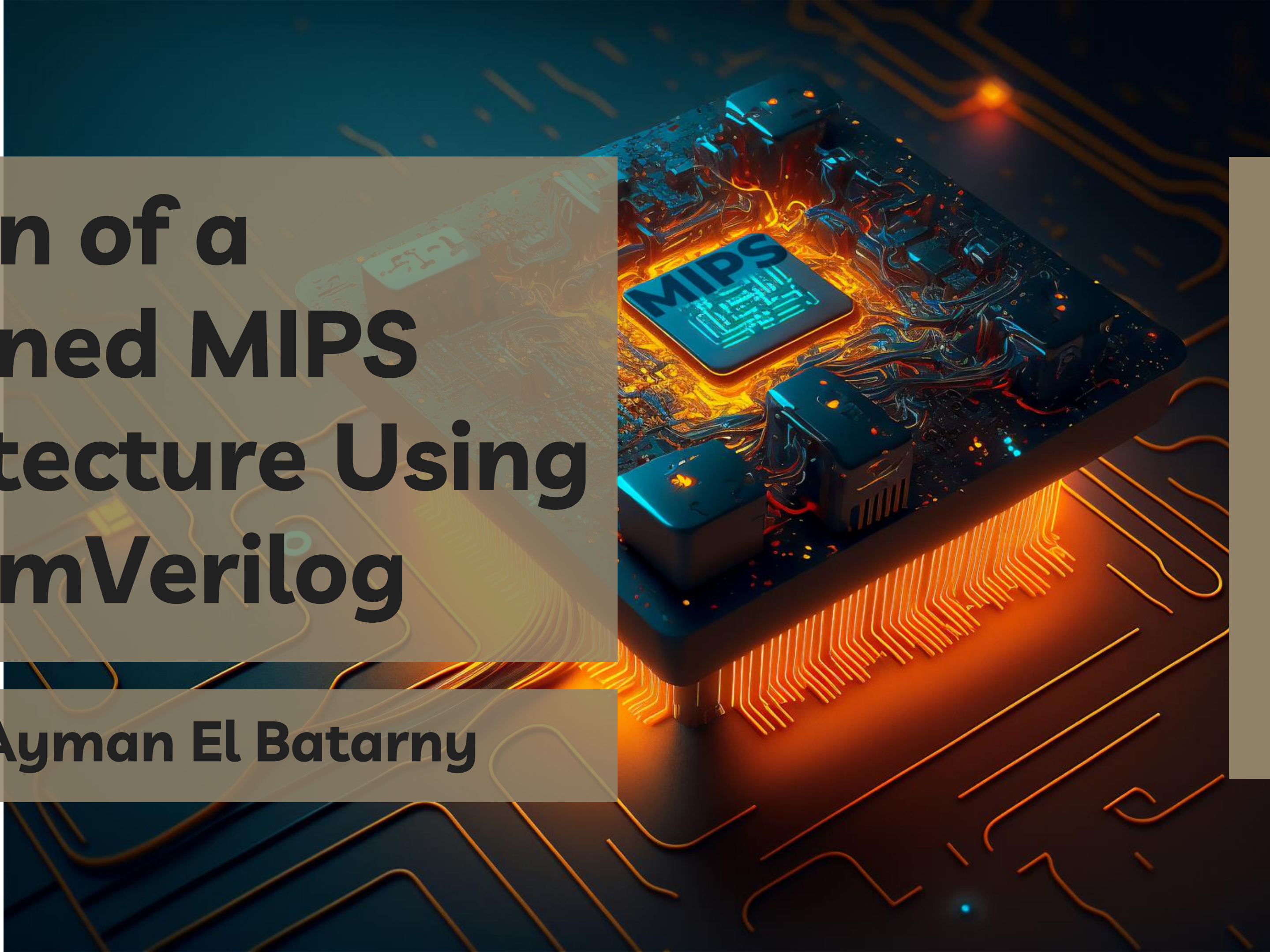


# Design of a Pipelined MIPS Architecture Using SystemVerilog

By: Amr Ayman El Batarny

## TABLE OF CONTENTS

## TABLE OF CONTENTS

## 1. Introduction: Designing and Implementing a Pipelined MIPS Architecture Using SystemVerilog

This document covers the design and implementation of a pipelined MIPS architecture using SystemVerilog, a type of Hardware Description Language (HDL). We will use ModelSim for simulation to ensure our design works correctly and Quartus for hardware synthesis, place-and-route (PnR) and timing analysis to turn our SystemVerilog code into a real hardware design.

We'll start with the basic single-cycle MIPS model, examining its limitations and why we need to optimize performance. This will lead us to pipelining, a method that speeds up instruction execution and boosts performance.

Our journey begins with the single-cycle model and gradually moves to the pipelined version. This step-by-step approach will help you understand how pipelining changes the MIPS architecture using SystemVerilog code.

We'll look into the details of the pipelined datapath and control unit, both designed in SystemVerilog. We'll address challenges like data hazards and how to handle them with techniques like forwarding and stalling. Control hazards will also be covered, including branch prediction and stalling to keep the pipeline running smoothly.

Additionally, we'll explore how exceptions are managed in the pipelined framework in SystemVerilog to maintain processor integrity.

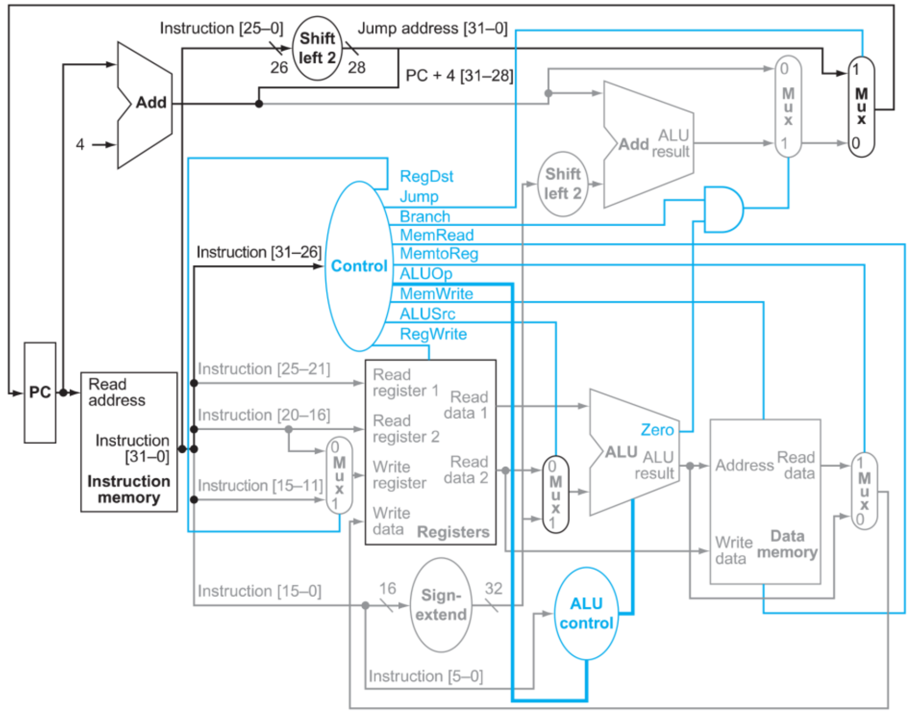
Finally, we'll provide a complete schematic for the pipelined MIPS processor design. To ensure you understand and can validate the design, we'll use ModelSim for simulation and Quartus for synthesis, fitting and timing analysis, transforming the SystemVerilog code into a hardware design ready for implementation.

## 2. A Step-by-Step Approach to Pipelined MIPS: Building on the Single-Cycle Model

The single-cycle MIPS design completes each instruction in one clock cycle, handling all stages from fetch to write-back within that cycle. Below are its key components and operations:

**Key Components**

- Program Counter (PC): Holds the current instruction address.
- Instruction Memory: Stores and provides instructions.
- Register File: General-purpose registers with two read ports and one write port.
- ALU (Arithmetic Logic Unit): Performs arithmetic and logic operations.
- Data Memory: Stores data with read/write access.
- Control Unit: Generates control signals for the datapath.
- Multiplexers: Select data inputs based on control signals.
- Sign-Extension Unit: Extends immediate values to 32 bits.

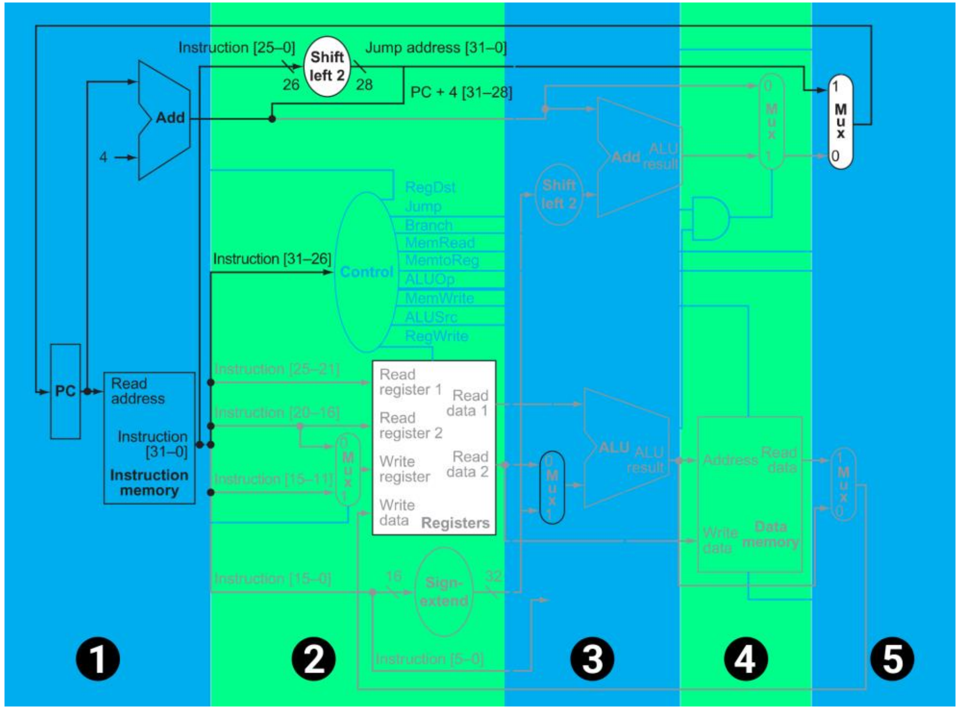


**Patterson and Hennessy's Computer Organization and Design, 5th Ed.**

## 2. A Step-by-Step Approach to Pipelined MIPS: Building on the Single-Cycle Model

**Operation Steps**

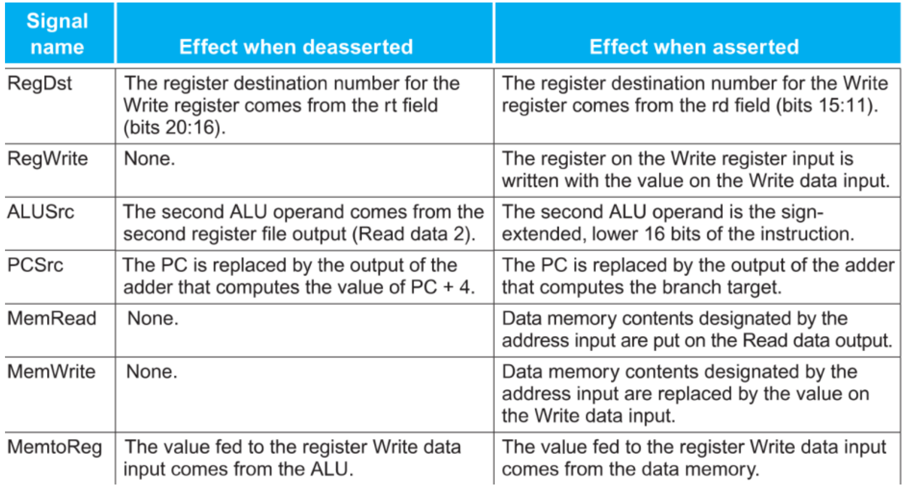
1. Instruction Fetch (IF): Fetch instruction using the PC.
2. Instruction Decode (ID): Decode instruction, read registers, and extend immediate values.
3. Execution (EX): ALU performs operations.
4. Memory Access (MEM): Access data memory for load/store instructions.
5. Write Back (WB): Write results back to the register file.



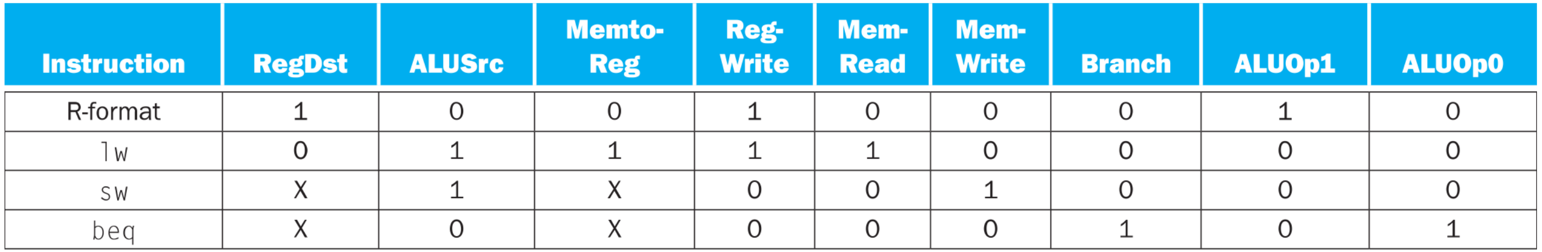
## 2. A Step-by-Step Approach to Pipelined MIPS: Building on the Single-Cycle Model

**Control Signals**

- RegDst: Destination register.
- 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.



| Signal name | Effect when deasserted | Effect when asserted |
|---|---|---|
| RegDst | The register destination number for the Write register comes from the rt field (bits 20:16). | The register destination number for the Write register comes from the rd field (bits 15:11). |
| RegWrite | None. | The register on the Write register input is written with the value on the Write data input. |
| ALUSrc | The second ALU operand comes from the second register file output (Read data 2). | The second ALU operand is the sign-extended, lower 16 bits of the instruction. |
| PCSrc | The PC is replaced by the output of the adder that computes the value of PC + 4. | The PC is replaced by the output of the adder that computes the branch target. |
| MemRead | None. | Data memory contents designated by the address input are put on the Read data output. |
| MemWrite | None. | Data memory contents designated by the address input are replaced by the value on the Write data input. |
| MemtoReg | The value fed to the register Write data input comes from the ALU. | The value fed to the register Write data input comes from the data memory. |



| Instruction | RegDst | ALUSrc | Memto-Reg | Reg-Write | Mem-Read | Mem-Write | Branch | ALUOp1 | ALUOp0 |
|---|---|---|---|---|---|---|---|---|---|
| R-format | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| `lw` | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| `sw` | X | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 |
| `beq` | X | 0 | X | 0 | 0 | 0 | 1 | 0 | 1 |

**Patterson and Hennessy's Computer Organization and Design, 5th Ed.**

## 2. A Step-by-Step Approach to Pipelined MIPS: Building on the Single-Cycle Model

**ALU Control**

The ALU control unit decodes instructions and generates control signals for the ALU based on the instruction type and function code. It uses a multi-level decoding approach to reduce size and potentially increase speed. A truth table with don't-care terms is used to simplify the design.

| Instruction opcode | ALUOp | Instruction operation | Funct field | Desired ALU action | ALU control input |
|---|---|---|---|---|---|
| LW | 00 | load word | XXXXXX | add | 0010 |
| SW | 00 | store word | XXXXXX | add | 0010 |
| Branch equal | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| R-type | 10 | subtract | 100010 | subtract | 0110 |
| R-type | 10 | AND | 100100 | AND | 0000 |
| R-type | 10 | OR | 100101 | OR | 0001 |
| R-type | 10 | set on less than | 101010 | set on less than | 0111 |

**Patterson and Hennessy's Computer Organization and Design, 5th Ed.**

## 3. Single-Cycle Implementation Inefficiencies: From Single-Cycle Limitations to Pipelined Performance

- In a single-cycle design, every instruction has the same clock cycle length, resulting in a CPI (Cycles Per Instruction) of 1.
- The clock cycle is determined by the critical path, typically a load instruction involving instruction memory, register file (read), ALU, data memory, and register file (write).
- The clock cycle is set to the worst-case delay, making common-case optimizations ineffective and violating the principle of optimizing for the common case.
- Despite a CPI of 1, single-cycle performance is poor due to a long clock cycle, and many instructions could fit in a shorter cycle.
- Complex operations and addressing modes can cause significant instruction delay variations.
- Single-cycle design requires each functional unit to be used once per cycle, necessitating unit duplication, increasing cost, and wasting area.
- Thus, single-cycle implementation is inefficient in performance and hardware cost.
- The penalty for a fixed clock cycle in single-cycle design is significant.
- Solutions include:
    - ❑ Variable-length clocks, which are difficult and may have more overhead than benefit.
    - ❑ Multi-cycle techniques with shorter cycles, varying the number of cycles per instruction class.
    - ❑ Pipelining, which uses a similar datapath but overlaps instruction execution, increasing efficiency and performance.

## 4. Speeding Up with Pipelines: A Look at Parallel Processing

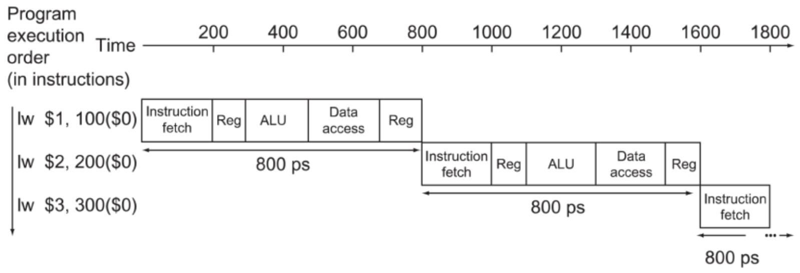
Pipelining is a technique used to increase the throughput of a processor by allowing multiple instructions to overlap in execution. Instead of waiting for one instruction to complete before starting the next, pipelining divides the instruction execution process into several stages, each handled by different hardware units working in parallel. This approach is analogous to an assembly line in manufacturing, where each stage of the pipeline processes a different instruction simultaneously.

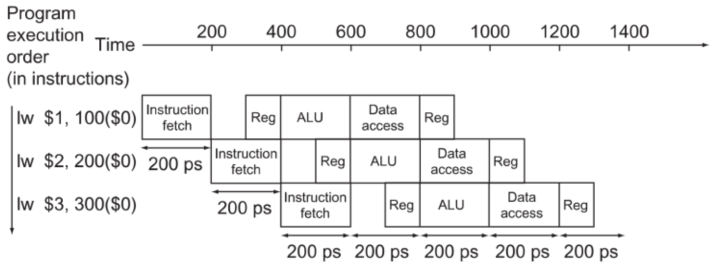
**Stages of the Pipeline**

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.

Each stage operates on a different instruction in the pipeline, allowing multiple instructions to be in different stages of execution at the same time.





**Patterson and Hennessy's Computer Organization and Design, 5th Ed.**

## 5. Pipelined Datapath: Building the datapath of the pipelined model

The pipelined datapath in MIPS architecture represents a streamlined approach to executing instructions by breaking down the instruction execution process into sequential stages. Each stage handles a specific part of the instruction, allowing multiple instructions to overlap in execution. Here’s an overview of the pipelined datapath in MIPS:

**Key Components of the Pipelined Datapath**

1. Instruction Fetch (IF):

Fetches the next instruction from memory based on the Program Counter (PC).

2. Instruction Decode (ID):

Decodes the instruction fetched in the previous cycle.

Reads necessary register values from the register file.

3. Execute (EX):

Performs arithmetic or logical operations specified by the instruction.

Computes memory addresses for load/store instructions.

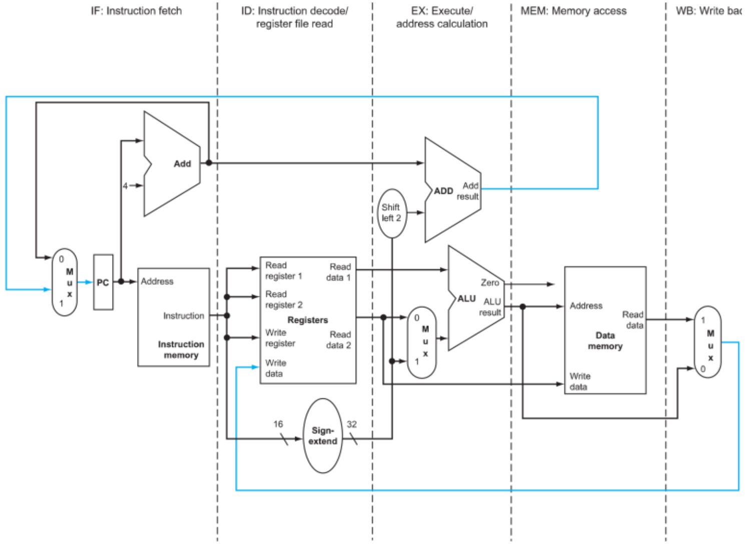
4. Memory Access (MEM):

Accesses data memory for load/store instructions.

Reads data from or writes data to memory.

5. Write Back (WB):

Writes the results of the instruction back to the register file.



## 5. Pipelined Datapath: Building the datapath of the pipelined model

**Pipeline Registers in MIPS Datapath**

1. IF/ID Register (Instruction Fetch/Decode Register):

Function: Holds the instruction fetched from memory during the Instruction Fetch (IF) stage.
Contents: Includes the fetched instruction opcode and any associated data, ready for decoding.

2. ID/EX Register (Instruction Decode/Execute Register):

Function: Stores information from the ID stage and prepares it for execution in the Execute (EX) stage.
Contents: Includes decoded instruction fields, such as control signals, register operands, and immediate values.

3. EX/MEM Register (Execute/Memory Access Register):

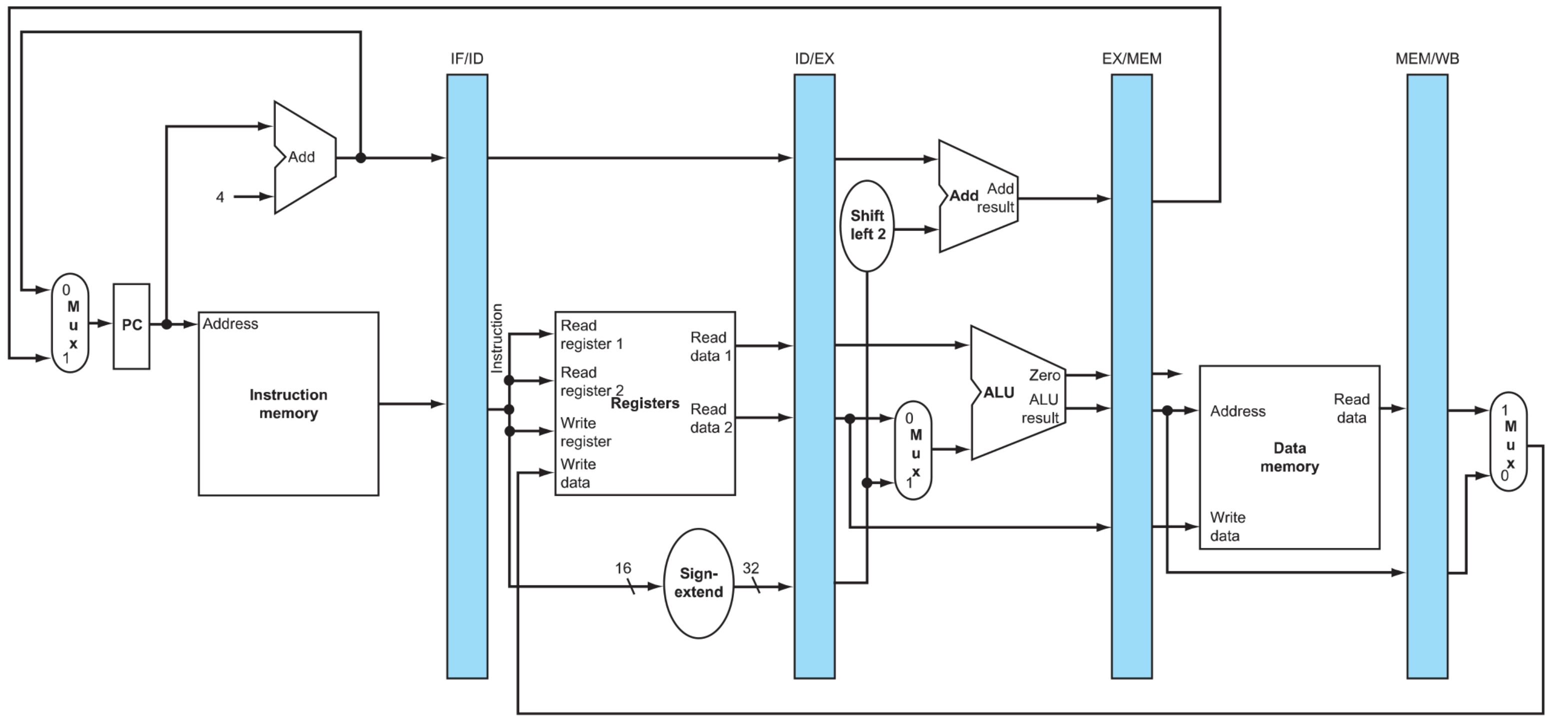
Function: Temporarily holds results from the EX stage and prepares for memory access.
Contents: Includes ALU results, memory addresses for load/store instructions, and control signals for memory operations.

4. MEM/WB Register (Memory Access/Write Back Register):

Function: Stores data retrieved from memory during the MEM stage and prepares for register write-back.
Contents: Includes data read from memory or data to be written back to registers.

# 5. Pipelined Datapath: Building the datapath of the pipelined model

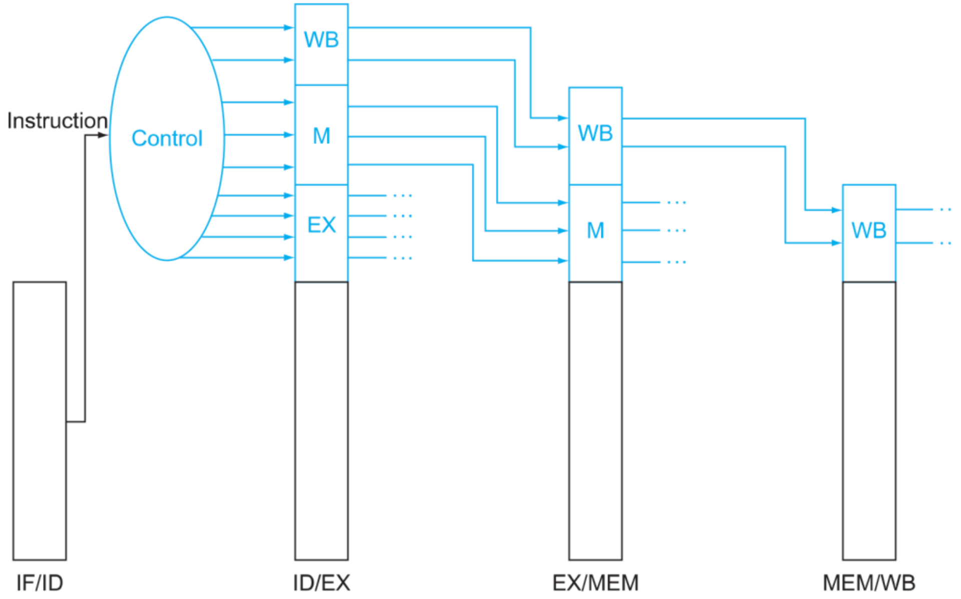
### Pipeline Registers in MIPS Datapath



**Patterson and Hennessy's Computer Organization and Design, 5th Ed.**

## 6. Pipelined Control: Building the control of the pipelined model

To specify control for the pipeline, we need only set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage, we can divide the control lines into three groups according to the pipeline stage.



**Patterson and Hennessy's Computer Organization and Design, 5th Ed.**

## 6. Pipelined Control: Building the control of the pipelined model

1. Instruction fetch: The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.
2. Instruction decode/register file read: As in the previous stage, the same thing happens at every clock cycle, so there are no optional control lines to set.
3. Execution/address calculation: The signals to be set are RegDst, ALUOp, and ALUSrc. The signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate for the ALU.
4. Memory access: The control lines set in this stage are Branch, MemRead, and MemWrite. The branch equal, load, and store instructions set these signals, respectively. Recall that PCSrc in Figure 4.48 selects the next sequential address unless control asserts Branch and the ALU result was 0.
5. Write-back: The two control lines are MemtoReg, which decides between sending the ALU result or the memory value to the register file, and RegWrite, which writes the chosen value.

| | Execution/address calculation stage control lines | | | | Memory access stage control lines | | | Write-back stage control lines | |
|---|---|---|---|---|---|---|---|---|---|
| Instruction | RegDst | ALUOp1 | ALUOp0 | ALUSrc | Branch | Mem-Read | Mem-Write | Reg-Write | Memto-Reg |
| R-format | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| sw | X | 0 | 0 | 1 | 0 | 0 | 1 | 0 | X |
| beq | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 | X |

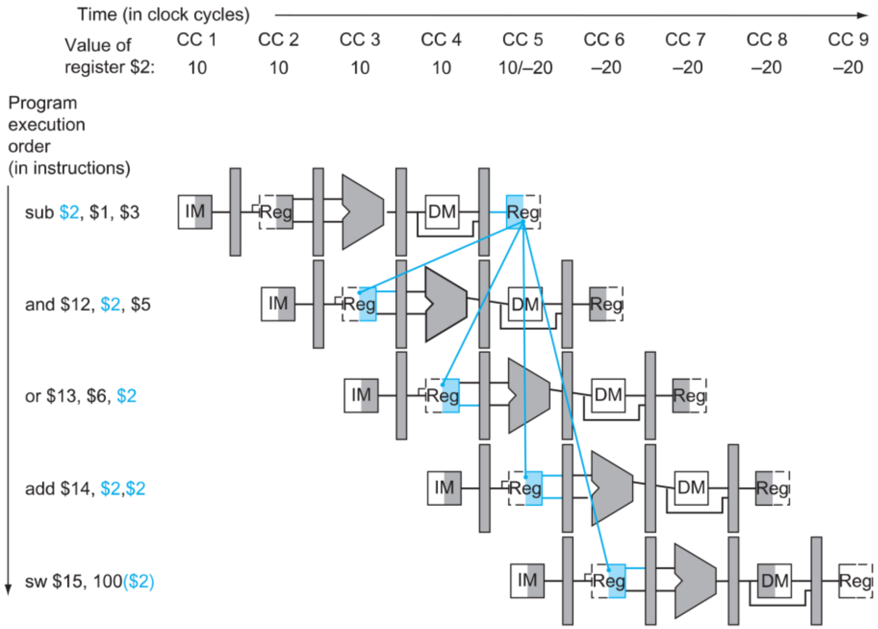
**Patterson and Hennessy's Computer Organization and Design, 5th Ed.**

## 7. Forwarding vs. Stalling: Strategies for Mitigating Data Hazards in Pipelines

Let's look at a sequence with many dependences, shown in color:

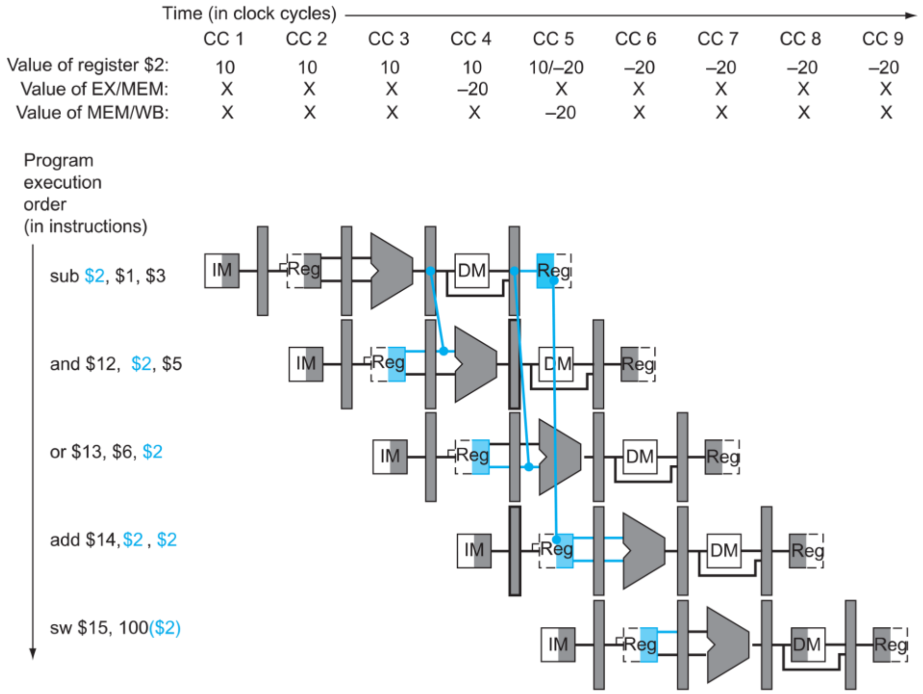
```
sub   $2, $1,$3        # Register $2 written by sub
and   $12,$2,$5        # 1st operand($2) depends on sub
or    $13,$6,$2        # 2nd operand($2) depends on sub
add   $14,$2,$2        # 1st($2) & 2nd($2) depend on sub
sw    $15,100($2)      # Base ($2) depends on sub
```

The last four instructions are all dependent on the result in register $2 of the first instruction. If register $2 had the value 10 before the subtract instruction and −20 afterwards, the programmer intends that −20 will be used in the following instructions that refer to register $2.



## 7. Forwarding vs. Stalling: Strategies for Mitigating Data Hazards in Pipelines

How does forwarding work? For simplicity in the rest of this section, we consider only the challenge of forwarding to an operation in the EX stage, which may be either an ALU operation or an effective address calculation. This means that when an instruction tries to use a register in its EX stage that an earlier instruction intends to write in its WB stage, we actually need the values as inputs to the ALU.


Time (in clock cycles)

| | CC 1 | CC 2 | CC 3 | CC 4 | CC 5 | CC 6 | CC 7 | CC 8 | CC 9 |
|---|---|---|---|---|---|---|---|---|---|
| Value of register $2: | 10 | 10 | 10 | 10 | 10/–20 | –20 | –20 | –20 | –20 |
| Value of EX/MEM: | X | X | X | –20 | X | X | X | X | X |
| Value of MEM/WB: | X | X | X | X | –20 | X | X | X | X |

Patterson and Hennessy’s Computer Organization and Design, 5th Ed.

## 7. Forwarding vs. Stalling: Strategies for Mitigating Data Hazards in Pipelines

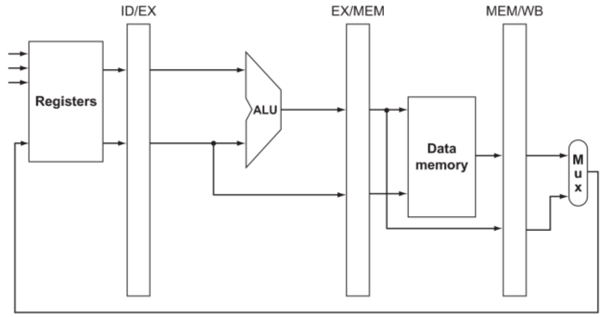
A notation that names the fi elds of the pipeline registers allows for a more precise notation of dependences. For example, “ID/EX.RegisterRs” refers to the number of one register whose value is found in the pipeline register ID/EX; that is, the one from the first read port of the register file. The first part of the name, to the left of the period, is the name of the pipeline register; the second part is the name of the field in that register. Using this notation, the two pairs of hazard conditions are:

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

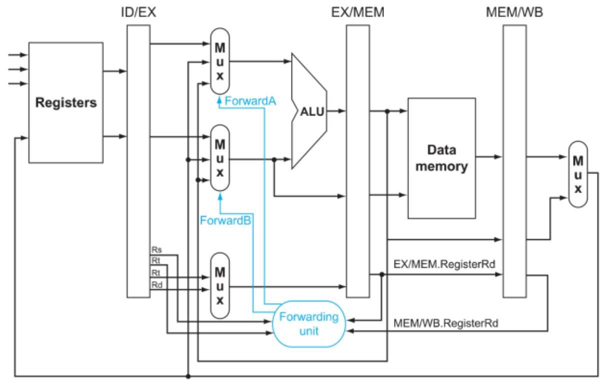
Let’s now write both the conditions for detecting hazards and the control signals to resolve them:

1. EX hazard:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```



a. No forwarding

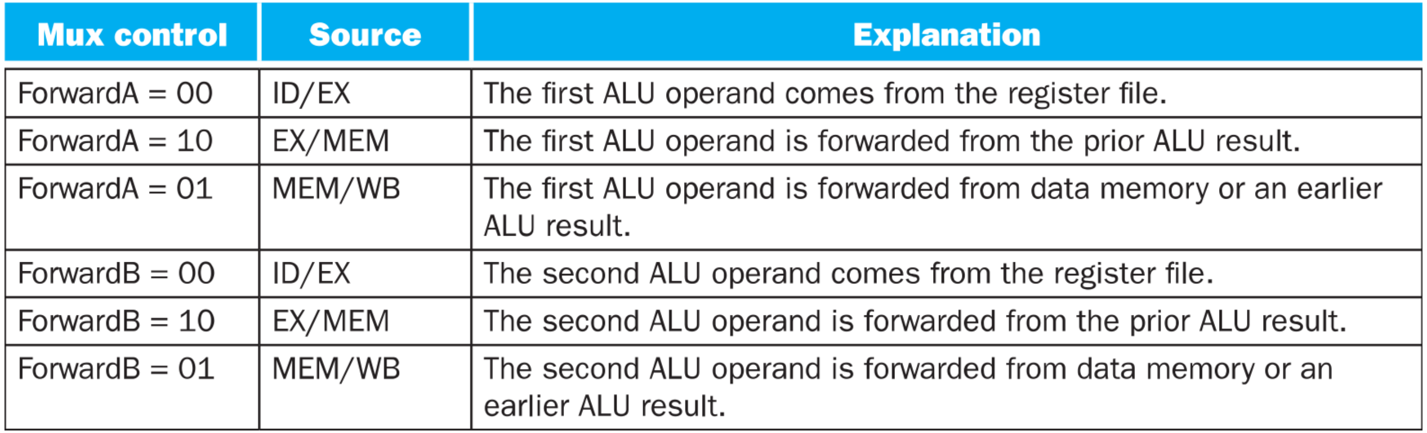


b. With forwarding

Patterson and Hennessy’s Computer Organization and Design, 5th Ed.

## 7. Forwarding vs. Stalling: Strategies for Mitigating Data Hazards in Pipelines

```
2. MEM hazard:
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠  0)
and ( MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠  0)
and  (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```



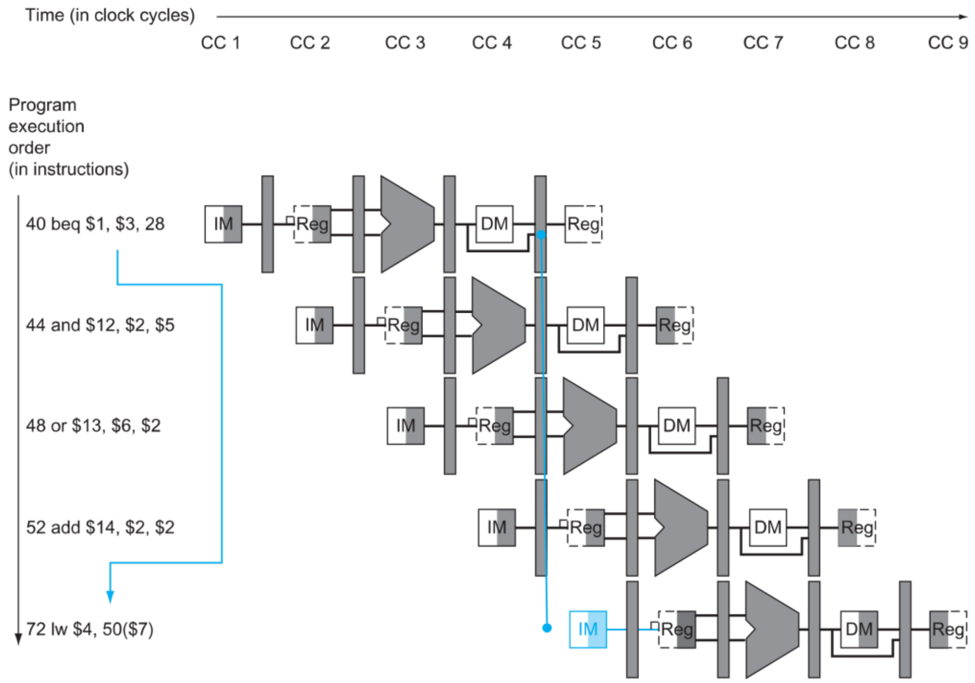
| Mux control | Source | Explanation |
|---|---|---|
| ForwardA = 00 | ID/EX | The first ALU operand comes from the register file. |
| ForwardA = 10 | EX/MEM | The first ALU operand is forwarded from the prior ALU result. |
| ForwardA = 01 | MEM/WB | The first ALU operand is forwarded from data memory or an earlier ALU result. |
| ForwardB = 00 | ID/EX | The second ALU operand comes from the register file. |
| ForwardB = 10 | EX/MEM | The second ALU operand is forwarded from the prior ALU result. |
| ForwardB = 01 | MEM/WB | The second ALU operand is forwarded from data memory or an earlier ALU result. |

## 8. Addressing Branching Challenges: Prediction vs. Stalling for Control Hazards

Control hazards are simpler and less frequent than data hazards, allowing us to use simpler resolution schemes. Here, we discuss two schemes for handling control hazards and an optimization to improve them.

**Assume Branch Not Taken:**

- Stalling for branches is too slow. Instead, predict branches as not taken and continue down the sequential path.
- If the branch is taken, discard the fetched and decoded instructions.
- To discard instructions, set control values to 0s, similar to load-use stalls. This must be done for instructions in the IF, ID, and EX stages when the branch reaches MEM.



**Patterson and Hennessy's Computer Organization and Design, 5th Ed.**

## 8. Addressing Branching Challenges: Prediction vs. Stalling for Control Hazards

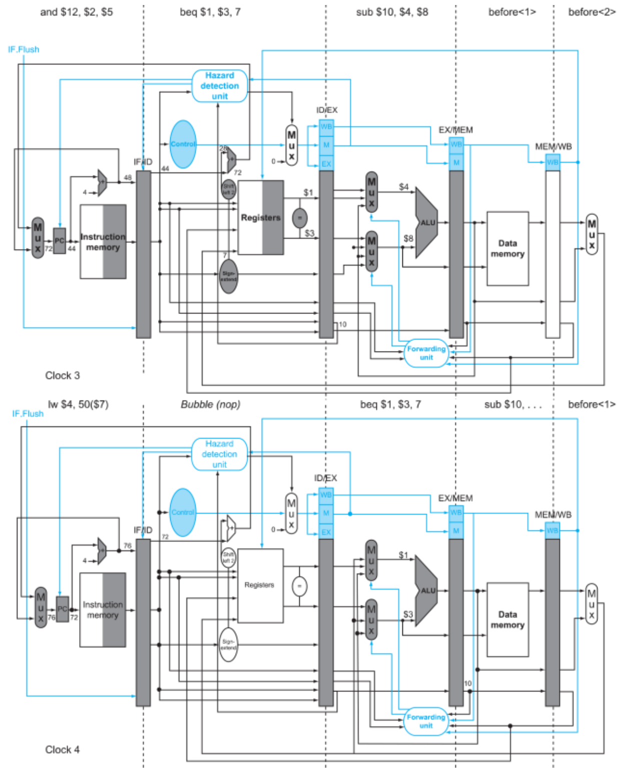
**Branch Decision:**

- For branch-equal, compare two registers read during the ID stage.
- This requires additional forwarding and hazard detection hardware to ensure branches dependent on pipeline results work correctly.
- New forwarding logic is needed for the equality test unit in ID, handling operands from ALU/MEM or MEM/WB pipeline latches.

**Handling Data Hazards in Branches:**

- If an ALU instruction immediately precedes a branch, a stall is required since the EX stage for the ALU instruction occurs after the ID cycle of the branch.
- If a load instruction is followed by a branch dependent on the load result, two stall cycles are needed, as the load result appears at the end of MEM but is needed at the start of ID for the branch.

Despite the difficulties, moving branch execution to the ID stage reduces branch penalties to one instruction if taken. An IF.Flush control line zeros the instruction field of the IF/ID pipeline register, turning the fetched instruction into a no-op (nop), which does nothing and changes no state.



**Patterson and Hennessy's Computer Organization and Design, 5th Ed.**

## 9. Maintaining Processor Integrity: How Exceptions are Managed in Pipelines

**Handling Exceptions in MIPS**
In MIPS architecture, exceptions like undefined instructions and arithmetic overflow are handled by saving the offending instruction's address in the Exception Program Counter (EPC) and transferring control to the operating system.

| Exception type | Exception vector address (in hex) |
|---|---|
| Undefined instruction | $8000\ 0000_{hex}$ |
| Arithmetic overflow | $8000\ 0180_{hex}$ |

**Key Steps:**
1. Saving EPC: The processor saves the address of the offending instruction in the EPC.
2. Transferring Control: Control is transferred to the operating system at a specified address (e.g., 8000 0180hex).
3. Handling Exceptions: The operating system takes appropriate action, possibly continuing or terminating the program using the EPC to restart if needed.
4. Communicating Cause: The reason for the exception is stored in the Cause register or handled via vectored interrupts.

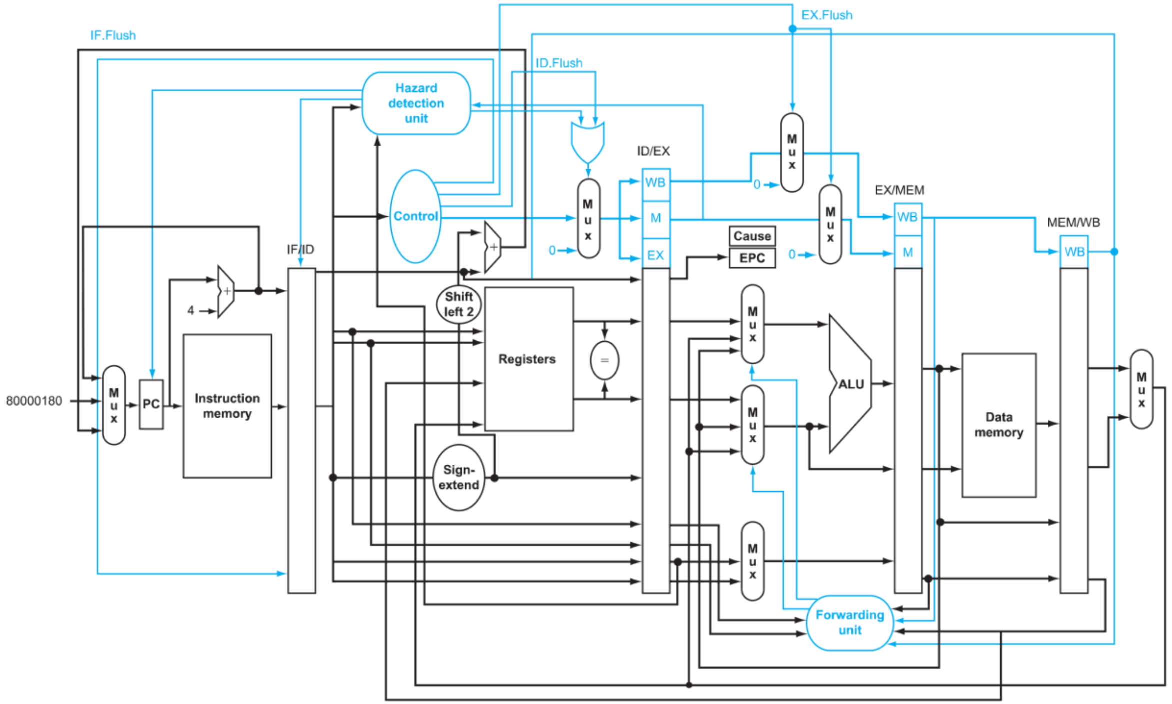
**Required Additions:**
- EPC: A 32-bit register for the instruction's address.
- Cause Register: Records the cause of the exception.

# 9. Maintaining Processor Integrity: How Exceptions are Managed in Pipelines

**Pipeline Considerations:**

- Flushing Instructions: Similar to handling branch mispredictions, instructions after the exception are flushed using signals like ID.Flush and EX.Flush.
- Fetching from Exception Address: The PC multiplexer is updated to fetch from the exception address (8000 0180hex).
- Preventing Overwrites: During exceptions, the instruction in the EX stage is prevented from writing to the WB stage to avoid overwriting important values.

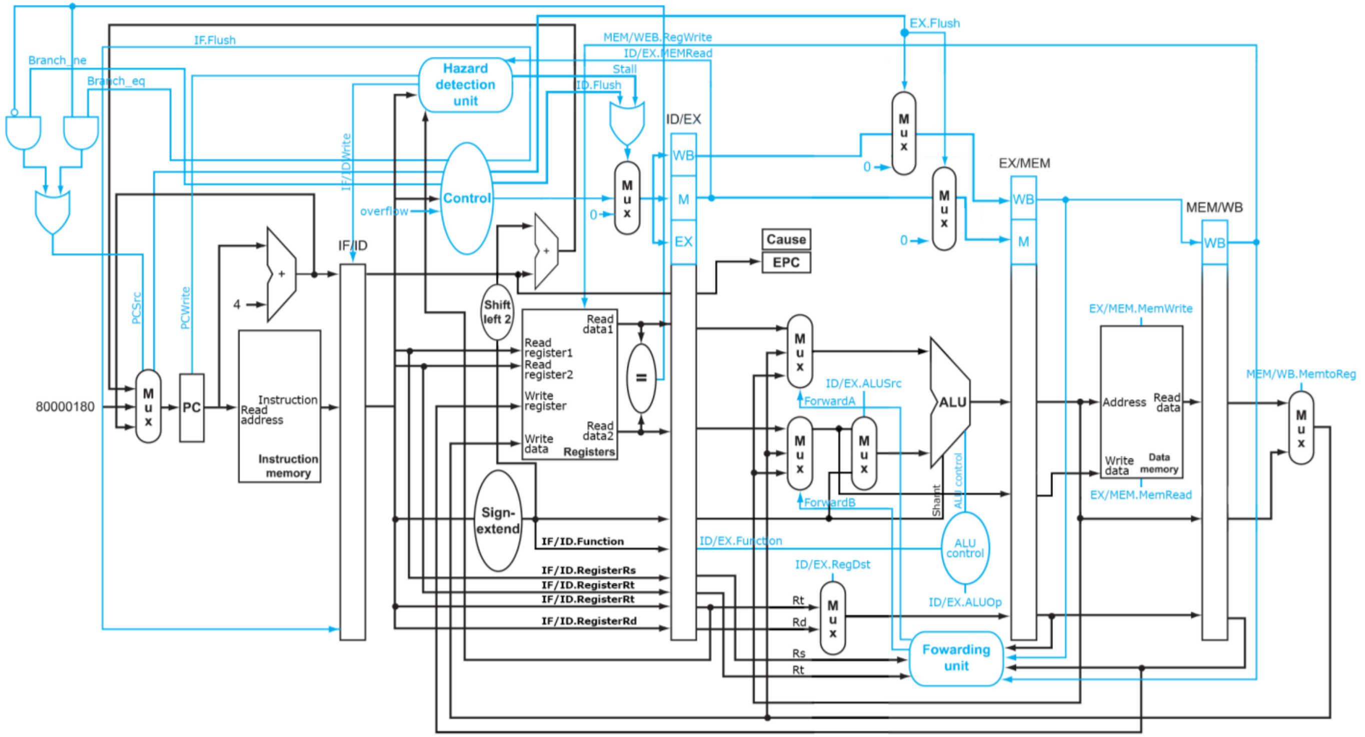
By carefully planning and using additional control signals and registers, exceptions are managed efficiently, ensuring the system can handle errors and continue or stop execution as necessary.



Patterson and Hennessy’s Computer Organization and Design, 5th Ed.

## 10. Unifying the Design: Complete Pipelined MIPS Processor Schematic

**Here's the full pipelined MIPS architecture which supports instructions: R-Format(arithmetic), addi, sw & lw, beq & bne.**



**Patterson and Hennessy's Computer Organization and Design, 5th Ed.**

## 11. Verifying Functionality: MIPS Assembly Test Code loaded into the instruction memory

The following assembly test code is used to verify the design and functionality of the pipelined MIPS processor implemented in SystemVerilog. This test program initializes an array with specific values, performs a summation of the array elements, and stores the result back into memory. The machine code of each instruction is provided at the end of each line.

```
addi $t0, $zero, 0x0020     #$t0 <> base address                0x20080020
addi $t1, $zero, 0x0013     #$t0 <= 0x0013                      0x20090013
addi $t2, $zero, 0x0095     #$t1 <= 0x0095                      0x200A0095
addi $t3, $zero, 0x0018     #$t2 <= 0x0018                      0x200B0018
addi $t4, $zero, 0x0003     #$t3 <= 0x0003                      0x200C0003
addi $t5, $zero, 0x0063     #$t4 <= 0x0063                      0x200D0063
addi $t6, $zero, 0x0047     #$t5 <= 0x0047                      0x200E0047
addi $t7, $zero, 0x0005     #$t6 <> counter                     0x200F0005
addi $s7, $zero, 0x0000     #$s7 <> accumulator                 0x20170000
sw   $t1, 0x0000($t0)       #array[0] = 0x0013                  0xAD090000
sw   $t2, 0x0004($t0)       #array[1] = 0x0095                  0xAD0A0004
sw   $t3, 0x0008($t0)       #array[2] = 0x0018                  0xAD0B0008
sw   $t4, 0x000C($t0)       #array[3] = 0x0003                  0xAD0C000C
sw   $t5, 0x0010($t0)       #array[4] = 0x0063                  0xAD0D0010
sw   $t6, 0x0014($t0)       #array[5] = 0x0047                  0xAD0E0014
sll  $s1, $t7, 0x0002       #$s1 <= ($t7) * 4                   0x000F8880
add  $s1, $s1, $t0          #$s1 <= ($s1) + ($t0)               0x02288820
lw   $s2, 0x0000($s1)       #$s2 <= array[counter]              0x8E320000
add  $s7, $s7, $s2          #$s7 <= ($s7) + ($s2)               0x02F2B820
addi $t7, $t7, 0xFFFF       #$t7 <= ($t7) + (-1)                0x21EFFFFF
bne  $t7, $zero, 0xFFFA     #if(($t7) == 0) goto (PC+4-6*4)     0x15E0FFFA
sw   $s7, 0x0018($t0)       #array[6] <= ($s7)                  0xAD170018
```

## 11. Verifying Functionality: MIPS Assembly Test Code loaded into the instruction memory

**Purpose of the Code**

The purpose of this assembly test code is to:

- Verify the correctness of data transfer instructions (load and store).
- Test arithmetic operations (addition).
- Ensure proper branching and loop execution.
- Ensure that the forwarding unit and hazard detection unit are working properly by including data and control hazards in the code.
- Validate the overall functionality of the pipelined MIPS processor design by simulating a simple array summation program.

By running this test code, we can observe the behaviour of the processor and check for correct execution of instructions, handling of data hazards, and correct implementation of the control flow within a pipelined architecture.

## 11. Verifying Functionality: MIPS Assembly Test Code loaded into the instruction memory

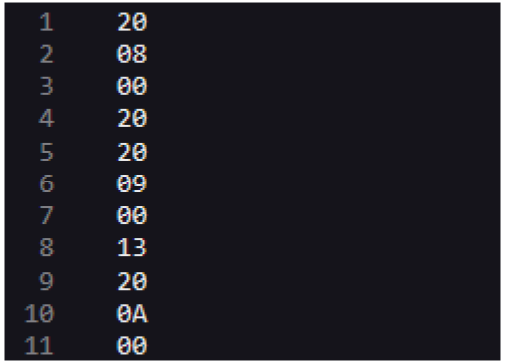
**Loading the instructions to the instruction memory**

Three `.dat` files are created to simulate the register file, data memory, and instruction memory. These files play crucial roles in loading and running the test code instructions for the MIPS processor simulation. Here's an explanation of the files:

- Each line in these files corresponds to a memory location, with locations ordered from 0 to 1023, from the top of the file to the bottom.
- The data is written in hexadecimal representation.
- Since MIPS memory is byte-addressable, each line in the `.dat` file represents the content of one memory location as a byte.

**InstructionMemory.dat Snippet**

Here is an example snippet from the InstructionMemory.dat file, showing the loaded instructions in hexadecimal format:



```
20
08
00
20
20
09
00
13
20
0A
00
```

## 12. Simulation Results (ModelSim): Simulating the RTL design using ModelSim

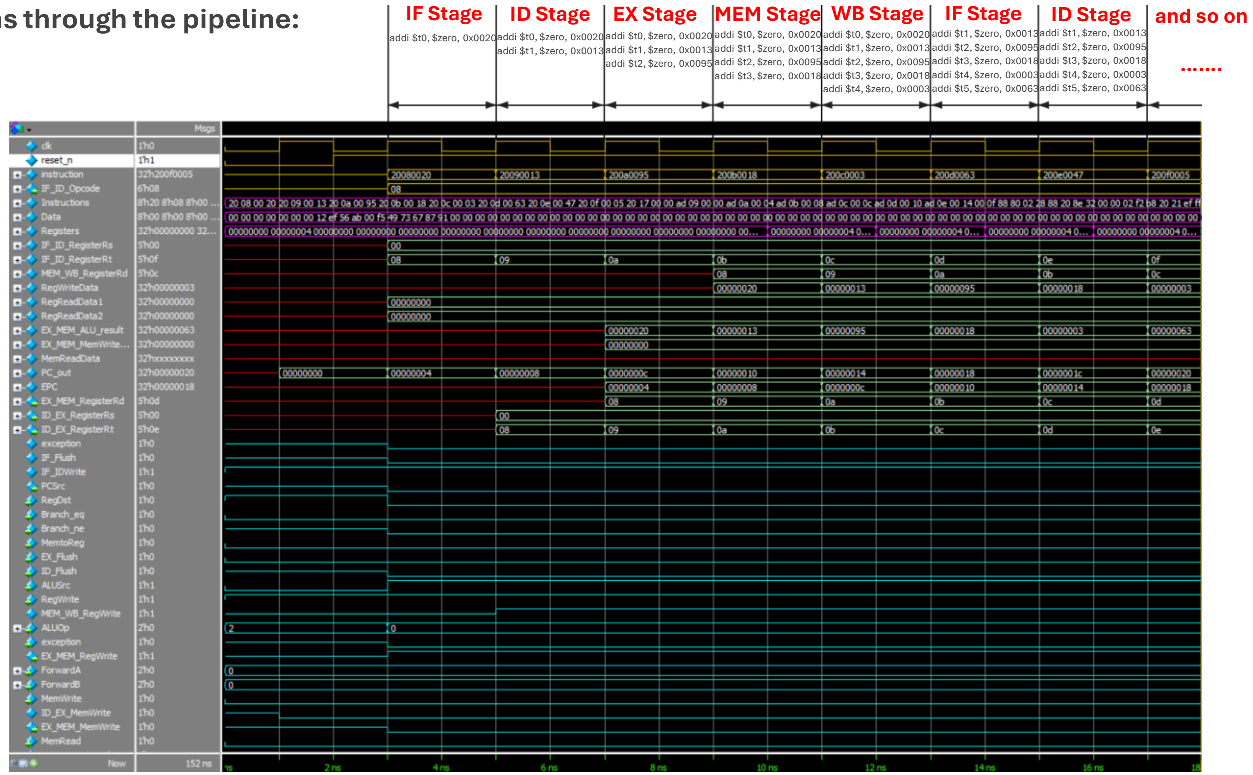
**Wave colors guidelines:**

- Gold for global inputs `(clk, reset_n)`, instruction and the opcode
- Magenta for the arrays `(Instructions, Data and Registers)`
- Cyan for the control lines
- Green for the remaining signals

The simulation is done using ModelSim and automated using a TCL script file.

## 12. Simulation Results (ModelSim): Simulating the RTL design using ModelSim

**Flow of instructions through the pipeline:**



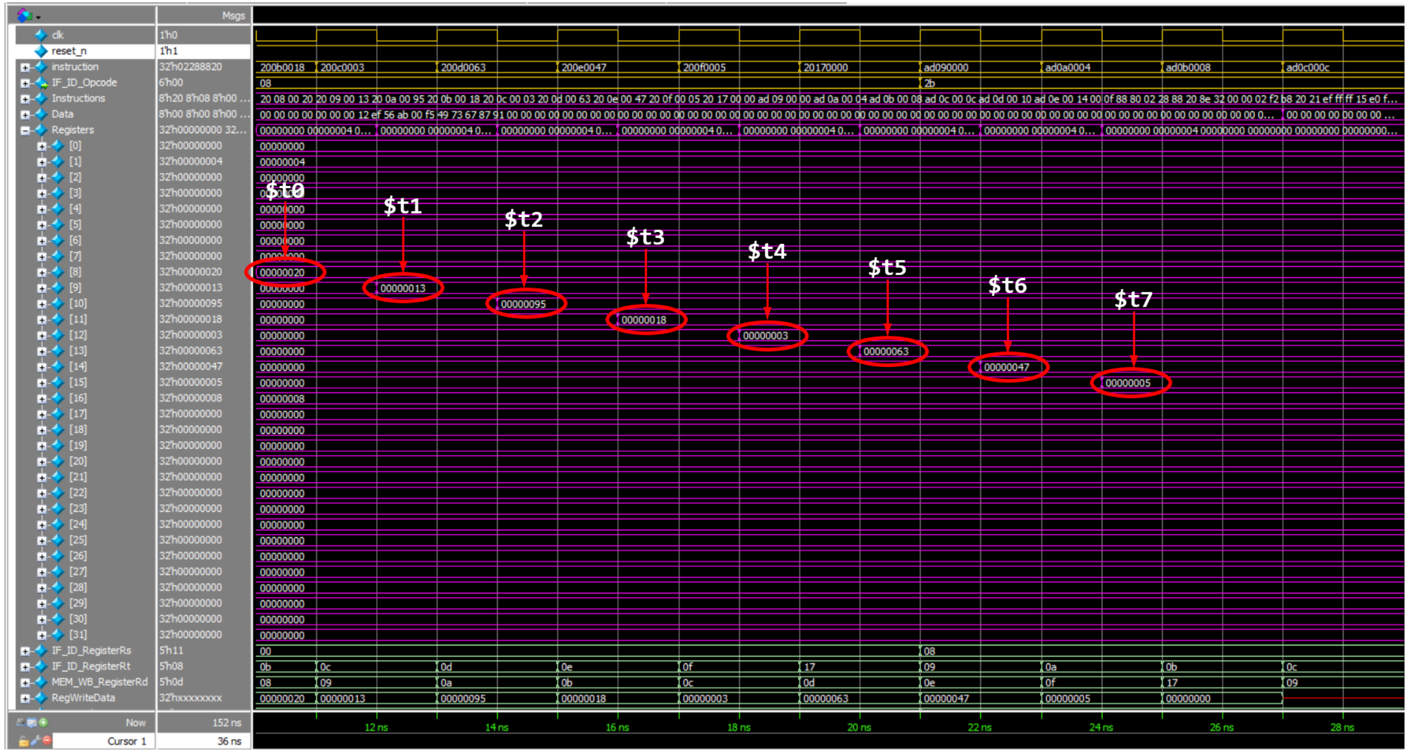
| IF Stage | ID Stage | EX Stage | MEM Stage | WB Stage | IF Stage | ID Stage | and so on |
|---|---|---|---|---|---|---|---|
| addi \$t0, \$zero, 0x0020 | addi \$t0, \$zero, 0x0020 | addi \$t0, \$zero, 0x0020 | addi \$t0, \$zero, 0x0020 | addi \$t0, \$zero, 0x0020 | addi \$t1, \$zero, 0x0013 | addi \$t1, \$zero, 0x0013 | ....... |
| | addi \$t1, \$zero, 0x0013 | addi \$t1, \$zero, 0x0013 | addi \$t1, \$zero, 0x0013 | addi \$t1, \$zero, 0x0013 | addi \$t2, \$zero, 0x0095 | addi \$t2, \$zero, 0x0095 | |
| | | addi \$t2, \$zero, 0x0095 | addi \$t2, \$zero, 0x0095 | addi \$t2, \$zero, 0x0095 | addi \$t3, \$zero, 0x0018 | addi \$t3, \$zero, 0x0018 | |
| | | | addi \$t3, \$zero, 0x0018 | addi \$t3, \$zero, 0x0018 | addi \$t4, \$zero, 0x0003 | addi \$t4, \$zero, 0x0003 | |
| | | | | addi \$t4, \$zero, 0x0003 | addi \$t5, \$zero, 0x0063 | addi \$t5, \$zero, 0x0063 | |

| | Msgs |
|---|---|
| clk | 1'h0 |
| reset_n | 1'h1 |
| instruction | 32'h200f0005 |
| IF_ID_Opcode | 6'h08 |
| Instructions | 8'h20 8'h08 8'h00 ... |
| Data | 8'h00 8'h00 8'h00 ... |
| Registers | 32'h00000000 32... |
| IF_ID_RegisterRs | 5'h00 |
| IF_ID_RegisterRt | 5'h0f |
| MEM_WB_RegisterRd | 5'h0c |
| RegWriteData | 32'h00000003 |
| RegReadData1 | 32'h00000000 |
| RegReadData2 | 32'h00000000 |
| EX_MEM_ALU_result | 32'h00000063 |
| EX_MEM_MemWrite... | 32'h00000000 |
| MemReadData | 32'hxxxxxxxx |
| PC_out | 32'h00000020 |
| EPC | 32'h00000018 |
| EX_MEM_RegisterRd | 5'h0d |
| ID_EX_RegisterRs | 5'h00 |
| ID_EX_RegisterRt | 5'h0e |
| exception | 1'h0 |
| IF_Flush | 1'h0 |
| IF_IDWrite | 1'h1 |
| PCSrc | 1'h0 |
| RegDst | 1'h0 |
| Branch_eq | 1'h0 |
| Branch_ne | 1'h0 |
| MemtoReg | 1'h0 |
| EX_Flush | 1'h0 |
| ID_Flush | 1'h0 |
| ALUSrc | 1'h1 |
| RegWrite | 1'h1 |
| MEM_WB_RegWrite | 1'h1 |
| ALUOp | 2'h0 |
| exception | 1'h0 |
| EX_MEM_RegWrite | 1'h1 |
| ForwardA | 2'h0 |
| ForwardB | 2'h0 |
| MemWrite | 1'h0 |
| ID_EX_MemWrite | 1'h0 |
| EX_MEM_MemWrite | 1'h0 |
| MemRead | 1'h0 |
| Now | 152 ns |

**Patterson and Hennessy's Computer Organization and Design, 5th Ed.**

## 12. Simulation Results (ModelSim): Simulating the RTL design using ModelSim

**Registers' contents after the sequence of addi instructions:**



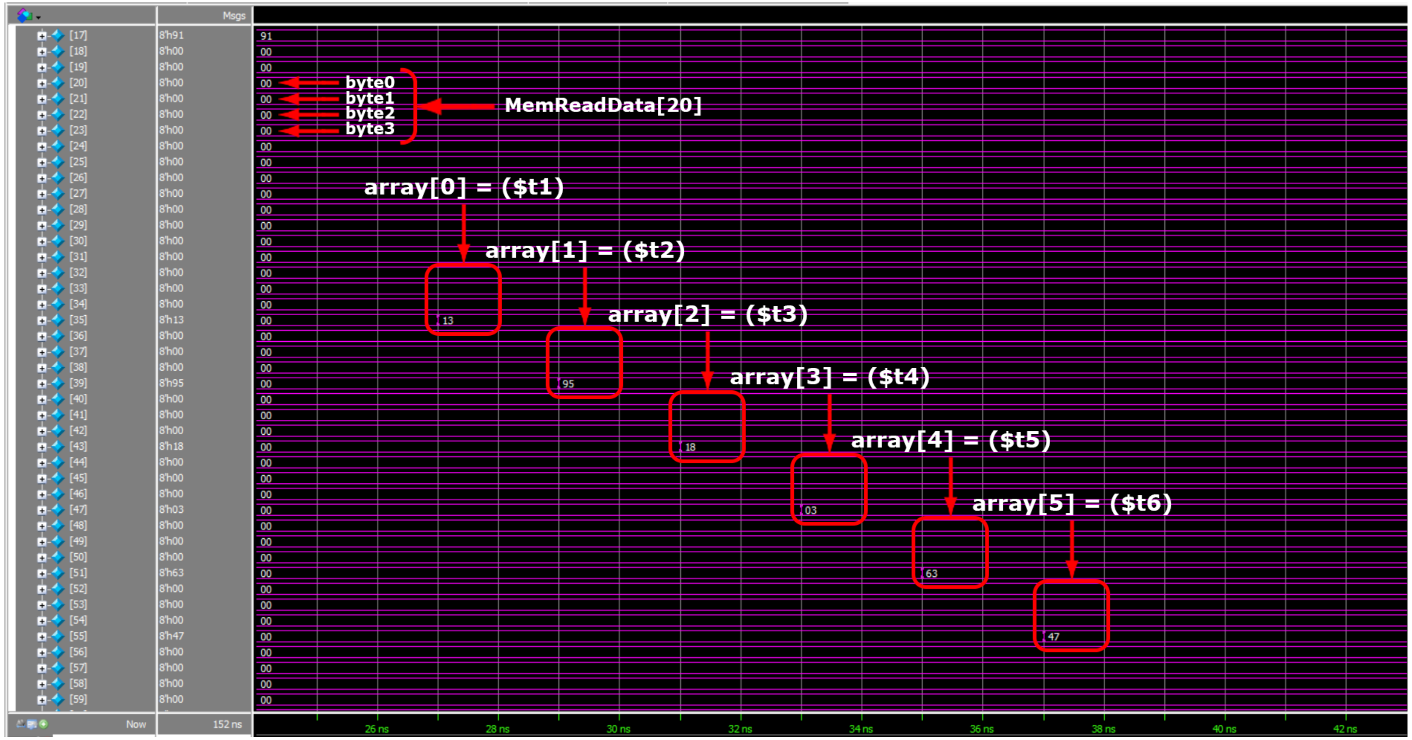
```
                       Msgs
clk                    1'h0
reset_n                1'h1
instruction            32'h02288820      200b0018  200c0003  200d0063  200e0047  200f0005  20170000  ad090000  ad0a0004  ad0b0008  ad0c000c
IF_ID_Opcode           6'h00             08                                                          2b
Instructions           8'h20 8'h08 8'h00 ...  20 08 00 20 20 09 00 13 20 0a 00 95 20 0b 00 18 20 0c 00 03 20 0d 00 63 20 0e 00 47 20 0f 00 05 20 17 00 00 ad 09 00 00 ad 0a 00 04 ad 0b 00 08 ad 0c 00 0c ad 0d 00 10 ad 0e 00 14 00 0f 88 80 02 28 88 20 8e 32 00 00 02 f2 b8 20 21 ef ff ff 15 e0 f...
Data                   8'h00 8'h00 8'h00 ...  00 00 00 00 00 00 00 12 ef 56 ab 00 f5 49 73 67 87 91 00 00 00 00 ...
Registers              32'h00000000 32...     00000000 00000004 0...
  [0]                  32'h00000000      00000000
  [1]                  32'h00000004      00000004
  [2]                  32'h00000000      00000000
  [3]                  32'h00000000      00000000
  [4]                  32'h00000000      00000000
  [5]                  32'h00000000      00000000
  [6]                  32'h00000000      00000000
  [7]                  32'h00000000      00000000
  [8]                  32'h00000020      00000020
  [9]                  32'h00000013      00000000  00000013
  [10]                 32'h00000095      00000000            00000095
  [11]                 32'h00000018      00000000                      00000018
  [12]                 32'h00000003      00000000                                00000003
  [13]                 32'h00000063      00000000                                          00000063
  [14]                 32'h00000047      00000000                                                    00000047
  [15]                 32'h00000005      00000000                                                              00000005
  [16]                 32'h00000008      00000008
  [17]                 32'h00000000      00000000
  [18]                 32'h00000000      00000000
  [19]                 32'h00000000      00000000
  [20]                 32'h00000000      00000000
  [21]                 32'h00000000      00000000
  [22]                 32'h00000000      00000000
  [23]                 32'h00000000      00000000
  [24]                 32'h00000000      00000000
  [25]                 32'h00000000      00000000
  [26]                 32'h00000000      00000000
  [27]                 32'h00000000      00000000
  [28]                 32'h00000000      00000000
  [29]                 32'h00000000      00000000
  [30]                 32'h00000000      00000000
  [31]                 32'h00000000      00000000
IF_ID_RegisterRs       5'h11             00                                                          08
IF_ID_RegisterRt       5'h08             0b        0c        0d        0e        0f        17        09        0a        0b        0c
MEM_WB_RegisterRd      5'h0d             08        09        0a        0b        0c        0d        0e        0f        17        09
RegWriteData           32'hxxxxxxxx      00000020  00000013  00000095  00000018  00000003  00000063  00000047  00000005  00000000

Now                    152 ns                      12 ns     14 ns     16 ns     18 ns     20 ns     22 ns     24 ns     26 ns     28 ns
Cursor 1               36 ns
```

Patterson and Hennessy's Computer Organization and Design, 5th Ed.

## 12. Simulation Results (ModelSim): Simulating the RTL design using ModelSim

**Initializing the array with contents of the registers**

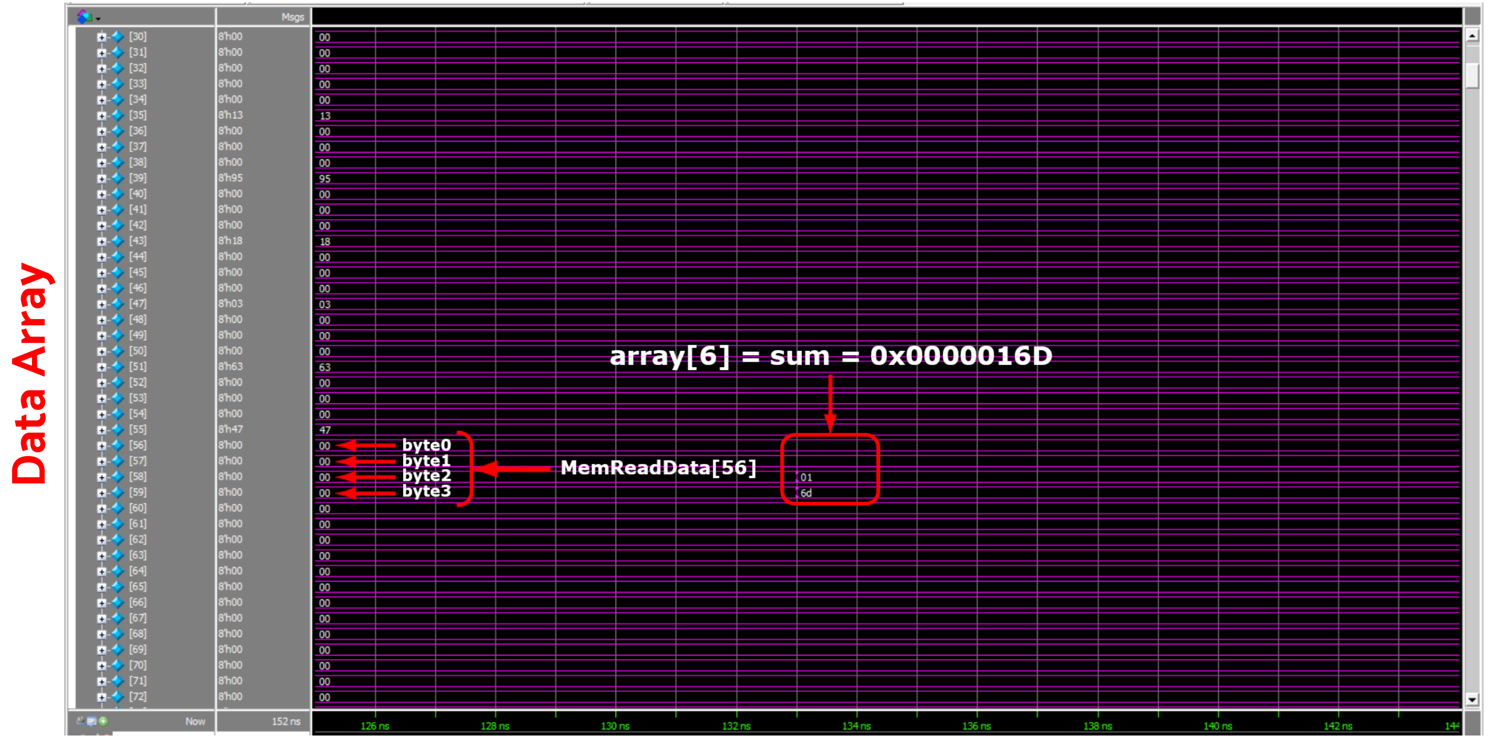
Data Array



```
[17]  8'h91   91
[18]  8'h00   00
[19]  8'h00   00
[20]  8'h00   00
[21]  8'h00   00
[22]  8'h00   00
[23]  8'h00   00
[24]  8'h00   00
[25]  8'h00   00
[26]  8'h00   00
[27]  8'h00   00
[28]  8'h00   00
[29]  8'h00   00
[30]  8'h00   00
[31]  8'h00   00
[32]  8'h00   00
[33]  8'h00   00
[34]  8'h00   00
[35]  8'h13   00  13
[36]  8'h00   00
[37]  8'h00   00
[38]  8'h00   00
[39]  8'h95   00  95
[40]  8'h00   00
[41]  8'h00   00
[42]  8'h00   00
[43]  8'h18   00  18
[44]  8'h00   00
[45]  8'h00   00
[46]  8'h00   00
[47]  8'h03   00  03
[48]  8'h00   00
[49]  8'h00   00
[50]  8'h00   00
[51]  8'h63   00  63
[52]  8'h00   00
[53]  8'h00   00
[54]  8'h00   00
[55]  8'h47   00  47
[56]  8'h00   00
[57]  8'h00   00
[58]  8'h00   00
[59]  8'h00   00
```

## 12. Simulation Results (ModelSim): Simulating the RTL design using ModelSim

Sum of the array elements (0x13 + 0x95 + 0x18 + 0x3 + 0x63 + 0x47 = 0x16D) is stored in array[6]



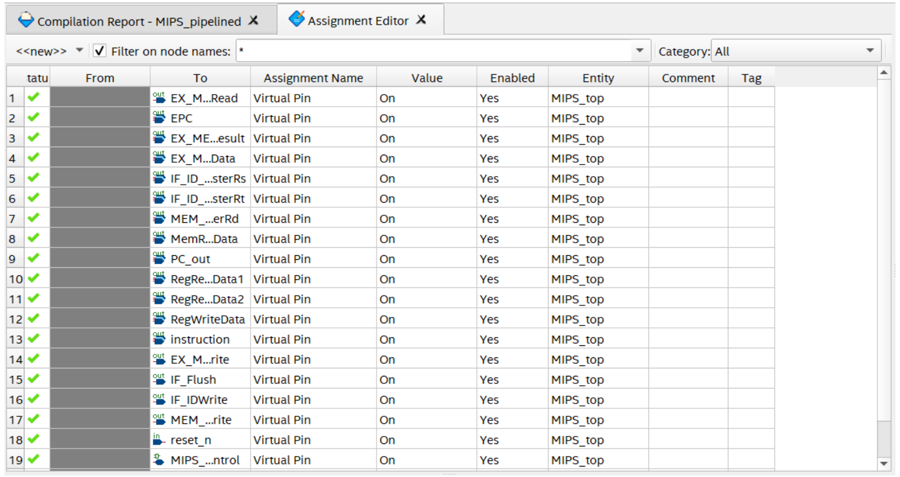
```
                Msgs
[30]    8'h00   00
[31]    8'h00   00
[32]    8'h00   00
[33]    8'h00   00
[34]    8'h00   00
[35]    8'h13   13
[36]    8'h00   00
[37]    8'h00   00
[38]    8'h00   00
[39]    8'h95   95
[40]    8'h00   00
[41]    8'h00   00
[42]    8'h00   00
[43]    8'h18   18
[44]    8'h00   00
[45]    8'h00   00
[46]    8'h00   00
[47]    8'h03   03
[48]    8'h00   00
[49]    8'h00   00
[50]    8'h00   00
[51]    8'h63   63
[52]    8'h00   00
[53]    8'h00   00
[54]    8'h00   00
[55]    8'h47   47
[56]    8'h00   00
[57]    8'h00   00
[58]    8'h00   00
[59]    8'h00   00
[60]    8'h00   00
[61]    8'h00   00
[62]    8'h00   00
[63]    8'h00   00
[64]    8'h00   00
[65]    8'h00   00
[66]    8'h00   00
[67]    8'h00   00
[68]    8'h00   00
[69]    8'h00   00
[70]    8'h00   00
[71]    8'h00   00
[72]    8'h00   00
Now     152 ns
```

Patterson and Hennessy’s Computer Organization and Design, 5th Ed.

## 13. Analysis, Synthesis & Fitting (Quartus Netlist Viewer): Showing RTL Viewer and Fitter Resource Usage Summary

**In our pipelined MIPS architecture project using Quartus, we used virtual pin assignments for:**

- **Simulation and Debugging:**
  Monitor internal signals without physical I/O pins.
- **Efficient Resource Utilization:**
  Optimize internal logic and routing without initial physical pin constraints.
- **Prototyping and Incremental Compilation:**
  Reduce compile times and enhance iterative design efficiency.



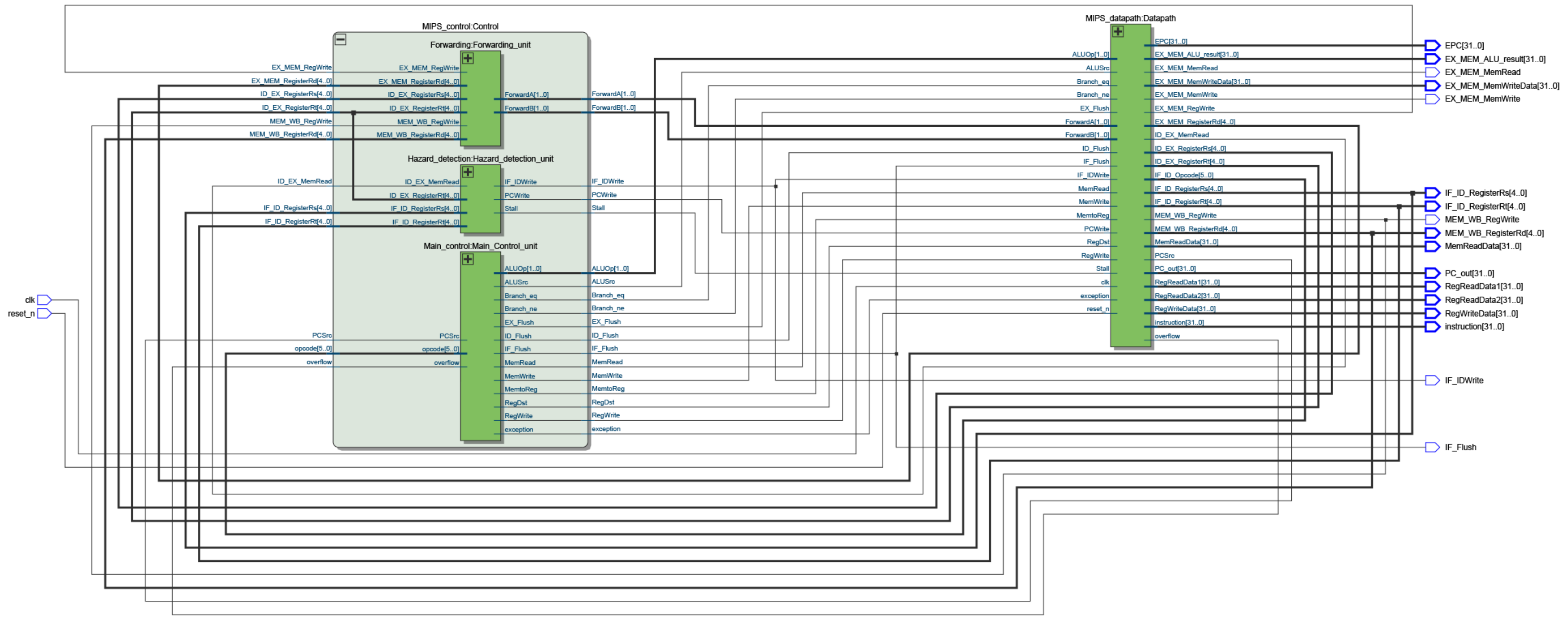
Compilation Report - MIPS_pipelined | Assignment Editor

<<new>> ☑ Filter on node names: * Category: All

| | tatu | From | To | Assignment Name | Value | Enabled | Entity | Comment | Tag |
|---|---|---|---|---|---|---|---|---|---|
| 1 | ✓ | | EX_M...Read | Virtual Pin | On | Yes | MIPS_top | | |
| 2 | ✓ | | EPC | Virtual Pin | On | Yes | MIPS_top | | |
| 3 | ✓ | | EX_ME...esult | Virtual Pin | On | Yes | MIPS_top | | |
| 4 | ✓ | | EX_M...Data | Virtual Pin | On | Yes | MIPS_top | | |
| 5 | ✓ | | IF_ID_...sterRs | Virtual Pin | On | Yes | MIPS_top | | |
| 6 | ✓ | | IF_ID_...sterRt | Virtual Pin | On | Yes | MIPS_top | | |
| 7 | ✓ | | MEM_...erRd | Virtual Pin | On | Yes | MIPS_top | | |
| 8 | ✓ | | MemR...Data | Virtual Pin | On | Yes | MIPS_top | | |
| 9 | ✓ | | PC_out | Virtual Pin | On | Yes | MIPS_top | | |
| 10 | ✓ | | RegRe...Data1 | Virtual Pin | On | Yes | MIPS_top | | |
| 11 | ✓ | | RegRe...Data2 | Virtual Pin | On | Yes | MIPS_top | | |
| 12 | ✓ | | RegWriteData | Virtual Pin | On | Yes | MIPS_top | | |
| 13 | ✓ | | instruction | Virtual Pin | On | Yes | MIPS_top | | |
| 14 | ✓ | | EX_M...rite | Virtual Pin | On | Yes | MIPS_top | | |
| 15 | ✓ | | IF_Flush | Virtual Pin | On | Yes | MIPS_top | | |
| 16 | ✓ | | IF_IDWrite | Virtual Pin | On | Yes | MIPS_top | | |
| 17 | ✓ | | MEM_...rite | Virtual Pin | On | Yes | MIPS_top | | |
| 18 | ✓ | | reset_n | Virtual Pin | On | Yes | MIPS_top | | |
| 19 | ✓ | | MIPS_...ntrol | Virtual Pin | On | Yes | MIPS_top | | |

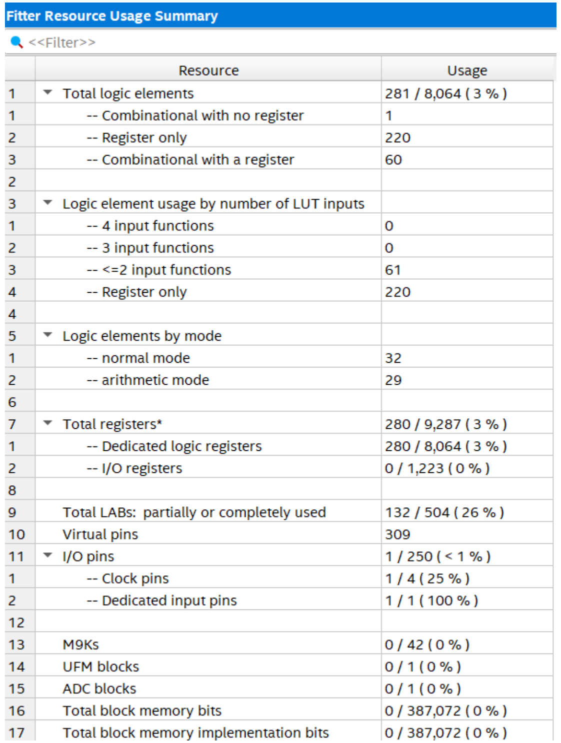
## 13. Analysis, Synthesis & Fitting (Quartus Netlist Viewer): Showing RTL Viewer and Fitter Resource Usage Summary

**RTL Viewer:**



## 13. Analysis, Synthesis & Fitting (Quartus Netlist Viewer): Showing RTL Viewer and Fitter Resource Usage Summary

**Fitter Resource Usage Summary:**



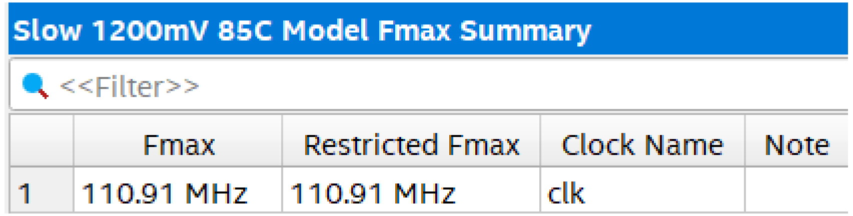
Fitter Resource Usage Summary

<<Filter>>

| | Resource | Usage |
|---|---|---|
| 1 | Total logic elements | 281 / 8,064 ( 3 % ) |
| 1 | -- Combinational with no register | 1 |
| 2 | -- Register only | 220 |
| 3 | -- Combinational with a register | 60 |
| 2 | | |
| 3 | Logic element usage by number of LUT inputs | |
| 1 | -- 4 input functions | 0 |
| 2 | -- 3 input functions | 0 |
| 3 | -- <=2 input functions | 61 |
| 4 | -- Register only | 220 |
| 4 | | |
| 5 | Logic elements by mode | |
| 1 | -- normal mode | 32 |
| 2 | -- arithmetic mode | 29 |
| 6 | | |
| 7 | Total registers* | 280 / 9,287 ( 3 % ) |
| 1 | -- Dedicated logic registers | 280 / 8,064 ( 3 % ) |
| 2 | -- I/O registers | 0 / 1,223 ( 0 % ) |
| 8 | | |
| 9 | Total LABs: partially or completely used | 132 / 504 ( 26 % ) |
| 10 | Virtual pins | 309 |
| 11 | I/O pins | 1 / 250 ( < 1 % ) |
| 1 | -- Clock pins | 1 / 4 ( 25 % ) |
| 2 | -- Dedicated input pins | 1 / 1 ( 100 % ) |
| 12 | | |
| 13 | M9Ks | 0 / 42 ( 0 % ) |
| 14 | UFM blocks | 0 / 1 ( 0 % ) |
| 15 | ADC blocks | 0 / 1 ( 0 % ) |
| 16 | Total block memory bits | 0 / 387,072 ( 0 % ) |
| 17 | Total block memory implementation bits | 0 / 387,072 ( 0 % ) |

## 14. Timing Analysis (Quartus Timing Analyzer): Showing Fmax

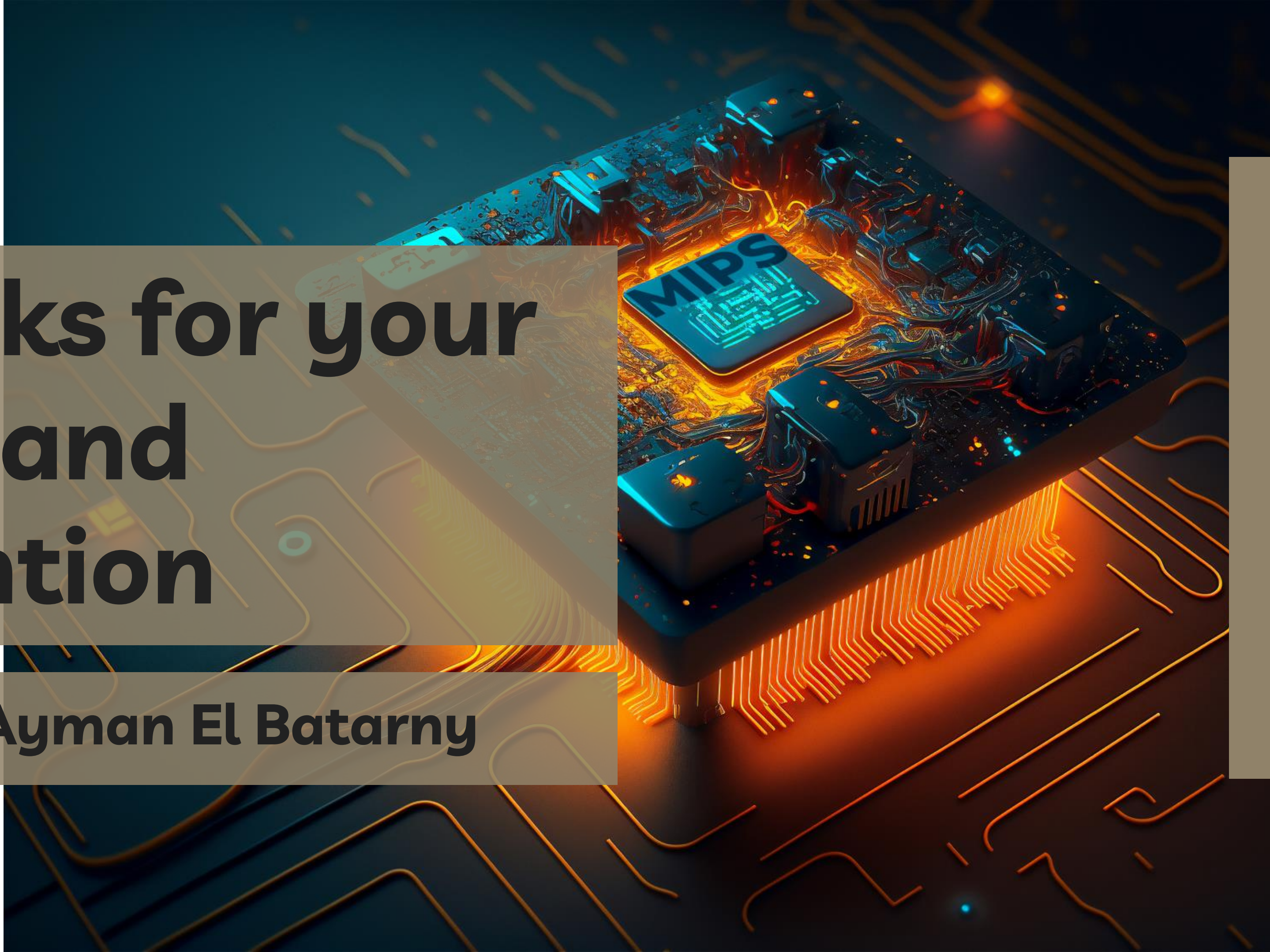
**Fmax (Slow 1200mV 85C Model):**



Slow 1200mV 85C Model Fmax Summary

<<Filter>>

| | Fmax | Restricted Fmax | Clock Name | Note |
|---|---|---|---|---|
| 1 | 110.91 MHz | 110.91 MHz | clk | |



# Thanks for your time and attention

By: Amr Ayman El Batarny