Verilog Implementation Report

Computer Organization Project 6

Jaewon Chung, 2015-11832
Due: 2019/05/13

1. Introduction

- I will implement a pipelined version of the CPU, possibly with data forwarding and one fairly complex branch predictor. With each version, I will compare its performance (in cycles) with the previous one.

2. Design

- This part is replaced by the handwritten design document attached at the back.

3. Implementation

- Baseline
  - Control signals are almost identical to those in my multicycle implementation, with a few additional ones to control pipeline stall and flush (IFIDWrite, IFFlush, IDEXWrite). Find their description annotated in the control.v file.
  - Pipeline registers and their content can be found in the second page of the design document.
  - Hazard Detector module: detects data hazards and sends out stall signals
  - Control module: producs control signals, and stalls/flushes the pipeline when data/control hazard is detected (control hazard is detected in cpu.v)
  - `num_inst = internal_num_inst - stall_penalty - jump_mispredict_penalty - branch_mispredict_penalty`
    internal_num_inst: incremented every cycle (assumes IPC = 1)
    others: incremented at clock negedge if the CPU is experiencing each of their corresponding event
    If many occur in the same cycle, the priority is branch_misprediction > stall > jump_misprediction.
    None of the three increment together in a single cycle.
- Baseline + Data Forwarding
  - A straightforward implementation of “Forwarding Paths (v1)” in p.22, Lecture 8.
  - Every reference to `RFRead1` and `RFRead2` is replaced to `Forwarded_Rs` and `Forwarded_Rt`. Data is forwarded from either EX, MEM, or WB.
  - Hazard Detector module: detects data hazards and sends out stall signals and data forwarding signals that control the MUX after the register file read port.

- Baseline + Data Forwarding + 2-bit Saturation Counter Branch Predictor
  - BTB only holds target addresses for branch and jump instructions.
  - 8 bit tag, 8 bit BTB index, 2 bit saturation counter that predicts 'taken' for counter values 2 and 3.
  - The counters are initialized to 01.
  - When a jump or branch instruction's target address is determined (in the ID or EX stage respectively), the results, including whether the branch was taken; where the target address was; and the address (PC) of that instruction, are fed back into the predictor. Then the predictor updates its counters and tables based on that information.
  - The predictor keeps a 2 bit local prediction history register that records the prediction (whether taken or not taken) of the last two instructions. This is used to determine whether the prediction was correct for branches resolved.
- Baseline + Data Forwarding + Alpha 21264 Style Tournament Branch Predictor
  - BTB and tag table is the same as the previous 2-bit saturation counter branch predictor.
  - Refer to the diagram I drew in the design documentation for a more detailed structure.
  - The three types of counters are all initialized to 01.
  - One element different from the design is the GHSR(Global History Shift Register). We have to keep ONLY the recent 12 branch/jump instructions' prediction result. Thus I figured there are two choices: the first is to just redefine the GHSR to record the recent 12 'instructions', and the second is to add a 'predecoding' stage that determines during the IF stage whether the fetched instruction is either a branch or a jump. I chose the second one.
  - The predictor keeps a 2 bit local prediction history register and a 2 bit (global vs local) choice history register. These are, as in the 2 bit saturation counter branch predictor, used to determine whether the prediction was correct. Determining this is crucial for updating the choice predictor's counter.

4. Discussion

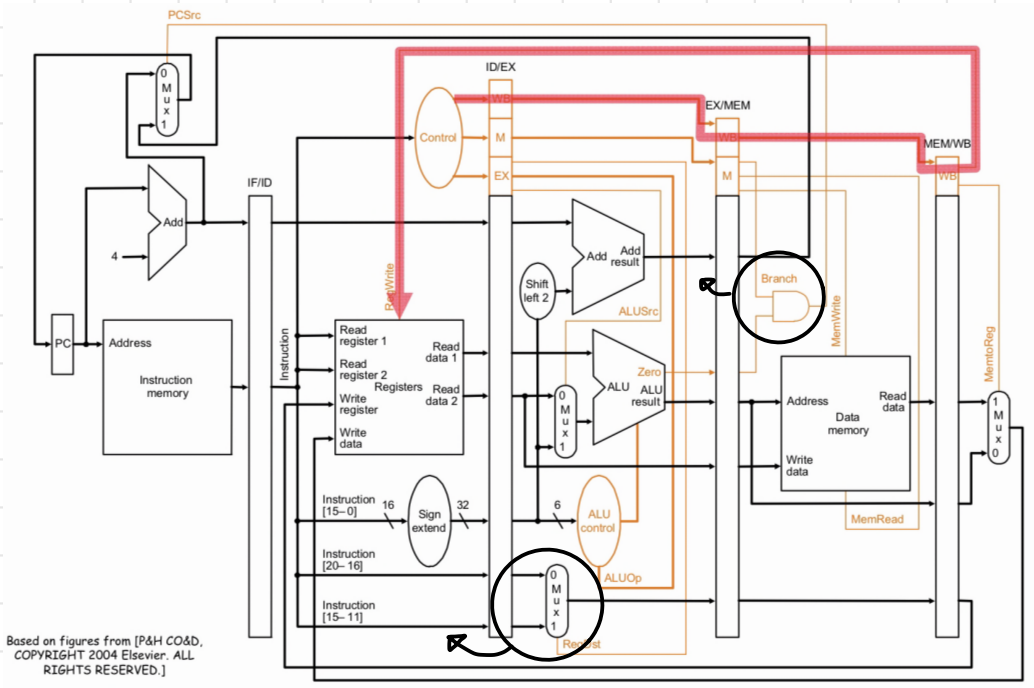
Below are decimal numbers.

# of cycles	Multi cycle	Pipeline (baseline)	Pipeline (data forwarding)	Pipeline (2-bit saturation)	Pipeline (tournament)
total	3488	2039	1254	1191	1166
stall penalty	N/A	789	4	4	4
jump misprediction	N/A	178	178	71	77
branch misprediction	N/A	88	88	132	100

5. Conclusion

- I believed I have achieved everything, and more of the goals specified in the introduction.

[Attachment]
1. Design document (baseline, data forwarding, and two branch predictors)



Pipelined CPU

<Baseline>

Control signal in blue.

IF

combinational : nextPC

= (IsBranch & BranchMisprediction) ? correct_branch :
(IsJump & JumpMisprediction) ? JumpAddress : Prediction;

posedge clk : if (PCWrite) PC <= nextPC.

negedge clk : if (IFIDWrite) fetch instruction from memory

ID

combinational : RFRead1, RFRead2 are read from RF.
Immediate Field is sign-extended
JumpAddress = JumpType ? RFRead1 : target;
output_port is serviced after hazard is resolved.
RF Write Address is determined by RegDst.

posedge clk : Write data (RegSrc, from WB) is written to the
Write address (RegDst, from WB)

EX

combinational : ALU input is determined by ALUSrcA, ALUSrcB
ALUResult (controlled by ALUOp) is calculated
(includes data calculations and PC+1 for JAL, JRL)

MEM

negedge clk : if (DataMemRead or DataMemWrite) Access data memory

WB

combinational : RF Write data is determined by RegSrc

Control module : IsBranch, IsJump, DataMemRead, DataMemWrite, Halt
produces at RegWrite, PCWrite, IFIDWrite, IFFlush, IDEXWrite
negedge clk

ALUSrcA : RFRead1(0), PC(1)
ALUSrcB : RFRead2(0), Sign-extended Imm(1)
RegSrc : ALUResult(0), MDR(1)
RegDst : RtAddress(00), RdAddress(01), 2(10)
OpenPort : asserted when WWD.
ALUOp : refer to single cycle implementation.
JumpType : target(0), Rs(1)

IF/ID register : Instruction, PC, nextPC

ID/EX register : IsBranch, ALUSrcB, ALUSrcA
DataMemRead, DataMemWrite, RegSrc, RegDst
ALUOp, RegWrite, Halt
RFRead1, RFRead2, Sign-extended Imm,
RFWriteAddress, nextPC, PC

EX/MEM register : DataMemRead, DataMemWrite, RegSrc, RegWrite
RFRead2, PC, ALUResult, RFWriteAddress

MEM/WB register : RegSrc, RegWrite
RFRead2, MemData, RFWriteAddress,
ALUResult

Stall due to data hazard. : Hazard Detector module !

Instruction	Opcode	Function Code	Format
ADD	15	0	R
SUB	15	1	R
AND	15	2	R
ORR	15	3	R
NOT	15	4	R
TCP	15	5	R
SHL	15	6	R
SHR	15	7	R
ADI	4	-	I
ORI	5	-	I
LHI	6	-	I
WWD	15	28	R
LWD	7	-	I
SWD	8	-	I
BNE	0	-	I
BEQ	1	-	I
BGZ	2	-	I
BLZ	3	-	I
JMP	9	-	J
JAL	10	-	J
JPR	15	25	R
JRL	15	26	R
HLT	15	29	R

needs Rs and Rt : group 1

needs Rs : group 2

Stall : PCWrite = 0. IFIDWrite = 0.
IFFlush = 0. IDEXWrite = 0

Rs = Inst[11:10], Rt = Inst[9:8]

$Dest_{stage}$ = RFWriteAddress of stage

$RegWrite_{stage}$ = RegWrite of stage

Stall Condition :

if (group 1 or group 2) // Rs

$Dest_{ID/EX}$ == Rs and $RegWrite_{ID/EX}$

$Dest_{EX/MEM}$ == Rs and $RegWrite_{EX/MEM}$

$Dest_{MEM/WB}$ == Rs and $RegWrite_{MEM/WB}$

if (group 1) // Rt

$Dest_{ID/EX}$ == Rt and $RegWrite_{ID/EX}$

$Dest_{EX/MEM}$ == Rt and $RegWrite_{EX/MEM}$

$Dest_{MEM/WB}$ == Rt and $RegWrite_{MEM/WB}$

Jump Misprediction detected at ID

⇒ JumpMisprediction = IsJump & (JumpAddress != nextPC)

- JumpMisprediction → to control module
- IsJump ← control module
- JumpAddress ← calcalated at ID
- nextPC ← IF/ID register

⇒ control module produces

PCWrite = 1

IFIDWrite = 0

IFFlush = 1

IDEXWrite = 1

Branch Misprediction detected at EX

⇒ Branch Misprediction

= IsBranch & [(BranchTaken & (nextPC != ALUResult)) |

(! BranchTaken & (nextPC != PC + 1))]

- IsBranch ← ID/Ex register
- BranchTaken ← ALU output
- nextPC ← ID/Ex register
- PC + 1 ← ID/Ex register

⇒ control module produces

PCWrite = 1

IFIDWrite = 0

IFFlush = 1

IDEXWrite = 0

Priority : Branch Misprediction > Stall > Jump Misprediction > Clean

- Branch Misprediction: no data dependency when detected
- Jump Misprediction: may have data dependency for JPR or JRL

Data Forwarding → new feature to HazardDetector module!

(LWD produces data at MEM stage. → stall one cycle
Others produce data at EX stage → no stall

Instruction	Opcode	Function Code	Format
ADD	15	0	R
SUB	15	1	R
AND	15	2	R
ORR	15	3	R
NOT	15	4	R
TCP	15	5	R
SHL	15	6	R
SHR	15	7	R
ADI	4	-	I
ORI	5	-	I
LHI	6	-	I
WWD	15	28	R
LWD	7	-	I
SWD	8	-	I
BNE	0	-	I
BEQ	1	-	I
BGZ	2	-	I
BLZ	3	-	I
JMP	9	-	J
JAL	10	-	J
JPR	15	25	R
JRL	15	26	R
HLT	15	29	R

needs Rs and Rt
needs Rs

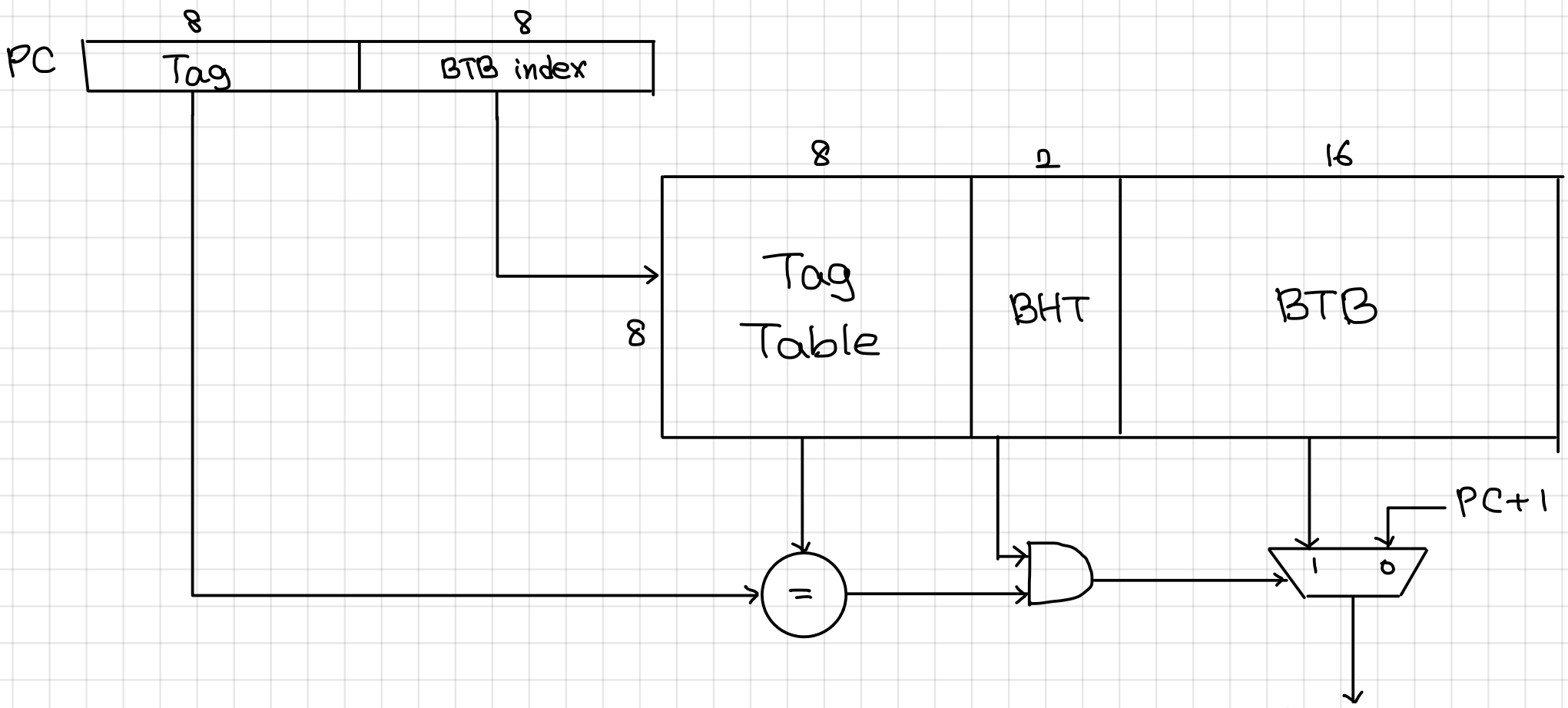
For an instruction in ID stage,

1. RS_{ID} == $dest_{EX}$ & $RegWrite_{EX}$
   if $OPCODE_{EX}$ == LWD,
      stall.
   else
      forward from EX.
2. RS_{ID} == $dest_{MEM}$ & $RegWrite_{MEM}$
   if $OPCODE_{MEM}$ == LWD.
      forward from d_data.
   else
      forward from ALUResult.
3. RS_{ID} == $dest_{WB}$ & $RegWrite_{WB}$
   forward from WB

⇒ done for Rs and Rt

※. $OPCODE_{MEM}$ == LWD is equivalent to
EX_MEM_DataMemRead == 1.

Branch Prediction by 2-bit saturation counter.



Prediction

Save only Jump and Branch instructions.

When a jump or branch is resolved, its instruction address, actual target, and whether it was taken is input to the module

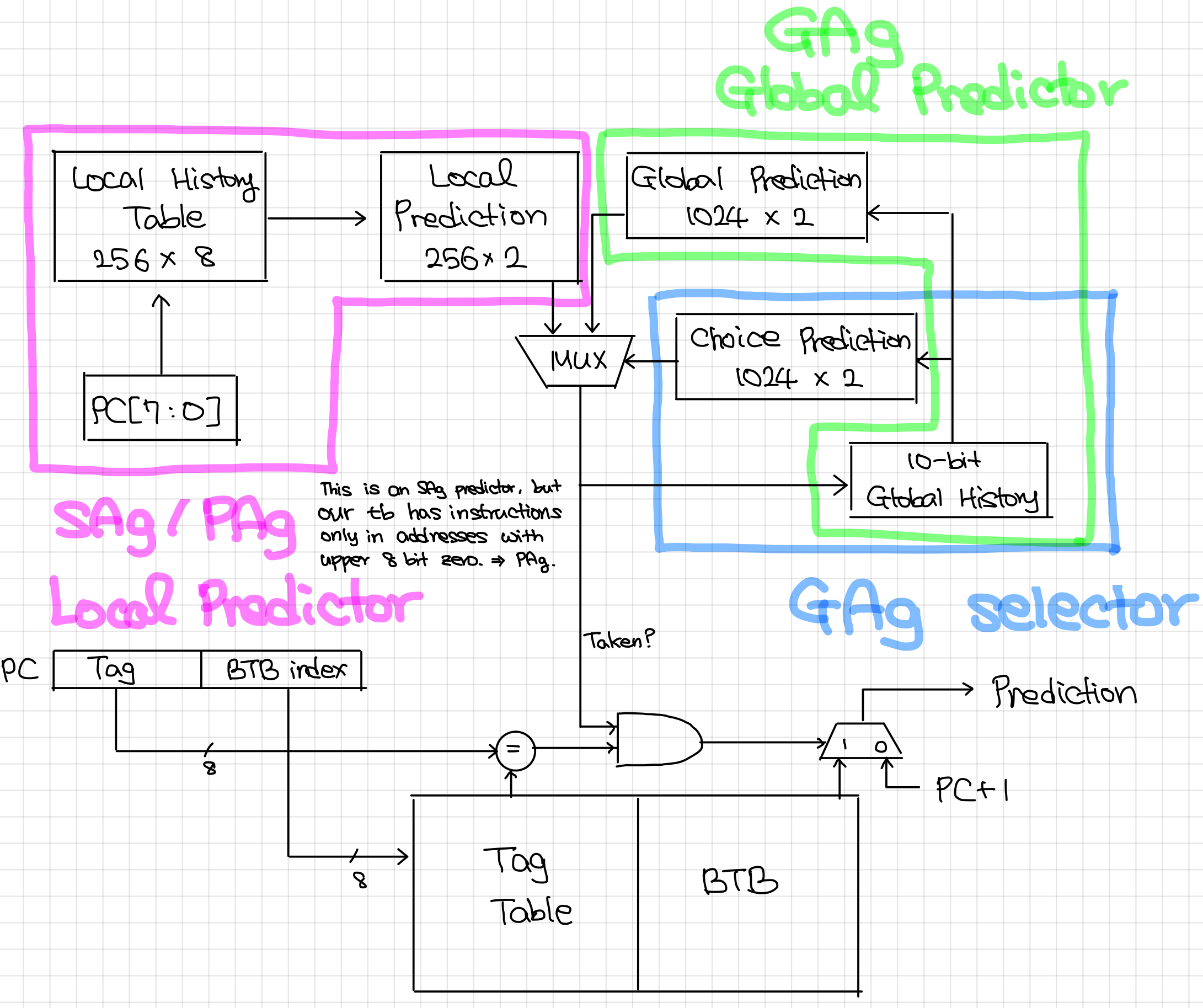
The predictor keeps track of the last two predictions it made.

Branch (in EX stage) Jump (in ID stage) resolved
⇒ BHT[Resolved PC[7:0]] counter update
TagTable[Resolved PC[7:0]] <= Resolved PC[15:8]
BTB[Resolved PC[7:0]] <= actual target

When a jump and branch is resolved in the same cycle,
1) we always update the table based on the branch outcome
2) if the branch was mispredicted, discard jump resolution.

Whether the prediction was correct is determined by comparing the Prediction Address History register and the input 'actual target'.

Branch Prediction by tournament predictor



When outcomes are resolved, resolved PC, its target, and whether it was taken is input to the predictor.

Updating the TagTable and BTB

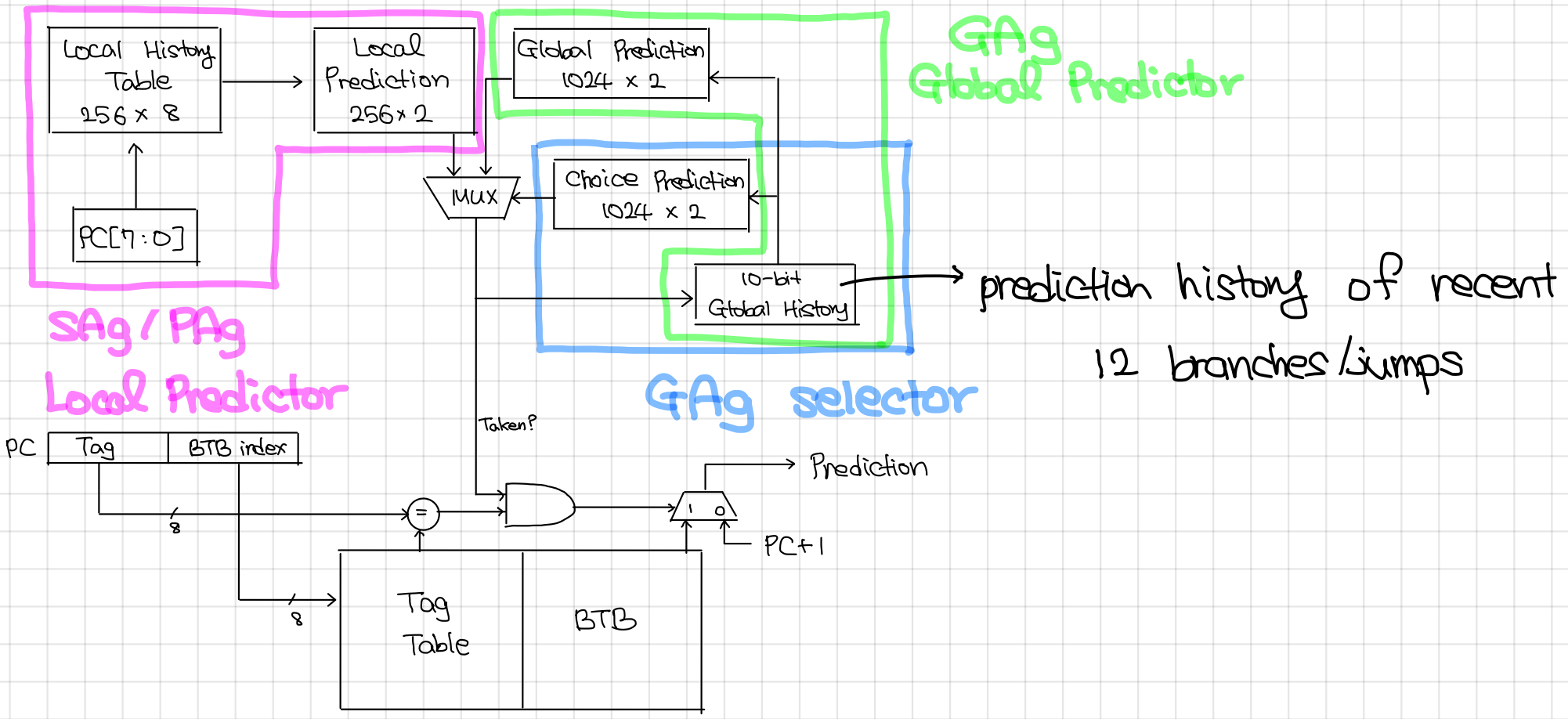
⇒ TagTable[ResolvedPC[7:0]] <= Resolved PC[15:8]

BTB[ResolvedPC[7:0]] <= actual target

When a jump and branch is resolved in the same cycle,

1) we always update the table based on the branch outcome
2) if the branch was mispredicted, discard jump resolution.

Branch Prediction by tournament predictor



Updating the tournament predictors :

Local registers to update predictors when actual outcome is resolved

⇒ GHSR actually 12 bits long. (11:0)

last two prediction histories of local, global, and choice.

① Branches resolved (in EX stage)

GHSR[1] update

LP[LHT[Resolved PC[7:0]]] counter update

LHT[Resolved PC[7:0]] history update

GP[GHSR[11:2]] counter update

CP[GHSR[11:2]] counter update if chosen predictor was wrong and the other was right.

② Jumps resolved (in ID stage)

GHSR[0] update

LP[LHT[Resolved PC[7:0]]] counter update

LHT[Resolved PC[7:0]] history update

GP[GHSR[10:1]] counter update

CP[GHSR[10:1]] counter update if chosen predictor was wrong and the other was right.