1. Pipelining
   1. RISC-V standard pipe (C-26 – C-29)
      1. 5 stage model
      2. Fetch (IF)
         1. Refers to PC which has address of next instruction to execute
         2. Gets the instruction from the PC address into the instruction register (IR)
         3. Increment the PC by 4 which the new program counter (NPC) holds
      3. Decode (ID) [also could be RO or GO]
         1. Looks at the IR to decode
         2. Determine opcode
         3. Get source 1 ID
         4. Get source 2 ID
         5. Determine destination ID
         6. Sign-extend immediate (lower 16 bits of the IR)
         7. Can do all of the above in parallel since RISC V uses fixed length encoding
         8. Stores the outputs of the general-purpose register into 2 temporary registers A & B and stores the immediate into the temporary register Imm
         9. Stores need a separate sign-extension since immediate is split into 2 pieces
      4. Execution (EX)
         1. Operates on the operands prepared in ID with one of these functions depending on instruction type
            1. Memory reference: ALU adds the operands to form the effective address and places the result into the register ALUOutput (ALUOutput = A + Imm)
            2. Reg-Reg ALU: ALU performs the operation specified by the function code (func3 and func7) on the value in register A and on the value in register B. The result is placed into temporary register ALUOutput (ALUOutput = A func B)
            3. Reg-Imm ALU: ALU performs the operation specified by the opcode on the value in register A and on the value in register A and on the value in register Imm. The result is placed into temporary register ALUOutput. (ALUOutput = A op Imm)
            4. Branch: ALU adds the NPC to the sign-extended immediate value in Imm, which is shifted left by 2 bits to create a word offset, to compute the address of the branch target. Reg A is checked to determine if a branch is taken by comparing to Reg B (only equal is considered here).
         2. Load-store architecture of RISC V means the effective address and execution cycles can be combined into a single clock cycle.
      5. Memory Access (MEM)
         1. The PC is updated for all instructions (PC = NPC)
         2. Memory reference:
            1. Access memory if needed. If a load, data returns from memory and placed in the LMD (Load memory data) register.
            2. If a store, data from B reg. is written into memory
            3. In both cases, the address used is the one computed the prior cycle and stored in ALUOutput
         3. Branch: if the instruction branches, the PC is replaced with the branch destination address in the register ALUOutput
      6. Write-back (WB)
         1. Reg-reg or reg-imm: Regs[rd] = ALUOutput
         2. Load: Regs[rd] = LMD
         3. Write the result into the register file whether it comes from the memeory system (LMD) or ALU (ALUOutput)
   2. NPC and added within-stage registers
      1. NPC = new program counter and holds the next sequential PC. Need at least 4 since each must retain starting point
      2. IR = instruction register which holds the instruction from the PC address fetched
      3. A & B = temporary registers to hold operands from the decode stage
      4. Imm = stores the sign-extended IR from the decode stage
   3. Pipeline latches (C-30 – C-32)
      1. Pipeline latches are the registers between each pipeline stage.
      2. These registers serve to convey values and control information from one stage to the next.
      3. IF/ID, ID/EX, EX/MEM, MEM/WB, and the PC which sits before the IF stage are all pipeline latches.
   4. 1596
      1. These latches can read their inputs, copy them to the outputs which then remain constant throughout the rest of the cycle
   5. Stalling for hazards
      1. Hazards make it necessary to stall the pipeline.
      2. Certain instructions will be allowed to proceed while others get delayed.
      3. No new instructions are fetched during a stall.
   6. Pipeline Timing Diagrams
      1. See homework
   7. Clock skew effects
      1. Clock skew is the maximum delay when the clock arrives at any two registers.
      2. When the clock doesn’t reach parts of CPU at the same time
      3. Contributes to the lower limit on the clock cycle
      4. Once the clock cycle is as small as the sum of the clock skew and latch overhead, no further pipelining is useful since no time is left in the cycle for useful work.
   8. Register delay effects
      1. Registers aren’t evenly distributed. May take time for register pulls or pushes
      2. Adds to overhead to setup and reaching a stable register before anything else can occur.
2. Hazards
   1. Hazards are a major barrier in pipelining that if not addressed results in incorrect execution
   2. Stalls are foolproof solution to the hazard problem
   3. Structural hazards
      1. Resource related conflict
      2. When the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.
   4. Data hazards
      1. When an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
      2. These are our data dependencies
   5. Control hazards
      1. Arise from the pipelining of branches and other instructions that change the PC
3. Data Dependencies
   1. True Data Dependencies
      1. An instruction that consumes a data item produced by another instruction exhibits a data dependency.
      2. Also indirect/transitive: when instruction j is data-dependent on instruction k, and instruction k is data-dependent on instruction i
         1. Fld f0, 0(x1)
         2. Fadd.d f4, f0, f2
         3. Fsd f4, 0(x1)
   2. False/Name Data Dependencies
      1. Something that occurs or exists when an instruction produces a value for the same item that was used by a prior instruction without data flowing between instructions
      2. Antidependency: When instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved to ensure that i reads the correct value.
         1. Fsd f4, 0(x1)
         2. Addi x1, x1, -8
      3. Output dependency: when instruction I and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j.
   3. Transitive Dependencies
      1. When a chain is formed in instruction reliance. See TDD for definition
         1. Add x1, x2, x3
         2. Sub x4, x5, x1
         3. Or x6, x5, x4
         4. Two direct and 1 indirect/transitive. X4 uses x1 in calculation so depends on the add x1
4. Adapting Pipe to Stall
   1. Detection
      1. Able to detect a true data dependence, necessary precondition for a data hazard.
   2. Cost of Stalls
      1. Reduces speedup
      2. Bubbles
         1. 3 stalls for true data dependence
         2. 4 stalls for a control instruction
      3. S = CPI unpipelined / 1 + stall cyles avg
5. Stall-Avoidance
   1. Register File Forwarding
      1. Register file forwarding is when you have all the writes early in the clock cycle and all the reads late in the clock cycle. This allows you to overlap dependent pipeline stages that would normally require one more pipeline stall cycle.
      2. There is minimal cost to this, just a structural convention.
   2. Full Forwarding/Bypassing (short circuiting)
      1. Bypassing is where you wire together stage of the pipe to get the register values updated from a different instructions execution stage that you need in another instruction executing earlier. This prevents you from having to stall though the Write Back stage of an instruction that a future instruction depends on.
      2. There is slightly more cost to this, you have to create connections between stages of the pipe that did not exist before
   3. Supporting Logic
   4. Benefits
      1. Forwarding through the register file allows you to save 1 CC from a true data dependency
      2. Bypassing allows you to save 3 CCs from a true data dependency
6. Delayed Branch
   1. The instruction immediately following the control instruction excecutes but requires a “safe” instruction
      1. Instruction in fall-through
      2. Instruction at target
      3. Instruction before branch
      4. noop
7. Simple (Static) Predictive Branching
   1. Predict taken: assume branch is taken bne-T
   2. Predict not-taken: assume branch is not taken bne-NT
8. Varying Latency Pipes
   1. Pipe with different lengths
      1. Div – 24 CCs
      2. Int add – 1 CC
      3. Float/int multiply – 7 CCs
      4. FP add – 4 CCs
   2. Costs
      1. Requires significantly more infrastructure
         1. Pipeline registers
         2. Clock Skew resolution
         3. Larger die space
      2. Introduces new structural hazards.
         1. WAR – Write after Read - incorrect order of write instructions results in an earlier value written to a variable overwriting a later value written to the same variable. Only in pipelines with multiple stages, and variable latency or the ability to stall only some instructions.
   3. Benefits
      1. Can allow you to have faster CC time. Due to the longer instructions being able to be accomplished in their own pipe the fast instructions can be accomplished more quickly and the overall CPU cycle time can be faster.
9. **Interrupts/Exceptions:** Are handled by Interrupt Service Routines(ISR)/ Exception Service Routines(ESR).

**ISR/ESR:** This is some other code that exists somewhere, usually part of the OS and handle that interrupt that code has to be executed.

# Work arounds:

## History File:

* + **Function**:
    - The history file serves as a repository for storing architectural state information associated with instructions that are currently in-flight within the pipeline.
    - When an instruction enters the pipeline, its relevant architectural state, such as register values and memory addresses, is stored in the history file. This ensures that the processor can track the progress and state of each instruction as it moves through the pipeline stages.

## Checkpoint Mechanism:

* + - The history file acts as a checkpoint mechanism, preserving the architectural state of instructions being processed in the pipeline.
    - In the event of an exception or interrupt, the contents of the history file are crucial for restoring the architectural state to the point just before the exception or interrupt occurred.
    - By reverting to this checkpoint, the processor can effectively resume execution from the interrupted point without losing progress or corrupting the program state.

## Future File:

* + **Function**:
    - The future file, also referred to as the reorder buffer or reservation station, plays a central role in managing speculative and out-of-order execution of instructions within the pipeline.
    - When instructions are fetched and decoded, they are initially placed in the future file before being dispatched to the execution units for processing.

## Speculative Execution Management:

* + - The future file maintains the original program order of instructions while allowing for speculative execution.
    - Instructions may be executed out of order to exploit parallelism and optimize performance. The future file keeps track of the correct program order to ensure instructions are eventually retired in the correct sequence.

## Exception and Interrupt Handling:

* + - During exception or interrupt events, the future file may need to be carefully managed to maintain program correctness and consistency.
    - Instructions that have been speculatively executed beyond the point of the exception or interrupt may need to be discarded or invalidated to prevent incorrect results or corruption of program state.
    - The pipeline may need to be reset to a known state, and the contents of the future file may need to be flushed or rolled back to ensure proper handling of the exception or interrupt.

In summary, the history file and future file are essential components of pipelined processors, each serving distinct functions in managing instruction execution and handling exceptions and interrupts. The history file preserves architectural state information for instructions in-flight, while the future file manages speculative execution and out-of-order execution, requiring careful handling during exception or interrupt events to maintain program correctness and consistency.

## Burden TRAP Routine:

ISR must itself handle pipeline issues resulting from exception.

## The Barrier of Exceptions:

-Page Faults

-Arithmetic Exceptions

-Cache Miss

### Classes of Exceptions:

* 1. ***Coerced Vs User-Requested***

User requested exceptions are the one that programmer requested.

I need to access the semaphore, there’s an OS routine that handles the semaphore, I need to access that. I wanna see is my input actually there, so I’m requesting is my input present there, I’m gonna ask OS to do that for me. These are user requested.

Coerced is not one programmer asked for, but that aroused itself.

Some times they are external to my code and sometimes they are not. If I run into Page Fault, I didn’t requested for page fault, I just doing load and store and page fault happened, that is an example of Coerced Exception. It’s not explicitly requested by this code.

### Synchronous Vs Non-Synchronous:

Synchronous Exceptions, one that always happens at a given place. divw x1, x0, x0 (always divide by zero), so always happens.

Non-Synchronous Exceptions, they might sometimes occur and they might not occur sometimes.

divw x1, x0, x2 (x2 may be sometimes zero, and sometimes maynot), so don’t always happens.

### Within Vs Between:

An exception occurs within a given stage. divw x1, x0, x10(here the exception actually occurs in the EX stage of the pipe, it occurs during a stage).

An exception occurs between two instructions.

disable.

### User Makable Vs Non-Maskable:

User Makable is User can choose not to listen to, user can enable or Non-maskable is User cannot disable it.

### Terminate Vs Resume:

The terminate class exception is the one that the program crashes from, it actually terminates the program, this is an exception that we cannot be recoverable.

There are some page faults actually cause termination, If I ask for memory location which does not exist in my page table that’s a terminating exception.

Resume means that the program can continue, program can pickup.

Page Faults are usually resume class exceptions, we want OS to check the page is and bring the page in and resolve the page fault.

# Implementing Exception Support:

***STEP 1:*** Once we realize there is an exception the next instruction fetch IF, has to become a TRAP fetch instruction. That is at somewhere somehow the CPU system needs to realize what’s the class of the exception is, ones it figures out what the class of the exception was, it needs to figure out which program/subprogram/subroutine is associated with that, then we execute a TRAP instruction where the argument of the TRAP instruction is, address of that code to execute.

**Eg:** Let’s say we got a page fault, there’s a page fault handling routine in the OS, then the next instruction fetch has to fetch that instruction which is a jump to that, it’s special jump we call it a TRAP. TRAP is a special jump instruction that fetches or go to the Interrupt Service Routines.

## Detailed explanation: source ChatGPT

Let's consider an example where a program encounters a page fault exception and how it's handled:

## Program Execution:

* + Imagine a program running on a computer system, performing various tasks that involve accessing memory.

## Page Fault Exception:

* + During the execution of the program, let's say the CPU encounters a page fault exception. This occurs when the program tries to access a memory page that is not currently in physical RAM but is stored on disk.

## Exception Handling:

* + When the CPU detects the page fault exception, it triggers an interrupt to transfer control to the operating system's exception handling mechanism.

## Exception Type Identification:

* + The operating system's exception handler determines the type of exception, which in this case is a page fault.

## Page Fault Handler:

* + The operating system's page fault handler identifies the program or process associated with the page fault and determines which memory page needs to be brought into RAM from disk.

## Trap Fetch Instruction:

* + To handle the page fault, the operating system replaces the next instruction fetch (IF) with a trap fetch instruction. This instruction instructs the CPU to jump to a specific address in memory where the code to handle the page fault resides.

## Executing TRAP Instruction:

* + The CPU executes the trap instruction, jumping to the address specified by the page fault handler. This address points to the code responsible for handling page faults.

## Handling Page Fault:

* + The code at the trap address performs tasks such as:
  + Fetching the required memory page from disk into RAM.
  + Updating the page tables to reflect the new location of the page in RAM.
  + Resuming the execution of the interrupted program from the instruction following the one that caused the page fault.

## Continued Execution:

* + With the required memory page now available in RAM, the program can continue its execution without encountering further page faults (assuming no additional memory accesses trigger page faults).

In summary, when a page fault exception occurs, the CPU transfers control to the operating system's page fault handler, which replaces the next instruction fetch with a trap instruction. This instruction directs the CPU to execute code specifically designed to handle page faults, allowing the program to continue execution once the required memory page is made available in RAM.

### STEP 2: Disable WRITES

The safest approach is to disable all writes, all instruction now that already in the pipeline are prevented from writing, they are prevented from writing to memory and registers.

Let’s assume we don’t have something like a history files or future files then the conventional approach is to disable writes.

NO REGISTER FILE/MEM WRITES ARE GOING TO OCCUR NOW FOR THE INSTRUCTIONS WHICH ARE ALREADY IN THE PIPE.

### STEP 3: Save the PC for the last instruction

That’s the last instruction that would have made it through the pipe, so that is not the currently executing instruction and not the either one before that, but some instruction that is already made it through the pipe and it’s in the WB phase of the pipe at this instance of time. We are gonna save that PC or that PC + 4, so that instruction and the next instruction will execute at some point in the future.

### STEP 4: Handle Exception

The instructions(where we jump to, the trap instructions mentioned in STEP 1), all of those instructions execute. They go through the pipe just everything go through the pipe, we have to handle exceptions what ever the body of code.

When we are done with the exception, we return from the trap we have to jump back where ever the PC is(STEP 3) and resume execution.

### STEP 5: RESUME EXCEPTION

We need to reload the program counter, and restore all the registers. It could be the ISR itself saves all the registers or when we return from the TRAP instructions that itself will restore the registers.

# Exception Handling Modes:

**Precise:** Code end behavior same regardless of if an exception.

**Non-Precise:** Code end behavior differs for certain exceptions.

# Multiple Exceptions:

## Resolution/Work-Around:

-Non-precise exception handling.

-Exception Status Vector(Special Register resides in the CPU storing all the details related to exception)

-Type of Exception

-Instruction

-Stage of the instruction

-[Time Stamp]

## Exception Handling:

When exception occurs first we record the details in ESV. We allow the pipeline to continue to execute.

In MEM or WB/ where ever the write happens, we check the ESV and then we execute the TRAP instruction based on time stamp(which ever is the earliest).

1. Loop unrolling
   1. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code.
   2. Can also be used to improve scheduling. It can eliminate the branch which allows the instructions from different iterations to be scheduled together.
   3. Can use techniques to reduce the burden loops. Think about a for loop
      1. For I = 1, N
      2. A(i) = B(i) + C(i)
      3. Has N branches and 3 N stalls
   4. If we change the for loop to include a little more data, we can reduce the stalls
      1. For I =1, N, i+2
      2. A(i) = B(i) + C(i)
      3. A(i+1) = B(i+1) + C(i+1)
      4. Has N/2 branches and 3/2 stalls.

Sure, here’s a brief overview of the topics you’ve listed:

1. **Pipelining**
   * **RISC-V standard pipe**: A five-stage pipeline architecture used in RISC-V processors. The stages are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB).
   * **NPC and added within-stage registers**: NPC (Next Program Counter) is used to hold the address of the next instruction. Within-stage registers are used to hold intermediate data between stages.
   * **Pipeline latches**: These are storage elements between pipeline stages that hold data from one stage until the next stage is ready to use it.
   * **Stalling for hazards**: A stall, or pipeline bubble, is introduced into the pipeline to prevent hazards (structural, data, control) from affecting the execution of instructions.
   * **Pipeline Timing Diagrams**: These diagrams are used to visualize the progression of instructions through the pipeline over time.
   * **Clock skew effects**: Clock skew can cause setup and hold time violations in pipelined systems, leading to incorrect operation.
   * **Register delay effects**: The time it takes for a signal to propagate from the input to the output of a register. This delay can affect the timing of the pipeline stages.
2. **Hazards**
   * **Structural hazards**: Occur when multiple instructions require the same hardware resource at the same time.
   * **Data hazards**: Occur when instructions that exhibit data dependence modify data in different stages of a pipeline.
   * **Control hazards**: Occur due to the delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).
3. **Data Dependencies**
   * **True Data Dependencies**: Occur when an instruction depends on the result of a previous instruction.
   * **False/Name Data Dependencies**: Occur when two or more instructions use the same register or memory location, but there’s no actual dependence between the instructions.
   * **Transitive Dependencies**: Occur when one instruction is dependent on a second instruction, and the second instruction is dependent on a third instruction.
4. **Adapting Pipe to Stall**
   * **Detection**: Identifying the conditions under which a stall should be inserted into the pipeline.
   * **Cost of Stalls**: The impact on performance due to the introduction of stalls in the pipeline.
5. **Stall-Avoidance**
   * **Register File Forwarding**: A technique used to avoid stalls by forwarding data directly from one pipeline stage to another.
   * **Full Forwarding/Bypassing**: A technique where the results are forwarded directly from any pipeline stage to either the decode stage or the execute stage.
   * **Supporting Logic**: The additional hardware logic required to implement forwarding/bypassing.
   * **Benefits**: The main benefit is the reduction in pipeline stalls, which leads to improved performance.
6. **Delayed Branch**: A technique used to deal with control hazards. The idea is to delay the effect of a branch instruction until we are sure the branch condition is known.
7. **Simple (Static) Predictive Branching**: A method of resolving branch hazards that assumes a particular outcome for the branch.
8. **Varying Latency Pipes**
   * **Costs**: The additional complexity in the hardware and control logic.
   * **Benefits**: The ability to handle instructions that have variable execution times.
   * **History Files**: Used to keep track of the past behavior of the pipeline.
   * **Future File**: Used to hold the state of the pipeline as instructions are completed.
   * **Trap Burdening**: The additional complexity of handling exceptions in a pipeline with varying latency.
9. **Exception/Interrupt Effects**: The impact of exceptions and interrupts on the pipeline, including the need to flush the pipeline and the potential for additional pipeline hazards.
10. **Loop unrolling**: A compiler optimization technique that eliminates the overhead of loop control instructions by replicating the body of the loop.

Remember, this is just a brief overview. Each of these topics is quite deep and may require further study to fully understand. Good luck with your studies! 📚