DATE

INDEX

(4-STAGE-PIPELINE (CUSTOM DESIGN))

1.0 PROBLEM STATEMENT

Handle these 6 types of exceptions :-

(1) Misaligned Write
(2) Misaligned Read
(3) Out of range memory access
(4) An invalid instruction is processed
(5) A misaligned Branch Address is attempted
(6) Attempt to Branch to an Address outside the range of the IMEM.

1.1 The exceptions are handled in 2 ways :-

on the basis of an input signal OVERRIDE_INTERRUPT

* If override-interrupt is set to 0, the execution stops upon incurring 1.0 (1) to (6). All instructions before the incorrect instruction proceed to completion & all instructions after the incorrect instruction donot enter the pipeline. Control returns to user
* If override-interrupt is set to 1, the incorrect instruction is not allowed to make any write-changes to the system & the control returns to the instruction after the incorrect instruction.

DATE

1.2 When do exceptions occur?

(a) Misaligned Write

- → Denoted by writeerrorflag going high
- → (a) Happens when a word is written to an address not a multiple of 4.

  (b) Happens when a half-word is written to an address not a multiple of 2.

(b) Misaligned Read

- → Denoted by readerror flag going high
- → (a) Happens when a word is read from an address not a multiple of 4.

  (b) Happens when a halfword is read from an address not a multiple of 2.

(c) out of range memory access

- → Denoted by outOfBoundAccess flag going high.
- → (a) Happens when attempted to read from an address outside the DMEM

  (b) Happens when attempted to write to an address outside DMEM.

(d) Invalid instruction is used

- → Denoted by <u>invalidflag</u> going high
- → Any instruction not in RV32I Base ISA, EFENCE, ECALL, EBREAK will be considered invalid (Note :- EFENCE, ECALL, EBREAK are considered invalid here as there is no OS)
- → The instruction 32'h0 is a no-op & is considered valid.

(e) & (f) <u>Misaligned Branch / out of bound branch</u>

- → Denoted by <u>branch-fail flag</u> going high.
- → When? if branch address is not a multiple of 4, or if attempt to branch to an instruction not in range of the IMEM itself

(NOTE: An invalid instruction is handled by the instruction-decoder. It drives all control signals to zero for an invalid signal/instruction. Hence, no damage will be incurred on system i.e. no data will be altered)

DATE

(NOTE:- If a misaligned write/read is encountered, all write signals (from the corresponding stage of the pipeline) is driven low. Hence, no data is altered)

(NOTE:- If an attempt to read/write from an out of bound memory is made, all write signals (from the corresponding stage of the pipeline) is driven low)

(NOTE:- If a branch-fail is detected, no write-changes are made to the system & the next instruction will be taken up (if control has to return to the system & not the user.))

1.3

WARNING:-

* Before an out of bound memory check is done, the test for misaligned read/write has to be complete. If an address is out of bounds & is misaligned too, only ~~not~~ the misaligned read (or write) flag will be driven high, & not the out of Bound Access flag.
out of Bound Access flag goes high only if the targetted memory location is aligned, but is out of bounds of the DMEM.
* JAL/JALR jumping to an invalid location will be detected by the (invalid flag). FLAG HIGH.

DATE

2.0 (a) :- How to demonstrate the system.

(i) USE THE IMEM PROVIDED IN THE ZIP FILE. I have modified it to contain errors.

(ii) On the VIO CONSOLE

ASYNC_OUT ⇒ ({reset, manclk, override_interrupt})

$$\begin{tikzcd}
\text{reset} \arrow[d] & \text{manclk} \arrow[d] & \text{override\_interrupt} \arrow[d] \\
\text{1bit} & \text{1bit} & \text{1 bit}
\end{tikzcd}$$

ASYNC_IN ⇒ ({pcactual (32 bits), x31 (32 bits), idata (32 bits), invalid (1 bit), readerror (1 bit), writeerror (1 bit), outofboundaccess (1 bit), branch_fail (1 bit)})

pc_actual ⇒ current program counter
x31 ⇒ content of R31
idata ⇒ instruction sent to the decoder
other elements of ASYNC_OUT → error flags
manclk ⇒ manual (toggle) clock of the CPU.

(iii) Once the vio console is setup, we have our Override_ interrupt as our input. The error processing F.S.M works on the negative clock edge. ✓ If after a posedge of our manclk, we see an error flag & we want the execution to stop, make override_ interrupt low before the toggling of the clock to get the negedge. Keep this signal low & toggle the clock. The P.C. doesnot increment.

PAGE

(iv) If you want the control to return to the system & not interrupt the flow of execution, set OVERRIDE_INTERRUPT to high upon seeing an error flag after the posedge of the monclock. Continue to set OVERRIDE _INTERRUPT to high for the next two clock cycles. On the next posedge, the 2nd pipeline stage of the instruction after the error-instruction, is being processed in the EXECUTE STAGE.(*)
If this too throws up an error flag, follow steps (iii) & (iv) depending on whether you want the control to return to the user (iii) or control returns to the next instruction (iv).

(*)(For a correct instruction, OVERRIDE_INTERRUPT is irrelevant)

Explanation of the IMEM used :- (2.1)

X FFFFFFFF // invalid instruction. Invalid flag = 1. If OVERRIDE-INTERRUPT kept @ 0, execution stops, else next instruction taken up, after 2 clock cycles.

✓ 00201083 // LH R1, 2(R0) ; correct instruction.

✓ 00100103 // LB R2, 1(R0) ; correct instruction

✓ 01009093 // SLLI 8-bits from R1 to R1 ; correct instruction.

✓ 00811113 // SLLI 4-bits from R2 to R2; Correct instruction.

classmate

✓ 001101b3 //ADD R3, R1, R2 ; correct instruction

✓ 00002203 //LW R4, 0(R0) ; correct instruction

✓ 02321263 // bne R4, R3 ; correct instruction
[Branch not taken as R4 = R3]

ⓧ 7FFFA023 // SW to an address not a multiple of 4.
writeerrorflag = 1. This flag is set high after the posedge for the instruction's 2nd stage. If OVERRIDE_INTERRUPT kept @0, execution stops, else next instruction taken up, after 2 clock cycles (from point of occurence).

ⓧ 7FFFA123 // SW to an address outside the DMEM range.
outofboundaccessflag = 1. This flag is set high after the posedge for the instruction's 2nd stage. If OVERRIDE_INTERRUPT kept @0, execution stops, else next instruction taken up, after 2 clock cycles (from point of occurence)

✓ 00601283 // LH R5, 6(R0) ; correct instruction.

✓ 00500303 // LB R6, 5(R0) ; correct instruction.

✓ 00400383 // LB R7, 4(R0) ; correct instruction.

✓ 01029293 // SLLI 8-bits from R5 to R5 ; correct instr.

✓ 00831313 // SLLI 4-bits from R6 to R6 ; correct instr.

✓ 00638433 // ADD R8, R7, R6 ; correct instr.

✓ 00540433 // ADD R8, R8, R5 ; correct instr.

ⓧ 00702483 //LW from an address not a multiple of 4.
readerrorflag = 1. This flag is set high after the posedge for the instruction's 2nd stage. If OVERRIDE_INTERRUPT kept @0, execution stops, else next instruction taken up, after 2 clock cycles (from the point of occurence)

✓ 00849463 // bne R9, R8. Since the last instruction had to load into R9, but was not executed, since there was a read error, the value of R9 to be used in this instruction depends on what the user wants:-

(i) If you want to use the actual value of R9 (actual value of R9 is the value of R9 before the previous erroneous load instruction could enter the pipeline (or) it is the ~~act~~ true value of R9, given that the erroneous load instruction didn't execute), stall the pipeline for 3 cycles, by keeping the <u>OVERRIDE_INTERRUPT</u> low for 3 cycles after the erroneous load instruction completes it's first stage.

(ii) If you want to use R9 as 0 (since R9 was attempted to be incorrectly loaded in the previous instruction, using 0 for R9 can also be a reasonable choice), keep the "override_interrupt" signal high throughout all stages of the previous misaligned load instruction.

(Here in simulation, (ii) is used)

(NOTE:- irrespective of (i) or (ii), the branch is taken up.

classmate

DATE

✓ fbdff56f // jal instruction ; correct instruction (not executed, before the branch in the previous instruction is taken)

✓ fb1fffef // jal instruction to 1st instruction ; correct instruction. This is the branch target of 00849463. (& loop is induced)

(PROCEDURE NOW REPEATS)

2.0 (b) * The easiest way to run the imem is to keep 'OVERRIDE_ INTERRUPT' signal to 1 (i.e. return control to the next instruction) throughout the demonstration & just toggle the (manclk) manual clock on the VIO console. (See the connections on the VIO console first)

✓ ⇒ correct INSTRUCTION

(×) ⇒ Incorrect INSTRUCTION (as per (1.0)(1) to (1.0)(6))

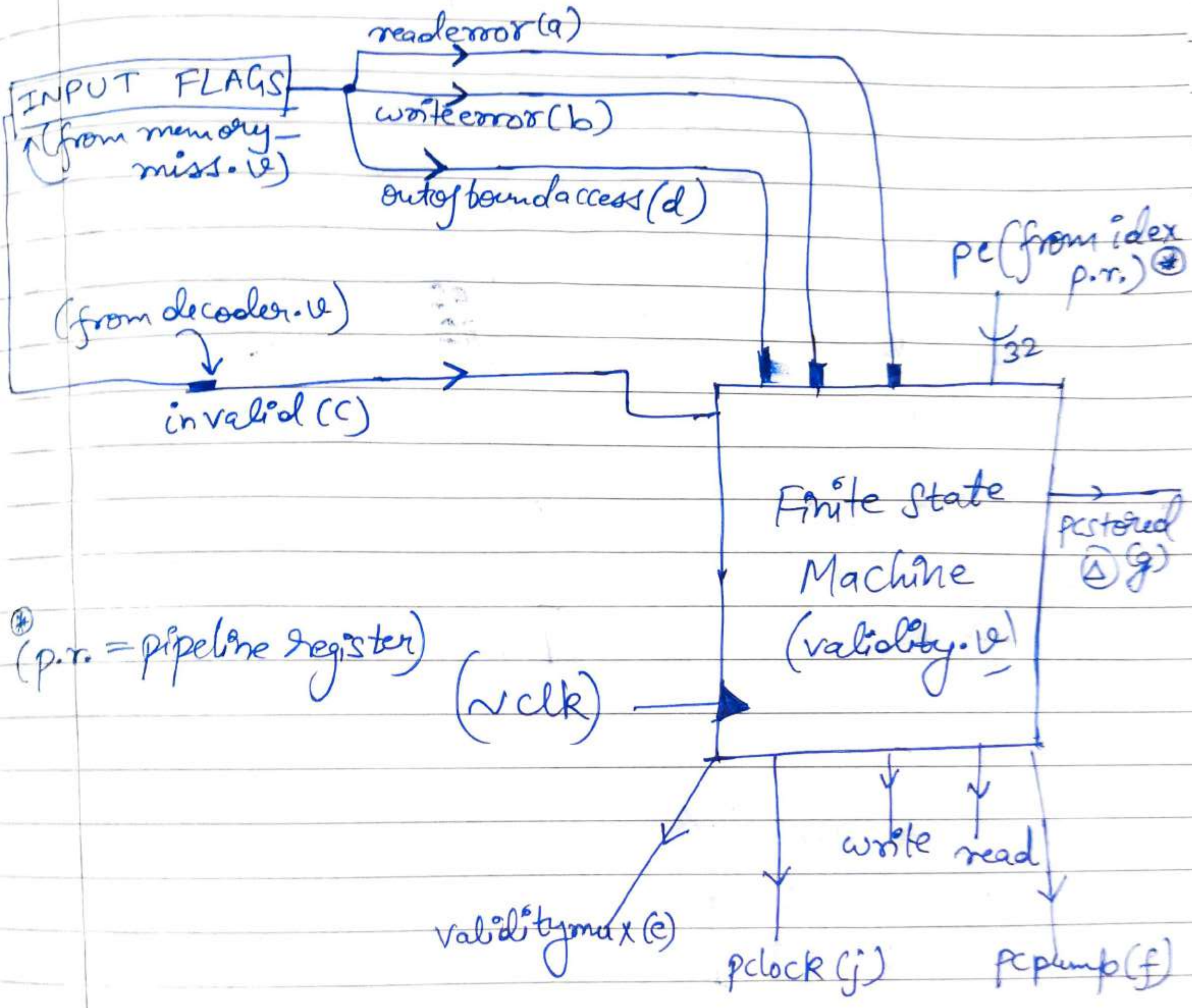
If 2.0 (b) is followed, correct flow of instructions (PC) is (0, 4, 8, 12, 16, 20, 24, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, 72 (idata = 0), 76, 80, 0, ...)

$\downarrow$ PC is 76 but idata (instr.) to decoder is no-op.

PAGE

DATE

3.0 ERROR DETECTION & HANDLING LOGIC



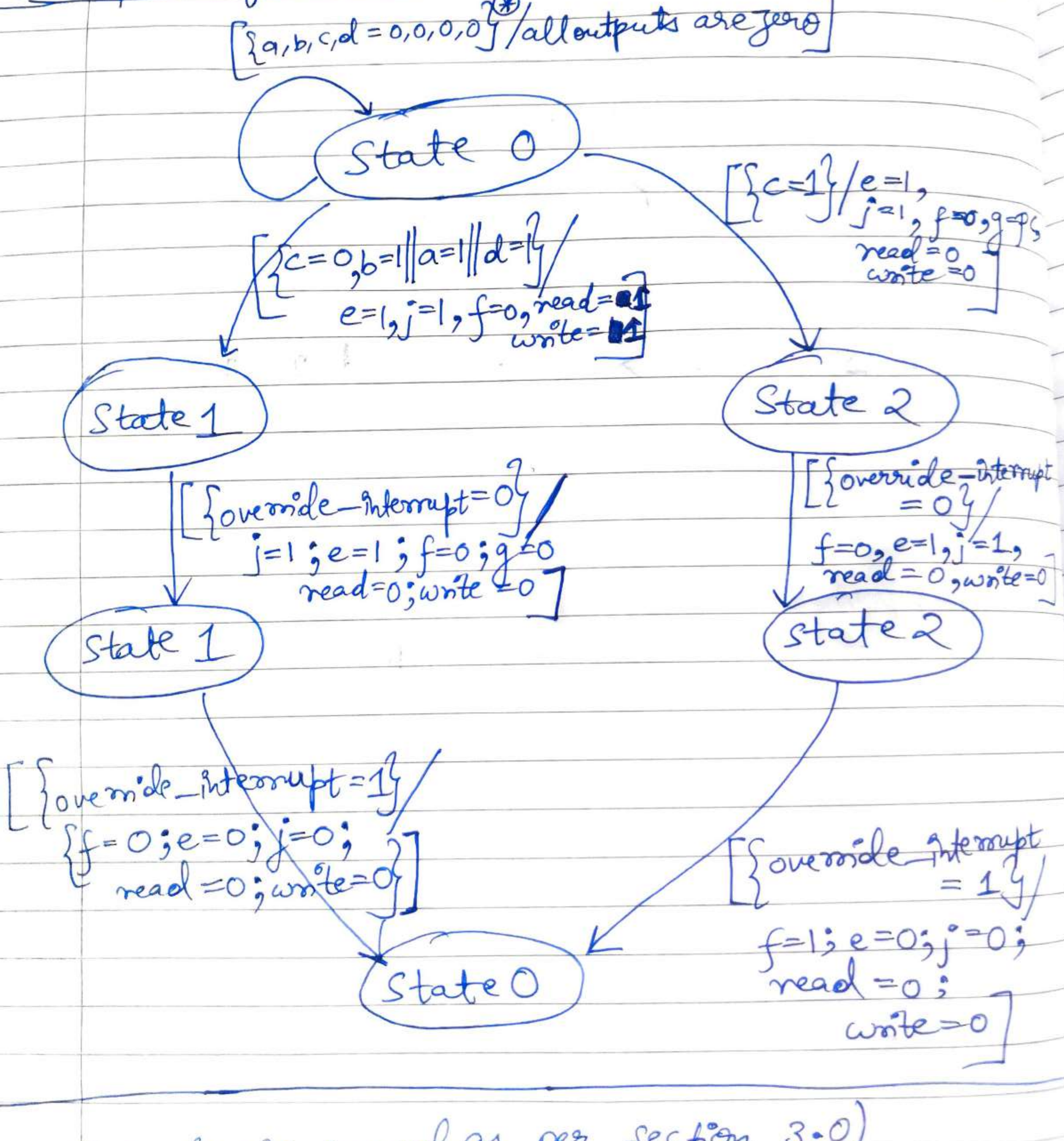
(*) (p.r. = pipeline register)

(Δ) → signal 'g' is not exactly the output of the FSM. The FSM output is further modified to yield (g) this way :- unless otherwise mentioned (g) retains its' previous value.

DATE

3.1

Description of the FSM (validity.v)


[{a,b,c,d = 0,0,0,0}(*)/all outputs are zero]
State 0
[{c=1}/e=1, j=1, f=0, g=0; read=0 write=0]
[{c=0, b=1||a=1||d=1}/ e=1, j=1, f=0, read=1 write=1]
State 1
State 2
[{override_interrupt=0}/ j=1 ; e=1 ; f=0 ; g=0 read=0; write=0]
[{override_interrupt =0}/ f=0, e=1, j=1, read=0, write=0]
State 1
state 2
[{override_interrupt=1}/ {f=0 ; e=0; j=0; read=0 ; write=0}]
[{override_interrupt = 1}/ f=1; e=0; j=0; read=0 ; write=0]
State 0

(signals are named as per section 3.0)

{(*) → missing input signals are don't care}

DATE

3.2 FUNCTIONS OF THE OUTPUTS OF FSM (VALIDITY.V)

1.) pclock(j) → (Acts as input to imem-pc-control-origin.v)
↳ If this is set to 1, PC doesnot increment i.e PC is locked. If low, normal operation of PC is sustained (depending on PC Pump)

2.) PCpump(f) → (acts as input to imem-pc-control-origin.v)
↳ If this is set to 1 (& pclock is 0), the value of FSM output pcstored (g) is forced into the PC @ posedge clock. If low (& pclock is 0), normal operation of PC is sustained (depending on PCfreeze → briefly described in Assign.4, this is nothing new)

3.) Validitymux(e) → (acts as input to idatamux1.v)
↳ If set to 1, '0' (or no-op instruction) is sent in to the decoder to process. If low, output of idatamux.v is sent to the decoder. Idatamux.v follows from assign 4.

4.) write → (acts as input to readwritecontrolmux.v) $\triangle$
↳ If set to 1, indicates a write alignment error being caught/write out of range error being caught, while the (error-creating instruction is in (it's) 2nd stage. The readwritecontrolmux.v upon detecting the write signal to be high disables all write-enable signals propagating from the 2nd stage of pipeline.

DATE

* read → (acts as input to readwritecontrolmux.v) $^{\triangle}$

↳ If set to 1, indicates a read-error being caught (it might be readalignment or read out of range error) while the error-creating instruction is in its 2nd stage. The readwritecontrolmux upon detecting a high read signal, disables DMEM reads & register write enable propagating from the 2nd stage of the pipeline

---

$\triangle$ → it's actually readcontrolmux.v

---

* pcstored → (Actsasinput to imem_pc_control_origin.v)
(g)

→ Stored the return address of the instruction, in case of an invalid instruction being detected (i.e. invalid = 1)

→ For other types of errors (like write, read, out of Bound access), the return address need not be stored in pcstored as the Pclock control signal stops PC from incrementing & hence. $\triangle$

($\triangle$ → the same logic could have been applied to the 'invalid' error, but that's how it is implemented)

DATE

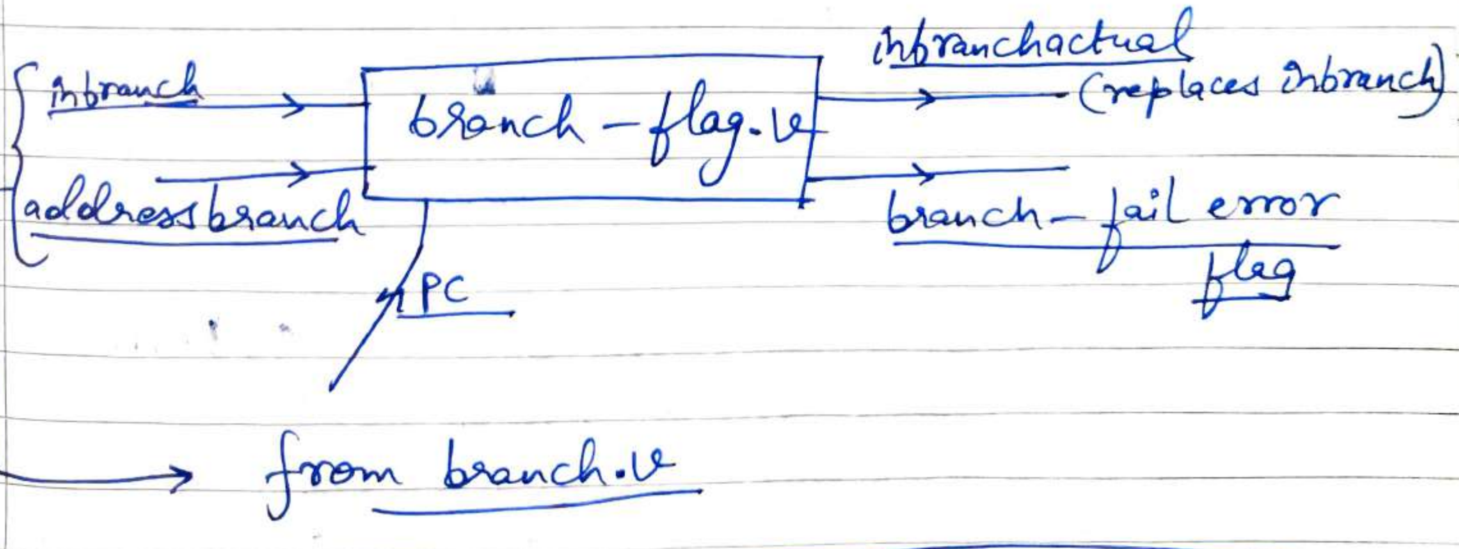
3.3 MISALIGNED / OUT OF RANGE BRANCHES

* In assignment 4, the entire branching logic was based on a control signal <u>inbranch</u>. In the 2nd stage of a branch instruction, if the branch was being taken up, inbranch was set to 1 & the current instruction was flushed out from the pipeline.

* Now, we use an extra module branch_flag. It takes in inputs → inbranch, pc from idex pipeline register (forwarded PC), address offset & produces a branch_fail and an inbranchactual.

* <u>Branch error handling :-</u>

Given the inbranch as 1, if the address offset is misaligned and/or the branch target is beyond the IMEM range, the inbranchactual is made 0 & branch_fail is set as 1.



PAGE

DATE

3.4 Summary.

(a) To get an idea of how the F.S.M. handles errors, look at the attached simulation waveform⊛ from 10ns to 50ns (invalid flag), 210ns to 250ns (writeerror flag), 250ns to 290ns (outof boundaccess error), 430ns to 470ns (~~[illegible]~~ readerror flag), together with the explanation given in section 3.0, 3.1, 3.2.
⊛(In simulation, override set to 1)
The remaining design of the pipeline follows from the assignment 4 submission. (4-staged pipeline used)

(b) The user has power of wanting to return the control to themselves or to the next instruction through OVERRIDE_INTERRUPT (as described in 1.1)

(c) If an instruction attempting to write to a register fails & that ~~[illegible]~~very register is a source register for the next instruction, user can exercise whether they want the original value of the register to be used or ~~[illegible]~~ zero to be used (depending on OVERRIDE_ INTERRUPT, explained in section 2.1 for instruction 32'h 00849463.