The PHINIX+ System Architecture Documentation
Part 1: The CPU

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

Version: 0.3.4
Date: 22nd 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 to signify and advancement from a previously designed processor, PHINIX, from whom most ideas where directly taken and improved upon. PHINIX used 16-bit word-addressing which turned out to be unwieldy and 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. A list of the most important novel features of the CPU are as follows:

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 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 Data Registers

The data registers are the most versatile set of registers available. Sixteen (16) are provided, all 32 bits in width, denoted *$xN* (where N, a single hexadecimal digit ranging from 0 to 9 and then from A to F).

Data registers are intended to store values loaded from or to be stored to memory and to be the subject of most arithmetic and logic operations the CPU performs.

Architectural Name	Convention Name	Saving
$x0	$zr	N/A
$x1	TBD	TBD
$x2	TBD	TBD
$x3	TBD	TBD
$x4	TBD	TBD
$x5	TBD	TBD
$x6	TBD	TBD
$x7	TBD	TBD
$x8	TBD	TBD
$x9	TBD	TBD
$xA	TBD	TBD
$xB	TBD	TBD
$xC	TBD	TBD
$xD	TBD	TBD
$xE	TBD	TBD
$xF	TBD	TBD

Table 2: PHINIX+'s data registers

Data register *$x0* is a register that constantly and forever holds the value zero. It aids in better usage of existing instructions 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. This is a nigh universal design pattern across the RISC processor family.

3.2 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, sixteen (16) are provided, again all 32 bits in width, denoted instead *$yN* (where N, 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 manipulating and storing pointers or as a bank of secondary storage when the data registers are not enough to hold all datums of a computation.

Architectural Name	Convention Name	Saving
$y0	TBD	TBD
$y1	TBD	TBD
$y2	TBD	TBD
$y3	TBD	TBD
$y4	TBD	TBD
$y5	TBD	TBD
$y6	TBD	TBD
$y7	TBD	TBD
$y8	TBD	TBD
$y9	TBD	TBD
$yA	TBD	TBD
$yB	TBD	TBD
$yC	TBD	TBD
$yD	TBD	TBD
$yE	TBD	TBD
$yF	TBD	TBD

Table 3: PHINIX+'s address registers

3.3 Condition Code Registers

The condition code registers are the most special one of the bunch. Eight (8) of them are provided, each one being just 1 bit in width. They are denoted *$cN* (where N, a single octal digit, ranging from 0 to 7).

A condition code register's purpose is to hold intermediate "flags", assisting in program control flow. The branch instructions to be later discussed are intimately tied with this set of registers.

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

Table 4: PHINIX+'s condition code registers

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

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 convention (with the exception of *$c0*, on which saving is not applicable due to its constant nature).

3.4 Calling Convention

The tables of registers previously showcased contained three columns. The first one, labeled *Architectural Name* denotes the systematic name given to the register for the purposes of a hardware point-of-view. The other two columns however, *Convention Name* and *Saving*, concern a software point-of-view instead. An implementer doesn't care how the registers are used because they are all generic so they get generic architectural names. A programmer however needs to organize the registers given to them in a consistent manner in order to ensure proper behavior when calling into subroutines, thus a *convention* for *calling*, a pre-agreed set of rules to ensure compatibility between interacting subroutines.

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. Developers however are free to come up with whatever 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. 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.