Introduction to Embedded Systems
EHB326E
Lectures

Prof. Dr. Müştak E. Yalçın

Istanbul Technical University

mustak.yalcin@itu.edu.tr

Describing a system as a state machine($Step_1$)

- List all possible states
- Declare all variables
- For each state, list possible transitions, with conditions, to other states
- For each state and/or transition, list associated actions
- For each state, ensure exclusive and complete exiting transition conditions
  - No two exiting conditions can be true at same time (Otherwise nondeterministic state machine)
  - One condition must be true at any given time

Note:

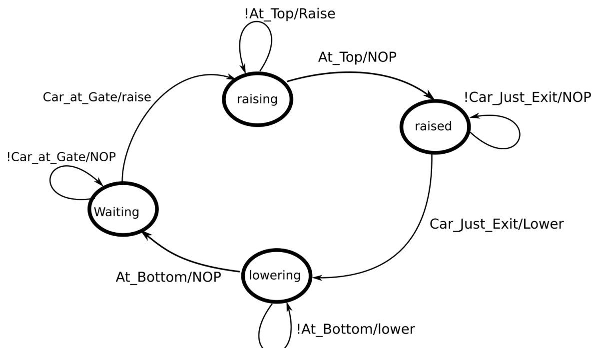
- Each transition is implicitly ANDed with rising clock edge !
- Any bit output is implicitly assigned a 0

For Moore machines the output generation is represented by assigning outputs with states. For Mealy machines conditional output generation is represented by assigning outputs to transitions! (EHB205 Int. Logic Design)

Simple parking gate controller:

The gate needs to do:

- If a car wants to come through, the gate needs to raise the arm until it is at the top position.
- Once the gate is at the top position, it has to stay there until the car has driven through the gate.
- After the car has driven through the gate needs to lower the arm until it reaches the bottom position.



Describing a system as a state machine

FSM is a powerful programming mechanism express a sequence of events. An event describes a physical activity. An event is a tuple (real-time stamp, value), where 'value' describes what has happened in the physical world (data value is irrelevant).

Example: The microcontroller reads two buttons and has two output pins that each drive a LED.

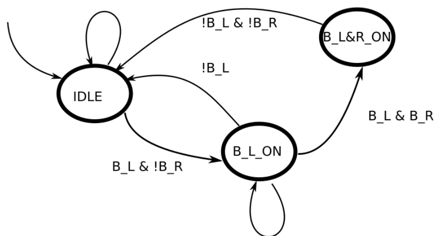
```
int main() {
    while (1) // Cyclic Executive !  {
              if (Button_1_Pressed())
                  turn_on_LED1
              else
                  turn_off_LED1
              if (Button_2_Pressed())
                  turn_on_LED2
              else
                  turn_off_LED2
}
}
```

A sequence of events is a series of physical events for which you care about the ORDER of events.

An event sequence is for example the following:
"ButtonLeft is pressed and held, followed by ButtonRight"

The challenge is to find a good method to capture this in C.

Using the standard cyclic executive, this is not so easy, because a cyclic executive looks only at one event at a time, with no memory of the past. Everything is instantaneous.



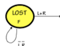
Lets build an Ant

- SENSORS: antennae L and R, each 1 if in contact with something.
- ACTUATORS: Forward Step F, ten-degree turns TL and TR (left, right).

GOAL: Make our ant smart enough to get out of a maze like:

STRATEGY: "Right antenna to the wall"

Lost in space



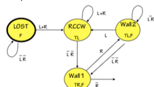
Action: Go forward until we hit something.

LOST F — L+R; $\bar{L}\bar{R}$

"lost" is the initial state

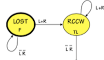
Then a little to the left

Action: Step and turn left a little, till not touching (again)



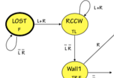
Bonk!

Action: Turn left (CCW) until we don't touch anymore



A little to the right…

Action: Step and turn right a little, look for wall

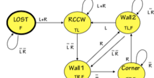


Equivalent State Reduction

Observation: $S_i \equiv S_j$ if

1. States have identical outputs; AND
2. Every input → equivalent states.

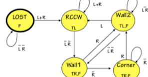
Reduction Strategy:
Find pairs of equivalent states, MERGE them.



Source: 6.004 Computation Structures (MIT OpenCourseWare)

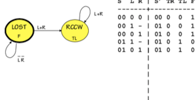
Dealing with corners

Action: Step and turn right until we hit perpendicular wall



6.004 - Spring 2009 2/24/09 L06 - FSMs 20

Building the Transition Table



S	L	R	S′	TR	TL	F
00	0	0	00	0	0	1
00	1	-	01	0	0	1
00	0	1	01	0	0	1
01	1	-	01	0	1	0
01	0	1	01	0	1	0

6.004 - Spring 2009 2/24/09 L06 - FSMs 23

Implementation Details

	S	L	R	S′	TR	TL	F
LOST	00	0	0	00	0	0	1
	00	1	-	01	0	0	1
	00	0	1	01	0	0	1
RCCW	01	1	-	01	0	1	0
	01	0	1	01	0	1	0
	01	0	0	10	0	1	0
WALL1	10	-	0	10	1	0	1
	10	-	1	11	1	0	1
WALL2	11	1	-	01	0	1	1
	11	0	0	10	0	1	1
	11	0	1	11	0	1	1

Complete Transition table

S1′

LR \ S_1S_0	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	1
10	0	0	0	1

$$S_1' = S_1\overline{S_0} + \overline{L}S_1 + \overline{L}\overline{R}S_0$$

S0′

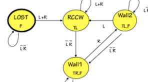
LR \ S_1S_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	1	1	1	0

$$S_0' = R + L\overline{S_1} + LS_0$$

6.004 - Spring 2009 2/24/09 L06 - FSMs 24

An Evolutionary Step

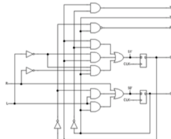
Merge equivalent states Wall1 and Corner into a single new, combined state.



Behaves exactly as previous (5-state) FSM, but requires half the ROM in its implementation!

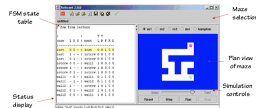
6.004 - Spring 2009 2/24/09 L06 - FSMs 22

Ant Schematic



6.004 - Spring 2009 2/24/09 L06 - FSMs 25

Roboant®



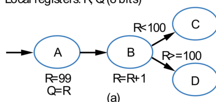
Featuring the new Mark-II ant: can add (M), erase (E), and sense (S) marks along its path.

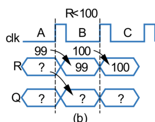
6.004 - Spring 2009 2/24/09 L06 - FSMs 26

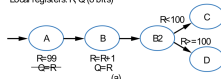
Source: 6.004 Computation Structures (MIT OpenCourseWare)

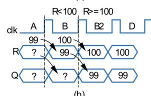
Creating a Datapath (Step$_2$)

- Make all data inputs and outputs to be datapath inputs and outputs.
- Implement the data storage by adding a register component into datapath for every declared register (RULE: always put a register before data output).
- Examine each state and transaction, adding and connecting new datapath components to implement new computation. Add multiplexors in front of shared components and define a control signal for them





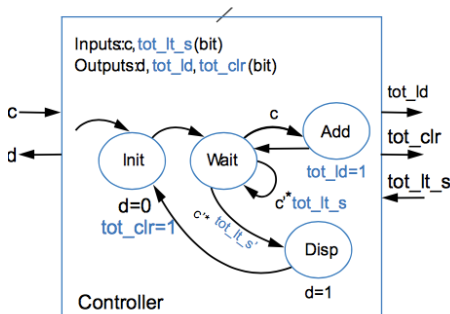




A state's actions configure the datapath and controller ! (the next clock edge will load the desired values)

Connecting the Datapath to a Controller and Design the Controller (Step$_3$ and Step$_4$)

- Just connect all control signals between controller and datapath
- Describe the control behaviour into a FSM replacing actions and conditions.



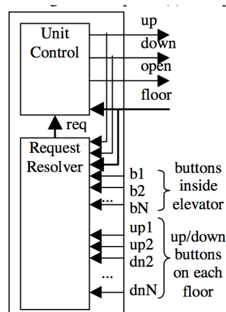
Study : Example 5.2 from Frank Vahid, Digital Design

State machine vs. sequential program model

- Different thought process used with each model
- State machine
  - Encourages designer to think of all possible states and transitions among states based on all possible input conditions
- Sequential program model:
  - Designed to transform data through series of instructions that may be iterated and conditionally executed
- State machine description excels in many cases
  - More natural means of computing in those cases

Example

Example :
"Move the elevator either up or down to reach the target floor. Once at the target floor, open the door for at least 10 seconds, and keep it open until the target floor changes. Ensure the door is never open while moving. Don't change directions unless there are no higher requests when moving up or no lower requests when moving down."



Design Technology

The system specification:

The designer describes the desired functionality in some language, often a natural language like English, but preferably an executable language !

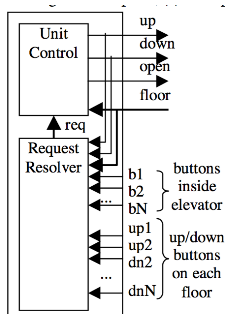
Designers must spend much time and effort simply understanding and describing the desired behaviour of a system, and some studies have found that most system bugs come from mistakes made describing the desired behaviour rather than from mistakes in implementing that behaviour.

D. Gajski, F. Vahid, S. Narayan and J. Gong, "Specification an Design of embedded systems," page 10 - 13.

Example

```
Inputs:  int floor; bit b1...bN; up1...upN-1; dn2...dnN;
Outputs:  bit up, down, open;
Global variables:  int req;

void main() {
Call concurrently:
UnitControl() and
RequestResolver()
          }
void UnitControl() {
up = down = 0; open = 1;
while (1) {
         while (req == floor);
         open = 0;
         if (req > floor) { up = 1;}
         else {down = 1;}
         while (req != floor);
          open = 1;
         delay(10);
     }
}
void RequestResolver() {
while (1)
...
req = ...
...
}
```



Read : Chapter 8: Embedded System Design, F. Vahdi & T Givargis

Concurrency & Parallelism

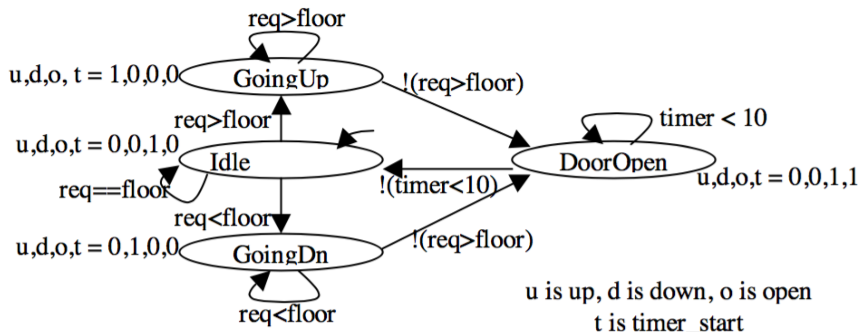
Concurrency is the ability to execute simultaneous operations because these operations are completely independent. Parallelism is the ability to execute simultaneous operations because the operations can run on different processors or circuit elements.

`UnitControl()` and `RequestResolver()` two concurrent process.

Hardware is always parallel. Software on the other hand can be sequential, concurrent, or parallel. Sequential and concurrent software requires a single processor, parallel software requires multiple processors

Modeling the example with FSMD

Functional Level :



General template: FSM to Seq. Prog. Language

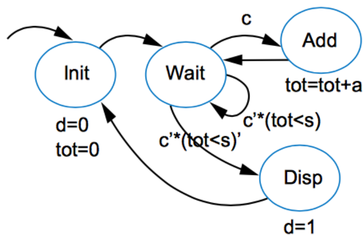
```
#define S0 0
#define S1 1
...
#define SN N
void StateMachine()
          int state = S0; // or whatever is the initial state.
          while (1) {
               switch (state) {
                    S0:
                    // Insert S0's actions here & Insert transitions Ti leaving S0:
                    if( T0's condition is true ) {state = T0's next state; /*actions*/ }
                    if( T1's condition is true ) {state = T1's next state; /*actions*/ }
                    ...
                    if( Tm's condition is true ) {state = Tm's next state; /*actions*/ }
                    break;
                    S1:
                    // Insert S1's actions here
                    // Insert transitions Ti leaving S1
                    break;
                    ...
                    SN:
                    // Insert SN's actions here
                    // Insert transitions Ti leaving SN
                    break;

          }
}
```

Sequential program model for the example

```
#define IDLE 0
#define GOINGUP 1
#define GOINGDN 2
#define DOOROPEN 3
void UnitControl() {
          int state = IDLE;
          while (1) {
               switch (state) {
               IDLE: up=0; down=0; open=1; timer_start=0;
                    if (req==floor) {state = IDLE;}
                    if (req > floor) {state = GOINGUP;}
                    if (req < floor) {state = GOINGDN;}
                    break;
               GOINGUP: up=1; down=0; open=0; timer_start=0;
                    if (req > floor) {state = GOINGUP;}
                    if (!(req>floor)) {state = DOOROPEN;}
                    break;
               GOINGDN: up=1; down=0; open=0; timer_start=0;
                    if (req < floor) {state = GOINGDN;}
                    if (!(req<floor)) {state = DOOROPEN;}
                    break;
               DOOROPEN: up=0; down=0; open=1; timer_start=1;
                    if (timer < 10) {state = DOOROPEN;}
                    if (!(timer<10)){state = IDLE;}
               break;
                    }
          }
```

Example 5.1 (Vahid, page 227) : Soda dispenser



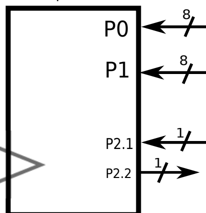

```
#define INIT 0
#define WAIT 1
#define ADD 2
#define DISP 3
void State_Machine_Soda_Disp() {
        int state = INIT;
        while(1){
            switch(state){
                INIT : d = 0; tot = 0;
                    state = WAIT;
                    break;
                WAIT :
                    if(c == 1){state = ADD; }
                    if((tot < s)&(!(c))){state = WAIT; }
                    if(!(tot < s)&(!(c))){state = DISP; }
                    break;
                ADD :
                    tot = tot + c;
                    state = WAIT;
                    break;
                DISP : d = 1;
                    state = INIT;
                    break;
            }
        }
}
```

Design of Soda Dispenser on general-purpose processor

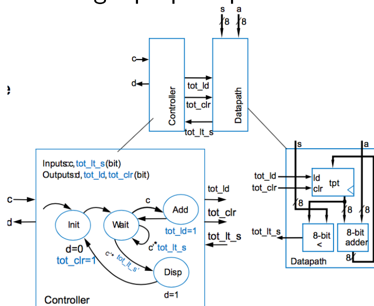
C Program

```
#define INIT 0
#define WAIT 1
#define ADD 2
#define DISP 3
void State_Machine_Soda_Disp() {
        int state = INIT;
        while(1){
            switch(state){
                INIT : d = 0; tot = 0;
                    state = WAIT;
                    break;
                WAIT :
                    if(c == 1){state = ADD; }
                    if((tot < s)&((c))){state = WAIT; }
                    if((tot < s)&((c))){state = DISP; }
                    break;
                ADD :
                    tot = tot + c;
                    state = WAIT;
                    break;
                DISP : d = 1;
                    state = INIT;
                    break;
            }
        }
}
```

Microprocessor



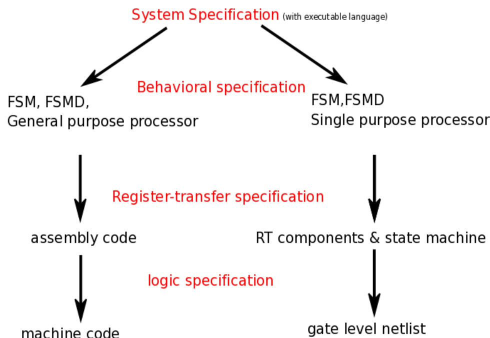
Design of Soda Dispenser on single-purpose processor



Embedded System Design

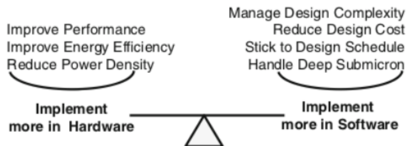
The designer refines Behavioural Specifications into register-transfer (RT) specifications

- by converting behavior on general-purpose processors to assembly code,
- by converting behavior on single-purpose processors to a connection of register-transfer components and state machines.



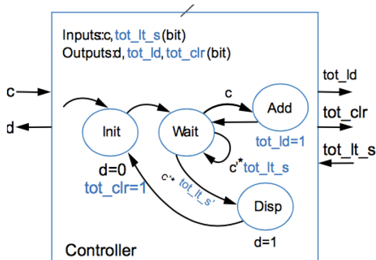
Hardware/Software codesign

Hardware/Software codesign is the design of cooperating hardware components and software components in a single design effort.



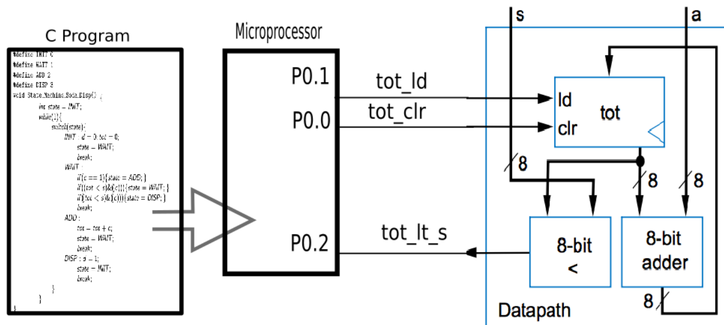
Hardware/Software codesign is the partitioning and design of an application in terms of fixed ('hardware') and flexible ('software') components.

Example 5.1 (Vahid, page 227) : Hardware/Software codesign of Soda Dispenser




```
#define INIT 0
#define WAIT 1
#define ADD 2
#define DISP 3
void State_Machine_Soda_Disp() {
        int state = INIT;
        while(1){
            switch(state){
                INIT : d = 0; tot_clr = 1;
                    state = WAIT;
                    break;
                WAIT :
                    if(c == 1){state = ADD; }
                    if((tot_lt_s)&((c))){state = WAIT; }
                    if((tot_lt_s)&((c))){state = DISP; }
                    break;
                ADD : tot_ld = 1
                    state = WAIT;
                    break;
                DISP : d = 1;
                    state = INIT;
                    break;
            }
        }
}
```

Hardware/Software codesign of Soda Dispenser



Example : Obtain Sequential Prog. of Example 5.2 from Frank Vahid, Digital Design