1.

(a)

Benefits:

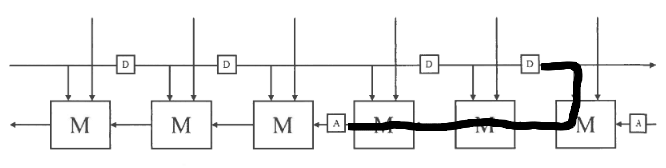
* Significantly higher throughput compared to a microprocessor
* Significantly higher energy efficiency compared to a microprocessor
* Significantly better customisation opportunities (e.g. in accuracy, where you can use 10 bits for calculations rather than fixed 16 or 32 bits) compared to a microprocessor
* Some designs can’t be implemented on a microprocessor and meet specifications
* Performance is fully deterministic – no instructions, cache misses, branch predictions, etc.
* Interfacing with very fast IO – e.g. driving a 4k monitor or 85MPixel camera sensor.

Drawbacks:

* Higher financial cost compared to a microprocessor?
* Might be infeasible for accelerating very dynamic parts of code? Not everything is suitable to be implemented on an FPGA.
* Implementing a design in HW is more difficult.

(b)

Critical path delay: 3 \* T (a critical path = the longest path between any two registers or a register and the input/output)



Q needs to rewire the 3-wide bus into 2-wide bus and single element.  
P can be split into two parts. Top part is a rewire block that orders wires s.t. they can go into M and bottom part is just M. Then append D at the end. I hope that two solutions for P are equivalent but honestly idk.

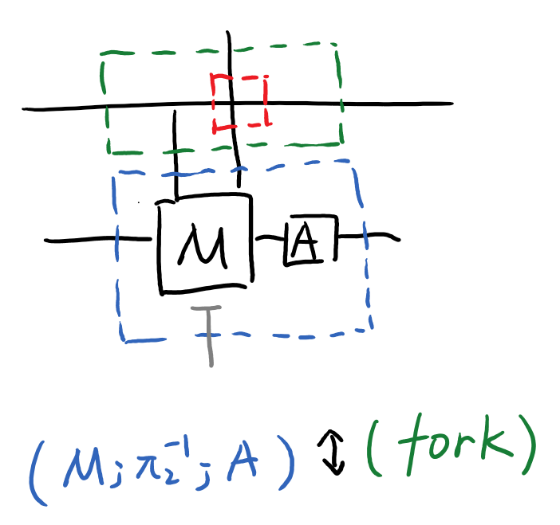
Q = <x, <y, y, y>> $wire <<x, <y, y>>, y>

P = VAR x, y, z . <<x, y>, z> $rel <M <x, <y, z>>, D y>  
P = (M; pi2^~1) <|> (fork; lsh; swap); pi2; snd file:///home/markus/Downloads/C318.pdf D

S = M; fst A

|  |  |
| --- | --- |
|  |  |
| This is what S looks like | This is what P looks like |
|  |  |

Now I get some clues about S,

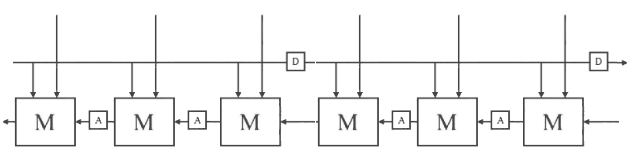


I assume red part is ‘id’, so it is omitted. Then

S = (M; pi2^~1; A) <|> (fork; snd; pi1) [See CuCell2]

P = (M; pi2^~1) <|> (fork; snd (pi1; D)) [See CuCell1]

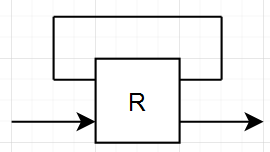
(c)



Critical path is 2T. Wires on top have no delay.

2.

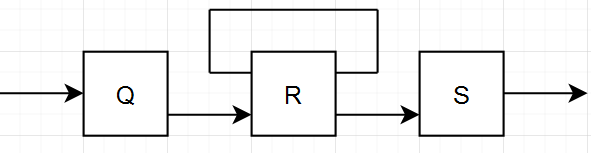
A)



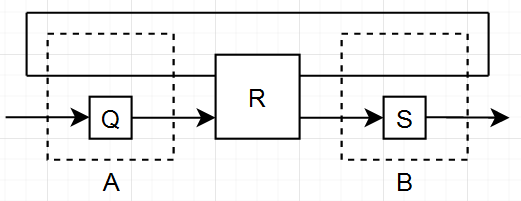
B)

A=[Q,id]  
B=[S,id]

LHS



RHS



C) C = snd Q

D)

μ swap = id  
μ ([Q, R]; swap) = Q; R

E)

Take the case from C). This can be easily turned in to a state machine if we take C = DI 0. Then is the output and is the state transition. Counter can be easily implemented with .

3)

a)

DFEVar b = io.scalarInput(“b”, floatType);

DFEVar[] xs = new DFEVar[N];

for (int i = 1; i <= N; i++) {

xs[i – 1] = io.input(“x\_” + i, floatType);

}

DFEVar y = constant.var(0, floatType);

for (int i = 0; i < N; i++) {

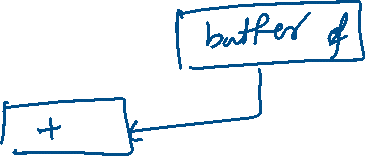
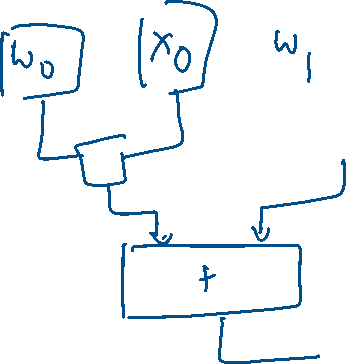
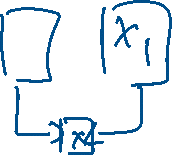
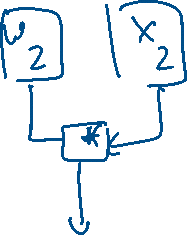
y = y + w[i] \* xs[i] + b;I

}

io.output(“y”, y >= 0 ? 1 : 0);

b)

Perhaps something like this?



c) Note that we want to find the number of *buffers*.

For *N* inputs, we will get *N* – 2 + (*N* – 1) + … buffers, which would be .

Adapt the design from the lecture. It’s pretty much the same thing – instead of the single adder you have the perceptron evaluation and before output you have a MUX checking if the result is

