

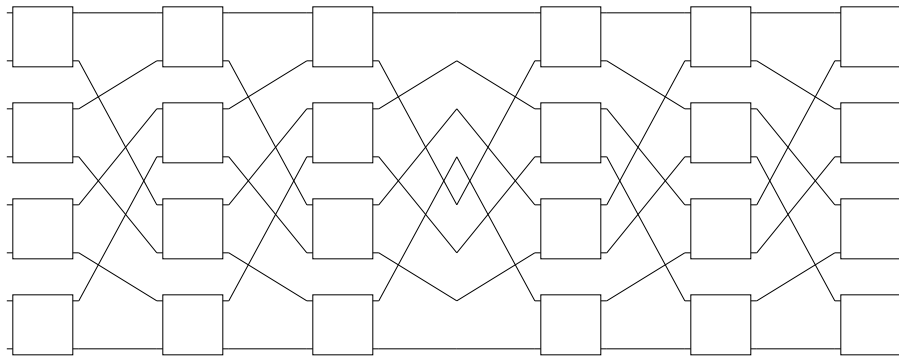
CS 573

Spring, 2004

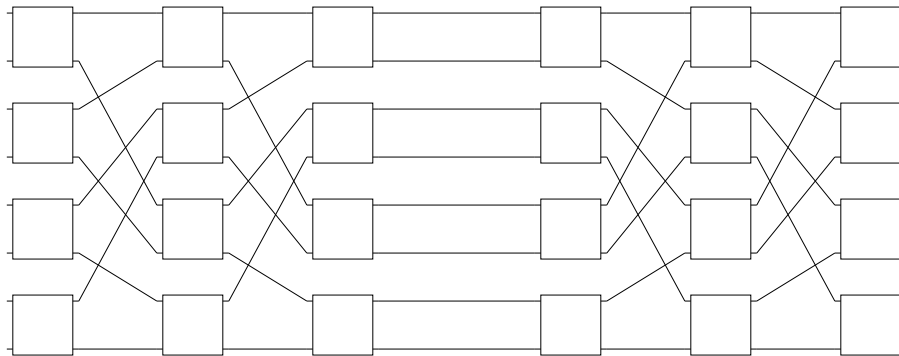
HW 3 Solutions

1. Showing two dynamic hypercube networks back-to-back is equivalent to a Benes network:

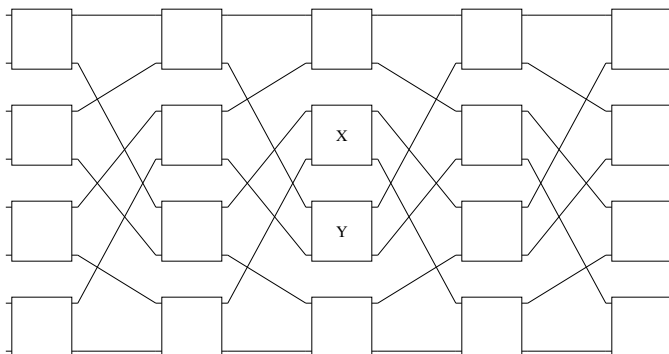
First, we show two dynamic hypercubes back to back.



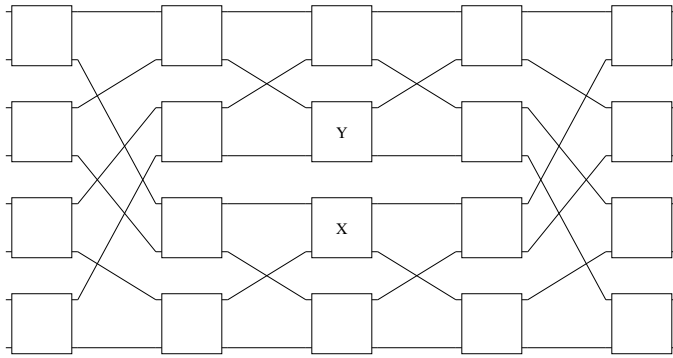
Now, we straighten out the kinks in the middle:



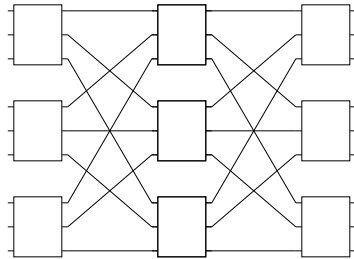
The center two columns of switch boxes don't do anything a single box doesn't do, so we combine them. We're also going to mark a couple of the switch boxes so we'll be able to keep track of them when we start moving them around.



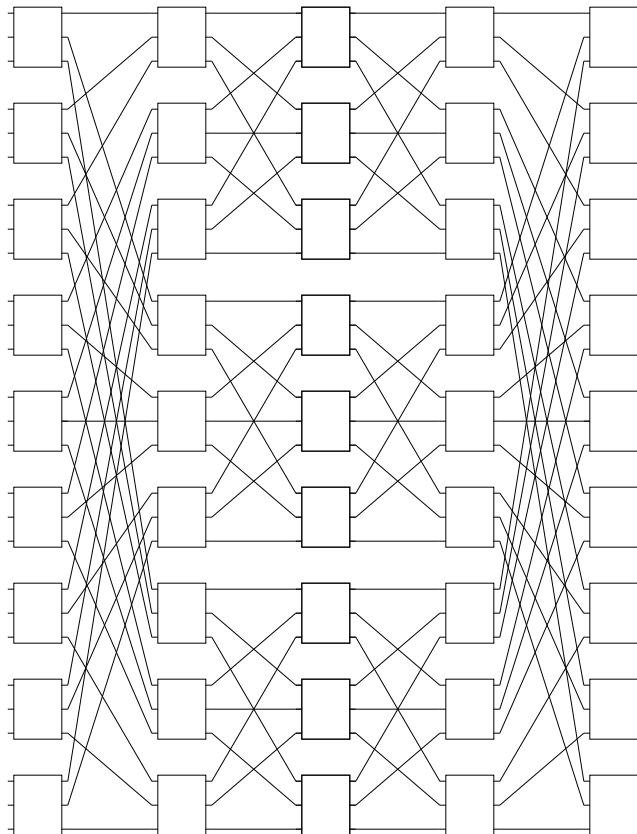
What we have here is actually the Benes network as formally defined in the paper. But it's not the familiar form we're used to seeing elsewhere. To get that, we just reverse boxes X and Y, and we have a Benes network.



2. First, we construct a 9x9 Benes network out of 3x3 switchboxes. We use three columns with three switchboxes per column (instead of 3 columns of 2 each like we would if we were building a 4x4 network out of 2x2 boxes); the three outputs of the first box go to the first output of each of the three boxes in the next stage in analogy with how the 2x2 construction works.



Now we stack three of these networks, put a column before and a column after, and connect in analogy to before.



3. Since the goal of this code is to let only one of the processes in, it can fail by either letting both in or by getting “wedged” so neither can get in. Since the code only writes variable values in reference to the processor’s own number (so it doesn’t look at one shared variable in order to set another), and it does all its variable setting before entering the `while` loop, it’s hard to imagine that it could get wedged. This implies we have to have a race condition, in which one of the processors inspects a variable before the other one has set it, and so enters the critical section when, if the other processor had won the race, it wouldn’t have. Since `flag` is the last variable set, and the first checked, that seems to be the likeliest candidate. In order to get it to fit on a line, I’m going to replace “turn” with “t”, “flag” with “f,” “true” with “1”, and “false” with “0”. Hopefully it’ll still be clear.

P0	W(t)1	W(f[0])1	R(f[1])0											
P1	W(t)0			W(f[1])1	r(f[0])1	R(t)1								

In words,

- P1 sets `turn` to 0
 - P0 sets `turn` to 1, sets `flag[0]` to true, and starts to enter the `while` loop.
 - P0 reads `flag[1]`, and finds it is false. It is able to enter the critical section.
 - P1 sets `flag[1]` to true, and starts to enter the `while` loop.
 - P1 reads `flag[0]`, and finds it true. So it goes on to check `turn`, and finds it is equal to 1, so P1 is also able to enter the critical section.
4. Looking at each of them:

Bit Names Addresses	M8 1100	M7 1011	M6 1010	M5 1001	C8 1000	M4 0111	M3 0110	M2 0101	C4 0100	M1 0011	C2 0010	C1 0001	P 0000	Syndrome
Parity	0	0	0	1	1	1	0	0	1	0	1	0	1	0
C8	0	0	0	1	1									0
C4	0					1	0	0	1					0
C2		0	0			1	0			0	1			0
C1		0		1		1		0		0		0		0

Since the syndrome is 0, there is no error.

Bit Names Addresses	M8 1100	M7 1011	M6 1010	M5 1001	C8 1000	M4 0111	M3 0110	M2 0101	C4 0100	M1 0011	C2 0010	C1 0001	P 0000	Syndrome
Parity	0	1	0	1	0	0	1	1	1	0	1	1	0	1
C8	0	1	0	1	0									0
C4	0					0	1	1	1					1
C2		1	0			0	1			0	1			1
C1		1		1		0		1		0		1		0

This time the syndrome is non-zero, and the parity bit is wrong. So there is a one-bit error, in bit 0110. Correcting that bit, the corrected value is 0101000110110.

Bit Names Addresses	M8 1100	M7 1011	M6 1010	M5 1001	C8 1000	M4 0111	M3 0110	M2 0101	C4 0100	M1 0011	C2 0010	C1 0001	P 0000	Syndrome
Parity	1	0	1	0	0	1	0	1	0	0	1	1	0	0
C8	1	0	1	0	0									0
C4	1					1	0	1	0					1
C2		0	1			1	0			0	1			1
C1		0		0		1		1		0		1		1

This time the syndrome is non-zero, but the parity bit is correct. Consequently there is a two-bit error, which we have detected but cannot correct.