Part I
Automotive Systems

1 051214 Eyad Alkassar Introduction

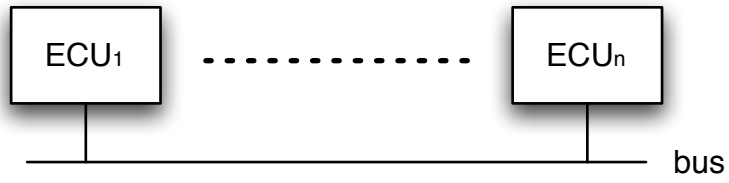


Figure 1: Distributed System

Our analysis and modeling of an automotive system will be done in the following framework (see Figure 1):

- n processors compute different tasks in a distributed manner. These processors are denoted with $ECU_u, u \in [1:n]$ (**e**lectronic **c**ontrol **u**nit)
- Each ECU has a single processor (VAMP) and communicates with other $ECUs$ over a bus system. The bus interface is specified in the so called *FlexRay* standard.
- Since communication in the FlexRay bus is timed, clocks play a crucial role in our models. Each ECU_u has an individual oscillator, with the properties
  - clock frequency ν
  - clock period $\tau_u = \frac{1}{\nu_u}$
- The physical realization of these oscillators only guarantees bounded clock drift:
  - bounded clock drift: $(\tau_u - \tau_{u'}) \leq \tau_{max} \cdot \delta$ with $\delta = 0.15\%$ and $\tau_{max} = max\{\tau_u | 1 \leq u \leq n\}$

2 Structure of the Lecture

The rest of the lecture will be structured in the following way:

1. **Serial interfaces**
2. **FlexRay-like Bus interface**

- construction
- integration with processors

3. **WCET: worst case execution time**

   - Based on WCET analysis we will show theorems of the following form: Having knowledge about the concrete C-program P, the used compiler, the underlying hardware it holds that i) P is correct ii) P terminates in less than T cycles.
   - The analysis of WCET is based on techniques from the UdS Spin-Off *AbsInt*. This analysis is based not only on the considered code in *Assembler* or *C*, but also on the gate-level implementation of the processor.

4. **OLOS: OSEK-time like OS**

   In this chapter we will build upon the hardware and FlexRay model an Operating System called OLOS (OLOS is a dialect of *Communicating Virtual Machines* or in short *CVM*. CVM implements the basic functionality of a μ-kernel). The Operating System is running on each ECU and provides task abstraction and communication primitives. Furthermore it implements the drivers for the FlexRay interfaces.

5. **D-OLOS: distributed OLOS**

   In this chapter we will connect many different $ECUs$ with OLOS running on top of them. This will provide us with the complete programming level of the user.

3 Serial Interfaces

3.1 Some formal stuff

We will use the following notations (where $a, b \in Z$):

- $[a : b] = \{a, a + 1, \ldots, b\}$
- $[a : b) = [a : b - 1]$,
  $(a : b) = [a + 1 : b - 1]$,
  $(a : b] = [a + 1 : b]$.
- $c + [a : b] = [a : b] + c = [a + c : b + c]$
- We model time as the set of real numbers: $time = R_0^+$

- A signal is a mapping from time to the values 0, 1 or Ω (which denotes an unknown value). $time \rightarrow \{0, 1, \Omega\}$

The content on the bus is written into the register, whenever the clock signal is set to one. Clocks to the registers are modeled in the following way:

- A clock is modeled as function, mapping time to boolean values, i.e. $c_k : time \rightarrow \{0, 1\}$
- A clock oscillates between the values 1 and 0. The ith time it changes it value from 0 to 1 (this position in time is called raising edge, the whole period between two raising edges is called cycle) is $e(i) = \alpha + i \cdot \tau$ with $i \in N_0$, where α is some offset value of the clock.
- With that we can define: $c_k(t) \equiv \exists i : t \in [e(i) : e(i) + \tau/2]$

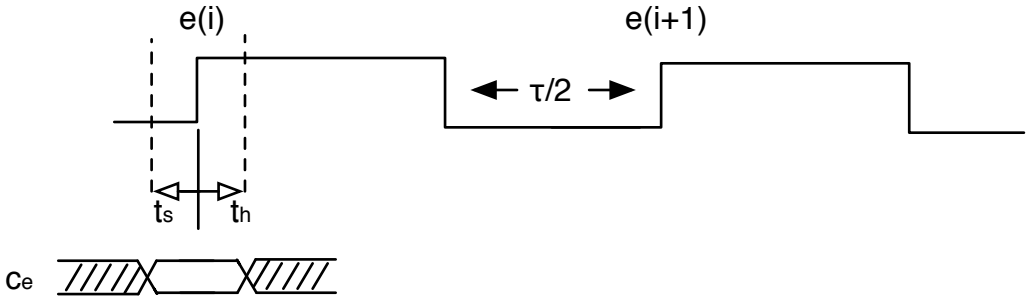


Figure 2: Holding and Setup time for a register reading from the bus on a raising edge

We have two operating conditions for a register at every raising edge $e(i)$ when some data is sampled: 1) the clock enable signal must stay stable and 2) if it is set to 1 (i.e. updating) the input signal must stay stable:

- **Clock enable stable** $\exists y \in \{0, 1\}\ \forall t \in (e(i) + [-t_s, t_h]) : ce(t) = y$, where y denotes whether there should be an update or not. t_s and t_h stand for setup and hold time (see Figure 2). The period $e(i) + [-t_s, t_h]$ is called sampling interval.
- **Data input stable** Let $B : time \rightarrow \{0, 1, \Omega\}$ be some input signal. If for the whole sampling interval the clock enable signal stays stable then it holds: $\exists x \in \{0, 1\}\ \forall t \in (e(i) + [-t_s, t_h]) : B(t) = x$.

Next we define the value a register holds at time t between two raising edges $e(i)$ and $e(i + 1)$. There are three periods (see Figure 3). In the first period the

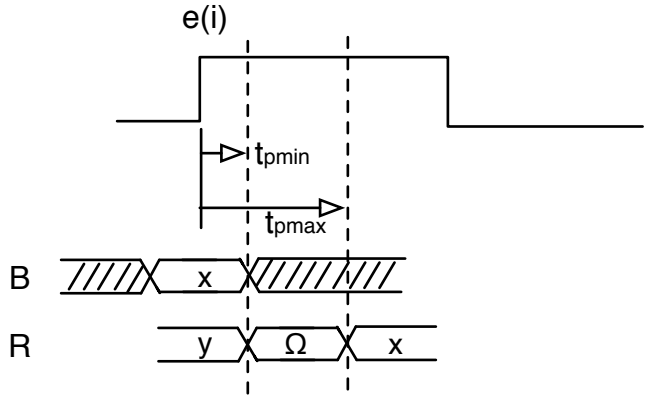


Figure 3: The content of the Register when reading from the bus at edge e(i)

content of the register stays the old one (before the first raising edge). Then it follows a period where the value is flipping and is therefore undefined. This period lasts from $e(i)+t_{pmin}$ to $e(i)+t_{pmax}$. Finally the Register holds the input value x:

$\forall t \in (e(i) : e(i+1)]$ if $ce(t) = 1$ then

$$R(t) \quad = \quad \begin{cases} R(e(i)) & : \quad t \in e(i) + (0 : t_{pmin}] \\ \Omega : & : \quad t \in e(i) + (t_{pmin} : t_{pmax}) \\ x : & : \quad t \in [e(i) + t_{pmax} : e(i+1)] \end{cases}$$

We define the value R^i to be the content of Register R at the end of cycle i, i.e. $R^i = R(e(i+1))$.

If the second operating condition of a register is violeted, i.e. the input data is not stable during the sampling interval of edge i, it could happen (with small probability) that the content of the register is undefined even after $e(i)+t_{pmax}$. This phenomenon is called *meta stability*. To avoid meta stability we read the content of the first Register into a second one, called $\widehat{R}$. Register $\widehat{R}$ is clocked as the first one. By that construction (see Figure 4) we lower the probability that $\widehat{R}$ is meta stable after $e(i) + t_{pmax}$ (from p for register R to, p^2 for $\widehat{R}$), i.e. it practically holds:

$$\forall i : \exists \widehat{x} \in \{0,1\} : \forall t \in (e(i) + t_{pmax}, e(i+1) + t_{pmin}] : \widehat{R}(t) = \widehat{x}$$

This only holds if t_{pmin} is greater or equal to the Register holding time. Else the second operating condition would be violated for Register $\widehat{R}$.

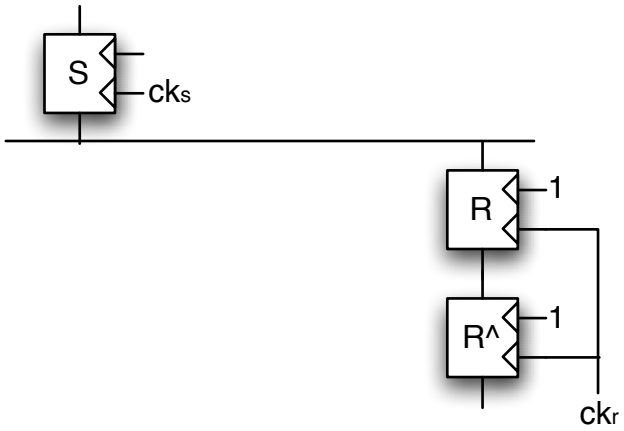
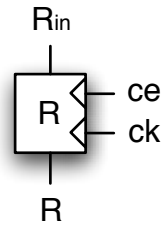


Figure 4: Bus link structure of Sender (S) and Receiver(R)

4 051219 Sebastian Bogan FlexRay Bus interface

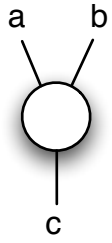
Register



$$R^{i+1} = \begin{cases} R^i_{in} & : c^i_e = 1 \\ R^i & : \text{otherwise} \end{cases}$$

Figure 5: Register R

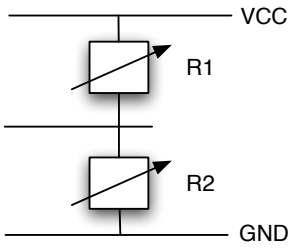
Gates



$$g \in \{\wedge, \vee, \ldots\}, \;\; c^i = g(a^i, b^i)$$

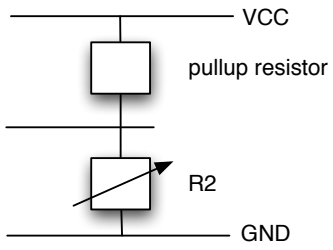
Figure 6: Gate g

Open Collector Outputs



$$\begin{array}{ll} 1 & : R_1\ low \wedge\ R_2\ high \\ 0 & : R_1\ high \wedge\ R_2\ low \\ highZ & : R_1\ high \wedge\ R_2\ high \end{array}$$

Figure 7: Open Collector



$1 \quad : R_2\ high$
$0 \quad : R_2\ low$

Figure 8: Pullup Resistor

We define the value of the Bus B at time t as conjunction over all sender values.

$$B(t) = S(t) \wedge \ldots \wedge S'(t)$$

With

$$0 \wedge \Omega = 0,\ \Omega \wedge 1 = 1,\ 1 \wedge \Omega = 1,\ \Omega \wedge \Omega = \Omega,\ \Omega \wedge 0 = 0$$

We define the content of the registers R (connected to bus B) and $\hat{R}$ at the time $e_r(j)$ as follows (Figure 4).

$$R^j = \begin{cases} B(e_r(j)) & : B(t) = B(e_r(j))\ \ \forall t \in e_r(j) + [-t_s, t_h] \\ \Omega & : otherwise \end{cases}$$

$$\hat{R}^j = \begin{cases} R^{j-1} & : R^{j-1} \in \{0,1\} \\ x \in \{0,1\} & : otherwise \end{cases}$$

Affected Cycles

Assume a sender puts a new value on the bus at time $e_s(i)$. Then for all receiver edges $e_r(j)$ such that $e_r(j) + t_h \leq e_s(i)$ sampling is not affected by this new value (not considering propagation delays).

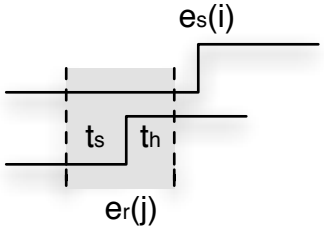


Figure 9: Not affected sampling

Definition 4.1 *$cy(i)$ is the index of the first receiver edge, that is affected by $e_s(i)$.*

$$cy(i) = \min\{j \mid e_r(j) + t_h > e_s(i)\}$$

That means, that a receiver edge j is affected by a sender edge i if it is in the region $(e_s(i) - t_h, e_s(i) - t_h + \tau_r]$.

$$j = cy(i) \implies e_s(i) - t_h < e_r(j) \leq e_s(i) - t_h + \tau_r$$

The formula above could as well be written as:

$$j = cy(i) \implies e_s(i) - t_h < e_r(j) \wedge e_r(j-1) \leq e_s(i) - t_h$$

or equivalently:

$$j = cy(i) \implies e_r(j-1) \leq e_s(i) - t_h < e_r(j)$$

From the FlexRay standard we know, the sender puts all bits 8 times on the bus, e.g.:

$$c_{e_s}^{i-1} = 1 \wedge c_{e_s}^{i} \ldots c_{e_s}^{i+7} = 0 \implies \forall t \in [e_s(i) + t_{\text{p-max}}, e_s(i+7)] : B(t) = S^i$$

That means the receiver samples S^i during at least 7 consecutive cycles.

Lemma 4.1 *7 consecutive cycles*

$$R^{cy(i)+\beta+k} = S^i$$

where

$$\begin{aligned} k &\in [0:6] \\ \beta &= \begin{cases} 0 & \textit{if } e_r(cy(i)) \geq e_s(i) + t_{\textit{p-max}} + t_s \\ 1 & \textit{otherwise} \end{cases} \end{aligned}$$

Proof. All sampling intervals of all receiver edges $cy(i) + k + \beta$ are in the region of the time where the bus is stable. Both for $\beta = 0$ and $\beta = 1$.

$\beta = 0$:

$$\begin{aligned} e_r(cy(i)) + 6\tau_r + t_h &\leq e_s(i) - t_h + \tau_r + 6\tau_r + t_h && \text{(Definition 4.1)} \\ &= e_s(i) + 7\tau_r \\ &< e_s(i) + 8\tau_s && \text{(bounded clock drift)} \end{aligned}$$

$\beta = 1$:

$$\begin{aligned} e_r(cy(i)) + 7\tau_r + t_h &< e_s(i) + t_{\text{p-max}} + t_s + 7\tau_r + t_h && \text{(Definition } \beta\text{)} \\ &\leq e_s(i) + \tau_{max}(1/2 + 1/10 + 7 + 1/10) && \text{(Definition } t_{\text{p-max}}, t_s, t_h\text{)} \\ &= e_s(i) + 7.7\tau_{max} \\ &< e_s(i) + 8\tau_s && \text{(bounded clock drift)} \end{aligned}$$

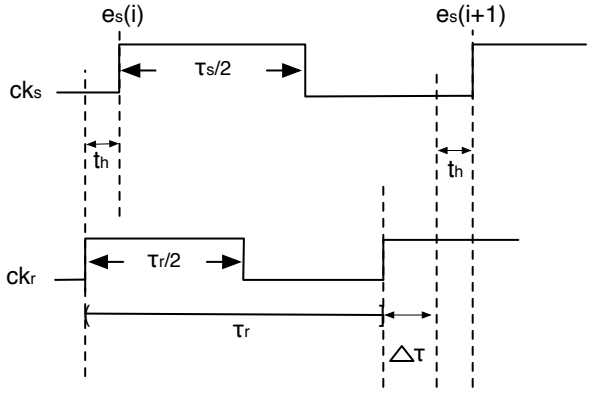


Figure 10: A situation where the same sent value affects the receiver in two cycles

5 051221 Matthias Daum

Lemma 5.1 *Values that were put on the bus at consecutive sender cycles are usually sampled at consecutive receiver cycles. More precisely,*

1. $cy(i+1) \in [cy(i) : cy(i)+2]$
2. *If* $cy(i+1) \neq cy(i)+1$, *then* $cy(i+1+k) = cy(i+1)+k$ *for all* $k \in [0:600]$.[1]

Proof. Both statements of the lemma are strongly related, hence we show both of them simultaneously.

Case distinction.

At first, we consider the case that $\tau_r \leq \tau_s$. The other case, $\tau_r > \tau_s$, is left for an exercise.

We define $\Delta\tau = \tau_s - \tau_r$ and $a = cy(i)$.

Now, we have to find out, under which circumstances $cy(i+1)$ is not equal to $cy(i)+1$. Such a situation is depicted in Figure 10: The sender puts a new value on the bus at time $e_s(i)$ and the receiver had its rising edge immediately after $e_s(i) - t_h$. Hence, the receiver will be affected by the sender in this cycle. However, the sampling interval in the next receiver cycle will already be over before the sender's next cycle starts. Consequently, the receiver will again sample the old value and the new value on its next cycle, i.e. $cy(i+1) = cy(i)+2$. Apparently, we experience this situation whenever $e_r(a) \leq e_s(i) - t_h + \Delta\tau$.

Case distinction.

Case 1: $e_r(a) \leq e_s(i) - t_h + \Delta\tau$

[1] Assuming a $\delta \leq 0.15\%$. Was corrected later to $k \leq 300$ for the new $\delta \leq 0.30\%$.

$$
\begin{aligned}
\implies e_r(a) + \tau_r &\leq e_s(i) - t_h + \Delta\tau + \tau_r \\
&= e_s(i) - t_h + \tau_s - \tau_r + \tau_r \\
\implies e_r(a+1) &\leq e_s(i) - t_h + \tau_s \\
&= e_s(i+1) - t_h
\end{aligned}
$$

And from $e_s(i) - t_h < e_r(a)$ it follows that

$$
\begin{aligned}
e_r(a) + 2\tau_r &> e_s(i) - t_h + 2\tau_r \\
&= e_s(i) - t_h + \tau_s + \tau_r - \Delta\tau \\
&= e_s(i+1) - t_h + \underbrace{\tau_r - \Delta\tau}_{>0} \\
\implies \quad e_r(a+2) &> e_s(i+1) - t_h
\end{aligned}
$$

We have just shown that $e_r(a+1) \leq e_s(i+1) - t_h < e_r(a+2)$. Hence, the value that was put on the bus at $e_s(i+1) - t_h$, will be sampled in cycle $a+2$, i. e.,

$$
cy(i+1) = a + 2 = cy(i) + 2 \ .
$$

Herewith, we have shown that statement 1 holds if $\tau_r \leq \tau_s$ and $e_r(a) \leq e_s(i) - t_h + \Delta\tau$. Now, we will consider statement 2 of the lemma.

We know that $e_s(i) - t_h < e_r(a)$, and hence:

$$
\begin{aligned}
e_r(a) + (1+k)\tau_r &> e_s(i) - t_h + (1+k)\tau_r \\
&= e_s(i) - t_h + k\tau_s - k\Delta\tau + \tau_r \\
&= e_s(i+k) - t_h - k\Delta\tau + \tau_r
\end{aligned}
$$

And thus, if $k\,\Delta\tau \leq \tau_r$, we can conclude

$$
e_r(a+k+1) > e_s(i+k) - t_h
$$

We have assumed that $e_r(a) \leq e_s(i) - t_h + \Delta\tau$, hence

$$
\begin{aligned}
e_r(a) + k\tau_r &\leq e_s(i) - t_h + \Delta\tau + k\tau_r \\
&= e_s(i) - t_h + \Delta\tau + k\tau_s - k\Delta\tau \\
&= e_s(i+k) - t_h - (k-1)\Delta\tau
\end{aligned}
$$

And thus, for $k \geq 1$, we can conclude

$$
\implies e_r(a+k) \leq e_s(i+k) - t_h
$$

We have just shown that $e_r(a+k) \leq e_s(i+k) - t_h < e_r(a+k+1)$ holds for $1 \leq k \leq \tau_r/\Delta\tau$. Hence, the value that was put on the bus at $e_s(i+k) - t_h$, will be sampled in cycle $a+k+1$, i. e.,

$$
\forall k.1 \leq k \leq \tau_r/\Delta\tau \implies cy(i+k+1) = a + k + 1 = cy(i) + k + 1 \ .
$$

The computation of the numeric upper bound of k is left as an exercise.

Case 2: $e_r(a) > e_s(i) - t_h + \Delta\tau$

As motivated at the beginning of our proof, $cy(i+1)$ is always $cy(i)+1$ in this case:

$$
\begin{aligned}
e_r(a+1) &> e_s(i) - t_h + \Delta\tau + \tau_r && \text{(assumption in Case 2)} \\
&= e_s(i) - t_h + \tau_s \\
&= e_s(i+1) - t_h \\
e_r(a) &\leq e_s(i) - t_h + \tau_r && \text{(Definition 4.1)} \\
&\leq e_s(i) - t_h + \tau_s && (\tau_r \leq \tau_s) \\
&= e_s(i+1) - t_h \\
\implies cy(i+1) &= cy(i) + 1
\end{aligned}
$$

□

6 060109 Jan Dörrenbächer

Putting a message $m[0:L-1]$ on the bus, where $m[i] \in \{0,1\}^8$ is message byte i.

When a message is put on the bus, we have to encode the actual message. In the following, we define the message encoding. At first, we define a few tokens:

- $TSS = 0$ – Transmission start sequence
- $FSS = 1$ – Frame start sequence
- $BSS[0:1] = 10$ – Byte start sequence. Note: the transition from 1 to 0 forces so called *sync edges* on the bus.
- $FES = 0$ – Frame end sequence
- $TES = 1$ – Transmission end sequence

We denote the encoded message m by $f(m)$.
The encoded message looks as follows:

$$f(m) = TSS \circ FSS \circ BSS \circ m[0] \circ \cdots \circ BSS \circ m[L-1] \circ FES \circ TES$$

The length of the encoded message measured in bits is $|f(m)| = 4+10 \cdot L = n$.

For the sake of error correction, every bit of the encoded message is transmitted eight times. For some $x = x[0:k-1] \in \{0,1\}^k$, we use the notation $8 \cdot x$ for $x[0]^8 \circ \cdots \circ x[k-1]^8$.
Thus, in order to deliver the message, we transmit $8 \cdot f(m)$.
On sender side, we expect a $y = y[0:n-1] \in \{0,1\}^n$ and $y = f(m)$, especially.

Definition 6.1 *The predicate $onbus(y, \gamma, \tau)$ is defined as follows:*

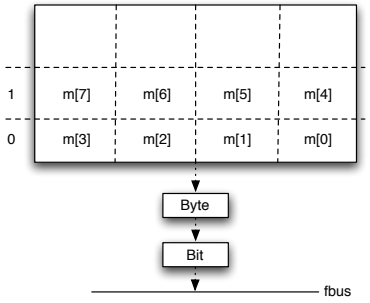
$$\forall j \in [0:n-1].\forall t \in [\gamma + 8 \cdot \tau \cdot j + t_{p_{max}} : \gamma + 8 \cdot \tau \cdot (j+1)] : B^t = y_j$$

Where y denotes the message $f(m)$, γ the starting point and τ the cycle time of the sender. The number of bits in the message is n.

Remark:
If the predicate $onbus(y, \gamma, \tau)$ holds, one can assume, that at some sender edge $e_s(i) = \gamma = \alpha_s + \tau_s \cdot i$, the message was generated by the flip flop.

SIO: serial I/O interface

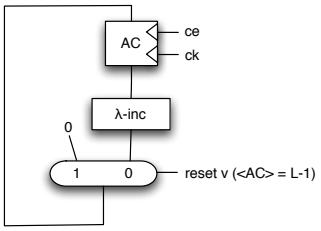
On sender side we have a word adressable message send buffer MS (see Fig. 11.



$$MS(b) = m(4b+3) \circ \cdots \circ m(4b)$$
$$8 \cdot f(m) = 1 = Bit^{y} \ldots Bit^{y+8(4+10L)}$$
$$Bit^{z}_{ce} : z = y-1, y+7, y+8j-1 \; for \; j \geq 2$$

Figure 11: Message Send Buffer

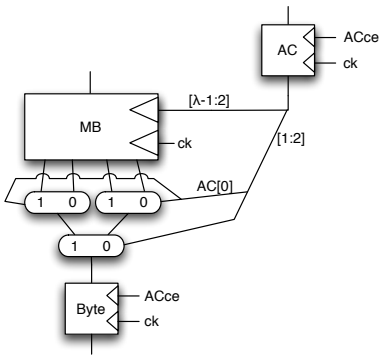
By means of the address counter in figure 12 we choose the appropriate byte within the word, stored in the message buffer. The address is incremented after the previous byte is transmitted. A byte transmission takes 80 cycles. If a reset signal is triggered, or if $\langle AC \rangle = L - 1$, the counter is reset.



$$\lambda = \lceil \log L \rceil$$
$$AC^{t}_{ce} = 1 \Leftrightarrow t = t_0 + 80k = t_k$$
$$for \; k = 0, \ldots, L-1$$
$$AC^{0}_{ce} = 0$$
$$\langle AC^{t} \rangle = i \Leftrightarrow t \in (t_{i-1} : t_i]$$
$$\langle AC^{t} \rangle = 0 \;\Leftrightarrow\; t < t_0 \vee t > t_{L-1}$$

Figure 12: Address Counter

By means of the byte generation circuit (figure 13) we get the byte which is transmitted later on. We take the 32-bit message out of the message buffer and select the desired byte using multiplexers.



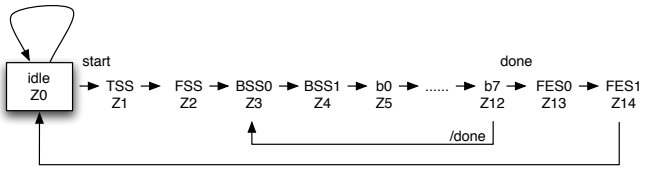
$$ACce^t = 1 \Leftrightarrow \quad t = t_k$$
$$= t_0 + 80 \cdot k, \; with \; k = 0, \ldots, L-1$$

$$AC^t = i \Leftrightarrow \quad (i = 0 \wedge t \leq t_0)$$
$$\vee (i > 0 \wedge t \in (t_{i-1}, t_i])$$
$$\vee (i = 0 \wedge t > t_{i-1})$$

$$Byte^t = m_i \Leftrightarrow \quad (i < L-1 \wedge t \in (t_{i-1}, t_i]$$
$$\vee (i = L-1 \wedge t > t_{L-1})$$

Figure 13: Byte Generation

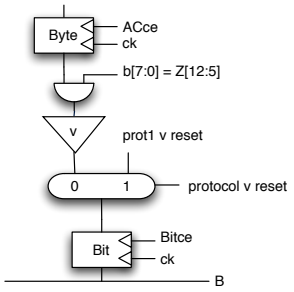
The flex ray controller is controlled by the state automaton depicted in figure 14. The signal $start$ is computed by the circuit in figure 27.



$$start^t = start'^{t-7} \vee \cdots \vee start'^t$$
$$done^t = (AC^t = 0)$$

Figure 14: State Automaton

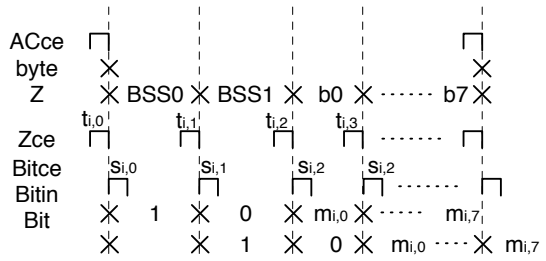
Figure 15 depicts the circuit which is used to determine the bit that should be transmitted.



$$prot1 = Z_2 \vee Z_3 \vee Z_{14}$$
$$protocol = \neg \bigvee_i l[i]$$

Figure 15: Circuit determines the transmitted bit

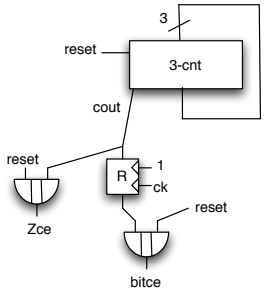
The timing diagram below shows the dependency of the bit which is put on the bus and the current state. We have the constant values 1 and 0, according to the states $BSS0$ and $BSS1$, which announce the byte start sequence. Afterwards, the actual bits 0 to 7, of the message, follow in the particular states $B0$ to $B7$.



$$
\begin{aligned}
Bit^t = m_{i,j}, \quad & t \in (S_{i,j+2}, S_{i,j+3}] \\
Bit^t = 1, \quad & t \in (S_{i,0} : S_{i,1}] \\
Bit^t = 0, \quad & t \in (S_{i,1} : S_{i,2}]
\end{aligned}
$$

Figure 16: Timing Diagram 1

The clock enable signal bit_{ce} for the bit register is computed by the circuit in figure 17. Since a bit has to be transmitted 8 times, we update the value of the bit register only after 8 cycles. To count the cycles we use a 3-bit counter.



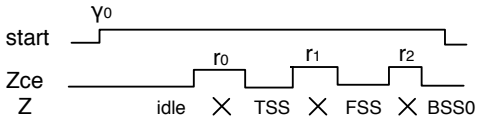
$Z_{ce} = c_{out} \vee reset$
(c_{out} is the overflow bit of the counter)
$bit^t_{ce} = reset^t \vee cout^{t-1}$

Figure 17: Computation of signal bit_{ce}

Figure 18 shows the start period of the transmission. According to the state, denoted by Z the bit is set to the $TSS, FSS, \ldots$.
According to the diagram in figure 18, the predicate

$$onbus(f(m), \gamma, \tau),\ \gamma \leq \gamma_0 + (7+1) \cdot \tau$$

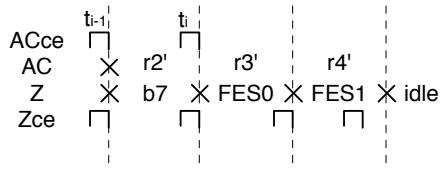
holds.



$$\begin{aligned}
r_0 &= min\{t | start^t \wedge Z_{ce}^t \leq \gamma_0 + 7 \cdot \tau\} \\
r_1 &= r_0 + 8 \\
r_2 &= r_0 + 16 = t_0 \\
Bit^t &= 0 : t \in (r_0 + 1, r_1 + 1] \\
Bit^t &= 1 : t \in (r_1 + 1, r2 + 1]
\end{aligned}$$

Figure 18: Timing Diagram 2

The end of the transmission is illustrated in figure 19. Here, Bit is set to $FES0$ and $FES1$. After the transmission we switch into the idle state.



$$\begin{aligned}
r_3' &= r_2' + 8 \\
r_4' &= r_3' + 8 \\
Bit^t &= \begin{cases} 0: & t \in (r_2' + 1 : r_3' + 1] \\ 1: & t \in (r_3' + 1 : r_4' + 1] \end{cases}
\end{aligned}$$

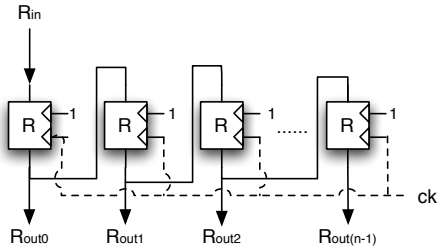
Figure 19: Timing Diagram 3

6.1 060111 Sebastian Bogan Receiver Construction

The present section defines the receiver construction. We start off defining the hardware. Within a series of lemmas we prove the receiver construction. At the end we prove that an entire message is sampled correctly.

6.1.1 Hardware

An n-bit shift register is constructed in the following way.



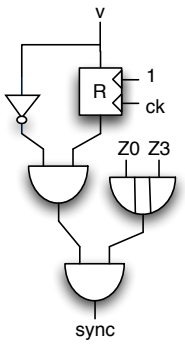
$$\begin{aligned} R^t_{out_n} &= R^{t-1}_{out_{n-1}} \\ &\vdots \\ R^t_{out_1} &= R^{t-1}_{out_0} \end{aligned}$$

Figure 20: n-Bit Shift Register

Definition 6.2 *An n-bit majority voter realizes the following function.*

$$n\text{-}major(x[0:n-1]) = 1 \Leftrightarrow \mid \{i \mid x_i = 1\} \mid \geq \lceil n/2 \rceil$$

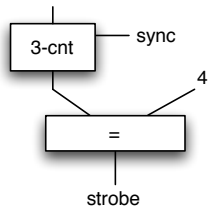
Definition 6.3 *The sync signal $sync$ is turned on in cycle t if the decoding state is either $idle$ (Z_0) or $BSS0$ (Z_3) and there is a falling edge (Figure 21). An n-bit shift register is constructed in the following way.*



$$\begin{aligned} sync^t = \quad & (idle^t \vee BSS0^t) \ \wedge \\ & (\neg v^t \wedge \ v^{t-1}) \end{aligned}$$

Figure 21: Sync

Definition 6.4 *$sy(h)$ denotes the h'th cycle after $cy(0)$, when sync is activated.*



$$strobe = (cnt = 4) \wedge \neg sync$$

Figure 22: 3-cnt + strobe

Definition 6.5 *The decoding automaton (Figure 22) is clocked by the strobe signal. By means of a 3-bit counter and the* $sync$ *signal one can define the* $strobe$ *signal. The strobe signal is realized as follows.*

Definition 6.6 $str(k)$ *denotes the* k*'th cycle, when strobe is activated after* $cy(0)$.

Now we assemble all pieces to construct the receiver.

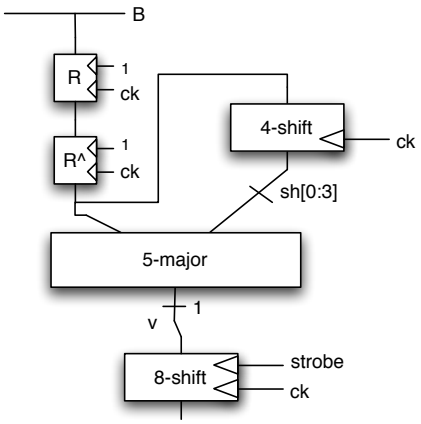


Figure 23: Hardware Construction Receiver

The automaton in figure 24 represents the decoding of a message.

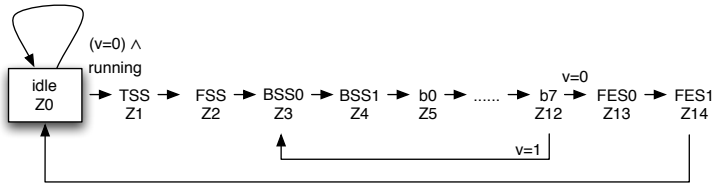


Figure 24: Decoding Automaton

6.1.2 Brainware

The final proof about correct sampling of an entire message is an inductive proof over sync intervals. The following proof shows, that the initial sync signal ($sy(0)$) is triggered at $cy(0) + [3:4]$.

Lemma 6.1 *Initial sync (at transmission start)*
Assumption:
Claim:

$$sy(0) \in cy(0) + [3:4]$$

Proof. The proof shows, that there is a falling edge between cycle 2 and 3 or between cycles 3 and 4. Hence there is a sync signal $cy(0) + [3:4]$.

$0 = R^{cy(0)+\beta+k} : k \in [0:6]$	(Lemma 4.1 and $onbus(f(m)_0, 0, \tau_s)$)
$0 = \hat{R}^{cy(0)+\beta+k+1}$	
$\beta = 0 \vee (\beta = 1 \wedge\ \hat{R}^{cy(0)+1} = 0) \implies$	(the bit is sampled correctly or there an setup- / hold-time violation occurred but the content of the $\hat{R}$ is nevertheless correct)
$v^{cy(0)+2} = 1 \wedge\ v^{cy(0)+3} = 0 \implies$	(there is 2 cycles delay by the majority voter and an extra delay due to $\hat{R}$ (Figure 23))
$sy(0) = cy(0) + 3$	

$\beta = 1 \wedge\ \hat{R}^{cy(0)+1} = 1$	(due to setup- / hold-time violation the content of $\hat{R}$ is wrong)
$v^{cy(0)+3} = 1 \wedge\ v^{cy(0)+4} = 0$	(again there is 2 cycles delay by the majority voter and an extra delay because of $\hat{R}$ (Figure 23))
$sy(0) = cy(0) + 4$	

While every bit of the original message is put on the bus 8 times, the receiver compensates glitches with the majority voter and picks a presumably good reading with the strobe signal. The following proof shows, that in the cycle range $cy(8i) + k'' : k'' \in [4:9]$ the value of the voted bits is the same as the value of the corresponding bit which was put 8 times on the bus.

Lemma 6.2 *Filtering of Bits Assumption:*

$$onbus(f(m)_i, e_s(8 \cdot i), \tau_s)$$

Claim:

$$v^{cy(8i)+k''} = f(m)_i : k'' \in [4:9]$$

Proof.

$\hat{R}^{cy(0)+\beta+k+1} = f(m)_i : k \in [0:6]$	(Lemma 4.1 and $onbus(f(m), e_s(8i), \tau_s)$)
$\hat{R}^{cy(0)+k'} = f(m)_i : k' \in [2:7]$	($\beta \in [0:1]$)
$v^{cy(8i)+k''} = f(m)_i : k'' \in [4:9]$	(2 cycles delay by the majority voter (Figure 23))

Definition 6.7 *A synchronization interval is the time between two consecutive sync signals, i.e.* $(sy(h) : sy(h+1)]$.

Definition 6.8 $sync^j$ *is used to denote, that the sync signal is turned on in cycle* j

Definition 6.9 *Analogous* $strobe^j$ *is used to denote, that the strobe signal is turned on in cycle* j*. Note that* $sync^j \implies \neg strobe^j$ *(Definition 6.5)*

In the end we want to argue about entire messages. The following proof shows, that in a range of [0:300] cycles after a sync signal the correct bits would be strobed. Later we will only need weaker statement as we know, there will be a sync signal about every 80 cycles.

Lemma 6.3 *If synchronization interval is not to long, then each bit strobed, is the correct bit.*
Assumptions:

- $sync^j = 1$, $j \in cy(8i) + [3:4]$.
- $str^t = 1$, $t = j + 8y + 4$ *(*$str^t = 1$*).*
- *The interval* $[j : t]$ *is not to long, i.e.* $t < j + 300$.
- *There is no sync in the interval* $(j : t)$*, i.e.* $sync^l = 0 : l \in (j : t)$

Claim:

$$v^t = f(m)_{i+y}$$

Proof.

$t \in$	$cy(8i) + [3:4] + 8y + 4$	(assumption)
	$= cy(8i) + 8y + [7:8]$	
	$\subseteq cy(8i + 8y) + [-1:1] + [7:8]$	(Lemma 5.1 ($cy(i+k) \in cy(i) + k + [-1:1] : k \leq 300$) and assumption about t)
	$= cy(8i + 8y) + [6:9]$	
$t \in$	$cy(8(i+y)) + k''$	
	for $k'' \in [6:9] \subset [4:9] \implies$	
$v^t =$	$f(m)_{i+y}$	(Lemma 6.2)

Lemma 6.4 *Strobing fast enough.*
Assumptions: the same as for Lemma 6.3.
Claim:

$$t + 1 < cy(8(i + y + 1)) + 3$$

Proof.

t	$\leq$	$cy(8i) + 8y + 8$	(see proof of Lemma 6.3)
		$\leq cy(8(i + y + 1)) + 1$	(Lemma 5.1 and then the next bit)
$t + 1$	$<$	$cy(8(i + y + 1)) + 3$	(again there is 2 cycles delay by the majority voter before it is voted)

Note $cy(8(i+y+1))+3$ is the earliest possible cycle, when bit $i+y+1$ can affect the voted bit v. But at that cycle the automaton is already in the next state $(t + 1)$.

As said earlier, the final proof is an inductive proof over sync intervals. The following is an important lemma for the induction start of the final proof. It states that the first regular sync signal occurs at $cy(8 * 3) + [3 : 4]$.

Lemma 6.5 *First sync at* $cy(8 * 3) + [3 : 4]$

$$sy(1) \in cy(8 * 3) + [3 : 4]$$

Proof. The proof follows the automaton starting at the initial sync.

$str(0) = sy(0) + 4$
$v^{str(0)} = 0 \implies$ (Lemma 6.3)
$TSS^{str(0)+1} = 1$

$str(1) = sy(0) + 4 + 8$
$v^{str(1)} = f(m)_1 = 1 \implies$ (Lemma 6.3)
$FSS^{str(1)+1} = 1$

$str(2) = sy(0) + 4 + 8 + 8$
$v^{str(2)} = f(m)_2 = 1 \implies$ (Lemma 6.3)
$BSS0^{str(2)+1} = 1$

$str(2) + 1 < cy(8 * 3) + 3 \implies$ (earliest possible occurrence of sync)
$sy(1) \in cy(8 * 3) + [3 : 4]$

The following is the final statement for the receiver construction. We will show, that an entire message is sampled correctly.

Lemma 6.6 *A Receiver syncs in the correct cycle range, votes the correct bit, strobes the correctly voted bit, and then steps correctly through automaton till the next sync (and start over again).*
Assumptions:

- *For all byte indices* $i \in [0 : L-1]$
- *For all bit indices* y *with* $y \in [1:10]$ *if* $i < L-1$ *and* $y \in [1:12]$ *otherwise.*

Claim:

$$sy(i+1) \in cy(8*(3+10i)) + [3:4] \tag{1.}$$

$$v^{str(2+10i+y)} = f(m)_{2+10i+y} \tag{2.}$$

$$str(2+10i+y) = sy(i+1) + 8y + 4 \tag{3.}$$

$$z^{str(2+10i+y)+1} = \begin{cases} BSS1 & : y = 1 \wedge\ i < l-1 \\ b_0 & : y = 2 \wedge\ i < l-1 \\ \dots & \\ b_7 & : y = 9 \wedge\ i < l-1 \\ BSS0 & : y = 10 \wedge\ i < l-1 \\ FES0 & : y = 10 \wedge\ i = l-1 \\ FES1 & : y = 11 \wedge\ i = l-1 \\ idle & : y = 12 \wedge\ i = l-1 \end{cases} \tag{4.}$$

Proof. By induction over sync intervals i.
induction start $i = 0$:

7 060116 Dominik Rester

Sender

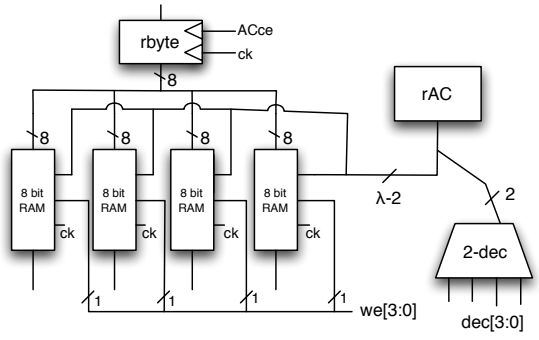
- sMB (Message Buffer)
- sB (send Buffer)
- sAC (send Address Counter)
- $sbyte$
- $scnt$ (counter always running)
- sZ (state of send automaton, clocked by $scntovfl$ every 8 cycles)
- start sending:
  - start rising edge at time $e_s(0)$
  - $\gamma \leq e_s(0) + 9\tau_s$: $onbus(f(m), \gamma, \tau_s)$

Receiver

- rMB (Message Buffer)
- rB (receive Buffer)
- rAC (receive Address Counter)
- $rbyte$
- $rcnt$ (receive counter cleared at sync edge)
- rZ (state of receive automaton, clocked by $strobe$. Usually after 8, occasionally after 7 or 9 cycles.
- start receiving if ready and a 0 is on the bus

7.0.3 Memory Interface of receiver

The message buffer is built as RAM. Like the memory of the ECU it can be accessed byte-wise but nevertheless takes a 32-bit data input. So, in order to write only a single byte of the 4 byte data input, one must signal to the RAM which of the 4 byte should be written and which not. Therefore the RAM is organized in 4 so-called banks labeled from mb_0 to mb_3 and each of them takes a separate write-enable signal. A more detailed view will be given later on.



$$we(i) = w \wedge dec[i]$$
$$w = rACce = (rZ = b_7) \wedge (rcnt = 5)$$

Figure 25: Connection of the receive buffer

7.0.4 Transmission Duration

Figure 26 shows the duration of transmitting message $m[l-1:0]$ via the flexray bus.

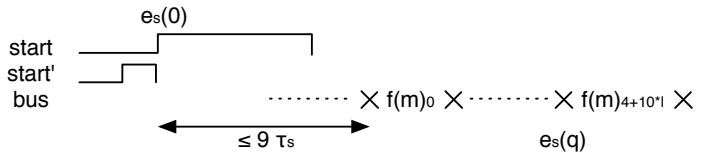


Figure 26: Transmission

It holds: $e_s(q) \leq e_s(0) + (9 + 8 \cdot (3 + 10 \cdot l)) \cdot \tau_s$ The message is sampled by the receiver into $rbyte$ not later than $cy(q) + 9 \cdot \tau_r$ After the transmission holds: $rZ^t = idle, t \leq cy(q) + 10 \cdot \tau_r \leq e_s(q) + \tau_r + 10 \cdot \tau_r$

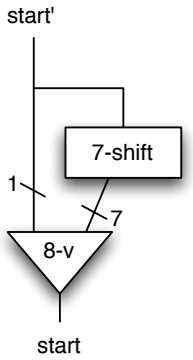


Figure 27: Computation of the $start$ signal

Lemma 7.1 *The duration of the message transfer from sender to receiver is denoted by $tl_{s,r}$ (transmission length).*

$$\begin{aligned} tl_{s,r} &\leq (9 + 8 \cdot (3 + 10 \cdot l))\tau_s + 11 \cdot \tau_r(counted\ from\ start) \\ tl_{s,r} &\leq (33 + 80 \cdot l) \cdot \tau_s + 11 \cdot \tau_r(counted\ from\ start') \end{aligned}$$

7.0.5 Computing the clock drift independantly of τ_{ref}

In General it holds:

$$|\tau_{ref} - \tau_i| \leq \tau_{ref} \cdot \delta\ ,\ \delta = 0,15\%$$

and we know:

$$|\tau_i - \tau_j| \leq |\tau_i - \tau_{ref}| + |\tau_{ref} - \tau_j| \leq 2 \cdot \delta \cdot \tau_{ref}$$

Since $|x| \geq \pm x$ we can write:

$$\tau_i - \tau_{ref} \leq |\tau_{ref} - \tau_i| \leq \tau_{ref} \cdot \delta$$

and with the same argument:

$$\tau_{ref} - \tau_i \leq \tau_{ref} \cdot \delta$$

The last two inequalities can be written as

$$\tau_i \leq \tau_{ref} \cdot (1 + \delta) \; and \; \tau_i \geq \tau_{ref} \cdot (1 - \delta)$$

Together we get:

$$\tau_{ref} \cdot (1 - \delta) \leq \tau_i \leq \tau_{ref} \cdot (1 + \delta)$$

which can also be written as:

$$\frac{\tau_i}{1 - \delta} \leq \tau_{ref} \leq \frac{\tau_i}{1 + \delta}$$

With this we replace τ_{ref} in the first inequality:

$$|\tau_i - \tau_j| \leq \frac{2 \cdot \delta}{1 + \delta} \cdot \tau_i$$

$\frac{2 \cdot \delta}{1 + \delta}$ we call Δ with $\Delta \approx 0,3005\%$

7.0.6 Flexray Schedule

Consider a flexray bus which connects p $ECUs$ where each ECU consists of a CPU and a flexray interface. (see Figure 28)

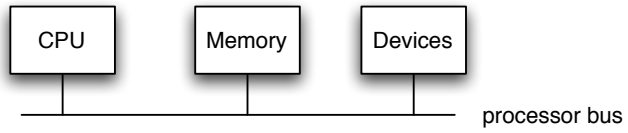


Figure 28: Internal view of an ECU

One scheduling round is depicted in figure 29 where ns denotes the number of slots in each round. The schedule is identical in each round and is determined by the function $send : [0 : ns - 1] \rightarrow [0 : p - 1]$ so that: $ECU_{send(s)}$ sends sB in slot s and all $ECUs$ (including the sending one) receive the message in rB The local schedule of an ECU_v is defined by the function $send_v[ns - 1 : 0] \in \{0, 1\}^{ns}$, $send_v(s) = 1 \Leftrightarrow send(s) = v$. The local schedule must be stored in a non volatile memory in the interfaces and loaded from there on startup.

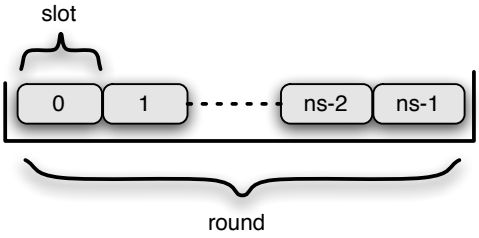


Figure 29: Schdedule overview

7.1 Flexray Interface

One ECU consists of a CPU (the $VAMP$ from [MP00]), a memory and several so-called devices. All of them are connected via the 32-bit processor bus.

The CPU can access registers of the devices by executing loadword or storeword instructions on special addresses, so-called I/O ports. The addresses of these I/O ports are not forwarded to the memory but to the corresponding device. Figure 30 shows an overview of the memory usage.

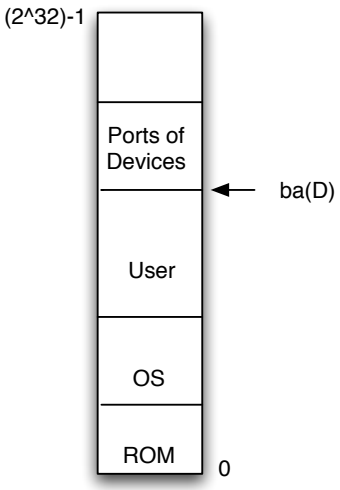


Figure 30: Memorymap

The I/O ports of a certain device D start at base address of device D: $ba(D)$. Consider a device with K many I/O ports, then a memory access with effective address $ea = ba(D) + j$ and $j \in [0 : K - 1]$ would result in an access to port j of device D (with our CVM microkernel this will only work in system mode, because no user addresses will be mapped to I/O ports).

7.1.1 Internal Flexray State

A flexray device has some internal registers, e.g.

- $f.ready$ which is initialized with 0 after reset
- F = $f.timer$ which is divided into ρ higher bits and σ lower bits. The higher bits are called $f.timer.slot$ or $F.slot$ and the lower bits are called $f.timer.cycle$ or $F.cycle$. The bidwiths are computed in the following way: $\rho = \lceil \log(ns + 1) \rceil$ and $\sigma = \lceil \log T \rceil$ where ns denotes the number of slots and devices will run T cycles between timer synchronizations, moreover timers are synchronized at start of rounds.

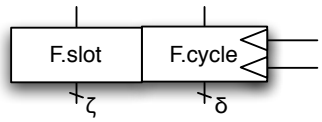


Figure 31: F.timer

7.1.2 Flexray I/O Ports

The flexray device has the following ports:

- status
- command: if d is a valid command, then writing data d to the command port of device D leads to the device executing command d, e.g. the last command of the init sequence which sets $ready = 1$ would be $setrd$.
- 2 send buffers: $sB_0, sB_1 \leftarrow$ occupy the same ports
- 2 receive buffers: $sB_0, sB_1 \leftarrow$ occupy the same ports

We have send and receive buffers in duplicate and switch dynamically the connections between the buffers and the processor/flexray bus. We call $p = F.slot[0]$ parity bit and use p to select which buffer is currently connected to the processor and which one is connected to the flexray bus.

- visible for flexray bus: sB_p, rB_p
- visible for processor: $sB_{\overline{p}}$, $rB_{\overline{p}}$

Register synonym	Description
command	command to be executed
ms	number of slots per round ($\leq ns - 1$)
l	number of bytes in the message ($\leq l_{max}$)
T	local number of cycles in slot
off	offset before start of the transmission (max. clock drift in a round)
$IWAIT$	time after command $setrd$ (set ready) until the start of the first transmission
$sendl_v[63:32]$, $sendl_v[31:0]$	local schedule

Table 1: Port configuration registers

8 060118 Sergey Tverdyshev

The local schedule $sendl_v$ is defined in terms of the scheduling function $send$: $sendl_v(s) = 1 \Leftrightarrow send(s) = v$.

Figure 30 depicts location of send buffers, receiver buffers and configuration ports in the memory map of an ECU. Let K be upper bound of a buffer size, e.g. send buffer size, receiver buffer size, number of configuration ports.[2]. The base address of the ports of a device D is computed by the function $ba : D \rightarrow \{0,1\}^{32}$. The result of $ba(D)$ has to be multiple of K. Every configuration port consists of several configuration registers (Table 1). All these registers are written during the initial phase.

8.0.3 Design of ports hardware

The first hardware construction we need is a (edge triggered) $K \times d$ - RAM.

$K \times d$ - RAM

Let $R : \{0,1\}^k \rightarrow \{0,1\}^d$, with $K = 2^k$ be a $K \times d$ - RAM. The result of read operation at cycle t, denoted as $Dout^t = R^t(ad^t)$, is the value stored in R on the address ad. We have the read data at the same cycle because we are defining a register based RAM. In case of a write access the next value of R^t is specified as follows:

$$R^{t+1}(a) = \begin{cases} Din^t & a = adr^t \wedge w^t = 1 \\ R^t(a) & else \end{cases}$$

Figure 32 presents how a RAM could be built. It usually consists of two parts: decoding of the input address (Figure 32 (a)) and implementation of write/read accesses. Figure 32 (b) shows how write/read access could be realized, where R is

[2]For our example K is 10

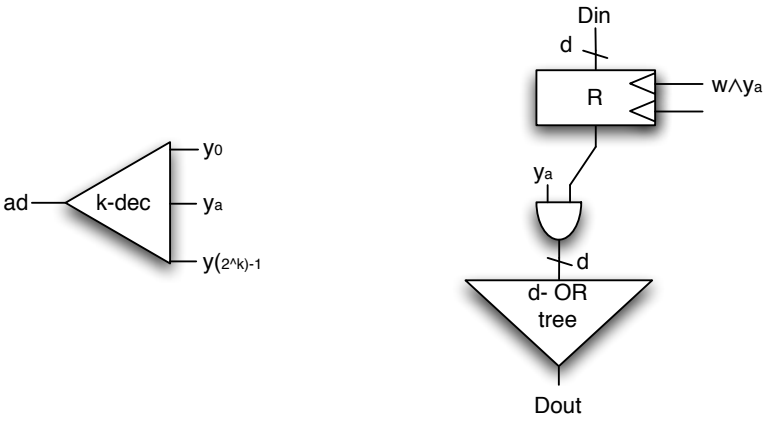


Figure 32: RAM construction: (a) address decoding, (b) read and write accesses to a register in RAM

a register in the RAM. It is important to note that if we write to and read from the very same address at the same cycle we will get the "old" value stored in the RAM.

$K \times d$ - Port RAM

A $K \times d$ - port RAM is based on $K \times d$ - RAM (Figure 33). The port RAM has

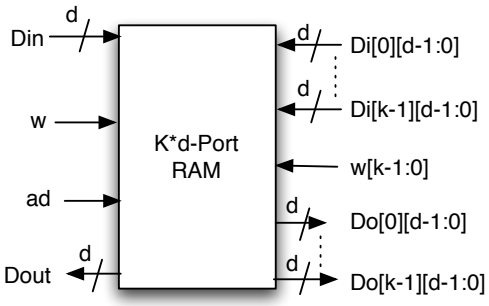


Figure 33: Port RAM

two interfaces: processor side (on the left hand side) and devices side (on the right hand side). The behavior of a port RAM is similar to $K \times d$ - RAM. The semantic of read operation is exactly the same, i.e. $Dout^t = R^t(ad^t)$ and $Do[i]^t = R^t(ad^t)$. The next value of R is defined as follows:

$$R^{t+1}(a) = \begin{cases} Di^t[a] & w^t[a] = 1 \\ Din^t & ad^t = a \wedge w^t[a] = 0 \wedge w^t \\ R^t(a) & else \end{cases}$$

It is important to note that in our definition the write signal on the devices side has greater priority than the write signal on the processor side.

8.0.4 Send Buffer

The core of a send buffer consists of two $x/4 \times 32$-RAMs, with $x = 2^{\lambda-2}$. A design of a send buffer is presented on Figure 34. On this figure Din_p is the data from the processor side. $dad[\lambda+1,0] \in \{0,1\}^{\lambda+2}$ is the device byte address and $dad[\lambda+1,2] \in \{0,1\}^{\lambda}$ is a device word address. The computation of read and write addresses is presented on Figure 35 and is controlled by parity bit p.

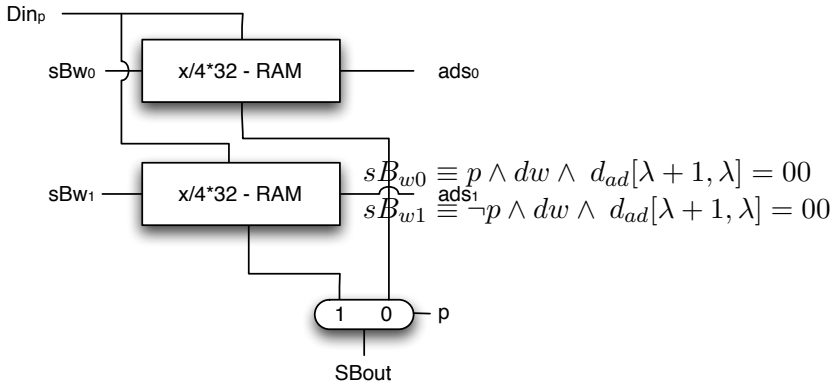


Figure 34: Send buffer

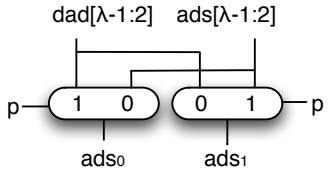


Figure 35: Address selection

In case of a write access the device write signal dw is computed as follows:

$$dw = wp\ \wedge \langle ads[31:K]0^K \rangle = ba(D)$$

, where wp id the write signal from the processor side. Computation of the write signals for every RAM (sB_{w0}, sB_{w1}) is presented on the Figure 34.

8.0.5 Receiver Buffer

The core of a receiver buffer (see Figure 36) consists of two sets of four RAM-banks each. We use four banks in order to implement byte-read/write accesses.

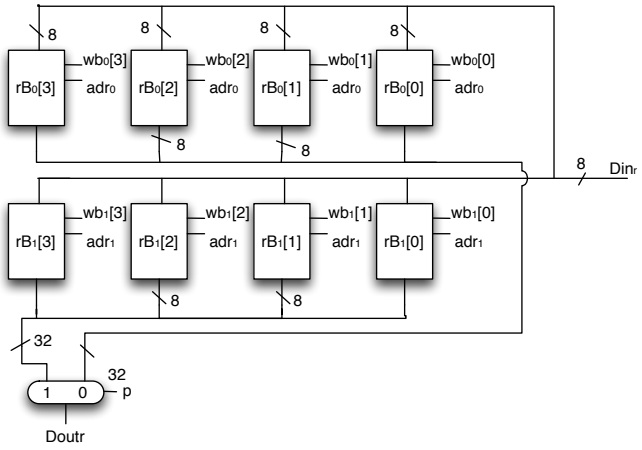


Figure 36: Receiver buffer

The signal Din_r represents data from the receiver side and $Dout_r$ data on the processor side. The address from the receiver side (adr) is taken from the receiver address counter (rAC). The final selection between dad and adr is done by multiplexer which is controlled by the parity (p) signal (see Figure 36).

Let $w_i \in \{0,1\}^4$ be unary representation of the last two bits of adr_i, where $i \in \{0,1\}$ (Figure 37). Then the write signal for bank $b \in \{0,1,2,3\}$ in rB_0 is computed as: $wb_0[b] = w_0[b] \wedge \neg p \wedge wr$, with wr write signal from the receiver side. Analogous, we compute write signals for rB_1: $wb_1[b] = w_1[b] \wedge p \wedge wr$.

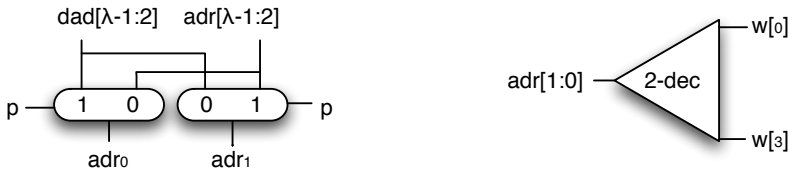
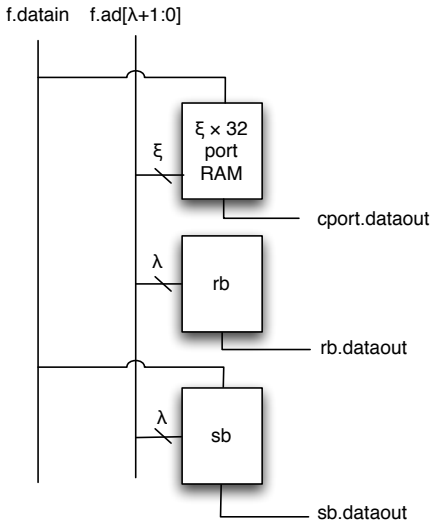


Figure 37: (a) Receiver address selection and (b) Write bit decoding

9 060123 Sergey Tverdyshev

9.0.6 Putting it all together

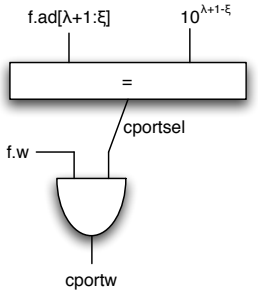
On the Figure 38 the data paths of ports hardware are presented. All addresses are computed based on $f.ad$, namely:



$$rb.ad \equiv f.ad[\lambda + 1 : 2]$$
$$sb.ad \equiv f.ad[\lambda + 1 : 2]$$
$$cport.ad \equiv f.ad[\xi - 1 : 2]$$

Figure 38: Data paths of ports hardware **[TODO:** Stefen: signals names**]**

The output data of on the device bus are selected between data from control ports and data from receiver buffer.The selection bit $cport.sel$ is set up if and only if the device address $f.ad$ lies above receiver buffers, i.e. some there in the ports memory range.



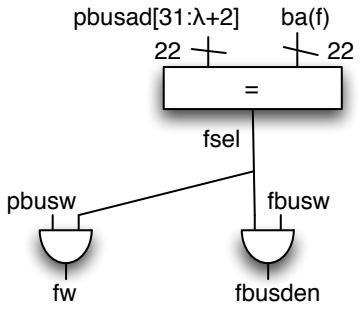
$$cport.sel \equiv f.ad[\lambda + 1 : \xi] = 10^{\lambda+1-\xi}$$
$$cport.w \equiv cport.sel \wedge f.w = 1$$

Figure 39: Data paths of ports hardware: output selection

The write signal for control ports $cport.w$ is set up if there is an access to the ports memory range and we have write access. The last signal we have to define is $sb.w$. This signal is computed as follows $sb.w = f.w \wedge f.ad[\lambda + 1, \lambda] = 00$.

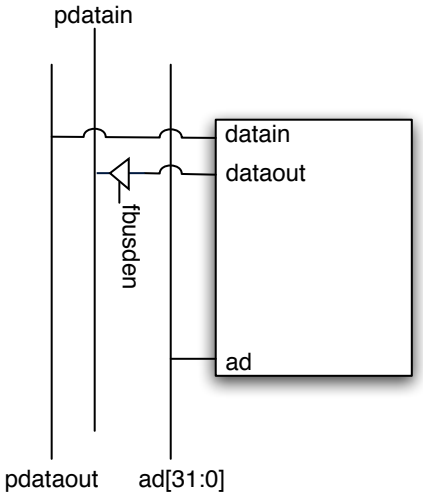
9.1 Connection a Device with FlexRay Interface to the Processor Bus

Figure 41 depicts how a device with the FlexRay-like interface can be connected with the processor bus. The processor output is connected directly to the device input, that is $pbus.dataout = f.datain$. The input data for the processor are taken from output of the device interface. However, since several devices can be connected to the $pbus.datain$, the data from the device interface are controlled by a driver (see Figure 41). The driver enable signal is computed as follows: first we have to check whether the address on the processor bus $pbus.ad[31:K]$ is the device address, i.e. compute $f.sel$. With this flag we can easily compute the control signal for the driver as well as the write signal for the device (Figure 40).



$$pbus.dataout \equiv f.datain$$
$$f.sel \equiv (pbus.ad[31:K] = ba(f))$$
$$f.w \equiv pbus.w \wedge f.sel$$
$$fpbusden \equiv pbus.r \wedge f.sel$$

Figure 40: Driver control signal



$$pbus.dataout = f.datain$$

Figure 41: Processor Bus with FlexRay Bus

Address of configuration register	Description
0	command to be executed
4	status
8	interrupt
12	$sendl_v[0]$
16	$sendl_v[1]$
λ	$IWAIT$

Table 2: Addresses of configuration registers

9.2 Semantic of Configuration Registers

The Table 2 presents several addresses of configuration registers.

The Figure 42 depicts how for a particular command the signal "command has to be executed" is computed. For example the reader can see that the command $setrd$ has number 0.

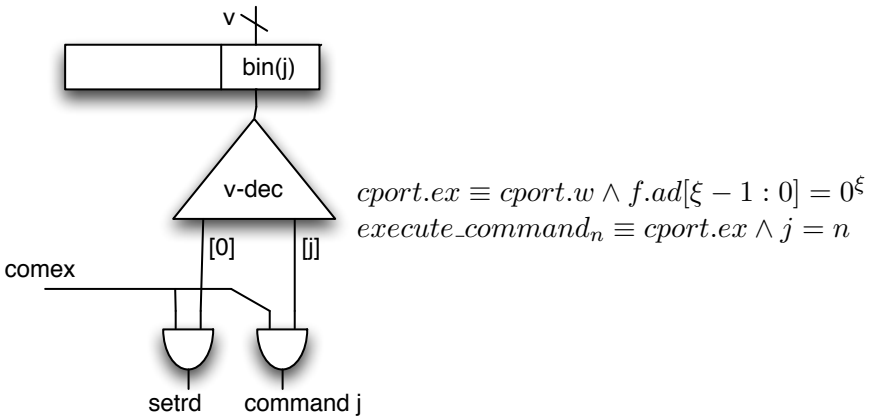


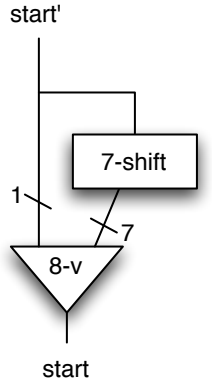
Figure 42: Computation of the command number

9.3 Construction of Starts Signals

The Figure 43 shows how the signal $start'$ could be computed based on the signal $start$.

1. the local scheduler registers $sendl_v[i]$ has to be set
2. the command register is set to $setrd$

An interesting issue is *when* all processors are initialized i.e. ready to operate. In order to estimate when all processors are ready we will use worst case execution time (WCET) analysis. Assume the function $WCET(prog)$ returns the number



$$start^t \equiv start'^t \vee \exists ts \in [t-7, t) : start'^{ts}$$

Figure 43: Computation of the start signal

of *processor cycles* which a processor needs to execute the program $prog$. Let $treset_v \in setR$ be the time of the falling edge of the $reset$ signal on the processor v. Now we can compute the maximum delay after the reset signal when all processor are running:

$$R = max_{v,u}\{treset_v - treset_u\} * \tau_{ref}$$

The configuration register $IWAIT$ (see Table 1) is initialized with the following value:

$$f^{init}.IWAIT = \lceil WCET(init_{prog}) * (1 + \Delta) + R * (1 + \delta) \rceil$$

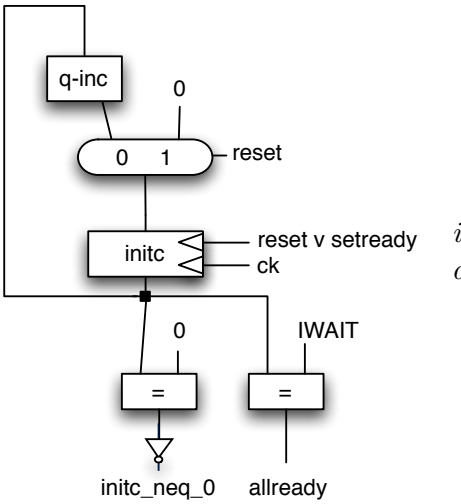
Lemma 9.1 *: Let t be the earliest falling edge of any reset signal: $t = min_v\{treset_v = 1\}$ and $t' = t+(IWAIT+1)*\tau_{min}$. We claim that all processors at time t' are running:*

$$\forall v : \quad f.ready_v(t') = 1$$

Definition 9.1 *Let R be a register or a hardware signal. Then the value of R at time t in ECU v is defined as follows:*

$R_v(t) = R_v^i$ *, with* $t \in [e_v(i), e_v(i+1))$

The waiting process of $IWAIT$ cycles is done by an initial counter. The counter is initialized with 0 and counts up to $IWAIT$. Let Q be the upper bound for $IWAIT$ then we need a $q = \lceil logQ \rceil$ - bits counter to implement that waiting process. The Figure 44 illustrates how such a hardware can be built. An interesting signal is all_ready. This signals is set up when the counter reached $IWAIT$ that should imply that all processors are up and running. This is captured by the following lemma:



$initc^t_{out} \equiv initc \neq 0^q$
$allready^t \equiv initc = IWAIT$

Figure 44: Implementation of Waiting Process of $IWAIT$ Cycles **[TODO:** Stefen: initc double declaration!!!. By wires crossing use dots**]**

Lemma 9.2 *: Let t' be the first cycle such that $\forall v. \quad t' > min_v\{treset_v\} \wedge all\_ready_v(t') = 1$ then*

$$\forall u. \;\; f.ready_u(t') = 1$$

10 060125 Eyad Alkassar Introduction

In Section [x] we described how the interface hardware of a sender and a receiver ECU could be designed for correctly sending and receiving messages via a bus. The correctness statements of this design were made under the assumption that only the sender was putting data on the bus.
So far we had not said anything about the communication protocol, i.e. defining *when* a certain ECU is allowed to send and when not. For that, we introduced in the last Section a global scheduling regime.
This regime is made of rounds and slots. In each round there is some constant number ns of slots. Each slot is dedicated to exactly one ECU, in which it is allowed to transmit data on the bus. A slot itself lasts a fixed count of cycles.
Since we are dealing with a distributed architecture each ECU keeps track of the current cycle and slot number. But a possible and uncorrected drift in the clocks, could lead to different local notions of the current slot number in each ECU, and with that to the simultaneous sending of many ECUs (see Figure 10).
Hence, we introduced some global synchronization algorithm on each ECU, and shrank the sending interval in each slot by some *offset* value from the right and from the left (see Figure 10).
The most important result we got in the last Section, was to show how maximum period of time initializing the scheduling algorithm on each ECU could be bounded. Hence, we specified a signal $allready_v$ and showed that whenever it turns one, all ECUs are in state ready:

$$allready_v(t) \Longrightarrow \forall u : ready_u(t)$$

In this Section we will

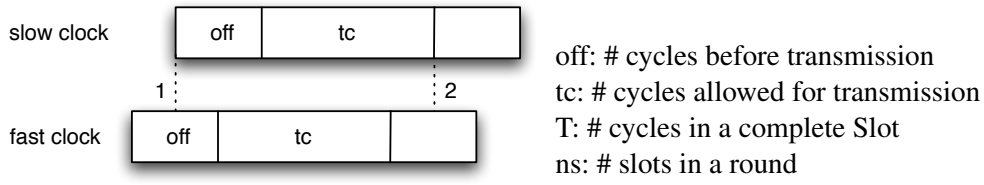


Figure 45: Local notion of a slot on two different ECUs

- first identify what **correctness property** we would like to prove.
- Second we **design the hardware** that is required for implementing the synchronization algorithm.
- Third we **prove** that this hardware implements a correct synchronization algorithm.

In the following we will denote with $R_v(t)$ the content of hardware Register R in ECU v at time t, i.e. $R_v(t) = R_v^i$ with $t \in [e_v(i) : e_v(i+1)]$. Small letters u and v are used for denoting numbers of ECUs.
Further we will w.log. denote with ECU_0 the ECU which sends at slot 0, i.e. $sendl_0[0] = 1$.

11 What do we want to prove?

Intuitively our correctness property should ensure that whenever a ECU *thinks* that it is allowed to start sending at some slot n, the slot counter of all other ECUs also hold the value n. I.e. neither will an ECU send before all others entered the same slot, nor after some ECU left it (see Figure 10, case 1 and 2).
For defining this formally, we introduce the following names, denoting when a slot and when transmission starts:

- $\alpha_v(r, s)$: start time of slot s of round r on ECU_v.
- $ts(r, s)$: time in round r and slot s in which transmission starts.
- $te(r, s)$: time in round r and slot s in which transmission ends.

With that we can state the gurantees that our synchronisation algorithm must satisfy.
Correctness of Scheduling Algorithm:

- When transmission in slot s of round r starts, then all other ECUs have locally also started the same slot:

$$\forall v : ts(r, s) > \alpha_v(r, s)$$

- When an ECU have locally started slot $s + 1$ of a round r, than transmission of slot s has already ended:

$$\forall v : te(r, s) < \alpha_v(r, s + 1)$$

We will denote our correctness property in the following with *SyncCorr(r,s)*.

12 The synchronization algorithm and its hardware implementation

The idea of our distributed synchronization algorithm is simple. Each ECU locally counts cycles (i.e. clock ticks) and slots in some counter called F_u. The slot counter $F_u.slot$ is increased whenever the cycle counter $F_u.cycle$ reaches a fixed number T of cycles per slot. Then $F_u.cycle$ is reset to zero. If in a slot an ECU is allowed to send, it will wait with it transmission until the cycle counter reaches the value

offset.
Synchronization of the counter takes place at the beginning of a new round. When an ECU reaches the end of a round, it waits until receiving the first sync edge (see Section []) on the bus, and then it resets its slot counter to zero and its cycle counter to *offset*.
This first sync edge is generated through the ECU_0, i.e. the ECU that is allowed to send in the first slot. Hence this ECU follows a slightly different algorithm than the other nodes.

Semantics of hardware counter F (FlexRay counter) The circuit has the two input signals $Fsync$ and $Fmax$, and the data input $Foffset$ (see Figure 12). We

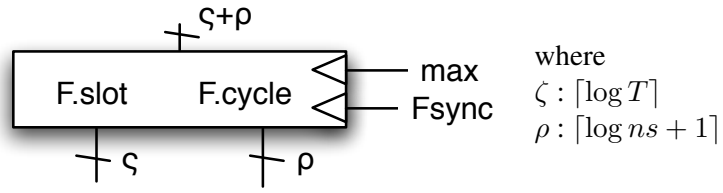


Figure 46: Schematics of Hardware Counter F (**F**lexRay)

interpret the input and output bit string to/of the circuit F as:

$$\mid F \mid = \langle F.slot \langle \cdot T \rangle + F.cycle \rangle$$

With that we can specify the semantics of the circuit F of Figure 12: F increases its value (i.e. $\mid F \mid$) when neither $Fsync$ nor $Fmax$ is one, until reaching the value $FMAX = ns * T$, denoting the count of cycles in a round. The signal $Fsync$ indicates a beginning of a new round. Therefore when it is one the value of the counter is set to input $Foffset$. When the signal $Fmax$ is one, the counter is set to the value FMAX. When reaching $FMAX$, the counter gets stuck until a new round is started. Formally the above is described through:

$$\mid F_u \mid = \begin{cases} \text{Foffset}_u^t & : Fsync^t \\ \mid F_u \mid & : \neg Fsync^t \wedge \neg Fmax^t \wedge \mid F \mid < FMAX \\ FMAX & : else \end{cases}$$

Hardware implementation The hardware implementation of the above specified counter is depicted below:

Defining signals In the following we will define the signals $Fsync_u$, max and $start'$.

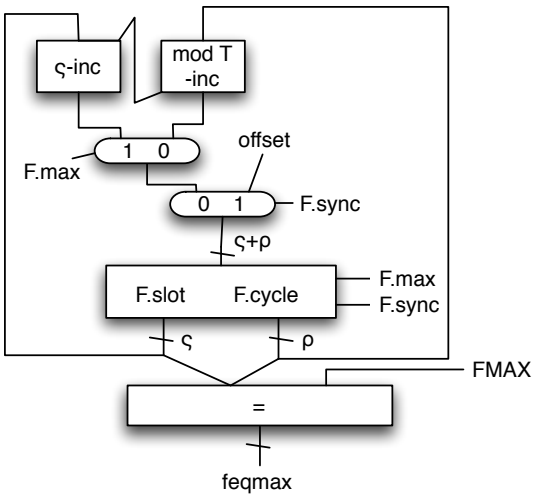


Figure 47: Harware impllementation of Slot and Cycle Counter

- **End of a round** We said $Fsync_u$ should denote the start of a new round on ECU_u. On an ECU a new round starts whenever its slot counter reached its maximum ns, it is in the idle state and it receives a sync edge on the bus, indicating that the first ECU started transmitting. Then the ECU resets its cycle counter to the offset value.
  We see, that the $Fsync_u$ signal of ECU_0 must be designed in a different way, because it is the one telling all the other ECUs through the sync signal that a new round started. In the case of ECU_0 the signal $Fsync_u$ is one, after initialization when ECU_0 can be sure that all other processes are ready to receive the sync edge or when F reaches FMAX-1. Formally we get:

$$\begin{aligned} FsyncF_v^t = \quad & (sendl[0] \wedge (F = FMAX - 1 \vee allready_v^t)) \\ & \vee (\neg sendl[0] \wedge F_{slot} = ns \wedge rZ = idle \wedge sync^t) \end{aligned}$$

- **Reseting** The Fmax signal is set to one when resetting:

$$Fmax_u^t = reset_u^t$$

- **Start of transmission** Now we can define the start' signal of the FlexRay hardware (see Section []), i.e. the signal indicating when the FlexRay interface should but its data on the bus. As described before an ECU starts transmitting when it reaches the slot dedicated to it and the cycle counter has the value *offset*:

$$start' = sendl[F.slot] \wedge F.cycle = \text{off}$$

  The circuit computing start' is implemented as depicted in Figure 48.

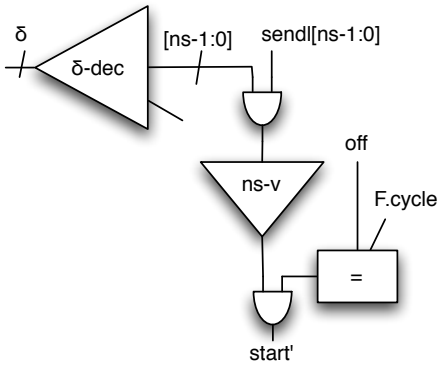


Figure 48: Computing start'

After having specified the counter, we know can give the concrete definitions of slot and round starting times for the ECUs (see informal definitions in 11). Assuming that at time t the signal $allready$ of ECU_0 is one. Then we define $\alpha_u(r,s)$ as follows.

- **For ECU_0:**

$$\begin{aligned}\alpha_0(0,0) &= t + off * \tau_0\\ \alpha_0(0,1) &= \alpha_0(0,0) + (T - off) * \tau_0\\ \alpha_0(r,s) &= \alpha_0(0,0) + ((r * ns + s) * t - off) * \tau_0\end{aligned}$$

- **For ECU_u $\mathbf{u \neq 0}$:** We define a new round r of an ECU, as the first time after the start of the round on ECU_0 when the local counter was reset (i.e. a sync edge was received).

$$\begin{aligned}\alpha_u(r,0) = \quad & \min\{t \mid t > \alpha_0(r,0)\\ & \wedge F_u(t - \tau_v).cycles = \text{off}\\ & \wedge F_u(t - \tau_v).slot = ns\}\\ \alpha_u(r,s) = \quad & \alpha_u(r,0) + (s * T - off) * \tau_u\end{aligned}$$

13 Proving correctness of the sync Algorithm

The idea of the proof is simple. We only have to choose the offset value *off* greater than the sum of

- the maximum clock drift after synchronization at the beginning of a round.
- and the maximum difference of receiving times of the sync edge at the beginning of a round.

then an easy induction over the round count leads to our claim *SyncCorr(r,s)*.

13.1 Time interval receiving the sync-edge

The time interval in which the sync edge sent by ECU_0 at the beginning of the new round r is received by some ECU_u could be determined by the following timing diagram. The diagram starts at the beginning of round r on ECU_0 and ends when the sync edge is received by some ECU_v. With the help of the depicted timing

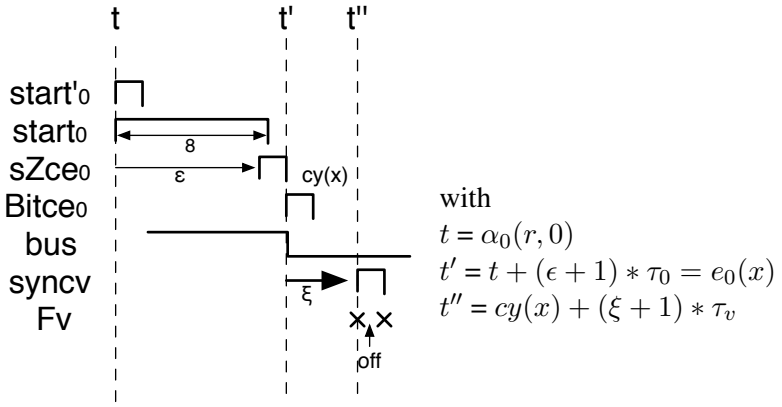


Figure 49: Timing diagram at the beginning of a round

diagram we can estimate $\alpha_v(r,0)$ in the following way:

$t' \leq t + 8 * \tau_0$	see Diagram
$t'' \leq t' + \tau_v + 5 * \tau_v$	the first $\tau_v$is due to possible sampling errors at receiver edge cy(x).The next $5 * \tau_v$result from computation delay and possible bit syncing errors
$\Longrightarrow \alpha_v(r,0) \leq t + 8 * \tau_0 + 6 * \tau_v$	
$\leq t + 15 * \tau_y$ for all y	

We also can estimate:

$t' \geq t + \tau_0$	see Diagram
$t'' \geq t + 4\tau_v \geq t' + 3\tau_y$ for all y	

In the stated diagram we implicitly assume that only ECU_0 is sending and all other ones are quiet, i.e. have locally reached the end of the round and their output registers are set to 1. Formally we catch this assumption through[3]:
$H(r) \equiv$ for all u and for all round starting times $t = \alpha_0(r,0)$:

$$
\begin{aligned}
& ready_u(t) \\
& \wedge\, R_v(t) = \widehat{R}_v(t) = sh[i]_v(t) = 1 \\
& \wedge\, rZ_v(t) = idle \\
& \wedge\ \mid F_v(t) \mid = ns * T
\end{aligned}
$$

[3]Die Annahme ist staerker als noetig

From the above analysis our we can finally define the searched time interval as follows.

Lemma 11: Local starting of new rounds All ECUs enter the new round r in a bounded time interval:

$$H(r) \Longrightarrow \forall u, v : \alpha_0(r,0) + 3 * \tau_u \leq \alpha_v(r,0) \leq \alpha_0(r,0) + 15 * \tau_u$$

060130 Abdul Qadar Kara Correctness

There is some correctness done in the previous definitions of $\alpha_u(r,s)$

Assuming that at time t the signal $allready$ of ECU_0 is one. Then we define $\alpha_u(r,s)$ as follows.

- **For ECU_0:**

$$\begin{array}{ll} \alpha_0(0,0) & = t + off * \tau_0 \\ \alpha_v(r,s) & = \alpha_v(r,0) + ((s*T) - off) * \tau_v (1 \leq s \leq ns \text{ length of round in local counter}) \\ \alpha_0(r,0) & = \alpha_0(r-1,ns) + off * \tau_0 \end{array}$$

Lemma 11

$$\forall_x . H(r) \rightarrow \alpha_0(r,s) + 3 * \tau_x \leq \alpha_v(r,0) \leq \alpha_0(r,0) + 15 * \tau_x$$

using:

$$\begin{array}{ll} \alpha_v(r,0) & = min\{t | t > \alpha_0(r,0) \wedge timer_v(t) = off \wedge timer_v(t - \tau_v) - ns\} \\ \forall_v . \alpha_v(r,s) & = \alpha_v(r,0) + ((s*T) - off) * \tau_v \end{array}$$

Transmission Start times (ts)

$$ts(r,s) \quad = \alpha_{send(s)}(r,s) + (off * \tau_{send(s)})$$

Now, we need to find the upper bound for transmission end time.
By **Lemma 4.9**,

$$\begin{array}{ll} tl_{s,r} & \leq ((33 + (80 * l)) * \tau_s) + (11 * \tau_r) \\ & \leq (45 + 80) * \tau_s \end{array}$$

Here ,transmission cycles, $tc = (45 + 80) * \tau_s$

Transmission End Times (te)

$$\begin{array}{ll} te(r,s) & = ts(r,s) + (tc * \tau_{send(s)}) \\ & = \alpha_{send(s)}(r,s) + ((off + tc) * \tau_{send(s)}) \\ \forall_v . & \\ & ts(r,s) \geq \alpha_v(r,s) \\ & te(r,s) \leq \alpha_v(r,s+1) \\ & \text{new round}(H(r+1)) \end{array}$$

Value of off

$\forall_{u,v,x}.$
$|\alpha_u(r,0) - \alpha_v(r,0)| \leq (15 * \tau_x)$
$s \geq 1$
$\alpha_u(r,s) = \alpha_u(r,0) + ((((s*T) - off)) * \tau_u)$
$\alpha_v(r,s) = \alpha_v(r,0) + ((((s*T) - off)) * \tau_v)$
$\alpha_u(r,s) - \alpha_v(r,s) \leq |\alpha_u(r,s) - \alpha_v(r,s)|$
$\leq (15 * \tau_x) + (((s*T) - off) * |\tau_u - \tau_v|)$
Instantiate x by v,
$\leq (15 * \tau_v) + (ns * T * \Delta * \tau_v)$
$= \tau_v(15 + (ns * T * \Delta))$
$= \tau_v * off$
$off = 15 + (ns * T * \Delta)$

Also,

$$\begin{aligned} \alpha_u(r,s) &\leq \alpha_v(r,s) + (off * \tau_v) \\ &= ts(r,s) \text{ if } v = send(s) \end{aligned}$$

Lemma 12

$$\begin{aligned} &\forall_u. \\ &H(r) \rightarrow \quad \alpha_u(r,s) \quad = ts(r,s) \end{aligned}$$

To prove:

$$\begin{aligned} &\forall_u \\ te(r,s) &= ts(r,s) + (tc * \tau_v)\ v = send(s) \\ &= \alpha_v(r,s) + ((off + tc) * \tau_v) \\ &> \alpha_u(r, s+1) \end{aligned}$$

Value of T

$$\begin{aligned} te(r,s) &= \alpha_v(r,s) + ((\tau_v * (off + tc))) \\ &\leq \alpha_v(r,s) + ((1+\Delta) * \tau_u * (off + tc)) \\ &\leq \alpha_u(r,s) + (off * \tau_u) + ((1+\Delta) * \tau_u * (off + tc)) \\ &= \alpha_u(r,s) + (off * (2+\Delta) * \tau_u) + ((1+\Delta) * tc * \tau_u) \\ &= \alpha_u(r,s) + (\tau_u * (off(2+\Delta) + tc(1+\Delta)) \\ &= \alpha_u(r,s) + (\tau_u * T) \\ &= \alpha_u(r, s+1) \end{aligned}$$

with

$$\begin{aligned} T = &\quad tc(1+\Delta) + ((15 + (ns * T * \Delta)) * (2+\Delta)) \\ &\quad 30 + (15 * \Delta) + ((2+\Delta) * ns * T * \Delta) + ((1+\Delta) * tc) \\ T = &\quad \frac{((1+\Delta)*tc)+30+(15*\Delta)}{1-((2+\Delta)*ns*\Delta)} \end{aligned}$$

Lemma 13

$$\forall_u. \quad H(r) \rightarrow te(r,s) \leq \alpha_u(r,s+1)$$

Lemma 14

$\forall_r.$

1. $H(r)$
2. $\forall_u.\alpha_u(r,s) \leq ts(r,s)$
3. $\forall_u.te(r,s) \leq \alpha_u(r,s+1)$

Proof

1. follows from Lemma 9
2. follows from Lemma 12
3. follows from Lemma 13

Corollary

$$\begin{array}{lcl} H(r+1) & & \\ te(r,ns-1) & \leq & \alpha_0(r,ns) \text{ From Lemma 14 on 3} \\ & < & \alpha_0(r,ns) + (off * \tau_0) \\ & = & \alpha_0(r+1,0) \\ & = & t \text{ From Lemma 11 round } r+1 \end{array}$$

Theorem

$\forall_{r,s} \quad p = s \; mod \; 2$

$$\forall_v.rb_{p,v}(\alpha(r,s+1)) = \begin{array}{ll} sb_{p,u}(\alpha(r,s)) & : \; s \neq 0 \\ sb_{p,u}(\alpha(r-1,ns)) & : \; s = 0 \end{array}$$

Processor AS ECU is comprised of processor and flexray. We have already introduced and verified the example of a flexray protocol, what remains is introduction of processor.

Following, we will just define a normal DLX machine and then later on, we will combine with the flexray bus. More details on the processor can be found in our book [4].

DLX machine The DLX configuration d has the following components:

$$\begin{array}{ll} d.gpr & 0,1^5 \rightarrow 0,1^{32} \\ d.spr & S \subseteq 0,1^5 \rightarrow 0,1^{32} \\ d.m & A \subseteq 0,1^{32} \rightarrow 0,1^8 \\ d.pc & \in 0,1^{32} \\ d.dpc & \in 0,1^{32} \end{array}$$

[4] Computer Architecture, Complexity and Correctness,M¸ller, S.M. and Paul, W.J.,Springer Verlag, 2000

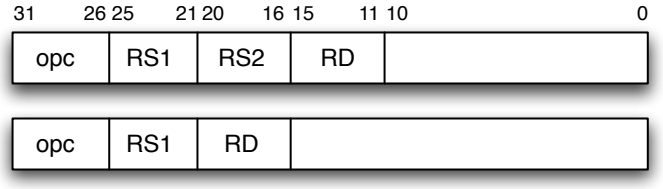


Figure 50: Instruction Types

Notation

$$m_x(a) = m(a + x - 1) \circ ... \circ m(a)$$

$$m_4(a) = \text{ memory word starting at byte address} a$$

Instruction Register

$$I(d) = d.m_4(d.dpc)$$

Opcode Opcode specifies how to interpret the remaining bit string (after opcode, $I(d)[25 : 0]$) as well as what operation to perform on that bit string. Its part of Instruction Register.

$$opc(d) = I(d)[31 : 26]$$

Instruction Formats There are three formats of instruction used in DLX machine. I-type, R-type and J-type.

For $X \in \{I, J, R\}$,

$$X - type(d) \leftrightarrow \quad opc(d) \in \{ \text{ Opcodes of all the instructions which are X-type } \}$$

$$\begin{array}{lll} RS1(d) = & I(d)[25:21] & \\ RD(d) = & I(d)[20:16]: & \text{I-type}(d) \\ & I(d)[15:11]: & \text{R-type}(d) \end{array}$$

Load Word Instruction(lw) Load Word(lw) instruction is an I-type instruction. It gets the memory contents from the effective address(ea) comprising of the contents of general purpose register $RS1$ and the immediate constant(imm)in the instruction $I(d)[15 : 0]$ and loads it into general purpose register RD. Effective address is evaluated as:

$$[ea(d)] = [d.gpr(RS1(d))] + [imm(d)] mod 2^{32}$$

As immediate constant is 16 bit constant, we use the modulo of 2^{32}, $ea(d)$ and $d.gpr(RS1(d))$ are 32 bit long.

Semantics of Load Word Instruction We denote the configuration of the DLX machine(d) by $\delta(d)$.Semantics of the load word(lw) instruction are:
Let,

$$\begin{array}{lll} \delta(d) = & d' & \\ d'.gpr(x) = & d.m_4(ea(d)): & x = RD(d) \\ & d.gpr(x): & x \neq RD(d) \end{array}$$

Ofcourse, there are some other changes in configuration d' like, Let,

$$\begin{array}{ll} d'.pc = & d.pc + 4 \\ d'.dpc = & d.pc \end{array}$$

One more requirement is that $ea(d)$ should be a valid memory address, i.e. $ea(d) \in A$.

Store Word Instruction(sw) Store Word(sw) instruction is an I-type instruction. It stores the contents of general purpose register RD to the memory at effective address(ea) comprising of the contents of general purpose register $RS1$ and the immediate constant(imm)in the instruction $I(d)[15 : 0]$. Effective address is evaluated as:

$$[ea(d)] = [d.gpr(RS1(d))] + [imm(d)] mod 2^{32}$$

As immediate constant is 16 bit constant, we use the modulo of 2^{32}, $ea(d)$ and $d.gpr(RS1(d))$ are 32 bit long.

Semantics of Store Word Instruction We denote the configuration of the DLX machine(d) by $\delta(d)$.Semantics of the store word(sw) instruction are:
Let,

$$\begin{array}{lll} \delta(d) = & d' & \\ d'.m_4(ea(d)) = & d.gpr(RD(d)) & \\ d'.m(x) = & d.m(x) & \langle x \rangle mod 2^{32} \notin \{\langle ea(d) \rangle mod 2^{32}, ..., \langle ea(d) + 3 \rangle 2^{32}\} \end{array}$$

The other changes in configuration d' are, Let,

$$\begin{array}{ll} d'.pc = & d.pc + 4 \\ d'.dpc = & d.pc \end{array}$$

One more requirement is that $ea(d)$ should be a valid memory address, i.e. $ea(d) \in A$.

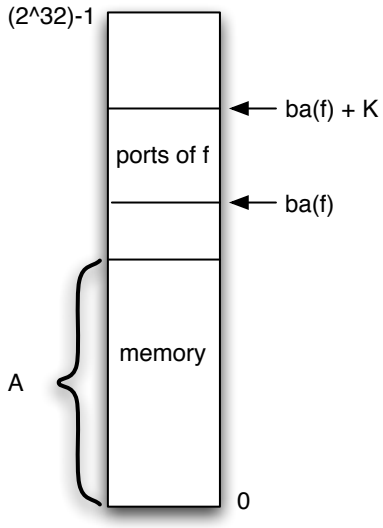


Figure 51: The memory map

14 060201 Matthias Daum Integrating the Flexray Interface Device

In the last lecture, we repeated how the processor works in isolation. Today, we will integrate the FlexRay interface as a device in our model. At first, we have to acknowledge, the different state transition levels, as summarized in Table 3.

DLX processor d	**FlexRay interface** f
Transition function $\delta_{\text{DLX}}(d) = d'$	hardware states f^t, f^{t+1}
on the assembler *instruction* level	where t is a *hardware cycle*

Table 3: State transition of the DLX processor vs. the FlexRay interface

The interface between processor and devices uses ports and memory-mapped I/O for information exchange. With the help of Figure 51, we can recall the memory map, and Figure 52 on the next page shows the processor bus, which has attached the CPU, the memory unit, and the FlexRay interface. The memory unit will serve all load and store requests that concern the address range A of the conventional memory. The FlexRay interface, however, will serve load and store requests that concern the address range $[ba(f) : ba(f) + K)$, where K is the number of ports.

Now, we try to define the access of the processor to ports of device f. The naïve

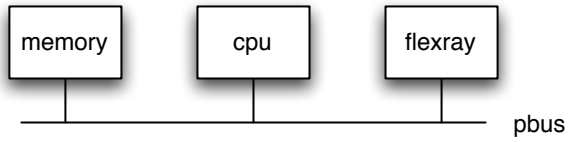


Figure 52: The processor bus

approach would be, e. g.,

$$lw(d) \wedge ea(d) = ba(f) + \gamma \text{ for all } \gamma \in [0 : K)$$
$$\implies d'.gpr(x) = \begin{cases} f.port_4(\gamma) & : x = RD(d) \\ d.gpr(x) & : \end{cases}$$

As we can see, the definition of the consecutive DLX configuration $d' = \delta_{\text{DLX}}(d, f)$ relies on some FlexRay interface configuration f. Where do we get these configurations from?

Similarly, a FlexRay interface configuration f^{t+1} has to be defined dependent on a DLX configuration, e. g., $f^{t+1} = \delta_{\text{f}}(d, f^t)$:

$$sw(d) \wedge ea(d) = ba(f) + \gamma \implies f^{t+1}.part_4(\gamma) = d.gpr(RD(d))$$

This works fine for many control ports, i. e., if $\gamma \in (8\lambda : K)$, but not for the send- and receive-buffer ports. If a port for the send buffers or for the receive buffers is accessed, we have to figure out, which of the both send buffers sb_0, sb_1—or receive buffers rb_0, rb_1, respectively—, we have to access. **[TODO:** We have exactly the same problem for lw, haven't we?**]**

As we know, the selection of the according buffer depends on the slot counter. In odd slots, we use the buffers sb_0 and rb_0, and on even slots, sb_1 and rb_0 are used. Hence, we should define a function $par(f)$ that computes the parity bit of the slot counter, i. e., $par(f) = f.F.slot \mod 2$. A load-word instruction for a $\gamma \in [4\lambda : 8\lambda)$ would now read from $rb_{\neg par(f)}$. **[TODO:** Explain the problem *either* on a load *or* on a store instruction.**]**

However, $par(f^t)$ is defined by the hardware construction but the CPU computation is defined on instruction level:

$$d^0, d^1, d^2, \ldots \qquad \text{with } d^{i+1} = \delta_{\text{DLX}}(d^i)$$

Hence, we need to define the corresponding computation for f on instruction level:

$$f_{\text{I}}^0, f_{\text{I}}^1, f_{\text{I}}^2, \ldots \qquad \text{where } f_{\text{I}}^i : f \text{during the execution of instruction } i$$

Suppose, we could define the function $par(i)$ that computes the parity of f's slot during the execution of instruction i. Then, we could define:

$$lw(d^i) \wedge ea(d^i) = ba(f_{\mathrm{I}}^i) + \gamma \text{ for all } \gamma \in [4\lambda : 8\lambda)$$
$$\implies d^{i+1}.gpr(x) = \begin{cases} f_{\mathrm{I}}^i.rb_{\neg par(i),4}(\gamma) & : x = RD(d) \\ d.gpr(x) & : \end{cases}$$

Caution! For any complex CPU hardware it is impossible to define $par(i)$ on machine instruction level.

Proof In real systems, the function $par(i)$ changes depending on the real-time timer. However, the real-time execution time of an instruction depends heavily on the hits in the cache, which is not visible in the DLX model.

Solution The device sends interrupts when the parity changes. Now, we can define the parity function as the number of interrupts received until instruction i (mod 2).

However, the number of received interrupts depends as well on real time. On a *pure* assembler level model, the arrival times of interrupts are inherently nondeterministic!

$\implies$ Even on assembler level, there is *no way* around nondeterministic models.

Definition 14.1 *Interrupts at instruction level.*

Let j be the index of a software interrupt, and let II denote the set of indices of internal interrupts. Now, we define the predicates

$is\text{-}ev(j) \iff j \in II$, *interrupt j is an internal event signal*
$is\text{-}eev(j) \iff j \notin II$, *interrupt j is an external event signal*

Figure 53 on the following page illustrates the different sources of interrupts: internal interrupt signals like overflows are generated by the ALU in the CPU itself, while external interrupts like the timer come from the outside.

In the following, we will denote the vector of external signals 'for' instruction i with eev^i. With this definiton, we can express the transition function δ_{DLX} by means of the old configuration and the interrupt vector as external input:

$$d' = \delta_{\mathrm{DLX}}(d, eev)$$

We define the interrupt cause vector ca with:

$$ca(d, eev)[j] = \begin{cases} eev[j] & : j \notin II \\ ev(d)[j] & : j \in II \end{cases}$$

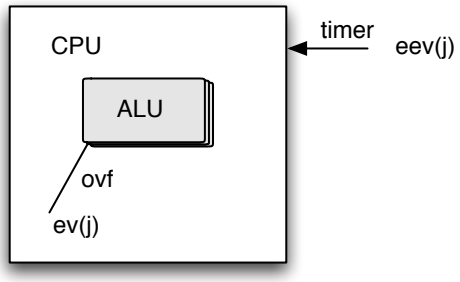


Figure 53: Interrupts and their sources

[TODO: There is something wrong with the interrupt numbers! The j^{th} internal interrupt signal is not necessarily the j^{th} signal in the combined vector!**]**

The masked interrupt cause mca is defined as

$$mca(d, eev)[j] = \begin{cases} ca(d, eev)[j] \wedge d.SR[j] & : j \text{ is maskable} \\ ca(d, eev)[j] & : \end{cases}$$

where *SR* is the status register.

With this definition, we define the predicate *jump to the interrupt service routine (jisr)* as

$$jisr(d, eev) = \bigwedge_j mca(d, eev)[j]$$

Whenever an interrupt occurs, the DLX assembler machine will jump to the start address of the interrupt service routine (*SISR*). We formalize this fact with the following implication:

$$jisr(d, eev) \implies d'.dpc = SISR$$

Of course, we have to redefine the semantics of our instructions such that the old definition is only valid if *jisr* does not hold:

$$\neg jisr(d, eev) \implies \text{old semantics}$$

14.1 Generating Timer Interrupts of f

At first, let us attempt the naïve approach. Suppose, the interrupt number of the timer interrupt is 14, hence we have just to define, when $eev[14]$ is seen at the processor. It is easy to tell that the timer interrupt will always occur if the slot number changes:

$$timerint^t \iff F^t.slot[0] \neq F^{t-1}.slot[0]$$

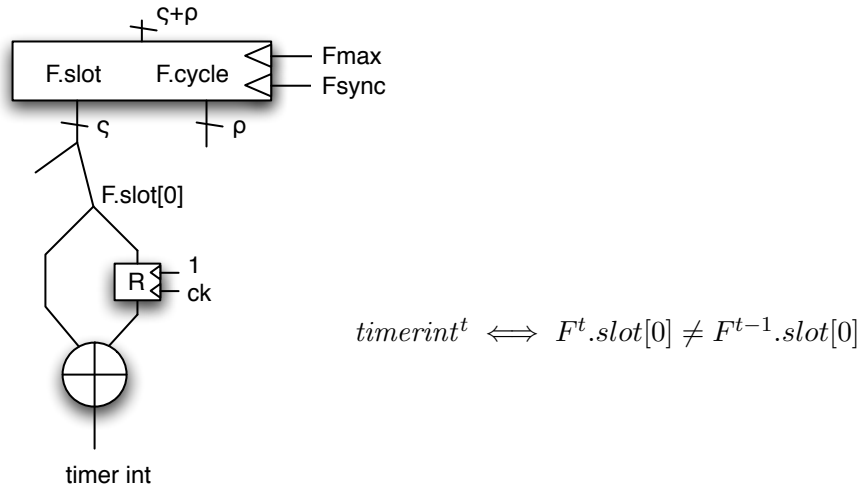


Figure 54: Computation of the timer interrupt

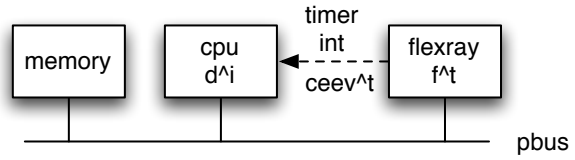


Figure 55: The processor bus and the timer interrupt line

The circuit diagram for the timer interrupt line is shown in Figure 54 on the next page.

However, we have a problem with its mathematical definition: It works again on the cycle level—we can only use it to define an external-signal vector $ceev^t$ for cycle t but not for eev^i, where i denotes the effected instruction. The situation is illustrated in Figure 55. This problem is generic for all external interrupts. How are external interrupts caught? How do we formalize, what happens?

For an answer, we have to understand how processors are constructed. In Figure 56 on the next page, we find two typical process designs. On the left hand side, we see a classical pipelined CPU with the typical five stages instruction fetch (IF), instruction decode (ID), instruction execution (EX), and finally write back (WB). On the right hand side, there is a CPU with *out of order* execution. We see the same stages, reservation stations RS, functional units FU, the producers P and the reorder buffer ROB.

Though the latter design is somewhat more complex, external interrupts are caught in both cases in the write back stage. Hence, we need a definition of $WB(i) = t$ such that instruction i is in stage WB during cycle t. We can define

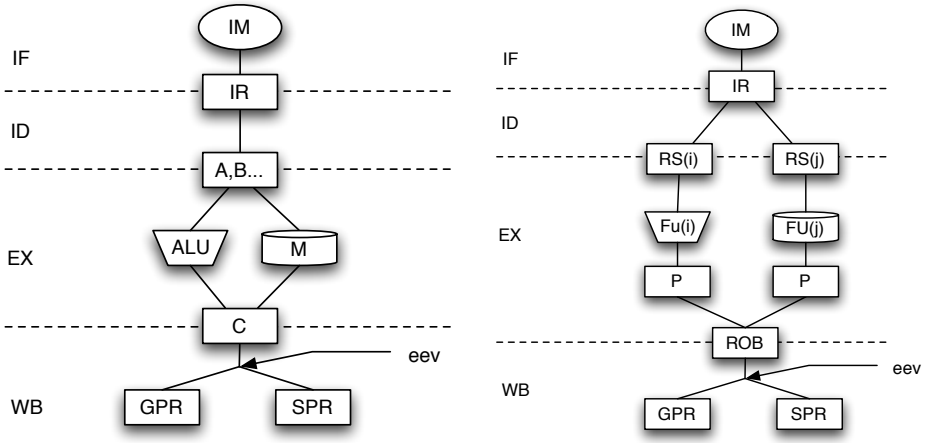


Figure 56: Typical processor constructions in comparison

this function, if we know the scheduler. The scheduler is a function

$$S(k,t) = i$$

which defines that instruction i is in stage k during cycle t. In other words, $WB(i) = t \iff S(WB, t)$.

Finally, we can define our external signal vector on instruction level as:

$$\boxed{eev^i = ceev^{WB(i)}}$$

15 060206 Jan Dörrenbächer

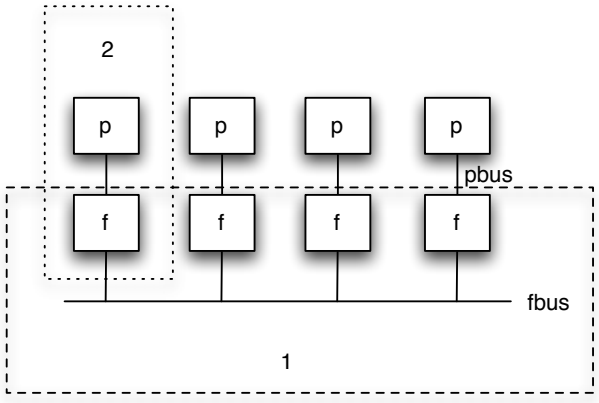


Figure 57: Flex Ray Bus

In the former lectures we always considered the whole block (figure 57, block 1) consisting of the flex ray bus and the different $ECUs$ with the flex ray controllers. According to this, we have a transition function based on the state of the whole system.

This is inconvenient since we want to state a theorem regarding to the interface between a flex ray controller and an upcoming processor (figure 57, block 2). Thus, we require a transition function for each entity consisting of the DLX processor and the flex ray controller:

$$\delta(d, f) = (d', f')$$

The current configuration of the DLX processor is given by d and the current flex ray configuration is denoted as f. The transition function computes the consecutive configurations d' and f'.

Some words on notation: The processor configuration for the ISA-computation (DLX) is denoted by d. The configuration of the actual hardware is given by h.

In the scope of flex ray, f gives the abstract configuration and fh the configuration of the hardware.

Note, the parity bit (switched by the timer) is inherently non-deterministic on pure ISA-level. The external event signals are given by $eev(j)$ for some j. The computation of the timer interrupt is depicted in figure 67.

16 Processor (hardware) Correctness

The main goal towards the processor correctness is to show, that the hardware simulates the instruction set architecture (ISA). Thus, we consider the computation

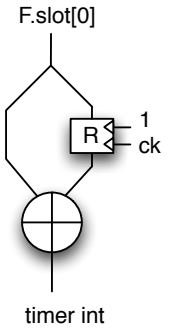


Figure 58: Timer Interrupt

of the hardware and the ISA and formulate correctness statements, afterwards.

On the hardware we have a sequence of configurations $h^0, h^1, \ldots$. We get this sequence by means of the transition function

$$h^t = \delta_H(h^{t-1}, ceev^{t-1})$$

The ISA computation looks similar. We again have a sequence $d^0, d^1, \ldots$ of ISA configurations obtained by the transition function

$$d^t = \delta_{DLX}(d^{t-1}, eev^{t-1})$$

At this point we get the problem, how we deal with the interrupts. In particular: Which $ceev^t$ can be seen by instruction $I(d^i)$? The answer is the following. The instruction $I(d^i)$ sees the interrupts of the hardware in the write back cycle, i.e.

$$ceev^{wB(i)}$$

where

$$wB(i) = \{t \mid S(wB, t) = i \wedge full^t_{wB}\}$$

The scheduling function S is defined subsequently. It computes the current instruction in a stage.

In the following we require some control signals:

- ue_k: update enable signal for all registers in stage k
- $full_k$: full bit of stage k (a bubble in the pipeline could mean $full_k = 0$).

Definition of the scheduling function: At the very beginning, i.e. in cycle 0, we are in all stages before the execution of instruction $I(d^0)$.

$$S(k, 0) = 0$$

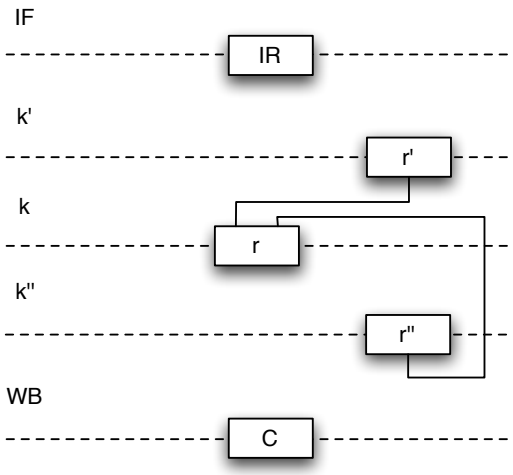


Figure 59: Processor Stages

In the instruction fetch (IF) stage we have to ensure, that the instructions are fetched in order.

$$S(IF,t) = \begin{cases} S(IF,t)+1, & if\ ue^t_{IF} = 1 \\ S(IF,t), & \text{otherwise} \end{cases}$$

In all other cases, the next instruction in stage k depends on the stages delivering input for k (see figure 59).

$$S(k,t+1) = \begin{cases} S(k',t), & \text{if } ue^t_k \wedge update^t \text{ was from stage k'} \\ S(k'',t), & \text{if } ue^t_k \wedge update^t \text{ was from stagek"} \\ S(k,t), & \text{otherwise} \end{cases}$$

16.1 Correctness Statements

The correctness statements which we want to conclude consider (i) the registers R in stage k of the hardware, (ii) the general purpose register file GPR and (iii) the memory.

For registers R and the general purpose register file, we have to show, that

$$h^t.R = d^{s(k,t)}.R$$

and

$$h^t.GPR[a] = d^{s(wB,t)}.gpr[a]$$

The proof of this equivalence is relatively easy since we have counterparts in both configurations.

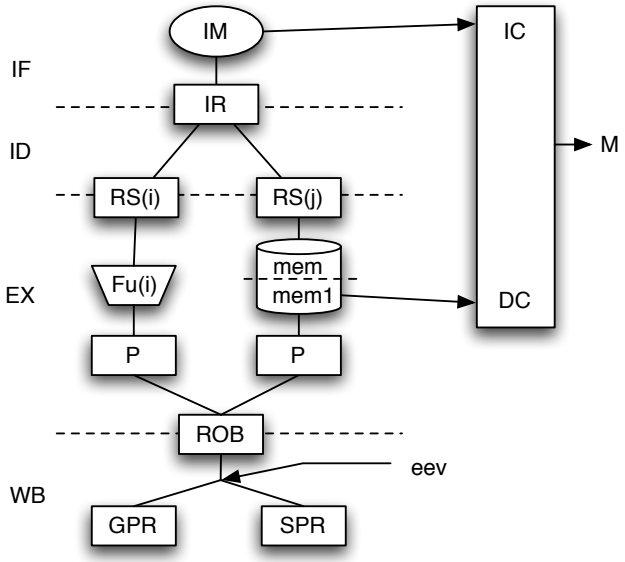


Figure 60: Processor Stages with Cache

Life is getting more exciting if we consider the memory. What we would like to have is the following:

$$h^t.m(a) = d^{s(mem1,t)}.m(a)$$

But, the memory is not a component of the hardware. Thus, we have to find another solution.

Elegant solution (provided by Sven Beyer):
Assumption: There is an interface to the memory system (between $mem1$ and data cache DC in figure 60) which provides the following signals:

- $msad$: the address to the memory system
- $msdin$: the data input
- $msdout$: the data output
- msr: the read request
- msw: the write request

Let $p(h)$ be a predicate on a hardware configuration h.

The last cycle before t, when p holds for $h^{t'}$ is given by

$$last_p(t) = \max\{t' < t \mid p(h^t)\}$$

There is a write access to the memory system at address a, if

$$mswrite(h, a) = (msadd(h) = a) \wedge (msw(h) = 1) \wedge \neg dbusy(h)$$

holds. The address a has to correspond with the address provided by the hardware in configuration h, the hardware must request a write access and the busy signal must be inactive.

By means of the definitions above, we can define

$$m(t)(a) = \begin{cases} msdin(h^{t'}) & : \exists t''.mswrite(h^{t''}, a) \\ m_{init}(a) & : otherwise \end{cases}$$

with

$$t' = last_{mswrite(h,a)}(t)$$

This allows us to reformulate the correctness statement for the memory:

$$m(t)(a) = d^{s(mem1,t)}.m(a)$$

The problem with this definition is, that it defines a function of time.

There are hidden parameters $h^0, h^1, \ldots, h^{t-1}$. As already mentioned, we get the consecutive configuration through

$$h^{t+1} = \delta_H(h^t, ceev^t)$$

A definition, where the memory is described as a function of a configuration, i.e. $m(h)(a)$, would be much more desirable.

In order to do this, we first have to go into detail with the memory system construction.

16.2 Memory System Construction

We have a w-way cache (figure 61) with the particular caches $c[i]$ defined as direct mapped caches.

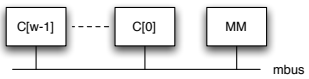


Figure 61: W-Way Cache

The cache address, depicted in figure 62, is subdivided in three fields containing the tag, the line address and the offset.

The architecture of the caches $c[i]$ is given in figure 63. Altogether, we have $L = 2^l$ lines of data. Each line is provided with a tag and valid bit.

A cache hit is signaled through

$$hit_i(h, a) = (a.tag = h.c[i].tag[a.line]) \wedge (h.c[i].v[a.line])$$

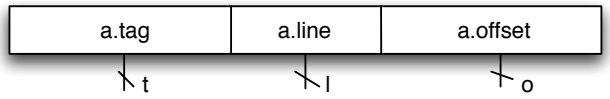


Figure 62: Cache Address

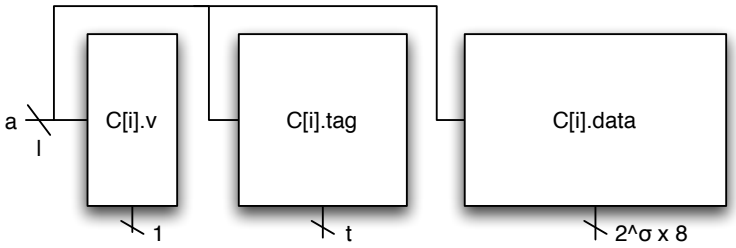


Figure 63: Cache Architecture

Since we have two caches, one for instructions IC and one for data DC, we write $Ihit_i(h, a)$ to signal a instruction cache hit and $Dhit_i(h, a)$ for a data cache hit.

Thus, the memory definition can be written as:

$$m(h)(a) = \begin{cases} h.DC[i].data[a.line][a.off] & : Dhit_i(h, a) \\ h.IC[i].data[a.line][a.off] & : Ihit_i(h, a) \\ h.mm(a) & : otherwise \end{cases}$$

As invariant, we demand that there will not be concurrent hits in the caches at the same address:

$$\neg(Dhit_i(h, a) \wedge Ihit_i(h, a))$$

The drawback of this definition is, that we do not have formal correctness proof until now.

16.2.1 Simulation Theorem

Having redefined the memory, we are now able to formulate the simulation theorem.

Theorem Let $sim(d^0, h^0)$ and $ceev^0, ceev^1, \ldots,$ be sequence of hardware external event signals.

Then, there is a sequence of ISA external event signals $eev^0, eev^1, \ldots,$ such that for the ISA-computation, defined by

$$d^{i+1} = \delta_{DLX}(d^i, eev^i)$$

we have for all cycles t:

- $h^t.RF[a] = d^{s(wB,t)}.RF[a]$
- $m(h^t)(a) = d^{s(mem1,t)}.m(a)$
- $h^t.R = d^{s(k,t)}.R$, where R is in stage k

Proof The proof depends on

- content of the lecture Computer Architecture I
- $eev^i = ceev^{wB(i)}$

Note: A programmer who does not know the hardware, does not know $s(k,t)$ and $wB(i)$. Thus, the occurrences of external event signals (interrupts) are non-deterministic. We can only remove this non-determinism if the hardware is known and interrupts are stable.

060208 Abdul Qadar Kara

Correctness Next we define the predicate $Corr(t)$ which is defined for the register file$R$ in stage k:

$Corr(t)$ holds if:

$$\begin{array}{lcl} h^t.R & = & d^{s(k,t)}.R \\ m(h^t)(a) & = & d^{s(mem1,t)}(a) \end{array}$$

This means that if at some cycle t, the contents of the memory as well as the registers in the hardware configuration are equal to their corresponding counterparts in the ISA structure of DLX machine, then their holds a correctness relation $(Corr(t))$between them during that cycle$t$.

Simulation Relation There are two types of simulation relations among the hardware configuration(h) and the ISA structure (d) of DLX machine we are interested in, in general, data simulation $(d\text{-}sim)$ which states that the content of the register file (except dpc and pc) and memory contents are equal in both hardware configuration as well as ISA structure of DLX machine, and control simulation $(c\text{-}sim)$, whcih tells us that both the configurations (i.e. hardware and ISA) have same current instruction and the instruction after that. Also is worth mentioning that we dont have any other instruction in the pipeline in hardware configuration otherwise, it might change the contents of either the memory or registers or both and it might also effect the control consistency if there is a jump instruction somewhere in the pipeline. Formally:

$$\begin{array}{llllll} d-sim(d,h) & d.R = h.R & \wedge & d.m = m(h) & R \notin \{PC, DPC\} \\ c-sim(d,h) & d.R = h.R & \wedge & drained(h) & R \in \{PC, DPC\} \end{array}$$

Predicate $drained(h)$ is defined as:

$$drained(h) \quad = \quad \bigwedge_{k \neq IF} / h.full_R$$

So, there should be no instructions in any stage in hardware configuration except in instruction fetch stage (IF).

Simulation Theorem

Theorem Let $sim(d^0, h^0)$ and $ceev^0, ceev^1, \ldots,$ be sequence of hardware external event signals.

Then, there is a sequence of ISA external event signals $eev^0, eev^1, \ldots,$ such that for the ISA-computation, defined by

$$d^{i+1} = \delta_{DLX}(d^i, eev^i)$$

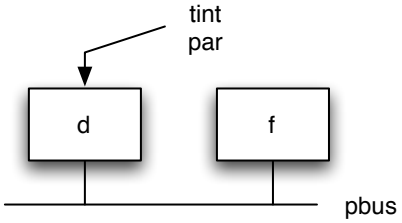


Figure 64: Connecting DLX Machine and Flexray Architecture

Claim Based on the simulation theorem defined in the previous lecture, we claim that for all cycles t the predicate $Corr(t)$ holds. This implies that the hardware configuration and the ISA structure of the DLX machine have same contents of registers as well as memory contents.

$$\forall_t.Corr(t)$$

Merging of DLX and Flexray at ISA level At this stage, we have non determinism because of parity par and timer interrupts $timerint$ are non deterministic at ISA level . What we do is, we generate a simulation dependent input sequence par^i and $timerint^i$ w.r.t. instruction i. This gives us new definition of transition function:

$$(d^{i+1}, f^{i+1}) \quad = \quad \delta(d^i, f^i, par^i, timerint^i)$$

Now, as we have the parity function defined (par^i), we can write :

$$lw(d^i) \wedge ea(d^i) = ba(f) + x + \gamma \quad (\gamma < x)$$
$$\implies d^{i+1}.gpr(y) = \begin{cases} f^i.rb_{\neg par(i),4}(\gamma) & : y = RD(d^i) \\ d.gpr(y) & : \end{cases}$$

Claim update Now that we have integrated both the DLX and Flexray hardware, we need to include that also in our previous correctness statement. Namely, now our correctness predicate (lets denote it with $Corr'(t)$) also includes the condition that the register file contents of both the hardware of flexray and its abstract interpretation should be equal. Formally, our predicate $Corr'(t)$ holds if:

$$\begin{array}{lll} Corr(t) & \text{holds} & \\ f_h^t.R & = & f^{s(mem1,t)}.R \end{array}$$

Note that this f_h depends on hardware cycle(the actual flexray interface and the subscript here is just used to make it more explicit.

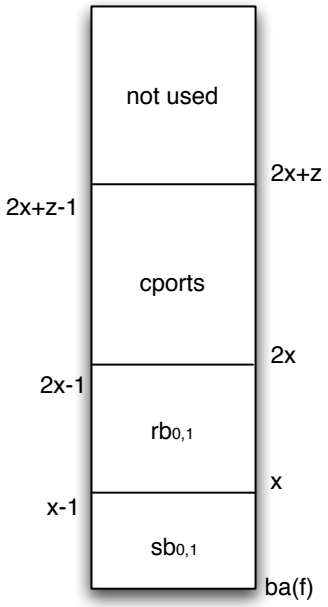


Figure 65: Flexray Ports

Right Definition for $timerint^i$ and par^i We already have presented the definition of $timerint^i$ which is to consider it as a normal external interrupt. Formally,

$$timerint^i \quad = \quad timerint(f_h^{WB(i)})$$

Parity Function par^i For the parity function, we know that it is dependent on the slot counter and it helps us to decide which send buffer (from sb_0, sb_1)and receive buffer (from rb_0, rb_1) do we access.

Now consider where we have either a load or a store instruction to one of the buffers from rb_x and sb_x where $x \in \{0, 1\}$.

Now, if we are lucky, we can perform it in one cycle because the parity function doesnt change (which helps us decide which buffer to access) in it. But this can only happen if the device is not busy, i.e. we dont get a busy signal. Another important point is that we get a cache hit, otherwise we need to run page fault handler and so the execution of the instruction wont be possible in a single cycle. Formally, for d^i:

$$\exists_t.\ mem1(i) = t : s(mem, t) = i \wedge full_{mem1} = 1$$
$$\implies par^i = f_h^{mem1(i)} F.slot \ \ mod \ \ 2$$

Otherwise, we would need to have some software requirements on the slot of the timer, mainly that they remain constant during such accesses and so the parity function would not change during such instructions.

$\forall_t$.During such accesses $s(mem1, t) = i \wedge f_h^i.F.slot$ remains constant.

WCET of DLX Programs Now we try to get the worst case execution time of our programs that run on our DLX machine.The input we use is an element from set of possible inputs E ($e \in E$). The calculation depends on simulations of these programs. We need to define some software restrictions or conventions explicitly in order to use the softwares that calculate the WCET of programs. These conventions are :

- Program P begins at base address $ba(PR)$ of the program region PR in memory.
- Input Data e begins at base address $ba(DR)$ of the data region DR in meory.
- Remaining registers in data region DR are initialized to 0.
- There is no access in the program P outside Program or Data regionsPR, DR.

On ISA level, these conventions would mostly be the part of initial configuration of DLX machine $d^0(P, e)$.

$$\begin{array}{lcl} d^0.R & = & 0 : R \notin \{PC, DPC\} \\ d^0.DPC & = & ba(PR) \\ d^0.PC & = & ba(PR) + 4 \end{array}$$

ISA Runtime The runtime of a program execution on any input data e would be the time time it takes from start till it executes the trap instruction. This instruction returns the control back to the Operating system. Formally,

$$T_{DLX}(P, e) \quad = \quad min\{t | trap(d^t(P, e))\}$$

ISA Result After the termination, we also need to check the result obtained by the program is correct and that the program evaluates a valid and expected result when given a correct input. Formally,

$$res_{DLX}(P, e) \quad = \quad d^{T_{DLX}(P,e)}(P, e)$$

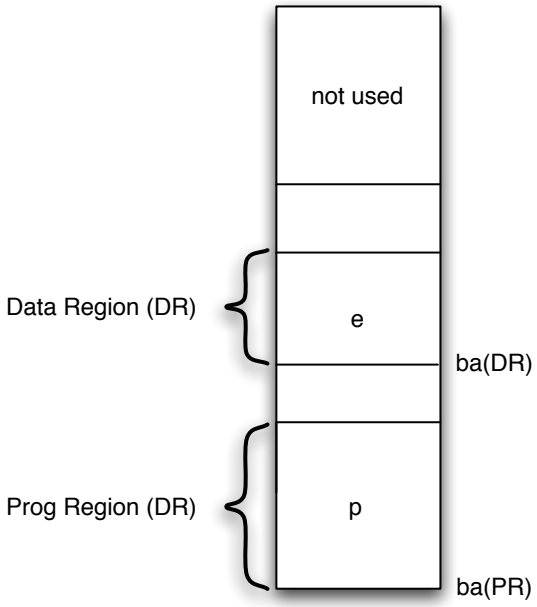


Figure 66: Memory Classification

Next we translate these on the hardware side.

Consider a hardware construction $H(P, E)$ shich takes a simulates a program P and can work on the set of possible inputs E. Then, we have its initial state as:

$$H(P, e) \quad = \quad \{h : sim(h, d^0(P, e)) \text{for some } e \in E\}$$

The runtime of this hardware configuration can be defined by both ways, either we check and stop whenever we have a trap instruction in our instruction register (or our DPC is pointing towards the address that contains a trap instruction opcode). The other way is that we check the 5^{th} bit of masked interrupt cause register mca which tell us that we have a trap instruction to execute. Formally,

$$\begin{array}{rcl} T_H(h) & = & min\{t | \exists_{t'} s(fetch, t') = s(WB, t) \wedge trap(m(h^{t'})_4(h^{t'}.DPC)\} \\ & = & min\{t | h^t.MCA[5] = 1\} \end{array}$$

$WCET(P, E)$ Worst case execution time of the hardware configuration time would simply be the maximum time it takes to fetch a trap instruction. Formally,

$$WCET(P, e) \quad = \quad max\{T_h(h) | h \in H(P, E)\}$$

In order to get the exact value of it, one has to use the softwares made by the Chair od Reinhard Wilhelm or AbsInt. One would get a certain number of cycles

t that would denote that any given input e would take a maximum of t cycles to execute.

Lemma

$$WCET(P, e) \quad < \quad t$$

Result Correctness of Hardware The result obtained by running a program P on some input e on our hardware configuration would give us,

$$res_H(h) \quad = \quad h^{T_H(h)}$$

Hardware Correctness We can also check the result correctness produced by the hardware configuration w.r.t. to the simulation relation we have between the ahardware configuration h and ISA structure d.

Lemma

$$sim(h, d^0(P, e)) \implies d - sim(res_H(h), res_{DLX}(P, e))$$

Another fact that can also be proven from the results above,

Lemma

$$sim(h, d^0(P, e)) \implies s(WB, T_H(h)) = T_{DLX}(P, e)$$

17 060213 Dominik Rester Small Corrections and Modifications

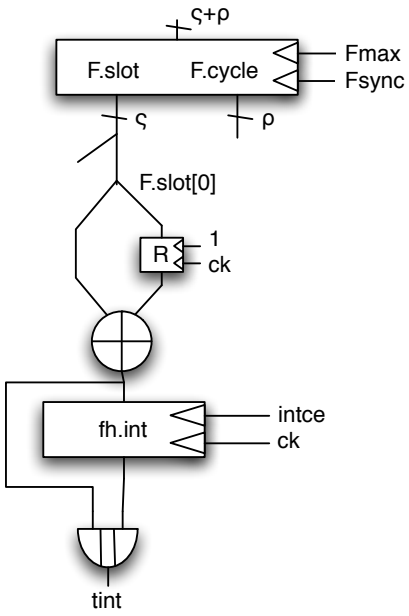


Figure 67: Generation of the timer interrupt

$ce(kT) = tint' \vee intclear$

The $intclear$ signal can be enabled with the intclear command, similarly to the setrd command.

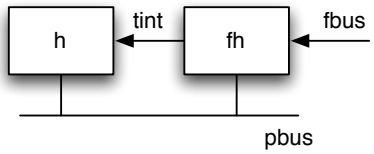


Figure 68: Hardware

On the ISA level we denote the next configuration by: $(d^{i+1}, f^{i+1}) = \delta_{DLX,F}(d^i, f^i, par^i, tint^i, rbDinf^i)$

If the parity changes its value between i and $i+1$, i.e. $par^i \neq par^{i+1}$, we have: $f^{i+1}.rb_{par^i} = rbDinf^i$

$h^t.R = d^{s(k,t)}.R$

$h^t.m = d^{s(mem1,t)}.m$

For the processor side, i.e. $x = \overline{par(fh^t)}$ with $par = fh^t.F.slot \mod 2$:

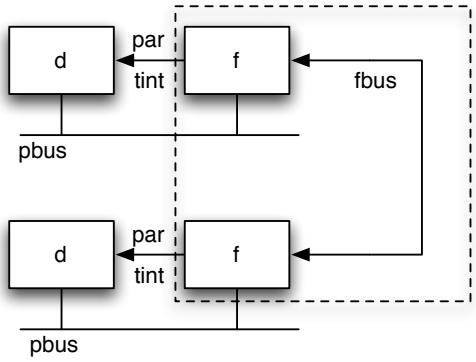


Figure 69: Instruction Set Architecture

$fh^t.sb_x = f^{s(mem1,t)}.sb_x$
$fh^t.rb_x = f^{s(mem1,t)}.rb_x$

17.0.2 Simulation Theorem

We define $tint^i = tint(fh^{WB(i)})$ $par^i = par(fh^{mem1(i)}) \stackrel{!}{=} par(fh^{WB(i)})$

Software condition:
$m_1WB(i) = \{t|(s(mem1,t) = i \wedge full^t_{mem1}) \vee (s(WB,t) = i \wedge full^t_{WB})\}$
$\forall i : I(d^i)$ is lw or sw Instruction with ea in portrange of $f \Rightarrow \exists x(i) : \forall t \in m1WB(i) : par(fh^t) = x(i)$

17.0.3 Completion of definition of scheduling functions across interrupts

$JISR(h^t) \Rightarrow drained(h^{t+1})$
$s(IF, t+1) = s(WB, t) + 1$
In general it holds: $\forall k s(k, t+1) = s(WB, t) + 1$

(end of corrections)

Definition:
We denote round with r, slot with s and define:

$$(r,s)+1 = \begin{cases} (r, s+1) & s < ns-1 \\ (r+1, 0) & s = ns-1 \end{cases}$$

$$(r,s)-1=\begin{cases}(r,s-1) & s\neq 0\\ (r-1,ns-1) & s=0\end{cases}$$

Furthermore we call $\varepsilon_v(r,s)$ the time when parity changes on ECU_v at the start of slot (r,s). The choice of the synchronization points α_v is the falling edge of $reset_v$. Remember, that ns is a even number.

$$\varepsilon_v(r,s)=\begin{cases}\alpha_v(r,s) & s\neq 0\\ \alpha_v(r-1,ns) & s=0\end{cases}$$

The times $\varepsilon_v(r,s)$ are clock edges $e_v(i)$ for some i
$\varepsilon_v(r,s)=e_v(t\varepsilon(r,s))=\alpha_v+t\varepsilon(r,s)\cdot\tau_v$
$\Leftrightarrow t\varepsilon_v(r,s)=\frac{\varepsilon(r,s)-\alpha_v}{\tau_v}$
TODO BILD

Theorem1:
We state, that the $receivebuffer$ gets the content of the sending ECU's $sendbuffer$ at the beginning of the slot:

$$fh_v^{t\varepsilon_v(r,s)}.rb_p=fh_u^{t\varepsilon_u((r,s)-1)}.sb_p$$

and equivalent, considering only instructions:

$$rbdin_v^{s(mem1,t\varepsilon_v(r,s))}=fh_u^{s(mem1,t\varepsilon_u((r,s)-1))}.sb_p$$

Variation on WCET

We call a the address of an instruction and define

$$T_{DLX}(P,e,a)=\min\{t|\delta_{DLX}^t(d^0(P,e)).DPC=a\}$$

as the number of instructions until the instruction an address a is fetched.

$$T_H(h,a)=\min\{t|\exists t':s(WB,t)=s(IF,t')\wedge f_H^{t'}(h).DPC=a\}$$

$WCET_H(P,a)=\max T_H(h,a)\ h\in H(P,E)$ Here we denote with h the configuration after the timer interrupt: $h=h_v^{t\varepsilon_v(r,s)+1}$ and with P the entire Program in Assembler.

18 060215 Alexandra Tsyban
C0 semantics

C0 language C0 is a restricted version of the C programming language. Basically, it is Pascal with C syntax. The reasons for the creation of such a language are: i) the example of the simple but powerfull Pascal semantics was given in the literature already by Wirth, ii) the C-like syntax is an industrial standart for modern programming languages.

Now we are going to give the functional operational semantics of C0. The meaning of a C0 program is a C0-machine. Thus, first of all we define the configuration of that machine:

$$c = (pr, rd, lms, hm)$$

This configuration contains the following components:

- pr - program rest. This is the sequence of C-instructions to be executed.
- rd - recursion depth. This components define the number of function calls.
- $lms : [0 : rd] \to \{local\ memories\}$ - stack of function frames. At the first frame (with index 0) the global variables are allocated.
- hm - heap memory. Here we store the variables which are allocated during the programming run.

With $top(c)$ we define the top most function frame $c.lms(c.rd)$. Also each c0-machine has some parameters:

- $TT : \{type\ names\} \to \{type\ descriptors\}$ - type table. This table collect all names of declared types.
- $FT : \{function\ names\} \to \{types\} \times \{bodies\}$ - function table. It maps function names to its types and bodies.

For the variables we use the following notation: (m, i), that means the i-th variable in the memory m. Of course, some variables might have a complex type. The following example shows you how the subvariables could be denoted. Imagine, we have the array of process control blocks, one for each process (the third global variable). This block might be a structure which one element might be an array for storing the values of general purpose registers. To denote the C0-variable which stores the register 17 of the process 112 we write:

$$(lms(0), 3)[112], gpr[17]$$

To get the value of the variables we use the function $va(m, i)$.

Pointers could point to the subvariables, that makes difficulties if we have the garbage collector.

C0 semantics Here we take a look only into the semantics of the function call. Assume the head of the current program rest in configuration c is function call:

$$c.pr \quad = \quad id = f(e_1, ..., e_n); r$$

The effect of the function call is defined as following. The program rest is extended with the body of the function

$$c'.pr = c.FT(f).body; r,$$

the new procedure frame is created

$$c'.rd = c.rd + 1,$$

all parameters are passed

$$...$$
$$top(c')(i) = va(c, e_i)$$
$$...,$$

and the address of the return destination is saved

$$top(c')(0) = \&(c, id)$$

The effect of the execution of the trap instruction could be defined in the similar way.

Compiler correctness Now we are going to the compiler correctness. The compiler translates C0-programs represented by C0-machine into DLX-program running on a DLX-machine. The compiler correctness hinges on the simulation relation $consis(c, alloc, d)$ about consistency between the C0-machine c and the DLX-machine d with the help of the allocation function $alloc$. This function maps the C0-variables to the addresses in the DLX-machine which they are corresponding to. The function might be changed by the garbage collector or the function call.

The consistency concerns:

- $r - consis$ - implementation of the stack and the heap.
- $e - consis$ - data consistency for elementary subvariables: the value of an elementary subvariable x is stored in the region of virtual memory allocated to it
$$d.vm_{asize(x)}(alloc(c, x)) = va(c, x)$$
- $p - consis$ - data consistency for pointers. Let y be a (sub)variable and p be a pointer to y: $va(c, p) = y$. Then
$$d.vm_4(alloc(c, p)) = alloc(c, y)$$
- $c - consis$ - control consistency. The program counter should point to the first address ($start$) of the compiled code ($code$) of the first statement ($head$) of the program rest:
$$d.pc = start(code(head(c.pr)))$$

$C0_A$ **semantics** Now we are going back to the operating system. After the handling of the interrupts the operating system have to be able to restore the state of the process as it was before the interrupt. For this purpose it should save the registers of the process into C-variables. But in the C you don't see the hardware, I/O ports and other processes. In this situation inline assembler is used. Defining the semantics of C0 language with inline assembler $C0_A$ we need to keep in mind that assembler could change some C variables.

$C0_A$ configuration has the same format as C0 configuration. It talks only about the C0-variables of the program. We only extend C0-statements by new statement of form $asm(s)$ for the inline assembler. After switching to the assembler many things become visible. Thus, the new transition function δ_{C0_A} is defined for two arguments: C0-machine c and the DLX-machine d corresponding to the C0-machine.

As long as no inline assembler code is used, we ignore DLX machine and perform computation as in C0-semantics

$$\delta_{C0_A}(c, d) = \delta_{C0}(c)$$

In the case the program rest has the form

$$c.pr = asm(u); r'$$

we execute t steps of the DLX machine which are enough for execution of whole assembler part u

$$d \rightarrow_u^t d' : \quad d = d_0 \rightarrow_{DLX} ... \rightarrow_{DLX} d_t = d'.$$

Of course, execution of this portion of code should not force the program counter jump out of the code. After the execution program counter should point to the instructions after the assembler portion, i.e. to the start of the code of the next statement.

The assembler code might change the C0-variables. In order to keep data consistency we define the sequence of C0-machine $c_0, ..., c_j, ...c_t$. If the execution of the current instruction in the state d_j changes the C0-variables x, this means it is the store word instruction and the effective address of this instruction equal to the address where the variable x is allocated, then we define the new value of this variable as the content of the destination register:

$$sw(d_j) \wedge ea(d_j) = alloc(c, x) \rightarrow va(c_{j+1}, x) = d_j.GPR(RD(d_j))$$

Now we define the transition function for this case as

$$\delta_{C0_A}(c, d) = c_t$$

CVM: Communication Virtual Machine Now we define something which we call communication virtual machine (CVM). CVM is the abstract (pseudo) parallel user model of the kernel. The configuration of the CVM has the following components:

$$cvm = (ca, ..., vm(i), ..., vmsize(i), ..., cp, ...)$$

- ca - C0-configuration of the abstract kernel k
- $vm(i)$ - DLX-configuration of th i-th user
- cp - specifies the current running process
  - $cp = 0$: kernel is running
  - $cp = i$: $vm(i)$ is running

Abstract kernel has some number of funny functions kcd that are implemented in $C0_A$ but the syntax is placed in $C0$. There is no inline assembler code in CVM because user processes are visible in the parallel model.

The kernel call is actually the trap instruction. The parameter of this trap instruction specifies the function of the kernel you suppose to call, i.e. $trap\ i$ calls function $kcd(i)$ of the kernel k.

CVM semantics and implementation Now we define the effect of the operating system on the entire system without inline assembler.

If the kernel runs $cp = 0$ then the C-instruction is executed as defined above

$$cvm'.ca = \delta_{C0}(cvm.ca)$$

If the user machine i runs $cp = i$ and there is no interrupts $/JISR(cvm.vm(i))$ then one assembler instruction is executed

$$cvm'.vm(i) = \delta_{DLX}(cvm.vm(i))$$

Since the user machines are the virtual machines it might be page fault interrupt. But this interrupt is not visible for the virtual machine. Swapping in and swapping out is performed and the program continues its execution. Here we should be able to talk about external devices, and we need also hardware correctness of the memory management unit.

One of the kernel functions which are not implemented in CVM is starting the process that is scheduled as next. The effect of this function on the CVM is the following. So, if during the kernel running $cp = 0$ the program rest is $cvm.ca.pr = startnext; r'$ then the next running process is the process saved in the variable cp of the kernel

$$cvm'.cp = va(cvm.ca, cp)$$

The implementation of this function is done by using inline assembler. It saves the registers of the kernel into the C0-variables and restores the registers of the process from the corresponding C0-variables.

The most exciting thing is using trap instruction by one of the user. Literally it is the same as function call, but here one should distinguish between the process and the kernel. Let user i execute the trap instruction $trap\ j$. And let $f = kcd(j)$

be the handler for the trap j with n number of parameters. Then program rest and recursion depth are changed as in the function call

$$cvm'.ca.rd = cvm.ca.rd + 1$$

$$cvm'.ca.pr = cvm.ca.FT(f).body; cvm.ca.pr,$$

but as the parameters the value of the registers are passed

$$top(cvm'.ca)(x) = cvm.vm(i).GPR[x]$$

Since after the interrupt all registers are saved in the C0-variables by the kernel, trap is implemented as an ordinary function call:

$$f(PCB[i].GPR[1:n])$$

After the proving the correctness of the performing the register store, it is nothing to prove.

OLOS

This section shows us how we can get OLOS from the CVM (OLOS states for the OSEK-time like operating system). Such an operation system runs on each ECU and provides task abstraction and communication primitives. Furthermore it implements the drivers for the FlexRay interface.

OLOS configuration First of all we define the components of the OLOS configuration $olos$:

- $olos.c(0)$ - the kernel. Actually it is ordinary $C0$-machine.
- $olos.c(i)$ - i-th user program (for all i greater than zero). They all are $C0$-machines as well.
- $olos.cp$ - current program. This components specifies the currently running program. If this components is equal to 0 then it is kernel, otherwise it is one of the user program.
- $olos.f$ - FlexRay controller.

There are also certain variables of the kernel. The variables s keeps the number of the current slot. The value of this variable is increased after each timer interrupt. $FTcom$ is an array of messages $msg(m)$, where each message msg: $int(l/4)$ consists of l bytes.

Communication As it was mentioned above we have multiprocessing on each ECU. This means for every ECU_v we have $olos_v$ and $FTcom_v$. In this situation we have two kinds of communication:

1. communication between two programs on the same ECU
2. access to other ECUs

Both kinds of the communication of the processes are performed via variables in $FTcom$.

Now we have two scheduling function. First one is the local scheduling tables $sendl$ for the local send (as before) and the second one is $fttrans(s)$. This scheduling function specifies the message that should be transmitted in the slot s.

Transmission If we want to transmit a message at some slot, we should put this message into the send buffer at the slot before and take it from the receive buffer at the beginning of the following slot.

So, we have two drivers called OLOS send $osend$ and OLOS receive $orec$. $osend$ will be started before the end of a slot, $orec$ will be started at the start of a slot.

Now let us imagine $osend$ in slot s. What is it supposed to do? It should copy the value of what is transmitted in the next slot. The effect of $osend$ in slot s is the following. The value of the message scheduled to be transmitted in the next slot

$$va(olos.c(0), FTcom(va(olos.c(0), fttrans(s+1))))$$

is copied to the send buffer

$$olos.f.sb_{va(olos.c(0),s) \bmod 2}.$$

Of course, this is copied only if $v = send(s)$.

For $orec$ we copy $olos.f.rb_{s \bmod 2}$ into $FTcom(fttranse(s-1))$. In two last formulas the operations $s+1$ and $s-1$ are performed modulo ns.

On the software side we have two system calls: $ttsend(e, e')$ and $ttrec(e, e')$, where e' is the index where $FTcom$ is accessed and e specifies (sub)variable which should be copied to/from the buffer.

The effect of the send is the following: if i-user process executes $ttsend$ ($olos.cp = i$) then

$$va(olos'.c(0), FTcom(va(olos.c(i), e'))) = va(olos.c(i), e)$$

For the receive it is vice versa:

$$va(olos'.c(i), e) = va(olos.c(0), FTcom(va(olos.c(i), e')))$$

The system calls are implemented via $trap$ instruction. For example,
$trap$ 1 - $ttsend$
$trap$ 2 - $ttrec$
...

WCET Now we are going to compile and link all together. As a result we get the program P from WCET-analysis. Consider the structure of P: after the timer interrupt first that will be executed is some part of initial code from kernel

test for the reset
save state of the last program which was running
increase the number of current slot $s = s + 1$
$orec$
schedule first user
...
$trap$ 3 (return to OS)
$osend$
wait in idle loop for timer interrupt

Recall that we have definition $t\epsilon_v(r, s)$ - the cycle on ECU_v in slot s in round r when timer interrupt is turned on. From the hardware construction we know that the difference between two timer interrupts is at list T minus offset:

$$t\epsilon_v((r, s) + 1) - t\epsilon_v(r, s) \geq T - off$$

here $(r, s) + 1$ is also modulo addition. This means that in cycle

$$t \in t\epsilon_v(r, s) + [0 : T - off]$$

hardware parity is constant

$$par(ft_v^t) = s \ mod \ 2$$

and timer interrupt is off

$$tint(ft_v^t) = 0$$

We should keep in mind that in the definition of WCET there is a hidden parameter:

$$WCET(P, e, a, ?)$$

where $e = d_v^{t\epsilon_v(r,s)}$. This hidden parameter is a sequence $I(r, s)$ of external interrupts in this slot. If we make the interval between the interrupts of length $T - off$ then all what we want to show is that for any e WCET for the program P until it is reaching a without being interrupted:

$$WCET(P, e, a, 0^{T-off}) \leq T - off$$