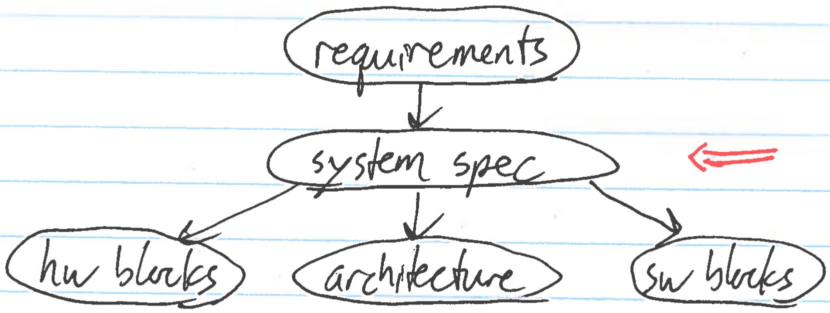
Models of Computation

- recall:



- the high-level system specification:
  - needs to express parallelism
  - is used for:
    1. ~~simulatable~~ simulation (if its executable)
    2. formal verification: proving properties mathematically
    3. partitioning & scheduling - map blocks to processing elements, determining temporal schedule
    4. code generation - automatically generate hw & sw (e.g. HLS)

- what about C/C++?

Problem #1: hard to express parallelism correctly
- threading model is fundamentally non-deterministic
- any interleaving of operations is possible by default
- race conditions occur by default and programmer must restrict parallelism to ensure correctness (mutex, semaphores, cond. vars, ...)
- however ↑ synchronization ⇒ ↓ parallelism ⇒ ↓ performance
- bugs are hard to find (e.g. Mars Pathfinder priority inversion)
- it would be safer to have a deterministic model ~~a~~ where the programmer specifies which parallelism to allow: trades safety for performance

Problem #2: there is no explicit concept of time
- must use (system) libraries for timing
- compiler can't check timing correctness
e.g. pthread library does have timed waits but it's not part of the language standard

High-Level Specification Language Requirements:

(1) Hierarchical
- we're not good at managing or reasoning systems of >5 objects
- types of hierarchy: behavioural (states, processes)
  structural (processor, PCB, node)

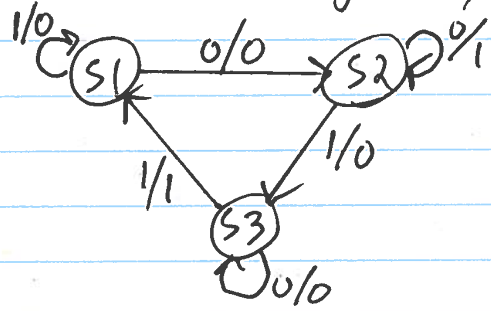
(2) Compositional Behaviour
- we can derive system properties/behaviours from those of the sub-systems

(3) Have Intrinsic Representation of Time
- support delays, timeouts, deadlines

(4) Efficient Implementation

Model Classification:
- control-oriented: e.g. state machines (good for reactive systems)
- data-oriented: e.g. dataflow (good for signal processing)

State Machines e.g. Mealy machines



state diagram

input	curr state	next state	output
0	S1	S2	0
1	S1	S1	0
0	S2	S2	1
1	S2	S3	0
0	S3	S3	0
1	S3	S1	1

state transition table

- can produce the output stream given initial state and input stream

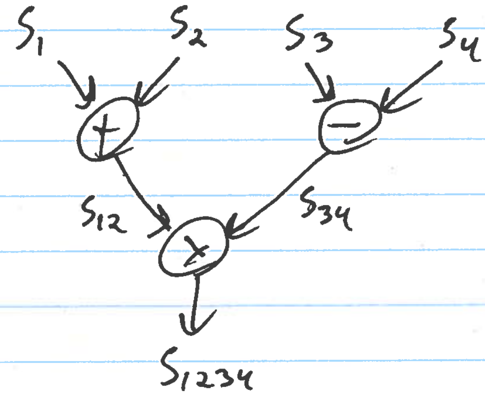
e.g.

state	S1	S1	S2	S2	S3	S1
input	1	0	0	1	1	...
output	0	0	1	0	1	

- problems: no parallelism, state explosion
- solution: hierarchical composition of state machines (e.g. Statecharts)

Simple Dataflow Graph

- nodes: data operations
- edges: data dependencies
- a stream is associated with each edge (FIFO)
- timing is based on the arrival of input tokens
- Node Firing Rule: execution consumes one ~~symb~~ token on each incoming edge and produces one ~~toke~~ token on each outgoing edge



S_1	2	4	-2	...
S_2	1	-3	4	...
S_3	0	5	8	...
S_4	3	9	2	...
S_{12}	3	1	2	...
S_{34}	-3	-4	6	...
S_{1234}	-9	-4	12	...

Limitations

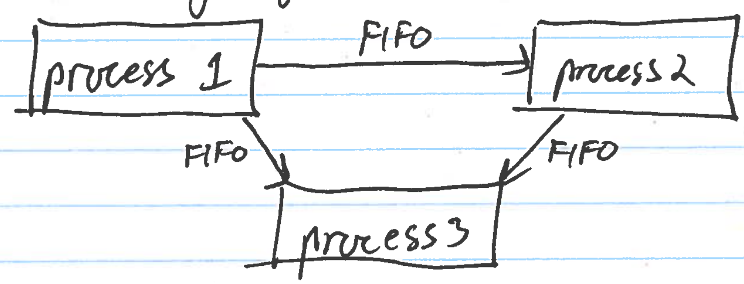
- no conditional execution
- no loops

Solution

- more expresive dataflow model (e.g. Kahn Process Networks)

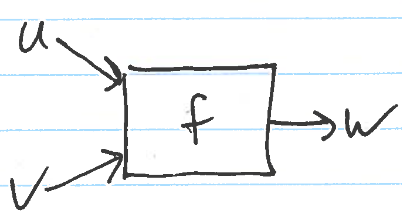
Kahn Process Networks (KPN)

- dataflow language

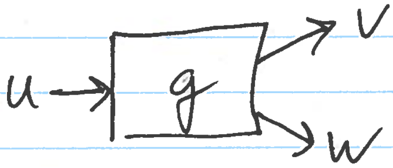


- a KPN is a set of processes that communicate over unidirectional communication channels
- channels: 1 reader, 1 writer, infinite FIFO of tokens
- tokens can be any data structure e.g. image
- processes execute concurrently
- each process can be described by imperative code (e.g. C)
- the language is augmented with a blocking wait() and a non-blocking send

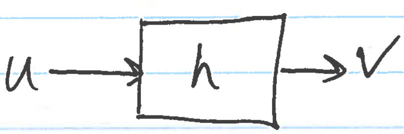
- example:




```
process f(in int u, in int v, out int w){
    bool b = true;
    for(;;) {
        int i = b ? wait(u) : wait(v);
        printf("%d ", i);
        send(i, w);
        b = !b;
    }
}
```

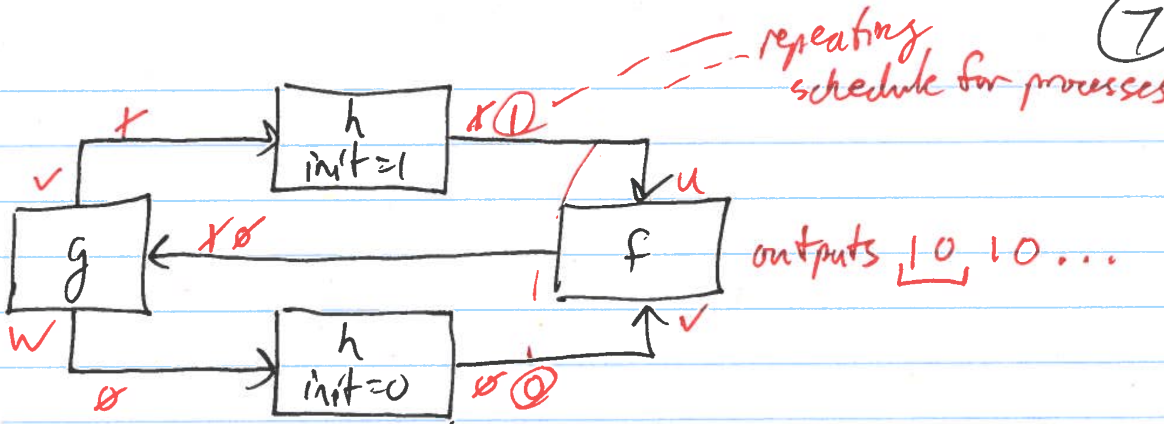



```
process g(in int u, out int v, out int w) {
    bool b = true;
    for(;;) {
        int i = wait(u);
        if(b) send(i, v);
        else send(i, w);
        b != b;
    }
}
```

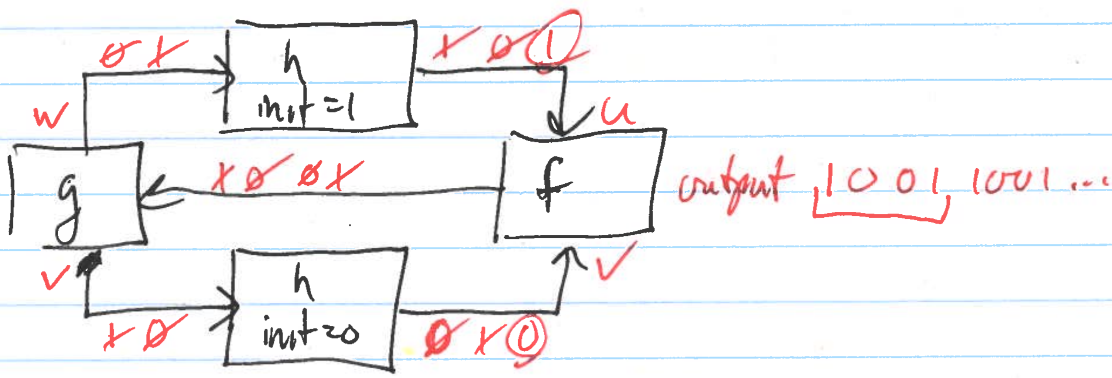


initialization process: sends initial value, then passes through values

```
process h(in int u, out int v, int init) {
    int i = init;
    send(i, v);
    for(;;) {
        i = wait(u);
        send(i, v);
    }
}
```



Flipping u/v and v/w gives outputs 01, 01...



KPN are deterministic
- the history of a channel is the sequence of tokens both written and read
- the histories of all channel depend only on the history of the input channels

Implication
- behaviour of a KPN is independent of timing (e.g. execution order, communication time, process execution time)
- no race conditions

- the process code/execution must be deterministic - its behaviour only depends on the sequence of input tokens (no shared data)

- questions of termination and boundedness are undecidable

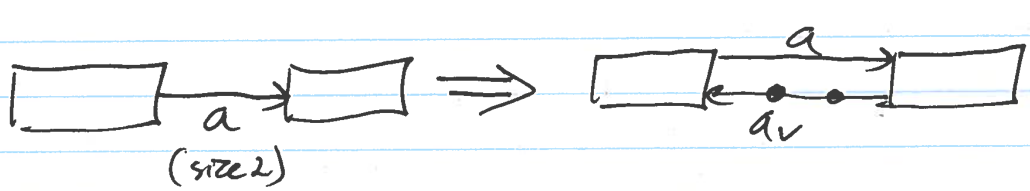
all processes are blocked on wait()

finite length queues

- if a KPN has a bounded implementation, then it can be transformed into a strictly bounded network without losing its determinism

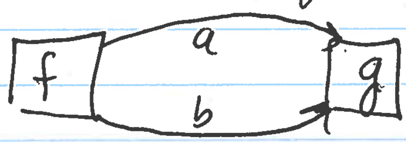
Tom Park's Algorithm
- simulation based
- schedules a KPN in bounded memory if possible
- starts with all buffers with size=1 and blocking sends
- use any working-conserving scheduling technique (run something if at least 1 process can run)
- if the system deadlocks due to blocking sends, increase the size of the smallest buffer and continue
- there no stopping condition: ~~else~~ just run for a "long" time and if buffers keep growing then probably a bounded implementation isn't possible

How to simulate finite buffers with a KPN simulation:
- every channel, a, add a virtual channel, av, in the opposite direction
- each virtual channel is initialized with n tokens where n is the buffer size



- send(a) ⇒ wait(a_r); send(a);
- wait(a) ⇒ wait(a); send(a_r);

- example of simulating a bounded buffer in a KPN simulation (which doesn't have blocking sends)



size(a) = 1

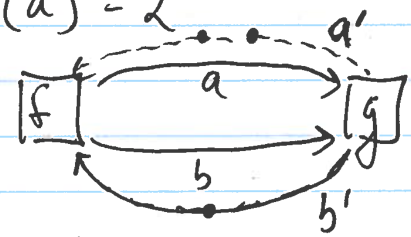
size(b) = 1

```
process f(out int a, out int b){
    for(;;) {
        send(1, a);
        send(1, a);
        send(1, b)
    }
}
```

```
process g(in int a, in int b) {
    for(;;) {
        wait(b);
        wait(a);
        wait(a);
    }
}
```

- this will deadlock on a blocking send

- set size(a) = 2




```
process f(out in a, in int a', ...){
    for(;;) {
        wait(a'); send(1, a);
        wait(a'); send(1, a);
        wait(b'); send(1, b);
    }
}
```

- won't deadlock with size(a) = 2

```
process g(in int a, out in a', ...){
    send(1, a'); send(1, a'); send(1, b');
    for(;;) {
        wait(b); send(1, b');
        wait(a); send(1, a');
        wait(a); send(1, a');
    }
}
```

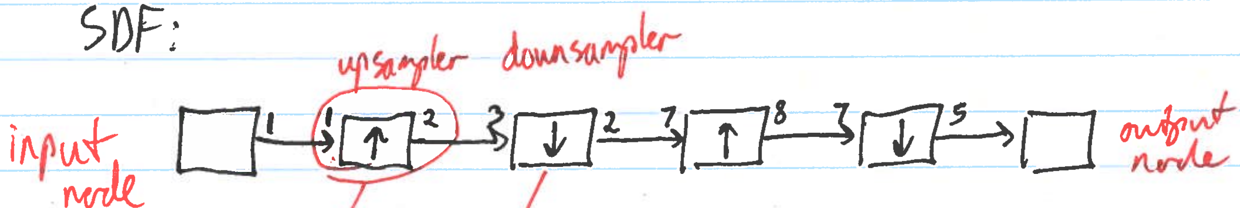
Synchronous Dataflow (SDF)

- Lee & Messerschmitt, 1987
- KPN is very flexible but a schedule can't be produced deterministically
- SDF adds restrictions to KPN to enable deterministic compile-time scheduling
- each node (process) has fixed production/consumption of tokens every time it fires
- node firing is atomic: it reads all input buffers at the same time
  - doesn't fire until the required # of tokens is available on each input
- the model ignores node execution time
- the system is described soley by the # of tokens read and written when each node fires

e.g. DAT-to-CD ~~audio~~ converter

DAT = digital audio tape samples at 44.1 kHz
CD = compact disc samples at 48 kHz

SDF:

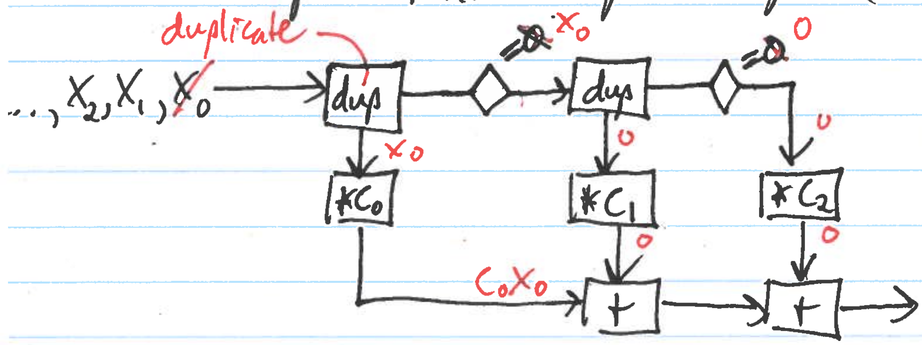


$$44.1 \times 2 / 3 * 2 / 7 * 8 / 7 * 5 = 44.1 \times \frac{160}{147} = 48$$

- can determine the relative firing rates of each node which leads to a periodic schedule

- unlike KPN, SDF doesn't permit initialization phases in nodes since input and output rates are fixed
- instead the SDF can start with initial tokens in buffers
  - these may be needed to avoid deadlock

- example: Finite Impulse Response (FIR) filter (SDF graph)



$y[0] = c_0 x_0 + 0 + 0$
$y[1] = c_0 x_1 + c_1 x_0 + 0$
$y[2] = c_0 x_2 + c_1 x_1 + c_0 x_0$

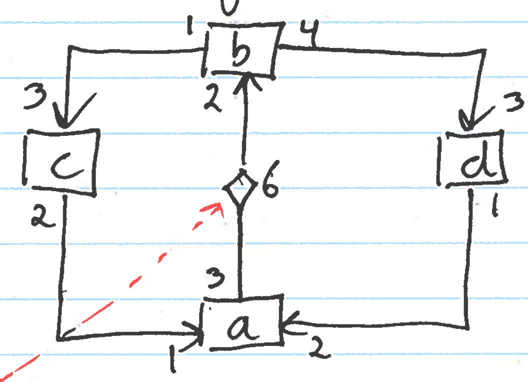
SDF scheduling algorithm

① establish node firing rates (per iteration of the periodic schedule) using balancing equations

② determine periodic schedule by simulating for 1 iteration (done when # tokens in each channel/buffer returns to its initial count)

- the resulting schedule will have bounded buffer size i.e. tokens will not accumulate

1 balancing equations



this channel/buffer has 6 initial tokens

- channel balancing equations

#output tokens firing rates

$$3a - 2b = 0$$ (no token accumulation)

#input tokens

$$4b - 3d = 0$$
$$b - 3c = 0$$
$$2c - a = 0$$
$$d - 2a = 0$$

$$\begin{array}{cccc} a & b & c & d \end{array}$$

$$\underbrace{\begin{bmatrix} 3 & -2 & 0 & 0 \\ 0 & 4 & 0 & -3 \\ 0 & 1 & -3 & 0 \\ -1 & 0 & 2 & 0 \\ -2 & 0 & 0 & 1 \end{bmatrix}}_{M} \underbrace{\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}}_{q} = 0 \qquad \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}\right)$$

M: topology matrix

q: node firing rates (per schedule iteration)

SDF Scheduling Theorem
- an SDF with n nodes has a periodic schedule iff $\text{rank}(M) = n-1$
- if $\text{rank}(M) = n-1$ $\exists$ a smallest positive integer solution to $Mq = 0$

- determine q without finding rank(M):
  i) take one node and run once e.g. $a=1$
  ii) iteratively determine rates of connected nodes

  e.g. $3(1) - 2b = 0; \quad b = 3/2$

  $-1(1) + 2c = 0; \quad c = 1/2$

  $-2(1) + d = 0; \quad d = 2$

$$x = \begin{bmatrix} 1 \\ 3/2 \\ 1/2 \\ 2 \end{bmatrix}$$

  iii) verify $M \cdot x = 0$ (it won't for an SDF that can't be scheduled periodically)
  iv) compute LCM of denominators e.g. 2
  (Least Common Multiple: $LCM(a,b) = \frac{a * b}{GCD(a,b)}$

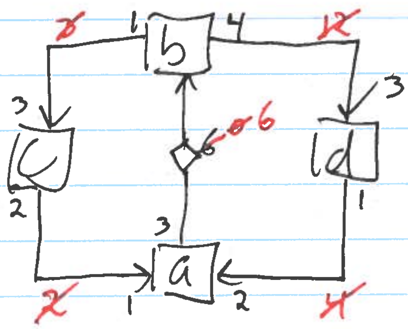
  $LCM(a,b,c) = LCM(a, LCM(b,c))$

  or multiply prime factors of highest power)
  v) $q = LCM \cdot x$

$$\text{e.g.} \quad \begin{array}{c|l} \begin{bmatrix} 2 \\ 3 \\ 1 \\ 4 \end{bmatrix} & \begin{array}{l} a \\ b \\ c \\ d \end{array} \end{array}$$

(2) determine periodic schedule
- done by simulation
- any schedule that doesn't cause buffer underflow works
- there may be multiple solutions

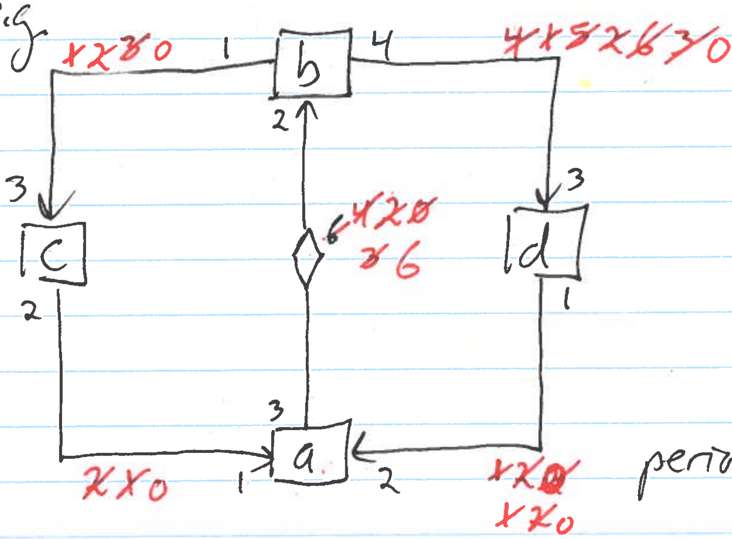
e.g.



$a = \cancel{2}\ 0$
$b = \cancel{3}\ 0$
$c = \cancel{1}\ 0$
$d = \cancel{4}\ 0$

✓ schedule = bbbcddddaa
periodic

e.g.



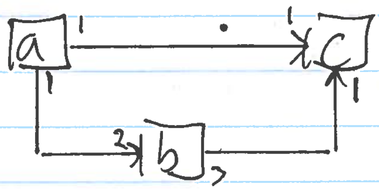
$a = \cancel{2}\ \cancel{1}\ 0$
$b = \cancel{3}\ \cancel{2}\ \cancel{1}\ 0$
$c = \cancel{1}\ 0$
$d = \cancel{4}\ \cancel{3}\ \cancel{2}\ \cancel{1}\ 0$

periodic schedule = bdbdbcadda

- this schedule requires smaller buffers

- special SDF cases

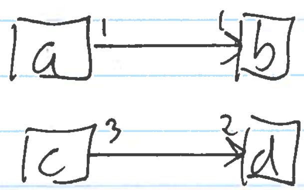
1) Inconsistent System, rank(M) = n



$$a - c = 0$$
$$\left.\begin{array}{l} a - 2b = 0 \\ 3b - c = 0 \end{array}\right\} \text{subst} \Rightarrow 3a - 2c = 0 \quad != $$

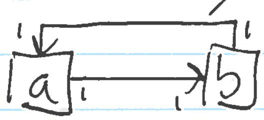
- the only solution is $q = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$ (never fire the nodes)

2) Underconstrained, rank(M) < n-1



- unconnected graphs; the firing rates of a/b and c/d are unrelated

3) Consistent system, No schedule



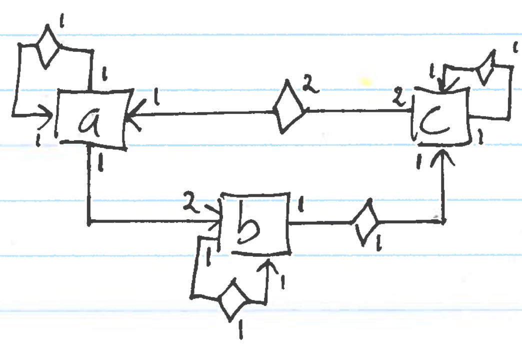
$q = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ - no initial inputs; can't run

- solution: need to add initial tokens
  - can often know the initial tokens needed based on the application
  - otherwise, can do a modified Park's algorithm
    - run simulation
    - if all nodes block on missing tokens, add tokens and continue

Multiprocessor SDF Scheduling

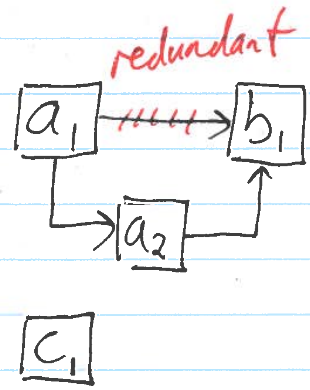
- need to know node execution times on each PE (processing element)
- compute q
- pick $j \geq 1$ (# of periodic repetitions)
- create a DAG (directed acyclic graph) of precedence relations between instances of node executions for $p = j\,q$
- if a node should not fire in parallel on different PEs (e.g. due to internal state), add a self-loop →

- example:

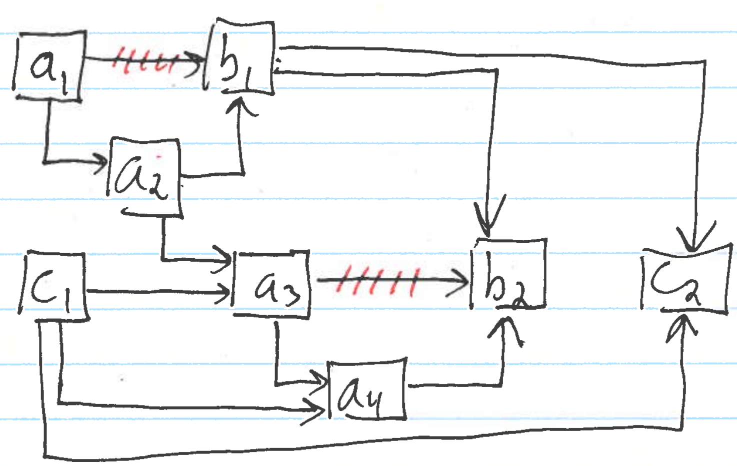


$$q = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}$$

$j = 1$

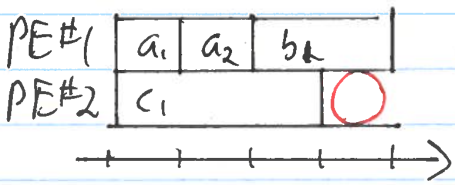


$$j = 2,\ p = \begin{bmatrix} 4 \\ 2 \\ 2 \end{bmatrix}$$



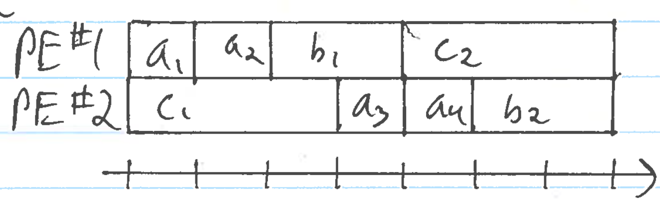
- scheduled on 2 homogeneous PEs
  $t_a = 1, \quad t_b = 2, \quad t_c = 3$

$j = 1$



throughput $= \frac{1 \text{ period}}{4}$

$j == 2$



throughput $= \frac{2}{7}$

- no idle time $\Rightarrow$ optimal

- to determine j, can increment j and schedule until throughput doesn't improve

Task Graph Scheduling

- a task graph is DAG of tasks (nodes) and precedence relations
- task latencies are known
- objective: schedule tasks on PEs to minimize <u>makespan</u> (time to execute the graph)

Listing Scheduling

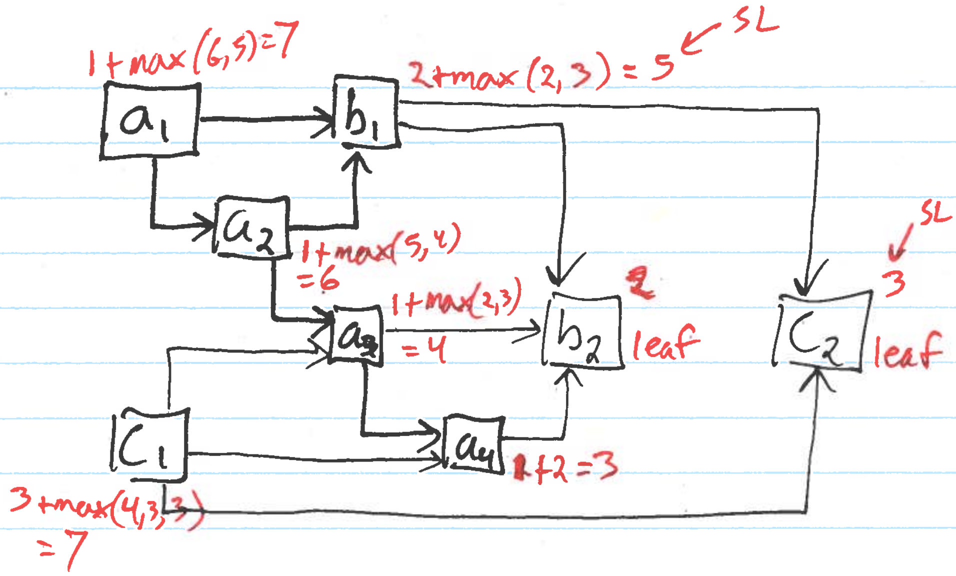
- it is a metaheuristic (doesn't specify how priorities are assigned)
  1) assign task priorities
  2) create list of active tasks (those not blocked by precedence constraints, in priority order)
  3) schedule first task from list on the PE where it can run earliest (obeying precedence constraints)
  4) remove this task from list and add any tasks it enables
  5) goto 3) until the list is empty

Priority Assignment

- use static level (SL) – the longest path to any leaf node

$$SL(x) = \max\left(\forall_{l \in \text{leaves}} \; \forall_{p \in \text{paths}(x..l)} \sum_{p_i = x}^{l} \text{latency}(p_i) \right)$$

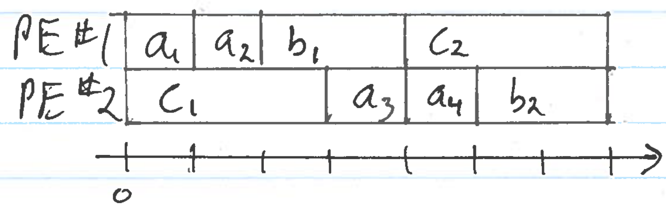
- to determine SL:
  1) for each leaf node (no successors), SL = node latency
  2) recursively set $SL(x) = \text{latency}(x) + \max_{\forall p_i \in \text{succ}(x)}(SL(p_i))$



$t_a = 1$
$t_b = 2$
$t_c = 3$

list

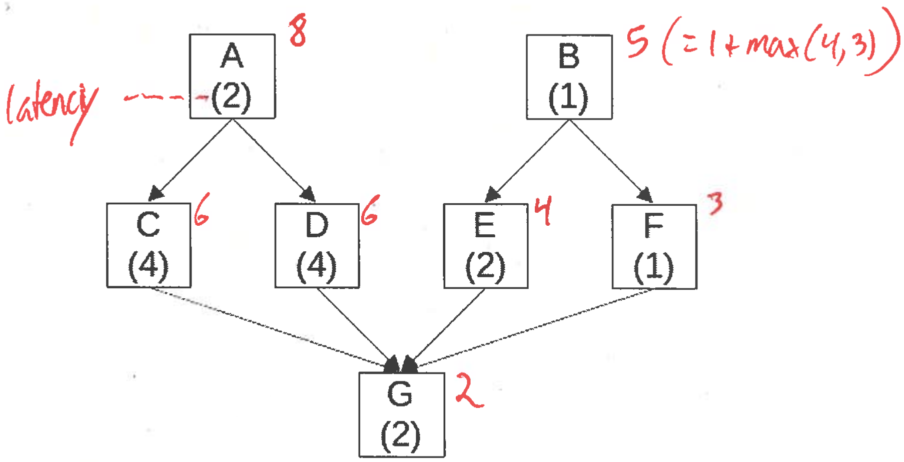
$\{\cancel{a_1}, c_1\}$
$\{\cancel{c_1}, a_2\}$
$\{\cancel{a_2}\}$
$\{\cancel{b_1}, a_3\}$
$\{\cancel{a_3}, c_2\}$
$\{\cancel{a_4}, c_2\}$
$\{\cancel{c_2}, b_2\}$
$\{\cancel{b_2}\}$
$\{\}$



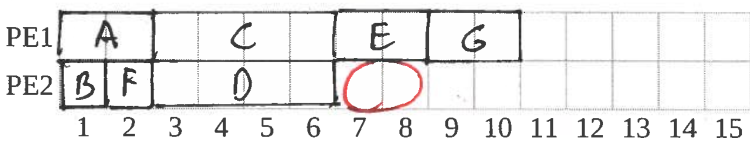
"forward list scheduling"
– schedule from $t=0$

Alternatives
- backward list scheduling: SL longest path to any root (no pred)
  - schedule from end

- dynamic list scheduling: if a lower priority task on the list can start earlier, then schedule it first
  - might reduce idle time
  - might penalize tasks on the critical path



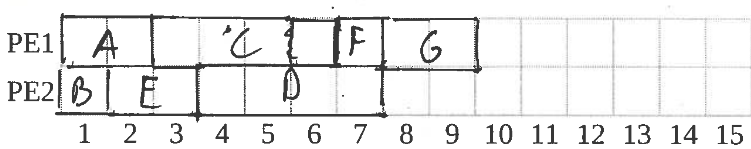
forward list scheduling



– 2 units of idle time before G runs

list
{~~A~~, B}
{~~C~~, D, B}
{~~D~~, B}
{~~B~~}
{~~E~~, F}
{~~F~~}
{~~G~~}
{ }

dynamic list scheduling



– no idle time before G runs (optimal)

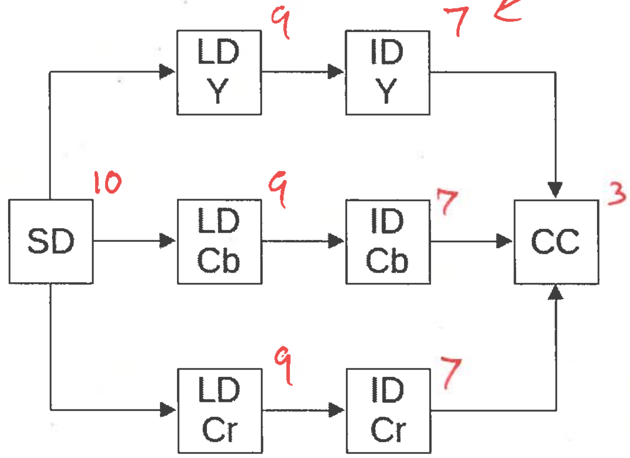
list
{~~A~~, B}
{C, D, ~~B~~}
{C, D, ~~E~~, F}
{~~C~~, D, F}
{~~D~~, F}
{~~F~~}
{~~G~~}
{ }

MJPEG423 decoder task graph

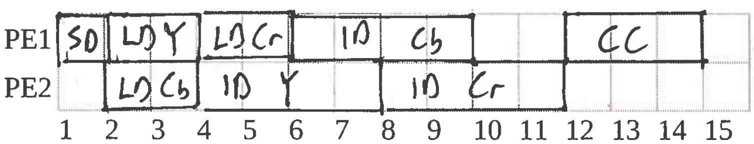
- assumptions:
  - 1 block per frame
  - lossless decode time is same for Y', Cb, Cr
- tasks:
  - read SD card (SD)
  - lossless decode (LD)
  - idct (ID)
  - colour conversion (CC)

page 22
- handout: 1st example (2 homogeneous PEs)
- handout: 2nd example (PE#2 - IDCT implemented in hardware)
  (limit ID tasks to PE#2)

example #1

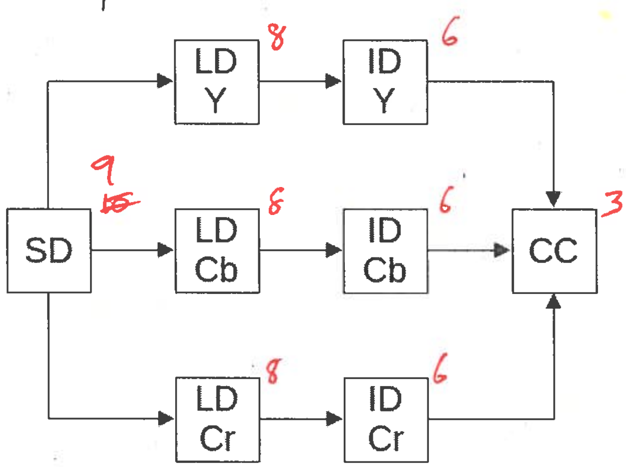


	PE #1	PE #2
SD	1	1
LD	2	2
ID	4	4
CC	3	3

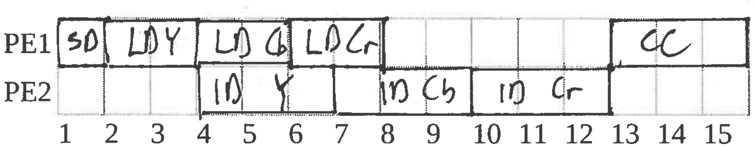


makespan = 14

example #2: PE #2 : FPGA



	PE #1	PE #2
SD	1	∞
LD	2	∞
ID	∞	3
CC	3	∞



makespan = 15

Periodic Task Graphs

- repeated execution of the task graph
- max period = makespan of a single iteration
- min period = $\lceil$workload / #PE$\rceil$, where workload is the sum of task latencies

page 24

- example #3 (2 periods): max = 14, min $\lceil 22/2 \rceil = 11$ (homogeneous PEs)

  - ~~period~~ - period = 11, latency = 23
  - extra buffers for IDY → CC, ID Cb → CC (, ID Cr → CC)

- example #4: max = 15 ← from pg 22, min $\lceil 19/2 \rceil = 10$

  - period = 10, latency = 20
  - extra buffers for ID Y → CC (, ID Cb → CC)

- extra buffer factor
  - regular load (repeating period):

    factor ≤ $\lceil$makespan / period$\rceil$ (makespan ← latency)

    e.g. example 3 : factor = $\lceil 23/11 \rceil = 3$

    example 4 : factor = $\lceil 20/10 \rceil = 2$

  - if ~~the~~ iterations vary:

    for each iteration i (← e.g. frame):

    $factor_i = 1$

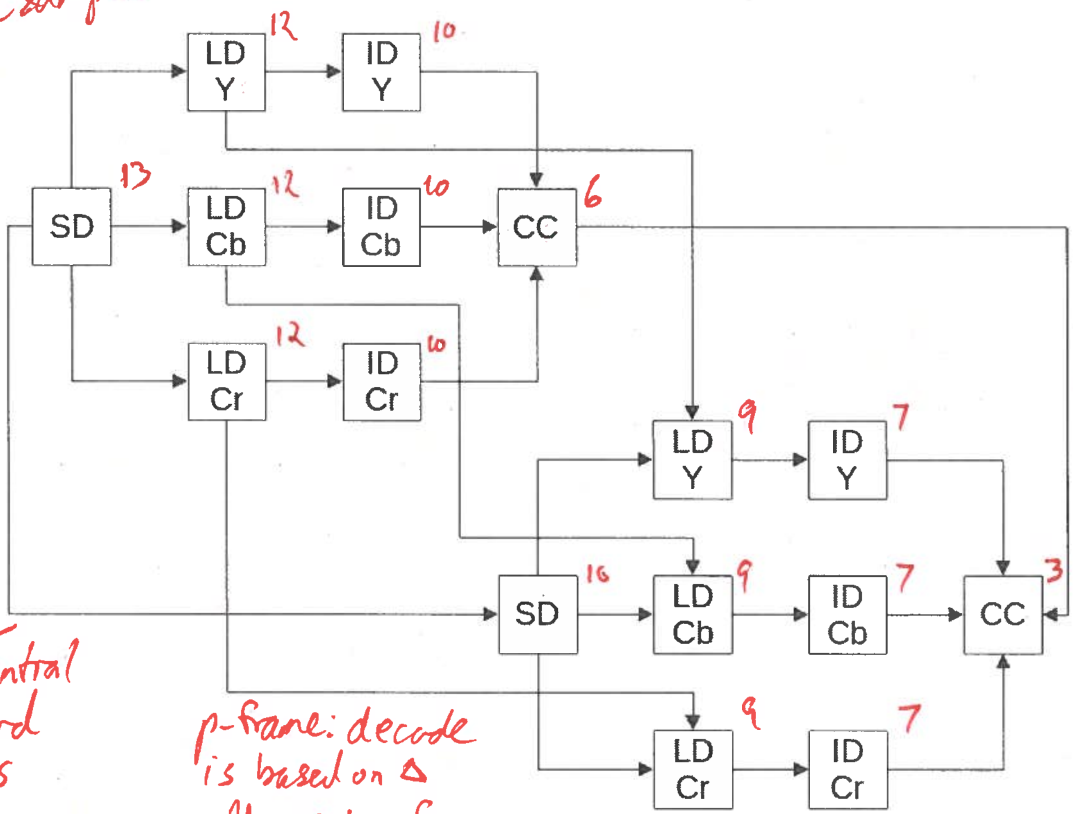
    s_i = start time, f_i = finish time

    for each other iteration j:

    $factor_i \mathrel{+}= (s_j \le s_i < f_j) \text{ or } (s_j < f_i \le f_j)$

  buffer factor = $\max(factor_i)$

example #3

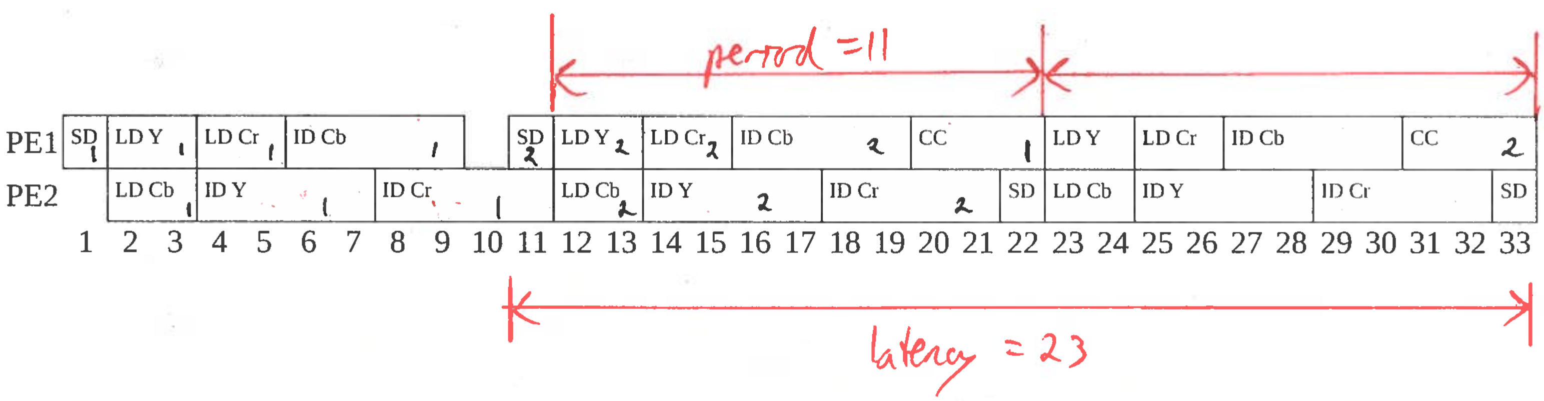


	PE #1	PE #2
SD	1	1
LD	2	2
ID	4	4
CC	3	3

– assume sequential buffer fill

sequential SD card reads

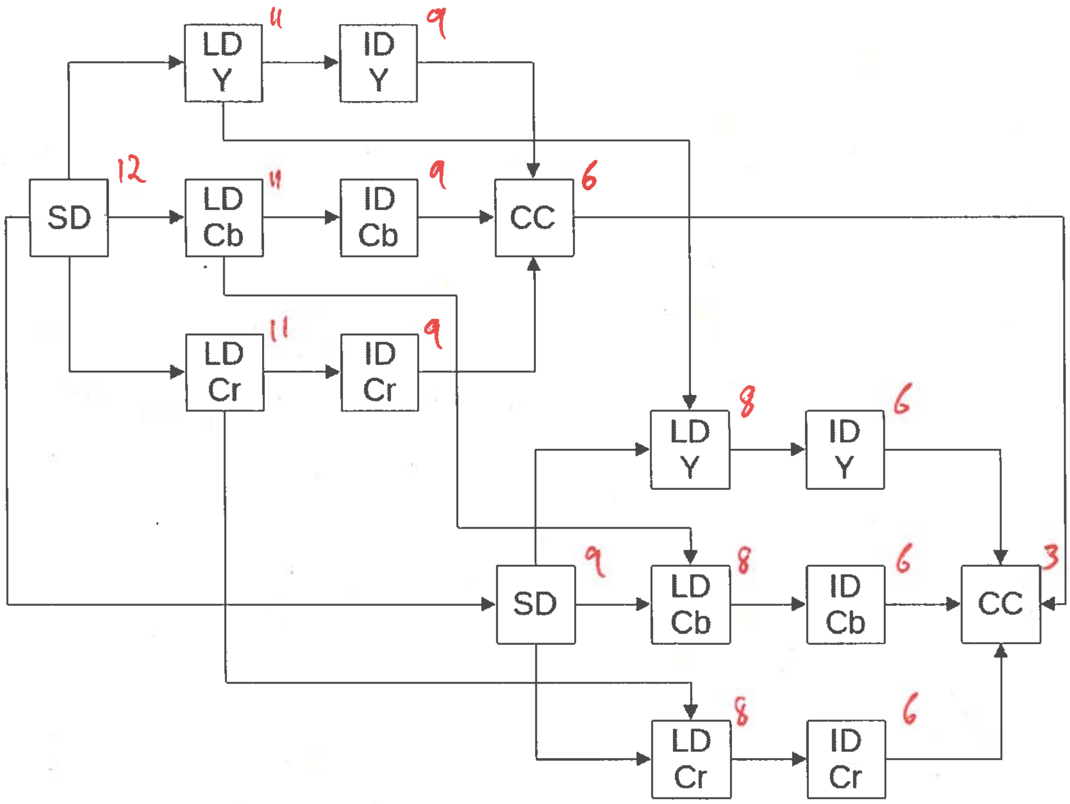
p-frame: decode is based on Δ with previous frame



example #4

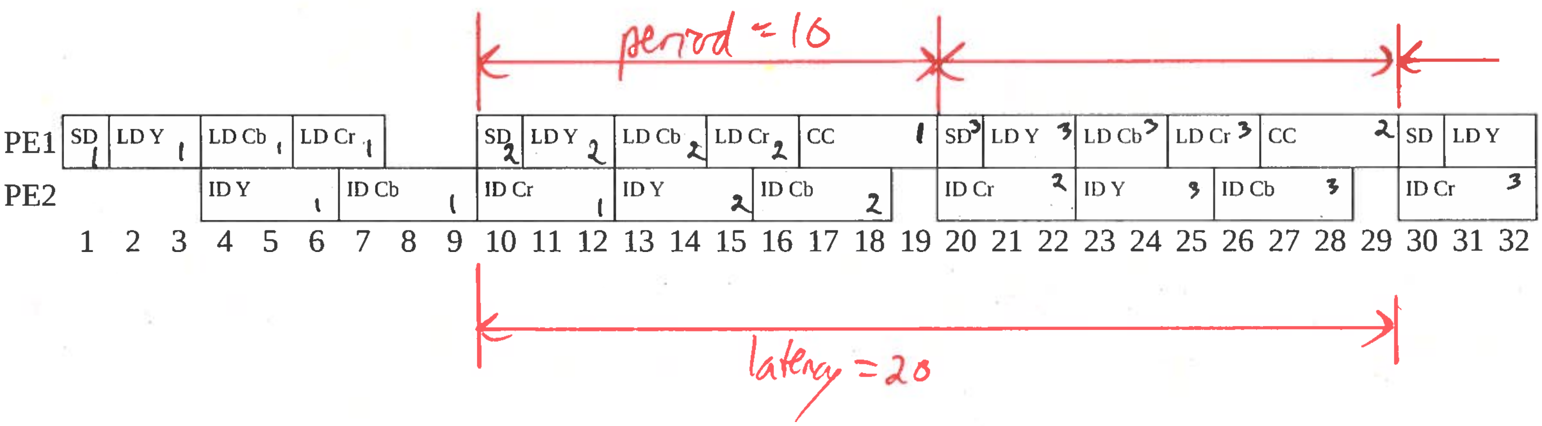
25

~~24~~



	PE #1	PE #2
SD	1	∞
LD	2	∞
ID	∞	3
CC	3	∞

hw IDCT



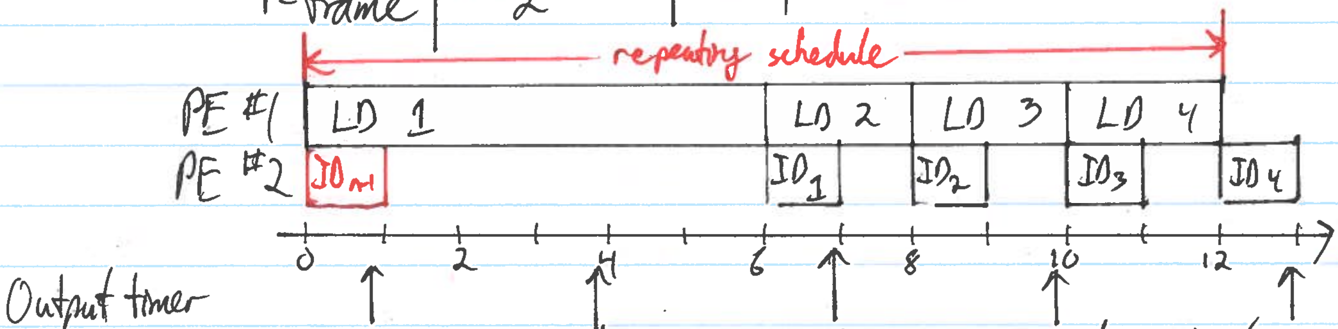
Periodic Output with Variable Load

e.g. I-frames and P-frames

sequence IPPP (only have LD and IDCT tasks)

1234

	LD (PE#1)	ID (PE#2)
I-frame	6	1
P-frame	2	1



- sequence period = 12, frame period = 12/4 = 3
- user a timer to perform output every 3 units (ISR vdma_out())
- worst-case: timer interrupt at $t = 7 - \varepsilon$

$\Rightarrow$ output frame 1 at $t = 10 - \varepsilon$

frame	1	2	3	4
s_i	0	6	8	10
f_i	10	12	14	16
$delay_i$	3	2	2	2
f_i^*	12	15	18	21

$delay_i = \lceil f_i / period \rceil - i$ (# missed output IRQs)

e.g. $delay_1 = \lceil 10/3 \rceil - 1 = 3$, $delay_2 = \lceil 12/3 \rceil - 2 = 2$

- when starting the sequence, skip $\max(delay_i)$ outputs

e.g. count = 3 ISR: if (count > 0) count--; else vdma_out()

- $f_i^* = (\text{delay} + i) \times \text{output period}$ adjusted finish time
- use s_i, f_i^* to compute buffer factor

- ~~[illegible]~~

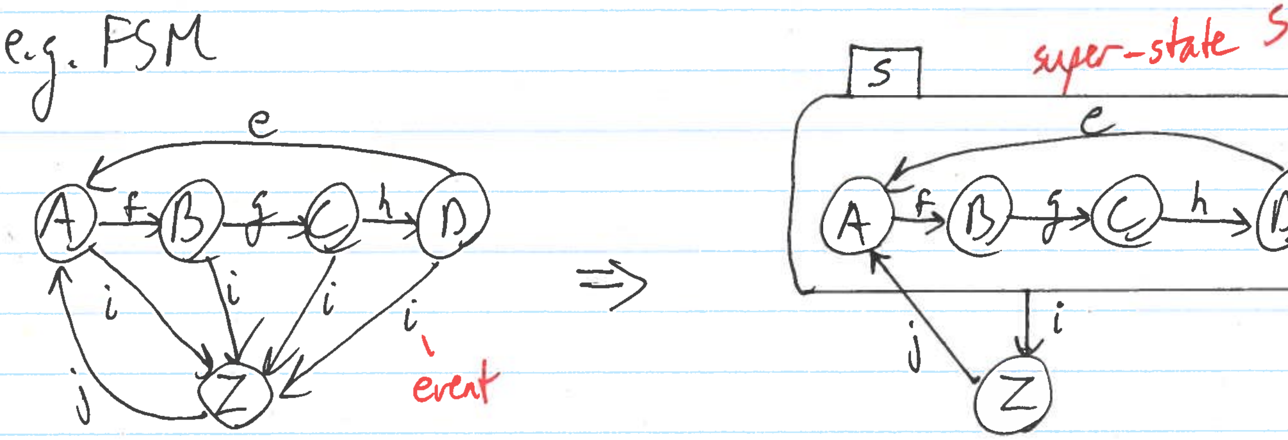
Executing a Task Graph

1) determine sequence on each PE
2) use a timer on each PE to start entry tasks
3) use synchronization primitives (e.g. semaphores) to block tasks until their predecessors have finished

State Charts

- hierarchical state machine
- tools exist to convert them to SW & HW

e.g. FSM

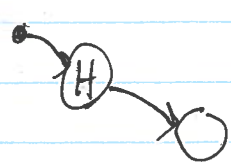


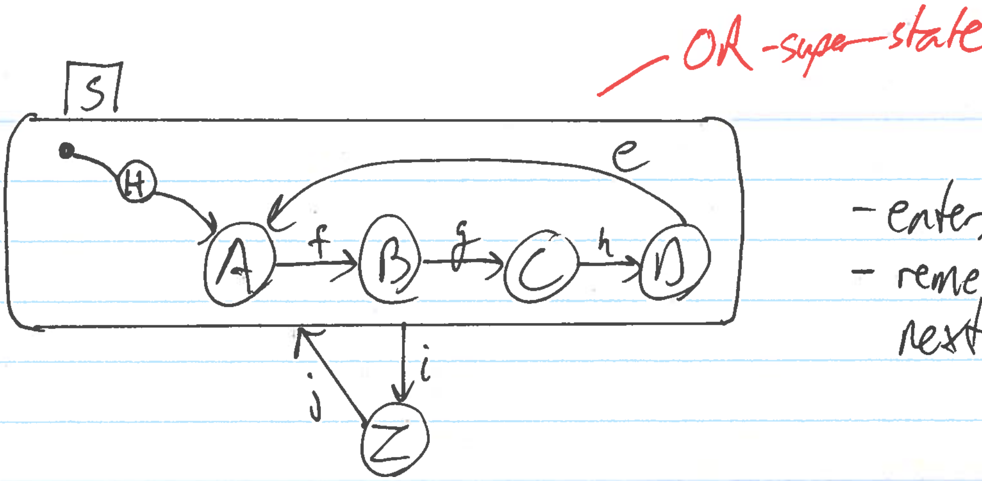
- definitions:

  active state = current state
  basic state has no sub-states
  super-states have substates
  ancester states = containing cases
  OR-super-state = in exactly 1 sub-state when it's active
  (models sequential execution)
  AND-super-state = in all substates when it's active
  (models concurrent execution)

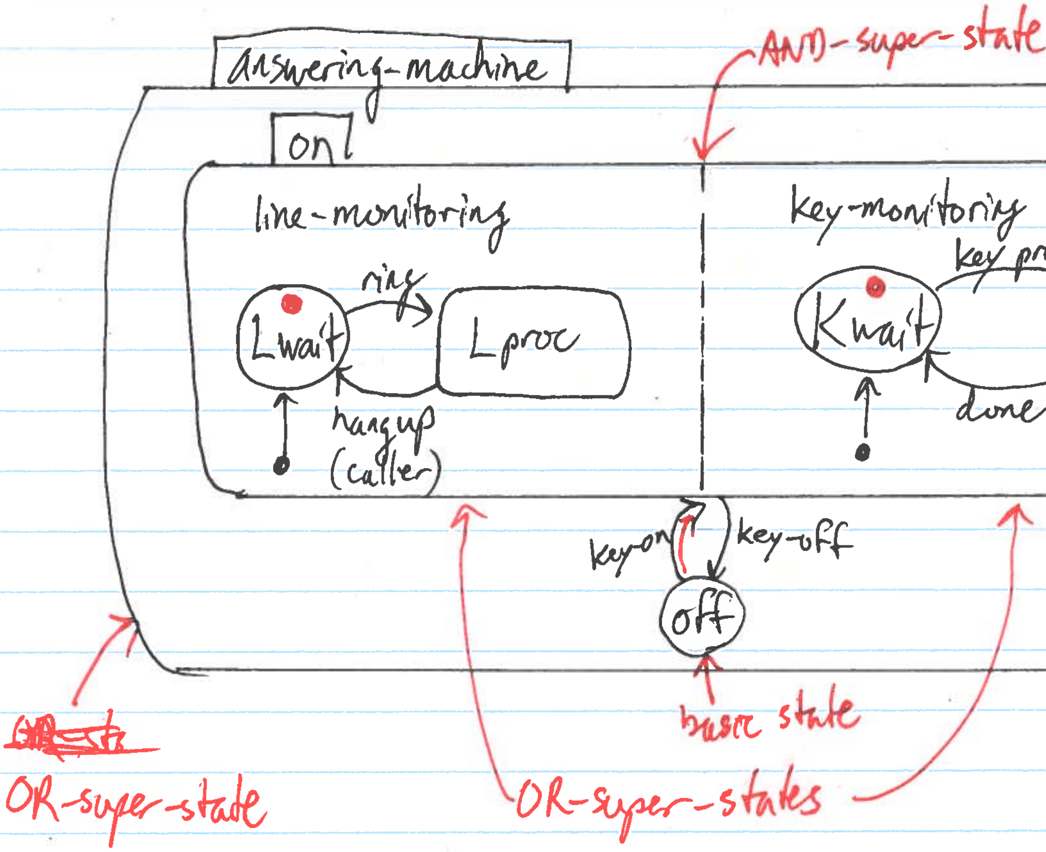
OR-super-state

- default state: 

- history 

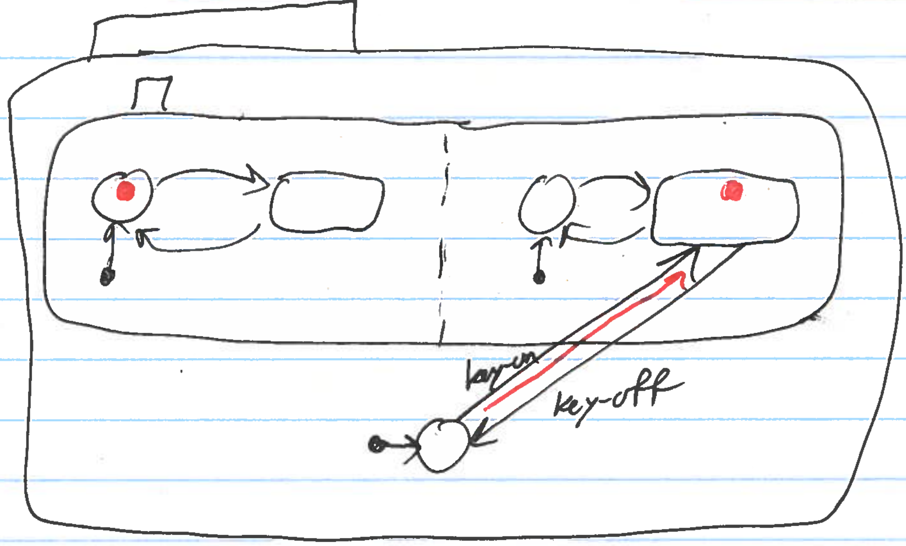


- enters A first time
- remembers last sub-state ~~if~~ next time S is active

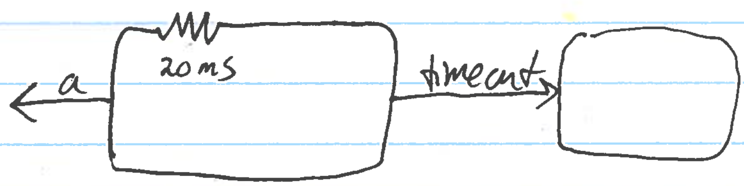


active states: Lwait, line-monitoring, Kwait, key-monitoring, on, answering-machine

- entering AND-super-state means entering all sub-states
- leaving " " leaving " "
- can transition into a sub-state of the AND-superstate

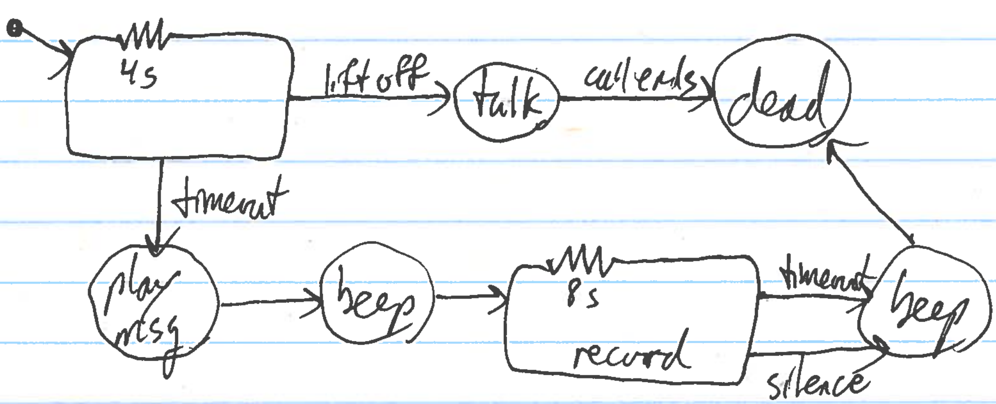


- Timers



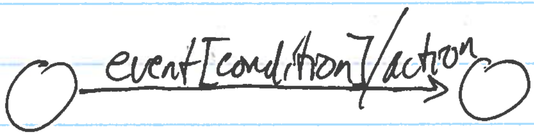
- stays in the state for at most the stated time
- must have a timeout edge

e.g.



Variables
- used to encode states with large numbers of values
  e.g. to model queueing system
  - use states to encode actions such as service
  - # of clients encoded as a variable

Transitions



- condition: based on variable values
- action: assignment to a variable or generate event

e.g.

$$\xrightarrow{\text{service-off}[c<=7]/c:=0}$$

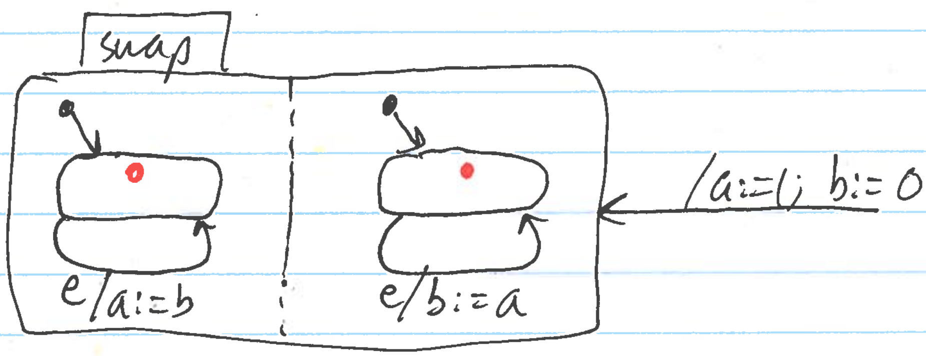
clients

State Chart Semantics

- models synchronous behaviour: all transitions fire at once
  1. evaluate events and conditions
  2. determine transitions which happen
  3. fire all transitions and apply any actions

- time semantics
  1. evaluate all internally generated events
  2. apply transitions and repeat until stable
  3. advance simulation time to next external event (or timeout)

- example:



- when event e arrives, left state assigns 0 to a; right state assigns 1 to b

- translates well to hardware
- can produce inefficient software implementations