Pipelined MIPS CPU on FPGA – Design and Verification Report

ADVANCED CPU ARCHITECTURE AND HARDWARE ACCELERATORS LAB

361.1.4693

Ahseen Alazazma 324038132

Aram Khater 314813452

1. LAB overview:

In this lab, we implemented a pipelined MIPS CPU architecture supporting full data hazard detection and forwarding mechanisms. The design includes all five standard pipeline stages — IF, ID, EX, MEM, and WB — with additional logic to handle stalls, flushes, and control flow changes.

The CPU was first verified using ModelSim through the simulation of a custom hand-written assembly program, ensuring functional correctness and proper pipeline behavior. After successful simulation, the design was synthesized using Quartus and deployed on an FPGA. Final validation was performed using SignalTap to capture and inspect live execution, confirming correct memory access, instruction sequencing, and pipeline control under real hardware conditions.

2. Test program

To evaluate the functionality of our MIPS CPU, we ran several test programs covering arithmetic operations, memory access (load/store), branches, and jumps. One representative example is a matrix addition program written in C and compiled to MIPS assembly.

```c
#define M 4

void addMats(int Mat1[M][M], int Mat2[M][M], int resMat[M][M]){
     define it yourself …
}

void main(){ //int=32bit
     int Mat1[M][M]={{1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,16}};
     int Mat2[M][M]={{13,14,15,16},{9,10,11,12},{5,6,7,8},{1,2,3,4}};
     int resMat[M][M];

     addMats(Mat1,Mat2,resMat); // resMat = Mat1 + Mat2
}
```

Figure 1: C program for adding two matrices

And it’s version in assembly:

```
.data
# Define M = 4
M:              .word 4

# Matrices: 4x4 (row-major order)
Mat1:       .word 1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16
Mat2:       .word 13,14,15,16, 9,10,11,12, 5,6,7,8, 1,2,3,4
resMat:     .space 64       # 4x4 integers = 16 * 4 bytes = 64 bytes

.text
.globl main

main:
    # initialize base addresses
    la $t0, Mat1            # $t0 = base of Mat1
    la $t1, Mat2            # $t1 = base of Mat2
    la $t2, resMat          # $t2 = base of resMat

    li $t3, 16              # total number of elements (4x4 = 16)
    li $t4, 0               # i = loop counter

loop:
    beq $t4, $t3, finish  # if i == 16, go to infinite loop

    lw $t5, 0($t0)        # load Mat1[i]
    lw $t6, 0($t1)        # load Mat2[i]
    add $t7, $t5, $t6     # resMat[i] = Mat1[i] + Mat2[i]
    sw $t7, 0($t2)        # store result

    # increment pointers and index
    addi $t0, $t0, 4      # Mat1++
    addi $t1, $t1, 4      # Mat2++
    addi $t2, $t2, 4      # resMat++
    addi $t4, $t4, 1      # i++

    j loop                # repeat

# Infinite loop after completion
finish:
    j finish              # stay here forever
```

Figure 2: assembly program

3. Simulation Verification (ModelSim)

ModelSim simulation confirmed correct instruction sequencing and data propagation through the pipeline stages. We used waveform inspection to verify:

- Proper forwarding of values (e.g., add after lw)
- Correct stall insertion on load-use hazards
- Accurate memory writes at the correct cycle
- Instruction alignment across pipeline registers

The following show how the data flow in pipeline:

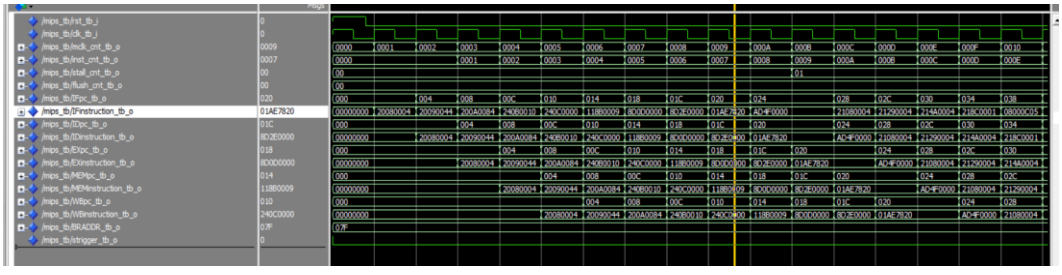


Figure 3: Pipeline flow

The following show correct handling when the branch taken the next fetched instruction is flushed and the flush counter increased by 1:

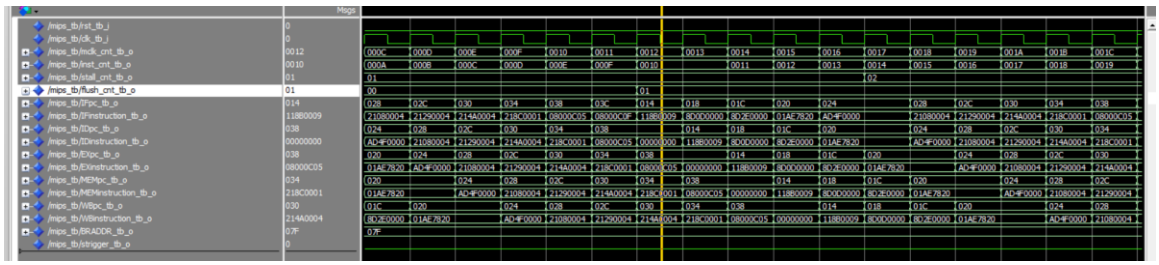


Figure 4: correct handling of incorrect instruction when branch taken.

The following show correct data-hazard handling when lw dependency needs stall and the stall counter increment by 1:

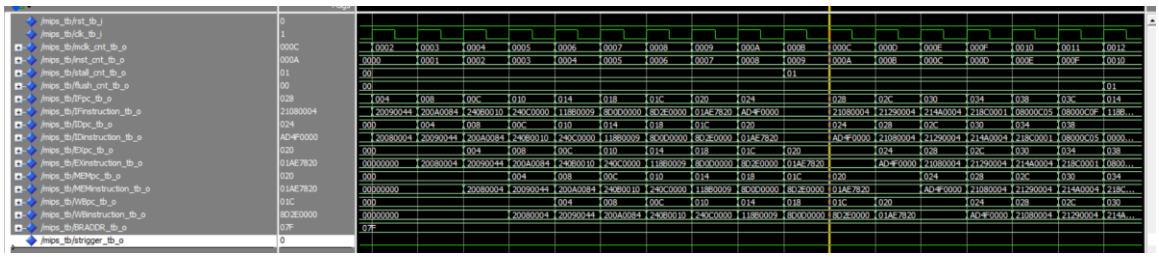


Figure 5: correct handling of data hazard (LW dependency)

The following show correct behavior when data hazard detected and the data can be forwarded:

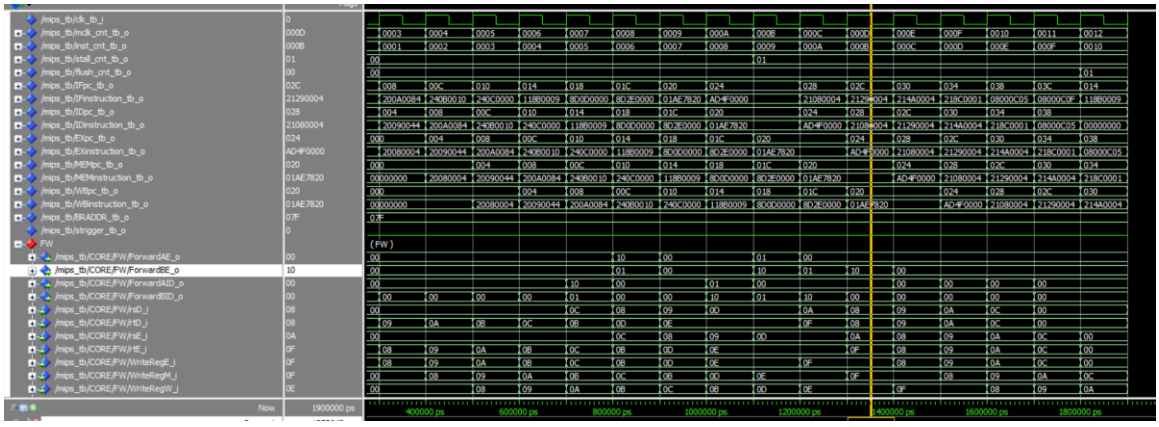


Figure 6: correct handling of data hazard (LW dependency) and forwarding from MEM stage.

The following show the breakpoint support of the mips CPU for debugging convenience:

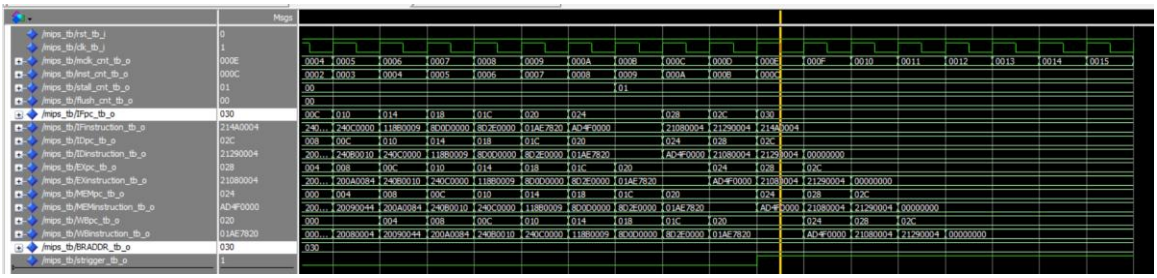


Figure 7: breakpoint support

The following show the content data memory after completing the program, which match the theoretical behavior:

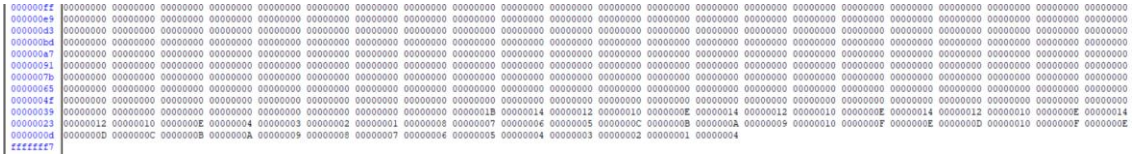


Figure 8: Data memory content (the matrices and the summed matrix)

4. Hardware Validation (Quartus + SignalTap)

After ModelSim verification, the CPU was synthesized and deployed on the FPGA.

Using SignalTap, we validated:

- Live instruction flow across pipeline stages

- Real-time stalls and flushes

- Breakpoint detection

- Output memory content

The contents of memory after execution were compared between Quartus and ModelSim and matched, confirming consistent behavior between simulation and hardware.

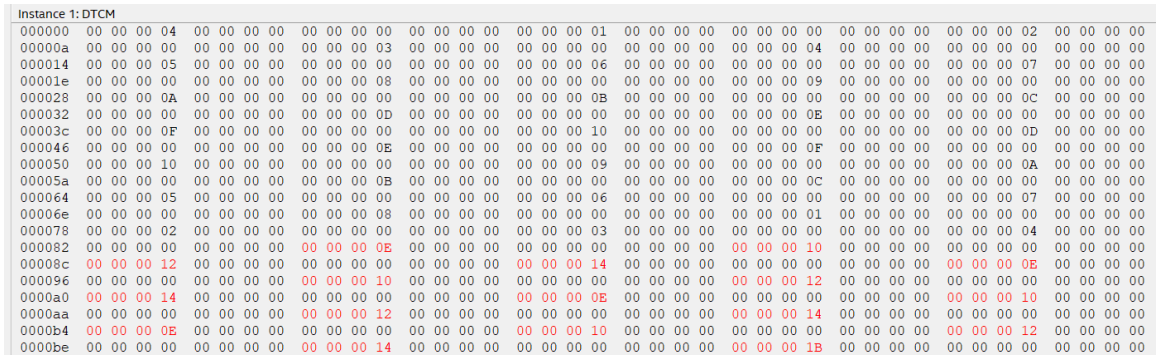


Figure 9: Quartus memory content

To validate runtime control, a breakpoint mechanism was implemented. Figure 9 shows a capture in SignalTap where execution halts when the PC reaches address 0x030, as expected.

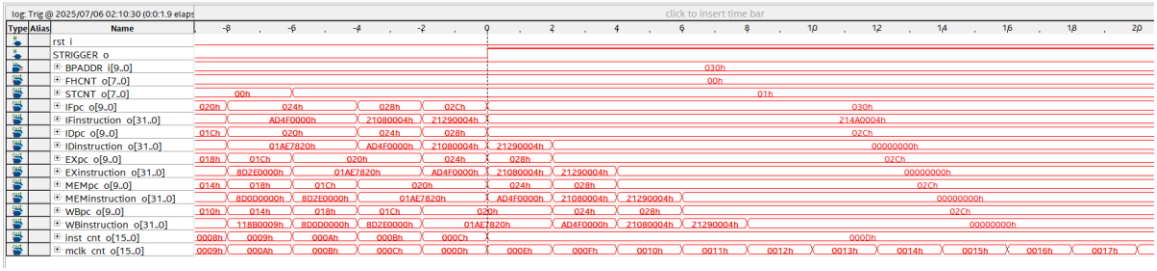


Figure 10: SignalTap breakpoint at PC=0x30

This confirms correct operation of the breakpoint detection and integration with pipeline control.

The following show correct stalling after data hazard detected and stall counter incremented by 1:

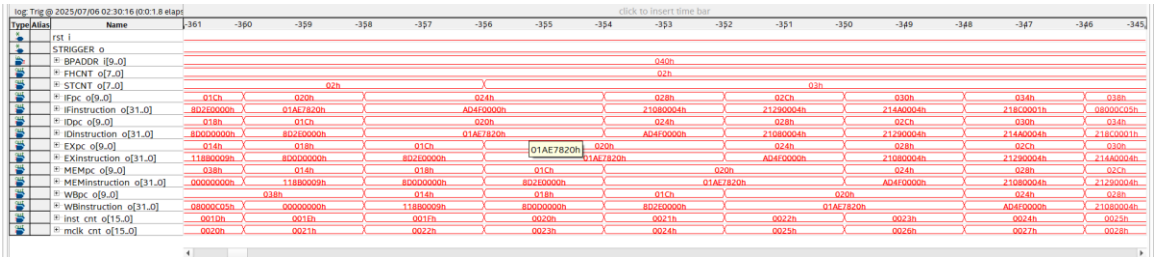


Figure 11: SignalTap, stall after data dependency.

The following show correct stalling after wrong instruction detected (branch taken) and flush counter incremented by 1:

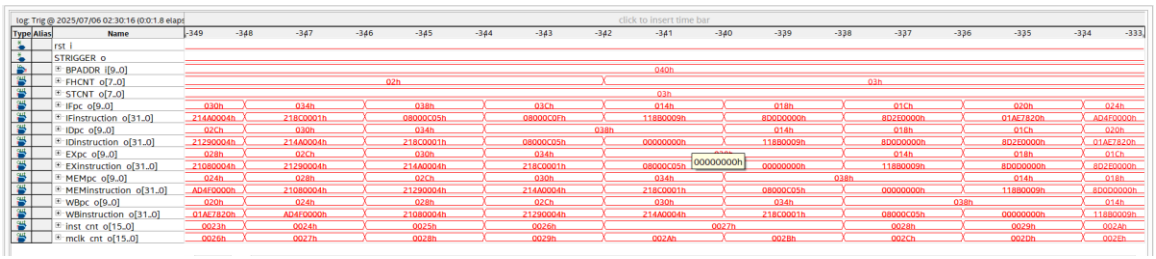


Figure 12: SignalTap, flush after branch taken.

5. Comparison

As shown in Figures X and Y, the data memory after execution is consistent between simulation (ModelSim) and real hardware (Quartus). This validates that memory control logic, address calculation, and write-back stages behave identically.

6. IPC calculation

The execution of the program took 199 clock cycle, 181 instructions, 16 stalls and 16 flush**.**

We can calculate the IPC using the following equation:

$$\textbf{Performance:}\quad IPC = \frac{\text{CLKCNT\_o} - (STCNT\_o + 4 + \text{dept}*\text{FHCNT\_o})}{\text{CLKCNT\_o}} = \frac{INSTCNT_o - \text{dept}*\text{FHCNT\_o}}{CLKCNT\_o}$$

where: IPC=1/CPI

Using the clock cycle counter and instruction counter on FPGA, we calculated the IPC (Instructions Per Cycle) of the matrix addition program to be approximately 0.83. This accounts for stalls due to load-use hazards and control flow changes like branches.

7. critical path and maximum clock for pipelined mips

Following the maximum clock speed for the pipelined mips:

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name
1	32.79 MHz	32.79 MHz	\G0:MCLK|altpll_c...UT_COUNTER|divclk
2	63.93 MHz	63.93 MHz	altera_reserved_tck
3	132.19 MHz	132.19 MHz	clk_i

Figure 13: Maximum clock speed of the pipelined mips

The following show the critical path of the pipelined mips CPU:

Figure 14: critical path of the pipelined mips.

8. critical path and maximum clock for pipeline mips

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name
1	34.98 MHz	34.98 MHz	\G0:MCLK|altpll_c...UT_COUNTER|divclk
2	72.16 MHz	72.16 MHz	altera_reserved_tck
3	124.63 MHz	124.63 MHz	clk_i

Figure 15: Maximum clock speed of the single cycle mips.

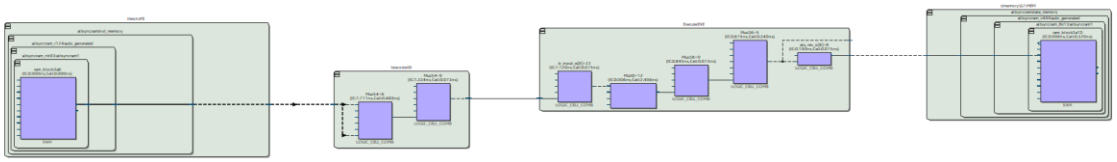
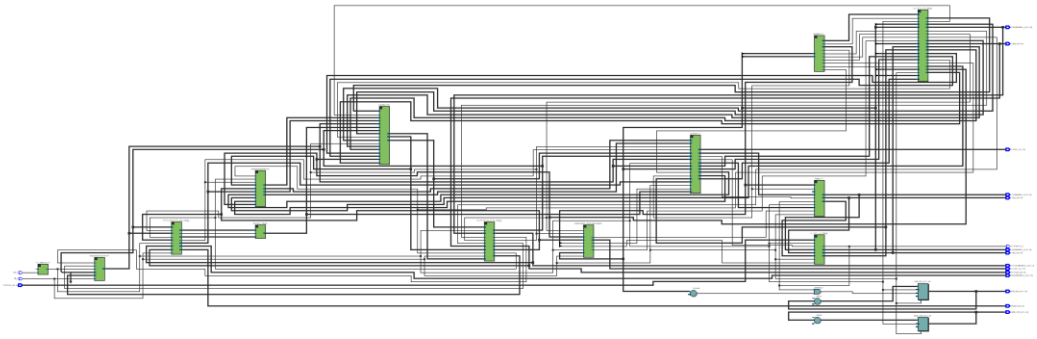


Figure 16: critical path of the single cycle mips.

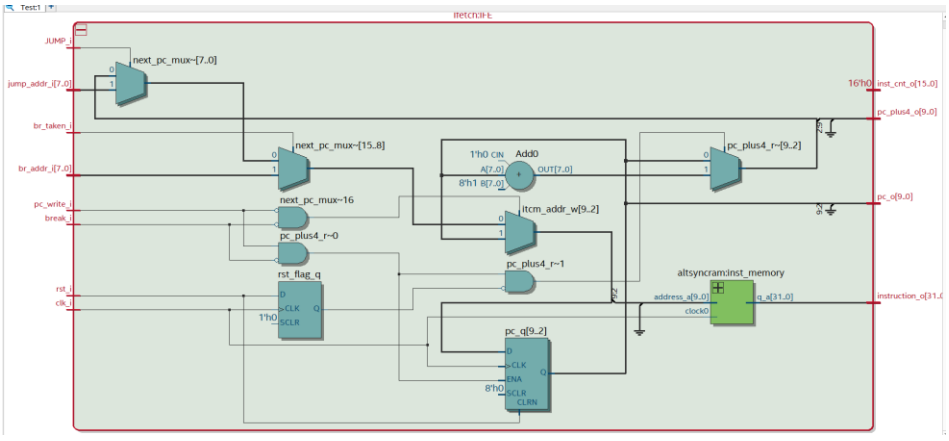
9. RTL

MIPS Pipeline Processor RTL



Instruction Fetch

Instruction Fetch (IF) stage is responsible for retrieving the next instruction to be executed. It reads the current value of the Program Counter (PC), uses it to access the instruction memory, fetches the instruction located at that address, and calculates the address of the next instruction by adding 4 to the PC. The fetched instruction and the incremented PC (PC + 4) are then passed to the next pipeline stage (Instruction Decode), while the PC is updated to point to the next instruction, unless a branch or jump alters the flow.

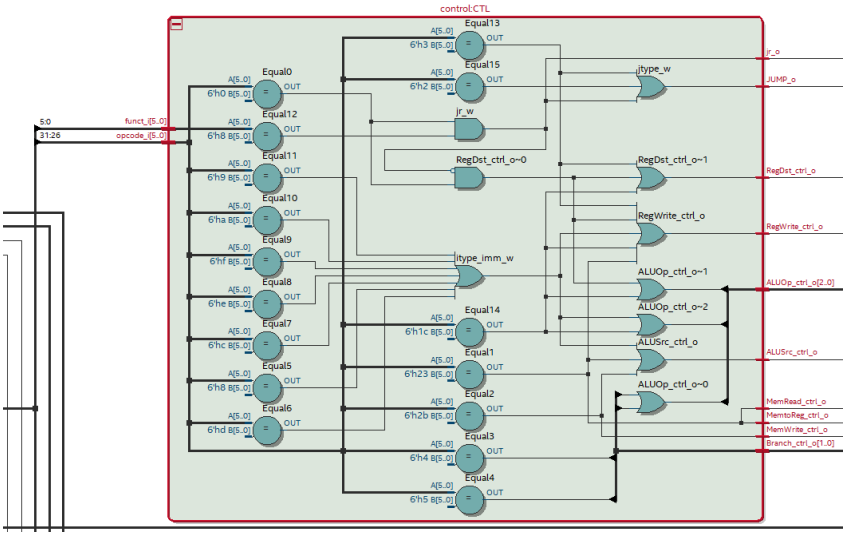


Instruction Decode

Instruction Decode (ID) stage interprets the fetched instruction and prepares necessary data for execution. It decodes the instruction to determine the operation type and identifies the source and destination registers. The processor reads the values of the source registers from the register file, and sign-extends the immediate field if the instruction requires it (e.g., for load/store or arithmetic immediate operations). Additionally, the ID stage calculates potential branch targets and checks branch conditions. All decoded information and register values are passed to the Execute (EX) stage via the ID/EX pipeline register.

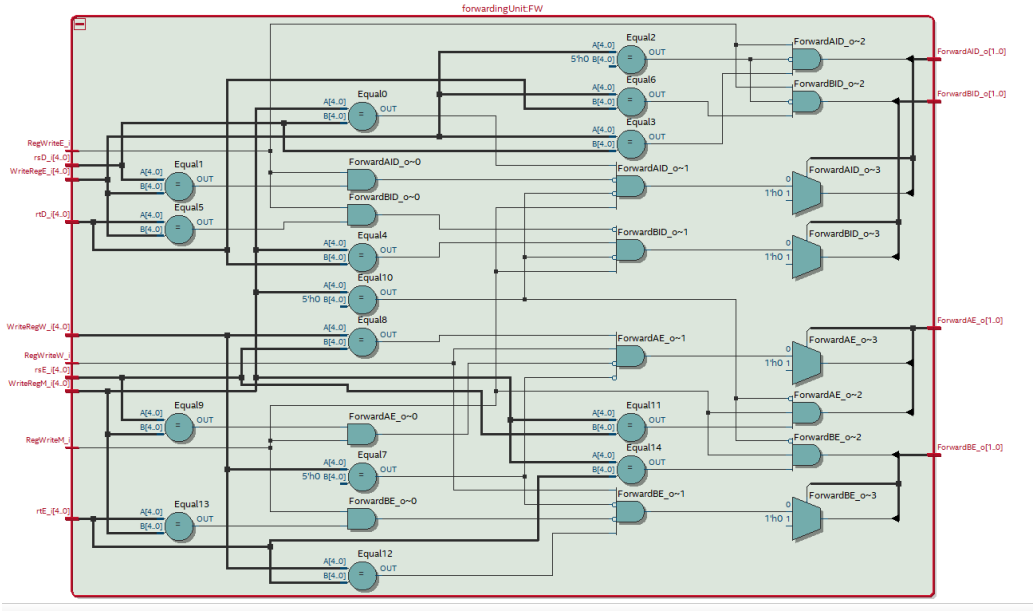
Control Unit

Control Unit is responsible for generating the appropriate control signals based on the decoded instruction. During the Instruction Decode (ID) stage, the Control Unit examines the opcode (and function code for R-type instructions) to determine the type of instruction (e.g., arithmetic, memory access, or branch) and produces signals that guide the operation of each pipeline stage. These signals control actions like register writing, ALU operations, memory reading/writing, and branching. The Control Unit ensures that each component in the datapath performs the correct function for the given instruction, enabling coordinated and correct execution.



Forwarding Unit

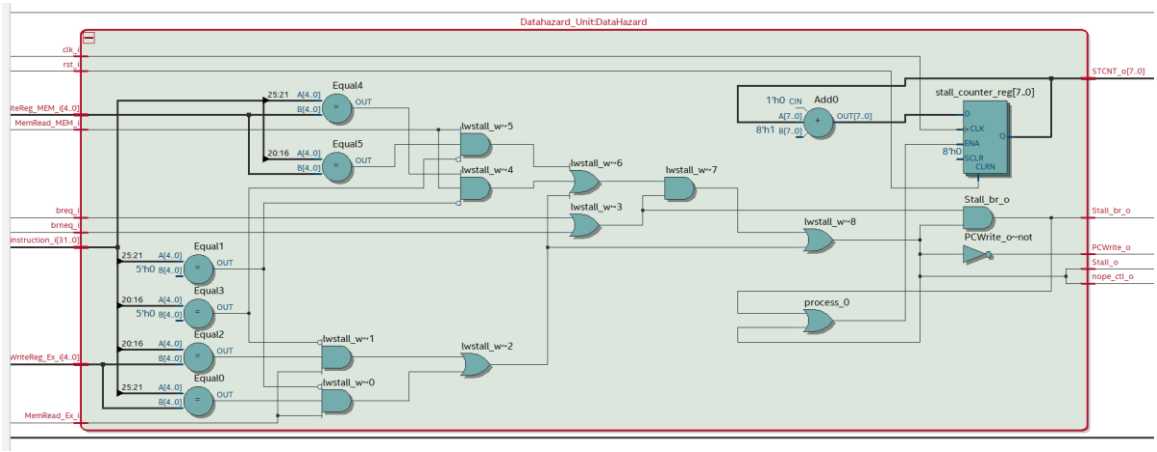
The Forwarding Unit in a MIPS pipeline processor is a hazard detection mechanism that resolves data hazards by forwarding data directly between pipeline stages without waiting for it to be written back to the register file. When an instruction depends on the result of a previous instruction that has not yet completed the Write Back (WB) stage, the Forwarding Unit detects this and routes the needed data from the EX or MEM stage back to the EX stage inputs. This allows instructions to execute without stalling the pipeline, improving performance and maintaining correct program behavior.



Hazard Unit

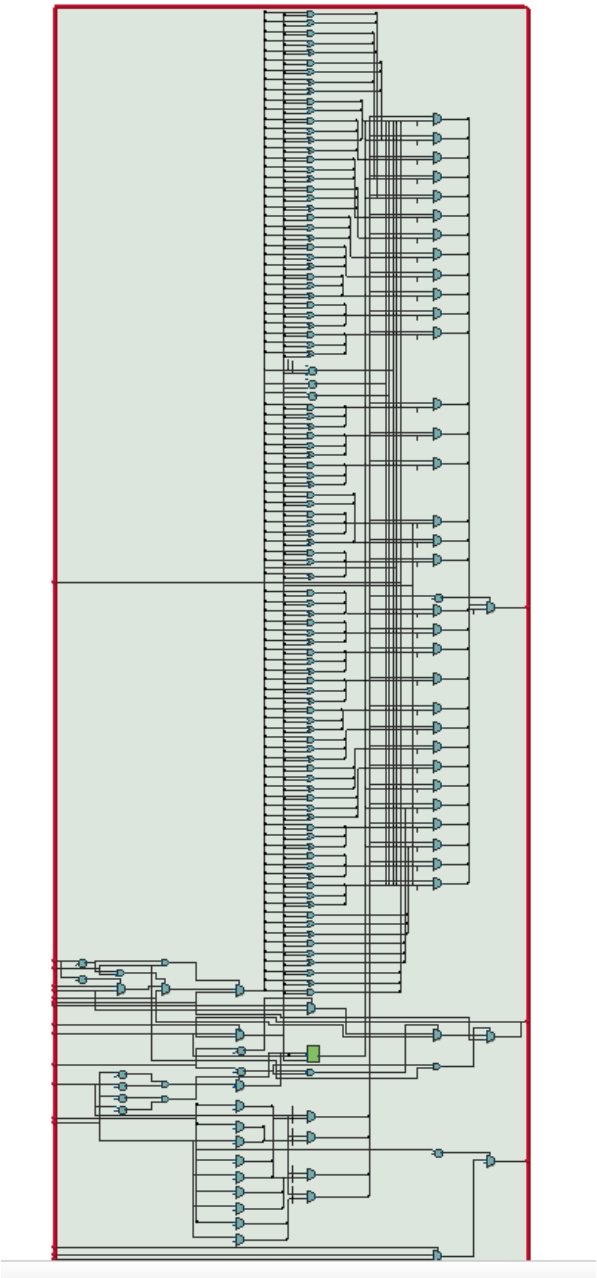
The Hazard Detection Unit in a MIPS pipeline processor is responsible for identifying situations that could cause incorrect execution due to data or control hazards, and applying solutions like stalls (pausing pipeline stages) or flushes (clearing incorrect instructions).

- It stalls the pipeline (usually by preventing updates to the PC and IF/ID registers) when a load-use data hazard is detected — for example, when an instruction tries to use a register immediately after a lw that loads it.
- It flushes instructions (inserting no-ops) when a control hazard occurs, such as after a taken branch or jump, to discard instructions that were fetched based on the wrong PC.



Execute Stage

The Execute (EX) stage in a MIPS pipeline processor performs the core computation or address calculation for the instruction. For arithmetic or logic instructions, the ALU carries out the operation (e.g., add, sub, and). For memory instructions like lw or sw, it calculates the effective memory address by adding the base register and the offset. This stage also uses forwarded data if needed and selects the correct ALU operands based on control signals. The results are then passed to the Memory (MEM) stage.



Data Memory

The Memory (MEM) stage in a MIPS pipeline processor handles data memory access for load and store instructions. If the instruction is a load (e.g., lw), it uses the address calculated in the EX stage to read data from memory. If it's a store (e.g., sw), it writes data from a register to the calculated memory address. For other instruction types (like arithmetic or branches), this stage typically just passes the result forward without accessing memory. The output of this stage is then forwarded to the Write Back (WB) stage.

Write Back Stage

The Write Back (WB) stage is the final stage, where the result of an instruction is written back to the register file. For R-type and I-type arithmetic instructions, the ALU result is written to the destination register. For load instructions (e.g., lw), the data read from memory is written to the target register. The control unit determines which value is written and to which register, based on the instruction type.