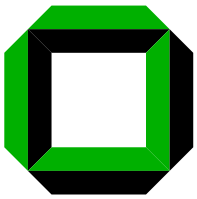

Universität Karlsruhe (TH)
Institut für Betriebs-
und Dialogsysteme

Lehrstuhl Systemarchitektur

Design and Implementation of a Debugging Unit for the OpenProcessor Platform

Stefan Bach

Studienarbeit

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter: Dipl.-Inform. Raphael Neider

20. Februar 2008

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 20. Februar 2008

_____

Stefan Bach

The OpenProcessor system is being developed a research platform for hardware/OS co-design research done at the University of Karlsruhe. This study thesis aims at developing a debugging solution for the OpenProcessor system, which at the time of writing still lacked any sophisticated debugging support.

The thesis surveys different debugging designs used in today's popular processing systems, derives an appropriate design of a debugging unit for the OpenProcessor platform and describes details of an implementation of that design in a hardware description language and its integration into the existing core.

Finally a software component that interacts with the implemented hardware solution is developed and documented. The debugging software offers an easy-to-use command line interface for debugging the OpenProcessor system.

Contents

1. Introduction

The System Architecture Group at the University of Karlsruhe is currently developing the OpenProcessor platform as a basis for future Hardware/OS co-design research. The project aims at a fully customizable computing environment, on which researchers can explore potential benefits of a tighter integration between hardware and operating system functionalities.

This study thesis presents the design and an implementation of a debugging unit for the OpenProcessor system. Throughout the time-line of this work, the OpenProcessor was still in an early stage of its development. The core system already worked according to its specification, but not much sophisticated hardware or software support for it had been developed. Therefore, at the time this thesis work was started, the software development process for the OpenProcessor was a very challenging task. The system allowed nearly no monitoring of software execution. The only possibility to get a glimpse of what is going on in the insides lied in using Verilog simulation. But this is a time-consuming process and it requires the user to follow the OpenProcessor on the lowest level, namely ones and zeros on the wires. Those circumstances rendered debugging quite hard and required good oracle skills from the debugging user.

The goal of this work is the elimination of said drawbacks in OpenProcessor-debugging. Before this thesis sets out to design the required features of OpenProcessor debugging support, it will first examine a number of successful debugging approaches from a wide range of computer systems. The studied systems will show that the basis for successful debugging lies mostly in the availability of breakpoint and single-stepping support, as well as detailed inspection functionalities that reveal the internal system state. Based on those findings, a design for a debugging unit that fits the OpenProcessor system is then conceived. And of course this design would be of little help to the suffering OpenProcessor software-developer, if it was to remain written only on a pile of paper. So to demonstrate the feasibility of the proposed design (and to actually enable its use), an implementation of the proposed debugging unit complements this work. Details on interesting aspects of the created hardware logic and its software counterpart are also presented.

The next chapter presents both background and related work for the OpenProcessor debugging system. Then Chapter 3 draws up the design and specification for the OpenProcessor debugging unit whereas Chapter 4 covers its implementation. Finally Chapter 5 concludes with a summary of the achieved results.

2. Background and Related Work

This chapter first presents some background on the OpenProcessor system in Section 2.1. Later on Section 2.2 surveys a broad spectrum of successful debugging designs and analyzes how their concepts relate to the requirements of the OpenProcessor system.

2.1. Overview of the OpenProcessor Platform

The OpenProcessor [6] platform is a simple version of a fully featured computing environment. It is built of a central processing unit (*CPU*), main memory, and various input/output (*I/O*) components. The OpenProcessor hardware is written in Verilog and implemented on an FPGA based development board. Figure 2.1 presents a logical view of the OpenProcessor platform and illustrates the boundary between the Verilog implementation and supporting hardware. For example the Ethernet support consists of an Ethernet PHY chip on the development board which works in conjunction with a Verilog-implemented I/O controller. At the time of writing, the OpenProcessor platform runs on a *Memec Virtex-4 MB Development Kit* [5] and takes up about 17% of the slices available in the *Xilinx XC4VLX60* FPGA [8] that is installed on the board.

The CPU of the OpenProcessor platform is a reduced instruction set core, which is composed of four pipeline stages. The core provides a custom instruction set architecture (*ISA*) based on a load/store design. The ISA defines 30 general purpose registers, a special zero register, and—to automatically store return addresses—a link register. Although there are no technical differences within the set of the 30 general purpose registers, the platform manual gives some special meaning to some through defined calling conventions and recommendations on which registers should be used to store, e.g., the stack pointer.

The pipeline of the OpenProcessor pushes instructions through an instruction fetch (*IF*), instruction decode (*ID*), execute/memory (*EX*), and finally a write back (*WB*) stage. The design includes load and result forwarding logic, which avoids the need for empty (*NOP*) instructions in the presence of data dependencies. Thus the ISA allows all register-register, register-immediate and memory instructions to directly follow each other. Only jump instructions require special attention. The ISA defines a single delay slot after all jump instructions, which is executed even if the jump is taken. Figure A.1 in Appendix A presents a schematic of the core, thus giving detailed information about its implementation.

The core connects to its surrounding components through the instruction and data memory interfaces and a dedicated interrupt line. The memory interfaces connect the core to a central bus, which provides access to the OpenProcessor devices through memory mapped I/O. Main memory is also attached to that central bus, either as an interface

to FPGA-internal SRAM or as an interface to external memory, such as DDR SDRAM on the development board.

A customizable chain of memory controllers connects the CPU core to the bus; each of the controllers contributes to virtual memory (*VM*) and caching. Figure 2.2 shows a standard configuration of the memory chain that supports unaligned memory accesses, a first-level cache, and a translation look-aside buffer (*TLB*) for virtual addressing. The final component in the chain maps the internal request protocol of the core to the WISHBONE [7] bus specification—a public domain System-on-Chip interconnect. All components in the system have WISHBONE-compatible controllers and connect to the CPU through a multiplexer based shared-bus arbiter (cf. [7, pg. 98]).

Currently the OpenProcessor platform includes a number of devices, some with system functionalities and others as I/O controllers. The most important system devices are a *programmable interrupt controller*, which arbitrates multiple interrupt sources and drives the IRQ line of the core, and a *timer / counter*, which is necessary to support a time-sliced operating system. The platform also provides an *event counter*, which monitors core behavior, and an interface to the *digital clock manager* of the FPGA.

Off-chip I/O is either completely controlled inside the FPGA, with the corresponding connectors attached directly to the FPGA pins, or is supported by additional hardware on the development board. On the Memec Development Kit the system provides the following input and output devices:

- RS 232 serial communication,
- a 10/100 MBit Ethernet port,
- access to a liquid crystal display (*LCD*) embedded on the development board,
- a VGA controller with an FPGA-internal frame buffer,
- a PS/2 input interface and
- general purpose I/O through switches and LEDs on the development board.

The OpenProcessor platform can flexibly load software to run on the system. Normally, part of the SRAM memory on the FPGA is pre-programmed with an operating system loader when configuring the chip. The loader software communicates through the serial RS 232 port and interprets records according to the SREC standard—a simple ASCII encoding for binary data by Motorola, which supports writes and jumps to arbitrary memory addresses. Thus any code can be easily pushed from a standard PC to the OpenProcessor platform.

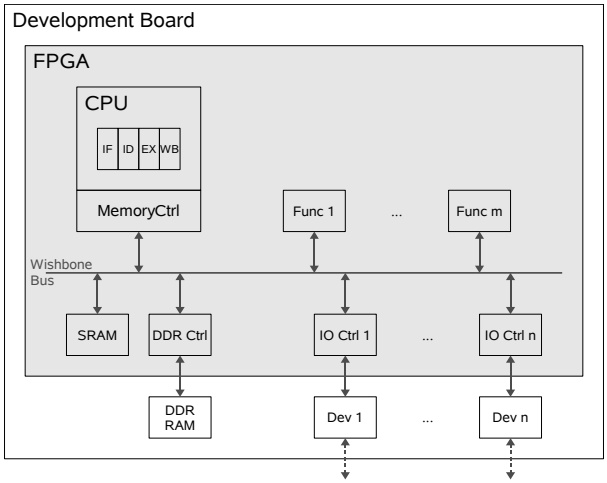

Figure 2.1: Logical view of the OpenProcessor Platform

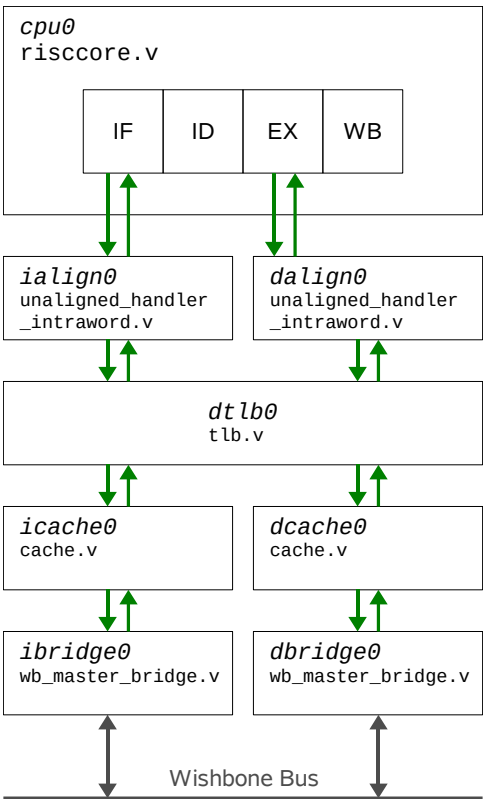

Figure 2.2: Standard memory access chain of the OpenProcessor

2.2. Survey of Popular Debugging Designs

The introduction to the OpenProcessor platform in Section 2.1 shows that the system possesses characteristics from a broad range of computing hardware:

- The OpenProcessor platform is modeled to resemble current computer systems. Although the performance of an FPGA-based research system cannot compete with commercial systems, the OpenProcessor platform is still able to run self-contained software, with I/O such as a display and keyboard directly attached to the FPGA pings.
- Considering the performance limitations and the OS loading process—receiving the software from an external source through a serial connection, the OpenProcessor platform also resembles modern micro controllers. Those systems provide a sophisticated environment for code execution but still rely on an external platform for software development and debugging.
- Finally, it is worth to look at the OpenProcessor platform from the viewpoint of its own implementation. Being built from Verilog code and run inside an FPGA, the system exhibits similarities to application specific integrated circuits (*ASIC*).

This chapter will look at the debugging aspect for all three views of the OpenProcessor platform. The following sections present a proved and popular example from each category, give an introduction to the associated debugging approach, and estimate how well the presented approach would fit the OpenProcessor platform.

2.2.1. IA-32 Processor Architecture

The by far most popular ISA in today's personal computer platforms is the family of IA-32 (also called x86) compatible systems. Modern personal computers are widely used not only to run programs but also to develop and debug applications right on the target hardware. Since the introduction of the Intel 80386 CPU, all implementations of the IA-32 instruction set architecture provide debugging support through special debug registers. Although modern CPUs have extended the debugging system, e.g. to support sophisticated tasks like performance monitoring, this section focuses on the initial debugging approach, which has been supported since the first 80386 chips.

Summary of the debugging design

The IA-32 debugging facilities [4] consist of two special exceptions—the debug exception (#DB) and the breakpoint exception (#BP)—and a set of debug registers DR0 to DR7 (Figure 2.3). While in privileged operating mode those registers can hold up to four breakpoints, which are stored as virtual addresses in the DR0 to DR3 registers. If an active breakpoint is hit on instruction loading, data access, or an I/O operation, the core raises the #DB exception. If a breakpoint is currently active and for which access-types it should trigger is set up in the register DR7, which also allows to enable or disable

the complete debugging system. In addition to the dedicated breakpoint registers the debugging system can also be configured for single-stepping, which then raises the #DB exception after every instruction, or it can be configured to trap on context switches to a certain task. Registers DR4 and DR5 are reserved for future use and register DR6 gives information about the cause of a triggered exception.

Further breakpoint support is provided through the #BP exception, which can be triggered everywhere in the code by executing a special INT 3 instruction. This allows the definition of breakpoints directly in the program text and allows instruction breakpoints to be set for more than four addresses.

The IA-32 debugging system does not require specific instructions to examine the processor state or the memory contents. Since debugging happens on the same machine that runs the software under test, the IA-32 instruction set already provides all means to examine and modify relevant state.

Relevancy for the OpenProcessor system

The described debugging system has worked well for over a decade. But since access to the debugging system is restricted to supervisor mode, user applications cannot directly access the debugging registers. Thus debugging requires support by the operating system through a debugging API. This requires the development of a reliable software interface with a privileged core—thus bugs in the debugging API can crash the whole system, if a raised exception accidentally stays unhandled. [2]

Due to design of OpenProcessoras a system for hardware as well as operating system research, it is likely for the OS running on the OpenProcessor system to be frequently changed. This makes the OS software itself error-prone, and it is a bad idea to try and rely on it for debugging purposes. Due to that reason the presented approach does not fit the needs for the OpenProcessor platform. However, the IA-32 debugging system conveys that a comprehensive breakpoint support is one of the main issue to facilitate application debugging.

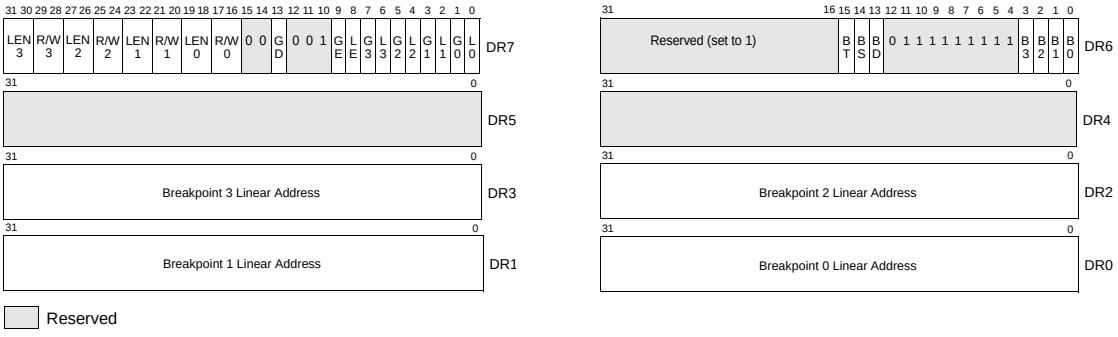

Figure 2.3: DR0-DR7 of the IA-32 Architecture [4]

2.2.2. Atmel megaAVR Micro controllers

This section looks at the debugging support in the Atmel megaAVR [1] family of micro controllers. The megaAVR is a popular device in that market, and it is often used in embedded systems education.

Summary of the debugging design

The megaAVR supports the standardized interface of the Joint Test Action Group (*JTAG*) [3] to program and debug the chip. The JTAG specification allows for serial communication with devices under test through a standardized Test Access Port (*TAP*). Each TAP controller included a standard state machine, which governs the input and output pins of the TAP. Input data is either directed to an instruction register or shifted into a data scan chain. When the TAP is used to accesses a scan chain, the value of the instruction register determines which internal device registers are chained to form the scan chain. JTAG only standardizes instruction register values to conduct boundary scans, which build up a scan chain from the I/O pins of a device. In addition, a manufacturer can introduce further capabilities to the TAP by interpreting proprietary instruction register values.

Figure 2.4 shows the JTAG related parts of the megaAVR micro controller. Debugging is supported by a breakpoint unit, a special debug communications register, and the pos-

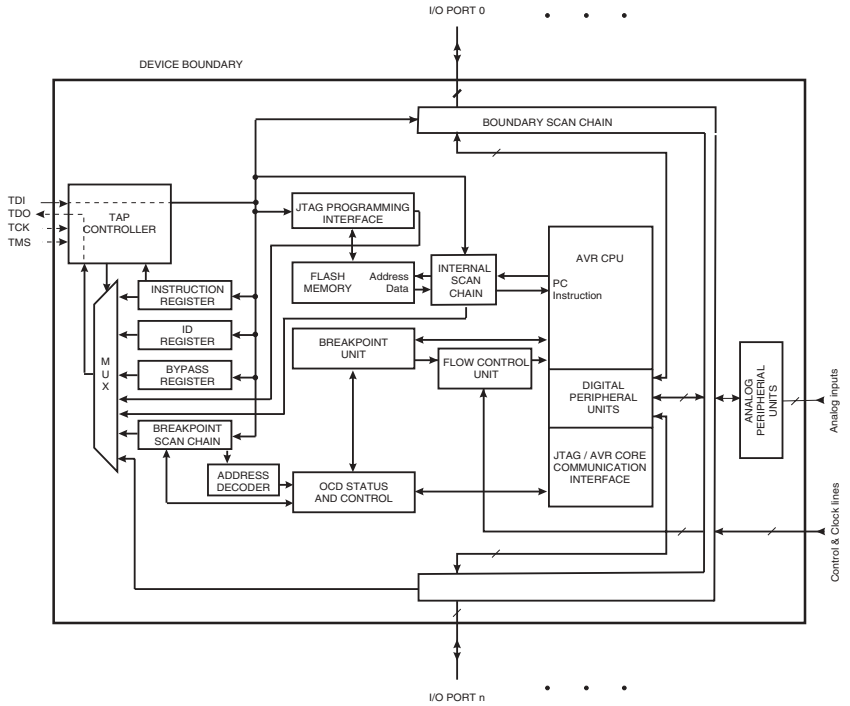

Figure 2.4: Block diagram of the JTAG test access port on the megaAVR [1]

sibility to access the program counter and the instruction register through JTAG. This allows to directly employ functionalities of the CPU core to examine and modify system state. For example, the user can instruct the core—using normal op-codes—to copy a register value from a general purpose register to the memory-mapped communication register. The user can then read the value through the JTAG interface. Atmel also provides private JTAG instructions to halt the core at any time and single step through the execution. Halting on special conditions is supported as well: The hardware breakpoint unit allows to define up to 4 breakpoints on either data or instruction addresses and a software BREAK instruction suspends normal execution if encountered by the core.

Atmel uses the described capabilities of the megaAVR micro controller to provide a software/hardware bundle that employs the micro controller to build an in-circuit emulator of itself. The bundle includes an IDE software that allows the user to debug programs that run on the micro controller in very much the same way, as if they were developed for a regular personal computer.

Relevancy for the OpenProcessor system

The described debugging approach has a major difference to the previous one: In contrast to the IA-32 setup, the debugging user of the megaAVR works with an interface that runs on a separate machine as the device and software under test. When debugging the micro controller, development work is done on a regular PC, which is linked to the device only through the JTAG port. This approach is well-suited for the needs of the OpenProcessor system, since debugging externally—only supported by some special hardware unit—allows to debug even sensitive parts of the operating system without the need that they function properly.

Knowing that a JTAG interface is widely available in micro controller families, where it is often used for debugging, creates hope that some standardized procedures might exist for this purpose. If this was the case, a similar implementation for the OpenProcessor system would allow to leverage existing IDE software for easy debugging. But was shown above, the JTAG standard only covers the communication protocol with the device under test and its application towards boundary scans. This implies that a JTAG test access port for the OpenProcessor platform cannot reduce the need for a customized software implementation, even if a micro controller-like debugging approach is implemented.

2.2.3. Simulation and Logic Analyzers

The last example in this chapter looks at the OpenProcessor platform mainly from the viewpoint of its implementation. For easy extensibility with respect to hardware/OS co-design research, the system is written in the hardware description language (*HDL*) Verilog. The use of an HDL is part of the development model for application specific integrated circuits, where debugging is mainly achieved through simulation and logic analyzers.

Summary of the debugging design

The simulation approach has already been extensively used while developing the OpenProcessor system. A simulation software parses the Verilog sources and simulates system behavior based on stimuli provided for the simulated input ports. This allows an in-depth monitoring of low-level system activity, as can be seen in Figure 2.5. But the simulation task is computationally very expensive, so this approach is only feasible for inspecting a limited number of cycles after system initialization. For example, the Icarus Verilog simulator would take about eight hours on an Intel Pentium 4 machine (3 GHz CPU, 1 GB RAM) to simulate one second of real system time.

A debugging approach that directly inspects the real hardware is realized through logic analyzers. Those devices connect to the I/O pins of a device and allow to monitor and record signal behavior at high frequencies. Logic analyzers usually provide the possibility to set a number of trigger conditions for the monitored signals and then record the signal behavior for a short period of time once those conditions are met. For Xilinx FPGAs the manufacturer provides an on-chip logic analyzer, which is built of normal on-chip resources available on the FPGA, e.g. it employs Block RAM to store signal behavior. An advantage of this design over traditional logic analyzers is that it removes the need to drive all interesting signals to external pins. This product—marketed as "ChipScope Pro"— is realized as an IP core that can be instantiated inside the tested design and attached to all internal signals directly through the HDL. The core communicates with an external software component through a JTAG interface of the FPGA, through which trigger conditions are controlled and recorded signal behavior can be read.

Relevancy for the OpenProcessor system

While the described debugging approach using the ChipScope logic analyzer is likely to suit the debugging needs for further hardware changes to the OpenProcessor platform, it lacks the generality required for software debugging. Some functionality similar to

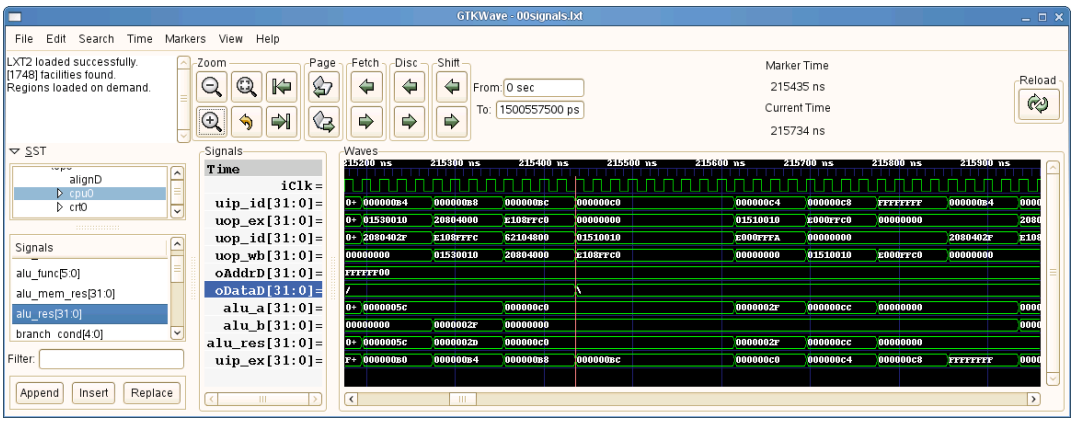

Figure 2.5: Simulation allows detailed inspection of every signal

breakpoints could be achieved through appropriate trigger conditions on the signals of the WISHBONE bus and important internals of the CPU core. But the results recorded this way would only allow insight to register and memory values of those addresses that were directly involved with the instructions that are executed during signal recording. Further inspections of the system state or single-stepping would not be possible.

With respect to the debugging of hardware changes and extensions of the OpenProcessor platform, utilizing ChipScope can support tracing of logic signals beyond the scope that is feasible for simulation. But due to the possibilities provided by an on-chip logic analyzer—low-level insight on every internal wire—, it is hard to think of a generalization that would provide a flexible hardware debugging system. More likely this form of debugging will be very useful in the early development stages of either new hardware components or changes to existing ones. If bugs arise in that stage, a hand carved instantiation of ChipScope can be used to monitor signals suspected of non-intended behavior. This debugging can on top be supported by a more general and software-focused debugging unit, like the one hinted on in the previous sections.

3. Design of the OpenProcessor Debugging Unit

This chapter will discuss a design for debugging the OpenProcessor system. The design is developed in a top-down approach, setting general debugging goals first and discussing their details in the following subsections. Finally the chapter concludes by summarizing the resulting design.

3.1. Overview and Requirements Analysis

The OpenProcessor system is mainly a platform for hardware/OS co-design research and thus one wishes for a debugging implementation that takes this into consideration. Since changes to the hardware as well as the software of the system are to be expected for the future, both areas are likely to need debugging support. But while software development is restricted by the narrowly defined ISA of the system, hardware development leaves the designer great flexibility. This renders it infeasible to conceive a design for hardware debugging on a level higher than basic signal analysis. As tools for that kind of hardware debugging already exist and have even been used in the development of the OpenProcessor this work will focus on the aspects of software-debugging.

Most important for software debugging is to empower the debugging user to understand the execution of the tested software on a step-by-step basis. This demand fits the debugging needs of many computer systems, which has led to comparable debugging functionality over a wide range of platforms. Even the examples of the previous chapter show that both the large-scale personal computer and the small micro controller have breakpoint support, single-step execution, and the possibility to inspect system state in common. Such functionality properly fulfills the requirement to understand execution at the finest granularity available to the programmer and also allows him to selectively step through only specific parts of the code. The capabilities of the OpenProcessor debugging system will thus follow the already proved ideas presented earlier.

Orthogonal to the debugging capabilities of a system is the design of its debugging control. The examples in the previous chapter have shown three different approaches, of which the debugging design of the microprocessor best fits the needs of the OpenProcessor system. Approaches that show tighter integration, such as the self-contained debugging system in personal computers, do not fit the OpenProcessor system, because they require reliable OS support. Apparently the latter is not available on a platform for operating system research. Low-level designs, e.g. using logic analyzers, are too inflexible to support comfortable software debugging and are better suited when debugging hardware changes. Therefore the design choice for the OpenProcessor debugging unit

aims at an on-chip debugging core with an interface to an external debugging control, for example running on a standard personal computer.

With respect to the design of a debugging system with the presented outline, the remaining work is divided into the following sections: First the details on halting the core at breakpoints with subsequent single-stepping through execution will be discussed in Section 3.2, afterward, Section 3.3 deals with access to system state, and Section 3.4 finally aggregates the designed functionality in the specification of a communication interface for the debugging unit—which simultaneously serves as a summary of the available debugging functionalities for the user.

3.2. Breakpoint and Single-Stepping Support

Being able to observe system behavior at a speed that is slow enough for a human user to follow builds the basis for all software-debugging efforts. This is commonly realized through the distinction between a *running* and a *halted* state for the CPU core. While the core executes normally in the first state, the latter provides single-step execution only upon explicit user request.

The pipeline architecture of the OpenProcessor already provides the basis for halting and single-stepping the core. Figure A.1 shows that all boundary registers between pipeline stages can be stalled through disasserting their clock enable registers. This usually happens automatically during normal execution when the core waits for a bus accesses to finish. The debugging unit can use this facility by augmenting the core with an external stall wire and thus being able to stop execution through freezing the pipeline registers.

The question of how to halt the core automatically leaves more interesting design choices. Common to the examples in Section 2.2 is the existence of multiple possibilities to halt the core: either through storing addresses in special breakpoint registers or by inserting a special break instruction in the program text.

Halting at a certain instruction address can be achieved through both alternatives, with each of them having certain advantages over the other. Breakpoint registers ease the addition of new breakpoints to a running system but can only be provided in a limited number, since each single breakpoint register requires a distinct hardware counterpart. A special break instruction on the other hand allows an unlimited number of code breakpoints but makes it harder to add them to a running system, since additional work is required to overwrite and restore the original instructions.

In contrast to instruction breakpoints, the implementation of data breakpoints leaves little design choices. Since memory addresses are partly calculated based on register contents, it is not feasible to pin down all instructions that might access a certain memory address. Thus breakpoint registers that monitor memory/bus accesses are the only option to support data breakpoints.

The OpenProcessor debugging unit should include support for both instruction and data breakpoint registers. Data breakpoints can be the only useful help, if the programmer cannot pin down which code section is responsible for some unexpected changes in

memory content. Support for a special break instruction will not be part of the initial design, since a sufficiently large number of breakpoint registers relieves the user from the need to rely on such an instruction. Should this assumption turn out wrong it will be easy to support a break instruction in the future. This extension would even be possible without amending the communication protocol of the debugging unit, since it anyway needs to inform the user about a change from the running to the halted state, when a normal breakpoint happens.

Two aspects remain to be discussed for this design area of the debug unit: if the breakpoint registers should hold physical or virtual addresses and the proper number of breakpoint registers. The preferred choice for the addressing scheme is the use of virtual addresses, since all core-internal address handling is done in that format. This allows the implementation of a breakpoint unit, which is integrated tightly with the core, but does not need to consider the details of the memory-access chain. And as far as the number of breakpoint registers is concerned, it could be learned from the examples in Section 2.2 that they provide four registers, which can be configured as either instruction or data breakpoints. This proved approach of sharing registers for instruction and data breakpoints is also suitable for the OpenProcessor system since it leaves the choice of how to utilize the provided resources to the debugging user. As for the actual number, the OpenProcessor platform has the advantage of running on reconfigurable hardware and thus can allow itself some flexibility. The debugging unit can be designed to support a range of breakpoint registers, with the exact number being specified only at synthesis time.

3.3. Accessing and Modifying System State

Orthogonal to the breakpoint support of the debugging unit discussed in the previous section is the design of its capabilities to access system state. External changes to the system state are theoretically even possible while the core is running, but this might result in unpredictable concurrency issues. Thus the debugging unit will be allowed to access system state only while the core is halted.

The system state can be seen in different categories, according to how central the saved state is to the core: On the one hand there is information stored directly in the CPU core, such as the program counter and the register file, and on the other hand there is external data that is available through the central bus, such as the main memory contents or device state. These categories area largely independent from each other and might pose different requirements for the debugging design, so each of them will be discussed in the following subsections.

3.3.1. The CPU Core and Register File

The CPU core provides the most non-uniform state compared to addressable memories like the main memory or the register file. While the CPU is halted, the core state is represented by the content of the pipeline registers together with some additional information, such as the execution mode and the state of the interrupt enable flag.

Since debugging needs only arise in situations where the software and/or hardware do not perform as expected, limited insight into the core state can make it hard or even impossible to track done bugs which are, e.g., rooted in a misunderstanding of pipeline-stage interactions. Thus, even though the bit count of the complete core state is very high, the design choice for the OpenProcessor debugging unit is to reveal the complete core data on the debugging interface.

Quite different from the scattered state in the pipeline registers of the core is the register file, which is organized in a RAM-like fashion. To allow full insight in the core, the debugging design requires the possibility to inspect the content of all registers while the core is halted.

Besides the decision on which data can be read through the debugging interface, it is also necessary to decide on the degree of write capabilities. Important for the debugging user is mainly the ability to see into the inner workings of the system, so that he can comprehend the system behavior. Additional write capabilities can further increase the usability of the debugging interface, since a user who is able to update the instruction pointer and the register contents has the freedom to try small attempts at bug-solving right on the running system. But those additional possibilities are not essential in enabling the user to track down bugs. Therefore, the first design of the OpenProcessor debugging unit will offer only capabilities to inspect core state, which allows for a more light-weight and thus quicker implementation.

There is only one exception where the debugging unit is required to influence core state: Since the process of single stepping through application code is most likely slow enough for interrupt requests to arise before each step, it is necessary to mask interrupts while the core is being debugged. The easiest design to achieve this solution is the introduction of an additional interrupt masking layer that is controlled through an extra input wire for the core. This allows the debugging unit to mask interrupts in all states except for when the core is running normally. The debugging of interrupt service routines is still possible with this approach by placing an instruction breakpoint on their entry points.

3.3.2. Memory and Bus Access

A larger portion of system state is the main memory external to the core. On the OpenProcessor platform main memory is accessed through a central system bus, which also allows the control of memory mapped devices. Figure 2.2 on page 4 shows a high-level view of the connection between the CPU core and the system bus. At the top of the memory-access chain all requests work with virtual addresses and are further down translated by the TLB to physical addresses. The request with the translated address is then served either by the caching modules or forwarded to the WISHBONE bus. The OpenProcessor specification defines no specific translation process from virtual to physical addresses, but leaves the details to the OS, which has direct control over the software-programmable TLB. This gives two possibilities for attaching the debugging unit to the WISHBONE bus with complementary advantages and disadvantages.

One option is to place the debugging unit at the top of the memory access chain, either

on the instruction or data path. This allows the debugger to work with virtual addresses and to observe all memory access results in exactly the same way as the currently running application sees them. An advantage of this scheme is the possibility to gain insight in those bug sources that cannot be observed directly at the WISHBONE bus, such as receiving outdated information from the cache. But the drawback of this approach is its limited access to only those memory regions that are currently mapped by the TLB. All other requests lead to TLB-misses and would require the debugging user to reprogram the TLB. Even though this is possible, this approach is unnecessarily complex and can only be done correctly if the user has sufficient knowledge about the memory mapping strategy employed by the OS.

Alternatively the debugging unit can be attached at the bottom of the memory access chain, as a separate master on the WISHBONE bus. This choice requires no interfering with the bus attachment of the core and thus promises an easier implementation. In contrast to the first approach this design presents physical addresses to the user and allows access to the whole address range without any trouble. Another benefit of this approach is the guarantee that all reading activities conducted during debugging leave the state of the TLB and the caches intact. The drawback of this design is the inability to see into the details of caching effects, and due to the use of physical addresses, this approach also requires the user to know about the details of address translation. Without that knowledge debugging of software that utilizes virtual addresses is not possible.

Both of the possible approaches have justification for inclusion as a part of the debugging unit and both can co-exist together to leave the final choice to the debugging user. But the initial design will only include the second option, a decision that is motivated by the wish for a quick implementation time and well suited to the current needs of OpenProcessor debugging. Since most of the OpenProcessor hardware is still undergoing development, the easy access to memory mapped devices that is provided by the second approach is likely to be in the debugging user's favor.

In contrast to the internal state of the core discussed earlier, this time it is appropriate to include not only read but also write capabilities in the design. Especially the debugging of memory-mapped hardware requires write access to allow proper testing. As the favored design attaches the debugging unit as a separate master to the WISHBONE bus, the implementation of writes can be done easily, since the bus runs unhinderedly even while the core is halted.

3.4. The State Machine of the Debugging Unit

Until now the detailed discussion mainly focused on designing the debugging functionalities. In addition the specification of a debugging control system is required, to expose the functionalities to the debugging user. As stated earlier, the favorable choice for the OpenProcessor system is an external debugging control, e.g. a debugging software on a standard PC, that connects to the OpenProcessor system.

To facilitate communication with the OpenProcessor debugging core both a connection interface and a protocol have to be decided upon. The development kit that currently

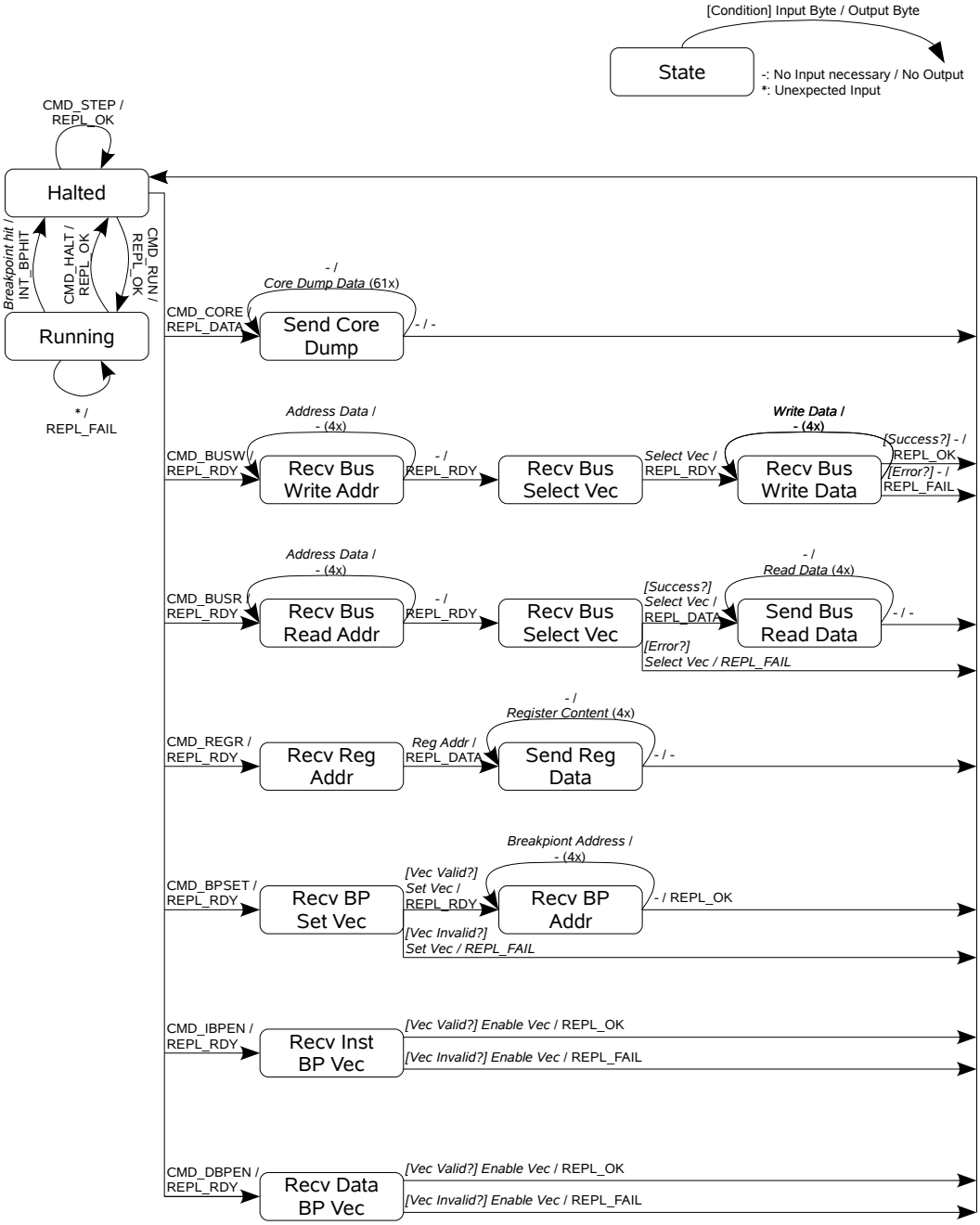

Figure 3.1: The debugging state machine and communications protocol, as seen by the user

hosts the OpenProcessor system offers two interface standards for the connection: Ethernet and RS 232. The benefits of Ethernet over RS 232 are a higher bandwidth and the ability to operate in a network of multiple machines. But those benefits increase implementation complexity, since the networking capabilities require a complex protocol stack. RS 232 on the other hand provides only a point-to-point connection and is fairly simple to implement. Its lower bandwidth poses no drawback for the debugging connection, since debugging commands and replies will be fairly small. Another fact that motivates the choice of RS 232 as the favored debugging connection is the availability of two RS 232 ports on the current development kit (one of which is implemented through an USB-RS 232 Bridge), as compared to only one Ethernet port. This allows the debugging unit to operate on a dedicated port and thus avoids the overhead for multiplexing a port between the OS and the debugging unit.

Based on the choice of RS 232 for the debugging connection, a byte-based communication protocol for sending debugging commands and receiving replies needs to be designed, since the RS 232 standard itself is byte-based. That protocol has to include commands that switch the OpenProcessor core between the running and the halted state and that allow the execution of further debugging operations, while the core is halted. Based on the requirements design above, the debugging system needs support for

- single stepping,
- a core dump command to reveal the internal state of the CPU core,
- read and write access to the WISHBONE bus,
- read access to the register file in the core, and
- commands to set and activate breakpoints on both instruction and data addresses.

For practical use, a communication protocol that supports all those commands, has been designed. The design stays close to a possible simple hardware implementation, both to allow the greatest flexibility in using the debugging unit and also to simplify its implementation. An abstract Finite State Machine (FSM) that models the a protocol is shown in Figure 3.1.

As can be seen from the figure, some commands require a number of parameters for their operation. Since the underlying connection is based on sending back and forth single bytes, a decision about the endianness had to be made. Due to the usage of x86-compatible systems, it has been decided that all data communication with the debugging unit will use little endian ordering, i.e. the least significant byte is transmitted first, which ensures easy compatibility with the host system of the debugging control software.

Two noteworthy details, which are not self explanatory from the state diagram, shall be discussed here.:

- The ISA of the OpenProcessor offers a variety of commands that deal with different word-sizes and signed/unsigned behavior. The proposed protocol however provides only one command for read access to the bus and one for write access, both expose the WISHBONE select vector to the user. This 4 Bit vector is intended to

select the active bytes of each 32 Bit operation, but its effective interpretation is currently chosen freely by some devices in the OpenProcessor system. While the chosen design path requires the user to know these internals of the WISHBONE specification, it also preserves the flexibility to deal with devices that might vote for non-standard interpretations of the select vector.

- The breakpoint set up is achieved through a total of three commands: one allows to configure the address in the breakpoint registers, the other two set the enable vectors for either the instruction or data breakpoints. All of the commands require a bit-vector as their first argument. When configuring the breakpoint addresses the vector acts as a one-hot selector of the register that should be configured; when enabling instruction or data breakpoints the vector directly controls which breakpoints are active. This design choice limits the number of available breakpoint registers to a maximum of eight, the number of bits in a byte. The debugging software is able to automatically discover the effectively implemented number of breakpoints in that range, since all breakpoint operations are designed to fail if and only if an out-of-range breakpoint is addressed.

4. Implementation of the OpenProcessor Debugging Unit

This chapter describes the implementation of a debugging unit for the OpenProcessor system that follows the design conceived in the previous chapter. As described earlier, the main component of the debugging system is a debugging controller inside the OpenProcessor system, which reveals the inner workings of the system through a debug interface.

Following an overview of the debugging unit, the first part of this chapter describes interesting details of the hardware implementation and evaluates its performance along the way to support the implementation decisions made. The second part of this chapter then presents a software implementation that handles communication with the debugging unit and presents a basic command line interface, which allows easy access to the debugging functionalities.

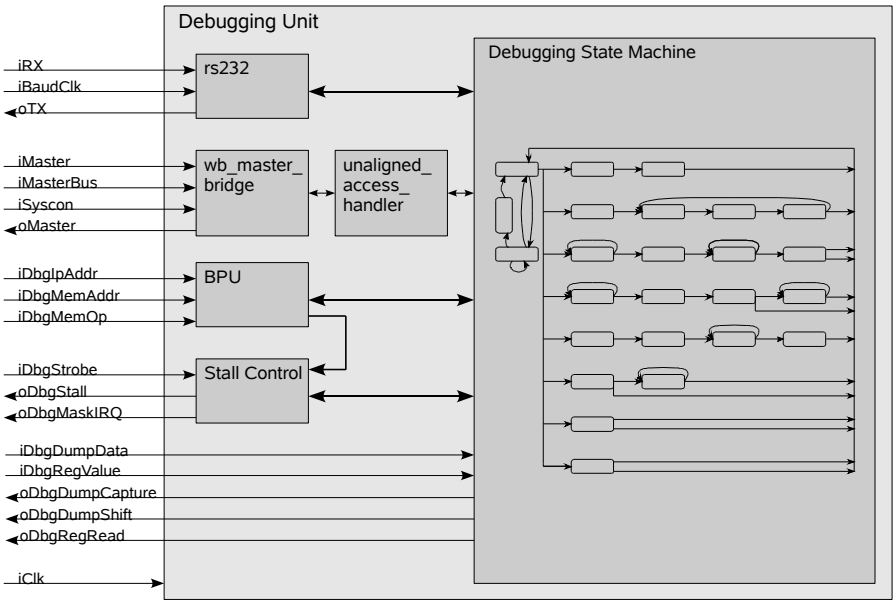

Figure 4.1: Overview of the newly implemented debugging unit

4.1. Overview

Figure 4.1 shows a high-level block-diagram of the implemented debugging unit. It is placed on the topmost hierarchy in the OpenProcessor system, where it connects to the CPU core, the central WISHBONE bus, and external pins for serial communication. The debugging unit reuses several components of the OpenProcessor system. The RS 232 controller already present on the WISHBONE bus is used to control the communication line, and the WISHBONE master bridge together with the unaligned access handler component handle bus accesses of the debugging unit. The RS 232 controller was re-used to reduce implementation work for the debugging unit, since it already provides a working low-level controller that translates the bit-wise connection to a byte oriented and buffered interface. The intention behind re-using the bus access components were to ensure total comparability between bus accesses by the debugging unit and the CPU core, so that the debugging user can be sure to see realistic data accesses while debugging. The further implementation of the debugging controller consists of a large synchronous state machine and—partly combinatorial—support logic that realizes a Breakpoint Unit (BPU) and supports stalling the core.

The central control of the debugging unit is realized by a state machine, which is in charge of the communication link to the debugging host. This state machine is closely related to the communication protocol, as seen from the user's point of view (Figure 3.1). Differences between the two stem from internal work which is sometimes necessary to process the debugging commands and required the addition of some extra states. Figure 4.2 documents the hardware implementation of the debugging state machine. The figure also documents the state identifiers that are stored internally in an eight bit state register and documents the byte-codes of the commands available in the communication protocol. States that are in addition to those in Figure 3.1 are shown with a dotted line.

A number of different implementations were evaluated with respect to their area and timing impacts. The evaluation results were achieved with version 9.2 of the ISE development suite, a set of FPGA design tools by Xilinx. Area usage was measured through synthesizing a design for the Xilinx XC4VLX60 FPGA. This FPGA offers 26,624 slices, which provide in total 53,248 flip-flops and 53,248 look-up-tables. Currently the OpenProcessor design is clocked at a main frequency of 50 MHz, which requires a period of 20ns or less after placing and routing the design. All final versions of the implementation are able to fulfill this constraint. The evaluation results for the initial OpenProcessor system without any debugging support are shown later on in this chapter in Table 4.1 next to the evaluations of the system with debugging support, which will be discussed in further detail in the following section.

4.2. Hardware Interface

During the implementation of the debugging unit, the CPU core of the OpenProcessor had to be augmented with a number of additional debug wires to reveal CPU internal state to the debugging unit. Those wires can be grouped into four independent categories

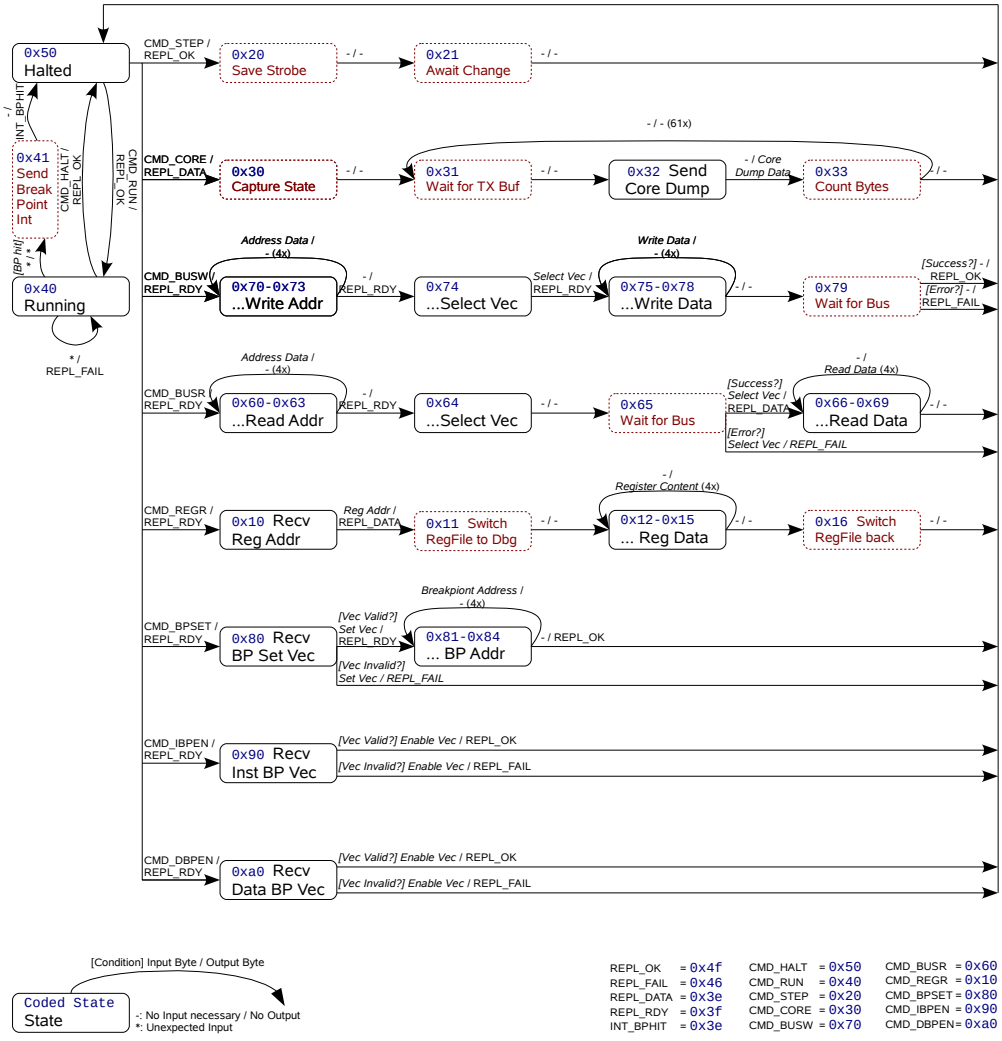

Figure 4.2: The extended FSM of the debugging unit, which is implemented in Verilog

according to the debugging functionality they support. The core and the debugging unit share three signals for stalling the core, three to supply information to the breakpoint unit, three to retrieve core dumps, and two signals for access to the register file. This section describes the implementation work for those four categories and states the reasons behind the chosen hardware design.

4.2.1. Stalling and Single-Stepping

As mentioned in the design section, stalling the CPU core of the OpenProcessor is easily achieved through overriding its pipeline stall wire asserting it to a logic 'high'. This functionality was implemented in the core, augmenting it with an external stall control. The same method was used on the internal interrupt masking logic of the core to allow interrupt masking by the debugging unit while in single-stepping mode.

Important for single-stepping is also the information about the end of a core cycle and the beginning of the next one. Since the core pipeline can be naturally stalled while waiting for a bus reply, a single core cycle can take up a varying number of clock cycles. This problem is solved by an informational strobe output of the core, which toggles at the beginning of every core cycle.

Figure 4.3 shows the relevant parts of the stall control in the debugging unit that deal with the signals mentioned above. The stall wire is driven by combinatorial logic to ensure prompt assertion if a breakpoint is hit. If breakpoint information had to travel through the state machine first, than the instruction that caused the breakpoint would already have advanced by one pipeline stage before the core gets halted. The execution of a single step of the halted core is still controlled by the state machine and happens in two phases. First the current value of the strobe wire is stored in a flip-flop, then the stall wire is reset until the strobe value changes. This change reasserts the stall wire, which is again done by combinatorial logic to ensure the core executes only a single step.

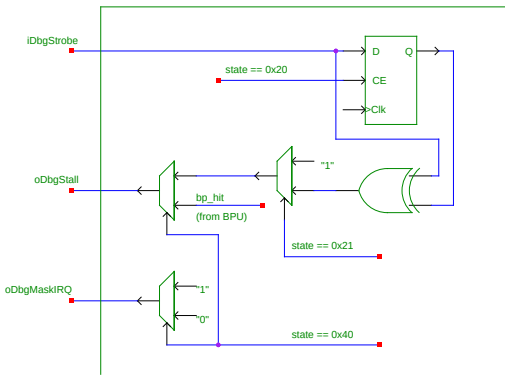

Figure 4.3: Stall logic for debugging

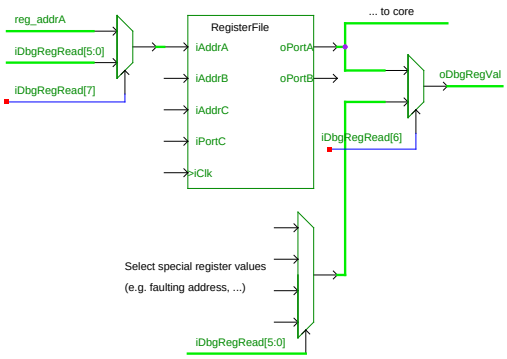

Figure 4.4: Debug access to the core registers

4.2.2. Breakpoint Support

Breakpoint support in the debugging unit was implemented in a straightforward manner. To inform the debugging unit about the currently used addresses, the core was augmented with signals that extract the current instruction address at the IF/ID pipeline boundary registers as well as the data memory address at the ID/EX pipeline registers. Through directly accessing pipeline registers, no extra logic is required to make the information available to the debugging unit. Information about an ongoing access could be gained by monitoring a strobe signal on the core-memory interface. Instead, to keep the debugging implementation separate from the memory connection of the core, a third signal was introduced that informs the debugging unit about the intention of the core to perform a data memory access.

Based on a Verilog pre-processor definition the debugging unit instantiates between one and eight registers to store breakpoint addresses. The presence of a breakpoint condition is then determined by combinatorial comparators. Table 4.1 shows the total costs of the OpenProcessor system with a varying number of breakpoints available. Each additional breakpoint register increases the costs by about 40 slices. This increase is justified, considering that each breakpoint requires at least 32 flip-flops to store its 32-bit address and 32 LUTs to realize comparators for both the instruction address and the data address.

4.2.3. Retrieving Core Dumps

The main goal for the core dump design is the possibility to get as much insight as possible into the core state. The first consequence of this goal was to increase information that is tracked by the core: Initially the core was designed so that later pipeline stages only carry decoded instruction information in their register, which gives no information about the original instruction word that led to the configuration of each pipeline stage. To give the debugger access to the instruction word of each pipeline stage the core was amended to also store an unmangled op-code in each pipeline stage.

In addition to the instruction in each pipeline stage, the debugging unit should also show detailed status for the ALU, the memory interface, and the internal control registers. This requires the extraction of a relatively large amount of data from the core—in the current implementation a core dump consists of about 60 bytes of information.

A naïve approach towards communicating this information from the core to the debugging unit would be to just augment the core with enough wires that reveal the desired information. But this implementation seems unwise considering the amount of information involved. The number of wires would put an enormous burden on the signal routing between the core and the debugging unit. Also the byte-wise data transmission of the debugging unit would result in a large multiplexer if one naïvely selects arbitrary bytes of the data.

To simplify synthesis, the final design was implemented in a way inspired by the JTAG boundary-scan technology. Similar to the boundary-scan shift register, a core dump shift register that is large enough to hold a complete core dump was inserted into the core—

Configuration	Area			Timing
	Total Slices	Flip-Flops	LUTs	Min Period
without Debug Unit	4,494	3,653	8,472	< 20ns
Debug Unit, 1 Bp	5,307	4,550	10,038	—
Debug Unit, 2 Bp	5,351	4,581	10,109	—
Debug Unit, 4 Bp	5,433	4,662	10,148	< 20ns
Debug Unit, 8 Bp	5,556	4,800	10,340	< 20ns
Debug Unit, DumpMux (4 Bp)	5,231	4,168	9,803	26.784ns
Debug Unit, Extra RegPort (4 Bp)	5,438	4,662	10,141	—

Table 4.1: Evaluation of the area and timing performance for configurations of the OpenProcessor system without the debugging unit, with the debugging unit configured for 1, 2, 4, or 8 breakpoints, and with unfavorable implementation variants for core dumps and register access.

this is basically 8 shift registers of length 60 Bits each. Two control wires allow the debugging unit to load the registers with the current core state and to shift the data byte-wise. Now only a single byte of information has to be routed from the core to the debugging unit and no further multiplexer complexity is necessary.

The necessity of the additional flip-flop costs to realize the core dump shift register is proved by a comparison between the naïve approach and the final implementation. Table 4.1 shows that the naïve approach saves about 200 slices of area but no longer fulfills the minimum period requirement of 20ns, the minimum period is extended by about a third, thus limiting the maximum frequency to about 38.6 MHz.

4.2.4. Reading Register Values

The register file that is used by the OpenProcessor core is implemented in dual-ported Block RAM modules (*BRAM*), which are part of the Xilinx FPGA. Initially the core used two BRAMs to simultaneously allow two read ports and one write port. The implementation of debug access to the register file left two possible choices. Either the register file could be augmented by a third read port (requiring an additional BRAM) or one of the existing ports could be shared by the core and the debugging unit (cf. Figure 4.4). The final implementation uses the second option, after both options had been evaluated. Table 4.1 shows that an extra port on the register file, although theoretically removing the need for multiplexing the port access, does not save any slices. Thus the multiplexer implementation is preferable since it does not consume an extra BRAM.

4.3. Software Interface

The completed hardware implementation of the debugging controller offers its functionality through an RS 232 communication interface. Theoretically, this allows debugging without the need for any specialized software. And indeed, the initial testing of the hardware implementation was done using just a regular terminal application. This test

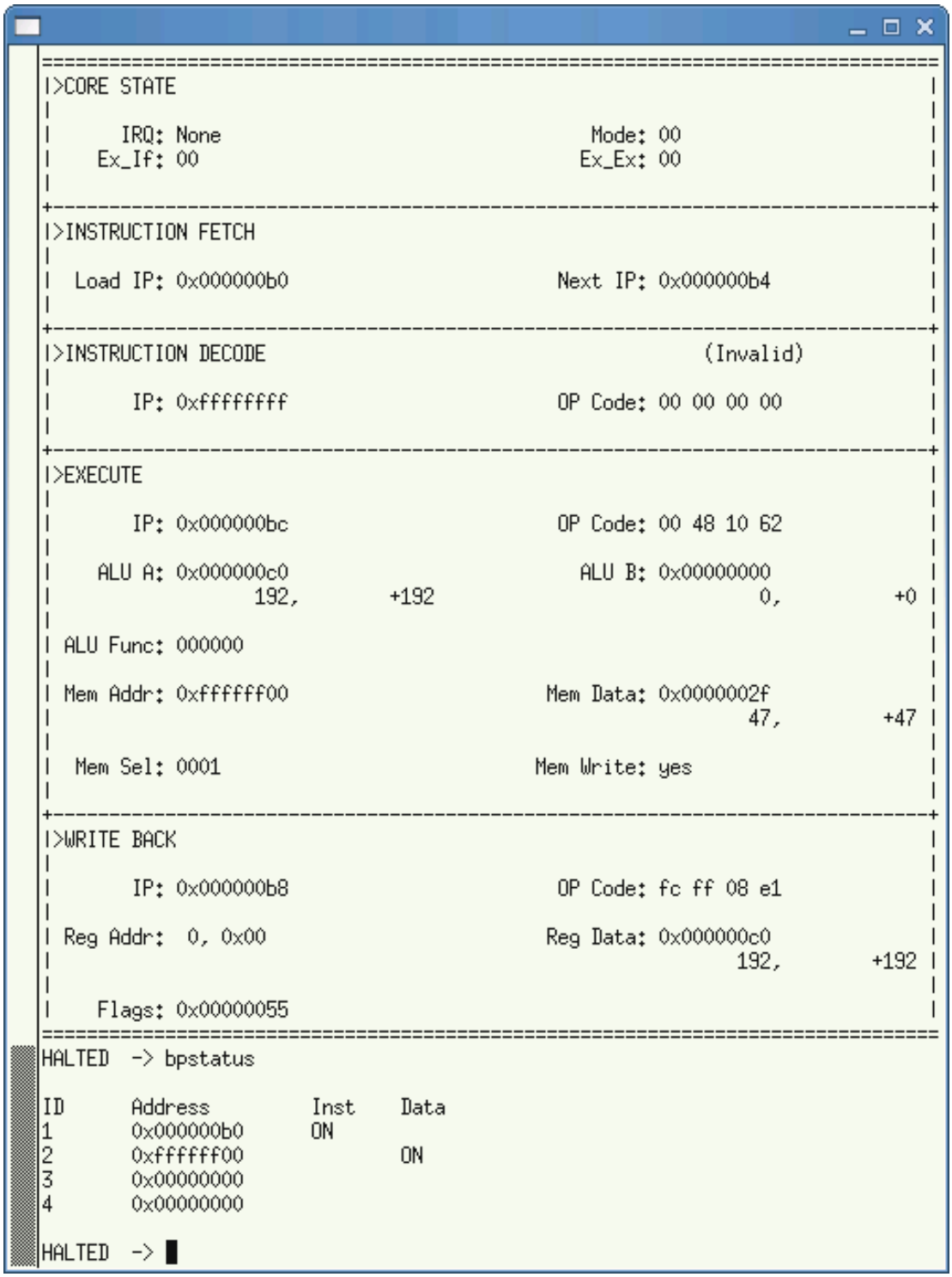

Figure 4.5: Screenshot of the debugging CLI showing the core dump and breakpoint features

method is supported by the binary representation of all commands and replies that are used to halt and single-step the OpenProcessor core, all required bytes for those commands fall in the printable range of the ASCII character set.

Later on, a Command Line Interface (*CLI*) software was developed, to make the full functionality of the debugging controller available to the user. The software is written in the C programming language and currently available for the Linux operating system. The CLI offers access to all the commands provided by the debugging core and furthermore allows comfortable breakpoint management and pretty-printed core dumps. Figure 4.5 shows a screenshot of the CLI in use.

The following subsections first give some implementation details of the multi-threaded software and later on document the complete user interface of the program.

4.3.1. Thread Structure of the Implementation

Most parts of the CLI code base are fairly simple, since most commands of the debugging unit are directly translated to commands in the CLI software. To provide a user-friendly interface, the CLI software uses the `readline` library for user input—a command line input library that provides tab-completion and a command history.

Due to the breakpoint support of the debugging unit a communication problem arises for the debugging software, which needs to simultaneously listen to two inputs: on the

```
struct DBGCMD {
      int   cmd;
      char  args[...]
      int   recv_char;
};
struct DBGCMD     next_cmd;         int         cli_done;
mutex             next_cmd_mutex;   mutex       cli_mutex;
condition         next_cmd_sleep;   condition   cli_sleep;
```

Main

```
lock(next_cmd_mutex);
  Initialize threads
  while (true)
    cond_wait(next_cmd_sleep)
    if (next_cmd.recv_char)
      print Interrupt message;
      next_cmd.recv_char = 0;
    if (next_cmd.cmd)
      select (next_cmd.cmd)
        case 1:
          ...
          send to debug unit
          while (!next_cmd.recv_char)
            cond_wait(next_cmd_sleep)
          process result
          next_cmd.recv_char = 0;
          ...
        ...
        case n: ...
      next_cmd.cmd = 0;
      lock(cli_mutex)
        cli_done = 1;
        signal(cli_sleep);
      unlock(cli_mutex);
```

Debug Unit Listener

```
while (true)
  getchar;
  lock(next_cmd_mutex);
    next_cmd.recv_char = ...;
    signal(next_cmd_sleep);
  unlock(next_cmd_mutex);
```

Command Line Listener

```
while (true)
  readline;
  if (local command?)
    process locally
  else
    /* debug unit command */
    lock(next_cmd_mutex);
      next_cmd.cmd = ...;
      next_cmd.args = ...;
      signal(next_cmd_sleep);
    unlock(next_cmd_mutex);
    lock(cli_mutex);
      while (!cli_done);
        cond_wait(cli_sleep);
      cli_done = 0;
    unlock(cli_mutex)
```

Figure 4.6: Pseudo code structure of the implemented debugging software

RS 232 connection the debugging unit might send a breakpoint interrupt message and on the command line the debugging user might enter some commands for execution. A standard way to handle such multiplexed I/O operations on Linux systems is the `select` call, which allows a program to wait on multiple sources for input. But due to the usage of blocking calls to the `readline` library, which do not easily work with the usage of `select`, a multi-threaded implementation approach was chosen.

Figure 4.6 gives an overview of the interlocking between the involved threads in pseudo code. Both communication partners, the user and the debugging unit, are handled in separate working threads. The synchronization between the working threads and all software-initiated communication is handled in the main thread. Both workers use a shared global mutex (`next_cmd_mutex`) to signal the main thread a received message: in case of the command line thread this corresponds to a complete command that was entered by the user, in case of the debug unit thread a message is a single received byte. When a new command was entered by the user, the command line thread is blocked on a second mutex (`cli_mutex`), until the main thread signals that it is done processing the current command.

This setup allows the main thread to always keep keep book of the expected internal state of the debugging unit. Since interrupt messages by the debugging unit (namely the "breakpoint hit" interrupt) are only sent if the debugging unit is in the "run" state and never while a command is being processed, the main thread can correctly categorize received bytes as either expected replies to a command or as breakpoint interrupts.

The presented solution to serialize both user requests and debugging interrupts is independent from the command line nature of the implemented software. Thus the general design of Figure 4.6 is suitable for other implementations as well, e.g. to realize a graphical interface for debugging.

4.3.2. Documentation of the User Interface

The debugging command line interface `dbgcli` offers access to all functionality of the OpenProcessor debugging unit. It takes a single optional parameter, the path to the serial devices that connects to the OpenProcessor system—if none is supplied a compile time parameter is used.

On startup `dbgcli` tests for a valid connection to the OpenProcessor system, discovers the number of breakpoints available, and whether the system is currently stalled or running. The following list documents the available commands of the software; all address and data parameters can be supplied both in decimal or hexadecimal notation, breakpoints are numbered starting from one:

run
: Switch from halted mode to running the core. The system will interrupt if a breakpoint is hit.

halt
: Halt the execution of a running core. This allows the invocation of further debugging commands.

step `[#cycles]`
Execute a single (or `#cycles`) steps on a halted system.

dump, synonym: **core**
Print a formatted dump of the current core state. If the core is not halted, it will be stopped while the dump is being retrieved.

buswr `#bits addr value`
Write the supplied value to the given address on the WISHBONE bus. `#bits` has to be one of 8, 16 or 32 and will decide which select vector is sent on the bus.

busrd `#bits addr`
Send a read request with the given address to the WISHBONE bus. `#bits` determines the select vector as above. The received value is truncated to the given width before output.

bpset `id addr`
Store the address in breakpoint register `id`.

bpon `i|d id`
Enable instruction (`i`) or data (`d`) breakpoints for the address stored in register `id`.

bpoff `i|d id`
Disable instruction (`i`) or data (`d`) breakpoints for the address stored in register `id`.

bpstatus
Print the current breakpoint configuration.

regrd `regnum`
Read the value of the given register. `regnum` is used as in the ISA, counting from zero.

5. Conclusion and Future Work

During the course of this study thesis a solution for software-debugging on the OpenProcessor system has been developed. Based on a survey of some representative debugging examples ranging from low-level hardware debugging to self-contained systems in personal computers, an approach for the OpenProcessor system was deduced that aimed for a micro controller-like software-debugging system. The further details of the system were designed and implemented in the OpenProcessor sources and now enable the user—together with a PC-based debugging software—to examine software execution on the OpenProcessor system.

The presented solution makes it possible for the first time to see details of the "real" OpenProcessor system while executes directly on the target hardware. Prior to this work, only simulation allowed detailed insight. The new enhancement of the system promises better and faster software development for the OpenProcessor since it also enables the developer to observe program phases that cannot be reached in simulation, due to a slowdown in simulation of over four orders of magnitude. But there is also a price one has to pay for real-time debugging. While a simulation allows insight to the complete state of the system, the debugging unit only provides insight to a part of the state. The part that is revealed by the designed debugging functionalities is very similar to the information existing software-debugging systems offer, but the usefulness for the OpenProcessor system still has to be proved in practice.

The current implementation of the debugging unit also leaves room for future work to further increase the debugging possibilities. One possibility is to amend the system by those debugging features that were intentionally left out in the initial design, i.e. memory access that involves caches and the TLB and extended write-capabilities, e.g. to the register file. While that functionality is not necessarily required for debugging, the OpenProcessor developers might ask for it to make software-development more comfortable.

The software component of the debugging system also leaves a lot of room for improvements. Currently a command line interface allows easy access to the state of the OpenProcessor but the retrieved data is not matched with the source code of the currently executing software, thus putting the additional burden on the developer to know his assembler code. Future work has to aim at the development of a tighter integrated development environment, that allows in-place debugging of the code written.

A. Schematic of the OpenProcessor Core

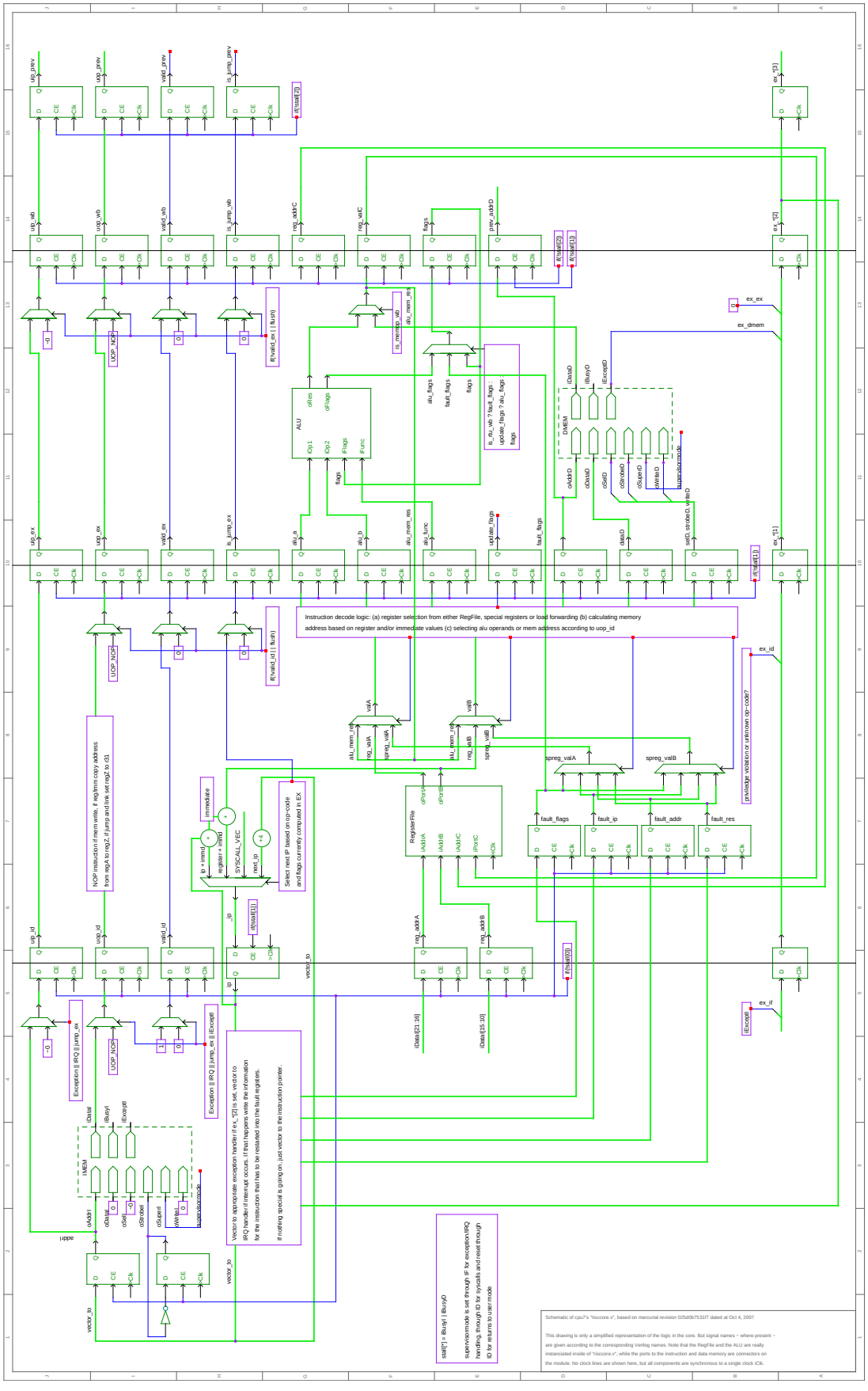

Figure A.1: Schematic of `risccore.v` in the initial OpenProcessor code

Bibliography

[1] Atmel. *8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash*. August 2007. Rev. 2467P-AVR-08/07.

[2] Microsoft Corporation. *Hard-Coded Breakpoint Is Encountered in GetARPBuffer*. `http://support.microsoft.com/kb/304245`, February 2007. Microsoft Knowledge Base: Q304245.

[3] IEEE. *Standard Test Access Port and Boundary-Scan Architecture*. 2001. IEEE Std 1149.1-2001.

[4] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Volume 3B: System Programming Guide, Part 2. November 2007.

[5] MemecBoard. *Virtex-4 MB Development Board User's Guide*. Calgary, Alberta, Canada, December 2005.

[6] Raphael Neider. *OpenProcessor v1*. University of Karlsruhe.

[7] Richard Herveille (Steward). *Specification for the WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. OpenCores Organization, September 2002. Revision B.3.

[8] Xilinx. *Virtex-4 User Guide*, March 2006.