Prof. Bernd Finkbeiner
Dr. Calogero Zarba
Moritz Hahn

Sample Solution for Problem Set 8
Due: July 3, 2007, 9:14 am

Embedded Systems

Problem 1 (EDD-scheduling)

For the schedule see figure 1. Maximum latency:

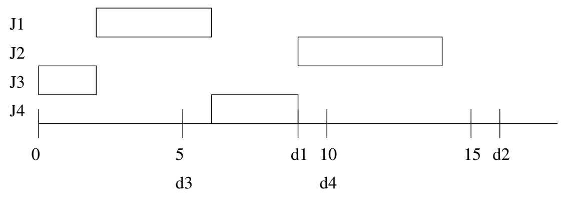

Figure 1: Exercise 1: Schedule

- latency for J_1 is $L_1 = f_1 - d_1 = 6 - 9 = -3$
- latency for J_2 is $L_2 = f_2 - d_2 = 14 - 16 = -2$
- latency for J_3 is $L_3 = f_3 - d_3 = 2 - 5 = -3$
- latency for J_4 is $L_4 = f_4 - d_4 = 9 - 10 = -1$

So, the maximum latency which is the maximum of all these values is -1.
The set of tasks is schedulable because $L_{\max}$ is non-negative.

Problem 2 (Scheduling)

(a) Sort the tasks ascending by their execution time C_i. Execute the tasks in this order without using preemption. The sort takes $O(n \log n)$ time.
(b) Notice first that preemption is again useless here. One can use the same method to show this as was done for EDD in the lecture, just substitute "lateness" by "response time".
Consider a non-preemptive schedule where there is a task τ_i and a task τ_j with $C_i > C_j$ but τ_i is scheduled before τ_j as is illustrated in figure 2. Let A_1 be the set of tasks that are scheduled before τ_i, A_2 the set of tasks scheduled between τ_i and τ_j and A_3 the set of tasks scheduled after τ_j. If one exchanges the execution order of τ_i and τ_j, then the response times of the tasks of A_1 and A_3 are not changed. The new response time f_i' of τ_i becomes the old response time of τ_j. The new response time f_j of τ_j and the response times of A_2 are improved by the difference between the response time of τ_i and τ_j. So, each time we do such an exchange, we get an improvement in the average response time. Finally, all tasks will be sorted by their computation time. Then, you will either have the same schedule as you got by (a) or a schedule which has the same average response time: You can exchange tasks with the same computation time without changing it.

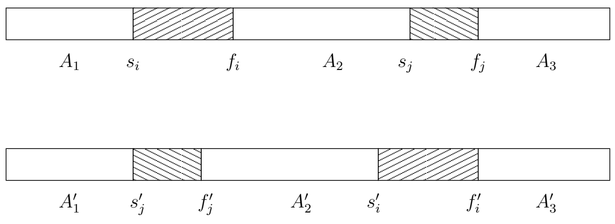

Figure 2: Exercise 2 (b): Reducing average response time by exchanging the order of tasks in a schedule

Problem 3 (WCET/VHDL)

To solve this exercise, we first have to find out about the execution time of the half- and full-adders. Because only NAND-gates may be used, the adders have to be implemented in these gates, as done in figure 3 (other circuits are also possible!). The delay of the sum output half-adder is $D_{\text{SHA}} = 3D_{\text{NAND}} = 30ns$, because the longest path from one input to this output consists of three NAND-gates. The delay of the carry output is $D_{\text{CHA}} = 20ns$. For the full-adder, we have $D_{\text{SFA}} = 60ns$ and $D_{\text{CFA}} = 50ns$. The OR- and AND-gates can be constructed in an obvious way with a delay of $D_{\text{OR}} = D_{\text{AND}} = 20ns$.
Now we can talk about the delay of the whole circuit. One longest path, which gives the response time of the whole circuit, is the one which states from the input of the half-adder in the upper left, goes through $e3$ and finally ends in d. Its delay is

$$D = D_{\text{AND}} + D_{\text{CHA}} + D_{\text{SFA}} + D_{\text{CHA}} + D_{\text{SFA}} + D_{\text{CHA}} + 4D_{\text{OR}} = 280ns$$

Problem 4 (WCET/Cache analysis)

(a) Consider figure 4: In the beginning we will have two cache misses because the cache is empty or filled with junk. After these accesses the cache is $\{b\}\{a\}$ because of the LRU rule. In the future there will only be cache hits, because now the cache is filled with a and b and there are no accesses to other elements. For the join of the two branches we will have different caches for the may- and the must-analysis because different merging rules must be used. However, It will result in the cache $\{a\}\{b\}$ in both cases.
In each case we have a hit for the last command.
Notice that for this analysis we assumed the cache to be invalid or empty initially. If you can't assume this, for the must analysis you have to assume that it is empty or filled with useless data but for the may analysis it should have the content $\{a, b\}$ in the first and $\{\}$ in the second cell.
(b) See figure 5: In contrary to (a) the cache does not change after the first two accesses: In FIFO we would have to throw out one entry if we had a cache miss. However, we never have one after the first two accesses, so the cache stays constant then.
In each case we have a hit for the last command.
(c) See figure 6: Whether one has a hit or a miss in the last step depends on the cache status. For example, if one has the command sequence *abba* and the cache with content *cc* initially, then one will have a hit in the last step. However, with the cache content *ca* initially and the same sequence, a miss will occur for the last command.

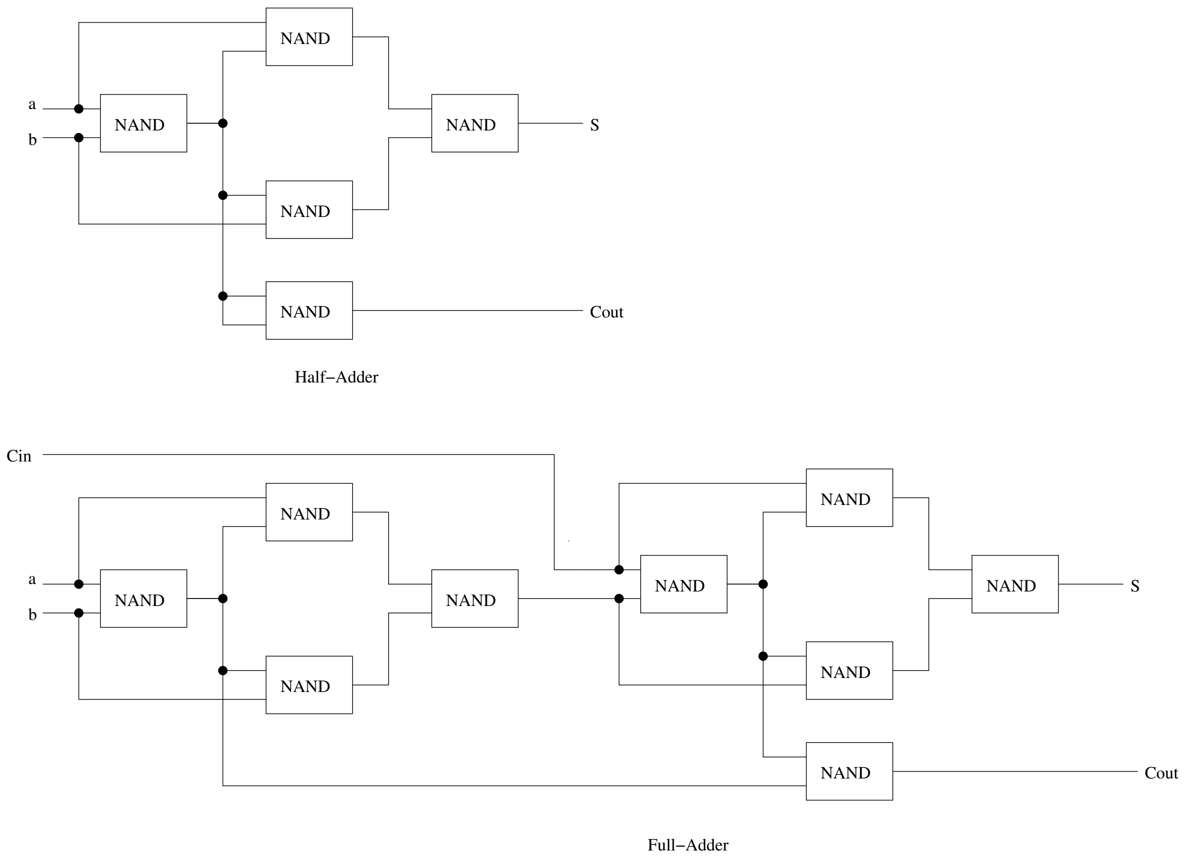

Figure 3: Exercise 2: Half- and full-adder circuits made of NAND gates

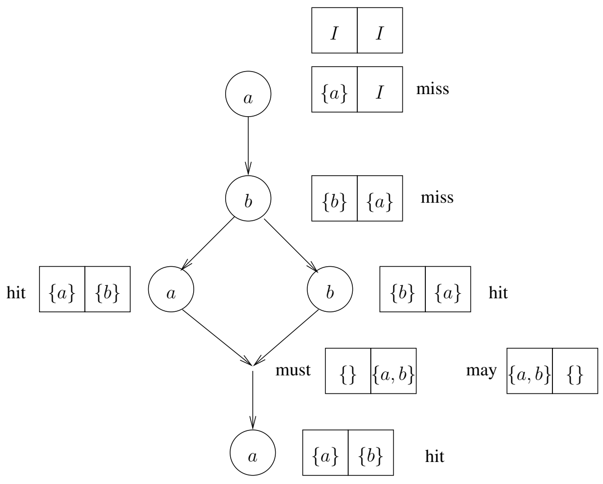

Figure 4: Exercise 3 (a): Must and may cache analysis with LRU

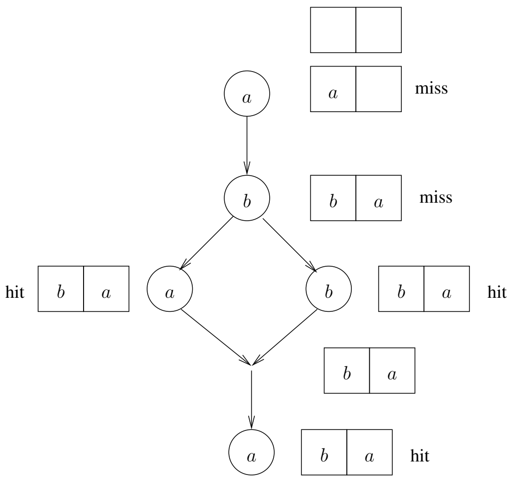

Figure 5: Exercise 3 (b): FIFO analysis with cache initially empty

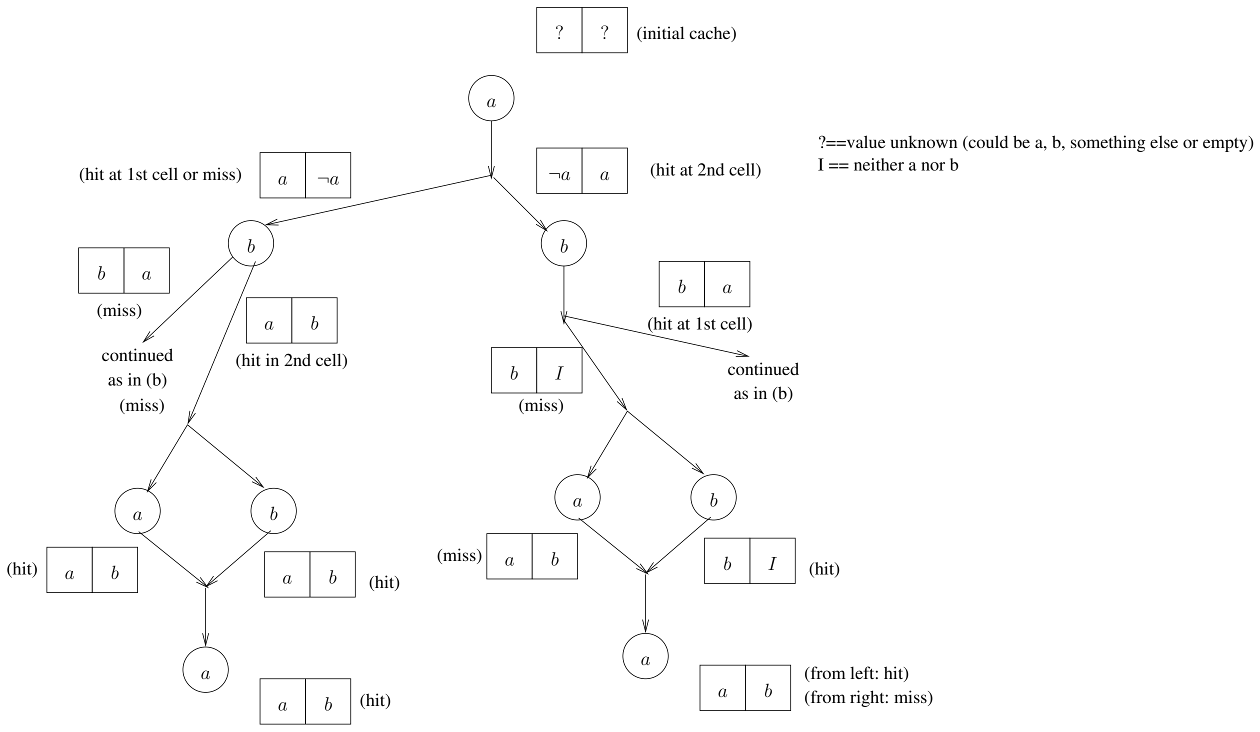

Figure 6: Exercise 3 (c): FIFO analysis with cache initially unknown