|  |  |
| --- | --- |
|  |  |
| Y86 Processor Architecture Verilog Implementation |  |
|  |  |
|  | PROJECTIntroduction to Processor Architecture |
|  | **Team Members:**Aryaman Mahajan (2022102034)Yash Nitin Dusane (2022102078) |

.

## Introduction

The Y86 Processor is being implemented in 2 ways:

1. Sequential
2. Pipeline

A screenshot of a computer

Description automatically generated

As a first step in designing a processor, we present a functionally correct, but somewhat impractical, Y86 processor based on sequential operation. This processor executes a complete Y86 instruction on every clock cycle. The clock must run slowly enough to allow an entire series of actions to complete within one cycle. Such a processor could be implemented, but its performance would be well below what could be achieved for this much hardware.

With the sequential design as a basis, we then apply a series of transformations to create a pipelined processor. This processor breaks the execution of each instruction into five steps, each of which is handled by a separate section or stage of the hardware. Instructions progress through the stages of the pipeline, with one instruction entering the pipeline on each clock cycle. As a result, the processor can be executing the different steps of up to five instructions simultaneously. Making this processor preserve the sequential behavior of the Y86 ISA requires handling a variety of hazard conditions, where the location or operands of one instruction depend on those of other instructions that are still in the pipeline.

## Sequential Implementation

A blue rectangle with black lines

Description automatically generated

Sequential Implementation has a simple basis of one instruction at a time.

It comprises of 6 stages, which are made common for all instructions. So each instruction has divided sub instructions for each stage.

A computer screen shot of a computer program

Description automatically generated

## Fetch Stage:

* Reads bytes of an instruction from memory using the PC value as address 🡪 Extracts the two 4-bit portions of instruction specifier byte referred to as **icode** and **ifun**
* Possibly fetches the register specifier byte giving one or both of the register operand specifiers rA and rB
* Also possibly fetches an 8-byte constant word valC 🡪 Computes valP as the address of the next instruction in the sequence, i.e. valP = PC + length of fetched instruction

A screenshot of a video game

Description automatically generated

Here according to the input, we have got the required values from the instruction set.

The instruction is:

irmovq $0x100, %rbx

irmovq $0x200, %rdx

addq %rdx, %rbx

rrmovq %rbx %rdi

pushq %rdx

subq %rdi %rdx

popq %rcx

nop

cmovge %rbx %rax

jmp

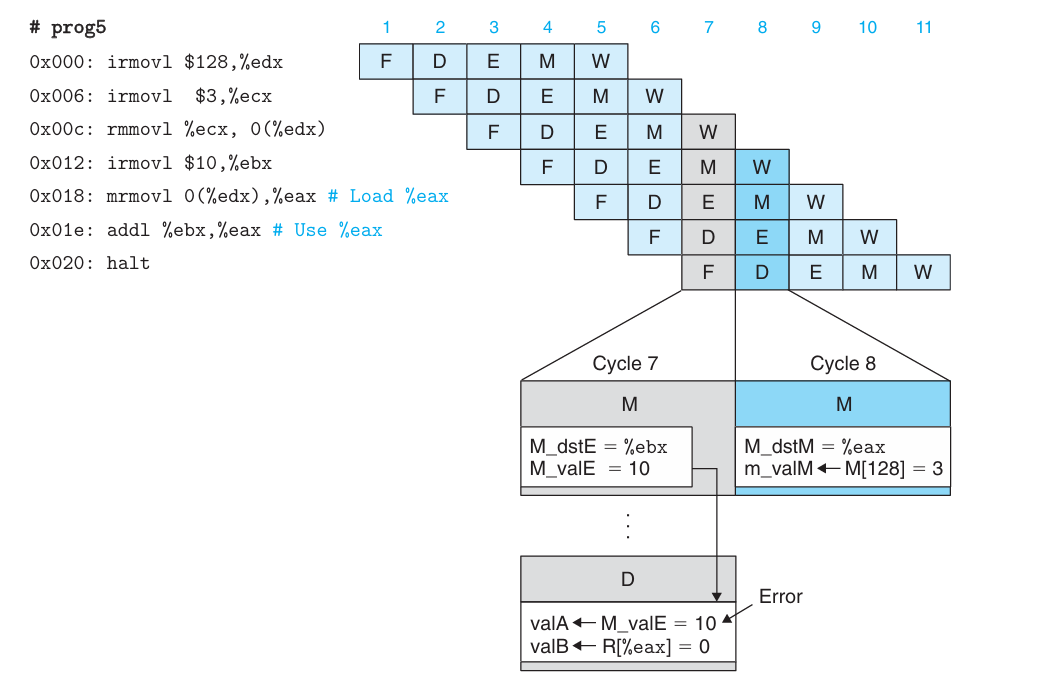
## Pipeline Hazards:

(After Data Forwarding)

1. Load Use Hazard (Data Hazard)

Waiting for the value required to reach the required destination using stall and bubble.

Here, writing into register requires at least it to reach memory stage to get the value to be written inside register from memory. Then, through data forwarding, we get the required value through m\_valM, at the expense of one nop operation till the previous instruction reaches memory stage and computes the value.



This condition arises in case of 2 instructions, mrmovq and popq, with any instruction based on the same register.

A diagram of a machine

Description automatically generated

A green and black rectangular with black text

Description automatically generated

A screen shot of a computer screen

Description automatically generated

1. Mis predicted Control

This shows for mis predicted jump. We predict jump to happen, but if isn’t, we need to know whether to follow next instruction, before it enters Decode stage. So here also stall and bubble is being used.

A screenshot of a computer

Description automatically generated

A diagram of a computer

Description automatically generated

A green and black rectangular with black letters and a red letter

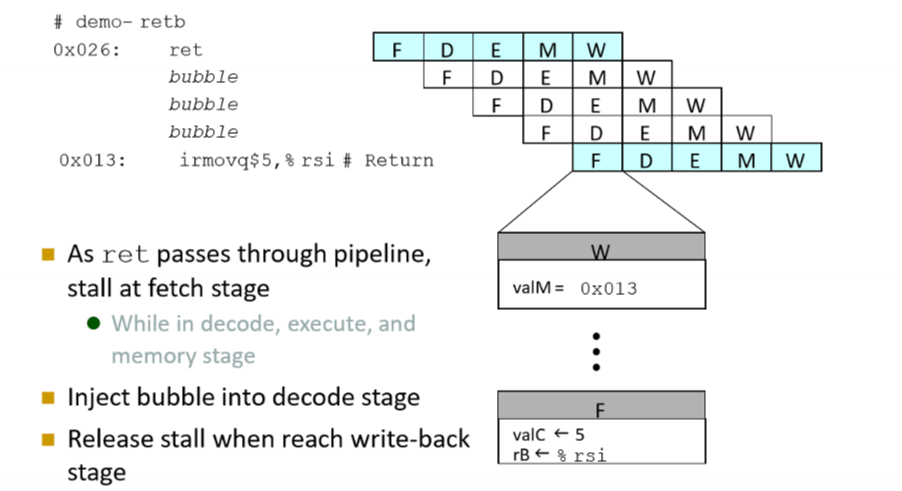
Description automatically generated

A screenshot of a computer

Description automatically generated

1. Return Instruction

The return instruction cannot have any prediction regarding next PC instruction, thus there is no choice than waiting for next instruction address to get into register from the stack memory, which was put during the call function. Thus 3 nops or 3 bubbles need to be inserted at decode stage (also 3 stalls at fetch stage).



A diagram of a computer program

Description automatically generated

A green and black rectangular with black text

Description automatically generated

A screenshot of a computer

Description automatically generated

*Challenges faced during implementation of Pipelined Architecture :*

1. PC Updation

Since pc update was not happening as expected upon using a 2 modular approach, hence we split fetch into two Verilog modules. One becomes the select pc module and one becomes the general fetch module. Th output of select is connected to input of fetch via the wrapper unit.

1. Handling hazards

In setting the control signals of forward and bubble for various modules we were making a conceptual glitch, which became known to us after a lot of testing. Setting these flags correctly and devising codes and working of them was challenging.