The PHINIX+ System Architecture Documentation
Part 1: The CPU

Come discuss with us at the official PHINIX+ Discord server:
https://discord.gg/EFKDF3VE9C

Version: 0.4.6
Date: 29th of August 2024
by Martin Andronikos

Contents

1 Preface

This section discusses the nature of the documentation itself, the scope and aim of the PHINIX+ project, and about the author as an individual and their motives. As a result, the use of the first person in the following section is unavoidable. The formal specification begins at Section 2 if such details are irrelevant for the reader.

1.1 About the Document

The purpose of the document is to describe with maximum possible detail all the features of the PHINIX+ system. It therefore tries to conform to the typical requirements expected from technical documentation. The most important details to be transparent about regarding the document itself are thus the decisions about the look of the document (styling) and about the licensing around the document.

1.1.1 Styling Decisions

This document was written using the "Typst" typesetting program. If the source code of the used template is not available or the reader is not aware of Typst's syntax, the decisions made regarding styling are hereby given:

- Pages are A4 sized with 25mm of vertical and 20mm of horizontal margins.
- For the bulk of the text the serif font "IBM Plex Serif" was used.
- For the headings and for the title the sans serif font "IBM Plex Sans" was used.
- For the code blocks the Nerd Fonts variant of the monospace font"Inconsolata" was used.
- Internal links (references) are in blue color with the exception of the contents page and footnotes.
- External links (hyperlinks) are underlined and in blue color (as shown above).

1.1.2 Licensing Decisions

1.2 About the Author

Though I do know my way around the field of processor design and implementation, I have no formal experience with the subject. Everything I know about regarding the topic I have learned by myself and with help from other people online. However, I am in the process of attending a computer engineering course at a polytechnic university.

Typesetting is also an activity which I have had to teach myself. My university did provide me a "Technical Document Writing" lesson though it was in reality of little help. As a result, if you would like to suggest something regarding the document, don't hesitate to reach out on Discord.

1.3 About the Project

This documentation and the overall design and direction of the PHINIX+ project is my personal project which I have been working on during my free time. I never got to experience early computing or the home computer revolution. As a result, I made it my goal to come up with a completely independent computational system that would mimic the experience of using systems of the late 1980s to early 1990s.

PHINIX+ attempts to be a platform from which many of the concepts common in the modern computing environment could be understood (such as Operating Systems) through re-implementation, as well as a platform on which the retro community could build upon. PHINIX+ thus tries to cater to many use cases and it should be wholly up to the implementer which of those use cases is most important for what they want out of the system.

2 Introduction

This document is the official specification for the PHINIX+ Central Processing Unit. It is intended to explain in detail the capabilities and the layout of the processor in an abstract manner in order to remain agnostic of the possible implementations of it. While this document doesn't try to make any assertions of a "correct" sort of implementation, the architecture was built with the intention to exploit pipelining to gain in performance.

2.1 Ancestral History

PHINIX+ is a "constructed" acronym which stands for *Pipelined High-speed INteger Instruction eXecutor*. The "+" in the name is meant to signify advancements from a previously designed processor, PHINIX, from which most ideas were directly taken and improved upon. PHINIX used 16-bit word-addressing, which turned out to be unwieldy and did not deliver in terms of memory capacity. PHINIX+ expands to 32 bits while also adding byte-addressing to simplify integration with the existing computing paradigms, all based around 8-bit units.

2.2 Influence Sources

PHINIX+ mainly derives from the *Reduced Instruction Set Computing* (RISC) paradigm. However, that does not mean it follows the established norm for a RISC processor, opting instead for a more expansive set of instructions, mainly concerning the improvement of flags management and bit math. The core principles of RISC, like the load-store paradigm and the general usage nature of the provided registers, do exist in PHINIX+ but not without being improved upon.

One of the most apparent features a programmer wishing to use PHINIX+ encounters is the dual register file. This is a feature influenced directly by the Motorola 68000 series of processors. Though that processor was in no way following RISC, the adoption of the dual register file was due to similar reasons. As a result, PHINIX+ has been lovingly nicknamed the *Actually-RISC™ m68k*[1].

2.3 Things Done Differently

As mentioned prior, PHINIX+ mostly follows RISC but has changed how a few things work in the interest of exploration. Many of the decisions taken could be considered "unorthodox", but one of the most important premises of this project is to try new ways of doing things for the educational value. Great care has been taken to devise methods that improve performance using the minimum amount of required hardware. Following is a list of the most important novel features of the CPU:

Feature	Justification
Dual register files.[2] (The separation of the registers into data and address register files.)	Allows for a trivial auto-increment operation, removing the need for special hardware for the stack and other pointers. This feature also allows for two independent operations to be executed in parallel with little increase to the size of the implementation.
Condition codes register file. (The ability to use any single-bit "flag" for any purpose.)	Makes operations on them a feasible prospect, reducing the amount of branches. The now explicit nature of flag operations makes each instruction wishing to modify them now opt-in instead of opt-out, reducing flag use.
Load-store instruction byte permutations. (The ability to choose a preferred ordering for the bytes when loading or storing them.)	Addresses the age-old dilemma of little- VS big-endian while both making the least significant bits of an address useful and eliminating the need for bus errors, but doing so without requiring the system to perform unaligned memory accesses.

Table 1: Notable novel features of PHINIX+

[1] Disclaimer, not actually a trademark.
[2] As mentioned prior in relation to the m68k.

3 Register Files

A register file is a grouping of individually addressable memory cells, also known as registers, that are closely coupled with the operation and structure of a processor architecture. PHINIX+ defines three of these register files, each with a slightly different purpose. The first two of the novel features outlined in Table 1 are thus explained in detail in this chapter.

Registers are the most common subjects (also referred to as operands) of the instructions a CPU executes, especially if it follows the load-store paradigm wherein, as a consequence, no arithmetic/logic instruction can directly operate on memory.

3.1 General Purpose Registers

The general purpose registers are a pair of register files, each file aimed towards a narrower set of use cases. This is the first novel feature of PHINIX+, initially noted in the first entry of Table 1. Each file consists of 16 registers for a total of 32 registers. The two register files are to be detailed in the following two subsections and an extra subsection listing out the registers in table form.

The quantity of general purpose registers PHINIX+ provides, 32, is a commonality amongst the majority of RISC processor architectures, consistently more than the usual CISC architecture. The reason for this is that having instructions that do less work would necessitate more space to store intermediate values in.

3.1.1 Data Registers

The data registers are the most versatile set of registers available. 16 are provided, each one being 32 bits in width, denoted *$xN* (where N is a single hexadecimal digit ranging from 0 to 9 and then from A to F). They are intended to hold values loaded from or to be stored to memory and to be the subject (operand) of most arithmetic and logic operations the CPU performs.

It is important to note that the first data register, *$x0* is not actually a register one can write to. Register *$x0* is a constant holding the value zero. This is a nigh universal design pattern across the RISC processor family and has been included in PHINIX+ for the same reasons.

Data register *$x0* being a constant zero aids in better usage of existing instructions (or conversely allows for the reduction of the needed instructions) in at least the following two ways:

- By allowing them to discard their result by storing to this register when only the condition code generated from that instruction is required. For example, a comparison instruction can be achieved by doing a subtraction and storing the result to register zero.
- By allowing instructions that address memory to coalesce under a single addressing mode, “reg + imm”. This is such because when supplying *$x0* as the register, the effective address becomes just the immediate, and by supplying a zero immediate, the effective address becomes the supplied register.[3]

3.1.2 Address Registers

The address registers are a secondary set of registers that are less versatile computation-wise than the data registers. Just like the data registers, 16 are provided, again, each one being 32 bits in width, denoted instead *$yN* (where N is a single hexadecimal digit, ranging from 0 to 9 and then from A to F).

While still having more available functionality than for just storing and manipulating pointers, the address registers’ primary purpose is nevertheless for the aforementioned task or for serving as a bank of secondary storage when the data registers are not enough to hold all datums of a computation.

[3] Addressing modes as they relate to PHINIX+ are discussed in TODO.

3.1.3 Data and Address Register Tables

The previously discussed general purpose registers are hereby illustrated in table form.

Data Registers	Address Registers
$x0	$y0
$x1	$y1
$x2	$y2
$x3	$y3
$x4	$y4
$x5	$y5
$x6	$y6
$x7	$y7
$x8	$y8
$x9	$y9
$xA	$yA
$xB	$yB
$xC	$yC
$xD	$yD
$xE	$yE
$xF	$yF

Table 2: PHINIX+'s general purpose registers

3.2 Condition Code Registers

The condition code registers, or "flag" registers, are a collection of 8 registers each one being only 1 bit in width. Their purpose is, as the name suggests, to hold intermediate condition codes for the purpose of program control flow. Branch instructions, which are later discussed, are intimately tied with this set of registers.[4] This is the second novel feature of PHINIX+, initially noted in the second entry of Table 1. They are denoted *$cN* (where N is a single octal digit, ranging from 0 to 7).

Condition Code Registers
$c0
$c1
$c2
$c3
$c4
$c5
$c6
$c7

Table 3: PHINIX+'s condition code registers

Condition code register *$c0* is a register that constantly and forever holds a zero bit. This simplifies the program logic needed to handle a cascading set of conditions and reduces the amount of instructions required to handle all the needed cases.

3.3 Register Calling Convention

The tables of registers previously showcased contained just one column, which lists the "architectural names" of the registers. They denote the systematic name given to the register for

[4] Branch instructions and how they relate to the condition code registers are discussed in TODO.

the purposes of a hardware point-of-view. An implementer doesn't care how the registers are used because they are all generic so they get generic architectural names.

In this section the registers are re-examined, this time concerning a software point-of-view instead. In contrast with the implementer, a programmer needs to organize the registers given to them in a consistent manner in order to ensure proper behavior when calling into subroutines, thus follow a *convention* for *calling*, a pre-agreed set of rules to ensure compatibility between interacting subroutines.

To assist in this endeavour, this document provides a reference, standard calling convention that any software written for the processor is advised to use, such that software written by different developers can interoperate. A calling convention's whole purpose is to provide a common ground for software development, so that someones's code is able to use someone else's.

Developers are free to come up with an alternative convention to better suit their needs, it just then falls unto them to interface with other existing software which is not compatible with their custom convention.

3.3.1 The Concept of Saving

In the act of a subroutine call, there are two hypothetical entities at play: the code performing the call–referred to as the caller, and the code being called–referred to as the callee. Using these roles as a framework for a calling convention yields the simplest form of segregation based on whose duty it is to "clean up" the register, revert its contents to a known-good state. Thus, each register can be assigned to either be saved by the caller or by the callee. The PHINIX+ calling convention bases its design on this principle.

In practice, what it means for a register to be caller-saved is that the code being called has no obligation to maintain the contents of the register to its initial value. After the subroutine returns, the caller has to assume that all of the caller-saved registers now contain garbage values. Thus, it's the duty of the caller to preserve the values of registers it wants to continue using after the subroutine call.

There are mainly two ways the caller can preserve the value of a caller-saved register that it wants to keep using after the subroutine returns. Those are:
- Exploiting the stack by pushing the value of the register onto its stack frame.[5]
- Moving the value of the register onto another, callee-saved register that had previously been saved itself.

Likewise, what it means in practice for a register to be callee-saved is that the code doing the call expects that the value of that register remains the same after the return of the subroutine without it having to do anything. Thus, it's the duty of the code being called, the callee, to preserve the value in some way before using it and then revert the register to the old value before returning.

There are, again, mainly two ways the callee can preserve the value of a callee-saved register that it later intends to use. Those are:
- Exploiting the stack by pushing the value of the register onto its stack frame.[5]
- Moving the value of the register onto another, caller-saved register.

3.3.2 Convention Tables Glossary

The three tables to follow, Table 4, Table 5, and Table 6, contain a compact description of how they are intended to be used by the software developer. The terms used in these tables are hereby explained.

The headers of the tables contain these four entries with the exception of the condition code registers table, which contains only the first and last entries:
- *Architectural Name* is the one column of the original table.

[5]The stack is a concept that is explained in detail in TODO.

- *Convention Name* lists out the name that a programmer will use inside their assembly language development environment to refer to the specific register. It is an abbreviated form of the register's intended purpose.
- *Description* expands on the *Convention Name* by listing out in full the intended purpose of the specific register. Not every register has a unique purpose, so smaller groupings of registers with a common purpose are additionally numbered.
- *Saving* lists out the assigned saving of the specific register, either caller-save or callee-save (in the cases where it applies to do so). The implications for each of the two savings are explained in the previous sub-section.

3.3.3 Data Register Convention

The following table is an expanded version of the before shown table of data registers (left half of Table 2). Three additional columns have been added in order to detail the proposed standard calling convention for the data register file.

Architectural Name	Convention Name	Description	Saving
$x0	$zr	Constant Zero	N/A
$x1	$at	Assembler Temporary	Caller
$x2	$rp	Return Pointer	Caller
$x3	$t0	Temporary Value #0	Caller
$x4	$t1	Temporary Value #1	Caller
$x5	$t2	Temporary Value #2	Caller
$x6	$a0	Subroutine Argument #0	Caller
$x7	$a1	Subroutine Argument #1	Caller
$x8	$a2	Subroutine Argument #2	Caller
$x9	$s0	Saved Value #0	Callee
$xA	$s1	Saved Value #1	Callee
$xB	$s2	Saved Value #2	Callee
$xC	$s3	Saved Value #3	Callee
$xD	$s4	Saved Value #4	Callee
$xE	$s5	Saved Value #5	Callee
$xF	$fp	Frame Pointer	Callee

Table 4: PHINIX+'s data registers

3.3.4 Address Register Convention

The following table is an expanded version of the before shown table of address registers (right half of Table 2). Three additional columns have been added in order to detail the proposed standard calling convention for the address register file in the same manner as before.

Architectural Name	Convention Name	Description	Saving
$y0	$a3	Subroutine Argument #3	Caller
$y1	$a4	Subroutine Argument #4	Caller
$y2	$a5	Subroutine Argument #5	Caller
$y3	$t3	Temporary Value #3	Caller
$y4	$t4	Temporary Value #4	Caller
$y5	$t5	Temporary Value #5	Caller
$y6	$t6	Temporary Value #6	Caller
$y7	$t7	Temporary Value #7	Caller
$y8	$s6	Saved Value #6	Callee

Architectural Name	Convention Name	Description	Saving
$y9	$s7	Saved Value #7	Callee
$yA	$gp	Globals Pointer	Callee
$yB	$sp	User Stack Pointer	Callee
$yC	$k0	System Reserved #0	Callee
$yD	$k1	System Reserved #1	Callee
$yE	$k2	System Reserved #2	Callee
$yF	$kp	System Stack Pointer	Callee

Table 5: PHINIX+'s address registers

The last four registers in the address register file are "privileged". That means that they are accessible only when the processor is in a special mode of operation reserved for managerial code, like an operating system's kernel.[6] When not in this mode, the registers act nominally the same as the data register *$zr*. An implementer may, however, choose to have the processor react on such a violating access by alerting the privileged code of such an action.[7]

3.3.5 Condition Code Register Convention

The following table is an expanded version of the before shown table of condition code registers (Table 3). Only one additional column has been added in order to detail the proposed standard calling convention for the condition code register file. Only a saving convention has been assigned to this file in contrast to the other two, due the nature of the file. All of the registers are in the same and only grouping with callee saving.

Architectural Name	Saving
$c0	N/A
$c1	Callee
$c2	Callee
$c3	Callee
$c4	Callee
$c5	Callee
$c6	Callee
$c7	Callee

Table 6: PHINIX+'s condition code registers

Having eight condition code registers, each one being 1 bit in width, means that saving and restoring the contents can be done in one fell swoop. The entire register file's contents can fit into a single byte. To aid in this mechanic, all of the registers have the same saving (with the exception of *$c0*, on which saving is not applicable due to its constant nature).

[6] PHINIX+'s privileged execution mode is explained in detail in TODO.
[7] Such an alert would constitute an interrupt. Interrupts are explained in detail in TODO.