A TRANSFORMATIONAL APPROACH TO ASYNCHRONOUS HIGH-LEVEL SYNTHESIS

Ganesh Gopalakrishnan [1]
ganesh@cs.utah.edu,
University of Utah,
Dept. of Computer Science,
Salt Lake City, UT 84112, USA

Venkatesh Akella [2]
akella@eecs.ucdavis.edu,
University of California,
Dept. of EE and Computer Engg.,
Davis, CA 95616, USA

UUCS-93-017

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

July 21, 1993

Abstract

Asynchronous high-level synthesis is aimed at transforming high level descriptions of algorithms into efficient asynchronous circuit implementations. This approach is attractive from the point of view of the flexibility it affords in performing high level program transformations on users' initial descriptions, the faithfulness with which it supports the communicating process model of computation, and the ease with which it accommodates computations that have data dependent control-flow decisions as well as data dependent execution times. In this paper, we take the reader through the entire process of synthesizing two asynchronous circuits using our high level synthesis tool, SHILPA, starting from input descriptions in hopCP, emphasizing the *program transformation* techniques employed in the process. Specifically, we show how tail-recursive loops with accumulating parameters can be software pipelined, by evaluating the accumulating parameters in separate processes. We then show how the resulting hopCP flow graphs (HFGs) are transformed through *action refinement* resulting in *normal form HFGs* (NHFGs). NHFGs are then technology mapped onto an Actel FPGA implementation. Our results are illustrated on a pipelined factorial circuit and a pipelined integer square-root circuit.

Keywords: Asynchronous/Self-timed Systems, High Level Synthesis, Program Transformations

[1]Supported in part by NSF Awards MIP 8902558 and 9215878

[2]The work reported here was part of this author's PhD dissertation work, and was supported by a University of Utah Graduate Fellowship

1 Introduction

High level synthesis tools hold considerable promise towards facilitating the rapid creation of error-free Very Large Scale Integrated (VLSI) designs [1]. Most of today's high level synthesis tools are designed to generate *synchronous* circuits from high level descriptions. For many practical reasons (explained in Section 2), it is becoming increasingly difficult to manually build or automatically synthesize large synchronous circuits. In this paper, we show how these problems can be largely avoided through *asynchronous high level synthesis*: the creation of efficient asynchronous circuits starting from high level descriptions.

Having built such an asynchronous high level synthesis tool, SHILPA (detailed in Section 3), one of the major challenges we are *now* faced with is in obtaining *hardware descriptions* that can be compiled into efficient asynchronous circuits. Users' initial hardware descriptions are written with emphasis on clarity, and as such do not result in efficient asynchronous circuits when compiled. Although the compiled circuits can be subject to circuit level optimizations [2], it is virtually impossible to employ circuit level optimizations to *compensate for* the lack of high level optimizations [3]. Therefore, it appears that it is essential to investigate high level optimizations for deriving efficient asynchronous circuits.

We propose the use of program transformations to transform initial hardware descriptions (written with emphasis on clarity) into ones that result in efficient asynchronous circuits. We present the following specific results. We show how tail-recursive functional programs with accumulating parameters [3] can be transformed into *software pipelined*[4] concurrent communicating process descriptions (Section 4). To concretely illustrate this idea, we provide an overview of how SHILPA compiles concurrent process descriptions in the input language hopCP into asynchronous circuits, and proceed to show the derivation of the circuit realizing the pipelined `factorial` function (Section 5.1). We then show how the hopCP process descriptions (internally stored as hopCP flow graphs (HFGs), an annotated form of Petri nets) can be further refined through a procedure *action refinement* into asynchronous circuit descriptions (Section 5) and technology mapped into a circuit description. In the Appendix, we also show how program transformations can be valuable in transforming imperative algorithms. This method is illustrated on the specification of the *integer square-root* computation. These results, as well as similar results obtained by others (*e.g.*, [5, 6, 7]) underscore the importance of high level program transformations in the derivation of efficient asynchronous circuits. Section 6, draws conclusions based on our results so far.

[3]Doing so would be analogous to trying to perform machine code optimizations that give the same effect as (e.g.) loop invariant optimizations, in a software compiler!

[4]Software pipelining [4] is a term which describes the fact that the next iteration of a loop can be begun before the current iteration finishes.

2 Motivations, and Related Work

From the point of view of circuit design in the large, asynchronous circuits are attractive in several ways (see Brzozowski and Seger [8], and Gopalakrishnan and Jain [9] which are two surveys, Sutherland [10] and Seitz [11] which are two widely cited articles, and the papers in [12] which are a collection of recent papers). Asynchronous circuits avoid clock-distribution problems that cost valuable design time in large synchronous systems [13]. Asynchronous circuits are easier to incrementally expand, as their operation is not based on global clock schedules. High level synthesis of computations with data depend timings using the synchronous paradigm is difficult [14]. Asynchronous circuits avoid this problem due to their handshake based "self scheduling" nature, and can even exploit the data dependent nature of operator timings [15] to gain average-case speed-up.

From the point of view of high level design derivation, asynchronous circuits pose several interesting challenges. Given the high level description of a problem, obtaining a circuit that is optimized for area and time in a pre-specified way is one such challenge. As in traditional compilers, the optimizations can be carried out at several levels, some of which are: source-to-source level; flow-graph level; and final code level. Circuit level optimizations of asynchronous circuits have been studied in [2, 16]. Flow-graph level optimizations have been studied in [17]. Source to source optimizations have been studied by Nielsen and Martin in [5], by van Berkel in [6], and Ebergen in [7]. The works by Nielsen, Martin, and Ebergen considers the derivation of regular structures. Our work is more along the lines of that by van Berkel [6] in that we deal primarily with less regular computations (and final circuits). Our work is also somewhat related to that of Johnson [18], except that our target is asynchronous circuits (while Johnson's target is synchronous circuits). Also, neither van Berkel nor Johnson explore the derivation of software pipelined designs through program transformations. We consider our approach to be attractive from the point of view of formal verifiability, and because it ties in quite well into the SHILPA system that has already been built and tested on a number of designs. The fact that SHILPA can currently synthesize Actel FPGA based asynchronous circuits from high level descriptions also affords us a flexible environment for experimentation with these ideas.

3 Overview of SHILPA

Recently there has been a growing interest in the automated high level synthesis of asynchronous circuits from concurrent process descriptions. Our work [19, 20] falls into this category. Improvements in SHILPA over other efforts in this area (*e.g.*, [16, 5, 21, 2]) are primarily the following: (i) hopCP, the source language for SHILPA, is a mixed process and functional language tailored for hardware description with distributed shared variables, barrier synchronization, and broadcast communication. It is more expressive than Martin's input language 'CHP', Brunvand's version of 'Occam', or van Berkel's language 'Tangram'. In this paper, we show how certain hopCP descriptions written in the functional notation can be transformed into the process notation; (ii) our graph-based compilation scheme is amenable to flow-analysis based optimizations; (iii) SHILPA is an integrated collection of tools in which the user can direct the outcome of the synthesis process through interactive commands.

In this paper, we focus on certain high level optimizations to transform hopCP descriptions into a form that engenders efficient asynchronous circuits. We first take a familiar example: a circuit to compute the factorial of an integer. We will initially transform the tail-recursive definition of factorial with an accumulating parameter into two concurrent processes, where the first process is the driver and the second process evaluates the accumulating parameter. In the Appendix (Section A), we will present similar transformations done on the imperative specification of the integer square-root function. In our presentations, we will employ a pseudo-hopCP notation, which will be explained along with the examples.

4 Transforming Functional Programs for Software-pipelining

Hardware description and synthesis using a purely functional notation has attracted much interest lately [18, 22]. Functional languages are attractive for system-level description due to their referential transparency, their ability to state the desired behavior without any operational committments, their use of higher order functions, and the sophisticated type system they come with [23]. In this presentation, we stick to a first-order tail-recursive notation similar to what [18] employs. The use of accumulating parameters is a popular way of converting non tail-recursive descriptions to the tail-recursive form. Other researchers who use the functional notation for hardware description have not (to the best of our knowledge) studied the problem of deriving software-pipelined designs from functional descriptions. Although Sheeran [22] has used the functional notation for hardware description, it was used for deriving regular designs. Ebergen's work [7] is in deriving regular asynchronous designs from functional descriptions. Busvine [24] has studied the problem of translating SML programs to *sequential* Occam2 code (he does not address deriving parallel/pipelined Occam2 programs).

We now illustrate our ideas on the familiar tail-recursive `factorial` description:

```
fact[n,a] <=   (n=0) -> result!a -> again?n -> fact[n,1]
             | (not (n=0)) -> fact[n-1, n*a]
```

Here, `->` denotes sequencing and `|` denotes guarded choice. Process `fact` has a list of formal parameters `[n,a]` which are initialized to suitable values to begin with. The construct `result!a` is an output communication command (as in CSP), and synchronizes with an input communication command of the form `result?variable` within another process. The construct `again?n` is an input communication command (as in CSP) which *rendezvous* with an output communication command of the form `again!expression` from another process.

From the above definition, one can note that the value of `n*a` is bound to 'a' upon each tail-call. This fact can be made clear by slightly modifying the definition of `fact`; we also introduce a concurrent process `pa` in the process:

```
fact[n,a] <=   (n=0) -> result!a -> again?n -> fact[n,1]
             | (not (n=0)) -> mult!(n,a) -> rslt?w -> fact[n-1, w]
||
pa[] <= mult?(x,y) -> rslt!(x*y) -> pa[]
```

Process `pa` is sent a pair `(n,a)` whenever `n` and `a` are to be multiplied. After multiplication, process `pa` sends back the result through port `rslt`.

Notice that the value of '`a`' is not used immediately in the body of `fact`. Thus, the next iteration of `fact` can be allowed to begin even before the evaluation of `n*a` finishes. This change is reflected by letting process `pa` "own" the formal parameter '`a`', and allowing process `pa` to multiply the two numbers in the background. We also eliminate '`a`' from the "next iteration" of `fact` (captured by process `fact'`). Notice that `fact'` does not wait for the result of the multiplication to come back from `pa` before it tail-recurses once again.

```
fact[n,a] <=   (n=0) -> result!a -> again?n -> fact[n,1]
             | (not (n=0)) -> mult!(n,a) fact'[n-1]
||
fact'[n] <=    (n=0) -> senda! -> again?n -> fact'[n]
             | (not (n=0)) -> rslt?w -> mult!(n,w) -> fact'[n-1]
||
pa[a] <= mult?(x,y) -> rslt!(x*y) -> pa[1]
```

Since `fact'` is devoid of its second argument, it appeals to process `pa` through command `senda!` whenever it needs to send the final answer. Thus, `pa` will end up having two commands, indicated by a guarded choice in its definition. When `n` is not equal to zero, `fact'` waits through the result of the previous multiplication (through `rslt?w`), starts the next multiplication, and proceeds.

Notice that `fact` and `fact'` are very similar, and it is redundant to keep both. We therefore devoid `fact` also of parameter `a`. Also, consider the steps `rslt?w -> mult!(n,w)`: `fact'` needn't obtain the results of the multiplication, (`w`) from `m`, only to send it back to process `pa` (as part of the tuple `(n,w)`). Thus, process `fact'` simply ends up sending '`n`' to process `pa`, and asks it to multiply with the value of `a` that process `pa` is already holding. Doing this (and renaming the combined factorial process `factpipe`) results in the following description:

```
factpipe[n] <=    (n=0) -> senda! -> again?n -> factpipe[n]
                | (not (n=0)) -> mult!n -> factpipe[n-1]
||
pa[a] <=   mult?n -> pa[n*a]
         | senda? -> result!a -> pa[1]
```

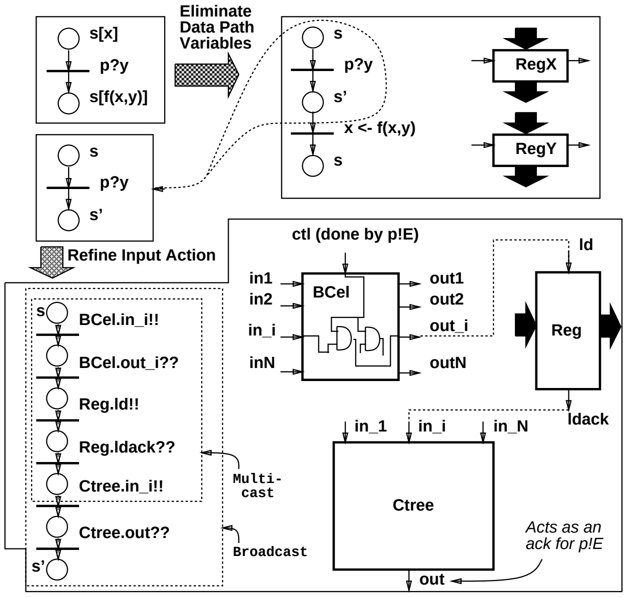

Figure 1: An Example of Action Refinement

Processes `factpipe` and `pa` when started in parallel with `n` holding the desired number, and with `a` initialized to `1` calculates factorial of `n`. Thereafter, `factpipe` seeks the next number to work on, through the input communication command `again?n`.

One of the advantages of using a mixed process/functional notation for the above derivation is quite apparent: *operational details* of program evaluation can be elegantly captured using the process description sub-language. The derivation evolves, gradually substituting the process component for the functional component. hopCP does have a formal semantic description [19]; however, we do not yet have an *algebra* that can support the above kind of derivations. (We would like to investigate such an algebra.) The work of Page and Luk [25] who have studied process-level transformations in Occam is quite relevant to cite in this connection.

5 From hopCP Descriptions to Asynchronous Circuits

In SHILPA, we synthesize *transition style* circuits with *data bundling* [10]. Each hopCP description is internally represented through a flow graph (called the hopCP flow graph, or HFG). An HFG is similar to a Petri-net in that it has both *places* and *transitions*; however, these are annotated with data path states and/or communication actions. In Figure 1, a transition ($S[x]$, $p?y$, $S[f(x,y)]$) from an HFG is shown. This transition starts at state $S[x]$ which evolves through input rendezvous action $p?y$ to state $S[f(x,y)]$. According to our conventions, process S is making a *tail call* back to itself, and updating its internal variable x with $f(x,y)$ in the process. The first step in refining this action is to allocate register x to hold the data state and register y to hold the input value received, as shown. The data state update is made explicit by introducing a *register transfer* action $x \leftarrow f(x,y)$.

Next, we take the transition (S, $p?y$, S') and refine it by invoking a pre-defined expansion for the input communication action. Since input rendezvous actions follow the multiway

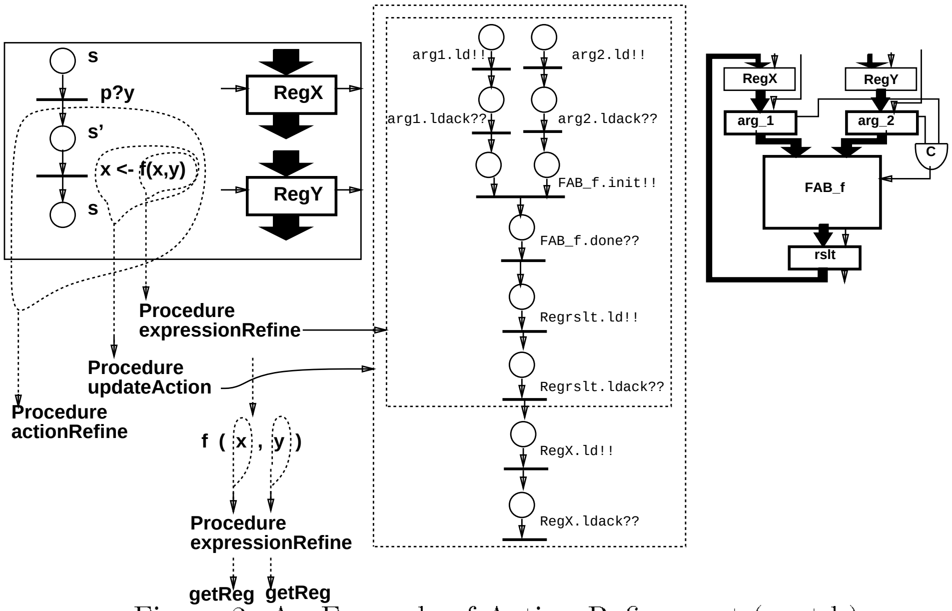

Figure 2: An Example of Action Refinement (contd.)

rendezvous semantics, in SHILPA we generate an interconnection of C-ELEMENTs called the "broadcast C-ELEMENT", or BCel, which has N+1 inputs and N outputs, where N is the number of receivers reading from the channel [5]. BCel has the property that as soon as a transition has been produced on input in_i and the sender has produced a transition on input ctl, a transition is produced on output out_i. This loads the data into a register (Reg) that is also allocated. The acknowledge from the register goes to a completion tree. The output of the completion tree is an acknowledgement for the sender that the value has been latched by all the receivers. The acknowledgement for each receiver is taken to be the register load acknowledge signal, resulting in the multicast semantics. This is indicated in Figure 1 by the dotted box labeled "Multicast", meaning that the last Petri-net transition enclosed by this dotted box, Ctree.in_i!!, can be taken as the acknowledgement signal by the ith receiver. If broadcast semantics is desired, the completion signal for the receivers should also be the output of the completion tree, as shown by the correspondingly named dotted box in the same figure.

Continuing with this example, we next take the register-transfer action $x \leftarrow f(x, y)$ for refinement, by invoking procedure actionRefine. This recursively calls procedure expressionRefine to refine expression $f(x, y)$. In compiling the application of a function f, its arguments are recursively refined, to begin with. In this case, the arguments are both variables, whose refinement results in calls to getReg, that retrieves the registers already allocated corresponding to these variables. Thereafter, a *function action block* (FAB) is allocated corresponding to f. The NHFG shown in Figure 2 is then generated. As can be seen, this NHFG captures control sequencing that first loads the argument registers of FAB_f, initiates the function evaluation, loads the result register, and then loads RegX to complete the required evaluation.

[5] A Petri net transition annotated with "M.p!!" reads " apply a signal transition on module M's port p; likewise, "M.q??" denotes awaiting a transition.

5.1 Obtaining a Pipelined Factorial Circuit

We synthesize those versions of processes `factpipe` and `pa` given last in Section 4. We show here only `factpipe`. First, the description is subject to action refinement through command:

```
{25} bliss.cs> SHILPA
val it = true : bool
- val (g,t,r,n,f,c,typ,fvd) = ar "example/factpipe.h";
Detecting Sharing .........
.......Found 1 shared actions
Inserting CALL and BCALL Modules appropriately .........
Modifying NHFG to reflect Sharing .........
Generating MERGE elements ....
```

The above command results in the initial resource list (the purpose of each resource is also explained):

```
- printResource r;
REG_9:argument for AB_8_arg_n   C_10:data query for again?n
REG_6:argument for AB_5_arg_n   REG_3:query var for n
PAB_8:1 for zero                REG_7:result for 5
AMUX_12:2 for n                 FAB_5:1 for (decr n)
XOR_13:3 for control!!
```

Next, we eliminate argument- and result registers that are not needed. The idea is: "eliminate short of creating combinational loops". (We could have retained the argument and result registers, had we been interested in micropipelining the design.) SHILPA automatically reconfigures the circuit to compensate for the lack of these registers:

```
val it = () : unit
- val (t1,r1,n1) = eliminate_argument_register (t,r,n)[8,5];
Generating MERGE elements ....
```

Finally, we invoke our *technology mapper*, to create an Actel FPGA wire-list file:

```
- hopCP2actel r1 n1 f typ fvd;
Module Name = factpipe
```

The resulting circuit for `factpipe` is in Figure 3 The circuit works as follows. Initially, transition `start` is applied to the `XOR`. This triggers module `ZERO` to test whether `n=0`. The "true" transition, `T`, triggers `SENDA_OUT`, which implements the `senda!` communication. When the acknowledgement `SENDA_IN` comes, it triggers the C element which fires when `AGAIN_IN` also arrives (the communication `again?n`), and when it does, loads the new `n` through the asynchronous multiplexor `AMUX` into register `n`. The "false" transition, `F`, triggers `MULT_OUT`, which implements `mult!n`. When `MULT_IN` arrives (the acknowledge for `mult!n`), the decrementer module `decr` is triggered. Its `ack` loads the result register of `n`, and is routed to register `n` through the asynchronous multiplexor `AMUX` upon tail-call.

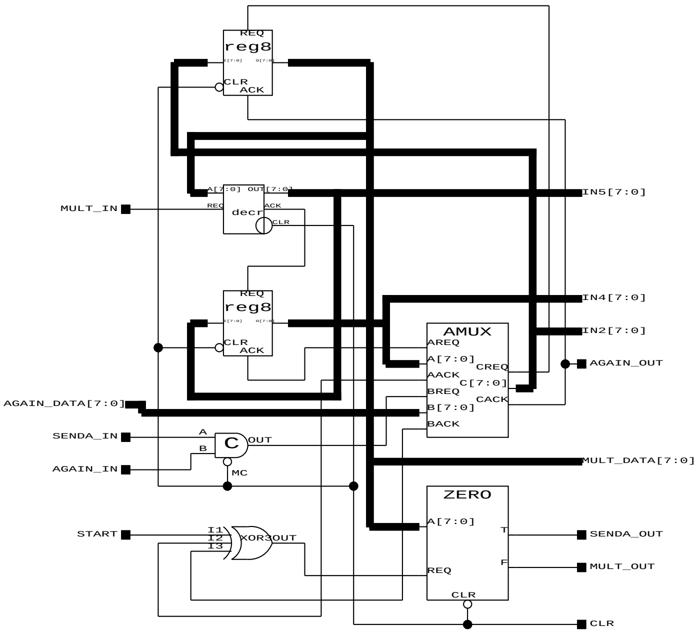

Figure 3: The Pipelined Factorial Circuit: process `factpipe`

6 Summary and Conclusions

We have found that a transformational approach to asynchronous circuit synthesis is promising in a number of ways. In this paper, we show how to derive software pipelined asynchronous circuit implementations of tail-recursive programs through program transformations. We then present the approach of transforming process descriptions from HFGs to NHFGs through *action refinement*, and point out its advantages: (i) it allows graph based algorithms to be used for optimizations; (ii) it is modular, refining each HFG fragment with its associated NHFG elaboration and the associated resources; (iii) users can modify the NHFG through interactive commands, and hence can have direct control over the final circuit that emerges; (iv) it also allows the application of graph-based performation evaluation techniques (see below).

A related question we are answering at this stage is the following: "when is it worthwhile to perform a high-level optimization"? This question has not been answered satisfactorily by the high-level synthesis community, for a collection of communicating processes. Ku [14] has done pioneering research in estimating the performance of concurrent computations. We are gravitating more towards the work pioneered by Zuberek [26], as well as Burns [27], as our HFG based internal representation fits well with the Petri net based representation used by Zuberek and Burns.

Through simulation studies, we have observed that software pipelining can be good (as we observed for a pipelined *minmax* circuit) or that the overheads can sometimes overshadow the benefits (as we observed when we pipelined a *serial-parallel* multiplication algorithm) [28]. As part of our future work, we plan to explore performance evaluation techniques in greater detail.

References

1. Michael C. McFarland, Alice C. Parker, and Raul Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.

2. Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits.* PhD thesis, Carnegie Mellon University, 1991.

3. Peter Henderson. *Functional Programming.* Prentice Hall, 1980.

4. John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, 1990.

5. Christian Nielsen and Alain Martin. The derivation of a multiply/accumulate unit. In *Proceedings of the 26th Hawaiian International Conference on System Sciences*, January 1993.

6. Kees van Berkel. An error decoder for the compact disc player as an example of vlsi programming. In *Proceedings of the 1992 EURODAC*, pages 69–74, March 1992.

7. Jo C. Ebergen. The derivation of a serial-parallel multiplier and divisor. Technical report, Department of Computer Science, University of Waterloo, 1990.

8. John Brzozowski and Carl-Johan Seger. Advances in Asynchronous Circuit Theory: Part I: Gate and Unbounded Intertial Delay Models; and Part II: Bounded Intertial Delay Models, MOS Circuits, Design Techniques. Technical report, University of Waterloo, 1990.

9. Ganesh Gopalakrishnan and Prabhat Jain. Some recent asynchronous system design methodologies. Technical Report UUCS-TR-90-016, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1990.

10. Ivan Sutherland. Micropipelines. *Communications of the ACM*, June 1989. *The 1988 ACM Turing Award Lecture.*

11. C. A. Mead and L. Conway. *An Introduction to VLSI Systems.* Addison Wesley, 1980. *Chapter 7, entitled "System Timing".*

12. Ganesh Gopalakrishnan and Erik Brunvand. Mini-track coordinators' introduction to papers on asynchronous circuits and systems, January 1993.

13. Information on the DEC Alpha Processor, FTP-able from DEC; details are obtainable from `hudson@tolkin.enet.dec.com`. Also, the story "How DEC developed Alpha", IEEE Spectrum, July, 1992, pages 26–31.

14. David Ku. *Constrained Synthesis and Optimization of Digital Integrated Circuits from Behavioral Specifications.* PhD thesis, Department of Computer Science, Stanford University, June 1991.

15. Ted E.Williams and Mark Horowitz. A zero-overhead self-timed 160ns 54bit cmos divider. *IEEE Journal of Solid State Circuits*, 26(11):1651–1661, November 1991.

16. Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In editor C.A.R. Hoare, editor, UT Year of Programming Institute on Concurrent Programming. Addison-Wesley, 1989.

17. Ganesh Gopalakrishnan and Venkatesh Akella. High level optimizations in compiling process descriptions to asynchronous circuits. *VLSI and Signal Processing*, May 1993. *To Appear in a Special Issue on Asynchronous Design.*

18. Stephen Johnson, B. Bose, and C. Boyer. A tactical framework for hardware design. In Graham Birtwistle and P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 349–383. Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.

19. Venkatesh Akella. Action refinement based transformation of concurrent processes into asynchronous hardware. Ph.D. research in progress.

20. Venkatesh Akella and Ganesh Gopalakrishnan. Shilpa: A high-level synthesis system for self-timed circuits. In *International Conference on Computer Aided Design (ICCAD), Santa Clara*, November 1992. To appear.

21. Kees van Berkel. *Handshake circuits: an intermediary between communicating processes and VLSI.* PhD thesis, 1992.

22. Mary Sheeran. mufp, a language for vlsi design. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 104–112, 1984.

23. John Hughes. Why functional programming matters. Technical Report 16, Programming Methodology Group, University of Goteborg and Chalmers Institute of Technology, Sweden, November 1984.

24. David Busvine. Translation of SML to Sequential Occam2. Technical report No. 91/7, Dept. of Computer Science, Heriott-Watt University, U.K., 1989.

25. Ian Page and Wayne Luk. Compiling Occam into Field-Programmable Gate Arrays. In *International Workshop on Field Programmable Logic and Applications*, September 1991. *September 4-6, 1991, Oxford University, UK.*

26. W.M. Zuberek. Timed petri nets and preliminary performance evaluation. In *7th Annual International Symposium on Computer Architecture*, pages 88–96, 1980.

27. Stephen Burns. Performance evaluation of asynchronous circuits. Technical Report TR-91-1, Computer Science Dept., California Institute of Technology, 1991.

28. Ganesh Gopalakrishnan and Venkatesh Akella. Specification, simulation, and synthesis of self-timed circuits. In *Proceedings of the 26th Hawaiian International Conference on System Sciences*, January 1993.

29. Nachum Dershowitz. The evolution of programs, Birkhauser, 1983.

A Transforming Imperative Programs

A large majority of algorithms are expressed in an imperative style. Therefore it is desirable to apply the techniques we have proposed thus far for for transforming imperative programs to perform optimizations such as software pipelining. We follow the lead of Dershowitz [29], who shows how imperative programs may be transformed to achieve operator strength reduction (replacing costly operations by cheaper equivalent operations), avoiding recomputing loop invariants, etc. We pick his *integer square-root* program [29, Page 176]. After program transformations, the resulting square root program has been devoid of costly operations such as *multiply*; it is given below in C:

```
main()
{ long a, u, v, w=2, t, z;
  printf("\n Give a\n"); scanf("%d",&a);
  while (2*a >= w) w=4*w;  u=-a; v=w/2;
  while (w>2) { w=w/4;     v=(v-w)/2;
              t=u+v;     if(t<=0){ u=t; v=v+w; }}
  z=(v-1)/2;  printf("z = %d\n", z);
```

Expressed in hopCP, this program reads:

```
(isqrt[] <= get_number?a -> getw[2,(mult2 a),a])
;
(getw[w,twicea,a] <=   ((twicea < w)  -> after_getw[w,(div2 w),0,(neg a)])
                     | ((not (twicea < w)) -> getw[(times4 w), twicea,a ]))
;
(after_getw[w,v,t,u] <=
         ((le2 w)  -> final_answer!(div2 (minus1 v)) -> isqrt[])
       | ((not (le2 w))  -> w := (div4 w)
                         -> v := (div2 (minus v w))
                         -> t := (plus u v)
                         ->    ((gt0 t) -> after_getw[w,v,t,u])
                             | ((not (gt0 t)) -> after_getw[w,(plus v w),t,t])))
```

After applying software pipelining transformations presented in this paper on process `after_getw` (which has an accumulating parameter), we have the following two equivalent (but pipelined) processes:

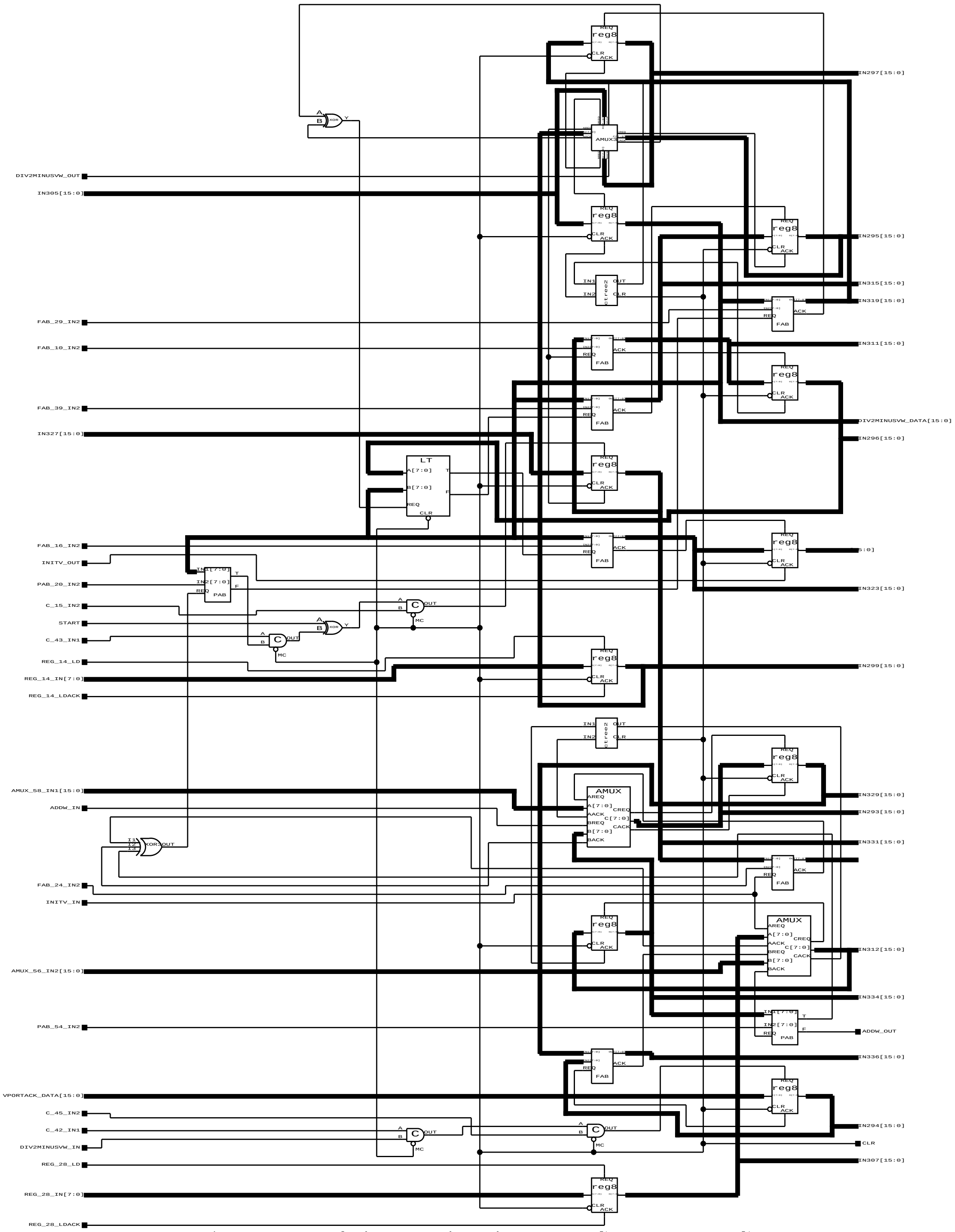

Figure 4: A Portion of the Pipelined Integer Square-root Circuit

```
(after_getw[w,t,u] <=
          ((le2 w)  -> send_final_answer!  -> psqrt[])
        | ((not (le2 w))  -> w := (div4 w) -> div2minusvw!w -> vport!
                          -> vportack?v1   -> t := (plus u v1)
                          ->    ((gt0 t)   -> after_getw[w,t,u])
                              | ((not (gt0 t))
                                  -> addw!w -> after_getw[w,t,t]) ))
||
(pv[v] <= (div2minusvw?w1 -> pv[(div2 (minus v w1))])
         |(vport? -> vportack!v -> pv[v])
         |(addw?w1 -> pv[(plus v w1)])
         |(send_final_answer? -> final_answer!(div2 (minus1 v)) -> pv[v])
         |(initv?v -> pv[v]))
```

The resulting circuit for `psqrt` (the pipelined counterpart of isqrt), `getw`, and the pipelined version of `after_getw` is shown in Figure 4.