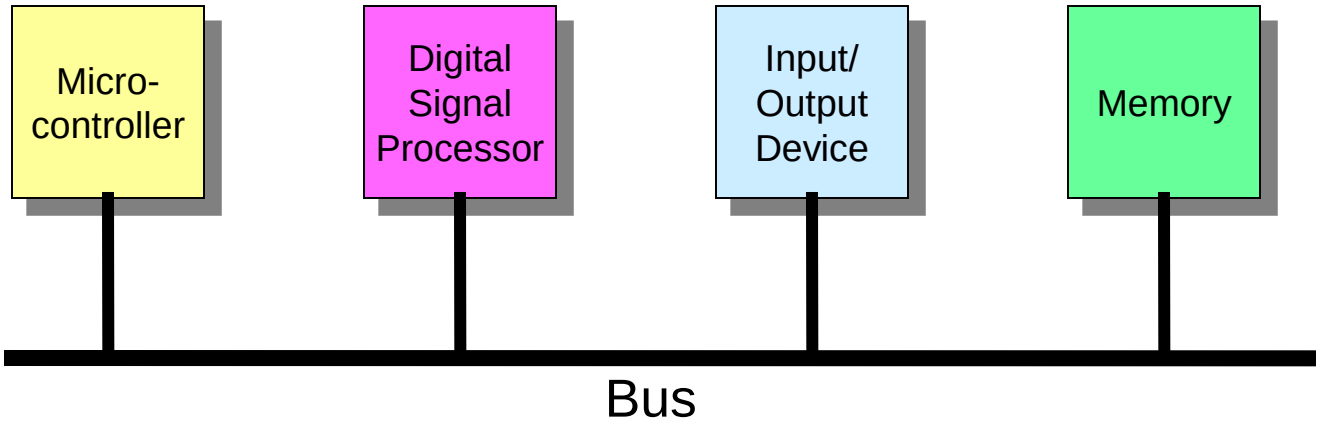
Buses

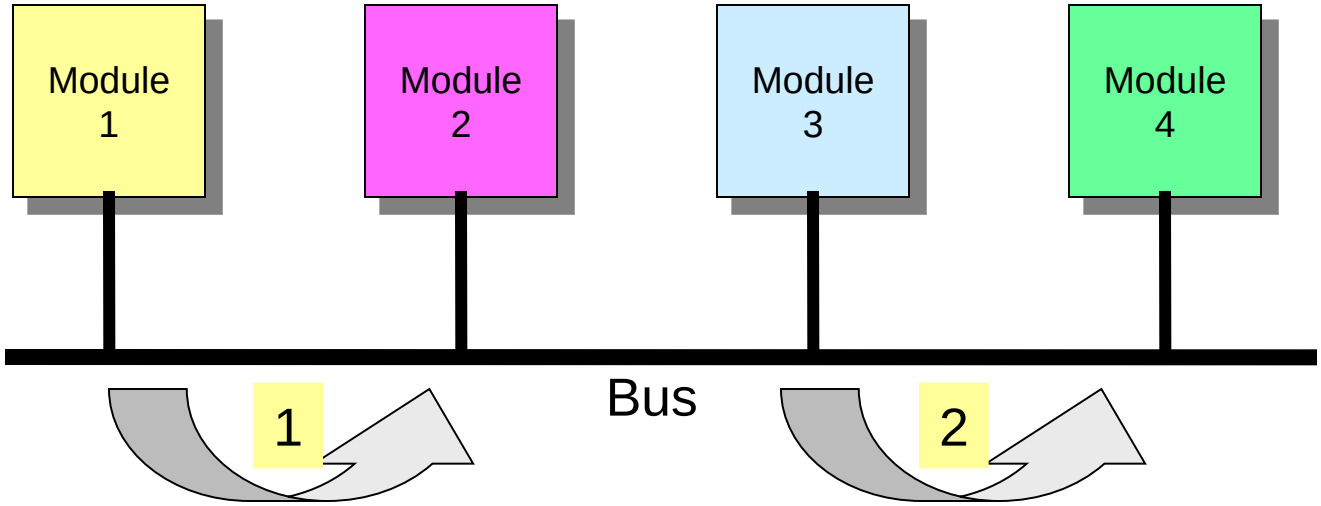
Introduction

- Buses are the simplest and most widely used interconnection networks
- A number of modules is connected via a single shared channel



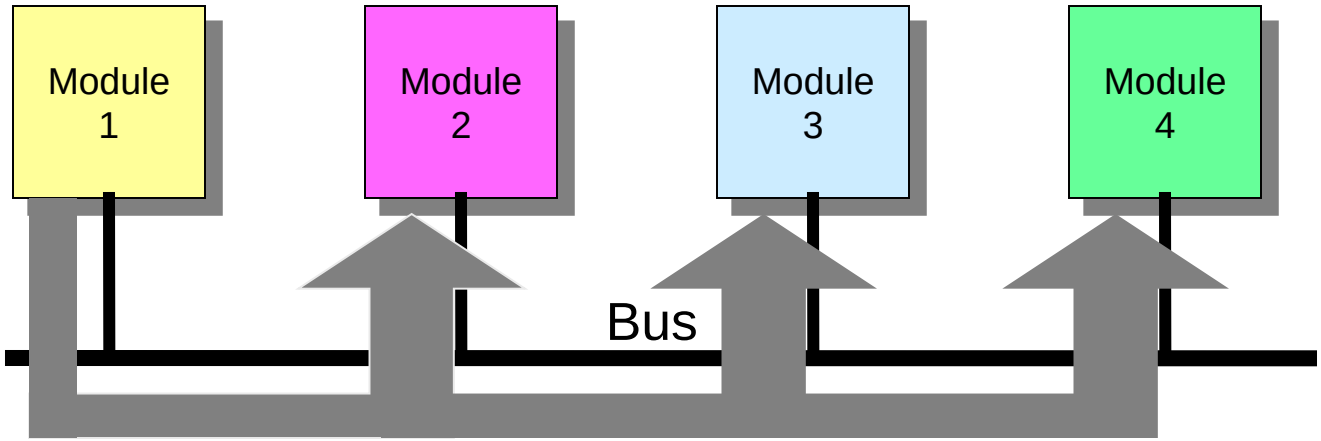
Bus Properties

- Serialization
  - ➔Only one component can send a message at any given time
  - ➔There is a total order of messages



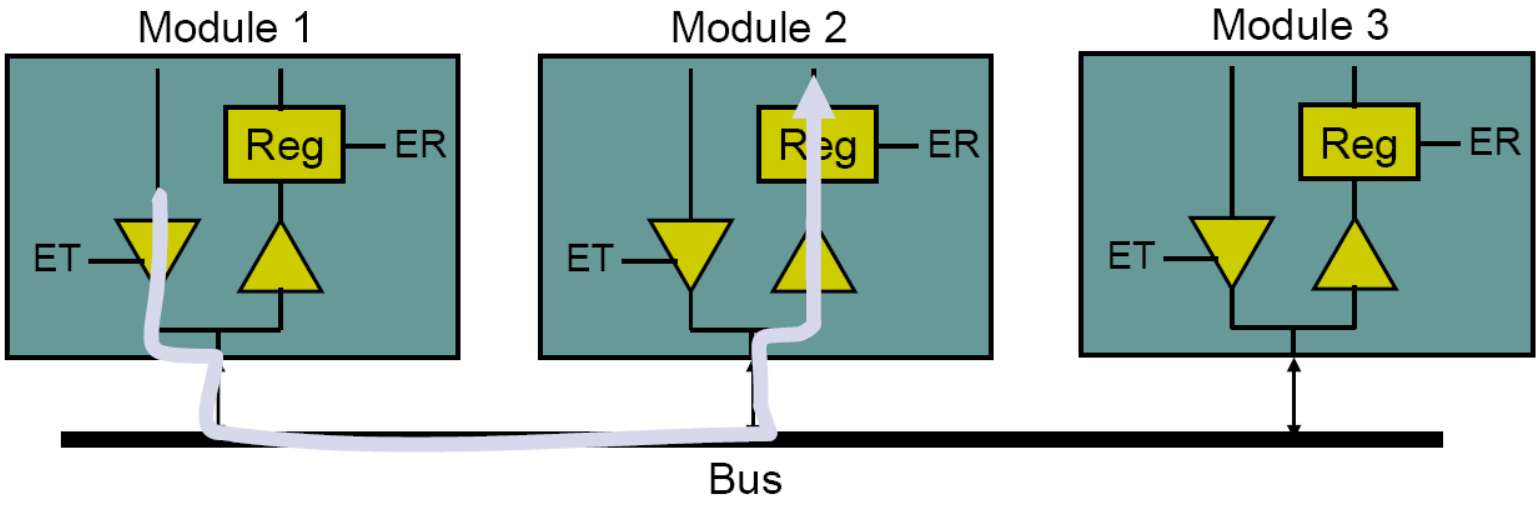
Bus Properties

- Broadcast
  - ➔ A module can send a message to several other components without an extra cost



Bus Hardware

- Principle for hardware to access the bus
  - Bus Transmit: ET active
  - Bus Receive: ER active

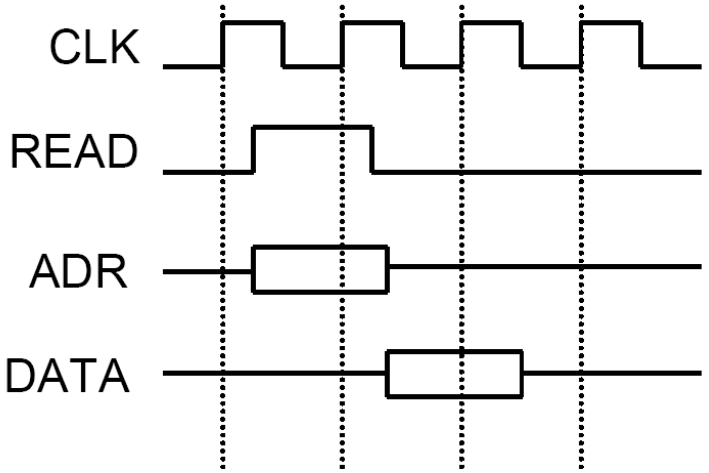


Cycles, Messages and Transactions

- Buses operate in units of cycles, messages and transactions
  - ➔ *Cycles*: A message requires a number of cycles to be sent from sender to receiver over the bus
  - ➔ *Message*: Logical unit of information (a read message contains an address and control signals for read)
  - ➔ *Transaction*: A transaction consists of a sequence of messages which together form a transaction (a memory read requires a memory read message and a reply with the requested data)

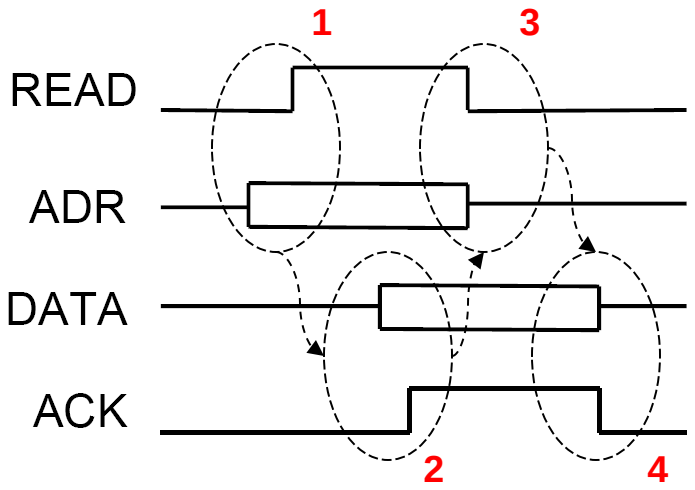
Synchronous Bus

- Includes a clock in the control lines
- A fixed protocol for communication that is relative to the clock
- Advantage: involves very little logic and can run very fast
- Disadvantages:
  - ➔ Every device on the bus must run at the same clock rate
  - ➔ To avoid clock skew, they cannot be long if they are fast



Asinchronous Bus

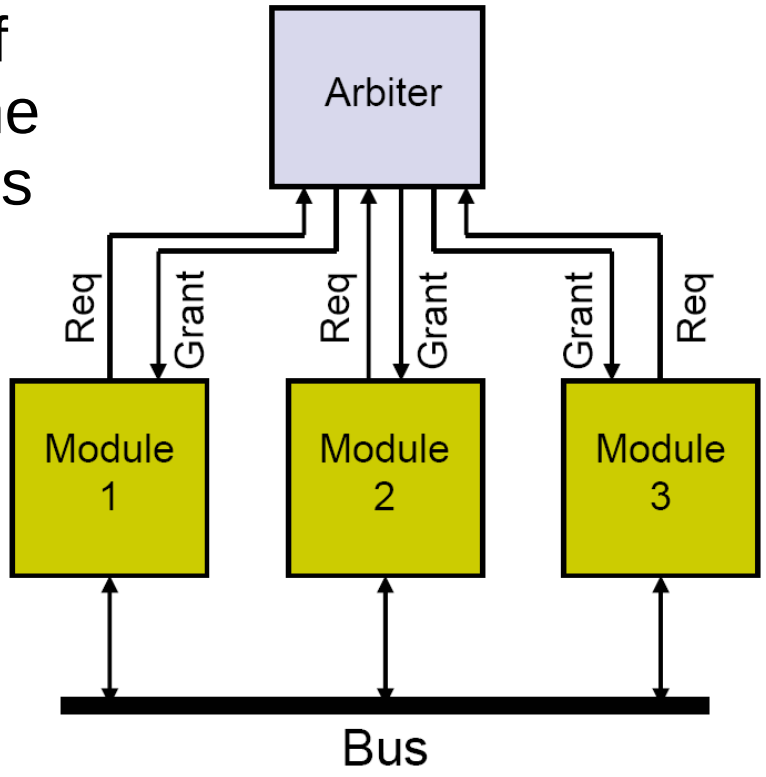
- It is not clocked
- It can accommodate a wide range of devices
- It can be lengthened without worrying about clock skew
- It requires a handshaking protocol



1. Master puts address on bus and asserts READ when address is stable
2. Memory puts data on bus and asserts ACK when data is stable
3. Master deasserts READ when data is read
4. Memory deasserts ACK

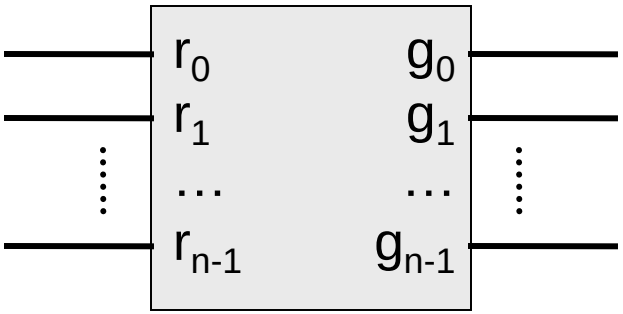
Bus Arbitration

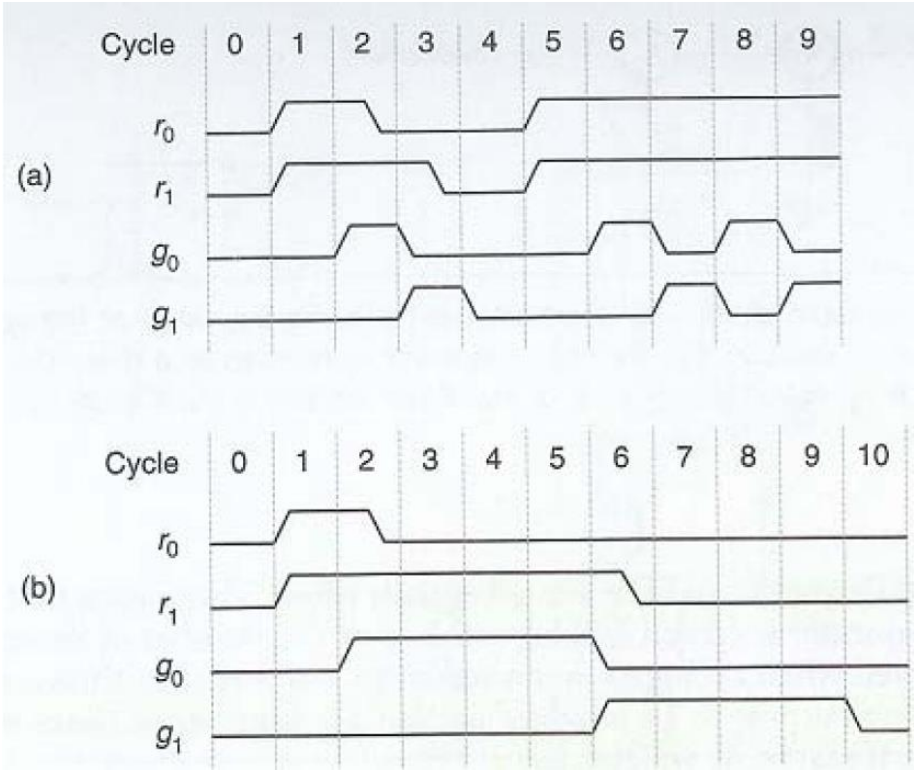
- Since only one bus master can use the bus at a given time bus arbitration is used
- An arbiter collects the requests of all bus masters and gives only one module the right to access the bus (bus grant)
- Arbiters are not only used in bus-system, but everywhere where several devices request shared resources



Arbiter Interfaces

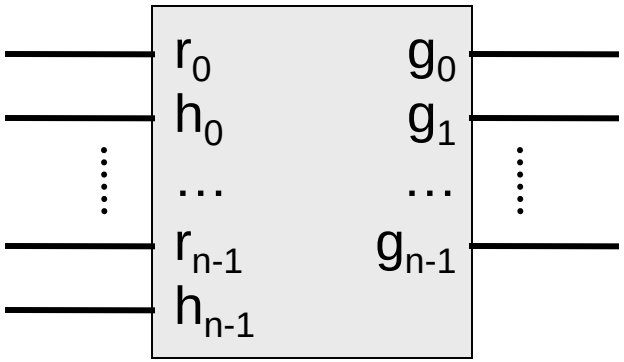
- This arbiter interface can be used to give a bus grant for a fixed number of cycles
  - (a): 1 cycle
  - (b): 4 cycles

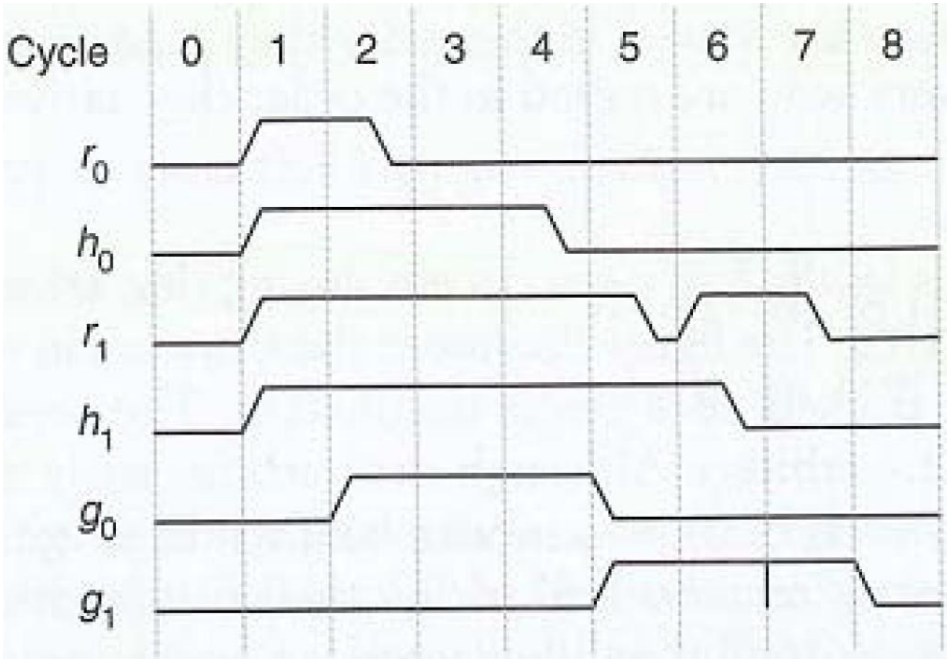




Arbiter Interfaces

- This arbiter allows for variable length grants
- The grant is hold as long as the “hold”-line is asserted
- In cycle 2 requester 0 gets the bus for 3 cycles
- In cycle 5 requester 1 gets the bus for 2 cycles
- In cycle 7 requester 1 gets the bus for one cycle



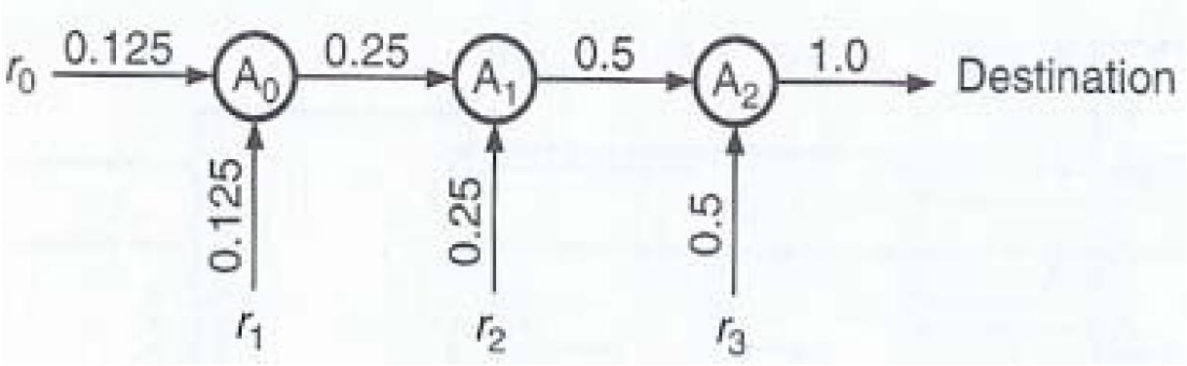


Fairness

- Fairness is a key property of an arbiter
- Some definitions
  - ➔ *Weak fairness*: Every request is eventually served
  - ➔ *Strong fairness*: Requests will be served equally often
  - ➔ *Weighted "strong" fairness*: The number of times requester *i* is served is equal to its weight w_i
  - ➔ *FIFO fairness*: Requests are served in the order the requests have been made

Local Fairness vs. Global Fairness

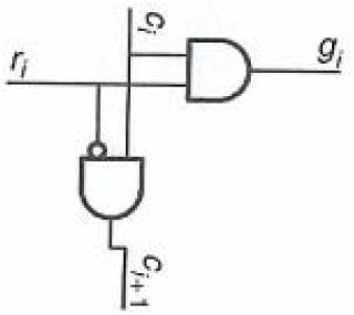
- Even if an arbiter is locally fair, a system with several arbiters employing that arbiter may not be fair

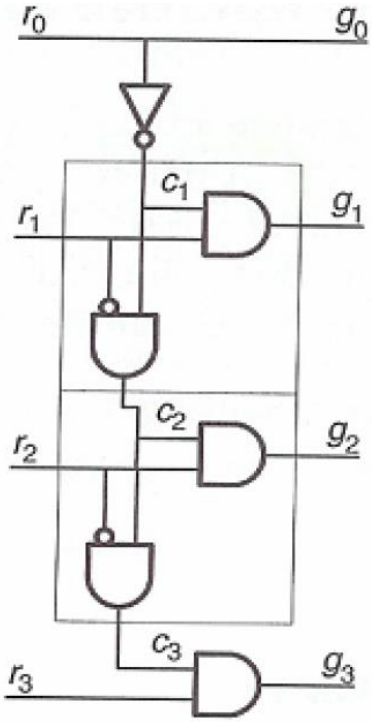


- Though each arbiter A_i allocate 50% of their bandwidth to its two inputs, r_0 only gets 12.5% of the total bandwidth, while r_3 gets 50%

Fixed-Priority Arbiter

- A fixed-priority arbiter can be constructed as an iterative circuit
- Each cell receives a request input r_i and a carry input c_i and generates a grant output g_i and a carry output c_{i+1}
- The resulting arbiter is not fair, since a continuously asserted request r_0 means that none of the other requests will ever be served!



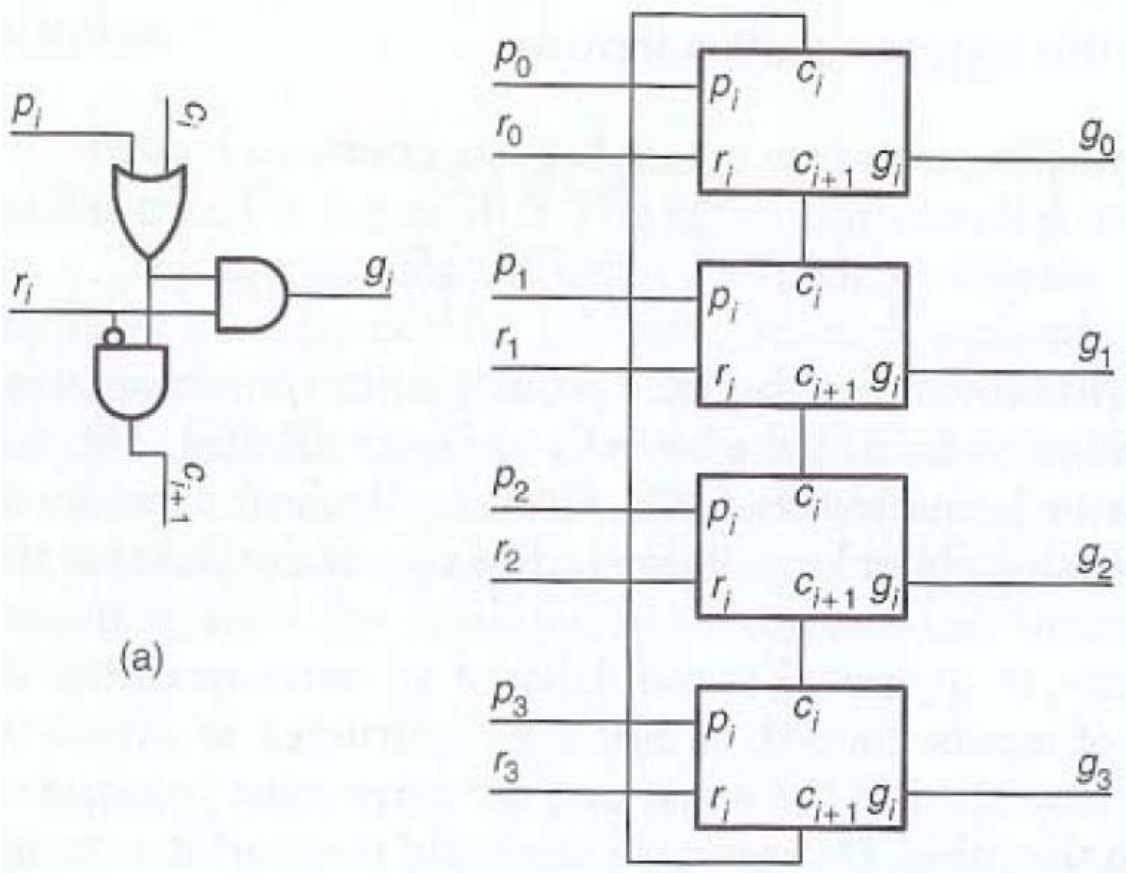


Variable-Priority Arbiters

- Oblivious Arbiter
- Round-Robin Arbiter
- Grant-Hold Circuit
- Weighted Round-Robin Arbiter

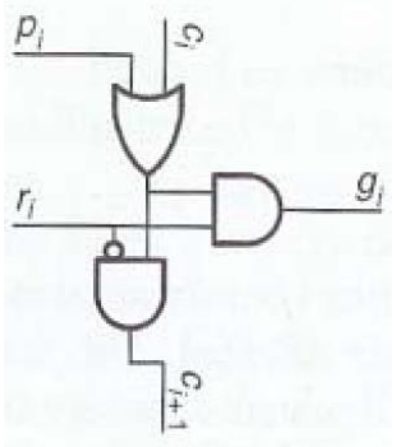
Fair Arbiters

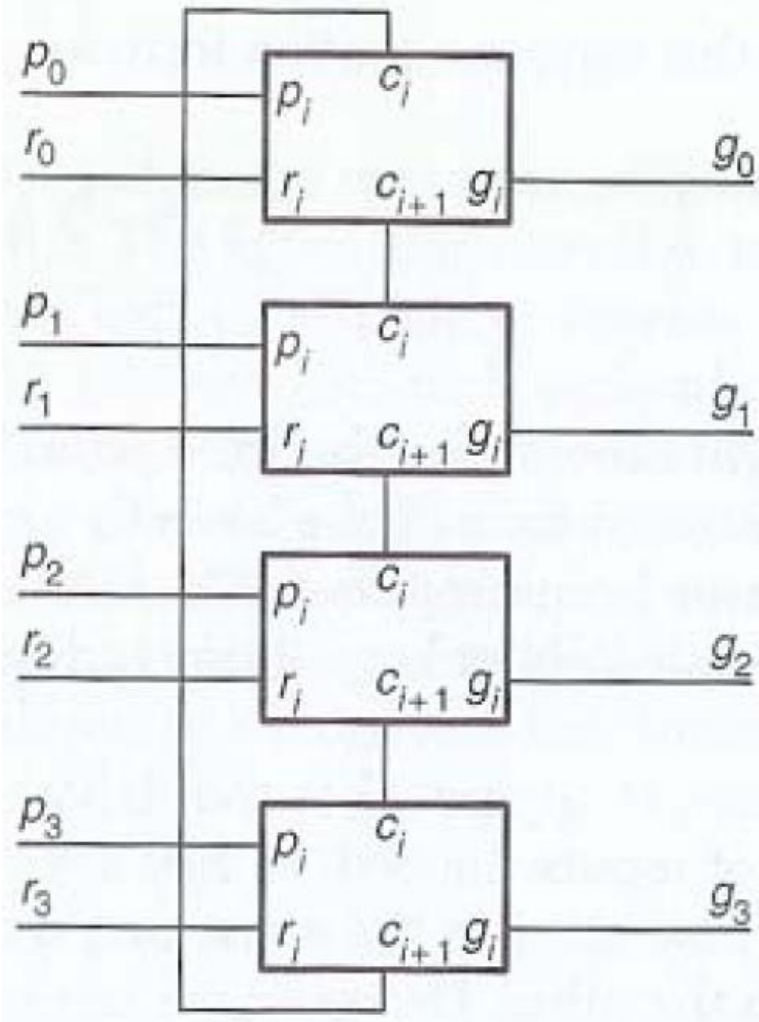
- A fair arbiter can be generated by changing the priority from cycle to cycle
- Depending on the priority generation, different arbitration schemes and degrees of fairness can be achieved



Fair Arbiters

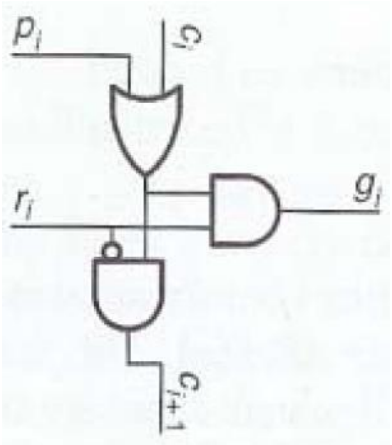
- Oblivious Arbiters
  - ➔ If p_i is generated without knowledge of r_i and g_i, the result is an oblivious (unconscious) arbiter
  - ➔ Examples are
    - ✓ Randomly generated p_i
    - ✓ Rotating priorities (by shiftregister)

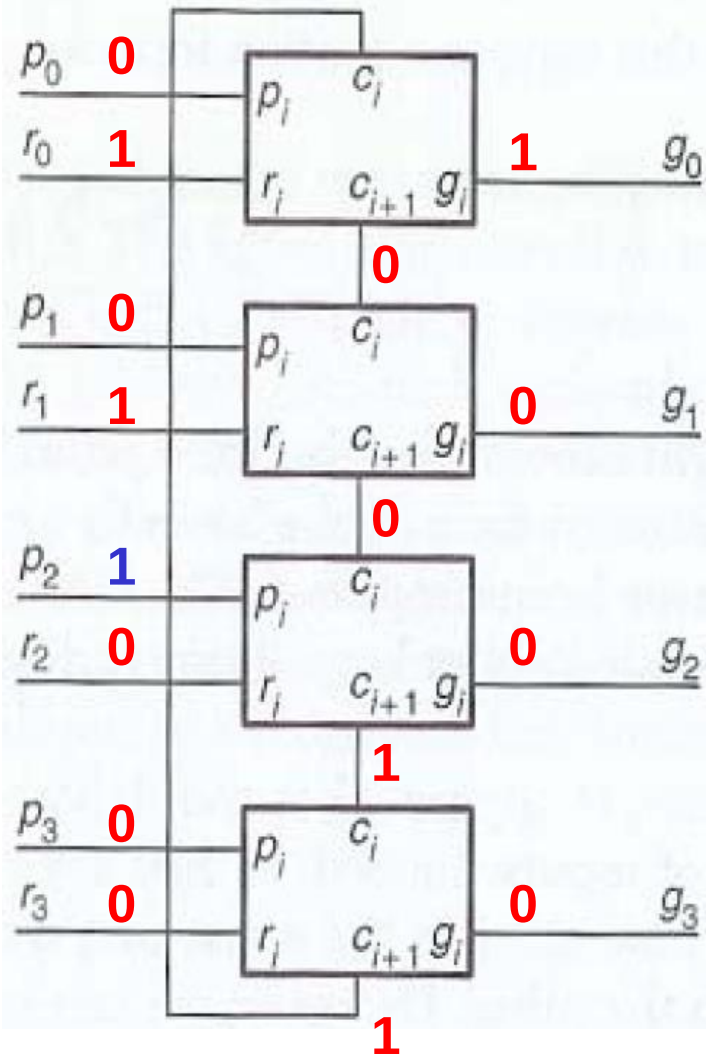




Oblivious Arbiters

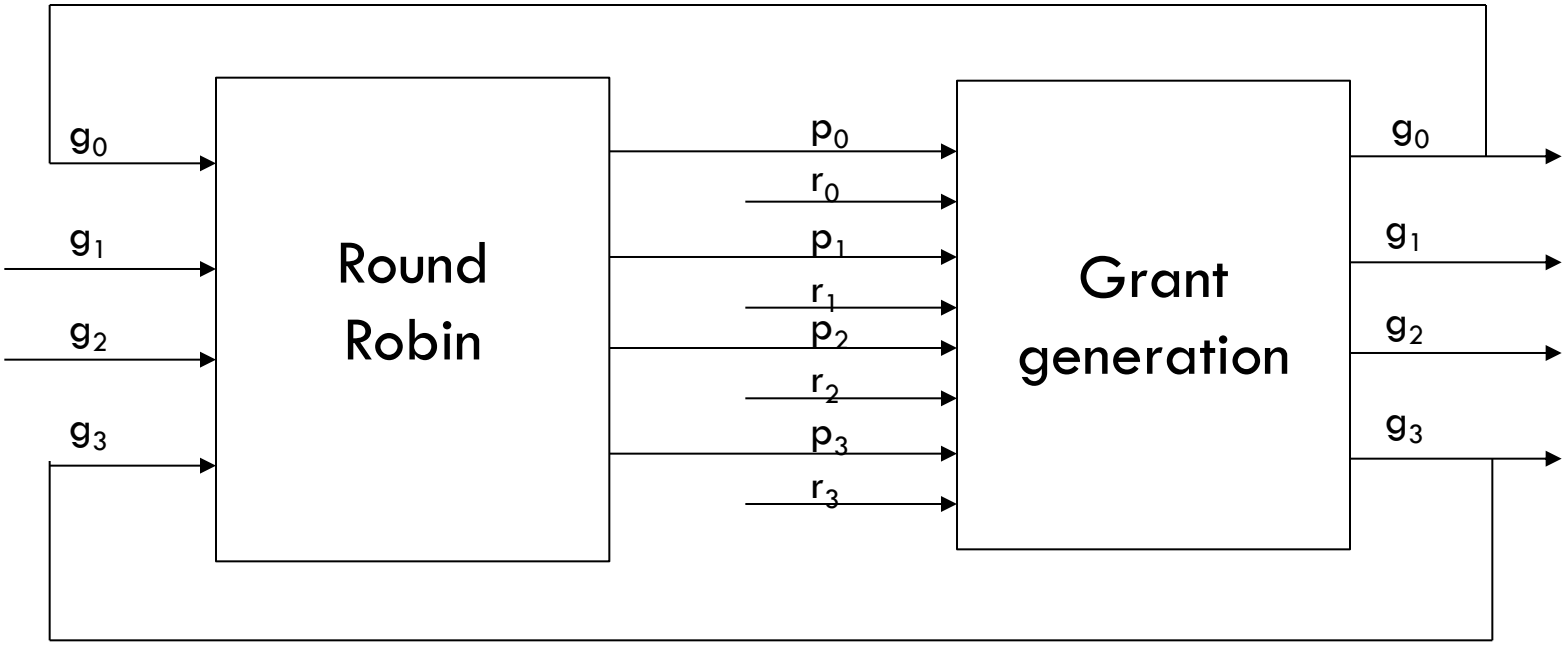
- Oblivious arbiters provide
  - ➔ weak fairness
  - ➔ but not strong fairness (i.e. if r_0 and r_1 are constantly asserted, r0 will receive the grant 3 times and r1 only 1 time)





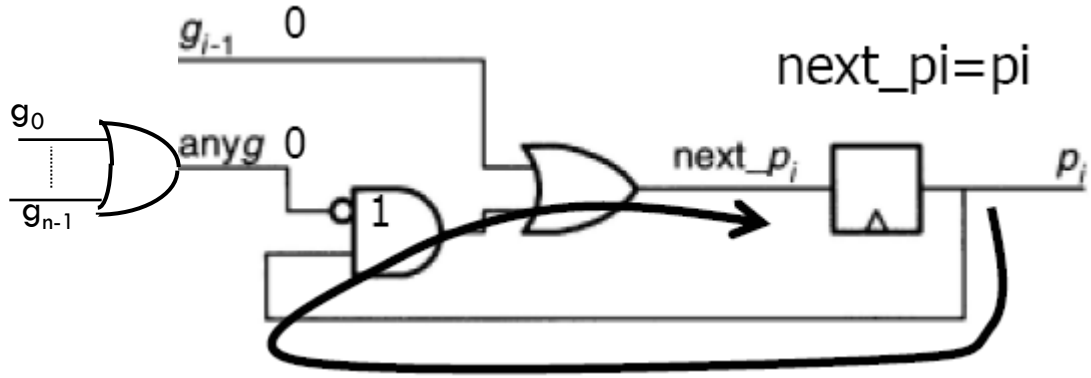
Round-Robin Arbiter

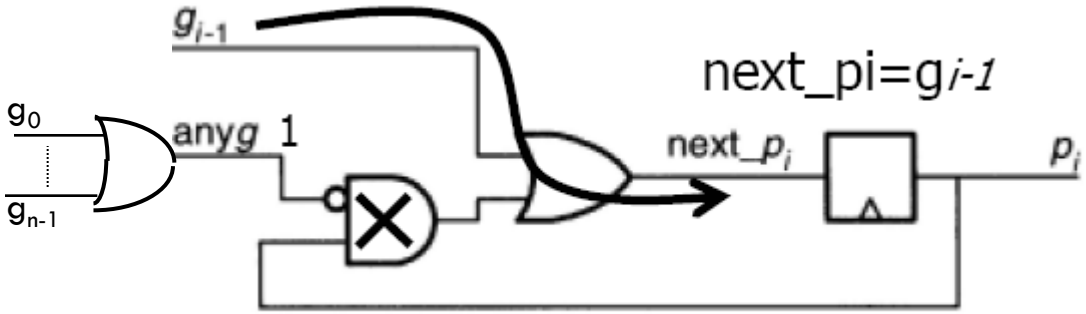
- A round-robin arbiter achieves strong fairness
  - ➔ A request that was just served gets the lowest priority
- One-bit round-robin arbiter



Round-Robin Arbiter

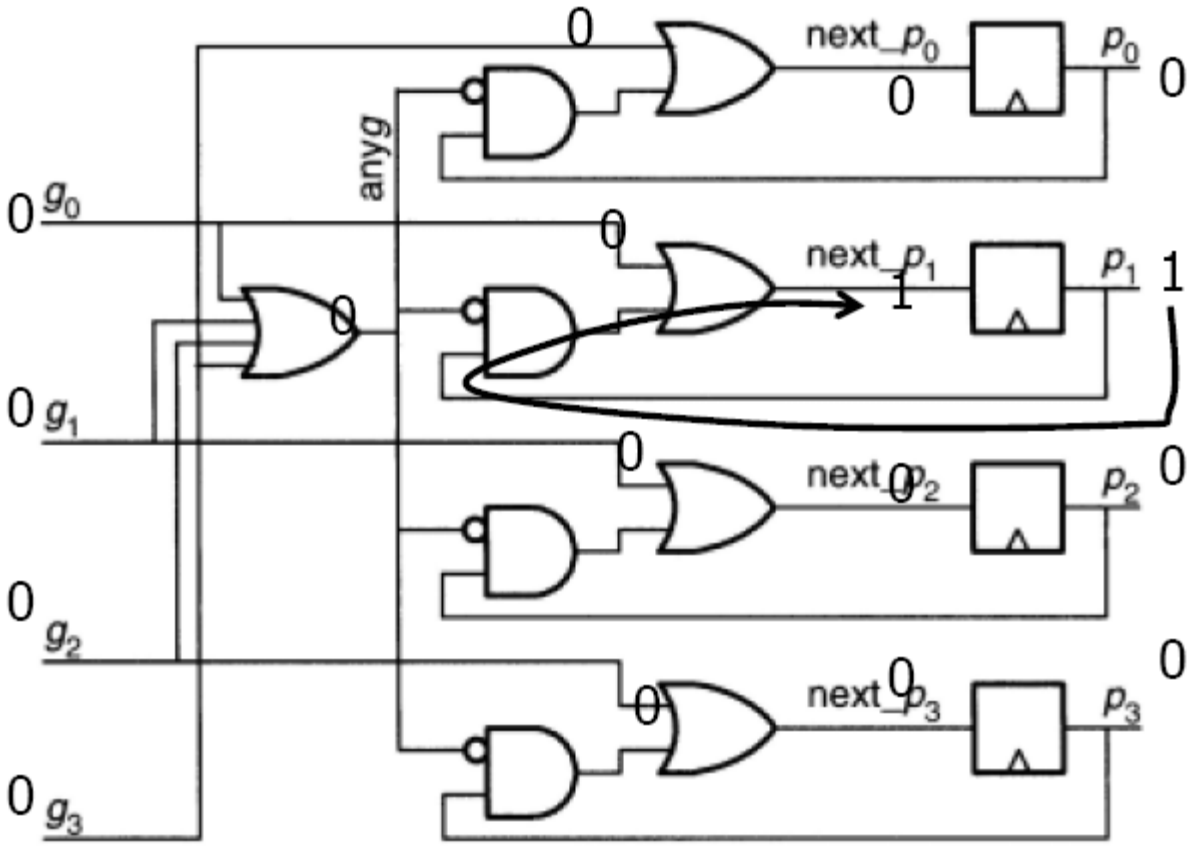
next_pi <= g_{i-1} OR (p_i AND NOT anyg)





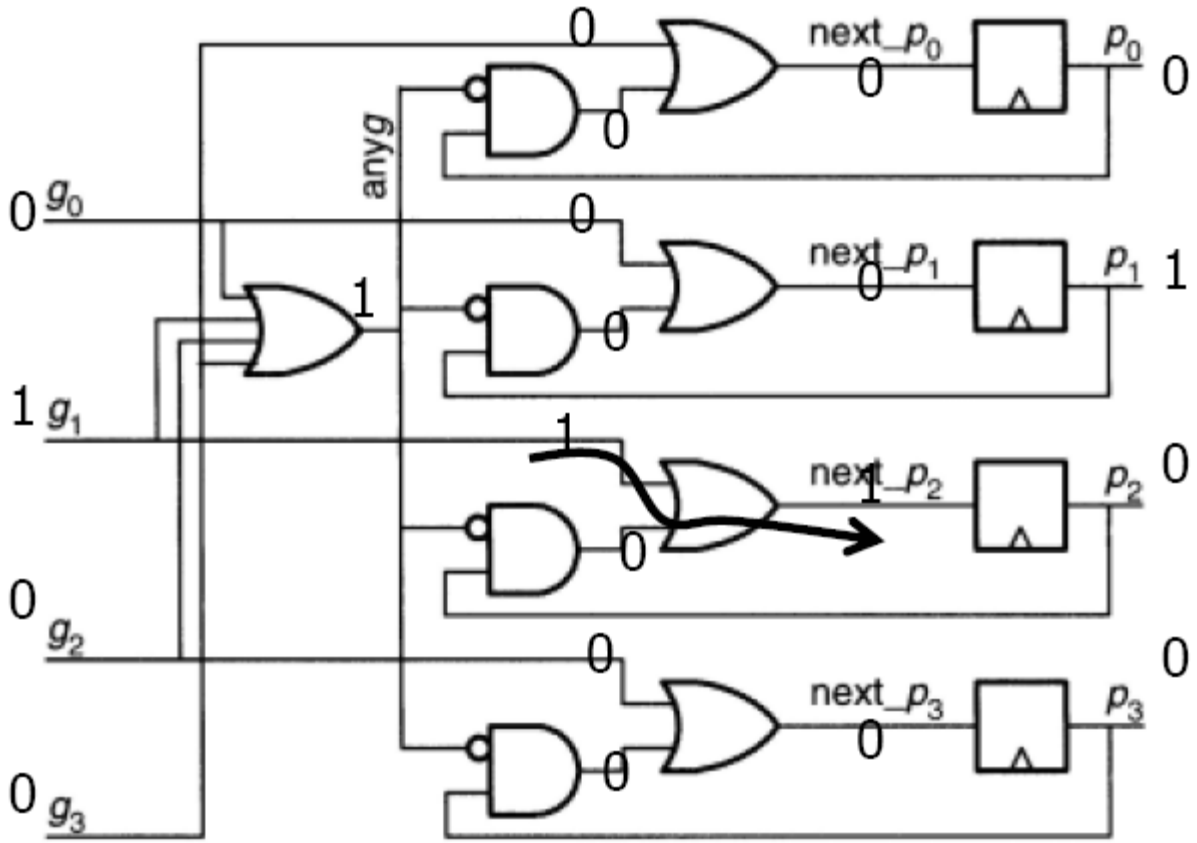
Round-Robin Arbiter

- Any *g* is low



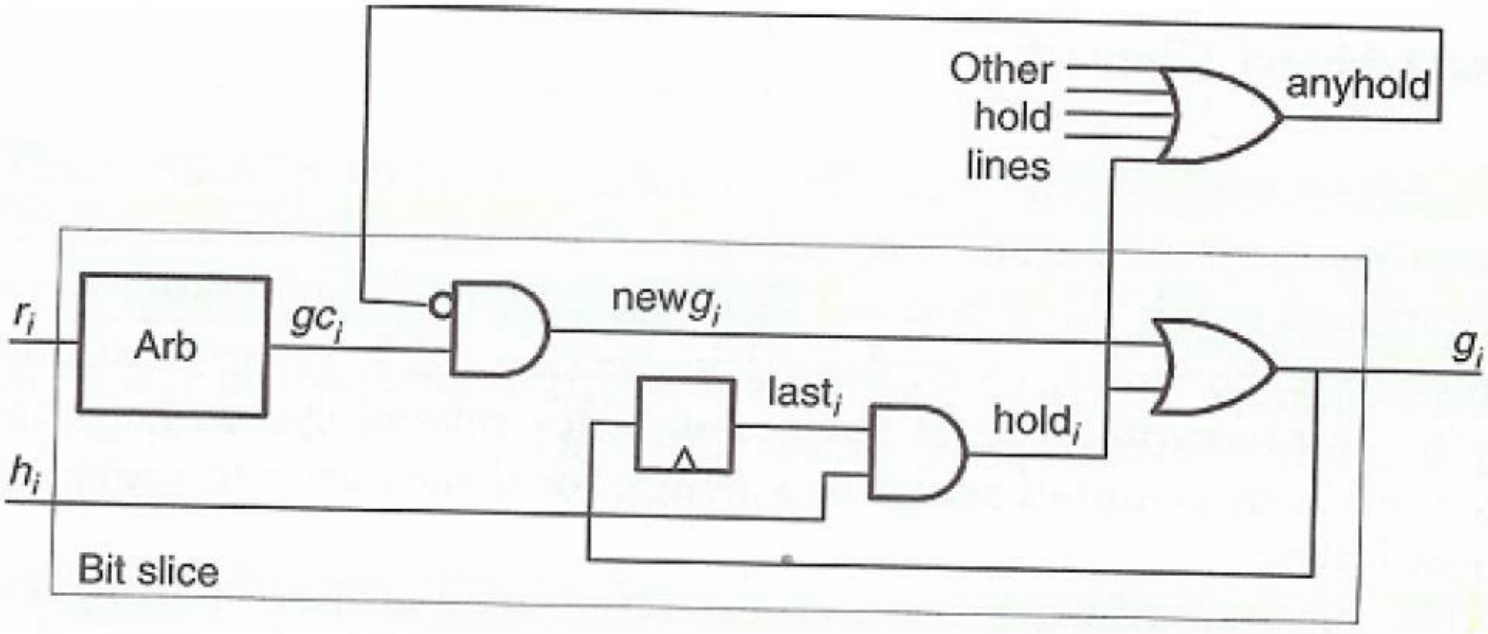
Round-Robin Arbiter

- One *g* is high



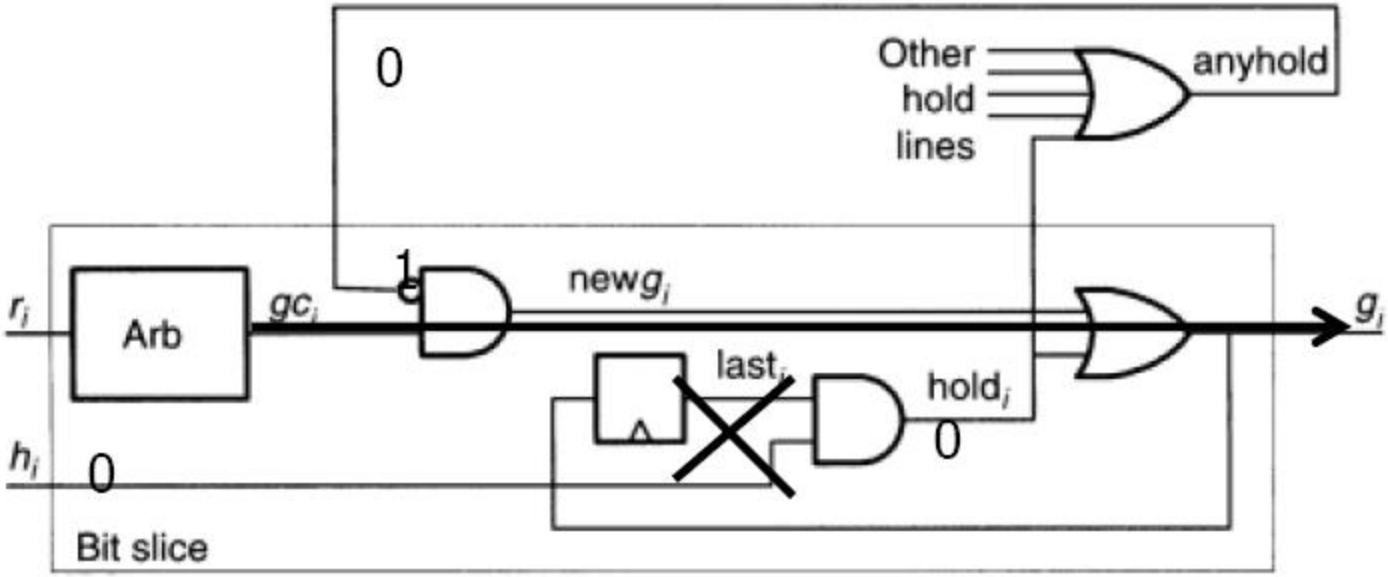
Grant-Hold Circuit

- Extends the duration of a grant
  - ➔ As long as hold is asserted further arbitration is disabled



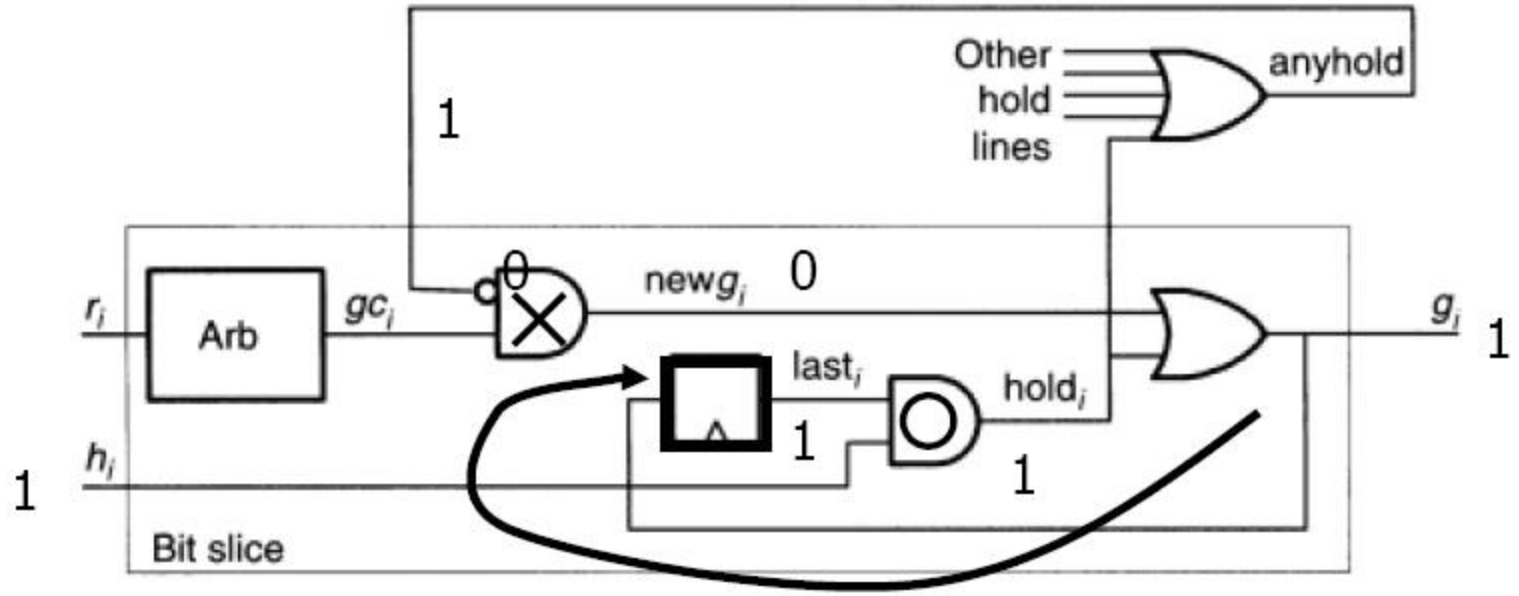
Grant-Hold Circuit

- An example for all h_i=0



Grant-Hold Circuit

- An example for holding a grant

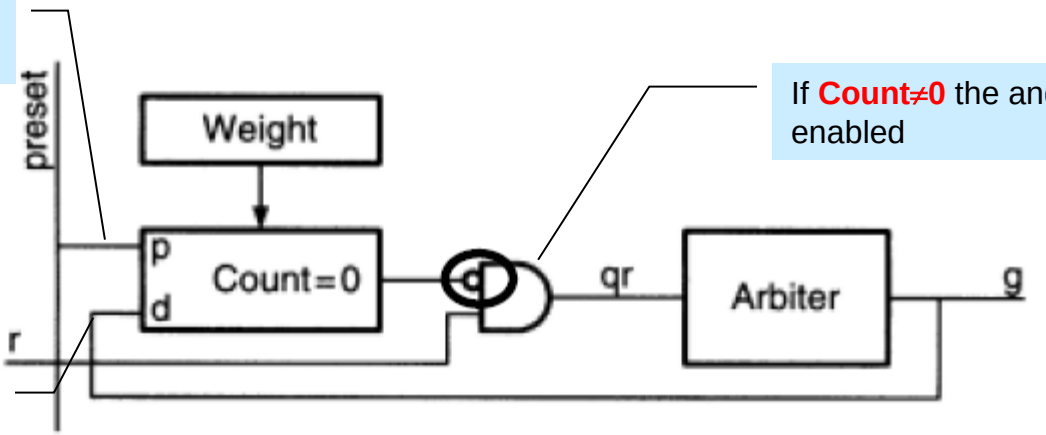


Weighted Round-Robin Arbiter

- A weighted round-robin arbiter allows to give requesters a larger number of grants than other requesters in a controlled fashion
- If three devices have the weight 1,2,3 they get 1/6, 1/3 and 1/2 of the grants
- The *preset* line is activated periodically after *N* (here 6 cycles) to load the counter with its weight
- If some arbiters do not issue any requests during that interval, the shared resource will remain idle until the next preset cycle

Initialized periodically every *W* cycles

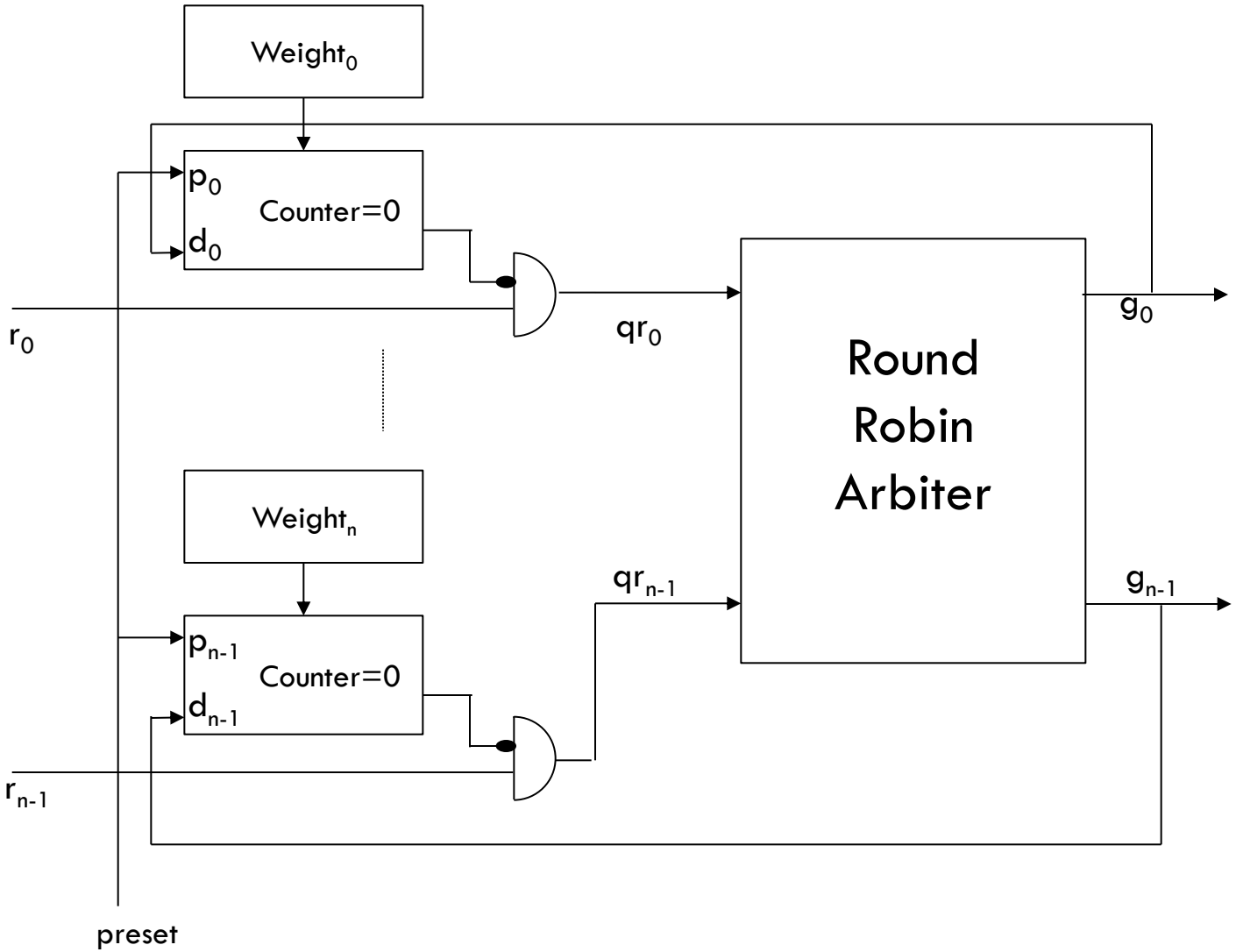
$W=\Sigma w_i$



If **Count≠0** the and gate is enabled

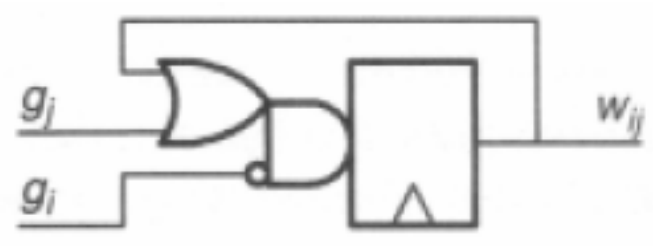
Decremented each time a grant is received

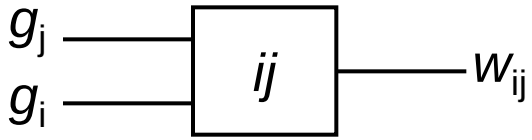
Weighted Round-Robin Arbiter



Matrix Arbiter

- Matrix $W=[w_{ij}]$ of weights
- $w_{ij}=1$ if request i takes priority over j





$$W_{ij}^{*} = ! g_i \cdot (g_i + W_{ij})$$

$$W_{ij} = ! W_{ji}$$

Matrix Arbiter

$$g_0 = r_0 \cdot !(r_1 \cdot W_{01} + r_2 \cdot W_{02})$$
$$W^*_{01} = !g_1 \cdot (g_0 + W_{01})$$
$$W^*_{02} = !g_2 \cdot (g_0 + W_{02})$$

$$g_1 = r_1 \cdot !(r_0 \cdot W_{10} + r_2 \cdot W_{12})$$
$$W^*_{10} = !g_0 \cdot (g_1 + W_{10})$$
$$W^*_{12} = !g_2 \cdot (g_1 + W_{12})$$

$$g_2 = r_2 \cdot !(r_0 \cdot W_{20} + r_1 \cdot W_{21})$$
$$W^*_{20} = !g_0 \cdot (g_2 + W_{20})$$
$$W^*_{21} = !g_1 \cdot (g_2 + W_{21})$$

Matrix Arbiter

$W_{01}=0, W_{02}=0, W_{10}=1, W_{12}=0, W_{20}=1, W_{21}=1$
$r_0=1, r_1=1, r_2=1$

$g_0 = r_0 \cdot !(r_1 \cdot W_{01} + r_2 \cdot W_{02}) = 1$
$W^*_{01} = !g_1 \cdot (g_0 + W_{01}) = 1$
$W^*_{02} = !g_2 \cdot (g_0 + W_{02}) = 1$

$g_1 = r_1 \cdot !(r_0 \cdot W_{10} + r_2 \cdot W_{12}) = 0$
$W^*_{10} = !g_0 \cdot (g_1 + W_{10}) = 0$
$W^*_{12} = !g_2 \cdot (g_1 + W_{12}) = 0$

$g_2 = r_2 \cdot !(r_0 \cdot W_{20} + r_1 \cdot W_{21}) = 0$
$W^*_{20} = !g_0 \cdot (g_2 + W_{20}) = 0$
$W^*_{21} = !g_1 \cdot (g_2 + W_{21}) = 1$

Matrix Arbiter

$W_{01}=1, W_{02}=1, W_{10}=0, W_{12}=0, W_{20}=0, W_{21}=1$

$r_0=1, r_1=1, r_2=1$

$g_0 = r_0 \cdot !(r_1 \cdot W_{01} + r_2 \cdot W_{02}) = 0$
$W^*_{01} = !g_1 \cdot (g_0 + W_{01}) = 0$
$W^*_{02} = !g_2 \cdot (g_0 + W_{02}) = 1$

$g_1 = r_1 \cdot !(r_0 \cdot W_{10} + r_2 \cdot W_{12}) = 1$
$W^*_{10} = !g_0 \cdot (g_1 + W_{10}) = 1$
$W^*_{12} = !g_2 \cdot (g_1 + W_{12}) = 1$

$g_2 = r_2 \cdot !(r_0 \cdot W_{20} + r_1 \cdot W_{21}) = 0$
$W^*_{20} = !g_0 \cdot (g_2 + W_{20}) = 0$
$W^*_{21} = !g_1 \cdot (g_2 + W_{21}) = 0$

Matrix Arbiter

$W_{01}=0, W_{02}=1, W_{10}=1, W_{12}=1, W_{20}=0, W_{21}=0$

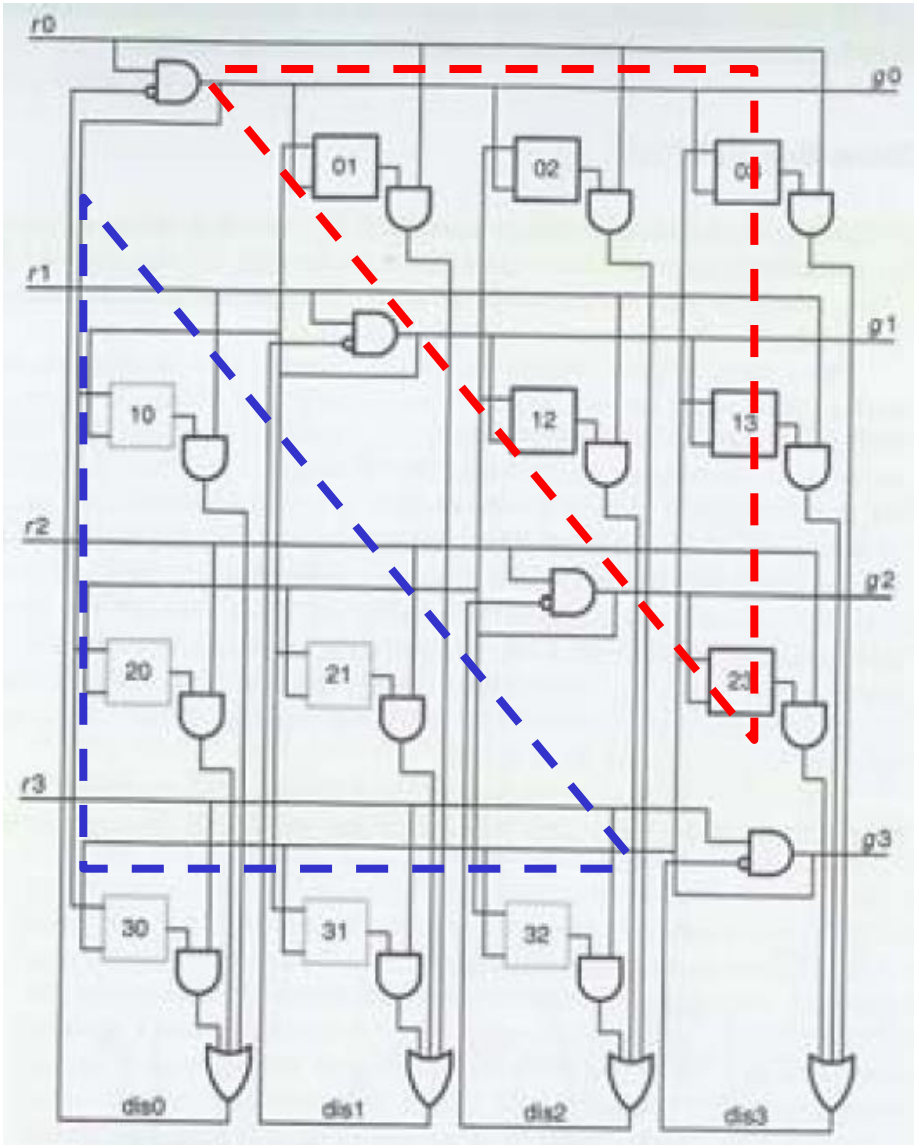
$r_0=1, r_1=1, r_2=1$

$g_0 = r_0 \cdot !(r_1 \cdot W_{01} + r_2 \cdot W_{02}) = 0$
$W^*_{01} = !g_1 \cdot (g_0 + W_{01}) = 0$
$W^*_{02} = !g_2 \cdot (g_0 + W_{02}) = 0$

$g_1 = r_1 \cdot !(r_0 \cdot W_{10} + r_2 \cdot W_{12}) = 0$
$W^*_{10} = !g_0 \cdot (g_1 + W_{10}) = 1$
$W^*_{12} = !g_2 \cdot (g_1 + W_{12}) = 0$

$g_2 = r_2 \cdot !(r_0 \cdot W_{20} + r_1 \cdot W_{21}) = 1$
$W^*_{20} = !g_0 \cdot (g_2 + W_{20}) = 1$
$W^*_{21} = !g_1 \cdot (g_2 + W_{21}) = 1$

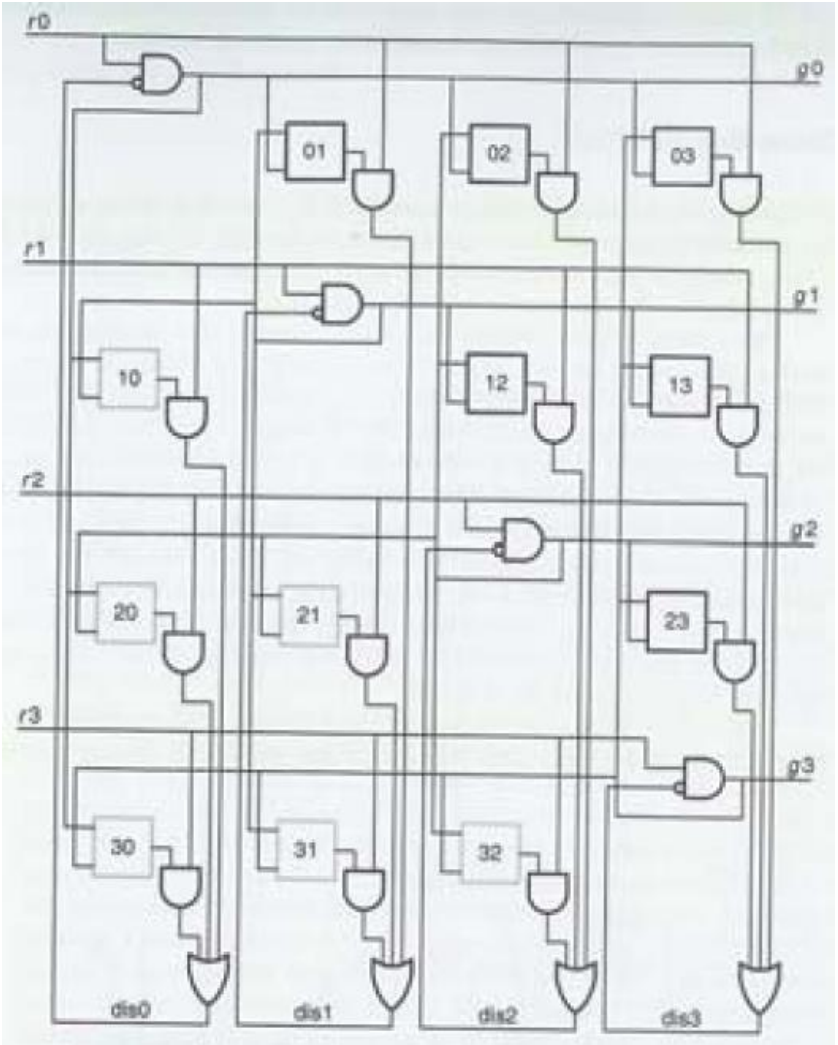
Matrix Arbiter



- $w_{ij} = !w_{ji}\ \forall i \neq j$
- A requester will be granted the resource if no other higher priority requester is bidding for the same resource
- Once a requester succeeds in being granted a resource, its priority is updated and set to be the lowest among all requesters
- Request *i* granted
  - ➔ [i,*] ← 0
  - ➔ [*,i] ← 1

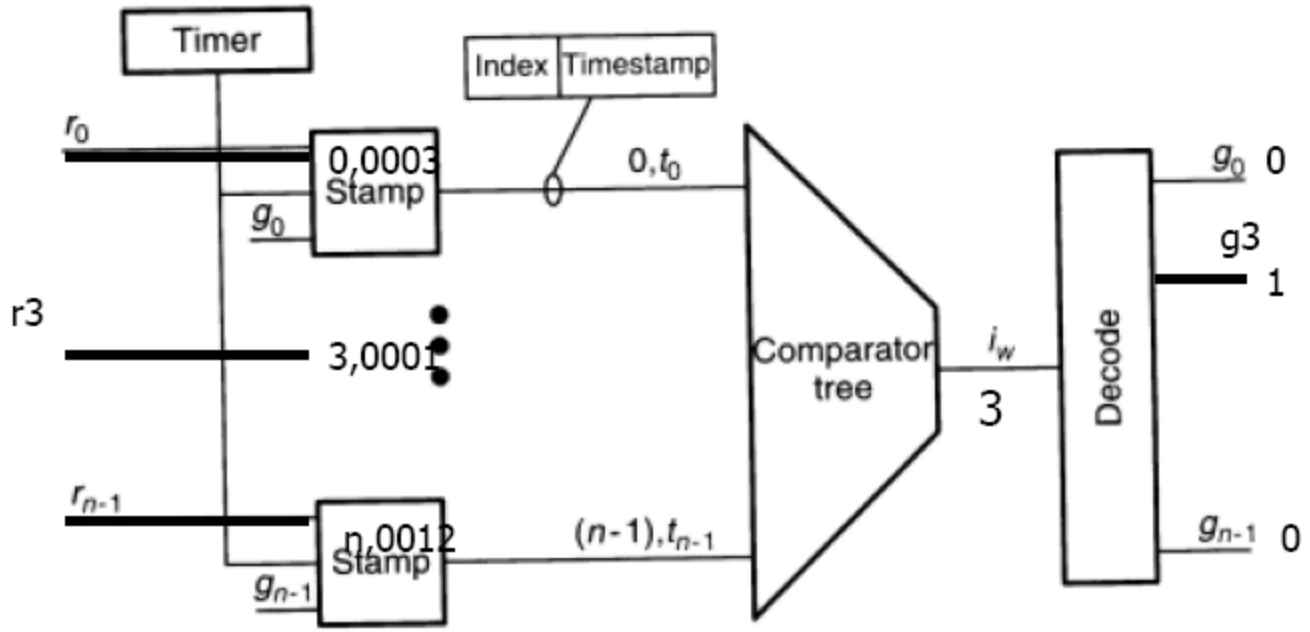
Matrix Arbiter

- A matrix arbiter implements a ***least recently served priority scheme*** by maintaining a triangular array of state bits w_{ij} for all $i < j$
- The Matrix arbiter is very good suited for a small number of inputs, since it is fast, easy to implement and provides strong fairness!
- Matrix must be initialized to a legal state



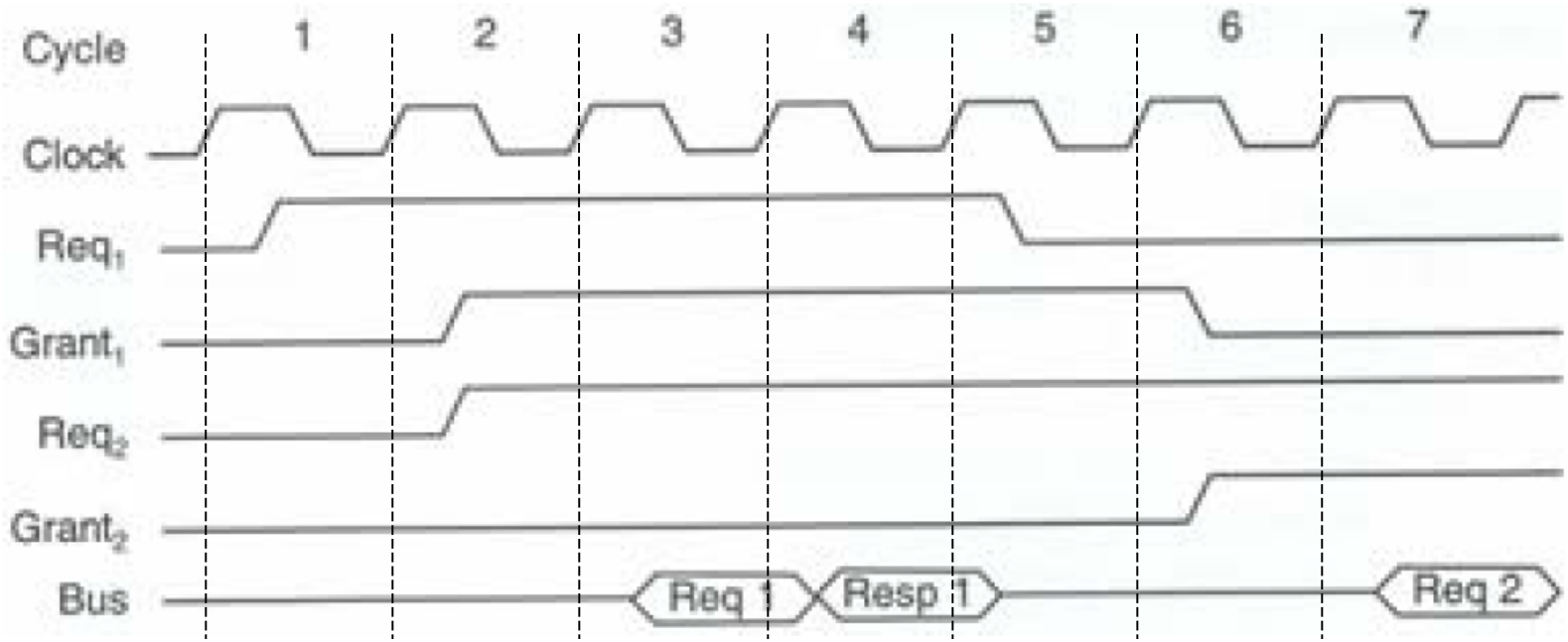
Queuing Arbiter

- A queuing arbiter provides FIFO fairness
- It assigns each request a time stamp when it is asserted
- The request with the earliest time stamp receives the grant



Low Performance Bus Protocol

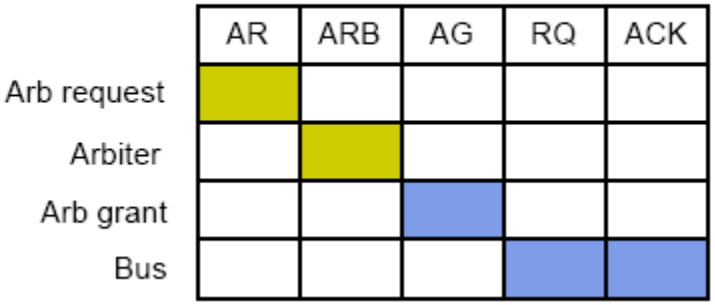
- Without a special bus protocol the bus is not efficiently used
- In the example module 2 requests the bus in cycle 2, but must wait until cycle 6 to receive the grant



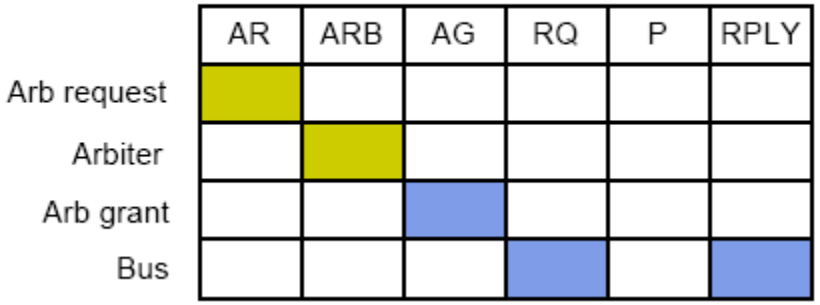
Bus Pipelining

- A memory access consists of several cycles (including arbitration)
- Since the bus is not used in all cycles, pipelining can be used to increase the performance

Write Access



Read Access



- Only one transaction can
  - ➔ Receive the grant during a given cycle
  - ➔ Use the bus during a given cycle

Bus Pipelining

- Pipelining leads to an efficient use of the bus

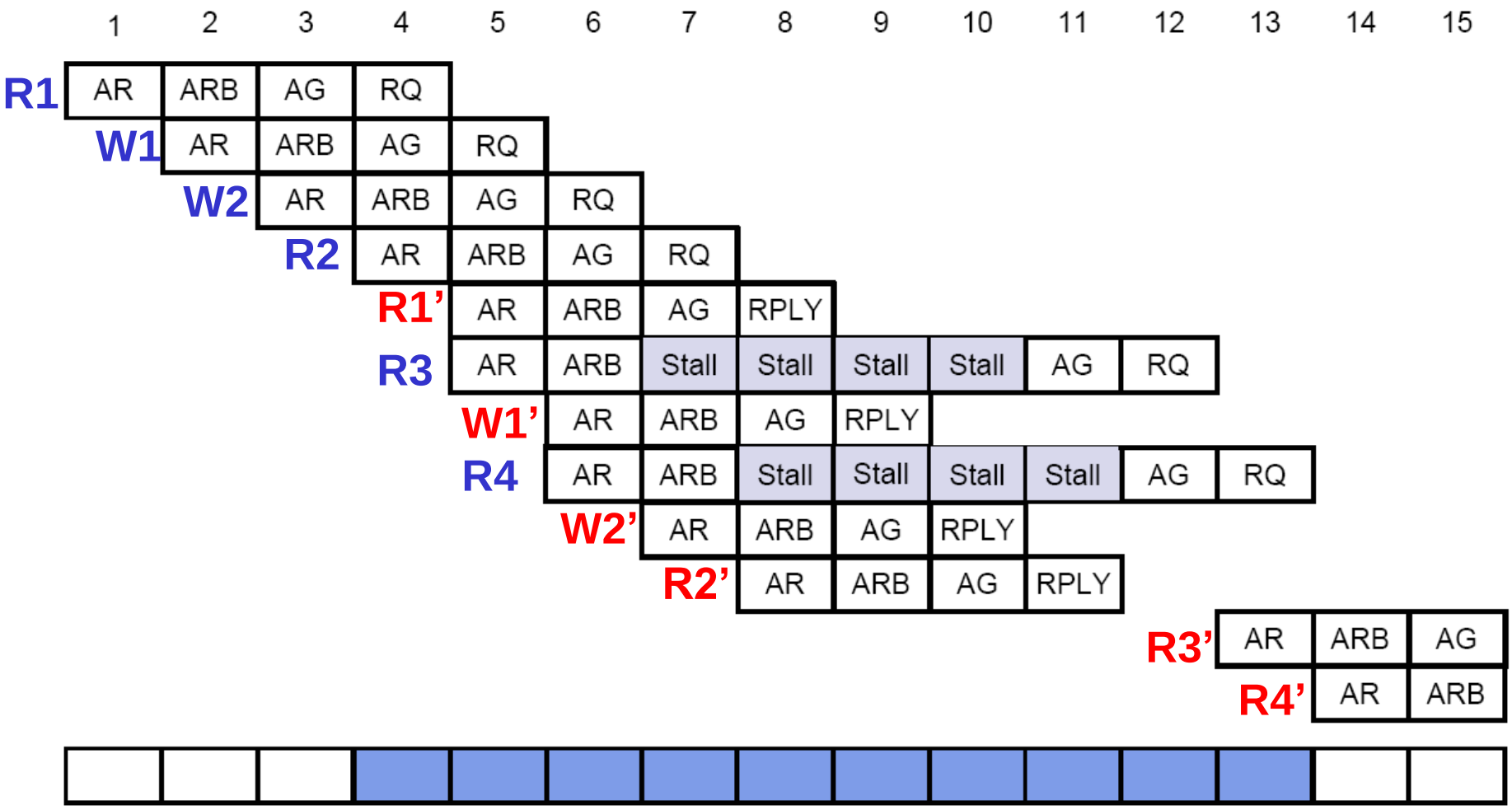
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1. Read	AR	ARB	AG	RQ	P	RPLY									
2. Write		AR	ARB	AG	Stall	Stall	RQ	ACK							
3. Write			AR	ARB	Stall	Stall	AG	Stall	RQ	ACK					
4. Read				AR	Stall	Stall	ARB	Stall	AG	Stall	RQ	P	RPLY		
5. Read							AR	Stall	ARB	Stall	AG	RQ	P	RPLY	
6. Read									AR	Stall	ARB	AG	Stall	Stall	RQ
Bus busy															

- Stalls are inserted since only one instance can use the bus
- Sometimes (cycle 12) two transactions can overlap
- However this cannot be done in cycle 5 (2. Write) since otherwise RPLY and ACK would overlap in cycle 6!

Split-Transaction Bus

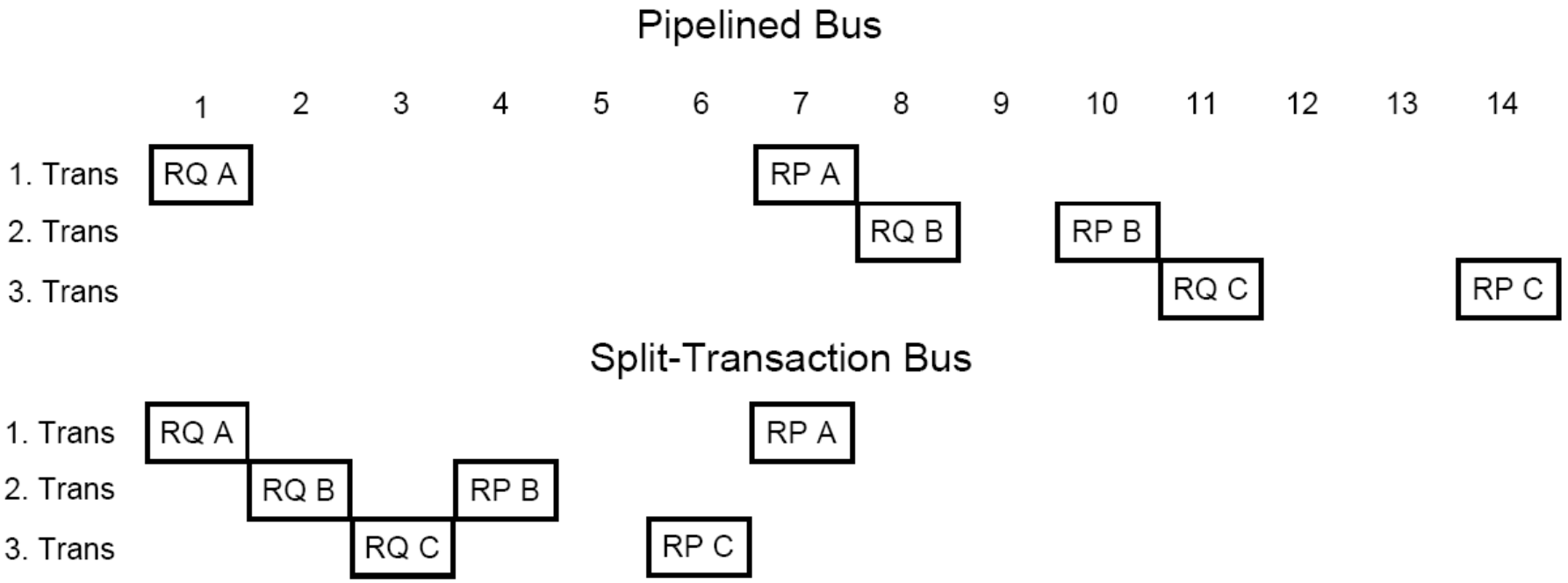
- In a split-transaction bus a transaction is splitted into a two transactions
  - ➔ *request*-transaction
  - ➔ *reply*-transaction
- Both transactions have to compete for the bus by arbitration

Split-Transaction Bus



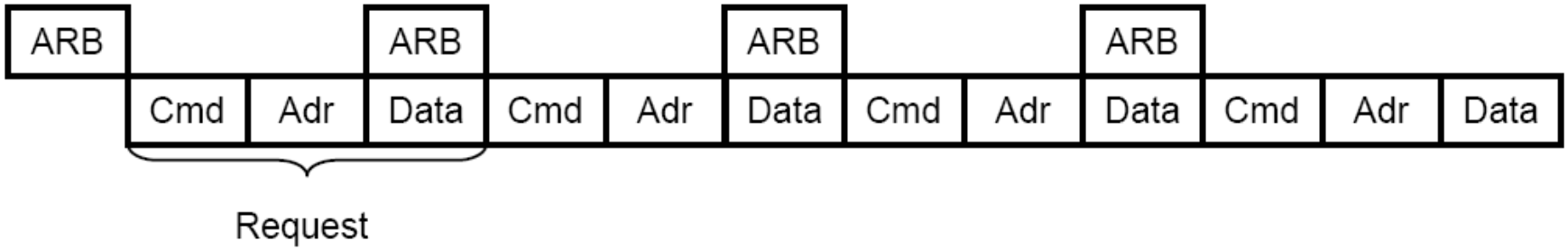
Split-Transaction Bus

- The advantages of the split-transaction bus are evident, if there is a variable delay for requests, since then transactions cannot overlap



Burst Messages

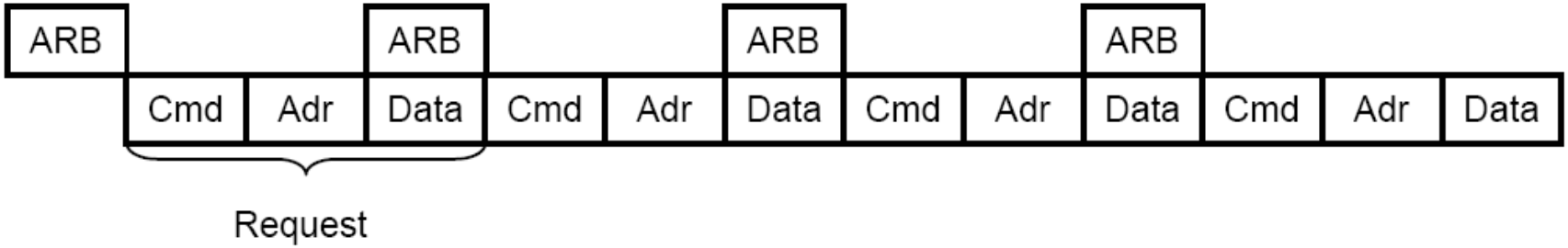
- There is a considerable amount of overhead in a bus transaction
  - ➔ Arbitration
  - ➔ Addressing
  - ➔ Acknowledgement

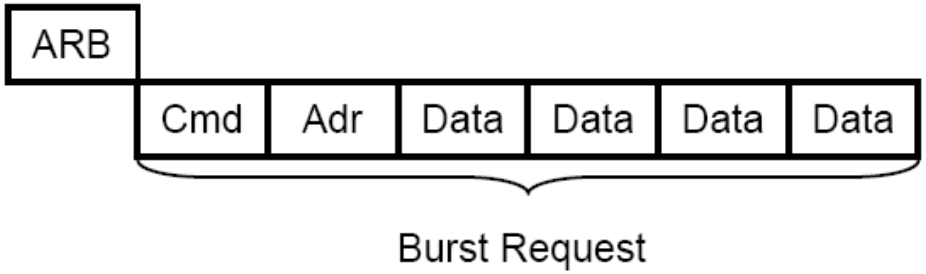


Efficiency = Transmitted Words / Message Size = 1/3

Burst Messages

- The overhead can be reduced, if messages are sent as blocks (bursts)





Efficiency = Transmitted Words / Message Size = 2/3

Burst Messages

- The longer the burst, the better the efficiency
- BUT
  - ➔ Other bus masters have to wait, which may be unacceptable in many systems (Real-Time)
- Possible solution
  - ➔ Maximum length for a burst
  - ➔ Interrupt of long messages
    - ✓ Restart or Resume