Computer Mathematics

Week 11

Sequencing and control

Finite State Machines

KUAS 京都先端科学大学
KYOTO UNIVERSITY of ADVANCED SCIENCE
Department of Mechanical and Electrical System Engineering

last week

sequential digital circuits

stateful logic

- level-triggered devices
- latches

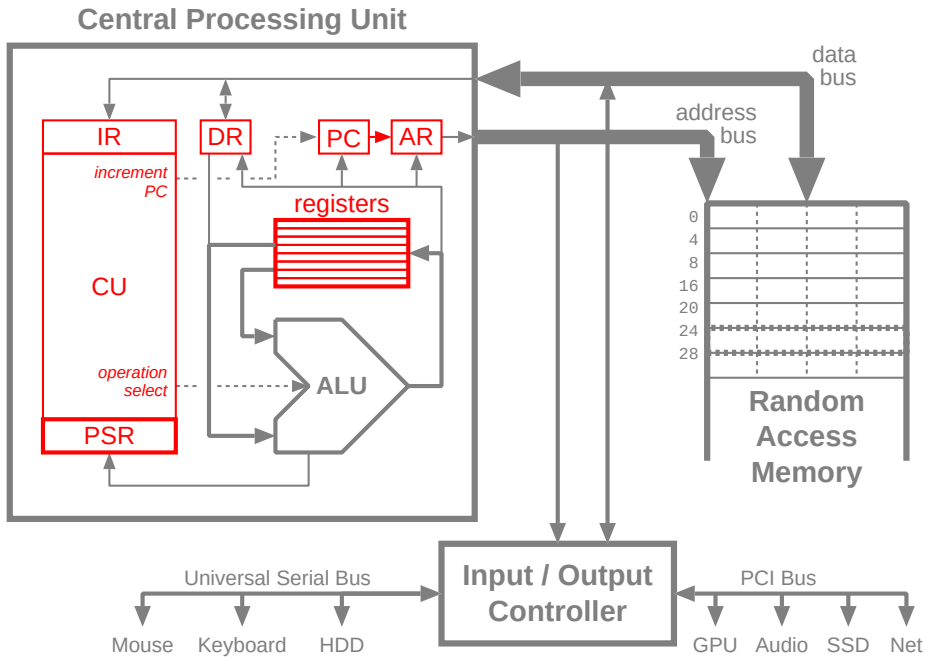
clocks

- edge-triggered devices

synchronous logic

- flip-flops
- registers and memory

CPU operation according to the clock cycle



this week

mathematics of control

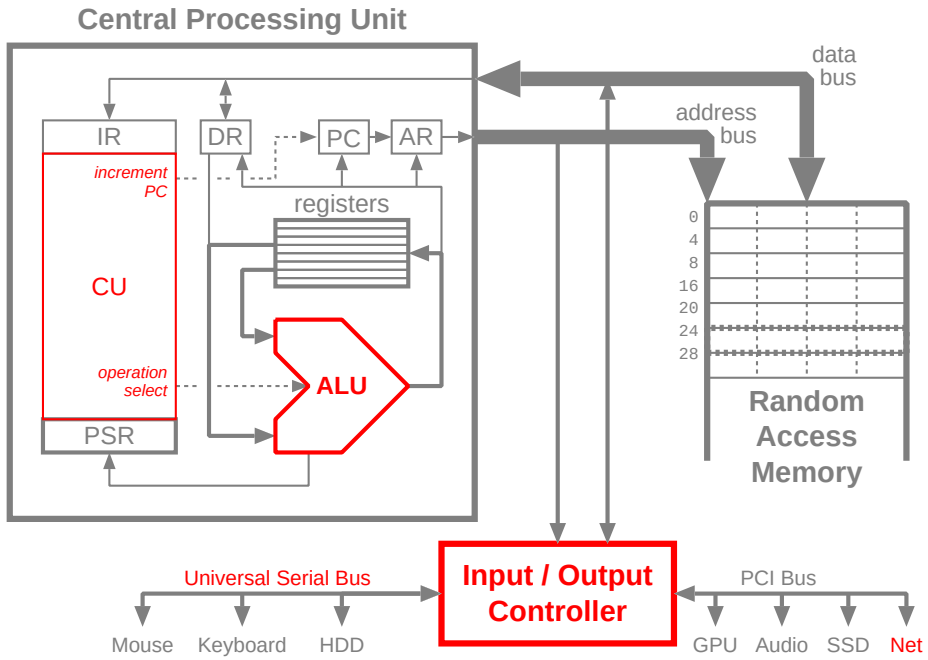
- models of stateful computation

finite state machines

- formal model
- representations

FSM applications

- pattern matching
- pattern generation
- sequencing



state machines

sequential logic: register (counter, etc.)

- has a stable *state* (the current output)
- *transitions* to the next state when the clock ticks
- next state is determined by current state and external inputs

sequential logic: the entire computer

- has a stable state (the contents of the registers and memory)
- transitions to the next state when the clock ticks
- next state is determined by current state and exernal inputs

one model describes very simple or very complex behaviour

- if we understand the model of simple behaviour, then
- we also understand complex behaviour (*any* computing device)

finite state machines

finite state machine (FSM) aka finite state automaton (FSA)

- simplest useful computing machine

theoretical value: abstract model of computation

- study the types and capabilities of different computing machines
- what resources are needed to compute particular types of problem?

practical value: physical process for computation

- building FSM in hardware makes CPUs, traffic lights, combination locks, . . .
- simulating FSM in software makes pattern matchers/generators, ‘AI’s, . . .
  - many software architectural solutions are described by state machines
  - e.g., communication protocols

finite state machines

FSM consists of a number of states

transition to next state is made in response to some *stimulus*

- a global clock ticks
- an external command (button, sensor) activates
- the next *symbol* (e.g., character from an input string) arrives

the next state depends only on

- the current state
- the input symbol (event type, sensor value, character value, etc.)

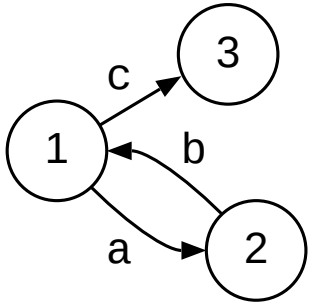
more sophisticated than it appears...

- inputting the same symbol doesn't always produce the same behaviour
- the current state affects the 'meaning' of the symbol

FSM representation: graph

representing a FSM as a *graph* makes it easy to visualise

- each state becomes a vertex (circle)
- each transition is an edge (arrow) between two states
- each edge is labelled with its associated input symbol (or event)



when in state 1

- input ‘a’ moves to state 2
- input ‘c’ moves to state 3

when in state 2

- input ‘b’ moves to state 1

the current state represents the entire history of the machine

a FSM can contain as many states as necessary

- can record arbitrarily long histories (sequences of inputs)
- just create a state for each possible position in the sequence

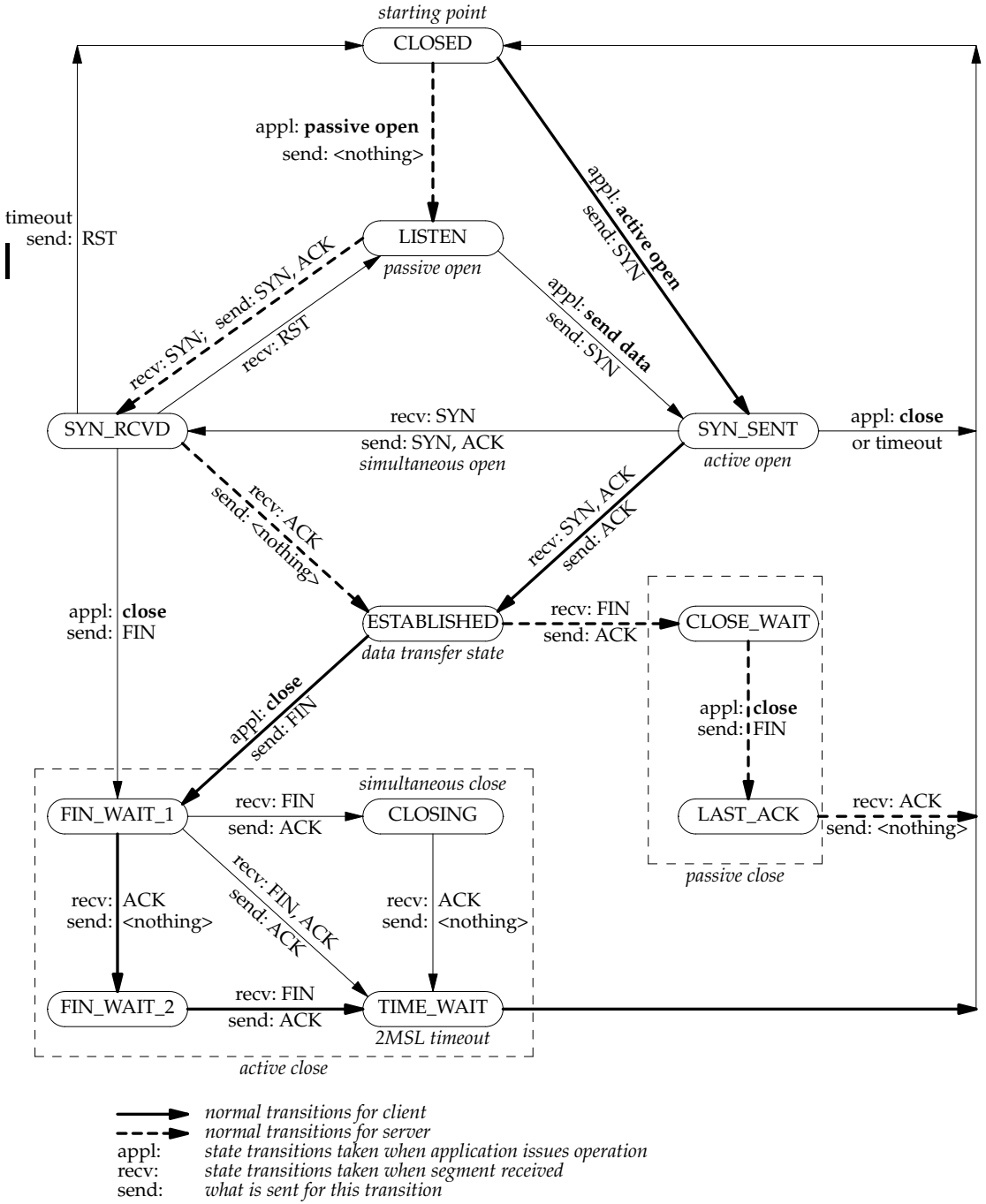
a practical application

TCP (Internet) communication protocol

- defined as FSM
- transitions depend on
  - application actions
  - network packets received

similar specifications for USB, etc.

FSMs can be compiled into code



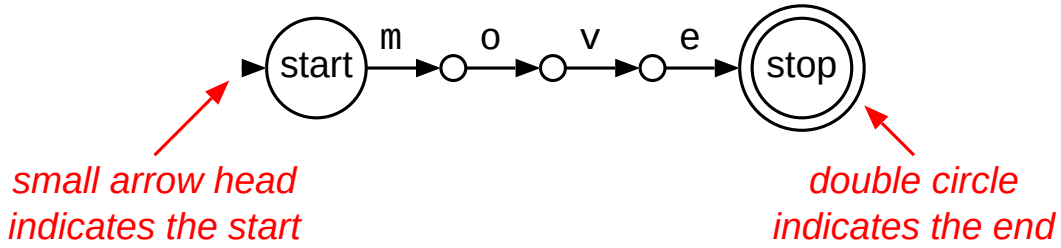
application to patterns and languages

closely connected with grammars and ‘languages’ that follow rules

define two of the FSM’s states as special

- a starting state, and
- a finishing state

what sequence of symbols will move from the start to the finish state?



any sequence that moves from start to finish is ‘accepted’ by the machine

- the machine is a *recogniser* for certain strings

equivalently, the FSM is a *generator* of acceptable sequence(s)

- output a string of symbols while moving from start to finish
- any generated string will also be accepted by the same FSM

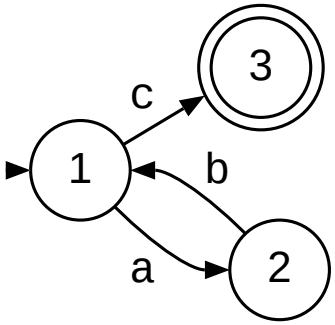
the complexity of the sequences is related to the complexity of the FSM

we now have a way to study symbol sequences and ask meaningful questions

patterns and languages

repeating patterns are made by *cycles* in the graph

let state 1 be the start, and state 3 be the finish



the machine will accept (generate) an infinite number of strings, starting with

- c
- abc
- ababc
- abababc
- . . .
- any number of ‘ab’ followed by a single ‘c’

examples

first page thereof

programming with loops

what does this program do?

```
import sys

while True:
    while True:
        c = sys.stdin.read(1)
        if c != ' ': break          # skip leading spaces
    while c != '' and c > ' ':
        sys.stdout.write(c)         # print non-spaces
        c = sys.stdin.read(1)
    while c != '' and c != '\n': # skip all trailing characters
        c = sys.stdin.read(1)
    if c == '': break               # stop at EOF
    sys.stdout.write(c)
```

⇒ loops.py

when you *know* what it is supposed to do, is it actually *correct*?

could you modify (or reuse) it easily for a slightly different pattern?

programming with FSMs

what about this program?

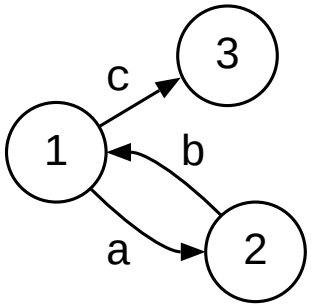
```
import sys

BEFORE = 1;  INSIDE = 2;  AFTER  = 3  # states

state = BEFORE

while True:
    c = sys.stdin.read(1)
    if '' == c: break  # end of file
    if  (state == BEFORE):
        if (c != ' '):
            state = INSIDE
            sys.stdout.write(c)
    elif (state == INSIDE):
        if  c == ' ' :
            state = AFTER
        else:
            if c == '\n': state = BEFORE
            sys.stdout.write(c)
    elif (state == AFTER):
        if c == '\n':
            state = BEFORE
            sys.stdout.write(c)
```

⇒ fsm.py

FSM representation: table

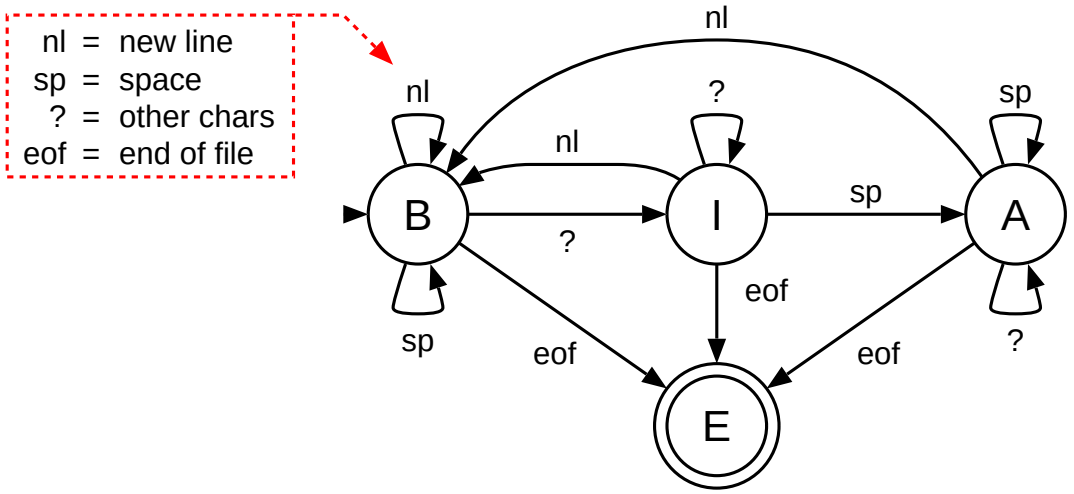
representing a FSM as a *table* makes it easy to simulate

- each row corresponds to a particular state
- each column corresponds to a particular input
- each entry in the table specifies a next state
- to make a transition
  - look up the entry at the current state (row) and input (column)
  - make the state stored there be the current state



when in this state ↓	a	b	c ← if you see this input
1	2		3 ← then go to this state
2		1	
3			

previous example's FSM



ignore spaces before the first word

print the first word until the next space

ignore all characters after the first word until a new line begins

an end-of-file character in any state stops the machine

	nl	sp	*other*	eof
BEFORE	BEFORE	BEFORE	INSIDE	END
INSIDE	BEFORE	AFTER	INSIDE	END
AFTER	BEFORE	AFTER	AFTER	END
END	*finish*			

programming with transition tables

```
import sys

NL = 0;  SP = 1;  OTHER = 2;  EOF = 3  # character types

char_to_type = { '' : EOF, '\n' : NL, ' ' : SP }

BEFORE = 0;  INSIDE = 1;  AFTER = 2;  END = 3  # FSM states

transitions = [
#              NL                  SP                  OTHER           EOF
    [ [ BEFORE, True ], [ BEFORE, False ], [ INSIDE, True  ], [ END, False ] ], # BEFORE
    [ [ BEFORE, True ], [ AFTER,  False ], [ INSIDE, True  ], [ END, False ] ], # INSIDE
    [ [ BEFORE, True ], [ AFTER,  False ], [ AFTER,  False ], [ END, False ] ], # AFTER
]

state = BEFORE

while state != END:
    c = sys.stdin.read(1)
    ctype = char_to_type[c] if c in char_to_type else OTHER
    nextState, printChar = transitions[state][ctype]
    if printChar: sys.stdout.write(c)
    state = nextState
```

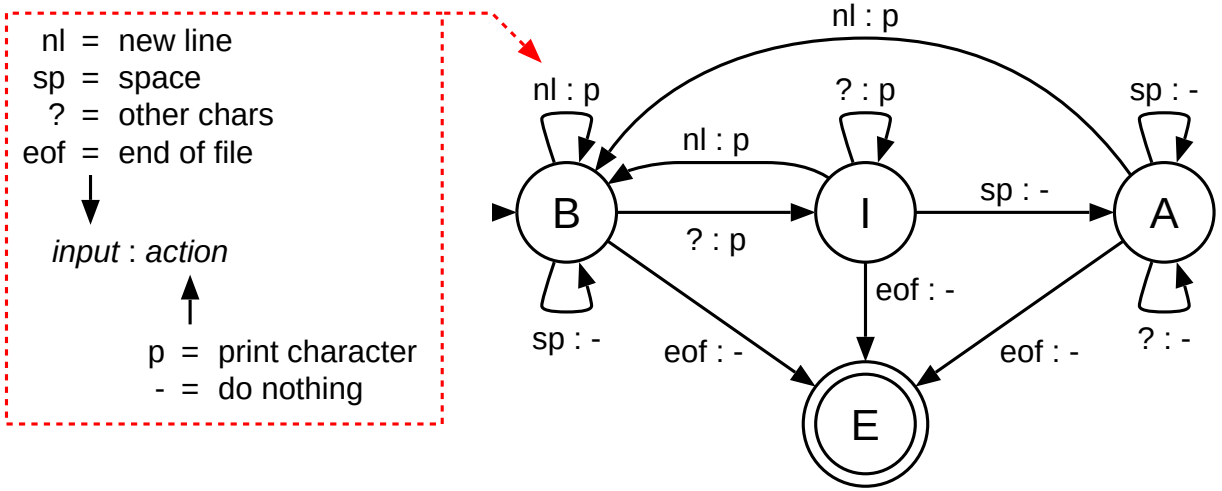
⇒ table.py

previous example's FSM with its actions

	nl	sp	*other*	eof
BEFORE	BEFORE (and print nl)	BEFORE	INSIDE (and print char)	END
INSIDE	BEFORE (and print nl)	AFTER	INSIDE (and print char)	END
AFTER	BEFORE (and print nl)	AFTER	AFTER	END
END	*finish*			

the actions (print or ignore character) should be included in the FSM graph

the transitions are annotated with '`:p`' (print input) or '`:-`' (don't print it)

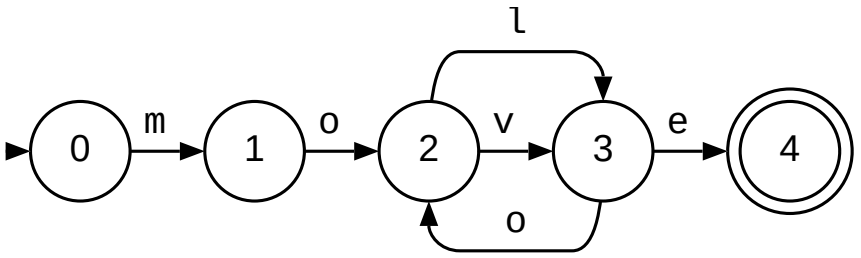


formal definition of a state machine

$M = \{\Sigma, S, s_0, F, \delta\}$

- an input alphabet (set of symbols) Σ
- a finite set of machine states S
- an initial state $s_0 \in S$
- a set of final (accepting) states F
- a transition function $\delta(s, \sigma) \mapsto s'$

M =



$$M = \left\{ \begin{array}{l} \Sigma = \{\mathtt{m}, \mathtt{o}, \mathtt{v}, \mathtt{l}, \mathtt{e}\} \\ S = \{0, 1, 2, 3, 4\} \\ s_0 = 0 \\ F = \{4\} \\ \delta = \begin{array}{lll} \delta(0, \mathtt{m}) \mapsto 1 & \delta(2, \mathtt{l}) \mapsto 3 & \delta(3, \mathtt{o}) \mapsto 2 \\ \delta(1, \mathtt{o}) \mapsto 2 & \delta(2, \mathtt{v}) \mapsto 3 & \delta(3, \mathtt{e}) \mapsto 4 \end{array} \end{array} \right\}$$

types of state machine

recogniser

- indicates whether the input is recognised
- transitions move from the start state to a single final state ($|F| = 1$)
- applications in parsing, pattern recognition, searching ⇒ wc.py

classifier

- final state indicates which class the input belong to ($|F| > 1$)

generator

- a recogniser whose transitions are labelled with single characters
- any path from start state to final state generates an acceptable sequence

transducer

- converts an input sequence into a sequence of outputs
- e.g., our word printing example
- applications in control (including the CPU), linguistics, etc.

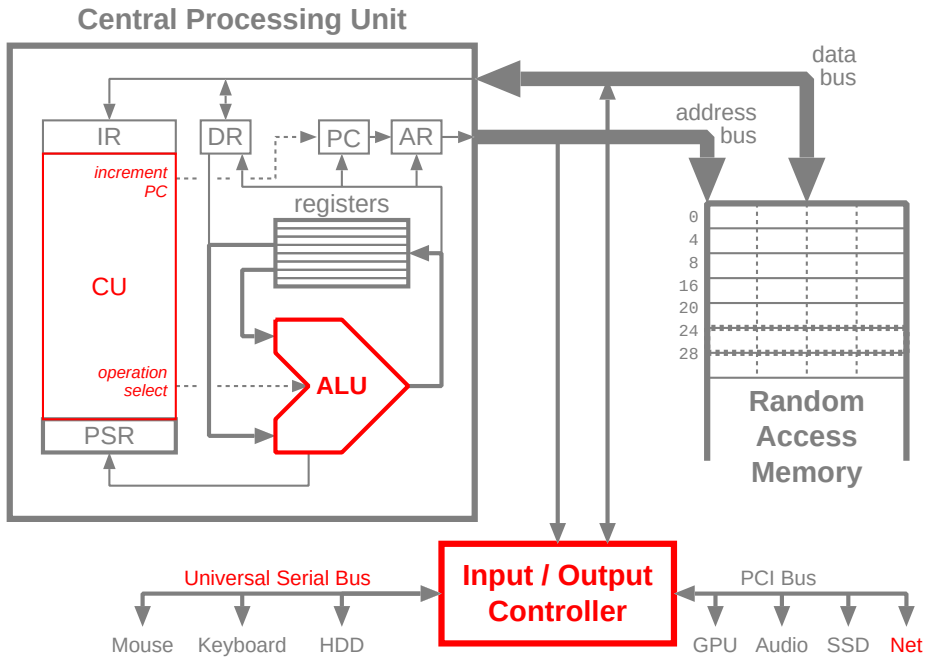
next week

Moore and Mealy machines

FSM applications

mathematical notation for FSMs

- regular expressions.



homework

run the example Python programs

- download from the course web site
- better still: type them in yourself

reinforce your understanding

- modify the table-driven FSM
- instead of printing (or not) the input, put action functions into the transition table

ask about anything you do not understand

- from any of the classes so far this semester (or the lecture notes)
- it will be too late for you to try to catch up later!
- I am always happy to explain things differently and practice examples with you

glossary

cycles — (in a graph) a path consisting of one or more edges that lead from a vertex back to the same vertex.

generate — running a state machine through a path from start to finish while recording the symbols attached to each transition, in order to produce a string of symbols that are acceptable by the machine.

graph — a mathematical representation of relationships between entities, in which vertexes represent objects or states and edges between then represent paths or relationships between them.

recognise — showing that an input sequence of symbols or events is accepted by a state machine, by following a path from start state to finish state, following the transitions that are indicated by successive symbols in the input sequence.

start state — the initial state of a state machine, before any transitions have been taken.

state — a stable condition of a state machine, representing the history of transitions taken since leaving the start state.

stimulus — (in a state machine) an input event, condition, value, etc., that causes a transition between states to be taken.

symbol — a character from a string, the name of an event, or some other identifiable value associated with a transition in a state machine.

transition — a path leading from one state to another, labelled with the input symbol that must be seen for the transitino to be taken.

transition table — a table that maps the current state and the current input symbol to the state that will be the next current state.