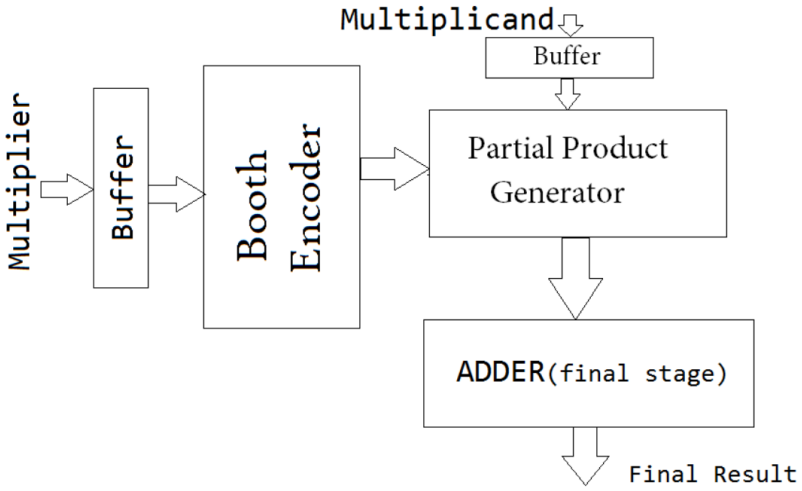
Assignment 2: Advance Computer Architecture

Problem1: Design a 32-bit Booth multiplier circuit using Verilog.

Description: Andrew Donald Booth proposed Booth's multiplication algorithm which can perform the multiplication operation of Two Signed Binary numbers in their respective 2's complement form. A.D. Booth's encoding technique is also called as Radix-2 Booth's encoding algorithm.

Multiplier Architecture:



Algorithm:

1. The Least Significant Bit (LSB) should be padded with zero (Y-1 = 0).

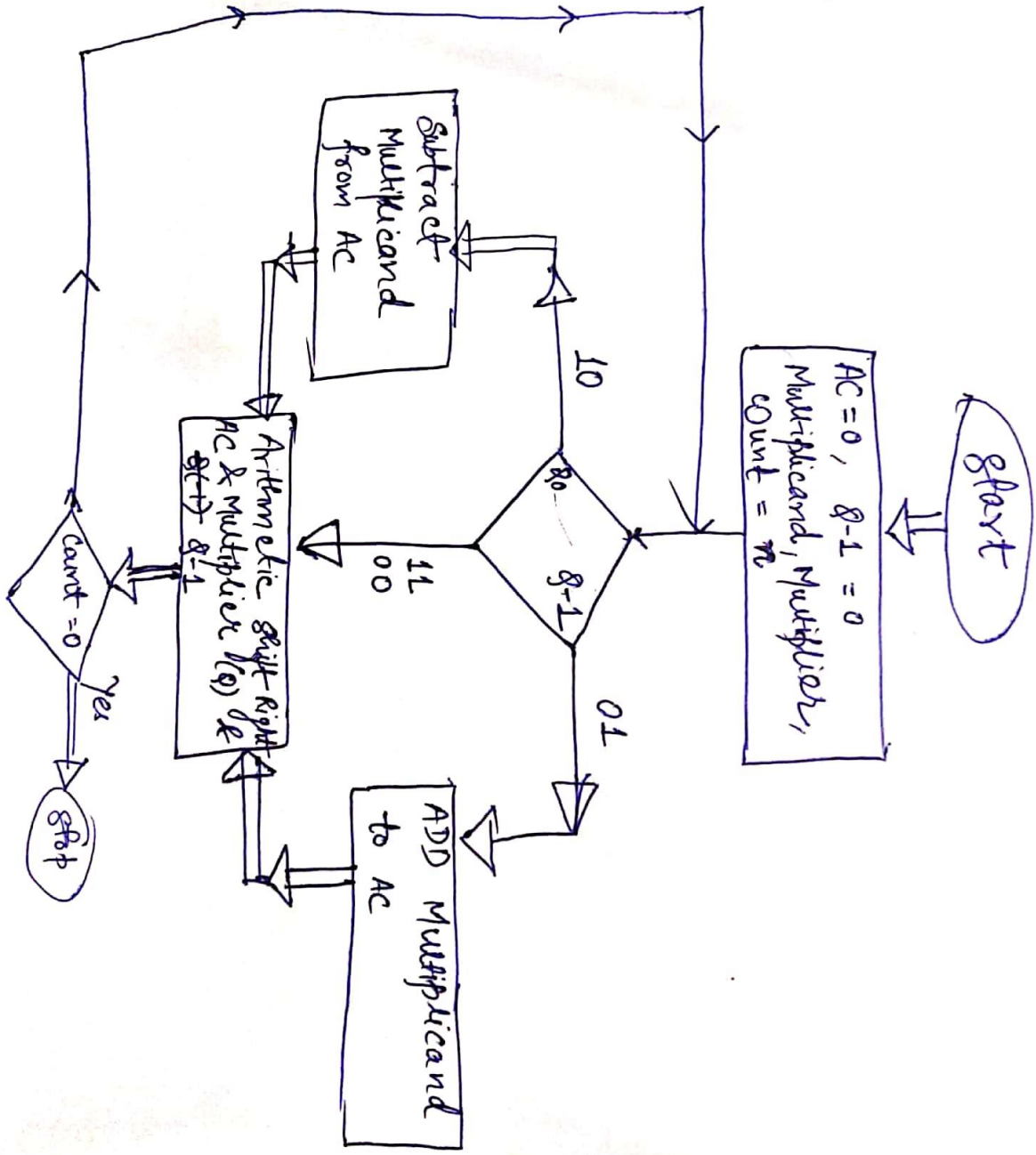
2. For Signed, the MSB must be padded with two zeros when n is even or else one zero. For Unsigned, the padding is not necessary when n is even.

3. Grouping of Multiplier into 3-bits must be done which in turn are overlapping.

4. Partial products are generated with the help of Booth's recoding table as stated above.

5. On adding the partial products the result can be found.

Flow Diagram:



Problem2: Design a 32-bit Non-restoring divider circuit using Verilog.

Non-restoring division uses the digit set {−1, 1} for the quotient digits instead of {0, 1}. The algorithm is more complex, but has the advantage when implemented in hardware that there is only one decision and addition/subtraction per quotient bit; there is no restoring step after the subtraction, which

potentially cuts down the numbers of operations by up to half and lets it be executed faster. The basic algorithm for binary (radix 2) non-restoring division of non-negative numbers is:

Algorithm:

R := N

D := D << n -- R and D need twice the word width of N and Q

for i = n − 1 .. 0 do -- for example 31..0 for 32 bits

if R >= 0 then

q[i] := +1

R := 2 * R − D

else

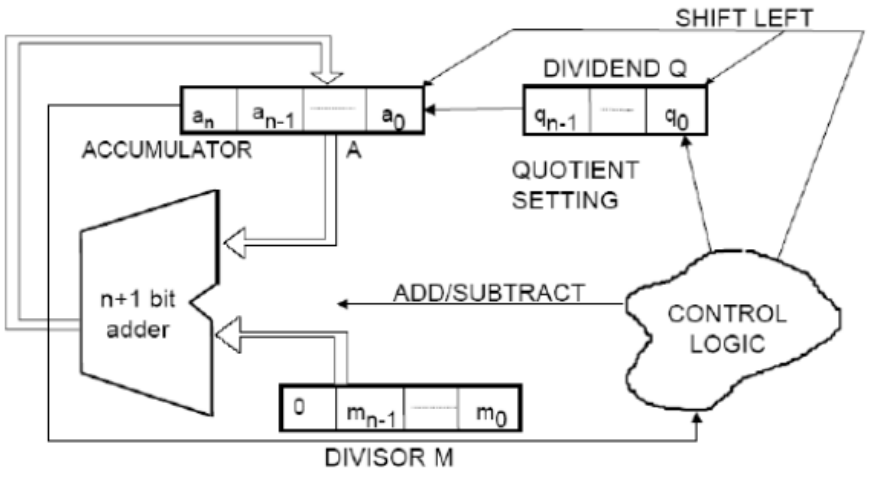
q[i] := −1

R := 2 * R + D

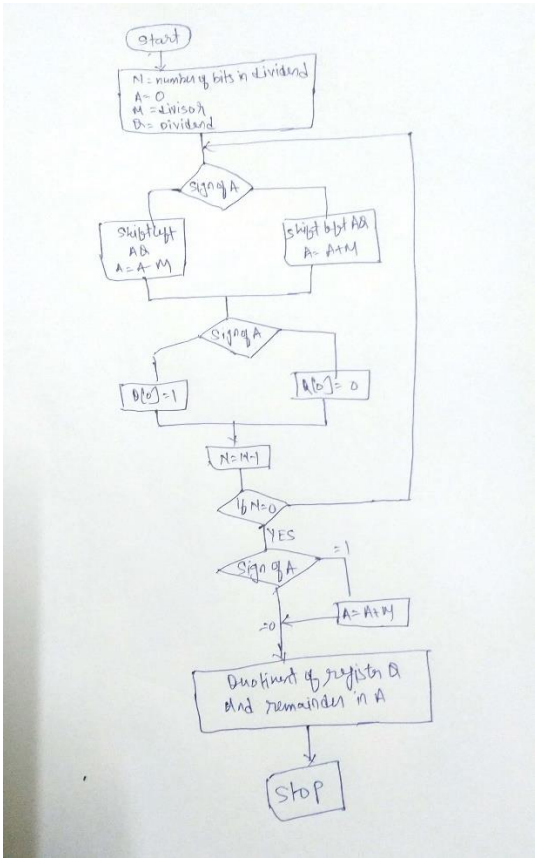
end if

end

Circuit Diagram:



Flow Diagram:



Problem 3:

Using QtSPIM, write and test an adding machine program that repeatedly reads in integers and adds them into a running sum. The program would stop when it gets an input that is 0, printing out the sum at that point.

In data section two ascii strings are taken str1 and strt to output "get input" and "ans".

In text section we started taking values into register $v0 and checking whether it is equal to 0 or not when the value becomes 0 we end the program and print the sum if value is not eqaul to 0 we go for the loop and add the sum in register $s0.

Problem 4:

Using QtSPIM, write and test a program that reads in three integers and prints out the sum of the largest two of three. You can break ties arbitrarily

Data section of this is same as above section in text section we take 3 inputs and store the imputs in register t0 ,t1 and t2.

Now we find two bigger numbers by using slt micro instruction which sets the value of register to 1 by comparing values,example (slt $t3, $t0, $t1 # $t3 = 1 iff $t0 < $t1 setting t3 to 1 if t0 is less than t1).

At last we print sum of two bigger numbers.

Problem5:

Using QtSPIM, write and test a program that reads in a positive integer using the SPIM system calls. If the integer is not positive, the program should terminate with the message "Invalid Entry"; otherwise the program should print out the names of the digits of the integers delimited by exactly one space. For example, if the number entered is "728", the output would be "seven two eight".

In data section of this question we store string values for 1,2,3,4 … etc to one,two,three,four ….

Example str0:

```
 .asciiz "Zero "
```

str1:

```
 .asciiz "One "
```

str2:

```
 .asciiz "Two "
```

str3:

.asciiz "Three "......

In the logic first we are counting the number of digits in the number the user has entered.

Using rep1 label we have counted the digits.

From rep2 loop we are finding msb stored in $t3 by comparing it with $t4.we are comparing $t3 with $t4 by beginning $t4 from 0 and incrementing $t4 one everytime until it matches $t3.

Once it matches we go to respective print label to print that number in words.after printing the digit we jump to p_digit label.here we are removing the most significant bit and go to rep2 label

to find out next most significant significant digit and repeat the process until the last bit.

Problem 6:

Write and test a MIPS assembly language program to compute and print the first 20 prime numbers. A number *n* is prime if no numbers except 1 and *n* divide it evenly. You should implement two routines:

In this code we are storing 2 in $s0 which is the smallest prime and 20 in $s1 this is the number of primes we have to find and in $s2 we are storing the count of primes we have found till now.

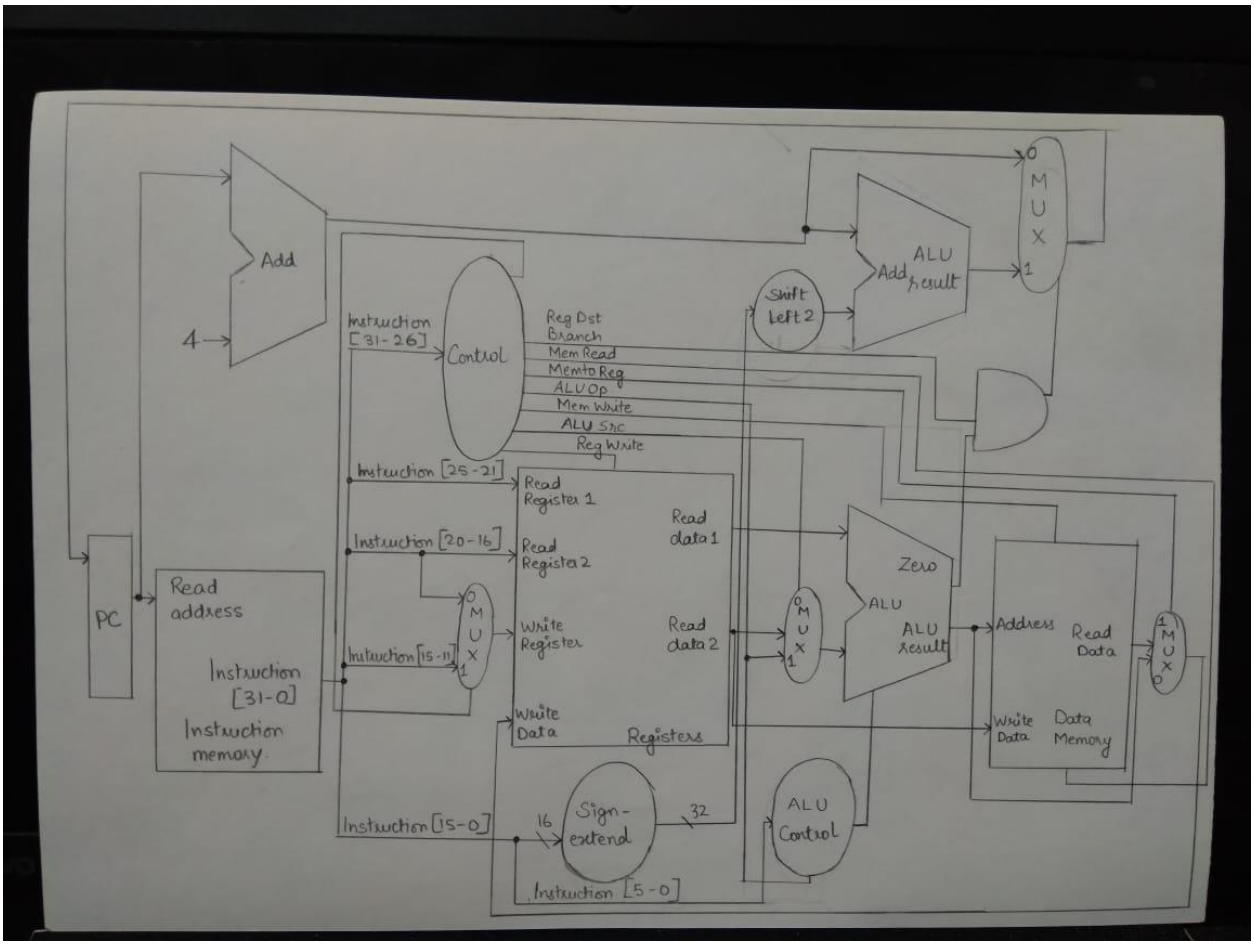
In rep1 loop we are starting number from 2 onwards and started checking for prime sending the number to test_prime and test_loop ,here we are dividing the number from (2 to that number itself) if it is divisible by any of the number other than 2 and itself we are storing 0 $v0

Indicating it is not prime and storing 1 in $v1 indicating it is prime and then transfering the control back to rep1 to line next to jal_prime and for doing that we are using jr $ra.

Problem 7.

Design a single-cycle MIPS-based (32-bit) processor using Verilog for the instructions such as R-type (ADD, SUB, AND, OR, SLT), I/M-type (LW/SW/ADDI/SUBI) and BEQ and J-type instructions (JAL, J). Consider only integer type of operation. To test the design one must use the mipstest.s file. Find out the delay of the design.

Design:



Most elements in the block diagram are implemented with modules. Exceptions include cases such as the "Shift left 2" elements, which are instead implemented directly with standard verilog syntax. These simplifications take advantage of the capabilities verilog offers.

ALU

The ALU is easily implemented with a case statement that checks the ALU operation code from the ALU control module and applies the matching operation to the data inputs. These operations are native to verilog and were easily implemented. In order to set the Zero signal, the ALU operation was used to set the ALU result, which could then be compared to zero sequentially.

ALU control

The ALU control examines an "rtype" signal from the Control module in order to select either the aluop code it receives from the Control module or the aluop code it generates from the function code it receives from the instruction. This aluop code selection is then

sent to the ALU. This module also examines the functioncode and uses it to determine if the instruction is a syscall. In that case, it sends a 1 over the syscall wire to the Registers module where the syscall is executed.

Control

The control module supplies signals controlling the execution of the data pipeline. This module examines the instruction's top six bits (the opcode), and sets the appropriate control signals. Generally this could be done with case statements, though in some cases it was easier to implement with simpler conditional logic.

Additions made to this module that aren't reflected in the above diagram include an "invertzero" signal that causes the ALU Zero signal to be inverted. This allows for the implementation of a number of instructions that requires the ability to detect when two values are not equal, such as in the BNE operation. An "rtype" signal was also added to allow the detection of R type instructions and the appropriate multiplexing of the aluop code. rtype is 1 when the functioncode should be used to control the alu, and 0 when the aluop code should be gotten from the control module.

Instruction memory

The instruction memory module stores the program to be executed. When running the simulation, a file containing instructions is read into this module's memory, which is than fed into the rest of the processor. This module uses the program counter (PC) to select an instruction set the output.

Registers

The registers module contains the register file where the registers are stored. On the clock, registers examines it's inputs and reads and/or writes to the register file (reg_file). After each write, the zero register is set back to 0 incase it was written to. On initialization the 0 is stored in all the registers. The registers module also includes logic for executing syscall execution. This allows easy access to a0 and v0 registers, reducing complexity. In the future this logic will be in a module that is instantiated within the registers module.

Data memory

Another memory module. Accepts a 32 bit address and data and stores the data when MemWrite is 1. When MemRead is 1, outputs the data found at the address.

Add

Adds two 32 bit signals and outputs the result. Used to generate PC+4, but may be used for arbitrary addition.

and

Ands two 1bit signals and outputs the result. Used to check if both branch and zero is 1, but may be used arbitrarily.

mux32_2

A 2 input, 32bit multiplexer used throughout the processor to toggle between different signals in the datapath.

mux1_2

A 2 input, 1bit multiplexer.

jump_address_constructor

This module accepts the PC+4 signal from the pc_incrementer adder and the 25-0 bits from the instruction. It shifts the instruction bits to the left by 2 and concatenates on the left and the shift left result on the right. This signal is then sent to the jump_mux and is used in jump instructions to update the program counter to the new jump address.

Average Delay:

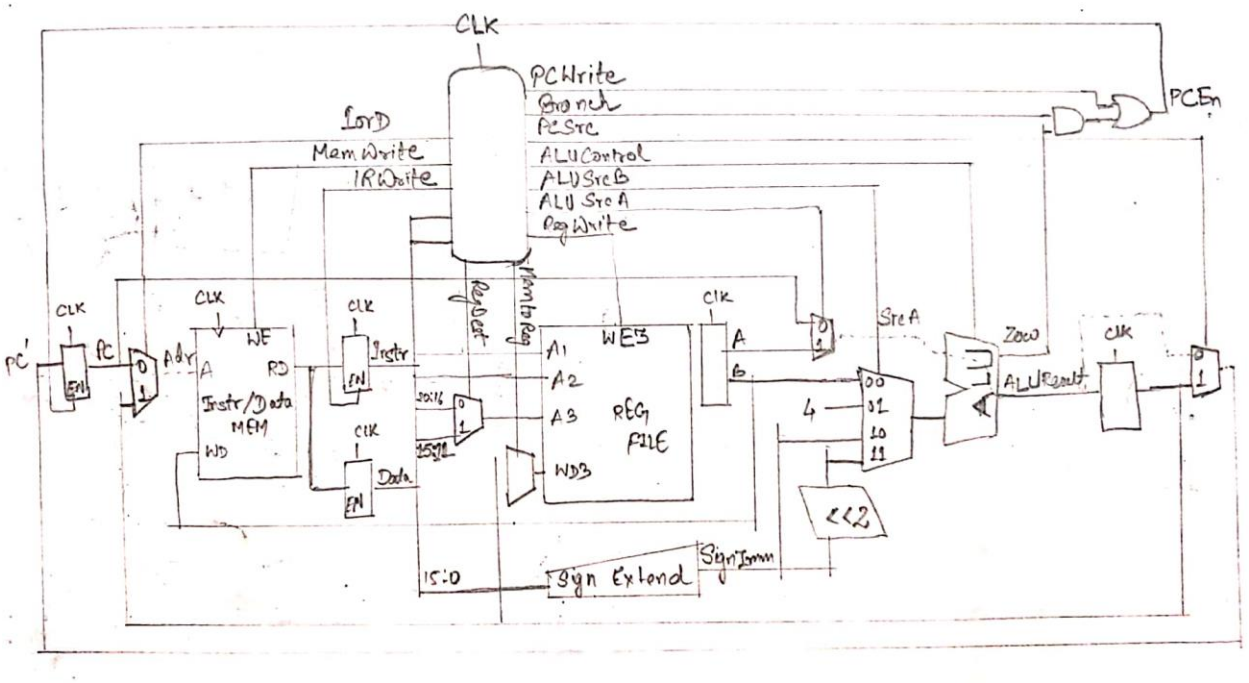
Total cycle time for 18 instruction=175,000ps

Number of instructions=18

Average delay=175000/18=9722.222ps.

***Problem* 8. Design a multicycle MIPS-based (32-bit) processor using Verilog for the instructions such as R-type (ADD, SUB, AND, OR, SLT), I/M-type (LW/SW/ADDI/SUBI) and BEQ and J-type instructions (JAL, J). Consider only integer type of operation. To test the design one must use the mipstest.s file.**

Design:



Description:

The multicycle Datapath builds upon the single cycle. The multicycle datapath has a PC register and a Register File similar to the single cycle datapath. Unlike the single cycle, the multicycle combines the data and instruction memories. Other components such as multiplexers, sign extenders, and ALU are included in the datapath.

The multicycle datapath follows the five-stage execution process: Fetch, decode, ALU, data access, and register write.

Fetch Control Signals

IodD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite = 1
PCWrite = 1

Decode Control Signals:

ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00

R-Type

If the opcode is a R type instruction, the result must be calculated using the ALU and stored back to the register. To carry out ALU calculation, ALUSrcA is set to 1, ALUSrcB is set to 0, and ALUOp is set to 10. ALUSrcA selects the $rs register to be used as SrcA and ALUSrcB selects $rt register to be used as SrcB of the ALU. ALUOp set to 10 indicates to the controller that ALU operation mode is dependent on the function field of the instruction. For result storage, RegDst and RegWrite are set to 1 and MemtoReg is set to 0. RegDst selects $rd register as the write destination and MemtoReg indicates the data to be written is from ALU. RegWrite serves as a write enable for the Register File.

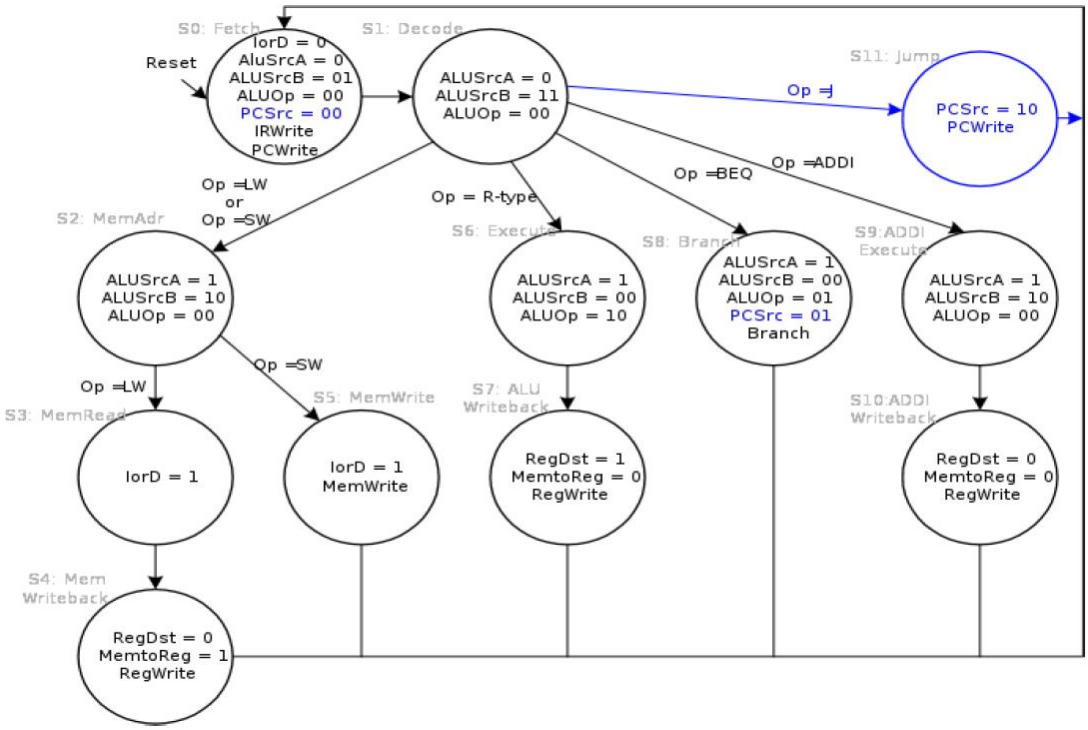
I-Type

Unlike the R type instruction, not all I-type instructions are carried out the same. The load word (lw), store word (sw), add immediate (addi), and branch if equal (beq) use different amounts of cycles. After the lw and sw instructions are decoded, the address for memory access must be computed by adding a base address located in the $rs register and a sign extended immediate. Control signals must be set to control the multiplexers that handle inputs at various sections of the datapath. The appropriate control signals for this step are ALUSrcA to 1, ALUSrcB to 10, and ALUOp to 00. Next, the calculated address is used to access memory; IorD is set to 1 to indicate that the incoming address is from the ALU. For sw, MemWrite is set to 1. The data located in the WriteData (WD) portion is stored to memory. Register $rt is always fed to WD, however the data from $rt is not written to memory unless MemWrite is asserted. The sw instruction is done, but lw has to write back to the register. Three control signals are set: RegDst

The fetch control signals are used to calculate PC+4 for the next instruction. The decode control signals are mainly used for branching. After the fetch and decode steps, the datapath of I-type, J-type, and R-type instructions vary. to 0, MemtoReg to 1, and RegWrite to 1. Similar to the memory portion, if RegWrite is not set to 1, the data inside WD3 will not be written to the Register File.
For addi, $rs is still added to a sign extended immediate, but instead of using the result to access memory, the result is stored in the address located in $rt. The control signals for the ALU computation remain the same as the lw and sw instructions. Afterwards, the result is written to the Register File. The control signals for this are RegDst and RegWrite to 1 and MemtoReg to 0.
The beq instruction is has less stages than all of the other I type instructions. The branch is evaluated immediately when the ALU result is calculated. To test if a branch is equal, the values stored in $rs and $rt are subtracted from each other. If the differences between the two registers are zero, the values are equal. The result of the subtraction is indicated by the zero signal from the ALU. Once the zero signal is set to one, the result is fed into a two input AND gate with the branch signal. The result of that AND gate will produce a 1 and make PCEn 1. While this takes place, the result of the ALU is fed into the PC. With PCEn set to 1, the value of PC will be overwritten with this result. The control signals needed to carry out the branch are ALUSrcA to 1, ALUSrcB to 00, ALUOp to 01, Branch to 1, and PCSrc to 1.

J-Type

Whenever a J-Type instruction is indicated by the opcode, the 26 least significant bits are taken from the instruction and modified as a pseudo direct address. After the instruction is decoded, the PCSrc control signal is set to 10 and the PCWrite to 1. The PCSrc signal allows the ALUResult to circumvent the register and go directly to the PC. The PCWrite makes the OR gate of PCEn to produced a 1 and enable the PC register to be overwritten.



Average Delay:

Total cycle time taken for 18 instruction=625000ps

Number of instructions=18

Avg Delay=625000ps/18=34,722.22ps.