Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Hanspeter Hug

Design And Implementation Of An Emulator For The IBM PowerNP 4GS3

Diploma Thesis DA-2004.08
November 2003 to March 2004

Supervisor: Lukas Ruf
Co-Supervisor: Matthias Bossardt
Professor: Bernhard Plattner

Abstract

This thesis presents the design and the implementation of an emulator for the IBM PowerNP 4GS3 network processor.

The IBM PowerNP 4GS3 combines an embedded PowerPC core and the highly specialized Embedded Processor Complex in one architecture resulting in complex interactions between multiple processors running in parallel.

The emulator is designed to be both flexible and accurate. Flexibility means that a small number of concepts are used to model the hardware. Accuracy means that software written for a real PowerNP 4GS3 runs unmodified on the emulator and vice versa.

Basic concepts, such as devices, signals and memory mapped I/O are discussed in a hardware independant fashion. It is then shown how the specific hardware architecture of the PowerNP 4GS3 is implemented using these concepts.

The emulator implementation discussed is in an advanced state of development. It is capable of booting and running the Linux operating system.

Contents

1 Introduction **5**

1.1 Related Work 5
1.2 Problem Statement 6
1.3 Document Structure 6
1.4 Acknowledgements 6

2 The PCE emulator framework **7**

2.1 Design Criteria 7
2.2 Building Blocks 7
2.2.1 Devices 8
2.2.2 Signals 8
2.2.3 Load / Store 10
2.2.4 Memory and Memory Mapped IO 12
2.2.5 Clocking 12

3 The PowerPC 405 Emulator **14**

3.1 Instruction execution 14
3.2 Caches 16
3.3 Memory Management Unit 16
3.4 Traps, Interrupts and Exceptions 18
3.5 Differences 18

4 The Embedded Processor Complex Emulator **20**

4.1 Core Language Processor 21
4.1.1 Distributed Address Space 23
4.1.2 Coprocessor Command Execution 24
4.2 Differences 27

5 Evaluation **29**

5.1 Verification 29

6 Summary **30**

6.1 Implementation Status 30
6.2 Outlook 30

A Problem **31**

B Core Language Processor Opcodes **34**

C README and Scripts 59

C.1 README 59

C.2 Cross Compiler README 59

C.3 PCE README 60

C.4 Emulator Start Script 62

C.5 Cross Compiler Build Script 62

C.6 PCE Build Script 63

D Source Tree Layout 64

E Configuration File Syntax 65

F Timeline 67

List of Figures

1 PCE example setup 8
2 Signal setup for a load 10
3 Signal setup for a store 10
4 The memory subsystem 13
5 Memory access example 13
6 PowerPC instruction word 15
7 The Embedded Processor Complex 20
8 The Dyadic Protocol Processor 20
9 Coprocessor memory locations 24
10 The CLP address encoding 24

1 Introduction

Emulators are virtual machines. The difference between an emulator and other virtual machines (such as for example the Java Virtual Machine) is, that an emulator is built according to the specification of an actual existing hardware architecture. That architecture was most likely designed without software emulation in mind. It may not even be exactly specified. Often a concrete implementation of a hardware architecture is at the same time its definition. This is the source for many of the problems involved in designing an emulator.

Emulators are used for four main purposes:

1. As a **replacement** for the actual hardware. This is particularly common for older hardware that is no longer produced but still needed for some reason. Also, in situations where performance is of no concern, it may be cheaper and/or more convenient to use an emulator that can run on any available hardware of choice rather than to use (and maintain) the actual hardware.

2. As a convenient way to **study** a hardware architecture in depth. An emulator may be written before the actual hardware goes into production. Obvious design errors and/or shortcomings may be detected by inspecting the emulator. This also allows to try out many different design alternatives, which would be impossible by using real hardware (due to time and cost constraints).

3. After the actual hardware is built, an emulator can assist in **software development**. Especially useful is the fact that emulators can easily be extended. New features, in particular I/O devices, can be temporarily added for the purpose of easier access during early software development.

4. Lastly, an emulator offers far superior **debugging** facilities than any real machine. An emulator, because it simulates a physical device, can violate the laws of physics. Debugging by using an emulator is by nature completely non-intrusive. An emulator, opposed to a real machine, can be stopped at any time, its state can be inspected and possibly altered and execution can be resumed without the software running on the emulated hardware ever noticing.

Depending on what an emulator is used for, different degrees of accuracy are required. For example, an emulator that is used to run software designed for a real hardware architecture need not be concerned with precise instruction timing or simulating all intermediate states of the hardware as long as all effects that can be observed by the software are the same as on the real hardware. An emulator that is used to analyse the emulated hardware, on the other hand, needs an emulation in far greater detail.

In general, an emulator needs to be so precise that the intended observer can see no difference between the emulated hardware and the real hardware. Who the intended observer is, is application specific. It could be, for example, software running on the emulated hardware. It could also be other, real, hardware or software not running on the emulated hardware.

1.1 Related Work

- **Bochs** [5] is an open source IA-32 (Intel Architecture) PC emulator. It features reasonable execution speed along with very complete and mature support for many peripheral devices.

- **VMware** [6] is a commercial IA-32 emulator. It offers a fast and accurate emulation of an entire IA-32 PC. On the downside, it is only available for IA-32 based host operating systems.

- The **TIK emulation framework** [7] is a modular emulation framework. It emulates a MipsR3051 CPU and is capable of running the Topsy operating system [8].

1.2 Problem Statement

The purpose of this project is to design and implement an emulator for the IBM PowerNP 4GS3 network processor. The emulator is intended to help software development for that architecture.

Problems that go beyond the usual problems of software engineering include:

- The PowerNP 4GS3 is a parallel processor. There are well over 80 processors, all working in parallel. Synchronization and communication between the individual processors is not trivial, partially due to the sheer number of processors involved and partially due to the fact that the processors are not all the same.
- The NP4GS3 combines two unrelated hardware architectures in one product. On one side there is a standard PowerPC 405 microprocessor, on the other side there is the Embedded Processor Complex, a set of highly specialized processors. While emulators exist for both parts individually, there is at present no emulator that integrates the two.

The architecture of the NP4GS3 is described in [1] with additional information in [4]. The PowerPC 405 architecture is described in [2] and [3].

1.3 Document Structure

The remainder of this document describes the NP4GS3 emulator in three main parts:

- Section 2 discusses the PCE emulator framework. Basic building blocks are introduced independant of any hardware architecture.
- Section 3 describes the PowerPC 405 emulator.
- Section 4 describes the Embedded Processor Complex emulator.

- An evaluation of the emulator is presented in section 5.
- The achieved project goals are summarized in section 6.

1.4 Acknowledgements

First and foremost, I would like to express my gratefulness for the support received from Lukas Ruf. I would like to acknowledge the opportunity to carry out this diploma thesis at the Communication Systems Group led by Prof. Dr. Bernhard Plattner at the Department of Information Technology and Electrical Engineering of the Swiss Federal Institute of Technology (ETH) Zurich.

2 The PCE emulator framework

This section describes the concepts of the PCE (Personal Computer Emulator) emulator framework. Despite its name, the framework is not limited to emulating Personal Computers. It provides enough flexibility to emulate almost any conceivable computer system.

The PCE emulator framework was originally conceived to support the implementation of an IBM PC emulator (hence its name). It was, however, built from the start with flexibility in mind, so that its key concepts would be generic in nature and that entirely different architectures could be implemented using the same concepts.

The main difficulty that any software emulator faces is representing hardware concepts in software. There are two viable approaches to solving this problem.

In a software friendly approach a given hardware architecture is translated into data and control structures that best suit the implementing programming language. Presumably these structures are optimized for both simplicity and efficiency. The benefits of this approach consequently are high execution speed and a comparatively short and simple implementation. The drawback is that due to the tight coupling between the software and hardware implementation small changes to the hardware specifications may result in major changes to the emulating software.

A more hardware friendly approach reverses the benefits and drawbacks of the software friendly approach. Instead of modelling any particular hardware architecture, generic hardware concepts are represented in software. This may result in more overhead when emulating a particular hardware architecture. On the other hand, any change to the hardware specification or indeed any newly emulated hardware part has a natural representation in software.

PCE follows the second approach and simulates hardware at a low level. The following sections describe how hardware concepts are modelled in the PCE framework.

2.1 Design Criteria

The design criteria were, in order of decreasing priority:

1. Flexibility. Any hardware device should be representable in software and small changes to the emulated hardware should result in small changes to the emulating software.
2. Accuracy. The emulating software should function identically to the emulated hardware in as many respects as possible. For example, it should not be possible for a program to tell the difference between the real hardware and the emulator.
3. Portability. The PCE framework and the NP4GS3 emulator are implemented in standard compliant C. They should both require nothing more than an ISO-C compliant compiler.
4. Efficiency. Unfortunately, efficiency and accuracy and to a certain degree effiency and flexibility are at odds. Still, the emulating software should be as efficient as possible under the constraints of flexibility and accuracy.

2.2 Building Blocks

In order to facilitate further discussions, let us consider the (constructed) example setup shown in figure 2.2. It shows to devices DevA and DevB that are connected by two signals SigC and SigD. SigC is an output signal of device A and an input signal of Device B. SigD is an output signal of device B and and input signal of device A. The key components are devices and signals. These concepts and their implementation will be discussed in the following sections.

Figure 1: The PCE example setup

2.2.1 Devices

The term device refers to a self-contained functional unit that is capable of sending and receiving signals. A device is defined by three properties:

1. **The device state** consists of all internal storage, such as registers, flip-flops, latches and so on.
2. **The state transition rules** define the next state based on the current state. These rules define the function performed by the device.
3. **Signals** are in a strict sense part of the device state. They are special, however, because they are the part of the device state that can be observed (output signal) or changed (input signal) by an outside entity.

Device states are represented in PCE as C structs. This struct is referred to as the *device context struct* or simply the *device struct*. For example, the two devices depicted in figure 2.2 could be described in C like this:

```
typedef struct {
  /* device A state fields */
} DevA_t;

typedef struct {
  /* device B state fields */
} DevB_t;
```

2.2.2 Signals

A signal corresponds to a physical connection in hardware. PCE makes the following assumptions about signals:

- Signals have binary values. There are no tristate signals nor signals with more states.
- Signals are point to point connections. The signal state can be changed (driven) by at most one device and the signal state can be observed (received) by at most one device.

In PCE signals are modelled as function calls. The receiving side of the signal is a C function that has as parameters a pointer to the device struct of the receiving device and the new value of the signal. Such a function is referred to as a *signal receive function* or simply a *receive function*. For example, the two signal receive functions depicted in figure 2.2 could be defined like this:

```
void devb_set_sigc (DevB_t *devb, unsigned char val);
void deva_set_sigd (DevA_t *deva, unsigned char val);
```

The sending side is a bit more involved because that is where the connection between the sending device and the receiving device is made. An output signal is implemented as a pointer to a function in the sending device's device struct. This pointer points to the receiving device's receive function. Therefore the sending device also needs a pointer to the receiving device's device struct, in order to pass it as first parameter to the receive function. Because it is not known at compile time which device will be the receiving device for a particular signal, this pointer is implemented as a pointer to void in the sending device's device struct.

For example, to implement the sending sides of the two signals in figure 2.2 the two device structs of devices A and B would be extended like this:

```
typedef struct {
  void *sigc_ext;
  void (*sigc) (void *ext, unsigned char val);
} DevA_t;

typedef struct {
  void *sigd_ext;
  void (*sigd) (void *ext, unsigned char val);
} DevA_t;
```

The function types described are designed such that any output signal can be connected to any input signal. All that is required is a type cast when the output signal's function pointer is set during initialization of the device context struct. This is necessary because the receive function expects a pointer to a specific device context struct as first parameter, while the sending device can provide only a generic pointer to void.

Either the function pointer or the device struct pointer (or both) of an output signal may be NULL. A function pointer that is NULL indicates a signal that is not connected. The driving device needs to check for this condition every time the signal is driven to a new value (i.e. every time the function pointer is dereferenced).

A receive function must be handle the case that a signal is driven to the same state in which it already is. This happens when the receive function is called multiple times with the same value as second parameter. Such resettings of the signal to the same value must be ignored by the receive function. This means that a sending device is not required to remember the state of its output signal (unless it needs this information for other purposes) but can simply set the signal whenever it *might* have changed. On the other hand, a receiving device needs to remember the last state of an input signal in order to detect changes in the signal state (edges).

Finally, to complete the example, here is the initialization code that would set up the various pointers for the setup in figure 2.2:

```
DevA_t deva;
DevB_t devb;

deva.sigc_ext = &devb
deva.sigc = &devb_set_sigc;

devb.sigd_ext = &deva
devb.segd = &deva_set_sigd;
```

2.2.3 Load / Store

Loads and stores are two higher level functions that can be constructed using simple signals as discussed in section 2.2.2. The signal setup for a load ist shown in figure 2.

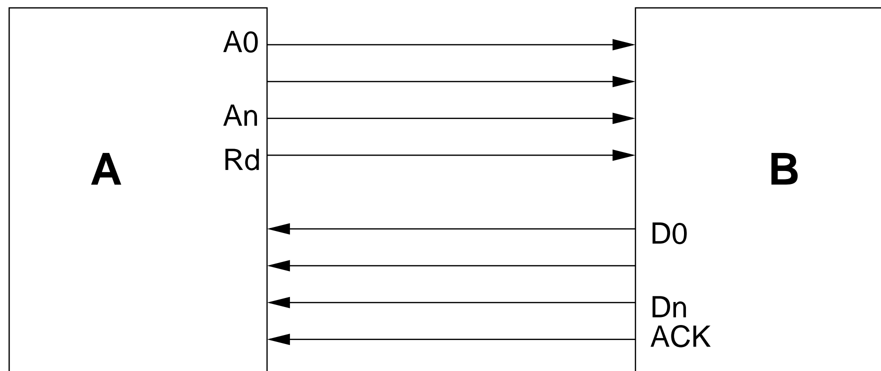


Figure 2: A signal setup for a load

Protocol for a load:

1. Device A sets its output signals A0..An to reflect the desired address.
2. Device A changes its output signal Rd from 0 to 1 to indicate the validity of the address to device B.
3. Device B detects the edge in its input signal Rd and responds by setting its output signals D0..Dn to reflect the value to be loaded.
4. Device B changes its output signal ACK from 0 to 1 to indicate the validity of the data to device A.
5. Device A examines its input signals D0..Dn and retrieves the value.
6. Device A changes its output signal Rd from 1 back to 0.
7. Device B changes its output signal ACK from 1 back to 0.

The setup for a store is shown in figure 3

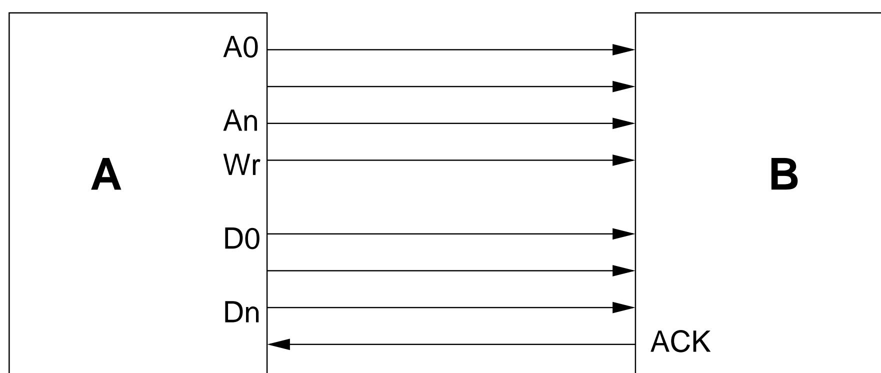


Figure 3: A signal setup for a store

Protocol for a store:

1. Device A sets its output signals A0..An to reflect the desired address.
2. Device A sets its output signals D0..Dn to reflect the value to be stored.
3. Device A changes its output signal Wr from 0 to 1 to indicate the validity of both the address and the data to device B.
4. Device B retrieves the address and value from its input signals and stores the value.
5. Device B changes its output signal ACK from 0 to 1.
6. Device A changes its output signal Wr from 1 back to 0.
7. Device B changes its output signal ACK from 1 back to 0.

The protocols described are not the only way loads and stores can be implemented in hardware.

While these protocols could be implemented with simple binary valued signals as discussed so far, doing so would be very inefficient. For example, on a 32-bit architecture (with 32-bit addresses and 32-bit data words), a single load as described above would require at least 68 function calls. This is particularly important because these operations occur frequently in real hardware.

To avoid a significant decrease in execution speed, loads and stores are not implemented using simple signals in PCE. Instead a new set of receive functions is defined and used for this special case. This way the whole protocol for both loads and stores is condensed into a single function call.

Stores are handled similarly to simple signals except that the address is passed as an additional parameter to the receive function and that the value is no longer restricted to 0 or 1. Because stores can be of different sizes, multiple receive functions are necessary, one for each size supported by a device. Typically (but not necessarily) a set of three functions for 8, 16 and 32 bits is provided, as in the example below.

Loads differ from stores in that the receive function now returns the loaded value instead of receiving it. For loads the term receive function is really misleading as the loaded value is passed from the receive function back to the caller.

The following example shows one possible extension to the device struct of device A to support the load and store operations depicted in figure 2 and figure 3. It also shows the corresponding receive functions of device B:

```
typedef struct {
  void *set_ext;
  void (*set8) (void *ext, unsigned long addr, unsigned char val);
  void (*set16) (void *ext, unsigned long addr, unsigned short val);
  void (*set32) (void *ext, unsigned long addr, unsigned long val);

  unsigned char (*get8) (void *ext, unsigned long addr);
  unsigned short (*get16) (void *ext, unsigned long addr);
  unsigned long (*get32) (void *ext, unsigned long addr);
} DevA_t;

void devb_set8 (DevB_t *devb, unsigned long addr, unsigned char val);
void devb_set16 (DevB_t *devb, unsigned long addr, unsigned short val);
void devb_set32 (DevB_t *devb, unsigned long addr, unsigned long val);

unsigned char devb_get8 (DevB_t *devb, unsigned long addr);
unsigned short devb_get16 (DevB_t *devb, unsigned long addr);
unsigned long devb_get32 (DevB_t *devb, unsigned long addr);
```

2.2.4 Memory and Memory Mapped IO

The entire memory subsystem, as seen by individual devices, is just another device. However, since it is a special device and as it serves as a connecting point between many independant devices by means of memory mapped IO it deserves closer inspection.

The basic unit in the memory subsystem, as implemented by PCE, is the memory block. A memory block is defined by a physical base address and size in bytes. The entire memory subsystem is then just a list of memory blocks, each representing a part of the whole address space.

A memory block provides receive functions for loads and stores as discussed in section 2.2.3. How these receive functions handle loads and stores depends on the type of the memory block. There are two types:

1. Memory blocks that represent actual memory. These blocks contain an array of bytes of the appropriate size. Loads and stores are performed directly on this array.
2. Memory blocks that represent memory mapped devices. These blocks contain a pointer to the device's device struct as well as to the device's load/store receive functions. They simply call the device's recive function with the same parameters that they received, thereby passing the request on to the device.

Typically devices that access the memory subsystems are not connected directly to a specific memory block. Instead, there is another set of load/store receive functions for the entire memory subsystem. These functions find the memory block corresponding to the request's address and pass the request on to that memory block. A device that is connected to these receive functions doesn't need to know about memory blocks since it sees the memory subsystem as one linear address space. An example of this situation is shown in figure 5. The CPU calls the memory subsystem's receive function (a). Based on the request address the memory subsystem determines that the referenced address belongs to memory block C and calls that block's receive function (b). The memory block, representing a memory mapped device, calls that device's receive function (c).

All memory blocks combined need not cover the entire address space represented by the memory subsystem. Stores to an address that does not correspond to any memory block will be silently ignored. Loads from such an address will return a default value.

The memory subsystem is shown in figure 4. The figure shows three devices DevA, DevC and DevN that are represented by three memory blocks BlkA, BlkC and BlkN in the memory subsystem. Memory block BlkB represents RAM. The device CPU accesses the memory subsystem and sees it as a linear address space. The double arrows between the CPU and the memory subsystem and between the memory blocks and the devices represent access functions as discussed in section 2.2.3. In the figure memory blocks BlkA, BlkC and BlkN represent memory mapped devices DevA, DevC and DevN. Memory block BlkB represents actual RAM.

2.2.5 Clocking

A very important input signal for all devices is the clock signal. This signal is the primary means for a device to initiate state transitions.

In reality the clock signal is a signal that periodically changes from 0 to 1 and back to 0. In the PCE emulator framework clock signals are not implemented as ordinary signals. Instead the signal receive function is simplified by not including the signal value as a parameter. A single call to a clock signal receive function is to be interpreted as one full clock cycle, thus saving one function call per clock cycle.

Additionally some devices support triggering multiple clock cycles with a single call to the clock signal receive function by including a count parameter. A call to such a receive function with a

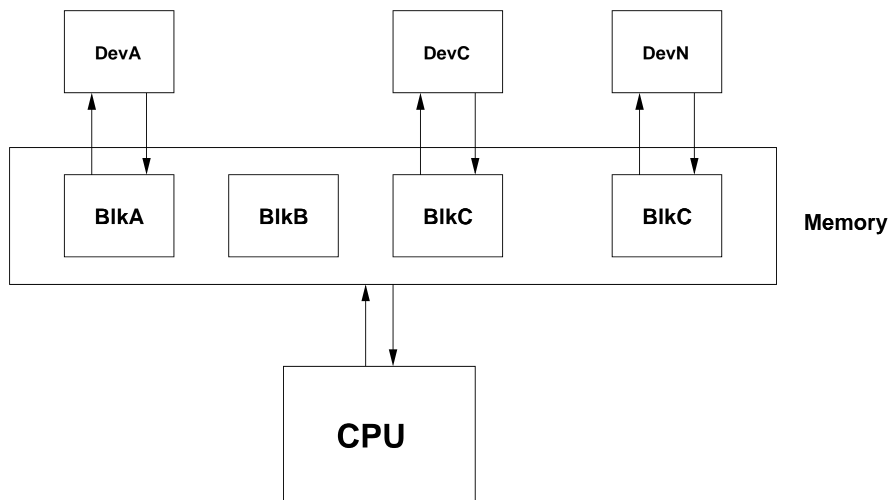


Figure 4: The memory subsystem

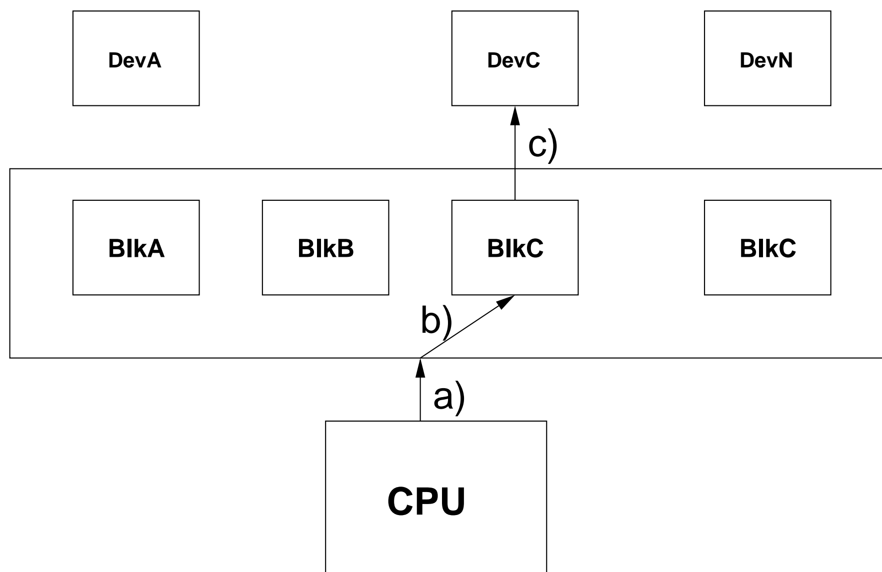


Figure 5: An example memory access

count value of n is equivalent to calling the function n times with a count value of 1. This feature is rarely used in practice because it is usually not permitted by the specification of the emulated hardware to clock a device more than once without clocking the rest of the system in between.

The clock signal is usually not sent to a device by another device but rather by the emulator main program. In fact, after initialization, all that the emulator main program does is calling each device's clock signal receive function in turn.

Not all devices need a clock signal, even if the real hardware they emulate does. Many devices simply react immediately to changes in their input signals by changeing some of their output signals. In those cases, the necessary state transitions can be made directly in the signal receive functions, leaving the clock function empty and removing the need for an extra clock signal.

3 The PowerPC 405 Emulator

The PowerPC 405 emulator is implemented as a PCE device as described in section 2.2.1. As such it has a device context struct as well as a set of signal receive functions, including one for clock signals.

This is the PowerPC 405's device context struct. Some non-essential fields have been omitted here for brevity:

```
typedef struct {
  /* memory access functions */
  void                 *mem_ext;
  p405_get_uint8_f     get_uint8;
  p405_get_uint16_f    get_uint16;
  p405_get_uint32_f    get_uint32;
  p405_set_uint8_f     set_uint8;
  p405_set_uint16_f    set_uint16;
  p405_set_uint32_f    set_uint32;

  /* registers */
  uint32_t             pc;
  uint32_t             gpr[32];

  p405_tlb_t           tlb;

  unsigned long        delay;

  /* opcode tables */
  p405_opcode_map_t    opcodes;
} p405_t;
```

Field description:

- mem_ext: A pointer to the memory device struct. This is used only as a parameter for the memory access functions.
- get_uint8, get_uint16, get_uint32, set_uint8, set_uint16, set_uint32: The memory access functions. These functions are used for all memory accesses by the PowerPC.
- pc: The program counter register
- gpr: The 32 general purpose registers
- tlb: The translation lookaside buffer. This structure is described below in section 3.3.
- delay: Contains the number of clock cycles before the next instruction is executed.
- opcodes: The table of opcode functions. This structure and opcode functions are described in section 3.1.

3.1 Instruction execution

Instruction execution in the PCE PowerPC emulator is performed by *opcode functions*. An opcode function receives a pointer to the PowerPC device struct as parameter and executes its instruction by updating the current device state as appropriate.

In addition to the updates needed to simulate the instruction being executed, an opcode function is expected to add the number of clock cycles that the current instruction needs to execute to the delay field of the device struct.

Here is an example of a no-op opcode function for the PowerPC 405. The instruction takes one clock cycle to execute and its only function is to advance the program counter by four (the size of the instruction word):

```
void p405_nop (p405_t *c)
{
  c->delay += 1;
  c->pc += 4;
}
```

In general, there is one opcode function for each opcode. The opcode is encoded in the instruction word referenced by the Program Counter (PC). Figure 6 shows the organization of a PowerPC instruction word.

Figure 6: A PowerPC instruction word. P is the primary opcode field, S is the secondary opcode field.

The 6 bit primary opcode field is present in all instruction words. Interpretation of the remaining bits depends on the value found in the primary opcode field. For certain values of the primary opcode field (for example 19, 31) there is a 10 bit secondary opcode field to further specify the instruction. For a complete list of PowerPC 405 opcodes see [2] and [3].

Instruction execution then works like this:

1. The instruction word is fetched from memory using the PC register as address. The instruction word is stored in the device struct for later use by the opcode function.
2. The primary opcode is extracted from the instruction word and used as an index into the 64-entry primary opcode table in the device struct. The entry found this way is a pointer to the opcode function for the primary opcode.
3. The opcode function is called. If the current instruction has no secondary opcode field then the opcode function executes the instruction directly.
4. If the current instruction has a secondary opcode field then the primary opcode function extracts the secondary opcode field and uses it as an index in the appropriate 1024-entry secondary opcode table in the device struct. The entry found this way is a pointer to a secondary opcode function. This function is called and executes the instruction. This two stage approach reduces the size of the opcode tables considerably.

The opcode tables in the PowerPC 405's device struct are defined like this:

```
typedef void (*p405_opcode_f) (struct p405_s *c);

typedef struct {
  p405_opcode_f op[64];
  p405_opcode_f op13[1024];
  p405_opcode_f op1f[1024];
} p405_opcode_map_t;
```

where the individual fields are:

- op: The opcode table for the 6-bit primary opcode

- op13: The opcode table for the 10-bit secondary opcode when the primary opcode is 19 (13 hexadecimal).
- op1f: The opcode table for the 10-bit secondary opcode when the primary opcode is 31 (1F hexadecimal).

3.2 Caches

The PowerPC 405 features separate caches for instructions and data. These caches are not emulated by the current implementation of the PCE PowerPC emulator. The absence of the caches has little impact on the software running on the emulated PowerPC because caches are, by design, largely transparent to software. The only difference that could be observed is that memory accessing instructions can take different amounts of time to execute depending on the presence or absence of caches.

Even though the caches themselves are not emulated, the instructions in the PowerPC 405 instruction set that access the caches directly need to be implemented. If they were not, executing such an instruction would cause an illegal instruction exception, which is inconsistent with the behaviour of the real hardware.

Fortunately, all but one of the cache manipulating instructions can, in the absence of cache units, be treated as no-ops. The one instruction that must be fully implemented is the dcbz (data cache block zero) instruction, which allocates and clears (sets to zero) a cache line. The opcode function for this instruction simply clears the bytes corresponding to the cache line directly in memory.

3.3 Memory Management Unit

The PowerPC 405 memory management unit (MMU) has been designed for simplicity. It consists of a 64-entry translation lookaside buffer (TLB) that holds all the information needed for memory protection and address translation.

For address translation, a TLB entry contains the logical and physical page addresses as well as the page size. The TLB may contain pages of different sizes at the same time.

For memory protection, a TLB entry contains bits that indicate write and execute permission. There is no bit for read permission as the presence of a valid TLB entry implies read permission.

Two fields of a TLB entry determine whether it is valid at all. The valid bit marks valid TLB entries. The TID field (transaction ID) is an 8 bit field. The TLB entry is considered valid only if its TID field is either 0 or the same as the PID (process ID) field in the processor's XER register. This feature is used to keep distinct sets of TLB entries (usually belonging to different processes) in the TLB and to make only one set valid without having to flush the others.

The PowerPC 405 TLB is, unlike with most other CPU's, managed completely by software. Any TLB miss causes a trap handler to be invoked. It is then up to that trap handler to update the TLB entries as appropriate and resume execution of the interrupted program.

A TLB entry is implemented in PCE as a struct. The entire TLB is implemented in two forms:

1. As an array of 64 TLB entries. This form is needed when TLB entries are referenced by index, which happens for example when the CPU updates an entry.
2. As a linked list of TLB entries. This form is used to search the TLB for addresses. The elements of the linked list are ordered such that the first element in the list is the entry that was most recently used. This is based on the assumption that TLB entries that were recently used have a higher probability of being used again in the near future.

```
typedef struct p405_tlbe_s {
  uint32_t           tlbhi;
  uint32_t           tlblo;
  uint8_t            tid;
  uint32_t           mask;
  uint32_t           vaddr;

  unsigned           idx;
  struct p405_tlbe_s *next;
} p405_tlbe_t;

typedef struct {
  p405_tlbe_t entry[P405_TLB_ENTRIES];
  p405_tlbe_t *first;
} p405_tlb_t;
```

The fields of a TLB entry are:

- tlbhi, tlblo: The 64-bit TLB entry as written into the TLB by the CPU. This is the entire TLB entry. The remaining fields of the TLB entry struct are extracted from this entry for easier (and faster) access.
- tid: The transaction ID
- mask: A bit mask that is used to convert a linear address to the page base address of the page that contains the linear address. The exact contents of this field depend on the page size. vaddr: The virtual page base address (i.e. the page base address before translation).
- idx: The index of this TLB entry in the TLB.

Note that the 64 TLB entries exist only once each. The same struct is an element both of the array as well as the linked list.

Other forms of organization, such as hash tables or binary search trees, might seem more appropriate at first sight.

Unfortunately it is not possible to implement the TLB as a hash table because the TLB can contain entries that represent pages of different sizes. It is not possible to transform a random address into the base address of the page that contains that address without knowing the size of that page. But if the TLB were implemented as a hash table, that base address would be needed to compute the hash value.

A binary search tree might, indeed, improve performance slightly. However, it comes at the cost of considerable overhead. Things are complicated by the fact that at any time the TLB can contain a random combination of invalid, valid and valid but not applicable entries. An entry that is valid but not applicable is one whose PID does not match the current PID. All in all, the minor performance gain that a binary search tree might provide is not worth the extra complexity introduced by it.

The implementation of the PowerPC 405 MMU provides two sets of functions to access memory. The first set is used by the opcode functions to load and store data parameters as well as by the execution unit to fetch instructions:

```
int p405_ifetch (p405_t *c, uint32_t addr, uint32_t *val);
int p405_dload8 (p405_t *c, uint32_t addr, uint8_t *val);
int p405_dload16 (p405_t *c, uint32_t addr, uint16_t *val);
int p405_dload32 (p405_t *c, uint32_t addr, uint32_t *val);
int p405_dstore8 (p405_t *c, uint32_t addr, uint8_t val);
int p405_dstore16 (p405_t *c, uint32_t addr, uint16_t val);
int p405_dstore32 (p405_t *c, uint32_t addr, uint32_t val);
```

These functions perform address translation (if enabled in the MSR) on the address and check the memory access for privilege violations. If the memory access is illegal for any reason, these functions cause a trap to be executed and return nonzero to indicate that execution of the current instruction should be aborted.

The second set of functions is used to inspect the CPU's memory without interfering, for example for debugging purposes:

```
int p405_get_xlat8 (p405_t *c, uint32_t addr, unsigned xlat, uint8_t *val);
int p405_get_xlat16 (p405_t *c, uint32_t addr, unsigned xlat, uint16_t *val);
int p405_get_xlat32 (p405_t *c, uint32_t addr, unsigned xlat, uint32_t *val);
int p405_set_xlat8 (p405_t *c, uint32_t addr, unsigned xlat, uint8_t val);
int p405_set_xlat16 (p405_t *c, uint32_t addr, unsigned xlat, uint16_t val);
int p405_set_xlat32 (p405_t *c, uint32_t addr, unsigned xlat, uint32_t val);
```

These functions do not check for protection violations but allow the access in all cases. Whether or not address translation should be done is not determined by the current state of the cpu but by the extra parameter xlat. These functions only fail if address translation is requested but the requested address cannot be found in the TLB. Even when they fail, they will never alter the device context.

3.4 Traps, Interrupts and Exceptions

The PowerPC emulator provides one function for each supported interrupt source. Calling this function will change the device state, forcing the CPU to execute the appropriate trap handler. These functions are typically called by opcode functions or by memory access functions whenever an interrupt condition is detected. After calling one of the interrupt functions, the current instruction should be aborted without further changing the device context. The next instruction executed will then automatically be the first instruction of the trap handler.

3.5 Differences

The following list summarises differences between the current PowerPC emulator implementation and a real PowerPC 405. Most of the differences are omissions rather than real differences:

- Both the instruction cache unit and the data cache unit are not implemented in the emulator. It has already been mentioned that this is largely transparent for software because the cache manipulating instructions are implemented as no-ops. A difference that software might observe is, that some of these instructions could cause traps if not used properly by the software. The emulated cache manipulating instructions will never cause traps.
- Both the data and instruction shadow TLBs are not implemented. The shadow TLBs hold copies of the most recently used TLB entries. Since they are truly transparant, even to system software, this has no impact on software running on the emulator.
- Not all special purpose registers of the PowerPC 405 are emulated. The emulator only supports those registers that are needed for normal operation. Not supported are in particular all debugging related registers and instructions. Other registers, such as the Core Configuration Register (CCR) are implemented, but writing it has no effect (if they weren't implemented at all, writing them would cause an illegal instruction exception).
- Some on-chip peripherals are not yet emulated. In particular, the PCI bus as well as the Processor Local Bus (PLB) are not yet supported.
- No floating point instructions are supported. This itself is not a difference as the real PowerPC 405 does not support them either, but because these instructions are not implemented at all, they cause an illegal opcode exception when actually they should cause a

floating point unit not present exception. This prevents an operating system from emulating the floating point instructions as is usually done. The simple remedy is to add opcode functions for all floating point instructions that do nothing but cause the appropriate exception.

- Machine check exceptions are not supported. They are not needed for normal operation as they are used to indicate hardware malfunctions to the operating system. They could be useful for testing purposes though.

4 The Embedded Processor Complex Emulator

This section describes the Embedded Processor Complex (EPC) emulator. Figure 7 shows an overview of the EPC. The key components are Dyadic Protocol Processors (DPPU), Core Language Processors (CLP) and a variety of coprocessors (COP). These components, their interaction and their implementation will be discussed in the following sections.

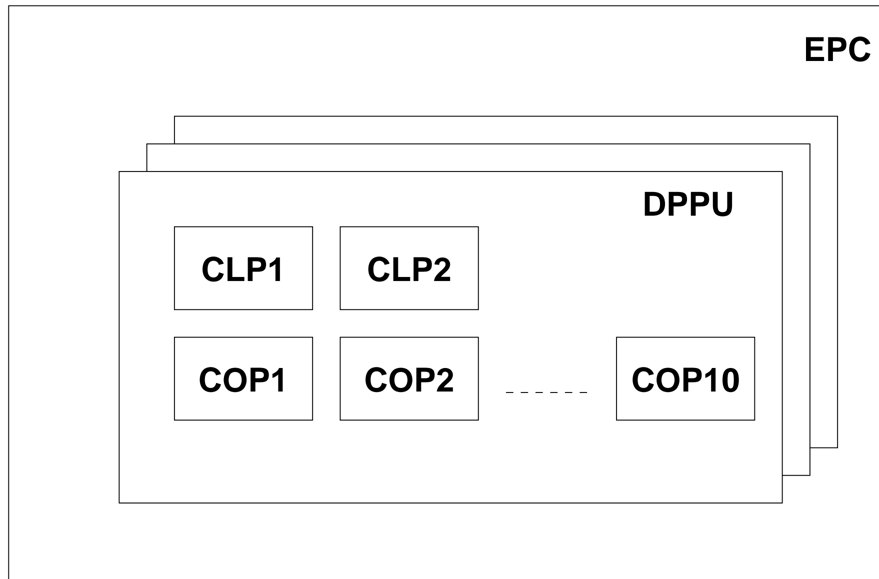


Figure 7: The Embedded Processor Complex

Figure 8 shows a single DPPU. It contains two CLP's and ten coprocesors. Each CLP has resources to run two threads. Threads T0 and T1 run on CLP0, threads T2 and T3 run on CLP1. Each thread appears to own 10 coprocessors but COP1 - COP10 exist only once each and are shared by the threads. Each coprocessor has resources for four threads (indicated in the figure by the dotted duplication of the coprocessors) giving each thread the illusion that it owns ten coprocessors of its own.

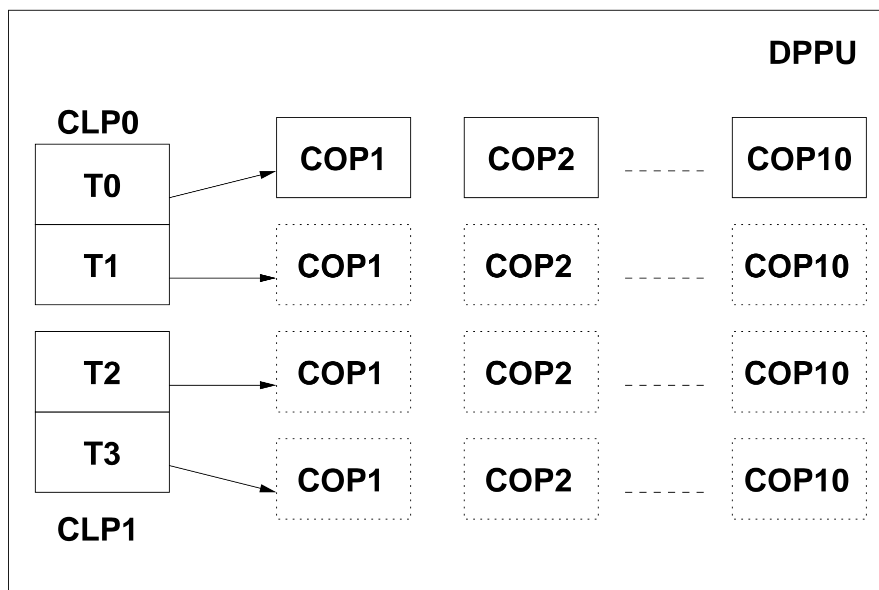


Figure 8: The Dyadic Protocol Processor

4.1 Core Language Processor

The Core Language Processor (CLP) is a picoprocessor designed specifically for packet processing. It executes picocode that contains the logic for packet handling in the NP4. Therefore, the CLP is to the EPC what the microprocessor is to a regular computer system.

In terms of instruction execution and the available instructions the CLP is not unlike any microprocessor. It features 16 32-bit general purpose registers along with a set of arithmetic and logical instructions comparable to other microprocessors.

Each CLP has resources for two parallel threads. Because all resources are physically present twice, thread switching can be done with zero overhead. Still, at most one thread can be executing at any time.

The CLP is implemented as a PCE device. Here is the device struct, with non-essential fields removed for brevity:

```
typedef struct np4_clp_s {
  /* instruction memory access function */
  void               *imem_ext;
  clp_get_uint32_f   ifetch;

  /* contexts for two threads */
  np4_cth_t          threads[2];
  unsigned           priority;

  /* opcode table */
  clp_opcode_f       opcodes[256];
} np4_clp_t;
```

Field description:

- imem_ext: A pointer to the instruction memory device struct.
- ifetch: The instruction memory access function used to fetch instruction words.
- threads: The device context structs for two threads.
- priority: Indicates the thread that has priority.
- opcodes: The table of opcode function pointers.

Note that since a CLP can only read its instruction memory in words of 32 bits and not write it at all, a full set of memory access functions is not required for the instruction memory.

Because the CLP's instruction set is somewhat simpler and smaller than that of the PowerPC 405, the opcode table in the device struct contains pointers that, in all cases, point directly to the opcode function for that opcode. The most significant 8 bits of the instruction word are used as index into the opcode table.

A large part of the CLP context is contained in the CLP Thread (CTH) struct. Logically this struct is part of the CLP device context. It is extracted into a separate struct for convenience only:

```
typedef void (*clp_cop_exec_f) (void *ext, unsigned cn,
  unsigned tn, unsigned op,
  unsigned long parhi, unsigned long parlo
);

typedef unsigned short (*clp_cop_get_busy_f) (void *ext, unsigned tn);

typedef unsigned short (*clp_cop_get_ok_f) (void *ext, unsigned tn);

typedef struct np4_cth_s {
  /* thread state */
  unsigned            state;

  char                priority;

  /* coprocessor access functions */
  void                *cop_ext;
  clp_get_uint8_f     cop_get8;
  clp_get_uint16_f    cop_get16;
  clp_get_uint32_f    cop_get32;
  clp_set_uint8_f     cop_set8;
  clp_set_uint16_f    cop_set16;
  clp_set_uint32_f    cop_set32;

  /* coprocessor exec and status functions */
  clp_cop_exec_f      cop_exec;
  clp_cop_get_busy_f  cop_get_busy;
  clp_cop_get_ok_f    cop_get_ok;

  /* registers */
  uint16_t            pc;
  uint16_t            flags;
  uint32_t            gpr[16];

  uint16_t            wait_mask;

  struct np4_clp_s    *clp;
} np4_cth_t;
```

Field description:

- state: The thread state. Possible thread states are idle, running, waiting or error.
- priority: Indicates whether this thread is the thread that has priority on its CLP.
- cop_ext: A pointer to the coprocessor array device struct. Coprocessor arrays are discussed in section 4.1.2.
- cop_get8, cop_get16, cop_get32, cop_set8, cop_set16, cop_set32: The coprocessor access functions. These functions are used by the thread to access the distributed address space as described in section 4.1.1.
- cop_exec: The coprocessor exec function. The thread uses this function to execute coprocessor commands.
- cop_get_busy: This function returns the current state for all ten copressecors.

- cop_get_ok: This function returns the 1-bit return code of the last executed coprocessor instruction for all ten coprocessors.
- pc: The program counter
- flags: The ALU condition codes
- gpr: The general purpose registers.
- wait_mask: This field is used when the thread is waiting for coprocessor command completion. It contains a bit mask that indicates which coprocessors the thread is waiting for.

Note that the coprocessor access functions, used for accessing the distributed address space, are ordinary load/store functions as discussed in section 2.2.3. The extra information needed here is coded into the request address as shown in figure 10 and discussed in the following section.

The coprocessor exec function and the coprocessor status function are, however, not ordinary signal functions. They are designed specifically for the purpose of communication between the CLP and the coprocessor. This communication is discussed below in section 4.1.2.

4.1.1 Distributed Address Space

A major difference between the CLP and other microprocessors is the way that memory is accessed. The CLP does not have the usual load/store instructions that access a linear address space. Instead it has a rich set of instructions that access registers located in the coprocessors. The memory model implemented in the CLP is therefore a distributed address space with each coprocessor representing a part of the whole address space.

Each coprocessor covers 1276 memory locations. They are organized as 252 directly adressable locations (scalars) and 4 indirectly adressable arrays of 256 memory locations each. The memory locations covered by a coprocessor are shown in figure 9. S0 - S251 are 252 scalars, A0,0 - A3,255 are 4 arrays of 256 elements each.

Therefore, an adress, to the CLP, is not just a linear address, but consists of these components:

- The CLP thread number. This is necessary because each thread has its own distinct address space.
- The coprocessor that covers the memory location
- Whether the memory location is a scalar or part of an array
- If the memory location is a scalar, the scalar index
- If the memory location is in an array, the array number and the index into that array

In order to simplify the emulator implementation, all these components are mapped into a single linear address. The advantage of this mapping is, that now the same concept of load/store receive functions as discussed in section 2.2.3 and as used by the PowerPC emulator can be used for the distributed address space of the EPC. Figure 10 shows how the various address components are mapped into a linear address.

Note that the so constructed linear address space has many holes in it. Additionally not all coprocessors actually implement all the address locations allocated to them.

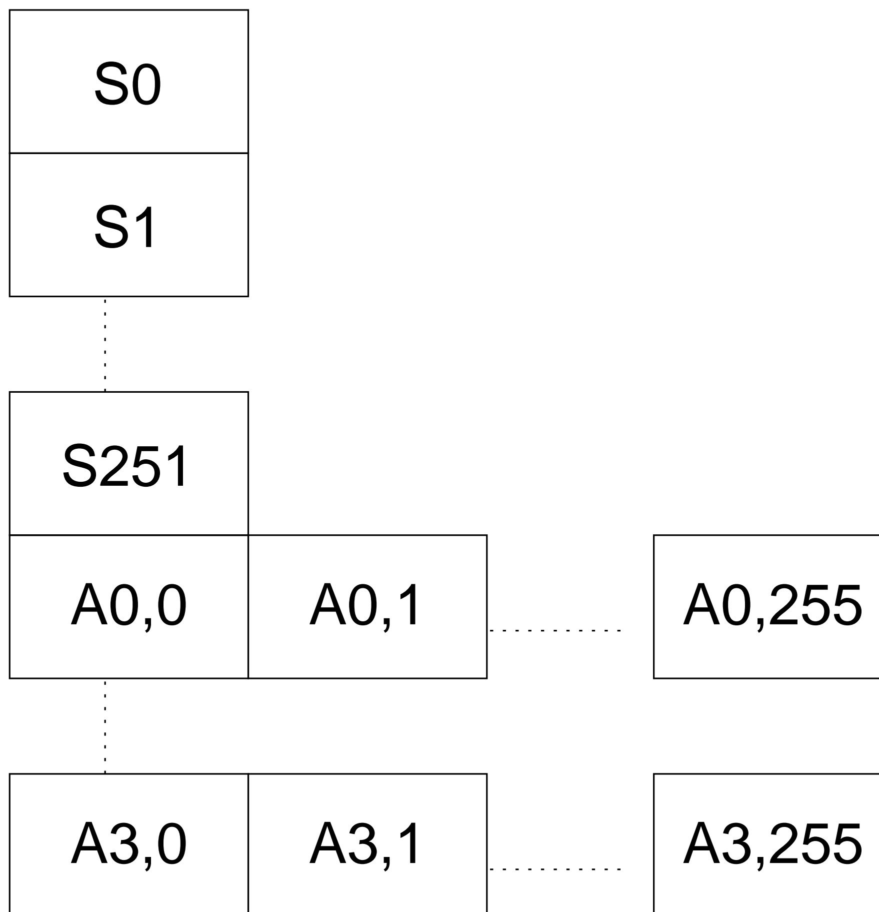


Figure 9: The memory locations covered by a single coprocessor

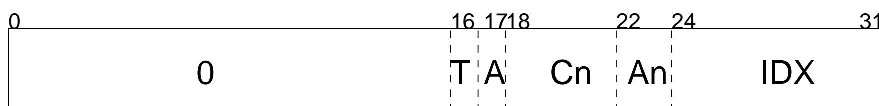


Figure 10: Encoding of the various parts of a CLP distributed memory address into a linear address. T is the thread number, A indicates that the adress refers to an array element and not to a scalar. Cn is the coprocessor number. An is the array number if an array is referenced and unused otherwise. IDX is either the scalar index (A=0) or the array index (A=1).

4.1.2 Coprocessor Command Execution

The ability to execute coprocessor commands is very important. Coprocessor commands extend a thread's capabilities by performing complex operations (such as tree searches, block data movements and checksum calculations) independant of (i.e. in parallel to) the execution of the thread.

The CLP has instructions to:

- Initiate command execution in a coprocessor. The command can be executed synchronously (the initiating CLP thread is stalled until the command completes) or asynchronously (the CLP threads continues immediately).
- Check if a coprocessor is busy or if it has finished executing an asynchronous command.
- Synchronize with a coprocessor by waiting for it to complete a command.
- After command completion, receive a 1-bit, command specific result code.

A coprocessor command consists of two parts:

1. A 16-bit command number. This tells the coprocessor which command is to be executed.
2. A 48-bit parameter. The contents of this parameter are command specific

In addition to the 48-bit parameter that is passed to the coprocessor when a command is executed, more parameters may be passed by writing them into coprocessor registers before the command is executed. What registers are used for this purpose (if any at all) is command specific.

A coprocessor can only execute one command at a time. It can, however, accept up to four commands, one from each thread. Commands that arrive while the coprocessor is executing another command are buffered and executed after the current command completes. This makes the sharing of coprocessors transparent to the CLP threads. The only consequence of two or more threads trying to execute a command on the same coprocessor at the same time is a longer delay for at least one of the threads.

The CLP can read the coprocessor status (busy or idle) of all ten coprocessors at any time out of its coprocessor status register. Note that this status is per thread so a coprocessor might appear idle to one thread even though it is currently executing a command for another thread. This status merely indicates whether the coprocessor is ready to receive a command from the thread reading the status.

Coprocessors are implemented as PCE devices. Because all ten coprocessors have some functionality in common, they share a common device context struct in the emulator:

```
typedef struct {
  cop_get_uint8_f   get8;
  cop_get_uint16_f  get16;
  cop_get_uint32_f  get32;

  cop_set_uint8_f   set8;
  cop_set_uint16_f  set16;
  cop_set_uint32_f  set32;

  cop_exec_f        exec;

  cop_clock_f       clock;

  struct np4_copa_s *copa;

  unsigned          cmd_cnt;
  unsigned          cmd_cur;
  cop_cmd_t         cmd[4];

  void              *ext;
} np4_cop_t;
```

Field description:

- get8, get16, get32, set8, set16, set32, exec, clock: These are all function pointers that point to the actual coprocessor's receive functions.
- copa: A pointer to the coprocessor array that this coprocessor is a part of.
- cmd_cnt: The number of coprocessor commands that are buffered in cmd.
- cmd_cur: The index of the current command in cmd.
- cmd: The command buffer. Additional commands that were accepted while the coprocessor was already executing a command are stored here.

This is a generic device struct. The exec and clock fields are function pointers that point to a specific coprocessor's signal receive functions. The ext pointer points to a coprocessor specific extension to the device struct. When a coprocessor is instantiated in the emulator, a generic coprocessor device struct and a coprocessor specific extension are allocated. These two together form the device context. Splitting the device struct in two parts is just a way to make coprocessor handling easier. There could just as well be ten different device structs albeit with considerable redundancy.

As an example of a coprocessor device struct extension, here is that of the string copy coprocessor:

```
typedef struct {
  uint16_t saddr;
  uint16_t daddr;
  uint16_t cnt;
} cop_sc_thread_t;

typedef struct {
  np4_cop_t       cop;

  cop_sc_thread_t th[4];
} cop_sc_t;
```

The ten coprocessors of a DPPU are further combined into a coprocessor array structure:

```
typedef struct np4_copa_s {
  uint16_t  busy[4];
  uint16_t  ok[4];

  np4_cop_t *cop[NP4_COP_CNT];
} np4_copa_t;
```

This structure is not motivated by the actual hardware. It is just convenient way to have the coprocessor status and result codes for all ten coprocessors and all four threads in one place. The CLP (which has a pointer to this coprocessor array in its device struct) uses it to access the coprocessors.

4.2 Differences

The following list summarises differences between the current EPC emulator implementation and a real EPC.

- Some CLP special purpose registers that are not essential for running software are not yet emulated. See below for a list.
- Most coprocessors (all but the Data Store and String Copy coprocessors) are not yet implemented. See below for an exact list of the missing coprocessors.
- The dispatch unit is not yet implemented.
- The hardware classifier is not yet implemented
- There are no ingress and egress data stores, even though the coprocessor to access them (Data Store Coprocessor) is already implemented.

The following coprocessors are not yet implemented:

- Tree Search Engine Coprocessor
- Control Access Bus Coprocessor
- Enqueue Coprocessor
- Checksum Coprocessor
- Policy Coprocessor
- Counter Coprocessor
- Semaphore Coprocessor

The following CLP special purpose registers are not yet implemented:

- TimeStamp: This register is implemented and increased monotonically. It is not increased at a rate of 1ms as required but rather every clock cycle.
- RandomNum, the random number generator
- IntVector0 - IntVector3
- IdleThreads
- QValid, the queue status
- My_TB, my target blade
- SW_Defined_A - SW_Defined_D
- Version_ID, the hardware version number

5 Evaluation

5.1 Verification

This section describes the procedures used to verify the implementation of the PowerPC and the EPC emulator.

Verifying software is an inherently difficult task. There is no known method for proving the correctness of a program in general, short of running the program with all possible inputs and comparing the output to the correct result. This is impossible in practice for any non-trivial program due to the enormous number of possible inputs.

The procedures described in this section serve to increase the confidence in the correctness of the emulator. They do not (and do not attempt to) prove that correctness.

The following steps were taken to verify the PowerPC 405 emulator:

1. During implementation each opcode function was immediately testet by running a short (3-4 assembly instructions) test program that contained that opcode. Execution of this test program was manually verified. Of course, this sort of crude testing insures the correctness of neither one particular opcode function nor of the emulator as a whole. It serves well, however, to catch many trivial errors (such as typing errors or copy and paste errors) that would be difficult and time consuming to fix after the implementation is complete.
2. After the PowerPC CPU core emulator was complete, more complicated test programs were run on it. Since there was no operating system running and no I/O devices implemented yet, these test programs focused mostly on integer computations. They were implemented in a mixture of assembly language and C (using gcc) and included prime number calculation, factorials, integer squareroots and MD5 (Message Digest 5) checksum calculations. Results were compared automatically with those produced by the same programs running under Linux on a PowerPC 603e. The subset of instructions used in these tests is known to be implemented identically on the PowerPC 603e and the PowerPC 405.
3. As a last step, the Linux operating system was run on the emulated PowerPC. The successful booting of Linux is an indication that the implementation is correct. To back up this assumption, small errors were introduced deliberately in the emulator implementation in order to observe their effect. Any of these errors prevented Linux from booting successfully. Without any (intentional) errors, Linux booted successfully and ran uninterrupted and without problems for periods of at least $5.0 * 10^{11}$ emulated clock cycles. It was insured (by counting the number of executions of all opcodes) that each opcode was executed at least once.

The EPC emulator is not yet complete and therefore can't be verified as a whole yet. In particular, the missing coprocessors prevent any non-trivial program from running successfully on the emulator. As a consequence the EPC emulator has not yet been extensively testet.

6 Summary

6.1 Implementation Status

This chapter summarises the goals achieved in this project.

The PowerPC 405 emulator. The PowerPC emulator is complete, save for the differences mention in section 3.5. It is completely independant of the rest of the emulator and ready for use in other projects.

Linux. An unmodified (both source and binary) Linux kernel 2.6.x is bootable on the emulated PowerPC system.

EPC. The EPC emulator is not yet complete but can run software that does not use the missing components mentioned in section 4.2.

Portability. The emulator has been compiled and testet under a wide range of hardware architectures, operating systems and C compilers.

6.2 Outlook

This section contains suggestions for improvements and extensions of the PowerNP 4GS3 emulator.

The next step should be to implement the missing EPC components mentioned in section 4.2. In particular, to allow communication between the PowerPC and the EPC the following components are required:

- The dispatch unit. This is required for interrupt triggering on the EPC.
- The Control Access Bus (CAB) Coprocessor. The CAB is used both for passing buffer addresses between the PowerPC and the EPC as well as for triggering interrupts (via the CAB doorbell registers).
- On the PowerPC side, the CAB is mapped into memory. A simple pseudo device (a device that does not correspond to a physical device) is required that intercepts accesses to the memory mapped CAB registers and passes the relevant values to the EPC.

To allow complete packet processing, the following additional components are required:

- The ingress and egress data stores. These need to be connected to the Data Store Coprocessor.
- The hardware classifier.
- The remaining coprocessors mentioned in section 4.2.

A Problem

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Wintersemester 2003

Diplomarbeit

für

Hanspeter Hug

Betreuer: Lukas Ruf
Stellvertreter: Matthias Bossardt

Ausgabe: 03.11.2003
Abgabe: 10.03.2004

Design and Implementation Of An Emulator For the IBM NP4GS3

---

1 Einführung

Die Entwicklung von Betriebssystemen erfordert Entwicklungsumgebungen, welche es gestatten, den Zustand des gesamten Systems zu jedem beliebigen Zeitpunkt in der Betriebssystem-Ausführung zu betrachten und nötigenfalls zu verändern. Ansätze, dieses Problem zu lösen, sind durch Plattformen mit hardware-seitiger Unterstützung für die Fehlersuche, wie z.B. dem JTAG [6] Interface, oder durch Software-Emulatoren gegeben, die gesamte Computer-Systeme nachbilden. Bekannte Beispiele der letztgenannten Gruppe sind vmware [11], bochs [1] und dessen Abkömmling Plex86 [8].
Am Institut für Technische Informatik und Kommunikationsnetze (TIK) der ETH Zürich (ETHZ) wurde im Rahmen der Diplomarbeit von Kaufmann [7] wurde ein Emulatorframework zusammen mit einer Beispielimplementation der MIPS-R3K Plattform entwickelt. Das vorliegende Emulatorframework wurde in C++ [9] implementiert. Die Architektur wurde so entwickelt, dass das Hinzufügen neuer Plattformen sehr einfach möglich sein soll.
Für Hochgeschwindigkeitsrouter werden heutzutage vermehrt sogenannte Netzwerkprozessoren eingesetzt (NP). Gewöhnlicherweise bestehen solche NPs aus einem Control Prozessor, spezialisierten Paket- und Co-Prozessoren, die zusammen auf einem einzigen Chip implementiert sind. Ein ihrer Vertreter ist der IBM NP4GS3 (NP4) [5], welcher mit einem PowerPC e405 Control Prozessor und 16 der zuvor erwähnten Paketprozessoren (sog. PicoProzessoren) aufgebaut ist. Im Rahmen der Arbeit zu PromethOS NP [2] wurde ein NP4 verwendet.
NPs werden in der Regel auf Netzwerkinterface Karten eingebettet. Im Falle von IBM wurde ein Simulator entwickelt, welcher gewisse Komponenten nachbildet, jedoch nicht das gesamte System emuliert. Die Entwicklung von Code und besonders dessen Debugging gestaltet sich somit besonders schwierig, da ausser professionellem JTAG-basierten Debugging keine Hilfsmittel zur Verfügung stehen.

2 Aufgaben: Design and Implementation Of An Emulator For the IBM NP4GS3

Im Rahmen dieser Diplomarbeit soll eine IBM NP4GS Emulation entwickelt werden, die auf dem vorhandenen Emulatorframework aufsetzt. Die Emulation soll in der Lage sein, das für den NP4 vorliegende Linux auszuführen. Besonderes Augenmerk soll auf die Verwendung der PicoProzessoren gelegt werden.
Die Emulation soll die Möglichkeit bieten, die (Weiter-)Entwicklung von Betriebssystemen auf geeignete Weise zu unterstützen. Wenn möglich, soll ein Interface zum weitverbreiteten gdb [3] der GNU Tools oder ein JTAG-kompatibles Interface entwickelt werden.

3 Vorgehen

- Richten Sie sich eine Entwicklungsumgebung (GNU Tools) unter Linux ein.
- Machen Sie sich vertraut mit den Unterlagen (Dokumentation und Source Code) zum Emulatorframework, der vorhandenen Beispielimplementation der MIPS-R3K Emulation und dem Netzwerkprozessor.
- Erstellen Sie einen Zeitplan, in welchem Sie die von Ihnen zu erreichenden Meilensteine Ihrer Arbeit identifizieren.
- Entwickeln Sie eine Architektur für den zu erstellenden Emulator. Beachten Sie die Multiprozessoreigenschaften eines Netzwerkprozessors sowie dessen Netzwerkinterfaces im Speziellen.
- Implementieren Sie Ihre Architektur.
- Verifizieren, evaluieren und demonstrieren Sie das Erreichte durch eine Beispielapplikation.
- Dokumentieren Sie die Resultate ausführlich.

optional Entwickeln Sie ein Interface zum GDB, welches sich in das Emulatorframework einfügt.
optional Implementieren und testen Sie dieses Interface zum GDB.
optional Dokumentieren Sie dieses Interface.

Auf eine klare und ausführliche Dokumentation wird besonders Wert gelegt. Es wird empfohlen, diese laufend nachzuführen und insbesondere die entwickelten Konzepte und untersuchten Varianten vor dem definitiven Variantenentscheid ausführlich schriftlich festzuhalten.

4 Organisatorische Hinweise

- Am Ende der zweiten Woche ist ein Zeitplan für den Ablauf der Arbeit vorzulegen und mit dem Betreuer abzustimmen.
- Mit dem Betreuer sind regelmässige, zumindest wöchentliche Sitzungen zu vereinbaren. In diesen Sitzungen sollen die Studenten mündlich über den Fortgang der Arbeit und die Einhaltung des Zeitplanes berichten und anstehende Probleme diskutieren.
- Am Ende des ersten Monates muss eine Vorabversion des Inhaltsverzeichnis zur Dokumentation dem Betreuer abgegeben und mit diesem besprochen werden.
- Nach der Hälfte der Arbeitsdauer soll ein kurzer mündlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt. Dieser Zwischenbericht besteht aus einer viertelstündigen, mündlichen Darlegung der bisherigen Schritte und des weiteren Vorgehens gegenüber Professor Plattner.
- Am Schluss der Arbeit muss eine Präsentation von **20 Minuten** im Fachgruppen- oder Institutsrahmen gegeben werden. Anschliessend an die Schlusspräsentation soll die Arbeit Interessierten praktisch vorgeführt werden.
- Die Arbeit muss regelmässig auf dem CVS-Server cvs.topsy.net gesichert werden. Es ist darauf zu achten, dass die **richtige CVS-Branch** verwendet wird.

- Der Topsy [10] Coding Style muss eingehalten werden.
- Bereits vorhandene Software kann übernommen und gegebenenfalls angepasst werden.
- Die Dokumentation ist mit dem Satzsystem LaTeX zu erstellen.
- Es ist ein mit Bindespiralen gebundener Schlussbericht (am TIK vorhanden) über die geleisteten Arbeit abzuliefern (4 Exemplare). Dieser Bericht besteht aus einer Zusammenfassung , einer Einleitung, einer Analyse von verwandten und verwendeten Arbeiten, sowie einer vollständigen Beschreibung der Konfiguration von den eingesetzten Programmen. Der Bericht ist in Deutsch oder Englisch zu halten. Die Zusammenfassung muss in Deutsch und Englisch verfasst werden.
- Die Arbeit muss auf CDROM archiviert abgegeben werden. Stellen Sie sicher, dass alle Programme sowie die Dokumentation sowohl in der lauffähigen, resp. druckbaren Version als auch im Quellformat vorhanden, lesbar und verwendbar sind.
  Mit Hilfe der abgegebenen Dokumentation muss der entwickelte Code zu einem ausführbaren Programm erneut übersetzt und eingesetzt werden können.
- Diese Arbeit steht unter der GNU General Public License (GNU GPL) [4].
- Für die Arbeit werden Informationen eingesetzt, welche nur durch das Unterzeichnen eines NDA (Non-Disclosure Agreement) mit IBM Corp. erhalten wurden. Die Arbeit als ganzes (Programmcode und Dokumentation) sowie alle Informationen, die unter das NDA fallen, dürfen nur an Dritte weitergegeben werden, wenn eine schriftliche Einwilligung von IBM Corp. vorliegt.
- Diese Arbeit wird als Diplomarbeit an der ETH Zürich durchgeführt. Es gelten die Bestimmungen hinsichtlich Kopier- und Verwertungsrechte der ETH Zürich.

Literatur

[1] Bochs. https://sourceforge.net/projects/bochs, 1998-2003.
[2] P. Erni. *Einsatz und Programmierung des IBM NP4GS3 Netzwerkprozessors für Aktive Netzwerkknoten unter Linux.* TIK, ETH Zurich, 2003.
[3] GDB, the GNU symbolic debugger. http://www.gnu.org, 2001.
[4] GNU General Public License v2. http://www.gnu.org/copyleft/gpl.html, June 1991.
[5] IBM Corp. IBM PowerNP NP4GS3 Databook. http://www.ibm.com, 2002.
[6] IEEE-1149. JTAG. http://grouper.ieee.org/groups/1149, 1997-2003.
[7] F. Kaufmann. *Design And Implementation Of A Modular Emulator For Topsy.* TIK, ETH Zurich, 2003.
[8] plex86. https://sourceforge.net/projects/plex86, 2001-2003.
[9] B. Stroustrup. *C++ Programmming Language.* Addison Wesley, 3rd edition, 1997.
[10] The Topsy Core Team. Topsy coding style. http://www.topsy.net/Standards, 2002.
[11] vmWare. http://www.vmware.com, 1998-2003.

Zürich, den 03.11.2003

B Core Language Processor Opcodes

This opcode list was extracted from [1] and [4].

```
cond mnemonic flags description
0000 e, z Z = 1 equal or zero
0001 ne, nz Z = 0 not equal or not zero
0010 c, ae C = 1 carry set, above or equal
0011 a C = 1 AND Z = 0 unsigned higher
0100 be C = 0 OR Z = 1 unsigned lower or equal
0101 nc, b C = 0 unsigned lower
0110 --Reserved
0111 -Don't Care Always
1000 p N = 0 signed positive
1001 n S = 1 signed negative
1010 ge N = V signed greater or equal
1011 g Z = 0 AND N = V signed greater than
1100 le Z = 1 OR (N != V) signed less than or equal
1101 l N != V signed less than
1110 o V = 1 overflow
1111 no V = 0 no overflow


==============================================================================
Control Opcodes
==============================================================================


------------------------------------------------------------------------------
Nop
------------------------------------------------------------------------------

The nop opcode executes one cycle of time and does not change any state
within the processor.


mnemonics

nop


opcode

mask   = FC000001
opcode = 08000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 1 0|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+


------------------------------------------------------------------------------
Exit
------------------------------------------------------------------------------

The exit opcode terminates the current instruction stream. The CLP will be
put into an idle state and made available for a new dispatch. The exit
```

```
command is executed conditionally, and if the condition is not true, it will
be equivalent to a nop opcode.


mnemonics

exit <c>


pseudocode

IF cond THEN
clp_state_machine <- idle
ELSE
nop
END IF


opcode

mask   = F8000000
opcode = 10000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 1 0 0|0 0|  cond |0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+


------------------------------------------------------------------------------
Test and Branch
------------------------------------------------------------------------------

The test and branch opcode tests a single bit within a GPR register. If the
desired condition is met, the thread will branch. The R field is the GPR
register to be tested. The bp field represents the bit position within the
GPR to be tested. The c field represents the test condition: if c = 1, the
branch is taken if the bit tested is a ‘1’; if c = 0, the branch is taken if
the bit tested is a ‘0’. The branch target ID is calculated by using the
disp16 field as a displacement from the program counter (PC). This command
does not set the ALU flags.


mnemonics

t0bit reg, #n, target
t1bit reg, #n, target


pseudocode

IF R(bp) = c THEN
PC <- PC + disp16
ELSE
nop
END IF
```

```
opcode

mask   = FC000000
opcode = 14000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 1 0 1|c|    bp   |   R   |           disp16              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
------------------------------------------------------------------------------
Branch And Link
------------------------------------------------------------------------------
```

The branch and link opcode performs a conditional branch, adds one to the value of the current program counter, and places it onto the program stack. Opcode fields Rt, h, and target16 determine the destination of the branch. If Rt is in the range of 1 - 15, it is the word address of a GPR register, the contents of which are used as the base of the branch destination. If Rt is 0, the base of the branch destination will be 0x0000. The h field indicates which half of the GPR register is used as the base address. An h = 0 indicates the even half of the GPR register (high half of the GPR) and h = 1 indicates the odd half. The target16 field is added to the base to form the complete branch destination. If the LinkPtr register indicates that the Branch and Link will overflow the 16-entry program stack, a stack error occurs.

mnemonics

```
bal <c> target
bal <c> reg
bal <c> reg + target
```

pseudocode

```
IF cond THEN
; put IA+1 onto the program stack
IF LinkPtr = EndOfStack THEN
StackErr <- 1
ELSE
ProgramStack(LinkPtr + 1) <- PC + 1
END IF

; load the IA with the branch target
IF Rt = 0 THEN
PC <- target16
ELSIF h = 0 THEN
PC <- GPR(Rt)(31:16) + target16
ELSE
PC <- GPR(Rt)(15:0) + target16
END IF
ELSE
nop
END IF
```

```
opcode

mask   = FB000000
opcode = 1B000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 1 1|h|1 1|  cond |   Rt  |           target16            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Return

The return opcode performs a conditional branch with the branch destination being the top of the program stack. If the LinkPtr register indicates that the stack is empty, a stack error occurs.

mnemonics

```
ret <c>
```

pseudocode

```
IF cond THEN
IF (LinkPtr = EmptyStack) THEN
StackErr <- 1
ELSE
PC <- ProgramStack(LinkPtr)
END IF
ELSE
nop
END IF
```

```
opcode

mask   = FF000010
opcode = 18000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 1 1 0 0 0|  cond |0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0|0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Branch Register

The branch register opcode performs a conditional branch. Opcode fields Rt and h determine the destination of the branch. The Rt field is the word address of a GPR register of which the contents will be used as the branch destination. The h field indicates which half of the GPR register will be

used. An h = 0 indicates the even half of the GPR register (high half of the GPR), and h = 1 indicates the odd half.

mnemonics

```
b <c> r16
```

pseudocode

```
IF cond THEN
IF h = 0 THEN
PC <- GPR(Rt)(31:16)
ELSE
PC <- GPR(Rt)(15:0)
END IF
ELSE
nop
END IF
```

opcode

```
mask   = F8000010
opcode = 18000010

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 1 1|h|0 0|  cond |   Rt  |0 0 0 0 0 0 0 0 0 0 0|1|0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Branch PC Relative

The branch PC relative opcode performs a conditional branch. Opcode fields PC and disp16 determine the destination of the branch. The contents of the PC field are used as the base of the branch destination. The disp16 field will be added to the base to form the complete branch destination.

mnemonic

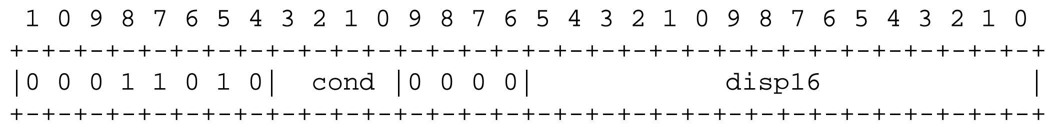
```
b <c> target
```

pseudocode

```
IF cond THEN
PC <- PC + disp16
ELSE
nop
END IF
```

opcode

```
mask   = FF000000
opcode = 1A000000
```



```
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 1 1 0 1 0|  cond |0 0 0 0|            disp16             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Branch Register + Offset

The branch register plus offset opcode will perform a conditional branch. The Opcode fields Rt and target16 determine the destination of the branch. If Rt is in the range of 1 - 15, it is the word address of a GPR register of which the contents of the high halfword (or even half) are used as the base of the branch destination. If Rt is 0, the base of the branch destination is x0000. The target16 field is added to the base to form the complete branch destination.

mnemonics

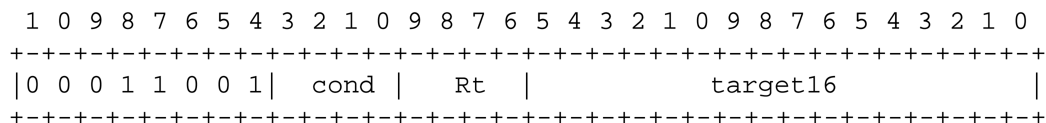
```
b <c> r16 + imm16
```

pseudocode

```
IF cond THEN
IF Rt = 0 THEN
PC <- target16
ELSE
PC <- GPR(Rt)(31:16) + target16
END IF
ELSE
nop
END IF
```

opcode

```
mask   = FF000000
opcode = 19000000
```



```
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 1 1 0 0 1|  cond |   Rt  |           target16            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

==

Data Movement Opcodes

==

ot5 GPR array type
00 half 1 half load

```
01 half 0 half load
02 word zext32(half) load
03 word sext32(half) load
04 word zext32(byte) load
05 word sext32(byte) load
06 word word load
08 byte 3 byte load
09 byte 2 byte load
0A byte 1 byte load
0B byte 0 byte load
0C half 1 zext16(byte) load
0D half 1 sext16(byte) load
0E half 0 zext16(byte) load
0F half 0 sext16(byte) load
10 half 1 half store
11 half 0 half store
16 word word store
18 byte 3 byte store
19 byte 2 byte store
1A byte 1 byte store
1B byte 0 byte store
```

Memory Indirect

The memory indirect opcode transfers data between a GPR and a coprocessor array via a logical address in which the base offset into the array is contained in a GPR. The logical address consists of a coprocessor number, coprocessor array, base address, and an offset. The C# field indicates the coprocessor which will be accessed. The Ca field indicates which array of the coprocessor will be accessed. The offset into the array is the contents of GPR, indicated by Ra as a base plus the six bits of immediate offset, off6. Ra is a half-word address in the range of 0 - 7. The x indicates whether or not the transaction will be in cell header skip mode. The memory indirect opcode is executed conditionally if the ALU flags meet the condition represented by the Cond field. The word address of the GPR register used to transfer or receive data from the array is indicated by the field R. The actual direction, size, extension format, and location of GPR bytes affected are indicated by the ot5 field.

mnemonics

```
ldr   <c> wreg, array[hreg + off6]
ldr   <c> hreg, array[hreg + off6]
ldr   <c> breg, array[hreg + off6]
ldrb  <c> wreg, array[hreg + off6]
ldrb  <c> hreg, array[hreg + off6]
ldrbs <c> wreg, array[hreg + off6]
ldrbs <c> hreg, array[hreg + off6]
ldrh  <c> wreg, array[hreg + off6]
ldrhs <c> wreg, array[hreg + off6]
str   <c> array[hreg + off6], wreg
str   <c> array[hreg + off6], hreg
str   <c> array[hreg + off6], breg
```

```
pseudocode

IF cond THEN
addr.coprocessor number <= C#
addr.coprocessor array <= Ca
addr.array offset <= GPR(Ra) + off6
addr.x_mode <= x

IF ot5 = load THEN
GPR(R,ot5) <= array(addr)
ELSE
array(addr) <= GPR(R,ot5)
END IF
END IF


opcode

mask   = E0000000
opcode = 20000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 1|   ot5    |  cond |    R   |x|  Ra |    C#  | Ca|    off6   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+


------------------------------------------------------------------------------
Memory Address Indirect
------------------------------------------------------------------------------
```

The memory address indirect opcode transfers data between a GPR and a coprocessor data entity (scalar or array) by mapping the coprocessor logical address into the base address held in the GPR indicated by Ra. Since not all of the coprocessors have arrays, maximum size arrays, or the maximum number of scalars, the address map will have many gaps. Data access via this method is the same as access via the more logic-based method. The final address is formed by the contents of GPR indicated by Ra plus off8 (eight bits). Ra is a half-word address in the range of 0 - 7. The x indicates whether or not the transaction will be in cell header skip mode.

The memory address indirect opcode is executed conditionally if the ALU flags meet the condition represented by the Cond field. The word address of the GPR register used to transfer or receive data from the array is indicated by the field R. The actual direction, size, extension format, and location of GPR bytes affected are indicated by the ot5 field.

```
mnemonics

ldr   <c> wreg, [hreg + off8]
ldr   <c> hreg, [hreg + off8]
ldr   <c> breg, [hreg + off8]
ldrb  <c> wreg, [hreg + off8]
ldrb  <c> hreg, [hreg + off8]
ldrbs <c> wreg, [hreg + off8]
ldrbs <c> hreg, [hreg + off8]
```

```
ldrh  <c> wreg, [hreg + off8]
ldrhs <c> wreg, [hreg + off8]
str   <c> [hreg + off8], wreg
str   <c> [hreg + off8], hreg
str   <c> [hreg + off8], breg
```

pseudocode

```
IF cond THEN
address <= GPR(Ra) + off8

addr.array_notScalar <= address(14)
addr.coprocessor number <= address(13:10)
addr.coprocessor array <= address(9:8) (if an array access)
addr.scalar address <= address(7:0) (if a scalar access)
addr.array offset <= address(7:0) (if an array access)
addr.x_mode <= x

IF ot5 = load THEN
IF addr.array_notScalar = 1 THEN
GPR(R,ot5) <= array(addr)
ELSE
GPR(R,ot5) <= scalar(addr)
END IF
ELSE
IF addr.array_notScalar = 1 THEN
array(addr) <= GPR(R,ot5)
ELSE
scalar(addr) <= GPR(R,ot5)
END IF
END IF
END IF
```

opcode

```
mask   = E0000000
opcode = 40000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 1 0|   ot5   |  cond |   R   |x|  Ra |0 0 0 0|      off8     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Memory Direct

The memory direct opcode transfers data between a GPR and a coprocessor array via a logical address that is specified in the immediate portion of the opcode. The logical address consists of a coprocessor number, coprocessor array, and an offset. The C# field indicates the coprocessor that is accessed. The Ca field indicates which array of the coprocessor is accessed. The offset is made up of eight bits: the most significant two bits are located in field off2 and the remaining six bits are in off6. The x indicates whether or not the transaction will be in cell header skip mode.

The memory direct opcode is executed conditionally if the ALU flags meet the condition represented by the Cond field. The word address of the GPR register used to transfer or receive data from the array is indicated by the R field. The actual direction, size, extension format, and location of GPR bytes affected are indicated by the ot5 field.

mnemonics

```
ldr   <c> wreg, array[off8]
ldr   <c> hreg, array[off8]
ldr   <c> breg, array[off8]
ldrb  <c> wreg, array[off8]
ldrb  <c> hreg, array[off8]
ldrbs <c> wreg, array[off8]
ldrbs <c> hreg, array[off8]
ldrh  <c> wreg, array[off8]
ldrhs <c> wreg, array[off8]
str   <c> array[off8], wreg
str   <c> array[off8], hreg
str   <c> array[off8], breg
```

pseudocode

```
IF cond THEN
addr.coprocessor number <= C#
addr.coprocessor array <= Ca
addr.array offset <= off2 || off6
addr.x_mode <= x

IF ot5 = load THEN
GPR(R,ot5) <= array(addr)
ELSE
array(addr) <= GPR(R,ot5)
END IF
END IF
```

opcode

```
mask   = E0004000
opcode = 60000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 1 1|   ot5   |  cond |    R    |x|0|of2|   C#  | Ca|   off6    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Scalar Access

The scalar access opcode transfers data between a GPR and a scalar register via a logical address that consists of a coprocessor number and a scalar register number. The C# field indicates the coprocessor that is accessed. The

scalar register number is indicated by the Cr field.

The scalar access opcode is executed conditionally if the ALU flags meet the condition represented by the Cond field. The word address of the GPR register used to transfer or receive data from the scalar register is indicated by the R field. The actual direction, size, extension format, and location of GPR bytes affected are indicated by the ot5 field.

mnemonics

```
ldr   <c> wreg, scalar
ldr   <c> hreg, scalar
ldr   <c> breg, scalar
ldrb  <c> wreg, scalar
ldrb  <c> hreg, scalar
ldrbs <c> wreg, scalar
ldrbs <c> hreg, scalar
ldrh  <c> wreg, scalar
ldrhs <c> wreg, scalar
str   <c> scalar, wreg
str   <c> scalar, hreg
str   <c> scalar, breg
```

pseudocode

```
IF cond THEN
addr.coprocessor number <= C#
addr.scalar number <= Ca

IF ot5 = load THEN
GPR(R,ot5) <= scalar(addr)
ELSE
scalar(addr) <= GPR(R,ot5)
END IF
END IF
```

opcode

```
mask   = E0004000
opcode = 60004000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 1 1|   ot5    |  cond |    R   |0|1|0 0|   C#  |       Cr      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Scalar Immediate

The scalar immediate opcode writes immediate data to a scalar register via a logical address that is completely specified in the immediate portion of the opcode. The logical address consists of a coprocessor number and a coprocessor register number. The C# field indicates the coprocessor that is

accessed. The Cr field indicates the scalar register number. The data to be written is the Imm16 field.

mnemonics

```
mov scalar, imm16
```

pseudocode

```
addr.coprocessor number <= C#
addr.scalar register number <= Ca
scalar(addr) <= Imm16
```

opcode

```
mask   = F0000000
opcode = D0000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 0 1|   C#  |       Cr      |             imm16             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Transfer Quadword

The transfer quadword opcode transfers quadword data from one array location to another using one instruction. The source quadword is identified by the C#s (coprocessor number), Cas (source array number), and QWoffs (quadword offset into the array). The destination quadword is identified by the C#d, Cad, and QWoffd. This transfer is only valid on quadword boundaries.

mnemonics

```
movm  arraydst, arraysrc
```

pseudocode

```
saddr.coprocessor number <= C#s
saddr.array number <= Cas
saddr.quadword offset <= QWoffs
daddr.coprocessor number <= C#d
daddr.array number <= Cad
daddr.quadword offset <= QWoffd

array(daddr) <= array(saddr)
```

opcode

```
mask   = F0000030
```

```
opcode = C0000010

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 0 0|  C#d  |Cad|0 0| QWoffd|0 0 0 0|  C#s  |Cas|0 1| QWoffs|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Zero Array

The zero array opcode zeroes out a portion of an array with one instruction. The size of the zeroed-out portion can be a byte, halfword, word, or quadword. Quadword access must be on quadword address boundaries. Accesses to a byte, halfword, or word can begin on any byte boundary and will have the same characteristics as any GPR-based write to the array. For example, if the array is defined to wrap from the end to the beginning, the zero array command wraps from the end of the array to the beginning. The C# field indicates the coprocessor that will be accessed. The Ca field indicates which array of the coprocessor is accessed. The QWoff is the quadword offset into the array, and the Boff is the byte offset into the array. The x indicates whether or not the transaction is in cell header skip mode.

The opcode is executed conditionally if the ALU flags meet the condition represented by the Cond field. The actual size of the access is defined by the size field (byte = 00, halfword = 01, word = 10, quadword = 11). For quadword accesses Boff should equal 0x0.

mnemonics

```
zarray  <c> array
zarrayb <c> array
zarrayh <c> array
zarrayq <c> array
```

pseudocode

```
IF cond THEN
addr.coprocessor number <= C#
addr.array number <= Ca
addr.quadword offset <= QWoff
addr.byte offset <= Boff

IF size = 00 THEN
array(addr) <= 0x00
ELSE IF size = 01 THEN
array(addr : addr + 1) <= 0x0000
ELSE IF size = 10 THEN
array(addr : addr + 3) <= 0x00000000
ELSE IF size = 11 THEN
array(addr : addr + 15) <= 0x00000000000000000000000000000000
END IF
END IF
```

```
opcode

mask   = F0000030
opcode = C0000030

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 0 0 0 0| sz| Cond  | Boff  |x|0 0 0|  C#s  | Ca|1 1| QWoff |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
==============================================================================
Coprocessor Execution Opcodes
==============================================================================
```

```
------------------------------------------------------------------------------
Execute Direct
------------------------------------------------------------------------------
```

The execute direct opcode initiates a coprocessor command in which all of the operation arguments are passed immediately to the opcode. The C# field indicates which coprocessor will execute the command. The CPopfield is the coprocessor operation field. The Imm16 field contains the operation arguments that are passed with the command. The a field indicates whether the command should be executed asynchronously. The p field indicates if the thread should give up priority.

```
mnemonics

cpx   cmd, #imm16
cpxa  cmd, #imm16
cpxp  cmd, #imm16
cpxap cmd, #imm16
```

```
pseudocode

exe.coprocessor number <= C#
exe.coprocessor operation <= CPop
exe.coprocessor arguments <= "0000000000000000000000000000" || Imm16

coprocessor <= exe

IF a = 1 THEN
PC <= PC + 1
ELSE
PC <= stall
END IF

IF p = 1 THEN
PriorityOwner(other thread) <= TRUE
ELSE
PriorityOwner(other thread) <= PriorityOwner(other thread)
END IF
```

```
opcode

mask   = F0010000
opcode = F0000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 1|   C#  |p|  CPop   |a|0|            imm16              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+


------------------------------------------------------------------------------
Execute Indirect
------------------------------------------------------------------------------

The execute indirect opcode initiates a coprocessor command in which the
operation arguments are a combination of a GPR register and an immediate
field. The field C# indicates which coprocessor the command is to be executed
on. The field CPop is the coprocessor operation field. The R field is the GPR
register to be passed as part of the operation arguments. The Imm12 field
contains the immediate operation arguments that are passed. The a field
indicates whether the command should be executed asynchronously. The p field
indicates if the thread should give up priority.


mnemonics

cpx   cmd, wreg, #imm12
cpxa  cmd, wreg, #imm12
cpxp  cmd, wreg, #imm12
cpxap cmd, wrge, #imm12


pseudocode

exe.coprocessor number <= C#
exe.coprocessor operation <= CPop
exe.coprocessor arguments <= Imm12 & GPR(R)

coprocessor <= exe

IF a = 1 THEN
PC <= PC + 1
ELSE
PC <= stall
END IF

IF p = 1 THEN
PriorityOwner(other thread) <= TRUE
ELSE
PriorityOwner(other thread) <= PriorityOwner(other thread)
END IF


opcode

mask   = F0010000
```

```
opcode = F0010000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 1|   C#  |p|  CPop   |a|1|   R   |    imm12              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Execute Direct Conditional

The execute direct conditional opcode is similar to the execute direct opcode except that the execute direct conditional opcode command can be issued conditionally based on the Cond field. To make room in the opcode for the Cond field, the coprocessor opcode field (op) is shortened to two bits. Because the high order four bits of the coprocessor operation are assumed to be zeros, conditional operations are restricted to the lower four commands of a coprocessor. The command is assumed to be synchronous because the opcode does not have a bit to indicate whether it is asynchronous or synchronous. Priority cannot be released with this opcode.

mnemonics

```
cpx <c> cmd, #imm16
```

pseudocode

```
IF cond THEN
exe.coprocessor number <= C#
exe.coprocessor operation <= "0000" || op
exe.coprocessor arguments <= "0000000000000000000000000000" || Imm16

coprocessor <= exe
END IF
```

opcode

```
mask   = F0030000
opcode = E0010000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 0|   C#  |  Cond | op|0|1|           imm16               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Execute Indirect Conditional

The execute indirect conditional opcode is similar to the execute indirect opcode except that the execute indirect command can be issued conditionally based on the Cond field. To make room in the opcode for the Cond field, the coprocessor opcode field (op) is shortened to two bits. Because the high

order four bits of the coprocessor operation are assumed to be 0s, conditional operations are restricted to the lower four commands of a coprocessor. The command is assumed to be synchronous because the opcode does not have a bit to indicate whether it is asynchronous or synchronous. Priority cannot be released with this opcode.

mnemonics

```
cpx <c> cmd, wreg, #imm12
```

pseudocode

```
IF cond THEN
exe.coprocessor number <= C#
exe.coprocessor operation <= "0000" || op
exe.coprocessor arguments <= Imm12 || GPR(R)

coprocessor <= exe
END IF
```

opcode

```
mask   = F0030000
opcode = E0030000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 0|   C#   |  Cond | op|1|1|   R   |        imm12          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Wait

The wait opcode synchronizes one or more coprocessors. The mask16 field (see the register bit list table in Section 7.3.3 on page 213) is a bit mask (one bit per coprocessor) in which the bit number corresponds to the coprocessor number. The thread stalls until all coprocessors indicated by the mask complete their operations. Priority can be released with this command.

mnemonics

```
wait  uint16
waitp uint16
```

pseudocode

```
IF Reduction_OR(mask16(i) = coprocessor.busy(i)) THEN
PC <= stall
ELSE
PC <= PC + 1
END IF
```

```
IF p = 1 THEN
PriorityOwner(other thread) <= TRUE
ELSE
PriorityOwner(other thread) <= PriorityOwner(other thread)
END IF


opcode

mask   = E0090000
opcode = E0000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 0|0 0 0 0|p|0 0 0|0|0 0|0|             mask16            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+


------------------------------------------------------------------------------
Wait and Branch
------------------------------------------------------------------------------

The wait and branch opcode synchronizes with one coprocessor and branch on
its one bit OK/KO flag. This opcode causes the thread to stall until the
coprocessor represented by C# is no longer busy. The OK/KO flag is then
compared with the field OK. If they are equal, the thread branches to the
address in the target16 field. Priority can be released with this command.


mnemonics

waitok  cn, target
waitko  cn, target
waitokp cn, target
waitkop cn, target


pseudocode

IF coprocessor.busy(C#) = 1 THEN
PC <= stall
ELSIF coprocessor.ok(C#) = ok THEN
PC <= target16
ELSE
PC <= PC + 1
END IF

IF p = 1 THEN
PriorityOwner(other thread) <= TRUE
ELSE
PriorityOwner(other thread) <= PriorityOwner(other thread)
END IF


opcode

mask   = F0090000
```

opcode = E0080000

```
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 0|   C#  |p|0 0 0|1|0|k|0|            target16           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

===

ALU Opcodes

===

```
AluOp operation flags
0000 add Z C N V
0001 add with carry Z C N V
0010 sub Z C N V
0011 sub with carry Z C N V
0100 xor Z   N
0101 and Z   N
0110 or Z   N
0111 sll Z C N
1000 srl Z C N
1001 sra Z C N
1010 ror Z C N
1011 cmp Z C N V
1100 test Z   N
1101 not Z   N
1110 mov
1111 reserved
```

Arithmetic Immediate

The arithmetic immediate opcode performs an arithmetic function on a GPR register in which the second operand is a 12-bit immediate value. The word GPR operand and destination GPR are specified in the R field and the immediate operand is represented by Imm4&Imm8. The actual portion of the GPR and extension of the immediate operand are shown in the ot3i table. The arithmetic function performed is given in the AluOp field. This AluOp functions of AND, OR, XOR, TST, and COMPARE are not used with this opcode and have a different opcode when the second operand is an immediate value. Arithmetic opcodes can be performed without changing the ALU status flags if the i field is a 1.

mnemonics

```
alu   <c> hreg, imm12
alu   <c> wreg, imm12
alui  <c> hreg, imm12
alui  <c> wreg, imm12
alus  <c> hreg, imm12
alus  <c> wreg, imm12
alusi <c> hreg, imm12
alusi <c> wreg, imm12
```

pseudocode:

```
IF cond THEN
alu.opr1 <= GPR(R, ot3i)
alu.opr2 <= ot3i(Imm4 || Imm8)

GPR(R, ot3i) <= Aluop.result(alu.opr1, alu.opr2)

IF i = 0 THEN
AluStatus <= Aluop.flags(alu.opr1, alu.opr2)
END IF
END IF
```

opcode

```
mask   = F0000000
opcode = 80000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 0 0 0|i| ot3i| cond  |   R   | imm4  | AluOp |  imm8         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

ot3i GPR imm
0 half 1 zext16
1 half 0 zext16
2 word zext32
3 word sext32
4 half 1 sext16
5 half 0 sext16
```

Logical Immediate

The logical immediate opcode performs the logical functions AND, OR, XOR, and TEST on a GPR register in which the second operand is a 16-bit immediate value. the R field specifies the word GPR operand and destination GPR, and the h field specifies the even (h = 0) or odd (h = 1) halfword of this GPR. The immediate operand is represented by Imm16. The arithmetic function performed is given in the Lop field. Logical opcodes can be performed without changing the ALU status flags if the i field is a 1.

mnemonics

```
alu   <c> hreg, imm16
alui  <c> hreg, imm16
```

pseudocode:

```
IF cond THEN
alu.opr1 <= GPR(R:h)
```

```
alu.opr2 <= Imm16

GPR(R:h) <= Lop.result(alu.opr1, alu.opr2)

IF i = 0 THEN
AluStatus <= Lop.flags(alu.opr1, alu.opr2)
END IF
END IF
```

opcode

```
mask   = F0000000
opcode = 90000000

 1  0  9  8  7  6  5  4  3  2  1  0  9  8  7  6  5  4  3  2  1  0  9  8  7  6  5  4  3  2  1  0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1  0  0  1|i|h|lop| cond  |    R    |            imm16            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

h GPR
0 half 0
1 half 1

lop operation pseudocode flags
00 xor result = orp1 XOR opr2 C V
01 and result = opr1 AND opr2 C V
10 or result = opr1 OR opr2 C V
11 test opr1 AND opr2 C V
```

Compare Immediate

The compare immediate opcode performs the compare functions on a GPR register in which the second operand is a 16-bit immediate value. The source GPR operand and destination GPR are specified in the R field, and the immediate operand is represented by Imm16. The actual portion of the GPR and extension of the immediate operand are shown in Figure 7-9. The compare immediate opcode always changes the ALU status flags.

mnemonics

```
cmp   <c> hreg, imm16
cmp   <c> wreg, imm16
cmps  <c> wreg, imm16
```

pseudocode

```
IF cond THEN
alu.opr1 <= GPR(R, ot2i)
alu.opr2 <= ot2i(Imm16)

AluStatus <= compare(alu.opr1, alu.opr2)
END IF
```

opcode

```
mask   = FC000000
opcode = A0000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 0 1 0 0 0|o2i| cond  |   R   |             imm16             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
ot2i GPR imm
00 half 1 imm16
01 half 0 imm16
10 word zext32
11 word sext32
```

Load Immediate

The operation arguments of the load immediate opcode are a combination of a GPR register and a 16-bit immediate field. The source GPR operand and destination GPR are specified by the R field and the immediate operand is represented by Imm16. The actual portion of the GPR and extension of the immediate operand are shown in Figure 7-10 and Figure 7-11. Load immediate opcode never changes the ALU status flags if executed.

mnemonics

```
ldr   <c> hreg, imm16
ldr   <c> breg, imm8
ldru  <c> wreg, imm16
ldro  <c> wreg, imm16
ldruo <c> wreg, imm16
```

pseudocode:

```
IF cond THEN
GPR(R, ot4i) <= ot4i(Imm16)
END IF
```

opcode

```
mask   = F0000000
opcode = B0000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 0 1 1| ot4i   | cond  |   R   |             imm16             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
ot4i GPR imm
0 half 1 imm16
```

```
1 half 0 imm16
2 word zext32
3 word sext32
4 word oext32
5 word oextr32
6 word zextr32
8 byte 3 imm16 & 0xff
9 byte 2 imm16 & 0xff
A byte 1 imm16 & 0xff
B byte 0 imm16 & 0xff
```

Arithmetic Register

The arithmetic register opcode performs an arithmetic function on a GPR register in which the second operand is also a GPR register. The first GPR operand and the destination GPR are specified by the word address R1 and the specific portion to be used is encoded in ot3r. The second operand source is represented by word address R2 with the h field determining the even or odd halfword if the ot3r field indicates a halfword is needed. If the AluOp is a shift or rotate command, the second operand source is used as the amount of the shift or rotate. Otherwise, ot3r indicates the relationship between the two operands. If the AluOp is a logical operation (AND, OR, XOR, TEST), the second operand source can then further be modified by the m and n fields. The m field is the mask field and, if active, creates a 1-bit mask in which the "1" bit is represented by the second operand source. The n field is the invert field and, if active, will invert the second operand source. If both the m and n fields are "1", the mask is created before the inversion.
Table 7-6 show how to use these fields to create other operations from the basic logic commands. The arithmetic function performed is given in the AluOp field. Arithmetic opcodes can be performed without changing the ALU status flags if the "i" field is a 1. Arithmetic opcodes can be performed without writing back the data if the "w" field is a 1.

mnemonics

```
alu  <c> reg, reg
alui <c> reg, reg
```

pseudocode

```
IF cond THEN
alu.opr1 <= GPR(R1, ot3r)
alu.opr2 <= GPR(R2, h)

IF m = 1 THEN
alu.opr2 <= bitmask(alu.opr2)
END IF

IF n = 1 THEN
alu.opr2 <= NOT(alu.opr2)
END IF

IF w = 0 THEN
```

```
GPR(R2, ot3i) <= Aluop.result(alu.opr1, alu.opr2)
END IF

IF i = 0 THEN
AluStatus <= Aluop.flags(alu.opr1, alu.opr2)
END IF
END IF


opcode

mask   = F0000010
opcode = C0000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 0 0|i| ot3r| cond  |   Rd  |   Rs  | AluOp |0 0 0|0|w|h|n|m|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

ot3r rd rs
0 half 1 half h
1 half 0 half h
2 word zext32(half h)
3 word sext32(half h)
4 word word
```

Count Leading Zeros

The count leading zeros opcode returns the number of zeros from left to right until the first 1-bit is encountered. This operation can be performed on a halfword or word GPR register. There is a second variation of this command in which the return value of this command is the bit position of the first "1" from left to right in the GPR. The GPR to be analyzed is determined by the R fields. If the GPR is a halfword instruction, the h field indicates whether the even or odd half is used. The destination register is always a halfword register and is represented by the Rd field and halfword indicator ‘hd’. The w field indicates whether the operation is a word or halfword. The n field determines if the command counts the number of leading zeros (n = 0) or if the command returns the bits position of the first one (n = 1). The only flag for this command is the overflow flag, which is set if the GPR being tested contains all zeros. The setting of this flag can be inhibited if the i field is a one.

mnemonics

```
ctlz    <c> hreg, hreg
ctlz    <c> hreg, wreg
ctlzi   <c> hreg, hreg
ctlzi   <c> hreg, wreg
ctlzbp  <c> hreg, hreg
ctlzbp  <c> hreg, wreg
ctlzbpi <c> hreg, hreg
ctlzbpi <c> hreg, wreg
```

```
pseudocode

IF cond THEN
alu.opr1 <= GPR(Rs, h)

alu.result <= count_zero_left_to_right(alu.opr1)

IF n = 1 THEN
GPR(Rd, hd) <= NOT(alu.result)
ELSE
GPR(Rd, hd) <= alu.result
END IF

IF i = 0 THEN
AluStatus <= Aluop.flags(alu.opr1, alu.opr2)
END IF
END IF


opcode

mask   = F8000000
opcode = A8000000

 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 0 1 0 1|d|w|i| cond  |   Rd   |   Rs   |0 0 0 0 0 0 0 0 0|h|n|0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

C README and Scripts

C.1 README

```
NP4GS3 emulator
===============

This archive consists of three independant parts:

powerpc-gcc/
Contains the sources for a minimal tool chain for
Linux/powerpc. Included are the GNU binutils, the
GNU gcc and the necessary libc and kernel header files.

pce/
Contains the NP4GS3 emulator

linux-2.6.3/
Contains the Linux source code. This is an unmodified
official kernel.


The "buildall" script in this directory configures, builds and
installs the powerpc cross compiler and the NP4GS3 emulator. The
installation prefix is passed as a parameter:

$ ./buildall /usr/local

The "buildall" script doesn't do any dependency tracking. Every
time it is startet, the build process starts over from scratch.

Except for the first installation it is probably best not to use
the buildall script but rather install the emulator and the
cross compiler separately. For further installation instructions
see pce/README.NP4 and powerpc-gcc/README.

To cross compile a kernel for the powerpc you can use the
script linux-2.6.3/cross.

There is a test environment in pce/test, including precompiled
kernel images and RAM and ROM images for the emulator. Use
pce/test/go.sh to start the emulator.


have fun
Hampa Hug <hampa@hampa.ch>
```

C.2 Cross Compiler README

```
Building a cross compiler for powerpc-linux

1) Build the binutils:

$ cd binutils-2.14
$ CFLAGS="-O2" ./configure --prefix=/usr/local --target=powerpc-linux
$ make
```

```
$ make install
$ cd ..


2) Copy the kernel header files and the libc to the target directory:

$ cd powerpc-linux
$ cp -a include/* /usr/local/powerpc-linux/include
$ cp -a lib/* /usr/local/powerpc-linux/lib
$ cd ..


3) Build the gcc:

$ cd gcc-3.3.3
$ CFLAGS="-O2" ./configure --prefix=/usr/local --target=powerpc-linux
$ make
$ make install
$ cd ..


Notes:

- Make sure that the powerpc binutils are in the PATH before
  attempting to build gcc.

- The header files and C library included here were taken from
  a debian-testing system for powerpc.

- The C library itself is not strictly required. Only the
  header files are required.

- Other versions of binutils and gcc should work just as
  well.

- To build a linux 2.6.x kernel do:
  $ cd ${linux-src}
  $ export ARCH=ppc
  $ export CROSS_COMPILE=powerpc-linux
  and then the usual:
  $ make config ; make
```

C.3 PCE README

```
Notes on building the PCE NP4GS3 emulator
-----------------------------------------

$ ./configure
$ make
$ make install

$ src/arch/np4gs3/np4gs3 --help
$ src/arch/np4gs3/np4gs3 -v -c yourconfigfile.cfg
-g


- No extra options for configure are required. There are some
```

```
  PCE specific options but they are needed for the IBM PC
  emulator only.

- The NP4GS3 emulator does not use terminals so it does not
  matter what terminals are compiled in (terminals are
  autodetected).

- The emulator can be built outside the source directory. In
  fact, this is recommended:
  $ mkdir build
  $ cd build
  $ ../configure [...]
  $ make

- The installation step is not required. The emulator binary
  (${builddir}/src/arch/np4gs3/np4gs3) can be run directly
  from the build directory.

- The NP4GS3 ROM to start linux is not built by default (a
  PowerPC crosscompiler is needed for that). If you have
  such a crosscompiler you can build the ROM by typing
  'make' in ${srcdir}/src/arch/np4gs3/rom. Otherwise there
  is a precompiled binary in ${srcdir}/test.

- To run the emulator you need a config file. A commented
  example config file is built in ${builddir}/src/arch/np4gs3/np4gs3.cfg.


Source Tree Layout
------------------

src/chipset/ppc405
Support chips for the PowerPC 405. At the moment this includes
only the UIC.

src/devices
Generic devices such as video cards disks and serial ports.
The only device relevant for the NP4 emulator is the
memory device.

src/libini
The library for reading configuration files.

src/lib
Some support functions used mostly by the user interface.

src/cpu/ppc405
The PowerPC 405 emulator

src/arch/np4gs3
The NP4GS3 emulator main program

src/arch/np4gs3/clp
The CLP emulator

src/arch/np4gs3/cop
The individual coprocessors
```

```
src/arch/np4gs3/rom
A trivial ROM used to boot linux


have fun
Hampa Hug <hampa@hampa.ch>
```

C.4 Emulator Start Script

```
#!/bin/bash

# check the source tree for the np4gs3 executable
for f in ../build/src/arch/np4gs3/np4gs3 ../src/arch/np4gs3/np4gs3 ; do
if test -x "$f" ; then
bin=$f
break
fi
done

# if the executable is not in the source tree, assume it can
# be found via the PATH
if test ! -x "$bin" ; then
bin="np4gs3"
fi

# make sure current is a symlink that points somewhere
if test ! -d current ; then
rm -f current
ln -s 2.6.3-2 current
fi

echo "$(date "+%Y-%m-%d %T")" > current/pce.log

exec $bin -c np4gs3.cfg -l current/pce.log -v "$@"
```

C.5 Cross Compiler Build Script

```
#!/bin/bash

if test -z "$prefix" ; then
prefix="$HOME/pkg/powerpc-linux"
fi

# Build the binutils:
(
cd binutils-2.14
CFLAGS="-O2" ./configure "--prefix=$prefix" --target=powerpc-linux
make
make install
)

# Copy the kernel header files and the libc to the target directory:
(
mkdir -p "$prefix/powerpc-linux/include"
mkdir -p "$prefix/powerpc-linux/lib"
```

```
cd powerpc-linux
cp -a include/* "$prefix/powerpc-linux/include"
cp -a lib/* "$prefix/powerpc-linux/lib"
)

# Build the gcc:
(
export PATH="$prefix/bin:$PATH"
cd gcc-3.3.3
CFLAGS="-O2" ./configure "--prefix=$prefix" \
--target=powerpc-linux \
--disable-shared \
--enable-languages=c \
--disable-nls
make
make install
)
```

C.6 PCE Build Script

```
#!/bin/bash

if test -z "$prefix" ; then
prefix="$HOME/pkg/pce"
fi

mkdir -p build
cd build
../configure --prefix="$prefix"

make
make install
```

D Source Tree Layout

The following list shows the subdirectories in the source tree along with a short description of what can be found in each directory.

```
src/arch/np4gs3
        The NP4GS3 emulator main program.

src/arch/np4gs3/clp
        The CLP emulator.

src/arch/np4gs3/cop
        The individual coprocessors.

src/arch/np4gs3/rom
        A trivial ROM used to boot linux on the emulated NP4GS3.

src/chipset/ppc405
        Support chips for the PowerPC 405. At the moment this includes
        only the UIC (universal interrupt controller).

src/devices
        Generic devices such as video cards disks and serial ports.
        The only device relevant for the NP4GS3 emulator is the
        memory device.

src/libini
        The library for reading configuration files.

src/lib
        Some support functions used mostly by the user interface.

src/cpu/ppc405
        The PowerPC 405 emulator.
```

The source tree contains additional directories. Those are not related to the NP4GS3 emulator.

E Configuration File Syntax

The emulator configuration file syntax is defined by the following EBNF (extended Backus-Naur form):

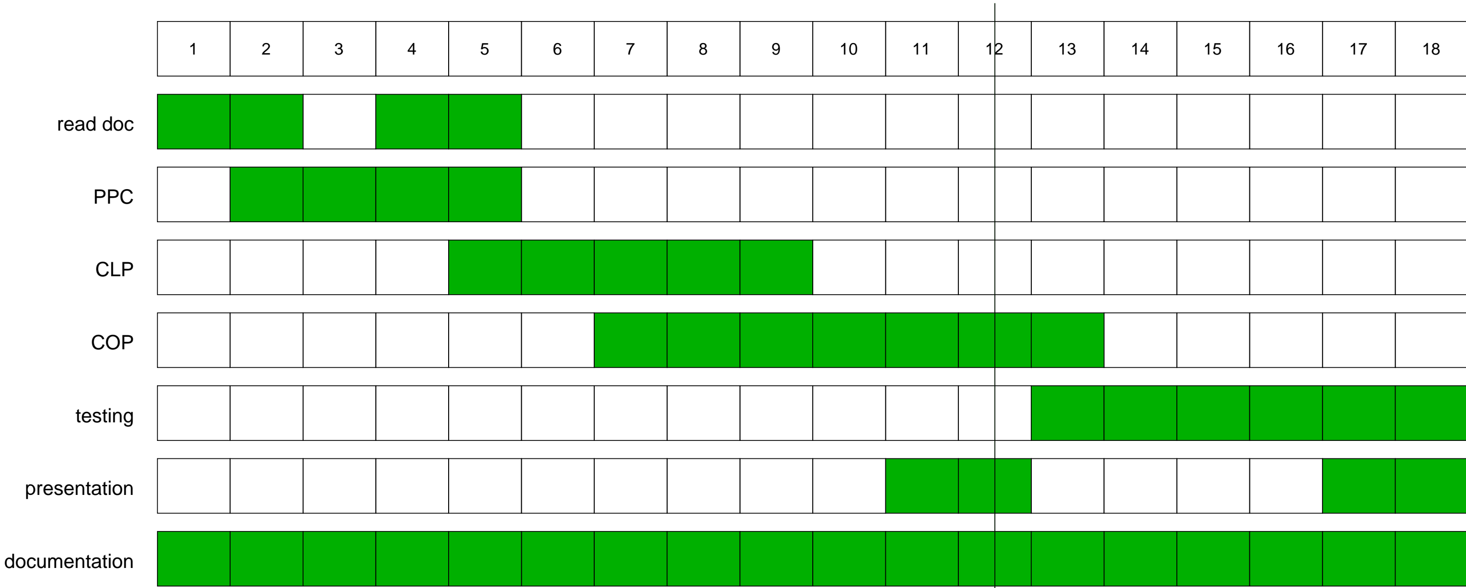
```
Config          = SectionBody.
SectionBody     = { Value | Section }
Section         = "section" String "{" SectionBody "}".
Value           = Identifier "=" (String | Float | Integer | Hex).
String          = (""" CharSequence """) | Identifier.
Float           = ["+" | "-"] Digit { Digit } [ "." Digit { Digit }].
Integer         = DecimalInteger | HexInteger.
DecimalInteger  = ["+" | "-"] Digit { Digit }.
HexInteger      = "0x" HexDigit { HexDigit }.
Identifier      = Letter { Letter | Digit }.
Letter          = "a" | ... | "z" | "A" | ... | "Z" | "_".
Digit           = "0" | ... | "9".
HexDigit        = Digit | "a" | ... | "f" | "A" | ... | "F".
```

Here is an example configuration file:

```
# Example config file for the NP4GS3 emulator

section np4gs3 {
        section powerpc {
                model = "ppc405"
        }

        # Multiple "ram" sections may be present
        section ram {
                # The linear base address
                base = 0x00000000

                # The size in bytes
                size = 0x04000000

                # The RAM image that is used to initialize the RAM
                file = "linux.bin"
        }

        # Multiple "rom" sections may be present
        section rom {
                # The file from which the rom code is loaded
                file = "np4gs3.rom"

                # The linear base address
                base = 0xffff0000

                # The rom size in bytes
                size = 65536
        }

        # The nvram is not used yet but can be configured all the same
        section nvram {
                base = 0xf0000000
                size = 0x2000
                file = "nvram.dat"
```

```
        }

        section epc {
                section imem {
                        size = 32768
                        file = "epc_imem.dat"
                }
        }

        # Up to two "serial" sections may be present. The first is
        # UART0, the second UART1. If the base addresses are changed,
        # linux won't find the UARTs
        section serial {
                io = 0xef600300
                irq = 0

                # The file that output is written to
                file = "current/uart0.out"
        }

        section serial {
                io = 0xef600400
                irq = 1
                file = "current/uart1.out"
        }

        section epc {
                section imem {
                        size = 32768
                        file = "imem.ram"
                }
        }
}
```

F Timeline



References

[1] IBM, *IBM PowerNP NP4GS3 Network Processor*; http://www.ibm.com, 2003.

[2] IBM, *PowerPC 405 Embedded Processor Core User's Manual*; http://www.ibm.com, 2001.

[3] IBM, *PowerPC 405GP Embedded Processor User's Manual*; http://www.ibm.com, 2003.

[4] IBM, *IBM PowerNP Assembler Language Programmer's Guide and Opcode Summary*; http://www.ibm.com, 2003.

[5] Bochs, http://bochs.sourceforge.net/, 1998-2004.

[6] VMware, http://www.vmware.com/, 1998-2004.

[7] F. Kaufmann, *Design And Implementation Of A Modular Emulator For Topsy*; TIK, ETH Zurich, 2003

[8] George Fankhauser, Christian Conrad, Eckart Zitzler, Bernhard Plattner, *Topsy - A Teachable Operating System, Version 1.1*; Computer Engineering and Networks Laboratory, ETH Zuerich, 1996, 2000.