The PHINIX+ System Architecture Documentation
Volume 1: The CPU

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

Version: 0.7.1
Date: 20th of February 2026
by Martin Andronikos

Contents

1 Introduction

Work In Progress

2 Registers

A register is an individually addressable memory cell that serves to store some core piece of data regarding the state of the processor. As a result they closely coupled with the operation and structure of the processor's architecture. A register file, then, is a grouping of these registers that serve a common purpose. PHINIX+ defines two of these register files as well as a few extra registers not part of a single register file, each one serving a different purpose.

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 Register File

The general purpose registers constitute the core of the programmer's model of PHINIX+. Every operation that manipulates some piece of data has to act on one of these registers – address generation, mathematical operations, memory and peripheral access, to name a few of them. Concretely, 16 of them are provided, each one being 32 bits in width, denoted *$gN* (where N is a single hexadecimal digit ranging from 0 to 9 and then from A to F).

The quantity of general purpose registers PHINIX+ provides, 16, while not the most common choice amongst RISC designs (that would be 32), is still in line with established ones like RV32E and ARM32.

It is important to note that the first data register, *$g0* is not actually a register one can write to. Register *$g0* 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 *$g0* 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 *$g0*.
- The most common operation, either as a standalone instruction or as a component in the function of more complex instructions, is addition. By having a constant-zero value (the neutral element of addition) available all the times allows addition to be effectively bypassed, expanding the usage of many instructions. For example, a move instruction can be achieved by doing an addition with one of the operands being *$g0*.

2.2 Condition Code Register File

The condition code, or "flag" registers exit to hold intermediate condition codes for the purpose of program control flow (predicated / conditional instructions not just limited to jumps) but they are also dedicated to doing bit-precise operations. Specifically, 8 of them are provided, each one being 1 bit in width, denoted *$cN* (where N is a single octal digit, ranging from 0 to 7).

Unlike with data register *$g0*, condition code register *$c0* does not hold a constant zero value.[1] Every register in this register file is accessible.

Having a constant zero condition register would reduce the needed variants for the instructions that handle them, however the importance of 8, and not just 7 available registers was important to make the bit-precise operations viable, since a byte contains 8 bits.

[1] This lack is addressed instead through the recommended calling convention provided in Volume TODO.

2.3 Special Purpose Registers

Apart from the main set of registers that are grouped into the two register files that have previously been analyzed, there's a few extra registers not part of a register file that exist to serve a specific purpose. Their exact number changes depending on which modules the specific implementation of PHINIX+ decides to also implement. The complete list is as follows:

- *$ip* is the address of the currently executing instruction – it stands for *Instruction Pointer*. It is automatically incremented during the process of instruction fetch, or directly written to by control flow instructions. Every implementation has to include this register.
- *$jp* stores the return address used when handling interrupts. It is only accessible when in privileged mode. It is automatically updated to the value of *$ip* when entering an interrupt. Only implementations with the IPM module include this register.[2]

$jp is called that because it's closely related to *$ip* and the letter J follows the letter I in the alphabet.

2.4 Table Summary

After having explained each of the categories, bellow follows a table that summarizes the complete set of registers that the PHINIX+ architecture makes available to the programmer.

General Purpose Registers
$g0
$g1
$g2
$g3
$g4
$g5
$g6
$g7
$g8
$g9
$gA
$gB
$gC
$gD
$gE
$gF

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

Special Purpose Registers
$ip
$jp

Table 1: PHINIX+'s complete set of registers

[2] Details on the IPM module are provided in Volume 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 (source) or to which a result will be put (destination). 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 4, 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. The purpose of the negation selector is to optionally invert the flag involved with the associated register.

It was a strategic choice 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 parts, each encoding a specific feature of the format, in terms of what fields it includes.

1. First there's a `W` or an `H` to signify the total size of the instruction, 32 and 16 bits respectively.[3]
2. Afterwards appears a `C` for each of the zero to two condition code register address fields.
3. Next appears a `G` for each of the zero to three general purpose register address fields. In the case of three, the count (only `3` in this case) prefixes the symbol instead of it being repeated.
4. Lastly, in formats including an immediate, comes an `I` and then either an `S`, a `B`, an `H`, or an `L`, corresponding to one of the aforementioned respective sizes.[4]

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

[3] While not in the base instruction set, reservations exist for half-width "compressed" instruction formats.
[4] One of those reservations for half-width instructions being the 4-bit "small" immediate.

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.

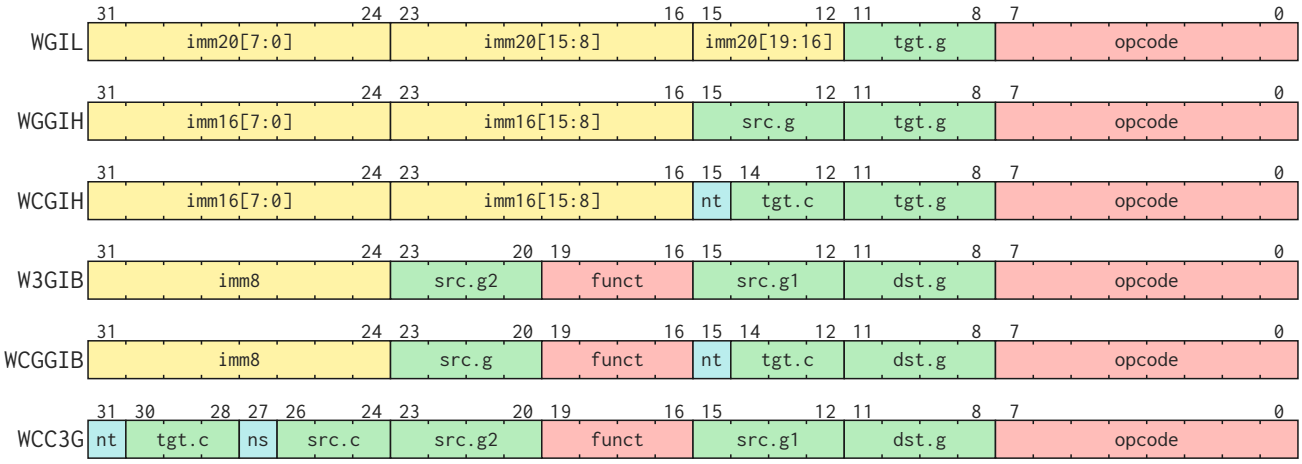
- Immediate value fields are labeled as `immN` where `N` is the total length of the field. Optionally, if the field is split up, there will follow a range tag in the form of `[H:L]` where `H` and `L` are the high and low indices respectively for the part of the field this section contains.
- Register 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 List of Formats

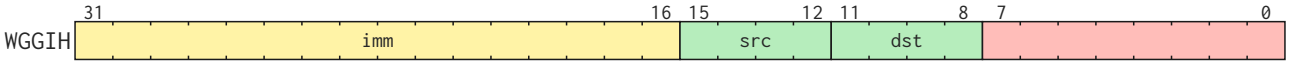
Having covered the composition and nomenclature behind the instruction formats, the list of instruction formats that PHINIX+ actually includes are shown in this section. There exist a total of six of them, accommodating the whole spectrum of needs, from those that require just register address fields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

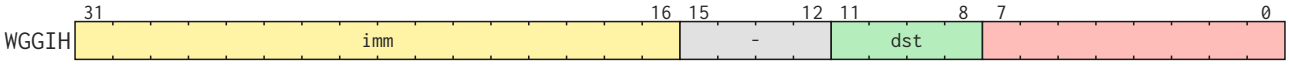
4.1 AUI – Add Upper Immediate



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]`

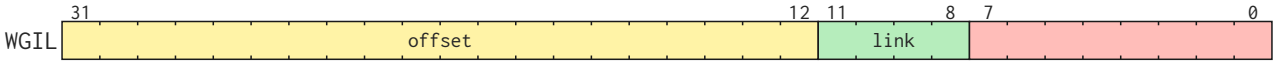
4.2 REL – Add Upper Immediate and Instruction Pointer



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

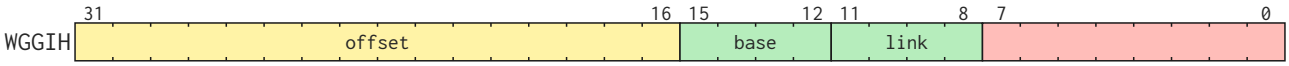


Description Save the address of the next instruction (instruction pointer plus 4) to the data register `link`. Add the sign-extended immediate value `offset` to the instruction pointer.

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

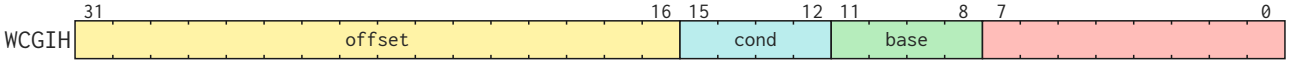


Description Save the address of the next instruction (instruction pointer plus 4) to the data register `link`. Add the data register `base` to the sign-extended immediate value `offset` and store the result to the instruction pointer.

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.