Realistic Worst-Case Execution Time Analysis in the Context of Pervasive System Verification

Steffen Knapp* and Wolfgang Paul

Saarland University, Computer Science Dept., 66123 Saarbrücken, Germany
{sknapp,wjp}@wjpserver.cs.uni-sb.de

Abstract. We describe a gate level design of a FlexRay-like bus interface. An electronic control unit (ECU) is obtained by integrating this interface into the design of the verified VAMP processor. We get a time triggered distributed real-time system by connecting several such ECU's via a common bus. We define a programming model for such a system at the instruction set architecture (ISA) level and prove that it is correctly implemented at the gate level. The proof combines theories of processor correctness, communication systems, program correctness and realistic worst-case execution time (WCET) analysis into a single unified mathematical theory.

1 Introduction

1.1 Pervasive Verification and Unified Theory

The results of this paper were obtained under the German Verisoft project that aims at the development of tools and methods for the pervasive formal verification of computer systems. Pervasive correctness theorems argue simultaneously about the correctness of several system components like: Processors, I/O devices and programs. For real-time systems the correctness proofs are also based on the fact that certain computations are performed within certain time bounds.

Here we consider a distributed real-time system. In the pervasive correctness proof of this system we combine theories of processor correctness, communication systems, program correctness and realistic worst-case execution time (WCET) analysis [Abs06] into a single unified mathematical theory in the following sense:

Concepts shared between theories must not only be defined using the same formalism (this can be done easily using e.g. set theory) they must be defined *literally* in the same way in all theories concerned. Hardware correctness proofs of processors, I/O-devices and networks must use the same hardware- and the same instruction set architecture (ISA)[1] model. Correctness proofs of communicating assembler programs must use literally the same ISA model, too.

* Work partially funded by the International Max Planck Research School for Computer Science (IMPRS) and the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

[1] In a nutshell the ISA is an assembler semantic with interrupts visible, i.e. syntactic sugar for the machine language.

T. Reps, M. Sagiv, and J. Bauer (Eds.): Wilhelm Festschrift, LNCS 4444, pp. 53–81, 2007.

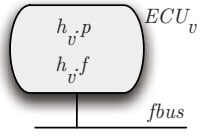

Fig. 1. Electronic Control Units

WCET analysis done for hardware models with real-time timers must clearly be based on cycle counts of the hardware. Yet in a pervasive theory of real-time systems this execution time analysis must be *formally* combined with program correctness proofs based on the ISA model, where caches are invisible and hence argumentation about the exact cycle count is impossible.

1.2 System Overview

The distributed real-time system considered here is very similar to systems used in the automotive industry: A fixed number p of electronic control units ECU_v for $v \in \{0, \ldots, p-1\}$ are connected via a FlexRay-like bus; applications run with a fixed schedule under an OSEKtime-like [OSE06] real-time operating system.

FlexRay is a communication protocol for safety critical real-time automotive applications, which has been developed by the FlexRay Consortium [Fle06]. It is a static time division multiplexing network protocol that supports clock synchronization. In this paper we do not deal with fault tolerance regarding the inter ECU communication.

The hardware of each ECU is clocked by an oscillator with a nominal clock period of say τ_{ref}. For all v the individual clock period τ_v of ECU_v is allowed to deviate from the nominal period by $\delta = 0.15\%$:

$$\mid \tau_v - \tau_{ref} \mid \leq \tau_{ref} \cdot \delta$$

This limitation can be easily achieved by current technology.

With $\Delta = 2\delta/(1-\delta)$ we easily bound for all u and v the relative deviation of individual clock periods among each other by:

$$\mid \tau_v - \tau_u \mid \leq \tau_v \cdot \Delta$$

The assembler programmer sees such a system *mostly* at the ISA level. An ECU configuration $d_v = (d_v.p, d_v.f)$, see Fig. 1, is a pair consisting of a DLX processor configuration $d_v.p$, as defined in [DHP05], and a configuration of a FlexRay-like interface (f-interface) $d_v.f$.

From an interface configuration $d_v.f$ it is easy to define two user-visible buffers: A send buffer $sb(d_v)$ and a receive buffer $rb(d_v)$. Each buffer is capable of holding a message of ℓ bytes.

In the distributed system all communications and computations proceed in rounds r where $r \in \mathbb{N}$. As depicted in Fig. 2 each round is divided into an even[2] number of slots s where $s \in \{0, \ldots, ns-1\}$. The tuple (r, s) refers to slot s in round r.

[2] In Sect. 4.1 we will argue why an even number of slots is required.

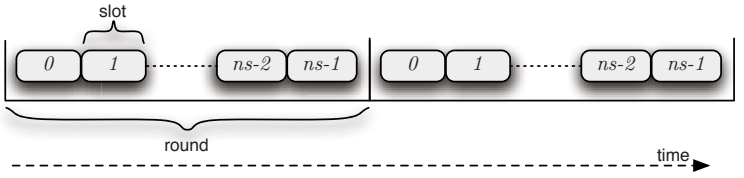

Fig. 2. Slots and Rounds

On each ECU, boundaries between slots are determined by local timer interrupts every T hardware cycles. At the beginning of each round, the local timers are synchronized.

Given a slot (r, s) we define the predecessor $(r, s) - 1$ and the successor $(r, s) + 1$ according to the lexicographical order of slots. We denote by $d_v(r, s)$ the first and by $e_v(r, s)$ the last ISA configuration of ECU_v during slot (r, s).

ECUs communicate according to a fixed schedule that is identical for each round: The function $send$ specifies for all rounds r the ECU that owns the bus during slot (r, s):

$$send : \{0, \ldots, ns - 1\} \rightarrow \{0, \ldots, p - 1\}$$

During slot (r, s) the content of the send buffer of $ECU_{send(s)}$ at the end of the previous round $(r, s) - 1$ is broadcast to the receive buffers of all units ECU_u and becomes visible there at the beginning of the next round $(r, s) + 1$:

$$\forall u, r, s : sb(e_{send(s)}((r, s) - 1)) = rb(d_u((r, s) + 1)).$$

1.3 Results

We present the following results:

1. We describe a gate level design of a FlexRay-like bus interface and elaborate the sketchy correctness proof from [BBG+05] in a distributed hardware model (Theorem 1). To the best of our knowledge this is the first detailed *gate level* correctness proof of an I/O device.
2. An ECU is obtained by integrating the f-interface into the verified VAMP processor [BJK+03, DHP05]. We develop an ISA model for such an ECU. This model is necessarily nondeterministic, because f-interfaces contain timers that interrupt the processor every say T hardware cycles; but cache misses (and hence hardware cycles) are invisible at the ISA level. We then prove the correctness of its hardware implementation (Theorem 2). To the best of our knowledge this is the first hardware correctness proof for a processor *together* with a device capable of generating timer interrupts.
3. Combining the first two results we obtain a correctness proof for the hardware of an entire distributed real-time system (Theorem 3). Again, to the best of our knowledge no such proof has been presented in the literature before.

4. The last result (Theorem 4) is technical. We show how pervasive correctness proofs for local ISA computations with timer interrupts (which are nondeterministic) and the underlying hardware can be obtained from i) conventional correctness proofs for ISA programs that cannot be interrupted ii) hardware correctness theorems and iii) WCET analysis.

2 Overview and Related Work

Consider a situation, where a sending ECU puts a bit on the bus and this bit is sampled into registers of receiving ECUs. Then, due to the clock drift between ECUs, we cannot guarantee that the set up and hold times of the receiving registers are obeyed at all clock edges. This problem occurs whenever computers without a common clock exchange data. It is solved by serial interfaces using a nontrivial protocol. Section 3 deals with the hardware correctness proof of a serial interface as prescribed by the FlexRay standard.

The main arguments have already been published in [BBG$^+$05], so we only summarize the results. We cannot completely argue on the digital levels. Certain lemmas concerning the data transmission on the bus argue about continuous time. Formal proofs for these arguments have already been obtained [Sch06]. Beautiful automatic correctness proofs for abstract versions of protocols for serial interfaces using k-induction are reported in [BP06]. It would be highly desirable to use results of this nature as lemmas in overall correctness proofs for serial interface hardware. However, this would require to formally justify the abstractions being used in [BP06] within a hardware model with set-up and hold times.

In Sect. 4 we deal with f-interfaces that are constructed with the help of serial interfaces. The interfaces have local timers that are synchronized at the start of each round. Using arguments from classical clock synchronization [WL88] we derive conditions on the number of cycles T of each slot, such that for all slots (r, s) the following holds: The send buffer of the sending unit $ECU_{send(s)}$ can be broadcast to the receive buffers of all units in a transmission window, when according to their local timers all ECUs are in slot (r, s). This section provides the crucial arguments of lemmas sketched without proof in [BBG$^+$05]. Detailed hardware constructions and proofs for the results of Sects. 3 and 4 can be found in the lecture notes [Pau05].

The ISA processor configuration is sketched in Sect. 5. Furthermore the semantics of the processor's DLX instruction set are defined by specifying the next state functions of the processor. Details regarding this function are to be found in [HP96, MP00]. Here we focus on the semantics of load / store instructions and on the interrupt mechanism.

In Sect. 6 we introduce an ISA model of a processor together with a f-interface. The next state function of the processor gets two new arguments. One of them is the input sampled by the device on the bus. The second new argument is an oracle input for the ISA computation of the ECU indicating whether a timer interrupt is generated or not. At first sight this looks odd because after all the ECU hardware is completely deterministic. But here we are not looking at the hardware, we are only looking at its ISA model. As pointed out above, in the ISA model, cache hits and misses are not visible. Hence the occurrence of timer interrupts is inherently nondeterministic at the ISA level.

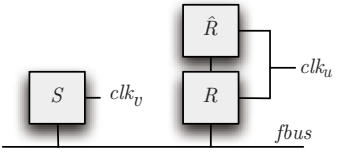

Fig. 3. Serial Interface

In Sect. 7 we show the hardware correctness theorems: The hardware of the entire distributed system simulates the ISA model for a particular choice of the oracle inputs (the latter is specified in Sect. 6.2). We first review the concept of scheduling functions and proof strategies from [SH98, MP00, BJK+03, DHP05]. The scheduling functions enable us to determine the interrupted ISA instructions in a straightforward way. This determines the oracle inputs and thus resolves the nondeterminism present in the pure ISA model.

Section 8 starts with fairly plain computation theory for uninterrupted ISA computations as well as for hardware computations. In particular we formally define the run time and the result of such computations. The definition of run times of hardware computations is again based on the scheduling functions. Then we formally combine the results of WCET analysis, of program correctness proofs for uninterrupted ISA computations and of the hardware correctness proofs into a single result: At the end of slots, the post conditions for memories and registers (but not for program counters) stated for the uninterrupted local ISA computation also hold for their counter parts in the hardware configuration.

3 Serial Interface

In this section we deal with the implementation and the correctness proof of a serial interface as prescribed by the FlexRay standard.

3.1 Hardware Model with Continuous Time

In the standard digital hardware model a computation proceeds in cycles i. The hardware configuration of ECU_v during cycle i of ECU_v is denoted by h_v^i.

Configurations h have components $h.R$ where R is a register content or the content of a memory. Circuits compute signals S from register contents or memory contents. The value of such a signal S is therefore –in well designed hardware– a function $S(h)$ of the hardware configuration.

We denote the clock enable signal, which triggers the update of registers R, by Rce. Then $Rce(h^i)$ is the value of the clock enable of register R in cycle i.

The problems solved by serial interfaces can by their very nature not be treated in the standard digital hardware model with a single digital clock clk. Nevertheless, we can describe each ECU_v in a standard digital hardware model having its own hardware configuration h_v.

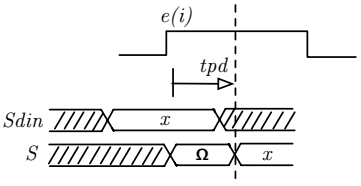

Fig. 4. Sender Register

In order to argue about a 1-bit sender register S of a sending unit ECU_v that is transmitting data via the FlexRay-like bus ($fbus$) to a 1-bit receiver register R of a receiving ECU, as depicted in Fig. 3, we have to extend the digital model.

For the 1-bit registers –and only for these registers– connected to the $fbus$ we extend the hardware model such that we can deal with the concepts of propagation delay (tpd), set-up time (ts), hold time (th) and metastability of registers from hardware data sheets. In the extended model used near the $fbus$ we therefore consider time to be a real valued variable t. The clock edge $e_v(i)$ starting cycle i on ECU_v is defined by

$$e_v(i) = c_v + i \cdot \tau_v \tag{1}$$

for some offset $c_v < \tau_v$. In this continuous time model the content of the sender register S at time t is denoted by $S(t)$.

Now we have enough machinery to define in the continuous time model the output of a sender register S_v on ECU_v during cycle i of ECU_v, i.e. for $t \in (e_v(i), e_v(i+1)]$. If in cycle $i-1$ the digital clock enable $Sce(h_v^{i-1})$ signal was off, we see the old digital value $h_v^{i-1}.S$ of the register during the whole cycle. If the update enable signal was on, then during some propagation delay $tpd < \tau_v - ts$ we cannot predict what we see, which is denoted by Ω. When the tpd has passed, we see the new digital value of the register, which is given by the digital input $Sdin(h_v^{i-1})$ during the previous cycle (see Fig. 4):

$$S_v(t) = \begin{cases} h_v^{i-1}.S & \neg Sce(h_v^{i-1}) \\ \Omega & Sce(h_v^{i-1}) \wedge t \leq e_v(i) + tpd \\ Sdin(h_v^{i-1}) & Sce(h_v^{i-1}) \wedge t > e_v(i) + tpd \end{cases}$$

The $fbus$ is an open collector bus modeled for all time t by:

$$fbus(t) = \bigwedge_v S_v(t)$$

Now consider a receiver register R_u on ECU_u whose clock enable is continuously turned on; thus the register always samples from the $fbus$. In order to define the new digital value $h_u^j.R$ of register R during cycle j on ECU_u we have to consider the value of the $fbus(t)$ in the time interval $(e_u(j) - ts, e_u(j) + th)$, i.e. from the clock edge minus the set-up time until the clock edge plus the hold time. If during that time the $fbus$ has a constant digital value x, the register samples that value:

$$\exists x \in \{0,1\}\ \forall t \in (e_u(j) - ts, e_u(j) + th) : fbus(t) = x \rightarrow h_u^j.R = fbus(e_u(j))$$

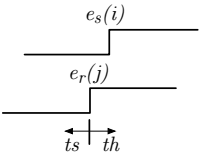

Fig. 5. Clock Edges

Otherwise we define $h_u^j.R = \Omega$. Thus we still have to argue how to deal with unknown values Ω as input to digital hardware. We use the output of register R only as input to a second register $\hat{R}$ whose clock enable is always turned on, too. If Ω is clocked into $\hat{R}$ we assume that $\hat{R}$ has an unknown but digital value:

$$h_u^j.R = \Omega \rightarrow h_u^{j+1}.\hat{R} \in \{0,1\}$$

Indeed, in industrial systems the counterpart of register $\hat{R}$ exists. The probability that R becomes metastable for an entire cycle *and* that this causes $\hat{R}$ to become metastable too is for practical purposes zero. This is exactly what has been formalized above.

Note that the above model uses different but fixed individual clock periods τ_v. There is no problem to extend the model to deal with jitter. Let $\tau_v(i)$ denote the length of cycle i on ECU_v, then we require for all v and i:

$$\tau_v(i) \in [\tau_{ref} \cdot (1-\delta), \tau_{ref} \cdot (1+\delta)]$$

The time $e_v(i)$ of the i-th clock edge on ECU_j is then defined as:

$$e_v(i) = \begin{cases} c_v & i = 0 \\ e_v(i-1) + \tau_v(i-1) & \text{otherwise} \end{cases}$$

This does not complicate the subsequent theory significantly.

3.2 Continuous Time Lemmas for the Bus

Consider a pair of ECUs, where ECU_s is the sender and ECU_r is a receiver in a given slot. Let i be a sender $cycle$ such that $Sce(h_s^{i-1}) = 1$, i.e. the output of S is not guaranteed to stay constant at time $e_s(i)$. This change can only affect the value of register R of ECU_r in cycle j if it occurs before the sampling edge $e_r(j)$ plus the hold time th: $e_s(i) < e_r(j) + th$. Figure 5 shows a situation where due to a hold time violation we have $e_s(i) > e_r(j)$. The first cycle that is possibly being affected is denoted by:

$$cy_{r,s}(i) = \min\{j \mid e_s(i) < e_r(j) + th\}$$

In what follows we assume that all ECUs other than the sender unit ECU_s put the 'idle' value 1 on the bus (hence $fbus(t) = S_s(t)$ for all t under consideration) and we consider only one receiving unit ECU_r. Because the indices r and s are fixed we simply write $cy(i)$ instead of $cy_{r,s}(i)$.

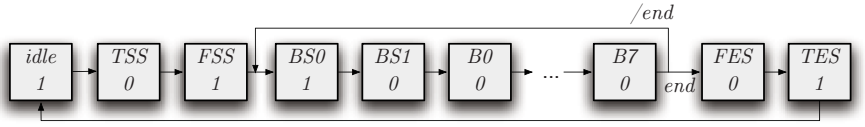

Fig. 6. Frame Encoding

There are two essential lemmas whose proof hinges on the continuous time model. The first lemma considers a situation, where we activate the clock enable Sce of the sender ECU in cycle $i-1$ but not in the following seven cycles[3]. In the digital model we then have $h_s^i.S = \ldots = h_s^{i+7}.S$ and in the continuous time model we observe $x = fbus(t) = S_s(t) = h_s^i.S$ for all $t \in [e_s(i) + tpd, e_s(i+8)]$. We claim that x is correctly sampled in at least six consecutive cycles.

Lemma 1 (Correct Sampling Interval). *Let the clock enable signal of the S register be turned on in cycle $i-1$, i.e. $Sce(h_s^{i-1}) = 1$ and let the same signal be turned off in the next seven cycles, i.e. $Sce(h_s^j) = 0$ for $j \in \{i, \ldots, i+6\}$ then:*

$$h_r^{cy(i)+k}.R = h_s^i.S \quad \textit{for } k \in \{1, \ldots, 6\}$$

The second lemma simply bounds the clock drift. It essentially states that within 300 cycles clocks cannot drift by more than one cycle; this is shown using $\delta \leq 0.15\%$.

Lemma 2 (Bounded Clock Drift). *The clock drift in the interval $m \in \{1, \ldots, 300\}$ is bounded by:*

$$cy(i) + m - 1 \leq cy(i+m) \leq cy(i) + m + 1$$

Detailed proofs of very similar lemmas are to be found in [Pau05, BBG+05, Sch06].

3.3 Serial Interface Construction and Correctness

For natural numbers n and bits y we denote by y^n the string in which bit y is replicated n times, e.g. $0^4 = 0000$. For strings $x[0:k-1]$ consisting of k bits $x[i]$ we denote by $8 \cdot x$ the string obtained by repeating each bit eight times:

$$8 \cdot x = x[0]^8 \cdots x[k-1]^8$$

Our serial interface transmits messages $m[0:\ell-1]$ consisting of ℓ bytes $m[i]$ from a send buffer sb of the sending ECU to a receive buffer rb of the receiving ECU.

The following protocol is used for transmission (see Fig. 6). A frame $f(m)$ is created from a message m by inserting falling edges between the bytes and adding some bits at the start and the end of the frame:

$$f(m) = 0110m[0] \cdots 10m[\ell-1]01$$

[3] This particular interval of 8 cycles is taken from the FlexRay standard [Fle06].

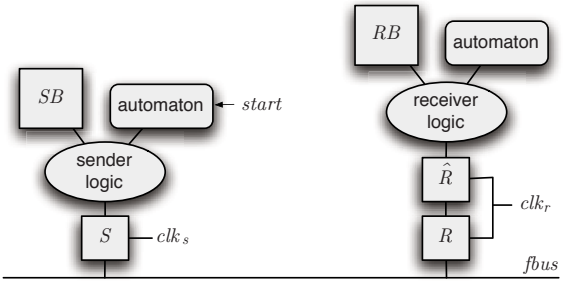

Fig. 7. Send and Receive Buffer

In $f(m)$ we call the first zero the transmission start sequence (TSS), the first one the frame start sequence (FSS), the last zero the frame end sequence (FES) and the last one the transmission end sequence (TES). The two bits producing a falling edge before each byte are called the byte start sequence $(BS0, BS1)$.

The sending ECU broadcasts $8 \cdot f(m)$ over the $fbus$. For each bit of the frame the update-enable signal is on for 1 cycle and then off for 7 cycles. All serial interfaces that are not actively transmitting put by construction the idle value (the bit 1) on the bus.

Figure 7 shows a simplified view of the hardware involved in the transmission of a message. On the sender side, there is an automaton keeping track of which bit of the frame is currently being transmitted. This automaton inserts the additional protocol bits around the message bytes. Hardware for sending each bit eight times and for addressing the send buffer is not shown.

On the receiver side there is the automaton from Fig. 6 trying to keep track of which bit of the frame is currently being transmitted (the automaton on the sender side is very similar). That it does so successfully requires proof.

The bits sampled in register $\hat{R}$ are processed in the following way. The voted bit v is computed by applying a majority vote to the last five sampled bits. These bits are given by the $\hat{R}$ register and a 4-bit shift register as depicted in Fig. 8.

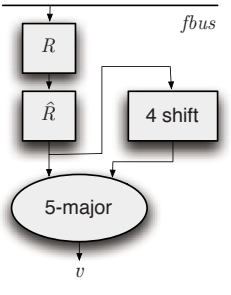

Fig. 8. Receiver Logic

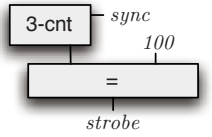

Fig. 9. Strobe Signal

According to Lemma 1 for each bit of the frame a sequence of at least six bits is correctly sampled. The filtering essentially maintains this property. If the receiver succeeds to sample that sequence roughly in the middle, he wins. For this purpose the receiver has a modulo-8 counter trying to keep track of which of the eight identical copies of a frame bit is currently being transmitted. When the counter value equals four the $strobe$ signal is turned on (see Fig. 9). For frame decoding the voted bit is sampled with this $strobe$ signal. The automaton trying to keep track of the protocol is also clocked with the $strobe$ signal.

Clocks are drifting, hence the hardware has to perform a low level synchronization. The counter is reset by a $sync$ signal in two situations: At the beginning of a transmission or at an expected falling edge during the byte start sequence. Abbreviating signals $s(h_r^i)$ with s^i we write:

$$sync^i = (idle^i \vee BS1^i) \wedge (\neg v^i \wedge v^{i-1})$$

The crucial part of the correctness proof is a lemma arguing simultaneously about three statements by induction over the receiver cycles:

1. The state of the automaton keeps track of the transmitted frame bit.
2. The $sync$ signal is activated at the corresponding falling edge of the voted bit v between $BS1$ and $BS0$.
3. Sequences of identical bit are sampled roughly in the middle.

We sketch the proof of this lemma. Statement 1 is clearly true in the $idle$ state. From statement 1 follows that the automaton expects the falling edges of the voted signal exactly when the sender generates them. Thus the counter is well synchronized after these falling edges. This shows statement 2. Immediately after synchronization the receiver samples roughly in the middle. There is a synchronization roughly every 80 sender cycles. By Lemma 2 and because $80 < 300$, the sampling point can wander by at most one bit between activations of the $sync$ signal. This is good enough to stay within the correctly sampled six copies. This shows statement 3. If transmitted frame bits are correctly sampled, then the automaton keeps track of them. This shows statement 1.

Let t_0 be the time (not the cycle) when the $start$ signal of the sender is activated. Let t_1 be the time, when all automata have reached the $idle$ state again and all write accesses to the receive buffer have completed. Let the number of 'transmission cycles' be defined by:

$$tc = 45 + 80 \cdot \ell$$

Intuitively, the product $80 \cdot \ell$ in the definition of tc comes from the fact that each byte produces 10 frame bits and each of these is transmitted 8 times. The four bits added at

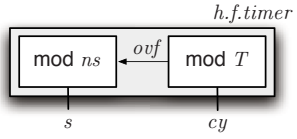

Fig. 10. Hardware Timer

the start and the end of the frame contribute $4 \cdot 8 = 32$. The remaining 13 cycles are caused by delays in the receiver logic, in particular by delay in the shift register before the majority voter. The correctness of message transmission is stated as follows:

Lemma 3 (Correct Message Transfer With Time Bound). *Messages are correctly transmitted, and the transmission does not last longer than tc sender cycles:*

$$rb(t_1) = sb(t_0)$$
$$t_1 - t_0 \leq tc \cdot \tau_s$$

4 FlexRay-Like Interfaces and Clock Synchronization

In this section we outline the implementation and the correctness proof of FlexRay-like interfaces (f-interfaces).

4.1 Hardware Components

Recall that we denote hardware configurations of ECU_v by h_v. If the index v of the ECU does not matter, we drop it. The hardware configuration is split into a processor configuration $h.p$ and an interface configuration $h.f$. In addition to the registers of the serial interface, the essential components of the hardware configuration $h.f$ of our (non fault tolerant) f-interface are

- double buffers $h.f.sb(par)$ and $h.f.rb(par)$, where $par \in \{0, 1\}$, implementing the user-visible send and receive buffers,
- the flipflops of a somewhat non trivial timer $h.f.timer$,
- configuration registers.

The organization of the hardware timer $h.f.timer$ is depicted in Fig. 10. The low-order bits $h.f.timer.cy$ count the cycles of a slot. Unless the timer is synchronized, slots have locally T cycles, thus the low-order bits are part of a modulo-T counter. The high-order bits $h.f.timer.s$ count the slot index s of the current slot (r, s) modulo ns. The timer is initialized with the value $(ns - 1, T - 1)$.

The timers on all ECUs but $ECU_{send(0)}$ stall when reaching the maximum value $(ns-1, T-1)$ and wait for synchronization. The timer on $ECU_{send(0)}$ always continues counting. Details regarding the synchronization mechanism are given in Sect. 4.2.

The overflow signal $ovf(h)$ between the low-order and the high-order bits of the counter can essentially serve as the timer interrupt signal $ti(h)$ generated by the interface hardware[4]:

$$ti(h^i) = ovf(h^i) \wedge \neg ovf(h^{i-1})$$

The low-order bit of the slot counter keeps track of the parity of the current slot and is called the hardware parity signal:

$$par(h) = h.f.timer.s[0]$$

In general the *fbus* side of the interface sees the two buffers $h.f.sb(par(h))$ and $h.f.rb(par(h))$. Messages are always transmitted between these buffers. The processor on the other hand writes to $h.f.sb(\neg par(h))$ and reads from $h.f.rb(\neg par(h))$. This does not work at boundaries of rounds unless the number of slots ns is even.

The configuration registers are written immediately after reset / power-up. They contain in particular the locally relevant portions of the scheduling function. Thus if ECU_v is (locally) in a slot with slot index s and $send(s) = v$ then ECU_v transmits the content of the send buffer $h.f.sb(par(h))$ via the *fbus* during some transmission interval $[ts(r,s), te(r,s)]$.

If we can guarantee that during the transmission interval *all* ECUs are locally in slot (r,s), then transmission is successful by Lemma 3. The clock synchronization algorithm together with an appropriate choice of the transmission interval guarantees exactly that.

4.2 Clock Synchronization

The idea of clock synchronization is easily explained: Imagine one slot is one hour and one round is one day. Assume different clocks drift by up to $drift = 5$ minutes per day. ECUs synchronize to the first bit of the message transmission due between midnight and 1 o'clock. Assume adjusting the clocks at the receiving ECUs takes up to $adj = 1$ minute. Then the maximal deviation during 1 day is $off = drift + adj = 6$ minutes. $ECU_{send(s)}$, which is the sender in hour s, is on the safe side if it starts transmitting from s o'clock plus off minutes until off minutes before $s+1$ o'clock, i.e. somewhen in between $s:06$ o'clock and $s+1:54$ o'clock.

At midnight life becomes slightly tricky: $ECU_{send(0)}$ waits until it can be sure that everybody believes that midnight is over and hence nobody is transmitting, i.e. until its local time $0:06$. Then it starts sending. All other ECUs are waiting for the broadcast message and adjust their clocks to midnight + $off = 0:06$ once they detect the first falling bit. Since that might take the receiving ECUs up to 1 minute it might be $0:07$ o'clock on the sender when it is $0:06$ o'clock at the receiver; thus after synchronization the clocks differ by at most $adj = 1$ minute.

We formalize this idea in the following way: Assume without loss of generality that $send(0) = 0$. All ECUs but ECU_0 synchronize to the transmission start sequence (TSS) of the first message of ECU_0. When ECU's waiting for synchronization

[4] In general one needs to keep an interrupt signal active until it is cleared by software; the extra hardware is simple.

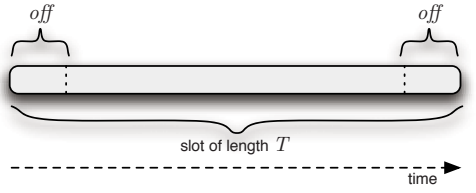

Fig. 11. Slots

($d.f.timer = (ns - 1, T - 1)$) receive this TSS, they advance their local slot counter to 0 and their cycle counter to *off*. Analysis of the algorithm implies that for all $v \neq 0$, ECU_v is waiting for synchronization, when ECU_0 starts message transmission in any slot $(r, 0)$.

First we define the start times $\alpha_v(r, s)$ of slot (r, s) on ECU_v. This is the start time of the first cycle t in round r when the timer in the previous cycle had the value:

$$h^{t-1}.f.timer = ((s - 1 \bmod ns), T - 1)$$

These are the cycles immediately after the local timer interrupts. For every round r, we also define the cycles $\beta_v(r)$ when the synchronization is completed on ECU_v. Formally this is defined as the first cycle $\beta > \alpha_v(r, 0)$ such that the local timer has value:

$$h^{\beta}.f.timer = (0, \mathit{off})$$

Timing analysis of the synchronization process in the complete hardware design shows that for all v and y adjustment of the local timer of ECU_v to value $(0, \mathit{off})$ is completed within an adjustment time $ad = 15 \cdot \tau_y$ after $\alpha_0(r, 0)$:

$$\begin{aligned} \beta_0(r) &= \alpha_0(r, 0) + \mathit{off} \cdot \tau_0 \\ \beta_v(r) &\leq \beta_0(r) + 15 \cdot \tau_y \end{aligned}$$

For $s \geq 1$ no synchronization takes place and the start of new slots is only determined by the progress of the local timer:

$$\alpha_v(r, s) = \begin{cases} \beta_v(r) + (T - \mathit{off}) \cdot \tau_v & s = 1 \\ \alpha_v(r, s - 1) + T \cdot \tau_v & s > 1 \end{cases}$$

ECU_0 synchronizes the other ECUs. Thus the start of slot $(r, 0)$ on ECU_0 depends only on the progress of the local counter:

$$\alpha_0(r, 0) = \alpha_0(r - 1, ns - 1) + T \cdot \tau_0$$

An easy induction on s bounds the difference between start times of the same slot on different $ECUs$:

$$\begin{aligned} \alpha_x(r, s) - \alpha_v(r, s) &\leq 15 \cdot \tau_v + (s \cdot T - \mathit{off}) \cdot (\tau_x - \tau_v) \\ &\leq 15 \cdot \tau_v + (ns \cdot T \cdot \Delta \cdot \tau_v) \\ &= \tau_v \cdot (15 + (ns \cdot T \cdot \Delta)) \\ &= \tau_v \cdot \mathit{off} \end{aligned} \tag{2}$$

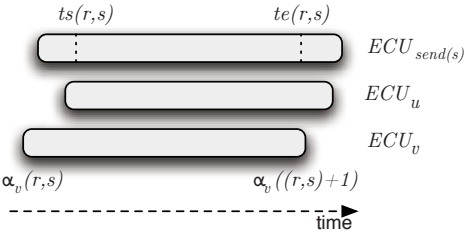

Fig. 12. Schedules

Thus we have $\mathit{off} = ad + \mathit{drift}$ with $ad = 15$ and $\mathit{drift} = ns \cdot T \cdot \Delta$.

Transmission is started in slots (r, s) by $ECU_{send(s)}$ when the local cycle count is off. Thus the transmission start time is:

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

By Lemma 3 the transmission ends at time:

$$\begin{aligned} te(r, s) &= ts(r, s) + tc \cdot \tau_{send(s)} \\ &= \alpha_{send(s)}(r, s) + (\mathit{off} + tc) \cdot \tau_{send(s)} \end{aligned}$$

The transmission interval $[ts(r, s), te(r, s)]$ must be contained in the time interval, when all ECUs are in slot (r, s), as depicted in Fig. 12.

Lemma 4 (No Bus Contention). *For all indices v and u of ECUs:*

$$\begin{aligned} \alpha_v(r, s) &\leq ts(r, s) \\ te(r, s) &\leq \alpha_u((r, s) + 1) \end{aligned}$$

Proof. The first inequality holds because of (2). Let $x = send(s)$:

$$\begin{aligned} \alpha_v(r, s) &\leq \alpha_x(r, s) + \tau_x \cdot \mathit{off} \\ &= ts(r, s) \end{aligned}$$

The second inequality determines the minimal size of T:

$$\begin{aligned} te(r, s) &\leq \alpha_x(r, s) + (\mathit{off} + tc) \cdot \tau_x \\ &\leq \alpha_u(r, s) + \mathit{off} \cdot \tau_u + (\mathit{off} + tc) \cdot (1 + \Delta) \cdot \tau_u \\ &\leq \alpha_u((r, s) + 1) \\ &= \alpha_u(r, s) + T \cdot \tau_u \end{aligned}$$

Further calculations are necessary at the borders between rounds. Details can be found in [Pau05]. □

From the local start times of slots $\alpha_v(r, s)$ we calculate the numbers of local start cycles $t_v(r, s)$ using (1)

$$\alpha_v(r, s) = c_v + t_v(r, s) \cdot \tau_v$$

and then solving for $t_v(r,s)$. Trivially the number $u_v(r,s)$ of the locally last cycle on ECU_v is:

$$u_v(r,s) = t_v((r,s)+1) - 1$$

Consider slot (r,s). Lemma 3 and Lemma 4 then imply that the value of the send buffer of $ECU_{send(s)}$ on the network side ($par = s \bmod 2$) at the start of slot (r,s) is copied to all receive buffers on the network side by the end of that slot.

Theorem 1 (Message Transfer With Cycles). *Let $x = send(s)$. Then for all v:*

$$h_x^{t_x(r,s)}.f.sb(s \bmod 2) = h_v^{u_v(r,s)}.f.rb(s \bmod 2)$$

This theorem talks only about digital hardware and hardware cycles. Thus we have shown the correctness of data transmission via the bus *and* we are back in the digital world.

5 Specifying an Instruction Set Architecture

In this section we sketch the DLX instruction set architecture (ISA).

5.1 Configurations and Auxiliary Concepts

Processor configurations d have the following components:

- $d.R \in \{0,1\}^{32}$: The current value of register R. For this paper, the relevant registers are: The program counter pc, the delayed PC dpc (which is used to specify the delayed branch mechanism detailed in [MP00]), the general purpose registers $gpr[x]$ with $x \in \{0,1\}^5$, the status register sr (it contains the mask bits for the interrupts) as well as a exception cause register eca (to be explained later on).
- The byte addressable memory $d.m : \{0,1\}^{32} \to \{0,1\}^8$. The content of the memory at byte address a is given by $d.m(a)$.

For addresses a, memories m, and natural numbers x we denote by $m_x(a)$ the concatenation of the memory bytes from address a to address $a + x - 1$ in little-endian order:

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

The instruction executed in configuration d is the memory word addressed by the delayed PC:

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

The six high-order bits of the instruction word constitute the opcode:

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

Instruction decoding can easily be formalized by predicates on $I(d)$. In some cases it suffices to inspect the opcode only. The current instruction is for instance a 'load word' (lw) instruction if the opcode (opc) equals 100011:

$$lw(d) \Leftrightarrow opc(d) = 100011$$

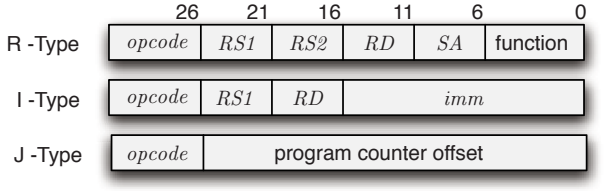

Fig. 13. Instruction Types

DLX instructions come in three instruction types as shown in Fig. 13. The type of an instruction defines how the bits of the instruction outside the opcode are interpreted. The occurrence of an register-type (R-type) instruction, e.g. a add or a subtract instruction, is for instance specified by:

$$rtype(d) \Leftrightarrow opc(d) = 000000$$

Definitions of immediate-constant-type (I-type) instructions and jump-type (J-type) instructions are slightly more complex.

Depending on the instruction type, certain fields have different positions within the instruction. For the register 'destination' operand (RD) we have for instance:

$$RD(d) = \begin{cases} I(d)[20:16] & itype(d) \\ I(d)[15:11] & \text{otherwise} \end{cases}$$

The effective address (ea) of load / store operations is computed as the sum of the content of the register addressed by the $RS1$ field $d.gpr(RS1(d))$ and the immediate field $imm(d) = I(d)[15:0]$. The addition is performed modulo 2^{32} using two's complement arithmetic. Formally, the sign extension of the immediate constant is defined by:

$$sxt(imm(d)) = imm(d)[15]^{16} imm(d)$$

This turns the immediate constant into a 32-bit constant while preserving the value as a two's complement number. It is like adding leading zeros to a natural number. Denoting ordinary binary addition modulo 2^{32} by $+_{32}$ we define:

$$ea(d) = d.gpr(RS1(d)) +_{32} sxt(imm(d))$$

This works because n bit two's complement numbers and n bit binary numbers have the same value modulo 2^n. For details see e.g. Sect. 2 of [MP00].

5.2 Basic Instruction Set

With the above few preliminary definitions in place we easily specify the next configuration d', i.e. the configuration after execution of $I(d)$. This obviously formalizes the instruction set. In the definition of d' we split cases depending on the instruction to be executed. As an example we specify the next configuration for a load word instruction.

The main effect of a load word instruction is that the general purpose register addressed by the RD field is updated with the memory word addressed by the effective address ea:

$$d'.gpr(RD(d)) = d.m_4(ea(d))$$

The PC is incremented by four in 32-bit binary arithmetic and the old PC is copied into the delayed PC:

$$\begin{aligned} d'.pc &= d.pc +_{32} 0^{30}10 \\ d'.dpc &= d.pc \end{aligned}$$

This part of the definition is identical for all instructions except control instructions. One also must specify what is not changed:

$$\begin{aligned} d'.m &= d.m \\ d'.gpr(x) &= d.gpr(x) \quad \text{for} \quad x \neq RD(d) \\ d'.sr &= d.sr \\ d'.eca &= d.eca \end{aligned}$$

The main effect of store word instructions is that the general purpose register content addressed by RD is copied into the memory word addressed by ea:

$$d'.m_4(ea(d)) = d.gpr(RD(d))$$

Completing this definition for all instructions, we get the definition of a DLX next state function:

$$d' = \delta_D(d)$$

5.3 Interrupts

Interrupts are triggered by interrupt event signals that might be internally generated (like illegal instruction, misalignment, or overflow) or externally generated (like reset and timer interrupt). Interrupts are numbered with indices $j \in \{0, \ldots, 31\}$. We classify the set of these indices in two ways:

1. maskable / not maskable. The set of indices of maskable interrupts is denoted by M.
2. external / internal. The set of indices of external interrupts is called E.

We denote external event signals by $eev[j]$ with $j \in E$ and we denote internal event signals by $iev[j]$ with $j \notin E$. We gather the external event signals into a vector eev and the internal event signals into a vector iev.

Formally these signals must be treated in a very different way. Whether an internal event signal $iev[j]$ is activated in configuration d is determined only by the configuration. For instance if we use $j = 1$ for the illegal instruction interrupt and $LI \subset \{0,1\}^{32}$ is the set of bit patterns for which d' is defined if $I(d) \in LI$, then:

$$iev(d)[1] \Leftrightarrow I(d) \notin LI$$

Thus the vector of internal event signals is a function $iev(d)$ of the current processor configuration d. In contrast, external interrupts are external inputs for the next state function. We therefore get a new next state function:

$$d' = \delta_D(d, eev)$$

The cause vector ca of all event signals is a function of the processor configuration d and the external input eev:

$$ca(d, eev)[j] = \begin{cases} eev[j] & j \in E \\ iev(d)[j] & \text{otherwise} \end{cases}$$

The masked cause vector mca is computed from ca with the help of the interrupt mask stored in the status register: If interrupt j is maskable and $sr[j] = 0$, it is masked out:

$$mca(d, eev)[j] = \begin{cases} ca(d, eev)[j] \wedge d.sr[j] & j \in M \\ ca(d, eev)[j] & \text{otherwise} \end{cases}$$

If any one of the masked cause bits is on, the jump to interrupt service routine ($JISR$) bit is turned on:

$$JISR(d, eev) = \bigvee_j mca(d, eev)[j]$$

If this occurs, many things happen. We mention only a few: The PCs are forced to point to the start addresses of the interrupt service routine (ISR). We assume it starts at the (binary) address 0:

$$d'.dpc = 0^{32}$$
$$d'.pc = 0^{30}10$$

All maskable interrupts are masked:

$$d'.sr = 0^{32}$$

The masked cause register is saved into the exception cause register:

$$d'.eca = mca(d, eev)$$

For a complete definition see Chap. 5 of [MP00].

6 ISA of Processors with f-Interfaces

In this section we integrate our f-interface into the ISA model of the processor.

6.1 I/O Ports and Message Buffers

As already mentioned earlier an ISA configuration d of a processor with an f-interface is a pair $(d.p, d.f)$, where $d.p$ is a processor configuration as described in the previous section. It has registers $d.p.R$ and a memory $d.p.m$. The range of the function m is however restricted to a subset $A \subset \{0,1\}^{32}$ of the entire address range:

$$d.p.m : A \to \{0,1\}^8$$

Identifying bit strings a with their value if they are interpreted as a binary number, we define A to be in the range of addresses below a certain address D:

$$A = \{a \mid a < D\}$$

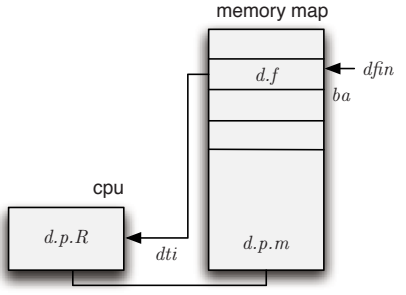

Fig. 14. Memory Mapped IO

Addresses from address D on are called I/O ports. They are reserved for I/O devices. Every device dv is assigned a base address $ba(dv)$ in the range of I/O ports (see Fig. 14):

$$ba(dv) \in \{D, \ldots, 2^{32} - 1\}$$

Here we only consider a single device and a single base address ba.

When a processor accesses a device with K I/O ports, then for $k = \lceil \log K \rceil$ the device configuration (here $d.f$) must contain a memory:

$$d.f.m : \{0,1\}^k \to \{0,1\}^8$$

In our case the memory of the device contains the send buffer, the receive buffer –each with ℓ bytes where ℓ is a multiple of 4– and say c configuration registers. Thus

$$K = 2 \cdot \ell + 4 \cdot c$$

We use the first ℓ bytes of this memory for the send buffer, the next ℓ bytes for the receive buffer and the remaining bytes for the configuration registers. We formalize this by defining for all indices of message bytes $y \in \{0, \ldots, \ell - 1\}$:

$$sb(d)(y) = d.f.m(y)$$
$$rb(d)(y) = d.f.m(\ell + y)$$

The semantics of accesses of the processor to the I/O ports are simply defined by a slight change of the semantics of lw and sw instructions. If the effective address[5] lies in the address range assigned to the device in the memory map, i.e. if

$$ea(d.p) = ba + x \quad \text{with} \quad 0 \leq x \leq K - 4$$

the essential effect of a load word instruction is

$$d'.p.gpr(RD(d.p)) = d.f.m_4(x)$$

and the essential effect of a store word instruction is:

$$d'.f.m_4(x) = d.p.gpr(RD(d.p))$$

[5] We require the effective address be word aligned.

6.2 Timer Interrupt and I/O

So far we have covered a single processor with a device but we have not considered timer interrupts. The consequences for integrating those in the processor construction and processor correctness proofs have already been outlined (e.g. for a hard-disk) in [HIP05]. In the remainder of this section we extend these results with timer interrupts generated by an f-interface.

As pointed out earlier, at the ISA level the timer interrupt must be treated as an oracle input dti. Furthermore we have to deal with external data input $dfin$ from the f-interface. Thus –ignoring reset– the next state function for the device has on the ISA level the format:

$$d' = \delta_D(d, dti, dfin)$$

If we denote by dti^i and $dfin^i$ the oracle input and the input from the $fbus$ for the i-th executed instruction, then we get computations $d^0, d^1, \ldots$ by defining (straight from the automata theory textbooks):

$$d^{i+1} = \delta_D(d^i, dti^i, dfin^i)$$

In our distributed system we have configurations d_v from many ECUs. Within this programming model we now introduce names, e.g. $j_v(r,s)$, for certain indices of local *instructions* on ECU_v.

Intuitively, the timer interrupts the instruction executed in the local configuration $d_v^{j_v(r,s)}$ of ECU_v, and this locally ends slot (r,s).

Based on these indices we can define some more useful concepts purely within the ISA model:

- $i_v(r,s) = j_v((r,s)-1)+1$: The index of the first local instruction in slot (r,s)
- $d_v(r,s) = d_v^{i_v(r,s)}$: The first local ISA configuration in slot (r,s)
- $e_v(r,s) = d_v^{j_v(r,s)}$: The last local ISA configuration in slot (r,s)

We can even define the sequence $dti(r,s)$ of oracle timer inputs dti^i where $i \in \{i_v(r,s), \ldots, j_v(r,s)\}$. It has the form

$$dti(r,s) = 1^a 0^b 1$$

where the timer interrupt is cleared by software instruction $i_v(r,s)+a-1$ and $a+b+1 = j_v(r,s) - i_v(r,s) + 1$ is the number of local instructions in slot (r,s).

Indeed we can complete, without any effort, the entire ISA programming model. The effect of an interrupt on the processor configuration has been defined in the previous section, thus we get for instance:

$$d_v(r,s).dpc = 0^{32}$$
$$d_v(r,s).pc = 0^{30}10$$

Also for the transition from $e_v(r,s)$ to $d_v((r,s)+1)$ and only for this transition we use the external input:

$$dfin^{j_v(r,s)} \in \{0,1\}^{8\cdot\ell}$$

Thus we assume that it consists of an entire message and we copy that message into the user-visible receive buffer:

$$rb(d_v((r,s)+1)) = dfin^{j_v(r,s)}$$

Of course we also know what this message is supposed to be: The content of the user-visible send buffer of $ECU_{send(s)}$ at the end of slot $(r,s)-1$:

$$dfin^{j_v(r,s)} = sb(e_{send(s)}((r,s)-1)$$

Thus

$$rb(d_v((r,s)+1)) = sb(e_{send(s)}((r,s)-1)) \tag{3}$$

This completes the user-visible ISA model. And with Theorem 1 we essentially already completed the hardware correctness proof of the implementation of (3). The nondeterminism is completely encapsulated in the numbers $j_v(r,s)$ as it should be, at least if the local computations are fast enough. All we need to do is to justify the model by a hardware correctness theorem and to identify the conditions under which it can be used.

7 Hardware Correctness

In this section we outline a hardware correctness proof that establishes a relationship between an ISA configuration and a hardware configuration.

7.1 Scheduling Functions

The processor correctness proofs considered here hinge on the concept of scheduling functions s. The hardware of pipelined processors consists of many stages k, e.g. fetch stage, issue stage, reservation stations, reorder buffer, write back stage, etc. (see Fig. 17). Stages can be full or empty due to pipeline bubbles. The hardware keeps track of this with the help of full bits $full_k$ for each stage as defined in [MP00]. Recall that $full_k(h^t)$ is the value of the full bit in cycle t. We use the shorthand $full_k^t$. Note that the fetch state is always full, i.e. $\forall t : full_0^t = 1$.

For hardware cycles t and stages k that are full during cycle t, i.e. such that $full_k^t$ holds, the value $s(k,t)$ of the scheduling function is the index i of the instruction that is in stage k during cycle t. If the stage is not full, it is the index of the instruction that was in stage k in the last cycle before t when the stage was full. Initially $s(0,0) = 0$ holds.

The formal definition of scheduling functions uses an extremely simple idea: Imagine that the hardware has registers that can hold integers of arbitrary size. Augment each stage with such a register and store in it the index of the instruction currently being executed in that stage. These indices are computed exactly as the tags in a Tomasulo scheduler. The only difference is that they have unbounded size because we want to count up to arbitrarily large indices. In real hardware this is not possible and not necessary. In an abstract mathematical model there is no problem to do this.

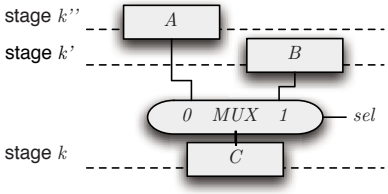

Fig. 15. Scheduling Functions

Each stage k of the processors under consideration has an update enable signal ue_k. Stage k gets new data in cycle t if the update enable signal ue_k was on in cycle $t-1$. We fetch instructions in order and hence define for the instruction fetch stage IF:

$$s(IF,t) = \begin{cases} s(IF,t-1)+1 & ue_{IF}^{t-1} \\ s(IF,t-1) & \text{otherwise} \end{cases}$$

In general, a stage k can get data belonging to a new instruction from one or more stages k'. Examples where more than one predecessor stage k' exists for a stage k are i) cycles in the data path of a floating point unit performing iterative division or ii) the producer registers feeding on the common data bus of a Tomasulo scheduler. In this situation we must define for each stage k a predicate $trans(k',k,t)$ indicating that in cycle t data are transmitted from stage k' to stage k. In the example of Fig. 15 we use the select signal sel of the multiplexer and define:

$$trans(k',k,t) = ue_k^t \wedge sel^t$$

If $trans(k',k,t-1)$ holds for some k', then we set $s(k,t) = s(k',t-1)$ for that k'. Otherwise $s(k,t) = s(k,t-1)$.

7.2 Simple Simulation Relations

For ECUs we first consider a 'naive' simulation relation $sim(d,h)$ between ISA configurations d and hardware configurations h. We require that the user-visible registers R have identical values:

$$h.p.R = d.p.R$$

Furthermore we require that the send and receive buffers on the processor side (indexed in the hardware by $\neg par(h)$) of the hardware have the same value as the user-visible buffers. Thus, we require for all indices $y \in \{0,\ldots,\ell-1\}$ of message bytes:

$$h.f.sb(\neg par(h))(y) = sb(d)(y)$$
$$h.f.rb(\neg par(h))(y) = rb(d)(y)$$

For the addresses a in the processor we would like to make a similar definition, but this does not work, because the user-visible processor memory is simulated in the hardware by a memory system consisting e.g. of an instruction cache $icache$, a data

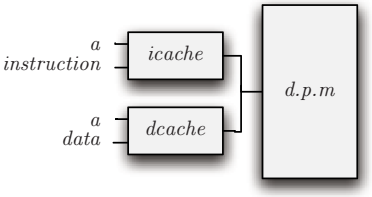

Fig. 16. Memory System

cache $dcache$ and a user main memory $mainm$. Thus there is a quite nontrivial function $m(h.p) : A \to \{0,1\}^8$ specifying the memory simulated by the memory system. We can define this functions in the following way: Imagine you apply in configuration h at the memory interface (either at the $icache$ or at the $dcache$) address a. Considering a hit in the instruction cache, i.e. $ihit(h.p, a) = 1$, the $icache$ would return $icache(h.p, a)$. Similarly, considering a hit in the data cache $dhit(h.p, a) = 1$ the $dcache$ would return $dcache(h.p, a)$. Then we can define[6]:

$$m(h.p)(a) = \begin{cases} icache(h.p, a) & ihit(h.p, a) \\ dcache(h.p, a) & dhit(h.p, a) \\ h.p.mainm(a) & \text{otherwise} \end{cases}$$

Using this definition we can require in the simulation relation for all addresses not being I/O ports, i.e. $a \in A$:

$$m(h.p)(a) = d.p.m(a)$$

In a pipelined machine this simulation relation almost never holds, because in one cycle different hardware stages k usually hold data from different ISA configurations; after all this is the very idea of pipelining. There is however an important exception: When the pipe is drained, i.e. all hardware stages except the instruction fetch stage are empty:

$$drained(h) \Leftrightarrow \forall k : k \neq IF \to full_k^t = 0$$

This happens to be the case after interrupts, in particular initially after reset and at the boundaries between slots when a timer interrupt is being generated.

7.3 Processor Correctness Theorem

Figure 17 shows in simplified form the stages of a processor with out of order processing and a Tomasulo scheduler.

Each user-visible register $d.R$ of the processor has a counter part $h.R$ belonging to the stage in the hardware specified by $stage(R)$. If the processor would have only

[6] In the processors under consideration the caches snoop on each other; data of address a is only in at most one cache [Bey05, BJK+03].

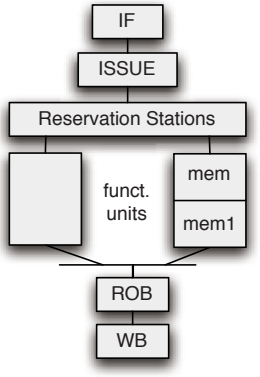

Fig. 17. Processor Pipeline

registers R and no memory, we could show by induction over t for all cycles t and stages k:

If $k = stage(R)$, then the value $h^t.p.R$ of the hardware register R in cycle t is the value $d^{s(k,t)}.p.R$ of the ISA register R for the instruction scheduled in stage k in cycle t:

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

For the memory we have to consider the memory unit of the processor consisting of two stages mem and $mem1$. Stage mem contains hardware for the computation of the effective address. The memory $m(h^t.p)$ that is simulated by the memory hierarchy of the hardware in cycle t, is identical with the ISA memory $d^{s(mem1,t)}.p.m$ for the instruction scheduled in stage $mem1$ in cycle t:

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

In the hardware the send and receive buffers are 'parallel' to the memory system, so we can reuse the scheduling functions. For the copy of the buffers on the processors side we get:

$$h^t.f.sb(\neg par(h^t)) = sb(d^{s(mem1,t)})$$
$$h^t.f.rb(\neg par(h^t)) = rb(d^{s(mem1,t)})$$

We summarize this by stating for all ECUs the correctness statement for the processor and the processor side of the interface for slot (r, s). It is proven by induction over the cycles of the slot. Recall from Sect. 4.2 that we know already the start *cycles* $t_v(r, s)$ for all ECUs. The statement of the theorem is identical for all ECU_v. Thus we drop the subscript v.

The theorem assumes that at the start of a slot the pipe is drained (e.g. by the timer interrupt that ended the previous slot) and that the simulation relation holds between the first hardware configuration $h(r, s) = h^{t(r,s)}$ and the first ISA configuration $d(r, s) = d^{i(r,s)}$ of the slot.

Theorem 2 (Hardware Correctness for One Slot). *Assume that* $drained(h(r,s))$ *and* $sim(d(r,s),h(r,s))$ *holds. Then for all* $t \in \{t(r,s),\ldots,t((r,s)+1)-1\}$*, for all stages* k *and for all registers* R *with* $stage(R)=k$*:*

$$\begin{aligned} h^t.p.R &= d^{s(k,t)}.p.R \\ m(h^t.p) &= d^{s(mem1,t)}.p.m \\ h^t.f.sb(\neg par(h^t)) &= sb(d^{s(mem1,t)}) \\ h^t.f.rb(\neg par(h^t)) &= rb(d^{s(mem1,t)}) \end{aligned}$$

The theorem is proven by induction over the cycles of the slot. Using the above theorem we can show:

Theorem 3 (Hardware Correctness for System)

$$\forall (r,s), v : drained(h_v(r,s)) \wedge sim(d_v(r,s),h_v(r,s))$$

Theorem 3 is proven by induction over the slots (r,s) using additional assumptions about registers not visible at the ISA level. In order to argue about the boundaries between two slots Theorem 2 and Lemma 4 must be applied on the last cycle of the previous slot.

7.4 The Interrupted Instruction

To support precise interrupts the cause signals of internal as well as of external interrupts are sampled in the write back stage (WB). The instruction interrupted by an active cause signal in cycle t is therefore the instruction scheduled in the writeback stage during cycle t. Thus the index $j(r,s)$ of the interrupted instruction, which resolves the nondeterminism and makes the proof work, is:

$$j(r,s) = s(WB, t(r,s))$$

For detailed processor correctness proofs dealing with sequences of internal and external interrupts (but without devices) see [Bey05, Dal06].

8 Pervasive Correctness Proofs

Finally, we show how pervasive correctness proofs for computations with timer interrupts can be obtained from i) correctness proofs for ISA programs that cannot be interrupted ii) hardware correctness theorems and iii) WCET analysis. As one would expect, the arguments are reasonably simple, but the entire formalism of the last sections is needed in order to formulate them.

We consider only programs of the form[7]:

```
{P;   a : jump a;   a+4 : NOP }
```

The program does the useful work in portion P and then waits in the idle loop for the timer interrupt. P initially has to clear and then to unmask the timer interrupt, which is masked when P is started (see Sect. 6.2).

[7] Note that we have a byte addressable memory and that in an ISA with delayed branch the idle loop has two instructions.

8.1 Computation Theory

We have to distinguish carefully between the transition function $\delta_D(d, dti, fdin)$ of the interruptible ISA computation and the transition function $\delta_U(d)$ of the non interruptible ISA computation, which we define as follows:

$$\delta_U(d) = \delta_D(d, 0, *)$$

Observe that this definition permits the non interruptible computation to clear the timer interrupt bit by software. Non interruptible computations starting from configuration d are obtained by iterated application of δ_U:

$$\delta_U^i(d) = \begin{cases} d & i = 0 \\ \delta_U(\delta_U^{i-1}(d)) & \text{otherwise} \end{cases}$$

For the ISA computation

$$d(r,s) = d^{i(r,s)}, d^{i(r,s)+1}, \ldots, d^{j(r,s)} = e(r,s)$$

that has been constructed in Theorem 2 we get:

Lemma 5. *For all instructions in a given slot, i.e.* $t \in [0 : (j(r,s) - i(r,s))]$*:*

$$d^{i(r,s)+t} = \delta_U^t(d(r,s))$$

This lemma holds due to the definition of $j(r,s)$ and the fact that the timer is masked initially such that the instructions of the interruptible computation are not interrupted.

We define the ISA run time $T_U(d,a)$, i.e. the time until the idle loop is reached, simply as the smallest i such that δ_U^i fetches an instruction from address a:

$$T_U(d,a) = \min\{i \mid \delta_U^i(d).p.dpc = a\}$$

Furthermore we define the result of the non interruptible ISA computation as:

$$res_U(d,a) = \delta_U^{T_U(d,a)}(d)$$

Correctness proofs for non interruptible computations can be obtained by classical program correctness proofs. They usually have the form $d \in E \rightarrow res_U(d,a) \in Q$ or, written as a Hoare triple $\{E\}P\{Q\}$.

We assume that the definition of Q does not involve the PC and the delayed PC. Because the idle loop only changes the PC and the delayed PC of the ISA computation we can infer on the ISA level that property Q continues to hold while we execute the idle loop:

$$\forall i \geq T_U(d,a) : \delta_U^i(d) \in Q$$

8.2 Pervasive Correctness

Assume $sim(d,h)$ holds. Then the ISA configuration d can be decoded from the hardware configuration by a function:

$$d = decode(h)$$

Clearly, in order to apply the correctness statement $\{E\}P\{Q\}$ to a local computation in slot (r, s), we have to show for the first ISA configuration in the slot:

$$d(r, s) \in E$$

In order to apply the processor correctness theorem the simulation relation must hold initially:

$$sim(d(r, s), h(r, s))$$

Now consider the last hardware configuration $g(r, s) = h^{t((r,s)+1)-1}$ of the slot. We want to conclude

Theorem 4. *The decoded configuration obeys the postcondition Q:*

$$decode(g(r, s)) \in Q$$

This only works if portion P is executed fast enough on the pipelined processor hardware.

8.3 Worst-Case Execution Time

We consider the set $H(E)$ of all hardware configurations h encoding an ISA configuration $d \in E$:

$$H(E) = \{h \mid decode(h) \in E\}$$

While the decoding is unique, the encoding is definitely not. Portions of the ISA memory can be kept in the caches in various ways.

For a hardware configuration $h = h^0$ we define the hardware run time $T_H(h, a)$ until a fetch from address a as the smallest number of cycles such that in cycle t an instruction, which has been fetched in an earlier cycle $t' < t$ from address a, is in the write back stage WB. Using scheduling functions this definition is formalized as:

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

Thus for ISA configurations satisfying E we define the worst-case execution time $WCET(E, a)$ as the largest hardware runtime $T_H(h, a)$ of a hardware configuration encoding a configuration in E:

$$WCET(E, a) = \max\{T_H(h, a) \mid h \in H(E)\}$$

As pointed out earlier such estimates can be obtained from (sound!) industrial tools based on the concept of abstract interpretation [Abs06]. AbsInt's WCET analyzer does not calculate the "real" worst-case execution time $WCET(E, a)$, but an upper bound $WCET'(E, a) \geq WCET(E, a)$. Nevertheless this is sufficient for correctness since $WCET'(E, a) \leq T - \mathit{off} \Rightarrow WCET(E, a) \leq T - \mathit{off}$. Assume we have:

$$WCET(E, a) \leq T - \mathit{off}$$

Within slot (r, s) we look at the ISA configuration $d(r, s) = d^{i(r,s)}$ and a local computation starting in hardware configuration $h(r, s) = h^{t(r,s)}$. Considering the computation

after hardware run time many cycles $T_H(h(r,s),a) < T - \mathit{off}$ we can conclude that the computation is not interrupted and the instruction in the write back stage (at the end of the computation) is the first instruction being fetched from a. By the definition of the ISA run time this is exactly instruction $i(r,s) + T_U(d(r,s),a)$, thus:

$$s(\mathit{WB}, t(r,s) + T_H(h(r,s),a)) = i(r,s) + T_U(d(r,s),a)$$

Let $h' = h^{t(r,s)+T_H(h(r,s),a)}$ be the hardware configuration in this cycle and let $d' = d^{i(r,s)+T_U(d(r,s),a)} = res_U(d(r,s),a)$ be the ISA configuration of the instruction in the write back stage.

In this situation the pipe is almost drained. It contains nothing but instructions from the idle loop. Thus the processor correctness theorem $sim(d',h')$ holds for all components of the configuration but the PC and the delayed PC. Therefore we weaken the simulation relation sim to a relation $dsim$ by dropping the requirement that the PCs and delayed PCs should match:

$$dsim(d',h')$$

Until the end of the slot in cycle $t(r,s)+T$ and instruction $j(r,s)$, only instructions from the idle loops are executed. They do not affect the $dsim$ relation, hence:

$$dsim(e(r,s), g(r,s))$$

Since $res_U(d(r,s),a) \in Q$ and Q does not depend on the program counters we have $e(r,s) \in Q$. We derive that $decode(g(r,s))$ coincides with $e(r,s)$ except for the program counters. And again, because this does not affect the membership in Q, we get the desired Theorem 4.

References

[Abs06] AbsInt Angewandte Informatik GmbH. Worst-Case Execution Time Analyzers. `http://www.absint.com/`, December 2006.

[BBG+05] S. Beyer, P. Böhm, M. Gerke, M. Hillebrand, T. In der Rieden, S. Knapp, D. Leinenbach, and W.J. Paul. Towards the formal verification of lower system layers in automotive systems. In *23nd IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 2005), 2–5 October 2005, San Jose, CA, USA, Proceedings*, pages 317–324. IEEE, 2005.

[Bey05] Sven Beyer. *Putting It All Together: Formal Verification of the VAMP*. PhD thesis, Saarland University, Computer Science Department, March 2005.

[BJK+03] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In D. Geist and E. Tronci, editors, *Proc. of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *LNCS*, pages 51–65. Springer, 2003.

[BP06] Geoffrey M. Brown and Lee Pike. Easy parameterized verification of biphase mark and 8N1 protocols. In *Proceedings of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, volume 3920 of *LNCS*, pages 58–72. Springer, 2006.

[Dal06] Iakov Dalinger. *Formal Verification of a Processor with Memory Management Units*. PhD thesis, Saarland University, Computer Science Department, July 2006.

[DHP05] Iakov Dalinger, Mark Hillebrand, and Wolfgang Paul. On the verification of memory management mechanisms. In D. Borrione and W. Paul, editors, *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *LNCS*, pages 301–316. Springer, 2005.

[Fle06] FlexRay Consortium. `http://www.flexray.com`, December 2006.

[HIP05] Mark Hillebrand, Thomas In der Rieden, and Wolfgang Paul. Dealing with I/O devices in the context of pervasive system verification. In *ICCD '05*, pages 309–316. IEEE Computer Society, 2005.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[MP00] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[OSE06] OSEK/VDX. `http://www.osek-vdx.org`, December 2006.

[Pau05] Wolfgang Paul. Lecture Notes from the lecture Computer Architecture 2: Automotive Systems. `http://www-wjp.cs.uni-sb.de/lehre/vorlesung/rechnerarchitektur2/ws0506/temp/060302_CA2_AUTO.pdf`, 2005.

[Sch06] Julien Schmaltz. A formal model of lower system layer. In Aarti Gupta and Panagiotis Manolios, editors, *Formal Methods in Computer-Aided Design, 6th International Conference, FMCAD 2006, San Jose, CA, USA, November 12–16, 2006, Proceedings*. IEEE Computer Society, 2006. To appear.

[SH98] Jun Sawada and Warren A. Hunt. Processor verification with precise exceptions and speculative execution. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV '98*, pages 135–146. Springer, 1998.

[WL88] J. Lundelius Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Communication*, 77(1):1–36, April 1988.