MASTER’S THESIS
The Design of a Micro-Controller
A Transformational Approach
by D. Timmermans

Supervisor: prof. dr. ir. C.H. van Berkel
Advisor: ir. J.L.W. Kessels

Eindhoven, Aug. 2002
Contents

Acknowledgements .................................................................................. 5

Glossary ................................................................................................. 6

1 Introduction ......................................................................................... 7
  1.1 Document Outline ........................................................................ 7
  1.2 Problem Description .................................................................... 7
  1.3 Problem Approach ....................................................................... 8
  1.4 Business Context ......................................................................... 8
  1.5 Tangram Language ....................................................................... 9
  1.6 STARC2 Micro-Controller .......................................................... 11
     1.6.1 Code Memory ....................................................................... 11
     1.6.2 Data Memory ......................................................................... 12
     1.6.3 Addressing Modes ................................................................. 13
     1.6.4 Instruction Set ....................................................................... 14
     1.6.5 STARC1 Differences ............................................................... 17
  1.7 Measurement Conventions ............................................................ 18

2 Initial Design ....................................................................................... 19
  2.1 Conceptual Architecture ............................................................. 19
     2.1.1 Execution Bits ....................................................................... 19
     2.1.2 Phase Distinction .................................................................. 20
     2.1.3 Object Distinction ................................................................ 21
  2.2 Pseudo Code ............................................................................... 22
     2.2.1 STARC2 Procedure ................................................................. 23
     2.2.2 Internal Interrupt Controller Procedure .............................. 23
     2.2.3 Stack Procedures .................................................................. 24
     2.2.4 Instruction Execution Procedure .......................................... 24
     2.2.5 Instruction Fetcher Procedure .............................................. 25
     2.2.6 Instruction Decoder Procedure ............................................ 25
     2.2.7 Data Fetcher Procedures ....................................................... 25
     2.2.8 Data Storage Procedures ..................................................... 27
     2.2.9 Arithmetic Representative Procedure .................................. 29
     2.2.10 Logical Representative Procedure ..................................... 29
     2.2.11 Boolean Manipulation Representative Procedure ............. 29
3 Macho Transformations ................................................................. 32
  3.1 Exploitation of Similarity ......................................................... 32
  3.2 Balance of Selections .............................................................. 33

4 Obtaining Low-Cost .................................................................... 36
  4.1 Narrowing of Control Path ....................................................... 36
  4.2 Unification of Assignments ....................................................... 39
  4.3 Squeezing Exploitation of Similarity ......................................... 40
  4.4 Centralisation of Execution Bit Evaluation ............................... 42
  4.5 Departure of Internal State ....................................................... 43

5 Obtaining High-Performance ...................................................... 45
  5.1 Prefetch Pipeline Stage – Successful ....................................... 45
  5.2 Prefetch Pipeline Stage – Failed .............................................. 47

6 Conclusions ................................................................................. 48
  6.1 Concerning Transformations .................................................... 48
  6.2 Concerning Asynchronous Technology ..................................... 51

7 Future Work ................................................................................ 53

8 References .................................................................................. 54

A Compilation of Initial STARC2 State .......................................... 55

B Proof of Function Chaining Lemma ............................................. 57
Acknowledgements

I would very much like to express my gratitude towards a number of people. Thanks, Kees van Berkel and Joep Kessels for supervising my research. Thanks, Ad Peeters and Marc Verra for giving me tool support. Thanks, my fellow students Rick Nas and Bill Donkervoet as well as all regular coffee club members for engaging into a large amount of off-topic discussions. Finally yet importantly, thanks, my lovely Anita for withstanding me in the hectic finale of this writing.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic and Logic Unit</td>
</tr>
<tr>
<td>BB</td>
<td>Behaviour Bits (indicate <em>how</em> hardware tasks behave and manipulate data on the level of expressions within guards and assignments)</td>
</tr>
<tr>
<td>CB</td>
<td>Control Bits (indicate <em>what</em> hardware tasks should be performed on the level of commands)</td>
</tr>
<tr>
<td>EB</td>
<td>Execution Bits (both control bits and behaviour bits)</td>
</tr>
<tr>
<td>EMI</td>
<td>Electro Magnetic Interference</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>MIPS</td>
<td>Million Instructions Per Second</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>PSW</td>
<td>Program Status Word</td>
</tr>
<tr>
<td>R0, ..., R7</td>
<td>Register zero, ..., Register seven</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RF</td>
<td>Register File</td>
</tr>
<tr>
<td>SP</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>STARC2</td>
<td>Security Transponder and Risc Controller, version 2</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integration</td>
</tr>
</tbody>
</table>
1 Introduction

This master’s thesis describes the results obtained from our research at Philips Research. We have carried out the work for this project from September 17th 2001 until July 12th 2002. Kees van Berkel and Joep Kessels were my supervisors at Eindhoven University of Technology and at Philips Research, respectively.

1.1 Document Outline

The outline of this document is as follows. Section 1.2 describes the goals of my investigations. Section 1.3 explains the transformational approach to the problem. Section 1.5 introduces the reader to the programming language Tangram, which is a high-level language to design asynchronous handshake circuits. Section 1.6 specifies the STARC2 micro-controller, which is used as on-going example to explain transformations on micro-controller designs. Section 1.7 explains what is measured to quantify results and how these measurements are exactly performed. Chapter 2 elaborates on our initial design of the STARC2. It does so by discussing the conceptual architecture in Section 2.1 and presenting pseudo code for this architecture in Section 2.2. Chapter 3 defines so-called macho transformations on the initial STARC2 design that benefits all three of area, power and speed. Chapter 4 presents transformations on the macho STARC2 design to obtain a low-cost design. Chapter 5 presents transformations on the macho STARC2 design to obtain a high-performance design. Chapter 6 presents conclusions from the results of the investigations performed in earlier chapters. Finally, Chapter 7 addresses some topics that are not investigated in this assignment but promise to be interesting anyway.

1.2 Problem Description

The dominant trend in micro-processor and micro-controller design has been high-performance. However, recently branches towards low-power and low-cost designs are emerging. In many portable and consumer applications such as automotive, wireless and household equipment there is no need for much processing power. There, low-cost and/or low-power are far more important than high performance.

At Philips Research, an interest has emerged in how to systematically obtain optimal results in the quantities area, power and speed when designing micro-controllers. Therefore, Philip Research has defined a graduate assignment that consists of three parts:

1. Design an initial STARC2 micro-controller (an introduction to this micro-controller is given in Section 1.6) in the Tangram programming language (an introduction to this programming language is given in Section 1.5). The main emphasis in this design is on readability.

2. Define a sequence of transformations on this micro-controller initial design such that its area (in terms of square millimetres) decreases significantly.
3. Define a sequence of transformations on this micro-controller initial design such that its speed (in terms of millions instructions per second) increases significantly.

The initial design should be as abstract as possible, which means that we only take implementation related design decisions necessary.

We have chosen the above-mentioned micro-controller and programming language for this assignment because the Automotive group at Philips Semiconductors is interested in a fully asynchronous low-cost STARC2 design. However, most of our results are more generally applicable. We want to investigate for which transformations the asynchronous technology is an advantage or disadvantage. A sequence of transformations to obtain low-power in our design would also be interesting, but we did not have the time to investigate this path.

1.3 Problem Approach

According to the problem description we aim at optimal results with respect to area and speed. If we would design a functionally correct STARC2 and carry through optimisations simultaneously, things could get unnecessarily complicated and keep us from obtaining good results. Instead, we have chosen to separate our concerns of functional correctness and optimised quantities. This is achieved by means of designing transformationally. First, we create an intuitive, highly readable, correct, initial design. Second, we define a number of transformations that successively obtain better results with respect to area, power and speed. Third, we define a number of transformations that successively obtain better results with respect to area at the penalty of power and/or speed. Fourth, we define a number of transformations that successively obtain better results with respect to speed at the penalty of area and/or power. A major advantage to this approach is that these transformations can also be applied to other micro-controller designs.

These transformations are performed on the Tangram program code and produce new Tangram program code that results in better quantities after compilation and simulation. The order of these transformations is of importance for two reasons. First, some of the transformations have requirements to the program code which is inserted in previous transformations. This is the case for the second, third and fifth low-cost transformations. In addition, the measurements and their documented effectiveness (in terms of differences in quantities) are not correct when the transformations are performed on different program code.

1.4 Business Context

As stated in the problem description, the Automotive group at Philips Semiconductors is interested in a fully asynchronous low-cost STARC2 design. Therefore, our research is directed to some extent by the requirements of this customer. This implied that every STARC2 design (that is the result of a transformation as well as the initial design) must be functionally correct, complete and exceed a certain programming quality.
The correctness requirement implies that a large amount of simulation and debugging iterations had to be carried out. An acceptance requirement involved a simulation of the STARC2 design running a STARC2 test program that successfully computes a certain checksum. This test program is designed to detect errors in the STARC2 design. Unfortunately, certain bugs in the tools that are necessary for compilation and simulation have taken up many weeks of spoiled time (for a long period of time even the simplest assignments did to function properly). Fortunately, all Tangram-related bugs are fixed now.

The completeness requirement also involved quite some work, partly because even the smallest programming details had to be dealt with and partly because the STARC2 specification documentation mainly consists of informal text. Refer to [5] for this specification.

The quality requirement implies that the Tangram program code easily allows for modifications and extensions. The customer expects to produce millions of chips for multiple products using these designs in the near future. Therefore, these designs need to be easily customisable to meet specific product-requirements.

***

Initially, our customer was interested in a fully asynchronous low-cost STARC1 design. After completion of an initial, functionally correct, design as well as a nearly finished low-cost derivative thereof, the customer decided to upgrade their needs from a STARC1 micro-controller to a STARC2 micro-controller. Since they wanted to have this (at the time still to be built) low-cost STARC2 design as soon as possible, we have chosen to base the low-cost STARC2 on the present low-cost STARC1 design with the required adjustments for it. When this was finished, we continued our investigations for a high-performance design by taking this low-cost STARC2 design as its base. Due to time pressure, we would construct the initial STARC2 design and the individual transformations simultaneously to the final write phase of this graduation project. Because of the particular events in time, the presented fifth low-cost design was the basis for the second high-performance design, leading to poor results (refer to Chapter 5). The major differences between the STARC1 and STARC2 are described in Subsection 1.6.5.

1.5 Tangram Language

Ever since the introduction of the relatively new asynchronous VLSI programming paradigm throughout the world, people consider designing on the level of asynchronous circuits inherently difficult. To abstract from the properties of asynchronous designs (e.g. intricate timing behaviour), Philips Research has designed a dedicated programming language. In this language, called Tangram, the programmer has the opportunity to express even the most complex algorithms using powerful constructs. The Tangram compiler translates these programs into the intermediate representation of so-called handshake circuits. These handshake circuits introduce the actual asynchronous behaviour of Tangram programs and a handshake circuit compiler can translate these into an actual VLSI netlist.
The remaining paragraphs in this section briefly introduce some of the language constructs. Since it is not the goal of this section to teach the reader the ropes of Tangram as a whole, we only list the Tangram ingredients that are necessary to understand the code fragments given in the remainder of this document. In addition, we do not consider crucial design rules one should follow when designing Tangram programs that are irrelevant to understand points we are trying to make (e.g. there are two language constructs for selection for which the proper choice depends on the stability of its guards: this is not discussed here). Code fragments use pseudo code (resembling Dijkstra's Guarded Command Language) to keep Tangram's syntactic sugar from distracting the reader. For a complete guide to the Tangram language, refer to [3]. For an elaboration on handshake circuits, refer to [4].

Code Fragment 1-1 shows sequential composition command. The semicolon symbol represents that command \( C_0 \) must execute (and finish) before command \( C_1 \) executes.

\[
C_0 ; C_1
\]

*Code Fragment 1-1: Sequential composition command.*

Code Fragment 1-2 shows parallel composition command. The two vertical lines symbol represents that both commands \( C_0 \) and \( C_1 \) execute simultaneously. Note that parallel composition has a higher binding priority than sequential composition.

\[
C_0 \mid C_1
\]

*Code Fragment 1-2: Parallel composition command.*

Code Fragment 1-3 shows the selection command. The boolean expressions \( B_0 \) through \( B_n \) are guards. All guards evaluate simultaneously and the selection command selects the command to be executed corresponding to the first guard in the list that evaluates to true. If all guards evaluate to false, it selects none of the commands and finishes immediately. If the keyword `force` prepends the `if` keyword, the command blocks until at least one of the guards evaluates to true (therefore always selecting one of the commands \( C_0 \) through \( C_n \)).

\[
\text{if } B_0 \rightarrow C_0 \\
\quad \text{if } B_1 \rightarrow C_1 \\
\quad \text{...} \\
\quad \text{if } B_n \rightarrow C_n \\
\text{fi}
\]

*Code Fragment 1-3: Selection command.*

Code Fragment 1-4 shows the infinite repetition command, which executes command \( C \), and continuously restarts execution of it as soon as \( C \) finishes.

\[
\text{forever do } C \text{ od}
\]

*Code Fragment 1-4: Infinite repetition command.*
Code Fragment 1-5 shows the assignment command. The number of bits variables $V_0$ through $V_n$ consist of must be equal to the number of bits expressions $E_0$ through $E_m$ consist of. The assignment command evaluates expressions $E_0$ through $E_m$ and stores their concatenated values into variables $V_0$ through $V_n$. When there is only one variable ($n = 0$), one omits the surrounding double angular brackets. Likewise, when there is only one expression ($m = 0$), one omits those surrounding double angular brackets.

$\langle\langle \ V_0, \ldots, \ V_n \rangle\rangle := \langle\langle \ E_0, \ldots, \ E_m \rangle\rangle$

**Code Fragment 1-5: Assignment command.**

Code Fragment 1-5 shows channel communication between two parallel processes. The exclamation mark symbol denotes a send command, which evaluates expression $E$ and then blocks until there is a parallel process willing to receive a value over channel $X$. The question mark symbol denotes a receive command, which blocks until there is a parallel process willing to send a value over channel $X$. When both commands are executing, variable $V$ becomes equal to the value of $E$ and both commands finish.

(... $X!E \ldots$ ) || (... $X?V \ldots$ )

**Code Fragment 1-6: Channel communication between two parallel commands. The ellipses indicate possible other commands.**

### 1.6 STARC2 Micro-Controller

The abbreviation STARC2 is short for Security Transponder and Risc Controller, version 2. The STARC2 micro-controller contains a so-called reduced instruction set computer core. The STARC2 is based on a so-called Harvard architecture, which implies that there is separate code and data memory. It has 16 kilobytes of addressable code memory and 256 bytes of addressable data memory. The instruction set has been optimised for serial data processing and features dedicated operations that speed up calculation of software implemented security algorithms. The reduced instruction set computer core is well suited for security and low-power applications. Refer to [5] for the official specification.

#### 1.6.1 Code Memory

The STARC2 can address 16 kilobytes of code memory. Each instruction consists of 16 bits and thus occupies two bytes, fetched simultaneously. The program counter addresses the code memory instruction by instruction, which in any case aligns to even addresses. The STARC2 features byte-wise addressing of code data, e.g. for lookup tables. After reset, program execution starts from location 0. The STARC2 features seven interrupt vectors; each interrupt binds a fixed location in the code memory. If one enables an interrupt, an interrupt causes the STARC2 to perform a CALL operation to the corresponding location, where it commences execution of the interrupt service routine. The external interrupt controller has to support the STARC2 core with the right interrupt
vector. One places a JMP instruction at the corresponding location, in order to redirect program execution to the location of the interrupt service routine.

1.6.2 Data Memory

The STARC2 features several addressing modes for the data memory, which are described below.

1.6.2.1 Register File

The register file consists of eight bytes in accordance to R0, ..., R7 located at data memory addresses 0 to 7, respectively.

1.6.2.2 Reserved

The memory space in the range of data memory address 8 and 17 is reserved. Accessing these bytes yields undefined results.

1.6.2.3 Program Status Word

The program status word contains several status flags that reflect the current state of the STARC2. The program status word resides at data memory address 18 and contains the overflow flag (for signed operations) at bit index five; halfcarry flag (carry after half the bits in a byte) at bit index six; and carry flag (carry after the last bit in a byte) at bit index seven. The other bits do not have any special function.

1.6.2.4 Stack Pointer

The stack pointer, located at data memory address 19, indicates the address of the top of stack. At each CALL, XCALL and interrupt service routine the program counter return address and the flags, together making up 16 bits, are saved to the data memory. One calls the part of data memory used for this purpose the stack space. The flags are always stored but only restored in case of a return from interrupt. The user must define the location of the stack space, but the user must take care that the stack will not run out of the user RAM. Since the stack pointer only performs 16 bits accesses it does not allow odd addresses. Therefore, the least significant bit of the stack pointer stays zero even if an instruction writes a one to it\(^1\). The stack pointer supports a downward increasing stack with a pre decrement and post increment feature. Because of the pre decrement it is necessary to position the stack pointer at the desired top of stack address plus one, e.g. the stack pointer must be set to 256 (which equals the last element of user RAM plus one) in order to lay the top of stack at the end of the user RAM. The first writing access accomplishes at address 254 and the 16 bit data is stored at address 254 and 255.

1.6.2.5 Data Bit Pointer

The data bit pointer consists of two separate registers, the eight bits wide byte register and the four bits wide bit register, and comprises the entire addressing space including the special function registers. The byte register selects a byte in the data memory address space. Three bits in the bit register, located at indices zero through two, select a specific

\(^1\) This property implies that the stack pointer is actually a variable having a bit width of seven bits.
bit inside the selected byte. Another bit in the bit register, called map bit and located at index seven, specifies whether the least significant bit is mapped to index zero (map bit equals false) or the most significant bit is mapped to index zero (map bit equals true). The remaining bits of the bit register have a fixed value of false. The combined register allows access to a single bit in bit-oriented instructions by selecting the proper addressing mode. It features an optional auto post increment. In auto post increment mode the data bit pointer cannot address itself as the auto increment has always priority.

1.6.2.6 Special Function Registers
The special function registers space enables input/output from/to the peripherals. The physical positions of the special function registers are in the corresponding peripheral devices. The address space for special function registers equals a fixed and consecutive range of addresses between the data bit pointer register and the user RAM. A dedicated bus communicates to the corresponding peripheral accesses to the special function registers.

1.6.2.7 User RAM
The user RAM starts immediately after the special function register space and ends at data memory address 255.

1.6.3 Addressing Modes
The STARC2 features immediate byte addressing, direct byte addressing, direct bit addressing, indirect byte addressing, indirect bit addressing and indexed byte addressing modes. The following describes them.

1.6.3.1 Immediate Byte Addressing Mode
Only the destination being a register supports immediate byte addressing. The eight bits value of the constant to be loaded into the register is included in the instruction word.

1.6.3.2 Direct Byte Addressing Mode
Only the first 128 bytes in the data memory support direct byte addressing. A seven bits address field in the instruction word specifies the direct byte addressing the operand. The register file, the program status word, the special function registers, the data bit pointer, the stack pointer and the user RAM can be addressed directly.

1.6.3.3 Indirect Byte Addressing Mode
All 256 bytes in the data memory support indirect byte addressing. In indirect addressing one of the registers R0 to R3 contain the address of the operand. A two-bit register select field in the instruction word specifies the register used for indirect addressing. The registers R0 and R1 support an indirect auto increment and decrement addressing mode. The possibilities are indirect post increment and indirect pre decrement addressing. Using post increment the content of the register is taken as data address and the register will be loaded with the value incremented by one afterwards. Using pre decrement the STARC2 immediately decrements the content of the register decremented by one and takes this value as data address. The decremented value will be loaded into the register afterwards. If the register addresses itself in auto addressing mode, it ignores the asserted data and
the register will be loaded with the incremented and decremented value of the register, respectively. That means that in auto addressing mode the register cannot address itself, as the auto addressing has always priority.

1.6.3.4 Direct Bit Addressing Mode
In direct bit addressing one of the eight operand bits is specified by a three bit select field in the instruction word (next to the direct byte address indicating the byte operand).

1.6.3.5 Indirect Bit Addressing Mode
In indirect bit addressing a data bit pointer addresses the entire addressing space. The indirect bit addressing supports an optional auto increment mode. In this mode the data bit pointer is increased by one after each access. In auto addressing mode the data bit pointer cannot address itself as the auto increment has always priority.

1.6.3.6 Indexed Byte Addressing Mode
Indexed byte addressing can only access code memory and it can only be a read or a branch operation executed. Reading lookup tables in code memory and branching by means of a vector table intends this addressing mode. The registers R4 and R5 assemble a 14 bits address to the designated code byte. In case of a branch, the STARC2 ignores the least significant bit and considers it zero, since all instructions are aligned at even addresses.

1.6.4 Instruction Set
The STARC2 offers a powerful instruction set featuring read-modify-write instructions. Instructions can be categorised into arithmetic operations, logical operations, boolean manipulation operations, data transfer operations, branch operations and miscellaneous operations. Below the instructions for each category are explained. \textit{Mem} denotes Memory operands, which either direct or indirect byte addressing can obtain (in some instructions also immediate byte addressing is allowed; this will be specified within its description). \textit{Reg} denotes register operands, which must specify the exact register (i.e. R0). \textit{Mem} (indicating the byte operand) and \textit{Bit} (indicating the bit index within the byte operand) denote bit operands, which direct or indirect bit addressing can obtain. \textit{Addr} denotes Code addresses, which absolute addressing must obtain.

1.6.4.1 Arithmetic Category

- \textbf{ADD Mem, Reg}
  Add one of the operands to the other, updating all flags. The instruction word determines which operand is the destination. When register is the destination, it also allows immediate byte addressing.

- \textbf{ADDC Mem, Reg}
  Add with carry one of the operands to the other, updating all flags. The instruction word determines which operand is the destination. When register is the destination, it also allows immediate byte addressing.

- \textbf{SUBC Mem, Reg}
  Subtract with carry one of the operands to the other, updating all flags. The
instruction word determines which operand is the destination. When register is the destination, it also allows immediate byte addressing.

- **INC Mem**
  Increment operand by one.

- **DEC Mem**
  Decrement operand by one.

1.6.4.2 Logical Category

- **AND Mem, Reg**
  Conjunct one of the operands to the other. The instruction word determines which operand is the destination. When register is the destination, it also allows immediate byte addressing.

- **OR Mem, Reg**
  Disjunct one of the operands to the other. The instruction word determines which operand is the destination. When register is the destination, it also allows immediate byte addressing.

- **XOR Mem, Reg**
  Exclusively disjunct one of the operands to the other. The instruction word determines which operand is the destination. When register is the destination, it also allows immediate byte addressing.

- **CLR Mem**
  Clear operand.

- **SET Mem**
  Set operand.

- **CPL Mem**
  Complement operand.

- **RLC Mem**
  Rotate left operand through carry.

- **RRC Mem**
  Rotate right operand through carry.

- **SHL Mem**
  Shift left operand into carry, inserting false.

- **SHR Mem**
  Shift right operand into carry, inserting false.

- **SHRB Bit**
  Shift right R0 into carry, inserting operand.

- **SWAP Mem**
  Swap nibbles within operand.
1.6.4.3 Boolean Manipulation Category

- **CLR BMem, Bit**
  Clear bit of operand.

- **SETB BMem, Bit**
  Set bit of operand.

- **CPLB BMem, Bit**
  Complement bit of operand.

- **MOVTOC Bit**
  Move bit of operand into carry.

- **MOVFROMC Bit**
  Move carry into bit of operand.

1.6.4.4 Data Transfer Category

- **MOV BMem, Reg**
  Move one of the operands to the other. The instruction word determines which operand is the destination. When register is the destination, it also allows immediate byte addressing.

- **MOVC**
  Move indexed code byte addressed by R4 and R5 into R0.

1.6.4.5 Branch Category

- **JMP Addr**
  Absolute jump to operand.

- **CALL Addr**
  Absolute subroutine call to operand.

- **XJMP**
  Indexed jump addressed by R4 and R5.

- **XCALL**
  Indexed subroutine call addressed by R4 and R5.

- **CPSE BMem, Reg**
  Compare one of the operands with the other, updating the flags as if performing a subtraction, and skip the next instruction if equal. The instruction word determines which operand is the imaginary destination of the subtraction. When register is the destination, it also allows immediate byte addressing.

- **DSZ BMem**
  Decrement operand by one and skip next instruction if becomes zero.

- **DSNZ BMem**
  Decrement operand by one and skip next instruction if not becomes zero.
• **ISZ Mem**  
  Increment operand by one and skip next instruction if becomes zero.

• **ISNZ Mem**  
  Increment operand by one and skip next instruction if not becomes zero.

• **SBS Mem, Bit**  
  Skip next instruction if bit in operand is set.

• **SBC Mem, Bit**  
  Skip next instruction if bit in operand is cleared.

• **SZ Mem**  
  Skip next instruction if operand is zero.

• **SNZ Mem**  
  Skip next instruction if bit in operand is not zero.

• **RET**  
  Return from subroutine.

• **RETI**  
  Return from interrupt, restoring flags.

### 1.6.4.6 Miscellaneous Category

• **NOP**  
  No operation.

### 1.6.5 STARC1 Differences

The STARC2 is the successor of the STARC1, which was initially to be designed for the customer. The major differences between the STARC1 and the STARC2 consist of the following:

• The STARC1 does not have an addressable stack pointer and its stack space does not reside in the data memory. Instead, it has a hidden stack pointer and a hidden stack space.

• The STARC1 does not have the indirect bit addressing mode and its corresponding data bit pointer.

• The STARC1 does not have the indirect auto increment and decrement addressing mode.

• The STARC1 places the program status word somewhere else in data memory.

• The STARC1 contains less code memory and therefore all pointers to code memory are of a different size.
1.7 Measurement Conventions

This section explains what we measure throughout the document, the exact measuring method, and how to interpret the resulting statistics. In order to compare Tangram designs quantitatively, we have chosen the following method. Taking a Tangram design, we compile this into a handshake circuit (using Tangram tool tg2hc). This handshake circuit is compiled into a standard-cell CMOS18\(^2\) netlist (using Tangram tool hc2v). On this netlist, so-called peephole optimisations (locally replacing small groups of standard cells by equivalent, but better, groups of standard cells) are performed (using Cadence tool ambit). Note that no care has been taken concerning the testability of the design. We simulate the resulting netlist and run a certain (externally delivered) STARC2 program (also called test bench) on it that is to act as representative for all STARC2 programs. This program lacks communications over the special function registers bus. We perform the following measurements:

1. We estimate the chip size by accounting for the size of each standard cell, as well as an appropriate wire model (using a quantity of area and square millimetres as unit).

2. The before-mentioned simulation results in a performance figure (using a quantity of speed and million instructions per second as unit).

3. The before-mentioned simulation results in a power figure (using a quantity of energy dissipation per instruction and pico joules as unit). These numbers are solely based on energy dissipation by transitions within standard cells (wire loads are not taken into account).

There are two notes concerning memories in the simulation. First, their energy dissipation is not taken into account here (energy dissipation allows for a modular separation, and we are primarily interested in the behaviour of the STARC2; instead we mention when a change in number of memory accesses has been detected due to transformations). Second, their delays are set at 2 ns (delays can overlap, therefore inhibit a modular separation; it is a realistic delay for on-chip memories in this CMOS18 process).

\(^2\) Process version 2.1, voltage 1.80, temperature 25.0.
2 Initial Design

This chapter hierarchically presents an initial design of the STARC2 micro-controller. Section 2.1 starts off by introducing the main architecture. It discusses the objects that make up the design, as well as their interaction with each other. After this, Section 2.2 discusses (most of) the actual pseudo code of the STARC2. In this architecture and design, we start at the top level and introduce more detail in a modular fashion as we go along. Emphasis is on readability. Because of this, we omit irrelevant details that do not contribute to the understanding of the actual design.

In Chapter 3 some design decisions are taken on the initial design that are good for its area, power and speed (so-called macho transformations). Chapters 4 and 5 introduce design decisions primarily concerning only low-cost or only high-performance, respectively.

2.1 Conceptual Architecture

Micro-controllers successively execute instructions from a code memory. They do so in an order specified by these very same instructions, by means of updating the program counter in such a way that a certain predictable instruction flow results. Furthermore, and most importantly, most instructions can also operate on data available to the micro-controller. Instructions obtain this data from the register file, special function registers, the flags, the code memory and the user RAM. The STARC2 can perform all this too.

2.1.1 Execution Bits

Before making conceptual distinctions, this subsection introduces the notion of execution bits, control bits and behaviour bits (which are used in the actual design). Each instruction resides encoded in its instruction word in order to reduce code size. Therefore, as a means of decoding information within instruction words, we define boolean functions over the instruction word. We call the results of these boolean functions execution bits and one can see them as extractions of properties of instructions. We distinguish two kinds of execution bits: control bits indicate what hardware tasks instructions should perform (on the level of commands) and behaviour bits indicate how these hardware tasks must behave and manipulate data (on the level of expressions within guards and assignments). Note that the concept of execution bits is a concept over expressions, not over variables\(^3\).

\(^3\) Naturally, the values of these execution bits can be stored in variables as any other value. However, we refrain from choosing to do so in this chapter.
2.1.2 Phase Distinction

One can look at the data functionality of a micro-controller from a conceptual viewpoint by distinguishing five phases in which the STARC2, and therefore the currently executed instruction, can be active. Each instruction traverses these phases, in one sequential order, as follows.

2.1.2.1 Fetch Instruction Phase

The first phase is the fetch instruction phase. In here, the STARC2 fetches from code memory the instruction pointed to by the program counter (this is the instruction to be executed now).

2.1.2.2 Decode Instruction Phase

The second phase is the decode instruction phase. In this phase, the STARC2 examines the fetched instruction and determines what data the instruction needs and what to do with it. Note that evaluation of execution bits is not restricted to this phase alone; other phases also make use of execution bits and must therefore evaluate them as well.

2.1.2.3 Fetch Data Phase

The third phase is the fetch data phase. This is the phase where all necessary data is actually gathered and made instantly accessible for the next phase. If the instruction does not need any data, which becomes clear in the second phase, this phase does not contribute to the effect of the instruction.

2.1.2.4 Compute Data Phase

The fourth phase is the compute data phase. In here, the STARC2 operates on the instantly accessible data and/or constants to obtain a certain result. If the instruction does not need to compute anything, which again becomes clear in the second phase, this phase does not contribute to the effect of the instruction either.

2.1.2.5 Store Data Phase

The fifth and final stage is the store data phase. This phase stores the result (in the form of a value and/or flags) obtained in the fourth phase into its destination location. If the instruction does not need to store data, which becomes clear in the second phase, this phase does not contribute to the effect of the instruction.

* * *

Besides the data functionality, there is also control functionality in each of the instructions. Some instructions (the branch instructions) explicitly define the instruction flow, i.e. the next instruction to execute. The rest of the instructions do not define how the instruction flow continues. For these instructions hold that the STARC2 next executes the instruction present at the code address successive to the current one. For simplicity reasons micro-controllers usually increment the program counter by one during the fetch instruction phase to fulfil the majority of instructions that do not have the explicit definition for control flow. The STARC2 amends for exceptions to the rule as soon as the necessary information about the next instruction is available (most branch instructions offer a choice between different instruction flows based on certain data).
Note that the functionality of an interrupt is equivalent to a CALL instruction. Therefore, one can also view the handling of interrupts in the context of these phases.

2.1.3 Object Distinction

Section 1.6 explained that the STARC2 continuously executes instructions and interrupts. Subsection 2.1.2 explained that instruction execution and interrupt handling boil down to traversing five phases. This subsection introduces objects that one can view as an intermediary between the conceptual phases and the actual design. These objects all represent part of the total STARC2 functionality and are to collaborate and interact with each other such a way that the complete STARC2 functionality is covered. Section 2.2 translates these objects into an actual design in pseudo code.

Since there are 40 different instructions, of which most abide to some form of regularity in structure, we refrain from giving descriptions for the actual instruction-specific objects. Instead, we give an object template that indicates what these instruction-specific objects look like.

2.1.3.1 STARC2 Object

This object is the top-level object that operates on the highest level of both hierarchy and abstraction. It repeatedly checks for pending interrupt requests, in which case it activates the internal interrupt controller object. Otherwise, it performs the next instruction by signalling the instruction execution object. When done, it starts over checking for interrupts, etc.

2.1.3.2 Internal Interrupt Controller Object

The internal interrupt controller object takes care of all communication with external interrupts as soon as the STARC2 object detects them. It signals the stack object to push the current program counter and the flags onto the stack. In addition, it receives the interrupt vector, acknowledges the corresponding interrupt request and points the program counter to the start of the appropriate interrupt service routine.

2.1.3.3 Stack Object

The stack object features two similar functions: pushing elements onto the stack and popping elements from the stack. When signalled to perform a push it moves the stack top by decrementing the stack pointer by one. After this, it writes the new element into the user RAM at the location of the stack top. When signalled to perform a pop it fetches from the user RAM an element at the location of the stack top pointed to by the stack pointer. After this, it moves the stack top by incrementing the stack pointer by one.

2.1.3.4 Instruction Execution Object

The instruction execution object supervises the traversal of the five instruction phases. First, it signals the instruction fetcher object to fetch the next instruction. Then, it signals the instruction decoder object to take care of selecting the proper actions concerning fetching, operating, and storing of data, as well as redirecting the default instruction flow in case of a branch instruction.
2.1.3.5 Instruction Fetcher Object

The instruction fetcher object takes care of supplying instructions that must execute. It fetches the next instruction word pointed to by the program counter from code memory. In addition, it increments the program counter by one so that the default (i.e. non-branch instruction) instruction flow is fulfilled.

2.1.3.6 Instruction Decoder Object

The instruction decoder object takes care of selecting the proper actions concerning fetching, operating, and storing of data, as well as redirecting the default instruction flow in case of a branch instruction. It derives from the instruction word which instruction it is (by means of control bits). Depending on the result, it signals the appropriate instruction-specific object. Remember that in this subsection the actual instruction-specific objects are not given.

2.1.3.7 Data Fetcher Object

The data fetcher object takes care of supplying data that another object must process. It features three functions: fetching the memory operand from data memory using the address derived from the appropriate addressing mode, fetching the register operand from the register file and fetching the bit index pointing in the memory operand.

2.1.3.8 Data Storage Object

The data storage object takes care of storing data that another object generates. It features two functions: storing a result value into data memory and updating pointers used for auto addressing.

2.1.3.9 Instruction-Specific Object Template

This is a template for instruction-specific objects. This means it is not an actual object, but only gives an indication of what instruction-specific objects look like. All instruction-specific objects consist of the fetch data phase, the compute data phase and the store data phase. Instruction-specific objects concerning instruction that use data start by signalling the data fetcher object to fetch the memory operand, register operand and/or bit index, simultaneously; otherwise, this phase is empty. Then, instruction-specific objects concerning instructions that generate data continue by performing the task that generates that resulting data; otherwise, this phase is empty. Instruction-specific objects concerning instructions that store data, update pointers or redirect the instruction flow finish by assigning a new value to the program counter and/or signalling the data storage object to store the result and/or pointer used by auto addressing, simultaneously. Instruction-specific objects concerning call and return instructions also signal the stack object to push and pop the program counter.

2.2 Pseudo Code

This section presents pseudo code for the initial design. It subsequently translates the objects into (pseudo code) procedures. An exception to this is the instruction-specific object template, for which it gives example procedures for one representative instruction
out of every instruction category. A procedure call implements the concept of activating an object or one of its multiple functions. In order to simplify (the implementation of) communication between (the implementation of) different objects, we choose to take variables as an interface to pass values to other objects. We define them globally so we can easily move commands around in subsequent transformations without destroying scope dependencies. Instead of formally specifying the values of control bits, we only indicate their meaning textually. By convention, the names of (boolean) control bits start with $CB$, and the names of integer functions on the instruction word start with $Instr$. Appendix A consists of a compilation of the variables (together making up the STARC2 state space), the constants and the channels used throughout the initial design. Note that it also shows the resources involving the bus implementation. However, we refrain from giving an implementation for the algorithm behind it, see also footnote Error! Bookmark not defined. at page Error! Bookmark not defined.

2.2.1 STARC2 Procedure

Code Fragment 2-1 shows the top-level design. It consists of an infinite repetition in which instructions and pending interrupts are processed. In case of an interrupt request, identified by a pending handshake on channel $ChanInt$, the internal interrupt controller $IntControl$ processes it. Otherwise, $ExecInstr$ executes an instruction.

```plaintext
STARC2 =
    \[ PC := 0; \]
    forever do →
        if probe(ChanInt) → IntControl()
        0 → probe(ChanInt) → ExecInstr()
        fi
    od |
```

*Code Fragment 2-1: The top level repeatedly processes instructions and interrupts.*

2.2.2 Internal Interrupt Controller Procedure

Code Fragment 2-2 shows the internal interrupt controller. It starts by pushing the program counter and the flags onto the stack. Then, the program counter receives the address to the start of the corresponding interrupt service routine, which $ChanInt$ transmits.

---

4 The choice for a suitable representative is based solely on (the presence of) an optimal coverage of mutual sub-functionality (within the instruction) that is not yet illustrated by former representatives.

5 In a pseudo code presentation, complex bit-level expressions only cloud the mind.
IntControl =
[[ StackPush();
  ChanInt?PC ]]

Code Fragment 2-2: The internal interrupt controller.

2.2.3 Stack Procedures

Code Fragment 2-3 shows stack-push of the value of the program counter and flags. First, the stack pointer is decremented by one. Then, the least and most significant bytes of the program counter and flags are stored into the user RAM at word addresses pointed to by the stack pointer, respectively.

StackPush =
[[ SP := SP - 1;
  RAM[<<0, SP>>] := LsByte(<<PC, Flags>>);
  RAM[<<1, SP>>] := MsByte(<<PC, Flags>>) ]]

Code Fragment 2-3: Stack-push of the value of the program counter and flags.

Code Fragment 2-4 shows stack-pop of the value of the program counter and possibly the flags. In one version of this procedure, it first fetches the most and least significant bytes of the stack word addressed by the stack pointer into the program counter and flags. The other version of the procedure does not update the flags. The first version is used in the RETI instruction, while the latter version is used in the RET instruction. Then, the stack pointer is incremented by one.

StackPop =
[[ MsByte(<<PC, Flags>>) := RAM[<<1, SP>>];
  LsByte(<<PC, Flags>>) := RAM[<<0, SP>>];
  SP := SP + 1 ]]

Code Fragment 2-4: Stack-pop of the value of the program counter and flags. Note that there are two versions: one that updates the flags and one that does not. A curly underline indicates the optional code.

2.2.4 Instruction Execution Procedure

Code Fragment 2-5 shows instruction execution. First, it fetches the next instruction using FetchInstr. Then, it decodes the instruction and performs the instruction's functionality using DecodeInstrFetchComputeStoreData.

ExecInstr =
[[ FetchInstr(); DecodeInstrFetchComputeStoreData() ]]

Code Fragment 2-5: Execution of an instruction.
2.2.5 Instruction Fetcher Procedure

Code Fragment 2-6 shows the instruction fetcher. It fetches the instruction in code memory (array ROM) pointed by the program counter into Instr. After this, the program counter is incremented by one to make it point to the next instruction in code memory.

```
FetchInstr =
  | [ Instr := ROM[PC];
      PC := PC + 1 ] |
```

*Code Fragment 2-6: The instruction fetcher.*

2.2.6 Instruction Decoder Procedure

Code Fragment 2-7 shows instruction decoding. For each possible instruction there is a separate control bit defined, which evaluates to true if the corresponding instruction must execute. A conditional statement consisting of 40 alternatives (one for each instruction) selects the corresponding code fragment for the actual instruction (using ExecXXX as naming convention, where XXX denotes the instruction).

```
DecodeInstrFetchComputeStoreData =
  | [ if CBAdd  → ExecADD();
      ∗ CBAaddc → ExecADDC();
      ∗ CBAnd   → ExecAND();

      ... ∗
      ∗ CBXcall  → ExecXCALL();
      ∗ CBXjmp  → ExecXJMP();
      ∗ CBXxor → ExecXOR();
      fi ] |
```

*Code Fragment 2-7: Decoding of the instruction and executing the corresponding procedure for it.*

2.2.7 Data Fetcher Procedures

Code Fragment 2-8 shows the bit-index operand fetcher. If DBP indirect addressing is used (indicated by CBDBP) then the value of variable DBPBit, exclusively disjuncted with a vector of DBPMAP, is stored into OpBit. Otherwise, the bit-index value encoded inside the instruction (InstrOpBit) is stored into OpBit.
Code Fragment 2-8: The bit-index operand fetcher.

Code Fragment 2-9 shows the register operand fetcher. The address of the register operand encoded within the instruction (indicated by \textit{InstrOpReg}) indexes the corresponding register value located in the register file (indicated by \textit{RF}). This value is stored into \textit{OpReg}.

\begin{verbatim}
FetchOpReg =
| [ OpReg := RF(\text{InstrOpReg}) ] |
\end{verbatim}

Code Fragment 2-9: The register operand fetcher.

Code Fragment 2-10 shows the memory operand fetcher. Depending on the addressing mode the actual address is stored into \textit{OpMemAddr} (except for immediate addressing, which uses no address at all). This address originates from register file, variable DBPByte or direct address encoded within the instruction. In case of pre decrement addressing, the register value is decremented by one (done here by definition of ‘pre’). Post incrementation is not performed until the store data phase (in procedure \textit{UpdateInd}). Finally, \textit{DerefOpMemAddr} dereferences the address inside \textit{OpMemAddr} to store the value it is addressing into \textit{OpMem} using.

\begin{verbatim}
FetchOpMem =
| [ if CBImm \rightarrow OpMem := InstrImmOpMem |
| \& CBInd \rightarrow OpMemAddr := RF(\text{InstrInd}); \text{DerefOpMemAddr}() |
| \& CBPreDecR0 \rightarrow OpMemAddr := R0 - 1; \text{DerefOpMemAddr}() |
| \& CBPostIncR0 \rightarrow OpMemAddr := R0; \text{DerefOpMemAddr}() |
| \& CBPreDecR1 \rightarrow OpMemAddr := R1 - 1; \text{DerefOpMemAddr}() |
| \& CBPostIncR1 \rightarrow OpMemAddr := R1; \text{DerefOpMemAddr}() |
| \& CBDBP \rightarrow OpMemAddr := DBPByte; \text{DerefOpMemAddr}() |
| \& CBPostIncDBP \rightarrow OpMemAddr := DBPByte; \text{DerefOpMemAddr}() |
| \& CBSDir \rightarrow OpMemAddr := InstrDir; \text{DerefOpMemAddr}() ] |
\end{verbatim}

Code Fragment 2-10: The memory operand fetcher.

Code Fragment 2-11 shows the memory address dereferencer. Depending on the actual address to be read from, either a register in the register file, the flags and the rest of the program status word (variable \textit{RestPSW}), the stack pointer, DBPBit and DBPMap, DBPByte, the bus or an element in user RAM is stored into \textit{OpMem}, respectively. The
actual implementation concerning bus communication (procedure ReadBus) does not show here, refer to [2] for a discussion on that topic.

DerefOpMemAddr =

\[
\begin{align*}
& \text{if } \text{OpMemAddr} < 8 \rightarrow \text{OpMem} := \text{RF(OpMemAddr)} \\
& \text{OpMemAddr} = 18 \rightarrow \text{OpMem} := <<\text{RestPSW}, \text{Flags}>> \\
& \text{OpMemAddr} = 19 \rightarrow \text{OpMem} := <<0, \text{SP}}>> \\
& \text{OpMemAddr} = 20 \rightarrow \text{OpMem} := <<\text{DBPBit}, 0, 0, 0, 0, \text{DBPMap}>> \\
& \text{OpMemAddr} = 21 \rightarrow \text{OpMem} := \text{DBPByte} \\
& \text{StartAddrSFR} \leq \text{OpMemAddr} < \text{StartAddrUserRAM} \rightarrow \text{ReadBus}() \\
& \text{StartAddrUserRAM} \leq \text{OpMemAddr} \leq \text{EndAddrUserRAM} \rightarrow \text{OpMem} := \text{RAM[OpMemAddr]}
\end{align*}
\]

Code Fragment 2-11: The memory address dereferencer.

### 2.2.8 Data Storage Procedures

Code Fragment 2-12 shows storage of the result value. Depending on the actual address to write to, a variable (a register, the program status word, the stack pointer, DBPBit, DBPMap or DBPByte), the bus or an element in user RAM is updated with the value of ResByte, respectively. Note that variables that instructions can use as auto pointers have a conjuncted control bit in their guard, preventing a possible overwrite of the updated pointer. In addition, note that the result value only overwrites the individual flag variables if the corresponding control bits allow it. This has been done to allow flag-modifying instructions (e.g. the ADD instruction) to update flags during their compute data phase without possibly overwriting them afterwards with a result value. Again, the actual implementation concerning bus communication does not show here.
StoreRes =

```plaintext
[if 0 ≤ ResAddr < StartAddrSFR →
  if ¬CBAutoR0 ∧ ResAddr = 0 → R0 := ResByte
  0 ¬CBAutoR1 ∧ ResAddr = 1 → R1 := ResByte
  0 ResAddr = 2 → R2 := ResByte
  ...
  0 ResAddr = 7 → R7 := ResByte
  0 ResAddr = 18 →
    RestPSW := ResByte0..4 ||
    if CBOverwriteOV → OV := ResByte5 fi ||
    if CBOverwriteHC → HC := ResByte6 fi ||
    if CBOverwriteC → C := ResByte7 fi
  0 ResAddr = 19 → SP := ResByte1..7
  0 ¬CPostIncDBP ∧ ResAddr = 20 →
    DBPBit := ResByte2..7 || DBMap := ResByte8
  0 ¬CPostIncDBP ∧ ResAddr = 21 → DBPByte := ResByte
fi]

StartAddrSFR ≤ ResAddr < StartAddrUserRAM → WriteBus()

```

*Code Fragment 2-12: Storage of the result value.*

Code Fragment 2-13 shows storage of the indirection pointer. Depending on the actual pointer used for accessing data (indicated by the appropriate control bits) the corresponding register or variable updates (unless no indirection is used). In case of pre decrement addressing the pointer is already decremented by one during memory operand fetch. In case of post increment DBP addressing, the concatenated DBPBit and DBPByte should be incremented to obtain the proper result (done here by definition of ‘post’).

UpdateInd =

```plaintext
[if CBAutoDecR0 → R0 := OpMemAddr
  0 CBAutoIncR0 → R0 := OpMemAddr + 1
  0 CBAutoDecR1 → R1 := OpMemAddr
  0 CBAutoIncR1 → R1 := OpMemAddr + 1
  0 CBAutoIncDBP → <<DBPBit, DBPByte>> := <<DBPBit, DBPByte>> + 1
fi]
```

*Code Fragment 2-13: Storage of the indirection pointer.*
2.2.9 Arithmetic Representative Procedure

The SUBC instruction serves the purpose of representative for the arithmetic category. Code Fragment 2-14 shows execution of it. First, it fetches the values of the memory using FetchOpMem and register operands using FetchOpReg into variables OpMem and OpReg, respectively. Based on the destination operand, it then subtracts the operands, writes the result value and result flags into variables ResByte and Flags, respectively, and writes the result address into variable ResAddr. Finally, it stores the result into its destination and (when necessary) stores the updated indirection address.

```c
ExecSUBC =
[[ FetchOpMem(); ]]

if CBDestOpMem \rightarrow <<ResByte, Flags\rangle := OpMem - OpReg - C ||
    ResAddr := OpMemAddr
[] CBDestOpReg \rightarrow <<ResByte, Flags\rangle := OpReg - OpMem - C ||
    ResAddr := InstrOpReg
fi;
StoreRes() || UpdateInd() ||
```

*Code Fragment 2-14: Execution of the SUBC instruction.*

2.2.10 Logical Representative Procedure

The RLC instruction serves the purpose of representative for the logical category. Code Fragment 2-15 shows execution of it. First, it fetches the value of the memory operand using FetchOpMem into OpMem. Then, it rotates the operand and the carry flag to the left, writes the result value into ResByte and the carry flag and writes the result address into ResAddr. Finally, it stores the result into its destination and (when necessary) stores the updated indirection address.

```c
ExecRLC =
[[ FetchOpMem(); ]]

StoreRes() || UpdateInd() ||
```

*Code Fragment 2-15: Execution of the RLC instruction.*

2.2.11 Boolean Manipulation Representative Procedure

The CLRB instruction serves the purpose of representative for the boolean manipulation category. Code Fragment 2-16 shows execution of it. First, it fetches the values of the

---

\(^6\) Remember that this is merely pseudo code; the subtraction operators in the expressions for this assignment and the one below should also provide resulting flag information.
memory operand using \textit{FetchOpMem} and bit-index operand using \textit{FetchOpBit} into \textit{OpMem} and \textit{OpBit}, respectively. Then, in \textit{OpMem} it substitutes the bit indexed by \textit{OpBit} with false, writes the result into \textit{ResByte} and writes the result address into \textit{ResAddr}. Finally, it stores the result into its destination and (when necessary) stores the updated indirection address.

\begin{verbatim}
ExecCLRBB =
|| [ FetchOpMem ()   | FetchOpBit () ;
    StoreRes ()   | UpdateInd () ]
\end{verbatim}

Code Fragment 2-16: Execution of the CLRB instruction.

\subsection*{2.2.12 Data Transfer Representative Procedure}

The \textit{MOVC} instruction serves the purpose of representative for the data transfer category. Code Fragment 2-17 shows execution of it. Depending on the value of the least significant bit of \textit{R4}, it fetches and stores the most/least significant byte of the code word addressed by \textit{R4} and \textit{R5} into \textit{R0}.

\begin{verbatim}
ExecMOVC =
|| [ if R4o → R0 := MSByte(ROM[<<R41..7, R50..5>>])
   0 ←R4o → R0 := LByte(ROM[<<R41..7, R50..5>>])
    fi ]
\end{verbatim}

Code Fragment 2-17: Execution of the MOVC instruction.

\subsection*{2.2.13 Branch Representative Procedure}

The \textit{ISZ} instruction serves the purpose of representative for the branch category. Code Fragment 2-18 shows execution of it. First, it fetches the value of the memory operand into \textit{OpMem}. Then, it increments the operand by one, writes the result value into \textit{ResByte} and writes the result address into \textit{ResAddr}. Finally, it increments the program counter by one if the result value equals zero, stores the result into its destination and (when necessary) stores the updated indirection address.

\begin{verbatim}
ExecISZ =
|| [ FetchOpMem();
    if ResByte = 0 → PC := PC + 1 fi ||
    StoreRes () || StoreInd () ]
\end{verbatim}

Code Fragment 2-18: Execution of the ISZ instruction.
2.2.14 Miscellaneous Representative Procedure

The NOP instruction serves the purpose of representative for the miscellaneous category. Code Fragment 2-19 shows (lack of) execution. As can be expected, nothing happens.

\[
\text{ExecNOP} = \text{[[ skip ]]} \]

Code Fragment 2-19: Execution of the NOP instruction.

\[
\text{** ** **}
\]

Measurement 2-1 shows the measured statistics for the initial design.

<table>
<thead>
<tr>
<th>Design</th>
<th>Area (mm(^2))</th>
<th>Energy/Instr (pJ)</th>
<th>Speed (MIPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>0.071</td>
<td>35.9</td>
<td>30.8</td>
</tr>
</tbody>
</table>

*Measurement 2-1: Measured statistics for the initial design.*
3 Macho Transformations

The initial design of the STARC2 micro-controller, as introduced in Section 2.2, is correct and intuitive. However, there is a small number of transformations to improve all quantities measured. Since these transformations are always beneficial to our design, we call them macho transformations\(^7\). The first transformation is not a new one. The second transformation is the work of the author.

3.1 Exploitation of Similarity

Since it appears to be beneficial to share hardware for identical commands, it seems wise to exploit this property as much as possible. In general, a set of expressions that are not textually identical to each other cannot share hardware. However, some expressions are equivalent nevertheless. The conjunction, disjunction, exclusive disjunction, addition and addition with carry operators are all commutative. Therefore, all commands of the form shown in Code Fragment 3-2 can replace the inferior commands shown in Code Fragment 3-1.

\[
\begin{align*}
\text{if } \text{CBDestOpMem } \rightarrow \text{ResByte} := \text{OpMem } \Theta \text{OpReg} & \lor \text{ResAddr} := \text{OpMemAddr} \\
\text{0 CBDestOpReg } \rightarrow \text{ResByte} := \text{OpReg } \Theta \text{OpMem} & \lor \text{ResAddr} := \text{InstrOpReg}
\end{align*}
\]

**Code Fragment 3-1:** Two instantiations of the same operator \(\Theta\) requires twice the amount of hardware for evaluation of the result value.

\[
\begin{align*}
\text{ResByte} := \text{OpMem } \Theta \text{OpReg} & \lor \\
\text{if } \text{CBDestOpMem } \rightarrow \text{ResAddr} := \text{OpMemAddr} \\
\text{0 CBDestOpReg } \rightarrow \text{ResAddr} := \text{InstrOpReg}
\end{align*}
\]

**Code Fragment 3-2:** One instantiation of the operator \(\Theta\) since its commutativity implies equivalence. This saves an instantiation and allows the other to factor out of the conditional statement. In addition, one multiplexer is saved.

Taking this motivation somewhat further, non-equivalent expressions can sometimes be generalised into a more (hardware) sharable form. Of course, this is only beneficial (in terms of area, power and speed) if the penalty for the generalisation is less than the benefits of sharing. The arithmetic operators (addition, addition with carry, subtraction, subtraction with carry, increment and decrement) appear to be good candidates for this exploitation. For \(p_0, p_1, p_2, p_3 \in \{-1, 0, 1\}\), generalised expression

\[
p_0 \times \text{OpMem} + p_1 \times \text{OpReg} + p_2 \times C + p_3
\]

\(^7\) The term macho transformation has been proudly introduced by Andrew Bailey.
can be configured equivalent to all of the above operators. The proper choices for each of
the parameters are dependent on the current instruction and therefore, behaviour bits can
implement them, as introduced in Subsection 2.1.1. Commands of the form shown in
Code Fragment 3-3 can replace the inferior commands that straightforwardly implement
the arithmetic operator functionality.

```
if CBDestOpMem → ResAddr := OpMemAddr
‖ CBDestOpReg → ResAddr := InstrOpReg
fi
```

*Code Fragment 3-3: One instantiation of the generalised expression saves a number
of instantiations and allows it to factor out of the conditional statement.*

In addition, a generalisation is possible for the assignments in which a single bit
in *OpMem* at index *OpBit* modifies. The corresponding generalised command is such that
the new value is a parameter of the command.

Similarly, a generalisation can take place for shifting and rotating. The
 corresponding generalised command is such that the inserted bit is a parameter of the
command.

* * *

These optimisations result in the improvements shown in Measurement 3-1. The
percentage figures within parentheses are a comparison with the results of the initial
design. Area decreases on operator implementations, data path multiplexers, parallel
compositions and conditional statements. Energy decreases because of fewer transitions
and speed increases because of fewer multiplexers in the data path and fewer parallel
compositions in the control path.

<table>
<thead>
<tr>
<th>Design</th>
<th>Area (mm²)</th>
<th>Energy/Instr (pJ)</th>
<th>Speed (MIPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Macho</td>
<td>0.055 (-23%)</td>
<td>32.8 (-9 %)</td>
<td>32.5 (+6 %)</td>
</tr>
</tbody>
</table>

*Measurement 3-1: Measured statistics for the first macho design.*

### 3.2 Balance of Selections

Code Fragment 2-7 chooses the appropriate instruction code based on the values of the
corresponding control bits. This translates into VLSI in a (straightforward) way that
selects the first alternative which guard evaluates to true. This implies, however, that
despite the fact that all control bits can evaluate simultaneously, the actual choice
between true guards begins with the first control bit and ends with the highest control bit
that evaluated to true. This (sequential) choice adds extra delay in our control path.

One way of reducing the average delay is to order the list of instructions such that
their expected execution is decreasing as a function of their place in the list.
Unfortunately, for this scheme to work best it requires the use of a histogram over all
applications that are to be run. Furthermore, most of its improvement is lost if the histogram lacks much concentration at a small subset of the entire instruction set.

Another approach, which promises to work much better because of its robustness, is to check individual bits that make up the instruction word one at a time. This divide-and-conquer strategy bases its remaining checks on the value of the current bit. This effectively drops the number of checks in our control path to the logarithm of the number of possible instructions.\(^8\) When using this technique, it is best to start checking those bits that divide the remaining possible instructions into two groups with approximately equal chances to occur. If information about this is lacking, we recommend using an order that contains the least number of total checks, since this expects to produce the least hardware, the least transitions and the most balanced tree of checks. The inventor of the STARC2 has constructed the instruction set architecture such that fewer bits identify frequently occurring instructions than less frequently occurring instructions. This lowers the average time needed to decode the instruction, much like the property of the declined approach in the previous paragraph.

```plaintext
DecodeInstr
FetchComputeStoreData =
| { if \textit{Instr}_{15} \rightarrow 
  if \textit{Instr}_{14} \rightarrow 
    if \textit{Instr}_{13} \rightarrow 
      if \textit{Instr}_{12} \rightarrow \text{ExecADD}() 
      \textit{Instr}_{12} \rightarrow \text{ExecADDC}() 
    \textit{fi} 
  \textit{fi} 
  \textit{Instr}_{13} \rightarrow \ldots 
  \textit{fi} 
  \textit{Instr}_{14} \rightarrow \ldots 
  \textit{fi} 
  \textit{Instr}_{15} \rightarrow 
    if \textit{Instr}_{14} \rightarrow 
      if \textit{Instr}_{13} \rightarrow \text{ExecJMP}() 
      \textit{Instr}_{13} \rightarrow \text{ExecCALL}() 
    \textit{fi} 
  \textit{fi} 
  \textit{Instr}_{14} \rightarrow \ldots 
 | }
```

*Code Fragment 3-4: Revised version for decoding of the instruction and executing it.*

---

\(^{8}\) This requires the instructions to be encoded efficiently without using much more bits than is absolutely necessary. Fortunately, this appears to be the case for the STARC2 instruction set architecture.
Code Fragment 3-4 shows the structure of a revised instruction executor using the bit-by-bit evaluation scheme discussed here.

In addition, one can collect and balance the conditional statements in procedures \textit{StoreRes} and \textit{FetchOpMem}, respectively. By collecting, we refer to the act of replacing several conditional statements (containing disjoint guards) by a single conditional statement.

\* \* \*

These optimisations result in the improvements shown in Measurement 3-2. The percentage figures within parentheses are a comparison with the results of the first macho transformation. Area decreases on parallel compositions and conditional statements. Energy decreases because of fewer transitions and speed increases because of a shorter control path (due to balancing).

<table>
<thead>
<tr>
<th>Design</th>
<th>Area (mm(^2))</th>
<th>Energy/Instr (pJ)</th>
<th>Speed (MIPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2\textsuperscript{nd} Macho</td>
<td>0.051 ((-7%))</td>
<td>28.2 ((-14%))</td>
<td>35.7 (+10 %)</td>
</tr>
</tbody>
</table>

\textit{Measurement 3-2: Measured statistics for the second macho design.}
4 Obtaining Low-Cost

An initial design for the STARC2 has been introduced in Chapter 2. In Chapter 3, it has been transformed into a better design with respect to area, power and speed. Naturally, not all possible transformations that are beneficial to at least one of the quantities are beneficial to all of them. In our opinion, the most powerful and interesting transformations are trading one penalty for another. The sections in this chapter are in line with this belief and stepwise reduce the area of the STARC2 considerably at the cost of both power and speed. All transformations targeting low area are based on reusing hardware in either control or data path. The first three transformations and the fifth transformation are the work of the author. The fourth transformation is not a new one.

4.1 Narrowing of Control Path

Instructions execute by taking the trace in the control path that corresponds to the instruction. Currently, in the decoding phase one control path is selected that is unique for the instruction. After this, the control paths temporarily reunite a number of times whenever they share a command that implements the instruction functionality. Finally, all control paths merge together to fetch and execute the next instruction. This indicates that the method bases on a redundancy of control. Another approach is to split up the control only if the alternatives do not share commands at all. This saves on a lot of unnecessary splitting and merging of control logic and therefore on area. We can obtain this by merging appropriate control paths together. Commands in the main control path only perform when the instruction needs to execute them. What remains is a single, narrow control path where successive commands either perform or skip.

First, we must rewrite the top level in such a way that the commands used by the internal interrupt controller are contained within the main control path. An observation with respect to the interrupt controller is that it implements a CALL instruction to the address specified by the interrupt channel. Therefore, a simple solution is to artificially create such an instruction whenever there is an interrupt, which procedure ExecInstr can then execute. Code Fragment 4-1 shows the revised version of the interrupt controller. It stores a CALL instruction word in Instr using the address received over ChanInt. In addition, we move the invocation of procedure ExecInstr outside the outer selection such that in case of an interrupt the created CALL instruction indeed executes. Note that instead of constructing an instruction word, we could also instruct procedure ExecInstr to execute this CALL functionality by directly altering the execution bits that perform the commands. However, the number of execution bits (over 40) exceeds the number of instruction word bits (equals 16) by far, which would introduce much more multiplexers. Since all commands other than for fetching instructions or communicating with external interrupts are now within the main control path, it is time to narrow it.
IntControl =
[[ ChanInt?IntVec;
    Instr := "CALL instruction to address IntVec" ]]

Code Fragment 4-1: The revised interrupt controller.

Starting with the fetch data phase, we discard all invocations to procedures FetchOpMem, FetchOpReg, FetchOpBit and StackPop. As a replacement, Code Fragment 4-2 introduces procedure FetchDataPhase that contains conditional invocations to them (and with respect to the stack pop only the RAM fetches are re-implemented). The newly introduced control bits evaluate to true when the respective operands are required in the remainder of procedure DecodeInstrFetchComputeStoreData.

FetchDataPhase =
[[ if CBFetchOpMem → FetchOpMem() fi ][
    if CBFetchOpReg → FetchOpReg() fi ][
    if CBFetchOpBit → FetchOpBit() fi ][
    if CBPopStack → OpMem := RAM[<0, SP>]; OpReg := RAM[<1, SP>] fi ]]

Code Fragment 4-2: A narrowed fetch data phase.

Continuing with the compute data phase, we discard all commands that implement functionality for this phase. As a replacement, Code Fragment 4-3 introduces procedure ComputeDataPhase that contains conditional statements re-implementing the discarded functionality.

ComputeDataPhase =
[[ if CBDestOpMem → ResAddr := OpMemAddr fi ][
    if CBDestOpReg → ResAddr := OpRegAddr fi ][
    if CBStackPush → SP := SP - 1 fi ][
    if CBStackPop → SP := SP + 1 fi ][
    if CBArithmetic → <<ResByte, Flags>> := p0×OpMem + p1×OpReg + p2×C + p3 fi ][
    if CBexecRLC → <<ResByte, C>> := <<C, OpMem>> fi ][
    if CBexecCLRb → ResByte := OpMem(OpMem9{10} ← false) fi ][
    if CBexecMOVc → if R40 → R0 := MSByte(ROM[<<R41..7, R5..5]])
        0 → R40 → R0 := LSByte(ROM[<<R41..7, R5..5>])
        fi
    fi ][
    ...
][

Code Fragment 4-3: A narrowed compute data phase.
Note that control bits CBArithmetic, CBStackPush and CBStackPop originate from the current transformation. Remember that the statement that is guarded by CBArithmetic originates from the first macho transformation.

Ending with the store data phase, we discard all invocations to procedures StoreRes and UpdateInd, the assignments to the program counter as well as stack elements. As a replacement, Code Fragment 4-4 introduces procedure StoreDataPhase that contains conditional statements re-implementing the discarded functionality. For branch instructions, boolean function EvalSkipFunc evaluates to true if the next instruction must be skipped.

```plaintext
StoreDataPhase =
| [ if CBStoreRes → StoreRes() fi ||
  if CBUpdateInd → UpdateInd() fi ||
  if CBGuardedSkip → if EvalSkipFunc → PC := PC + 1 fi fi ||
  if CBPushStack → RAM[<<0, SP>>] := LByte(<<PC, Flags>>);
    RAM[<<1, SP>>] := MByte(<<PC, Flags>>)
  fi ||
... ]
```

**Code Fragment 4-4: A narrowed store data phase.**

All commands within the original DecodeInstrFetchComputeStoreData and within its invocations are narrowed now. Code Fragment 4-5 shows a new procedure DecodeInstrFetchComputeStoreData that sequentially combines the three narrowed phases.

```plaintext
DecodeInstrFetchComputeStoreData =
| [ FetchDataPhase();
  ComputeDataPhase();
  StoreDataPhase() ]
```

**Code Fragment 4-5: A narrowed decode and execute procedure.**

** * * * **

Measurement 4-1 shows the results of the narrowing process. The percentage figures within parentheses are a comparison with the results of the second macho transformation. Unfortunately, this transformation does not decrease the area at all. It only worsens the energy and speed figures. The large amount of parallel compositions in the main control path explains the increased energy dissipation. The wideness of the parallel compositions, the fact that all guards are checked and the fixed traversal of the three sequential phases cause the speed decrease. Despite these drawbacks, this transformation is an important one due to the structure of the resulting design: all conditional assignments to the same variables are topologically close together. The next
transformation can use this property to join these assignments together, thereby reducing control considerably.

<table>
<thead>
<tr>
<th>Design</th>
<th>Area (mm²)</th>
<th>Energy/Instr (pJ)</th>
<th>Speed (MIPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Low-cost</td>
<td>0.051 (-0 %)</td>
<td>50.5 (+79 %)</td>
<td>28.5 (-20 %)</td>
</tr>
</tbody>
</table>

*Measurement 4-1: Measured statistics for the first low-cost design.*

### 4.2 Unification of Assignments

As stated before, the first low-cost transformation is not a particularly good one in isolation. Luckily, the resulting design promises a significant potential area decrease. This originates from the large amount of similar control logic located topologically close together in the form of conditional assignment to the same variables. This allows for extensive sharing of hardware for these (conditional) assignments on the level of data (roughly similar expressions within the assignments) and control (both the assignment logic and their surrounding parallel compositions). This transformation addresses the control part, while the next transformation mostly addresses the data part. Note that replacing multiple conditional assignments (to the same variable) by a single assignment containing a conditional expression (while keeping the same choice structure) effectively changes the former control bits into behaviour bits, as seen in the following paragraphs.

Starting with the fetch data phase, fetching the memory operand into OpMem and fetching the evenly aligned stack element part into OpMem can easily share hardware. Procedure `FetchOpMem` simply needs the capability to dereference the stack pointer. In addition, remove the assignments in procedure `FetchMemOp` to variable `OpMemAddr` from of the surrounding conditional statement and replace them by a single assignment to `OpMemAddr` using a conditional expression to select amongst the alternatives. Furthermore, procedure `DerefOpMemAddr` can combine its assignments to variable `OpMem` into a single assignment\(^9\).

Continuing with the compute data phase, each of the overflow flag, halfcarry flag, carry flag, result bit, result byte and result address combine their assignments to them into a single assignment for each of the variables.

Ending with the store data phase, each of the registers, remainder of the program status word, stack pointer, data bit pointer parts and program counter combine their assignments to them into a single assignment for each of the variables. In addition,

---

\(^9\) This example illustrates the fact that following one out of two closely resembling definitions can have significant impact on output, as can be seen in the measurements of this transformation. These measurements are entirely the result of the rewrite strategy suggested by this.

\(^{10}\) The Tangram programming language does not allow commands in its expressions. Therefore, the commands implementing the protocol involving the special function register bus keep their own assignment to OpMem. In addition, the current Tangram implementation performs memory accesses within a conditional expression unconditionally (but only uses the accessed value when necessary). To avoid unnecessary accesses we keep them in a separate assignment as well. Since these irregularities do not contribute to the understanding of the proposed transformations, we abstract from these Tangram specific implementation implications in the rest of this document.
storing the result value into memory and storing the evenly aligned stack element part into memory can easily share hardware. Procedure StoreRes simply needs the capability to dereference the stack pointer.

* * *

Measurement 4-2 shows the results of the unification process. The percentage figures within parentheses are a comparison with the results of the first low-cost transformation. The removal of Area decreases due to conditional statements, assignments and parallel operators. Energy decreases due to fewer transitions in the control path. Unfortunately, speed also decreases considerably, since the delays for the remaining assignments are worst-case over all possible alternatives within the conditional expressions\(^\text{11}\).

<table>
<thead>
<tr>
<th>Design</th>
<th>Area (mm(^2))</th>
<th>Energy/Instr (pJ)</th>
<th>Speed (MIPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2(^{nd}) Low-cost</td>
<td>0.035 (−31 %)</td>
<td>48.4 (−4 %)</td>
<td>18.8 (−34 %)</td>
</tr>
</tbody>
</table>

*Measurement 4-2: Measured statistics for the second low-cost design.*

4.3 Squeezing Exploitation of Similarity

As stated before, this transformation mainly aims at reducing logic within expressions in general and conditional expressions in particular. Since a macho transformation already performed the modifications that are beneficial to all measured quantities, this transformation aims at squeezing the remaining logic redundancy out of the design, at the penalty of energy and/or speed.

First, we present an abstract method for systematically decreasing the area for implementing conditional expressions. It is generally applicable to the whole class of expressions where one chooses a result out of a number of alternative functions, which have an identity element. It bases on the fact that a multiplexer choosing between two non-fixed values (called full multiplexer) is approximately twice as expensive as a multiplexer choosing between one non-fixed value and one fixed value\(^\text{12}\) (called half multiplexer). Assume an expression \(E\) consisting of a multiplexer that, using behaviour bit \(B\), selects amongst two alternatives: a function \(F\) over parameters \(X\) and \(Y\), having identity element \(I\), and arbitrary expression \(G\) over parameter \(Y\).

\[
E = \text{if } B \rightarrow F(X, Y) \triangleright -B \rightarrow G(Y) \text{ fi}
\]

Expression \(E\) is equivalent to \(E'\) whenever \(G\) equals \(Y\) if \(B\) equals true (called the function chaining lemma, refer to Appendix B for a proof).

\[
E' = F(\text{ if } B \rightarrow X \triangleright -B \rightarrow I \text{ fi}, G(Y))
\]

\(^{11}\) In Tangram, each expression is associated with only one (fixed and worst-case) delay, even if it consists of a conditional expression. A conditional statement containing several assignments, on the other hand, is associated with a dedicated (fixed and worst-case) delay for each of its alternatives.

\(^{12}\) To get into the low-level details: for every bit in the representation of the values, a NOR gate and an inverter gate can implement the bit part of the latter multiplexer. For these inverter gates hold that they optimise away approximately half of the time.
Taking a closer look, $E$ contains a full multiplexer, while $E'$ contains a half multiplexer. Another observation is that if all function alternatives within a conditional expression operate on the same parameters $X$ and $Y$, one can recursively apply the function chaining lemma to rewrite a chain of full multiplexers into a chain of half multiplexers. The final $G$ in the chain then corresponds to $Y$, fulfilling its requirement.

Code Fragment 4-6 shows an expression for the result value based on this method. For sake of simplicity, we have ignored the generation of flags. Functions $Add$, $Xor$, $And$ and $Or$ produce the results that intuitively can be expected. The behaviour bits denoted by similar names indicate whether the corresponding function should evaluate. Functions without identity elements should evaluate in the alternative denoted by the ellipsis. Functions $Left$ and $Right$ return the values of $OpMem$ and $OpReg$ if the destination operand is the memory operand, respectively; otherwise, $Left$ and $Right$ swap their values. When functions $Add$ and $Xor$ are the only functions ‘turned on’, they perform the subtraction functionality. When all functions are ‘turned off’, they perform the MOV instruction functionality. If we extend the functionality of function $Right$ to be able to return the number 1, we can also implement the increment and decrement functionalities with it.

\[
\begin{align*}
\text{ResByte} := & \text{if BBIdentityFunctions } \rightarrow \\
& \text{Add( if BBAdd } \rightarrow \text{ Left } 0 \text{ } \neg\text{BBAdd } \rightarrow 0 \text{ fi,} \\
& \text{Xor( if BBXor } \rightarrow \text{ Left } 0 \text{ } \neg\text{BBXor } \rightarrow 0 \text{ fi,} \\
& \text{And( if BBAnd } \rightarrow \text{ Left } 0 \text{ } \neg\text{BBAnd } \rightarrow 255 \text{ fi,} \\
& \text{Or( if BBOr } \rightarrow \text{ Left } 0 \text{ } \neg\text{BBOr } \rightarrow 0 \text{ fi,} \\
& \text{Right)))) \\
& 0 \text{ } \neg\text{BBIdentityFunctions } \rightarrow \ldots \\
& \text{fi}
\end{align*}
\]

*Code Fragment 4-6: Recursive application of the function chaining lemma on the result value expression. The ellipsis denotes the expression that handles all functions that had no identity element and could neither be composed from its presented functions.*

Second, the increments and decrements involved in the auto addressing mode can extend to update the stack pointer as well. In general, the costs for sharing increment hardware (i.e. one multiplexer for each bit) are bigger than the increment hardware itself (the single full adder and one half adder for each other bit). A solution to this is to make sure that all pointer values must reside in the same variable before the increment or decrement, so that there is no need for the extra multiplexers. To fulfil this property, $MemOpAddr$ obtains the value of the stack pointer within the fetch data phase in case of a stack instruction. However, this requires a conditional expression and therefore introduces multiplexers too. Fortunately, then the stack pointer does not have assignments anymore that depend on its own value (due to the duplicate information),
which allows for a cheaper implementation of the stack pointer as a variable\textsuperscript{13}. In addition, the requirement that a single variable contains all pointers used during the instruction execution is also needed in the transformation of Section 4.5, a knowledge that rather diminishes the overhead.

Finally, there are two opportunities where relatively big expressions resemble each other enough to share their hardware effectively. The first concerns the command in the fetch instruction phase that increments the program counter by one, which can easily share the hardware of the command in the store data phase for implementing branch functionality (since this also includes a program counter increment). The only thing to do is putting the corresponding assignment in a procedure and giving it a boolean parameter indicating whether or not it must enforce the increment instead of the usual behaviour (indicated by its behaviour bits). The second concerns the commands that fetch the register operand from the register file and fetch a stack element part from user RAM, which again can easily share the hardware for fetching the memory operand. The only thing to do is putting the corresponding commands in a procedure and giving it some parameters: one for indicating the result variable (either OpMem or OpReg) and one for each execution bit used within the procedure (either directing the memory operand fetch, the register operand fetch or the partial stack pop). Consequently, the memory operand fetch must (artificially) be sequentialised with the register operand fetch and the partial stack pop (the latter two never occur within the same instruction).

\textbf{***}

Measurement 4-3 shows the results of this transformation. The percentage figures within parentheses are a comparison with the results of the second low-cost transformation. Area decreases due to the reuse of expressions (this is disappointingly little because only local optimisations are performed instead of structural ones). Energy increases slightly due to the extra transitioning of the (big) fetching procedure for OpReg. Unfortunately, speed decreases over-proportionally due to the sequentialisation of the two fetches.

<table>
<thead>
<tr>
<th>Design</th>
<th>Area (mm(^2))</th>
<th>Energy/Instr (pJ)</th>
<th>Speed (MIIPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3\textsuperscript{rd} Low-cost</td>
<td>0.033 (−6 %)</td>
<td>50.0 (+3 %)</td>
<td>15.1 (−20 %)</td>
</tr>
</tbody>
</table>

\textit{Measurement 4-3: Measured statistics for the third low-cost design.}

4.4 Centralisation of Execution Bit Evaluation

By evaluating all execution bits simultaneously, we expect to save area on the large amount of delay matches corresponding to them. Since the value of the instruction word does not change during the period of time that the execution bits are used, the evaluated values for the execution bits need not be stored in variables. Instead, we can keep them

\textsuperscript{13} Latches are smaller than flip-flops (in terms of VLSI area). However, latches (in contrast to flip-flops) do not allow for so-called auto-assignments where an assignment to a variable consists of an expression that depends that same variable.
stable on wires (which can be designed in Tangram as read-only parameters in a newly to be introduced procedure UseExecutionBits).

** **

Measurement 4-4 shows the results of this transformation. The percentage figures within parentheses are a comparison with the results of the third low-cost transformation. Area decreases slightly, since the execution bits can share a single (worst case) delay matcher in the control path. Energy decreases relatively twice as much than area, since Tangram’s handshake protocol causes a definite rise and a definite fall in the control path (from which the entire area decrease comes). Speed increases because execution bits evaluate simultaneously and only once per instruction word, instead of introducing extra delay every time they are used. Apparently, this transformation is, at least with the previous design as starting point, a true macho transformation instead of a low-cost transformation.

<table>
<thead>
<tr>
<th>Design</th>
<th>Area (mm²)</th>
<th>Energy/Instr (pJ)</th>
<th>Speed (MIPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4th Low-cost</td>
<td>0.032 (−3 %)</td>
<td>47.0 (−6 %)</td>
<td>16.2 (+7 %)</td>
</tr>
</tbody>
</table>

*Measurement 4-4: Measured statistics for the fourth low-cost design.*

### 4.5 Departure of Internal State

In the final transformation, we aim at minimising the amount of area allocated for variables in the STARC2. One can observe that the STARC2 does not use all variables throughout the whole execution of instructions. Therefore, we wish to get rid of the variables whose value can also most of the time reside somewhere else. It seems that only those variables are essential that simultaneously require presence (within an expression or as destination of an assignment). Concerning the STARC2, worst-case we need variables that keep track of the program counter, the value of the memory operand, the value of the register operand, the address of the memory operand, the data bit pointer bit part and the flags. One can easily check this by following the control path for each possible trace (i.e. instruction word). Apparently, we can put the values of R0 through R7, SP, DBPByte, BusAddress, BusData, ResAddr, ResByte and ResBit outside of the STARC2 core. We choose to store them in the user RAM, where space is much cheaper, at their logical addresses within the address space. This way we do not only save hardware on variables, but also on referencing and dereferencing logic (we do not need to pick out certain addresses because their corresponding variable values reside elsewhere in the address space). It does not even cost more RAM space, if one can only produce entirely filled blocks of RAM cheaply. We then reuse variables OpMemAddr and OpMem as general pointer (replacing R0 through R3, SP, DBPByte, BusAddress and ResAddr) and general value (replacing R0 through R7, BusData, ResByte and ResBit), respectively. The auxiliary memory accesses introduced by this transformation causes additional sequencing in the both fetch data and store data phases. Code Fragment 4-7 shows the sequential nature of the design obtained.
DecodeInstrFetchComputeStoreData =
  || "Fetch Memory Address (_HIT)"
  || "Fetch Register Operand (REG)"
  || "Fetch Memory Operand (MEM)
  || "Update Memory Address (MAR)
  || "Compute Result Address" || "Compute Result" || "Store Stack Element (STK)"
  || "Store Result/Stack Element (STK)" || "Update PC" || "Update DBP Bit"

Code Fragment 4-7: Sequentialised structure of instruction decoding and execution.
A tape symbol indicates that the command possibly accesses memory. Despite six
tape symbols, at most five can actually occur during instruction execution.

**

Measurement 4-5 shows the results of this transformation. The percentage figures within
parentheses are a comparison with the results of the fourth low-cost transformation. Area
decreases considerably, roughly equally due to fewer variables as to logic that used the
former variables. Energy concerning the STAR C2 decreased due to fewer transitions. Note that energy concerning RAM increased considerably due to auxiliary accesses to it
(accessing roughly six times as much than before). Speed decreases due to the
sequentialised pointer accesses that can occur.

<table>
<thead>
<tr>
<th>Design</th>
<th>Area (mm²)</th>
<th>Energy/Instr (pJ)</th>
<th>Speed (MIPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5th Low-cost</td>
<td>0.024 (-25 %)</td>
<td>43.3 (-8 %)</td>
<td>15.4 (-5 %)</td>
</tr>
</tbody>
</table>

*Measurement 4-5: Measured statistics for the fifth low-cost design.*
5 Obtaining High-Performance

An initial design for the STARC2 has been introduced in Chapter 2. In Chapter 3, it has been transformed into a truly better design with respect to area, power and speed. Chapter 4 transformed this macho design into a low-cost design, at the penalty of both power and speed. According to the problem description stated in Section 1.2, this chapter must transform the macho design into a high-performance design, expectantly at the penalty of both area and power.

Due to certain events in time (caused by the STARC version upgrade mentioned in Section 1.4), we have chosen long before the writing of this document to deviate slightly from the problem description during the research period. The difference boils down to transforming the low-cost design (from Chapter 4) into the high-performance design, instead of the starting with the macho design (from Chapter 3). As it turns out, this has disappointing, yet interesting, implications on the resulting design (in terms of its measurements). These implications are discussed in the conclusions of Chapter 6. Section 5.2 briefly describes the transformation.

During the writing of this document we have decided to (briefly) redo the transformation, but this time on the actual macho design in order to make partial amends for the poor results of the transformation of Section 5.2. Section 5.1 describes this transformation, which turns out to be much more successful. We present the successful transformation first, since we wish to decouple the so-called concept of pipelining from the failed attempt (as it is not the reason for failure at all). Both transformations are not new; they are applications of the well-known principle of pipelining. For more information about pipelining, refer to [1].

5.1 Prefetch Pipeline Stage – Successful

As stated before, time pressure has forced us to perform only one well-known transformation. It is the probably the simplest form of a technique called pipelining. The basic idea behind it is that we can partition instruction execution into a number of pipeline stages, similar to the concept of instruction phases. However, these phases execute sequentially, while the strength of pipelining comes from executing pipeline stages in parallel. Naturally, we cannot have stages simultaneously executing the same instruction due to the dependencies on subsequent results. Fortunately, this does not hold us back from having stages simultaneously executing subsequent instructions.

Here, we partition the pipeline as follows: we take the fetch phase as one stage (the prefetch stage) and the remaining phases as the second stage (the execute stage). We consider this the simplest partitioning due to its potentially simple interface in between the two stages. In order to not have both stages access to the code memory (which would require a proper resource sharing protocol), as an exception to the proposed partitioning we move the functionality of the MOVC instruction to the prefetch phase. Taking this exception in consideration, we obtain the following interface: the prefetch stage communicates instruction words to the execute stage (indicating what should be done) and the address of the successive instruction in code memory (in case of a call instruction
this information needs to be pushed on the stack), while the execute stage communicates when to skip the next instruction and when to jump to a specific address (both in case of a branch instruction). The two stages communicate through channels. Code Fragment 5-1 shows the prefetch stage details. The execute stage is obtained from the existing code fragments using the following modifications: the body of procedure FetchInstr is replaced by receiving the instruction word and a copy of the program counter, each assignment of jump address to the program counter is replaced by a send over channel ChanJump, each conditional program counter increment is replaced by a send over channel ChanSkip, and replace the invocation to procedure ExecMove by a skip command.

```
PrefetchStage =

    |[ forever do
      
      Instr := ROM(PC);  
      PC := PC + 1;

      forceif probe(ChanFetchInstr)  →
        ChanFetchInstr!<<Instr, PC>>  || if CBMovec → ExecMovec() fi

        ² probe(ChanJump)  → ChanJump?PC
        ² probe(ChanSkip)  → ChanSkip?-

    od ]|
```

* * *

Measurement 5-1 shows the results of this transformation\(^{14}\). The relative differences are a comparison with the results of the second macho transformation. Area increases slightly due to both the extra control and the communication between the two pipeline stages. Energy increases mostly because whenever the STARC2 must skip or jump over the next instruction, it has still fetched it (because this is done ‘ahead of time’). Note that there are more accesses to code memory because of this, therefore spending more energy in that memory as well (in our test bench the number of instruction fetches increased by 30 percent). Speed increases because the execute stage does not have to wait for the instruction fetch and program counter increment, since this is already done (in parallel) during the previous instruction execution.

\(^{14}\) The presented numbers approximate measurement results, rather than precise measurement results. The reason for this is that a bug in the peephole optimisation tool that generates a malfunctioning netlist, which does not simulate our test bench correctly. Since the netlist generated by the handshake compiler does simulate correctly, we have adjusted these measurement results proportionally to the effects of the peephole optimisations when performed on the macho STARC2 design.
<table>
<thead>
<tr>
<th>Design</th>
<th>Area (mm²)</th>
<th>Energy/Instr (pJ)</th>
<th>Speed (MIPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Succ. High-performance</td>
<td>0.053 (+4 %)</td>
<td>33.5 (+19 %)</td>
<td>40.7 (+14 %)</td>
</tr>
</tbody>
</table>

*Measurement 5-1: Measured statistics for the successful high-performance design.*

## 5.2 Prefetch Pipeline Stage – Failed

We have performed similar modifications as discussed in Section 5.1 to the low-cost design. However, the resulting speed is only half of the speed of the macho design. An advantage is that the resulting area is also only half of the area of the macho design.

** Measurement 5-1 shows the results of this transformation. The relative differences are a comparison with the results of the fifth low-cost transformation. Area increases due to both the extra control and the communication between the two pipeline stages. Energy increases mostly because whenever the STARC2 must skip or jump over the next instruction, it has still fetched it (because this is done ‘ahead of time’). Note that there are more accesses to code memory because of this, therefore spending more energy in that memory as well. Speed increases because the execute stage does not have to wait for the instruction fetch and program counter increment, since this is already done (in parallel) during the previous instruction execution.**

<table>
<thead>
<tr>
<th>Design</th>
<th>Area (mm²)</th>
<th>Energy/Instr (pJ)</th>
<th>Speed (MIPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail. High-performance</td>
<td>0.026 (+8 %)</td>
<td>49.7 (+15 %)</td>
<td>17.9 (+16 %)</td>
</tr>
</tbody>
</table>

*Measurement 5-2: Measured statistics for the failed high-performance design.*

---

15 The presented numbers are (again) approximate measurement results, rather than precise measurement results. Probably the same bug in the peephole optimisation tool as mentioned in Section 5.1 keeps us from properly measuring.
6 Conclusions

Section 6.1 discusses the results of the transformations presented in previous chapters. Section 6.2 discusses the advantages and disadvantages of synchronous technology versus asynchronous technology with respect to our work.

6.1 Concerning Transformations

Let us forget about power in this section; we have not actively aimed at improving this quantity anyway. We start with an overview of the measurement results, as shown in Figure 6-1.

![Graphical overview of STAR22 designs in terms of their area (in mm², horizontal) and speed (in MIPS, vertical). LC stands for Low-Cost, HP stands for High-Performance. Moving to the left implies low-cost optimisations, moving to the top implies high-performance optimisations, and moving to the upper-left implies macho optimisations.](image)

*Figure 6-1: Graphical overview of STAR22 designs in terms of their area (in mm², horizontal) and speed (in MIPS, vertical). LC stands for Low-Cost, HP stands for High-Performance. Moving to the left implies low-cost optimisations, moving to the top implies high-performance optimisations, and moving to the upper-left implies macho optimisations.*
This figure allows for a number of observations. The first high-performance transformation resulted without a doubt in the fastest STARC2, performing a third faster than the initial design. The fifth low-cost transformation resulted in the smallest STARC2, occupying only one third the area of the initial design. The second through the fifth low-cost transformation, as well as the second high-performance transformation cluster closely together. For most transformations it can be observed, that improving one quantity worsens another quantity. It is interesting to determine which transformations resulted in the better trade-offs, i.e. which designs gained more than they lost in terms of square millimetres and million instructions per second. For this purpose, we have compiled a table that is sorted on the value of speed figures divided by area figures, as shown in Table 6-1. Note that this value equals the slope of the line through the origin point and the corresponding design point in Figure 6-1. In addition, note that we have included one of the world’s most famous micro-processors in the table to compare our STARC2 designs with\textsuperscript{16}.

<table>
<thead>
<tr>
<th>Design</th>
<th>Speed/Area (MIPS/mm\textsuperscript{2})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1\textsuperscript{st} High-performance</td>
<td>768</td>
</tr>
<tr>
<td>2\textsuperscript{nd} Macho</td>
<td>700</td>
</tr>
<tr>
<td>2\textsuperscript{nd} High-performance</td>
<td>688</td>
</tr>
<tr>
<td>5\textsuperscript{th} Low-cost</td>
<td>642</td>
</tr>
<tr>
<td>1\textsuperscript{st} Macho</td>
<td>590</td>
</tr>
<tr>
<td>1\textsuperscript{st} Low-cost</td>
<td>559</td>
</tr>
<tr>
<td>2\textsuperscript{nd} Low-cost</td>
<td>537</td>
</tr>
<tr>
<td>4\textsuperscript{th} Low-cost</td>
<td>506</td>
</tr>
<tr>
<td>3\textsuperscript{rd} Low-cost</td>
<td>458</td>
</tr>
<tr>
<td>Initial</td>
<td>434</td>
</tr>
<tr>
<td>Intel Pentium 4</td>
<td>9.22</td>
</tr>
</tbody>
</table>

\textit{Table 6-1: Sorted table indicating which designs have resulted in the best trade-offs in terms of speed divided by area.}

Evidently, the first (and successful) high-performance design triumphs at the first row of the table. Interestingly, it is the only transformation that performs more efficiently (considering its amount of logic) than the second macho design, which is (possibly indirectly) the starting point of most other designs. These designs either increase their

\textsuperscript{16} The comparison with Intel's Pentium 4 (running at roughly 2000 MIPS and taking up 217 mm\textsuperscript{2}) is somewhat difficult. Their technology process and layout process differ completely. Nevertheless, Intel has obtained their very high speed at a huge area penalty, rendering their processor far less efficient than even our least efficient STARC2 design.
area more than that they increase their speed or they decrease their speed more than that they decrease their area (all expressed relatively). Fortunately, the initial design appears to be the least efficient design, which implies that no design introduced more inefficiency than there was from the start. Finally, observe that the second high-performance transformation increases performance efficiency despite its disappointing absolute speed.

Let us compare the difference in speed between the second low-cost design and the second macho design (which mostly differs at the presence of much more conditional behaviour) to the difference in speed between the second macho design and the first high-performance design (which only differs by the presence of pipelining). Apparently, adding conditional behaviour increases the speed more than adding a pipeline stage (both absolutely and relatively). However, the concept of pipelining is known for its feasibility of applying extra pipeline stages recursively (increasing the speed until at least seven or eight stages are present). On the contrary, the concept of conditional behaviour (even at the extreme) probably cannot be applied that much to reduce the average delay by a competing factor. A hybrid form, consisting of several pipeline stages in which as much is done conditionally might provide an optimal performance. However, it might become a problem to decouple the individual delays from individual stages, which would render the conditioning useless.

Let us look at the third low-cost design. The area decrease is low despite the promising method based on the function chaining lemma. When applying this lemma we can observe that, next to becoming smaller, the conditional expression also becomes deeper. One can see this as sequencing the data path, effectively reusing part of the logic implementing the function where its operands meet. However, Tangram has the tendency to increase the amount of corresponding control logic because more delay matching must be performed. Furthermore, STARC2’s relative thin data path (eight bits wide) only offers eight multiplexers for every function. If we applied this method on a 32 bits machine, we expect the gain to be much higher.

Let us look at the fifth low-cost design. The area decreases considerably here. An observation there says that roughly half the area loss is accounted for by addressing logic. The STARC2 allowed largely for this amount because its data memory space included available addresses for the removed variables. Therefore, no redirection of fetch and store actions on these variables was necessary to implement the removal of the variables correctly. Micro-controllers that, unlike the STARC2, have no addresses of its core variables in its standard memory space are unlikely to obtain the same results for the fifth low-cost transformation.

The 80C51 micro-controller, which is similar to the STARC2 in many aspects, has also been designed using the Tangram language, as discussed in [6]. There, emphasis is on low-power. That design differs drastically from our STARC2 designs with respect to the control part. The control path is split up according to certain groups of similar instructions at certain places of the instruction execution. This way, only the parts of the resulting hardware that actually contribute to the functionality of the current instruction are active. This reduces energy consumption considerably, which was the intention there. However, it introduces extra control hardware, which renders the method unsuitable for our low-cost designs. On the other hand, the property of (extensive) conditional execution of commands can benefit the speed of our high-performance designs.
A remark about the transformations presented in this document is that due to time pressure we were unable to perform as much high-performance optimisations as we could for low-cost. This fact alone should not imply that the STARC2 is inherently better suited for low-cost than for high-performance. In fact, there are still numerous possibilities to explore much faster designs such as more pipelining, more conditioning of commands, more command-level parallelism, simultaneous instruction words fetching, etc.

Besides our customer, Philips Research also intends to keep four designs in their library of IP (Intellectual Property) blocks. The initial design and the second macho design are kept to be able to create STARC2 designs based on entirely different requirements than low-cost or high-performance (think for example of low-power and low-EMI). In addition, the fifth low-cost design and the first high-performance design are kept to have these optimised STARC2's readily available.

6.2 Concerning Asynchronous Technology

This section discusses the advantages and disadvantages of synchronous technology versus asynchronous technology with respect to our work.

First, some remarks about the programming language Tangram. The second low-cost transformation was able to decrease area considerably by moving control logic into the data path. Apparently, Tangram's current generation of control logic is far from optimal. This is mostly explained by the fact that Tangram aims at low-power applications; low-cost or high-performance aspects are not taken to the extreme in the generation of netlists. For example, Tangram's control logic area grows approximately linearly with the amount of commands to be performed, and it does not consider the fact that an expression may already be stable when it is to be evaluated, but instead waits for the worst-case delay of the entire combinational logic implementing the expression. However, a respectable feature of the language is that it allows for both high-level (e.g. sequencing and repetition constructs) and low-level (e.g. bit manipulation operators) to occur in the same design, compile easily to netlists and quickly provide measured quantities. This offers the capability for designers to quickly explore the design space: for optimisation ideas on both the large-scale architectural level as well as the small-scale bit level.

Let us take a closer look at implications of pipelining asynchronously. In a multiple staged pipeline, the phenomenon of data dependency quickly plays part, which means that certain stages must stall their execution until the required input data is present. This causes an amount of overhead when implementing this with synchronous technology. Asynchronously, this can be easily solved by taking advantage of the blocking property of (handshake) channels: they automatically halt execution until data is available. On the other hand, so-called bypasses are usually introduced when required data is present, but not yet stored in the proper location (e.g. memory or register file). These bypasses take the data from the specific pipeline stage where it is currently present and pass it to the place where it is required. A problem arises when different stages are not fully synchronised, which is probably the case when using asynchronous technology. Then, it is difficult to know when the data can safely be read. Two solutions present
themselves: artificially synchronise those stages that are connected by such a bypass network (this undoes any speed advantage of variable delays) or duplicate the data that must be read by the bypasses to allow them a more suitable lifespan (this costs extra area).

Concerning the differences in advantage of synchronous technology over asynchronous technology for the presented transformations, we can start by pointing out the fact that the method behind function chaining promises to be more effective, as stated in Section 6.1. Optimisations resulting in more sharing of logic in the data path are expected to have the same impact because both synchronous and asynchronous technologies assign the same delays to them. Conditional execution of commands, causing variable delays in asynchronous technology, does not improve speed in synchronous technology, since only worst-case delays are taken into account (due to the constant clock frequency). In particular, the clock frequency is severely lowered when moving most of the STARC2 state into the data memory (the fifth low-cost transformation, assuming a constant number of clock cycles per instruction). Finally, transformations resulting in less control logic are not very efficient using synchronous technology, since the amount of control logic is only a fraction of the amount of control logic generated by Tangram.
7 Future Work

In this chapter we indicate some loose ends of our research, that may be interesting to investigate upon in the future for both us as well as the reader.

First, no time has been spent on actively aiming at low-power transformation. Measurements indicate that some of the transformations presented in this document are beneficial to the energy consumption, but this topic is far from covered in this document. In addition, there is potential for obtaining much more high-performance than currently achieved. It might be interesting to see whether extreme pipelining (i.e. introducing more than ten stages), which is difficult to obtain synchronously, is inherently easier or even much more difficult to manage. An investigation on how much speed one can gain by conditionally executing commands to the extreme (e.g. by splitting commands into subcommands or by introducing more hierarchy in the conditional control path, effectively only checking for certain conditions conditionally) might be fruitful. The aspect of high-performance in the context of asynchronous technology strikes many as an inherent paradox (mostly due to the large amount of control overhead). The author of this document does not share this vision, and one of his emerging research interests is to prove feasibility. As can be concluded from this chapter, many topics remain open for investigation. Therefore, we thank you for your reading time and let us get started!
8 References


5. *STARC2 Data Sheet Core*, 2000. (Proprietary)

## A Compilation of Initial STARC2 State

Following is a compilation of the constants, variables and channels used throughout the initial design along with their bit widths and description.

<table>
<thead>
<tr>
<th>Name</th>
<th>Bit Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusAccess</td>
<td>0</td>
<td>This specification channel represents actual access over the special function registers bus.</td>
</tr>
<tr>
<td>BusAddress</td>
<td>8</td>
<td>This specification variable represents the address communicated over the special function registers bus.</td>
</tr>
<tr>
<td>BusData</td>
<td>8</td>
<td>This specification variable represents the data communicated over the special function registers bus.</td>
</tr>
<tr>
<td>BusRMW</td>
<td>1</td>
<td>This specification variable represents the read-modify-write predicate for the special function registers bus.</td>
</tr>
<tr>
<td>BusWrite</td>
<td>1</td>
<td>This specification variable represents the write predicate for the special function registers bus.</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>This specification variable represents the carry flag.</td>
</tr>
<tr>
<td>ChanInt</td>
<td>3</td>
<td>This specification channel represents the source for receiving the interrupt vectors. The accompanying handshake represents the interrupt request and acknowledge.</td>
</tr>
<tr>
<td>DBPBit</td>
<td>3</td>
<td>This specification variable represents the bit index part of the data bit pointer.</td>
</tr>
<tr>
<td>DBPByte</td>
<td>8</td>
<td>This specification variable represents the byte part of the data bit pointer.</td>
</tr>
<tr>
<td>DBPMap</td>
<td>1</td>
<td>This specification variable represents the map part of the data bit pointer.</td>
</tr>
<tr>
<td>HC</td>
<td>1</td>
<td>This specification variable represents the halfcarry flag.</td>
</tr>
<tr>
<td>Instr</td>
<td>16</td>
<td>This interface variable serves the purpose of containing the current instruction word.</td>
</tr>
<tr>
<td>OpBit</td>
<td>3</td>
<td>This interface variable serves the purpose of containing the bit index (within the memory operand) of the bit operand.</td>
</tr>
<tr>
<td>Variable</td>
<td>Size</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>OpMem</strong></td>
<td>8</td>
<td>This interface variable serves the purpose of containing the memory operand.</td>
</tr>
<tr>
<td><strong>OpMemAddr</strong></td>
<td>8</td>
<td>This interface variable serves the purpose of containing the address of the memory operand.</td>
</tr>
<tr>
<td><strong>OpReg</strong></td>
<td>8</td>
<td>This interface variable serves the purpose of containing the value of the register operand.</td>
</tr>
<tr>
<td><strong>OV</strong></td>
<td>1</td>
<td>This specification variable represents the overflow flag.</td>
</tr>
<tr>
<td><strong>PC</strong></td>
<td>13</td>
<td>This specification variable represents the program counter.</td>
</tr>
<tr>
<td><strong>R0, ..., R7</strong></td>
<td>8</td>
<td>These specification variables represent the registers zero through seven.</td>
</tr>
<tr>
<td><strong>ResAddr</strong></td>
<td>8</td>
<td>This interface variable serves the purpose of containing a computed integer result value.</td>
</tr>
<tr>
<td><strong>ResBit</strong></td>
<td>1</td>
<td>This interface variable serves the purpose of containing the computed bit result.</td>
</tr>
<tr>
<td><strong>ResByte</strong></td>
<td>8</td>
<td>This interface variable serves the purpose of containing the computed byte result.</td>
</tr>
<tr>
<td><strong>RestPSW</strong></td>
<td>5</td>
<td>This specification variable represents the five bits in the program status word without special function.</td>
</tr>
<tr>
<td><strong>SP</strong></td>
<td>7</td>
<td>This specification variable represents the stack pointer.</td>
</tr>
<tr>
<td><strong>StartAddrSFR</strong></td>
<td>8</td>
<td>This constant specifies the lower (inclusive) address boundary of the special functions registers space.</td>
</tr>
<tr>
<td><strong>StartAddrUserRAM</strong></td>
<td>8</td>
<td>This constant specifies the lower (inclusive) address boundary of the user RAM space, as well as the upper (exclusive) address boundary of the special functions registers space.</td>
</tr>
<tr>
<td><strong>EndAddrUserRAM</strong></td>
<td>8</td>
<td>This constant specifies the lower (inclusive) address boundary of the user RAM space, as well as the upper (exclusive) address boundary of the special functions registers space.</td>
</tr>
</tbody>
</table>
B Proof of Function Chaining Lemma

Function Chaining Lemma

Assume an expression $E$ consisting of a multiplexer that, using behaviour bit $B$, selects among two alternatives: a function $F$ over parameters $X$ and $Y$, having identity element $I$, and arbitrary expression $G$ over at least $Y$.

$$E = \text{if } B \rightarrow F(X, Y) \oplus \neg B \rightarrow G(Y) \text{ fi}$$

Expression $E$ is equivalent to $E'$ whenever $G$ equals $Y$ if $B$ equals true.

$$E' = F(\text{ if } B \rightarrow X \oplus \neg B \rightarrow I \text{ fi}, G(Y))$$

Proof

$$E =$$

{ definition of $E$ }

$$\text{if } B \rightarrow F(X, Y) \oplus \neg B \rightarrow G \text{ fi}$$

= \{ $B \Rightarrow G = Y$ \}

$$\text{if } B \rightarrow F(X, G) \oplus \neg B \rightarrow G \text{ fi}$$

= \{ $F(I, \alpha) = \alpha$ \}

$$\text{if } B \rightarrow F(X, G) \oplus \neg B \rightarrow F(I, G) \text{ fi}$$

= \{ distribution of $F$ over multiplexer \}

$$F(\text{ if } B \rightarrow X \oplus \neg B \rightarrow I \text{ fi}, G)$$

= \{ definition of $E'$ \}

$$E'$$