Carlstedt Elektronik AB
computer research and development

ISSN 1650-2027 15; ISBN 91-89546-15-6; version 2011-10-21 page 1

rp8601

a complete applicative processing concept

by

Gunnar Carlstedt, Carlstedt Elektronik AB

Abstract:

An applicative real-time multiprocessor is described. Machine language, communication and integrated circuit implementation are applicative and based on graph rewriting.

User processes are applicative and is a derivation from HDLs. The machine language is a full-fledged applicative language with influences from functional and logic languages.

High order functions are complemented by a more general concept where programs are representations described in the basic language. It is an extension of a quoting mechanism.

Times for ports are synchronized. Input and output are timed. A real-time system performs a unification between a set of courses of events and the port values. Binding strength defines the port direction.

The machine is based on graph reduction described by equivalence rules. The control of a multiprocessor is by emitting such rules and implementing a structure arithmetic unit rewriting a two level expression. Garbage collection is a real time execution in parallel with the problem domain.

The multiprocessor consists of a rich network and a massive number of reduction processors. They store expressions as words as well in communication ports, clock, arithmetic unit as memory. The memory is associative.

Keywords:

allocation, alternative, applicative process, applicative, behavior, data-path, equivalence rule, garbage collection, graph rewriting, H language, identifier domain, integrated circuit, interpretation, massive parallelism, memory, multiprocessor, processor, real-time, representation, scheduling, unification

INTRODUCTION

I performed from the early 1970's a number of multiprocessor architecture projects. With the experience together with funding from Swedish air-force a project aiming at a new design concept was formed. The project was named **rp8601** (=reduction processor, year 1986, number 1). Further on ASEA (now ABB Ltd) took over the project. ASEA was split into ABB Ltd and Incentive AB, who made a considerable investment in the project. The project was shut down 1994 due to other interests of the mother company.

This report describes the rp8601 architecture. During this large project a number of changes were made due to requests by the funder. However the report describes the original ideas together with the experience from the project.

The concept WYSWYG (What You See What You Get) for document editors was introduced during the 1980´s. The change was dramatic compared to earlier methods. Young people of today would think of the earlier methods as awkward, slow, farfetched and error prone.

Early computer languages like FORTRAN developed to more rich languages. The size of the languages exploded, with Algol 68 as an extreme. Subsets as Jovial, Pascal and Modula was much less in size. Unix came with C. The development was by software system engineers. Meanwhile a mathematically based community introduced languages based on *lambda calculus*. Lisp was followed by a number of functional languages like ML and further on by Haskell. Meantime a new paradigm, the *logic language* based on *Horn clauses,* was created with Prolog as its first language. These late languages are applicative and would be the computer language counterpart to WYSWYG.

None of these languages could directly model hardware, time and processes. Simula 67 introduced a concept of message passing. Later on more mathematically based initiatives created *process algebras* such as CSP and CCS. Message passing could be viewed as a side effect. Even if

they use a functional approach when describing creating and receiving the messages the system described would not be side-effect free.

Other system maintenance and control is not supported by the languages. Instead other types of maintenance concepts are shoe horned into the software.

Meanwhile the silicon circuit community designed their abstractions. They described hardware. In contrary to the software approaches everything was in parallel. Spice was an early language where everything was components and analogue signals. Later VHDL was introduced - a committee work, probably with people from different communities. It worked in two modes - the *procedural* and the *applicative*! Hardware description languages, HDL, introduced the concept of component, parameterised components, libraries and communication between components. They also introduced wave forms - a way to describe the output/input for a process.

The applicative languages has now been present for about 30 years and still they are not popular. The community is rather small and in the development was more to be expected. However organizations using the concept have good experience. The HDLs are a must in the industry. At the time rp8601 started there was hardly no industrial projects using applicative languages and the experience was low.

The wish list from system designers was long. The rp8601 was a challenge to replace the von Neumann computer architecture by an applicative approach. Much was to be developed. Were there any key issues that has to be solved in order to facilitate a complete applicative computer system?

PROBLEMS - CHALLENGES

Traditional von Neumann computers have influenced the thinking in computing in such a high way that it is hard to step outside this paradigm. This is the case from high level architecture and down to gates. The rp8601 is an attempt to step outside that paradigm. This will influence almost everything from languages over architecture down to gates. In this section a birds view is used to point out challenges to solve.

Massive multiprocessing

rp8601 approach is a scalable architecture from small and simple to large massive parallel approaches. Traditional high performance computers consists of rather few traditional processors with some type of communication devices. Applications are allocated to processors and memory cells.

The rp8601 goal is to have a space of molecules being processors. Within this structure the representation of an application is floating. There is some type of communication between the molecules. From performance point of view this allows the highest possible allocation of executing resources. From flexibility point of view the structure could be squeezed to fit an application.

The idea is that the problem should be loaded in the computer by almost neglecting its structure in a corresponding way that cash memories implements an invisible memory hierarchy.

In order to facilitate this, the problem loaded must be represented in such a way that it contains all information on synchronization, communication etc needed in a distributed environment. From my understanding the only available form is a directed graph. The vertices are tiny processes controlling itself and its directed edges. One of the possible execution mechanisms is graph reduction.

The rp8601 implements this by using a small process called closure. It stores an expression in the size of 4 words.

Key issues: performance, flexibility, scalability

Trace graph - a tool

Traditionally, single processor application performance analysis is based on counting processor cycles. The hardware influence is almost only a weighted average between execution time of instructions. When cash memories are used the problem of "locality" or working set complicates the analysis.

When using a multiprocessor based on traditional processors with or without cash memories this analysis is not easy. Communicating processes form an execution order that is almost impossible to analyze. Only with a detailed knowledge of the application an estimate could be done.

The graph reduction situation is quite different. The graph is well specified. A method based on a "trace graph" is used. It contains all expressions used when executing the graph. It corresponds to an expansion of the graph to all its parts used during the execution.

An analysis of the trace graph gives:

- each edge in the graph uses one read, one write operation and one communication,
- each expression-rewrite corresponds to one vertex rewrite
- the trace graph has a depth being the length from the root to the leaves

- the size is described by the number vertices and the number of edges

In a large problem the depth is generally much less than the size. The effective latency time is the longest of the vertices rewrites (including reads and writes) and communication over edges.

The time of a vertex rewrite is constant. The communication time is dependent on the "length" of the edge. For a given graph the number of vertex operations is fixed. There is a mean number of edges directed to a vertex. There is a ratio of link communication time to vertex rewrite time. These three numbers signify an execution of an application.

To improve performance vertices with many edges are to be duplicated and placed more local. Vertices are allocated in order to reduce the edge length.

The challenge is to use such a tool to analyze how to allocate and schedule a problem.

Key issues: performance analysis

Address domains

Traditional multiprocessors may have a shared memory or a number of local memories. They use an address being number to depict a memory cell. Because the rp8601 consists of a structure of molecule processors such an address scheme is not possible.

All edges pointing to the same vertex are named a net in rp8601. The address is a mechanism to specify the nets. There are graph rewrite operations on nets. Therefore the mechanism of addressing is to be integrated with the rewrite operations.

The rp8601 implements the addressing by one domain for each net. A domain specifies a number of molecules. There is a constructor for a domain specifying its size and its placement in the molecule structure. An address is a number depicting a molecule and the domain constructor.

The constructor must be able to specify overlapping domains of the same size or any size in order not to place restrictions on the graph. Performance is dependent on the size and placement of a domain. It is heavily influenced by the communication structure.

The challenge is to design the domain constructor.

Key issues: addressing, synchronization, critical region, performance

Partitioning

Traditional computers may have ports and processors implemented in separate subsystems. Special purpose protocols are used to maintain a system structure and the intermodule communication. Traditionally, such communication may be: on low level I2C busses, device specific like SCSI, and system general as ethernet.

rp8601 uses a molecular approach. The application is an expression being loaded. The computer may be divided into a number of separate clusters each containing a number of molecules. With some mechanism parts of the expressions are allocated to clusters. By using the domain mechanism the the communication works.

In the long run this may be a method to integrate computers to ONE system. Communication is by expressions instead of awkward protocols. However high level intersystem communication protocol must still exists - they are the applications.

Some applications are rather stiff and controlled. Others are more lose where subsystems may be separated, i e lose communication. Such systems may be started again and reconnected.

rp8601 allows such disconnection, of course with a delay of pending expressions. However, there is a more fundamental issue to be solved: time. When systems are separated they loose time synchronization. Time elapses with different speed in the subsystems. When reconnected they have different time. It is not understood how to handle this.

Key issues: distributed computing, protocol, time, real systems

Graph reduction

Graph reduction is straight forward to implement in a single processor. Using many processors complexity increases. The problem to solve is addressing, critical regions and synchronization.

rp8601 solves this by letting each vertex be a separate process. There are two rewrite operations: one for just the vertex and another for the net pointing to the vertex. The first one is simply like a traditional instruction. The challenge is to design the net rewrite.

rp8601 uses the H protocol in the domain of a net. The protocol publishes equivalence rules. They may be performed any time and several times.

Key issue: applicative, synchronization, critical region

Real time - behavior

Traditional computers don't know about time. A special register giving a date may be used. Behavior is implemented by synchronization messages. In small single processors this may be sufficient, but for distributed and large machines problems

arise. The arrival order of message may cause racing problems and in-determinant results.

The challenge is to make real-time first class. In order to do this the mechanism of a port and the basic (boot) execution mechanism has to be given a theoretical model. In the rp8601 case an applicative model. As I understand this has never been done before!

In order to be first class real-time behavior must fulfill a number of properties: storable, readable, referable, be operated, recorded and play backed. When recorded or play backed it may involve ports. Behavior may be either a structure of transactions or the corresponding with absolute or relative time stamps. Ports must be synchronized.

The rp8601 models a behavior by a data structure. As such all types of first class operations may be performed on it. Clock is a global feature. Absolute time occurs synchronized in all molecules. The port mechanism is a behavior simultaneously recording and performing play back.

Key issues: first class, port, synchronization, protocol, theoretical model

Parallel modules, boot and I/O

In a traditional computer the boot mechanism is arbitrary. Starting a system with many or a vast amount of processors is feasible. However when starting at different time communication protocols have to perform a synchronized start. Using a vast amount of molecular processors a more delicate mechanism is needed.

The challenge is to understand what computing is in the regards of ports and internal execution.

rp8601 solves this in a very different way. First, modules are expressions (data structures) being fixed or behaviors. Modules communicate by unifying parts of such expressions . Ports are behaviors. Input/output is performed by unifying an internal behavior with the port behavior!

Booting is performed by emitting an expression into the computer. Several such expression may be emitted at any time. Thus booting may be performed for subsystems using different ports. A read-only memory may emit such a structure when started.

The mechanism is applicative.

Key issues: synchronization, port, booting, communication, applicative

Executing memory

A traditional von Neumann computer like x86 (not rp8601) has a memory and a processor. Both are separated, however a cash memory may be included in the processor package. The interface is a protocol performing read and write of memory locations. The processor has very little structure except for some registers. It could be considered a heap of gates.

The basis for rp8601 is a molecular approach. Of course, it is not feasible to have one physical processor in each memory cell, but memory cells may be multiplexed by a processor. This mimics a memory cell having a virtual processing capacity.

Graph reduction is generally not numerics. Instead it is based on pattern matching. As such a completely different electrical implementation could be used. It is based on transportation and associative comparisons in the memory, data-path and the control module. The challenge is to design such a processor.

The H processor implements one minute domain.

Key issues: processor, control, circuit

Complete approach

In traditional computers there is an instruction set architecture. The architecture has no formal mathematical model. Thus any type of execution may be performed. This results in a flexibility. It can be used for any type of maintenance or implementation of tools etc. An example is a debugger.

Graph reduction implements an application being applicative. On the lowest level the implementation uses an "instruction"-level language. It could be a traditional instruction set architecture or an applicative language. If an applicative language is used there must be means to implement the mentioned maintenance and tools.

The challenge is to design an application machine level language fulfilling these requirements.

The rp8601 uses the H language being a single level approach.

Key issues: machine language, low level control

Representation

I am watching a person performing the “real” steps of throwing a dice every 3rd second with the result 4, 6, and 2. On a paper I write “ throwing a dice every 3rd second with the result 4, 6, and 2”.

The first one is a real behavior and the second one a representation for this course of events. Instead of using plain text other more strict forms may be used.

Another example is “5+3”. Written on a paper it is 3 tokens. Written in a computer language it represents the constant 8. But if I use an editor and want

to change it to “5+4”, the text stands for 8, how do I proceed?

The new concept of “representation” is defined: there is the duality *item* and *representation* of item. By making a **derepr** of a representation an instance is created. Thus **derepr** “5+3” results in 5+3 and subsequently in 8. The **repr** goes the other way but is tricky, there are an infinite number of ways to express 8.

A simplified version of this is the Quote mechanism in Lisp. Modern languages don’t have the concept. Functional languages use high-order functions to alter the semantics of a function, but this is not at all the same.

The introduction of *representation* gives a new understanding of symbols, function application, module instantiation, version handling etc. It also enables a mathematical understanding of execution, debugging and tracing.

Key issues:language, representation

Behavior

Parallelism, the process concept and communication are all part of the same issue. The understanding of them is inherited from the sequential form of earlier languages. They are just other forms of doing assign (:=).

A new concept “*course of events*” (sv = händelseförlopp) is defined. Here are some examples:

1. a movie 51 minute long
2. throwing a a dice every 3rd second with the result 4, 6, and 2.

Both could be modeled. They can be stored, copied, and read. They can be performed any number of times. They could be analyzed, a part may be extracted, run stepwise, different speed, forward and backward. They are first class!

In the second case the outcome may be another course of events

3. throwing a a dice every 3rd second with the result 1, 4, and 3.

A new concept “*behavior*” is defined. It is a set of course of events. This set should be considered all course of events for a particular process. Thus “throwing a dice 3 times every 3rd second” has the outcome of 6^3 number of course of events. Another behavior would be when the delay between the throws is within the interval 0…10s.

The behavior concept described is a “natural one”. A mathematical one used in computing consists of other concepts also being behaviors. Thus a computational behavior is the cross product of

- ★ abstractions
- ★ symbols
- ★ modules
- ★ executions
- ★ resources

Each as to be explain further. Below the word “item” is used for what is described. An item may be a course of events or a behavior.

Abstraction

This is the top down method to construct. Generally it includes the use of modules for the decomposition, and basic symbols for depicting items. Symbols is used to form relations being equalities. In classical languages this is the intention of a function body. As such it may evolve by time, i e being a course of events!

Symbols

This is a syntactic element. Hardware, global address, name, character etc are examples. They have an appearance but only depict a particular “knowledge”. A more complicated one is the language used. As such it may evolve by time, i e being a course of events.

Modules

This a “component”. Several such are fitted together. Generally parameters are used to define them. In almost all computer languages equality is used to fit them together, however implemented in different ways (argument, variable, element of message etc). The components may evolve, i e being a course of events.

Resources

This is the “physical” part of an implementation. The part is an item or generally a set of items. As such they could be physical parts as harbors, vessels etc as processors and memory cells.

Execution

This is the procedure to make an instance of an item using resources. As such it includes the binding of values to memory cells, ports and the rewrite process (order and allocation in time) to change the values of memory cells. Or as in the example above bind cargo to vessels and harbors.

Key issues: time, behavior, protocol

Indeterminism

I am using a calculator pressing 5 + 3 = and the response is 8 as it should. Then calculator continues to return 8 every time I repeat the sequence. Now assume that there is an error in the calculator. It returns arbitrary result each time. The first example is a *deterministic* and the second one a *nonde-*

terministic behavior. This is the established mathematical understanding.

Assume now that we have a real time computer system. Two experiments are done with the same history. In both cases exactly the same input is used by the computer. A computer is an extremely large finite state machine. Thus it produces the same result in both cases. Now assuming that arbitrary but equal input is used. The results would be the same. The result is deterministic!

On the other hand, if we record the input to the system, even if the history is the same, generally the next input is variable, thus the input from *real world* is nondeterministic!

The input to the computer is read by a device with somewhat uncertainty (jittering) in time. Thus two identical input sequences may not produce the same result from the input device. The input is *nondeterministic*. The output from the computer may therefore be nondeterministic.

As a designer of a real time system I want the same result from the same input, i e deterministic.

The general opinion in the computer community is that real time systems are nondeterministic and should so be! The reason behind this is probably that the implementation is constructed by message passing! **My understanding is different**.

Using a large asynchronous multiprocessor using message passing technique may cause message arrival order to be nondeterministic. The system may therefore behave nondeterministically.

In order to produce deterministic results the message passing semantics have to be redesigned or replaced by other mechanisms.

Using a perfect method in a large system would cause the system to be stiff. Because of design flaws, machine errors, overload situations the output cannot be guaranteed. In such cases nondeterminism may be a design goal!

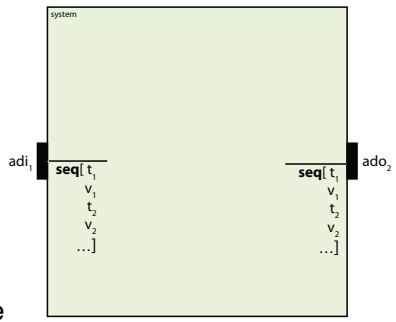
Key issues: process, communication

OVERVIEW

A guided tour within the rp8601 is in this section. The explanations are sketchy and simplified in order to make the description more comprehensive.

Process concept

Ports are for input and output. The port has an outside being the real world and an inside being a data structure. The general idea behind the rp8601 port is that the internal data structure should be a model of the real world behavior regardless of being an input or output port.



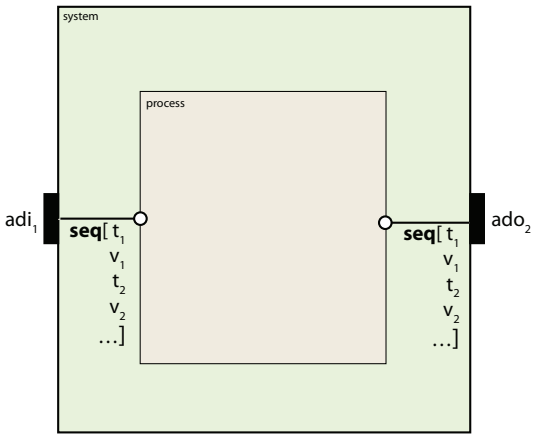
The internal data structure is a course of events. Beside an example is shown. It consists of a list tagged **seq** and a number of time stamps (t123.3 etc) separating the values (56 etc) on the real world side.

seq[t123.3 ***56*** t127.2 ***23*** t129.4]

There is a system process (behavior) that is the boot mechanism, see figure below:

unify[
  apply[
    par[*adi1 ado2*]]
  bootstrap]

The boot is a unification between the ports and the bootstrap. The ports are described by the symbols (hardware addresses *adi1* and *ado2* depicting the ports). The standard list in rp8601 is tagged **par**. Here the list is considered a representation for the ports.

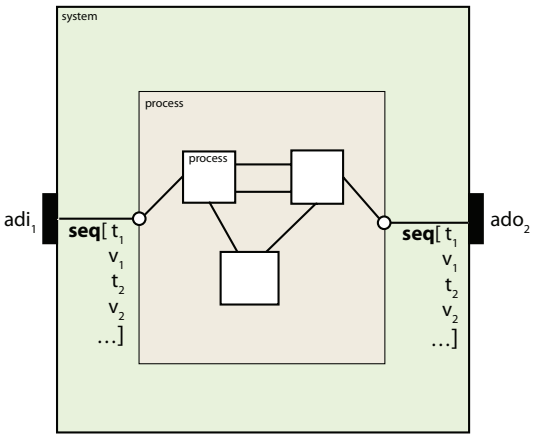


The **apply**-construct transforms the representation into the port internal course of events.

unify[
  par[*in1 out2*]
  bootstrap]

The unification between the boot strap and the port behavior causes the binding of the internal behavior of the bootstrap to the in- and outports.

As an example the bootstrap process could further be divided into 3 processes (components) as shown below:



The lines (nets) are behaviors local to the bootstrap process. They are somewhat the same as variables in normal languages and nets in hardware description languages. Each of these nets are bound with unification to the local processes. The local components are relations between their behaviors connected to their interface.

All these behaviors (excluding the input output) are deterministic. The concept of unification, i e equality, makes the total structure deterministic.

In a unification of two expressions one could be bound to a value and the other not. The equality forces the unbound to the value of the bound one. Thus there is a flow of value from the bound to the unbound. This mechanism causes the flow of values from input towards the output and eventually to the output port data structure.

The bootstrap module concept is the same as the internal module concept, thus making the module concept orthogonal throughout the system.

Generators

Generators are modules used to create predefined representation for behaviors. It is a module having a hidden internal structure and an external being its interface. Generators corresponds to parameterised components in hardware description languages and procedures in logic language. A distant relative in the process algebras are processes.

In this section some few toy like generators are used to show the concept. The earlier discussed bootstrap is described. Generally the behavior consists of two elements where the first is the input and the last the output.

Constant

Assume that the input port has one and only one course of events. The output port the corresponding. This is a toy behavior of no use:

```
par[
    seq[t123.3 56 t127.2 23 t129.4]
    seq[t123.3 48 t127.2 56 t129.4]]
```

It is not known what happens when input differs from the sequence.

Alternatives

Assume that two such toy course of events exists. They are described using an **alt**-structure containing the two toy course of events.

```
alt[
    par[
        seq[t123.3 56 t127.2 23 t129.4]
        seq[t123.3 48 t127.2 56 t129.4]]
    par[
        seq[t123.3 33 t127.2 24 t129.4]
        seq[t123.3 48 t127.2 33 t129.4]]
```

The unification of the bootstrap distributes the unification to each of the course of events. Assume that the input is the same as in the earlier example (constant). Among the two alternatives the second differ from the input. It is pruned from the alternatives and only the first course of events remains.

Any

The last example with two course of events is turned into two behaviors. The first received value 56 is replaced by a predefined behavior **any**. It stands for all possible course of events. It could be considered a wild card.

```
alt[
    par[
        seq[t123.3 any t127.2 23 t129.4]
        seq[t123.3 48 t127.2 56 t129.4]]
    par[
        seq[t123.3 any t127.2 24 t129.4]
        seq[t123.3 48 t127.2 33 t129.4]]]
```

In this case the input time sequence must have the timing shown. The first value received could be any value. The second value must be either 23 or 24, selecting 56 or 33 as output.

Symbol

The earlier bootstrap is somewhat changed. The input value should be transported to the output by a symbol. the symbol describes one behavior:

```
def[
    par[
        seq[t123.3 S t127.2 any t129.4]
        seq[t123.3 48 t127.2 S t129.4]]
    S=any]
```

A **def**-construct is a placeholders for values. The first element is the proper behavior being the result of the construct. The remaining elements are values to be referred. In this case the S=**any**-behavior. The S= is just a way to tell the reader that this is the S element.

In this case the proper values are the earlier input-output behavior modified by a wildcard **any** in the last interval. In the input behavior the first value

uses a symbol S. The second value in the output behavior uses the same symbol S.

During execution the input binds S by unification of the input value and **any** resulting in S=input value. Later on the output uses this symbol for output.

If the timing sequence is fulfilled all values from the first input interval is transferred to the output in the second interval.

Copies

In the earlier example S was used a some sort of variable. Constants can be used in the same way. The earlier example is modifier with the symbol T:

```
def[
    par[
        seq[ T S t127.2 any t129.4]
        seq[ T 48 t127.2 S t129.4]]
    S=any
    T= t123.3]
```

The value t123.3 is shared. This is an important feature for representations. Symbols are the only way to specify sharing. Thus all shared structures are in the **def**-structure.

Representation

In the earlier example the concept of representation has been ignored. Below a component and three nets are used. The component consists of a **def'**-tree structure. The nets by a **par**-list.

```
unify[
    par[any 1 2]
    apply[
        def'[
            par[sel'[2] sel'[2] any']
            any']]]
```

The **apply**-construct (as a unary operator) transforms the the representation to its behavior.

The **def'**-structure is a representation. When transformed it becomes a **def**-structure, the ' differs. Its first element is the behavior proper being a **par**-structure with two **sel'**-elements and one **any'** element. They are transformed to **sel** and **any** respectively. The **par**-construct is identical in the representation.

Links are not possible to have i the representation. Instead **sel**-elements are used. It selects an addressed element in the surrounding **def**-structure. The address is a tree-index allowing the access of an element far down in a tree structure.

Thus the rewritten example is:

```
unify[
    par[any 1 2]
    def[
        par[S S any]
        S=any]]]
```

In some steps it will be rewritten to:

```
par[1 1 2]
```

Function application

Function application is such a general concept that it must have full support. The function application is implemented by the non-unary **apply**-construct.

An example from extended lambda calculus where two numbers 1 and 2 are summed in an embedded expression is:

```
λ($1, $2).($1+$2) (1, 2)
```

The same expression is expressed as follows:

```
apply[
    def'[
        apply'[+' sel'[2] sel'[3]]
        any'
        any']
    1 2]
```

Functions have no restrictions on formal arguments as described by the **any'**-elements. Thus a shortcut is performed:

```
def[
    apply[+ sel[2] sel[3]]
    1
    2]
```

The function is transformed from representation. The operator symbol +' has been converted to an instruction. In the **def**-construct the actual arguments are placed after the proper element.

In the next two steps the **sel**-elements selects the actual arguments 1 and 2.

```
def[
        apply[+ 1 2]
        1
        2]
```

During the next step the function application is performed:

```
def[ 3 1 2]
```

The entire **def**-construct is canonical. The result is the proper element:

```
3
```

Logical structure

The execution mechanism could be modeled in an abstract logic model described here. The expressions used in the examples are compound expressions delimited by parenthesis:

```
unify[5, expr, a]
```

This structure could not be stored in a memory cell. Instead a number of global symbols, here depicted

identifiers are used. A definition of the expression for the identifier is placed in each memory cell. It is a function depicted *closure*.

id_1, id_2, …, id_n
id_2=**unify**[5, id_{expr}, **sel**(n_a)] @(env_{defa}, tags)

By using the identifiers an arbitrary non circular graph may be constructed. The elements to the left of @ corresponds directly to the elements of the expression above. To the right of @ there is sometimes a reference to a **def**-construct. A number of tags specifies how the closure should be rewritten.

There are some important variants:

id_2=**unify**[5, id_{expr}, **sel**(n_a)] @(env_{defa}, tags)
id_2=**unify**[5, id_{expr}, **sel**(n_a)] @(env_{defa}, garb)
id_2=@(**free**)

The first one is the general one used. The second one is tagged to be tested whether it is garbage. The last one is an unused memory cell.

Execution is by rewriting expressions. They form a number of trees. Any closure may be rewritten. Some would not cause a change. Memory cells are tagged to indicate if to be rewritten, waiting for result and how far to be rewritten. Only pending expressions are rewritten.

In a gross view an expression tree is followed from root to leaves. In some case in a pre-traversal order. Signaling up or down in the tree is performed by rewriting identifiers *id*. Generally, identical identifiers are rewritten in one step.

Garbage collection is performed by rewriting the garb tagged expressions. Those not referred by another expression are turned into **free** cells.

Multiprocessor

The logic structure is implemented in a multiprocessor. It consists of a communication network, ports and reduction processors. The individual units have a hardware address.

Each reduction processor contains a number of memory cells, one larger cell able to store a two level expression and some network cells.

The core unit contains the larger memory cell. A numeric arithmetic unit is connected to this cell. Within the large memory cell values of single elements could be permuted.

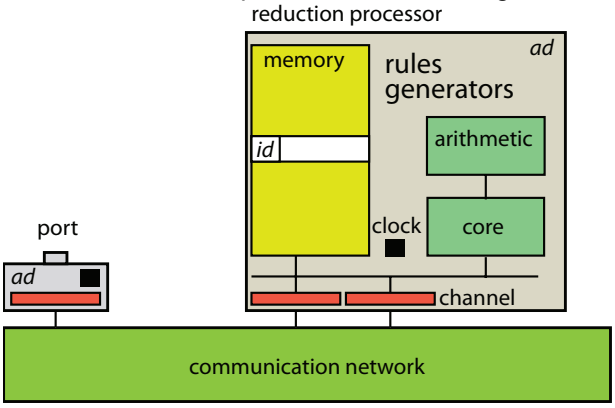
There are some channel units performing communication through the communication network. The channel unit contains one network memory cell. It can store one closure.

Execution is performed either by a network cell or a core cell. All rewriting is performed by inspecting a closure. Therefore a global intra-processor rewriting is performed by distributing a particular closure to those processors involved.

Rewriting is of two types:

- an *identifier* is rewritten. In parallel all fields in all closures within the local memory is rewritten. By a search mechanism only involved elements are changed.
- a *closure* is rewritten. Depending on the expression one or two levels are used. If two, the other son expressions are read. Rewritings is by permutation of elements and setting tags.

Rewrite rules are implemented in a set of gates.



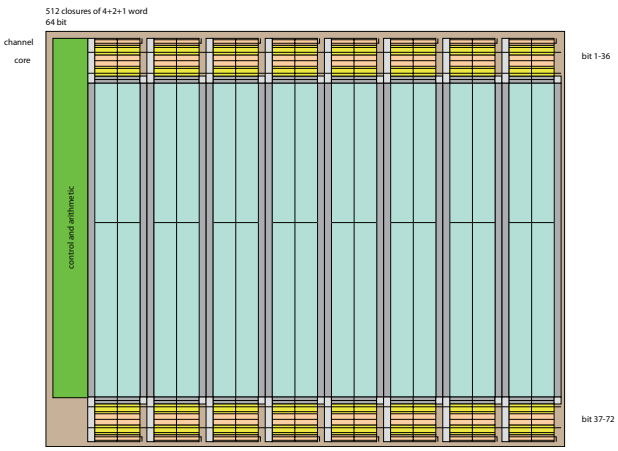
Time synchronism

Ports time stamp values and control output by time. Clocks in all ports are synchronized to sub-microsecond level.

Each separate chip has one phase locked clock. Phase-locking is performed using the communication in the communication network.

Chip architecture

There was an implementation under design. Experience from that design showed that the main footprint should be altered. This change is shown here. The early design read out a full closure. The new one reads the closure elements in sequence.



The main part of the chip is memory. The shown memory consists of 8 banks of memory cells. Half of the bits are read/written on one edge, the remaining on the opposite edge. The memory is associative and uses the element as search key.

The core unit and the channel units are place along the other side.

THE H LANGUAGE

The language of the rp8601 is "**H**". There are several H languages, all being the same but expressed differently:

- the $H_{machine}$ language consists of a number of lists and scalars. Lists are interconnected by links. Scalars can only be found within lists.
- the H_{abs} is language corresponding to $H_{machine}$ where linked expressions are placed inline. H_{abs} may also be just a scalar. Sharing an expression is expressed by a local symbol referring a definition. This is only a syntactic way to express links. The definition of the shared object is (any order may be used and scope is not defined):

  exp : symbol

There are sugared variants. They are not discussed here. The H_{abs} is used here because it is the only language that can be written in text.

The H language has a mapping onto real world properties or traits. When written as a syntactic concept it has just an existence, but when executed in a machine the relation to the real world is used. Some of the mechanism can only be understood with the knowledge of a machine. They are not discussed in this section but in subsequent sections. Essentially these mechanisms are not unique for rp8601, but are general concepts of all computing paradigms.

The H language is an applicative language. It has a semantics enabling the control of almost all what is necessary in administrating a system - without having side effects!

The H language is both data and program - all with the same syntax. There is no difference between data and program. Thus there is only one syntactic form.

The H language semantics is not defined by the methods used for classic, functional or other languages. The syntactic form is all. Other languages has some side-effects both in their semantics and syntax.

The semantics is here expressed by equivalence rules. They take one syntactic form and describes it to be equivalent to another form. The syntactic form is here depicted expression.

A reduction is the property of a machine using the H language. It assumes a direction of the equivalence. The machine performs graph reduction by replacing an expression by another expression. In the equivalence rules presented further down the left expression is rewritten to the right expression.

H syntax

Below the complete syntax of Habs is shown. There are token, scalar and list forms.

A ***token*** is shown as one bold text.

A scalar is shown as a text ***scalar****(v)* with parenthesis containing an argument *v*. It stands for a numeric value where *n* indicates an integer and *r* a real value.

A list is shown as a text ***list****[…]* followed by a list. If the list is specified "$e_1 \ldots e_n$" it could have any number of elements. "$e_1\ e_2 \ldots e_n$" has at least one element. Otherwise it stands for a fixed number specified by the symbols.

delta
epsilon
token
  token **= some'**, **something'**, **empty'**,
    nothing', **any'**

int (n)
cyc (n)
real (r)
cycfix (r)

symb[n]
phys[n]

type [$e_1 \ldots e_n$]
  type = **par**, **seq**, **symb'**, **phys'**, **part'**, **bag'**,
    alt', **pri'**, **if', unify'**, **def'**, **sel'**, **apply'**,
    repr', **red'**, **alloc'**, **net'**, **time'**
part[$e_1 \ldots e_n$]
some
something
empty

bag[$e_1 \ldots e_n$][1]
alt[$e_1 \ldots e_n$]
pri[$e_1 \ldots e_n$]
if[$e_1 \ldots e_n$]
unify[$e_1 \ldots e_n$]
nothing
any

def[e $e_1 \ldots e_n$]
sel[$i_1\ i_2 \ldots i_n$]
sel[e $i_1\ i_2 \ldots i_n$]

[1] the bag construct was not a used in the project

apply[e_1]
apply[e_1 e_2 … e_n]
repr[e_1]

red[e_1 e_2 … e_n]
error

alloc[e ad t_f t_t]
alloc[e ad t_f]
alloc[e ad]
alloc[e]

net[id ad]

time[T, acc_{nom}, acc_{ach}]

The $H_{machine}$ language have additional syntactic constructs. They are the general ones recoded and packed into a single scalar:

sel[i_1 i_2 … i_n]
phys[e_1]
ID[e_1] -- the H_{abs} link

The list variants also have a tag that represents the state of reduction. It will not be defined here. There are two for a user very important ones - **breadth** and **depth** used in scheduling.

Numerics

In conventional computers there was first integer and then floating point. Computer languages have been influenced by this and implement these two numeric types. However working with real-time, physical space and sets there is more to be asked for. Thus the H-language has more numeric types.

Integer and reals are used and implemented as in conventional computers.

Cyclic numbers

Some real world properties are cyclic as angles. Another is infinite spaces. Time is a good example. Time could be viewed as being a number in the range minus to plus infinity and having infinite precision. When time is 23:59 on the day and you add 2 minutes it is 00:01 in a cyclic number system with the length of 24:00.

Assume that there is an infinite range of N. A new number Ncyc is defined as:

$$N_{cyc} = N \bmod I_{cyc}$$

used to depict a value in *N*. The representation is ambiguous, there are infinite amount of values in N corresponding to N_{cyc}. However when using this representation the user “knows” which one being proper. There are operations on this cyclic type:

- difference gives a normal number. Cyclic plus or minus a normal number gives a cyclic. Comparing two cyclic is possible if difference is less than $I_{cyc}/2$.

A cyclic integer is obvious. Compared to a conventional integer it is overflow and the most significant bit that must be handled different. Compares must handle the first bit different.

A cyclic real must be considered different from floating point. The exponent should be considered fixed. Thus the real corresponds to an integer with the scale 2^{exp}. Difference is performed by alignment to the smallest exponent and with normalization. In the add-operation alignment is to the cyclic number (not to the one with largest exponent as in floating point) and there is no normalization.

The length of the cycle I_{cyc} could be handle in powers of 2. It is possible to go from one cyclic space to another. Therefore the exponent may be changed arbitrarily together with a corresponding shift with $2^{-\Delta exp}$ of the mantissa. When increasing the exponent an “earlier” value with the new exponent is used as normal. Its least significant bits are set in the mantissa. The application “knows” that the operation is correct.

Built-in ranges

The machine performs operations on sets when rewriting unify numerics. Range operations are common in pattern matching. Therefore the numerics are appearing in three sets

≤ n: …n
= n: n
≥ n: n…

The middle one is the common numeric one. The other two are sets. The result of a unification is obvious except for the one defining a range, however being canonical.

Lists

There are typed lists. The use of different lists is an implementation necessary to separate constructors in the application domain from H_{abs} syntactic elements. The general list structure has type **par**. **seq** is part of the process concept and is used to differ it from par.

The list consists either of the elements specified, or by a **part** construct. The latter could be considered an open list representing zero or more elements. The part construct partitions a typed list into a hierarchical set of partitions:

par[
  e_1
  part[
    e_2
    part[e_3 e_4]
    part[e_5]]
  part[]
  e_6]

The construct is needed to implement lists. A part containing no elements is equivalent with the token **empty**. The token **empty** in a list could be dropped. A coloured empty token **point** to be used in array arithmetic with infinite spaces has also been considered.

Parts of lists may be shared.

The part construct also facilitates implementation of generators altering the amount of elements in lists, especially where large or unknown amount of elements are used. Tail recursion is eliminated.

Bags and alternatives

Mathematical theory have sets and bags. Elements in sets are unique. Elements in bags may be identical. The rp8601 bag construct is something between: it contains elements and it is not known whether they are unique. Identical elements placed in the bag is not guaranteed to be separated. It is the use of the bag construct that defines whether it is a true mathematical bag or not.

The bag construct is particularly used when pruning alternatives in unification. In this case the mathematical bag semantics is not guaranteed.

Bags are built analogous to lists. Element order in a bag has no meaning. Corresponding to **part** the **alt** construct is used. **nothing** stands for an alt construct being empty. In a bag or alt construct the nothing token can be dropped.

bag-constructs may be shared, but not **alt** constructs.

In many constructs the **alt** and **pri** construct may be expanded outside the construct e g:

par[e1 **alt**[e2 e3] e4]

is equivalent to

alt[
par[e1 e2 e4]
par[e1 e3 e4]

Condition

Conditions are special cases to handle alternatives. The **pri** construct orders alternatives. The construct contains an ordered list of alternatives. The construct equals the first one not being nothing.

The **if** construct is a list of expressions. It could be considered a guarded expression. If some expression is **nothing** the construct equals **nothing** otherwise the first expression.

The combination of pri and if constructs forms a conditional expression:

pri[
if[e_1 c_{11} …]
if[e_2 c_{21} …]
…
if[e_n c_{n1} …]]

Sharing and constraints

Graphs are directed constructs. Graphs are formed by constructing (= referencing, see below) using already defined graphs e_2 … e_n. No loops are possible. Lists form an hierarchical construct. They don't allow sharing.

Constraints are conditions defining when an expression is valid. The **def**-construct is used to implement sharing and constraints:

def[e_1 e_2 … e_n]

is considered to be a typed list with special properties. If the typed list exists, it is the equivalent to e_1. The typed list is said to be the *environment* to all sub-expressions of the typed list. When referenced all elements e_1 e_2 … e_n are present.

The **sel** construct selects another sub-expression using indexes i_1 i_2 … i_n, each being an integer, using the environment *env*:

sel[i_1 i_2 … i_n] @env

which is equivalent to

sel[env i_1 i_2 … i_n]

The first index should be greater than 1 in order not to refer to the first expression. The current understanding here is not clear.

Representation

In classic programming there are data and program. In H they are the same. However the mode of an expression is either the text or the mathematical interpretation. In some applicative languages there is a Quote function. It is related to representation.

In H there are two forms: the representation and expression forms. Both are expressions. A representation form may be the result of an expression. An expression form is the result of an **appl** (-ication) construct:

apply[e_1]

is recursively applied to all subexpressions of e_1. Any subexpression not yet evaluated is evaluated before the application.

Almost all syntactic elements have equivalence rules used in reductions. In order not to perform such a reduction, an element has a dual correspondence. Thus **unify** has the representation form **unify'**. The corresponding holds for their simplified forms as **any** and **any'**. The simplest scalars and

typed lists do not have dual forms. They are the same in both forms.

High order forms of representation is protected by a **repr** construct. An application of such a construct returns the protected expression. By cascading **repr** constructs, any high order mechanism may be constructed.

The high order forms allow high order forms to be "compiled" by a compilation function. This type of functionality could not be expressed in normal applicative languages. High order functions are currently not available in logic languages or process algebras. This is due to the currently used high order function mechanism. The representation in *H* is a more general mechanism.

The "functional application" is a special form of the application using several expressions. The first expression e_1 is of representation form and the other expressions are actual arguments as shown earlier:

apply[e_1 e_2 … e_n]

is equivalent to

```
def[
    apply[ e1 ]
    e2
    …
    en]
```

It is very simple to rewrite. The **apply**[e1] creates an instance of e_1. Subsequently necessary **sel** constructs access the *environment* in which the actual arguments are present.

Special generators

All canonical expressions are an **alt** construct of expressions. It contains all expressions of the H language. When counting expressions it counts to 1. The token **any** stands for any such expression.

One of those expressions is **something**. The contents is hidden. **some** (plural) stands for a **part** construct with at least one **something**.

Unification

Unification is an arithmetic operation. In other languages it is generally a binding of a symbol to an expressions. The definition of H is in the other direction, it expresses a vast number of alternative expressions where only some are possible. Those possible have proper values for the symbols. The unification evaluates an equality construct, either it results in a proper expression or it prunes the alternative in charge.

An implementation of the symbol arithmetic uses the unification to bind symbols! The complexity comes there.

Unification is simple. If several expressions included are identical, the result is such an expression otherwise **nothing**.

Error

In computing there may be hardware errors. In order to cope with this the machine language, not H_{abs}, has an additional equivalence rule:

$e_1 = e_2$

That is: any expression e_1 is equivalent to any expression e_2. Hence the machine is assumed to perform random reduction. To this is added: with low frequency. This is definitively not applicative computing. This is the way hardware errors work.

error is a scalar token and **red** (-undancy) is a construct being an ordered list of expressions. The overall view of its use is that **error** propagates upwards in expressions reaching eventually a **red** construct where it is stopped. It is removed from the red construct. The red construct is equivalent to its first expression. Several red constructs may be included in an expression at various places.

Beware of the use of this mechanism. It is dangerous, however probably one of the few ways to cope with errors. Its use could be visualized:

> An expression contains subexpressions randomly altering to new ones. The "new" expressions are from user point acceptable but may be different.

A good example of its use: two expressions *a* and *b* in a **red** construct are stored in different physical memories and sharing the same expression e_r. An alloc construct, see further down, places an expression in a physical location:

```
def[
    red[
a:      alloc[ er ad1 ]
b:      alloc[ er ad2 ]]
    er : exp ]
```

Symbols and machine symbols

Symbols are different from symbols in other computer languages. They should not be mixed-up.

Assume there are some unique properties. Some could be expressed as data structures others are real world traits. Examples are "that car", "the character L" etc. In the real world they just exist. In understanding the property there may be some description *exps* or none!

The H language has a representation form being:

symb'[exps]

The *exps* expression may be empty. There is a corresponding expression form:

symb[ ]

It is used in the following way: Some representation form (e g a program) has an internal expression being **symb'**. Performing apply on this structure creates a **symb** construct. Its only purpose is to be a placeholder for something not known. Using the the same **symb'**[*exps*] constructs at different position creates *different* placeholders, hence they are considered different when comparing.

A created **symb** construct could be referenced from several expressions. They can also be compared.

Physical address

There is a special case of symbols, the machine symbols, here denoted **phys**. These symbols map onto hardware items of the actual physical machine. The difference with a symbol is that the *exps* of the phys is known by the machine. Such symbols are used for "instruction code", "port address" etc.

The semantical understanding is: when building the hardware the symbol was created. The symbol is carried by the phys construct to the machine.

Internal symbols

The H language does not have symbols. Instead it uses **sel** constructs to select particular expressions from an environment. This selection mechanism is equivalent to an internal symbol.

Some expressions of an environment are the token **any**. At this point it acts as a generator for an arbitrary H expression.

Semantics

A graph in a machine defines an expression P. Let each symbol be assigned an expression. The equivalence rules expands this expression to PE. In some cases PE may be empty. Now let each symbol have all arbitrary expressions of H. The cross products of these expressions generates a set PES. Most symbol expression combinations do not generate expressions. The remaining expressions of PES define the expressions in a symbol combination.

Symbol arithmetic

In functional applications symbols are placeholders bound to an expression. In logic applications there are a relation between symbols. These relations causes constraints on symbols. Arithmetic on symbols is to solve these constraints.

isym used below is included for the explanation, but is not part of the H language.

The graph consists of an arbitrary directed graph. It uses an arbitrary amount of **def** constructs to specify shared data structures, each being an **isym** construct. They are referred to by **sel** constructs. The references could be part of **unify** constructs.

Together the **isym**, **sel** and **unify** constructs form an equation system to solve. The solution may result in several alternative graphs. Thus the symbols are constrained. The complexity of the solving mechanism could be drafted in the levels:

1. all referred sharing points are bound to canonical expressions. This is the functional case. There is no additional symbol arithmetic.
2. at each sharing point there is at most a canonical expression bound to a symbol. This case corresponds to "equalities between several symbols" assigned to one expression.
3. as 2 but there are several unifications containing a symbol and canonical expressions. The unifications can always be reduced to a canonical expression.
4. as 3 but there are a set of alternative solutions each needing separate reductions.
5. other

The H language could express all four. The machine must have more built in rules when using more complex solutions. The current approach is to solve the first 2 complexities.

A sketch of arithmetic type 2

An isym construct can have these forms:

1. **isym**[]
2. **isym**[e_{can}]
3. **isym**[$e_{can,1}$ … $e_{can,N}$]
4. **isym**[s_2]
5. **isym**[s_2 e_{can}]
6. **isym**[$e(s_2)$]

In the first (1.) variant the symbol is not bound. If this is the final solution it is equivalent to **any**. In the second example the symbol is bound to the expression e_{can} being canonical. The third variant can be further reduced. The **isym** construct is here equivalent to a **unify** construct. If it rewrites to **nothing** the isym expression equals nothing and it is further propagated in the graph.

The fourth (4.) case contains a reference to another symbol. It states that the current symbol is equivalent to s_1. The situation is as follows:

s_1: **isym**[s_2]
s_2: **isym**[e]

Two symbols are identical. In this case one of them should refer to the other. Using several symbols

referring to each other may cause loops. Symbols are ordered in H. Symbols may refer to symbols before. If $s_2 < s_1$ the referred order is correct. If $s_2 = s_1$ the s_2 should be dropped. Otherwise referencing should be swapped:

s_1: **isym**[ ]
s_2: **isym**[s_1 e]

The fifth (5.) case is the same as above. If $s_2 < s_1$ the symbols should be swapped as above. Otherwise the expression *e* should be moved:

s_1: **isym**[e]
s_2: **isym**[s_1]

The sixth (6.) example contains an expression with a subexpression being a symbol s_2. In order not to form a loop it must refer to a symbol before. It does not involve additional symbol arithmetic.

A sketch of arithmetic type 3

Assume that there is a definition containing one symbol involved in a unification:

s_1: **isym**[e_1]
unify[s_1 e_2]

All such constructs are lifted from the expressions into the symbols. The remaining part may have other constructs causing the binding of a symbol (not shown):

s_1: **isym**[e_1 **unify**[e_2]]
s_1

There are the following symbols being constrained by the expressions e_1 and e_2:

s_1: **isym**[e_1]
s_2: **isym**[s_1 e_2]

In the same way as above the constrain expression should be moved towards symbols before. It would result in:

s_1: **isym**[e_1 e_2]
s_2: **isym**[s_1]

The first symbol could be further reduced as in case 3 above.

A sketch of arithmetic type 4

Assume that somewhere in a **def** construct there is a **pri** construct. It contains alternative expressions each referring to the symbol:

```
def[
    pri[
        e1(s1)
        e2(s1)]
    s1: isym[]]
```

Each of these alternatives may have constraints. They may differ. To solve this, the **pri** construct is moved outside the **def** construct:

```
pri[
    def[   e1(s1)
           s1: isym[]]
    def[   e1(s2)
           s2: isym[]]]
```

This creates two parallel symbol equation systems. Their solutions are separated. In this case symbol s_1 is first evaluated. If e_1 does not produces a result s_2 has to be solved.

Arithmetic of type 5

Some arithmetic could not be solved except using a special algorithm, e g solving for a trigonometric result equalling its argument:

```
s1: isym[]
unify[
    s1
    apply[ cos s1]]
```

H equivalence rules

A root expression is an expression being a top of a graph of expression. A machine may contain any number of root expression. Zero expressions results in an idling machine. Many expression results in separate executions. In this section one root expression is concerned.

Wellformed

Assume that there is one root expression. It contains generators modeling a “large” graph, in some cases infinite. This graph is rewritten to another graph.

The H language could be reduce to ZERO expressions in some cases. In order to execute there is a main reduction direction to create an existing root expression. Such an expression does only exist if it is not reduced to **error** or **nothing**. The concept of wellformed is introduced:

- An expression is *soft error free* if no expression could create an error from **error’** that propagates to the root. There are simple means by introducing **red** constructs that will guarantee this.
- An expression is *nothing free* if no expression of any type can generate a **nothing** token propagating to the root. There are simple means by introducing **pri** constructs that will guarantee this.

From this follows that all expressions introduced into the by time expanding root graph must be wellformed.

Because the concept of representation is used, any type of user application may result in an expression of representation form. In some cases it is from an input through a port or in other cases from a pseu-

dorandom generator. Hence it is clear that future such expressions are not known.

For proper graph reduction the machine should know this before executing. This is of course impossible. Even if the machine during the start tries to expand the root graph as much as possible it would be impossible to find out the wellformedness.

To lighten the burden of wellformedness the wellformedness should apply to each time step of port behavior and with applicable input data. In such cases the machine would perform graph reductions for the root graph creating one additional output.

During one such step it may be the case that wellformedness is not guaranteed and the root graph is reduced to zero expressions. From H language point of view the already produced output is a failure. No execution should have occurred. The machine and user have made an error that the machine cannot foresee.

Here it assumed that such false output may be the case and is forgiven!

Driving

With driving means the mechanism initiating each graph reduction.

The root graph is rewritten in order to find expressions to be **nothing** free. **nothing** may be introduced by an expression of representation form, an empty **alt** construct or a **unify** construct. An expression tree must be scanned for this. Unifications must be performed.

The need for a particular subexpression causes the lazy evaluation mechanism to traverse down an expression to the particular subexpression.

Reduction rules

In a separate table the reduction rules are sketched. They are defined by equality. The left expression is reduced to the right expression. There are also restrictions on each expression defining the order the rewrite rules shall be used. They are not shown.

PROCESS CONCEPT

In this section there is an orientation about applicative processes. In rp8601 the concept is named behavior.

Assume there are a number of resources r_n, time periods T_n and expressions e_n:

$c_n = [\ e_n\ r_n\ T_n\ ]$
$C = \{\ldots\ c_n\ \ldots\}$

A course of events c_n is the triplet of these. A complete course of events C is a set of such course of events. T is absolute time period from t_b to t_e. The resource r_n is to house the course of events. It could be a virtual one like speed of subject or the voltage on a physical port. The expression e_n characterizes the course of events.

The course of events above is a continuous course of events. If the time period is changed to just an event at time T_n it is a discrete course of events.

A real course of events is what is actual happening in the physical world. The course of events could not be seen. It just exists. A modeled course of events is what is described above, or just short course of events.

Modeled course of events may be included in time sequences where course of events are time slices. Each such course of events could be modeled relative its beginning. Such a time slice is a relative course of events in contrast to an absolute. Instead of a time period it has a duration. The discrete one has a pulse.

Model

Current real-time systems cannot model course of events. Instead there is an interaction between the real behavior outside the system and the internal system. Animation software describes modeled course of events. It could be played. A modeled course of events in the rp8601 is implemented with a data structure. It may be analyzed, simulated or played.

Behaviors

There are a number of course of events C_n included in a behavior B:

$B = \{\ldots\ C_n\ \ldots\}$

Such a set could be the complete set of course of events for a particular item. It could be real word item, a real-time system or some internal process in a computer system. Examples of course of events and behavior expressed in the H language are:

C: **seq**[c_a c_b c_c c_d c_e c_f]

B: **alt**[
seq[c_a c_b c_c c_d c_e c_f]
seq[c_a c_b c_g c_h c_i]
seq[c_j c_k c_l c_m]]

A real-time system uses a program that can produce the behavior *B*. After the system has executed it exhibit *C*. The real time system runs and at a particular time, an occasion, it exhibits a state. At the beginning of execution the real-time system has the state *B*, at the end of execution state *C*. At an occasion *b* the state *S*:

S: **seq**[c_a c_b
alt[

ConElem = {**delta**, **epsilon**, **some'**, **something'**, **empty**', **nothing'**, **any'**, **int**(n), **cyc**(n), **real**(r)**, cycfix**(r), **some, something**}
ConH = {**symb'**, **phys'**, **part'**, **bag'**, **alt'**, **pri'**, **if'**, **unify'**, **def'**, **sel'**, **apply'**, r**epr'**, **red'**, **alloc'**, **net**', **time**'}
ConList = {**par**, **seq**} ∪ ConH
RedElem = {**nothing**, **any**}
RedList = {**part, if, unify, def, sel**, **apply, repr**, **alloc, net**, **time**}
OPS = **phys'**[…] being built-in operations
INS = **symb**[n] being built-in instructions

type [e_1 … **part**[e_x … e_y] … e_n] = *type* [e_1 … e_x … e_y … e_n] *type*∈ConList
part[ ] = **empty**
type [e_1 … **empty** … e_n] = *type* [e_1 … … e_n] *type*∈ConList

bag [e_1 … **alt**[e_x … e_y] … e_n] = **bag** [e_1 … e_x … e_y … e_n]
type [e_1 … **nothing** … e_n] = *type* [e_1 … … e_n] *type*∈{**bag**, **alt**, **pri**}
type [e_1 … **nothing** … e_n] = **nothing** *type*∈ConList ∪ RedList
type [ ] = **nothing** *type*∈{**alt**, **pri**, **if**}

type [e_1] = e_1 *type*∈{**alt**, **pri**, **if**, **unify**}
type [e_1 … e_x … e_y … e_n] = *type* [e_1 … e_y … e_x … e_n] *type*∈{**bag**, **alt**, **unify**}
type [e_1 e_2 … e_x … e_y … e_n] = *type* [e_1 … e_y … e_x … e_n] *type*∈{**if**, **def**}, x>1, y>1
type [e_1 … *type*[e_x … e_y] … e_n] = *type* [e_1 … e_x … e_y … e_n] *type*∈{**alt**, **pri**, **if**, **unify**}
type1[e_1 … *type2*[e_x … e_y] … e_n]
= *type*2[*type1* [e_1 … e_x … e_n] … *type1*[e_1 … e_y … e_n]] *type1*∈ConList ∪ RedList, *type2* = {**alt**, **pri**}

if[e_1 e_2 … e_n] = e_1 ∀x>1: notNothing(e_x)
if[e_1 e_2 …**any** … e_n] = **if**[e_1 e_2 … … e_n]

unify[ ] = **any**
unify[e_1 …**any** … e_n] = **unify**[e_1 … … e_n]
unify[e e] = e *type*∈ConElem
unify[**symb**[n] **symb**[n]] = **symb**[n]
unify[*type*[a_1 … a_n] *type*[b_1 … b_n]] = t*ype*[**unify**[a_1 b_1] … **unify**[a_n b_n]]
unify[*type1*[a_1 … a_x] *type2*[b_1 … b_y]] = **nothing** *(x ≠ y* ∪ *type1≠type2)* ∩ *type*∈ConList

def[e_1 … e_n] = **def**[e_1@[e_1 … e_n] … e_n@[e_1 … e_n]]
type [e_x … e_y] @[e_1 … e_n]
= *type* [e_x@[e_1 … e_n] … e_y@[e_1 … e_n]] type≠**def**
e @[e_1 … e_n] = *e* *type*∈ConElem ∪ {**nothing**, **any**, **error**}
sel[i_1 … i_n]@[e_1 … e_n] = **sel**[@[e_1 … e_n] i_1 … i_n]
sel[e_1] = e_1
sel[@[e_1 … e_x … e_n] i_x … i_n] = **sel**[e_x … i_n]
sel[*type*[e_1 … e_n] i_x … i_n] = **sel**[e_1 i_x … i_n] *type*∈{**def**, **red**}, ∀x-1: notNothing(e_x)

apply[**repr**[e_1]] = e_1
apply[**symb**'[e_1 … e_n]] = **symb**[uniqueNumber]
apply[**phys**'[e_1 … e_n]] = hardwareDependent
apply[*type'* [e_x … e_y]] = *type*[apply[e_x] … apply[e_y]]] *type'*∈ConH ∩ {**symb**, **phys**}
apply[e_1 e_2 … e_n] = **def**[**apply**[e_1] e_2 … e_n] e_1 ∉INS
apply[e_1 e_2 … e_n] = **nothing** e_1 ∈INS, ∃x>1: not notNothing(e_x)

red[ ] = **error**
red[e_1 …] = e_1
red[e_1 … **error** … e_n] = **red**[e_1 … … e_n]

Equivalence rules for H language. Internal symbols is an implementation issue.

```
        part[ cc cd ce cf ]
        part[ cg ch ci ]]]
```

The course of events c_a c_b have been passed. They are said to be the history. The remaining part is a behavior described by the alt construct. The real-time system prunes the alternative gradually as time elapses.

The history is generally a very large data structure. Some of this data structure may be forgotten because it is of no use for the application:

```
S: seq[ some cb
       alt[
           part[ cc cd ce cf ]
           part[ cg ch ci ]]]
```

Generators

Behaviors may be very complex. Most such could not be programmed in any available computer language. Simplified, but good enough, mechanisms may be used. They are discussed.

There are parallel course of events s_1, s_2 and s_3:

```
P: par[
s1:    seq[ ca cb cc cd ce cf ]
s2     seq[ ca cb cg ch ci ]
s3     seq[ cj ck cl cm ]]
```

They may have different duration. The specification of c_x defines the time periods.

A grammar may be used to generate a behavior. It should include the basic concept, seq and par structure. The method is versatile when generating waveforms.

In an analogous way state machines may be implemented. State machines are equivalent with grammars. Conversion from one specification to the other is possible. One state in a state machine is described by a generator for that state. The state machine contains a number of generators G_n and an initial generator *IN*:

```
def [
   IN
   G1 … GN]

Gx: part[ cx
       alt[ Gy … Gz ]]
```

Grammar and state machine together with the basic method form three different forms to represent a behavior. Conversion between them is possible. All basic behaviors cannot be expressed in a grammar or a state machine.

Constructing

Assume that there exists a number of behaviors modeling proper issues. Now use three of them b_1, b_2 and b_3:

```
b1: seq[ … ]

b1

localA(b1, b2)

b2

b2: seq[ … ]

b2

localB(b2, b3)
b3: seq[ … ]
```

Here b_1 and b_2 are duplicated being 2 and 3 separate behaviors. Two groups *A* and *B* are formed. *A* contains b_1 and b_2 and a local expression using them. B contains b_2 and a local behavior b_3 and an expression using them. Group A and B communicate by the b_2 behavior. A and B can both be rewritten as separate modules. In the module concept the local references are implemented as unification with the external behaviors:

```
def[
   b1: seq[ … ]
   b2: seq[ … ]

   unify[ par[ b1 b2 ] A ]
   unify[ b2 B ]

   A: def[
         par[d1 d2]
         localA(d1, d2)]

   B: def[
         c2
         localB(c2, b3)
         b3: seq[ … ]]
```

Messages

H does not have message passing. However a parameter transfer can be written in a way rather similar to message passing. In order to describe this there are three behaviors. Earlier it was shown how such three behavior are used to connect two modules. One in each module and one between the modules. Now ignore the modules. View three corresponding elements of the behaviors:

```
b1:    seq[ … duration[ en adn Tf Tt] … ]
b2:    seq[ … pulse[ en adn Tf] pulse[… Tt] … ]
b3:    seq[ … duration[ en adn Tf Tt] … ]
```

There is a flow from b_1 to b_2 and b_3. The specification of the basic course of events has not yet been standardized. There are two constructors *duration* and *pulse*. The first duration is rewritten by some not shown function to a pulse when issued at the beginning of the period T_f. The next pulse comes at the next duration. T_t is known there. With the knowledge of two adjacent pulses the duration in b_3 could be evaluated.

There is as shown above a method going from duration to pulses. The pulses correspond somewhat to conventional messages.

A behavior uses symbols to transfer expressions. The expressions may be used in functions evaluating new expressions. These new expressions my be needed in real time. In a well-formed real behavior the evaluation is possible in real time. This means that a value to be bound in future cannot be used now!

Time and order

An absolute time has been used in course of events of earlier examples. This could be lightened. It is discussed here. For the moment assume there is only pulse course of events and ignore everything except the time T. Use the symbol Δ to indicate a pulse.

There are two parallel courses of events

par[
  seq[T_1 Δ_1 T_2 … T_3 Δ_3 …]
  seq[T_4 Δ_4 T_5 … T_6 Δ_6 …]

In this example all pulses are placed in time. There is no ambiguity. If $T_1 = T_2 = T_3$ it is possible to eliminate all by the first T_1. The same applies to the second behavior.

Now assume all times except T_1 and T_4 are slightly different. There may be a behavior that is not strict (not using) in these times. In this case just the order have importance and the times could be dropped. This could be generalized by using cascaded sequences:

par[
  seq[T_1 **seq**[Δ_1 … Δ_3]]
  seq[T_4 **seq**[Δ_4 … Δ_6]]]

With this method it is possible to encapsulate pulses in sequences that are understood by the application. There may be parallel behaviors. They may be rewritten to sequences of parallel behaviors:

seq[T_1 **par**[
    seq[Δ … Δ]
    seq[Δ … Δ]]
  T_4 **par**[
    seq[Δ … Δ]
    seq[Δ … Δ]]]

All these changes relies on a wellformed expression understanding the sequences. Each parallel sequence could be considered a single transaction.

There may be expressions having a semantics over unsynchronized behaviors. They have to inspect the time order.

EXECUTION MECHANISM

In this section a “machine” performing the execution of the H-language is described. It is a theoretical machine that has to be refined. It is based on the availability of unlimited resources. Thus word length, number of memory cells etc are of arbitrary size. In subsequent sections this model is refined.

Graph rewriting

Graf rewriting is based on an application divided into a number of expression. Expressions refer to each other by identifiers.

The execution mechanisms consists of a number of processes each being an expression or an identifier. If an expression or identifier is not fulfilling certain characteristics the process is waiting otherwise it is active. Therefore numerous processes may rewrite in parallel. Therefore ***massive parallelism*** is available being a potential for highest possible execution speed.

The software engineer expressing his solution in a language may use a “parallel” or “sequential” way. A good example on this difference is parallel versus standard sequential sort algorithms.

Another more *fundamental problem* in applicative languages based on λ-calculus is the use of tail-recursion to construct lists. Tail recursion is a very sequential implementation. Modern functional languages as Mathematica has replaced this by generators!

Closures

The rp8601 implements the expression by closures as described earlier. The general closure is a definition of a function *expr* referred to by an identifier id:

$id_n = expr_n$

The expression $expr_n$ may be self contained or use identifiers to refer other expressions. A shown earlier **def** constructs are used to form directed graphs. Each such constructs forms a typed list, the environment *env*, that is accessed by a **sel** construct.

$id_n = expr_n$ (… **sel**[ad_1…ad_m]…) @ env_n

From mathematical point of view this is a high order function application using the environment as an argument.

In order to implement a rewrite mechanism some rewrite rules needs a state local to the closure. Thus the closure is augmented by some *tags*:

$id_n = expr_n$ @ env_n, $tags_n$

Identifiers

An identifier id_n is defined in a closure and is referred to from other closures by the same identifier id_n. They all depict the expression of the referred closure. The identifier consist of a unique pattern (=number) and a state. The state is a comprehensive information of the state of reduction of the expression $expr_n$.

The graph rewrite mechanism is based on inspection of a closure with all its contents including referred identifiers and its own identifier. The state may be proper to perform a rewrite or not.

If not there may be a need to evaluate a subexpression referred by an identifier. The state of the identifier is changed indicating a request to rewrite. If that already is the case execution waits. A subexpression is pending.

After a rewrite operation the characteristics of the closure may have changed. Hence its identifier tags have to be changed.

Parallel issues

As described, the identifier is a sort of address and a signal firing graph reduction.

All identifiers id_n (having the same pattern) is an identifier domain. All have to be synchronized, but it is almost impossible in a parallel machine. The state may differ.

The state is from one first state through a number of states to a final state. There is no loop back (however an implementation has a cycle, as described in next section). The individual state of id_n is to be not more forward than id_n of the closure, and it should catch up (caused by a delay) to the proper state.

The effect of such a delay is an unnecessary delay or an additional unnecessary graph rewrite.

Rewrite mechanism

Rewriting of a graph will cause a number of small rewrite steps. Each such step signals another expression. Hence there is a sequence of closures involved. Without going into details, there is a traverse from the root of the tree down to a leaf and back again. In a branch of the tree the traverse may be to each branch in a certain order or some branches may be traversed in parallel.

The application may have the property not to *terminate* if a particular order is not the case. An example is an if-else expression. The separate branches should not be executed before the logic condition!

There are cases where termination is possible but the number of resources may be prohibitive.

The graph rewrite mechanism is controlled. An expressions is a list. Execution order is in order of the elements, if not specified differently by a rewrite rule. A list is divided into two parts. The first part must be executed in order and the second part in arbitrary order or in parallel. The elements of the first part are tagged **depth** and in the second part **breadth**. It is visible to the programmer in the H language!

ALLOCATED RESOURCES

In this section a “machine” performing the execution of the H-language is described as defined in the previous section. The “machine” is a real machine with limited number of resources, however not a machine implementation.

Memory cell

A closure is represented by some information stored in bits. This information is stored in a memory cell that is a process performing “rewrite” operations.

A memory cell m_m is **free** or has a stored closure c_n. In the latter case it is either in the state **off** performing no rewrites or being active. When active there are several states, e g **idle**, **wait**, and **exec**.

A memory cell has a physical address ad_m. A physical address is used in a very different way in a classic computer. They should not be mixed up.

Allocation

There are a number of a memory cells m_m. In this section it is assumed that the amount of cells is finite, but the number of cells may be large.

The memory cells may differ in size each storing a closure expression of different size, but greatly limited. The rewrite capacity may also differ, thus memory cells are specialized:

$$m_m = \{ad_m, c_n\}$$

or with more details

$$m_m = \{ad_m, \{state_m, \{id_n, \{expr_n, env_n, tags_n\}\}\}\}$$

Assume that there is a time t, actual or virtual time. It could be assumed to be a real value stepping to a higher value.

The basic mechanisms by *allocation* is to bind a closure c_n to a particular memory cell m_m with the physical address ad_m during a time period T_a being from t_a to t_{a+1}. During the life of the machine a particular closure may be present during the period T_{cn}. During the period it may be allocated to (stored in) different memory cells:

$$Tc_n = t_1 \ldots t_n \ldots t_a$$

… {{ad_m, c_n}, t_x, t_{x+1}} …

The memory cell m_m is *scheduled* to perform a rewrite operation at a number of events $trw_{m,r}$ during the period Trw_m.

The implementation of a *machine* is defined by an alloc structure being part of the H language:

alloc[r_n, ad_m, t_a, t_{a+1}, sch]

From a logic point of view this construct is an identity. The application state is defined by an expression defined in a representation r_n. The representation is stored in the physical address ad_m during a time period t_a to t_{a+1}. *sch* indicates how rewrite operations are scheduled.

Now assume that there is “machine” software taking as input a description of a physical machine and an application (being a representation) to execute. This machine evaluates to a set of course of events, one for each closure. The alloc structures are the events.

A soft machine

A soft machine is an interpreter for the “machine”. The semantics of the alloc construct allows physical or soft interpretation of a part of an application, not just complete trees.

A soft machine may have a side effect being a communication with a user. As such it can be used to implement various types of software as debuggers, performance meters etc.

A physical machine

A physical machine is a hardware implementation. In this section its details are ignored. Because of the physical nature of the machine the implementation of the **alloc** construct can and must be altered as shown below.

The general use of the alloc construct is to perform a sequence of rewrite operations. Formally the alloc constructs work on representation, thus it converts the representation to an expression, performs the operation, and converts the result back to a representation. These conversions between expression and representation are therefore not necessary except for in the root closure of the entire system. They could be dropped.

If not dropped they must be implemented in hardware requesting more resources and as such recursively new resources in order to implement their alloc constructs and there is no termination. Therefore the hardware MUST use the expression form and not its representation.

Physical address

If a closure c_n is stored in the memory cell with the physical address ad_m being the address of the alloc construct the address is of no use. The address is implicitly there because of the position.

When moving or copying a closure to other memory cells the physical address is needed. In such cases the alloc construct could be placed in a separate memory cell!

Thus an implementation does not need to store the physical address in every memory cell.

Time period

The alloc structure must be kept for the time before, and during the time period $t_a…t_{a+1}$. After the time period the alloc structure is made free. An implementation may therefore have only one alloc construct for a part.

In a physical implementation memory cells are made free by garbage collection. Therefore an entire tree of closures is made free when the topmost and all references into the tree are made free. Of this reason it is not necessary to store the time period in each closure.

Scheduling control

The scheduling control *sch* cannot be dropped. This information corresponds to priority mechanisms in normal processors. As such they are implementations for improving speed and response time. Therefore this concept must be implementation dependent.

In order to solve this I shall give a hint how to implement this:

Analogous to the alloc construct an additional **schedule** is used to schedule one rewrite operation. Let the scheduling mechanism (a software) create a behavior included in the scheduling information *sch* for each thread of execution.

- The scheduling information is stored in the processor as long as it executes one thread.
- When there is an “interrupt” the closure to be executed is rewritten into a scheduling construct.
- Then there is a search among the scheduling constructs to find the next one to execute.

Operations

There are the following “operations” in the machine:

store

stores a closure c_n without a rewrite. This is the general memory function.

rw
rewrites a closure c_n in a memory cell.

rpid
rewrites an identifier id_n in all memory cells.

mv
moves a closure c_n by switching the contents of one memory cell m_f to another m_t, generally being empty. The amount of used memory cells is kept. This is implemented by two successive alloc constructs, one for the current period and one for the coming period.

cp
copies a closure c_n from one memory cell m_f to another m_t being empty. The amount of used memory cells is increased by one. This is implemented by two alloc constructs having overlapping periods.

unpack
unpacks a compound expression exp_n containing one subexpression exp_s into two closures. One free memory cell is written to a new content exp_s and its place in exp_n is replaced by id_s. This is implemented by two alloc constructs having the same period.

pack
packs a closure c_n with the expression exp_n containing the identifier id_s, which is replaced by the expression exp_s of another closure c_s. This is implemented by two alloc constructs having the same period.

rm
freeing a memory cell by storing the closure expression **free**.

Identifier domain

Identifiers are special. They are the glue between the expressions. When implemented they form a sort of physical distributed process.

An identifier id_n is defined in a closure C_d. The closure C_d is said to be the domain closure of id_n and its expression domain expression of id_n. An identifier id_n is stored in a set of closures C_n with the set of physical addresses AD_n. The domain of an identifier id_n are all the physical addresses in AD_n. A domain specification is a construct containing AD_n.

Rewriting an identifier is perform in its domain.

Garbage collection

Garbage collection is performed by rewriting all used memory cells. The identifier domain is analyzed. If it is the memory cell itself the memory cell is made free.

This type of implementation would consume many rewrite operations. By adding a particular state “potential garbage” to each memory cell only those cells need to be rewritten. The state is implemented by adding a state **gc** to each identifier. In a memory cell m_{gc} containing a closure $id_{gc} = expr_{gc}$ where the identifier id_{gc} is tagged gc the memory cell is to perform garbage collection rewrite.

When a memory cell is made free all the identifiers within its expression are tagged gc, thus propagating the gc down the graph to used closures.

This method reduces the amount of garbage collection rewrites. The garbage collection is a parallel process to the application process. There are lots of other in literature reported garbage collection methods. They are generally based on freezing at least a part of the memory during the operation. After such an operation there is no garbage.

The method used in rp8601 has garbage almost all the time and the memory is never frozen. If memory becomes full and normal execution is impossible garbage collection can still be performed releasing memory cells for execution.

Physical processor

A soft implementation of the alloc construct uses variables to transport values between memory cells. A hardware implementation consist just of registers an communication paths. Of that reason some memory cells must be kept together.

All “operations” in this section are assumed to be atomic, i e unbreakable. A single level memory cell has a closure with an expression being just a list. In a compound memory cell this expression consists of at least two levels of lists.

The *rpid* operation is processed in the entire memory in an atomic operation. In the next section this restriction is released.

The *store* and *rm* operations could be performed in a single level memory cell.

In many cases *rw* consists of a one level expression. In those cases the operation may be performed in a single level memory cell. Otherwise a compound memory cell is needed.

The *mv*, *cp*, *pack* and *unpack* operations must be performed in some single level memory cells in addition to the main memory cell due to the need to transport values between memory cells.

Memory type

In a hardware implementation the memory cell may be implemented in several ways giving the storage mechanism different characteristics.

There may be memory cells being volatile, non-volatile, one time write, fixed, removable etc. By using the alloc construct these memory cells may be written. Depending on the memory there are restrictions on the allocation, e g only canonical values may be stored in one time write memories.

Removable memory

Memory cells may be placed in different physical memories. These memories may be changes in the computer structure or being add on memories such as USB-memories.

The structure of a machine including the add on memories is controlled by some software creating a behavior being alloc-constructs. In order to implement a removable device a sort of interface is needed. A machine could be modeled by two different expression trees L and E, being the content of local and external memories, respectively:

```
def[
2   par[
3       alloc[L(ei), adlocal, …],
4       interface(ei, emem),
5       alloc[E=emem, adext, …]],
6   ei, emem]
```

The program contains a definition of three modules, one on each row 3 to 5 and two symbols on line 6.

On line 3 the main program/data structure L is allocated to the local address ad_{local}. The construct $L(e_i)$ is not proper but indicates that *L* contains a reference to the symbol e_i.

Line 4 contains a module here also sketched to refer to the symbols e_i and e_{mem}.

On line 5 the external memory is allocated containing the external data structure *E* depicted by the symbol e_{mem}.

The symbols are one time written by unification internal to either *L* or *interface*. Both *L* and *E* are assumed to be of representation form. In a real implementation line 2 to 5 is implemented as a behavior causing different binding at different periods.

During a first period *E* may be empty and in a subsequent period *E* contains an expression. Then the interface controls the memory to be disconnected. In a non volatile storage *E* remains. At a later period the interface connects the memory and makes *E* available.

Because of the lazy mechanism *E* is assumed not to be used during the time the external memory is disconnected. However, if so the reference e_{mem} has no definition (i e closure) and is reduced to **error**. In a real example there may be many references into *E*.

If *E* contains “pure data” the representation *E* describes a canonical structure. *E* may also be parts of a computation, i e a state in the middle of a computation. In such cases *E* represents a non canonical structure. *E* may be moved to another computer and parts may be rewritten there. Subsequently it may returned.

This implies that a subexpression of any type may be sent to another computer and processed there. All under the control of the H language!

PARTITIONED RESOURCES

In this section the memory cells described in the last section are partitioned into a number of memories. Here called clusters. The memories are separate processes performing communication and graph rewriting. The aim is to focus on implementable structures.

A cluster

A cluster is a number of memory cells. It may contain zero, one or several compound memory cells. Some additional special memory cells, network cells, are added.

The design goal of a cluster is:

- a small graph may be stored as a whole inside a cluster
- all machine operations discussed in the preceding section could be rewritten locally inside the cluster.
- some clusters are special and performs port semantics and have external physical connections.

All operations except *rpid* and *mv* are always performed locally. **rpid** is performed locally or globally. **mv** is a global operation.

mv operation

The mv operation is performed when closure c_{mv} contains an **alloc** construct. It contains a physical address ad_{mv}, that is the target of the move.

The move operation acts as the closure is a token randomly walking between clusters until it reaches its target address ad_{mv}. In an overloaded system it may not reach its final position! This is assumed to be a design error of the application.

Global graph rewriting

The clusters could be modeled as separate processes communicating according to some protocol performing atomic rewrite operations. **mv** is a special case where the rewrite operation has the side effect of moving a closure.

The global graph rewrite is constrained to the ones possible to describe in ONE closure:

id_n
$id_n@env$
: this is an identity rewrite of identifier id_n. It is a signal that a reduction is needed. In some cases the environment is needed. Actually it is ***net****[id_n ad_r]*, see next section.

$id_n = constant_{red}$
$id_n = id_{red}$
$id_n@env_n = constant_{red}$
$id_n@env_n = id_{red}$
: in these four rewrites an identifier id_n is rewritten to another scalar. A **rpid** operation is implemented by this rewrite.

$id_n = exp_{red}$
: an identifier id_n is rewritten to an expression exp_{red} including tags.

A global **cp** “operation” is performed by the first rewrite signaling the source of id_n and after a delay the last rewrite returns the expression exp_{red}.

The communication protocol is intentionally designed as a rewrite operation. Sending global rewrites more then ones does not harm. It may cause some additional computation. If a system does not have atomic operations there may be a race causing a rewrite to be late. This does not harm because the applicative language will always rewrite to the same result even if the step in a rewrite chain is stepped backwards!

An expression is always reduced from **free** towards a final canonical result. However there is an additional step back to **free** when garbage collecting. This case must be handled with care!

Parallel global rewrite

An identifier id_n has a domain AD_n. A rewrite rule is always from an identifier to some expression. Thus only the memory cells m_n of the domain AD_n are involved.

A typical application contains a hugh number of domains. All these domains may be rewritten in parallel. Beware of the garbage collection rewrite.

Implementation

All clusters have at least one special memory cell, the *network cell*. It is assumed only to be used temporarily. A global rewrite operation issues and receives rewrite rules in this memory cell. The operation is atomic:

- empty
- receiving a closure from the identifier domain and performing a rewrite operation in the local cluster
- from the local cluster a rewrite rule is received and sent to the identifier domain.

Identifier domains

As shown above all rewrite operations, local as well as global, may be performed in parallel as long as the identifier domain is know.

An identifier id_n has a domain as described earlier. Closures may be copied, moved and rewritten. The domain is therefore a dynamic concept. There may be many physical addresses in a domain and the size of the domain specification may be considerable and prohibitive.

A domain specification may contain more addresses than actually needed. It is an *oversized domain*. The only drawback is that the reach of the global rewrite operations may be larger than necessary causing to many local executions for a global rewrite operation. However the size of the domain specification may be considerably reduced.

The domain specification may contain standard sets all being supersets of the identifier domain. One trivial and very useful such specification is the local cluster! Other ones are supersets of clusters as shown in the next section.

DOMAIN AND TRACE GRAPH

A trace graph is a model graph showing the reduction history in a graph reduction machine. The graph is not a structure used by the machine - instead it is all expressions unfold which are accessed and rewritten.

A **trace** structure is a closure c_{tr} read before a graph rewrite. Each such trace may also contain the physical address ad_{tr} where the closure is stored and the time period T_{tr} it was stored:

trace[c_{tr}, ad_{tr}, T_{tr}]

The closure contains an expression referring to other trace structures. The complete graph is a trace graph *TR*. It contains all details necessary to understand the course of events during the execution.

The trace graph could be use to show the result of an execution as well as being a tool for planning an execution.

Applications

The **apply** construct is used to create an instance of a representation. It is the tool to unfold expressions. Each such operation creates an augmented copy. This copying mechanism is the major mechanism building the trace graph.

Correspondence to classic computing

The copying mechanism corresponds to instruction fetch in classic computers. The **rw** of expressions corresponds to arithmetic instruction execution. The **rpid** corresponds to store instruction execution.

Performance

There are cluster operations as **rw**, **pack**, **unpack**, **cp**, **rm**, and network operations **mv** and **rpid**. In a trace graph the the links corresponds to network operations. Trace constructs corresponds to cluster operations.

An application *P* is the “program” together with all its arguments to execute. The execution causes the trace graph TR_P. It has the following characteristics:

- the number of links is the *volume* of network operations
- the number of trace constructs is the *volume* of cluster operations
- the *depth* is the maximum number of trace constructs cascaded
- the *breadth* equals the volume of network operations divided by the dept.
- there is a statistics being the *number of references* in identifier domains.

Sharing of expressions causes the difference between the volume of network and cluster operations. Generally they are in the same range.

Characteristics of cluster operations

Actual hardware has execution time for each individual type of cluster operation. They are here assumed to be equal.

Characteristics of network operations

The network performance depends on the number of transports and its pattern. Each network operation accesses the domain of its identifier. The transport load depends on the size of the domain.

In order to understand this, a space is used. Essentially it should model the communication. However a 2- dimensional (area) is used here. All physical addresses should be placed in this area. All domains should be drawn. From this graph it is possible to understand different applications. Some are easy to layout with small domains other not. Assume that there exists a near optimum layout. The *total size of the domains* is a characteristic of the complexity of the application *P*.

Copies

Some closures are referenced from many places. This may create large domains. By using copies it becomes a two level structure. The topmost domain has a large size, the domain of each copy is smaller. The use of copies may considerably reduce the *total size of the domains.*

Copies are in memory cells, if dynamically allocated their size may be reduced.

Tuning performance

Dynamic allocation of closures and copies is the method to tune performance. Each application has its own characteristics. There are no general implementation method.

With the **alloc** construct dynamic allocation may be performed.

Constraints

During the execution some clusters may run out of memory. Nothing is lost but execution is stopped. There is a need of some forced reallocation.

Such reallocations has not yet been studied.

Example 1: Large code

I will use the word code in the following few sections for function, modules, data etc being the part that is called the same in classic processors. Classic computing is assumed to execute in a processor (e g x86) with a cash.

A behavior has the characteristics: the code size is large, the part of the code accessed is small, each new instant behavior uses an almost random part of the code. Code is not used several times.

The code is allocated evenly to many (all) clusters. Whenever there is a branch the behavior is allocated to the cluster containing the branch. Identifiers domains become small. Load is spread out among clusters. Increasing the frequency of executing the behavior increases the load on all clusters.

In classic computing the cash do not contain the code. Thus the code part is spooling through the cash. The memory bandwidth limits execution speed.

Example 2: Uneven load

The same type of code as in example 1 is used, but the different parts have different load *L*.

Copies are use of different parts. The number of copies is proportional to *L*, where the smallest amount must be 1. Depending on the total load some code may have a substantial amount of copies. All clusters have approximately the same load.

A behavior must branch to different copies almost in a random way. Some hash technique may be used to select what copy to use.

Classic computing has the same application as in the example 1.

Example 3: Filter

A behavior uses almost all code and it is used may times. Typical programs are filters.

Copies of code is allocated to many (all) clusters. A part of the behavior fits into one cluster. The behavior must be distributed to the clusters based on some key. The clusters all have approximately the same load.

The cash works fine in classic computing.

Towards automatic allocation

Finding the proper allocation is not the scope of this report. However I will suggest one method. Assume that a system could be executed with gradually increasing load.

Beginning with a low load the allocation may be tuned. Loads of various parts of the code is measured. Copies are created in proportion to the load. Parts are scattered almost randomly over the clusters. The scattering distance is proportional to identifier domain sizes.

The optimization is performed by simulated annealing. The mean re-scattering distance follows the temperature.

This optimization is performed when gradually increasing the load.

Identifier domains

Identifier domains are the key to get performance. The domain size is given by the application. The implementation of the identifier domain may give considerable differences.

Domain specification

In a way analogous to the alloc construct the physical address is given by:

net[id_d ad_r]

A memory cell m_d contains a closure $id_d = exp_d$. It is referred by a net construct. Zero or one memory cell m_r refer to the net *cons*truct. The physical address of m_r is ad_r. From H language point of view the net construct is an identity.

If there is no reference to the net construct it may be garbage collected. From garbage collection point of view there is at most ONE reference.

To build a complete domain specification there is one net construct for each reference. The net constructs refer to one and the same closure.

The alloc construct is designed to facilitate software control of the allocation. In the same way the net construct may be used. There may be a software being a communication protocol between several sites implementing the network operations.

Oversized domain specification

Of the same reason as alloc construct could not be implemented in each memory cell, the net construct could not. Oversized domains are used in hardware. It contains more memory cells than necessary.

Clusters are arranged into a space. The space is divided into several clusters and this is repeated recursively until there is a cluster of proper minimal size.

There may be borders between clusters being the same for all levels of the partitioning. A domain lying over such a border could only be contained in the largest cluster. To solve this, the partitioning schema may also use a skew moving a cluster approximately half of its size in some direction.

One memory cell m_n contains an identifier id_n. An oversized domain is a set of memory cells *M*. There is a specification being a number used by all memory cells m_n of M depicting the same M.

An example: Memory cells are placed in a square grid. There is a minimal such unit. The side doubles for each level. 2 bits are used for skew in x and y-directions, respectively. The other bits define the side length to 2^s. 5 bits specifies the size in the range 1…65536 and skews.

Domain size used

An identifier domain contains a number of physical addresses referring one memory cell. An identifier domain is said to be *strict* if all referring addresses use (perform access) the referred memory cell.

This could be the case in lazy programming languages.

During graph reduction a reference id_n requests the domain closure at time t_n. Some reference may never request. The times t_n are in the set TR_n.

The domain closure may be ready at any time, before, during TR_n or never. There is and should only be response when the domain closure is ready. If after TR_n it is possible to make a global rewrite to the entire domain, otherwise it may be subsets.

If resource cost don't differ complete domains and subsets may take part in the global rewrite operation. Otherwise the cost may be high. If there is no other means the entire oversized domain must take part.

To eliminate this high cost a combination of the actual and the oversized domain specifications are used:

- The actual domain specification is used as default. Temporary net constructs are added. There are no references to them. Hence they will be garbage collected or removed when necessary.
- When there is a request from a reference id_n to a closure c_n, a **net** construct is placed near the memory cell m_n containing c_n.
- The normal graph reduction continues. There may be several accesses to c_n creating several net constructs.
- When c_n is ready **rpid** uses the net constructs by rewriting to them and not the entire domain.
- The net construct is subsequently performing the global rewrite in the specific physical address ad_r. Then the net construct is removed.

Because it is a combination of two methods there must be some indication which one to use. Garbage collection uses the default one. **rpid** uses the **net** constructs if nothing special have happened as run out of memory.

This method changes the normal access to point-to-point accesses. It is a dramatic reduction in network resources.

REDUCTION PROCESSOR

An implementation of a cluster in an integrated circuit is described in this section. In the project the various parts of such a circuit was studied. No complete unit was designed. However several test implementation of parts were. This section describes another circuit based on the experience from these designs.[2]

A number of variables are used here to describe sizes and other physical characteristics. After such a variable a parentheses may be seen containing the value used in the project. At the end of this section there is a table specifying all these values.

The cluster implemented as an integrated circuit is here depicted reduction processor. It has n_{net} (4) number of network ports. The processor contains one (1) memory cell containing a two level expression, n_{net} (4) network cells, and n_{mem} (512) memory cells containing a single level expression, and one (0) time cell.

The basic list contains n_{elem} (4) elements. The numeric part uses a word-length of w_{word} (32) bits.

The state information of the identifiers has a width of $w_{IDstate}$ (6) bits. Hence the total word length to store is w_{mem} (38) bits. The list type has a word length of w_{type} (5) bits.

The physical address was fixed for the memory cells. The lower part in the physical address selected word within a reduction processor. More significant bits selected reduction processor. Identifiers used the physical address as pattern.

The reduction processor is a complete unit needing no other supporting circuitry except for the communication network.

A port is implemented analogous to the reduction processor. It may in some other variants of the reduction processor be integrated in the reduction processor.

Unit

The reduction processor is implemented as a unit that has the following ports:

network ports
: There n_{net} (4) network ports. They are bidirectional high speed serial ports.

The reduction processor is built based on a structure of numerous list elements. There are also some few *list type* elements being short. In this implementation an additional element of full size is used for the *type*. It may be the case that this element could share the environment *env* element. In the next section this is the case.

A data-path is used with the following abutted units using a data-path width w_{mem} (38):

network unit
: is a communication device performing transmission of a closure through network pins. It consists of a shift register. The unit contains an additional latch and a bidirectional data-path of w_{tr} (4) bits connected to a *network transceiver unit*. It controls the latch and data-path. Reading an writing the registers is controlled by the *reduction rule unit*.

core unit
: is a special unit containing one closure with a two level expression. It has local data-paths making permutation of various expression elements possible. It is controlled by the *reduction rule unit*.

---

[2] The project used the same philopsopy for the design of the reduction processor. In this report a processor is based on sequential arithmetic of individual elements in contrary to a full closure. The full closure approach had the problem that words become very long. It was possible to cope with this, but between numeric arithmetic and the remaining part there was a swap of wire order from bit plane order to closure order consisting of several hundred wires.

time unit
: is a unit containing absolute time. Essentially it is a counter that is stepped by time and two accuracy elements. Stepping is controlled by the *clock unit*. The remaining part is controlled by the *reduction rule unit*.

All these units communicates through a switching network implementing communication between the units over data-paths with the width w_{mem}.

There is a memory abutting to the data-path above. It contains a local bus connecting n_{banks} (1) of memories. There are also some few simple circuits calculating the total result from all memory banks. The major unit of the memory and reduction processor is:

memory unit
: is a bank of memory cells. It has the width of an element w_{mem} (38). It contains a number of words partitioned into groups for one closure $3+n_{expr}$ (6). The group contains the identifier *id*, the environment *env*, and a n_{elem} *(4)* number of elements of the expression *expr* and the expression *type*. The type has the word length w_{type} (5). It is not necessary to implement it as a separate word but may be integrated in the environment *env* (not researched). A bank contains n_{clo} (512) closures.

There are some additional cells being peripheral to the the data-paths. Essentially they do not have any regular structure:

clock unit
: is a unit being a phase locked oscillator. The output speed could be set. The phase locking mechanism reads the network ports through the *network transceiver units*. It also steps the real time clock in the *time unit*.

network transceiver unit
: is a bidirectional transceiver for a network port. On the external side bit serial signals are used. On the internal side words w_{tr} (4) bits are used. Single bit could not be used because of speed and power. One transceiver controls one *network unit*.

routing unit
: is a unit performing combinatoric domain arithmetic on identifiers. It also performs routing arithmetic used to choose communication port. It contains latches for parts of the identifier. It is controlled by the *reduction rule unit.*

arithmetic unit
: is a unit containing three registers being of the size of one element w_{mem}. During arithmetic they contain the three first elements of an expression, where the first one contains the function identifier and the remaining the operands. They are copies of the corresponding words in the core cell. Combinatoric circuits perform the built in arithmetic. It is controlled by the *reduction rule unit.*

The control of the data-path is by *clock unit, network transceiver unit*(s) all having minor control but being asynchronous. Essentially they are synchronised to the data-path. More about this in the next section. The main control under this is by:

reduction rule unit
: is a unit containing the tags of the core cell. It performs the combinatoric arithmetic to create the control of graph rewriting used by the core cell, net cells and time unit. Its implementation is not combinatorisc.

Timing and control

The clock unit contains a high frequency oscillator. It generates two output clocks being phased. The high frequency f_{com} is used by the network transceivers, and the low frequency f_{data} the remaining part of the processor.

The logic cycle implements one logic operation as graph rewrite and phase locked oscillator control. A logic cycle consists of several clock cycles. Each logic operation has a preplanned use of the following cycles. The use of the cycles depends on the register contents.

The data-path performs preplanned register transfers through a switching network. Generally full closures are transferred. The memory uses several phases to perform one operation. The reduction rule unit controls the use of the cycles.

Arithmetic unit

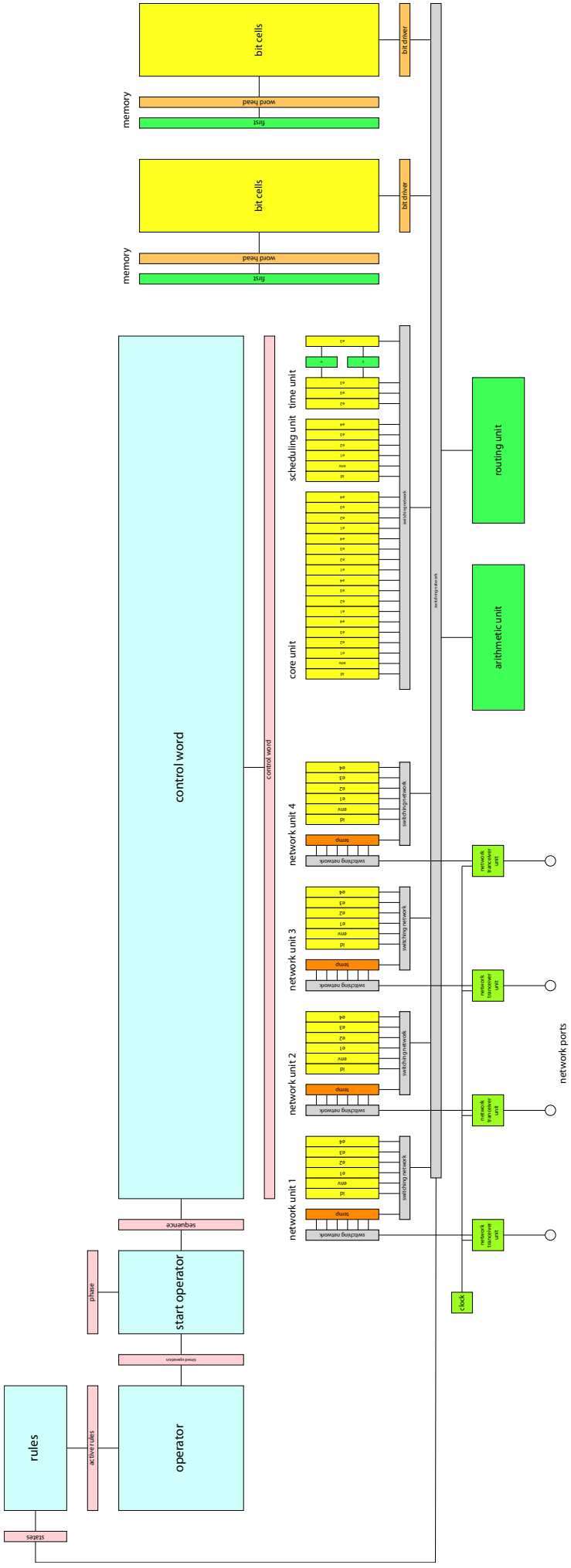
The arithmetic unit contains 3 registers being duplicated elements of the core expression. The first register contains a hardware address being the operation code and the two others are arguments. Thus monadic and dyadic operations may be performed. The arithmetic unit also contains an additional state if the built in operations are not combinatoric.

The implementation of the sub-operations are combinatoric and could be assumed to be a heap of gates. However they should internally be designed with care.

Optimization

For the rp8601 architecture it is important to optimize the size. A numeric application is distributed to all reduction processors. An implementation may use different amount of reduction processors. The total sum of the areas of the reduction processors should be minimized. The major part of the area is the memory unit. The core and network units can

Block diagram of the reduction processor



be made small. The arithmetic unit may take a considerable size as reported in literature. See also next section.

Simple integer arithmetic like add, compare etc has a negligible size. Floating point add may be of small or moderate size. Integer or floating point multiply may be rather small or take a considerable size.

There is a trade-off between size of arithmetic unit and the number of reduction processors. The optimum is not known. It is certainly application dependent.

The project implementation

The project used a simple adder with two dual directed shifting networks of 4 and 1 bits. It implements almost all operations in one cycle. Integer multiply use a 17 and divide a 33 cycle algorithm.

Floating point is implemented with separate alignment, normalization and numeric operations. Alignment was implemented in maximum 10 fast cycles corresponding to a mean value of 2.2 cycles. Normalization is performed in one cycle. Add and multiply parts are performed in 1 and 13 cycles, respectively. This results in a floating point add of 4.2 cycles and a multiply of 15 cycles. The normal mix in filter applications is even, thus the mean floating point operation takes about 9 cycles.

The message is: for floating point a very simple arithmetic unit is at most 9 times worse than a large parallel implementation. Including surrounding glue operations it is less.

Time and clock units

Time T is represented as a **cycfix**. This value is stored in a register in the time unit. It is stepping as time elapses. The time T should be the same in all reduction processors. In order to achieve this the frequency of the local oscillators must be the same and then T have to be fixed to show the same value.

VCO

The clock unit consists of a voltage controlled oscillator VCO and a control value c_{vco}. The oscillator is to be controlled to be phase locked to all corresponding oscillators in a system of reduction processors. Internally it may not have the same frequency but a frequency unit 2^n Hz could be derived. Such units are phase locked.

The communication speed on the network ports should be the same or a multiple of 2. Thus the transitions should occur in known places. The clock units use these to control the c_{vco} of the *VCO*. If the clock has the highest accuracy in the system it is not controlled. Otherwise the mean of the adjustment from the most accurate network ports are used.

Phase locking

Each reduction processor has a time construct being:

time [T, acc_{nom}, acc_{ach}]

T is the time and the other two accuracies being reals. The acc_{nom} is the accuracy stamped on the crystal or the corresponding. The acc_{ach} is the accuracy achieved by the phase locking algorithms. The time construct is included in the H language.

The phase locking algorithm sets the system frequency to the most accurate clock. If there are several such clocks the approximate mean frequency is used. When synchronized, time should be set to the latest time of all clocks. The reason to this is that time must always go forward!

Reduction processors emit the time construct into the network. Recipients read it and adjust the time and VCO accordingly.

When a VCO is adjusted the transitions on the network ports will keep the VCO synchronized. If the traffic is low additional time constructs are to be emitted in order to keep accuracy. It is assumed that the traffic load due to time constructs are negligible.

Initially when a clock starts the achieved accuracy acc_{ach} is 0 and the time constructs are emitted with this value. The maximum value of the local and the received acc_{ach} sets the local one. Some filtering is used. In this way the acc_{ach} of the most accurate clock is calculated. When adding a new clock in the system the same takes place, but there is already a time T set. The acc_{ach} tells this. The local time is set to the received value.

When no time constructs are received and the acc_{ach} is higher than acc_{nom} the acc_{ach} is reduced by time. Eventually it reaches the nominal accuracy acc_{nom}.

Network synchronizing

A network consists of nodes being communicating hardware units. Each such unit is assumed to have a clock. Communication is performed point-to-point. Thus nearby clocks synchronize.

If there is one most accurate clock the achieved acc_{ach} accuracy is distributed from this clock and outwards in the network causing acc_{ach} to propagate. If there are several such clocks their acc_{ach} is distributed in the same way.

Memory unit

There are a number n_{banks} of parallel memory banks, each being a memory unit[3]. Some simple selection circuitry selects the proper memory bank.

Memory operations

The memory has the basic mechanism as storage. It implements the basic operations *store*, *rpid*, *mv*, *cp* and *rm*. For the mv and cp operations the core unit supplies a closure as argument.

The memory is associative.

An associative memory CAM is different from a normal random access memory RAM. The RAM has a fixed built in address but the CAM searches for particular patterns and sets non, one or several mark bits of each word. They could subsequently be used in some operation over all words and the mark bits are subsequently set. The mark bit is used to select a word to be read or written. Operations do generally need to know whether at least one mark bit is set.

In a conventional RAM there is a fixed address for each word. The access could be divided in an address decode (= search) that sets a dynamic mark bit that drives the memory word to read or write. The CAM has no such address (-decoder) but uses the stored contents.

Within a closure there is an additional state being an OR between mark bits.

A memory could be design with all words equal. If so an addition state *first word* is added to each memory word.

The rp8601 memory uses the following sub-operations:

idle
: the memory stores something. The memory could be set in a low power electric state.

init
init first word
: sets the contents to an initial value. Depending on the physical design it could be performed in several ways. One operation or one for each word of the memory may be needed. This operation is only performed at power up.

find *free, exec*
: searches the memory for a closure being **free** or **exec**, respectively. The mark bit is set accordingly.

find Id id_{arg}
ind env id_{arg}
find Ids id_{arg}
: searches the memory for a closure where the identifier, environment or each element, respectively, *pattern* equals id_{arg} and sets the mark bit accordingly.

first word
and
: OR between the mark bits in the preceding operation was evaluated. If true the mark bit of the first word is set, and if not the mark bits are reset, respectively.

first
: performs an operation on all mark bits. Words are organized in a fixed order defined by the memory structure. The mark bit of the first word having a mark bit set is set, the remaining are cleared.

read
: performs a read of the word having a mark bit set. One and only one word is read. The mark bit is shifted to the next word (neighbor) of the closure. A sequence of read operations is used to read a complete closure.

write
: performs the write of zero, one or many words in a single step. Several writes are used analogous with the read operation.

There is an output indicating that at least one mark bit is set.

Memory structure

The memory is a 2-dimensional device. Vertically there are words, and horizontally bits. The main parts of the memory are:

bit driver unit
: is used for reading and driving one bit-line pair of the memory. There is a port used by the core cell for the transport of a value. The unit performs the operations **idle**, **write** *0/1*, **search** *-/0/1* and **read** *0/1*. The search could be ignored (-) in order to mask out bits. Depending on memory operation the individually bits are controlled.

word head unit
: is used to drive the word-line of a memory word or read the compare result. There is one register bit *mark*. It could be dynamic. The circuitry performs the operations **idle**, **cmp** *bits*, **first** and **access**. *cmp* sets mark according to a wired-and on the word-line. *first* reads from

3 The project implemented a memory where the closure was one complete word with several heads. Thus all read and write was to all bits of the closure. There was one memory bank.

the first unit. *access* drives the word-line and shifts the mark bit to next word.

first unit
reads the mark bits and calculates a new mark bit indicating the first mark bit in the memory. It also performs the wired-or between all mark bits of the memory.

memory cell unit
is used to store a one bit value. It is controlled by the bit-line pair and the word-line. The combinations possible are: **idle**, **write** 0/1, **read** 0/1, **equal** -/0/1 T_{wl}/F_{wl}, i e 10 variants. A classic RAM cell has 5 variants.

Reduction rule unit

The clock unit creates clock signals as the main control of a reduction processor. Using these the reduction unit unit controls the reduction processor. It contains bits from several registers and a local state of control. This state is used to plan all cycles of a logic operation. In order to simplify wiring the contents of some few registers in the core cell is duplicated and placed in the reduction rule unit. They are copied when transferred to the core cell.

During the initial part of a logic cycle a cell is chosen to be placed in the core cell. Asynchronous units as the network cells and time cell have stored some few bits being requests for operation. The reduction unit selects one of the cells to be rewritten and transfers its contents to the core cell during some few cycle. Rewrites occur. After the completion the result is restored if necessary. Going back and forth between core cell and net cell/memory waists cycles. By good planning this could possibly be eliminated in some grade. However for simplicity its described in this way.

The reduction unit reads the state and performs an "associative" look up for a sequence of control signals.[4]

The control unit is based on:

phase
This is the steps in the reduction processor. It contains a list of bits with a walking "1" indicating the the phase. There are n_{phases} phases. They are stored in a shift register. Generally there are only rather few phases needed. The maximum amount is not known. It is in the range of 90.

state
This is a register strobing the data-path of the core cell. During transfers the content is built up. It contains the *type* of lists, the *identifier states* of the first level expression and some return signals from them memory indicating "found" etc. It has some additional hand shaking bits from the asynchronous units. The width w_{state} is unknown but is expected to be around 80 bits.

rule
There are n_{rules} of rules *rule* being a logic AND condition of the state. They all produce a boolean result. n_{rules} is not known but is expected to be in the range 200…500.

operator
An operator is a tuple: start phase t_{op} and sequence of control words op_{seq}. The t_{op} is used to "fire" a sequence the corresponding phase. There is a logic OR between rules selecting several operators. There are n_{ops} cases. The number is not known but is expected to be around 150.

sequence
A sequence is a serial of phases with control signals. Each sequence op_{seq} consists of one control word c_w for each phase. A sequence is started and last for op_{length} phases. Each sequence has its own length op_{length}. Summing all op_{length} gives the total number of control words $n_{seq.}$ There may be several sequences running simultaneously. There is one shift register for each sequence having a length corresponding of op_{length} bits.

control word
A control word is one bit for each control signal in the data-path and other units. The control word has w_{cw} control signals. It is not known but is slightly more than 254. The number of control words n_{seq} is not known. A rough estimate is 480.

As shown there are the tables: *rule*, operator $[t_{op,}$ $op_{seq}]$, start operator t_{op} and control word c_w. Their values are "programmed" into the hardware.

This could be abstacted by using the H language in:

```
apply[
   ORbag
   bag[
      alt[ if[
               during[top, seq[ cw ]]
               unify[rule, state]]
            … ]]
```

[4] In the project the data-path was a full closure and the operation on the entire closure was in one step. A reduction needed some few steps. In this report the execution is a course of events where individual elements are executed. The project reduction unit was designed the same way however not using a sequential control. There was a compiler from reduction rules to control signals.

Where ORbag performs or between all control signals of the bag.

Parameters

parameter	project		this	
w_{word}	32		32	
$w_{IDstate}$	6		6	
w_{mem}	38		38	
w_{type}	5		5	
n_{elem}	4		4	
w_{tr}	4		8	
n_{net}	4		4	
n_{banks}	1		8	
n_{clo}	512			
n_{phases}			90	
w_{state}			80	
n_{rules}			500	
n_{ops}			150	
n_{seq}			480	
w_{cw}			254	
f_{com}	536	MHz	10	GHz
f_{data}	25	MHz	5	GHz

CIRCUIT

In this section there is a circuit implementation of the reduction processor described in the preceding paragraph. In the project the various parts of such a circuit was studied. No complete component was designed. However several test circuits was manufactured.

The technologies at time of the project is far from what is present. However the circuits has not changed that much. In this chapter the circuits are described. The performance is from a virtual 40nm technology[5]. Contemporary processor technology is 18nm. All figures are rough estimates.

Circuit basics

The technology aimed at is a CMOS technology of contemporary performance. No special feature is requested for.

Wired and/or

Much of the processor is regular array like components. They are used in associative memories, registers and read-only memory structures. They all have some type of logic AND or OR on some wires. In CMOS such structures are dynamic: first set to 1 (and)/0 (or) and then a transistor implements the logic by reversing the signal. A 1 (and)/0 (or) case is represented by a non reversed signal. Its value is maintained by the charge of a wire capacitance.

From test point of view circuits are to be tested at low speed. The solution to this is to have a small current into the node. Here this is accomplished by a transistor in weak inversion.

Multiplexer

In structure arithmetic the general operations are selections of complete word values from different sources. Each bit may be implemented either as a logic equation or as a transmission. From logic point of view they are identical. In this processor almost everything is structure arithmetic.

A transmission may be performed in one or two directions. In the latter case an extreme complicated logic results. It is comparable to functional versus logic language using symbols. Logic implementation result in many wires and addition gates. Because of the distribution of source values in a data-path the logic method also adds many wires in the one direction case. This increases power and area.

Some technologies are based on the use of such functional approach[6]. In this section pass transistor methods are used as in transmission gates.

Testing

Modern testing methods are based on testing storage and combinatoric circuit parts separately. The combinatoric circuits are tested by stimulating them by introducing test values in registers and reading results in other registers. Storage values are read out and read in by shifting values through the registers acting as shift registers. The performance of such registers are slightly changed.

The storage has to be divided into two parts of storage: registers and memories. Some small register structures have to be tested as memories.

Registers are tested by sending a value through the registers as being one shift register. Memories do not have the additional costly logic for testing. Instead they have to be tested by stimulating them from a register. Clever organization of memories and registers make this feasible.

There may be a need to connect registers to a data-path in order to stimulate/analyze it. The core unit in this reduction processor must have such an

---

5 The figures about performance is taken from a technology approximately equal to TSMC 40nm technology (Europractice). It uses 1.1V power, has a ring oscillator delay of 5ps, a gate oxide of 25nm. Typical saturatioin currents are 600A/m and 300A/m per gate width for N and P transistors, respectively. Static memory cells have size 0.25μm^2 and gate density 2 gatesμm^2.

6 It was tested with a very negative result.

additional register. It should read or write through the main switching network.

Registers have to be special. In one mode they act as normal registers and in the other “test” mode they act as test shift registers.

Power consumption

Power in circuits are caused by charging wire capacitances. In order to reduce power, the main structure of the processor must be influenced and not just on lowest level.

Registers may use clocks. A clock is always (beware of controlling the clock) switching. Depending on the circuit design there is a considerable amount of power, even if the logic state is stable. The main cause of this is the control of logic or pass transistors. This causes the idle power consumption.

Using few clock controlled registers reduces this power consumption. Registers circuits are changed to have set or store modes (e g latches). Without clever design there may be racing problems.

In this processors only main control signals are implemented in clocked registers. There are about some hundreds of them. They will cause the idle power consumption when the clock is running.

Data-paths in this machine transfer values from one register to another. There are no cycle formed. Therefore simple control wires are used to control the storage elements. Control signals have to be stable and have no glitches. It is accomplished by storing the control signals in clocked registers.

Data-paths generally have many transistors connected. In order to reduce its power the wires are divided into hierarchical set of wires. Only that part used is switched. This technique is used in the network components and the core component.

Control wires are generally loaded by at least one transistor of each bit-plane in a data-path. The capacitance is considerable. Driving such wires by logic gates may cause glitches causing addition switches. In this processor all control signals are latched an have stable signals.

A data-path have over hundred control signals. The number of simultaneously switched control signals (not total number) have to be minimized. There is in some gates a trade-off between power consumption in data-path wires and control wires. In this processor this trade-off has been used.

Arrays like memories do have many transistors connected to each horizontal and vertical wire. The switching of or/and arithmetic causes generally all wires except on in the or-plane to be switched. Its hard to change this.

In some cases the OR characteristic is known. The or-plane could be implemented with set and reset transistors. Such an implementation needs two “address” wires. In such a memory used word wires and 2 address wires switches. It is a hugh reduction in power consumption. This techniques is used in the implementation of sequences.

Switching speed

Propagation delays are caused by serial connections of transistors and parallel load of wire and transistor capacitances.

The number of serial transistors is caused by the logic. Partitioning a logic as hierarchical structures adds one pass transistor for each level in an approximately linear way. On the contrary the capacitive load may shrink exponentially. There is a trade-off. Buses are divided in at least a two level structure in this processor.

High-speed states could be coded in a dense form like a number. Decoders are gate structure of several levels. They generally have gates with several inputs where the delay is quadratic with the number of inputs. In this processor one register is used for each state. The state is available in shortest delay after the clock signal.

Array structures perform set and reset of word wires. The reset is the logical function. Generally there is some few transistors, including the programmed one, in serial. The word wire capacitive load is mostly loaded by connected transistors. The only way to reduce this delay is either improve the contents pattern or divide the memory in separate structures. None is used in this processor.

The array “address” wires are generally decoded. Such a decoder adds delay as a gate.

Buffering is either a scaling of transistors or adding cascaded scaled inverters. Some transistor scaling may enhance performance. In this processor addition of inverters are used, especially on control wires and long data-path wires.

Data-flow sectioning is a method to separate logic components. By adding registers between the logic, the step cycle could be reduced, however the number of steps are increased, and also latency time. Adding additional registers causes additional power consumption by the registers and their clock. This sectioning is used in the network units in this processor.

Data-flow sectioning of wires is a method to separate one wire into sections. In this way the large capacitance load may be divided. Between sections are registers added. Because the total capacitance is slightly increased the power consumption increases, but on the contrary the propagation de-

lay for each section is divided. Hence the throughput is increase but not the latency time. This method is used for the wires between net units, memories and the core unit in this processor.

Components

There are two major components, the latch and the master slave latch.They use two phases. They have 8 and 16 transistors. Almost all transistors switch during a cycle. If high resistivity poly with low capacitance is available the master slave latch uses 8 transistors and 2 resistors. The power consumption is halved and propagation delay time is shortened.

Multiplexers are complementary pairs of pass transistors.

Data-path

The data-path is one component. It implements n_{net} net units, one core unit, one time unit and one scheduling unit. All except the last one has been define in the preceding section. The last one is something needed but not known. It is at least one register holding an interrupt time and an equal comparator between time and interrupt time.

All these units and the memory are connected to a bidirectional switching network. It is partitioned into two levels an divided into three sections.

The net units is part of one section. Each net unit has a local switching network. All units are normal registers except an additional buffer latch. It contains a mast slave latch and connections to a switching network connected to the transceiver.

The memories are on its own section of the switching network.

All registers of the core unit, time unit and scheduling units are forming one register bank. The time register contains a master slave register with an additional input from a +1 arithmetic. The remaining registers are normal registers. The registers are connected to a two level switching network.

The time unit has one adder stepping with +1 and a comparator. Stepping is slow and the adder is a normal daisy chain circuit. The comparator is a wired AND doing a compare the cycle after the step.

Reduction rule unit

The reduction rule component implements the reduction unit with all its tables.

Registers

The *phase*, *control word*, and *sequence* are registers working all the time. There is no way to change their behavior by time. In a quiescent state their clock are still working and power is consumed.

The *active rules register* is an output from the AND-plane of the rules control memory. It works as a buffer and is used to eliminate glitches. Some few of the registers bits are set, the remaining are off. Hence there are few switching bits. The register has a preplanned control signal. Generally it is only needed to strobe under some few steps of a logic operation.

The *timed operation registers* is an analogous buffer from the output of the OR-plane of the operation control memory.

Read only arrays

The AND-plane of the *rules* control memory consists of one transistor for each strict input. Each output wire is reset when result is transferred to the active rules register.

The OR-plane of the *operator* control memory consists of one transistor for each strict input. Each output wire is reset when result is transferred to the timed operation registers.

The OR-plane of the *start operator* control memory has a pass transistor between the the timed operators register an the input of a sequence shift register. The pass transistor is controlled by the phase. The output wires are reset by the first step of the sequence register.

The GATE-plane of the *control word* control memory contains inverse pairs of wires for each sequence step. There is one transistor used for 0 and one for 1 output. All output wires from the memory are controlled. Thus there are only switching on control wires being active. Only some few wires are switching.

Memory

The memory is the storage element of rp8601. As will be shown in this section it is the major component of the hardware. It dominates the chip area, power dissipation and delay time. It is the key component of the rp8601 paradigm. The major difference with a conventional processor is the ratio processor area versus memory cell area. rp8601 has a very small non-memory part allowing more reduction processors for the same amount of memory!

								switching transistors		
	width	no	cwires	bits	control	T/cell	T	idle	step	reduction
data-path										
core	38	22	4	836	88	8	6 688	0	608	
time T	38	1	6	38	6	18	684	0		
time +1	38	1				12	456			
time =	38	1				6	228			
acc	38	2	4	76	8	8	608	0		
schedul	38	6	4	228	24	8	1 824	0		
comm	38	2	8	76	16	20	1 520	0	1 520	
net	38	28	4	1 064	112	8	8 512			
				2 318	254	88	20 520			
reduction rule										
phase	90	1	2	90	2	16	1 440	736		
state	80	1	2	80	2	16	1 280			1 280
active rules	500	1	2	500	2	8	4 000			80
timed operator	150	1	2	150	2	8	1 200		24	1 200
sequence	80	1	6	80	6	18	1 440	480	54	
control word	254	1	3	254	3	19	4 826	2 032	266	
				1 154	17		14 186			
control memories										
rules	500	80		40 000		1	40 000			20 000
operator	150	500		75 000		1	75 000			75 000
start operator	150	90		13 500		1	13 500		450	13 500
control word	480	254		121 920		2	243 840		28	
				250 420			372 340	3 248	2 950	111 060
activ signals										
control word	14									
sequencies	3									
active rules	10									
activ core nodes	2									
n step/reduction	20									
energy/transistor	63,0	aJ								
fdata	5,0	GHz								
P/transistor	0,16	µW								
area array/transistor	0,0625	μm^2								
area data-path/transistor	0,5000	μm^2								
P	1,18	mW						0,51	0,23	0,44
area	40 624	μm^2								

Physical performance exclusive memory and arithmetic unit

There is no such memories as the one asked for. This is a more commercial issue than a technical. Circuits are described but not the constellation. The challenge is to get the proper design of the memory.[7]

The RAM memory cell is almost fixed to a 6 transistor cell with scaled transistors. There are CAM memory cells of size of 12 (with and without scaled transistors), 10 (with active load), and 8 (with polysilicon resistors) transistors presented in literature.

The project has invented cells with 6 (active load), 4 (with polysilicon resistors), and a dynamic of 4 transistors. All these memory cells are described in order of decreasing chip area. They are easy to plug into the memory architecture. The general trend is that the bit-line drive is more complicated for the smallest memory cells.

[7] The project used one large closure as memory cell. There was no significant drawback compared to single element wide memory word. There was a need to have one additional wire for each word-line. In the layout it did not change the size of a memory cell. The use of memory words of element size has the issue where to store the list type.

First operation

The first operation has not been used in memories before. In conventional computers, lists are built to administrate what the first operation does.

It is used to select one of several when searching for a *free* closure cell and a *waiting* closure.

The free closure cell could be implemented by additional register hardware. Hence each free closure is assigned a number as on the hight of a stack. The stack height is placed in a register and in the free closure. There is a search for this cell when creating a new closure. The method is simple and good!

In the case of a waiting closure there is not such an order. Threads could be stacked and a fixed priority order used. To do this it is necessary to add an additional closure to link the thread. This additional closure stores the stack height.

As a computer architect I say: this is possible but it is a violence on the scheduling mechanism, especially the breadth first mechanism, which is hard to implement.

Considering the operation as a hardware implementation. It is an additional priority decoder. Such one could be a long daisy chain of and-gates. The semantics is simple but this implementation is prohibited for large memories.

Cascading the memory in hierarchical sections may reduce the propagation delay to log memory size. Each level needs a local daisy chain and a wired OR of priority requests. Thus each level needs on transistor for the OR and an AND-gate for the daisy chain together with two wires.

Assuming a size of 16 for each level, 5 levels are 1M closures or about 50Mbyte. The raw delay is 5 OR and 80 daisy chain steps. By certain slower than one memory access. However, the access could be buffered by a register element, and in that case speed is not an issue. The additional area is in the range of 2 memory cells. Thus the overhead is in the range of 5-10%.

It is motivated to use a hardware *first* implementation!

Word head

The memory stores elements. They are formed into groups being closures. There may be a hardware partitioning of the words into closures. An other way is to store a marker in the first word of a closure.

The find memory operations must know what parts being accessed. ***find free***, ***find exec***, and ***find id*** searches among the first element being the identifier of the closure. ***find ids*** searches all elements. Adding one bit to the word makes all memory words identical. The borders between closures are set at initialization. This first bit may also be read-only, a mask places a simplified memory cell consisting of one wire and one transistor.

The logic of the word head is simple. A straight forward circuit consists of 14 transistors, 10 N and 4 P transistors. It uses 5 control wires. For some of the memory cells mentioned 2 additional P transistors are needed. The local speed and power dissipation are good.

This cell should fit at the edge of a word, either at one edge or interleaved at both edges. Using a technology rich in wires makes one edge implementation feasible[8].

The memory cell design may complicate the word head by need of special sense and drive circuits.

Bit-line driver

The logic function of the bit-line driver is very simple: a latch, an external bus transmission gate, a decoder (one gate) used for controlling the various memory operations, plus driver and sense amplifier. The two latter are the complex ones and have the major impact of the design.

The memory cell used defines what driver and sense amplifier to use. Switching the bit-lines consumes the major part of the power dissipation. The read delay time is defined by the integration of sense amplifier and the memory cell.

Memory cell

As shown above the memory cell gives the characteristics of the memory.

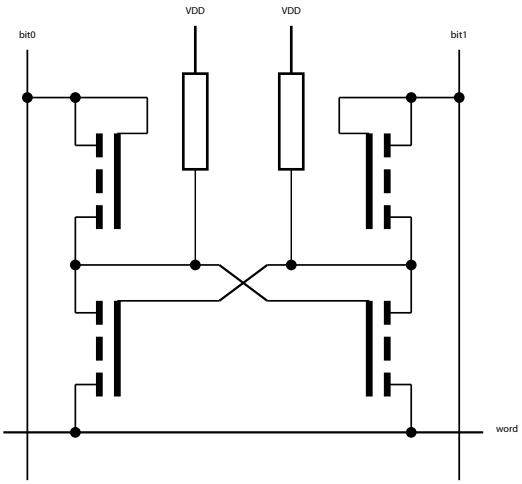
The area is given by the number of transistors, the use of both P and N channel transistors, the amount of wires in bit as well as word direction. In the bit direction the cells use 2 wires. In the word directions the earlier cells use two wires and the project cells use one wire. To this comes VDD and GND. GND is not used in the project cells. Complementary cells are used in the earlier cells and only N transistors for two of the project cells[9].

This report should focus on a memory cell containing 4 N-channel transistors, 2 polysilicon resistors, 2 bit wires, one word-line and an additional VDD wire.

---

8 The word head was of this design in the project.

9 The project studied CAM memory cells. This study found that there are alternatives to the large cells presented in literature. Technoligy has been tuned to form good SRAM cells. The technology used in the project did not have those qualities. Thus it remains to use modern SRAM technology to analyse those cells.



In short it works as follows:

- in idling state bit-lines are low
- in write 0/1 bit-lines are complementary and the word-line low
- in reading the word-line is low, the currents in the bit-line are complementary. A sense amplifier increases the signal levels
- an equal operation is performed in two steps: the bit-lines and word-line are low; the key is complementary driven on the bit-lines. Two of the transistors work as diodes, thus the word-line is assigned the maximum value from these diodes only for those bits being conducting. It is used as an XOR gate for NOT EQUAL.

In a more detailed analysis all three operations use the same voltage scheme - how to differ between the variants? It is a matter of scaling and driving force.

Performance

The memory cell is compared to SRAM cell and other CAM cells.

The *idle* performance is defined by the load characteristics: polysilicon or complementary CMOS transistors. All circuits perform good.

The *write* is performed in almost the same way in all cells. Scaling between transistors may influence. Time to write depends mainly on bit-line driving capacity. The project cells use diode giving slow switching characteristics. These are not as important as the bit-line drive.

Reading differs between the cells. Generally the cells have a stable state with one of the N-channel transistors heavily conducting. There is a transistor between the bit-line and the internal node. In some way it is scaled or as in the project cells being a diode (giving lowest current). They all produces a low current into the bit-line. A sense amplifier measures the current by integrating the charge in the bit-line capacitances. Thus the combination of bit-line capacitance and current is the key factor.

The *equal* operation differs a lot (SRAM does use this operation), all non project cells have half an exclusive or implemented by 4 transistors. They work in forward direction and conducts heavily. Equal is fast. The project memory cells uses diode configured transistors to conduct. The current is lower than in the other case, and when voltage rises on the word-line it becomes even less! Thus equal is slow!

There is no reason that power dissipation should differ much between the cells. All have approximately the same capacitive load in their nodes, bit-lines, and word-lines.

There have not been any comparative layout between the discussed cells. However the use of fewer wires, less amount of transistors, and of only N-channel types ought to give a substantial lower area for the project cell.

Performance

The table on page 33 gives details for the different units. The units are designed with the parameters shown at the end of the last section.

Performance is measure in per unit the width, number of units and number of control wires per unit, the amount of bits stored, total number of control wires, number of transistors per cell and total amount.

The calculation of power consumption is based on all energy loss is caused by switching transistors in an idle state, one active clock step and for one logical operation (reduction).

In the lower left part of the table the characteristics of a transistor is shown. Under active signals the characteristics of the control is defined. From this the area and power consumption in a running reduction processor is calculated.

The power consumption exclusive memory and arithmetic unit without net traffic is about 1.2mW and the area 0.04mm^2. Special small units are probably negligible.

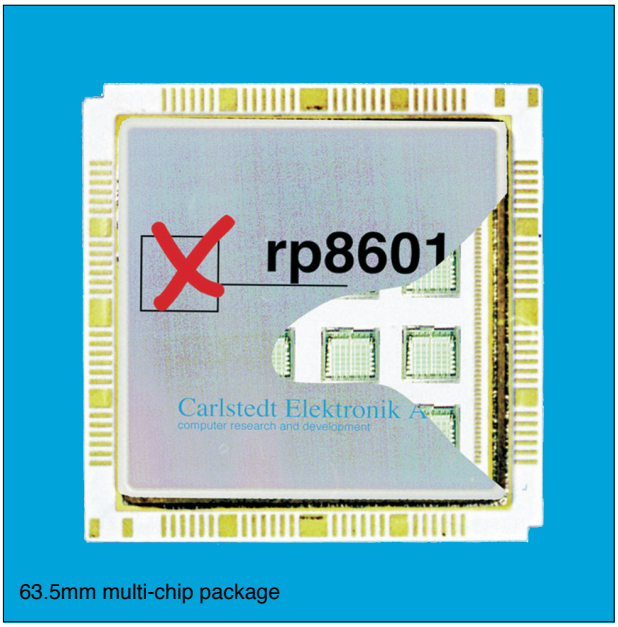
An arithmetic unit as described in last section is probably less than 20.000 transistors, corresponding to an area of 0.01mm^2. It is not fully utilized, but have maximum power consumption of 3mW.

A real implementation result in higher values due to ignoring wire capacitances and some glue needed to separate transistors.

PACKAGING

The project developed two multi-chip packages for a multiprocessor. There was a smaller one, size 63.5mm square, based on air cooling and larger water cooled assembly being a pile of packages, each 152.4mm square, for about 2000 chips each housing hundreds of reduction processors (today).

The water cooling system used a frame placed between stacked multi-chip packages. The water pressure pressed a thin foil against the package. Water flows diagonally through the frame. Integrated in the foil was wires and pads to be in contact with pads on the package. The foil formed the inter-package communication wires. A separate power connector was used on each edge of the frame. They formed a connection between the multi-chip package and 6 vertical high current power supply bars, VDD and GND.
The thermal path was short: from the die, through silver paste, the substrate, and through the foil to the water. Because of the water pressure and the flexible foil there was a negligible distance between the substrate and the foil. The design goal was about 200W dissipation per package.

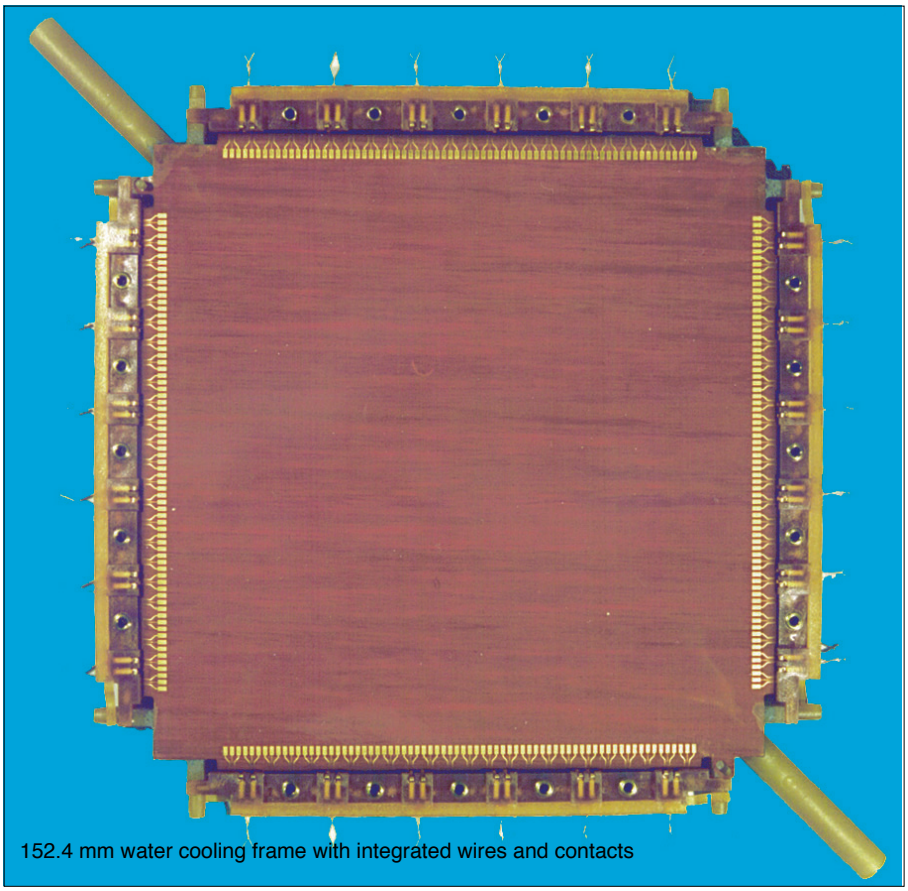


63.5mm multi-chip package

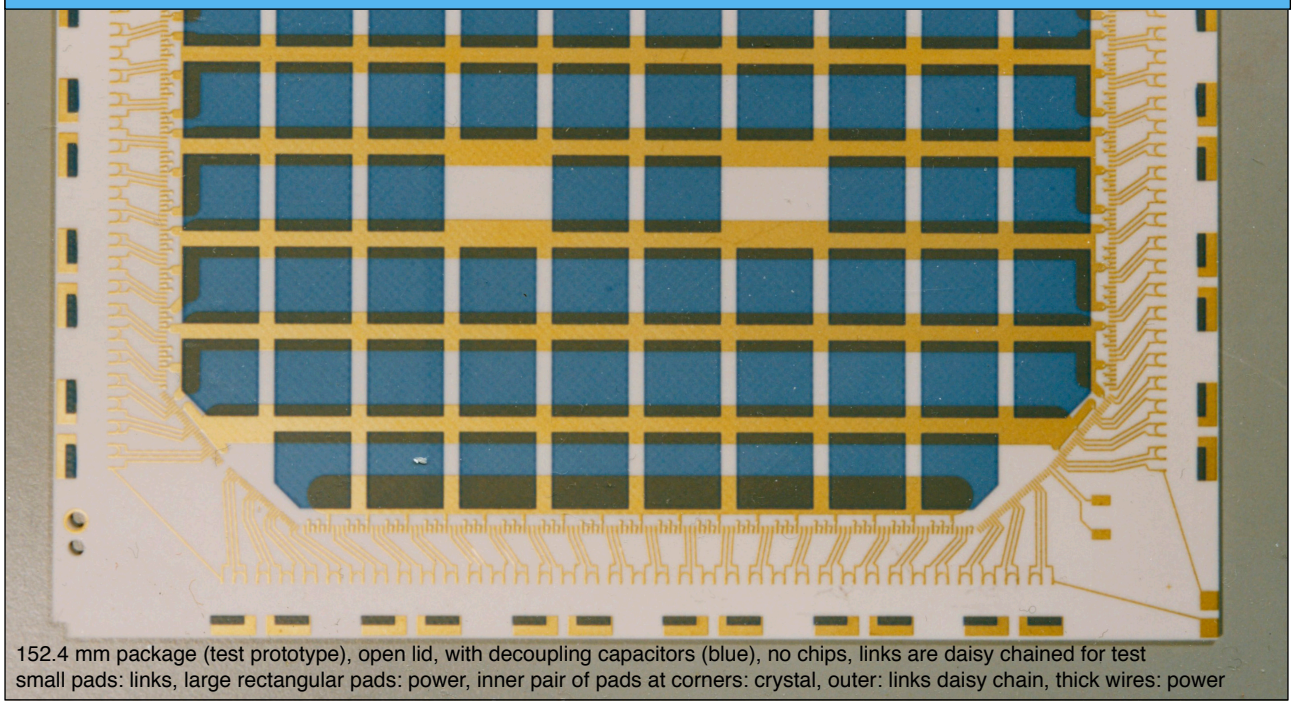
The large multi-chip package consisted of a laser cut substrate 1.27mm thick. A laminated ceramic sheet with a rim to form a cavity was used as lid. The cavity was filled with gel (injected through two holes) in order to sustain pressure.

Signal and power wires was printed with thick film gold paste. Vias were laser cut. A decoupling high-capacitive layer was used for decoupling power wires. The contact points were gold balls. There were 4x24 communication links between neighboring packages. A link consisted of 3 wires. They formed strip-lines.

The package was designed and tested to fulfill military range temperature and vibration.


152.4 mm water cooling frame with integrated wires and contacts

152.4 mm package



152.4 mm package (test prototype), open lid, with decoupling capacitors (blue), no chips, links are daisy chained for test
small pads: links, large rectangular pads: power, inner pair of pads at corners: crystal, outer: links daisy chain, thick wires: power

CONCLUSION

A computer architecture has been described, All major parts from language through logic implementation, and chip circuitry to final packaging has been described.

It is scalable from a small about 1mm^2 computer area to at least 4000x50mm^2 used in a pile of multi-chip packages. The major part is memory. Arbitrary many reduction processors may be used. The smallest one could house about 250kB memory and some few processors. The larger piled one about 50Gbyte and 200,000 reduction processors. Each processor performs some hundreds of million reductions/s.

Partitioning an application is not an issue. The application works fine in an arbitrary large structure. Performance many be tuned to use most of the performance.

The technology is available now. However there is a need to put it all together in a working concept.

FINAL WORDS

Future developments

The rp8601 is a new paradigm. It has never been tried before. Behind some parts there are considerable experience, and others very little. The project was not finalized.

The classical von Neumann machine has advanced from a very simple word oriented machine to the advanced and complex processor used today. Features have been added to reach economically viable devices. The design in this report is to be considered to be the first attempt in a new paradigm. Hence it should be followed by subsequent refinements.

Closure size

A closure is from logic point a very efficient item. The implementation of the closure being a function introduces identifiers to link closures. They are additional information needing more memory space. Assuming that each element of a closure being one word, and a typical closure contains 3 elements, the packed variant needs 3 words but the rp8601 one 6-7 words. It is an overhead 2-2.3 times.

In order to solve this problem a system consisting of two parts may be used. On for packed back end storage of information and one other. The packed one should store not single closures but sub-trees where the overhead is then negligible.

Memory size

In conventional machines there are 8, 16, 32, 64 bit words. The compiler selects a memory layout to be efficient. In rp8601 each element in a closure has fixed size. Storing a structure of byte-size information in a conventional machine takes 8 bits but in rp8601 about 250 bits, an overhead by 30 times.

In a conventional machine the address is to RAM where data can be abutted. In rp8601 the smallest element is a word of an associative memory. Dividing the word in parts keeping the search mechanism is challenge.

There are some obvious ways by introducing more word heads that could be cascaded. Word heads have a significant size but not prohibitive. At 8-bit words the additional area is probably 100%, allocation becomes harder and the memory slower.

Another method is to implement efficient operators packing/unpacking short words into one element. In such case the storage overhead is as shown in the preceding section.

Dataflow type of instantiation

The H language is described to cope with any type of application. Some could be described by a single fully expanded representation (e g fast Fourier transform being one large function) using only argument access by **sel** construct. When using a functional top down reduction a lot of pending memory cells are used. They cause an overhead in memory utilization.

These programs can benefit from not using a functional method of representation instantiation but a data-flow type. This type of instatiation starts from the leaves and builds the tree towards the root.

Such a representation is a graph laid out over all processors. Current understanding is that this add-on is not a language issue. There is very little that has to be added as rewrite rules and nothing in hardware.

Hardware simulation

A reduction processor consists of some regular structures like memory, data-path and reduction rule unit. They are the key to performance. There are around 20 components consisting of totally about 200 transistors.

These cells should be designed and an approximative layout should be made in an up-to-date technology. A complete processor consisting of one component on each place with loading stubs can characterize the electrical performance. A Spice simulation should give the electrical characteristics.

Simulation of reduction rules

In order to understand the complexity of the language, a simple simulator should be written were rewrite rules are clearly indicated.The basic execution mechanism should be verified and the equivalence rules should be the result.

Programming experience

The H language is a new untested language type. It has not yet been used. There is a need to test the language for real applications.

Future research

The storage for expressions should be researched, as well logic structure as circuit. It is fundamental to the success of the architecture. When cost of chip area becomes negligible compared to other system cost the current rp8601 computer paradigm is viable.

Patents

There were 34 patents and 234 patent applications for parts of the processor filed in major industrial countries throughout the world. None of them are now maintained.

Authours contact address

gunnar@hylab.se or gunnar@carlstedt.se

ACKNOWLEDGEMENT

I am grateful to have had a number of skillful people around me. During my time as computer architect Ingemar Carlsson let me participate in many advanced project giving me an invaluable experience. After some years Erik Tengvald become my companion and further on Robert Tjärnström. As a group we were very productive.

The project participants

Cyrrus Morteza Afgahi; Birgitta Andersson; Wignesan Balakrishnan; Kenneth Berggren; Thomas Berndtsson; Kristina Björklund; Staffan Bonnier; Jim Brown; Göran Båge; Andrzej Ciepielewski; Gunnar Ekolin; Christina Eliasson; Bertil Emmertz; Bertil Engman; Lars Ganrot; Ulf Gunneflo; Staffan Gustafsson; Sverker Hansson; Carl Hemmingsson; Ingemar Hernefjord; Peter Hesseltun; Torbjörn Holmberg; Hans Jakobsson; Odd Johansson; Martin Jones; Ing-Marie Jonsson; Jörgen Jonsson; Liselotte Jonsson; Feliks Kluzniak; Jonas Lagerblad; Bengt Lennartsson; Ralf Lundberg; Jeanette Munro; Lars-Åke Nilsson; Tony Nordström; Sten Norrman; John Oldfield; Ingvar Olsson; Lars Pareto; Mats Persson; Thomas Persson; Roland Pettersson; Mats Rimborg; Borhan Roohipour; Lars Samuelsson; Johan Schubert; Nahid Shahmerhri; Hershel Shermis; Bertil Sigfridsson; Roger Skagervall; Jan Stein; Hans-Erik Strömvall; Rolf Sundblad; Leif Söder; Anne Thurén; Staffan Truvé; Göran Uddeborg; Kennet Vilhemsson; Jesper Vasell; Handong Wu; Anders Ödmark; Kenneth Östberg; Magnus Österholm;