EXE+ALU+BRANCH DRAC specification version v0.1

Victor Soria Pardos

February 4, 2020

CONTENTS

1 General purpose of the module

The exe_stage module is the top view of the functional units of DRAC. Inside the exe_stage, ALU, DIV, MUL and BRANCH modules are instantiated. Exe_stage is also in charge of managing exceptions&interruptions, stall signals, computing branch miss-prediction, bypasses and data operands. Exe_stage receives:

- Csr interrupt signals from the datapath to manage interruptions
- Current instruction in execution stage to perform the execution phase
- Previous instruction result, to bypass the result to the dependent data operands
- Dcache interface response, with stall signal and load result

The Arithmetic Logic Unit (ALU) module is in charge of executing arithmetic (additions and subtractions), logic (OR, XOR, AND) and shift operations (Logical rigth and left, and arithmetic left). Operations pass from the exe_stage module to ALU and then results return to exe_stage module. ALU receives two data operands of 64 bits and an *enum* type of 7 bits containing the type of instruction. And returns a 64 bit signal containing the result of the operation.

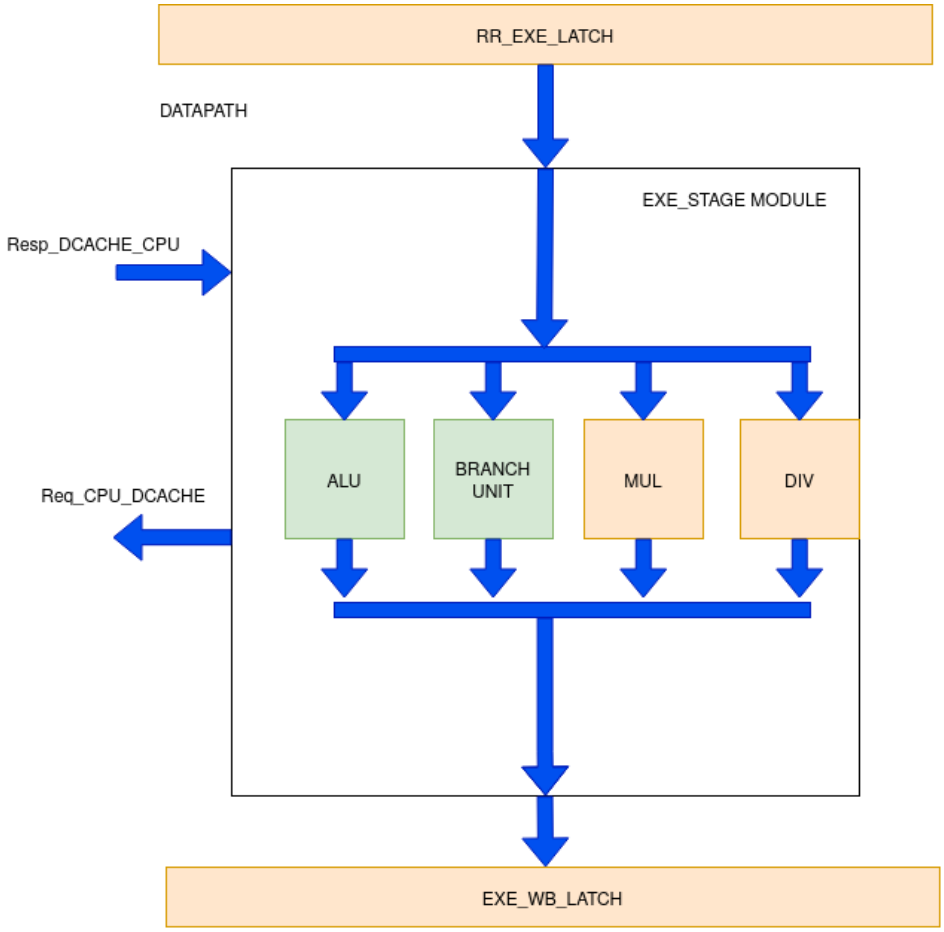
The Branch Unit (BU) module is in charge of computing if a branch is taken or not and the target to jump in. Operations pass from the exe_stage module to BU and then results return to exe_stage module. BU receives three data operands of 64 bits, an *enum* type of 7 bits containing the type of instruction and the PC of the instruction. And returns, if the branch was taken, what was the target of the branch.

2 Design placement

Exe_stage module should be instantiated on the datapath module, between Read Registers and Write Back stages. All input and output signals of exe_stage are connected to the datapath module.

ALU module is meant to be placed inside exe_stage module. There should be only one instance per core. An all signals of the module are connected inside exe_stage module.

BU module is meant to be placed inside exe_stage module. There should be only one instance per core. An all signals of the module are connected inside exe_stage module.



3 Parameters

All parameters, enums and types used in this module are defined in drac_pkg or risc_pkg.

4 Interface

In this section we describe the interface signals of each module for the different components connected to it.

4.1 Execution Stage Interfaces

4.1.1 Input Interface

Signal name	Width or Struct	Input	Description
clk_i	1	exe_stage <- datapath	Clock for the module
rstn_i	1	exe_stage <- datapath	Reset for the module. Asynchronous, acrive low reset
kill_i	1	exe_stage <- datapath	Kill signal, to abort current instruction execution
csr_interrupt_i	1	exe_stage <- datapath	Interruption detected on the CSR
csr_interrupt_cause_i	64	exe_stage <- datapath	Cause of the interruption detected by the CSR
from_rr_i	rr_exe_instr_t	exe_stage <- datapath	Current instruction in execution stage
from_wb_i	wb_exe_instr_t	exe_stage <- datapath	Bypass information from writeback
io_base_addr_i	40	exe_stage <- datapath	Base pointer of the input/output address space
resp_dcache_cpu_i	resp_dcache_cpu_t	exe_stage <- dcache_interface	Response from dcache interface

4.1.2 OUTPUT INTERFACE

Signal name	Width or Struct	Output	Description
to_wb_o	exe_wb_instr_t	exe_stage -> datapath	Output instruction to datapath with writeback format
stall_o	1	exe_stage -> datapath	Execution unit needs to stop the pipeline (from fetch to execution)
correct_branch_pred_o	1	exe_stage -> control_unit	Branch was predicted incorrectly
exe_if_branch_pred_o	1	exe_stage -> datapath	Struct that contains all information about the branch being executed in execution stage. Enum type exe_if_branch_pred_t
req_cpu_dcache_o	req_cpu_dcache_t	exe_stage -> dcache_interface	Request to dcache interface

4.2 ALU INTERFACES

4.2.1 INPUT INTERFACE

Signal name	Width	Input	Description
data_rs1_i	64	alu <- exe_stage	Data operand number 1
data_rs2_i	64	alu <- exe_stage	Data operand number 2
instr_type_i	7	alu <- exe_stage	Enum containing the type of instruction

4.2.2 OUTPUT INTERFACE

Signal name	Width	Output	Description
result_o	64	alu -> exe_stage	Result of the alu operation

4.3 Branch Unit Interfaces

4.3.1 Input Interface

Signal name	Width	Input	Description
instr_type_i	7	branch_unit <- exe_stage	Enum containing the type of instruction
pc_i	64	branch_unit <- exe_stage	Program Counter of current instruction in Execution Stage
data_rs1_i	64	branch_unit <- exe_stage	Data operand number 1
data_rs2_i	64	branch_unit <- exe_stage	Data operand number 2
imm_i	64	branch_unit <- exe_stage	Inmmediate operand

4.3.2 Output Interface

Signal name	Width or Struct	Output	Description
taken_o	branch_pred_decision_t	branch_unit -> exe_stage	Branch taken or not
result_o	64	branch_unit -> exe_stage	Branch target program counter

5 Reset behavior

There is no reset behaviour because none of these modules have sequential logic. However, inside exe_stage there are other modules that do have clock and reset. Therefore, exe_stage propagates these signals to these modules.

6 What could not happen

6.1 What could not happen in exe_stage module

In case that **from_rr_i.instr.ex.valid** is equal to one, the following conditions are always true:

- to_wb_o.ex.valid == 1
- to_wb_o.ex.origin == from_rr_i.instr.ex.origin
- to_wb_o.ex.cause == from_rr_i.instr.ex.cause

6.2 What could not happen in ALU module

In case that instr_type is one of:

- ADD
- ADDW

- SUB
- SUBW
- SLL
- SLLW
- SLT
- SLTU
- XOR
- SRL
- SRLW
- SRA
- SRAW
- OR
- AND

Signals data_rs1_i and data_rs2_i must have known values. This means that signals must be either 0 or 1, but not meta-stable values. And in case instr_type is none of the above result_o should always be 0.

6.3 What could not happen in Branch Unit module

In case that **instr_type_i** is one of:

- JAL
- JALR
- BEQ
- BNE
- BLT
- BGE
- BLTU
- BGEU

Signals pc_i, imm_i, data_rs1_i and data_rs2_i must have known values. This means that signals must be either 0 or 1, but not meta-stable values. And in case **instr_type** is none of the above **taken_o** should always be *PRED_NOT_TAKEN*.

In case that **instr_type_i** is JAL, **taken_o** is always *PRED_NOT_TAKEN*, because the jump was done at decode stage.

In case that **instr_type_i** is JALR, **taken_o** is always *PRED_TAKEN*.

In case that **taken_o** is *PRED_NOT_TAKEN*, **result_o** is always **pc_i** plus *0x4*.

The two lower bits of **result_o** and **link_pc_o** must be always 0.

7 BEHAVIOR

7.1 DESCRIPTION OF EXE_STAGE MODULE

In this section we describe what is the behaviour of exe_stage when an instruction comes from the datapath at the rising edge of the cycle. All the information is sent inside *from_rr_i* signal.

First step is resolve the bypasses from write back stage. If the instruction coming from write-back is valid, and the field *from_rr_i.intr.rs1* equals *from_wb_i.rd* then *Data operand 1*. Similar process is done for *Data operand 2*.

In case struct fields *from_rr_i.intr.use_pc* or *from_rr_i.intr.use_imm* are set to 1, *Data operand 1* will be *from_rr_i.intr.pc* and *Data operand 2* will be *from_rr_i.instr.result*, respectively.

Then the instruction and data operands are feed to all the functional units (Branch Unit, ALU, DIV Unit and MUL Unit). There is a special case in which if *from_rr_i.instr.unit* is equal to *UNIT_MEM*, and *from_rr_i.instr.valid* is set to 1, then *req_cpu_dcache_o* is filled the necessary data.

When the functional units finish the requested operation, *to_wb_o.result* is filled with the *result_o* signal of the corresponding functional unit of the instruction. The rest of the signals in *to_wb_o* struct are filled with the equivalent signals from the *from_rr_i* struct.

The only signals in *to_wb_o* that are not directly wired from *from_rr_i* are:

- *to_wb_o.result* mentioned earlier.
- *to_wb_o.csr_addr* is the lower 12 bits of *from_rr_i.result*, that contains the immediate value.
- *to_wb_o.ex.valid* contains if the current instruction is rising an exception. It must be set to 1 if there has been an exception in previous stages (*from_rr_i.instr.ex.valid* == 1). It must set to 1 if csr_interrupt_i is set (CSR has received an interruption), or there has been an exception on dcache (Load or store Misaligned, load or store access fault), or the branch is jumping to a misaligned address.
- *to_wb_o.ex.cause* contains what has been the cause of the exception or interruption.
- *to_wb_o.ex.origin* contains the program counter or address that caused the exception, depending on the cause.
- *to_wb_o.branch_taken* is set to 1 if the current branch is a jump or branch and is jumping to the target address.
- *to_wb_o.result_pc* contains what should be the program counter of the next instruction.

The struct *exe_if_branch_pred_o* contains all the information about branch prediction at execution stage that must be forwarded to fetch stage. It contains the program counter of the current instruction at execution stage, what is the program counter target computed by the branch, the decision if the branch is taken or not and if the instruction at execution stage is a real branch.

stall_o signal warns control unit that the execution stages needs at least one more cycle to complete the current instruction. It activates when *stall_o* signal of DIV, MUL or a memory access is set to 1, and the functional unit needed by the instruction matches the module setting the stall.

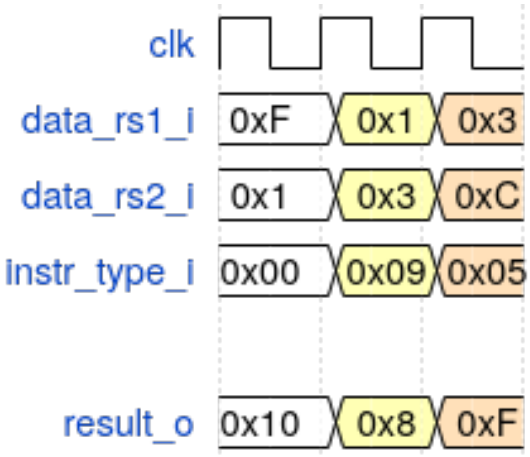
7.2 DESCRIPTION OF ALU MODULE

In this section we describe what is the behaviour of ALU module for each of the possible inputs.

- ADD: *result_o* contains the addition of *data_rs1_i* plus *data_rs2_i* using the standard two's complement notation.
- ADDW: *result_o* contains the addition of *data_rs1_i* plus *data_rs2_i* using the standard two's complement notation. *result_o* bits from 63 to 32 must be the sign extension of bit 31.
- SUB: *result_o* contains the subtraction of *data_rs1_i* minus *data_rs2_i* using the standard two's complement notation.
- SUBW: *result_o* contains the subtraction of *data_rs1_i* minus *data_rs2_i* using the standard two's complement notation. *result_o* bits from 63 to 32 must be the sign extension of bit 31.
- SLL: *result_o* contains the logical left shift of *data_rs1_i* by the six lower bits of *data_rs2_i*.
- SLLW: *result_o* contains the logical left shift of *data_rs1_i* by the five lower bits of *data_rs2_i*. *result_o* bits from 63 to 32 must be the sign extension of bit 31.
- SLT: *result_o* contains 1 if *data_rs1_i* < *data_rs2_i*, 0 otherwise. < operator uses the standard two's complement notation.
- SLTU: *result_o* contains 1 if *data_rs1_i* < *data_rs2_i*, 0 otherwise. < operator uses the standard unsigned notation.
- XOR: *result_o* contains the bit-wise XOR operation bit-wise *data_rs1_i* and *data_rs2_i*.
- SRL: *result_o* contains the logical right shift of *data_rs1_i* by the six lower bits of *data_rs2_i*.
- SRLW: *result_o* contains the logical right shift of *data_rs1_i* by the five lower bits of *data_rs2_i*. *result_o* bits from 63 to 32 must be the sign extension of bit 31.
- SRA: *result_o* contains the arithmetic right shift of *data_rs1_i* by the six lower bits of *data_rs2_i*.
- SRAW: *result_o* contains the arithmetic right shift of *data_rs1_i* by the five lower bits of *data_rs2_i*. *result_o* bits from 63 to 32 must be the sign extension of bit 31.
- OR: *result_o* contains the bit-wise OR operation bit-wise *data_rs1_i* and *data_rs2_i*.
- AND: *result_o* contains the bit-wise AND operation bit-wise *data_rs1_i* and *data_rs2_i*.

7.2.1 Examples

In the following example we can see three consecutive operations in the ALU module. In first cycle the ALU performs an ADD operation, in the second cycle does a SLL, and in the third cycle ALU computes a bit-wise OR. ALU returns always the result on the same cycle



7.3 Description of Branch Unit module

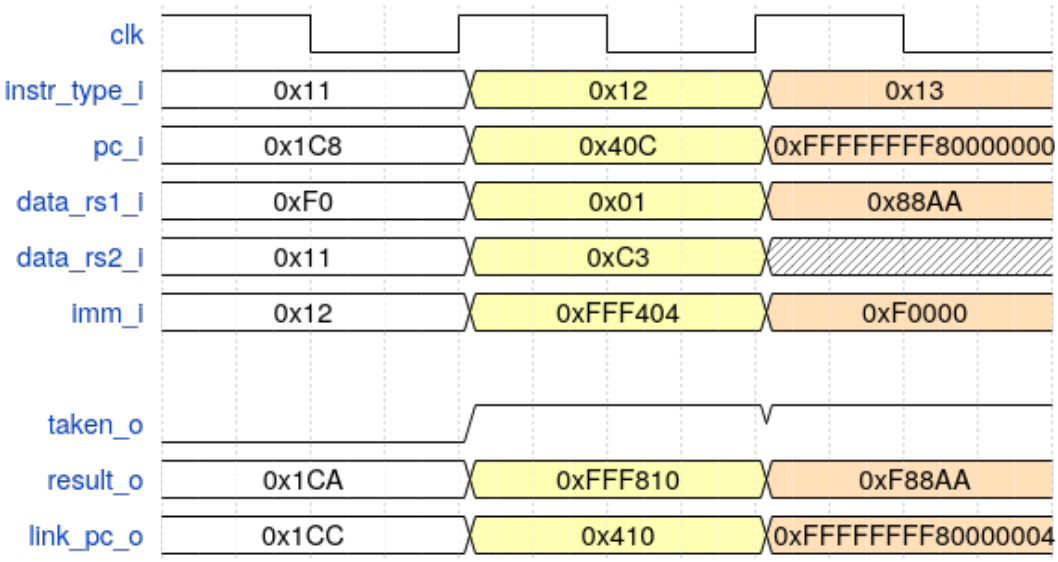
In this section we describe what is the behaviour of BU module for each of the possible inputs.

- JAL: *link_pc_o* contains the addition of *pc_i* plus *0x4*, *taken_o* is **always** equal to *PRED_NOT_TAKEN*.
- JALR: *link_pc_o* and *result_o* contain the addition of *pc_i* plus *0x4*, *taken_o* is **always** equal to *PRED_TAKEN*.
- BEQ: *taken_o* is equal to *PRED_TAKEN* if *data_rs1_i* is **equal** to *data_rs2_i*. Signal *result_o* is equal to *pc_i* plus *imm_i* if *taken_o* is equal to *PRED_TAKEN*, otherwise *result_o* is equal to *pc_i* plus *0x4*.
- BNE: *taken_o* is equal to *PRED_TAKEN* if *data_rs1_i* is **not** equal to *data_rs2_i*. Signal *result_o* is equal to *pc_i* plus *imm_i* if *taken_o* is equal to *PRED_TAKEN*, otherwise *result_o* is equal to *pc_i* plus *0x4*.
- BLT: *taken_o* is equal to *PRED_TAKEN* if *data_rs1_i* is **lower** than *data_rs2_i*. Lower refers to signed comparison in twoâĂŹs complement notation . Signal *result_o* is equal to *pc_i* plus *imm_i* if *taken_o* is equal to *PRED_TAKEN*, otherwise *result_o* is equal to *pc_i* plus *0x4*.
- BGE: *taken_o* is equal to *PRED_TAKEN* if *data_rs1_i* is **not lower** than *data_rs2_i*. Lower refers to signed comparison in twoâĂŹs complement notation . Signal *result_o* is equal to *pc_i* plus *imm_i* if *taken_o* is equal to *PRED_TAKEN*, otherwise *result_o* is equal to *pc_i* plus *0x4*.

- BLTU: *taken_o* is equal to *PRED_TAKEN* if *data_rs1_i* is **lower** than *data_rs2_i*. Lower refers to **unsigned representation**. Signal *result_o* is equal to *pc_i* plus *imm_i* if *taken_o* is equal to *PRED_TAKEN*, otherwise *result_o* is equal to *pc_i* plus *0x4*.
- BGEU: *taken_o* is equal to *PRED_TAKEN* if *data_rs1_i* is **not lower** than *data_rs2_i*. Lower refers to **unsigned representation**. Signal *result_o* is equal to *pc_i* plus *imm_i* if *taken_o* is equal to *PRED_TAKEN*, otherwise *result_o* is equal to *pc_i* plus *0x4*.

7.3.1 Examples

In the following example we can see three consecutive operations in the Branch Unit module. In first cycle the an BEQ operation, in the second cycle does a BNE, and in the third cycle JALR. Branch Unit returns always the result on the same cycle



8 Special cases, corner cases

8.1 EXE_STAGE

-
- Instructions not supported by the core (Floating points,)

8.2 ALU

- ADD, SUB and SLL overflows
- Instructions not supported by the ALU

8.3 BRANCH UNIT

- When *pc_i* plus *imm_i* have the two lower bits set to 1
- Instructions not supported by the Branch Unit
- When *data_rs1_i* plus *imm_i* have the two lower bits set to 1