Hierarchical Action Refinement: A Methodology for Compiling Asynchronous Circuits from a Concurrent HDL

Venkatesh Akella and Ganesh Gopalakrishnan

UUCS-91-012

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

August 5, 1991

Abstract

A hardware specification formalism called hopCP is introduced, hopCP provides an uniform notation to describe the causal relationships between a set of nonatomic actions which capture the computational, concurrency, control and communication aspects of hardware behavior. A systematic approach to synthesize asynchronous circuits from hopCP, based on hierarchical action refinement, is presented. Actions in hopCP could denote control, value communication, and computation (e.g.: evaluation of functional expressions). The salient features of our scheme include systematic resource allocation and control decomposition with the ability to use standard off the shelf components and the flexibility to retarget the compilation to different data transfer protocols and hardware primitives. The synthesis of LRU (least recently used) circuit used in the design of cache memory and virtual memory systems is presented to illustrate our ideas.

Hierarchical Action Refinement: A Methodology for Compiling Asynchronous Circuits from a Concurrent HDL

Venkatesh Akella and Ganesh Gopalakrishnan
Department of Computer Science
University of Utah
Salt Lake City, UT 84112

Abstract

A hardware specification formalism called hopCP is introduced. hopCP provides an *uniform* notation to describe the *causal relationships* between a set of *nonatomic actions* which capture the computational, concurrency, control and communication aspects of hardware behavior. A systematic approach to synthesize *asynchronous* circuits from hopCP, based on hierarchical action refinement, is presented. Actions in hopCP could denote control, value communication, and computation (e.g.: evaluation of functional expressions). The salient features of our scheme include systematic resource allocation and control decomposition with the ability to use standard *off the shelf* components and the flexibility to *retarget* the compilation to different data transfer protocols and hardware primitives. The synthesis of a LRU (least recently used) circuit used in the design of cache memory and virtual memory systems is presented to illustrate our ideas.

1 Introduction

In this paper we shall describe a systematic methodology to implement asynchronous circuits starting from high level specifications written in hopCP. hopCP is a process+functional language which evolved from a lockstep process oriented model for hardware called HOP [7] and is designed to support the specification, verification, and synthesis of mixed synchronous/asynchronous systems. hopCP takes a *sequence domain* view of hardware where the behavior of a system is captured by the *causal relationship* between a set of *actions* which the hardware can perform. The actions in hopCP could denote control, value communication and computation (e.g.: evaluation of functional expressions). The latter is possible because the specification formalism does not prescribe any *timing discipline* such as duration of an action, clocking/handshaking protocols, etc.

hopCP incorporates the notions of *nonatomicity* of actions which was advocated in the past by Lamport[9], Pratt[14], and Milne [13] among others. A nonatomic action can be imparted structure by *action refinement*. This gives the ability to model a hardware

system at different levels of refinement. Let us explain this with an example. At the architecture level description of a system, it is perfectly reasonable and is in fact sufficient to consider the *add* instruction as just a single action. But, reasoning at the gate level soon reveals that an execution of the *add* instruction actually involves several microinstructions like instruction fetch, decode, operand fetch, that are spread across an *interval of time* (hence nonatomic). Going further down to the circuit level we see that each of the microinstructions themselves need certain signals (voltages) going high or low. The advantages of hierarchical action refinement include the ability to defer the choice of a timing discipline in high level synthesis and the flexibility to treat synchronous and asynchronous hardware design in the same framework. It also facilitates the formal verification of detailed designs against abstract specifications by using the notion of *reduction* (action abstraction in our framework) as explored by Kwong[8].

In this paper we show that with just three refinement rules, specified elegantly by an *action grammar*, we can refine the actions in hopCP to a wide range of asynchronous circuit implementation schemes which include transition signaling, level-based signaling, dual-rail data and bundled data. The overall methodology of compiling hopCP specifications into asynchronous hardware resembles conventional programming language compilation *with the code generation phase being replaced by action refinement.* This gives us the ability to incorporate several optimization schemes based on data flow analysis which are available from decades of conventional compiler research. We will briefly hint upon some of them in this paper.

Related Work

At present, two of the most prominent efforts in the compilation of *large* asynchronous circuits from high level specifications are by Martin [10] and Brunvand [4]. Our approach, though greatly influenced by the above works, differs from them in the following respects: (i) hopCP being a process+functional language, enables us to specify computational aspects of hardware elegantly and facilitates asynchronous datapath synthesis through the analysis of functional programs. Since functional programs specify parallelism naturally and are referentially transparent, it is easier to obtain concurrent and pipelined implementations from them (ii) our approach incorporates systematic resource allocation and control decomposition followed by global optimization as in conventional (synchronous) high-level synthesis efforts as described in [11] (iii) notion of *nonatomic* actions and *action refinement* permit hierarchical reasoning of asynchronous circuits and facilitates Lamport-style [9] verification of properties of our circuits.

Overview of the paper

In the next section we will briefly introduce the hopCP notation and techniques to model communication, computation and concurrency aspects of hardware in it. We will illustrate the formalism by describing the specification of the LRU module in detail. In section 3 we will describe the architecture of the hardware compiler focusing on action refinement rules, high level optimizations and the primitive modules to which we compile our specifications. In section 4 we will illustrate the action refinement strategy by synthesizing a LRU circuit

and finally we will summarize our results and describe future directions of our work.

2 Definition and Informal Semantics of hopCP

Behavioral Description in hopCP

The behavior of a hardware system is captured by a *state transition system* called HFG (hopCP Flow Graph[1]) which is defined as follows:

Definition 2.1 *A HFG is a 4-tuple,* $\langle S_i, S, Act, \rightarrow \rangle$ *where* S *is the set of* states, $S_i \subseteq S$ *is the set of* initial *states,* Act *is the set of* actions *and* $\rightarrow \subseteq (\mathcal{P}^+(S) \times Act \times \mathcal{P}^+(S))$, *where* $\mathcal{P}^+(S)$ *is the* non empty *power set of* S.

S is typically a pair $\langle cs, ds \rangle$ with cs denoting the *control state* and ds representing the *data state.* Control state is analogous to the program counter and data state to the state of registers and memory. Act is the set of *actions* which capture the *computational* and *communication* aspects of the hardware in hopCP. There are three types of actions which are defined using the following syntactic domains:

$$\begin{array}{lcllcl} C & = & Set\ of\ Control\ Actions & D & = & Set\ of\ Data\ Actions \\ E & = & Set\ of\ Expression\ Actions & A & = & C + D + E \\ Act & = & \mathcal{P}(A),\ Set\ of\ Compound\ Actions & & & \end{array}$$

where a *compound action* is defined as a set of actions which are potentially nonatomic and compound, and which are performed *concurrently* i.e. in an undetermined order. The symbol + denotes *discriminated union.* Let us briefly explain the individual constituents of the domain A of actions.

Control Actions capture the *synchronization* aspects of hardware behavior in our formalism. These are *Boolean* signals and are also referred to as *events* in our formalism. Events are directional i.e. they could be *input* (passive) or *output* (active). For example, $p!$ denotes an output event on *port* p while $q?$ denotes an input event on *port* q.

Data Actions capture *synchronization and value communication* aspects of hardware behavior. Data actions are further classified into *data queries* which denote *input* or passive actions and *data assertions* which refer to *output* or active actions. For example, $p!exp$ denotes asserting the value of an expression exp on *port* p while $q?x$ denotes querying a value x from *port* q.

Expression Actions capture the *computation* aspects of a hardware module and are described in a simple expression language whose abstract syntax can be described as follows. Let $e \in$ Expressions E, $x \in$ Variables VAR, and $v \in$ Constants C (which includes primitive values like $true$, $false$ and predefined function symbols like $+, -, *, shift$ etc.)

$$e ::= v \mid x \mid \lambda x.e \mid e\ e \mid \textbf{if } e \textbf{ then } e \textbf{ else } e \mid fix\ \ e\ .$$

[1]More precisely, it is a *hypergraph*, if the states are viewed as vertices and the actions as edges.

(e e) denotes function application, and *fix* is the fixpoint-finding combinator. In the current version of hopCP we do not support general recursion. Tail recursion in the form of iteration is allowed however. Also, though λ *abstraction* has been introduced as a basic construct, we do not intend expression actions to be *higher-order* (at least not in the current version of hopCP). We see that *computation* is captured just as an *action* in the *HFG* which is possible because of the nonatomic view of an action. Implementing an expression action will involve its decomposition into subactions corresponding to the evaluation of the constituent expression.

Operations on Flowgraphs

hopCP provides three combinators (or functions) *progress, choice and parcomp* to construct HFGs denoting more complex behaviors from HFGs describing simpler ones. The first two are similar to the *prefix* and *choice* in CCS while *parcomp* is like the *composition* operator in CSP except that its semantics has been augmented to take care of *nonatomic* actions. We shall briefly present the informal semantics of the three combinators (the detailed semantics are described in [1]). Let P, Q and R denote HFGs and $a, b \in Act$.

progress $\quad P \Leftarrow a \rightsquigarrow Q$

The *progress* combinator $\rightsquigarrow$, defines a new *HFG* P which can perform an action a and behave as described by *HFG* Q. It is similar to *sequencing* operator found in programming languages. If a is empty then behaviors denoted by P and Q are identical.

choice $\quad P \Leftarrow a \rightsquigarrow Q \ [] \ b \rightsquigarrow R$

The choice operator describes *alternate* behavior where the actions a and b are also known as *guards*. The operator has the potential to model *nondeterminism* (which is not used in this paper). In addition, we exclude ouput actions (i.e. output control or data assertions) from guards and restrict expression actions in guards to be simple *predicate tests* to facilitate an easy implementation without sacrificing any expressive power.

parcomp $\quad P \Leftarrow Q \parallel R$

This is the familiar *composition* operator found in process calculi which deals with the *synchronization* and *value communication* of behavioral modeling. The semantics of *parcomp* (which is explained in detail in [1]) takes care of the interactions of two *nonatomic* actions a and b by introducing a new class of synchronization called *partial synchronization*. *Partial synchronization* is said to occur between two actions a and b that are refined into subactions $a_1, a2, \ldots, a_m$ and $b_1, b_2, \ldots, b_n$, when some of the a_i and b_j are the same.

In hopCP we define *multiway synchronization* (also called barrier synchronization) as the basic mode of behavioral interaction. A multiway synchronization is said to occur when more than two processes *wait*, *synchronize* and then *resume* their respective activity. This is a very natural construct to model *broadcast* style of communication in VLSI architectures like SIMD systems, bus based systems, etc. We have designed a generic module to implement barrier synchronization with primitive asynchronous circuit elements.

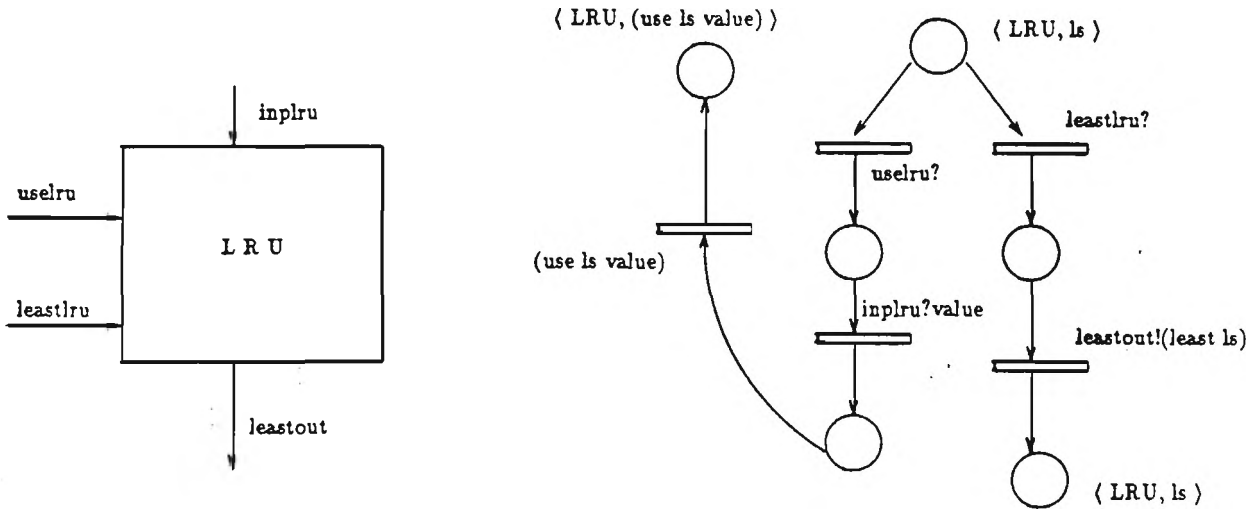


Figure 1: Specification of a LRU Module in hopCP

Structural Specifications in hopCP

In hopCP the basic structural entity being modeled is a *module* whose domain is defined as follows:

$$MODULE = HFG \times PORT$$

where $PORT \subseteq IDENT(Set\ of\ Identifiers)$

A structural description in hopCP primarily depicts the interconnection of modules and the visible connections to the external world. To facilitate such description we propose three *operators* called *rename*, *export* and *connect*. *Rename* is used to connect ports by assigning them a common name while *export* like CSP's *hide* operator provides abstraction. *Connect* is a *structural composition* operator that *infers* the composite behavior of two modules after checking that the physical connection between them is valid in the circuit sense (no unconnected wires, self loops, output to output connections etc). It is naturally defined in terms of *parcomp* which can be viewed as a *behavioral composition* operator. In other words, *connect* checks for well-formedness and derives a module by inferring the HFG component through *parcomp* and ports through suitable inference rules.

Specification of a LRU Module in hopCP

The *textual description* of the HFG denoting the behavior of a LRU module is given below and is illustrated in the next page.

$$\begin{aligned} LRU[ls] \Leftarrow\ & uselru? \rightsquigarrow inplru?value \rightsquigarrow LRU[(use\ ls\ value)] \\ [\!]\ & leastlru? \rightsquigarrow leastout!(least\ ls) \rightsquigarrow LRU[ls]\ . \end{aligned}$$

LRU is the top level *control state* and ls is its associated *data state*. The structural description of the LRU shown in the figure is comprised of the behavioral description LRU and the set of ports $\{uselru, leastlru, inplru, leastout\}$. The behavior is captured by essentially five actions which can be classified as follows:

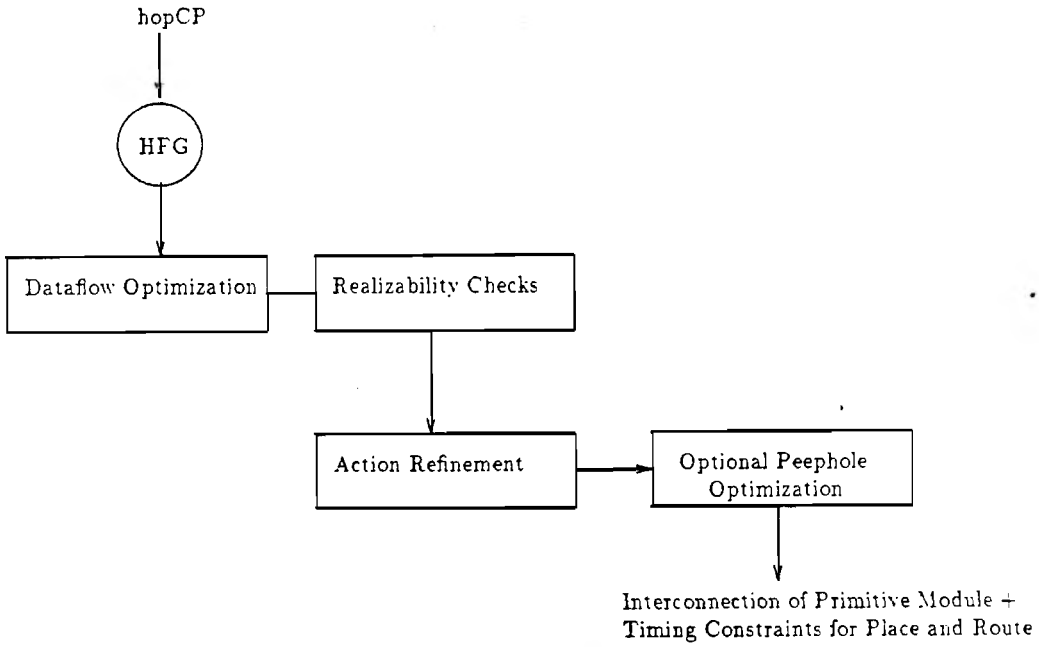


Figure 2: Architecture of Action Refinement Based Compiler

Action	Category	Informal Description
uselru?, *leastlru?*	Control Actions	Input control actions on ports *uselru* and *leastlru*.
inplru?value	Data Action	Acquire value from port *inplru* and bind it to an internal resource.
leastout!(*least ls*)	Data Action	Evaluate expression (*least ls*) and assert the value on port *leastout*.
(*use ls value*)	Expression Action	New data state is the value of the expression (*use ls value*)

3 Architecture of the Action Refinement based Compiler

In this section we will describe a methodology to translate the *HFG* specification of hardware in hopCP into asynchronous circuits. The architecture of the compiler is shown in figure 2.

Note that the flow diagram of the compiler resembles a compiler for programming languages. One can view the *HFG*s as the *intermediate code* and the action refinement based compilation strategy as the code generation scheme. In a nutshell, the compilation of asynchronous circuits from hopCP specifications involves performing global optimization and realizability checks on the *HFG*s and then successively *rewriting* them into an existing set of basic (primitive) modules by a set of action refinement rules taking into account the user prescribed restrictions on the resources and data transfer protocols. We shall briefly describe the *optimization* and *realizability* phases, and then focus on the action

refinement phase which is the focus of this paper.

Data Flow Optimization

Several architecture specific optimizations can be incorporated in the *HFGs* without violating the semantics of the descriptions. These are *global* optimizations in the sense that they capitalize on the knowledge of the overall behavior of the hardware. The onus of showing that the optimizations do not alter the intended semantics is on the compiler writer. The optimizations that can be performed include *reachability analysis* and *live variable analysis* to remove redundant states and expressions from the specification, *eager evaluation of loops* as suggested in [3] to extract *pipelined* behavior, and *delayed acknowledgements* in asynchronous systems to improve the throughput. The details of such optimizations are explored in [2].

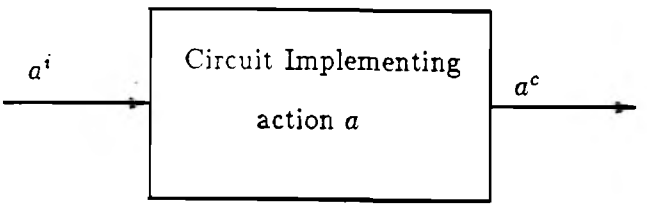
Realizability Checks

Realizability checks ensure that a *HFG* represents *physically realizable* hardware, i.e., it does not suffer from potential disasters like deadlock, livelock, hazards and computation interference. These properties are easily checked in the hopCP formalism by appealing to the properties of *safety*, *liveness* and *persistence* from [5].

Implementing *HFG*s in Asynchronous Hardware

The final phases of the synthesis procedure involve translating the *HFG* specifications into hardware. This involves: (i) identifying a set of basic building blocks (or primitive modules) which have an implementation in VLSI; (ii) formalizing the notion of action refinement and deriving refinement rules for each action category in hopCP.

Every action in hopCP is implemented as an *action block* as shown below, where $a^i, a^c \in SIGNAL$ and denote the initiation and completion of action a respectively.



Definition 3.1 *An action block is an abstraction for a piece of hardware which has explicit* initiate *and* complete *terminals. Execution of an action in a given state of the system is tantamount to* initiating *the corresponding action block and* waiting *for the block to acknowledge the completion of the action. More formally, the implementation of an action can be defined as a function* $IMP : SIGNAL \times Act \mapsto SIGNAL$, *where SIGNAL is the domain of electrical transitions (of voltages).*

The domain of the IMP function corresponds to the action being implemented and its *initiate* signal while the range of the IMP function corresponds to its *completion* signal.

Target Architecture or Instruction Set

A *target architecture* of the compilation scheme is characterized by

- Set of modules known as primitive modules which have a VLSI implementation and satisfy the definition of an *action block.*
- Data transfer protocol (eg: dual rail, transition signaling) and resource specification.

Our module library consists of the basic modules like *merge*, *join*, and *asynchronous latch* described in [4] and macromodules like *call element*, *call-with-boolean-result*, and *asynchronous multiplexer* which are built out of the basic modules. hopCP descriptions of some of the relevant modules is presented in the Appendix for reference.

We view these modules as the *instruction set* of the target architecture while drawing an analogy with conventional compilers. We need not restrict ourselves to the above modules. In fact, we can have more complicated modules in our target architecture as long as they conform to the view of an *action block.* So, potentially we can have ALUs and other combinational modules as primitives, which will then be recognized by the action refinement procedure and used subsequently. This gives us two major advantages which many of the contemporary asynchronous compilation strategies do not possess:

- *retargetability*, which means that we can use the same refinement strategy to compile hardware for different sets of primitive modules;
- ability to use existing hardware for standard functions like shifting, arithmetic and logic as long as they can be cast into the framework of an action block. One possible way of doing it is to use enable input of the module as the initiate terminal and generate a fictitious completion signal by using delays (obtained by careful worst case timing analysis of the circuit, using SPICE for example).

Action Refinement

Action refinement is the crux of the compiler. It involves *incremental* resource allocation and control decomposition. Action Refinement is captured by a simple action grammar whose rules specify an implementation for each action category in hopCP. The end-product of refinement is a *netlist* of asynchronous datapath elements and a *distributed* controller. The action grammar of hopCP is:

$$Act ::= prim\_act \mid Act, Act \mid Act \rightarrow Act \mid Act \mid Act$$

For $a, a_1 \in Act$, *action block refinement* is defined to be the *geometric* or *circuit* interpretation corresponding to the *action refinement.* Action block refinement is shown in figure 3 ($\Longrightarrow$ stands for *is refined as*).

Note that the refinement shown for $(a \implies a_1, Act)$ results in an unbalanced completion tree of C elements (hence is potentially slower). The rule can be rewritten such that we get a *balanced* tree of C elements which is more efficient. Now we will show how to implement different types of actions in hopCP using the above rules for action refinement.

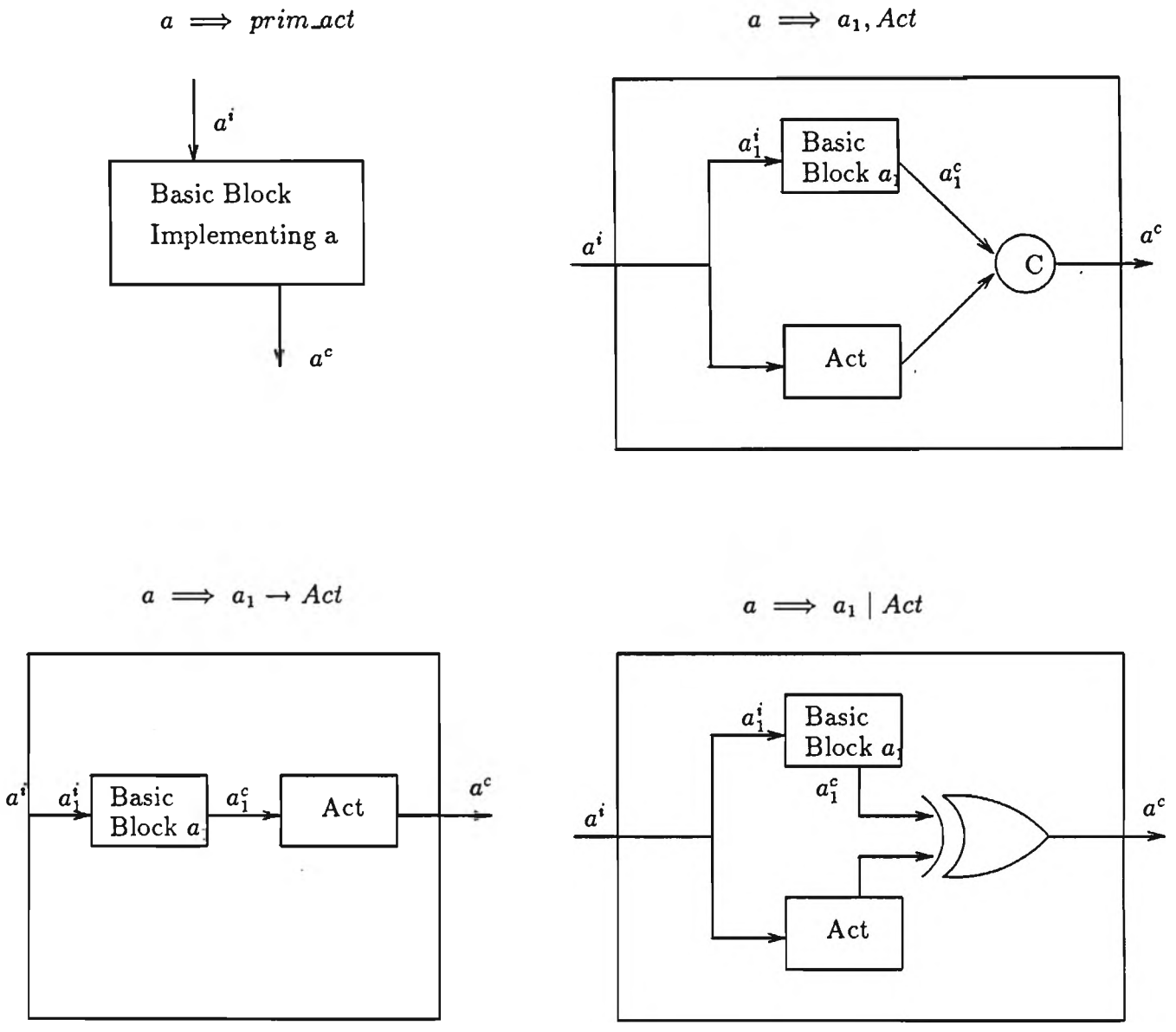
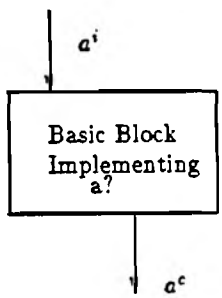
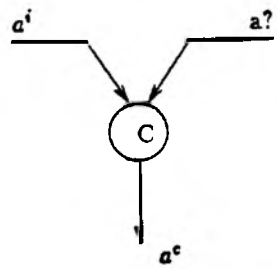


Figure 3: Circuit Representation of Action Refinement

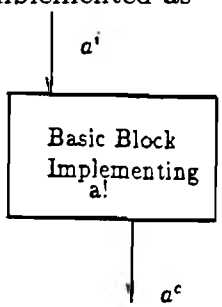
Control Actions

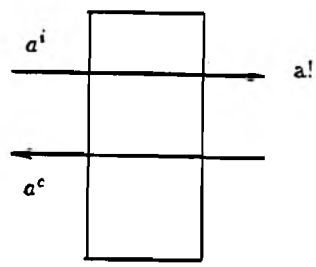
We discussed two kinds of control actions in hopCP namely the *input or passive* actions and the *output or active* actions. Control actions are primitive actions in our implementation strategy. This means that they will be identified with a pre-existing module from our set of basic blocks as shown below:
$(a? \Longrightarrow prim\_act)$ is implemented as





$(a! \Longrightarrow prim\_act)$ is implemented as





Note the identification of explicit *initiate* and *complete* signals in even the basic output action. This abstraction of even viewing an output action as an *action block* (whose manifestation in physical hardware is only pair of wires) helps to maintain the homogeneity of the refinement scheme.

Data Actions

There are several styles of value communication in asynchronous circuits depending on the style of data encoding (single rail or dual rail) and the kind of signaling protocol (transition based or level based). The protocol could be return to zero (four phase) or non return to zero (two phase). In this paper we will illustrate refinements for two phase transition signaling with both single rail data and dual rail data.

Single Rail Data with 2 phase Transition Signaling

We present the refinement with the commonly used bundled data assumption [15] (a bundle of data wires is tagged with a control wire which signifies the *validity* of the data).
Input data action or data query: $p?x$ denotes a data query where p is the name of a port and x is the storage for the value received from the port. Using the action grammar we get

$$p?x \Longrightarrow (p? \rightarrow REG\_x.ld!)$$

where $p?$ is an input control action (whose implementation was discussed above) and

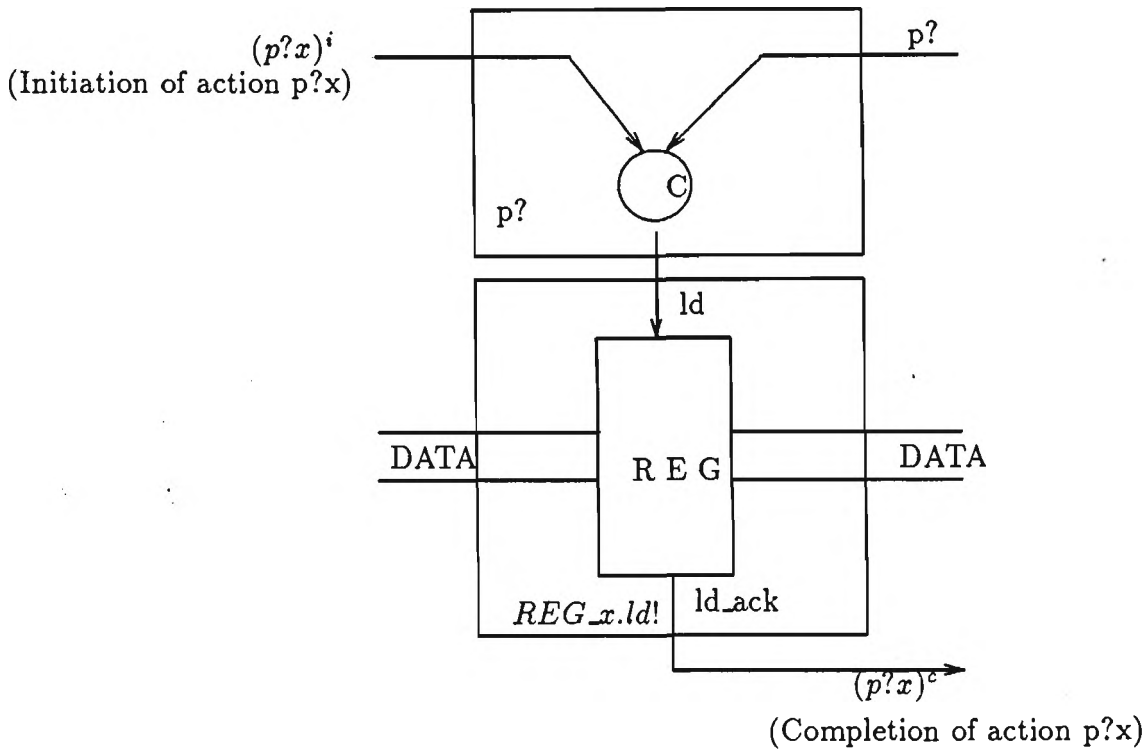


Figure 4: Implementing an Input Data Action with Data Bundling Assumption

$REG\_x.ld!$ is also refined as *prim_act* since it can be implemented directly using the register module in our target architecture. REG_x in the refinement rule indicates the allocation of a register resource to store the incoming value. This is illustrated in figure 4. Output data action or data assertion: $p!exp$ denotes a data assertion, where p is the name of the port and exp is expression whose value has to be asserted. Obviously, exp will be represented by an expression action in hopCP. It is refined as follows:

$$p!exp \implies (eval\_exp \rightarrow p!)$$

where $eval\_exp$ denotes the action block implementing the expression action exp (which will be described soon) and $p!$ denotes an output control action whose implementation has already been discussed. The circuit corresponding to this refinement is illustrated in figure 5.

Note that this strategy is possible only because of our notion of implementing every action as an *action block*. We capitalized on it when we assumed the existence of a block implementing exp with a completion signal which can announce the data on the port $p!$.

Dual Rail Data with 2 phase Transition Signaling

This is an example of a *delay-insensitive* data transfer protocol where every bit of data is transmitted with two wires and the data announces its availability (therefore control

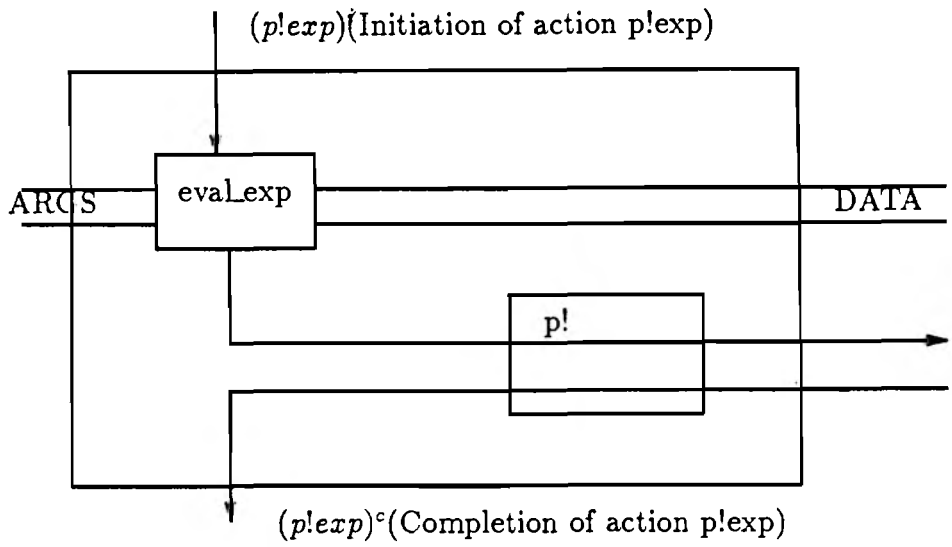


Figure 5: Implementing an Output Data Action with Data Bundling Assumption

signals and bundling assumption are not required). For a n bit data representation we need $2n$ wires. An input data action $p?x$, will be refined hierarchically as shown below:

$$
\begin{aligned}
p?x &\Longrightarrow a_1 \rightarrow a_2 \\
a_1 &\Longrightarrow p_0, p_1, \ldots, p_n \\
p_0 &\Longrightarrow p_{00}? \mid p_{01}? \\
p_1 &\Longrightarrow p_{10}? \mid p_{11}? \\
&\vdots \\
p_n &\Longrightarrow p_{n0}? \mid p_{n1}?
\end{aligned}
$$

where port p capable of handling n bit data is refined into $2n$ input control actions, whose implementation has already been addressed. a_2, as in the implementation of single rail data action, denotes the data storage aspect of the implementation of a input data action. Please note that it is not *always* necessary to store the input value in a register. If the value is being used immediately (and it is the only place that it gets used) then it can picked up *on the fly* without storing it in a register. This is an example of a global optimization which can be detected by the data flow analysis phase of the compiler. Figure 6 illustrates the circuit realization. Note the use of CAL component to implement the actions $p_{i0} \mid p_{i1}$ triggered by a_1. This matches the behavior of the $CAL2X1$ shown in the Appendix.

A data assertion $p!exp$ can be implemented in dual rail data encoding scheme in an analogous fashion using an initial refinement to an expression action and an output control action such that they use $2n$ wires to represent the value of each of the n data bits. We avoid elaborating this refinement.

Expression Actions

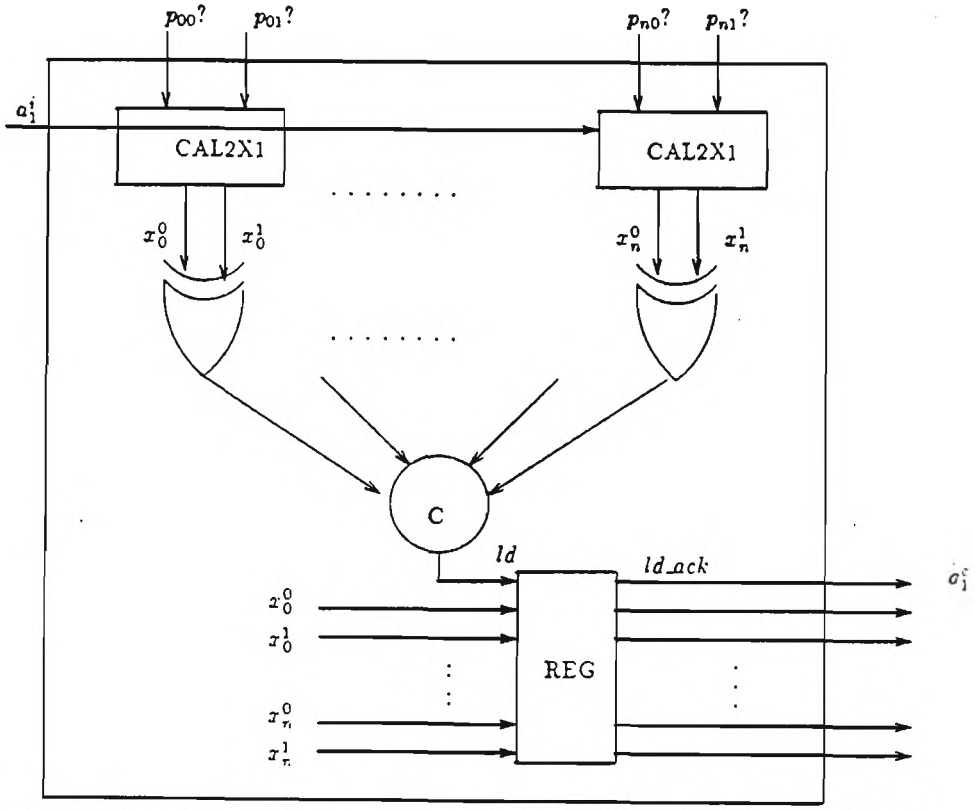


Figure 6: Implementing an Input Data Action with Dual-Rail Data

Compilation of expression actions involves deriving an expression graph from the specification and constructing an evaluation (or reduction) mechanism for each expression action. The evaluation process is captured in hopCP itself and is successively refined using the action grammar described above. This is possible because hopCP is equipped with constructs to specify the operational details of expression evaluation like value communication and synchronization. The compilation of expression actions introduces two new challenges: (i) handling resource constraints and (ii) sharing results to avoid recomputation. We intend to implement expressions actions using *normal-order* lazy graph reduction to evaluate expression graphs which ensures that every redex is evaluated *at-most once* and we plan to handle resource constraints by *intelligent* serialization by taking the granularity of the participating actions. An example of expression action compilation is shown in the next section.

Section Summary

In this section we have shown how an abstract hopCP specification can be systematically refined to a *desired* target architecture (set of primitive modules, resource specification, and data transfer protocol) using three simple action refinement rules. We also briefly hinted on global optimizations and realizability checks. The concept of an *action block* as an abstraction for a physical circuit with initiation and completion signals was the key feature of the compilation scheme. The sequence domain notion of viewing actions as merely occurrences with certain causal relationship was carried through to an almost

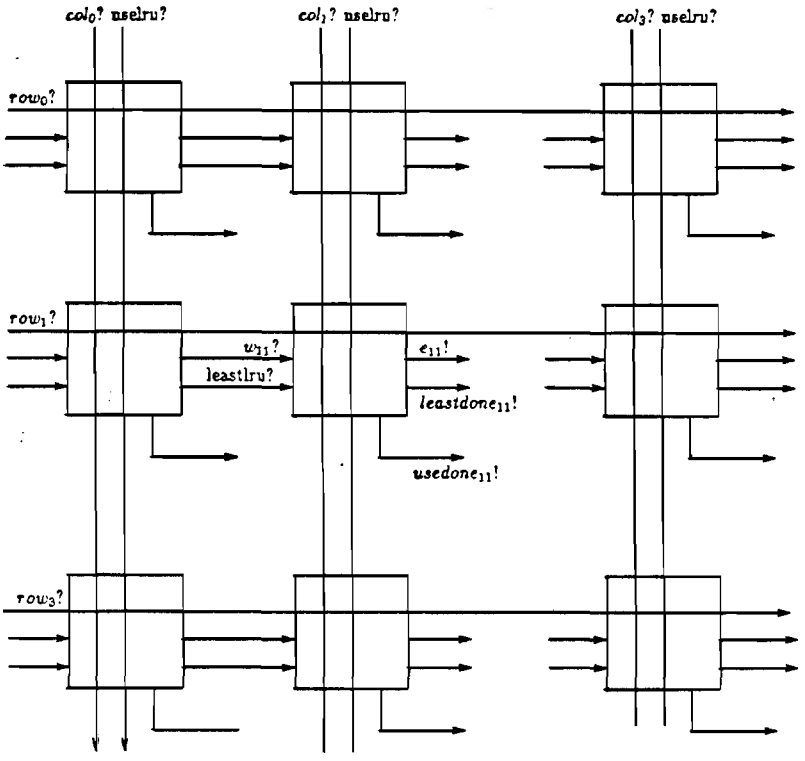


Figure 7: Architecture of a 4 bit LRU

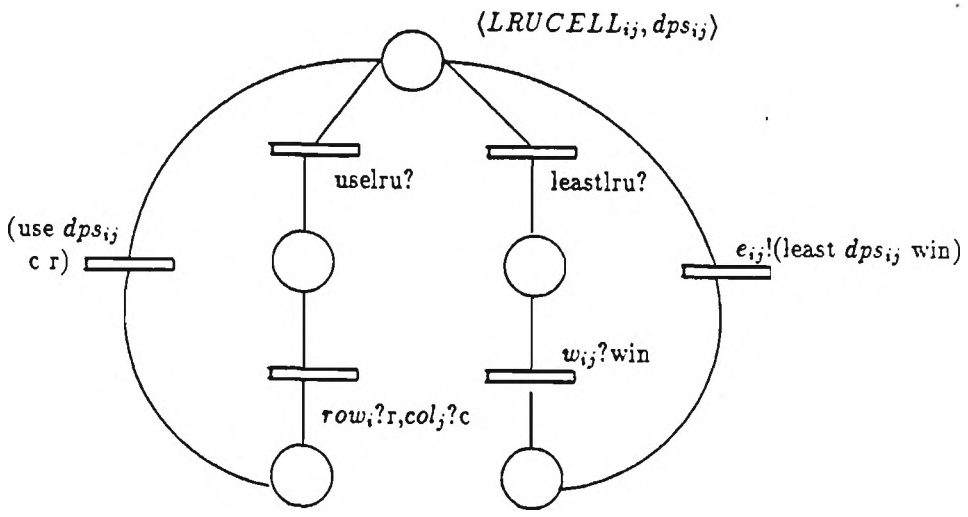
direct implementation by our refinement strategy. The notion of an *action block* and the hierarchical structure of an action block mirrors the notion of an *action* and its refinement. In the next section we will illustrate our compilation scheme on a fairly nontrivial synthesis of a LRU circuit which has a mixture of data, control and expression actions.

4 Synthesis of an Asynchronous LRU circuit

We will illustrate the action refinement based implementation of the LRU specification described in section 2. In literature there are several *algorithms* to implement the LRU computation. We target our compilation to the algorithm described in [16], primarily because it is suitable for a *distributed* implementation. We will assume that the capacity of the LRU is 4 locations (it can be scaled to any number of locations) and will use transition signaling and dual rail data for communication. The internal architecture is shown in figure 7 which is a 4×4 grid for a 4 bit LRU.

Our task of compilation boils down to the implementation of one generic cell from the above architecture, which can then be replicated and systematically connected to give the whole LRU. The behavioral specification of the LRU cell is shown in figure 8.

Note that this is a direct translation of the initial specification shown in figure 1, except for refinement $inplru?value \implies row?r, col?c$ and the change in the data path variable ls, which denotes the aggregate of 4 locations of the LRU, to dps_{ij}, which denotes one bit represented by each cell. The HFG representation of $LRUCELL$ is shown in



$$LRUCELL_{ij}[dps_{ij}] \Leftarrow uselru? \rightsquigarrow row_i?r, col_j?c \rightsquigarrow LRUCELL[(use\ dps_{ij}\ c\ r)]$$
$$[] \quad leastlru? \rightsquigarrow w_{ij}?win \rightsquigarrow e!(least\ dps_{ij}\ win) \rightsquigarrow LRUCELL[dps_{ij}]$$

where the expression actions are described in Standard ML

```
fun use x y z = if (y=0) then 1 else (if (x=1) then 1 else z);
fun least a b = if (b=1) then 1 else a;
```

Figure 8: Behavioral description of a LRUCELL in hopCP

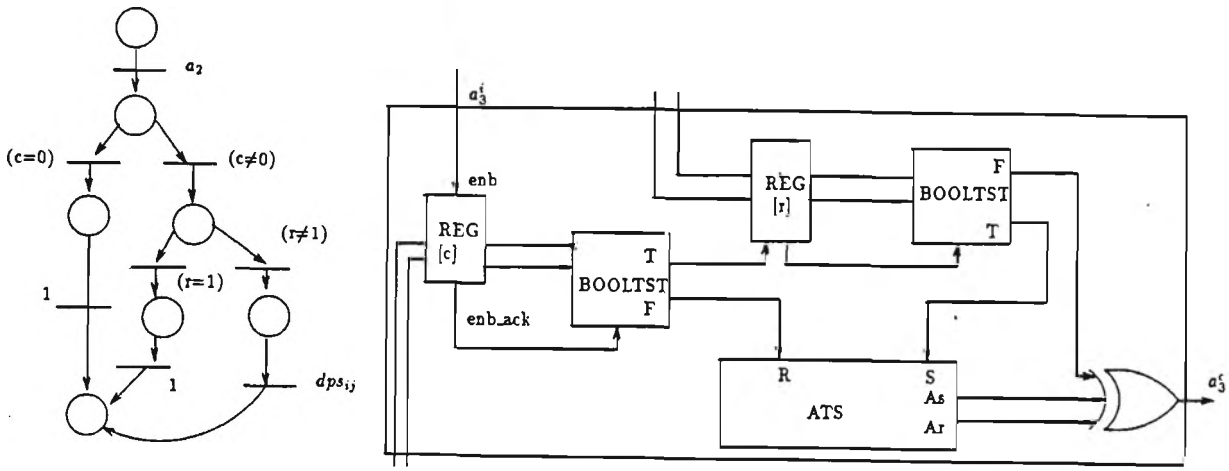


Figure 9: Implementation of Expression Action ($use\ dps_{ij}\ c\ r$)

figure 8. This can be checked for realizability. The actions have been abbreviated by $\{a_1, a_2, a_3, a_4, a_5, a_6\}$ where $a_1 = uselru?$, $a_2 = a_{21}, a_{22}$, $a_{21} = row_i?r$, $a_{22} = col_j?c$, $a_3 = (use\ dps_{ij}\ c\ r)$, $a_4 = leastlru?$, $a_5 = w_{ij}?win$ and $a_6 = e!(least\ dps_{ij}\ win)$. Each action is implemented as an *action block* and the *interconnection* of the blocks is expressed using the IMP function as follows:

$$IMP(init, a_1) = a_1^c;\ \ IMP(a_1^c, a_2) = a_2^c;\ \ IMP(a_2^c, a_3) = usedone_{ij};$$

$$IMP(init, a_4) = a_4^c;\ \ IMP(a_4^c, a_5) = a_5^c;\ \ IMP(a_5^c, a_6) = leastdone_{ij};$$

where, as described in definition 3.1, the first parameter of the IMP function denotes the *initiate* signal for the action a_i, while a_i^c denotes its *completion* signal and *init* is the signal which initiates the whole LRU computation.

For each action a_1 through a_6 we will identify its action category and invoke the compilation rule (and the corresponding circuit) from the description in the previous section. a_1 and a_4 are *input control* actions which have a direct implementation. The implementation of a_2 is interesting because it involves refining it to a_{21} and a_{22} and then using the implementation strategy for the action grammar rule $Act \Longrightarrow Act, Act$ shown earlier. Action a_5 is an input data action which can be directly implemented with our technique described in the previous section.

Implementation of action a_3 involves the compilation of an expression action. The HFG shown in figure 9 captures the evaluation of the expression $(use\ dps_{ij}\ c\ r)$. The predicate tests $(c = 0)$ and $(r = 1)$ are realized using the $BOOLTST$ module and the datapath state dps_{ij} is implemented by a ATS module.

Action a_6 is an output data action which is refined as follows:

$$a_6 \Longrightarrow (eval\ (least\ dps\ win) \rightarrow e!)$$

where (*eval* (*least dps win*)) is an expression action (which can be implemented like a_3) and $e!$ is an output control action whose implementation was discussed in section 3. The final circuit of the $LRUCELL_{ij}$ which includes implementations of all the constituent actions and the interconnect is shown in the Appendix. Note that as discussed in the beginning of this section, sixteen such cells are needed to implement a four bit LRU.

Verification Issues

Formal verification of circuits produced by the action refinement procedure involves checking for functional correctness and establishing properties like safety, liveness, speed independence and delay insensitivity. Our work on verification issues is in progress along the following lines. Functional correctness is ensured by showing that action refinement is a semantic preserving transformation. Safety and liveness properties are checked as in [5] by ensuring that the final HFG in the action refinement scheme is one-safe and free of *dead* states.

By the definition of action-blocks, our action refinement scheme results in circuits which are *speed independent* since every action is triggered by an explicit *initiate* signal and the environment waits for the *completion* signal before doing anything else. Our circuits can be verified for delay insensitivity using Ebergen's approach [6]. We specify each component of our final netlist in Ebergen's Trace Theory, where our $SIGNAL$ domain is interpreted as the set of symbols, in a directed trace structure. The proof obligation is in showing that the directed trace structures of the interconnection of primitive components and the wires in our final netlist corresponds to a *DI decomposition*. This is done by ensuring that we do not have any *dangling wires*, *output interference* and *computation interference*. This can be very easily performed on our example by simulation. We should however remark that the above analysis is valid only if we use dual-rail encoding (or some other delay insensitive data encoding) in our datapaths; on the other hand if use *bundled data* in our refinement we get only speed independent asynchronous circuits.

5 Conclusions and Future Work

We presented a methodology for systematic synthesis of asynchronous circuits from a high level specification. The key feature of our specification formalism was the uniform framework to capture communication, computation, and concurrency aspects of hardware via the causal relationships between a set of *nonatomic* or refinable actions. Salient features of our compilation approach include the ability to synthesize asynchronous circuits with a range of data transfer protocols and primitive modules using a finite set of action refinement rules. We are currently planning to extend our action refinement procedure to synthesize *synchronous* circuits. This is possible because HFGs prescribe only a causal constraint on the actions and hence it is possible to impose a timing discipline like two phase clocking without violating the causal relationship between actions during refinement. We are also looking at incorporating data flow related global optimizations and absolute time constraints in our action refinement based compilation.

Acknowledgements We are indebted to Jo Ebergen of University of Waterloo, Erik Brunvand of University of Utah and Tam-Anh Chu of Cirrus Logic for explaining us their

approaches to asynchronous circuit synthesis several times. We were influenced by their pioneering work in this area. This work is supported in part by NSF grant MIP 8902558 and University of Utah Graduate Research Fellowship.

References

[1] Venkatesh Akella. *hopCP: Language Definition, Semantics and Examples.* Tech Report UUCS-90-10, Dept. of Computer Science, University of Utah, Aug 1990.

[2] Venkatesh Akella and Ganesh Gopalakrishnan. *From Process-Oriented Functional Specifications to Efficient Asynchronous Circuits.* Submitted to *International Conference on Computer Design, ICCD-91*, Cambridge, October 1991.

[3] Francois Anceau. *The Architecture of Microprocessors.* Published by Addison-Wesley, 1985.

[4] Erik Brunvand and Robert F. Sproull. *Translating Concurrent Communicating Programs into Delay-Insensitive Circuits.* International Conference on Computer-aided Design, ICCAD 89, April 1989.

[5] Tam-Anh Chu. *Synthesis of Self-timed VLSI Circuits from Graph Theoretic Specifications.* PhD thesis, Department of EECS, Massachusetts Institute of Technology, September 1987.

[6] Jo C. Ebergen. *Translating Programs into Delay Insensitive Circuits.* Centre for Mathematics and Computer Science, Amsterdam, 1989. CWI Tract 56.

[7] Ganesh C. Gopalakrishnan, Richard Fujimoto, Venkatesh Akella, and Narayana Mani. *HOP: A Process Model for Synchronous Hardware. Semantics, and Experiments in Process Composition. Integration: The VLSI Journal*, 209–247, August 1989.

[8] Y. S. Kwong. *On Reduction of Asynchronous Systems.* Theoretical Computer Science, Vol 5, pages 25-50, 1977.

[9] Leslie Lamport. *On Interprocess Communication* Report No. 8, DEC System Research Center, Palo Alto, December 1985.

[10] Alain J. Martin. *Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits* Developments in Concurrency and Communication, C. A. R. Hoare ed. Addison-Wesley, 1990.

[11] Michael C. McFarland, Alice C. Parker, and Raul Camposano. *The High-Level Synthesis of Digital Systems.* In Proceedings of the IEEE, pages 301-317, February 1990.

[12] C. A. Mead and L. Conway. *An Introduction to VLSI Systems.* Addison Wesley, 1980.

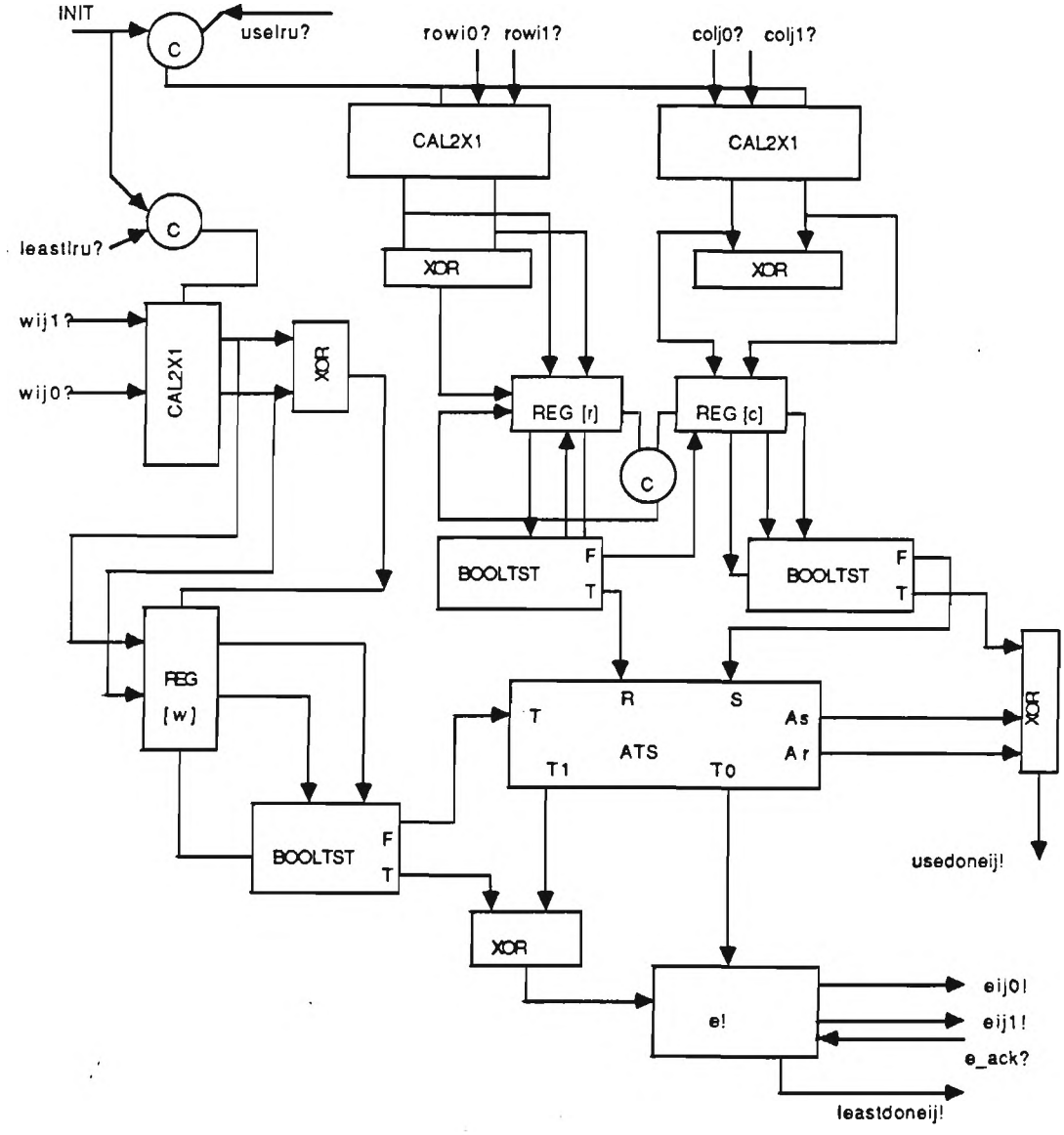
[13] George J. Milne. *CIRCAL and the Representation of Communication, Concurrency, and Time. ACM Transactions on Programming Languages and Systems*, 7(2):270–298, April 1985.

[14] Vaughan Pratt. Modeling Concurrency with Partial Orders. *International Journal of Parallel Programming*, (1):33–72, February 1986.

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

[16] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice Hall, Englewood Cliffs, NJ, 1987.

A Complete Circuit of a LRUCELL



B hopCP description of Basic Modules

We shall provide the hopCP descriptions of the primitive modules used in the synthesis of the LRU circuit are:

JOIN: (also known as C element)

$$CELM\ [\,] \Leftarrow a?, b? \rightsquigarrow c! \rightsquigarrow CELM\ [\,]$$

MERGE: (also known as XOR element)

$$\begin{aligned} XOR\ [\,] \ \Leftarrow\ & a? \rightsquigarrow c! \rightsquigarrow XOR\ [\,] \\ [\!] \ & b? \rightsquigarrow c! \rightsquigarrow XOR\ [\,] \end{aligned}$$

REGISTER: (latch with initiate and complete signals) (also known as event-controlled latch)

$$\begin{aligned} REG\ [x] \ \Leftarrow\ & ld? \rightsquigarrow ld\_ack! \rightsquigarrow REG\ [x] \\ [\!] \ & enb? \rightsquigarrow enb\_ack! \rightsquigarrow REG\ [x] \end{aligned}$$

BOOLTST: (datapath module to test if the input is 1 or 0)

$$\begin{aligned} BOOLTST\ [\,] \Leftarrow p?x \ \rightsquigarrow\ & (x = 0) \rightsquigarrow T! \rightsquigarrow BOOLTST\ [\,] \\ [\!] \ & (x = 1) \rightsquigarrow F! \rightsquigarrow BOOLTST\ [\,] \end{aligned}$$

CAL2X1: (also known as decision wait module)

$$\begin{aligned} CAL2X1\ [\,] \ \Leftarrow\ & a?, b? \rightsquigarrow p! \rightsquigarrow CAL2X1\ [\,] \\ [\!] \ & a?, c? \rightsquigarrow q! \rightsquigarrow CAL2X1\ [\,] \end{aligned}$$

ATS: (also known as test set module)

$$\begin{aligned} ATS\ [x] \Leftarrow\ & S? \rightsquigarrow As! \rightsquigarrow ATS\ [1] \\ & [\!] R? \rightsquigarrow Ar! \rightsquigarrow ATS\ [0] \\ & [\!] T? \rightsquigarrow (\ (x = 0) \rightsquigarrow T0! \rightsquigarrow ATS\ [x] \\ & \quad [\!] (x = 1) \rightsquigarrow T1! \rightsquigarrow ATS\ [x]\ ) \end{aligned}$$