

# 32-bit Two-Phase Clocked Pipelined RISC-like Processor Using Verilog HDL

P Manoj Kumar (21IE10027)

Self Project : Autumn - 2023 : RISC32

## 1 32-bit RISC-like Processor Specification

<b>Registers</b>	32 32-bit general-purpose registers R0..R31, organized as a register bank with two read ports and one write port.
<b>Special Register</b>	R0 is a special read-only register that is assumed to contain the value of 0.
<b>Memory</b>	Memory is byte addressable with a 32-bit memory address. All operations are on 32-bit data, and all loads and stores occur from memory addresses that are multiples of 4.
<b>Program Counter</b>	A 32-bit program counter (PC).
<b>Addressing Modes</b>	Register addressing, Immediate addressing, Base addressing for accessing memory (with any of the registers used as base register), PC relative addressing for branch

## 2 Instruction Set

Type	Instruction	Format	Example
Arithmetic & Logic	ADD	R1, R2, R3	$R1 = R2 + R3$
	SUB	R1, R2, R3	$R1 = R2 - R3$
	AND	R1, R2, R3	$R1 = R2 \& R3$
	OR	R1, R2, R3	$R1 = R2 \mid R3$
	XOR	R1, R2, R3	$R1 = R2 \wedge R3$
	NOT	R1, R2	$R1 = \sim R2$
	SLA	R1, R2, R3	$R1 = R2 \ll R3[0]$
	SRA	R1, R2, R3	$R1 = R2 \gg R3[0]$
	SRL	R1, R2, R3	$R1 = R2 \gg\gg R3[0]$
Immediate Addressing	ADDI	R1, #25	$R1 = R1 + 25$
	SUBI	R1, #-1	$R1 = R1 - 1$
	SLAI	R1, #1	$R1 = R1 \ll 1$
Load & Store	LD	R1, 10(R2)	$R1 = \text{Mem}[R2 + 10]$
	ST	R1, -2(R3)	$\text{Mem}[R3 - 2] = R1$
Branch	BR	#10	$PC = PC + 10$
	BLT	R1,R2 #-10	$PC = PC - 10 \text{ if } (R1 < R2)$
	BGT	R1,R2 #30	$PC = PC + 30 \text{ if } (R1 > R2)$
	BEQ	R1,R2 #-75	$PC = PC - 75 \text{ if } (R1 == R2)$
	BNE	R1,R2 #20	$PC = PC + 20 \text{ if } (R1 != R2)$
Register Transfer	MOVE	R1, R2	$R1 = R2$
Program Control	HALT		Halts, stops the execution
	NOP		No operation

### 3 Register Bank

Register	Code	Register	Code	Register	Code
R0	00000	R11	01011	R22	10110
R1	00001	R12	01100	R23	10111
R2	00010	R13	01101	R24	11000
R3	00011	R14	01110	R25	11001
R4	00100	R15	01111	R26	11010
R5	00101	R16	10000	R27	11011
R6	00110	R17	10001	R28	11100
R7	00111	R18	10010	R29	11101
R8	01000	R19	10011	R30	11110
R9	01001	R20	10100	R31	11111
R10	01010	R21	10101		

### 4 ALU Module Description

Input/Output	Description
operand1 (32 bits)	First operand
operand2 (32 bits)	Second operand
mode (4 bits)	Mode selector for the operation
en (1 bit)	Enable signal
out (32 bits)	Result of the operation

Mode	Operation
0000	Addition of operand1 and operand2
0001	Subtraction of operand2 from operand1
0010	Bitwise AND of operand1 and operand2
0011	Bitwise OR of operand1 and operand2
0100	Bitwise XOR of operand1 and operand2
0101	Bitwise negation of operand1
0110	Arithmetic left shift of operand1 by operand2 positions
0111	Logical left shift of operand1 by operand2 positions
1000	Arithmetic right shift of operand1 by operand2 positions
1001	Logical right shift of operand1 by operand2 positions
1010	Addition of operand1 and operand2 with operand2 zero-extended to 32 bits

The operation is performed when the **en** signal is asserted.

## 5 Instructions and Function Op-code

### 5.1 Arithmetic and Logical Instructions

Instructions	6-bit opcode	Function
ADD	00 0000	000001
SUB	00 0000	000010
AND	00 0000	000011
OR	00 0000	000100
XOR	00 0000	000101
NOT	00 0000	000110
SLA	00 0000	000111
SLL	00 0000	001000
SRA	00 0000	001001
SRL	00 0000	001010
ADDI	01 0000	-
SUBI	01 0001	-
ANDI	01 0010	-
ORI	01 0011	-
XORI	01 0100	-
NOTI	01 0101	-
SLAI	01 0110	-
SLLI	01 0111	-
SRAI	01 1000	-
SRLI	01 1001	-
MOVE	01 1010	-
LD	10 0001	-
ST	10 0010	-
BLT	11 0000	-
BGT	11 0001	-
BEQ	11 0010	-
BNE	11 0011	-
BR	11 0100	-

### 5.2 Special Type of Instructions

**NOP:** 11111111111111111111111111111110

**HLT:** 11111111111111111111111111111111

## 6 Instruction Formats

### 6.1 R-Type Instructions

Field	Bits	Description
Opcode	31-26	00 0000
RS	25-21	Source Register 1
RT	20-16	Source Register 2
RD	15-11	Destination Register
Shamt	10-6	Shift Amount
Func	5-0	Function

#### 6.1.1 Examples

Instruction	Binary Representation
ADD R1, R2, R3	000000 00010 00011 00001 00000 000001
SLA R5, R5, R7	000000 00101 00111 00101 00000 001001

### 6.2 I-Type Instructions

Field	Bits	Description
Opcode	31-26	opcode for different instructions
RS	25-21	Source Register
RT	20-16	Destination Register
Immediate Data	15-0	16-bit Imm. Data

#### 6.2.1 Examples

Instruction	Binary Representation
ADDI R2, R3, #25	010000 00011 00010 00000000000011001
MOVE R10, R5	011010 00101 01010 00000000000000000
LD R2, 10(R6)	100001 00110 00010 00000000000001010
ST R2, -2(R11)	100010 01011 00010 11111111111111110
BLT R2, R5, #-10	110000 00010 00101 11111111111101110
BGT R3, R5, #30	110001 00011 00101 00000000000111110
BEQ R4, R8, #-75	110010 00100 01000 11111111011010101
BNE R5, R8, #-75	110011 00101 01000 11111111011010101

### 6.3 J-Type Instructions

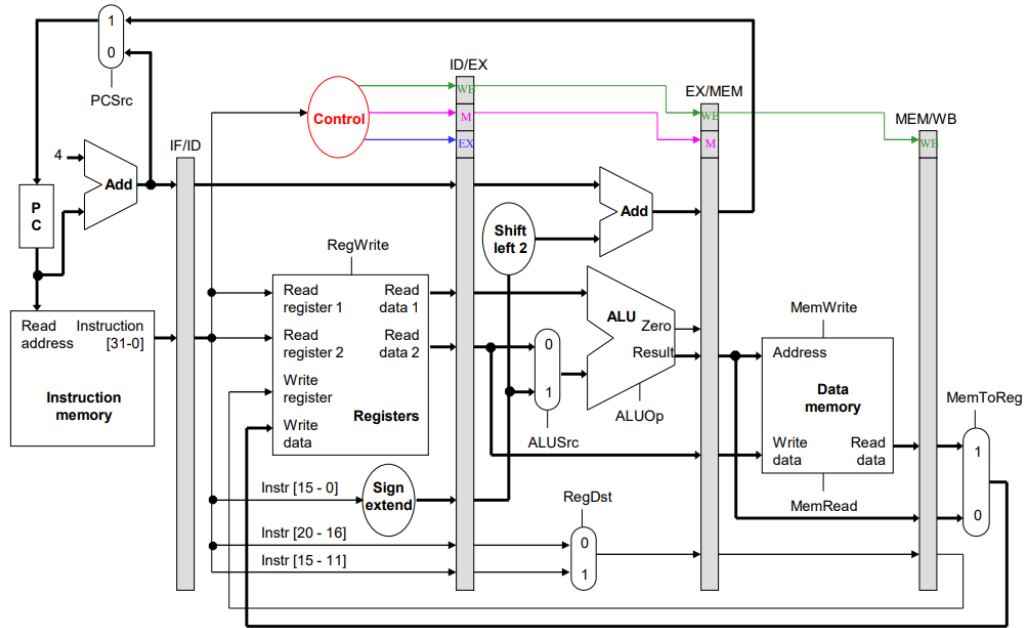
Field	Bits	Description
Opcode	31-26	opcode for different instructions
Immediate Data	25-0	26-bit Imm. Data

#### 6.3.1 Examples

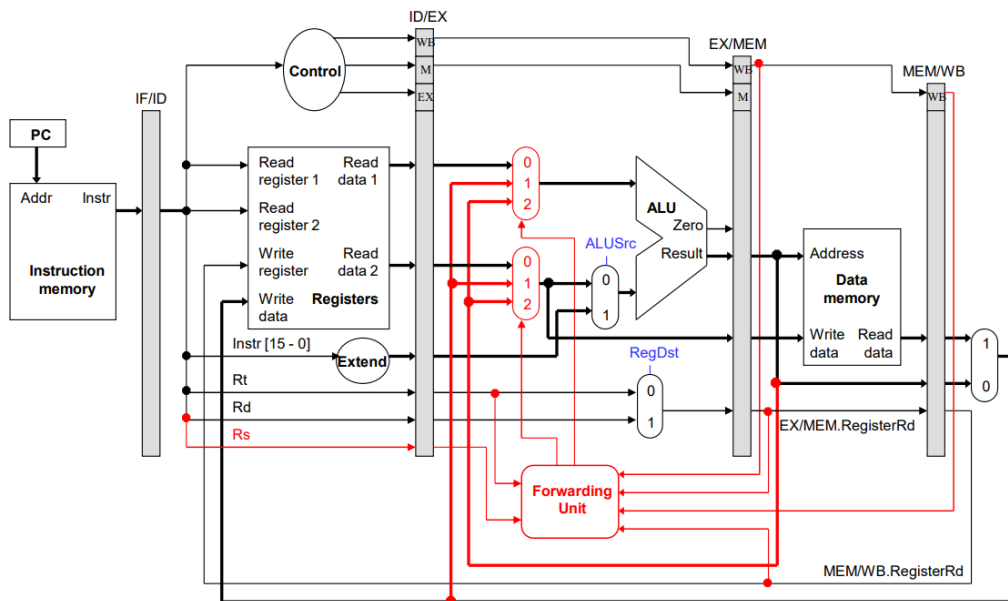
Instruction	Binary Representation
BR #10	110100 000000000000000000000001010

## 7 Datapath

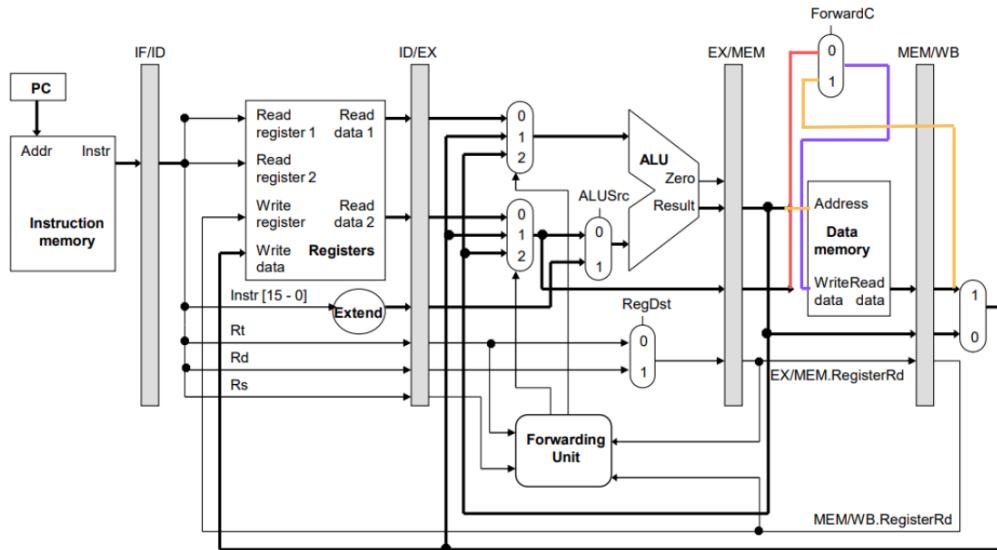
### 7.1 Simple Pipelined Datapath with Control Signals



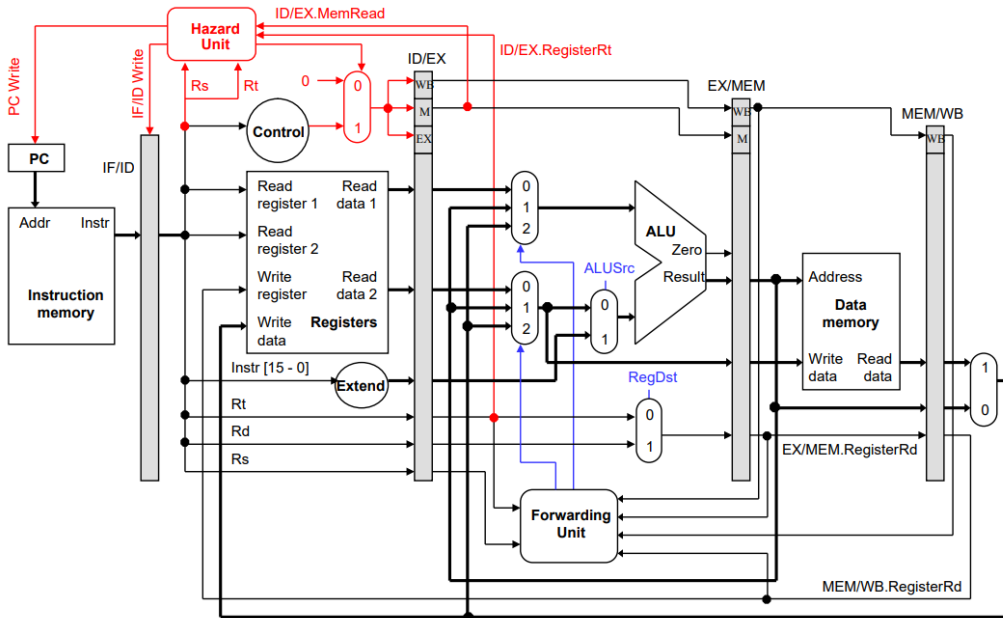
### 7.2 With Forwarding Unit



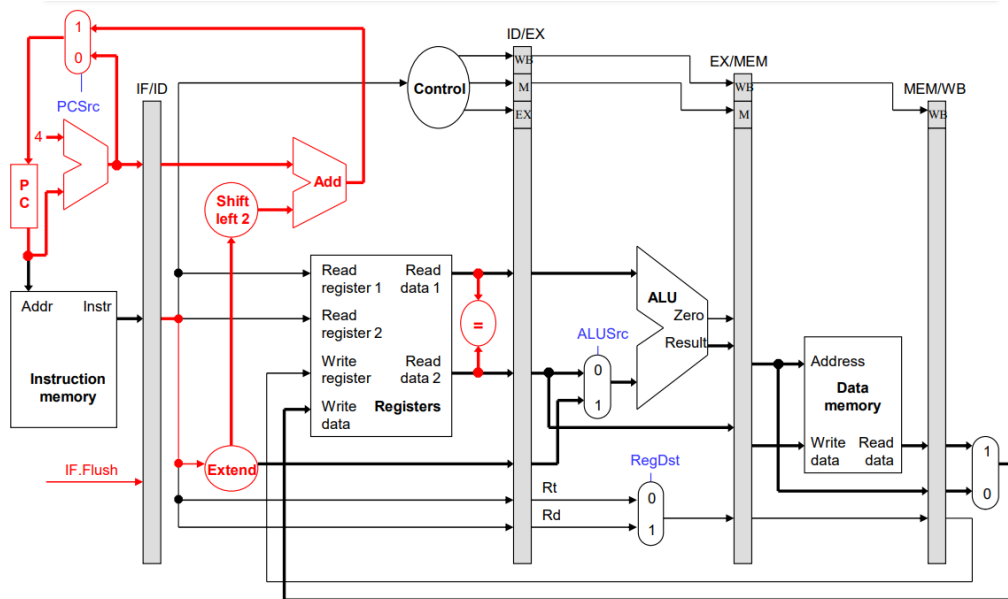
### 7.3 With Load/Store Bypassing Unit



### 7.4 With Hazard Detection Unit/Stall Control Unit



## 7.5 With Branching



## 8 Controller

### 8.1 Functioning of Main Control Signals

<b>Jump</b>	Mux Select input for selecting between Conditional jump and Unconditional jump
<b>PcSrc</b>	Mux Select input for selecting between PC + 4 and Branch Address, which depends on the output of comparator
<b>ALUSrc</b>	Mux Select input for selecting between Register Data 2 and Sign-extended Immediate data
<b>ALUfunc</b>	Control Signal for ALU operation
<b>RegDest</b>	Mux Select input for selecting between IR[20:16] (Rt) and IR[15:11] (Rd)
<b>ReadDmem</b>	Control Signal for reading data from Data Memory
<b>WriteDmem</b>	Control Signal for writing data into Data Memory
<b>RegWrite</b>	Control Signal for writing data into Register Bank
<b>MemtoReg</b>	Mux Select input for selecting between Read data from data memory and ALU result

## 8.2 Main Control Signals According to Function

Instr	ALUfunc	RegDest	ALUsrc	ReadDmem	WriteDmem	RegWrite	MemtoReg	Jump
ADD	0000	1	0	0	0	1	0	0
SUB	0001	1	0	0	0	1	0	0
AND	0010	1	0	0	0	1	0	0
OR	0011	1	0	0	0	1	0	0
XOR	0100	1	0	0	0	1	0	0
NOT	0101	1	0	0	0	1	0	0
SLA	0110	1	0	0	0	1	0	0
SLL	0111	1	0	0	0	1	0	0
SRA	1000	1	0	0	0	1	0	0
SRL	1001	1	0	0	0	1	0	0
ADDI	0000	1	1	0	0	1	0	0
SUBI	0001	1	1	0	0	1	0	0
ANDI	0010	1	1	0	0	1	0	0
ORI	0011	1	1	0	0	1	0	0
XORI	0100	1	1	0	0	1	0	0
NOTI	0101	1	1	0	0	1	0	0
SLAI	0110	1	1	0	0	1	0	0
SLLI	0111	1	1	0	0	1	0	0
SRAI	1000	1	1	0	0	1	0	0
SRLI	1001	1	1	0	0	1	0	0
MOVE	0000	1	1	0	0	1	0	0
LD	0000	1	1	1	0	1	0	0
ST	0000	1	1	0	1	0	1	0
BR	0000	0	0	0	0	0	0	1
BLT	0000	1	0	0	0	0	0	0
BGT	0000	1	0	0	0	0	0	0
BEQ	0000	1	0	0	0	0	0	0
BNE	0000	1	0	0	0	0	0	0

## 8.3 Verilog Code for the Controller

```

1 module controller (
2     input wire clk,reset,
3     input wire [31:0] instr,
4     input wire branch_condition,
5     output wire alusrc,
6     output wire [3:0] alufunc,
7     output wire regdest,
8     output wire readdmem,writedmem,
9     output wire regwrite,memtoreg,
10    output wire jump,pcsrc
11 );
12

```



```

13 reg [11:0] control_signals;
14 wire branch;
15 assign {alusrc,alufunc,regdest,
16         readmem,writemem,regwrite,memtoreg,jump,branch} = control_signals;
17
18 always @(negedge clk or negedge reset)
19 begin
20     if(reset) begin
21         control_signals <= 12'b0;
22     end
23     else if(instr == 32'b0)
24         control_signals <= 12'b0;
25     else #2 begin
26         case (instr[31:30])
27             2'b00 : begin
28                 {control_signals[11],control_signals[6:0]} <= 8'b01001000;
29                 case (instr[5:0])
30                     6'b000001 : control_signals[10:7] <= 4'b0000; // ADD
31                     6'b000010 : control_signals[10:7] <= 4'b0001; // SUB
32                     6'b000011 : control_signals[10:7] <= 4'b0010; // AND
33                     6'b000100 : control_signals[10:7] <= 4'b0011; // OR
34                     6'b000101 : control_signals[10:7] <= 4'b0100; // XOR
35                     6'b000110 : control_signals[10:7] <= 4'b0101; // NOT
36                     6'b000111 : control_signals[10:7] <= 4'b0110; // SLA
37                     6'b001000 : control_signals[10:7] <= 4'b0111; // SLL
38                     6'b001001 : control_signals[10:7] <= 4'b1000; // SRA
39                     6'b001010 : control_signals[10:7] <= 4'b1001; // SRL
40                 endcase
41             end
42             2'b01 : begin
43                 {control_signals[11],control_signals[6:0]} <= 8'b10001000;
44                 case (instr[31:26])
45                     6'b010000 : control_signals[10:7] <= 4'b0000; // ADDI
46                     6'b010001 : control_signals[10:7] <= 4'b0001; // SUBI
47                     6'b010010 : control_signals[10:7] <= 4'b0010; // ANDI
48                     6'b010011 : control_signals[10:7] <= 4'b0011; // ORI
49                     6'b010100 : control_signals[10:7] <= 4'b0100; // XORI
50                     6'b010101 : control_signals[10:7] <= 4'b0101; // NOTI
51                     6'b010110 : control_signals[10:7] <= 4'b0110; // SLAI
52                     6'b010111 : control_signals[10:7] <= 4'b0111; // SLLI
53                     6'b011000 : control_signals[10:7] <= 4'b1000; // SRAI
54                     6'b011001 : control_signals[10:7] <= 4'b1001; // SRLI
55                     6'b011010 : control_signals[10:7] <= 4'b0000; // MOVE
56                 endcase
57             end
58             2'b10 : begin
59                 case (instr[31:26])
60                     6'b100001 : control_signals <= 12'b100000101100; // LD
61                     6'b100010 : control_signals <= 12'b100000010000; // ST
62                 endcase

```

```

63         end
64         2'b11 : begin
65             case (instr[31:26])
66                 6'b110000 : control_signals <= 12'b0000000000001; // BLT
67                 6'b110001 : control_signals <= 12'b0000000000001; // BGT
68                 6'b110010 : control_signals <= 12'b0000000000001; // BEQ
69                 6'b110011 : control_signals <= 12'b0000000000001; // BNE
70                 6'b110100 : control_signals <= 12'b0000000000010; // BR
71             endcase
72         end
73     endcase
74 end
75 end
76
77 assign pcsrc = jump | (branch & branch_condition);
78
79 endmodule

```

## 8.4 Forwarding Control Signals

forwardA and forwardB are the mux select signals for the forwarding muxes which is fed to ALU.

### 8.4.1 Verilog Code for Simpler Forwarding Unit used in EX Stage

```

1 module forwarding_unit (
2     input wire [4:0] id_ex_rs,id_ex_rt,
3     input wire [4:0] ex_mem_rd,mem_wb_rd,
4     input wire ex_mem_regwrite,mem_wb_regwrite,
5
6     output wire [1:0] forwardA,forwardB
7 );
8
9 wire a1,a2,b1,b2,b3,b4,x,x1,y,y1;
10
11 /*
12 forward A : 00 : read from register bank
13 forward A : 01 : read from wb_writedata
14 forward A : 10 : read from ex_mem_result
15
16 forward A
17 -----
18 00 : default
19 01 : ex_mem_rd == id_ex_rs and ex_mem_regwrite == 1
20 10 : (not of 01 condition) and (mem_wb_regwrite == 1 and mem_wb_rd ==
    id_ex_rs)
21
22 same as forwardB, use rt in place of rs.
23 */
24

```

```

25
26 assign a1 = (ex_mem_regwrite == 1'b1) ;
27 assign b1 = (ex_mem_rd == id_ex_rs);
28 assign x = a1 & b1;
29
30 assign a2 = (mem_wb_regwrite == 1'b1) ;
31 assign b2 = (mem_wb_rd == id_ex_rs);
32 assign y = a2 & b2;
33
34 assign forwardA[1] = x;
35 assign forwardA[0] = ~x & y ;
36
37
38 assign b3 = (ex_mem_rd == id_ex_rt);
39 assign x1 = a1 & b3;
40
41 assign b4 = (mem_wb_rd == id_ex_rt);
42 assign y1 = a2 & b4;
43
44 assign forwardB[1] = x1;
45 assign forwardB[0] = ~x1 & y1 ;
46
47 endmodule

```

#### 8.4.2 Verilog Code for Modified Forwarding Unit used in ID Stage

```

1 module forwarding_unit_id (
2     input wire [4:0] rs,rt,
3     input wire [4:0] dest,ex_mem_destadd,mem_wb_destadd,
4     input wire id_ex_regwrite,ex_mem_readdmem,ex_mem_regwrite,
5         ex_mem_memtoreg,mem_wb_regwrite,
6
7     output wire [2:0] forwardA,forwardB
8 );
9 /*
10 forward A : 000 : read from register bank
11 forward A : 001 : read from data memory
12 forward A : 010 : read from ex_mem_result
13 forward A : 011 : read from result of ALU
14 forward A : 100 : read from wb_writedata
15
16 forward A
17 -----
18 000 : default
19 001 : ex_mem_readdmem == 1 and ex_mem_destadd == rs and ex_mem_memtoreg ==
    1 and ex_mem_regwrite == 1
20 010 : (not of 01 condition) and (ex_mem_destadd == rs and ex_mem_memtoreg
    == 0 and ex_mem_regwrite == 1)

```

```

20 011 : (not of 01 condition) and (not of 10 condition) and (dest == rs and
    id_ex_regwrite == 1)
21 100 : (not of 01 condition) and (not of 10 condition) and (not of 11
    condition) and (mem_wb_regwrite == 1 and mem_wb_destadd == rs)
22
23 same as forwardB, use rt in place of rs.
24 */
25 reg [2:0] fA,fB;
26
27 assign forwardA = fA;
28 assign forwardB = fB;
29
30 always @* begin
31
32     if (ex_mem_readdmem == 1 && ex_mem_destadd == rs && ex_mem_mentoreg == 1
        && ex_mem_regwrite == 1) begin
33         fA <= 3'b001;
34     end
35     else if (!(ex_mem_readdmem == 1 && ex_mem_destadd == rs &&
        ex_mem_mentoreg == 1 && ex_mem_regwrite == 1) &&
36         (ex_mem_destadd == rs && ex_mem_mentoreg == 0 && ex_mem_regwrite
            == 1)) begin
37         fA <= 3'b010;
38     end
39     else if (!(ex_mem_readdmem == 1 && ex_mem_destadd == rs &&
        ex_mem_mentoreg == 1 && ex_mem_regwrite == 1) &&
40         !((ex_mem_destadd == rs && ex_mem_mentoreg == 0 &&
            ex_mem_regwrite == 1)) &&
41         (dest == rs && id_ex_regwrite == 1)) begin
42         fA <= 3'b011;
43     end
44     else if (!(ex_mem_readdmem == 1 && ex_mem_destadd == rs &&
        ex_mem_mentoreg == 1 && ex_mem_regwrite == 1) &&
45         !((ex_mem_destadd == rs && ex_mem_mentoreg == 0 &&
            ex_mem_regwrite == 1)) &&
46         !((dest == rs && id_ex_regwrite == 1)) &&
47         (mem_wb_regwrite == 1 && mem_wb_destadd == rs)) begin
48         fA = 3'b100;
49     end
50     else
51         fA <= 3'b000;
52 end
53 always @* begin
54
55     if (ex_mem_readdmem == 1 && ex_mem_destadd == rt && ex_mem_mentoreg == 1
        && ex_mem_regwrite == 1) begin
56         fB <= 3'b001;
57     end
58     else if (!(ex_mem_readdmem == 1 && ex_mem_destadd == rt &&
        ex_mem_mentoreg == 1 && ex_mem_regwrite == 1) &&

```

```

59         (ex_mem_destadd == rt && ex_mem_membreg == 0 && ex_mem_regwrite
        == 1)) begin
60     fB <= 3'b010;
61 end
62 else if (!(ex_mem_readdmem == 1 && ex_mem_destadd == rt &&
        ex_mem_membreg == 1 && ex_mem_regwrite == 1) &&
63         !((ex_mem_destadd == rt && ex_mem_membreg == 0 &&
        ex_mem_regwrite == 1)) &&
64         (dest == rt && id_ex_regwrite == 1)) begin
65     fB <= 3'b011;
66 end
67 else if (!(ex_mem_readdmem == 1 && ex_mem_destadd == rt &&
        ex_mem_membreg == 1 && ex_mem_regwrite == 1) &&
68         !((ex_mem_destadd == rt && ex_mem_membreg == 0 &&
        ex_mem_regwrite == 1)) &&
69         !((dest == rt && id_ex_regwrite == 1)) &&
70         (mem_wb_regwrite == 1 && mem_wb_destadd == rt)) begin
71     fB <= 3'b100;
72 end
73 else
74     fB <= 3'b000;
75 end
76
77 endmodule

```

## 8.5 Stall Control Signals

```

1 module stall_control (
2     input wire id_ex_readdmem,
3     input wire writedmem,
4     input wire [4:0] id_ex_rt,
5     input wire [4:0] rs,rt,
6
7     output wire stall,pc_write,if_id_write
8 );
9 wire x,y,z;
10 /*
11 if ((id_ex_readdmem == 1) and ((id_ex_rt == rs) or (id_ex_rt == rt)))
12     then stall
13 */
14 assign x = id_ex_readdmem == 1;
15 assign y = (id_ex_rt == rs) | (id_ex_rt == rt);
16 assign z = x & y & ~writedmem;
17
18 assign {stall,pc_write,if_id_write} = (z==1) ? 3'b100 : 3'b011;
19
20 endmodule

```

## 8.6 Flush Control Signals

```
1 module flush_control (
2     input wire alusrc,
3     input wire [3:0] alufunc,
4     input wire regdest, readdmem, writedmem, regwrite, memtoreg,
5     input wire flush,
6     output wire id_alusrc,
7     output wire [3:0] id_alufunc,
8     output wire id_regdest, id_readdmem, id_writedmem, id_regwrite, id_memtoreg
9 );
10
11 wire notflush;
12
13 assign id_alusrc = (flush == 1'b1) ? 1'b0 : alusrc;
14 assign id_alufunc[0] = (flush == 1'b1) ? 1'b0 : alufunc[0];
15 assign id_alufunc[1] = (flush == 1'b1) ? 1'b0 : alufunc[1];
16 assign id_alufunc[2] = (flush == 1'b1) ? 1'b0 : alufunc[2];
17 assign id_alufunc[3] = (flush == 1'b1) ? 1'b0 : alufunc[3];
18 assign id_regdest = (flush == 1'b1) ? 1'b0 : regdest;
19 assign id_readdmem = (flush == 1'b1) ? 1'b0 : readdmem;
20 assign id_writedmem = (flush == 1'b1) ? 1'b0 : writedmem;
21 assign id_regwrite = (flush == 1'b1) ? 1'b0 : regwrite;
22 assign id_memtoreg = (flush == 1'b1) ? 1'b0 : memtoreg;
23
24
25 endmodule
```

## 8.7 Load / Store Bypassing Unit

```
1 module load_store_bypassing_unit (
2     input wire mem_wb_memtoreg, ex_mem_writedmem,
3     input wire [4:0] ex_mem_destadd, mem_wb_destadd,
4
5     output wire forwarddatamem
6 );
7
8 wire x, y;
9
10 assign x = ex_mem_destadd == mem_wb_destadd;
11 assign y = mem_wb_memtoreg & ex_mem_writedmem;
12
13 assign forwarddatamem = x & y;
14
15 endmodule
```

## 9 Verilog Code for MIPS Pipeline / Top Module

```
1  'include "mux.v"
2  'include "mux3.v"
3  'include "mux5.v"
4  'include "register.v"
5  'include "register_bank.v"
6  'include "instr_mem.v"
7  'include "data_mem.v"
8  'include "adder.v"
9  'include "ALU.v"
10 'include "signext.v"
11 'include "condition.v"
12 'include "controller.v"
13 'include "forwarding_unit.v"
14 'include "forwarding_unit_id.v"
15 'include "stall_control.v"
16 'include "flush_control.v"
17 'include "load_store_bypassing_unit.v"
18
19 module mips(
20     input wire clk,clkbar,reset
21 );
22
23 // Define wires used in datapath
24 wire readim,pc_en;
25 wire [31:0] pcnext,pcbranch,pcplus4,pcout,instr,instr_out;
26 wire [31:0] if_id_npcout,if_id_irout;
27 wire if_id_flush;
28
29 wire [31:0] rData1,rData2,imm,offset;
30 wire [4:0] rs,rt,rd;
31 wire [31:0] ReadData1out,ReadData2out;
32 wire [3:0] alufunc;
33 wire branch_condition,alusrc,regdest,readdmem,writedmem,regwrite,memtoreg,
    jump,pcsrc;
34 wire stall,if_flush,flush;
35 wire if_id_write,pc_write;
36 wire [3:0] id_alufunc;
37 wire id_alusrc,id_regdest,id_branch,id_readdmem,id_writedmem,id_pcsrc,
    id_regwrite,id_memtoreg;
38 wire [31:0] id_ex_rData1,id_ex_rData2,id_ex_immout;
39 wire [4:0] id_ex_rt,id_ex_rd,id_ex_rs;
40 wire [3:0] id_ex_alufunc;
41 wire id_ex_alusrc,id_ex_regdest,id_ex_readdmem,id_ex_writedmem,
    id_ex_regwrite,id_ex_memtoreg;
42
43 wire [2:0] forwardA,forwardB;
44 wire [31:0] aluin1,aluin2,writedatadmem,result;
45 wire zero;
```

```

46 wire [4:0] dest;
47 wire ex_mem_zero;
48 wire [4:0] ex_mem_destadd;
49 wire [31:0] ex_mem_result,ex_mem_writedata;
50 wire ex_mem_readdmem,ex_mem_writedmem,ex_mem_regwrite,ex_mem_memtoreg;
51
52 wire forwardC;
53 wire [31:0] finalWriteData,readmemData;
54 wire [31:0] mem_wb_readData1,mem_wb_readData2;
55 wire [4:0] mem_wb_destadd;
56 wire mem_wb_regwrite,mem_wb_memtoreg;
57
58 wire [31:0] wb_writedata;
59
60 // IF Stage
61 assign pcwriteen = pc_en & pc_write; // PC enable
62 register pcreg(clk,reset,pcwriteen,pcnext,pcout); // Program Counter
    Register
63 instr_mem imem(clk,1'b1,pcout[7:2],instr,pc_en); // Instruction Memory
64 adder addpc(pcout,32'h00000004,pcplus4); // Adder for PC + 4
65 mux muxbr(if_id_npcout,pcbranch,pcsrc,pcnext); // Mux for Branching
66
67 // IF/ID Latch Stage
68 register IF_ID_npcreg(clkbar,reset,if_id_write,pcplus4,if_id_npcout);
69 register IF_ID_irreg(clkbar,reset,if_id_write,instr,if_id_irout);
70 register #(1) IF_ID_flushbit(clkbar,reset,if_id_write,if_flush,if_id_flush)
    ;
71
72 // ID Stage
73 assign rs = if_id_irout[25:21];
74 assign rt = if_id_irout[20:16];
75 assign rd = if_id_irout[15:11];
76 register_bank rbank(clk,mem_wb_regwrite,reset,rs,rt,mem_wb_destadd,
    wb_writedata,rData1,rData2); // Register Bank
77 signext ext(if_id_irout[15:0],imm); // Sign Extend
78 mux offsetmux({imm[29:0],2'b00},{4{if_id_irout[25]}},if_id_irout[25:0],2'
    b00},jump,offset);
79 adder pcbranchadder(if_id_npcout,offset,pcbranch); // Next Address
80
81 // ID Stage Forwarding Unit
82 forwarding_unit_id forwarding(rs,rt,
83     dest,ex_mem_destadd,mem_wb_destadd,id_ex_regwrite,ex_mem_readdmem,
        ex_mem_regwrite,ex_mem_memtoreg,mem_wb_regwrite,forwardA,
        forwardB);
84 mux5 readdatamux1(rData1,readmemData,ex_mem_result,result,wb_writedata,
    forwardA,ReadData1out); // Final Read Data 1
85 mux5 readdatamux2(rData2,readmemData,ex_mem_result,result,wb_writedata,
    forwardB,ReadData2out); // Final Read Data 2
86
87 // controller

```



```

88 controller controllerunit(clk,reset,if_id_irout,branch_condition,alusrc,
    alufunc,regdest,readdmem,writedmem,regwrite,memtoreg,jump,pcsrc);
89
90 // comparison block
91 condition comparisonblock(branch_condition,if_id_irout[27:26],ReadData1out,
    ReadData2out);
92
93 // Stall Control
94 stall_control stallunit(id_ex_readmem,writedmem,id_ex_rt,rs,rt,stall,
    pc_write,if_id_write);
95
96 // flush control
97 assign if_flush = pcsrc;
98 assign flush = if_flush | stall;
99 flush_control flushunit(alusrc,alufunc,regdest,readdmem,writedmem,regwrite,
    memtoreg,flush,
100                                id_alusrc,id_alufunc,id_regdest,id_readmem,
                                id_writedmem,id_regwrite,id_memtoreg);
101
102 // ID/EX Stage
103 register ID_EX_rData1(clkbar,reset,1'b1,ReadData1out,id_ex_rData1);
104 register ID_EX_rData2(clkbar,reset,1'b1,ReadData2out,id_ex_rData2);
105 register ID_EX_imm(clkbar,reset,1'b1,imm,id_ex_immout);
106 register #(5) ID_EX_rs(clkbar,reset,1'b1,rs,id_ex_rs);
107 register #(5) ID_EX_rt(clkbar,reset,1'b1,rt,id_ex_rt);
108 register #(5) ID_EX_rd(clkbar,reset,1'b1,rd,id_ex_rd);
109
110 // ID/EX Stage Control Signals for EX Stage
111 register #(1) ID_EX_ALUsrc(clkbar,reset,1'b1,id_alusrc,id_ex_alusrc);
112 register #(4) ID_EX_ALUfunc(clkbar,reset,1'b1,id_alufunc,id_ex_alufunc);
113 register #(1) ID_EX_Regdest(clkbar,reset,1'b1,id_regdest,id_ex_regdest);
114
115 // ID/EX Stage Control Signals for MEM Stage
116 register #(1) ID_EX_MemRead(clkbar,reset,1'b1,id_readmem,id_ex_readmem);
117 register #(1) ID_EX_Memwrite(clkbar,reset,1'b1,id_writedmem,id_ex_writedmem
    );
118
119 // ID/EX Stage Control Signals for WB Stage
120 register #(1) ID_EX_Regwrite(clkbar,reset,1'b1,id_regwrite,id_ex_regwrite);
121 register #(1) ID_EX_MemtoReg(clkbar,reset,1'b1,id_memtoreg,id_ex_memtoreg);
122
123 // Forwarding Unit
124 // forwarding_unit forward(id_ex_rs,id_ex_rt,ex_mem_destadd,mem_wb_destadd,
    ex_mem_regwrite,mem_wb_regwrite,forwardA,forwardB);
125
126 // EX Stage
127 // mux3 mux3A(id_ex_rData1,wb_writedata,ex_mem_result,forwardA,aluin1);
128 // mux3 mux3B(id_ex_rData2,wb_writedata,ex_mem_result,forwardB,
    writadatadmem);
129 assign aluin1 = id_ex_rData1;

```

```

130 mux alumux(id_ex_rData2,id_ex_immout,id_ex_alusrc,aluin2); // ALU mux
131 ALU alu(aluin1,aluin2,id_ex_alufunc,1'b1,zero,result); // Arithmetic Logic
    Unit
132 mux #(5) destmux(id_ex_rt,id_ex_rd,id_ex_regdest,dest); // Mux for
    Destination Address
133
134 // EX/MEM Stage
135 register EX_MEM_resultreg(clkbar,reset,1'b1,result,ex_mem_result);
136 register #(1) EX_MEM_zeroreg(clkbar,reset,1'b1,zero,ex_mem_zero);
137 register EX_MEM_writeData(clkbar,reset,1'b1,id_ex_rData2,ex_mem_writeData);
138 register #(5) EX_MEM_destreg(clkbar,reset,1'b1,dest,ex_mem_destadd);
139
140 // EX/MEM Stage Control Signals for MEM Stage
141 register #(1) EX_MEM_MemRead(clkbar,reset,1'b1,id_ex_readdmem,
    ex_mem_readdmem);
142 register #(1) EX_MEM_Memwrite(clkbar,reset,1'b1,id_ex_writedmem,
    ex_mem_writedmem);
143
144 // EX/MEM Stage Control Signals for WB Stage
145 register #(1) EX_MEM_Regwrite(clkbar,reset,1'b1,id_ex_regwrite,
    ex_mem_regwrite);
146 register #(1) EX_MEM_MemtoReg(clkbar,reset,1'b1,id_ex_memtoReg,
    ex_mem_memtoReg);
147
148 // MEM Stage
149 load_store_bypassing_unit loadstore(mem_wb_memtoReg,ex_mem_writedmem,
    ex_mem_destadd,mem_wb_destadd,forwardC); // load store btypassing unit
150 mux loadstoreselect(ex_mem_writeData,mem_wb_readData2,forwardC,
    finalWriteData);
151 data_mem dmem(clk,reset,ex_mem_writedmem,ex_mem_readdmem,ex_mem_result,
    finalWriteData,readmemData); // Data Memory
152
153 // MEM/WB Stage
154 register MEM_WB_readData1(clkbar,reset,1'b1,ex_mem_result,mem_wb_readData1)
    ;
155 register MEM_WB_readData2(clkbar,reset,1'b1,readmemData,mem_wb_readData2);
156 register #(5) MEM_WB_destreg(clkbar,reset,1'b1,ex_mem_destadd,
    mem_wb_destadd);
157
158 // MEM/WB Stage Control Signals for WB Stage
159 register #(1) MEM_WB_Regwrite(clkbar,reset,1'b1,ex_mem_regwrite,
    mem_wb_regwrite);
160 register #(1) MEM_WB_MemtoReg(clkbar,reset,1'b1,ex_mem_memtoReg,
    mem_wb_memtoReg);
161
162 // WB Stage
163 mux wbmux(mem_wb_readData1,mem_wb_readData2,mem_wb_memtoReg,wb_writedata);
    // Mux for Writeback
164
165 endmodule

```

## 10 Problem Solving Using the Processor

### 10.1 Computing GCD of Two Numbers

To find the greatest common divisor (GCD) of two numbers, we'll use registers *R1* and *R2* to hold the first and second numbers, respectively. Additionally, we'll store these values in `data_memory[0]` and `data_memory[1]` respectively. Finally, the calculated GCD will be stored in `data_memory[2]`.

Instruction	Binary Code
ADDI R1, R0, #143;	010000_00000_00001_0000000010001111
ADDI R2, R0, #78;	010000_00000_00010_0000000001001110
ST R1, 0(R0);	100010_00000_00001_0000000000000000
ST R2, 1(R0);	100010_00000_00010_0000000000000001
BLT R1, R2, #2; // if R1<R2 (X)	110000_00011_00011_0000000000000010
BGT R1, R2, #4; // if R1>R2 (Y)	110001_00011_00011_0000000000000100
BEQ R1, R2, #6; // if R1=R2 (Z)	110010_00011_00011_0000000000000110
SUB R3, R2, R1; // if R1<R2 (X)	000000_00010_00001_00011_00000_000010
MOVE R2, R3;	011010_00011_00010_0000000000000000
BR #-6;	110100_111111111111111111111111010
SUB R3, R1, R2; // if R1>R2 (Y)	000000_00001_00010_00011_00000_000010
MOVE R1, R3;	011010_00011_00001_0000000000000000
BR #-9;	110100_1111111111111111111111110111
ST R1, 2(R0); // if R1=R2 (Z)	100010_00000_00001_0000000000000010
HLT;	11111111111111111111111111111111

## 10.2 Output of Pipelined Processor

```

0 register values :      x      x      x data_memory:      x      x      x
5 register values :      0      0      0 data_memory:      0      0      0
WARNING: mips_tb.v:25: $readmemh(gcd_test.txt): Not enough words in the file for the requested range [0:63].
55 register values :     143      0      0 data_memory:      0      0      0
65 register values :     143      78      0 data_memory:      0      0      0
66 register values :     143      78      0 data_memory:     143      0      0
76 register values :     143      78      0 data_memory:     143      78      0
115 register values :    143      78      65 data_memory:     143      78      0
125 register values :     65      78      65 data_memory:     143      78      0
155 register values :     65      78      13 data_memory:     143      78      0
165 register values :     65      13      13 data_memory:     143      78      0
205 register values :     65      13      52 data_memory:     143      78      0
215 register values :     52      13      52 data_memory:     143      78      0
255 register values :     52      13      39 data_memory:     143      78      0
265 register values :     39      13      39 data_memory:     143      78      0
305 register values :     39      13      26 data_memory:     143      78      0
315 register values :     26      13      26 data_memory:     143      78      0
355 register values :     26      13      13 data_memory:     143      78      0
365 register values :     13      13      13 data_memory:     143      78      0
406 register values :     13      13      13 data_memory:     143      78      13

```

## 10.3 Output of Non-Pipelined Processor

```

0 register values :      x      x      x data_memory:      x      x      x
5 register values :      0      0      0 data_memory:      0      0      0
WARNING: mips_tb.v:24: $readmemh(gcd_test.txt): Not enough words in the file for the requested range [0:63].
75 register values :     143      0      0 data_memory:      0      0      0
125 register values :     143      78      0 data_memory:      0      0      0
165 register values :     143      78      0 data_memory:     143      0      0
175 register values :     143      78      0 data_memory:     143      0      0
215 register values :     143      78      0 data_memory:     143      78      0
225 register values :     143      78      0 data_memory:     143      78      0
275 register values :     143      78      65 data_memory:     143      78      0
425 register values :     65      78      65 data_memory:     143      78      0
525 register values :     65      78     -13 data_memory:     143      78      0
625 register values :     65      78      13 data_memory:     143      78      0
675 register values :     65      13      13 data_memory:     143      78      0
775 register values :     65      13      52 data_memory:     143      78      0
925 register values :     52      13      52 data_memory:     143      78      0
1025 register values :    52      13      39 data_memory:     143      78      0
1175 register values :    39      13      39 data_memory:     143      78      0
1275 register values :    39      13      26 data_memory:     143      78      0
1425 register values :    26      13      26 data_memory:     143      78      0
1525 register values :    26      13      13 data_memory:     143      78      0
1675 register values :    13      13      13 data_memory:     143      78      0
1775 register values :    13      13      0 data_memory:     143      78      0
1965 register values :    13      13      0 data_memory:     143      78      13
1975 register values :    13      13      0 data_memory:     143      78      13

```

## 10.4 Conclusion

As observed from the outputs of both processors, the first one takes 406 units of time to compute the GCD, while the second one takes 1975 units. This results in a speed-up of  $\frac{1975}{406} \approx 4.8567$ .

## 10.5 Sorting a set of 10 integers using bubble sort

The program implements the bubble sort algorithm to sort a set of 10 integers stored in the array `arr`. The array, starting at address 100, contains the elements [20, 50, 10, 30, 70, 40, 60, 80, 100, 90]. In the testbench, the entire array is loaded into Data Memory for sorting.

Instruction	Binary Code
ADDI R1, R0, #100; // arr[0]	010000_00000_00001_00000000001100100
MOVE R2, R0; // i=0	011010_00000_00010_0000000000000000
MOVE R3, R0; // j=0	011010_00000_00011_0000000000000000
ADDI R4, R0, #10; // n=10	010000_00000_00100_0000000000001010
ADDI R5, R0, #10; // n-i for inner loop	010000_00000_00101_0000000000001010
MOVE R6, R1; // for iterating addr by i	011010_00001_00110_0000000000000000
MOVE R7, R1; // for iterating addr by j	011010_00001_00111_0000000000000000
SUBI R4, R4, #1; // decrement n	010001_00100_00100_0000000000000001
MOVE R3, R0; // outer_loop // j=0	011010_00000_00011_0000000000000000
SUBI R5, R5, #1; // decreasing size for inner_loop	010001_00101_00101_0000000000000001
ADD R7, R0, R1; // resetting addr itr j	000000_00000_00111_00000_00000_000010
LD R8, 0(R7); // inner_loop // arr[j]	100001_00111_01000_0000000000000000
ADDI R7, R7, #1; // addr itr j += 1	010000_00111_00111_0000000000000001
LD R9, 0(R7); // arr[j+1]	100001_00111_01001_0000000000000000
ADDI R3, R3, #1; // j++	010000_00011_00011_0000000000000001
BLT R8, R9, #3; // if R8 < R9 then Branch(Y)	110000_01000_01001_0000000000000011
ST R8, 0(R7); // swap	100010_00111_01000_0000000000000000
ST R9, -1(R7);	100010_00111_01001_1111111111111111
LD R9, 0(R7);	100001_00111_01001_0000000000000000
BEQ R3, R5, #1; // Exiting from inner_loop(W)(Y)	110010_00011_00101_0000000000000001
BR #-10; // Address to inner_loop(Z)	110100_111111111111111111111110110
ADDI R2, R2, #1; // After Exiting From inner_loop	010000_00010_00010_0000000000000001
BNE R2, R4, #-15; // i!=n, go to outer loop (X)	110011_00010_00100_11111111111110001
NOP;	11111111111111111111111111111110
HLT;	11111111111111111111111111111111

## 10.6 Output of Pipelined Processor

20 Array:	20	50	10	30	70	40	60	80	100	90
276 Array:	20	50	50	30	70	40	60	80	100	90
286 Array:	20	10	50	30	70	40	60	80	100	90
376 Array:	20	10	50	50	70	40	60	80	100	90
386 Array:	20	10	30	50	70	40	60	80	100	90
546 Array:	20	10	30	50	70	70	60	80	100	90
556 Array:	20	10	30	50	40	70	60	80	100	90
646 Array:	20	10	30	50	40	70	70	80	100	90
656 Array:	20	10	30	50	40	60	70	80	100	90
886 Array:	20	10	30	50	40	60	70	80	100	100
896 Array:	20	10	30	50	40	60	70	80	90	100
1026 Array:	20	20	30	50	40	60	70	80	90	100
1036 Array:	10	20	30	50	40	60	70	80	90	100
1266 Array:	10	20	30	50	50	60	70	80	90	100
1276 Array:	10	20	30	40	50	60	70	80	90	100

## 10.7 Output of Non-Pipelined Processor

20 Array:	20	50	10	30	70	40	60	80	100	90
1365 Array:	20	50	50	30	70	40	60	80	100	90
1375 Array:	20	50	50	30	70	40	60	80	100	90
1415 Array:	20	10	50	30	70	40	60	80	100	90
1425 Array:	20	10	50	30	70	40	60	80	100	90
1965 Array:	20	10	50	50	70	40	60	80	100	90
1975 Array:	20	10	50	50	70	40	60	80	100	90
2015 Array:	20	10	30	50	70	40	60	80	100	90
2025 Array:	20	10	30	50	70	40	60	80	100	90
3015 Array:	20	10	30	50	70	70	60	80	100	90
3025 Array:	20	10	30	50	70	70	60	80	100	90
3065 Array:	20	10	30	50	40	70	60	80	100	90
3075 Array:	20	10	30	50	40	70	60	80	100	90
3615 Array:	20	10	30	50	40	70	70	80	100	90
3625 Array:	20	10	30	50	40	70	70	80	100	90
3665 Array:	20	10	30	50	40	60	70	80	100	90
3675 Array:	20	10	30	50	40	60	70	80	100	90
5115 Array:	20	10	30	50	40	60	70	80	100	100
5125 Array:	20	10	30	50	40	60	70	80	100	100
5165 Array:	20	10	30	50	40	60	70	80	90	100
5175 Array:	20	10	30	50	40	60	70	80	90	100
6015 Array:	20	20	30	50	40	60	70	80	90	100
6025 Array:	20	20	30	50	40	60	70	80	90	100
6065 Array:	10	20	30	50	40	60	70	80	90	100
6075 Array:	10	20	30	50	40	60	70	80	90	100
7515 Array:	10	20	30	50	50	60	70	80	90	100
7525 Array:	10	20	30	50	50	60	70	80	90	100
7565 Array:	10	20	30	40	50	60	70	80	90	100
7575 Array:	10	20	30	40	50	60	70	80	90	100

## 10.8 Conclusion

As observed from the outputs of both processors, the first one takes 1276 units of time to compute the GCD, while the second one takes 7575 units. This results in a speed-up of  $\frac{7575}{1276} \approx 5.936$ .