Simple 8-bit Processor Design

ABSTRACT

The purpose of this project is to design, stimulate, and test an 8-bit multi-cycle microprocessor. Description of the processor will be written using Verilog HDL in register transfer level. Stimulation will be performed using ModelSim to demonstrate the executions of the processor's 11 instructions.

SPECIFICATION

The processor should have an 8-bit address bus and an 8-bit bidirectional data bus for accessing memory and registers. Processor's read and write operations should satisfy the timing specifications of external memory chips.

The processor should be able to do 11 instructions as described below(summarized in table 1):

• Load and Store instructions, including load-immediate, load and store, move data between memory and the register file.

• ALU instructions, including move, and, sub and add, perform ALU operations on register values and write the results back into the register file. Although the move is not performing a computation, it is using the transfer operation of the ALU. Note that the other ALU instructions, and, sub, and add, use register R1 as an implicit operand.

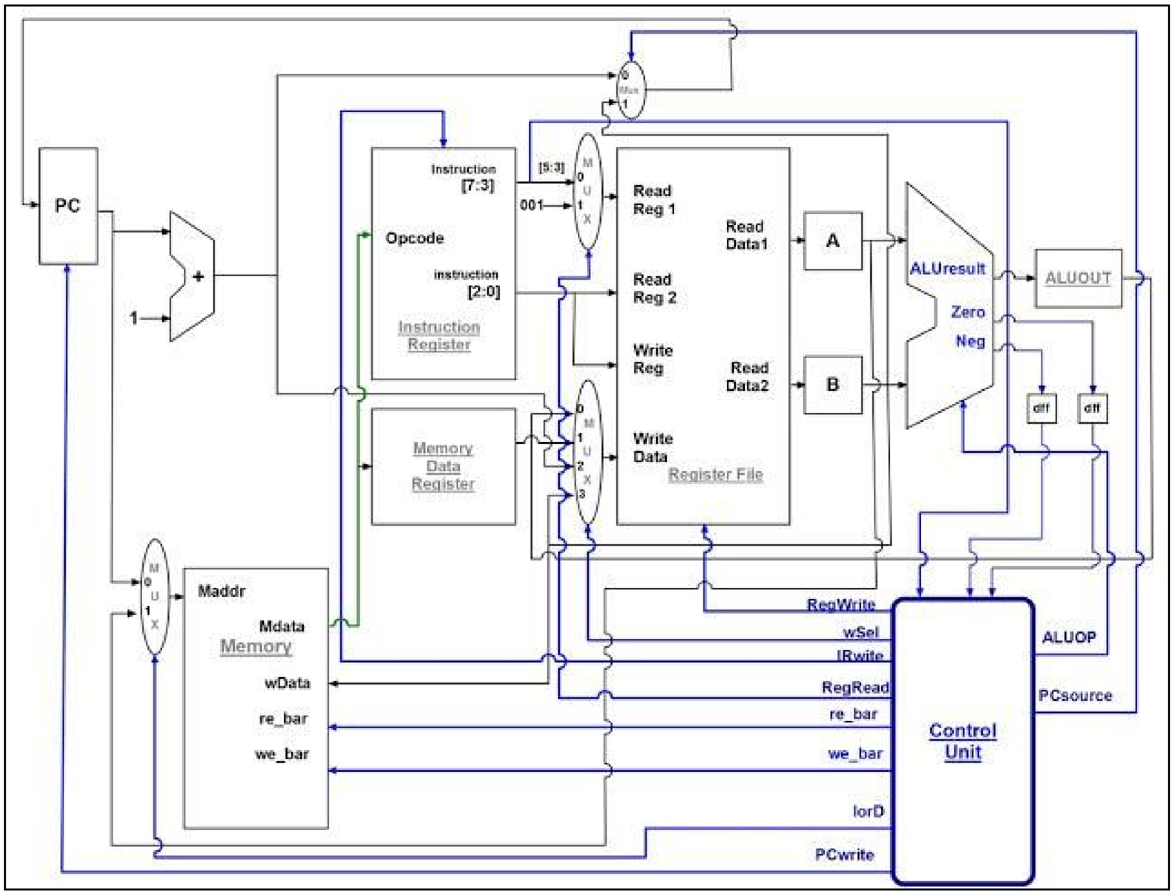
• Jump and Branch instructions, including branch, branch-if-negative, branch-if-zero, and jump-and-link, change the instruction execution sequence of a program.

Instruction	**Assembly Form**	**Instruction Format**			**Operation performed**
Move	MOV RA, RB	00	RA	RB	RB <= RA
Load	LD RA, RB	01	RA	RB	RB <= M[RA]
Store	ST RA, RB	10	RA	RB	M[RA] <= RB
And	AND RB	11	000	RB	RB <= RB AND R1*
Add	ADD RB	11	001	RB	RB <= RB + R1*
Sub	SUB RB	11	010	RB	RB <= RB – R1*
Branch	BR RB	11	011	RB	PC <= RB
Branch if zero	BRZ RB	11	100	RB	PC <= RB if Z=1
Branch if negative	BRN RB	11	101	RB	PC <= RB if N=1
Jump and link	JAL RB	11	110	RB	PC <= RB, RB <= PC + 1
Load immediate	LDI RB	11	111	RB	RB<=M[next byte], PC<=PC+1

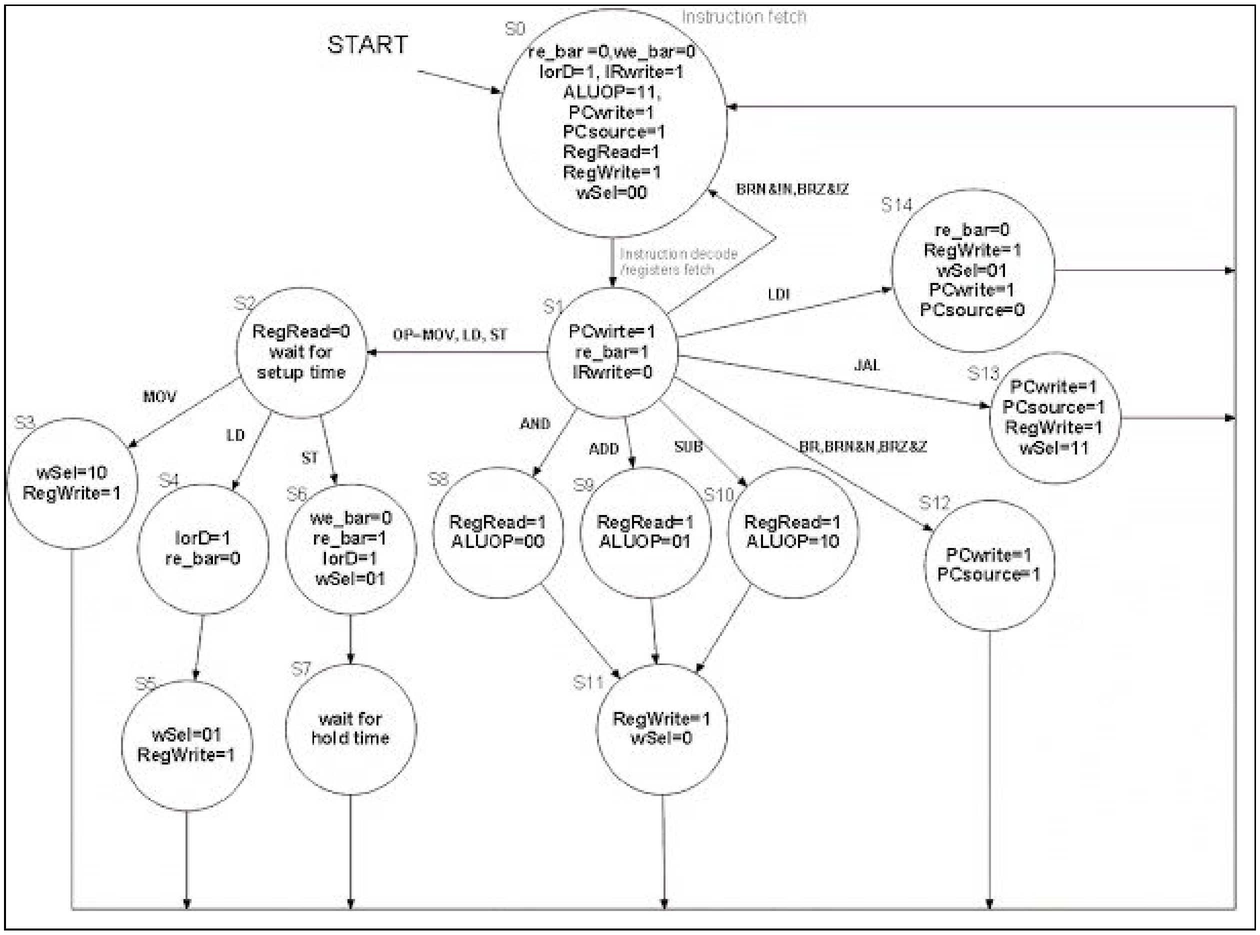
The processor executes instructions in 4 stages:
1. Instruction fetch: Fetch the instruction from the memory(EPROM), latch it into the Instruction Register (IR) and increment the program counter.
2. Instruction decode: decode the instruction to decide which actions to perform. Determine the instruction from the opcode field and determine the operands from the operand fields.
3. Computation, Jump or Branch completion, or Memory access:
o Computation: Perform the ALU operation on the operand or operands specified by the instruction.
o Jump or branch completion: Load the PC with the target address specified by the instruction. Store the link address in the register file if instruction is JAL.
o Memory access: Place the address on the address bus.
4. Memory access completion or Write-back: Perform memory access, or store the ALU result in the destination register specified by the instruction.

DESIGN

I designed the processor using multicycle implementation, and used finite state machine to implement the control units of the processor. A complete data path of our multicycle implementation is shown in figure below:



To make the processor to execute instructions 4 steps as explained above, I use necessary number of registers to hold the data between each steps and clock cycles. I use 4 mux to select the appropriate inputs for the pc, registers, and memory. To simplify the design, I use an addition adder to perform the operation of PC+1, although this operation could be done using the
ALU. With such a simple design of using adder for PC, the implementation of the processor become more simple and flexible, especially in accomplishing instructions like JAL and LDI.
To allow the processor to select appropriate data in different step of execution and clock cycles, I implement the control unit of the processor with a finite state machine(Moore), which has total of 15 states. A detailed state diagram of the control unit is showed in the state diagram below:

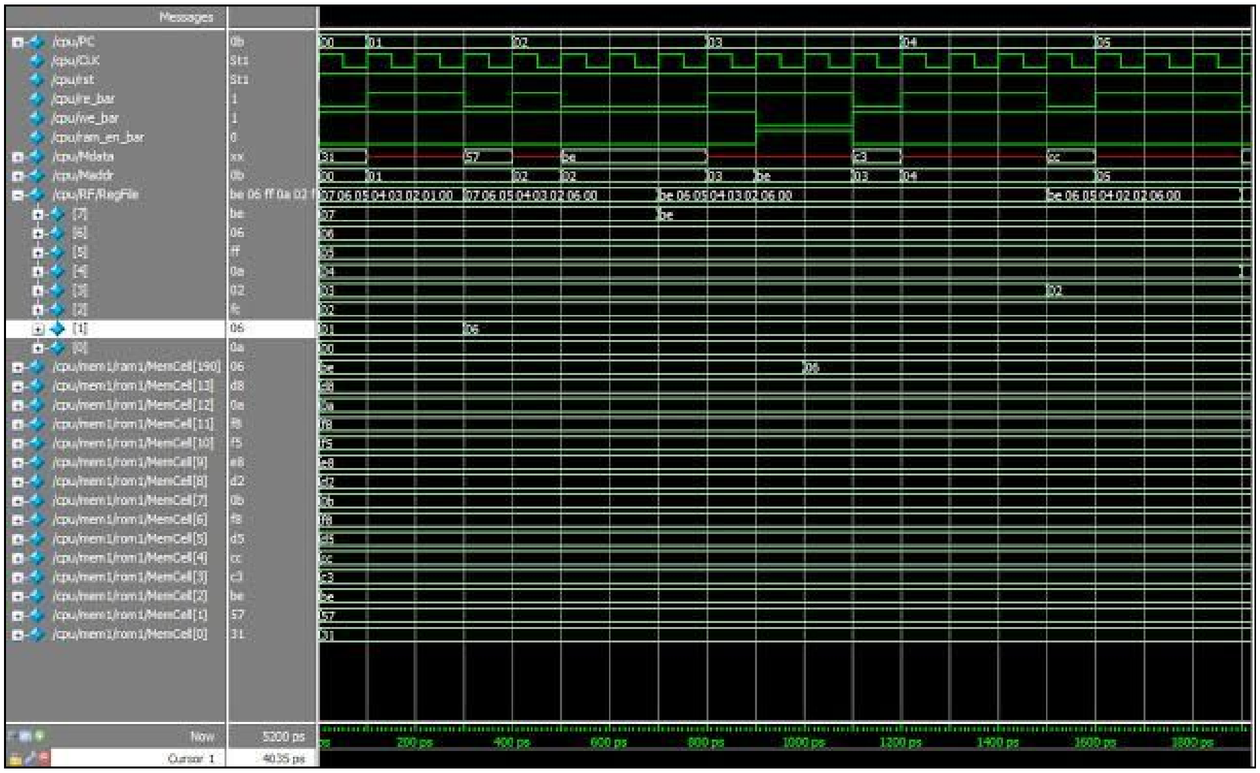


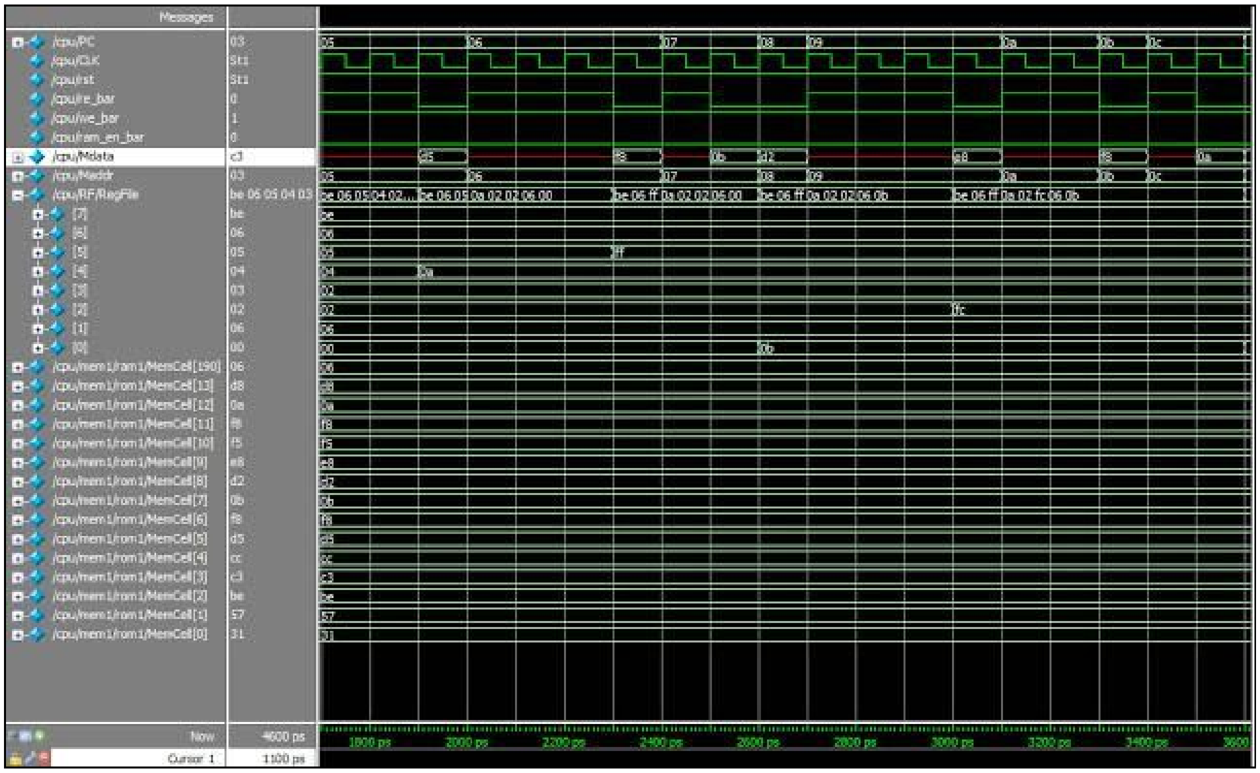
STIMULATE AND TEST

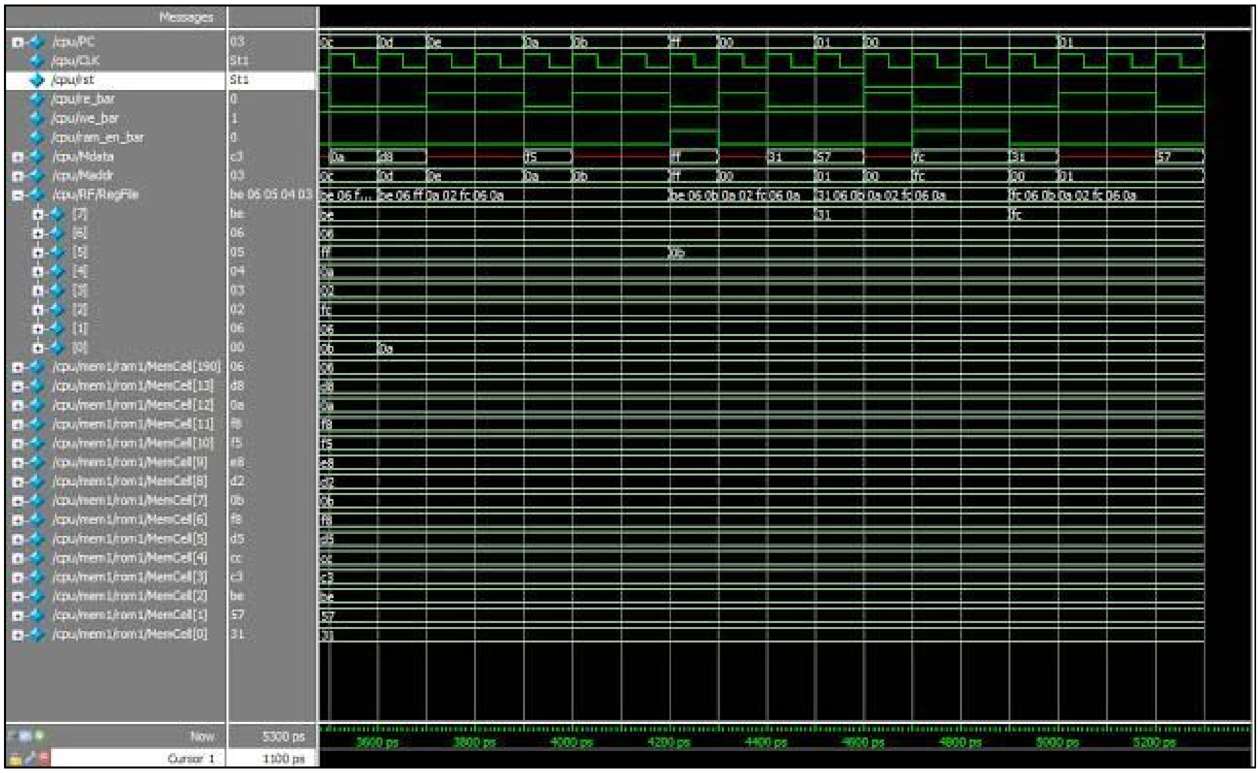
To test and stimulate our implementation of the multicycle processor, I use Verilog to write a RTL description for my design. Source codes are in APPENDIX SECTION. I also use Modelsim to perform a functional stimulation. To verify the functionality of the processor, I have pre-loaded the EPROM with the instructions listed in Table 2. I initially set the PC to be 0 and set the data in the registers R0 to R7 to be 0, 1,2,3,4,5,6,7, respectively.

PC / Maddr	Machine Code	Instruction/Assembly Code	Operations Performed/Results	PC after instruction
0	00110001	MOV R6,R1	R6 <= 01	1
1	01010111	LD R2, R7	R7 <= M[2] (M[2]=be)	2
2	10111110	ST R7, R6	M[be] <= R6 (R6=06)	3
3	11000011	AND R3	R3 <= 02 (R1=06, 03AND06=02)	4
4	11001100	ADD R4	R4 <= 0a (04+06 = 0a)	5
5	11010101	SUB R5	R5 <= ff (05-06=ff), Neg<=1	6
6	11111000	LDI R0	R0 <= 0b (M[7]=0b)	8
7	00001011	"Immediate value =0b"	Skip since instruction is LDI	
8	11010010	SUB R2	R2 <= fc (02 – 06 = fc), Neg<=1	9
9	11101000	BRN R0	Check Neg=1, PC <= 0b	0b
0a	11110101	JAL R5	R5<=0b , PC<=ff	ff
0b	11111000	LDI R0	R0 <= 0a	0d
0c	00001010	"Immediate value =0a"	Skip since instruction is LDI	
0d	11011000	BR R0	PC <= 0a	0a

The wave forms shows the execution of the instructions in table 2. The results are expected; all instructions are completed properly as I trace the instructions in the wave forms. In the simulation, the processor execute instructions from PC=0 to PC=9, and then it jump to PC=0a. After instructions in PC=0b and 0c, it jump back to PC=0a to execute the instruction "JAL R5", which writes hex value of 0b in register R5 and jump to PC=ff. At PC=ff (M[ff]=opcode = 1111111), instruction is "LDI R7" which should load the value in M[next byte of ff] to register R7, but there is no next byte after memory address ff. As shown in figure 5, the result of this instruction is loaded the value in M[0] to register R7. This shows that the LDI as well as all other memory access instructions can always access memory. The waveforms also verified all the instructions are working properly in our implementation. The results of the testing instructions are listed in table 2 column 4 and in the waveforms(figure 3-5) shown below.


Messages
/cpu/PC 0b
/cpu/CLK St1
/cpu/rst St1
/cpu/re_bar 1
/cpu/we_bar 1
/cpu/ram_en_bar 0
/cpu/Mdata xx
/cpu/Maddr 0b
/cpu/RF/RegFile be 06 ff 0a 02 f
[7] be
[6] 06
[5] ff
[4] 0a
[3] 02
[2] fc
[1] 06
[0] 0a
/cpu/mem1/ram1/MemCell[190] 06
/cpu/mem1/rom1/MemCell[13] d8
/cpu/mem1/rom1/MemCell[12] 0a
/cpu/mem1/rom1/MemCell[11] f8
/cpu/mem1/rom1/MemCell[10] f5
/cpu/mem1/rom1/MemCell[9] e8
/cpu/mem1/rom1/MemCell[8] d2
/cpu/mem1/rom1/MemCell[7] 0b
/cpu/mem1/rom1/MemCell[6] f8
/cpu/mem1/rom1/MemCell[5] d5
/cpu/mem1/rom1/MemCell[4] cc
/cpu/mem1/rom1/MemCell[3] c3
/cpu/mem1/rom1/MemCell[2] be
/cpu/mem1/rom1/MemCell[1] 57
/cpu/mem1/rom1/MemCell[0] 31
Now 5200 ps
Cursor 1 4035 ps
200 ps
400 ps
600 ps
800 ps
1000 ps
1200 ps
1400 ps
1600 ps
1800 ps


Messages
/cpu/PC 03
/cpu/CLK St1
/cpu/rst St1
/cpu/re_bar 0
/cpu/we_bar 1
/cpu/ram_en_bar 0
/cpu/Mdata c3
/cpu/Maddr 03
/cpu/RF/RegFile be 06 05 04 03
[7] be
[6] 06
[5] 05
[4] 04
[3] 03
[2] 02
[1] 06
[0] 00
/cpu/mem1/ram1/MemCell[190] 06
/cpu/mem1/rom1/MemCell[13] d8
/cpu/mem1/rom1/MemCell[12] 0a
/cpu/mem1/rom1/MemCell[11] f8
/cpu/mem1/rom1/MemCell[10] f5
/cpu/mem1/rom1/MemCell[9] e8
/cpu/mem1/rom1/MemCell[8] d2
/cpu/mem1/rom1/MemCell[7] 0b
/cpu/mem1/rom1/MemCell[6] f8
/cpu/mem1/rom1/MemCell[5] d5
/cpu/mem1/rom1/MemCell[4] cc
/cpu/mem1/rom1/MemCell[3] c3
/cpu/mem1/rom1/MemCell[2] be
/cpu/mem1/rom1/MemCell[1] 57
/cpu/mem1/rom1/MemCell[0] 31
Now 4600 ps
Cursor 1 1100 ps
1800 ps
2000 ps
2200 ps
2400 ps
2600 ps
2800 ps
3000 ps
3200 ps
3400 ps
3600



Component Test

TetraMax is used to find all faults and generate test sets with ATPG.

All the stuck-at faults for the combinational components:

	2to1 mux	4to1 mux	Decoder for controller	ALU
Detected Faults	114	276	22	489
Possibly detected	0	0	0	0
Undetectable Faults	0	0	0	11
ATPG untestable	0	0	0	0

Not detected	0	0	0	0
Total Faults	114	276	22	500
Test coverage	100%	100%	100%	100%
Number of tests	10	20	10	32

For components 2to1 mux and 4to1 mux, 114 and 275 faults were found, respectively, with no undetectable faults. The 2to1 mux needs 10 tests to detect all faults, and the 4to1 mux needs 20 tests to detect all faults. For the 7 to 4 decoder used in the control unit, 22 faults are found and 10 tests are generated. For the ALU, total of 500 faults were found, but 11 of these faults are redundant. To detect all detectable faults for the ALU, 32 test patterns are needed.