Faculty of Engineering, Cairo University

Computer Engineering Department

Computer Architecture Course Project

Orthrus - Phase 1 Report

Ahmad Khaled	Section 1 BN. 03
Maryam Shalaby	Section 2 BN. 21
Zeinab Rabie	Section 1 BN. 22
Omnia Zakaria	Section 1 BN. 23

Contents

1 Introduction

Orthrus is a pipelined static dual-issue microprocessor implementing a RISC ISA similar to the MIPS ISA. In the following sections we outline the instruction format, the general design of the processor, as well as the pipeline stages design.

2 Instruction Set Architecture

The structure for the IR for One-Operand instructions is given in Figure 1.The structure for the IR for Two-Operand instructions is given in Figure 2. The structure for the IR for Memory instructions is given in Figure 3. The structure for the IR for Branch & Change of Control instructions is given in Figure 4.

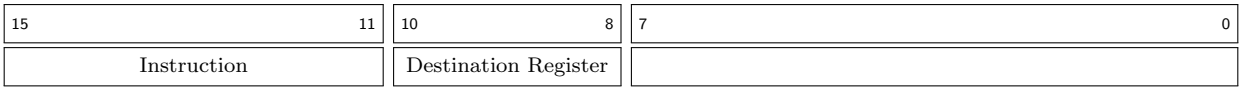
	R_0	R_1	R_2	R_3
Code	000	001	010	011
	R_4	R_5	R_6?	R_7
Code	100	101	110	111

Table 1: Register Addressing Codes

Instruction	NOP	SETC	CLRC	NOT	INC	DEC	OUT	IN
OP Code	00000	00001	00010	00011	00100	00101	00110	00111

Table 2: One-Operand Instruction Codes

Figure 1: IR Structure For One-Operand Instructions



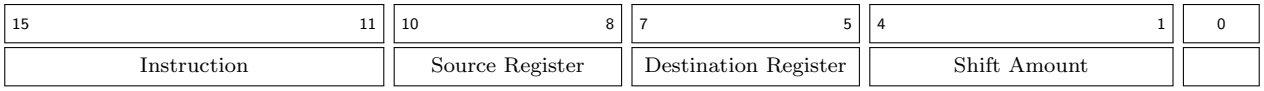
Instruction specifies the instruction to execute. Possible codes given in. Table 2.

Destination Register specifies the destination register. Possible codes given in Table 1.

Instruction	MOV	ADD	SUB	AND	OR	SHL	SHR
OP Code	01000	01001	01010	01011	01100	01101	01110

Table 3: Two-Operand Instruction Codes

Figure 2: IR Structure For Two-Operand Instructions



Instruction specifies the instruction to execute. Possible codes given in. Table 3.

Source Register specifies the source register. Possible codes given in Table 1.

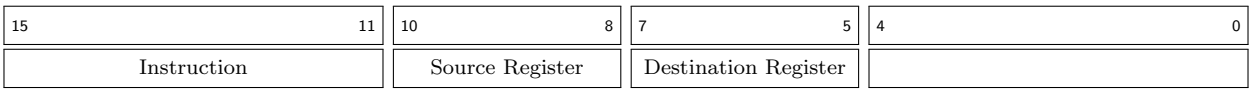
Destination Register specifies the destination register. Possible codes given in Table 1.

Shift Amount specifies the number of bits to shift the source Register by.

Instruction	PUSH	POP	LDM	LDD	STD
OP Code	01111	10000	10001	10010	10011

Table 4: Memory Instruction Codes

Figure 3: IR Structure For Memory Instructions



Instruction specifies the instruction to execute. Possible codes given in Table 4.

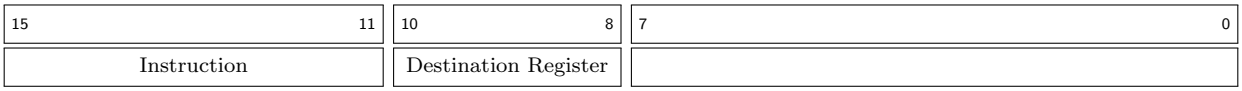
Source Register specifies the Source Register. Possible codes given in Table 1.

Destination Register specifies the Destination Register. Possible codes given in Table 1.

Instruction	JZ	JN	JC	JMP	CALL	RET	RTI
OP Code	10100	10101	10110	10111	11000	11001	11010

Table 5: Branch & Change of Control Instruction Codes

Figure 4: IR Structure For Branch & Change of Control Instruction



Instruction specifies the instruction to execute. Possible codes given in Table 5.

Destination Register specifies the destination register. Possible codes given in Table 1.

3 Design

3.1 Overall System Schematic

Figure 5 and Figure 6 show the overall design of Orthrus, along with the main connections. Note that some of the connections specific to hazards and forwarding are omitted for clarity, and that some of the connections between pipeline register buffers are also omitted for clarity (but if two registers share the same name across two register buffers then they should be connected). The units in each stage are responsible for generating control signals for themselves given the inputs from the previous stage, from the memory, and from the Hazard/Forwarding unit. In the following subsections, we discuss each unit shown in the diagram in detail. We mostly suppress the details of hazard handling by assuming that all the units mentioned always have a **stall** input and a **flush** input that cause the unit to stall its corresponding stage (by propagating a No-Op to the next instruction) or flush whatever instruction is in it (by propagating a No-Op to the next instruction and having the previous pipeline buffer reset). A more thorough discussion of hazards is given in the Hazard unit subsection. We have five pipeline stages (fetch / decode / execute / memory access / write back). All the pipeline registers needed are shown in the overall schematic. They are all 32-bits in width. The pipeline stages are five, each enclosed in rectangular blocks with dashed outlines.

3.2 Memory Unit

The memory block used is interfaced by two address lines, with the second line given priority whenever a read / write is requested by it. A single word is read or written when an address is on Address2. When an address is on the first line, **two words** are read at a time starting from the word at the address given by Address1. The first line is only for reading from the memory and has no write option. The read operation is instantaneous while the write operation takes 1 clock cycle (the value is written onto the memory on the falling edge of the clock).

3.3 Fetch Stage

In the fetch stage, the Program Counter (PC) register is connected to Address1 of the memory, the data (from MemDataOut) is then read into the fetch and pre-decode unit which has three outputs: the IRs (Instruction Registers) corresponding to the two streams in Orthrus, IR1 and IR2, as well as the increment to add to the PC for fetching the next instruction.
The pre-decode unit handles several issues:

1. If an Interrupt or Reset have been latched (we assume that dedicated hardware has already latched them) then the PC is fetched from dedicated memory locations (M[0] for RST and M[1] for ITR) and the appropriate instructions are passed into the pipes.

2. The unit must do not let two memory-accessing instructions simultaneously into the two streams (since this would result in a structural hazard) and hence passes a No-Op into the second stream when this is detected.

3. The unit also passes a No-Op into the second stream when an immediate instruction is loaded, since an immediate instruction takes two words.

4. The unit also passes a No-Op when the two instructions try to use the output port.

5. If a CALL or CALL-like instruction is detected, then the next instruction passed is a No-Op (since it will be flushed anyway).

The flow chart for the unit's operation is given by Figure 7. For the flow chart, the instructions that access memory are: PUSH / POP / LDD / STD / CALL / RET / RTI. In addition to the Pre-Decode unit mentioned, the fetch stage also includes an incrementor to increment the PC as well as a small unit (shown in Figure 8) to choose whether to get the PC from the incremented PC, normal branches, from RET / RTI (which are executed in the last stage by necessity of fetching from memory), or from special memory places (for RST / ITR).

3.4 Decode Stage

The Decode Stage is responsible for translating the received IR into signals that will drive the next stages. Those signals are grouped into a control word, shown in Figure 9.
The flow charts in Figures 10 and 11 show the decision making process that sets the Control Word.And it is divided into Four branches.

1. One-Operand Instructions: At which the decode stage passes the ALU Operation, the contents and address of the destination register (RT) and since they are one-operand instructions, the source register (RS) is nonexistent and thus its value is passed as 0 to the ALU.

2. Two-Operand Instructions: If the instruction is of type shift, the Decoding unit extends the shift amount (Shamt in IR) and places into RT and forwards value of RS register for ALU operation. As for other instructions, RT and RS values and addresses are both forwarded.

3. Memory Operation: If Push or Pop, it sets the ALU op to F= SP-1 and F= SP+1 respectively (the ALU has a dedicated incrementor-decrementor for the SP register, as it is 32b vs all other outputs being 16b). As for Loads and Stores, needed values are just passed along. (F=B)

4. Branching: Branching check is done in the decoding stage, and if condition is true, RT value is input to PC. By setting IsBranch and Branch Address.

3.5 Execute Stage

In execute stage, set logic is carried out by ALU and the Flag register is set accordingly. The two Pipes edit ONE flag register, and due to its latency in sequence, pipe 2 is prioritized when a conflict arises in setting the flags. It is a simple issue to handle as show in Figure 12, there was no need for the Hazard unit to handle it. In addition to the shown logic, a few simple MUXs (not shown) are responsible for copying the FLAGs register into the FLAGS_LNK register (or restoring it) when INTR/RTI are detected, respectively.

3.6 Memory Stage

In the Memory access stage, according to operation type, MAddr (The address to send the memory) and MData (The data to send the memory / also receives the data in Store operations) are set if needed (load for example doesn't need MData to be set) as shown in the flow chart in Figure 13. In addition to managing this, the memory stage unit also outputs signal WPC_Write when the PC's value should be taken from the memory.

3.7 Write Back Stage

In the write back stage, we have to deal with the following outcomes

1. Writing back to the R[RT] the Value in ARes if the current operation involves writing to a register.
2. Writing to the Out port the value in ARes.
3. Updating SP values with ARes if it was any operation that changes the SP.
4. Nothing needs to be written back, the stage does nothing.

3.8 Hazard Detection Unit

For clarity, the hazard detection wires aren't wired in the overall schematic into the components inside each stage. To be as clear as possible, we mention our assumptions:

1. A **stall** input is available to all components. When this input is set to true, the component passes zeros (i.e. a No-Op) as an output regardless of the input present in the previous buffer.
2. A **flush** input is available to all pipeline register buffers, when turned on the by the detection unit all the pipeline buffers are immediately replaced by zeros. This helps us correct mispredicted branches.
3. If some line is forwarded by the Hazard unit to any stage, it **MUST** be taken by that stage instead of the original line. So if, for example, a line is shown in the schematic as connected to the RS line coming in from the register buffer but the Hazard unit forwards another input then the new input must be taken instead. Every forwarded line has an associated bit that tells the stage whether or not to take it.

We deal with three types of Hazards:

1. The only data hazards that affect the system are RAW data hazards, as the system's sequential integrity is preserved by making sure stalls affect both pipes. WAR hazards have no impact because we send the needed data into the separate pipeline registers when needed, RAR hazards cause no problems, and WAW hazards on different registers do not affect the system's correctness because they write in separate places. Since pipe 2 (the lower pipe) has the later instruction in sequence extra precautions for data dependencies were taken as shown in the figures. Figure 15 shows how RAW hazards (to the general purpose registers) are handled and Figure 16 shows the checks made when handling the SP, since it is on its own accord in the pipe. A Special case is in branch operations for RT that should contain memory address to branch address as shown in figure 19.
2. Structural Hazards: WAW hazards on the same register are handled by enforcing a priority system with the higher priority of writing to some register given to the second pipe since it is the one containing the newer instruction, as shown in Figure 17.
   Because this is a Von-Neumann architecture, we have only one memory: A structural hazard arises when fetching instructions while simultaneously accessing the memory (since the memory can only be read from one address at a time), in this case the MEM stage is given priority

by stalling the fetch. See Figure 18. A similar hazard arises if both pipes desire access for the memory, this is solved by not allowing two memory-accessing instructions together back in the Fetch Stage.

3. Control Hazards: The following are the main hazards:

   (a) We always predict not taken in branches. If the prediction fails, then the fetch instruction preceding it is flushed.

   (b) If a CALL or unconditional JMP is detected, then the instructions in Fetch are flushed. If RET or RTI are detected, then no new instructions are loaded until the new PC value is fetched (in the memory stage), as shown in Figure 20.

   (c) If either an INT signal or a RST signal is detected, no new instructions are fetched, then the Fetch Stage handles this by changing the PC to the appropriate value directly, and there is no need for the hazard to be handled elsewhere.

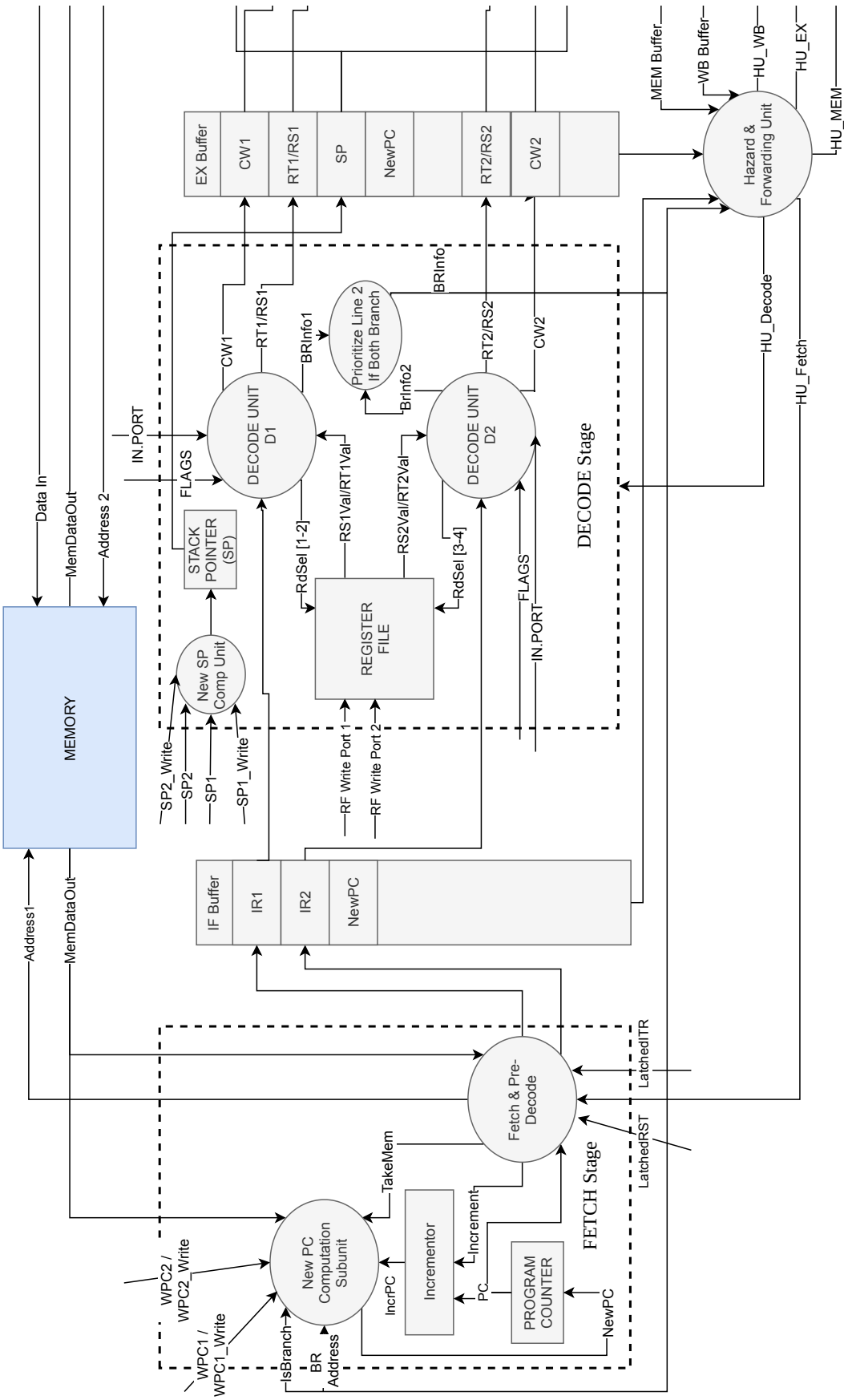


Figure 5: Overall Schematic Part 1

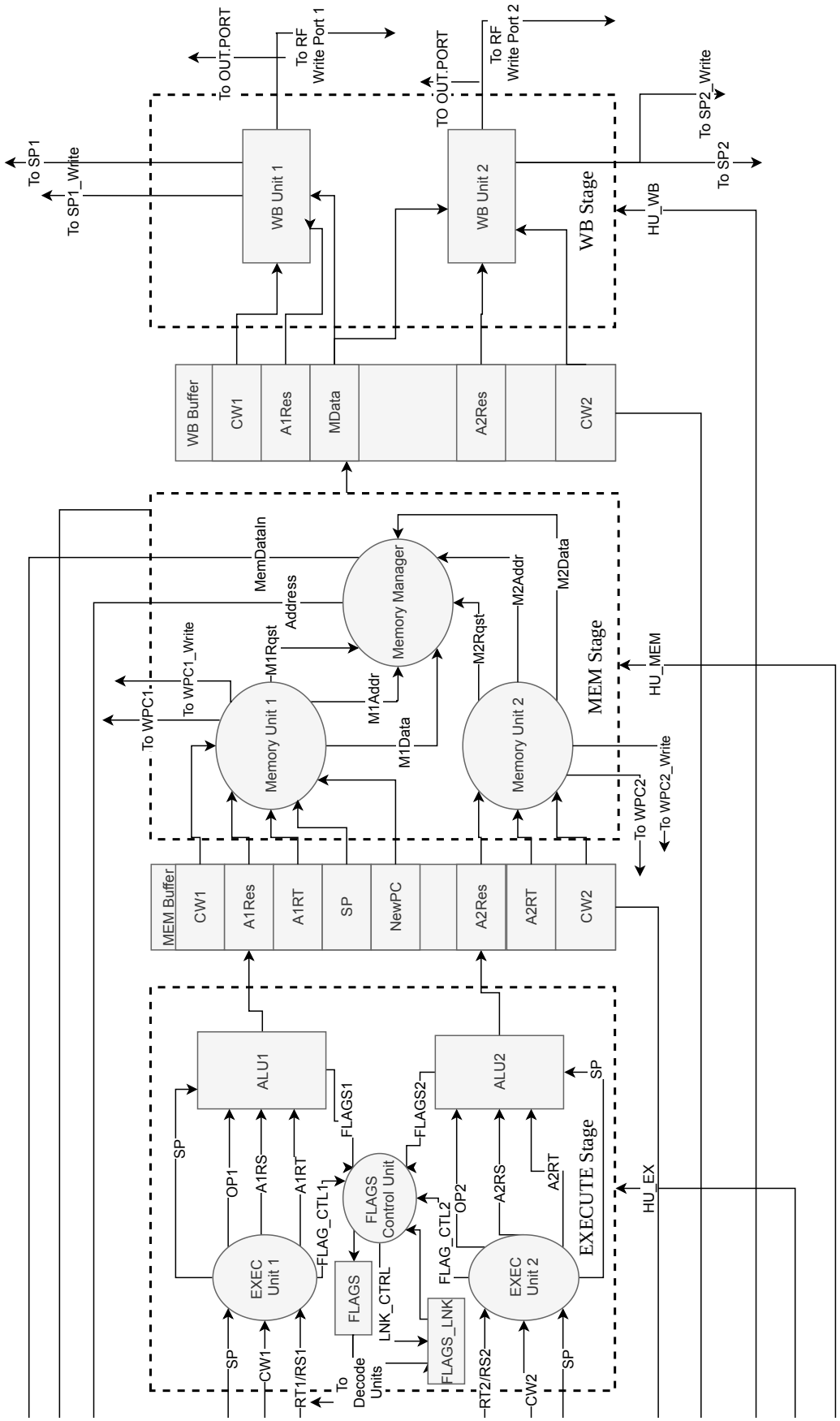


Figure 6: Overall Schematic Part 2

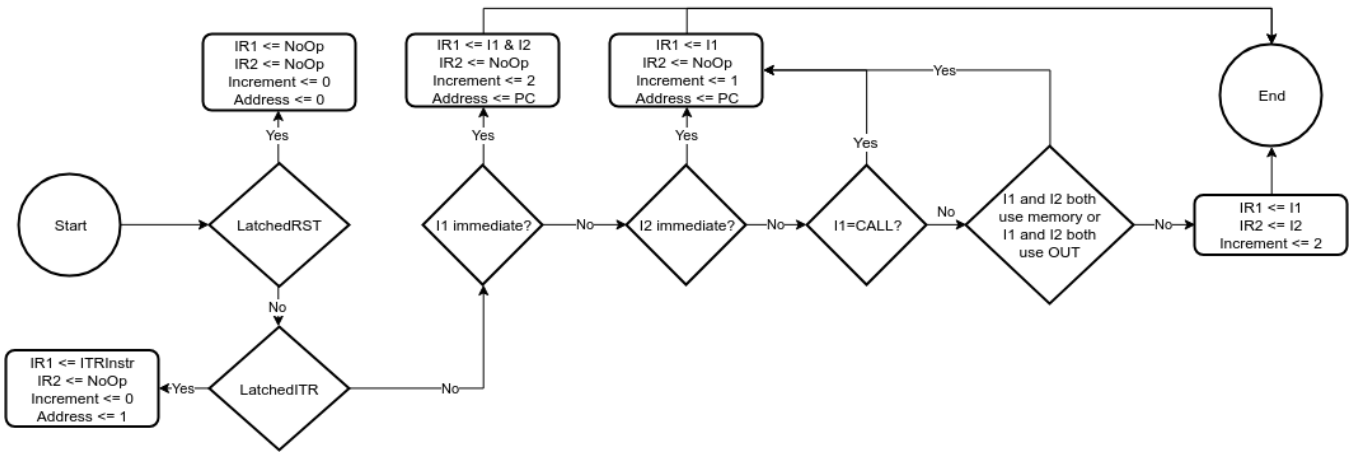


Figure 7: Fetch and Pre-Decode Unit Operation

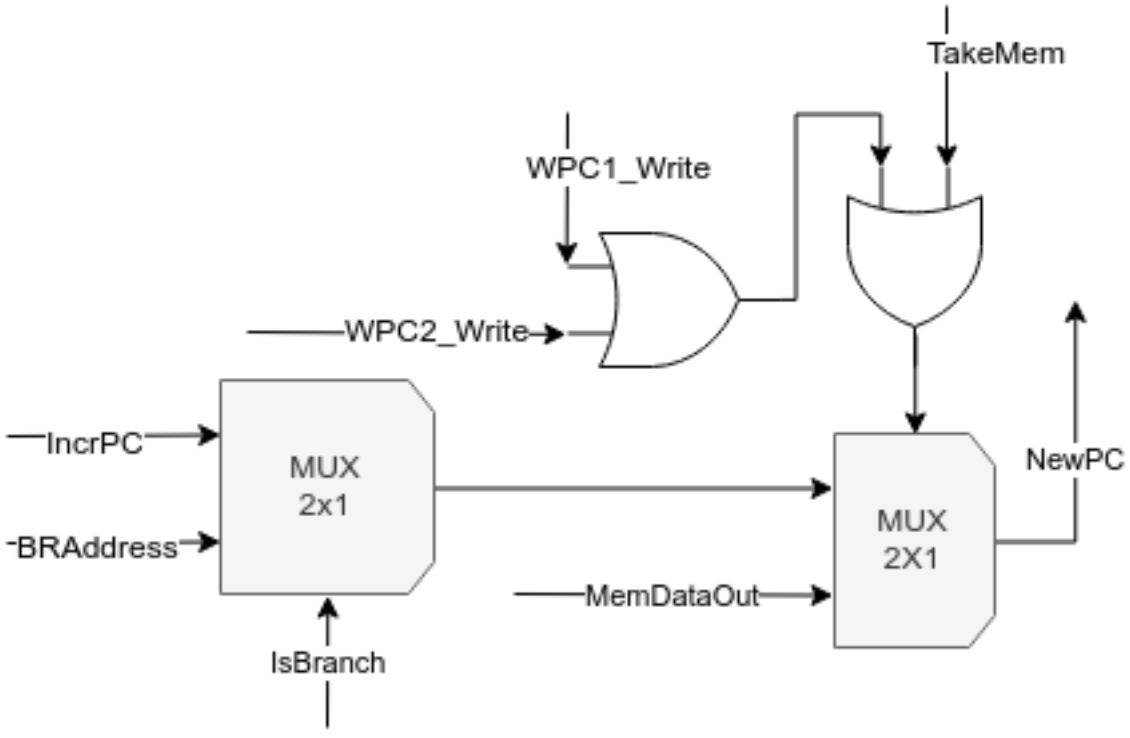


Figure 8: Next PC MUXing Unit

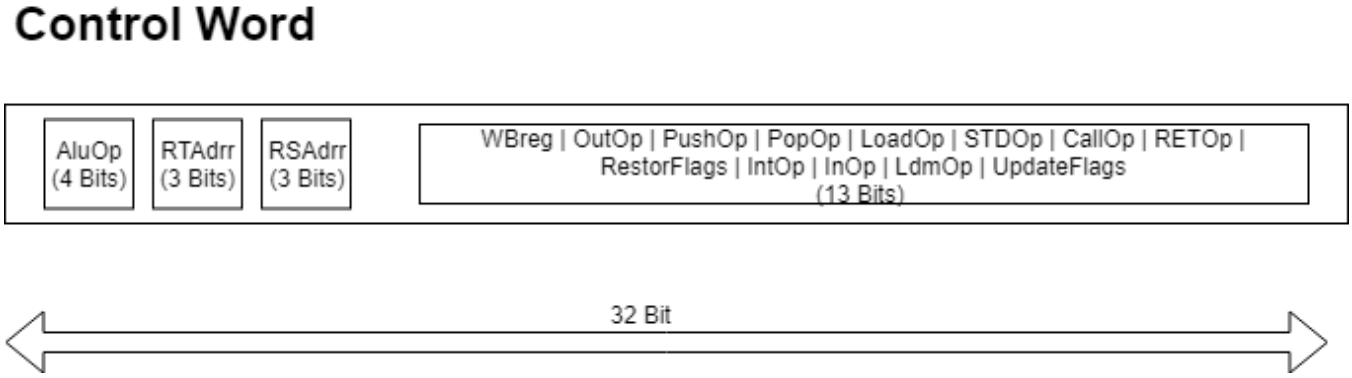


Figure 9: Control Word

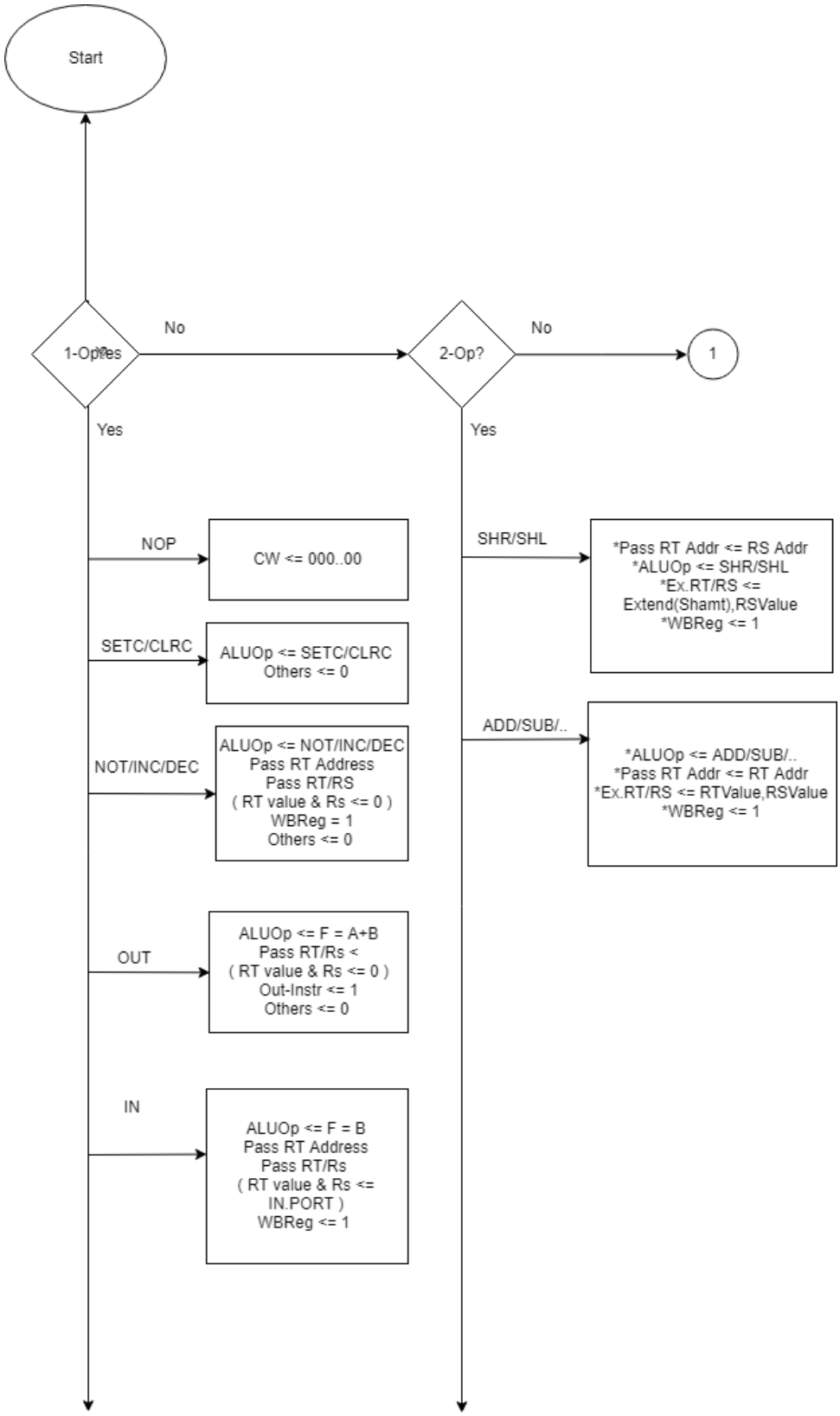


Figure 10: Decode Flow Chart P1

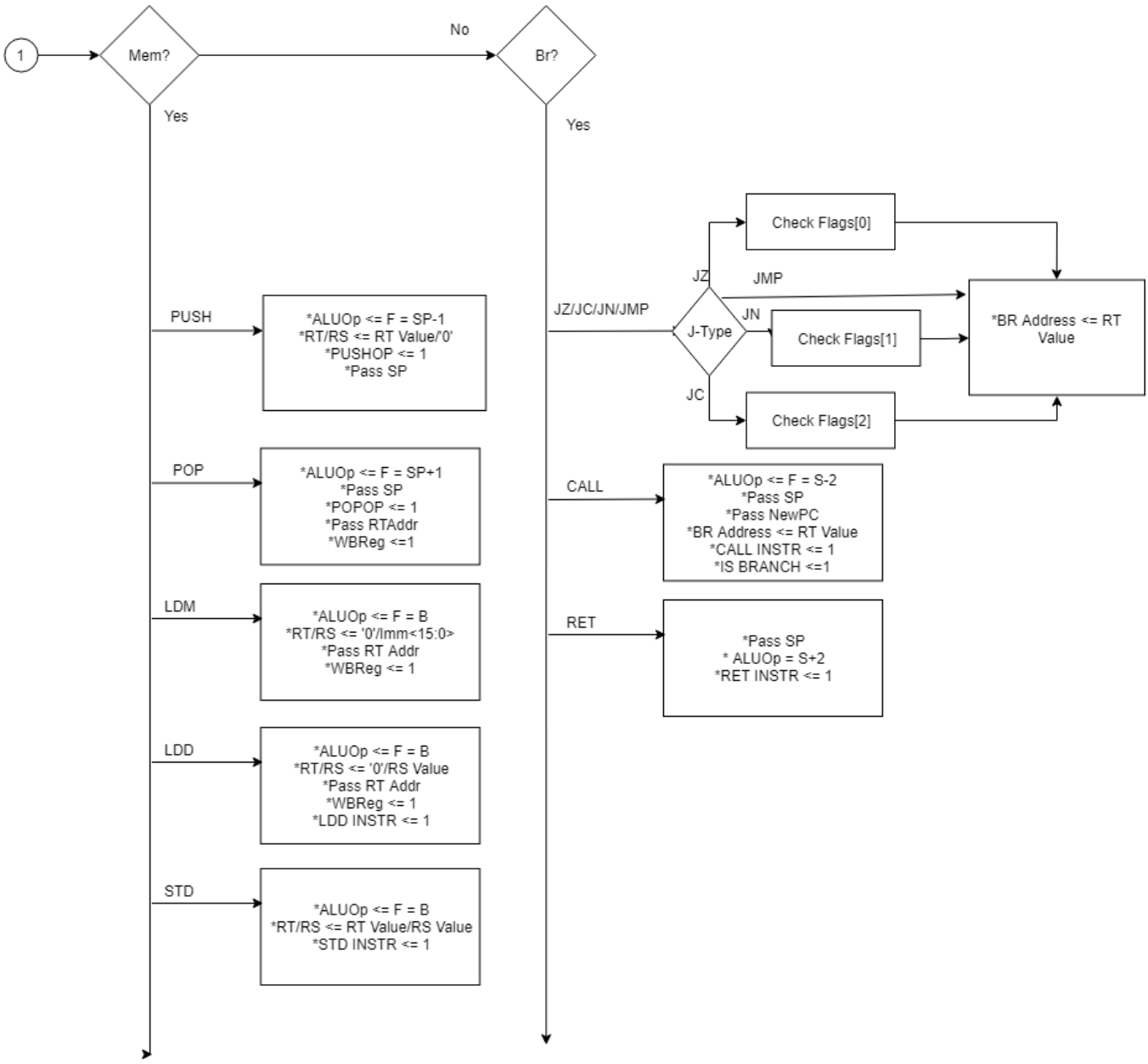


Figure 11: Decode Flow Chart P2

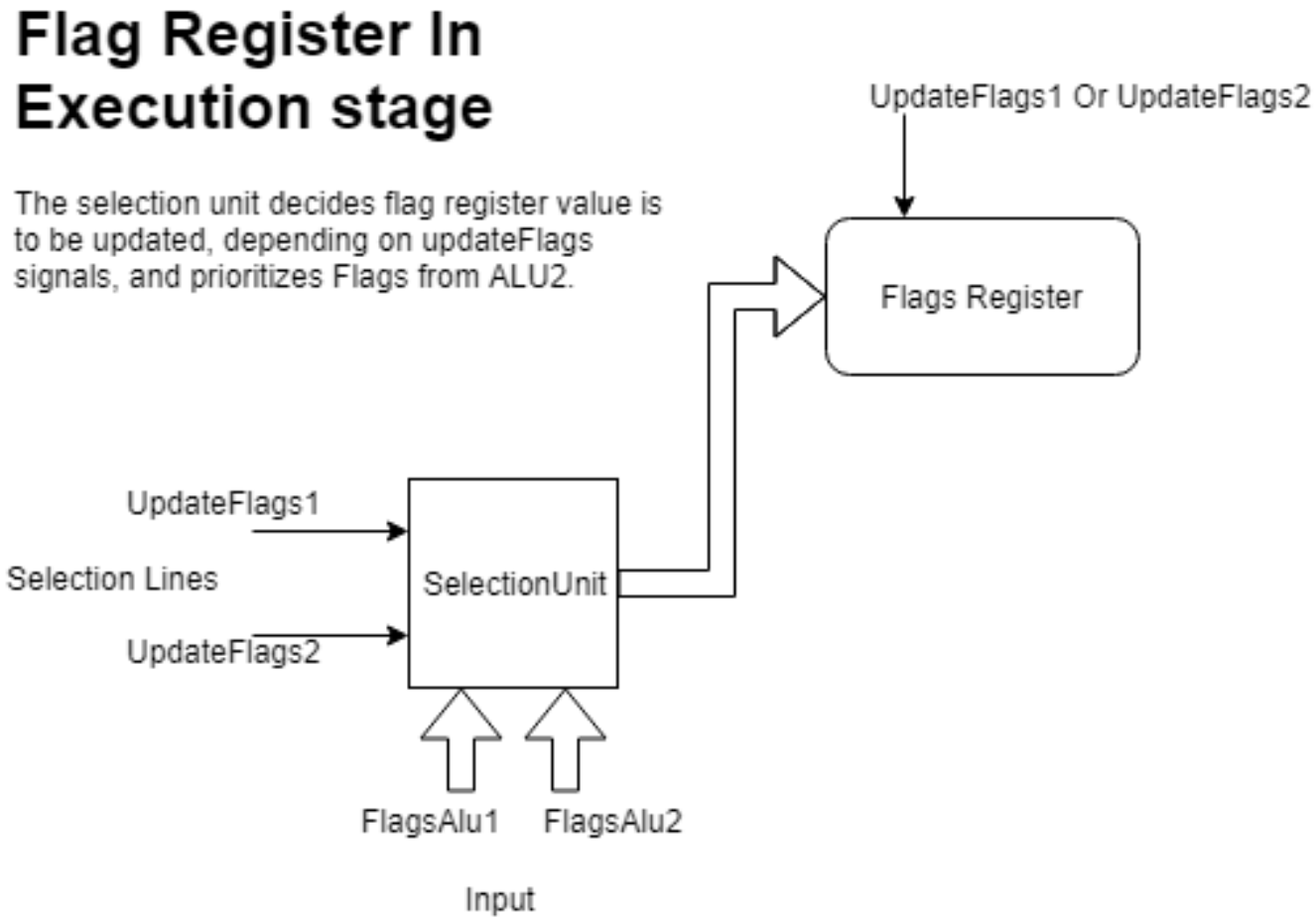


Figure 12: Flag Register

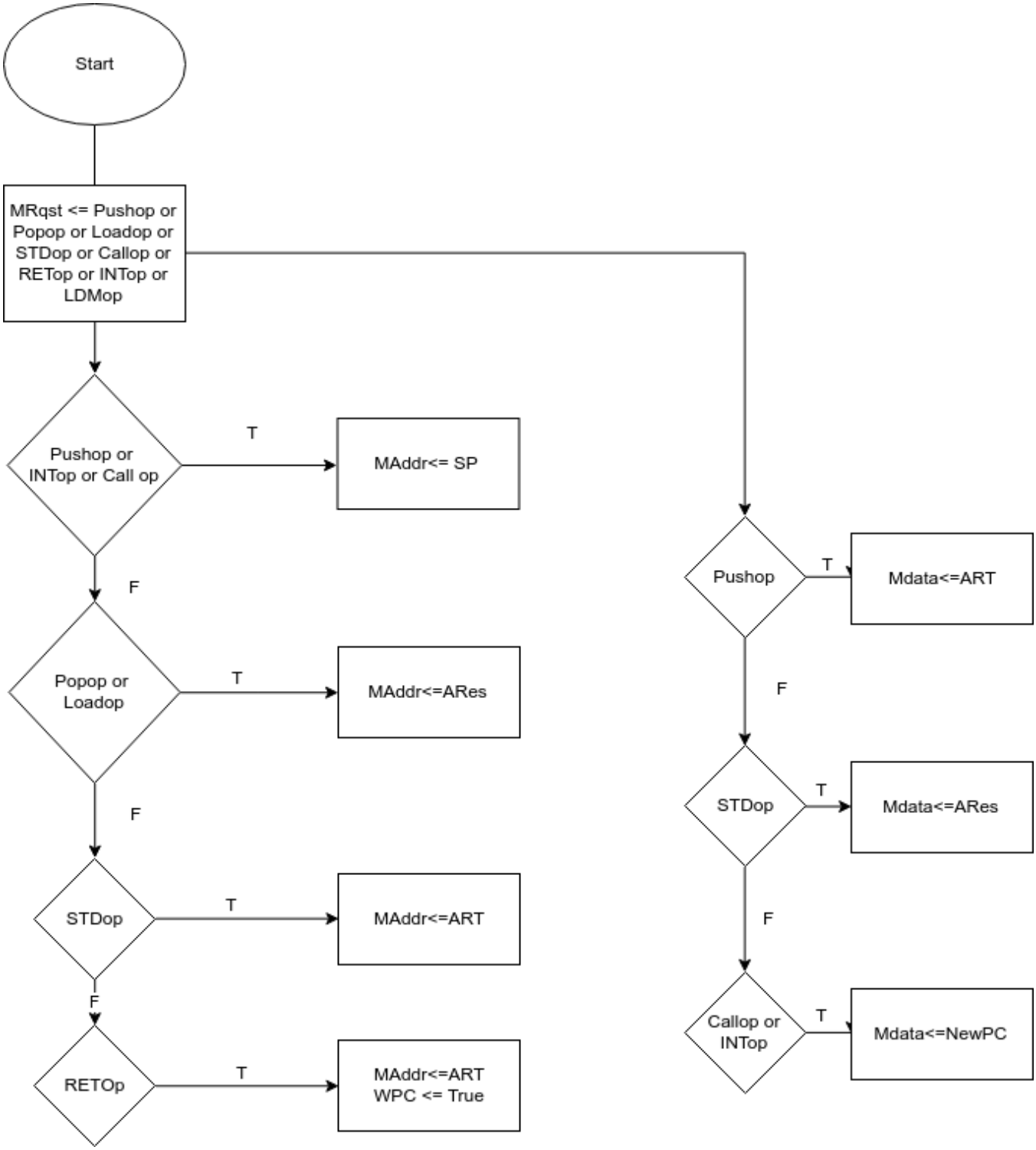


Figure 13: Memory Stage

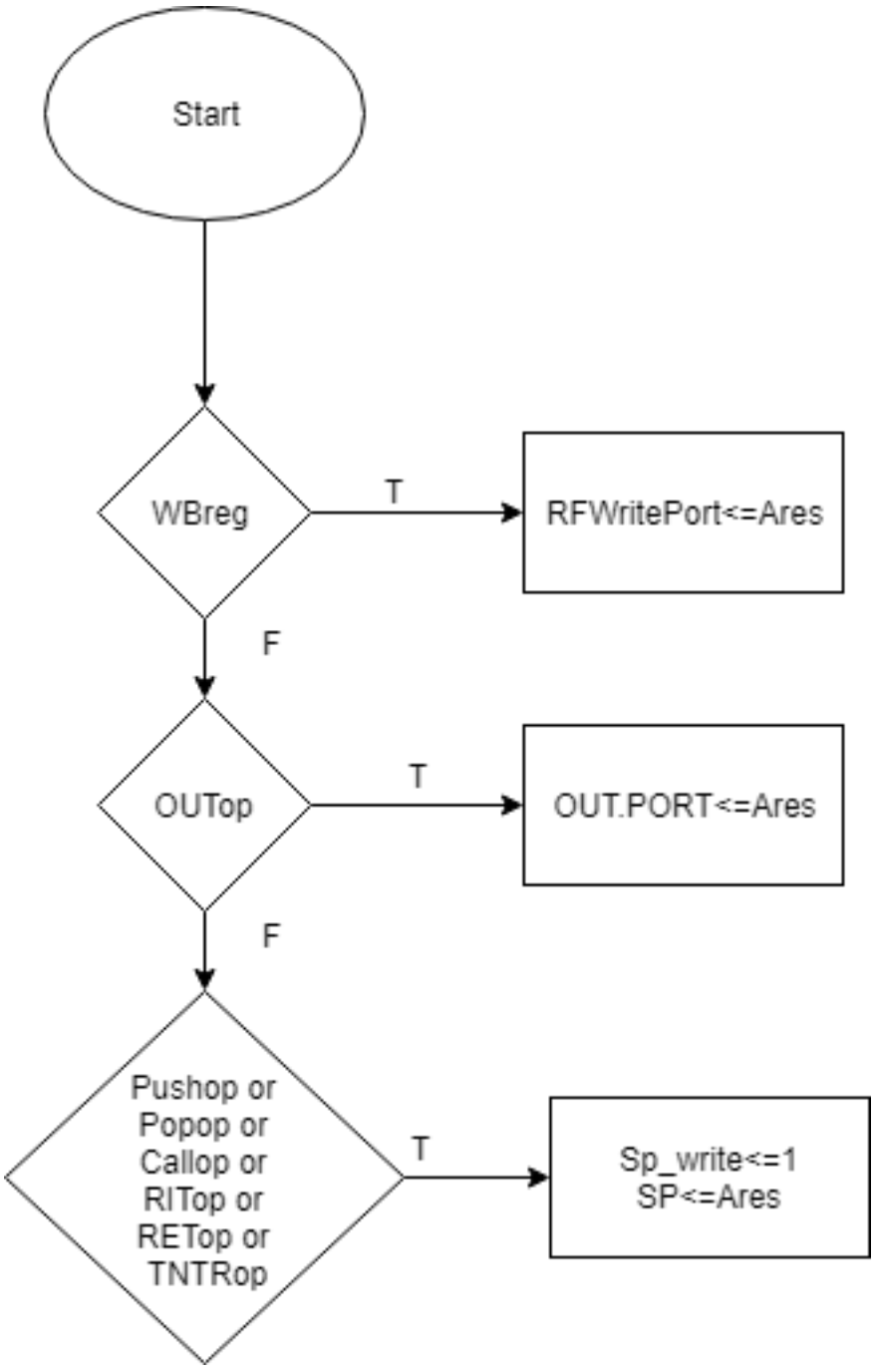


Figure 14: Write Back Stage

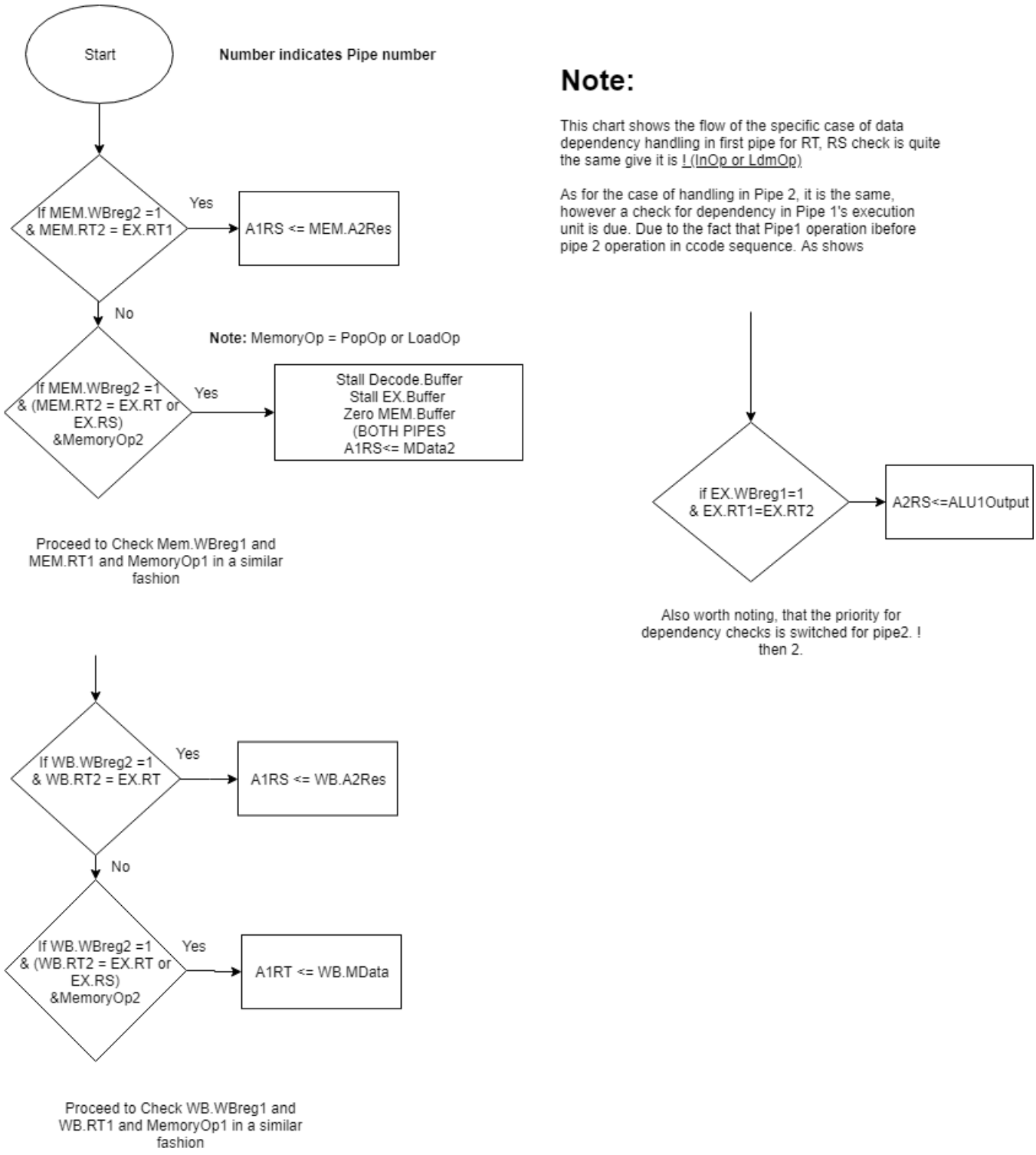


Figure 15: RAW Data Hazards - General

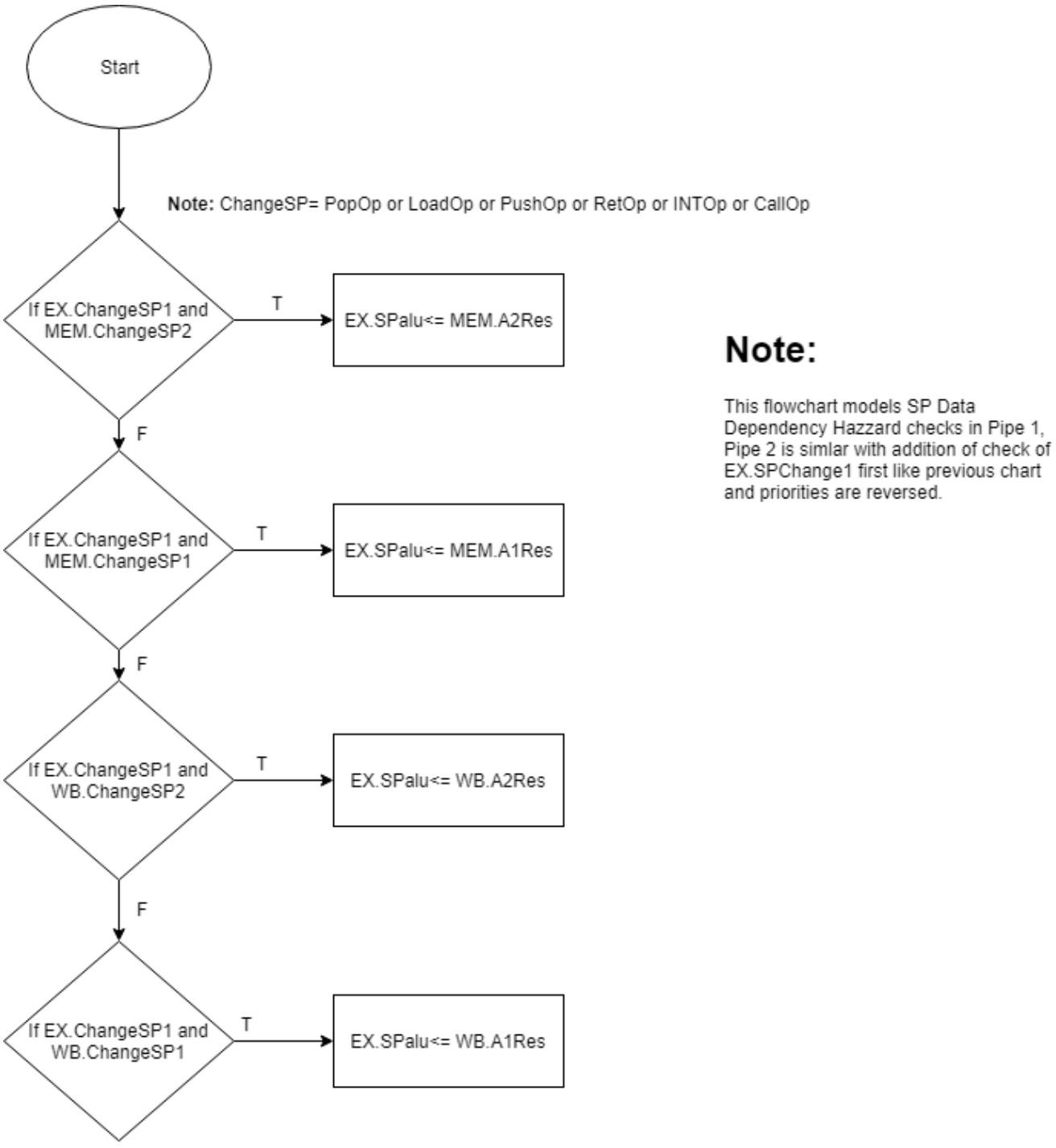


Figure 16: RAW Data Hazards - SP

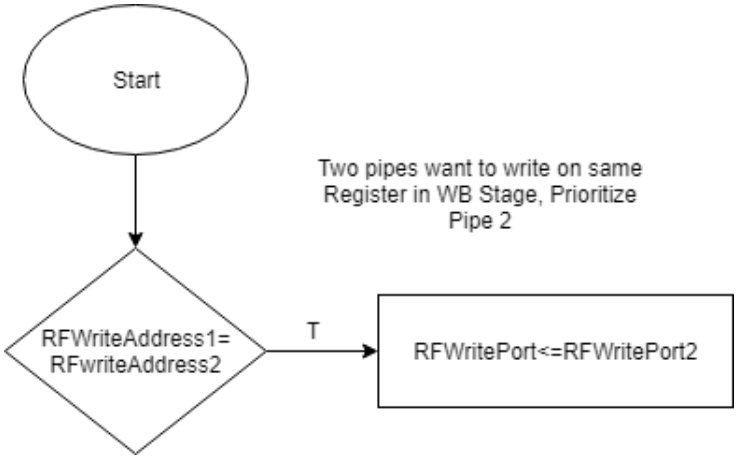


Figure 17: WB-WB Hazard

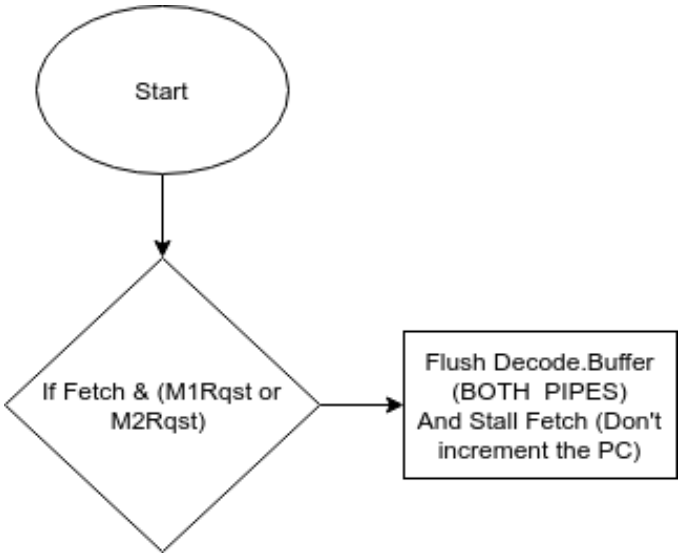


Figure 18: Fetch-MEM and MEM-MEM Hazards

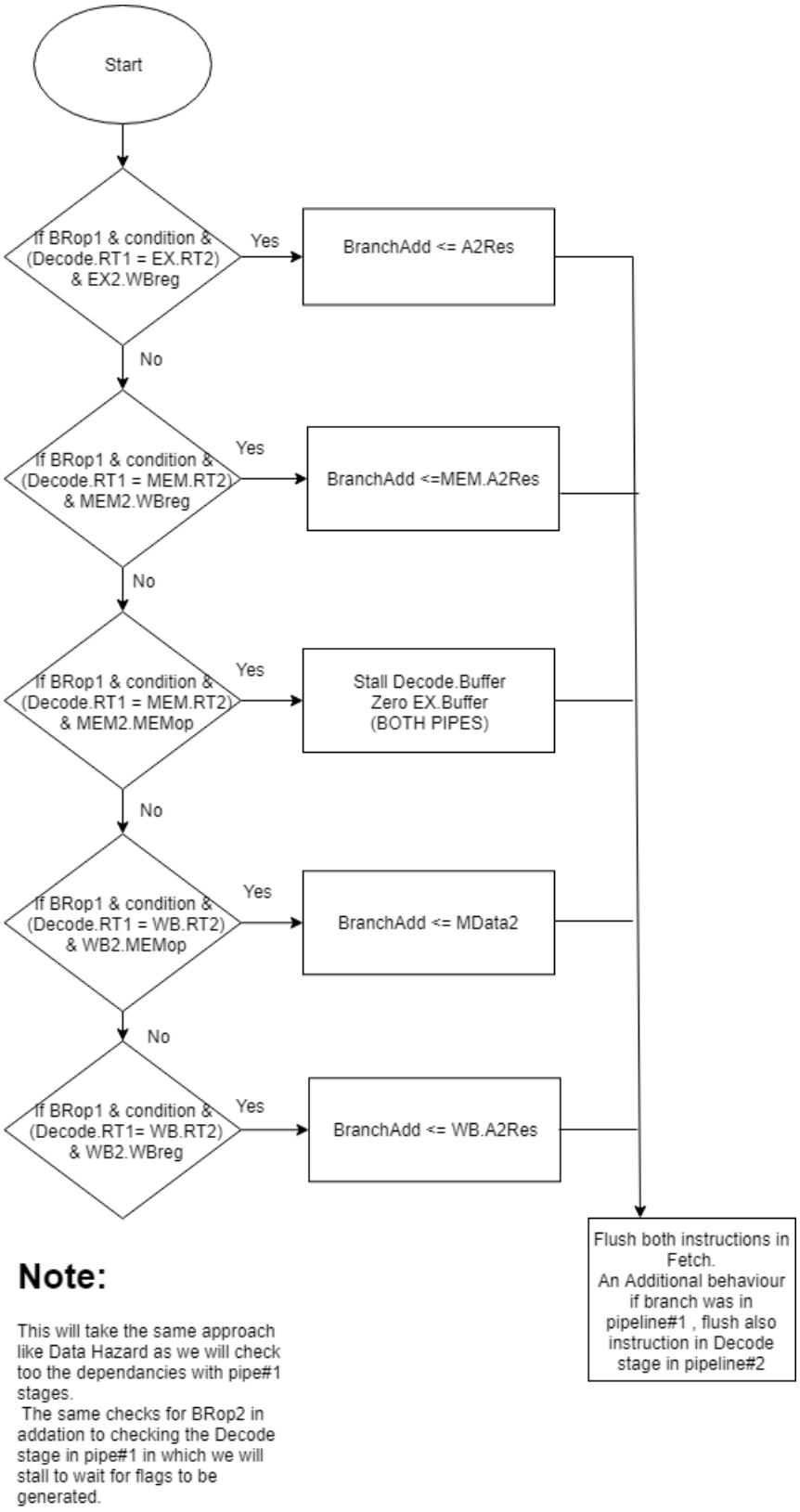
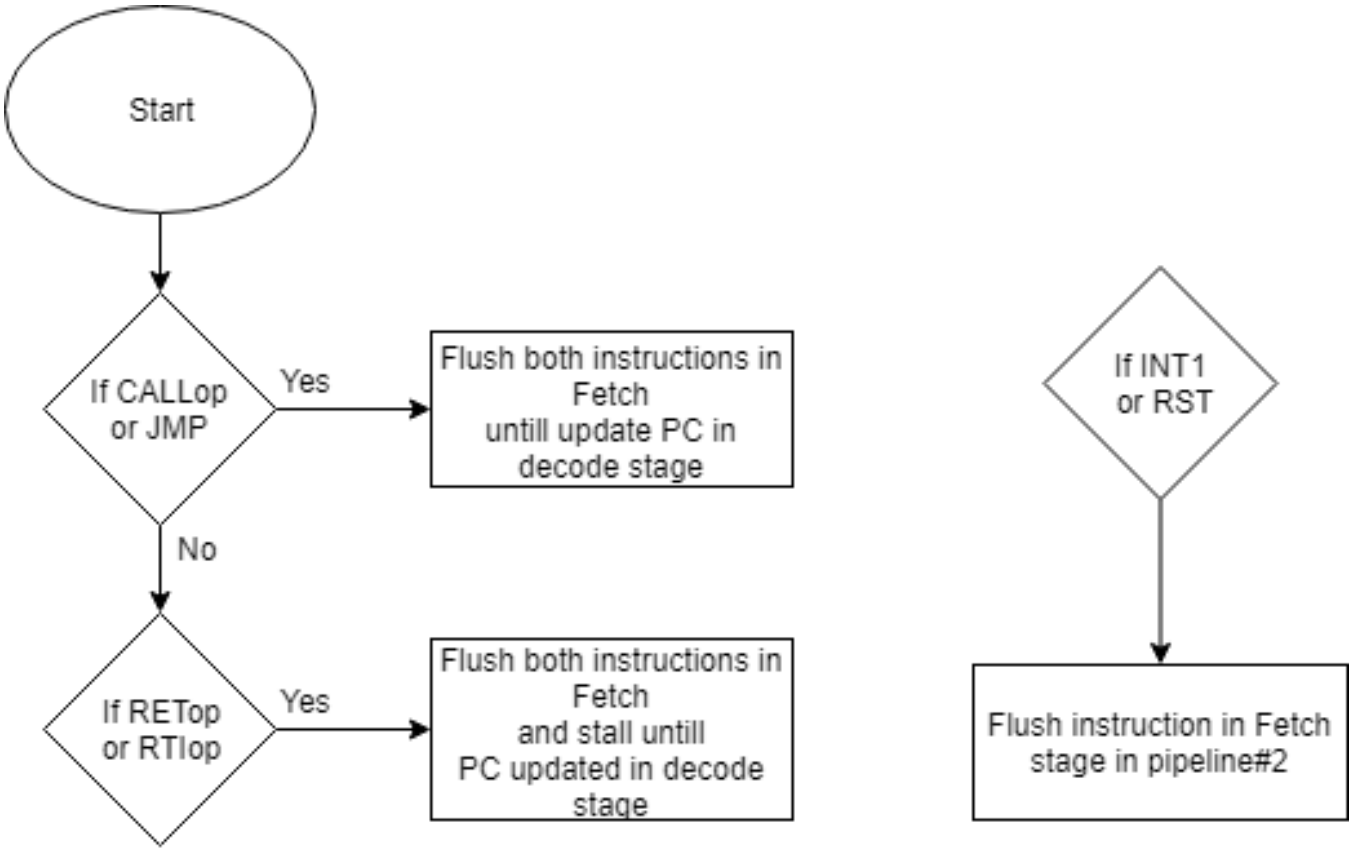


Figure 19: RAW for RT in Branch Instruction



Note:

In case this operation was in pipeline#1, then flush also instruction in Decode stage in pipeline#2.

Figure 20: Control Hazard - CALL/JMP/RET/RTI