Bernhard Vacarescu

Implementation of an 8-bit Microprocessor Architecture on an FPGA using VHDL and Timing Efficiency Analysis

BACHELORARBEIT

zur Erlangung des akademischen Grades

BSc

in Elektrotechnik

eingereicht an der

Technischen Universität Graz

am Institut für Technische Informatik

Betreuer:

Ass.Prof. Dott. Dott. mag. Dr.techn. Carlo Alberto Boano, MSc

Dipl.-Ing. Michael Spörk, BSc

Februar 2018, Graz

Abstract

The teaching material at universities often seems to be quite abstract as it is not referring to actual existing components. At the course "Technische Informatik I" here at Graz University of Technology, the concept of an 8-bit CISC microprocessor architecture is used to teach how a processor operates. However, the material is only a theoretical architecture that has not been tested in practice. The goal of this thesis is to take this concept and describe the architecture using the hardware description language VHDL in order to make it possible to implement the architecture on a Field Programmable Gate Array (FPGA). Furthermore, the functionality of the processor can be successfully verified by simulation and we show that the design is working properly on a Xilinx Artix®-7 FPGA. Our work enables students to use this architecture to get a better understanding about the operation of a processor, including features like microprogramming and interrupt handling. Students can hence make use of a real working hardware equivalent to the course material and get acquainted with it during the "Technische Informatik Labor" course.

Table of Contents

List of Illustrations

List of Tables

List of Abbreviations

AC Accumulator

ALU Arithmetic Logic Unit

AMux ALU Multiplexer

CC Condition-Code- or Status-Register

CISC Complex Instruction Set Computer

CLK Clock

EAR Effective Address Register

ENC Enable C Decoder

FPGA Field Programmable Gate Array

GSR Global Set/Reset

IR Instruction Register

MAR Memory Address Register

MBR Memory Buffer Register

MIR Micro Instruction Register

MMCM Mixed-Mode Clock Manager

MMux MPC Multiplexer

MPC Micro Program Counter

MPM Micro Program Memory

OPR Operand Register

PC Program Counter

PLL Phase Locked Loop

RAM Random Access Memory

RD read

RISC Reduced Instruction Set Computer

RTL Register Transfer Level

SP Stack Pointer

THS Total Hold Slack

WR write

XR Index Register X

YR Index Register Y

1 Introduction

The aim of this thesis is the implementation of an 8-bit microprocessor architecture on a Field Programmable Gate Array (FPGA) using the hardware description language VHDL. The concept of the microprocessor is already existing as a course material from the "Technische Informatik I" course. Only small changes regarding the control bits of the architecture were made to the existing concept. Additionally, a more detailed concept for handling interrupts was added. The already updated design of the architecture can be seen in Figure 1. The slightly changed instruction set and further specifications of the microprocessor can be found in Appendices A and B.

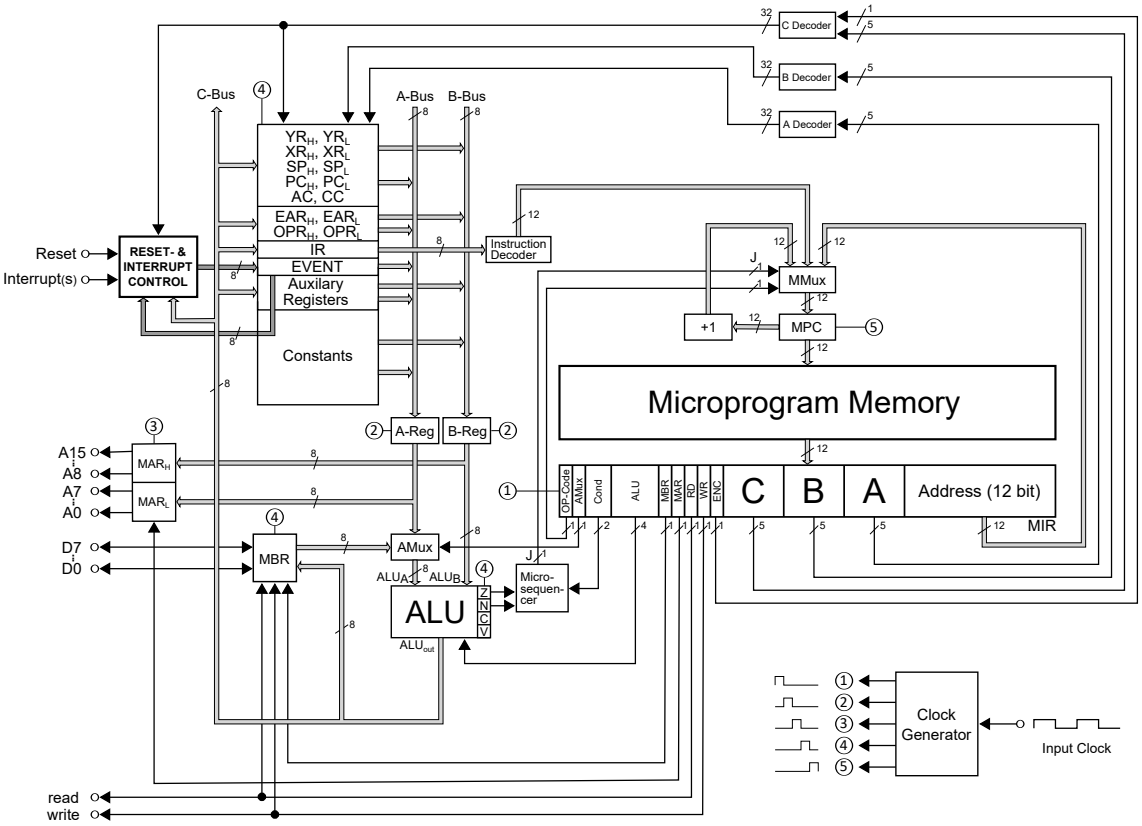

Figure 1: Microprocessor Structure.

For designing, simulating, and finally generating the bitstream for the FPGA, the VIVADO Design Suite (Version 2016.2) was used. An introduction to the VIVADO software, some basic VHDL constructs, a short description of the used hardware, and a description of the workflow of bringing an FPGA design to hardware, can be found in Appendix C. As this document is used for the bachelor-course "Technische Informatik Labor", this document is written in German.

In Chapter 2, the architecture of the implemented microprocessor and its advantages and disadvantages compared to other architectures are described. Thereafter, in Chapter 3, the structure of the VHDL modules is described, followed by the different simulation styles that were used to verify the correct functionality of the modules in Chapter 4. The following Chapter 5 describes the process of implementing the

design in hardware. The target platform for the completed design is the Digilent Basys 3 Board [1], as this is the board that is also used in the course "Technische Informatik Labor". Consequently, all considerations in this thesis are related to this board. In Chapter 6, the potential to improve the processing power of the microprocessor is analysed and, finally, in Chapter 7, a possible usage and further possible improvements for the processor are described.

In Appendix D, the complete microcode that was written to fulfil the complete instruction set of the processor is included.

The code written for this project and some other files can be found online at GitHub (https://github.com/bevac/microprocessor), among others:

- VHDL modules of the processor parts including testbench files;
- C-code of programs that create test files for the simulation of some modules;
- Excel file including the complete microcode;
- some of the test programs that were used to test the functionality of the completed processor module;
- a waveform configuration file that can be used with the VIVADO Simulator when simulating the processor module.

2 Background

The concept of the microprocessor architecture was already illustrated in Figure 1. The following sections describe the characteristics of this particular architecture and how the processor operates, including the purpose of the internal registers, the different addressing modes, and how the parts of the processor are clocked to execute an instruction.

2.1 Accumulator Machine

The processor uses only one internal register for arithmetic operations. The first operand of a calculation is stored in the Accumulator (AC) and the second operand (if needed) has to be loaded from the external memory. As only one address may be part of the OP-code (the address of the second operand), this type of architecture is also called a *1-address-architecture* [2].

There also exist so called *"Register Machines"*, which feature more internal registers (typically 8 to 32)[1]. The instructions of this architecture type address two registers that are containing the operands. If the result is stored back into a register, which was also used as an operand register, it is a so called *2-address-architecture*. Additionally, there is also the possibility that a destination register is part of the OP-code (*3-address-architecture*) [2].

To perform the same operations on an accumulator machine and a register machine, the accumulator machine typically needs more instructions, as every part of a calculation has to use the AC register. Intermediate results may need to be stored in the external memory. On the other hand, the OP-code of the accumulator machine is shorter, as no registers have to be addressed (the instructions are using the AC register inherently) [2].

2.2 CISC Architecture

The used processor is a so called Complex Instruction Set Computer (CISC) architecture. This type of architecture features powerful addressing modes. Due to its complex addressing modes, no easy implementation in hardware is possible, and therefore, a microprogrammed control unit has to be used to split one macro instruction into several microinstructions. Towards this goal, a microprogram memory is used. At each memory location the control bits that encode the behaviour of the processor for a certain microinstruction are stored. By starting the execution at distinct addresses, certain macro programs are performed by performing several microinstructions [2].

[1] The accumulator based architecture of this theses also includes more internal registers that can be used for instance to store intermediate results of calculations. However, only the AC register can be used directly as an operand of a processor instruction. There are also some other instructions that affect special registers of the architecture, as explained in more detail in Section 2.5.

Another architecture type is the so called Reduced Instruction Set Computer (RISC) architecture that is not using complex addressing modes for each operation. Instead of separate operations to load and store operands are used. Consequently, a RISC machine may need more instructions to perform the same operation as one more complex instruction of a CISC machine needs. However, the easy instructions of the RISC machine can be implemented directly in hardware, whereas the instructions of the CISC machine need more clock cycles to execute. One advantage of the CISC architecture is that new instructions can be implemented easily by writing a program that is stored in the microprogram memory without the need of changing the rest of the hardware [2].

2.3 Processing Steps of the Architecture

In Figure 2, the repeating operation cycle of the processor can be seen.

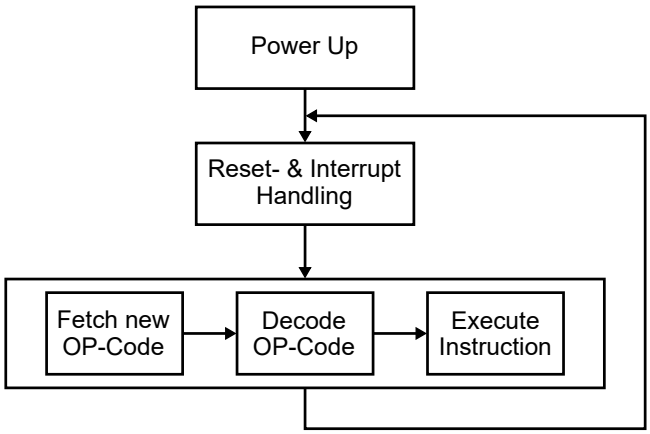

Figure 2: Operation Cycle of the Processor.

Already after the first power up, the processor checks if a reset- or interrupt signal has occurred, and, if necessary, it serves this event. Otherwise, the processor fetches the next OP-Code that should be executed from the external memory. The OP-code gets decoded and the instruction represented by the OP-code gets executed. As the processor is a *von Neumann architecture*, the program data and the instructions are stored in the same memory. Consequently, executing an instruction may also need to load data from the Random Access Memory (RAM). After a completed instruction, the processor again checks for a reset- or interrupt condition.

2.4 Internal Registers

The following list explains the internal registers of the microprocessor that are used to control the program flow and store intermediate results. Some registers are 16-bit registers, but they actually consist of two 8-bit registers (HI-byte and LO-byte).

- The Index Register Y (YR) and Index Register X (XR) are used to store a 16-bit address. These index registers can be used for addressing purposes as described in more detail in Section 2.5.
- The Stack Pointer (SP) is also a 16-bit address-register. The address stored in the SP register marks the top of the stack.
- The Program Counter (PC) is a 16-bit address register too. It points to the code of the program that is currently processed.
- The AC is the general-purpose register used for arithmetic operations, as already described in Section 2.1.
- The Condition-Code- or Status-Register (CC) stores the status flags of the Arithmetic Logic Unit (ALU). In the complete instruction list in Appendix B, you can see which status bits are updated in the CC after a certain instruction. The fourth bit of this register is the *Interrupt-Disable-Flag*. If this flag is set, no interrupt is served by the processor (except not maskable interrupts such as a reset or the software interrupt).
- The Effective Address Register (EAR) is a 16-bit address register. It temporarily stores an address that is needed during the execution of an instruction (e.g., the effective address of an operand).
- The Operand Register (OPR) is a 16-bit register where an operand can be stored temporarily.
- The Instruction Register (IR) stores the OP-code of the executed instruction. At the fetch-operation of the processor the next OP-Code is transferred from RAM to the IR. Then the OP-Code gets decoded: the content of the IR is shifted four bits to the left by the *Instruction Decoder* and the result is stored into the Micro Program Counter (MPC) that encodes the starting point in the Micro Program Memory (MPM) to execute the macro-instruction of the loaded OP-Code.
- The EVENT register can be loaded also by a signal on an external reset or interrupt line, respectively. The content of this register encodes if an interrupt (and also which interrupt) has to be served by the processor. The concept of serving interrupts is described in more detail in Section 3.6.
- The registers U, V, W, Z are just used as helping registers, as for some calculations more temporary values have to be stored.
- There are also registers storing constant values, which are used for executing microprograms.

2.5 Instruction Set and Addressing Modes

The processor offers a variety of instructions like loading values from memory, storing values to memory, performing branches, stack operations, and different arithmetic operations regarding the AC like addition, rotation and so on. The complete list of the available instructions can be found in Appendix B.

Depending on the instruction, several of the following addressing modes are available (an example of the related assembler notation that is used in "Technische Informatik I Übung" is given).

Inherent Addressing. Only the OP-code is needed for the execution of the instruction, as the operand is already defined by the instruction itself (e.g. `PUSHA`).

Immediate Addressing. The OP-code is followed by the operand needed for the instruction. Depending on the instruction, the operand can be one or two byte long (e.g. `ADDX #0004` or `ADDA #04`). As it can be seen from the examples, the instruction `ADDX` uses a 2-byte operand (the operand is given in hexadecimal format) whereas the instruction `ADDA` uses an 1-byte operand. The instructions are adding the operand to XR respectively to the AC register. As XR stores a 16-bit value and AC an 8-bit value also the operands have different lengths.

Absolute Addressing. The OP-Code is followed by an address. In case of a store or jump instruction this already is the address, where a certain value has to be stored, respectively the address where the program should continue. Otherwise this is the address, where the operand (one or to byte long) – which is needed for the execution of the instruction – is stored. (e.g. `STA 0020` or `ADDA 0020`)

Register Indirect Addressing (inclusive offset). The OP-code is followed by one byte. The value of this byte (range -128 to 127) is added to the content of XR respectively YR. The result of this operation is the address, where the actual operand for the instruction can be found. For store and jump instructions this is again already the desired address. (e.g. `STA @(X+02)` or `ADDA @(Y+FF)`)

PC-Relative Addressing. This addressing mode is used for branches. The OP-code is followed by one byte (range -128 to 127) that might be added to the content of the PC to perform a branch in the program if certain conditions are met (e.g. `BNE LOOP` - for branches labels are used in assembler code to define the branch destination[2]).

2.6 Clock

Inside the processor the Input Clock (CLK) is divided into 5 sub-clocks, as it can be seen in Figure 3. Every sub-clock is used to clock different parts of the processor (at the rising edge of the particular clock). Figure 1 shows which sub-clock affects which register. During one clock cycle of the input clock, one microinstruction is executed.

CLK 1. Loads the next microinstruction from the MPM - which is addressed by the address stored in the MPC register - into the Micro Instruction Register (MIR). The content of the MIR are the control-bits for certain parts of the processor and are responsible for performing the desired task of the loaded microinstruction. A complete list containing the information about the effect of every single control bit in the MIR can be found in Appendix A.

[2] Of course labels can also be used for absolute addressing.

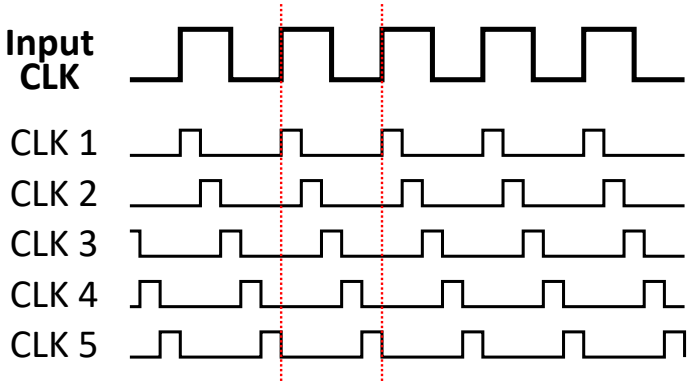

Figure 3: Clocks inside the Microprocessor.

CLK 2. Loads the registers A and B with the content of an internal register of the processor. The registers (from which the content should be loaded) are defined by the select bits (A or B, respectively) in the MIR.

CLK 3. Loads the Memory Address Register (MAR) with the contents of the registers A and B if the corresponding enable bit in the MIR is set. The MAR is responsible for addressing the external RAM.

CLK 4. This sub-clock is responsible for different tasks. Inside the ALU the flags (zero-, negative-, overflow-, and carry-flag) are updated at this clock cycle. Also a new value can be loaded into the Memory Buffer Register (MBR). Depending on the control bits MBR, read (RD), and write (WR), the value appearing on the output of the ALU or the content of the RAM (where address stored in the MAR points to) can be loaded into the MBR. Last but not least, the output of the ALU can be stored into an internal register of the architecture if the Enable C Decoder (ENC) bit is set in the MIR (the registers storing constants can not be overwritten). For storing a value to the EVENT register, a special logic is used as described in detail in Section 3.6. The ALU permanently calculates an output for ALU_{out} (just combinatorial logic), but as at the fourth clock cycle the ALU output can be stored, and the internal flags are updated, these values must have stabilized until the rising edge of CLK 4. The input for ALU_{A} is selected by the ALU Multiplexer (AMux).

CLK 5. The MPC is loaded with the address of the microinstruction in the MPM, which should be executed next. The MPC Multiplexer (MMux) selects the next address between three possibilities. Either the content of the IR (shifted four bits to the left), the address stored in the MIR, or the current value of the MPC increased by 1 is chosen, depending on the OP-code bit in the MIR and the output J of the *Microsequencer*. J depends on the Cond-bits in the MIR and the Z- and the N-flag of the ALU.

3 VHDL Implementation

To make debugging easier, some parts of the processor were implemented as separate VHDL modules, which makes it possible to test the modules separately instead of just testing the complete processor. In the processor module, these submodules were connected using a structural VHDL modelling style. Structural modelling style means that just the connections between different modules are described [3].

However, most of the submodules are described using a behavioural style of modelling. In this modelling style, only the behaviour of a circuit is described. It is possible to describe the behaviour in a quite abstract way or to use a so called Register Transfer Level (RTL) modelling style, which means that the transfers of contents between registers are described. For instance, an addition could be described quite abstract by just using the '+' sign in the VHDL code, but to have more control on how the inferred hardware will look like, a more detailed description of the adder circuit is necessary [3, 4].

Below, some particular properties of the implemented VHDL modules are described. Some modules are using not only a structural or behavioural style of modelling, but a mixture of both.

The code of the models (except for the Clock Generator module, which was created using an *IP Core* of Xilinx) can be found online at https://github.com/bevac/microprocessor.

3.1 Multiplexer and Microsequencer

A very general code was used for describing the functionality of a multiplexer. This module is used for the ALU Multiplexer (AMux) and the MPC Multiplexer (MMux). A general description was necessary as the AMux and the MMux have a different bus width and a different number of select lines. Without a general description, separate modules would have been necessary.

The microsequencer module uses a quite simple behavioural description (that actually would not necessarily need a separate module inside the processor module). Actually it is just a small multiplexer, which would make it also possible to use the same general multiplexer module instead of the extra microsequencer module.

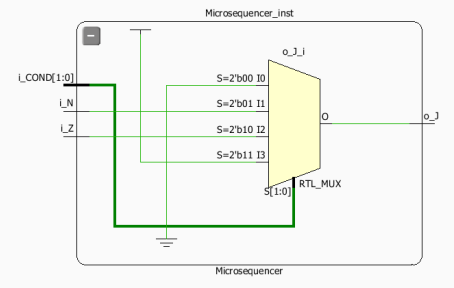

Figure 4: Elaborated Design of the Microsequencer.

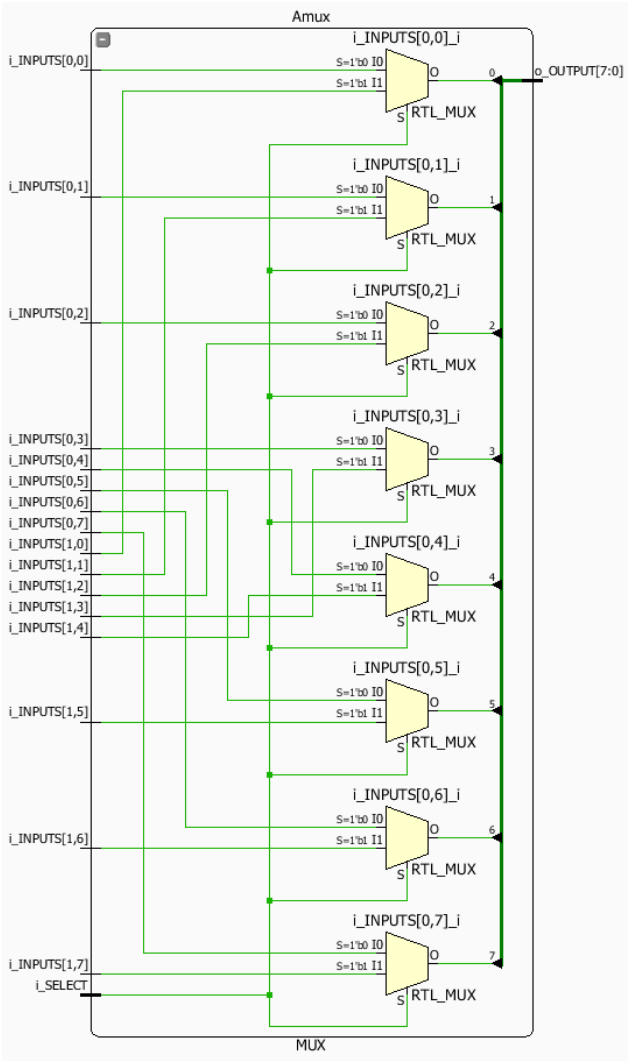

Figure 5: Elaborated Design of the AMux.

3.2 ALU

The Arithmetic Logic Unit (ALU) is responsible for performing arithmetic operations. It is described using two processes. The first process includes a case statement to decide which operation has to be performed depending on the select bits. The second process is clocked and is responsible for storing the new flag values of Z, N, C, and V.

The ALU itself contains as a sub-module the description of a serial adder (that is used for the addition operation). The serial adder is build out of a chain of full adders (also a separate module).

In Figure 6 the elaborated design that was generated from the VHDL code can be seen. Four register store the current content of the flag registers and are updated when the operation finished. A big multiplexer connects the result of the selected operation with the output. Further multiplexers select the correct value to update the flag registers.

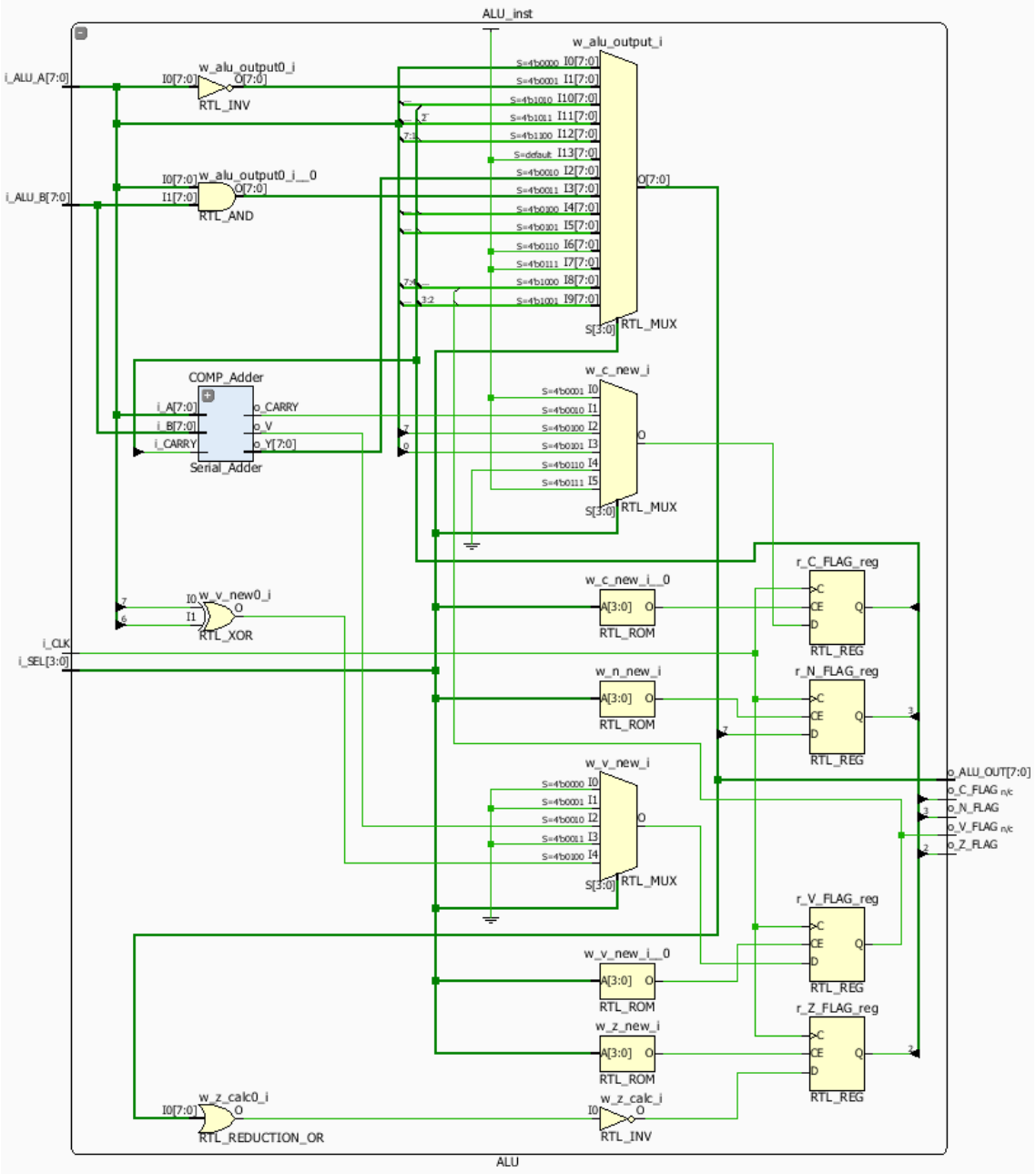

Figure 6: Elaborated Design of the ALU.

3.3 Communication

This module is responsible for the communication between the processor and the external RAM. It includes the Memory Address Register (MAR) and the Memory Buffer Register (MBR). The data bus is bidirectional (both processor and memory can read/write data from/to the bus). As processor and memory must not write data to the bus at the same time, tri-state buffers have to be used. This type of buffer is needed if more sources are connected to a single bus. The used source writes data to the bus, whereas all other sources have to be in a high-impedance state [4].

For the communication to work properly, the external memory is also connected to the RD- and WR signals generated by the processor. If the WR signal is activated, only the processor is allowed to write data on the data bus (i.e., the output of the memory has to be in a high-impedance state) and the data on the data bus is stored in the external memory at the location of the address stored in the MAR. If the RD signal is activated, the output of the processor has to be in a high-impedance state, such that only the content of the external memory at the address location stored in MAR appears on the data bus and can be read in by the MBR.

In Table 1, you can see the values of the select lines of a microinstruction that control a read- or write operation.

	Select Bits	Operation
Read	0110	$MAR_L \leftarrow A_{reg}$, $MAR_H \leftarrow B_{reg}$
	1010	$MBR \leftarrow M[MAR]$
Write	0100	$MAR_L \leftarrow A_{reg}$, $MAR_H \leftarrow B_{reg}$
	1001	$MBR \leftarrow ALU_{out}$
	0001	$M[MAR] \leftarrow MBR$

Table 1: Communication Control.
(Select Bits: MBR MAR RD WR)

As shown in Table 1, a read-instruction needs two microinstructions to be executed and a write-instruction needs even three cycles. The reason for this is the assumption of a slow external memory. From the description of the clock cycles in Section 2.6, it can be seen that the MAR can be loaded at the rising edge of CLK 3, and that the MBR can be loaded at the rising edge of CLK 4. Theoretically, it would be possible that in the same clock cycle at CLK 3 the MAR is loaded and at CLK 4 the MBR is loaded from the external memory (read-operation), respectively the current content of the MBR is written to memory until the end the microinstruction (write-operation). However, also the specifications of the external memory have to be considered to make sure that this would actually be possible.

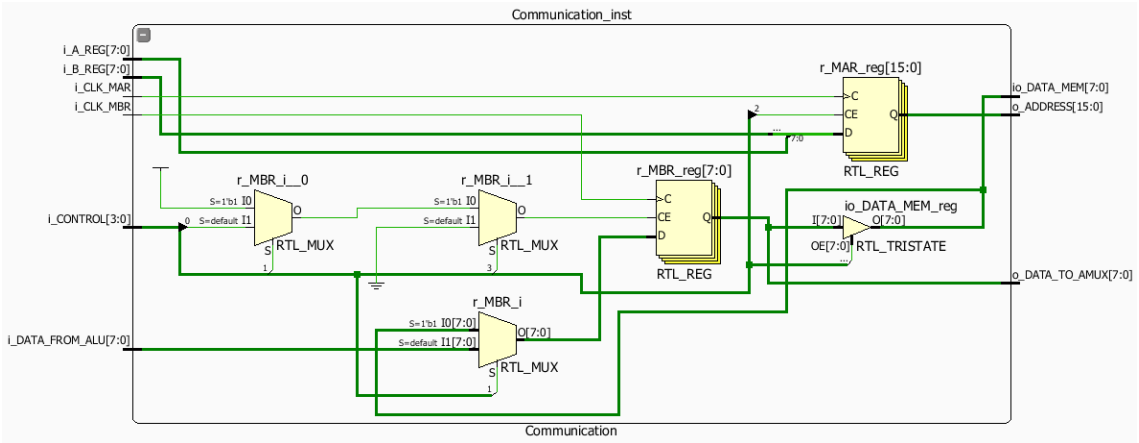

Figure 7: Elaborated Design of Communication Module.

3.4 Clock Generator

To create the five sub-clocks from the input clock, a Mixed-Mode Clock Manager (MMCM) or Phase Locked Loop (PLL), which are part of the FPGA, should be used. As it is quite complex to write code that would infer an MMCM in VIVADO, the *Clocking Wizard* was used. The Clocking Wizard is an *IP Core* provided by Xilinx. It provides a graphical user interface to configure an MMCM or a PLL. Figure 8 shows the *Block Design* that was created including the Clocking Wizard block [5, 6].

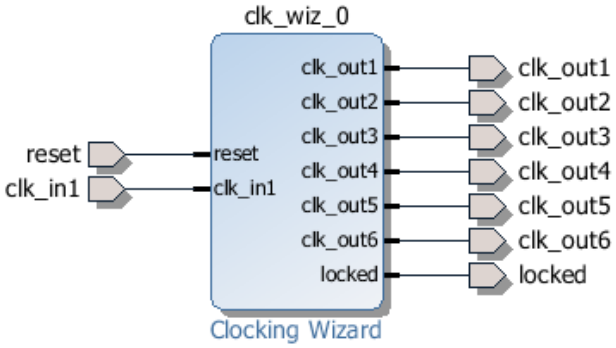

Figure 8: Block Design using Clocking Wizard.

As `clk_in1` the 100 MHz clock provided by the Basys3 Board is used. According to Figure 3 the output clocks `clk_out1` to `clk_out5` should also work at a frequency of 100 MHz. But as at the design step of implementing the design on the target platform (as described in Chapter 5) it turned out that this frequency is too high for the design to work properly, the frequency was reduced to 10 MHz. `clk_out6` outputs a 10 MHz signal with 50 % duty-cycle and can therefore be seen as as the "virtual input clock" which then accords with the input clock of Figure 3. `clk_out6` is further used instead of the actual input clock for clocking some other parts of the design, such as the debouncing circuit, which is described in more detail in Section 3.6.

Following settings were made to configure the clocking wizard [5]:

- The input clock was set to 100 MHz as the desired hardware board features a 100 MHz clock. As source type *Single ended clock capable pin* was selected, as the 100 MHz input clock was used as the only input for the MMCM.[3]
- The sixth output clock was configured as already described above. To make it possible to define a different frequency for the output clocks than for the input clock, the *Frequency Synthesis* has to be activated under the *Clocking Options*.
- The other five output clocks were created and configured to have the same frequency as the sixth output clock, but only with a 20 % duty cycle and the phase was corrected such that every clock starts after the falling edge of the previous clock (phase as a multiple of 72°).

[3] If the input clock would be used also to clock other parts of the design, it would be necessary to change the source type to *No buffer* or *Global buffer* [7].

- The *reset* input port was enabled such that it could be used to reset the MMCM. In the final design, actually no hardware reset (which completely resets the hardware of the processor) was included: only a microprogram based reset routine provided by the processor is included, as described more detailed in Sections 3.5 and 3.6. Consequently, this reset input was tied to ground in the processor module, but may be used if a hardware reset is added to the design.
- The *Safe Clock Startup* option was activated under the *Clocking Options*. This option only activates the output clocks when they are stable, which is achieved by using a BUFGCE (BUFG clock buffer including a clock enable line) that is activated if the *locked* signal is sampled High for 8 input clocks. Locked is asserted when the output clocks are stable.
- The locked signal is also used as an output of the Clocking Wizard. It is only used for simulation purposes and is not connected to anything else in the processor module.

A so called *Wrapper* is created of the block design. This creates a module with the functionality of the block design by creating the necessary code. This module was then used as a component in the processor-module.

In the following, when talking about the input clock, actually the "virtual input clock" with a 10 MHz frequency is meant.

3.5 Power-On Reset

One important quality of the design is that it powers up in a known state, otherwise it might corrupt the complete RAM that is connected to the processor. In the VHDL code it is possible to define initial values for registers. This initial values are loaded by the FPGA at power up by the Global Set/Reset (GSR) routine. In most cases this automatic routine is sufficient, but especially for clocked circuits it might be problematic if the reset signal is not released exactly at the same moment for all registers [8]. One further problem is that these errors only appear in hardware and not in the simulation, as the defined initial values are assigned to all registers at the start of the simulation at exactly the same time.

Actually, there is no part in the design at which the reset signal has to be released at exactly the same time, as all sub-clocks are used to clock different parts of the design, and registers that are loaded by the same sub-clock do not depend on the other registers clocked by the same sub-clock. It is just important that the sub-clocks are not used before the MMCM is working properly. Therefore, the Safe Clock Startup is activated in the Clocking Wizard as described in Section 3.4. Consequently, no special concept for reset at power-on should be necessary.

It is just irritating that, when simulating the Clock Generator, one peak appears at all output clocks at the beginning of the simulation - even when the described options are used - presumably as the locked signal starts in an unknown state ('X'). But probably this occurs just in the simulation and in hardware the locked signal of the MMCM will be cleared before the GSR is deasserted. However, even if this

single clock peak would appear (when the GSR is already finished), this should not be problematic. The GSR routine preloads the Micro Instruction Register (MIR) with an instruction that performs a jump to address `0x000` in the Micro Program Memory (MPM) and the MPC is preloaded with `0x000`. The first time a rising edge at CLK 1 appears, the microprogram at address `0x000` in the MPM, which is just responsible for increasing the MPC by 1, is loaded into the MIR. This increase of the MPC happens at the next rising edge of CLK 5. Therefore, it is no problem if the first appearing sub-clock is not CLK 1. However, once the microinstruction of address `0x001` is loaded into the MIR, the clocking has to work properly.

3.6 Reset- and Interrupt Logic

The original design of the microprocessor architecture just defined that only the reset- and interrupt pins have access to write into the EVENT register of the processor. As in this scenario the processor can not reset the event register after a served interrupt, it would be only possible to provide interrupts that are active as long as the corresponding reset- or interrupt line is active. The reset- and interrupt control was completely redesigned to achieve that interrupts are only served once after pressing the corresponding reset- or interrupt button.

At the beginning of the external memory, an interrupt vector table is saved as it can be seen in Table 2. The reset- and interrupt concept is similar to the concept of an AVR Microcontroller, like for instance the ATmega 2560 [9].

Vector No.	Program Address	Source	Interrupt Definition
1	`0x0000`	RESET	Power-on Reset, Reset Pin
2	`0x0004`	SWI	Software Interrupt
3	`0x0008`	INT 1	External Interrupt 1
4	`0x000C`	INT 2	External Interrupt 2

Table 2: Interrupt Vector Table.

Every interrupt vector uses four bytes of memory. The arrangement of these four bytes can be seen in Table 3.

Byte Nr.	Content	Functionality
1	`0x7E`	jump instruction
2	Addr.	Address where the program should jump to
3		
4	`0x00`	not used

Table 3: Memory Content for one Interrupt Vector.

The interrupt vector starts with the OP-code for a jump instruction. Then a 16-bit address follows, i.e., the destination for the jump instruction. The last byte of an interrupt vector is never used. The reason for this decision is described later in this section, when describing how the call of an interrupt is implemented.

At address `0x0000` of the external memory the RESET vector is located. At power-on the program counter points to this address. Consequently, the first instruction the processor executes is a jump to the address comprised in the reset vector, which is the start address of the main program. Via an external button, this interrupt vector can also be executed (it restarts the main program and additionally resets some internal registers of the processor beforehand).

The SWI vector at `0x004` jumps to the code of the interrupt handler that should be performed if a software interrupt occurs (special OP-code instruction).

The further interrupt vectors can be called by external buttons (or other sensors). For the processor two interrupt pins were implemented. Nevertheless, following the described concept even more interrupt pins can be added to the design. Figure 9 shows the concept used for reset- & interrupt handling.

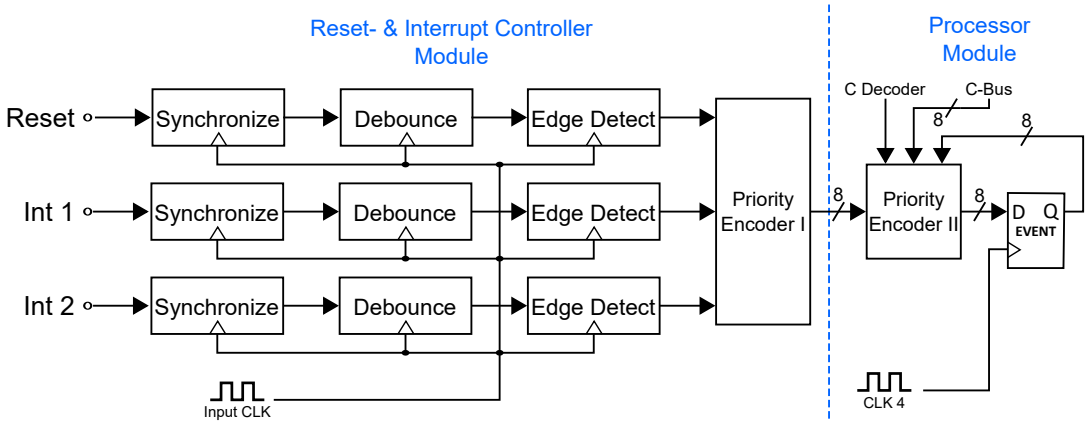

Figure 9: Reset- & Interrupt Logic.

Every button is connected to the submodules *Synchronize, Debounce* and *Edge Detect*, which are described later in this section. If at *Priority Encoder I* pulses from more buttons appear at the same time, only the pulse with the highest priority is selected. The convention is that the lower the address of the interrupt vector, the higher is its priority (except for the software interrupt, which is executed like a normal OP-code, and has hence no priority over the hardware interrupts[4]). The output of Priority Encoder I is the interrupt vector number (as listed in Table 2) as a 8-bit value of the interrupt that occurred. If no pulse is appearing, this 8-bit value is `0b10000000` (`0x80`), which indicates that no interrupt condition appeared. As the microcode detects an interrupt by checking the sign bit (just the most significant bit encodes if an interrupt occurred), up to 128 interrupts could be implemented using this technique.

Priority Encoder II (which is already part of the processor module) decides which value has to be loaded into the event register. The event register is clocked by CLK 4 and can be updated at every rising edge of this sub-clock. If no event has to be served, the indicator that no event happened (`0x80`) is stored in the event register,

[4] The reason for placing the software interrupt anyway after the reset vector is that the software interrupt uses the fixed interrupt vector address of `0x0004`, and if further hardware interrupts are added, no changes in the microcode have to be made.

otherwise the interrupt vector number of the interrupt that has to be served is stored. Following rules are applied:

- If no new event appears, the content of the event register stays the same.
- If a value from the C-Bus should be written into the event register, this value is loaded into the event register. This is used to delete an interrupt that is already served by overwriting it with `0x80` (no new event).
- If Priority Encoder II outputs a lower number (higher priority) than the number that is currently stored in the event register (this is also the case if the content is `0x80`), this address is loaded into the event register and replaces the prior event.

In Figure 2 the operating cycle of the processor was illustrated. After each instruction, the processor checks if an interrupt has to be performed and therefore jumps to address `0x001` in the microprogram memory. In Appendix D, the corresponding microcode can be found. This interrupt check works as following:

- First, it is checked if an event has to be served by saving the event-register into an auxiliary register (as the content of the event register may be replaced during further checks, but still the old interrupt should be served). If the negative flag of the ALU is set, no new event occurred (this is the reason for defining the content of the event register with `0b10000000` if no event happened, as this causes the negative flag of the ALU to be set when the content of the event register is passed through).
- If the content of the event register is not `0x80`, it is further checked if the content is `0x00`. If this condition is met, a special reset routine is called, such that it resets certain internal registers and then jumps to the reset vector.
- For any other content of the event register it is clear that some other hardware interrupt appeared[5]. By multiplying the number of the interrupt vector by four, one gets the address of the corresponding interrupt vector as it can be seen from Table 2. This multiplication by four can easily be performed by shifting the content of the event register two bits to the left. This is the reason that the interval of the interrupt vectors is chosen to be four byte (even if the last byte of each interrupt vector is not used), as a multiplication by 4 is much easier to implement than a multiplication by 3. Before a jump to the corresponding interrupt handler is performed, the current content of the program counter and the condition code register are pushed on the stack. The end of an interrupt handler uses the `RTI` command that then restores these values and continues with the main program.

The processor also offers the possibility to disable interrupts by setting the corresponding bit in the condition code register. Before serving an interrupt, also this flag is checked. Nevertheless, the interrupt handling is quite simple. As only one register stores the information about an interrupt that has to be served, it might occur that an interrupt with less priority is not executed. Although this is quite unlikely, as the

[5] As already mentioned the software interrupt is executed like a normal OP-code and therefore its interrupt vector address is never loaded into the event register.

two interrupts have to appear in a very short time interval (except if interrupts are disabled, the content of the event register can be overwritten until the interrupts are enabled again) it still might happen. To overcome this issue, a more complex structure for interrupt handling would be necessary like for instance the one used by the ATMega2560 [9]. It would be necessary to have a bit-flag for every available interrupt in an internal register, to ensure that interrupts can not be overwritten. Additionally, it would be possible to introduce interrupt-mask registers, which can be used to mask a certain interrupt instead of offering just the possibility to activate all or no interrupts. Furthermore, always the next interrupt with the highest priority has to be served followed by clearing the corresponding flag. In this case, the only situation in which an interrupt can be missed is that the same interrupt appears again before the previous one has been served.

For all interrupts that are not used (no interrupt service routine programmed) the jump instruction (`0x7E`) of the interrupt vector should be replaced by an `RTI` instruction (`0x3B`). Even if such an interrupt gets executed, it just finishes the interrupt service routine and continues with the main program.

Additionally, after each executed interrupt, always one instruction of the main program is executed before serving the next interrupt. Therefore, it can not happen that the processor is just serving fast appearing interrupts without serving the main program at all.

Synchronizing Circuit

The synchronizing circuit consists of just two D-flip flops connected in series as it can be seen in Figure 10 [10]. This circuit is used to synchronize the input signal to the clock of the digital circuit. As the input is asynchronous, the transition of the input signal can happen anytime in regard to the rising edge of the clock. Depending on the circuit, such an asynchronous input can cause wrong behaviour. Therefore, the input gets synchronized to the used clock domain, as this ensures that the synchronized input can only change at rising clock edges. A second problem that might appear is metastability. This means that a flip flop can come into a state between HI and LO if setup- and hold-times are violated (which is possible especially for asynchronous inputs). Therefore, the second flip flop is used, as it is very unlikely that also the second flip flop comes into a metastable state. The synchronisation and prevention of metastability comes to the cost of a delayed input signal.

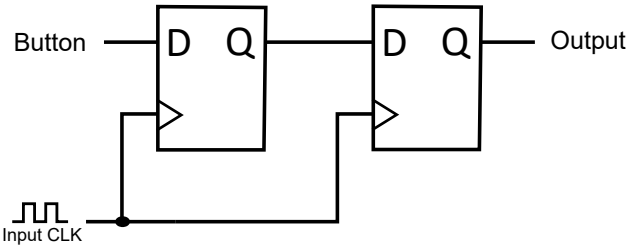

Figure 10: Synchronizing Circuit.

Debouncing Circuit

Mechanical buttons that are used as inputs for the interrupts do not output a stable voltage value immediately after they are pressed. Instead, the voltage is fluctuating until it reaches a stable value. To make sure that this bouncing voltage is not recognised as several interrupts, the debouncing circuit ensures that the signal value on the output changes only if the value at the input is stable for a certain amount of time. In Figure 11, the principle of this circuit is illustrated [11].

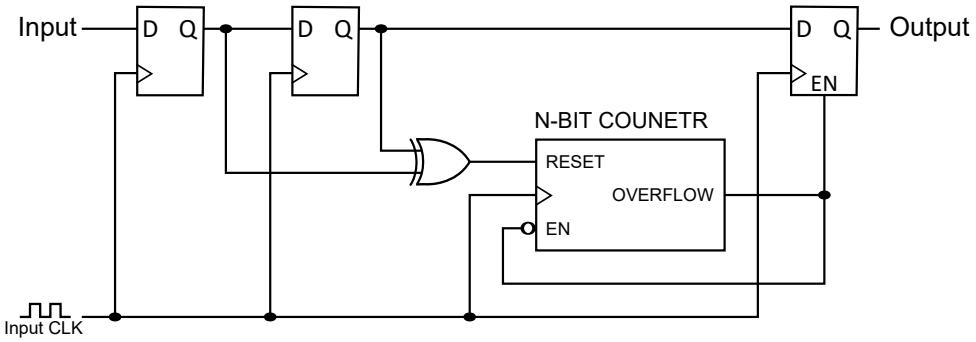

Figure 11: Debouncing Circuit.

The XOR gate resets the counter if the signal on the input changes from HI to LO or from LO to HI. The counter then starts counting as long as the input signal does not change again. When the counter overflows, the counter is stopped and the input signal is transferred to the output. If the input is fluctuating so fast that the counter never overflows, the output never changes its value.

To determine the needed bit size for the counter, the following equation is used:

$$P = \frac{(2^N + 2)}{f}$$

where N is the bit size of the counter, f the frequency of the input clock and P the duration the input signal has to stay on the same voltage level that is transferred to the output. Using a 17-bit counter, a debouncing time of about 13 ms is achieved when a 10 MHz clock is used.

Edge Detection

This circuit is used to generate only one pulse on the output when a rising edge appears as input. This ensures that, even if a button causing an interrupt is pressed longer, the interrupt handler will be performed only once. If an interrupt should be executed repetitively as long as the button is pressed, this circuit has to be omitted for this interrupt. In Figure 12 the circuit is illustrated [10].

As soon as the input changes to HI, also the output of the AND gate changes to HI. This HI value is also the output at the next rising edge of the input clock. At the same clock edge also the output of the first flip flop changes to HI, which causes the output of the AND gate change to zero. Therefore, the circuit outputs only one pulse (lasting one clock cycle) if a rising edge appears on the input. The input has first to change to LO before a new rising edge can trigger a new pulse.

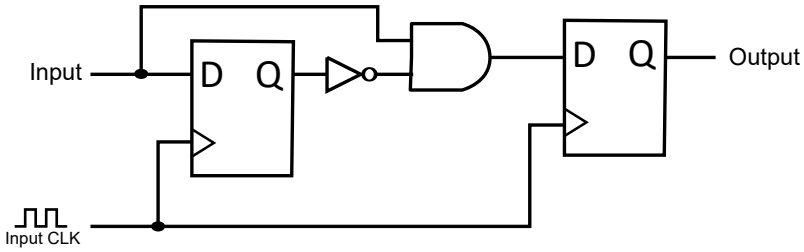

Figure 12: Rising Edge Detection Circuit.

3.7 Processor

In the processor module all previously described modules are connected together. Furthermore, also some behavioural descriptions that are not implemented in separate modules are included. The behavioural descriptions include loading of the MIR, of registers A and B and of the MPC. Also the description of how to load data into the internal registers (including the decoders) – especially the specific loading specifications for the event register – are included.

One further important functionality of the processor module is the initialization of the MPM from a text file. The complete microcode can be found in Appendix D. The microcode was written in Microsoft Excel, but, as for initializing a text file containing just the memory content is needed, a VBA script was used to create the text file (which makes it also easier to recreate the memory file, if changes are made to the memory content). To read in the binary values from the text file an *impure function* is used. A function needs to be defined impure, if it is possible that it returns different values even though it is called using the same parameters [4].

4 Behavioural Simulation

One important part of a VHDL design is to test the correct functionality of the design. Especially for bigger designs, it is advisable to test also the separate modules of the complete design, as it might be very difficult to detect errors if only the complete design is tested.

To verify a design, a so called testbench is used. In Figure 13, the concept of a testbench is illustrated [4]. The module that should be tested (unit under test) is used as a component in the architecture of the testbench. Then different input signals are applied to the module that is tested, and it is checked if the output of the design is correct.

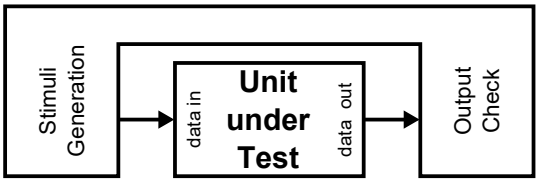

Figure 13: Testbench Concept.

Depending on the complexity of the modules, different approaches for the functionality verification were used. Of course the best way to validate a design is to test all possible inputs. But for big designs this is not possible as it would be too time consuming. For this cases, the most important stimuli combinations should be tested, followed by a series of random stimuli [4].

The testbecnh files of the project are available online at https://github.com/bevac/microprocessor.

4.1 Verification using the Waveform Viewer

For designs with simple functionality (like the Microsequencer or the Clock Generator) the simplest solution for testing a design is to write a VHDL code that applies all different input signal combinations to the design and than check the output by viewing the waveforms of the simulation result. As this check has to be done manually, it is only reasonable for small design and it is more likely to overlook errors [10].

4.2 Complete Design Verification using VHDL

This concept was used for the verification of the serial adder. For creating the stimuli (like for the previous approach) a loop was used, that applies all possible input combinations, but the correctness of the output was automatically checked by the testbench. The serial adder was described using a chain of full adders. The output of this serial adder was then checked by the addition performed using a '+' sign in VHDL code. If any error occurs, a report message is written to the console.

4.3 Verification using Text Files created in C

For some modules like the ALU, the communication module, or the general multiplexer, a different approach using textfiles was used. This concept is illustrated in Figure 14 [10].

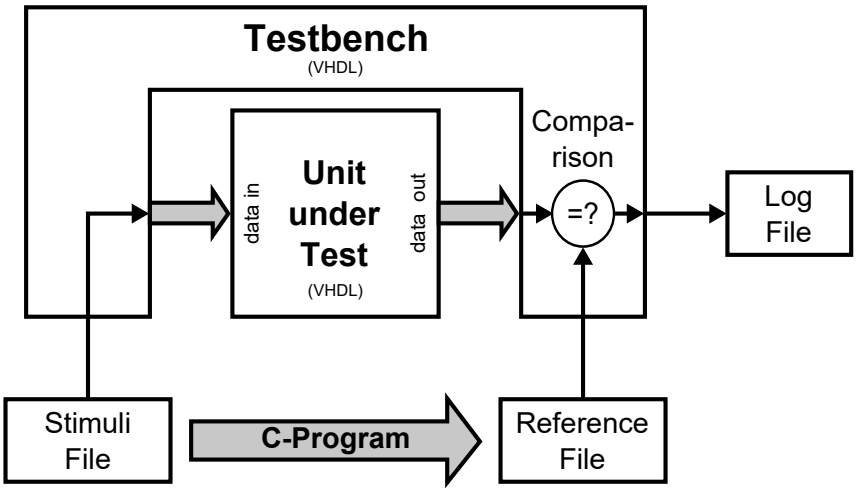

Figure 14: Verification using Text Files.

First a stimuli file is created, containing the stimuli for the unit under test. Then the behaviour of the design is modelled in a C-program that writes the correct outputs to the stimuli into a reference file. The stimuli inputs included in the stimuli file are read in by the testbench and then applied to the design. Then the corresponding outputs of the design are compared to the outputs included in the reference file. In the log file the results of the comparisons are noted. Additionally, for every error a message is written to the console.

For the performed tests, only one file was used for the stimuli and the related outputs. The stimuli file was also created by the C-program that then computes the corresponding outputs. For testing the ALU and the multiplexer, at the beginning particular stimuli were created, followed by random stimuli using the C-functions `rand` for generating random numbers and `srand` for initializing the pseudo-random number generator.

For testing the communication module three text files were used. The first file includes a random content of the RAM, which is read at the beginning of the testbench. A second file contains the stimuli that are applied to the design and the contents of the MAR and the MBR after each operation, to check the correct functionality. The third file (which is automatically generated by a C-program) contains the content of the RAM after all stimuli of the stimuli file have been applied. The file content is compared to the content of the RAM of the testbench. This testing strategy is used as an additional check in the end, to check whether the RAM content is correct after completing all operations.

4.4 Testing the complete Processor

In comparison to the other submodules, an adequate verification for the complete processor is quite complex. It is not possible to just create a random RAM content as input for the processor, as a valid program has to follow certain rules. Some important rules are the following:

- Only OP-codes that are part of the instruction set are allowed. This means that it has to be ensured, that if a fetch is performed, the PC also points to a RAM address that stores a valid OP-code.
- In link to the previous point, every OP-code has to be followed by zero up to two additional bytes (depending on the addressing mode of the respective OP-code) before the next legal OP-code appears in RAM.
- Furthermore, a jump or branch is not allowed to an address that does not hold a legal OP-code.
- A store operation should not affect the program, so that one of the already mentioned problems could occur.

Therefore, concrete programs using different instructions of the processor were programmed (by writing a file with the content of the external memory that performs a certain program) for testing purposes. Also a special program to test the functionality of the interrupts was written (which was then also used for testing if the processor module is also working on the FPGA). In this program, after power up, the interrupts are activated and a value is written to a certain memory location. Both external interrupts access exactly this value and can also store the result of their computation back at the same memory location. Whereas one interrupt routine adds 1 before storing it back, the other routine is subtracting 1 before storing the result back. This simple program checks perfectly if the processor is operating correctly and additionally whether the interrupts are working (though it does not check if all microinstructions are implemented correctly).

Unfortunately, for a typical testbench (as illustrated in Figure 13) in VIVADO it is not yet possible to access internal signals of the tested module. Therefore, it is not possible to check internal values of the design in the VHDL code of the testbench[6] (they can only be displayed in the Waveform Viewer). However, the testbench should make it as easy as possible to detect errors in the design, even if not every part of the design can be checked automatically by the testbench. For this reason, a log file is used, which after each write operation (when the WR-signal changes from HI to LO) logs the content that was written to RAM including the destination address. Also every RD operation is logged. Additionally, the current simulation time after a read or write operation is written to the log file, which makes it easier to find certain events of the simulation in the Waveform Viewer. The latter is a very useful extra tool to check if all parts of the processor are working correctly, as every internal signal can be added as a waveform.

[6] It would be necessary to write a testbench on a "deeper level", which means to include the code to test the design in the module itself, as on this level the signals can be accessed.

Similar to the testbench of the communictaion module, a memory input file (containing the program code) is first read in by the testbench and then the processor is started to run the program. Every read- and write operation is logged. Additionally, a file containing the simulation parameters that configures the simulation has to be included:

- A boolean value indicates whether a memory output file is provided. If so, the content of this file is compared to the memory content of the RAM in the simulation after the processor is stopped.
- A second boolean variable defines how the simulation determines when to stop. The first possibility is that the parameter file also contains an integer indicating an address at which the processor stops the execution when this address is read in by the MAR. For this option, a second integer indicates how often this address has to appear, until the processor stops. A second option is that the processor performs a certain combination of inputs to the reset pin and the interrupt pins before it stops. In this case, a list is included in the parameter file, which contains the inputs that should be applied to the certain input pins, or, respectively, wait instructions between signal assignments.

The hardest part about testing the complete processor is to detect the reason for occurring errors, as they can appear because of a failure in the design or because of an error in the microcode. Both of them can cause undefined behaviour. The mentioned log file in combination with the Waveform Viewer helps to detect all kinds of errors, as every change in the memory is logged (of course it requires that the communication between the RAM and the processor is working correctly). To check the functionality of the processor instructions, which perform no store-operation, these results were written explicitly to the RAM.

In the microcode every unused line was filled with a microprogram that performs a jump to the last address of the microprogram memory, which makes it easy to detect if a line in the MPM that should never be accessed is executed.

5 Hardware Implementation

After the successful verification of the designs functionality, the next big step is to bring the design on the FPGA. As target platform the Basys 3 Board from Digilent is used, as this board is also used in the course "Technische Informatik Labor". The board includes an Artix®-7 FPGA from Xilinx with already connected hardware peripherals like switches, buttons, 7-segment displays, and a 100 MHz clock.

5.1 Design

For the final design we did not use an external memory. Instead, the RAM was implemented inside the FPGA. Additional modules were implemented in VHDL to make it possible to check whether the design is working. In Figure 15 the concept of the final design is illustrated.

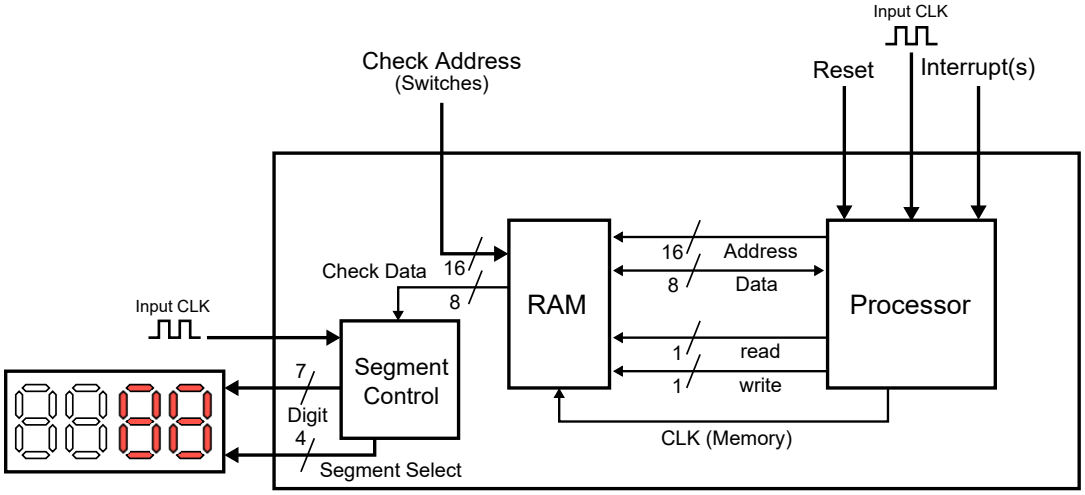

Figure 15: Concept for Hardware Implementation.

Functionality Check

To make it possible to check the content of the RAM, the available hardware of the Basys 3 Board is used[7]. 16 switches are used to encode a certain address, whose memory content from this memory location can be displayed in hexadecimal format on two of the four available 7-segment displays. Every digit of the 7-segment display can be controlled using the same seven signal lines to encode the segments that should light up. Furthermore, four signal lines are used to encode which of the four digits is affected by the seven other signal lines. Consequently, it is not possible to display two different digits on separate 7-segment displays at the same time. However, using a state machine that switches fast enough between the displayed digits, it is possible

[7] For debugging purposes it would be possible to use special IP Cores to track internal signals of the design. However, for testing, the described concept was implemented using the available hardware on the board [12].

to achieve the effect for the human eye that different digits, seem to be displayed on separate 7-segment displays [1]. To achieve this functionality, two new modules were implemented, which are responsible for the functionality of the Segment Control block in Figure 15.

RAM Configuration

In the testbenches for the processor and the communication module an asynchronous RAM was modelled (uses no clock for reading and writing data). But this is not possible when implementing the RAM inside the FPGA. When using a distributed RAM (using LUTs) at least the write operation needs to be clocked. As a block RAM is used (preferable when implementing big memories), also the read operation needs to be clocked [13].[8] Additionally the processor and the RAM are communicating over a bidirectional bus. The FPGA features to use tri-state buffers at the I/O pins of the FPGA, but there are no internal tri-state buffers available. Consequently the synthesis tool is automatically converting this bidirectional description into two separate busses for read and write operations.

The RAM blocks inside the used FPGA can be programmed in *Simple Dual Port Mode*. At this mode two address ports are used: one for storing the address of a read operation, and the other one for storing the address of a write operation [13]. In addition, it is possible to use two different clock signals for the read and the write operation. In order to achieve optimal timing for the read operation, CLK 1 and for the write operation CLK 5 could be used as illustrated in Figure 16. The sequence is in connection with the description of the communication module in Section 3.3.

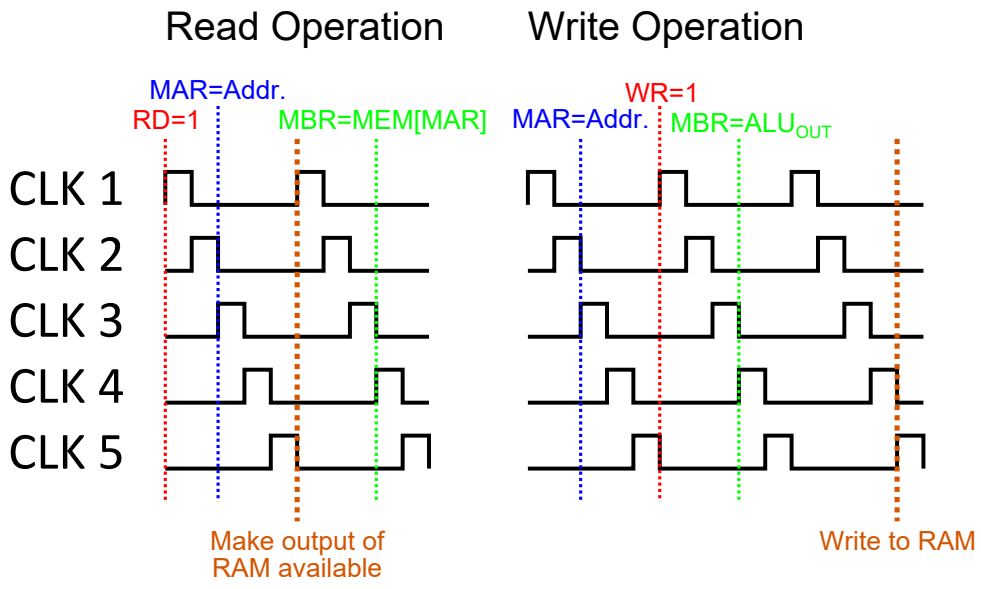

Figure 16: Clocking of the RAM in Simple Dual Port Mode.

[8] There are two possible read modes. One mode needs only one clock cycle for the read operation, whereas the other mode needs two clock cycles (as it uses an additional output register). The used VHDL description sticks to one clock cycle read operation.

Read operation. At the first cycle at CLK 1 the RD signal is set and at CLK 3 the address from which the processor should read is loaded into the MAR. At CLK 4 of the second read cycle the content of the memory is read in by the processor. The optimal moment to make the content of the RAM available is exactly between these two events. Therefore CLK 1 is used to clock the read operation. In Section 3.3 it was described that theoretically it would be possible to achieve that the read operation can be performed completely in one clock cycle. Therefore the clock that makes the correct output of the RAM available would have to appear between CLK 3 and CLK 4 in order to load the MAR and the MBR in one clock cycle.

Write operation. At CLK 3 of the first clock cycle the MAR is loaded with the address at which a value should be written and then in clock cycle two at CLK 4 the MBR is loaded with the content that should be written to RAM. When using CLK 5 to store data to the RAM, the last clock edge of clock cycle three is used to store the data. The write operation at CLK 5 is enabled by a set WR signal. Consequently, this write operation is also performed at CLK 5 of the second write cycle. If this operation already writes the correct data to the RAM the third cycle actually would not be needed. Otherwise at clock cycle two a wrong value might be written to the RAM, which is then corrected at the end of the third write cycle. Such a behaviour could be problematic if the memory location is controlling output peripherals (e.g. output port of a memory mapped I/O). To solve this problem it has to be ensured that at CLK 5 of the second clock cycle already the correct data is read in (whereas the third cycle would not be necessary then). Other possibilities would be to change the microcode in a way that the WR bit is only set at the third clock cycle or delaying the WR signal by one clock signal using hardware (e.g. flip flop in between).

Unfortunately, using the Simple Dual Port Mode of the RAM both ports are used for the communication between the RAM and the processor. Consequently it is not possible to perform another simultaneous read operation which should be used to display the value of a memory location (addressed by the external switches) on the 7-segment displays. To solve this problem different solutions are possible:

- Using flip flops to create the RAM. However, using this method, huge parts of the available resources of the FPGA have to be used.
- Duplicate the memory by using the same memory twice. This can be achieved by performing the write operations to both memories, but using the read port of one memory for the communication between processor and RAM and the read port of the other memory for displaying the memory content of one address on the 7-segment displays. This, however, doubles the needed resources.
- Use the *True Dual Port Mode* of the RAM [13]. As this concept is the one used in this thesis, it is described in more detail below.

In the True Dual Port Mode, each of the ports can be used for reading and writing data from/to RAM. One port is used for the communication between processor and RAM and the other port is just used for reading data to display it on the 7-segment displays. However, using this concept it is not possible to use two different clock signals for the read and write operations of one port. In contrast to Figure 16, CLK 1

was used for both (read and write) operations. This also ensures that only at the third clock cycle at CLK 1 data can be written to RAM (as WR is enabled at CLK 1 of the second clock cycle). Of course this also reduces the maximum allowed delay time until the correct data is available at the output of the RAM.

5.2 Constraints and Hardware Implementation

Constraints are used in the design to specify requirements that have to be met by the design to work correctly in hardware. *Placement Constraints* were used in the design to define the pin connections of the FPGA with the hardware of the Basys 3 Board [14]. No special constraints concerning the input clock (and the output clocks of the MMCM) were specified, as the options of the Clocking Wizard are already used by the tools. After finishing the implementation of the design, the tools reported timing issues. The Total Hold Slack (THS) was a negative value, which indicates, that the hold time of a certain flip flop was not met. To eliminate this problem the clock speed of MMCM outputs was reduced to 10 MHz[9].

Before bringing the design to hardware, further functional and timing simulations could be performed. The simulations after synthesis are used to ensure that the VHDL description is correctly transformed into a netlist which describes a design that still behaves the same way as the design described in VHDL. The post-implementation simulation checks that the behaviour is correct, even after placing the components on the specific FPGA. The timing simulations also consider the delays of components, which is ideal to check for timing issues. Consequently, it might happen that a behavioural simulation is working properly, but in post-synthesis or/and post-implementation simulation the design fails.

It was directly tried to upload the design to the FPGA without performing these additional simulations described above. Using the testing program described in Section 4.4 and checking the memory contents the correct functionality could be verified. However, these simulations would have been very useful if the design had not worked correctly.

[9] It was not exactly analysed what the maximum possible clock frequency would be. Consequently, still a higher clock frequency might be possible without failing timing requirements.

6 Improving Processing Power

During the design process of the processor architecture already many considerations were made to make the architecture as fast as possible. Especially the reset- and interrupt logic was designed to take less time without making rough changes to the processor architecture. Furthermore, it was tried to implement the microcode for all instructions as efficient as possible.

In the following sections it is analysed which changes could be made to the architecture in order to increase the processing power even more.

6.1 Reset- and Interrupt Check

In Figure 2 the operating cycle of the processor was illustrated. As after each executed instruction, a check is performed if a reset- or interrupt condition appeared, the routine is actually polling and not a real interrupt routine [10]. To make this polling as fast as possible, the operation of this check was implemented as described in Section 3.6.

If no reset or interrupt has to be performed, only one line of microcode has to be executed before the next OP-code can be fetched. The average number of microcode lines that one macro instruction needs for execution (inclusive fetching it) is about 19 instructions[10]. Consequently, about 5% of the processing time are needed just for event checking, even if no interrupts are used. Keeping the architecture as it is, and using just polling, this number cannot be reduced further.

As it can be seen from the previous observations, when a processor offers the ability for interrupts, a bad handling of this interrupts can waste much processing power. Consequently, it is important to make interrupts as less time-consuming as possible. The remaining 5% can only be eliminated by using a real interrupt routine (instead of using just polling). This could be achieved by replacing the MMux by a more complex interrupt controller. This controller would have to detect the end of an instruction (this can be done by checking if the address bits in the MIR redirect to the microprogram address of a new instruction fetch) and branch to the fetch of a new OP-code or to serve an incoming interrupt. This controller could also have direct access to the interrupt disable flag in the status register to eliminate further checks when an interrupt is detected.

[10] This number is the average of all instructions of the microprocessor architecture. It does not consider how many microcode lines per instruction are needed for an ordinary program, as this would also depend on the usage of the provided instructions.

6.2 ALU Instructions

The ALU uses four bits to encode the operation that should be performed. Consequently, it is possible to perform 16 different calculations, but only 13 out of these 16 possible encodings are used. Therefore, it would be possible to implement some further operations directly in hardware. For instance, it would be possible to implement an OR operation (disjunction). At the moment a disjunction uses several microprogram cycles as De Morgan's laws are used to calculate the disjunction by using the conjunction (AND) and negation operations of the processor. By implementing the disjunction directly in hardware, only one microinstruction would be needed to calculate the result.

6.3 Microprogram Memory

As the Instruction Register (IR), like all internal registers of the architecture, is an 8-bit register, this allows up to 256 different OP-codes that could be addressed.[11] As more than 100 starting addresses for macro programs are not used, it would be possible to implement further useful macro programs, without needing additional hardware.

For instance, the instructions *Increment* (`INC`) and *Decrement* (`DEC`) for AC, XR and YR could be very useful as often an increment or decrement by 1 is needed (especially for loops). This instructions would accomplish a little improvement in timing, particularly for the 16-bit registers XR and YR. For instance incrementing XR by 1 using only already existing operations, would need the instruction `ADDX #0001`, which has to load two bytes from RAM (the operand containing the number one) in order to add it to the current content of XR, but an instruction `INCX` would use an inherent addressing without the need of loading any operand from the RAM.

In general, the deeper the level on which an operation is implemented, the faster it can be performed. This can be explained with the example of a multiplication [10]. As mentioned in Section 6.2 there are still empty slots for additional operations in the ALU. Therefore, the fastest way of implementing a multiplication would be to implement it directly in hardware (so only one microcode line would be required for the multiplication itself, without considering loading and storing operands)[12]. However, a multiplication of two 8-bit values can have a 16-bit result and, with the current hardware structure, it is not possible to have a 16-bit ALU output and store two bytes at one clock cycle in two different internal registers. However, it would be possible to write a macroprogram that performs a multiplication by using shifting operations, additions, and branches in microcode. There would be, however,

[11] The total number is a little bit lower, as the fetch-, reset-, and interrupt routines are placed at the beginning of the MPM and fill a few starting addresses. A few other instructions consume some space of not used OP-code starting addresses too. However, by placing this code between other instructions it would be possible to use all 256 starting points.

[12] In addition it has to be considered, that multiplications of signed values, unsigned values and a signed and an unsigned value are slightly different. Therefore also e.g. the AVR microncontrollers have three different multiplication operations [15].

still the problem that the result is a 16-bit value. It would be possible to use the mechanism which also the AVR microcontrollers uses, where two internal scratch registers (R0 and R1) are used to save the result of a multiplication [15]. As an AVR microcontroller is a register machine, all assembly instructions can directly access a bunch of registers. However, the processor considered in this thesis is an accumulator machine, so only the accumulator (and some special registers like for instance XR) can be accessed by the assembly language. If, for instance, the auxiliary registers U and V should be used as registers for a multiplication result, special macroprograms that can load values from this internal registers would be needed. If no special macro instructions for multiplication are considered in the processor architecture, a function that achieves this operation has to be written by using the macro instructions to add, shift and perform branches. Of course this is the slowest method to perform a multiplication.

6.4 Clock Speed

Last but not least, the easiest way of increasing the timing efficiency of a processor is to increase its clock speed. But of course there are limitations. If the signal propagation delays come in the range of the clock period, this can lead to misbehaviour [10].

A critical part of this design is the communication with the RAM (especially if it is implemented as an external memory and not inside the FPGA). If the RAM is not fast enough to save data from the processor or is not able to make data available that should be read in by the processor fast enough, the clock speed has to be decreased (or a faster RAM is needed).

Another critical part of the design is the slowest operation of the ALU. The result of the ALU has to be ready as soon as it is loaded into another register (also the flags have to be calculated correctly).

6.5 Conflicting Improvements

As already mentioned in Section 6.3, it is more effective to implement more complex operations directly in hardware to increase the processing power. But it has also to be considered that more complex instructions can limit the possible clock speed to a lower frequency. Consequently, it depends much on the usage of the processor if a certain improvement really helps to increase the processing power of a distinct application. For instance, for some applications it might be useful to have included floating point operations, even if the clock speed is lower, as these instructions are needed frequently in the program. However, using the same processor with a program which is not using any of these floating point operations just decreases the execution speed of the program.

7 Conclusion and Outlook

In this thesis a design of a 8-bit microprocessor was described successfully using VHDL and then implemented on an FPGA. In the following sections some possible further steps are described to completely check the correctness of the design and further possible improvements to increase the usefulness of the processor (beside already described possibilities to increase the precessing power in Chapter 6).

Simulations and Clock Speed

Although most of the instructions of the processor were tested in behavioural simulation, not every special case was considered (especially if all status bits are set correctly). This would require further testing programs.

During the design process, the internal clock speed was reduced to 10 MHz, as after performing implementation VIVADO warned about timing failures in the design. Actually, it might be possible that the design would work in hardware, but the tools detected timing issues because of missing constraints. By adapting the constraints and performing post-synthesis and post-implementation simulations it can be analysed how far the clock speed can be increased.

If instead of the internal RAM, an external RAM is used, it would also be very important to constrain the inputs and output port connections of the RAM (e.g., delays) to have valid timing checks.

Assembler

To test the processor, the content of the RAM was written by hand (using machine code) to achieve a certain functionality. In order to make it easier for other users to develop programs for this processor, it would be useful to have an assembler (or even a compiler for higher languages) that translates the assembly language into the needed machine code. This would make writing code much faster and much better readable. This would also speed up the writing of further programs for testing.

Microcontroller

At the moment the processor can only communicate with the (external or internal) RAM. Therefore, it might be useful to have the possibility to communicate also with further peripherals. This could be achieved for instance by using a memory mapped IO. A part of the complete address space could be used for special registers that are controlling peripherals[13].

When using a RAM directly inside the FPGA it might be a good idea to change the communication module and the connections with processor and RAM in order to use separate busses for read and write operations instead of just relying upon the

[13] To make it possible to read (and also write) in parallel from this registers without addressing them it is not possible to use distributed or block RAM. Instead flip flops have to be used.

synthesis tool, that they will transform the bidirectional description in an optimal way. Probably, this also reduces the synthesis time.

At the moment the content of the RAM has to be part of the bitstream. Therefore, to change the content, a completely new bitstream has to be generated. Consequently, it would be useful to have additional hardware (ideally inside the FPGA) that allows to change the content of the RAM without loading a new bitstream to the FPGA (like programming an AVR microcontroller).

Bibliography

[1] *Basys 3™ FPGA Board Reference Manual*. Digilent Inc. Mar. 23, 2017. URL: https://reference.digilentinc.com/_media/reference/programmable-logic/basys-3/basys3_rm.pdf (visited on 10/22/2017).

[2] Schiffmann W. *Technische Informatik 2. Grundlagen der Computertechnik*. 5th ed. Springer, 2005.

[3] Kafig W. *VHDL 101. Everything you need to know to get started*. Elsevier Inc., 2011.

[4] Molitor P. and Ritter J. *VHDL. Eine Einführung*. Pearson Studium, 2004.

[5] *Clocking Wizard v5.3. LogiCORE IP Product Guide*. PG065. Xilinx Inc. Oct. 5, 2017. URL: https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v5_3/pg065-clk-wiz.pdf (visited on 10/22/2017).

[6] *Vivado Design Suite User Guide. Designing IP Subsystems Using IP Integrator*. UG994 (v2016.2). Xilinx Inc. June 8, 2016. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug994-vivado-ip-subsystems.pdf (visited on 10/22/2017).

[7] *7 Series FPGAs Clocking Resources*. UG472 (v1.13). Xilinx Inc. Mar. 1, 2017. URL: https://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf (visited on 10/22/2017).

[8] Ken Chapman. *Get Smart About Reset: Think Local, Not Global*. WP272 (v1.0.1). Xilinx Inc., Mar. 7, 2008. URL: https://www.xilinx.com/support/documentation/white_papers/wp272.pdf (visited on 10/22/2017).

[9] *Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V. DATASHEET*. Atmel Corporation (Parent: Microchip Technology). 2014. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf (visited on 02/10/2018).

[10] Gehrke W. et al. *Digitaltechnik. Grundlagen, VHDL, FPGAs, Mikrocontroller*. 7th ed. Springer, 2016.

[11] *Debounce Logic Circuit (with VHDL example)*. last changed: 09/08/2017. eewiki (maintained by Digi-Key). URL: https://eewiki.net/pages/viewpage.action?pageId=4980758 (visited on 10/18/2017).

[12] *Vivado Design Suite User Guide. Programming and Debugging*. UG908 (v2016.2). Xilinx Inc. June 8, 2016. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug908-vivado-programming-debugging.pdf (visited on 11/08/2017).

[13] *7 Series FPGAs Memory Resources*. UG473 (v1.12). Xilinx Inc. Sept. 27, 2016. URL: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf (visited on 11/08/2017).

[14] *Vivado Design Suite User Guide. Using Constraints*. UG903 (v2016.2). Xilinx Inc. June 8, 2016. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug903-vivado-using-constraints.pdf (visited on 11/08/2017).

[15] *AVR Instruction Set Manual*. Atmel Corporation (Parent: Microchip Technology). Nov. 2016. URL: http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf (visited on 02/10/2018).

Appendix

A Microprocessor Control Information

1. Register der Microarchitektur

Adresse HEX	Adresse DEC	Adresse BIN	Register	Lesen	Schreiben	Funktion	
0	00	0 0 0 0 0	YR_H	$\mathbb{A}$-Bus $\mathbb{B}$-Bus	$\mathbb{C}$-Bus	Programmiermodell der Makroarchitektur	
1	01	0 0 0 0 1	YR_L				
2	02	0 0 0 1 0	XR_H				
3	03	0 0 0 1 1	XR_L				
4	04	0 0 1 0 0	SP_H				
5	05	0 0 1 0 1	SP_L				
6	06	0 0 1 1 0	PC_H				
7	07	0 0 1 1 1	PC_L				
8	08	0 1 0 0 0	AC				
9	09	0 1 0 0 1	CC				
10	0A	0 1 0 1 0	EAR_H	$\mathbb{A}$-Bus $\mathbb{B}$-Bus	$\mathbb{C}$-Bus	Register für effektive Adress	
11	0B	0 1 0 1 1	EAR_L				
12	0C	0 1 1 0 0	OPR_H			Operandenregister (16-Bit Operationen)	
13	0D	0 1 1 0 1	OPR_L				
14	0E	0 1 1 1 0	IR	Befehlsdek.	$\mathbb{C}$-Bus	Instruktions Register	
15	0F	0 1 1 1 1	EVENT	$\mathbb{A}$-Bus	Extern/$\mathbb{C}$-Bus	Event Register	
16	10	1 0 0 0 0	U	$\mathbb{A}$-Bus $\mathbb{B}$-Bus	$\mathbb{C}$-Bus	Hilfsregister	
17	11	1 0 0 0 1	V				
18	12	1 0 0 1 0	W				
19	13	1 0 0 1 1	Z				
20	14	1 0 1 0 0	10000000	$\mathbb{A}$-Bus $\mathbb{B}$-Bus	—	0x80	Kein Event
21	15	1 0 1 0 1	11111011			0xFB (-5)	
22	16	1 0 1 1 0	11111100			0xFC (-4)	
23	17	1 0 1 1 1	11111101			0xFD (-3)	
24	18	1 1 0 0 0	11111110			0xFE (-2)	
25	19	1 1 0 0 1	11111111			0xFF (-1)	
26	1A	1 1 0 1 0	00010000			0x10 (+16)	Interr. Mask
27	1B	1 1 0 1 1	00001000			0x08 (+8)	Negative Mask
28	1C	1 1 1 0 0	00000100			0x04 (+4)	Zero Mask
29	1D	1 1 1 0 1	00000010			0x02 (+2)	Overflow Mask
30	1E	1 1 1 1 0	00000001			0x01 (+1)	Carry Mask
31	1F	1 1 1 1 1	00000000			0x00 (0)	

2. OP-CODE

Mikro Instr.	Operation	Beschreibung
0	$\overline{\text{J}}$: MPC ←MPC + 1	Inkrementiere MPC oder
	J: MPC ←Adresse	lade Sprungadresse (s. COND)
1	MPC ←f(IR)	Lade Startadresse für Mikroprogramm

3. A-MUX

Mikro Instr.	Operation	Beschreibung
0	ALU_A←A_{reg}	ALU_A-Eingang von A-Bus
1	ALU_A←MBR	ALU_A-Eingang von Memory

4. COND

Mikro Instr.	Operation	Beschreibung
0 0	J ←0	Kein Sprung
0 1	N: J ←1	Sprung, wenn N=1
1 0	Z: J ←1	Sprung, wenn Z=1
1 1	J ←1	Unbedingter Sprung

5. ALU

Mikro Instr.	Operation	N	Z	V	C	Beschreibung
0 0 0 0	$ALU_{out} \leftarrow ALU_A$	↕	↕	0	–	Transfer
0 0 0 1	$ALU_{out} \leftarrow \overline{ALU_A}$	↕	↕	0	1	Negation
0 0 1 0	$ALU_{out} \leftarrow ALU_A + B_{reg} + C$	↕	↕	↕	↕	Addition mit Carry
0 0 1 1	$ALU_{out} \leftarrow ALU_A \wedge B_{reg}$	↕	↕	0	–	Logisches UND
0 1 0 0	$ALU_{out} \leftarrow$ rol ALU_A	↕	↕	↕	↕	Rotation (mit Carry)
0 1 0 1	$ALU_{out} \leftarrow$ ror ALU_A	↕	↕	–	↕	Rotation (mit Carry)
0 1 1 0	C $\leftarrow$0	–	–	–	0	Lösche Carry
0 1 1 1	C $\leftarrow$1	–	–	–	1	Setze Carry
1 0 0 0	$ALU_{out} \leftarrow (N,Z,V,C) \circ ALU_A$	–	–	–	–	Zum Aktualisieren von CC
1 0 0 1	$ALU_{out} \leftarrow (N,Z,V) \circ ALU_A$	–	–	–	–	
1 0 1 0	$ALU_{out} \leftarrow (N,Z,C) \circ ALU_A$	–	–	–	–	
1 0 1 1	$ALU_{out} \leftarrow (Z) \circ ALU_A$	–	–	–	–	
1 1 0 0	$ALU_{out} \leftarrow (C) \circ ALU_A$	–	–	–	–	

6. MBR–MAR–RD–WR

Mikro Instr.	Operation	Beschreibung	
0 0 0 0		Kein Zugriff auf externen Speicher	
0 1 1 0	$MAR_L \leftarrow A_{reg}$ $MAR_H \leftarrow B_{reg}$	(1. Zyklus)	Lese extern Speicher
1 0 1 0	MBR $\leftarrow$M[MAR]	(2.Zyklus)	
0 1 0 0	$MAR_L \leftarrow A_{reg}$ $MAR_H \leftarrow B_{reg}$	(1. Zyklus)	Schreibe extern Speicher
1 0 0 1	MBR $\leftarrow ALU_{out}$	(2.Zyklus)	
0 0 0 1	M[MAR] $\leftarrow$MBR	(3.Zyklus)	

7. ENC

Mikro Instr.	Operation	Beschreibung
0		Keine Speicherung von ALU_{out} in Register
1	Reg[$\mathbb{C}$] $\leftarrow ALU_{out}$	Speicherung von ALU_{out} in Register selektiert von $\mathbb{C}$

8. Registermodell der Makroarchitektur

High-Byte	Low-Byte
YR_H	YR_L
XR_H	XR_L
SP_H	SP_L
PC_H	PC_L
	AC
	CC

Index-register	YR (16 bit)
Index-register	XR (16 bit)
Stack-Pointer	SP (16 bit)
Program-Counter	PC (16 bit)
Accumulator	AC (8 bit)
Status-register	CC (8 bit)

(CC: Condition-Code register)

Bit	7	6	5	4	3	2	1	0
	-	-	-	I	N	Z	V	C

Flags: **I**: Interrupt-Disable, **N**: Negative, **Z**: Zero, **V**: Overflow, **C**: Carry

$\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$: Je 5 Bit zur Adressierung der 32 Register über die Adreßbusse $\mathbb{A}$, $\mathbb{B}$ und $\mathbb{C}$

ADRESSE: 12 Bit zur Adressierung des nächsten Mikrobefehls bei Sprunganweisungen im Mikroprogramm

B Microprocessor Instruction Set

Befehl		Mnem	OP-Code für Adressierart (a) Inh.	(b) Imm.	(c) Abs.	(d) @x	(e) @y	(f) Rel.	Statusbits N	Z	V	C
Lade	AC	**LDA**	·	86	B6	A6	96	·	↕	↕	0	–
	XR	**LDX**	·	8E	BE	AE	9E	·	↕	↕	0	–
	YR	**LDY**	·	C0	F0	E0	D0	·	↕	↕	0	–
	SP	**LDS**	·	C1	F1	E1	D1	·	↕	↕	0	–
Speichere	AC	**STA**	·	·	B7	A7	97	·	↕	↕	0	–
	XR	**STX**	·	·	BF	AF	9F	·	↕	↕	0	–
	YR	**STY**	·	·	F2	E2	D2	·	↕	↕	0	–
	SP	**STS**	·	·	F3	E3	D3	·	↕	↕	0	–
Addiere	AC	**ADDA**	·	8B	BB	AB	9B	·	↕	↕	↕	↕
	XR	**ADDX**	·	C4	F4	E4	D4	·	↕	↕	↕	↕
	YR	**ADDY**	·	C5	F5	E5	D5	·	↕	↕	↕	↕
	SP	**ADDS**	·	C6	F6	E6	D6	·	↕	↕	↕	↕
Subtrahiere	AC	**SUBA**	·	80	B0	A0	90	·	↕	↕	↕	↕
	XR	**SUBX**	·	C7	F7	E7	D7	·	↕	↕	↕	↕
	YR	**SUBY**	·	C8	F8	E8	D8	·	↕	↕	↕	↕
	SP	**SUBS**	·	C9	F9	E9	D9	·	↕	↕	↕	↕
Vergleiche	AC	**CMPA**	·	81	B1	A1	91	·	↕	↕	↕	↕
	XR	**CMPX**	·	8C	BC	AC	9C	·	↕	↕	↕	↕
	YR	**CMPY**	·	CA	FA	EA	DA	·	↕	↕	↕	↕
	SP	**CMPS**	·	CB	FB	EB	DB	·	↕	↕	↕	↕
Addiere mit Übertrag AC		**ADCA**	·	89	B9	A9	99	·	↕	↕	↕	↕
Subtrahiere mit Übertrag AC		**SBCA**	·	82	B2	A2	92	·	↕	↕	↕	↕
Lösche AC		**CLRA**	4F	·	·	·	·	·	0	1	0	0
2er Kompl AC		**NEGA**	40	·	·	·	·	·	↕	↕	↕	↕
1er Kompl AC		**COMA**	43	·	·	·	·	·	↕	↕	0	1
Und AC		**ANDA**	·	84	B4	A4	94	·	↕	↕	0	–
Oder AC		**ORA**	·	8A	BA	AA	9A	·	↕	↕	0	–
Exor AC		**EORA**	·	88	B8	A8	98	·	↕	↕	0	–
Und CC		**ANDC**	·	1C	·	·	·	·	direkt beeinflusst			
Oder CC		**ORC**	·	1A	·	·	·	·	direkt beeinflusst			
Schiebe arithmetisch links		**ASLA**	48	·	·	·	·	·	↕	↕	↕	↕
Schiebe arithmetisch rechts		**ASRA**	47	·	·	·	·	·	↕	↕	–	↕
Schiebe logisch links		**LSLA**	48	·	·	·	·	·	↕	↕	↕	↕
Schiebe logisch rechts		**LSRA**	44	·	·	·	·	·	0	↕	–	↕
Rotiere durch Übertrag links		**ROLA**	49	·	·	·	·	·	↕	↕	↕	↕
Rotiere durch Übertrag rechts		**RORA**	46	·	·	·	·	·	↕	↕	–	↕
Unbedingter Sprung		**JMP**	·	·	7E	6E	0E	·	–	–	–	–
Unterprogramm Aufruf		**JSR**	·	·	37	·	·	·	–	–	–	–
Unterprogramm Ende		**RTS**	39	·	·	·	·	·	–	–	–	–
Keine Operation		**NOP**	12	·	·	·	·	·	–	–	–	–
Software Interrupt		**SWI**	3F	·	·	·	·	·	–	–	–	–
Interrupt Routine Ende		**RTI**	3B	·	·	·	·	·	alte Bits vom Stack			
Verzweige immer		**BRA**	·	·	·	·	·	20	–	–	–	–
Verzweige wenn	C = 0	**BBC**	·	·	·	·	·	24	–	–	–	–
	C = 1	**BCS**	·	·	·	·	·	25	–	–	–	–
	V = 0	**BVC**	·	·	·	·	·	28	–	–	–	–
	V = 1	**BVS**	·	·	·	·	·	29	–	–	–	–
	N = 0	**BPL**	·	·	·	·	·	2A	–	–	–	–
	N = 1	**BMI**	·	·	·	·	·	2B	–	–	–	–
	Z = 0	**BNE**	·	·	·	·	·	26	–	–	–	–
	Z = 1	**BEQ**	·	·	·	·	·	27	–	–	–	–

Befehl		**Mnem**	**OP-Code für Adressierart** (a) Inh.	(b) Imm.	(c) Abs.	(d) @x	(e) @y	(f) Rel.	**Statusbits** **N**	**Z**	**V**	**C**
Speichere auf Stack	AC	**PSHA**	50	·	·	·	·	·	–	–	–	–
	XR	**PSHX**	51	·	·	·	·	·	–	–	–	–
	YR	**PSHY**	52	·	·	·	·	·	–	–	–	–
	SP	**PSHS**	53	·	·	·	·	·	–	–	–	–
	PC	**PSHP**	54	·	·	·	·	·	–	–	–	–
	CC	**PSHC**	55	·	·	·	·	·	–	–	–	–
Hole vom Stack	AC	**PULA**	58	·	·	·	·	·	–	–	–	–
	XR	**PULX**	59	·	·	·	·	·	–	–	–	–
	YR	**PULY**	5A	·	·	·	·	·	–	–	–	–
	SP	**PULS**	5B	·	·	·	·	·	–	–	–	–
	PC	**PULP**	5C	·	·	·	·	·	–	–	–	–
	CC	**PULC**	5D	·	·	·	·	·	Statusbits vom Stack			

a) Inhärent
b) Immediate
c) Absolut
d) @(x + Offset)
e) @(y + Offset)
f) Relativ

Statusbits:
↕ Statusbit wird beeinflut
– Statusbit unverändert
0 Lösche Statusbit
1 Setze Statusbit

C Technische Informatik Labor Introduction

VIVADO und VHDL Einführung

Bernhard Vacarescu

Dieses Dokument soll einen Überblick über die Entwicklungsumgebung von VIVADO (Version 2016.2) geben. Es wird im Folgenden der Designflow eines einfachen FPGA-Designs beschrieben, also welche Schritte notwendig sind, um ein Design zu entwerfen und es schließlich auf einem FPGA implementieren zu können. Nebenbei werden einige wichtige Konstrukte der Hardwarebeschreibungssprache VHDL erläutert, welche im Rahmen dieser Einführung benötigt werden. Da es sich hier um eine möglichst kompakte Einführung handeln soll, es sich bei VIVADO jedoch um ein sehr umfangreiches Softwarepaket handelt, und VHDL eine Programmiersprache mit einem sehr umfangreichen Befehlssatz ist, kann hier natürlich nicht auf alles detailliert eingegangen werden. Für umfangreichere Informationen sei daher auf weiterführende Literatur verwiesen. In Kapitel 4 werden noch ein paar wichtige Informationen zu VHDL inklusive spezieller Befehle angeführt, welche im Einführungsbeispiel nicht behandelt werden, jedoch für die Laboreinheit trotzdem von Bedeutung sein können.

1 Einführung

FPGA

Kurz beschrieben handelt es sich bei einem FPGA um einen Baustein, dessen Hardware programmiert werden kann. Die Hauptbestandteile eines FPGAs sind einzelne Logikblöcke (*Complex Logic Blocks*, kurz CLBs) und ein Verbindungsnetzwerk (*Switch Matrix*) welches Logikblöcke miteinander verbindet. Ebenso sind *I/O-Blöcke* von Bedeutung, welche die Verbindung zu den Pins des FPGA herstellen (dabei können noch zusätzliche Funktionen in den Blöcken vorhanden sein) und ebenfalls mit dem Verbindungsnetzwerk verbunden sind.

Logikblöcke können intern einen unterschiedlichen Aufbau haben, ein wichtiger Bestandteil von ihnen sind *Lookup-Tables* (LUTs). Durch diese lassen sich verschiedene logische Funktionen beschreiben. Bei den LUTs handelt es sich um einen 1-bit breiten Speicherblock, dessen Inhalt programmiert werden kann. Über die Adressleitungen (Eingänge des LUTs) wird nun die adressierte Speicherzelle am Ausgang des LUTs ausgegeben. Ein weiterer wichtiger Bestandteil von CLBs sind Flip-Flops, damit die LUTs nicht nur für kombinatorische sondern auch für getaktete Schaltungen verwendet werden können. Über das Verbindungsnetzwerk werden die genauen Verbindungen zwischen einzelnen Logikblöcken definiert. Es können also Schalter programmiert werden, welche die Verbindungen von CLBs untereinander herstellen (es kann nicht jeder Logikblock mit jedem anderen verbunden werden). Dies geschieht zum Beispiel über Multiplexer. Die Bits der Select-Leitungen können programmiert werden und setzten so die gewünschten Verbindungen um.

Ein sogenannter *Bitstream* enthält die gesamte Information darüber, welche logischen Funktionen die LUTs darstellen und wie das Verbindungsnetzwerk aufgebaut ist. Dieser Bitstream wird auf den FPGA geladen, wodurch seine interne Struktur nun festgelegt ist.

VHDL

VHDL steht für VHSIC Hardware Description Language (VHSIC ist wiederum eine Abkürzung für Very High Speed Integrated Circuits). Sie wurde in den frühen 80er Jahren im Auftrag des Verteidigungsministeriums der Vereinigten Staaten entwickelt.

Die Intention von VHDL war, eine Sprache zum Beschreiben des Verhaltens von Schaltungen zur Dokumentation und zur Simulation zu konstruieren. Später wurde auch die Schaltungssynthese zu einem Verwendungszweck.

Wichtig ist, dass nicht alle Bestandteile des VHDL-Befehlssatzes auf einem FPGA implementiert werden können. Jedoch ist auch der nicht synthetisierbare Teil des Befehlssatzes für die sogenannten *Test Benches* von Bedeutung, welche verwendet werden, um zu testen, ob ein beschriebenes Design auch wie gewünscht funktioniert.

Der Designflow

In Abbildung 1 erkennt man die Herangehensweise zum Erstellen einer Schaltung auf einem FPGA. Zuerst muss das Design in einer Hardwarebeschreibungssprache wie eben VHDL beschrieben werden. Das fertige Design sollte man durch Simulationen testen, um zu überprüfen, ob die Schaltung auch wie gewollt funktioniert. Bei großen Designs sollte nicht nur die endgültige Schaltung getestet werden, sondern es sollten auch die einzelne Module, aus denen die endgültige Schaltung besteht, getestet werden, da dadurch eine mögliche Fehlersuche erleichtert wird. Bei der *Schaltungssynthese* wird durch das Entwicklungstool versucht, die beschriebene Schaltung mit Bausteinen, die auf einem FPGA zur Verfügung stehen, aufzubauen. Es wird dabei eine *Netzliste* erstellt, welche die Struktur der Schaltung beschreibt. Bei der folgenden *Implementierung* werden konkrete Verbindungen der Bausteine auf dem verwendeten FPGA festgelegt. Auch nach Synthese und Implementierung ist es sinnvoll Simulationen durchzuführen, um festzustellen, ob sich die Schaltung noch wie gewünscht verhält, und besonders ob zeitliche Bedingungen (wie zum Beispiel Setup- und Hold-Time bei Flip-Flops) vom Design eingehalten werden. Schlussendlich kann ein *Bitstream* erstellt und auf den FPGA geladen werden, welcher die Hardware des FPGA programmiert. Nun muss noch getestet werden, ob die korrekte Funktion auch für die in Hardware realisierte Schaltung gegeben ist. Abschließend kann man noch weitere Verbesserungen an dem Design (wie zum Beispiel eine Verringerung des Energieverbrauchs) durchführen.

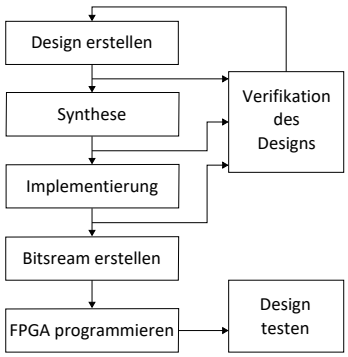

Abbildung 1: FPGA Designflow

Im **Flow Navigator** von VIVADO, der sich auf der linken Seite befindet, wenn man bereits ein Projekt geöffnet hat, erkennt man auch einen ähnlichen Aufbau zum eben beschriebenen Designflow. Im Groben muss diese Spalte von oben nach unten durchgearbeitet werden, um vom anfänglichen Schaltungsentwurf zu einer Implementierung auf einem FPGA zu gelangen. Durch das Durchführen von Änderungen an dem Design springt man im Designflow natürlich immer wieder ein paar Schritte zurück. Je nachdem wie weit im Designentwurf man sich gerade im Designflow befindet, sind entsprechende Unterpunkte im Flow Navigator verfügbar. Im folgenden werden ein paar wichtige Punkte aus dem Flow Navigator näher erläutert.

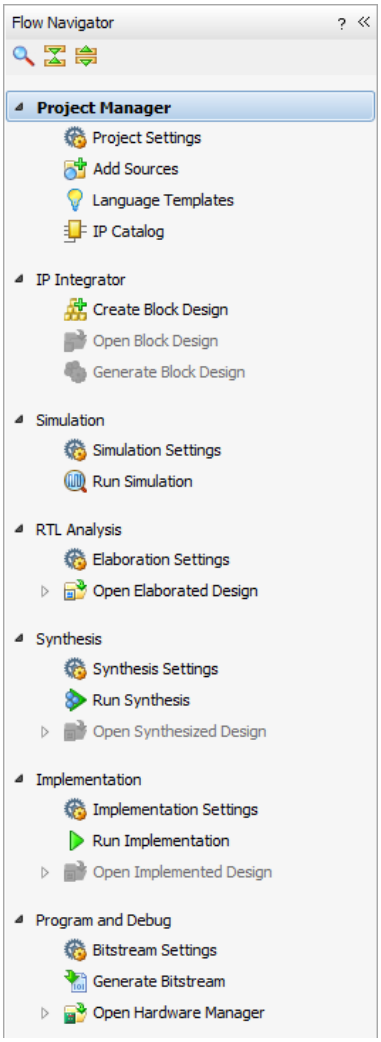

Abbildung 2: Flow Navigator

Project Manager: Unter *Add Sources* werden dem Projekt neue Dateien hinzugefügt. Im *IP Catalog* befinden sich viele vorgefertigte Module (*IP Cores*)[1], welche über eine GUI konfiguriert und dem Design hinzugefügt werden können. Es wird ein sogenannter *Wrapper* erstellt, welcher die Schnittstelle zum erstellten Modul darstellt. Der ganze notwendige Code des Moduls wird jedoch automatisch erstellt.

IP Integrator: Es kann ein Block Design erstellt werden, bei dem mehrere IP Cores hinzugefügt werden können und deren Verdrahtung grafisch festgelegt werden kann. Vom gesamten Block Design kann wieder ein Wrapper erstellt werden.

Simulation: Hier können unter *Run Simulation* verschiedenen Simulationen gestartet werden.

RTL Analysis: Hier kann das *Elaborated Design* (Darstellung des Designs vor der Synthese) betrachtet werden. Im Schaltplan (*Schematic*) findet die Darstellung mit Gattern, Flip Flops, Multiplexern usw. statt.

Synthesis: Hier kann mittels *Run Synthesis* nun die eigentliche Synthese durchgeführt werden. In der Darstellung des Schaltplanes sind nun nur Bauteile vorhanden, die innerhalb des FPGAs vorhanden sind.

Implementation: Unter *Run Implementation* kann die Implementierung gestartet werden, bei der die konkreten Verbindungen innerhalb des FPGAs festgelegt werden.

Program and Debug: Hier kann vom fertigen Design der Bitstream erstellt werden (*Generate Bitstrem*), welcher im *Hardware Manager* dann auf den FPGA geladen werden kann.

Nach Durchführung von Synthese und Implementierung lassen sich noch viele zusätzliche Optionen auswählen. Dazu zählen unter anderem das Hinzufügen von *Contraints* (Bedingungen, die das Design einhalten soll) und viele Möglichkeiten um zu überprüfen, ob das Design die Anforderungen einhalten kann.

[1] Es ist auch möglich aus selbst erstellten Modulen IP Cores zu erstellen.

Verwendete Hardware

Im Labor wird das Basys 3 Board von Digilent verwendet. Dieses beinhaltet als FPGA einen Artix®-7 von Xilinx (genaue Bezeichnung: XC7A35T-1CPG236C).

Auf dem Board befinden sich bereits eine Vielzahl von Funktionen, die mit bestimmten Pins des FPGA verbunden sind. Auf dem Board befinden sich unter anderem Schalter, Taster, LEDs, 7-Segment-Anzeigen sowie ein Oszillator, der ein 100 MHz Clock-Signal erzeugt.

Eine detaillierte Dokumentation, in der die einzelne Bestandteile des Boards und deren Ansteuerungsmöglichkeiten beschrieben sind, findet man in der Dokumentation des Boards.

2 Designentwurf in VHDL

Das Design

Die gesamte Einführung beschäftigt sich mit dem in Abbildung 3 dargestellten Entwurf. Es sei angemerkt, dass das Gesamtverhalten der Schaltung viel einfacher beschrieben werden könnte als im Folgenden dargestellt, jedoch bietet dieses Design die Möglichkeit, viele der Konstrukte von VHDL näher zu bringen.

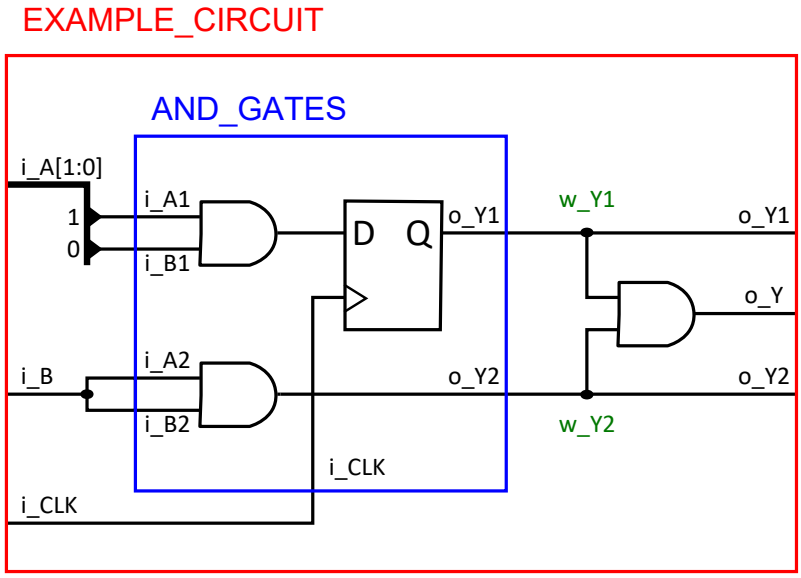

Abbildung 3: Mit VHDL zu entwerfendes Beispieldesign

Man erkennt die Schaltung `EXAMPLE_CIRCUIT`, welche drei Eingänge besitzt: einen 2-bit Eingang `i_A[1:0]`[2], einen 1-bit Eingang `i_B` und einen Eingang für das Taktsignal `i_CLK`. Ebenso besitzt die Schaltung drei 1-bit Ausgänge (`o_Y`, `o_Y1` und `o_Y2`).

[2] Der Eingang `i_A` wurde zu VHDL-Demonstrationszwecken als Bus ausgeführt.

Innerhalb des Moduls `EXAMPLE_CIRCUIT` ist weiters das Modul `AND_GATES` eingebunden. Dieses berechnet jeweils zwischen zwei Eingängen (zwischen `i_A1` und `i_B1` und zwischen `i_A2` und `i_B2`) eine UND-Verknüpfung und gibt das Ergebnis am jeweiligen Ausgang (`o_Y1` bzw. `o_Y2` des Moduls `AND_GATES`) aus. Hinter das erste UND-Gatter ist jedoch noch ein D-Flip-Flop geschaltet (soll auf eine steigende Taktflanke reagieren). Somit kann sich der Ausgang `o_Y1` nur bei einer steigenden Taktflanke von `i_CLK` ändern. Die Schaltung beinhaltet somit sowohl kombinatorische als auch getaktete Logik.

Weiters ist noch die genaue Verschaltung des Bausteines `AND_GATES` mit den Ein- und Ausgängen von `EXAMPLE_CIRCUIT` erkennbar. Während die Ausgänge `o_Y1` und `o_Y2` des Moduls `AND_GATES` durchgeschleift werden, wird zusätzlich eine UND-Verknüpfung zwischen den beiden berechnet und das Ergebnis schließlich bei `o_Y` ausgegeben. Im gesamten folgenden VHDL-Code werden die Signalbezeichnungen aus Abbildung 3 verwendet.

Projekt in VIVADO erstellen und erste Dateien hinzufügen

Nach dem Starten von VIVADO kann unter **Quick Start** mittels **Create New Project** ein neues Projekt erstellt werden.

Abbildung 4: Neues Projekt erstellen

Der folgende Wizard führt einen durch die Schritte zur Erstellung eines neuen Projektes. Zuerst kann der Projektname (es wurde `VHDL_Introduction` gewählt) und der Speicherort auswählt werden. Im Dateinamen und im restlichen Pfad sollte sich nirgends ein Leerzeichen befinden. Auf der darauf folgenden Seite wählen wir **RTL Project** als **Project Type**. Sofern man die Checkbox **Do not specify sources at this time** nicht anhakt, können im folgenden Fenster bereits vor der Projekterstellung bestehende Dateien zum Projekt hinzugefügt werden bzw. neue Dateien erstellt werden.

Wir haken die Checkbox nicht an und können im folgenden Fenster durch einen Klick auf **Create File** gleich unsere erste Datei für das Modul `AND_GATES` hinzufügen. Hierbei sollte darauf geachtet werden, dass als **File type** bei der Erstellung eines neuen Files **VHDL** auswählt ist. Ebenso sollte im Hauptfenster **Add Sources** bei **Target language** und **Simulator language** als Sprache VHDL ausgewählt werden.

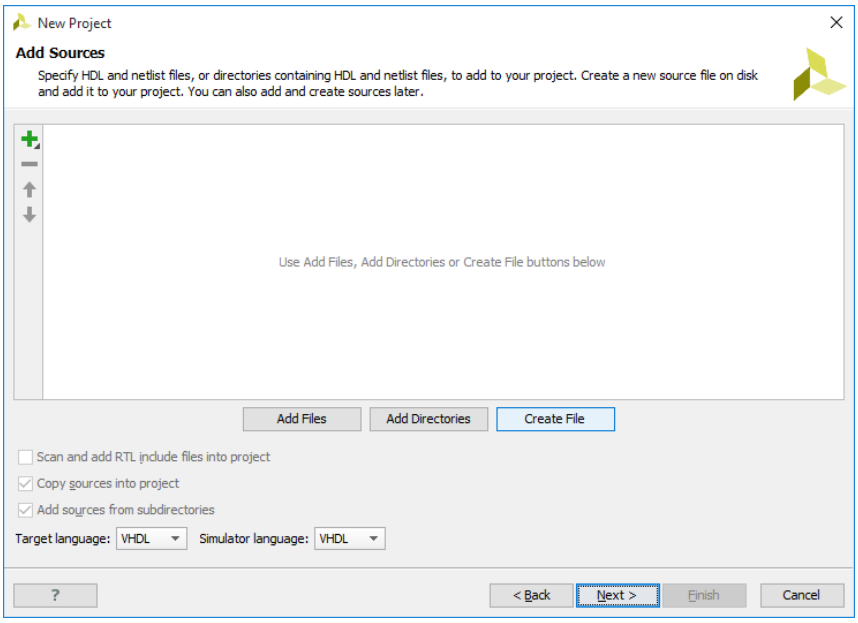

Abbildung 5: Dem Projekt Dateien hinzufügen

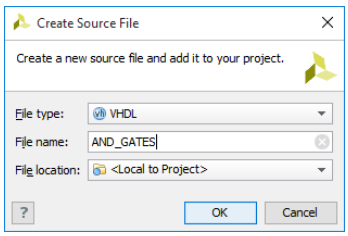

Abbildung 6: Datei für das Modul `AND_GATES` hinzufügen

Bei den weiteren Seiten **Add Existing IP** und **Add Constraints** werden wir noch keine Dateien hinzufügen bzw. erstellen.

Auf der Seite **Default Part** kann nun der verwendete FPGA ausgewählt werden. Am einfachsten geht dies entweder durch Suchen des Bauteilnamens oder Eingabe der korrekten **Filter**-Werte (siehe Abbildung 7). In unserem Fall ist dies der FPGA, der auf dem Basys 3 Board vorhanden ist. Nachdem man alles eingestellt hat, kann unter **Finish** die Projekterstellung abgeschlossen werden.

Da wir bei der Projekterstellung bereits unsere erste Datei für das Modul `AND_GATES` zu unserem Projekt hinzugefügt haben, öffnet sich gleich nach der Projekterstellung ein Fenster **Define Module**, bei dem wir die Schnittstellen unseres Moduls festlegen können. Weiters kann ein

Name für die *Entity* und die *Architecture* des Moduls festgelegt werden (was diese beiden Begriffe bedeuten wird später noch erläutert). Unter **I/O Port Definitions** erstellen wir die Schnittstellen nun so wie in unserem Design in Abbildung 3 festgelegt (siehe Abbildung 8). Es ist nicht zwingend notwendig die Schnittstellen hier einzugeben, jedoch wird dadurch ein Teil des VHDL-Codes automatisch erstellt und man erspart sich das manuelle Schreiben.

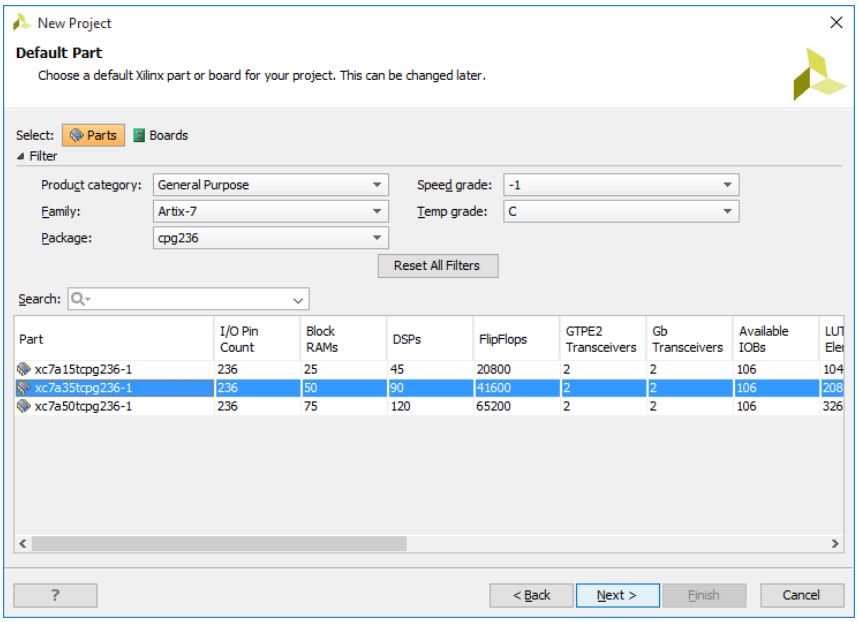

Abbildung 7: FPGA auswählen

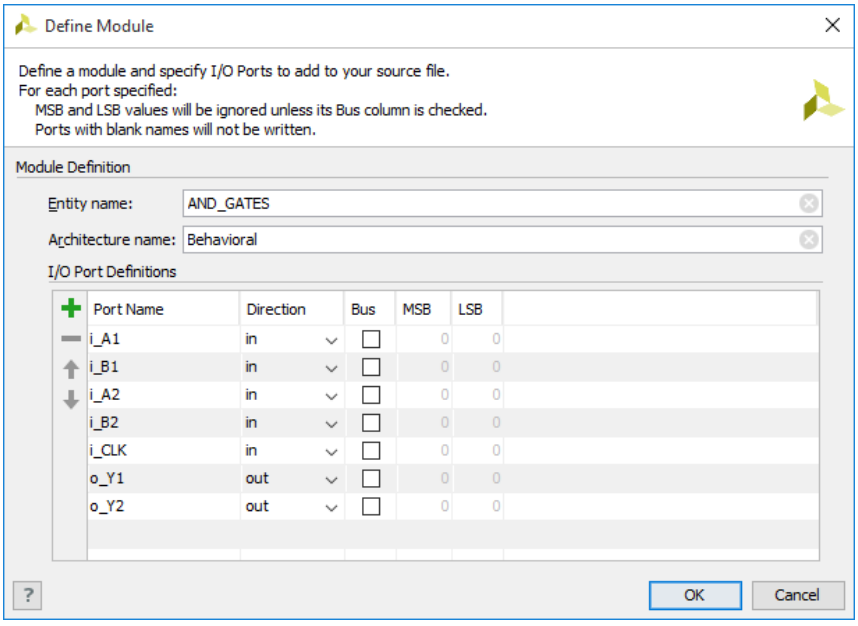

Abbildung 8: Schnittstellen des Moduls `AND_GATES` definieren

Das neu erstellte File befindet sich nun unter den **Design Sources** und kann durch einen Doppelklick darauf bearbeitet werden.

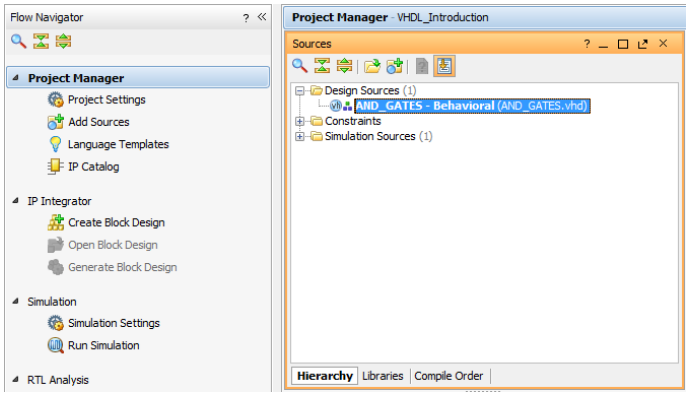

Abbildung 9: Neu erstelltes File zum Bearbeiten öffnen

Man erkennt, dass in der Datei `AND_GATES.vhd` bereits einiges an Programm-Code automatisch generiert wurde. Zu Beginn wird definiert, welche Bibliotheken (und welche Inhalte daraus) verwendet werden sollen. Die eingebunden Bibliotheken können Datentypendefinitionen, Funktionen usw. beinhalten und sind vergleichbar mit einer `#include` Anweisung in C.

```
22 library IEEE;
23 use IEEE.STD_LOGIC_1164.ALL;
```

Weiters wurde bereits eine *Entity* für das Modul erstellt. Die Entity beschreibt die Schnittstellen eines Moduls, also welche Ein- und Ausgänge (`in` oder `out`) sie besitzt.[3] Als Datentyp für die jeweiligen Symbole ist `STD_LOGIC` ausgewählt. Hierbei handelt es sich um eine 9-wertige Logik, die außer den binären Werten '0' und '1' noch weitere Zustände wie zum Beispiel 'Z' (hochohmig), '-' (don't care), 'U' (uninitialisiert) und noch ein paar weitere definiert. Während für die Simulation alle Zustände interessant sein können, sind für die Implementierung in Hardware hauptsächlich die Zustände '0' und '1' und ggf. 'Z' (für Tri-State Buffer) von Bedeutung.

```
34 entity AND_GATES is
35     Port ( i_A1 : in STD_LOGIC;
36            i_B1 : in STD_LOGIC;
37            i_A2 : in STD_LOGIC;
38            i_B2 : in STD_LOGIC;
39            i_CLK : in STD_LOGIC;
40            o_Y1 : out STD_LOGIC;
41            o_Y2 : out STD_LOGIC);
42 end AND_GATES;
```

Den Abschluss bildet die *architecture*. Diese wird nun verwendet um das interne Verhalten des Moduls zu beschreiben.

```
44 architecture Behavioral of AND_GATES is
45
46 begin
47
48
49 end Behavioral;
```

[3] Es gibt auch noch bidirektionale Schnittstellen, die dann mit `inout` bezeichnet werden.

Vor der `begin`-Anweisung der Architektur können noch interne Signale deklariert werden, die innerhalb des Moduls verwendet werden, aber keine Schnittstelle zur Außenwelt des Moduls darstellen. Unsere weitere Aufgabe ist es nun, nach der `begin`-Anweisung das Verhalten unseres Moduls `AND_GATES` korrekt zu modellieren.

VHDL Modellierung des ersten Moduls

Im folgenden Listing ist eine komplette mögliche Architekturbeschreibung des Moduls `AND_GATES` enthalten:

```
44 architecture Behavioral of AND_GATES is
45   signal r_FLIPFLOP : STD_LOGIC := '0';
46 begin
47
48   process (i_CLK)
49   begin
50     if(rising_edge(i_CLK)) then
51       r_FLIPFLOP <= i_A1 AND i_B1;
52     end if;
53   end process;
54
55   o_Y1 <= r_FLIPFLOP;
56   o_Y2 <= i_A2 AND i_B2;
57
58 end Behavioral;
```

Als erstes wurde ein internes Signal `r_FLIPFLOP` erstellt, welches immer den aktuellen Wert unseres D-Flip-Flops darstellen soll. Bei einer Signaldefinition kann über den Operator `:=` ein Initialisierungswert für ein Signal festgelegt werden.

In der eigentlichen Verhaltensbeschreibung nach der `begin`-Anweisung fällt gleich das `process`-Konstrukt auf. Danach befindet sich in Klammern die sogenannte *Sensitivitätsliste*, die in diesem Fall nur ein Signal (und zwar `i_CLK`) enthält. Immer wenn sich der Wert eines Signals ändert, welches sich in der Sensitivitätsliste eines Prozesses befindet, wird der zugehörige Prozess einmal ausgeführt. Danach wird er wieder deaktiviert, bis sich erneut ein Signal in der Sensitivitätsliste ändert. Jeder Prozess muss eine Sensitivitätsliste oder zumindest eine `wait`-Anweisung (wird später bei den Test Benches vorkommen) enthalten.

In unserem Fall wird der Prozess immer geweckt, wenn sich das Clock-Signal ändert (was Sinn ergibt, da sich bei einem Flip-Flop nur bei der Änderung des Clock-Signals der gespeicherte Wert ändern kann). Es ist wichtig zu verstehen, dass alles was sich innerhalb eines Prozesses befindet sequentiell (ähnlich einem normalen C-Programm) abgearbeitet wird. Ein Unterschied zu einem C-Programm hingegen ist, dass sich Werte von Signalen erst ändern, nachdem der zugehörige Prozess deaktiviert wird. Alle Zuweisungen an Signale finden also erst statt, nachdem ein Prozess mit einer Sensitivitätsliste vollständig abgearbeitet wurde (oder wenn ein Prozess eine `wait`-Anweisung erreicht). Was dies genau bedeutet wird in Kapitel 4 durch ein kleines Beispiel etwas detaillierter behandelt.

Innerhalb des Prozesses befindet sich eine `if`-Abfrage, die nur bei einer steigenden Flanke des Signals `i_CLK` ausgeführt wird, wozu die Funktion `rising_edge()` verwendet wird.[4] In dieser `if`-Bedingung findet nun die eigentliche Signalzuweisung statt (dem Signal `r_FLIPFLOP` wird die UND-Verknüpfung der beiden Signale `i_A1` und `i_B` 1 zugewiesen. Eine Signalzuweisung geschieht mittels des Operators `<=`.

Unter dem Prozess befinden sich noch zwei weitere Signalzuweisungen. In der einen Signalzuweisung wird dem Ausgang `o_Y1` der Wert von `r_FLIPFLOP` zugewiesen und in der anderen dem Ausgang `o_Y2` der Wert der UND-Verknüpfung von `i_A2` und `i_B2`. Hierbei handelt es sich um sogenannte implizite Prozesse, die immer dann aufgerufen werden, wenn sich eines der Signale auf der rechten Seite des Zuweisungsoperators ändert. Es wäre auch möglich aus diesen impliziten Prozessen explizite Prozesse zu machen, wie im folgenden Code gezeigt:

```
44   process (r_FLIPFLOP)
45   begin
46     o_Y1 <= r_FLIPFLOP;
47   end process;
48
49   process (i_A2, i_B2)
50   begin
51     o_Y2 <= i_A2 AND i_B2;
52   end process;
```

Es wurde bereits erwähnt, dass der Code innerhalb eines Prozesses sequentiell abgearbeitet wird. Jedoch laufen alle Prozesse eines Moduls parallel zueinander ab (die Hardware, die man beschreiben möchte arbeitet schließlich parallel), es spielt somit keine Rolle in welcher Reihenfolge einzelne Prozesse beschrieben werden. Dieses Konzept zu verstehen ist für das Designen mit VHDL sehr wichtig.

Anmerkung: VHDL ist case insensitive, es wird also keine Groß- oder Kleinschreibung berücksichtigt, das Signal `i_A1` könnte im selben Modul auch über `i_a1` angesprochen werden. Es kann für unterschiedliche Signale im selben Modul somit auch nicht derselbe Name (mit unterschiedlicher Klein- und Großschreibung) verwendet werden.

Design Betrachten (Elaborated Design)

Durch das Betrachten des *Elaborated Design* lässt sich recht gut inspizieren, ob die VHDL-Beschreibung auch die gewünschte Repräsentation in Hardware hat. Dazu klickt man im Flow Navigator unter **RTL Analysis** auf **Open Elaborated Design**. Danach klickt man der Reihe nach, wie in Abbildung 10 gezeigt, zuerst auf **RTL Netlist**, danach wählt man aus, welchen Teil der Schaltung man betrachten möchte (in unserem Fall wollen wir das gesamte Design auswählen) und durch einen Klick auf das mittlere Symbol der darüberlegenden Leiste wird die **Schematic** des gewählten Moduls angezeigt.

Man erkennt im Schaltplan aus Abbildung 11 sehr schön, dass wir genau das erhalten, was wir mit unserem Beispielentwurf aus Abbildung 3 modellieren wollten.

[4] Der Prozess wird bei jeder Änderung des Signals `i_CLK` aufgerufen, jedoch nur bei jeder steigenden Flanke soll auch eine Übernahme eines neuen Wertes stattfinden können.

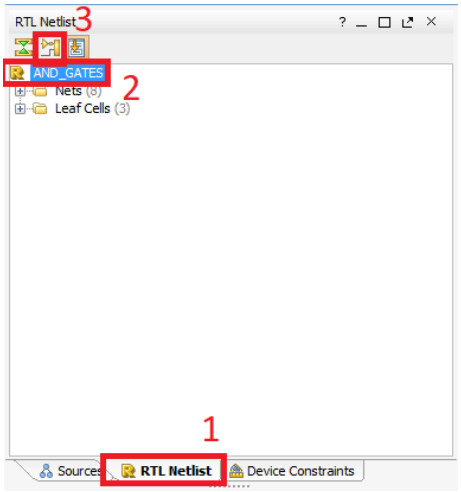

Abbildung 10: Öffnen des Schaltplanes des Elaborated Designs

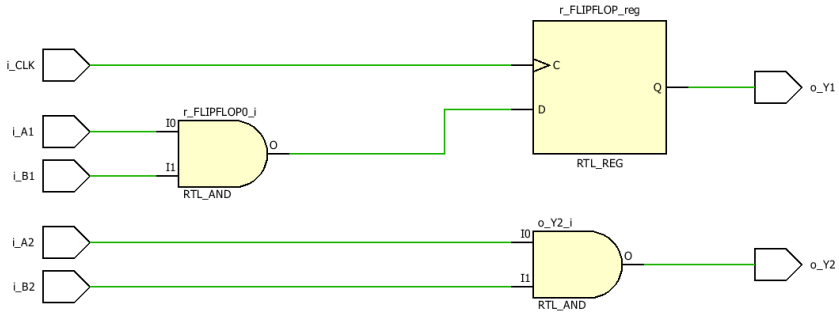

Abbildung 11: Schematic des Moduls `AND_GATES`

Anmerkung: Es kann immer passieren, dass es beim Erstellen des Elaborated Designs oder bei anderen Schritten wie z.B. der Simulation oder Synthese zu Fehlern kommt. Dies kann einerseits an Syntaxfehlern im VHDL-Code liegen. Solche Syntaxfehler werden in den Modulbeschreibungen rot unterstrichen. Jedoch können auch andere Fehler im Design zu Problemen führen, die erst bei der Kompilierung auftreten. Genauere Informationen findet man dann durch einen Klick auf **Messages** in der untersten Leiste der VIVADO-Umgebung. Gegebenenfalls ist auch ein Pfad zu einer Datei angegeben, in welcher genauere Informationen bezüglich der Designfehler zu finden sind. Diese Dateien öffnet man am Besten nicht mit dem standardmäßigen Windows Editor, da dieser leider die Zeilenumbrüche nicht anzeigt. Außerdem ist es wichtig (zum Beispiel damit man nach einer Änderung im Code ein neues Elaborated Design erstellen kann) den geänderten Code auch zu speichern. Dateien, welche noch ungespeicherte Änderungen enthalten, weisen im Reiter über dem Texteditor zusätzlich einen Stern (*) neben dem Dateinamen auf.

VHDL Modellierung der Gesamtschaltung

Wir haben bereits das Modul `AND_GATES` erfolgreich beschrieben. Nun wollen wir jedoch das komplette Design `EXAMPLE_CIRCUIT` erstellen. Dazu wählen wir im Flow Navigator unter **Project Manager** den Punkt **Add Sources** aus. Im sich öffnenden Fenster wählen wir die Option **Add or create design sources**. Danach können wir, wie bereits bei unserem Modul `AND_GATES`, ein neues File erstellen (wir nennen es `EXAMPLE_CIRCUIT`.vhd) und unsere Schnittstellen definieren.

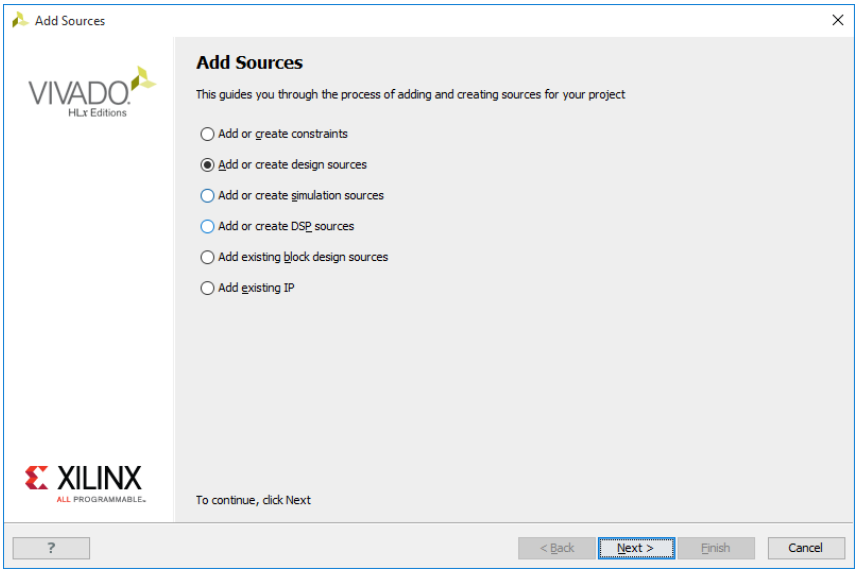

Abbildung 12: Weiteres VHDL-Modul hinzufügen

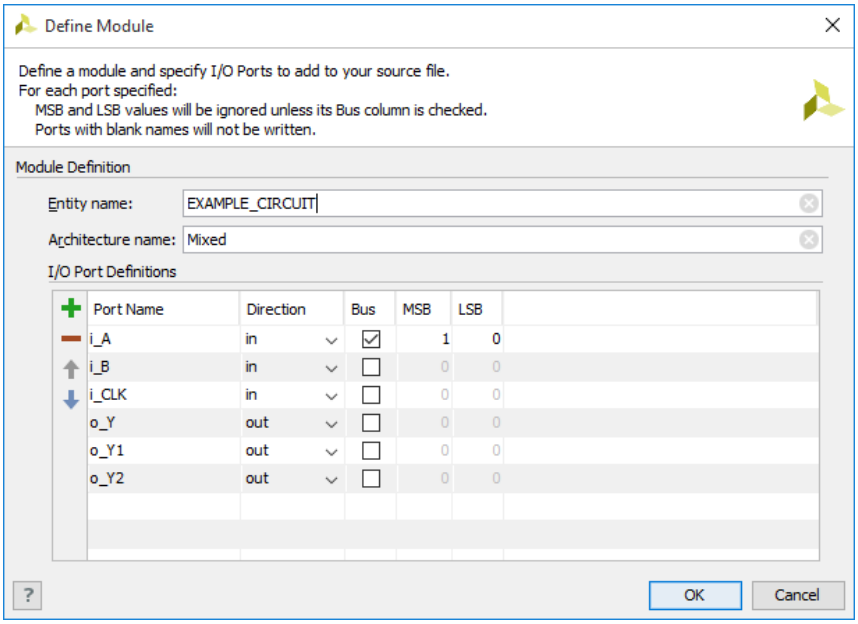

Abbildung 13: Schnittstellen von `EXAMPLE_CIRCUIT` definieren

Wie in Abbildung 13 dargestellt, ist `i_A` nun als Bus ausgeführt. Der Datentyp ist nun `STD_LOGIC_VECTOR`, was einem Array aus `STD_LOGIC` entspricht.

Um das Listing auswählen zu können, kann man entweder im Flow Navigator auf **Project Manager** klicken um das **Sources**-Fenster zu sehen, oder wenn man sich noch im Hauptpunkt **RTL-Analysis** befindet einfach auf den **Sources**-Button (neben **RTL-Netlist**) klicken.

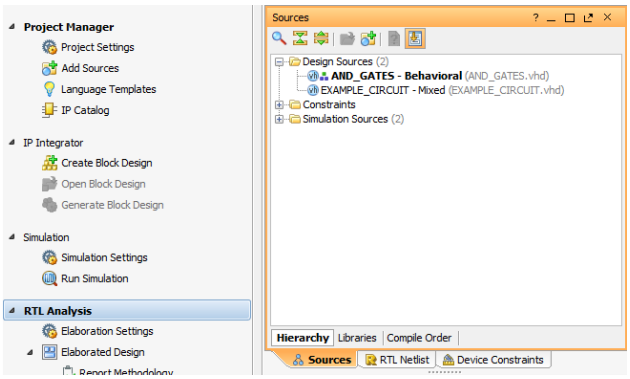

Abbildung 14: `EXAMPLE_CIRCUIT`.vhd öffnen

Hier ist nun der entsprechende Code, der für die Beschreibung der Schaltung verwendet werden kann, ersichtlich (Datei `EXAMPLE_CIRCUIT.vhd`):

```
1  -- Bibliotheken einbinden
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5
6  -- Schnittstellenbeschreibung
7  entity EXAMPLE_CIRCUIT is
8      Port ( i_A : in STD_LOGIC_VECTOR (1 downto 0);
9             i_B : in STD_LOGIC;
10            i_CLK : in STD_LOGIC;
11            o_Y : out STD_LOGIC;
12            o_Y1 : out STD_LOGIC;
13            o_Y2 : out STD_LOGIC);
14 end EXAMPLE_CIRCUIT;
15
16
17 -- Beschreibung der inneren Struktur
18 architecture Mixed of EXAMPLE_CIRCUIT is
19
20   -- intern verwendete Komponente deklarieren
21   component AND_GATES
22     Port ( i_A1 : in STD_LOGIC;
23            i_B1 : in STD_LOGIC;
24            i_A2 : in STD_LOGIC;
25            i_B2 : in STD_LOGIC;
26            i_CLK : in STD_LOGIC;
27            o_Y1 : out STD_LOGIC;
28            o_Y2 : out STD_LOGIC);
29    end component;
30
```

```
31      -- interne Signale der Architektur
32      signal w_Y1 : STD_LOGIC;
33      signal w_Y2 : STD_LOGIC;
34
35  begin
36
37    -- Instanz einer verwendeteten Komponente erstellen und
38    -- Verbindungen zuweisen
39    AND_GATE_inst: AND_GATES
40      Port Map(
41        i_A1 => i_A(1),
42        i_B1 => i_A(0),
43        i_A2 => i_B,
44        i_B2 => i_B,
45        i_CLK => i_CLK,
46        o_Y1 => w_Y1,
47        o_Y2 => w_Y2);
48
49    -- weitere Signalzuweisungen (implizite Prozesse)
50    o_Y <= w_Y1 AND w_Y2;
51    o_Y1 <= w_Y1;
52    o_Y2 <= w_Y2;
53
54  end Mixed;
```

Man erkennt, dass im Deklarationsteil der Architektur nun nicht nur zwei Signale definiert wurden, sondern auch ein `component` deklariert wurde. Hier wird die verwendete Komponente `AND_GATES`, welche im Modul `EXAMPLE_CIRCUIT` verwendet werden soll, mit ihren Schnittstellen definiert (entspricht der Entity-Deklaration von `AND_GATES`). In der eigentlichen Architekturbeschreibung wird nun eine Instanz der Komponente `AND_GATES` erstellt (hier `AND_GATES_inst` genannt) und es wird in der `Port Map` mitgeteilt, wie die Ports des Moduls `AND_GATES` mit den außerhalb verfügbaren Signalen des Moduls `EXAMPLE_CIRCUIT` verbunden werden sollen. Diese Zuweisung der Verbindungen geschieht über den Operator `=>`.

Hier erkennt man auch, wie auf einzelne Elemente eines Vektors zugegriffen werden kann (zum Beispiel wird das Element 1 des Busses `i_A` aus dem Modul `EXAMPLE_CIRCUIT` mit dem Eingang `i_A1` des Moduls `AND_GATES` verbunden). Weiters sieht man, dass es kein Problem ist, wenn Signale in verschiedenen Modulen die gleiche Bezeichnung haben. Jedoch sollte einem bewusst sein, dass es sich bei `i_CLK` auf der linken Seite der Port Map um den Eingangsport des Moduls `AND_GATES` handelt, während es sich bei `i_CLK` auf der rechten Seite um das Signal `i_CLK` aus dem Modul `EXAMPLE_CIRCUIT` handelt.

In unserem Modul `AND_GATES` haben wir nur das Verhalten der Schaltung beschrieben, man spricht hierbei von einer *Verhaltensbeschreibung*. Wenn man in einem Modul nur andere Module einbindet und deren Verbindungen beschreibt (so wie das Modul `AND_GATES` im Modul `EXAMPLE_CIRCUIT` verwendet wird) spricht man von einer *strukturellen Beschreibung*. Häufig kommt es auch zu einer Mischform der beiden Modellierungsarten wie es auch im Modul `EXAMPLE_CIRCUIT` der Fall ist, da einerseits das Modul `AND_GATES` eingebunden wird und andererseits auch noch andere Signalzuweisungen (im obigen Code implizite Prozesse) vorhanden sind.

Da das Modul `EXAMPLE_CIRCUIT` das Modul `AND_GATES` als Komponente enthält, hat VIVADO von selbst das neu erstellte Modul `EXAMPLE_CIRCUIT` als *Top Modul* festgelegt. Dies erkennt man daran, dass im Source-Fenster das Modul `EXAMPLE_CIRCUIT` fett gedruckt ist. Die meisten Operationen, wie das Erstellen des Elaborated Designs, Synthese usw. beziehen sich immer auf das Top Modul. Man kann aber auch manuell ein anderes Modul als Top definieren indem man im Source-Fenster auf das entsprechende File mit der rechten Maustate klickt und im erscheinenden Menü **Set as Top** auswählt.

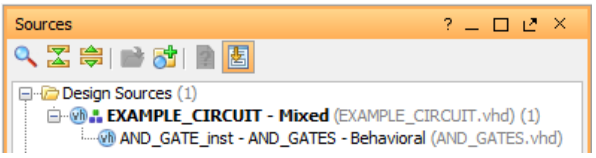

Abbildung 15: Source-Fenster mit `EXAMPLE_CIRCUIT`.vhd als Top

Durch Erstellen des Elaborated Design können wir nochmals überprüfen, ob das erstellte Design unseren Vorstellungen entspricht. Wenn man sich im Unterpunkt RTL-Analysis befindet (und bereits einmal ein Elaborated Design erstellt hat), erscheint im oberen Bereich der VIVADO-Umgebung die Meldung, dass das Elaborated Design nicht mehr aktuell ist. Durch einen Klick auf **Reload** kann es erneut geladen werden.

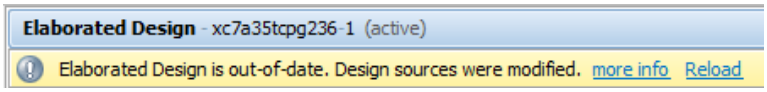

Abbildung 16: Neu Laden des Elaborated Designs

Alle Untermodule werden hier in einem extra Block zusammengefasst, welcher durch einen Klick auf das Plus-Symbol im Detail betrachtet werden kann. Man erkennt in Abbildung 17 sehr gut, dass das Elaborated Design genau dem entspricht, was wir auch modellieren wollten.

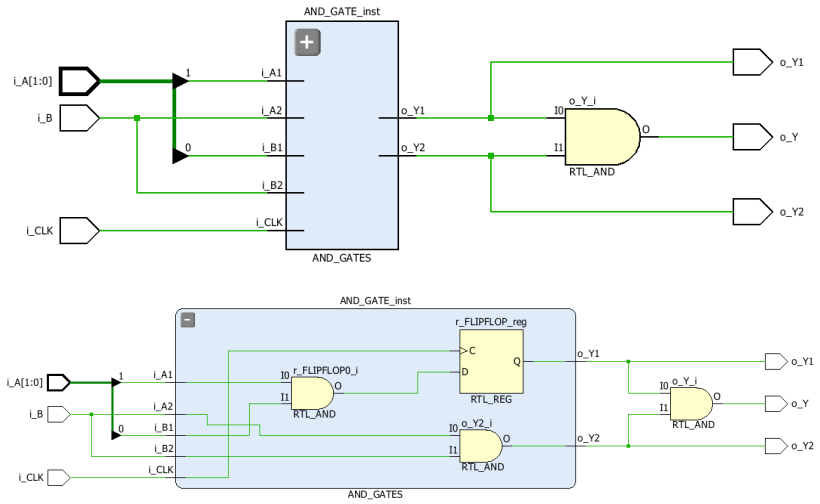

Abbildung 17: Elaborated Design von `EXAMPLE_CIRCUIT`

3 Der restliche Designflow in VIVADO

Simulation

Eine Simulation wird dazu verwendet, um zu überprüfen, ob sich ein Design wie gewünscht verhält. In unserem Beispiel erkennt man anhand des Elaborated Design sehr schnell, dass das beschriebene Design genau der Schaltung entspricht, die wir auch beschreiben wollten. Jedoch gerade bei großen Projekten ist dies nicht mehr so leicht möglich. Zum Testen verwenden wir eine sogenannte *Test Bench*. In Abbildung 18 ist das Prinzip einer Test Bench illustriert. Innerhalb einer Test Bench wir das zu testende Design eingebunden (*unit under test*). Nun werden Stimuli an das Design angelegt, und es wird überprüft, ob die Ausgangssignale des Designs korrekt sind.

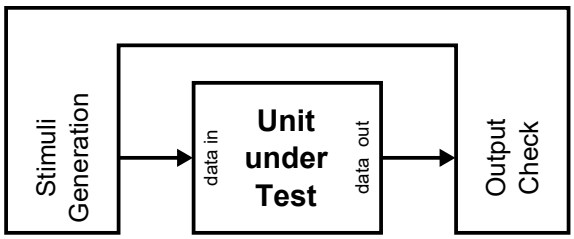

Abbildung 18: Test Bench Konzept

Um die Test Bench zu erstellen, klickt man wieder einmal auf **Add Sources**, wählt nun aber **Add or create simulation sources**. Es wurde der Name `example_tb.vhd` für die Erstellung der Test Bench Datei gewählt.[5] Bei der Erstellung einer Test Bench müssen unter **Define Module** keine Schnittstellen hinzugefügt werden, da die verwendete Test Bench selbst keine Inputs oder Outputs besitzt. Das eben erstellte File findet man bei den **Simulation Sources**

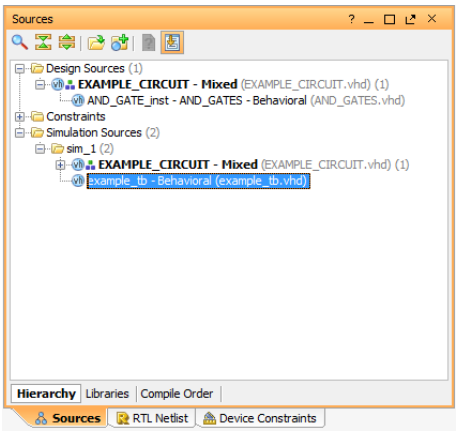

Abbildung 19: Simulation Sources

[5] Die Datei wird automatisch zum bereits vorhandenen *Simulation Set* `sim_1` hinzugefügt. Im Fenster **Add or Create Simulation Sources** kann oben auch ein anderes Simulation Set (unter **Specify Simulation Set**) gewählt bzw. erstellt werden, zu dem die neu erstellten Dateien hinzugefügt werden sollen. So können verschiedene Simulationen für ein Projekt erstellt werden.

Nun muss noch der Code für die Test Bench erstellt werden. Innerhalb der Test Bench wird das Modul `EXAMPLE_CIRCUIT` als `component` (unit under test) eingebunden. Dadurch sollte in den **Simulation Sources** in unserem Simuation Set `sim_1` automatisch das File `example_tb.vhd` als Top Modul festgelegt werden (sollte dann im Sources-Fenster fett gedruckt sein).

Hier der Code der Datei `example_tb.vhd`:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity example_tb is
5
6  end example_tb;
7
8  architecture Behavioral of example_tb is
9
10   component EXAMPLE_CIRCUIT
11     Port ( i_A : in STD_LOGIC_VECTOR (1 downto 0);
12            i_B : in STD_LOGIC;
13            i_CLK : in STD_LOGIC;
14            o_Y : out STD_LOGIC;
15            o_Y1 : out STD_LOGIC;
16            o_Y2 : out STD_LOGIC);
17    end component;
18
19    signal r_A : STD_LOGIC_VECTOR (1 downto 0) := "00";
20    signal r_B : STD_LOGIC := '0';
21    signal r_CLK : STD_LOGIC := '0';
22    signal w_Y : STD_LOGIC;
23    signal w_Y1 : STD_LOGIC;
24    signal w_Y2 : STD_LOGIC;
25
26 begin
27
28   uut: EXAMPLE_CIRCUIT
29     Port Map(
30       i_A => r_A,
31       i_B => r_B,
32       i_CLK => r_CLK,
33       o_Y => w_Y,
34       o_Y1 => w_Y1,
35       o_Y2 => w_Y2);
36
37   PROC_CLK: process
38   begin
39     wait for 5 ns;
40       r_CLK <= '1';
41     wait for 5 ns;
42       r_CLK <= '0';
43   end process;
44
45   PROC_STIM: process
46   begin
47     r_B <= '1';
48     r_A(1) <= '1';
49     wait for 12 ns;
50     r_A(0) <= '1';
```

```
51      wait for 20 ns;
52      r_B <= '0';
53      wait for 10 ns;
54      r_A(1) <= '0';
55      wait for 10 ns;
56      r_A <= "10";
57      wait;
58    end process;
59
60  end Behavioral;
```

Wie bereits erwähnt, wird das Modul `EXAMPLE_CIRCUIT` in unserer Test Bench als Komponente (unit under test) eingebunden. In der Test Bench selbst kann nicht direkt auf die Schnittstellen des eingebundenen Moduls `EXAMPLE_CIRCUIT` zugegriffen werden. Deshalb müssen interne Signale im Deklarationsteil der Architektur der Test Bench erstellt werden, die dann mit den jeweiligen Eingangs- bzw. Ausgangsports von `EXAMPLE_CIRCUIT` in der Port Map verbunden werden.

Weiters sehen wir in unserer Test Bench zwei Prozesse. Den Prozessen wurde noch ein optionales Label gegeben (`PROC_CLK` und `PROC_STIM`). Durch ein Label kann einem Prozess ein Name gegeben werden, um einen besseren Überblick zu bekommen, wozu der Prozess dient.

Wie man erkennen kann, hat keiner der beiden Prozesse eine Sensitivitätsliste. Solch ein Prozess wird, nachdem er das Ende erreicht hat, gleich wieder von vorne begonnen. In Abschnitt 4 wird kurz etwas detaillierter darauf eingegangen, wie die Simulation einer VHDL-Beschreibung funktioniert. Es sei hier momentan nur angemerkt, dass jeglicher Prozess in irgendeiner Weise (zumindest für einen kurzen Zeitraum) deaktivierbar sein muss, da die Simulation bei der Ermittlung der Signalwerte für einen Simulationszeitpunkt sonst in eine Endlosschleife gerät. Dazu muss ein Prozess entweder eine Sensitivitätsliste (der Prozess wir aktiviert wenn sich ein Signal in der Sensitivitätsliste geändert hat und wieder deaktiviert, nachdem er komplett abgearbeitet wurde) oder eine `wait`-Anweisung (der Prozess wird beim Erreichen der wait-Anweisung deaktiviert) enthalten. Es gibt unterschiedliche `wait`-Anweisungen, die den Prozess für eine bestimmte Zeit deaktivieren oder bis eine gewisse Bedingung erfüllt ist.

Der Prozess `PROC_CLK` ist für die Generierung des Clock-Signals zuständig. Es wird dem Clock-Signal abwechselnd eine `'1'` bzw. eine `'0'` zugewiesen und dazwischen wird der Prozess immer für 5 ns deaktiviert. Im Prozess `PROC_STIM` werden den internen Signalen (welche mit den Eingangssignalen von `EXAMPLE_CIRCUIT` verbunden sind) der Reihe nach verschiedene Werte zugewiesen und dazwischen wird der Prozess für eine gewisse Zeit deaktiviert. Man sieht hier auch, dass zur Wertzuweisung an einen Vektor (wenn an mehrere Signale eines Vektors gleichzeitig eine Zuweisung erfolgt) doppelte Anführungszeichen verwendet werden. Die letzte `wait`-Anweisung deaktiviert den Prozess für immer.

Im Flow Navigator kann unter **Run Behavioural Simulation** (eine Auswahlmöglichkeit von **Run Simulation**) die Simulation gestartet werden.

Wenn man die Simulationseinstellungen nicht ändert, werden nach Simulationsstart standardmäßig 1000 ns simuliert und es öffnet sich der *Waveform Viewer* in dem alle Signale des Moduls `EXAMPLE_CIRCUIT` dargestellt werden. Hier kann man nun durch Betrachten der Signalverläufe überprüfen, ob die Schaltung sich korrekt verhält. Für komplexere Designs wäre es sehr aufwendig diese mittels Betrachtung von Signalverläufen zu verifizieren. Dafür gibt es noch weitere Methoden mit denen eine Test Bench von alleine überprüft, ob die Schaltung sich korrekt verhält, darauf wird hier jedoch nicht weiter eingegangen.

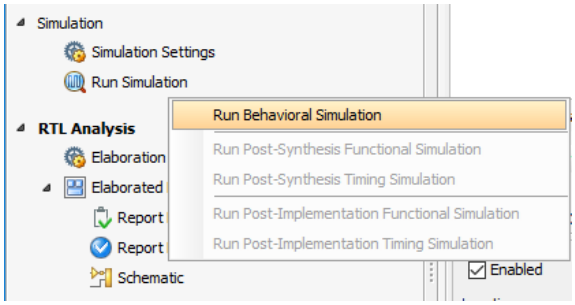

Abbildung 20: Simulation starten

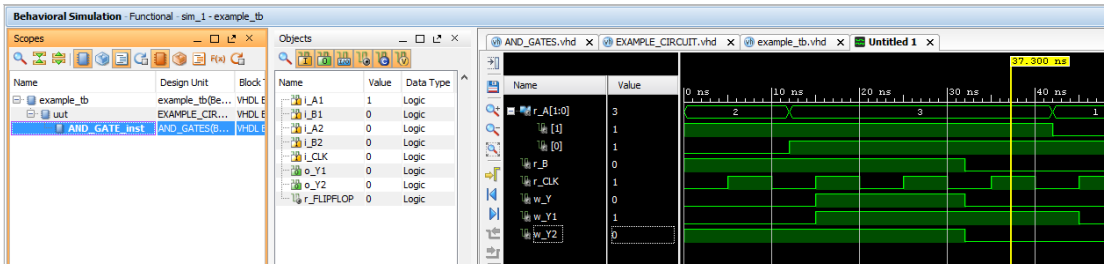

Abbildung 21: Waveform Viewer

Im Bereich **Scopes** der Simulation lassen sich alle Untermodule des getesteten Designs auswählen, und man kann auch interne Signale von Untermodulen dem Waveform Viewer hinzufügen.

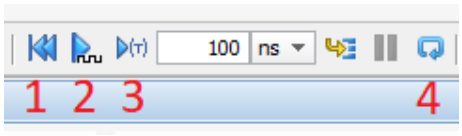

Abbildung 22: Simulationszeit festlegen

In der Leiste über dem Waveform Viewer ist es möglich die Simulationszeit festzulegen (Nummerierung siehe Abbildung 22):

1. Restart: Die Simulation auf den Zeitpunkt 0 zurücksetzten.
2. Run All: Die Simulation so lange durchführen, bis sich kein Signal mehr ändert (bis alle Prozesse dauerhaft deaktiviert sind).
3. Run for specified time: Simulation wird für die daneben angegebene Zeit durchgeführt.
4. Relaunch: Startet die Simulation komplett neu. Dies ist zum Beispiel notwendig, wenn man etwas im Code geändert hat und die Simulation deshalb neu starten muss.

In unserem Beispiel ist es nicht unbedingt sinnvoll den Befehl **Run All** zu verwenden. Die Simulation würde so lange fortgesetzt werden, bis sie manuell gestoppt wird. Dies liegt daran, dass der Prozess `PROC_CLK` niemals vollständig deaktiviert wird, sondern immer nur für 5 ns

und danach wieder eine Signaländerung stattfindet. Nur der Prozess `PROC_STIM` wird bei der `wait`-Anweisung am Ende vollständig deaktiviert.[6]

Synthese und Implementierung

Nachdem wir gezeigt haben, dass die Schaltung das richtige Verhalten zeigt, werden wir mit der Synthese und Implementierung fortfahren. Vorerst können wir schon mal unser Design synthetisieren, indem wir im Flow Navigator **Run Synthesis** wählen. Wenn die Synthese erfolgreich war, ist es momentan noch nicht sinnvoll die Implementierung durchzuführen, da spätestens die Bitstreamgenerierung fehlschlagen wird, da noch keine I/O-Ports des FPGA zu den Input- und Output-Signales zugewiesen wurden. Daher öffnen wir vorerst nur das synthetisierte Design.

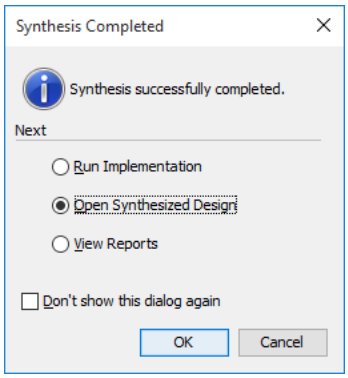

Abbildung 23: Fenster nach erfolgreicher Synthese

Ähnlich wie beim Erstellen des Elaborated Designs kann man vom synthetisierten Design die Schematic betrachten. Man erkennt hier sehr schön die Zusammenstellung des Designs durch Verwendung eines Flip-Flops und zwei LUT2-Bausteinen (sind so konfiguriert, dass sie UND-Verknüpfungen implementieren). Man erkennt außerdem auch, dass das Synthesetool erkannt hat, dass der Ausgang `o_Y2` gleich dem Eingang `i_B` sein muss, und es daher die dazwischen liegende mitmodellierte UND-Verknüpfung entfernt hat.

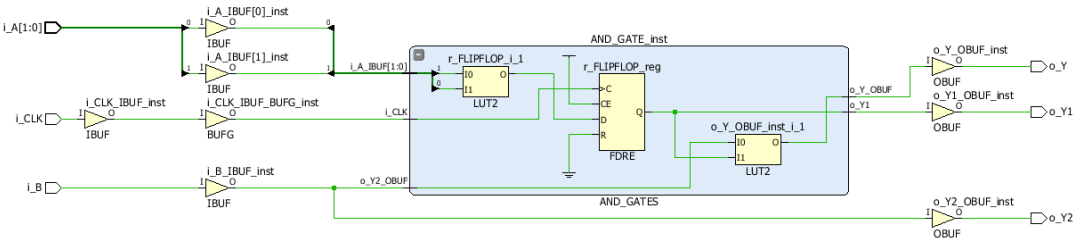

Abbildung 24: Schematic des synthetisierten Designs

[6] Es wäre zum Beispiel mögliche ein boolsches Signal zu verwenden, welches am Ende des Prozesses `PROC_STIM` gesetzt wird, und im Prozess `PROC_CLK` dafür sorgt, dass keine Signaländerungen mehr durchgeführt werden.

Constraints

Constraints werden verwendet, um den Tools zur Synthese und/oder Implementierung mitzuteilen, welche zusätzlichen Bedingungen das Design einhalten muss. Zum Beispiel können *Timing Constraints* verwendet werden um den Tools mitzuteilen, welche Eigenschaften eingehende Clock-Signale haben, und welche Verzögerungszeiten Ein- und Ausgänge des Designs haben. Dadurch können Überprüfungen durchgeführt werden, ob das zeitliche Verhalten der Schaltung eingehalten wird (z.B. Setup- und Hold-Zeiten von Flip-Flops). In unserem Fall werden wir nur *Placement Constraints* festlegen, um zu definieren, mit welchen konkreten Pins des FPGA unsere Ein- und Ausgänge des Moduls `EXAMPLE_CIRCUIT` verbunden werden sollen.

Um dies zu tun, erstellen wir uns zuerst ein Constraints-File, indem wir im Flow Navigator wieder auf den Befehl **Add Sources** klicken und diesmal **Add or create constraints** auswählen Danach erstellen wir ein neue Datei (es wurde der Name `constr.xdc` gewählt).

Im geöffneten synthetisierten Design wählen wir in der oberen Leiste nun **I/O Planning** aus.

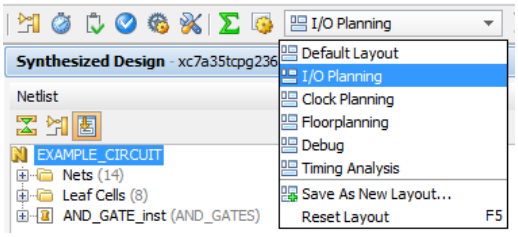

Abbildung 25: I/O Planning öffnen

Nun können wir die I/O-Ports konfigurieren. Unter **I/O Std** muss überall **LVCMOS33** ausgewählt werden (im Datenblattet des Basys 3 Boards findet man die Information, dass die Schnittstellen mit 3.3 V arbeiten). Unter **Package Pin** kann genau ausgewählt werden, mit welchem FPGA-Pin welches Signal verbunden werden soll. Die genauen Bezeichnungen findet man im Datenblatt des Boards bzw. sind sie teilweise auf dem Board neben die jeweiligen Elemente gedruckt. Es werden die drei links außen liegenden Schalter des Board für die Eingänge `i_A` und `i_B` verwendet, als Ausgänge dienen die drei darüberliegenden LEDs, und für das Clock-Signal wird der mittlere der fünf Buttons verwendet. Es wird also kein echtes Clock-Signal verwendet, sondern dieses wird sozusagen durch einen Button-Druck simuliert.

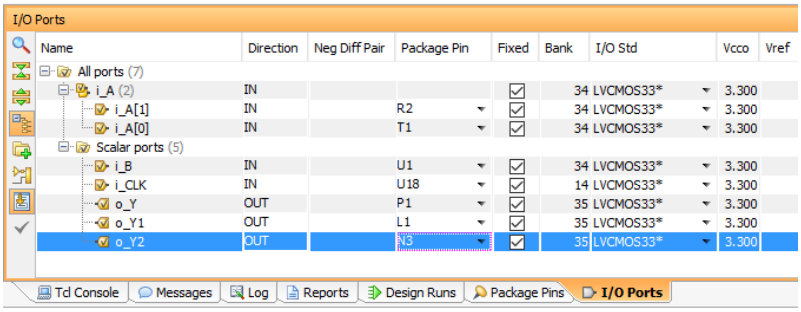

Abbildung 26: I/O-Ports konfigurieren

Nachdem alles konfiguriert wurde, kann man in der obersten Menüleiste unter **File** den Befehl **Save Constraints** wählen, welcher die Constraints für die gerade zugewiesenen I/O-Ports erstellt. Sofern man das zuvor erstellte Constraints-File noch nicht als Ziel (**Target**) für neue Constraints ausgewählt hat, wird man gefragt, ob die Constraints zum File `constr.xdc` hinzugefügt werden sollen.[7] Durch Bestätigung werden die gerade erstellten Definitionen in das Constraints-File geschrieben.

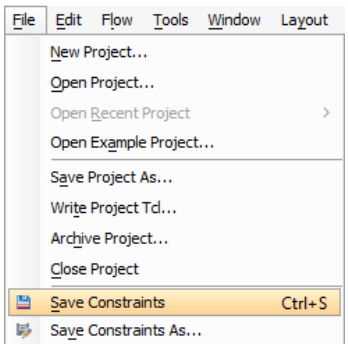

Abbildung 27: Constraints speichern

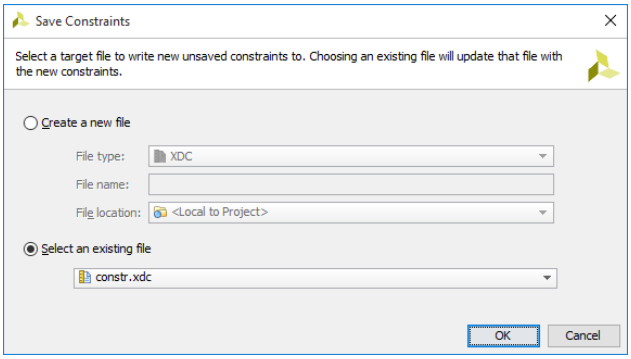

Abbildung 28: Zielfile festlegen

Wir können nun das Constraints-File betrachten um zu sehen, welcher Code erstellt wurde. Bei den Sources erkennt man auch, dass das File nun als Target für neue Constraints deklariert wurde.

Man sieht im Constraints-File, dass den einzelnen Signalen der `IOSTANDARD` und der jeweilige Pin des FPGA zugeordnet werden. Theoretisch könnte man die benötigten Constraints auch manuell ins Constraints-File schreiben, durch die automatische Constraints-Erstellung mittels des IO-Plannings wird das Vorgehen jedoch um einiges erleichtert.

[7] Es können in einem Projekt auch mehrere Constraints-Files verwendet werden. Die Constraints werden sequentiell (von oben nach unten) und ein File nach dem anderen eingelesen. Für manche Constraints ist die Reihenfolge, in der sie definiert werden, wichtig.

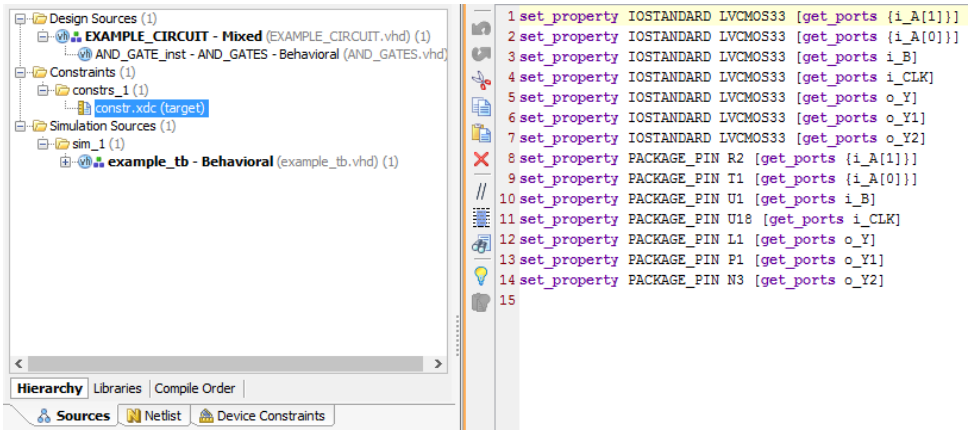

Abbildung 29: Inhalt con `constr.xdc`

Jetzt werden wir jedoch manuell noch einen Constraint hinzufügen, da die Implementierung sonst mit einer Fehlermeldung abschließen würde (siehe Abbildung 30). Normalerweise werden für ein Clock-Signal, welches im FPGA zum Takten von Flip-Flops verwendet wird, spezielle Eingänge verwendet. Nun simulieren wir unser Clock-Signal jedoch mit einem Button, dessen Eingang nicht für ein Clock-Signal geeignet ist, was bei der Implementierung zu einem Fehler führt. Wir fügen einen Constraint hinzu (siehe Abbildung 31), wie in der Fehlermeldung bei der Implementierung beschrieben, um den Tools mitzuteilen, dass dieser Fehler ignoriert werden soll (er wird nur in eine Warnung umgewandelt). Die Bezeichnung `i_CLK_IBUF` kann abweichen und hängt vom gewählten Namen für das Clock-Signal ab. Dieser Constraint wird somit nur benötigt, wenn wir ein Clock-Signal auf einem nicht dafür geeigneten Pin einspeisen, dies aber dennoch zulassen möchten.

Abbildung 30: Fehler, der bei Implementierung des Designs auftreten würde

```
14 set_property PACKAGE_PIN N3 [get_ports o_Y2]
15 set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets i_CLK_IBUF]
16
```

Abbildung 31: Constraint, der den Fehler in eine Warnung umwandelt

Nach dem Speichern des geänderten Constraints-Files können wir nun endgültig auf **Run Implementation** klicken. Es wird zuerst ggf. nochmals eine Synthese durchgeführt, falls diese nicht mehr aktuell ist.[8]

Design auf die Hardware bringen

Bevor man ein Design auf einen FPGA lädt, gibt es unter **Run Simulation** die Möglichkeit Simulationen nach Synthese bzw. Implementierung durchzuführen. Es gibt einerseits funktionale Simulationen, mit denen überprüft werden kann, ob sich das Design auch nach Synthese bzw. Implementierung richtig verhält (es könnte passieren, dass im Zuge der Erstellung der Netzliste bei der Synthese die VHDL-Beschreibung falsch übersetzt wird und sich nicht mehr richtig verhält). Weiters können Simulationen durchgeführt werden, die auch das zeitliche Verhalten der Schaltung (u.a. durch Timing-Constraints festgelegt) berücksichtigen. In der von uns durchgeführten Simulation wird von idealen Zeitbedingungen ausgegangen. Daher kann eine rein funktionale Simulation zwar funktionieren, aber in Hardware dennoch zu Fehlern führen, da zeitliche Bedingungen nicht korrekt eingehalten werden. Diese speziellen Simulationen werden wir dennoch nicht weiter behandeln. Deshalb klicken wir nach geglückter Implementierung im sich öffnenden Fenster gleich auf **Generate Bitstream**. Optional kann dies auch über den Flow Navigator durchgeführt werden. Dadurch wird nun das File erstellt, welches schließlich auf den FPGA geladen wird, um diesen korrekt zu konfigurieren.

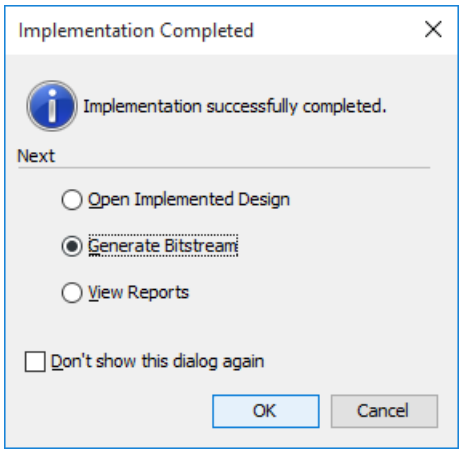

Abbildung 32: Bitstream generieren

Nachdem dies erfolgreich beendet wurde, kann im Flow Navigator, oder im erscheinenden Fenster, **Open Hardware Manager** gewählt werden.

Über die Auswahl von **Open target** und **Auto Connect** wird eine Verbindung zum FPGA hergestellt. Über **Program device** und Wahl unseres FPGAs (`xc7a35t_0`) kann dieser programmiert werden.

[8] Synthese, Implementierung und Bitstreamgenerierung bauen immer aufeinander auf. Wenn man also z.B. einen Bitstream erstellen will und Synthese und Implementierung nicht mehr aktuell sind, werden diese zuvor ausgeführt.)

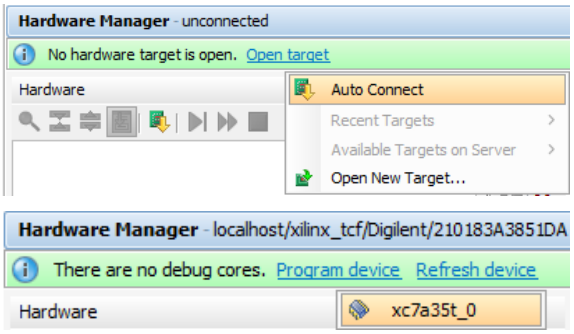

Abbildung 33: FPGA verbinden und programmieren

Im folgenden Fenster muss nur mehr der Bitstream gewählt werden, welcher auf den FPGA zu laden ist. Es sollte bereits der richtige Bitstream ausgewählt sein, weshalb nur mehr auf **Program** gedrückt werden muss. Prinzipiell lässt sich hier aber jedes gespeicherte Bitstream-File auswählen (muss nicht mit dem derzeitigen Projekt zusammenhängen).

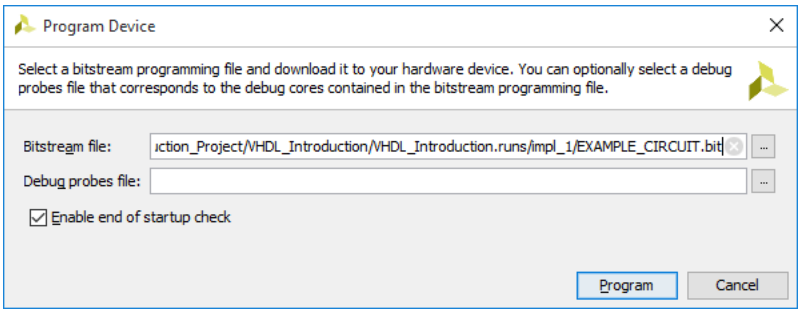

Abbildung 34: Zu programmierenden Bitstream auswählen

Wir haben es nun geschafft unser erstes FPGA-Design erfolgreich zu beschreiben und auf den FPGA zu laden.[9] Nun muss nur noch getestet werden, ob das Design auch in Hardware korrekt funktioniert.

[9] Mittels der derzeitigen Konfigurierung wird das Design über den PC auf den FPFA geladen und geht nach Trennen der Stromversorgung wieder verloren und muss erneut auf den FPGA geladen werden. Es ist jedoch auch möglich eine Konfigurierung auf den nicht flüchtigen Flash-Speicher auf dem Basys 3 Board zu speichern, von welchem der Bitstream nach dem Einschalten des FPGAs automatisch geladen wird.

4 Zusätzliche VHDL Informationen

Simulation

Im Folgenden wird kurz darauf eingegangen, wie die Simulation von VHDL-Modulen funktioniert. Es wurde bereits beschrieben, dass alle Prozesse parallel ablaufen, während die Abarbeitung der Befehle innerhalb der Prozesse sequentiell erfolgt. Da der Computer, der die Simulation durchführt, aber nur sequentiell arbeiten kann, gibt es ein spezielles System um die Parallelität sequentiell zu simulieren.

Für jeden Zeitpunkt, an dem es zu Signaländerungen kommt, wird eine Simulation durchgeführt, um festzustellen, welche Signaländerungen stattfinden. Wenn sich ein Signal ändert, werden alle Prozesse geweckt, bei denen dieses Signal in der Sensitivitätsliste steht. Alle Signaländerungen, die in den abgearbeiteten Prozessen auftreten, werden aufgezeichnet. Erst wenn alle gerade gestarteten Prozesse wieder inaktiv sind, werden die aufgezeichneten Signaländerungen auch zugewiesen. Dadurch werden Signaländerungen erst wirksam, nachdem alle derzeit aktiven Prozesse wieder inaktiv werden. Wenn die nun durchgeführten Signaländerungen wieder andere Prozesse wecken, werden diese auch ausgeführt und die entsprechenden Signaländerungen ermittelt. Erst wenn alle Prozesse inaktiv sind, und durch eine zuvor durchgeführte Signalzuweisung keine neuen Prozesse geweckt werden, ist die Simulation für diesen Zeitpunkt beendet und es kann der nächste Zeitpunkt simuliert werden, an dem sich wieder ein Signal ändert (z.B. durch eine abgelaufene `wait`-Anweisung). Durch diese Simulationsweise können die eigentlich parallel ablaufenden Prozesse sequentiell simuliert werden, und die Reihenfolge, in der die geweckten Prozesse simuliert werden, spielt keine Rolle.

Es kann somit zur Bestimmung der neuen Signalwerte für einen Simulationszeitpunkt zu mehreren Simulationsdurchgängen kommen, in denen erst neue Signalwerte zugewiesen werden und danach ermittelt wird, ob dies weitere Signaländerungen zur Folge hat. Bei mehreren Simulationszyklen für einen bestimmten Zeitpunkt spricht man auch von *Delta Delays* zwischen den Simulationszyklen, da zwischen den Simulationszyklen in der simulierten (idealen) Hardware keine Zeit vergeht.

Signalzuweisung in Prozessen bei der Simulation und Variablen

Hier wird nochmals kurz auf die Sensitivitätsliste und ihren Bezug zur Simulation mit einem kleinen Beispiel näher eingegangen.

```
1 process (A, B, C)
2 begin
3   C <= A AND B;
4   D <= C;
5 end process;
```

Es wird davon ausgegangen, dass die vier Signale `A`, `B`, `C` und `D` bereits definiert sind und momentan alle den Wert `'1'` haben. In der Sensitivitätsliste befinden sich die Signale `A`, `B` und `C`. Das bedeutet, wenn sich eines von diesen Signalen ändert, wird der Prozess aufgerufen. Angenommen Signal `A` ändert seinen Wert auf `'0'`. Dadurch wird der Prozess aktiviert und es wird ermittelt, dass sich `C` auf `'0'` ändern soll (Zeile 3). Jedoch hat `C` in Zeile 4 noch seinen

alten Wert, welcher erneut an `D` zugewiesen wird (`D` ist weiterhin `'1'`). Erst nachdem der Prozess beendet ist, findet die eigentliche Zuweisung von `'0'` an `C` statt. Durch diese Änderung wird der Prozess mit einem Delta Delay Verzögerung erneut aufgerufen. In diesem Durchlauf wird `D` nun der Wert `'0'` zugewiesen. Trotz der sequentiellen Abarbeitung innerhalb eines Prozesses würde es bei dieser rein kombinatorischen Beschreibung keine Rolle spielen ob Zeile 3 und 4 vertauscht werden würden. Jedoch ist es wichtig, dass sich `A`, `B` und `C` in der Sensitivitätsliste befinden. Würde zum Beispiel `C` in der Liste fehlen, würde der zweite Simulationszyklus (in dem `D` vorhin der Wert `'0'` zugewiesen wurde) gar nicht stattfinden.

Wenn das Signal `C` außerhalb des Prozesses gar nicht benötigt wird, wäre es möglich `C` nicht vor dem Prozess als Signal zu deklarieren, sondern (wie im folgenden Code gezeigt) innerhalb des Prozesses als *Variable* zu verwenden.

```
1 process (A, B)
2   variable C : STD_LOGIC := '0';
3 begin
4   C := A AND B;
5   D <= C;
6 end process;
```

Variablen entsprechen im Gegensatz zu Signalen (in der Regel) keinen physikalischen Leitungen. Sie dienen in Prozessen zur Zwischenspeicherung von Werten und können nicht in der Sensitivitätsliste vorkommen. Bei einer Variable findet die Wertzuweisung wirklich sofort statt. Das bedeutet, wenn der Prozess durch eine Signaländerung von `A` oder `B` aufgerufen wird, hat in Zeile 5 die Variable `C` bereits den Wert, der ihr in Zeile 4 zugewiesen wurde. Variablenzuweisungen benötigen einen anderen Zuweisungsoperator als Signale (`:=` anstatt `<=`).

Sensitivitätsliste und Synthese

```
1  PROC1: process (A)
2  begin
3    Y1 <= A AND B;
4  end process;
5
6  PROC2: process (A, B)
7  begin
8    Y2 <= A AND B;
9  end process;
10
11 PROC3: process (C)
12 begin
13   Y3 <= A AND B;
14 end process;
```

Im obigen Codebeispiel sieht man drei verschiedene Prozesse. Alle führen eine Signalzuweisung aus der UND-Verknüpfung der Signale `A` und `B` durch. Jedoch sind die Inhalte der Sensitivitätsliste unterschiedlich. `PROC2` beschreibt eine normale geläufige UND-Verknüpfung, eine Änderung von `Y2` ist nur möglich, wenn sich Signal `A` oder `B` ändert. In `PROC1` soll die Zuweisung nur durchgeführt werden, wenn sich das Signal `A` ändert und `PROC3` wird nur ausgeführt, wenn sich das im Prozess selbst gar nicht vorkommende Signal `C` ändert. Die Simulation führt dies wie beschrieben aus, jedoch berücksichtigt nicht jedes Synthesetool die Sensitivitätslisten bzw. berücksichtigt sie nicht

jedes Tool in gleicher Weise. Deshalb sollte man sich im klaren sein, welche Beschreibungen das Synthesetool auch wie gewünscht umsetzt, sonst kann unter anderem ein Grund dafür sein, dass die synthetisierte Schaltung sich nicht gleich verhält wie die Simulation.

Um sicherzustellen, dass sich die simulierte und die synthetisierte Schaltung gleich verhalten, sollten sich in der Sensitivitätsliste von kombinatorischen Prozessen alle Signale befinden, die auf der rechten Seite von Zuweisungen stehen. In getakteten Prozessen sollte in der Sensitivitätsliste nur das Clock-Signal (und falls verwendet, Signale für einen asynchronen Reset) stehen.[10]

Die `case`-Anweisung

Innerhalb eines Prozesses ist es möglich Fallunterscheidungen über `if-else`-Anweisungen, wie im folgenden Codebeispiel gezeigt, zu beschreiben.

```
1  process (A)
2  begin
3    if A = "00" then
4      Y <= "10";
5    elsif A = "01" then
6      Y <= "11";
7    else
8      Y <= "00";
9    end if;
10 end process;
```

Im Prinzip wird hier das Verhalten eines Multiplexers beschrieben, der abhängig vom 2-bit breiten Signal `A` auswählt, welcher Wert dem ebenfalls 2-bit breiten Signal `Y` zugewiesen werden soll. Es sollten hier immer alle möglichen Fälle (alle Möglichkeiten für Signal `A`) in der Beschreibung enthalten sein bzw. sollte am Ende eine `else`-Anweisung die restlichen Fälle behandeln. Ansonsten werden in der synthetisierten Schaltung unerwünschte Latches eingebaut, da bei fehlenden Zuweisungen der Signalwert von `Y` auf dem alten Wert gehalten werden muss.

Anstellen des `if-else`-Konstruktes ist es auch möglich eine `case`-Anweisung zu verwenden. So können Fallunterscheidungen einfach beschrieben werden. Im unteren Code würde das vorige Beispiel umgeschrieben so aussehen:

```
1  process (A)
2  begin
3    case A is
4      when "00" =>
5        Y <= "10";
6      when "01" =>
7        Y <= "11";
8      when others =>
9        Y <= "00";
10   end case;
11 end process;
```

[10] Bei einem getakteten Prozess sollte es keine Auswirkung auf die synthetisierte Schaltung haben, wenn zusätzliche Signale in der Sensitivitätsliste stehen, da die `if`-Bedingung des Clock-Signales sowieso nur ausgeführt wird, wenn sich das entsprechende Clock-Signal ändert. Jedoch wird unnötig Simulationszeit beansprucht, wenn der getaktete Prozess auch bei anderen Signaländerungen aufgerufen wird, obwohl es sowieso zu keiner Änderung bei einem Signal kommen kann.

Es ist hier ebenso wichtig immer alle möglichen Fälle anzugeben. Es sollte daher immer (wenn nicht alle Fälle abgedeckt sind) ein `when others` Konstrukt vorhanden sein, welches dem obigen `else` entspricht. Zu beachten ist hier, dass `STD_LOGIC` eine 9-wertige Logik besitzt. Wenn es sich bei Signal `A` um einen `STD_LOGIC_VECTOR` handelt, wären selbst durch Angabe aller möglichen Bitkombinationen (aus `'0'` und `'1'`) nicht alle Fälle abgedeckt. Für die Synthese wäre dies zwar ausreichend, jedoch nicht für die Simulation.

Typendefinition

Es ist auch möglich eigene benutzerdefinierte Datentypen zu erstellen. Dies kann zum Beispiel für die Beschreibung von Zustandsautomaten praktisch sein.

```
1 type ZUSTAND_TYPE is (Z0, Z1, Z2);
2 signal zustand : ZUSTAND_TYPE := Z2;
```

Im obigen Code wird der Datentyp `ZUSTAND_TYPE` definiert, welcher die drei möglichen Zustände `Z0`, `Z1` und `Z2` besitzt. Diese Definition kann zum Beispiel im Deklarationsteil einer `architecture` durchgeführt werden. Anschließend wird gleich ein Signal `zustand` des Typs `ZUSTAND_TYPE` erstellt und mit `Z2` initialisiert.

Nun kann das Signal `zustand` zum Beispiel mit einer `case`-Anweisung verwendet werden, um zu beschreiben, was für jeden der 3 möglichen Signalwerte von `zustand` (`Z0`, `Z1` und `Z2`) passieren soll. Dies ist ideal zur Modellierung von Zustandsübergängen und Ausgangs-Signalen bei Zustandsautomaten.

Referenzen

- Molitor P. und Ritter J. *VHDL. Eine Einführung*. Pearson Studium, 2004.
- Gehrke W. u. a. *Digitaltechnik. Grundlagen, VHDL, FPGAs, Mikrocontroller*. 7. Aufl. Springer, 2016.
- Kafig W. *VHDL 101. Everything you need to know to get started*. Elsevier Inc., 2011.
- Wilson P. *Design Recipes for FPGAs. Using Verilog and VHDL*. 2. Aufl. Elsevier Inc., 2016.
- Molenkamp E. und Mekenkamp E. *Process with 'Incomplete' Sensitivity Lists and Their Synthesis Aspects*. Netherlands, 1997. URL: http://ieeexplore.ieee.org/document/623933/ (besucht am 31. 10. 2017).
- *Basic FPGA Tutorial*. SO-LOGIC electronic consulting. 2017. URL: https://www.so-logic.net/documents/upload/Basic_FPGA_Tutorial_Vivado_VHDL.pdf (besucht am 16. 09. 2017).
- *Basys 3™ FPGA Board Reference Manual*. Digilent Inc. 23. März 2017. URL: https://reference.digilentinc.com/_media/reference/programmable-logic/basys-3/basys3_rm.pdf (besucht am 22. 10. 2017).
- VHDL, Verilog and FPGA Tutorial. URL: https://www.nandland.com/ (besucht am 16. 09. 2017).

MICROPROGRAM MEMORY (MPM) V1.0.2

Color-Codes and Definitions

Amux: 0
Is used if the value of the Amux does not matter. This happens for instance if the ALU is not needed for any operation during this cycle or at a carry set/clear-operation where no input is needed.

ALU: 1111
This operation was by default not defined in the architecture. New definition: ALUout <- 0xFF and status bits are not affected (also for 1101 and 1100). This prevents that the status-bits are affected if the ALU is not used. Actually it would also be possible to use an ALU-update operation that does not change status bits, but as convention always 1111 is used. By the way, in VHDL it is necessary to define all possibilities as otherwise latches would be inferred. If it is not important that the status-bits are not changing theoretically every ALU-operation would be possible instead of 1111.

MBR-MAR-RD-WR: **0000**
Is used if no read- or write-operation should be performed. Actually some other combinations also could be possible without any effect, but as convention 0000 is used.
Read operation (2 cycles): 0110 and 1010
Write operation (3 cycles): 0100, 1001, 0001

C/B/A: 11111
Is used if the value of C/B/A does not matter. For C this is the case if ENC is cleared. Writing to 11111 is actually not allowed as this register holds the constant 0, but a probably unintended write is prevented in the VHDL code. For A or B this is the case if they are not needed for ALU-operations or for A, if the MBR output is used for the ALU input A.

Address: 0x000
Used if the value of the address does not matter as no jump is possible.

Bold values: Important that these bits are set correctly.

Rows containing only "X" are replaced by the following bit-code:

0	0	**1**	**1**	1	1	1	1	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

This code performs a jump in the microcode to the last address. Whenever such a line happens to be executed the program gets stuck by continuing executing the last microcode line. This makes it easier to find eventual existing errors in the microcode when simulating the processor.

Instruction / Type / CC changes, No. of instructions to execute	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A					Address										Instruction Mnemonic	(Micro) Instruction Comments
POWER UP	0x000	**0**	0	**0**	**0**	1	1	1	1	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MPC ← MPC+1	nothing happens at the power up cycle, only incrementing the MPC by one
EVENT CHECK	0x001	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	1	1	1	1	1	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	Z ← EVENT; if N=1 jump to 0x004 (FETCH)	save EVENT-register (as it can be changed externally while decoding); if no event detected (defined if N=1) continue with FETCH-routine (fetch next instruction)
RESET CHECK	0x002	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	ALUout ← Z; if Z=1 jump to 0x009 (RESET)	check if the detected event is RESET (defined if Z=1); if so jump to the RESET-routine
INTERR. FLAG CHECK	0x003	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	**0**	**1**	**0**	**0**	**1**	1	1	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	ALUout ← CC AND Interr. (Dis.) Mask; If Z=1 jump to 0x014 (INTERRUPT ROUTINE)	If no RESET detected an interrupt should be performed (jump to INTERRUPT-routine) if Interr. Disable Flag is not set; else continue with FETCH of next instruction
FETCH ROUTINE	0x004	**0**	0	**0**	**0**	1	1	1	1	**0**	1	1	**0**	**0**	1	1	1	1	1	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	fetches the next instruction from the memory, the PC is increased and consequently points to the next OP-code or the first byte of the operand of the instruction
5	0x005	**0**	0	**0**	**0**	0	1	1	0	1	0	1	0	**0**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ←MEM[MAR]; C ← 0	
	0x006	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 1 + C	
	0x007	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	0	0	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
	0x008	**1**	**1**	0	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**1**	**1**	**0**	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	IR ← MBR; MPC ← f(IR)	

Instruction / Type CC changes, No. of instructions to execute	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A					Address												Instruction Mnemonic	(Micro) Instruction Comments
RESET ROUTINE	0x009	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	EVENT ← 0x80 (no event)	reset EVENT register (no interrupt)
11	0x00A	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	SPH ← 0xFF	reset SP (last address of memory)
	0x00B	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	SPL ← 0xFF	
	0x00C	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	PCH ← 0x00	reset PC (point to RESET vector at beginning of RAM: jump instruction with start address of main)
	0x00D	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	PCL ← 0x00	
	0x00E	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	1	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	XRH ← 0xFF	reset address registers X and Y (point to last address of memory)
	0x00F	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	XRL ← 0xFF	
	0x010	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	YRH ← 0xFF	
	0x011	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	YRL ← 0xFF	
	0x012	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	AC ← 0x00	reset AC
	0x013	0	0	1	1	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	1	CC ← 0x10 (Interr. (Dis.) Mask); jump to 0x001	CC: interrupts disabled and other bits cleared
INTERRUPT ROUTINE	0x014	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	EVENT ← 0x80 (no event)	reset EVENT register (no interrupt)
28	0x015	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	C ← 0	load address of interrupt-vector, which is the content of old EVENT (which was previously stored in Z) * 4, as every int.-vector consists of 4 bytes (jump instruction + 2 byte address + unused byte); as long less than 64 int.-vectors are used it would theoretically be possible to leave out the rotations of EARH
	0x016	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	EARH ← 0	
	0x017	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← rol(Z)	
	0x018	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1	1	0	1	0	1	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	EARH ← rol(EARH) (C for sure 0)	
	0x019	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← rol(EARL)	
	0x01A	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	EARH ← rol(EARH) (C for sure 0)	
	0x01B	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	1	1	0	0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0xFF + C	push PC
	0x01C	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0xFF + C	
	0x01D	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
	0x01E	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← PCL	
	0x01F	0	0	0	0	0	1	1	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	M[MAR] ← MBR; C ← 0	
	0x020	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	1	1	0	0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0xFF + C	
	0x021	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	1	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0xFF + C	
	0x022	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
	0x023	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	MBR ← PCH	
	0x024	0	0	0	0	1	1	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	M[MAR] ← MBR; C ← 0	
	0x025	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	1	1	0	0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0xFF + C	push CC with disable interrupt flag cleared (interrupts activated); at RTI no need to clear the flag again, just restore CC and interrupts are enabled again
	0x026	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0xFF + C	
	0x027	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
	0x028	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	MBR ← CC	
	0x029	0	0	0	0	0	1	1	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	M[MAR] ← MBR; C ← 0	
	0x02A	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	CC ← NOT(CC)	disable interrupts (using OR)
	0x02B	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	1	1	0	1	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	OPRL ← NOT(Interr. (Dis.) Mask)	
	0x02C	0	0	0	0	0	0	1	1	0	0	0	0	1	0	1	1	0	0	1	0	0	0	1	0	0	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	CC ← CC AND OPRL	
	0x02D	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	CC ← NOT(CC)	
	0x02E	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	PCH ← EARH	continue at interrupt vector and fetch the jump instruction of the corresponding interrupt vector
	0x02F	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	1	0	PCL ← EARL; jump to 0x004 (FETCH)	
	0x030 - 0x0DF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
JMP @y	0x0E0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
-	0x0E1	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
	0x0E2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← YRL + MBR + C	add operand to YRL
6	0x0E3	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1	0	0	1	0	1				ALUout ← MBR; if N=1 jump to 0x0E5	depending if the operand is positive or negative 0x00 or 0xFF has to be considered as high bye of the operand
	0x0E4	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	PCH ← YRH + 0 + C; jump to 0x001	
	0x0E5	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	PCH ← YRH + 0xFF + C; jump to 0x001	
	0x0E6 - 0x11F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A					Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																									
NOP	Inh.	0x120	**0**	0	**1**	**1**	1	1	1	1	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	jump to 0x001	nothing happens, status-bits unaffected
		0x121 - 0x19F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ORC	Imm.	0x1A0	**0**	0	**0**	**0**	1	1	1	1	0	1	1	0	**0**	1	1	1	1	1	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	using DeMorgan: Y = A OR B = NOT(NOT(A) AND NOT(B))
N, Z, V, C		0x1A1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	1	0	1	0	**1**	**0**	**1**	**0**	**0**	**1**	1	1	1	1	1	**0**	**1**	**0**	**0**	**1**	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; CC ← !CC	
		0x1A2	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRL ← NOT(MBR)	
		0x1A3	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	0	0	0	0	0	0	0	0	0	0	CC ← CC & OPRL	
7		0x1A4	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	1	1	1	1	1	**0**	**1**	**0**	**0**	**1**	0	0	0	0	0	0	0	0	0	0	CC ← NOT(CC) (V ← 0, C ← 1)	Changes of N, Z, V, C depend on the operand
		0x1A5	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x1A6	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	PCH ← PCH + 0 + C; jump to 0x001	
		0x1A7 - 0x1BF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ANDC	Imm.	0x1C0	**0**	0	**0**	**0**	1	1	1	1	0	1	1	0	**0**	1	1	1	1	1	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0x1C1	**0**	0	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0x1C2	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	CC ← MBR & CC (V ← 0, C still 1)	Changes of N, Z, V, C depend on the operand
5		0x1C3	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x1C4	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	PCH ← PCH + 0 + C; jump to 0x001	
		0x1C5 - 0x1FF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
BRA	Rel.	0x200	**0**	0	**0**	**0**	1	1	1	1	0	1	1	0	**0**	1	1	1	1	1	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	adding operand to PC
-		0x201	**0**	0	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0x202	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← MBR + PCL + C	
		0x203	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	ALUout ← MBR; if N=1 jump to 0x205	depending if the operand is positive or negative 0x00 or 0xFF has to be considered as high bye of the operand
5		0x204	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	PCH ← PCH + 0 + C; jump to 0x001	
		0x205	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	PCH ← PCH + 0xFF + C; jump to 0x001	
		0x206 - 0x23F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
BBC	Rel.	0x240	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1	**0**	**0**	**1**	**0**	**0**	**1**	0	0	0	0	0	0	0	0	0	0	ALUout ← CC & Carry Mask	jump if C=0 (PC + operand), otherwise increment PC
-		0x241	**0**	0	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	C ← 1; if Z=1 jump to 0x244	
		0x242	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x243	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	PCH ← PCH + 0 + C; jump to 0x001	
		0x244	**0**	0	**0**	**0**	1	1	1	1	0	1	1	0	**0**	1	1	1	1	1	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0x245	**0**	0	**0**	**0**	1	1	1	1	1	0	1	0	**0**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR] (C still 1)	
4 / 7		0x246	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← MBR + PCL + C	
		0x247	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	ALUout ← MBR; if N=1 jump to 0x249	
		0x248	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	PCH ← PCH + 0 + C; jump to 0x001	
		0x249	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	PCH ← PCH + 0xFF + C; jump to 0x001	
		0x24A - 0x24F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type	MPC-Addr.	OP	Amux	Cond	ALU	MBR	MAR	RD	WR	ENC	C	B	A	Address									Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																									
BCS	Rel.	0x250	0	0	0	0	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	ALUout ← CC & Carry Mask	jump if C=1
-		0x251	0	0	1	0	0	1	1	0	0	0	1	1	1	1	1	0	0	1	0	1	1	0	0	C ← 1; if Z=1 jump to 0x258	
		0x252	0	0	0	0	1	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0x253	0	0	0	0	1	1	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	MBR ← MEM[MAR] (C still 1)	
		0x254	0	1	0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	PCL ← MBR + PCL + C	
		0x255	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	1	1	0	1	1	1	1	ALUout ← MBR; if N=1 jump to 0x257	
		0x256	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C; jump to 0x001	
		0x257	0	0	1	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	0	0	1	PCH ← PCH + 0xFF + C; jump to 0x001	
		0x258	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x259	0	0	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	1	1	0	1	PCH ← PCH + 0 + C; jump to 0x001	
4 / 7																									
		0x25A - 0x25F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
BNE	Rel.	0x260	0	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	1	0	0	0	ALUout ← CC & Zero Mask	jump if Z=0
-		0x261	0	1	0	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0	0	1	C ← 1; if Z=1 jump to 0x264	
		0x262	0	0	0	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x263	0	0	1	1	0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x264	0	0	0	0	1	1	1	0	0	1	1	1	0	0	1	1	1	1	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0x265	0	0	0	1	1	1	1	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	MBR ← MEM[MAR] (C still 1)	
		0x266	0	1	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	PCL ← MBR + PCL + C	
		0x267	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	1	1	0	1	1	1	0	ALUout ← MBR; if N=1 jump to 0x269	
		0x268	0	0	1	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x269	0	0	1	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	0	0	1	PCH ← PCH + 0xFF + C; jump to 0x001	
4 / 7																									
		0x26A - 0x26F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
BEQ	Rel.	0x270	0	0	0	0	0	1	1	0	0	0	1	1	1	1	0	0	0	1	0	0	0	ALUout ← CC & Zero Mask	jump if Z=1
-		0x271	0	1	0	0	1	1	1	0	0	0	1	1	1	1	1	1	0	0	1	1	0	0	1	C ← 1; if Z=1 jump to 0x278	
		0x272	0	0	0	0	1	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0x273	0	0	0	1	1	1	1	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	MBR ← MEM[MAR] (C still 1)	
		0x274	0	1	0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	PCL ← MBR + PCL + C	
		0x275	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	1	1	0	1	1	1	1	ALUout ← MBR; if N=1 jump to 0x277	
		0x276	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x277	0	0	1	1	0	0	1	0	0	0	1	1	0	0	0	0	0	1	1	0	0	0	1	PCH ← PCH + 0xFF + C; jump to 0x001	
		0x278	0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x279	0	1	1	0	0	1	0	0	0	0	1	1	0	0	1	1	1	1	0	0	0	0	0	PCH ← PCH + 0 + C; jump to 0x001	
4 / 7																									
		0x27A - 0x27F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
BVC	Rel.	0x280	0	0	0	0	0	1	1	0	0	0	0	1	1	1	1	0	1	0	0	0	0	ALUout ← CC & Overflow Mask	jump if V=0
-		0x281	0	1	0	0	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	1	0	0	1	C ← 1; if Z=1 jump to 0x284	
		0x282	0	0	0	0	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x283	0	0	1	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x284	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0x285	0	0	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	MBR ← MEM[MAR] (C still 1)	
		0x286	0	1	0	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	PCL ← MBR + PCL + C	
		0x287	0	1	0	1	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	0	ALUout ← MBR; if N=1 jump to 0x289	
		0x288	0	0	1	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x289	0	0	1	1	0	0	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	0	1	PCH ← PCH + 0xFF + C; jump to 0x001	
4 / 7																									

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A					Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																										
		0x28A - 0x28F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
BVS	Rel.	0x290	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	ALUout ← CC & Overflow Mask	jump if V=1
-		0x291	0	0	1	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	0	0	1	1	0	0	0	C ← 1; if Z=1 jump to 0x298	
		0x292	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0x293	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR] (C still 1)	
		0x294	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	PCL ← MBR + PCL + C	
		0x295	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	0	0	1	0	1	1	1	ALUout ← MBR; if N=1 jump to 0x297	
4 / 7		0x296	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x297	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0xFF + C; jump to 0x001	
		0x298	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x299	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x29A - 0x29F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
BPL	Rel.	0x2A0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	1	1	1	0	1	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	ALUout ← CC & Negative Mask	jump if N=0 (positive)
-		0x2A1	0	0	1	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	0	1	0	0	1	0	0	C ← 1; if Z=1 jump to 0x2A4	
		0x2A2	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x2A3	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x2A4	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
4 / 7		0x2A5	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR] (C still 1)	
		0x2A6	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	PCL ← MBR + PCL + C	
		0x2A7	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	0	1	0	1	0	0	1	ALUout ← MBR; if N=1 jump to 0x2A9	
		0x2A8	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x2A9	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0xFF + C; jump to 0x001	
		0x2AA - 0x2AF	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
BMI	Rel.	0x2B0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	1	1	1	0	1	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	ALUout ← CC & Negative Mask	jump if N=1 (negative)
-		0x2B1	0	0	1	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	0	1	1	1	0	0	0	C ← 1; if Z=1 jump to 0x2B8	
		0x2B2	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0x2B3	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR] (C still 1)	
		0x2B4	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	PCL ← MBR + PCL + C	
4 / 7		0x2B5	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	0	1	1	0	1	1	1	ALUout ← MBR; if N=1 jump to 0x2B7	
		0x2B6	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x2B7	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0xFF + C; jump to 0x001	
		0x2B8	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x2B9	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x2BA - 0x36F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		

Instruction / CC changes, No. of instructions to execute	Type	MPC-Addr.	OP	Amux	Cond	ALU	MBR	MAR	RD	WR	ENC	C	B	A	Address	Instruction Mnemonic	(Micro) Instruction Comments
JSR	Abs.	0x370	0	0	0 0	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MARH ← PCH; MARL ← PCL	
-		0x371	0	0	0 0	0 1 1 1	1	0	1	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	load effective address into EA-reg
		0x372	0	1	0 0	0 0 0 0	0	0	0	0	1	0 1 0 1 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	EARH ← MBR (C still 1)	
		0x373	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 1 1	0 0 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	PCL ← PCL + 0 + C	
		0x374	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 1 0	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	PCH ← PCH + 0 + C	
		0x375	0	0	0 0	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MARH ← PCH; MARL ← PCL	
		0x376	0	0	0 0	0 1 1 1	1	0	1	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
		0x377	0	1	0 0	0 0 0 0	0	0	0	0	1	0 1 0 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	EARL ← MBR (C still 1)	
		0x378	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 1 1	0 0 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	PCL ← PCL + 0 + C	increment PC (now points to the OP-code of the next instruction) and push it on the stack
		0x379	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 1 0	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	PCH ← PCH + 0 + C	
		0x37A	0	0	0 0	0 1 1 0	0	0	0	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	C ← 0	
22		0x37B	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 0 1	0 0 1 0 1	1 1 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0	SPL ← SPL + 0xFF + C	
		0x37C	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 0 0	0 0 1 0 0	1 1 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0	SPH ← SPH + 0xFF + C	
		0x37D	0	0	0 0	1 1 1 1	0	1	0	0	0	1 1 1 1 1	0 0 1 0 0	0 0 1 0 1	0 0 0 0 0 0 0 0 0 0 0 0	MARL ← SPL; MARH ← SPH	
		0x37E	0	0	0 0	0 0 0 0	1	0	0	1	0	1 1 1 1 1	1 1 1 1 1	0 0 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MBR ← PCL	
		0x37F	0	0	0 0	0 1 1 0	0	0	0	1	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	M[MAR] ← MBR; C ← 0	
		0x380	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 0 1	0 0 1 0 1	1 1 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0	SPL ← SPL + 0xFF + C	
		0x381	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 0 0	0 0 1 0 0	1 1 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0	SPH ← SPH + 0xFF + C	
		0x382	0	0	0 0	1 1 1 1	0	1	0	0	0	1 1 1 1 1	0 0 1 0 0	0 0 1 0 1	0 0 0 0 0 0 0 0 0 0 0 0	MARL ← SPL; MARH ← SPH	
		0x383	0	0	0 0	0 0 0 0	1	0	0	1	0	1 1 1 1 1	1 1 1 1 1	0 0 1 1 0	0 0 0 0 0 0 0 0 0 0 0 0	MBR ← PCH	
		0x384	0	0	0 0	0 0 0 0	0	0	0	1	1	0 0 1 1 1	1 1 1 1 1	0 1 0 1 1	0 0 0 0 0 0 0 0 0 0 0 0	M[MAR] ← MBR; PCL ← EARL	load the jump address from EA-reg to PC
		0x385	0	0	1 1	0 0 0 0	0	0	0	0	1	0 0 1 1 0	1 1 1 1 1	0 1 0 1 0	0 0 0 0 0 0 0 0 0 0 0 1	PCH ← EARH; jump to 0x001	
		0x386 - 0x38F	X	X	X X	X X X X	X	X	X	X	X	X X X X X	X X X X X	X X X X X	X X X X X X X X X X X X		
RTS	Inh.	0x390	0	0	0 0	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 0 1 0 0	0 0 1 0 1	0 0 0 0 0 0 0 0 0 0 0 0	MARL ← SPL; MARH ← SPH	return from subroutine (same as PULP)
-		0x391	0	0	0 0	0 1 1 1	1	0	1	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MBR ← M[MAR]; C ← 1	
		0x392	0	1	0 0	0 0 0 0	0	0	0	0	1	0 0 1 1 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	PCH ← MBR	
		0x393	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 0 1	0 0 1 0 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	SPL ← SPL + 0 + C	
		0x394	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 0 0	0 0 1 0 0	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	SPH ← SPH + 0 + C	
10		0x395	0	0	0 0	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 0 1 0 0	0 0 1 0 1	0 0 0 0 0 0 0 0 0 0 0 0	MARL ← SPL; MARH ← SPH	
		0x396	0	0	0 0	0 1 1 1	1	0	1	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MBR ← M[MAR]; C ← 1	
		0x397	0	1	0 0	0 0 0 0	0	0	0	0	1	0 0 1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	PCL ← MBR	
		0x398	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 0 1	0 0 1 0 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	SPL ← SPL + 0 + C	
		0x399	0	0	1 1	0 0 1 0	0	0	0	0	1	0 0 1 0 0	0 0 1 0 0	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 1	SPH ← SPH + 0 + C; jump to 0x001	
		0x39A - 0x3AF	X	X	X X	X X X X	X	X	X	X	X	X X X X X	X X X X X	X X X X X	X X X X X X X X X X X X		

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B				A				Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																								
RTI	Inh.	0x3B0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
from Stack		0x3B1	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← M[MAR]; C ← 1	pull CC with activated interrupts (interrupt dis. flag cleared) except from SWI where CC probably pulled with deactivated flag (as SWI not maskable)
		0x3B2	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	CC ← MBR (C still 1)	
		0x3B3	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	SPL ← SPL + 0 + C	
		0x3B4	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	SPH ← SPH + 0 + C; jump to 0x001	
		0x3B5	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
		0x3B6	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← M[MAR]; C ← 1	
		0x3B7	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	PCH ← MBR	
15		0x3B8	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	SPL ← SPL + 0 + C	
		0x3B9	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	SPH ← SPH + 0 + C	pull old PC; jump to FETCH (at least one operation between two interrupts)
		0x3BA	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
		0x3BB	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← M[MAR]; C ← 1	
		0x3BC	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	PCL ← MBR	
		0x3BD	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	SPL ← SPL + 0 + C	
		0x3BE	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	1	1	0	0	0	0	0	1	0	0	SPH ← SPH + 0 + C; jump to 0x004 (FETCH)	
		0x3BF - 0x3EF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
SWI	Inh.	0x3F0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARH ← 0	load address of SWI interrupt vector (defined 0x0004)
- ; 23		0x3F1	0	0	1	1	0	0	0	0	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	0	1	0	EARL ← 4; jump to 0x01A	jump within interrupt routine
		0x3F2 - 0x3FF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
NEGA	Inh.	0x400	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	AC ← NOT(AC) (C ← 1)	calculating 2's complement
N, Z, V, C		0x401	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	AC ← AC + 0 + C	
3		0x402	0	0	1	1	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	CC ← (N, Z, V, C) ° CC; jump to 0x001	
		0x403 - 0x42F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
COMA	Inh.	0x430	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	AC ← NOT(AC) (V ← 0, C ← 1)	calculating 1's complement
N, Z, V (0), C (1) ; 2		0x431	0	0	1	1	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	CC ← (N, Z, V, C) ° CC; jump to 0x001	
		0x432 - 0x43F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
LSRA	Inh.	0x440	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	C ← 0	logic shift right AC
N (0), Z, C		0x441	0	0	0	0	0	1	0	1	0	0	0	0	1	0	1	0	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	AC ← ror(AC)	
3		0x442	0	0	1	1	1	0	1	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	CC ← (N, Z, C) ° CC; jump to 0x001	
		0x443 - 0x45F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
RORA	Inh.	0x460	0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	ALUout ← ror(CC)	brings old carry into ALU
N, Z, C		0x461	0	0	0	0	0	1	0	1	0	0	0	0	1	0	1	0	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	AC ← ror (AC)	rotate right AC
3		0x462	0	0	1	1	1	0	1	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	CC ← (N, Z, C) ° CC; jump to 0x001	
		0x463 - 0x46F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type	MPC-Addr.	OP	Amux	Cond	ALU	MBR	MAR	RD	WR	ENC	C	B	A	Address	Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																	
ASRA	Inh.	0x470	0	0	0 1 0	0 0 1	0	0	0	0	0	1	1	0	0 0 1 1 1 0	ALUout ← AC; if N=1 jump to 0x472	arithmetic shift right AC
N, Z, C		0x471	0	0	1 1 0	0 1 1	0	0	0	0	0	1	1	0	1 0 0 0 1 1	C ← 0; jump to 0x473	
		0x472	0	0	0 0 0	0 1 1	0	0	0	0	0	1	1	1	0 0 0 0 0 0	C ← 1	
4		0x473	0	0	0 0 0	0 1 0	0	0	0	0	1	0	0	0	0 0 0 0 0 0	AC ← ror (AC)	
		0x474	0	0	1 1 0	1 0 0	0	0	0	0	1	0	0	1	1 0 0 0 0 0	CC ← (N, Z, C) ° CC; jump to 0x001	
		0x475 - 0x47F	X	X	X X X	X X X	X	X	X	X	X	X	X	X	X X X X X X		
ASLA / LSLA	Inh.	0x480	0	0	0 0 1	1 0 0	0	0	0	0	0	1	1	1	0 0 0 0 0 0	C ← 0	arithmetic/logic shift left AC
N, Z, V, C		0x481	0	0	0 0 0	0 1 0	0	0	0	0	1	0	0	0	0 0 0 0 0 0	AC ← rol(AC)	
3		0x482	0	0	1 1 1	0 0 0	0	0	0	0	1	0	1	0	0 0 0 0 0 0	CC ← (N, Z, V, C) ° CC; jump to 0x001	
		0x483 - 0x48F	X	X	X X X	X X X	X	X	X	X	X	X	X	X	X X X X X X		
ROLA	Inh.	0x490	0	0	0 0 0	0 1 0	1	0	0	0	0	1	1	0	0 0 1 0 0 0	ALUout ← ror(CC)	brings old carry into ALU
N, Z, V, C		0x491	0	0	0 0 0	0 1 0	0	0	0	0	1	0	0	1	0 0 0 0 0 0	AC ← rol (AC)	rotate left AC
2		0x492	0	0	1 1 1	0 0 0	0	0	0	0	1	0	1	0	0 0 0 0 0 1	CC ← (N, Z, V, C) ° CC; jump to 0x001	
		0x493 - 0x4EF	X	X	X X X	X X X	X	X	X	X	X	X	X	X	X X X X X X		
CLRA	Inh.	0x4F0	0	0	0 0 0	0 0 0	0	0	0	0	1	0	0	1	1 1 1 1 0 0	AC ← 0 (V ← 0)	
N (0), Z (1), V (0), C (0)		0x4F1	0	0	0 0 0	1 1 0	0	0	0	0	0	1	1	1	1 1 1 1 0 0	C ← 0	
3		0x4F2	0	0	1 1 1	0 0 0	0	0	0	0	1	0	0	1	0 0 0 0 0 0	CC ← CC ° (N, Z, V, C); jump to 0x001	
		0x4F3 - 0x4FF	X	X	X X X	X X X	X	X	X	X	X	X	X	X	X X X X X X		
PSHA	Inh.	0x500	0	0	0 0 0	1 1 0	0	0	0	0	0	1	1	1	1 1 1 1 0 0	C ← 0	
-		0x501	0	0	0 0 0	0 1 0	0	0	0	0	1	0	0	1	1 1 0 0 1 0	SPL ← SPL + 0xFF + C	subtract 1
		0x502	0	0	0 0 0	0 0 1	0	0	0	0	1	0	0	0	0 1 1 0 0 1	SPH ← SPH + 0xFF + C	
6		0x503	0	0	0 1 1	1 1 0	1	1	0	0	0	1	1	0	0 0 1 0 0 0	MARL ← SPL; MARH ← SPH	
		0x504	0	0	0 0 0	0 0 0	1	0	0	1	0	1	1	1	1 1 1 1 0 0	MBR ← AC	
		0x505	0	1	1 1 1	1 1 1	0	0	0	1	0	1	1	1	1 1 1 1 0 0	M[MAR] ← MBR; jump to 0x001	
		0x506 - 0x50F	X	X	X X X	X X X	X	X	X	X	X	X	X	X	X X X X X X		

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A					Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																										
PSHX	Inh.	0x510	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	C ← 0	
-		0x511	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0xFF + C	subtract 1
		0x512	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0xFF + C	
		0x513	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
		0x514	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← XRL	push LO-byte first (big endian; stack grows from bottom)
		0x515	0	0	0	0	1	1	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	M[MAR] ← MBR; C ← 0	
11		0x516	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0xFF + C	same for HI-byte
		0x517	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0xFF + C	
		0x518	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
		0x519	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	MBR ← XRH	
		0x51A	0	0	1	1	1	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	M[MAR] ← MBR; jump to 0x001	
		0x51B - 0x51F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
PSHY	Inh.	0x520	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	C ← 0	
-		0x521	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0xFF + C	
		0x522	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0xFF + C	
		0x523	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
		0x524	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	MBR ← YRL	
		0x525	0	0	0	0	0	1	1	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	M[MAR] ← MBR; C ← 0	
11		0x526	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0xFF + C	
		0x527	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0xFF + C	
		0x528	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
		0x529	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	MBR ← YRH	
		0x52A	0	0	1	1	1	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	M[MAR] ← MBR; jump to 0x001	
		0x52B - 0x52F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
PSHS	Inh.	0x530	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	OPRL ← SPL	push current SP value before decreasing it
-		0x531	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	OPRH ← SPH	
		0x532	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	C ← 0	
		0x533	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0xFF + C	
		0x534	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0xFF + C	
		0x535	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
		0x536	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	MBR ← OPRL	
13		0x537	0	0	0	0	0	1	1	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	M[MAR] ← MBR; C ← 0	
		0x538	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0xFF + C	
		0x539	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0xFF + C	
		0x53A	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
		0x53B	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MBR ← OPRH	
		0x53C	0	0	1	1	1	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	M[MAR] ← MBR; jump to 0x001	
		0x53D - 0x53F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction (CC changes, No. of instructions to execute)	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A					Address												Instruction Mnemonic	(Micro) Instruction Comments
PSHP	Inh.	0x540	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	C ← 0	pushes PC on the stack (points to OP-code after PSHP instruction)
-		0x541	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0xFF + C	
		0x542	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0xFF + C	
		0x543	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
		0x544	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MBR ← PCL	
		0x545	0	0	0	0	0	1	1	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	M[MAR] ← MBR; C ← 0	
11		0x546	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0xFF + C	
		0x547	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0xFF + C	
		0x548	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
		0x549	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	MBR ← PCH	
		0x54A	0	0	1	1	1	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	M[MAR] ← MBR; jump to 0x001	
		0x54B - 0x54F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
PSHC	Inh.	0x550	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	C ← 0	
-		0x551	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0xFF + C	
		0x552	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0xFF + C	
6		0x553	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
		0x554	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	MBR ← CC	
		0x555	0	0	1	1	1	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	M[MAR] ← MBR; jump to 0x001	
		0x556 - 0x57F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
PULA	Inh.	0x580	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
-		0x581	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MBR ← M[MAR]; C ← 1	
		0x582	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	AC ← MBR	
5		0x583	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0 + C	add 1
		0x584	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	SPH ← SPH + 0 + C; jump to 0x001	
		0x585 - 0x58F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
PULX	Inh.	0x590	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	same procedure as PULA, but repeat 2 times for HI- and LO byte
-		0x591	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MBR ← M[MAR]; C ← 1	
		0x592	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	XRH ← MBR	
		0x593	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0 + C	
		0x594	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0 + C	
10		0x595	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
		0x596	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MBR ← M[MAR]; C ← 1	
		0x597	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	XRL ← MBR	
		0x598	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0 + C	
		0x599	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	SPH ← SPH + 0 + C; jump to 0x001	
		0x59A - 0x59F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A					Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																									
PULY	Inh.	0x5A0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
-		0x5A1	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← M[MAR]; C ← 1	
		0x5A2	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	YRH ← MBR	
		0x5A3	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0 + C	
		0x5A4	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0 + C	
		0x5A5	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
10		0x5A6	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← M[MAR]; C ← 1	
		0x5A7	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	YRL ← MBR	
		0x5A8	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0 + C	
		0x5A9	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	SPH ← SPH + 0 + C; jump to 0x001	
		0x5AA - 0x5AF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
PULS	Inh.	0x5B0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
-		0x5B1	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← M[MAR]; C ← 1	
		0x5B2	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRH ← MBR	temporarily save HI-byte of new SP value (as LO-byte still has to be loaded, which needs to increase the old SP by 1)
		0x5B3	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0 + C	
		0x5B4	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0 + C	
9		0x5B5	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
		0x5B6	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← M[MAR]	
		0x5B7	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	SPL ← MBR	
		0x5B8	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	SPH ← OPRH; jump to 0x001	
		0x5B9 - 0x5BF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
PULP	Inh.	0x5C0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
-		0x5C1	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← M[MAR]; C ← 1	
		0x5C2	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCH ← MBR	
		0x5C3	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0 + C	
		0x5C4	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + 0 + C	
10		0x5C5	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
		0x5C6	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← M[MAR]; C ← 1	
		0x5C7	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← MBR	
		0x5C8	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0 + C	
		0x5C9	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	SPH ← SPH + 0 + C; jump to 0x001	
		0x5CA - 0x5CF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
PULC	Inh.	0x5D0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	MARL ← SPL; MARH ← SPH	
-		0x5D1	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← M[MAR]; C ← 1	
		0x5D2	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	CC ← MBR (C still 1)	
		0x5D3	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + 0 + C	
5		0x5D4	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	SPH ← SPH + 0 + C; jump to 0x001	
		0x5D5 - 0x6DF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction CC changes, No. of instructions to execute	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A					Address												Instruction Mnemonic	(Micro) Instruction Comments
JMP	@x	0x6E0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
-		0x6E1	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0x6E2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	PCL ← XRL + MBR + C	add operand to XRL
6		0x6E3	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	1	0	0	1	0	1	ALUout ← MBR; if N=1 jump to 0x6E5	
		0x6E4	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	PCH ← XRH + 0 + C; jump to 0x001	depending if the operand is positive or negative 0x00 or 0xFF has to be considered as high bye of the operand
		0x6E5	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	PCH ← XRH + 0xFF + C; jump to 0x001	
		0x6E6 - 0x7DF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
JMP	Abs.	0x7E0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
-		0x7E1	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MBR ←MEM[MAR]; C ← 1	
		0x7E2	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	EARH ← MBR	temporarily save new HI-byte of PC as before PC has to be increased to read the LO-byte of new PC
		0x7E3	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x7E4	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
9		0x7E5	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0x7E6	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MBR ←MEM[MAR]	
		0x7E7	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	PCL ← MBR	directly load to PCL as old PC not needed anymore
		0x7E8	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	PCH ← EARH; jump to 0x001	
		0x7E9 - 0x7FF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
SUBA	Imm.	0x800	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0x801	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0x802	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	OPRL ← NOT(MBR) (C ← 1)	invert
		0x803	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	AC ← AC + OPRL + C	add + 1 (2's complement)
8		0x804	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	CC ← (N, Z, V, C) ° CC	
		0x805	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	C ← 1	
		0x806	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x807	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x808 - 0x80F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
CMPA	Imm.	0x810	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	CMPA same as SUBA without saving the result in AC
N, Z, V, C		0x811	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0x812	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	OPRL ← NOT(MBR) (C ← 1)	
		0x813	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	1	1	1	0	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	ALUout ← AC + OPRL + C	
8		0x814	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	CC ← (N, Z, V, C) ° CC	
		0x815	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	C ← 1	
		0x816	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x817	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x818 - 0x81F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A					Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																										
SBCA	Imm.	0x820	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	subtraction with carry
N, Z, V, C		0x821	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0x822	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRL ← NOT(MBR)	
		0x823	0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	ALUout ← ror(CC)	brings old carry into ALU
		0x824	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	AC ← AC + OPRL + C	
9		0x825	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	CC ← (N, Z, V, C) ° CC	
		0x826	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	C ← 1	
		0x827	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x828	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	1	1	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x829 - 0x83F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
ANDA	Imm.	0x840	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0x841	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0x842	0	1	0	0	0	0	1	1	0	0	0	0	1	0	1	0	0	0	0	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	AC ← AC & MBR (V ← 0, C still 1)	
		0x843	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	CC ← (N, Z, V) ° CC	
6		0x844	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x845	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x846 - 0x85F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
LDA	Imm.	0x860	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0x861	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0x862	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	AC ← MBR (V ← 0, C still 1)	
		0x863	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	CC ← (N, Z, V) ° CC (C still 1)	
6		0x864	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x865	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x866 - 0x86F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
Codeend of STA		0x870	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
		0x871	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	MBR ← AC (V ← 0)	
		0x872	0	0	0	0	0	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	M[MAR] ← MBR; C ← 1 (other flags unaffected)	
		0x873	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	CC ← (N, Z, V) ° CC (C still 1)	
		0x874	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x875	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x876 - 0x87F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A					Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																										
EORA	Imm.	0x880	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	only AND and NOT usable:
N, Z, V (0)		0x881	0	0	0	0	0	0	0	1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; U ← NOT(AC)	NOT(NOT(A AND NOT(B)) AND NOT(NOT(A) AND B));
		0x882	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRL ← MBR	because of the total negation it is better using XNOR:
		0x883	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1	1	1	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	V ← NOT(OPRL)	NOT(NOT(A) AND NOT(B)) AND (NOT(A AND B))
		0x884	0	0	0	0	0	0	1	1	0	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	AC ← AC AND OPRL	AC: A; OPRL: B; U: NOT(A); V: NOT(B)
		0x885	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	AC ← NOT(AC)	AC: NOT(A AND B)
	11	0x886	0	0	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	U ← U AND V	
		0x887	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	U ← NOT(U) (C ← 1)	U: NOT(NOT(A) AND NOT(B))
		0x888	0	0	0	0	0	0	1	1	0	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	AC ← AC AND U (C still 1)	
		0x889	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x88A	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x88B - 0x88F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ADCA	Imm.	0x890	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0x891	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0x892	0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	ALUout ← ror(CC)	brings old carry into ALU
		0x893	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	AC ← AC + MBR + C	
	8	0x894	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	CC ← (N, Z, V, C) " CC	
		0x895	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	C ← 1	
		0x896	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x897	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x898 - 0x89F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ORA	Imm.	0x8A0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0x8A1	0	0	0	0	0	0	0	1	1	0	1	0	1	0	1	0	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; AC ← NOT(AC)	
		0x8A2	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRL ← NOT(MBR)	
		0x8A3	0	0	0	0	0	0	1	1	0	0	0	0	1	0	1	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	AC ← AC NOT OPRL	
		0x8A4	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	AC ← NOT(AC) (V ← 0, C ← 1)	
	8	0x8A5	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	CC ← (N, Z, V) " CC	
		0x8A6	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x8A7	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x8A8 - 0x8AF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ADDA	Imm.	0x8B0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0x8B1	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	addition without carry (therefore clear carry)
		0x8B2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	AC ← MBR + AC + C	
		0x8B3	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	CC ← (N, Z, V, C) " CC	
	7	0x8B4	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	C ← 1	
		0x8B5	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0x8B6	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0x8B7 - 0x8BF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type	CC changes, No. of instructions to execute	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B				A				Address										Instruction Mnemonic	(Micro) Instruction Comments
CMPX	Imm.		0x8C0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C			0x8C1	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
			0x8C2	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
			0x8C3	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
			0x8C4	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
			0x8C5	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
			0x8C6	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	load negated LO-byte (subtrahend) and subtract it from the LO-byte of X-reg using addition with 2's complement of the subtrahend and save directly to temp U-reg
			0x8C7	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	OPRL ← NOT(MBR) (C ← 1)	
		14 / 15 / 16	0x8C8	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	U ← XRL + OPRL + C	
			0x8C9	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	1	1	1	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	ALUout ← XRH + OPRH + C	now subtract the HI-byte which was loaded from the memory (subtrahend) using 2's complement addition (add with carry from LO-byte addition) from the HI-byte of X-reg (sets N-, V- and C-flag correctly); then the status flags are updated, if Z=0 the flags are already correct, otherwise the LO-byte of the subtraction (stored in U) has to be checked if it is zero too and only the Z-flag is updated again if it has to be changed to zero
			0x8CA	0	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	1	0	0	0	1	1	0	0	1	1	1	0	0	0	CC ← (N, Z, V, C) ° CC; if Z=1 jump to 0x8CE	
			0x8CB	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	C ← 1	
			0x8CC	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
			0x8CD	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
			0x8CE	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	1	0	1	1	1	1	ALUout ← U; if Z=1 jump to 0x8CB	
			0x8CF	0	0	1	1	1	0	1	1	0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	1	1	0	0	0	1	1	0	0	1	1	CC ← (Z) ° CC; jump to 0x8CB	
			0x8D0 - 0x8DF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
LDX	Imm.		0x8E0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)			0x8E1	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ←MEM[MAR]; C ← 1	
			0x8E2	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	OPRH ← MBR (C still 1)	load HI-byte to OPRH
			0x8E3	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
			0x8E4	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
			0x8E5	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
			0x8E6	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	MBR ←MEM[MAR]; C ← 1	
			0x8E7	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	XRL ← MBR (C still 1)	save LO-byte directly to final destination XRL
			0x8E8	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	XRH ← OPRH (V ← 0, C still 1)	
			0x8E9	0	0	1	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	1	1	0	0	0	1	1	1	0	1	0	CC ← (N, Z, V) ° CC; if Z=1 jump to 0x8EC (C still 1)	now save HI-byte to final destination XRH (sets N-flag correctly); then the status flags are updated, if Z=0 the flags are already correct, otherwise also the LO-byte has to be checked if it is zero and only the Z-flag is updated again if it has to be changed to zero
			0x8EA	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
			0x8EB	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; j ump to 0x001	
			0x8EC	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	1	0	1	1	0	1	ALUout ← XRL; if Z=1 jump to 0x8EA (C still 1)	
			0x8ED	0	0	1	1	1	0	1	1	0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	1	1	0	0	0	1	1	1	0	1	0	CC ← (Z) ° CC; jump to 0x8EA (C still 1)	
			0x8EE - 0x8EF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
Codeend of STX			0x8F0	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
			0x8F1	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	MBR ← XRH (V ← 0)	save HI-byte of X-reg (XRH) to the destination address and update flags (N and V are already set correctly, but Z might be wrong for complete X-reg)
			0x8F2	0	0	0	0	0	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	M[MAR] ← MBR; C ← 1 (other flags unaffected)	
			0x8F3	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	CC ← (N, Z, V) ° CC (C still 1)	
			0x8F4	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	increase the effective address-register to also store the LO-byte at the next address in the memory
			0x8F5	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
			0x8F6	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
			0x8F7	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	MBR ← XRL	
			0x8F8	0	0	1	0	0	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1	0	0	0	0	M[MAR] ← MBR; C ← 1; if Z=1 jump to 0x8FC	save now LO-byte of X-reg (XRL); if Z=0 no jump is performed and the Z-flag is updated accordingly; if Z=1 (XRL is zero) also XRH has to be checked if it is zero before updating the Z-flag
			0x8F9	0	0	0	0	1	0	1	1	0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	CC ← (Z) ° CC (C still 1)	
			0x8FA	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
			0x8FB	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
			0x8FC	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	1	0	1	0	0	0	1	1	1	1	1	ALUout ← XRH; jump to 0x8F9 (C still 1)	

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A				Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																									
		0x8FD - 0x8FF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
SUBA	@y	0x900	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0x901	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0x902	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0x903	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	0	0	0	1	ALUout ← MBR; if N=1 jump to 0x905	
14		0x904	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	0	0	1	0	0	0	0	1	EARH ← YRH + 0 + C; jump to 0x906	
		0x905	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
		0x906	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	0	0	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x801	continue within SUBA Imm. (0x800)
		0x907 - 0x90F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
CMPA	@y	0x910	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0x911	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0x912	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0x913	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	0	0	1	0	1	ALUout ← MBR; if N=1 jump to 0x915	
14		0x914	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	0	0	1	0	0	0	1	1	0	EARH ← YRH + 0 + C; jump to 0x916	
		0x915	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
		0x916	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	0	0	1	0	1	MARH ← EARH; MARL ← EARL; jump to 0x811	continue within CMPA Imm. (0x810)
		0x917 - 0x91F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
SBCA	@y	0x920	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0x921	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0x922	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0x923	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	0	1	0	1	ALUout ← MBR; if N=1 jump to 0x925	
15		0x924	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	0	0	1	0	0	1	1	0	EARH ← YRH + 0 + C; j ump to 0x926	
		0x925	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
		0x926	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	0	0	1	0	1	MARH ← EARH; MARL ← EARL; jump to 0x821	continue within SBCA Imm. (0x820)
		0x927 - 0x93F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ANDA	@y	0x940	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0x941	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0x942	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0x943	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	0	0	1	0	1	ALUout ← MBR; if N=1 jump to 0x945	
12		0x944	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	0	0	1	0	1	0	0	1	1	0	EARH ← YRH + 0 + C; jump to 0x946	
		0x945	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
		0x946	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	0	0	1	0	1	MARH ← EARH; MARL ← EARL; jump to 0x841	continue within ANDA Imm. (0x840)
		0x947 - 0x95F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type CC changes, No. of instructions to execute	MPC-Addr.	OP	Amux	Cond	ALU	MBR	MAR	RD	WR	ENC	C	B	A	Address										Instruction Mnemonic	(Micro) Instruction Comments
LDA	@y	0x960	0	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0x961	0	0	0	0	1	1	0	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0x962	0	1	0	0	0	0	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0x963	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	1	1	0	ALUout ← MBR; if N=1 jump to 0x965	
12		0x964	0	0	1	1	0	0	1	0	0	0	0	0	1	1	1	1	1	0	0	1	1	0	EARH ← YRH + 0 + C; jump to 0x966	
		0x965	0	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	0	0	0	EARH ← YRH + 0xFF + C	
		0x966	0	0	1	1	1	1	1	0	1	1	0	0	1	1	1	1	0	0	0	1	1	0	MARH ← EARH; MARL ← EARL; jump to 0x861	continue within LDA Imm. (0x860)
		0x967 - 0x96F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
STA	@y	0x970	0	0	0	1	1	1	0	1	0	0	1	1	0	0	1	1	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0x971	0	0	0	0	1	1	0	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0x972	0	1	0	0	0	0	1	0	0	1	0	0	1	1	1	1	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0x973	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	1	1	0	ALUout ← MBR; if N=1 jump to 0x975	
11		0x974	0	0	1	1	0	0	0	0	0	1	0	0	0	0	1	1	1	1	0	0	0	0	EARH ← YRH + 0 + C; jump to 0x870	continue at codeend of STA
		0x975	0	0	1	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0	1	1	0	EARH ← YRH + 0xFF + C; jump to 0x870	
		0x976 - 0x97F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
EORA	@y	0x980	0	0	0	1	1	1	0	1	0	0	1	1	0	0	1	1	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0x981	0	0	0	0	1	1	0	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0x982	0	1	0	0	0	0	1	0	0	1	0	0	1	1	1	1	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0x983	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	1	1	0	ALUout ← MBR; if N=1 jump to 0x985	
17		0x984	0	0	1	1	0	0	0	0	0	1	0	0	0	0	1	1	1	1	0	0	0	0	EARH ← YRH + 0 + C; jump to 0x986	
		0x985	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0	1	1	0	EARH ← YRH + 0xFF + C	
		0x986	0	1	1	1	1	1	1	0	1	1	0	0	1	0	0	1	1	0	0	0	1	0	MARH ← EARH; MARL ← EARL; jump to 0x881	continue within EORA Imm. (0x880)
		0x987 - 0x98F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ADCA	@y	0x990	0	0	0	1	1	1	0	1	0	0	1	1	0	0	1	1	1	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0x991	0	0	0	0	1	1	0	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0x992	0	1	0	0	0	0	1	0	0	1	0	0	1	1	1	1	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0x993	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	1	1	0	ALUout ← MBR; if N=1 jump to 0x995	
14		0x994	0	0	1	1	0	0	0	0	0	1	0	0	0	0	1	1	1	1	0	0	0	0	EARH ← YRH + 0 + C; jump to 0x996	
		0x995	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0	1	1	0	EARH ← YRH + 0xFF + C	
		0x996	0	1	1	1	1	1	1	0	1	1	0	0	1	0	0	1	1	0	0	0	1	0	MARH ← EARH; MARL ← EARL; jump to 0x891	continue within ADCA Imm. (0x890)
		0x997 - 0x99F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ORA	@y	0x9A0	0	0	0	0	1	1	1	1	0	1	1	1	0	0	1	1	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0x9A1	0	0	0	0	1	1	0	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0x9A2	0	1	0	0	0	0	1	0	0	1	0	0	1	1	1	1	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0x9A3	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	1	1	0	ALUout ← MBR; if N=1 jump to 0x9A5	
14		0x9A4	0	0	1	1	0	0	0	0	0	1	0	0	0	0	1	1	1	1	0	0	1	1	EARH ← YRH + 0 + C; jump to 0x9A6	
		0x9A5	0	0	0	0	0	1	0	0	0	1	0	0	1	1	0	0	1	1	0	0	0	0	EARH ← YRH + 0xFF + C	
		0x9A6	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	0	0	0	1	1	0	MARH ← EARH; MARL ← EARL; jump to 0x8A1	continue within ORA Imm. (0x8A0)

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A				Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																									
		0x9A7 - 0x9AF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ADDA	@y	0x9B0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0x9B1	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0x9B2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0x9B3	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	1	1	ALUout ← MBR; if N=1 jump to 0x9B5	
	13	0x9B4	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0	1	1	EARH ← YRH + 0 + C; jump to 0x9B6	
		0x9B5	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
		0x9B6	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	1	0	1	1	0	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x8B1	continue within ADDA Imm. (0x8B0)
		0x9B7 - 0x9BF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
CMPX	@y	0x9C0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0x9C1	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0x9C2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0x9C3	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	0	1	ALUout ← MBR; if N=1 jump to 0x9C5	
		0x9C4	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	1	0	0	EARH ← YRH + 0 + C; jump to 0x9C6	
		0x9C5	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
	20 / 21 / 22	0x9C6	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
		0x9C7	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0x9C8	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0x9C9	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0x9CA	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0x9CB	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	1	1	0	0	MARH ← EARH; MARL ← EARL; jump to 0x8C6	continue within CMPX Imm. (0x8C0)
		0x9CC - 0x9DF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
LDX	@y	0x9E0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0x9E1	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0x9E2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0x9E3	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	0	1	ALUout ← MBR; if N=1 jump to 0x9E5	
		0x9E4	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	EARH ← YRH + 0 + C; jump to 0x9E6	
		0x9E5	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
		0x9E6	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
	18 / 19 / 20	0x9E7	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0x9E8	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	OPRH ← MBR (C still 1)	load HI-byte to OPRH
		0x9E9	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0x9EA	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0x9EB	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	1	1	0	0	MARH ← EARH; MARL ← EARL; jump to 0x8E6	continue within LDX Imm. (0x8E0)
		0x9EC - 0x9EF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A					Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																										
STX	@y	0x9F0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0x9F1	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0x9F2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
17 / 18		0x9F3	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	0	1	0	1	ALUout ← MBR; if N=1 jump to 0x9F5
		0x9F4	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0	0	EARH ← YRH + 0 + C; jump to 0x8F0	continue at codeend of STX
		0x9F5	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	0	EARH ← YRH + 0xFF + C; jump to 0x8F0	
		0x9F6 - 0x9FF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
SUBA	@x	0xA00	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xA01	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xA02	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xA03	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	1	0	1	ALUout ← MBR; if N=1 jump to 0xA05
14		0xA04	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	1	1	1	1	0	1	0	0	0	0	0	1	1	0	EARH ← XRH + 0 + C; jump to 0xA06
		0xA05	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	EARH ← XRH + 0xFF + C	
		0xA06	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	0	0	0	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x801	continue within SUBA Imm. (0x800)
		0xA07 - 0xA0F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
CMPA	@x	0xA10	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xA11	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xA12	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xA13	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	1	0	1	ALUout ← MBR; if N=1 jump to 0xA15
14		0xA14	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	1	1	1	1	0	1	0	0	0	0	1	0	1	1	0	EARH ← XRH + 0 + C; jump to 0xA16
		0xA15	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	EARH ← XRH + 0xFF + C	
		0xA16	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	0	0	1	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x811	continue within CMPA Imm. (0x810)
		0xA17 - 0xA1F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
SBCA	@x	0xA20	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xA21	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xA22	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xA23	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	1	0	1	0	1	ALUout ← MBR; if N=1 jump to 0xA25
15		0xA24	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	1	1	1	1	0	1	0	0	0	1	0	1	1	0	EARH ← XRH + 0 + C; jump to 0xA26
		0xA25	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	EARH ← XRH + 0xFF + C	
		0xA26	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	0	0	1	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x821	continue within SBCA Imm. (0x820)
		0xA27 - 0xA3F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ANDA	@x	0xA40	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xA41	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xA42	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xA43	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	1	0	1	ALUout ← MBR; if N=1 jump to 0xA45
12		0xA44	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	1	1	1	1	0	1	0	0	1	0	0	0	1	1	0	EARH ← XRH + 0 + C; jump to 0xA46
		0xA45	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	EARH ← XRH + 0xFF + C	
		0xA46	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	0	0	1	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x841	continue within ANDA Imm. (0x840)

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A				Address									Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																								
		0xA47 - 0xA5F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
LDA	@x	0xA60	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xA61	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xA62	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xA63	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	1	1	ALUout ← MBR; if N=1 jump to 0xA65	
	12	0xA64	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	1	1	1	1	0	1	0	0	1	1	0	EARH ← XRH + 0 + C; jump to 0xA66	
		0xA65	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← XRH + 0xFF + C	
		0xA66	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	0	1	0	0	0	0	1	1	0	0	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x861	continue within LDA Imm. (0x860)
		0xA67 - 0xA6F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
STA	@x	0xA70	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xA71	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xA72	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xA73	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	1	1	ALUout ← MBR; if N=1 jump to 0xA75	
	11	0xA74	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	1	1	1	1	0	0	0	0	1	1	0	EARH ← XRH + 0 + C; jump to 0x870	continue at codeend of STA
		0xA75	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	0	1	1	0	0	0	0	1	1	0	EARH ← XRH + 0xFF + C; jump to 0x870	
		0xA76 - 0xA7F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
EORA	@x	0xA80	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xA81	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xA82	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xA83	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	1	ALUout ← MBR; if N=1 jump to 0xA85	
	17	0xA84	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	1	1	1	1	0	1	0	0	0	0	1	EARH ← XRH + 0 + C; jump to 0xA86	
		0xA85	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← XRH + 0xFF + C	
		0xA86	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	0	1	0	0	0	1	0	0	0	0	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x881	continue within EORA Imm. (0x880)
		0xA87 - 0xA8F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ADCA	@x	0xA90	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xA91	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xA92	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xA93	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	0	1	ALUout ← MBR; if N=1 jump to 0xA95	
	14	0xA94	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	1	1	1	1	0	1	0	0	1	0	1	1	0	EARH ← XRH + 0 + C; jump to 0xA96	
		0xA95	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← XRH + 0xFF + C	
		0xA96	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	0	1	0	0	0	1	0	0	0	0	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x891	continue within ADCA Imm. (0x890)
		0xA97 - 0xA9F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B				A				Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																								
ORA	@x	0xAA0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xAA1	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xAA2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xAA3	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	0	0	1	ALUout ← MBR; if N=1 jump to 0xAA5	
	14	0xAA4	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	1	1	1	1	0	1	0	1	0	0	1	EARH ← XRH + 0 + C; jump to 0xAA6	
		0xAA5	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← XRH + 0xFF + C	
		0xAA6	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	1	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x8A1	continue within ORA Imm. (0x8A0)
		0xAA7 - 0xAAF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ADDA	@x	0xAB0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xAB1	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xAB2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xAB3	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	0	1	0	1	ALUout ← MBR; if N=1 jump to 0xAB5	
	13	0xAB4	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	1	1	1	1	0	1	0	1	1	0	1	EARH ← XRH + 0 + C; jump to 0xAB6	
		0xAB5	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← XRH + 0xFF + C	
		0xAB6	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	1	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x8B1	continue within ADDA Imm. (0x8B0)
		0xAB7 - 0xABF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
CMPX	@x	0xAC0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xAC1	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xAC2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xAC3	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	0	0	1	0	1	ALUout ← MBR; if N=1 jump to 0xAC5	
		0xAC4	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	1	1	1	1	0	1	0	1	1	0	0	EARH ← XRH + 0 + C; jump to 0xAC6	
		0xAC5	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← XRH + 0xFF + C	
	20 / 21 / 22	0xAC6	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
		0xAC7	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xAC8	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xAC9	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xACA	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xACB	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	1	0	1	1	0	0	0	1	1	0	0	1	1	0	MARH ← EARH; MARL ← EARL; jump to 0x8C6	continue within CMPX Imm. (0x8C0)
		0xACC - 0xADF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
LDX	@x	0xAE0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xAE1	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xAE2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xAE3	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	0	1	0	1	ALUout ← MBR; if N=1 jump to 0xAE5	
		0xAE4	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	1	1	1	1	0	1	0	1	1	1	0	EARH ← XRH + 0 + C; jump to 0xAE6	
		0xAE5	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← XRH + 0xFF + C	
		0xAE6	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	1	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
	18 / 19 / 20	0xAE7	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xAE8	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	OPRH ← MBR (C still 1)	load HI-byte to OPRH
		0xAE9	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0	1	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xAEA	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xAEB	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	1	0	1	1	0	0	0	1	1	0	0	1	1	0	MARH ← EARH; MARL ← EARL; jump to 0x8E6	continue within LDX Imm. (0x8E0)

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B				A				Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																								
		0xAEC - 0xAEF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
STX	@x	0xAF0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xAF1	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xAF2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xAF3	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	1	0	1	0	1	ALUout ← MBR; if N=1 jump to 0xAF5	
17 / 18		0xAF4	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0	EARH ← XRH + 0 + C; jump to 0x8F0	continue at codeend of STX
		0xAF5	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	EARH ← XRH + 0xFF + C; jump to 0x8F0	
		0xAF6 - 0xAFF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
SUBA	Abs.	0xB00	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xB01	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xB02	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARH ← MBR (C still 1)	
		0xB03	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xB04	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	0	1	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
16		0xB05	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	0	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0xB06	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xB07	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← MBR	
		0xB08	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	0	0	0	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x801	continue within SUBA Imm. (0x800)
		0xB09 - 0xB0F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
CMPA	Abs.	0xB10	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xB11	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xB12	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARH ← MBR (C still 1)	
		0xB13	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xB14	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	0	1	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
16		0xB15	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	0	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0xB16	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xB17	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← MBR	
		0xB18	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	0	0	1	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x811	continue within CMPA Imm. (0x810)
		0xB19 - 0xB1F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
SBCA	Abs.	0xB20	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xB21	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xB22	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARH ← MBR (C still 1)	
		0xB23	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xB24	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	0	1	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
17		0xB25	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0xB26	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xB27	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← MBR	
		0xB28	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	0	0	1	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x821	continue within BCA Imm. (0x820)
		0xB29 - 0xB3F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type CC changes, No. of instructions to execute	MPC-Addr.	OP	Amux	Cond	ALU	MBR	MAR	RD	WR	ENC	C	B	A	Address	Instruction Mnemonic	(Micro) Instruction Comments
ANDA	Abs.	0x840	0	0	0 0 1	1 1 1	0	1	1	0	0	1 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0x841	0	0	0 0 0	0 1 1	1	0	1	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
		0x842	0	1	0 0 0	0 0 0	0	0	0	0	1	0 1 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	EARH ← MBR (C still 1)	
		0x843	0	0	0 0 0	0 1 0	0	0	0	0	1	0 0 1	1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	PCL ← PCL + 0 + C	
		0x844	0	0	0 0 0	0 0 1	0	0	0	0	1	1 1 0	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0	PCH ← PCH + 0 + C	
14		0x845	0	0	0 0 0	1 1 1	1	0	1	1	0	0 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
		0x846	0	0	0 0 0	1 1 1	1	0	1	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]	
		0x847	0	1	0 0 0	0 0 0	0	0	0	0	1	0 1 1	1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	EARL ← MBR	
		0x848	0	0	1 1 1	1 1 1	1	0	1	1	0	0 1 1	0 1 0 1 1	1 1 0 0 1	0 0 0 1 0	MARH ← EARH; MARL ← EARL; jump to 0x841	continue within ANDA Imm. (0x840)
		0x849 - 0x85F	x	x	x x x	x x x	x	x	x	x	x	x x x	x x x x x	x x x x x	x x x x x	x	
LDA	Abs.	0x860	0	0	0 0 1	1 1 1	0	1	1	0	0	1 1 0	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0x861	0	0	0 0 0	0 1 1	1	0	1	0	0	1 1 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
		0x862	0	1	0 0 0	0 0 0	0	0	0	0	1	0 1 0	1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	EARH ← MBR (C still 1)	
		0x863	0	0	0 0 0	0 1 0	0	0	0	0	1	0 1 1	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0	PCL ← PCL + 0 + C	
		0x864	0	0	0 0 0	1 0 0	0	0	0	0	1	0 1 1	0 0 0 1 1	1 1 1 1 1	0 0 0 0 0	PCH ← PCH + 0 + C	
14		0x865	0	0	0 0 1	1 1 1	1	0	1	1	0	0 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
		0x866	0	0	0 0 1	1 1 1	1	0	1	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]	
		0x867	0	1	0 0 0	0 0 0	0	0	0	0	1	0 1 1	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0	EARL ← MBR	
		0x868	0	1	1 1 1	1 1 1	1	0	1	1	0	0 1 1	0 1 0 1 1	0 0 1 1 1	1 1 0 0 1	MARH ← EARH; MARL ← EARL; jump to 0x861	continue within LDA Imm. (0x860)
		0x869 - 0x86F	x	x	x x x	x x x	x	x	x	x	x	x x x	x x x x x	x x x x x	x x x x x	x	
STA	Abs.	0x870	0	0	0 0 1	1 1 1	0	1	1	0	0	1 1 0	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0x871	0	0	0 0 1	1 1 1	1	0	1	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
		0x872	0	1	0 0 0	0 0 0	0	0	0	0	1	0 1 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	EARH ← MBR (C still 1)	
		0x873	0	0	0 0 0	0 1 0	0	0	0	0	1	0 0 1	1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	PCL ← PCL + 0 + C	
		0x874	0	0	0 0 0	0 0 1	0	0	0	0	1	1 1 0	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0	PCH ← PCH + 0 + C	
14		0x875	0	0	0 1 1	1 1 1	1	0	1	1	0	0 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
		0x876	0	0	0 1 1	1 1 1	1	0	1	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]	
		0x877	0	1	1 1 0	0 0 0	0	0	0	0	1	0 0 1	1 1 1 1 1	1 1 0 0 0	0 0 0 1 1	EARL ← MBR; jump to 0x870	continue within codeend of STA
		0x878 - 0x87F	x	x	x x x	x x x	x	x	x	x	x	x x x	x x x x x	x x x x x	x x x x x	x	
EORA	Abs.	0x880	0	0	0 1 1	1 1 1	0	1	1	0	0	1 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0x881	0	0	0 0 0	0 1 1	1	0	1	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
		0x882	0	1	0 0 0	0 0 0	0	0	0	0	1	0 1 0	1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	EARH ← MBR (C still 1)	
		0x883	0	0	0 0 0	0 1 0	0	0	0	0	1	0 0 1	1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	PCL ← PCL + 0 + C	
		0x884	0	0	0 0 0	0 0 1	0	0	0	0	1	1 1 0	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0	PCH ← PCH + 0 + C	
19		0x885	0	0	0 0 1	1 1 1	1	0	1	1	0	0 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
		0x886	0	0	0 0 1	1 1 1	1	0	1	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]	
		0x887	0	1	0 0 0	0 0 0	0	0	0	0	1	0 1 1	1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	EARL ← MBR	
		0x888	0	1	1 1 1	1 1 1	1	0	1	0	0	1 1 1	0 1 0 1 1	1 1 0 0 1	0 0 0 1 0	MARH ← EARH; MARL ← EARL; jump to 0x881	continue within EORA Imm. (0x880)
		0x889 - 0x88F	x	x	x x x	x x x	x	x	x	x	x	x x x	x x x x x	x x x x x	x x x x x	x	

Instruction	Type	MPC-Addr.	OP	Amux	Cond	ALU	MBR	MAR	RD	WR	ENC	C	B	A	Address										Instruction Mnemonic	(Micro) Instruction Comments
	CC changes, No. of instructions to execute																									
ADCA	Abs.	0xB90	0	0	0 0	1	1	0	1	1	0	1	1 0	0 0	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xB91	0	0	0 0	1	1	1	0	1	0	0	1 1	1 1	1	1	1	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xB92	0	1	0 0	0	0	0	0	0	0	1	0 1	1 1	1	1	1	0	0	0	0	0	0	0	EARH ← MBR (C still 1)	
		0xB93	0	0	0 0	0	1	0	0	0	1	0	0 1	1 1	1	1	1	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xB94	0	0	0 0	0	0	1	0	0	0	1	1 0	0 0	1	1	1	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
16		0xB95	0	0	0 0	1	1	1	0	1	0	0	0 1	1 0	0	0	1	1	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0xB96	0	0	0 0	1	1	1	0	1	0	0	1 1	1 1	1	1	1	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xB97	0	1	0 0	0	0	0	0	0	1	0	1 1	1 1	1	1	1	0	0	0	0	0	0	0	EARL ← MBR	
		0xB98	0	0	1 1	1	1	1	0	1	1	0	0 1	1 0	1	1	1	0	0	0	1	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x891	continue within ADCA Imm. (0x890)
		0xB99 - 0xB9F	X	X	X X	X	X	X	X	X	X	X	X X	X X	X	X	X	X	X	X	X	X	X	X		
ORA	Abs.	0xBA0	0	0	0 0	1	1	1	0	1	0	1	1 0	0 0	1	1	1	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xBA1	0	0	0 0	0	1	1	0	1	0	0	1 1	1 0	1	1	1	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xBA2	0	1	0 0	0	0	0	0	0	0	1	0 1	1 0	1	1	1	0	0	0	0	0	0	0	EARH ← MBR (C still 1)	
		0xBA3	0	0	0 0	0	0	0	0	0	0	1	0 1	1 0	1	1	1	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xBA4	0	0	0 0	0	0	1	0	0	0	1	0 1	1 0	1	1	1	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
16		0xBA5	0	0	0 0	1	1	1	0	1	0	0	1 1	1 0	1	1	1	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0xBA6	0	0	0 0	1	1	1	0	1	0	0	1 1	1 0	1	1	1	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xBA7	0	1	0 0	0	0	0	0	0	0	1	0 1	1 1	1	1	1	0	0	0	0	0	0	0	EARL ← MBR	
		0xBA8	0	1	1 1	1	1	1	0	1	1	0	1 1	1 0	1	1	1	0	0	0	1	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x8A1	continue within ANDA Imm. (0x8A0)
		0xBA9 - 0xBAF	X	X	X X	X	X	X	X	X	X	X	X X	X X	X	X	X	X	X	X	X	X	X	X		
ADDA	Abs.	0xBB0	0	0	0 0	1	1	1	0	1	0	1	1 0	0 0	1	1	1	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xBB1	0	0	0 0	1	1	1	0	1	0	0	1 1	1 0	1	1	1	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xBB2	0	1	0 0	0	0	0	0	0	0	1	0 1	1 0	1	1	1	0	0	0	0	0	0	0	EARH ← MBR (C still 1)	
		0xBB3	0	0	0 0	0	1	0	0	0	1	0	0 1	1 0	1	1	1	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xBB4	0	0	0 0	0	0	0	0	0	0	1	0 0	0 1	1	1	1	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
15		0xBB5	0	0	0 0	1	1	1	0	1	0	0	1 1	1 0	0	0	1	1	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0xBB6	0	0	0 0	1	1	1	0	1	0	0	1 1	1 0	1	1	1	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xBB7	0	1	0 0	0	0	0	0	0	1	0	1 1	1 0	1	1	1	0	0	0	0	0	0	0	EARL ← MBR	
		0xBB8	0	1	1 1	1	1	1	0	1	1	0	1 1	1 0	1	1	1	0	0	0	1	0	0	1	MARH ← EARH; MARL ← EARL; jump to 0x8B1	continue within ADDA Imm. (0x8B0)
		0xBB9 - 0xBBF	X	X	X X	X	X	X	X	X	X	X	X X	X X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type	MPC-Addr.	OP	Amux	Cond	ALU	MBR	MAR	RD	WR	ENC	C	B	A	Address	Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																	
CMPX	Abs.	0xBC0	0	0	00	1111	0	1	1	0	0	11111	00110	00111	0000 0000 0000	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xBC1	0	0	00	0111	1	0	1	0	0	11111	11111	11111	0000 0000 0000	MBR ← MEM[MAR]; C ← 1	
		0xBC2	0	1	00	0000	0	0	0	0	1	01010	11111	11111	0000 0000 0000	EARH ← MBR (C still 1)	
		0xBC3	0	0	00	0010	0	0	0	0	1	00111	00111	11111	0000 0000 0000	PCL ← PCL + 0 + C	
		0xBC4	0	0	00	0010	0	0	0	0	1	00110	00110	11111	0000 0000 0000	PCH ← PCH + 0 + C	
		0xBC5	0	0	00	1111	0	1	1	0	0	11111	00110	00111	0000 0000 0000	MARH ← PCH; MARL ← PCL	
		0xBC6	0	0	00	0111	1	0	1	0	0	11111	11111	11111	0000 0000 0000	MBR ← MEM[MAR]; C ← 1	
22 / 23 / 24		0xBC7	0	1	00	0000	0	0	0	0	1	01011	11111	11111	0000 0000 0000	EARL ← MBR (C still 1)	
		0xBC8	0	0	00	1111	0	1	1	0	0	11111	01010	01011	0000 0000 0000	MARH ← EARH; MARL ← EARL	
		0xBC9	0	0	00	1111	1	0	1	0	0	11111	11111	11111	0000 0000 0000	MBR ← MEM[MAR]	
		0xBCA	0	1	00	0001	0	0	0	0	1	01100	11111	11111	0000 0000 0000	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xBCB	0	0	00	0010	0	0	0	0	1	01011	01011	11111	0000 0000 0000	EARL ← EARL + 0 + C	
		0xBCC	0	0	00	0010	0	0	0	0	1	01010	01010	11111	0000 0000 0000	EARH ← EARH + 0 + C	
		0xBCD	0	0	11	1111	0	1	1	0	0	11111	01010	01011	1000 1100 0110	MARH ← EARH; MARL ← EARL; jump to 0x8C6	continue within CMPX Imm. (0x8C0)
		0xBCE - 0xBDF	X	X	XX	XXXX	X	X	X	X	X	XXXXX	XXXXX	XXXXX	XXXX XXXX XXXX		
LDX	Abs.	0xBE0	0	0	00	1111	0	1	1	0	0	11111	00110	00111	0000 0000 0000	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xBE1	0	0	00	0111	1	0	1	0	0	11111	11111	11111	0000 0000 0000	MBR ← MEM[MAR]; C ← 1	
		0xBE2	0	1	00	0000	0	0	0	0	1	01010	11111	11111	0000 0000 0000	EARH ← MBR (C still 1)	
		0xBE3	0	0	00	0010	0	0	0	0	1	00111	00111	11111	0000 0000 0000	PCL ← PCL + 0 + C	
		0xBE4	0	0	00	0010	0	0	0	0	1	00110	00110	11111	0000 0000 0000	PCH ← PCH + 0 + C	
		0xBE5	0	0	00	1111	0	1	1	0	0	11111	00110	00111	0000 0000 0000	MARH ← PCH; MARL ← PCL	
		0xBE6	0	0	00	0111	1	0	1	0	0	11111	11111	11111	0000 0000 0000	MBR ← MEM[MAR]; C ← 1	
20 / 21 / 22		0xBE7	0	1	00	0000	0	0	0	0	1	01011	11111	11111	0000 0000 0000	EARL ← MBR (C still 1)	
		0xBE8	0	0	00	1111	0	1	1	0	0	11111	01010	01011	0000 0000 0000	MARH ← EARH; MARL ← EARL	
		0xBE9	0	0	00	0111	1	0	1	0	0	11111	11111	11111	0000 0000 0000	MBR ← MEM[MAR]; C ← 1	
		0xBEA	0	1	00	0000	0	0	0	0	1	01100	11111	11111	0000 0000 0000	OPRH ← MBR (C still 1)	load HI-byte to OPRH
		0xBEB	0	0	00	0010	0	0	0	0	1	01011	01011	11111	0000 0000 0000	EARL ← EARL + 0 + C	
		0xBEC	0	0	00	0010	0	0	0	0	1	01010	01010	11111	0000 0000 0000	EARH ← EARH + 0 + C	
		0xBED	0	0	11	1111	0	1	1	0	0	11111	01010	01011	1000 1110 0110	MARH ← EARH; MARL ← EARL; jump to 0x8E6	continue within LDX Imm. (0x8E0)
		0xBEE - 0xBEF	X	X	XX	XXXX	X	X	X	X	X	XXXXX	XXXXX	XXXXX	XXXX XXXX XXXX		
STX	Abs.	0xBF0	0	0	00	1111	0	1	1	0	0	11111	00110	00111	0000 0000 0000	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xBF1	0	0	00	0111	1	0	1	0	0	11111	11111	11111	0000 0000 0000	MBR ← MEM[MAR]; C ← 1	
		0xBF2	0	1	00	0000	0	0	0	0	1	01010	11111	11111	0000 0000 0000	EARH ← MBR (C still 1)	
		0xBF3	0	0	00	0010	0	0	0	0	1	00111	00111	11111	0000 0000 0000	PCL ← PCL + 0 + C	
20 / 21		0xBF4	0	0	00	0010	0	0	0	0	1	00110	00110	11111	0000 0000 0000	PCH ← PCH + 0 + C	
		0xBF5	0	0	00	1111	0	1	1	0	0	11111	00110	00111	0000 0000 0000	MARH ← PCH; MARL ← PCL	
		0xBF6	0	0	00	1111	1	0	1	0	0	11111	11111	11111	0000 0000 0000	MBR ← MEM[MAR]	
		0xBF7	0	1	11	0000	0	0	0	0	1	01011	11111	11111	1000 1111 0000	EARL ← MBR; jump to 0x8F0	continue at codeend of STX
		0xBF8 - 0xBFF	X	X	XX	XXXX	X	X	X	X	X	XXXXX	XXXXX	XXXXX	XXXX XXXX XXXX		

Instruction	Type	MPC-Addr.	OP	Amux	Cond	ALU	MBR	MAR	RD	WR	ENC	C	B	A	Address	Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																	
LDY	Imm.	0xC00	0	0	0 0	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xC01	0	0	0 0	0 1 1 1	1	0	1	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MBR ←MEM[MAR]; C ← 1	
		0xC02	0	1	0 0	0 0 0 0	0	0	0	0	1	0 1 1 0 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	OPRH ← MBR (C still 1)	load HI-byte to OPRH
		0xC03	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 1 1	0 0 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	PCL ← PCL + 0 + C	
		0xC04	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 1 0	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	PCH ← PCH + 0 + C	
		0xC05	0	0	0 0	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MARH ← PCH; MARL ← PCL	
		0xC06	0	0	0 0	0 1 1 1	1	0	1	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MBR ←MEM[MAR]; C ← 1	
		0xC07	0	1	0 0	0 0 0 0	0	0	0	0	1	0 0 0 0 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	YRL ← MBR (C still 1)	save LO-byte directly to final destination YRL
12 / 13 / 14		0xC08	0	0	0 0	0 0 0 0	0	0	0	0	1	0 0 0 0 0	1 1 1 1 1	0 1 1 0 0	0 0 0 0 0 0 0 0 0 0 0 0	YRH ← OPRH (V ← 0, C still 1))	
		0xC09	0	0	1 0	1 0 0 1	0	0	0	0	1	0 0 1 0 0	1 1 1 1 1	0 1 0 0 1	1 1 0 0 0 0 0 0 1 1 0 0	CC ← (N, Z, V) ° CC; if Z=1 jump to 0xC0C (C still 1)	now save HI-byte to final destination YRH (sets N-flag correctly); then the status flags are updated, if Z=0 the flags are already correct, otherwise also the LO-byte has to be checked if it is zero and only the Z-flag is updated again if it has to be changed to zero
		0xC0A	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 0 1 1	0 0 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	PCL ← PCL + 0 + C	
		0xC0B	0	0	1 1	0 0 1 0	0	0	0	0	1	0 0 1 1 0	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 1	PCH ← PCH + 0 + C; jump to 0x001	
		0xC0C	0	0	1 0	0 0 0 0	0	0	0	0	0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 1	1 1 0 0 0 0 0 0 1 0 1 0	ALUout ← YRL; if Z=1 jump to 0xC0A (C still 1)	
		0xC0D	0	0	1 1	1 0 1 1	0	0	0	0	1	0 1 0 0 1	1 1 1 1 1	0 1 0 0 1	1 1 0 0 0 0 0 0 1 0 1 0	CC ← (Z) ° CC; jump to 0xC0A (C still 1)	
		0xC0E - 0xC0F	X	X	X X	X X X X	X	X	X	X	X	X X X X X	X X X X X	X X X X X	X X X X X X X X X X X X		
LDS	Imm.	0xC10	0	0	0 0	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xC11	0	0	0 0	0 1 1 1	1	0	1	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MBR ←MEM[MAR]; C ← 1	
		0xC12	0	1	0 0	0 0 0 0	0	0	0	0	1	0 1 1 0 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	OPRH ← MBR (C still 1)	load HI-byte to OPRH
		0xC13	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 1 1	0 0 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	PCL ← PCL + 0 + C	
		0xC14	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 1 0	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	PCH ← PCH + 0 + C	
		0xC15	0	0	0 0	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MARH ← PCH; MARL ← PCL	
		0xC16	0	0	0 0	0 1 1 1	1	0	1	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MBR ←MEM[MAR]; C ← 1	
12 / 13 / 14		0xC17	0	1	0 0	0 0 0 0	0	0	0	0	1	0 0 1 0 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	SPL ← MBR (C still 1)	save loaded LO-byte directly to final destination SPL
		0xC18	0	0	0 0	0 0 0 0	0	0	0	0	1	0 0 0 1 0	1 1 1 1 1	0 1 1 0 0	0 0 0 0 0 0 0 0 0 0 0 0	SPH ← OPRH (V ← 0, C still 1)	
		0xC19	0	0	1 0	1 0 0 1	0	0	0	0	1	0 1 1 0 0	1 1 1 1 1	0 1 0 0 1	1 1 0 0 0 0 0 1 1 1 0 0	CC ← (N, Z, V) ° CC; if Z=1 jump to 0xC1C (C still 1)	now save HI-byte to final destination SPH (sets N-flag correctly); then the status flags are updated, if Z=0 the flags are already correct, otherwise also the LO-byte has to be checked if it is zero and only the Z-flag is updated again if it has to be changed to zero
		0xC1A	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 0 1 1	0 0 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	PCL ← PCL + 0 + C	
		0xC1B	0	0	1 1	0 0 1 0	0	0	0	0	1	0 0 1 1 0	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 1	PCH ← PCH + 0 + C; jump to 0x001	
		0xC1C	0	0	1 0	0 0 0 0	0	0	0	0	0	1 1 1 1 1	1 1 1 1 1	0 0 1 0 1	1 1 0 0 0 0 0 1 1 0 1 0	ALUout ← SPL; if Z=1 jump to 0xC1A (C still 1)	
		0xC1D	0	0	1 1	1 0 1 1	0	0	0	0	1	0 1 1 0 0	1 1 1 1 1	0 1 0 0 1	1 1 0 0 0 0 0 1 1 0 1 0	CC ← (Z) ° CC; jump to 0xC1A (C still 1)	
		0xC1E - 0xC1F	X	X	X X	X X X X	X	X	X	X	X	X X X X X	X X X X X	X X X X X	X X X X X X X X X X X X		
Codeend of STY		0xC20	0	0	0 0	1 1 1 1	0	1	0	0	0	1 1 1 1 1	0 1 0 1 0	0 1 0 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MARH ← EARH; MARL ← EARL	
		0xC21	0	0	0 0	0 0 0 0	0	1	0	0	1	0 1 1 1 1	1 1 1 1 1	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0	MBR ← YRH (V ← 0)	save HI-byte of Y-reg (YRH) to the destination address and update flags (N and V are already set correctly, but Z might be wrong for complete Y-reg)
		0xC22	0	0	0 0	0 1 1 1	0	0	0	1	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	M[MAR] ← MBR; C ← 1 (other flags unaffected)	
		0xC23	0	0	0 0	1 0 0 1	0	0	0	0	1	0 1 0 0 1	1 1 1 1 1	0 1 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0	CC ← (N, Z, V) ° CC (C still 1)	
		0xC24	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 1 0 1	1 0 1 1 1	0 1 0 1 1	1 1 1 1 1 0 0 0 0 0 0 0	EARL ← EARL + 0 + C	increase the effective address-register to also store the LO-byte at the next address in the memory
		0xC25	0	0	0 0	0 0 1 0	0	0	0	0	1	0 1 0 1 0	0 1 0 1 0	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	EARH ← EARH + 0 + C	
		0xC26	0	0	0 0	1 1 1 1	0	1	0	0	0	1 1 1 1 1	0 1 0 1 0	0 1 0 1 1	0 0 0 0 0 0 0 0 0 0 0 0	MARH ← EARH; MARL ← EARL	
		0xC27	0	0	0 0	0 0 0 0	1	0	0	1	0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0	MBR ← YRL	
		0xC28	0	0	1 0	0 1 1 1	0	0	0	1	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	1 1 0 0 0 0 1 0 1 1 0 0	M[MAR] ← MBR; C ← 1; if Z=1 jump to 0xC2C	
		0xC29	0	0	0 0	1 0 1 1	0	0	0	0	1	0 1 1 0 0	1 1 1 1 1	0 1 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0	CC ← (Z) ° CC (C still 1)	save now LO-byte of Y-reg (YRL); if Z=0 no jump is performed and the Z-flag is updated accordingly; if Z=1 (YRL is zero) also YRH has to be checked if it is zero before updating the Z-flag
		0xC2A	0	0	0 0	0 0 1 0	0	0	0	0	1	0 0 0 1 1	0 0 1 1 1	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0	PCL ← PCL + 0 + C	
		0xC2B	0	0	1 1	0 0 1 0	0	0	0	0	1	0 0 1 1 0	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 1	PCH ← PCH + 0 + C; jump to 0x001	
		0xC2C	0	0	1 1	0 0 0 0	0	0	0	0	0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	1 1 0 0 0 0 1 0 1 0 0 1	ALUout ← YRH; jump to 0xC29 (C still 1)	
		0xC2D - 0xC2F	X	X	X X	X X X X	X	X	X	X	X	X X X X X	X X X X X	X X X X X	X X X X X X X X X X X X		

Instruction Type	MPC-Addr.	OP	Amux	Cond	ALU	MBR	MAR	RD	WR	ENC	C	B	A	Address	Instruction Mnemonic	(Micro) Instruction Comments
Codeend of STS	0xC30	0	0	0 0 1	1 1 1	0	1	0	0	0	1 1 1	0 1 0	0 1 1	0 0 0 0 0 0	MARH ← EARH; MARL ← EARL	
	0xC31	0	0	0 0 0	0 0 0	1	0	0	1	0	1 1 1	1 1 1	0 0 1	0 0 0 0 0 0	MBR ← SPH (V ← 0)	save HI-byte of SP-reg (SPH) to the destination address and update flags (N and V are already set correctly, but Z might be wrong for complete SP-reg)
	0xC32	0	0	0 0 0	0 1 1	0	0	0	1	0	1 1 1	1 1 1	1 1 1	0 0 0 0 0 0	M[MAR] ← MBR; C ← 1 (other flags unaffected)	
	0xC33	0	0	0 0 1	0 1 0	0	0	0	0	1	0 1 0	1 1 1	0 1 0	0 0 0 0 0 0	CC ← (N, Z, V) * CC (C still 1)	
	0xC34	0	0	0 0 0	0 1 0	0	0	0	0	1	0 1 0	1 1 1	1 1 1	0 0 0 0 0 0	EARL ← EARL + 0 + C	Increase the effective address-register to also store the LO-byte at the next address in the memory
	0xC35	0	0	0 0 0	0 1 0	0	0	0	0	1	0 1 0	0 1 0	1 1 1	0 0 0 0 0 0	EARH ← EARH + 0 + C	
	0xC36	0	0	0 0 0	1 1 1	1	0	0	0	0	1 1 1	0 1 0	1 1 1	0 0 0 0 0 0	MARH ← EARH; MARL ← EARL	
	0xC37	0	0	0 0 0	0 0 0	0	1	0	0	1	1 1 1	1 1 1	0 0 1	0 0 0 0 0 0	MBR ← SPL	
	0xC38	0	0	1 0 0	0 1 1	1	0	0	0	1	1 1 1	1 1 1	1 1 1	1 1 0 0 0 1	M[MAR] ← MBR; C ← 1; if Z=1 jump to 0xC3C	save new LO-byte of SP-reg (SPL); if Z=0 no jump is performed and the Z-flag is updated accordingly; if Z=1 (SPL is zero) also SPH has to be checked if it is zero before updating the Z-flag
	0xC39	0	0	0 0 0	1 0 1	1	0	0	0	1	0 1 0	1 1 1	0 1 0	0 0 0 0 0 0	CC ← (Z) * CC (C still 1)	
	0xC3A	0	0	0 0 0	0 0 1	0	0	0	0	1	1 1 0	0 1 0	1 1 1	0 0 0 0 0 0	PCL ← PCL + 0 + C	
	0xC3B	0	0	1 1 0	0 0 1	0	0	0	0	1	0 0 0	0 1 1	1 1 1	0 0 0 0 0 1	PCH ← PCH + 0 + C; jump to 0x001	
	0xC3C	0	0	1 1 0	0 0 0	0	0	0	0	0	1 1 1	1 1 1	0 1 0	0 1 1 1 1 0	ALUout ← SPH; jump to 0xC39 (C still 1)	
	0xC3D - 0xC3F	X	X	X X X	X X X	X	X	X	X	X	X X X	X X X	X X X	X X X X X X		
13 / 14 / 15	0xC40	0	0	0 1 1	1 1 1	0	1	1	0	0	1 1 1	0 0 1	1 1 1	0 0 0 0 0 0	MARH ← PCH; MARL ← PCL	
ADDX (mm. N, Z, V, C)	0xC41	0	0	0 0 0	0 1 1	1	0	1	1	0	0 1 1	1 1 1	1 1 1	0 0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
	0xC42	0	1	0 0 0	0 0 0	0	0	0	0	1	0 1 1	0 0 0	1 1 1	0 0 0 0 0 0	OPRH ← MBR (C still 1)	load HI-byte (addend) to OPRH
	0xC43	0	0	0 0 0	0 0 1	0	0	0	0	1	0 0 0	1 1 1	1 1 1	0 0 0 0 0 0	PCL ← PCL + 0 + C	
	0xC44	0	0	0 0 0	0 0 1	0	0	0	0	1	0 0 0	1 1 1	1 1 1	0 0 0 0 0 0	PCH ← PCH + 0 + C	
	0xC45	0	0	0 0 1	1 1 1	1	0	1	1	0	0 1 1	1 1 1	0 0 1	0 0 0 0 0 0	MARH ← PCH; MARL ← PCL	
	0xC46	0	0	0 0 0	0 1 1	0	1	0	0	0	1 1 1	1 1 1	1 1 1	0 0 0 0 0 0	MBR ← MEM[MAR]; C ← 0	load LO-byte (addend) and add it with the LO-byte of X-reg (without carry) and save directly to final destination XRL
	0xC47	0	1	0 0 0	0 0 1	0	0	0	0	1	0 0 0	1 1 1	1 1 1	0 0 0 0 0 0	XRL ← XRL + MBR + C	
	0xC48	0	0	0 0 0	0 1 0	0	0	0	0	1	0 0 0	0 1 0	0 1 1	1 0 0 0 0 0	XRH ← XRH + OPRH + C	
	0xC49	0	0	1 0 1	0 0 0	0	0	0	0	1	0 1 0	0 1 1	0 1 0	1 1 0 0 1 1	CC ← (N, Z, V, C) * CC; if Z=1 jump to 0xC4D	now add the HI-byte which was loaded from the memory (addend) to the HI-byte of X-reg (sets N-, V- and C-flag correctly); then the status flags are updated, if Z=0 the flags are already correct, otherwise the LO-byte of X-reg has to be checked if it is zero too and only the Z-flag is updated again if it has to be changed to zero
	0xC4A	0	0	0 0 0	0 1 1	1	0	0	0	0	1 1 1	1 1 1	1 1 1	0 0 0 0 0 0	C ← 1	
	0xC4B	0	0	0 0 0	0 0 1	0	0	0	0	0	1 1 1	1 1 0	0 1 1	1 1 1	0 0 0 0 0 0	PCL ← PCL + 0 + C
	0xC4C	0	0	1 1 0	0 0 1	0	0	0	0	1	0 0 0	1 1 0	0 0 0	1 1 1	1 1 0 0 0 0	PCH ← PCH + 0 + C; jump to 0x001
	0xC4D	0	0	1 0 0	0 0 0	0	0	0	0	0	1 1 1	1 1 1	1 1 0	0 0 0 1 1 1	1 0 0 1 0 0	ALUout ← XRL; if Z=1 jump to 0xC4A
	0xC4E	0	0	1 1 1	0 1 1	0	0	0	0	0	1 0 0	0 1 1	1 1 1	1 0 0 0 1 1	1 0 0 1 0 0	CC ← (Z) * CC; jump to 0xC4A
	0xC4F	X	X	X X X	X X X	X	X	X	X	X	X X X	X X X	X X X	X X X X X X		
13 / 14 / 15	0xC50	0	0	0 1 1	1 1 1	0	1	1	0	0	1 1 1	0 0 0	1 1 1	0 0 0 0 0 0	MARH ← PCH; MARL ← PCL	
ADDY (mm. N, Z, V, C)	0xC51	0	0	0 0 0	0 1 1	1	0	1	1	0	0 1 1	1 1 1	1 1 1	0 0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
	0xC52	0	1	0 0 0	0 0 0	0	0	0	0	1	0 0 0	1 1 0	0 1 1	1 1 1	0 0 0 0 0 0	OPRH ← MBR (C still 1)
	0xC53	0	0	0 0 0	0 1 0	0	0	0	0	1	0 0 0	1 1 0	0 1 1	1 1 1	0 0 0 0 0 0	PCL ← PCL + 0 + C
	0xC54	0	0	0 0 0	0 1 0	0	0	0	0	1	0 0 0	1 1 0	0 1 1	1 1 1	0 0 0 0 0 0	PCH ← PCH + 0 + C
	0xC55	0	0	0 0 1	1 1 1	0	1	1	0	0	1 1 1	0 0 0	1 1 1	0 0 0 0 0 0	MARH ← PCH; MARL ← PCL	
	0xC56	0	0	0 0 0	0 1 1	0	1	1	0	0	1 1 1	1 1 0	0 1 1	1 1 1	0 0 0 0 0 0	MBR ← MEM[MAR]; C ← 0
	0xC57	0	1	0 0 0	0 0 1	0	0	0	0	1	0 0 0	0 0 1	1 1 1	1 1 1	0 0 0 0 0 0	YRL ← YRL + MBR + C
	0xC58	0	0	0 0 0	0 1 0	0	0	0	0	1	0 0 0	0 0 0	0 1 1	0 0 0 0 0 0	YRH ← YRH + OPRH + C	
	0xC59	0	0	1 0 1	0 0 0	0	0	0	0	1	0 1 0	0 1 1	1 1 1	0 0 0 1 1 1	CC ← (N, Z, V, C) * CC; if Z=1 jump to 0xC5D	now add the HI-byte which was loaded from the memory (addend) to the HI-byte of Y-reg (sets N-, V- and C-flag correctly); then the status flags are updated, if Z=0 the flags are already correct, otherwise the LO-byte of Y-reg has to be checked if it is zero too and only the Z-flag is updated again if it has to be changed to zero
	0xC5A	0	0	0 0 0	1 1 1	0	0	0	0	1	0 1 1	1 1 1	1 1 1	0 0 0 0 0 0	C ← 1	
	0xC5B	0	0	0 0 0	0 1 0	0	0	0	0	1	0 0 0	1 1 1	1 1 1	0 0 0 0 0 0	PCL ← PCL + 0 + C	
	0xC5C	0	0	1 1 0	0 1 0	0	0	0	0	1	0 0 0	1 1 0	0 1 1	1 1 1	0 0 0 0 0 0	PCH ← PCH + 0 + C; jump to 0x001
	0xC5D	0	0	1 0 0	0 0 0	0	0	0	0	0	0 1 1	1 1 1	1 1 1	0 0 0 0 1 1	ALUout ← YRL; if Z=1 jump to 0xC5A	
	0xC5E	0	0	1 1 1	0 1 1	0	0	0	0	1	0 0 0	0 1 1	1 1 1	1 0 0 0 1 1	1 0 0 1 0 0	CC ← (Z) * CC; jump to 0xC5A

Instruction / Type CC changes, No. of instructions to execute	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A					Address									Instruction Mnemonic	(Micro) Instruction Comments
	0xC5F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
ADDS Imm.	0xC60	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C	0xC61	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
	0xC62	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRH ← MBR (C still 1)	load HI-byte (addend) to OPRH
	0xC63	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
	0xC64	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
	0xC65	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
	0xC66	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	load LO-byte (addend) and add it with the LO-byte of SP-reg
	0xC67	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + MBR + C	(without carry) and save directly to final destination SPL
13 / 14 / 15	0xC68	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + OPRH + C	
	0xC69	0	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	1	1	0	0	0	1	1	0	1	1	CC ← (N, Z, V, C) ° CC; if Z=1 jump to 0xC6D	now add the HI-byte which was loaded from the memory (addend)
	0xC6A	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	C ← 1	to the HI-byte of SP-reg (sets N-, V- and C-flag correctly); then the
	0xC6B	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	status flags are updated, if Z=0 the flags are already correct,
	0xC6C	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	otherwise the LO-byte of SP-reg has to be checked if it is zero too
	0xC6D	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	1	1	0	0	0	1	1	0	1	0	ALUout ← SPL; if Z=1 jump to 0xC6A	and only the Z-flag is updated again if it has to be changed to zero
	0xC6E	0	0	1	1	1	0	1	1	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	1	1	0	0	0	1	1	0	1	0	CC ← (Z) ° CC; jump to 0xC6A	
	0xC6F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
SUBX Imm.	0xC70	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C	0xC71	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
	0xC72	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRH ← NOT(MBR), (C ← 1)	load negated HI-byte (subtrahend) to OPRH
	0xC73	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
	0xC74	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
	0xC75	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
	0xC76	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	load negated LO-byte (subtrahend) and subtract it from the LO-
	0xC77	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRL ← NOT(MBR) (C ← 1)	byte of X-reg using addition with 2's complement of the
	0xC78	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	XRL ← XRL + OPRL + C	subtrahend and save directly to final destination XRL
14 / 15 / 16	0xC79	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	XRH ← XRH + OPRH + C	
	0xC7A	0	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	1	1	0	0	0	1	1	1	1	0	CC ← (N, Z, V, C) ° CC; if Z=1 jump to 0xC7E	now subtract the HI-byte which was loaded from the memory
	0xC7B	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	C ← 1	(subtrahend) using 2's complement addition (add with carry from
	0xC7C	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	LO-byte addition) from the HI-byte of X-reg (sets N-, V- and C-flag
	0xC7D	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	correctly); then the status flags are updated, if Z=0 the flags are
	0xC7E	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1	1	0	1	1	0	1	ALUout ← XRL; if Z=1 jump to 0xC7B	already correct, otherwise the LO-byte of X-reg has to be checked if
	0xC7F	0	0	1	1	1	0	1	1	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	0	1	1	1	1	1	CC ← (Z) ° CC; jump to 0xC7B	it is zero too and only the Z-flag is updated again if it has to be changed to zero

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B				A				Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																								
SUBY	Imm.	0xC80	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xC81	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xC82	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xC83	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xC84	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
		0xC85	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0xC86	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	load negated LO-byte (subtrahend) and subtract it from the LO-byte of Y-reg using addition with 2's complement of the subtrahend and save directly to final destination YRL
		0xC87	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRL ← NOT(MBR) (C ← 1)	
		0xC88	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	YRL ← YRL + OPRL + C	
14 / 15 / 16		0xC89	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	YRH ← YRH + OPRH + C	now subtract the HI-byte which was loaded from the memory (subtrahend) using 2's complement addition (add with carry from LO-byte addition) from the HI-byte of Y-reg (sets N-, V- and C-flag correctly); then the status flags are updated, if Z=0 the flags are already correct, otherwise the LO-byte of Y-reg has to be checked if it is zero too and only the Z-flag is updated again if it has to be changed to zero
		0xC8A	0	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	1	1	0	1	0	0	1	1	0	0	0	1	1	1	0	CC ← (N, Z, V, C) ° CC; if Z=1 jump to 0xC8E	
		0xC8B	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	C ← 1	
		0xC8C	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xC8D	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0xC8E	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	0	0	1	0	1	1	1	1	ALUout ← YRL; if Z=1 jump to 0xC8B	
		0xC8F	0	0	1	1	1	0	1	1	0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	0	1	0	0	0	1	1	CC ← (Z) ° CC; jump to 0xC8B	
SUBS	Imm.	0xC90	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xC91	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xC92	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xC93	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xC94	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
		0xC95	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0xC96	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	load negated LO-byte (subtrahend) and subtract it from the LO-byte of SP-reg using addition with 2's complement of the subtrahend and save directly to final destination SPL
		0xC97	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRL ← NOT(MBR) (C ← 1)	
		0xC98	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	SPL ← SPL + OPRL + C	
14 / 15 / 16		0xC99	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	SPH ← SPH + OPRH + C	now subtract the HI-byte which was loaded from the memory (subtrahend) using 2's complement addition (add with carry from LO-byte addition) from the HI-byte of SP-reg (sets N-, V- and C-flag correctly); then the status flags are updated, if Z=0 the flags are already correct, otherwise the LO-byte of SP-reg has to be checked if it is zero too and only the Z-flag is updated again if it has to be changed to zero
		0xC9A	0	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	0	1	0	0	0	1	0	CC ← (N, Z, V, C) ° CC; if Z=1 jump to 0xC9E	
		0xC9B	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	C ← 1	
		0xC9C	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xC9D	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0xC9E	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	1	0	1	1	1	0	0	0	1	1	0	0	1	1	ALUout ← SPL; if Z=1 jump to 0xC9B	
		0xC9F	0	0	1	1	1	0	1	1	0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	0	1	1	0	0	1	1	CC ← (Z) ° CC; jump to 0xC9B	
CMPY	Imm.	0xCA0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xCA1	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xCA2	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xCA3	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xCA4	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
		0xCA5	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0xCA6	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	load negated LO-byte (subtrahend) and subtract it from the LO-byte of Y-reg using addition with 2's complement of the subtrahend and save directly to temp U-reg
		0xCA7	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRL ← NOT(MBR) (C ← 1)	
		0xCA8	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	U ← YRL + OPRL + C	
14 / 15 / 16		0xCA9	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	ALUout ← YRH + OPRH + C	now subtract the HI-byte which was loaded from the memory (subtrahend) using 2's complement addition (add with carry from LO-byte addition) from the HI-byte of Y-reg (sets N-, V- and C-flag correctly); then the status flags are updated, if Z=0 the flags are already correct, otherwise the LO-byte of the subtraction (stored in U) has to be checked if it is zero too and only the Z-flag is updated again if it has to be changed to zero
		0xCAA	0	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1	0	1	1	1	0	CC ← (N, Z, V, C) ° CC; if Z=1 jump to 0xCAE	
		0xCAB	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	C ← 1	
		0xCAC	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xCAD	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0xCAE	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	0	1	0	1	0	1	1	ALUout ← U; if Z=1 jump to 0xCAB	
		0xCAF	0	0	1	1	1	0	1	1	0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	0	1	1	0	0	1	1	CC ← (Z) ° CC; jump to 0xCAB	

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B				A				Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																								
CMPS	Imm.	0xCB0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xCB1	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xCB2	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xCB3	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xCB4	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
		0xCB5	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0xCB6	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	load negated LO-byte (subtrahend) and subtract it from the LO-byte of SP-reg using addition with 2's complement of the subtrahend and save directly to temp U-reg
		0xCB7	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRL ← NOT(MBR) (C ← 1)	
14 / 15 / 16		0xCB8	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0	0	0	0	U ← SPL + OPRL + C	
		0xCB9	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	1	1	1	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	ALUout ← SPH + OPRH + C	now subtract the HI-byte which was loaded from the memory (subtrahend) using 2's complement addition (add with carry from LO-byte addition) from the HI-byte of SP-reg (sets N-, V- and C-flag correctly); then the status flags are updated, if Z=0 the flags are already correct, otherwise the LO-byte of the subtraction (stored in U) has to be checked if it is zero too and only the Z-flag is updated again if it has to be changed to zero
		0xCBA	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	1	0	CC ← (N, Z, V, C) ° CC; if Z=1 jump to 0xCBE	
		0xCBB	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	C ← 1	
		0xCBC	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xCBD	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	1	PCH ← PCH + 0 + C; jump to 0x001	
		0xCBE	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	1	1	1	0	1	ALUout ← U; if Z=1 jump to 0xCBB	
		0xCBF	0	0	1	1	1	0	1	1	0	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	1	1	0	CC ← (Z) ° CC; jump to 0xCBB	
		0xCC0 - 0xCFF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
LDY	@y	0xD00	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xD01	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xD02	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0xD03	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	1	0	1	ALUout ← MBR; if N=1 jump to 0xD05
		0xD04	0	0	1	1	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1	0	0	0	0	1	1	0	EARH ← YRH + 0 + C; jump to 0xD06
		0xD05	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
18 / 19 / 20		0xD06	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
		0xD07	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xD08	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	OPRH ← MBR (C still 1)	load HI-byte to OPRH
		0xD09	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xD0A	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xD0B	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	0	0	0	0	1	1	0	MARH ← EARH; MARL ← EARL; jump to 0xC06	continue within LDY Imm. (0xC00)
		0xD0C - 0xD0F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
LDS	@y	0xD10	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xD11	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xD12	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0xD13	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	1	0	1	0	1	ALUout ← MBR; if N=1 jump to 0xD15
		0xD14	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1	0	0	0	1	0	1	1	0	EARH ← YRH + 0 + C; jump to 0xD16
		0xD15	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
		0xD16	0	0	0	0	1	1	1	0	1	1	0	0	1	1	1	1	1	1	0	1	0	1	0	1	1	0	1	1	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
18 / 19 / 20		0xD17	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xD18	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	OPRH ← MBR (C still 1)	load HI-byte to OPRH
		0xD19	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xD1A	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xD1B	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	0	0	0	0	1	1	0	MARH ← EARH; MARL ← EARL; jump to 0xC16	continue within LDS Imm. (0xC10)
		0xD1C - 0xD1F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A					Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																										
STY	@y	0xD20	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xD21	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xD22	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
17 / 18		0xD23	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	0	1	1	ALUout ← MBR; if N=1 jump to 0xD25	
		0xD24	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	1	0	0	0	0	EARH ← YRH + 0 + C; jump to 0xC20	continue at codeend of STY
		0xD25	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	1	0	0	0	1	0	0	0	0	EARH ← YRH + 0xFF + C; jump to 0xC20	
		0xD26 - 0xD2F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
STS	@y	0xD30	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xD31	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xD32	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
17 / 18		0xD33	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	1	0	1	ALUout ← MBR; if N=1 jump to 0xD35	
		0xD34	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	1	1	0	0	0	EARH ← YRH + 0 + C; jump to 0xC30	continue at codeend of STS
		0xD35	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	1	0	0	0	1	1	0	0	0	EARH ← YRH + 0xFF + C; jump to 0xC30	
		0xD36 - 0xD3F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ADDX	@y	0xD40	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xD41	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xD42	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0xD43	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	0	1	ALUout ← MBR; if N=1 jump to 0xD45	
		0xD44	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1	0	1	0	0	1	1	EARH ← YRH + 0 + C; jump to 0xD46	
		0xD45	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
		0xD46	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
19 / 20 / 21		0xD47	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xD48	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRH ← MBR (C still 1)	load HI-byte (addend) to OPRH
		0xD49	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xD4A	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xD4B	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	0	0	1	0	0	0	0	MARH ← EARH; MARL ← EARL; jump to 0xC46	continue within ADDX Imm. (0xC40)
		0xD4C - 0xD4F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ADDY	@y	0xD50	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xD51	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xD52	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0xD53	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	0	1	ALUout ← MBR; if N=1 jump to 0xD55	
		0xD54	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	1	0	1	1	0	EARH ← YRH + 0 + C; jump to 0xD56	
		0xD55	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
		0xD56	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
19 / 20 / 21		0xD57	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xD58	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRH ← MBR (C still 1)	load HI-byte (addend) to OPRH
		0xD59	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xD5A	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xD5B	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	0	0	1	0	1	1	0	MARH ← EARH; MARL ← EARL; jump to 0xC56	continue within ADDY Imm. (0xC50)
		0xD5C - 0xD5F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B				A				Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																								
ADDS	@y	0xD60	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xD61	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xD62	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0xD63	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	0	0	1	ALUout ← MBR; if N=1 jump to 0xD65	
		0xD64	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	0	1	0	1	1	0	EARH ← YRH + 0 + C; jump to 0xD66	
		0xD65	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
		0xD66	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
19 / 20 / 21		0xD67	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xD68	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	OPRH ← MBR (C still 1)	load HI-byte (addend) to OPRH
		0xD69	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xD6A	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xD6B	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	0	0	1	1	0	MARH ← EARH; MARL ← EARL; jump to 0xC66	continue within ADDS Imm. (0xC60)
		0xD6C - 0xD6F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
SUBX	@y	0xD70	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xD71	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xD72	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0xD73	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	0	1	ALUout ← MBR; if N=1 jump to 0xD75	
		0xD74	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	0	1	0	1	1	0	EARH ← YRH + 0 + C; jump to 0xD76	
		0xD75	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
		0xD76	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
20 / 21 / 22		0xD77	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xD78	0	1	0	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xD79	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xD7A	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xD7B	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	0	0	1	1	0	MARH ← EARH; MARL ← EARL; jump to 0xC76	continue within SUBX Imm. (0xC70)
		0xD7C - 0xD7F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
SUBY	@y	0xD80	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xD81	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xD82	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xD83	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	1	0	ALUout ← MBR; if N=1 jump to 0xD85	
		0xD84	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	1	1	EARH ← YRH + 0 + C; jump to 0xD86	
		0xD85	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
		0xD86	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
20 / 21 / 22		0xD87	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xD88	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xD89	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xD8A	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xD8B	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	0	0	1	1	0	MARH ← EARH; MARL ← EARL; jump to 0xC86	continue within SUBY Imm. (0xC80)
		0xD8C - 0xD8F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B				A				Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																								
SUBS	@y	0xD90	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xD91	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xD92	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0xD93	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	ALUout ← MBR; if N=1 jump to 0xD95	
		0xD94	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	EARH ← YRH + 0 + C; jump to 0xD96	
		0xD95	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
		0xD96	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
	20 / 21 / 22	0xD97	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xD98	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	OPRH ← NOT(MBR) (c ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xD99	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xD9A	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xD9B	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	MARH ← EARH; MARL ← EARL; jump to 0xC96	continue within SUBS Imm. (0xC90)
		0xD9C - 0xD9F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
CMPY	@y	0xDA0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xDA1	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xDA2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0xDA3	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	ALUout ← MBR; if N=1 jump to 0xDA5	
		0xDA4	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	EARH ← YRH + 0 + C; jump to 0xDA6	
		0xDA5	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
	20 / 21 / 22	0xDA6	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
		0xDA7	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xDA8	0	1	0	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRH ← NOT(MBR) (c ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xDA9	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xDAA	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xDAB	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	MARH ← EARH; MARL ← EARL; jump to 0xCA6	continue within CMPY Imm. (0xCA0)
		0xDAC - 0xDAF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
CMPS	@y	0xDB0	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xDB1	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xDB2	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	EARL ← YRL + MBR + C	
		0xDB3	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**1**	ALUout ← MBR; if N=1 jump to 0xDB5	
		0xDB4	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**0**	EARH ← YRH + 0 + C; jump to 0xDB6	
		0xDB5	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← YRH + 0xFF + C	
		0xDB6	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
	20 / 21 / 22	0xDB7	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xDB8	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRH ← NOT(MBR) (c ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xDB9	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xDBA	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xDBB	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**0**	MARH ← EARH; MARL ← EARL; jump to 0xCB6	continue within CMPS Imm. (0xCB0)
		0xDBC - 0xDFF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type (CC changes, No. of instructions to execute)	MPC-Addr.	OP	Amux	Cond	ALU	MBR	MAR	RD	WR	ENC	C	B	A	Address	Instruction Mnemonic	(Micro) Instruction Comments
LDY	@x	0xE00	0	0	0 0	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 0 1 1 0	0 0 1 1 1	0000 0000 0000	MARH ← PCH; MARL ← PCL	
	N, Z, V (0)	0xE01	0	0	0 0	0 1 1 0	1	0	1	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0000 0000 0000	MBR ← MEM[MAR]; C ← 0	
		0xE02	0	1	0 0	0 0 1 0	0	0	0	0	1	0 1 0 1 1	0 0 0 1 1	1 1 1 1 1	0000 0000 0000	EARL ← XRL + MBR + C	
		0xE03	0	1	0 1	0 0 0 0	0	0	0	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	1110 0000 0101	ALUout ← MBR; if N=1 jump to 0xE05	
		0xE04	0	0	1 1	0 0 1 0	0	0	0	0	1	0 1 0 1 0	0 0 0 1 0	1 1 1 1 1	1110 0000 0110	EARH ← XRH + 0 + C; jump to 0xE06	
		0xE05	0	0	0 0	0 0 1 0	0	0	0	0	1	0 1 0 1 0	0 0 0 1 0	1 1 0 0 1	0000 0000 0000	EARH ← XRH + 0xFF + C	
	18 / 19 / 20	0xE06	0	0	0 0	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 1 0 1 0	0 1 0 1 1	0000 0000 0000	MARH ← EARH; MARL ← EARL	
		0xE07	0	0	0 0	0 1 1 1	1	0	1	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0000 0000 0000	MBR ← MEM[MAR]; C ← 1	
		0xE08	0	1	0 0	0 0 0 0	0	0	0	0	1	0 1 1 0 0	1 1 1 1 1	1 1 1 1 1	0000 0000 0000	OPRH ← MBR (C still 1)	load HI-byte to OPRH
		0xE09	0	0	0 0	0 0 1 0	0	0	0	0	1	0 1 0 1 1	0 1 0 1 1	1 1 1 1 1	0000 0000 0000	EARL ← EARL + 0 + C	
		0xE0A	0	0	0 0	0 0 1 0	0	0	0	0	1	0 1 0 1 0	0 1 0 1 0	1 1 1 1 1	0000 0000 0000	EARH ← EARH + 0 + C	
		0xE0B	0	0	1 1	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 1 0 1 0	0 1 0 1 1	1100 0000 0110	MARH ← EARH; MARL ← EARL; jump to 0xC06	continue within LDY Imm. (0xC00)
		0xE0C - 0xE0F	X	X	X X	X X X X	X	X	X	X	X	X X X X X	X X X X X	X X X X X	XXXX XXXX XXXX		
LDS	@x	0xE10	0	0	0 0	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 0 1 1 0	0 0 1 1 1	0000 0000 0000	MARH ← PCH; MARL ← PCL	
	N, Z, V (0)	0xE11	0	0	0 0	0 1 1 0	1	0	1	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0000 0000 0000	MBR ← MEM[MAR]; C ← 0	
		0xE12	0	1	0 0	0 0 1 0	0	0	0	0	1	0 1 0 1 1	0 0 0 1 1	1 1 1 1 1	0000 0000 0000	EARL ← XRL + MBR + C	
		0xE13	0	1	0 1	0 0 0 0	0	0	0	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	1110 0001 0101	ALUout ← MBR; if N=1 jump to 0xE15	
		0xE14	0	0	1 1	0 0 1 0	0	0	0	0	1	0 1 0 1 0	0 0 0 1 0	1 1 1 1 1	1110 0001 0110	EARH ← XRH + 0 + C; jump to 0xE16	
		0xE15	0	0	0 0	0 0 1 0	0	0	0	0	1	0 1 0 1 0	0 0 0 1 0	1 1 0 0 1	0000 0000 0000	EARH ← XRH + 0xFF + C	
	18 / 19 / 20	0xE16	0	0	0 0	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 1 0 1 0	0 1 0 1 1	0000 0000 0000	MARH ← EARH; MARL ← EARL	
		0xE17	0	0	0 0	0 1 1 1	1	0	1	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0000 0000 0000	MBR ← MEM[MAR]; C ← 1	
		0xE18	0	1	0 0	0 0 0 0	0	0	0	0	1	0 1 1 0 0	1 1 1 1 1	1 1 1 1 1	0000 0000 0000	OPRH ← MBR (C still 1)	load HI-byte to OPRH
		0xE19	0	0	0 0	0 0 1 0	0	0	0	0	1	0 1 0 1 1	0 1 0 1 1	1 1 1 1 1	0000 0000 0000	EARL ← EARL + 0 + C	
		0xE1A	0	0	0 0	0 0 1 0	0	0	0	0	1	0 1 0 1 0	0 1 0 1 0	1 1 1 1 1	0000 0000 0000	EARH ← EARH + 0 + C	
		0xE1B	0	0	1 1	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 1 0 1 0	0 1 0 1 1	1100 0001 0110	MARH ← EARH; MARL ← EARL; jump to 0xC16	continue within LDS Imm. (0xC10)
		0xE1C - 0xE1F	X	X	X X	X X X X	X	X	X	X	X	X X X X X	X X X X X	X X X X X	XXXX XXXX XXXX		
STY	@x	0xE20	0	0	0 0	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 0 1 1 0	0 0 1 1 1	0000 0000 0000	MARH ← PCH; MARL ← PCL	
	N, Z, V (0)	0xE21	0	0	0 0	0 1 1 0	1	0	1	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0000 0000 0000	MBR ← MEM[MAR]; C ← 0	
		0xE22	0	1	0 0	0 0 1 0	0	0	0	0	1	0 1 0 1 1	0 0 0 1 1	1 1 1 1 1	0000 0000 0000	EARL ← XRL + MBR + C	
	17 / 18	0xE23	0	1	0 1	0 0 0 0	0	0	0	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	1110 0010 0101	ALUout ← MBR; if N=1 jump to 0xE25	
		0xE24	0	0	1 1	0 0 1 0	0	0	0	0	1	0 1 0 1 0	0 0 0 1 0	1 1 1 1 1	1100 0010 0000	EARH ← XRH + 0 + C; jump to 0xC20	continue at codeend of STY
		0xE25	0	0	1 1	0 0 1 0	0	0	0	0	1	0 1 0 1 0	0 0 0 1 0	1 1 0 0 1	1100 0010 0000	EARH ← XRH + 0xFF + C; jump to 0xC20	
		0xE26 - 0xE2F	X	X	X X	X X X X	X	X	X	X	X	X X X X X	X X X X X	X X X X X	XXXX XXXX XXXX		
STS	@x	0xE30	0	0	0 0	1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 0 1 1 0	0 0 1 1 1	0000 0000 0000	MARH ← PCH; MARL ← PCL	
	N, Z, V (0)	0xE31	0	0	0 0	0 1 1 0	1	0	1	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	0000 0000 0000	MBR ← MEM[MAR]; C ← 0	
		0xE32	0	1	0 0	0 0 1 0	0	0	0	0	1	0 1 0 1 1	0 0 0 1 1	1 1 1 1 1	0000 0000 0000	EARL ← XRL + MBR + C	
	17 / 18	0xE33	0	1	0 1	0 0 0 0	0	0	0	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	1110 0011 0101	ALUout ← MBR; if N=1 jump to 0xE35	
		0xE34	0	0	1 1	0 0 1 0	0	0	0	0	1	0 1 0 1 0	0 0 0 1 0	1 1 1 1 1	1100 0011 0000	EARH ← XRH + 0 + C; jump to 0xC30	continue at codeend of STS
		0xE35	0	0	1 1	0 0 1 0	0	0	0	0	1	0 1 0 1 0	0 0 0 1 0	1 1 0 0 1	1100 0011 0000	EARH ← XRH + 0xFF + C; jump to 0xC30	
		0xE36 - 0xE3F	X	X	X X	X X X X	X	X	X	X	X	X X X X X	X X X X X	X X X X X	XXXX XXXX XXXX		

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B				A				Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																								
ADDX	@x	0xE40	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xE41	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xE42	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xE43	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	0	1	1	ALUout ← MBR; if N=1 jump to 0xE45	
		0xE44	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	1	1	1	1	1	0	0	1	0	0	0	EARH ← XRH + 0 + C; jump to 0xE46	
		0xE45	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← XRH + 0xFF + C	
		0xE46	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
19 / 20 / 21		0xE47	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xE48	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	OPRH ← MBR (C still 1)	load HI-byte (addend) to OPRH
		0xE49	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xE4A	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xE4B	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	0	0	1	1	0	MARH ← EARH; MARL ← EARL; jump to 0xC46	continue within ADDX Imm. (0xC40)
		0xE4C - 0xE4F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ADDY	@x	0xE50	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xE51	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xE52	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xE53	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	0	1	ALUout ← MBR; if N=1 jump to 0xE55	
		0xE54	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	1	1	1	1	0	0	1	0	1	1	0	EARH ← XRH + 0 + C; jump to 0xE56	
		0xE55	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← XRH + 0xFF + C	
		0xE56	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
19 / 20 / 21		0xE57	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xE58	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	OPRH ← MBR (C still 1)	load HI-byte (addend) to OPRH
		0xE59	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xE5A	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xE5B	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	0	0	1	1	0	MARH ← EARH; MARL ← EARL; jump to 0xC56	continue within ADDY Imm. (0xC50)
		0xE5C - 0xE5F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
ADDS	@x	0xE60	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xE61	0	0	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 0	
		0xE62	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	EARL ← XRL + MBR + C	
		0xE63	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	0	1	ALUout ← MBR; if N=1 jump to 0xE65	
		0xE64	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	1	1	1	1	0	0	1	1	0	0	1	EARH ← XRH + 0 + C; jump to 0xE66	
		0xE65	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	EARH ← XRH + 0xFF + C	
		0xE66	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
19 / 20 / 21		0xE67	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xE68	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	OPRH ← MBR (C still 1)	load HI-byte (addend) to OPRH
		0xE69	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xE6A	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xE6B	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	0	0	1	1	0	MARH ← EARH; MARL ← EARL; jump to 0xC66	continue within ADDS Imm. (0xC60)
		0xE6C - 0xE6F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type	MPC-Addr.	OP	Amux	Cond	ALU	MBR	MAR	RD	WR	ENC	C	B	A	Address	Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																	
SUBX	@x	0xE70	0	0	0 0 1	1 1 0	0	1	1	0	0	1 1 1	0 0 1 1 0	0 1 1 1 0	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xE71	0	0	0 0 0	0 1 1	0	1	0	1	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 0	
		0xE72	0	1	0 0 0	0 0 1	0	0	0	0	0	1 0 1	0 0 1 1 1	1 1 1 1 1	0 0 0 0 0	EARL ← XRL + MBR + C	
		0xE73	0	1	0 1 0	0 0 0	0	0	0	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	1 1 1 0 1	ALUout ← MBR; if N=1 jump to 0xE75	
		0xE74	0	0	1 1 0	0 1 0	0	0	0	0	1	0 1 0	0 0 0 0 0	1 1 1 1 1	0 1 1 1 0	EARH ← XRH + 0 + C; jump to 0xE76	
		0xE75	0	0	0 0 0	0 0 0	0	0	0	0	1	0 1 0	0 0 0 0 0	1 1 1 0 0	0 0 0 0 0	EARH ← XRH + 0xFF + C	
		0xE76	0	0	0 0 0	0 1 1	1	0	1	1	0	0 1 1	1 1 1 1 1	0 0 1 1 1	0 0 0 0 0	MARH ← EARH; MARL ← EARL	
20 / 21 / 22		0xE77	0	0	0 0 0	0 1 1	1	0	1	1	0	1 1 1	1 1 1 1 1	0 0 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]	
		0xE78	0	1	0 0 0	0 0 0	1	0	0	0	1	0 0 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xE79	0	0	0 0 0	0 0 1	0	0	0	0	1	0 1 1	0 1 0 1 1	1 1 1 1 1	0 0 0 0 0	EARL ← EARL + 0 + C	
		0xE7A	0	0	0 0 0	0 0 1	0	0	0	0	1	0 1 0	0 1 0 1 1	1 1 1 1 1	0 0 0 0 0	EARH ← EARH + 0 + C	
		0xE7B	0	1	1 1 1	1 1 1	1	0	1	1	0	1 1 1	0 1 0 1 1	0 1 1 1 1	1 1 1 0 1	MARH ← EARH; MARL ← EARL; jump to 0xC76	continue within SUBX Imm. (0xC70)
		0xE7C - 0xE7F	X	X	X X X	X X X	X	X	X	X	X	X X X	X X X X X	X X X X X	X X X X X	X X X X X	
SUBY	@x	0xE80	0	0	0 0 0	1 1 1	0	1	1	0	0	1 1 1	0 0 1 1 0	0 1 1 1 0	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xE81	0	0	0 0 0	0 1 1	0	1	0	1	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 0	
		0xE82	0	1	0 0 0	0 0 1	0	0	0	0	0	1 0 1	1 0 0 0 1	1 1 1 1 1	0 0 0 0 0	EARL ← XRL + MBR + C	
		0xE83	0	1	0 1 0	0 0 0	0	0	0	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	1 1 1 0 1	ALUout ← MBR; if N=1 jump to 0xE85	
		0xE84	0	0	1 1 0	0 0 1	0	0	0	0	1	0 1 0	0 0 0 0 0	1 1 1 1 1	0 1 1 1 0	EARH ← XRH + 0 + C; jump to 0xE86	
		0xE85	0	0	0 0 0	0 0 1	0	0	0	0	1	0 1 0	0 0 0 0 0	1 1 1 0 0	0 0 0 0 0	EARH ← XRH + 0xFF + C	
		0xE86	0	0	0 0 0	1 1 1	1	0	1	1	0	1 1 1	0 1 0 0 1	0 1 1 1 0	0 0 0 0 0	MARH ← EARH; MARL ← EARL	
20 / 21 / 22		0xE87	0	0	0 1 1	1 1 1	1	0	1	1	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]	
		0xE88	0	1	0 0 0	0 0 1	0	0	0	0	1	0 0 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xE89	0	0	0 0 0	0 0 1	0	0	0	0	1	0 1 1	0 1 0 1 1	1 1 1 1 1	0 0 0 0 0	EARL ← EARL + 0 + C	
		0xE8A	0	0	0 0 0	0 0 1	0	0	0	0	1	0 1 0	0 1 0 1 1	1 1 1 1 1	0 0 0 0 0	EARH ← EARH + 0 + C	
		0xE8B	0	1	1 1 1	1 1 1	1	0	1	1	0	1 1 1	0 1 0 1 0	0 1 1 1 1	1 1 1 0 1	MARH ← EARH; MARL ← EARL; jump to 0xC86	continue within SUBY Imm. (0xC80)
		0xE8C - 0xE8F	X	X	X X X	X X X	X	X	X	X	X	X X X	X X X X X	X X X X X	X X X X X	X X X X X	
SUBS	@x	0xE90	0	0	0 1 1	1 1 1	0	1	1	0	0	1 1 1	0 0 1 1 0	0 1 1 1 0	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xE91	0	0	0 0 0	0 1 1	0	1	0	1	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 0	
		0xE92	0	1	0 0 0	0 0 1	0	0	0	0	0	1 0 1	1 0 0 0 1	1 1 1 1 1	0 0 0 0 0	EARL ← XRL + MBR + C	
		0xE93	0	1	0 1 0	0 0 0	0	0	0	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	1 1 1 0 1	ALUout ← MBR; if N=1 jump to 0xE95	
		0xE94	0	0	1 1 0	0 0 1	0	0	0	0	1	0 1 0	0 0 0 0 0	1 1 1 1 1	0 1 1 1 0	EARH ← XRH + 0 + C; jump to 0xE96	
		0xE95	0	0	0 0 0	0 0 1	0	0	0	0	1	0 1 0	0 0 0 0 0	1 1 1 0 0	0 0 0 0 0	EARH ← XRH + 0xFF + C	
		0xE96	0	0	0 0 0	1 1 1	1	0	1	1	0	1 1 1	0 1 0 0 1	0 1 1 1 0	0 0 0 0 0	MARH ← EARH; MARL ← EARL	
20 / 21 / 22		0xE97	0	0	0 1 1	1 1 1	1	0	1	1	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]	
		0xE98	0	1	0 0 0	0 0 0	1	0	0	0	1	0 0 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xE99	0	0	0 0 0	0 0 1	0	0	0	0	1	0 1 1	0 1 0 1 1	1 1 1 1 1	0 0 0 0 0	EARL ← EARL + 0 + C	
		0xE9A	0	0	0 0 0	0 0 1	0	0	0	0	1	0 1 0	0 1 0 1 1	1 1 1 1 1	0 0 0 0 0	EARH ← EARH + 0 + C	
		0xE9B	0	1	1 1 1	1 1 1	1	0	1	1	0	1 1 1	0 1 0 1 0	0 1 1 1 1	1 1 1 0 1	MARH ← EARH; MARL ← EARL; jump to 0xC96	continue within SUBS Imm. (0xC90)
		0xE9C - 0xE9F	X	X	X X X	X X X	X	X	X	X	X	X X X	X X X X X	X X X X X	X X X X X	X X X X X	

Instruction	Type	MPC-Addr.	OP	Amux	Cond	ALU	MBR	MAR	RD	WR	ENC	C	B	A	Address	Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																	
CMPY	@x	0xEA0	0	0	0 0 1	1 1 1	0	1	1	0	0	1 1 1	0 0 1 1 0	0 1 1 1 0	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xEA1	0	0	0 0 0	0 0 1 1 0	1	0	1	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 0	
		0xEA2	0	1	0 0 0	0 0 0 1 0	0	0	0	0	1	0 0 1 1 1	0 0 0 1 1	1 1 1 1 1	0 0 0 0 0	EARL ← XRL + MBR + C	
		0xEA3	0	1	0 1 0	0 0 0 0 0	0	0	0	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	1 1 0 1 0	ALUout ← MBR; if N=1 jump to 0xEA5	
		0xEA4	0	0	1 1 0	0 1 0 0 0	0	0	0	0	1	0 1 0 0 0	0 1 0 1 0	1 1 1 1 1	1 1 0 1 0	EARH ← XRH + 0 + C; jump to 0xEA6	
		0xEA5	0	0	0 0 0	0 0 0 1 0	0	0	0	0	1	0 1 0 0 0	0 1 0 1 0	1 1 0 0 1	0 0 0 0 0	EARH ← XRH + 0xFF + C	
		0xEA6	0	0	0 0 0	0 1 1 1 1	1	0	1	1	0	1 1 1 1 1	0 1 0 1 0	0 1 1 1 1	0 0 0 0 0	MARH ← EARH; MARL ← EARL	
20 / 21 / 22		0xEA7	0	0	0 0 0	0 1 1 1 1	1	0	1	1	0	1 1 1 1 1	0 1 0 1 0	0 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]	
		0xEA8	0	1	0 0 0	0 0 0 1 0	0	0	0	0	1	0 1 1 0 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xEA9	0	0	0 0 0	0 0 0 1 0	0	0	0	0	1	0 1 1 0 1	0 1 0 1 0	1 1 1 1 1	0 0 0 0 0	EARL ← EARL + 0 + C	
		0xEAA	0	0	0 0 0	0 0 0 1 0	0	0	0	0	1	0 1 0 1 0	0 1 0 1 0	1 1 1 1 1	0 0 0 0 0	EARH ← EARH + 0 + C	
		0xEAB	0	1	1 1 1	1 1 1 1 1	0	1	1	1	0	1 1 1 1 1	0 1 0 1 0	0 1 1 1 1	1 1 0 1 0	MARH ← EARH; MARL ← EARL; jump to 0xCA6	continue within CMPY Imm. (0xCA0)
		0xEAC															
		-	x	x	x x x	x x x x x	x	x	x	x	x	x x x x x	x x x x x	x x x x x	x x x x x	x x x x x	
		0xEAF															
CMPS	@x	0xEB0	0	0	0 0 1	1 1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 0 1 1 0	0 1 1 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xEB1	0	0	0 0 1	1 0 1 1 0	1	0	1	0	0	1 1 1 1 1	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 0	
		0xEB2	0	1	0 0 0	0 0 1 0 0	0	0	0	0	1	0 0 1 1 0	0 0 0 1 1	1 1 1 1 1	0 0 0 0 0	EARL ← XRL + MBR + C	
		0xEB3	0	1	0 1 0	0 0 0 0 0	0	0	0	0	0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	1 1 0 1 0	ALUout ← MBR; if N=1 jump to 0xEB5	
		0xEB4	0	0	1 1 0	0 0 0 1 0	0	0	0	0	1	0 1 0 0 0	0 1 0 1 0	1 1 0 0 1	0 0 0 0 0	EARH ← XRH + 0 + C; jump to 0xEB6	
		0xEB5	0	0	0 0 0	0 0 0 1 0	0	0	0	0	1	0 1 0 0 0	0 1 0 1 0	1 1 0 0 1	0 0 0 0 0	EARH ← XRH + 0xFF + C	
		0xEB6	0	0	0 0 1	1 1 1 1 0	1	1	0	1	0	1 1 1 1 1	0 1 0 1 0	0 1 1 1 1	0 0 0 0 0	MARH ← EARH; MARL ← EARL	
20 / 21 / 22		0xEB7	0	0	0 0 1	1 1 1 1 1	1	0	1	1	0	1 1 1 1 1	0 1 0 1 0	0 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]	
		0xEB8	0	1	0 0 0	0 0 0 1 0	0	0	0	0	1	0 1 1 0 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xEB9	0	0	0 0 0	0 0 0 1 0	0	0	0	0	1	0 1 1 0 1	0 1 0 1 0	1 1 1 1 1	0 0 0 0 0	EARL ← EARL + 0 + C	
		0xEBA	0	0	0 0 0	0 0 0 1 0	0	0	0	0	1	0 1 0 0 1	0 1 0 1 0	1 1 1 1 1	0 0 0 0 0	EARH ← EARH + 0 + C	
		0xEBB	0	1	1 1 1	1 1 1 1 1	0	1	1	0	0	1 1 1 1 1	0 1 0 1 0	0 1 1 1 1	1 1 0 1 0	MARH ← EARH; MARL ← EARL; jump to 0xCB6	continue within CMPS Imm. (0xCB0)
		0xEBC															
		-	x	x	x x x	x x x x x	x	x	x	x	x	x x x x x	x x x x x	x x x x x	x x x x x	x x x x x	
		0xEFF															
LDY	Abs.	0xF00	0	0	0 1 1	1 1 1 1 0	1	1	1	0	0	1 1 1 1 1	0 0 1 1 0	0 1 1 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xF01	0	0	0 0 0	0 1 1 1 1	1	0	1	1	0	1 1 1 1 1	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
		0xF02	0	1	0 0 0	0 0 0 0 0	0	0	0	0	1	0 0 1 1 0	0 0 0 1 1	1 1 1 1 1	0 0 0 0 0	EARH ← MBR (C still 1)	
		0xF03	0	0	0 0 0	0 1 0 0 0	0	0	0	0	1	0 0 1 1 1	0 0 1 1 1	1 1 1 1 1	0 0 0 0 0	PCL ← PCL + 0 + C	
		0xF04	0	0	0 0 0	0 1 0 0 0	0	0	0	0	1	0 0 1 1 0	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0	PCH ← PCH + 0 + C	
		0xF05	0	0	0 0 0	0 1 1 1 1	1	0	1	1	0	1 1 1 1 1	0 0 1 1 0	0 1 1 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
		0xF06	0	0	0 0 0	0 1 1 1 1	1	0	1	1	0	1 1 1 1 1	0 0 1 1 0	0 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
20 / 21 / 22		0xF07	0	1	0 0 0	0 0 0 0 0	0	0	0	0	1	0 0 1 1 0	0 0 0 1 1	1 1 1 1 1	0 0 0 0 0	EARL ← MBR (C still 1)	
		0xF08	0	0	0 1 1	1 1 1 1 0	1	1	1	0	0	1 1 1 1 1	0 1 0 1 0	0 1 1 1 1	0 0 0 0 0	MARH ← EARH; MARL ← EARL	
		0xF09	0	0	0 0 0	0 1 1 1 1	1	0	1	1	0	1 1 1 1 1	0 0 1 1 0	0 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
		0xF0A	0	1	0 0 0	0 0 0 0 0	0	0	0	0	1	0 1 1 0 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	OPRH ← MBR (C still 1)	load HI-byte to OPRH
		0xF0B	0	0	0 0 0	0 0 1 0 0	0	0	0	0	1	0 1 1 0 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	EARL ← EARL + 0 + C	
		0xF0C	0	0	0 0 0	0 0 1 0 0	0	0	0	0	1	0 1 0 1 0	0 1 1 1 1	1 1 1 1 1	0 0 0 0 0	EARH ← EARH + 0 + C	
		0xF0D	0	1	1 1 1	1 1 1 1 0	1	1	1	0	0	1 1 1 1 1	0 1 0 1 0	0 1 1 1 1	1 1 0 1 0	MARH ← EARH; MARL ← EARL; jump to 0xC06	continue within LDY Imm. (0xC00)
		0xF0E															
		-	x	x	x x x	x x x x x	x	x	x	x	x	x x x x x	x x x x x	x x x x x	x x x x x	x x x x x	
		0xF0F															

Instruction	Type	MPC-Addr.	OP	Amux	Cond		ALU				MBR	MAR	RD	WR	ENC	C					B					A					Address										Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																																										
LDS	Abs.	0xF10	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xF11	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xF12	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARH ← MBR (C still 1)	
		0xF13	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xF14	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
		0xF15	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0xF16	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
20 / 21 / 22		0xF17	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← MBR (C still 1)	
		0xF18	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← EARH; MARL ← EARL	
		0xF19	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xF1A	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	OPRH ← MBR (C still 1)	load HI-byte to OPRH
		0xF1B	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xF1C	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xF1D	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	0	0	0	1	0	1	1	0	MARH ← EARH; MARL ← EARL; jump to 0xC16	continue within LDS Imm. (0xC10)
		0xF1E - 0xF1F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
STY	Abs.	0xF20	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xF21	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xF22	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARH ← MBR (C still 1)	
		0xF23	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xF24	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
20 / 21		0xF25	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0xF26	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xF27	0	1	1	1	0	0	0	0	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	EARL ← MBR; jump to 0xC20	continue at codeend of STY
		0xF28 - 0xF2F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
STS	Abs.	0xF30	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V (0)		0xF31	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
		0xF32	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	EARH ← MBR (C still 1)	
		0xF33	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xF34	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
20 / 21		0xF35	0	0	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0xF36	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xF37	0	1	1	1	0	0	0	0	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	0	EARL ← MBR; jump to 0xC30	continue at codeend of STS
		0xF38 - 0xF3F	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Instruction	Type	MPC-Addr.	OP	Amux	Cond	ALU	MBR	MAR	RD	WR	ENC	C	B	A	Address	Instruction Mnemonic	(Micro) Instruction Comments
ADDX	Abs.	0xF40	0	0	0	1	1	1	0	1	0	1	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xF41	0	0	0	0	1	1	1	0	0	1	1	0	0	MBR ← MEM[MAR]; C ← 1	
		0xF42	0	1	0	0	0	0	0	0	0	1	0	1	0	EARH ← MBR (C still 1)	
		0xF43	0	0	0	0	0	1	0	0	1	0	1	1	1	PCL ← PCL + 0 + C	
		0xF44	0	0	0	0	0	0	1	0	0	1	1	0	0	PCH ← PCH + 0 + C	
		0xF45	0	0	0	0	1	1	0	1	0	0	1	1	0	MARH ← PCH; MARL ← PCL	
		0xF46	0	0	0	0	0	0	1	0	0	1	1	0	0	MBR ← MEM[MAR]; C ← 1	
21 / 22 / 23		0xF47	0	1	0	0	0	0	0	0	1	0	1	1	0	EARL ← MBR (C still 1)	
		0xF48	0	0	0	0	1	1	1	0	0	1	1	0	0	MARH ← EARH; MARL ← EARL	
		0xF49	0	0	0	0	0	0	1	0	0	1	1	0	0	MBR ← MEM[MAR]; C ← 1	
		0xF4A	0	1	0	0	0	0	0	0	0	1	1	0	0	OPRH ← MBR (C still 1)	load HI-byte (addend) to OPRH
		0xF4B	0	0	0	0	0	0	0	0	1	0	1	1	1	EARL ← EARL + 0 + C	
		0xF4C	0	0	0	0	0	0	0	0	1	0	1	1	1	EARH ← EARH + 0 + C	
		0xF4D	0	0	1	1	1	1	1	0	0	1	1	1	1	MARH ← EARH; MARL ← EARL; jump to 0xC46	continue within ADDX Imm. (0xC40)
		0xF4E															
		-	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
		0xF4F															
ADDY	Abs.	0xF50	0	0	0	1	1	1	0	1	0	1	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xF51	0	0	0	1	1	1	0	1	0	1	0	1	1	MBR ← MEM[MAR]; C ← 1	
		0xF52	0	1	0	0	0	0	0	0	0	1	0	1	1	EARH ← MBR (C still 1)	
		0xF53	0	0	0	0	0	1	0	0	1	0	1	1	1	PCL ← PCL + 0 + C	
		0xF54	0	0	0	0	0	0	1	0	0	1	1	0	0	PCH ← PCH + 0 + C	
		0xF55	0	0	0	1	1	1	0	1	0	0	1	1	0	MARH ← PCH; MARL ← PCL	
		0xF56	0	0	0	0	0	0	0	0	0	1	0	0	0	MBR ← MEM[MAR]; C ← 1	
21 / 22 / 23		0xF57	0	1	0	0	0	0	0	0	1	0	1	1	1	EARL ← MBR (C still 1)	
		0xF58	0	0	0	0	1	1	1	0	0	1	0	1	1	MARH ← EARH; MARL ← EARL	
		0xF59	0	0	0	0	0	0	0	0	1	0	1	1	1	MBR ← MEM[MAR]; C ← 1	
		0xF5A	0	1	0	0	0	0	0	0	0	1	0	1	1	OPRH ← MBR (C still 1)	load HI-byte (addend) to OPRH
		0xF5B	0	0	0	0	0	1	0	0	1	0	1	1	1	EARL ← EARL + 0 + C	
		0xF5C	0	0	0	0	0	0	1	0	0	1	0	1	1	EARH ← EARH + 0 + C	
		0xF5D	0	1	1	1	1	1	1	0	0	1	1	0	1	MARH ← EARH; MARL ← EARL; jump to 0xC56	continue within ADDY Imm. (0xC50)
		0xF5E															
		-	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
		0xF5F															
ADDS	Abs.	0xF60	0	0	0	1	1	1	0	1	0	1	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xF61	0	0	0	0	0	1	1	0	0	1	1	0	1	MBR ← MEM[MAR]; C ← 1	
		0xF62	0	1	0	0	0	0	0	0	0	1	0	1	1	EARH ← MBR (C still 1)	
		0xF63	0	0	0	0	0	1	0	0	1	0	1	1	1	PCL ← PCL + 0 + C	
		0xF64	0	0	0	0	0	0	1	0	0	1	1	0	0	PCH ← PCH + 0 + C	
		0xF65	0	0	0	1	1	1	0	1	0	0	1	1	0	MARH ← PCH; MARL ← PCL	
		0xF66	0	0	0	0	0	0	1	0	0	1	1	0	0	MBR ← MEM[MAR]; C ← 1	
21 / 22 / 23		0xF67	0	1	0	0	0	0	0	0	1	0	1	1	0	EARL ← MBR (C still 1)	
		0xF68	0	0	0	1	1	1	0	1	0	0	1	1	0	MARH ← EARH; MARL ← EARL	
		0xF69	0	0	0	0	0	0	1	0	0	1	1	0	0	MBR ← MEM[MAR]; C ← 1	
		0xF6A	0	1	0	0	0	0	0	0	0	1	1	0	0	OPRH ← MBR (C still 1)	load HI-byte (addend) to OPRH
		0xF6B	0	0	0	0	0	1	0	0	1	0	1	1	1	EARL ← EARL + 0 + C	
		0xF6C	0	0	0	0	0	0	1	0	0	1	0	1	1	EARH ← EARH + 0 + C	
		0xF6D	0	1	1	1	1	1	1	0	0	1	1	0	1	MARH ← EARH; MARL ← EARL; jump to 0xC66	continue within ADDS Imm. (0xC60)
		0xF6E															
		-	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
		0xF6F															

Instruction	Type	MPC-Addr.	OP	Amux	Cond	ALU	MBR	MAR	RD	WR	ENC	C	B	A	Address	Instruction Mnemonic	(Micro) Instruction Comments
CC changes, No. of instructions to execute																	
SUBX	Abs.	0xF70	0	0	0 0	1 1 1	0	1	1	0	0	1 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xF71	0	0	0 0	0 1 1	1	0	1	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
		0xF72	0	1	0 0	0 0 0	0	0	0	0	1	0 1 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	EARH ← MBR (C still 1)	
		0xF73	0	0	0 0	0 0 1	0	0	0	0	1	0 1 1	1 0 0 1 1	1 1 1 1 1	0 0 0 0 0	PCL ← PCL + 0 + C	
		0xF74	0	0	0 0	0 0 1	0	0	0	0	1	0 1 1	0 0 0 1 1	1 1 1 1 1	0 0 0 0 0	PCH ← PCH + 0 + C	
		0xF75	0	0	0 0	1 1 1	1	0	1	0	0	1 1 1	0 0 0 1 1	0 0 0 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
		0xF76	0	0	0 0	0 1 1	1	0	1	0	0	1 1 1	0 0 0 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
		0xF77	0	1	0 0	0 0 0	0	0	0	0	1	0 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	EARL ← MBR (C still 1)	
22 / 23 / 24		0xF78	0	0	0 0	0 1 1	1	0	1	0	0	1 1 1	0 1 0 1 1	0 0 1 1 1	0 0 0 0 0	MARH ← EARH; MARL ← EARL	
		0xF79	0	0	0 0	0 1 1	1	0	1	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]	
		0xF7A	0	1	0 0	0 0 1	0	0	0	0	1	0 1 1	0 0 0 1 1	1 1 1 1 1	0 0 0 0 0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xF7B	0	0	0 0	0 0 1	0	0	0	0	1	0 1 1	0 1 0 1 1	1 1 1 1 1	0 0 0 0 0	EARL ← EARL + 0 + C	
		0xF7C	0	0	0 0	0 0 1	0	0	0	0	1	0 1 1	0 0 1 1 1	1 1 1 1 1	0 0 0 0 0	EARH ← EARH + 0 + C	
		0xF7D	0	0	1 1 1	1 1 1	1	1	0	0	1	1 1 1	0 1 0 1 1	1 1 1 1 1	0 0 0 1 1	MARH ← EARH; MARL ← EARL; jump to 0xC76	continue within SUBX Imm. (0xC70)
		0xF7E - 0xF7F	X	X	X X X	X X X	X	X	X	X	X	X X X	X X X X X	X X X X X	X X X X X	X	
SUBY	Abs.	0xF80	0	0	0 0 1	1 1 1	0	1	1	0	0	1 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xF81	0	0	0 0 1	1 1 1	1	0	1	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
		0xF82	0	1	0 0 0	0 0 0	0	0	0	0	1	0 1 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	EARH ← MBR (C still 1)	
		0xF83	0	0	0 0 0	0 0 1	0	0	0	0	1	0 1 0	0 1 1 1 1	1 1 1 1 1	0 0 0 0 0	PCL ← PCL + 0 + C	
		0xF84	0	0	0 0 0	0 0 1	0	0	0	0	1	0 1 0	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0	PCH ← PCH + 0 + C	
		0xF85	0	0	0 0 1	1 1 1	0	1	1	0	0	1 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
		0xF86	0	0	0 0 1	1 1 1	0	1	1	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
22 / 23 / 24		0xF87	0	1	0 0 0	0 0 0	0	0	0	0	1	0 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	EARL ← MBR (C still 1)	
		0xF88	0	0	0 0 1	1 1 1	1	0	1	1	0	0 1 1	0 1 0 1 1	0 0 1 1 1	0 0 0 0 0	MARH ← EARH; MARL ← EARL	
		0xF89	0	0	0 0 1	1 1 1	1	0	1	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]	
		0xF8A	0	1	0 0 0	0 0 1	0	0	0	0	1	0 1 1	0 0 1 1 0	1 1 1 1 1	0 0 0 0 0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xF8B	0	0	0 0 0	0 0 1	0	0	0	0	1	0 1 1	0 1 0 1 1	1 1 1 1 1	0 0 0 0 0	EARL ← EARL + 0 + C	
		0xF8C	0	0	0 0 0	0 0 1	0	0	0	0	1	0 1 0	0 1 0 1 1	1 1 1 1 1	0 0 0 0 0	EARH ← EARH + 0 + C	
		0xF8D	0	1	1 1 1	1 1 1	1	1	0	0	1	1 1 1	0 1 0 1 1	1 1 1 1 1	0 0 0 1 1	MARH ← EARH; MARL ← EARL; jump to 0xC86	continue within SUBY Imm. (0xC80)
		0xF8E - 0xF8F	X	X	X X X	X X X	X	X	X	X	X	X X X	X X X X X	X X X X X	X X X X X	X	
SUBS	Abs.	0xF90	0	0	0 1 1	1 1 1	0	1	1	0	0	1 1 1	0 0 1 1 0	0 0 1 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xF91	0	0	0 0 0	0 1 1	1	0	1	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
		0xF92	0	1	0 0 0	0 0 0	0	0	0	0	1	0 1 0	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	EARH ← MBR (C still 1)	
		0xF93	0	0	0 0 0	0 0 1	0	0	0	0	1	0 0 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	PCL ← PCL + 0 + C	
		0xF94	0	0	0 0 0	0 0 1	0	0	0	0	1	0 0 1	1 0 0 1 1	1 1 1 1 1	0 0 0 0 0	PCH ← PCH + 0 + C	
		0xF95	0	0	0 0 1	1 1 1	1	0	1	0	0	1 1 1	0 0 0 1 1	0 0 0 1 1	0 0 0 0 0	MARH ← PCH; MARL ← PCL	
		0xF96	0	0	0 0 1	1 1 1	1	0	1	0	0	1 1 1	0 0 0 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]; C ← 1	
		0xF97	0	1	0 0 0	0 0 0	0	0	0	0	1	0 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	EARL ← MBR (C still 1)	
22 / 23 / 24		0xF98	0	0	0 1 1	1 1 1	0	1	1	0	0	1 1 1	0 1 0 1 1	0 0 1 1 1	0 0 0 0 0	MARH ← EARH; MARL ← EARL	
		0xF99	0	0	0 0 0	1 1 1	1	0	1	0	0	1 1 1	1 1 1 1 1	1 1 1 1 1	0 0 0 0 0	MBR ← MEM[MAR]	
		0xF9A	0	1	0 0 0	0 0 1	0	0	0	0	1	0 1 1	0 0 0 1 1	1 1 1 1 1	0 0 0 0 0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xF9B	0	0	0 0 0	0 0 1	0	0	0	0	1	0 1 1	0 1 0 1 1	1 1 1 1 1	0 0 0 0 0	EARL ← EARL + 0 + C	
		0xF9C	0	0	0 0 0	0 0 1	0	0	0	0	1	0 1 1	0 0 1 1 1	1 1 1 1 1	0 0 0 0 0	EARH ← EARH + 0 + C	
		0xF9D	0	1	1 1 1	1 1 1	1	1	0	0	1	1 1 1	0 1 0 1 1	1 1 1 1 1	0 0 1 1 0	MARH ← EARH; MARL ← EARL; jump to 0xC96	continue within SUBS Imm. (0xC90)
		0xF9E - 0xF9F	X	X	X X X	X X X	X	X	X	X	X	X X X	X X X X X	X X X X X	X X X X X	X	

Instruction	Type	MPC-Addr.	OP	Amux	Cond	ALU	MBR	MAR	RD	WR	ENC	C	B	A	Address										Instruction Mnemonic	(Micro) Instruction Comments
CMPY	Abs.	0xFA0	0	0	0	1	1	0	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
CC changes: N, Z, V, C		0xFA1	0	0	0	0	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
No. of instructions to execute: 22 / 23 / 24		0xFA2	0	1	0	0	0	0	0	0	0	0	1	0	1	1	1	0	0	0	0	0	0	0	EARH ← MBR (C still 1)	
		0xFA3	0	0	0	0	0	1	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xFA4	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	1	1	1	1	1	0	0	PCH ← PCH + 0 + C	
		0xFA5	0	0	0	0	1	1	1	0	0	0	1	1	0	0	0	1	1	1	1	1	0	0	MARH ← PCH; MARL ← PCL	
		0xFA6	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	MBR ← MEM[MAR]; C ← 1	
		0xFA7	0	1	0	0	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	0	0	EARL ← MBR (C still 1)	
		0xFA8	0	0	0	0	1	1	1	0	0	1	1	1	0	1	0	0	1	1	0	0	0	0	MARH ← EARH; MARL ← EARL	
		0xFA9	0	0	0	0	1	1	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	MBR ← MEM[MAR]	
		0xFAA	0	1	0	0	0	0	0	0	0	1	0	0	1	1	1	0	0	0	0	0	0	0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xFAB	0	0	0	0	0	1	0	0	0	1	0	1	1	1	1	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xFAC	0	0	0	0	0	0	1	0	0	1	0	1	0	1	1	0	0	0	0	0	0	0	EARH ← EARH + 0 + C	
		0xFAD	0	0	1	1	1	1	1	1	0	1	1	1	0	1	1	1	1	0	0	1	0	1	MARH ← EARH; MARL ← EARL; jump to 0xCA6	continue within CMPY Imm. (0xCA0)
		0xFAE - 0xFAF	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
CMPS	Abs.	0xFB0	0	0	0	1	1	1	0	1	0	0	1	1	1	0	0	0	1	1	1	0	0	0	MARH ← PCH; MARL ← PCL	
N, Z, V, C		0xFB1	0	0	0	0	1	1	1	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	MBR ← MEM[MAR]; C ← 1	
No. of instructions to execute: 22 / 23 / 24		0xFB2	0	1	0	0	0	0	0	0	0	0	1	0	1	1	1	0	0	0	0	0	0	0	EARH ← MBR (C still 1)	
		0xFB3	0	0	0	0	0	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	PCL ← PCL + 0 + C	
		0xFB4	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	PCH ← PCH + 0 + C	
		0xFB5	0	0	0	1	1	1	0	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	MARH ← PCH; MARL ← PCL	
		0xFB6	0	0	0	0	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	0	0	MBR ← MEM[MAR]; C ← 1	
		0xFB7	0	1	0	0	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	0	0	EARL ← MBR (C still 1)	
		0xFB8	0	0	0	0	1	1	1	1	0	1	1	1	0	1	0	0	1	0	1	1	0	0	MARH ← EARH; MARL ← EARL	
		0xFB9	0	0	0	0	1	1	1	1	1	0	1	1	0	0	0	1	1	1	1	1	0	0	MBR ← MEM[MAR]	
		0xFBA	0	1	0	0	0	0	0	1	0	0	0	1	1	0	0	1	1	1	1	1	0	0	OPRH ← NOT(MBR) (C ← 1)	load negated HI-byte (subtrahend) to OPRH
		0xFBB	0	0	0	0	0	1	0	0	0	0	1	0	1	1	1	0	0	0	0	0	0	0	EARL ← EARL + 0 + C	
		0xFBC	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	1	1	1	1	1	0	0	EARH ← EARH + 0 + C	
		0xFBD	0	0	1	1	1	1	1	1	0	1	1	1	0	0	1	0	1	1	0	1	1	0	MARH ← EARH; MARL ← EARL; jump to 0xCB6	continue within CMPS Imm. (0xCB0)
		0xFBE - 0xFFF	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		