5

Software in Hardware Description Languages

5.1 Introduction

A whole range of methods can be listed for the joint simulation of hardware and software, which are concisely summarised by Rowson in [355]. The most important criteria here are: precision with regard to timing; simulation speed; the availability of models; and the possibility of debugging the simulated software. The simulation speed and timing precision are normally in competition with one another. The approaches described in what follows provide various compromises in this context, see Table 5.1.

The most precise, but consequently also the most expensive, simulation option is to describe the processor core in question with such accuracy that the signal timing is reproduced exactly at the connections. The software is available as information in the storage model and is processed during the simulation of hardware. This particularly exact modelling is associated with the longest running times.

We can abstract from this model and demand only that the signals at the terminals are correct at every active edge of the clock signal. This can firstly simplify the model, because for the most part the signal delays can be disregarded in a synchronously executed processor core. Furthermore, the number of simulation events is significantly reduced in comparison to the precise timing. Both lead to a significant acceleration of the simulation.

In the next step we can move to the modelling of the command set and its execution. In this procedure the values are correctly illustrated in the registers and in the memory but details such as the pipelining of instructions may be neglected. As a result a large part of the timing information is lost.

The approaches described up to this point each require suitable processor models. However, techniques exist that do not necessitate the modelling of the hardware. This is the case firstly if the communication between software and hardware runs asynchronously and the time between the communications thus plays no role. In this case it is sufficient to compile the software for the simulation workstation and

Mechatronic Systems Georg Pelz
 ISBN: 0-470-84979-7

Table 5.1 Methods of hardware/software co-simulation according to Rowson [355]

Approach	Speed in inst. / sec.	Model necessary
Exact pin timing	1–100	Yes
Cyclically precise pin timing	50–1000	Yes
Instruction level	2000–20 000	Yes
Timing disregarded	Typically limited by the hardware simulation	No

to connect to the hardware by means of a type of 'handshake'. Thus the software will be executed at the full speed of the simulation workstation. A further situation in which the timing may be neglected to a certain degree is the situation in which the execution of the software is defined in a fixed time period. Accordingly, events and new inputs are only exchanged at fixed time points. Now, if we can ensure that the software is always fast enough to conclude the calculations before the end of the current grid interval, then the timing can be disregarded. Makki *et al*. [254] suggest this for a realisation with hardware description languages, but details are not provided. Another approach is followed by van Zanten *et al*. [407] and Adamski *et al*. [3]. In this the controller core and the mechanics model — both formulated in the programming language C — are linked together and simulated jointly in the initial system investigations. The controller software is thus considered without taking into account the underlying hardware. However, this simple model of the co-simulation of hardware and software is often not adequate. The reasons for this are numerous. For example, one reason is the possible influence of an underlying real-time operating system. Also, the occurrence of further interrupts — perhaps for communication with other controllers — often frustrates the use of this variant. Finally, in some cases the aim is for the simulation to reach the speed limit, for example, in order to construct fast controllers with short calculation intervals.

Further increases in speed can only be achieved by omitting parts of the model or by the use of emulation. The latter two options will not be considered further in the following.

It is often necessary for the development of the electronics for mechatronic and micromechatronic systems to record the timing between software, electronics and mechanics with a large degree of precision, in order to thereby correctly evaluate the dynamics between the domains. A good compromise here is a simulation that reflects the temporal behaviour of the running software with regard to processor cycles. The consideration of approaches for the cyclically correct co-simulation of software, electronics and mechanics forms the focal point of this chapter. In addition to the abstractions already mentioned we must also give some thought to the realisation of the co-simulation. One possibility is to use a simulator backplane, see Gasteier and Glesner [112] or Ghosh *et al*. [118]. By contrast, the methods represented in Sections 5.2 and 5.3 increasingly point in the direction of a model transformation on the basis of hardware description languages. Finally, the method described in Section 5.4 aims at the cyclically correct coupling of software

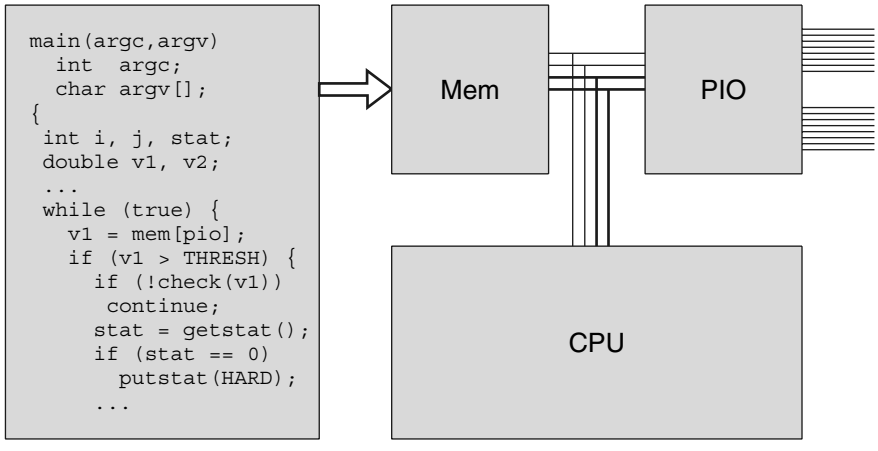

Figure 5.1 Execution of software by the simulation of hardware

processing and hardware, but, in contrast to the backplane, this is achieved at the modelling level by hardware description languages.

5.2 Simulation of Hardware for the Running of Software

The simplest and at the same time the least efficient method for the cyclically correct co-simulation of digital hardware and software is the mere description of the hardware using hardware description languages, see for example Buchenrieder and Rozenblit [51] or Le Marrec *et al.* [218], as well as Figure 5.1. In a first approximation this takes place on the level of the blocks involved such as CPU, main memory, etc. At the start of the simulation, the modelled main memory is filled with the appropriate content so that a simulation of the hardware draws the execution of the software along with it. One such model was implemented and simulated for Motorola 68HC05 architecture. It includes behavioural models for the CPU, the main memory and a parallel interface. These models include the necessary interfaces to communicate with each other via the address and data bus. The performance of such a model lies at around 500 assembler instructions per CPU second on a SUN-Sparc 20. This is clearly too slow for the time spans in the range of seconds to be considered in mechatronics. Therefore, this approach will not be described in more detail at this point.

5.3 Co-simulation by Software Interpretation

A first step towards accelerating the cyclically correct co-simulation of hardware and software is motivated by the observation that the precise consideration of

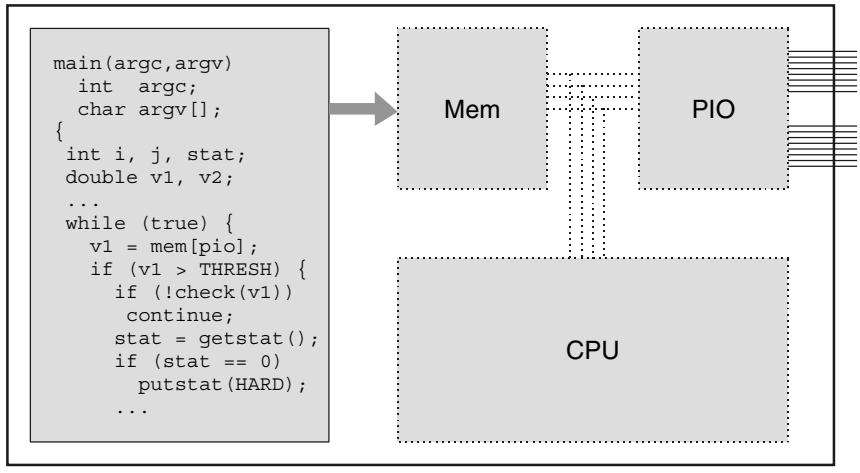

Figure 5.2 Execution of software by simulation at controller level

bus traffic between CPU and main memory, like many other details, does not contribute significantly to the investigation of the system as a whole. Rather, it is virtually always sufficient to imitate the interface behaviour of the controller, see Figure 5.2. This facilitates a whole range of simplifications in the model. Thus it may be possible to represent the memory primarily by an array of integer numbers or bit vectors. Memory access can be formulated as access to the array. The data and address bus and the associated logic are thus dispensed with completely.

In a more precise consideration, the objective of the model in question also alters. Where before it was primarily a question of describing the hardware correctly, now such a model becomes an interpreter for the running software. This is beneficial in two respects. Firstly, the model is significantly simplified, secondly there is a considerable acceleration of the simulation. Interpretative models with various characteristics exist. For example, Gupta *et al.* [130] link an interpretative software simulator to the simulator responsible for the hardware for each simulator coupling, taking into account cyclically correct timing. Furthermore, Ecker outlines the formulation of a software interpreter in VHDL, see [92], in which precise timing is largely disregarded. Finally, Pelz *et al.* [326], [327] suggest a cyclically correct implementation of a software interpreter for the Motorola 68HC05 architecture in VHDL, which is coupled to mechanics models in hardware description languages. This approach will be described in more detail in what follows. It offers a simulation speed of around 5000 assembler instructions per CPU second on a SUN-Sparc 20. Thus the performance of the simulation lies above that of the method described in the previous paragraph by approximately an order of magnitude.

Hardware description 5.1 that follows provides an example of the description of a (fictitious) processor at interpreter level. The characteristics of the processor architecture largely relate to the register variables and the command set. The model consists of a process in which one assembler instruction is executed in each loop. At the beginning the instruction is fetched from the main memory,

the Opcode is separated, and the addresses of the operands evaluated. There then follows a large CASE instruction, which serves to decode the operation in question. A few instructions are provided for each opcode, which may perform arithmetic or logical actions, set the PC in the event of jumps, calculate flags and much more.

```
ARCHITECTURE interpreter OF cpu IS
  -- Type declaration for register and main memory ...
  TYPE registers IS ARRAY (0 TO  31) OF
                    std_logic_vector(31 downto 0);
  TYPE memory    IS ARRAY (0 TO 512) OF
                    std_logic_vector(31 downto 0);
BEGIN
  cycle: PROCESS
    VARIABLE reg       : registers;                -- Registers
    VARIABLE mem       : memory;                     -- Memory
    VARIABLE pc        : natural;           -- Programme counter
    VARIABLE adr       : natural;            -- Address variable
    VARIABLE inst      : std_logic_vector        -- Instruction
                         (31 downto 0);
    VARIABLE disp      : std_logic_vector       -- Displacement
                         (31 downto 0);
    VARIABLE opcode    : std_logic_vector            -- Opcode
                         ( 7 downto 0);
    VARIABLE r3, r1, r2: natural;               -- Register adr.
    VARIABLE i8        : integer;                -- 8 bit number
    VARIABLE zflag     : std_logic;                 -- Zero flag
    ...
BEGIN
    ...
    inst        := mem(pc);                 -- Fetch instruction
    pc          := pc + 1;                       -- Increment PC
    opcode      := inst(31 downto 24);         -- Extract opcode
    r3          := To_Nat(inst(23 downto 16));      -- Determine
    r1          := To_Nat(inst(15 downto  8)); -- register adr.
    r2          := To_Nat(inst( 7 downto  0));-- from instruct.
    i8          := To_Int(inst( 7 downto  0)); -- Immediate Op.
    - Decode opcode ...
    CASE opcode IS
    WHEN op_add =>                                    -- Perform
    reg(r3)     := reg(r1) + reg(r2);                -- addition
    zflag       := (reg(r3) = 0) ? '1' : '0';      -- Zero flag?
    ...
  WHEN op_add_immediate =>                            -- Perform
    reg(r3)     := reg(r1) + i8;                 -- addition imm.
    zflag       := (reg(r3) = 0) ? '1' : '0';       -- Zero flag?
    ...
```

```
  WHEN op_sub   =>=>                                    -- Perform
    reg(r3)     := reg(r1) - reg(r2);                 -- subtraction
    zflag       := (reg(r3) = 0) ? '1' : '0';          -- Zero flag?
    ...
  ...
  WHEN op_and   =>                                      -- Perform
    reg(r3)     := reg(r1) and reg(r2);               -- logical AND
    zflag       := (reg(r3) = 0) ? '1' : '0';          -- Zero flag?
    ...
  ...
  WHEN op_load  =>                                     -- Load reg.
    disp        := mem(pc);                        -- Determine disp.
    pc          := pc + 1;                            -- Increment PC
    adr         := To_Nat(reg(r1) + disp);  -- Determine address
    reg(r3)     := mem(adr);                                 -- Load
    ...
  WHEN op_store =>                                     -- Save reg.
    disp        := mem(pc);                        -- Determine disp.
    pc          := pc + 1;                            -- Increment PC
    adr         := To_Nat(reg(r1) + disp);  -- Determine address
    mem(adr)    := reg(r3);                          -- Store in mem.
    ...
  WHEN op_branch_on_zero =>                           -- Jump command
    IF   (zflag = '1') THEN                            -- If flag = 1
      disp      := mem(pc);                        -- Determine disp.
      pc        := pc + 1;                            -- Increment PC
      adr       := pc + To_Nat(disp);     --   Determine address
      pc        := adr;                                    -- Set PC
      ...
    END IF;
  ...
  WHEN others =>
    -- Unknown opcode ...
    ASSERT false REPORT "illegal instruction"
      SEVERITY warning;
    WAIT;
  END CASE;
 END PROCESS;
END ARCHITECTURE;
```

Hardware description 5.1 VHDL description of a simple processor as software interpreter

5.4 Co-simulation by Software Compilation

5.4.1 Introduction

The approach described in the previous section interprets software during the running time in order to process it. This generates a considerable cost to be paid

during simulation. The better alternative is to shift the compilation cost from the running time to a pre-simulation stage. This generally means that two versions of the software exist. One is compiled for the simulation workstation, the other is compiled for the processor on which it is to run in the system. Now, if the software exists in a higher programming language and we are only interested in the function and not in the timing, then the differences between the processors do not play a significant role. The prerequisite for this is that the software always calculates a certain result within a predetermined time period. A whole range of approaches to HW/SW co-simulation are based upon this principle such as, for example, the work of Becker *et al*. [21] or Thomas *et al*. [399].

We can expand upon this methodology so that cyclically correct timing is also taken into account. However, to achieve this we have to make a detour in the modelling. In a first step the assembler or machine programme is compiled into a C routine that both reflects the functionality and correctly takes into account the timing of the software execution on the basis of the clock cycles of the target processor. Živojnović and Meyr show this in [438] for pure digital electronics, with both software and electronics being described in C modules so that these only have to be linked together. Pelz *et al*. expand upon this approach in [328] based upon a compiled co-simulation of software, electronics and mechanics by implementing an appropriate synchronisation between simulator and software model in a hardware description language. Here the representation of the assembler programme in C is automated by a compiler based upon a disassembler. Overall this method can also be regarded as a modelling of software, see Figure 5.3.

In what follows this approach of representing system software in C routines and linking it into a simulator on the basis of hardware description languages will be investigated in further detail.

5.4.2 Software representation

In a first stage, the system software should be represented in a C routine that takes into account both the function and timing on the level of machine instructions. In

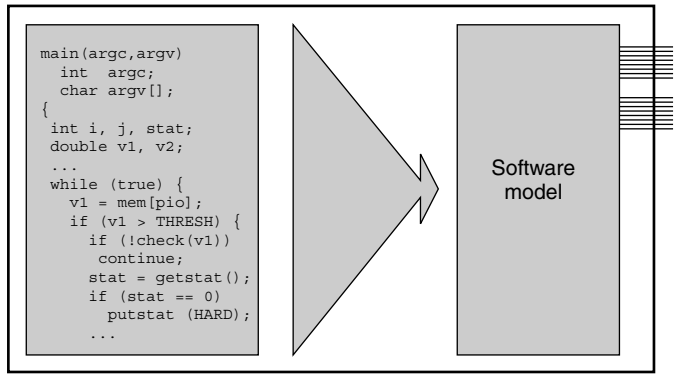

Figure 5.3 Execution of software by modelling at software level

order to subsequently bring about synchronisation, it must be possible to leave the routines at any desired points and re-enter them again later; they must therefore be 're-entrant'.

Furthermore, it is necessary that they have a memory so that the applicable system state can be held in the form of a context. Such a context thus includes the registers of the underlying processor and the complete main memory. Furthermore, a second context is saved in parallel so that the synchronisation — as described more precisely later — can refer back to an old state.

The basic idea is now to store short blocks of C instructions, which each represent an assembler instruction, one after the other in a routine. The sequence of C blocks thus corresponds with the sequence of assembler instructions, so that sequential progress through the assembler instructions corresponds with sequential progress through the C blocks.

A C block for an assembler instruction in principle contains the following components:

- Execution of the operation, e.g. for arithmetic and logical operations.
- Setting of the flags, depending upon operation.
- Setting the programme counter, normally by an increment based upon the byte number of the operation, or in the event of jumps an addition (relative) or an assignment (absolute).
- Protecting the return address on the stack in the event of subprogramme calls.
- Addition of the number of required cycles on the cycle counter.
- Calculation of the current time from the cycle counter.
- Control of the debugger.
- Details of the representation will be described in Section 5.4.4 on the basis of an example.

5.4.3 Synchronisation

Introduction

The synchronisation between hardware and software serves to effect the correct chronological sequence of events in the software model and hardware model in the simulator. A significant prerequisite for a simple and efficient solution is that the simulation of the hardware runs in a linear manner and at most is delayed only now and then. All other strategies would have an effect deep within the logic or circuit simulator used, thus shifting the problem from the modelling level to the tool level, which would often rule out solutions based upon commercial simulators.

In order to achieve this the software should run for a defined time span. This is effected by calling up an external C routine from the hardware model. With regard

to the timing of the return of the software, the question is raised as to whether the sequence of load or store instructions includes reference to the I/O ports, i.e. whether it wants to exchange data from within itself with the hardware. If this is the case then the processing of the software is interrupted immediately. Otherwise the software runs until the predefined time point. Upon return, the C routine informs the hardware of the time point t that it reached. Since the software has run in zero time from the point of view of the hardware, the hardware should now be simulated up to time point t so that time equality exists between software and hardware, and thus data can be exchanged if necessary. However, the sequence described up until now only functions as long as no interrupt is triggered. In the event of an interrupt occurring, the state of the software is initially brought to the time point at which the interrupt occurred. Then synchronisation occurs and the programme counter is set to the interrupt vector, whereupon the normal sequence can once again be resumed. The forms of synchronisation described thus far will be considered in more detail in what follows.

Synchronisation without interrupt

Let us initially assume that no access to I/O ports has occurred during the processing of the software, see Figure 5.4. Before the software can once again proceed for a certain period of time, a synchronisation must take place. This means primarily that we wait until the hardware has also been simulated up to the time point at which the software currently stands. When the software and hardware show the same value for time, the software can once again proceed and the described procedure runs from the start.

Figure 5.5 illustrates the case of access to the I/O port. Here the occurrence of a corresponding load or store command leads to the software sequence being

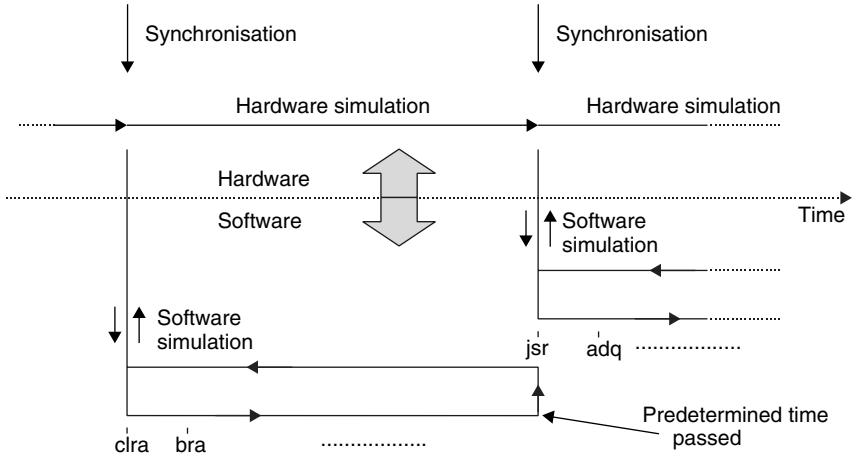

Figure 5.4 Synchronisation between hardware and software after the time allotted for the software has elapsed

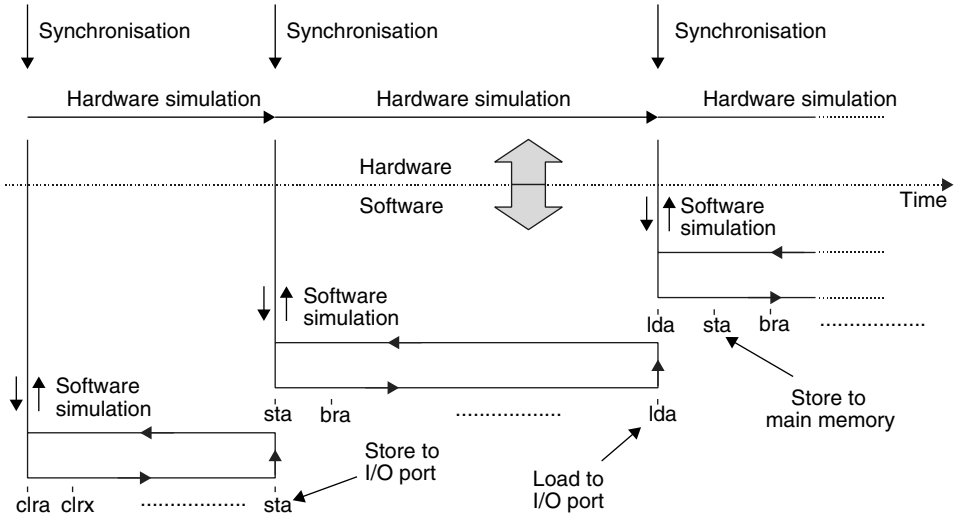

Figure 5.5 Synchronisation of software and hardware after the occurrence of load and store instructions (lda and sta) relating to the I/O ports

interrupted. Then we again wait until the hardware has reached the current time of the software. At this point the appropriate values can be exchanged between hardware and software. Then the software is restarted.

Synchronisation after an interrupt

This case occurs if the software has been executed up until time point t and it is found during the hardware simulation that an interrupt has been triggered at time point $t' < t$, that has invalidated the current progress of the software simulation, see Figure 5.6. The problem is solved in two stages. In the first stage the software has to be brought back to its state at the time of the interrupt t'. We first jump back to the old state that is stored at the start of every software operation. This is also called a time-warp in the literature on the general coupling of simulators, see the work of Jefferson [168] and [169]. Then the software is simulated up until the time of the interrupt. We can think of this as a type of 'replay' of a sequence that has already played out in the past. After the replay the software shows the precise state at time point t'. A synchronisation point is then inserted here, which permits the interrupt to be taken into account at exactly the right time. Then the software simulation begins again from the instruction that refers to the interrupt vector.

5.4.4 Example of software modelling

The representation of the software shall be explained on the basis of an example in what follows. Programme 5.1 shows parts of an assembler programme and

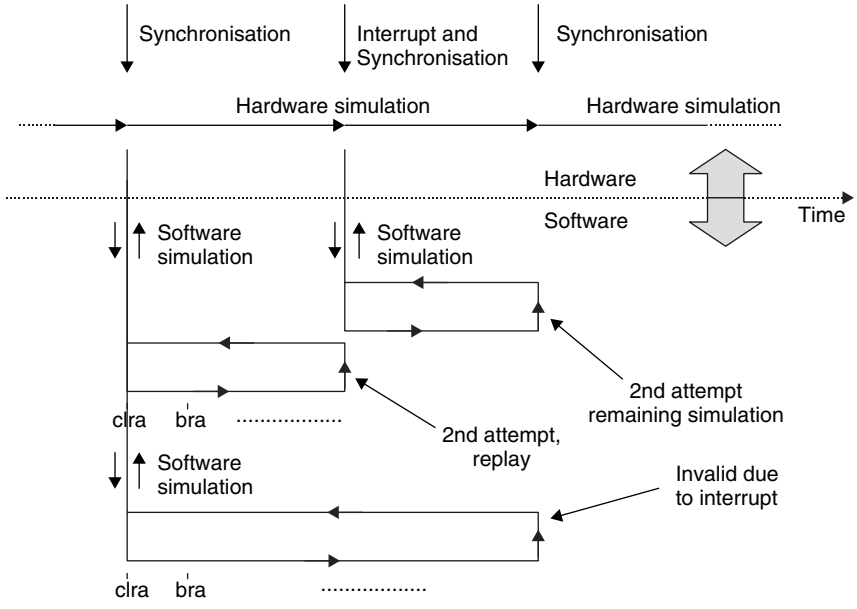

Figure 5.6 Synchronisation of software and hardware after the occurrence of an interrupt

Programme 5.2 shows the corresponding C routine which was automatically generated. Both the assembler instructions in question and the context of the C routine are compatible with the architecture and the command set of the Motorola 68HC05 microcontroller.

```
PORTA: EQU $0010 ; Declaration of PORT A as address
PORTB: EQU $0001 ; Declaration of PORT B as address
PORTC: EQU $0002 ; Declaration of PORT C as address
PORTD: EQU $0003 ; Declaration of PORT D as address

       org $100  ; Position in the memory: 0100 Hex
START:           ; Start label
       lda PORTA ; (load A) Load port A in accumulator
       jsr SRX   ; (jump subroutine) Execute subroutine SRX
       bra SRY   ; (branch) Branch to label SRY
       ...
SRY:             ; Label SRY
       ...

       org $200  ; Position in the memory: 0200 Hex
SRX:             ; Label of the subroutine SRX
       ...
```

Programme 5.1 Excerpt from assembler programme

Upon its call up, the fundamental sequence of the C routine initially rests upon determining whether this is the first time the routine has been run. If so, a whole

range of initialisations are necessary, such as, for example, filling the memory with the programme, resetting the register and jumping to the first instruction.

If the C routine has been called before, the correct context must first of all be created. If it is a replay the old stored context is activated by exchanging (`exchange_context`) with the current context. Then the old context is always protected by copying (`copy_context`). The jump to the `hub` brings about a jump to the instruction referred to in the programme counter of the current context.

The `lda`, `jsr` and `bra` instructions from the assembler programme can also be found in the C routine. There are called by labels (`l256, l258, l261`), which permit jumping to the instructions using the `goto` command.

First the `lda` should be considered more closely. Depending upon the targetted address this command fetches a value from the memory or from a port and stores it in the accumulator. First a routine is called up for this instruction, which controls the debugger and thus permits it to visualise the software sequence, indicate values, and control the software sequence by means of breakpoints. The user interface of the debugger is shown in Figure 5.7. The next instruction decides whether the

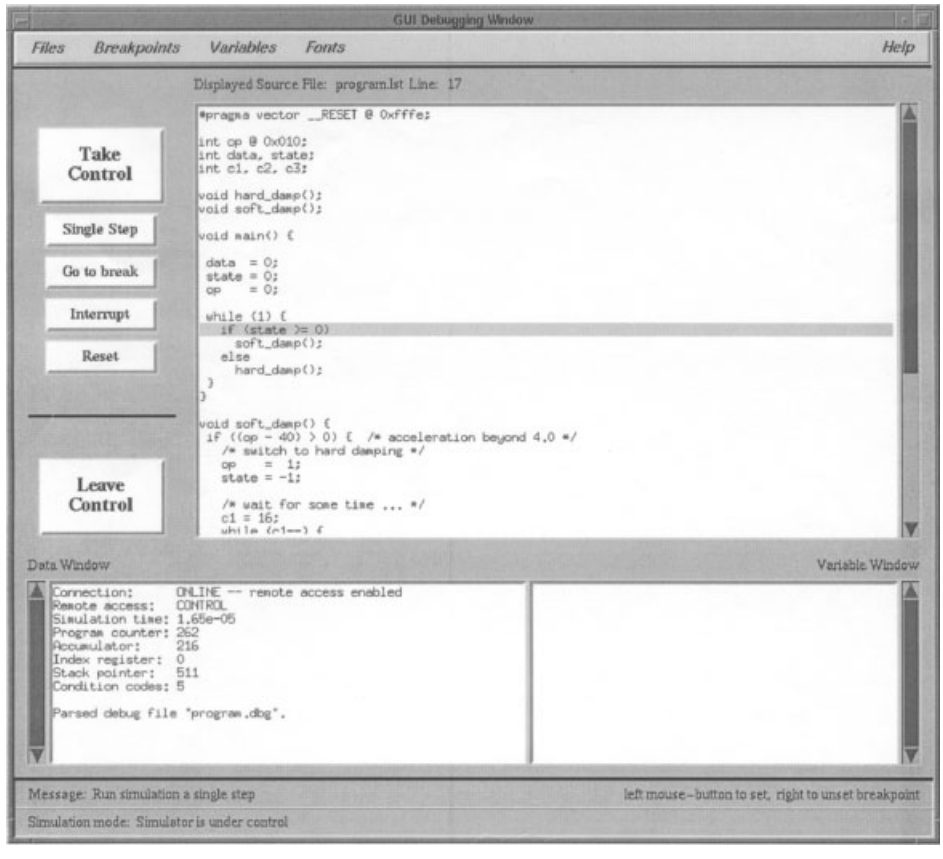

Figure 5.7 Software debugger for virtual hardware

address given in the direct addressing is a port represented in the memory area or a memory location in the main memory. In the first case the addressed port is accessed via the routine `fetch_io`. In the second case the accumulator `c1->ac` is set to the value of the memory cell at which the byte points to the opcode. It should also be mentioned that the pointer `c1` points to the current context. A type declaration of the context is located at the start of the C routine. Then the cycle counter is incremented by 3 and the programme counter by 2. Finally, the affected flags are updated and the current time `t_cur` calculated. The two other commands shown are processed in a similar manner.

The `jsr` instruction describes the call of a subroutine, so that the return address is initially stored on the stack in two bytes. Then the address of the subroutine is calculated from the two bytes following the opcode and entered into the programme counter. Then the cycle counter is incremented and the current time calculated. Finally there is a jump to the label `hub` at which the large `switch` instruction initiates a jump to the correct label. This diversion is necessary because in C it is not generally possible to jump to a variable destination by means of a `goto` command.

Finally, the `bra` instruction includes the calculation of a relative jump, which can also be in a backwards direction. The second byte of this instruction — the width of the jump — should thus be regarded as a signed number, which is expressed in the appropriate C instruction. After the normal incrementation of the cycle counter and the calculation of the time the actual jump again takes place via the hub.

```
typedef struct context {
 /* Programme counter (pc), Accumulator (ac), Index register
    (ix), Stack pointer (sp), Flag register (cc), Cycle
    counter (cyc), Main memory (m) ...*/
 unsigned int ac, ix, sp, pc, cc, cyc, ...;
 unsigned int m[MEMORYSIZE];
 ...
} CONTEXT;
static CONTEXT con1, con2, *c1=&con1, *c2=&con2;

software_sim(t_start, t_stop ... ) ... ; {
 if (t_start > 0.0) {
   if  (t_start ss< t_cur_old) {
     /* t_cur_old = Time when routine was last left,
        Replay! ... */
     exchange_context(&c1,&c2);
     ...
   }
   copy_context(c1,c2);
   ...
   goto hub;
 }
 else {
```

```
  /* Start time = 0, first call:
     Initialise debugger, logger, context etc.
     Fill the main memory with the programme */
  c1->m[256] = 182; c1->m[257] = 16;
  c1->m[258] = 205; c1->m[259] = 1; c1->m[260] = 20;
  c1->m[268] = 32;  c1->m[269] = 3;
  ...
  /* Initialise context ... */
  c1->pc = 256*c1->m[MEMORYSIZE-2] + k1->m[MEMORYSIZE-1];
  c1->ac = 0; c1->ix = 0; c1->sp = 511; c1->cc = 0;
  ...
  goto hub;
  /* Assembler programme in C ... */
  l0256:     /* lda, Load Accumulator, direct addr. */
    debugger(...);              /* Control debugger */
    if (is_io(c1->m[c1->pc+1]))/* IO or main memory? */
      c1->ac=fetch_io(c1->m[c1->pc+1]);/* IO access */
    else
      c1->ac=c1->m[c1->m[c1->pc+1]];/* Main memory access */
    c1->cyc+=3; c1->pc+=2;    /* Increment cyc, pc */
    set_flags(...);            /* Update the flags */
    t_cur=c1->cyc*CYCTIME;      /* Update the time */
  l0258:        /* jsr, Jump Subroutine, ext. addr. */
    debugger(...);              /* Control debugger */
    c1->m[c1->sp--]=(c1->pc+3)%256;/* Protect return */
    c1->m[c1->sp--]=(c1->pc+3)/256;/*   address on stack */
    c1->pc=256*k1->m[c1->pc+1]+c1->m[c1->pc+2];/* Set pc */
    c1->cyc+=5;                   /* Increment cyc */
    t_cur=c1->cyc*CYCTIME;          /* Update time */
    goto hub;                  /* Initiate the jump */
  l0261:      /* bra, Branch, relative addressing */
    debugger(...);            /* Control debugger */
    c1->pc=c1->pc+2+c1->m[c1->pc+1]>127 ?/* Calculate rel. */
    (-(256-c1->m[c1->pc+1])):(c1->m[c1->pc+1]);/* jump */
    c1->cyc+=3;                     /* Increment cyc */
    t_cur=c1->cyc*CYCTIME;            /* Update time */
    goto hub;                   /* Initiate the jump */
  ...
  hub:
  switch(c1->pc) {
    case 256: goto l0256;
    case 258: goto l0258;
    case 261: goto l0261;
    ...
}}}
```

Programme 5.2 Simplified software model in programming language C

The main task of synchronisation is to act as an interface between software and external hardware. Externally it adopts the connections of the processor. Internally the C routine is called up. In accordance with the preceding representation of the synchronisation algorithms, a formulation in a hardware description language will now be represented, see Hardware description 5.2. The language used here is MAST (Avant!) because the research work in question, see [328], was performed in this language.

The majority of the synchronisation lies in a WHEN instruction, the body of which is executed if its condition is true. It is thus largely comparable with a process in VHDL. In the body there is initially an interrogation of the interrupt line to determine whether a replay is necessary. This is performed if necessary, and then the actual execution of the software takes place. Upon return from the C routine the software reports that it was able to simulate until time point `t_cur`. Then an event at time `t_cur` is initiated upon the `softsync` signal. When this occurs, software and hardware are synchronised. Thus data can be exchanged and a new software operation started. This is taken into account accordingly by the `WHEN` instruction.

```
template m6805 ...                          # Interface description
  ...
{
 state time t_cur           # Current software time upon return
 state time t_old       #  Start time of the last software call
 state time step    #  Desired length of the software operation
 state logic_4 softsync   #  Carries events for synchronisation
 foreign software_sim                       # External C routine
 ...

 # If simulation beginning or event at the softsynch
 # variable or active edge on the interrupt line ...

 when (time_init | event_on(softsync) |
       (event_on(interrupt)&(interrupt==l4_0)) ) {

   if (event_on(interrupt)&(interrupt==l4_0)) {    # Interrupt!
     # Replay, time supplies the current time
     (t_cur, ...) = software_sim(t_old,time,...)    # C-Routine
     ...
   }
   ...
   (t_cur, ...) = software_sim(time,time+step,...)  # C-Routine
   schedule_event(t_cur, softsync, l4_1) )     # softsync event
   t_old = time                              # Save old start time
   ...
 }
}
```

Hardware description 5.2 Simplified description of the synchronisation between hardware and software in the hardware description language MAST

5.4.5 Debugging of software

The visualisation of software cannot be achieved in a worthwhile manner using the tools of an electronics or mechanics simulator. Ideally, the tools used for pure software development would be used. Such debuggers show the instruction currently being executed and the content of the variables. Furthermore, it is possible to act upon the sequence of the software by setting breakpoints and then investigating particular points in steps. It should also be possible to change the value of the variables during execution.

However, we are dealing with software that is run on virtual hardware. Furthermore, feedback effects from electronic and possibly mechanical system components, also have to be taken into account. Such a debugger has been developed, see Pelz *et al*. [328], and correspondingly incorporated into the software model. Figure 5.7 shows the user interface that has been developed for this.

The two buttons 'Take Control' and 'Leave Control', which allow us to take over the control of the simulation or leave it again, are of primary importance. In control mode we can move forward in 'Single Step' mode or proceed directly to the next halt point 'Go to break'. An 'Interrupt' interrupts such a sequence, whilst 'Reset' restores the original state. In the top window the system programme is displayed at assembler or programming language level. Clicking on a line sets or recalls a break. The bottom left window shows the most important system information, and particularly the current content of the register, whilst the bottom right window shows the variable contents.

5.5 Summary

In this chapter the inclusion of software using hardware description languages was investigated. Using the results obtained we can now look at systems that incorporate software components in addition to electronics and other domains. Significant features are the cyclically correct management of software operation on a controller, efficiency as a result of the compiled simulation of the software, and the options of linking in a debugger for the visualisation and control of the simulation process. Using the methods for the modelling of mechanics in hardware description languages, dealt with in the next chapter, yields a universal modelling process for mechatronic and micromechatronic systems that can be executed directly upon available commercial simulators.