DESIGN OF 32-BIT MIPS PROCESSOR

JULY 16, 2025

BY: SHEHAB ELDEEN KHALED

GITHUB REPO LINK

Contents

1- Introduction

2- Single Cycle MIPS Processor

- Architecture.
- Design blocks.
- Supported instructions.
- Control signals
- ALU decoder truth table
- Waveform
- FPGA flow
- Single cycle processor limitations
- Possible Solution to Single cycle processor limitations

3- Pipelined MIPS Processor

- Stages of pipeline.
- Key stages of pipeline datapath.
- Pipeline hazards.
- Solving data hazards with forwarding.
- Solving data hazards with stall.
- Solving control hazards with stall.
- Architecture.
- Supported instructions.
- Control signals.
- Design blocks.
- Waveform.
- FPGA flow.
- MIPS assembly testing code.
- Single cycle processor limitations.

➢ Introduction:

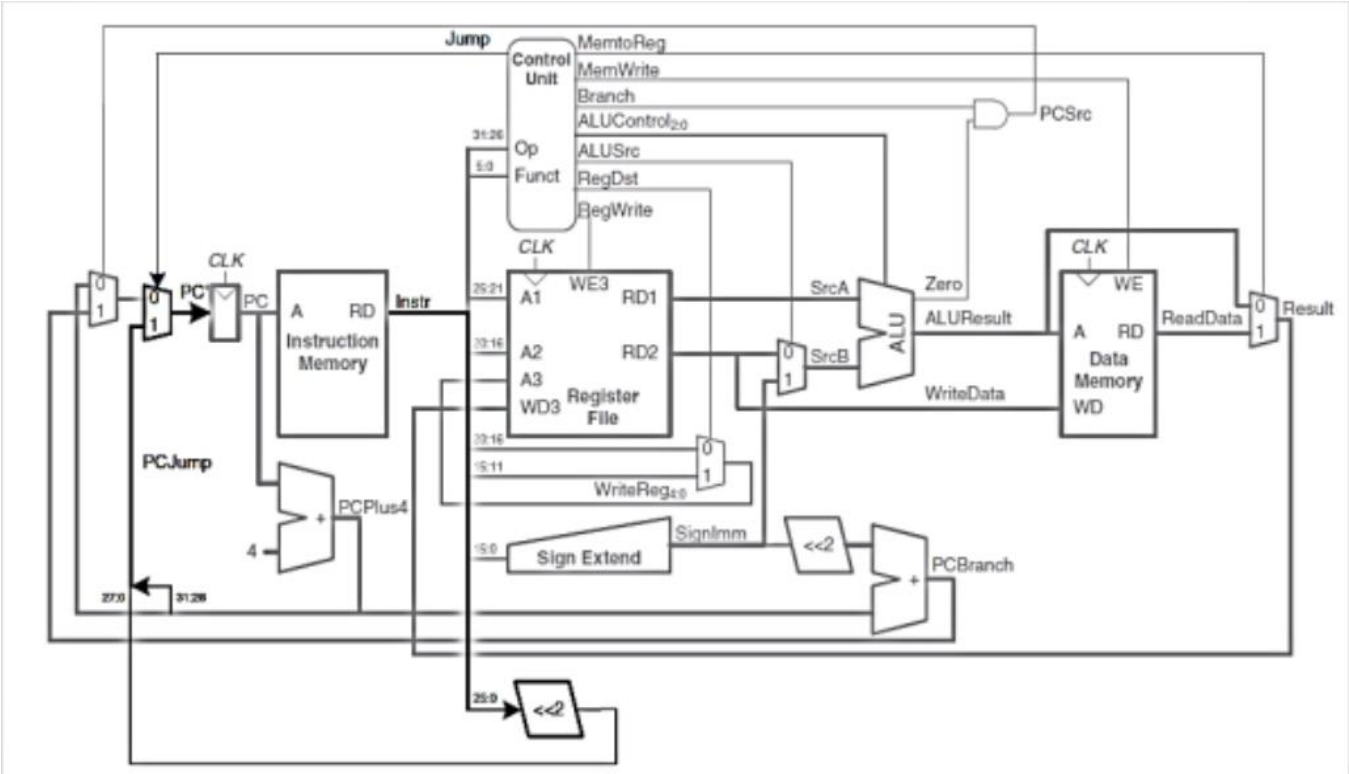
⇨ This document presents the design and implementation of both Single-Cycle and Pipelined MIPS processors using Verilog HDL. **Questasim** is used for simulation to verify the correctness of the design, while **Vivado** is used for applying the FPGA flow from elaboration and synthesis to place-and-route (PnR), and performing timing analysis.

⇨ We start with the single-cycle model and progressively transition to the pipelined version. This step-by-step progression is designed to help you grasp how pipelining modifies the MIPS architecture using Verilog code.

➢ Single cycle MIPS Processor:

⇨ Architecture:

➢ The single-cycle MIPS design executes each instruction in a one clock cycle, handling all stages from fetch to write-back within that cycle.



⇨ Design blocks:

- **PC:** Program Counter to track instruction addresses
- **ALU:** Arithmetic Logic Unit for executing ALU operations (add, sub, AND, OR, slt).
- **Control unit:** Decodes instructions and generates control signals.
- **Data memory:** Simulates read/write access to memory.
- **Instruction memory:** Stores the instruction set for simulation.
- **Mux:** Multiplexers used in various data path decision points.
- **Adder:** Performs PC + 4 or branch target calculation.
- **Register file:** Implements 32 general-purpose registers with read/write functionality.
- **Shift left2:** Shifts input by 2 bits (used in branch offset computation).
- **Shift left jump:** Shifts the jump target address.
- **Sign extend:** Sign extends 16-bit immediate to 32 bits.

⇨ Supported instructions:

- R-type: add, sub, AND, OR, slt.
- I-type: lw, sw, beq, addi.
- J-type: j

⇨ Control signals:

- **RegDst:** Destination register selection.
- **ALUSrc:** ALU input selection.
- **MemToReg:** Data to write to register file.
- **RegWrite:** Enables register write.
- **MemRead:** Enables data memory read.
- **MemWrite:** Enables data memory write.
- **Branch:** Branch decision.
- **ALUOp:** Specifies ALU operation.
- **Jump:** Enabled when jump instruction is executed

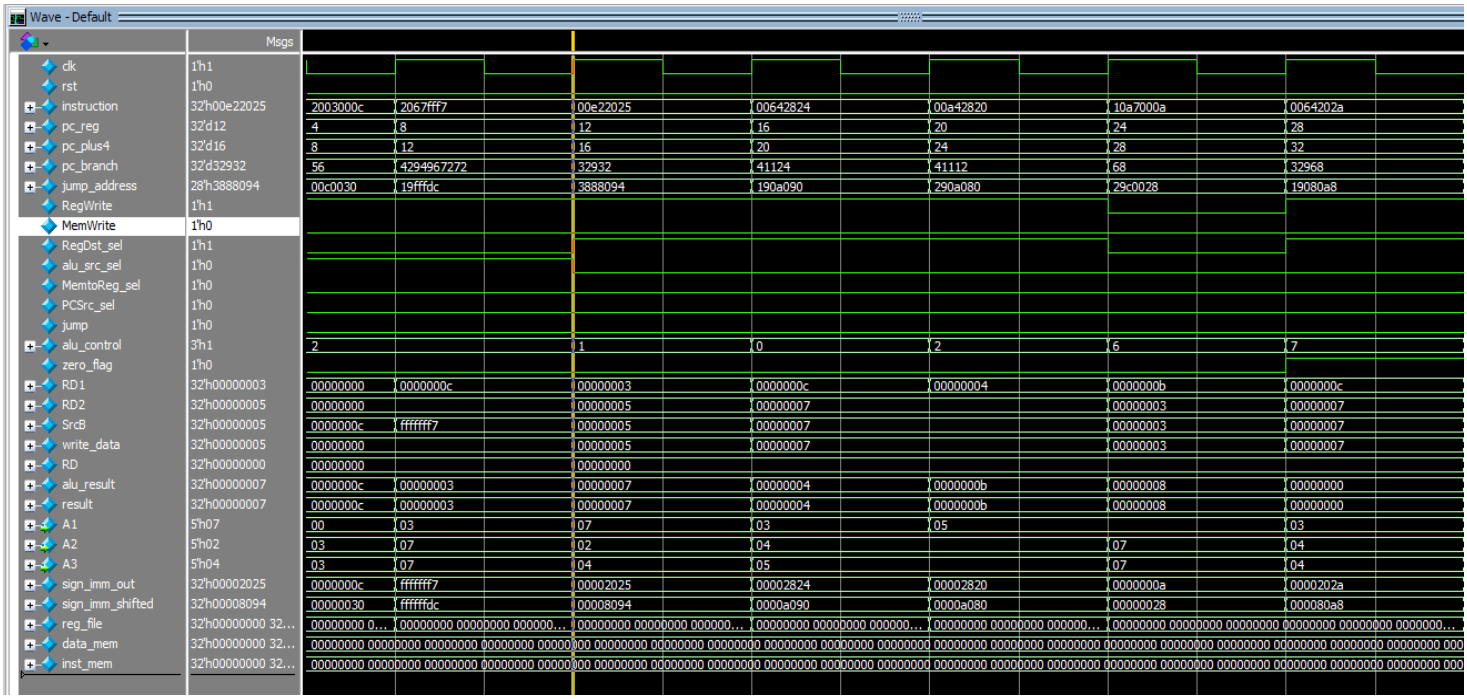
⇨ Control signals for each instruction:

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1

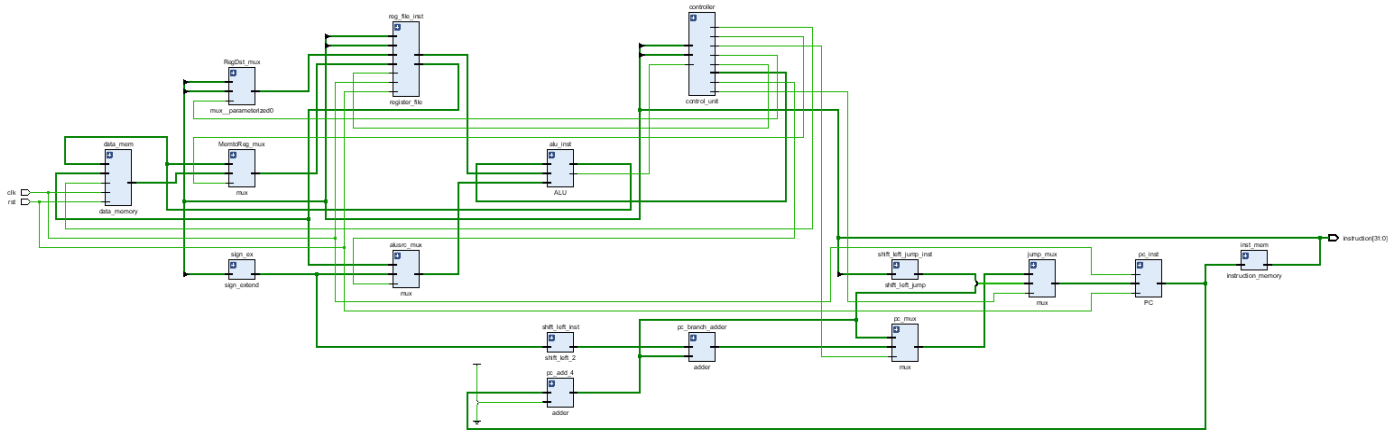
⇨ ALU decoder truth table:

ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)

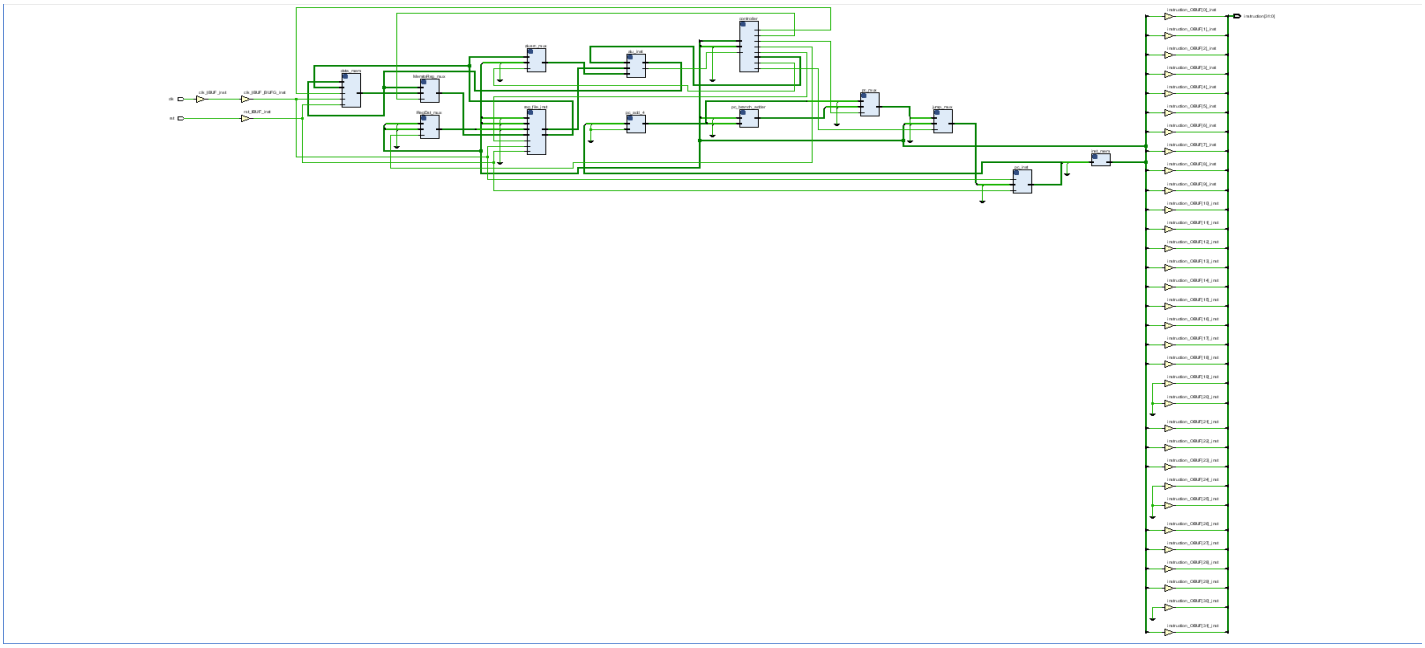
⇨ Waveform:



⇨ RTL Elaboration:



⇨ RTL Synthesis:



⇨ Timing Analysis after Synthesis:

Design Timing Summary

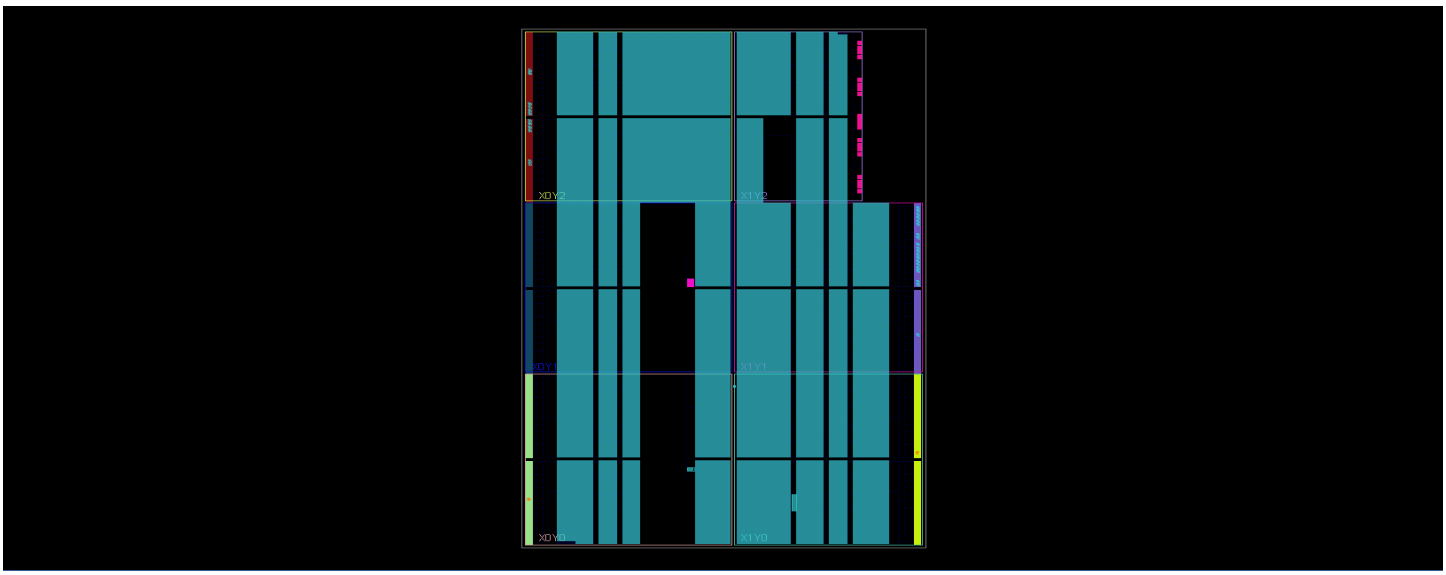
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	8.340 ns	Worst Hold Slack (WHS):	0.664 ns	Worst Pulse Width Slack (WPWS):	9.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	65994	Total Number of Endpoints:	65994	Total Number of Endpoints:	33003

All user specified timing constraints are met.

⇨ Utilization Report after Synthesis:

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Bonded IOB (106)	BUFGCTRL (32)
N mips	10222	33002	4416	2176	34	1
alu_inst (ALU)	90	0	0	0	0	0
alusrc_mux (mux___3)	16	0	0	0	0	0
controller (control_unit)	9	0	0	0	0	0
data_mem (data_me...	9911	32768	4352	2176	0	0
inst_mem (instruction...	21	0	0	0	0	0
jump_mux (mux___2)	5	0	0	0	0	0
MemtoReg_mux (mux)	16	0	0	0	0	0
pc_add_4 (adder___1)	1	0	0	0	0	0
pc_branch_adder (add...	11	0	0	0	0	0
pc_inst (PC)	0	10	0	0	0	0
pc_mux (mux___1)	5	0	0	0	0	0
reg_file_inst (register_...	135	224	64	0	0	0
RegDst_mux (mux___p...	2	0	0	0	0	0

⇨ Implementation on FPGA:



⇨ Timing Analysis after implementation:

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.213 ns	Worst Hold Slack (WHS):	0.070 ns	Worst Pulse Width Slack (WPWS):	8.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	69786	Total Number of Endpoints:	69770	Total Number of Endpoints:	35128

All user specified timing constraints are met.

⇨ Utilization Report after implementation:

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	LUT Flip Flop Pairs (20800)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)	BSCANE2 (4)
mips	11395	34928	4426	2176	8127	11281	114	694	1	34	2	1
alu_inst (ALU)	89	0	0	0	38	89	0	0	0	0	0	0
alusrc_mux (mux__3)	16	0	0	0	14	16	0	0	0	0	0	0
controller (control_unit)	9	0	0	0	7	9	0	0	0	0	0	0
data_mem (data_me...)	9911	32768	4352	2176	7668	9911	0	0	0	0	0	0
dbg_hub (dbg_hub)	473	727	0	0	196	449	24	295	0	0	1	1
inst_mem (instruction...	21	0	0	0	10	21	0	0	0	0	0	0
jump_mux (mux__2)	5	0	0	0	2	5	0	0	0	0	0	0
MemtoReg_mux (mux)	16	0	0	0	11	16	0	0	0	0	0	0
pc_add_4 (adder__1)	0	0	0	0	3	0	0	0	0	0	0	0
pc_branch_adder (add...	11	0	0	0	4	11	0	0	0	0	0	0
pc_inst (PC)	0	10	0	0	2	0	0	0	0	0	0	0
pc_mux (mux__1)	5	0	0	0	4	5	0	0	0	0	0	0
reg_file_inst (register_...	135	224	64	0	77	135	0	0	0	0	0	0
RegDst_mux (mux__p...	2	0	0	0	2	2	0	0	0	0	0	0
u_ila_0 (u_ila_0)	703	1199	10	0	265	613	90	393	1	0	0	0

⇨ Single cycle processor limitations:

In a single-cycle design, all instructions are executed within a fixed-length clock cycle, resulting in a **CPI (Cycles Per Instruction) of 1.**

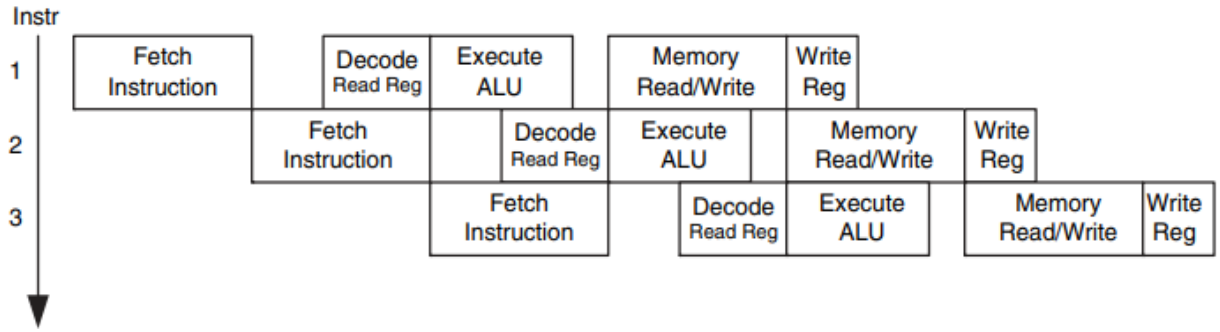
- The clock period is dictated by the longest delay path, typically associated with load instructions that involve accessing instruction memory, reading from the register file, performing ALU operations, accessing data memory, and writing back to the register file.
- This worst-case delay determines the clock cycle time, making it impossible to optimize for more common, faster instructions—violating the principle of optimizing for the common case.
- Although the CPI is 1, overall performance is limited by the long clock cycle, as many instructions could be executed in much less time.
- Instruction delays vary widely due to differences in complexity and addressing modes.
- The single-cycle approach requires each functional unit to be used exactly once per instruction cycle, leading to duplication of hardware resources, which increases cost and wastes silicon area.
- As a result, single-cycle designs are inefficient both in terms of performance and hardware utilization.

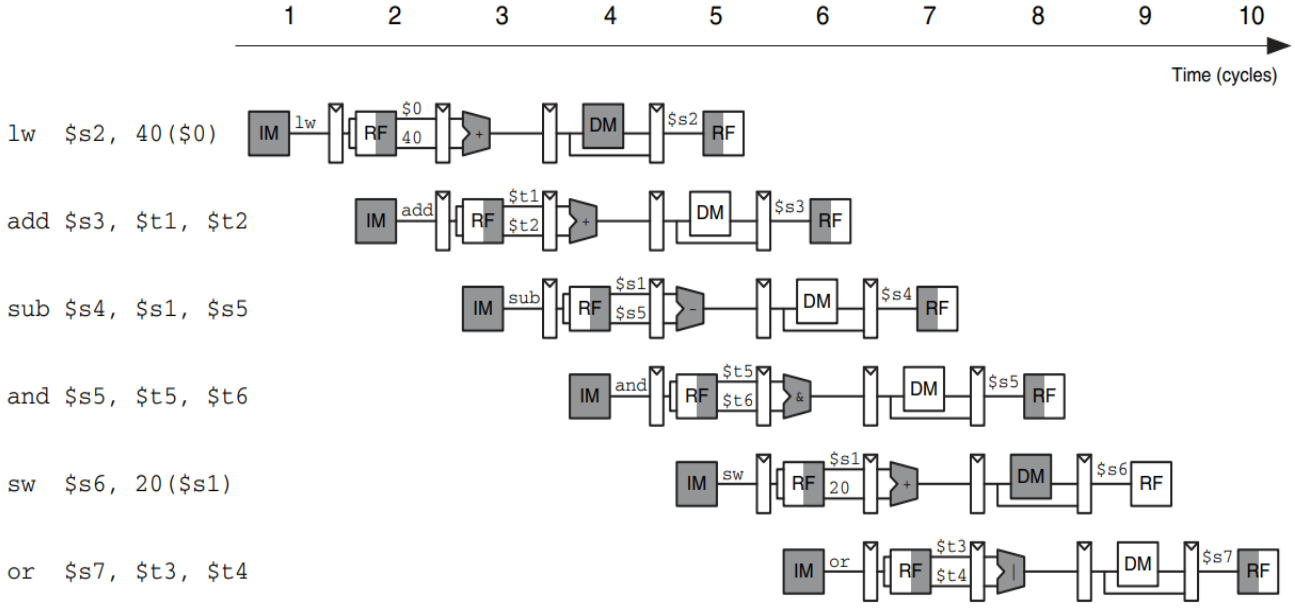
⇨ Possible solutions include:

- Adopting multi-cycle designs, where the clock cycle is shorter and the number of cycles varies depending on the instruction type.
- Implementing pipelining, which maintains a similar data path but overlaps instruction execution, significantly improving efficiency and performance and **this is what we will do.**

Pipelined MIPS Processor:

Pipelining is a method used to enhance processor throughput by overlapping the execution of multiple instructions. Rather than completing one instruction before starting the next, so five instructions can execute simultaneously, one in each stage. because each stage has only one-fifth of the entire logic, the clock frequency is almost five times faster. pipelining breaks down instruction execution into multiple stages, with each stage handled by separate hardware components operating concurrently. This technique is like an assembly line in manufacturing, where different stages work on different instructions at the same time, enabling faster overall processing.





⇨ Stages of the Pipeline:

The typical stages of a pipelined MIPS processor include:

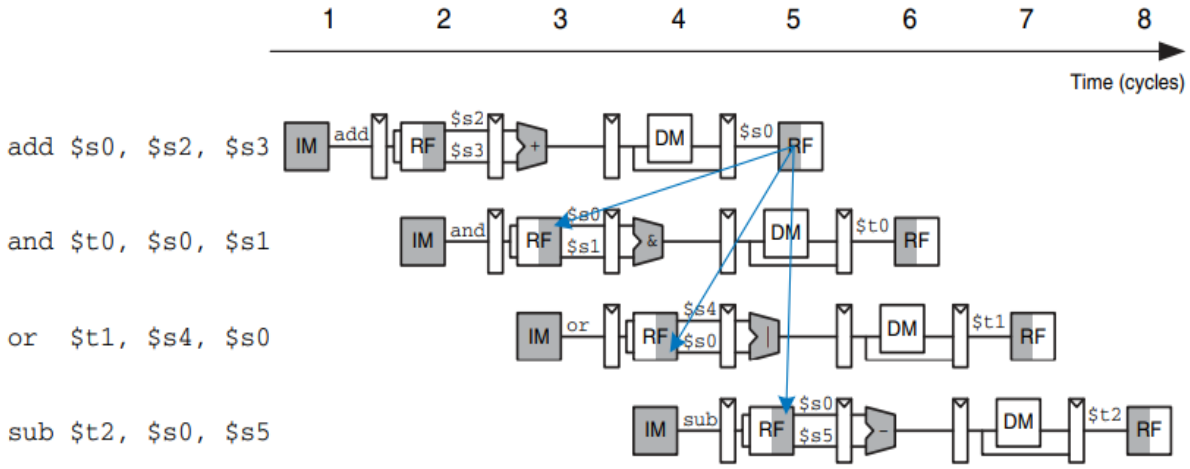
- **Instruction Fetch (IF):** The instruction is retrieved from memory.
- **Instruction Decode (ID):** The instruction is decoded, and necessary registers are read.
- **Execution (EX):** The ALU performs the required operation.
- **Memory Access (MEM):** Data memory is accessed if needed.
- **Write Back (WB):** The result is written back to the register file.

⇨ Key Stages of the Pipelined Datapath:

- **Instruction Fetch (IF):**
  Retrieves the next instruction from memory using the current value of the Program Counter (PC).
- **Instruction Decode (ID):**
  Interprets the fetched instruction and reads the required operands from the register file.
- **Execute (EX):**
  Carries out arithmetic or logic operations as specified by the instruction and calculates memory addresses for load/store operations.
- **Memory Access (MEM):**
  Handles reading from or writing to data memory, primarily for load and store instructions.
- **Write Back (WB):**
  Stores the final result of the instruction execution back into the register file.

⇨ Pipelining Hazards:

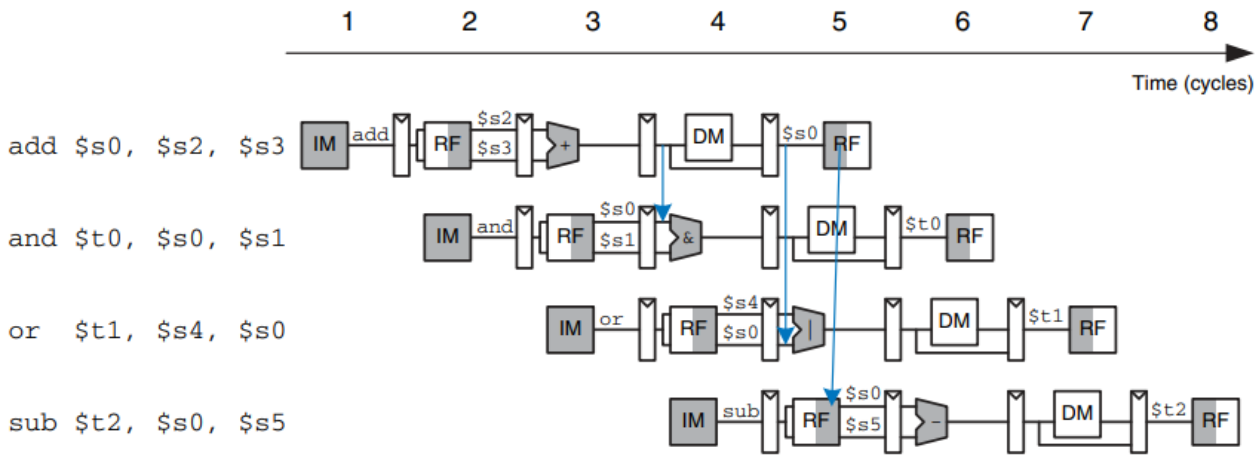
In a pipelined system, multiple instructions are handled concurrently. When one instruction is dependent on the results of another that has not yet completed, a hazard occurs.



- This figure illustrates hazards that occur when one instruction writes a register ($s0) and subsequent instructions read this register. This is called a read after write (RAW) hazard. The **add** instruction writes a result into $s0 in the first half of cycle 5. However, the **AND** instruction reads $s0 on cycle 3, obtaining the wrong value. The **OR** instruction reads $s0 on cycle 4, again obtaining the wrong value. The sub instruction reads $s0 in the second half of cycle 5, obtaining the correct value, which was written in the first half of cycle 5.
- Hazards are classified as data hazards or control hazards:
  1- **Data hazard:** occurs when an instruction tries to read a register that has not yet been written back by a previous instruction.
  2- **Control hazard:** occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place.

⇨ Solving data hazards with forwarding:

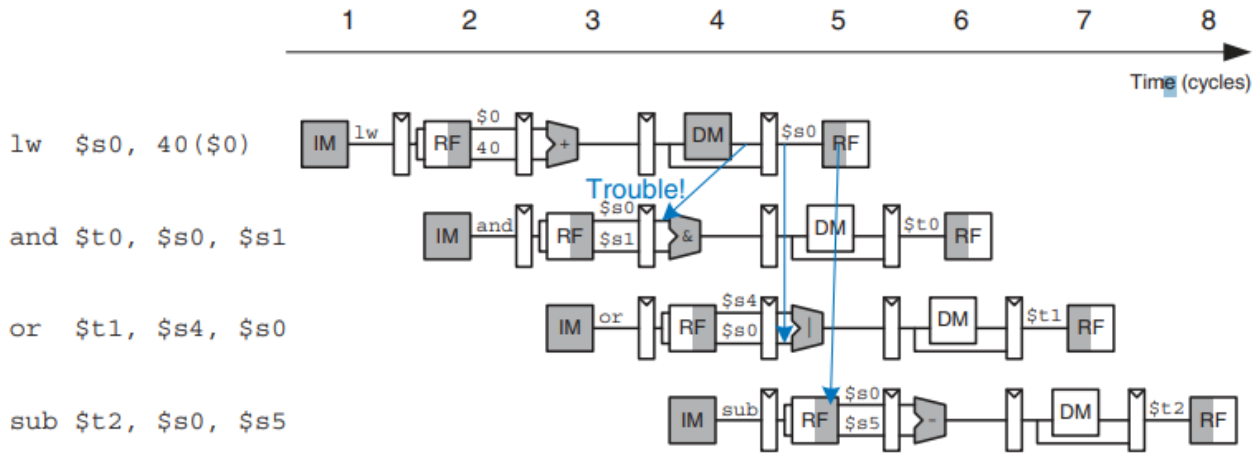
Forwarding is necessary when an instruction in the Execute stage has a source register matching the destination register of an instruction in the Memory or Writeback stage.



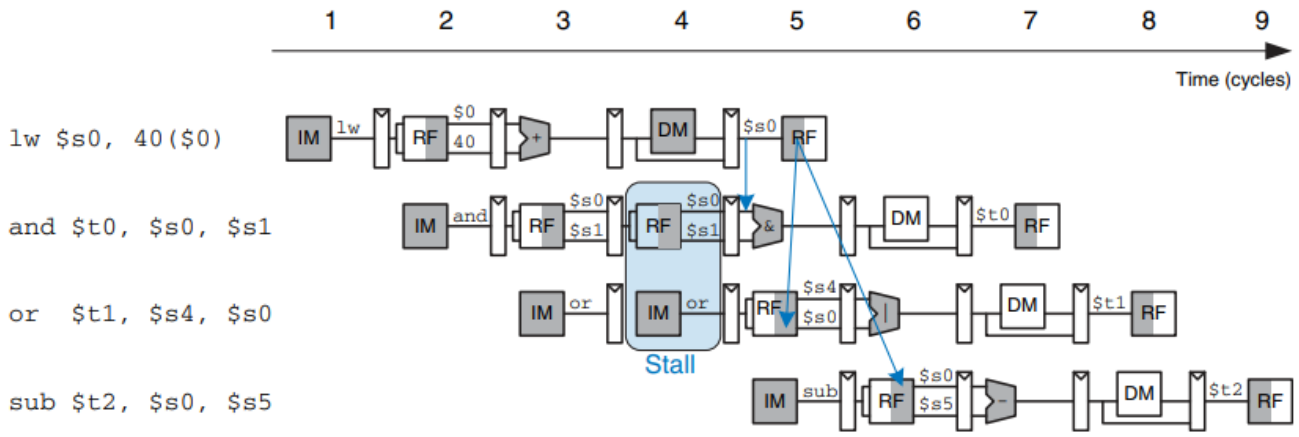
- This figure illustrates this principle. In cycle 4, $s0 is forwarded from the Memory stage of the add instruction to the Execute stage of the dependent and instruction. In cycle 5, $s0 is forwarded from the Writeback stage of the add instruction to the Execute stage of the dependent or instruction
- So the function of the forwarding logic for SrcA is given below. The forwarding logic for SrcB (ForwardBE) is identical except that it checks rt rather than rs.

```
always @(*) begin
    if ((rsE != 0) && (rsE == write_regM) && (RegWriteM)) begin
        forwardAE = 2'b10;
    end
    else if ((rsE != 0) && (rsE == write_regW) && (RegWriteW)) begin
        forwardAE = 2'b01;
    end
    else begin
        forwardAE = 2'b00;
    end
end
```

⇨ Solving data hazards with stall:



- As shown in this figure, Forwarding is sufficient to solve RAW data hazards when the result is computed in the Execute stage of an instruction, because its result can then be forwarded to the Execute stage of the next instruction. Unfortunately, the lw instruction does not finish reading data until the end of the Memory stage, so its result cannot be forwarded to the Execute stage of the next instruction. We say that the lw instruction has a two-cycle latency, because a dependent instruction cannot use its result until two cycles later.
- The LW instruction receives data from memory at the end of cycle 4. But the AND instruction needs that data as a source operand at the beginning of cycle 4. There is no way to solve this hazard with forwarding.
- The alternative solution is to **stall the pipeline**, holding up operation until the data is available as in the figure below.
- Stalling a stage is performed by disabling the pipeline register, so that the contents do not change. When a stage is stalled, all previous stages must also be stalled, so that no subsequent instructions are lost. The pipeline register directly after the stalled stage must be cleared to prevent bogus information from propagating forward. Stalls degrade performance, so they should only be used when necessary.
- Stalls are supported by adding enable inputs (EN) to the Fetch and Decode pipeline registers and a synchronous reset/clear (CLR) input to the Execute pipeline register.



- This figure shows stalling the dependent instruction (AND) in the Decode stage. and enters the Decode stage in cycle 3 and stalls there through cycle 4. The subsequent instruction (or) must remain in the Fetch stage during both cycles as well, because the Decode stage is full. In cycle 5, the result can be forwarded from the Writeback stage of lw to the Execute stage of and. In cycle 5, source $s0 of the or instruction is read directly from the register file, with no need for forwarding.
- When a **lw** stall occurs**, StallD** and **StallF** are asserted to force the Decode and Fetch stage pipeline registers to hold their old values. **FlushE** is also asserted to clear the contents of the Execute stage pipeline register.
- The **MemtoReg** signal is asserted for the lw instruction. Hence, the logic to compute the stalls and flushes is:

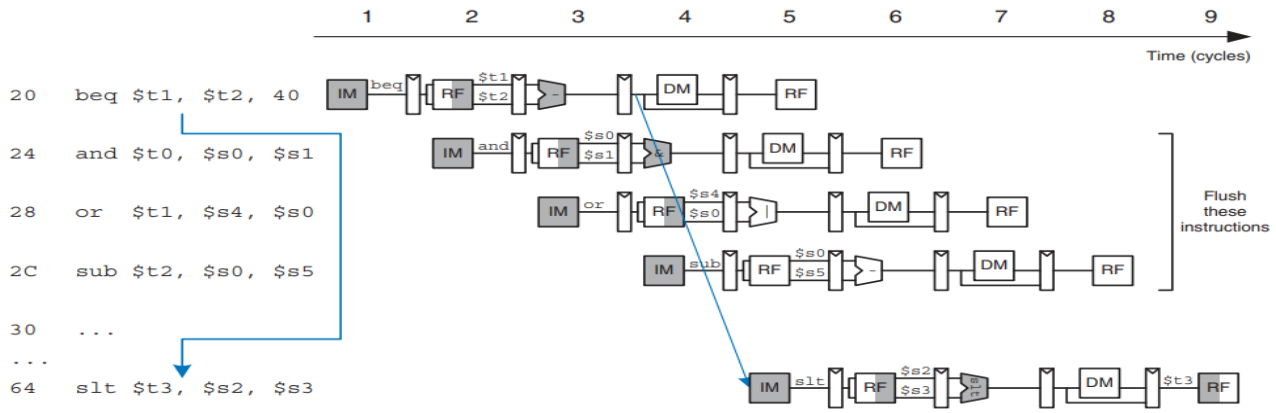
  lwstall = ((rsD = = rtE) OR (rtD = = rtE)) AND MemtoRegE

  StallF = StallD = FlushE = lwstall

⇨ Solving control hazards with stall:

The **beq** instruction presents a control hazard: the pipelined processor does not know what instruction to fetch next, because the branch decision has not been made by the time the next instruction is fetched.

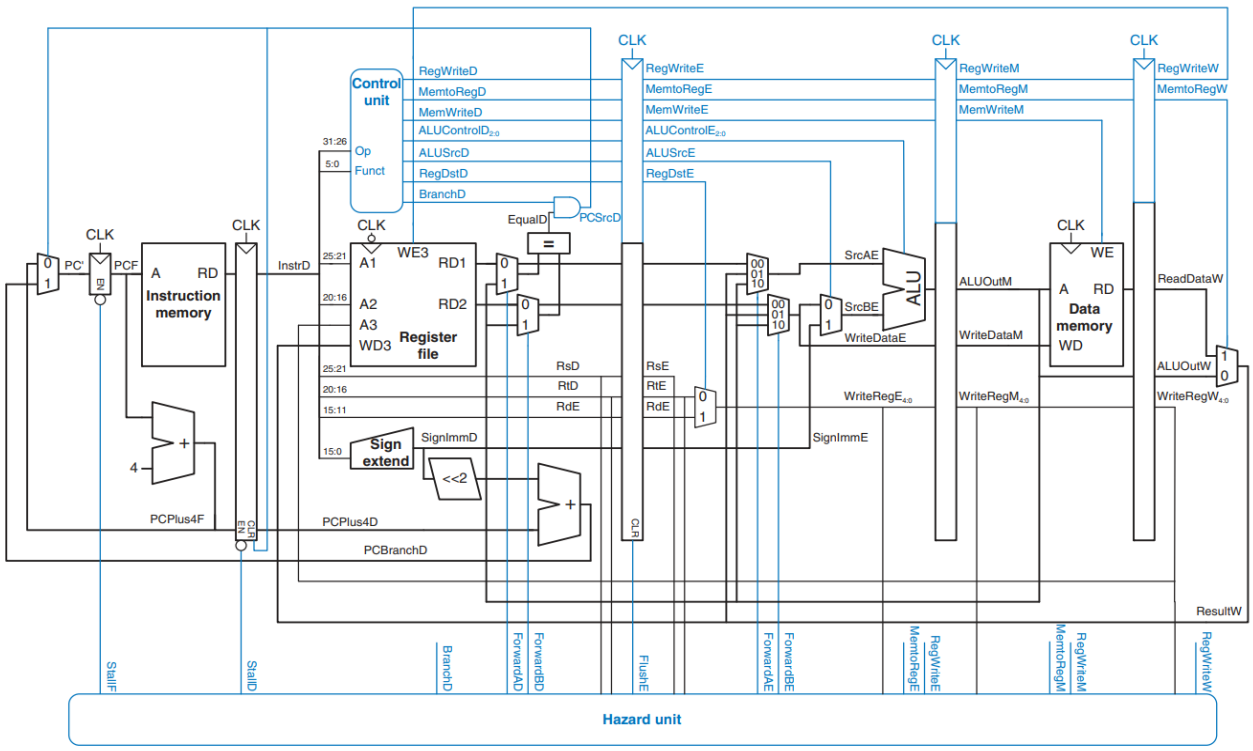
- The solution is to predict whether the branch will be taken and begin executing instructions based on the prediction. Once the **branch decision** is available, the processor can throw out the instructions if the prediction was wrong. Suppose that we predict that branches are not taken and simply continue executing the program in order. If the branch should have been taken, the three instructions following the branch must be **flushed (discarded)** by clearing the pipeline registers for those instructions.



- This figure shows such a scheme, in which a branch from address 20 to address 64 is taken. The branch decision is not made until cycle 4, by which point the **AND**, **OR**, and **sub** instructions at addresses 24, 28, and 2C have already been fetched. These instructions must be flushed, and the **slt** instruction is fetched from address 64 in cycle 5
- The function of the stall detection logic for a branch is given below. The processor must make a branch decision in the Decode stage. If either of the sources of the branch depend on an ALU instruction in the Execute stage or on a lw instruction in the Memory stage, the processor must stall until the sources are ready.
  branchstall = BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD)
  OR BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD)

- Now the processor might stall due to either a load or a branch hazard:
  StallF = StallD = FlushE = lwstall OR branchstall

⇨ Pipelined processor with full hazard handling architecture:



⇨ Supported instructions:

- Arithmetic: add, sub, addi
- Logical: or, and
- Comparison: slt
- Memory: lw, sw
- Control Flow: beq, j

⇨ Control signals:

- **RegDst:** Destination register selection.
- **ALUSrc:** ALU input selection.
- **MemToReg:** Data to write to register file.
- **RegWrite:** Enables register write.
- **MemRead:** Enables data memory read.
- **MemWrite:** Enables data memory write.
- **Branch:** Branch decision.

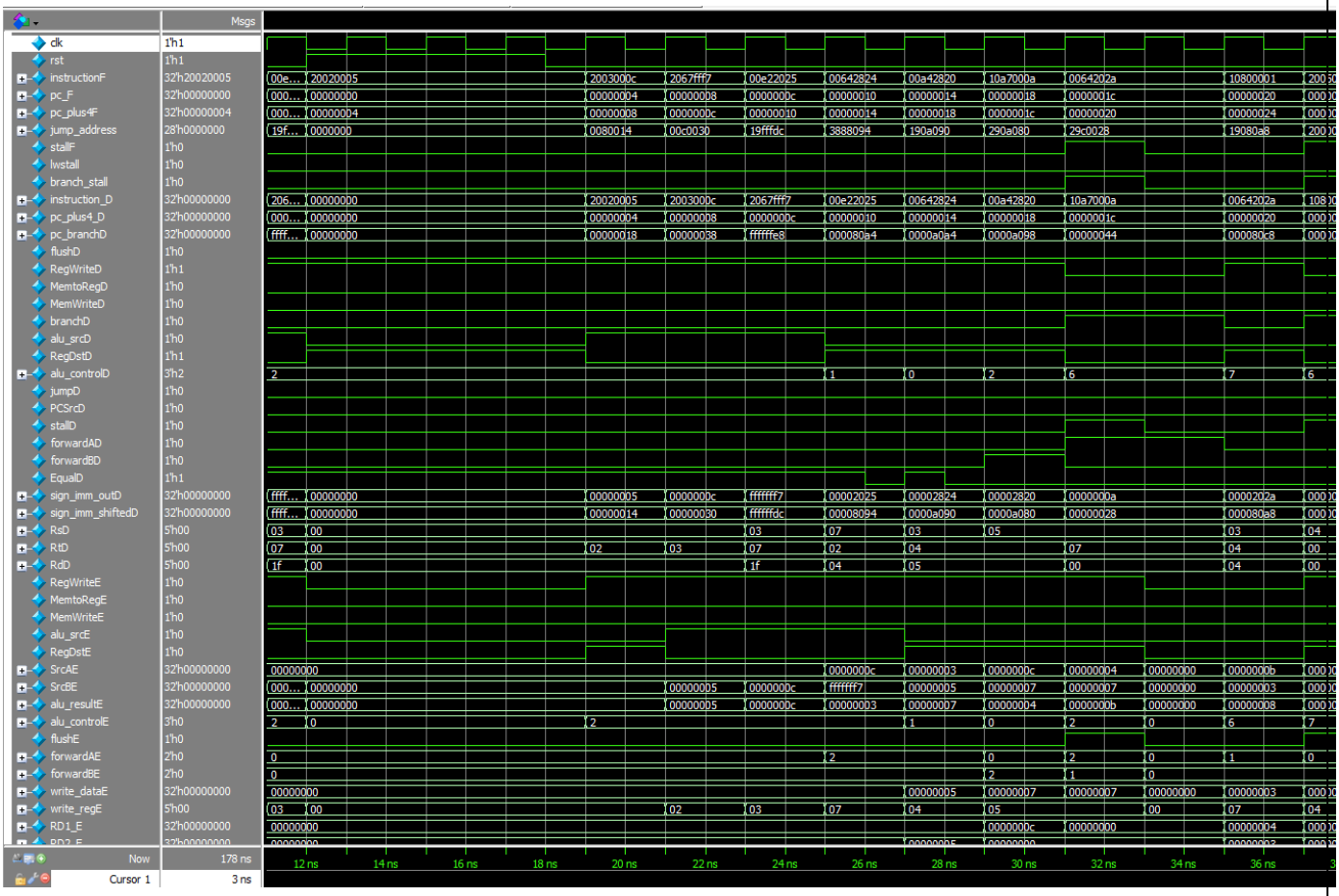
⇨ Control signals for each instruction:

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1

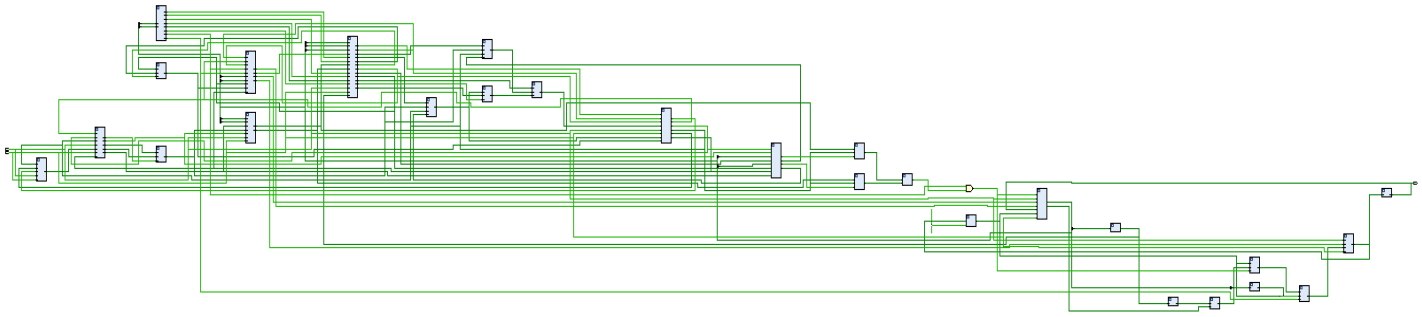
⇨ Design blocks:

- **PC:** Program Counter to track instruction addresses
- **ALU:** Arithmetic Logic Unit for executing ALU operations (add, sub, AND, OR, slt).
- **Control unit:** Decodes instructions and generates control signals.
- **Data memory:** Simulates read/write access to memory.
- **Instruction memory:** Stores the instruction set for simulation.
- **Mux:** Multiplexers used in various data path decision points.
- **Adder:** Performs PC + 4 or branch target calculation.
- **Register file:** Implements 32 general-purpose registers with read/write functionality.
- **Shift left2:** Shifts input by 2 bits (used in branch offset computation).
- **Shift left jump:** Shifts the jump target address.
- **Sign extend:** Sign extends 16-bit immediate to 32 bits.
- **IF/ID Register:** Fetch decode pipeline register.
- **ID/EX Register:** Decode execute pipeline register.
- **EX/MEM Register:** Execute memory pipeline register.
- **MEM/WB Register:** Memory write back pipeline register.
- **Hazard unit:** Detects and handles data/control hazards.
- **Forwarding unit:** Handles forwarding logic to prevent stalls.

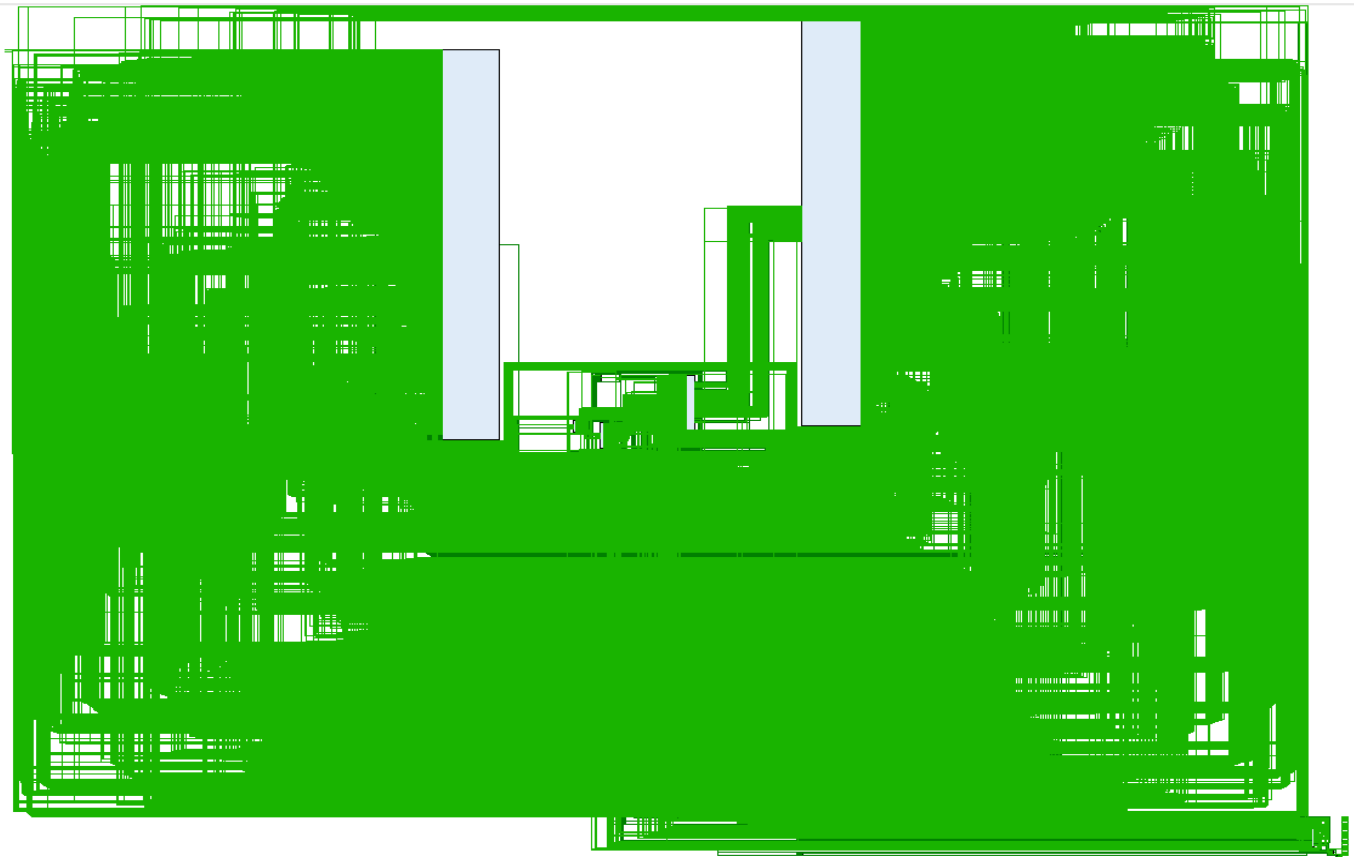
⇨ Waveform:



⇨ RTL Elaboration:



⇨ RTL Synthesis:



⇨ Timing analysis after synthesis:

Design Timing Summary

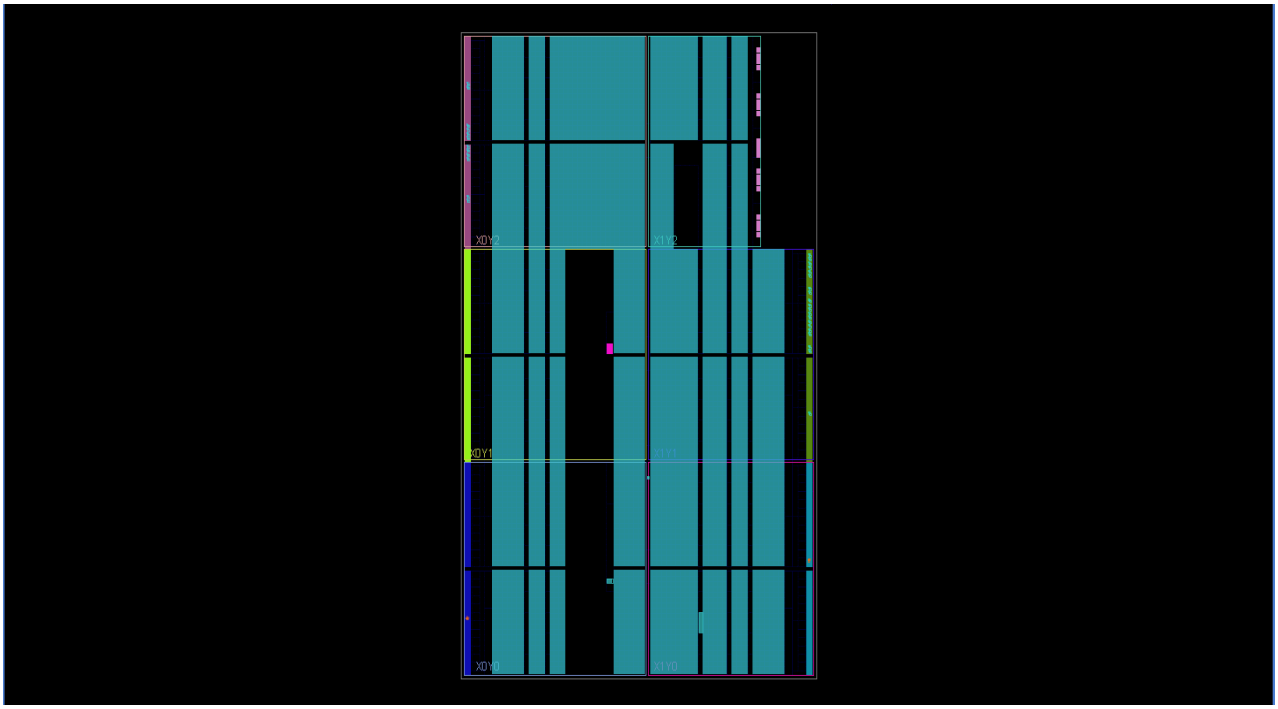
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.343 ns	Worst Hold Slack (WHS):	0.153 ns	Worst Pulse Width Slack (WPWS):	9.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	66860	Total Number of Endpoints:	66860	Total Number of Endpoints:	33829

All user specified timing constraints are met.

⇨ Utilization report after synthesis:

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Bonded IOB (106)	BUFGCTRL (32)
N mips	10643	33828	4401	2176	34	1
ALU_inst (ALU)	0	0	0	0	0	0
comparator_inst (com...	0	0	0	0	0	0
data_mem (data_me...	8736	32768	4352	2176	0	0
Dec_Ex (decode_exec...	319	89	0	0	0	0
EX_MEM (Execute_Me...	1274	637	0	0	0	0
F_D (fetch_decode)	107	30	0	0	0	0
inst_mem (instruction...	23	0	0	0	0	0
MEM_WB (Memory_Wr...	45	70	0	0	0	0
pc_add_4 (adder)	0	0	0	0	0	0
pc_branch_adder (add...	0	0	0	0	0	0
pc_inst (PC)	4	10	0	0	0	0
reg_file (register_file)	135	224	49	0	0	0

⇨ Implementation on FPGA:



⇨ Timing analysis after implementation:

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	2.504 ns	Worst Hold Slack (WHS):	0.043 ns	Worst Pulse Width Slack (WPWS):	8.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	70652	Total Number of Endpoints:	70636	Total Number of Endpoints:	35954

All user specified timing constraints are met.

⇨ Utilization report after implementation:

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	LUT Flip Flop Pairs (20800)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)	BSCANE2 (4)
N mips	11815	35754	4411	2176	8149	11701	114	904	1	34	2	1
ALU_inst (ALU)	0	0	0	0	12	0	0	0	0	0	0	0
comparator_inst (com...	0	0	0	0	3	0	0	0	0	0	0	0
data_mem (data_me...	8736	32768	4352	2176	7597	8736	0	0	0	0	0	0
dbg_hub (dbg_hub)	474	727	0	0	213	450	24	299	0	0	1	1
Dec_Ex (decode_exec...	319	89	0	0	155	319	0	38	0	0	0	0
EX_MEM (Execute_Me...	1272	637	0	0	819	1272	0	65	0	0	0	0
F_D (fetch_decode)	107	30	0	0	61	107	0	26	0	0	0	0
inst_mem (instruction...	23	0	0	0	15	23	0	0	0	0	0	0
MEM_WB (Memory_Wr...	45	70	0	0	65	45	0	0	0	0	0	0
pc_add_4 (adder)	0	0	0	0	3	0	0	0	0	0	0	0
pc_branch_adder (add...	0	0	0	0	3	0	0	0	0	0	0	0
pc_inst (PC)	4	10	0	0	6	4	0	0	0	0	0	0
reg_file (register_file)	135	224	49	0	94	135	0	0	0	0	0	0
u_ila_0 (u_ila_0)	701	1199	10	0	273	611	90	364	1	0	0	0

⇨ MIPS assembly testing code:

```
#   Assembly code              Description                    Address(Hexa)   Machine code
main:   addi $2, $0, 5         # initialize $2 = 5                0           20020005
    addi $3, $0, 12            # initialize $3 = 12               4           2003000c
    addi $7, $3, -9            # initialize $7 = 3                8           2067fff7
    or   $4, $7, $2            # $4 <= 3 or 5 = 7                 c           00e22025
    and $5,  $3, $4            # $5 <= 12 and 7 = 4               10          00642824
    add $5,  $5, $4            # $5 = 4 + 7 = 11                  14          00a42820
    beq $5,  $7, end           # shouldn't be taken               18          10a7000a
    slt $4,  $3, $4            # $4 = 12 < 7 = 0                  1c          0064202a
    beq $4,  $0, around        # should be taken                  20          10800001
    addi $5, $0, 0             # shouldn't happen                 24          20050000
around: slt $4,  $7, $2        # $4 = 3 < 5 = 1                   28          00e2202a
    add $7,  $4, $5            # $7 = 1 + 11 = 12                 2c          00853820
    sub $7,  $7, $2            # $7 = 12 - 5 = 7                  30          00e23822
    sw   $7, 68($3)            # mem[80] = 7                      34          ac670044
    lw   $2, 80($0)            # $2 = mem[80] = 7                 38          8c020050
    j end                      # should be taken                  3c          08000011
    addi $2, $0, 1             # shouldn't happen                 40          20020001
end:    sw   $2, 84($0)        # write adress [84] = 7            44          ac020054
```

- The following assembly test code is used to verify the design and functionality of the single cycle and pipelined MIPS processors implemented in Verilog.
- This code is loaded to the instruction memory