ΗΥ-225: Οργάνωση Υπολογιστών
Άνοιξη 2006

Τμ. Επ. Υπολογιστών
© Πανεπιστήμιο Κρήτης

Σειρά Ασκήσεων 11: Βασική FSM Ελέγχου του Επεξεργαστή

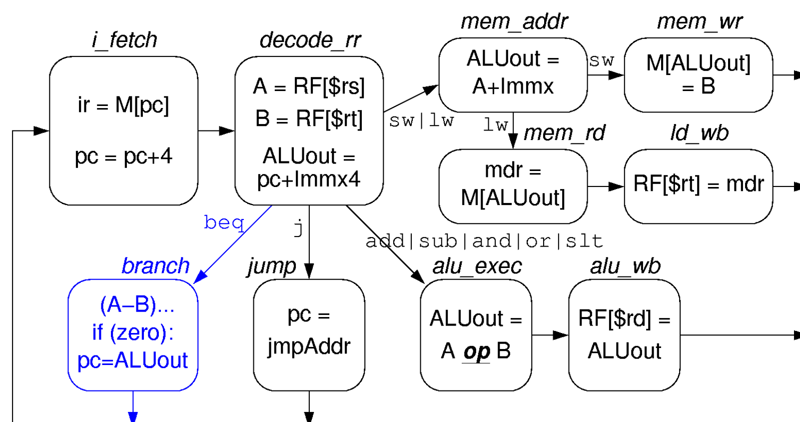
Προθεσμία έως Παρασκευή 14 Απριλίου 2006, ώρα 23:59 (βδομάδα 9)

11.1 Βασική FSM και Εξισώσεις Μεταφοράς Καταχωρητών:

Γιά να λειτουργήσει το datapath του επεξεργαστή που περιγράψατε στην άσκηση 10.1 (υλοποίηση πολλαπλών κύκλων ρολογίου), χρειάζεται μιά Μηχανή Πεπερασμένων Καταστάσεων (**FSM** - Finite State Machine) που να γεννά τα επιθυμητά σήματα ελέγχου σε κάθε κύκλο ρολογιού. Θα ξεκινήσουμε, σε αυτή τη σειρά ασκήσεων, με τη βασική FSM ελέγχου που είδαμε στο μάθημα, που περιγράφεται στην παράγρ. 5.4 του βιβλίου (σελ. 377-398), και που δίδεται στο σχήμα εδώ με τις εξισώσεις μεταφοράς καταχωρητών γιά την κάθε κατάσταση. Οι τιμές των σημάτων ελέγχου, στην κάθε κατάσταση, προκύπτουν από αυτές τις εξισώσεις όπως εξηγήσαμε στο μάθημα, και όπως εκτίθεται στο βιβλίο, σελίδες 383-396. Σε αυτή τη σειρά ασκήσεων θα υλοποιήσετε **μόνο** τις εντολές:

sw, lw;   add, sub, and, or, slt;   j

που φαίνονται με μαύρο χρώμα στο διάγραμμα --τις υπόλοιπες εντολές, περιλαμβανόμενων και των διακλαδώσεων, θα τις υλοποιήσετε σε επόμενη σειρά ασκήσεων.



Κάθε κουτί στο διάγραμμα αυτό παριστά μία κατάσταση της FSM. Το όνομα πάνω από το κουτί είναι το όνομα της κατάστασης. Η FSM που έχουμε εδώ παραμένει ακριβώς ένα κύκλο ρολογιού σε κάθε κατάσταση. Τον επόμενο κύκλο ρολογιού πηγαίνει στην επόμενη κατάσταση του διαγράμματος. Όταν υπάρχουν περισσότερες από μία επόμενες καταστάσεις, όπως συμβαίνει στις καταστάσεις *decode_rr* και *mem_addr*, τότε η μετάβαση που θα κάνει η FSM εξαρτάται από την εκάστοτε εκτελούμενη εντολή, δηλαδή τα περιεχόμενα του καταχωρητή εντολών (IR): θα ακολουθηθεί η μετάβαση πάνω στο βέλος της οποίας είναι γραμμένο το opcode της παρούσας εντολής.

Μέσα στο κουτί κάθε κατάστασης είναι γραμμένες οι εξισώσεις μεταφοράς καταχωρητών (register transfer) που περιγράφουν τις λειτουργίες που πρέπει να γίνουν όποτε η FSM βρίσκεται σε εκείνη την κατάσταση, δηλαδή, στην περίπτωσή μας, μέσα σε εκείνον τον κύκλο ρολογιού. Πρόκειται γιά την περιγραφή του επεξεργαστή μας σε "Επιπεδο Μεταφοράς Καταχωρητών" (**RTL** - Register Transfer Level) --ένα επίπεδο αφαίρεσης αρκούντως αφηρημένο γιά να βολεύει τους σχεδιαστές, αλλά και αρκούντως συγκεκριμένο γιά να επιτρέπει σχεδόν αυτόματη ή και πλήρως αυτόματη σύνθεση του τελικού κυκλώματος από εργαλεία (λογισμικό) σχεδίασης, όπως το συχνά χρησιμοποιούμενο πακέτο "Synopsys". Οι μεταφορές καταχωρητών εδώ είναι:

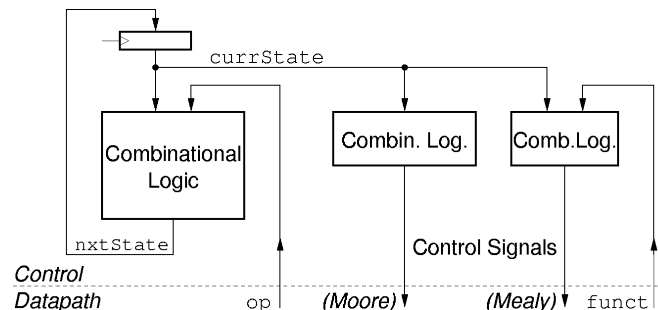
- Ανάγνωση Εντολής από τη Μνήμη: ir = M[pc];
- Ανάγνωση Καταχωρητών Πηγής από το Αρχείο Καταχωρητών: A = RF[$rs]; B = RF[$rt]; Το σύμβολο $rs σημαίνει το σχετικό πεδίο του καταχωρητή εντολών, `ir[25:21]`, και ομοίως $rt σημαίνει `ir[20:16]`. Τους καταχωρητές αυτούς τους διαβάζουμε πάντα, πριν (εν παραλλήλω με) την αποκωδικοποίηση της εντολής και τη διαπίστωση του ποιά εντολή είναι και τι format έχει. Υπάρχουν εντολές γιά τις οποίες τα πεδία ir[25:21] και/ή ir[20:16] δεν είναι αριθμοί καταχωρητών, άρα οι τιμές που διαβάζουμε είναι "σκουπίδια", όμως αυτό δεν πειράζει αφού αυτές

θα αγνοηθούν στη συνέχεια, ενώ με την απλή ανάγνωση που κάναμε δεν έχουμε καταστρέψει καμιά πληροφορία. Από την άλλη μεριά, γιά όσες εντολές χρειάζονται μία ή και τις δύο τιμές που διαβάσαμε, πετυχαίνουμε να έχουμε τις τιμές αυτές διαθέσιμες όσο νωρίτερα γίνεται, χωρίς να υφιστάμεθα την καθυστέρηση της προηγούμενης αποκωδικοποίησης της εντολής. Γιά να είναι αυτό εφικτό, τα πεδία $rs και $rt (καταχωρητές πηγής) βρίσκονται πάντα στην **ίδια, σταθερή θέση** μέσα σε όλες τις εντολές MIPS που τα χρησιμοποιούν σαν καταχωρητές πηγής, **ανεξαρτήτως format** αυτών των εντολών.
- Υπολογισμός Διεύθυνσης Μνήμης: ALUout = A + Immx;
- Προσπέλαση Μνήμης γιά Δεδομένα: mdr = M[ALUout]; (ανάγνωση μνήμης), ή M[ALUout] = B; (εγγραφή μνήμης).
- Πράξη Αριθμητικής/Λογικής Μονάδας γιά εντολή ALU R-format: ALUout = A **op** B; όπου η πράξη **op** εξαρτάται από τη συγκεκριμένη εντολή που εκτελούμε μέσα σε αυτή την κατηγορία εντολών. Το σήμα ελέγχου της ALU σε αυτή την περίπτωση είναι παράδειγμα σήματος τύπου "Mealy", και όχι "Moore" --βλέπε παρακάτω.
- Εγγραφή Αποτελέσματος στο Αρχείο Καταχωρητών: RF[$rt] = mdr; (δεδομένα από ανάγνωση μνήμης), ή RF[$rd] = ALUout; (δεδομένα από εντολή ALU R-format).
- Εκτέλεση Άλματος χωρίς Συνθήκη: pc = jmpAddr;
- Διεύθυνση Προορισμού Διακλάδωσης υπό Συνθήκη (αφορά επόμενη και όχι αυτή την άσκηση): γιά εκπαιδευτικούς σκοπούς, υπολογίζουμε τη συνθήκη διακλάδωσης μέσω αφαίρεσης (A-B) στην ALU, και στη συνέχεια κοιτάμε αν το αποτέλεσμα της αφαίρεσης είναι μηδέν. Επίσης, γιά λόγους οικονομίας κυκλωμάτων και γιά εκπαιδευτικούς σκοπούς, ο υπολογισμός της διεύθυνσης προορισμού της διακλάδωσης γίνεται στον αθροιστή της μίας και μοναδικής ALU που διαθέτουμε. Γιά λόγους ταχύτητας, επιδιώκουμε οι διακλαδώσεις να εκτελούνται σε τρείς μόνο, αντί τεσσάρων, κύκλους ρολογιού. Αφού ο υπολογισμός της διεύθυνσης προορισμού χρειάζεται την ALU, αφού η ALU είναι απασχολημένη τον 3ο κύκλο ρολογιού, και αφού προσπαθούμε να εκτελέσουμε τη διακλάδωση σε 3 μόνο κύκλους, πρέπει ο υπολογισμός της διεύθυνσης προορισμού, ALUout = pc+Immx4, να γίνει τον 1ο ή τον 2ο κύκλο. Τον 1ο κύκλο (ενώ διαβάζουμε την εντολή από τη μνήμη), δεν είναι δυνατόν να γίνει η πρόσθεση pc+Immx4, διότι δεν είναι γνωστό ακόμα το Immx4.
  Τον 2ο κύκλο, όμως, μπορεί να υπολογιστεί η διεύθυνση προορισμού, αφού το Immx4 είναι κομάτι του IR, σε γνωστή θέση, άρα είναι γνωστό από την αρχή του 2ου κύκλου. Ακολουθούμε λοιπόν τη λύση αυτή, και κάνουμε την πρόσθεση πάντα, ταυτόχρονα και ανεξάρτητα από την αποκωδικοποίηση της εντολής. Αυτό σημαίνει ότι σε πολλές περιπτώσεις το αποτέλεσμα αυτής της πρόσθεσης θα είναι "σκουπίδια", όπως π.χ. στις περιπτώσεις εντολών με R-format, όπου το πεδίο ir[15:0] δεν είναι αριθμός "immediate" αλλά σύνθεση τριών άλλων, άσχετων πεδίων. Όμως, αυτό δεν μας πειράζει, αφού οι εντολές αυτές αγνοούν το αποτέλεσμα αυτής της πρόσθεσης (γράφουν άλλο, ανεξάρτητο αποτέλεσμα στον ALUout). Οι εντολές διακλάδωσης, από την άλλη μεριά, όταν στον τρίτο κύκλο χρειαστούν τη διεύθυνση προορισμού, θα την έχουν έτοιμη και έγκυρη. Η απόφαση γιά τη χρήση της (pc=ALUout, συνθήκη διακλάδωσης αληθής) ή την αγνόησή της (pc αμετάβλητος, συνθήκη διακλάδωσης ψευδής), αρκεί να ληφθεί στο τέλος του 3ου κύκλου ρολογιού, ελέγχοντας το σήμα επίτρεψης φόρτωσης (pcld) κατά τον χρόνο εκείνο. Πρόκειται γιά περίπτωση καθυστερημένου σήματος "Mealy", όπως θα πούμε σε επόμενη άσκηση.
- Αύξηση του Μετρητή Προγράμματος (PC) κατά 4: γιά λόγους οικονομίας κυκλωμάτων και γιά εκπαιδευτικούς σκοπούς, η αύξηση του PC κατά 4 γίνεται στον αθροιστή της μίας και μοναδικής ALU που διαθέτουμε --προφανώς σε κύκλους ρολογιού που αυτή είναι ελεύθερη από άλλες υποχρεώσεις. Γιά τις εντολές μνήμης και αριθμητικής/λογικής, η αύξηση του PC κατά 4 θα μπορούσε να γίνει κατά τον 4ο (ή 5ο) κύκλο ρολογιού τους. Όμως, γιά λόγους απλότητας και ομοιομορφίας, προτιμήθηκε η αύξηση του PC κατά 4 να γίνεται κατά τον πρώτο κύκλο (i_fetch) όλων ανεξαιρέτως των εντολών, ταυτόχρονα (εν παραλλήλω), δηλαδή, με την ανάγνωση της εντολής από τη μνήμη. Παρατηρήστε ότι η ανάγνωση γίνεται από τη διεύθυνση μνήμης που υποδεικνύει η παλαιά (προ της αύξησης) τιμή του PC. Παρατηρήστε επίσης ότι η αύξηση αυτή κατά τον πρώτο κύκλο θα βολεύει και την υλοποίηση των εντολών jal, αργότερα, δεδομένου ότι αυτές πρέπει να αποθηκεύσουν στον $31 αυτήν ακριβώς την --αυξημένη κατά 4-- τιμή. Παρατηρήστε τέλος ότι η διεύθυνση προορισμού των διακλαδώσεων υπό συνθήκη ορίστηκε επίτηδες να είναι (PC+4) + 4*Imm16, αντί απλώς PC + 4*Imm16, γιά να βολεύει σε σχέση με αυτή την αύξηση του PC κατά 4, στον πρώτο κύκλο ρολογιού, "στα τυφλά", πάντα, γιά όλες τις εντολές.

11.2 Βασική Υλοποίηση FSM και Σημάτων Ελέγχου:

Προς το παρόν, θα υλοποιήσουμε την FSM ελέγχου με τον απλό τρόπο που φαίνεται στο παρακάτω σχήμα. Η "καρδιά" της FSM είναι ο καταχωρητής **κατάστασης** --πάνω αριστερά στο σχήμα-- που

περιέχει (θυμάται) την παρούσα κατάσταση, "**currState**", του συστήματος. Ενα μπλοκ Συνδυαστικής Λογικής (combinational logic) --κάτω από αυτόν στο σχήμα-- υπολογίζει (αποφασίζει) ποιά θα είναι η επόμενη κατάσταση, "**nxtState**", αμέσως μετά την επερχόμενη ακμή του ρολογιού, σαν συνάρτηση της παρούσας κατάστασης και των εισόδων του κυκλώματος ελέγχου --εν προκειμένω του op, δηλαδή του ir[31:26]. Οι έξοδοι του κυκλώματος ελέγχου, δηλαδή τα σήματα ελέγχου του datapath, είναι επίσης συναρτήσεις της παρούσας κατάστασης, και --μερικά απ' αυτά-- μπορεί να είναι συναρτήσεις και ορισμένων εισόδων του κυκλώματος ελέγχου --εν προκειμένω του funct, δηλαδή του ir[5:0], και αργότερα και άλλων, όπως π.χ. του "zero".



- Οι έξοδοι "τύπου **Moore**" είναι συναρτήσεις μόνο της παρούσας κατάστασης, --όχι και των εισόδων της FSM (οι είσοδοι επηρεάζουν μόνο την επόμενη κατάσταση). Το πλεονέκτημά τους είναι ότι απλοποιούν τη σχεδίαση, ενώ το μειονέκτημά τους είναι ότι ένα τέτοιο κύκλωμα ελέγχου αδυνατεί να αντιδράσει σε εξωτερικά "ερεθίσματα" (σήματα, εισόδους) στον ίδιο κύκλο ρολογιού όπου αυτά εμφανίζονται, και μπορεί μόνο να αντιδράσει σε αυτά στον επόμενο κύκλο ρολογιού, και μάλιστα χρειάζεται και διαφορετικές καταστάσεις γιά την κάθε διαφορετική του αντίδραση.
- Οι έξοδοι "τύπου **Mealy**" είναι συναρτήσεις και της παρούσας κατάστασης, και των εισόδων της FSM. Τα πλεονεκτήματά τους είναι ότι επιτρέπουν οικονομία στο πλήθος καταστάσεων, καθώς και ταχεία αντίδραση σε εξωτερικά ερεθίσματα. Όμως, το τελευταίο απαιτεί και προσεκτικότερη σχεδίαση: όσα σήματα ελέγχου τύπου Mealy εξαρτώνται από αργοπορημένα σήματα εισόδου, θα είναι και αυτά με τη σειρά τους πολύ αργοπορημένα. Στην παρούσα άσκηση, το μόνο σήμα εισόδου στο κύκλωμα ελέγχου που επηρεάζει εξόδους τύπου Mealy είναι το πεδίο funct. Το σήμα αυτό σταθεροποιείται πολύ νωρίς μέσα στον κύκλο ρολογιού διότι είναι κατευθείαν έξοδος του καταχωρητή ir και άρα σταθεροποιείται ίσα-ίσα μόλις περάσει η καθυστέρηση εξόδου του καταχωρητή ir μετά την ακμή ρολογιού.

Και οι δύο παραπάνω τύποι σημάτων ελέγχου έχουν την παρακάτω ιδιότητα: τα σήματα ελέγχου υπολογίζονται στον ίδιο κύκλο στον οποίο τα χρησιμοποιεί το datapath. Θεωρώντας τώρα το σύνολο του επεξεργαστή, δηλ. και το datapath και το κύκλωμα ελέγχου, είναι σαν να χωρίζουμε τον κύκλο του επεξεργαστή σε δύο μέρη: στο πρώτο μέρος του κύκλου γίνεται ο υπολογισμός των σημάτων ελέγχου και το datapath "κάθεται", ενώ στο δεύτερο μέρος του κύκλου το datapath εργάζεται με βάση τα μόλις υπολογισμένα σήματα ελέγχου και το κύκλωμα ελέγχου "κάθεται". Σε επόμενη άσκηση θα δούμε μια τεχνική που επιτρέπει στο κύκλωμα ελέγχου και στο datapath να δουλεύουν ταυτόχρονα, πετυχαίνοντας έτσι υψηλότερη ταχύτητα λειτουργίας του επεξεργαστή.

Τέλος, παρατηρήστε ότι το block συνδυαστικής λογικής για τον υπολογισμό των σημάτων ελέγχου είναι "κοντύτερο" (στον κατακόρυφο άξονα) από το block για τον υπολογισμό της επόμενης κατάστασης. Το παραπάνω υποδηλώνει ότι θέλουμε τα σήματα ελέγχου να υπολογίζονται όσο γίνεται πιο γρήγορα, διότι χρησιμοποιούνται από το datapath στον ίδιο κύκλο ρολογιού, ενώ η συνδυαστική λογική για τον υπολογισμό της επόμενης κατάστασης έχει στη διάθεση της ολόκληρο τον κύκλο ρολογιού για να εκτελέσει τον υπολογισμό της.

Άσκηση 11.3: Περιγραφή FSM και Μεταβάσεων σε "Συνθέσιμη Verilog"

Περιγράψτε το κύκλωμα ελέγχου του επεξεργαστή σας, που θα ακολουθεί την FSM που δόθηκε στην αρχή και θα υλοποιεί μόνο τις εντολές που αναφέρθηκαν στην αρχή της εκφώνησης. Γράψτε την περιγραφή σας σαν ένα καινούριο module, "`control`", στο αρχείο "`control11.v`". Το module αυτό θα παίρνει σαν είσοδο το ρολόϊ, και βέβαια θα έχει σαν εισόδους ή εξόδους όλα τα σήματα επικοινωνίας με το datapath που είχαν καθοριστεί στην άσκηση 10.1.

Η περιγραφή της FSM θα γίνει με το στυλ "RTL" (Register Transfer Level) της Verilog, που είναι ο συνηθισμένος "συνθέσιμος" τρόπος που γράφουμε στις γλώσσες περιγραφής υλικού. Το στυλ "RTL" σημαίνει ότι δηλώνουμε τους καταχωρητές που περιέχει το κύκλωμά μας και τη λογική μεταξύ των

καταχωρητών *χωρίς* να υπολογίζουμε εμείς τις καθυστερήσεις του κυκλώματός μας. Ξέρουμε (κυρίως από εμπειρία) πόση λογική "χωράει" σε ένα κύκλο ρολογιού που θέλουμε να πετύχουμε, και απλώς την περιγράφουμε. Έπειτα, τα εργαλεία σύνθεσης αναλύουν το κύκλωμα, το μεταφράζουν σε λογικές πύλες και flip-flops, το βελτιστοποιούν, και υπολογίζουν επακριβώς τις καθυστερήσεις (βήματα που δεν θα γίνουν στα πλαίσια αυτού του μαθήματος...). Για την FSM της άσκησης, θεωρούμε ότι όλη η λογική της χωράει σε έναν κύκλο ρολογιού.

Ξεκινήστε ορίζοντας το ποιές είναι οι καταστάσεις της FSM και πώς καθορίζεται η επόμενη κατάσταση με βάση την παρούσα κατάσταση και την τιμή των εισόδων. Ακολουθείστε το παρακάτω μικρό παράδειγμα με τις εξηγήσεις που δίδονται μετά από αυτό.

```
`define LW      6'b100011
`define SW      6'b101011

parameter [3:0]            // symbolic constants for state encoding
    i_fetch    = 4'b0001,  //   --"one hot" encoding style
    decode_rr  = 4'b0010,
    mem_addr   = 4'b0100,
    alu_exec   = 4'b1000;

reg  [3:0] currState;      // current State: sequential logic
reg  [3:0] nxtState;       // next State: output of comb. logic
                           //   (although "reg" --see below)
// state register
always @(posedge clk) begin
        currState <= #0.2  nxtState;
end

// define nxtState as a function of currState and op:
always @(currState or op) begin  // whenever any input changes:
        case (currState)
            i_fetch:
                nxtState <= #0.5  decode_rr;
            decode_rr:
                if ((op == `LW) || (op == `SW))
                    nxtState <= #0.5  mem_addr;
                else if ( ... )
                    nxtState <= #0.5  ...
                else
                    nxtState <= #0.5  alu_exec;
            mem_addr:
                nxtState <= #0.5  ...
            alu_exec:
                nxtState <= #0.5  ...
            default:
                nxtState <= #0.5  i_fetch;
        endcase
end
```

Ορισμός και Κωδικοποίηση Καταστάσεων:
Αρχικά ορίζουμε ποιές είναι οι καταστάσεις της FSM, δίνοντας τους συμβολικά ονόματα (i_fetch, decode_rr, mem_addr, alu_exec,...) και μία προτεινόμενη, πιθανή κωδικοποίηση (0001, 0010, 0100, 1000,...). Αυτό το κάνουμε με την εντολή "parameter" της Verilog. Η διαφορά της "parameter" από την "define" είναι ότι η τιμή για τα συμβολικά ονόματα που ορίζονται με τη βοήθεια της "parameter" μπορεί να αλλάξει αργότερα --έτσι το εργαλείο σύνθεσης θα μπορέσει να βελτιστοποιήσει, πιθανόν, την κωδικοποίηση των καταστάσεων. Στο παραπάνω παράδειγμα διαλέξαμε κωδικοποίηση του στυλ "one hot", δηλαδή ο καταχωρητής κατάστασης έχει τόσα bits όσες και οι καταστάσεις συνολικά, κάθε bit αντιστοιχεί σε μία συγκεκριμένη κατάσταση, και κάθε κατάσταση κωδικοποιείται με όλα τα bits μηδέν εκτός από το "δικό της" bit που είναι "αναμένο" (ζεστό, hot). Αυτό το στυλ κωδικοποίησης ταιριάζει σε FSM με όχι πολύ μεγάλο πλήθος καταστάσεων, και συνήθως δίνει υψηλή ταχύτητα λειτουργίας της FSM. Συμπληρώστε τον κατάλογο με όλες τις καταστάσεις που θα χρειαστείτε, και διορθώστε την κωδικοποίησή τους (αν ακολουθείστε το στυλ one hot, προσέξτε πόσα bits χρειάζεστε).

Καταχωρητής Κατάστασης:
Ο καταχωρητής κατάστασης φορτώνεται σε *κάθε ακμή ρολογιού* με την νέα κατάσταση, `nxtState`, που υπολογίσαμε στον προηγούμενο κύκλο ρολογιού. Ο καταχωρητής κατάστασης περιγράφεται με τον κώδικα "*always @(posedge clk) ...*". Το παραπάνω σημαίνει: όταν έρθει η θετική ακμή του ρολογιού εκτέλεσε τον κώδικα μέσα στο block που ακολουθεί, δηλ. δες ποιά είναι τώρα (ακριβώς πριν την ακμή) η τιμή του "nxtState" και ανάθεσέ την στο "currState" αφού πρώτα περάσουν 0.2 ns (καθυστέρηση εξόδου του καταχωρητή). Το εργαλείο σύνθεσης καταλαβαίνει τον παραπάνω κώδικα και παράγει έναν

καταχωρητή (με τις πραγματικές πλέον καθυστερήσεις και όχι αυτές που εμείς υποθέσαμε για απλότητα).

Υπολογισμός Επόμενης Κατάστασης:
Ο υπολογισμός της επόμενης κατάστασης, `nxtState`, γίνεται με συνδυαστική λογική βάσει της παρούσας κατάστασης, currState, και των εισόδων της FSM, όπως φαίνεται στα αριστερά του διαγράμματος block. Επειδή η συνδυαστική αυτή λογική είναι αρκετά μεγάλη, με πολλές υποπεριπτώσεις, είναι πιο βολικό η περιγραφή της να γίνει με τις εντολές "**case**" και "**if**" της Verilog, αναθέτοντας (assign) διάφορες τιμές στη nxtState στους διάφορους κλάδους της "**case**" και των "**if-then-else**". Γιά να δουλέψει αυτό, σε Verilog, πρέπει το nxtState να δηλωθεί σαν "reg" και όχι σαν "wire".

> [Στα wire επιτρέπεται μόνο σύνδεση με modules ή continuous assignment. Στα continuous assignments επιτρέπονται μόνο αριθμητικές/λογικές πράξεις και πράξεις επιλογής όπως "(a==b) ? x : y", αλλά όχι case ή if-then-else. Τα case και if-then-else επιτρέπονται μόνο μέσα σε behavioral blocks, δηλαδή μέσα σε "initial" και "always". Αναθέσεις (assignments) μέσα σε behavioral blocks επιτρέπονται μόνο σε κόμβους τύπου reg, και όχι σε wire. Οι κόμβοι τύπου reg έχουν "μνήμη": όσο και όταν δεν τους ανατίθεται μιά νέα τιμή αυτοί διατηρούν την παλαιά τους τιμή. Οταν λοιπόν, γιά λόγους ευκολίας μας, θέλουμε να περιγράψουμε συνδυαστική λογική μέσω behavioral blocks, χρειάζεται **προσοχή** να αναθέτουμε πάντα νέα τιμή στην έξοδο όποτε αλλάζει κάποια είσοδος, και η νέα τιμή να είναι συνάρτηση μόνο των εισόδων. Αν υπάρχει κάποια υποπερίπτωση των τιμών εισόδου κατά την οποία δεν γίνεται ανάθεση νέας τιμής στην έξοδο (π.χ. ξεχάσαμε το "else" σε κάποιο "if-then", ή ξεχάσαμε το "default" σε κάποιο "case"), τότε σημαίνει ότι θέλουμε να διατηρείται η παλαιά τιμή, που σημαίνει μνήμη.]

Παρ' ότι, λοιπόν, το nxtState δηλώνεται σαν reg αντί wire, εμείς θα ορίσουμε την συμπεριφορά του έτσι ώστε να προκύπτει ότι πρόκειται γιά συνδυαστική λογική. Αυτό το πετυχαίνουμε με τους εξής τρόπους:

- **(α)** Η παρένθεση του "always" statement περιλαμβάνει *όλες* τις εισόδους από τις οποίες εξαρτάται το nxtState (και χωρίς περιορισμό σε ορισμένες μόνο ακμές τους --posedge ή negedge). Αυτό σημαίνει ότι το μπλοκ αυτό του κώδικα (always block) πρέπει να ξαναεκτελείται *πάντα* όποτε αλλάζει η τιμή οιασδήποτε από αυτές τις εισόδους. Εάν σε επόμενη άσκηση το nxtState εξαρτάται και από το funct ή και από άλλες εισόδους, μην ξεχάσετε να προσθέστε όλες τους, με "or" ανάμεσά τους, σε αυτό το "sensitivity list" (κατάλογο ευαισθησιών ή εξαρτήσεων).
- **(β)** Αναθετουμε τιμή στο nxtState σε *κάθε* σκέλος των case και if-then-else statements, ώστε να μην υπάρχει περίπτωση το nxtState να "θυμάται" την παλαιά τιμή.
- **(γ)** Η τιμή που αναθέτουμε στο nxtState είναι συνάρτηση *μόνο* των εισόδων που εμφανίζονται στην παρένθεση του always, διότι μόνον αυτών η αλλαγή προκαλεί επανεκτέλεση του always block και άρα ανάθεση νέας τιμής στο nxtState.

Το κομάτι "**#0.5**" μετά το "<=" της **κάθε** ανάθεσης σημαίνει "η νέα τιμή να εφαρμοστεί στο nxtState με **καθυστέρηση 0.5 ns**" (σε σχέση με το πότε εκτελείται το always block, δηλ. από τη "στιγμή" που αλλάζουν τιμή τα σήματα που εμφανίζονται στο "@(...)"). Κάνουμε την απλοποιητική παραδοχή ότι η λογική που γεννά το nxtState έχει καθυστέρηση (μόνο) **0.5 ns** --αυτό αντιστοιχεί σε απλή λογική, δύο περίπου (μόνο) επιπέδων πυλών, με μικρό fan-in και fan-out. Μην ξεχνάτε να βάζετε πάντα αυτή την καθυστέρηση, διότι αλλοιώς το κύκλωμά σας θα μοιάζει γρηγορότερο απ' όσο θα είναι στην πραγματικότητα. Η ανάθεση γίνεται με "**<=**", και όχι απλώς με "=", γιά ένα λόγο που έχει να κάνει με την εσωτερική λειτουργία της Verilog, και ο οποίος εκτίθεται παρακάτω γιά την ενημέρωση των ενδιαφερομένων αλλά δεν χρειάζεται να σας απασχολήσει.

> [Η ανάθεση "`a = #0.5 b;`" προκαλεί "μπλοκάρισμα" του always block στο σημείο της καθυστέρησης γιά 0.5 ns. Στη χρονική αυτή περίοδο που το always block είναι "κολλημένο" στο σημείο εκείνο, το block αυτό **δεν** ελέγχει αν άλλαξε κάποιο από τα σήματα εισόδου του με σκοπό να επανυπολογίσει την τιμή εξόδου του --δηλαδή στο διάστημα της καθυστέρησης δεν δουλεύει το "`@(b or other_inputs)`". Την ανεπιθύμητη αυτή συμπεριφορά δεν έχει ο άλλος τύπος ανάθεσης, δηλαδή η ανάθεση "`a <= #0.5 b;`".]

<u>Άσκηση 11.4:</u> Περιγραφή των Εξόδων της FSM σε "Συνθέσιμη Verilog"

Τώρα που η FSM κινείται σωστά μεταξύ των καταστάσεων που πρέπει, το επόμενο βήμα είναι να περιγράψουμε τις εξόδους της FSM, δηλαδή τα σήματα ελέγχου γιά το datapath. Τα σήματα τύπου Moore είναι συναρτήσεις μόνο της currState. Τα σήματα τύπου Mealy είναι συναρτήσεις τόσο της

currState όσο και εισόδων του ελέγχου. Και στις δύο περιπτώσεις θα κάνουμε πάλι την απλοποιητική παραδοχή ότι η λογική που τα γεννά έχει καθυστέρηση (μόνο) **0.5 ns**, όπως και γιά το nxtState. Ακολουθήστε το εξής παράδειγμα γιά τα σήματα τύπου Moore:

```
reg irld;            // irld: "reg" for convenience of behavioral descr:
always @(currState) begin // a combinational function of currState, only
if (currState == i_fetch)
    irld <= #0.5  1'b1;    // load IR at end of i_fetch
else
    irld <= #0.5  1'b0;    // else do not load IR
end
```

Όπως και με το nxtState πιό πάνω, εδώ ορίζουμε ένα σήμα που προκύπτει από συνδυαστική λογική και μόνο. Παρ' όλα αυτά, επειδή το περιγράφουμε με το behavioral block "always", το ορίζουμε σαν reg. Όμως, ακολουθούμε όλες τις παραπάνω οδηγίες του nxtState, κι έτσι αυτό που περιγράφουμε είναι τελικά συνδυαστική λογική. Το κομάτι "**#0.5**" στην **κάθε** ανάθεση σημαίνει, και πάλι, καθυστέρηση 0.5 ns --μην ξεχνάτε να βάζετε πάντα αυτή την καθυστέρηση (και η ανάθεση είναι "<=" αντί "=" γιά τον ίδιο λόγο όπως παραπάνω). Για τα σήματα τύπου Mealy, η λίστα των εισόδων πρέπει να περιλαμβάνει και τα αντίστοιχα σήματα εισόδου κάθε φορά, π.χ. "`always @(currState or funct)`".

> [Σημείωση: κρατήστε υπ' όψη σας ότι σε περίπτωση που κάποιο από τα σήματα εισόδου έχει τιμή "x", τότε εκτέλειται το "else" ή το "default", κατά περίπτωση. Έτσι, στην αρχή του κύκλου decode_rr, καθώς αλλάζουν τιμή τα op και funct, τα σήματα που εξαρτώνται από αυτά (nxtState και σήματα τύπου Mealy) θα παίρνουν προσωρινά "παράξενες" τιμές, και αργότερα θα σταθεροποιούνται στις κανονικές τους τιμές. Το φαινόμενο αυτό, που μοντελοποιεί καθυστερήσεις που όντως υπάρχουν στο κύκλωμα, θα γίνει περισσότερο αισθητό αργότερα, όταν θα χρησιμοποιήσετε και το σήμα zero σαν είσοδο του ελέγχου.]

Άσκηση 11.5: Ολοκλήρωση Ελέγχου/Datapath και Αποσφαλμάτωση

Αφού τελειώσετε με τον έλεγχο, αντιγράψτε τα αρχεία της 10ης σειράς ασκήσεων στα "**`datapath11.v`**" και "**`test11.v`**". Αλλάξτε το top-level module ώστε να μην δίνετε τιμές με το χέρι στα σήματα ελέγχου, αλλά αυτά να τροφοδοτούνται από το **`control`** module που φτιάξατε (αρχείο "**`control11.v`**"). Αρχικοποιήστε:

i. Βάλτε στη μνήμη αρκετές εντολές για να ελέγξετε το κύκλωμά σας (π.χ. μία εντολή από κάθε είδος).
ii. Αρχικοποιήστε τον μετρητή προγράμματος, **pc**, με τον ίδιο τρόπο όπως και στην άσκηση 10.2. (κανονικά, αυτό θα το έκανε το σήμα reset που ένα πραγματικό κύκλωμα πρέπει πάντα να έχει).
iii. Αρχικοποιήστε τους καταχωρητές του κυκλώματος ελέγχου, δηλαδή τον **currState**, ώστε στον πρώτο κύκλο ρολογιού να εκτελεστεί ένα instruction fetch (κανονικά, και αυτό θα το έκανε το σήμα reset σ' ένα πραγματικό κύκλωμα).

Προσομοιώστε τη λειτουργία του επεξεργαστή και διορθώστε τα λάθη (debugging - αποσφαλμάτωση, ή "ξεζουζούνισμα"), τόσο στο κύκλωμα ελέγχου όσο και στο datapath που έχετε από την άσκηση 10.1. Θα πρέπει να ***τρέχουν σωστά*** τουλάχιστο μία από κάθε είδος εντολής, με μη τετριμένες τιμές των τελεστέων τους! Επίσης φτιάξτε ένα μικρό ατέρμονα βρόχο με τη βοήθεια της εντολής "j", και ελέγξτε ότι το προγραμματάκι εκτελείται σωστά.

Αφού τρέξουν όλα, υπολογίστε με τη βοήθεια του SignalScan πόση είναι η ελάχιστη δυνατή περίοδος του ρολογιού σας, και αλλάξτε το top-level ώστε να τρέχει ο επεξεργαστής σας με αυτό το **γρηγορότερο δυνατό ρολόϊ**. Πόσα MHz καταφέρατε να "πιάσετε"; (Μην κάνετε ζαβολιές: μην ξεχάσετε τις καθυστερήσεις "#0.5" στη γέννηση των σημάτων ελέγχου!)

Παραδώστε, όπως και στις προηγούμενες ασκήσεις, τον κώδικά σας "**control11.v**", τις αλλαγές που κάνατε στο datapath, "**datapath11.v**", το νέο test bench, "**test11.v**", τη νέα αρχικοποίηση μνήμης, "**memory11.hex**" ή "**memory11.bin**", και ένα χαρακτηριστικό στιγμιότυπο, "**signals11.jpg**", από το Signalscan της άσκησης 11.5 σε μορφή jpg, πακεταρισμένα στο αρχείο "**`ask11.tar`**", μέσω:

```
tar -cvf ask11.tar   control11.v datapath11.v test11.v memory11.hex signals11.jpg
~hy225/bin/submit 11
```

Up to the Home Page of CS-225
© copyright University of Crete, Greece.
Written by S. Lyberis, G. Sapountzis, and M. Katevenis; last updated: 28 Mar. 2006, by M. Katevenis.