“Nibbling bits”, από τη Λογική στην Αρχιτεκτονική

Μια εισαγωγή στη ροή Σχεδίασης Επεξεργαστών

Α.Ο. Φαρμάκης

In association with the ACM Student Chapter University of Patras

Περιεχόμενα της σημερινής παρουσίασης

Η τέχνη του σχεδιασμού επεξεργαστών (1/3)

Ο σχεδιασμός ευέλικτων και δυνατών επεξεργαστών γενικού και ειδικού σκοπού είναι αυτό που κάνει όλη την επιστήμη μας πραγματικότητα.

Κανείς δεν θα πρέπει να εκπλήσσεται ότι η αρχιτεκτονική υπολογιστών είναι κάτι που μπορεί να «τεντωθεί» σε οριακά παράλογες διαστάσεις πολυπλοκότητας.

Η τέχνη του σχεδιασμού επεξεργαστών (2/3)

Παραδείγματα: Επεξεργασία SIMD δεδομένων (διανύσματα, μητρώα, τανυστές), σχεδιασμός zero-trust, ενδομνημική επεξεργασία, ετερογενή υπολογιστική.

Τα κοινά: Όλα τα παραπάνω στηρίζονται στα ίδια θεμέλια.

Σημερινός στόχος: Τονισμός της ροής σχεδίασης και των εκτιμήσεων που αφορούν ολόκληρη τη διαδικασία σχεδιασμού ενός επεξεργαστή.

Η τέχνη του σχεδιασμού επεξεργαστών (3/3)

Για την ομαλή παρουσίαση μιας τέτοιας ροής, σχεδιάστηκε ένας επεξεργαστής (NibbleBuddy) χρησιμοποιώντας τις απλούστερες δυνατές μεθόδους.

Ο NibbleBuddy, όπως θα δούμε στη συνέχεια, είναι ένας 4-bit επεξεργαστής τύπου συσσωρευτή μονού κύκλου με Harvard αρχιτεκτονική μνήμης και είναι Turing complete*.

Τι σημαίνουν όλα αυτά όμως;

Αρχιτεκτονικές μνήμης

Von Neumann: Τα δεδομένα και το πρόγραμμα βρίσκονται στον ίδιο χώρο.

Harvard: Τα δεδομένα και το πρόγραμμα βρίσκονται σε ξεχωριστές μνήμες. Συναντάται συχνά σε microcontrollers.

Modified Harvard: Σαν τη κλασσική Harvard αρχιτεκτονική, που όμως επιτρέπει τη πρόσβαση στη μνήμη προγράμματος ως δεδομένα.

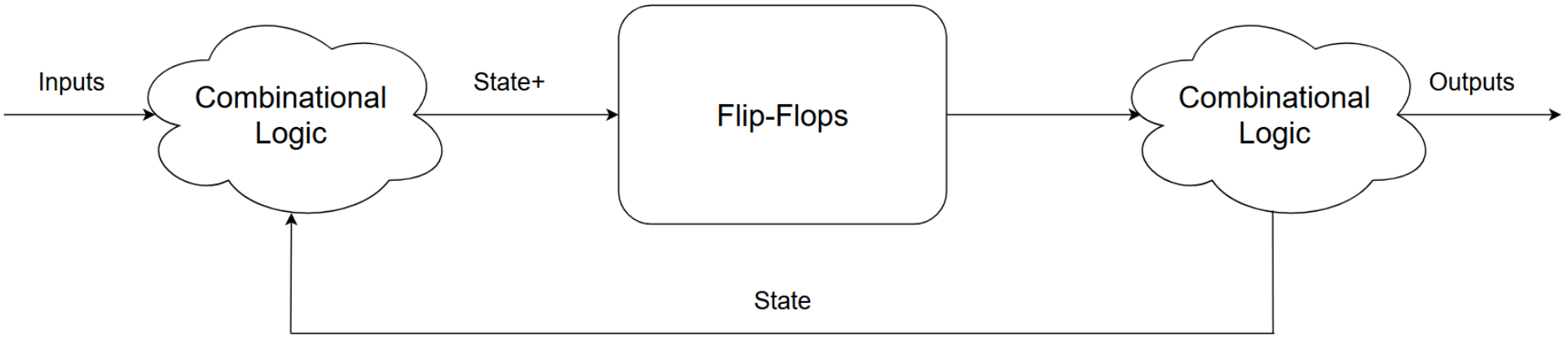
Απαιτήσεις κύκλων λειτουργίας (1/4)

Γενικώς, σε επεξεργαστές μονού κύκλου προτιμάται η Harvard αρχιτεκτονική μνήμης, για διάφορους λόγους.

Σε επεξεργαστές πολλαπλών κύκλων και μερικώς επικαλυπτόμενων λειτουργιών, τα πράγματα είναι σαφώς διαφορετικά!

Απαιτήσεις κύκλων λειτουργίας (2/4)

Βέβαια, όποια από αυτές τις επιλογές περί κύκλων ρολογιού και να επιλέξουμε, ένας επεξεργαστής παραμένει ένα είδος μηχανής πεπερασμένων καταστάσεων!



Απαιτήσεις κύκλων λειτουργίας (3/4)

Οι βασικότερες λειτουργίες όλων των εντολών ενός επεξεργαστή είναι οι εξής:

- **Προσκόμιση** (fetch)
- **Αποκωδικοποίηση** (decode)
- **Εκτέλεση** (execute)

Οι απαιτήσεις κύκλων λειτουργίας καθορίζει και το κύκλωμα ελέγχου ροής εκτέλεσης!

Απαιτήσεις κύκλων λειτουργίας (4/4)

Σε επεξεργαστές πολλαπλών κύκλων ή/και μερικώς επικαλυπτόμενων λειτουργιών, σαφώς το κύκλωμα ελέγχου θα είναι ακολουθιακό.

Στις περιπτώσεις μονού κύκλου, θα είναι συνδυαστικό κύκλωμα, το οποίο είναι πιο απλό σχεδιαστικά!

Βέβαια, το συνδυαστικό μονοπάτι θα είναι μεγαλύτερο...

Μηχανές N-τελεστών (1/4)

Τα είδη (και το πλήθος) των τελεστών καθορίζει τις εντολές που θα είναι διαθέσιμες στο Instruction Set Architecture (ISA) που θα ορίσουμε.

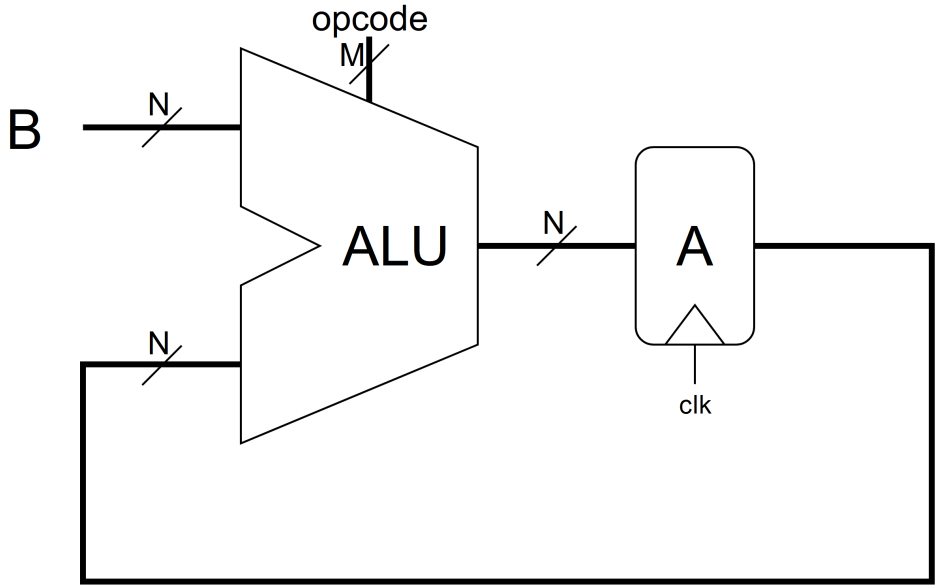
Μια ISA αποτελεί μια "γενίκευση" της αρχιτεκτονικής που ορίζουμε → πολλές διαφορετικές υλοποιήσεις που είναι δυαδικά συμβατές (σκεφτείτε πίνακες αληθείας).

Θα περιοριστούμε σε περιπτώσεις που αφορούν απευθείας και άμεση διευθυνσιοδότηση όμως, χάριν απλότητας!

Μηχανές N-τελεστών (2/4)

1-operand machine (accumulator):

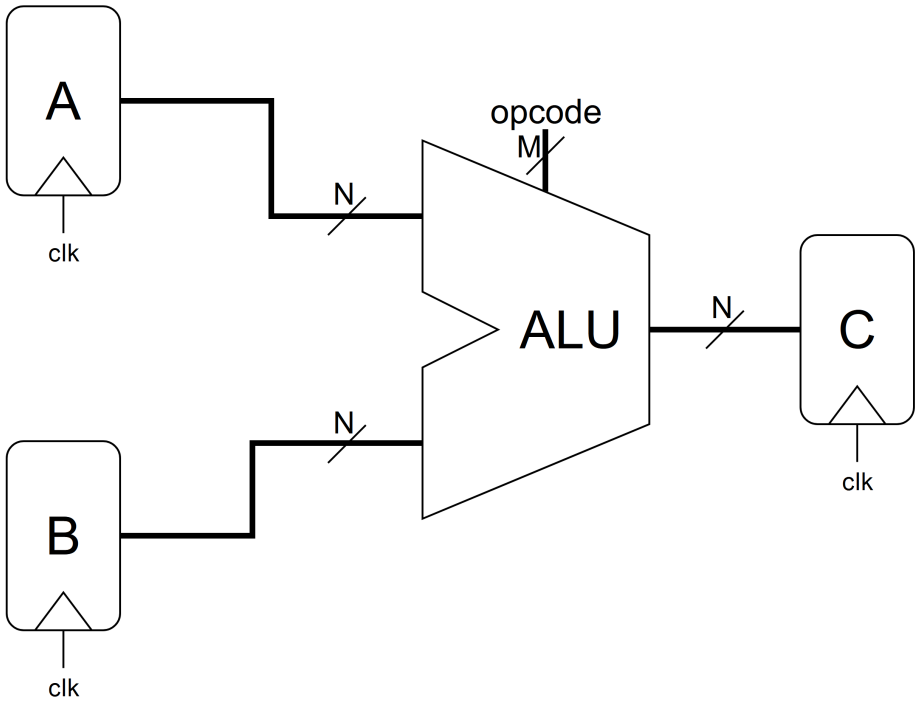
+ Πολύ απλό στη χρήση και στην υλοποίηση
+ Εξαιρετικό στις επαναληπτικές πράξεις
- Το load / store “πονάει”
- Μικρή ευελιξία



Μηχανές N-τελεστών (3/4)

3-operand machine (register-to-register):

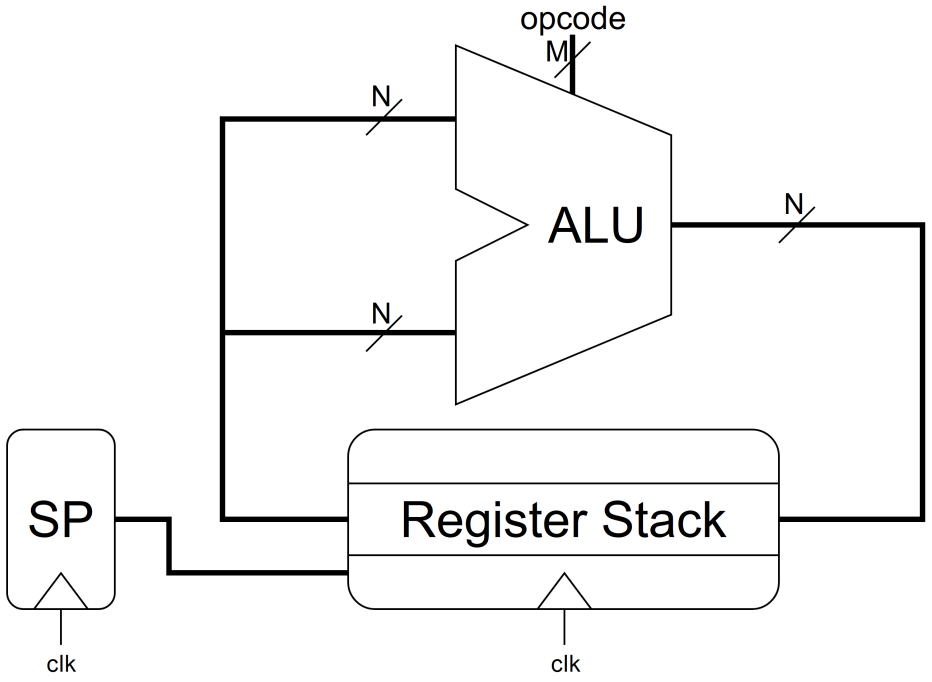
+ Απλό στη χρήση
+ Ευέλικτο
- Μεγαλύτερα μήκη εντολών
- Πιο σύνθετο writeback



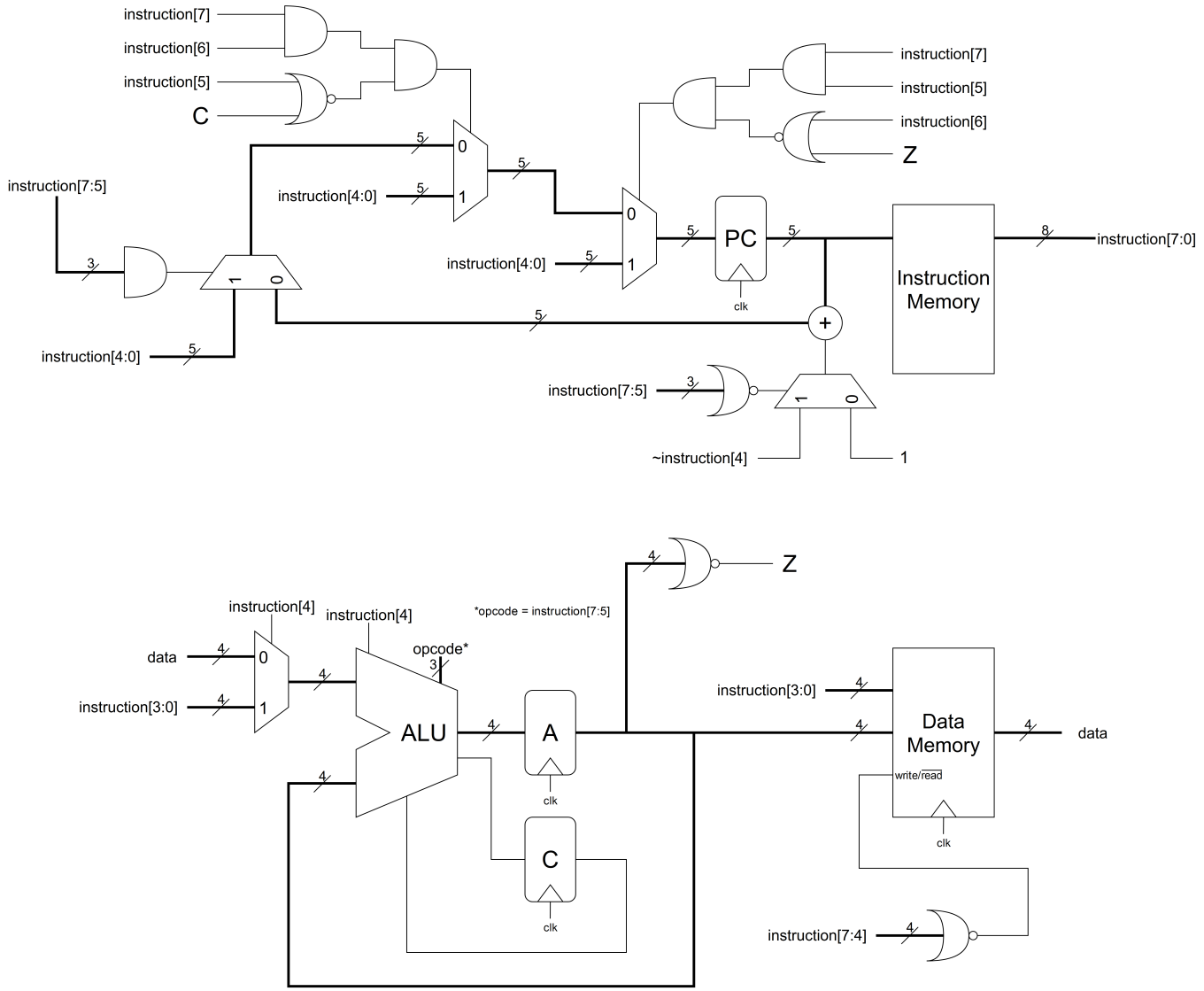
Μηχανές N-τελεστών (4/4)

0-operand machine (stack):

+ Μικρές εντολές
+ Απλό μοντέλο αξιολόγησης έκφρασεων (Reverse Polish Notation)
- Όχι τυχαίες προσβάσεις στη μνήμη
- Μεγάλο bottleneck οι συνεχείς πρόβασεις στη στοίβα

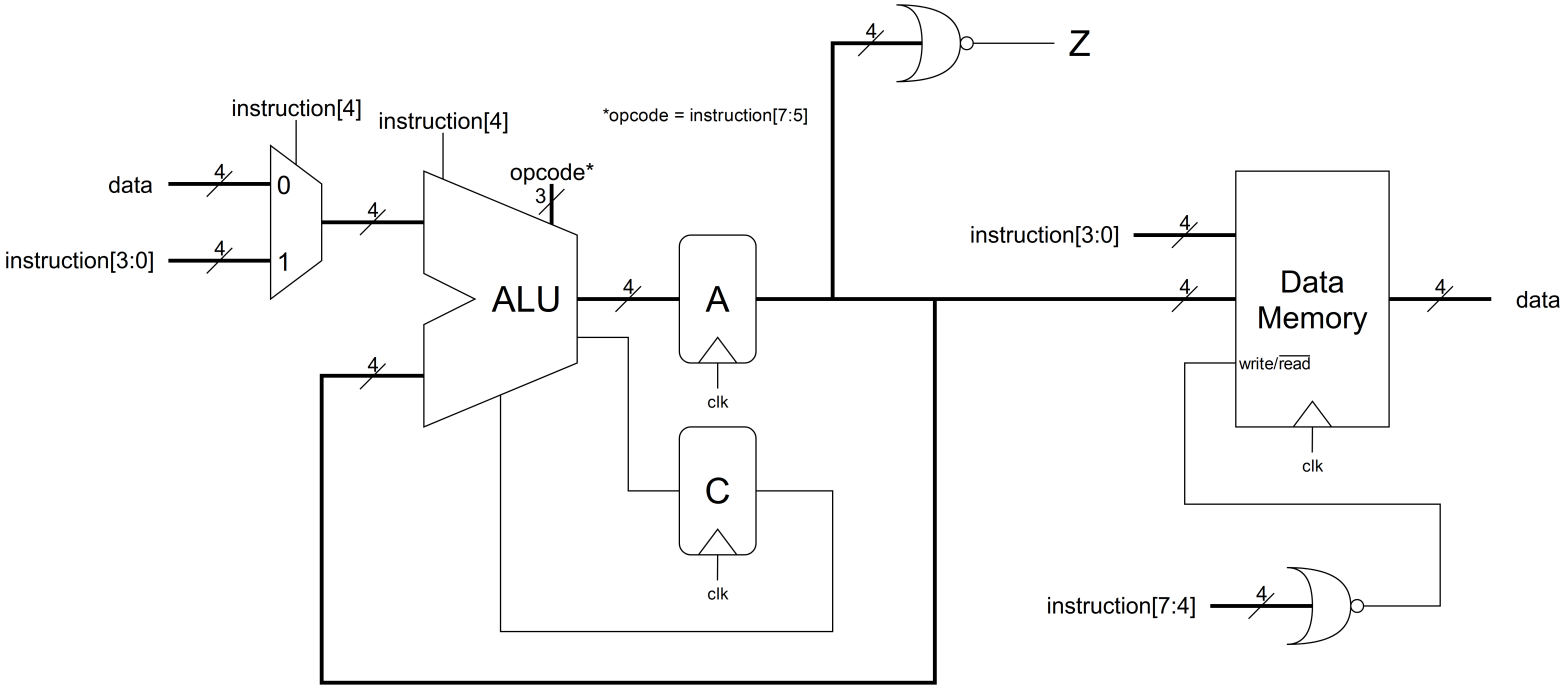


Η αρχιτεκτονική του NibbleBuddy



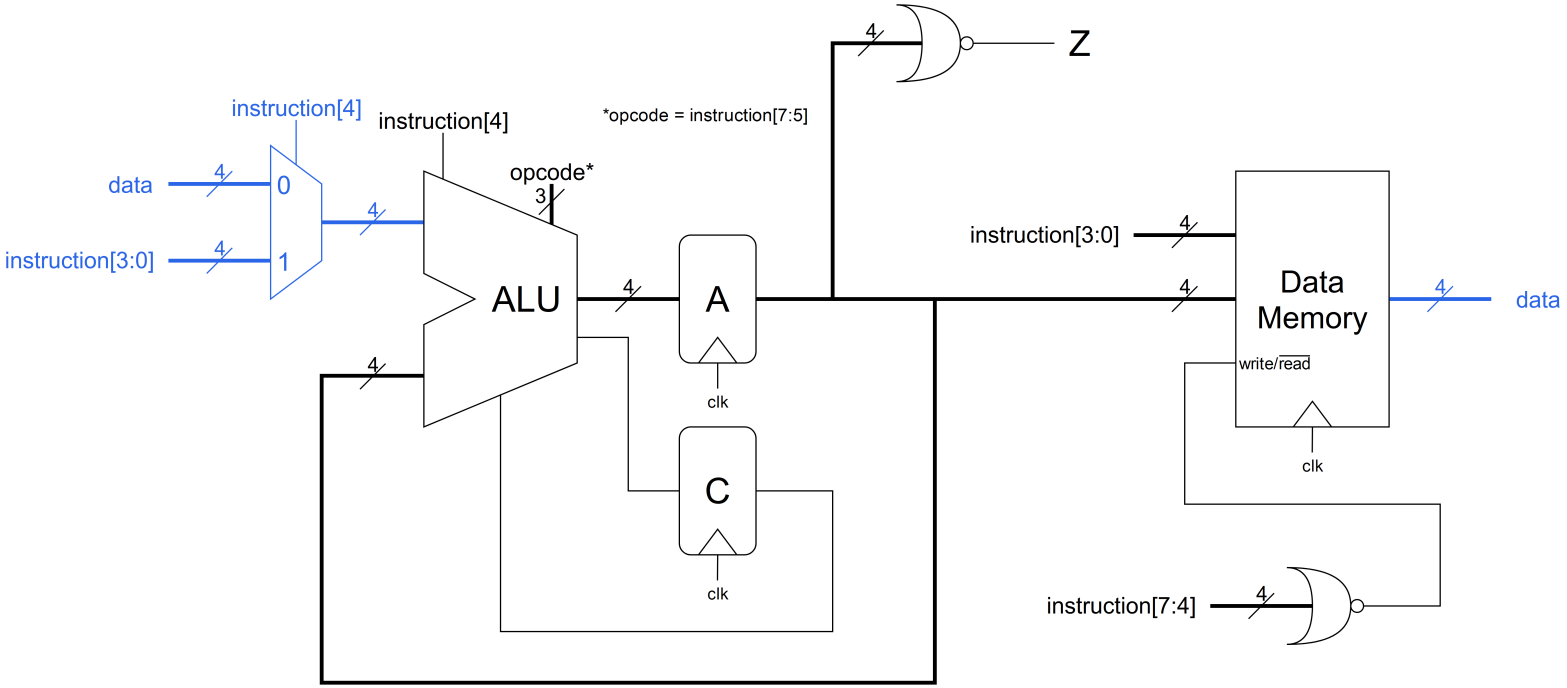
Λογική διαδρομής δεδομένων (1/6)

Η διαδρομή δεδομένων είναι απλή. Αποτελείται από έναν συσσωρευτή, τη λογική διευθυνσιοδότησης, διαχείριση μνήμης και τη παραγωγή σημαιών C (carry) και Z (zero).



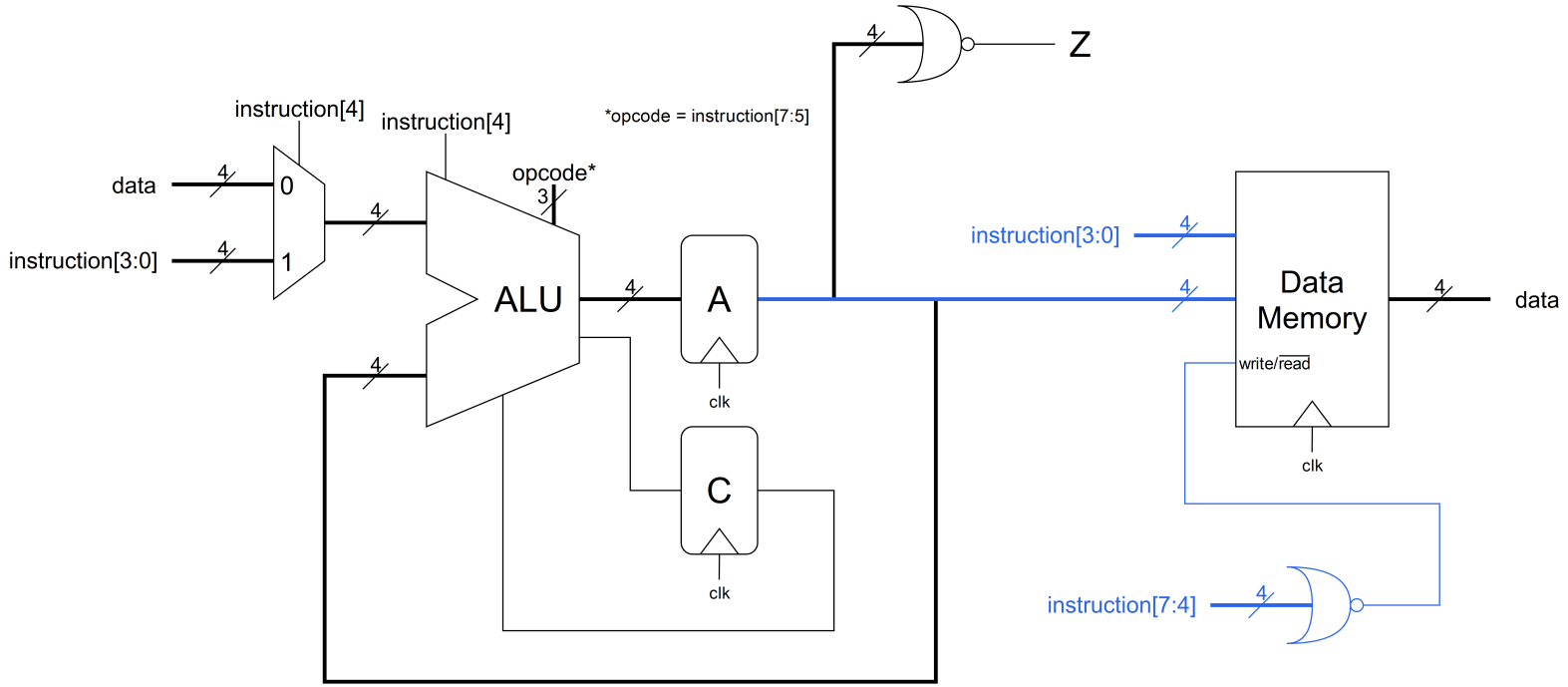
Λογική διαδρομής δεδομένων (2/6)

Η διαδρομή δεδομένων είναι απλή. Αποτελείται από έναν συσσωρευτή, τη λογική διευθυνσιοδότησης, τη διαχείριση μνήμης και τη παραγωγή σημαιών C (carry) και Z (zero).



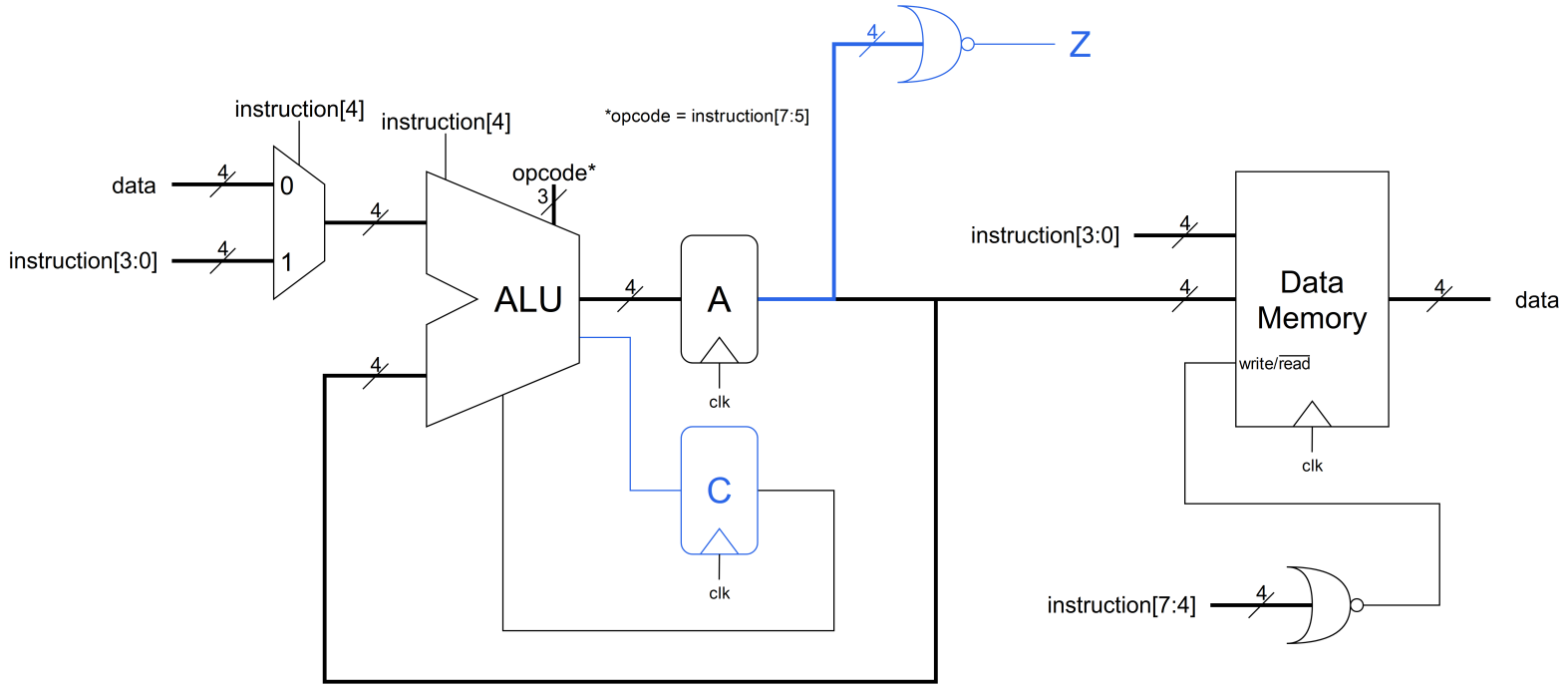
Λογική διαδρομής δεδομένων (3/6)

Η διαδρομή δεδομένων είναι απλή. Αποτελείται από έναν συσσωρευτή, τη λογική διευθυνσιοδότησης, τη διαχείριση μνήμης και τη παραγωγή σημαιών C (carry) και Z (zero).



Λογική διαδρομής δεδομένων (4/6)

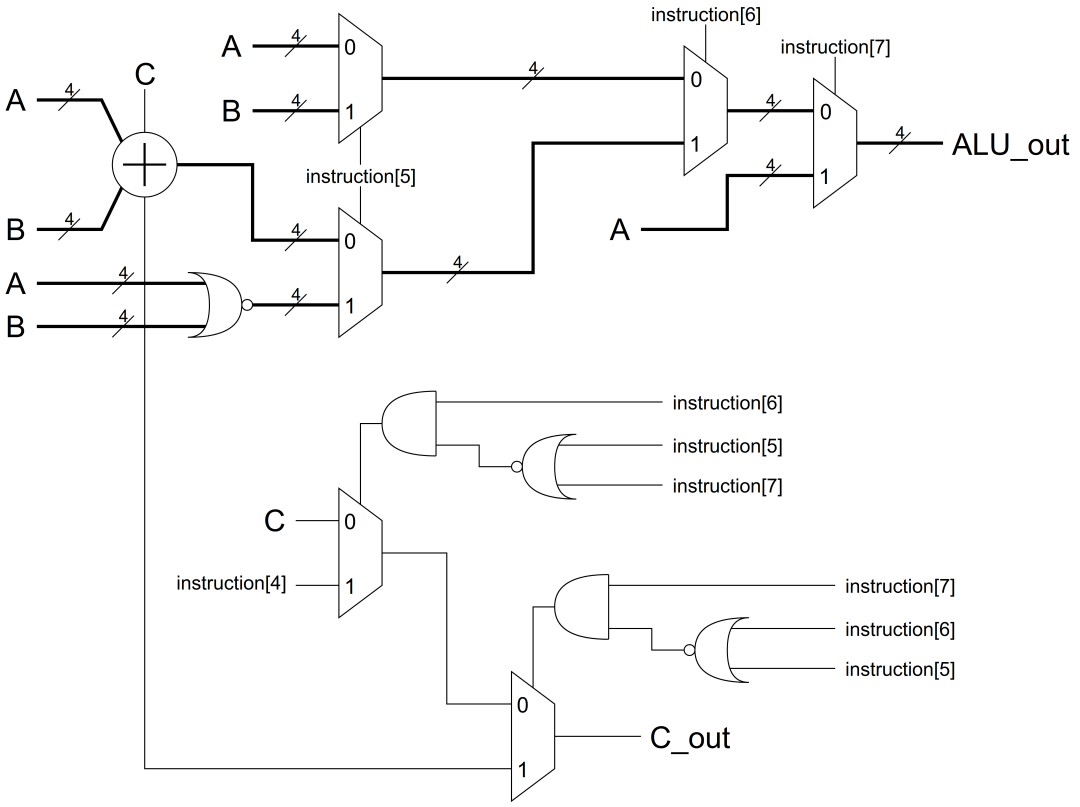
Η διαδρομή δεδομένων είναι απλή. Αποτελείται από έναν συσσωρευτή, τη λογική διευθυνσιοδότησης, τη διαχείριση μνήμης και τη παραγωγή σημαιών C (carry) και Z (zero).



Λογική διαδρομής δεδομένων (5/6)

Μέχρι στιγμής, αντιμετωπίζαμε την ALU της διαδρομής δεδομένων σαν “μαύρο κουτί”. Ας δούμε τι περιέχει μέσα!

Ο καταχωρητές A και C διατηρούν τις τιμές τους όταν δε χρησιμοποιούνται σε κάποια πράξη.



Λογική διαδρομής δεδομένων (6/6)

Αναλύοντας τη διαδρομή δεδομένων, βλέπουμε ότι διαχειρίζεται τις εξής πράξεις / εντολές:

- **ADC** (ADd with Carry)
- **NOR** (NOR)
- **LDA** (LoaD Accumulator)
- **STA** (STore Accumulator)
- **SETC** (SET Carry flag)

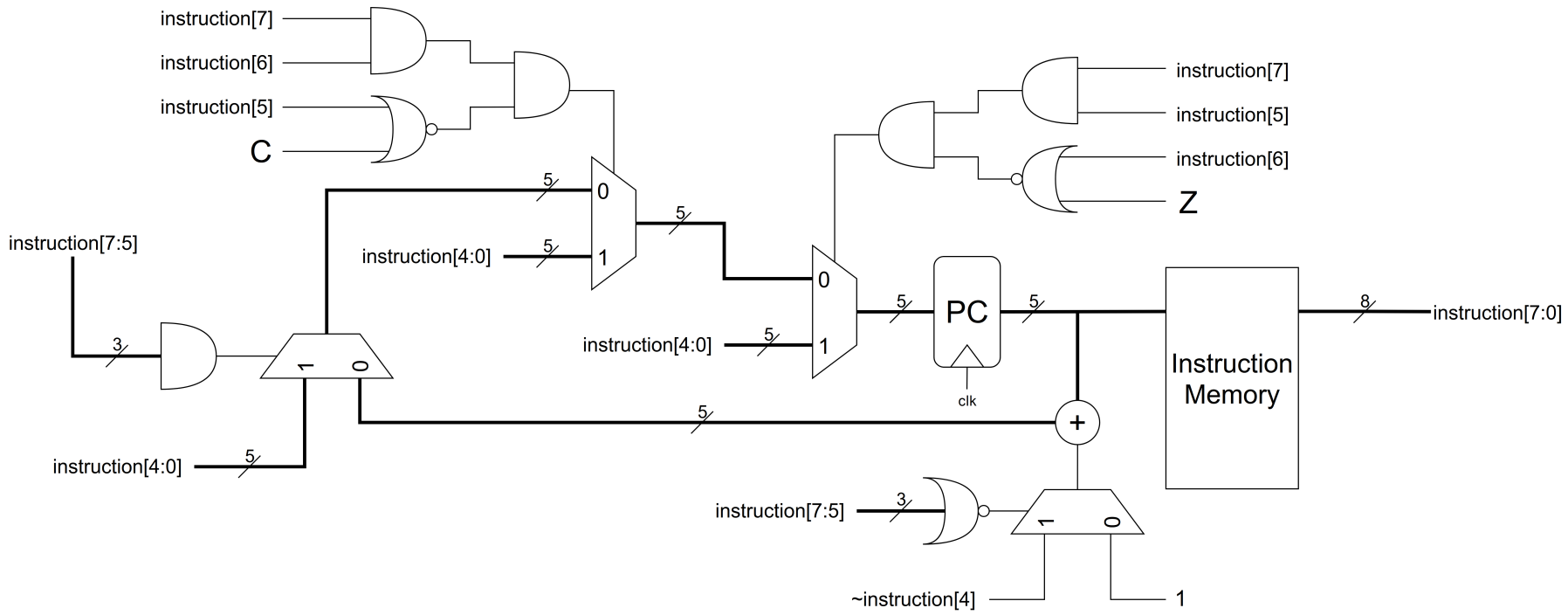
Άλματα και μετρητής προγράμματος (1/3)

Ακόμη και τα πιο απλοϊκά προγράμματα περιέχουν δύο (2) πολύ σημαντικά δομικά στοιχεία: βρόχους επανάληψης και εκτέλεση υπό συνθήκη.

Για να τα πετύχουμε αυτά, πρέπει να μπορούμε να επηρεάσουμε το μετρητή προγράμματος (program counter), ο οποίος "δείχνει" που βρισκόμαστε στην εκτέλεση του προγράμματός μας.

Άλματα και μετρητής προγράμματος (2/3)

Η λογική που ελέγχει το μετρητή προγράμματος είναι η εξής:



Άλματα και μετρητής προγράμματος (3/3)

Αναλύοντας τη λογική του μετρητή προγράμματος, βλέπουμε ότι διαχειρίζεται τις εξής πράξεις / εντολές:

- **HALT** (HALT)
- **JMP** (JuMP)
- **JNC** (Jump if Not Carry)
- **JNZ** (Jump if Not Zero)

Γλώσσα μηχανής, Assembly & HDLs (1/11)

Συνολικά, η ISA που ορίσαμε έχει τις εξής εντολές πλέον:

Από το datapath:

- **ADC** (ADd with Carry)
- **NOR** (NOR)
- **LDA** (LoaD Accumulator)
- **STA** (STore Accumulator)
- **SETC** (SET Carry flag)

Από το program counter:

- **HALT** (HALT)
- **JMP** (JuMP)
- **JNC** (Jump if Not Carry)
- **JNZ** (Jump if Not Zero)

Γλώσσα μηχανής, Assembly & HDLs (2/11)

Σαφώς και μπορούμε να δούμε τις πράξεις δυαδικά...

STA	0000 [4-bit data address]
HALT	0001 XXXX
LDA	001 [5 bits for direct / immediate and operand]
ADC	010 [5 bits for direct / immediate and operand]
NOR	011 [5 bits for direct / immediate and operand]
SETC	100 [Set bit] XXXX
JNZ	101 [5-bit instruction address]
JNC	110 [5-bit instruction address]
JMP	111 [5-bit instruction address]

Γλώσσα μηχανής, Assembly & HDLs (3/11)

...αλλά αυτό είναι πολύ μπερδευτικό! Τι μπορούμε να κάνουμε για αυτό;

Θα κατασκευάσουμε έναν Assembler!

Γλώσσα μηχανής, Assembly & HDLs (4/11)

Ένας Assembler κάνει το λεγόμενο tokenization, δηλαδή "σπάει" τις εκφράσεις σε μικρότερα κανονικά κομμάτια όπως έχουν οριστεί, και μετά χρησιμοποιεί ένα λεξικό να αντιστοιχίσει αυτά τα τμήματα σε γλώσσα μηχανής.

Έστω η εντολή LDA (0x04) → 00110100
Αντίστοιχα, ADC 0x0D → 01001101
Ακόμη, η εντολή JNZ (0x1A) → 10111010

Γλώσσα μηχανής, Assembly & HDLs (5/11)

Η δημιουργία αυτού του επεξεργαστή είναι αρκετά απλή ώστε να γίνει και “με το χέρι” αν χρειαστεί. Σε πιο μεγάλα κυκλώματα, χρησιμοποιούνται γλώσσες περιγραφής υλικού (hardware description languages) όπως η Verilog.

Οι σχεδιασμοί αυτοί είναι γρήγοροι, ευέλικτοι, και η λογική μπορεί να βελτιστοποιείται από τα εκάστοτε προγράμματα.

Γλώσσα μηχανής, Assembly & HDLs (6/11)

Τα ports και η συνδυαστική λογική του κυκλώματος:

```
module nibble_accumulator_harvard_cpu (reset, clk, instruction, data_in, data_out, address_out,
write_to_memory, program_counter);

    input reset, clk;
    input [7:0] instruction;
    input [3:0] data_in;
    output [3:0] data_out;
    output [3:0] address_out;
    output write_to_memory;
    output reg [4:0] program_counter;

    wire [3:0] operand;
    wire zero;

    reg [3:0] accumulator;
    reg carry;

    assign address_out = instruction[3:0];
    assign data_out = accumulator;
    assign write_to_memory = (instruction[7:4] == 4'b0000);            // STA here if we don't have HALT
    assign operand = (instruction[4]) ? instruction[3:0] : data_in;    // Data from memory or literal
    assign zero = (accumulator == 4'd0);
```

Γλώσσα μηχανής, Assembly & HDLs (7/11)

Η ακολουθιακή λογική του κυκλώματος (καταχωρητές):

```
    always @(posedge clk) begin
        if (reset) begin
            program_counter <= 5'h00;
            accumulator <= 8'h00;
            carry <= 1'b0;
        end
        else begin
            program_counter <= program_counter + 5'h01;
            case (instruction[7:5])
                3'b000: if (instruction[4]) program_counter <= program_counter;      // HALT or STA
                3'b001: accumulator <= operand;                                      // LDA
                3'b010: {carry, accumulator} <= accumulator + operand + carry;       // ADC
                3'b011: accumulator <= ~(accumulator | operand);                     // NOR
                3'b100: carry <= instruction[4];                                     // SETC
                3'b101: if (!zero) program_counter <= instruction[4:0];              // JNZ
                3'b110: if (!carry) program_counter <= instruction[4:0];             // JNC
                3'b111: program_counter <= instruction[4:0];                         // JMP
            endcase
        end
    end

endmodule
```

Γλώσσα μηχανής, Assembly & HDLs (8/11)

Η μνήμη (ROM) προγράμματος με δοκιμαστικό πρόγραμμα:

```
module instruction_memory (address, data);

     input [4:0] address;
     output reg [7:0] data;

     always @(address) begin
          case(address)
                0: data = 8'b010_1_0001;     // ADC 0x01
                1: data = 8'b101_0_0000;     // JNZ (0x00)
                2: data = 8'b011_1_0000;     // NOR 0x00
                3: data = 8'b000_0_1000;     // STA (0x08)
                4: data = 8'b111_0_1000;     // JMP (0x08)
                5: data = 8'b011_0_1111;     // NOR (0x0F)
                6: data = 8'b001_0_1000;     // LDA (0x08)
                7: data = 8'b111_0_1101;     // JMP (0x0D)
                8: data = 8'b100_0_xxxx;     // SETC 0
                9: data = 8'b110_0_0101;     // JNC (0x05)
               13: data = 8'b000_1_xxxx;     // HALT
               default: data = 8'b00;
          endcase
     end

endmodule
```

Γλώσσα μηχανής, Assembly & HDLs (9/11)

Η μνήμη (RAM) δεδομένων:

```
module data_memory (reset, clk, address, write, data_in, data_out);

     input reset, clk;
     input write;
     input [3:0] address;
     input [3:0] data_in;
     output [3:0] data_out;

     reg [3:0] memory [15:0];

     assign data_out = (!write) ? memory[address] : 4'b0000;

     integer i;
     always @(posedge clk) begin
          if (reset) begin
               for (i = 0; i < 16; i = i + 1) begin
                    memory[i] <= 0;
               end
          end
          else if (write) memory[address] <= data_in;
     end

endmodule
```

Γλώσσα μηχανής, Assembly & HDLs (10/11)

Βάζοντάς τα όλα μαζί:

```
module NibbleBuddy (reset, clk, instruction, data_in, data_out, address, write, program_counter);

     input reset, clk;
     output [7:0] instruction;
     output [3:0] data_in;
     output [3:0] data_out;
     output [3:0] address;
     output write;
     output [4:0] program_counter;

     nibble_accumulator_harvard_cpu CPU (reset, clk, instruction, data_in, data_out,
                                         address, write, program_counter);
     instruction_memory IMem (program_counter, instruction);
     data_memory DMem (reset, clk, address, write, data_out, data_in);

endmodule
```

Γλώσσα μηχανής, Assembly & HDLs (11/11)

...και δοκιμή προγράμματος!

```
module NibbleBuddy_test_program ();

     reg reset, clk;
     wire [7:0] instruction;
     wire [3:0] data_in, data_out, address_out;
     wire write;
     wire [4:0] program_counter;

     NibbleBuddy processor (reset, clk, instruction, data_in, data_out, address, write, program_counter);

     initial begin
          reset = 1'b1;
          clk = 1'b1;
          #5 reset = 1'b0;
          #5 clk = 1'b0;
          repeat (50) begin
               #10 clk = 1'b1;
               #10 clk = 1'b0;
          end
     end

endmodule
```

Κλείνοντας...

Τις ευχαριστίες μου για την υπομονή σας! Ελπίζω η παρουσίαση να ήταν ενδιαφέρουσα!

Το υλικό της παρουσίασης μπορείτε να το βρείτε και στο GitHub: