Τμήμα Μηχανικών Η/Υ και Πληροφορικής
Τομέας Εφαρμογών & Θεμελιώσεων των Υπολογιστών
Πανεπιστήμιο Πατρών

Παράλληλοι Αλγόριθμοι
Μηχανή Διαφορών Babbage

Διδάσκων:
Εύη Παπαϊωάννου

Αλέξανδρος-Οδυσσέας Φαρμάκης
Ε΄ Έτος Προπτυχιακών Σπουδών, ΑΜ1072551

Πάτρα, Εαρινό Εξάμηνο, 2023-24

Περιεχόμενα:

Κεφάλαιο 1: Η Μηχανή Διαφορών του Babbage

1.1 Ιστορική αναδρομή

Ο Charles Babbage, Βρετανός μαθηματικός, είναι διάσημος για την πρωτοποριακή του δουλειά στον τομέα των υπολογιστικών μηχανημάτων κατά τον 19ο αιώνα. Η προσπάθειά του να μηχανοποιήσει πολύπλοκους μαθηματικούς υπολογισμούς οδήγησε στη σύλληψη και την ανάπτυξη της Μηχανής Διαφορών το 1819 και την είχε ολοκληρώσει μέχρι το 1822 (Difference Engine 0). Ανακοίνωσε την εφεύρεσή του στις 14 Ιουνίου 1822, σε έγγραφο προς τη Βασιλική Αστρονομική Εταιρεία, με τίτλο "Note on the application of machinery to the computation of astronomical and mathematical tables". Η Μηχανή Διαφορών σχεδιάστηκε για να αυτοματοποιεί τον υπολογισμό πολυωνυμικών συναρτήσεων, που αποτελούσε κουραστική και επιρρεπή σε σφάλματα χειροκίνητη διαδικασία, αλλά αποτελούσε ιδιαίτερα διαδεδομένη στην παραγωγή μαθηματικών πινάκων ζωτικής σημασίας για διάφορες επιστημονικές και μηχανικές προσπάθειες της εποχής.

Η Μηχανή Διαφορών αντιπροσώπευε ένα τεράστιο άλμα στην υπολογιστική τεχνολογία της εποχής. Βασίστηκε στην αρχή των πεπερασμένων διαφορών του Νεύτωνα. Το σχέδιο του Babbage ενσωμάτωσε περίπλοκα μηχανικά εξαρτήματα για την αυτόματη εκτέλεση αυτών των υπολογισμών, εξαλείφοντας την ανάγκη για ανθρώπινη παρέμβαση, για την οποία έλαβε σημαντική οικονομική υποστήριξη από τη βρετανική κυβέρνηση το 1823, η οποία έδωσε στον Babbage £1500 (που σήμερα, εν έτει 2024, είναι περίπου 15500€) για να ξεκινήσει το έργο.

Ενώ το φιλόδοξο όραμα του Babbage για την πλήρη κατασκευή της Μηχανής Διαφορών δεν υλοποιήθηκε πλήρως κατά τη διάρκεια της ζωής του λόγω τεχνικών πολυπλοκοτήτων και περιορισμών χρηματοδότησης, το έργο του έθεσε τα θεμέλια για τη μετέπειτα ανάπτυξη των υπολογιστικών μηχανών. Το εννοιολογικό πλαίσιο και οι αρχές μηχανικής πίσω από τη Μηχανή Διαφορών παρείχαν ανεκτίμητες γνώσεις για τις μελλοντικές γενιές εφευρετών και μηχανικών, διαμορφώνοντας τελικά την τροχιά της σύγχρονης πληροφορικής.

1.2 Ανάλυση του αλγορίθμου

Η Μηχανή Διαφορών του Babbage βασίζεται στη μέθοδο των διαφορών του Νεύτωνα, η οποία αποτελεί αναδρομική διαδικασία υπολογισμού πολυωνύμων και παρακάμπτει έτσι την ανάγκη πολλαπλασιασμού. Δεδομένου ενός πολυωνύμου $f(n)$, υπολογίζεται η διαφορά $f(n) - f(n\text{-}1)$ και η τιμή $f(0)$. Το αποτέλεσμα αυτής της διαφοράς ονομάζεται $g(n)$, και με παρόμοιο τρόπο υπολογίζεται η διαφορά $g(n) - g(n\text{-}1)$ και η τιμή $g(1)$. Κάθε νέο τέτοιο πολυώνυμο προσδιορίζεται στο αμέσως επόμενο ακέραιο που ορίζεται από την εκάστοτε διαφορά, μέχρις ότου φτάσουμε σε σταθερό πολυώνυμο. Στα πλαίσια αυτής της εργασίας, θα δοθεί κώδικας που υλοποιεί οποιοδήποτε πολυώνυμο μέχρι και 5ου βαθμού χρησιμοποιώντας αυτή τη μέθοδο.

Για την καλύτερη κατανόηση αυτού, ας δούμε ένα απλουστευμένο παράδειγμα. Έστω το πολυώνυμο $f(n) = 2n^2 + 3n + 5$. Η διαφορά του, βάσει του προηγούμενο, είναι $f(n) - f(n\text{-}1) = 4n + 1 = g(n)$. Αναδρομικά τότε, για n ≥ 0, το πολυώνυμο $f(n)$ μπορεί να εκφραστεί ως:

$$f(n)=\begin{cases} 5 & , \ n=0 \\ f(n-1)+g(n), & n\geq 0 \end{cases}$$

Αντίστοιχα, μπορούμε να βρούμε τη διαφορά $g(n) - g(n\text{-}1) = 4 = h(n)$, οπότε αναδρομικά έχουμε:

$$g(n)=\begin{cases} 0 & , \ n<1 \\ 5 & , \ n=1 \\ g(n-1)+h(n), & n\geq 1 \end{cases}$$

Τέλος, έχουμε:

$$h(n)=\begin{cases} 0 \ , & n<2 \\ 4 \ , & n\geq 2 \end{cases}$$

1.3 Αναπαράσταση με ψευδοκώδικα

Μια αναπαράσταση του αλγόριθμου που μπορεί να γίνει με ψευδοκώδικα είναι ο εξής:

```
integer input a, b, c, d, f, g;      // Polynomial coefficients (not using e as to avoid confusion)
positive integer input n;            // Variable we want to evaluate the polynomial at
integer output poly_out;             // Polynomial output for given n
integer i;                           // Recursion tracker
integer u, v, w, x, y, z;            // Piece-wise polynomial values used for recursion

parallel for (i = 0; i <= n; i++) begin
     call u = compute_u(a, b, c, d, f, g, i);
     call v = compute_v(a, b, c, d, f, i);
     call w = compute_w(a, b, c, d, i);
     call x = compute_x(a, b, c, i);
     call y = compute_y(a, b, i);
     call z = compute_z(a, i);
end

// When the parallel block is done, assign the value of u to the output poly_out
if (i = n) poly_out = u;

function compute_u(a, b, c, d, f, g, i) begin
     if (n == 0)
          return g;
     else
          return compute_u(a, b, c, d, f, g, i-1) + compute_v(a, b, c, d, f, i);
end
```

```
function compute_v(a, b, c, d, f, i) begin
     if (i < 1)
          return 0;
     else if (i == 1)
          return (5*a + (-10*a+4*b) + (10*a-6*b+3*c) + (-5*a+4*b-3*c+2*d) + (a-b+c-d+f));
     else
          return compute_v(a, b, c, d, f, i-1) + compute_w(a, b, c, d, i);
end

function compute_w(a, b, c, d, i) begin
     if (i < 2)
          return 0;
     else if (i == 2)
          return (20*a*8 + (-60*a+12*b)*4 + (70*a-24*b+6*c)*2 + (-30*a+14*b-6*c+2*d));
     else
          return compute_w(a, b, c, d, i-1) + compute_x(a, b, c, i);
end

function compute_x(a, b, c, i) begin
     if (i < 3)
          return 0;
     else if (i == 3)
          return (40*a*9 + (-120*a+24*b)*3 + (150*a-36*b+6*c));
     else
          return compute_x(a, b, c, i-1) + compute_y(a, b, i);
end

function compute_y(a, b, i) begin
     if (i < 4)
          return 0;
     else if (i == 4)
          return (120*a*4 + (-240*a+24*b));
     else
          return compute_y(a, b, i-1) + compute_z(a, i);
end

function compute_z(a, i) begin
     if (i < 5)
          return 0;
     else
          return 120 * a;
end
```

Κεφάλαιο 2: Υλοποίηση μέσω της Verilog

2.1 Επεξεργαστικό μοντέλο του αλγορίθμου

Η Μηχανή Διαφορών του Babbage μπορεί να μοντελοποιηθεί ως Μηχανή Πεπερασμένων Καταστάσεων με Διαδρομή Δεδομένων (Finite State Machine with Datapath, FSMD), όπου η λειτουργία της μηχανής διέπεται από ένα πεπερασμένο σύνολο καταστάσεων. Σε αυτό το μοντέλο, τα μηχανικά εξαρτήματα της Μηχανή Διαφορών, όπως τα γρανάζια και οι μοχλοί, αντιπροσωπεύουν τα στοιχεία του datapath, τα οποία είναι υπεύθυνα για την εκτέλεση των αριθμητικών πράξεων, ενώ η μονάδα ελέγχου, το οποίο αποτελεί το FSM, στέλνει και λαμβάνει σήματα ελέγχου από και προς το datapath, και χειρίζεται την αλληλουχία των λειτουργιών με βάση τη τωρινή κατάσταση και τυχόν σημάτων ελέγχου. Κάθε κατάσταση στο FSMD αντιστοιχεί σε συγκεκριμένο στάδιο της διαδικασίας υπολογισμού. Με την αναπαράσταση της Μηχανής Διαφορών ως FSMD, η συμπεριφορά και η λειτουργικότητά του μπορούν να αναλυθούν ευκολότερα και το κάθε εξάρτημά του μπορεί να βελτιστοποιηθεί με τεχνικές τύπου “διαίρει και βασίλευε”, δηλαδή αντί να σχεδιάζουμε ένα μεγάλο σύστημα, σχεδιάζουμε τα μικρότερα εξαρτήματα ξεχωριστά και μετά τα “κουμπώνουμε” στο τέλος.

2.2 Σχεδιαστικά ζητήματα

Επομένως, προχωράμε στη διάρθρωση του αλγορίθμου αυτού ως εξειδικευμένο ψηφιακό σύστημα. Η ιδέα των παράλληλων αλγορίθμων γενικότερα δανείζεται πολύ καλά στο υλικό, μιας και όλο είναι εκ φύσεως παράλληλο επειδή κάθε μονοπάτι συνδυαστικής λογικής βρίσκεται ενδιάμεσα σε καταχωρητές που λαμβάνουν το ίδιο σήμα ρολογιού. Επίσης, η ιδέα χρήσης ενός FSM για τον έλεγχο ψηφιακών συστημάτων είναι μια αρκετά διαδεδομένη ιδέα που βλέπει εφαρμογές παντού, από τα απλούστερα κυκλώματα μέχρι και σε ελεγκτές ενσωματωμένων συστημάτων, επεξεργαστών γενικού σκοπού και άλλα.

Ας δούμε τώρα το datapath. Αφού έχουμε μια πολυωνυμική ακολουθία τύπου $u(n) = an^5 + bn^4 + cn^3 + dn^2 + fn + g$, η ανάλυση που κάναμε νωρίτερα θα μας δώσει 6 επιμέρους πολυώνυμα, εκ των οποίων το τελευταίο είναι το σταθερό πολυώνυμο. Επομένως, θα χρησιμοποιήσουμε 7 - 1 = 6 καταχωρητές - συσσωρευτές για κάθε τέτοιο πολυώνυμο στο datapath.

Από άποψη απόδοσης, ο αντίστοιχος ακολουθιακός αλγόριθμος έχει πολυπλοκότητα $O(m \cdot n)$, όπου m ο βαθμός του πολυωνύμου + 1 (εδώ 5 + 1 = 6) και n ο αριθμός στο οποίο θέλουμε να υπολογίσουμε το πολυώνυμο, και υποθέτοντας ότι έχουμε m αναγνώσεις στη μνήμη για την αρχικοποίηση των αρχικών τιμών του κάθε επιμέρους πολυωνύμου που χρησιμοποιείται έχουμε χρόνο εκτέλεσης $(m + 1) \cdot n$. Ο παράλληλος αλγόριθμος της Μηχανής Διαφορών κάνει χρήση m “επεξεργαστών” (οι συσσωρευτές) και έχει πολυπλοκότητα $O(n)$, και λόγω της συγκεκριμένης δομής του FSM ελέγχου έχουμε χρόνο εκτέλεσης 2 (pre-calc states) + n + 1 (done state) = n + 3.

Επομένως, εξάγουμε τις εξής μετρικές:

$$\text{Speed-Up} = \frac{m \cdot (n+1)}{n+3} \simeq m \text{ , όσο n}\uparrow.$$
$$W = (n+3) \cdot m \text{ .}$$
$$E = \frac{m \cdot (n+1)}{(n+3) \cdot m} = \frac{n+1}{n+3} \simeq 1 \text{ , όσο n}\uparrow,$$

οι οποίες είναι αρκετά λογικές, αφού το συγκεκριμένο σύστημα είναι ειδικού σκοπού και σχεδιασμένο ακριβώς ώστε να εκτελεί αυτό τον αλγόριθμο όσο το πιο βέλτιστα γίνεται.

2.3 Κώδικας Verilog της Μηχανής Διαφορών

Στο τμήμα 2.1 αναφέραμε ότι η Μηχανή Διαφορών του Babbage μπορεί να μοντελοποιηθεί ως FSMD. Για την υλοποίηση αυτής της ιδέας, θα χρησιμοποιήσουμε μια γλώσσα περιγραφής υλικού (hardware description language, HDL), και στη προκειμένη περίπτωση είναι η Verilog. Με αυτή τη λογική, έχουν συγγραφεί τρία (3) κομμάτια Verilog, τα οποία είναι το `babbage_fsm.v`, που αποτελεί τη λογική του FSM ελέγχου, το `babbage_datapath.v`, που υλοποιεί τη διαδρομή δεδομένων που εκτελεί τις ζητούμενες πράξεις, και το `babbage_top.v` που ενώνει αυτά τα δύο στοιχεία μεταξύ τους σε μία ενιαία λειτουργική μονάδα.

Η μηχανή πεπερασμένων καταστάσεων `babbage_fsm.v`:

```
module babbage_fsm (reset, clk, start, done, ready, precalc_enable_1, precalc_enable_2, calc_enable,
done_tick);
     localparam IDLE = 3'b000, PRECALC1 = 3'b001, PRECALC2 = 3'b011, CALC = 3'b010, DONE = 3'b110,
BUFFER = 3'b100;

     input reset, clk, start, done;
     output reg ready, precalc_enable_1, precalc_enable_2, calc_enable, done_tick;

     reg [2:0] state, next;

     // Sequential always block for the state register
     always @(posedge clk or posedge reset) begin
          if (reset) state <= 3'b000;
          else state <= next;
     end
```

```
    // Combinational always block, so we'll use
    // blocking statements here
    always @(*) begin
        ready = 1'b0;
        next = 2'bx;
        case (state)
            IDLE:
            begin
                calc_enable = 1'b0;
                precalc_enable_1 = 1'b0;
                precalc_enable_2 = 1'b0;
                done_tick = 1'b0;
                if (start) next = PRECALC1;
                else begin
                    ready = 1'b1;
                    next = IDLE;
                end
            end
            PRECALC1:
            begin
                precalc_enable_1 = 1'b1;
                precalc_enable_2 = 1'b0;
                next = PRECALC2;
            end
            PRECALC2:
            begin
                precalc_enable_1 = 1'b0;
                precalc_enable_2 = 1'b1;
                next = CALC;
            end
            CALC:
            begin
                precalc_enable_2 = 1'b0;
                calc_enable = 1'b1;
                if (done) next = DONE;
                else next = CALC;
            end
            DONE:
            begin
                done_tick = 1'b1;
                next = BUFFER;
            end
            BUFFER:
            begin
                done_tick = 1'b1;
                next = IDLE;
            end
        endcase
    end

endmodule
```

Η διαδρομή δεδομένων babbage_datapath.v:

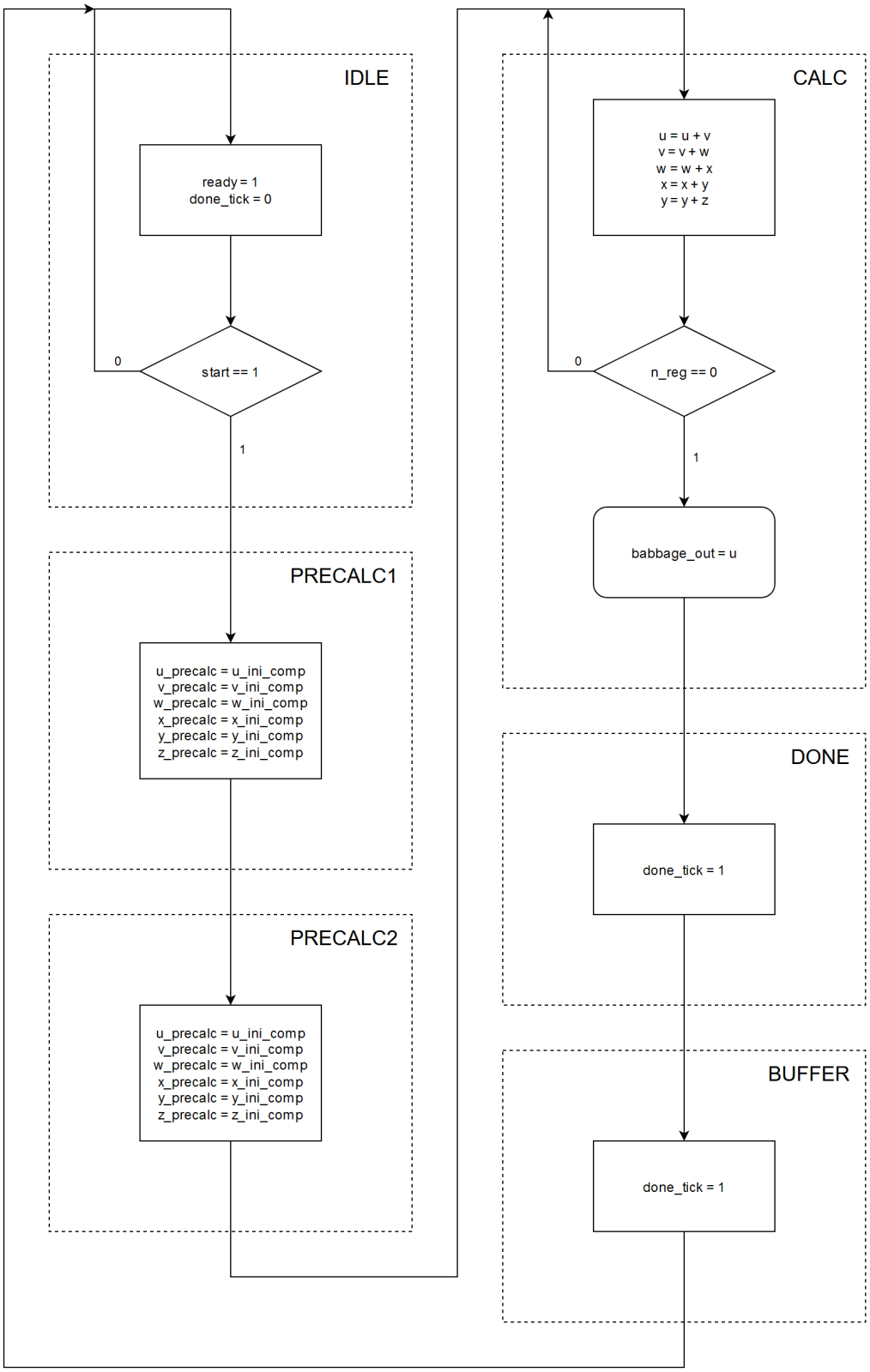
```
module babbage_datapath (reset, clk, ready, precalc_enable_1, precalc_enable_2, calc_enable, a, b, c,
d, f, g, n, babbage_out, done);

     // Control I/O
     input reset, clk, ready, precalc_enable_1, precalc_enable_2, calc_enable;
     output done;

     // Polynomial coefficients
     input signed [1:0] a;
     input signed [2:0] b;
     input signed [3:0] c;
     input signed [3:0] d;
     input signed [5:0] f;
     input signed [9:0] g;

     // Evaluation number
     input [6:0] n;

     // Output polynomial evaluation
     output reg signed [31:0] babbage_out;

     // Piece-wise polynomial and intermediate registers
     reg signed [31:0] u, v, w, x, y, z;
     reg signed [31:0] u_precalc, v_precalc, w_precalc, x_precalc, y_precalc, z_precalc;

     // Evaluation number register (which will decrement to 0)
     reg [6:0] n_reg;

     // Initial computation for piece-wise polynomials based on polynomial coefficients
     wire signed [31:0] u_ini_comp, v_ini_comp, w_ini_comp, x_ini_comp, y_ini_comp, z_ini_comp;

     // Initial evaluations for u, v, w, x, y, z piece-wise polynomials.
     // Multiplications and such with given numbers are optimized during
     // synthesis, so no need to write this otherwise.
     // For example, you won't get a multiplier to get 120*a, you'll get an adder
     // scheme that takes 128*a and subtracts 8*a (which are shift multiples) from it.
     // It will be the critical path as is, so here it is worth pipelining!
     // However, for simplicity's sake, we will keep these as is as to highlight
     // the algorithm behind the Babbage Difference Engine.
     assign u_ini_comp = g;
     assign v_ini_comp = 5*a + (-10*a+4*b) + (10*a-6*b+3*c) + (-5*a+4*b-3*c+2*d) + (a-b+c-d+f);
     assign w_ini_comp = 20*a*8 + (-60*a+12*b)*4 + (70*a-24*b+6*c)*2 + (-30*a+14*b-6*c+2*d);
     assign x_ini_comp = 40*a*9 + (-120*a+24*b)*3 + (150*a-36*b+6*c);
     assign y_ini_comp = 120*a*4 + (-240*a+24*b);
     assign z_ini_comp = 120*a;

     // Save initial computations to precalc registers
     always @(posedge reset or posedge clk) begin
          if (reset) begin
               u_precalc <= 0;
               v_precalc <= 0;
               w_precalc <= 0;
               x_precalc <= 0;
               y_precalc <= 0;
               z_precalc <= 0;
               n_reg <= 0;
          end
```

```
        else if (precalc_enable_1) begin
            u_precalc <= u_ini_comp;
            v_precalc <= v_ini_comp;
            w_precalc <= w_ini_comp;
            x_precalc <= x_ini_comp;
            y_precalc <= y_ini_comp;
            z_precalc <= z_ini_comp;
            n_reg <= n;
        end
    end

    // Assignments made to accumulator registers for piece-wise polynomials
    // Babbage output and done control signal to FSM is below as well
    always @(posedge reset or posedge clk) begin
        if (reset) begin
            u <= 0;
            v <= 0;
            w <= 0;
            x <= 0;
            y <= 0;
            z <= 0;
            babbage_out <= 0;
        end
        else if (precalc_enable_1) begin
            u <= 0;
            v <= 0;
            w <= 0;
            x <= 0;
            y <= 0;
            z <= 0;
            babbage_out <= 0;
        end
        else if (precalc_enable_2) begin
            u <= u_precalc;
            v <= v_precalc;
            w <= w_precalc;
            x <= x_precalc;
            y <= y_precalc;
            z <= z_precalc;
        end
        else if (calc_enable) begin
            if (n_reg == 0) begin
                n_reg <= 0;
                babbage_out <= u;
            end
            else begin
                n_reg <= n_reg - 1;
                u <= u + v;
                v <= v + w;
                w <= w + x;
                x <= x + y;
                y <= y + z;
            end
        end
    end

    assign done = calc_enable & (n_reg == 0) & ~ready;

endmodule
```

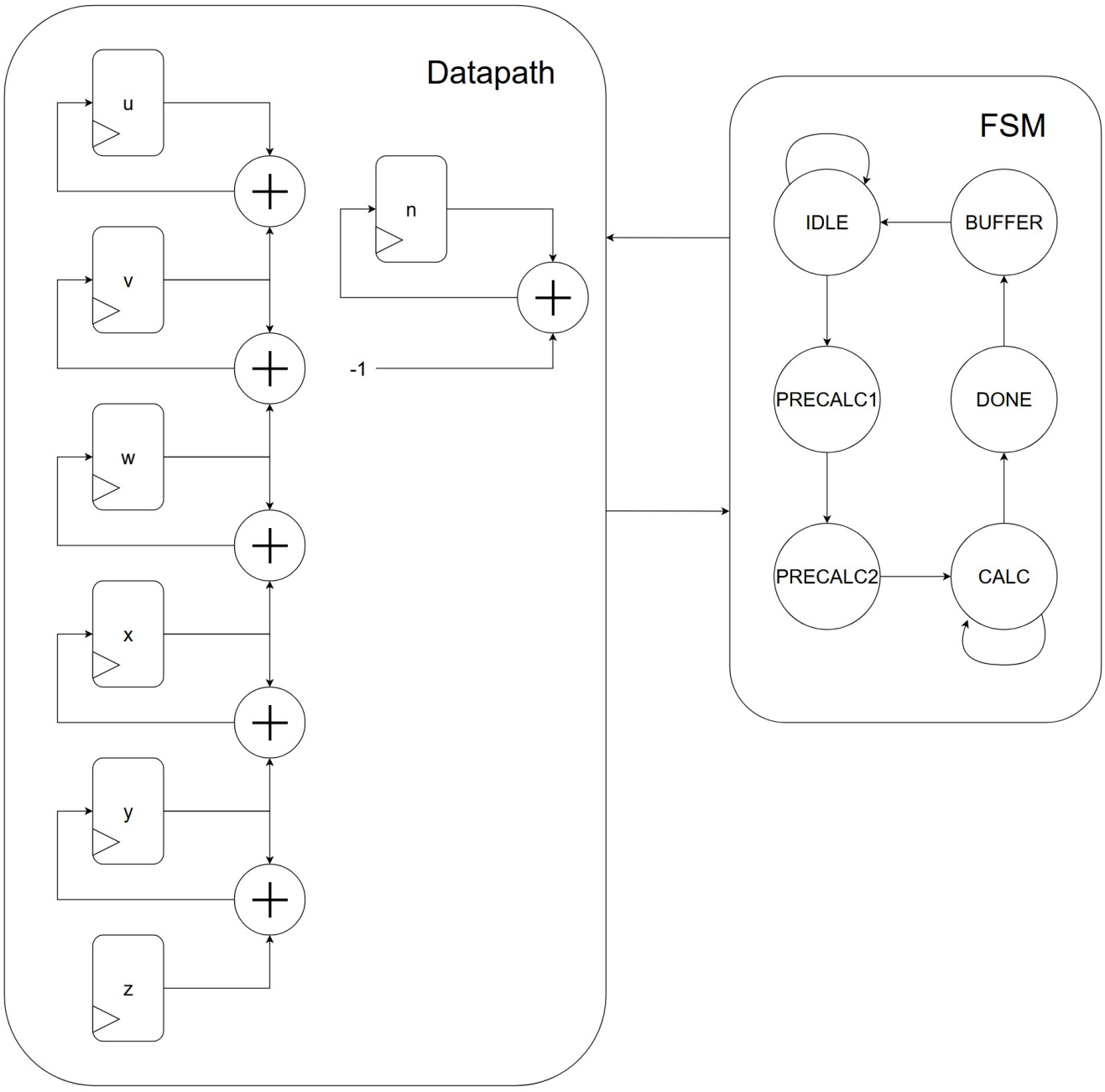
Το top module babbage_top.v:

```
module babbage_top (reset, clk, start, a, b, c, d, f, g, n, ready, babbage_out, done_tick);

    // Control I/O
    input reset, clk, start;

    // Polynomial coefficients
    input signed [1:0] a;
    input signed [2:0] b;
    input signed [3:0] c, d;
    input signed [5:0] f;
    input signed [9:0] g;

    // Evaluation number
    input [6:0] n;

    output signed [31:0] babbage_out;
    output ready, done_tick;

    wire precalc_enable_1, precalc_enable_2, calc_enable, done;

    babbage_datapath datapath (reset, clk, ready, precalc_enable_1, precalc_enable_2, calc_enable, a,
b, c, d, f, g, n, babbage_out, done);
    babbage_fsm control_fsm (reset, clk, start, done, ready, precalc_enable_1, precalc_enable_2,
calc_enable, done_tick);

endmodule
```

Επισημαίνεται ότι η τεχνική σχεδιασμού του παραπάνω FSMD βασίστηκε στη μέθοδο σχεδίασης Αλγοριθμικών Μηχανών Καταστάσεων (Algorithmic State Machine, ASM), η οποία υιοθετεί διαγράμματα ροής παρόμοια με αυτά που χρησιμοποιούνται για την αναπαράσταση αλγορίθμων, και τα block που σχηματίζονται υποδεικνύουν μία κατάσταση, που αντιστοιχίζεται σε έναν κύκλο ρολογιού. Το διάγραμμα του αλγορίθμου φαίνεται στην επόμενη σελίδα.


IDLE
ready = 1
done_tick = 0
start == 1
0
1
PRECALC1
u_precalc = u_ini_comp
v_precalc = v_ini_comp
w_precalc = w_ini_comp
x_precalc = x_ini_comp
y_precalc = y_ini_comp
z_precalc = z_ini_comp
PRECALC2
u_precalc = u_ini_comp
v_precalc = v_ini_comp
w_precalc = w_ini_comp
x_precalc = x_ini_comp
y_precalc = y_ini_comp
z_precalc = z_ini_comp
CALC
u = u + v
v = v + w
w = w + x
x = x + y
y = y + z
n_reg == 0
0
1
babbage_out = u
DONE
done_tick = 1
BUFFER
done_tick = 1

Παρακάτω φαίνεται ένα απλουστευμένο διάγραμμα του FSM και του Datapath που απαρτίζουν το τελικό FSMD σχεδιασμό, βάσει και του διαγράμματος ASM που παρουσιάστηκε προηγουμένως.



2.4 Μονάδα δοκιμής (testbench)

Τα testbenches δημιουργούνται για κυκλώματα Verilog (και οποιασδήποτε άλλης γλώσσας περιγραφής υλικού) για τη συστηματική επαλήθευση της λειτουργικότητας και της απόδοσης ενός σχεδιασμού, εισάγοντας διάφορα διανύσματα εισόδου και παρατηρώντας τις αντίστοιχες εξόδους, συγκρίνοντάς τις με κάποιες αναμενόμενες τιμές, που βοηθάει περαιτέρω στη διόρθωση τυχόν λαθών σχεδιασμού. Επιτρέπουν έτσι στους σχεδιαστές να επικυρώσουν τη συμπεριφορά των κυκλωμάτων τους υπό διαφορετικές συνθήκες, διασφαλίζοντας την αξιοπιστία, την ορθότητα και την τήρηση των προδιαγραφών σχεδίασης πριν από την πραγματική υλοποίηση στο υλικό.

Στη συγκεκριμένη περίπτωση, έχουμε 68.719.476.736 συνδυασμούς εισόδων, όπου σαφώς δε βγάζει νόημα να κάνουμε δοκιμές με το χέρι, πόσο μάλλον αυτές οι απειροελάχιστες συγκριτικά δοκιμές να είναι αρκετές για την αξιόπιστη δοκιμή αυτού του κυκλώματος.

Παρακάτω δίνεται υλοποίηση ενός τέτοιου testbench, που ελέγχει ένα πλήθος τυχαία παραγόμενων διανυσμάτων που αποτελεί ένα σημαντικά μεγάλο υποσύνολο όλως των διανυσμάτων δοκιμής. Βέβαια, λόγω του απαραίτητου χρόνου εξομοίωσης, αυτό το σύνολο μπορεί να μικρύνει εφόσον χρειαστεί, όπου προφανώς το διάστημα εμπιστοσύνης για την αποτελεσματική δοκιμή του κυκλώματος μικραίνει αναλόγως.

Το testbench `babbage_tb_random_vectors.v`:

```
module babbage_tb_random_vectors;

    // Control Inputs
    reg reset, clk, start;

    // Polynomial coefficients
    reg signed [1:0] a;
    reg signed [2:0] b;
    reg signed [3:0] c, d;
    reg signed [5:0] f;
    reg signed [9:0] g;

    // Evaluation number
    reg [6:0] n;

    wire signed [31:0] babbage_out;
    wire ready, done_tick;

    // Testbench-related paramters, variables and files
    parameter period = 20;
    parameter fileout = "babbage_results.txt";
    reg signed [31:0] expected_babbage;
    integer i, file;
    reg error;
    integer total_errors = 0;
    integer repeats = 200000;
```

```
    // Sign-extended polynomial coefficients
    reg signed [31:0] a_ext, b_ext, c_ext, d_ext, f_ext, g_ext;

    // Sign-extended evaluation number
    reg signed [31:0] n_ext;

    // Design Under Test (DUT)
    babbage_top DUT (reset, clk, start, a, b, c, d, f, g, n, ready, babbage_out, done_tick);

    // Clock generation
    initial begin
        clk = 0;
        forever begin
            #(0.5*period) clk = ~clk;
        end
    end

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars;
        file = $fopen(fileout, "w");
        $display("Beginning test ...\n");
        $fwrite(file, "Beginning test...\n\n");
        i <= 0;
        error <= 1'b0;
        start <= 0;
        reset <= 0;
        #(4*period + 1); // Small additional delay to show reset is truly asynchronous!
        reset <= 1;
        expected_babbage <= 0;
        #(0.25*period);
        if (expected_babbage != babbage_out) begin
            $display("$0t, Error in reset test");
            $fwrite(file, "$0t, Error in reset test\n");
            total_errors <= total_ errors + 1;
            error <= 1'bx;
        end
        else begin
            error <= 1'b0;
            $display("Reset test passed!\n");
            $fwrite(file, "Reset test passed!\n");
        end
        reset <= 0;
        $display("Iterations tests following...");
        $fwrite(file, "Iterations tests following ...\n\n");
        $display("Testing procedure is as follows:");
        $fwrite(file, "Testing procedure is as follows:\n");
        $display("Evaluate a polynomial u(n) = a n^5 + b n^4 + c n^3 + d n^2 + f n + g,");
        $fwrite(file, "Evaluate a polynomial u(n) = a·n^5 + b·n^4 + c·n^3 + d·n^2 + f·n + g,\n");
        $display("once using the Babbage Difference Engine (DUT), and once using the multiplication
method as the unit test");
        $fwrite(file, "once using the Babbage Difference Engine (DUT), and once using the
multiplication method as the unit test\n");
        $display("Inputs: n (integer larger than or equal to 0)");
        $fwrite(file, "Inputs: n (integer larger than or equal to 0)\n");
        $display("Coefficients: a, b, c, d, f, g (all signed integers)");
        $fwrite(file, "Coefficients: a, b, c, d, f, g (all signed integers)\n");
        $display("Bit lengths for all these can be changed, but for a proper 32-bit integer result");
        $fwrite(file, "Bit lengths for all these can be changed, but for a proper 32-bit integer
result\n");
```

```
          $display("we have set a -> 2-bit, b -> 3-bit, c -> 4-bit, d -> 4-bit, f -> 6-bit, g -> 10
bit, n -> 7-bit");
          $fwrite(file, "we have set a -> 2-bit, b -> 3-bit, c -> 4-bit, d -> 4-bit, f -> 6-bit, g ->
10 bit, n -> 7-bit\n");
          $display("and then sign extended so that there is no chance of overflow in the result for any
of these values.");
          $fwrite(file, "and then sign extended so that there is no chance of overflow in the result
for any of these values.\n");
          $display("Modifications are possible of course, but do so accordingly for accurate results
for any of these values.\n");
          $fwrite(file, "Modifications are possible of course, but do so accordingly for accurate
results for any of these values.\n\n");
          repeat (repeats) begin
               i <= i + 1;
               #(period);
               a <= $random % 4;
               b <= $random % 8;
               c <= $random % 16;
               d <= $random % 16;
               f <= $random % 64;
               g <= $random % 1024;
               n <= $random % 128;
               #(period);
               a_ext <= {{30{a[1]}}, a};   // Sign-extend a to 32-bits
               b_ext <= {{29{b[2]}}, b};   // Sign-extend b to 32-bits
               c_ext <= {{28{c[3]}}, c};   // Sign-extend c to 32-bits
               d_ext <= {{28{d[3]}}, d};   // Sign-extend d to 32-bits
               f_ext <= {{26{f[5]}}, f};   // Sign-extend f to 32-bits
               g_ext <= {{22{g[9]}}, g};   // Sign-extend g to 32-bits
               n_ext <= {{22{1'b0}}, b};   // Extend n to 32-bits
               #(period);
               start <= 1;
               #(period);
               start <= 0;
               #(period);
               expected_babbage <= $signed(a_ext * (n_ext ** 5) + b_ext * (n_ext ** 4) + c_ext * (n_ext
** 3) + d_ext * (n_ext ** 2) + f_ext * n_ext + g_ext);
               $fwrite(file, "=============================================\n");
               $fwrite(file, "Iteration %d:\n", i);
               $fwrite(file, "a = %d\n", a);
               $fwrite(file, "b = %d\n", b);
               $fwrite(file, "c = %d\n", c);
               $fwrite(file, "d = %d\n", d);
               $fwrite(file, "f = %d\n", f);
               $fwrite(file, "g = %d\n", g);
               $fwrite(file, "n = %d\n", n);
               wait (done_tick == 1) begin
                    #(3*period);
                    if (expected_babbage != babbage_out) begin
                         $fwrite(file, "Error in test babbage_out = %d, expected babbage_out=%d\n",
babbage_out, expected_babbage);
                         $display("$0t, Error in test babbage_out = %d ", babbage_out);
                         $display("$0t, expected babbage_out = %d", expected_babbage);
                         total_errors <= total_ errors + 1;
                         error <= 1'bx;
                    end
                    else begin
                         $fwrite(file, "Expected: %d\n", expected_babbage);
                         $fwrite(file, "Test babbage_out = %d passed!\n", babbage_out);
                         error <= 1'b0;
                    end
```

```
                end
                $fwrite(file, "=============================================\n\n");
            end
            $display("Finished with %0d errors", total_errors);
            $fwrite(file, "\nFinished with %0d errors", total_errors);
            $fclose(file);
            $finish;
        end

endmodule
```

Κεφάλαιο 3: Μέθοδοι εκτέλεσης

3.1 Εξομοίωση στο EDA Playground

α) Τυχαία διανύσματα δοκιμής (με αρχείο εξόδου): https://edaplayground.com/x/Q2HB
β) Χειροκίνητη δοκιμή (με κυματομορφές των σημάτων): https://edaplayground.com/x/gkRP

Τα βήματα για την εκτέλεσή του είναι:
1) Δημιουργία λογαριασμού στο edaplayground.com.
2) Προαιρετική αλλαγή τυχόν παραμέτρων. Ενδεικτικά:
στο α), το “repeats”, το οποίο ορίζει το πλήθος επαναλήψεων εφαρμογής τυχαίων διανυσμάτων.
στο β), το “a”, “b”, “c”, “d”, “f”, “g” και “n”, που αποτελούν του συντελεστές και τη τιμή στην οποία θα υπολογιστεί το πολυώνυμο.
3) Πάτησε το κουμπί “Run” στο πάνω αριστερά κομμάτι της σελίδας.

Βίντεο επίδειξης αυτής της λειτουργίας: **Εξομοίωση**

3.2 Υλοποίηση σε FPGA

Οι σχεδιαστές υλικού συχνά χρησιμοποιούν τα λεγόμενα Field-Programmable Gate Arrays (FPGAs) για την επαλήθευση σχεδιασμών λόγω της ευκολίας επαναδιαμόρφωσης που προσφέρουν, επιτρέποντας έτσι τη γρήγορη δημιουργία πρωτοτύπων και τη δοκιμή διαφορετικών αρχιτεκτονικών. Τα FPGA προσφέρουν μια πιο κοντινή αναπαράσταση της τελικής υλοποίησης ενός σχεδιασμού σε σύγκριση με την προσομοίωση, επιτρέποντας στους σχεδιαστές να εντοπίσουν και να διορθώσουν πιθανά ζητήματα νωρίς κατά τη διαδικασία σχεδιασμού. Παρέχουν επιπλέον μια πλατφόρμα για δοκιμές σε πραγματικό χρόνο σύνθετων σχεδιασμών, προσφέροντας μεγαλύτερη ευελιξία και επεκτασιμότητα σε σύγκριση με τα σταθερά Application-Specific Integrated Circuit (ASIC). Επιπλέον, τα FPGA διευκολύνουν την ενοποίηση στοιχείων υλικού και λογισμικού, επιτρέποντας επίσης την ολοκληρωμένη επαλήθευση σε επίπεδο συστήματος. Τέλος, υποστηρίζουν την επικύρωση χρονικών περιορισμών και μετρήσεων απόδοσης, διασφαλίζοντας ότι ο σχεδιασμός πληρεί τις επιθυμητές προδιαγραφές πριν την ολοκλήρωση της εφαρμογής.

Βίντεο επίδειξης της Μηχανής Διαφορών Babbage σε FPGA του εργαστηρίου VLSI: **FPGA υλοποίηση**

Παράρτημα: Βασικά στοιχεία της Verilog

A.1 Τι είναι και τι δεν είναι η Verilog

Η Verilog είναι μια γλώσσα περιγραφής υλικού (HDL) με C-like συντακτικό που χρησιμοποιείται κυρίως στη σχεδίαση ψηφιακών κυκλωμάτων, επιτρέποντας στους σχεδιαστές να μοντελοποιούν, να προσομοιώνουν και να επαληθεύουν ψηφιακά συστήματα σε διάφορα επίπεδα αφαίρεσης. Υποστηρίζει παραλληλισμό και συγχρονισμένες λειτουργίες, επιτρέποντας την αποτελεσματική αναπαράσταση της συμπεριφοράς του υλικού και χρησιμοποιείται ευρέως σε τομείς όπως το σχεδιασμό ASIC, το προγραμματισμό FPGA και την δοκιμαστικότητα ψηφιακών συστημάτων (design-for-testability). Τέλος, διευκολύνει τη δημιουργία πολύπλοκων ψηφιακών συστημάτων περιγράφοντάς τα χρησιμοποιώντας μονάδες (modules), σήματα και συμπεριφορές, καθώς οι σχεδιαστές μπορούν να χρησιμοποιήσουν τη Verilog για να καθορίσουν τη δομή και τη λειτουργικότητα βασικά οποιωνδήποτε ψηφιακών ηλεκτρονικών συστημάτων, συμπεριλαμβανομένων καταχωρητών, συνδυαστικής και ακολουθιακών κυκλωμάτων. Ο κώδικας Verilog μπορεί ύστερα να περάσει μια διαδικασία σύνθεσης σε στοιχεία υλικού έτοιμα για παράδοση σε κάποια βιομηχανία για να κατασκευαστούν.

Η Verilog δεν είναι μια γλώσσα προγραμματισμού γενικής χρήσης όπως η C, αλλά είναι ειδικά προσαρμοσμένη για το σχεδιασμό υλικού. Σε αντίθεση πάλι με τη C, η Verilog δεν εκτελείται σε κάποια CPU ή GPU, αφού όπως είπαμε πριν περιγράφει τη συμπεριφορά των ψηφιακών κυκλωμάτων. Επιπλέον, δεν εστιάζει στην αλγοριθμική επεξεργασία ή τη διαδοχική εκτέλεση, δίνει όμως έμφαση στις ταυτόχρονες λειτουργίες σε υλικό, οπότε δεν μεταγλωττίζεται σε κώδικα μηχανής, αλλά παράγει ψηφιακό κύκλωμα. Ακόμη, δεν διαθέτει βιβλιοθήκες για εργασίες όπως δικτύωση, αλληλεπιδράσεις με το λειτουργικό σύστημα, δυναμική εκχώρηση μνήμης, αναδρομή ή πολύπλοκες δομές δεδομένων που βρίσκονται συνήθως σε γλώσσες προγραμματισμού. Τέλος, δεν υποστηρίζει τεχνικές εντοπισμού σφαλμάτων λογισμικού όπως breakpoints και watchpoints. Ο εντοπισμός σφαλμάτων γίνεται συνήθως μέσω εξομοίωσης υλικού και ανάλυσης των αντίστοιχων κυματομορφών. Λόγω όλων αυτών των διαφορών, δε γίνεται να αντιμετωπίζεται η Verilog σαν τη C ή άλλες γλώσσες προγραμματισμού, οπότε και συμπεριλαμβάνεται το εξής παράρτημα.

A.2 Τύποι δεδομένων και λέξεις-κλειδιά της Verilog

Η Verilog έχει δύο (2) κύρια (και συνθέσιμα) data types.
- wire: Αντιπροσωπεύουν συνδέσεις μεταξύ δομικών στοιχείων σαν απλά καλώδια, οπότε δε κρατάνε τη τιμή τους.
- reg: Αντιπροσωπεύουν την αποθήκευση δεδομένων, οπότε διατηρούν τη τιμή τους μέχρι να εκχωρηθεί ρητά. Μπορεί να χρησιμοποιηθούν για τη μοντελοποίηση latch, flip-flop και λοιπά, αλλά δεν αντιστοιχούν ακριβώς.

Η Verilog είναι γλώσσα τεσσάρων (4) τιμών.
- 1 / 0: Λογικές τιμές 1 και 0 αντίστοιχα.
- Z: Κατάσταση υψηλής εμπέδησης. Είναι η περίπτωση όπου τίποτα δεν καθορίζει την τιμή ενός καλωδίου.
- X: Μοντελοποιεί την περίπτωση όπου ο προσομοιωτής δεν μπορεί να αποφασίσει τη τιμή του δικτυώματος. Αποτελεί την αρχική κατάσταση των reg, και συμβαίνει επίσης όταν ένα καλώδιο οδηγείται στο 0 και στο 1 ταυτόχρονα ή η έξοδος έχει Z εισόδους.

Η Verilog περιέχει και διάφορες άλλες λειτουργίες, όπως parameters (σταθερές δηλωμένες στην αρχή), integer και real αριθμούς, strings και άλλα.
Ενδεικτικά:

```
parameter a = 3;
parameter var = 5.5;
integer num = 32’b1; // 32-bit integer written in binary
real var_decimal = 10.24;
reg [8*12:1] str_var = “Hello World!”; // ASCII string
```

Μερικές ακόμα λέξεις-κλειδιά, ειδικά από το testbench, είναι οι εξής:
`initial`: ένα μη-συνθέσιμο procedural block που χρησιμοποιείται κατά την εξομοίωση.
`forever`: ένας μη-συνθέσιμος βρόχος ατέρμονης επανάληψης που μπαίνει σε procedural block.
`repeat`: ένας μη-συνθέσιμος βρόχος πεπερασμένων επαναλήψεων που μπαίνει σε procedural block.
`wait`: εξετάζει αν ισχύει μια συνθήκη, και αν είναι ψευδής, οι δηλώσεις που την ακολουθούν θα παραμένουν αποκλεισμένες έως ότου η συνθήκη γίνει αληθής.
`$display`: Συνάρτηση εμφάνισης μηνύματος στο τερματικό.
`$fopen`: Συνάρτηση για το άνοιγμα αρχείου.
`$fclose`: Συνάρτηση για το κλείσιμο αρχείου.
`$fwrite`: Συνάρτηση για την εγγραφή σε αρχείο.
`$finish`: Συνάρτηση τερματισμού της εξομοίωσης.
`$random`: Συνάρτηση παραγωγής τυχαίου 32-bit αριθμού.
`$dumpfile`: Συνάρτηση που χρησιμοποιείται για την αποθήκευση των αλλαγών στις τιμές καλωδίων και καταχωρητών σε αρχείο.
`$dumpvars`: Συνάρτηση που χρησιμοποιείται για να καθορίσει ποιες μεταβλητές θα αποθηκευτούν στο αρχείο που αναφέρεται από το `$dumpfile`.

A.3 Τελεστές

Type	Symbol	Description	Note
Arithmetic	+	add	
	-	subtract	
	*	multiply	
	/	divide	may not synthesize
	%	modulus (remainder)	may not synthesize
	**	power	may not synthesize
Concatenation	{…, …}	concatenates bits	explicit bit lengths
Bitwise	~	not	
	|	or	
	&	and	
	^	xor	
	~& or &~	nand	mix two operators
Relational	>	greater than	
	<	less than	
	>=	greater or equal than	
	<=	less or equal than	
	==	equal	
	!=	not equal	
Logical	!	negation	
	||	logical OR	
	&&	logical AND	
	==	logical equality	
Shift Operators	>>	right shift	
	<<	left shift	
	>>>	right shift with MSB shifted right (sign)	
	<<<	same as <<	

Στη περίπτωση χρήσης παραπάνω από ενός τελεστή στη Verilog, η σειρά εκτέλεσής τους είναι η εξής:
1. Bitwise / Logical πράξεις (**~**, **|**, **&**, **^**, **~&**, **!**, **||**, **&&**)
2. Concatenation (**{…, …}**)
3. Πολλαπλασιασμός, διαίρεση, modulo (*****, **/**, **%**)
4. Αριθμητική πρόσθεση / αφαίρεση (**+**, **-**)
5. Ολισθήσεις (**<<**, **>>**, **<<<**, **>>>**)
6. Σχεσιακές διαφορών (**<**, **>**, **<=**, **>=**)
7. Σχεσιακές ισότητας (**==**, **!=**)

A.4 Συνδυαστική λογική

Υπάρχουν τρεις τρόποι να περιγράψει κανείς συνδυαστικά κυκλώματα στη Verilog, οι οποίοι είναι:
- Πύλες (not, and, or, xor, nand, nor, xnor), η οποία γενικώς αποφεύγεται
- Με αναθέσεις τύπου assign
- Μέσα σε κάποιο always block

Οι τελευταίες δύο μέθοδοι αποκαλούνται και περιγραφή στο επίπεδο μεταφοράς των καταχωρητών (Register Transfer Level, RTL), η οποία δείχνει τι λογική συνάρτηση ακολουθούν τα σήματα ενός συστήματος χωρίς να εστιάζει σε κάποια συγκεκριμένη υλοποίησή του.

Για την αντιπαράθεση των τριών αυτών τεχνικών, δίνονται αντίστοιχες ενδεικτικές υλοποιήσεις σε Verilog για ένα πολυπλέκτη 4-σε-1.

Για τη περιγραφή στο επίπεδο των πυλών, έχουμε:

Gate-Level Description

```
module multiplexer_gates (a, b, c, d, sel, y);

     input a, b, c, d;
     input [1:0] sel;
     output y;

     wire ns0, ns1, w0, w1, w2, w3;

     not(ns0, sel[0]);
     not(ns1, sel[1]);

     and(w0, ns1, ns0, a);
     and(w1, ns1, sel[0], b);
     and(w2, sel[1], ns0, c);
     and(w3, sel[1], sel[0], d);

     or(y, w0, w1, w2, w3);

endmodule
```

Για τις περιγραφές στο επίπεδο μεταφοράς των καταχωρητών, έχουμε:

RTL Description (assign)

```
module multiplexer_assign (a, b, c, d, sel, y);

     input a, b, c, d;
     input [1:0] sel;
     output y;

     assign y = (sel == 2'b00) ? a :
                (sel == 2'b01) ? b :
                (sel == 2'b10) ? c : d;

endmodule
```

RTL Description (always)

```
module multiplexer_always (a, b, c, d, sel, y);

     input a, b, c, d;
     input [1:0] sel;
     output reg y;

     always @(a or b or c or d or sel) begin
          if (sel == 2'b00) y <= a;
          else if (sel == 2'b01) y <= b;
          else if (sel == 2'b10) y <= c;
          else y <= d;
     end

endmodule
```

A.5 Ακολουθιακή λογική

Για τη περιγραφή ακολουθιακής λογικής, πρέπει να χρησιμοποιηθεί το always. Αν κάποια μετάβαση δεν ορίζεται ρητά, υπονοείται η διατήρηση κατάστασης, δηλαδή μνήμη. Ακολουθεί παράδειγμα ενός ακμοπυροδότητου καταχωρητή παράλληλης φόρτωσης με ασύγχρονη λειτουργία reset.

```
module register_with_parallel_load (reset, clk, enable, d, q);

     input [7:0] d;
     input reset, clk, enable;
     output reg [7:0] q;

     always @(posedge reset or posedge clk) begin
          if (reset) q <= 0;
          else if (enable) q <= d;
     end

endmodule
```

A.6 Μέθοδοι κωδικοποίησης FSM

Οι μηχανές πεπερασμένων καταστάσεων αποτελούνται από τα εξής στοιχεία:
- Ακολουθιακή λογική (καταχωρητής κατάστασης)
- Συνδυαστική λογική (λογική επόμενης κατάστασης και λογική εξόδου)

Συνεπώς, θα σχεδιάζουμε τα FSM μας, είτε είναι Moore είτε είναι Mealy, με δύο (2) ξεχωριστά block. Επίσης, υπάρχει και το ζήτημα της κωδικοποίησης καταστάσεων (binary, gray, one-hot) που σχετίζεται με θέματα χώρου, ενέργειας, ταχύτητας και θορύβου.

A.7 Ιεραρχική σχεδίαση

Τα modules μπορούν να κάνουν αναφορά σε άλλα modules για να σχηματίσουν μια ιεραρχία, που διευκολύνει το σχεδιασμό σύνθετων αρχιτεκτονικών και προωθεί την επαναχρησιμοποίηση διάφορων επιμέρους σχεδιασμών. Ας δούμε για παράδειγμα πώς γίνεται να φτιαχτεί ένα multiplexer 16-σε-1 από επιμέρους multiplexer 4-σε-1 που είδαμε προηγουμένως.

Ένας τρόπος:

```
module multiplexer_16_to_1 (a, b, c, d, sel, y);

     input [3:0] a, b, c, d;
     input [3:0] sel;
     output y;

     wire w0, w1, w2, w3;

     multiplexer_assign mux0 (a[0], b[0], c[0], d[0], sel[1:0], w0);
     multiplexer_assign mux1 (a[1], b[1], c[1], d[1], sel[1:0], w1);
     multiplexer_assign mux2 (a[2], b[2], c[2], d[2], sel[1:0], w2);
     multiplexer_assign mux3 (a[3], b[3], c[3], d[3], sel[1:0], w3);
     multiplexer_assign mux4 (w0, w1, w2, w3, sel[3:2], y);

endmodule
```

Ένας λίγο πιο έξυπνος τρόπος:

```
module multiplexer_16_to_1_generate (a, b, c, d, sel, y);

     input [3:0] a, b, c, d;
     input [3:0] sel;
     output y;

     wire [3:0] w;

     // genvar controls the generation loop
     genvar i;
     generate
     for (i=0; i<=3; i=i+1) begin
          multiplexer_assign mux_first_layer (a[i], b[i], c[i], d[i], sel[1:0], w[i]);
     end
     multiplexer_assign mux_final (w[0], w[1], w[2], w[3], sel[3:2], y);

endmodule
```

Αναφορές:

[1] It Began with Babbage: The Genesis of Computer Science (2013, 1st Edition, Oxford University Press) - Subrata Dasgupta
[2] Digital Design: Principles and Practices, (2021, 5th Edition, Pearson) - John F. Wakerly
[3] Σχεδίαση Συστημάτων με Χρήση Υπολογιστών (E-CAD) (2004, 2η Έκδοση, Πανεπιστήμιο Πατρών) - Χαρίδημος Βέργος
[4] FPGA Prototyping by Verilog Examples: Xilinx Spartan-3 Version (2008, 1st Edition, Wiley) - Pong P. Chu
[5] Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes (1991, 1st Edition, Morgan Kaufmann) - Frank Thomson Leighton
[6] Introduction to Parallel Processing, Algorithms and Architectures (2002, 1st Edition, Kluwer) - Behrooz Parhami
[7] The Design and Analysis of Parallel Algorithms (1989, 1st Edition, Prentice-Hall) - Selim G. Akl
[8] CMOS VLSI Design: A Circuits and Systems Perspective (2011, 4th Edition, Pearson) - Neil H. E. Weste, David M. Harris, 4th Edition
[9] Digital VLSI Systems Design: A Design Manual for Implementation of Projects on FPGAs and ASICs Using Verilog (2007, 1st Edition, Springer) - Seetharaman Ramachandran
[10] Algorithm Design (2005, 1st Edition, Pearson) - Jon Kleinberg, Éva Tardos