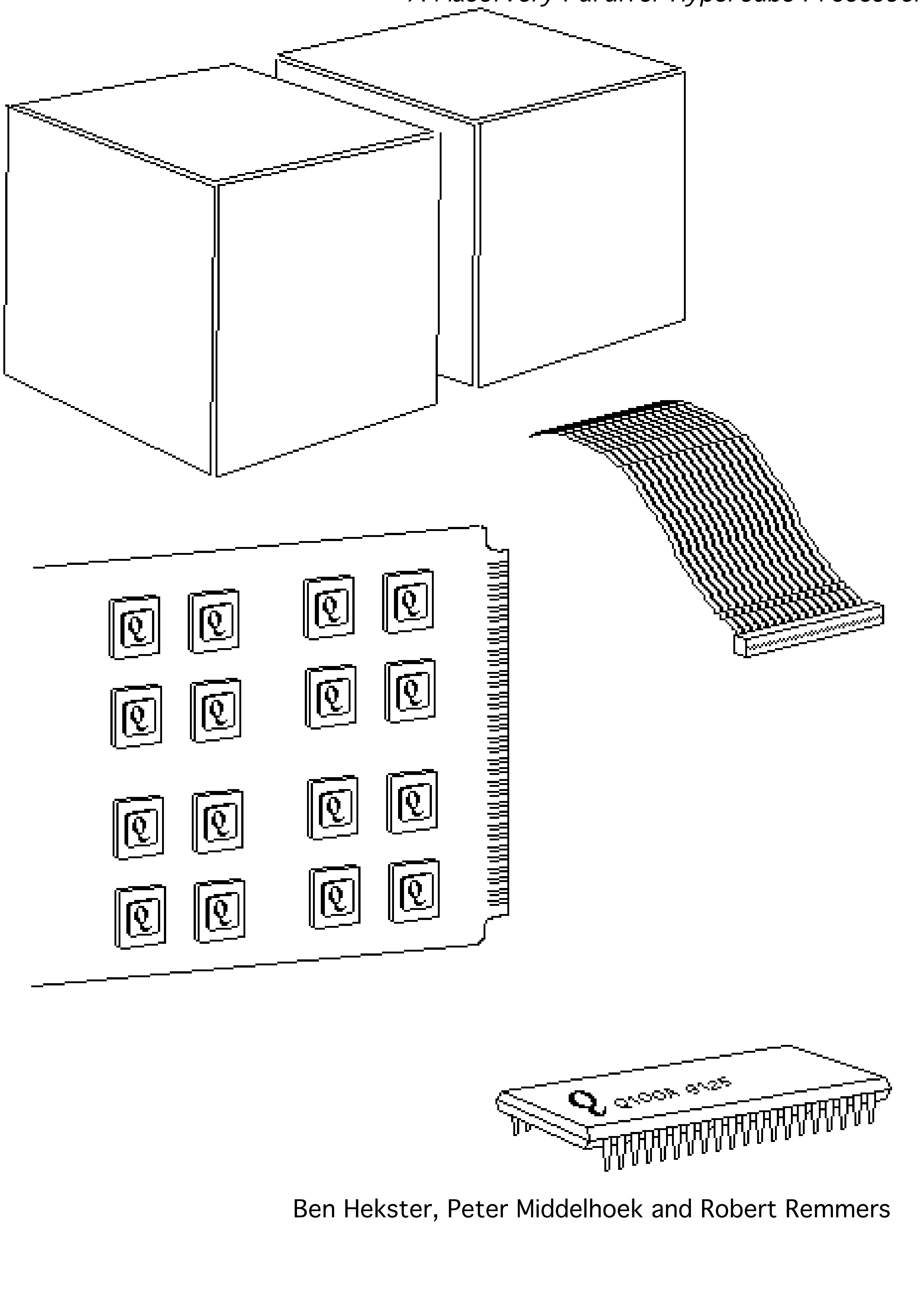
Qbe*rt

A Massively Parallel Hypercube Processor



Ben Hekster, Peter Middelhoek and Robert Remmers

Nothing comes easy, and that's a fact
Nothing comes easy, but a broken back
Nothing comes easy, it never will
Nothing comes easy, but a broken will
—Work Hard, Depeche Mode [People are People]

We are paid by those
who learn by our mistakes
—The Working Hour, Tears for Fears [Songs from the Big Chair]

The sweetest perfection to call my own
The slightest correction couldn't finely hone
—Sweetest Perfection, Depeche Mode [Violator]

Preface

This document reports the assignment which we performed for the senior-year course in *VLSI System Design,* the design and realization of the Qbe*rt processor. Mainly because all of the relevant terminology has already established itself in English we chose to use that language, rather than using often awkward Dutch translations of such terms.

Justification Although we understand that for this assignment it is typical to implement another sequential processor with all the usual processor elements, such as a microcontroller and registers, we feel that this process has already been exhaustively investigated by others before us. Several levels of such an implementation have already been adequately addressed in another course[1] and [4].

Instead, we set for ourselves a somewhat more challenging and uncertain goal by choosing to design and implement a *hypercube processor*. Such a processor is designed from the ground up to work together with many other identical elements in a massively parallel environment, to solve computational problems exceeding the capabilities of conventional sequential processors.

A very significant factor that we had to consider in our decision to step outside of the clearly trodden path of sequential processor architecture was the uncertainty that lay before us. When we initially started considering the architecture of the interprocessor communication protocols and hardware it was not at all definite that our concepts were indeed at all realizable. The fact that one teacher from the Computer Science department had considered these issues before and not yet found a practicable solution was at the least daunting.

Even though we would like to imagine that our processor is umbly modeled after Thinking Machines' well-known *Connection Machine* processor, it is important to realize that most of the lower-level implementational information is still considered very much proprietary and not actually publicly known [1]. The machine and processors are relatively new so many of its more unconventional concepts are not yet well established in literature. This implies that such issues had to be inferred from often vague and on occasion incorrect information.

Rather than using a messy *ad hoc* type of report we have chosen for a highly structured form. Chapters which describe higher-level aspects of the processor give rise to subsequent chapters which deal with the design and implementation of their subcomponents. As such the ordering of chapters in this report does not imply a chronological relationship.

Concluding, we would say unanimously that we enjoyed having actually designed our own processor. Although we regretted not being able to spend more time to build a more complete processor or to commit to extensive testing of a processed product, other duties precluded this.

Throughout the text this symbol is used to indicate features or functionality that could not be included due to time constraints

Ben Hekster, Peter Middelhoek, Robert Remmers

---

[1] *Uitrusting van Digitale Systemen*

Contents

Created on the Apple Macintosh, with Microsoft Word and Claris MacDraw II.
Printed in 10-point New Century Schoolbook

Printing date Monday, October 2, 2000

1 Design

It is becoming clear that sequential processors are beginning to reach absolute limits to their performance. Further improvements will have to be sought in new and revolutionary concepts such as parallel computing. A massively parallel (or *fine-grained*) computer takes this concept to its extreme, using a very large number of perhaps relatively simple processors to achieve great overall processing speeds. A well-known and rather intriguing example of such an architecture is Thinking Machines' *Connection Machine* which combines 65536 1-bit processors in a hypercube topology.

We decided early on to attempt to emulate the hypercube architecture and processor, because it is rather different from anything we have previously encountered and its usefulness in solving practical processing problems beyond the means of conventional processors has been established. We were very interested in investigating for ourselves the problems and issues in such architectures. This process culminated in the design of the Qbe*rt (for *cube routed topology*) processor.

It was neither possible nor desirable to copy the architecture of the Connection Machine in any level of detail. In following sections, we will deal with the overall organization of the machine of which our processor will form the cornerstone and make appropriate comparisons with analogous Connection Machine structures where possible.

This chapter describes the architecture of our Connection Machine computer, processor chip and processor node. Careful distinction is made between these three entities throughout the report.

Processor Topology As stated, the Qbe*rt and the Connection Machine employ a number of processors connected in a hypercube structure. A **hypercube** is the higher-dimensional generalization of the usual 3-dimensional cube. A single processor, called a **node** of the hypercube, is located at each of the corners of the cube and may be uniquely identified by its **address**. This is the ordered n-tuple of its coordinates (a_{n-1}, a_{n-2}, …, a_0), where n is the **dimension** of the hypercube. The following figure depicts a four-dimensional hypercube.

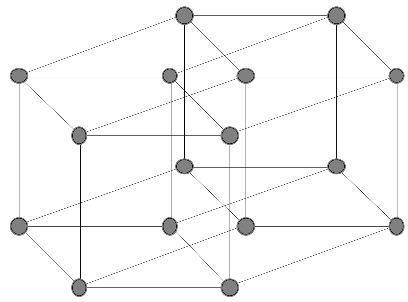


Figure 1.1. Four-dimensional hypercube

Note that each processor has exactly one neighbor to which it is connected along each of the dimensions.

For reasons of efficiency the Connection Machine actually has 16 processors on a single chip. This does not detract in any way from the hypercube nature of the computer—the particular distribution of processor nodes over chips is irrelevant to Connection Machine programs.[2] The Qbe*rt processor chip contains only four nodes. This was a decision made early on in the design before we had accurate information about chip area usage. The number was chosen as the minimum number of nodes that were required for the design of the on-chip interconnection logic to retain its essential structure.

[2]This disregards optimizations that might be achieved by carefully distributing strongly bound computational tasks over nearby processors

Some advanced features of the Connection Machine, such as the ability to connect its processors in an arbitrary two-dimensional mesh, and the 'flipper' network which allows processors to access the local memory of other processors on the same chip, are not supported by Qbe*rt. Also, although the router design supports the simultaneous sending of messages between any two processors, processor instructions currently specify only a single relative address for all messages that are sent.

Processing Model Our multiprocessing computer, like the Connection Machine, operates on a single-instruction multiple data (SIMD) model. This means that every processor executes the same instruction on different data. There is one **microcontroller** which broadcasts instruction words to every processor chip simultaneously. Since broadcasting is not a message-like sender/receiver transaction they are not sent through the hypercube network but through a dedicated **instruction bus**. Use of separate buses for instructions and data (also known as a Harvard architecture) eliminates a source of contention between the processor and router.

The Connection Machine's processing model is actually somewhat more refined than this, as it can be configured as four separate quarters which can each be connected to separate or shared front-end computers (such as a DEC VAX or Symbolics LispMachine). Each of the quarters is equipped with its own microcontroller which allows them to operate more or less independently. In contrast, our design uses only a single very simple microcontroller. It fetches instructions from a separate program memory and sends the bits of the instruction word time-multiplexed over four cycles over the instruction bus. Furthermore, the clock signals it generates keep the processors synchronized, which is crucial during message transmissions.

For initialization of the processor's memories and retrieval of the stored results, the front-end could be equipped with one or more serial links which would be connected into the processor network:

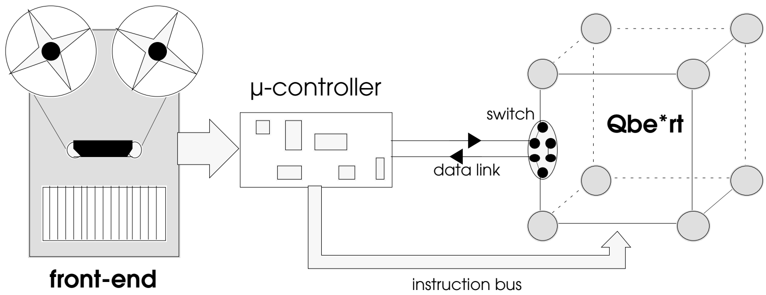


Figure 1.2. A Qbe*rt computer system

Because they are different structures, and the mandate of this course is the design of a single VLSI chip, we have not completed exhaustive designs of the front-end or the microcontroller.

Programs running on the front-end computer can use both the parallel computing power of the Connection Machine, and the host for scalar and program flow instructions. As to the latter, the Connection Machine and Qbe*rt both incorporate a form of *conditional execution* of instructions by allowing sets of processors to be either used or excepted from an instruction depending on the contents of local memory.

Programming The unusual architecture of the Connection Machine and Qbe*rt raises questions as to its usefulness and performance. With most programmers now used exclusively to sequential Von Neumann programming, massively parallel computers require a radically different approach. The use of many very limited processors to attack real-world problems may require some illustration.

The answer to the question whether massively parallel computers are in fact useful is clear, considering the success of the Connection Machine and the fact that new models (the CM2 and CM200) have been introduced. Beneficial performance largely depends on the ability to calculate many results in parallel. Fortunately there are many such problems—examples of productive uses of the Connection Machine are documented in [2] and [3].

An advantage of 1-bit processing is that it allows problems to be solved to exactly the precision actually required. For instance, operands using only 24 rather than 32 bits of precision can be processed with 24-bit operations, eliminating wasted processing power.

The Qbe*rt instruction has the following assembler mnemonic format:

operationb operationc, operanda, operandb, operandc, resultc, condition-sense, condition-out

For example,

```
ADD SUB $23, $24, $A, $4, $5, true
```

is a valid Qbe*rt instruction.

Architecture

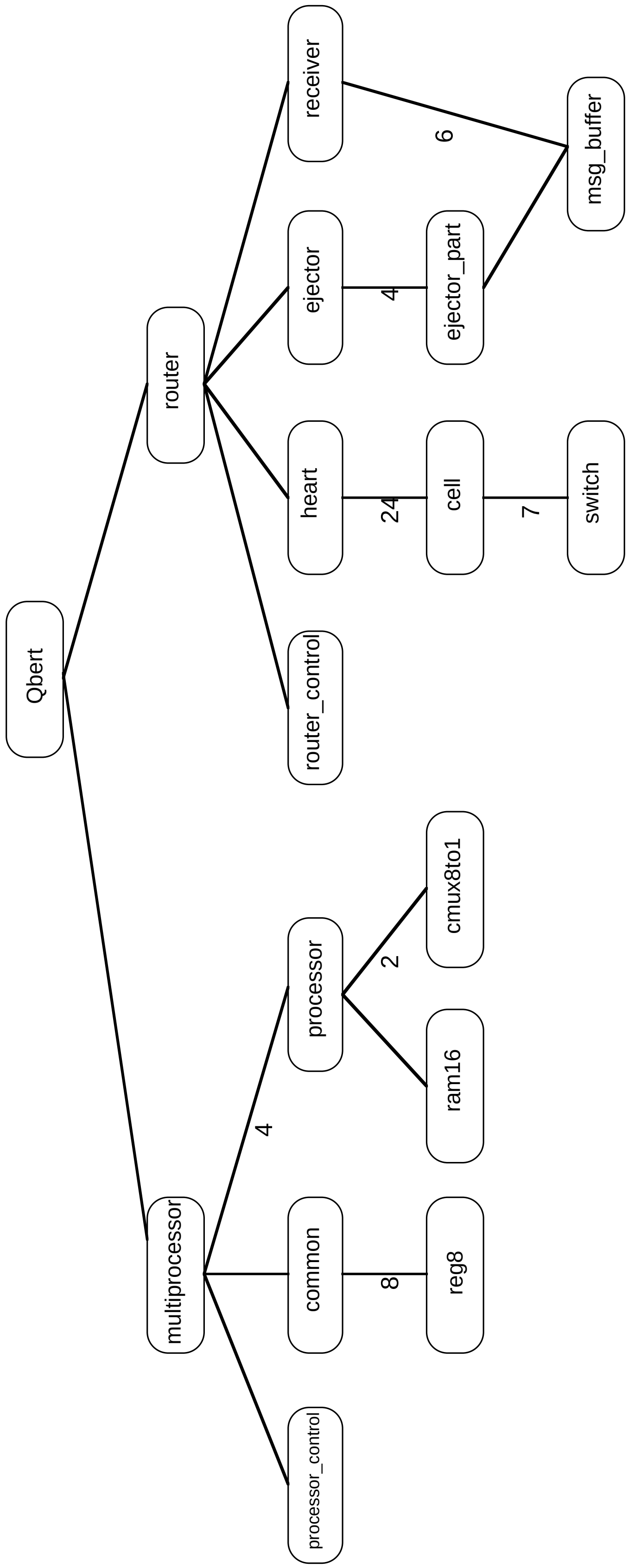


Figure 1.3. Qbe*rt processor architecture

Message Routing

A processor may send **messages** to any processor (including itself) in the hypercube. In some cases the processor to which the message is directed is located on the same chip as the sender. In other cases the processor is located on a different chip and messages need to be transmitted through external links. The delivery of messages between processors is accomplished by virtue of **message routing**.

Every chip contains a single router shared by the local processors, which performs the necessary directing of interprocessor messages. The router receives messages from the local processors (along the **internal dimensions**) as well as from the routers on neighboring chips (**external dimensions**).

The purpose of the router is the sending and forwarding of messages to their destinations, as specified by their relative addresses. This may involve sending messages to one of the hypercube neighbor routers across the **external network** or to one of the local processors through the **internal network**. This includes messages directed to the sender itself. If all of the external connections it might want to traverse are already in use it may be necessary to **misroute** a message along a dimension that it does not need to go through. In the original 80's arcade game Q*bert™ the destination was similarly reached from a single starting point by making successive routing decisions.

The router uses **relative addresses** for addressing and delivering messages. This means that processors do not need to be assigned addresses from the start and that the processor network is completely symmetric. In fact, a program may be run however it is rotated or mirrored across the hypercube as long as its topology remains unchanged. As messages travel across the hypercube network their relative addresses are modified at each router stage to indicate the crossing of a dimension. A message has reached its ultimate destination when its relative address is zero.

A message contains the destination node's relative address, followed by a single data bit. The message format is depicted in the figure:

Figure 1.4. Bit-serial message format

The m bit is 1 if the bit stream carries an actual message.

The design of the heart is such that it can be shown that messages[3] are never lost. The heart makes a best effort to deliver messages to their proper destination, but when multiple messages want to cross the same dimension this cannot be guaranteed.

There are possible cases in which a message cannot be routed into a dimension that it needs to traverse. In the extreme case a message cannot be routed along any of the dimensions it still needs to cross. If such a message cannot reach the ejector for any reason it will automatically be **misrouted** by being sent across a dimension that it does not need to go to. The message is not lost—it is merely delayed in reaching its ultimate destination since one of the following routers will need to redirect it back along that dimension.

The Qbe*rt computer is designed for use with any number 2^n of processors, where $n \in \{2, \ldots, 8\}$. No hardware reconfiguration is required other than the addition or removal of processor chips in the network. The processors are not aware of the actual number of nodes in the hypercube and always operate as if the maximum number was available. Missing external dimensions must be 'folded back' onto the chips to retain the hypercube topology. In effect, this makes routers work for missing nodes to reroute messages back to available processors.

[3]The proof is left as an exercise to the interested reader

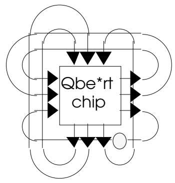


Figure 1.5. A two-dimensional Qbe*rt hypercube

Performance Our target clock speed was 10 MHz (100 ns cycle), which should still keep the design relatively insusceptible to difficulties often associated with high-speed circuits, such as signal reflections, and allows the use of relatively slow memory. At this speed a standard personal computer can still be used as a front-end.

With this and other basic design parameters we examined the feasibility and performance of a maximum implementation (2^8 processors) of a Qbe*rt computer. In the current implementation all except send instructions have a fixed duration of four clock cycles. Send instructions have a variable length which depends on all the messages that are sent in the network at the same time. At a minimum of 4 clock cycles per instruction the maximum Qbe*rt computer thus delivers a peak performance of 640 native MIPS. A more realistic rating might use a 32-bit integer addition as a basic ‘instruction’. This requires 32 native instructions on a single processor programmed as a simple sequential ripple adder. In parallel, Qbe*rt performs 256 additions in $32 \cdot 4$ cycles, equivalent to 20 MIPS. Using 24 bit operands this increases to 25 MIPS.

In science and engineering floating-point performance is probably more important. A floating-point addition can be divided into four steps: mantissa alignment, addition, normalization, and rounding, which we conservatively assume each to require one native instruction per bit. The addition of two double-precision (64-bit) floating point values therefore requires $4 \cdot 64 \cdot 4 = 1024$ clock cycles, giving a quite reasonable performance of 2.5 Mflops (actual results will differ because the operation starts with two operands and merges them into one result operand). This result compares favorably with the 100 Mflops of the original CM1 computer [7].

A 256-processor machine built from 64 chips has $64 \cdot 6$ bidirectional communication links, each transmitting 10 bits per send instruction. If a send instruction requires 14 clock cycles† (4 to execute the instruction and 10 to transmit the message) the overall communication bandwidth is

$$10 \cdot 10^6 \text{ cycle/s} \cdot \frac{10}{14} \text{ bit/cycle} \cdot 64 \cdot 6 = 2.74 \text{ Gbaud}$$

of which 10% is actually data, giving a maximum data transfer rate of 274 Mbit/s (over 1 Mbit/s per processor).

Unfortunately this calculation is too optimistic since each processor can only actually send one message per instruction. This gives a maximum utilized bandwidth of

$$10 \cdot 10^6 \text{ cycle/s} \cdot \frac{10}{14} \text{ bit/cycle} \cdot 256 = 1.83 \text{ Gbaud}$$

of which 10% is data.

In a congested network it is possible that a message cannot directly reach its destination. In that case it is buffered in an en-route processor and the instruction is extended by additional send cycles (each 10 clock cycles) until all messages have reached their destination. In a scenario where four send cycles are required to deliver every message the data transfer rate drops to 46 Mbit/s.

The Qbe*rt design uses 256 bits of memory per processor, giving a total memory capacity of 8 kB. This is quite low considering that Amdahl's rule postulates that one byte of memory is required for every instruction that the processor can execute per second, which is roughly 20 Mbyte for Qbe*rt. Amdahl's rule also states that one bit/s of I/O capacity is needed for each instruction/s. Here

† As stated, the actual number of cycles per send instruction depends on the presence of other messages in the network and the amount of congestion.

Qbe*rt does well with a typical transfer rate of 46 Mbit/s and a peak transfer rate of 183 Mbit/s.

Figure 1.6 shows how this capability could be used to communicate with a host. Note that it requires a 1.83 Gbit/s communication link between the front-end and Qbe*rt, equivalent to 19 FDDI or 183 Ethernet links.

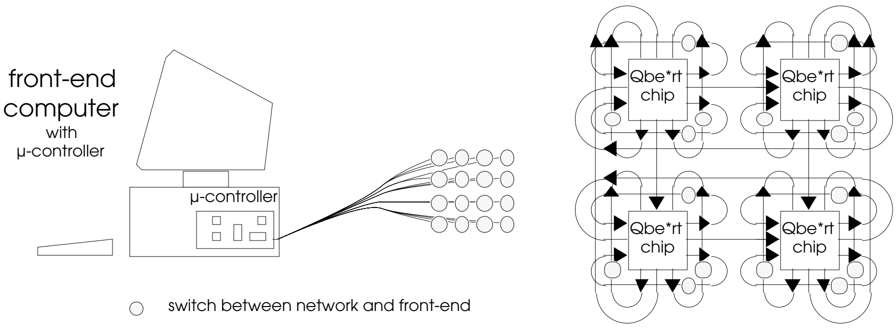


Figure 1.6. A four-dimensional Qbe*rt hypercube with 16 processors

It is well to realize that this architecture is best suited for massively parallel computers, with a number of processors on the order of 2^{16}. The current Qbe*rt design with 2^8 processors only just barely falls into this category and should be considered an experimental proof-of-concept machine.

2 Observations

With the benefit of hindsight we comment on the design presented in this report.

Design parameters Originally we had thought (making a rough transistor-count estimate) that four processors on a single chip could reasonably be implemented on a 100 mm^2 die and still leave room for RAM and ROM of undetermined size (the exact specifications of the memories were purposely left vague so that they could be adapted to available space). Early cell occupancy estimates seemed to confirm this. Incredibly, final placement and routing produced a total effective cell occupancy of only 25% with 90% of the die used, and 64 signal overflows and 72 signal errors. At these rates obviously the RAM and ROM could not have been incorporated at any size.

Many of the decisions were influenced by technical reasons. Others were made considering the need to limit the implementation time.

Improvements Although we cannot approach the incredible performance of our model (the Connection Machine) we believe that our design amply demonstrates the feasibility of designing and realizing a massively parallel machine using limited resources. In contrast, the latest Connection Machine model (CM-200) achieves a performance of 9 Gflops by using 256 times as many processors as and with the assistance of shared floating-point coprocessors [5].

There are several ways in which the performance of an improved Qbe*rt could be substantially (an order of magnitude at least) increased:

- Currently, the pin count limitation requires a four-fold multiplexing of the instruction word. If the chip was packaged with more pins than the 40 that where available in our design it would be possible to read the entire instruction at once and probably execute all non-send instructions in two cycles.
- The clock speed could be increased. A frequency of 50 MHz would still allow use of standard on-chip RAM (with an access time in the order of 10 ns), but would require significant reworking of the router and network communication.
- Increasing communication throughput. The current message protocol provides one bit of transmitted data per message, using a nine-bit message header which is needed to dynamically instantiate the message path. By enhancing the protocol to enable continuous links between processors, the path would only need to be established once and the message headers could be eliminated from the actual data transfers.
- Using resource management techniques such as hardware scoreboarding [6], we should be able to again double performance by fully pipelining instruction execution.

Concluding... Although we realize that our design and implementation are far from complete in real-world or commercial terms, we are convinced that they contain the essential aspects of a massively parallel computer. All the steps in the design process up to the final 'tape-out' have been gone through. Unavoidably we feel that we could greatly improve on our design if we had to do it all over again, but considering that this was the first time that we had designed and implemented a VLSI circuit we are not unhappy with the result.

We are somewhat disappointed in the tools. Our complaints range from simple (a user-interface that displays text that is almost impossible to read due to both a too small size and the use of non-contrasting colors) to the unsatisfactory performance of the placer and router tools. The lack of speed of the tools was at least annoying. We sometimes had to wait, especially when testing the complete Qbe*rt chip, for well over an hour just to get a program started.

We were surprised with the performance of the placer and wire-router. Although initial estimates showed that our circuitry would occupy about 25 % of the silicon real estate, after running the placer and router (on a more powerful workstation) it turned out that it was not possible to have several parts correctly placed and routed in the given area. [9] shows that placement densities of 70 % and higher are very much obtainable using the standard cell approach. This is true both for regular data-path structures like the router part of Qbe*rt and irregular control-type structures. We manually examined some signal paths in the placed part and where surprised by the mess which the placer had made of it.

Clearly a lot of work needs to be done somewhere to obtain a more reasonably efficient placer. The alternative, doing a lot of the placement manually, is not very appealing. The tools are a very significant improvement over manually designing, implementing and testing a VLSI circuit, but it is also clear that they are currently far from perfect and that a lot of development needs to be done before we can really generate a chip in a day.

Finally we would again like to state that we greatly enjoyed designing Qbe*rt and believe that it has been a valuable learning experience.

3 Blocks

The following sections describe each of the blocks created in the implementation in great detail. These descriptions are presented in a common structured format, as shown in the following general fashion. Symbol and circuit diagram print-outs and simulation results of all the blocks are given in a separate appendix.

Function Short description of the block's function

Use How the block is used in the design

Uses List of subcomponents used in the block

Interface

signal	→	signal type	Input to the block
signal	←		Output from the block
signal	↔		Bidirectional

Diagram Implementational diagram of internal structure

Timing Timing of the block's interface signals

Implementation Detailed description of the block's implementation, in terms of its own circuitry.

Comments Caveats, bugs, out of the ordinary remarks

Qbert

Function Elementary massively parallel computer

Use Qbert implements a two-dimensional processing element of a massively parallel hypercube computer. One chip contains four 1-bit processors and a local and global router which allows a higher- (up to eight-) dimensional hypercube to be constructed from multiple processor chips.

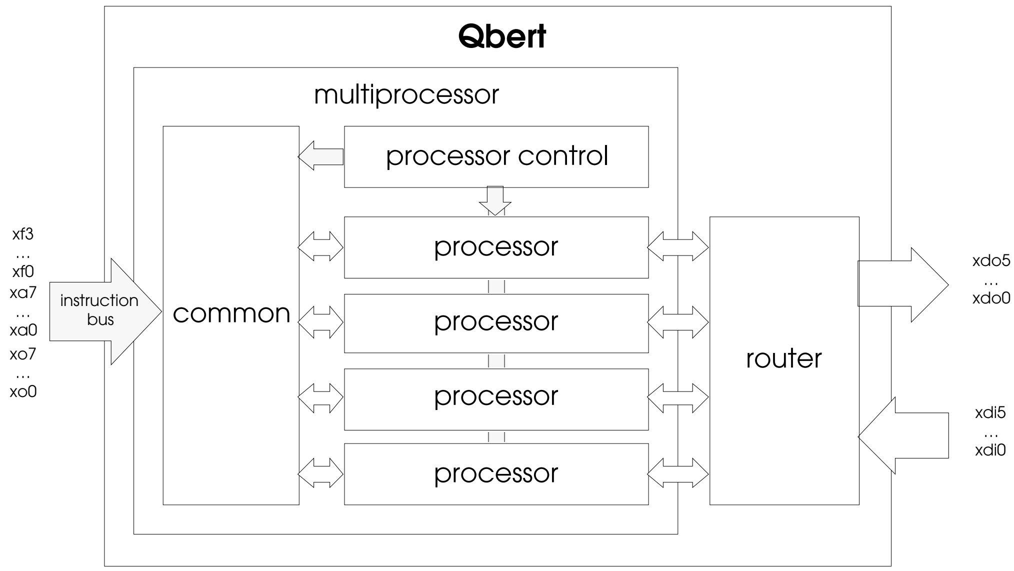
Uses multiprocessor, router

Interface

clk1	→	positive duty clock	Control clock (180° phase lead)
clk2	→	positive duty clock	Data clock
reset*	→	active low control	System reset
xcond	→	active high control	Enable/disable
xbusyin	→	active high control	Undelivered message(s) in network
xbusyout	→	active high control	Undelivered message(s) in node
xf*f*	→	instruction code	Select flag
xa*a*	→	instruction code	Operand address
xo*w*	→	instruction code	Opcode
xdi*d*	→	data	External dimension network data input
xdo*d*	←	data	External dimension network data output

$f \in \{0, \ldots, 3\}$ specifies one of 16 flags
$a \in \{0, \ldots, 7\}$ specifies an eight bit operand address
$w \in \{0, \ldots, 7\}$ specifies an eight bit opcode
$d \in \{0, \ldots, 5\}$ indicates the external dimension

Diagram



Timing None

Implementation None

Comments The Qbe*rt chip design could be modified to incorporate more processors on a single chip. The multiprocessor part scales especially easily—only 12 new processors have to be added and the RAM sized accordingly. The router part is more difficult, but since it is more or less independent extra stages could be added to its switching network without impacting the multiprocessor part.

This property is very useful since it permits the application of newer fabrication techniques when they become available, thereby reducing the chip count of a Qbe*rt computer.

multiprocessor

Function Four single-instruction multiple-data (SIMD) 1-bit processors

Use A multiprocessor implements the processor nodes in a Qbe*rt chip.

Uses Processor, common, processor_control

Interface

clk1	→	positive duty clock	Control clock (180° phase lead)
clk2	→	positive duty clock	Data clock
reset*	→	active low control	System reset
xcond	→	active high control	Enable/disable
xbusyin	→	active high control	Undelivered message(s) in network
receive*	→	active low control	Processor message delivery
send*	←	active low control	Initiate message send
xf*f*	→	instruction code	Select flag
xa*a*	→	instruction code	Operand/node address
xo*w*	→	instruction code	Opcode
di*d*	→	data	Router message data input
mi*p**	→	active low control	Router input message available
do*p*	←	data	Router message data output
mo*p*	←	active high control	Router output message available

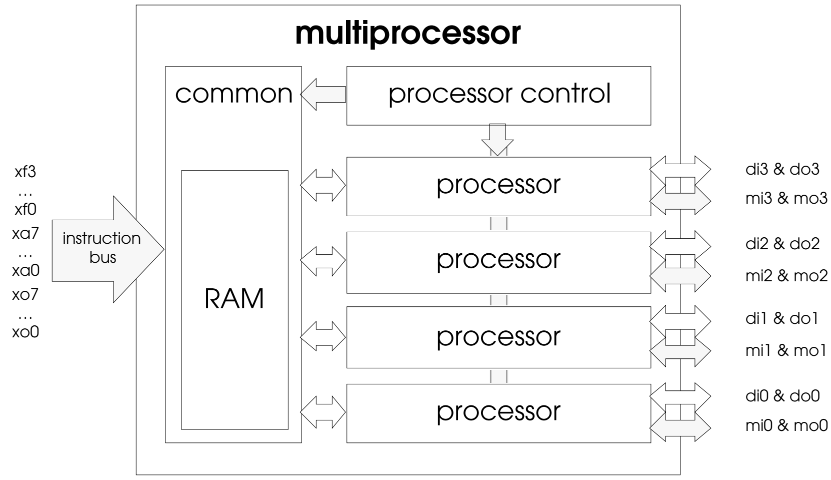
$f \in \{0, \ldots, 3\}$ specifies one of 16 flags

$a \in \{0, \ldots, 7\}$ specifies an eight bit operand/node address

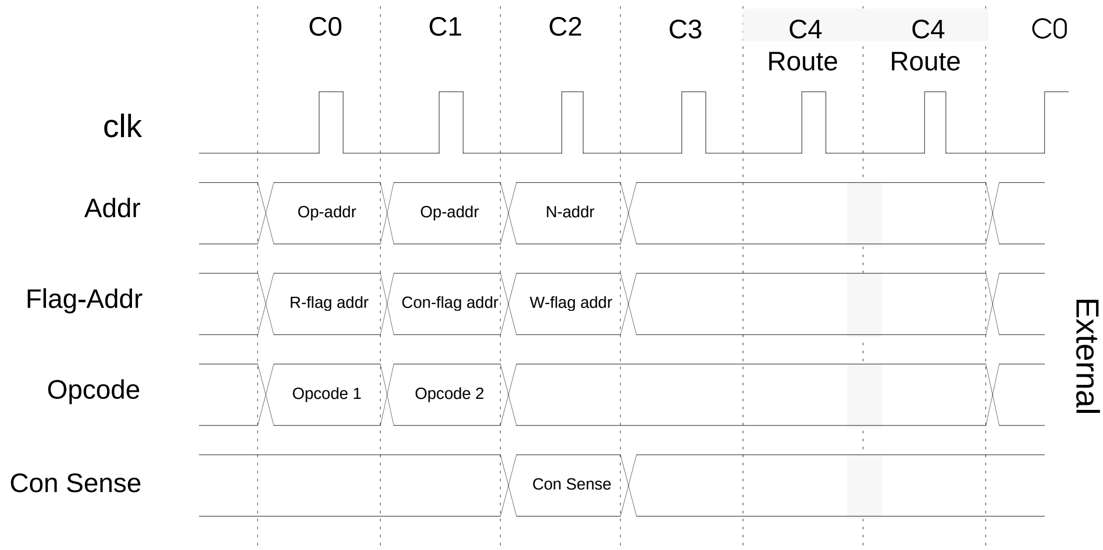
$w \in \{0, \ldots, 7\}$ specifies an eight bit opcode

$p \in \{0, \ldots, 3\}$ specifies one of the on-chip processors

Diagram



Timing



External timing

Implementation To optimize the usage of hardware real estate it is advantageous to have several processors on a single chip. The processors can share a single multiple-bit wide RAM, instruction decode ROM, single hypercube router and shared output pins for the microword bus.

Comments

With relatively little effort the fourth cycle (C3) and the first (C0) could be made to overlap, in effect pipelining the processor and improving its performance by 30 %.

processor_control

Function Controls operation of the processors on a Qbe*rt chip

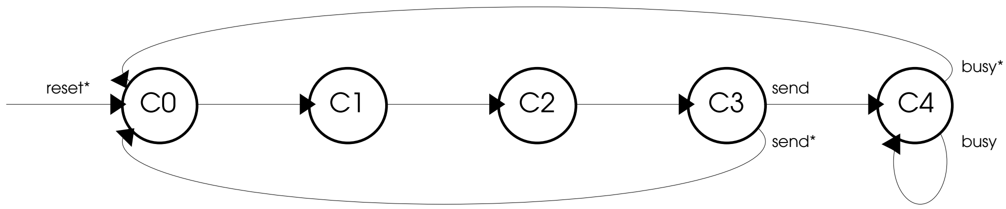
Use Within a microprocessor it generates the nanoinstructions for the local processors. It determines the sequencing of the register-to-register transfers within the processors and coordinates interaction with the router.

Uses None

Interface

clk	→	positive duty clock	Control clock
reset*	→	active low control	System reset
send	→	active high control	Start send cycle
busy	→	active high control	Undelivered message(s) in network
c0*	←	active low control	In C0 state
c1*	←	active low control	In C1 state
c2*	←	active low control	In C2 state
c3*	←	active low control	In C3 state
c4*	←	active low control	In C4 state

Diagram



Processor state diagram

Timing Since the processor control outputs are often used in other parts of the processor as enabling signals during clock pulses or transitions, the processor clock must have a different phase than the processor system clock. See the appendix for further details on processor timing.

Implementation The processor control is basically a state-transition engine with a token ring structure. The ring is implemented as a string of flip-flops, of which at any time only one contains the token (a logical 0), defining the state of the processor.

Comments

The state-transition engine, although not minimal in silicon usage, allows for easy modification of the controller. This was very useful during development, and should facilitate straightforward addition or removal of extraneous states if pipelining were incorporated in the processor. In particular, it is possible to have the first and fourth cycle of an instruction overlap.

common

Function The components (besides the router) that are shared by the four local processors

Use Instruction word and instruction decode hardware are shared between the local processors. The four local 256-bit times 1-bit wide data RAMs are combined to a 256*4-bit RAM so they can share the same address decode hardware.

Uses reg8

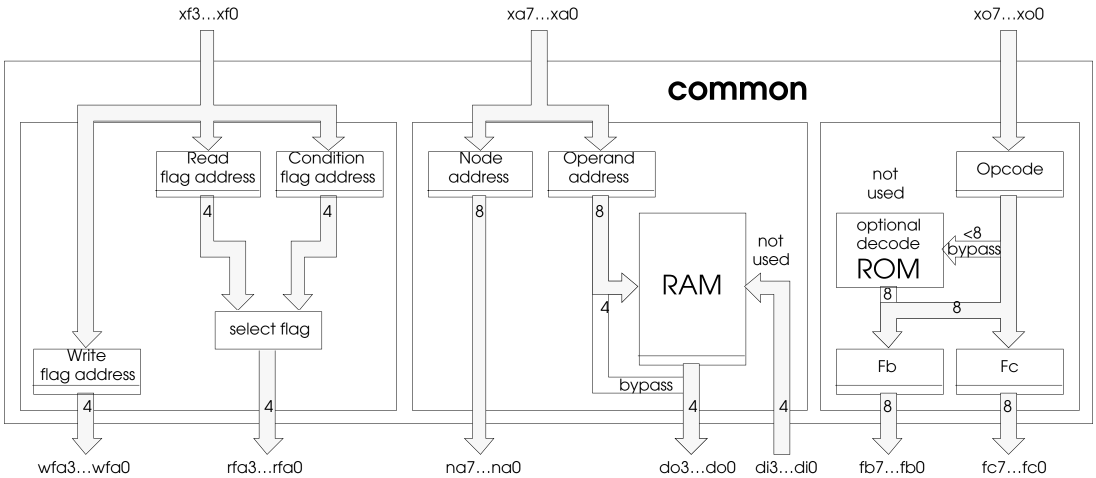
Interface

clk	→	positive duty clock	Data clock
reset*	→	active low control	Reset
xf*f*	→	data	Microword flag address field
xa*a*	→	data	Microword address field
xo*w*	→	data	Microword function (operation) code field
rfs*	→	active low control	Read-flag address strobe
wfs*	→	active low control	Write-flag address strobe

cfs*	→	active low control	Condition-flag address strobe
oas*	→	active low control	Operand address strobe
nas*	→	active low control	Message destination node address strobe
os*	→	active low control	Function (operation) code strobe
fbs*	→	active low control	B-result function code strobe
fcs*	→	active low control	C-result function code strobe
c*rfsel	→	control	Condition/Read-flag select
rfa*f*	←	data	Condition/Read-flag address
wfa*f*	←	data	Write-flag address
na*a*	←	data	Message destination node address
fb*w*	←	data	B-result function code
fc*w*	←	data	C-result function code
do*p*	←	data	A/B-operand output
di*p*	→	data	B-result feedback

$f \in \{0, \ldots, 3\}$ specifies one of 16 flags
$a \in \{0, \ldots, 7\}$ specifies an eight-bit operand/node address
$w \in \{0, \ldots, 7\}$ specifies an eight-bit opcode
$p \in \{0, \ldots, 3\}$ specifies one of the on-chip processors

Diagram



Timing None

Implementation During the implementation it turned out that we would have enough external pins available to use a full unencoded opcode, making the complete 65536-instruction set available and an instruction decoding ROM unnecessary.

Comments The RAM was not actually incorporated because it had to be especially generated at IMEC and would have required a stronger commitment to future testing of a processed chip than we were capable of making. As noted in the Observations, it could not have been implemented anyway. The RAM is not required for otherwise functional testing of the processor. The RAM size is 256 × 4 bits and has an access time of roughly 90 ns.

reg8

Function 8-bit parallel-in/out register

Use Among others, reg8s find use in storing operand and result addresses, and instruction codes.

Uses None

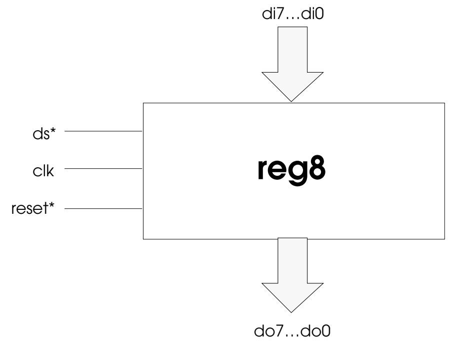
Interface

clk	→	positive duty clock	Data clock
reset*	→	active low control	Register reset

ds*	→	active low control	Data write strobe
di*d*	→	data	Write data
do*d*	←	data	Read data

d ∈ {0, …, 7} specifies the 8-bit data

Diagram



Timing None

Implementation None

Comments A reg8 part is used in some cases where fewer than 8 bits are actually required, mainly to minimize cell implementation and verification time. The space overhead is not considered significant.

processor

Function single-bit processor

Use An elementary processor forms a single node of a hypercube-topology computer, and one-fourth of the processing power of a single Qbe*rt chip.

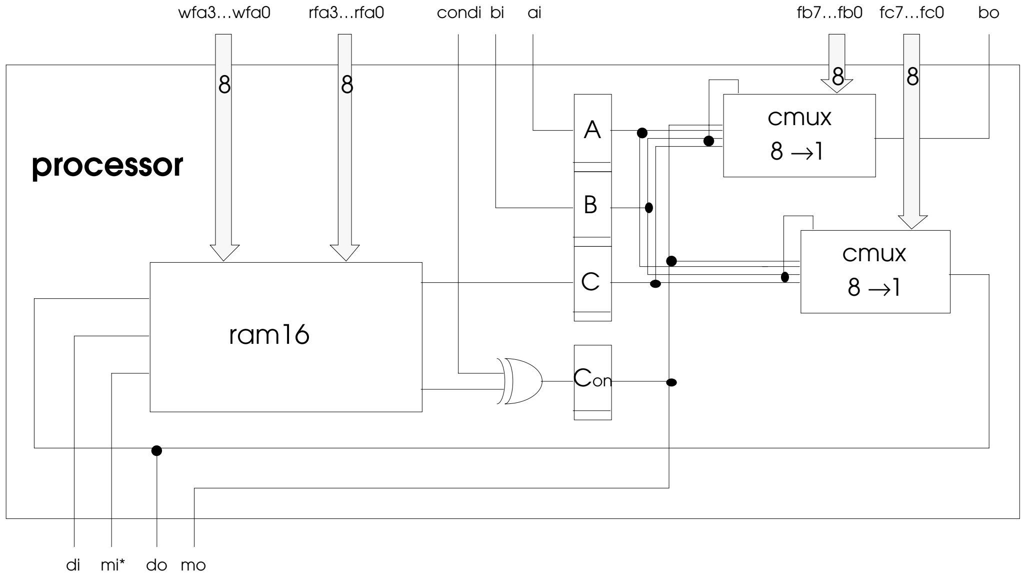
Uses ram16, cmux8to1

Interface

clk	→	positive duty clock	Data clock
reset*	→	active low control	Processor reset
condi	→	control	Send condition polarity
conds*	→	active low control	Condition-flag register strobe
ai	→	data	A-operand
as*	→	active low control	A-operand strobe
bi	→	data	B-operand
bs*	→	active low control	B-operand strobe
cs*	→	active low control	C-operand strobe
fb*w*	→	data	B-result function code
fc*w*	→	data	C-result function code
bo	←	data	B-result
wfa*f*	→	data	Write-flag address
wfs*	→	active low control	Write-flag address strobe
rfa*f*	→	data	Read-flag address
mi*	→	active low control	Message available from input
di	→	data	Message input data
mo	←	active high control	Message available for output
do	←	data	Message output data

f ∈ {0, …, 3} specifies the flag address
w ∈ {0, …, 7} specifies an eight bit opcode

Diagram


wfa3…wfa0
rfa3…rfa0
condi
bi
ai
fb7…fb0
fc7…fc0
bo
8
8
8
8
processor
A
B
C
Con
cmux
8 →1
cmux
8 →1
ram16
di
mi*
do
mo

Timing

Implementation The processor is made up of a very simple ALU with taking one-bit operands, combined with a local 16×1 flag register file. The ALU in fact has a very powerful instruction set because it can perform any one of the 65536 possible logical operations on its three operands for both output operands.

Any instruction can be made to send its result to another processor in the hypercube.

Comments None

ram16

Function 16-bit random access memory with special port

Use The ram16 is used to store 16 read, write or condition flags of a processor. Received messages use a special port into address 0001_2 so that they can be received asynchronously of the processor.

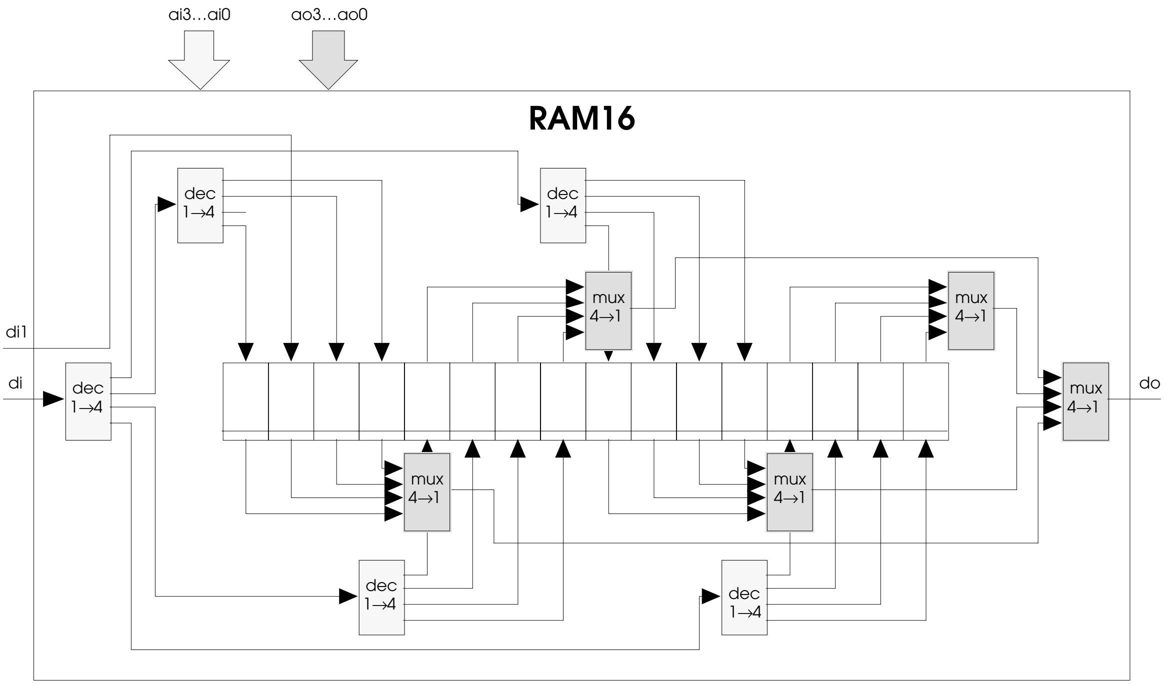
Uses None

Interface

clk	→	positive duty clock	Data clock
reset*	→	active low control	Reset
write*	→	active low control	Write data strobe
write1*	→	active low control	Write data (0001_2) strobe
ai*f*	→	address	Write data address
ao*f*	→	address	Read data address
di	→	data	Write data
di1	→	data	Write data (0001_2)
do	←	data	Read data

f ∈ {0, …, 3} specifies one out of 16 flags

Diagram



Timing None

Implementation The current ram16 has one read and two write ports. Read and write operations can be performed concurrently.

Comments Future versions of Qbe*rt processors employing instruction pipelining would require multiple read ports.

cmux8to1

Function 8-to-1 conditional multiplexer

Use Two cmux8to1 parts are used to select the appropriate values for the B_{out} and C_{out} results, with the A, B and C operands forming the three address bits. The two opcodes are used as input data.

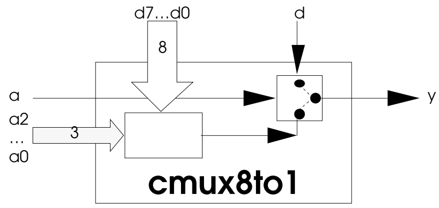
Uses None

Interface

a	→	active high control	Multiplex inhibit (y follows d instead)
as	→	address	Input address
d*w*	→	data	Muliplex data input
d	→	data	Multiplex inhibit data input
y	←	data	Output

s ∈ {0, …, 2} selects one out of eight inputs
w ∈ {0, …, 7} constitutes eight inputs

Diagram



Timing None

Implementation None

Comments The cmux8to1 part is in effect used as an elementary 3-to-1 bit ALU. It probably shows best one of the advantages of a reduced operand-width processor.

router

Function Hypercube message router and local distributor

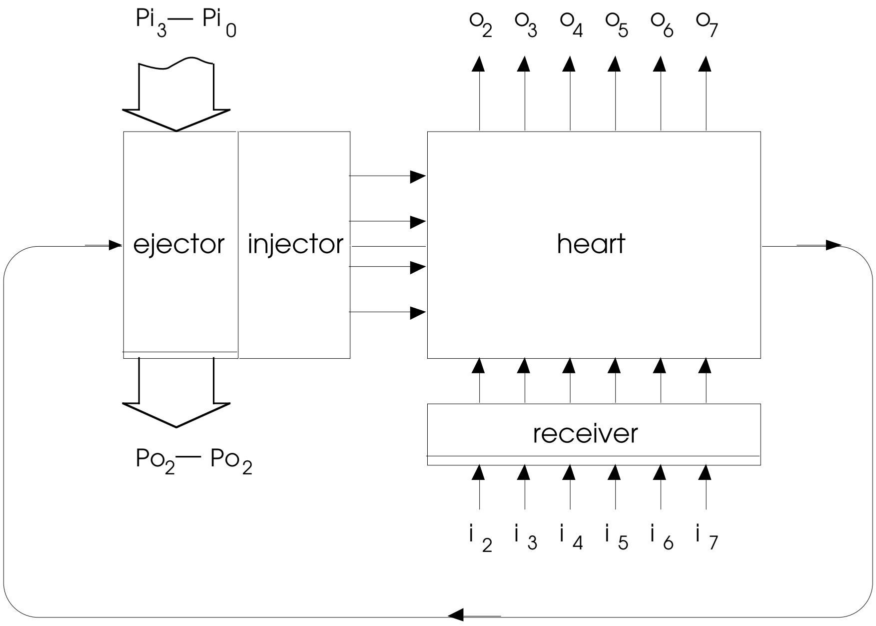
Use The router forms the core of the Qbe*rt chip, handling all of the communications needs between the $2^8 = 256$ processors in the network. One router, having separate inputs and outputs for each of the local processors, suffices for the entire chip. The router contains high-speed circuitry for assimilation, distribution and routing of local interprocessor messages without using the global network.

Uses router_control, ejector, receiver, heart

Interface

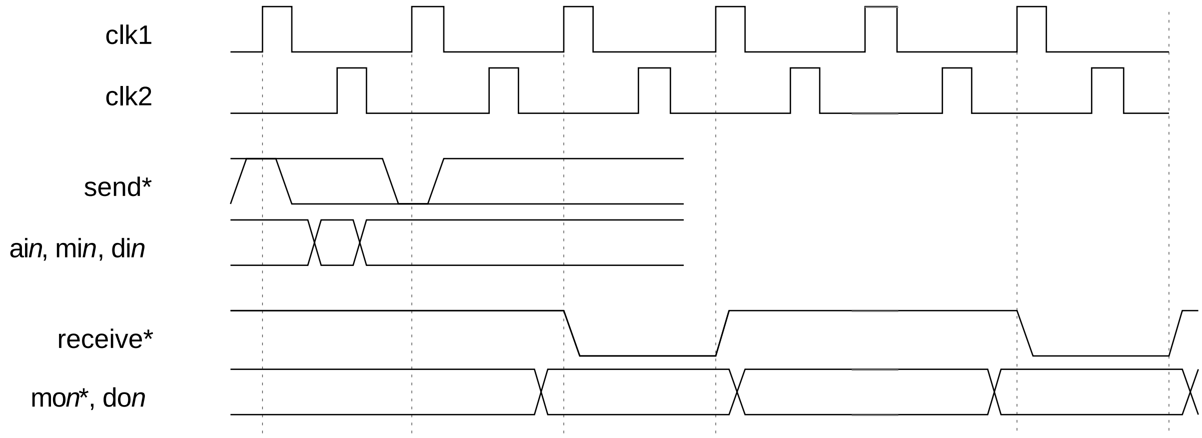
clk1	→	positive duty clock	Control clock (180° phase lead)
clk2	→	positive duty clock	Data clock
reset*	→	active low control	System reset
send*	→	active low control	Initiate message send
receive*	←	active low control	Processor message delivery
xbusyin	→	active high control	Undelivered messages in network
xbusyout	←	active high control	Undelivered messages in node
ai*n*	→	data	Message destination node relative address
di*n*	→	data	Processor message data input
mi*n*	→	active high data	Processor input message available
do*n*	←	data	Processor message data output
mo*n**	←	active low data	Processor output message available
xdi*n*	→	data	External dimension network data input
xdo*n*	←	data	External dimension network data output

Diagram



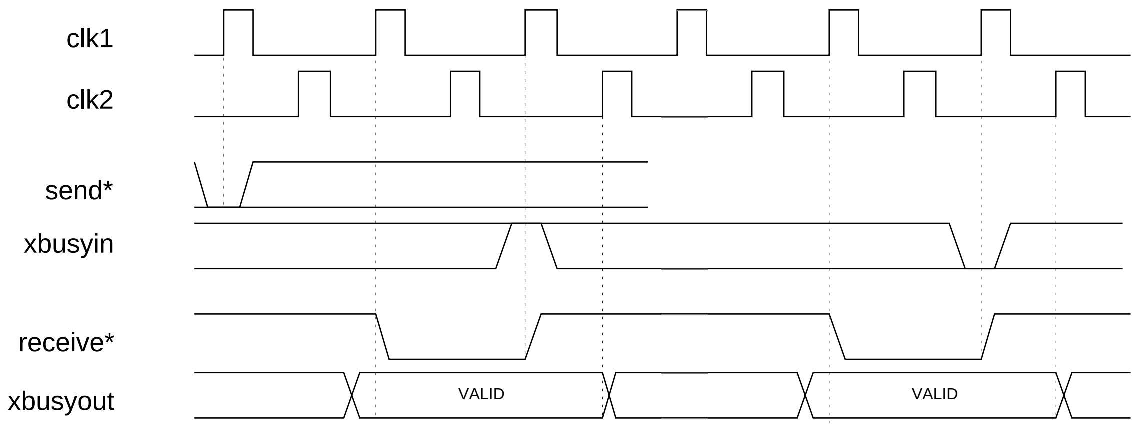
The injector accepts messages from the on-chip processors at the beginning of a send cycle. Any messages destined for the local processors are ejected immediately without having to endure a routing and transmission phase. Left-over messages, and messages received from external processors are routed through the heart to appropriate destinations.

Timing Since the router control outputs are often used in other parts of the router as enabling signals during clock pulses or transitions, the control clock must have a different phase than the router (data) clock.



Router/processor interface timing

As the timing diagram shows, a send phase is initiated by an active low send* signal which is received from the global controller. Because the router *preloads* messages, it is imperative that the message address, data and availability bits be present before the beginning of the cycle. During the send cycle, the receive* control signal may go low an undetermined number of times, depending on the number of messages that were sent and congestion on the network. It is the processor's responsibility to pick up any messages at these times. The presence of a message for a particular processor is indicated by an active message-out mo*n** signal.



Router/network interface timing

Because a send phase may involve several consecutive message send cycles, the the routers need to be able to signal to the controller when all messages have been delivered so that the microword of the next instruction may be fed to the ALU. The routers use the xbusyin and xbusyout signals to achieve this. While a router still contains undelivered messages it asserts the xbusyout signal. All these signals from each of the routers must be combined externally into a common signal which indicates the presence of undelivered messages in the entire network, and is fed back to the routers as the xbusyin signal.

As the diagram shows, while xbusyin is active a router will remain in the send phase so that another message send cycle is performed. It is important to realize that the routers on all of the chips remain synchronized, and therefore all execute the same number of send cycles even though some routers may not contain undelivered messages. However, they may receive new messages during such cycles.

Note that there is approximately one cycle time (100 ns) for xbusyin to be generated from the time that the xbusyout signals become valid. If this should prove to be not enough time, the clock cycle period could be lengthened or the routing state could be extended with additional clock periods.

Implementation Messages are either parallel-loaded into the ejector directly from the local processors during load cycles, or are shifted in from the heart during the shift phase. When new messages have arrived in the ejector in either way, it ejects and removes from itself all messages directed to one of its local processors.

Remaining messages and messages in the receiver are examined by the heart, which uses their availability bits and relative addresses to rearrange itself appropriately. Once the router has determined its proper state it is locked in, and the messages contained in the receiver and ejector are shifted through the heart. Depending on how it is switched, these messages are routed across the external dimensions or shifted into the ejector.

Comments None

router_control

Function Orchestration of operation and interaction of router subcomponents

Use A single router_control block is used in a router to control its operation.

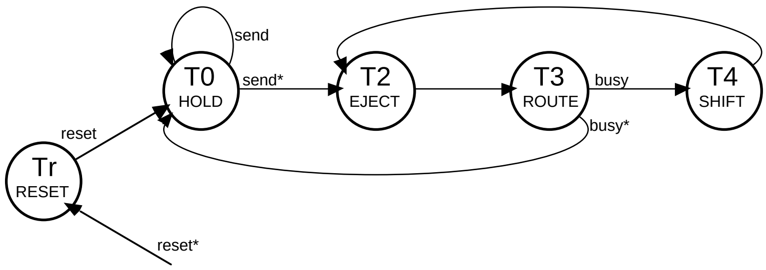
Uses None

Interface

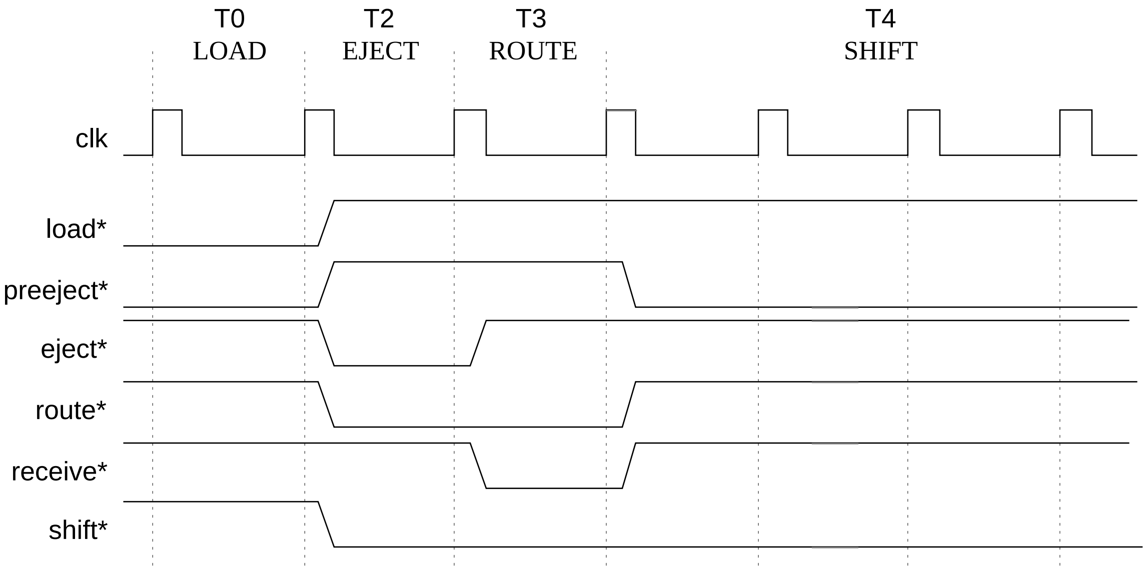
clk	→	positive duty clock	Control clock
reset*	→	active low control	State reset
send*	→	active low control	Initiate message send
busy	→	active high control	Undelivered messages in network

load*	←	active low control	Load messages from local processors
preeject*	←	active low control	Prepare for ejection (duration unspecified)
eject*	←	active low control	Eject messages to local processors
receive*	←	active low control	Received messages ready for local processors
route*	←	active low control	Compute message routing
shift*	←	active low control	Transmit serial messages

Diagram



Timing Since the router control outputs are often used in other parts of the router as enabling signals during clock pulses or transitions, the router clock must have a different phase than the router system clock.



Implementation The router control is basically a state-transition engine with a token ring structure. The ring is implemented as a string of flip-flops, of which at any time only one contains the token (a logical 0), defining the state of the router. The router is in its hold state (T0) after a reset.

The duration of the shift state (T4) is controlled by a 4-bit 10-counter. This corresponds to the length of the serial message, the number of bits that must be transmitted to complete a message.

Note: The explicit state T1 (load) was made obsolete. A load is now always performed in T0 (preload).

Comments The routing phase was extended to two consecutive cycles, the first of which overlaps with message ejection, after it was determined that the routing could not complete within one cycle. The ejectors were optimized to eject messages very rapidly (just after the clock in the ejection state) so that the correct message availability information is presented to the heart as early as possible. Therefore, actually only approximately one and a half of these two cycles (150 ns) are effectively useful. Still, this saved us from having to introduce a two-cycle routing state.

receiver

Function Receiver for messages arriving from external neighborly chips

Use A single receiver is used within the router for the collection of serial messages from other nodes of the hypercube

Uses Msg_buffer

Interface

clk	→	positive duty clock	Data clock
reset*	→	active low control	Buffer reset
reset0*	→	active low control	Message erase
shift*	→	active low control	Transmit serial messages
empty*	←	active low control	No messages in receiver
si*d*	→	data	Data input (external dimension d)
do*d*	←	active high control	Dimension d message available
	←	data	Data output (during shift phase)
ao*da*	←	data	Address output

$d \in \{0, \ldots, 5\}$ indicates the external dimension
$a \in \{2, \ldots, 7\}$ specifies a message external relative address bit

Diagram None

Timing None

Implementation During shifting, messages are received from the external neighbors and shifted into their respective message buffers. Simultaneously, messages previously present in the buffers are shifted out.

Comments

Although msg_buffer blocks are used as the primary component of a receiver because they were already available, most of their functionality is not used by the receiver.

ejector

Function Combined serial message injector, ejector and distributor to local processors

Use The ejector accepts messages from local processors and injects them into the external network message stream. Messages supplied to it by the processors and messages received from the network router heart destined for local processors are detected and ejected immediately, thus avoiding a costly global network message exchange when unnecessary. All other messages are (re-)injected into the network router heart.

Uses Ejector_part, funnel

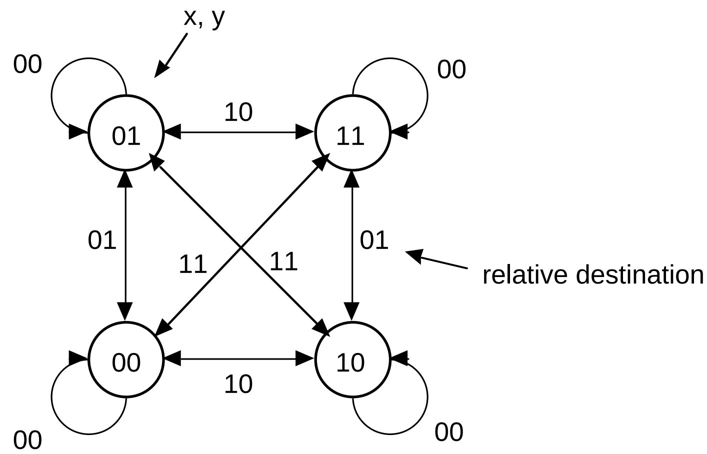
Interface

clk	→	positive duty clock	
load*	→	active low control	Load parallel data message word
shift*	→	active low control	Shift the data out
reset*	→	active low control	Ejector reset
preeject*	→	active low control	Prepare for ejection
eject*	→	active low control	Eject messages for local processors
empty*	←	active low control	Indicates if the ejector is empty
ai*a*	→	address	Input message relative destination address
mi*p*	→	active high control	Processor p message input available
di*p*	→	data	Processor p data input
si*e*	→	data	Ejector_part e shift input
mo*p**	←	active low control	Processor p message output available

do*p*	←	data	Processor *p* data output
so*e*	←	active high control	Ejector_part *e* message available
	←	data	Ejector_part *e* shift output (during shift phase)
ao*ea*	←	address	Ejector_part message relative address

$a \in \{0, \ldots, 7\}$ specifies a message external relative address bit
$e \in \{0, \ldots, 3\}$ specifies one of the ejector_parts
$p \in \{0, \ldots, 3\}$ specifies one of the on-chip processors

Diagram The ejector_parts generate signals specifying the *relative* destination on-chip processor. This implies that when, for instance, ejector_part 2 indicates relative processor 0, its message in fact has as its destination local processor 2, *not* processor 0. The diagram shows how the absolute destination address *y* of a message in ejector_part *x* can be derived from its relative address.



Ejector_part *x* to processor *y*, with relative destinations

Timing The timing of the ejector is essentially the same as that of the individual ejector_parts (see ejector_part).

Implementation Multiple messages for a single local processor are simultaneously ejected and funnelled to produce a single combined message which is passed to the processor.

The ejector provides four address buses from the corresponding ejector_parts to the heart together with their shift data outputs, which are used for routing the heart. Messages destined for one of the local processors are ejected, which involves clearing the message availability bit (thus changing the shift output). This ensures that such messages do not influence heart routing.

The ejector generates an empty* signal which indicates if no messages are present in the ejector. This signal is valid after message ejection, triggered by an eject*, which changes the corresponding message-available bits.

Comments

ejector_part

Function Ejecting messages for the on-chip processors and determining which processor is targeted.

Use Four ejector_part blocks are used in an ejector for the simultaneous reception and ejection of messages to as many local processors, as well as the injection into the global message stream.

Uses Msg_buffer

Interface

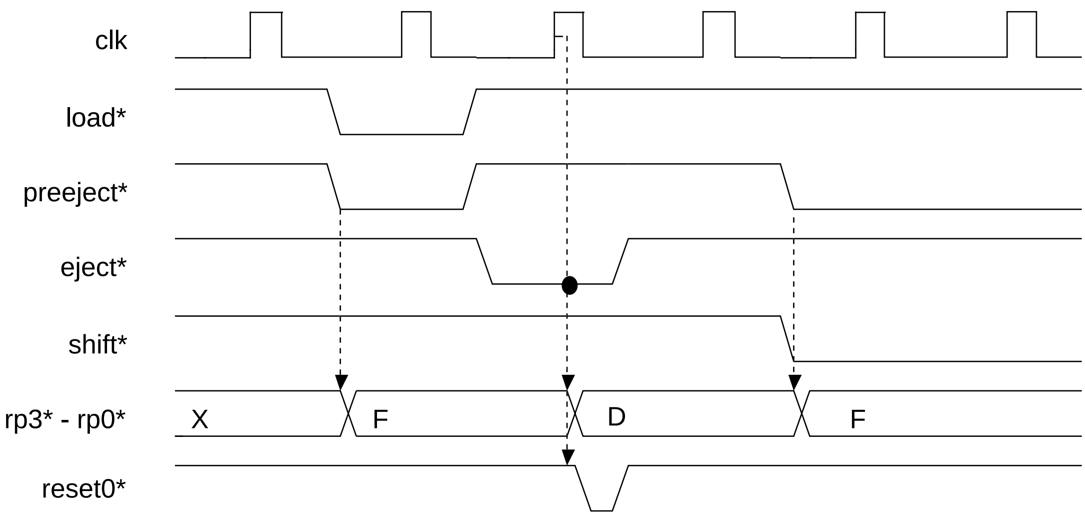
clk	→	positive duty clock

reset*	→	active low control	Message reset
load*	→	active low control	Message load
shift*	→	active low control	Message shift
preeject*	→	active low control	Prepare for ejection
eject*	→	active low control	Eject message for local processor
mi	→	active high control	Parallel load message available
di	→	data	Parallel load message data bit
ai*a*	→	address	Parallel load message destination address
si	→	data	Serial message shift input
msg*	←	active low control	Message present
do	←	data	Parallel eject message data bit
ao*a*	←	address	Parallel output message external destination address
so	←	active high control	Message present
	←	data	Serial message shift output (during shift phase)
rp*p**	←	active low control	Message ejected for processor *p*

$a \in \{0, \ldots, 7\}$ specifies a message external relative address bit
$p \in \{0, \ldots, 3\}$ specifies one of the on-chip processors

Diagram None

Timing In the timing diagram processor 1 was arbitrarily taken as the destination processor.



Implementation In ejector_part the six most significant bits of the destination address of the message word are fed into a logical OR. During ejection (eject* active), if there is a message available (q0* of the msg_buffer is low) whose destination is an on-chip processor (all bits are zero), the flip-flop output is reset at the positive transition of the clock. This generates a short low pulse on the msg_buffer's asynchronous reset0* input, resetting the message available bit.

The flip-flop is preset by preeject* before ejection takes place, to ensure that only negative transitions of its output occur. The flip-flop's function is to continue to indicate the ejection of a message after its available bit has been reset.

One of four signals specifying the relative on-chip destination processor address is made low, selected by the two least significant bits of the message destination address.

Comments Originally, the message available bit was not cleared until the next cycle. Due to routing timing constraints elaborated on later, it was desirable to do this as quickly as possible.

funnel

Function Multiplexing of data from multiple sources to a single output

Use Combining ejected data from the four ejector_part blocks of the ejector to a single input to the processor.

Uses None

Interface

px*p**	→	active low	Data input present from source *p*
d*p*	→	data	Data input from source *p*
px*	←	active low	Data output present
dx	←	data	Data output

$p \in \{0, \ldots, 3\}$ specifies one of four data sources

Diagram None

Timing None

Implementation Multiple data which is presented to the funnel is combined by means of a logical AND. The data output present signal is asserted when any data is passed through the funnel.

Comments

In future, rather than combining multiple data through a logical AND, any particular logical function combination might be selected by specifying an function selector.

msg_buffer

Function 10-bit serial shift register with parallel load and output. Independent asynchronous reset of the lowest-order data bit.

Use Msg_buffer is used as the core of an ejector_part, in which the message word is both parallel loaded/ejected and serially shifted in/out—in the case of the receiver, only the latter function is actually used.

Uses None

Interface

clk	→	positive duty clock	
reset*	→	active low control	Reset data word
reset0*	→	active low control	Reset lowest-order bit of data word
load*	→	active low control	Load parallel data
shift*	→	active low control	Shift right
si	→	data	Shift input
d*i*	→	data	Data bus input
so	←	data	Shift output
q*i*	←	data	Data bus output
q0*	←	inverted data	Lowest-order data bit output

$i \in \{0, \ldots, 9\}$ specifies bits of the data word

Diagram None

Timing None

Implementation The input of each flip-flop i (storing bit i of the data word) is the output of a four-input multiplexer i. Three inputs (one is unused) of the multiplexer are, respectively, the output of the previous higher-order bit flip-flop i+1, the parallel-load data bit di and the output of flip-flop i itself. Each of these inputs is selected by an particular combination of shift* and load*. If both these signals are inactive the stored data is unchanged.

Comments The flip-flops are positive edge-triggered, but unfortunately not of the master-slave type—i.e. at the positive edge of the clock the input immediately appears at the output, which often gives rise to hold-time violations during simulation.

The multiplexer has an inverting output. To preserve polarity its inputs are inverted as well, i.e. q*i**, q*i*+1*, d*i** (with q*i** being the the inverting output of flip-flop i).

The behavior is undefined if load* and shift* are both low at a positive clock edge.

heart

Function Routing of messages between internal processors and external neighbors.

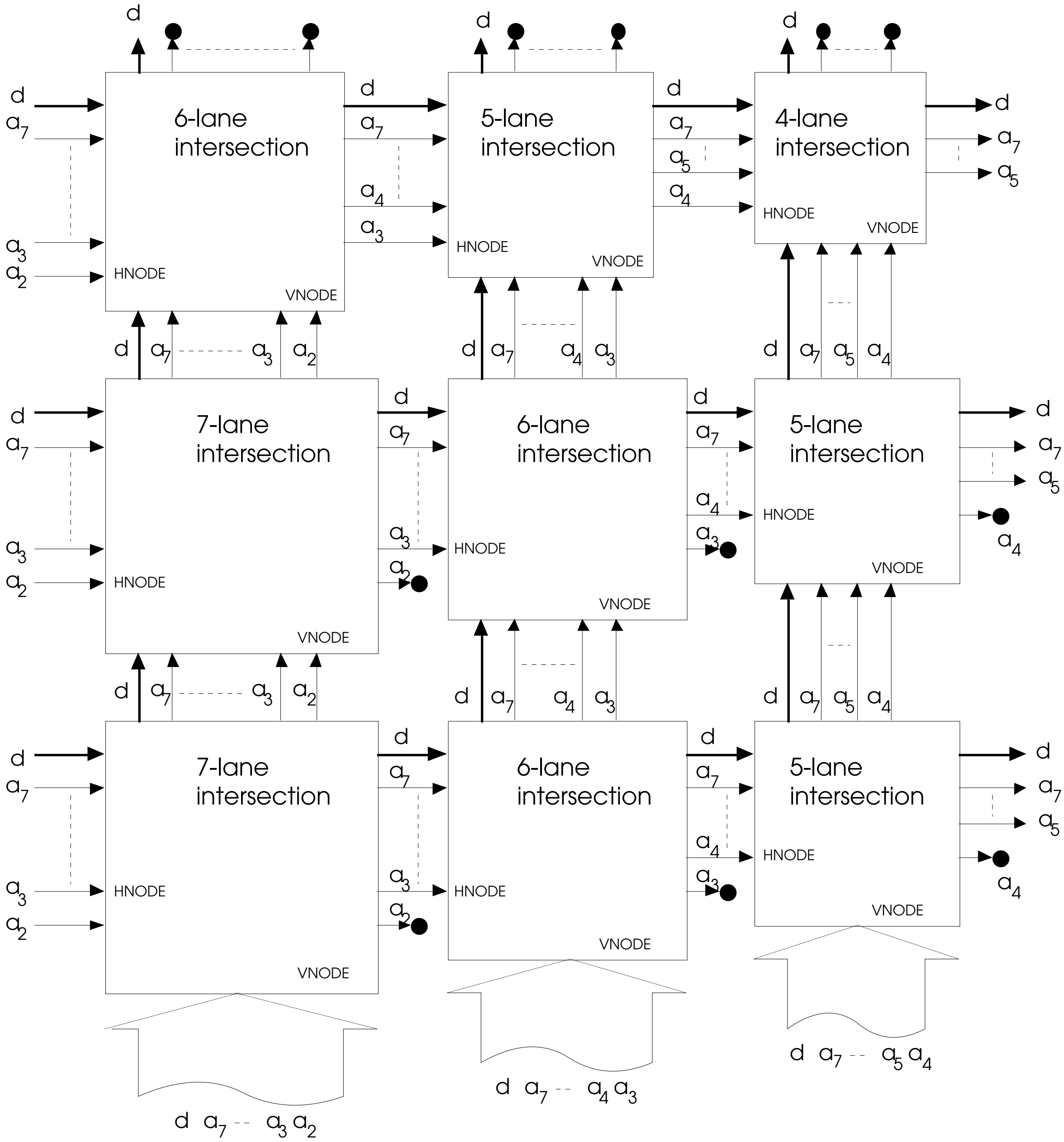
Use The heart forms the core of the router. Messages which have been loaded or shifted into the ejector, and messages which were received from external neighbors are routed by the heart to their proper destinations as well as possible.

Uses cell

Interface

dw*e*	→	active high control	(route phase) Message present from ejector e
	→	data	(shift phase) Bit-serial message from ejector e
ds*d*	→	active high control	(route phase) Message present from receiver of dimension d
	→	data	(shift phase) Bit-serial message from receiver d
de*e*	←	data	(shift phase) Bit-serial message to ejector e
dn*d*	←	data	(shift phase) Bit-serial message to neighbor of dimension d
aw*ea*	→	active high data	Relative address bit a of message from ejector e
as*da*	→	active high data	Relative address bit a of message from receiver of dimension d
route*	→	active low control	Route phase

Diagram



Timing Simulation shows heart routing times of up to 120 ns, which is longer than one cycle at the target clock frequency of 10 MHz.

Implementation The heart is certainly one of the most conceptually complex yet fundamental parts of the Qbe*rt processor. There are two modes of operation: in the route phase (defined by an active route*), stable messages at the inputs are used to determine their destinations when they are serially shifted through during the shift phase.

It is clear that since individual cells do not create or destroy messages, and interconnections between cells are one-to-one, that the entire heart does not create or destroy messages. This is an important result since it means that messages always end up *somewhere.*

The heart routing algorithm can be induced from the individual cell switching algorithm. Generally, however, the heart can be said to route messages approximately according to the following rules:

- Messages may be prioritized according to lowest-to-highest numbered ejector followed by highest-to-lowest ordered dimensional receiver;

- The highest-priority message that wants to be routed across an external dimension is routed across the highest dimension that it needs to cross. The next lower-priority message that wants to cross the same dimension takes the place and hence acquires the priority of its contender;
- Messages from receivers are only routed across same or lower-ordered dimensions, or to an ejector;
- The highest-priority message that does not want to be routed across any external dimension is routed to an ejector of equal or higher priority;
- Messages which are trying to reach an ejector are *misrouted* across a dimension they do not need to travel if all ejectors are taken by higher-priority messages.

Misrouting may be seen as a last-ditch effort by the heart to route the message *anywhere* at all when all paths to its proper destination are already taken by other messages.

It can be verified that the cells used here do in fact implement these rules. The data bits of the cells are used to switch the message-available and the address bits in parallel, in effect switching the messages' relative addresses while at the same time creating a data path for the subsequent transmission of data.

Note the strict causality in the routing process. The lower left cell (cell 35) is the first to find its correct orientation. Although its neighboring cells (25 and 34) will start routing simultaneously, they will not be able to find their correct orientations until 35 has done so, because their orientation depends on which message is routed to their input by cell 35. There is a frantic flurry of activity in the early stages of routing as distant cells try continuously to adapt to the changing messages appearing at their inputs†. Gradually the heart settles as the steady-state condition propagates to its farthest reaches. At this point route* may be made inactive, locking the heart in its orientation.

Comments The routing algorithm used is not optimal, but it ensures that no message is ever lost and always eventually reaches its destination. For instance, note that messages can fortuitously acquire much greater priorities when they are redirected in favor of another message.

As a matter of interest—an unrouted version of the heart showed an occupation of 11.2 mm^2, or 7.76% of the chip surface.

We also investiagated using a combinatorial tree-like structure in which the entire heart is set in parallel. It quickly became obvious that this would have required an unacceptable extent of silicon area. A hybrid scheme was also considered but was not found to be practical.

We had originally envisaged a scheme whereby the message bits were clocked through the heart by means of a shift register-like structure. Unfortunately, this implies that since the distance that a message travels through the heart is variable and generally unpredictable, the messages do not enter or exit the heart simultaneously but are delayed with respect to one another.

cell

Function Cross-dimensional routing cell for interprocessor messages

† If don't care values (X) are specified for address bits which really are irrelevant to the eventual state, simulation produces an incredible amount of “no data available” warnings as the ‘don't care’ messages occasionally leak through when parts of the heart are still in a transitional state

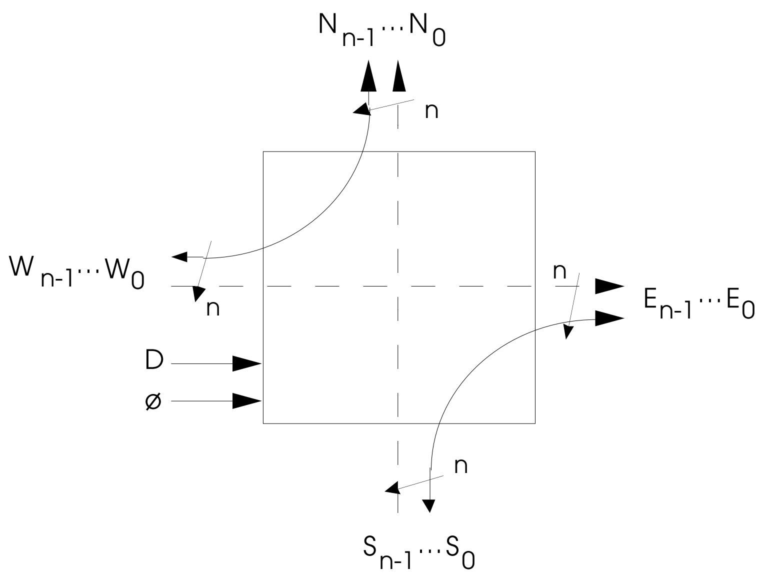
Use Within the heart, during the routing phase a cell examines the two messages and their relative destinations which enter it from the west and the south and determines how the messages should be switched—horizontally (west to east and south to north) or diagonally (west to north and south to east). During the data shifting phase the cell freezes this orientation.

Uses Switch

Interface

wmsg	→	active high data	West message present
smsg	→	active high data	South message present
emsg	←	active high data	East message present
nmsg	←	active high data	North message present
wnode	→	active high data	West message directional preference
snode	→	active high data	South message directional preference
enode	←	active high data	East message directional preference
nnode	←	active high data	North message directional preference
w4-w0	→	data	West data input
s4-s0	→	data	South data input
e4-e0	←	data	East data output
n4-n0	←	data	North data output
switch*	→	active low control	While active, the cell changes orientation according to the messages input to it. When inactive, the orientation is held

Diagram



Timing None

Implementation

The cell switch algorithm is as follows. If there is a message from the west, its routing preference is accorded priority. Otherwise, the cell is switched according to the message from the south (even if there is no message from the south, in which case the orientation of the cell is a *don't care*).

The routing preference of a message is specified by its node relative address bit. A logical 1 in an address bit of a message indicates that the message still needs to travel across the corresponding dimension, so the message will want to be routed towards the north. Conversely, a 0 indicates the message has already reached the desired hypercube coordinate of that dimension.

The word “preference” is used here purposely. It is very much possible that a message may not be routed in the direction in which it needs to go, in particular when a higher-priority message needs to be routed in the same direction.

Comments Fan-in of data and control inputs is double normal gate load.

switch

Function Horizontal/diagonal cross-switch of one-bit data between two inputs and outputs

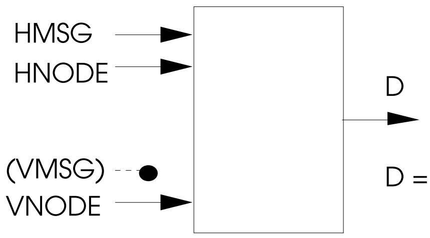
Use Many switches are combined into a *cell*, which routes many bits of data simultaneously through the heart.

Uses None

Interface

h	→	control	Logical 1 switches cell horizontally/ vertically, logical 0 switches cell diagonally
h*	←	control	Should be inverse of h. Switching characteristics are undefined if not
w	→	data	West data input
s	→	data	South data input
e	←	data	East data output. Follows west input while h is 1, south input while h is 0
n	←	data	North data output. Follows south input while h is 1, west input while h is 0

Diagram



$$D = (HMSG \wedge HNODE) \vee (\overline{HMSG} \wedge \overline{VNODE})$$

Timing None

Implementation This block cross-switches two data inputs to two outputs, depending on the value of the h and h* inputs.

Comments Fan-in of data and control inputs is double normal gate load.

4 References and Appendix

[1] Charles E. Leiserson, *Advanced Parallel and VLSI Computation,* lecture notes

[2] W. Daniel Hillis, *The Connection Machine*, Scientific American, June 1987, pp. 86-93

[3] W. Daniel Hillis and Guy L. Steele Jr., *Data Parallel Algorithms*, Communications of the ACM, December 1986, Volume 2 Number 12, pp. 1170-1183

[4] Blaauw, *Digital System Implementation*, Prentice-Hall

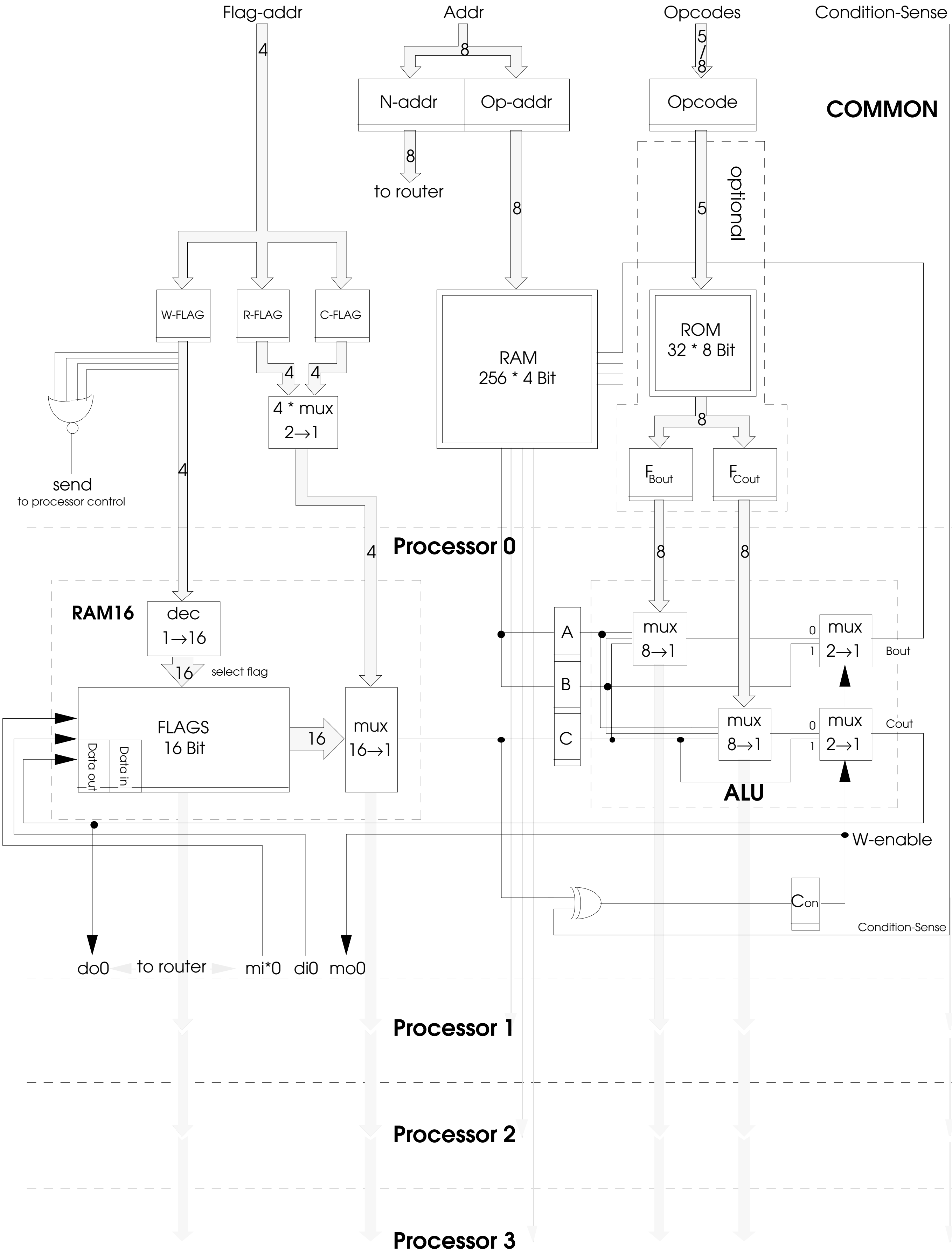
[5] Ware Myers, *Faster…*, Computer, July 1991, pp. 96

[6] H.G. Sachs, H. McGhan, L.F. Hanson, N.A. Brookwood, *Design and Implementation Trade-offs in the Clipper C400 Architecture*, IEEE Micro, June 1991, pp. 18-21, 74-79

[7] O.A. McBryan, *The Connection Machine: PDE solution on 65536 processors*, Parallel Computing, 9, 1988-1989, pp. 1-24

[8] Neil Weste and Kamran Eshraghian, *Principles of CMOS VLSI Design*, 1985-1988

[9] Peter F.A. Middelhoek, *Timing and Placement in the C400 FPU*, Internal report Intergraph Corporation Advanced Processor Division


Flag-addr
Addr
Opcodes
Condition-Sense
N-addr
Op-addr
to router
Opcode
COMMON
optional
W-FLAG
R-FLAG
C-FLAG
RAM
256 * 4 Bit
ROM
32 * 8 Bit
4 * mux
2→1
send
to processor control
F Bout
F Cout
Processor 0
RAM16
dec
1→16
select flag
FLAGS
16 Bit
Data out
Data in
mux
16→1
A
B
C
mux
8→1
mux
2→1
Bout
Cout
ALU
W-enable
Con
Condition-Sense
do0
to router
mi*0
di0
mo0
Processor 1
Processor 2
Processor 3

Index