Computer Mathematics

Week 12

Sequencing and control

Regular Expressions

KUAS 京都先端科学大学

KYOTO UNIVERSITY of ADVANCED SCIENCE

Department of Mechanical and Electrical System Engineering

last week

mathematics of control

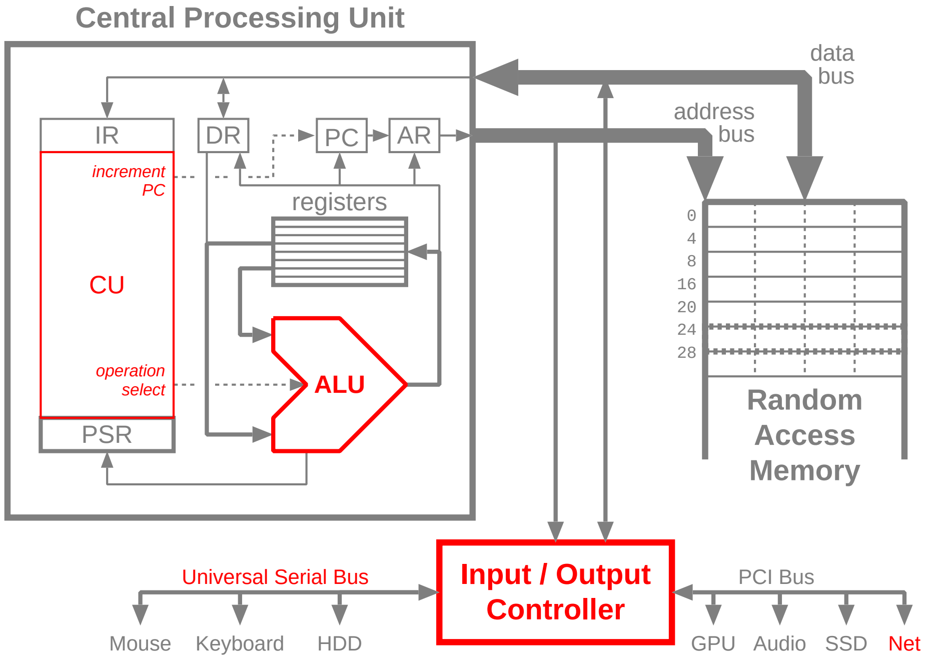
- models of stateful computation

finite state machines

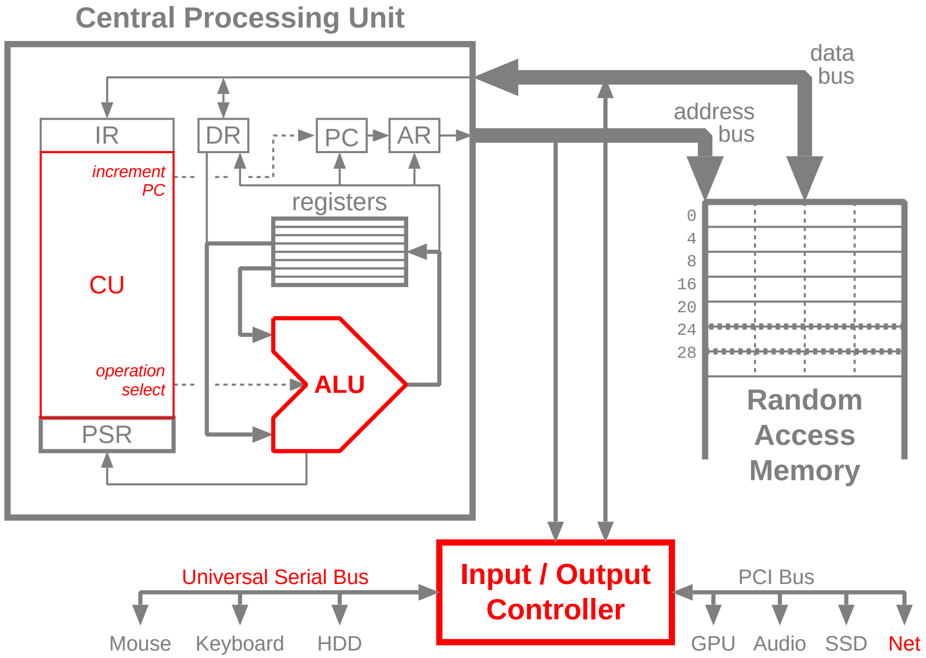
- formal model
- representations

FSM applications

- pattern matching
- pattern generation
- sequencing



this week



FSM applications

mathematical notation for FSMs

- regular expressions corresponding to FSMs

construction of FSMs

- from arbitrary regular expressions

FSM applications

hardware operations that take more than one cycle to complete

memory load/store

- real memory is slower than the CPU

$\Rightarrow$ use a counter to introduce idle cycles until data ready

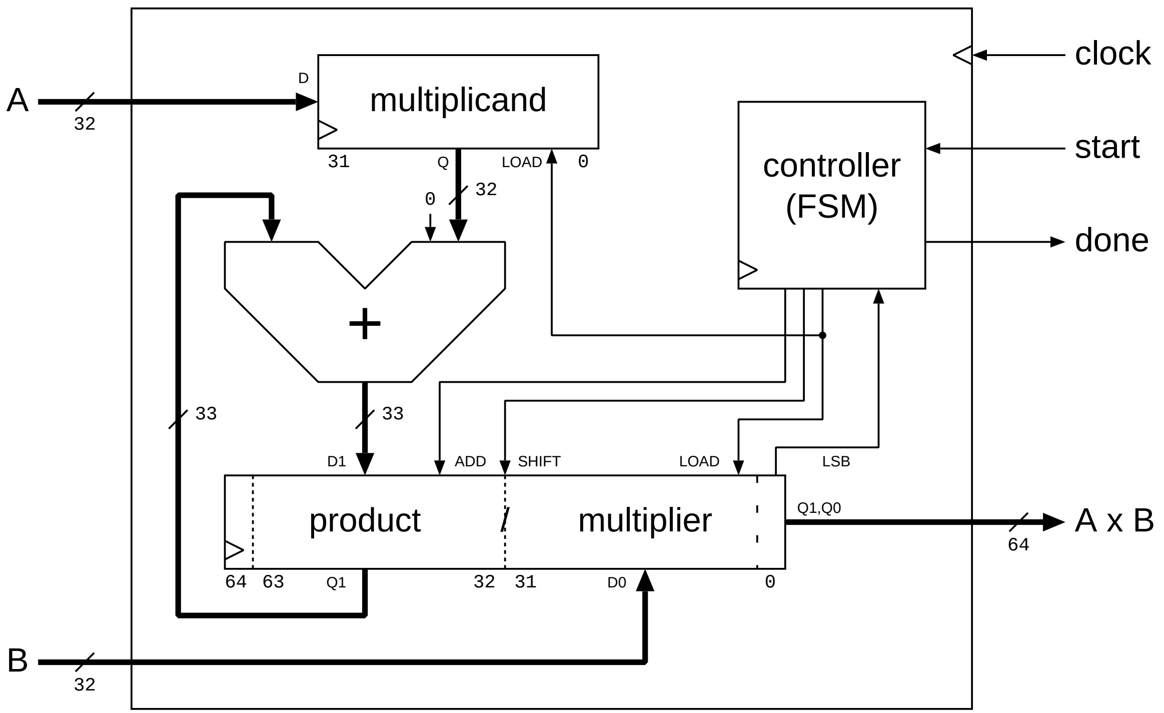
iterative ALU operations

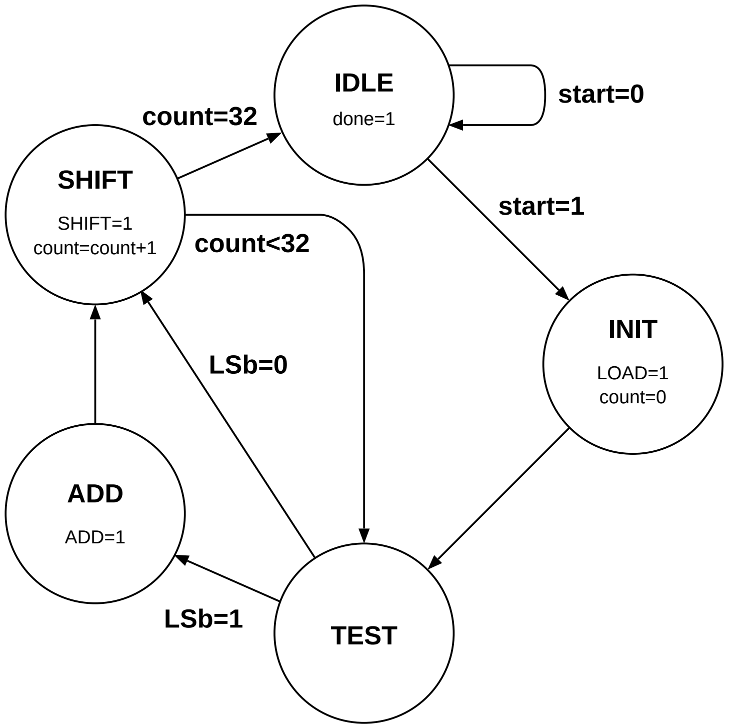
- multiplication $\Rightarrow$ (shift, multiply, add partial product) $\times N$ bits
- division $\Rightarrow$ (shift, subtract, keep/reject partial remainder) $\times N$ bits

background activities

- serial communications $\Rightarrow$ counter and shift register running autonomously
- transfer of data to/from device without CPU involvement
  $\Rightarrow$ device controller performs RAM read/write cycles

FSM example — hardware multiplication



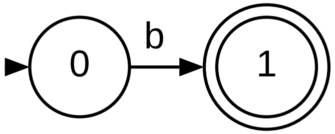


A, B	32-bit inputs
A×B	64-bit product
clock	global clock
start	begin multiply
done	multiply finished

LOAD	load multiplicand, load multiplier, clear product to 0
ADD	load product
SHIFT	shift product, multiplier right
LSb	least significant bit of multiplier

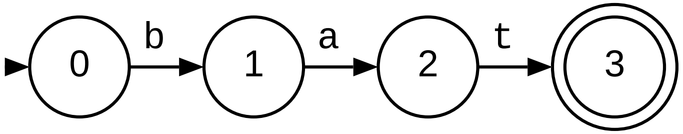
a mathematical notation for sequences

single transitions correspond to a single symbol



⇒ `b`

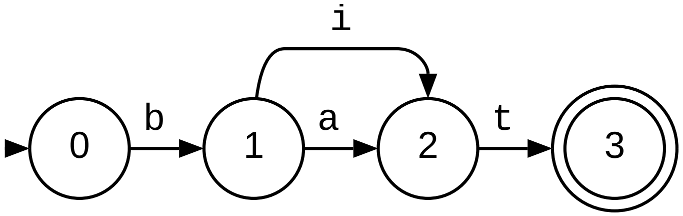
a *sequence* of transitions corresponds to a linear sequence of symbols



⇒ `bat`

alternative paths through the FSM produce alternative sequences of symbols

- written with a ‘|’ character between the alternatives

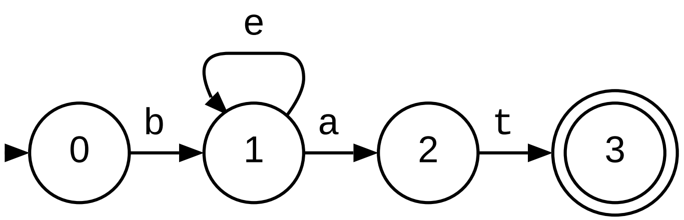


⇒ `bat|bit`
`b(a|i)t`

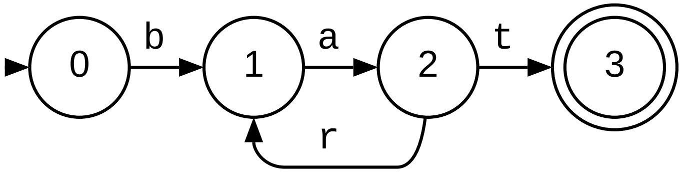
a mathematical notation for sequences

cycles in the path produce repeated sub-sequences of symbols

- a '$*$' *after* an item indicates that it repeats zero or more times



⇒ `b(e*)at`
`be*at`



⇒ `ba(ra)*t`

regular expressions

REs are sequences of symbols combined using concatenation, | , and $*$

parentheses can be used to group items

- limiting, or extending, the ‘reach’ of an operator within an expression

RE operator precedence:

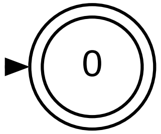
lowest	|	alternation separates entire sequences
	concatenation	creates sequences of single items
highest	$*$	operates on the single item immediately before it
	(...)	creates a single item from the RE ‘ ... ’

`f(oo)*|ba*(rs|z*es)` ⇒

f	foo	foooo	foooooo	foooo ... oooo
brs	bars	baars	baaars	baa ... aars
bzes	bazes	baazes	baaazes	baa ... aazes
bzzes	bazzes	baazzes	baaazzes	baa ... aazzes
bzzzes	bazzzes	baazzzes	baaazzzes	baa ... aazzzes
bzz ... zzes	bazz ... zzes	baazz ... zzes	baaazz ... zzes	baa ... aazz ... zzes

empty sequences

the FSM with no transitions generates (or recognises) the *empty string*

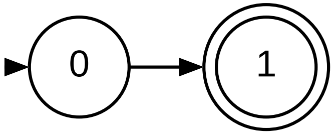


$\Rightarrow \quad \epsilon$

- written as ϵ, the Greek letter *epsilon* (ϵ for ‘ϵmpty’), or literally as an empty string
- ϵ is the string containing no symbols at all

FSM transitions corresponding to the empty string are called ϵ-*transitions*

- labelled with ϵ, or with no label at all



$\Rightarrow \quad \epsilon$

ϵ-transitions

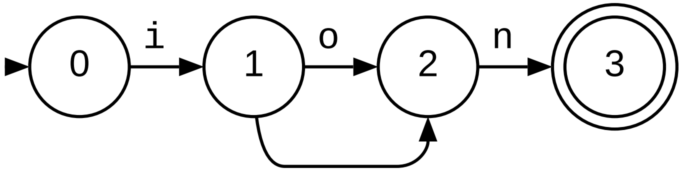
- generate no symbols at all
- can be followed immediately ‘for free’ when recognising a sequence
  - no input is necessary

empty sequences

ϵ-transitions (empty strings) are surprisingly useful

forward-skipping ϵ-transitions produce optional ‘zero or one’ sequences of symbols

- ‘?’ *after* an item indicates that it is optional



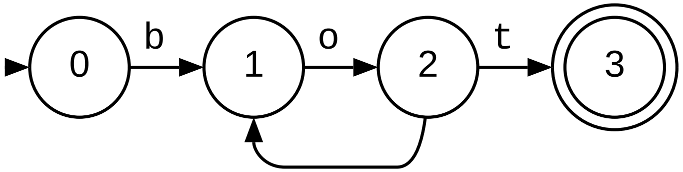
in|ion

⇒ i(o|ϵ)n

io?n

backward-skipping ϵ-transitions produce ‘one or more’ repetitions of a sequence

- denoted by a ‘+’ character after the item that repeats



bot|boot|

⇒ boo*t

bo+t

the ? and + notations are not *fundamental*, and are used for convenience only

- $e?$ can always be rewritten as $e|\epsilon$
- $e+$ can always be rewritten as $ee*$

creating FSMs from REs

REs are a very compact way to represent patterns of symbols

- e.g., patterns of characters within text

FSMs are a very efficient mechanism for recognising patterns of symbols

- read symbol, look up in transition table, move to next state, repeat

let’s turn an arbitrary RE into its equivalent FSM

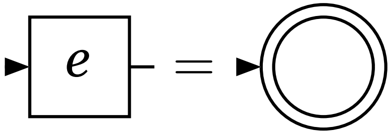
- so that we can search for arbitrary patterns of symbols very efficiently

creating FSMs from REs

let "$\blacktriangleright\boxed{e}\vdash$" represent the FSM corresponding to the regular expression e, then…

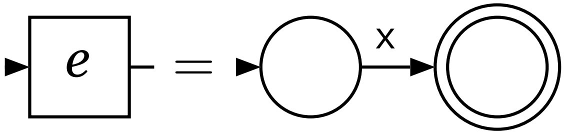
empty strings are just the empty FSM

$e = \epsilon$



single symbols are a single labelled transition

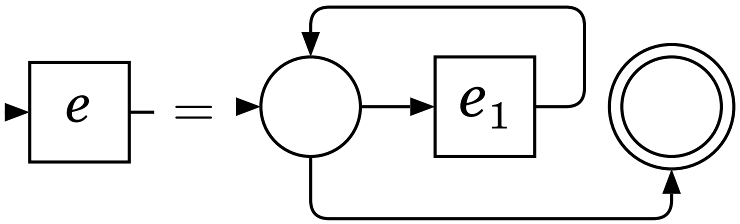
$e = \mathrm{x}$



repetition of an expression

- add two new states and
- two ϵ-transitions to the FSM

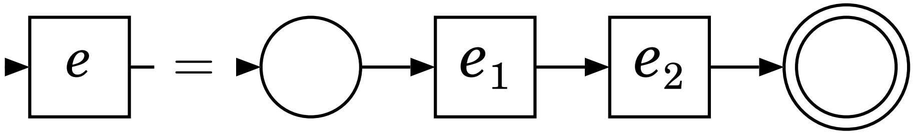
$e = e_1*$



concatenation of expressions

- place their FSMs in series

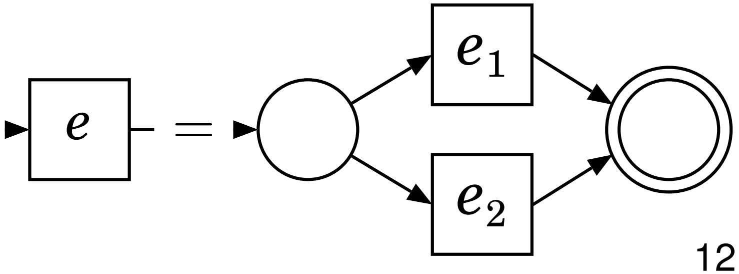
$e = e_1\, e_2$



alternation between expressions

- place their FSMs in parallel

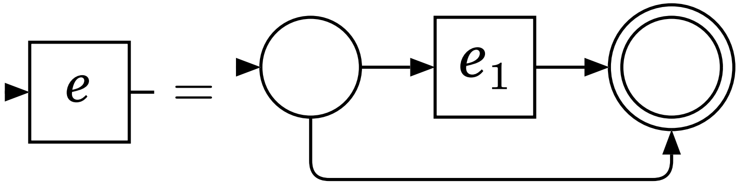
$e = e_1 \mid e_2$



creating FSMs from REs

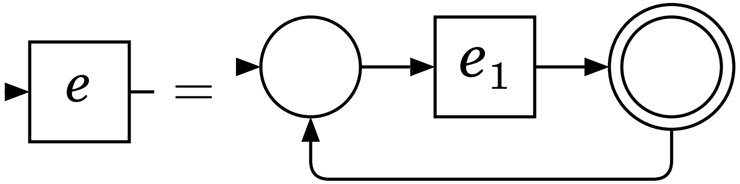
the two ‘convenience’ operators, $?$ and $+$, are similarly easy

optional ‘zero or one’ sequences $e = e_1?$



- ϵ short-circuit from their start to final state

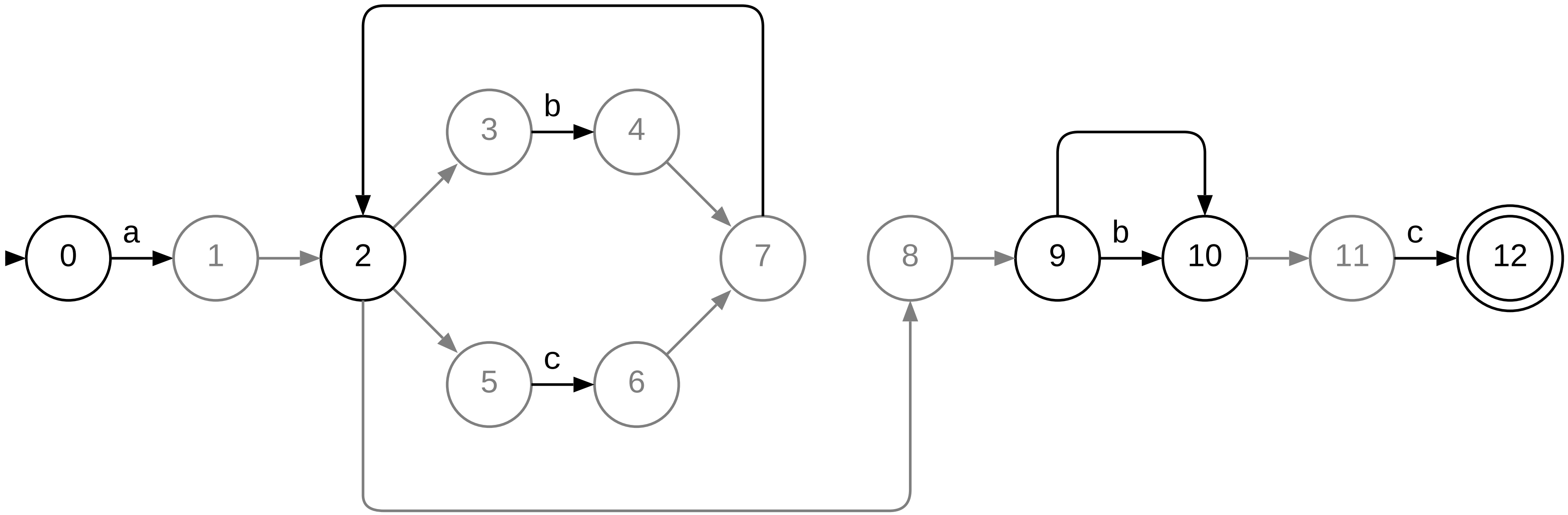
repeating ‘one or more’ sequences $e = e_1+$



- ϵ return from the final state back to the start

example

a(b|c)*b?c



(states and transitions in grey are artefacts of the construction algorithm, and are redundant)

this FSM raises several questions

- how do we choose which of the ϵ-transitions to follow from state 2?
- in state 2, how do we know if a b should lead us to state 4 or to state 10?
- how do we get rid of all those redundant states and transitions?

how do we ‘execute’ this FSM at all? ⇒ next week!

practical regular expressions

lots of additional features (too many)

wildcard character

- '`.`' matches any character
  - e.g., '`x...y`' matches `x` followed four characters later by `y`

anchoring

- '`^`' matches the beginning of a line
- '`$`' matches the end of a line
  - e.g., '`^hello$`' matches lines containing only `hello`

character classes

- '`[abc]`' matches `a`, `b`, or `c`
- '`[^abc]`' matches anything *except* `a`, `b`, or `c`
- '`[a-z]`' matches any lower-case letter, '`[0-9]`' matches any decimal digit
- '`[a-zA-Z_][a-zA-Z_0-9]`' matches an identifier

practical regular expressions

on the command line (Terminal.app):

- the program `egrep` finds lines in files that match a regular expression
- type ‘`man egrep`’ to find out how it works
- type ‘`man re_format`’ to read about the RE extensions it supports

in Python:

```
import re
rex = re.compile("[0-9]")
print rex.match("nope")
print rex.match("42")
```

in other languages

- JavaScript: `myString.search(`*/regular-expression/*`)`
- PERL: */regular-expression/* =~ `myString`
- `awk:` */regular-expression/* { *do-something* }
- and many more...

next week

non-deterministic machines

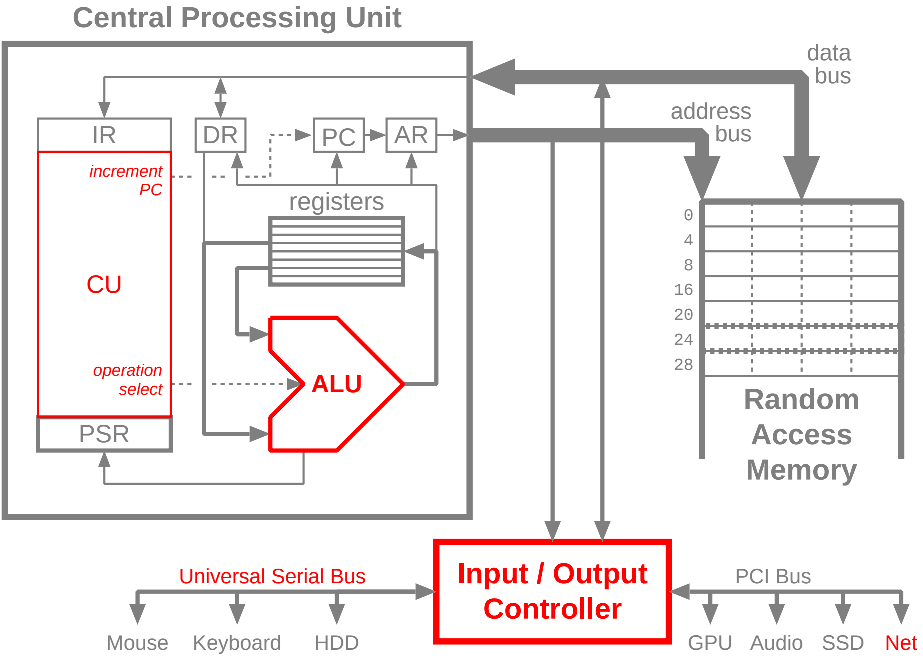
- how to simulate them

eliminating non-determinism

- make an equivalent deterministic machine
- using a clue from the simulation
  - and some real mathematics

deterministic machines

- advantages and disadvantages
- relative performance



homework

reinforce your understanding

- practice using regular expressions in Python

complete the multiplier example on page 5

- assume you have a 6-bit binary counter with reset
- draw the truth table for each of the 5 states
  - next state based on current state, inputs, counter, etc.
  - outputs based on current state, inputs, etc.
- write a Python program to simulate the multiplier

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

alternative — a choice between two or more possibilities (paths through a graph, sequences of symbols, etc.). In a FSM, alternatives appear as two or more distinct parallel paths between two states.

cycle — a path through a graph that arrives at a given state more than once.

empty string — a string that contains no symbols. Generated (and recognised) by an epsilon transition in a FSM.

epsilon — the Greek letter ϵ, representing something very small or non-existent.

epsilon transition — a transition that generates (and recognises) the empty string. The transition can be followed without producing or consuming any symbols at all.

fundamental — something that is essential.

sequence — a linear series of symbols, events, etc. In a FSM, a sequence appears as a series of states connected linearly.