The PHINIX+ System Architecture Documentation
Volume 1: The CPU

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

Version: 0.6.3
Date: 22nd of January 2026
by Martin Andronikos

Contents

1 Introduction

This volume 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.

1.1 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.

1.2 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].

1.3 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.

2 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 formally introduced 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.

2.1 General Purpose Registers

The general purpose registers are a pair of register files, each one 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 sections and an extra section 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.

2.1.1 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 *$x0*.
- The most common operation, either as a standalone instruction or as a component in the function of more complex instructions, is addition. By having a constant-zero value (the neutral element of addition) available all the time allows addition to be effectively bypassed, expanding the usage of many instructions. For example, a move instruction can be achieved by doing an addition with one of the operands being *$x0*.

2.1.2 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.

2.1.3 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

2.2 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.[3] 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

Just like with data register *$x0*, condition code register *$c0* holds a constant zero value, except in this case it's a single zero-bit.

Having a constant zero condition register likewise reduces the needed variants for the instructions that handle them while still providing more options for the programmer.

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

3 Instruction Formats

Instructions are the set of fundamental operations the hardware itself supports. They, along with the register file, are the two most important things for a specification to cover, since they are the majority of what defines the capabilities, scope and quirks of the architecture. However, to best align with the RISC paradigm, each instruction has to fit into one of the defined instruction formats; the topic of this chapter.

The reasoning behind the inclusion of instruction formats is quite simple yet impactful. The benefits of constraining the capabilities of the instructions to fit into the available formats outweighs that constraint. Most key are the following principles:

- Simplification of the architecture and implementations as per RISC principles
- Organization of the instructions into groups of similar-acting ones
- Orthogonality – the principle of operand source indifference

3.1 Format Composition

Instruction formats are a set of pre-defined layouts consisting of various arrangements of fields that each instruction gets to use in a similar but unique way to do its task. There exist three different types of fields across all the defined formats that PHINIX+ uses, those being:

- *Control* fields, used for identifying the specific operation that is being requested. There are a few sub-types of control fields, each with a different size. Namely, there are *operation code* fields with a size of 8 bits, *function* fields with a size of 4 bits, and *negation selector* fields with a size of 1 bit.
- *Register address* fields, used for selecting a specific register from a (predetermined by a control field) register file whose value to use as an operand for the operation or to which a result will be put. Since they have to address either a general purpose register or a condition code register, they are either 4 or 3 bits in size, respectively.
- *Immediate value* fields, used to directly supply a value (of a limited numeric range) to use as an operand for the operation. Much variation in the size of these fields is required in order to provide maximum versatility with the limited space. As a result, immediate value fields exist with sizes of 8, 16, or 20 bits.

A negation selector control field always accompanies a condition code register address field, and so the pair is 4 bits in size overall. This was a strategic choice such that every component of the formats is a multiple of 4 bits in size, making it viable for a human to read machine code in hexadecimal format.

3.2 Format Nomenclature

Each of the formats has been given a shorthand name derived from its features. This name is split into four sections, each encoding a specific feature of the format, in terms of what fields it includes.

1. The first can either be `W` or `H`, signifying the total size of the instruction, which can be 32 (word) or 16 (half) bits respectively.
2. Afterwards comes the specifier for the amount of condition code register address fields that the format carries. For each one, a `C` is placed in the name. In practice the amount ranges from zero up to two fields.
3. Next comes the specifier for the amount of general purpose register address fields. For each one, a `G` is placed in the name. However if the amount is more than two, use a digit to prefix the letter instead. In practice the amount ranges from zero up to three fields.
4. Lastly comes the, also optional, specifier for the immediate value field. If it is present, an `I` is placed in the name, and then an `L`, `H`, `B`, or `S` to signify the size of the immediate value, which can be 20 (long), 16 (half), 8 (byte), or 4 (small) bits in length, respectively.

As a regular expression, the above rules would be: `[WH](C?C?)(G|GG|3G)?(I[LHBS])?`

3.3 Format Labeling

In the instruction format illustrations that follow, each field will also contain a label to identify itself and so that in the next chapter about the list of instructions themselves they can be given meaningful names.

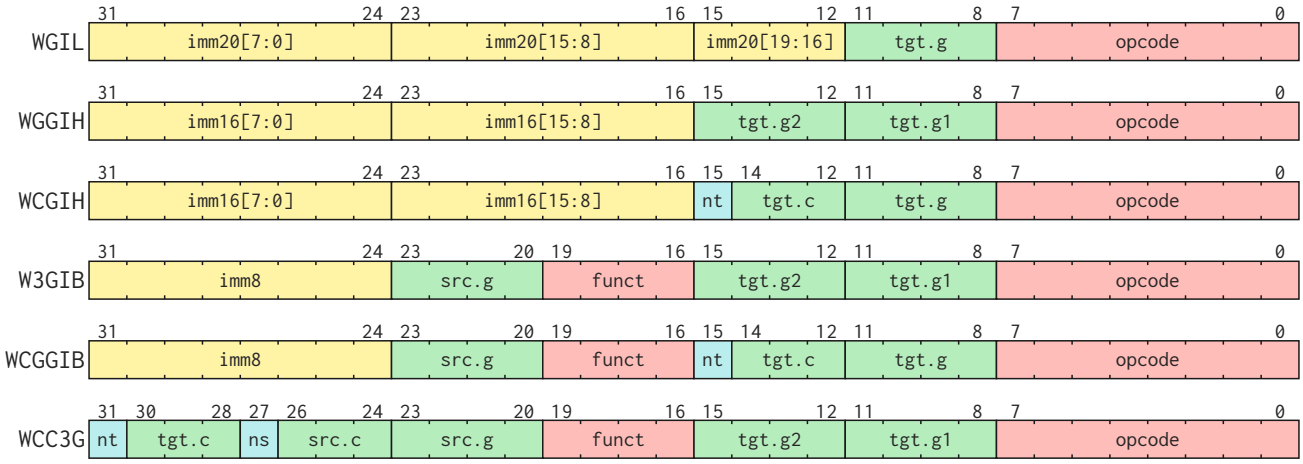
- Immediate value fields are labeled as `immN` where `N` is the total length of the field. Optionally, if the field is split up, there will follow a range tag in the form of `[H:L]` where `H` and `L` are the high and low indices respectively for the part of the field this section contains.
- Register address fields are more varied. They can either be labeled as `src`, which means this address is only used to read from a register (source), or as `tgt`, which means this address can be used for both reading from and writing to a register (target). Then follows either `.g` for general purpose register address fields or `.c` for condition code ones. Finally there may be a number to differentiate multiple instances.
- Control fields are much simpler since there's only the three aforementioned sub-types. Operation code fields are labeled `opcode`, function fields are labeled `funct`, and negation selector fields are labeled either `nt` or `ns` depending whether their companion register address field is either a target or a source address, respectively.

> Immediate value fields get split up and shuffled around and thus require range specifiers in their labeling because an attempt was made to maximize the overlap between different formats. Specifically, the field is split byte-wise and then reversed, so that bigger versions of the field overlap their low parts with the smaller versions and thus hardware requirements are reduced.

3.4 The List

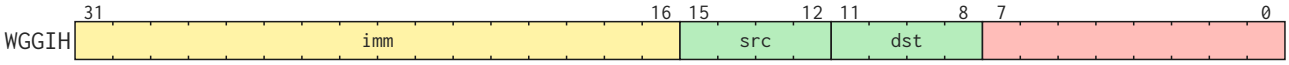
Having covered the composition and nomenclature behind the instruction formats, the list of instruction formats that PHINIX+ actually includes are shown in this section. There exist a total of six of them, accommodating the whole spectrum of needs, from those that require just register address fields to those that require just one big immediate value field.

> Each field has been color-coded for ease of pattern recognition. Specifically, control fields are colored red or teal, register address fields are colored green, and immediate value fields are colored yellow.



4 The Instruction Set

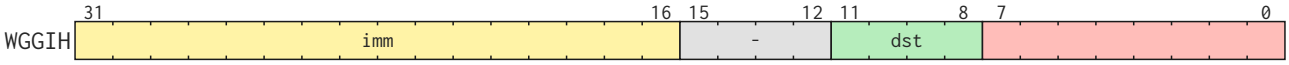
4.1 AUI – Add Upper Immediate



Description Shift the immediate value `imm` left 16 bits, add it to the data register `src` and store the result to the data register `dst`.

Effect `DRF[dst] = (imm << 16) + DRF[src]`

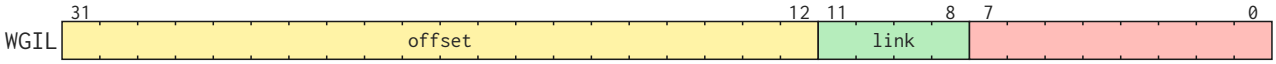
4.2 REL – Add Upper Immediate and Instruction Pointer



Description Shift the immediate value `imm` left 16 bits, add it to the address of the next instruction (instruction pointer plus 4) and store the result to the data register `dst`.

Effect `DRF[dst] = (imm << 16) + $ip + 4`

4.3 JNLi – Unconditional Far Jump and Link, Immediate Displacement

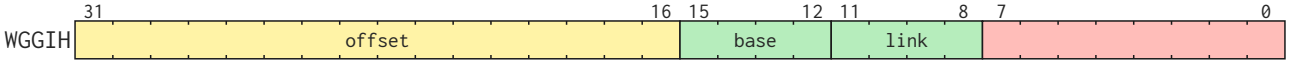


Description Save the address of the next instruction (instruction pointer plus 4) to the data register `link`. Add the sign-extended immediate value `offset` to the instruction pointer.

Effect `DRF[link] = $ip + 4 , $ip += SXT(offset << 1)`

Extras The `offset` is in increments of 16 bits (half-words) and so is shifted left by 1. Thus the attainable range of the jump is ± 1 MiB.

4.4 JNLr – Unconditional Jump and Link, Register + Immediate

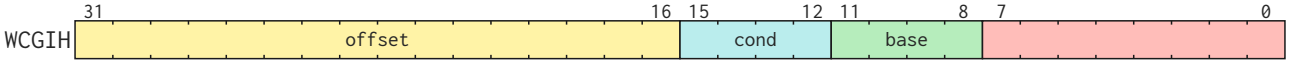


Description Save the address of the next instruction (instruction pointer plus 4) to the data register `link`. Add the data register `base` to the sign-extended immediate value `offset` and store the result to the instruction pointer.

Effect `DRF[link] = $ip + 4 , $ip = SXT(offset << 1) + DRF[base]`

Extras The `offset` is in increments of 16 bits (half-words) and so is shifted left by 1.

4.5 JMP – Conditional Jump, Register + Immediate



Description Conditionally on the optionally negated condition code register `cond`, add the data register `base` to the sign-extended immediate value `offset` and store the result to the instruction pointer.

Effect `IF cond { $ip = SXT(offset << 1) + DRF[base] }`

Extras The `offset` is in increments of 16 bits (half-words) and so is shifted left by 1.