *add* respectively.

For block *add* (figure 4.a), the previous S-STG corresponds to sentence S3. The operation code must change from "0" (addition in S3) to "1" (subtraction in S5). This can only happen after the output value for S3 has been stored into R3 (co_{R3}+).

In figure 4.b (R1), the previous sentence that defined R1 was S1. Before starting to latch the new value, the CP must wait for the previous uses of R1 to finish (add_1 and mul_1). The new value is only used by mul_1 (use (R1,S5)).

Figure 4.c shows the S-STG for the input multiplexor

to *add2*: Transition vi_{R4}+ should be preceded by the last occurrence of ci_{R4}-. Since this is the first use of R4 by *add2*, Reset+ must be used [9] (this also happens for all the transitions that have no preceding transitions). <sel_1,sel_0> must change from "00" (selecting R2 in S3) to "10" (selecting R4). Thus, only the transition sel_1+ must be generated (similarly for sel+ in figure 4.d). Since this is the first and last use of R4's value by *add2*, the full handshake for <vi_{R4}, ci_{R4}> must be generated.

7. Conclusions

An asynchronous architectural model for behavioral synthesis has been presented. The architecture is seen as a set of communicating processes, each one consisting of a data-processing part and a control process. Self-timed modules are use for data-processing, while control is distributed all over the control processes.

A synthesis procedure for the control has been proposed, based on the generation of an STG for each process. Hazard-free circuits are obtainable by using the existing approaches to synthesize STGs.

Open research areas are still left open to make faster and smaller circuits based on the proposed architecture model. Here we direct the attention to some of them: increasing parallelism in STGs, using two-phase handshake, deadlock detection, partitioning data-flow graphs into loosely-coupled data-processing sections, considering control dependencies, etc. Behavioral synthesis for asynchronous circuits is still an immature area that requires much more research work. New approaches for operation scheduling, module binding, control synthesis, etc, must be conceived. This paper is an effort in that direction.

References

[1] R. Camposano and W. Wolf. *High-Level VLSI Synthesis*. Kluwer Academic Publishers, 1991.

[2] R.E. Miller. *Switching Theory*, vol. 2. Wiley and Sons, 1965.

[3] S.H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley Interscience, 1969.

[4] T.A. Chu, *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, MIT, June 1987.

[5] T. Meng, R.W. Brodersen, and D.G. Messerschmitt. Automatic Synthesis of Asynchronous Circuits from High-Level Specifications. *IEEE Transactions on CAD of ICs and Systems*, Vol. 8 (11), Nov. 1989, pp. 1185-1205.

[6] A.J. Martin. Compiling Communicating Processes into Delay-insensitive VLSI Circuits. *Distributed Computing*, Vol. 1 (4), Springer-Verlag, pp. 226-234, 1986.

[7] Kees van Berkel et al. The VLSI-programming language Tangram and its translation into handshake circuits. *Proc. of the European Conference on Design Automation*, pp. 384-389, Feb. 1991.

[8] M. Hirayama. A Silicon Compiler System Based on Asynchronous Architecture. *IEEE Transactions on CAD of ICs and Systems*, Vol. 6 (3), May 1987, pp. 297-304.

[9] J. Cortadella and R.M. Badia. *High-Level Synthesis of Asynchronous Systems*. Research Report UPC/DAC RR-91/19, Oct. 1991.

[10] C.L. Seitz. *Introduction to VLSI Systems*. Chapter 7, Mead and Conway (Eds.), Addison Wesley, 1981.

[11] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading MA, 1986.

[12] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. *Proc. of the 28th Design Automation Conference*, June 1991, pp. 302-308.

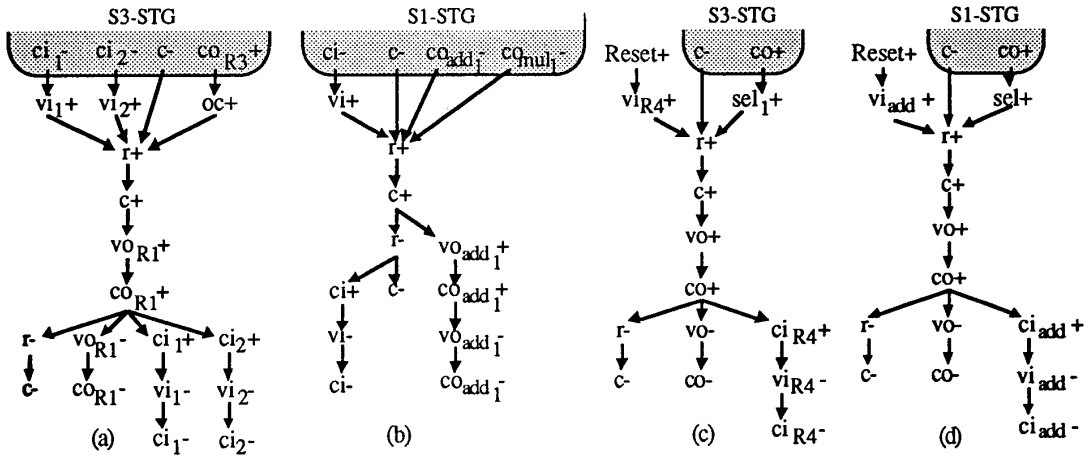

Figure 4. S5-STGs (R1 = R3 - R4)
(a) Block *add*. (b) register R1. (c) Input mux to *add2* (reading R4). (d) input mux to R1