# CVA6: An application class RISC-V CPU core

Prepared by :- K Charan, B Tech (ECE)

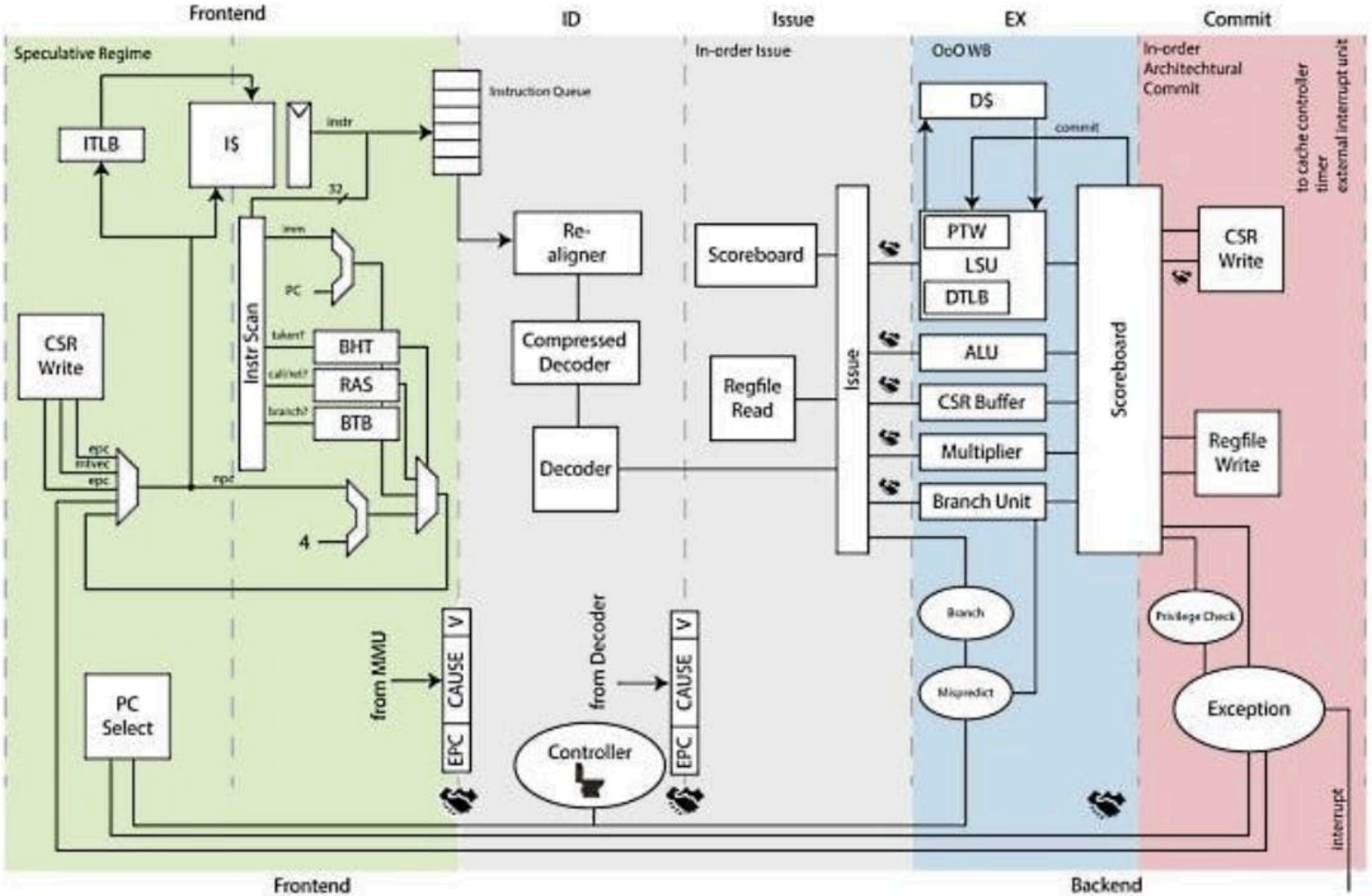
Date :- 01.11.2024

```
Under the supervision of Prof Vikramkumar Pudi, from EE dept. at IIT Tiruapti
```

## Introduction to CVA6

CVA6 (formerly known as Ariane) is an open-source, 64-bit application class RISC-V processor core. It is designed to be a high-performance, energy-efficient processor suitable for various applications.

# CVA6 Processor Desgin Block Diagram



# Key Components:

1. PC Generation: Generates the next Program Counter (PC).
2. Instruction Fetch (IF): Fetches instructions from memory.
3. Instruction Decode (ID): Decodes fetched instructions.
4. Issue Stage: Issues instructions to execution units.
5. Execute Stage (EX): Executes instructions using ALU, LSU, etc.
6. Commit Stage: Updates architectural state and manages exceptions.

# Detailed Component Functions

```
1. PC Generation Stage:

    - Generates the next instruction address.
    - Communicates with IF using handshake signals.

2. Instruction Fetch Stage (IF):

    - Requests instructions based on the PC.
    - Translates addresses via Memory Management Unit (MMU).

3. Instruction Decode Stage (ID):

    - Extracts and decodes instructions for processing.

4. Issue Stage:

    - Issues decoded instructions to functional units like ALU, CSR, LSU.

5. Execute Stage (EX):

    - Contains functional units that perform arithmetic and logic operations.

6. Commit Stage:

    - Updates registers and handles exceptions.
```

# What is CVA6?

1. A six-stage, single-issue, in-order CPU.
2. Implements the 64-bit RISC-V instruction set.
3. Supports three privilege levels: Machine (M), Supervisor (S), User (U).
4. Designed for running Linux and other Unix-like operating systems.

# RISC-V Instruction Sets Implemented in CVA6

1. RV64I (64-bit Integer)
2. RV64M (Multiplication and Division)
3. RV64A (Atomic Operations):
   i. Provides atomic read-modify-write operations.
   ii.Essential for multi-threaded programming and synchronization in concurrent environments.
4. RV64C (Compressed Instructions)
5. RV64F (Single-Precision Floating Point)
6. RV64D (Double-Precision Floating Point)
7. RV64B (Bit Manipulation)
8. RV64V (Vector Operations)

**This comparison highlights key features including architecture, pipeline stages, supported instruction sets, and performance characteristics.**

| Feature/Processor | CVA6 | Ibex | PULP | Rocket Core |
|---|---|---|---|---|
| Architecture Type | RISC-V | RISC-V | RISC-V | RISC-V |
| Pipeline Stages | 6 stages | 5 stages | 5 stages | 5 stages (out-of-order) |
| Issue Type | Single-issue, in-order | Single-issue, in-order | Single-issue, in-order | Out-of-order |
| Instruction Set | RV64I, RV64M, RV64A, RV64C | RV32I, RV32M | RV32I, RV32M | RV64I, RV64M |
| Privilege Levels | M (Machine), S (Supervisor), U (User) | U (User) only | M (Machine), U (User) | M (Machine), S (Supervisor), U (User) |
| Floating Point Support | Yes (RV64F/ RV64D) | No | Yes (RV32F/ RV32D) | Yes (RV64F/RV64D) |
| Atomic Operations | Yes (RV64A) | No | Yes (RV32A) | Yes (RV64A) |
| Compressed Instructions | Yes (RV64C) | No | No | Yes (RV64C) |

# Contd..

| | | | | |
|---|---|---|---|---|
| **Target Applications** | General-purpose computing | Embedded applications | Low-power IoT | High-performance computing |
| **Cache Architecture** | L1 instruction and data caches; TLBs for MMU support | No cache support | L1 cache support | L1 instruction and data caches; TLBs for MMU support |
| **Development Focus** | Open-source with production quality; Linux-capable | Lightweight and small footprint; educational use only | Low power and energy efficiency focus | High performance with out-of-order execution |
| **Configuration Options** | Highly parameterizable | Limited configurability | Configurable for power/ performance trade-offs | Configurable but complex due to out-of-order design |

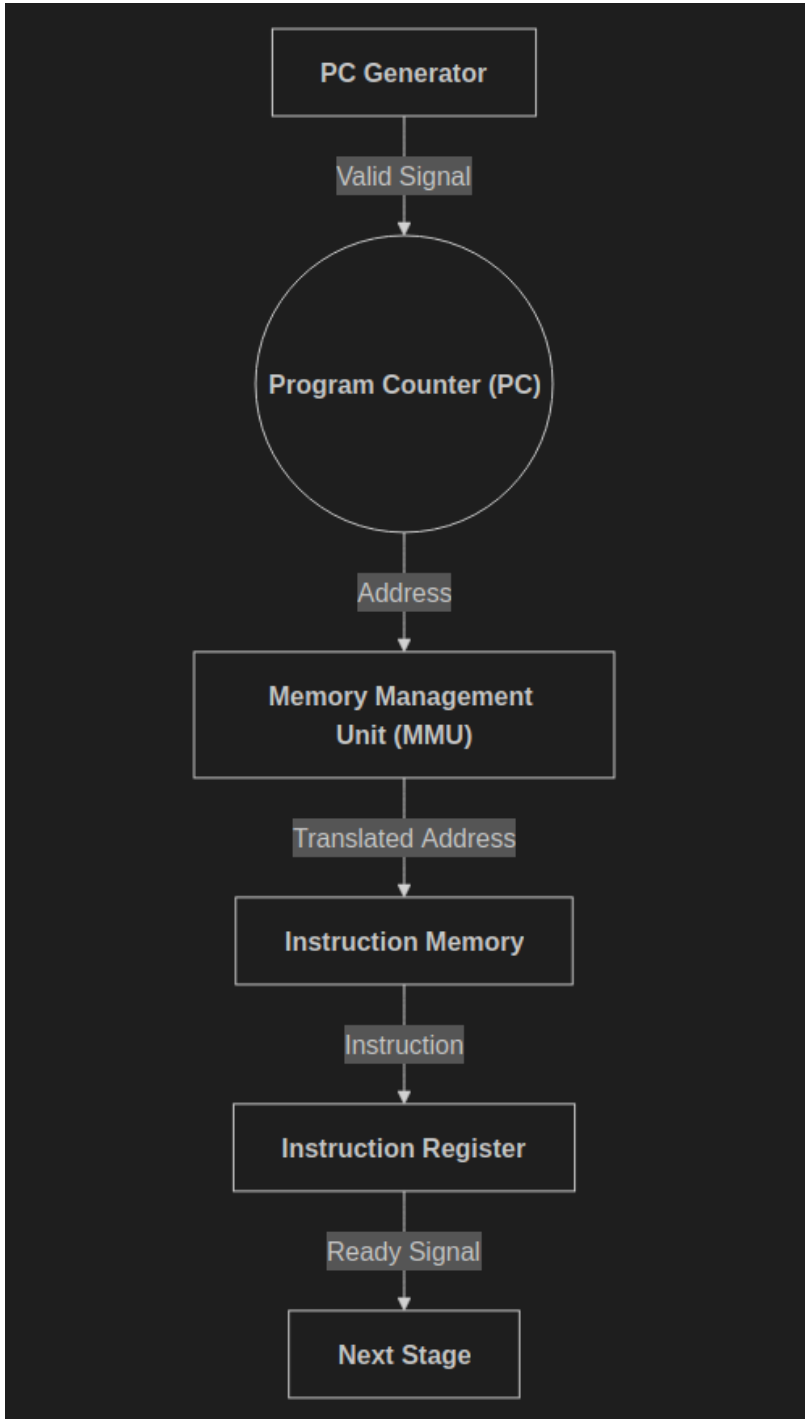
# Instruction Fetch (IF) Stage Overview

## Functionality:

```
i. Retrieves instructions from memory.
ii. Handles address translation requests via MMU.
```

## Control Signals:

```
i. Valid signal from PC Gen indicates readiness.
ii.Ready signal to indicate successful instruction fetch.
```


PC Generator
Valid Signal
Program Counter (PC)
Address
Memory Management Unit (MMU)
Translated Address
Instruction Memory
Instruction
Instruction Register
Ready Signal
Next Stage

# Chisel Source Code for IF Stage

```
import chisel3._
import chisel3.util._

class InstructionFetch extends Module {
    val io = IO(new Bundle {
        val pc = Input(UInt(64.W))             // Program Counter input
        val instruction = Output(UInt(32.W))   // Fetched instruction output
        val mmuRequest = Output(Bool())        // MMU request signal
        val ready = Input(Bool())              // Ready signal from MMU
    })

    // Memory initialization (for simulation purposes)
    val instructionMemory = Mem(1024, UInt(32.W))

    // Fetch instruction based on PC
    io.instruction := instructionMemory(io.pc)
    io.mmuRequest := true.B                    // Request MMU for address translation

    when(io.ready) {
        // Logic to handle when ready signal is asserted
    }
}
```

# Overview of the Instruction Decode Stage

- Purpose:

  i. The ID stage is responsible for decoding fetched instructions.
  ii. Prepares instructions for execution by identifying operation types and operands.

- Key Functions:

  i. Extracts opcode and operand information from the instruction.
  ii.Generates control signals for subsequent stages.
  Handles interrupts and exception management.

# Control Signals in the ID Stage

**Control Signals:**

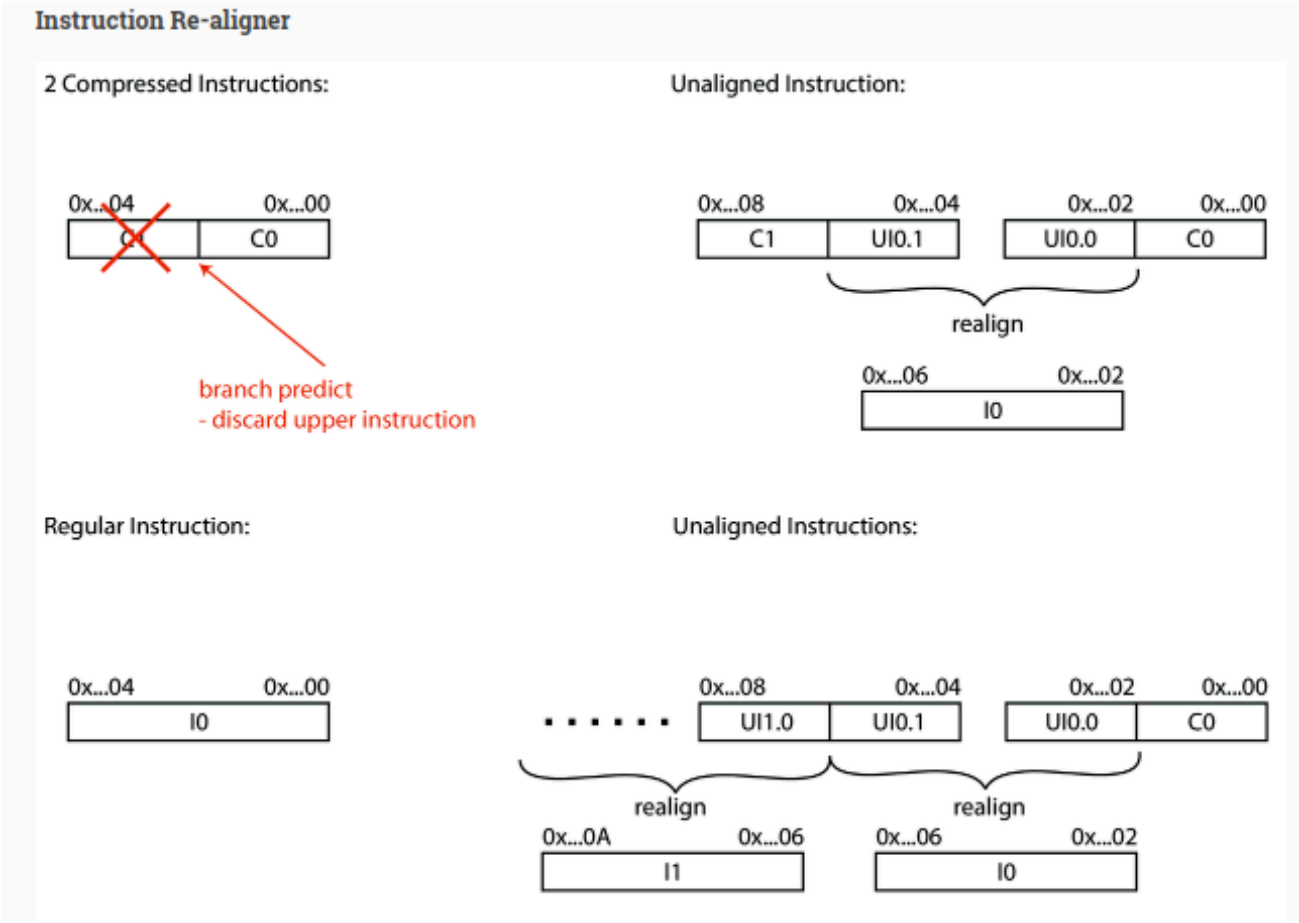
1. RegWrite: Indicates if a register write operation should occur.
2. MemRead: Signals that data should be read from memory.
3. MemWrite: Indicates a write operation to memory.
4. ALUSrc: Determines if the second ALU operand comes from a register or immediate value.
5. Branch: Indicates if the instruction is a branch instruction.

**Opcode Handling:**

i. The opcode determines the instruction type (e.g., R-type, I-type).
ii. Control signals are generated based on the opcode.

# Instruction Decode

Instruction decode is the first pipeline stage of the processor’s back-end. Its main purpose is to distill instructions from the data stream it gets from IF stage, decode them and send them to the issue stage.



# ID stage Flow chart

# ID Stage implementation Chisel Source code

```
class IDStage(implicit val p: Parameters) extends Module {
  val io = IO(new Bundle {
    // Input from IF stage
    val inst = Input(UInt(32.W))

    // Output to EX stage
    val ctrl_signals = Output(new ControlSignals)
    val rs1_data = Output(UInt(xLen.W))
    val rs2_data = Output(UInt(xLen.W))
  })

  // Decode instruction
  val decoder = Module(new Decoder)
  decoder.io.inst := io.inst
  val ctrl_signals = decoder.io.ctrl_signals

  // Read register file
  val regfile = Module(new RegisterFile)
  regfile.io.rs1_addr := decoder.io.rs1_addr
  regfile.io.rs2_addr := decoder.io.rs2_addr
  io.rs1_data := regfile.io.rs1_data
  io.rs2_data := regfile.io.rs2_data

  // Handle exceptions and interrupts
  val exception_unit = Module(new ExceptionUnit)
  exception_unit.io.inst := io.inst
  exception_unit.io.ctrl_signals := ctrl_signals
  // ...

  io.ctrl_signals := exception_unit.io.ctrl_signals
}
```

# interaction of CSR, BTB, BHT, and RAS components in control signal processing from IF -> ID Stage

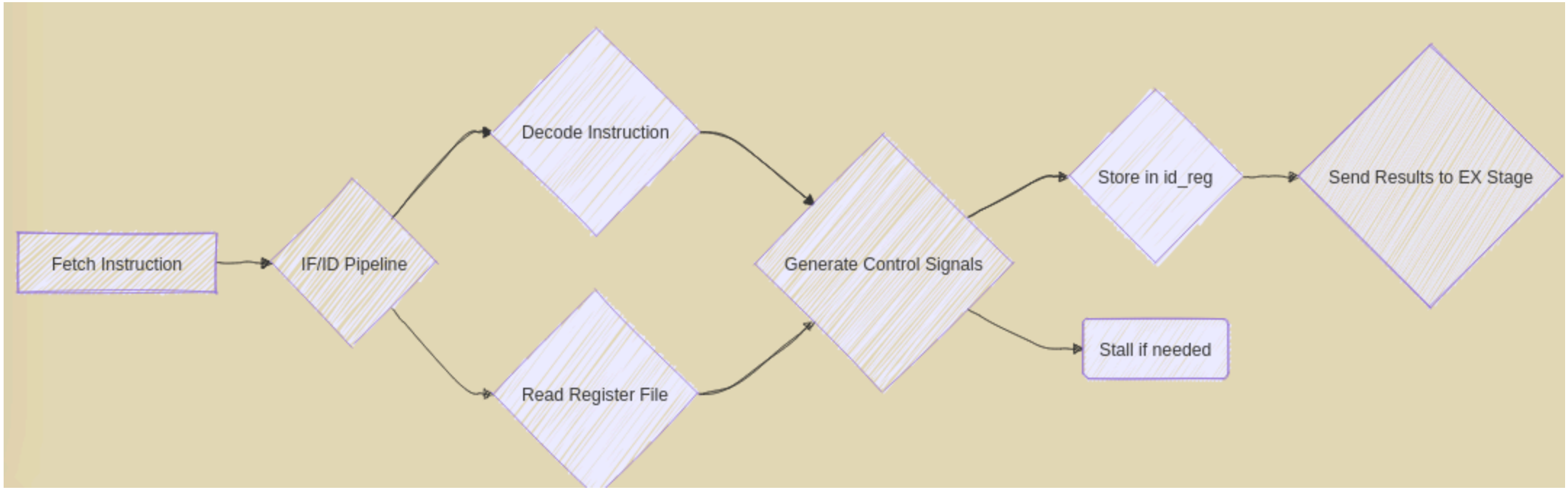
## Opcodes for these control signal components typically vary by processor architecture. However, here are some general examples:

```
; CSR operations
csrr rd, csr    ; Read CSR
csrw csr, rs1   ; Write CSR
csrwi csr, imm  ; Write immediate to CSR

; Branch operations (affecting BTB and BHT)
beq rs1, rs2, offset  ; Branch if equal
bne rs1, rs2, offset  ; Branch if not equal
jal rd, offset        ; Jump and link (function call)

; Return (affecting RAS)
ret                   ; Return from subroutine
```

# Chisel implementation of the pipelined ID stage:

```
class PipelinedIDStage(implicit val p: Parameters) extends Module {
  val io = IO(new Bundle {
    // ... (input and output ports)
  })

  // Pipeline registers
  val id_reg = RegInit(0.U(32.W))

  // IF/ID Pipeline Stage
  val if_id = Module(new IFIDPipelineStage)
  if_id.io.inst_in := io.inst
  id_reg := if_id.io.inst_out

  // RF/CSG Pipeline Stage
  val rf_csg = Module(new RFCSGPipelineStage)
  rf_csg.io.inst := id_reg
  io.ctrl_signals := rf_csg.io.ctrl_signals
  io.rs1_data := rf_csg.io.rs1_data
  io.rs2_data := rf_csg.io.rs2_data
}

class IFIDPipelineStage extends Module {
  val io = IO(new Bundle {
    val inst_in = Input(UInt(32.W))
    val inst_out = Output(UInt(32.W))
  })

  io.inst_out := io.inst_in
}

class RFCSGPipelineStage extends Module {
  val io = IO(new Bundle {
    val inst = Input(UInt(32.W))
    val ctrl_signals = Output(new ControlSignals)
    val rs1_data = Output(UInt(xLen.W))
    val rs2_data = Output(UInt(xLen.W))
  })

  // ... (decode instruction, read register file, generate control signals)
}
```

# Flow chart


Fetch Instruction
IF/ID Pipeline
Decode Instruction
Read Register File
Generate Control Signals
Store in id_reg
Stall if needed
Send Results to EX Stage

# Thank You!