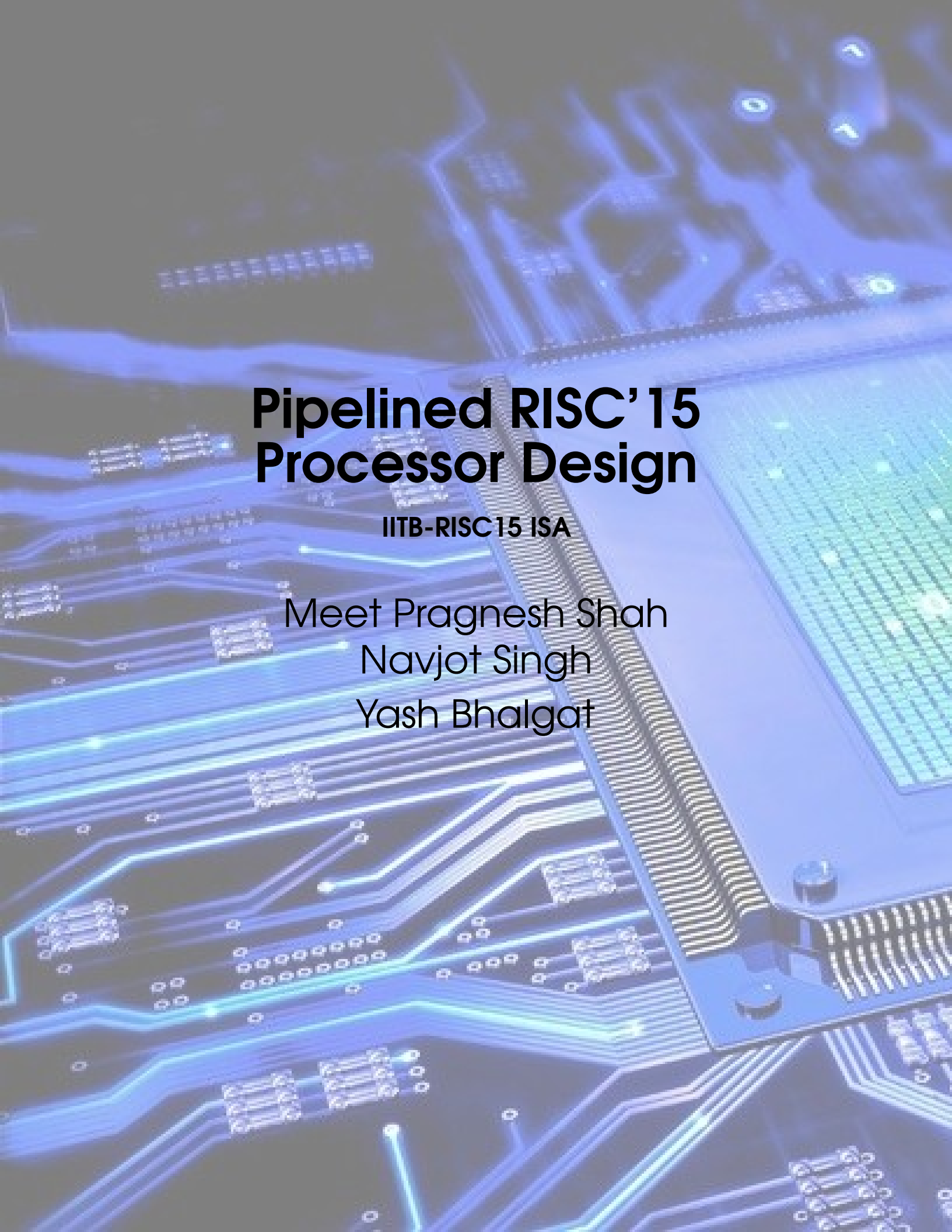


Pipelined RISC'15 Processor Design

IITB-RISC15 ISA

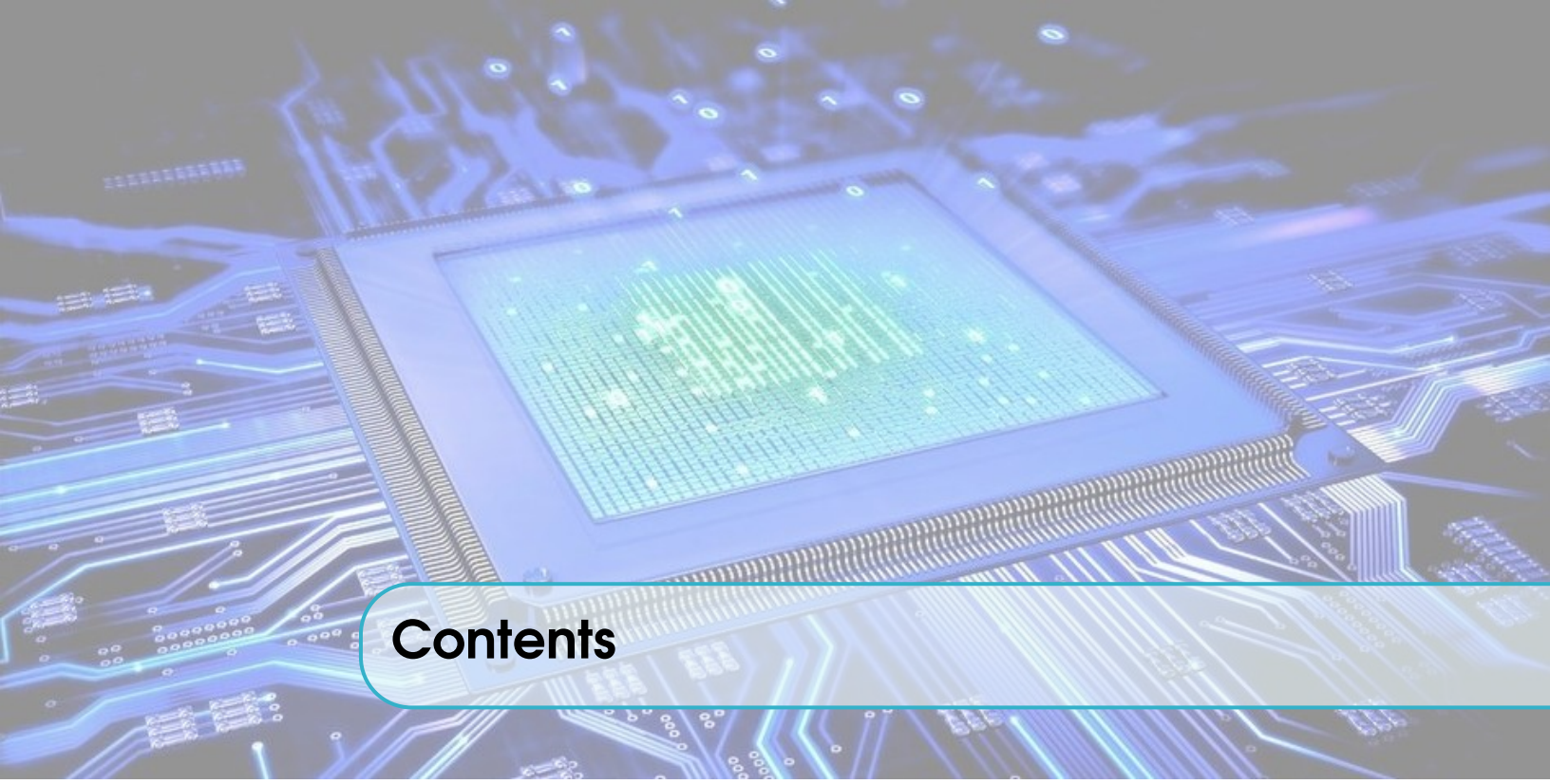
Meet Pragnesh Shah
Navjot Singh
Yash Bhalgat

EE309 (MICROPROCESSORS) / EE337 (MICROPROCESSORS LABORATORY)

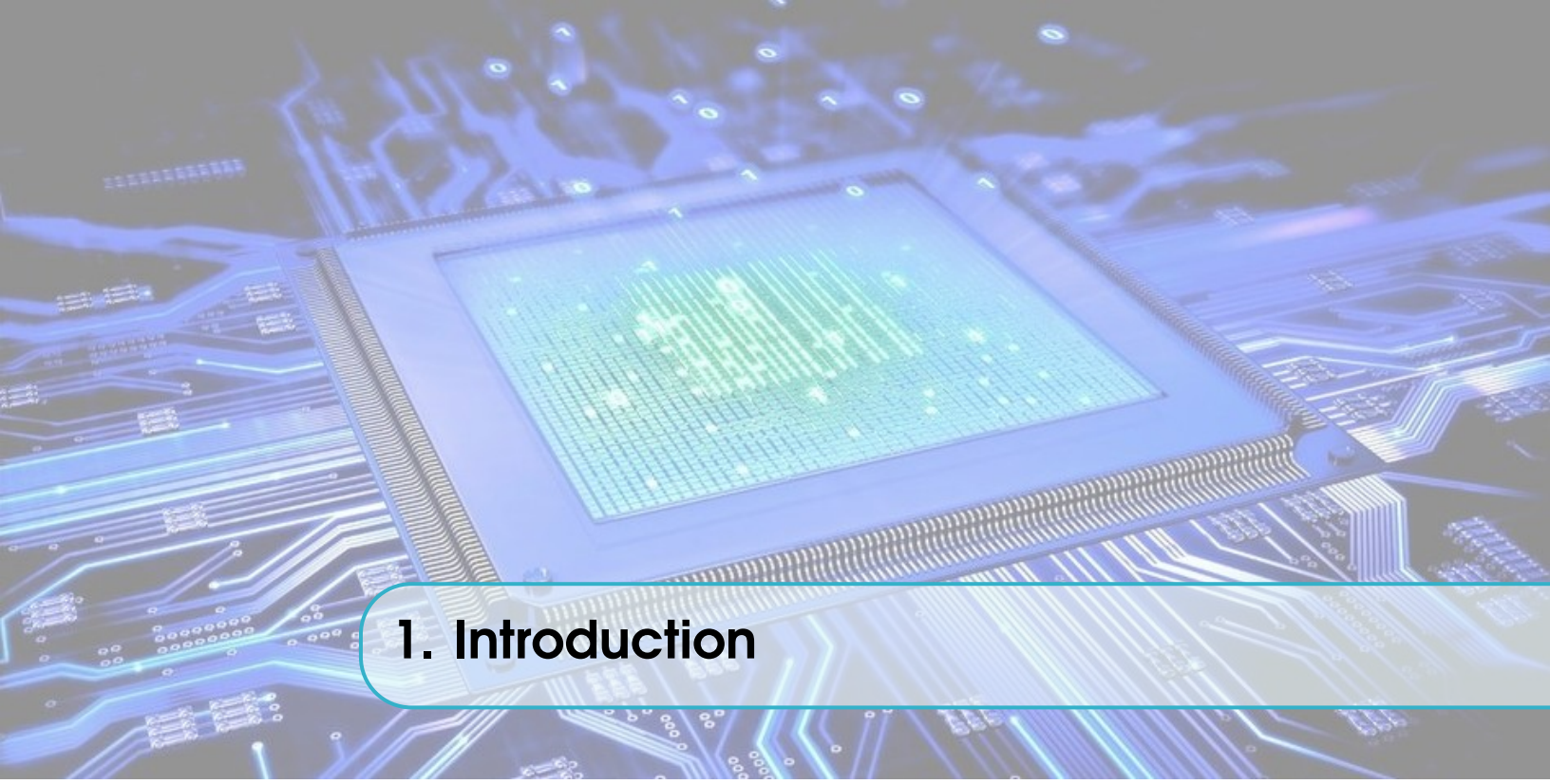
INDIAN INSTITUTE OF TECHNOLOGY, BOMBAY

https://github.com/meetshah1995/Pipelined-RISC15-Design

This project was done under the supervision of Prof. Virendra Singh, instructor for the course along with the lab course instructors Prof. Shankar Balachandran and Prof. Rajbabu Velmurugan and our TA Mrs. Shoba Gopalakrishnan



Contents



1. Introduction

1.1 Prologue

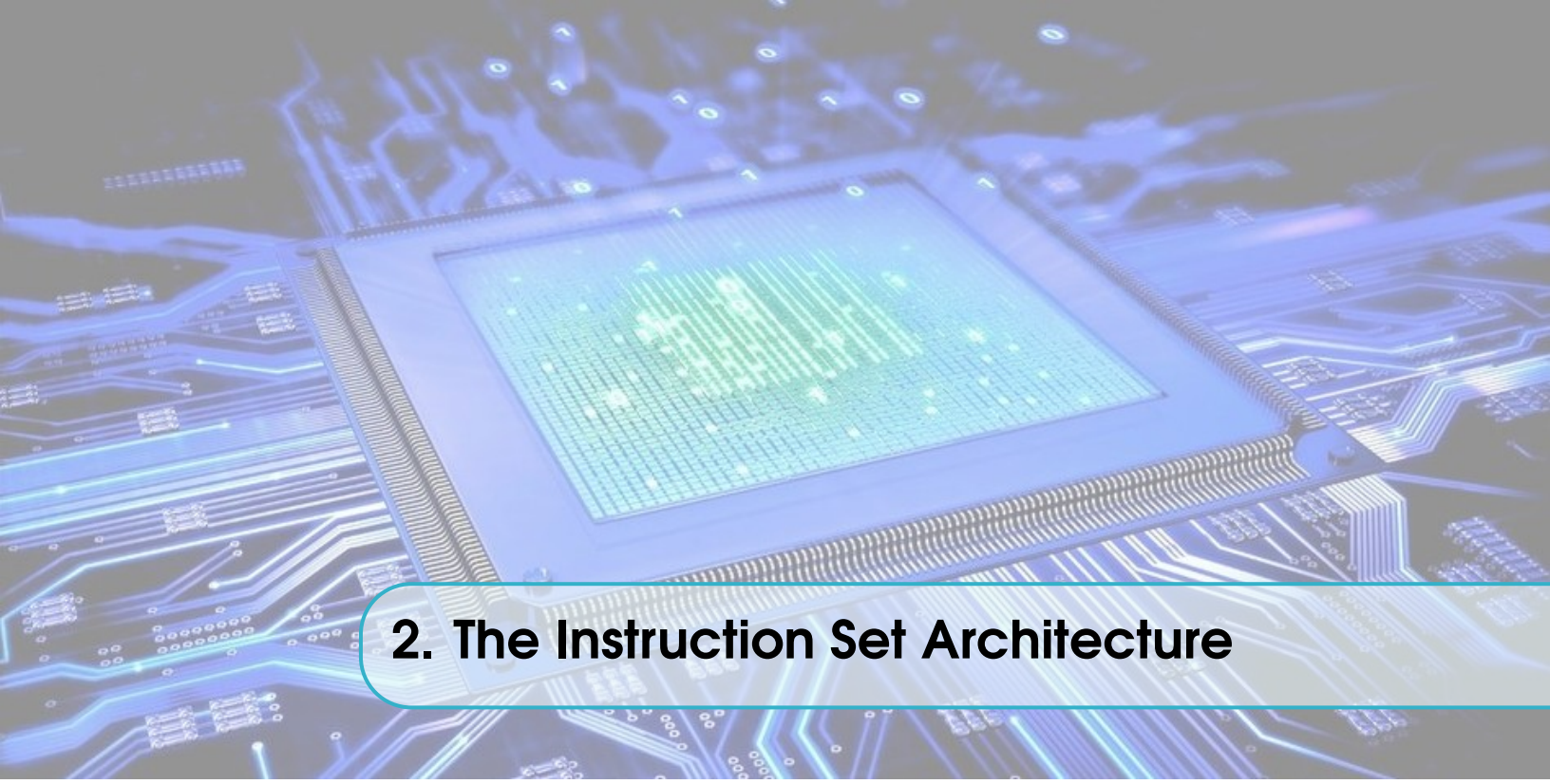
The IITB-RISC'15 processor is based on Little Computer Architecture. It has a 16 bit architecture, having 8 registers (R0 to R7) and uses point to point connections along with a condition code register with flags CY (Carry) and Z (Zero). The IITB-RISC'15 is very simple but, it is general enough to solve complex problems. The architecture allows predicated instruction execution and multiple load and store execution. There are 3 types of instruction formats (namely R, I and J) and a total of 15 instructions. They would be presented in a later chapter.

1.2 Abstract

The Processor can be divided into 3 components : (i) Datapath , (ii) Contoller , (iii) Memory .The design is based on multicycle RISC architecture. This is done because the different instructions take different execution times, and with a multicycle implementation, each instruction gets over in lesser time and hence, the performance is optimized. This also reduces the resources which would be required during simulation on FPGA. It is restricted to have the value of PC (program counter) to always be stored in R7.

1.3 Software & Packages Used

- Altera Quartus v.14.1
- ModelSim
- Sublime Text Editor [Special Thanks]
- Coffee ;)



2. The Instruction Set Architecture

2.1 IITB-RISC'15 ISA

The architecture allows predicated instruction execution and multiple load and store execution. There are 3 types of instruction formats (namely R, I and J) and a total of 15 instructions. The datapath evolves accordingly as these instruction execution flow is designed, using different MUX control signals and read/write enable signals.

R Type Instruction format

Opcode (4 bit)	Register A (RA) (3 bit)	Register B (RB) (3-bit)	Register B (RB) (3-bit)	Unused (1 bit)	Condition (CZ) (2 bit)

I Type Instruction format

Opcode (4 bit)	Register A (RA) (3 bit)	Register C (RC) (3-bit)	Immediate (6 bits signed)

J Type Instruction format

Opcode (4 bit)	Register A (RA) (3 bit)	Immediate (9 bits signed)

Figure 2.1: The instruction set

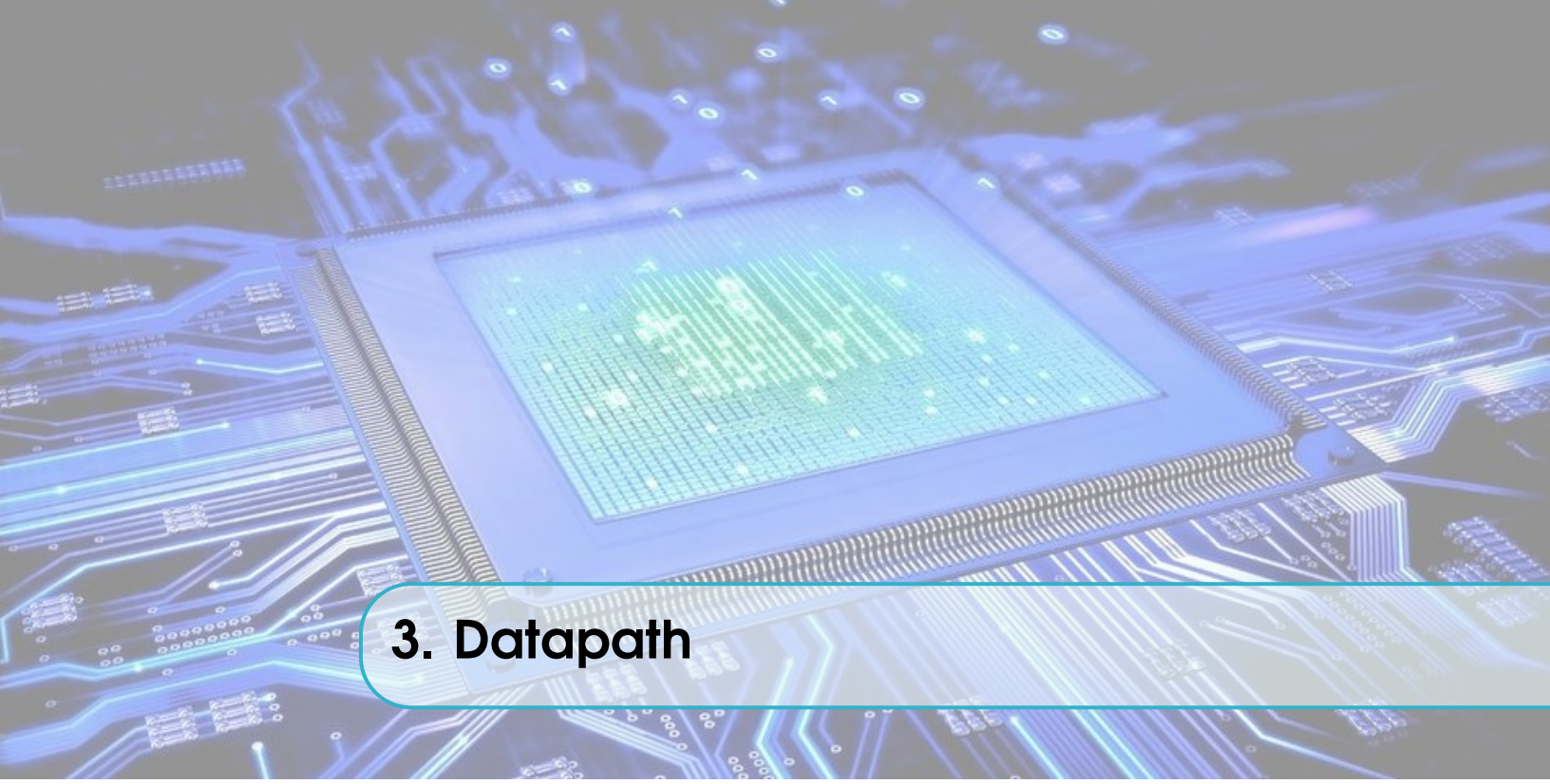
ADD:	00_00	RA	RB	RC	0	00
ADC:	00_00	RA	RB	RC	0	10
ADZ:	00_00	RA	RB	RC	0	01
ADI:	00_01	RA	RB	6 bit Immediate		
NDU:	00_10	RA	RB	RC	0	00
NDC:	00_10	RA	RB	RC	0	10
NDZ:	00_10	RA	RB	RC	0	01
LHI:	00_11	RA	9 bit Immediate			
LW:	01_00	RA	RB	6 bit Immediate		
SW:	01_01	RA	RB	6 bit Immediate		
LM:	01_10	RA	0 + 8 bits corresponding to Reg R7 to R0			
SM:	01_11	RA	0 + 8 bits corresponding to Reg R7 to R0			
BEQ:	11_00	RA	RB	6 bit Immediate		
JAL:	10_00	RA	9 bit Immediate offset			
JLR:	10_01	RA	RB	000_000		

Figure 2.2: 16-bit instruction encoding

RA: Register A

RB: Register B

RC: Register C



3. Datapath

3.1 Elements

The datapath consists of the described units:

(1) Register File : The module consists of eight 16-bit clocked registers (R0 to R7). This module has 4 inputs : 3 input addresses (add1, add2, add3) 8bit each and 1 data input (16bit). There are 2 output data sequences, corresponding to the first two input addresses, add1 and add2. Control signal Rf_wen is involved to enable the writing to the registers.

(2) ALU : The logic unit supports two operations : ADD and NAND, to implements different instructions and for incrementing desired variables. The module is combinational in nature, meaning the desired data output would be obtained in the same cycle when the two operands are fed. Two flags CY (carry) and Z (zero) are associated with the ALU, which would be written to the CCR module defined in the datapath. A clocked register, T1, stores the computed ALU result.

(3) Multiplexers : For proper instructor execution, different modules need to be enabled or disabled, along with proper routing of data to these modules. This is achieved using MUXes, having variable no. of control signals bits. These would be described later.

(4) CCR module : This module is used to generate a conditional write_enable signal to the Register File module based on previous CY and Z bits. Thus, this conditional signal would be required to implement ADZ, ADC, NDZ and NDC instruction.

(5) Registers : Temporary registers A and B are used to store output results of the register file. Register T1 is similarly used to store the ALU result.

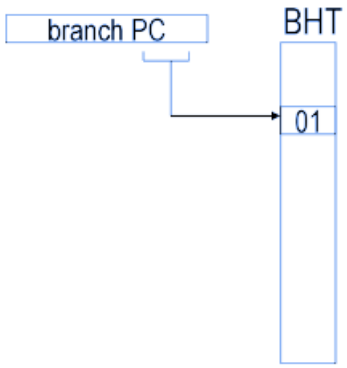
(6) Memory* : This module is connected to different modules in the datapath. It has a size of 64bytes, controlled by write_enable signal, with multiplexed input data.

(7) Priority Encoder : This module serves an essential purpose in reducing the cycles in LM and SM instructions by rendering the RF addresses corresponding to the set bits in instructions last 8 bits.

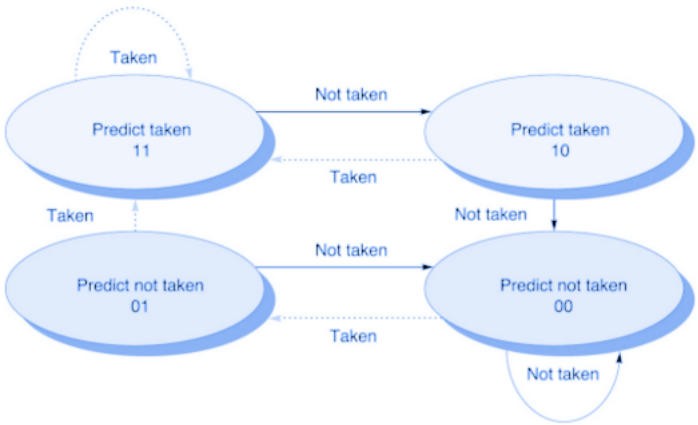
(8) Sign Extenders : This module appends zeroes at the beginning of the immediate bits,
Usage: extending Imm6 and Imm9 in immediate addressing

(9) Data Shifters : This module is used to convert a 9 bit value to 16 bit by padding zeroes at the end.

(9) Branch Predictor : Since the exact specifications for the branch predictors weren't given to us, we are just making a two-bit correlated branch predictor with a standard size BHT (Branch History Table) where we store the addresses and their branching history.



Our memory size is **64 Bytes** and ISA has 3 branching instructions out of 16, so it is safe to assume that we can take the Global Branch History Table to be of size **8 bytes** (1/8th instruction is a branch). so our branch predictor is a **(8,2)** Branch Predictor.



This blocks for GBHT and BHT can be seen in the datapath with appropriate connections to the PC and the branching.

3.2 Datapath

High Resolution SVG version of the datapath diagram can be found at :
`https://www.ee.iitb.ac.in/student/~meetshah1995/files/datapath_pipeline.svg`

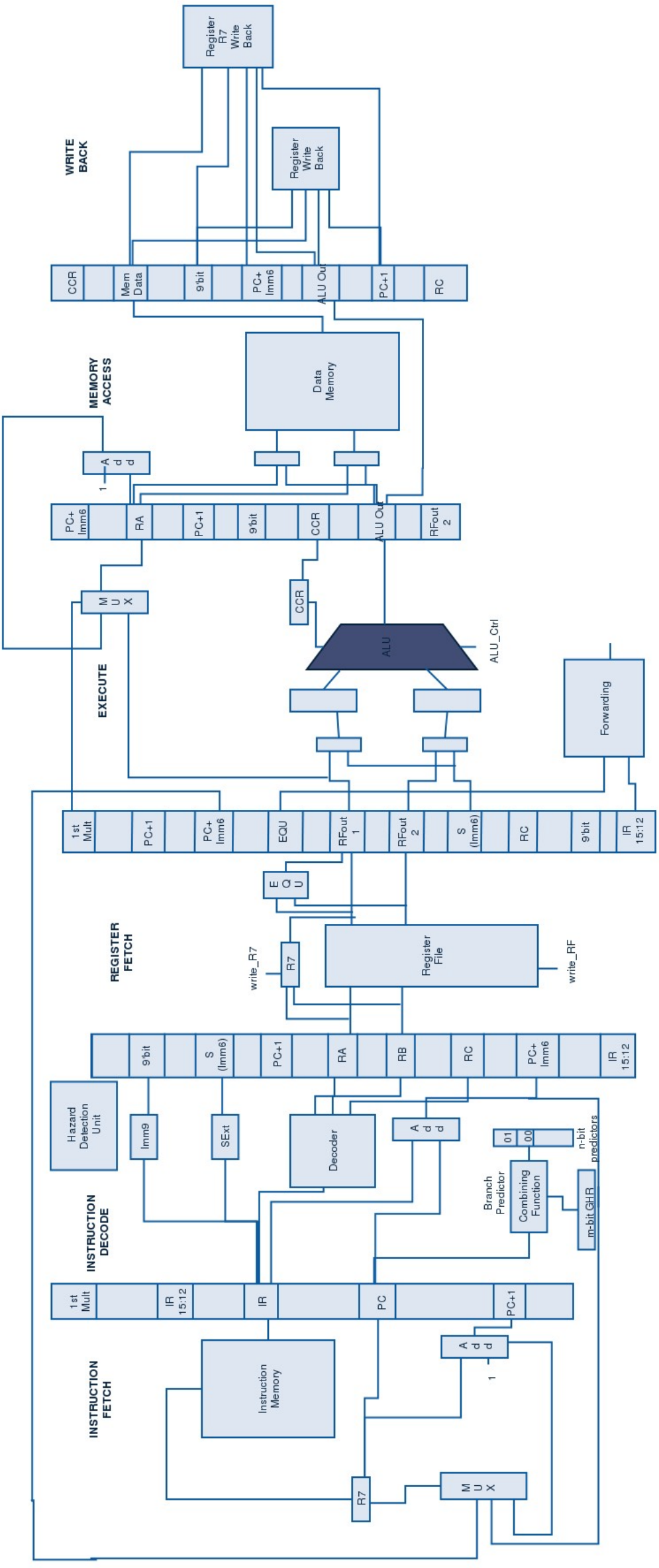
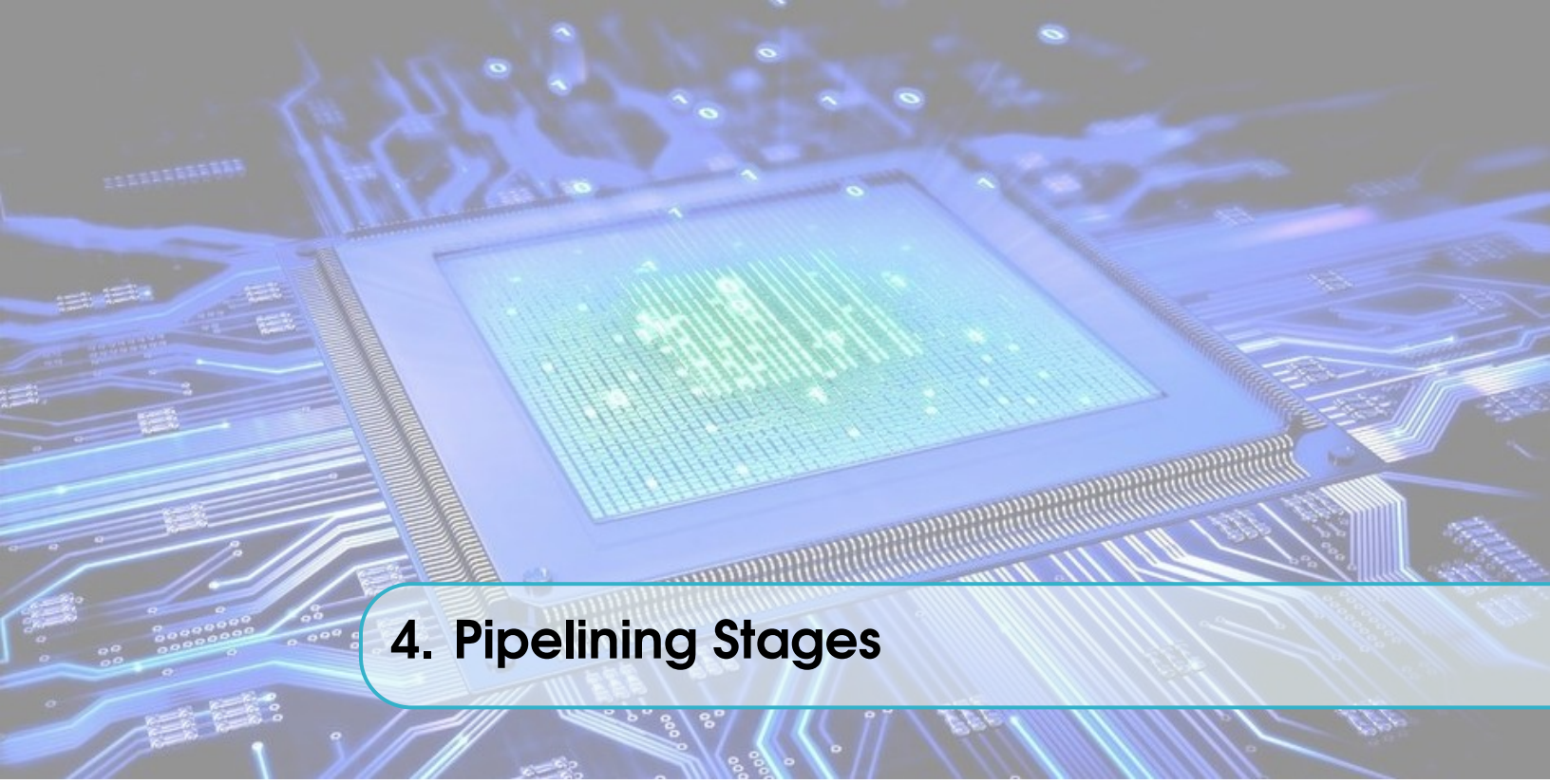


Figure 3.1: Proposed Datapath



4. Pipelining Stages

Instruction pipe-lining is a technique that implements a form of parallelism called instruction-level parallelism within a single processor. It therefore allows faster CPU throughput (the number of instructions that can be executed in a unit of time) than would otherwise be possible at a given clock rate. The basic instruction cycle is broken up into a series called a pipeline, each part forming a stage of execution. We propose using 6 such different stages for the execution of an instruction are defined below. This forms the 'Assembly Line' for implementing the task at hand.

4.1 Instruction Fetch

The instruction is fetched from the instruction memory in this stage of the process flow. This stage marks the beginning of each instruction execution.

4.2 Instruction Decode

The instruction is moved to IR. Controller generates the corresponding control signals to Register file and other datapath elements to extract needed values from the Register file. The stage is also used to interpret different decision (such as Branch action, etc), thus Branch control is also implemented in this stage.

4.3 Register Fetch

Operation takes place in the Register File. E.g for ADD operation, required register data would be read.

4.4 Execute

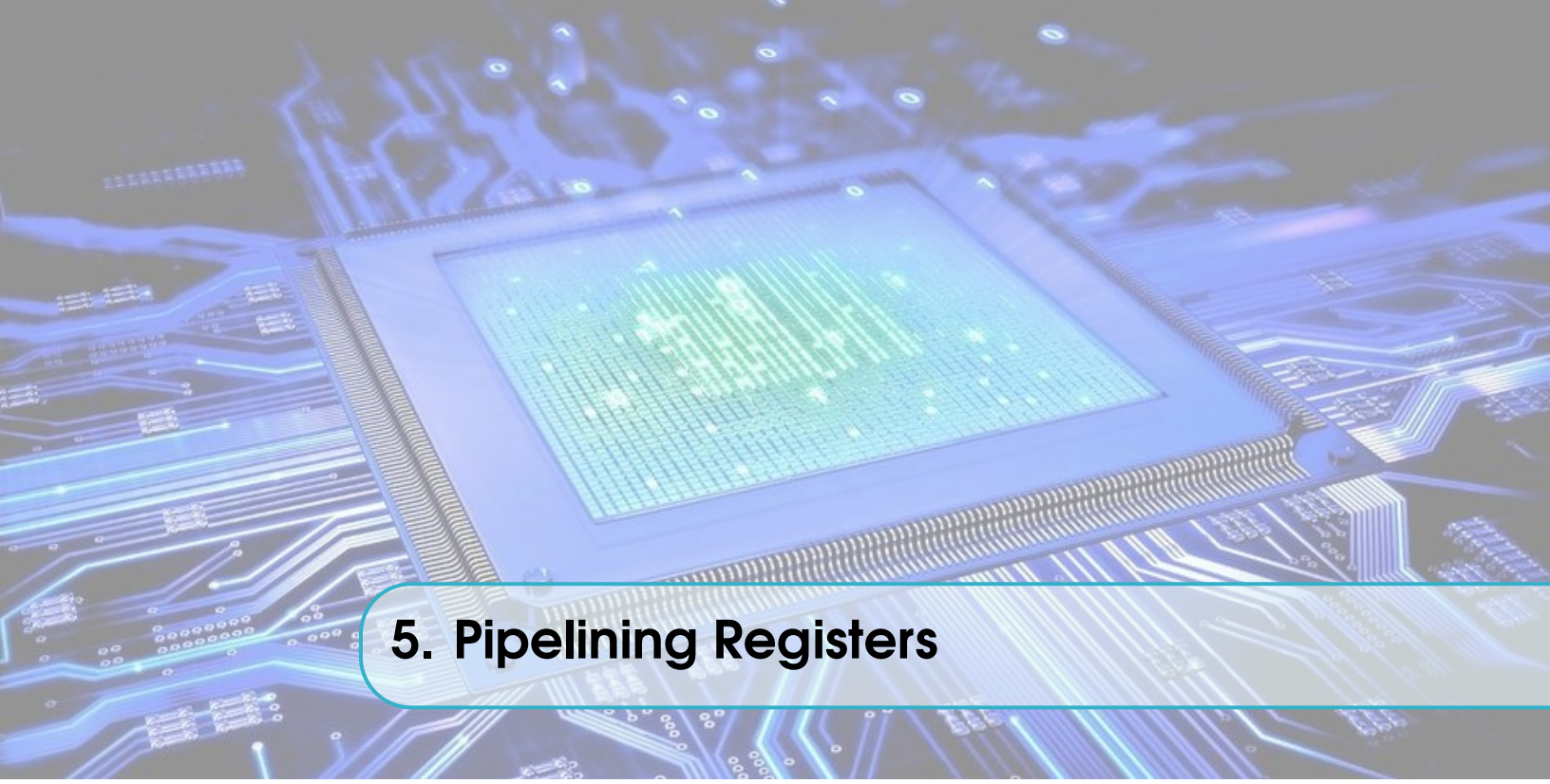
Required operation like ADD or NAND is performed on the data items which were previously read from registers.

4.5 Memory Access

This stage is required to generate the memory address where the corresponding data result is supposed to be written, e.g in PC or some other register in the register file.

4.6 Write Back

Control signals are generated to write the signal to the desired location.



5. Pipelining Registers

At the boundary of each state, we need certain signals or relevant data which was generated from the previous stage. Or, we may need to store some special data items which might be required for certain instruction execution. This is achieved by inserting pipelining registers at the these boundaries. (Please refer to the datapath to have a visual feel).

5.1 Pipelining Register 1

It has a flush input which if HIGH, sets IR to NOP instruction. “first_multiply” register is used when new instruction is “Load/Store multiple”.

5.2 Pipelining Register 2

The addresses Ra, Rb, Rc are stored here as an output from the decode stage. The value of PC+1 is stored here along with top 4 bits of IR to be used in the execute stage. It also contains the 1-bit control register and the “first_multiple”.

5.3 Pipelining Register 3

Holds the output of the Reg File. PC+Imm6 is passed to MUX to store into PC during branch instruction. The EQU output and opcode bits are passed to the Forwarding module

5.4 Pipelining Register 4

Holds the PC+Imm value for the branching instruction depending on the value of the flag. Includes the value of the CCR register and Ra from IR. This is used in the LM/SM instruction

5.5 Pipelining Register 5

This holds the output of the ALU and the memory-output which is used in the write_back stage.

6. Hazards

6.1 Data Hazards

op	R1 = R2 op R3											
op	R4 = R1 op R2											
		IF	ID	Reg	Ex	Mem	WB				Muxes F1 or F2 use a	
			IF	ID	Reg	Ex	Mem	WB				
op	R1 = R2 op R3											
op with CCR dependence												
		IF	ID	Reg	Ex	Mem	WB				forward x into ccr	
			IF	ID	Reg	Ex	Mem	WB				
op	R1 = R2 op R3											
x												
op with CCR dependence												
		IF	ID	Reg	Ex	Mem	WB				forward y into ccr	
			x	x	x	x	x	x				
				IF	ID	Reg	Ex	Mem	WB			

op	R1 = R2 op R3											
op	R4 = R1 op R2											
		IF	ID	Reg	Ex	Mem	WB				Muxes F1 or F2 use a	
			IF	ID	Reg	Ex	Mem	WB				
op	R1 = R2 op R3											
op with CCR dependence												
		IF	ID	Reg	Ex	Mem	WB				forward x into ccr	
			IF	ID	Reg	Ex	Mem	WB				
op	R1 = R2 op R3											
x												
op with CCR dependence												
		IF	ID	Reg	Ex	Mem	WB				forward y into ccr	
			x	x	x	x	x	x				
				IF	ID	Reg	Ex	Mem	WB			

op	R1 = R2 op R3											
op	R4 = R1 op R2											
		IF	ID	Reg	Ex	Mem	WB				Muxes F1 or F2 use a	
			IF	ID	Reg	Ex	Mem	WB				
op	R1 = R2 op R3											
op with CCR dependence												
		IF	ID	Reg	Ex	Mem	WB				forward x into ccr	
			IF	ID	Reg	Ex	Mem	WB				
op	R1 = R2 op R3											
x												
op with CCR dependence												
		IF	ID	Reg	Ex	Mem	WB				forward y into ccr	
			x	x	x	x	x	x				
				IF	ID	Reg	Ex	Mem	WB			

op	R1 = R2 op R3											
op	R4 = R1 op R2											
		IF	ID	Reg	Ex	Mem	WB				Muxes F1 or F2 use a	
			IF	ID	Reg	Ex	Mem	WB				
op	R1 = R2 op R3											
op with CCR dependence												
		IF	ID	Reg	Ex	Mem	WB				forward x into ccr	
			IF	ID	Reg	Ex	Mem	WB				
op	R1 = R2 op R3											
x												
op with CCR dependence												
		IF	ID	Reg	Ex	Mem	WB				forward y into ccr	
			x	x	x	x	x	x				
				IF	ID	Reg	Ex	Mem	WB			

op	R1 = R2 op R3															
X	X															
op	R4 = R1 op R2															
		IF	ID	Reg	Ex	Mem	WB				Muxes F1 or F2 use b					
				IF	ID	Reg	Ex	Mem	WB							
								Unless X = another op conflict, in which case use a.								
								(closest conflict)								
R7 =R1 (unconditional operation) R2		IF	ID	Reg	Ex	Mem	WB									
			x	x	x	x	x	x								
				x	x	x	x	x	x						Forward a to PC	
					x	x	x	x	x	x						
						IF	ID	Reg	Ex	Mem	WB					

R3 = R1 op R2		IF	ID	Reg	Ex	Mem	WB									
R7 = R1 op(conditional) R2			IF	ID	Reg	Ex	Mem	WB								
(next)				x	x	x	x	x	x							
					x	x	x	x	x	x						
				Case 1:		IF	ID	Reg	Ex	Mem	WB	(if the first instruction did not set CCR)				
				Case 2:		x	x	x	x	x	x					
		(forward from a)					IF	ID	Reg	Ex	Mem	WB	(if the first instruction did set CCR)			
op	R1 = R2 op R3															
load	R4 = mem(R1 +Im6)															
		IF	ID	Reg	Ex	Mem	WB				Muxes F1 or F2 use a					
			IF	ID	Reg	Ex	Mem	WB								

op	R1 = R2 op R3																		
X	X																		
load	R4 = mem(R1 +Im6)																		
		IF	ID	Reg	Ex	Mem	WB				Muxes F1 or F2 use b								
				IF	ID	Reg	Ex	Mem	WB										
load	R4 = mem(R1 +Im6)																		
op	R2 = R4 op R3																		
		IF	ID	Reg	Ex	Mem	WB				Muxes F1 or F2 use c								
	stall		x	x	x	x	x												
				IF	ID	Reg	Ex	Mem	WB										
								load		load									
								x	and	x	use just forward, stall is implemented by the intermediate instruction								
load	R4 = mem(R1 +Im6)							op		load									
load	R3 = mem(R4 +Im6)																		
			IF	ID	Reg	Ex	Mem	WB			Mux F1 uses c								
		stall		x	x	x	x	x											
					IF	ID	Reg	Ex	Mem	WB									

op	R1 = R2 op R3																		
store	mem(R4+Im6) = R1																		
		IF	ID	Reg	Ex	Mem	WB												
			IF	ID	Reg	Ex	Mem	WB											
op	R1 = R2 op R3																		
store	mem(R1+Im6) = R2																		
		IF	ID	Reg	Ex	Mem	WB												
			IF	ID	Reg	Ex	Mem	WB											
op	R1 = R2 op R3																		
store	mem(R1+Im6) = R1																		
		IF	ID	Reg	Ex	Mem	WB												
			IF	ID	Reg	Ex	Mem	WB											

op	R1 = R2 op R3																		
x	x																		
store	mem(R1+Im6) = R2																		
		IF	ID	Reg	Ex	Mem	WB												
			x	x	x	x	x	x											
				IF	ID	Reg	Ex	Mem	WB										
op	R1 = R2 op R3																		
x	x																		
store	mem(R1+Im6) = R1																		
		IF	ID	Reg	Ex	Mem	WB												
			x	x	x	x	x	x											
				IF	ID	Reg	Ex	Mem	WB										
load	R2 = mem(R1+Im6)																		
store	mem(R3+Im6) = R2																		
		IF	ID	Reg	Ex	Mem	WB												
			IF	ID	Reg	Ex	Mem	WB											

load	R2 = mem(R1+Im6)											
store	mem(R2+Im6) = R3											
		IF	ID	Reg	Ex	Mem	WB					
	stall		x	x	x	x	x	x				
				IF	ID	Reg	Ex	Mem	WB		Mux F1 uses c	
load	R2 = mem(R1+Im6)											
store	mem(R2+Im6) = R2											
		IF	ID	Reg	Ex	Mem	WB					
	stall		x	x	x	x	x	x				
				IF	ID	Reg	Ex	Mem	WB		Mux F1 uses c	

After JAL and JLR stalls are always required.followed by a forward												
JAL	R2 = PC+1	IF	ID	Reg	Ex	Mem	WB					
x	x		x	x	x	x	x	x				
				IF	ID	Reg	Ex	Mem	WB		Mux F4 uses d	
JAL	R2 = PC+1	IF	ID	Reg	Ex	Mem	WB					
op	R3 = R2 op R1		x	x	x	x	x	x			Muxes F1 or F2 use b	
				IF	ID	Reg	Ex	Mem	WB		Mux F4 uses d	
JAL	R2 = PC+1	IF	ID	Reg	Ex	Mem	WB					
load	R3 = mem(R2 +Im6)		x	x	x	x	x	x			Muxes F1 or F2 use b	
				IF	ID	Reg	Ex	Mem	WB		Mux F4 uses d	
JAL	R2 = PC+1	IF	ID	Reg	Ex	Mem	WB					
store	mem(R2 +Im6) = R3		x	x	x	x	x	x			Muxes F1 or F2 use b	
				IF	ID	Reg	Ex	Mem	WB		Mux F4 uses d	

JAL	R2 = PC+1	IF	ID	Reg	Ex	Mem	WB					
store	mem(R2 +Im6) = R2		x	x	x	x	x	x			Muxes F1 or F2 use b	
				IF	ID	Reg	Ex	Mem	WB		Mux F4 uses d	
JAL		IF	ID	Reg	Ex	Mem	WB					
op R1 = R2+R7			x	x	x	x	x	x			use g to forward appropriate value of R7	
				IF	ID	Reg	Ex	Mem	WB			
JLR		IF	ID	Reg	Ex	Mem	WB					
			x	x	x	x	x	x				
				x	x	x	x	x	x			
					IF	ID	Reg	Ex	Mem	WB	Mux F4 uses e	
Note JLR has no error, if followed by R7 operation												

Beq												
		IF	ID	Reg	Ex	Mem	WB					
			IF	ID	NOP	NOP	NOP	NOP				
				IF	NOP	NOP	NOP	NOP	NOP		flush the first two pipline registers.	
											NOP = no operation, when IR = 16'b1111000000000000	
				Shown what happens if branch is taken, else branch not taken, normal with no NOP								

LDW R7		IF	ID	Reg	Ex	Mem	WB									
or LDM 1xxxxxxx			x	x	x	x	x	x								
				x	x	x	x	x	x						Forward c to be loaded into PC!	
					x	x	x	x	x	x						
						x	x	x	x	x	x					
							x	x	x	x	x	x				
							IF	ID	Reg	Ex	Mem	WB				
LHI R7,Im6																
	only one stall															
	forward h to be loaded into PC															
LHI R5,Im6		IF	ID	Reg	Ex	Mem	WB				Forward i to the inputs of ALU					
R1 R2 op R5			IF	ID	Reg	Ex	Mem	WB								
LHI R5,Im6		IF	ID	Reg	Ex	Mem	WB				Forward j to the inputs of ALU					
x																
R1 R2 op R5				IF	ID	Reg	Ex	Mem	WB							