A List-Decoding Algorithm for Polar Codes

I. Tal, A. Vardy, *IEEE Trans. on Information Theory*, pp. 2213–2226, 2015.

Viterbi & SCL

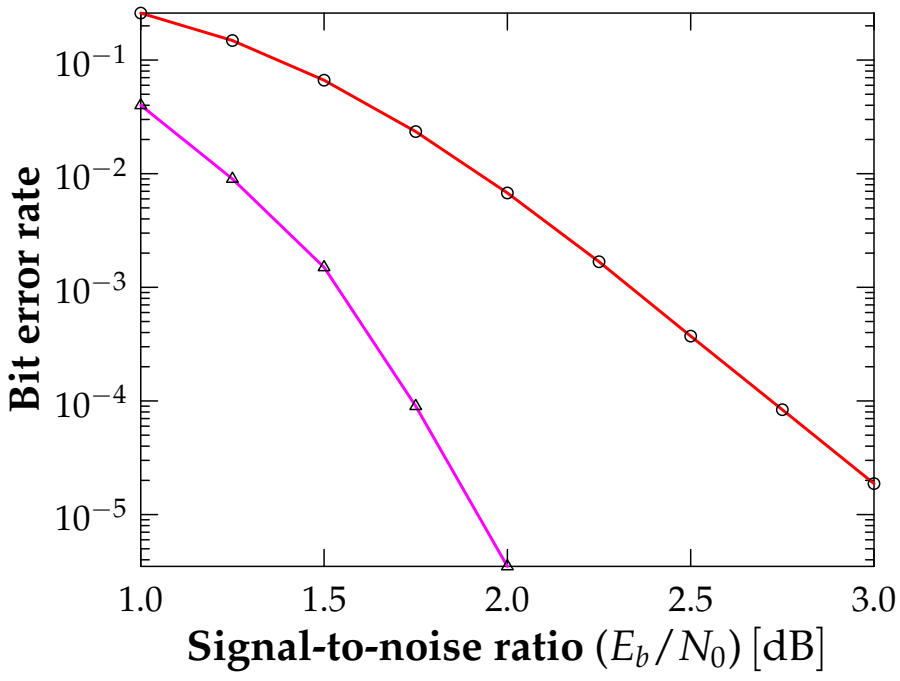
Connections between Viterbi's decoding algorithm and the Successive Cancellation List decoding algorithm

	Viterbi	SCL
Decoding algorithm?	✓	✓
Code family	convolutional	polar
Optimal?	✓	almost, sometimes
Recursion?	✓	✓
Data structures?	✓	✓

Successive Cancellation

decide on	based on
$\hat{u}_0$	$W_0(\mathbf{y}_0^{n-1}\|u_0=0) \lessgtr W_0(\mathbf{y}_0^{n-1}\|u_0=1)$
$\downarrow$	
$\hat{u}_1$	$W_1(\mathbf{y}_0^{n-1},\hat{u}_0\|u_1=0) \lessgtr W_1(\mathbf{y}_0^{n-1},\hat{u}_0\|u_1=1)$
$\downarrow$	
$\hat{u}_2$	$W_2(\mathbf{y}_0^{n-1},\hat{u}_0^1\|u_2=0) \lessgtr W_2(\mathbf{y}_0^{n-1},\hat{u}_0^1\|u_2=1)$
$\downarrow$	
$\vdots$	$\vdots$
$\downarrow$	
$\hat{u}_i$	$W_i(\mathbf{y}_0^{n-1},\hat{u}_0^{i-1}\|u_i=0) \lessgtr W_i(\mathbf{y}_0^{n-1},\hat{u}_0^{i-1}\|u_i=1)$
$\downarrow$	
$\vdots$	$\vdots$
$\downarrow$	
$\hat{u}_{n-1}$	$W_{n-1}(\mathbf{y}_0^{n-1},\hat{u}_0^{n-1}\|u_{n-2}=0) \lessgtr W_{n-1}(\mathbf{y}_0^{n-1},\hat{u}_0^{n-2}\|u_{n-1}=1)$

SC not good enough



Legend:

- successive cancellation, $n = 2048$, $k = 1024$
- LDPC (Wimax standard, $n = 2304$)

- Is the SC decoder under-performing?
- Are the polar codes themselves weak at this length?

A critical look at SC

Successive Cancellation Decoding

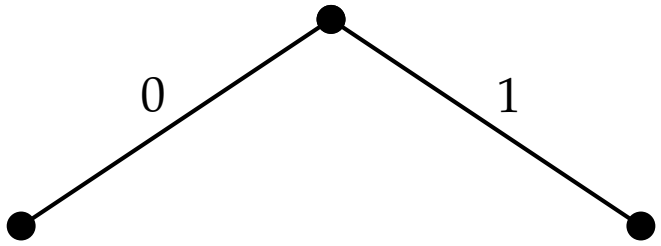
for $i = 0, 1, \ldots, n-1$ **do**
  if $\widehat{u}_i$ *is frozen* **then** set $\widehat{u}_i$ accordingly ;
  else
    if $W_i(\mathbf{y}_0^{n-1}, \widehat{u}_0^{i-1}|0) > W_i(\mathbf{y}_0^{n-1}, \widehat{u}_0^{i-1}|1)$ **then**
      set $\widehat{u}_i \leftarrow 0$;
    else
      set $\widehat{u}_i \leftarrow 1$;

Potential weaknesses (interplay):

- Once an unfrozen bit is set, there is "no going back". A bit that was set at step i can not be changed at step $j > i$.
- Knowledge of the value of future frozen bits is not taken into account.

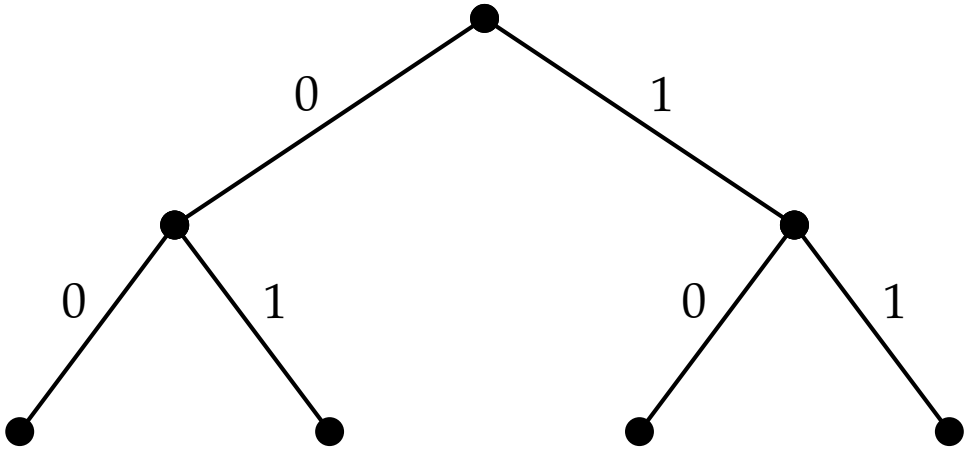
List decoding of polar codes

Key idea: Each time a decision on $\widehat{u}_i$ is needed, split the current decoding path into two paths: **try both $\widehat{u}_i = 0$ and $\widehat{u}_i = 1$.**



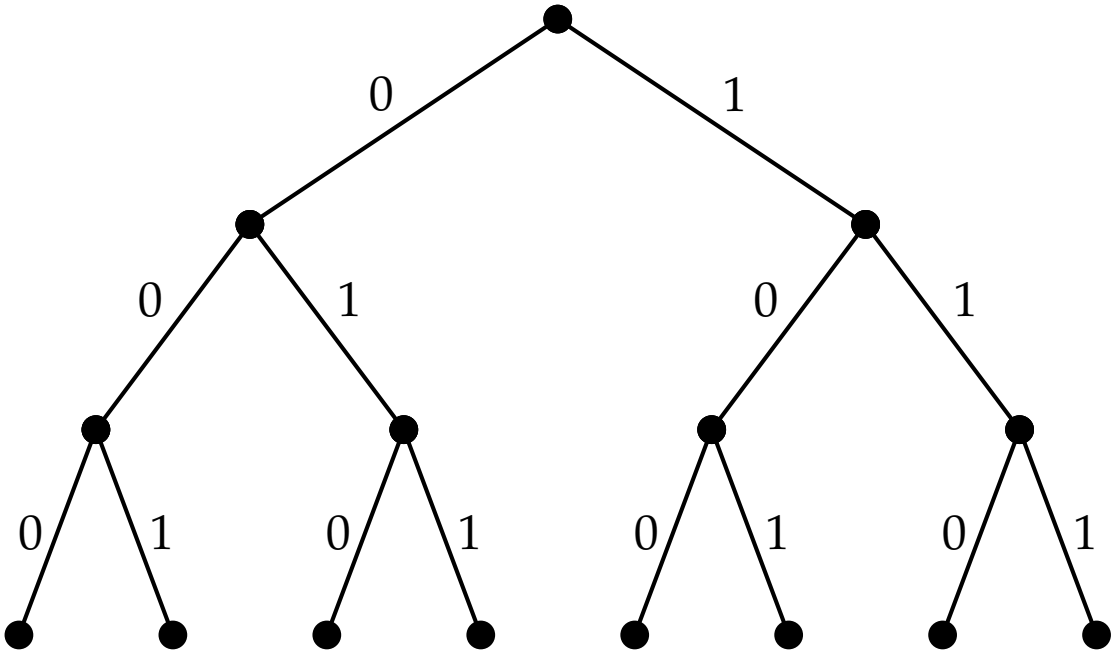
List decoding of polar codes

Key idea: Each time a decision on $\widehat{u}_i$ is needed, split the current decoding path into two paths: **try both $\widehat{u}_i = 0$ and $\widehat{u}_i = 1$.**



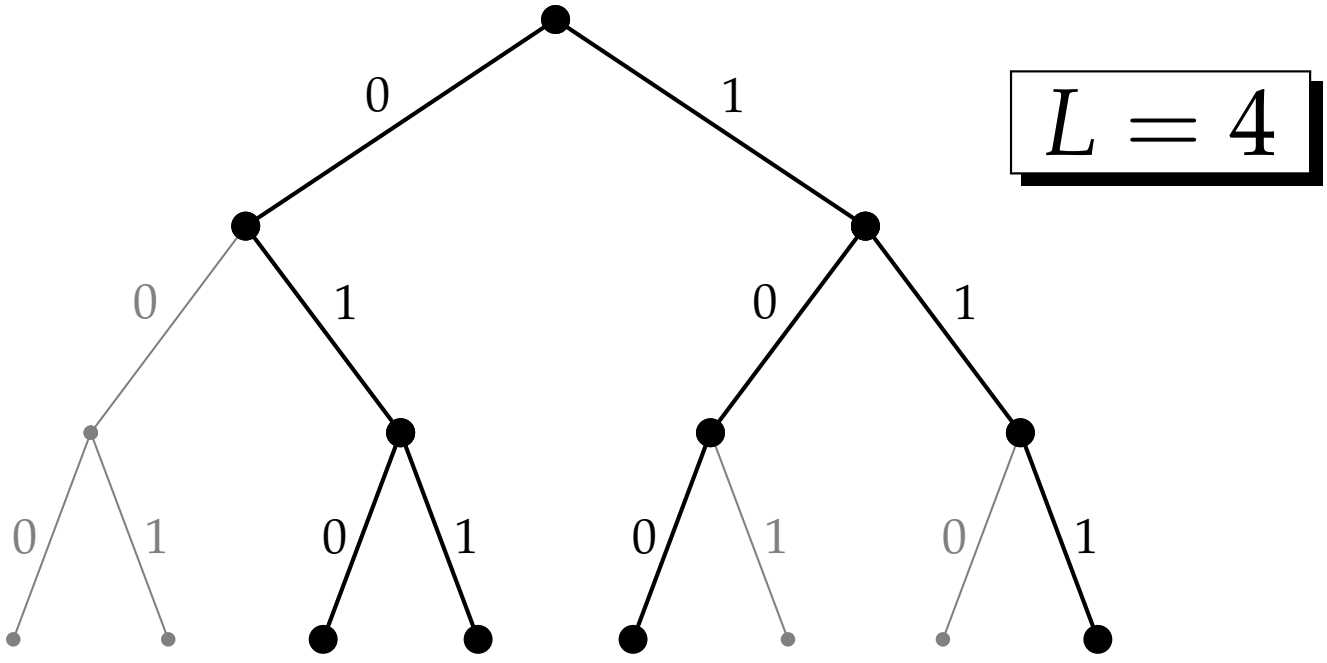
List decoding of polar codes

Key idea: Each time a decision on $\widehat{u}_i$ is needed, split the current decoding path into two paths: **try both $\widehat{u}_i = 0$ and $\widehat{u}_i = 1$.**



List decoding of polar codes

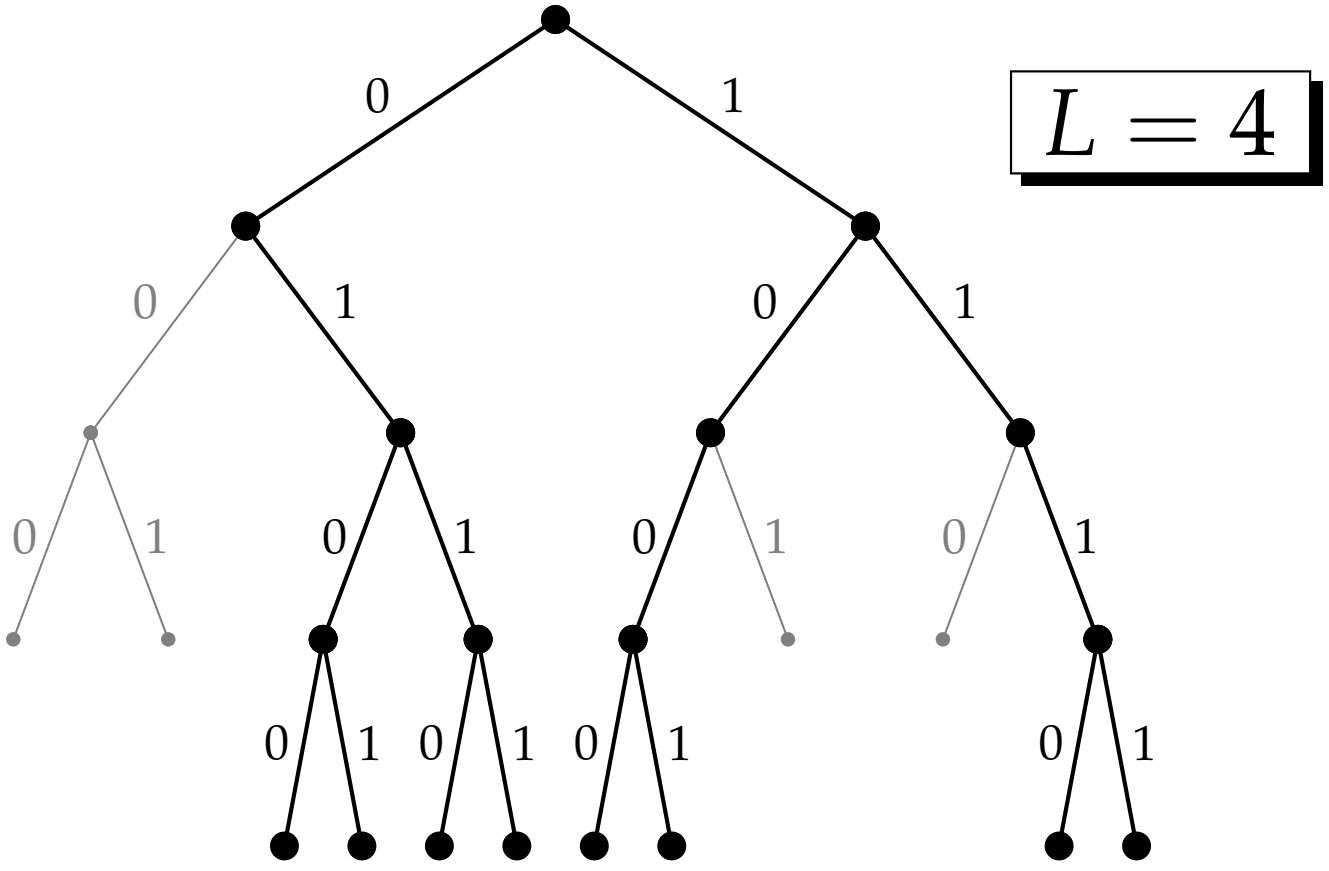
Key idea: Each time a decision on $\widehat{u}_i$ is needed, split the current decoding path into two paths: **try both $\widehat{u}_i = 0$ and $\widehat{u}_i = 1$.**



When the number of paths grows beyond a prescribed threshold L, discard the worst (least probable) paths, and keep only the L best paths.

List decoding of polar codes

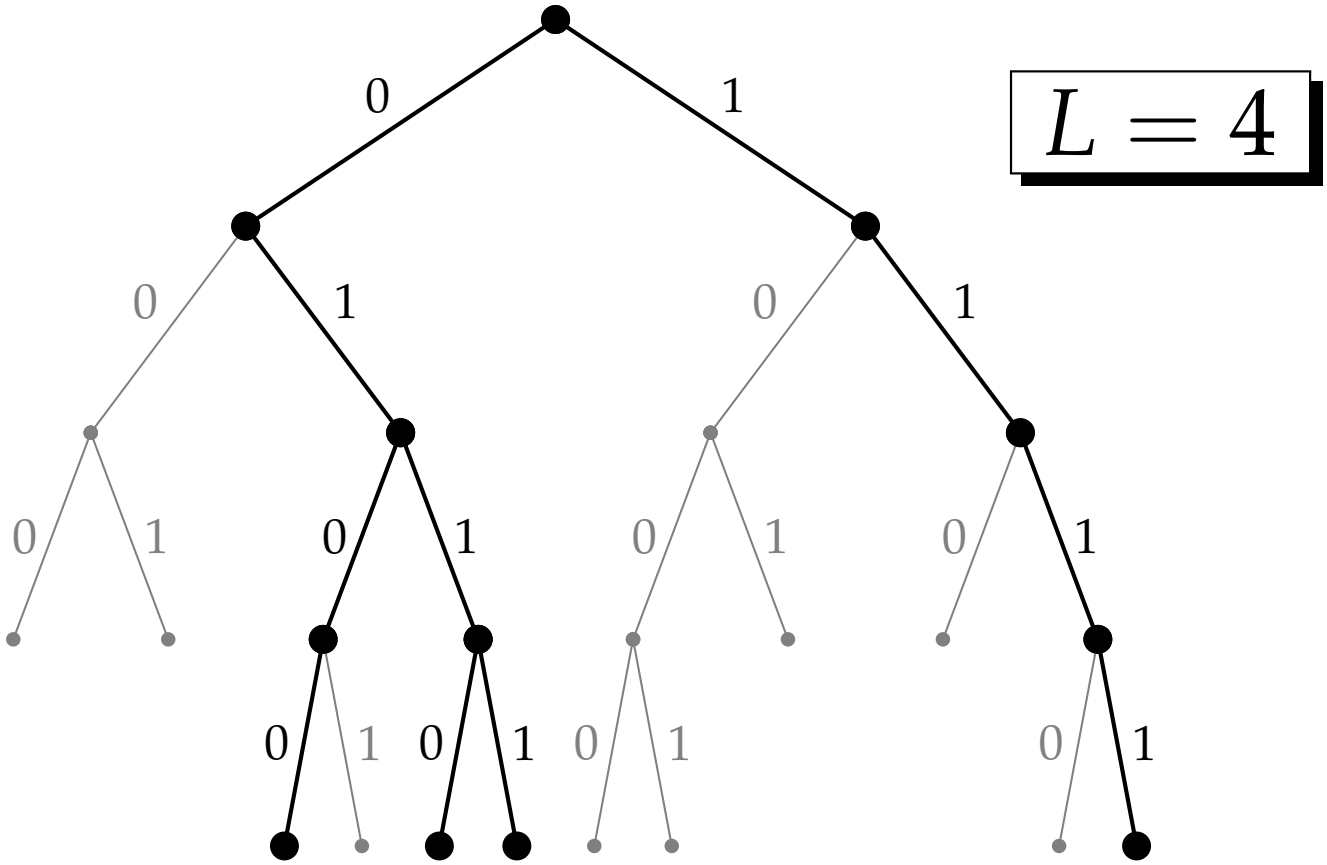
Key idea: Each time a decision on $\widehat{u}_i$ is needed, split the current decoding path into two paths: **try both $\widehat{u}_i = 0$ and $\widehat{u}_i = 1$.**



When the number of paths grows beyond a prescribed threshold L, discard the worst (least probable) paths, and keep only the L best paths.

List decoding of polar codes

Key idea: Each time a decision on $\widehat{u}_i$ is needed, split the current decoding path into two paths: **try both $\widehat{u}_i = 0$ and $\widehat{u}_i = 1$.**



When the number of paths grows beyond a prescribed threshold L, discard the worst (least probable) paths, and keep only the L best paths.

At the end, select the single most likely path.

List-decoding: complexity issues

The idea of branching while decoding is not new:

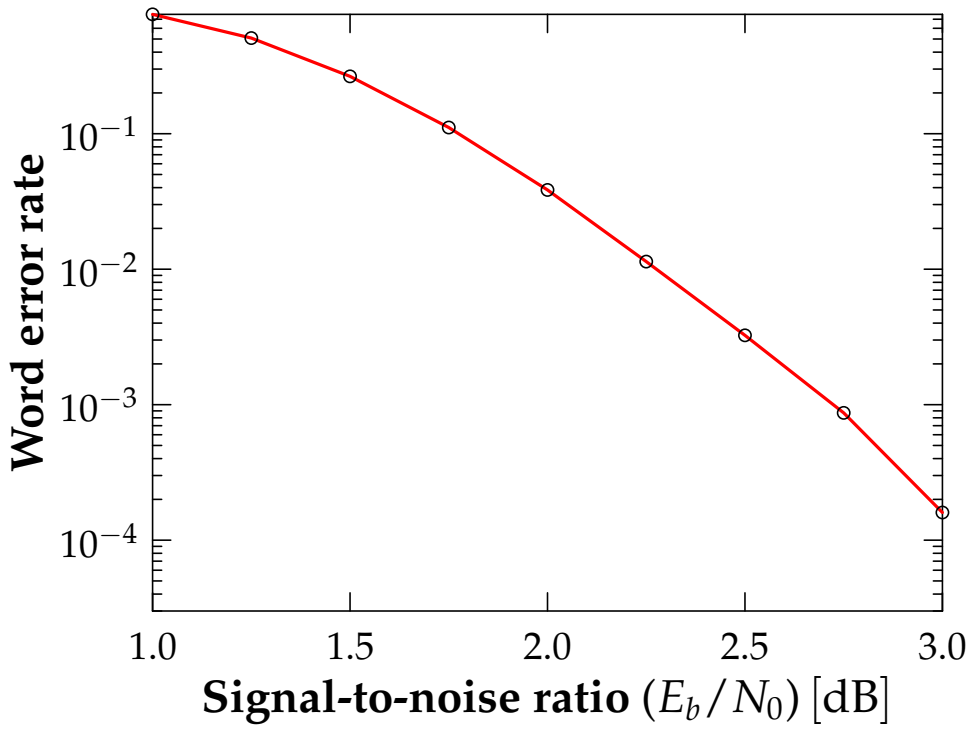
I. Dumer, K. Shabunov, Soft-decision decoding of Reed-Muller codes: recursive lists, *IEEE Trans. on Information Theory*, **52**, pp. 1260–1266, 2006.

Our contribution

- In a naive implementation, the time would be $O(L \cdot n^2)$.
- We show that this can be done in $O(L \cdot n \log n)$ time and $O(L \cdot n)$ space.

We will return to the complexity issue later. For now, let's see how decoding performance is affected.

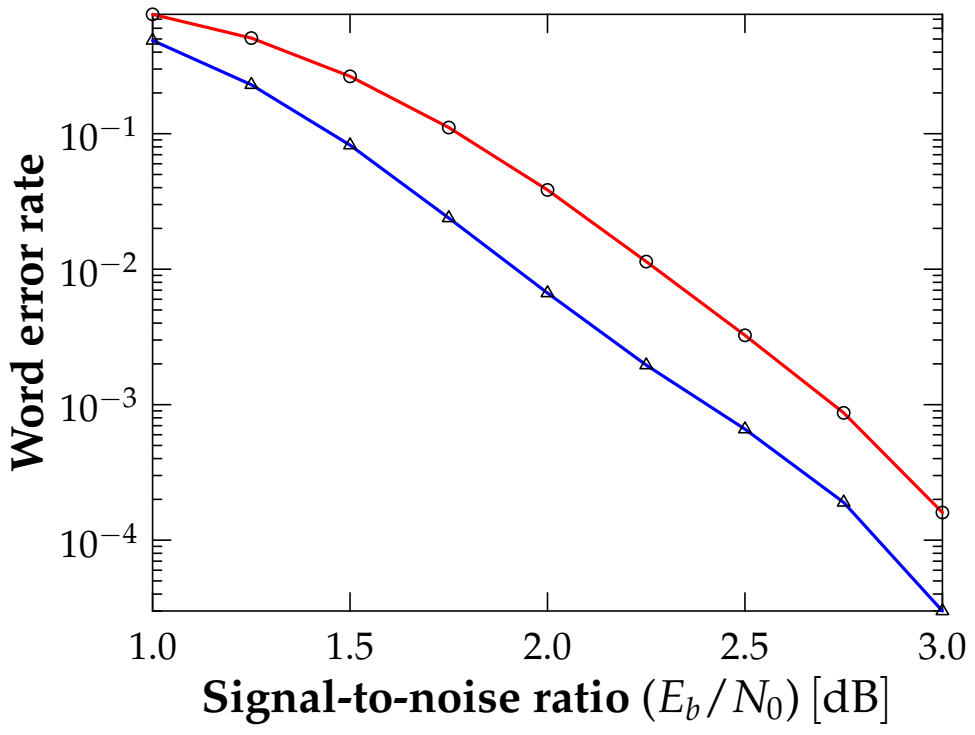
Approaching ML performance



Legend:

$n = 2048$, $L = 1$

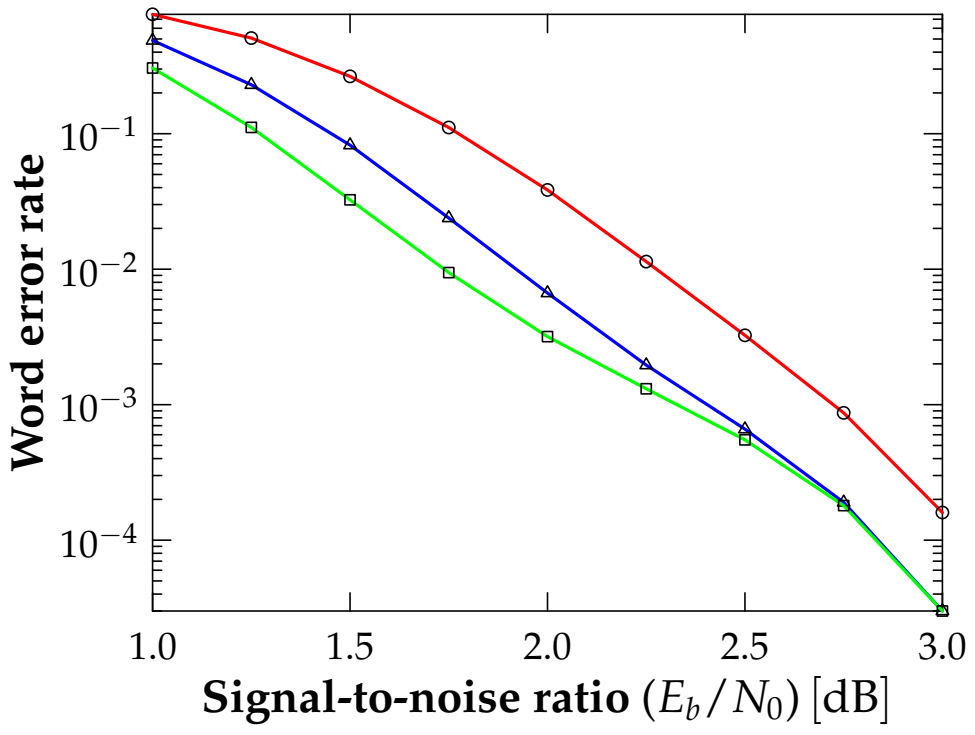
Approaching ML performance



Legend:

- $n = 2048$, $L = 1$
- $n = 2048$, $L = 2$

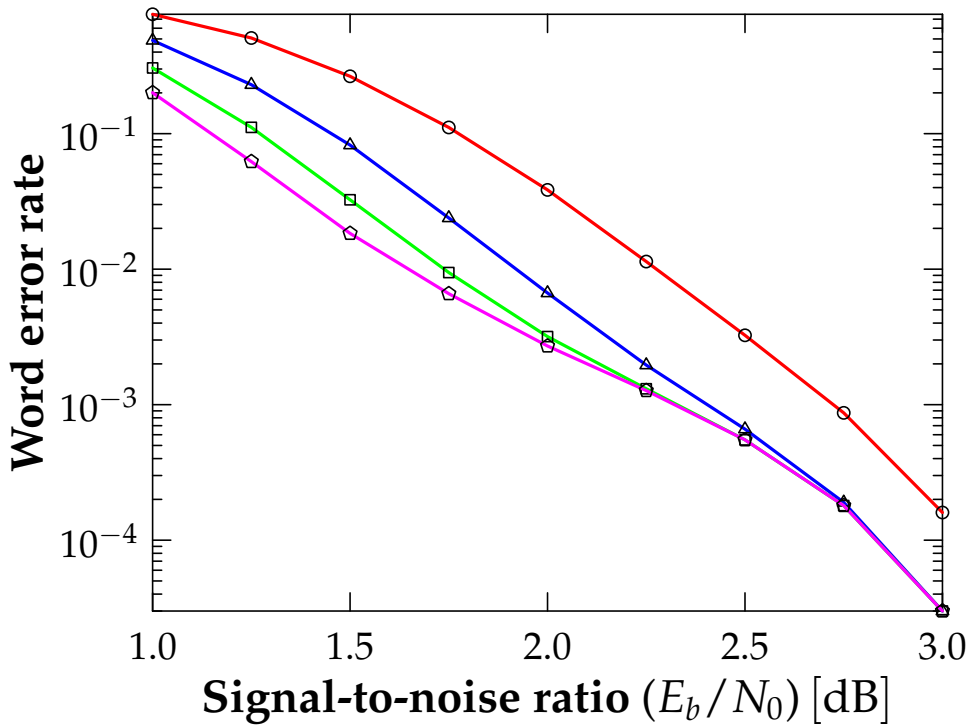
Approaching ML performance



Legend:

- $n = 2048$, $L = 1$
- $n = 2048$, $L = 2$
- $n = 2048$, $L = 4$

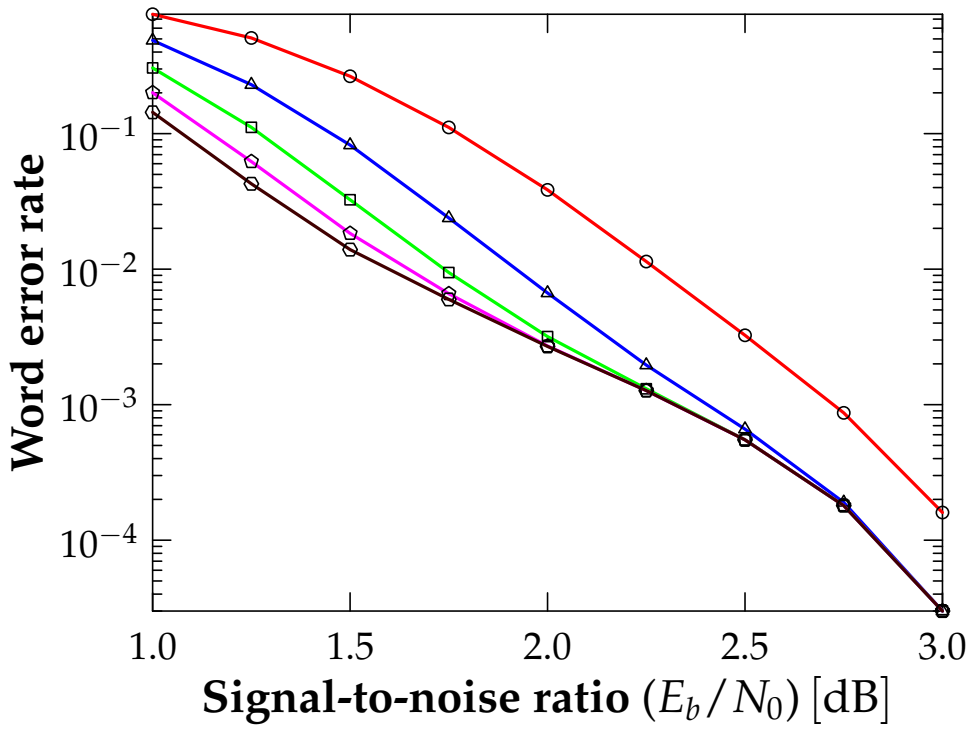
Approaching ML performance



Legend:

- $n = 2048$, $L = 1$
- $n = 2048$, $L = 2$
- $n = 2048$, $L = 4$
- $n = 2048$, $L = 8$

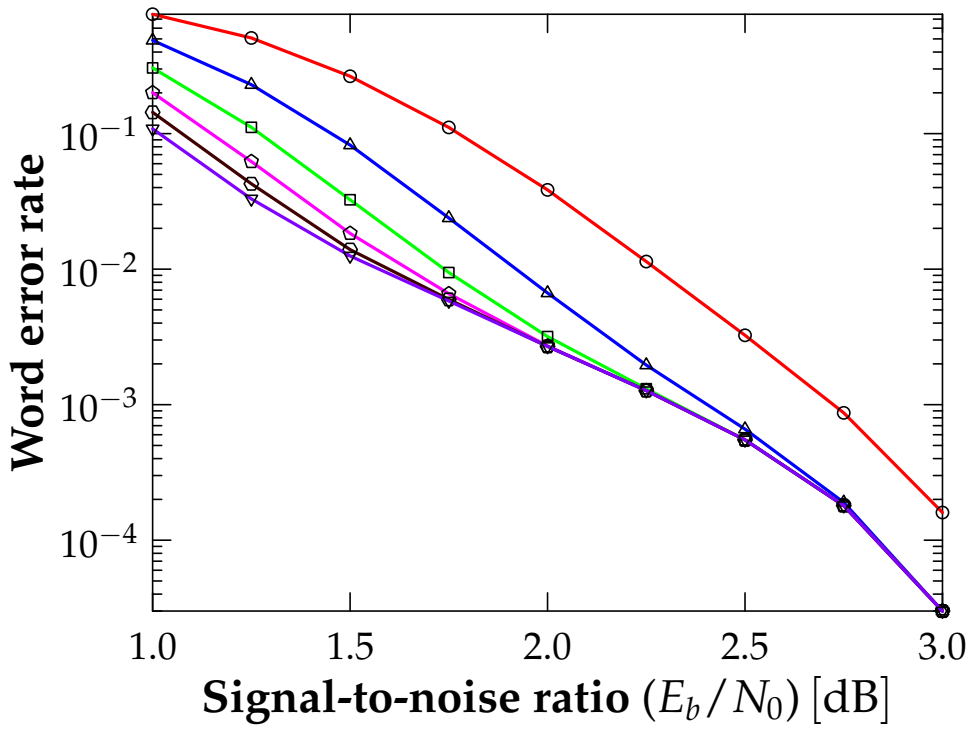
Approaching ML performance



Legend:

- $n = 2048$, $L = 1$
- $n = 2048$, $L = 2$
- $n = 2048$, $L = 4$
- $n = 2048$, $L = 8$
- $n = 2048$, $L = 16$

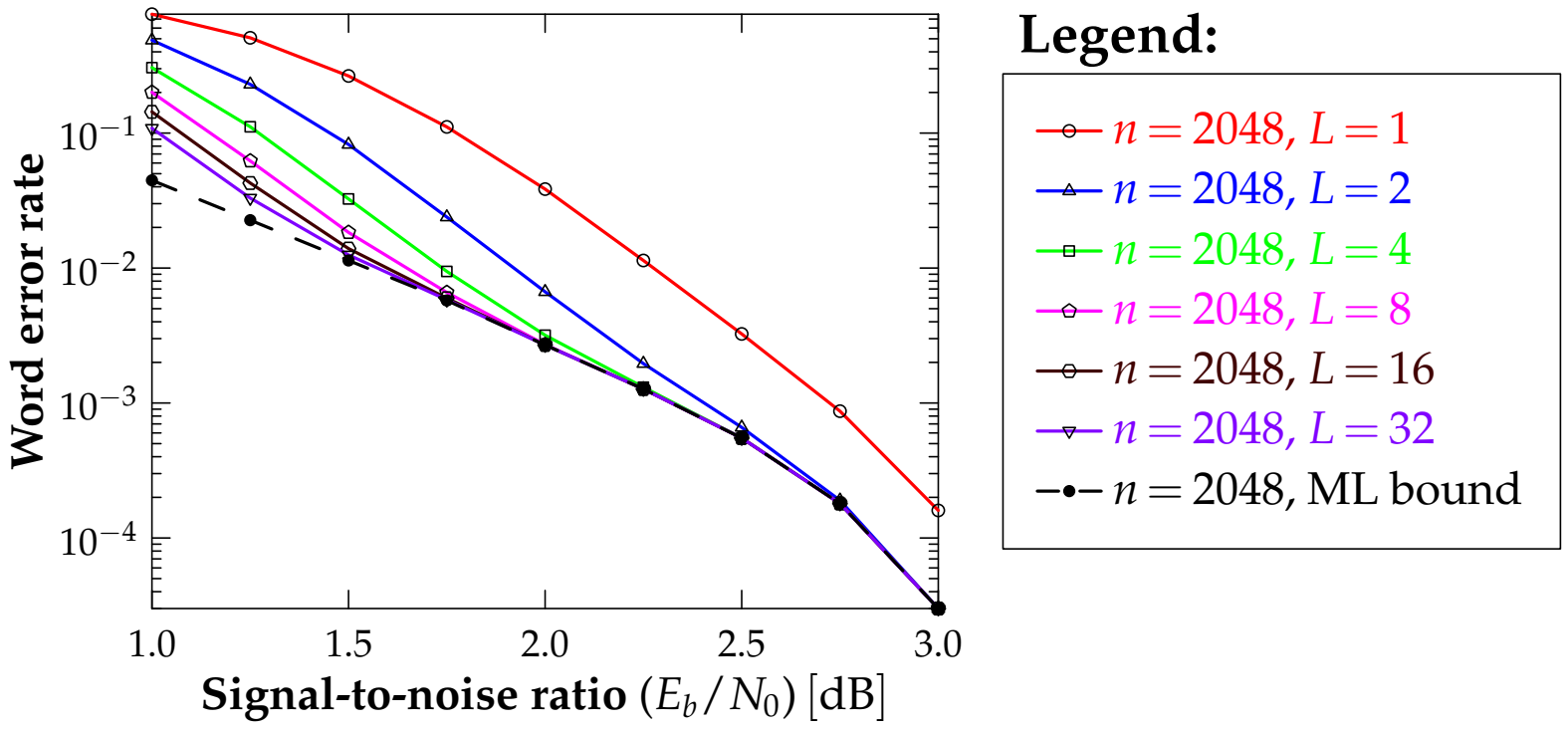
Approaching ML performance



Legend:

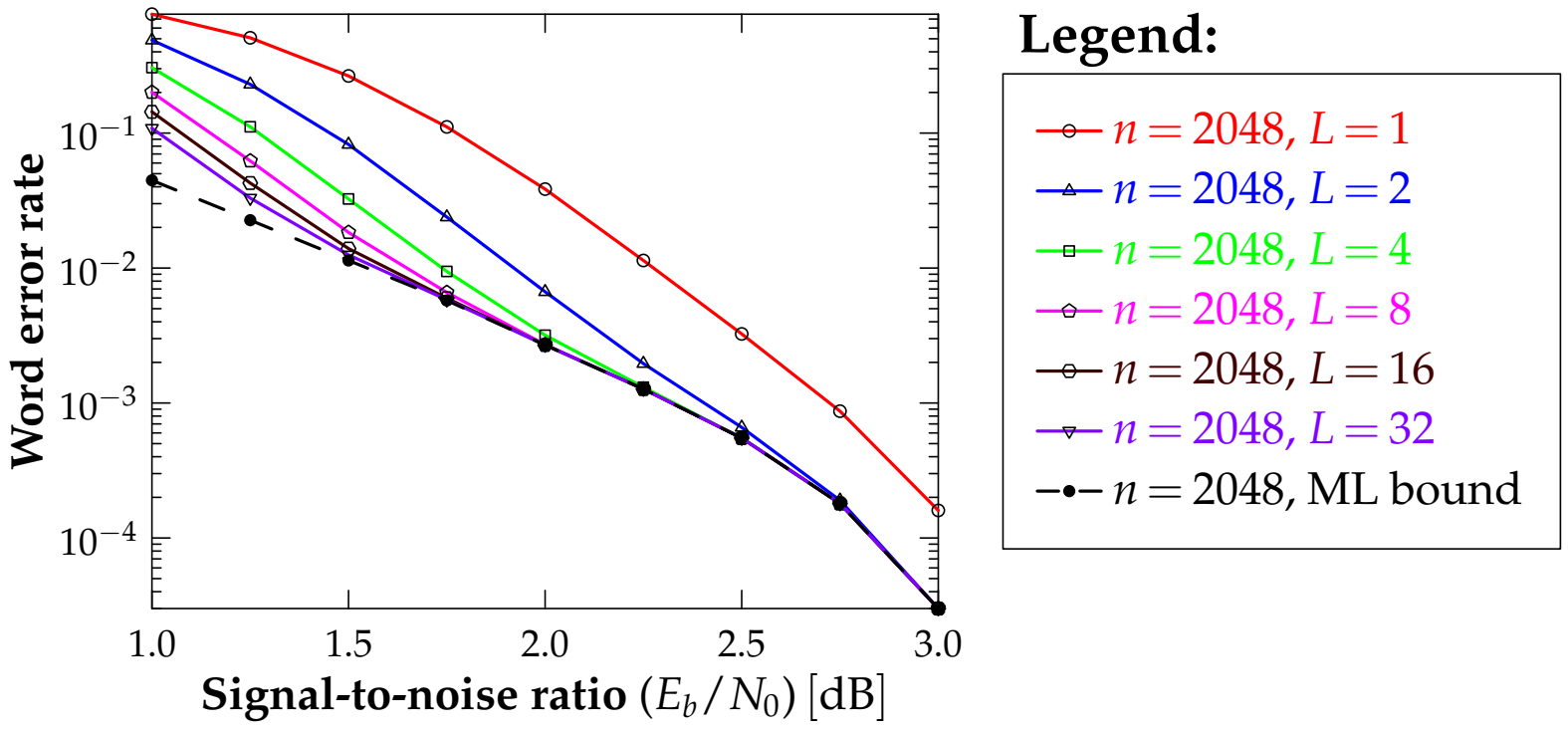
- $n = 2048$, $L = 1$
- $n = 2048$, $L = 2$
- $n = 2048$, $L = 4$
- $n = 2048$, $L = 8$
- $n = 2048$, $L = 16$
- $n = 2048$, $L = 32$

Approaching ML performance



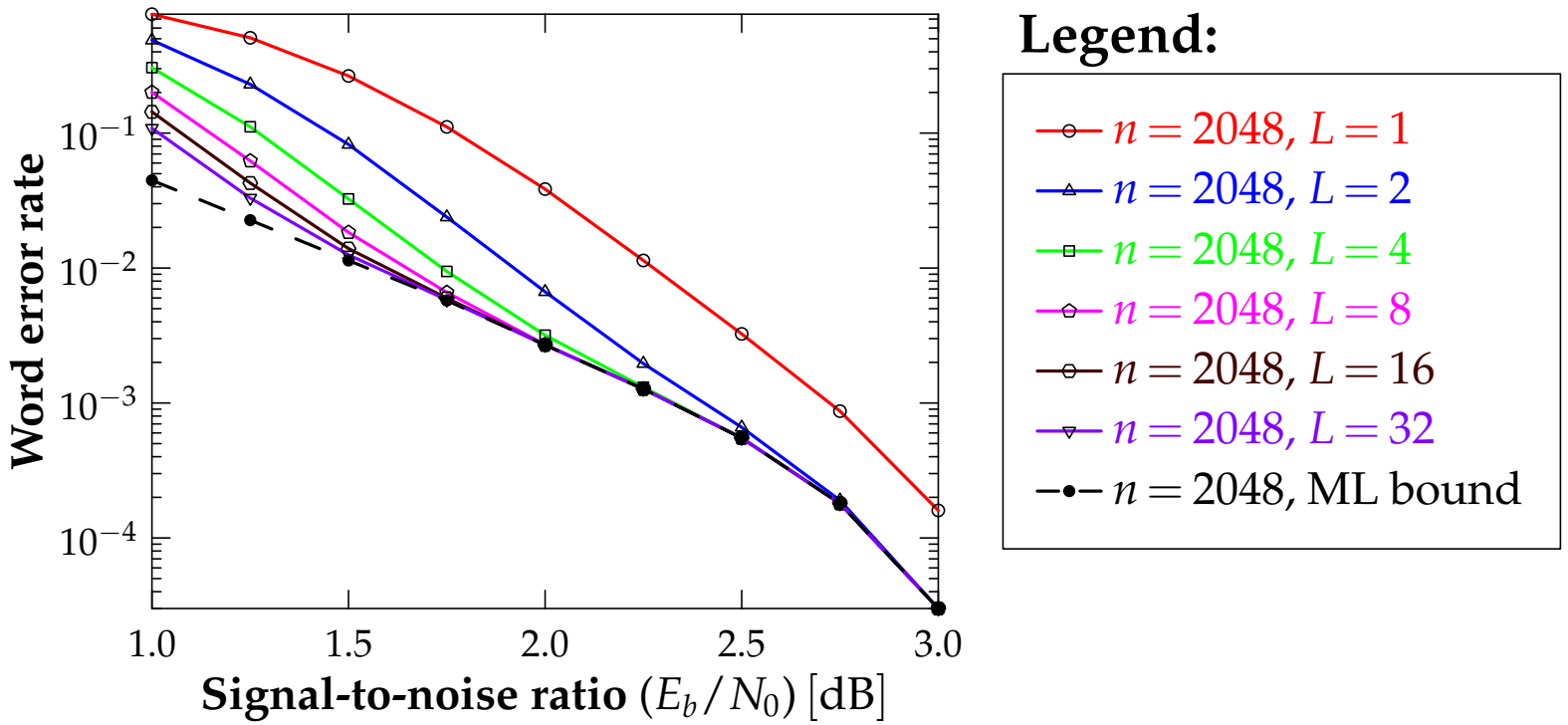
- List-decoding performance quickly approaches that of maximum-likelihood decoding as a function of list-size.

Approaching ML performance



- List-decoding performance quickly approaches that of maximum-likelihood decoding as a function of list-size.
- Good: our decoder is essentially optimal.
- Bad: Still not competitive with LDPC…

Approaching ML performance



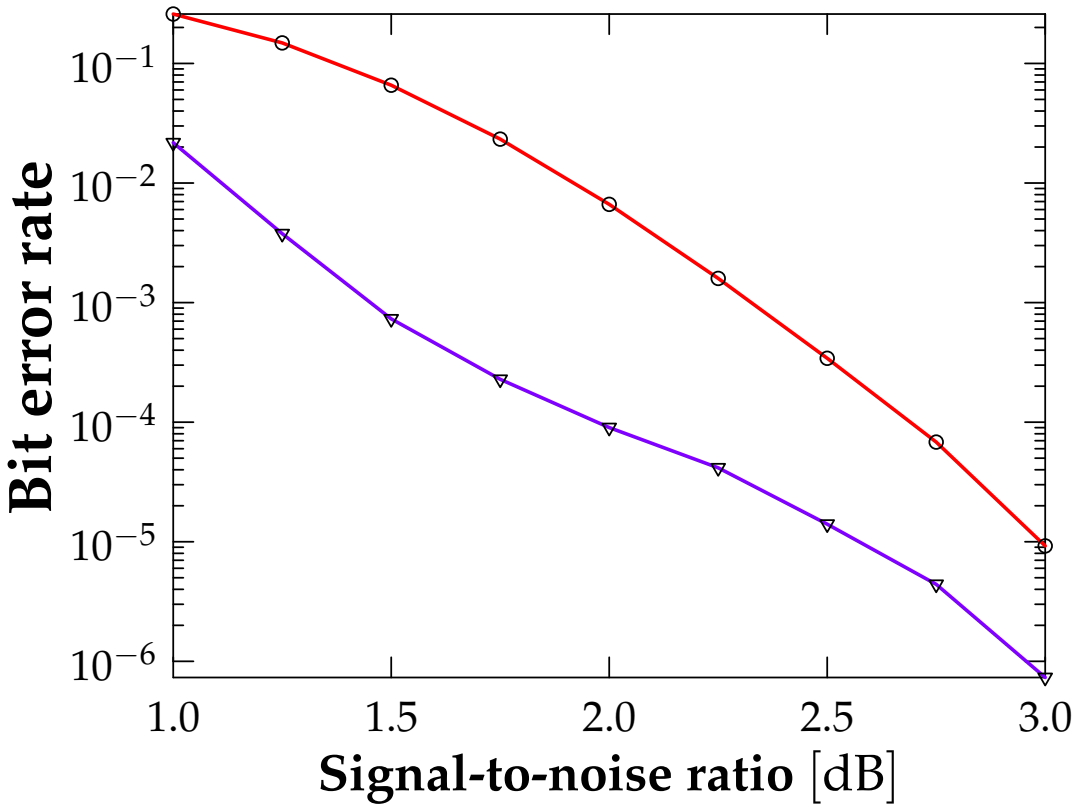
- List-decoding performance quickly approaches that of maximum-likelihood decoding as a function of list-size.
- Good: our decoder is essentially optimal.
- Bad: Still not competitive with LDPC...
- Conclusions: Must somehow "fix" the polar code.

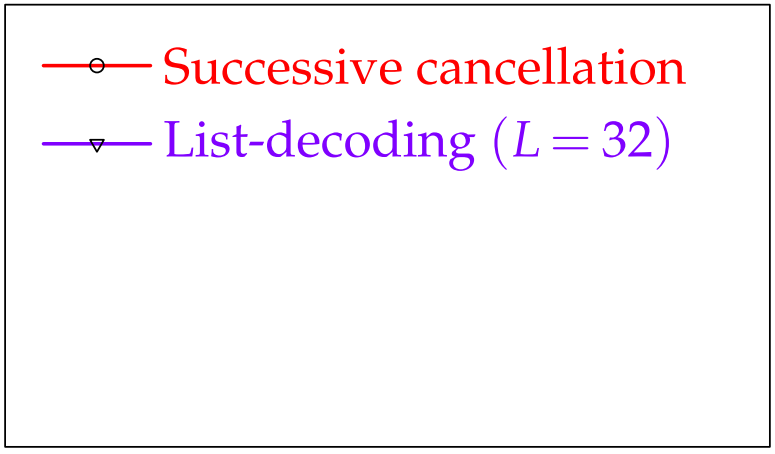
A simple concatenation scheme

- Recall that the last step of decoding was "pick the most likely codeword from the list".
- An error: the transmitted codeword is not the most likely codeword in the list.
- However, very often, the transmitted codeword is still a member of the list.
- We need a "genie" to single-out the transmitted codeword.
- Idea: Let there be $k + r$ unfrozen bits. Of these,
  - Use the first k bits to encode information.
  - Use the last r unfrozen bits to encode the CRC value of the first k bits.
  - Pick the most probable codeword on the list with correct CRC.

Approaching LDPC performance

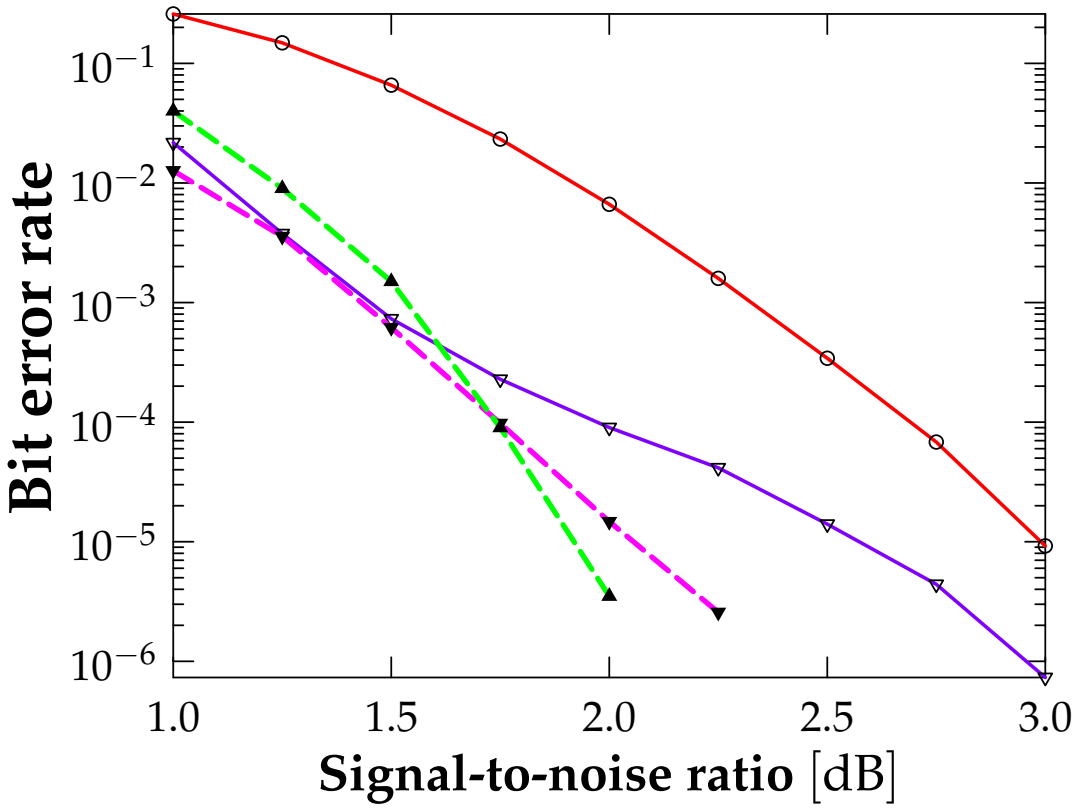
Simulation results for a polar code of length $n = 2048$ and rate $R = 0.5$, optimized for a BPSK-AWGN channel with $E_b/N_0 = 2.0\,\mathrm{dB}$.

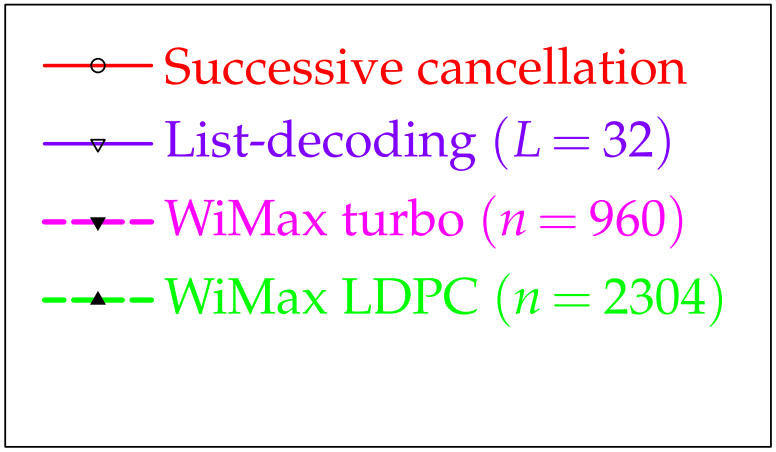




Approaching LDPC performance

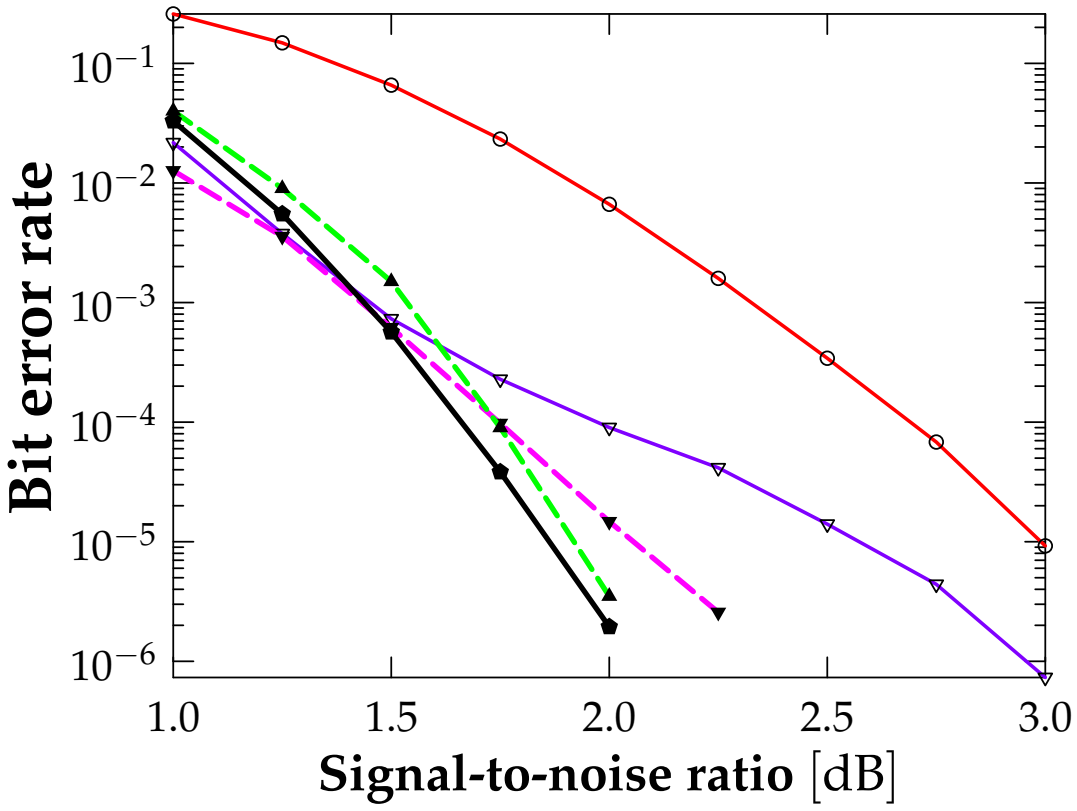
Simulation results for a polar code of length $n = 2048$ and rate $R = 0.5$, optimized for a BPSK-AWGN channel with $E_b/N_0 = 2.0\,\mathrm{dB}$.

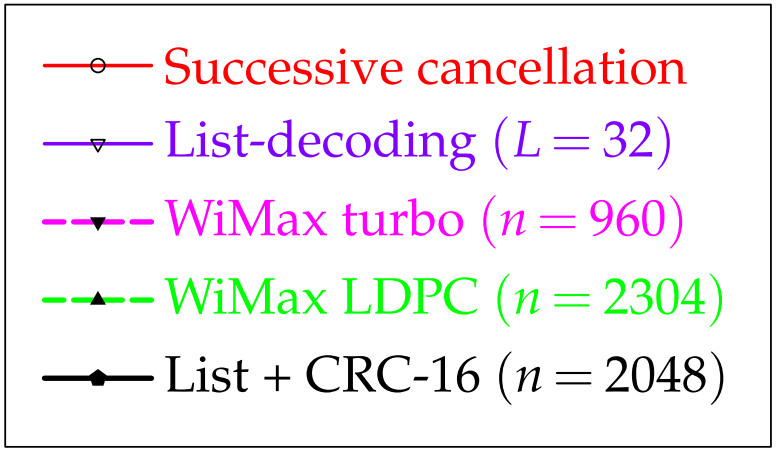




Approaching LDPC performance

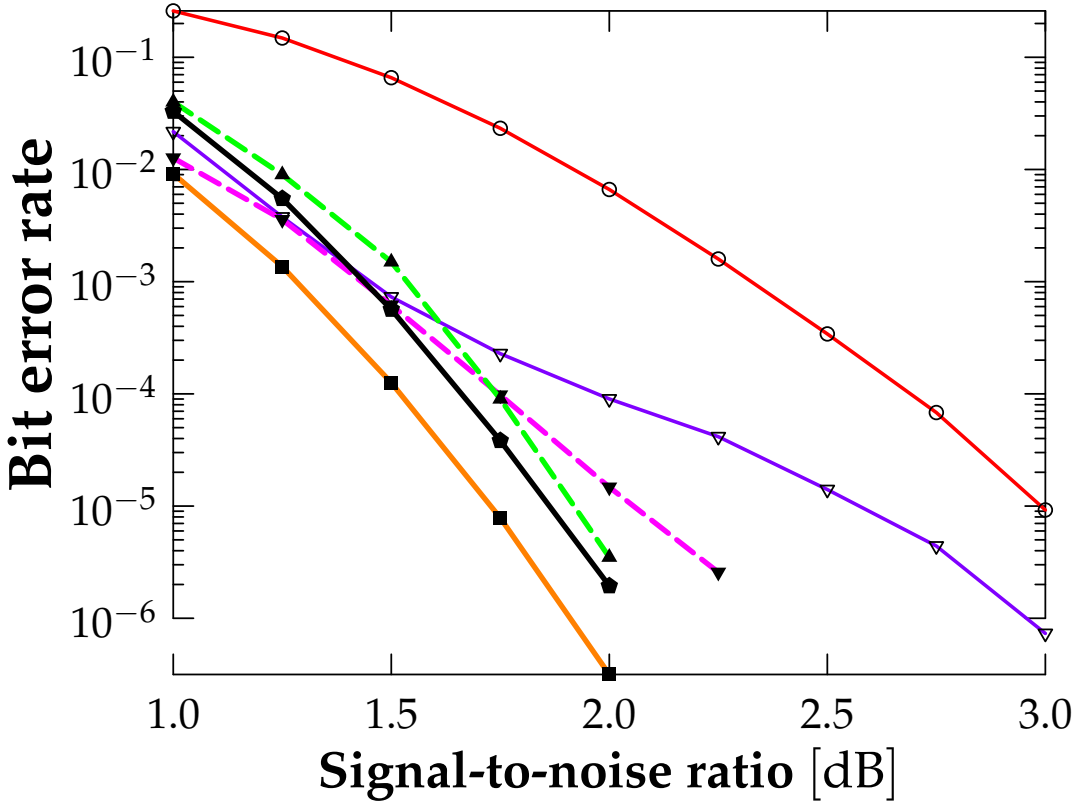
Simulation results for a polar code of length $n = 2048$ and rate $R = 0.5$, optimized for a BPSK-AWGN channel with $E_b/N_0 = 2.0\,\mathrm{dB}$.

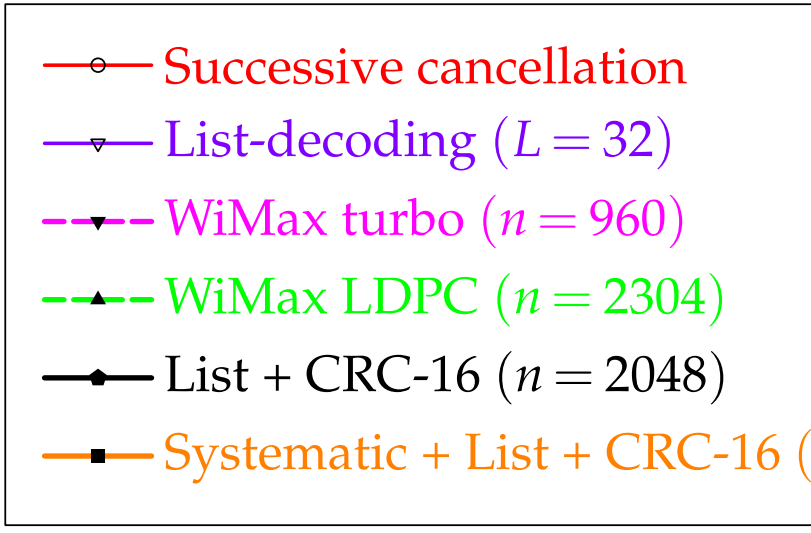




Approaching LDPC performance

Simulation results for a polar code of length $n = 2048$ and rate $R = 0.5$, optimized for a BPSK-AWGN channel with $E_b/N_0 = 2.0\,\mathrm{dB}$.



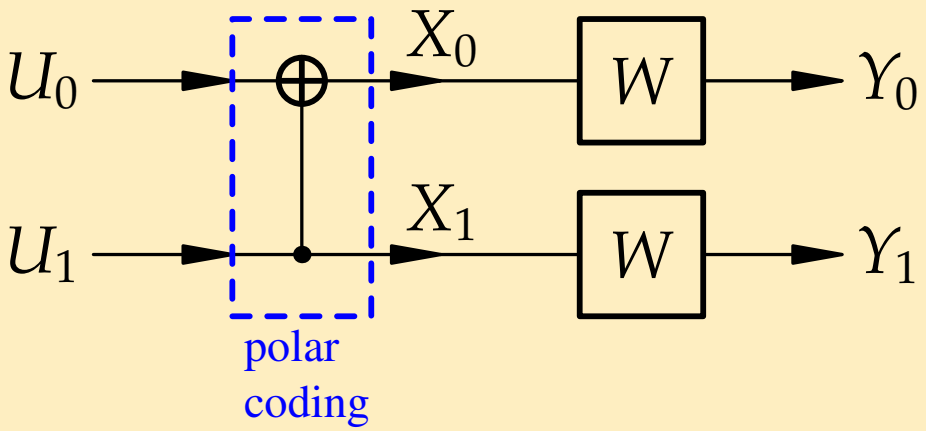


Quadratic complexity of list decoding

Naive implementation recap

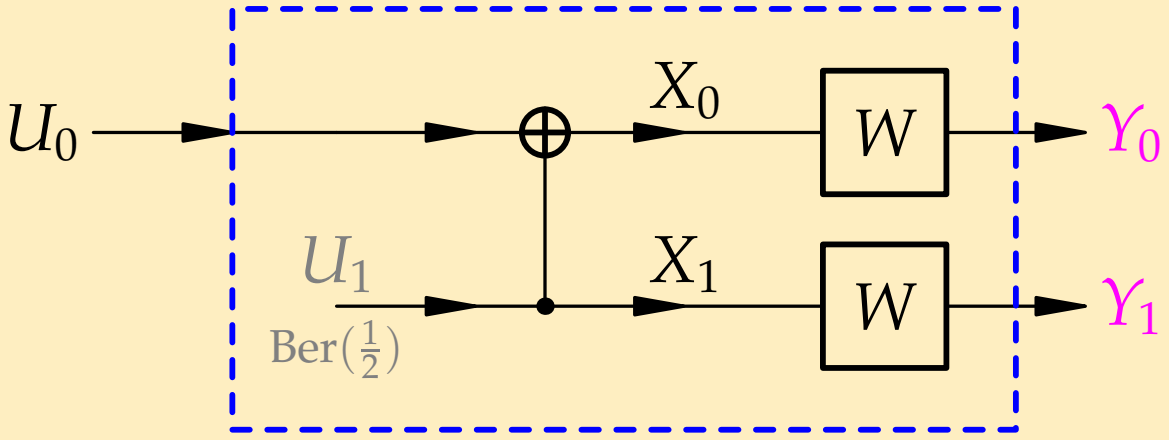
- In a naive implementation, the decoding paths are independent. They don't share information.
- Each decoding path has a set of variables associated with it. For example, at stage i, each decoding path must remember the values of the bits $\widehat{u}_0, \widehat{u}_1, \ldots, \widehat{u}_{i-1}$.
- It turns out (as we shall see) that each decoding path has $\Theta(n)$ memory associated with it.
- When a path is split in two, one decoding path is left with the original variables while the other must be handed a copy of them.
- Each copy operation takes $O(n)$ time.
- Thus, the overall time complexity is $O(L \cdot n^2)$.

A very short introduction to polar codes

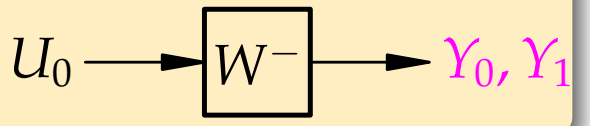


A very short introduction to polar codes

$W^- : \mathcal{X} \to \mathcal{Y}^2$



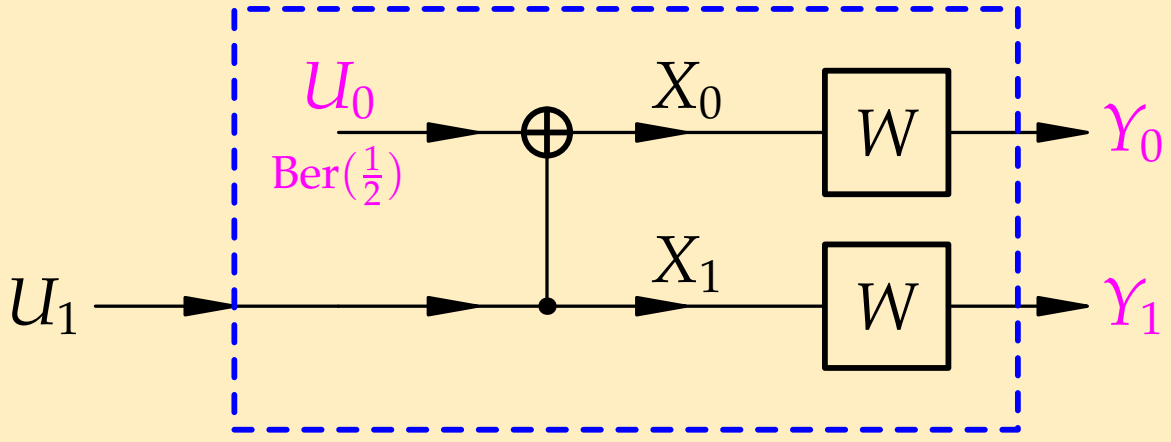
W^-, condensed



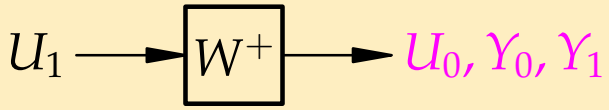
$$W^-(y_0, y_1|u_0) = \sum_{u_1 \in \mathcal{X}} \frac{1}{2} W(y_0|u_0 \oplus u_1) W(y_1|u_1) \ .$$

A very short introduction to polar codes

$W^+ : \mathcal{X} \to \mathcal{X} \times \mathcal{Y}^2$

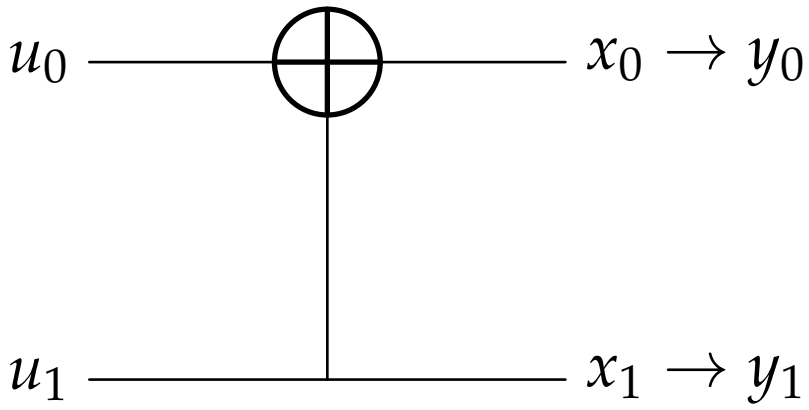


W^+, condensed

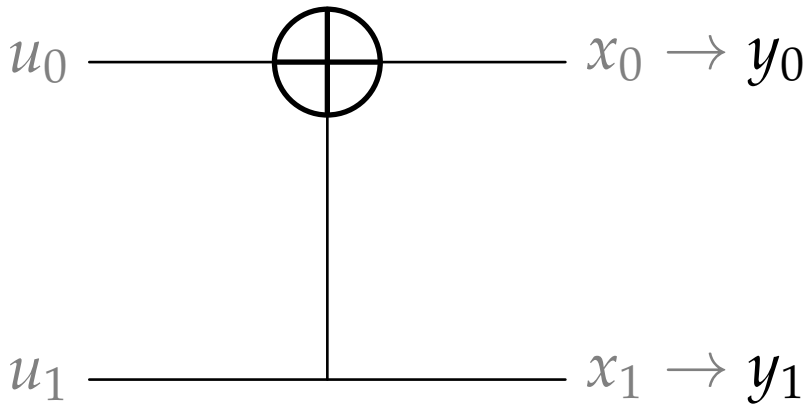


$$W^+(u_0, y_0, y_1 | u_1) = \frac{1}{2} W(y_0 | u_0 \oplus u_1) W(y_1 | u_1) \ .$$

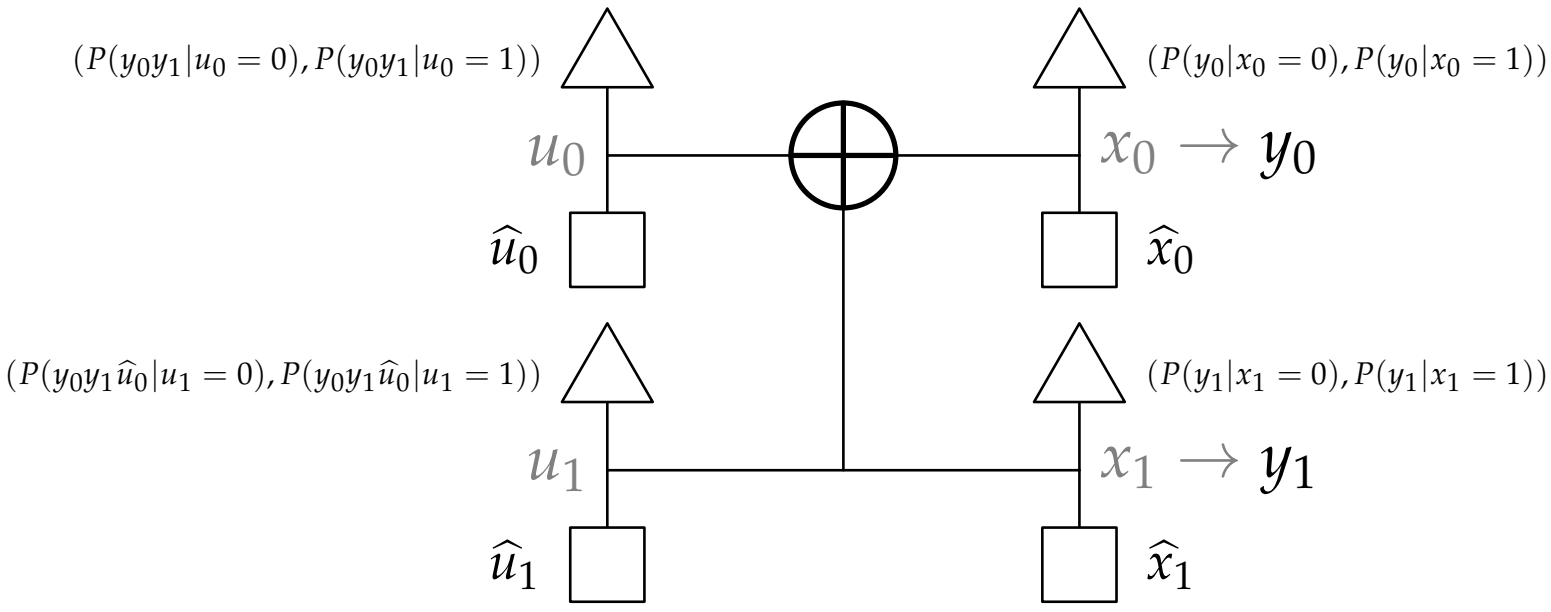
A closer look at successive cancellation

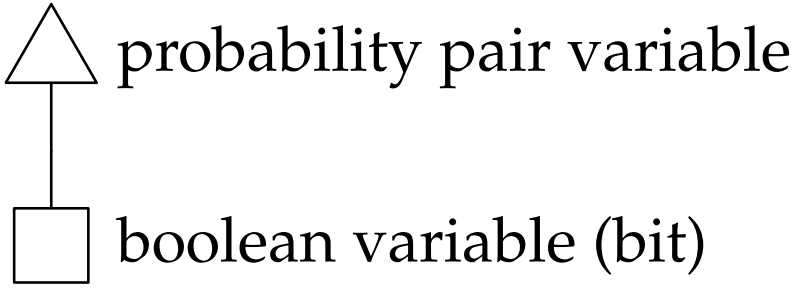


A closer look at successive cancellation

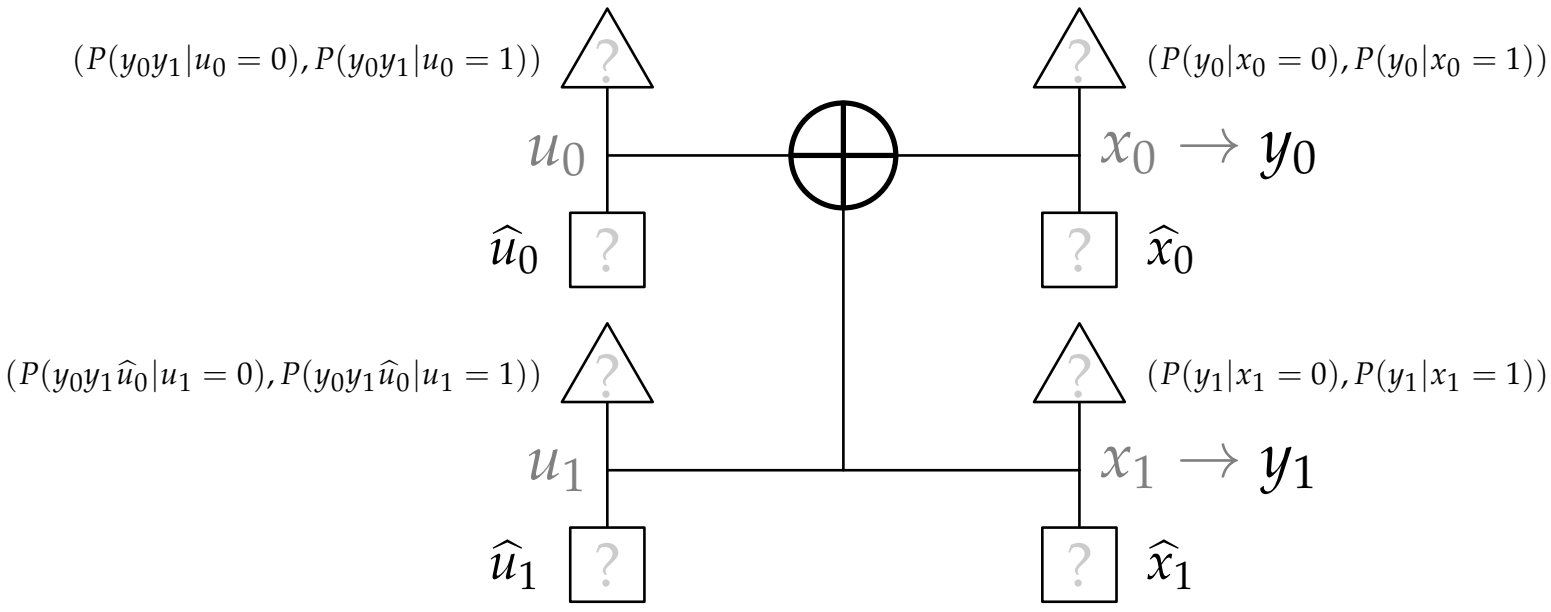


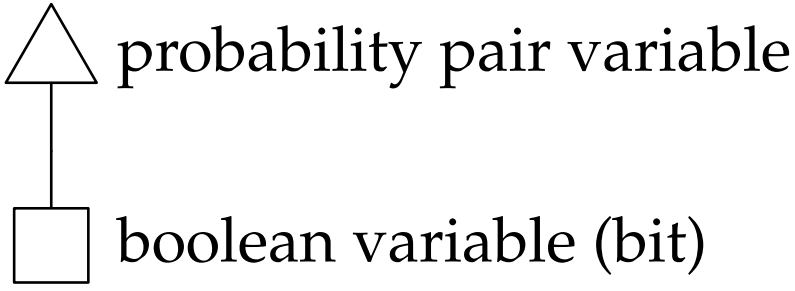
A closer look at successive cancellation



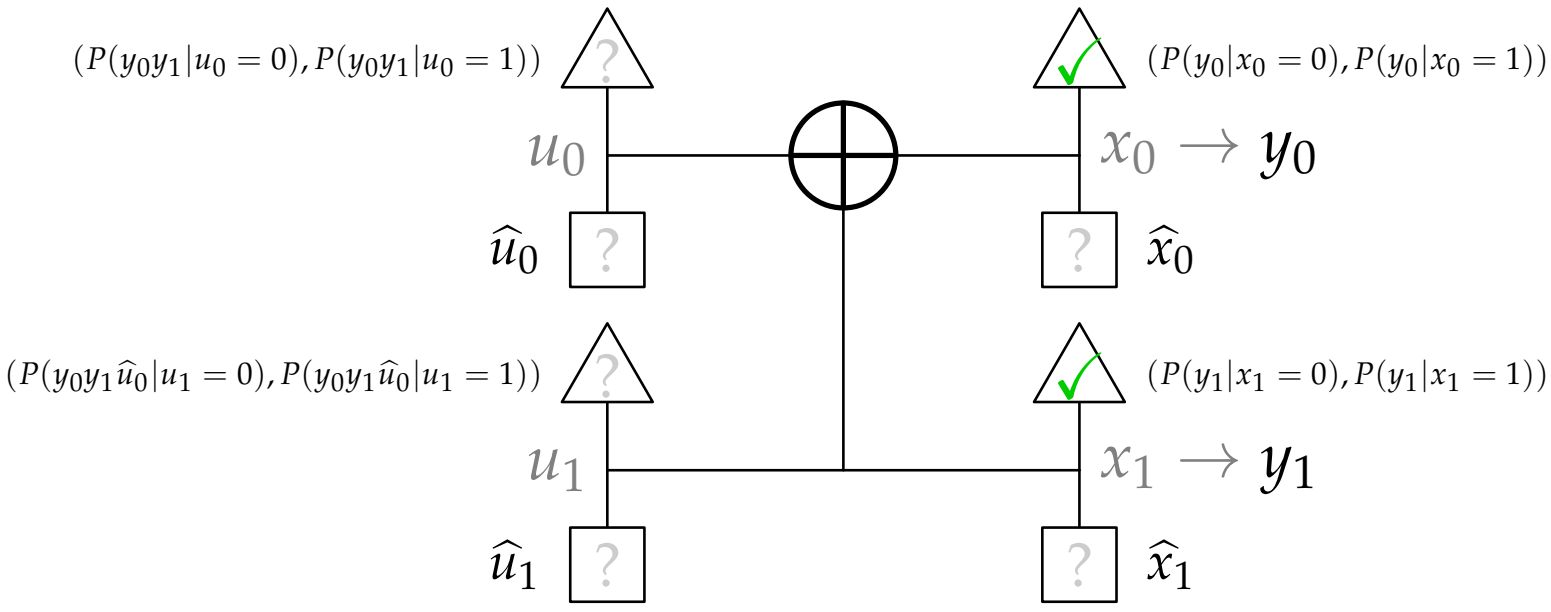


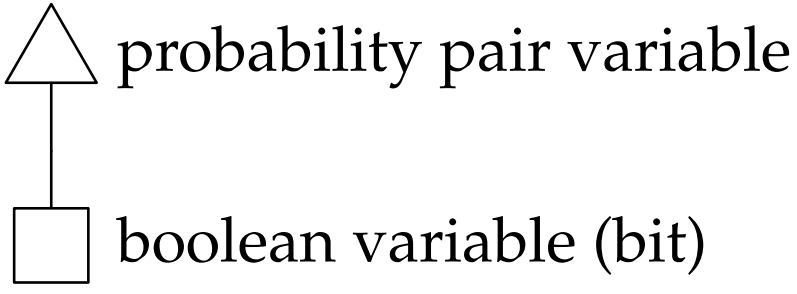
A closer look at successive cancellation



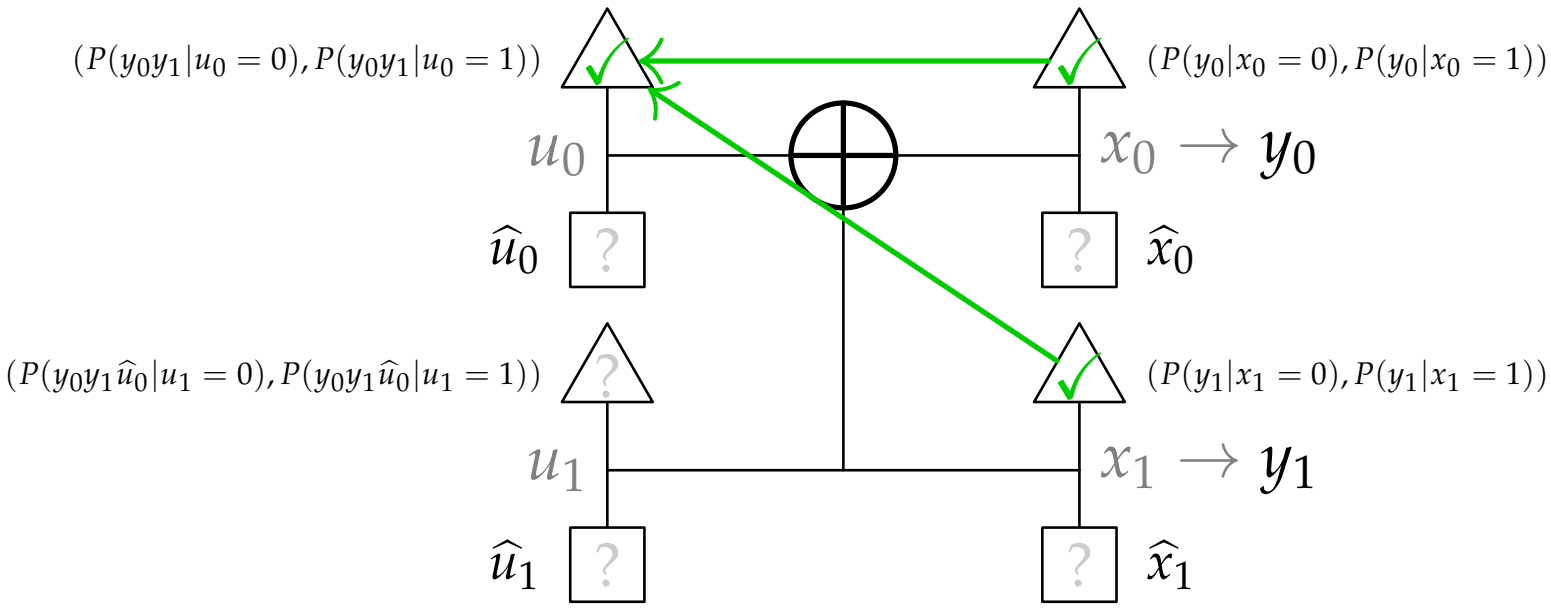


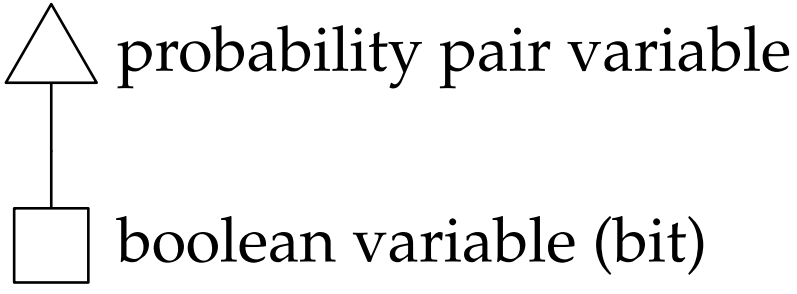
A closer look at successive cancellation



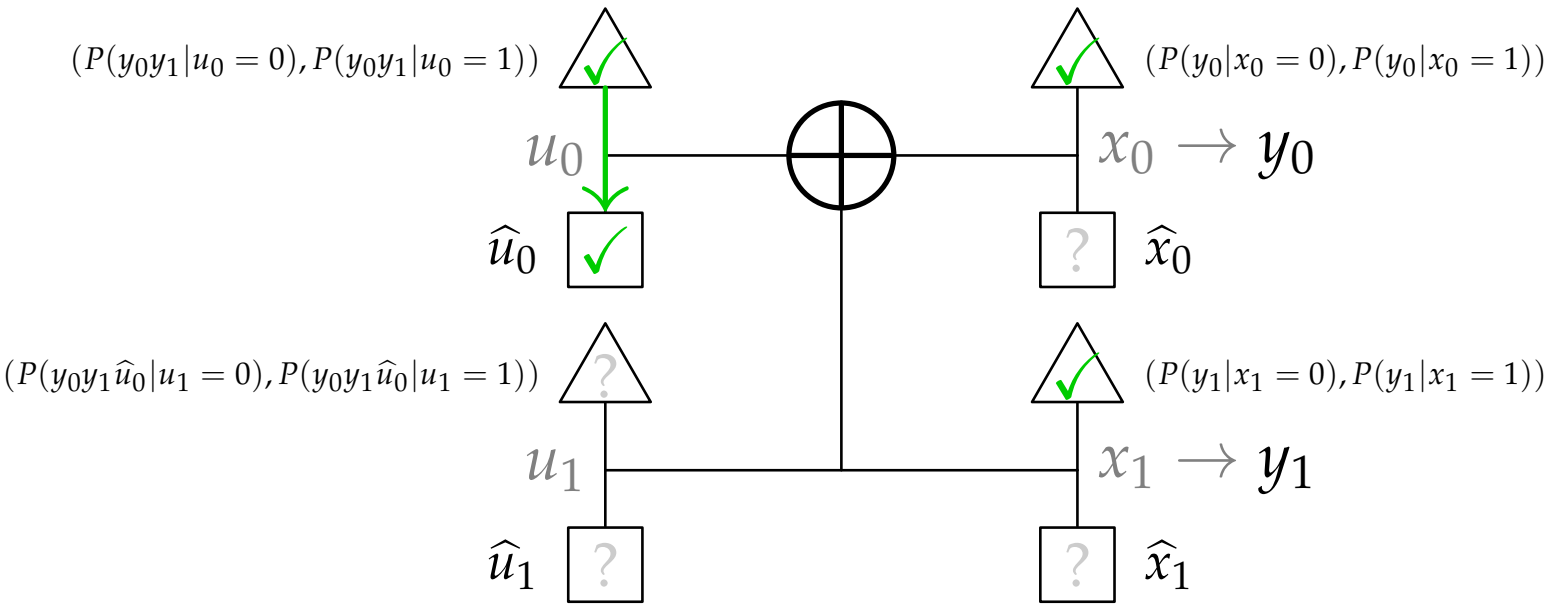


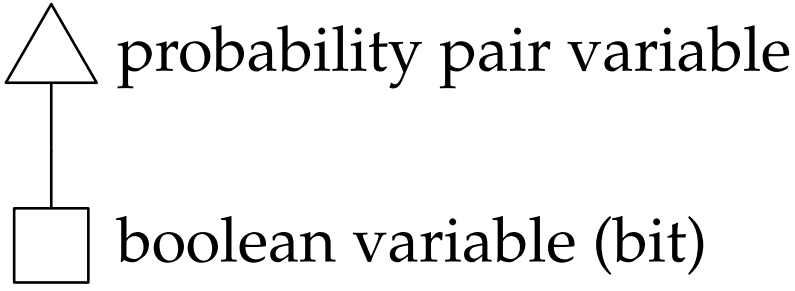
A closer look at successive cancellation



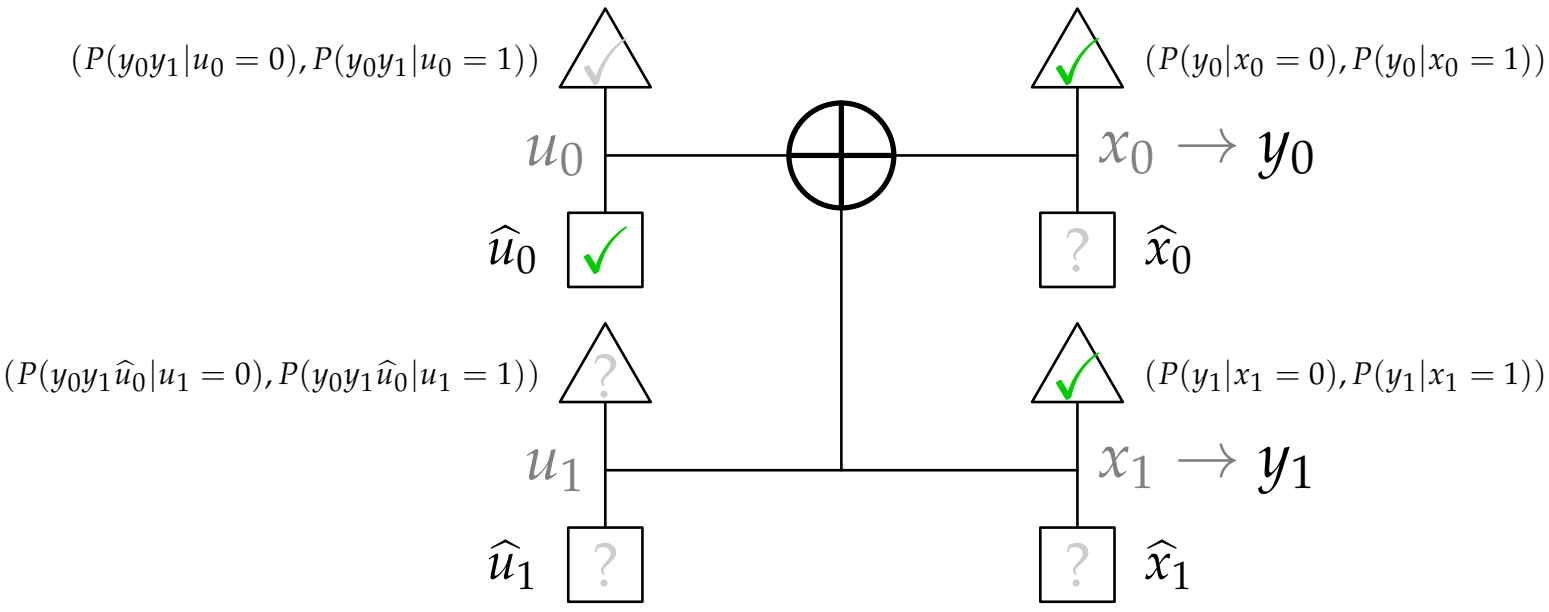


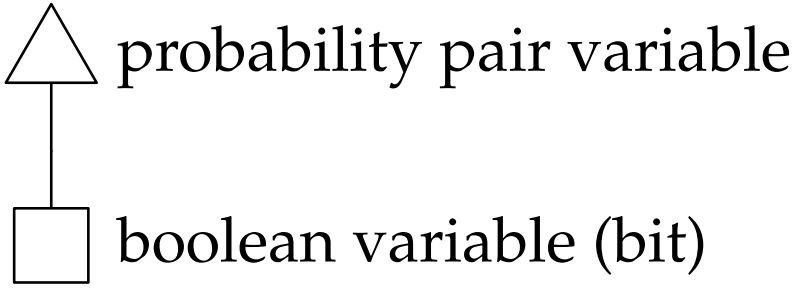
A closer look at successive cancellation



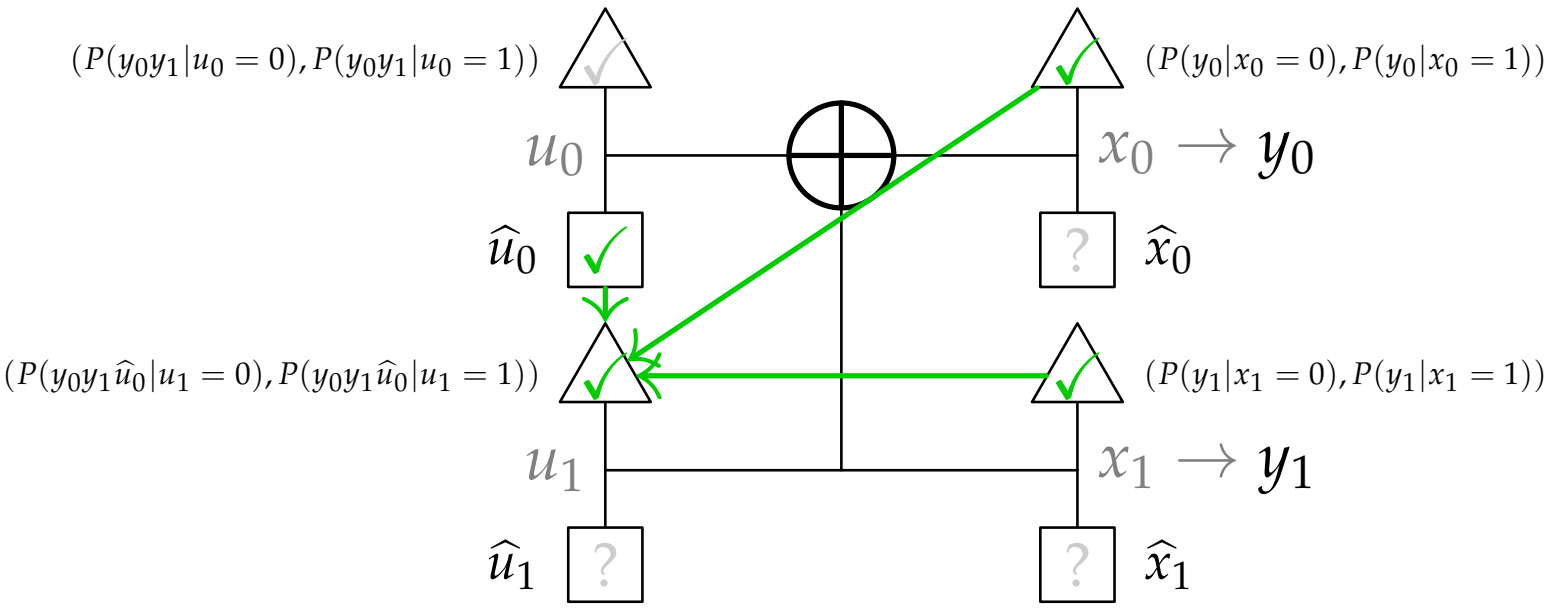


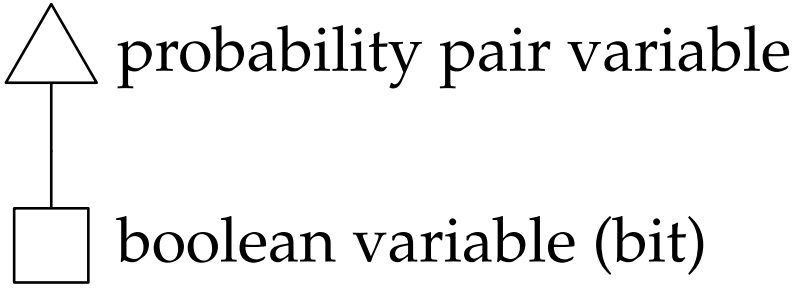
A closer look at successive cancellation



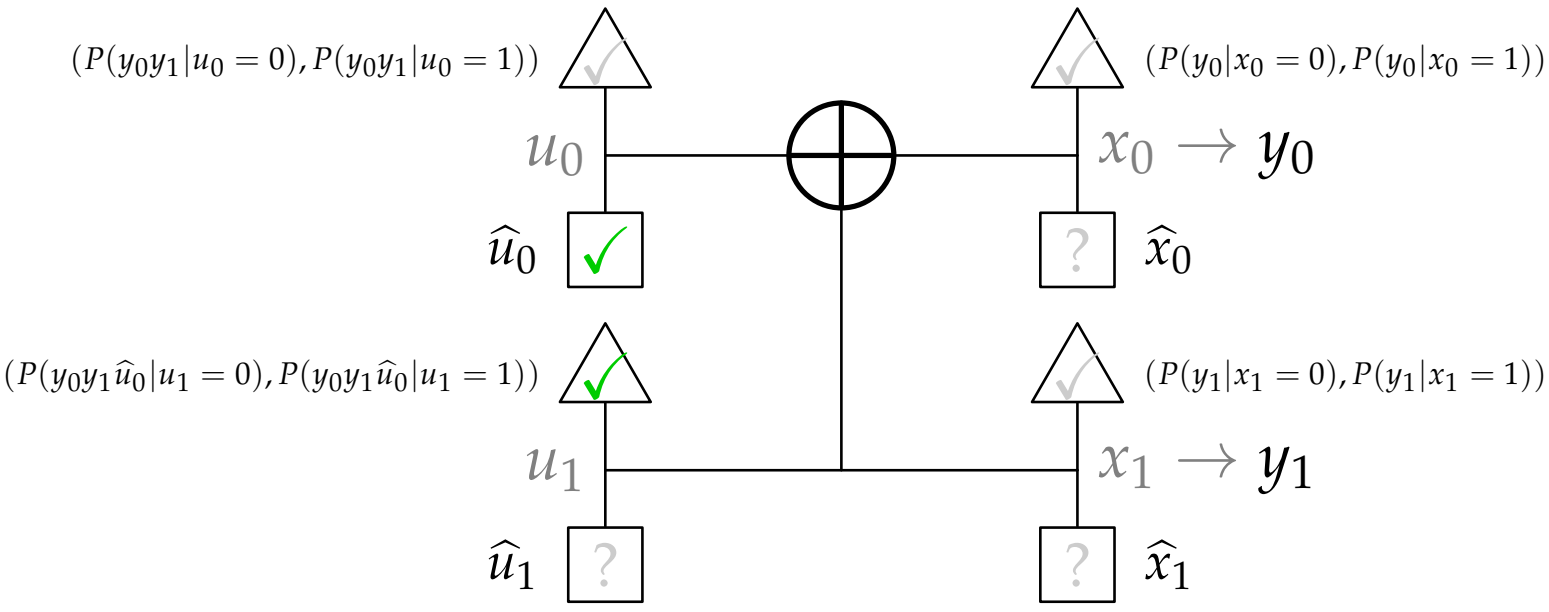


A closer look at successive cancellation





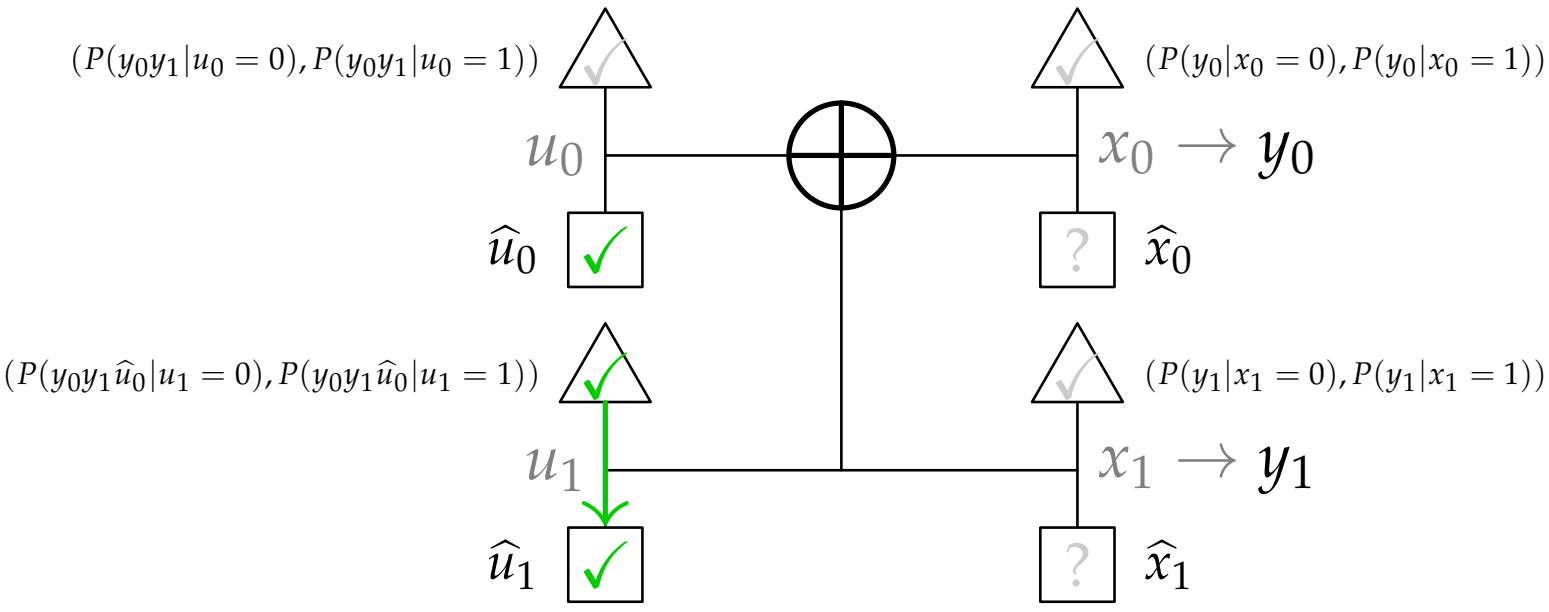
A closer look at successive cancellation

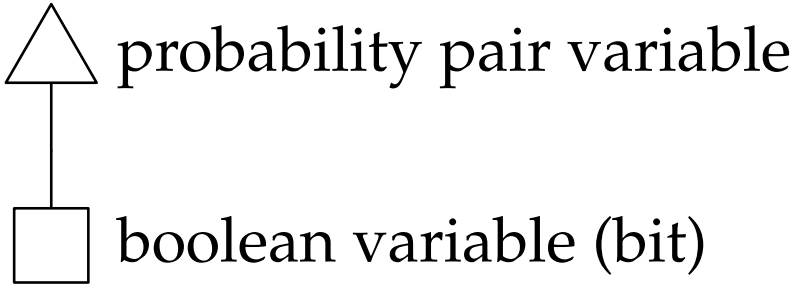


probability pair variable

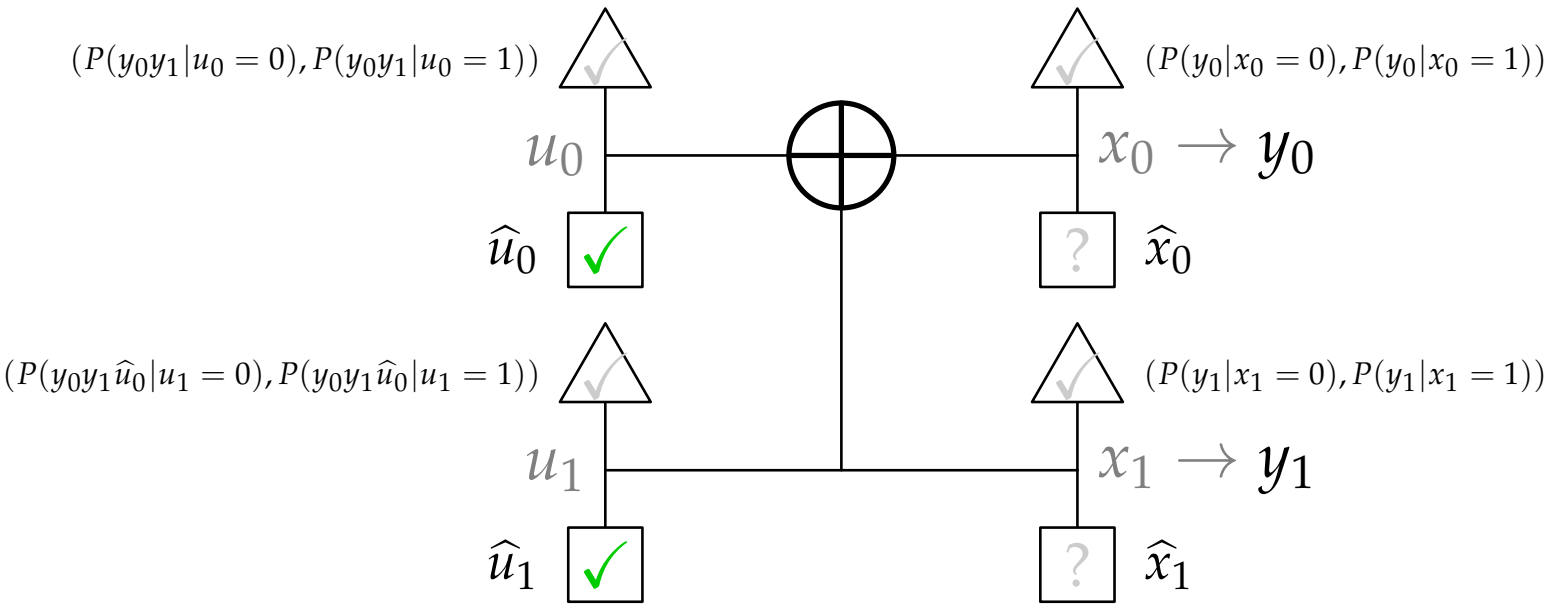
boolean variable (bit)

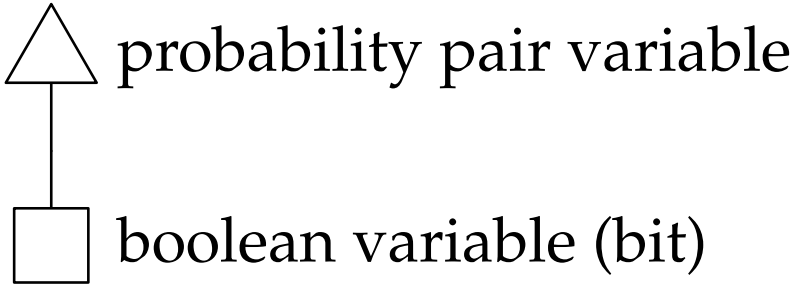
A closer look at successive cancellation



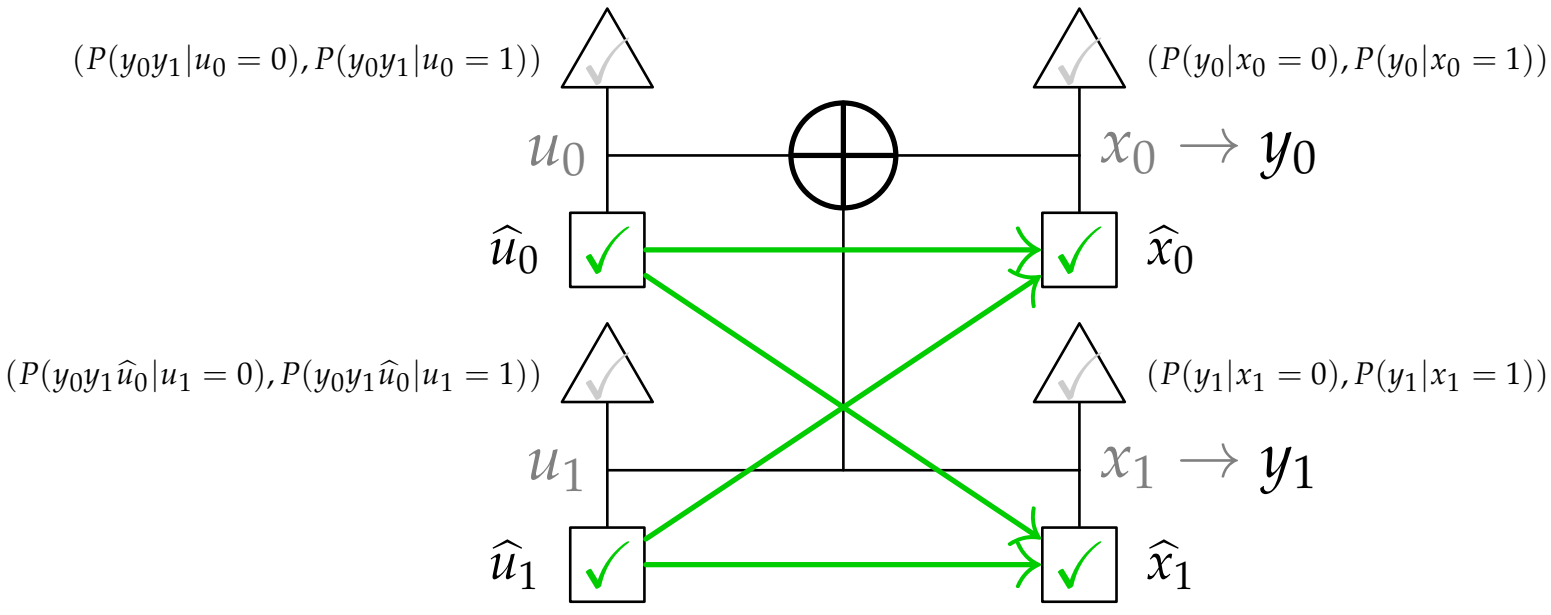


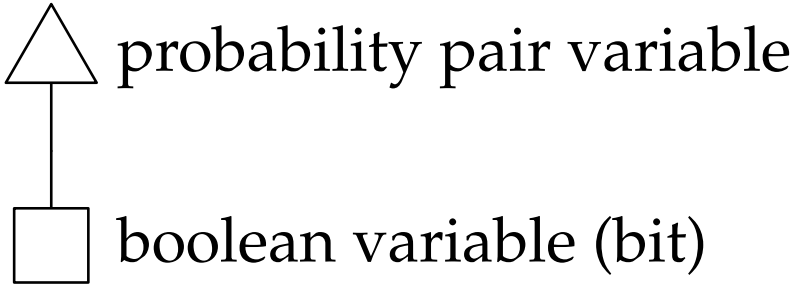
A closer look at successive cancellation



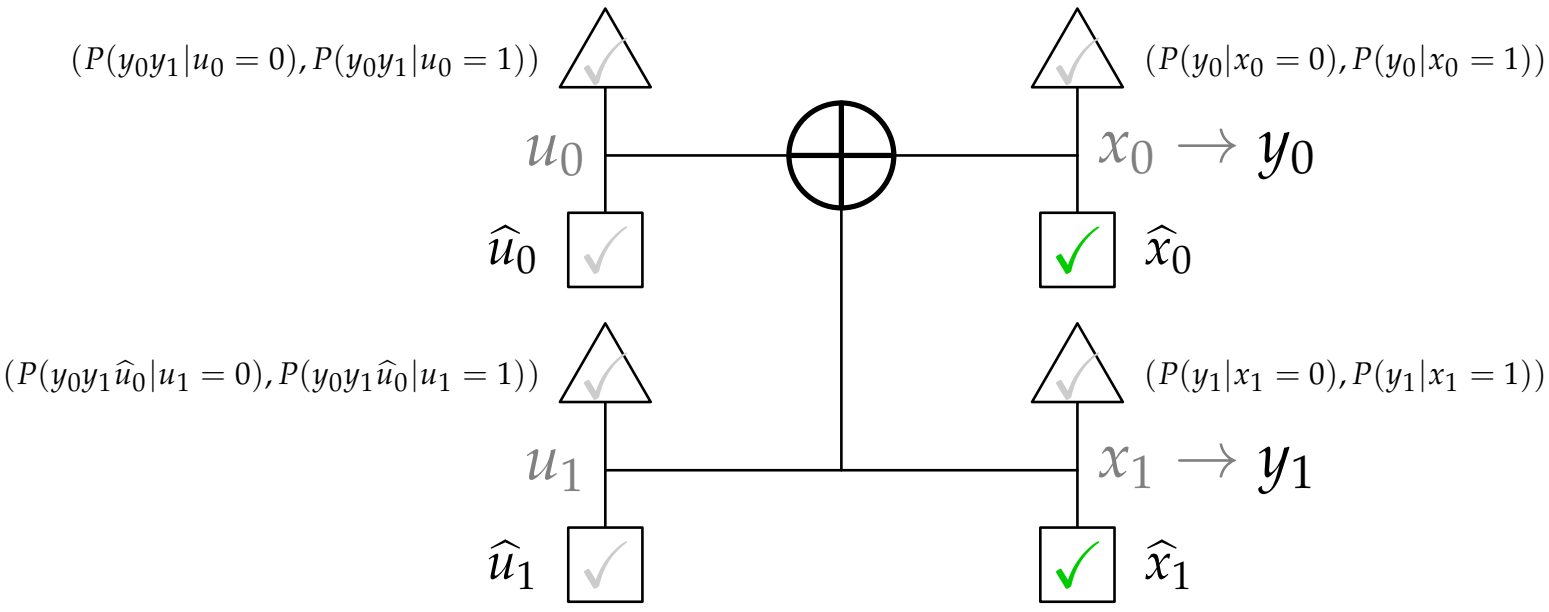


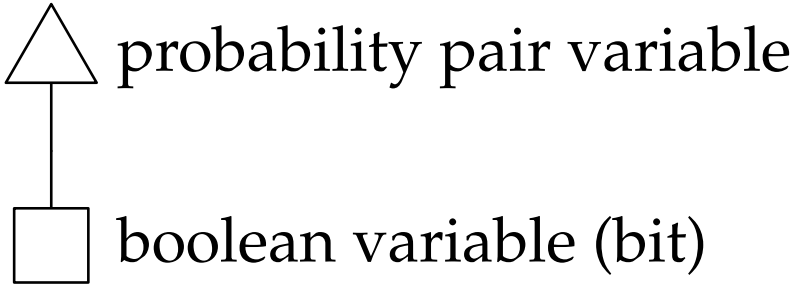
A closer look at successive cancellation





A closer look at successive cancellation

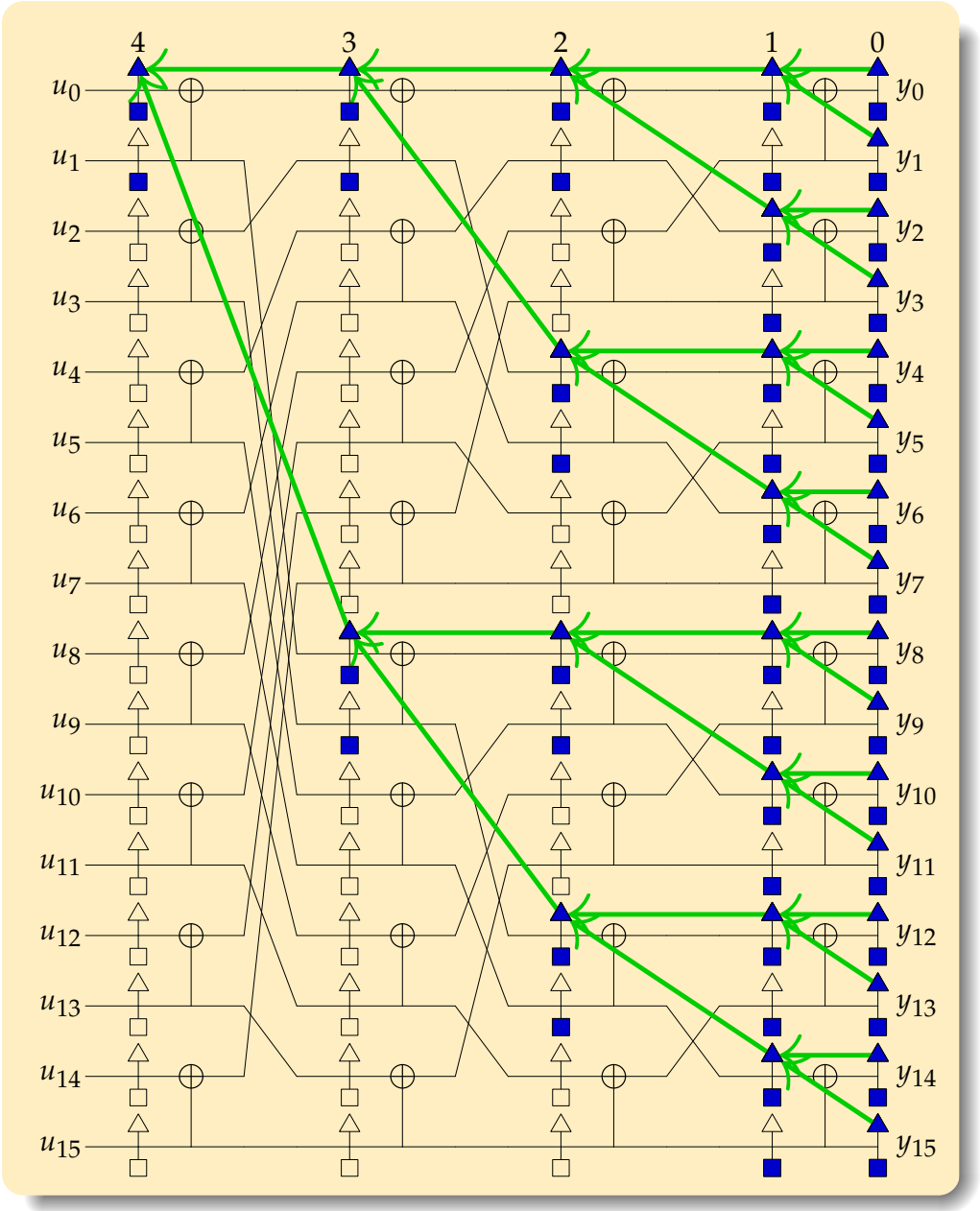




A larger example

Key point

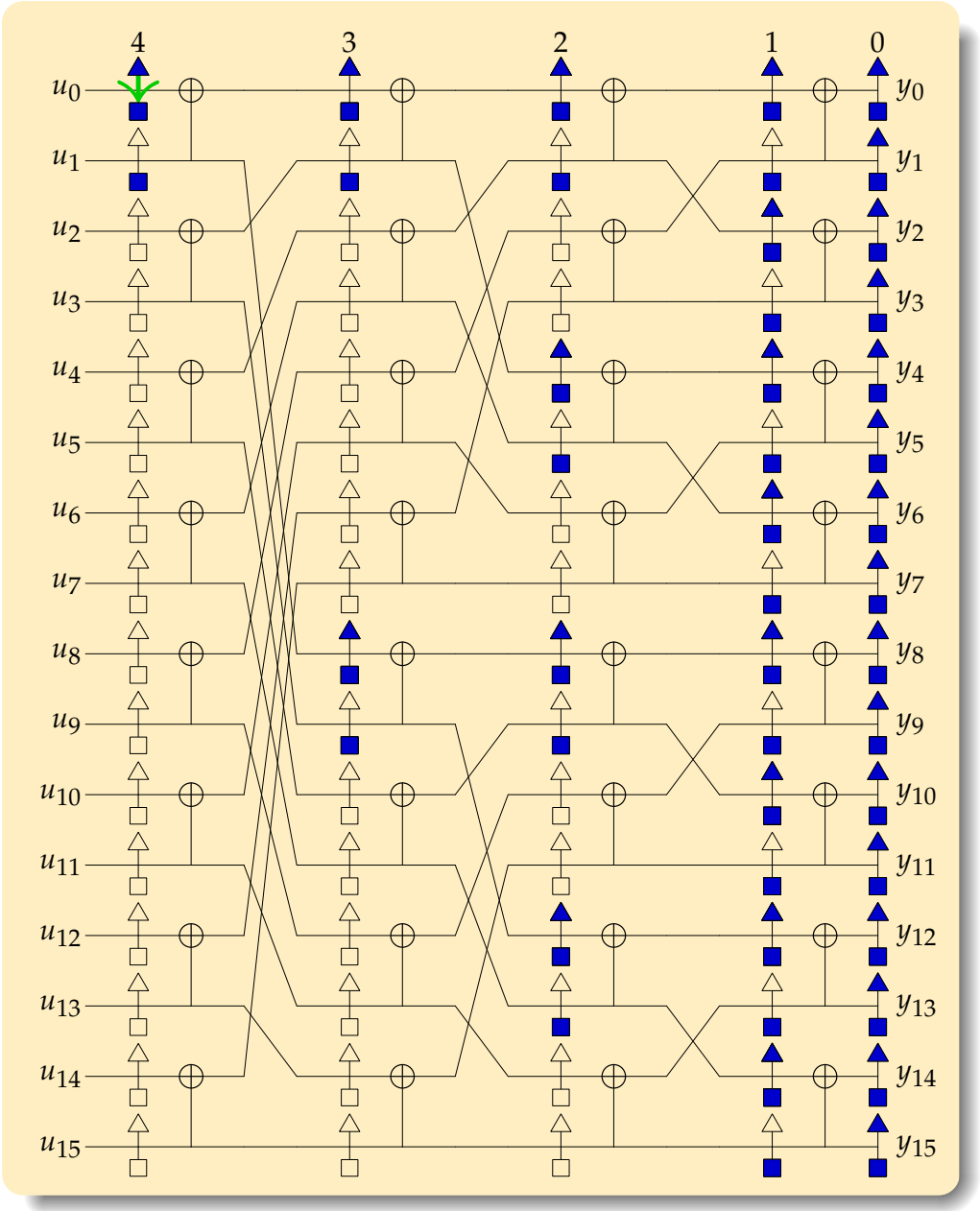
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

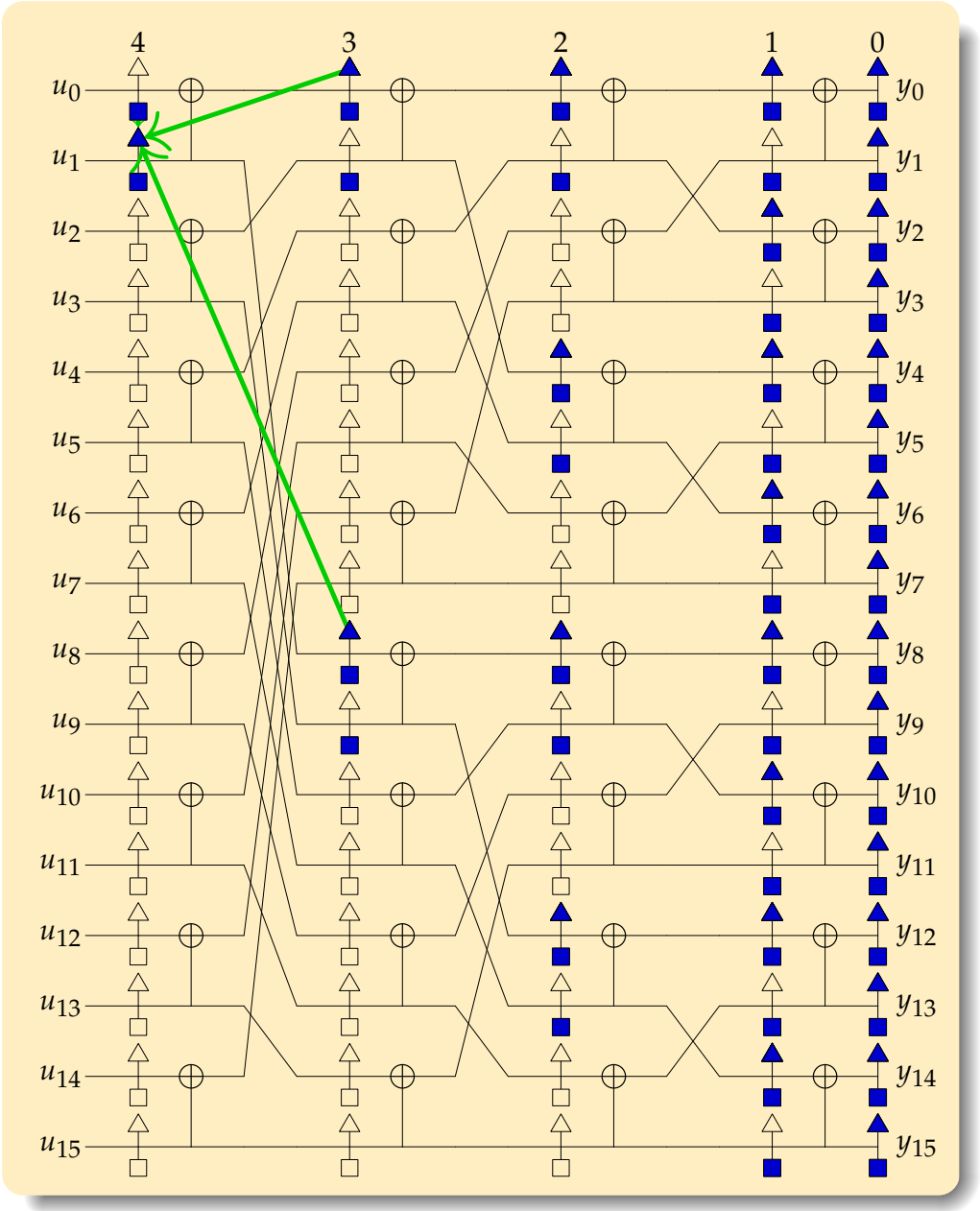
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

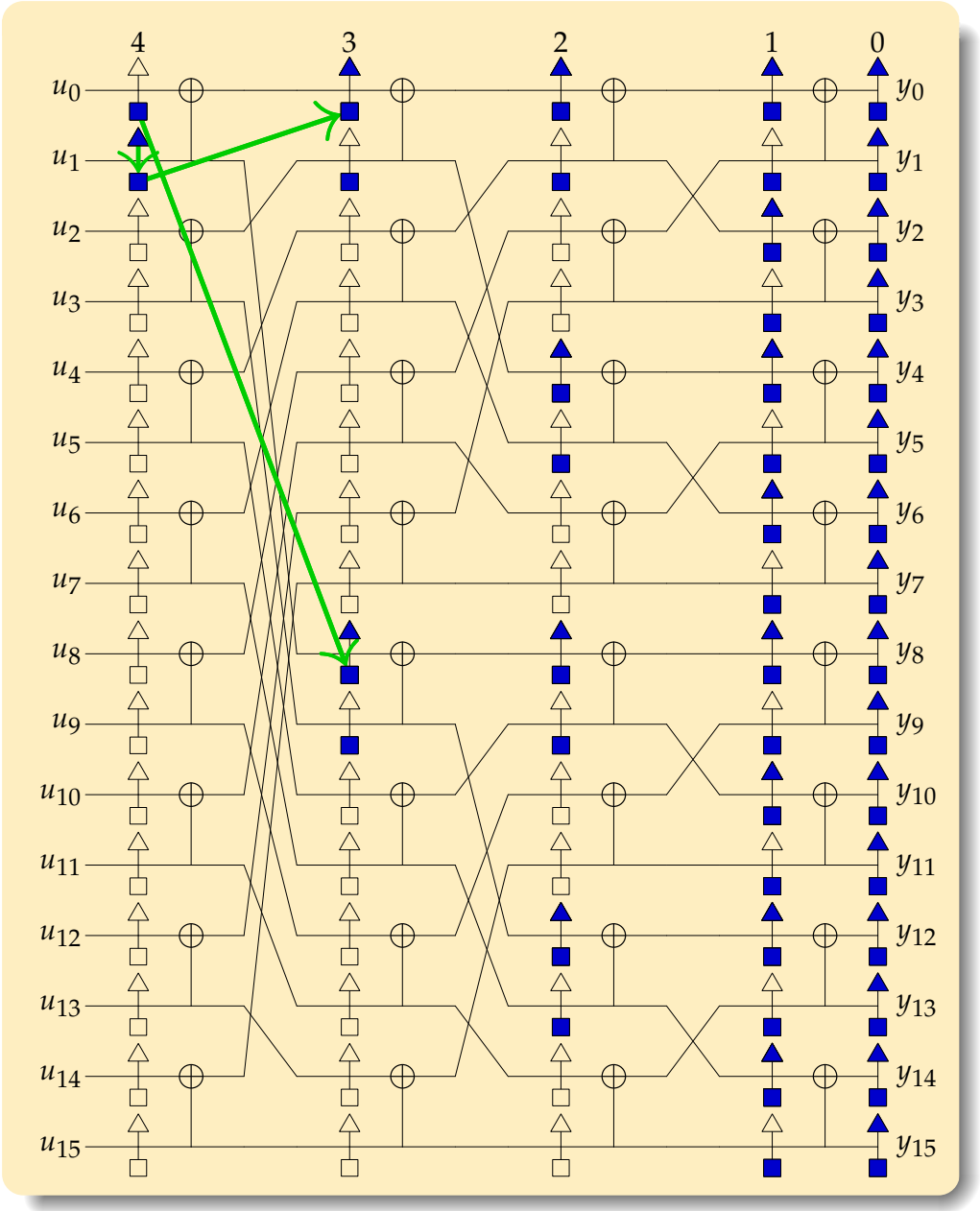
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

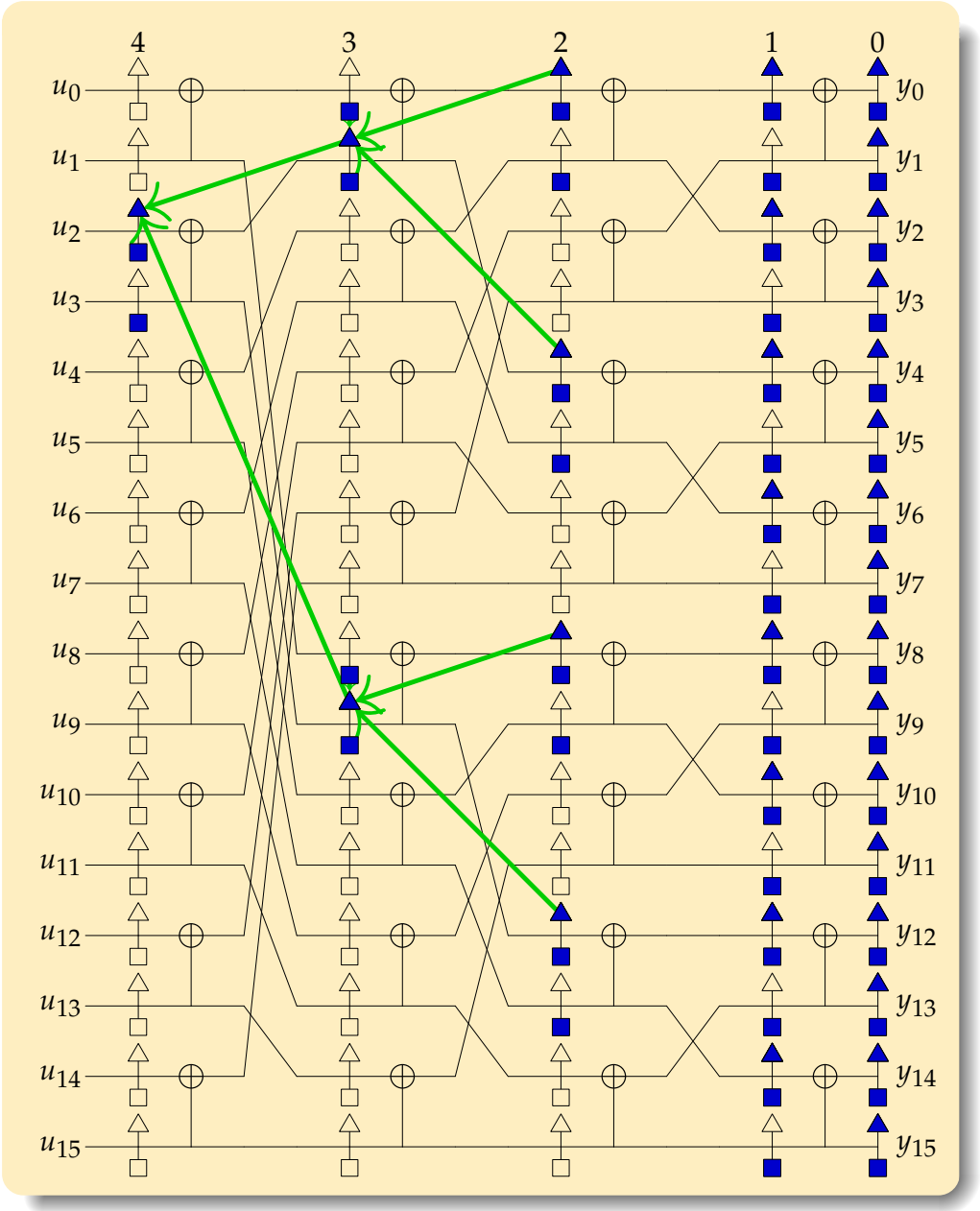
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

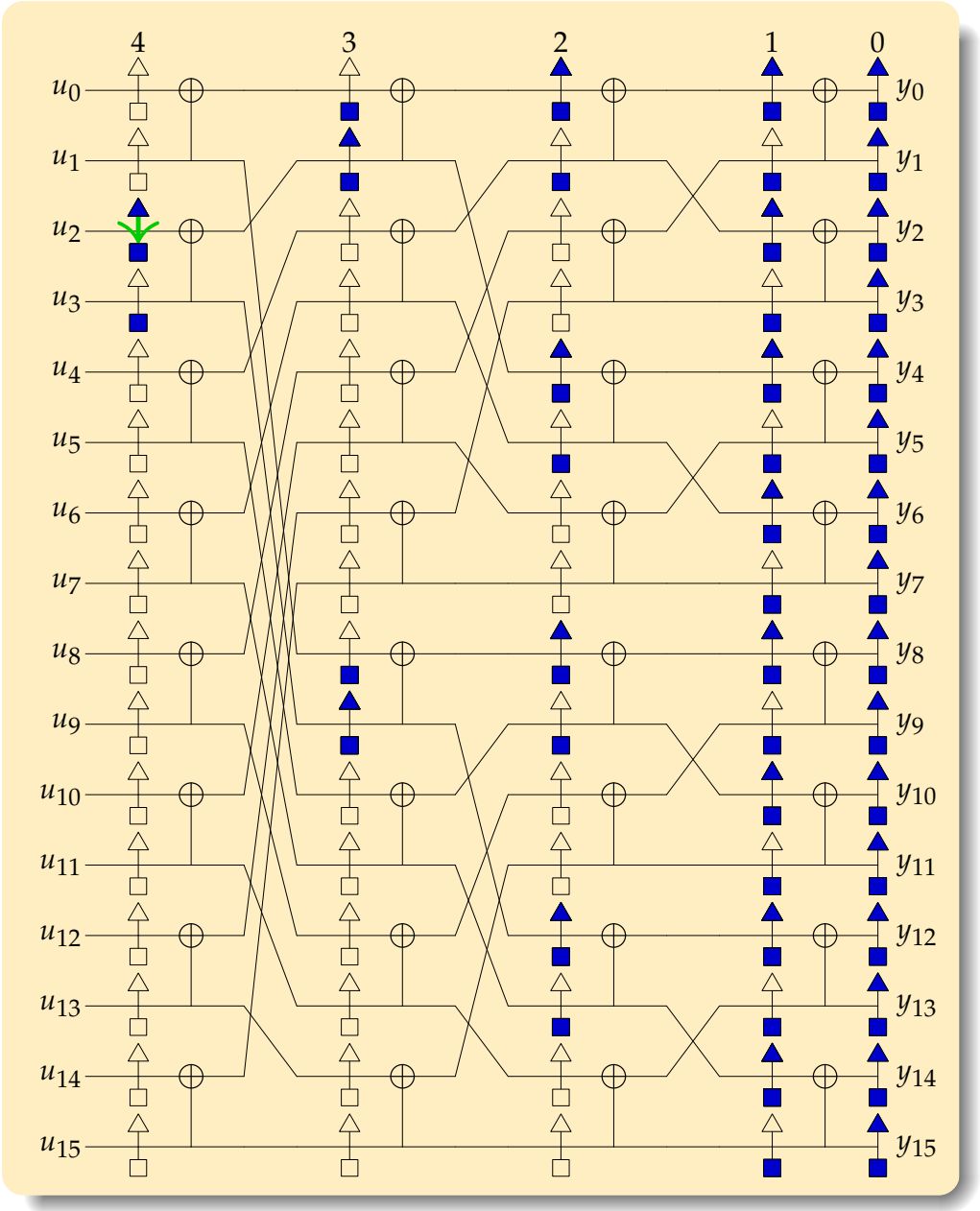
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

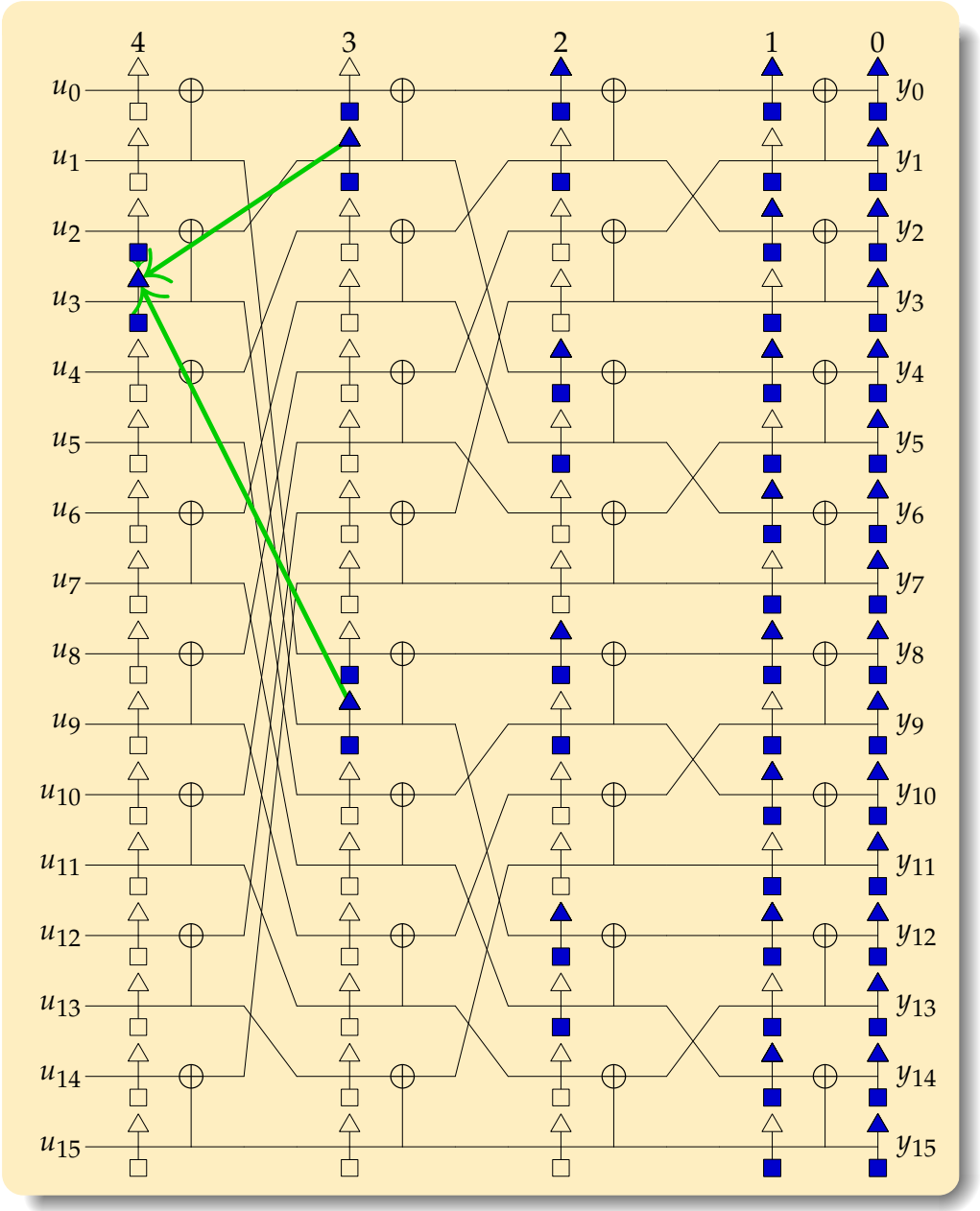
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

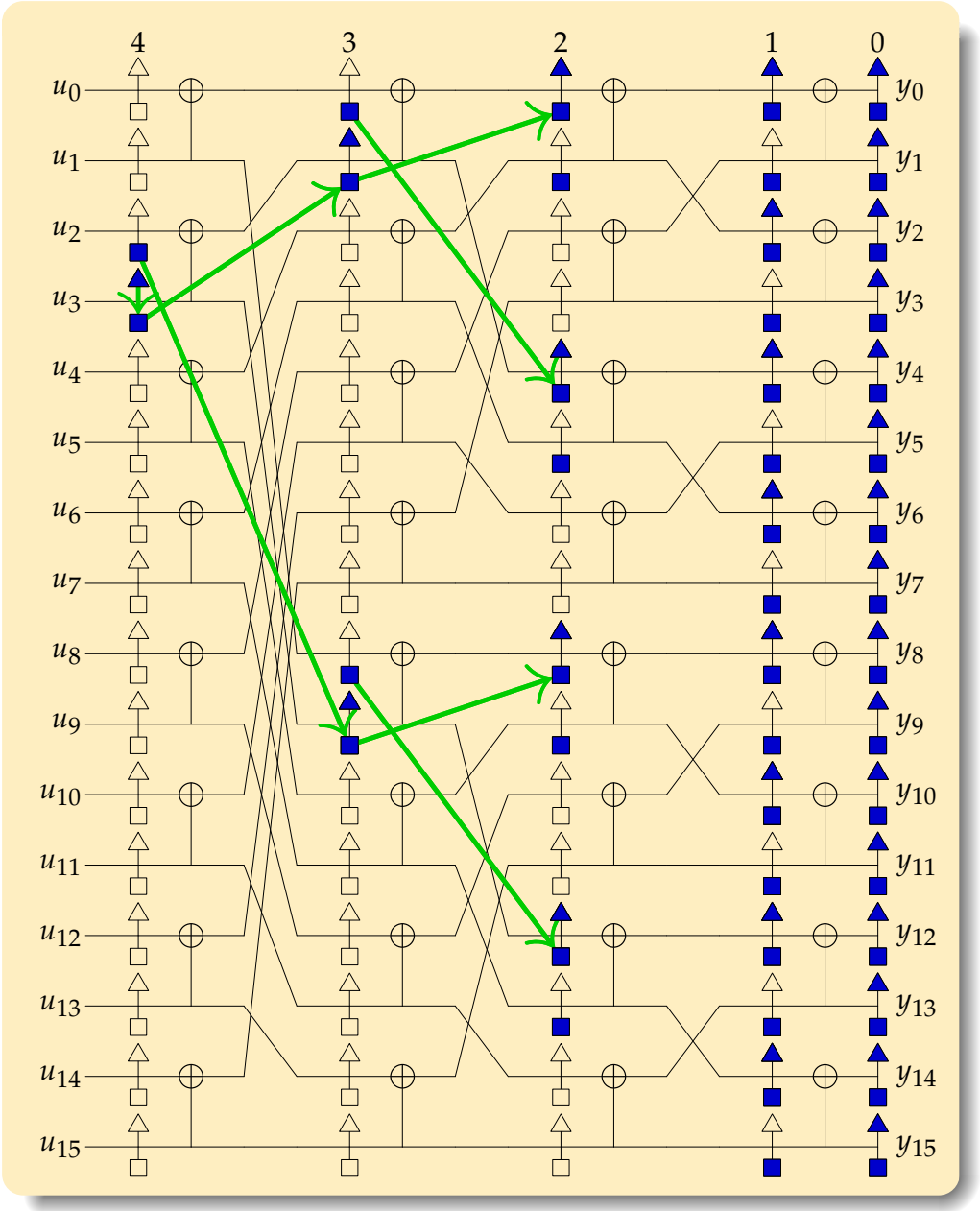
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

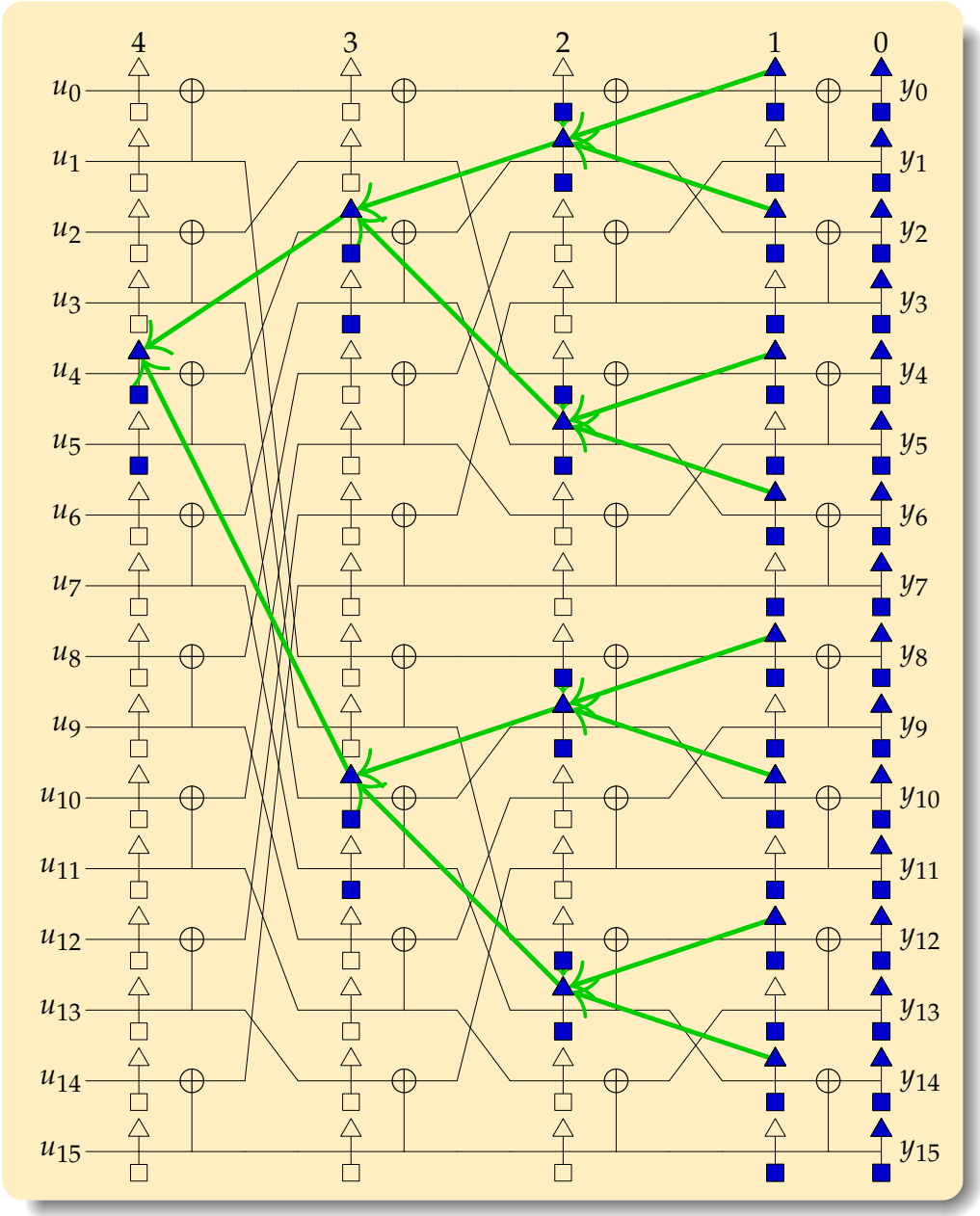
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

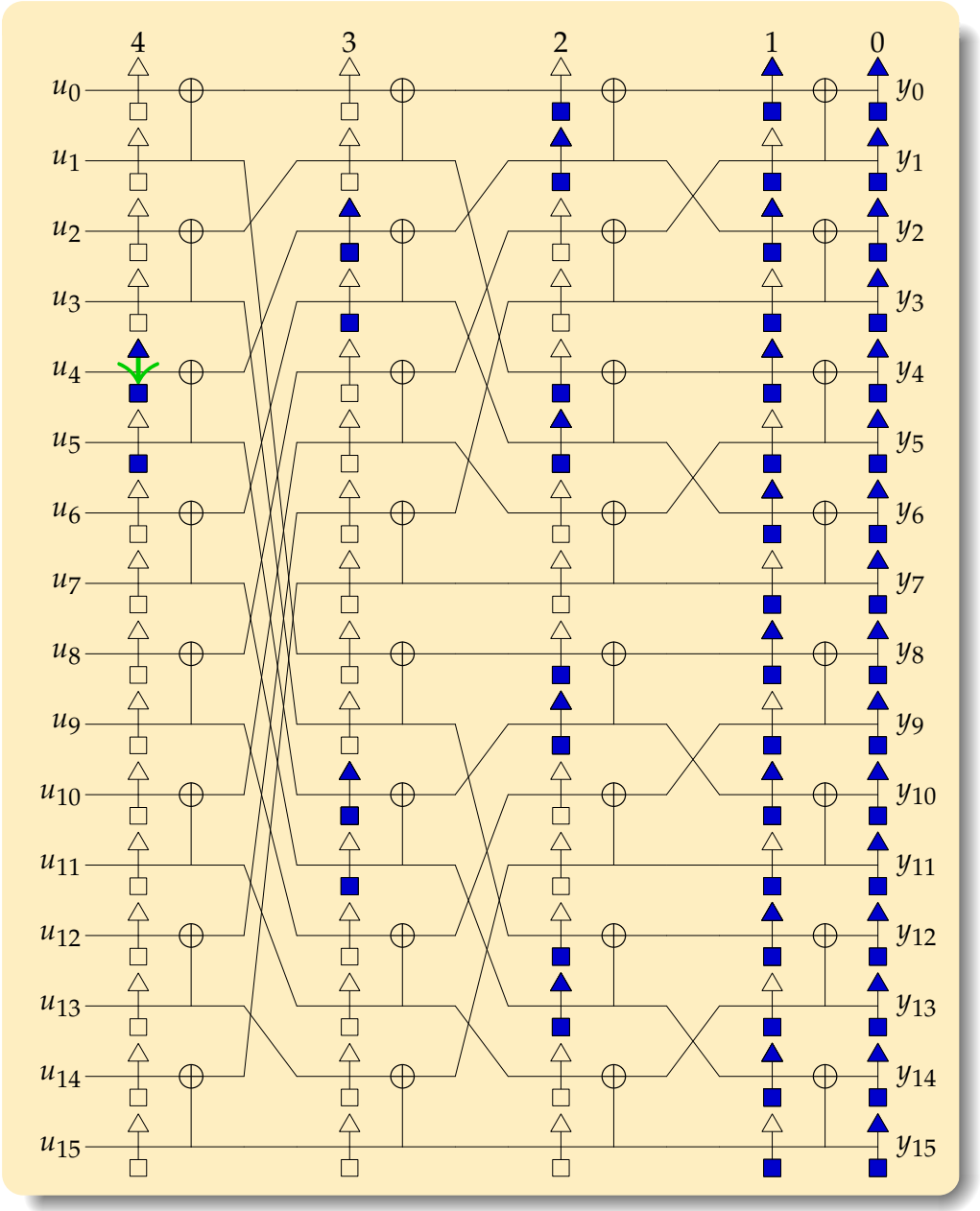
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

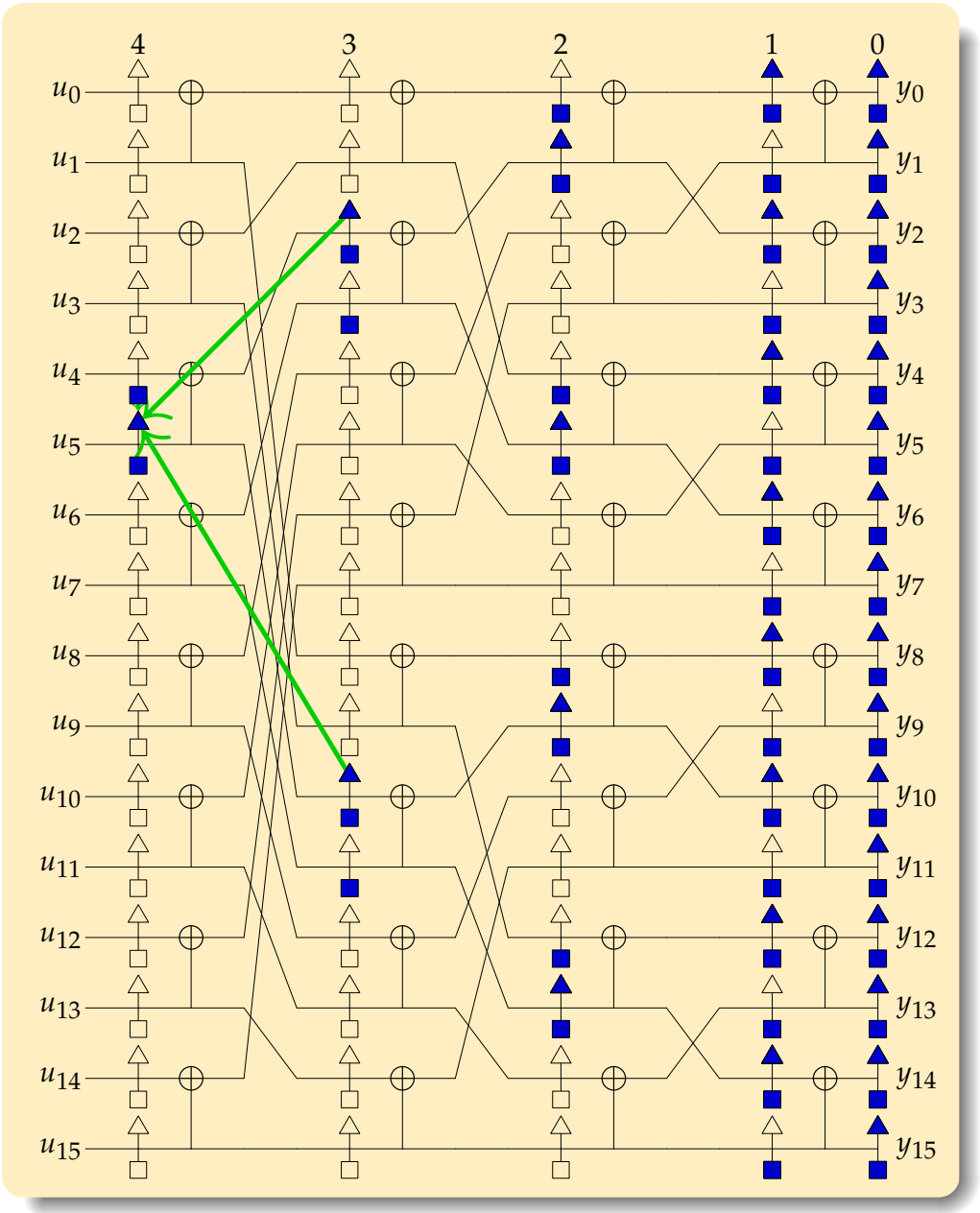
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

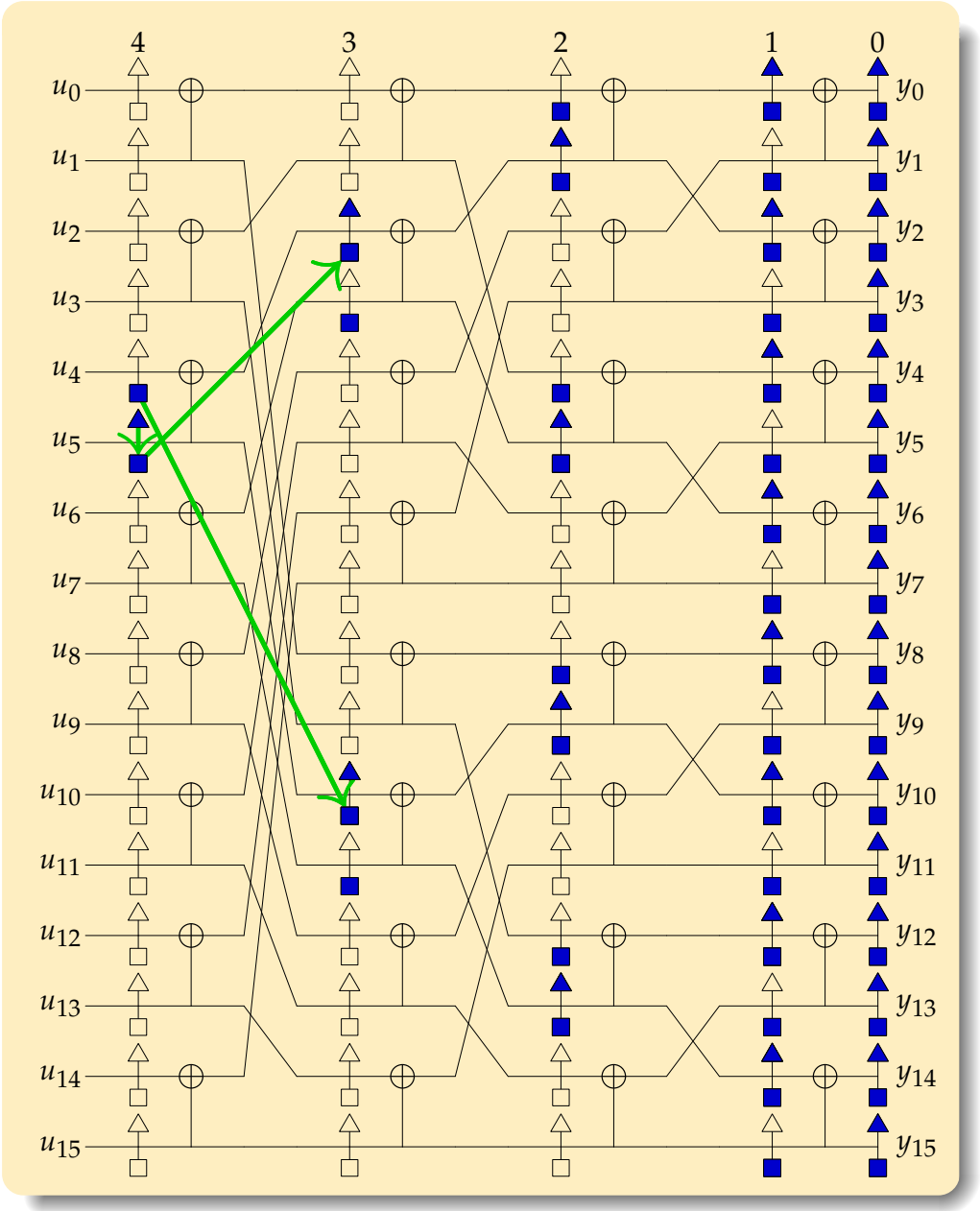
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

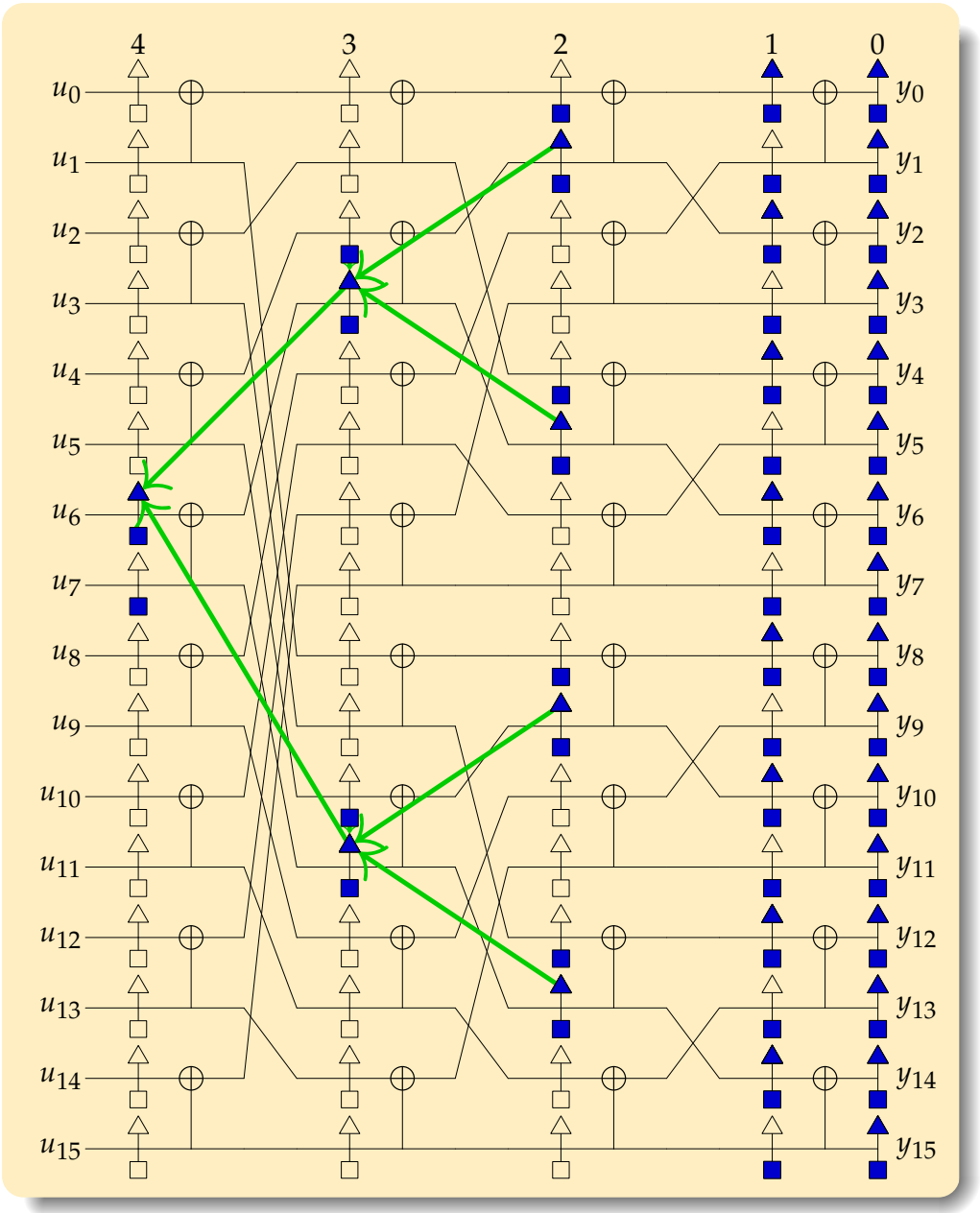
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

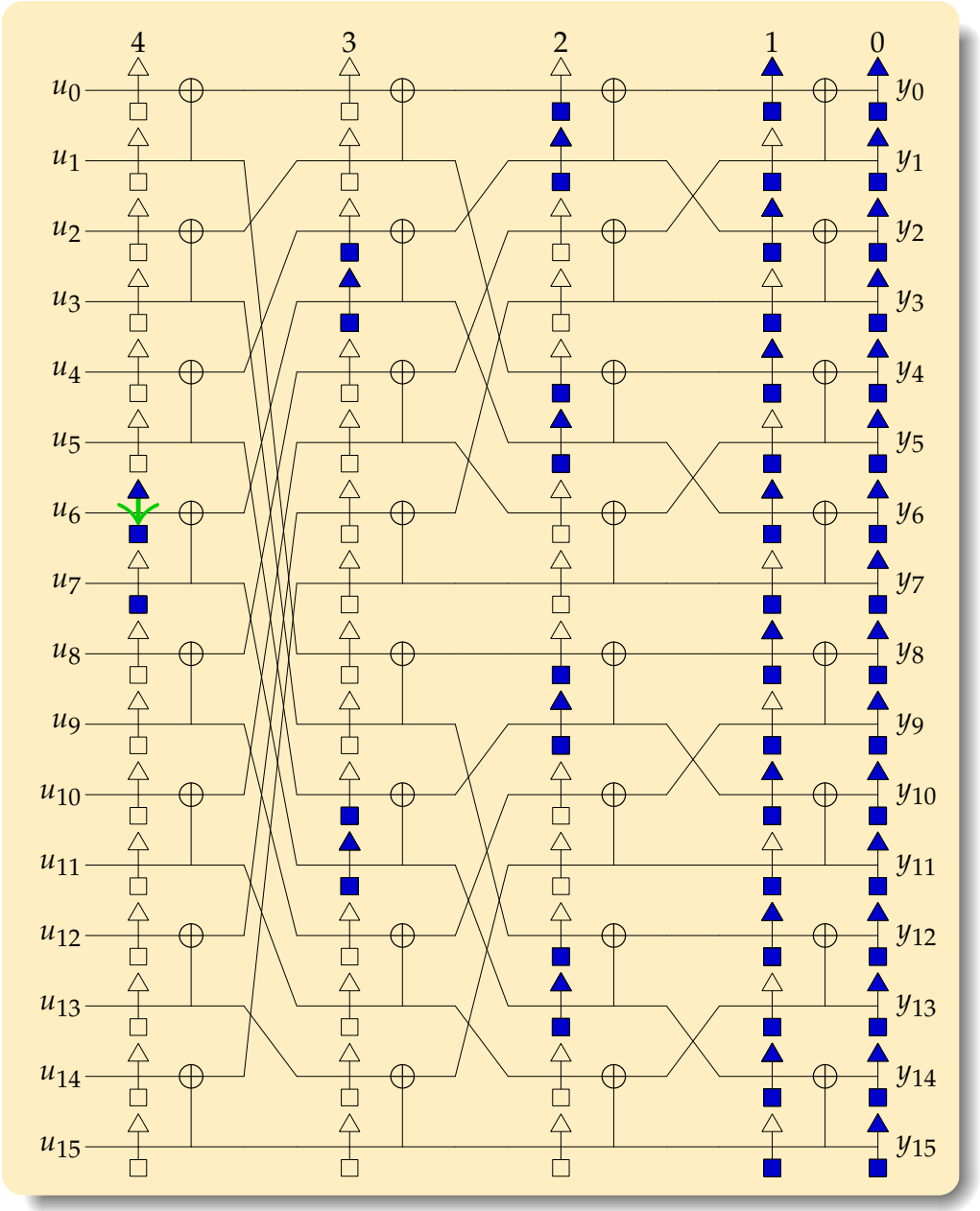
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

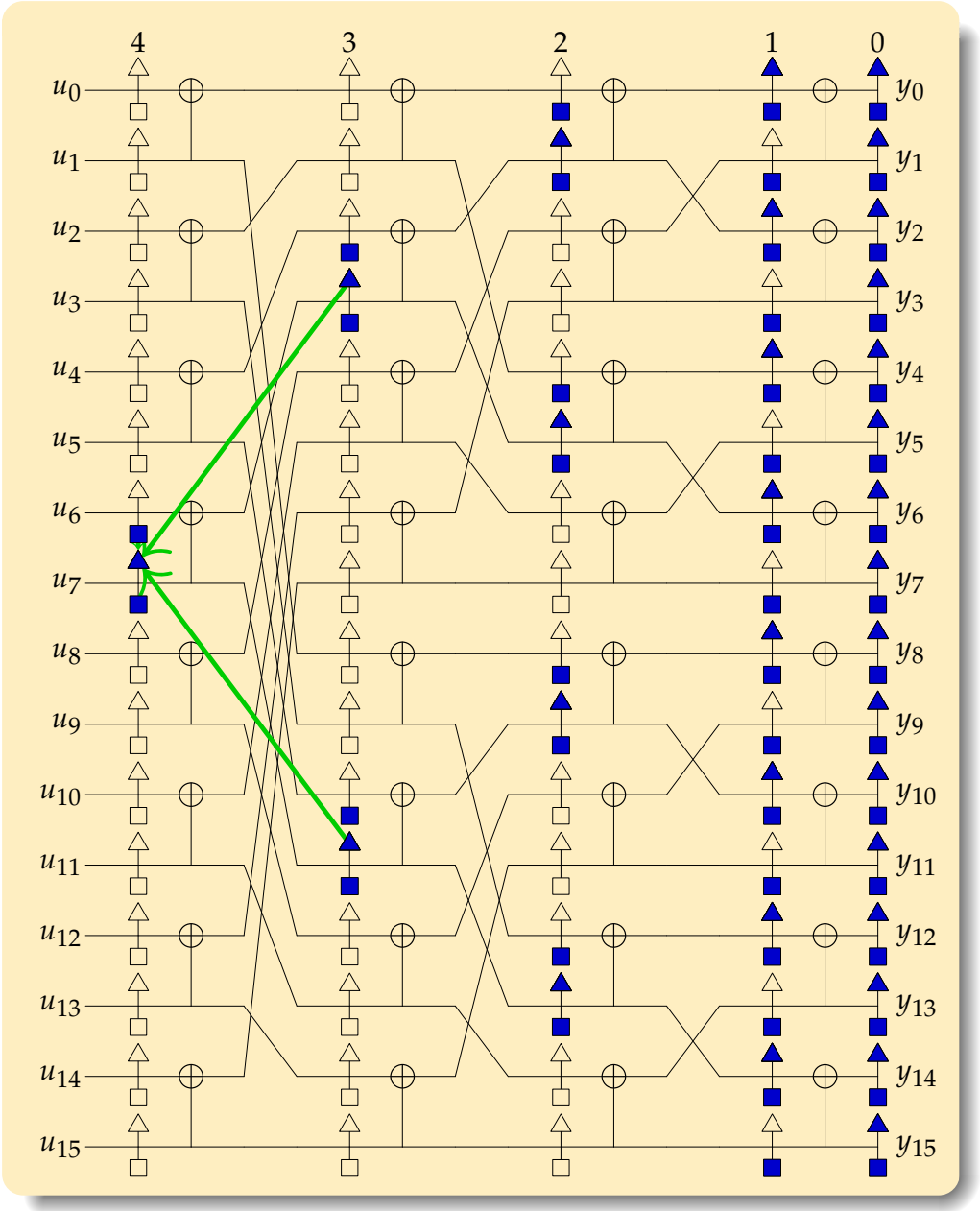
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

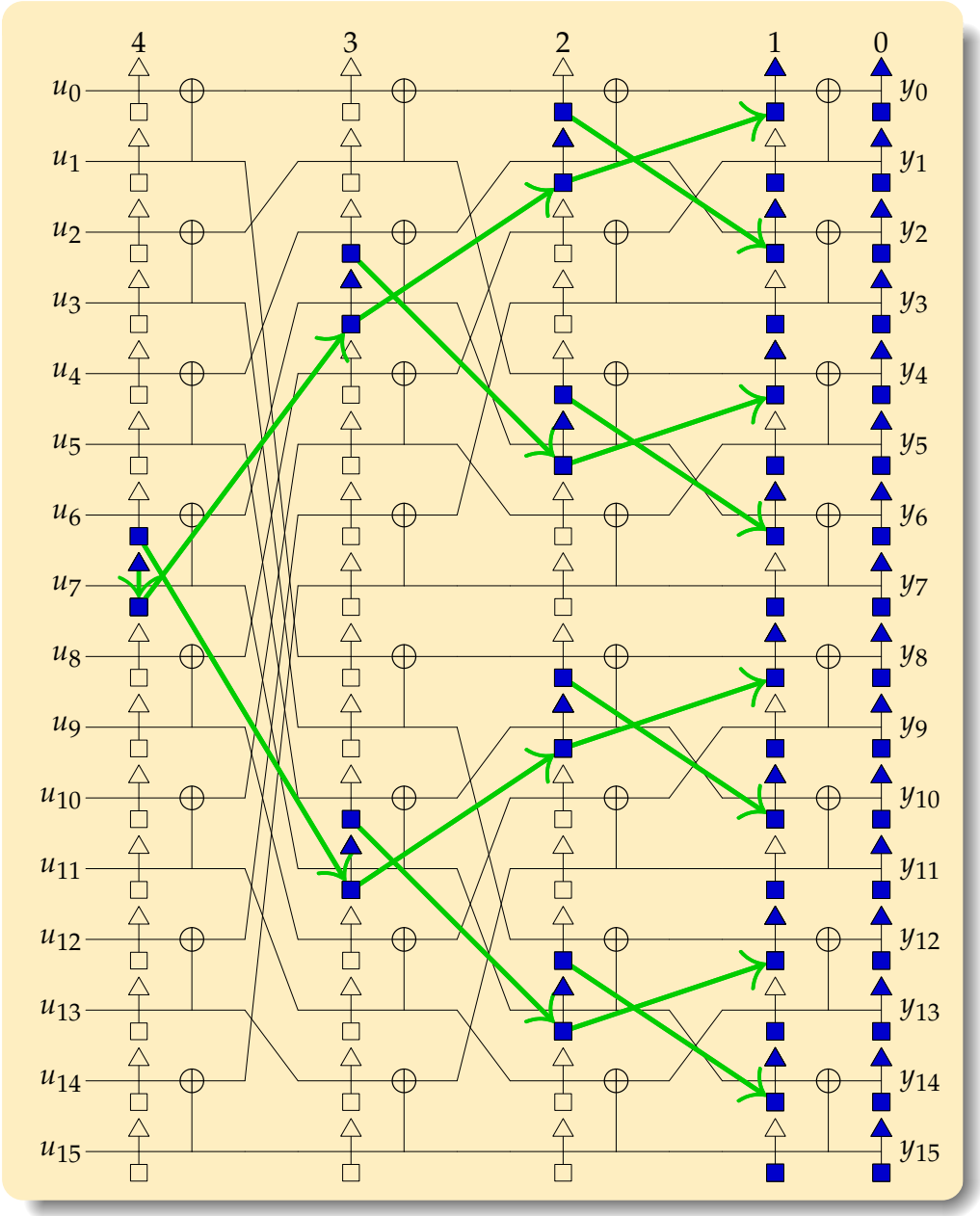
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

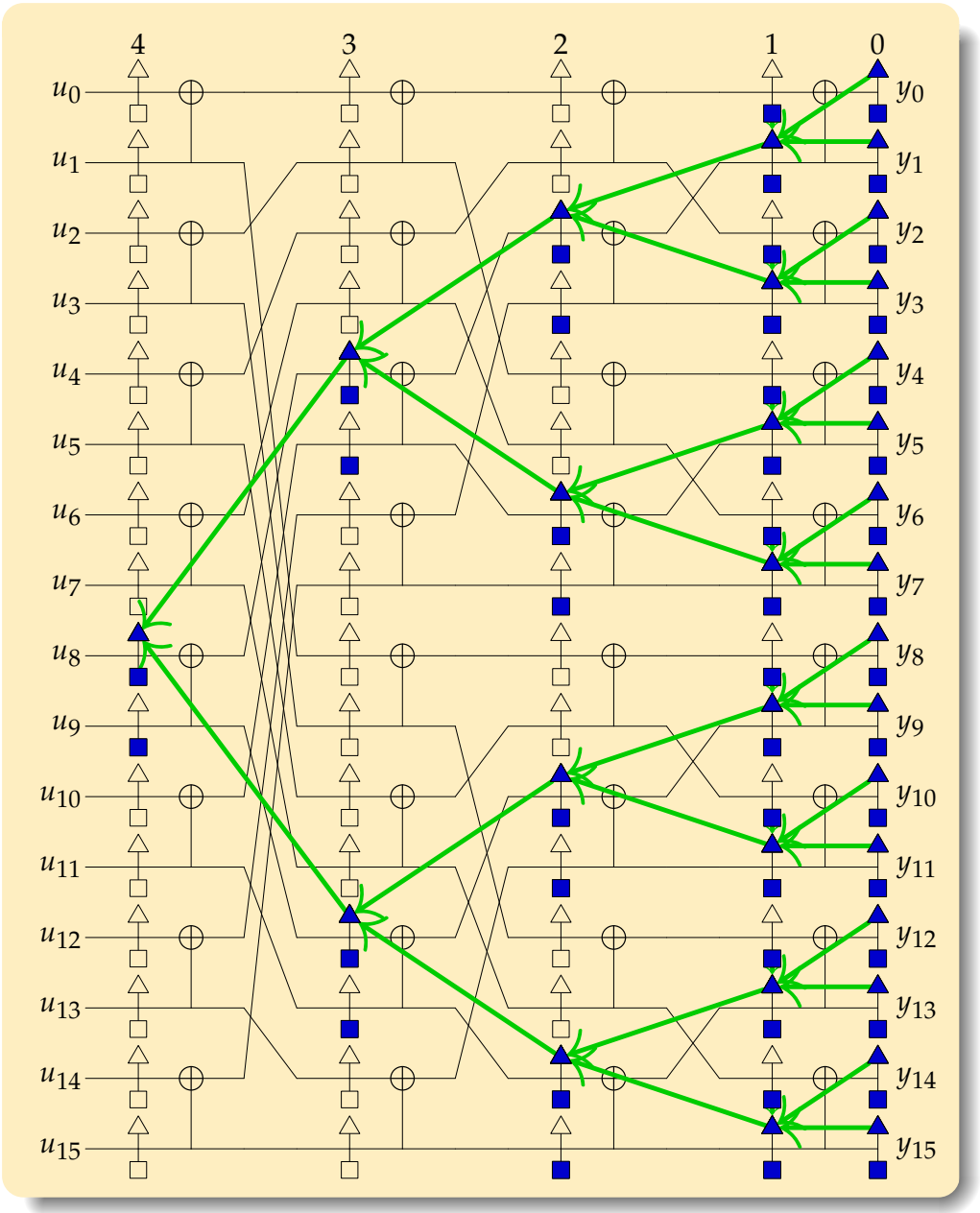
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

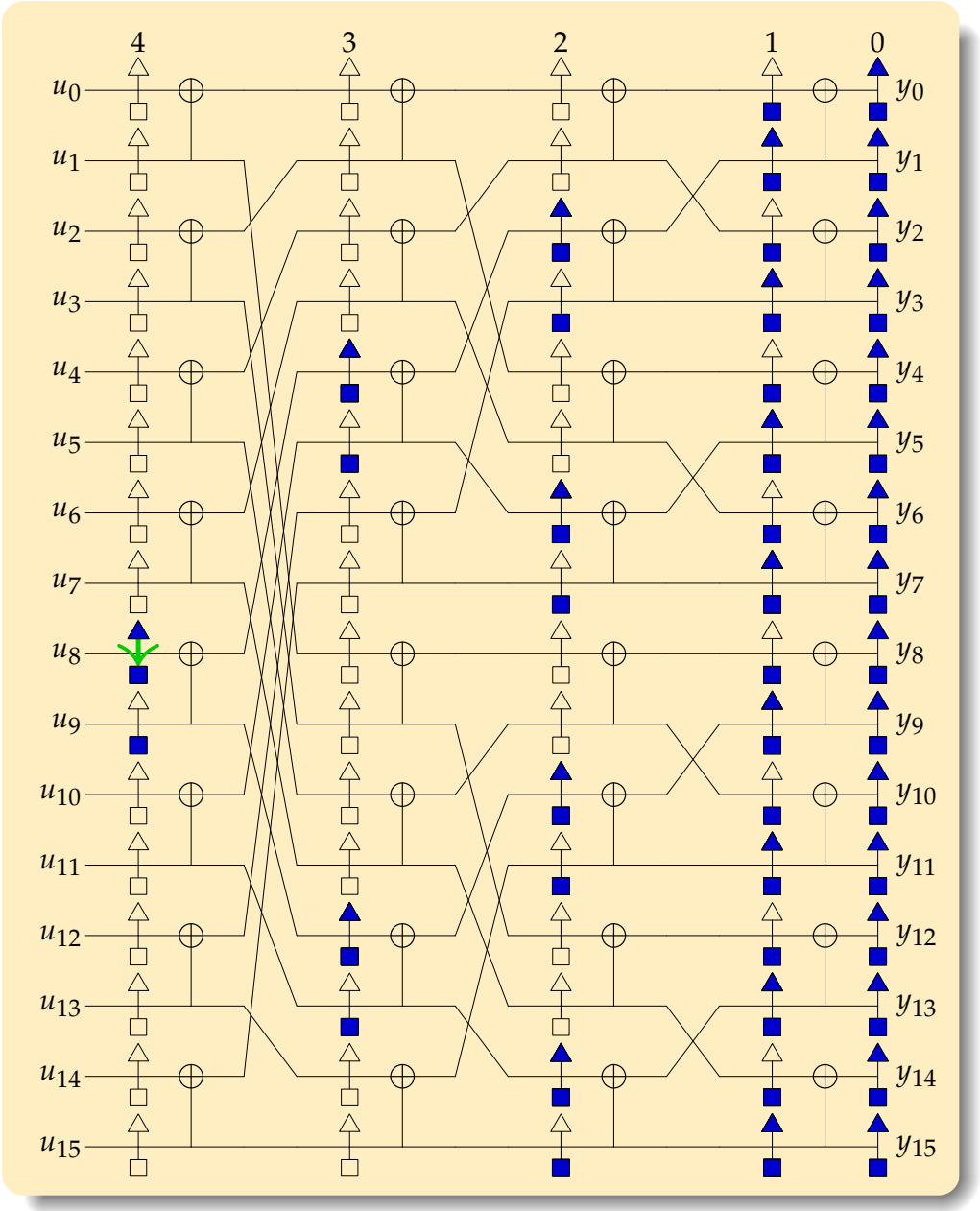
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

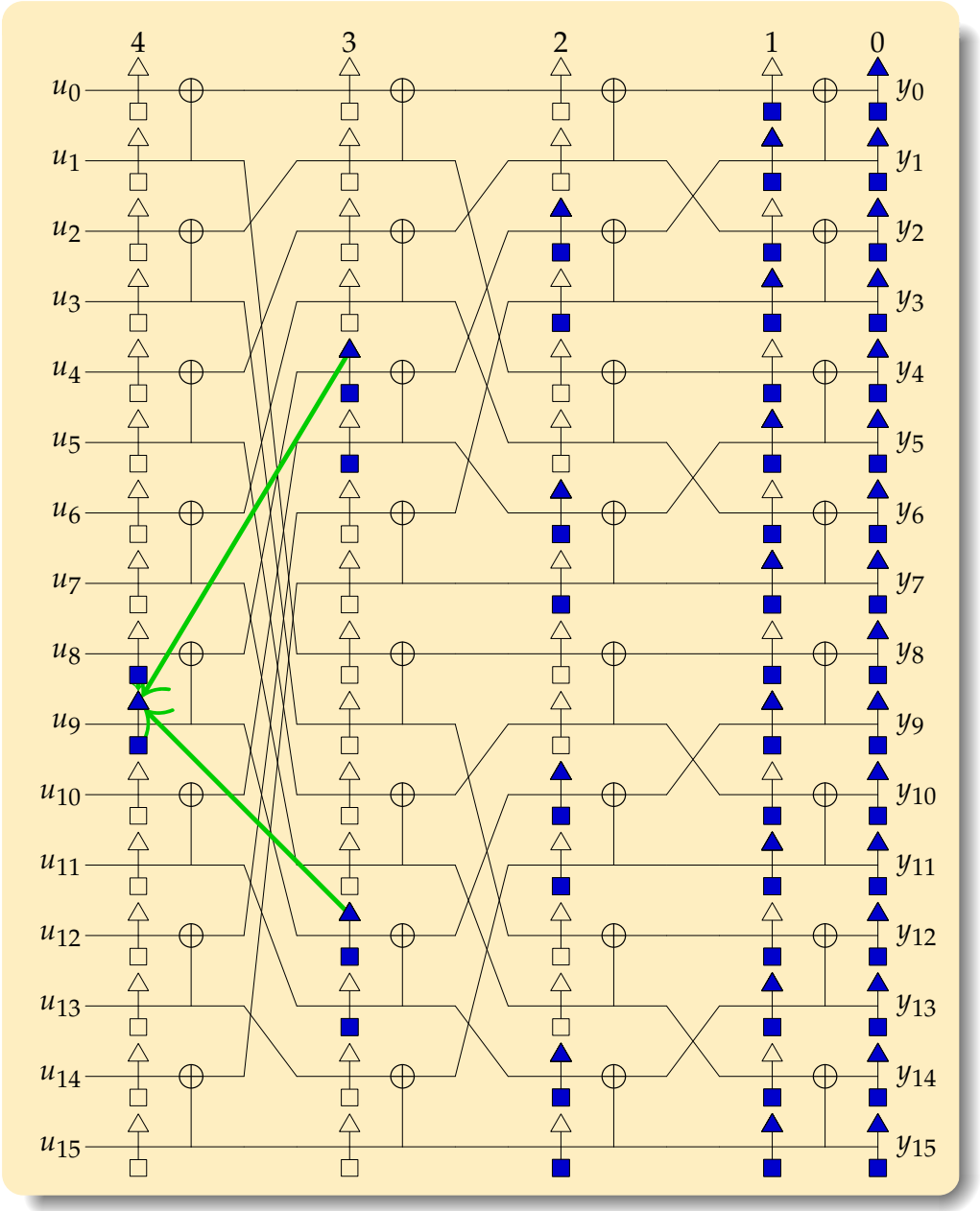
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

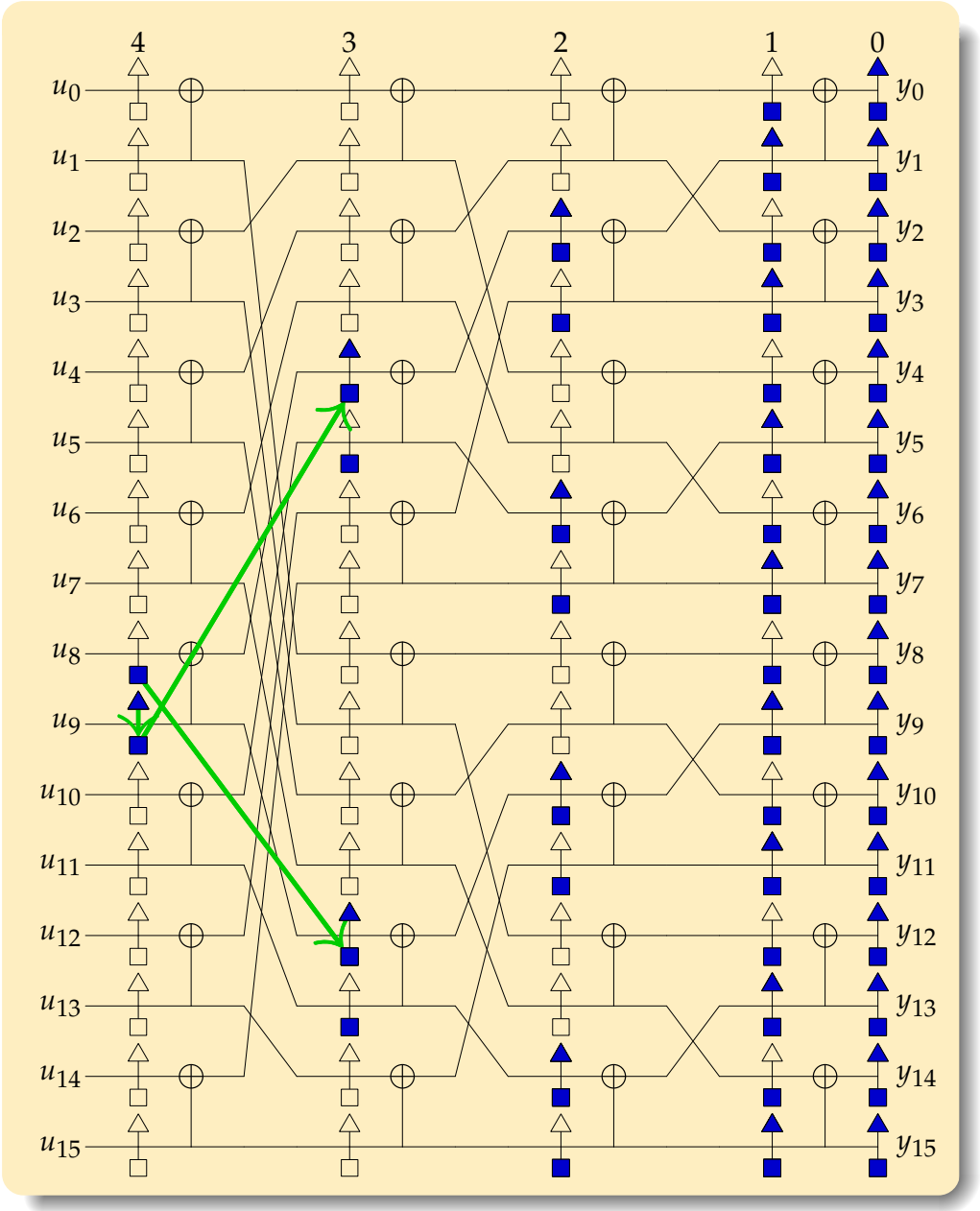
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

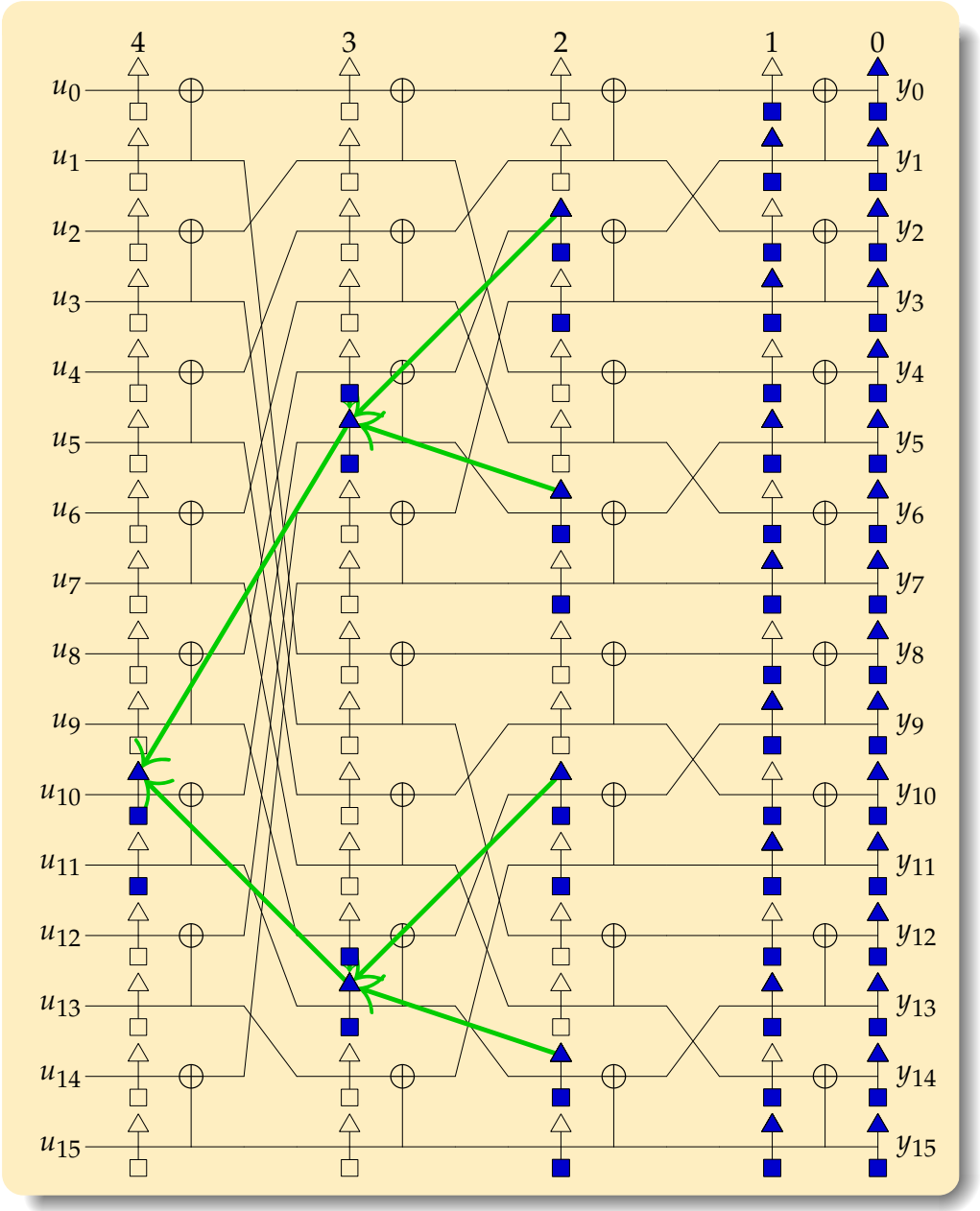
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

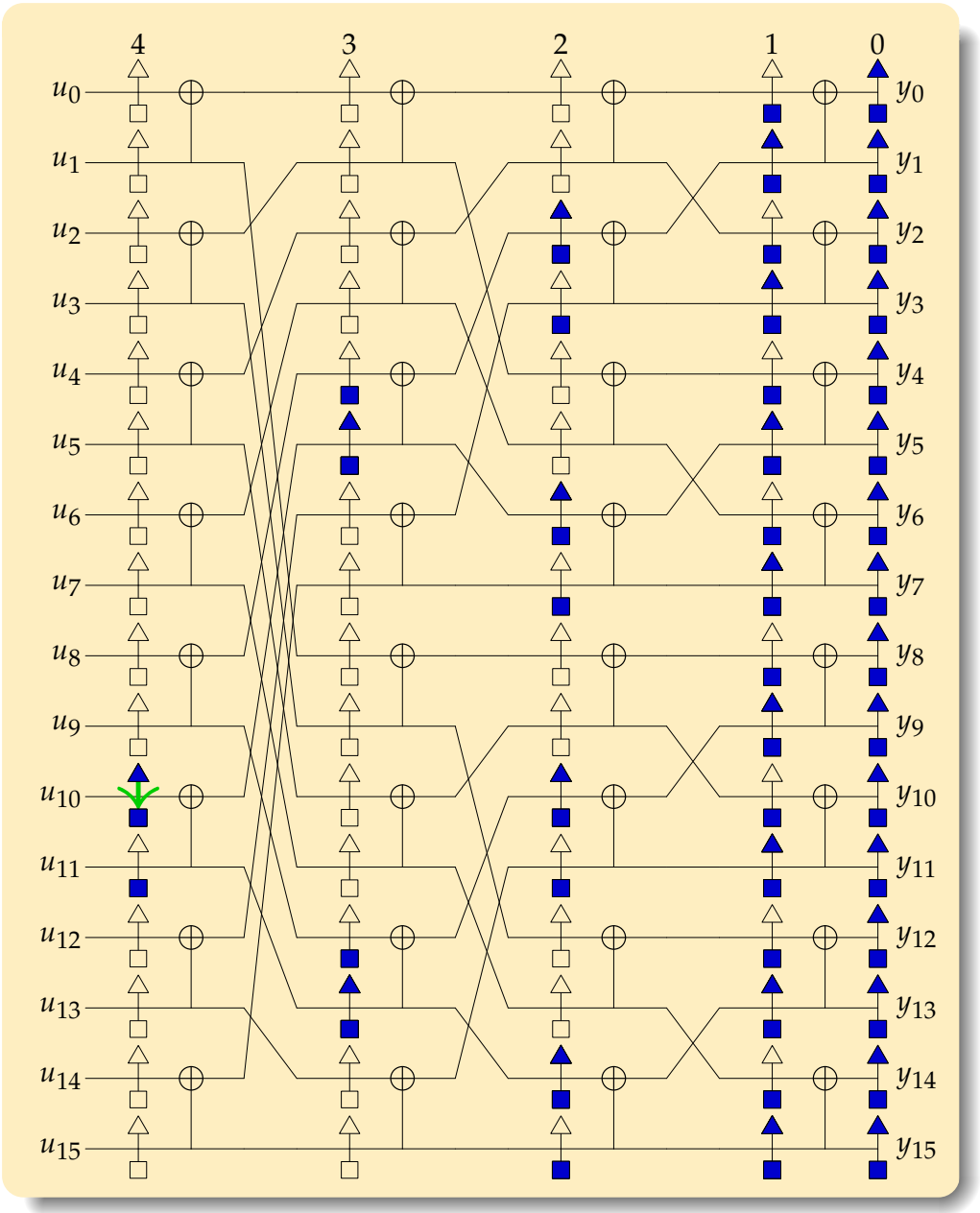
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

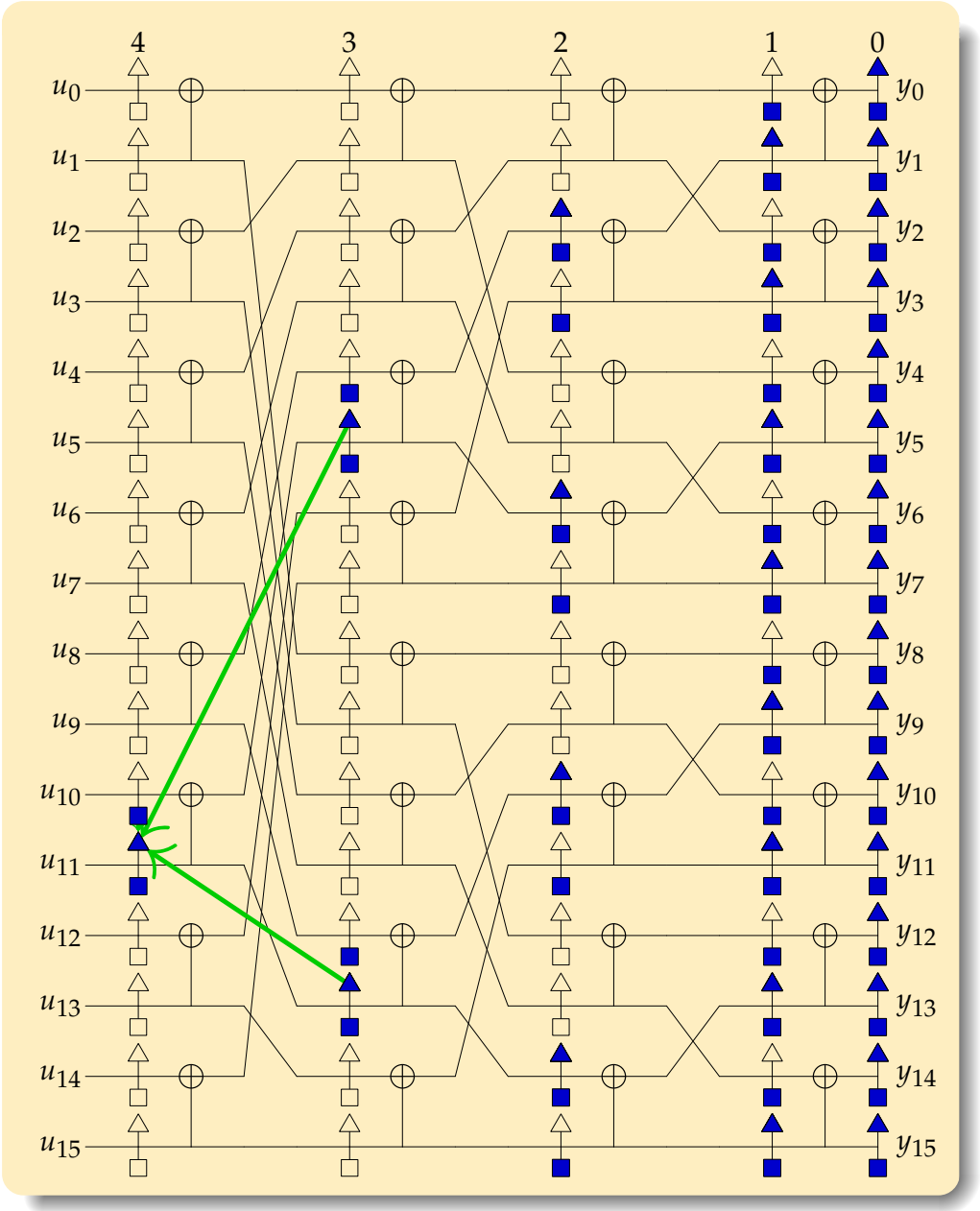
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

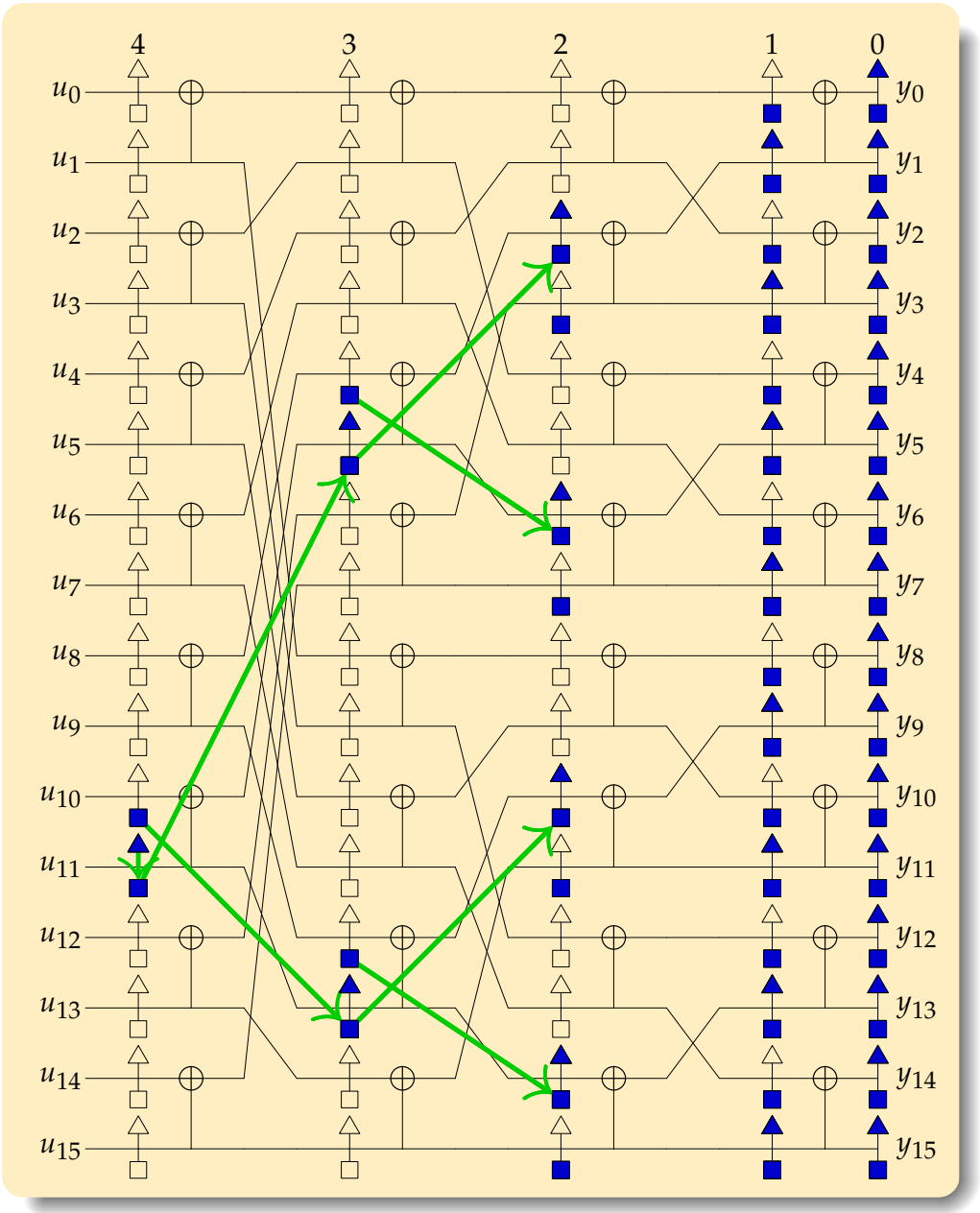
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

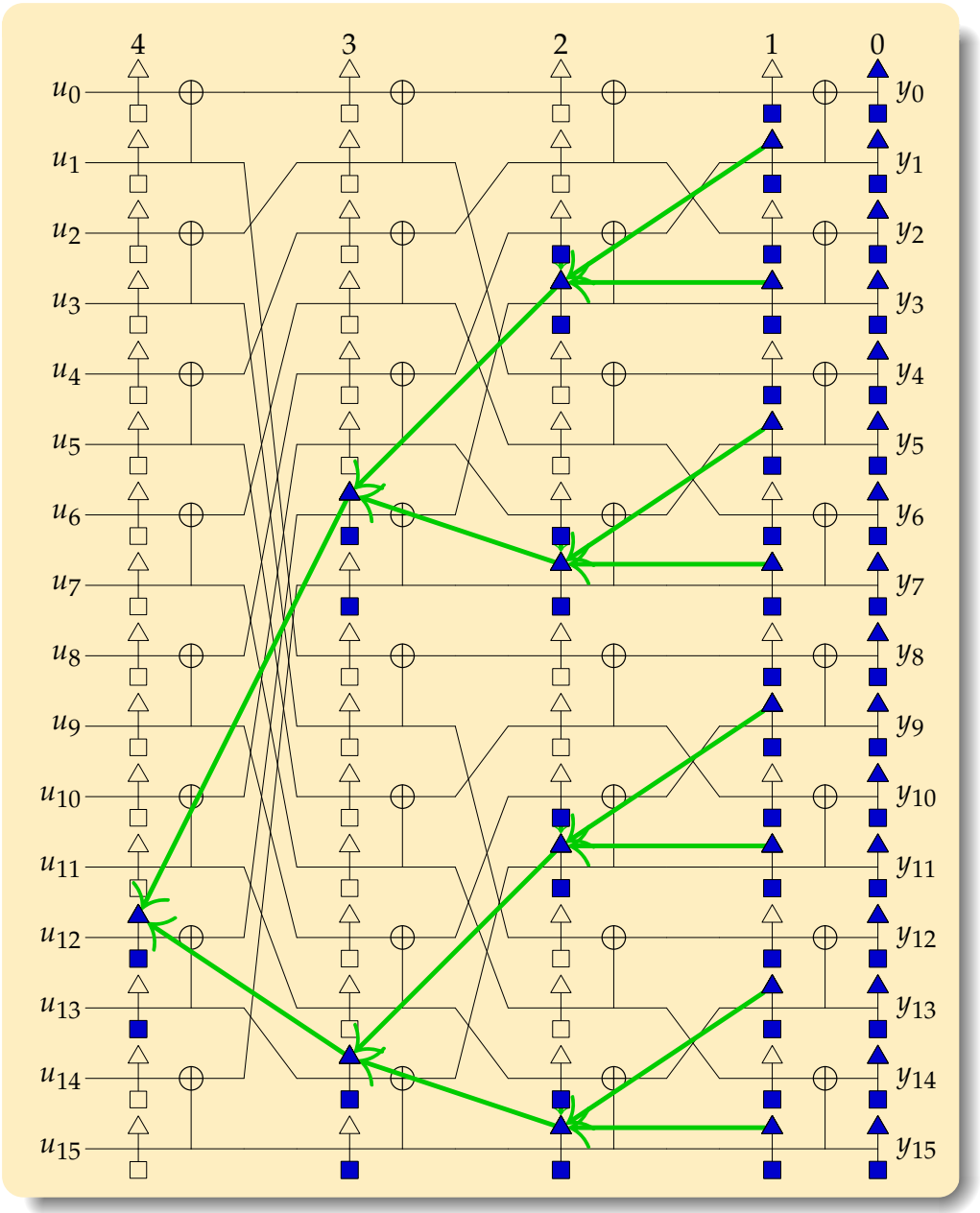
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

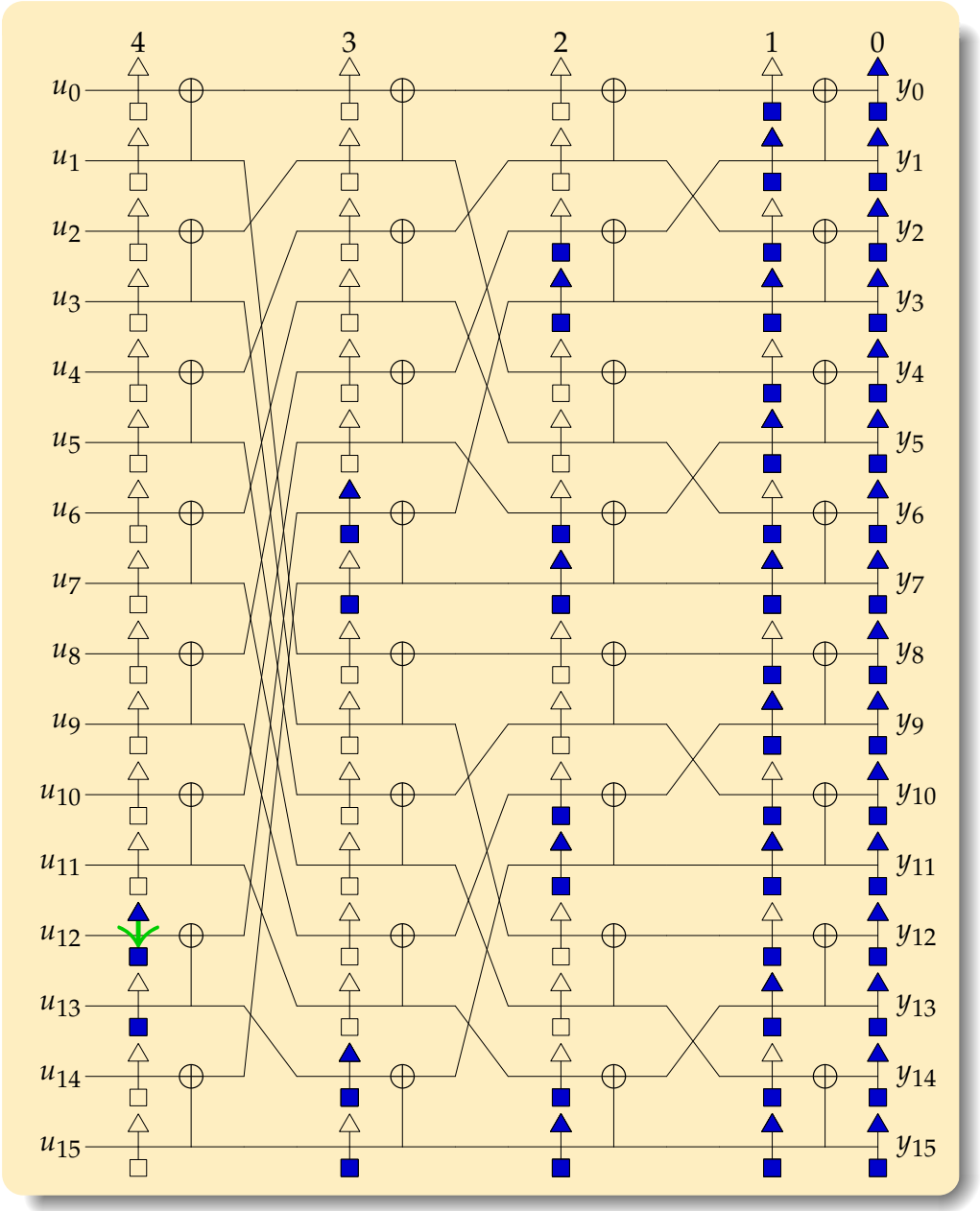
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

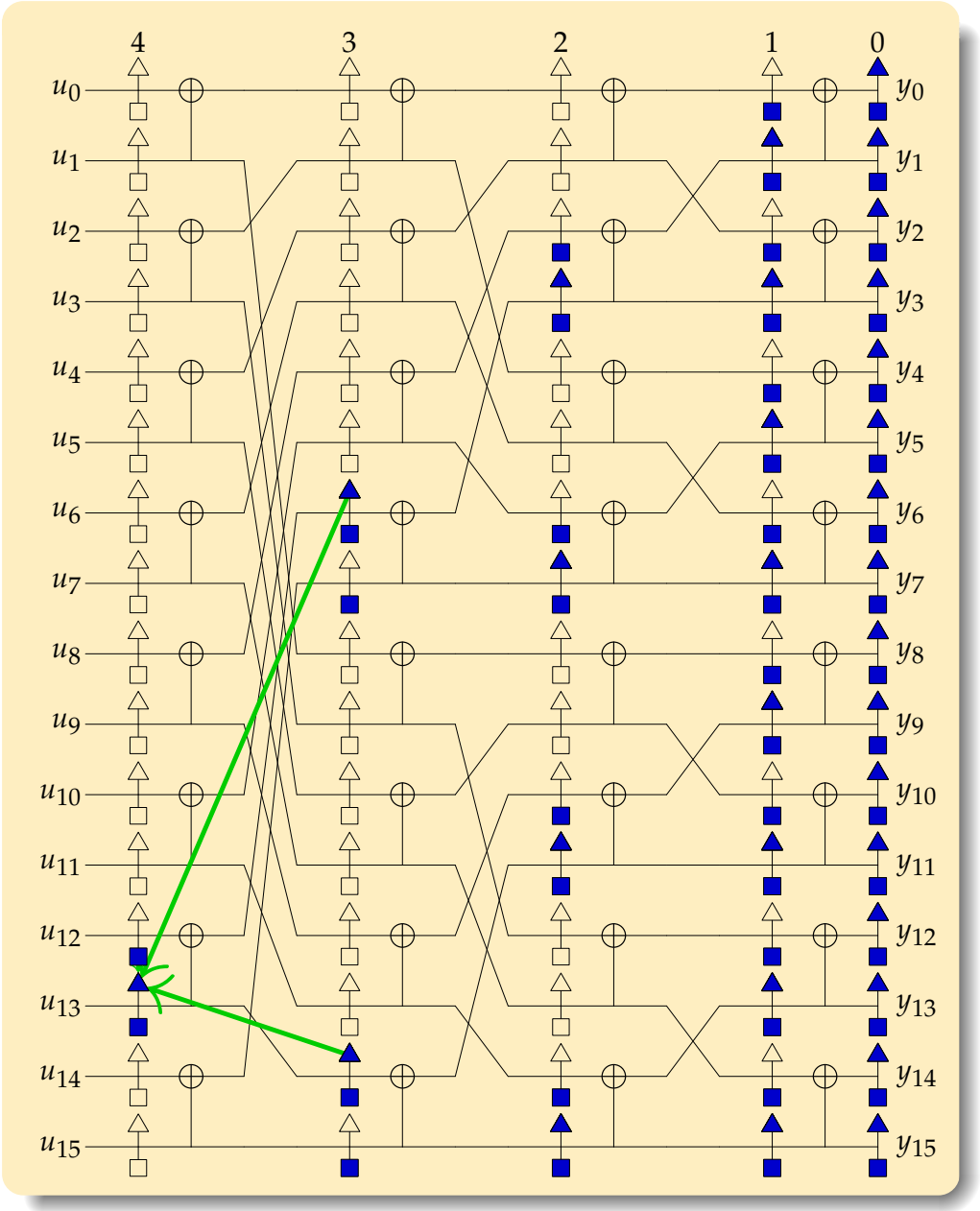
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

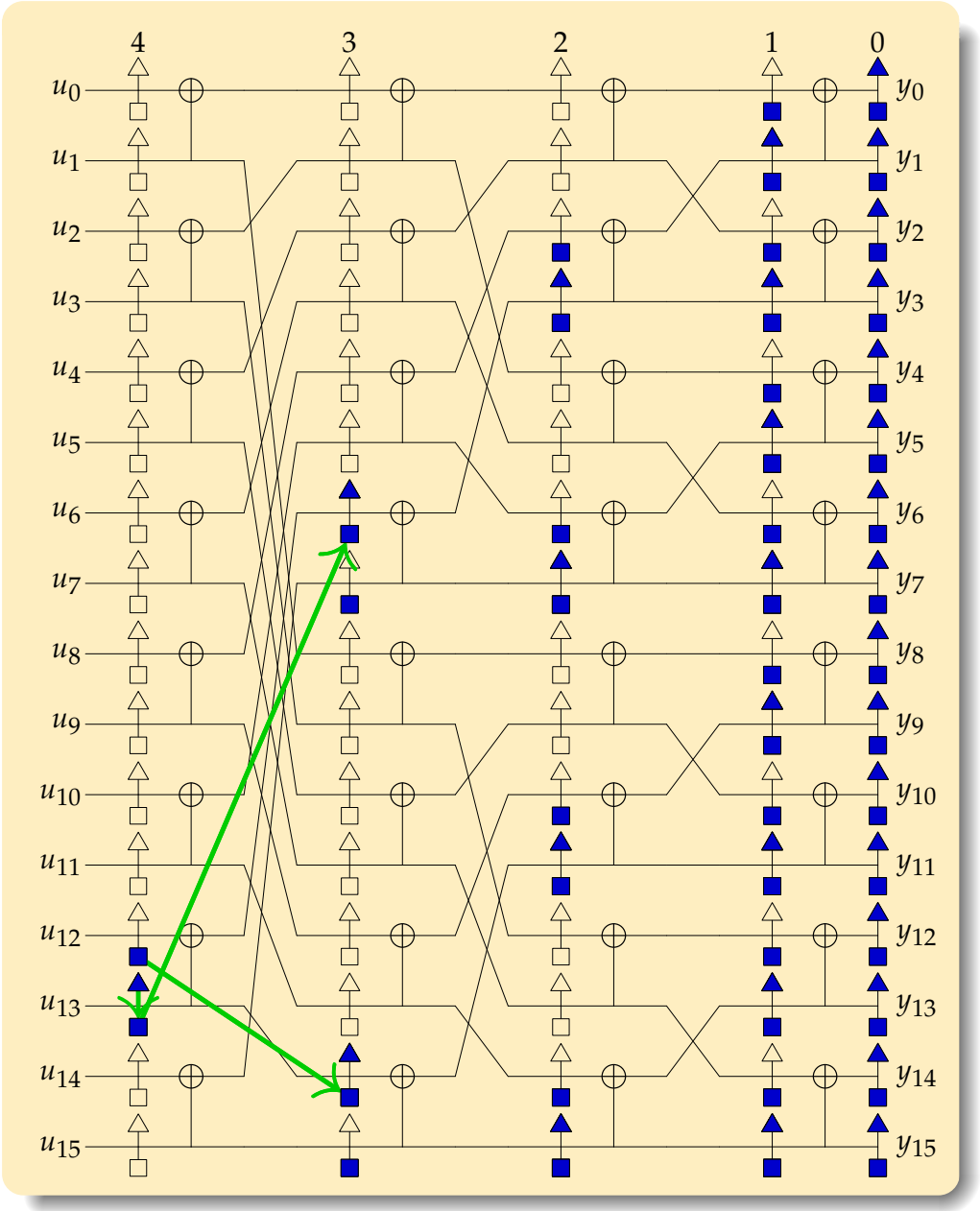
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

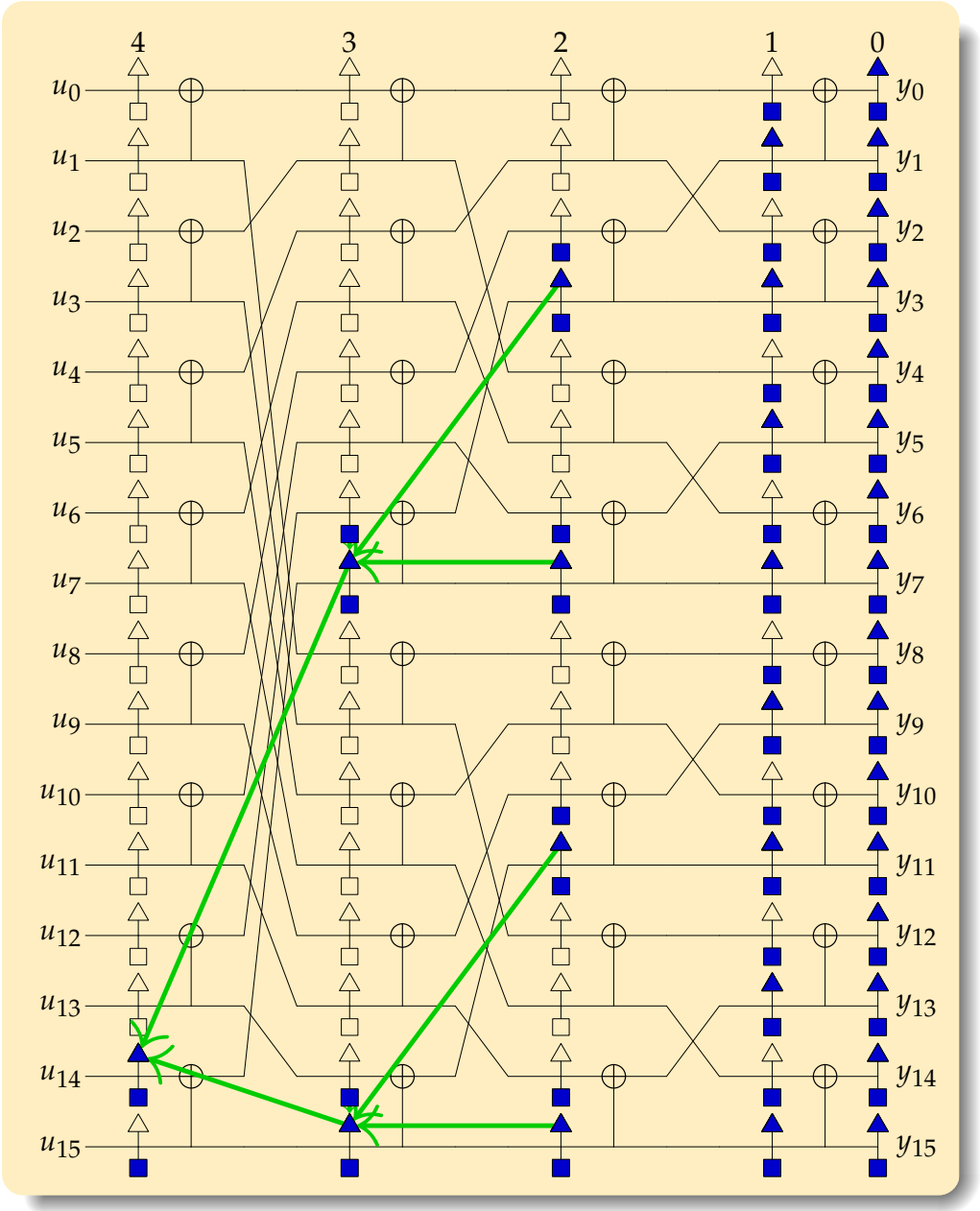
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

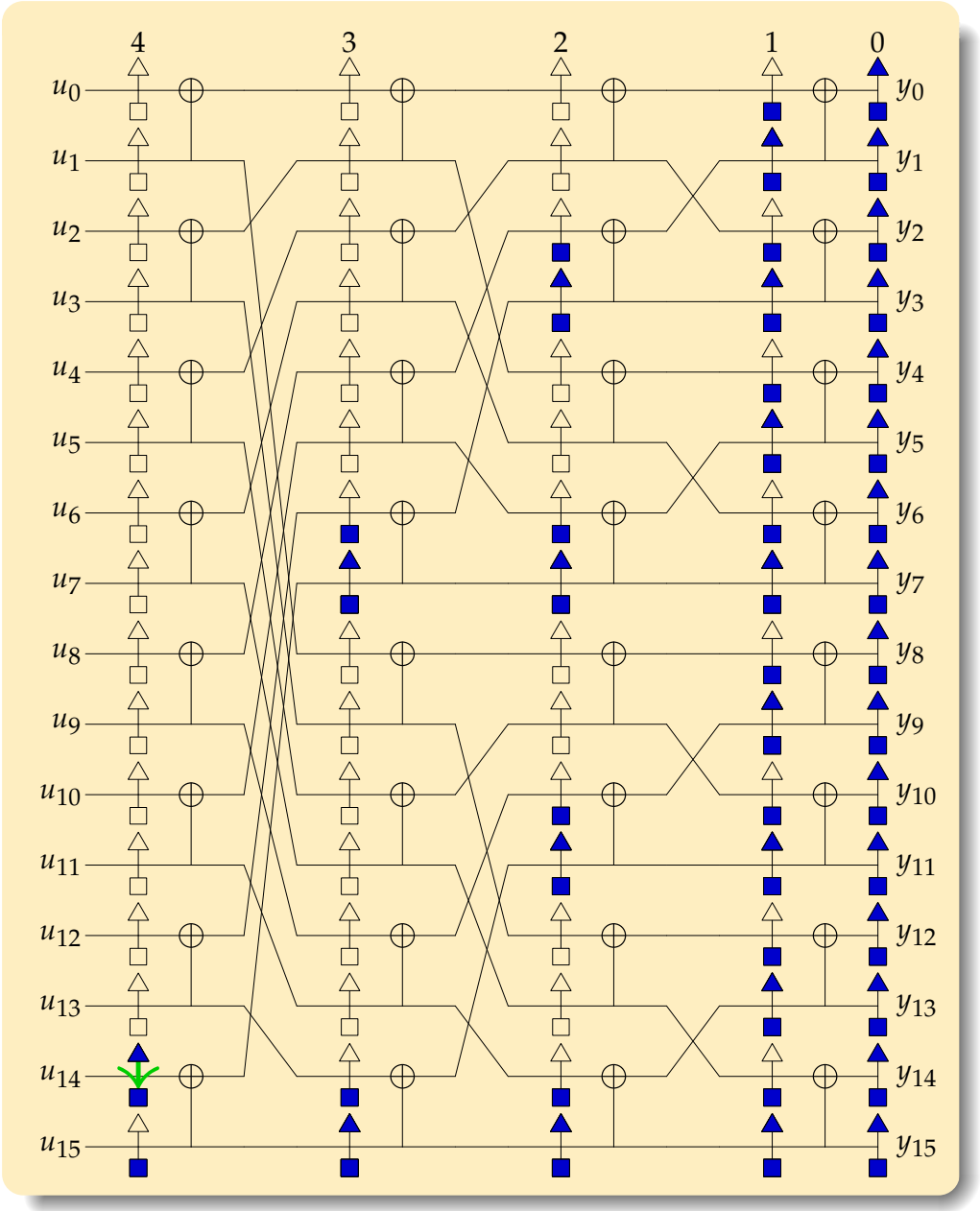
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

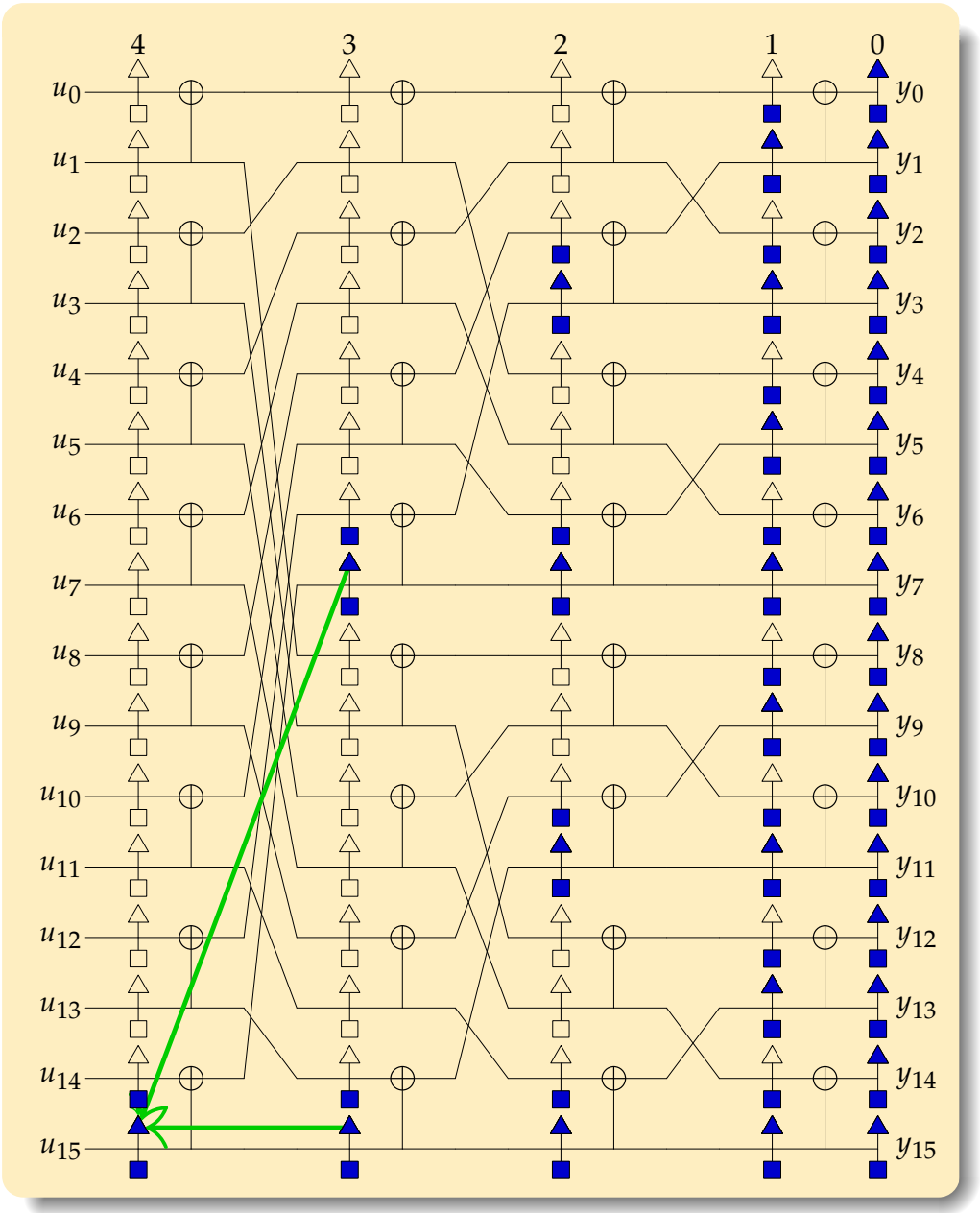
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

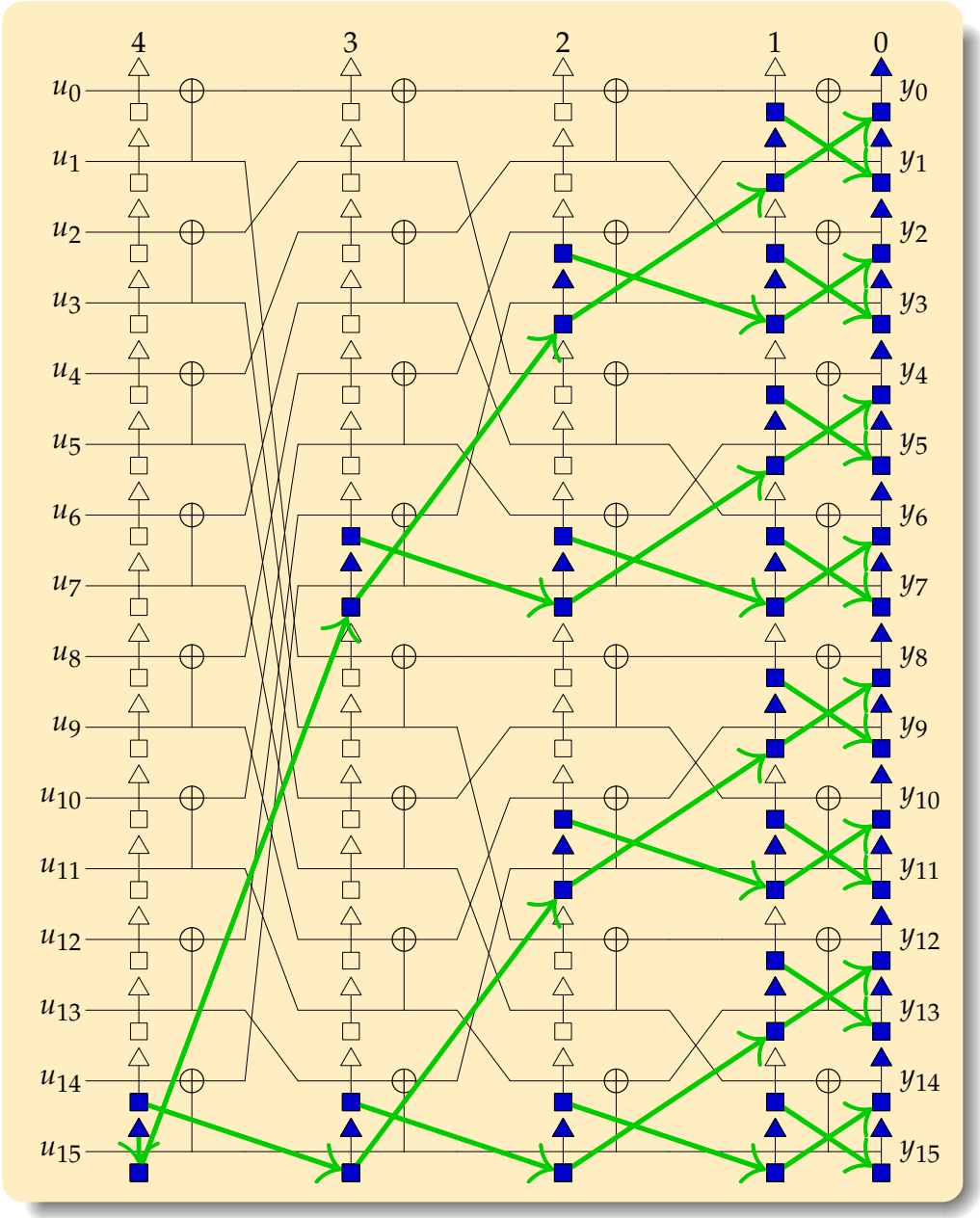
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

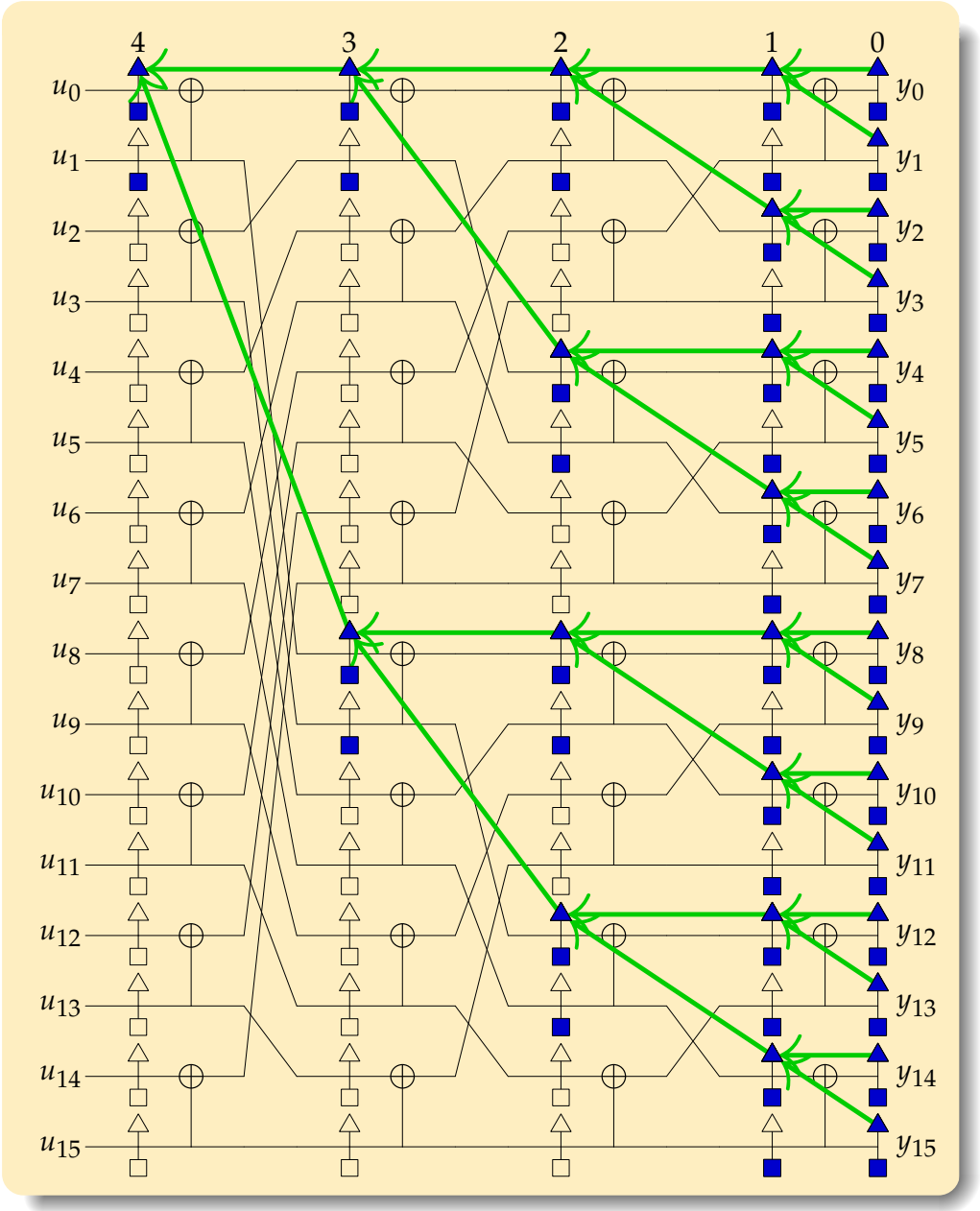
The memory needed to hold the variables at level t is $O(n/2^t)$.



A larger example

Key point

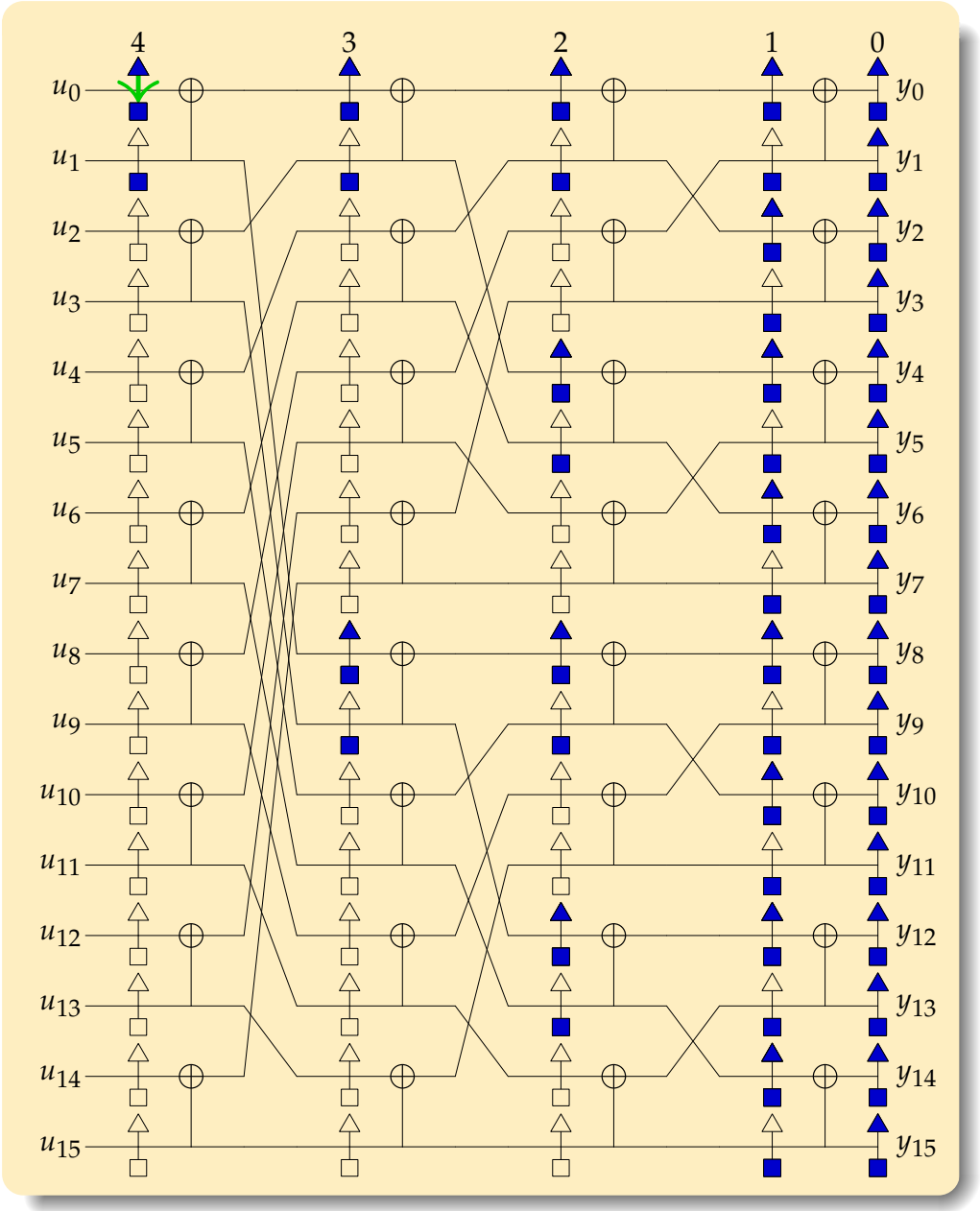
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

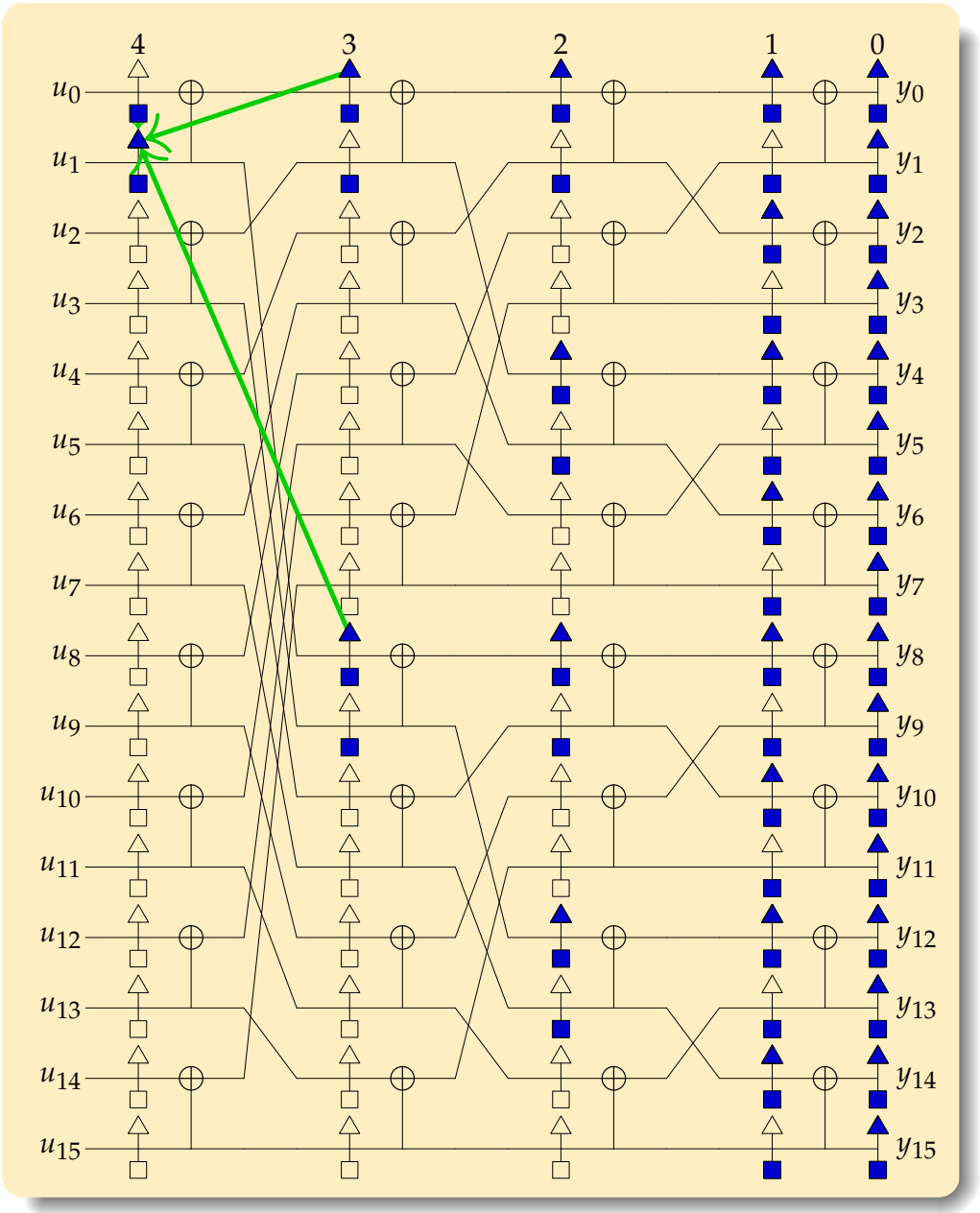
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

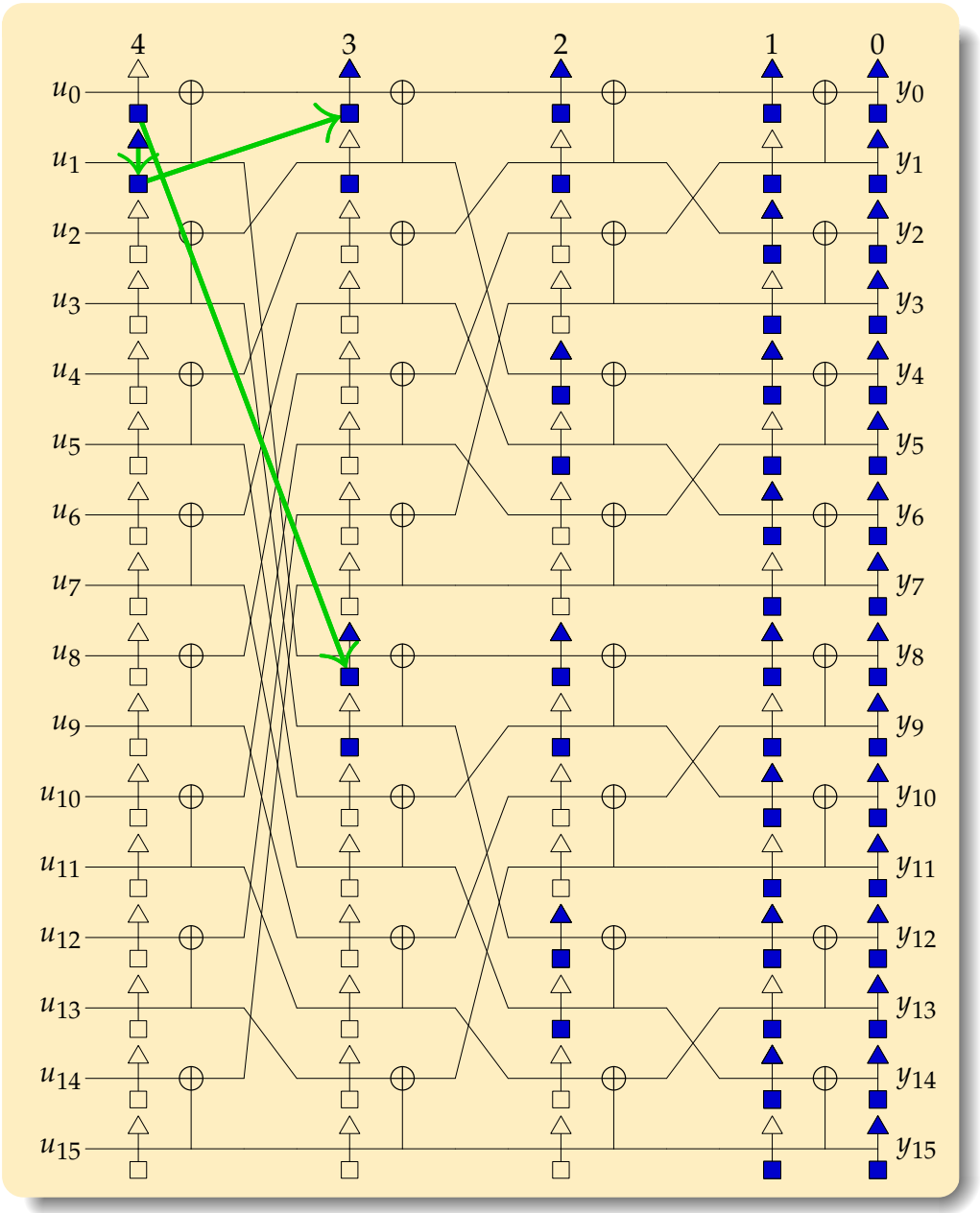
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

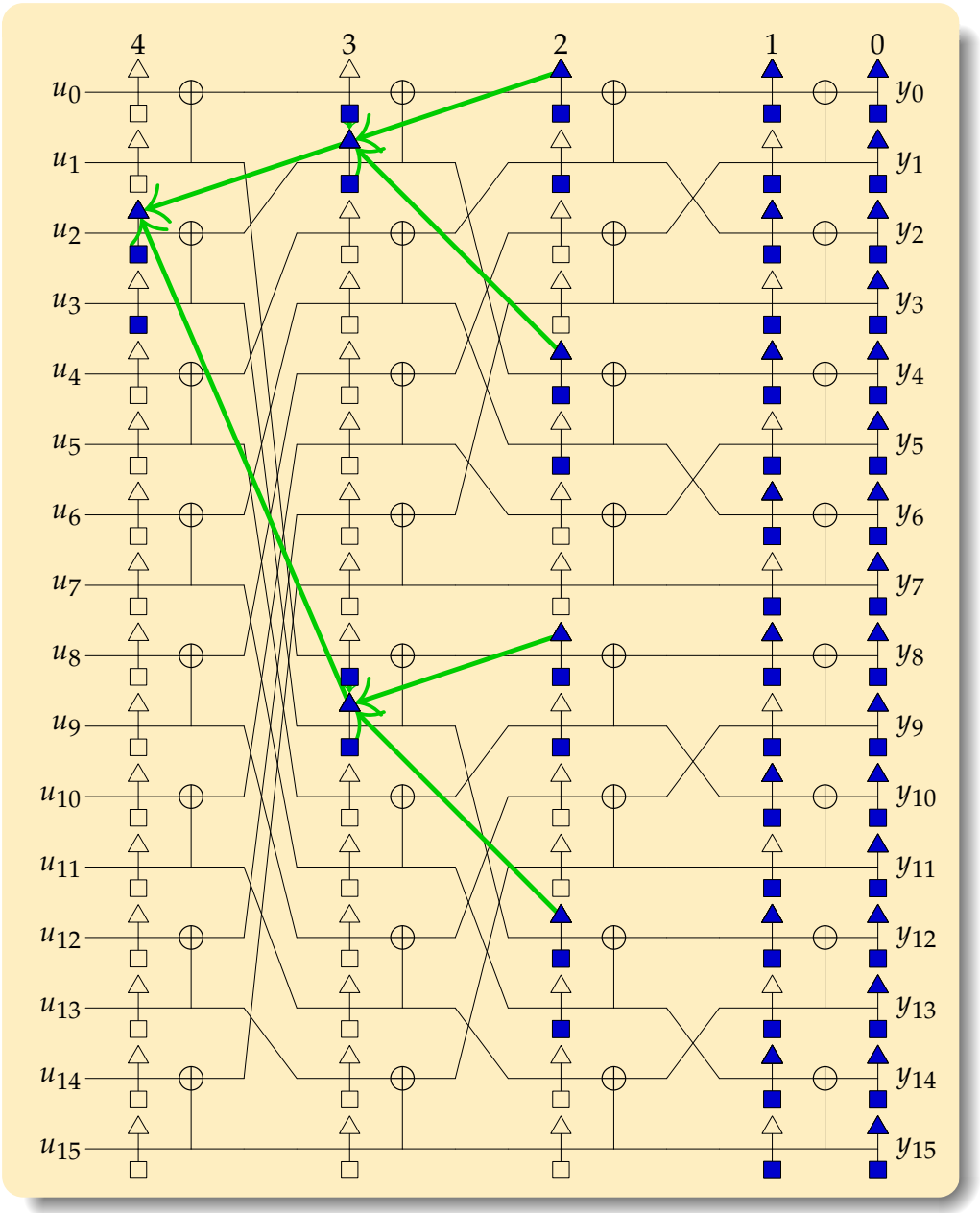
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

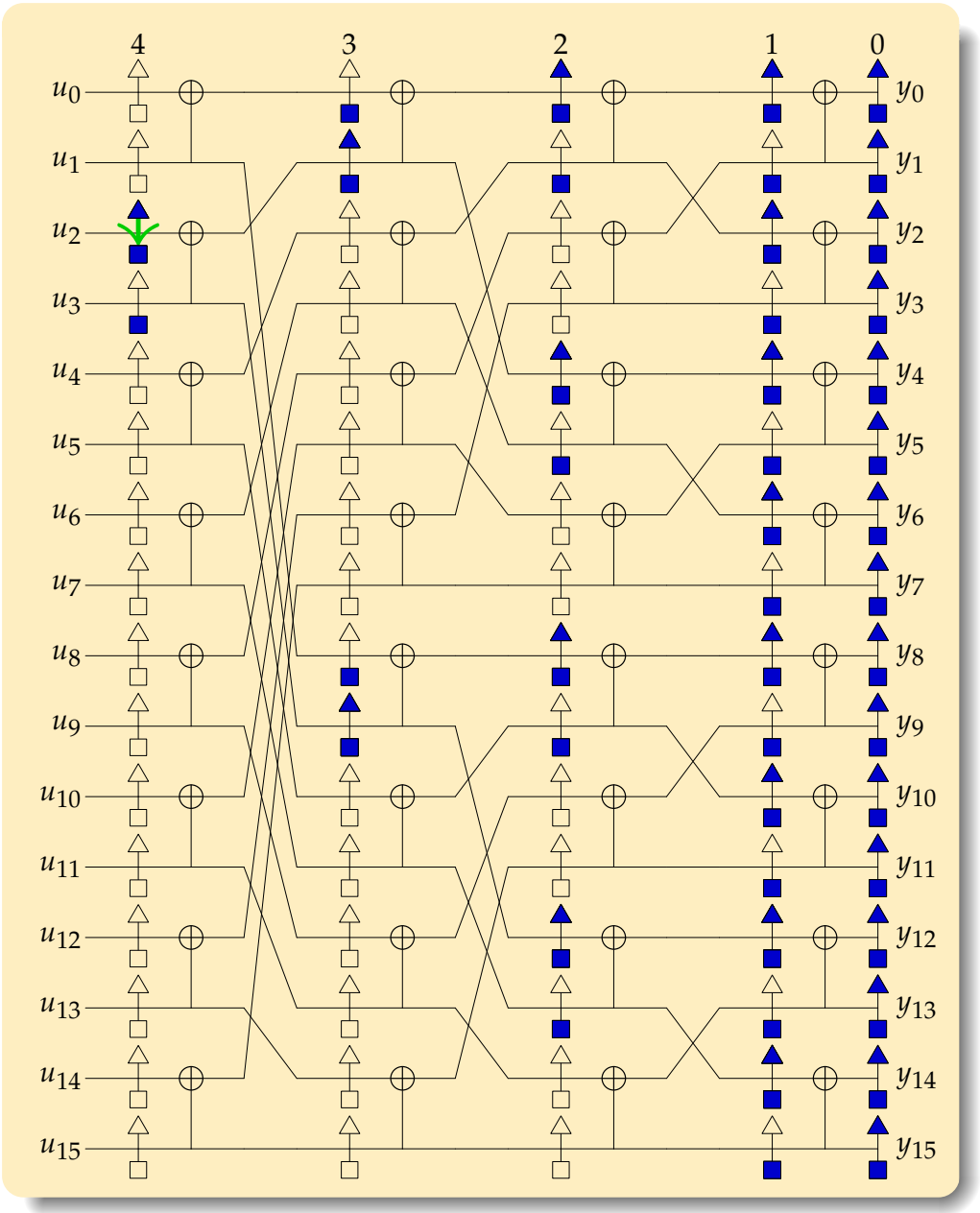
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

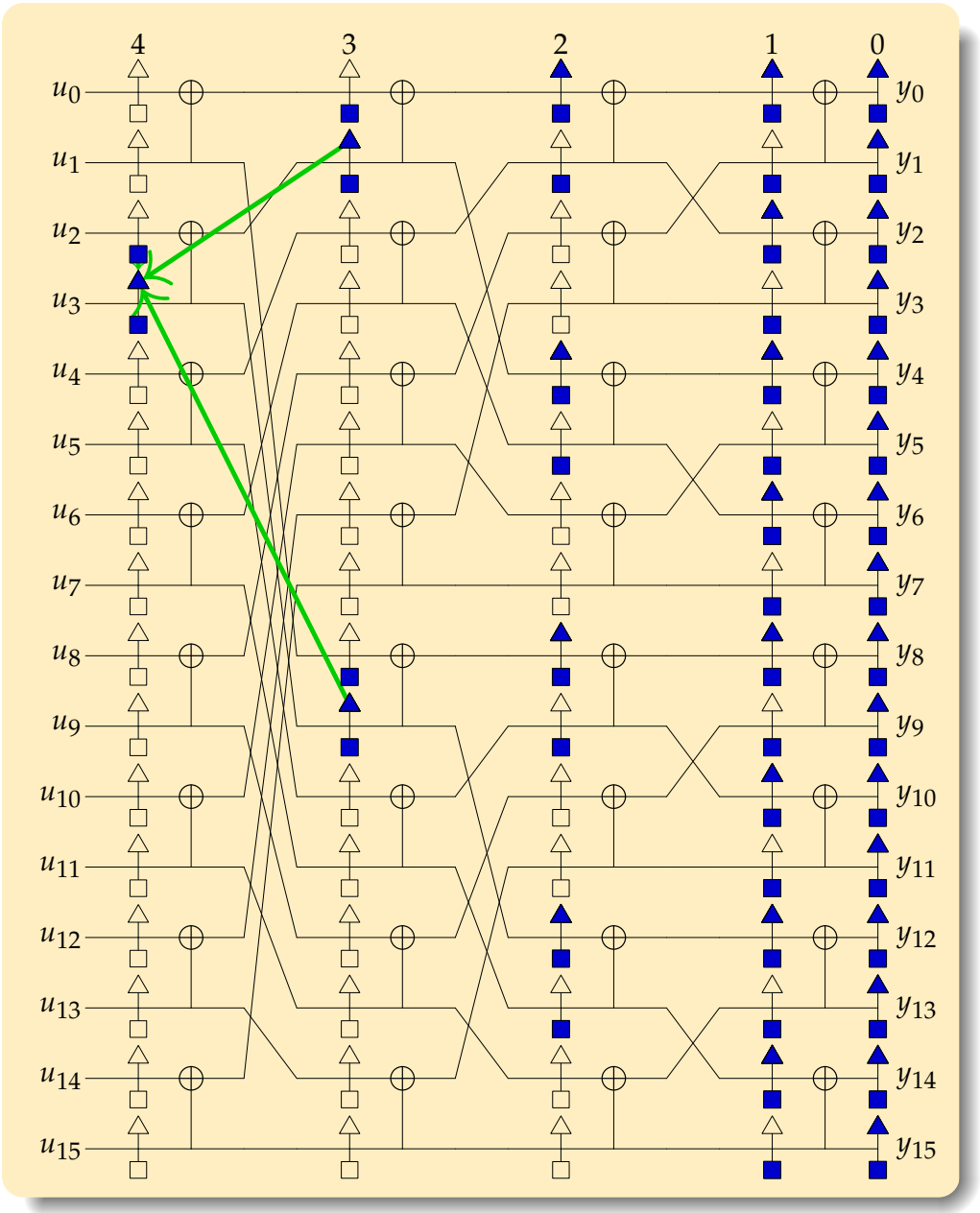
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

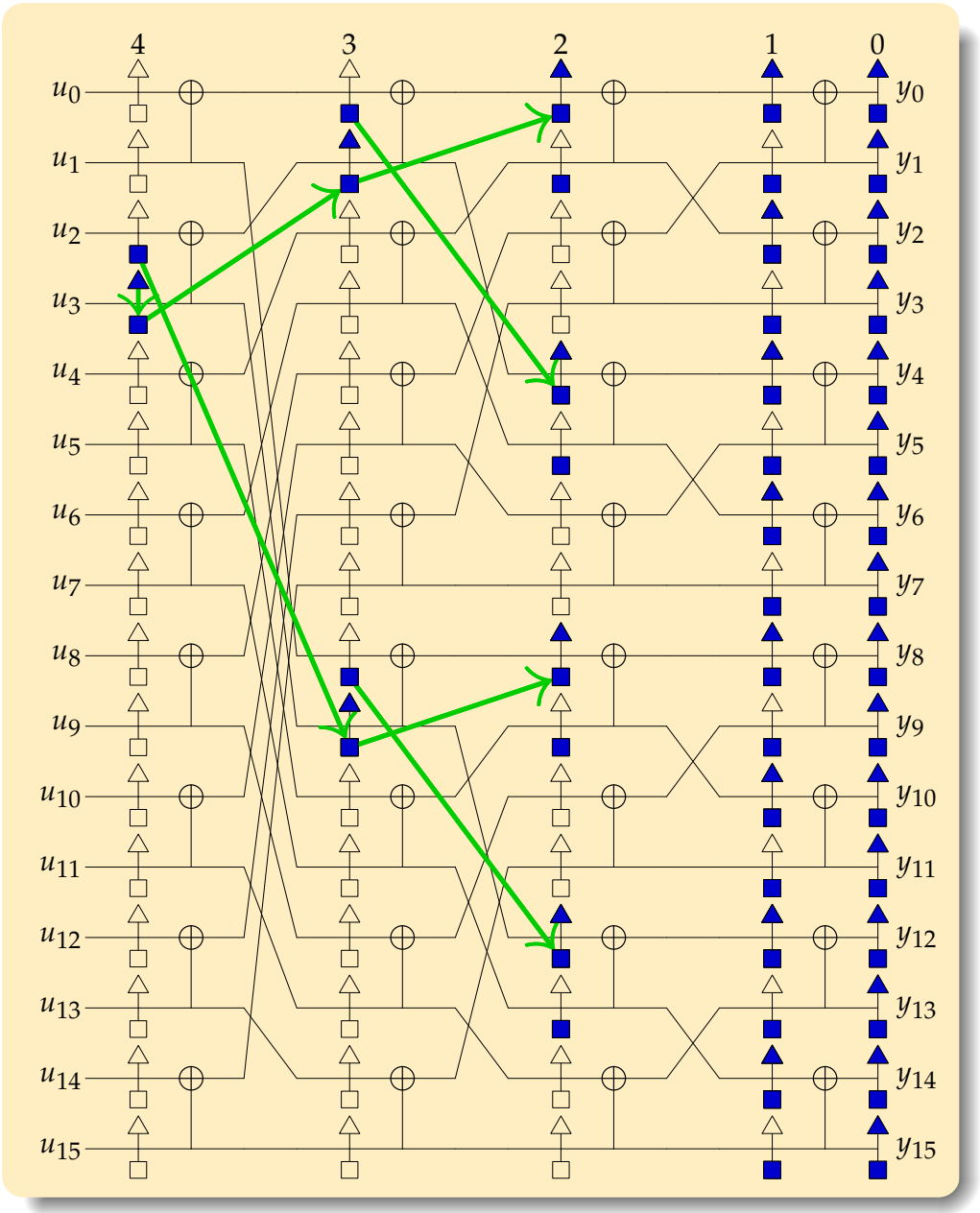
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

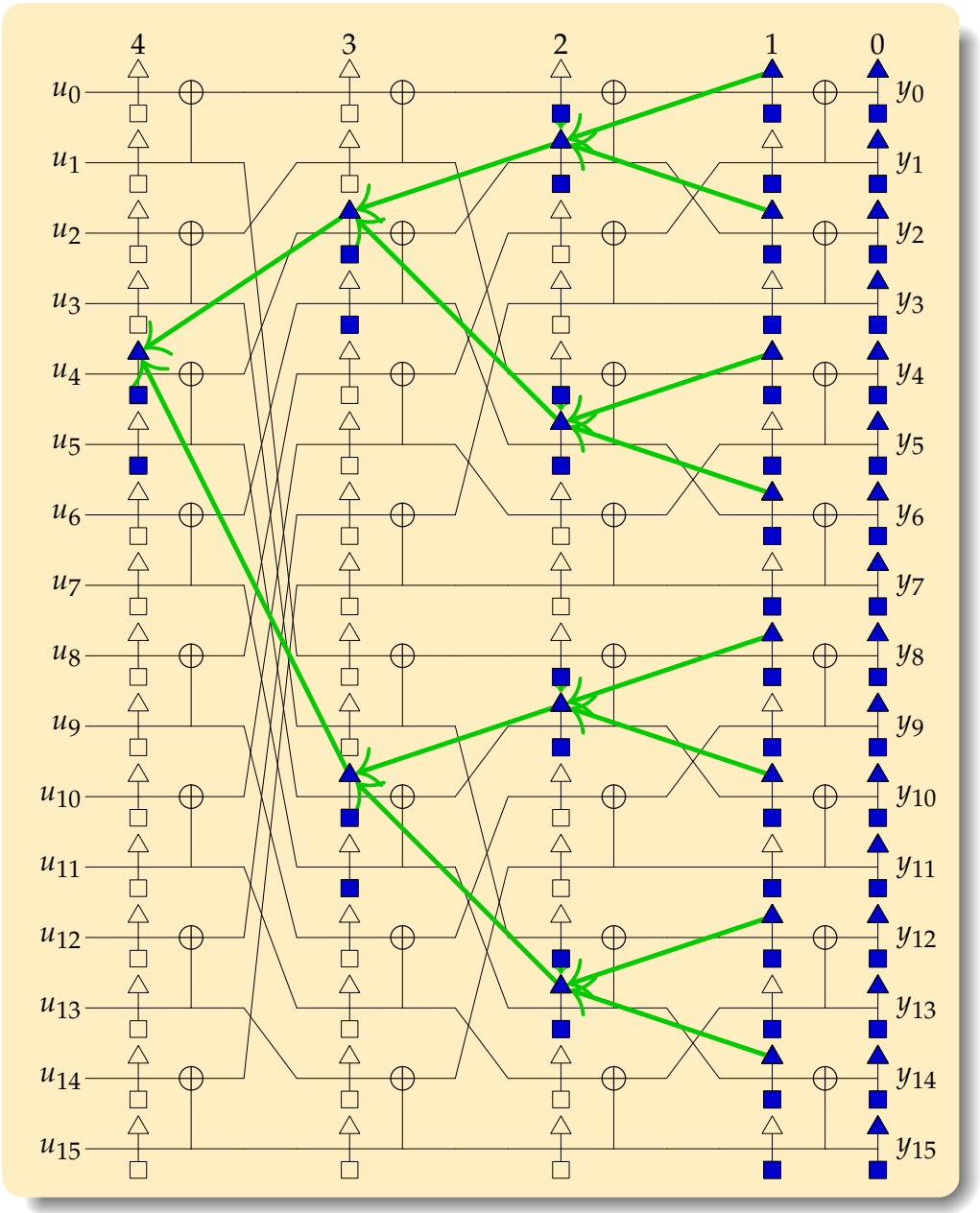
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

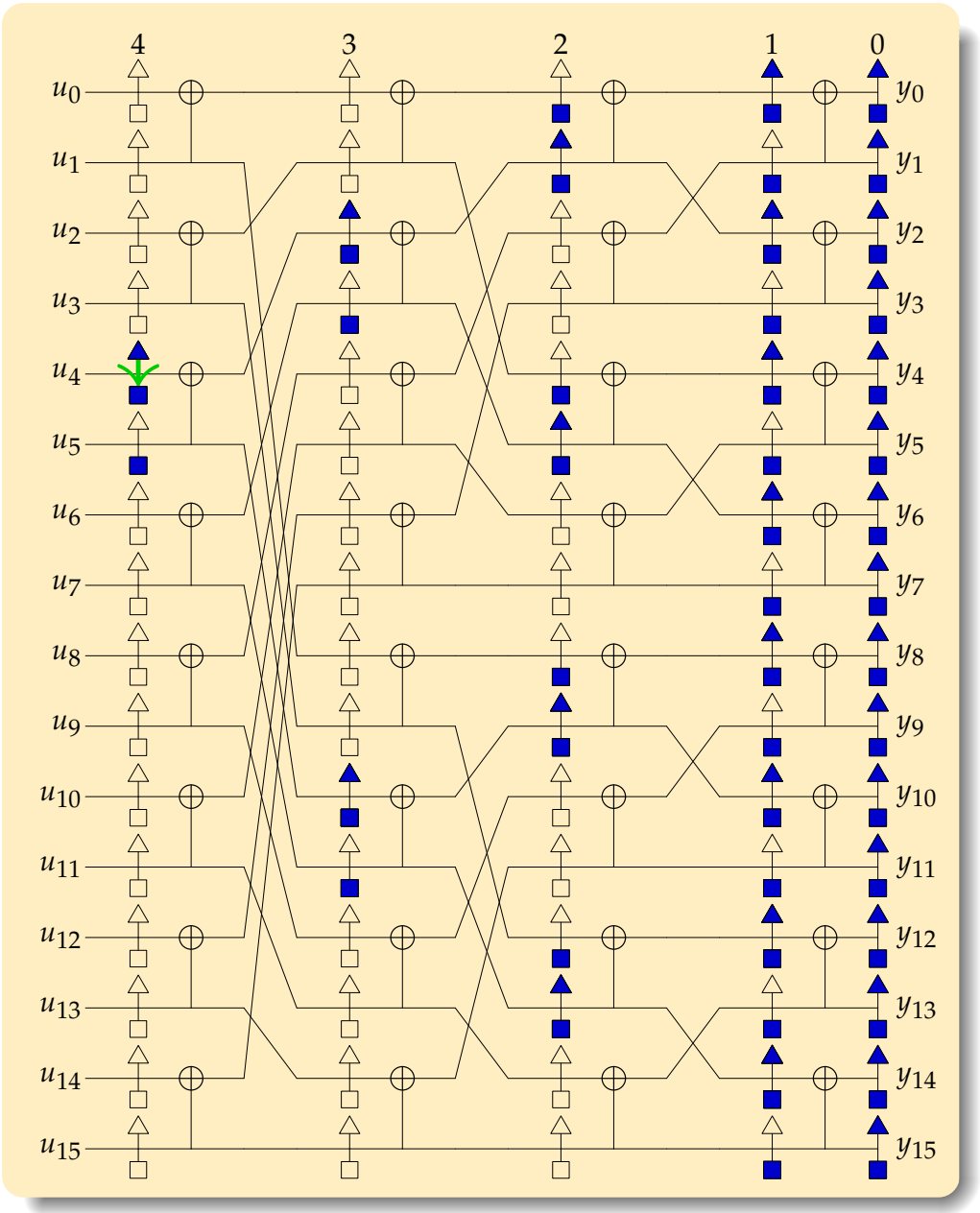
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

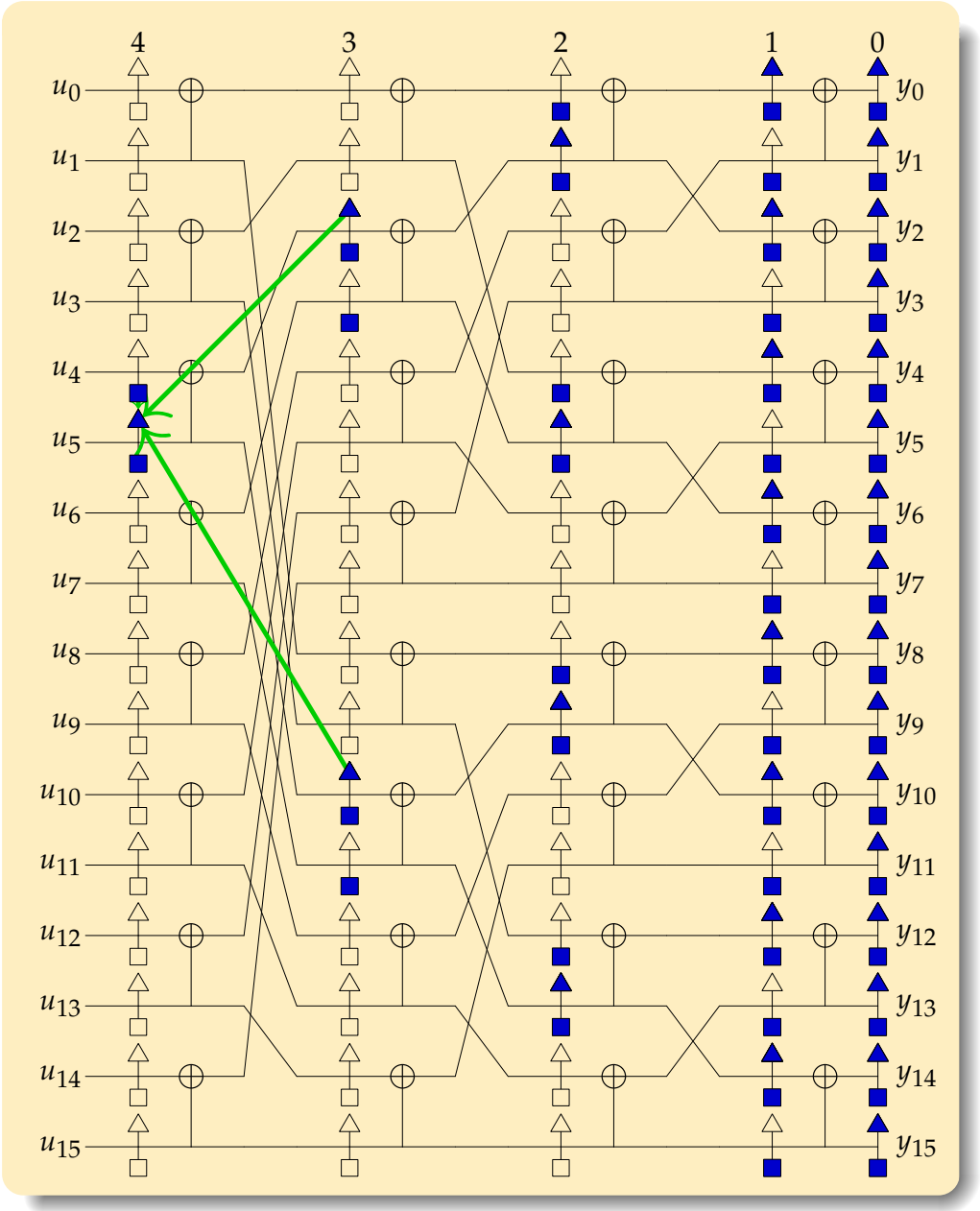
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

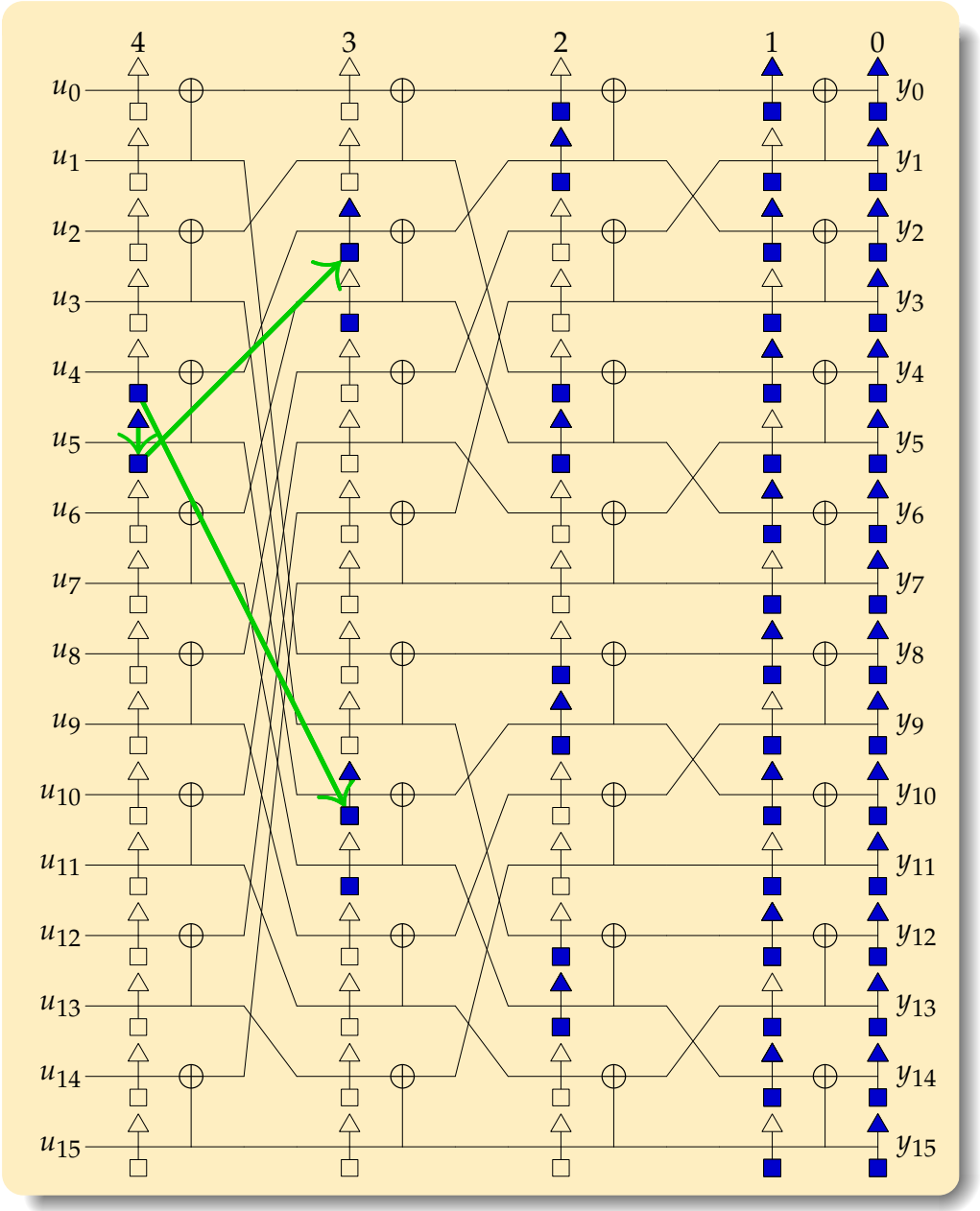
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

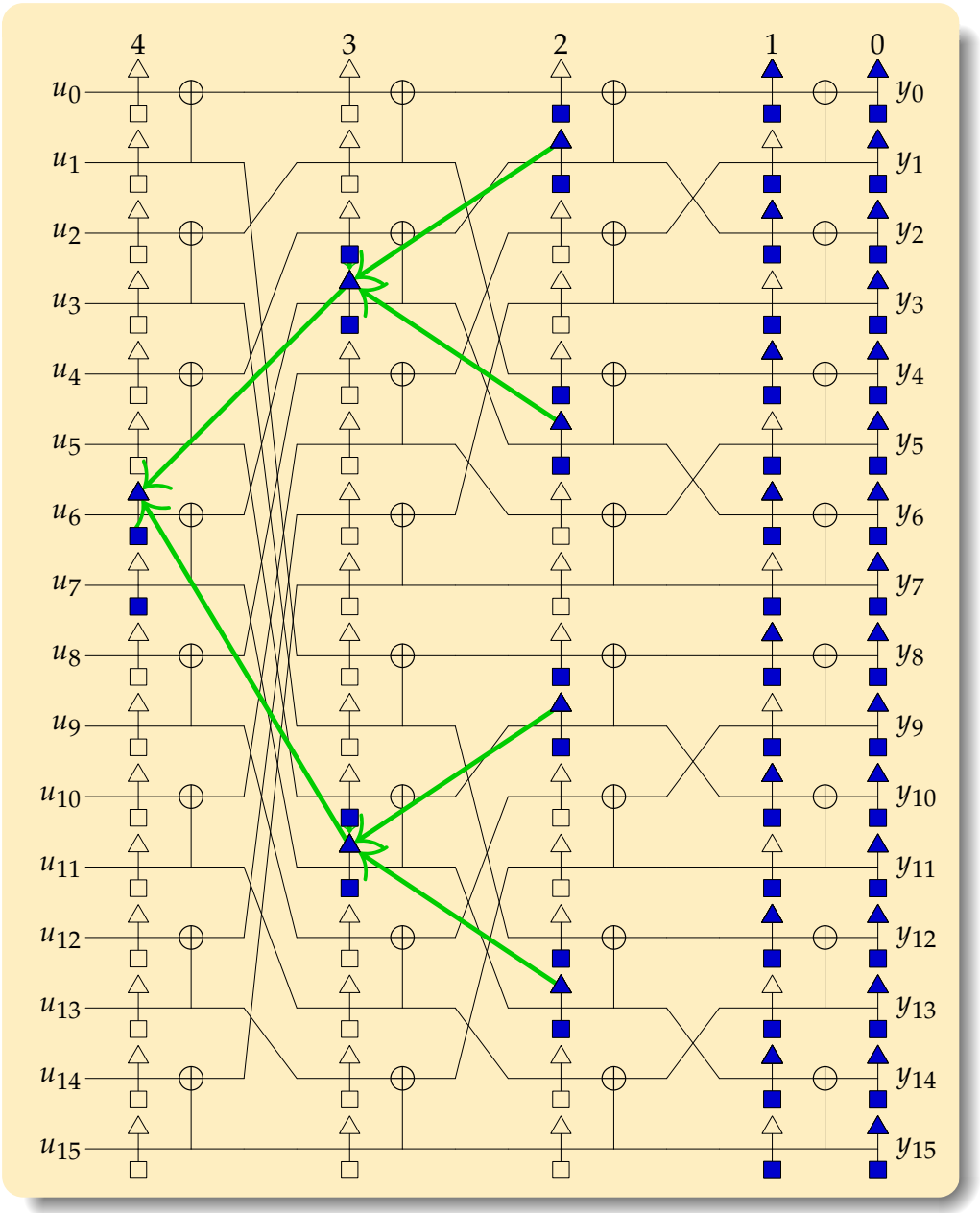
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

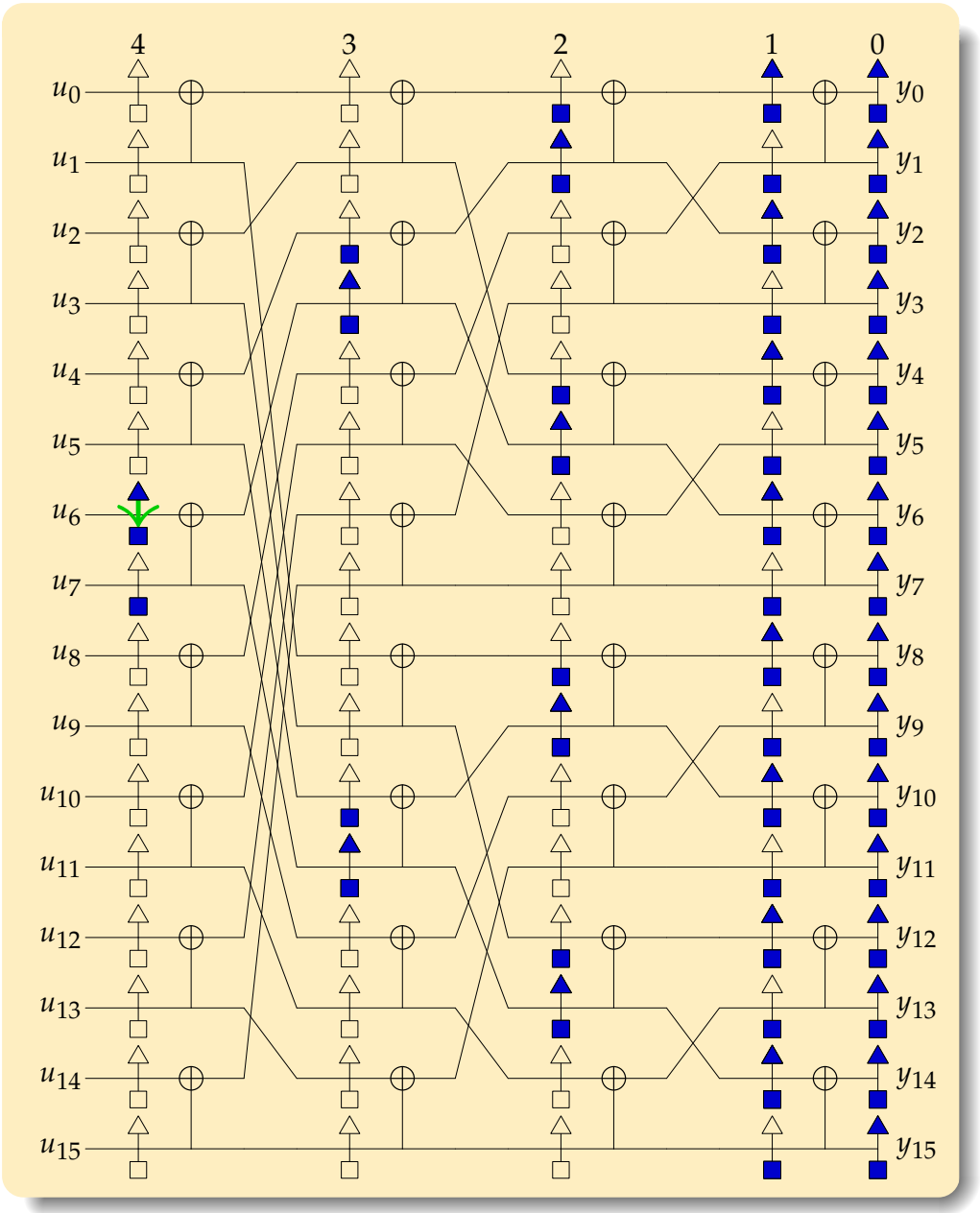
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

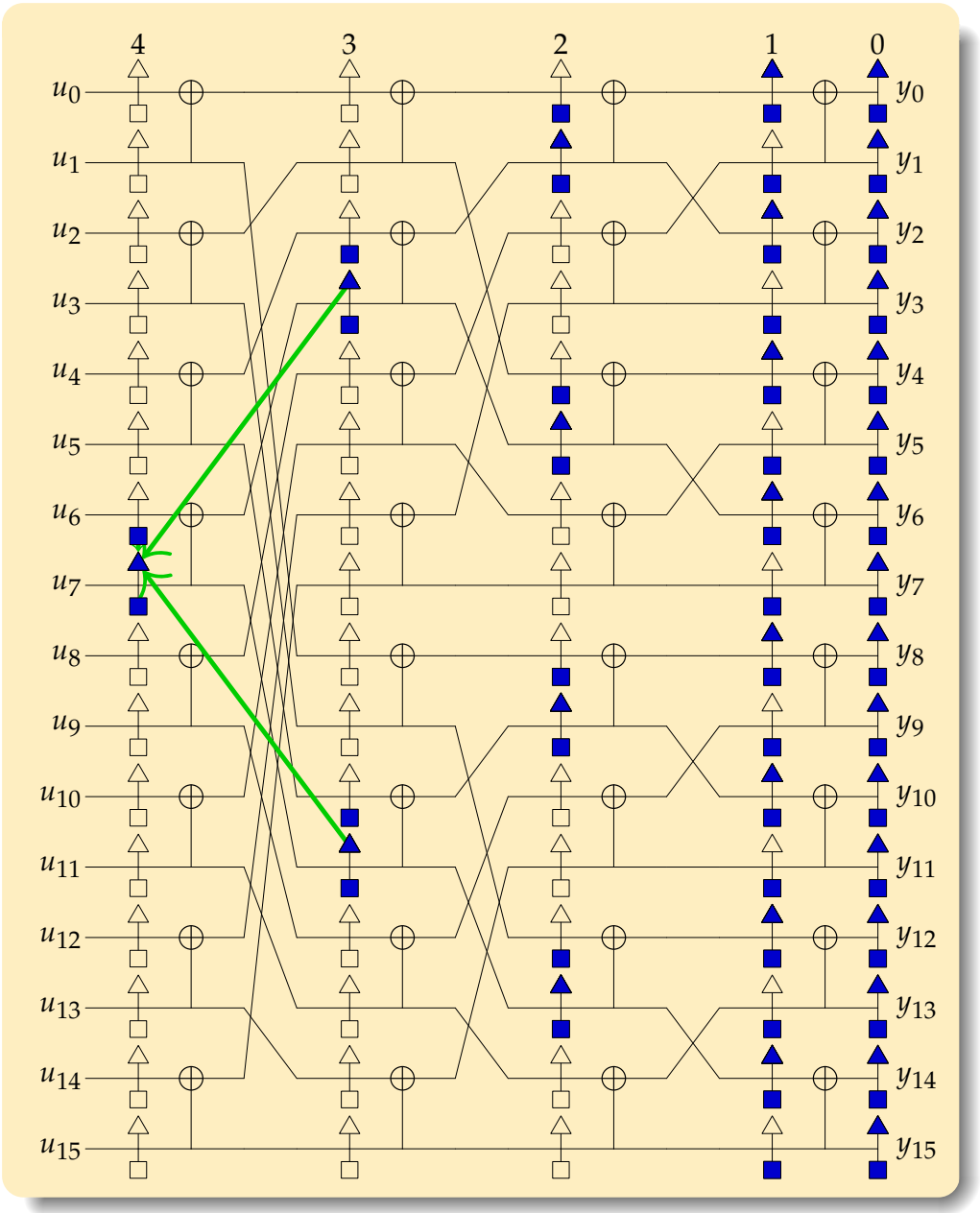
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

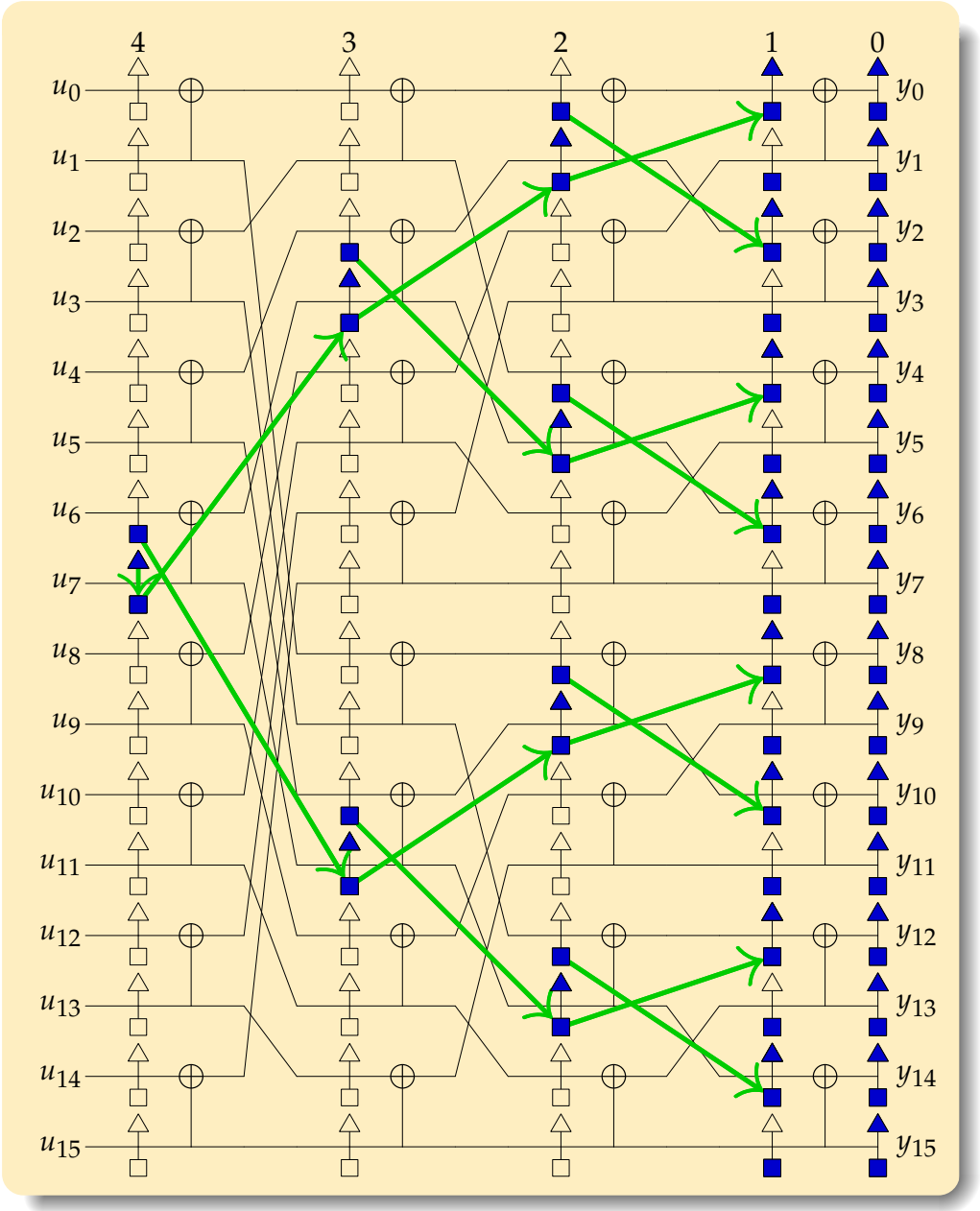
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

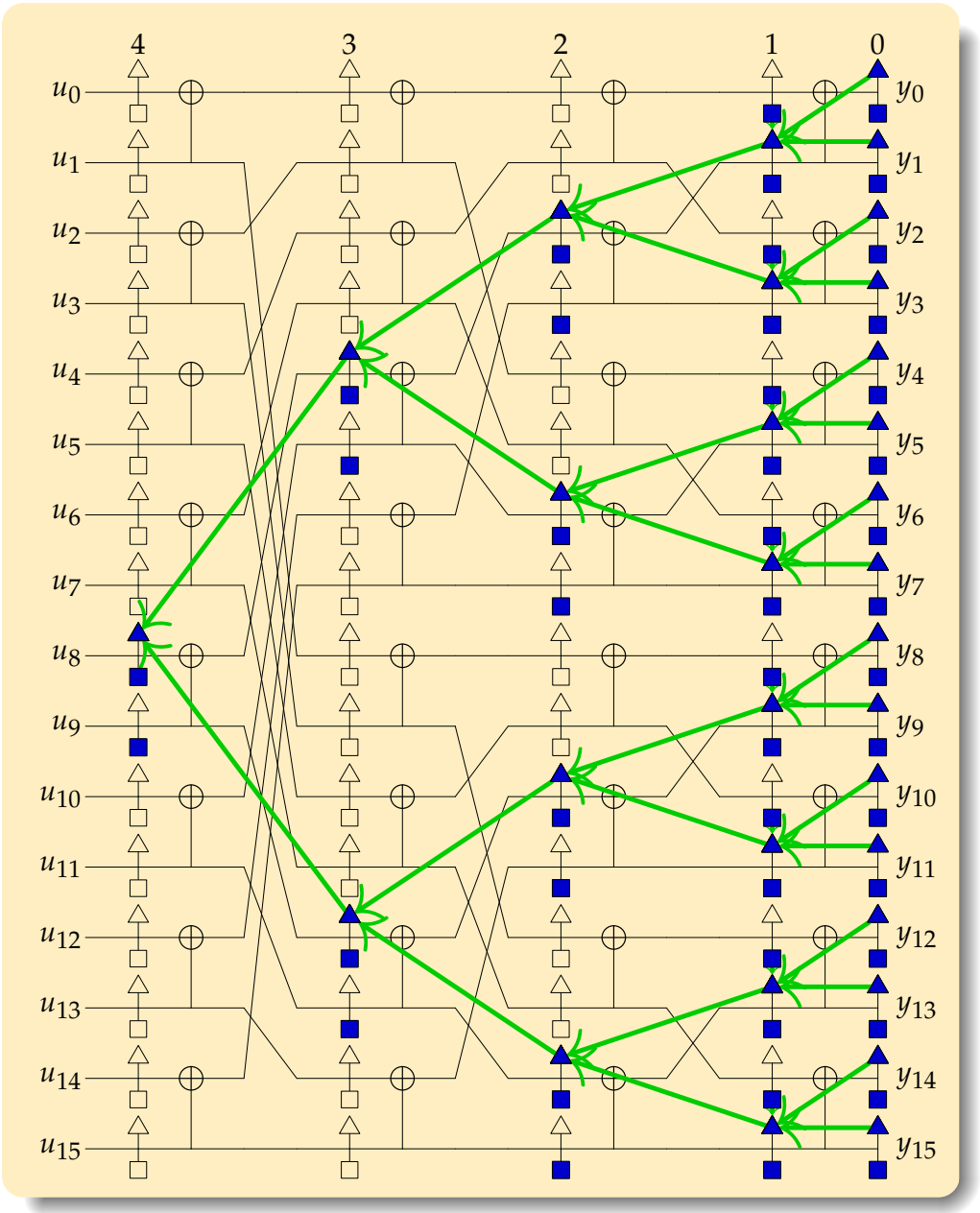
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

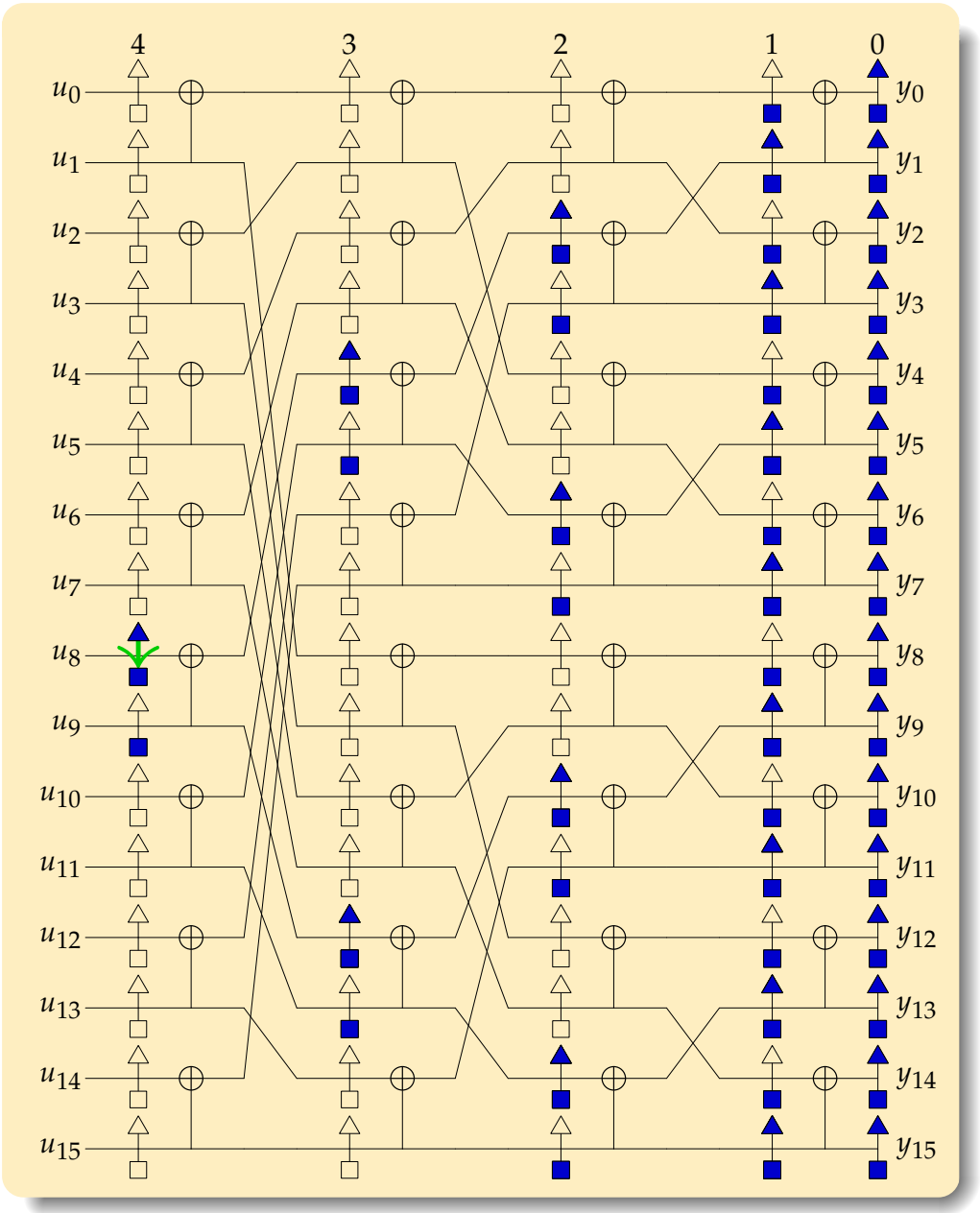
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

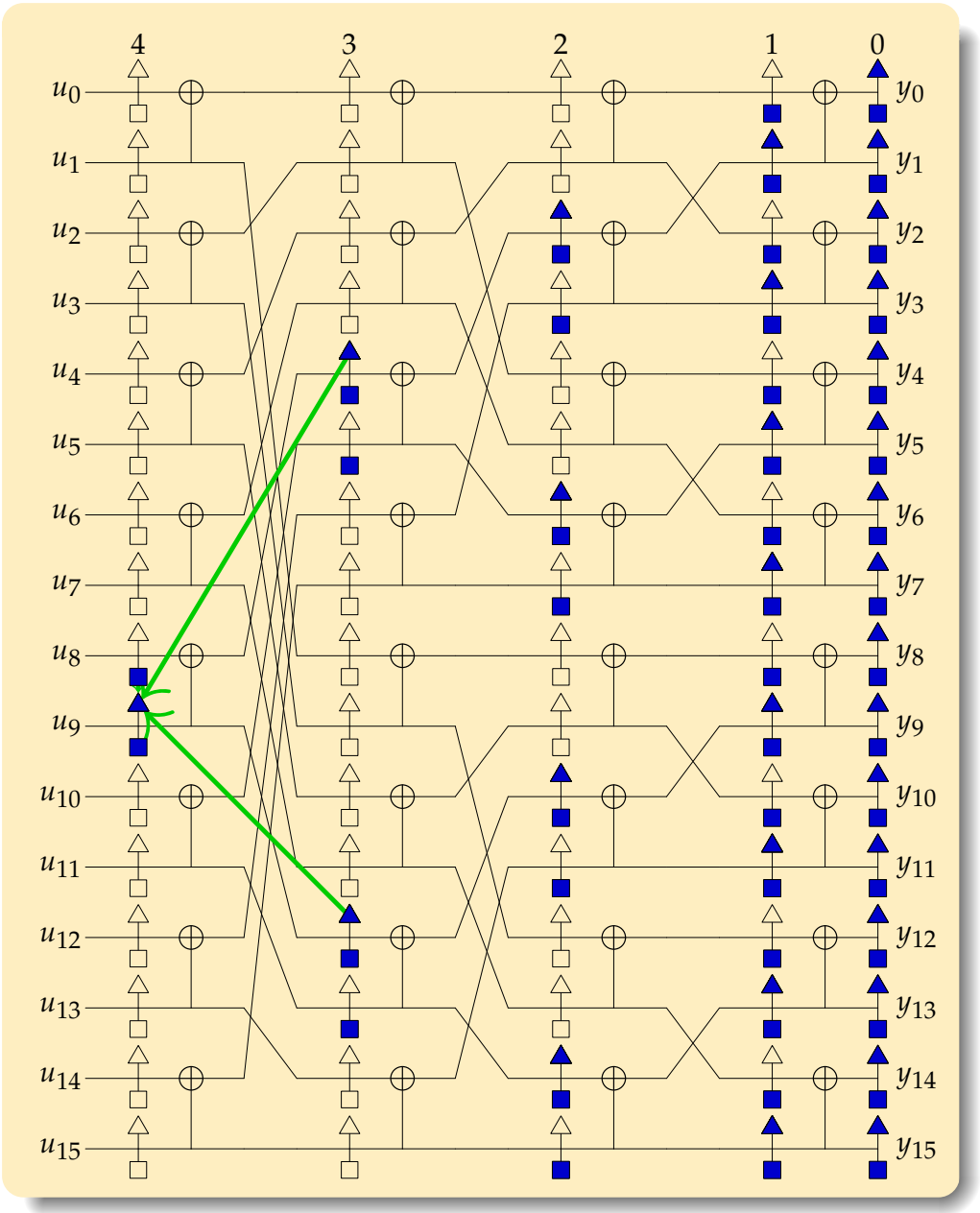
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

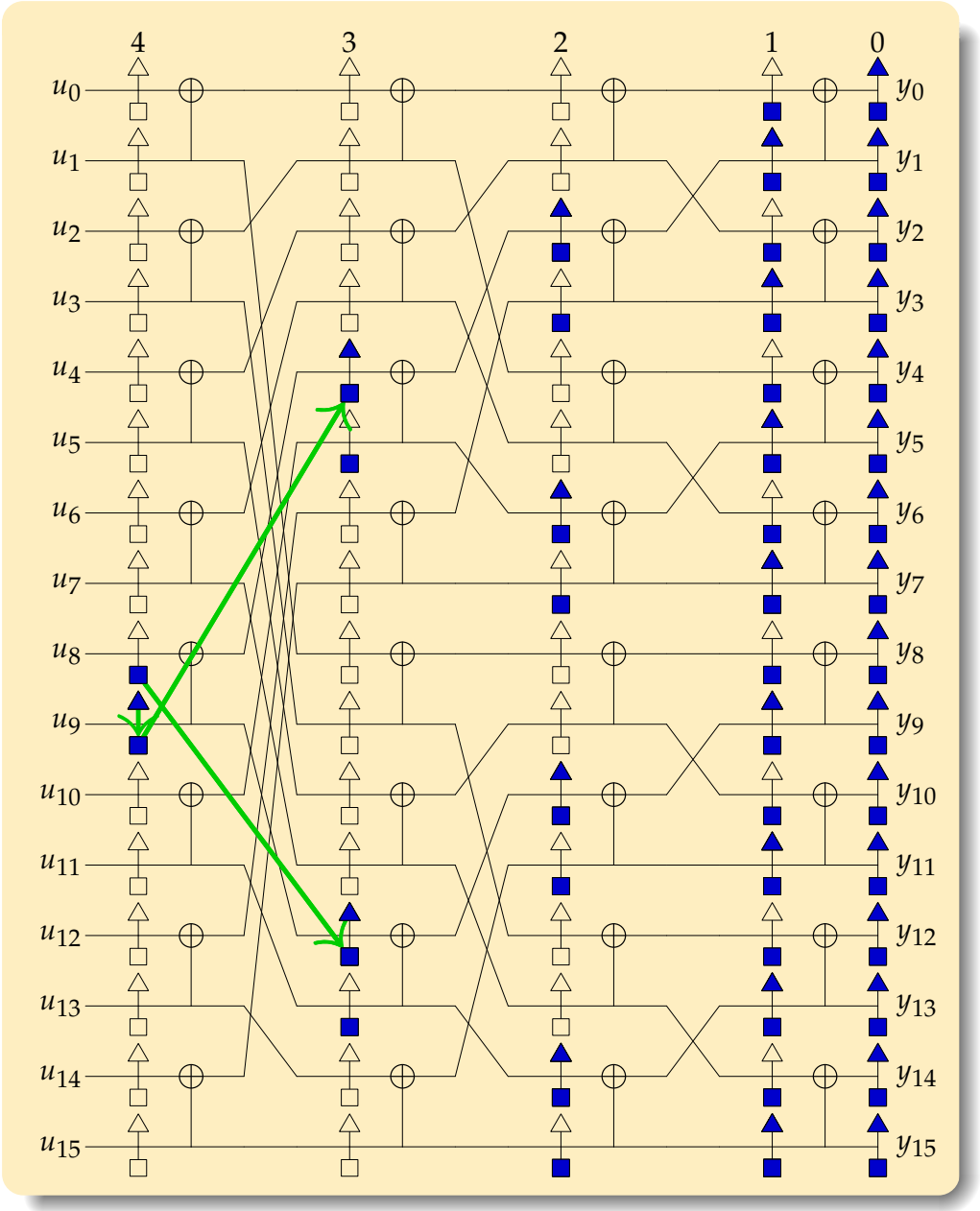
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

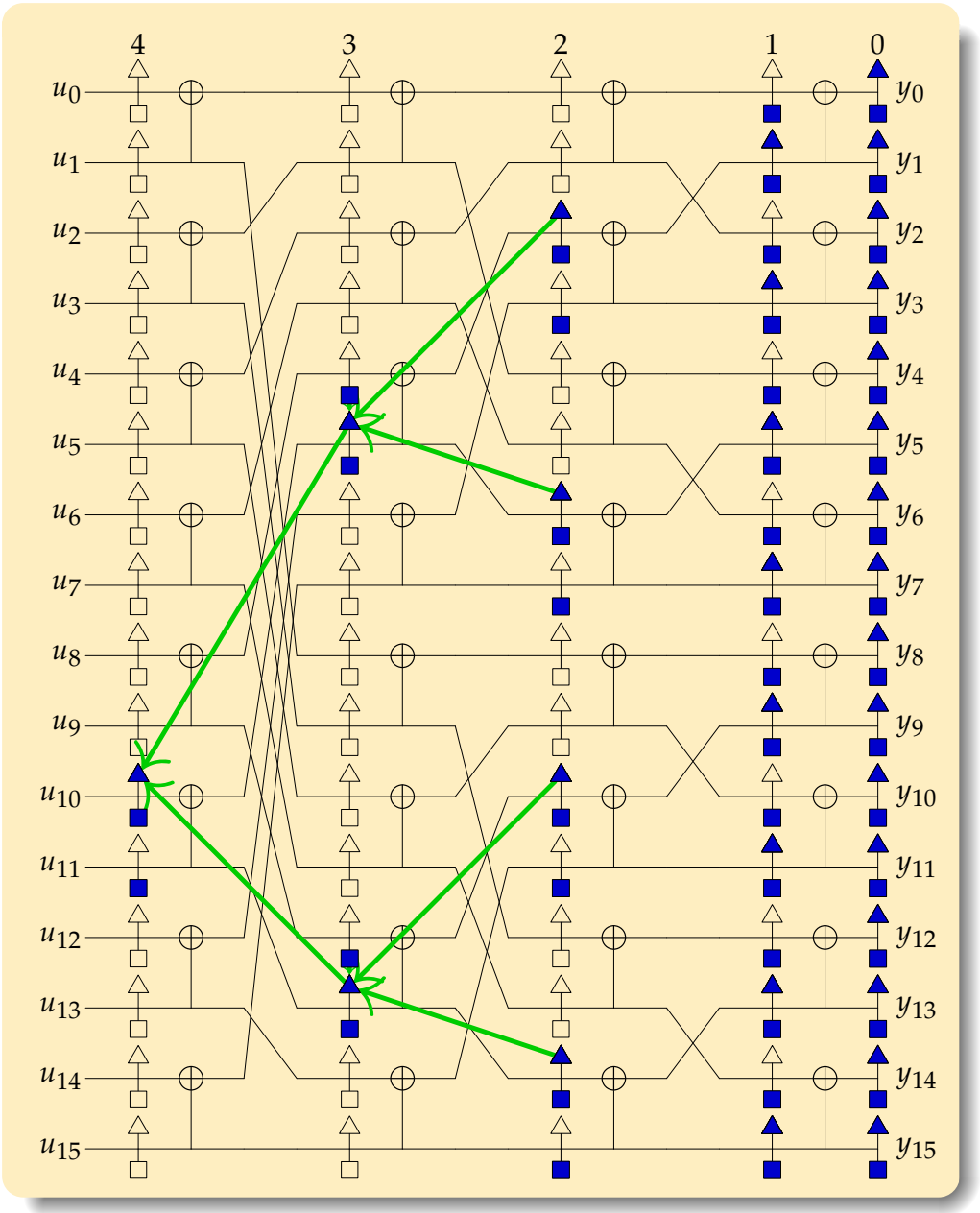
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

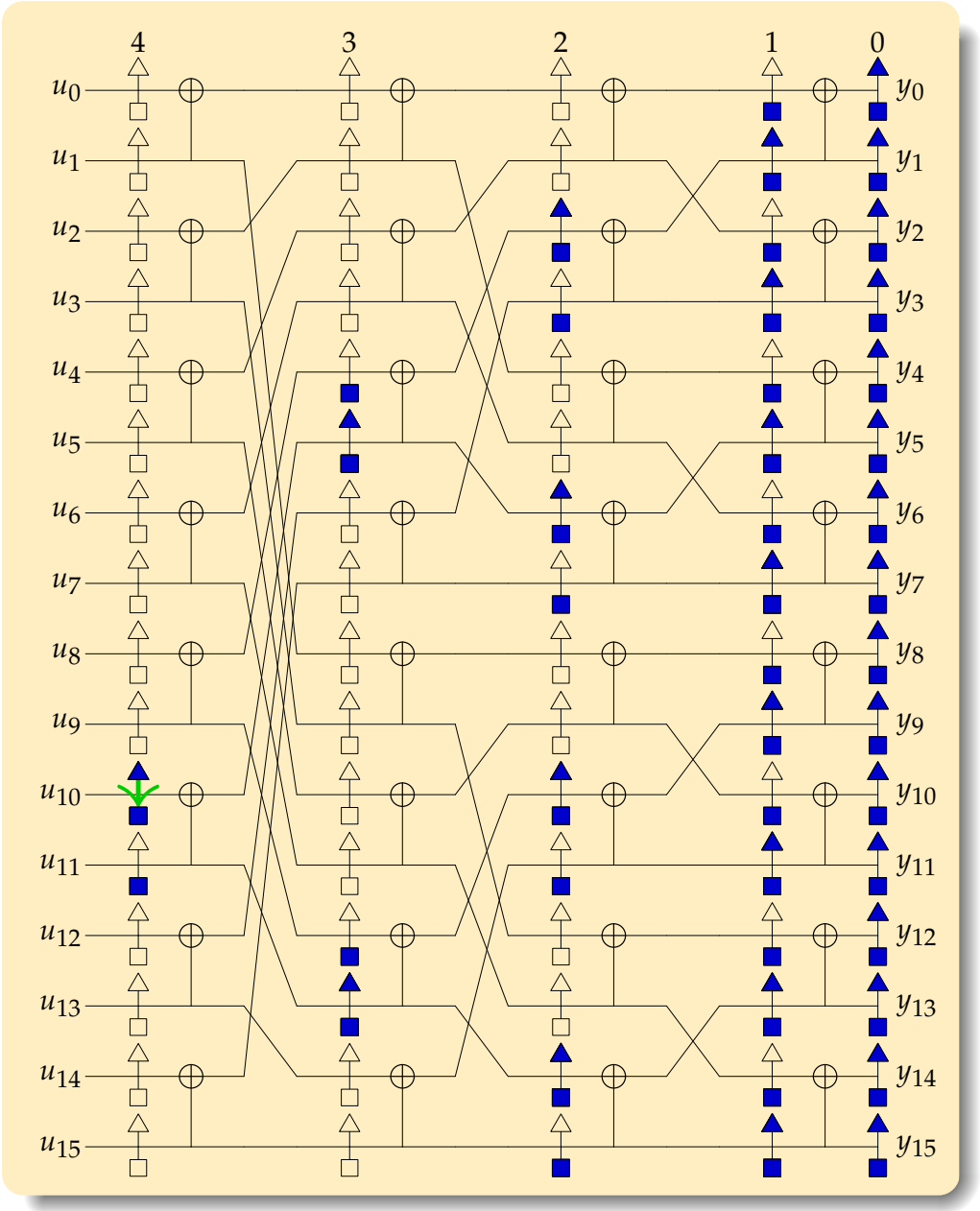
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

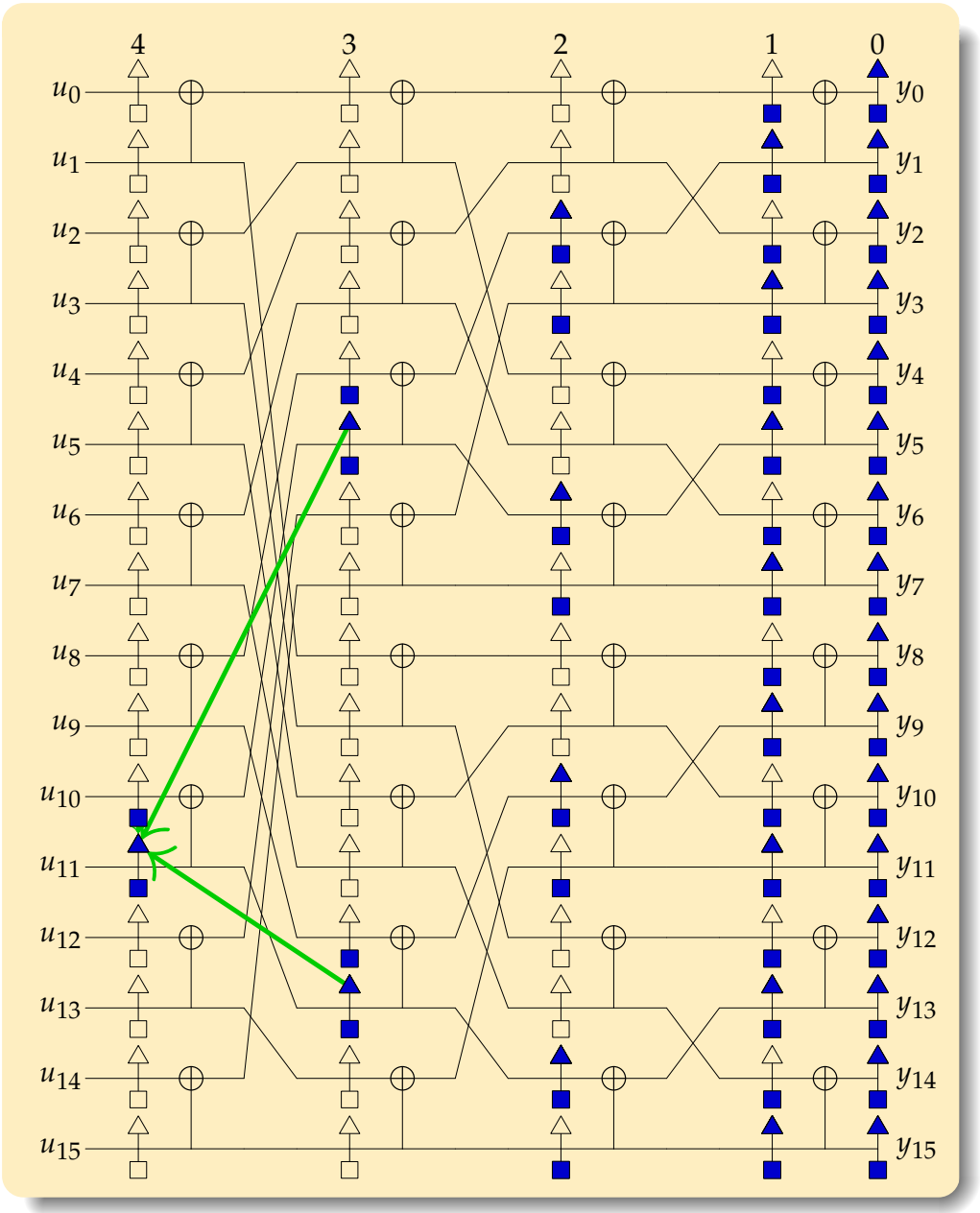
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

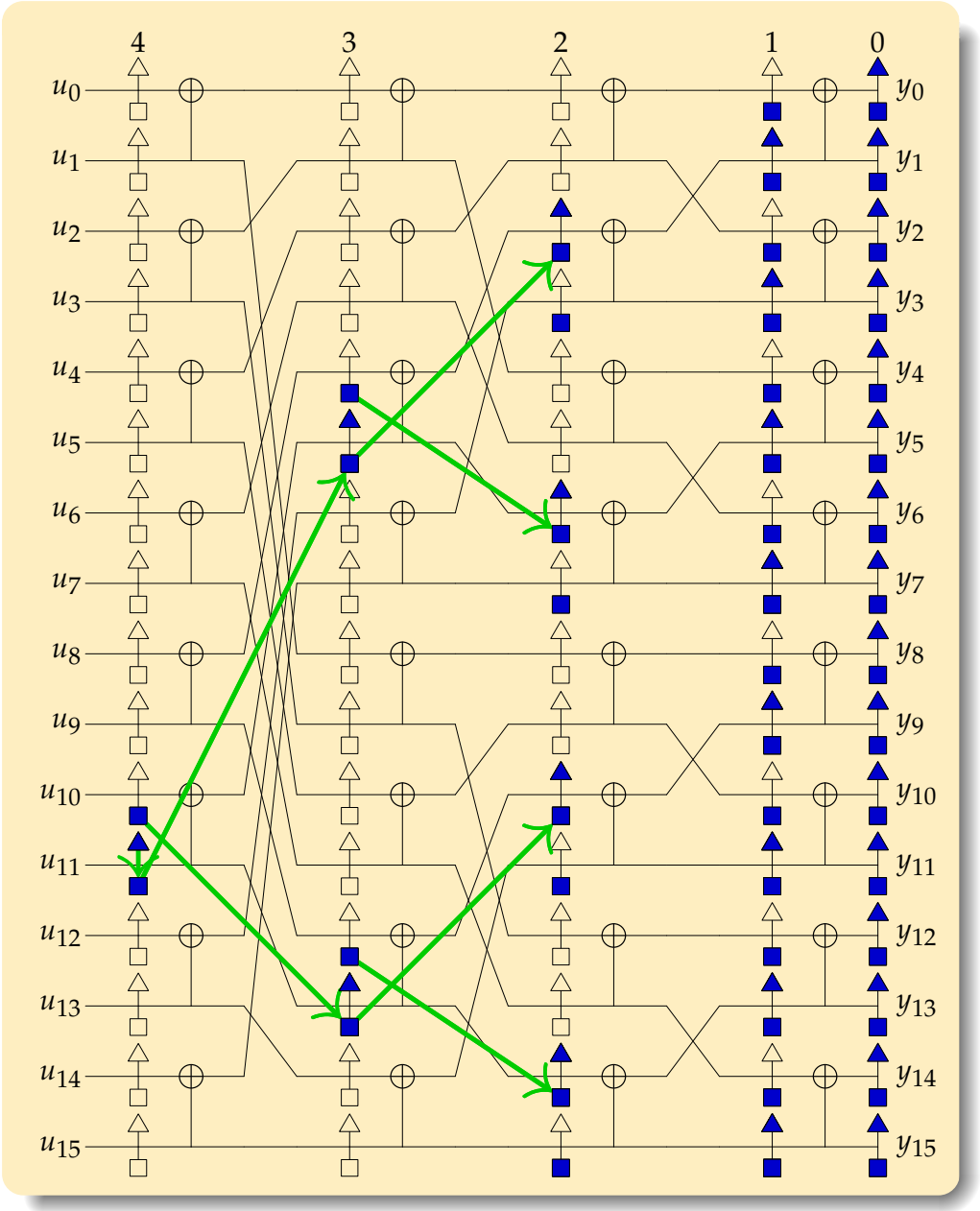
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

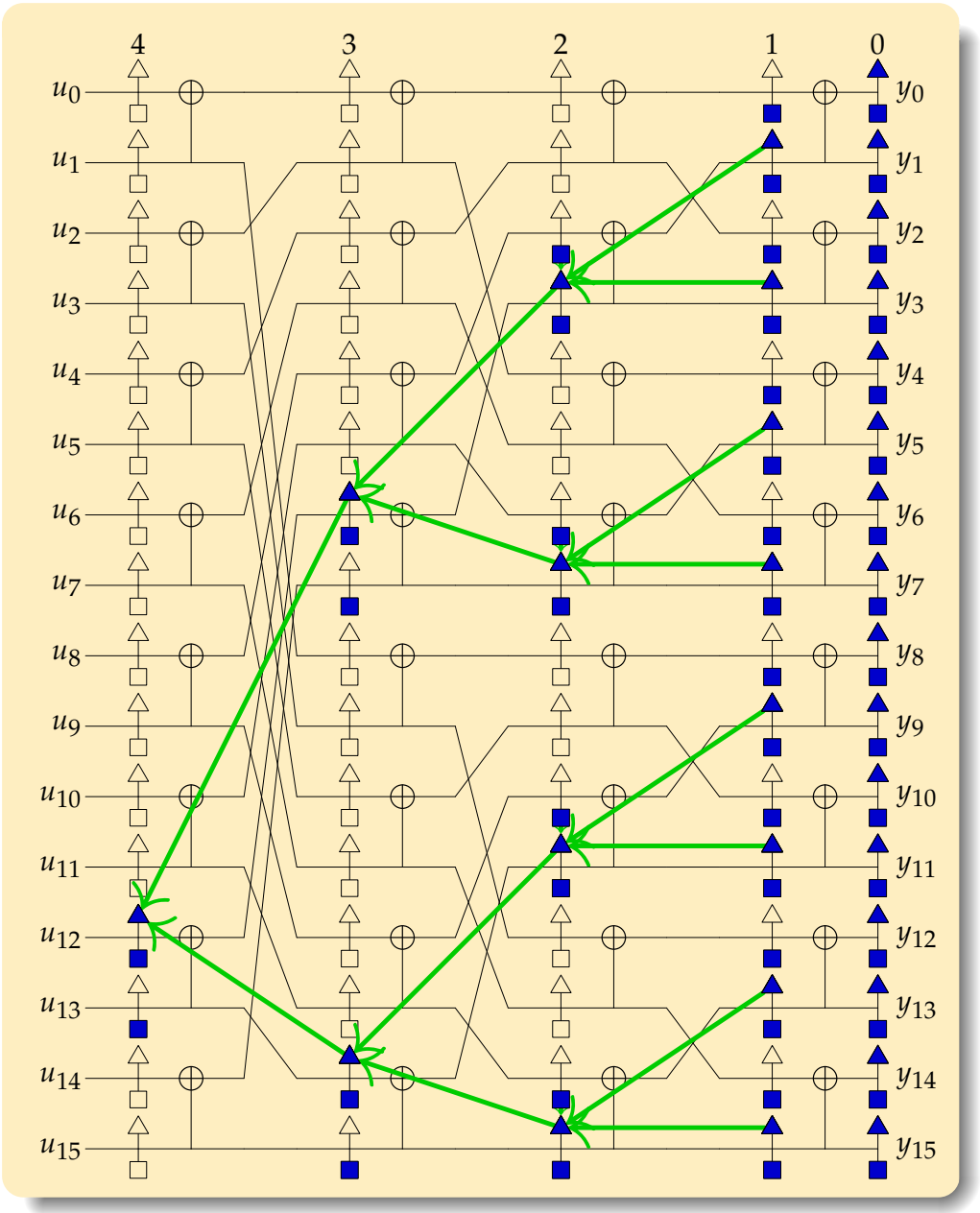
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

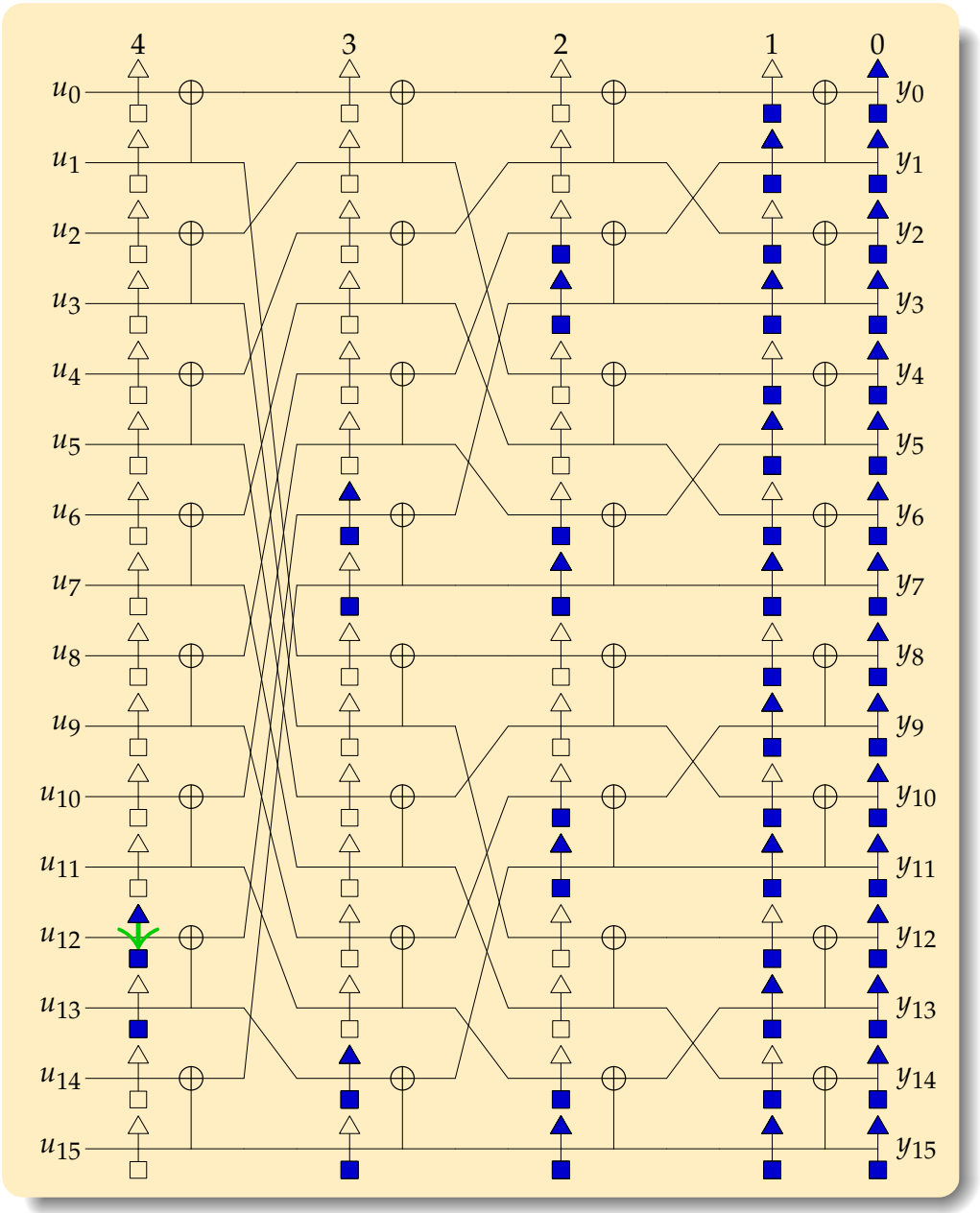
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

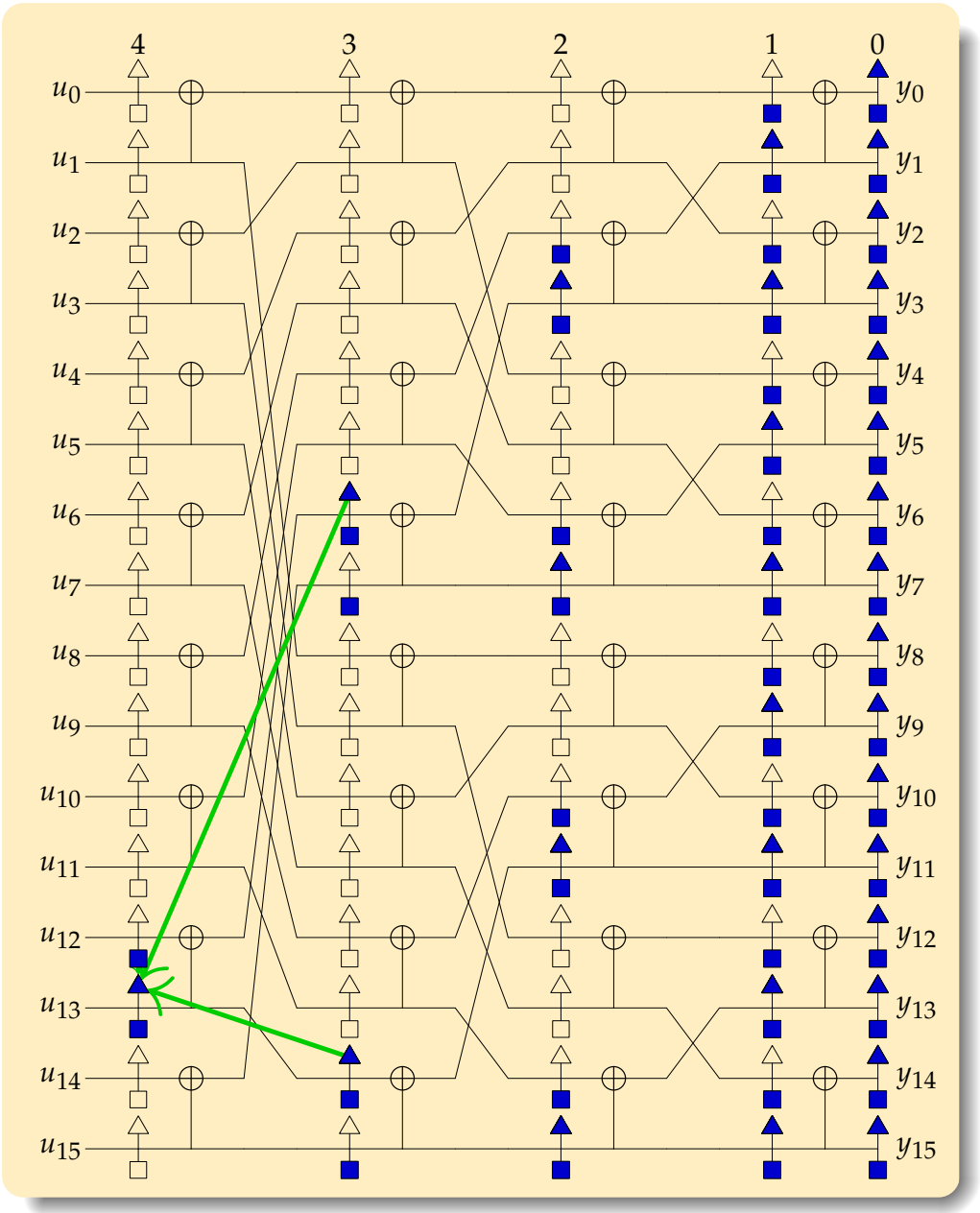
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

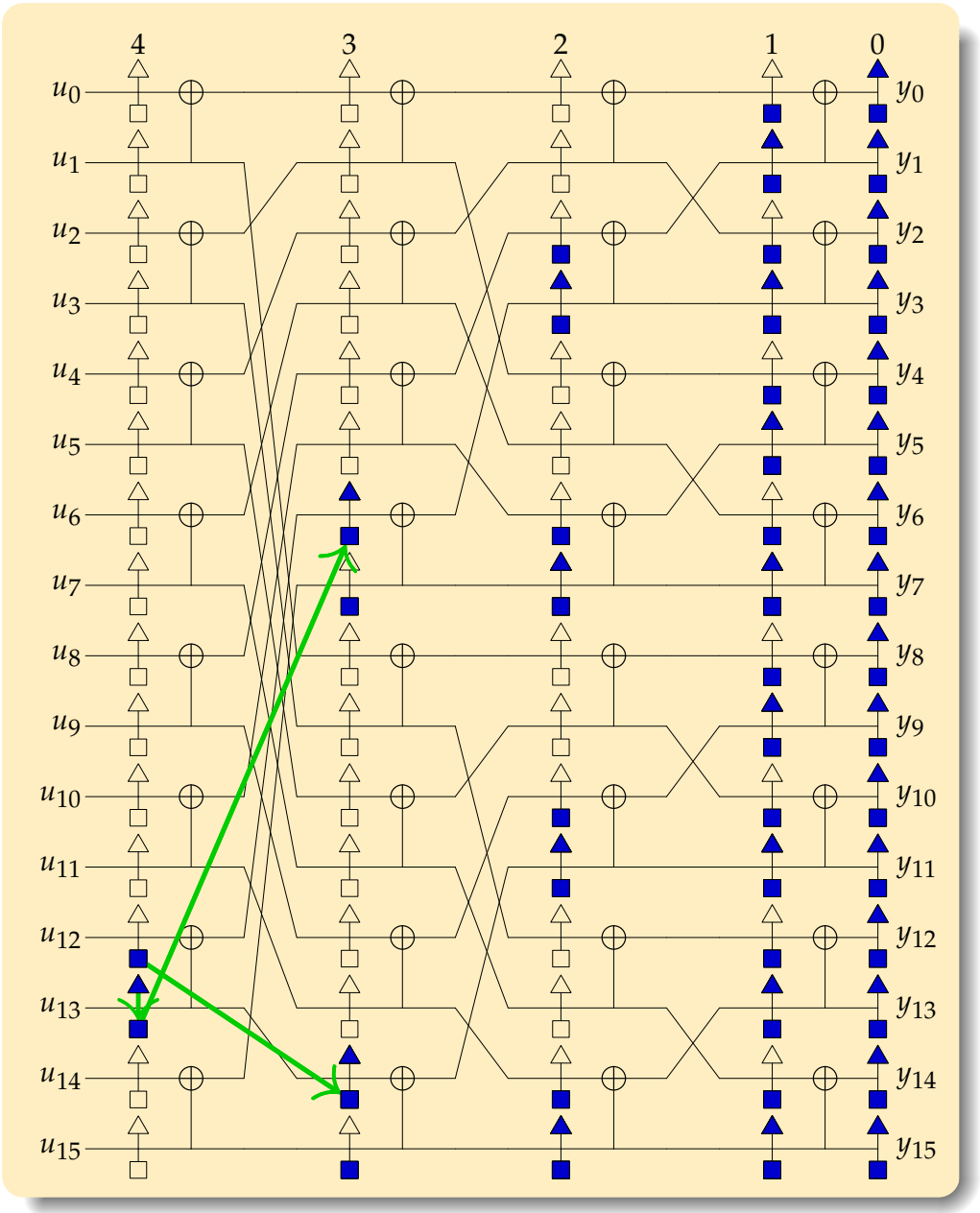
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

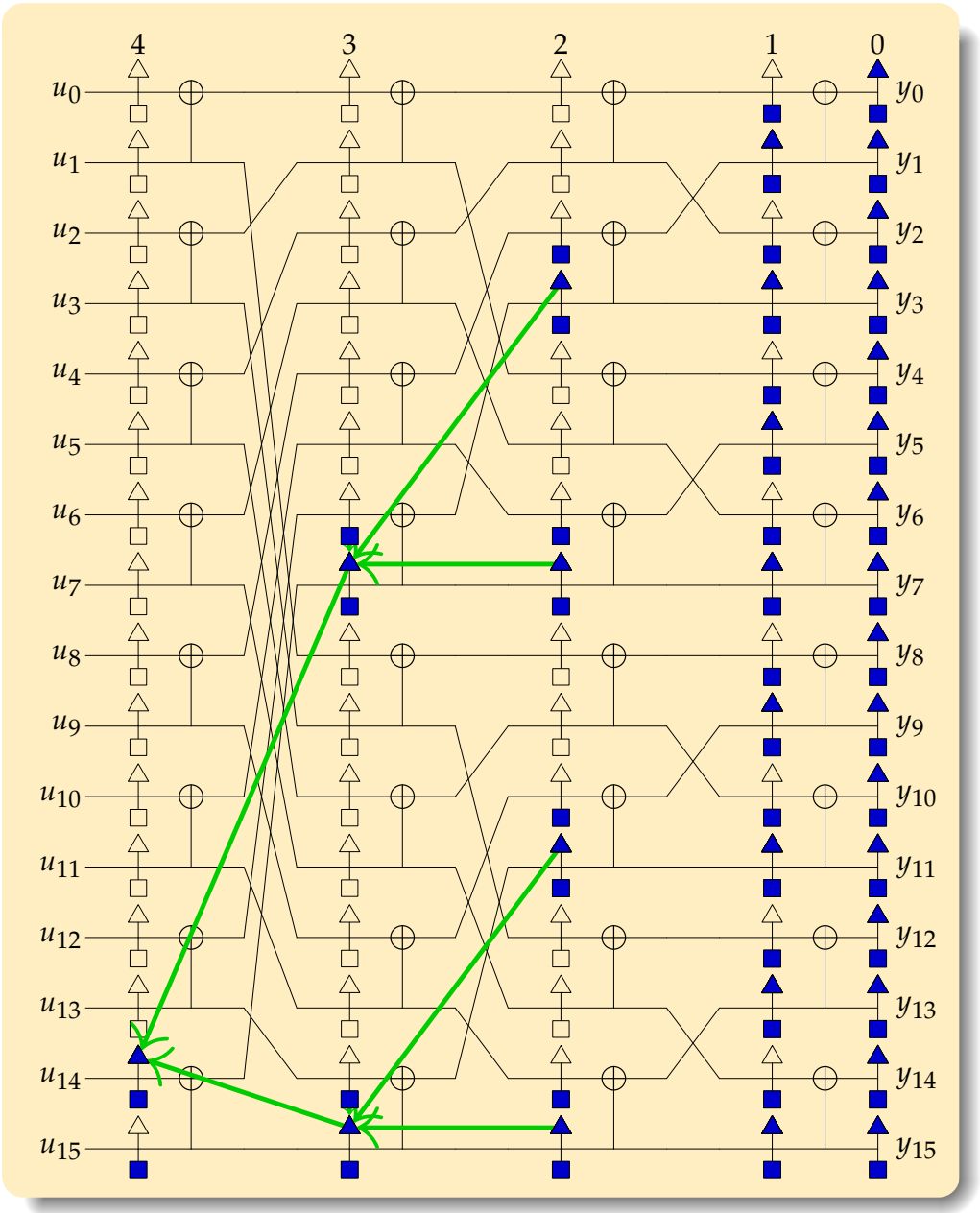
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

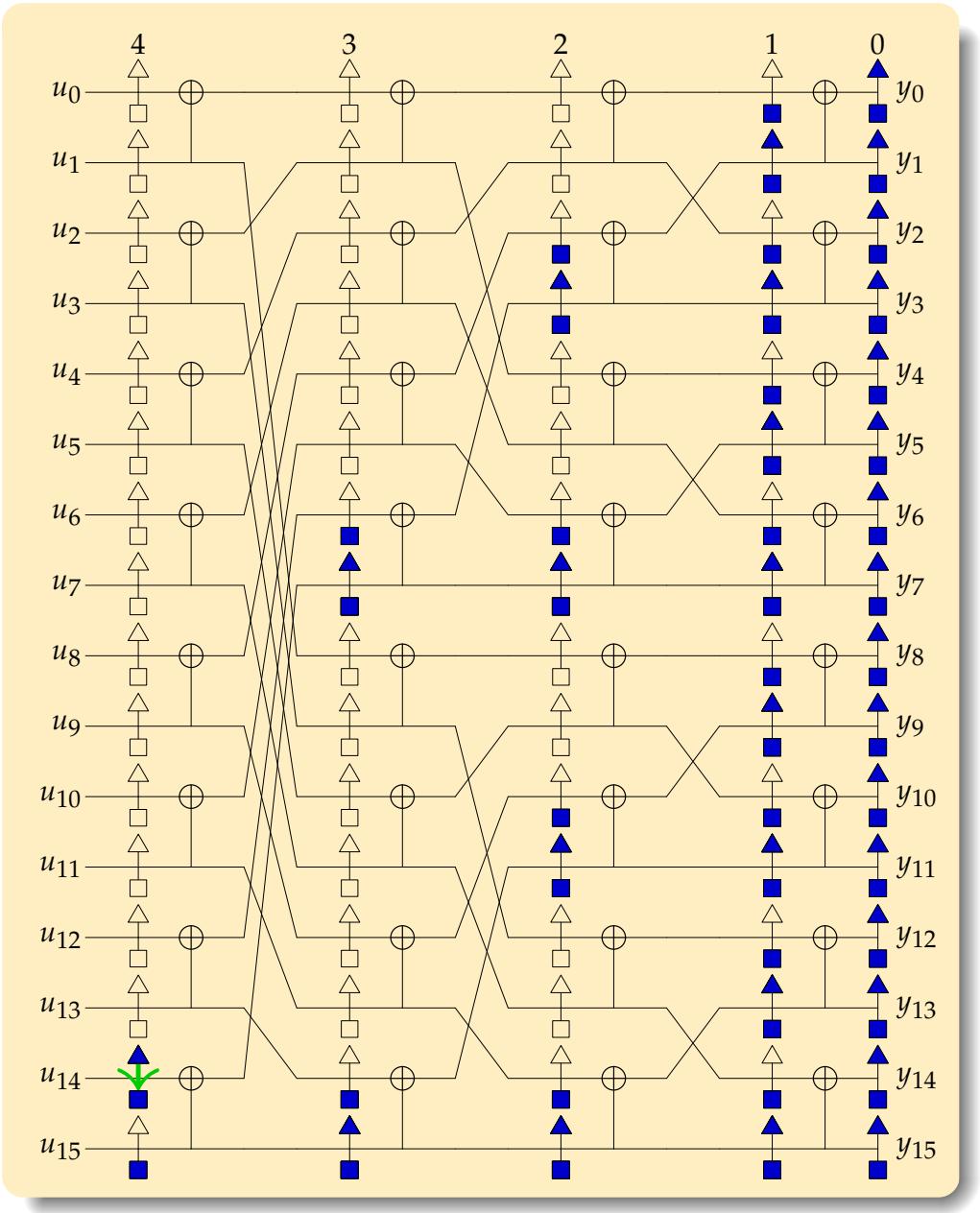
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

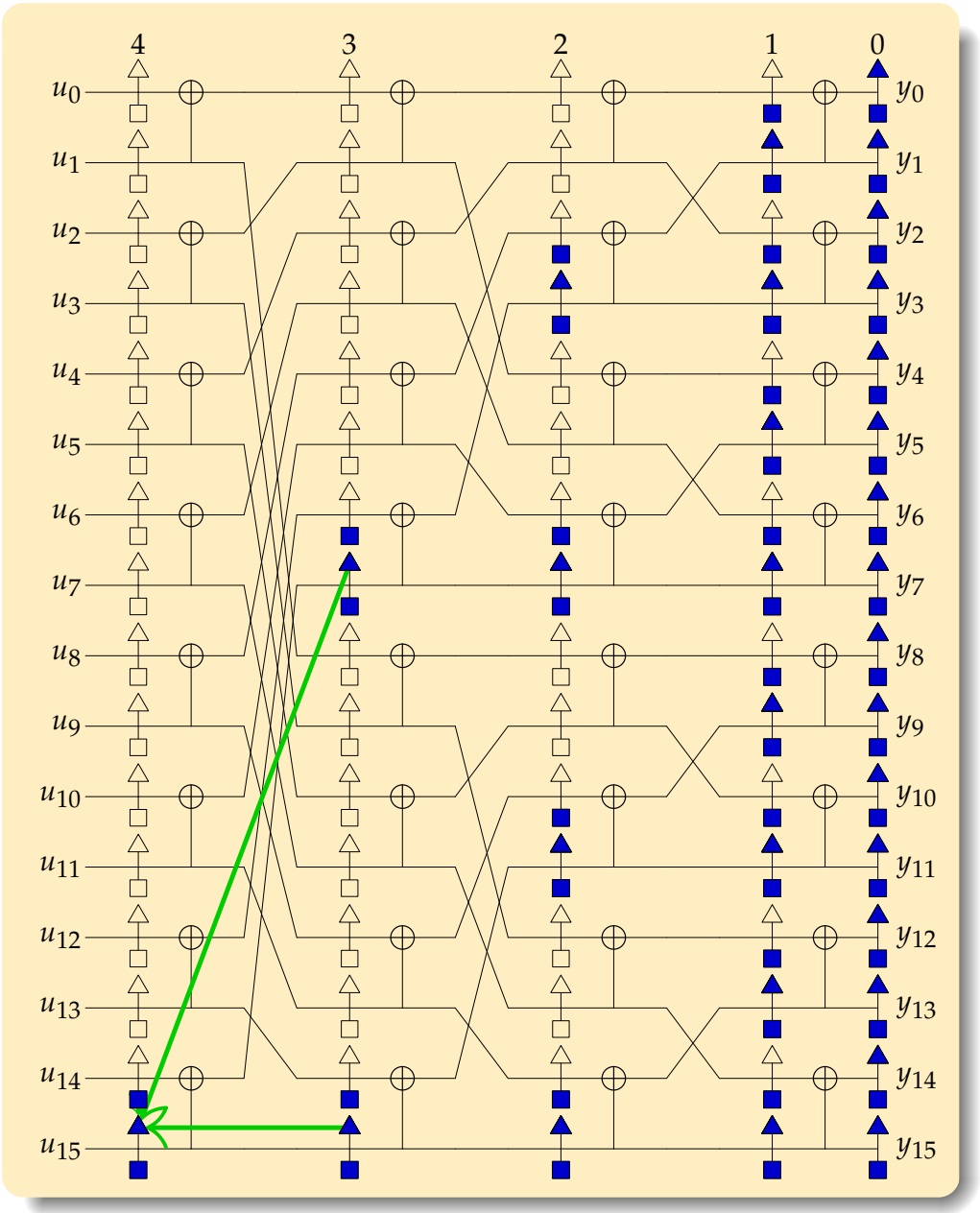
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

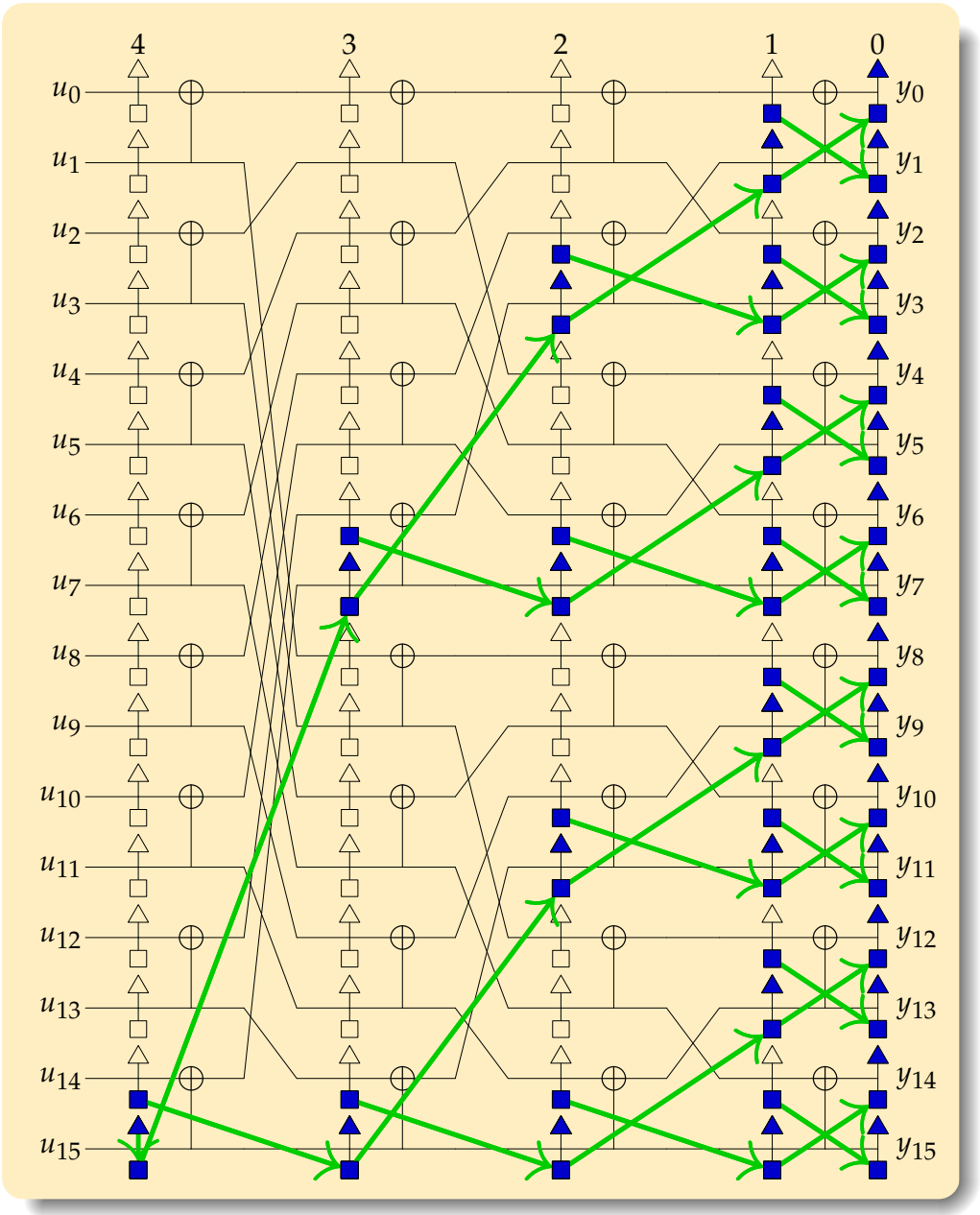
Level t is written to once every $O(2^{m-t})$ stages.



A larger example

Key point

Level t is written to once every $O(2^{m-t})$ stages.



Application to list decoding

- In a naive implementation, at each split we make a copy of the variables.
- We can do better:
  - At each split, flag the corresponding variables as belonging to both paths.
  - Give each path a unique variable (make a copy) only before that variable will be written to.
  - If a path is killed, deflag its corresponding variables.
- Thus, instead of wasting a lot of time on copy operations at each stage, we typically perform only a small number of copy operations.

This was a mile high view, there are many details to be filled (book-keeping, data structures), but the end result is a running time of $O(L \cdot n \log n)$ with $O(L \cdot n)$ memory requirements.