DrMIPS – Configuration Manual

Bruno Nova

September 17, 2015

Contents

1 Introduction **4**

2 CPU file **5**

2.1 components 6

2.1.1 Add 6

2.1.2 ALU 7

2.1.3 ALU_Control 7

2.1.4 And 7

2.1.5 Concat 7

2.1.6 Const 8

2.1.7 Control 8

2.1.8 Dist 8

2.1.9 DMem 9

2.1.10 Ext_ALU 9

2.1.11 Fork 9

2.1.12 Fwd_Unit 10

2.1.13 Hzd_Unit 10

2.1.14 IMem 10

2.1.15 Mux 11

2.1.16 Not 11

2.1.17 Or 11

2.1.18 PipeReg 11

2.1.19 PC 12

2.1.20 RegBank 12

2.1.21 SExt 12

2.1.22 SLL 13

2.1.23 Xor 13

2.1.24 ZExt 13

2.2 wires 14

2.3 reg_names 14

2.4 instructions 15

3 Instruction set file **16**
3.1 types 17
3.2 instructions 17
3.3 pseudo 18
3.4 control 19
3.5 alu 19

Chapter 1

Introduction

DrMIPS provides several unicycle and pipeline MIPS datapaths. These datapaths are defined in JSON (`http://json.org/`) files, and have the `.cpu` extension. These CPU files can be modified, and new ones can be created.

The instruction sets used by the datapaths are also defined in JSON files, having the `.set` extension. These can also be created and modified.

This manual explains, with some detail, the syntax of both of these files. Chapter 2 explains the syntax of the CPU files while Chapter 3 explains the syntax of the instruction set files.

Chapter 2

CPU file

The different versions of the MIPS CPU are defined in CPU files. These can be edited/configured and additional ones can be created. This chapter explains the syntax of these files.

The CPU files are formatted in JSON. A partial example of a CPU file is shown below:

```
{
  "components": {
    "MUX_DST": {"type": "mux", "x": 205, "y": 260, "size": 5, "
        sel": "RegDst", "out": "OUT", "in": ["0", "1"], "desc":
         {"default": "Selects rt or rd as the destination
        register.", "pt": "Selecciona o rt ou rd como registo
        de destino."}},
    ...
  },
  "wires": {
    {"from": "DIST_INST", "out": "15-11", "to": "MUX_DST", "in
        ": "1", "start": {"x": 185, "y": 270}, "points": [{"x":
         195, "y": 270}, {"x": 195, "y": 282}]},
    {"from": "MUX_DST", "out": "OUT", "to": "REG", "in": "
        WriteReg", "end": {"x": 250, "y": 277}},
    ...
  },
  "reg_names": ["zero", "at", "v0", "v1", "a0", "a1", "a2", "a3
      ", "t0", "t1", "t2", "t3", "t4", "t5", "t6", "t7", "s0",
      "s1", "s2", "s3", "s4", "s5", "s6", "s7", "t8", "t9", "k0
      ", "k1", "gp", "sp", "fp", "ra"],
  "instructions": "default.set"
}
```

The various sections that compose the CPU files are detailed in the following sections of this chapter.

2.1 components

This section defines all the components of the CPU and their properties. An example of the definition of a component is shown below:

```
1 "MUX_DST": {"type": "mux", "x": 205, "y": 260, "size": 5, "sel
      ": "RegDst", "out": "OUT", "in": ["0", "1"], "desc": {"
      default": "Selects rt or rd as the destination register.",
      "pt": "Selecciona o rt ou rd como registo de destino."}}
```

Each component is identified by a unique ID (`MUX_DST` in this example). The properties of the component are defined between curly braces as a JSON object. Many properties are specific to each type of component, but some exist for all components. These are:

- **type**: The type of the component. The component of the example is a multiplexer. The different types of components are explained next.
- **latency**: Optional. An integer with the latency of the component in ps. The default latency is 0 ps.
- **x**: The x-coordinate of the top-left corner of the component in the graphical datapath. The minimum value is 0 and corresponds to the left border of the datapath. There is no maximum value.
- **y**: The y-coordinate of the top-left corner of the component in the graphical datapath. The minimum value is 0 and corresponds to the top border of the datapath. There is no maximum value.
- **desc**: Optional. Component specific description, in each language, shown in the tooltip of the component. The value is a JSON object where the description for each language is defined in the form `"language_code": "Description."`. The `language_code` identifier is the code of the language, like `pt` or `pt_BR`. The special language code `default` should define the default description in English that is used when the language-specific description is not available.

The following subsections explain the different types of components available and their specific properties. The titles of the subsections are the values that should be written in the `type` property of the components. This property is case-insensitive.

2.1.1 Add

An adder that sums the values of the inputs. The specific properties are:

- **in1**: Identifier of the first input.

- **in2**: Identifier of the second input.
- **out**: Identifier of the output.

2.1.2 ALU

The basic ALU. Only one ALU or Extended ALU can be present. The specific properties are:

- **in1**: Identifier of the first input.
- **in2**: Identifier of the second input.
- **control**: Identifier of the control input (that selects the operation to perform).
- **out**: Identifier of the result output.
- **zero**: Identifier of the 1 bit *zero* output.

2.1.3 ALU_Control

The component that controls the ALU. Only one can be present. The specific properties are:

- **aluop**: Identifier of the *ALUOp* input.
- **func**: Identifier of the *func* input (from the function field of the instruction).

2.1.4 And

A logical *AND* port. The specific properties are:

- **in1**: Identifier of the first input.
- **in2**: Identifier of the second input.
- **out**: Identifier of the output.

2.1.5 Concat

A "concatenator" that concatenates the values of the two inputs into a single output. The value of the output is the concatenation of the value of the first input (as higher order bits) with the value of the second input (as lower order bits). The size of the output is equal to the sum of the sizes of the inputs. The specific properties are:

- **in1**: Properties of the first input.

- **in2**: Properties of the second input.
- **out**: Identifier of the output.

The properties of both inputs are defined as JSON objects. The properties of these objects are:

- **id**: Identifier of the input.
- **size**: Size of the input (in bits).

2.1.6 Const

A component that outputs a constant value. The specific properties are:

- **out**: Identifier of the output.
- **val**: The constant value.
- **size**: Size of the output (in bits).

2.1.7 Control

The control unit. The datapath must have one control unit. This component has only one specific property: **in**, which is the identifier of the input.

2.1.8 Dist

This component distributes the bits of the input through several outputs. The specific properties are:

- **in**: Properties of the input as a JSON object. The properties of the object are:
  - **id**: Identifier of the input.
  - **size**: Size of the input (in bits).
- **out**: Properties of the outputs as a JSON array. Each element of the array defines the properties of an output as a JSON object. The properties of the objects of the array are:
  - **msb**: The index of the most significant bit from the input.
  - **lsb**: The index of the less significant bit from the input.
  - **id**: Optional. Identifier of the output. If omitted, the identifier corresponds to "`<msb>-<lsb>`".

2.1.9 DMem

The data memory. Only one can be present. The specific properties are:

- **size**: Size of the memory (number of 32 bits memory positions).
- **address**: Identifier of the *Address* input.
- **write_data**: Identifier of the *WriteData* input.
- **out**: Identifier of the output.
- **mem_read**: Identifier of the *MemRead* control input.
- **mem_write**: Identifier of the *MemWrite* control input.

2.1.10 Ext_ALU

An extended ALU. This ALU stores the `hi` and `lo` registers and is capable of calculating multiplications and divisions. Only one ALU or Extended ALU can be present. The specific properties are the same as the basic ALU:

- **in1**: Identifier of the first input.
- **in2**: Identifier of the second input.
- **control**: Identifier of the control input (that selects the operation to perform).
- **out**: Identifier of the result output.
- **zero**: Identifier of the 1 bit *zero* output.

2.1.11 Fork

This component forks a wire into several other wires with the same size. The specific properties are:

- **in**: Identifier of the input.
- **size**: Size of the input and outputs (in bits).
- **out**: Array with the identifiers of the outputs.

2.1.12 Fwd_Unit

The pipeline forwarding unit. Only one can be present. The specific properties are:

- **ex_mem_reg_write**: Identifier of the *EX/MEM.RegWrite* input.
- **mem_wb_reg_write**: Identifier of the *MEM/WB.RegWrite* input.
- **ex_mem_rd**: Identifier of the *EX/MEM.Rd* input.
- **mem_wb_rd**: Identifier of the *MEM/WB.Rd* input.
- **id_ex_rs**: Identifier of the *ID/EX.Rs* input.
- **id_ex_rt**: Identifier of the *ID/EX.Rt* input.
- **fwd_a**: Identifier of the *ForwardA* output.
- **fwd_b**: Identifier of the *ForwardB* output.

2.1.13 Hzd_Unit

The pipeline hazard detection unit. Only one can be present. The specific properties are:

- **id_ex_mem_read**: Identifier of the *ID/EX.MemRead* input.
- **id_ex_rt**: Identifier of the *ID/EX.Rt* input.
- **if_id_rs**: Identifier of the *IF/ID.Rs* input.
- **if_id_rt**: Identifier of the *IF/ID.Rt* input.
- **stall**: Identifier of the output.

2.1.14 IMem

The instruction memory. The datapath must have one instruction memory. The specific properties are:

- **in**: Identifier of the input.
- **out**: Identifier of the output.

2.1.15 Mux

A multiplexer. The specific properties are:

- **size**: The size of the inputs and output (in bits).
- **sel**: Identifier of the selector input.
- **out**: Identifier of the output.
- **in**: Array with the identifiers of the inputs.

2.1.16 Not

A logical *NOT* port. The specific properties are:

- **in**: Identifier of the input.
- **out**: Identifier of the output.

2.1.17 Or

A logical *OR* port. The specific properties are:

- **in1**: Identifier of the first input.
- **in2**: Identifier of the second input.
- **out**: Identifier of the output.

2.1.18 PipeReg

A pipeline register that separates two stages of the pipeline. A pipelined datapath must have exactly 4 of these registers (corresponding to a 5-stage pipeline). Additionally, the identifiers of these components must be: `IF/ID`, `ID/EX`, `EX/MEM` and `MEM/WB`. The specific properties are:

- **regs**: Definition of the registers recorded. The value is a JSON object where each property defines a register: the identifier is the identifier of the register and corresponding input and output, and the value is the size of the register (in bits).
- **flush**: Optional. Identifier of the *Flush* control input.
- **write**: Optional. Identifier of the *Write* control input.

2.1.19 PC

The program counter. The datapath must have one program counter. The specific properties are:

- **in**: Identifier of the input.
- **out**: Identifier of the output.
- **write**: Optional. Identifier of the *Write* control input.

2.1.20 RegBank

The register bank. The datapath must have one register bank. The specific properties are:

- **num_regs**: The number of registers. Must be greater than 1 and a power of 2.
- **read_reg1**: Identifier of the *ReadReg1* input.
- **read_reg2**: Identifier of the *ReadReg2* input.
- **read_data1**: Identifier of the *ReadData1* output.
- **read_data2**: Identifier of the *ReadData2* output.
- **write_reg**: Identifier of the *WriteReg* input.
- **write_data**: Identifier of the *WriteData* input.
- **reg_write**: Identifier of the *RegWrite* control input.
- **forwarding**: Optional. If `true`, the register bank will use internal forwarding (for pipelined datapaths).
- **const_regs**: Optional. JSON array that defines the constant registers. Each element can be either the index of the register or a JSON object with the following properties:
  - **reg**: Index of the register.
  - **val**: The constant value of the register.

2.1.21 SExt

A sign extender. The specific properties are:

- **in**: Properties of the input.
- **out**: Properties of the output.

The properties of both the input and output are defined as JSON objects. The properties of these objects are:

- **id**: Identifier of the input/output.
- **size**: Size of the input/output (in bits).

2.1.22 SLL

A shift-left logical. The specific properties are:

- **in**: Properties of the input.
- **out**: Properties of the output.
- **amount**: Number of bits to shift left.

The properties of both the input and output are defined as JSON objects. The properties of these objects are:

- **id**: Identifier of the input/output.
- **size**: Size of the input/output (in bits).

2.1.23 Xor

A logical *XOR* port. The specific properties are:

- **in1**: Identifier of the first input.
- **in2**: Identifier of the second input.
- **out**: Identifier of the output.

2.1.24 ZExt

A zero extender. The specific properties are:

- **in**: Properties of the input.
- **out**: Properties of the output.

The properties of both the input and output are defined as JSON objects. The properties of these objects are:

- **id**: Identifier of the input/output.
- **size**: Size of the input/output (in bits).

2.2 wires

This section defines all the wires that connect the components of the CPU. An example of the definition of a wire is shown below:

```
1 {"from": "DIST_INST", "out": "15-11", "to": "MUX_DST", "in":
    "1", "start": {"x": 185, "y": 270}, "points": [{"x": 195, "
    y": 270}, {"x": 195, "y": 282}], "end": {"x": 205, "y":
    282}}
```

Each wire connects an output of a component to an input of another component. A wire is defined as a JSON object with several properties. These are:

- **from**: The ID of the component that the wire connects from (origin).
- **out**: The ID of output of the origin component that the wire connects from.
- **to**: The ID of the component that the wire connects to (destination).
- **in**: The ID of input of the destination component that the wire connects to.
- **start**: Optional. A JSON object that defines the start position of the wire in the graphical datapath, if the default one is unsuitable.
- **points**: Optional. An array of JSON objects that define the positions of the intermediate points of the wire, if desired.
- **end**: Optional. A JSON object that defines the end position of the wire in the graphical datapath, if the default one is unsuitable.

The positions used in the `start`, `points` and `end` properties above are JSON objects with two integer properties: `x` and `y`. Each input and output of each component is, by default, "attached" to one of the four sides of the component. The positions of the inputs and outputs in the datapath and, thus, the start and end positions of the connected wires, are calculated automatically[1] but can be overwritten by the `start` and `end` properties.

2.3 reg_names

This **optional** section defines the "friendly" names of the registers (i.e. `$zero`, `$t0`, etc.). The value is an array of strings that defines the names of

[1] The inputs and outputs on each side of a component are, by default, ordered alphabetically by their IDs.

the registers, from register `$0` to the last one, **without the leading dollar sign**. The registers can always be referred by their indexes (`$0`, `$1`, etc.) in the simulator.

2.4 instructions

This section declares the instruction set that the CPU uses. The value is the **relative** path to the desired instruction set file. These files are explained in the next chapter.

Chapter 3

Instruction set file

The instruction sets used by the different versions of the MIPS CPU are defined in instruction set files. These can be edited/configured and additional ones can be created. This chapter explains the syntax of these files.

Like the CPU files, the instruction set files are formatted in JSON. A partial example of an instruction set file is shown below:

```
{
  "types": {
    "R": [{"id": "op", "size": 6}, {"id": "rs", "size": 5}, {"
        id": "rt", "size": 5}, {"id": "rd", "size": 5}, {"id":
        "shamt", "size": 5}, {"id": "func", "size": 6}],
    "I": [{"id": "op", "size": 6}, {"id": "rs", "size": 5}, {"
        id": "rt", "size": 5}, {"id": "imm", "size": 16}],
    "J": [{"id": "op", "size": 6}, {"id": "target", "size":
        26}]
  },
  "instructions": {
    "add":  {"type": "R", "args": ["reg", "reg", "reg"], "
        fields": {"op": 0, "rs": "#2", "rt": "#3", "rd": "#1",
        "shamt": 0, "func": 32}, "desc": "$t1 = $t2 + $t3"},
    ...
  },
  "pseudo": {
    "move": {"args": ["reg", "reg"], "to": ["add #1, #2, $0"],
        "desc": "$t1 = $t2"},
    ...
  },
  "control": {
    "0": {"RegDst": 1, "RegWrite": 1, "ALUOp": 2, "ALUSrc": 0,
        "MemToReg": 0},
    ...
  },
  "alu": {
    "aluop_size": 2,
    "func_size": 6,
```

```
22      "control_size": 3,
23      "control": [
24        {"aluop": 0, "out": {"control": 2}},
25        {"aluop": 2, "func": 32, "out": {"control": 2}},
26        ...
27      ],
28      "operations": {
29        "2": "add",
30        ...
31      }
32    }
33  }
```

The various sections that compose the instruction set files are detailed in the following sections of this chapter.

3.1 types

This section defines the existing types of instructions and their fields. All instruction are 32 bits in size. An example of the definition of an instruction type is shown below:

```
1  "R": [{"id": "op", "size": 6}, {"id": "rs", "size": 5}, {"id":
       "rt", "size": 5}, {"id": "rd", "size": 5}, {"id": "shamt",
       "size": 5}, {"id": "func", "size": 6}]
```

Each instruction type is identified by its identifier (`R` in this example). The fields of an instruction are defined in a JSON array. Each individual field is a JSON object and defines the field's identifier and size in bits. The first field in this example is the `op` field with a size of 6 bits. The first field of the instructions is considered the *opcode* and must have the same size in all types.

3.2 instructions

This section defines the available instructions and how they are encoded. An example of the definition of an instruction is shown below:

```
1  "add":  {"type": "R", "args": ["reg", "reg", "reg"], "fields":
       {"op": 0, "rs": "#2", "rt": "#3", "rd": "#1", "shamt": 0, "
       func": 32}, "desc": "$t1 = $t2 + $t3"}
```

Each instruction is identified by its unique mnemonic (`add` in this example). An instruction is defined as a JSON object with several properties.

These are:

- **type**: The type of the instruction.
- **args**: An array with the types of each argument (may be omitted if the instruction has no arguments). The different types of arguments are:
  - **reg**: A register.
  - **int**: An integer value. Can also be a label (allowed because of the `la` pseudo-instruction).
  - **target**: A label in the code or direct instruction index for a jump.
  - **offset**: A label in the code or direct instruction offset for a branch.
  - **label**: A label in the code or data segment or direct address/index.
  - **data**: A label in the data segment or direct address plus an offset register for a load or store instruction (an argument like `label($t0)`). The user may omit the offset register.
- **fields**: A JSON object that defines the values of the fields (the fields defined in the instruction's type). The value of each field can either be a constant integer or come from an argument (specified as `"#1"`, `"#2"`, etc.). For values that come from a `data` argument it is necessary to specify if they come from the base address or from the offset register. This is done by appending `.base` or `.offset` to the reference to the argument (`#1`, `#2`, etc.).

  In this example: `op` has the constant value `0`, `rs` has the value from the 2nd argument, `rt` has the value from the 3rd argument, etc.
- **desc**: Optional. A string that should contain a short symbolic description of what the instruction does for the user to see.

3.3 pseudo

This section defines the available pseudo-instructions. An example of the definition of a pseudo-instruction is shown below:

```
1 "subi": {"args": ["reg", "reg", "int"], "to": ["li $1, #3", "
      sub #1, #2, $1"], "desc": "$t1 = $t2 - 23"}
```

Each pseudo-instruction is identified by its unique mnemonic (`subi` in this example). Each mnemonic can only identify either an instruction or a pseudo-instruction, not both. A pseudo-instruction is defined as a JSON object with several properties. These are:

- **args**: An array with the types of each argument (may be omitted if the pseudo-instruction has no arguments). These types are the same ones that are available for instructions, explained in the previous section.
- **to**: An array that lists the real instructions that the pseudo-instruction is converted to when assembled. The values of the arguments specified by the user can be referenced by using `#1` for the 1st argument, `#2` for the 2nd, and so on.
- **desc**: Optional. A string that should contain a short symbolic description of what the pseudo-instruction does for the user to see.

3.4 control

This section defines how the control unit works. More specifically, it defines the values of the outputs (control signals) for each possible value at the input (the *opcode*). An example of the definition of the values of the control signals for one *opcode* is shown below:

```
"0": {"RegDst": 1, "RegWrite": 1, "ALUOp": 2, "ALUSrc": 0, "
    MemToReg": 0}
```

The values of the control signals for a specified *opcode* are defined as a JSON object. The values are in decimal format and the sizes (in bits) of the control signals are determined automatically. Control signals that have the value `0` can be omitted.

3.5 alu

This section defines how the ALU and ALU control work. A partial example of this section is shown below:

```
"alu": {
  "aluop_size": 2,
  "func_size": 6,
  "control_size": 3,
  "control": [
    {"aluop": 0, "out": {"control": 2}},
    {"aluop": 2, "func": 32, "out": {"control": 2}},
    ...
  ],
  "operations": {
    "2": "add",
    ...
  }
```

```
14 }
```

The `aluop_size`, `func_size` and `control_size` properties define the sizes (in bits) of the `ALUOp` and `func` inputs of the ALU control and of the control input of the ALU, respectively.

The `control` subsection defines how the ALU control works (in a JSON array). Each element in the array defines (as a JSON object) the values of the outputs for the specified values of the inputs. The `aluop` and `func` properties correspond to the values of the inputs for the correspondence. If the value of the `func` input is not relevant for the correspondence (*don't care*, can be any value), the property should be omitted. The `out` property defines the values of the outputs for the correspondence. The values are specified as a JSON object and the sizes (in bits) of the outputs are determined automatically. All values are in decimal format.

The `operations` subsection defines (as a JSON object) the correspondence between the values of the control input of the ALU and the arithmetic operation it performs. The available operations are: **add**, **sub**, **and**, **or**, **slt**, **xor**, **sll**, **srl**, **sra**, **nor**, **mult**, **div**, **mfhi**, **mflo**. Note that the last four operations require an "extended" ALU in the CPU, instead of a "normal" ALU.