Pranjali Pramod Pathre
2019112002

PROJECT REPORT

10rd March 2021

OVERVIEW

Processor architecture design based on the Y86 ISA using Verilog.

INSTRUCTIONS TO RUN THE CODE FILE

1. Download the codes.
2. Run in terminal: iverilog -o processor_test processor_test.v processor.v
3. ./a.out
4. gtkwave processor_test.vcd
5. For changing the instruction set change the file instr.mem to the required file.
6. Individual testbenches have also been provided for checking the functionality for different modules.

SPECIFICATIONS

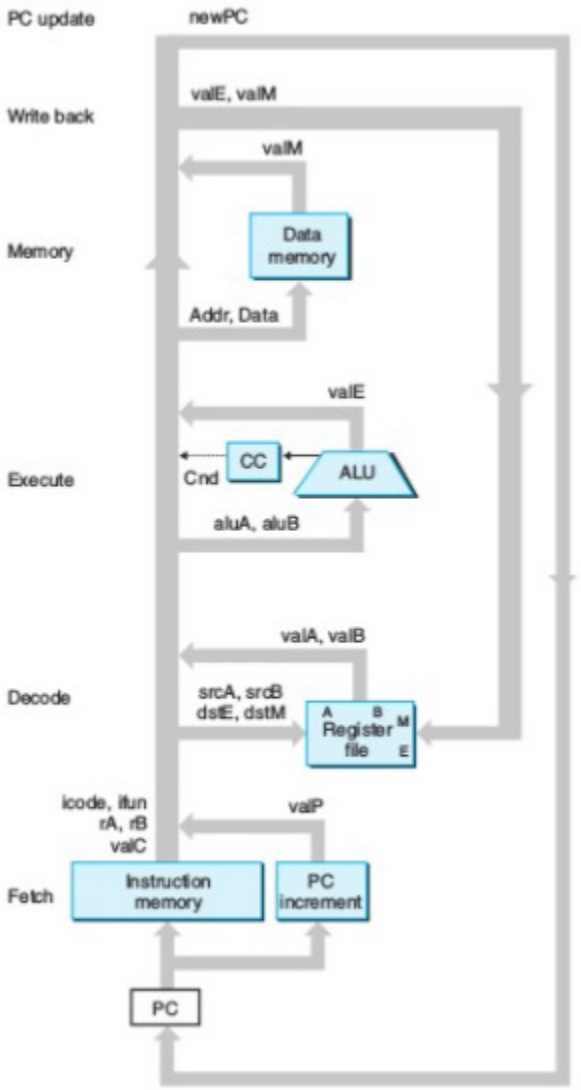
ALU is built with the following functionality:

- ADD - 64bits
- SUB - 64bits
- AND - 64bits
- XOR - 64bits

Input and output of all the blocks is in signed 2's complement.

SEQ Hardware Structure

The computations required to implement all of the Y86-64 instructions can be organized as a series of six basic stages: fetch, decode, execute, memory, write back, and PC update. The below given figure shows an abstract view of a hardware structure that can perform these computations.



The hardware units are associated with the different processing stages:

Fetch. Using the program counter register as an address, the instruction memory reads the bytes of an instruction. The PC incrementer computes valP the increment program counter.

Decode. The register file has two read ports, A and B, via which register values valA and valB are read simultaneously.

Execute. The execute stage uses the arithmetic/logic (ALU) unit for different purposes according to the instruction type. For integer operations, it performs the specified operation. For other

instructions, it serves as an adder to compute an incremented or decremented stack pointer, to compute an effective address, or simply to pass one of its inputs to its outputs by adding zero.

The condition code register (CC) holds the three condition code bits. New values for the condition codes are computed by the ALU. When executing a conditional move instruction, the decision as to whether or not to update the destination register is computed based on the condition codes and move condition. Similarly, when executing a jump instruction, the branch signal Cnd is computed based on the condition codes and the jump type.

Memory. The data memory reads or writes a word of memory when executing a memory instruction. The instruction and data memories access the same memory locations, but for different purposes. Write back. The register file has two write ports. Port E is used to write values computed by the ALU, while port M is used to write values read from the data memory.

PC update. The new value of the program counter is selected to be either valP, the address of the next instruction, valC, the destination address specified by a call or jump instruction, or valM, the return address read from memory.

The below given figure gives a more detailed view of the hardware required to implement SEQ.

To map the computations into hardware, we want to implement control logic that will transfer the data between the different hardware units and operate these units in such a way that the specified operations are performed for each of the different instruction types.

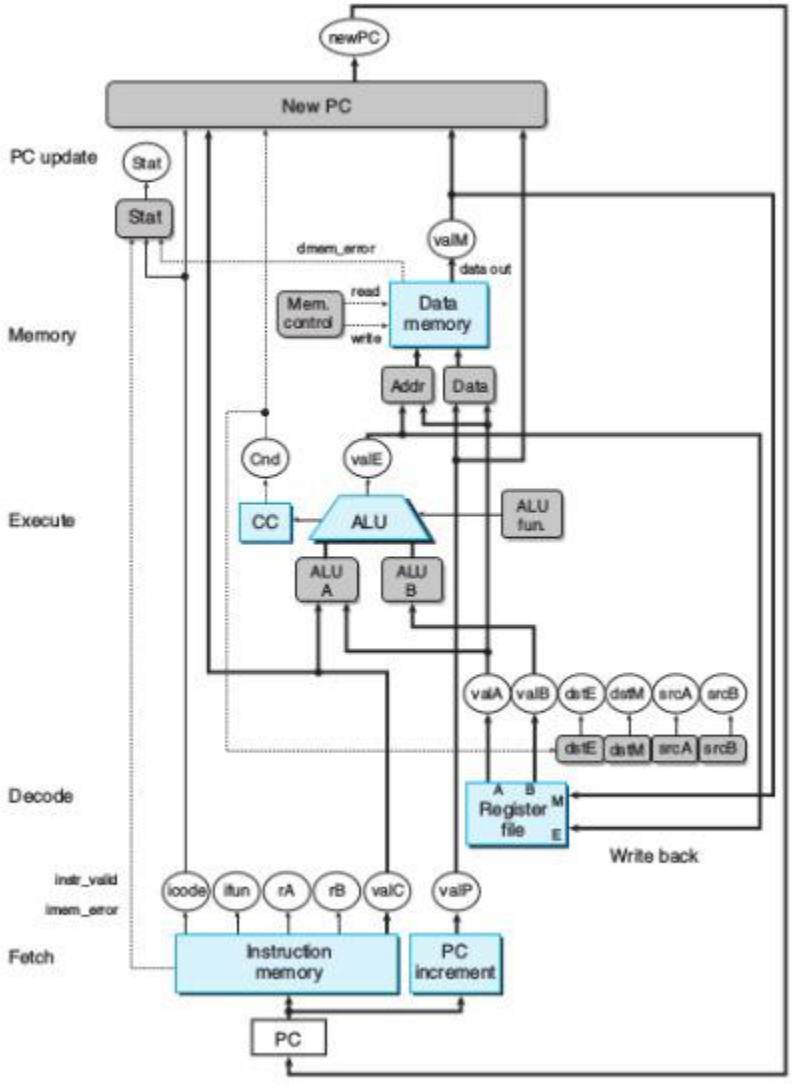
Now we proceed through the individual stages and create detailed designs for these blocks. The hardware structure given in the next figure operates in a fundamentally different way, with a single clock transition triggering a flow through combinational logic to execute an entire instruction. Let us see how the hardware can implement the behavior listed in these tables.

My implementation of SEQ consists of combinational logic (and the pipelined) and two forms of memory devices: clocked registers (the program counter and condition code register) and random access memories (the register file, the instruction memory, and the data memory).

I have assumed that reading from a random access memory operates much like combinational logic, with the output word generated based on the address input. Sequential Y86-64 Implementations memories (such as the register file), and we can mimic this effect for larger circuits using special clock circuits. Since our instruction memory is only used to read instructions, we can therefore treat this unit as if it were combinational logic.

The program counter, the condition code register, the data memory, and the register file are controlled via a single clock signal that triggers the loading of new values into the registers and the writing of values to the random access memories. The program counter is loaded with a new

instruction address every clock cycle. The program counter is loaded with a new instruction address every clock cycle. The condition code register is loaded only when an integer operation instruction is executed. The data memory is written only when an rmmovq, pushq, or call instruction is executed. The two write ports of the register file allow two program registers to be updated on every cycle, but we can use the special register ID 0xF as a port address to indicate that no write should be performed for this port.



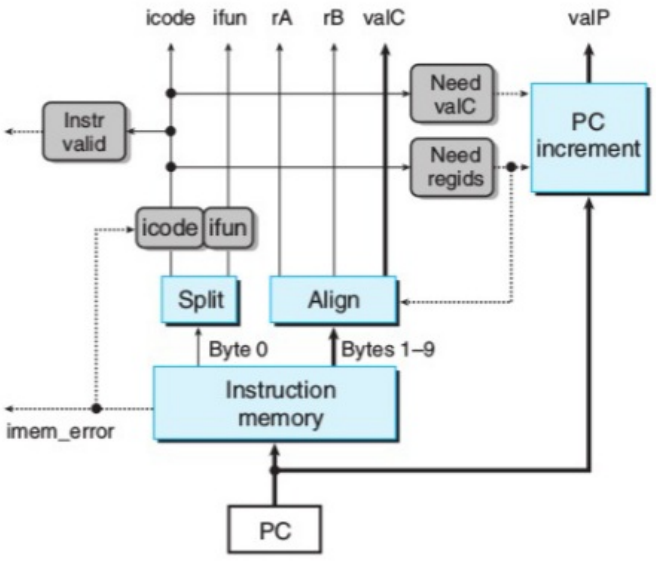
I have set the register file and the data memory to having separate connections for reading and writing, since the read operations propagate through these units as if they were combinational logic, while the write operations are controlled by the clock.

The descriptions for the control logic blocks required to implement SEQ are summarised in the below given table. These values represent the encodings of the instructions, function codes,

register IDs, ALU operations, and status codes. By convention, I have used uppercase names for constant values.

Name	Value (hex)	Meaning
IHALT	0	Code for `halt` instruction
INOP	1	Code for `nop` instruction
IRRMOVQ	2	Code for `rrmovq` instruction
IIRMOVQ	3	Code for `irmovq` instruction
IRMMOVQ	4	Code for `rmmovq` instruction
IMRMOVQ	5	Code for `mrmovq` instruction
IOPL	6	Code for integer operation instructions
IJXX	7	Code for jump instructions
ICALL	8	Code for `call` instruction
IRET	9	Code for `ret` instruction
IPUSHQ	A	Code for `pushq` instruction
IPOPQ	B	Code for `popq` instruction
FNONE	0	Default function code
RESP	4	Register ID for `%rsp`
RNONE	F	Indicates no register file access
ALUADD	0	Function for addition operation
SAOK	1	Status code for normal operation
SADR	2	Status code for address exception
SINS	3	Status code for illegal instruction exception
SHLT	4	Status code for `halt`

FETCH STAGE:




```
// The fetch stage includes the instruction memory hardware unit.
```

```
// This unit reads 10 bytes from memory at a time, using the PC as the address
of
// the first byte (byte 0).


// This byte is interpreted as the instruction byte and is split (by the module
labeled "Split") into two 4-bit quantities.
// The output of this module is two variables icode and ifun which are
respectively the instruction and function codes.
// These variables equals either the values read from memory or, in the event
that the instruction address is not valid (as indicated by the signal
imem_error),
// the values corresponding to a nop instruction. Based on the value of icode,
we can compute three 1-bit signals


module split(input [7:0] ibyte, output [3:0] icode, output [3:0] ifun);
   assign icode = ibyte[7:4];
   assign ifun = ibyte[3:0];
endmodule
```

The module split serves to split the first byte of an instruction into the instruction code and function fields. We see that this module has a single eight-bit input ibyte and two four-bit outputs icode and ifun. Output icode is defined to be the high-order four bits of ibyte, while ifun is defined to be the low-order four bits.

```
// instr_valid - Indicates if this byte corresponds to a legal Y86-64
instruction. This signal is used to detect an illegal instruction.
// need_regids - Indicates if this instruction includes a register specifier
byte.
// need_valC - Indicates if this instruction includes a constant word.
// Extract immediate word from 9 bytes of instruction
module align(input [71:0] ibytes, input need_regids, output [ 3:0]  rA, output
[ 3:0]  rB, output [63:0] valC);
   assign rA = ibytes[7:4];
   assign rB = ibytes[3:0];
   assign valC = need_regids ? ibytes[63:0] : ibytes[71:8];
endmodule
```

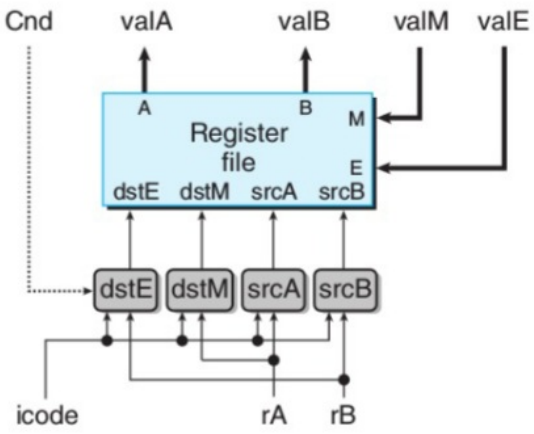
The align module describes how the processor extracts the remaining fields from an instruction, depending on whether or not the instruction has a register specifier byte. Again we see the use of continuous assignments and bit vector subranges. This module also includes a conditional expression, similar to the conditional expressions of C. In Verilog, however, this expression provides a way of creating a multiplexor—combinational logic that chooses between two data inputs based on a one-bit control signal.

```
// PC incrementer depends on whether we have set bit in need_regids and
need_valC so that we increment the value of PC by the appropriate value to
fetch the next instruction
module PC_increment(input [63:0] pc, input  need_regids, input need_valC,output
[63:0] valP);
   assign valP = pc + 1 + 8*need_valC + need_regids;
Endmodule
```

The PC incrementer hardware unit generates the signal valP, based on the current value of the PC, and the two signals need_regids and need_valC. For PC value p, need_regids value r, and need_valC value i, the incrementer generates the value p + 1 + r + 8i.

The signals instr_valid and imem_error (generated when the instruction address is out of bounds) are used to generate the status code in the memory stage.

Decode and Write-Back Stages




```
// `include "ClockedRegER.v"
// Decode stage
// The instruction fields are decoded to generate register identifiers for four addresses (two read and two write) used by
// the register file. The values read from the register file become the signals valA and valB.
// The two write-back values valE and valM serve as the data for the writes.
module regfile( input [ 3:0] dstE, input [63:0] valE, input [ 3:0] dstM, input [63:0] valM, input [ 3:0] srcA, output [63:0] valA,input [ 3:0] srcB, output [63:0] valB, input reset, input clock, output [63:0] rax, rcx, rdx, rbx, rsp, rbp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14);
    // We need to update the value in the registers contained in the register file in each cycle and update them with the required value which will be used in the next cycle.
    // output [63:0] rax, rcx, rdx, rbx, rsp, rbp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14;
    // Input data for each register
    wire [63:0] data_rax, data_rcx, data_rdx, data_rbx, data_rsp, data_rbp, data_rsi, data_rdi, data_r8, data_r9, data_r10, data_r11, data_r12, data_r13, data_r14;
    // Input write controls for each register
    wire wire_rax, wire_rcx, wire_rdx, wire_rbx, wire_rsp, wire_rbp, wire_rsi, wire_rdi, wire_r8, wire_r9, wire_r10, wire_r11, wire_r12, wire_r13, wire_r14;
    // AT the beginning of thr cycle we update the necessary values in the registers on the positive edge of the clock
    // Implement with clocked registers
```

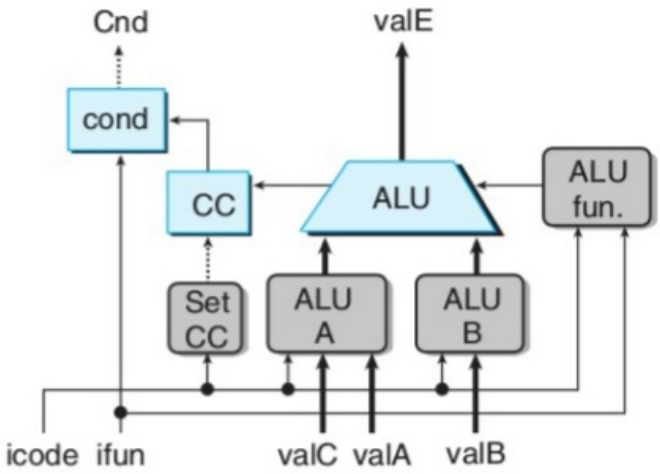
```
    clcreg64 reg_rax(rax, data_rax, wire_rax, 1'b0, 64'b0, clock);
    clcreg64 reg_rcx(rcx, data_rcx, wire_rcx, 1'b0, 64'b0, clock);
    clcreg64 reg_rdx(rdx, data_rdx, wire_rdx, 1'b0, 64'b0, clock);
    clcreg64 reg_rbx(rbx, data_rbx, wire_rbx, 1'b0, 64'b0, clock);
    clcreg64 reg_rsp(rsp, data_rsp, wire_rsp, 1'b0, 64'b0, clock);
    clcreg64 reg_rbp(rbp, data_rbp, wire_rbp, 1'b0, 64'b0, clock);
    clcreg64 reg_rsi(rsi, data_rsi, wire_rsi, 1'b0, 64'b0, clock);
    clcreg64 reg_rdi(rdi, data_rdi, wire_rdi, 1'b0, 64'b0, clock);
    clcreg64 reg_r8( r8,  data_r8,  wire_r8,  1'b0, 64'b0, clock);
    clcreg64 reg_r9( r9,  data_r9,  wire_r9,  1'b0, 64'b0, clock);
    clcreg64 reg_r10(r10, data_r10, wire_r10, 1'b0, 64'b0, clock);
    clcreg64 reg_r11(r11, data_r11, wire_r11, 1'b0, 64'b0, clock);
    clcreg64 reg_r12(r12, data_r12, wire_r12, 1'b0, 64'b0, clock);
    clcreg64 reg_r13(r13, data_r13, wire_r13, 1'b0, 64'b0, clock);
    clcreg64 reg_r14(r14, data_r14, wire_r14, 1'b0, 64'b0, clock);
//  Output the value of valA for the necessary register.
//  valA value is outputted on the basis of whether the value in srcA matches
with which register number.
    assign valA = srcA == 4'h0 ? rax : srcA == 4'h1 ? rcx : srcA == 4'h2 ? rdx
: srcA == 4'h3 ? rbx : srcA == 4'h4 ? rsp : srcA == 4'h5 ? rbp : srcA == 4'h6 ?
rsi : srcA == 4'h7 ? rdi : srcA == 4'h8 ? r8 : srcA == 4'h9 ? r9 : srcA == 4'ha
? r10 : srcA == 4'hb ? r11 : srcA == 4'hc ? r12 : srcA == 4'hd ? r13 : srcA ==
4'he ? r14 : 0;
//  Output the value of valB for the necessary register.
//  valB value is outputted on the basis of whether the value in srcB matches
with which register number.
    assign valB = srcB == 4'h0 ? rax : srcB == 4'h1 ? rcx : srcB == 4'h2 ? rdx
: srcB == 4'h3 ? rbx : srcB == 4'h4 ? rsp : srcB == 4'h5 ? rbp : srcB == 4'h6 ?
rsi : srcB == 4'h7 ? rdi : srcB == 4'h8 ? r8 : srcB == 4'h9 ? r9 : srcB == 4'ha
? r10 : srcB == 4'hb ? r11 : srcB == 4'hc ? r12 : srcB == 4'hd ? r13 : srcB ==
4'he ? r14 : 0
//  Next we update the data wire corresponding to each register based on
whether the value in valM matches whichever register number.
    assign data_rax = dstM == 4'h0 ? valM : valE;
    assign data_rcx = dstM == 4'h1 ? valM : valE;
    assign data_rdx = dstM == 4'h2 ? valM : valE;
    assign data_rbx = dstM == 4'h3 ? valM : valE;
    assign data_rsp = dstM == 4'h4 ? valM : valE;
    assign data_rbp = dstM == 4'h5 ? valM : valE;
    assign data_rsi = dstM == 4'h6 ? valM : valE;
    assign data_rdi = dstM == 4'h7 ? valM : valE;
    assign data_r8 =  dstM == 4'h8 ? valM : valE;
    assign data_r9 =  dstM == 4'h9 ? valM : valE;
    assign data_r10 = dstM == 4'ha ? valM : valE;
    assign data_r11 = dstM == 4'hb ? valM : valE;
    assign data_r12 = dstM == 4'hc ? valM : valE;
    assign data_r13 = dstM == 4'hd ? valM : valE;
    assign data_r14 = dstM == 4'he ? valM : valE;
//  Next we update the data of every register wire corresponding to each
register based on whether the value in valE matches whichever register number.
    assign wire_rax = dstM == 4'h0 | dstE == 4'h0
```

```
    assign wire_rcx = dstM == 4'h1 | dstE == 4'h1
    assign wire_rdx = dstM == 4'h2 | dstE == 4'h2
    assign wire_rbx = dstM == 4'h3 | dstE == 4'h3
    assign wire_rsp = dstM == 4'h4 | dstE == 4'h4
    assign wire_rbp = dstM == 4'h5 | dstE == 4'h5
    assign wire_rsi = dstM == 4'h6 | dstE == 4'h6
    assign wire_rdi = dstM == 4'h7 | dstE == 4'h7
    assign wire_r8 = dstM ==  4'h8 | dstE == 4'h8
    assign wire_r9 = dstM ==  4'h9 | dstE == 4'h9
    assign wire_r10 = dstM == 4'ha | dstE == 4'ha
    assign wire_r11 = dstM == 4'hb | dstE == 4'hb
    assign wire_r12 = dstM == 4'hc | dstE == 4'hc
    assign wire_r13 = dstM == 4'hd | dstE == 4'hd
    assign wire_r14 = dstM == 4'he | dstE == 4'he
endmodule
```

The instruction fields are decoded to generate register identifiers for four addresses (two read and two write) used by the register file. The values read from the register file become the signals valA and valB. The two write-back values valE and valM serve as the data for the writes.

The four blocks at the bottom of the above figure generate the four different register IDs for the register file, based on the instruction code icode, the register specifiers rA and rB, and possibly the condition signal Cnd computed in the execute stage. Register ID srcA indicates which register should be read to generate valA. Register ID dstE indicates the destination register for write port E, where the computed value valE is stored.

Execute Stage



This unit performs the operation add, subtract, and, or exclusive-or on inputs aluA and aluB based on the setting of the alufun signal. These data and control signals are generated by three control

blocks, as diagrammed in the figure above. The ALU output becomes the signal valE. In The ALU computation for each instruction is shown as the first step in the execute stage. The operands are listed with aluB first, followed by aluA to make sure the subq instruction subtracts valA from valB. We can see that the value of aluA can be valA, valC, or either −8 or +8, depending on the instruction type.

```
// 0- ADD, 1- SUB, 2- AND, 3- exclusive-or
module alu(input [63:0] aluA, input [63:0] aluB, input [3:0] alufun, output
[63:0] valE, output [2:0] new_cc);
   assign valE = alufun == 4'h0 ? aluB + aluA : alufun == 4'h1 ? aluB - aluA :
alufun == 4'h2 ? aluB & aluA : alufun == 4'h3 ? aluB^aluA : alufun;
   // new_cc[2] corresponds to zero flag new_cc[1] corresponds to ;
   assign new_cc[2] = (valE == 0);
   assign new_cc[1] = valE[63];
   assign new_cc[0] = alufun == 4'h0 ? (aluA[63] == aluB[63]) & (aluA[63] !=
valE[63]) : alufun == 4'h1 ? (~alu[63] == aluB[63] & (aluB[63] != valE[63])):
0;
Endmodule
```

The execute stage also includes the condition code register. Our ALU generates the three signals on which the condition codes are based—zero, sign, and overflow—every time it operates. However, we only want to set the condition codes when an OPq instruction is executed. We therefore generate a signal set_cc that controls whether or not the condition code register should be updated.

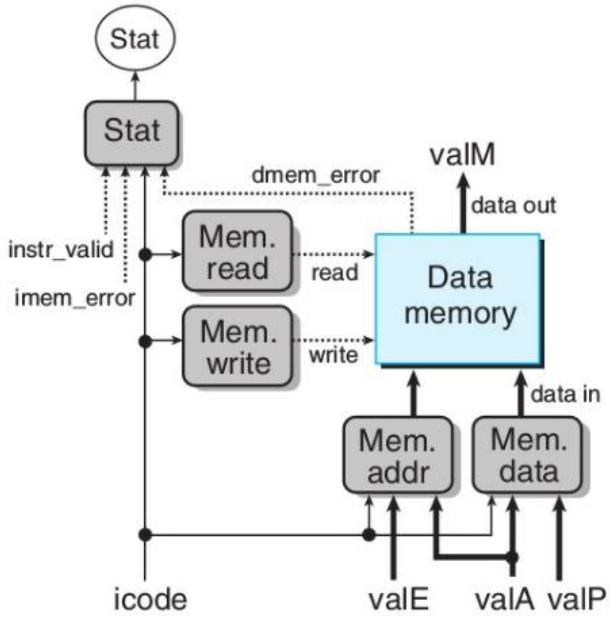
```
module cc(output [2:0] cc, input [2:0] new_cc, input set_cc, input reset, input
clock);
   pipelinedreg3 c(cc, new_cc, ~set_cc, 3'b100, clock);
Endmodule
```

“Cond” unit uses a combination of the condition codes and the function code to determine whether a conditional branch or data transfer should take place (Figure 4.3). It generates the Cnd signal used both for the setting of dstE with conditional moves and in the next PC logic for conditional branches. For other instructions, the Cnd signal may be set to either 1 or 0, depending on the instruction’s function code and the setting of the condition codes, but it will be ignored by the control logic.

```
module cond(input [3:0] ifun, input [2:0] cc, output Cnd);
   wire zf = cc[2];
   wire sf = cc[1];
   wire of = cc[0];
   assign Cnd = (ifun == 4'h0) | (ifun ==  4'h1 & ((sf^of)|zf)) | (ifun == 4'h2
& (sf^of)) | (ifun == 4'h3 & zf) | (ifun == 4'h4 & ~zf) | (ifun == 4'h5 &
(~sf^of)) | (ifun == 4'h6 & (~sf^of)&~zf);
```

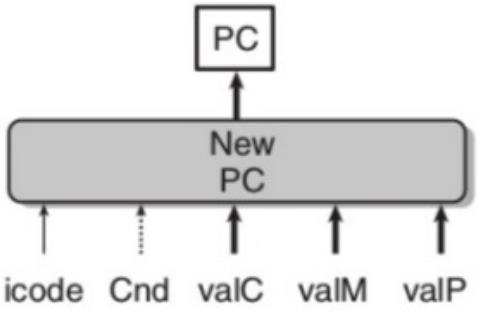
```
endmodule
```

MEMORY STAGE



The memory stage has the task of either reading or writing program data. Two control blocks generate the values for the memory address and the memory input data (for write operations). Two other blocks generate the control signals indicating whether to perform a read or a write operation. When a read operation is performed, the data memory generates the value valM.

SEQ PC update stage



The final stage in SEQ generates the new value of the program counter. As the final steps in given code shows how the new PC will be valC, valM, or valP, depending on the instruction type and whether or not a branch should be taken.

```
module PC_increment(input [63:0] pc, input  need_regids, input need_valC,output
[63:0] valP);
   assign valP = pc + 1 + 8*need_valC + need_regids;
Endmodule
```

Hardware registers in the processor have been designed using clocked registers with enable and reset.

Clocked register (short for “conditionally-enabled, resettable register”) that I have used as a building block for the hardware registers in our processor. The idea is to have a register that can

be loaded with the value on its input in response to a clock. Additionally, it is possible to reset the register, causing it to be set to a fixed constant value. We see that the register data output out is declared to be of type reg (short for “register”). That means that it will hold its value until it is explicitly updated. This contrasts to the signals of type wire that are used to implement combinational logic. The statement beginning always @(posedge clock) describes a set of actions that will be triggered every time the clock signal goes for 0 to 1 (this is considered to be the positive edge of a clock signal.) Within this statement, we see that the output may be updated to be either its input or its reset value. The assignment operator <= is known as a non-blocking assignment. That means that the actual updating of the output will only take place when a new event is triggered, in this case the transition of the clock from 0 to 1. We can see that the output may be updated as the clock rises. Observe, however, that if neither the reset nor the enable signals are 1, then the output will remain at its current value.

Code for clocked register with 64 bits. Similarly 4,3,1, bit register values have also been used.

```
module clcreg64(output [63:0] out, input [63:0] in, input enable, input reset,
input [63:0] value, input clock);
   reg [63:0] out;
   always
   // Flush the required value in the out reg depending on the reset and enable
on every positive edge of clock.
       @(posedge clock)
       begin
       // Reset if the reselt value is 1 i.e., out=value else if enable is 1
in=out
       if (reset)
           out <= value;
       else if (enable)
           out <= in;
       end
endmodule
```

PIPELINED PROCESSOR

All the outputs summarised in this report are the output of the fully functioning pipelined structure (optimised over sequential register by adding the pipelined registers.).

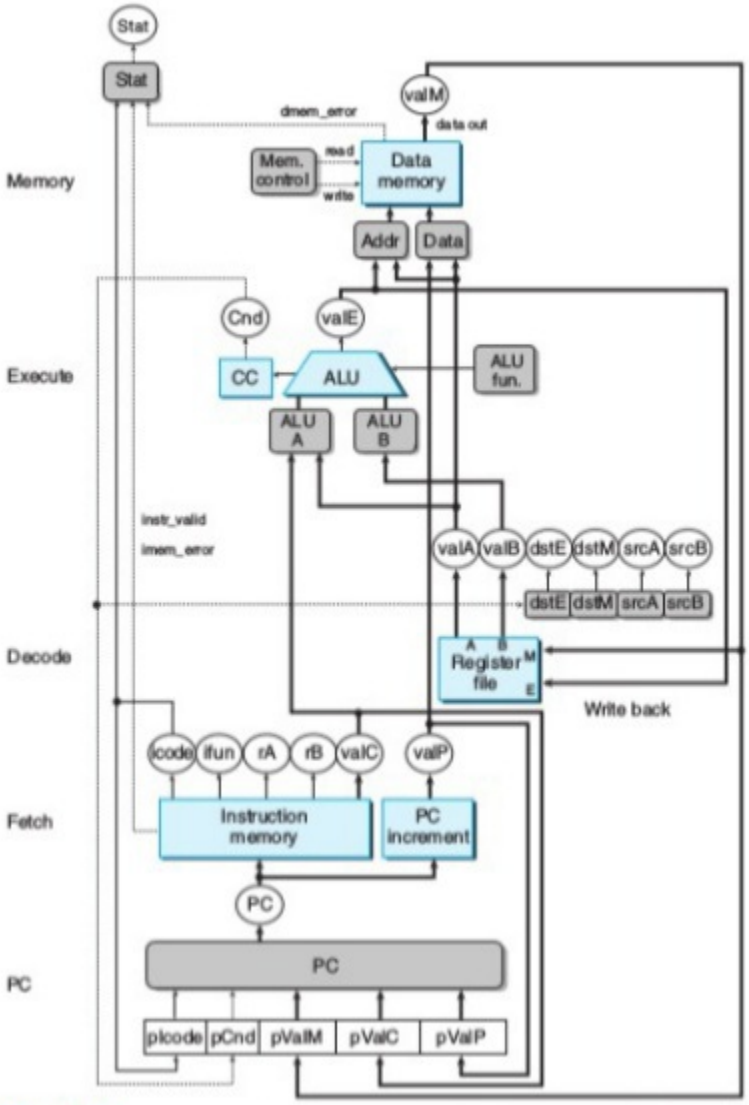


Figure 4.40 **SEQ+ hardware structure.** Shifting the PC computation from the end of the clock cycle to the beginning makes it more suitable for pipelining.

The pipeline registers separates the stages.

The pipeline registers are labeled as follows:

- F holds a predicted value of the program counter, as will be discussed shortly.
- D sits between the fetch and decode stages. It holds information about the most recently fetched instruction for processing by the decode stage.
- E sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.

- M sits between the execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage. It also holds information about branch conditions and branch targets for processing conditional jumps.
- W sits between the memory stage and the feedback paths that supply the computed results to the register file for writing and the return address to the PC selection logic when completing a ret instruction

Rearranging and Relabeling Signals:

Our sequential implementations SEQ only process one instruction at a time, and so there are unique values for signals such as valC, srcA, and valE. In our pipelined design, there will be multiple versions of these values associated with the different instructions flowing through the system. I have adopted a naming scheme where a signal stored in a pipeline register can be uniquely identified by prefixing its name with that of the pipe register written in uppercase. For example, the four status codes are named D_stat, E_stat, M_stat, and W_stat. I have also refered to some signals that have just been computed within a stage. These are labeled by prefixing the signal name with the first character of the stage name, written in lowercase.

The following module preg shows how we can use our basic register to construct a pipeline register:

```
// Bubble is injected using the indicator as reset signal. Also we must specify
the value that will be loaded
module pipelinedreg64(output [63:0] out, input [63:0]  in, input stall, input
[63:0] bubbleval, input clock);
   reg [63:0] out;
   initial begin
    out <= bubbleval;
end
always @(posedge clock) begin
   if (!stall)
       out <= in;
   end
endmodule
```

The register file is implemented using 15 clocked registers for the 15 program registers. Combinational logic is used to select which program register values are routed to the register file outputs, and which program registers to update by a write operation.

OVERALL PROCESSOR DESIGN:

IMPLEMENTATION HIGHLIGHTS:

The following are samples of the Verilog code for our implementation of PIPE, showing the implementation of the fetch stage. The following are declarations of the internal signals of the fetch stage. They are all of type wire, meaning that they are simply connectors from one logic block to another.

```
wire [63:0] f_predPC, F_predPC, f_pc;
   wire f_ok;
   wire imem_error;
   wire [ 2:0] f_stat;
   wire [7:0] f_ibyte;
   wire[71:0] f_ibytes;
   wire [ 3:0] f_icode;
   wire [ 3:0] f_ifun;
   wire [ 3:0] f_rA;
   wire [ 3:0] f_rB;
   wire [63:0] f_valC;
   wire [63:0] f_valP;
   wire need_regids;
   wire need_valC;
   wire instr_valid;
   wire F_stall, F_bubble, F_reset;
```

The following signals must be included to allow pipeline registers F and D to be reset when either the processor is in RESET mode or the bubble signal is set for the pipeline register.

```
wire resetting;
wire D_reset = 1;
wire F_reset = 1;
assign resetting = 1;
assign D_reset = 1;
```

The different elements of pipeline registers F and D are generated as instantiations of the preg register module.

```
pipelinedreg64 F_predPC_reg(F_predPC, f_predPC, F_stall, 64'b0, clock);
   // D Register
   pipelinedreg3 D_stat_reg(D_stat, f_stat, D_stall,   SBUB, clock);
   pipelinedreg64 D_pc_reg(D_pc, f_pc, D_stall,   64'b0, clock);
   pipelinedreg4 D_icode_reg(D_icode, f_icode, D_stall,   INOP, clock);

   pipelinedreg4 D_ifun_reg(D_ifun, f_ifun, D_stall, FNONE, clock);
   pipelinedreg4 D_rA_reg(D_rA, f_rA, D_stall, RNONE, clock);
   pipelinedreg4 D_rB_reg(D_rB, f_rB, D_stall, RNONE, clock);
   pipelinedreg64 D_valC_reg(D_valC, f_valC, D_stall, 64'b0, clock);
   pipelinedreg64 D_valP_reg(D_valP, f_valP, D_stall, 64'b0, clock);
   // E Register
   pipelinedreg3 E_stat_reg(E_stat, D_stat, E_stall,  SBUB, clock);
```

```
   pipelinedreg64 E_pc_reg(E_pc, D_pc, E_stall,  64'b0, clock);
   pipelinedreg4 E_icode_reg(E_icode, D_icode, E_stall,  INOP, clock);
   pipelinedreg4 E_ifun_reg(E_ifun, D_ifun, E_stall,  FNONE, clock);
   pipelinedreg64 E_valC_reg(E_valC, D_valC, E_stall,  64'b0, clock);
   pipelinedreg64 E_valA_reg(E_valA, d_valA, E_stall,  64'b0, clock);

   pipelinedreg64 E_valB_reg(E_valB, d_valB, E_stall,  64'b0, clock);
   pipelinedreg4 E_dstE_reg(E_dstE, d_dstE, E_stall,  RNONE, clock);
   pipelinedreg4 E_dstM_reg(E_dstM, d_dstM, E_stall,  RNONE, clock);
   pipelinedreg4 E_srcA_reg(E_srcA, d_srcA, E_stall,  RNONE, clock);
   pipelinedreg4 E_srcB_reg(E_srcB, d_srcB, E_stall,  RNONE, clock);
   // M Register
   pipelinedreg3 M_stat_reg(M_stat, E_stat, M_stall,  SBUB, clock);
   pipelinedreg64 M_pc_reg(M_pc, E_pc, M_stall,  64'b0, clock);
   pipelinedreg4 M_icode_reg(M_icode, E_icode, M_stall,  INOP, clock);
   pipelinedreg4 M_ifun_reg(M_ifun, E_ifun, M_stall,  FNONE, clock);
   pipelinedreg1 M_Cnd_reg(M_Cnd, e_Cnd, M_stall,  1'b0, clock);
   pipelinedreg64 M_valE_reg(M_valE, e_valE, M_stall,  64'b0, clock);
   pipelinedreg64 M_valA_reg(M_valA, e_valA, M_stall,  64'b0, clock);
   pipelinedreg4 M_dstE_reg(M_dstE, e_dstE, M_stall,  RNONE, clock);
   pipelinedreg4 M_dstM_reg(M_dstM, E_dstM, M_stall,  RNONE, clock);
   // W Register
   pipelinedreg3 status_reg(status, m_stat, W_stall,  SBUB, clock);
   pipelinedreg64 W_pc_reg(W_pc, M_pc, W_stall,  64'b0, clock);
   pipelinedreg4 W_icode_reg(W_icode, M_icode, W_stall,  INOP, clock);
   pipelinedreg64 W_valE_reg(W_valE, M_valE, W_stall,  64'b0, clock);
   pipelinedreg64 W_valM_reg(W_valM, m_valM, W_stall,  64'b0, clock);
   pipelinedreg4 W_dstE_reg(W_dstE, M_dstE, W_stall,  RNONE, clock);
   pipelinedreg4 W_dstM_reg(W_dstM, M_dstM, W_stall,  RNONE, clock);
```

The code generators in the fetch stage can be found as :

```
assign imem_error = 1'b0;
   assign dmem_error = 1'b0;
   assign f_pc =(((M_icode == IJXX) & ~M_Cnd) ? M_valA : (W_icode == IRET) ?
W_valM :F_predPC);
   assign instr_valid =
   f_icode == INOP ? 1'b1 :
   f_icode == IHALT ? 1'b1 :
   f_icode == IRRMOVQ ? 1'b1 :
   f_icode == IIRMOVQ ? 1'b1 :
   f_icode == IRMMOVQ ? 1'b1 :
   f_icode == IMRMOVQ ? 1'b1 :
   f_icode == IOPQ ? 1'b1 :
   f_icode == IJXX ? 1'b1 :
   f_icode == ICALL ? 1'b1 :
   f_icode == IRET ? 1'b1 :
   f_icode == IPUSHQ ? 1'b1 :
```

```
    f_icode == IPOPQ ? 1'b1 : 1'b0;

    assign f_stat =(imem_error ? SADR : ~instr_valid ? SINS : (f_icode == IHALT)
? SHLT :SAOK);
    assign need_regids =
    f_icode == IRRMOVQ ? 1'b1:
    f_icode == IOPQ ? 1'b1:
    f_icode == IPUSHQ ? 1'b1:
    f_icode == IPOPQ ? 1'b1:
    f_icode == IIRMOVQ ? 1'b1:
    f_icode == IRMMOVQ ? 1'b1:
    f_icode == IMRMOVQ ? 1'b1: 1'b0;
```

Similarly it is done for other stages.

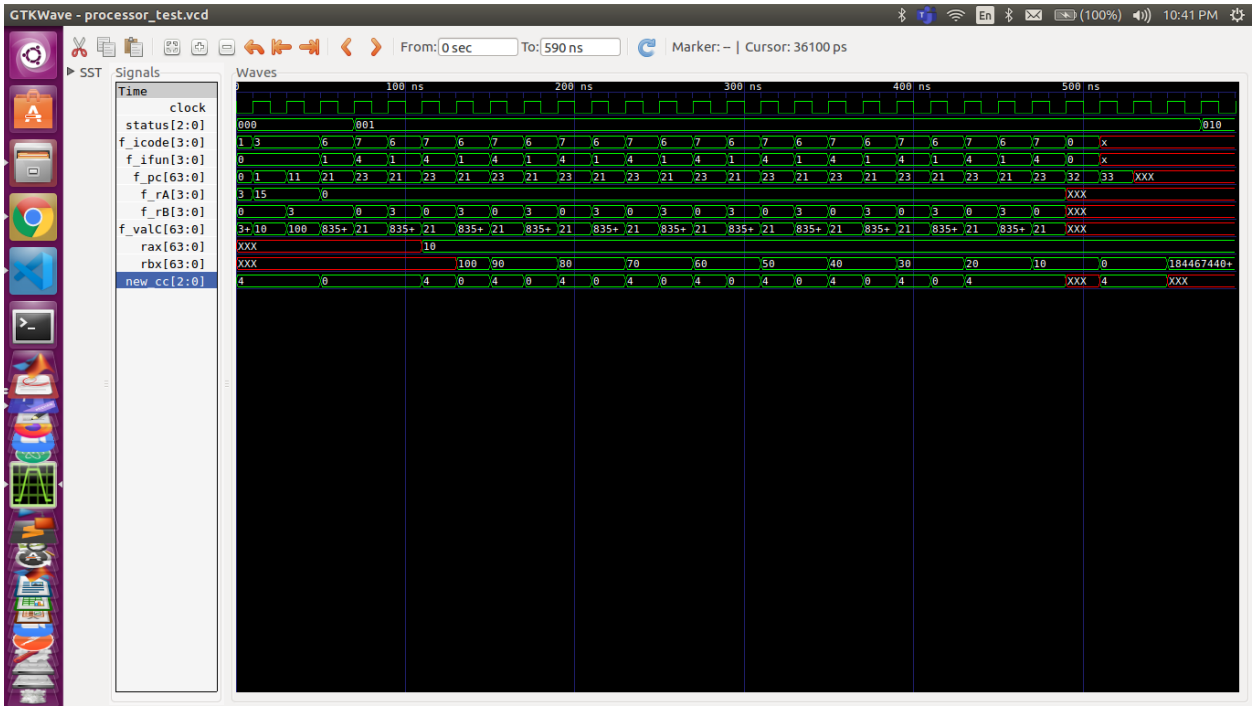
Processor features:

- Word Size = 8
- Word count = 8192
- Instruction memory count = 2048
- Clock time period = 20ns

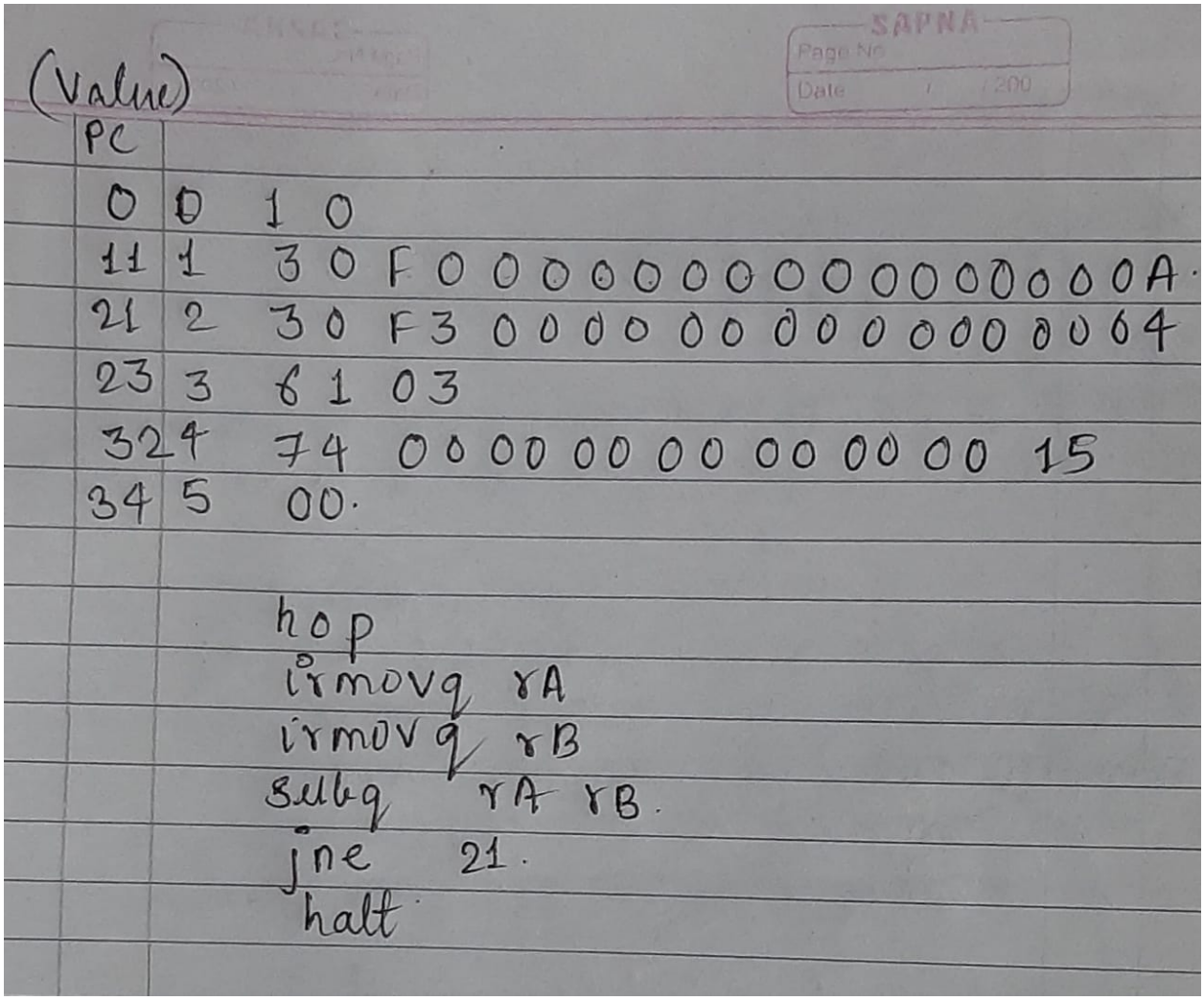
  TOTAL MEMORY SIZE = 81920 Bits = 10240 Bytes = 10.24 kilo byte

CHECKING THE CODE FOR DIFFERENT TEST CASES:

1.



These instructions verify the working of jump and few other statements.

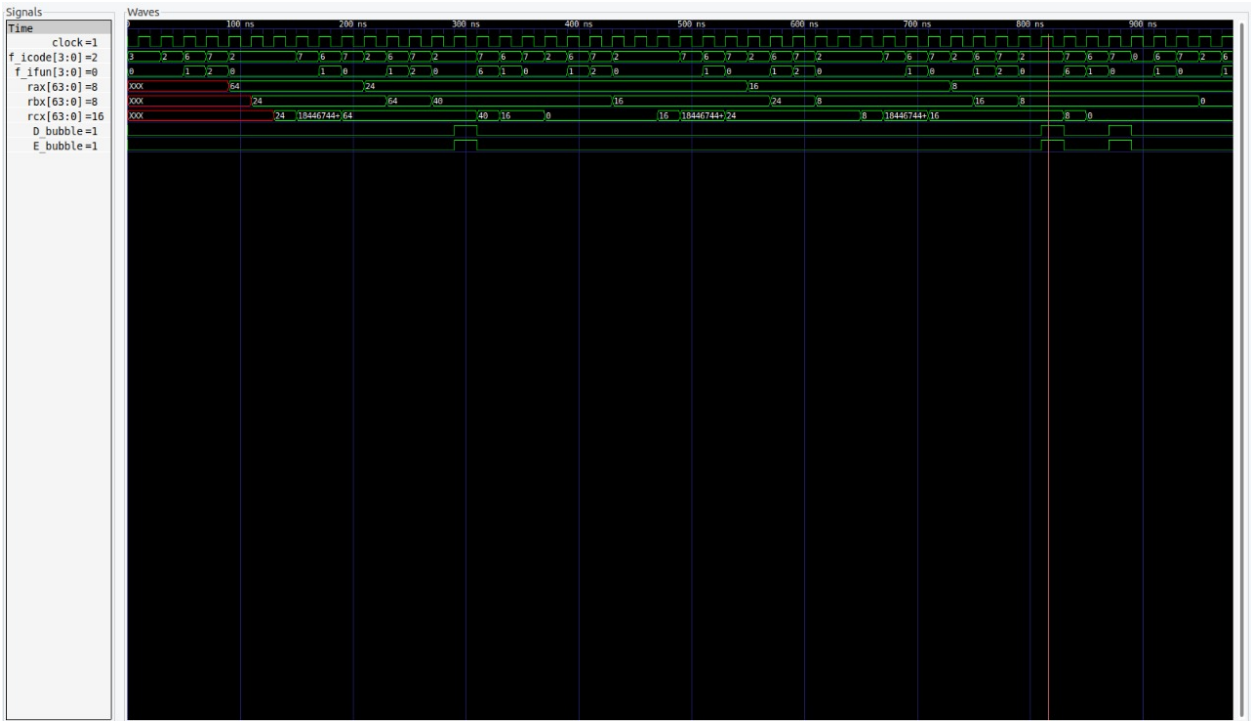


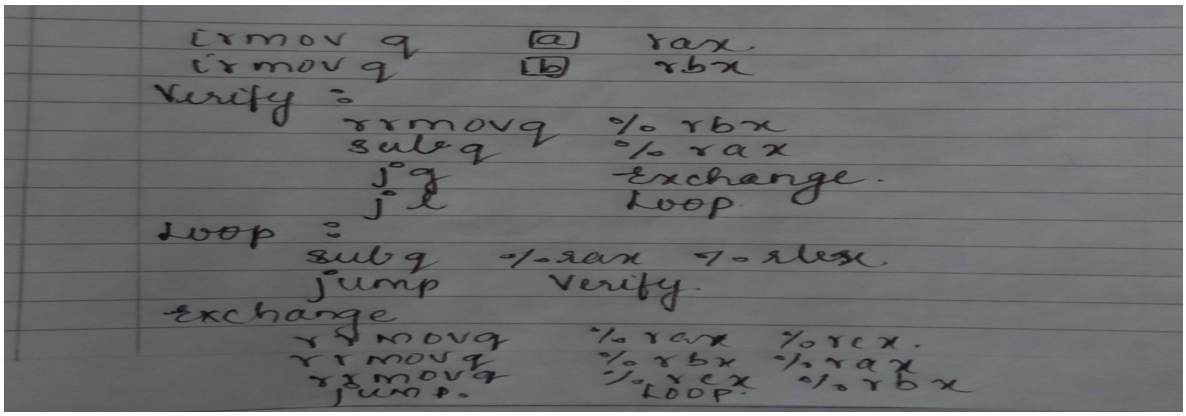
2. GCD function:

a = 24, b = 64

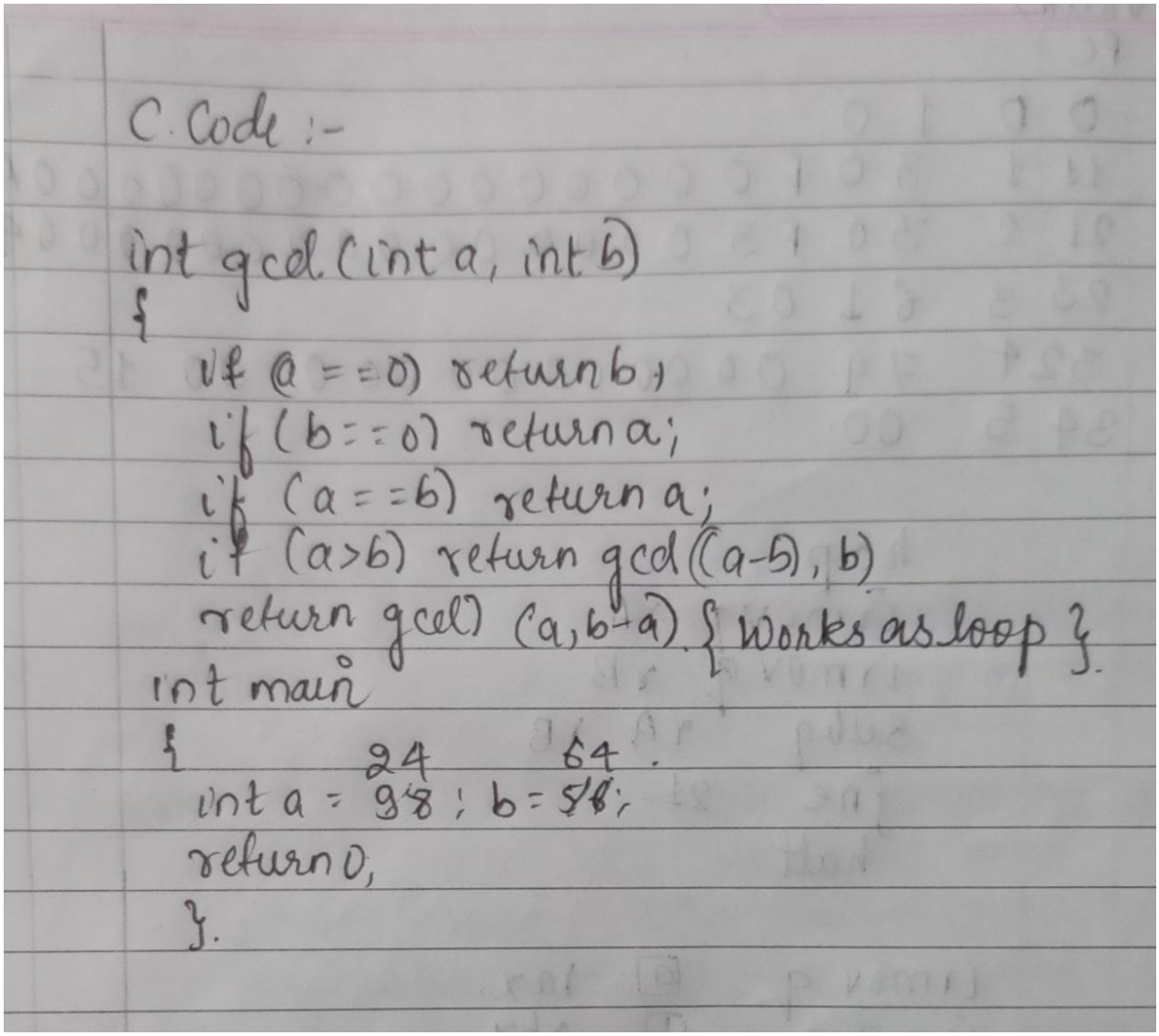
Final output in register rbx

Therefore GCD of the given two numbers is 8.





Instructions: As given in file instr1.mem. Changing the value of rA and rB to check for different input values.



3. Data hazard resistance can be checked by changing the instruction file to instr.mem.

CHALLENGES ENCOUNTERED:

Challenges encountered were to work with pipelined registers and implementing the jump instruction properly. And also implementing the call and ret instruction was pretty difficult.