IITB-RISC

Project 1

Multi-Cycle Implementation

OV Shashank; Avineil Jain; Pratik Brahma; Yogesh Mahajan
IIT BOMBAY

CONTENTS

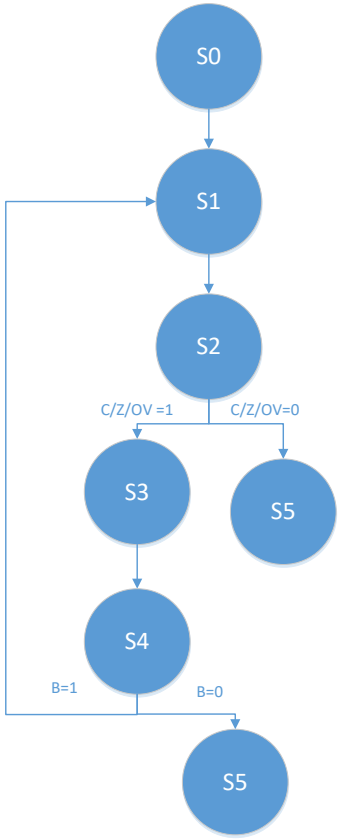
MICROPROCESSOR DESIGN

STATE ELABORATION

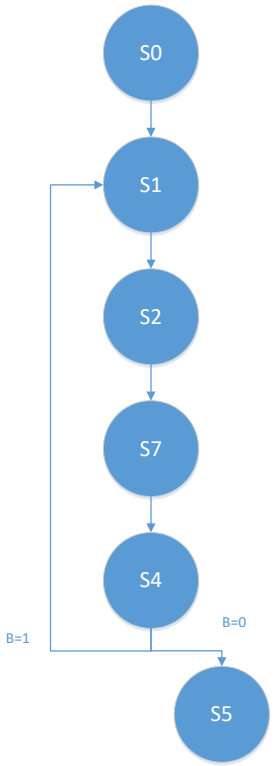
S1	R7 → MEM (A) MEM (D)→ IR R7 → ALU +1 → ALU ALU → PC
S2	$I_{6-8} \rightarrow A1_{RF}$ $I_{9-11} \rightarrow A2_{RF}$ D1 → E1 D2 → E2,T1 $I_{0-7} \rightarrow PE_{INPUT}$
S3	E1 → ALU E2 → ALU ALU → T1
S4	I_{3-5} /$I_{6-8} \rightarrow A3_{RF}$ T1 → $D3_{RF}$
S5	PC → $D3_{RF}$ "111" → $A3_{RF}$ T2 → ALU 0 → ALU
S6	$I_{0-8} \rightarrow SE_{9-16} \rightarrow LS_7$ $LS_7 \rightarrow D3_{RF}$ $I_{9-11} \rightarrow A3_{RF}$
S7	E1 → ALU $I_{0-5} \rightarrow SE_{6-16} \rightarrow$ ALU ALU → T1 , MEM(A)
S8	MEM (DO) → T2, $D3_{RF}$ $I_{9-11} \rightarrow A3_{RF}$
S9	D2 → MEM_{10}(DI) PC → $D3_{RF}$ "111" → $A3_{RF}$
S10	do { MEM_{DAT} (DO) → T2}
S11	T2 → $D3_{RF}$ $PE_{OUTPUT} \rightarrow A3_{RF}$ T1 → ALU +1 → ALU ALU → T1,MEM(DI)} while (! invalid_next);
S12	$PE_{OUTPUT} \rightarrow A2_{RF}$ T1 → MEM(A)
S13	T1 → ALU +1 → ALU ALU → T1 while (! invalid_next);
S14	R7 → ALU $I_{0-5} \rightarrow SE_{6-16} \rightarrow$ ALU ALU → PC
S15	PC → $D3_{RF}$ $I_{9-11} \rightarrow A3_{RF}$ R7 → ALU $I_{0-8} \rightarrow SE_{9-16} \rightarrow$ ALU ALU → PC
S16	$D1_{RF}$ → PC $I_{9-11} \rightarrow A3_{RF}$ PC → $D3_{RF}$

STATE FLOW DIAGRAM

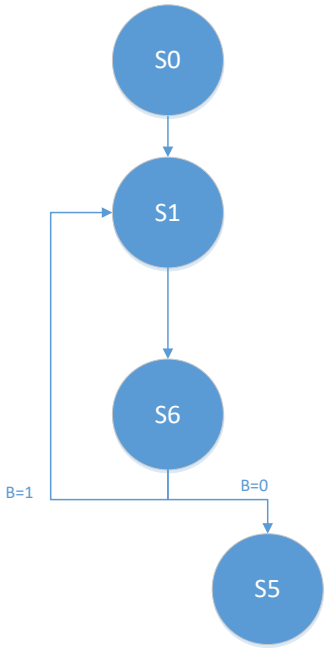
ADD/ADC/ADZ/NDU/NDC /NDZ/ADO/NDO



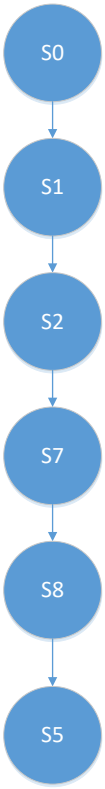
ADI



LHI

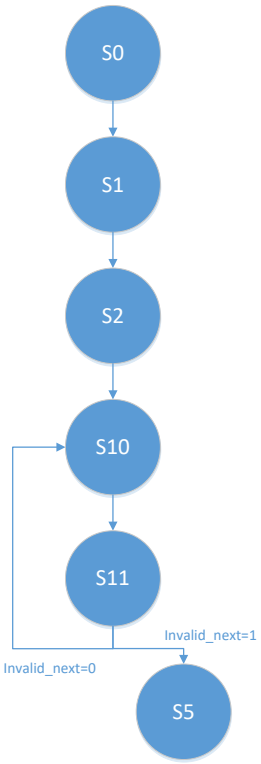


LW

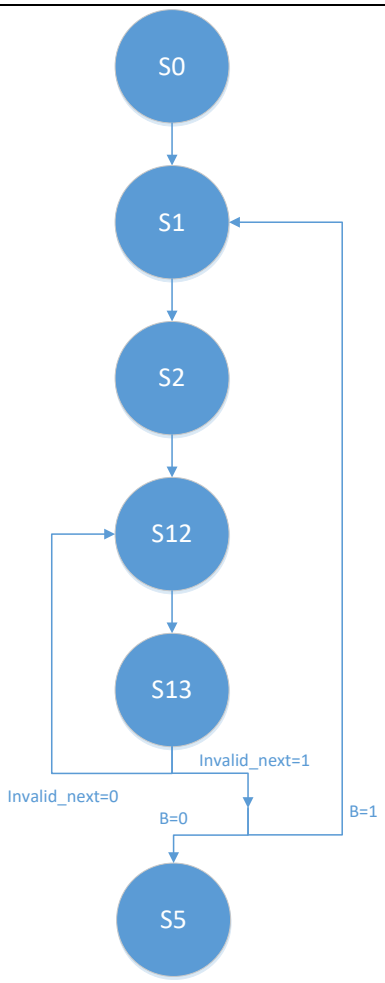


SW

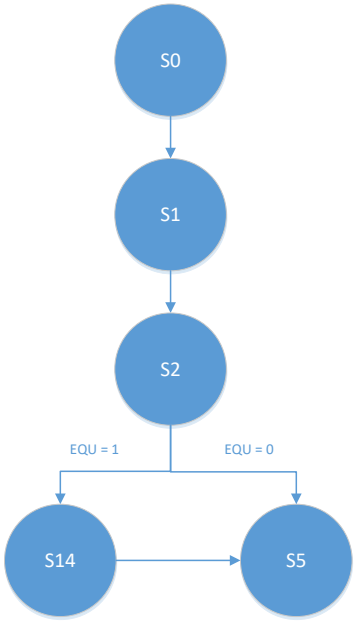
LM



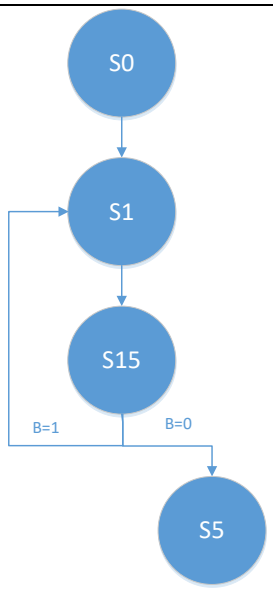
SM



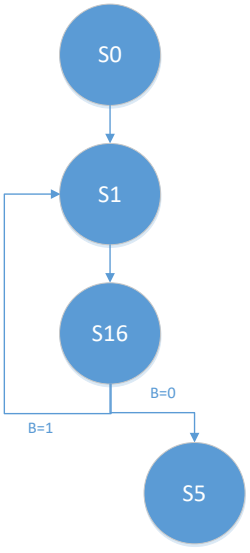
BEQ



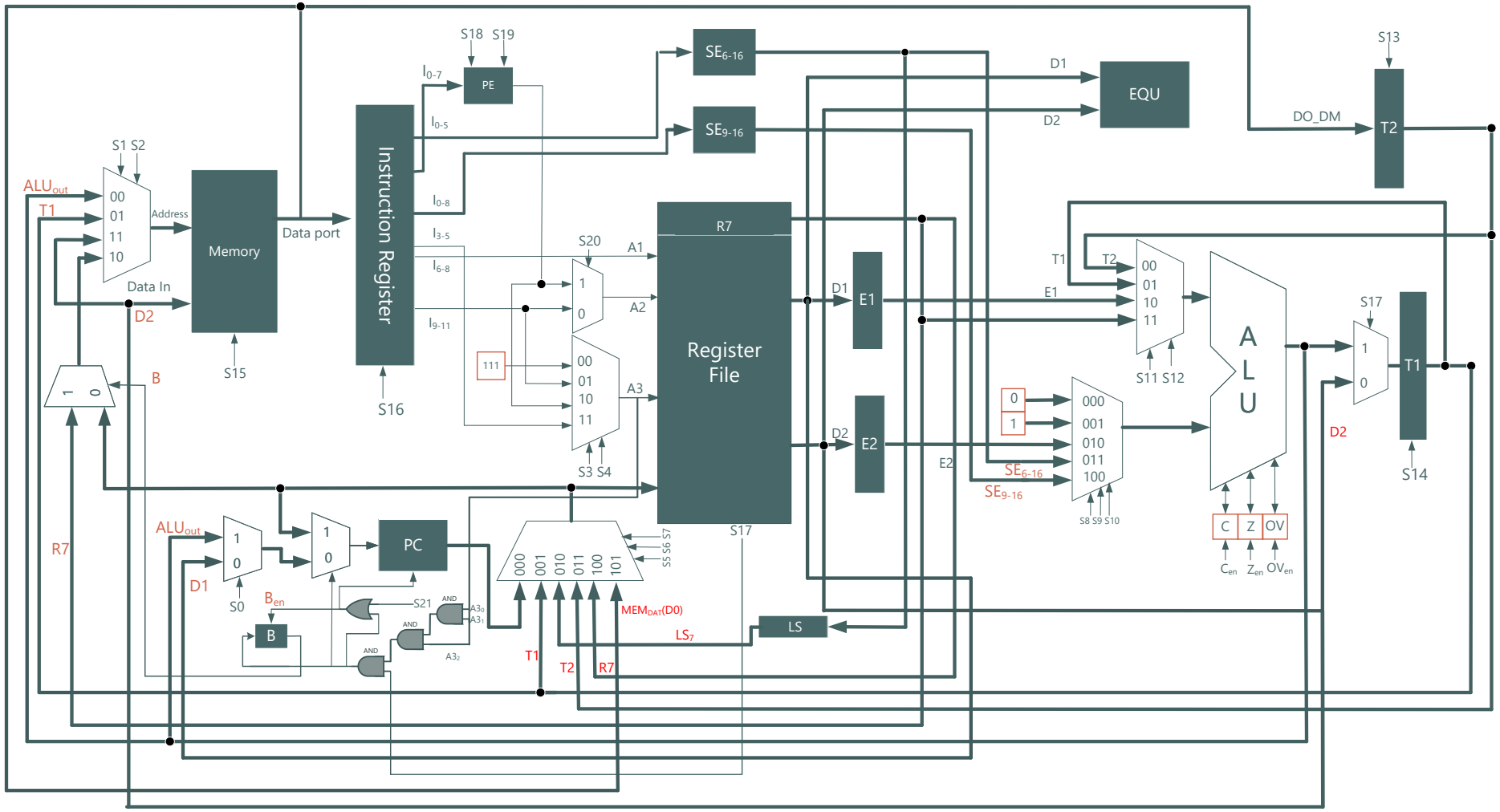
JAL



JLR



DATAPATH DESIGN



INSTRUCTION SET

IITB - RISC

INSTRUCTION SET

INSTRUCTIONS THAT AFFECT FLAG SETTINGS

Instruction	Flags			Instruction	Flags		
	C	Z	OV		C	Z	OV
ADD	X	X	X	NDU		X	
ADC	X	X	X	NDC		X	
ADZ	X	X	X	NDZ		X	
ADO	X	X	X	NDO		X	
ADI	X	X	X	LW		X	

INSTRUCTION SET AND ADDRESSING MODES

R_n	Registers R6-R0 of the Register Bank
data 6	Signed 6-bit constant included in instruction
data 9	Signed 9-bit constant included in instruction
rel 6	Signed (two's complement) 6-bit offset byte. Used by BEQ. Range is -32 to +31 short words relative to the present instruction
rel 9	Signed (two's complement) 9-bit offset byte. Used by JAL. Range is -256 to +255 short words relative to the present instruction
reg	8-bits representing registers R7-R0. Used by LM and SM
R_7	Program Counter
R	Registers R7-R0 of the Register Bank

INSTRUCTION ENCODING

ADD:	00_00	RA	RB	RC	0	00
ADC:	00_00	RA	RB	RC	0	10
ADZ:	00_00	RA	RB	RC	0	01
ADO:	00_00	RA	RB	RC	0	11
ADI:	00_01	RA	RB	data 6		
NDU:	00_10	RA	RB	RC	0	00
NDC:	00_10	RA	RB	RC	0	10
NDZ:	00_10	RA	RB	RC	0	01
NDO:	00_10	RA	RB	RC	0	11
LHI:	00_11	RA	data 9			
LW:	01_00	RA	RB	rel 6		
SW:	01_01	RA	RB	rel 6		
LM:	01_10	RA	reg			
SM:	01_11	RA	reg			
LLI:	10_11	RA	data 9			
BEQ:	11_00	RA	RB	rel 6		
JAL:	10_00	RA	rel 9			
JLR:	10_01	RA	RB	000_000		

INSTRUCTION SET

IITB - RISC

INSTRUCTION TIMING REFERENCE

Mnemonic		Flags			Oscillator Period
		C	Z	OV	
ADD	R_n, R, R	X	X	X	5
ADD	R_7, R, R	X	X	X	4
ADC	R_n, R, R	0	X	X	5
ADC	R_7, R, R	0	X	X	4
ADC	R_n, R, R	1	X	X	3
ADZ	R_n, R, R	X	0	X	5
ADZ	R_7, R, R	X	0	X	4
ADZ	R_n, R, R	X	1	X	3
ADO	R_n, R, R	X	X	0	5
ADO	R_7, R, R	X	X	0	4
ADO	R_n, R, R	X	X	1	3
NDU	R_n, R, R	X	X	X	5
NDU	R_n, R, R	X	X	X	4
NDC	R_n, R, R	0	X	X	5
NDC	R_7, R, R	0	X	X	4
NDC	R_n, R, R	1	X	X	3
NDZ	R_n, R, R	X	0	X	5
NDZ	R_7, R, R	X	0	X	4
NDZ	R_n, R, R	X	1	X	3
NDO	R_n, R, R	X	X	0	5
NDO	R_7, R, R	X	X	0	4
NDO	R_n, R, R	X	X	1	3

Mnemonic		Flags/ Condition	Oscillator Period
ADI	R_n, R, data6	X	5
ADI	R_7, R, data6	X	4
LHI	R_n, data9	X	3
LHI	R_7, data9	X	2
LLI	R_n, data9	X	3
LLI	R_7, data9	X	2
LW	R, R, rel6	X	5
SW	R, R, rel6	X	4
LM	R, reg	X	3+2*reg_count
SM	R, reg	X	3+2*reg_count
JAL	R_n, rel9	X	3
JAL	R_7, rel9	X	2
JLR	R_n, R	X	3
JLR	R_7, R	X	2
BEQ		Not Equal	3
BEQ		Equal	4

CUSTOM INSTRUCTIONS

- ADO: ADD if the overflow flag is set. The rest of the instruction format is the same as the other conditional execution instructions
- NDO: NAND if the overflow flag is set. The rest of the instruction format is the same as the other conditional execution instructions
- LLI: Load lower immediate. Load the immediate 9 bits into the register as mentioned in the instruction without sign extension by padding zeros in the upper 7 bits.

NOTE: The microprocessor is supposed to be a signed microprocessor and hence the overflow bit was introduced. It may be considered as the signed equivalent of the carry bit. The carry bit here still represents the unsigned version and hence the two must be used carefully.

CODES

IITB - RISC

VHDL CODES

(All the codes have been hyperlinked here for easy viewing; Notepad++ or gedit is recommended)

COMPONENTS

- ALU
- Priority Encoder
- Load/Store Multiple Logic Block
- Registers
- Memory (Generated using Altera's MegaWizard)
- Register File
- Sign Extender

MICROPROCESSOR BLOCKS

- Data Path: This consists of the entire data path along with all the transfers and the predicates corresponding to an RTL layout of the microprocessor. All the T and S signals are pretty accurately detailed as comments at the beginning of the architecture.
- Control Path: This consists of the controller Moore FSM. It has been decomposed into three processes; The first one describes the flip flops which control the states, the second one describes the next state logic and finally the third process controls the output logic based on the present state. (The order in the code might not exactly follow this order). All the T signals have been accompanied by the expected result they are to cause in the data path.
- IITB_RISC: The top-level entity than combines the data path and the controller FSM. The register 0 has been shown as an output so that the processes of the microprocessor can be displayed outside in hardware.

TESTING

- Test_final: This is a test bench which can be used to simulate the entire microprocessor in Altera's modelsim. It basically functions to provide the clock and reset signals to the microprocessor.

Testing results have been stored as images and are also available along with this document.

ASSEMBLER

The code assembler.py was developed as a combined effort of Kalpesh and Shashank. It basically works by taking as its input a text file containing code written in an assembly like language, and creates an INTEL HEX file which can be directly provided to Quartus which uses it to preload memory. A sample hex file produced by the code is provided along with this document. Also, a sample text input file has also been included.

QUARTUS PROJECT

The quartus project folder used for testing is also included along with this submission. This can be used for quick compilation and viewing in Quartus. Necessary settings have been implemented to allow for timing analysis and RTL and gate level simulation