Plurion[1]: an Object Oriented Microprocessor

Jecel Mattos de Assumpção Jr.

[1] Plurion is a trademark of Merlintec Computers and is being registered in process 823954170 at INPI

[2] Everything described here is subject to change. Please check the first date in Appendix A (Document History) to make sure that you are reading the latest version of this document.

Abstract

A microprocessor for the next billion computer users should run advanced new software at the lowest possible cost and power, which can be achieved when backwards compatibility is not a requirement. A set of simple processors is a far better use of large transistor budgets than trying to make a single thread be as fast as possible. Direct support for object oriented programming can speed up advanced software at a very small cost.

Chapter 1

Introduction

The Plurion Architecture must balance two different goals:

- it must outperform traditional architectures by a large enough factor to make it worth investing in a custom design rather than a software solution on an existing processor
- it must be understandable to an advanced Smalltalk programmer. That is, it must be the next logical step in this progression:

1. using the graphical user interface
2. simple scripting
3. applications in Smalltalk
4. system programming in Smalltalk

This text is an attempt to make understanding how Plurion works the natural step "5". A more dynamic presentation would certainly yield better results, but the direct mapping of the implementation to high level concepts is the key to making this work.

1.1 Why Smalltalk?

Programming languages are tools, and like all tools each one is better for some tasks than for others. For the particular job of creating simulations in computers (of both real things like the solar system and of abstract things like a spreadsheet) the various object-oriented programming languages have been very successful. Though Smalltalk was the first to be entirely based on objects, it has not been the most widely used among this group. So the question which is the title of this subsection seems obvious.

But the real question which should be asked is "why not Smalltalk?" It is very rare for people familiar with several OO (object-oriented) languages including Smalltalk not to have it as their favorite. In the past they often had to use other options for

different jobs due to lack of hardware resources, though that is no longer a good reason. Sometimes the massive marketing for some language or other causes clients or bosses to demand the use of the latest fad. But given the choice, Smalltalk gives the best results.

There are good technical reasons for this: simplicity, dynamic environment, reflection, blocks, dynamic types and many others. An interesting trend can be observed in programming language design over the past two decades. Some designer finds the ideas in Smalltalk interesting, but feels that the language itself is too large to be practical. So they create something smaller instead that makes the programmers put up with all kinds of limitations. Unfortunately, simplicity and compactness don't last very long as trying to implement real applications (instead of the simple examples used when doing the initial design) forces the inclusion of more and more features. By the time the language is anywhere close to doing what Smalltalk does it is far larger and yet retains most of the limitations of the first version. A good example of this is the relation between Self (a Smalltalk dialect) and Java at Sun Microsystems.

1.2 Why a new processor?

Given that Smalltalk has now become practical on all reasonable computers due to the increased memory and processor speeds, it might seem pointless to develop a new processor optimized specifically for it. Generic processors and software implementation tricks should be more than enough and the failed language specific machines a thing of the past, right?

Wrong! There is no such thing as a generic processor. Every one is optimized for a specific task. This can be easily seen in the history of Intel's x86. The very first version, the 8086, was optimized for both hand written assembly language programs and also for applications compiled from Pascal sources. The next design, the iAPX286, added features to support advanced capability based operating systems in the tradition of Multics. By the time the 386 was released the future obviously belonged to the C language and the Unix operating system and the processor was radically re-architected to handle them as well as its competitors (Motorola 680X0, National 32000 and early RISCs) did. For full backwards compatibility, even the latest Pentium 4 still includes the frame instructions from the Pascal days and the advanced segments from the Multics-like days. Just like it still has the BCD math instructions from its 4004 calculator days. But no modern software uses them and so current designers don't put any effort into making them work well. If someone did try to make software that used these features it would not be efficient compared to C/Unix and, sadly, nobody would find this strange.

1.3 Free Hardware

Technology is a cultural construct - it is about people and environments and not about machines. If a company fires its ten engineers and then hires ten new ones to replace them, it won't take very long for it to find out that most of what it considered its technology has just walked out the door. Even if the previous team did a great job of

documenting as much as possible, it can never compare to what is in people's heads. It is like a group of archaeologists digging up an extensive library from some ancient civilization: the information helps but that culture is still dead.

The environment is the other needed ingredient. Artifacts such as machines, programs and texts are important. If the ten original engineers from our imaginary company arrive for work one day and find that a fire has destroyed all such material, they will take a long time to recover what was lost. This scenario isn't as bad as the first, but both people and things are essential to keep a culture alive.

It is possible to build a computer to be distributed to the next billion users while staying apart from these users. But that would waste a lot of potential. A far better alternative is for the developers and users to form a single community, which is only possible if there are no secrets or restrictions. A design meant to be understandable to advanced programmers is pointless if they are not allowed to see it. So all materials related to Plurion will be available to anyone who is interested. To balance the needs of a commercial project with the interests mentioned here, all of the information for a given version of the system will be made available under a very free (MIT-style) license at the same time that the next version becomes available commercially. Since the older versions will be better for learning than the following ones, this will actually help the users even more than it might initially seem.

1.4 Project Style

When approaching a problem from outside, all kinds of simple solutions seem obvious. Reality is far more complicated up close. That is why people hired as coaches seem stupid to those watching from the stands or from their homes and why voters can't understand how politicians seem to lose 20 IQ points once elected.

The simple and obvious solutions normally are just one big mess, with complex entanglements between their fuzzily defined parts. So a sign of project maturity is modularity. Dividing the system into isolated blocks, each of which can be separately understood by a person. "People sized chunks". The idea is that if you understand each part and understand how they are put together, then you know everything there is to know.

One idea that goes against this is "synergy", where the whole is more than the sum of its parts. This is avoided in most projects due to the impression that it is a step back into the bad, tangled design. Another issue is that modular designs can be created by groups of people while synergetic designs are almost always a product of a single mind. In theory, at least, it is possible to evolve modular designs by changing one component at a time since this is not supposed to affect the others.

The great advantage of a synergetic design is that it is smaller and simpler than an equivalent modular one. And though the initial impression is that it is harder to learn ("you have to understand everything before you understand anything") that is not actually the case. It does take longer for things to "click", but once they do it is possible to have the entire design in your mind instead of just one part at a time as with modules.

A mix of the two styles was adopted for Plurion. Where possible, things were kept isolated in layers or modules. But where great advantages in terms of size could be

gained by binding different parts tightly together, there was no hesitation to do so.

It is important to have a very clear definition for *simplicity* since that is a major goal. At least three alternatives could be considered:

absolute simplicity is obtained when nothing more can be eliminated while still having a working system. An example would be a paint application where one key changes the color of the pixel under the cursor and a second key moves the cursor to the next pixel in some predefined order. Any picture that can be made with the most sophisticated program can also, in theory, be made with this one. If any of the two keys were eliminated then this would no longer be true. A second example would be a programming system with a completely static memory allocation scheme. This was the case for Occam.

practical simplicity avoids eliminating elements which have alternatives in theory but which save a lot of effort in practice. The original MacPaint would be a good example of this. Simple tools combined with different brushes to allow many interesting patterns to be easily created. The stack-like memory allocation schemes of Forth or sbrk() in Unix are another example.

total simplicity is based on the idea that making one part of a system simpler normally causes some other part to become more complex. Yet the number of times that the different parts must be written varies. Many applications are created to run on a single operating system, for example, so shifting complexity to the application means a far larger total effort. A garbage collector is much more complex than stack-style memory allocation but the applications that use it are simpler.

For this project the goal of simplicity is the version described as *total simplicity* above. This means that the lower levels of the system, which are used over and over, tend to have a very advanced design in order to support smaller and simpler higher levels.

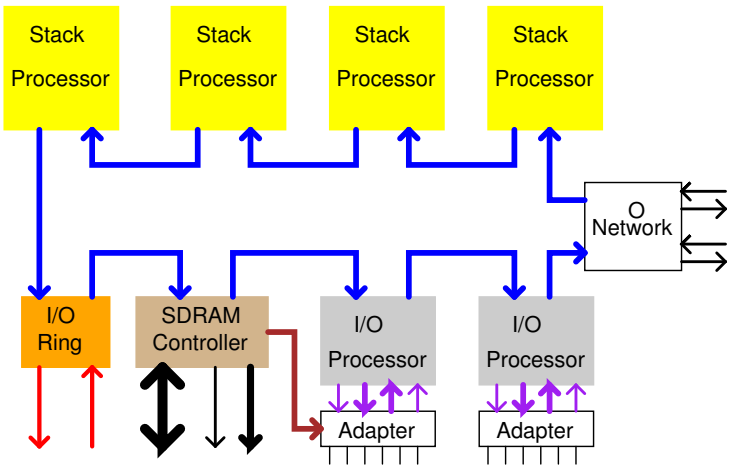
Chapter 2

Architecture

2.1 Interprocessor Connections

The basic elements of Plurion can be seen in the following figure. This design is recommended for four or less Stack Processors, though it can actually be used for any number.

Plurion: small architecture



The most complex component is the Stack Processor (SP). Most of this document is dedicated to the various details about this block, which is responsible for executing the application and system code. The name "Plurion" is meant to invoke the idea of a plurality of processors.

Another important part is the I/O Processor (IOP). It is far simpler than the Stack Processor and can only execute very short code fragments from its internal memory. In

slower implementations each IOP is connected to an adapter circuit (which is normally different for each one) which handles sub-microsecond changes in the interface signals, while the software running in the IOP handles changes under a millisecond. For very high frequency implementations the software can do everything that is needed and the adapter can be eliminated. Higher level and less time critical functions are handled by the main software running on the SPs. Thanks to the IOP this software can have a very abstract view of the external world for the peripherals will seem to be very smart and work in terms of objects and messages.

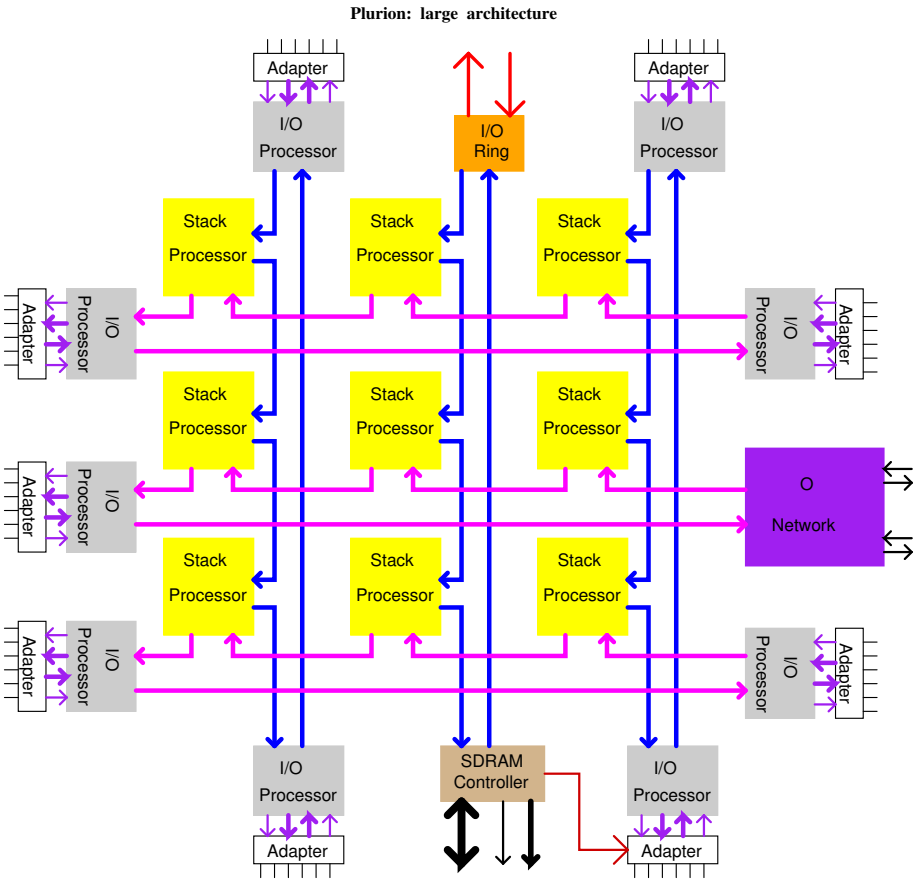
The SDRAM controller allows conventional memory chips to be used to build systems. Though the figure shows only one such block, any particular implementation can have several if needed. A direct connection is shown between the memory controller and the adaptor for one of the IOPs. This is for video output and avoids having such traffic occupy the internal network.

The connection between these blocks is in the form of a unidirectional "register insertion" ring network. This means that in normal operation each block repeats its input on its output with a single clock delay. When a node needs to transmit a command or data to another, it simply inserts a register (actually a FIFO memory) initialized to the contents it wants to send. Now the output represents the input delayed by a number of clock cycles. We can say that the ring has grown by a certain number of words. If the node needs to send something else it can't because its register is busy, so we have a distributed arbitration scheme. On any given cycle, a node can indicate that it has nothing to send using the *idle* signal. The node receiving the message doesn't repeat it but indicates idle instead. When a node outputs some data from its register but has idle in the input, then the amount of data in the register will decrease by one word. Eventually it will be entirely empty and the node will be free to send a new message.

One or more I/O Ring blocks extend the internal network to an external unidirectional ring where additional IOPs implement specialized interfaces. Normally this is done in the form of small expansion boards.

The last block shown in the figure is the O Network. It is basically an extension of the internal ring network to inter node communication. It uses the same register insertion architecture, though the ring is bidirectional unlike the internal network. Most implementations use just one O Network block but it is possible to have a chip with nothing but a number of such blocks. That would be a router and is very interesting for building large configurations of parallel computers using Plurion nodes.

An alternative interconnection scheme is recommended when nine or more SPs are used:



This uses the fact that each SP can have either one (as in the first figure) ring interface or two (as shown here). When a single SP is connected to two networks, it can pass messages between them. It only accepts such messages when it is free to transmit on the other network, so a message destined for another ring can circulate a few times on the current one if the SP in the intersection of the two has recently transmitted something. This is also the scheme that the O Network block uses to transfer messages between the internal and external rings, though it has far larger buffers than the ones inside the SP interfaces and so can more easily deal with busy networks.

2.2 Configuration Options

The architecture described in this paper can be implemented in many different forms. The register transfer level (RTL) files were created in such a way that changing one of the following options normally requires only the addition of a single line defining a symbol used in the files. None of these options can be changed at run time.

2.2.1 data and tag sizes

The Plurion architecture follows the venerable footsteps of such great computers as the Burroughs mainframes and the Lisp Machines in implementing dynamic strong typing at the hardware level. This is done with two systems which are described here and in the next section. The most basic system is the inclusion of a hardware recognized *tag* in every single word. They tell the hardware how the rest of the bits in the word should be interpreted so that it is not possible for the software to do the wrong thing.

31+1 is the option most compatible with Squeak Smalltalk. The single tag bit indicates if the data should be interpreted as a signed integer (tag=0) between -1,073,741,824 and 1,073,741,823 or as an object reference (tag=1). The bits for object references are divided into a "group" part and an "object" part. The top two bits indicate the size of the two parts (0/29, 6/23, 14/15 or 22/7) where the case with no group part really indicates references to symbols.

32+4 is the best option and should be used whenever possible, which is the case when a recent FPGA or custom chip is used and the main memory has extra bits normally used for error checking and correction (ECC). As the data is the same size as used in other machines and languages, exchanging data with them is simpler. The meaning of the tags is:

tag	interpretation of the data bits
0000	**signed integer between -2,147,483,648 and 2,147,483,647**
0001	**floating point**
0010	**character**
0011	**symbol**
0100	**bytecodes**
0101	**start of bitmap**
0110	**middle of bitmap**
0111	**end of bitmap**
1000	**object reference (24 bits for group, 8 bits for object)**
1001	**object reference (16 bits for group, 16 bits for object)**
1010	**object reference (8 bits for group, 16 bits for object)**
1011	**future objects**
1100	
1101	
1110	
1111	

2.2.2 object/map association

The hardware has total control over the objects with special values in the tag bits, but for the generic object references an alternative scheme is needed. This is done by associating each object with a map, which can be thought of as extending the tag bits

by a whole word. Instructions which must have object specific behavior (like *Send*) use the map to do the right thing.

first field is the traditional solution used in Smalltalk implementations. It requires one memory access whenever the map is needed. Traditionally, the map is an object reference, but as far as the hardware is concerned it is an opaque bunch of bits. The software is free to select bits so that they can have further interpretations.

cache allows an arbitrarily complex algorithm to find the map for a given object, and depends on an efficient cache access to eliminate most of the costs for repeated look ups. In the previous options only a single map could be associated with each object, but it is very interesting to allow different associations depending on the context.

2.2.3 data path serialization

Most instructions execute in a single clock cycle while operating on up to two machine words. It is possible to trade off time for space by selecting a level of serialization in the data path which is different than one. A 31+1 wide implementation with a level of serialization of four would take four clock cycles to deal with two word wide operands, but the reduction of registers and logic to only 8 bits would make the implementation smaller than normal. It would not be one fourth of the size since not everything would be narrower, the control logic would be more complex and some extra multiplexing circuits would have to be added.

2.2.4 instruction cache size

The instruction cache must hold a whole number of 8 word entries. The absolute minimum size is 1 entry, because with 0 the whole design would have to be changed to directly address the external memory. This has a significant impact on performance, however, and so the largest practical size for any given implementation technology should always be selected.

2.2.5 data cache size

The data cache must hold a whole number of 4 word entries. The absolute minimum size is 1 entry, because with 0 the whole design would have to be changed to directly address the external memory.

2.2.6 stack cache size

The stack cache must hold a whole number of 8 word data frames and a similar number of 4 word return frames. The absolute minimum size is 2 entries of each kind because with just 1 a reload would cause deadlocks and with 0 the whole design would have to be changed to directly address the external memory. Given the difference in size between the two kinds of entries, a power of two size for the total cache will make it

impossible to allocate an equal number for them both. That is not a problem - while it is common for each return frame to correspond to one data frame, it is possible for it to correspond to several.

Depending on what other options were selected, it is possible for the system to have to flush/reload the stack cache on each process switch (when the group is not encoded in the object reference). In that case the "as large as possible" rule doesn't apply since the added overhead of process switching might eliminate any gain in send/return speed.

2.2.7 shared instruction caches within processor pairs

Normally each stack processor has its own instruction, data and stack caches. When the technology allows dual port memories to be used without any penalty (FPGAs, for example) it is possible to use a single memory block for the instruction cache in two neighboring stack processors. This allows the cache to be twice as large as it would be otherwise. The improvement in the hit rate would not be a good as it might initially seem since the two processors will be fighting over a shared resources (entries), but that could be compensated by changing the scheduling algorithm to use any of the two for a given process since both have access to its working set.

2.2.8 shared data caches within processor pairs

This is exactly the same as the previous case, but with the data cache instead of instruction cache. Though available as a separate option, it is a good idea to have both shared or neither shared, specially if the scheduler is to behave as described above. The stack cache can never be shared.

2.2.9 size of SDRAM data

Memory bandwidth is the most critical element in obtaining high performance. The use of three caches for each stack processor and separate local memories for each I/O processor is meant to increase the internal bandwidth available independently of the external memory. But a wide path to main memory is still a good idea.

16 is used in the Oliver truck terminal and the XSA development kits from Xess. A 32+4 data/tag system is not possible in this case. It is the lowest cost option practical.

32 is a trivial way to improve a design that was previously 16 bit, but most of the other problems would remain.

36 would allow all interesting data/tag widths, but is hard to implement with a small number of chips. Three 16 bit wide memories would be 48 bits, so half of one chip would be wasted.

64 is the most common width for low cost memory modules today (DIMM and SODIMM, SDRAM and DDR SDRAM). A 32+4 data/tag system is not possible in this case.

72 allows all interesting data/tag widths and while some variations (DDR SODIMM with ECC) are hard to find in distributors that deal with end users, they are available via OEM distributors.

2.2.10 Memory Controller Options

Either normal or double data rate (DDR) synchronous dynamic memory (SDRAM) can be used, but for now this is not a run time option. Older dynamic memories and static memories are not supported. Although a bit awkward, the controller can optionally be used to access Flash memory. A better alternative is to use an I/O processor for that but that isn't possible when the two kinds of memory share cpu pins (as in the Oliver truck terminal).

2.2.11 hardware objects

The stack processors have a core part and caches, but also include a number of hardware objects. They are accessed with the *Special Send* instructions and can receive two word and send back one result from/to the stack cache on each clock cycle. Three of these objects must be included in each stack processor in order to allow them to be complete enough to run most programs: ALU, CMP and MEM. In theory, that is enough for them to run any program since all other hardware objects can have software equivalents which are invoked by the same instructions when they are missing in an implementation. An example of such an object is MAC. When it is not present and the program tries to execute a multiplication instruction, for example, then a corresponding software method is executed instead and the main program can't tell the difference except for the speed.

A major complication is the case where a given hardware object is present on some of the stack processors, but not all of them. If it is a particularly large one (like an inverse DCT conversion hardware for speeding up MPEG movie decoding) this might be the only practical alternative. The software method should not executed when invoked on a processor without the hardware, but should instead suspend the process and tell the scheduler to force it to run on a processor that does have it. The instruction would be retried and this time it would work just fine.

2.2.12 number of stack processors

This is selected "manually" by changing the top design file to include the required number. Since the ring network makes it easy to snap them together into a large system, this is no big deal.

2.2.13 number of I/O processors

This also requires hand coding of the main design, as in the previous option. An additional complication is that while the stack processors are entirely internal to Plurion, the I/O processors must have access to external pins. Between the core of the I/O processor and the pins there usually needs to be a simple circuit to customize for the

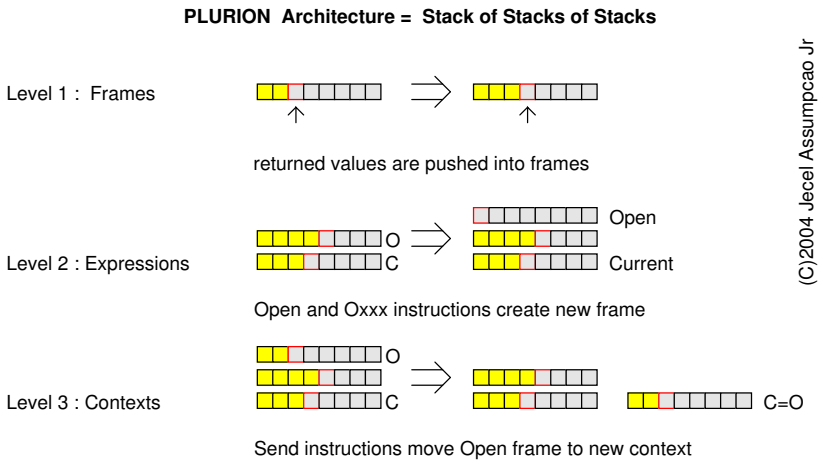
different interfaces. In the case of FPGAs the I/O processors start out with their programming preloaded, so that is an additional customization that must be included in the design. On ASICs operating at sufficiently high frequencies, the adapter circuits can be replaced with software solutions and a generic interface to the pins.

2.2.14 I/O processors are multiplexed?

The detailed description of the I/O processors in this paper shows it switching between 16 fixed priority tasks in the spirit of the old Xerox PARC Alto computer. That allows a significant amount of hardware to be shared between different interfaces. When there are enough resources available for a design, however, a more interesting alternative is to have 16 separate I/O processors, each running a single task, instead of multiplexing one. A single system might have a mix of multiplexed and separate I/O processors if that is the best solution for a particular application.

Chapter 3

Execution Model



Though inspired by the Lisp machines and their "split call" instructions and also by my own previous work in alternate code representation for Smalltalk, this design has not been previously considered in computing science as far as I know.

At the lowest level we have "frames", which are small and fixed sized stacks. Each one can hold up to 8 elements. As results are generated by the hardware, they are pushed into some frame.

The next level is a stack of frames, called the expression stack. It includes a variable number of frames. The base frame in this stack is called the current frame and is indicated by "C" in the drawings. The top element in this stack is called the open frame, indicated by "O".

The "Open" instruction (and the "Oxxx" variations of other instructions as explained below) allocates a new, empty frame and pushes it on top of the expression stack. Many instructions use the data in O and then pop it from the expression stack, pushing their result into the new O. It took a while for me to convince myself that this

was the desired behavior, just like the idea of using objects and messages for something as basic as integer math seemed so absurd that most languages since Smalltalk have gotten it wrong. This simple expression

```
3 + 4
```

compiles to

```
OiLit 3 iLit 4 +
```

where the first instruction allocates a new frame and the third destroys it. As long as the hardware can handle such operations in a single clock (without making us slow down that clock, of course) then this is perfectly reasonable no matter how wasteful it might seem.

The "Open" has a very nice correspondence to the source code. In Neo Smalltalk it means an increased in the level of underlining and in Smalltalk-80 with all the optional parenthesis present it indicates an open parenthesis. Note that evaluation order in the machine code corresponds to left-to-right in the source, unlike in Oliver where it is inverted. Here is an example with the comments showing how the frames in the expression stack work:

.... ((3 + 4) * 5) ....

```
; <....  42> initial expression stack
Open ; <....  42> <> new frame
OiLit 3 ; <....  42> <> <3> another new frame with one element
iLit 4 ; <....  42> <> <3 4>
+ ; <....  42> <7> popped a frame
iLit 5 ; <....  42> <7 5>
* ; <....  42 35>
```

Other instructions will be described further down which have different effects on the expression stack.

The top level is called the context stack, and includes a variable number of expression stacks. It grows due to the "Send" instructions (and variations) and shrinks due to the "Return" instruction (and its one variation). During the execution of a Send, O is converted into the base of a new expression stack. That causes O and C to be the same, which is normal.

3.1 Return Stack

Though not normally visible to the programmer like the data stack frames described above, it is the return stack that give these frames their structure. For each frame shown in the figure at the begining of this chapter, a four word return frame with the following format is allocated:

name	description
PC	program counter: indicates the next bytecode to be executed
SP	stack pointer: indicates both where a copy of the data frame may be found in main memory and (in the three lowest bits) which word in that frame will receive data that is pushed
Sender	indicates another return frame which will become the active one when we answer with some result
Current/NextLexical	for open frames this indicates the corresponding Current frame (in the return stack), while for block current frames this indicates the current frame for the next outer lexical level

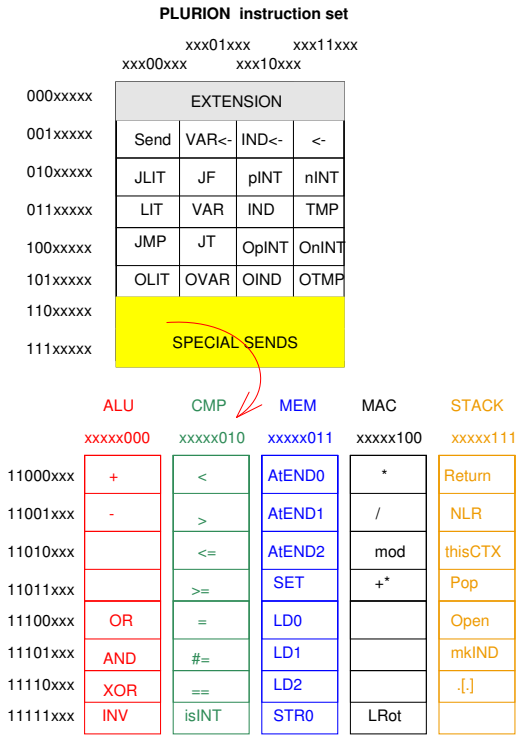
The PC is present in every return stack frame, but isn't necessarily valid in all of them. In particular, when an O frame becomes C due to a message send then the value of the PC in the calling method is stored in the return frame right below the ex-O. The PC in frames below that (if any) should be ignored.

When a new frame is allocated and O is set to point to it, the previous value of O is saved in the Sender word of the new frame and the value of C is stored in the Current/NextLexical word. This is what links the data frames into the stacks of stacks of stacks structure. When a frame is used as C inside a block method the Current/NextLexical word is used by the .[.] instruction to set the frames of the outer lexical scope for that block as an extension of C. The reuse of the same word for two different purposes slightly complicates algorithms that walk the return stack but that is a small price to pay for not having a wasted word in nearly all return frames.

The SP is a little tricky - unlike Sender and Current/NextLexical it does not refer to another return frame (four words) but to a data frame (eight words) instead. The data frames are aligned to eight word multiples, so SP points to the first free word at the top of the stack (indicated as the red box in the execution model figure). The lowest three bits are the word within the frame while the other bits form the address of the frame itself in main memory. The top bit, however, as a special meaning. If zero, then the indicated frame is directly the data stack we are interested in. That stack is limited to eight words, which while enough for most cases is too small a limit for parts of the system. So if the top bit of SP is one, then the indicated frame holds eight words indicating up to eight other frames that actually form the stack (up to 64 words in size). The three lowest bits of SP indicate the word in the indirect frame which actually holds the stack pointer. The contents of any words after that one are not valid. The top bit of the word fetched from the indirect block is interpreted in exactly the same way, so two (512 words), three (4096 words) or more indirections are possible. This tree of frames with a root in SP is automatically created and expanded (and contracted, as the case may be) on stack overflows or by the *Xtnd* instruction.

Chapter 4

Instruction Set



4.1 Extension Prefix

The extension instruction uses a 3 bit operation indicator and a 5 bit value. This will simply append its value to the value of the next instruction which would normally only

have 3 bits (there are 20 instructions like that). So with one extend prefix, we can have 8 bit values. With two prefixes, 13 bits and so on.

4.2 Basic Instructions

As indicated above, there are 20 instructions with 5 bit operation indicators and 3 bit values. Many of these instructions combine their value with the current instruction pointer (PC) to obtain the effective address. This is indicated as PC#value. In the case of 3 bit values (no extension prefixes) and 32 (or 36) bit words, we have PC#value=(PC&~31)+(value<<2).

The three indirect instructions seem to require two values: one to indicate where in C is the pointer to the array and another to select a word in that array. The three lowest bits in the value indicate the pointer (which must be in the first 8 words of its frame) while the higher bits, if any, select the word. So if no extension prefix is used then word 0 of the array is accessed. That is by far the most common case.

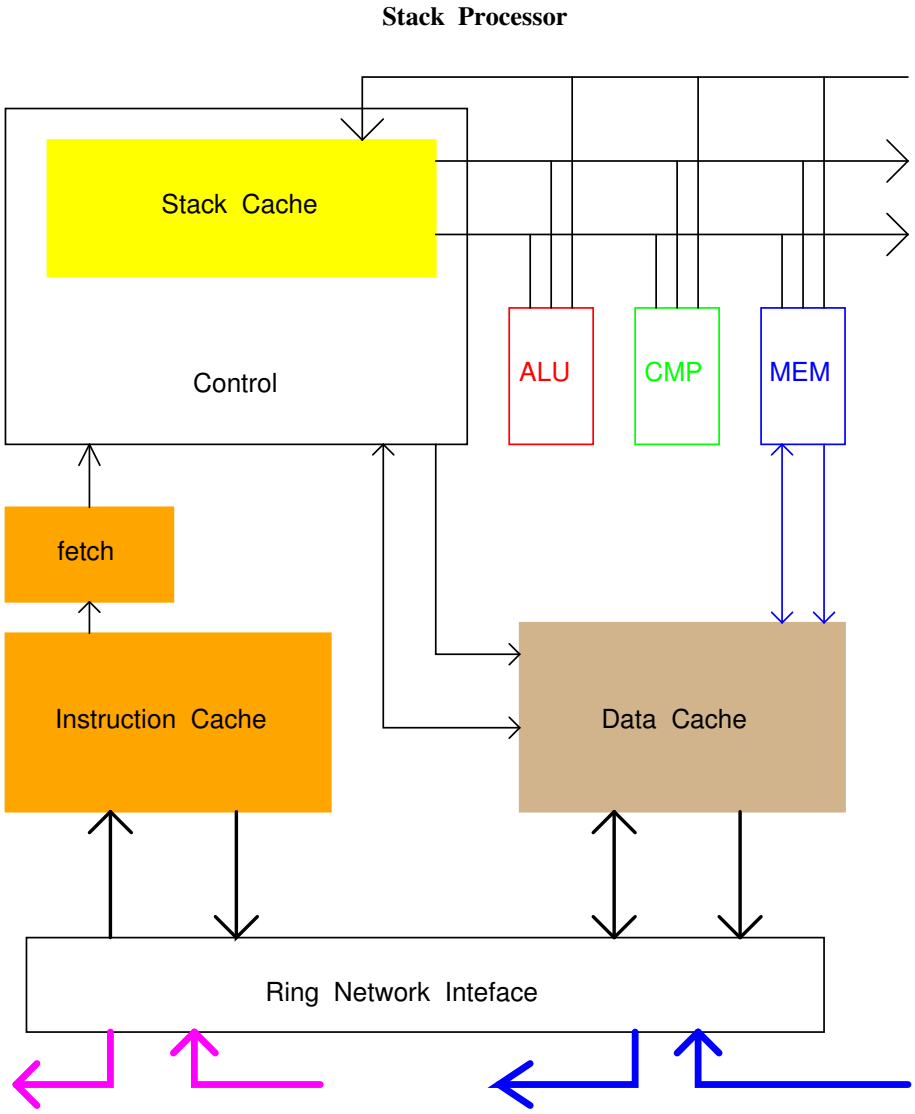
The basic instructions are:

opcode	name	description
pInt	positive integer	just pushes its value into O
OpInt	open and positive integer	allocates a new frame and pushes its value there
nInt	negative integer	just pushes the complement of its value into O
OnInt	open and negative integer	allocates a new frame and pushes the complement of its value there
Lit	literal	pushes the word at PC#value into O
OLit	open and literal	allocates a new frame and pushes the word at PC#value there
Var	instance variable	pushes the word in object SELF indexed by value into O
OVar	open and instance variable	allocates a new frame and pushes the word in object SELF indeed by value there
Tmp	temporary	pushes the indicated word from C into O
OTmp	open and temporary	allocates a new frame and pushes the indicated word from C into O
IND	indirect	pushes the indicated word from the array pointed to by the indicated word in C into O
OIND	open and indirect	allocates a new frame and pushes the indicated word from the array pointed to by the indicated word in C into O

opcode	name	description
Var⇐	assign instance variable	pops a value from O and stores it in the word in object SELF indexed by value
⇐	assign temporary	pops a value from O and stores it in C at the selected word
IND⇐	assign indirect	pops a value from O and stores it in the array pointed to by the indicated word in C at the selected word
Jmp	jump	changes PC to PC#value
JF	jump if false	pops a value from O and changes PC to PC#value if that was False
JT	jump if true	pops a value from O and changes PC to PC#value if that was True
JLit	jump literal	changes PC to contents of word at PC#value
Send	send	creates a new expression stack from O and switches execution to ”PIC mode”

4.3 Special Sends

There are eight special send instructions. What each of them does depends on which hardware object is selected as the receiver by the value field of the instruction. With three bits in this field, up to eight hardware objects could be selected without using the extension prefixes. With a one byte prefix, up to 512 different hardware objects could be the receiver for a special send, though 8 of these would be the same as in the one byte version of the instruction.



What happens if the selected hardware object is not present? If no other processor includes that hardware, then the instruction is converted into a regular Send instruction. The first element in O is the new receiver while the selector is derived from the original instruction. If, however, it is known that some other processor in the chip does include the needed hardware then a trap is generated and the scheduler moves this task to that other processor where it will resume execution and should be able to run the instruction.

In addition to missing hardware, it is also possible for the hardware to be unable to execute the instruction due to incompatible operands. The ALU, for example, depends on its operands being small integers and with such values that the instruction won't

cause an overflow or similar error.

4.3.1 STACK (hardware object 7)

This is a group of control instructions which change the stacks in non standard ways. Return and NLR (non local return) pop the context stack and transfer the result from the old O to the new O. The non local variation can pop a certain number of contexts at a time.

thisCTX pushes a reference to the current context as a full object into O. This has a side effect of marking that context so it is not automatically freed due to a Return instruction.

Open has already been described.

mkIND takes a number as an argument and allocates an array of the required size, pushing its pointer in C. This is used for values which must be shared between two or more levels of blocks.

The .[.] instruction deals with blocks. It creates a special object which points to the current context (and so is similar to *thisCTX*) such that later it can be used to create a new context which is linked to the current one.

The *Pop* instruction isn't needed, but very convenient to have. Since the compiler can keep track of the stack level for each instruction, it would be possible to use "<- 6" instead of "Pop" to get exactly the same effect when we know that element 6 happens to be the current top of the stack.

4.3.2 ALU (hardware object 0)

Normal math operations, all except the last one use a frame with exactly two elements in it."Inv" only needs one element.

4.3.3 RawALU (hardware object 1)

Exactly the same instructions as for hardware object 0, but all 32 bits are used and no tag checking is done. This is only needed on the 32 bit implementations since on 36 bit machines even the tagged operations manipulate 32 bits.

4.3.4 CMP (hardware object 2)

Comparison instructions, which like the ALU use a frame with exactly two elements (except for the last, which needs one). Except for the "==" and "isInt" instructions, the others require their operands to be small integers or they are converted into a Send. The result for all these instructions is either a True or a False.

4.3.5 MEM (hardware object 3)

These are very unusual, more normally found in DSPs. The processor has three "streams" (0 to 2) which each have four registers:

- object (all memory operations are in the context of some object)
- index (which word is part of the operation)
- step (how much to change index in each operation
- limit (index can't go beyond this)

The *Set* instruction will set the four registers for the lowest currently uninitialised stream. These are stored at positions 0 to 3 for the first stream, 4 to 7 for the second and 8 to 11 for the third stream. This means that any arguments which will be needed later should be saved elsewhere before using these instruction. If less than four values are present in O, the ones omitted use these defaults values: SELF, 0, 1, object size. So a frame with two values would indicate the object and index, while the step would be 1 and the limit would be the size of the object.

Str0 will pop a value from O and store it in memory as indicated by stream 0. *Ld0* to *Ld2* will read a value from memory as instructed by the respective stream. After any of these operations, the value of index is updated as

```
index := (index + step) mod limit
```

If as a result of this the index "wrapped around", then executing *AtEnd0*, *AtEnd1* or *AtEnd2* (depending on which stream we wish to test) will push a True value.

Here is how these instructions might be used to add a column of one array to a row of another:

```
; the following segment is out of order:
; it needs to save the second argument before it is destroyed
below
OTmp 2 ; push second argument, array B
iLit 0 ; index = 0
OTmp 2
Send "rowSize"
; the following segment is out of order:
; it needs to save the first argument before it is destroyed
below
OTmp 1 ; push first argument, array A
; here things get back to normal
OTmp 0 ; push SELF, which is the array to receive the result
Set0 ; index = 0, step = 1, limit = SELF size
; the above instruction destroyed Tmp0 through Tmp3
;(though Tmp0 happened to receive its previous value)
```

```
    ; now the topmost frame is the one created by the OTmp 1 instruction
    above
    Set1 ; index = 0, step = 1, limit = A size
    ; now the topmost frame is the one created by the first OTmp
    2 instruction above
    Set2 ; index = 0, step = B rowSize, limit = B size

loop:

    Open ; space for arguments to add
    Ld1
    Ld2
    +
    Str0 ; SELF[..]  := A[..]+B[..]
    atEnd0 ; check if finished
    JF loop
```

This kind of thing is very important when doing garbage collection and other low level tasks. At the same time it doesn't add much overhead to simple operations like #at: ant #at:put:.

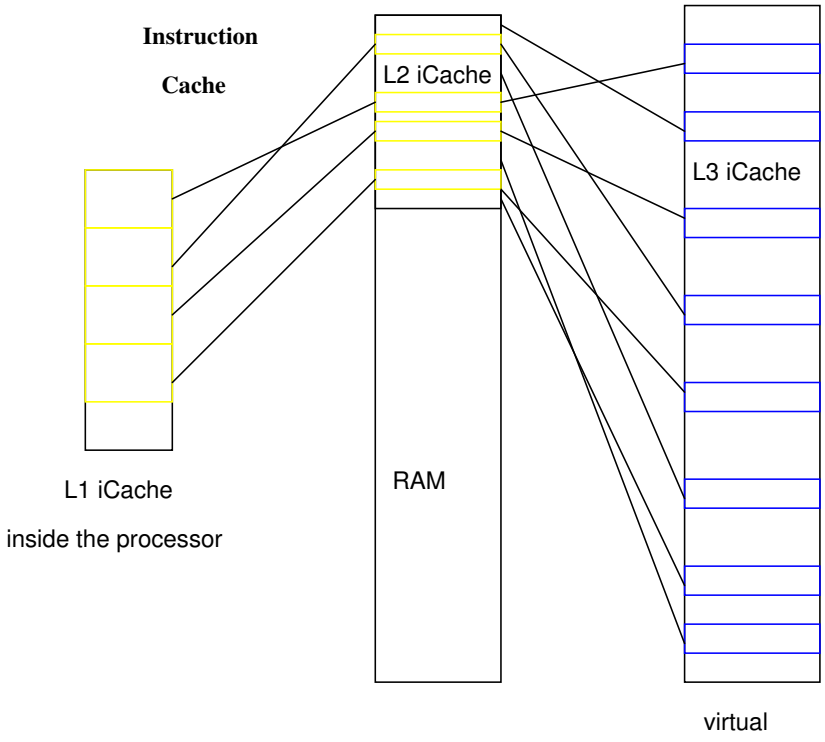
4.3.6 MAC (hardware object 4)

While the first four hardware objects must be present in all Stack Processors, the Multiply and ACcummulate unit is optional. This is why it isn't shown in the block diagram. Its connections are exactly like the ones for the ALU or CMP blocks.

Besides multiplication and division instructions, this hardware also handles rotation (left and right shifts can be done simply by selecting a convenient operand for the division and multiplication, respectively). A variation of the multiply instruction adds the result to the top of the stack in the current context. A normal multiplication will leave its result on that stack, so is used to start a sequence of multiply-add.

Chapter 5

Instruction Cache

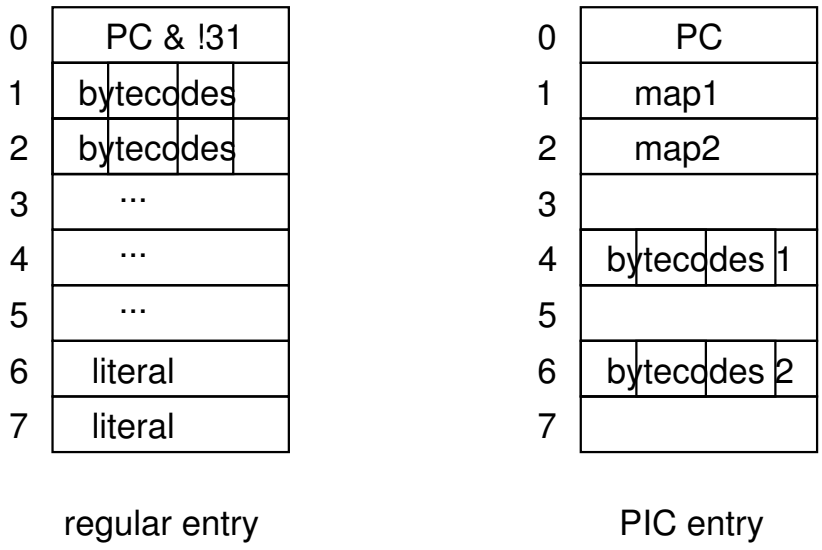


The first level (L1) of the instruction cache is a part of the Stack Processors. It holds a subset of the data present in the second level (L2), which is implemented as a reserved area in the main memory. Depending on the application and the size of main memory, from 2 to 8MB are typically used for this purpose. The data in L2 is supposed to be a subset of the data in the 4GB (typical) third level instruction cache (L3), but this third level doesn't actually exist. Instead, every time a line is requested

from the second level and it isn't present, a special software handler is called which determines what the content of that line should be and then loads into L2. This might be as simple as copying some words from the main memory or could be as complicated as translating from a different instruction set (Java bytecodes, for example) or doing one of several optimizations. Besides being virtual, L3 is considered a cache instead of just instruction memory since it can be seen as a subset of an even larger memory.

Each line in any of the three levels consists of 8 words. There are two different formats:

Instruction Cache Entry Formats



The first word in the line indicates the value of the (virtual) Program Counter associated with that line. The actual address of the line in the first two levels is obtained via a hashing algorithm from the PC. For L3 the PC is the address of the line. In most designs, the tag associated with each line is stored separately from the data in that line. In FPGAs with only a few, relatively large RAM blocks which can be accessed very fast it is more convenient to combine them as shown. This also makes implementing L2 in normal memory much simpler.

Each instruction is one byte in size so the PC can indicate 32 of them in a line. Since the first word is used by the line's tag, however, bytes 0 to 3 can never be used for instructions. In fact, if execution ever is in danger of spilling over to the next line then the compiler will insert an explicit jump at the end of the current one to avoid that. In most processors this would be considered a major complication, but in Squeak 3.2, for example, the average length of a method was 27 bytecodes. And even that number is highly inflated due to Squeak's inclusion of all of a block's code in the parent method.

The lowest 5 bits of the first word in the line are 0 since the line must match for any one of 32 (actually 28) different PC values. But for the second format shown in the figure, those 5 bits are always some value other than 0 (which makes it simple to distinguish between the two). The Inline Cache (IC) was introduced in the first high performance implementation of Smalltalk (the Deutsch-Schiffman dynamic compilation system[2]) and extended into the Polymorphic Inline Cache (PIC) in Self 3.0[3]. While other processors see code as a strictly linear sequence, Plurion allows small fragments to branch off in a separate dimension from the main code for each *Send* instruction. When executing these fragments we say that the processor is in the "PIC mode".

In this mode the value of the PC remains that of the *Send* instruction and an additional 5 bit pointer is used to select the actual instruction to be executed. For a given cache line to be used in this mode, not only must the PC match word 0 in that line but the "map" associated with the message receiver (object 0 in the open frame) must match either word 1 or 2. If it doesn't, then a second cache line is searched and if that doesn't match we have a cache miss and must continue the search in L2. When the map does match word 1, the PC extension is set to 16 (word 4) and if it matches word 2 the extension is set to 24 (word 6). Words 3, 5 and 7 of the cache line can be used for either instructions or literal values. Any Jump or Return instructions cause the processor to exit PIC mode and go back to executing in the normal mode.

Chapter 6

Data Cache

In the late 1970s most processors were slower than memory chips. But while the latter have only increased their bandwidth some 50 times, processors have become more than 2000 times faster. This has made using indirections, which are the key to flexible software, extremely costly. The Mushroom project[1] addressed this issue by using indirections out of the critical paths. Most early Smalltalk implementations used an object table, so that a reference from one object to another was an index into this table. This was a very flexible design which allowed objects to move around in memory or even to be swapped to disk and back very efficiently. But the most common operation, accessing some field in an object, became more expensive since it required an extra memory access to find the object's actual address in the table. The Mushroom solution was to make the data cache be virtually addressed. With normal caches, you give the physical address of the word you want and it gives you the data if it has it. To find the physical address, you would have to look in the object table. In Mushroom you give the cache the index in the table and the field number that you are interested in and it gives you the data (if it has it) directly. Only in the relatively rare cases where the data is not in the cache do we need to look into the object table.

Most current Smalltalks have abandoned the object table in order to perform reasonably well on today's processors. Each object points directly to others. There is no way to use the Mushroom trick in a software implementation.

6.1 Neo Smalltalk Object Model

Though Plurion is a great option for running programs in Java or Smalltalk versions such as Squeak, it was specifically designed with Neo Smalltalk in mind. One of the most important aspects of an object model is how various objects are grouped together. Here are ten independent ways of doing this:

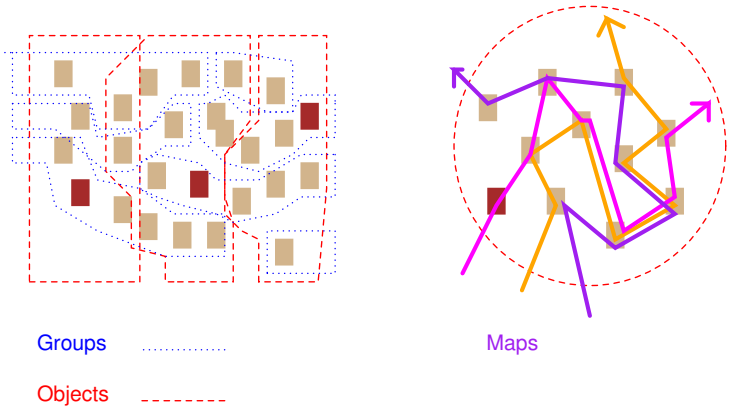
grouping	other names related to ths concept	description

grouping	other names related to ths concept	description
disk	package, segment, soup, image, snapshot	In theory, every individual object could be stored seperately in permanent memory. The opposite strategy, where all objects are saved in a single file, has often been used. Intermediate solutions give the users more control over disk management.
process	thread, transaction, actor	The most common implementations have concurrency as totally independent of objects, but that is not natural and is harder to program. When making execution an integral part of objects, this can be applied to all objects or they can be separated into active and passive objects.
security	capability, user, session	When multiple users share a system, it is important to protect one from the actions of the others. Even with a single user it is a good idea to make accidents less likely and/or less severe.
type	class, clone family	Even when a programmer is presented with a model in which each object is totally independent of the others, that is not practical in terms of implementation. The system must use some notion of type to associate executable code with the received messages.
inheritance	sub/superclass, trait, mixin, parent/child	Factoring and refactoring is an important process in software development. Being able to express concepts in terms of differences in relation to other concepts can provide support for this.
composite	part, assembly	Just like in the real world, objects can be combined to create larger objects.
version		An object's state varies over time as a side effect of normal operation. In addition, programming changes on live objects can result in new or removed slots as well as changes in behavior.

grouping	other names related to ths concept	description
reflection	metaspace, mirror, class, aspect	While objects often simulate real world objects, they actually are computer constructs built with resources like memory and processing power. Behavioral reflection can deal with the latter and structural reflection with the former.
viewpoint	layer, subject, piece, namespace	Simple systems exist in total isolation and can be completely objective. Larger systems can't hope to be entirely consistent and must be prepared to handle different interpretations of the same concepts.
role		Concepts in different parts of the system can represent a single entity.

For any two dimensions, an example can easily be found that shows that they really are independent. Unfortunately, people can't deal with 10 dimensional models. The only solution is to merge them into a smaller number of dimensions even if some flexibility is sacrificed. This is true of every existing software system, of course. But normally the lost flexibility is not a careful design tradeoff but simply the lack of knowledge about the choices available.

This drawing shows how the organization was reduced to essentially two axis. The basic building block is called the "facet". Each facet has a collection of slots, which are pairs of names and values. The most visible organization to the users is the collection of facets into objects.



The other dimension is called "groups", which is a separate way of collecting the

facets in a manner totally separate from objects. Though also visible in the user interface, groups are more subtle than objects. The other concept shown in the figure is "maps". Each map selects an ordered subset of facets from an object. Several objects can use the same map as long as their facets can be considered equivalent in some way. This is normally possible when these objects were all created by cloning one of them.

For each object, one facet has a special status and is shown in a different color in the figure. As will be seen later, in some parts of the system it is possible to refer directly to facets while in others objects are the only entities that can be addressed. In the latter case a reference to this "top facet" is equivalent to a reference to the object.

Here is how the dimensions are associated with the concepts of groups, objects and maps:

disk - groups are transferred between disk and local memory as a unit. Groups can't be changed (see *version* below) and so are never overwritten on disk.

process - each group has its own execution thread which is independent of all the others. Objects are associated with the group holding their top facet when represented in memory, so a message from an object to another in a different group can be detected and treated differently from messages to "local" objects

security - each group has a unique ID that can't be guessed, so the only way to access an object is to have been previously given a reference to it. In addition, what messages can be sent to that object will depend on who is sending (see *viewpoint* below) so the infrastructure for a very flexible "capability based" security system is in place. The actual security depends on how the higher level software uses this

type - maps are the hardware's notion of types, so a single object can have more than one type depending on context (viewpoint)

inheritance - facets are shared between an object and its clones, which results in a fuzzy and loose version of inheritance

composite - objects have a set of references to other objects (specifically, their facets hold references to the top facets of other objects)

version - once created, a group is immutable. The only way for an object to change is to create a new group with facets that override the old ones. The association between the old and new groups is explicit

reflection - normally systems are built such that every object is associated with a set of meta-objects that control their behavior. That isn't easy for everyone to understand, so an alternative model is to have some "meta-methods" to be invoked on the object itself when needed. This won't get in the way of base-level behavior if these extra methods are only available from a special viewpoint

viewpoint - each process (and so each group) is associated with a single viewpoint. When a message is sent between objects in separate processes the current viewpoint (default context) is not changed so that the sender's viewpoint is used by the receiver. The viewpoint can be changed explicitly at any time

role - when the current viewpoint is changed, all associations between objects and their maps are affected. In contrast changing the role allows a single object to be associated with a different map in the same viewpoint

The key element is how {objectID, viewpoint} => map is implemented.

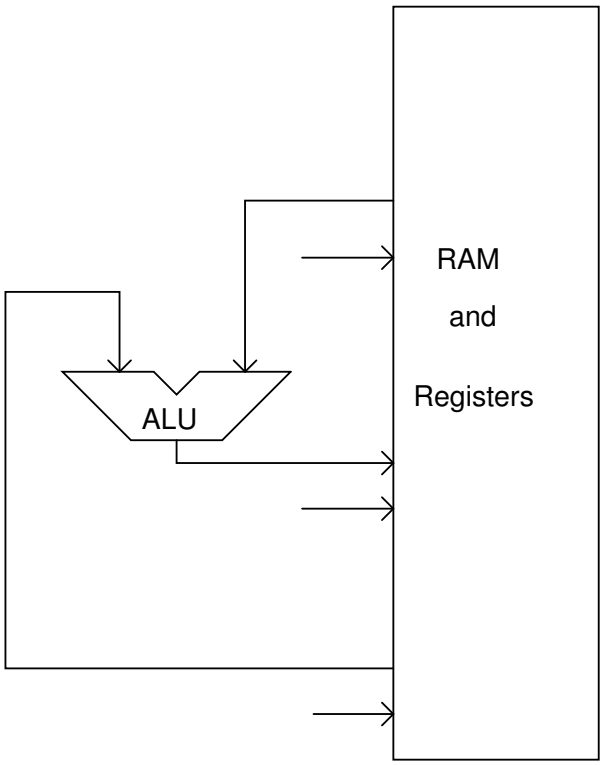
6.2 Virtual Memory and the Data Cache

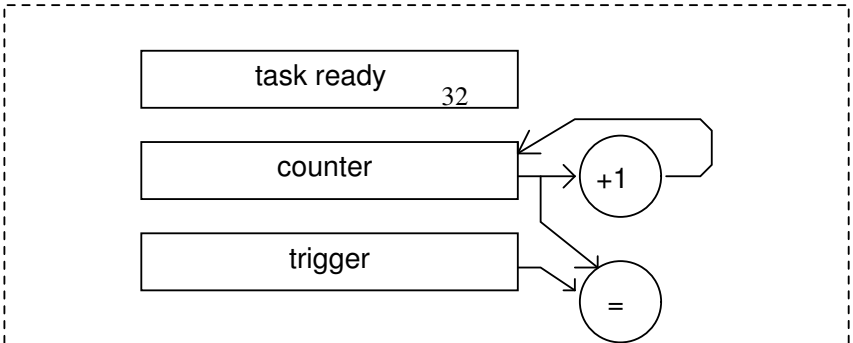
The first level of the data cache is accessed by a 2 word logical address: the objectID and the field offset.

Chapter 7

I/O Processor

I/O Processor





7.1 Instructions

The processor has a register based architecture (2 address). Each task has its own 8 register bank, with register 7 being the program counter (bit 15 of register 7 is the flag which normally indicates whether the previous result was zero, but can have other meanings for some instructions). So 128 words of memory (3F80 to 3FFF) are registers, though the registers for any unused task can be used for data or instructions.

There are two instruction formats:

bits 15 to 12	11 to 10	9 to 8	7 to 5	4 to 3	2 to 0
operation	save	destination mode	destination register	source mode	source register
xx11	save	destination mode	destination register	5 bit unsigned value	

The meaning of the operation codes is:

0000	ADD	add source to destination
0001	ADC	add source to destination and set flag to value of carry
0010	ADV	add source to destination and set flag to value of overflow
0011	ADI	add immediate to destination
0100	SUB	subtract source from destination
0101	SBB	subtract source from destination and set flag to value of borrow
0110	LSH	logical shift destination by number of bits indicated in source
0111	SBI	subtract immediate from destination
1000	XOR	exclusive or source with destination
1001	OR	or source with destination
1010	MOV	move source to destination
1011	MVI	move immediate to destination
1100	AND	and source with destination
1101	BIC	bit clear - and inverted source with destination
1110	ROT	rotate destination by number of bits indicated in source
1111	DBI	decrement destination and branch by immediate

The meaning of the save field is:

00		always save the result to the destination
01	ns	never save the result
10	ifz	save result if flag is zero
11	ifnz	save result if flag is not zero

The meaning of the mode fields is:

00	Rx	register	the register itself is used as a source or destination
01	*Rx	index	the register is used as the address of a source or destination in memory
10	*Rx++	post increment	like index, but the register is incremented after being used as an address
11	*–Rx	pre decrement	like index, but the register is decremented before being used as an address

7.2 Execution

The worst case is when switching from another task to an instruction with pre-decrement or post-increement in both source and destination:

- save PC and flags (old task)
- load PC and flags (new task)
- decode instruction and address register
- update source register
- update destination register
- fetch source/destination from memory
- execute and possibly save the result

When not switching tasks and with registers for both the source and destination then the instruction can execute in just three clock cycles. This is the most common case. At 54MHz (for an example implementation) this means the total processing power is around 18 MIPS, which is a respectable speed for the simple tasks that it must handle even when divided among 16 coroutines.

7.3 Example Adapter

When the I/O Processor is used in the Oliver truck terminal, the ports in the adapter circuits are defined as:

port	width	decription (read)	description (write)
0	16	task ready - each bit corresponds to a task and if it is 0 then the task is suspended and if it is 1 then the task is ready to run	sets the task ready value
1	16	counter - is incremented at 54MHz	trigger - when this matches the counter, task 15 is set to run
2	13		sets the sound DAC and video control (vsync, hsync, blank, burst1, burst0)
3	15	LCD data	sets LCD data and LCD control (e, w, rs, backlight, cs1, cs2, cs3)
4	16		sets chroma - bottom 16 bits of the 32 bit counter that generates a 3.58MHz sine wave (previous value is shifted up 16 bits)
5	15	keyboard columns, serial (tx1, rx1, tx2, rx2) and bar code reader	sets keyboard column value for one clock cycle (used for precharge), sets tx1, sets change in rx1 to wake up task 4, sets tx2, sets change in rx2 to wake up task 5, sets change in bar code reader to wake up task 8
6	0		
7	0		

where the 16 tasks are defined as (in order of decreasing priority):

task	name	description
15	wait	makes the single comparison circuit do the work of 16. The other tasks make requests to be wakened at a specific clock cycle (as indicated by a 16 bit counter running at 54MHz, which cycles every 1.21 ms). The comparison is set to the value of the soonest requested cycle and then the task goes to sleep. When it wakes up it also wakes up the indicated task (though since that has a lower priority it will have to wait for task 15 to finish its job first), and it finds the next soonest requested cycle and sets the comparison register to that.
14	addWait	receives a request from other tasks to be woken at a given clock cycle and then changes the tables used by task 15 so this will happen. Care is taken so that being interrupted by task 15 doesn't cause a conflict
13	videoBuf	reads four words from memory into the video buffer
12	videoV	generates the vertical timing for the video. It can also change the program counter for task 11 (which is suspended at this point) to select between different kinds of horizontal lines
11	videoH	generates the horizontal timing for the video. It wakes up task 12 once per line
10	videoChroma	accepts requests from the main processor to change the setting for the chroma frequency
9		
8	barcode	detects pulse widths in the bar code reader input
7	lcd	sends data to the liquid crystal display
6	sound	reads a byte of memory and sends it to the audio DAC
5	com2RX	receive data for serial port 2
4	com1RX	receive data for serial port 1
3	com2TX	transmit data for serial port 2
2	com1TX	transmit data for serial port 1
1	keyboard	scans the keys to check for any change
0	rtc	keeps track of the real time

Chapter 8

Testing

There are two situations that must be addressed: finding problems with the design itself during the development phase and separating the good chips from the bad ones during fabrication. In addition, it would be nice if during normal operation there are resources to help with low level software development.

Implementations using FPGAs don't have to worry about manufacturing defects since the chip was fully tested before shipping using entirely different methods from the ones described below. And booting is much simpler since the memories in a FPGA can be initialized to any desired value during programming, while in an ASIC they have unknown values after reset and must be explicitly loaded. Since a solution for ASICs also works for FPGAs (even if it is overkill), a single design was created for both implementations.

The strategy adopted includes making subcomponents in the processors addressable from the internal network and adding a special ROM to one of the I/O Processors. After reset, that processor starts executing code from that ROM. An external signal can select between a short test plus boot or an extensive test. The sequence for both is very similar:

1. a checksum for the test ROM is generated. If different than expected, indicate failure
2. do simple tests for internal network. If they fail, indicate it
3. pseudo random test vectors for I/O Processor blocks are generated
4. results are compared among all I/O Processors. If they don't match, indicate failure
5. pseudo random test vectors for Stack Processor blocks are generated
6. results are compared among all Stack Processors. If they don't match, isolate fault and indicate partial failure
7. do simple tests for memory controller. If they fail, indicate it

8. do simple tests for PCI interface. If they fail, indicate it

9. load all memory blocks from external Flash memory

The addressing of the subcomponents allows test vectors to be written to a single block or to be broadcast to all equivalent blocks. Storing the test vectors in the ROM would make it far too large, so a pseudo random generator is used instead. The expected results are also missing from the ROM but since there is more than one each of the I/O and Stack Processors, a simple comparison of the results is enough to determine if they are good or not. The probability of two different processors having exactly the same defect so that the wrong results always match is extremely low. When there are more than two processors, it is even possible to determine which one of them is bad. For the I/O processor that doesn't matter since the whole chip is unusable in that case, but for Stack Processors it is possible to still use the component but with a reduced performance. This allows the fabrication yield to be higher than it normally would be.

A: Document History

Date	Changes
5 Aug 05	• added indirect instructions for blocks and eliminated tail send and delegate instructions in anticipation of major changes to chapter 3 • changed opcode encoding • eliminated project log • changed tag encoding
25 Jul 05	• changed jumps to use word addresses instead of byte addresses
27 Jun 05	• changed opcode enconding to be a little more uniform at the cost of separating the bits the indicate the selector for special sends • added RawALU hardware object to get around tag checking for 32 bit implementations
10 May 05	• moved stack instructions back to special sends to speed up decoding
28 Apr 05	• moved description of I/O Processor to its own chapter closer to the end

12 Apr 05	• removed 15+1 and 30+2 datapath options • changed I/O Processor to a PDP-11 style design
4 Dec 04	• pop instruction added (not really need, but nice to have) • instructions for MEM changed to keep state in the current context, so the example was changed as well
25 Nov 04	• added description of "return stack" to the execution model • eliminated the "no names" free hardware license, replacing it with a "publish previous version" model • added description of configurations
12 Nov 04	• changed instruction cache format for narrow width machines, eliminating the alternate PIC entry format
18 Oct 04	• rearranged bytecodes: replaced negative extensions and iLit instructions with the four number instructions, added JLit, moved CTRL special sends to Stack instructions block • added object/map association options • added alternate PIC entry format for narrow width machines
20 Sep 04	• replaced PCI interface with I/O Ring • eliminated comment about patents and replaced "preliminary version: do not distribute" with "work in progress" on the cover
9 Sep 04	• added definition for simplicity in project style

16 Aug 04	• moved * and / instructions from ALU to MAC
13 Aug 04	• changed “path” to “map” in object model figure to be compatible with the instruction cache figure • created Bibliography • added small space around tables • initial text about the architecture and instruction cache
12 Aug 04	• changed processor interface from bus (or switches in large configuration) to ring network • copied I/O Processor description from Swiki • added some text to Testing
4 Aug 04	• added object model figure
30 Jul 04	• added testing chapter and removed test from configuration • added project log appendix
28 Jul 04	• added security to object model
27 Jul 04	• added composite to object model
26 Jul 04	• added object model to data cache explanation • changed basic instruction explanation from list to table • more text in the introduction

23 Jul 04	• added the document history
21 Jul 04	• added drawing of instruction cache levels • more text in the introduction
19 Jul 04	• added drawing of small and large configurations • added drawing of stack processor and I/O processor • added configuration options • initial text for introduction • fixed scale of instruction set drawing
17 Jul 04	• created this document by copying two drawings and description of architecture and instruction set from an email • created chapters and abstract • changed encoding of instruction set

Bibliography

[1] “An Object-Based Memory Architecture” Ifor Williams and Mario Wolczko. In Implementing Persistent Object Bases: Proceedings of the Fourth International Workshop on Persistent Object Systems, Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, pages 114-130. Morgan Kaufmann Publishers, Inc., 1991.

[2] “Efficient Implementation of the Smalltalk-80 System” L. P. Deutsch and A. M. Schiffman. In Proceedings of the 11th Annual ACM SIGACT News-SIGPLAN Notices Symposium on the Principals of Programming Languages. Salt Lake City, Utah, January 1984.

[3] “Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches” Urs Hölzle, Craig Chambers, and David Ungar. In ECOOP ’91 Conference Proceedings, Geneva, Switzerland, July, 1991. Published as Springer Verlag LNCS 512, 1991.