
OSACA

Release 0.4.13

Mar 24, 2023

Contents:

1	OSACA	3
1.1	Open Source Architecture Code Analyzer	3
2	Getting started	5
2.1	Installation	5
2.2	Dependencies:	5
3	Design	7
4	Usage	9
4.1	Throughput & Latency analysis	10
4.2	Marker insertion	10
4.3	Benchmark import	11
4.4	Database check	13
5	Examples	15
6	Credits	17
7	License	19
8	API Reference	21
8.1	osaca package	21
	Python Module Index	29
	Index	31



1.1 Open Source Architecture Code Analyzer

For an innermost loop kernel in assembly, this tool allows automatic instruction fetching of assembly code and automatic runtime prediction including throughput analysis and detection for critical path and loop-carried dependencies.

2.1 Installation

On most systems with python pip and setuputils installed, just run:

```
pip install --user osaca
```

for the latest release.

To build OSACA from source, clone this repository using `git clone https://github.com/RRZE-HPC/OSACA` and run in the root directory:

```
python ./setup.py install
```

After installation, OSACA can be started with the command `osaca` in the CLI.

2.2 Dependencies:

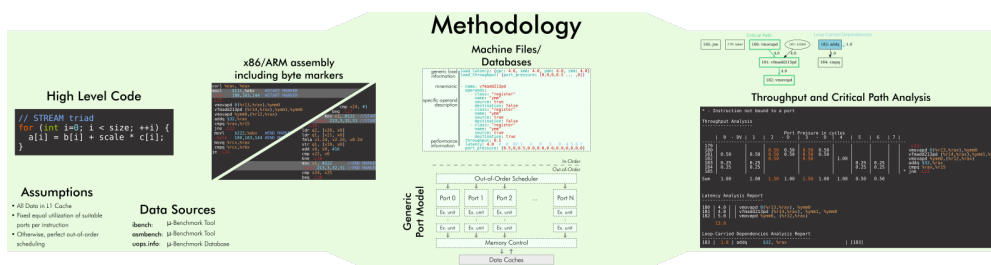
Additional requirements are:

- [Python3](#)
- [Graphviz](#) for dependency graph creation (minimal dependency is *libgraphviz-dev* on Ubuntu)
- [Kerncraft](#) `>=v0.8.4` for marker insertion
- [ibench](#) or [asmbench](#) for throughput/latency measurements

CHAPTER 3

Design

A schematic design of OSACA's workflow is shown below:



The usage of OSACA can be listed as:

```
osaca [-h] [-V] [--arch ARCH] [--fixed] [--db-check]
      [--import MICROBENCH] [--insert-marker]
      [--export-graph GRAPHNAME] [--ignore-unknown] [--verbose]
      FILEPATH
```

- h, --help** prints out the help message.
- V, --version** shows the program's version number.
- arch ARCH** needs to be replaced with the target architecture abbreviation. Possible options are SNB, IVB, HSW, BDW, SKX and CSX for the latest Intel micro architectures starting from Intel Sandy Bridge and ZEN1, ZEN2 for AMD Zen architectures. Furthermore, TX2 for Marvell's ARM-based ThunderX2 architecture is available.
- fixed** Run the throughput analysis with fixed port utilization for all suitable ports per instruction. Otherwise, OSACA will print out the optimal port utilization for the kernel.
- db-check** Run a sanity check on the by "--arch" specified database. The output depends on the verbosity level. Keep in mind you have to provide an existing (dummy) filename in anyway.
- import MICROBENCH** Import a given microbenchmark output file into the corresponding architecture instruction database. Define the type of microbenchmark either as "ibench" or "asmbench".
- insert-marker** OSACA calls the Kerncraft module for the interactively insertion of **IACA** byte markers or OSACA AArch64 byte markers in suggested assembly blocks.
- export-graph EXPORT_PATH** Output path for .dot file export. If "." is given, the file will be stored as ".osaca_dg.dot". After the file was created, you can convert it to a PDF file using **dot**.

--ignore-unknown	Force OSACA to apply a throughput and latency of 0.0 cy for all unknown instruction forms. If not specified, a warning will be printed instead if one or more instruction form is unknown to OSACA.
-v, --verbose	Increases verbosity level

The **FILEPATH** describes the filepath to the file to work with and is always necessary

Hereinafter OSACA's scope of function will be described.

4.1 Throughput & Latency analysis

As main functionality of OSACA, the tool starts the analysis on a marked assembly file by running the following command with one or more of the optional parameters:

```
osaca --arch ARCH [--fixed] [--ignore-unknown]
                        [--export-graph EXPORT_PATH]
      file
```

The `file` parameter specifies the target assembly file and is always mandatory.

The parameter `ARCH` is positional for the analysis and must be replaced by the target architecture abbreviation.

OSACA assumes an optimal scheduling for all instructions and assumes the processor to be able to schedule instructions in a way that it achieves a minimal reciprocal throughput. However, in older versions ($\leq v0.2.2$) of OSACA, a fixed probability for port utilization was assumed. This means, instructions with N available ports for execution were scheduled with a probability of $1/N$ to each of the ports. This behavior can be enforced by using the `--fixed` flag.

If one or more instruction forms are unknown to OSACA, it refuses to print an overall throughput, CP and LCD analysis and marks all unknown instruction forms with X next to the mnemonic. This is done so the user does not miss out on this unrecognized instruction and might assume an incorrect runtime prediction. To force OSACA to apply a throughput and latency of 0.0 cy for all unknown instruction forms, the flag `--ignore-unknown` can be specified.

To get a visualization of the analyzed kernel and its dependency chains, OSACA provides the option to additionally produce a graph as DOT file, which represents the kernel and all register dependencies inside of it. The tool highlights all LCDs and the CP. The graph generation is done by running OSACA with the `--export-graph EXPORT_GRAPH` flag. OSACA stores the DOT file either at the by `EXPORT_GRAPH` specified filepath or uses the default filename "osaca_dg.dot" in the current working directory. Subsequently, the DOT-graph can be adjusted in its appearance and converted to various output formats such as PDF, SVG, or PNG using the [dot command](#), e.g., `dot -Tpdf osaca_dg.dot -o graph.pdf` to generate a PDF document.

4.2 Marker insertion

For extracting the right kernel, one has to mark it in beforehand. Currently, only the detection of markers in the assembly code and therefore the analysis of assembly files is supported by OSACA.

Marking a kernel means to insert the byte markers in the assembly file in before and after the loop. For this, the start marker has to be inserted right in front of the loop label and the end marker directly after the jump instruction. IACA requires byte markers since it operates on opcode-level. To provide a trade-off between reusability for such tool and convenient usability, OSACA supports both byte markers and comment line markers. While the byte markers for x86 are equivalent to IACA byte markers, the comment keywords `OSACA-BEGIN` and `OSACA-END` are based on LLVM-MCA's markers.

4.2.1 x86 markers

Byte markers

```
movl    $111,%ebx      #IACA/OSACA START MARKER
.byte   100,103,144    #IACA/OSACA START MARKER
.loop:
# loop body
jb      .loop
movl    $222,%ebx      #IACA/OSACA END MARKER
.byte   100,103,144    #IACA/OSACA END MARKER
```

Comment line markers

```
# OSACA-BEGIN
.loop:
# loop body
jb      .loop
# OSACA-END
```

4.2.2 AArch64 markers

Byte markers

```
mov     x1, #111        // OSACA START
.byte   213,3,32,31     // OSACA START
.loop:
// loop body
b.ne    .loop
mov     x1, #222        // OSACA END
.byte   213,3,32,31     // OSACA END
```

Comment line markers

```
// OSACA-BEGIN
.loop:
// loop body
b.ne    .loop
// OSACA-END
```

OSACA in combination with Kerncraft provides a functionality for the automatic detection of possible loop kernels and inserting markers. This can be done by using the `--insert-marker` flag together with the path to the target assembly file and the target architecture.

4.3 Benchmark import

OSACA supports the automatic integration of new instruction forms by parsing the output of the micro- benchmark tools `asmbench` and `ibench`. This can be achieved by running OSACA with the command line option `--import MICROBENCH`:

```
osaca --arch ARCH --import MICROBENCH file
```

MICROBENCH specifies one of the currently supported benchmark tools, i.e., “asmbench” or “ibench”. ARCH defines the abbreviation of the target architecture for which the instructions will be added and file must be the path to the

generated output file of the benchmark. The format of this file has to match either the basic command line output of `ibench`, e.g.,

```
[INSTRUCTION FORM]-TP:    0.500 (clock cycles)    [DEBUG - result: 1.000000]
[INSTRUCTION FORM]-LT:    4.000 (clock cycles)    [DEBUG - result: 1.000000]
```

or the command line output of `asmbench` including the name of the instruction form in a separate line at the beginning, e.g.:

```
[INSTRUCTION FORM]
Latency: 4.00 cycle
Throughput: 0.50 cycle
```

Note that there must be an empty line after each throughput measurement as part of the output so that one instruction form entry consists of four (4) lines.

To let OSACA import the instruction form with the correct operands, the naming conventions for the instruction form name must be followed:

- The first part of the name is the mnemonic and ends with the character “-” (not part of the mnemonic in the DB).
- The second part of the name are the operands. Each operand must be separated from another operand by the character “_”.
- For each **x86** operand, one of the following symbols must be used:
 - “r” for general purpose registers (rax, edi, r9, ...)
 - “x”, “y”, or “z” for xmm, ymm, or zmm registers, respectively
 - “i” for immediates
 - “m” for a memory address. Add “b” if the memory address contains a base register, “o” if it contains an offset, “i” if it contains an index register, and “s” if the index register additionally has a scale factor of *more* than 1.
- For each **AArch64** operand, one of the following symbols must be used:
 - “w”, “x”, “b”, “h”, “s”, “d”, or “q” for registers with the corresponding prefix.
 - “v” followed by a single character (“b”, “h”, “s”, or “d”) for vector registers with the corresponding lane width of the second character. If no second character is given, OSACA assumes a lane width of 64 bit (d) as default.
 - “i” for immediates
 - “m” for a memory address. Add “b” if the memory address contains a base register, “o” if it contains an offset, “i” if it contains an index register, and “s” if the index register additionally has a scale factor of *more* than 1. Add “r” if the address format uses pre-indexing and “p” if it uses post-indexing.

Valid instruction form examples for x86 are `vaddpd-x_x_x, mov-r_mboi`, and `vfmadd213pd-mbis_y_y`.

Valid instruction form examples for AArch64 are `fadd-vd_vd_v, ldp-d_d_mo`, and `fmov-s_i`.

Note that the options to define operands are limited, therefore, one might need to adjust the instruction forms in the architecture DB after importing. OSACA parses the output for an arbitrary number of instruction forms and adds them as entries to the architecture DB. The user must edit the ISA DB in case the instruction form shows irregular source and destination operands for its ISA syntax. OSACA applies the following rules by default:

- If there is only one operand, it is considered as source operand
- In case of multiple operands the target operand (depending on the ISA syntax the last or first one) is considered to be the destination operand, all others are considered as source operands.

4.4 Database check

Since a manual adjustment of the ISA DB is currently indispensable when adding new instruction forms, OSACA provides a database sanity check using the `-db-check` flag. It can be executed via:

```
osaca --arch ARCH --db-check [-v] file
```

`ARCH` defines the abbreviation of the target architecture of the database to check. The `file` argument needs to be specified as it is positional but may be any existing dummy path. When called, OSACA prints a summary of database information containing the amount of missing throughput values, latency values or μ -ops assignments for an instruction form. Furthermore, it shows the amount of duplicate instruction forms in both the architecture DB and the ISA DB and checks how many instruction forms in the ISA DB are non-existent in the architecture DB. Finally, it checks via simple heuristics how many of the instruction forms contained in the architecture DB might miss an ISA DB entry. Running the database check including the `-v` verbosity flag, OSACA prints in addition the specific name of the identified instruction forms so that the user can check the mentioned incidents.

CHAPTER 5

Examples

For clarifying the functionality of OSACA a sample kernel is analyzed for an Intel CSX core hereafter:

```
double a[N], double b[N];
double s;

// loop
for(int i = 0; i < N; ++i)
    a[i] = s * b[i];
```

The code shows a simple scalar multiplication of a vector *b* and a floating-point number *s*. The result is written in vector *a*. After including the OSACA byte marker into the assembly, one can start the analysis typing

```
osaca --arch CSX PATH/TO/FILE
```

in the command line.

The output is:

```
Open Source Architecture Code Analyzer (OSACA) - v0.3
Analyzed file:      scale.s.csx.03.s
Architecture:      csx
Timestamp:         2019-10-03 23:36:21

P - Throughput of LOAD operation can be hidden behind a past or future STORE_
↳ instruction
* - Instruction micro-ops not bound to a port
X - No throughput/latency information for this instruction in data file
```

Combined Analysis Report

```
-----
                                Port pressure in cycles
      | 0 - 0DV | 1 | 2 - 2D | 3 - 3D | 4 | 5 | 6 | 7 _
↳ || CP | LCD |
```

(continues on next page)

(continued from previous page)

[illegible]

It shows the whole kernel together with the optimized port pressure of each instruction form and the overall port binding. Furthermore, in the two columns on the right, the critical path (CP) and the longest loop-carried dependency (LCD) of the loop kernel. In the bottom, all loop-carried dependencies are shown, each with a list of line numbers being part of this dependency chain on the right.

You can find more (already marked) examples and sample outputs for various architectures in the [examples](#) directory.

CHAPTER 6

Credits

Implementation: Jan Laukemann

CHAPTER 7

License

AGPL-3.0

8.1 osaca package

8.1.1 Subpackages

osaca.api package

Provides interfaces to other tools.

osaca.api.kerncraft_interface module

Module contents

osaca.parser package

Parser module for parsing the assembly code.

osaca.parser.attr_dict module

Attribute Dictionary to access dictionary entries as attributes.

```
class AttrDict (*args, **kwargs)
```

Bases: dict

```
    static convert_dict (dictionary)
```

Convert given dictionary to *AttrDict*.

Parameters *dictionary* (dict) – dict to be converted

Returns *AttrDict* representation of dictionary

osaca.parser.base_parser module

Parser superclass of specific parsers.

class BaseParser

Bases: object

COMMENT_ID = 'comment'

DIRECTIVE_ID = 'directive'

IMMEDIATE_ID = 'immediate'

LABEL_ID = 'label'

IDENTIFIER_ID = 'identifier'

MEMORY_ID = 'memory'

REGISTER_ID = 'register'

SEGMENT_EXT_ID = 'segment_extension'

INSTRUCTION_ID = 'instruction'

OPERANDS_ID = 'operands'

static detect_ISA (*file_content*)

Detect the ISA of the assembly based on the used registers and return the ISA code.

parse_file (*file_content*, *start_line*=0)

Parse assembly file. This includes *not* extracting of the marked kernel and the parsing of the instruction forms.

Parameters

- **file_content** (*str*) – assembly code
- **start_line** (*int*) – offset, if first line in *file_content* is meant to be not 1

Returns list of instruction forms

parse_line (*line*, *line_number*=None)

parse_instruction (*instruction*)

parse_register (*register_string*)

is_gpr (*register*)

is_vector_register (*register*)

get_reg_type (*register*)

construct_parser ()

process_operand (*operand*)

get_full_reg_name (*register*)

normalize_imd (*imd*)

is_reg_dependend_of (*reg_a*, *reg_b*)

osaca.parser.parser_AArch64v81 module**osaca.parser.parser_x86att module****class ParserX86ATT**

Bases: *osaca.parser.base_parser.BaseParser*

construct_parser()

Create parser for x86 AT&T ISA.

parse_register(register_string)

Parse register string

parse_line(line, line_number=None)

Parse line and return instruction form.

Parameters

- **line** (*str*) – line of assembly code
- **line_number** (*int*, *optional*) – default None, identifier of instruction form

Returns *dict* – parsed asm line (comment, label, directive or instruction form)

parse_instruction(instruction)

Parse instruction in asm line.

Parameters *instruction* (*str*) – Assembly line string.

Returns *dict* – parsed instruction form

process_operand(operand)

Post-process operand

process_directive(directive)**process_memory_address(memory_address)**

Post-process memory address operand

process_label(label)

Post-process label asm line

process_immediate(immediate)

Post-process immediate operand

get_full_reg_name(register)

Return one register name string including all attributes

normalize_imd(imd)

Normalize immediate to decimal based representation

is_flag_dependend_of(flag_a, flag_b)

Check if flag_a is dependent on flag_b

is_reg_dependend_of(reg_a, reg_b)

Check if reg_a is dependent on reg_b

is_basic_gpr(register)

Check if register is a basic general purpose register (ebi, rax, ...)

is_gpr(register)

Check if register is a general purpose register

is_vector_register (*register*)
Check if register is a vector register

get_reg_type (*register*)
Get register type

Module contents

Collection of parsers supported by OSACA.

Only the parser below will be exported, so please add new parsers to `__all__`.

get_parser (*isa*)

class AttrDict (**args, **kwargs*)
Bases: dict

static convert_dict (*dictionary*)
Convert given dictionary to *AttrDict*.

Parameters *dictionary* (*dict*) – dict to be converted

Returns *AttrDict* representation of dictionary

class BaseParser

Bases: object

COMMENT_ID = 'comment'

DIRECTIVE_ID = 'directive'

IMMEDIATE_ID = 'immediate'

LABEL_ID = 'label'

IDENTIFIER_ID = 'identifier'

MEMORY_ID = 'memory'

REGISTER_ID = 'register'

SEGMENT_EXT_ID = 'segment_extension'

INSTRUCTION_ID = 'instruction'

OPERANDS_ID = 'operands'

static detect_ISA (*file_content*)

Detect the ISA of the assembly based on the used registers and return the ISA code.

parse_file (*file_content, start_line=0*)

Parse assembly file. This includes *not* extracting of the marked kernel and the parsing of the instruction forms.

Parameters

- **file_content** (*str*) – assembly code
- **start_line** (*int*) – offset, if first line in *file_content* is meant to be not 1

Returns list of instruction forms

parse_line (*line, line_number=None*)

parse_instruction (*instruction*)

```

parse_register (register_string)
is_gpr (register)
is_vector_register (register)
get_reg_type (register)
construct_parser ()
process_operand (operand)
get_full_reg_name (register)
normalize_imd (imd)
is_reg_dependend_of (reg_a, reg_b)

class ParserX86ATT
    Bases: osaca.parser.base_parser.BaseParser

    construct_parser ()
        Create parser for x86 AT&T ISA.

    parse_register (register_string)
        Parse register string

    parse_line (line, line_number=None)
        Parse line and return instruction form.

        Parameters
        • line (str) – line of assembly code
        • line_number (int, optional) – default None, identifier of instruction form

        Returns dict – parsed asm line (comment, label, directive or instruction form)

    parse_instruction (instruction)
        Parse instruction in asm line.

        Parameters instruction (str) – Assembly line string.

        Returns dict – parsed instruction form

    process_operand (operand)
        Post-process operand

    process_directive (directive)

    process_memory_address (memory_address)
        Post-process memory address operand

    process_label (label)
        Post-process label asm line

    process_immediate (immediate)
        Post-process immediate operand

    get_full_reg_name (register)
        Return one register name string including all attributes

    normalize_imd (imd)
        Normalize immediate to decimal based representation

    is_flag_dependend_of (flag_a, flag_b)
        Check if flag_a is dependent on flag_b

```

is_reg_dependent_of (*reg_a*, *reg_b*)
Check if *reg_a* is dependent on *reg_b*

is_basic_gpr (*register*)
Check if register is a basic general purpose register (ebi, rax, ...)

is_gpr (*register*)
Check if register is a general purpose register

is_vector_register (*register*)
Check if register is a vector register

get_reg_type (*register*)
Get register type

class ParserAArch64

Bases: *osaca.parser.base_parser.BaseParser*

construct_parser ()
Create parser for ARM AArch64 ISA.

parse_line (*line*, *line_number=None*)
Parse line and return instruction form.

Parameters

- **line** (*str*) – line of assembly code
- **line_number** (*int*, *optional*) – identifier of instruction form, defaults to None

Returns *dict* – parsed asm line (comment, label, directive or instruction form)

parse_instruction (*instruction*)
Parse instruction in asm line.

Parameters **instruction** (*str*) – Assembly line string.

Returns *dict* – parsed instruction form

process_operand (*operand*)
Post-process operand

process_memory_address (*memory_address*)
Post-process memory address operand

process_sp_register (*register*)
Post-process stack pointer register

resolve_range_list (*operand*)
Resolve range or list register operand to list of registers. Returns None if neither list nor range

process_register_list (*register_list*)
Post-process register lists (e.g., {r0,r3,r5}) and register ranges (e.g., {r0-r7})

process_immediate (*immediate*)
Post-process immediate operand

process_label (*label*)
Post-process label asm line

process_identifier (*identifier*)
Post-process identifier operand

get_full_reg_name (*register*)
Return one register name string including all attributes

normalize_imd (*imd*)
 Normalize immediate to decimal based representation

ieee_to_float (*ieee_val*)
 Convert IEEE representation to python float

parse_register (*register_string*)

is_gpr (*register*)
 Check if register is a general purpose register

is_vector_register (*register*)
 Check if register is a vector register

is_flag_dependend_of (*flag_a, flag_b*)
 Check if flag_a is dependent on flag_b

is_reg_dependend_of (*reg_a, reg_b*)
 Check if reg_a is dependent on reg_b

get_reg_type (*register*)
 Get register type

osaca.semantics package

Semantic part of OSACA.

osaca.semantics.arch_semantics module

osaca.semantics.hw_model module

osaca.semantics.isa_semantics module

osaca.semantics.kernel_dg module

osaca.semantics.marker_utils module

Module contents

8.1.2 Submodules

8.1.3 osaca.db_interface module

8.1.4 osaca.frontend module

8.1.5 osaca.osaca module

8.1.6 osaca.utils module

find_datafile (*name*)
 Check for existence of name in user or package data folders and return path.



O

`osaca.parser`, [24](#)

`osaca.parser.attr_dict`, [21](#)

`osaca.parser.base_parser`, [22](#)

`osaca.parser.parser_x86att`, [23](#)

`osaca.utils`, [27](#)

A

`AttrDict` (class in `osaca.parser`), 24
`AttrDict` (class in `osaca.parser.attr_dict`), 21

B

`BaseParser` (class in `osaca.parser`), 24
`BaseParser` (class in `osaca.parser.base_parser`), 22

C

`COMMENT_ID` (*BaseParser* attribute), 22, 24
`construct_parser()` (*BaseParser* method), 22, 25
`construct_parser()` (*ParserAArch64* method), 26
`construct_parser()` (*ParserX86ATT* method), 23, 25
`convert_dict()` (*AttrDict* static method), 21, 24

D

`detect_ISA()` (*BaseParser* static method), 22, 24
`DIRECTIVE_ID` (*BaseParser* attribute), 22, 24

F

`find_datafile()` (in module `osaca.utils`), 27

G

`get_full_reg_name()` (*BaseParser* method), 22, 25
`get_full_reg_name()` (*ParserAArch64* method), 26
`get_full_reg_name()` (*ParserX86ATT* method), 23, 25
`get_parser()` (in module `osaca.parser`), 24
`get_reg_type()` (*BaseParser* method), 22, 25
`get_reg_type()` (*ParserAArch64* method), 27
`get_reg_type()` (*ParserX86ATT* method), 24, 26

I

`IDENTIFIER_ID` (*BaseParser* attribute), 22, 24
`ieee_to_float()` (*ParserAArch64* method), 27
`IMMEDIATE_ID` (*BaseParser* attribute), 22, 24
`INSTRUCTION_ID` (*BaseParser* attribute), 22, 24

`is_basic_gpr()` (*ParserX86ATT* method), 23, 26
`is_flag_dependend_of()` (*ParserAArch64* method), 27
`is_flag_dependend_of()` (*ParserX86ATT* method), 23, 25
`is_gpr()` (*BaseParser* method), 22, 25
`is_gpr()` (*ParserAArch64* method), 27
`is_gpr()` (*ParserX86ATT* method), 23, 26
`is_reg_dependend_of()` (*BaseParser* method), 22, 25
`is_reg_dependend_of()` (*ParserAArch64* method), 27
`is_reg_dependend_of()` (*ParserX86ATT* method), 23, 25
`is_vector_register()` (*BaseParser* method), 22, 25
`is_vector_register()` (*ParserAArch64* method), 27
`is_vector_register()` (*ParserX86ATT* method), 23, 26

L

`LABEL_ID` (*BaseParser* attribute), 22, 24

M

`MEMORY_ID` (*BaseParser* attribute), 22, 24

N

`normalize_imd()` (*BaseParser* method), 22, 25
`normalize_imd()` (*ParserAArch64* method), 26
`normalize_imd()` (*ParserX86ATT* method), 23, 25

O

`OPERANDS_ID` (*BaseParser* attribute), 22, 24
`osaca.parser` (module), 24
`osaca.parser.attr_dict` (module), 21
`osaca.parser.base_parser` (module), 22
`osaca.parser.parser_x86att` (module), 23
`osaca.utils` (module), 27

P

`parse_file()` (*BaseParser method*), 22, 24
`parse_instruction()` (*BaseParser method*), 22, 24
`parse_instruction()` (*ParserAArch64 method*), 26
`parse_instruction()` (*ParserX86ATT method*), 23, 25
`parse_line()` (*BaseParser method*), 22, 24
`parse_line()` (*ParserAArch64 method*), 26
`parse_line()` (*ParserX86ATT method*), 23, 25
`parse_register()` (*BaseParser method*), 22, 24
`parse_register()` (*ParserAArch64 method*), 27
`parse_register()` (*ParserX86ATT method*), 23, 25
`ParserAArch64` (*class in osaca.parser*), 26
`ParserX86ATT` (*class in osaca.parser*), 25
`ParserX86ATT` (*class in osaca.parser.parser_x86att*), 23
`process_directive()` (*ParserX86ATT method*), 23, 25
`process_identifier()` (*ParserAArch64 method*), 26
`process_immediate()` (*ParserAArch64 method*), 26
`process_immediate()` (*ParserX86ATT method*), 23, 25
`process_label()` (*ParserAArch64 method*), 26
`process_label()` (*ParserX86ATT method*), 23, 25
`process_memory_address()` (*ParserAArch64 method*), 26
`process_memory_address()` (*ParserX86ATT method*), 23, 25
`process_operand()` (*BaseParser method*), 22, 25
`process_operand()` (*ParserAArch64 method*), 26
`process_operand()` (*ParserX86ATT method*), 23, 25
`process_register_list()` (*ParserAArch64 method*), 26
`process_sp_register()` (*ParserAArch64 method*), 26

R

`REGISTER_ID` (*BaseParser attribute*), 22, 24
`resolve_range_list()` (*ParserAArch64 method*), 26

S

`SEGMENT_EXT_ID` (*BaseParser attribute*), 22, 24