Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων

Επανάληψη - Μικροεπεξεργαστές

Vasileios Tenentes

University of Ioannina

E-mail: tenentes@cse.uoi.gr

Outline

1: Κίνητρο για επεξεργαστές χαμηλής κατανάλωσης ισχύος

2: Ταχύτητα/απόδοση επεξεργαστών

3: Δυναμική κατανάλωση ισχύος και σχεδιαστικές μέθοδοι μείωσής της

4. Προσδιορισμός της δυναμικής κατανάλωσης ισχύος μέσω προσομοίωσης

5: Στατική κατανάλωση ισχύος και σχεδιαστικές μέθοδοι μείωσής της

6: Stress-tests

7: Σχεδίαση με Verilog και ο MicroCPU

8: Προγραμματισμός του MicroCPU σε assembly και επεκτασιμότητα των λειτουργιών του

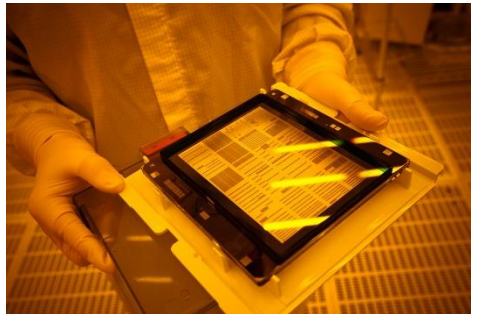
Physical applications of microprocessors

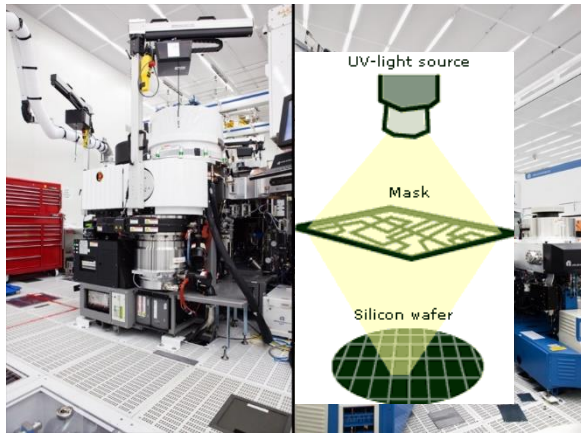
Κατηγορίες εφαρμογών μικροεπεξεργαστών

High Performance Computing – Servers
Mobile computing (συσκευές φορητές)
Embedded Devices (ενσωματωμένες συσκευές – π.χ. πλυντήρια)
IoT Εφαρμογές (ενσωματωμένες συσκευές με δίκτυο)
Safety critical applications (π.χ. τρένα, αεροπλάνα, αυτοκίνητα)

Shortly: Mask – Deposit – Etch - Repeat

Any design of SoCs based on CMOS becomes eventually a lithography mask





Applied Materials' Maydan Technology Center,
a state-of-the-art clean room in Santa Clara, California

So why we don’t design at the physical level directly?
Το παρακάτω flow ονομάζεται electronic design automation (EDA)

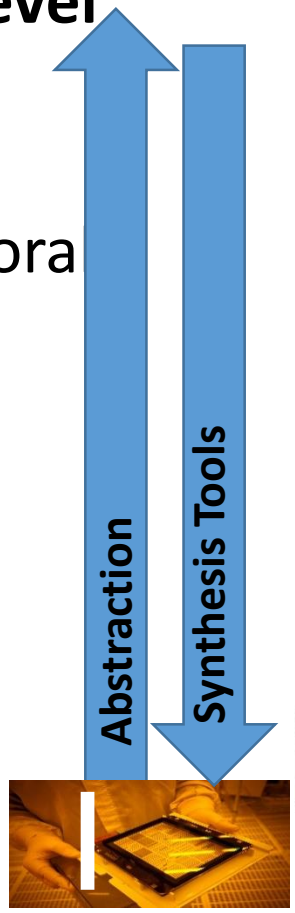
System Architecture Level

Register-Transfer Level (functional and behaviora description)

Logic Level (gates)

Transistor Level

Physical Level (lithography mask)

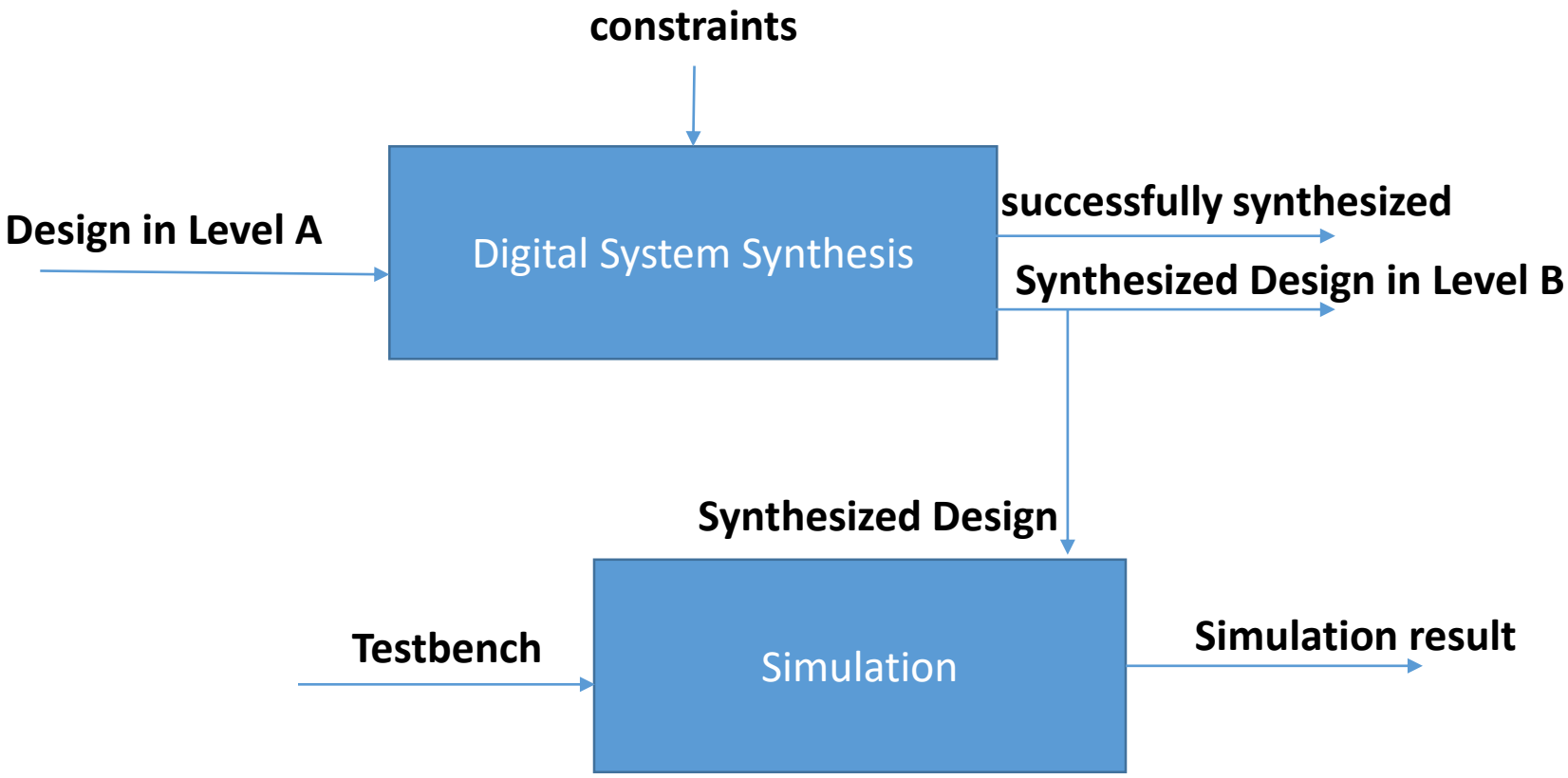


We need **abstraction** of the designing process through **synthesis tools**. It leads to:

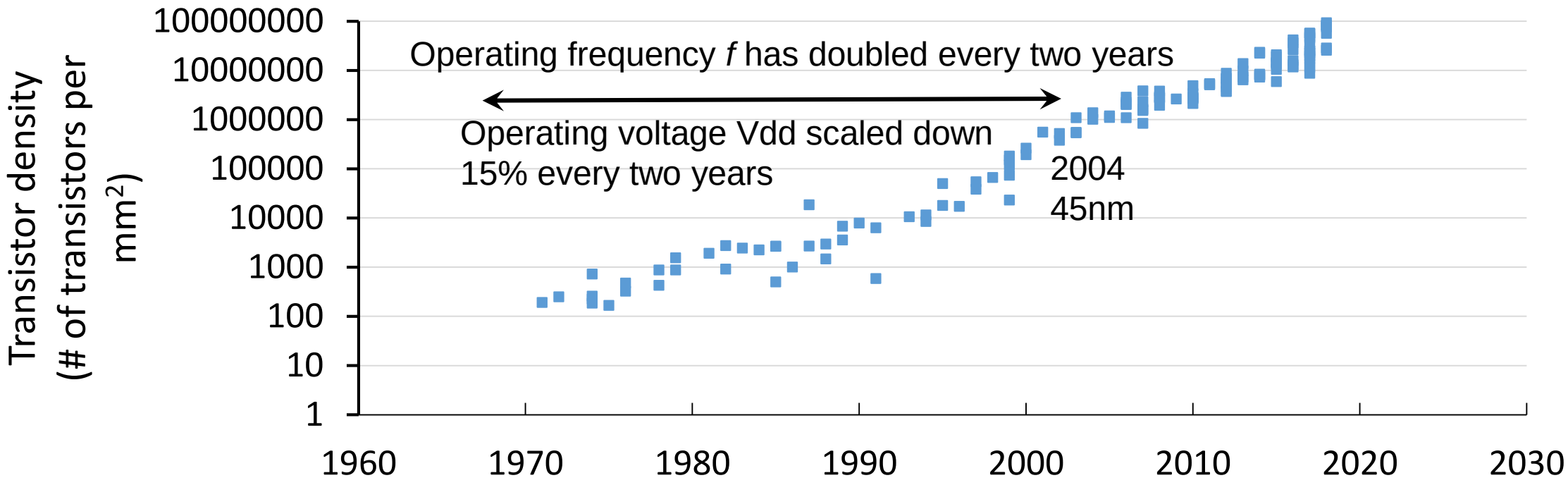
- Faster and cheaper design/manufacturing cycle
- More functionality
- Reusable designs
- !! provides control of specification requirements!
- Applications diversity

V. Tenentes

Synthesis is implementing abstraction



Transistor Density aspect of Moore's Law is holding well



Power density problems that arise

Drivers/Motivation for tackling the power density problem

1. Extension of battery life for mobile and IoT applications
2. Reduce cooling costs in enterprise servers (high performance computing)
3. Improve system reliability (high power density is causing reliability issues!!!)

Performance of Microprocessors

What about performance?

What is performance?

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

The BIG Picture

$$\text{Time} = \text{Seconds/Program} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

The only reliable measure of computer performance is time!

What about performance?

The BIG Picture

$$\text{Time} = \text{Seconds/Program} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

The only reliable measure of computer performance is time!

Check Yourself

A given application written in Java runs 15 seconds on a desktop processor. A new Java compiler is released that requires only 0.6 as many instructions as the old compiler. Unfortunately, it increases the CPI by 1.1. How fast can we expect the application to run using this new compiler? Pick the right answer from the three choices below:

a. $\dfrac{15 \times 0.6}{1.1} = 8.2 \text{ sec}$

b. $15 \times 0.6 \times 1.1 = 9.9 \text{ sec}$

c. $\dfrac{15 \times 1.1}{0.6} = 27.5 \text{ sec}$

Million instructions per second (MIPS)

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

Relationship between MIPS, clock rate and CPI:

$$\text{MIPS} = \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

$$\text{Time} = \text{Seconds/Program} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

MIPS Limitations

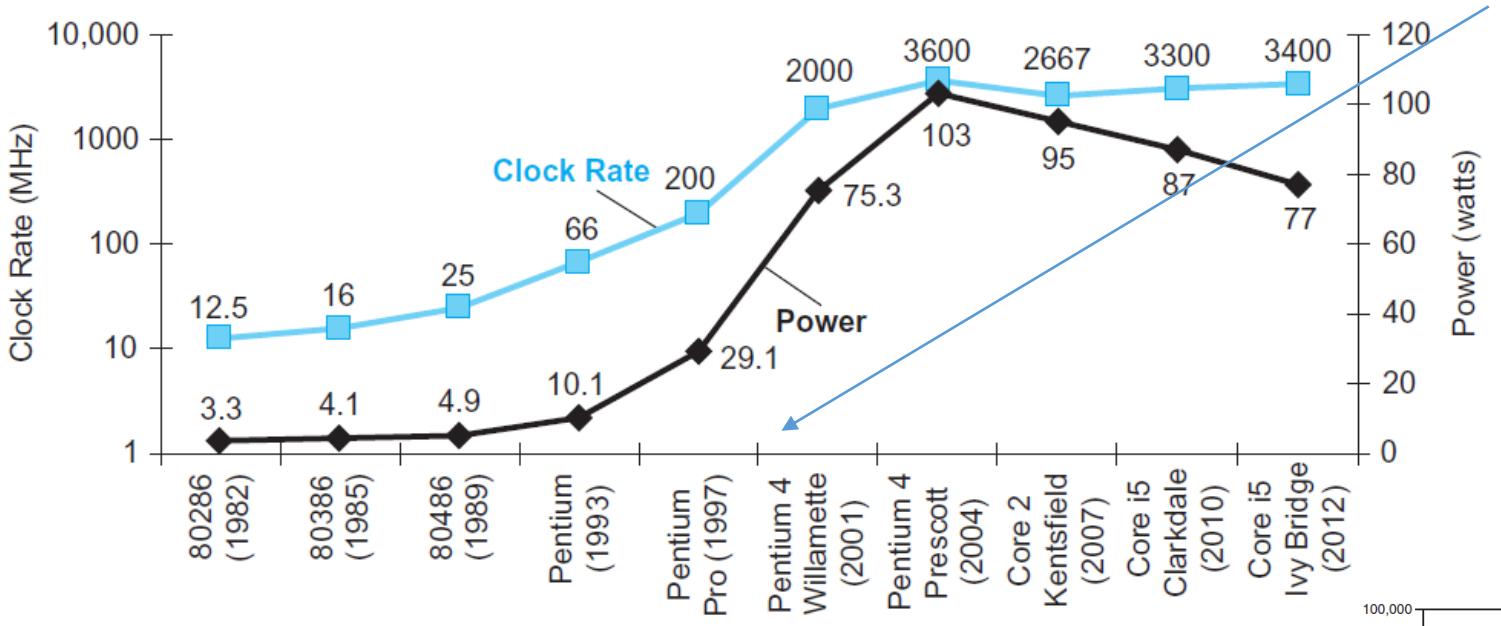
- It cannot compare computers with different instruction sets using MIPS, since the instruction counts will certainly differ and MIPS does not consider time.
- It varies between programs on the same computer; thus, a computer cannot have a single MIPS rating.

Measurement	Computer A	Computer B
Instruction count	10 billion	8 billion
Clock rate	4 GHz	4 GHz
CPI	1.0	1.1

a. Which computer has the higher MIPS rating?

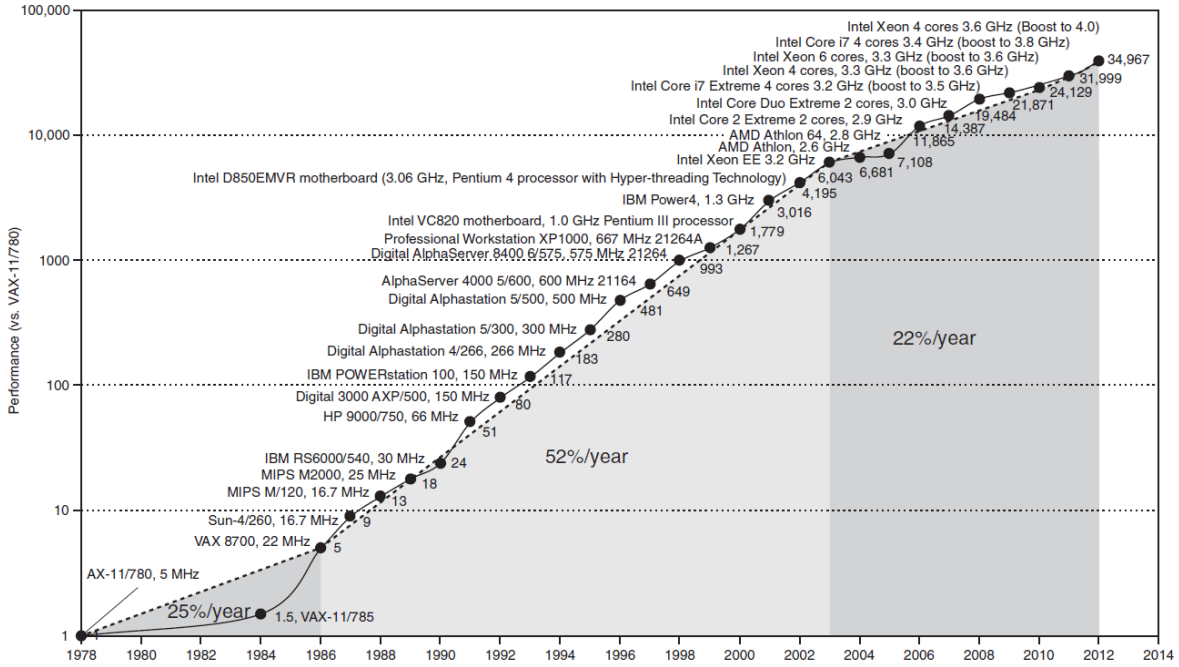
b. Which computer is faster?

The power Wall! (Power Wall # 1)



Pentium 4 made a dramatic jump in clock rate and power but less in performance

Multi-core microprocessors maintained performance increase for few years

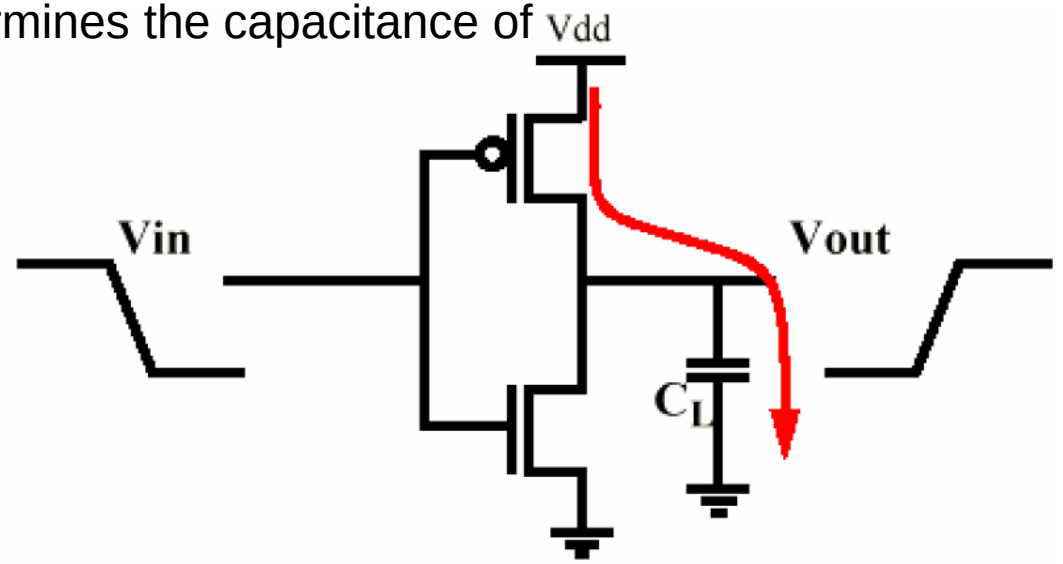


Dynamic Power

Dynamic Power

$$Power \propto 1/2 \times Capacitive\ load \times Voltage^2 \times Frequency\ switched$$

Frequency switched is a function of the clock rate.
Capacitive load per transistor is a function of both the number of transistors connected to an output (called the *fanout*) and the technology, which determines the capacitance of both wires and transistors



Models are a way to describe what we perceive and why not use it for analysis and prediction

Dynamic Power P_d model for a transistor:

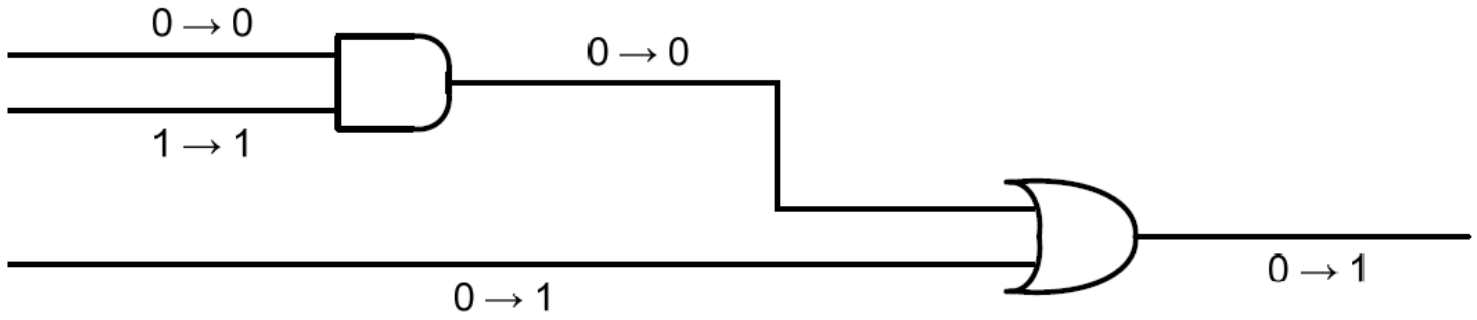
$$P_d = V^2_{dd} * F_{clk} * C_L * E_{SW} * 0.5$$

F_{clk} = clock frequency, C_L = capacitance seen by gate,
E_{SW} = gate switching activity (0-1 toggles)

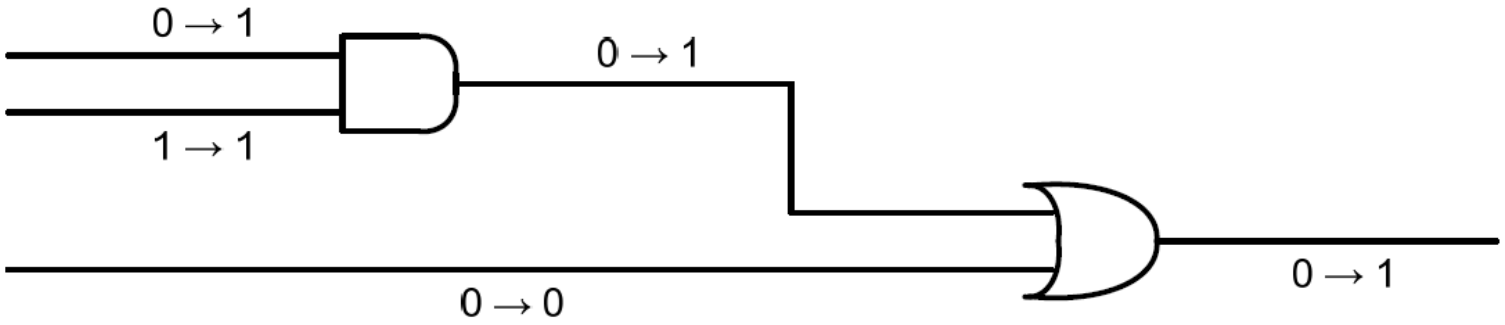
****there is another component of dynamic, which we haven't talked yet…

Workload dependence of dynamic power

- Switching activity depends on input patterns



OR gate consumes power if input 2 of OR gate changes form 0 to 1



AND and OR gate consume power if input 1 of AND gate changes form 0 to 1

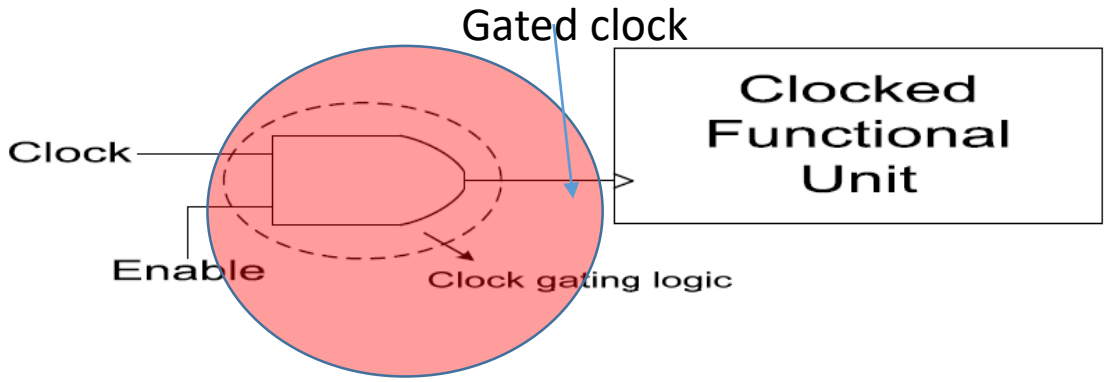
Dynamic power reduction design techniques

Dynamic power reduction happens at various design levels

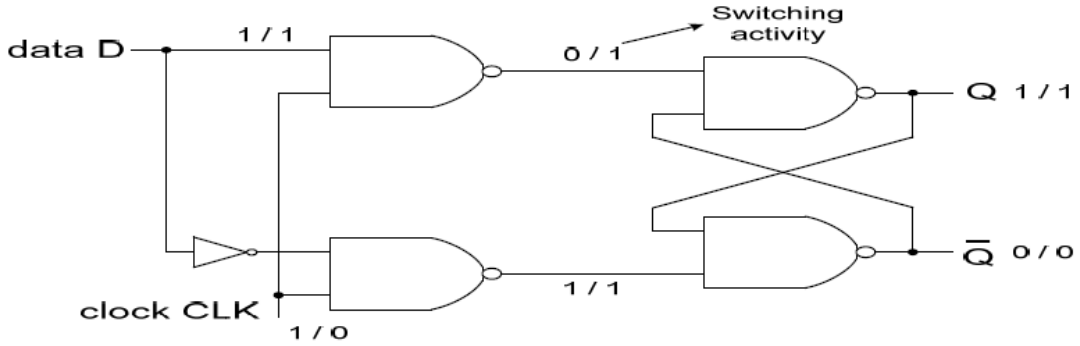
- Physical layout level = shorter interconnect → smaller C_L
- Transistor level = transistor with smaller W → smaller C_L
- Architectural level
  - Switching activity reduction: Clock gating
  - V_{dd} reduction: Multiple V_{dd}

Clock gating: architectural level dynamic power reduction

•When clocked modules (memory, register files, …) are inactive (stored values not charging), use clock gating to disable the clock feeding module



D Flip-Flop



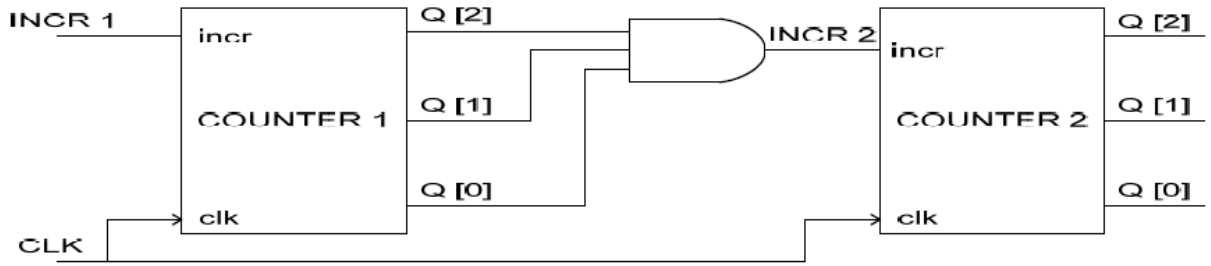
•FF consumes power due to switching activity even when input is not changing

•Clock gating reduces switching activity/dynamic power, up to 50% reduction

Example 1: Clock gating two, 3-bit counters

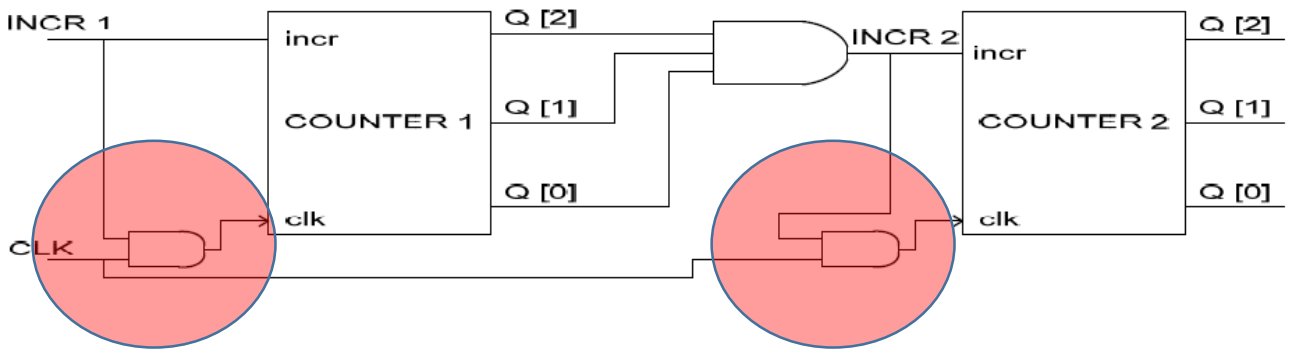
•Each counter increment on every clock cycle when INCR1 and INCR2 high

• INCR1 always high, COUNTER 1 increment every clock; COUNTER 2 increment only once every 8 clocks, but consume power during each clock cycle



•Clock gating logic (2 AND) cause counters to be clocked only when their increment signals high

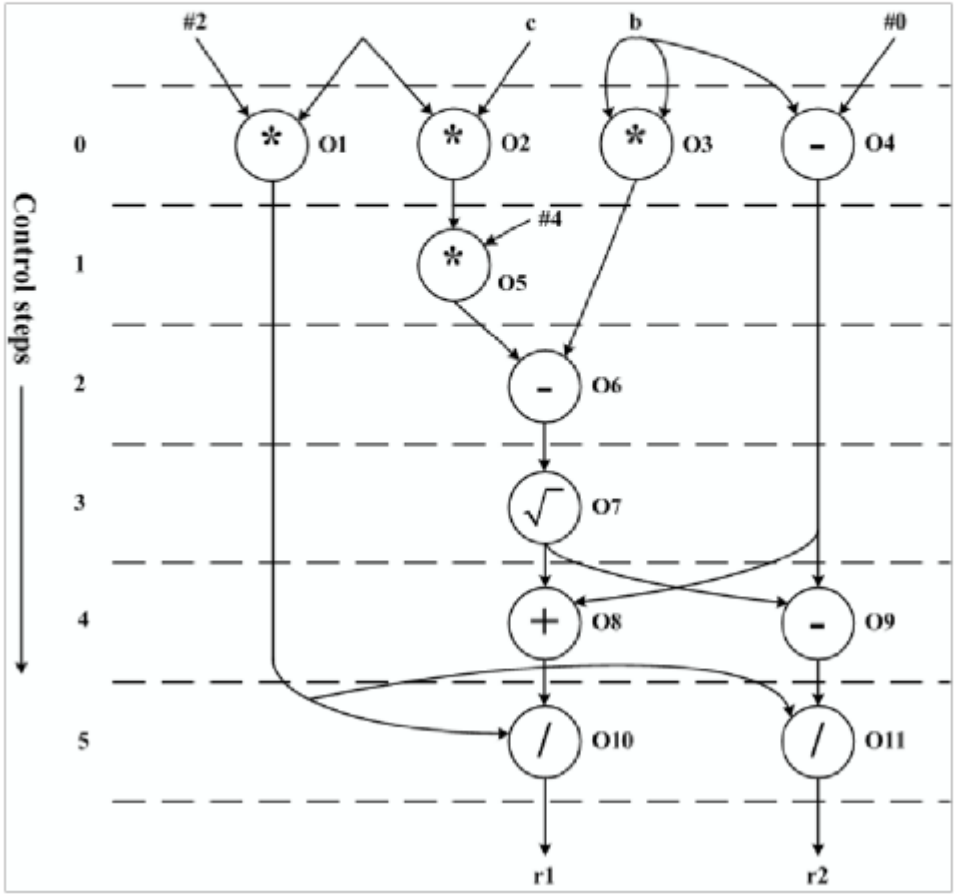
•COUNTER 2 not clocked when INCR2 is low



Πολλαπλές τάσης Τροφοδοσίας Multi-Vdd (multiple power domains) by example

- Modules on critical path use highest Vdd (to meet required timing constraints)
- Modules on non-critical paths use lower Vdd (reducing power)

Example



Assume Vdd= 2.2V, Vt=0.5v, FUs delay=10ns

Critical path operations: o2,05,o6,07,o8,o9, o10, o11
Operations: o1,o3,o4 are candidates for multi-cycled FUs

Critical path length= 10+10+10+10+10+10= 60ns
Delay of multi-cycled o1= 50ns, o3=20ns, o4=40ns

Delay of digital circuit, $D= Kd*Vdd/(Vdd-Vt)^2$

Where D functional unit delay, Vdd supply voltage, Vt threshold voltage of the functional unit, and Kd a constant

Consider multi-cycled FU o1

$50= (Kd*Vdd)/(Vdd-0.5)^2$ ……. (1)
$10=(Kd*2.2)/(2.2-0.5)^2$ … …. (2)

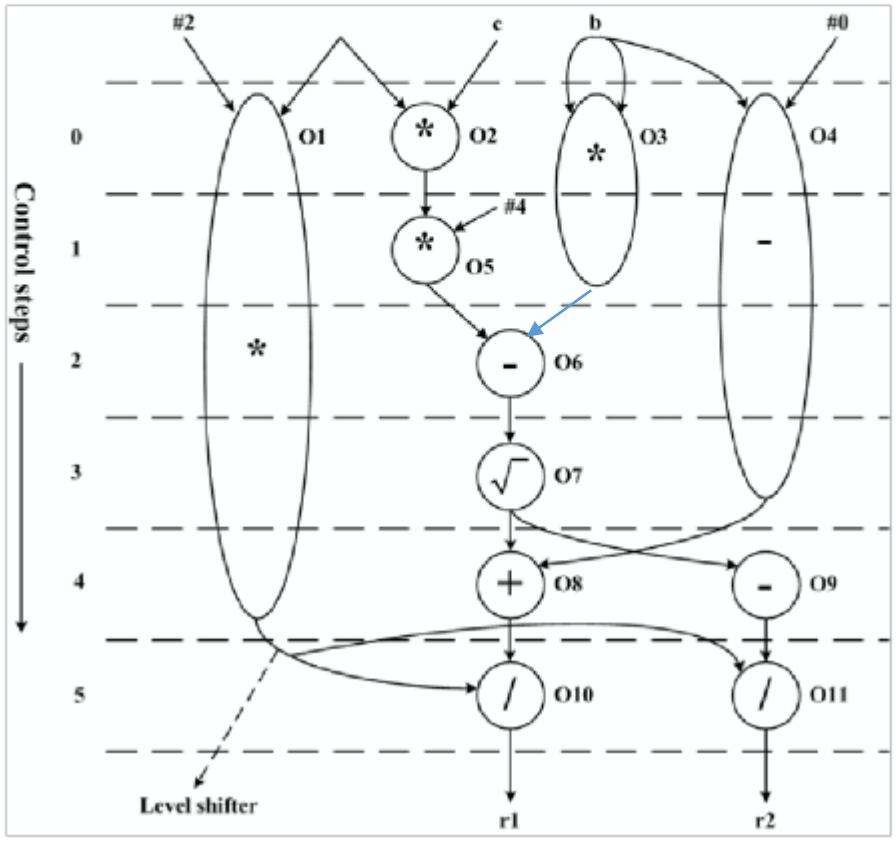
…

Multi-Vdd (multiple power domains)

From Eq.(2), Kd=13.1. Substitute Kd in Eq.(1) and solve for Vdd gives Vdd=1V.

The multi-cycled * (o1) will operate with Vdd=1V.

Similarly, multi-cycled FU o3, Vdd=1.48V.



Multi-Vdd Design Requirements

- Use of multiple supply voltages
- Level shifters needed after reduced voltage module

Multi-Vdd design flow will be presented at leakage reduction techniques

Dynamic Power Computation through Simulation

Dynamic Power and Switching activity

- Dynamic power of a circuit depends on its switching activity, which is how often the circuit elements are switching
- Switching activity Esw is the probability of elements (gate, transistors etc.) to switch from 0 to 1 or from 1 to 0:

$$P_d = V^2_{dd} * F_{clk} * C_L * E_{SW} * 0.5$$

F_{clk} = clock frequency, C_L = capacitance seen by gate,
E_{SW} = gate switching activity (0-1 toggles)

Relative power

$$Power \propto 1/2 \times Capacitive\ load \times Voltage^2 \times Frequency\ switched$$

Relative Power

EXAMPLE

Suppose we developed a new, simpler processor that has 85% of the capacitive load of the more complex older processor. Further, assume that it has adjustable voltage so that it can reduce voltage 15% compared to processor B, which results in a 15% shrink in frequency. What is the impact on dynamic power?

ANSWER

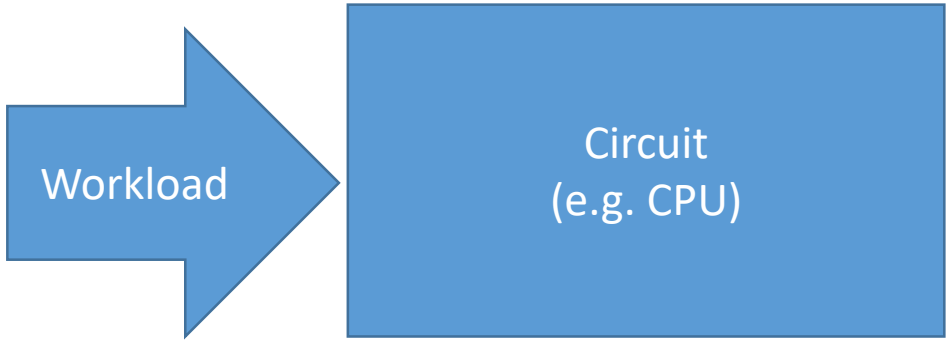
$$\frac{\text{Power}_{\text{new}}}{\text{Power}_{\text{old}}} = \frac{\langle \text{Capacitive load} \times 0.85 \rangle \times \langle \text{Voltage} \times 0.85 \rangle^2 \times \langle \text{Frequency switched} \times 0.85 \rangle}{\text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}}$$

Thus the power ratio is

$$0.85^4 = 0.52$$

Hence, the new processor uses about half the power of the old processor.

Βασικοί όροι



Στο διάγραμμα αυτό
Κύκλωμα **(Circuit):** είναι ένα λογικό κύκλωμα (π.χ. μια CPU ή μια αριθμητική λογική μονάδα) που εκτελεί κάποιους υπολογισμούς

φόρτος εργασίας κυκλώματος (**Workload**): είναι οι χρονοσειρές στις εισόδους του κυκλώματος κατά τη διάρκεια εκτέλεσης ενός προγράμματος

Μπορούμε να βρούμε το switching activity ενός κυκλώματος
Α) με μεγάλη ακρίβεια αν γνωρίζουμε τον φόρτο εργασίας του
Β) κατά προσέγγιση (χρησιμοποιώντας Monte Carlo ή Signal probabilities) όταν δεν γνωρίζουμε το φόρτο εργασίας του

Switching activity of known workload

When the workload (the input vectors) that is executed on the elements is known then the switching activity is computed by **simulating the execution** and counting switches at the output

Example on a 2-input AND gate:

Time	: t1	t2	t3	t4	t5
Input1	: 0	1	0	1	1
Input2	: 1	1	0	0	1
Output	: 0	1	0	0	1

The data above is called **timeseries** or **digital waveforms**.

Beside the timeseries of the workload, **the maximum number of possible switches** at the output of the elements is also needed to compute switching activity. In the case at hand it is: 0->1->0->1->0, which is 4 switches.

Generally for timeseries with N number of vectors, the worst case is N-1.

In this example, we observe 3 number of swithes at the output of the gate. So switching activity is:

Esw=(# of switches observed)/(# of maximum possible switches)=> Esw = 3/4

Switching activity of elements when the workload is not known

When **workload is unknown** then switching activity needs to be statistically evaluated. The task is to identify the most probable switching activity.

This is achieved by either:

1. **Using Monte Carlo**: generating **statistical workload** and transform the problem as a problem of known workload. To transform the problem using Monte Carlo:
   - Consider the inputs of the elements as random variables and take random samples for all of them. These samples, **which are called Monte Carlo permutations,** can be handled as **known workload**.
   - Then perform the switching activity computation of the known statistically generated workload by simulating and counting switches (also consider example MCAND4.m)
2. **Using signal probabilities**: signal probabilities propagation can be used for identifying the switching activity of elements and is presented in the next slide

Switching activity of elements when the workload is not known using signal probabilities

Signal probability P of a digital signal is the probability of the signal to be at logic-high (logic-1, '1', 'true')

Signal probability propagation process on a logic gate:

1. Signal probability values are set at the inputs of the gate.
2. Signal probability is propagated at the output of the gate using the signal probability function of the gate and the signal probabilities of the inputs.

How to evaluate the **signal probability function** of a gate?

Signal probability function is a function Pout(Pin1, Pin2, …., PinN) of the input signal probabilities. You do not need to remember the signal probability functions of all gates. They can be easily computed using probabilities theory and by examining the truth table of a gate.

To find the function you need to **gather Sum of Products (SOP – άθροισμα γινομένων)** from the truth table. Then you can perform the following check to be sure that the function is correct:

Check: set 0.5 as the signal probability to all inputs (Pin1=Pin2=….=PinN=0.5) and compute the signal probability value at the output. This is performed by counting on the truth table, the number of input combinations that lead to logic-1 at the output. Then, this value must be divided by the total number of input combinations (rows) in the truth table to get the Pout. So when Pin1=Pin2=0.5, Pout must honour:
Check=(# of rows in the truth table with Out=1)/(# total number of rows in the truth table).

To finalise the check, replace Pin1,Pin2,….,PinN with **0.5** value at the computed signal probability function. It must be equal to **Check.**

Examples in the next slide

Switching activity of elements when the workload is not known using signal probabilities

example on a 2-input AND gate

- **Sum of Products:** from the truth table of the 2-input AND gate signal Out is 1 only when In1=In2=1.
  Therefore, the candidate function as a sum of product of this, it becomes: **Pout_AND(Pin1, Pin2)=Pin1*Pin2**
- **Check**: The output becomes logic-`1' only during one input combination (out of the 4 input combinations). So it is **Check=1/4**. The check is successful because Pout_AND(0.5, 0.5)=0.5*0.5=0.25=1/4

Truth table 2-input AND

In1	In2	Out
0	0	0
0	1	0
1	0	0
1	1	1

example on a 2-input OR gate

- **Sum of Products:** from the truth table of the 2-input OR gate, signal Out is 1 in three cases: when In1=In2=1; when In1=1 and In2=0; and when In1=0 and In2=1. So the candidate function as a sum of products of these, it becomes: **Pout_OR(Pin1, Pin2)= Pin1*Pin2 + Pin1*(1-Pin2) + (1-Pin1)*Pin2**
- **Check**: The output becomes logic-`1' during three input combination (out of the 4 input combinations). So it is **Check=3/4**. The check is successful because: Pout_OR(0.5, 0.5)=0.5*0.5+0.5*(1-0.5)+(1-0.5)*0.5=> Pout_OR(0.5,0.5)=0.75=3/4

Truth table 2-input OR

In1	In2	Out
0	0	0
0	1	1
1	0	1
1	1	1

Switching Activity Computation

The switching activity of a gate is then computed using its output signal probability: a gate's output switches when two successive values of it are complementary. This occurs either: (1) when a bit is '1' and the successive bit is '0'; (2) or when a bit is '0' and the successive bit is '1':

So it is Esw=Pout*(1-Pout)+(1-Pout)*Pout=> **Esw=2*Pout(Pin1, Pin2)(1-Pout(Pin1, Pin2)).** With this equation, switching activity can be computed, when signal probabilities at the inputs is known. Note that if we consider the probability at the inputs to be 0.5, then this formula collapses to **Esw=2*Pout(0.5,0.5)(1-Pout(0.5,0.5))➔ Esw=2*Check*(1-Check)**.

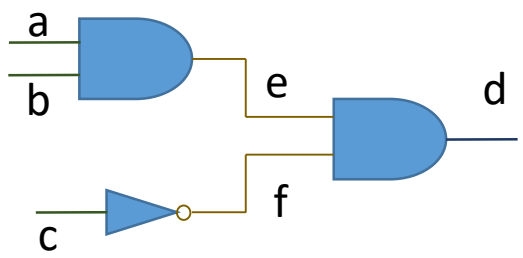
Some tricks:

- Signal probabilities of complementary gates are also complementary. Example: Pout_OR=1-Pout_NOR
- Although the check must be honoured you can use the Monte Carlo technique to evaluate if the probability function considered for a gate is correct or not.

Όμως τι γίνεται με κυκλώματα πιο πολύπλοκα από μια λογική πύλη;

- Για να βρούμε το switching activity ενός κυκλώματος πρέπει να βρούμε το switching activity όλων των δομικών του **στοιχείων**. Τα **δομικά στοιχεία** μπορεί να είναι πιο πολύπλοκα από λογικές πύλες αλλά στην περίπτωσή μας θα μας απασχολήσουν μόνο οι λογικές πύλες

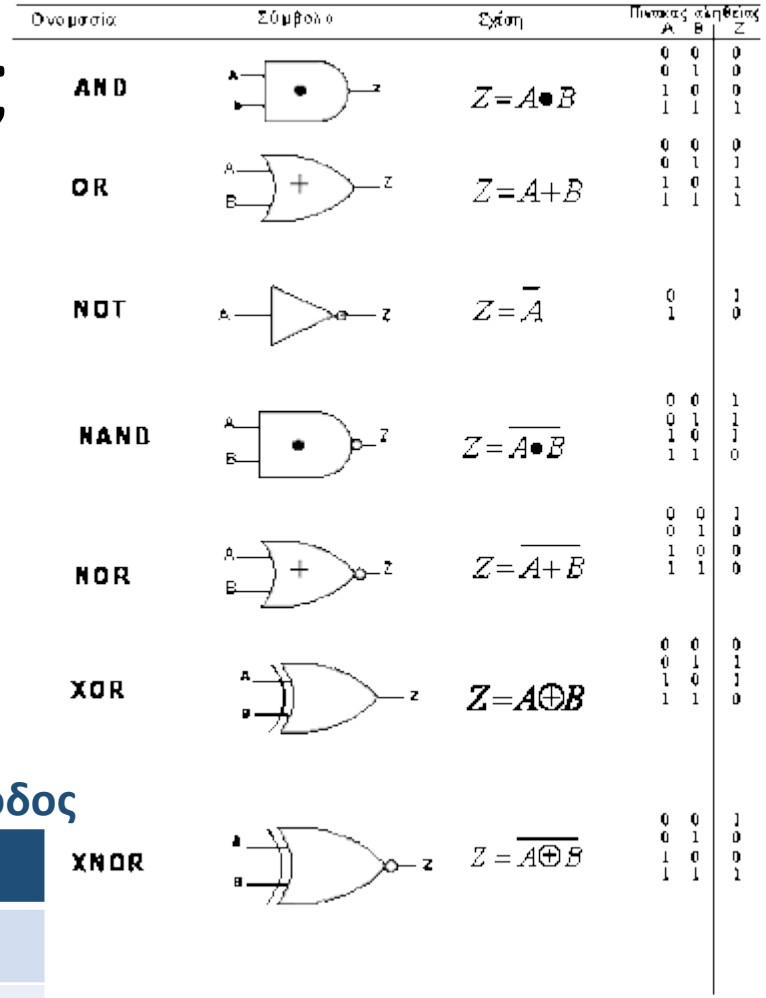
Παράδειγμα



Με γνωστό φόρτο εργασίας

	φόρτος εργασίας			Ενδιάμεσες έξοδοι		Τελική έξοδος
χρόνος	a	b	c	e	f	d
t1	0	1	0			?
t2	1	1	0			?
t3	1	0	0			?
t4	0	0	1			?

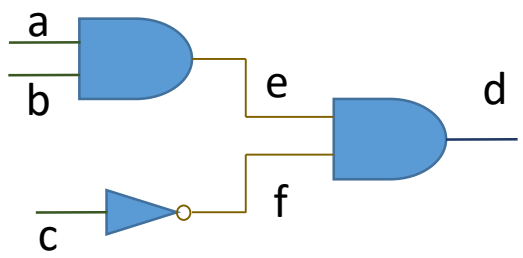
Βασικές Λογικές Πύλες



Όμως τι γίνεται με κυκλώματα πιο πολύπλοκα από μια λογική πύλη;

- Για να βρούμε το switching activity ενός κυκλώματος πρέπει να βρούμε το switching activity όλων των δομικών του **στοιχείων**. Τα **δομικά στοιχεία** μπορεί να είναι πιο πολύπλοκα από λογικές πύλες αλλά στην περίπτωσή μας θα μας απασχολήσουν μόνο οι λογικές πύλες

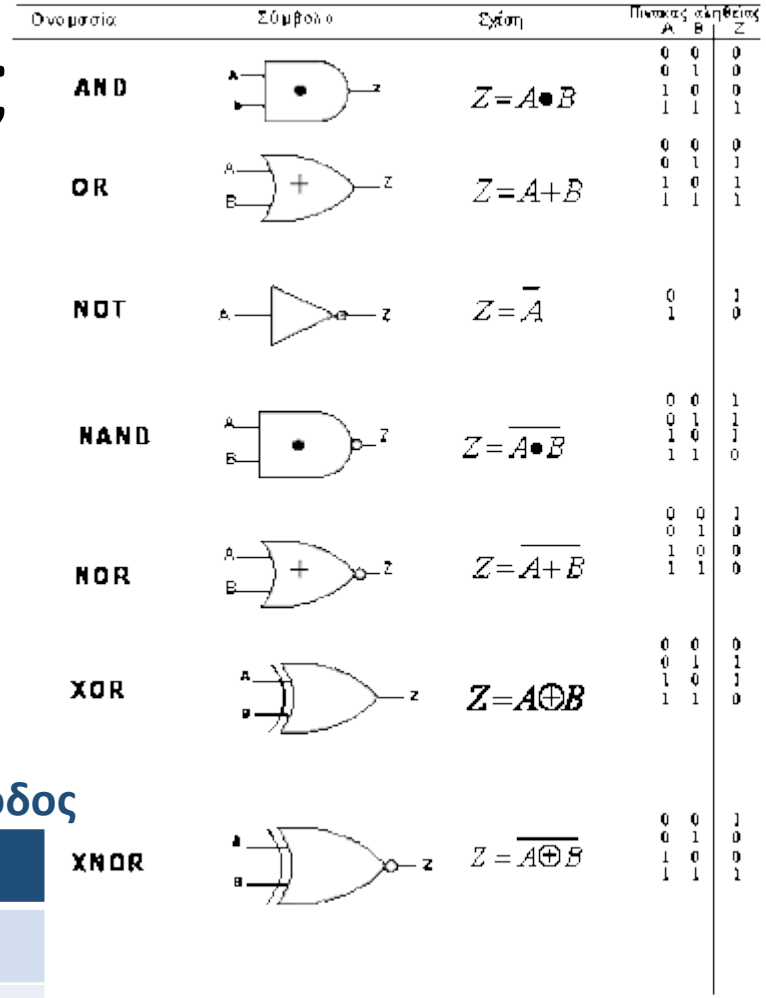
Παράδειγμα



Με γνωστό φόρτο εργασίας

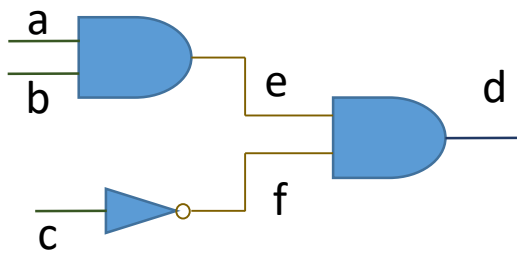
	φόρτος εργασίας			Ενδιάμεσες έξοδοι		Τελική έξοδος
χρόνος	a	b	c	e	f	d
t1	0	1	0	0	1	0
t2	1	1	0	1	1	1
t3	1	0	0	0	1	0
t4	0	0	1	0	0	0

Βασικές Λογικές Πύλες

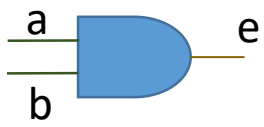


Ανάλυση και των ενδιάμεσων εξόδων

	φόρτος εργασίας			Ενδιάμεσες έξοδοι		Τελική έξοδος
χρόνος	a	b	c	e	f	d
t1	0	1	0	0	1	0
t2	1	1	0	1	1	1
t3	1	0	0	0	1	0
t4	0	0	1	0	0	0



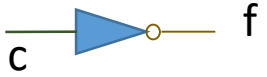
έξοδος e



t	a	b	e
t1	0	1	0
t2	1	1	1
t3	1	0	0
t4	0	0	0

Esw(e)=2/3

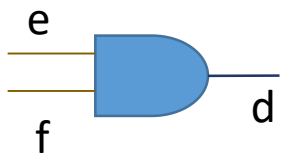
έξοδος f



t	c	f
t1	0	1
t2	0	1
t3	0	1
t4	1	0

Esw(f)=1/3

έξοδος d

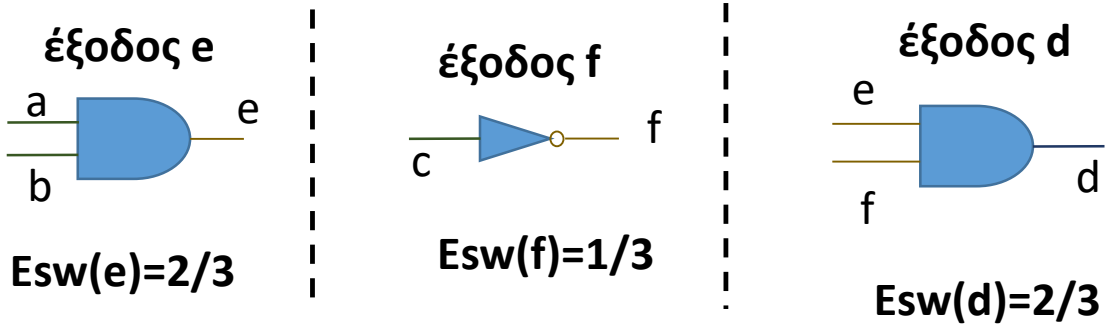


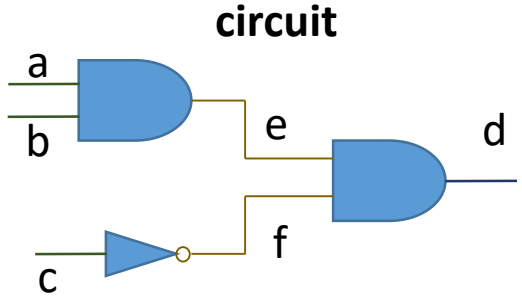
t	e	f	d
t1	0	1	0
t2	1	1	1
t3	0	1	0
t4	0	0	0

Esw(d)=2/3

Θυμηθείτε: **Esw=(# of switches) / (max # of switches)**

Ανάλυση και των ενδιάμεσων εξόδων





Πρέπει να εφαρμόσουμε τον παρακάτω τύπο σε κάθε στοιχείο ξεχωριστά με βάση το δικό του C_L ώστε να βρούμε τα Pd(e), Pd(f) και Pd(d)

$$P_d = V^2_{dd} * F_{clk} * C_L * E_{SW} * 0.5$$

F_{clk} = clock frequency, C_L = capacitance seen by gate,
E_{SW} = gate switching activity (0-1 toggles)

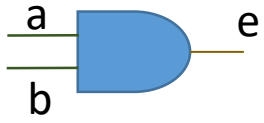
Και στο τέλος να βρούμε την τιμή:
Pdynamic(circuit)= Pd(e)+Pd(f) +Pd(d)

Σύνοψη: Για να βρούμε το switching activity ενός κυκλώματος με γνωστό φόρτο εργασίας πρέπει:

1. Να προσομοιώσουμε το κύκλωμα για το φόρτο εργασίας
2. Για κάθε έξοδο κάθε στοιχείου του κυκλώματος να υπολογίσουμε το switching activity

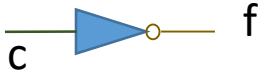
Ανάλυση και των ενδιάμεσων εξόδων

έξοδος e



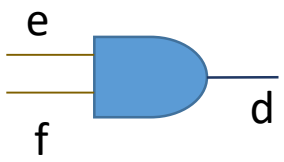
Esw(e)=2/3

έξοδος f



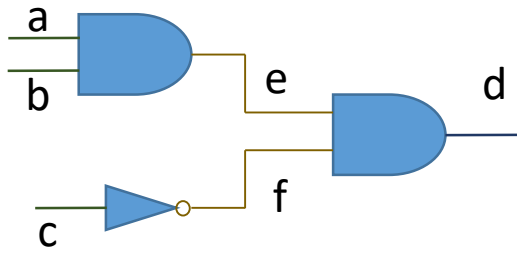
Esw(f)=1/3

έξοδος d



Esw(d)=2/3

circuit



Πρέπει να εφαρμόσουμε τον παρακάτω τύπο σε κάθε στοιχείο ξεχωριστά με βάση το δικό του C_L ώστε να βρούμε τα Pd(e), Pd(f) και Pd(d)

$$P_d = V^2_{dd} * F_{clk} * C_L * E_{SW} * 0.5$$

F_{clk} = clock frequency, C_L = capacitance seen by gate,
E_{SW} = gate switching activity (0-1 toggles)

Και στο τέλος να βρούμε την τιμή:
Pdynamic(circuit)= Pd(e)+Pd(f) +Pd(d)

```
//Αρχείο περιγραφής κυκλώματος
e =AND(a,b)
f  =NOT(c)
d =AND(e,f)
```

```
//Συνηθίζεται αυτή η μορφή αρχείου
//περιγραφής κυκλώματος
AND(e, a, b)
NOT(f, c)
AND(d, e, f)
```

Περιγραφή Στη Μνήμη και Προσομοίωση Κυκλωμάτων

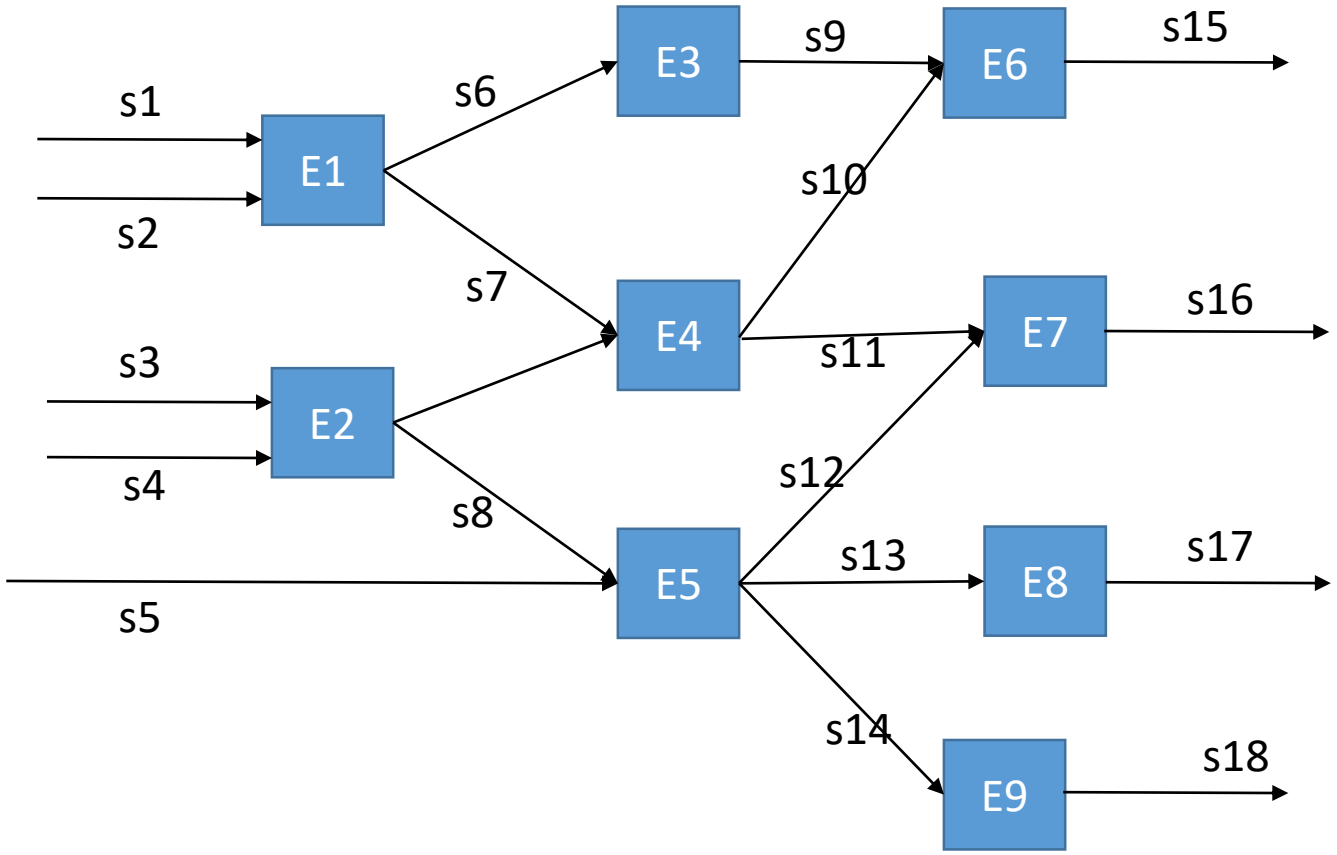
Έστω ότι έχουμε όλα τα Elements σε έναν πίνακα:

ElementsTable={E1, E2, E3, E4, E5, E6, E7, E8, E9} με τη σωστή σειρά που πρέπει να τα επεξεργαστούμε.

Ομοίως χρειαζόμαστε έναν πίνακα με τα signals:

SignalsTable={s1,s2,s3,s4,….,s18}

Και χρειαζόμαστε έναν τρόπο να ξέρουμε ποια Elements διαβάζουν ποια signals και ποια Elements γράφουν ποια signals.



hints:

1. ένα σήμα γράφεται ΜΟΝΟ από ΕΝΑ Element
2. Κάθε Element γράφει ΜΟΝΟ ένα signal (αυτό είναι απλούστευση, υπάρχουν πολύπλοκες λογικές πύλες με περισσότερες εξόδους)
3. Ένα SIGNAL μπορεί να διαβαστεί από πολλά Elements

Περιγραφή Πολύπλοκων Κυκλωμάτων στη Μνήμη

Και χρειαζόμαστε έναν τρόπο να ξέρουμε ποια Elements διαβάζουν ποια signals και ποια Elements γράφουν ποια signals.

//μπορεί να είναι πίνακας από κωδικοποιημένες τιμές (enumeration) ή πίνακας από strings

ElementTypes={NOT, AND, OR, XOR, NAND, NOR, XNOR}

// Πίνακας που κρατάει στοιχεία της δομής Element

ElementsTable={E1, E2, E3, E4, E5, E6, E7, E8, E9}

//πίνακας που κρατάει τις τιμές των σημάτων

SignalsTable={s1,s2,s3,s4,….,s18}

//Χρειαζόμαστε μια αναπαράσταση για το Element

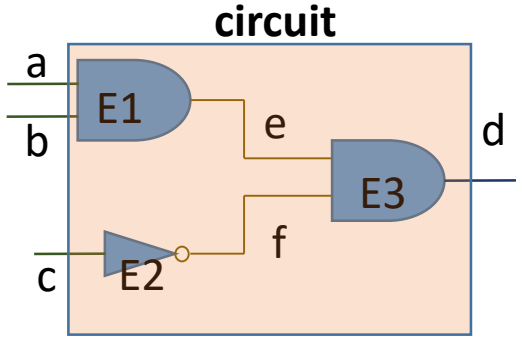
//μπορεί να είναι κλάση/δομή/κτλ.

Δομή Element

τιμή τύπος; //index προς το ElementTypes

πίνακας inputs; // πίνακας από indexes προς το SignalsTable

τιμή output; // index προς το SignalsTable



Παράδειγμα Περιγραφής:

SignalsTable={0,0,0,0,0,0}

//a,b,c,d,e,f

E1.type='AND'; E1.inputs=[1,2]; E1.output=5;

E2.type='NOT'; E2.inputs=[3]; E2.output=6;

E3.type='AND'; E3.inputs=[5,6]; E3.output=4;

ElementsTable={E1,E2,E3}

//μετά ακολουθεί το **processing**

for i=1:size(ElementsTable)

process(ElementsTable[i])

Σημείωση για αρχικοποίηση

Αν έχουμε βρει Top_inputs

Top_inputs=[1,2,3]

Τότε μπορούμε να αρχικοποίησουμε π.χ. σε a=1,b=0, c=1

Με τον εξής τρόπο

a=1 → **SignalsTable[1]=1**

b=0 -> **SignalsTable[2]=0**

c=1 → **SignalsTable[3]=1**

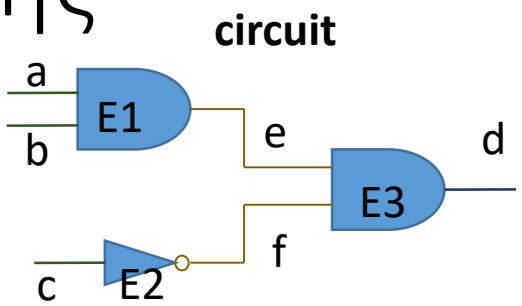
Το Βασικό Processing της Προσομοίωσης

Αν ο ElementsTable είναι ταξινομημένος τότε το simulation γίνεται με το σχήμα:

for i=1:size(ElementsTable)

process(ElementsTable[i])

Η *process* είναι μια συνάρτηση που κοιτάει το Type, τα inputs και outputs του Element που εκτελεί την κατάλληλη συνάρτηση μεταφοράς με τις κατάλληλες εισόδους εξόδους. Πρέπει να γράψει το αποτέλεσμα για κάθε Element στον πίνακα SignalsTable



Περιγραφή:

SignalsTable={0,0,0,0,0,0}

//a,b,c,d,e,f

E1.type='AND'; E1.inputs=[1,2]; E1.output=5;

E2.type='NOT'; E2.inputs=[3]; E2.output=6;

E3.type='AND'; E3.inputs=[5,6]; E3.output=4;

ElementsTable={E1,E2,E3}

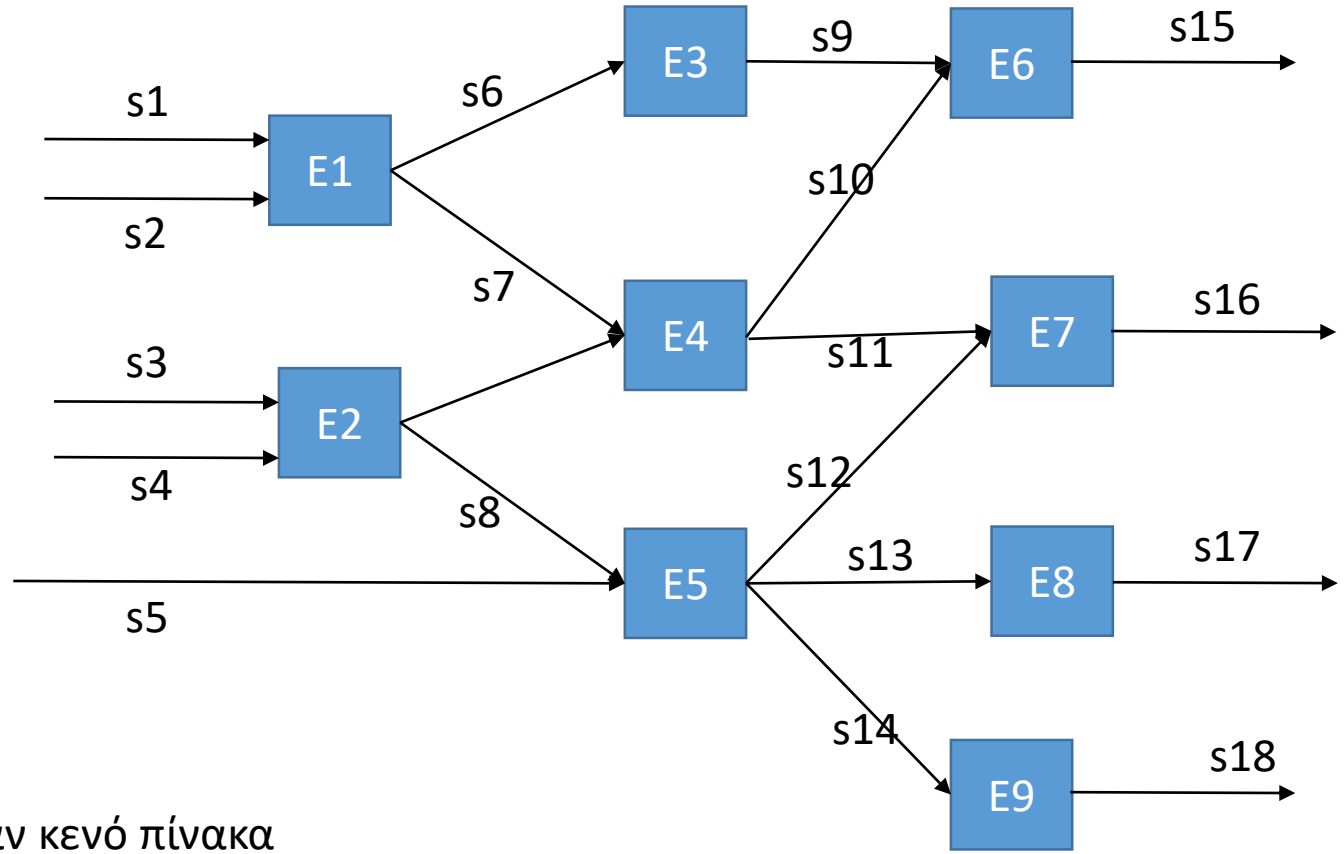
Παράδειγμα της process με ψευδοκώδικα:

```
function process(element)
 if(element.type=='AND')
      SignalsTable[element.output]=spAND(SignalsTable[element.input[1]], SignalsTable[element.input[2]])
 elseif(element.type=='NOT')
      SignalsTable[element.output]=spNOT(SignalsTable[element.input[1]])
```

Και αν ο Πίνακας Elements δεν είναι ταξινομημένος;

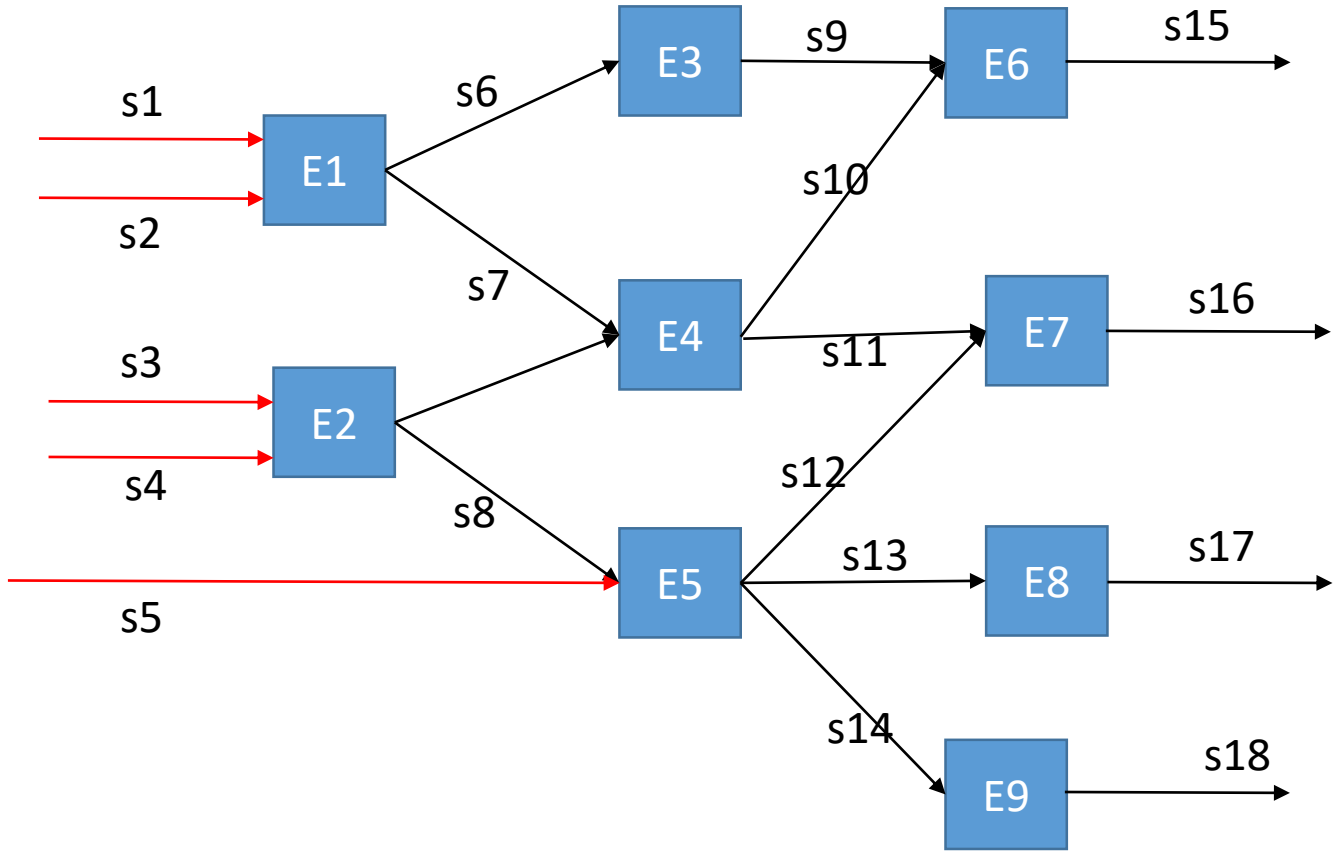
Όλα αυτά ισχύουν αν ο πίνακας δεν είναι ταξινομημένος με τη σωστή σειρά επεξεργασίας

Πως ταξινομούμε τον πίνακα;



Ξεκινάμε με έναν κενό πίνακα
ElementsTableSorted={}

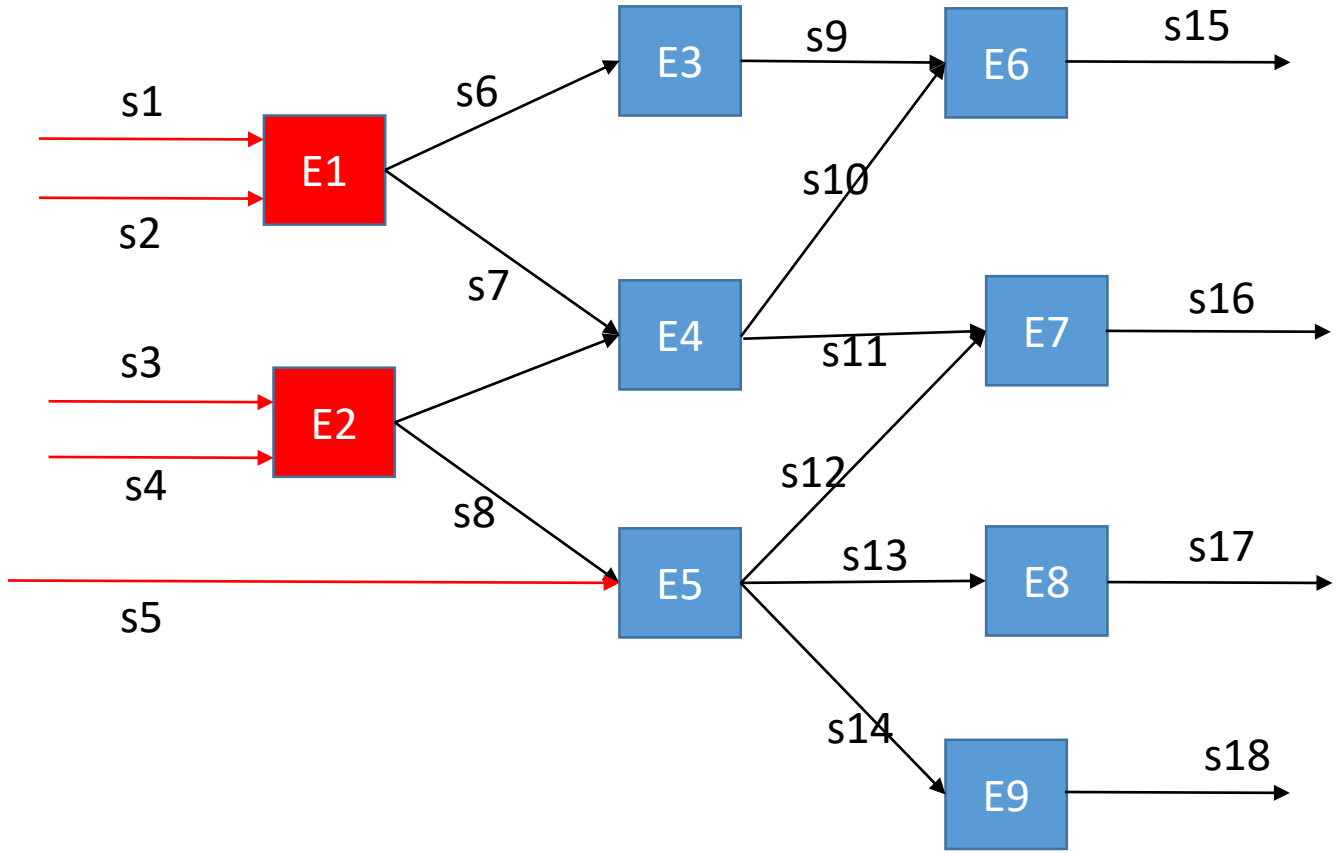
Και αν ο Πίνακας Elements δεν είναι ταξινομημένος;



ElementsTableSorted={}

Βήμα 1ο: μαρκάρουμε τα top level pin inputs

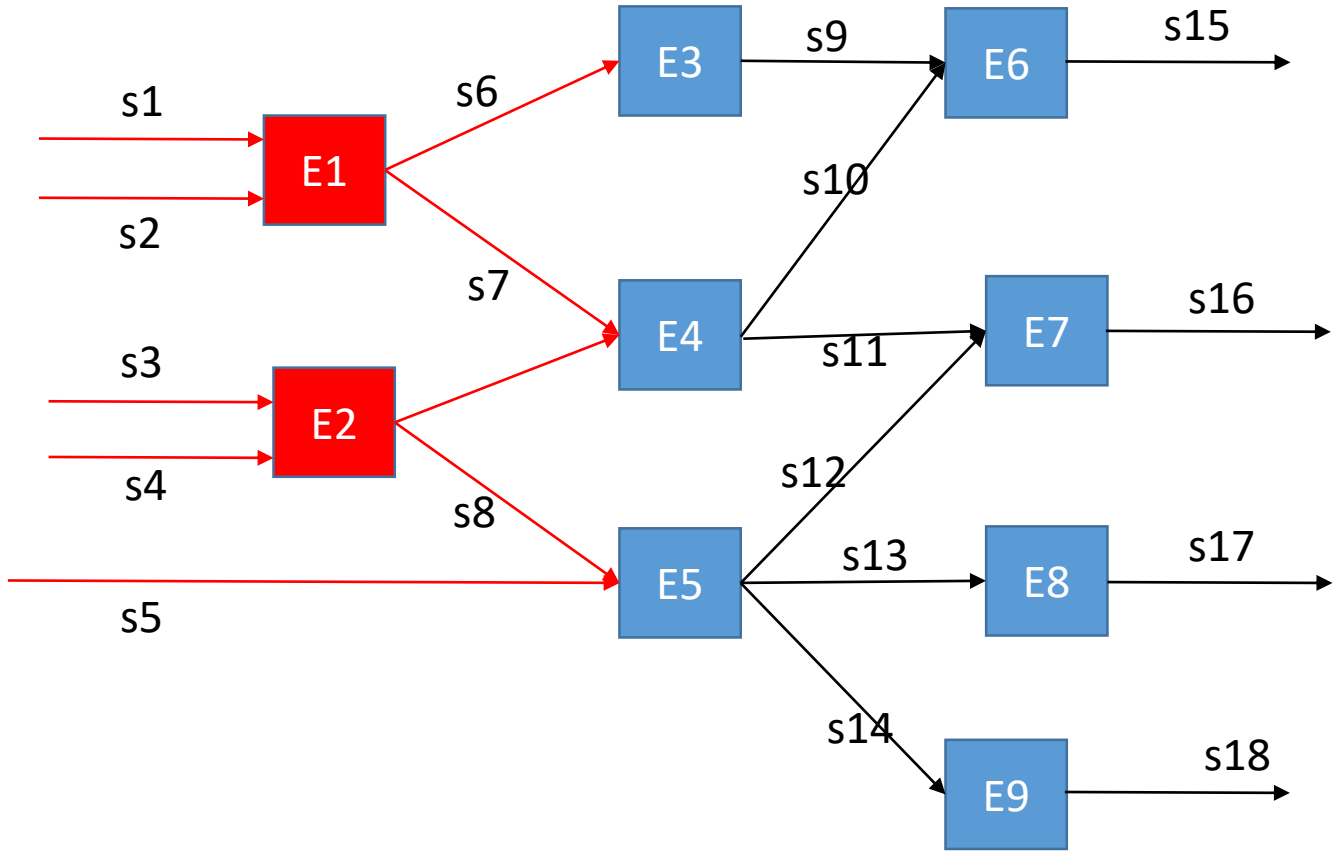
Και αν ο Πίνακας Elements δεν είναι ταξινομημένος;



Βήμα 2ο: μαρκάρουμε όλα τα elements των οποίων όλες οι είσοδοι είναι μαρκαρισμένες και τα βάζουμε μέσα στον πίνακα ElementsTableSorted

ElementsTableSorted={E1,E2}

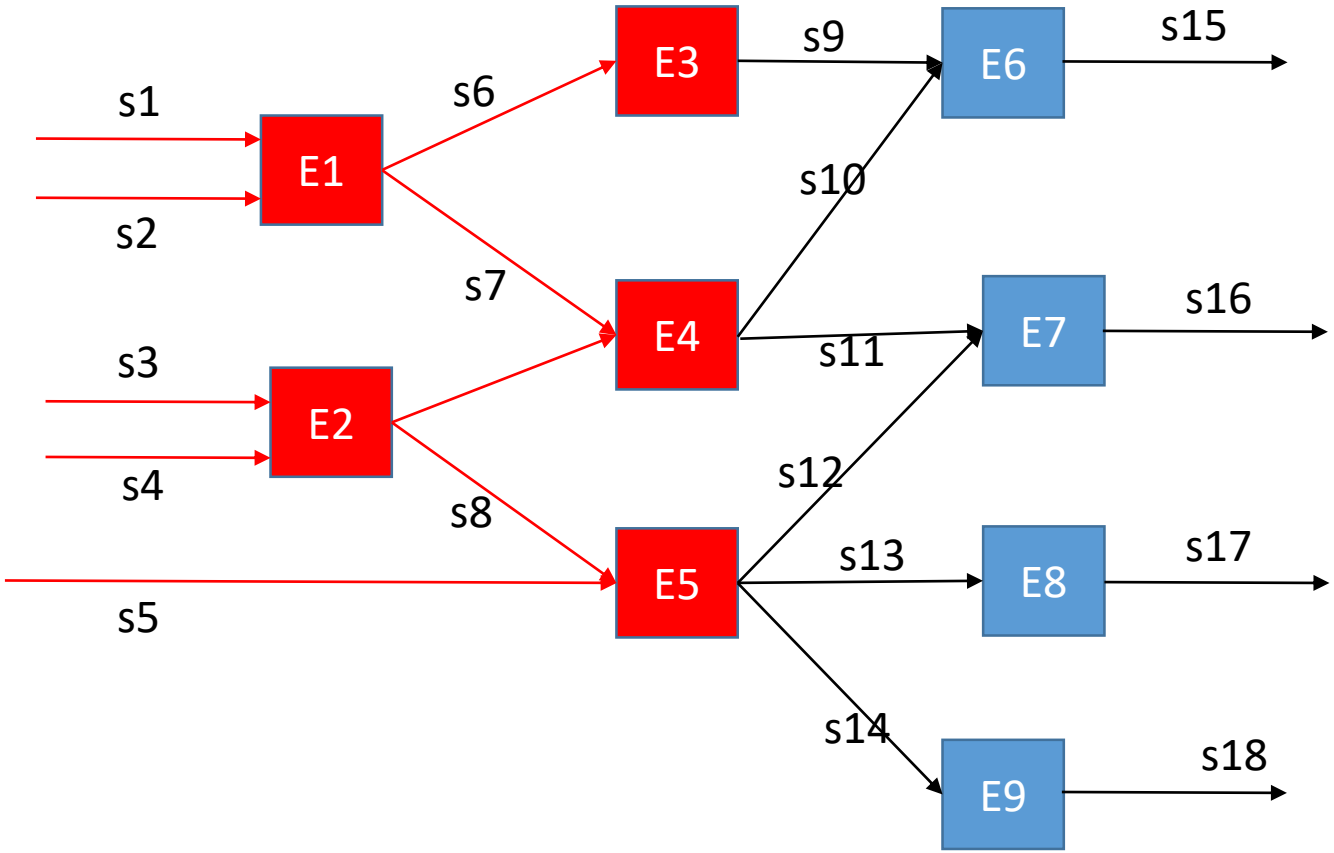
Και αν ο Πίνακας Elements δεν είναι ταξινομημένος;



Βήμα 3ο: μαρκάρουμε τις εξόδους των Elements που βάλαμε στον πίνακα ElementsTableSorted

ElementsTableSorted={E1,E2}

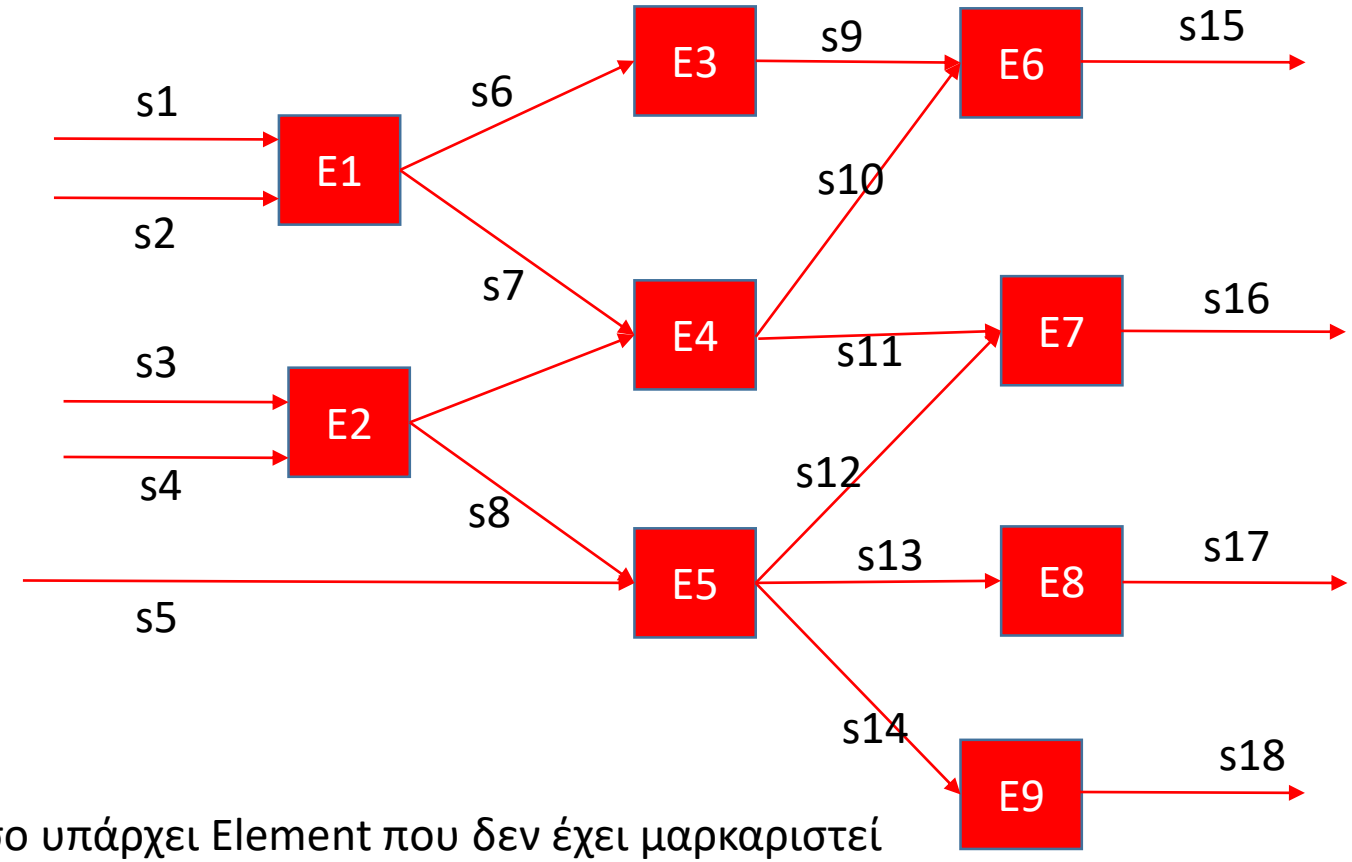
Και αν ο Πίνακας Elements δεν είναι ταξινομημένος;



Βήμα 4ο: μαρκάρουμε όλα τα NEA elements (που δεν έχουμε ήδη μαρκάρει) των οποίων όλες οι είσοδοι είναι μαρκαρισμένες και τα βάζουμε μέσα στον πίνακα ElementsTableSorted

ElementsTableSorted={E1,E2,E3,E4,E5}

Και αν ο Πίνακας Elements δεν είναι ταξινομημένος;



Για όσο υπάρχει Element που δεν έχει μαρκαριστεί

Εκτελούμε πάλι το 3ο Βήμα

Εκτελούμε πάλι το 4ο Βήμα

ElementsTableSorted={E1,E2,E3,E4,E5,E6,E7,E8,E9}

Static Power

so far we spoke only about power when a transistor is ON…
Well.. it still consumes power even when it's OFF!!!

That power is called **static power**

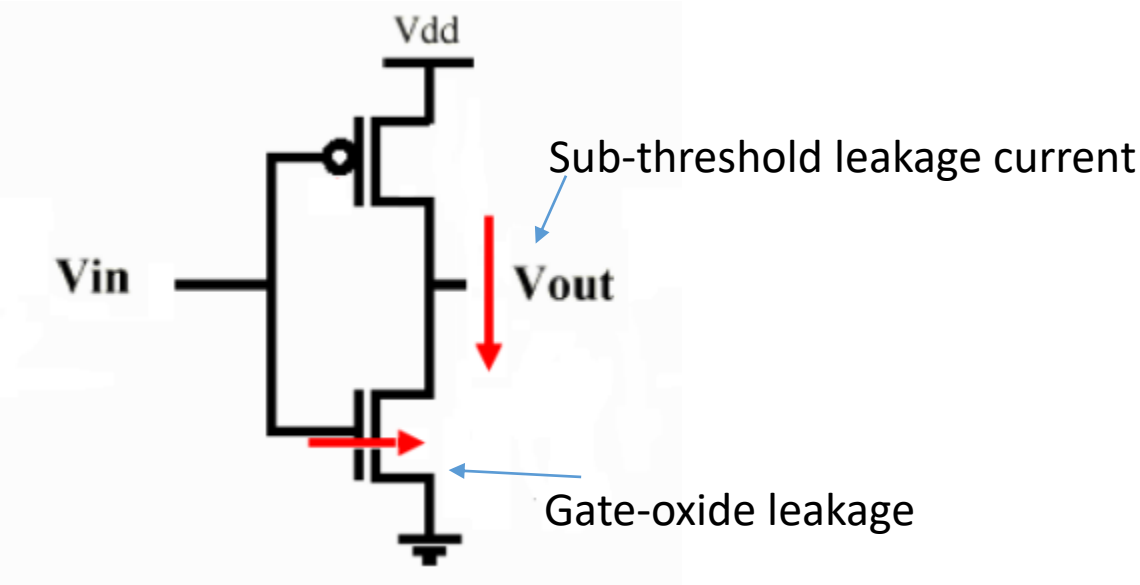
In fact it is:

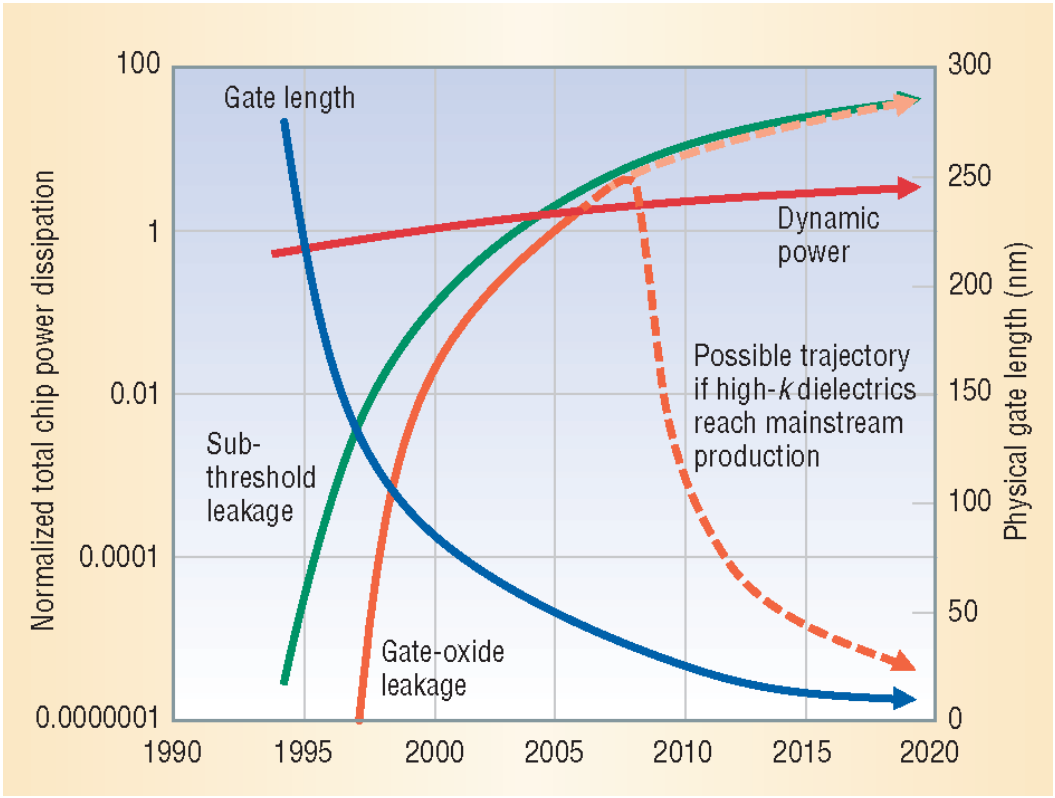
Power=(Dynamic Power) + (Short Circuit Power) + (Static Power)

We will talk about this later

Static power (caused by leakage currents) is typically responsible for 40% of the energy consumption. Thus, increasing the number of transistors increases power dissipation, even if the transistors are always off

Static power (caused by leakage currents)

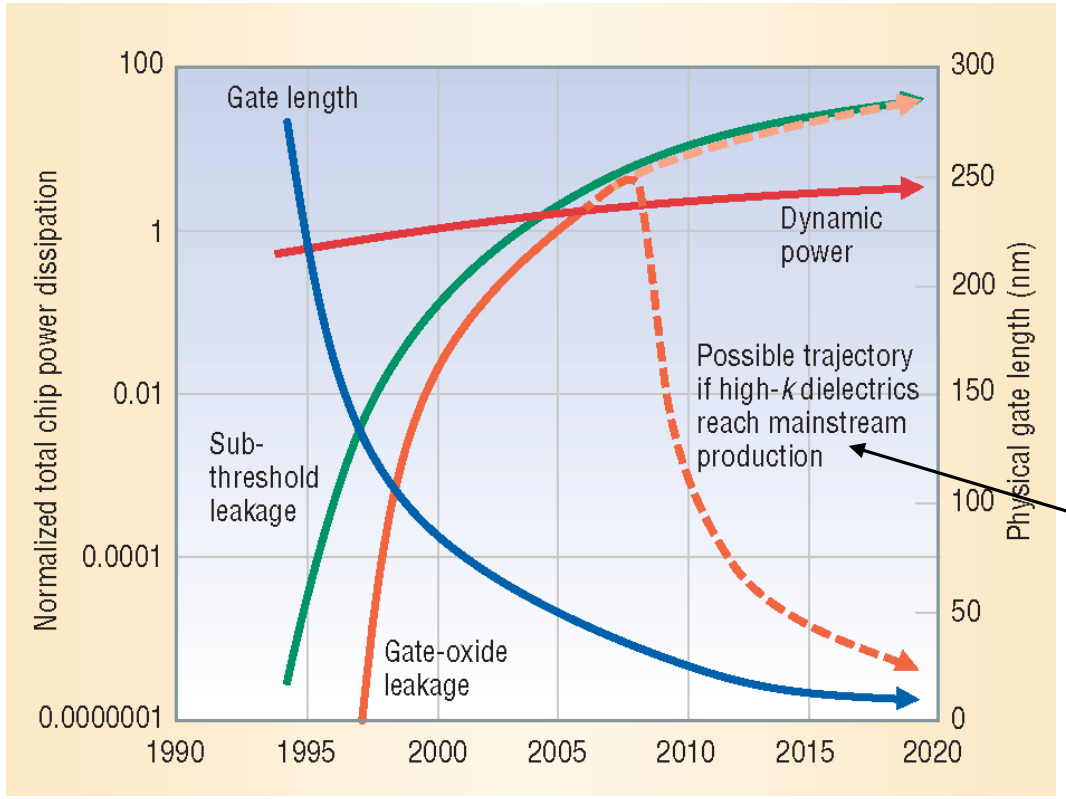


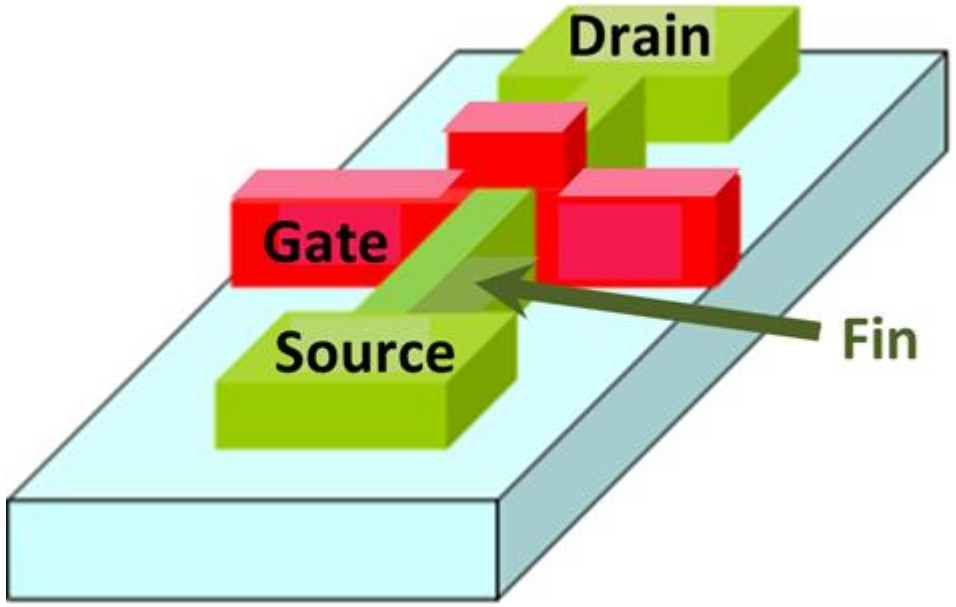


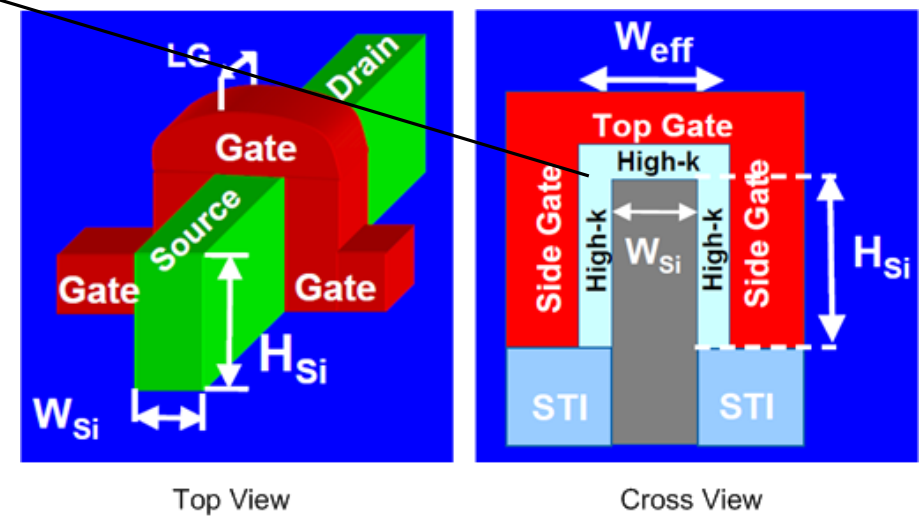
- Static power is as large as dynamic power for next generation ICs, unless specific countermeasures are taken
- Main components
  - Sub-threshold leakage current
  - Gate-oxide leakage current

$$P_{static} = V_{dd} * I_{leakage}$$

FINFET and High-k materials have tackled efficiently gate-oxide leakage current





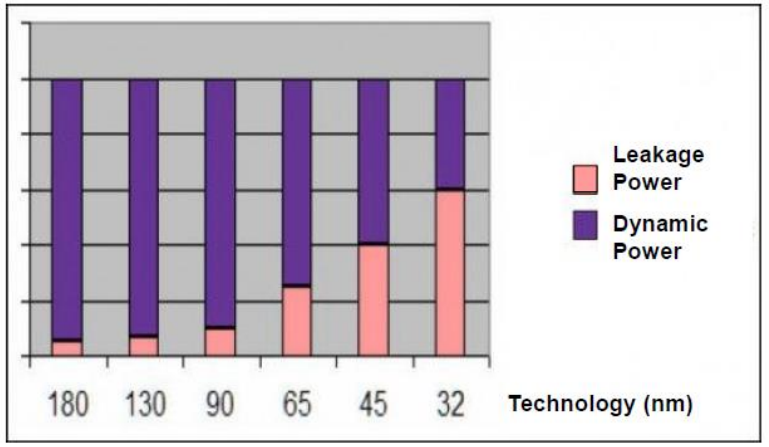


Top View

Cross View

Leakage power and threshold voltage background

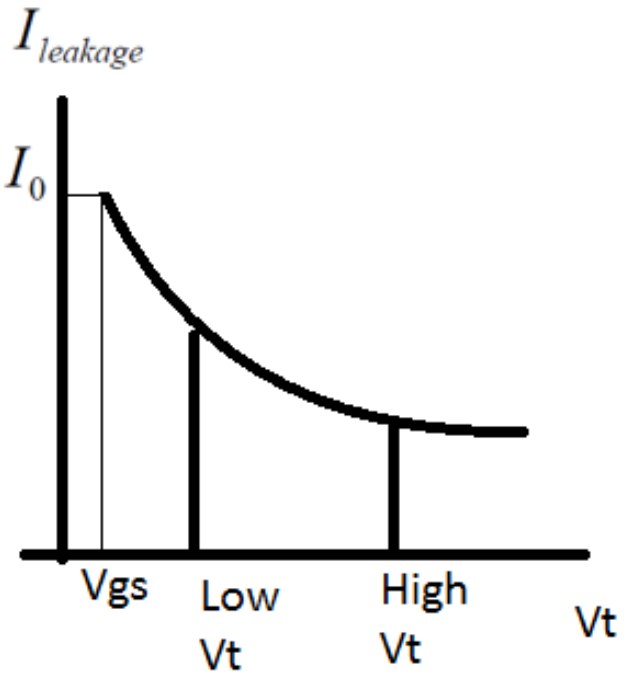
Subthreshold leakage



- Leakage Power consumption occurs as long as the circuit is powered on
- Sub-threshold current between source and drain in MOS transistor occurs when gate voltage (V_{gs}) is below transistor threshold voltage V_t
- sub-threshold current increases by 3x with each technology generation due to scaling of the transistor threshold voltage V_t
- At 45nm, total power (60% dynamic and 40% leakage)
- V_{dd} reduces with technology scaling (15% lower operating voltage, 30% smaller transistor), V_t must scale to deliver transistor performance.

$$P_{leakage} \approx V_{dd} . I_{leakage}$$

$$I_{leakage} \approx I_0 e^{\left(\frac{V_{gs}-V_t}{nV_T}\right)}, I_0 = \mu_0 C_{ox} \frac{W}{L}(n-1)V_T^2$$



Leakage Power Reduction

1. Technology Level
2. Stacking Effect
3. Body Bias
4. Adaptive Body Bias
5. Sleep Transistor (Power Gating)

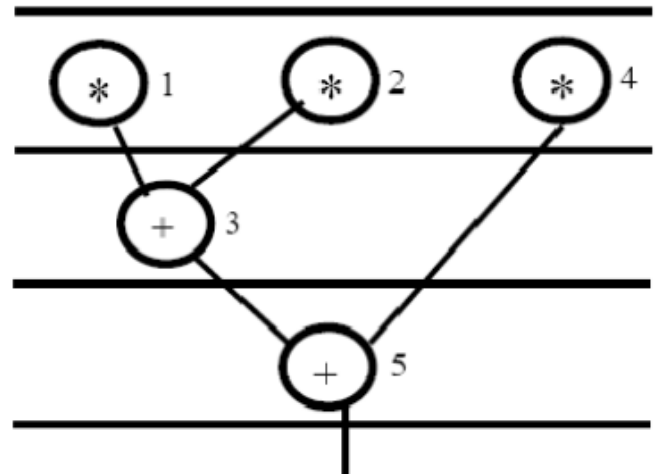
1.Technology level

Multiple V_t technology where fabrication process provides both high and low threshold transistor V_t (low: fast and leaky and high: slow and less leaky) for N and P transistors. Default design uses high V_t and careful replacing high V_t with low V_t, one can get low V_t performance and yet significantly lower leakage power.

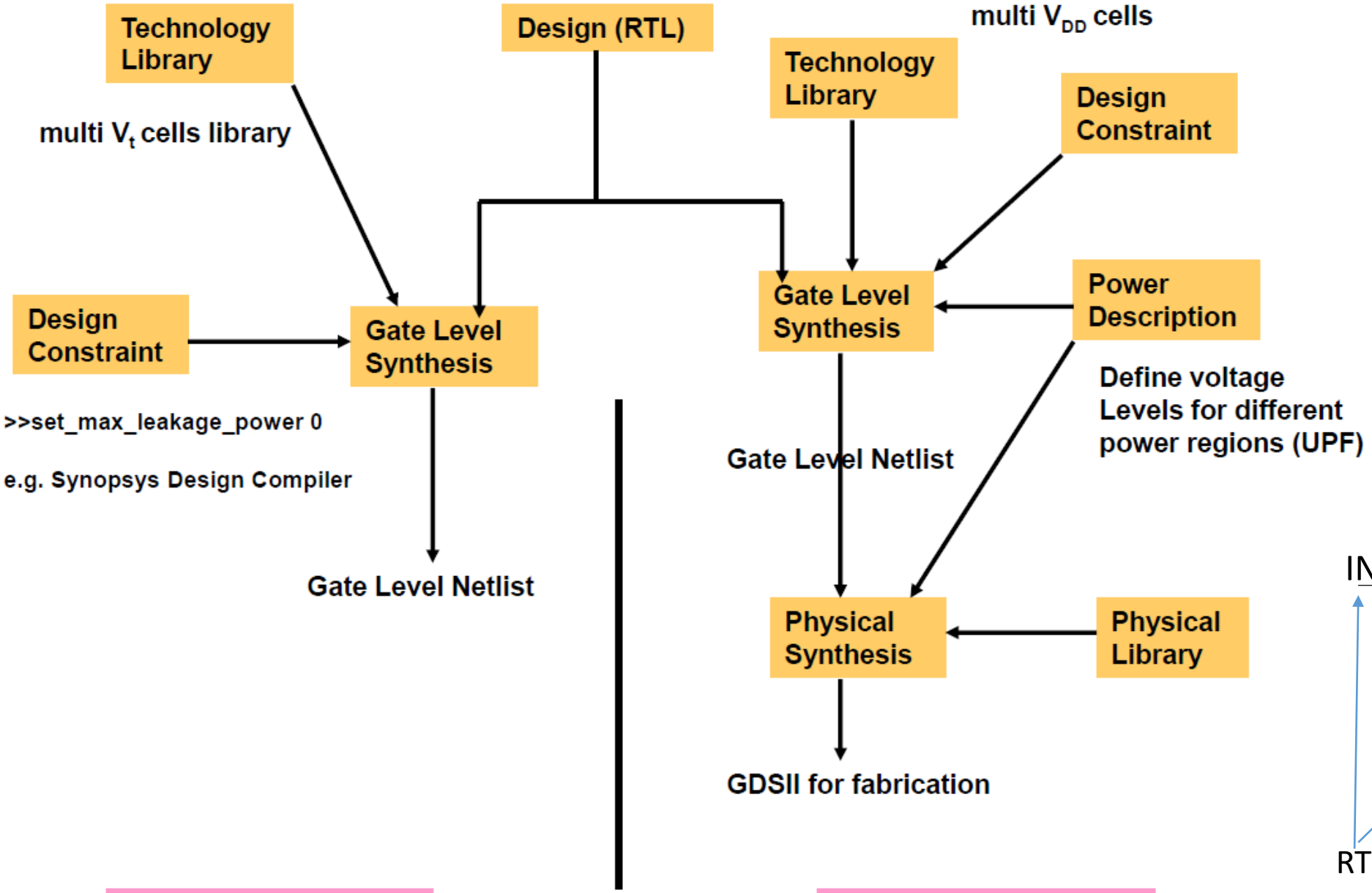
Synopsys Galaxy Design Platform provides an automated way of embedding *multi-V_t* to reduce leakage power

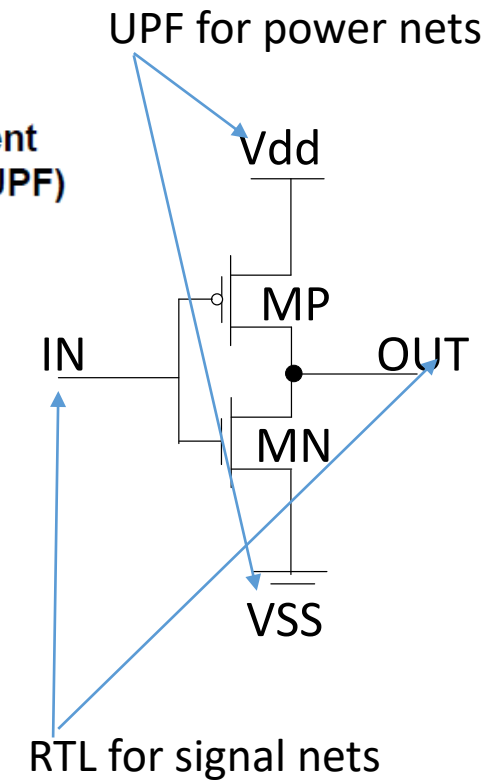
Example

critical path delay determines performance of the design, all other paths have lower delay. Use low V_t transistors in modules on the critical path, and high V_t transistors in modules on non-critical paths.



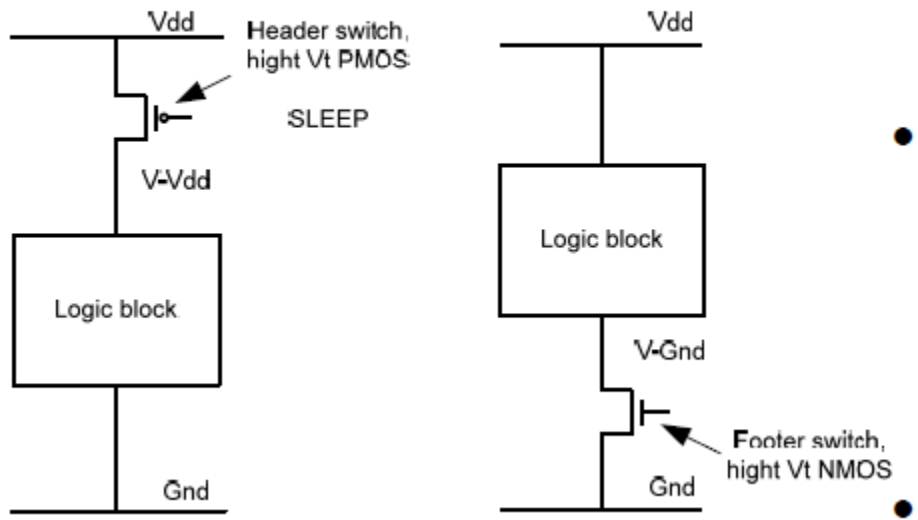
Multi Vt and Multi Vdd Design Flow





5. Power Gating

- Sleep transistors
  - Insert high V_t MOSFET between power rail and logic blocks
  - Header transistor off lead to logic block floating to near zero
  - Footer transistor off lead to logic block floating to near 1

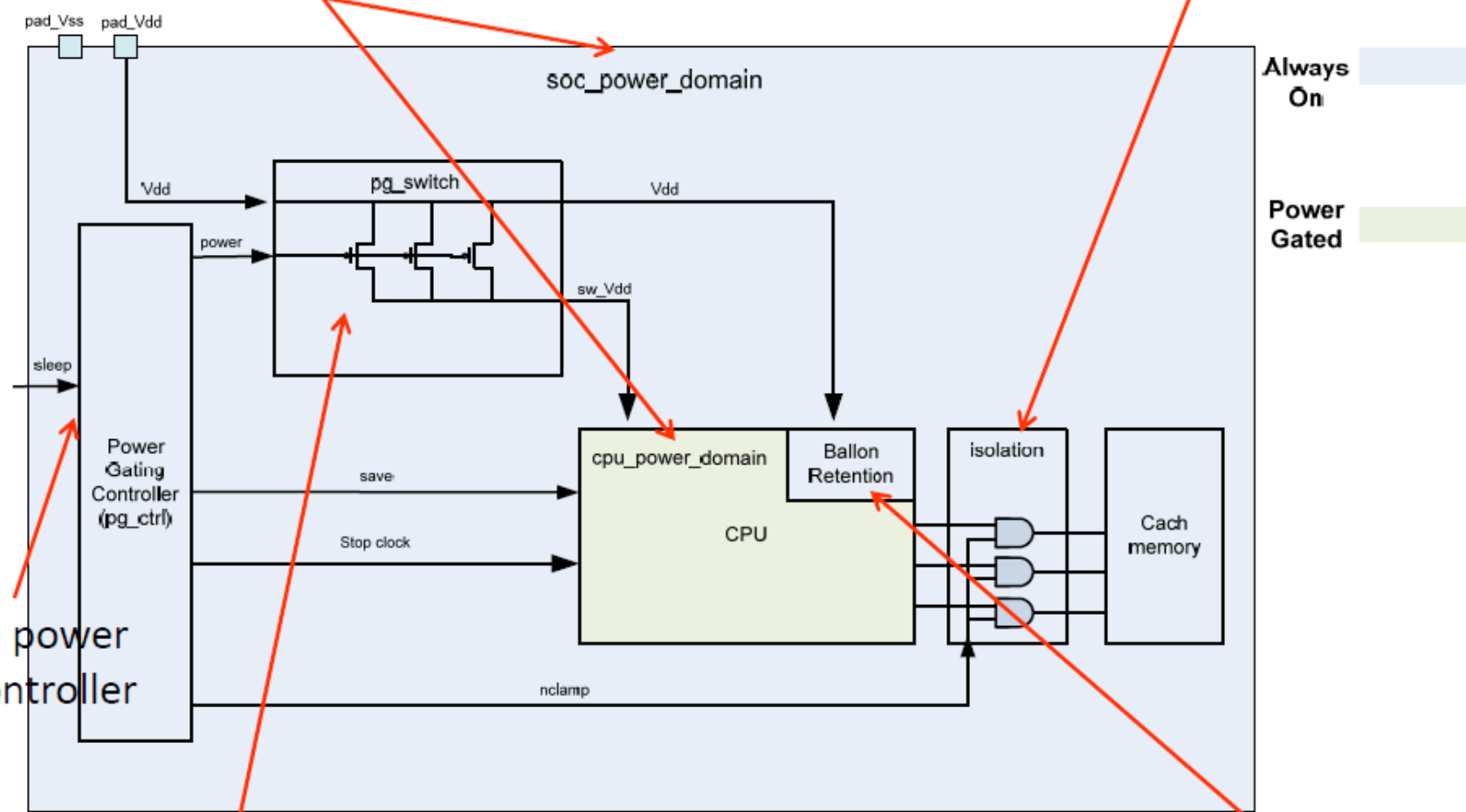


- Header switches preferred if multiple power rails since gated block logics float to zero irrespective of supply voltages
- See ref [8] for more details

Example: SoC with Power Gated CPU

1. Define power domain and create supply net for each power domain

3. Create output isolation



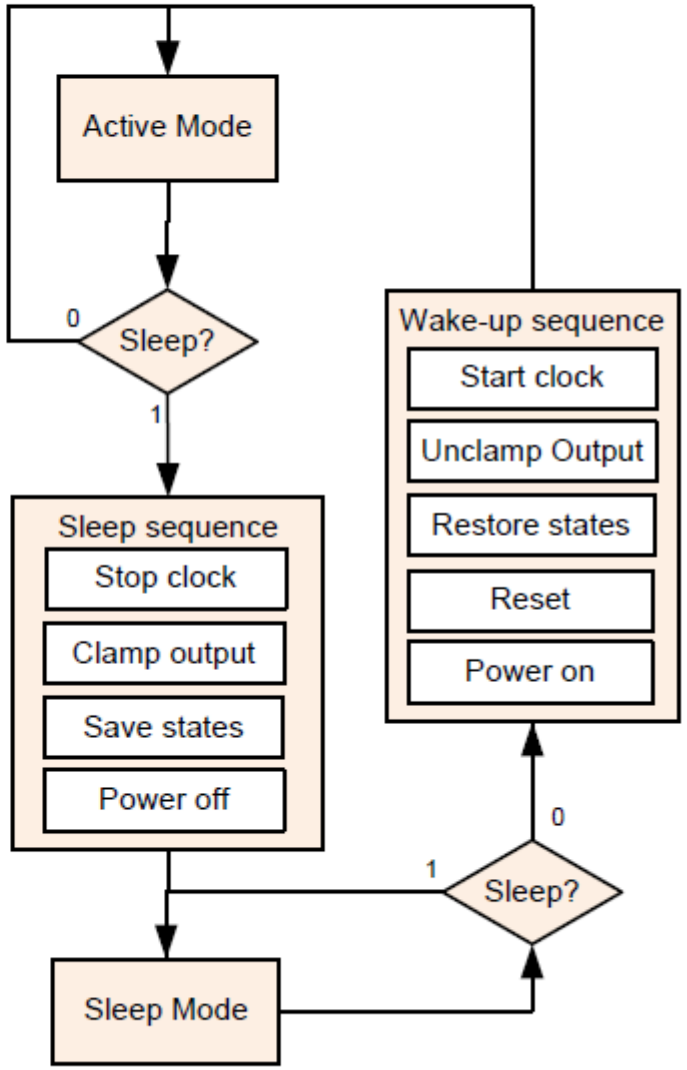
5. Create power gating controller

2. Create power switches

4. Create state retention

5. Create power gating controller FSM (RTL model) – Power Gating Protocol

- Require a controller to:
  - clamp the power gated block output and save the states before power off
  - Restore the states and unclamp the output after power on



Παραγωγή stress-tests
Προγράμματα που στρεσάρουν τη λειτουργία του επεξεργαστή μεγιστοποιώντας ποσοτικά κάποιον ποιοτικό δείκτη (π.χ. κατανάλωση ισχύος, εκλυόμενη θερμότητα, θόρυβος ισχύος κτλ.)

Εφαρμογές των stress-tests

Κατά την κατασκευή των πρωτοτύπων επεξεργαστών, stress-tests που μεγιστοποιούν το **θόρυβο** πάνω σε κυκλώματα, χρησιμοποιούνται για τον υπολογισμό της **ελάχιστης τάσης τροφοδοσία**ς.

Stress-tests που μεγιστοποιούν τη **θερμοκρασία** που εκλύεται από τον επεξεργαστή χρησιμοποιούνται για τον υπολογισμό του **συστήματος ψύξης**.

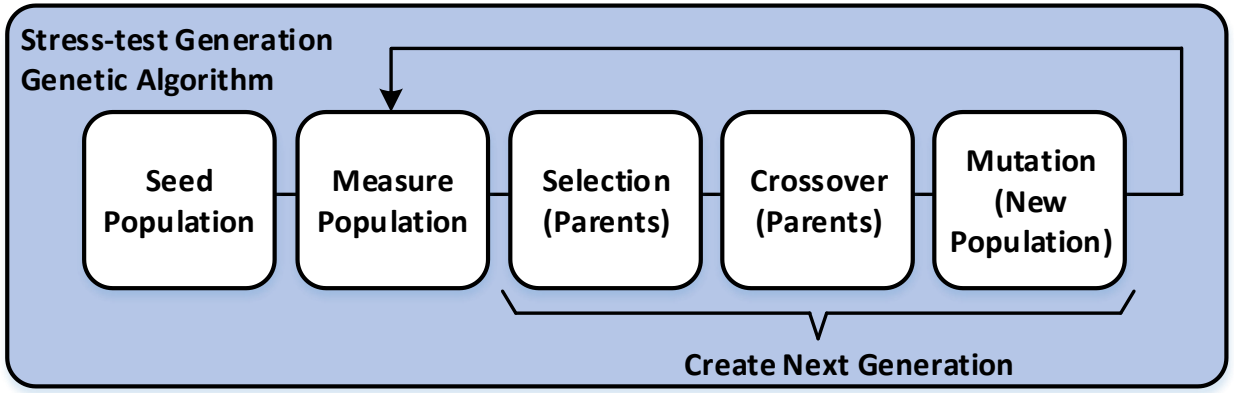
Stress-tests που μεγιστοποιούν την **κατανάλωση ενέργειας/ισχύος** του επεξεργαστή χρησιμοποιούνται για τον σχεδιασμό του **συστήματος τροφοδοσίας**.

Έχουν εφαρμογές επίσης στον χώρο της **ασφάλειας**.

Υπολογισμός μέγιστης δυναμικής κατανάλωσης

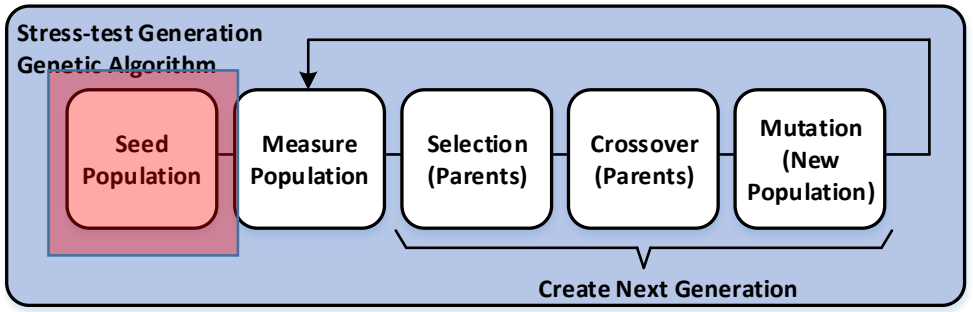
Δεν είναι εύκολος στόχος γιατί:
Χρειαζόμαστε έναν συστηματικό τρόπο να εντοπίσουμε το **workload** που προκαλεί τη **μέγιστη δυναμική κατανάλωση ισχύος**. Συγκεκριμένα αυτό το workload ονομάζεται **power stress-test**.

Η αυτοματοποιημένη μεθοδολογία παραγωγής stress-tests βασίζεται σε γενετικούς αλγορίθμους.



Η τεχνική μπορεί να εφαρμοστεί σε επεξεργαστές και λογικά κυκλώματα.

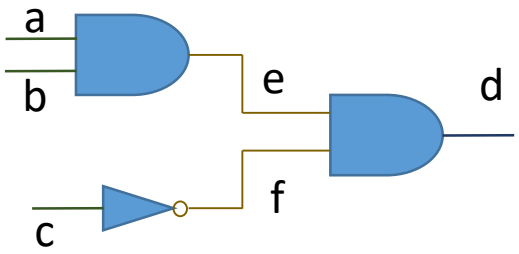
Βήμα 1ο: Αρχικός πληθυσμός (σπόρος – seed)



Υπενθυμίζουμε ότι το workload είναι μια χρονοσειρά από τιμές στις εισόδους του κυκλώματος.

Παράμετροι του αλγορίθμου

- **Μήκος φόρτου εργασίας L (workload length)**: είναι το μέγεθος μιας χρονοσειράς και το μετράμε σε πλήθος διανυσμάτων εισόδου.
- **Μέγεθος πληθυσμού N (population size)**: το πλήθος των ξεχωριστών φόρτων εργασίας που θα εξερευνάει ο αλγόριθμος σε κάθε του βήμα.



Αρχικά ξεκινάμε με έναν πληθυσμό από N τυχαία workloads. Κάθε ξεχωριστό workload (**individual workload**) είναι μια χρονοσειρά μήκους L.

π.χ. αρχικός πληθυσμός για L=3, N=4

Individual workload 1

t	a	b	c
t1	1	0	1
t2	0	0	1
t3	1	1	0

Individual workload 2

t	a	b	c
t1	0	0	1
t2	0	1	0
t3	1	0	1

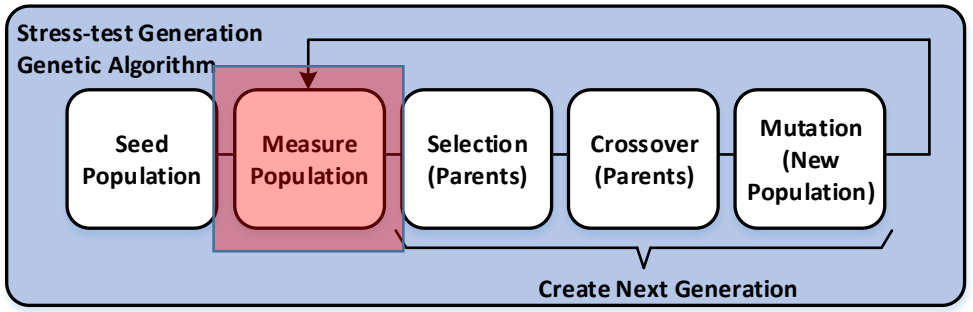
Individual workload 3

t	a	b	c
t1	0	1	1
t2	1	1	1
t3	0	0	1

Individual workload 4

t	a	b	c
t1	1	1	1
t2	0	0	0
t3	1	0	1

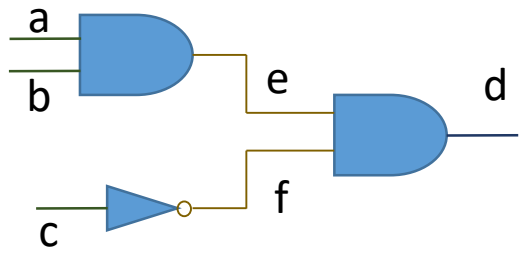
Βήμα 2ο: Μέτρηση της κατανάλωσης κάθε individual



Πρέπει να μετρήσουμε πόσο ένας individual πετυχαίνει **τον στόχο**.
ΜΕΤΡΑΜΕ ΤΟΝ ΣΤΟΧΟ

Στο στάδιο αυτό για κάθε individual στον πληθυσμό θα πρέπει να γίνει υπολογισμός της **δυναμικής κατανάλωσης ισχύος**.

Μπορεί να γίνει με υπολογισμό μέσω προσομοίωσης του switching activity κάθε individual.



π.χ. αρχικός πληθυσμός για L=3, N=4

Individual workload 1

t	a	b	c
t1	1	0	1
t2	0	0	1
t3	1	1	0

Individual workload 2

t	a	b	c
t1	0	0	1
t2	0	1	0
t3	1	0	1

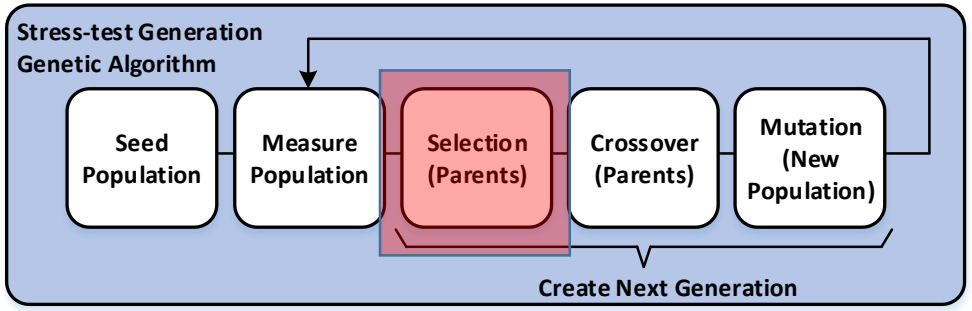
Individual workload 3

t	a	b	c
t1	0	1	1
t2	1	1	1
t3	0	0	1

Individual workload 4

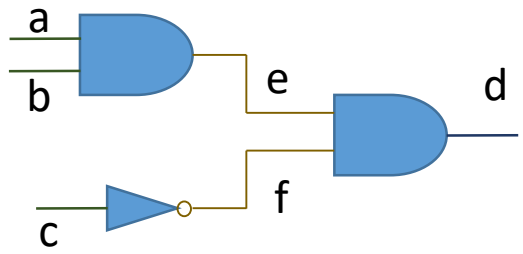
t	a	b	c
t1	1	1	1
t2	0	0	0
t3	1	0	1

Βήμα 3ο: Φυσική επιλογή



Στο στάδιο αυτό επιλέγουμε από τον πληθυσμό ως **γονείς** τα δύο individuals που πετυχαίνουν το καλύτερο αποτέλεσμα στον στόχο που έχουμε θέσει.

Στην προκειμένη περίπτωση επιλέγουμε τα 2 individual workloads με το μεγαλύτερο switching activity και τα ονομάζουμε **γονείς (parents)**.



Ας υποθέσουμε ότι αυτοί είναι οι 2 γονείς που επιλέγουμε

Individual workload 2

t	a	b	c
t1	0	0	1
t2	0	1	0
t3	1	0	1

Individual workload 3

t	a	b	c
t1	0	1	1
t2	1	1	1
t3	0	0	1

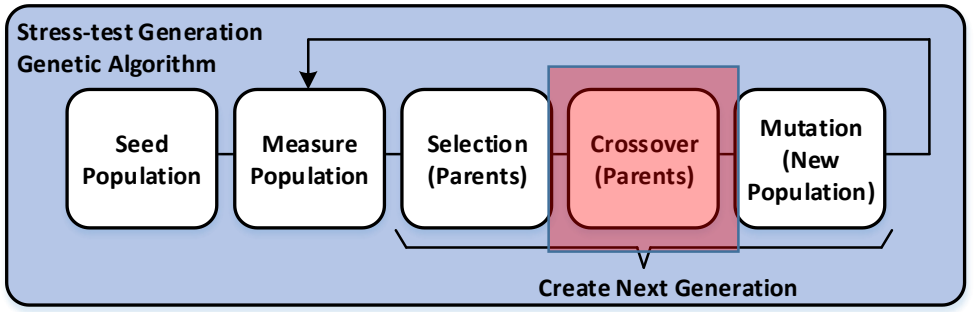
Individual workload 1

t	a	b	c
t1	1	0	1
t2	0	0	1
t3	1	1	0

Individual workload 4

t	a	b	c
t1	1	1	1
t2	0	0	0
t3	1	0	1

Βήμα 4ο: Διασταύρωση



Από τους 2 γονείς παράγουμε έναν νέο πληθυσμό.
Σίγουρα στο νέο πληθυσμό κρατάμε τους γονείς γιατί

- ως την ώρα δίνουν το καλύτερο αποτέλεσμα
- δεν είμαστε ακόμα βέβαιοι ότι τα παιδιά τους θα έχουν καλύτερο αποτέλεσμα

Ας υποθέσουμε ότι αυτοί είναι οι 2 γονείς που επιλέγουμε

Γονέας Α

t	a	b	c
t1	0	0	1
t2	0	1	0
t3	1	0	1

Γονέας Β

t	a	b	c
t1	0	1	1
t2	1	1	1
t3	0	0	1

Individual workload 1
2nd generation

t	a	b	c
t1	0	0	1
t2	0	1	0
t3	1	0	1

Individual workload 2
2nd generation

t	a	b	c
t1	0	1	1
t2	1	1	1
t3	0	0	1

Τα υπόλοιπα «**παιδιά**» παράγονται με διασταύρωση. Μπορεί να γίνει με διάφορους τρόπους. Εμείς θα χρησιμοποιήσουμε τη τυχαία πρόσμιξη με διαίρεση. Για κάθε individual/παιδί που θα παράγουμε, θα επιλέγουμε μια **τυχαία γραμμή διαχωρισμού R** από 1 έως L-1. Οι πρώτες R γραμμές θα επιλέγονται από τον έναν γονέα και οι υπόλοιπες L-R θα επιλέγονται από τον άλλον. Επίσης μπορούμε να αλλάζουμε τυχαία από ποιον γονέα θα επιλέγουμε τις αρχικές γραμμές ρίχνοντας ένα νόμισμα C.

Παράμετροι R,C του αλγορίθμου

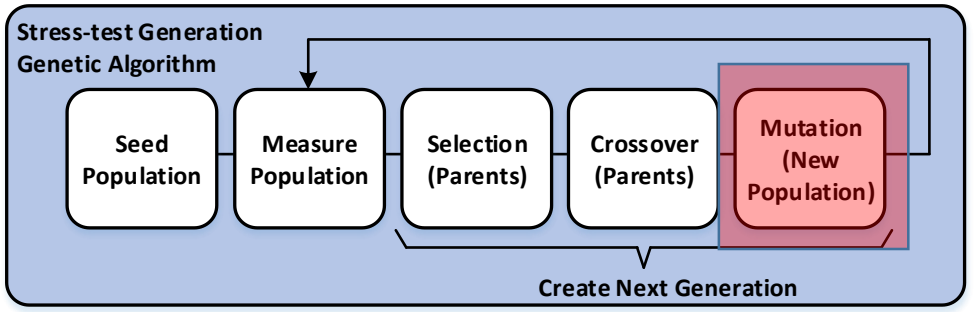
C=1, R=2
(όταν C=1, τότε 1ος γονέας ο Α)
Individual workload 3
2nd generation

t	a	b	c
t1	0	0	1
t2	0	1	0
t3	0	0	1

C=2, R=1
(όταν C=2, τότε 1ος γονέας ο Β)
Individual workload 4
2nd generation

t	a	b	c
t1	0	1	1
t2	0	1	0
t3	1	0	1

Βήμα 5ο: Μετάλλαξη



Με μια πολύ μικρή πιθανότητα που ονομάζεται **συντελεστής μετάλλαξης** (**mutation rate**) **m** αλλάζουμε τα bits των παιδιών που προήλθαν από διαίρεση στο προηγούμενο βήμα. Δεν μεταλλάσσουμε τους δύο γονείς της γενιάς. Το m μπορεί να είναι πολύ μικρό π.χ. m=0.01

Individual workload 1
2nd generation

t	a	b	c
t1	0	0	1
t2	0	1	0
t3	1	0	1

Individual workload 2
2nd generation

t	a	b	c
t1	0	1	1
t2	1	1	1
t3	0	0	1

C=1, R=2
Individual workload 3
2nd generation

t	a	b	c
t1	0	0	1
t2	0	1	0
t3	0	0	1

C=2, R=1
Individual workload 4
2nd generation

t	a	b	c
t1	0	1	1
t2	0	1	0
t3	1	0	1

Δεν μεταλλάσσονται

Μεταλλάσσονται

Individual workload 1
2nd generation

t	a	b	c
t1	0	0	1
t2	0	1	0
t3	1	0	1

Individual workload 2
2nd generation

t	a	b	c
t1	0	1	1
t2	1	1	1
t3	0	0	1

Mutated individual 3
2nd generation

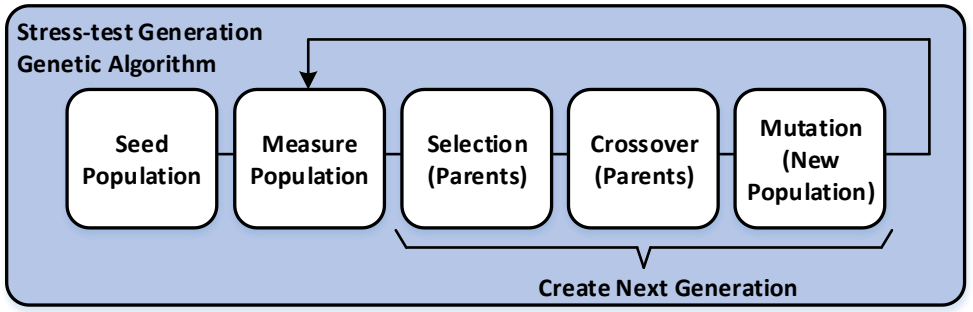
t	a	b	c
t1	0	0	1
t2	0	0	0
t3	0	0	1

Mutated individual 4
2nd generation

t	a	b	c
t1	0	1	0
t2	0	1	0
t3	1	0	1

Παράμετρος του αλγορίθμου m

Σύνοψη παραμέτρων



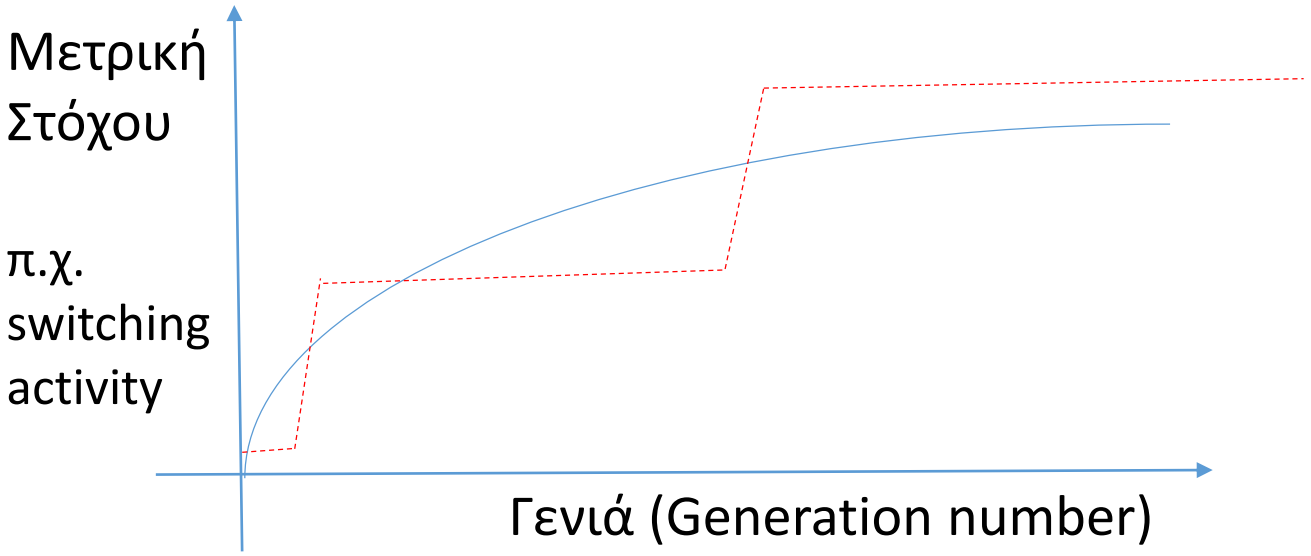
Παράμετροι του αλγορίθμου

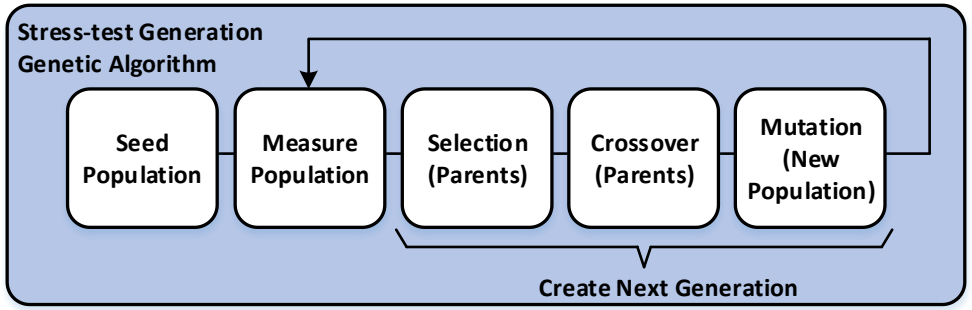
- **Μήκος φόρτου εργασίας L (workload length)**: είναι το μέγεθος μιας χρονοσειράς και το μετράμε σε πλήθος διανυσμάτων εισόδου.
- **Μέγεθος πληθυσμού N (population size)**: το πλήθος των ξεχωριστών φόρτων εργασίας που θα εξερευνάει ο αλγόριθμος σε κάθε του βήμα.
- **Ρυθμός μεταλλάξεων m (mutation rate):** είναι μια πιθανότητα κάποια bits από κάποια παιδιά να αλλάξουν.

Τυχαίες μεταβλητές

- **Τυχαία γραμμή διαχωρισμού R διασταυρώσεων:** είναι το πλήθος των γραμμών που θα επιλεγούν από τον πρώτο γονέα κατά τη διαδικασία της κατασκευής νέων **παιδιών.**
- **Τυχαία νόμισμα C επιλογής πρώτου γονέα:** από τον γονέα αυτών θα επιλέγονται οι πρώτες γραμμές κάποιου παιδιού. Σε κάθε παιδί αλλάζει ο πρώτος τυχαία ο πρώτος γονέας.

Συνεχίζουμε έως ότου δεν υπάρχει βελτίωση





- Στον αλγόριθμο που δείξαμε μπορούμε να βάλουμε διάφορες μετρικές στόχους (objective functions) στο βήμα measure για να βελτιστοποιήσει.
- Δουλεύει όχι μόνο για λογικά κυκλώματα αλλά και για προγράμματα μικροεπεξεργαστών, δηλαδή εντολές assembly
  - Μπορείτε να σκεφτείτε εφαρμογές με άλλες objective functions σε επεξεργαστές;

Σχεδίαση με Verilog και ο MicroCPU

Διάγραμμα Ροής Σχεδιασμού Συστημάτων
Παράδειγμα 1 – Hello World
Παράδειγμα 2 – 4bit counter
Αναπαράσταση αριθμών
Παράδειγμα 3 – full adder -1bit counter
Παράδειγμα 4. 4bit-Adder
Παράδειγμα 5. Parity checker
Παράδειγμα 6: unconnected ports
Behavioural Modeling
Μοντελοποίηση Συμπεριφοράς

Παραμετροποιήσιμα Modules
Δηλώσεις Συνθήκης
Δηλώσεις Επανάληψης
Δηλώσεις συνεχούς ανάθεσης
Single-bit Arithmetic logic unit

Βασική Αρχιτεκτονική Επεξεργαστών
Pipelining

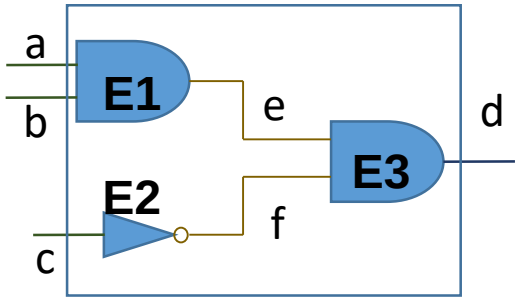
Σχεδίαση σε Verilog επεξεργαστή

- **Μοντελοποίηση Μνήμης και Σχεδίαση του Ελεγκτή Μνήμης**
- **Σχεδίαση Αριθμητικής Λογικής Μονάδας**
- **Σχεδίαση Πεπερασμένου Αυτόματου Μονάδας Ελέγχου**

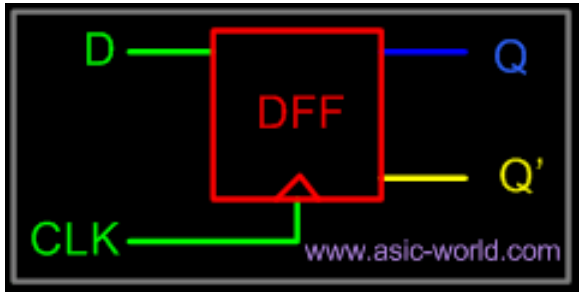
Μορφές/Αναπαραστάσεις Σχεδιασμού με Verilog

Η Verilog μας επιτρέπει να σχεδιάσουμε στα εξής επίπεδα αφαίρεσης:

- **Behavioral level:** η συμπεριφορά περιγράφεται με αλγορίθμους. Κάθε αλγόριθμος αποτελείται από διαδοχικές εντολές. Functions, Tasks και Always είναι τα βασικά στοιχεία στον κώδικα Verilog αυτού του επιπέδου.
- **Register-Transfer Level (RTL)**: Περιγράφονται τα χαρακτηριστικά του κυκλώματος (σήματα) και η μεταφορά των δεδομένων ανάμεσα σε καταχωρητές. Χαρακτηριστικό αυτού του επιπέδου είναι τα σαφή όρια χρονισμού των λειτουργιών κάθε στοιχείου, των σημάτων και των καταχωρητών. Γενικά μπορούμε να πούμε ότι «κάθε κώδικας που συντίθεται είναι RTL κώδικας».
- **Gate Level**: Όπως και στην RTL μορφή, κάθε σήμα και καταχωρητής και οι σχέσεις τους είναι σαφή, όπως και ο χρονισμός τους. Τα σήματα μπορούν να έχουν μόνο (`0', `1', `X', `Z`) τιμές. Τα elements μπορούν να είναι μόνο πρωτογενείς πύλες (AND, OR, NOT κτλ.). Συνήθως δεν γράφουμε σε αυτό το επίπεδο αλλά παράγεται αυτόματα από εργαλεία σύνθεσης.

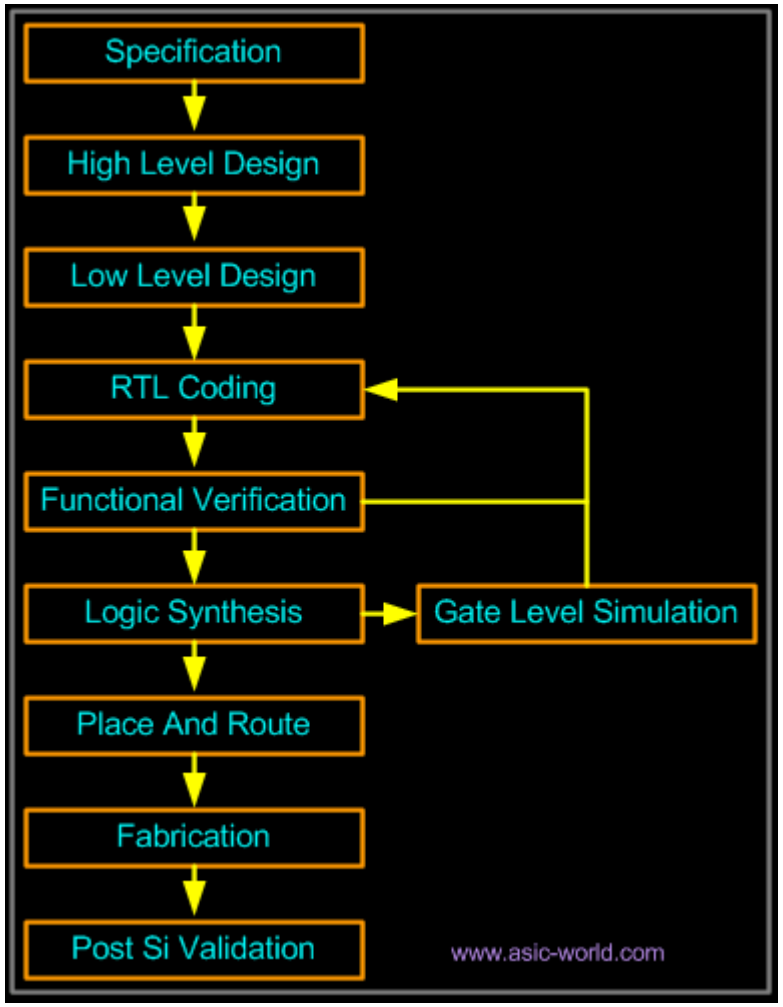



```
module example1(a,b,c,d);
    input a,b,c;
    output d;
    wire a,b,c,d,e,f;

    and E1(e,a,b);
    not E2(f,c);
    and E3(d,e,f);

endmodule
```




```
// D flip-flop Code
module d_ff ( d, clk, q, q_bar);
    input d ,clk;
    output q, q_bar;
    wire d ,clk;
    reg q, q_bar;
    always @ (posedge clk)
    begin
        q <= d;
        q_bar <= ! d;
    end
endmodule
```

Τυπικό Διάγραμμα Ροής Σχεδιασμού Ψηφιακών Κυκλωμάτων



Εργαλεία που απαιτούνται για κάθε στάδιο σχεδιασμού:
Specification: Επεξεργαστής κειμένου π.χ. Word, Open Office.
High Level Design: Επεξεργαστής κειμένου π.χ. Word, Open Office. Εργαλεία προβολής χρονοσειρών π.χ. waveformer ή testbencher ή ακόμα και στο Word.
Micro Design/Low level design: Επεξεργαστής κειμένου π.χ. Word, Open Office. Εργαλεία προβολής χρονοσειρών π.χ. waveformer ή testbencher ή ακόμα και στο Word.
RTL Coding: Ο αγαπημένος σας Text Editor π.χ. Notepad++, Vim, Emacs
Simulation: Modelsim, VCS, Vivado, Nsim, Verilog-XL, Veriwell, Finsim, Icarus.
Synthesis: Design Compiler, Leonardo Spectrum, Quartus, Vivado. FPGA vendors όπως η Intel και η Xilinx δίνουν τέτοια εργαλεία δωρεάν.
Place & Route: For FPGA specific P&R. ASIC απαιτούν ακριβά P&R εργαλεία όπως το IC compiler και το Encounter.
Post Si Validation: Για Application Specific ICs (ASIC) and Field Programmable Gate Arrays (FPGAs), το Τσιπ ελέγχετε σε κανονικές συνθήκες σε Motherboard με drivers των συσκευών.

Τα στάδια σχεδιασμού/κατασακευής μικρεπεξεργαστών

Specification: Το στάδιο που καθορίζονται οι βασικοί παράμετροι του συστήματος/σχεδιασμού μας.
Εργαλεία: Επεξεργαστής κειμένου π.χ. Word, Open Office.
Παράδειγμα: θέλω να σχεδιάσω έναν counter, οποίος θα προσθέτει σήματα των 4 bit με σύγχρονο reset και θα είναι ακμοπυροδότητος (active high enable). Όταν το reset είναι ενεργό θα μηδενίζεται.

High Level Design: Διαχωρισμός του σχεδίου σε δομικά τμήματα με βάση τη λειτουργία τους και καθορισμός της επικοινωνίας τους. Προσοχή είναι γενικό και σχηματικό το στάδιο αυτό. Δεν εξετάζει πολλές λεπτομέρειες.
Εργαλεία: Επεξεργαστής κειμένου π.χ. Word, Open Office. Εργαλία σχεδίασης διαγραμμάτων π.χ. Powerpoint, visio. Εργαλεία προβολής χρονοσειρών π.χ. waveformer ή testbencher ή ακόμα και στο Word.
Παράδειγμα: σε έναν επεξεργαστή έχουμε registers, ALU, Instruction Decoder, Memory Interface etc.

Micro Design ή Low Level Design: είναι το στάδιο που ο σχεδιαστής περιγράφει κάθε τμήμα πως θα υλοποιηθεί. Έχει λεπτομέρειες όπως State machines, counters, Mux, decoders, internal registers. Είναι καλό να σχεδιάζονται και ενδεικτικά waveforms -κυμματομορφές σε αυτό το στάδιο. Είναι το στάδιο που παίρνει τον περισσότερο χρόνο.
Εργαλεία: Επεξεργαστής κειμένου π.χ. Word, Open Office. Εργαλία σχεδίασης διαγραμμάτων π.χ. Powerpoint, visio. Εργαλεία προβολής χρονοσειρών π.χ. waveformer ή testbencher ή ακόμα και στο Word
Παράδειγμα: σε έναν επεξεργαστή έχουμε τα βασικά στοιχεία όπως registers, ALU, Instruction Decoder, Memory Interface etc., αλλά έχουμε και control logic που αποτελείται από πολυπλέκτες και μηχανές καταστάσεων, και υλοποιεί τις λεπτομέρειες για τον συντονισμό των στοιχείων.

RTL Coding: Στο στάδιο αυτό το Micro design μετατρέπεται σε Verilog/VHDL κώδικα με χρήση συνθέσιμων/κατασκευάσιμων στοιχείων.
Εργαλεία: Ο αγαπημένος σας Text Editor π.χ. Notepad++, Vim, Emacs
Παράδειγμα: στο παράδειγμα βλέπουμε σε Verilog έναν full adder. Προσθέτει δύο bits και το κρατούμενο εισόδου και γράφει στην έξοδο το αποτέλεσμα και το κρατούμενο εξόδου.
Functional verification : είναι το στάδιο που πιστοποιούμε ότι τα δομικά στοιχεία του σχεδιασμού μας λειτουργούν. Χρησιμοποιούμε προσομοιωτές. Για να ελέγξουμε ότι ο RTL κώδικάς μας πληροί τις προδι
αγραφές, πρέπει κάθε δομικό RTL στοιχείο να λειτουργεί σωστά. Για τον λόγο αυτό, γράφουμε ένα testbench, που θέτει εισόδους και ελέγχει αποκρίσεις στο RTL στοιχείο που σχεδιάσαμε. 60-70% του χρόνου σχεδιασμού το σπαταλάμε εδώ.
Εργαλεία: Modelsim, VCS, Vivado, Nsim, Verilog-XL, Veriwell, Finsim, Icarus **Παράδειγμα**: στο παράδειγμα βλέπουμε ένα **testbench** για έναν counter.

Τα στάδια σχεδιασμού/κατασακευής μικρεπεξεργαστών

Logic Synthesis: Σύνθεση είναι μια αυτοματοποιημένη διαδικασία την οποία εκτελεί το λογισμικό σύνθεσης το οποίο δέχεται σαν είσοδο:

- RTL κώδικα του υλικού
- Περιγραφή της τεχνολογίας κατασκευής (παρέχεται από τα εργοστάσια κατασκευής)
- Περιορισμούς/προδιαγραφές π.χ. ελάχιστη ταχύτητα ρολογιού, μέγιστη κατανάλωση ισχύος, εμβαδό τσιπ κτλ. DRC (Design Rules Checking)

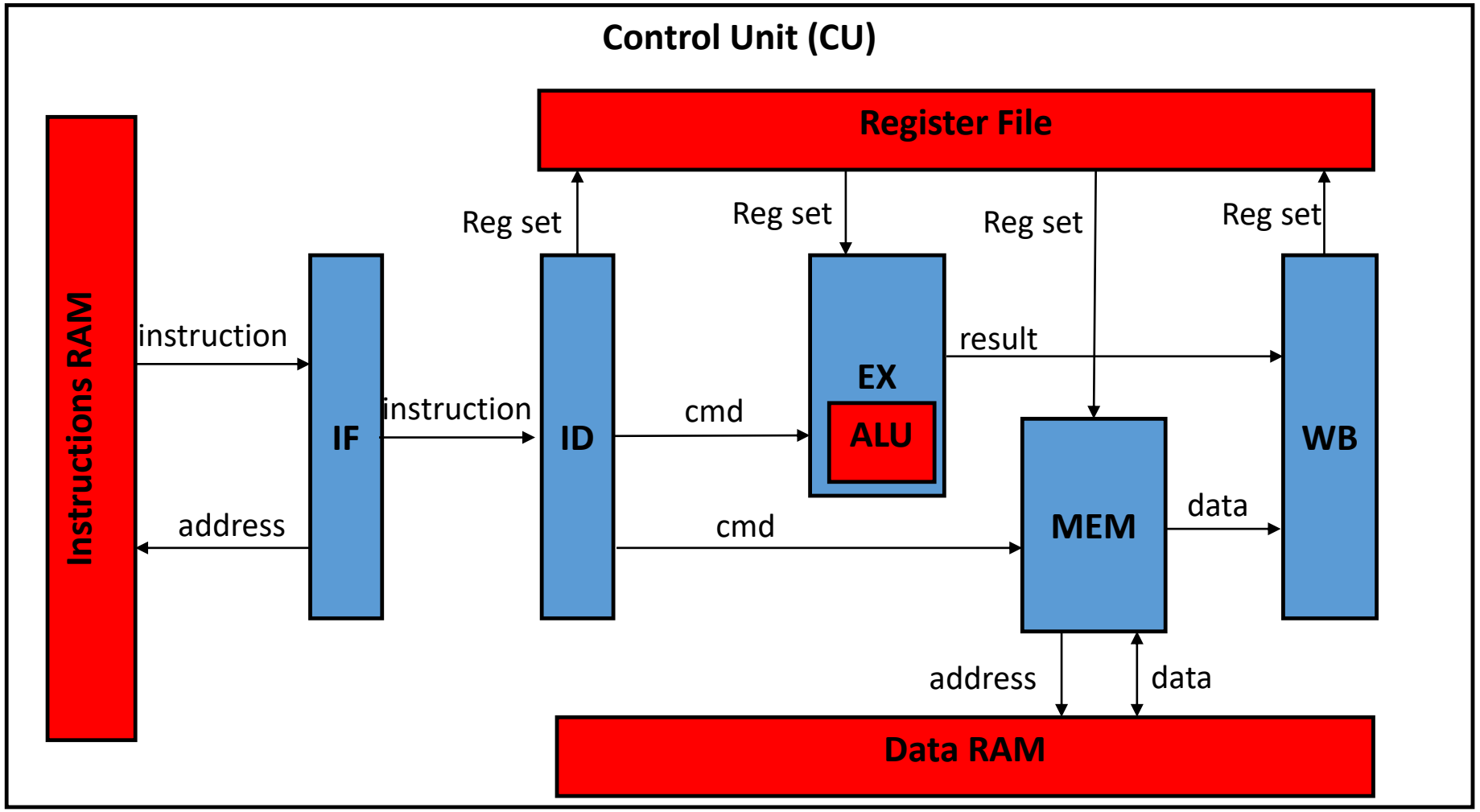
Το λογισμικό σύνθεσης προσπαθεί αυτόματα να παράγει ένα σχέδιο (το **gate-level netlist**) που έχει την ίδια λειτουργικότητα με το παρεχόμενο, αλλά χρησιμοποιώντας μόνο δομικά στοιχεία από την τεχνολογία κατασκευής. Παράλληλα προσπαθεί να ικανοποιήσει τους περιορισμούς.
Εργαλεία: Design Compiler, Leonardo Spectrum, **Quartus, Vivado**. FPGA vendors όπως η Intel και η Xilinx δίνουν τέτοια εργαλεία δωρεάν.
Παράδειγμα: στο παράδειγμα βλέπουμε ένα RTL design και ένα Tech lib να συνθέτουν ένα gate-level netlist ενός flip-flop.

Place and Route : Το gate-level netlist από την σύνθεση επεξεργάζεται από το place and route λογισμικό (P&R ή **layout tools)** σε Verilog gate-level netlist μορφή. Τα gates και flip-flops τοποθετούνται στο τσιπ, το clock-tree συντίθεται και όλα τα στοιχεία συνδέονται με interconnections. Σε ASICs, η έξοδος από το P&R tool είναι το layout (αρχείο GDS) το οποίο χρησιμοποιείται από το foundry για κατασκευή. Σε FPGAs είναι ένα μικροπρόγραμμα (λέγεται bistream), το οποίο χρησιμοποιείται για να αλλάξει το σχεδιασμό του FPGA.
Εργαλεία: For FPGA specific P&R Vivado, Quartus. Τα ASIC απαιτούν ακριβά P&R εργαλεία όπως το IC compiler και το Encounter.
Παράδειγμα: Placement μικροεπεξεργαστή και flip-flop.

Fabrication: κατασκευαστική διαδικασία μέσω λιθογραφίας

Post Silicon Validation: Μόλις κατασκευαστεί το chip (silicon) στο εργαστήριο ελέγχεται σε πραγματικές συνθήκες πριν προωθηθεί στην αγορά. Από τη διαδικασία αυτή μπορεί αν εντοπιστούν κάποια bugs στο design, οπότε θα πρέπει να διορθωθεί στο σχεδιασμό.

Διοχέτευση 5 σταδίων μικροεπεξεργαστή



Components του επεξεργαστή

Πράξεις που εκτελούνται από κάθε στάδιο. Κυρίως από τη λογική της Control Unit

Στάδια εκτέλεσης διοχέτευσης

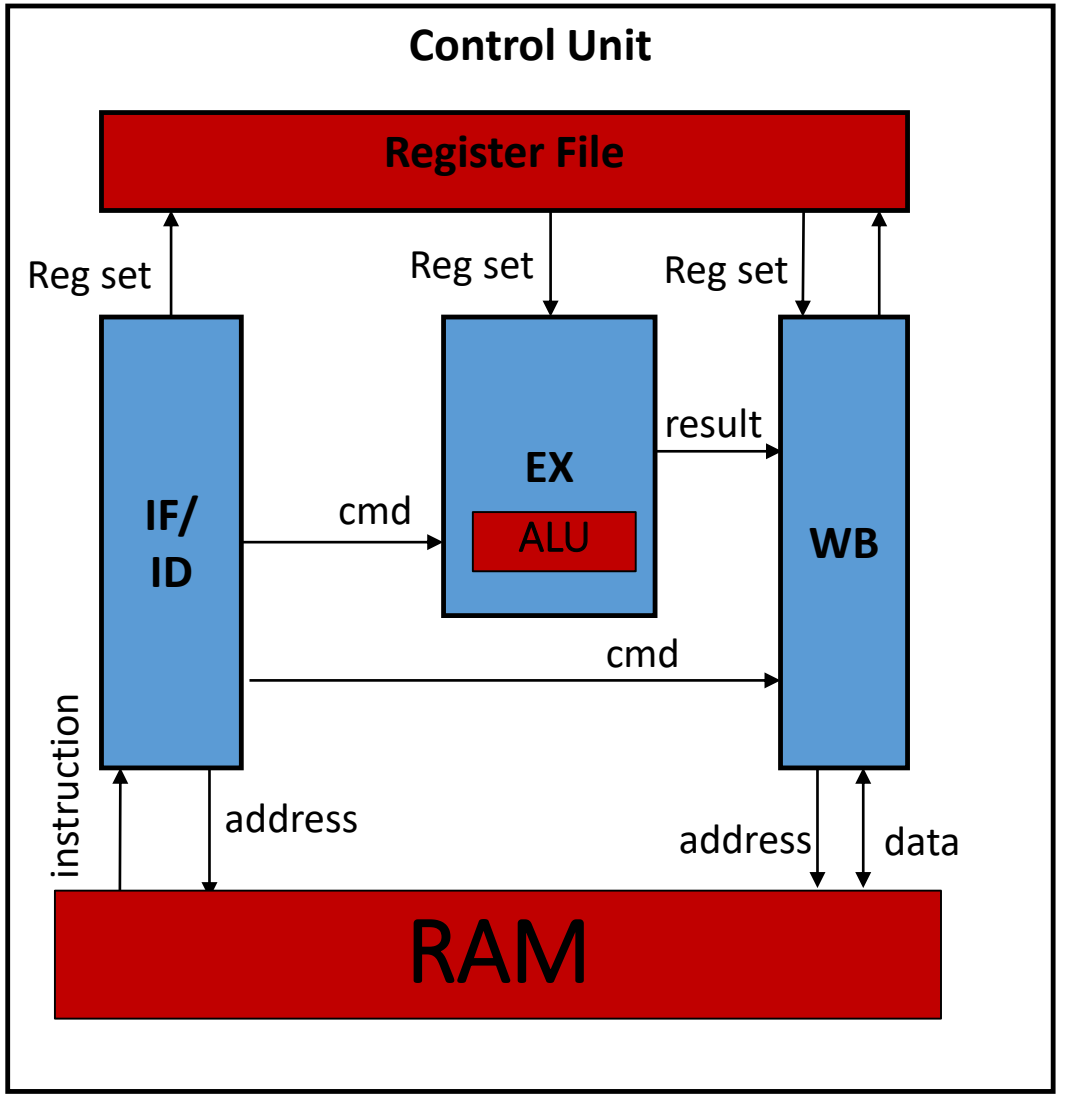
Instruction Fetch (IF): Γίνεται ανάγνωση της επόμενης εντολής (instruction) από την μνήμη εντολών
Αυτό επιτυγχάνεται θέτοντας το **address** στην τιμή του program counter (PC). Θυμηθείτε ο PC κρατάει σε ποια θέση στη μνήμη βρίσκεται η εντολή που εκτελείται
Instruction Decode (ID): ένα κύκλωμα (της μορφής case(instruction)…endcase) ελέγχει για ποια εντολή πρόκειται και ενημερώνει τα επόμενα components για το ποια δεδομένα θα χρειαστούν από την εντολή. Π.χ. αν είναι εντολή που εμπλέκει την μνήμη, τους καταχωρητές ή κάποιο reference/pointer.
Execute (EX): Εκτελεί εντολές που εμπλέκουν επεξεργασία, όπως αριθμητικές/λογικές πράξεις. Μπορούμε να πούμε ότι οδηγεί την Arithmetic Logic Unit (ALU), η οποία δεν φαίνεται στο σχήμα.
Memory Access/Transactions (MEM): ένα κύκλωμα που αναλαμβάνει να διασυνδέση Register file προς την μνήμη. Από το decode θα ενημερωθεί αν πρόκειται κάποια μεταφορά δεδομένων από καταχωρητές προς την μνήμη. Προσέξτε ότι δεν γράφει αυτό το task καταχωρητές, αλλά στέλνει δεδομένα προς το WB το οποίο αναλαμβάνει να γράψει καταχωρητές.
Register Writeback (WB): αναλαμβάνει να γράψει τους καταχωρητές με αποτελέσματα από το στάδιο EX ή με δεδομένα από την μνήμη.
Προσέξτε στο στάδιο αυτό ότι κάποιο αποτέλεσμα της ALU δεν γράφεται στην μνήμη ποτέ αλλά γράφεται στους καταχωρητές και μετά με την επόμενη εντολή προς την μνήμη.

Αρχιτεκτονική του MicroCPU με 3 στάδια διοχέτευσης

Ο επεξεργαστής που θα σχεδιάσουμε θα έχει την αρχιτεκτονική διοχέτευσης που φαίνεται εδώ.

Components του επεξεργαστή

Πράξεις που εκτελούνται. Κυρίως από λογική της Control Unit



Μοντελοποίηση της Μνήμης και Σχεδίαση του Ελεγκτή Μνήμης

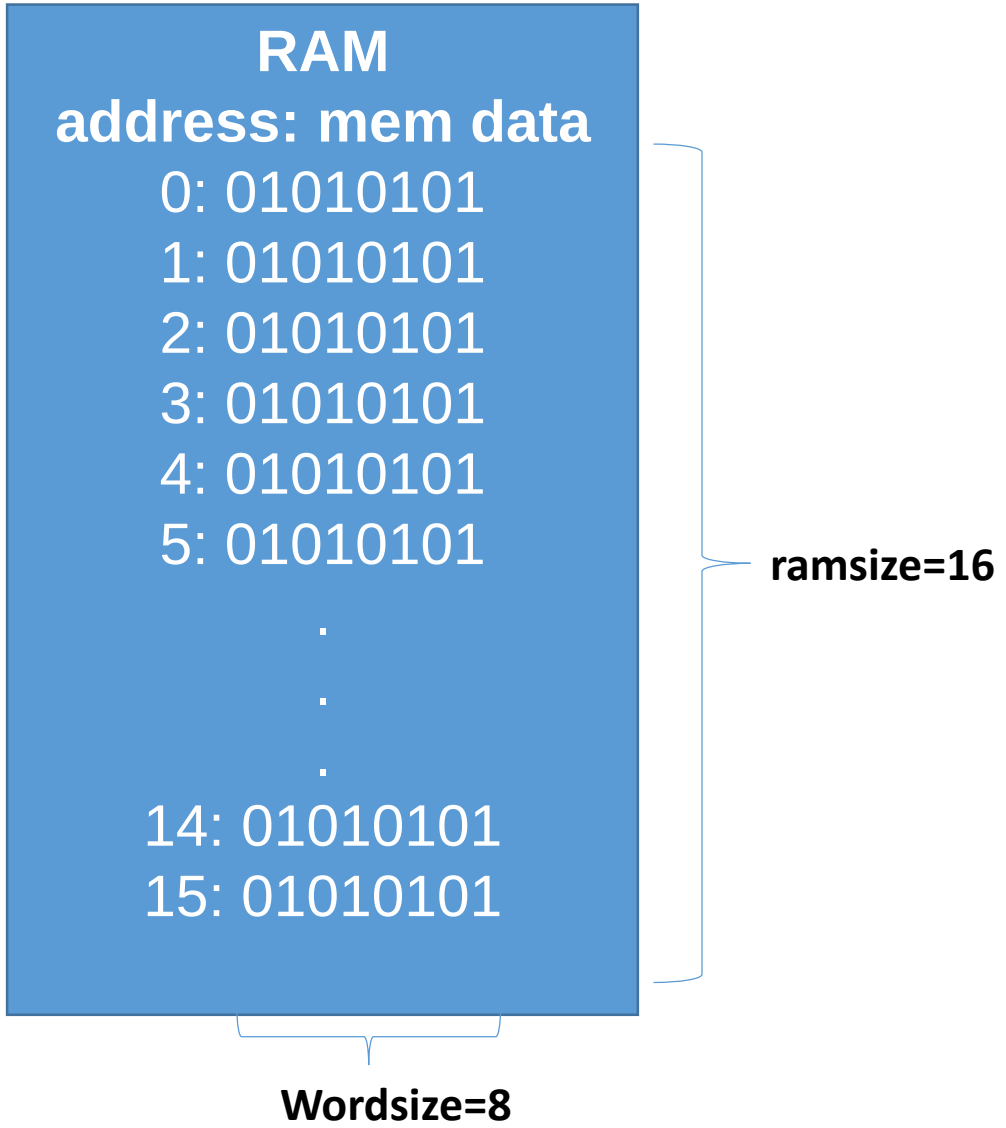
Μοντελοποίηση Μνήμης στην Verilog

Η μνήμη στην Verilog μοντελοποιείται ως καταχωρητές:
reg [wordsize-1:0] my_memory [0:ramsize-1]

Εδώ **wordsize**, που είναι και το **πλάτος της μνήμης**, σημαίνει ότι **μια λέξη** αυτής της μνήμης έχει μέγεθος wordsize bits.

Το **ramsize** είναι το μέγεθος της μνήμης σε πλήθος λέξεων.

Παράδειγμα:
reg [7:0] my_memory [0:15]



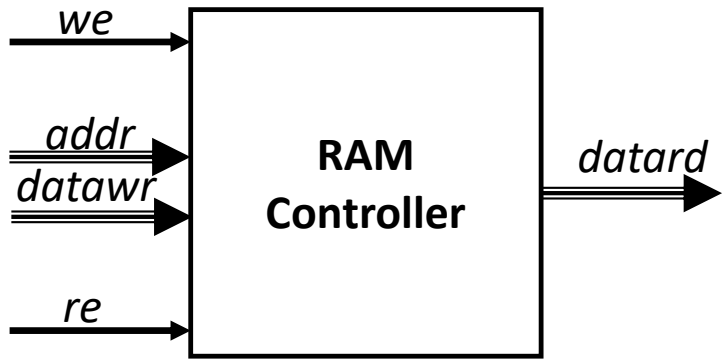
Αποθήκευση δεδομένων
my_memory [address]=data_in;

Ανάγνωση δεδομένων
data_in=my_memory[address];

Παρατήρηση:
Συνήθως γίνεται ανάγνωση και αποθήκευση μιας διεύθυνσης τη φορά, γιατί αλλιώς αυξάνει το κόστος της μνήμης.

Ελεγκτής Μνήμης

Ο ελεγκτής ελέγχει την εγγραφή και ανάγνωση της μνήμης.



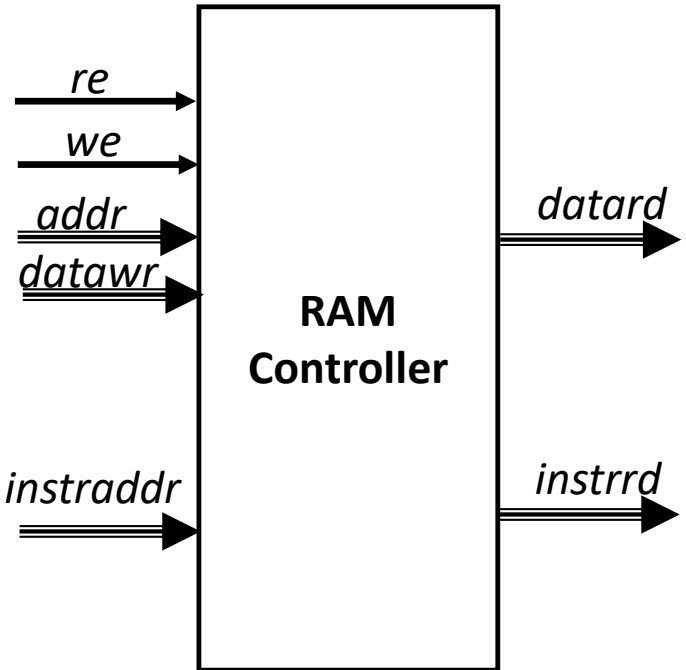
Παράδειγμα ελεγκτή μνήμης μιας εισόδου εγγραφής και μιας ανάγνωσης:

```
module ramcontroller(we, addr, datawr, re, datard);
parameter WORD_SIZE=8;
parameter ADDR_WIDTH=8; //πλήθος bits για διευθύνσεις
//αυτό είναι το μέγεθος της μνήμης είναι δύναμη
//του 2 εις το πλήθος των bits διεύθυνσης
parameter RAM_SIZE=1<<ADDR_WIDTH;
//σήματα write και read enable
input we, re;
//διεύθυνση εγγραφής και ανάγνωσης
input [ADDR_WIDTH-1:0] addr;
input [WORD_SIZE=8] datawr;
//σήμα εξόδου για να σταλούν τα δεδομένου που αναγν.
output [WORD_SIZE-1:0] datard;
//η δήλωση της μνήμης
reg [WORD_SIZE-1:0] mem[RAM_SIZE-1:0];

//καταχωρητής για τα δεδομένου που θα διαβαστούν
reg [WORD_SIZE-1:0] datard;

always @ (addr or we or re or datawr)
begin
 if(we)begin
  #2 mem[addr]=datawr;
 end
 if(re) begin
  #2 datard=mem[addr];
 end
end
endmodule
```

Ελεγκτής μνήμης πολλαπλών εισόδων/εξόδων του MicroCPU

Θα χρησιμοποιήσουμε μια μνήμη με έναν ελεγκτή μιας εισόδου εγγραφής και δύο εισόδων ανάγνωσης για τον επεξεργαστή μας. Την δυνατότητα εισόδου θα την χρησιμοποιήσουμε για να γράφουμε δεδομένα στην μνήμη και τις δύο θύρες ανάγνωσης θα τις χρησιμοποιούμε, την μία για να διαβάζουμε εντολές προς εκτέλεση και την άλλη δεδομένα… ΝΑΙ και οι εντολές αποθηκεύονται στην μνήμη.

Παράδειγμα ελεγκτή μνήμης μιας εισόδου εγγραφής και δύο εισόδων ανάγνωσης. Η μία είναι δεδομένων και η άλλη εντολών/instructions:



Μπλοκ για δεδομένα

```
always @ (addr or we or re or datawr)
begin
  if(we)begin
    mem[addr]=datawr;
  end
  if(re) begin
    datard=mem[addr];
  end
end
```

Μπλοκ για instructions

```
always @ (instraddr)
begin
   instrrd=mem[instraddr];
end
```

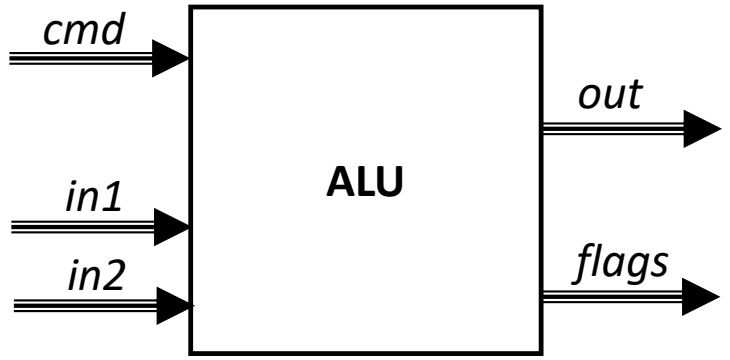
Σχεδίαση της Αριθμητικής Λογικής Μονάδας – Arithmetic Logic Unit (ALU)

Βασικά για την ALU

Η αριθμητική/λογική μονάδα είναι υπεύθυνη για την επεξεργασία των δεδομένων στον επεξεργαστή.

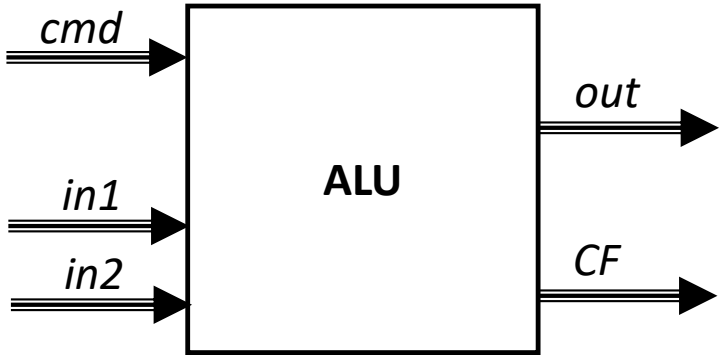
Εκτελεί αριθμητικές (πρόσθεση, αφαίρεση, πολλαπλασιασμό κτλ) και λογικές πράξεις (bitwise not, or, xor κτλ.) με τους καταχωρητές.

Επιστρέφει τα δεδομένα της σε κάποιον καταχωρητή και τα flags από τις αριθμητικές πράξεις (π.χ. κρατούμενα υπολογισμών carry flag, διαίρεσης με το μηδέν (zero flag) κτλ.)



Η ALU του MicroCPU

```
module MCPU_Alu(cmd,in1,in2,out,CF);
parameter CMD_SIZE=4;
parameter WORD_SIZE=8;

parameter  [1:0]  CMD_AND  = 0; //2'b00
parameter  [1:0]  CMD_OR   = 1; //2'b01
parameter  [1:0]  CMD_XOR  = 2; //2'b10
parameter  [1:0]  CMD_ADD  = 3; //2'b11

input [CMD_SIZE-1:0] cmd;
input [WORD_SIZE-1:0] in1;
input [WORD_SIZE-1:0] in2;
output[WORD_SIZE-1:0] out;
//carry flag
output CF;
wire [CMD_SIZE-1:0] cmd;
wire [WORD_SIZE-1:0] in1;
wire [WORD_SIZE-1:0] in2;
reg  [WORD_SIZE-1:0] out;
//carry flag
reg  CF;
always @ (cmd, in1, in2)
  #2
case(cmd)
  CMD_AND : begin
             out = in1&in2;
     end
  CMD_OR : begin
            out = in1|in2;
           end
  CMD_XOR : begin
            out = in1^in2;
           end
  default : begin
            {CF,out} = in1+in2;
           end
 endcase
endmodule
```



Άσκηση 6.1b: Να γράψετε ένα testbench με το οποίο να δίνετε τυχαίες εντολές και operands στην ALU του MicroCPU και προσομοιώνοντάς το στο Modelsim να βεβαιώσετε την ορθότητα του σχεδιασμού. Προσοχή μπορεί να υπάρχει λάθος στον κώδικα.

Το σύνολο των εντολών – Instructions Set

Εντολές μηχανής (instruction set) του MicroCPU

- **Κλασσικές εντολές επεξεργασίας** που εμπλέκουν την ALU για αριθμητικές και λογικές πράξεις, π.χ.:
  AND operand1 operand2 operand3
  OR operand1 operand2 operand3
  XOR operand1 operand2 operand3
  ADD operand1 operand2 operand3
  Το αποτέλεσμα γράφεται στον operand1. Οι πράξεις εμπλέκουν δύο operands τους operand2 και operand3.
  Operands για τον MicroCPU είναι μόνο οι καταχωρητές του.
- **Εντολές μετακίνησης δεδομένων από καταχωρητή σε καταχωρητή**:
  MOV Rd R
  τα δεδομένα αντιγράφονται από τον καταχωρητή R στον καταχωρητή Rd (το Rd προκύπτει από το register destination).
- **Εντολές φόρτωσης δεδομένων από μνήμη σε καταχωρητή**:
  Load_FROM_MEM Rd address
  αντιγράφει τα δεδομένα από την διεύθυνση μνήμης address στον καταχωρητή Rd.
- **Εντολές αποθήκευσης δεδομένων από καταχωρητή στην μνήμη**:
  STORE_TO_MEM address R
  αποθηκεύει στη διεύθυνση μνήμης address τα δεδομένα του καταχωρητή R.
- **Εντολές Αρχικοποίησης τιμών**:
  OP_SHORT_TO_REG Rd value
  αντιγράφει την τιμή value των 8 bits στον καταχωρητή Rd.
- **Εντολές διακλάδωσης:**
  BNZ Rc address
  μετακινεί τον program counter στην τιμή address αν ο καταχωρητής Rc δεν είναι 0. (Branch if Not Zero)

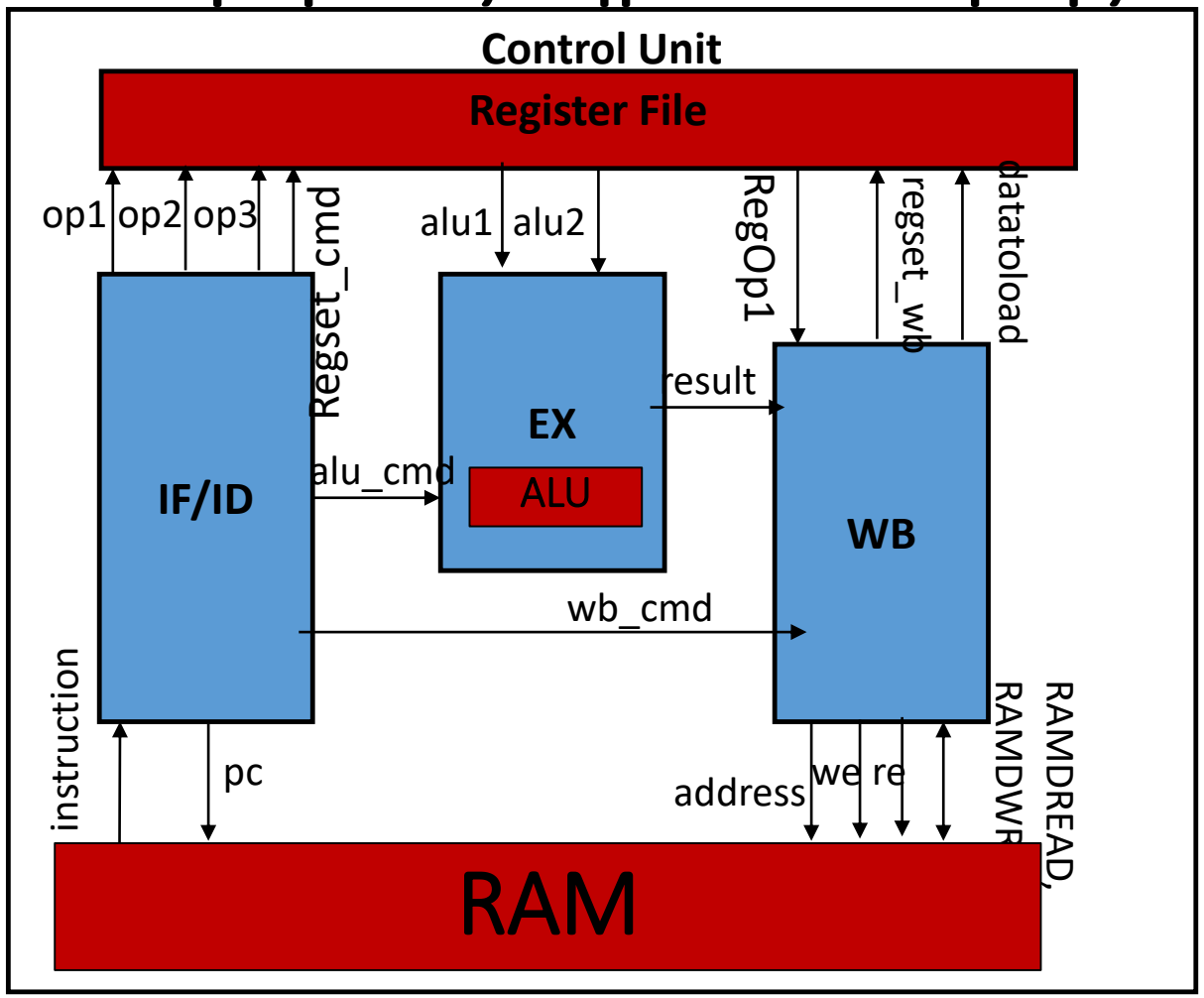
Σχεδίαση του αρχείου καταχωρητών – Register File

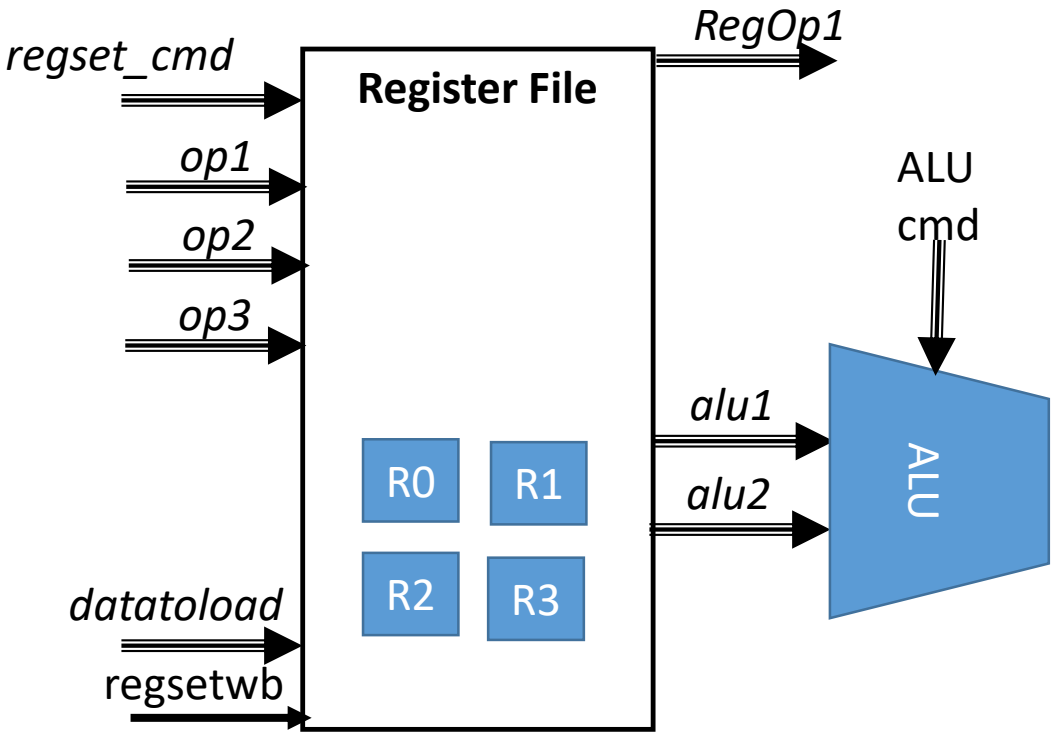
Το αρχείο καταχωρητών – register file

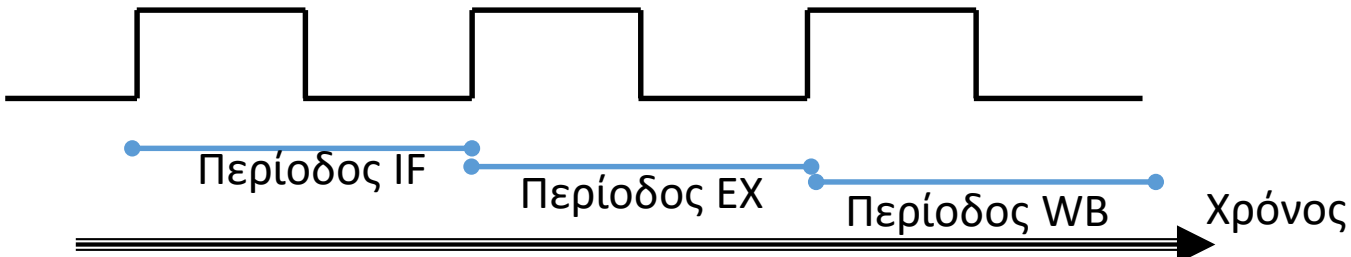
Το αρχείο καταχωρητών του MicroCPU έχει μόνο 4 καταχωρητές R0,R1,R2 και R3 των 16 bits.

Τουλάχιστον ένας καταχωρητής συμμετέχει σε κάθε εντολή
AND Rd Ra Rb
OR Rd Ra Rb
XOR Rd Ra Rb
ADD Rd Ra Rb
MOV Rd R
Load_FROM_MEM Rd address
STORE_TO_MEM address R
OP_SHORT_TO_REG Rd value
BNZ Rc address

Λεπτομέρειες σηματοδότησης του register file





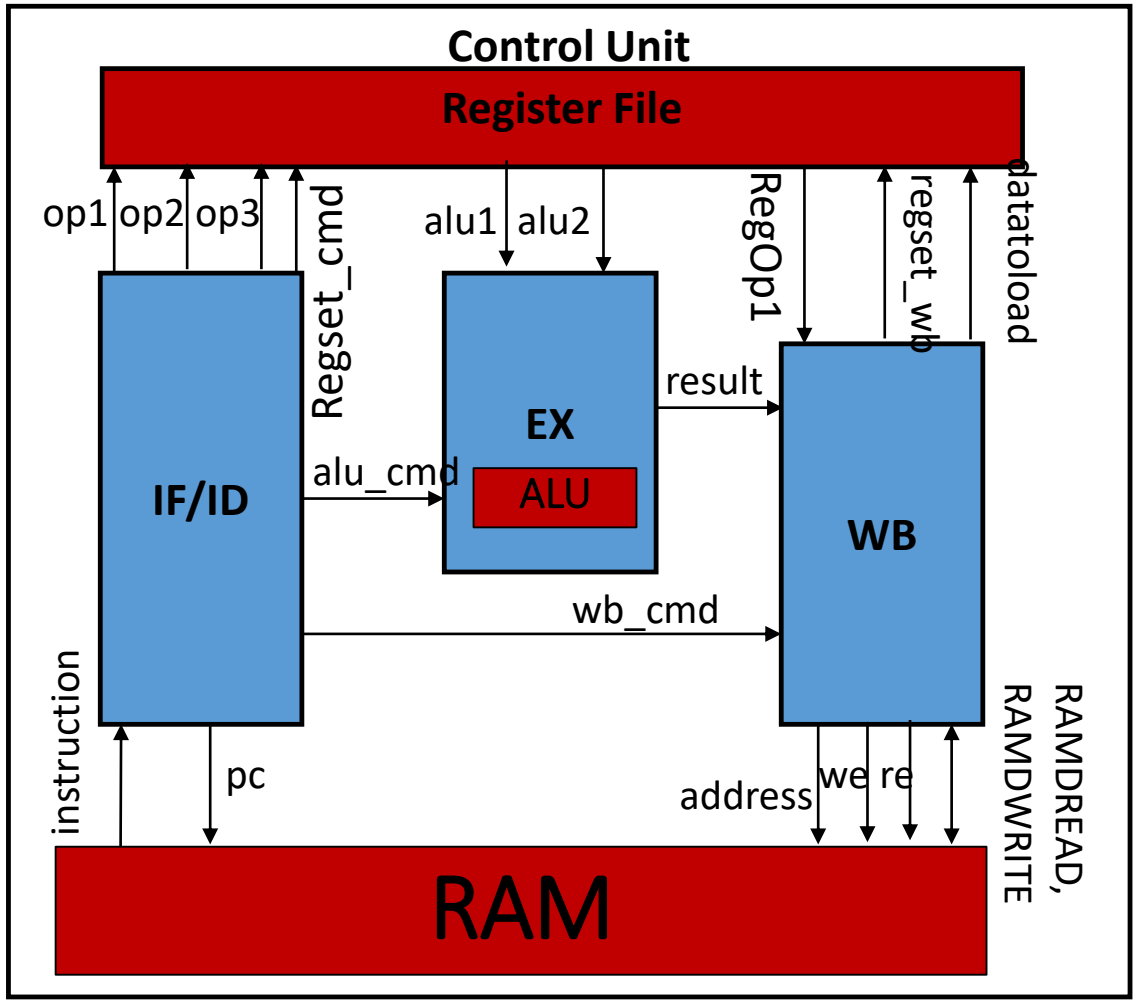


Η μονάδα ελέγχου – Control Unit

Οι λειτουργίες της μονάδας ελέγχου

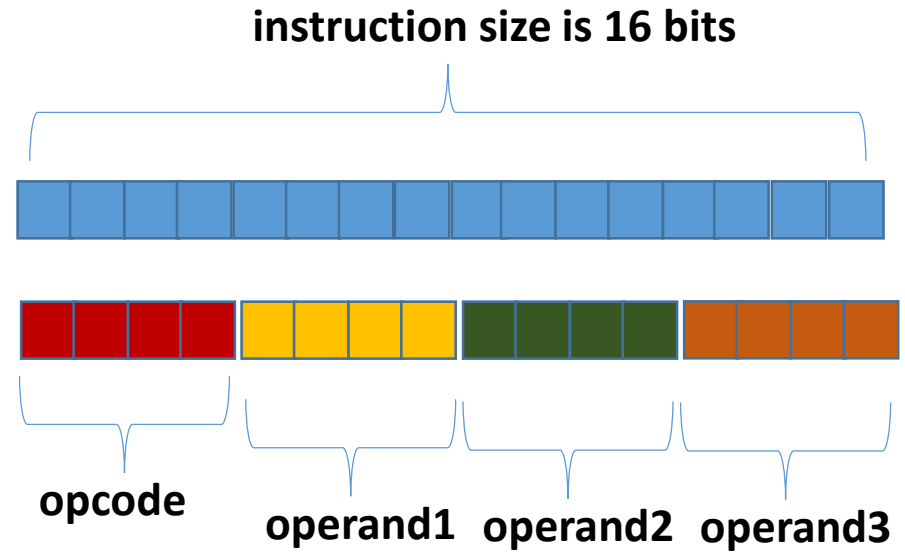
Η μονάδα ελέγχου είναι υπεύθυνη για την αλλαγή των καταστάσεων της διοχέτευσης και την κατάλληλη σηματοδότηση των μονάδων (ALU, registerfile, RAM) την κατάλληλη χρονική στιγμή.

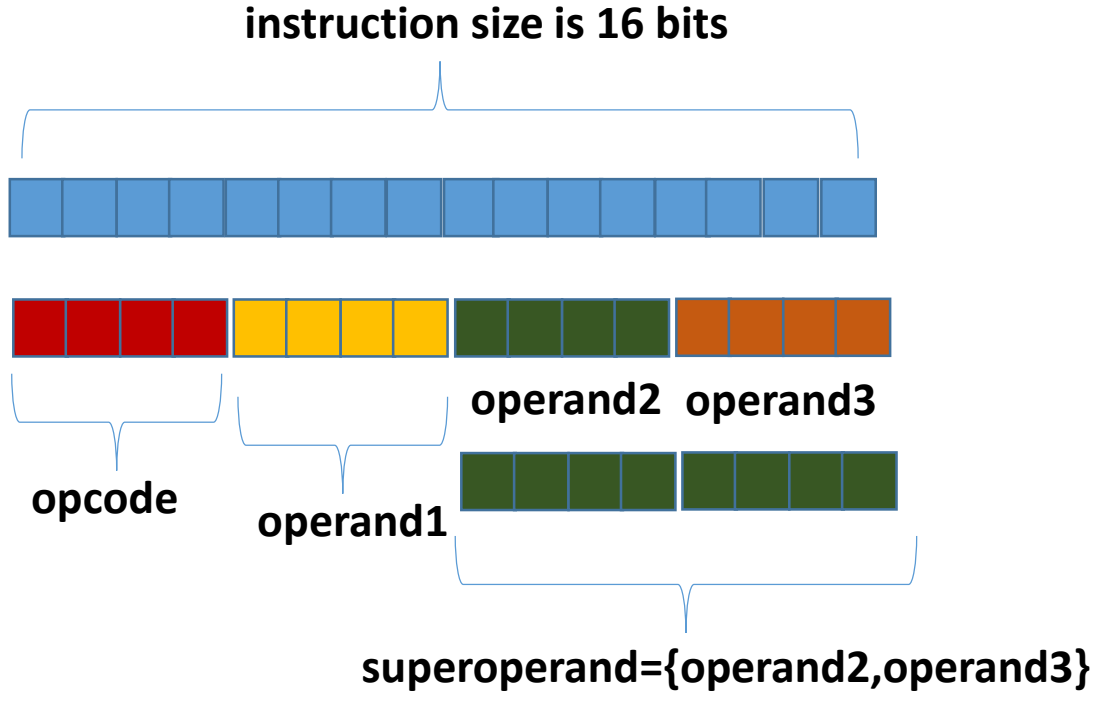
Εμπεριέχει ως αντικείμενα όλα τα δομικά modules του MicroCPU. Είναι επίσης το top module του MicroCPU.



Η μονάδα ελέγχου του MicroCPU – κωδικοποίηση/αποκωδικοποίηση εντολών

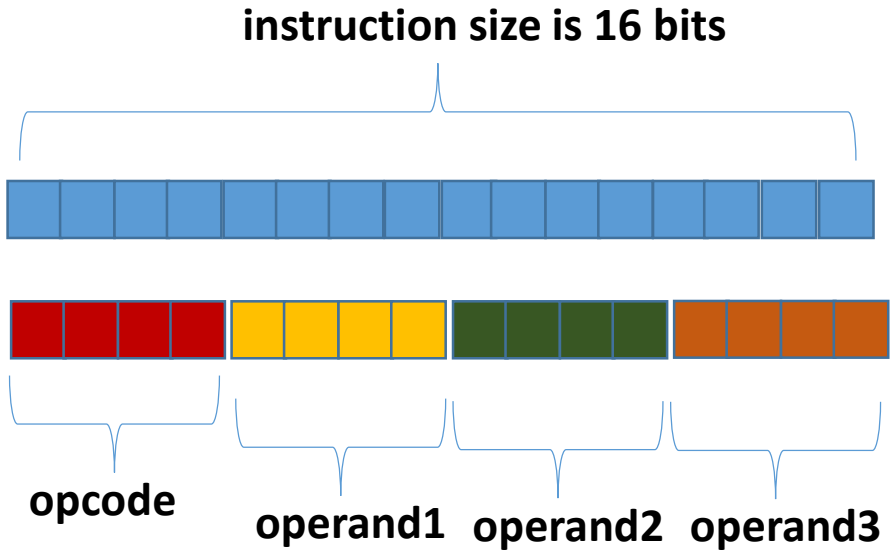
```
module MCPU(clk, reset);
parameter WORD_SIZE=16; //WORD_SIZE is the word size of our
processor. Each word is 16 bits or 2 bytes
parameter ADDR_WIDTH=8; //that means 2^8=256 size of memory of
WORD_SIZE word
parameter OPCODE_SIZE=4;
parameter ALU_CMD_SIZE=2;
parameter OPERAND_SIZE=4;

//instructions will have the following structure:
//OPCODE OPERAND1 OPERAND2 OPERAND3
parameter INSTRUCTION_SIZE = OPCODE_SIZE + OPERAND_SIZE*3;

wire [INSTRUCTION_SIZE-1:0] instruction;

input clk;
input reset;
wire [OPCODE_SIZE-1:0] opcode;
wire [OPERAND_SIZE-1:0] operand1;
wire [OPERAND_SIZE-1:0] operand2;
wire [OPERAND_SIZE-1:0] operand3;
```





Η μονάδας ελέγχου του MicroCPU – κωδικοποίηση εντολών – opcode options – instruction set

```
//opcodes for the supported instruction set
parameter  [OPCODE_SIZE-1:0]  OP_AND  = 0; //4'b0000
parameter  [OPCODE_SIZE-1:0]  OP_OR   = 1; //4'b0001
parameter  [OPCODE_SIZE-1:0]  OP_XOR  = 2; //4'b0010
parameter  [OPCODE_SIZE-1:0]  OP_ADD  = 3; //4'b0011
parameter  [OPCODE_SIZE-1:0]  OP_MOV  = 4; //4'b0100
parameter  [OPCODE_SIZE-1:0]  OP_LOAD_FROM_MEM = 5; //4'b0101
parameter  [OPCODE_SIZE-1:0]  OP_STORE_TO_MEM = 6; //4'b0110
parameter  [OPCODE_SIZE-1:0]  OP_SHORT_TO_REG = 7; //4'b0111
parameter  [OPCODE_SIZE-1:0]  OP_BNZ   = 8; //4'b1000
```

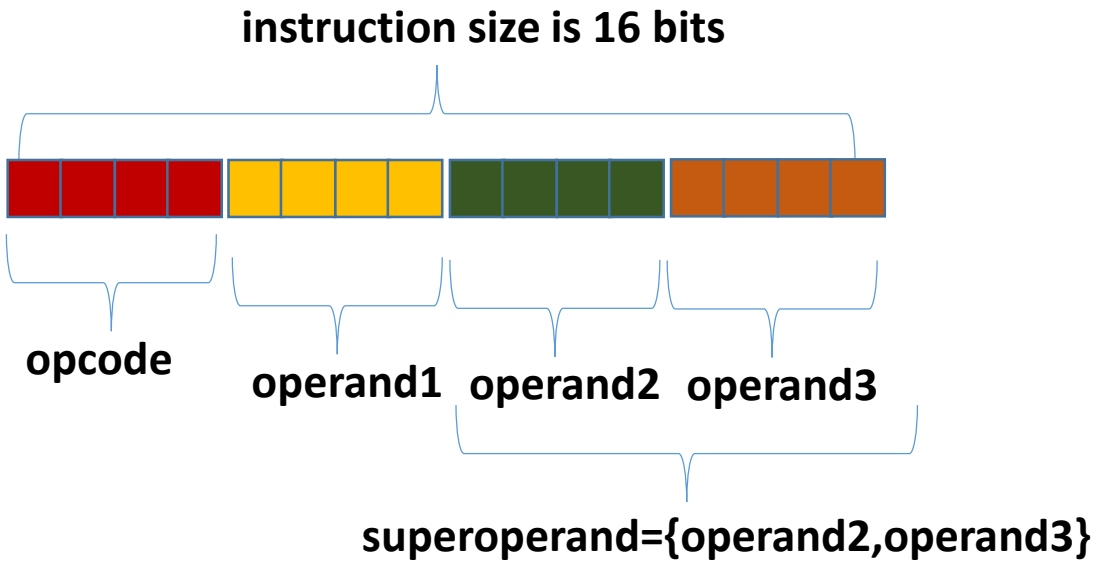


Η μονάδας ελέγχου του MicroCPU – Δομική περιγραφή Συνδυαστικού τμήματος

```
//this is always operand 3 value as a register
assign RAMDWRITE=RegOp1;
assign instruction=IREAD;
assign opcode=instruction[INSTRUCTION_SIZE-1:INSTRUCTION_SIZE-OPCODE_SIZE];
assign aluopcode=opcode[ALU_CMD_SIZE-1:0];
assign operand1=instruction[OPCODE_SIZE*3-1:2*OPCODE_SIZE];
assign operand2=instruction[OPCODE_SIZE*2-1:OPCODE_SIZE];
assign operand3=instruction[OPCODE_SIZE-1:0];
assign IADDR=pc;
wire [WORD_SIZE-1:0] MemOrConstant;
assign MemOrConstant=(opcode==OP_SHORT_TO_REG)?
                                {8'b00000000, operand2, operand3}:
                                RAMDREAD;
assign regdatatoload=
        (regset_cmd==regfileinst.NORMAL_EX)?alu_out:MemOrConstant;
```

Η βασική λειτουργία αυτού του τμήματος είναι η αποκωδικοποίηση των εντολών που διαβάζονται από την μνήμη. Αρχικά η διεύθυνση ανάγνωση εντολής από την μνήμη συνδέεται με τον program counter pc. Επομένως, το τρέχων instruction θα είναι πάντα στα IREAD, το οποίο συνδέεται με το instruction.

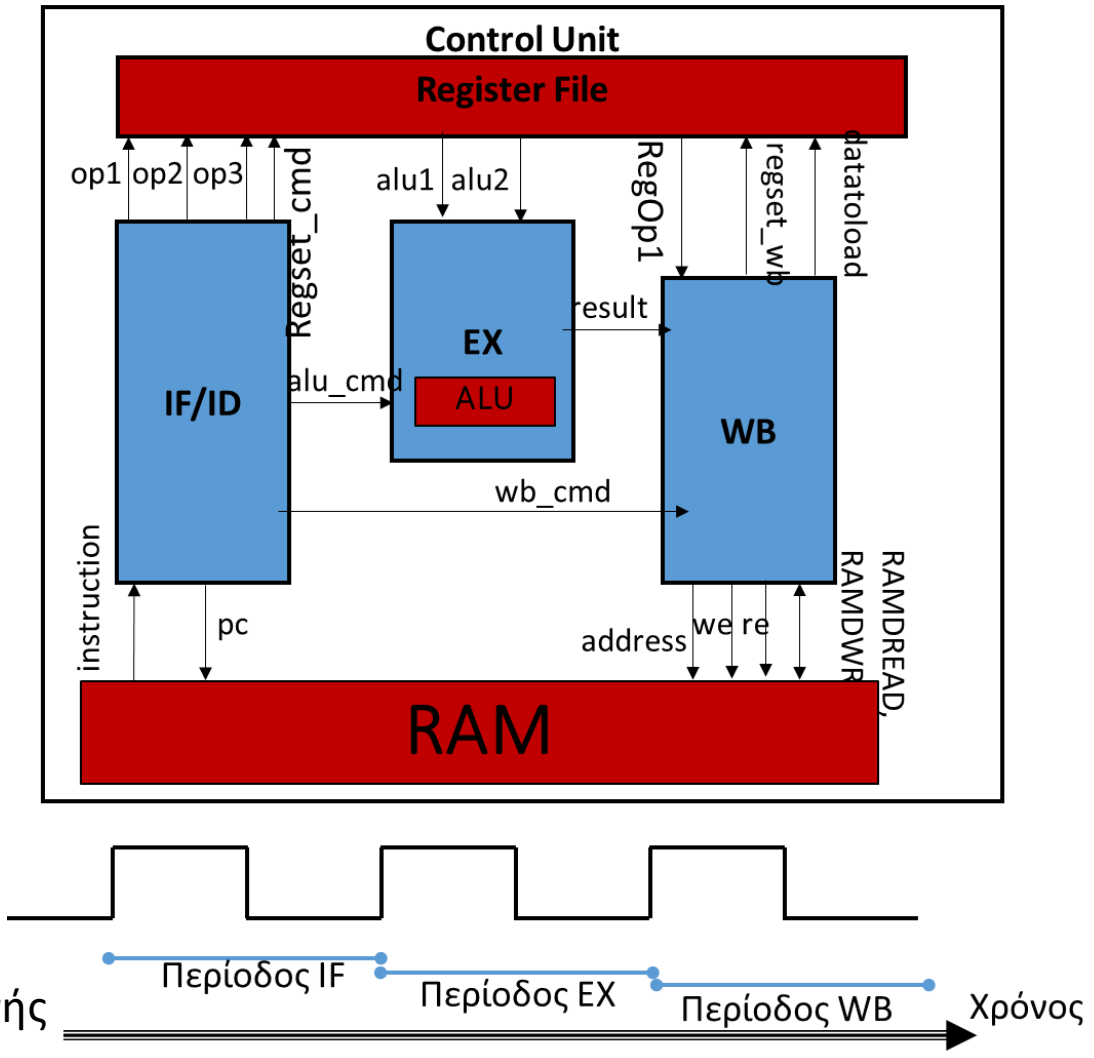
Το instruction είναι η εντολή που διαβάζεται από την μνήμη, η οποία σπάει σε 4 στοιχεία των 4ρων bits, τα οποία είναι: το opcode και τα 3 operands,(operand1, operand2, operand3).



Εδώ σχεδιάζεται το κύκλωμα που γράφει τα δεδομένα στο σήμα regdatatoload το οποίο έχει τα δεδομένα που γράφονται στο registerfile. Στο registerfile γράφονται είτε όταν εκτελούνται εντολές επεξεργασίας όπως είναι οι ADD, OR, XOR κτλ (το καταλαβαίνει από το NORMAL_Ex ότι πρόκειται για τέτοιες εντολές) είτε όταν γράφονται σταθερές των 8 bits ως **superoperand** από το instruction προς τους καταχωρητές (το καταλαβαίνει από το opcode== OP_SHORT_TO_REG).

Η μονάδας ελέγχου του MicroCPU – καταστάσεις για το ακολουθιακό τμήμα

```
//parameter CPU_STATES_BITS=2;
//Instruction Fetch State
parameter  [1:0]  IF_STATE  = 2'b00;
//Execute Fetch State
parameter  [1:0]  EX_STATE  = 2'b01;
//WriteBack State
parameter  [1:0]  WB_STATE  = 2'b10;
//HALTED State
parameter  [1:0]  HLT_STATE  = 2'b11;

reg [1:0] state;
reg [1:0] next_state;
```



Αυτές είναι οι καταστάσεις λειτουργίας της μηχανής καταστάσεων της μονάδας ελέγχου. Ταυτίζονται στο σχεδιασμό μας με τα στάδια της διοχέτευσης.

Παραδείγματα προγραμμάτων assembly και επεκτασιμότητα στον MicroCPU

Outline

- **Επεξήγηση του Προγράμματος Παραγωγής της ακολουθίας Fibonacci**

- **Επεκτασιμότητα του MicroCPU**
  - **Επέκταση του πλήθους καταχωρητών**
  - **Επέκταση του συνόλου των εντολών (παράδειγμα sub)**
  - **Παράδειγμα testbench για εκτέλεση αφαίρεσης με τους 16 νέους καταχωρητές**

- **Παραδείγματα βρόγχων**
  - **Βρόγχος if (R1!=α)**
  - **Ειδική περίπτωση if (R1!=1)**
  - **Έλεγχος μονά-ζυγά**

Εκτέλεση Προγράμματος στον Επεξεργαστή MicroCPU

Για να εκτελέσουμε ένα πρόγραμμα στον MicroCPU πρέπει να φτιάξουμε ένα testbench στο οποίο θα δημιουργήσουμε ένα instance/αντικείμενο του module του MicroCPU. Έπειτα πρέπει να αποθηκεύσουμε το πρόγραμμα που θέλουμε να εκτελέσουμε στην μνήμη του MicroCPU και μετά με κατάλληλη σηματοδότηση των σημάτων reset και clk μπορεί να γίνει η εκτέλεση του προγράμματος.

Φυσικά το πρόγραμμα πρέπει να είναι σε γλώσσα μηχανής, επομένως πρέπει αρχικά να μελετήσουμε την αρχιτεκτονική του συνόλου εντολών του MicroCPU που βρίσκετε στο επόμενο κεφάλαιο.

Ως παράδειγμα ακολουθεί έχουμε ένα πρόγραμμα που υπολογίζει τους αριθμούς Fibonacci

Επεξήγηση του Προγράμματος Παραγωγής της ακολουθίας Fibonacci

Εκτέλεση Προγράμματος στον Επεξεργαστή MicroCPU

```
module MCPUtb();
reg reset, clk;
MCPU cpuinst (clk, reset);

initial begin
  reset=1;
  #10  reset=0;
end
always begin
  #5 clk=0;
  #5 clk=1;
End

/********ASSEMBLER*****/
integer file, i;
reg[cpuinst.WORD_SIZE-1:0] memi;
parameter  [cpuinst.OPERAND_SIZE-1:0]  R0  = 0; //4'b0000
parameter  [cpuinst.OPERAND_SIZE-1:0]  R1  = 1; //4'b0001
parameter  [cpuinst.OPERAND_SIZE-1:0]  R2  = 2; //4'b0010
parameter  [cpuinst.OPERAND_SIZE-1:0]  R3  = 3; //4'b0011
```

Αρχικά φτιάχνουμε ένα instance του module MCPU, το οποίο είναι το top-level module του επεξεργαστή MicroCPU. Το τροφοδοτούμε με τα σήματα clk και το reset.

Το μπλοκ αυτό εκτελείτε στην αρχή της προσομοίωσης μόνο μια φορά και θέτει το σήμα reset του MicroCPU στο λογικό-1 για 10 ps. Μετά από 10ps το θέτει στην τιμή λογικό-0.

Το μπλοκ αυτό εκτελείτε για πάντα και είναι η γεννήτρια του ρολογιού clk του MicroCPU

Εκτέλεση Προγράμματος στον Επεξεργαστή MicroCPU

```
/********ASSEMBLER*****/
integer file, i;
reg[cpuinst.WORD_SIZE-1:0] memi;
parameter  [cpuinst.OPERAND_SIZE-1:0]  R0  = 0; //4'b0000
parameter  [cpuinst.OPERAND_SIZE-1:0]  R1  = 1; //4'b0001
parameter  [cpuinst.OPERAND_SIZE-1:0]  R2  = 2; //4'b0010
parameter  [cpuinst.OPERAND_SIZE-1:0]  R3  = 3; //4'b0011
```

Τώρα ξεκινάει η δήλωση μεταβλητών και καταχωρητών που θα μας είναι χρήσιμες για την προσομοίωση.
file: θα αποθηκεύσουμε το πρόγραμμα σε γλώσσα μηχανής σε ένα αρχείο
Δηλώνουμε τους integers R0,R1,R2,R3 και R4 στους κωδικούς των αντίστοιχων καταχωρητών του MicroCPU για να μπορούμε να χρησιμοποιήσουμε τα σύμβολα Rx κατά την δημιουργία του προγράμματός μας. Αυτό μας δίνει την δυνατότητα να έχουμε μια άμεση μεταγλώττιση των συμβόλων στους κωδικούς. Είναι δηλαδή μια απλοποιημένη μορφή assembler.

Εκτέλεση Προγράμματος στον Επεξεργαστή MicroCPU

```
initial
begin
   for(i=0;i<256;i=i+1)
   begin
     cpuinst.raminst.mem[i]=0;
   end
   cpuinst.regfileinst.R[0]=0;
   cpuinst.regfileinst.R[1]=0;
   cpuinst.regfileinst.R[2]=0;
   cpuinst.regfileinst.R[3]=0;
```

Στο μπλοκ αυτό αρχικά μηδενίζουμε όλες τις λέξεις της μνήμης και όλους τους καταχωρητές του MicroCPU. Θέλουμε έτσι να αποφύγουμε τα undefined Xes στην προσομοίωσή μας.

Εκτέλεση Προγράμματος στον Επεξεργαστή MicroCPU

```
i=0;  cpuinst.raminst.mem[0]={cpuinst.OP_SHORT_TO_REG, R0, 8'b00000000};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SHORT_TO_REG, R1, 8'b00000001};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SHORT_TO_REG, R2, 8'b00000010};

i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_MOV, R0, R1, 4'b0000};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_MOV, R1, R2, 4'b0000};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_ADD, R2, R0, R1};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_STORE_TO_MEM, R2, 8'b00010100};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_LOAD_FROM_MEM, R3, 8'b00010100};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_ADD, R0, R0, R0};

i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_BNZ, R2, 8'b00000011};
```

Ο κώδικας που ακολουθεί, γράφει στην μνήμη του MCPU το **πρόγραμμα/benchmark** που θέλουμε να εκτελέσει.
Κάθε γραμμή είναι μια εντολή. Μεταφράζω:
Θέση μνήμης: Εντολή

```
0: R0=0;
1: R1=1;
2: R2=2;
do{
  3: R0=R1;
  4: R1=R2;
  5: R2=R0+R1;
  6:mem[20]=R2;
7:R3=mem[20];
  8:R0=R0+R0
}
9:while(R2!=0)
10:…
```

Εκτέλεση Προγράμματος στον Επεξεργαστή MicroCPU

```
  file = $fopen("program.list","w");
  for(i=0;i<cpuinst.raminst.RAM_SIZE;i=i+1)
  begin
   memi=cpuinst.raminst.mem[i];

   $fwrite(file, "%b_%b_%b_%b\n",
     memi[cpuinst.INSTRUCTION_SIZE-1:cpuinst.INSTRUCTION_SIZE-cpuinst.OPCODE_SIZE],
     memi[cpuinst.OPCODE_SIZE*3-1:2*cpuinst.OPCODE_SIZE],
     memi[cpuinst.OPCODE_SIZE*2-1:cpuinst.OPCODE_SIZE],
     memi[cpuinst.OPCODE_SIZE-1:0]);
  end
  $fclose(file);
end
endmodule
```

Στην συνέχεια ανοίγει το αρχείο “program.list” και γράφει στο αρχείο όλη την μνήμη του MicroCPU.

Το παράδειγμα με την ακολουθία Fibonacci

```
i=0;  cpuinst.raminst.mem[0]={cpuinst.OP_SHORT_TO_REG, R0, 8'b00000000};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SHORT_TO_REG, R1, 8'b00000001};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SHORT_TO_REG, R2, 8'b00000010};

i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_MOV, R0, R1, 4'b0000};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_MOV, R1, R2, 4'b0000};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_ADD, R2, R0, R1};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_STORE_TO_MEM, R2, 8'b00010100};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_LOAD_FROM_MEM, R3, 8'b00010100};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_ADD, R0, R0, R0};

i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_BNZ, R2, 8'b00000011};
```

Κάθε γραμμή είναι μια εντολή.
Μεταφράζω:
Θέση μνήμης: Εντολή

```
0: R0=0;
1: R1=1;
2: R2=2;
do{
  3: R0=R1;
  4: R1=R2;
  5: R2=R0+R1;
  6:mem[20]=R2;
  7:R3=mem[20];
  8:R0=R0+R0
}
9:while(R2!=0)
```

Το πρόγραμμα από ψευδοκώδικα σε γλώσσα μηχανής

Ψευδοκώδικας

Κάθε γραμμή είναι μια εντολή.
Μεταφράζω:
Θέση μνήμης: Εντολή

```
0: R0=0;
1: R1=1;
2: R2=2;
do{
  3: R0=R1;
  4: R1=R2;
  5: R2=R0+R1;
  6:mem[20]=R2;
  7:R3=mem[20];
  8:R0=R0+R0
}
9:while(R2!=0)
```

βοηθητική γλώσσα μηχανής (ψευδο-assembly)

```
i=0;  cpuinst.raminst.mem[0]={cpuinst.OP_SHORT_TO_REG, R0, 8'b00000000};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SHORT_TO_REG, R1, 8'b00000001};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SHORT_TO_REG, R2, 8'b00000010};

i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_MOV, R0, R1, 4'b0000};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_MOV, R1, R2, 4'b0000};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_ADD, R2, R0, R1};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_STORE_TO_MEM, R2, 8'b00010100};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_LOAD_FROM_MEM, R3, 8'b00010100};
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_ADD, R0, R0, R0};

i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_BNZ, R2, 8'b00000011};
```

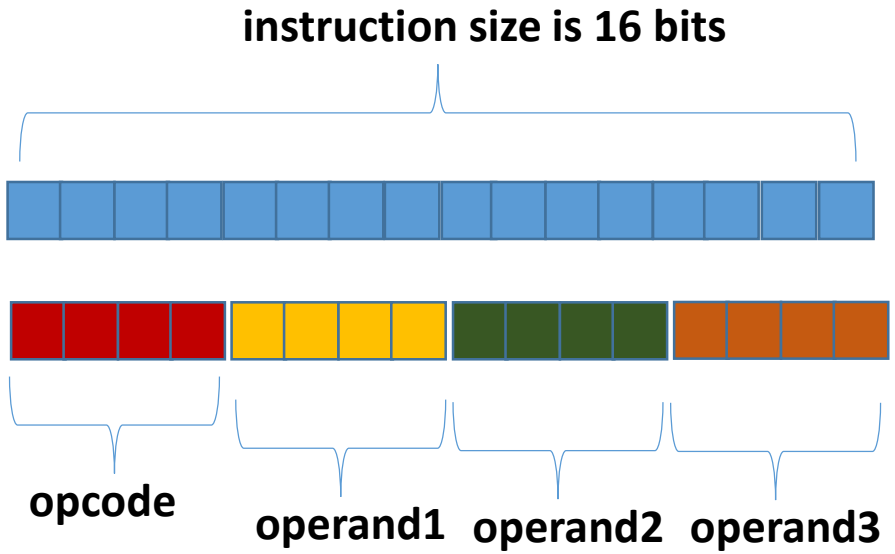
γλώσσα μηχανής

```
0111_0000_0000_0000
0111_0001_0000_0001
0111_0010_0000_0010

0100_0000_0001_0000
0100_0001_0010_0000
0011_0010_0000_0001
0110_0010_0001_0100
0101_0011_0001_0100
0011_0000_0000_0000

1000_0010_0000_0011
```

Εμείς προγραμματίζουμε σε αυτό το στάδιο μέσα στο testbench

Υπενθύμιση των opcodes

```
//opcodes for the supported instruction set
//Opcodes για ALU
parameter  [OPCODE_SIZE-1:0]  OP_AND  = 0; //4'b0000
parameter  [OPCODE_SIZE-1:0]  OP_OR   = 1; //4'b0001
parameter  [OPCODE_SIZE-1:0]  OP_XOR  = 2; //4'b0010
parameter  [OPCODE_SIZE-1:0]  OP_ADD  = 3; //4'b0011

//Opcodes για registers και μνήμη
parameter  [OPCODE_SIZE-1:0]  OP_MOV  = 4; //4'b0100
parameter  [OPCODE_SIZE-1:0]  OP_LOAD_FROM_MEM = 5; //4'b0101
parameter  [OPCODE_SIZE-1:0]  OP_STORE_TO_MEM = 6; //4'b0110
parameter  [OPCODE_SIZE-1:0]  OP_SHORT_TO_REG = 7; //4'b0111
parameter  [OPCODE_SIZE-1:0]  OP_BNZ   = 8; //4'b1000
```



Επεκτασιμότητα του MicroCPU και παραδείγματα βρόγχων

Επέκταση του πλήθους καταχωρητών

Στην δήλωση του καταχωρητή αρχείων μπορούμε να παραμετροποιήσουμε το REGS_NUMBER_WIDTH.

```
module MCPU_Registerfile(op1, op2, op3, RegOp1, alu1, alu2, datatoload, regsetwb, regsetcmd);
parameter WORD_SIZE=8;
parameter OPERAND_SIZE=4;
parameter REGS_NUMBER_WIDTH=4;
parameter REGISTERS_NUMBER=1<<REGS_NUMBER_WIDTH;
.
.
.
```

Το regs_number_width μας δίνει έμμεσα τον αριθμό των καταχωρητών REGISTERS_NUMBER, σύμφωνα με τη σχέση:

$REGISTERS\_NUMBER=2^{REGS\_NUMBER\_WIDTH}$
Αυτό ακριβώς σημαίνει το:
parameter REGISTERS_NUMBER=1<<REGS_NUMBER_WIDTH;

Επομένως αν θέλουμε να έχουμε 16 καταχωρητές πόσο πρέπει να είναι
το REGS_NUMBER_WIDTH? Επίσης, μην ξεχνάτε ότι οι παράμετροι γίνονται overload κατά την δημιουργία των instances από την MCPU, επομένως οι αλλαγές πρέπει να γίνουν εκεί!!!

Εντολές μηχανής (instruction set) του MicroCPU-υπενθύμιση

- **Κλασσικές εντολές επεξεργασίας** που εμπλέκουν την ALU για αριθμητικές και λογικές πράξεις, π.χ.:
  AND operand1 operand2 operand3
  OR operand1 operand2 operand3
  XOR operand1 operand2 operand3
  ADD operand1 operand2 operand3
  Το αποτέλεσμα γράφεται στον operand1. Οι πράξεις εμπλέκουν δύο operands τους operand2 και operand3.
  Operands για τον MicroCPU είναι μόνο οι καταχωρητές του.
- **Εντολές μετακίνησης δεδομένων από καταχωρητή σε καταχωρητή**:
  MOV Rd R
  τα δεδομένα αντιγράφονται από τον καταχωρητή R στον καταχωρητή Rd (το Rd προκύπτει από το register destination).
- **Εντολές φόρτωσης δεδομένων από μνήμη σε καταχωρητή**:
  Load_FROM_MEM Rd address, για λόγους ευκολίας θα την λέω LFM
  αντιγράφει τα δεδομένα από την διεύθυνση μνήμης address στον καταχωρητή Rd.
- **Εντολές αποθήκευσης δεδομένων από καταχωρητή στην μνήμη**:
  STORE_TO_MEM address R, για λόγους ευκολίας θα την λέω STM
  αποθηκεύει στη διεύθυνση μνήμης address τα δεδομένα του καταχωρητή R.
- **Εντολές Αρχικοποίησης τιμών**:
  OP_SHORT_TO_REG Rd value, για λόγους ευκολίας θα την λέω STR
  αντιγράφει την τιμή value των 8 bits στον καταχωρητή Rd.
- **Εντολές διακλάδωσης:**
  BNZ Rc address
  μετακινεί τον program counter στην τιμή address αν ο καταχωρητής Rc δεν είναι 0. (Branch if Not Zero)

```
//OPCODES
OP_AND  = 0; //4'b0000
OP_OR   = 1; //4'b0001
OP_XOR  = 2; //4'b0010
OP_ADD  = 3; //4'b0011
//Opcodes για registers και μνήμη
OP_MOV  = 4; //4'b0100
OP_LOAD_FROM_MEM = 5; //4'b0101
OP_STORE_TO_MEM = 6; //4'b0110
OP_SHORT_TO_REG = 7; //4'b0111
OP_BNZ   = 8; //4'b1000
```

Επέκταση του συνόλου των εντολών

Αρχικά πρέπει να προσθέσουμε το νέο opcode της νέας εντολής (**cpu.v**).

Πρόκειται για εντολή αριθμητικών ή λογικών πράξεων;
Θα εκτελεστεί από την ALU.

Πρόκειται για εντολή μετακίνησης δεδομένων;
Θα εκτελεστεί από το RegisterFile

Πρόκειται για εντολή διακλάδωση;
Θα εκτελεστεί από την MCPU

Επέκταση του συνόλου των εντολών (παράδειγμα sub)

Αρχικά πρέπει να προσθέσουμε το νέο opcode της νέας εντολής (**cpu.v**).

Πρόκειται για εντολή αριθμητικών ή λογικών πράξεων;
Θα εκτελεστεί από την ALU.

Πρόκειται για εντολή μετακίνησης δεδομένων;
Θα εκτελεστεί από το RegisterFile

Παράδειγμα:
Ας υποθέσουμε ότι θέλουμε να βάλουμε την εντολή που εκτελεί την αριθμητική πράξη της **αφαίρεση SUB**.

Θα* είναι της μορφής*: SUB Rd Ra Rb και θα εκτελεί: Rd=Ra-Rb

Θα εκτελείται από την ALU. Άρα επεκτείνουμε τα opcodes που σχετίζονται με την ALU:

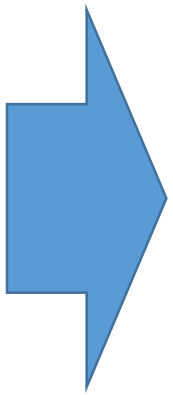
```
//Opcodes για ALU
parameter  [OPCODE_SIZE-1:0]  OP_AND  = 0; //4'b0000
parameter  [OPCODE_SIZE-1:0]  OP_OR   = 1; //4'b0001
parameter  [OPCODE_SIZE-1:0]  OP_XOR  = 2; //4'b0010
parameter  [OPCODE_SIZE-1:0]  OP_ADD  = 3; //4'b0011
parameter  [OPCODE_SIZE-1:0]  OP_SUB  = 4; //4'b0100
//Opcodes για registers και μνήμη
parameter  [OPCODE_SIZE-1:0]  OP_MOV  = 4; //4'b0100
parameter  [OPCODE_SIZE-1:0]  OP_LOAD_FROM_MEM = 5; //4'b0101
parameter  [OPCODE_SIZE-1:0]  OP_STORE_TO_MEM = 6; //4'b0110
parameter  [OPCODE_SIZE-1:0]  OP_SHORT_TO_REG = 7; //4'b0111
parameter  [OPCODE_SIZE-1:0]  OP_BNZ  = 8; //4'b1000
```

```
//Opcodes για ALU
parameter  [OPCODE_SIZE-1:0]  OP_AND  = 0; //4'b0000
parameter  [OPCODE_SIZE-1:0]  OP_OR   = 1; //4'b0001
parameter  [OPCODE_SIZE-1:0]  OP_XOR  = 2; //4'b0010
parameter  [OPCODE_SIZE-1:0]  OP_ADD  = 3; //4'b0011
parameter  [OPCODE_SIZE-1:0]  OP_SUB  = 4; //4'b0100
//Opcodes για registers και μνήμη
parameter  [OPCODE_SIZE-1:0]  OP_MOV  = 5; //4'b0101
parameter  [OPCODE_SIZE-1:0]  OP_LOAD_FROM_MEM = 6; //4'b0110
parameter  [OPCODE_SIZE-1:0]  OP_STORE_TO_MEM = 7; //4'b0111
parameter  [OPCODE_SIZE-1:0]  OP_SHORT_TO_REG = 8; //4'b1000
parameter  [OPCODE_SIZE-1:0]  OP_BNZ  = 9; //4'b1001
```

Επέκταση του συνόλου των εντολών (παράδειγμα sub)

Πρέπει να γίνουν αλλαγές και στην ALU (**alu.v**).

```
module MCPU_Alu(cmd,in1,in2,out,CF);
parameter CMD_SIZE=2;
parameter WORD_SIZE=16;

parameter  [CMD_SIZE-1:0]  CMD_AND  = 0; //2'b00
parameter  [CMD_SIZE-1:0]  CMD_OR   = 1; //2'b01
parameter  [CMD_SIZE-1:0]  CMD_XOR  = 2; //2'b10
parameter  [CMD_SIZE-1:0]  CMD_ADD  = 3; //2'b11
parameter  [CMD_SIZE-1:0]  CMD_SUB  = 3; //2'b11
```



```
module MCPU_Alu(cmd,in1,in2,out,CF);
parameter CMD_SIZE=3;
parameter WORD_SIZE=16;

parameter  [CMD_SIZE-1:0]  CMD_AND  = 0; //3'b000
parameter  [CMD_SIZE-1:0]  CMD_OR   = 1; //3'b001
parameter  [CMD_SIZE-1:0]  CMD_XOR  = 2; //3'b010
parameter  [CMD_SIZE-1:0]  CMD_ADD  = 3; //3'b011
parameter  [CMD_SIZE-1:0]  CMD_SUB  = 4; //3'b100
```

Προστέθηκε το νέο cmd για να υποδεικνύετε στην ALU πότε γίνεται αφαίρεση και αναγκαστικά άλλαξε το πλήθος των bits του cmd από 2 σε 3 bits γιατί αλλιώς δεν έφταναν τα bits για την κωδικοποίηση της αφαίρεσης.

Επίσης, μην ξεχνάτε ότι οι παράμετροι γίνονται overload κατά την δημιουργία των instances από την MCPU, επομένως οι αλλαγές που σχετίζονται με πλήθος bits από παραμετροποίηση πρέπει να γίνουν και εκεί!!!

Στην προκειμένη περίπτωση είναι το ALU_CMD_SIZE στο cpu.v το οποίο πρέπει να γίνει 3 bits

Επέκταση του συνόλου των εντολών (παράδειγμα sub)

Πρέπει να γίνουν επιπλέον αλλαγές στην ALU (**alu.v**).
Ήρθε βασικά η ώρα της υλοποίησης της πράξης που εκτελεί η εντολή:

```
always @ (cmd, in1, in2)
.
.
.
CMD_SUB : begin
        {CF,out} = in1-in2;
end
.
.
.
```

Προσθέτουμε την πράξη στο case των πράξεων της ALU

Επέκταση του συνόλου των εντολών (παράδειγμα sub)

Πρέπει να γίνουν επιπλέον αλλαγές στην Control Unit (**cpu.v**).

Επίσης, μην ξεχνάτε ότι οι παράμετροι γίνονται overload κατά την δημιουργία των instances από την MCPU, επομένως οι αλλαγές που σχετίζονται με πλήθος bits από παραμετροποίηση πρέπει να γίνουν και εκεί!!!

Στην προκειμένη περίπτωση είναι το ALU_CMD_SIZE στο cpu.v το οποίο πρέπει να γίνει 3 bits

```
.
.
.
parameter ALU_CMD_SIZE=3;
.
.
.
```

Γιατί παρακάτω γίνεται:

```
MCPU_Alu #(.CMD_SIZE(ALU_CMD_SIZE), .WORD_SIZE(WORD_SIZE))
aluinst (.cmd(alu_cmd),
        .in1(alu_in1),
        .in2(alu_in2),
        .out(alu_out),
        .CF(CARRY));
```

Επίσης πρέπει να πείτε στην control unit ότι πρόκειται για εντολή εκτέλεσης πράξης:

```
.
.
.
case(opcode)
        OP_AND,OP_OR,OP_XOR,OP_ADD, OP_SUB:
    begin
      regset_cmd <= #2 regfileinst.NORMAL_EX;
    end
```

Άρα πρέπει να προστεθεί η εντολή στο case των normally executed από την Control Unit

Εκτέλεση αφαίρεσης με τους 16 νέους καταχωρητές

Γράφουμε ένα testbench για να τεστάρουμε τους 16 καταχωρητές και την αφαίρεση:

```
///SUB test sequence
i=0;  cpuinst.raminst.mem[0]={cpuinst.OP_SHORT_TO_REG, R0, 8'b00001010}; //0: R0=10;
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SHORT_TO_REG, R1, 8'b00000011}; //1: R1=3;
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SUB, R12, R0, R1};            //2: R12=R0-R1
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SUB, R12, R1, R0};            //3: R12=R1-R0
```

Βρόγχος if (R1!=α)

Αν θέλουμε να πούμε
If(R1!=0)
μπορούμε πολύ εύκολα να κάνουμε:
BNZ R1 Διεύθυνση

Αν θέλουμε να πούμε
If(R1!=1)
Τότε χρειαζόμαστε το εξής:
Rx=R1-1
BNZ Rx Διεύθυνση

Γενικά όταν έχουμε σχέση βρόγχου της μορφής
If(R!=a)
Τότε κάνουμε το εξής:
Rx=R-a;
BNZ Rx Διεύθυνση

Παράδειγμα στόχος:

```
If(R1!=4)
{
  R5=R5+6;
}
else
{
 R7=R7 xor R8;
}
R9=R2
```

OP_SHORT_TO_REG Rd value,
για λόγους ευκολίας θα την λέω STR

Παράδειγμα λύση:

```
//φτιάχνω πρώτα τα constants
1: STR R3 4 //R3=4
2: STR R4 6 //R4=6

3: SUB R2 R1 R3
4: BNZ R2 7//πήγαινε στην διεύθυνση που είναι μέσα στο if
  5: XOR R7 R7 R8 //κώδικας μέσα στο else
6: BNZ R3 8 // ξέρουμε ότι το R3 δεν είναι 0
//οπότε είναι σα να γράφουμε πήγαινε στην γραμμή 8
  7: ADD R5, R4 //κώδικας μέσα στο if
8: MOV R9 R2
```

Ειδική περίπτωση if (R1!=1)

Μπορεί να γίνει χωρίς αφαίρεση με χρήση της λογικής μάσκας XOR

Αν θέλουμε να ελέγξουμε ότι ο
R1=XXXX_XXXX_XXXX_XXXX, είναι 1, τότε μπορούμε να φτιάξουμε την λογική μάσκα:
R2=0000_0000_0000_0001 και να υπολογίσουμε το αποτέλεσμα της πράξης

R3= R1 xor R2
R3= XXXX_XXXX_XXXX_XXXX xor 0000_0000_0000_0001

XXXX_X0XX_0XXX_XXX1
xor 0000_0000_0000_0001
= 0000_0000_0000_0000

Το R3 θα είναι 0 μόνο και μόνο όταν το R1 είναι μονάδα.

Άρα:
MOV R2 0000_0000_0000_0001
XOR R3 R1 R2
BNZ R3 διεύθυνση

Έλεγχος μονά-ζυγά

Μας νοιάζει δηλαδή να ελέγξουμε μόνο το πρώτο bit (LSB) ενός καταχωρητή αν είναι 1 ή 0
R1=XXXX_XXXX_XXXX_XXXX
Μπορεί να γίνει με χρήση της λογικής μάσκας AND

Αν θέλουμε να ελέγξουμε ότι ο
R1=XXXX_XXXX_XXXX_XXXX, είναι μονός, τότε μπορούμε να φτιάξουμε την λογική μάσκα:
R2=0000_0000_0000_0001 και να υπολογίσουμε το αποτέλεσμα της πράξης

R3= R1 AND R2
R3= XXXX_XXXX_XXXX_XXXX AND 0000_0000_0000_0001

X1XX_XX1X_XX1X_XXX?
xor 0000_0000_0000_0001
===0000_0000_0000_000?=R3
Το R3 θα είναι 0 μόνο και μόνο όταν το R1 είναι ζυγός.

Άρα:
MOV R2 0000_0000_0000_0001
AND R3 R1 R2
BNZ R3 διεύθυνση

Άλλος τρόπος:
Μπορούμε να ολισθήσουμε τον καταχωρητή αριστερά κατά 15 θέσεις (BITS-1) ώστε να μείνει μόνο το LSB μέσα στον καταχωρητή. Μετά με έλεγχο BNZ μπορούμε να δούμε αν είναι μονός ή ζυγός

Ολισθήσεις και NOT

Logical Shift Left (LSL) : Χρησιμοποιούμε τον operator <<. Π.χ.: b=a<<3, το b είναι το a ολισθημένο κατά 3 θέσεις αριστερά.

Logical Shift Right (LSR): Χρησιμοποιούμε τον operator >>. Π.χ.: b=a>>3, το b είναι το a ολισθημένο κατά 3 θέσεις δεξιά.

NOT Rd Ra, αντιστρέφει τα ψηφία του Ra και τα γράφει στον Rd (γίνεται με out=~in1 στην ALU)

Περιφερειακές συσκευές και Interrupts

Συνθέσιμη Υλοποίηση περιφερειακών

Η υλοποίηση των περιφερειακών στην μνήμη δεν συμβαίνει επειδή οι μνήμες κατασκευάζονται με ειδικές τεχνολογίες κατασκευής και μπορούμε να έχουμε πρόσβαση ανάγνωσης/εγγραφής μέσω των controllers ram.

Για τον λόγο αυτό η προηγούμενη υλοποίηση του περιφερειακού είναι καθαρά για λόγους προσομοίωσης, δεν γίνεται έτσι στην πραγματικότητα, δηλαδή δεν είναι συνθέσιμο.

Υλοποίηση σταθμοσκόπησης (polling)

Στην πραγματικότητα γράφεται ένα μπλοκ μέσα στο cpu το οποίο ανανεώνει περιοδικά τις εισόδους των περιφερειακών διαβάζοντας τα δεδομένα από τις θέσεις μνήμης που τους αντιστοιχούν στο memory map. Μετά το μπλοκ αυτό συλλέγει τις αποκρίσεις των περιφερειακών και τις αποθηκεύει στις θέσεις μνήμης που τους αντιστοιχούν μέσα στο memory map.

Το αρνητικό αυτής της υλοποίησης είναι ότι πρέπει να απασχολεί την μνήμη για να γράφει δεδομένα από/προς τα περιφερειακά με την τεχνική round robin. Όσο μεγαλώνει ο αριθμός των περιφερειακών λοιπόν χάνονται κύκλοι σε αντιγραφές από και προς την μνήμη.

Υλοποίηση με interrupts

Στην υλοποίηση αυτή το μπλοκ είναι ευαίσθητο σε μερικά μόνο σήματα. Έτσι απασχολεί την μνήμη μόνο όταν πραγματικά χρησιμοποιείτε το περιφερειακό και έτσι αποτρέπεται η άσκοπη χρήση της μνήμης. Παράδειγμα, είναι το πληκτρολόγιο, το οποίο ενεργοποιεί interrupt σήματα, τα οποία ενεργοποιούν εγγραφές προς την μνήμη.

Software vs Hardware διαχείριση συσκευών

Στο σημείο αυτό πρέπει να πούμε πως η ίδια ορολογία εμφανίζεται και στο λογισμικό στον τομέα των λειτουργικών συστημάτων, το οποίο μπορεί να έχει αρμοδιότητες στη διαχείριση των περιφερειακών συσκευών. Στην περίπτωση του λειτουργικού συστήματος, τα παραπάνω γίνονται από το λογισμικό. Δηλαδή, το λογισμικό αναλαμβάνει την διαχείριση της ροής του κώδικα assembly των προγραμμάτων που οδηγούν τις συσκευές και το κάνει αυτό είτε με χρήση σταθμοσκόπησης στον κώδικα όλων των οδηγών (έτσι γινόταν παλιά από το Dos) είτε με χρήση interrupts.

Interrupts

Τα interrupts λαμβάνονται από τον μικροεπεξεργαστή είτε από το λογισμικό είτε από τα περιφερειακά και διακόπτουν τη ροή εκτέλεσης. Η ροή μεταβαίνει σε σημείο όπου γίνεται εξυπηρέτηση του interrupt και μετά επιστρέφεται στο σημείο όπου έγινε η διακοπή. Τα Interrupts έχουν πολλές διαβαθμίσεις. Από το hardware μέχρι το software – ακόμα και σε επίπεδο assembly. Θα δούμε ένα παράδειγμα στον MicroCPU πως συνδέονται τα interrupts με το λογισμικό. Όταν κάνετε ένα fopen ή όταν κάνετε μία scanf στην ουσία χρησιμοποιείτε μια μορφή interrupts.