Project #4 Report

2020-1xxxx

(Updated 2022. 12. 21.)

Introduction

The goal of this project was to understand how a pipelined processor works by extending the existing 5-stage pipelined RISC-V simulator **`snurisc5`** to support new instructions and a branch prediction scheme using the BTB(Branch Target Buffer).

Part 1: Supporting **`push`** and **`pop`** instructions

The operation of the **`push`** and **`pop`** instructions are defined as follows:

```
push rs2:  R[sp] <- R[sp] - 4
           M[R[sp]] <- R[rs2]
pop  rd :  R[rd] <- M[R[sp]]
           R[sp] <- R[sp] + 4
```

The **`push`** instruction behaves as **`sw rs2,-4(sp)`** in the EX, MM stage and as **`addi sp,sp,-4`** in the WB stage. Thus, we decoded the **`push`** instruction as a variant of **`sw`** instruction with *rs1* and *rd* as *sp* and the *imm* as -4, along with the write-back of the ALU output as with the **`addi`** instruction.

```
SW     : [ Y, BR_N, OP1_RS1, OP2_IMS, OEN_1, OEN_1, ALU_ADD, WB_X  , REN_0, MEN_1, M_XWR, MT_W, ],
ADDI   : [ Y, BR_N, OP1_RS1, OP2_IMI, OEN_1, OEN_0, ALU_ADD, WB_ALU, REN_1, MEN_0, M_X  , MT_X, ],

PUSH   : [ Y, BR_N, OP1_SP , OP2_IN4, OEN_1, OEN_1, ALU_ADD, WB_ALU, REN_1, MEN_1, M_XWR, MT_W, ],
```

The **`pop`** instruction behaves a bit more trickily, since it uses the original value of the *sp* as the address, instead of the ALU output. Moreover, it writes back to two registers. Therefore, the followings have to be considered:

1. The original value of the *sp* register should be sent to, or recovered in the MM stage.
2. Both the ALU output and the loaded value should be sent to the WB stage.
3. The write-back logic should be able to handle two simultaneous register writes.

#3 was already implemented in the skeleton code. For #1, we could reuse the separate datapath for store data by carefully adding some MUXs. For #2, we have to add new datapath to send the secondary write-back data from the EX/MM stage to the WB stage. To avoid a drastic modification of the core, we limited the *rd2* to *sp* for the **`pop`** instruction.

Handling the new Data Hazards

Data hazards occur when an instruction depends on the result of a preceding instruction which has not been written back yet. (There could be more cases for more complex processors, but what we need to consider is only the RAW data hazard for the **snurisc5**.)

The **push** and **pop** instructions also can be involved in data hazards. Some cases could be solved by the existing logic while other cases could not. Here are some cases which don't need additional work:

1. The *rs2* of a **push** instruction in the ID stage is the *rd* of an instruction in EX/MM/WB stage.
2. The *rd* of a **pop** instruction in EX/MM/WB stage is either the *rs1* or *rs2* of an instruction in the ID stage. (load-use hazard)

For these cases, the original dependency check logic would forward the data if the desired data exist in the pipeline or stall the pipeline until the hazard can be resolved by forwarding. However, those cases need some modification on the logic:

3. The *sp* register is the *rd* of an instruction in EX/MM/WB stage when a **push** or **pop** instruction in the ID stage. (implicit dependency on *sp*)
4. The *sp* register is either the *rs1* or *rs2* of an instruction in the ID stage when a **push** instruction in EX/MM/WB stage. (implicit dependency)
5. The *sp* register is either the *rs1* or *rs2* of an instruction in the ID stage when a **pop** instruction in EX/MM/WB stage. (implicit dependency)

We solved #3 by explicitly decoding the implicitly encoded *sp* as *rs1* for the **push** or **pop** instruction before the dependency check. We solved #4 also by explicitly decoding *sp* as *rd* for the **push** instruction. This approach can reduce additional cost by utilizing the existing logic. However, #5 requires additional forwarding logics. We added a control signal which is indicating the *sp* write-back of a **pop** instruction and implemented additional dependency checks with the signal. Here's an example of *rs1*:

```
Forward_Rs1     =   FWD_EX      if (EX.reg_rd == Pipe.ID.rs1) and rs1_oen and   \
                                   (EX.reg_rd != 0) and EX.reg_c_rf_wen else    \
                    FWD_EX      if (Pipe.ID.rs1 == 2) and rs1_oen and           \ # +added
                                   EX.reg_rd2                           else    \  # +added
                    FWD_MM      if (MM.reg_rd == Pipe.ID.rs1) and rs1_oen and   \
                                   (MM.reg_rd != 0) and Pipe.MM.c_rf_wen else   \
                    FWD_MM2     if (Pipe.ID.rs1 == 2) and rs1_oen and           \ # +added
                                   MM.reg_rd2 and Pipe.MM.c_rf_wen      else    \ # +added
                    FWD_WB      if (WB.reg_rd == Pipe.ID.rs1) and rs1_oen and   \
                                   (WB.reg_rd != 0) and WB.reg_c_rf_wen else    \
                    FWD_WB2     if (Pipe.ID.rs1 == 2) and rs1_oen and           \ # +added
                                   WB.reg_rd2                           else    \ # +added
                    FWD_NONE
```

(Forwarding logic for the *rs2* was implemented with the same way.)

Modification in the datapath

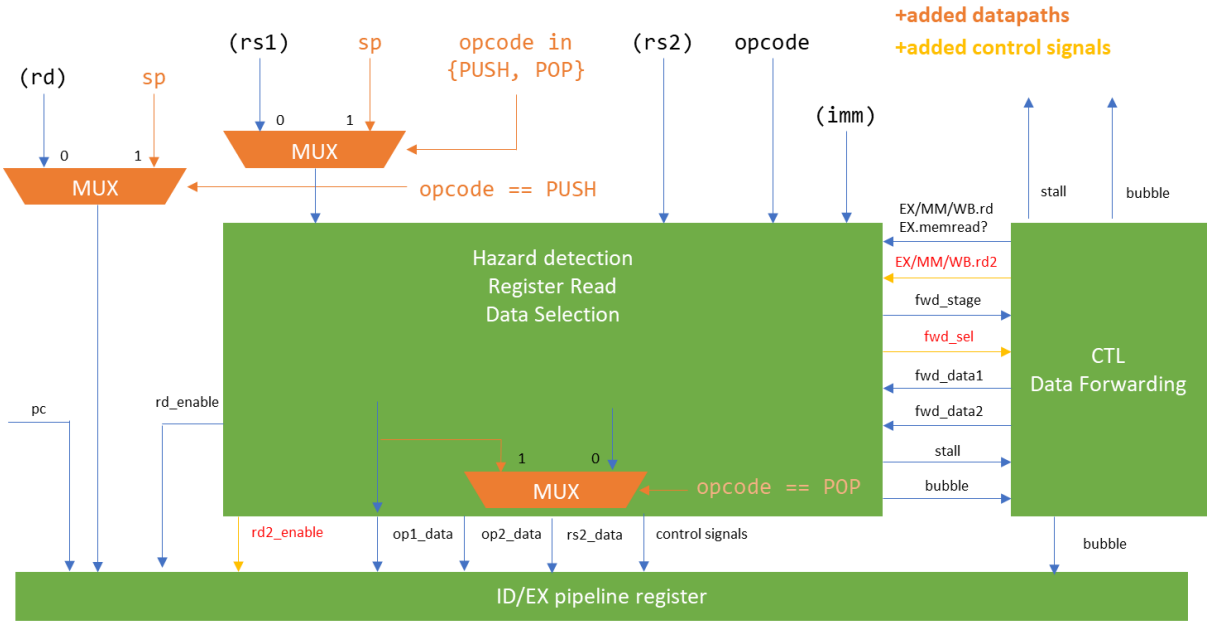

Figure 1: overview of the changes in the datapath: ID stage

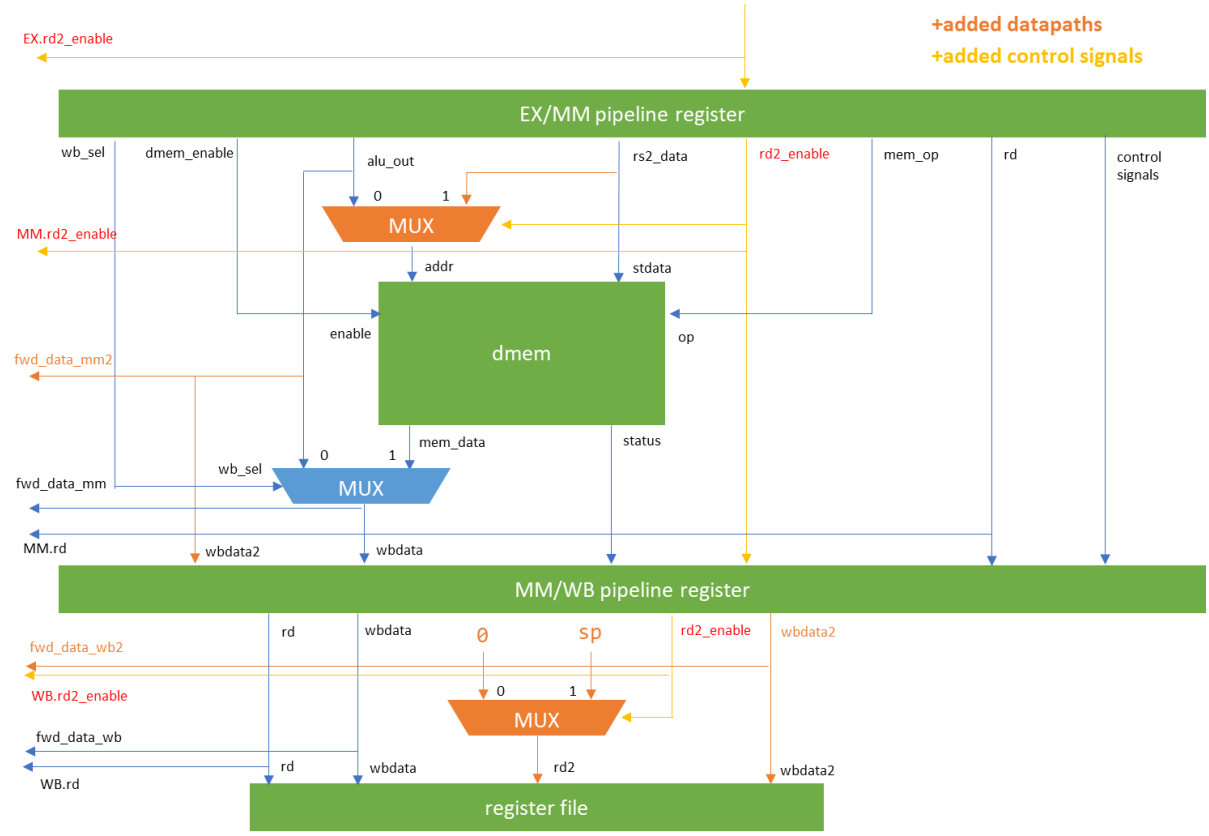

Figure 2: overview of the changes in the datapath: EX ... WB stage

We modified the datapath to support the `push` and `pop` instructions and deal with the new data hazards occur due to those new instructions. Here are the changes in the datapath:

1. (Figure 1) Two MUXs were added to select *rd* and *rs1* between the decoded value and *sp*. The first MUX selects *sp* as *rd* for the `push` instruction. The second MUX selects *sp* as *rs1* for the `push` and `pop` instructions. This change was made for utilizing the exist ALU for new *sp* address calculation and reusing the existing hazard detection logic. (Case #3, #4)
2. (Figure 1,2) Two MUXs were added. The first selects the *sp* value as `rs2_data` for the `pop` instruction and the second selects the original *sp* value passed by `rs2_data` as the source memory address for the `pop` instruction. (Consideration #1)
3. (Figure 2) A datapath was added to pass the ALU output to the WB stage for the `pop` instruction. This data is written back to the *sp* register at the WB stage when the `rd2_enable` signal is enabled. (Consideration #2)
4. (Figure 2) A MUX was added to selects *sp* as *rd2* when *rd2* is enabled. The `rd2_enable` signal is only enabled for the `pop` instruction. (Consideration #3)
5. (Figure 1,2) The `rd2_enable` signals of EX/MM/WB stage was wired to the dependency check logic. (Case #5) Corresponding datapaths were added and wired from MM/WB stage to the data forwarding unit to resolve *sp*-dependency from the `pop` instruction.

Changes in the control signals

We added some control signals to support the `push` and `pop` instructions and implement additional data forwarding for solving the new data hazards. The followings are the changes in the control signals:

Tables below show the values of control lines for the operand MUXs that select operand in the ID stage, which is additionally generated by the decoder in our extended implementation.

Table C-1: Rs1 selection for hazard detection and operand read.

MUX control	Condition	Description
Rs1_sel = 0	`Opcode not in {PUSH, POP}`	Selects the decoded *rs1* value for *rs1*.
Rs1_sel = 1	`Opcode in {PUSH, POP}`	Selects the *sp* for *rs1*.

Table C-2: Rd selection for hazard detection and write-back.

MUX control	Condition	Description
Rd_sel = 0	`Opcode != PUSH`	Selects the decoded *rd* value for *rd*.
Rd_sel = 1	`Opcode == PUSH`	Selects the *sp* for *rd*.

Table C-3: Rs2 data selection for forwarding the store data or the original *sp* value.

MUX control	Condition	Description
Rs2_sel = 0	`Opcode != POP`	Selects the value of *rs2* for `rs2_data`.
Rs2_sel = 1	`Opcode == POP`	Selects the value of *sp* for `rs2_data`. (forwarding op1)

Table C-4: The `rd2_enable` signal. (= same as Rs2_sel)

`rd2_enable` signal	Condition	Description
`rd2_enable = 0`	`Opcode == POP`	Default value (when the instruction is not a **pop**).
`rd2_enable = 1`	`Opcode != POP`	Selects the `rs2_data` value over the ALU output for the memory address in the MM stage. Selects *sp* over *x0*(no write) as *rd2* for the second write-back port in the WB stage.

Tables below show the control values for the forwarding MUXs. Changes are highlighted in yellow. Conditions are checked sequentially from the top to the bottom; the top has the highest priority while the bottom has the lowest. (If forwarding is needed, the value is forwarded from the most recent one.)

Table S-1: Forward selection for op1(the first ALU operand).

MUX control	Source	Condition / Explanation
Fwd_op1 = FWD_EX	EX	`Rs1_Oen and ((EX.Rd_enable and (EX.Rd != 0) and (EX.Rd == ID.Rs1)) or (EX.Rd2_enable and (ID.Rs1 == 2)))` The first ALU operand is forwarded from the prior ALU result.
Fwd_op1 = FWD_MM	MM	`Rs1_Oen and MM.Rd_enable and (MM.Rd != 0) and (MM.Rd == ID.Rs1)` The first ALU operand is forwarded from data memory or an earlier ALU result.
Fwd_op1 = FWD_MM2	MM	`Rs1_Oen and MM.Rd2_enable and (ID.Rs1 == 2)` The first ALU operand is forwarded from an earlier ALU result. (for the **pop** instruction)
Fwd_op1 = FWD_WB	WB	`Rs1_Oen and WB.Rd_enable and (WB.Rd != 0) and (WB.Rd == ID.Rs1)` The first ALU operand is forwarded from the first write-back port.
Fwd_op1 = FWD_WB2	WB	`Rs1_Oen and WB.Rd2_enable and (ID.Rs1 == 2)` The first ALU operand is forwarded from the second write-back port. (for the **pop** instruction)
Fwd_op1 = FWD_NONE	ID	The first ALU operand comes from the register file or *pc*.

Table S-2: Forward selection for op2(the second ALU operand).

MUX control	Source	Condition / Explanation
Fwd_op2 = FWD_EX	EX	`(Op2_sel == Op2_Rs2) and ((EX.Rd_enable and (EX.Rd != 0) and (EX.Rd == ID.Rs2)) or (EX.Rd2_enable and (ID.Rs2 == 2)))` The second ALU operand is forwarded from the prior ALU result.
Fwd_op2 = FWD_MM	MM	`(Op2_sel == Op2_Rs2) and MM.Rd_enable and (MM.Rd != 0) and (MM.Rd == ID.Rs2)` The second ALU operand is forwarded from data memory or an earlier ALU result.
Fwd_op2 = FWD_MM2	MM	`(Op2_sel == Op2_Rs2) and MM.Rd2_enable and (ID.Rs2 == 2)` The second ALU operand is forwarded from an earlier ALU result. (for the `pop` instruction)
Fwd_op2 = FWD_WB	WB	`(Op2_sel == Op2_Rs2) and WB.Rd_enable and (WB.Rd != 0) and (WB.Rd == ID.Rs2)` The second ALU operand is forwarded from the first write-back port.
Fwd_op2 = FWD_WB2	WB	`(Op2_sel == Op2_Rs2) and WB.Rd2_enable and (ID.Rs2 == 2)` The second ALU operand is forwarded from the second write-back port. (for the `pop` instruction)
Fwd_op2 = FWD_NONE	ID	The second ALU operand comes from the register file or the decoded immediate.

The table below describes the value of Op2_sel control lines:

Table S-2a: Op2 selection

MUX control	Source	Description
Op2_sel = Op2_Rs2	Register	The second ALU operand comes from the register file.
Op2_sel = Op2_Imm@ @ in {I, S, B, U, J}	Decoded Immediate	The second ALU operand comes from the decoded @-type immediate value which is encoded in the opcode.
Op2_sel = Op2_Imm#4 # in {P(+), N(-)}	+4 / -4	The second ALU operand is +4(#=P) or -4(#=N).
Op2_sel = Op2_X	0	The second ALU operand is 0.

Table S-3: Forward selection for rs2 data(the `rs2_data` in Figure 1, 2).

MUX control	Source	Condition / Explanation
Fwd_rs2 = FWD_EX	EX	`Rs2_Oen and ((EX.Rd_enable and (EX.Rd != 0) and (EX.Rd == ID.Rs2)) or (EX.Rd2_enable and (ID.Rs2 == 2)))` The `rs2_data` is forwarded from the prior ALU result.
Fwd_rs2 = FWD_MM	MM	`Rs2_Oen and MM.Rd_enable and (MM.Rd != 0) and (MM.Rd == ID.Rs2)` The `rs2_data` is forwarded from data memory or an earlier ALU result.
Fwd_rs2 = FWD_MM2	MM	`Rs2_Oen and MM.Rd2_enable and (ID.Rs2 == 2)` The `rs2_data` is forwarded from an earlier ALU result. (for the **pop** instruction)
Fwd_rs2 = FWD_WB	WB	`Rs2_Oen and WB.Rd_enable and (WB.Rd != 0) and (WB.Rd == ID.Rs2)` The `rs2_data` is forwarded from the first write-back port.
Fwd_rs2 = FWD_WB2	WB	`Rs2_Oen and WB.Rd2_enable and (ID.Rs2 == 2)` The rs2_data is forwarded from the second write-back port. (for the **pop** instruction)
Fwd_rs2 = FWD_NONE	ID	The `rs2_data` comes from the register file.

Part 2: Branch prediction with Branch Target Buffer

The BTB to implement has a direct-mapped structure as below:

```
BTB structure:
     +---+---------+----------------------+
  0  | V | tag (T) |  target address (A)  |
     +---+---------+----------------------+
  1  |   |         |                      |
     +---+---------+----------------------+
     |   |   ...   |        ...           |
     +---+---------+----------------------+
N-1  |   |         |                      |
     +---+---------+----------------------+
```

Here are the specifications:

1. The BTB has the power-of-2 (i.e. $N=2^k$, $0 \le k \le 8$) entries.
2. The BTB has a direct-mapped organization, where `(pc >> 2) % N` is used as an index. We omit the last 2 bits in the `pc` because they will be always `0b00` (all the instruction addresses are aligned to the 4-byte boundary in `pyrisc`). The remaining bits of the pc (i.e. `pc >> (k + 2)`) are used as the tag bits. If the valid bit (`V`) is 1, it indicates a valid entry.
3. Because of the direct-mapped organization, two or more branch instructions can be mapped to the same entry in the BTB. If the corresponding entry is already occupied by another branch instruction at the time you want to add a new entry, it is simply overwritten with the information of the current branch instruction.
4. The BTB is checked in the IF stage.
5. The BTB is updated in the EX stage. You may need to add/remove an entry to/from the BTB in the EX stage when the prediction was wrong.
6. There can be a case when the fetched instruction is correct, even though the prediction was wrong (e.g. jumping to PC+4) • In such cases, even though it is inefficient, flush the instructions in the IF and ID stages.
7. It is very difficult to predict the target address of the `jalr` instruction as it depends on a register value. To make the problem simpler, the `jalr` instruction is handled in the same way as in `snurisc5`; the instructions next to the `jalr` instruction are fetched until we have the target address in the EX stage and then the two instructions being executed in the IF and ID stages are converted into BUBBLEs while the target address is forwarded to the next `pc` value immediately. On the other hand, the `jal` instruction can be treated in the same way as the other branch instructions.

#1, #2, #3 specify the organization of BTB itself while others specify the behavior of the pipeline.

Here are our implementations:

1. Just implemented the BTB with #1, #2, #3. Since #1, we could use `pc[31:k+2]` as the tag and `pc[k+1:2]` as the index. All the entries in the BTB are initialized with {V=0, Tag=0, Target=0}. Add operation just overwrites the corresponding entry with given tag and target address. Remove operation just sets the corresponding entry with {V=0, Tag=0, Target=0}.
2. By #4, #7, We implemented a logic in the IF stage which checks the instruction is whether Branch/JAL or not. A RISC-V Branch/JAL instruction can be identified with the `opcode := inst[6:0]`, since the opcode `0b1100011` is dedicated for Branch instructions and the opcode `0b1101111` is dedicated for JAL instructions. (See Figure 3 in the next page.) If the instruction has an opcode `0b1100011` or `0b1101111` and BTB is hit, the next *pc* value is updated with the target address from the BTB. Two datapaths were added to look up the BTB and retrieve the target address when BTB is hit.
3. By #5, we added datapaths from the EX stage to the BTB to add/remove BTB entries with the *pc* value and the target address.
4. By #6, we added a datapath from the IF stage to the EX stage through pipeline registers. Mistaken branch detection would be done by comparing the branch prediction result to the determined value instead of just comparing the next pc value to the calculated one. This approach costs less in terms of transistor count.

Finding out a mispredicted branch

In the original **`snurisc5`**, finding out a mispredicted branch is done by just comparing the determined execution flow to not-taken since it uses always-not-taken branch prediction scheme.

In our implementation, we just rewired the hard-wired not-taken with the prediction result (Implementation #4) and compare it to the determined one to find out a mispredicted branch.

Handling a mispredicted branch

A mispredicted branch is handled the same way as in **`snurisc5`**(wrongly fetched instructions are converted into BUBBLEs) but the *pcplus4* value was wired from the EX stage to the IF stage because a taken branch could be mispredicted one in our implementation. The handling process also involves the update(add/remove) of a BTB entry for Branch/JAL instructions in our implementation. (The **`jalr`** instruction is always treated as mistaken thus a mistaken jalr should not involve any changes on the BTB.)

Modification in the datapath

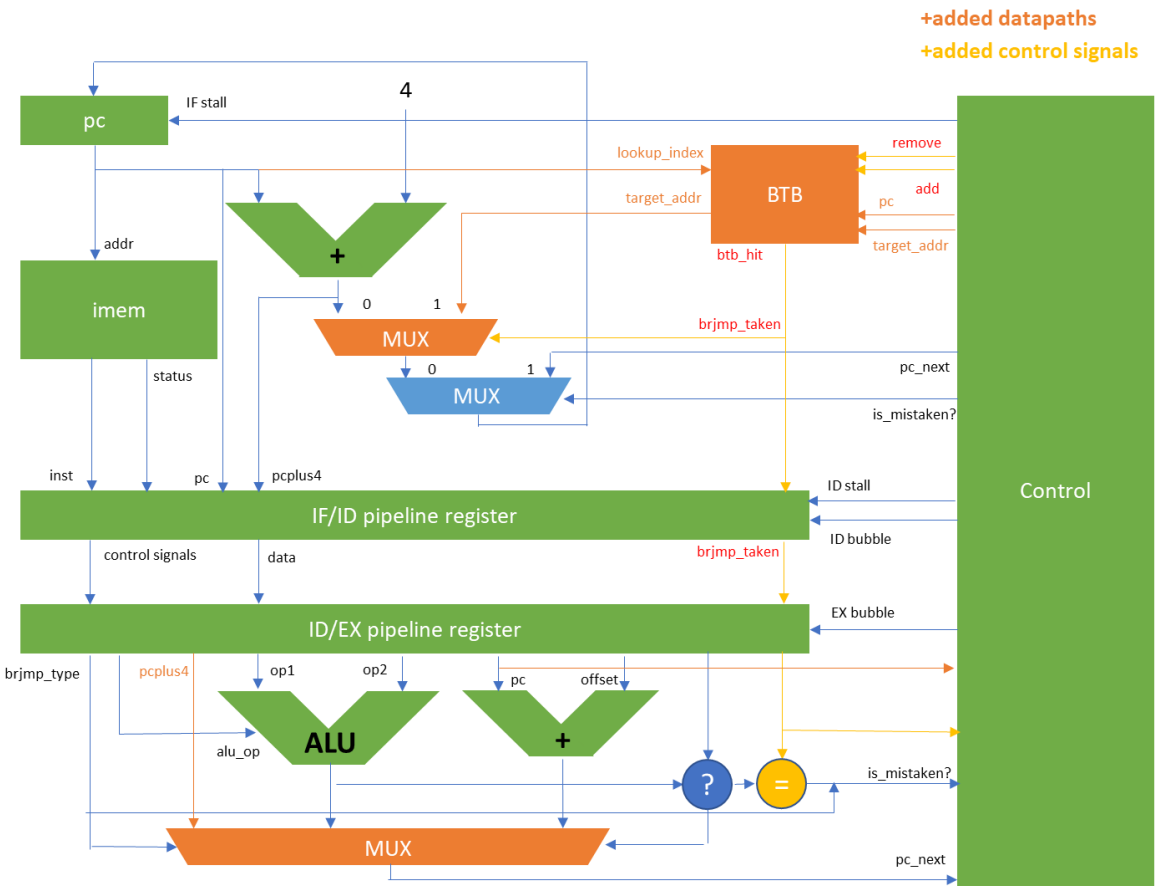

Figure 3: overview of the changes in the datapath. (some stages/datapaths are omitted)

We modified the datapath to implement the branch prediction scheme with BTB. Here are the changes in the datapath (see Figure 3):

1. A MUX and two datapaths were added in the IF stage to lookup the BTB and selects the target address over pc+4 as the next pc value if BTB is hit.
2. A signal path from the IF stage to the EX stage was added to keep track of the branch prediction result to determine whether the prediction result was right or wrong.
3. A comparator was added to compare the prediction result with the determined one.
4. A datapath was added to send pc+4 value in the EX stage to the IF stage.
5. Two datapaths with pc, target address were added to update the BTB when the prediction result was wrong.

Changes in the control signals

We added some control signals to support the branch prediction scheme with BTB. The followings are the changes in the control signals:

Table I-1: The next-pc selection signal.

MUX control	Condition	Description
B_Taken = 0	`not BTB_hit`	The next-pc value is pc+4.
B_Taken = 1	`BTB_hit`	The next-pc value is retrieved from the BTB.

Table I-2: The next pc MUX.

Mistaken?	BTB hit?	Target chosen	Description
B_wrong = 0	B_Taken = 0	--	The next pc is pc+4.
B_wrong = 0	B_Taken = 1	--	The next pc is retrieved from the BTB.
B_wrong = 1	--	PC_sel = PC_4	The next pc is pcplus4 value in the EX stage.
B_wrong = 1	--	PC_sel = BRJMP	The next pc is branch target calculated in the EX stage.
B_wrong = 1	--	PC_sel = JR	The next-pc is ALU output in the EX stage.

(Table I-2*a*) (Table I-1) (Table I-2*b*)

Table I-2*a*: The determined result in the EX stage.

MUX control	Condition	Description
B_wrong = 0	`(B_Taken == PC_sel) and not (B_Type == JR)`	The prediction result is same to the determined one and branch type is not JALR.
B_wrong = 1	`(B_Taken != PC_sel) or (B_Type == JR)`	The prediction result is different to the determined one or the branch type is JALR.

Table I-2*b*: The PC selection mux in the EX stage.

MUX control	Condition	Description
PC_sel = PC_4	`(B_Type = None) or (not Condition[B_Type]) and (B_Type != JALR)`	The branch/jump is not taken.
PC_sel = BRJMP	`(B_Type == JAL) or Condition[B_Type] and (B_Type != JALR)`	The branch/JAL is taken.
PC_sel = JR	`B_Type == JALR`	The JALR is taken.

Table I-3a: The 'add' enable signal to the BTB.

MUX control	Condition	Description
BTB_add = 0	`not B_wrong or (PC_sel != BRJMP)`	Do nothing (default).
BTB_add = 1	`B_wrong and (PC_sel == BRJMP)`	Overwrite the corresponding entry for the pc value with given target address.

Table I-3b: The 'remove' signal to the BTB.

MUX control	Condition	Description
BTB_remove = 0	`Not B_wrong or (PC_sel != PC_4)`	Do nothing (default).
BTB_remove = 1	`B_wrong and (PC_sel == PC_4)`	Remove the corresponding entry for given pc.