Automata and Grammars (BIE-AAG)

4. Program implementation of NFA and DFA, circuit impl.

Jan Holub

Department of Theoretical Computer Science

Faculty of Information Technology

Czech Technical University in Prague

Program implementation

2 basic approaches:

- Table driven – transition function δ stored in a variable (e.g. in a 2D array, linked list, . . .), active state is stored in a variable
- Hardcoded – transition function is stored as lines of code in the program body, state is represented by position in the program

+ special cases

DFA

Example:

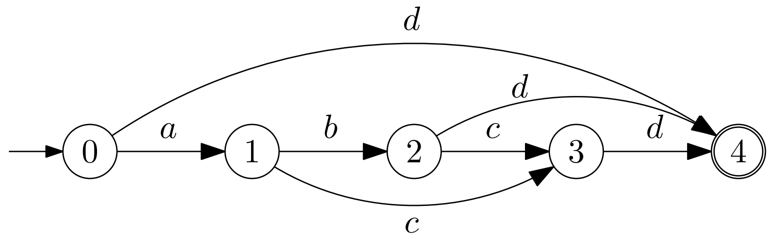


Table Driven

transition_table:

	a	b	c	d
0	1	-	-	4
1	-	2	3	-
2	-	-	3	4
3	-	-	-	4
4	-	-	-	-

```
int DFA_TD(){
  int state=0, symbol;

  while( (symbol = getchar()) != EOF ) {
    state = transition_table[state][symbol];
  }
  return is_final[state];
}
```

Hard Coded

```
int DFA_HC(){
  int symbol;

  state0: if ((symbol = getchar()) == EOF) return 0;
          switch (symbol){
            case 'a': goto state1;
            case 'd': goto state4;
            default: return(-1);
          };
  state1: if ((symbol = getchar()) == EOF) return 0;
  switch (symbol){
    case 'b': goto state2;
    case 'c': goto state3;
    default: return(-1);
  };
  state2: if ((symbol = getchar()) == EOF) return 0;
  switch (symbol){
    case 'c': goto state3;
    case 'd': goto state4;
    default: return(-1);
  };
  state3: if ((symbol = getchar()) == EOF) return 0;
  switch (symbol){
    case 'd': goto state4;
    default: return(-1);
  };
  state4: if ((symbol = getchar()) == EOF) return 1;
  return(-1);
}
```

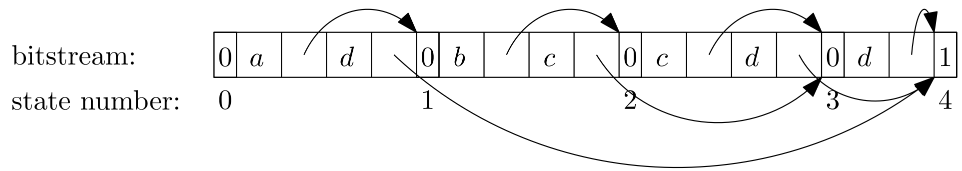

Comparison of the two basic approaches

	Table Driven	Hardcoded
time complexity	$\mathcal{O}(n)$	$\mathcal{O}(n)$
program size	$\mathcal{O}(1)$	$\mathcal{O}(\lvert Q\rvert * \lvert\Sigma\rvert)$
memory for the variables	$\mathcal{O}(\lvert Q\rvert * \lvert\Sigma\rvert)$	1

Bitstream implementation

Special case:

- Acyclic finite automaton, states ordered from left to right by the transitions.
- No returning back, suitable for caching.



Hardware implementation

What we need:

- encode the input alphabet,
- encode the automaton states, remember active state,
- implement the transition function using the encoded states and input symbols,
- recognize final states.

Input alphabet encoding

We encode the input alphabet in binary code

- 8 bits can be used for text (extended ASCII).
- Less bits can be used for smaller alphabets. Space is saved, but a conversion is necessary.

State representation

States are most usually encoded in binary code or Gray code.

- Active state is stored in data register.
- The transition function assigns a code of target state to every combination of source state and input symbol.
- Initialization – zeroing of the data register (if the start state is encoded as zero).

Input alphabet encoding

We encode the input alphabet in binary code

- 8 bits can be used for text (extended ASCII).
- Less bits can be used for smaller alphabets. Space is saved, but a conversion is necessary.

Example – $\Sigma = \{a, b, c, d\}$

Symbol	Code
a	00
b	01
c	10
d	11

For binary code representation $\lceil \log_2(|\Sigma|) \rceil$ bits suffice.

State representation

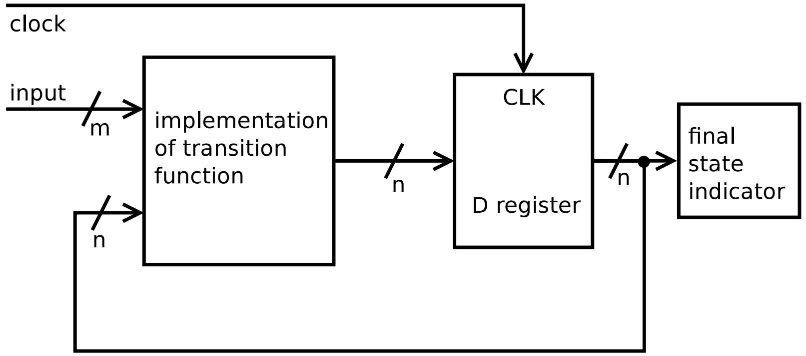
Alternatively, code 1 of N can be used.

- Every state is represented by one 1-bit flip-flop. If set to logical 1, state is active.
- Every transition is implemented by a function of source state and code of input symbol.
- Initialization – by setting the start state's register to log1, others to log0.

Circuits in both cases are synchronous. Transition to a new state is directed by clock signal.

Implementation – binary code

- Active state is stored in a data register.
- At every tact a new target state is stored in the register. The state is a function of the previous state and input symbol.



Transition function implementation

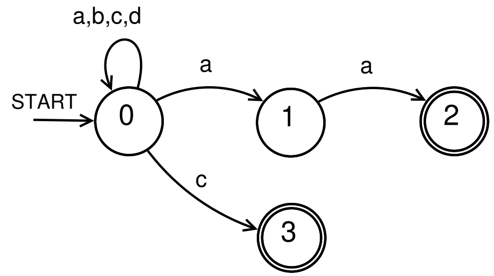
- By combinational logic.
  - Individual codes of the target state are determined by a combinational function of the bits of the source state and input symbol.
  - Final state is indicated by a combinational function of the bits of the active state.
- Using programmable memory.
  - Code of the target state is stored in a memory location whose address is a combination of the code of the source state and code of the input symbol.
  - Indication of the final state can be a part of the transition function. Value of this function is then invariant in respect to input (same for all inputs in the given state).

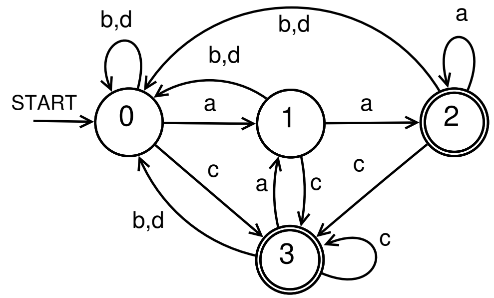
Example

We construct a finite automaton over an alphabet $\Sigma = \{a, b, c, d\}$ that accepts all words that end with "aa"or "c".

The automaton must be deterministic and total.

Example – automaton





Example – transition table

Q / X	a	b	c	d
→ 0	1	0	3	0
1	2	0	3	0
← 2	2	0	3	0
← 3	1	0	3	0

Q / X	00	01	10	11
$q_1 q_0$	$q_1' q_0'$	$q_1' q_0'$	$q_1' q_0'$	$q_1' q_0'$
→ 00	01	00	10	00
01	11	00	10	00
← 11	11	00	10	00
← 10	01	00	10	00

Gray code is more suitable for state encoding in this case. Input alphabet is encoded in binary. In our case we save a few bits, but input conversion is necessary.

Implementation with combinational logic

x1, x0 (columns); q0, q1 (rows)

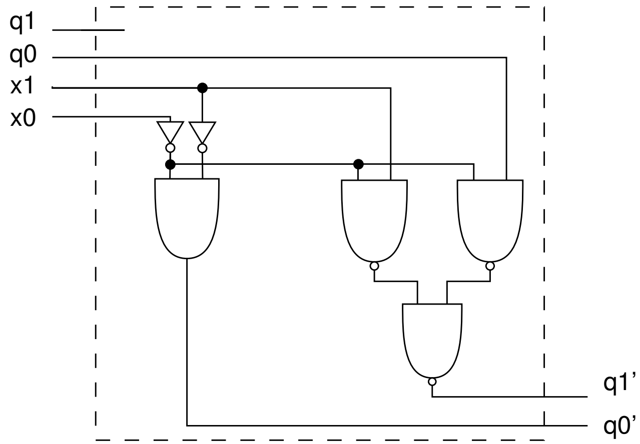
1	0	0	0
1	0	0	0
1	0	0	0
1	0	0	0

x1, x0 (columns); q0, q1 (rows)

0	0	0	1
1	0	0	1
1	0	0	1
0	0	0	1

$$q_0' = \overline{x_0} \cdot \overline{x_1} \qquad q_1' = \overline{x_0} \cdot x_1 + \overline{x_0} \cdot q_0$$

Implementation with combinational logic



$$q_0' = \overline{x_0} \cdot \overline{x_1} \qquad q_1' = \overline{x_0} \cdot x_1 + \overline{x_0} \cdot q_0$$

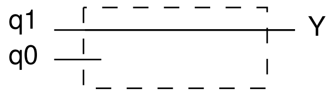
Final states

Final states are indicated by output function that depends only on the active state (Moore automaton).

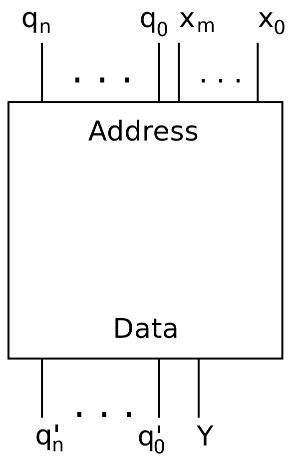
Q	Y
0	0
1	0
2	1
3	1

Q	Y
00	0
01	0
11	1
10	1

$$Y = q_1$$



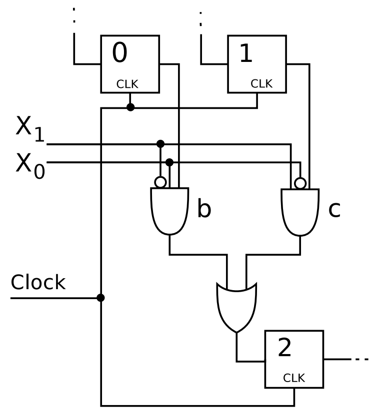
Using a programmable memory



Address	Content	Address	Content
$q_1q_0\ x_1x_0$	$q'_1q'_0Y$	$q_1q_0\ x_1x_0$	$q'_1q'_0Y$
0 0 0 0	0 1 0	1 0 0 0	0 1 1
0 0 0 1	0 0 0	1 0 0 1	0 0 1
0 0 1 0	1 0 0	1 0 1 0	1 0 1
0 0 1 1	0 0 0	1 0 1 1	0 0 1
0 1 0 0	1 1 0	1 1 0 0	0 1 1
0 1 0 1	0 0 0	1 1 0 1	0 0 1
0 1 1 0	1 0 0	1 1 1 0	1 0 1
0 1 1 1	0 0 0	1 1 1 1	0 0 1

We incorporate indication of final state Y into the transition function (value is invariant with the input).

Implementation fragment – code 1 of N



- Contrary to previous, we implement every transition separately.
- Apparently that is going to cost more components.
- Nondeterminism is simulated easily – more states are active at the same time.
- Automaton accepts if some final state is active – big OR gate on all final states

Implementation fragment – code 1 of N

Every state is represented by one 1-bit register. Every transition is implemented by a logical function (inputs b=01, c=10).