FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

ASCARTS

Design of an Asynchronous Processor using a High-Level Specification Language

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Technische Informatik

eingereicht von

Claudia Hermann
Matrikelnummer 0125532

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Mitwirkung: Dipl.-Ing. Dr.techn. Jakob Lechner

Wien, 21. Jänner 2016

Claudia Hermann

Andreas Steininger

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

ASCARTS

Design of an Asynchronous Processor using a High-Level Specification Language

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Computer Engineering

by

Claudia Hermann
Registration Number 0125532

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Assistance: Dipl.-Ing. Dr.techn. Jakob Lechner

Vienna, 21st January, 2016

Claudia Hermann | Andreas Steininger

Erklärung zur Verfassung der Arbeit

Claudia Hermann
Am Haanbaum 3/7, 3001 Mauerbach

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 21. Jänner 2016

Claudia Hermann

Kurzfassung

Das Ziel dieser Arbeit ist die Entwicklung des asynchronen Prozessors ASCARTS basierend auf dem Befehlssatz des synchronen Prozessors SCARTS. Für die Umsetzung des Prozessordesigns wird das Open-Source Framework Balsa verwendet, welches die Modellierung asynchroner Schaltungen auf einer abstrakten Ebene durch Verbergen der eigentlichen Handshake-Implementierung erlaubt. Die Benutzerfreundlichkeit der Balsa-Sprache sowie der zugehörigen Werkzeugkette für einen komplexen Schaltungsaufbau ist zu bewerten. Ist das Balsa-Framework ausgereift genug um einen ganzen asynchronen Prozessor zu entwerfen und zu synthetisieren? Ein weiteres Ziel dieser Arbeit ist es, die Unterschiede in der Prozessorarchitektur von synchronen und asynchronen Prozessoren mit identischen ISAs zu identifizieren. Daten- und Steuerkonflikte, die durch die asynchrone Pipeline-Architektur verursacht werden, erfordern innovative Lösungen. Datenkonflikte werden mithilfe eines asynchronen Weiterleitungsmechanismus, basierend auf gespeicherten Informationen über vorhergehende Befehle, vermieden. Um Steuerkonflikte zu lösen wird ein Farbalgorithmus, ähnlich zu den in AMULET1 und SAMIPS verwendeten, implementiert. Um die Prozessorschnittstelle zu herkömmlichen synchronen Speichern zu verbinden, werden die Handshake-Signale in synchrone Signale transformiert. Das Balsa-Design wird mit der quasi delay-insensitive four-phase dual-rail Handshake-Implementierung synthetisiert. Die synthetisierte Netzliste wird auf die UMC 90 nm Technologie abgebildet, auf der Timing-Simulationen durchgeführt werden. Da Synchronisation über das Handshake-Protokoll erreicht wird, passt sich das Design an die tatsächlich vorherrschenden Betriebsbedingungen an. Das führt zu variablen Average-Case-Ausführungszeiten der einzelnen Befehle. Daher ist die Echtzeitfähigkeit ohne genaue Worst-Case-Analyse nicht klar ersichtlich. Die Timing-Simulationsergebnisse von ASCARTS mit Average-Case-Corner Synthese werden mit den Ergebnissen von SCARTS mit Worst-Case-Corner Synthese verglichen. Ein zentrales Ergebnis der vorgestellten ASCARTS Implementierung ist die Prozessorbeschreibung selbst, geschrieben in einer High-Level-HDL, die unabhängig von der tatsächlichen Handshake-Implementierung ist. Daher könnte die Balsa-Beschreibung von ASCARTS nützlich für zukünftige Forschungsarbeiten an unterschiedlichen asynchronen Implementierungsstilen sein.

Abstract

The purpose of this thesis is the development of the asynchronous processor ASCARTS based on the instruction set of the synchronous processor SCARTS. For the realization of the processor design the open-source framework Balsa, which allows modeling asynchronous circuits at an abstract level by hiding the actual handshake implementation, is used. The usability of the Balsa language as well as the associated toolchain for a complex circuit design is to be evaluated. Is the Balsa framework sophisticated enough to design and synthesize an entire asynchronous processor? A further objective of this thesis is to identify the differences in the processor architecture between synchronous and asynchronous processors with identical ISAs. Data and control hazards caused by the asynchronous pipeline architecture require innovative solutions. Data hazards are avoided by an asynchronous forwarding mechanism based on information stored about previous instructions. To resolve control hazards, a coloring algorithm, similar to the one used for AMULET1 and SAMIPS, is implemented. To connect the processor interface to conventional synchronous memory, the handshake signals are transformed to synchronous signals. The Balsa design is synthesized using the quasi delay-insensitive four-phase dual-rail handshake implementation. The synthesized netlist is mapped to UMC's 90 nm technology on which timing simulations are conducted. As synchronization is achieved via the handshake protocol the design adapts to the actual prevailing operating conditions. This results in variable, average-case execution time of individual instructions. Consequently, real-time capability is not evident without an accurate worst-case analysis. The timing simulation results of ASCARTS with Average-Case-Corner synthesis are compared to the results of SCARTS with Worst-Case-Corner synthesis. A key outcome of the presented ASCARTS implementation is the processor description itself, written in a high-level HDL, which is independent of the actual handshake implementation. Therefore, the Balsa description of ASCARTS might be useful for future research of different asynchronous implementation styles.

Contents

Contents xi

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Contribution . . . 3
- 1.3 Outline . . . 4

2 Background 5
- 2.1 Fundamentals of Asynchronous Logic . . . 5
- 2.2 Balsa . . . 14

3 State of the Art 21
- 3.1 AMULET1 . . . 21
- 3.2 AMULET2 . . . 22
- 3.3 AMULET3 . . . 23
- 3.4 SAMIPS . . . 23
- 3.5 Asynchronous MIPS R3000 . . . 23
- 3.6 Lutonium . . . 24

4 SCARTS - Basis For The Asynchronous Processor 25
- 4.1 Processor Architecture . . . 25
- 4.2 Extension Modules . . . 28
- 4.3 Instruction Set Architecture . . . 31

5 ASCARTS - The Asynchronous Processor 33
- 5.1 Processor Interface . . . 34
- 5.2 Processor Architecture . . . 34
- 5.3 Extension Modules . . . 43
- 5.4 Instruction Set Architecture . . . 46

6 Design Flow 51
- 6.1 Behavioral Simulation . . . 51
- 6.2 Synthesis . . . 53
- 6.3 Structural Simulation . . . 54

6.4 Design Implementation 56
6.5 Timing Simulation 57

7 Results **59**
7.1 Resource Usage 59
7.2 Performance 60

8 Conclusion and Outlook **65**

A Test Programs **67**

B Instruction Set **71**
B.1 Overview 72
B.2 Notation 76
B.3 Instructions 76

Bibliography **103**

CHAPTER 1

Introduction

Embedded systems play an important role nowadays. They can be found in many different application fields solving different tasks like monitoring, controlling and processing signals. In the last couple of years especially portable devices gained in importance. The powerful smartphones and tablets are getting an every day part in our lives. The more powerful smartphones as well as tablets are getting more and more popular. The processors of these devices are getting faster, more efficient and more flexible to handle a major variety of tasks. At the same time they are supposed to be more energy-efficient so that the run-time of the batteries does not decrease.

Current processor development is based on synchronous and asynchronous design principles. Synchronous circuit designs rely on a global periodic signal, called clock, to synchronize the data flow. With every clock pulse the results of the combinational logic of all components on the chip are stored in registers, as illustrated in Figure 1.1.

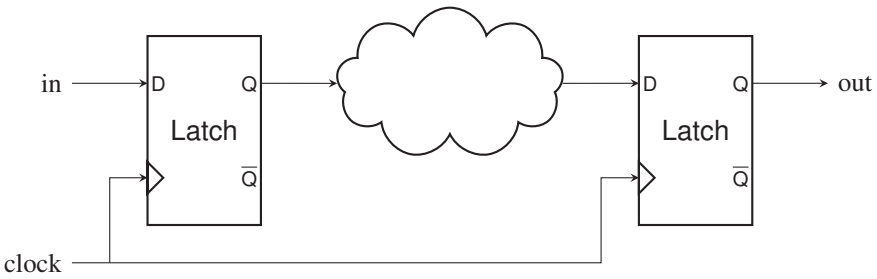


Figure 1.1: Synchronous Circuit

The frequency of the clock has to be chosen in such a way that on all paths signal changes can be propagated in one cycle. The maximum possible clock frequency is determined by the path with the longest propagation delay, the critical path. To accomplish that in any environment, the chip has to be designed for worst-case scenarios, such as worst-case temperature, low voltage supply or process variations.

Conversely asynchronous designs are self-timed. Clock signals are replaced by handshakes to control the data flow between adjacent components. Each component signals a request once there is new data to be processed. Likewise it sends an acknowledge signal once it has finished processing the data and is ready to accept new one. See Figure 1.2 for an illustration.

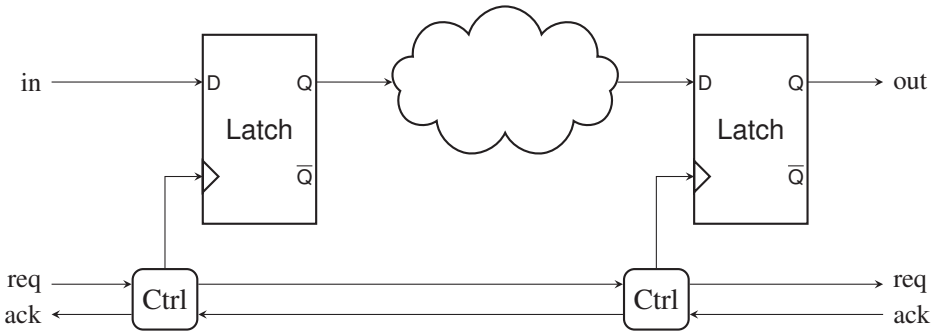


Figure 1.2: Asynchronous Circuit

1.1 Motivation

Synchronous processors dominate the market nowadays. The reason for this dominance is that synchronous designs are simple and well-understood. In addition, all industrial-strength computer aided design tools for design and verification target clocked circuits. Both facts lead to fast development and short design cycles of synchronous processors. In recent technologies, however, the global time assumption becomes a performance and power limiting issue. The different characteristics of asynchronous circuits can be exploited to design circuits which overcome the problematic factors of synchronous circuits [18].

One big disadvantage of synchronous designs is the global clock network. On one hand the network occupies a huge amount of area on the chip. On the other hand it takes a lot of effort to distribute the clock in a way that the skew between different clock signals is minimal. The faster the clocks are, the more difficult it is to route the wires properly, as the margin gets even smaller. On each clock transition the clock network consumes a large amount of power, no matter if the processor is performing useful computations or is in idle state. Furthermore, as all computations start at the clock edge, the most current is drawn at this time and shortly thereafter. This leads to high electromagnetic interferences at the clock frequency. However, in asynchronous designs there is no global clock signal to be distributed. The local clock pulses, derived from handshake signals, are only generated where and when they are needed. Hence, the power and electromagnetic dissipation is lower and the current consumption is more uniformly distributed without spikes, as the signal switching of the individual components is not correlated.

In synchronous designs the operating speed is determined by the critical path. Mapped to processors the operating frequency is as slow as the slowest instruction or stage in pipelined designs, respectively. In contrast, the asynchronous counterparts start immediately with a new computation once the previous one is finished. So the speed of such processors varies over

time and is only as slow as the current instruction or the slowest stage of the current instruction in pipelined designs, respectively. For correct operation in all conditions worst-case timing assumptions have to be made in synchronous designs. This leads to idle times in optimal conditions. However, due to the fact that a handshake protocol is used for synchronization, asynchronous designs adapt to the actual prevailing operating conditions. Because of their insensitivity to delays and their simple handshake interface, asynchronous circuits are rather modular and, hence, have a better composability.

Although replacing synchronous logic by asynchronous logic opens up ways to further improve processor performance, omitting the clock has some drawbacks. The asynchronous control logic induces overheads in area, which again leads to increased power consumption. Besides, the more complex data paths of asynchronous logic have a negative impact on the operating speed.

The complex structures of asynchronous circuits make it more difficult to design and harder to verify or debug them. As there are not any strategies to efficiently develop asynchronous designs there is lack of computer aided design tools, and even the few available ones are not well maintained. The missing tool chain and the extensive know-how required to design such circuits lead to the fact that only few people exist who are able to develop asynchronous processors.

1.2 Contribution

The objective of this thesis is the development of the asynchronous processor ASCARTS. It is based on the instruction set of the synchronous processor SCARTS [3]. So far only a design tool was developed at *Vienna University of Technology*, *Department of Computer Engineering*, which transforms a conventional synchronous circuit description into an asynchronous circuit [12]. However, the resulting asynchronous circuit, generated from the SCARTS design, is not satisfactory. Hence, the manual design of an asynchronous counterpart was demanded.

The development of the processor without any tools implies the design of the underlying handshake protocol as well. To be able to concentrate on the actual processor design, a tool which hides the handshake circuits is used. The decision was made in favor of the open-source project Balsa [4]. It is a high-level synthesis system for asynchronous circuits. A feature of Balsa is that the development of asynchronous circuits is abstracted from the handshake implementation. The actual handshake implementation can be selected during synthesis. This way, the design can be synthesized with different realizations of the handshake protocol. Thus, the effects on the performance with the individual protocols can be observed. Finally a gate-level netlist is generated from the high-level description. This netlist is mapped to UMC's 90 nm technology[1].

In addition, by designing ASCARTS with the aid of Balsa the usability of such tools for developing asynchronous circuits is to be evaluated.

[1] http://freelibrary.faraday-tech.com/

1.3 Outline

The second chapter is divided into two parts. First the fundamentals of asynchronous logic are explained. The focus lies on the different ways handshakes can be realized. The second part covers an introduction to Balsa. In the third chapter important asynchronous processors already developed are presented. In the fourth chapter the architecture of SCARTS is summarized. Chapter five gives detailed information about the ASCARTS architecture. Implementation details of the individual units are revealed as well. In chapter six the steps of the design flow are described. Several stages are passed through to obtain the final design description of ASCARTS. In the design process simulations at three different levels are performed. Results achieved from simulations are presented in chapter seven. Finally the findings are summarized and an outlook to future work is given.

CHAPTER 2

Background

2.1 Fundamentals of Asynchronous Logic

In synchronous circuits a global clock defines points in time where signals are valid and stable. In asynchronous systems control signals have to be valid all the time. Each transition changes the state of the circuit. To avoid hazards and races, additional control logic is needed for synchronization, communication and sequencing of operations [18].

2.1.1 Handshake

Handshaking is used between neighboring components for synchronization or to agree on exchange of data. In general the active party of a handshake-channel initiates the handshake by a *request* signal. The passive party completes the handshake by an *acknowledge* signal.

It can be distinguished between different types of channels depending on the direction of data or if data is transmitted at all.

Push Channel

If the sender is the active party, the sender indicates the capture condition for the receiver with the request signal. Likewise the receiver indicates the issue condition for the sender with the acknowledge signal. The validity of the data is denoted by the request signal. A push channel is illustrated in Figure 2.1a.

Pull Channel

If the receiver is the active party, the receiver indicates the issue condition for the sender with the request signal. Likewise the sender indicates the capture condition for the receiver with the acknowledge signal. The validity of the data is denoted by the acknowledge signal. A pull channel is illustrated in Figure 2.1b.

Sync Channel

Channels without a data path can be used for synchronization. A sync channel is illustrated in Figure 2.1c.

Bidirectional Channel

A channel where data is transmitted in both directions is illustrated in Figure 2.1d. The data passing from the active party to the passive party acts as a push channel. Hence, the validity is denoted by the request signal. The data passing from the passive party back to the active party acts as a pull channel. Hence, the validity is denoted by the acknowledge signal.

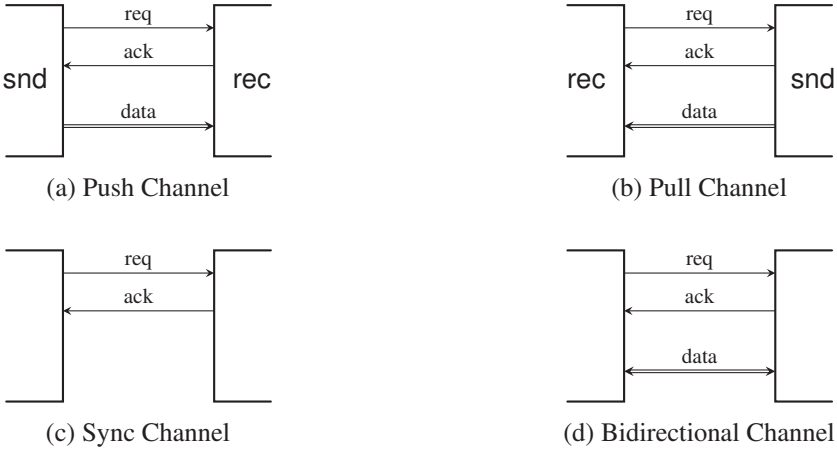


Figure 2.1: Handshake Channels

The handshake establishes a closed loop control for the data flow between two parties. New data is only issued if the previous data is acknowledged and, therefore, captured. On the one hand this makes the circuit timing adaptive. On the other hand this is not fault tolerant as deadlocks occur if data gets lost.

2.1.2 Handshake Styles

The request signal determines the start of one communication cycle, whereas the acknowledge signal determines the end of one communication cycle. There are two approaches how to implement the control protocol. These are presented on the basis of a push channel.

Four-Phase Protocol

The four-phase protocol uses Boolean levels on the request and acknowledge wires to encode the state of the communication cycle. The term *four-phase* refers to the number of edge transitions on the control wires during one protocol cycle. Figure 2.2a illustrates the four phases of the four-phase protocol for a push channel.

First the sender issues new data and sets request to high. Second the receiver captures the data and sets acknowledge to high. This indicates that data is not needed anymore. In response the sender sets request to low. Data might not be valid anymore. Finally the receiver sets acknowledge to low. A new communication cycle can be initiated by the sender.

This protocol uses level signaling for the control signals. The state of the signals is always known without knowledge of the history. However, the advantage of level signaling comes at the cost of unnecessary time and energy for the return-to-zero transitions of the signals.

Two-Phase Protocol

In the two-phase protocol each transition on the request and acknowledge wires, irrespective if it is a rising or falling edge, represents a signal event. The term *two-phase* refers to the number of edge transitions on the control wires during one protocol cycle. Figure 2.2b illustrates the two phases of the two-phase protocol for a push channel.

First the sender issues new data and sets request to high. Second the receiver captures the data and sets acknowledge to high. This indicates that data is not needed anymore. At this point data might become invalid. A new communication cycle can be initiated by the sender.

This protocol uses transition signaling for the control signals. The state of the signals is not representative for the state of the handshake protocol. The disadvantage of transition signaling is the more complex implementation as state-holding elements are needed to store the current signal states.

(a) Four-Phase Protocol (b) Two-Phase Protocol

Figure 2.2: Single-Rail Protocol (Push Channel)

2.1.3 Design Techniques

To build control circuits with transition signaling, logic elements that perform logical operations on events are necessary.

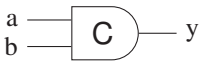
Or Element

The Boolean `xor` gate acts as the `or` element for events. When either input of the `xor` gate changes its state the output also changes its state. Thus, an event received on any input of the `xor` gate triggers an event on the output. For more than two inputs the `parity` function acts as the `or` element for events.

Muller C-Element

There is no Boolean function that performs an `and` operation on events. The `and` gate and the `or` gate are asymmetric functions regarding the logical level. An `or` gate immediately propagates a rising transition on a single input to its output, without waiting for a transition on its second input. Likewise, the `and` gate immediately propagates a falling input transition. Thus, a gate is needed that behaves like an `and` gate for rising transitions and like an `or` gate for falling transitions.

Building an `and` element of combinational gates is not an option. The *Muller C-Element* [16] is a sequential gate and acts as the `and` element for events. The symbol and the truth table are illustrated in Figure 2.3.



a	b	y
0	0	0
0	1	no change
1	0	no change
1	1	1

Figure 2.3: *Muller C*-Element

The *Muller C-Element* is designed in a way that the output only changes its level if both inputs of the element have the same logical level. If the levels differ at the inputs, the element holds the previous level on the output, which is stored internally. Thus, an event is only generated at the output after an event took place on each input. The *Muller C-Element* can be generalized for three or more inputs with the only requirement that all inputs must have the same logical level before the output changes to that level.

This state-holding element is not a conventional gate but rather like a latch. It is an indispensable component when designing asynchronous circuits.

Micropipeline

A *Micropipeline* [19, 18] is a circuit built of *Muller C-Element*s and inverters as depicted in Figure 2.4. Request and acknowledge signals are passed between adjacent stages. Observing one stage of the pipeline the request signal of the predecessor is only propagated if the acknowledge signal of the successor is of the opposite level. In other words a stage can only pass the predecessor's request if the previous request is already acknowledged by the successor. Data might also be passed between stages of the *Micropipeline*, but for clearness the data path is omitted in this figure and only the control signals are shown.

The *Micropipeline* is an elastic event-driven pipeline. Without internal processing of data it acts as a first-in first-out memory. Due to its modularity an arbitrary number of stages can be linked together.

There are some remarkable facts about the *Micropipeline* due to its simplicity and symmetry. The handshake style can be four-phase or two-phase. The difference is only in how the signals are

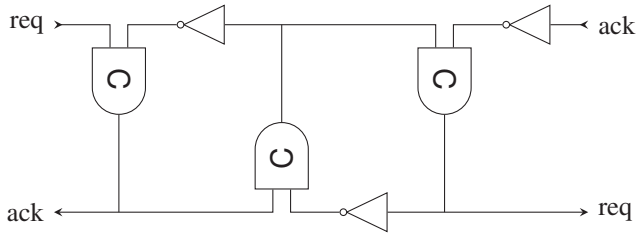


Figure 2.4: Micropipeline

interpreted. The pipeline works from either side, only request and acknowledge signals have to be swapped. The circuit works correctly regardless of the gate delays. It is delay-insensitive.

Because of its powerfulness a *Micropipeline* can be found in all asynchronous circuits as a control circuit.

2.1.4 Timing Models

Asynchronous circuits can be categorized by the employed timing assumptions [11].

Bounded Delay

In this timing model the delays of all gates and wires must be known or at least bounded. Obviously assumptions on the worst-case operating conditions have to be made. However, constraints on the path delays decrease robustness of a circuit against variations within the manufacturing process. Hence, circuits with bounded delay deliver worst-case performance.

With the use of *Micropipelines* for local handshaking the skew of the individual stages does not add up. Thus, timing assumptions only need to be made locally.

Delay-Insensitive

In this timing model the delays of all gates and wires are finite but unbounded. As no assumptions on the computation time are made it can handle arbitrary delay. Obviously this realization of a circuit is robust to process variations and interconnect delays. Hence, delay-insensitive circuits deliver average-case performance.

To achieve validity in delay-insensitive circuits, the state of the output of a gate has to be preserved until all inputs are valid. For this purpose only *Muller C-Gate*s and single input gates like inverters and buffers are applicable. Thus, a *Micropipeline* is delay-insensitive but circuits with Boolean logic gates are not.

Quasi Delay-Insensitive

In this timing model the delays of all gates are finite but unbounded. The delays of branches of certain critical wire forks have identical wire delay. These forks are called isochronic forks [18]. The same coding techniques as for delay-insensitive circuits are applicable.

2.1.5 Data path Implementation Styles

The implementation and behavior of a handshake protocol depends on the encoding of the control and data paths. Furthermore, the handshake style, i.e. four-phase or two-phase, has an effect on the protocol. First techniques to achieve validity are described. Then the most common handshake protocols are presented [18].

Validity

A bit at the output is valid if the inputs are valid and the output is stable. Validity is either indicated by separate control signals or by codewords. In case of a data bus all bits at the input have to be valid before the data is acknowledged.

Employing the bounded delay model delay elements have to be placed on the control path to assure that the outputs of combinational logic and the inputs of the circuit have settled before they are captured. Validity of datawords is guaranteed by adding an adequate amount of delay elements to match worst-case computation times.

Employing the quasi delay-insensitive model correct circuit behavior is achieved by an appropriate coding technique or data representation. To have an unambiguous valid state in quasi delay-insensitive circuits, no intermediate valid codewords are allowed. To identify a new codeword, successive codewords must be separated by at least one signal transition. Validity of datawords is determined by a separate completion detection circuit.

Single-Rail

The term *single-rail* refers to all encodings where the individual data bits are encoded with Boolean values, each on a single wire. All data wires are bundled together accompanied by a separate request and acknowledge signal.

Figure 1.2 shows a circuit with a single-rail protocol. For circuits implementing the single-rail protocol assumptions about the computation time of the combinational logic have to be made. To maintain correct behavior, matching delays in the control path are needed to compensate the delay of the data path in the combinational logic. Refer to Figure 2.5 for a *Micropipeline* with delay elements on the control paths. The control path of the pipeline is quasi delay-insensitive, whereas the data path follows the bounded delay model.

Waveforms that show the behavior of the data and control signals for both four-phase and two-phase handshaking are depicted in Figure 2.2.

Four-Phase Single-Rail An implementation of a single-rail pipeline in combination with a four-phase handshake is illustrated in Figure 2.5. The output signal of a *Muller C-Element* controls the attached latch. When a new request arrives from the predecessor and the the previous request is acknowledged by the successor the latch starts passing the data. Once this stage acknowledges the request to the predecessor the latch is switched to capture mode.

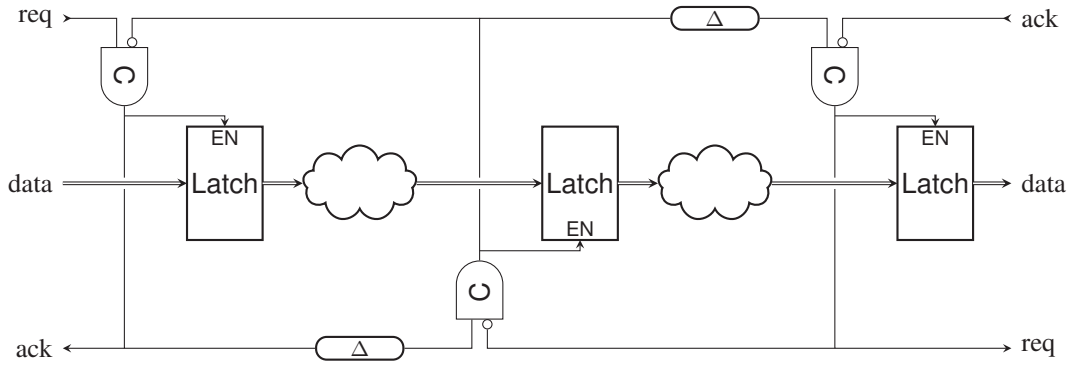


Figure 2.5: Four-Phase Single-Rail Pipeline

Two-Phase Single-Rail As the two-phase handshake implementation is event-driven the latch control must be event-driven as well. For this reason special latches are required. Figure 2.6 depicts a single-rail pipeline in combination with a two-phase handshake. This pipeline can be distinguished from the one with a four-phase handshake by special capture-pass latches along the data path. If an event triggers the `C` input of the latch it starts to output the captured data. On the other hand if an event triggers the `P` input of the latch it starts passing the data.

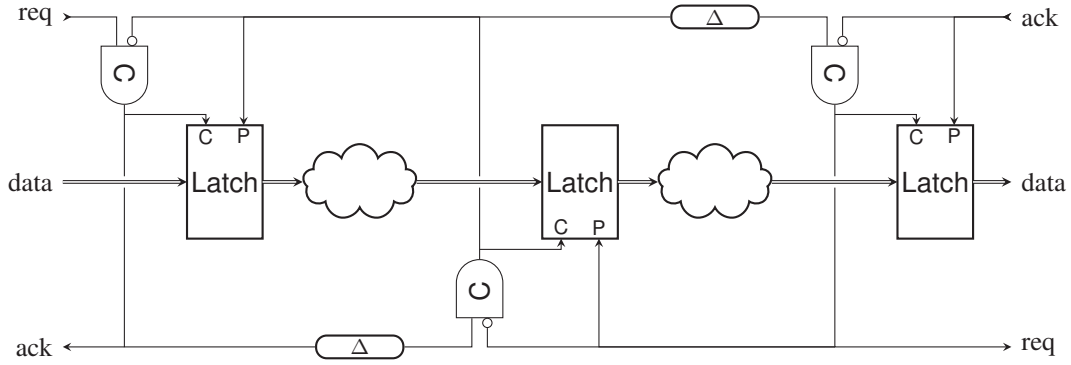


Figure 2.6: Two-Phase Single-Rail Pipeline

Multi-Rail

The term *multi-rail* refers to all protocols with combined encoding of data and the request signal accompanied by a common acknowledge signal on push channels. To accomplish combined encoding of data and request, more than two signal states per bit are required. Hence, more than one wire per bit is needed. Likewise, the encoding of data and the acknowledge signal is combined accompanied by a common request signal on pull channels. For convenience, only push channels are explained in greater detail. Pull channels can be derived.

Figure 2.7 shows a 1-bit wide four-phase dual-rail pipeline without data processing. It can

be seen as two *Micropipelines* with a common acknowledge signal. The request wires of the pipelines carry besides the request the data, encoded in a codeword. In this figure one wire is used to represent a logical `one`, the other wire is used to represent a logical `zero`. The request is acknowledged when either of the two wires is set to high.

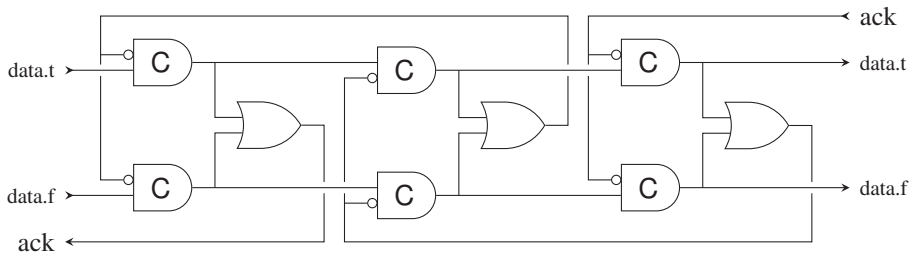
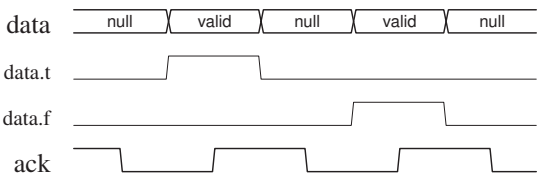


Figure 2.7: Four-Phase Dual-Rail Pipeline

To ensure correct indication in combinational blocks for multi-rail protocols, state-holding elements are required. With the combined encoding of data and the request signal a reliable communication is achieved regardless of gate delays. Due to isochronic forks, which can usually be found in these circuits, the control path as well as the data path of the pipeline are quasi delay-insensitive.

Null Convention Logic (Four-Phase Dual-Rail) *Null Convention Logic* is a four-phase protocol using two wires per data bit [6]. One wire is used to represent a logic `one`, the other wire is used to represent a logic `zero`. Valid datawords are always spaced with empty codewords. Figure 2.8 illustrates the channel encoding for the *Null Convention Logic* protocol. In this example the code `10` is transmitted.

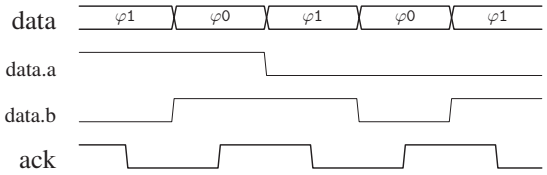


	data.t	**data.f**
null	0	0
valid ‘0’	0	1
valid ‘1’	1	0
illegal	1	1

Figure 2.8: Null Convention Logic Protocol (Push Channel)

There is exactly one transition change for each bit between two valid codewords. For a switch from the empty codeword to a dataword the corresponding wire changes from `zero` to `one`. For a switch from a dataword to the empty codeword the corresponding wire changes back from `one` to `zero`. Once the completion detector indicates a valid dataword the acknowledge signal is asserted. Since this is a four-phase protocol the empty codeword resets all data wires before another codeword can be transmitted. Hence, the acknowledge signal is only deasserted when all bits are empty.

Level-Encoded Two-Phase Dual-Rail (LEDR) *LEDR* is a two-phase dual-rail protocol with level signaling [2]. In the *Null Convention Logic* protocol every second codeword does not carry information. An alternative solution is to use two codewords for each logic value, each of them belongs to a different phase. To distinguish consecutive datawords, the corresponding codewords have alternating phases. Figure 2.9 illustrates the channel encoding for the *LEDR* protocol. In this example the code `11000` is transmitted. As can be seen from the example the throughput of a circuit with *LEDR* encoding is twice as high as for circuits with *Null Convention Logic* encoding.



	data.a	**data.b**
φ0 '0'	0	0
φ1 '0'	0	1
φ1 '1'	1	0
φ0 '1'	1	1

Figure 2.9: Level-Encoded Two-Phase Dual-Rail Protocol (Push Channel)

The codewords are chosen in a way that all codewords of the other phase are reached by exactly one transition. An encoding for the *LEDR* protocol can be seen in Figure 2.9. Completion detection works similarly as for *Null Convention Logic*.

Transition-Encoded Two-Phase Dual-Rail A two-phase dual-rail protocol can also be implemented with transition signaling. Each transition represents a codeword. Figure 2.10 illustrates the channel encoding for a *Transition-Encoded Two-Phase Dual-Rail* protocol. In this example the code `1001` is transmitted.

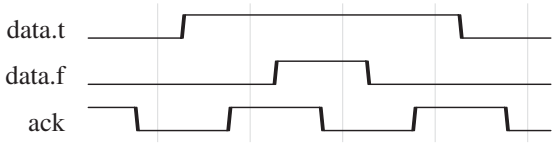


Figure 2.10: Transition-Encoded Two-Phase Dual-Rail Protocol (Push Channel)

The current state of a codeword depends on the previous transitions. Thus, it is not visible but must rather be stored in state-holding elements. This is problematic for the identification of consistent datawords. Likewise it is difficult to implement a transition-based completion detection.

2.2 Balsa

For designing asynchronous circuits that are only synchronized by handshake communications a special development environment is of advantage. Conventional tools for design and verification are tailored to the clocked approach. As asynchronous development is not widely spread not many design tools optimized for asynchronous circuits are available. The development environment of choice for implementing ASCARTS is the open-source project Balsa, which was developed at the *APT group of the School of Computer Science, The University of Manchester*. This section gives an overview over Balsa. For a more detailed description see [4].

Balsa is a language for describing asynchronous hardware systems as well as a framework for developing such systems. A key feature of the language is transparent compilation. There is a one to one mapping between language constructs and intermediate handshake circuits. The advantage of this approach is that incremental changes in the code result in predictable changes in the generated circuits.

The Balsa framework comprises a collection of tools. The most important ones are listed in Table 2.1.

`balsa-c`	compiles Balsa files into intermediate Breeze description
`breeze-sim`	behavioral simulator for Breeze files on handshake level
`balsa-netlist`	produces a netlist from Breeze description

Table 2.1: Balsa Tools

Balsa is a strongly typed language, hence, all casts have to be explicit. The only exception to this are implicit casts of numeric types to wider numeric types. Thus, special care has to be taken with arithmetic operations.

The language supports modular compilation to realize a library mechanism. With the `import` statement precompiled Balsa components, i.e. Breeze design files, can be included into the current Balsa design file allowing component reuse.

2.2.1 Control Flow

In this section Balsa or rather asynchronous specific language constructs are explained in more detail and are illustrated by examples. For conventional constructs, which might be familiar, refer to [4].

Sync Control only handshake channels communicate via `sync` commands. Both ends have to signal a `sync` request to successfully complete the handshake. In Listing 2.1 first the handshake on the sync channel `i` is completed. Afterwards a handshake on the sync channel `o` is initialized.

```
procedure sync_buffer (
  sync i ;
  sync o
)
is
begin
  loop
    sync i
    ;
    sync o
  end
end
```

Listing 2.1: Sync Buffer

Channel Assignment The operators $\rightarrow$ and $\leftarrow$ are channel assignment operators implying a handshake data transfer over that channel. $\rightarrow$ assigns the data read from the input channel to a variable or an output channel and completes the handshake. $\leftarrow$ assigns data to an output channel from a variable or expression and initializes the handshake. Each channel has exactly one input and one output port. Hence, passing a signal to several procedures is not possible as used to from conventional languages. For each procedure a separate channel is needed with explicit signal assignment. In Listing 2.2 first the data from the input channel `i` is assigned to `x` and simultaneously the handshake on that channel is completed. Afterwards the data from `x` is written to the output channel `o` and simultaneously a handshake on that channel is initialized.

The operator $:=$ assigns the result of an expression to a variable. In Listing 2.3 data of `x` is transferred to `y`. The transfer is implicitly performed by a handshaking communication channel. Hence, `y` $:=$ `x` can also be written as `ch` $\leftarrow$ `x` $\|$ `ch` $\rightarrow$ `y`.

```
import [balsa.types.basic]

procedure single_place_buffer (
  input i : byte;
  output o : byte
)
is
  variable x : byte
begin
  loop
    i -> x
    ;
    o <- x


  end
end
```

Listing 2.2: Single-Place Buffer

```
import [balsa.types.basic]

procedure two_place_buffer (
  input i : byte;
  output o : byte
)
is
  variable x, y : byte
begin
  loop
    i -> x
    ||
    o <- y
    ;
    y := x
  end
end
```

Listing 2.3: Two-Place Buffer

Sequencing The ; operator connects two statements or blocks sequentially. A sequence operator is used in Listing 2.2. First the input channel `i` is read. Once the handshake on that channel is completed it is written to the output channel `o`.

Parallel Composition The || operator connects two statements or blocks concurrently and independently. Both commands must be completed before processing the subsequent command or commands. This operator has precedence to the sequence operator. A parallel composition operator is used in Listing 2.3. Reading the input channel `i` and writing to the output channel `o` occurs concurrently. Once the handshakes on both channels are completed the value of `x` is copied to `y`.

Choice For situations where a handshake may occur only on one channel out of several a multiplexer is needed. In the Balsa language self-selecting multiplexer is available for this purpose. This operator waits for a handshake request on any of the channels. Once a handshake initialization is detected the block associated with the selected channel is processed before the handshake on that channel is acknowledged. This is a type of handshake enclosure which is described in more detail in Section 2.2.3.

In Listing 2.4 the `select` operator is used to choose between input channel `i` and input channel `j`. The system design must ensure that handshakes on the channels in a `select` statement never occur simultaneously.

For truly independent channels another choice operator `arbitrate` is available. As with non-deterministic choices the possibility of entering metastability arises, additional hardware is needed to avoid or resolve that state. Due to this overhead arbiters are expensive in both area and speed. Hence, the `arbitrate` statement should only be used if really necessary. In Listing 2.5 the `arbitrate` operator is used to choose between input channel `i` and input channel `j`. If a handshake request arrives on both channels simultaneously the order in which the corresponding blocks are processed is not deterministic.

Both choice operators can be used for control only channels as well as channels transferring data or a mixture of both. In Listing 2.4 a choice operator is applied to data bearing channels, whereas in Listing 2.5 a choice operator is applied to dataless channels.

2.2.2 Design Structure

A design consists of several files containing procedures, type and constant declarations. If several procedures are defined in the top-level design file the bottom one is taken as main procedure.

Additionally, to the previous mentioned declarations a procedure can also have local variable and channel declarations. The scope of each declaration starts at the point of declaration and ends at the end of the enclosing block. If a declaration within an enclosed block uses the same name as a previously made declaration in the enclosing context, the outer declaration is hidden until the end of the enclosed block.

```
import [balsa.types.basic]

procedure choice (
  input i : byte;
  input j : byte;
  output o : byte
)
is
begin
  loop
    select i then
        o <- i
    | j then
        o <- j
    end
  end
end
```

Listing 2.4: Non-Arbitrated Choice

```
procedure arbitrated_choice (
  sync i;
  sync j;
  sync o
)
is
begin
  loop
    arbitrate i then
        sync o
    | j then
        sync o
    end
  end
end
```

Listing 2.5: Arbitrated Choice

Shared Procedures Each instantiation of a procedure induces separate hardware. If the calls to a procedure are mutually exclusive the individual instantiations can share their hardware. For these cases Balsa provides shared procedures. Calls to such procedures access common hardware, hence, multiplexers are needed to coordinate the access. In Listing 2.6 a counter with the possibility to change the direction is implemented. In the design two adders are instantiated, one used for incrementing the register and one used for decrementing the register. As the adders are not used concurrently in Listing 2.7 the design is rewritten to share a single adder.

Parameterized Procedures Procedures can be parameterized to be used as generic library components with varying parameter types. No hardware is generated for parameterized procedures until it is instantiated with defined parameters. In Listing 2.8 a buffer definition with generic input and output types is given. Aliasing of procedures with defined parameters, see last line of Listing 2.8, is optional. Instead of calling the aliased procedure `byte_buffer(i, o)` a call to the procedure could also be `generic_buffer(byte, i, o)`.

2.2.3 Handshake Enclosure

With channel assignments in Balsa once the data is consumed the handshake is completed and the data may be removed from the channel. Hence, if the data is required more than once it has to be stored in a variable. Listing 2.9 shows a channel multiplier. The data of input channel `i` has to be stored in a temporary variable as it is needed twice. To avoid temporary variables, Balsa provides two constructs where the handshake is held open until a sequence of commands is processed. One possibility is the `select` statement as used in Listing 2.4. If there is no choice necessary an alternative command is provided as used in Listing 2.10. In this listing the $\rightarrow$ operator is used to hold the handshake on input channel `i` open. Using these constructs the channel acts like a variable in the enclosed block.

```
import [balsa.types.basic]

procedure counter (
  input direction : bit;
  output count : byte
)
is
  variable tmp_direction : bit
  variable reg : byte




begin
  loop
    direction -> tmp_direction ;
    if tmp_direction then

      reg := (reg + 1 as byte)
    else

      reg := (reg - 1 as byte)
    end ;
    count <- reg
  end
end
```

Listing 2.6: Counter

```
import [balsa.types.basic]

procedure counter (
  input direction : bit;
  output count : byte
)
is
  variable tmp_direction : bit
  variable reg : byte
  variable inc : 2 signed bits

  shared adder is
  begin
    reg := (reg + inc as byte)
  end
begin
  loop
    direction -> tmp_direction ;
    if tmp_direction then
      inc := 1 ;
      adder()
    else
      inc := -1 ;
      adder()
    end ;
    count <- reg
  end
end
```

Listing 2.7: Counter with Shared Adder

```
import [balsa.types.basic]

procedure generic_buffer (
  parameter X : type;
  input i : X;
  output o : X
)
is
  variable x : X
begin
  loop
    i -> x ;
    o <- x
  end
end

procedure byte_buffer is generic_buffer(byte)
```

Listing 2.8: Generic Buffer

Figure 2.11 shows the circuit behavior without handshake enclosure and the circuit behavior with enclosed handshake. The waveform in the figure shows a four-phase protocol. Green logical `1` denotes an active request and red logical `x` denotes an active acknowledge. Green logical `0` represents a removed request and with blue logical `z` the acknowledge is removed as well. Comparing the two traces reveals the difference in behavior for these two approaches. The advantage of enclosed handshakes is that the channel can be read several times, even zero times, without deadlocking and at the same time avoiding internal latches for storing the data. On the other side the performance decreases because the handshake completion is delayed. Hence, the data provider is locked and cannot continue processing independently.

```
import [balsa.types.basic]

procedure channel_multiplier (
  input i : byte;
  output o : byte;
  output p : byte
)
is
  variable x : byte
begin
  loop
    i -> x ;
    o <- x ||
    p <- x
  end
end
```

Listing 2.9: Channel Multiplier

```
import [balsa.types.basic]

procedure channel_multiplier (
  input i : byte;
  output o : byte;
  output p : byte
)
is
begin
  loop
    i -> then
      o <- i ||
      p <- i
    end
  end
end
```

Listing 2.10: Enclosed Channel Multiplier

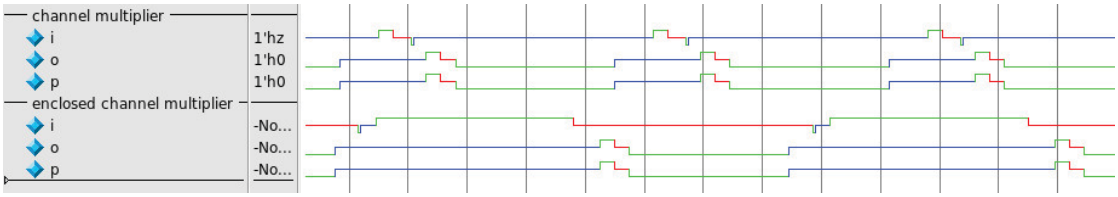


Figure 2.11: Waveform of Channel Multipliers

2.2.4 Simulation

For simulation purposes additional built-in types and functions are available originally defined in C. Built-in types represent pointers to `BalsaObject` structures, which can be used as parameters, ports and variables. However, arithmetic functions cannot be applied to them.

Built-in functions for generating and manipulating `String` objects as well as for string conversion are available. For file access the built-in object type `File` and associating functions which operate on that type are provided. To simulate memory, a library with the built-in type `BalsaMemory` and built-in functions for reading from and writing to that object exist. In addition, a Balsa procedure

with a read and write port implementing the memory access logic is provided by the library. An in-depth description of the built-in library types and functions can be found in [4].

CHAPTER 3

State of the Art

This chapter gives an overview of the most important asynchronous processors.

At *The University of Manchester* three versions of an ARM-compatible microprocessor were developed, AMULET1 [22], AMULET2 [8], AMULET3 [9]. Asynchronous implementations of the MIPS R3000 processors were developed at *The University of Birmingham* [23] and at *California Institute of Technology* [13], respectively. At the *California Institute of Technology* Lutonium [14], an asynchronous 8051 microcontroller, was also developed. An asynchronous 80C51 microcontroller [20] was developed at *Eindhoven University of Technology* and *Philips Research Labs*. This processor is, however, not presented in this chapter.

3.1 AMULET1

AMULET1 [22] is an asynchronous processor based on the *ARMv3* microprocessor architecture. It was developed using an approach derived from Sutherland's *Micropipelines* [19]. Unusual properties comprise non-deterministic, but bounded, prefetch depth beyond a branch instruction, a data dependent throughput and a novel register locking mechanism.

The architecture consists of four major functional units, the *Address Interface*, *Register Bank*, *Execute* unit and *Data Interface*. These operate concurrently and asynchronously, synchronizing with each other only to exchange data. Instruction stream *coloring* is used for discarding an uncertain number of prefetched instructions from the interrupted instruction stream. Whenever the instruction stream terminates the color of the instruction is changed. By checking the color at the decode stage and at the ALU result stage wrongly prefetched instructions and their results can be identified and discarded. A failing memory access on instruction fetch only causes a trap to be entered for instructions actually required, i.e. if the color matches. To handle read and write operations correctly in an asynchronous environment, the register bank uses *FIFO Lock Registers* [17] for pending write backs. Separate ones are used to lock registers subjected to memory writes and ALU writes, respectively.

As implementation style the two-phase single-rail protocol was chosen. However, instead of capture-pass latches conventional latches are used. Thus a conversion from two-phase protocol to four-phase protocol as well as the back conversion is needed.

The chip was developed with the aid of conventional synchronous tools using a conventional design flow. For high-level simulation a proprietary simulator from ARM Limited was used. Transistor level simulation and all layout was performed with commercial VLSI CAD tools. In addition software was developed, which checks the validity of data bundles of the single-rail protocol between request and acknowledge signals. The components for the data path are primarily fully customized, whereas for control logic the layout consists mostly of compiled standard cell circuits.

3.2 AMULET2

AMULET2 [8] is an enhanced version of AMULET1. Analysis of AMULET1 revealed that the pipeline depths are too large. Some stages hardly increase performance but cost area and power dissipation. Others even decrease performance. With regard to these aspects the pipelines were reorganized. To avoid pipeline stalls when a register dependency is detected, *last result* registers are used to emulate register forwarding. The latter one is only reasonable in synchronous design as it relies on pipeline synchronization. Separate *last result* registers are used for the ALU, which is used when the result calculated by the ALU is required as an operand, and the memory, which is used when the operand being loaded from the memory is required. Only the results of instructions with the *always* guard, i.e. instructions that are unconditionally executed, are forwarded to be sure that the value will become available. AMULET2 employs a *Branch Target Cache* to predict the target of the next branch instruction. It is a static branch predictor that caches the addresses and targets of the 20 recently taken branch instructions. AMULET2 detects instructions that branch to themselves as *halt* instructions and triggers a mechanism to stall the processor.

As implementation style the four-phase single-rail protocol was adopted. In AMULET1 many two- to four-phase conversions were needed for pipeline registers and dynamic logic. Besides, the implementation of two-phase control elements was inefficient.

The chip was developed with off-the-shelf commercial EDA tools and design libraries, which are optimized for synchronous designs. For high-level simulation a proprietary simulator from ARM Limited was used. Several potential deadlocks were found early in the design cycle by varying the delays of gates within the asynchronous control paths. Many of the asynchronous control circuits were synthesized using a specific asynchronous logic tool, Forcage. The data path and most of the cache is fully customized. However, large amounts of the control logic were compiled from standard cells supplemented with physically compatible cells.

3.3 AMULET3

AMULET3 [9, 10] is the third generation of asynchronous ARM-compatible microprocessors. It is based on the ARMv4T microprocessor architecture including full compliance with the 16-bit thumb instruction set.

The instruction port fetches 32-bit words unless it is known that only one half-word is needed when operating in thumb mode. As the pipeline is elastic and units are only invoked when needed by the instructions, the pipeline's depth varies. Some instructions may be discarded already before the execution stage. Others may be expanded into multiple cycles inside the pipeline. Instruction stream *coloring* is used for discarding prefetched instructions in a similar way as in earlier AMULET processors. However, in AMULET3 the instructions are already discarded in the decode stage. This adaption leads to power reduction as registers are not read unnecessarily.

The instruction prefetch unit comprises some novel features. Instructions that branch to themselves are reinterpreted as *halt* instructions. On execution this causes the prefetch unit to stop, which, in turn, stops the asynchronous subsystem. AMULET3 employs a *Branch Target Buffer* with 16 entries. It provides all the information contained in the instruction. Therefore, instruction fetch is suppressed. This leads to faster instruction delivery and less power consumption.

As the previous AMULET processors the implementation is based on an asynchronous micropipeline structure using four-phase control signals. Balsa was used to implement parts of the DMA controller, which is included in the AMULET3i microprocessor subsystem.

3.4 SAMIPS

SAMIPS [23] is an asynchronous processor core based on the MIPS R3000 architecture. Its pipeline consists of five stages, the *Instruction Fetch*, *Instruction Decode*, *Execution*, *Memory Access*, *Register Write Back*. The control signals are generated in the instruction decode stage and then bundled together with the data and passed through the pipeline. To avoid data hazards, an asynchronous register forwarding mechanism was developed based on history information. An extended version of the *coloring* mechanism of the one used in AMULET1 is used for discarding prefetched instructions. The color of the instructions as well as of the processor change every time a control hazard occurs. Instructions are only executed if the color matches that of the processor.

SAMIPS was modeled in Balsa. For behavioral simulation LARD, a hardware description language for describing asynchronous systems [5], was used. The memory model for the simulation was also written in LARD. LARD is able to detect deadlocks by terminating the simulator when a deadlock occurs with subsequent channel activity analysis.

3.5 Asynchronous MIPS R3000

An asynchronous MIPS R3000 processor [13] was developed to optimize the average cycle time and the average energy per instruction. By minimizing Et^2 both high speed at high voltage and

low power at low voltage can be achieved. MIPS R3000 was chosen because it is the archetype of a commercial RISC microprocessor. However not the complete MIPS was implemented.

The processor consists of two distinct parts, the fetch loop, consisting of the *PC* unit, the *Fetch* and the *Decode*, and the execution pipeline, consisting of the execution units, register unit and *Write Back*. All execution units operate in parallel and can execute different instructions concurrently. The whole design is very finely pipelined.

To determine the sequence of canceled instructions for precise exceptions, the decoder maintains a queue of unit numbers that represent the order of the dispatched instructions. The central mechanism for exception handling is the write back unit. It sends a discard message to the register unit if the instruction was canceled. The PC unit generates a bit, which signals the write back unit that the instruction stream is valid again. A pseudo-instruction is inserted into the instruction stream before the first instruction of the exception handler, which requests to store the information about the exception. The register unit has two result buses, which are used alternately to reduce pipeline stalls. The processor has two caches. The cache core is pipelined internally allowing two look-ups to proceed concurrently. When a cache miss occurs the handshake between the cache and the main pipeline is stretched making the latency variations transparent to the rest of the system.

This processor is quasi delay-insensitive using four-phase control signals. The data is encoded using `1-of-N` and dual-rail codes.

3.6 Lutonium

Lutonium [14] is an asynchronous 8051-architecture microcontroller. It is designed for low Et^2, the best trade-off between energy and cycle time. The choice of the 8051 ISA is justified by the fact that it is the most popular microcontroller. On the other hand, though, it is a complex and irregular instruction set that tends to increase the energy consumption.

For energy efficiency, switching activity is minimized. No register or execution unit receives control unless it will process data. Nothing is computed or routed unless necessary. Thus the pipeline is non-speculative. It is filled by the instruction fetch unit with instructions only if these are definitely going to be executed. Therefore, cycles executing interrupt requests or branches are stretched. For energy and time savings implicit operands and special registers have their own channels. As instruction Fetch is the limiting part of the design two consecutive bytes of code are fetched simultaneously. Even though that introduces some speculation if the last byte of a basic block happens to fall on an even address, the speed advantage compensates the energy costs. To not make the optimizations pointless, direction registers were added to the peripheral interfaces so that passive pull-ups are not required.

Lutonium is quasi delay-insensitive using four-phase control signals. The processor description is composed of concurrent *Communicating Hardware Processes* [15].

CHAPTER 4

SCARTS - Basis For The Asynchronous Processor

SCARTS is an abbreviation for **S**calable **C**omputer **A**rchitecture for **R**eal-**T**ime **S**ystems. It is a synchronous processor developed and actively maintained at *Vienna University of Technology*, *Department of Computer Engineering*. This processor was specifically designed as a soft-core processor for embedded systems with real-time requirements. To fulfill these requirements, the processor has to be adaptable and real-time capable [3]. Adaptability is accomplished by the usage of extension modules to extend the functionality of the processor. Real-time capability is achieved by constant execution time of all instructions, even if a conditional instruction is not executed, and deterministic behavior of interrupt execution.

The features that are adopted for the implementation of ASCARTS are described in more detail. For other information about SCARTS refer to [7, 21].

4.1 Processor Architecture

SCARTS is a RISC[1] processor with a 16-bit address bus and a customizable data path. The data path can be switched between 16-bit for small processor cores used for FPGAs with little resources and 32-bit for improved performance and higher storage capacity due to 32-bit addresses. Figure 4.1 shows the block diagram of SCARTS. The processor has four pipeline stages. These are *Instruction Fetch*, *Instruction Decode*, *Execute* and *Write Back*. Data hazards are resolved by the usage of a register forwarding unit. Control hazards are also resolved in hardware. These mechanisms keep programs simple and worst-case execution times rather predictable.

The register file has 16 registers, whereof two are special purpose registers. In addition four frame pointers are available.

[1] RISC, **R**educed **I**nstruction **S**et **C**omputing, is a load/store architecture.

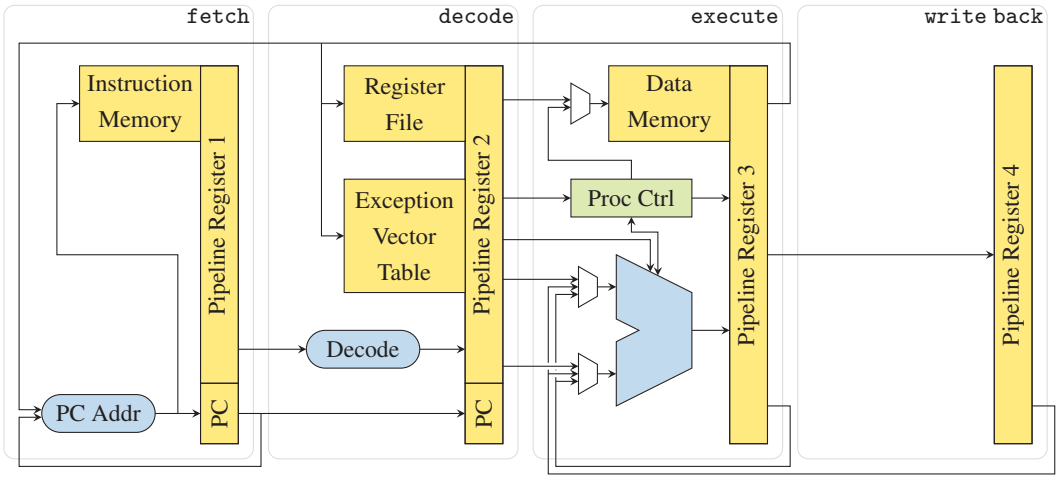


Figure 4.1: Block Diagram of SCARTS

SCARTS supports 32 exceptions, 16 of which are hardware triggered interrupts and 16 are software triggered traps. The exception vector table is stored in a separate memory. The instruction set provides special instructions to build and to manipulate the exception vector table.

To get high flexibility, the processor can be extended by extension modules, which are mapped to data memory address space. The processor status and the interrupt handling logic are controlled by the processor control module. Therefore, it is obligatory to include that module for correct processor operation. For a detailed description of extension modules see Section 4.2.

A sleep mode is provided for power saving. Once the processor is in sleep mode it only wakes up if an interrupt is triggered.

4.1.1 Memory Architecture

The memory architecture of SCARTS is a *Harvard Architecture*, i.e. instruction memory and data memory are two separate physical memories with independent signal buses. The instruction memory uses word access since instructions are read as words. As far as the data memory is concerned a byte access memory organization was implemented. The byte ordering of this memory is little endian. Memory accesses have to be aligned.

The physical interface of extension modules is also mapped into data memory. Each module allocates 32 bytes of the memory starting at the top of the memory. The size of the internal dynamic memory is configurable as well as the address space reserved for extension modules. The default memory mapping of SCARTS' data memory with a 32-bit configuration can be seen in Figure 4.2. The extension module interface is mapped to the top of the address space. The processor control module is by default mapped to the slot with the base address of -32. All frame pointers point to the top of the internal dynamic memory on program start-up.

Addressing modes of an architecture define how the physical instruction operand addresses in the memory are calculated by a processor [21]. The memory addressing modes supported by

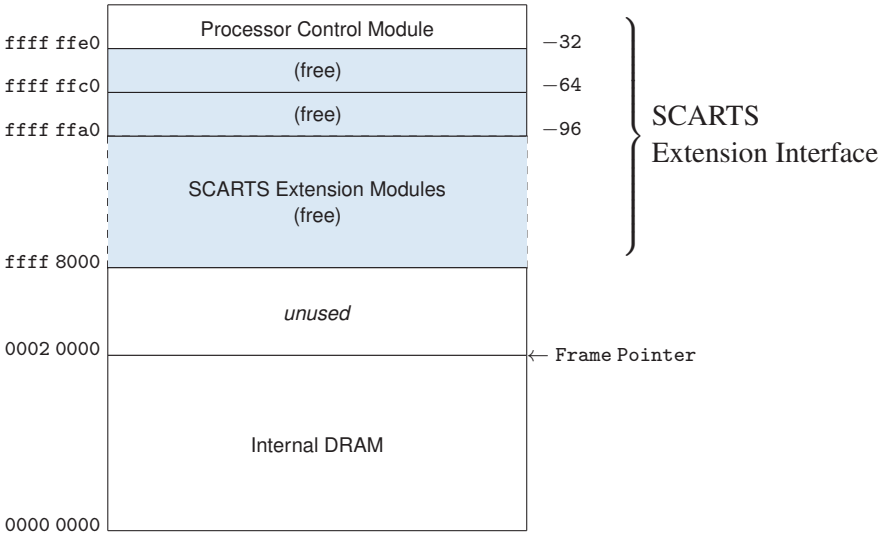


Figure 4.2: Data Memory Address Space for 32-bit Configuration

SCARTS are presented in Table 4.1.

Mode	Description	Instruction Memory	Data Memory
Absolute	address constants	—	—
Register Indirect	address values in registers	√	√
FP-Relative	displacement value added to frame pointer	—	√
PC-Relative	displacement value added to program counter	√	—

Table 4.1: Memory Addressing Modes

4.1.2 Register File

The register file consists of 16 registers. All registers, `r0` to `r15`, are accessible from all instructions. Two registers, `r14` and `r15`, additionally have special functionality. The return address after a subroutine call is saved to the `rts` register, mapped to `r14`. The `rte` register, mapped to `r15`, holds the return address in case of an interrupt. For nested subroutine calls and interrupts the appropriate register has to be saved manually.

4.1.3 Frame Pointers

The SCARTS processor provides four frame pointers to control four different frames. Frames are similar to stacks. The difference, however, is that data can be stored to and read from frames in

arbitrary order by specifying an offset to the frame pointer. Due to 6-bit offset values 64 words can be addressed from one frame pointer as only word access is allowed.

With post-increment and post-decrement frame pointer access instructions auto-increment and auto-decrement functionality is supported. As the address space of frames expands from the top to the bottom by recommendation `pop` can be easily realized with a post-increment instruction and an offset of 0. On the other hand, `push` needs pre-decrement functionality, which has to be simulated by a post-decrement instruction with an offset of -1.

4.2 Extension Modules

SCARTS can be adapted for different requirements by integrating extension modules. To keep the usage of different modules simple and consistent, a common interface was designed. Each extension module is accessed through 32 bytes irrespective of the data path size. The interface is mapped with the module's base address into data memory. Thus, no additional instructions are required as conventional memory instructions can be used.

The first two bytes of the interface define the status register, which is read-only. The subsequent two bytes form the configuration register for the module. Each of these two registers is split into a generic part and a custom part. The generic low byte of the status register and the generic low byte of the configuration register define the common interface to the extension modules. The high bytes of both registers as well as the other 28 bytes can be used module specific. In Figure 4.3 the register interface specification of extension modules is depicted.

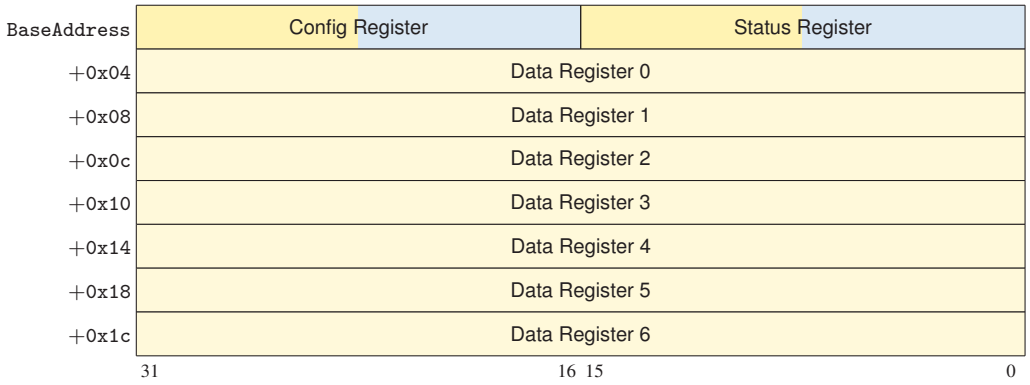


Figure 4.3: Register Interface for Extension Modules

Figure 4.4 shows the generic parts of the status register and the configuration register. The register description of these registers is listed in Table 4.2 and Table 4.3. The functionality for these bytes has to be implemented together with the customized part for each module.

4.2.1 Processor Control Module

The processor control module is essential for the processor and, hence, cannot be omitted. It contains the processor status register, the frame pointers, is responsible for controlling exceptions

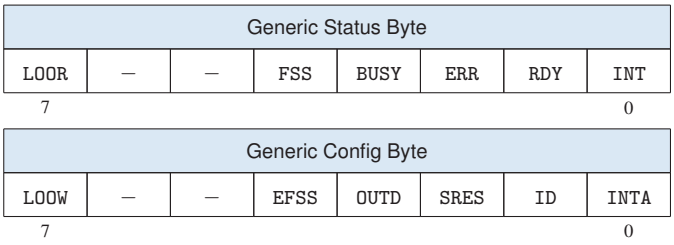


Figure 4.4: Generic Status and Configuration Byte

INT	The *Interrupt* bit is set if an interrupt was triggered by the module.
RDY	The *Ready* bit indicates that the module is ready to operate.
ERR	The *Error* bit is set if an error occurred.
BUSY	The *Busy* bit indicates that the module is not ready for new tasks.
FSS	The *Fail-Safe-State* bit can be set by the module or by the *Enter-Fail-Safe-State* bit of the configuration register and indicates that the module is in the fail-safe-state.
LOOR	The *Loop-Read* bit shows the *Loop-Write* bit of the configuration register with one cycle delay. It is used to determine the presence of a module.

Table 4.2: Generic Status Byte

INTA	The *Interrupt Acknowledge* bit acknowledges the interrupt and clears the *Interrupt* bit in the status register.
ID	If the *Identify* bit is set, manufacturer and version number can be read from data register 0.
SRES	The *Soft-Reset* bit triggers a software reset of the module.
OUTD	The *Output-Disable* bit disables the module.
EFSS	If the *Enter-Fail-Safe-State* bit is set, the module enters the fail safe state.
LOOW	The *Loop-Write* bit is shown as *Loop-Read* bit of the status register with one cycle delay. It is used to determine the presence of a module.

Table 4.3: Generic Configuration Byte

and activates the sleep mode. In Figure 4.5 the register interface of the processor control module is depicted. Figure 4.6 shows the register description of the customized parts of the status register and the configuration register.

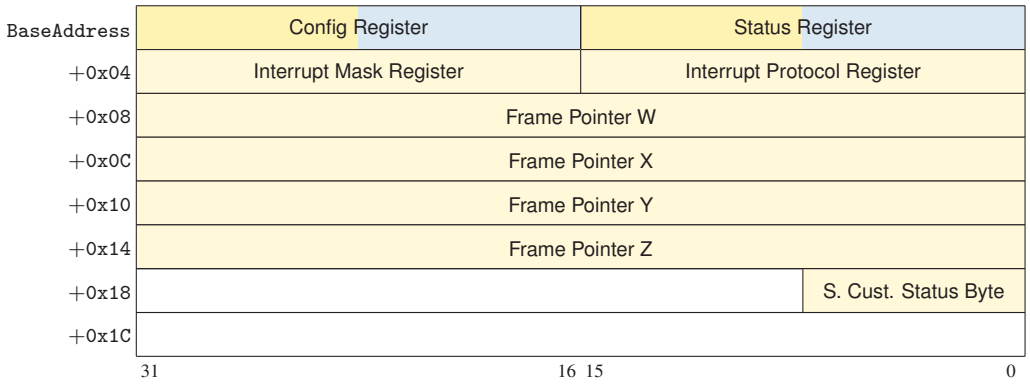


Figure 4.5: Register Interface for Processor Control Module

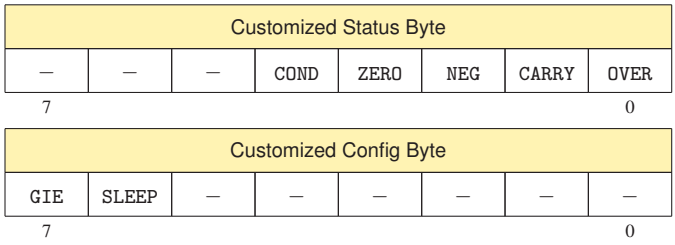


Figure 4.6: Customized Status and Configuration Byte of Processor Control Module

The customized status byte comprises the status flags of ALU[2] operations. These flags are *Overflow*, *Carry*, *Negative*, *Zero* and *Condition*. The customized configuration byte provides the *Global Interrupt Enable* bit and the *Sleep* bit to put the processor into sleep mode. The customized status byte is saved to the *Saved Customized Status Register* upon an exception occurrence and restored from it after the execution of the interrupt handler.

In case of a reduced-16-bit data path the upper 16 bits of the frame pointer registers are unused. The frame pointer registers are described in Section 4.1.3.

Interrupts For enabling interrupts globally the `GIE` bit of the customized configuration register has to be set. To disable interrupts individually, a mask bit for each of the 16 interrupts is provided in the *Interrupt Mask Register*. The *Interrupt Protocol Register* captures incoming interrupts. The corresponding bit number is set until the interrupt service routine is executed or it is cleared by the programmer.

[2]The **A**rithmetic and **L**ogic **U**nit is a component in a processor that performs arithmetic and logical operations.

4.3 Instruction Set Architecture

The SCARTS instruction set comprises 122 instructions of which 48 are conditional. All instructions have a fixed length. They are encoded in 16-bit words, which reduces hardware complexity. They have a constant execution time of one clock cycle per instruction, which is an important feature for real-time capability. As only 16-bit instructions are used there was not enough scope to design an instruction set with a consistent structure. Therefore, the length of opcodes and immediate values vary.

4.3.1 Conditional Instructions

Besides conditional jumps, the instruction set also comes with conditional arithmetic and logic instructions. These instructions test the condition flag in the status register, which is set or cleared by compare and test instructions, respectively. If the condition applies the instruction is executed else it is replaced by a `nop` instruction. Thus, with the support of conditional instructions the execution time of programs is constant and independent of data. Hence, the worst-case execution time can be calculated easier. For small conditional sequences the execution time is even shorter, compared to conventional sequences, as no jumps and pipeline flushes are required.

CHAPTER 5

ASCARTS - The Asynchronous Processor

The asynchronous processor was developed to be compatible with the toolchain used for the synchronous counterpart. Hence, it is named ASCARTS, which stands for **A**synchronous **SCARTS**. In order to achieve this most of the just introduced features of SCARTS have been adopted for the implementation of ASCARTS.

As the asynchronous design style is quite different from the synchronous one, differences in the architecture are inevitable. Recall that in asynchronous designs the execution time varies depending on the instruction and its operands. Thus, worst-case execution time has to be calculated with greater expense, which in turn weakens the feature of real-time capability. The design of this processor is platform-independent. The ASCARTS can be used as a soft-core processor for FPGAs[1], or the design is synthesized with a library for ASICs[2]. The latter will result in a more efficient design in terms of area, power and performance as most of the FPGAs are not designed for implementing asynchronous circuits.

Besides the asynchronous design-related modifications, other decisions are changed as well. The customizable data path was not implemented in that way again. Only a 32-bit data path is available anymore. The 16-bit version was omitted as experience showed that the performance gain is disproportionate to the additional hardware resources used, especially because modern FPGAs provide enough resources. However, the instructions are still 16-bit words. The instruction set architecture was refined as well to facilitate the decoding process.

To keep the work within reasonable limits, the processor control module was slimmed down to the indispensable parts for the processor. Besides the exceptions, the sleep mode, which is provided for power saving in the synchronous processor, was omitted. The justification for implementing

[1] A **F**ield **P**rogrammable **G**ate **A**rray is an integrated circuit that can be programmed after manufacturing.
[2] A **A**pplication-**S**pecific **I**ntegrated **C**ircuit is an integrated circuit that is customized for a particular application.

the sleep mode in SCARTS is the high power consumption while the processor is idle. Due to the asynchronous logic design this justification is no longer met in ASCARTS. The optional extension modules have not yet been implemented either as their functionality is not sufficiently relevant for an asynchronous prototype of SCARTS.

5.1 Processor Interface

The interface of ASCARTS basically consists of the interfaces to the instruction memory and data memory, respectively. The data memory interface consists of the `read data` input signal and the `address` and `write data` output signals. Additionally, it includes the two control signals `write enable` and `byte enable`. The instruction memory interface has only the `read data` input signal and the `address` output signal. As this memory is read only no `write data` and `write enable` signals are needed. The `byte enable` signal can also be omitted as always the whole instruction word is read in ASCARTS. The interface of ASCARTS is depicted in Figure 5.1.

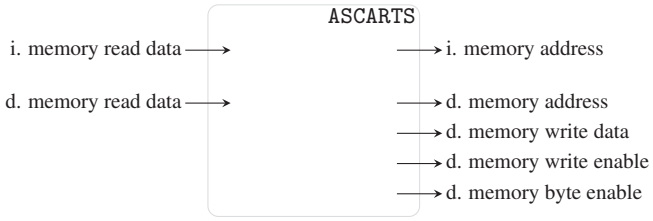


Figure 5.1: ASCARTS Interface

5.2 Processor Architecture

The ASCARTS coarse pipeline structure is identical to SCARTS. The processor's pipeline is four stages long. These comprise *Instruction Fetch*, *Instruction Decode*, *Execution and Memory Control* and *Write Back*. Between these stages buffers latch the data and control signals. Otherwise the first stage would have to wait until the last stage processed its data, which makes a pipelined structure useless. The third stage combines the execution and memory access as the ALU is barely used by memory access instructions. Only some need the adder for address calculations. While executing one of these instructions the processor slows down a bit. On the other hand an almost unnecessary stage can be omitted, which decreases required resources.

The asynchronous pipeline architecture causes complicated data and control hazards. Innovative solutions are required to resolve these hazards. A data hazard occurs when an instruction depends on a result of a preceding instruction that is still processed in the pipeline. In SCARTS the instructions proceed in the pipeline simultaneously. Therefore, it is known if and from which stage the result has to be forwarded. For ASCARTS an asynchronous forwarding method was developed, which is based on information stored about previous instructions. As the order of the instructions is preserved in the pipeline, a data hazard can be detected knowing the previous operand registers. A more detailed explanation can be found in the description of the affected

units. A similar approach was already implemented for the asynchronous SAMIPS processor [23]. As the order of the instructions is preserved, i.e. instructions cannot overtake each other in the pipeline, on a register read write-dependencies can be exposed with the stored information.

A control hazard occurs when an instruction or exception changes the control flow. In such a case the pipeline has to be flushed and the results of prefetched instructions have to be discarded. In SCARTS the exact number of prefetched instructions is known, and therefore, can be discarded with a simple counter. To resolve control hazards in the asynchronous design, a *coloring* algorithm similar to that was already used for AMULET1 [22] and SAMIPS [23] was implemented. The current state of the processor and the instruction words in the pipeline are colored. The instruction words have the color of the processor state at the time they were fetched. Every time a control hazard occurs the color of the processor state changes. An instruction is only processed in the execution stage if its color matches the color of the processor. Thus, instructions following a control hazard are discarded until instructions with the new color arrive.

As memory access is clocked a controller is needed to transform handshakes into a clock signal. The Balsa language does not support clocked logic, therefore, the memory is excluded from the asynchronous design. The asynchronous processor design has two memory ports to feed the separated external instruction memory and data memory of the *Harvard Architecture*.

The block diagram of ASCARTS is depicted in Figure 5.2.

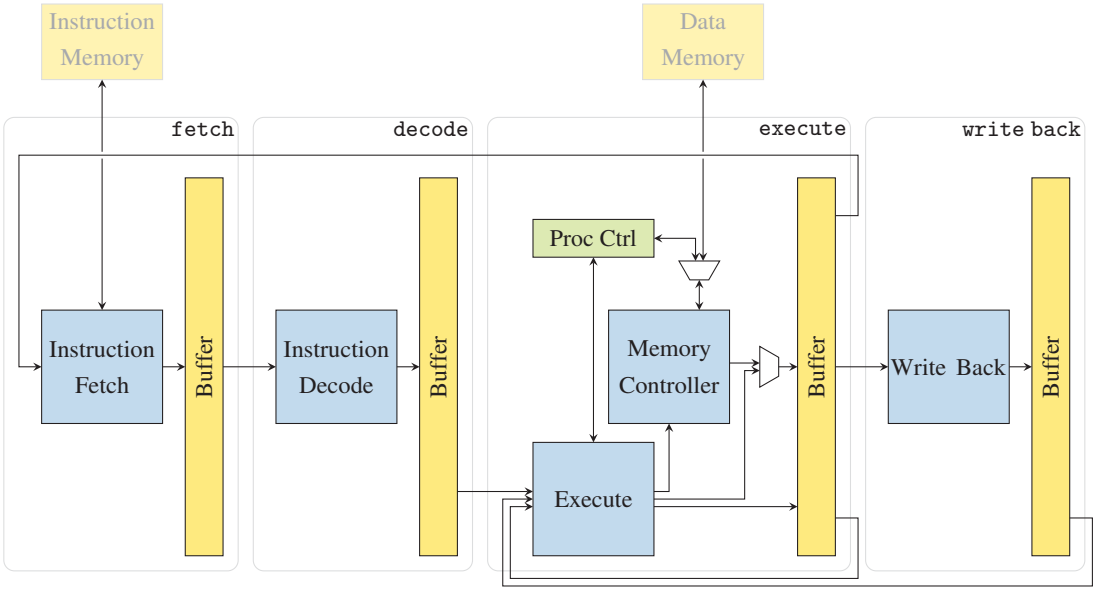


Figure 5.2: Block Diagram of ASCARTS

5.2.1 Fetch Stage

The first pipeline stage is *Instruction Fetch*. In this stage the physical instruction memory address for the next instruction is calculated. It can either be the current program counter plus one

or in case of a branch the branch address. In the second case the color, which is attached to the instruction words, is updated as well. The actual address is selected by an arbiter. Due to varying delays it is not predictable which of the possible new program addresses arrive first. As handshakes are used for synchronization, the address that arrives first is taken. Hence, in contrast to SCARTS, the behavior is non-deterministic and the number of prefetched instructions in the pipeline is not predictable.

In Balsa to be able to distinguish between the next calculated program counter and a branch address the former needs a sync pulse the arbiter can trigger on. Whenever the next program counter is selected a new sync pulse is generated. See Listing 5.1 for the code snippet. Due to handshake enclosure for choice operators, as described in Section 2.2.3, two sync buffers are needed for the sync pulse. With only one sync buffer in the `arbitrate` statement the handshake for `next_pc_dly_sync` would only be completed after the handshake for the subsequent `next_pc_sync` was completed. But the sync buffer only completes a handshake for another `next_pc_sync` once the handshake for the previous `next_pc_dly_sync` was completed. This would result in a deadlock.

```
SyncBuffer(next_pc_sync, next_pc_dly_sync) ||
SyncBuffer(next_pc_dly_sync, next_pc_2dly_sync) ||

begin
  sync next_pc_sync ;
  loop
    ...

    arbitrate next_pc_2dly_sync then
      ... ||
      sync next_pc_sync
    | branch_address_i then
      ...
    end
  end
end
```

Listing 5.1: Balsa Code for Arbiter

The calculated physical address is sent to the instruction memory. The received instruction word is colored and forwarded along with the address of the subsequent instruction word to the decode stage. Figure 5.3 shows the block diagram of this stage.

5.2.2 Decode Stage

The second pipeline stage is *Instruction Decode*. This stage decodes the highest bits of the instructions as opcode. Based on the opcode operands or their addresses are extracted and control signals for the subsequent stages are generated. Besides the decoding unit, the register file resides in this stage. Thus, register access is carried out as well. The detection of data hazards is also performed in a separate unit. The obtained signals from the latter unit are transferred to the

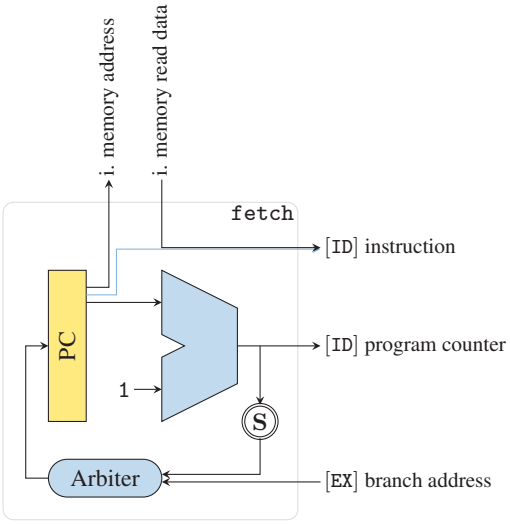


Figure 5.3: Fetch Stage

forward control unit in the execution stage. The obtained operands and the generated control signals of that stage, including the color attached to the current instruction, are forwarded to the subsequent stages. Figure 5.3 shows the block diagram of this stage.

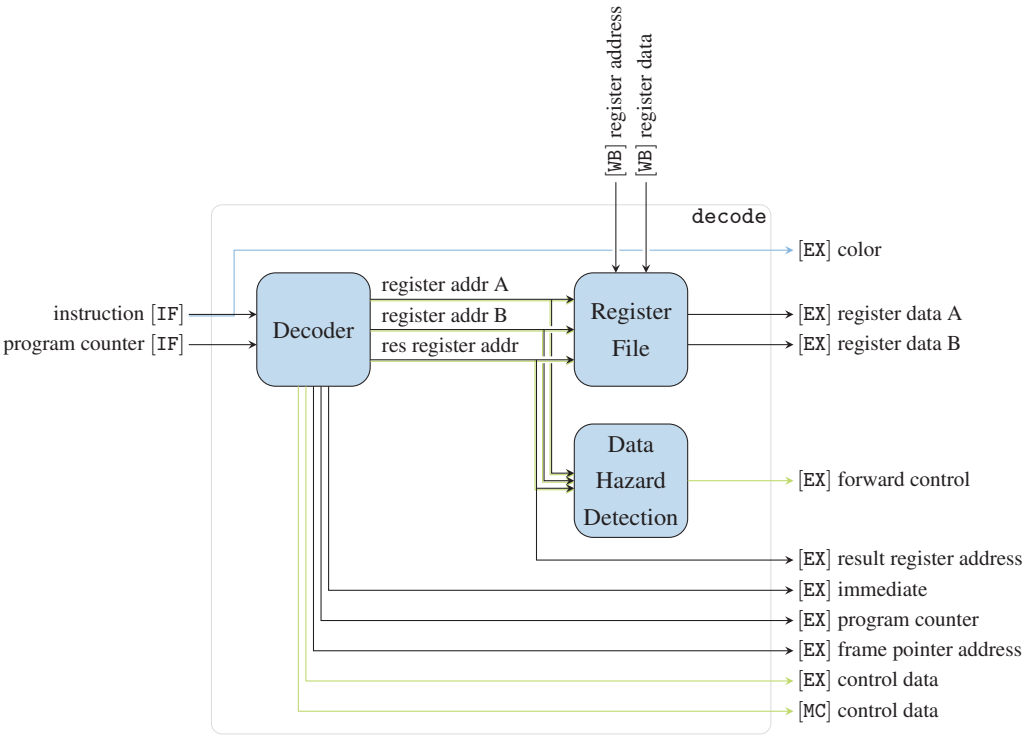


Figure 5.4: Decode Stage

Decoder

In the decode unit the actual instruction decoding takes place. Register addresses, immediates and frame pointer addresses are extracted. The program counter is forwarded in case it is needed in the execution stage. Control signals indicating if a register is actually read or written back are generated. These signals are forwarded to the register file as read enable and to the data hazard detection control, respectively. Furthermore, control signals for the subsequent stages are generated. These signals comprise the condition type, the function and the memory code, which define the operation to be performed, for the execution and the memory control stage, respectively, and a signal for arithmetic operations if it is with carry.

Due to a redesign of the instruction set some operands, operand addresses and control signals can be generated without decoding the whole instruction. As the composition for the instructions is structured the decoding can be split into conditional and non-conditional instructions. Within these it can be further split into instructions with register operands and immediate operands. Most of the instructions can be summarized in groups so that the whole group can be decoded together.

Because of the asynchronous nature signals are not sequentially transmitted and can arrive at the next stage in a different order than they are issued. To avoid inconsistency, for the data and control signal generation handshake enclosure is used. Only once the handshake of every single signal is completed by the buffer the instruction is considered as decoded. The intention was to forward signals only if they are actually needed. This, however, may induce hazards. In particular, the execution stage may receive exclusive operands from consecutive instructions simultaneously, which causes a race condition. Subsequently, the processor may malfunction or deadlock. For example, instruction decode transmits a register value as second operand. For the subsequent instruction an immediate as second operand has to be transmitted. As there is no pending handshake on the immediate channel a new handshake can be started immediately. Once the execution stage tries to read the second operand of the previous instruction two values are available. For cases like this, additional control signals are generated, which specify the source to overcome that problem. For each conflict the additional signal avoids an early issue of the subsequent operand before the previous one is read.

Register File

The register file contains 16 registers, which are accessed by two read ports and one write port. Register file reading addresses are directly passed from the decode unit. The register file writing address is passed from the write back stage. Read and write accesses have to be serialized to avoid invalid data. The type of access is selected by an arbiter. Due to varying delays it is not predictable which of the accesses occur first. Hence, in contrast to SCARTS, the behavior is non-deterministic. In case of a read access all ports to be read for the current instruction are read concurrently. To indicate if a read port has to be actually read, in addition to the data and address channels, the two read ports are each accompanied by a control channel.

Data Hazard Detection

To identify possible data hazards, a data hazard detection table is utilized. The stored information about the preceding instructions is used to generate forward control signals for proper register forwarding in the subsequent stages. As most of the instructions are conditional not all expected results may be calculated. Hence, information of all instructions that may write back their results has to be stored and passed to the forward control unit in the execution stage. Additionally, the current values of all operand registers are passed on as well in case that the conditional instructions are not executed.

All write back register addresses of the preceding instructions still being processed in the pipeline are stored in the history table. As far as ASCARTS is concerned instructions being processed in the execution and memory control stage and in the write back stage are affected. As at most two instructions that may produce register results reside in those stages, the table only needs two hold two entries, one for each stage. Each entry consists of two items. One bit indicates if the instruction in the corresponding stage will write back its result provided that it is executed. The second item specifies the address of the register written to. In Table 5.1 an example of a data hazard detection table is illustrated.

Stage	**Pending Write**	**Register Address**
Execution	`t`	13
Write Back	`f`	/

Table 5.1: Example of Data Hazard Detection Table

The data hazard detection unit is called whenever a new register read request is issued. It consists of two parts. For each register read the data hazard detection table is checked if it may be overwritten by an instruction still in the pipeline. Along with the actual register data control signals are passed to the forward control unit in the execution stage. One control signal specifies if the data read from the register file is valid or if the forwarded result of one of the two stages has to be taken. As in asynchronous logic every data transfer has to be acknowledged all forwarded results have to be acknowledged as well, no matter if the current operation actually needs it or not. Therefore, for each subsequent pipeline stage the other control signal is issued whenever a forwarded result of that stage is expected. It specifies if that result has to be kept for further usage or if it can be discarded. In the second part the entry at the first slot is shifted to the second one in the table. Then the first slot is filled with the write back register information of the current instruction.

The advantage of this mechanism is that the data hazard detection table does not need to be updated on the actual register write back in the write back stage. By implementation only the information of the instructions still in the pipeline is stored. Hence, the worst case that can happen is that the forward control unit takes the forwarded results, even though the registers have already been up-to-date.

5.2.3 Execution Stage

The third pipeline stage is *Execution*. The core of this stage is the ALU. Besides arithmetic and logic operations, it calculates memory address and memory data as well as the branch target address, which is passed to the instruction fetch stage. In the forward control unit the operand values of the ALU are updated based on control signals from the decode stage and forwarded results from the subsequent stages. In the control hazard detection unit the coloring algorithm is implemented to handle control hazards, in particular branches. Additionally, this stage has an interface to the memory control stage. Apart from the address and data channels a control signal to enable the memory is part of the interface. As the frame pointer and status register reside in the processor control module an interface to that module is available as well. Multiplexers and de-multiplexers are used to select the ALU operands on the one hand and the memory address and data on the other hand. Figure 5.5 shows the block diagram of this stage.

A special multiplexer, the result multiplexer, selects the result's source between the execution and memory control stage, which is then forwarded to the write back stage. Refer to Figure 5.2 for the result multiplexer.

ALU

For each ALU operation the handshakes with common signals are established. These include the function code, which defines the operation to be performed, the ALU enable signal from the control hazard detection unit and the condition type. All other handshakes are only established if they are needed for the specific operation. The condition type is evaluated and compared with the condition flag in the status register. If the condition flag matches or the instruction is non-conditional the operation is performed. If the operation is not performed the remaining handshakes have to be acknowledged. These comprise the ALU operands and function specific control signals. If those signals were not consumed no new ones could be generated. This would lead to a deadlock.

For add and shift operations there is one shared procedure each, which is used for all functions. However, there is a separate adder to generate the branch address of the jump immediate instruction as the design of the generic adder does not fit for this addition.

The flags for the status register are also computed. For the arithmetic, shift and compare functions all flags are generated. For logical functions the overflow and carry flags remain unmodified. The compare and bit test functions additionally compute the condition flag.

In addition, signals for frame pointer access in the processor control module are generated. Once the result, i.e. the frame pointer value, is available the memory address is calculated. As the frame pointer can also be manipulated the ALU has to synchronize on the control signal indicating the completion of the frame pointer manipulation. Refer to Section 5.3.1 for a more detailed description of the frame pointer bank.

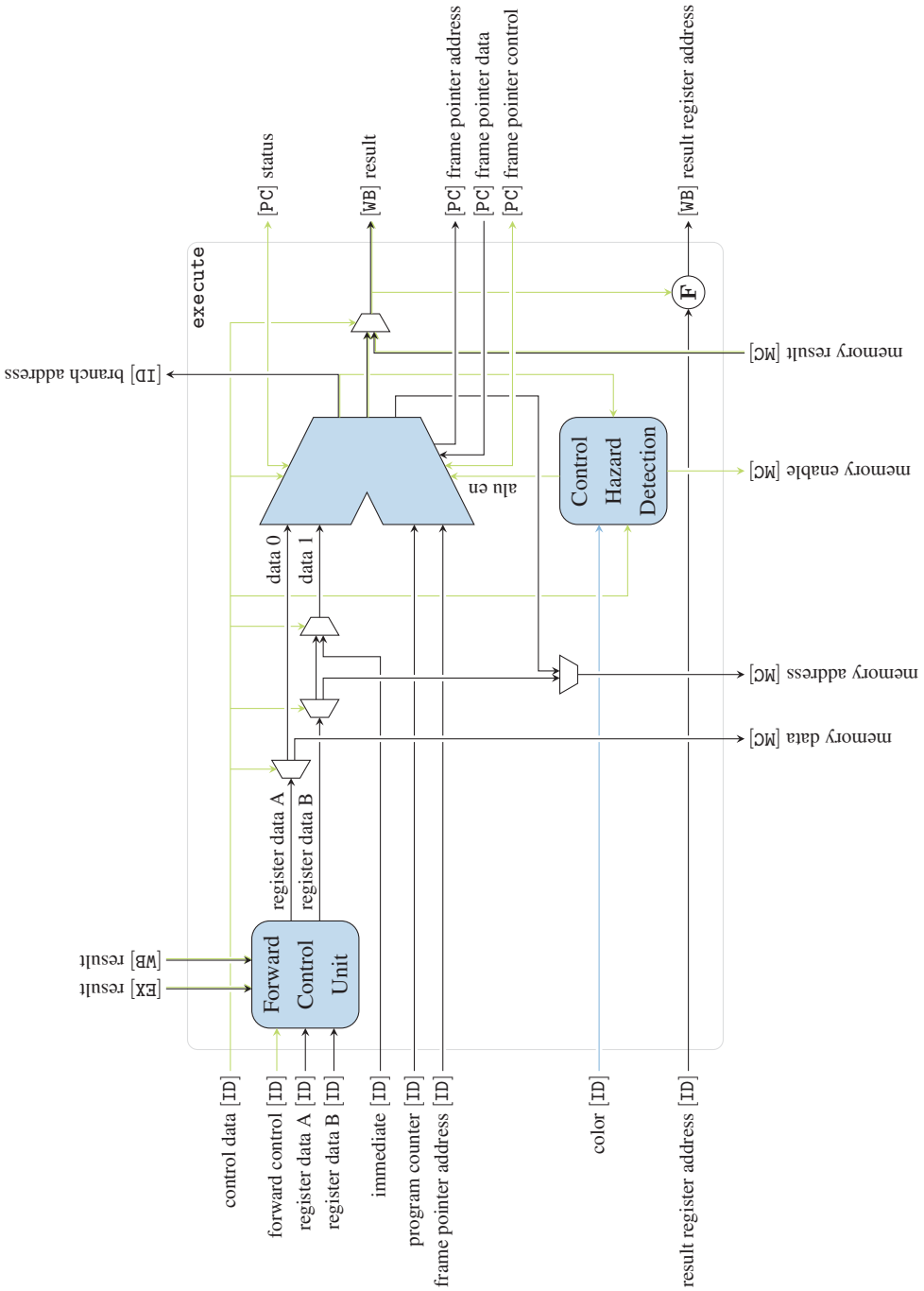


Figure 5.5: Execution Stage

Forward Control Unit

The forward control unit provides the operand values on the basis of control signals generated in the data hazard detection unit. It consists of two phases. First, the forwarded results of each subsequent pipeline stage are analyzed if they have to be forwarded as they update one of the read registers of the current instruction. Otherwise the forwarded results are discarded. In the second phase, starting with the read register values, the actual operand values are updated by examining the forwarded results from the subsequent pipeline stages in reverse order, i.e. first the result of the write back stage, then the one of the execution stage. If the instruction currently processed in the pipeline stage under examination produces a valid result and that result was kept in the preceding phase the value of the specific operand is updated. Once all pipeline stages are examined the current operand values are returned.

The forwarded results have to be buffered before being processed to not cause any deadlocks in this unit. On basis of the results of the preceding instructions this unit generates operands, which in turn are used to generate the result of this instruction. Since the whole stage is enclosed in one handshake the calculated result of the current instruction will not be latched before the result of the previous one is released. But as the result of the previous instruction is looped back to this stage as input, this input and, hence, the result of the previous instruction is only released once the result of the current instruction is latched. Hence, a deadlock would occur without buffers.

Control Hazard Detection

The current state of the processor is stored as color in a register. Whenever a control hazard occurs the color of the processor is updated. For each instruction the passed color is compared with the stored color. If the instruction was fetched before the processor state changed the colors do not match. Consequently, the instruction is discarded. In case of identical colors the instruction is processed. The appropriate control signals to enable and disable ALU and memory for the inspected instruction are generated.

Result Multiplexer

The result multiplexer selects the result's source based on a control signal generated in the decode stage. The signal specifies whether the result is taken from the execution stage or from the memory control stage. Along with the result a control signal indicating the validity of the result is passed to the write back stage. A result is denoted invalid if there is no result because the instruction was not processed due to a failed condition or a control hazard. This signal is required by the forward control unit. That unit issues a handshake request if a result is expected. With that control signal the handshake can be acknowledged if no actual result exists. Additionally, the register address is passed through this unit if the result is valid, otherwise it is discarded.

5.2.4 Memory Control Stage

The *Memory Control* is linked to the execution stage without a buffer. In this stage the signals of the memory interface are generated. Write enable and byte enable signals are generated based on

the type of memory access. Write data is aligned in accordance to the data access type. Read data is converted to fit the signed or unsigned data access, depending on the instruction.

Depending on the memory address the internal dynamic memory or an extension module is accessed as represented in Figure 4.2. If the memory address is valid and the instruction is executed, indicated by the enable signal from the execution stage, handshakes to the selected memory are established. If unused ranges are addressed or the instruction is not executed no memory access signals are generated. In case of a write access the write data is discarded. In case of a read access instead of the result a sync handshake is generated to signal that no result will be generated for this instruction. With this control signal deadlocks are avoided as other stages do not wait any longer for the result. Figure 5.6 shows the block diagram of this stage.

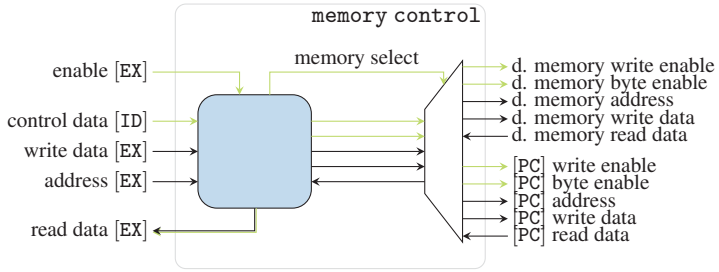


Figure 5.6: Memory Control Stage

5.2.5 Write Back Stage

The last pipeline stage is *Write Back*. If there is any result of this instruction, it is written back to the dedicated register. The write back operation actually occurs in the register file unit.

5.3 Extension Modules

The modular interface for extension has not changed from SCARTS. For a detailed description of extension modules refer to Section 4.2. In Figure 4.3 the register interface specification of extension modules is depicted.

However, the generic parts of the status register and the configuration register are not yet implemented. These bytes provide solely debug and status information, as well as functionality which is not essential for the proper functioning of the asynchronous processor. Hence, the generic bytes of extension modules are empty, see Figure 5.7.

5.3.1 Processor Control Module

The core functionality of the processor control module is the same as for SCARTS. It contains the processor status register and the frame pointers. In Figure 5.8 the register interface of the slimmed processor control module is depicted. Figure 5.9 shows the register description of the customized parts of the status register and the configuration register.

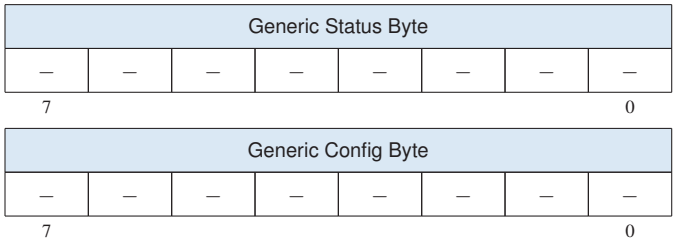


Figure 5.7: Generic Status and Configuration Byte

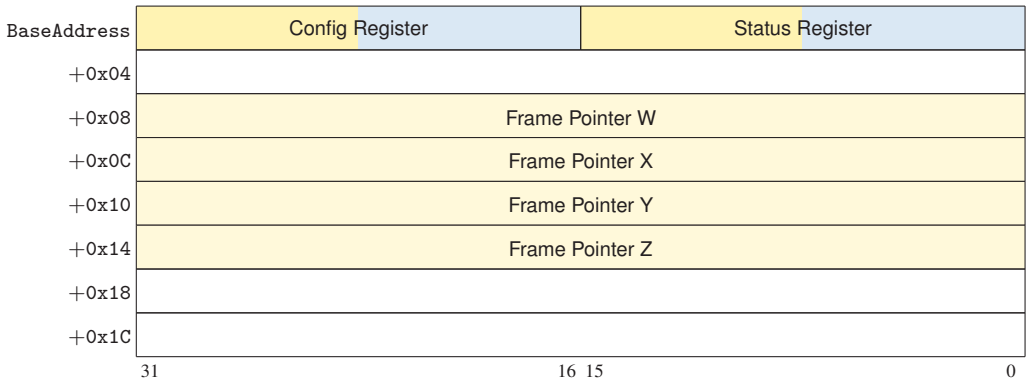


Figure 5.8: Register Interface for Processor Control Module

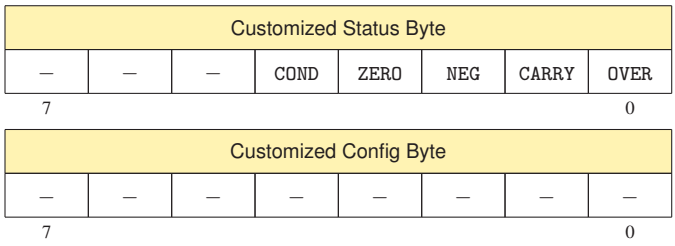


Figure 5.9: Customized Status and Configuration Byte of Processor Control Module

The customized part of the status register is identical to the one of SCARTS. As no exceptions are implemented and the sleep mode was omitted the customized part of the configuration register is empty. In SCARTS the customized status byte can be externally written by writing the new data to the saved customized status register and calling the instruction `rte` afterwards. As exceptions and, hence, `rte` were not implemented in ASCARTS this functionality is lost.

The processor control module contains individual units for each logical part of the processor control. The status control unit handles the read and write access of the status flags of the ALU. The frame pointer bank accesses and modifies the frame pointers. An additional unit, the processor control unit, routes the signals from the data memory and from the ALU, respectively, to service the individual units within this module. Figure 5.10 shows the block diagram of this extension module.

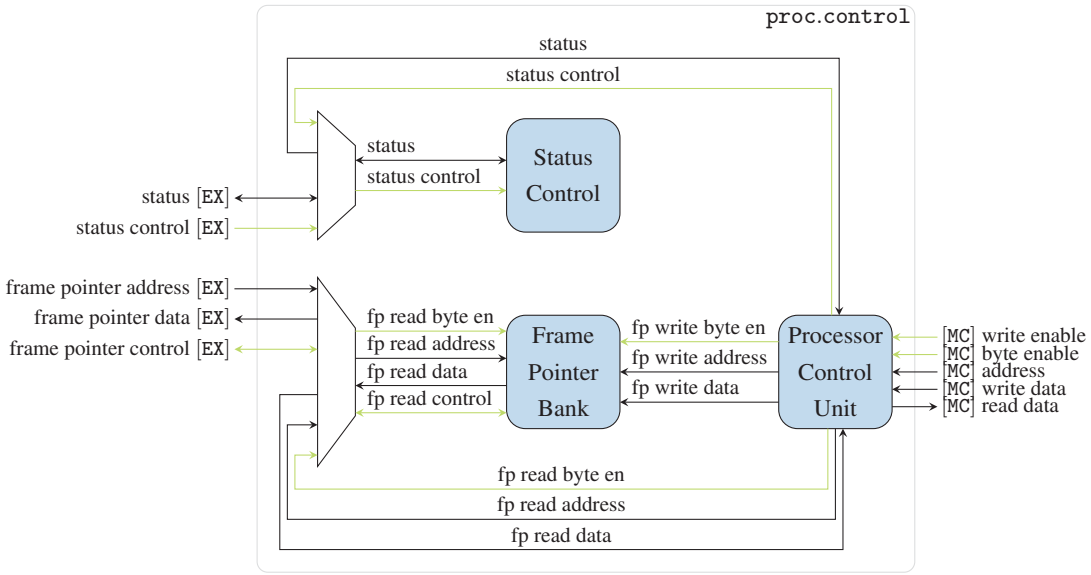


Figure 5.10: Processor Control Module

Processor Control Unit

As the status control unit and the frame pointer bank can both be manipulated via the data memory as well as directly by the processor a control unit to route the handshake signals of these entities is needed. This routing is the task of the processor control unit. In the actual processor control unit the communications received on the channels from the data memory are split so that only the information for a specific unit is provided to that unit. Based on the address offset the appropriate unit is selected and the corresponding signals are generated. These signals are then multiplexed with the signals received from the ALU. For each obtained signal to a unit the handshake signals on the corresponding channels are initiated. The write port of the frame pointer bank is routed directly since the ALU has only read access. Similarly, only the read port of the status control unit has to be multiplexed as there is no write access from the processor control unit.

Frame Pointer Bank

In the frame pointer bank, besides read and write access, the frame pointers can be manipulated after read. For this purpose, in addition to the conventional channels of the read port, a control signal is transmitted. This signal indicates if a manipulation has to take place and whether the frame pointer has to be incremented or decremented. To improve the performance, the manipulation takes place after the read result is returned. With this optimized technique the ALU can already use the frame pointer earlier. To avoid race conditions, a signal is generated after the manipulation is completed. The ALU has to synchronize on this before another access to the frame pointer bank is performed.

However, the implementation of the frame pointer bank differs in one corner case from the synchronous implementation. If the manipulated frame pointer resides on the mapped memory address as read by the frame pointer instruction, the frame pointer after manipulation is read. However, in the synchronous design the frame pointer before manipulation is read. The reason for that is that as the synchronous design is clocked the frame pointer is only updated with the next clock pulse. However, in the asynchronous design the frame pointer is updated immediately. Delaying the frame pointer update to after an optional access would cause a huge overhead.

5.4 Instruction Set Architecture

The instruction set architecture of SCARTS is quite efficient, however, the encoding of the opcodes[3] can be improved. The purpose of a revised instruction set architecture is to organize the opcodes, so that not all bits have to be decoded to obtain the operation to be performed.

The formats of the instructions in the ASCARTS architecture remained unchanged solely the encoding of the opcodes was redefined. Additionally, the explicit illegal operation instruction was omitted as the processing of that one does not differ from any other illegal instruction, i.e. instructions that are not defined in the instruction set. Thus, the resulting instruction set architecture comprises 121 instructions of which 48 are conditional.

5.4.1 Instruction Format

The instructions with fixed length encoding of 16 bit can be distinguished into the formats illustrated in Figure 5.11. Appendix B.1 lists all instructions sorted by their opcode. The most significant bit of the opcode splits the instruction set into two groups, the conditional instructions and the non-conditional instructions. The group of conditional instructions, however, also include non-conditional instructions.

Conditional Instructions

The opcode for all conditional instructions is constructed identical as depicted in Figure 5.12 and described in Table 5.2. Instructions with condition type of `01` are always the same as the ones with condition type of `1x`, only ignoring the condition flag. Instructions with condition type of `00`,

[3] An opcode or **op**eration **code** specifies the operation to be performed.

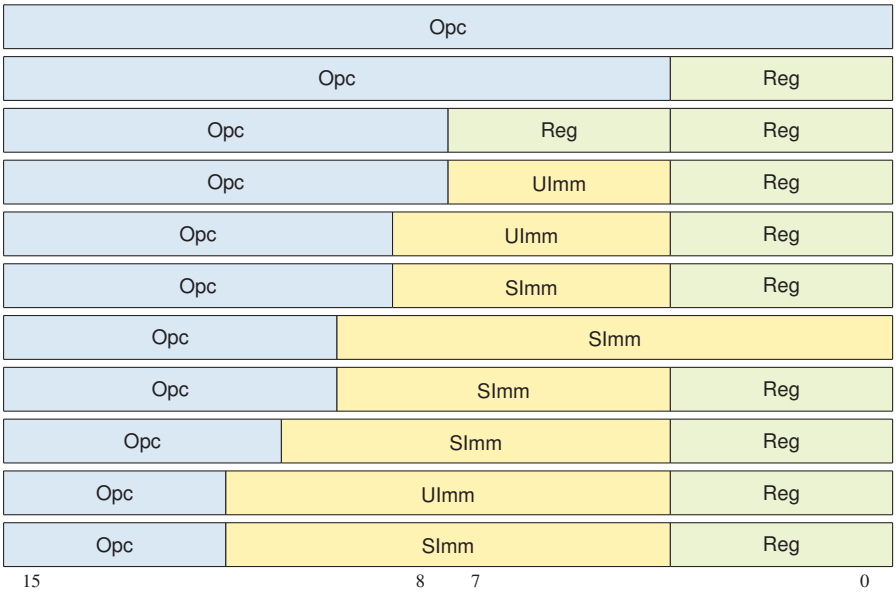


Figure 5.11: Instruction Formats

on the other hand, are independent, truly non-conditional, instructions residing in the conditional instructions group.

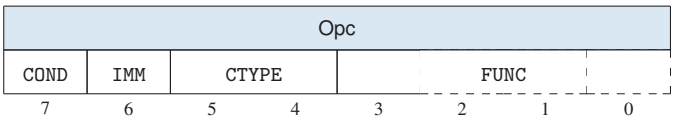


Figure 5.12: Opcode for Conditional Instructions

`COND`	The *Condition* bit splits the instruction set in conditional and non-conditional instructions. It is `0` for conditional instructions.
`IMM`	The *Immediate* bit specifies if the instruction has one immediate operand.
`CTYPE`	The two *Condition Type* bits define the circumstances under which the instruction is executed. `0x` The instruction is always executed. `10` The instruction is executed if condition flag is false. `10` The instruction is executed if condition flag is true.
`FUNC`	The *Function Code* declares the actual operation which has to be processed. It varies in length depending on the width of the operands.

Table 5.2: Opcode for Conditional Instructions

The function code encodes the basic function to be processed as well as additional parameters

needed the function. The encoding was constructed in a way to forward bits directly as parameters to the ALU without decoding them. Arithmetic functions can be executed with or without carry bit. Shift functions need the information if the shift is processed arithmetically or logically. Comparison and memory access functions can be signed or unsigned. This information is always stored in the least significant bit of the function code. For memory access functions another bit specifies if it is a load or a store function so that the remaining bits only define the addressing quantities of the function. Furthermore, the function code is chosen in a way so that for the same function the same code is used, even if the type of operands or any other parameters differ.

5.4.2 Non-Conditional Instructions

The remaining non-conditional instructions all have one register operand and one immediate operand. However, the length of the immediate varies. Most of these instructions address the data memory relative to frame pointers. The composition of the opcodes for frame pointer instructions is depicted in Figure 5.13 and described in Table 5.3.

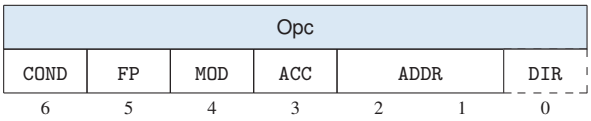


Figure 5.13: Opcode for Frame Pointer Instructions

`COND`	The *Condition* bit splits the instruction set in conditional and non-conditional instructions. It is `1` for conditional instructions.
`FP`	The *Frame Pointer* bit declares if the instruction addresses the data memory relative to frame pointers. It is `1` for frame pointer instructions.
`MOD`	The *Modification* bit specifies if the frame pointer is manipulated by auto-increment or auto-decrement.
`ACC`	The *Access* bit defines if it is a load or store instruction.
`ADDR`	The two *Address* bits specify the frame pointer used.
`DIR`	If the *Modification* bit is not set the *Direction* bit declares if the frame pointer manipulation is auto-increment or auto-decrement. Otherwise that bit is used as an additional bit for the immediate operand.

Table 5.3: Opcode for Frame Pointer Instructions

In addition to the modification of the encoding, instructions that load immediates were renamed to more significant acronyms. The affected instructions are listed in Table 5.4.

Above all, the opcodes of all instructions in the instruction set are chosen so that the `nop` instruction has the encoded value `0x0000`, which is recommended practice. To stay compatible

ASCARTS	SCARTS	
`ldi`	`ldi`	loads the sign-extended immediate into the register.
`ldli`	`ldliu`	loads the immediate into the low byte of the register.
`ldhi`	`ldhi`	Loads the sign-extended immediate into the high three bytes of the register.

Table 5.4: Load Immediate Instructions for ASCARTS And SCARTS

with the SCARTS toolchain, a script is provided, which converts the object files from the SCARTS Instruction Set Architecture to the ASCARTS Instruction Set Architecture.

Detailed information about the individual instructions can be found in Appendix B.

CHAPTER 6

Design Flow

Since most of the current FPGAs are designed and optimized for implementing synchronous circuits, mapping delay-insensitive dual-rail designs for FPGA technology would result in high overheads. To obtain more reasonable area and performance results, the ASCARTS processor was implemented in UMC's 90 nm technology with an industrial standard cell library from Faraday[1]. Since a full layout was beyond the scope of this master thesis, all results are based on the synthesized netlist and on pre-layout timing simulations, annotated with rough timing estimates of the synthesis tool.

For design verification at different points of the development process three levels of simulation are available. The Balsa description of ASCARTS is compiled to an intermediate Breeze description. Based on the Breeze description behavioral simulation is done by the technology independent simulator included in Balsa. The Balsa processes are synthesized using the four-phase dual-rail protocol implementation style. The obtained register-transfer level netlist is verified by a structural simulation. After the synthesized netlist has been mapped to UMC's 90 nm technology a timing simulation is performed on the obtained gate-level netlist. Figure 6.1 shows an overview of the design flow for ASCARTS.

6.1 Behavioral Simulation

Behavioral simulation verifies syntax and functionality of a design containing high-level constructs. Neither timing information nor specifics on how the design will be implemented are provided. During the design process functional errors are identified early with Balsa's own behavioral simulation system. A test harness, this is a procedure without parameters, has to be generated. Test sequences are described in this procedure with Balsa. With `breeze-sim` the test harness procedure is called and a textual appearance of the output is produced. Alternatively the simulation can be run from the graphical interface `balsa-mgr`. Intermediate files are generated,

[1] http://freelibrary.faraday-tech.com/

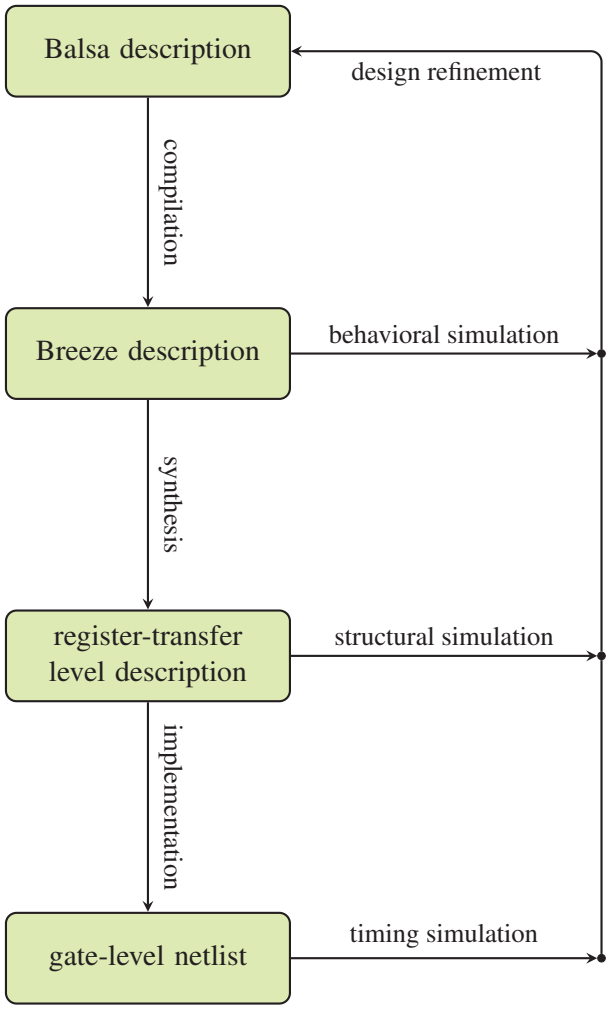


Figure 6.1: Design Flow of ASCARTS

which can be used in the waveform viewer or for a representation of the handshake circuit graph. At that stage the simulation is independent of the handshake protocol.

Balsa contains a built-in memory model, which can be included in the test harness for simulation purposes. With the built-in function `BalsaMemoryNew` a memory object is instantiated, which is interfaced by the procedure `BalsaMemory`. Additionally, the test harness comprises a separate load memory procedure for initializing the instruction memory. To accomplish that, the built-in function `FileOpen` is used to open an object file containing the program to be simulated. To reflect the *Harvard Architecture*, two instances of Balsa memory are employed, one for instruction memory and one for data memory. The instantiation of the memories is presented in Listing 6.1. These built-in memories are only used for behavioral simulation.

```
inst_mem := BalsaMemoryNew () ||
data_mem := BalsaMemoryNew () ;

BalsaMemory({inst_mem_addr_width, inst_mem_data_width},
  <- inst_mem,
  inst_mem_addr_s,
  inst_mem_write_enable_s,
  inst_mem_write_data_s,
  inst_mem_read_data_s) ||

BalsaMemory({data_mem_addr_width, 8},
  <- data_mem,
  data_mem_addr_s,
  data_mem_write_enable_s,
  data_mem_write_data_s,
  data_mem_read_data_s) ||

begin
  load_mem(<- "main.obj", inst_mem_addr_s, inst_mem_write_data_s,
      inst_mem_write_enable_s) ;
  loop inst_mem_write_enable_s <- 0 end ||
  ascarts(inst_mem_addr_s, inst_mem_read_data_s, data_mem_addr_s,
      data_mem_read_data_s, data_mem_write_data_s, data_mem_write_enable_s)
end
```

Listing 6.1: Balsa Memory Instantiation

6.1.1 Instruction Memory

The data width of the instruction memory instantiated is 16 bits as the memory to be simulated holds 16-bit instruction words. This simplifies the memory access as only one word access for each instruction fetch is needed.

6.1.2 Data Memory

A drawback of the built-in Balsa memory interface is that it does not contain byte-enable channels. Hence, the data memory had to be instantiated with a data width of 8 bits. To provide word and half-word access as well, a converter was written, which translates the word and half-word accesses into byte accesses. This is done by serializing the corresponding number of byte accesses. That approach leads to several cycles for one data memory access. However, this does not affect the functionality of the design as logic has zero delay in behavioral simulations.

6.2 Synthesis

After the required functionality has been achieved the design is synthesized[2] with the Balsa tools to get a register-transfer level netlist. For synthesizing systems described in Balsa a technology and

[2] Synthesis converts the functional design description to low-level hardware primitives.

a back-end protocol for implementing the handshakes have to be chosen. Different technologies and handshake implementation styles available in the Balsa framework are presented in *Balsa: A Tutorial Guide*[4]. As far as ASCARTS is concerned the `balsa-tech-example` technology is used to produce a Verilog netlist. This technology is adequate since it is based on a reasonable set of example cells described in Verilog, which can easily be mapped to the gates of the targeted UMC's 90 nm technology standard cell library in the subsequent step. Thus, creating an own Balsa technology back-end for the target library can be avoided. The selected handshake protocol implementation is the *dual_b* style, a delay-insensitive four-phase dual-rail encoding.

6.3 Structural Simulation

Once the Balsa design is successfully synthesized it can be simulated like every other HDL design. The simulator of choice is Mentor ModelSim[3]. The post-synthesis netlist is analyzed by structural simulation revealing structural errors like initialization issues. Like behavioral simulation structural simulation is accomplished without timing information.

6.3.1 Memory Access

The memory interface provided by the register-transfer level netlist incorporates asynchronous ports, whereas memories are accessed in a synchronous way. Hence, the asynchronous handshake signals have to be transformed to synchronous signals. Besides, a clock has to be generated whose pulses trigger the start of memory operations.

Depending on the direction of the interface signals they are implemented as pull channels or as push channels. Therefore, read data, which is an input to ASCARTS, is implemented as pull channel. All other signals are outputs and push channels, respectively. For the request signals of push channels completion detection circuits are needed, which generate control signals indicating validity. Figure 6.2 shows a completion detection circuit for a 3-bit dataword. The `data.valid` signal indicates that a valid dataword is available. Likewise, the `data.invalid` signal indicates that all data wires are reset, representing the empty codeword. For each of the push channels this completion detection circuit is adapted to fit the data width. For 1-bit datawords only one control signal is needed, which is generated by a circuit consisting of only one `or` gate combining the two wires.

The following source code snippet shows the generation of these signals for the data memory address. As far as the encoding is concerned the second wires carry the actual dataword.

```
data_mem_addr_valid_s <= and_vector(data_mem_addr_s_0r0d or
    data_mem_addr_s_0r1d);
data_mem_addr_invalid_s <= not or_vector(data_mem_addr_s_0r0d or
    data_mem_addr_s_0r1d);
data_mem_address_s <= data_mem_addr_s_0r1d;
```

Once all required request signals are valid a pulse on the memory clock is generated to indicate that datawords at the memory interface are valid and ready for being processed. In case of a

[3] http://www.model.com

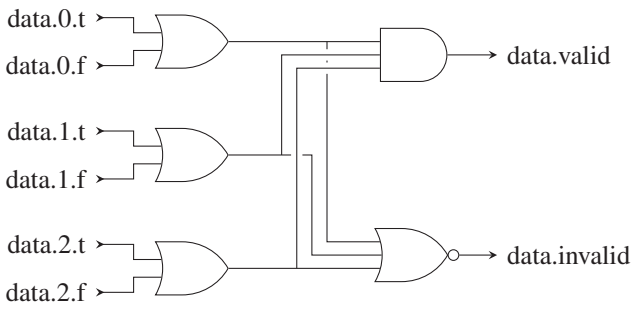


Figure 6.2: Completion Detection

memory write the write data has to be present and in case of a memory read a request for the read data must have been placed. The pulse generation for clocking the data memory is presented in the following source code snippet.

```
data_mem_clk_s <= data_mem_addr_valid_s and data_mem_write_enable_valid_s and
     ((not data_mem_write_enable_s and data_mem_read_data_s_0r) or
    data_mem_write_data_valid_s) and data_mem_byte_enable_valid_s(0) and
    data_mem_byte_enable_valid_s(1) and data_mem_byte_enable_valid_s(2) and
    data_mem_byte_enable_valid_s(3);
```

Once the data is successfully written to or read from the memory, respectively, all request signals are acknowledged. In case of a memory read the read request is acknowledged by a valid dataword. Memory access completion is signaled by a rising edge of a second clock, which is actually the memory clock delayed by the access time of the memory block. To achieve the delay, serially linked buffers are placed behind the memory clock. The length of the delay can be easily adapted by changing the amount of buffers linked together. For each channel the acknowledge signal is deasserted once the associated request signal becomes invalid. Figure 6.3 shows the conversion between the asynchronous handshake protocol and the synchronous memory access for the data memory.

6.3.2 Instruction Memory

A synchronous RAM with a data width of 16 bits is used as instruction memory. The write port is not in use as the instruction memory is read-only once it is initialized. The instruction memory is preloaded by ModelSim as part of the simulation process. The initialization values are read from the object file provided as a parameter.

6.3.3 Data Memory

To realize byte access, the data memory is implemented by four parallel synchronous RAMs of 8-bit words. Each byte of the 32-bit datawords is assigned to one specific memory block specified by the two least significant bits of the memory address. The rest of the memory address is used to index the bytes of the datawords within that memory. In ASCARTS all four memory blocks

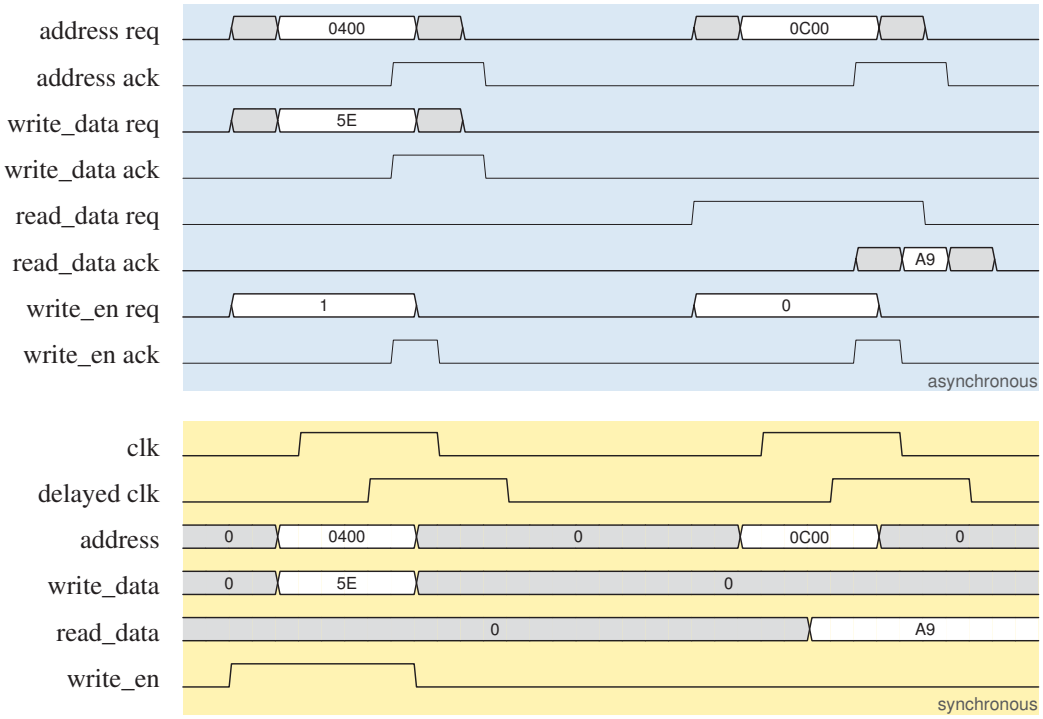


Figure 6.3: Synchronous Data Memory Access Conversion

have different clocks. For each clock there is only a pulse generated if that specific memory is activated.

6.4 Design Implementation

To obtain a gate-level netlist, Synopsys Design Compiler[4] is used to map the generic cells of the register-transfer level netlist to cells of the UMC's 90 nm technology library. For the mapping process the UMC's 90 nm technology standard cell library is extended with cells specific for asynchronous circuits, like *Muller C-Gate*s. To successfully translate the generic cells of the `balsa-tech-example` technology, they first have to be modified to match the cells of the adapted standard cell library. Primitives have to be replaced by descriptions consisting of cells available in the library. In particular the affected cells are the `Mutex` cell and different variations of the *Muller C-Gate*. Note that floorplanning and place and route of the resulting netlist is not performed. Thus, the interconnections have zero delay and only the cell delay is taken into account in the timing analysis. Nevertheless, the area and timing results, presented in chapter 7, can be considered meaningful enough for rough area and timing estimates for the evaluation of ASCARTS and the comparison with its synchronous version.

[4] http://www.synospys.com

6.5 Timing Simulation

Timing simulation verifies the operation of the design after the implementation process of the design is accomplished. The timing simulation is done with ModelSim. To reveal any timing violations in the design like race conditions or set-and-hold violations, the netlist used for this simulation is back-annotated with timing information for gate delays provided in an SDF file[5] generated during the implementation process.

For the simulation in ModelSim to work correctly a forced reset has to be done for all `DRLatch` cells to initialize to a defined state.

6.5.1 Memory Delays

As no real memory blocks are synthesized for the targeted UMC's 90 nm technology a memory access delay of 1 ns is assumed. This is easily feasible within 90 nm technology as there was already a static RAM developed with 320 ps memory access time for that technology [1]. Simulation shows, that the fixed delay of the serially linked buffers, which delays the output of the memory, is 24 ps and each buffer has an additional delay of 40 ps. Hence, 25 linked buffers have a total access delay of 1024 ps, which satisfies the requirement.

[5] The **S**tandard **D**elay **F**ormat file contains the cells and interconnections delays calculated during the place and route process as well as timing constraints and technology parameters.

CHAPTER 7

Results

ASCARTS was realized in UMC's 90 nm standard cell library with additional *Muller C* cells. The asynchronous handshake protocol used for the measurements is the four-phase dual-rail protocol. As layouting such a complex circuit is out of scope of this thesis this step was not conducted for the asynchronous processor. A pre-layout design is only an inaccurate model of an integrated circuit as real interconnection numbers are only available after layout. Hence, the final post-layout area and performance results can differ quite significantly from the numbers presented here. Nevertheless, for sake of comparing different circuit designs pre-layout results can still provide meaningful insights.

The cell area and the MIPS performance of ASCARTS were measured and compared with the synchronous version. SCARTS was realized in the same cell library. However, it provides exception handling functionality in addition. The power consumption measurements were omitted as they are only realistic for post-layout designs.

7.1 Resource Usage

Analyzing ASICs one relevant circuit characteristic is the area allocated by the processor design. Synopsys is used to measure the cell area of ASCARTS as well as of SCARTS. As for both processors only pre-layout designs exist, the interconnect area cannot be measured. Therefore, the total area is not defined. The measurements for ASCARTS contain the transformation of the memory signals, however, the actual memory is not included. In Table 7.1 the cell area of ASCARTS and its subcomponents is listed. The total cell area of ASCARTS also includes buffers and logic to link the subcomponents. For comparison the area of SCARTS is added to the table as well. The size of the area is given in number of two-input drive-strength-one `NAND` gates. As can be seen from the table SCARTS is approximately half the size of the asynchronous processor implementation, even though exception handling was implemented. The reason for that is clear. The dual-rail protocol needs twice the number of rails for each signal, which induces a higher area overhead than was saved by omitting the clock network. In addition ASCARTS' Balsa

description has not yet been optimized for area. Neither does `balsa-netlist` optimize when generating a Verilog netlist out of the Breeze description.

Instruction Fetch	1647
Instruction Decode	22 668
Execution	42 323
Memory	5565
Processor Control	11 062
ASCARTS	94 344
SCARTS	43 728

Table 7.1: Cell Area in number of `NAND` Gates

7.2 Performance

As ASCARTS is based on the instruction set of SCARTS no separate timing analysis of the instruction set itself has to be conducted. For measuring the performance of ASCARTS the library `fsd0ai_a_generic_core_tt1v25c.db` was used for synthesis. This is a library for *Average-Case Corner* synthesis with typical parameters, i.e. typical transistors, 1 V supply voltage and 25 °C temperature. As delay-insensitive designs adapt to the actual prevailing operating conditions average performance is more significant. For SCARTS, however, the clock has to be adjusted for correct operation even in worst-case conditions. Hence, the library `fsd0a_a_generic_core_ss0p9v125c.db` was used. This library is for *Worst-Case Corner* synthesis, i.e. with slow transistors, low supply voltage of 0.9 V and high temperature of 125 °C. For ASCARTS timing simulations of the netlist with the SDF file, generated by Synopsys, were conducted by ModelSim. For SCARTS the performance was calculated with the frequency obtained during synthesis and the executed instructions obtained by the ASCARTS simulations. As mentioned before, only approximate interconnection delays are available for the timing analyses of each of the processors.

Due to the asynchronous nature of ASCARTS the execution time of the individual instructions varies. In Figure 7.1 part of a ModelSim simulation is shown. It can be seen clearly that the individual instructions have different execution times.

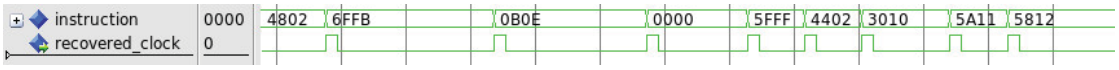


Figure 7.1: Variable Instruction Execution Times

In Table 7.2 different instructions with their average throughput rates are listed. The given rates are calculated assuming that the preceding instructions utilize the individual stages for the same amount of time and there are not any dependencies. For other compositions these rates are not

significant. Depending on the preceding instructions and the stages utilized the execution times can differ substantially from the given rates. For example if one instruction is a memory access, whereas the succeeding instruction does an ALU operation, the throughput rate is higher. As the memory stage and the execution stage can operate simultaneously the throughput rate is limited by the fetch and decode stages. For each group of instructions one representative, which is executed unconditionally, is listed in the table. Simulations showed that the execution time does not depend on the type of operands, however, on the type of memory accessed. Conditional instructions, when executed, are on average 1.71 ns slower than their unconditional counterparts. Not executed conditional instructions have a throughput rate of 73.37 MIPS. The throughput rates range from less than 40 MIPS to up to 126.81 MIPS for `nop`. With the difference of more than the factor 3 the potential of asynchronous processors is shown.

`nop`		126.81 MIPS
`add`		48.19 MIPS
`and`		67.02 MIPS
`mov`		72.46 MIPS
`sr`		38.76 MIPS
`not`		67.75 MIPS
`cmp_eq`		49.09 MIPS
`ldi`		72.25 MIPS
`ldw`	DRAM	57.66 MIPS
	extension module	46.60 MIPS
`stw`	DRAM	68.71 MIPS
	extension module	62.29 MIPS

Table 7.2: Average Instruction Throughput Rates

The execution time of an instruction also heavily depends on the values of the operands. As an example, compare Table 7.3 for the throughput rates of `add` and the corresponding operand pairs. As can be seen, the execution time mainly correlates with the number of bits that differ between the two operands and only marginal with the total number of `1`s.

For obtaining the average performance of ASCARTS different programs were executed. Two programs calculate the factorial of 12, where one was implemented using recursive function calls. The emphasis of these programs lies on arithmetic operations. Quicksort was implemented as a sorting algorithm of 64 values with numerous memory operations and rather few branches. The primality test checks if 412 is a prime number. The emphasis of this program lies on comparison operations. As no other logic besides the comparisons is performed, the program consists of relatively many branches. The benchmark programs are listed in Appendix A.

On the basis of the execution time and the number of fetched and executed instructions, respectively, the average processor frequency as well as the average number of executed instructions per second are calculated for each test program. Table 7.4 compares the average performance of ASCARTS and SCARTS for the different programs. The first column for each processor indicates

Operand 1	Operand 2	
0x00000000	0x00000000	56.65 MIPS
0x000000ff	0x00000000	52.42 MIPS
0x0000ffff	0x00000000	48.38 MIPS
0x00ffffff	0x00000000	44.76 MIPS
0xffffffff	0x00000000	42.14 MIPS
0x00000000	0xffffffff	41.86 MIPS
0xffffffff	0xffffffff	46.66 MIPS
0xff00ff00	0x00ff00ff	41.99 MIPS
0xff00ff00	0xff00ff00	53.70 MIPS

Table 7.3: Different Instruction Throughput Rates For Add

the frequency at which instructions are fetched. The second column specifies the average number of executed instructions per second. SCARTS data were calculated assuming that memory access takes one clock cycle, which corresponds to about 3401 ps. By comparison, ASCARTS memory access time is 1024 ps, see Section 6.5.

In case of ASCARTS the frequency numbers are not related to a periodic clock or to other physical switching activities. They refer to *average* instruction fetch frequencies, which were computed for analyzing the execution of the benchmark programs and for comparing to SCARTS. For the synchronous processor version the instruction fetch frequency is always identical to the processor's clock frequency as it fetches one instruction per clock cycle.

	ASCARTS		SCARTS	
	Fetched Instr.	Executed Instr.	Fetched Instr.	Executed Instr.
factorial	64.55 MHz	44.17 MIPS	294.00 MHz	226.83 MIPS
factorial recursive	63.67 MHz	43.91 MIPS	294.00 MHz	226.15 MIPS
quicksort	62.52 MHz	47.60 MIPS	294.00 MHz	243.18 MIPS
primality test	66.69 MHz	40.94 MIPS	294.00 MHz	207.14 MIPS

Table 7.4: Average Performance

Analyzing the table anomalies are conspicuous. First, running the different programs on ASCARTS result in different processor frequencies. This is the characteristic of asynchronous processors. As different instructions are processed at different speeds, depending on the instructions executed in a program the frequency varies.

Besides, the relation between frequency and executed instructions differ between the asynchronous and the synchronous processor. This is explained by the different number of prefetched instructions in the pipeline, which have to be discarded after a taken branch. Depending on the timing, ASCARTS fetches two to three additional instructions after a branch instruction before

loading the branch address. However, the number of prefetched instructions does not comprise `nop` instructions. These are immediately discarded and, hence, do not occupy the pipeline. Therefore, instructions are prefetched as long as the pipeline provides space. This results in an undefined number of prefetched instructions in ASCARTS, if including `nop` instructions. Due to the synchronous design SCARTS fetches exactly two instructions before switching to the branch address.

Furthermore, the number of executed instructions varies over the different programs. This anomaly, however, is processor-independent. The executed programs vary in the relative number of taken branches. *primality test* has noticeable many branches and, hence, a lower MIPS rate, whereas *quicksort* has noticeable few branches and, hence, a higher MIPS rate.

CHAPTER 8

Conclusion and Outlook

The objective of this thesis was the development of an asynchronous processor with Balsa, an academic hardware description language, which allows modeling asynchronous circuits at an abstract level and removes the burden to specify the implementation details of handshake circuits. Furthermore, it was part of the thesis to evaluate the usability of the Balsa language as well as the associated toolchain for a complex circuit design. As described in the previous chapters, it was indeed possible to successfully design an entire asynchronous processor using the Balsa framework for specification, behavioral simulation and asynchronous synthesis. Only for technology mapping to produce a gate-level netlist and for timing simulations the commercial tools Synopsys Design Compiler and Mentor ModelSim, respectively, had to be used.

Nevertheless, it has become evident during the project that Balsa has some weaknesses and, being an academic tool, obviously cannot offer the same degree of maturity like commercial EDA tools. The language itself is not fully developed and the Balsa synthesis framework is no longer under active development. Even though the Balsa simulator is sufficiently mature for small designs, while implementing ASCARTS its limits were revealed. Error messages are missing or are not always meaningful as they mostly refer to internal names. Moreover, possible race conditions are not always detected.

Developing asynchronous circuits in general demands for alternative development paradigms. It requires a different mindset as the focus has to be on explicit synchronization of processes via handshake signals. Asynchronous development is more complex than the synchronous one. One first has to get used to that development process, especially if already familiar with synchronous development. Besides, in-depth knowledge is advantageous.

As asynchronous processors use handshakes as synchronization, rather than a clock network, some architectural differences between synchronous and asynchronous processors had to be considered. Hazards require different solutions. Instruction stream coloring is used to handle control hazards. For data hazards a dynamic history table along with certain control signals is utilized. Signals between pipeline stages do not have constant delay, therefore, special synchronization signals are

needed to preserve a valid state. In addition, arbiters are needed where it is not predictable which signal arrives first.

Regarding real-time capability worst-case execution time has to be calculated with greater expense as asynchronous designs induce non-determinism. Instruction execution time is not constant. Also the number of fetched instructions varies depending on the timing. However, with more effort upper bounds could be found. With a worst-case corner analysis the worst-case execution times for instructions can be determined. The maximum number of prefetched instructions can be analyzed as well.

Despite the remarkable results, the current processor still lacks of important features. To be entirely compatible with SCARTS, exception handling has yet to be implemented, as well as processor relevant extension modules have to be completed. Furthermore, the *GNU Toolchain* needs to be adapted to the improved instruction set. There is also still enough potential for optimizations. For example control signals can be combined to reduce the number of handshakes, which in turn reduces the area on the chip. To assure real-time capability, a more accurate analysis still has to be performed on ASCARTS.

APPENDIX A

Test Programs

```
int main(int argc, char *argv[]) {
  int i;
  int res = 1;

  for (i = 2; i <= 12; i++) {
    res *= i;
  }

  return res;
}
```

Listing A.1: Factorial

```
int factorial(int f) {
  int res = 1;

  if (f > 2) {
    res = factorial(f - 1);
  }
  return res * f;
}

int main(int argc, char *argv[]) {
  return factorial(12);
}
```

Listing A.2: Factorial Recursive

```
#define MAXSTACK 64

static void exchange(int *a, int *b) {
  int t = *a;
  *a = *b;
  *b = t;
}

void quicksort(int *base, int nmemb) {
  void *lbStack[MAXSTACK], *ubStack[MAXSTACK];
  int sp;
  unsigned int offset;

  /*********************
   *  ANSI-C qsort()   *
   *********************/

  lbStack[0] = base;
  ubStack[0] = base + nmemb - 1;

  for (sp = 0; sp >= 0; sp--) {
    int *lb, *ub, *m;
    int *p, *i, *j;

    lb = lbStack[sp];
    ub = ubStack[sp];

    while (lb < ub) {

      /* select pivot and exchange with 1st element */
      offset = (ub - lb) >> 1;
      p = lb + offset;
      exchange (lb, p);

      /* partition into two segments */
      i = lb + 1;
      j = ub;
      while (1) {
        while (i < j && (*lb - *i) > 0) i += 1;
        while (j >= i && (*j - *lb) > 0) j -= 1;
        if (i >= j) break;
        exchange (i, j);
        j -= 1;
        i += 1;
      }

      /* pivot belongs in A[j] */
      exchange (lb, j);
      m = j;

      /* keep processing smallest segment, and stack largest */
      if (m - lb <= ub - m) {
        if (m + 1 < ub) {
```

```
          lbStack[sp] = m + 1;
          ubStack[sp++] = ub;
        }
        ub = m - 1;
      }
      else {
        if (m - 1 > lb) {
          lbStack[sp] = lb;
          ubStack[sp++] = m - 1;
        }
        lb = m + 1;
      }
    }
  }
}

int main(int argc, char *argv[]) {
  int maxnum;
  int *lb, *ub;
  //int a[] = { 1804289383, 846930886, 1681692777, 1714636915, 1957747793,
      424238335, 719885386, 1649760492, 596516649, 1189641421, 1025202362,
      1350490027, 783368690, 1102520059, 2044897763, 1967513926, 1365180540,
      1540383426, 304089172, 1303455736, 35005211, 521595368, 294702567,
      1726956429, 336465782, 861021530, 278722862, 233665123, 2145174067,
      468703135, 1101513929, 1801979802, 1315634022, 635723058, 1369133069,
      1125898167, 1059961393, 2089018456, 628175011, 1656478042, 1131176229,
      1653377373, 859484421, 1914544919, 608413784, 756898537, 1734575198,
      1973594324, 149798315, 2038664370, 1129566413, 184803526, 412776091,
      1424268980, 1911759956, 749241873, 137806862, 42999170, 982906996,
      135497281, 511702305, 2084420925, 1937477084, 1827336327 };
  int *a;
  a = 0x0000;

  maxnum = 64;
  lb = a;  ub = a + maxnum - 1;

  quicksort(a, maxnum);
  return 0;
}
```

Listing A.3: Quicksort

```
int main(int argc, char *argv[]) {
  int prime = 421;
  int i;

  for (i = 2; i <= prime / 2; i++) {
    if (prime % i == 0) {
      return 0;
    }
  }

  return 1;
}
```

Listing A.4: Primality Test

APPENDIX B

Instruction Set

The notation used to describe the instructions as well as the description itself is taken from [21].

B.1 Overview

nop	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
add	0	0	0	1	0	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
add_cf	0	0	1	0	0	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
add_ct	0	0	1	1	0	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
ldw	0	0	0	0	0	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
addc	0	0	0	1	0	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
addc_cf	0	0	1	0	0	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
addc_ct	0	0	1	1	0	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
stw	0	0	0	0	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
sub	0	0	0	1	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
sub_cf	0	0	1	0	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
sub_ct	0	0	1	1	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
cmp_eq	0	0	0	0	0	0	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
subc	0	0	0	1	0	0	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
subc_cf	0	0	1	0	0	0	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
subc_ct	0	0	1	1	0	0	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
cmp_lt	0	0	0	0	0	1	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
and	0	0	0	1	0	1	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
and_cf	0	0	1	0	0	1	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
and_ct	0	0	1	1	0	1	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
cmpu_lt	0	0	0	0	0	1	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
eor	0	0	0	1	0	1	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
eor_cf	0	0	1	0	0	1	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
eor_ct	0	0	1	1	0	1	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
cmp_gt	0	0	0	0	0	1	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
or	0	0	0	1	0	1	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
or_cf	0	0	1	0	0	1	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
or_ct	0	0	1	1	0	1	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
cmpu_gt	0	0	0	0	0	1	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
mov	0	0	0	1	0	1	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
mov_cf	0	0	1	0	0	1	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
mov_ct	0	0	1	1	0	1	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0

ldh	0	0	0	0	1	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
sr	0	0	0	1	1	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
sr_cf	0	0	1	0	1	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
sr_ct	0	0	1	1	1	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
ldhu	0	0	0	0	1	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
sra	0	0	0	1	1	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
sra_cf	0	0	1	0	1	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
sra_ct	0	0	1	1	1	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
sth	0	0	0	0	1	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
sl	0	0	0	1	1	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
sl_cf	0	0	1	0	1	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
sl_ct	0	0	1	1	1	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
rts	0	0	0	0	1	0	1	1	0	0	0	0	1	1	1	0
rrc	0	0	0	1	1	0	1	1	0	0	0	0	X_3	X_2	X_1	X_0
rrc_cf	0	0	1	0	1	0	1	1	0	0	0	0	X_3	X_2	X_1	X_0
rrc_ct	0	0	1	1	1	0	1	1	0	0	0	0	X_3	X_2	X_1	X_0
ldb	0	0	0	0	1	1	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
not	0	0	0	1	1	1	0	0	0	0	0	0	X_3	X_2	X_1	X_0
not_cf	0	0	1	0	1	1	0	0	0	0	0	0	X_3	X_2	X_1	X_0
not_ct	0	0	1	1	1	1	0	0	0	0	0	0	X_3	X_2	X_1	X_0
ldbu	0	0	0	0	1	1	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
neg	0	0	0	1	1	1	0	1	0	0	0	0	X_3	X_2	X_1	X_0
neg_cf	0	0	1	0	1	1	0	1	0	0	0	0	X_3	X_2	X_1	X_0
neg_ct	0	0	1	1	1	1	0	1	0	0	0	0	X_3	X_2	X_1	X_0
stb	0	0	0	0	1	1	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
jsr	0	0	0	1	1	1	1	0	0	0	0	0	X_3	X_2	X_1	X_0
jsr_cf	0	0	1	0	1	1	1	0	0	0	0	0	X_3	X_2	X_1	X_0
jsr_ct	0	0	1	1	1	1	1	0	0	0	0	0	X_3	X_2	X_1	X_0
rte	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
jmp	0	0	0	1	1	1	1	1	0	0	0	0	X_3	X_2	X_1	X_0
jmp_cf	0	0	1	0	1	1	1	1	0	0	0	0	X_3	X_2	X_1	X_0
jmp_ct	0	0	1	1	1	1	1	1	0	0	0	0	X_3	X_2	X_1	X_0

ldvec	0	1	0	0	0	0	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
stvec	0	1	0	0	0	0	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
addi	0	1	0	1	0	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
addi_cf	0	1	1	0	0	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
addi_ct	0	1	1	1	0	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
btest	0	1	0	0	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
bclr	0	1	0	1	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
bclr_cf	0	1	1	0	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
bclr_ct	0	1	1	1	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
bset	0	1	0	1	0	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
bset_cf	0	1	1	0	0	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
bset_ct	0	1	1	1	0	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
sri	0	1	0	1	1	0	0	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
sri_cf	0	1	1	0	1	0	0	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
sri_ct	0	1	1	1	1	0	0	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
srai	0	1	0	1	1	0	0	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
srai_cf	0	1	1	0	1	0	0	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
srai_ct	0	1	1	1	1	0	0	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
sli	0	1	0	1	1	0	1	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
sli_cf	0	1	1	0	1	0	1	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
sli_ct	0	1	1	1	1	0	1	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
trap	0	1	0	1	1	0	1	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
trap_cf	0	1	1	0	1	0	1	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
trap_ct	0	1	1	1	1	0	1	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
jmpi	0	1	0	1	1	1	I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
jmpi_cf	0	1	1	0	1	1	I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
jmpi_ct	0	1	1	1	1	1	I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
cmpi_eq	0	1	0	0	1	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
cmpi_lt	1	0	1	1	0	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
cmpi_gt	1	0	1	1	1	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0

ldi	1	0	0	0	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
ldhi	1	0	0	1	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
ldli	1	0	1	0	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
ldfpw_inc	1	1	0	0	0	0	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
ldfpw_dec	1	1	0	0	0	0	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
ldfpx_inc	1	1	0	0	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
ldfpx_dec	1	1	0	0	0	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
ldfpy_inc	1	1	0	0	1	0	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
ldfpy_dec	1	1	0	0	1	0	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
ldfpz_inc	1	1	0	0	1	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
ldfpz_dec	1	1	0	0	1	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
stfpw_inc	1	1	0	1	0	0	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
stfpw_dec	1	1	0	1	0	0	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
stfpx_inc	1	1	0	1	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
stfpx_dec	1	1	0	1	0	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
stfpy_inc	1	1	0	1	1	0	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
stfpy_dec	1	1	0	1	1	0	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
stfpz_inc	1	1	0	1	1	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
stfpz_dec	1	1	0	1	1	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
ldfpw	1	1	1	0	0	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
ldfpx	1	1	1	0	0	1	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
ldfpy	1	1	1	0	1	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
ldfpz	1	1	1	0	1	1	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
stfpw	1	1	1	1	0	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
stfpx	1	1	1	1	0	1	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
stfpy	1	1	1	1	1	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
stfpz	1	1	1	1	1	1	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0

B.2 Notation

$+$	Addition operator
$-$	Subtraction operator
$>$	Greater than operator
$<$	Less than operator
$=$	Equality operator
$:=$	Assignment operator
|	Bitwise logical or operator
&	Bitwise logical and operator
^	Bitwise exclusive or operator
~	Bitwise complement operator
«	Bitwise shift left operator
»	Bitwise shift right operator
(int)	Casts the operand to a sign-extended integer of natural width
(int$X$)	Casts the operand to a sign-extended integer of X bits in width
(uint)	Casts the operand to a non sign-extended integer of natural width
(uint$X$)	Casts the operand to a non sign-extended integer of X bits in width
cond, carry	The cond-flag and the carry-flag, respectively
rX, rY	A general purpose register operand
simm$X$	A signed immediate operand of X bits in width
uimm$X$	An unsigned immediate operand of X bits in width
mem(α)	A memory access of address α of natural width
mem$_X$(α)	A memory access of address α of X bits in width

B.3 Instructions

B.3.1 Arithmetic/Logical Instructions

add – Add

Opc								Reg				Reg			
0	0	0	1	0	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

add rX, rY	carry := ((int64) rX + (int64) rY) » 32
	rX := rX + rY

add_cf – Add if cond-flag false

Opc								Reg				Reg			
0	0	1	0	0	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`add_cf rX, rY`

```
if (!cond)
  carry := ((int64) rX + (int64) rY) » 32
  rX := rX + rY
```

add_ct – Add if cond-flag true

Opc								Reg				Reg			
0	0	1	1	0	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`add_ct rX, rY`

```
if (cond)
  carry := ((int64) rX + (int64) rY) » 32
  rX := rX + rY
```

addi – Add immediate

Opc						SImm						Reg			
0	1	0	1	0	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`addi rX, simm6`

```
carry := ((int64) rX + (int64) simm6) » 32
rX := rX + (int) simm6
```

addi_cf – Add immediate if cond-flag false

Opc						SImm						Reg			
0	1	1	0	0	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`addi_cf rX, simm6`

```
if (!cond)
  carry := ((int64) rX + (int64) simm6) » 32
  rX := rX + (int) simm6
```

addi_ct – Add immediate if cond-flag true

Opc						SImm						Reg			
0	1	1	1	0	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`addi_ct rX, simm6`	`if (cond)`
	`  carry := ((int64) rX + (int64) simm6) » 32`
	`  rX := rX + (int) simm6`

addc – Add with carry

Opc								Reg				Reg			
0	0	0	1	0	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`addc rX, rY`	`carry′ := ((int64) rX + (int64) rY) » 32`
	`rX := rX + rY + carry`
	`carry := carry′`

addc_cf – Add with carry if cond-flag false

Opc								Reg				Reg			
0	0	1	0	0	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`addc_cf rX, rY`	`if (!cond)`
	`  carry′ := ((int64) rX + (int64) rY) » 32`
	`  rX := rX + rY + carry`
	`  carry := carry′`

addc_ct – Add with carry if cond-flag true

Opc								Reg				Reg			
0	0	1	1	0	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`addc_ct rX, rY`	`if (cond)`
	`  carry′ := ((int64) rX + (int64) rY) » 32`
	`  rX := rX + rY + carry`
	`  carry := carry′`

and – Bitwise logical and

Opc								Reg				Reg			
0	0	0	1	0	1	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`and rX, rY`	`rX` := `rX` & `rY`

and_cf – Bitwise logical and if cond-flag false

Opc								Reg				Reg			
0	0	1	0	0	1	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`and_cf rX, rY`	`if (!cond)` `rX` := `rX` & `rY`

and_ct – Bitwise logical and if cond-flag true

Opc								Reg				Reg			
0	0	1	1	0	1	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`and_ct rX, rY`	`if (cond)` `rX` := `rX` & `rY`

bclr – Bit clear

Opc							UImm					Reg			
0	1	0	1	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`bclr rX, uimm5`	`rX` := `rX` & ~(1 « `uimm5`)

bclr_cf – Bit clear if cond-flag false

Opc							UImm					Reg			
0	1	1	0	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`bclr_cf rX, uimm5`	`if (!cond)` `rX` := `rX` & ~(1 « `uimm5`)

bclr_ct – Bit clear if cond-flag true

Opc							UImm					Reg			
0	1	1	1	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

```
bclr_ct rX, uimm5        if (cond)
                           rX := rX & ~(1 « uimm5)
```

bset – Bit set

Opc							UImm					Reg			
0	1	0	1	0	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

```
bset rX, uimm5           rX := rX | (1 « uimm5)
```

bset_cf – Bit set if cond-flag false

Opc							UImm					Reg			
0	1	1	0	0	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

```
bset_cf rX, uimm5        if (!cond)
                           rX := rX | (1 « uimm5)
```

bset_ct – Bit set if cond-flag true

Opc							UImm					Reg			
0	1	1	1	0	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

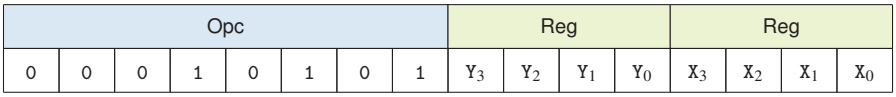
```
bset_ct rX, uimm5        if (cond)
                           rX := rX | (1 « uimm5)
```

btest – Bit test

Opc							UImm					Reg			
0	1	0	0	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

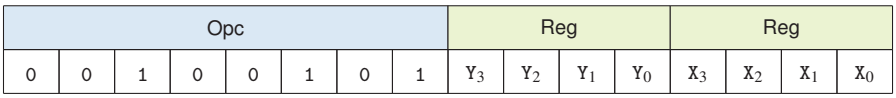
```
btest rX, uimm5          cond := (rX & (1 « uimm5)) ≠ 0
```

eor – Bitwise logical exclusive or



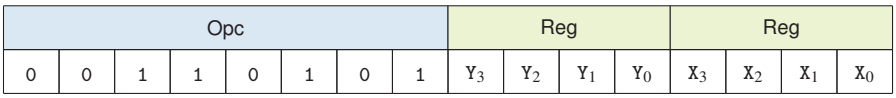
`eor rX, rY`	`rX := rX ^ rY`

eor_cf – Bitwise logical exclusive or if cond-flag false



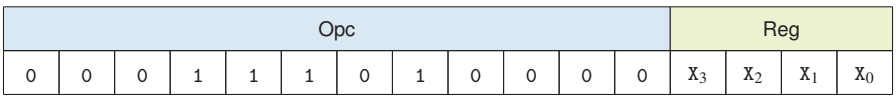
`eor_cf rX, rY`	`if (!cond)` `  rX := rX ^ rY`

eor_ct – Bitwise logical exclusive or if cond-flag true



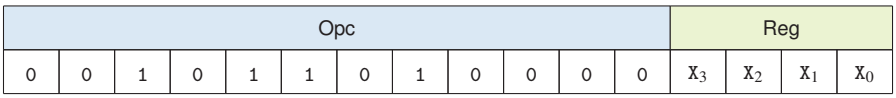
`eor_ct rX, rY`	`if (cond)` `  rX := rX ^ rY`

neg – Negative



`neg rX`	`rX := ~rX + 1`

neg_cf – Negative if cond-flag false



`neg_cf rX`	`if (!cond)` `  rX := ~rX + 1`

neg_ct – Negative if cond-flag true

Opc												Reg			
0	0	1	1	1	1	0	1	0	0	0	0	X_3	X_2	X_1	X_0
15							8	7							0

`neg_ct rX`	`if (cond)` `rX := ~rX + 1`

not – Bitwise logical not

Opc												Reg			
0	0	0	1	1	1	0	0	0	0	0	0	X_3	X_2	X_1	X_0
15							8	7							0

`not rX`	`rX := ~rX`

not_cf – Bitwise logical not if cond-flag false

Opc												Reg			
0	0	1	0	1	1	0	0	0	0	0	0	X_3	X_2	X_1	X_0
15							8	7							0

`not_cf rX`	`if (!cond)` `rX := ~rX`

not_ct – Bitwise logical not if cond-flag true

Opc												Reg			
0	0	1	1	1	1	0	0	0	0	0	0	X_3	X_2	X_1	X_0
15							8	7							0

`not_ct rX`	`if (cond)` `rX := ~rX`

or – Bitwise logical or

Opc								Reg				Reg			
0	0	0	1	0	1	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`or rX, rY`	`rX := rX | rY`

or_cf – Bitwise logical or if cond-flag false

Opc								Reg				Reg			
0	0	1	0	0	1	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

```
or_cf rX, rY        if (!cond)
                      rX := rX | rY
```

or_ct – Bitwise logical or if cond-flag true

Opc								Reg				Reg			
0	0	1	1	0	1	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

```
or_ct rX, rY        if (cond)
                      rX := rX | rY
```

rrc – Rotate right with carry

Opc												Reg			
0	0	0	1	1	0	1	1	0	0	0	0	X_3	X_2	X_1	X_0
15							8	7							0

```
rrc rX              carry′ := rX & 1
                    rX := rX » 1
                    rX := rX | (carry « 31)
                    carry := carry′
```

rrc_cf – Rotate right with carry if cond-flag false

Opc												Reg			
0	0	1	0	1	0	1	1	0	0	0	0	X_3	X_2	X_1	X_0
15							8	7							0

```
rrc_cf rX           if (!cond)
                      carry′ := rX & 1
                      rX := rX » 1
                      rX := rX | (carry « 31)
                      carry := carry′
```

rrc_ct – Rotate right with carry if cond-flag true

Opc												Reg			
0	0	1	1	1	0	1	1	0	0	0	0	X_3	X_2	X_1	X_0
15							8	7							0

```
rrc_ct rX
    if (cond)
      carry' := rX & 1
      rX := rX » 1
      rX := rX | (carry « 31)
      carry := carry'
```

sl – Shift left

Opc								Reg				Reg			
0	0	0	1	1	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

```
sl rX, rY
    carry := (rX » (32 − (rY & 0x1F))) & 1
    rX := rX « (rY & 0x1F)
```

sl_cf – Shift left if cond-flag false

Opc								Reg				Reg			
0	0	1	0	1	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

```
sl_cf rX, rY
    if (!cond)
      carry := (rX » (32 − (rY & 0x1F))) & 1
      rX := rX « (rY & 0x1F)
```

sl_ct – Shift left if cond-flag true

Opc								Reg				Reg			
0	0	1	1	1	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

```
sl_ct rX, rY
    if (cond)
      carry := (rX » (32 − (rY & 0x1F))) & 1
      rX := rX « (rY & 0x1F)
```

sli – Shift left immediate

Opc								UImm				Reg			
0	1	0	1	1	0	1	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`sli rX, uimm4`	`carry` := (`rX` » (32 − `uimm4`)) & 1
	`rX` := `rX` « `uimm4`

sli_cf – Shift left immediate if cond-flag false

Opc								UImm				Reg			
0	1	1	0	1	0	1	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`sli_cf rX, uimm4`	`if` (!`cond`)
	`carry` := (`rX` » (32 − `uimm4`)) & 1
	`rX` := `rX` « `uimm4`

sli_ct – Shift left immediate if cond-flag true

Opc								UImm				Reg			
0	1	1	1	1	0	1	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`sli_ct rX, uimm4`	`if` (`cond`)
	`carry` := (`rX` » (32 − `uimm4`)) & 1
	`rX` := `rX` « `uimm4`

sr – Shift right

Opc								Reg				Reg			
0	0	0	1	1	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`sr rX, rY`	`carry` := `rX` & (1 « ((`rY` & 0x1F) − 1))
	`rX` := `rX` » (`rY` & 0x1F)

sr_cf – Shift right if cond-flag false

Opc								Reg				Reg			
0	0	1	0	1	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

```
sr_cf rX, rY
    if (!cond)
      carry := rX & (1 « ((rY & 0x1F) − 1))
      rX := rX » (rY & 0x1F)
```

sr_ct – Shift right if cond-flag true

Opc								Reg				Reg			
0	0	1	1	1	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

```
sr_ct rX, rY
    if (cond)
      carry := rX & (1 « ((rY & 0x1F) − 1))
      rX := rX » (rY & 0x1F)
```

sra – Shift right arithmetic

Opc								Reg				Reg			
0	0	0	1	1	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

```
sra rX, rY
    carry := rX & (1 « ((rY & 0x1F) − 1))
    rX := (int) rX » (rY & 0x1F)
```

sra_cf – Shift right arithmetic if cond-flag false

Opc								Reg				Reg			
0	0	1	0	1	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

```
sra_cf rX, rY
    if (!cond)
      carry := rX & (1 « ((rY & 0x1F) − 1))
      rX := (int) rX » (rY & 0x1F)
```

sra_ct – Shift right arithmetic if cond-flag true

Opc								Reg				Reg			
0	0	1	1	1	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`sra_ct rX, rY`

```
if (cond)
  carry := rX & (1 « ((rY & 0x1F) − 1))
  rX := (int) rX » (rY & 0x1F)
```

srai – Shift right arithmetic immediate

Opc								UImm				Reg			
0	1	0	1	1	0	0	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`srai rX, uimm4`

```
carry := rX & (1 « (uimm4 − 1))
rX := (int) rX » uimm4
```

srai_cf – Shift right arithmetic immediate if cond-flag false

Opc								UImm				Reg			
0	1	1	0	1	0	0	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`srai_cf rX, uimm4`

```
if (!cond)
  carry := rX & (1 « (uimm4 − 1))
  rX := (int) rX » uimm4
```

srai_ct – Shift right arithmetic immediate if cond-flag true

Opc								UImm				Reg			
0	1	1	1	1	0	0	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`srai_ct rX, uimm4`

```
if (cond)
  carry := rX & (1 « (uimm4 − 1))
  rX := (int) rX » uimm4
```

sri – Shift right immediate

Opc								UImm				Reg			
0	1	0	1	1	0	0	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

```
sri rX, uimm4
```

```
carry := rX & (1 « (uimm4 − 1))
rX := rX » uimm4
```

sri_cf – Shift right immediate if cond-flag false

Opc								UImm				Reg			
0	1	1	0	1	0	0	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

```
sri_cf rX, uimm4
```

```
if (!cond)
  carry := rX & (1 « (uimm4 − 1))
  rX := rX » uimm4
```

sri_ct – Shift right immediate if cond-flag true

Opc								UImm				Reg			
0	1	1	1	1	0	0	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

```
sri_ct rX, uimm4
```

```
if (cond)
  carry := rX & (1 « (uimm4 − 1))
  rX := rX » uimm4
```

sub – Subtract

Opc								Reg				Reg			
0	0	0	1	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

```
sub rX, rY
```

```
carry := ((int64) rX − (int64) rY) » 32
rX := rX − rY
```

sub_cf – Subtract if cond-flag false

Opc								Reg				Reg			
0	0	1	0	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`sub_cf rX, rY`	`if (!cond)`
	`  carry` $:=$ `((int64) rX` $-$ `(int64) rY)` $\gg$ `32`
	`  rX` $:=$ `rX` $-$ `rY`

sub_ct – Subtract if cond-flag true

Opc								Reg				Reg			
0	0	1	1	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`sub_ct rX, rY`	`if (cond)`
	`  carry` $:=$ `((int64) rX` $-$ `(int64) rY)` $\gg$ `32`
	`  rX` $:=$ `rX` $-$ `rY`

subc – Subtract with carry

Opc								Reg				Reg			
0	0	0	1	0	0	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`subc rX, rY`	`carry'` $:=$ `((int64) rX` $-$ `(int64) rY)` $\gg$ `32`
	`rX` $:=$ `rX` $-$ `rY` $+$ `carry`
	`carry` $:=$ `carry'`

subc_cf – Subtract with carry if cond-flag false

Opc								Reg				Reg			
0	0	1	0	0	0	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`subc_cf rX, rY`	`if (!cond)`
	`  carry'` $:=$ `((int64) rX` $-$ `(int64) rY)` $\gg$ `32`
	`  rX` $:=$ `rX` $-$ `rY` $+$ `carry`
	`  carry` $:=$ `carry'`

subc_ct – Subtract with carry if cond-flag true

Opc								Reg				Reg			
0	0	1	1	0	0	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`subc_ct rX, rY`	`if (cond)`
	`  carry′ := ((int64) rX − (int64) rY) » 32`
	`  rX := rX − rY + carry`
	`  carry := carry′`

B.3.2 Comparison Instructions

cmp_eq – Compare equal

Opc								Reg				Reg			
0	0	0	0	0	0	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`cmp_eq rX, rY`	`cond := (int) rX = (int) rY`

cmp_gt – Compare greater than

Opc								Reg				Reg			
0	0	0	0	0	1	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`cmp_gt rX, rY`	`cond := (int) rX > (int) rY`

cmp_lt – Compare less than

Opc								Reg				Reg			
0	0	0	0	0	1	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`cmp_lt rX, rY`	`cond := (int) rX < (int) rY`

cmpi_eq – Compare equal immediate

Opc					SImm							Reg			
0	1	0	0	1	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`cmpi_eq rX, simm7`	`cond := (int) rX = (int) simm7`

cmpi_gt – Compare greater than immediate

Opc					SImm							Reg			
1	0	1	1	1	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`cmpi_gt rX, simm7`	`cond` $:=$ `(int) rX` $>$ `(int) simm7`

cmpi_lt – Compare less than immediate

Opc					SImm							Reg			
1	0	1	1	0	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`cmpi_lt rX, simm7`	`cond` $:=$ `(int) rX` $<$ `(int) simm7`

cmpu_gt – Compare greater than unsigned

Opc								Reg				Reg			
0	0	0	0	0	1	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`cmpu_gt rX, rY`	`cond` $:=$ `(uint) rX` $>$ `(uint) rY`

cmpu_lt – Compare less than unsigned

Opc								Reg				Reg			
0	0	0	0	0	1	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`cmpu_lt rX, rY`	`cond` $:=$ `(uint) rX` $<$ `(uint) rY`

B.3.3 Constant Manipulating Instructions

ldhi – Load high byte immediate

Opc				SImm								Reg			
1	0	0	1	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldhi rX, simm8`	`rX` $:=$ `rX` $\&$ `0xFF`
	`rX` $:=$ `rX` $\mid$ `((int) simm8` $\ll$ `8)`

ldi – Load immediate

Opc				SImm								Reg			
1	0	0	0	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldi rX, simm8`	`rX` := (`int`) `simm8`

ldli – Load low byte immediate

Opc				UImm								Reg			
1	0	1	0	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldli rX, uimm8`	`rX` := `rX` & `~0xFF` `rX` := `rX` | `uimm8`

B.3.4 Control Transfer Instructions

jmp – Jump

Opc												Reg			
0	0	0	1	1	1	1	1	0	0	0	0	X_3	X_2	X_1	X_0
15							8	7							0

`jmp rX`	`PC` := `rX`

jmp_cf – Jump if cond-flag false

Opc												Reg			
0	0	1	0	1	1	1	1	0	0	0	0	X_3	X_2	X_1	X_0
15							8	7							0

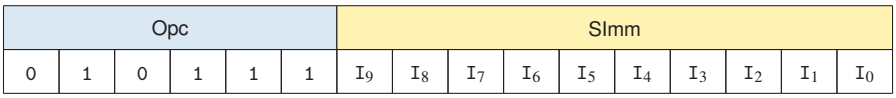
`jmp_cf rX`	`if` (!`cond`) `PC` := `rX`

jmp_ct – Jump if cond-flag true

Opc												Reg			
0	0	1	1	1	1	1	1	0	0	0	0	X_3	X_2	X_1	X_0
15							8	7							0

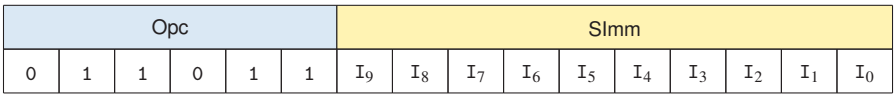
`jmp_ct rX`	`if` (`cond`) `PC` := `rX`

jmpi – Jump immediate



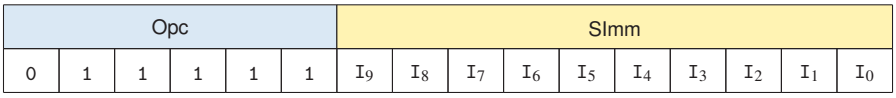
`jmpi simm10`	PC := PC + simm10

jmpi_cf – Jump immediate if cond-flag false



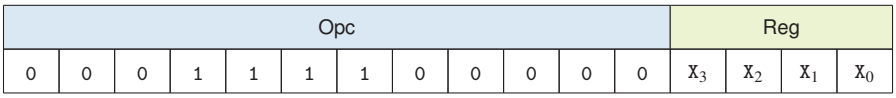
`jmpi_cf simm10`	if (!cond) PC := PC + simm10

jmpi_ct – Jump immediate if cond-flag true



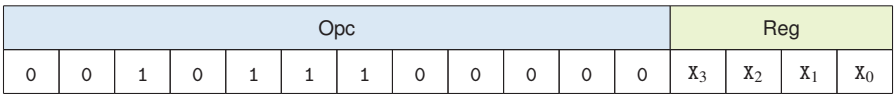
`jmpi_ct simm10`	if (cond) PC := PC + simm10

jsr – Jump to subroutine



`jsr rX`	r14 := PC PC := rX

jsr_cf – Jump to subroutine if cond-flag false



`jsr_cf rX`	if (!cond)
	  r14 := PC
	  PC := rX

jsr_ct – Jump to subroutine if cond-flag true

Opc												Reg			
0	0	1	1	1	1	1	0	0	0	0	0	X_3	X_2	X_1	X_0
15							8	7							0

```
jsr_ct rX          if (cond)
                     r14 := PC
                     PC := rX
```

rts – Return from subroutine

Opc															
0	0	0	0	1	0	1	1	0	0	0	0	1	1	1	0
15							8	7							0

```
rts                PC := r14
```

B.3.5 Data Movement Instructions

mov – Move

Opc								Reg				Reg			
0	0	0	1	0	1	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

```
mov rX, rY         rX := rY
```

mov_cf – Move if cond-flag false

Opc								Reg				Reg			
0	0	1	0	0	1	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

```
mov_cf rX, rY      if (!cond)
                     rX := rY
```

mov_ct – Move if cond-flag true

Opc								Reg				Reg			
0	0	1	1	0	1	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

```
mov_ct rX, rY      if (cond)
                     rX := rY
```

B.3.6 Exception Instructions

ldvec – Load exception vector

Opc							SImm					Reg			
0	1	0	0	0	0	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldvec rX, simm5`	`rX` := `EVT (simm5)`

nop – No operation

Opc															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15							8	7							0

`nop`

rte – Return from exception

Opc															
0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
15							8	7							0

`rte`	`status` := `status`′
	`PC` := `r15`

stvec – Store exception vector

Opc							SImm					Reg			
0	1	0	0	0	0	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`stvec rX, simm5`	`EVT (simm5)` := `rX`

trap – Trap

Opc								Reg				Reg			
0	1	0	1	1	0	1	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`trap rX, uimm4`	`status`′ := `status`
	`r15` := `PC`
	`PC` := `EVT (uimm4)`

trap_cf – Trap if cond-flag false

Opc								Reg				Reg			
0	1	1	0	1	0	1	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

```
trap_cf rX, uimm4        if (!cond)
                           status' := status
                           r15 := PC
                           PC := EVT (uimm4)
```

trap_ct – Trap if cond-flag true

Opc								Reg				Reg			
0	1	1	1	1	0	1	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

```
trap_ct rX, uimm4        if (cond)
                           status' := status
                           r15 := PC
                           PC := EVT (uimm4)
```

B.3.7 Load/Store Instructions

ldb – Load byte

Opc								Reg				Reg			
0	0	0	0	1	1	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

```
ldb rX, rY               rX := (int) mem8 (rY)
```

ldbu – Load byte unsigned

Opc								Reg				Reg			
0	0	0	0	1	1	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

```
ldbu rX, rY              rX := (uint) mem8 (rY)
```

ldfpw – Load word with frame pointer W

Opc						SImm						Reg			
1	1	1	0	0	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldfpw rX, simm6`	`rX` := `mem (FPW + (int) simm6)`

ldfpw_dec – Load word with frame pointer W and decrement W

Opc							SImm						Reg			
1	1	0	0	0	0	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldfpw_dec rX, simm5`	`rX` := `mem (FPW + (int) simm5)`
	`FPW` := `FPW − 4`

ldfpw_inc – Load word with frame pointer W and increment W

Opc							SImm						Reg			
1	1	0	0	0	0	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldfpw_inc rX, simm5`	`rX` := `mem (FPW + (int) simm5)`
	`FPW` := `FPW + 4`

ldfpx – Load word with frame pointer X

Opc						SImm						Reg			
1	1	1	0	0	1	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldfpx rX, simm6`	`rX` := `mem (FPX + (int) simm6)`

ldfpx_dec – Load word with frame pointer X and decrement X

Opc							SImm						Reg			
1	1	0	0	0	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldfpx_dec rX, simm5`	`rX` := `mem (FPX + (int) simm5)`
	`FPX` := `FPX − 4`

ldfpx_inc – Load word with frame pointer X and increment X

Opc							SImm					Reg			
1	1	0	0	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldfpx_inc rX, simm5`	$\texttt{rX} := \texttt{mem}\,(\texttt{FPX} + (\texttt{int})\,\texttt{simm5})$
	$\texttt{FPX} := \texttt{FPX} + 4$

ldfpy – Load word with frame pointer Y

Opc						SImm						Reg			
1	1	1	0	1	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldfpy rX, simm6`	$\texttt{rX} := \texttt{mem}\,(\texttt{FPY} + (\texttt{int})\,\texttt{simm6})$

ldfpy_dec – Load word with frame pointer Y and decrement Y

Opc							SImm					Reg			
1	1	0	0	1	0	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldfpy_dec rX, simm5`	$\texttt{rX} := \texttt{mem}\,(\texttt{FPY} + (\texttt{int})\,\texttt{simm5})$
	$\texttt{FPY} := \texttt{FPY} - 4$

ldfpy_inc – Load word with frame pointer Y and increment Y

Opc							SImm					Reg			
1	1	0	0	1	0	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldfpy_inc rX, simm5`	$\texttt{rX} := \texttt{mem}\,(\texttt{FPY} + (\texttt{int})\,\texttt{simm5})$
	$\texttt{FPY} := \texttt{FPY} + 4$

ldfpz – Load word with frame pointer Z

Opc						SImm						Reg			
1	1	1	0	1	1	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldfpz rX, simm6`	$\texttt{rX} := \texttt{mem}\,(\texttt{FPZ} + (\texttt{int})\,\texttt{simm6})$

ldfpz_dec – Load word with frame pointer Z and decrement Z

Opc							SImm					Reg			
1	1	0	0	1	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldfpz_dec rX, simm5`	$\texttt{rX} := \texttt{mem}\,(\texttt{FPZ} + (\texttt{int})\,\texttt{simm5})$
	$\texttt{FPZ} := \texttt{FPZ} - 4$

ldfpz_inc – Load word with frame pointer Z and increment Z

Opc							SImm					Reg			
1	1	0	0	1	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldfpz_inc rX, simm5`	$\texttt{rX} := \texttt{mem}\,(\texttt{FPZ} + (\texttt{int})\,\texttt{simm5})$
	$\texttt{FPZ} := \texttt{FPZ} + 4$

ldh – Load half-word

Opc								Reg				Reg			
0	0	0	0	1	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldh rX, rY`	$\texttt{rX} := (\texttt{int})\,\texttt{mem}_{16}\,(\texttt{rY})$

ldhu – Load half-word unsigned

Opc								Reg				Reg			
0	0	0	0	1	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldhu rX, rY`	$\texttt{rX} := (\texttt{uint})\,\texttt{mem}_{16}\,(\texttt{rY})$

ldw – Load word

Opc								Reg				Reg			
0	0	0	0	0	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`ldw rX, rY`	$\texttt{rX} := \texttt{mem}\,(\texttt{rY})$

stb – Store byte

Opc								Reg				Reg			
0	0	0	0	1	1	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`stb rX, rY`	$\texttt{mem}_8\,(\texttt{rY}) := \texttt{rX}\,(7:0)$

stfpw – Store word with frame pointer W

Opc						SImm						Reg			
1	1	1	1	0	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`stfpw rX, simm6`	$\texttt{mem}\,(\texttt{FPW} + (\texttt{int})\,\texttt{simm6}) := \texttt{rX}$

stfpw_dec – Store word with frame pointer W and decrement W

Opc							SImm					Reg			
1	1	0	1	0	0	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`stfpw_dec rX, simm5`	$\texttt{mem}\,(\texttt{FPW} + (\texttt{int})\,\texttt{simm5}) := \texttt{rX}$
	$\texttt{FPW} := \texttt{FPW} - 4$

stfpw_inc – Store word with frame pointer W and increment W

Opc							SImm					Reg			
1	1	0	1	0	0	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

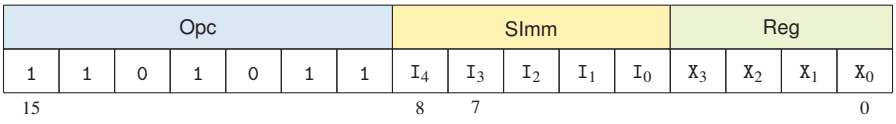
`stfpw_inc rX, simm5`	$\texttt{mem}\,(\texttt{FPW} + (\texttt{int})\,\texttt{simm5}) := \texttt{rX}$
	$\texttt{FPW} := \texttt{FPW} + 4$

stfpx – Store word with frame pointer X

Opc						SImm						Reg			
1	1	1	1	0	1	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

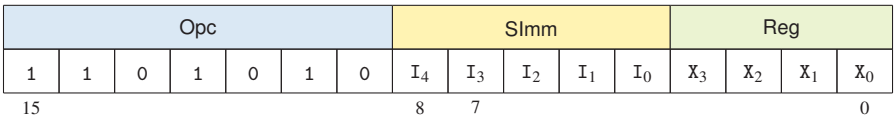
`stfpx rX, simm6`	$\texttt{mem}\,(\texttt{FPX} + (\texttt{int})\,\texttt{simm6}) := \texttt{rX}$

stfpx_dec – Store word with frame pointer X and decrement X



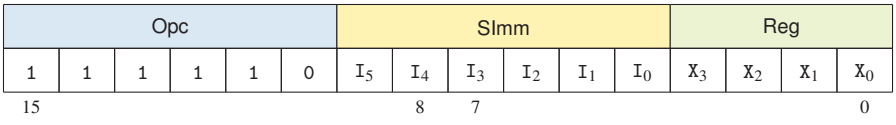
`stfpx_dec rX, simm5`	$\texttt{mem}\,(\texttt{FPX} + (\texttt{int})\,\texttt{simm5}) := \texttt{rX}$ $\texttt{FPX} := \texttt{FPX} - 4$

stfpx_inc – Store word with frame pointer X and increment X



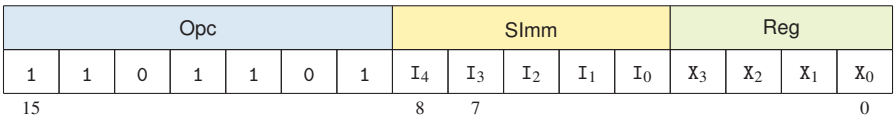
`stfpx_inc rX, simm5`	$\texttt{mem}\,(\texttt{FPX} + (\texttt{int})\,\texttt{simm5}) := \texttt{rX}$ $\texttt{FPX} := \texttt{FPX} + 4$

stfpy – Store word with frame pointer Y



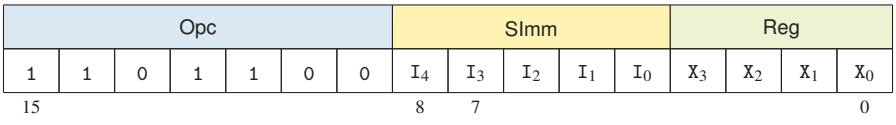
`stfpy rX, simm6`	$\texttt{mem}\,(\texttt{FPY} + (\texttt{int})\,\texttt{simm6}) := \texttt{rX}$

stfpy_dec – Store word with frame pointer Y and decrement Y



`stfpy_dec rX, simm5`	$\texttt{mem}\,(\texttt{FPY} + (\texttt{int})\,\texttt{simm5}) := \texttt{rX}$ $\texttt{FPY} := \texttt{FPY} - 4$

stfpy_inc – Store word with frame pointer Y and increment Y



`stfpy_inc rX, simm5`	$\texttt{mem}\,(\texttt{FPY} + (\texttt{int})\,\texttt{simm5}) := \texttt{rX}$ $\texttt{FPY} := \texttt{FPY} + 4$

stfpz – Store word with frame pointer Z

Opc						SImm						Reg			
1	1	1	1	1	1	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`stfpz rX, simm6`	$\texttt{mem}\,(\texttt{FPZ} + (\texttt{int})\,\texttt{simm6}) := \texttt{rX}$

stfpz_dec – Store word with frame pointer Z and decrement Z

Opc							SImm					Reg			
1	1	0	1	1	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`stfpz_dec rX, simm5`	$\texttt{mem}\,(\texttt{FPZ} + (\texttt{int})\,\texttt{simm5}) := \texttt{rX}$
	$\texttt{FPZ} := \texttt{FPZ} - 4$

stfpz_inc – Store word with frame pointer Z and increment Z

Opc							SImm					Reg			
1	1	0	1	1	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15							8	7							0

`stfpz_inc rX, simm5`	$\texttt{mem}\,(\texttt{FPZ} + (\texttt{int})\,\texttt{simm5}) := \texttt{rX}$
	$\texttt{FPZ} := \texttt{FPZ} + 4$

sth – Store half-word

Opc								Reg				Reg			
0	0	0	0	1	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`sth rX, rY`	$\texttt{mem}_{16}\,(\texttt{rY}) := \texttt{rX}\,(15:0)$

stw – Store word

Opc								Reg				Reg			
0	0	0	0	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15							8	7							0

`stw rX, rY`	$\texttt{mem}\,(\texttt{rY}) := \texttt{rX}$

Bibliography

[1] H. Akiyoshi, H. Shimizu, T. Matsumoto, K. Kobayashi, and Y. Sambonsugi. A 320ps access, 3ghz cycle, 144kb sram macro in 90nm cmos technology using an all-stage reset control signal generator. In *Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International*, pages 460–508 vol.1, Feb 2003.

[2] Mark E. Dean, Ted E. Williams, and David L. Dill. Efficient self-timing with level-encoded 2-phase dual-rail (ledr). In *Proceedings of the 1991 University of California/Santa Cruz Conference on Advanced Research in VLSI*, pages 55–70, Cambridge, MA, USA, 1991. MIT Press.

[3] Martin Delvai. Handbuch für spear (scalable processor for embedded applications in real-time environments). Technical report, E182 - Institut für Technische Informatik; Technische Universität Wien, 2002.

[4] Doug Edwards and Andrew Bardsley. Balsa : A Tutorial Guide. 2006.

[5] Philip Endecott and Stephen Furber. Modelling and simulation of asynchronous systems using the lard hardware description language. In *Proceedings of the 12th European Simulation Multiconference on Simulation - Past, Present and Future*, pages 39–43. SCS Europe, 1998.

[6] K.M. Fant and S.A Brandt. Null convention logic^{TM}: a complete and consistent logic for asynchronous digital circuit synthesis. In *Application Specific Systems, Architectures and Processors, 1996. ASAP 96. Proceedings of International Conference on*, pages 261–273, Aug 1996.

[7] Martin Fletzer. Spear2 - an improved version of spear. Master's thesis, Institut f. Technische Informatik, Embedded Computing Systems Group, 2008.

[8] S.B. Furber, J.D. Garside, P. Riocreux, S. Temple, P. Day, Jianwei Liu, and N.C. Paver. Amulet2e: an asynchronous embedded controller. *Proceedings of the IEEE*, 87(2):243–256, Feb 1999.

[9] J.D. Garside, W.J. Bainbridge, A Bardsley, D.M. Clark, D.A Edwards, S.B. Furber, J. Liu, D.W. Lloyd, S. Mohammadi, J.S. Pepper, O. Petlin, S. Temple, and J.V. Woods. Amulet3i-an asynchronous system-on-chip. In *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proceedings. Sixth International Symposium on*, pages 162–175, 2000.

[10] J.D. Garside, S.B. Furber, and S.-H. Chung. Amulet3 revealed. In *Advanced Research in Asynchronous Circuits and Systems, 1999. Proceedings., Fifth International Symposium on*, pages 51–59, 1999.

[11] S. Hauck. Asynchronous design methodologies: an overview. *Proceedings of the IEEE*, 83(1):69–93, Jan 1995.

[12] Jakob Lechner. Fsl tool. Master's thesis, Institut f. Technische Informatik, Embedded Computing Systems Group, 2008.

[13] AJ. Martin, A Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and Tak Kwan Lee. The design of an asynchronous mips r3000 microprocessor. In *Advanced Research in VLSI, 1997. Proceedings., Seventeenth Conference on*, pages 164–181, Sep 1997.

[14] AJ. Martin, M. Nystrom, K. Papadantonakis, P.I Penzes, P. Prakash, C.G. Wong, J. Chang, K.S. Ko, B. Lee, E. Ou, J. Pugh, E. Talvala, J.T. Tong, and A Tura. The lutonium: a sub-nanojoule asynchronous 8051 microcontroller. In *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*, pages 14–23, May 2003.

[15] A.J. Martin, M. Nystrom, and C.G. Wong. Three generations of asynchronous microprocessors. *Design Test of Computers, IEEE*, 20(6):9–17, Nov 2003.

[16] D.E. Muller and W.S. Bartky. *A Theory of Asynchronous Circuits*. Number Bd. 1 in Report: Digital Computer Laboratory. Univ., 1956.

[17] N.C. Paver, P. Day, S.B. Furber, J.D. Garside, and J.V. Woods. Register locking in an asynchronous microprocessor. In *Computer Design: VLSI in Computers and Processors, 1992. ICCD '92. Proceedings, IEEE 1992 International Conference on*, pages 351–355, Oct 1992.

[18] J. Sparsø. Asynchronous circuit design - a tutorial. In *Chapters 1-8 in "Principles of asynchronous circuit design - A systems Perspective"*, pages 1–152. Kluwer Academic Publishers, Boston / Dordrecht / London, dec 2001.

[19] I. E. Sutherland. Micropipelines. *Commun. ACM*, 32(6):720–738, June 1989.

[20] H. van Gageldonk, K. Van Berkel, A Peeters, D. Baumann, D. Gloor, and G. Stegmann. An asynchronous low-power 80c51 microcontroller. In *Advanced Research in Asynchronous Circuits and Systems, 1998. Proceedings. 1998 Fourth International Symposium on*, pages 96–107, Mar 1998.

[21] Martin Walter. *The SCARTS Hardware/Software Interface*. 2nd ed. OSADL Academic Works, 2011.

[22] J.V. Woods, P. Day, S.B. Furber, J.D. Garside, N.C. Paver, and S. Temple. Amulet1: an asynchronous arm microprocessor. *Computers, IEEE Transactions on*, 46(4):385–398, Apr 1997.

[23] Q. Zhang and G. Theodoropoulos. Modelling samips: a synthesisable asynchronous mips processor. In *Simulation Symposium, 2004. Proceedings. 37th Annual*, pages 205–212, April 2004.