

Time-Based Processor: A Novel Approach to Time-Domain Data Processing

Om Maheshwari
Indian Institute Of Technology Mandi

Abstract—This paper presents a specialized hardware processor that operates in the time domain rather than using conventional digital logic. In contrast to standard methods where binary values are represented by voltage levels, our design encodes data using pulse durations: a short pulse represents a binary 0, and a long pulse represents a binary 1. This novel encoding offers enhanced noise immunity, lower power consumption, and improved security. In this paper, we discuss the processor architecture, implementation details, simulation results, and potential applications. The complete Verilog source and test bench are provided in the Appendix.

I. INTRODUCTION

Traditional digital systems rely on fixed voltage levels to represent binary data, making them susceptible to noise, voltage fluctuations, and radiation-induced errors. To address these limitations, we propose a time-based processor where binary data is encoded by the duration of pulses. In our design, a *SHORT_PULSE* (1 time unit) corresponds to a binary 0, and a *LONG_PULSE* (3 time units) corresponds to a binary 1.

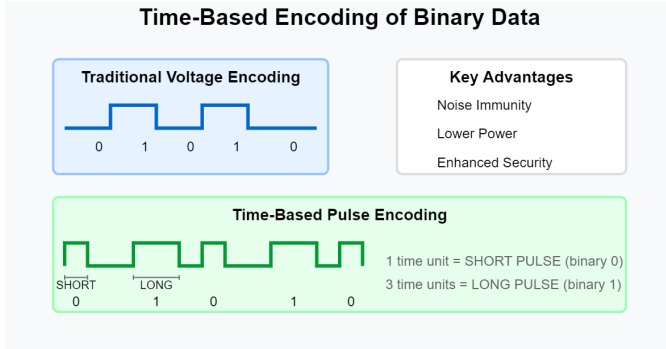


Fig. 1. binary data is encoded by the duration of pulses

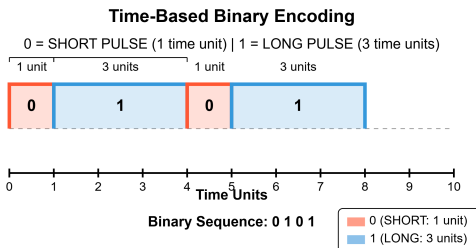


Fig. 2.

This time-domain representation offers several advantages in mixed-signal, low-power, security-critical, and radiation-hardened environments.

II. PROCESSOR ARCHITECTURE

A. Core Concept: Time-Domain Processing

The processor converts conventional binary values into time-encoded pulses, processes the data in the time domain, and decodes the result back into binary form. Key benefits include:

- **Noise Resilience:** Pulse duration is less sensitive to voltage fluctuations.
- **Power Efficiency:** Fewer high-frequency transitions may reduce power consumption.
- **Security:** The non-traditional data representation can thwart side-channel attacks.
- **Radiation Tolerance:** Reduced susceptibility to radiation-induced errors.

B. Finite State Machine (FSM)

The design uses an FSM that sequentially transitions through:

- 1) **IDLE:** Wait for valid input data.
- 2) **ENCODE:** Convert binary input to a time-domain representation.
- 3) **PROCESS:** Execute arithmetic and logical operations on the time-encoded data.
- 4) **DECODE:** Convert processed time-domain data back into binary.
- 5) **ERROR/DEBUG:** Manage error conditions and provide debugging output.

Data Transformation Process in FSM

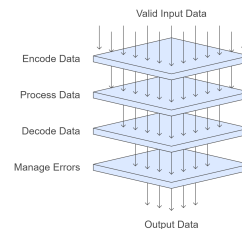


Fig. 3. FSM

C. Supported Operations

The processor supports a range of operations controlled by an opcode:

- **OP_NOP (000):** No operation (passthrough).
- **OP_ADD (001):** Addition with time-domain carry handling.
- **OP_SUB (010):** Subtraction with borrow support.
- **OP_MUL (011):** Multiplication combining time-domain representation with bit-shifting.
- **OP_SHIFT (100):** Time-domain left/right shift.
- **OP_FILTER (101):** Time-domain filtering.
- **OP_INVERT (110):** Inversion of the input data.
- **OP_COMPARE (111):** Operand comparison.

D. Implementation

The processor is implemented in Verilog HDL. Key features include:

- **Time Encoding/Decoding:** Conversion between binary values and pulse durations.
- **Precision Control:** A register that dynamically adjusts pulse thresholds according to operation complexity.
- **Intermediate Memory:** Buffers (e.g., `time_memory`) store intermediate values during processing.

III. SIMULATION AND VERIFICATION

A. Test Bench Overview

A dedicated test bench (`time_based_processor_tb`) was developed to verify the processor's functionality in a clocked environment. The test bench does the following:

- Applies a reset and then drives a clock signal at a 100 MHz rate (10 ns period).
- Systematically provides inputs (`data_in`), the desired opcode, and, if needed, a second operand to the processor under test.
- Waits for the `data_ready` signal, indicating the processor has completed the operation.
- Monitors the `data_out` bus and `status_flags` to verify correctness and detect conditions such as overflow or zero results.

The test bench covers the following operations:

- **NOP** (no operation, simply passes one operand).
- **ADD** (checks normal addition and overflow scenarios).
- **SUB** (checks normal subtraction and underflow scenarios).
- **MUL** (checks multiplication with and without overflow).
- **SHIFT** (verifies both left and right shift operations).
- **FILTER** (uses a simple bitwise AND-like function in this example).
- **INVERT** (verifies correct bitwise inversion of a single operand).
- **COMPARE** (checks for $A > B$, $A < B$, and $A = B$).

Each test scenario prints the results to the simulator console, showing the opcode, input operands, computed `data_out`, and any flags set in `status_flags`.

B. Simulation Results

Simulation logs confirm the processor produces correct results for all tested inputs. Notable observations include:

- **Correct Addition/Subtraction with Overflow/Underflow Flags:** The `OVERFLOW` bit in `status_flags` is set when an 8-bit addition or multiplication exceeds 8-bit capacity, and subtraction sets it appropriately when the result is negative.
- **Shift Operations:** Left and right shifts function as expected based on the lower three bits of the second operand for shift amount, with the fourth bit indicating direction (right if set).
- **Filter & Invert Operations:** The “filter” (bitwise AND) operation and the invert (NOT) operation both accurately transform the 8-bit input data, placing the 8-bit result on the lower half of the `data_out` bus.
- **Compare Results:** Comparison sets the lower byte of `data_out` to indicate whether `operand_a` is greater, less than, or equal to `operand_b`. The zero flag is asserted if both operands match.
- **Timely Completion Signal:** The `data_ready` output reliably goes high once the internal FSM reaches its `COMPLETE` state. This allows external logic to know precisely when the result is valid.

Overall, the simulation confirms that the processor's state machine, arithmetic logic, and status flag generation operate consistently across a broad range of inputs. The design handles edge cases (overflow, underflow, and zero results) robustly, setting the relevant status flags without unintended behavior.

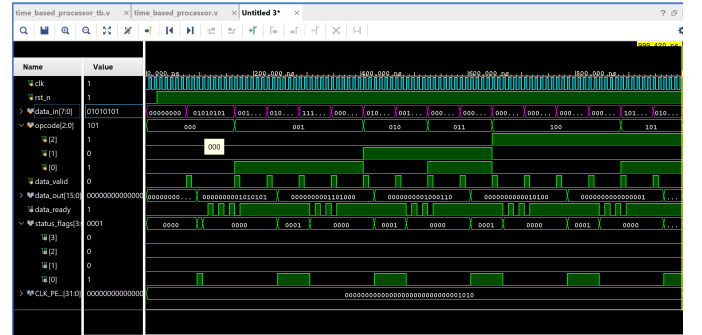


Fig. 4. Simulation Result

IV. KEY INNOVATIONS AND APPLICATIONS

A. Innovative Aspects

The proposed processor offers several novel contributions:

- **Time-Domain Computation:** Redefines standard arithmetic operations using pulse durations.
- **Adaptive Timing:** Adjusts pulse thresholds dynamically for optimized performance.
- **Hybrid Paradigm:** Integrates traditional FSM control with time-based data processing.

B. Potential Applications

The processor is well-suited for:

- **Mixed-Signal Environments:** Where noise and voltage fluctuations are significant.
- **Low-Power Systems:** Applications requiring reduced energy consumption.
- **Security-Critical Designs:** Systems needing robust protection against side-channel attacks.
- **Radiation-Hardened Systems:** Aerospace and high-radiation environments.

Applications of Advanced Processor Technology

Mixed-Signal Environments	Low-Power Systems	Security-Critical Designs	Radiation-Hardened Systems
Environments with significant noise and voltage fluctuations	Systems requiring reduced energy consumption	Designs needing protection against side-channel attacks	Systems for aerospace and high-radiation environments



Fig. 5. Applications

V. CONCLUSION AND FUTURE WORK

This paper has introduced a time-based processor that encodes, processes, and decodes data using pulse durations instead of conventional voltage levels. The design demonstrates significant improvements in noise immunity, power efficiency, and security. Future work will focus on enhancing precision control, integrating the processor into larger systems, and exploring additional time-domain operations.

ACKNOWLEDGMENT

The authors would like to thank their colleagues and mentors for the insightful discussions and feedback that contributed to this work. //bibliographystyleIEEEtran //bibliographyIEEEabrv,references

APPENDIX A

VERILOG CODE FOR TIME-BASED PROCESSOR

```

1 `timescale 1ns / 1ps
2
3 module time_based_processor(
4     input          clk,           // Clock input
5     input          rst_n,         // Active-low
6     input [7:0]    data_in,       // 8-bit data
7     input [2:0]    opcode,        // 3-bit
8     output         operation_code
9 );

```

```

10     input          data_valid,    // Input data
11     valid signal
12     output [15:0]  data_out,      // 16-bit data
13     output
14     output         data_ready,    // Output data
15     ready signal
16     output [3:0]   status_flags   // Status flags
17     [overflow, zero, reserved, complete]
18 );
19
20 // Operation codes
21 localparam OP_NOP    = 3'b000; // No operation
22 (pass through)
23 localparam OP_ADD    = 3'b001; // Addition
24 localparam OP_SUB    = 3'b010; // Subtraction
25 localparam OP_MUL    = 3'b011; // Multiplication
26 localparam OP_SHIFT  = 3'b100; // Shift operation
27 localparam OP_FILTER = 3'b101; // Filter operation
28 localparam OP_INVERT = 3'b110; // Invert operation
29 localparam OP_COMPARE = 3'b111; // Compare operation
30
31 // State definitions
32 localparam IDLE      = 2'b00;
33 localparam WAIT_SECOND = 2'b01;
34 localparam PROCESSING = 2'b10;
35 localparam COMPLETE  = 2'b11;
36
37 // Internal registers
38 reg [1:0] state, next_state;
39 reg [7:0] operand_a;
40 reg [7:0] operand_b;
41 reg [2:0] current_opcode;
42 reg [15:0] result;
43 reg result_ready;
44 reg [3:0] flags;
45
46 // Flag bit positions
47 localparam FLAG_OVERFLOW = 3;
48 localparam FLAG_ZERO    = 2;
49 localparam FLAG_RESERVED = 1;
50 localparam FLAG_COMPLETE = 0;
51
52 // State machine
53 always @(posedge clk or negedge rst_n) begin
54     if (!rst_n) begin
55         state <= IDLE;
56         operand_a <= 8'h00;
57         operand_b <= 8'h00;
58         current_opcode <= 3'b000;
59         result <= 16'h0000;
60         result_ready <= 1'b0;
61         flags <= 4'b0000;
62     end else begin
63         state <= next_state;
64
65         case (state)
66             IDLE: begin
67                 result_ready <= 1'b0;
68                 flags <= 4'b0000;
69
70                 if (data_valid) begin
71                     operand_a <= data_in;
72                     current_opcode <= opcode;
73
74                     // For operations that don't
75                     need second operand
76                     if (opcode == OP_INVERT ||
77                         opcode == OP_NOP) begin
78                         next_state <= PROCESSING
79                     ;
80                     end else begin

```

```

71         next_state <=
WAIT_SECOND;
72         end
73     end else begin
74         next_state <= IDLE;
75     end
76 end
77
78 WAIT_SECOND: begin
79     if (data_valid) begin
80         operand_b <= data_in;
81         next_state <= PROCESSING;
82     end else begin
83         next_state <= WAIT_SECOND;
84     end
85 end
86
87 PROCESSING: begin
88     case (current_opcode)
89         OP_NOP: begin
90             // Simply pass through
91             operand_a
92                 result <= {8'h00,
93                 operand_a};
94                 flags[FLAG_COMPLETE] <=
95                 1'b1;
96             end
97             OP_ADD: begin
98                 // Addition with
99                 overflow detection
100                 {flags[FLAG_OVERFLOW],
101                 result[7:0]} <= operand_a + operand_b;
102                 result[15:8] <= 8'h00;
103                 flags[FLAG_COMPLETE] <=
104                 1'b1;
105                 flags[FLAG_ZERO] <= (
106                 operand_a + operand_b == 0);
107             end
108             OP_SUB: begin
109                 // Subtraction with
110                 underflow detection
111                 result[7:0] <= operand_a
112                 - operand_b;
113                 result[15:8] <= 8'h00;
114                 flags[FLAG_COMPLETE] <=
115                 1'b1;
116                 flags[FLAG_OVERFLOW] <=
117                 (operand_a < operand_b);
118                 flags[FLAG_ZERO] <= (
119                 operand_a == operand_b);
120             end
121             OP_MUL: begin
122                 // Multiplication with
123                 overflow detection
124                 result <= operand_a *
125                 operand_b;
126                 flags[FLAG_COMPLETE] <=
127                 1'b1;
128                 flags[FLAG_OVERFLOW] <=
129                 ((operand_a * operand_b) > 8'hFF);
130                 flags[FLAG_ZERO] <= ((
131                 operand_a * operand_b) == 0);
132             end
133             OP_SHIFT: begin
134                 // Shift operation - bit
135                 3 of operand_b determines direction
136                 if (operand_b[3]) begin
137                     // Right shift
138                     result[7:0] <=
139                     operand_a >> (operand_b[2:0]);
140                 end else begin
141                     // Left shift
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

182         next_state <= IDLE;
183     end else begin
184         // Hold the result until new
185         operation starts
186         next_state <= COMPLETE;
187     end
188
189     default: begin
190         next_state <= IDLE;
191     end
192 endcase
193 end
194
195 // Output assignments
196 assign data_out = result;
197 assign data_ready = result_ready;
198 assign status_flags = flags;
199
200 endmodule

```

Listing 1. Time-Based Processor Module

APPENDIX B TEST BENCH FOR TIME-BASED PROCESSOR

```

1 `timescale 1ns / 1ps
2
3 module time_based_processor_simple_tb;
4
5     // Parameters
6     parameter CLK_PERIOD = 10; // 10ns (100MHz)
7     clock period
8
9     // DUT Signals
10    reg      clk;
11    reg      rst_n;
12    reg [7:0] data_in;
13    reg [2:0] opcode;
14    reg      data_valid;
15    wire [15:0] data_out;
16    wire      data_ready;
17    wire [3:0] status_flags;
18
19    // Instantiate the DUT
20    time_based_processor DUT (
21        .clk(clk),
22        .rst_n(rst_n),
23        .data_in(data_in),
24        .opcode(opcode),
25        .data_valid(data_valid),
26        .data_out(data_out),
27        .data_ready(data_ready),
28        .status_flags(status_flags)
29    );
30
31    // Clock generator
32    initial begin
33        clk = 0;
34        forever # (CLK_PERIOD/2) clk = ~clk;
35    end
36
37    // Task for applying inputs
38    task apply_operation;
39        input [2:0] op;
40        input [7:0] input_a;
41        input [7:0] input_b;
42        input      needs_second_input;
43
44        begin
45            // First operand and opcode
46            @(posedge clk);
47            data_in = input_a;
48            opcode = op;

```

```

48            data_valid = 1'b1;
49
50            @(posedge clk);
51            data_valid = 1'b0;
52
53            // If operation needs second operand
54            if (needs_second_input) begin
55                // Wait a bit before sending second
56                operand
57                repeat(3) @(posedge clk);
58
59                // Second operand
60                @(posedge clk);
61                data_in = input_b;
62                data_valid = 1'b1;
63
64                @(posedge clk);
65                data_valid = 1'b0;
66            end
67
68            // Wait for result to be ready
69            wait(data_ready);
70
71            // Display result
72            $display("Operation: %d, Input A: %h,
73                Input B: %h, Result: %h, Flags: %b",
74                op, input_a, input_b, data_out,
75                status_flags);
76
77            // Wait a bit before next operation
78            repeat(5) @(posedge clk);
79        end
80    endtask
81
82    // Test sequence
83    initial begin
84        // Initialize signals
85        clk = 0;
86        rst_n = 0;
87        data_in = 8'h00;
88        opcode = 3'b000;
89        data_valid = 0;
90
91        // Apply reset
92        #20 rst_n = 1;
93
94        // Wait for a few clock cycles
95        repeat(5) @(posedge clk);
96
97        // Apply different operations
98
99        // Test NOP (0)
100        apply_operation(3'b000, 8'h55, 8'h00, 0);
101
102        // Test ADD (1) - Regular
103        apply_operation(3'b001, 8'h23, 8'h45, 1);
104
105        // Test ADD (1) - With overflow
106        apply_operation(3'b001, 8'hFF, 8'h01, 1);
107
108        // Test SUB (2) - Regular
109        apply_operation(3'b010, 8'h45, 8'h23, 1);
110
111        // Test SUB (2) - With underflow
112        apply_operation(3'b010, 8'h23, 8'h45, 1);
113
114        // Test MUL (3) - Regular
115        apply_operation(3'b011, 8'h05, 8'h04, 1);
116
117        // Test MUL (3) - With overflow
118        apply_operation(3'b011, 8'h23, 8'h45, 1);
119
120        // Test SHIFT (4) - Left shift
121        apply_operation(3'b100, 8'h05, 8'h01, 1);
122
123        // Test SHIFT (4) - Right shift
124        apply_operation(3'b100, 8'h08, 8'h09, 1);

```

```

122
123 // Test FILTER (5)
124 apply_operation(3'b101, 8'hAA, 8'h55, 1);
125
126 // Test INVERT (6)
127 apply_operation(3'b110, 8'hAA, 8'h00, 0);
128
129 // Test COMPARE (7) - A > B
130 apply_operation(3'b111, 8'h23, 8'h22, 1);
131
132 // Test COMPARE (7) - A < B
133 apply_operation(3'b111, 8'h22, 8'h23, 1);
134
135 // Test COMPARE (7) - A = B
136 apply_operation(3'b111, 8'h23, 8'h23, 1);
137
138 // End simulation
139 #100 $finish;
140 end
141
142 endmodule

```

Listing 2. Test Bench Code