VLSI System Design - Project

The goal of this project is to design a processor, the UKM910, and the necessary interfaces to connect to a VGA port for graphical output and a PS/2 port for keyboard input. This system is then to be synthesized on an FPGA board, and should be capable of running an assembler program implementing a simple calculator that takes input from the keyboard and prints the results to a VGA monitor.

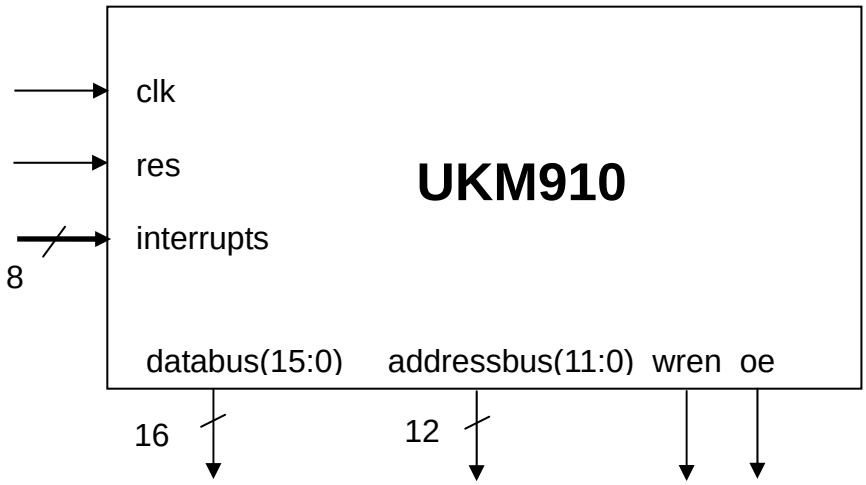
The following document first describes the specifications of the individual components and then explains the testbenches provided to test the individual components in simulation. This is followed by an introduction for the FPGA board used and a description of the tools provided for the development of the UKM910 software.

Contents

Part A: System Specification

Component I - The UKM910 Processor

The UKM910 Architecture is an advanced version of the UKM901. It shares the basic architecture with its predecessor. The following figure shows the block diagram:



Architecture

In addition to the registers present in the UKM901, the UKM910 has the following registers:

Number	Register Contents	Name	Description
0	`0000 TTTT TTTT TTTT`	SP	Stack Pointer
1	`0000 TTTT TTTT TTTT`	PTR1	Pointer 1
2	`0000 TTTT TTTT TTTT`	PTR2	Pointer 2
3	`0000 TTTT TTTT TTTT`	PTR3	Pointer 3
4	`0000 0000 0000 OV C N Z`	PSW	Processor Status Word
5	`0000 000 GIE I7..I0`	IEN	Interrupt Enable
6	`0000 0000  I7..I0`	IFLAG	Interrupt Flags

These registers can be read/written using the LOADR/STORER instructions. They are addressed via their register numbers. The stack pointer SP and the general purpose pointers PTR1 .. PTR3 are used for indirect addressing with the CALL/RET and LOADI*/STOREI* family of commands. The Processor status word holds information about the result of the last executed arithmetic expression, namely the overflow, carry, negative and zero flags. The interrupt enable register determines which interrupts are active. An interrupt is only active when bit 8 of the IEN register (GIE, global interrupt enable) is set **and** the individual interrupt enable bit (I7 .. I0) of the interrupt is also set. If an interrupt occurs, the corresponding flag in the IFLAG register is set.

Instruction Set

Table 1 summarizes the UKM910 instruction set. The OC, C, N and Z columns specify whether the corresponding flags in the PSW register have to be updated after this instruction.

LOADR/STORER are used to load/store the numbered registers (SP, PTR*, PSW, IEN, IFLAG) by their register number. **LOADI/STOREI** are used to write to/read from memory addresses via *indirect addressing*. Instead of specifying the memory address as an operand, as with LOAD/STORE, the memory address stored inside the given register number is used. The variants **LOADIINC/LOADIDEC** and **STOREIINC/STOREIDEC** are useful for reading/writing a large number of consecutive values from/to memory (e.g. arrays). In addition to reading/writing from memory, they also increment or decrement the address pointer, so the next execution of the same command automatically reads or writes the next or previous memory address.

CALL/RET
CALL executes a subroutine. Unlike JUMP, the next value of the program counter is pushed onto the stack. At the end of a subroutine, the RET instructions returns to the point in the program right after where the subroutine was called by taking the value of the program counter from the stack („pop“). This ensures that subroutine calls can be nested, i.e. a subroutine can call another subroutine.
RETI is to be used to return from an interrupt handler (see below).

Interrupt Handling

The following describes the steps taken when an interrupt occurs (a rising edge on one of the external interrupt lines):

The corresponding flag in the IFLAG register is set immediately (this happens asynchronously, not at the next clock cycle)

If GIE is set and the corresponding enable bit in the IEN register is set, the interrupt is handled:

1. GIE is set to ‘0’
2. The corresponding flag in the IFLAG register is cleared
3. The execution of the program continues at the corresponding interrupt address (exactly like a CALL instruction)
4. When the interrupt handler returns (via RETI), GIE is enabled again, and the program continues at the position where it left before the interrupt occurred.

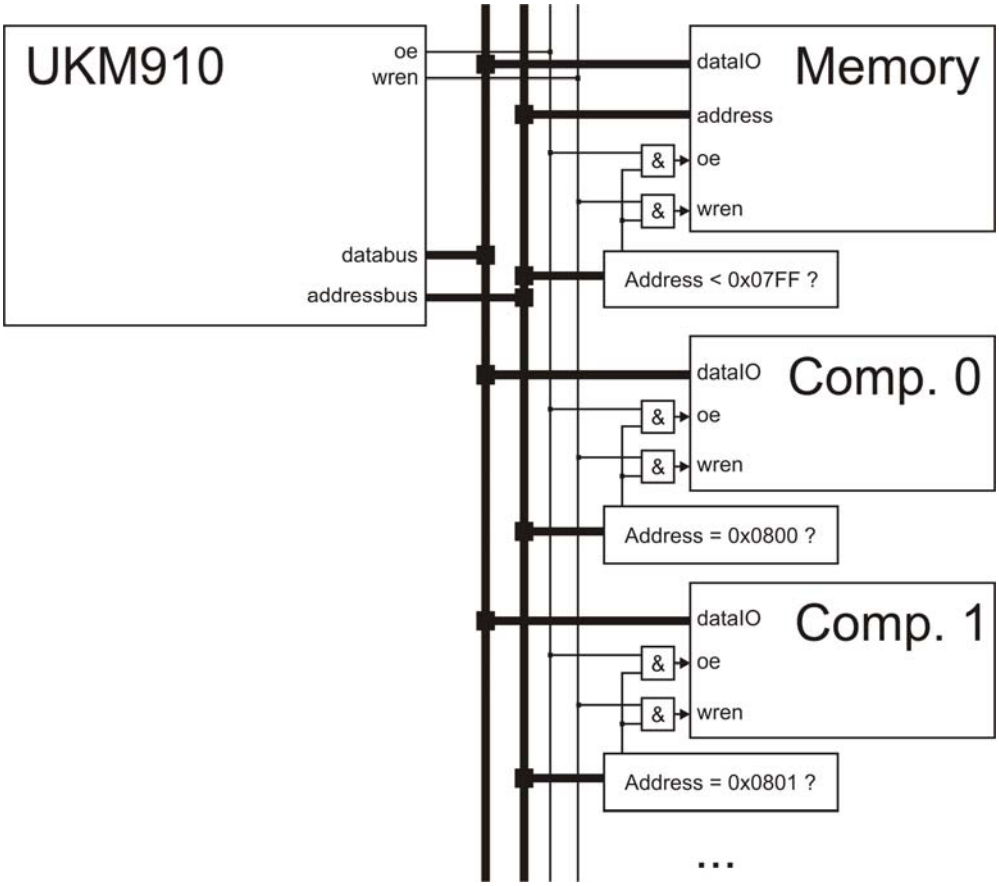
The interrupt handler addresses are:

Interrupt	Address
0	0x000 (identical to reset)
1	0x001
2	0x002
3	0x003
4	0x004
5	0x005

6	0x006
7	0x007

Interfacing External Components

Various possibilities exist to interface external components like the PS2 adapter or the VGA interface, the most straightforward one for this application is „memory mapping“. Here, memory addresses are used to access the extra components, which are equipped with according interfaces. This way, these components can be accessed simply using the processor’s load and store commands. The following figure illustrates this principle for a case where the memory size is reduced to 2048 entries and the addresses starting at 0x0800 are used to interface external components.



Command	Name	OPCODE, binary	OPCODE, hex	Action	OV	C	N	Z
No Operation	NOP	0000 0000 0000 0000	0000	no operation				
Addition	ADD	0001 SSSS SSSS SSSS	1SSS	$ACC := $ACC + [SOURCE]	X	X	X	X
Subtraction	SUB	0010 SSSS SSSS SSSS	2SSS	$ACC := $ACC – [SOURCE]	X	X	X	X
Logical And	AND	0011 SSSS SSSS SSSS	3SSS	$ACC := $ACC AND [SOURCE]			X	X
Logical Not	NOT	0100 0000 0000 0000	4000	$ACC := $ACC			X	X
Complement	COMP	0101 0000 0000 0000	5000	$ACC := $ACC+1	X	X	X	X
Rotate Right	ROTR	0110 0000 0000 0000	6000	$ACC := rotate right $ACC			X	X
Shift Right	SHR	0111 0000 0000 0000	7000	$ACC := shift right $ACC			X	X
Load	LOAD	1000 SSSS SSSS SSSS	8SSS	$ACC := [SOURCE]			X	X
Store	STORE	1001 TTTT TTTT TTTT	9TTT	[TARGET] := $ACC				
Load Register	LOADR	1010 0000 0000 0SSS	A00S	$ACC := $SOURCE			X	X
Load Indirect	LOADI	1010 0000 0001 00SS	A01S	$ACC := [$SOURCE]			X	X
Load Indirect, post-inc.	LOADIINC	1010 0000 0010 00SS	A02S	$ACC := [$SOURCE]; $SOURCE := $SOURCE + 1			X	X
Load Indirect, pre-dec.	LOADIDEC	1010 0000 0011 00SS	A03S	$SOURCE := $SOURCE - 1; $ACC := [$SOURCE]			X	X
Return	RET	1010 0001 0000 0000	A100	$PC := [$SP]; $SP := $SP + 1				
Return from Interrupt	RETI	1010 0010 0000 0000	A200	$PC := [$SP]; $SP := $SP + 1; $IEN[8] := 1				
Store Register	STORER	1011 0000 0000 0TTT	B00T	$TARGET := $ACC				
Store Indirect	STOREI	1011 0000 0001 00TT	B01T	[$TARGET] := $ACC				
Store Indirect, post-inc.	STOREIINC	1011 0000 0010 00TT	B02T	[$TARGET] := $ACC; $TARGET := $TARGET + 1				
Store Indirect, pre-dec.	STOREIDEC	1011 0000 0011 00TT	B03T	$TARGET := $TARGET - 1; [$TARGET] := $ACC				
Jump	JUMP	1100 TTTT TTTT TTTT	CTTT	$PC := TARGET				
Branch on Zero	BZ	1101 TTTT TTTT TTTT	DTTT	$PC := TARGET if ACC==0				
Branch on Negative	BN	1110 TTTT TTTT TTTT	ETTT	$PC := TARGET if ACC<0				
Call Function	CALL	1111 TTTT TTTT TTTT	FTTT	$SP := $SP – 1; [$SP] := $PC + 1; $PC := TARGET;				

Table 1:UKM910 Instruction Set

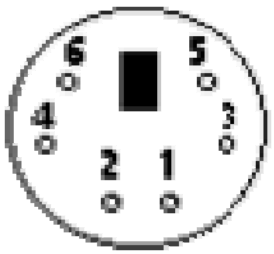
Component II - PS/2 Interface

Problem Description

The aim is to develop an interface using a standard PS/2 keyboard protocol. The keyboard will be used as an input interface with the processor (UKM 910) through the FPGA.

The Physical Interface

The physical PS/2 port is one of two styles of connectors: The 5-pin DIN or the 6-pin mini-DIN. The available FPGA demo board supports the 6-pin mini-DIN.
Vcc/Ground provides power to the keyboard/mouse. The keyboard or mouse should not draw more than 275 mA from the host and care must be taken to avoid transient surges. The Data and Clock lines are both open-collector with pull-up resistors to Vcc. The bus is "idle" when both lines are high (open-collector).



1 - Data
2 - Not Implemented
3 - Ground
4 - Vcc (+5V)
5 - Clock
6 - Not Implemented

Figure 1. PS/2 Socket.

The Protocol

Keyboard transmits data in a clocked serial format consisting of a start bit, 8 data bits (LSB first), an odd parity bit and a stop bit. The clock signal is only active during data transmit. The generated clock frequency is usually in the range 10 - 20 kHz. Each bit should be read on the falling edge of the clock.

Below is a diagram explaining the data packet and the timing related to the mouse clock while receiving a byte from the PS-2 keyboard:

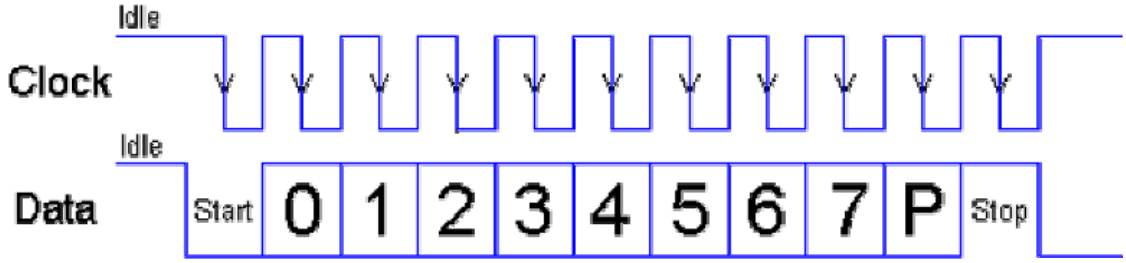


Figure 2. Keyboard transmission timing diagram.

The above waveform represents a one byte transmission from the Keyboard. The keyboard may not generally change its data line on the rising edge of the clock as shown in the diagram.

The data line only has to be valid on the falling edge of the clock. The Least Significant Bit is always sent first. Below is some samples form the generated scan codes from the keyboard:

Key	Scan code (Binary)
1	00010110
2	00011110
3	00100110
4	00100101
5	00101110
6	00110110
7	00111101
8	00111110
9	01000110
0	01000101
+	01111001
-	01111011
*	01111100
/	01001010

Table 1. PS/2 Keyboard scan codes (samples).

PS/2 Module Interface

```
Entity PS2_Module is
  port(         Clk          : in  std_logic;          -- System Clock
                Reset        : in  std_logic;          -- System Reset
                OE           : in  std_logic;          -- Output enable
                PS2_Clk      : in  std_logic;          -- Keyboard Clock Line
                PS2_Data     : in  std_logic;           -- Keyboard Data Line
                IRQ          : out std_logic;          -- Interrupt signal
                PS2_DataOutZ: out std_logic_vector(7 downto 0)); -- Eight bits Data Out
End Entity;
```

Tips and Tricks

The keyboard protocol offers an asynchronous communication, and to achieve the desirable communication with the processor it would be necessary to have synchronous communication protocol. So, the following figures will propose some ideas how to convert asynchronous to synchronous communication.

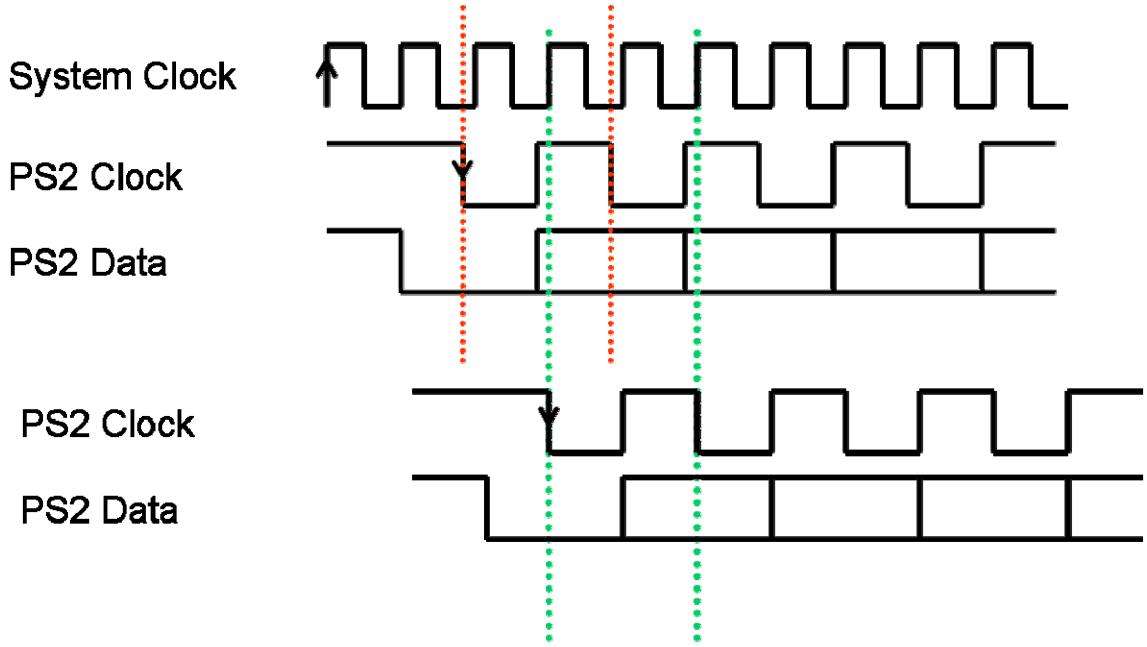


Figure 3. Timing diagram illustrating the conversion from asynchronous to synchronous.

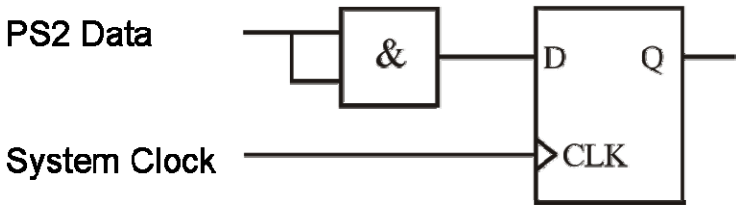


Figure 4. Hardware for asynchronous to synchronous conversion.

Component III - Display Interface

Problem

An interface to output data to a screen using a standard VGA interface should be implemented. The desired resolution for the screen is 640x480 Pixel, the color depth possible using this FPGA board is 3 Bit/Pixel (thus 8 possible colors).

As the monitor picture needs to be refreshed at a rate of ~60Hz, it is necessary to buffer the data to be outputted. Buffering the individual pixels would require at least 640x480x3 Bit = 112.5 kB, an amount of memory not available on the FPGAs used here.

As only characters need to be outputted in the targeted calculator application, a buffering scheme which is a lot more memory-efficient is to be used: Only the characters to be displayed are buffered. Each character is 8 Pixels wide and 12 Pixel high, thus only 80x40 Characters need to be buffered, requiring only 3.2 kB video RAM.
It is therefore necessary to implement a VGA interface which accesses the character RAM, translates the character information to the corresponding pixel information and outputs this information with the correct timing.

VGA Timing

There are two main signals used to control the VGA interface: the vertical synchronization signal vsync and the horizontal synchronization signal hsync. Historically, these signals were used to control the electron beam of a monitor: First, a pulse on vsync and hsync would reset the electron beam to the upper left corner of the screen. Then, the first line of pixel would be outputted, and after that, a hsync pulse would reset the electron beam to the beginning of the next line and so on, until all lines are written. After that, a new vsync signal resets the beam back to the initial position.

There are extra constraints upon this timing, as extra time is required before and after a vsync and hsync pulse, called ‘front porch’ and ‘back porch’ of the corresponding pulse. This is illustrated in Fig. 1.

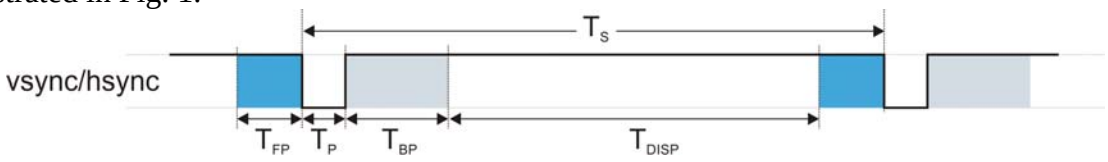


Figure 1: Timing contraints on the vsync/hsync signals

The synchronization between horizontal and vertical synchronization signals should be as illustrated in Fig. 2, thus the front porch of hsync starts with the falling edge of the vsync signal.

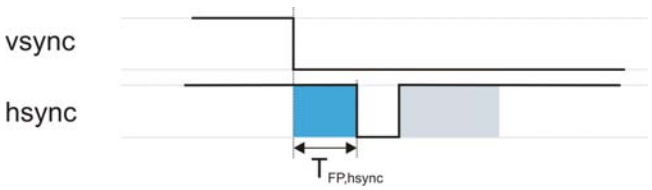


Figure 2: Synchronization between vertical and horizontal synchronization signals

The overall timing is as illustrated in Fig. 3: The pixel data is outputted when both vsync and hsync are ‘in display mode’. The individual pixels are outputted with the speed of the pixel clock (please note that the number of hsync cycles during the various vsync periods as well as the number of pixel lines does not reflect the actual numbers used).

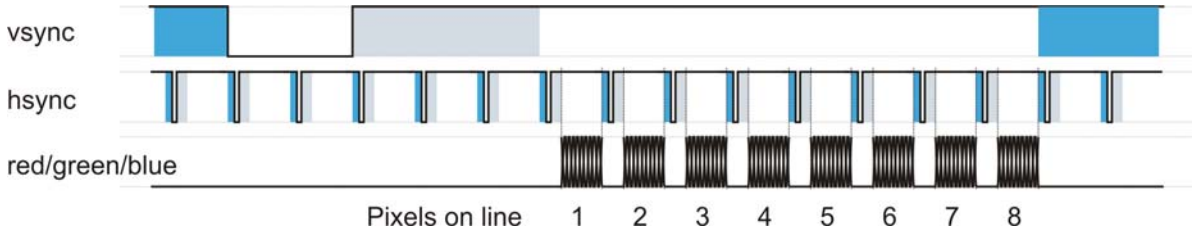


Figure 3: Overall timing diagram

The actual timing parameters to drive a display with a resolution of 640x480 pixels are as follows:

Symbol	**Parameter**	**Value**		
f_{pixel}	Pixel clock	25 MHz		
Horizontal Timing				
Symbol	**Parameter**	**Time**	**Pixel Clocks**	
T_S	Sync pulse time	32 µs	800	
$T_{FP,hsync}$	Front porch	0.64 µs	16	
$T_{P,hsync}$	Pulse width	3.84 µs	96	
$T_{BP,hsync}$	Back porch	1.92 µs	48	
T_{DISP}	Display Time	25.6 µs	640	
Vertical Timing				
Symbol	**Parameter**	**Time**	**Pixel Clocks**	**Lines**
T_S	Sync pulse time	16.7 ms	416800	521
$T_{FP,vsync}$	Front porch	0.320 ms	8000	10
$T_{P,vsync}$	Pulse width	0.064 ms	1600	2
$T_{BP,vsync}$	Back porch	0.928 ms	23200	29
T_{DISP}	Display Time	15.36 ms	384000	480

Character Table

To be able to output characters, it is necessary to have the pixel information for each character. This data is provided in the file „fontROMPackage.vhdl“. The coding of the individual characters is as shown in Fig. 4.

Figure 4: Character Coding Table

The numbers in the columns give the least significant bits of the code, the ones in the rows the most significant bits. The character „C“ e.g. has the code 0x043.

Each character has a size of 8x12 Pixels. The array „fontROMArray“ in the mentioned package is organized as follows:

- It has a size of 256x12 entries, each entry 8 Bits wide.
- The first 256 entries contain the first row of the character with the corresponding code, the second 256 entries the second row and so on. Thus, the address to obtain a specific row of a specific character is composed as follows:

  Bits 7…0: Character code
  Bits 11…8: Line Number (can be 0…11)

  E.g. to get pixel line 5 of the character C, one has to read array entry number 0x543.
- The entries contain the pixel lines MSB first, thus accessing the entries as std_logic_vector(7 downto 0), the vector entry 0 is the leftmost pixel.

Using the Package

- Compile the file „fontROMPackage.vhdl“ along with the other VHDL files.
- In the VHDL file where you want to use the character array, include the statement

```
use work.fontROM.all;
```

  It is then possible to access e.g. the character pixel line 5 of the character C using

```
signal pixelLineNow : std_logic_vector(7 downto 0);
...
pixelLineNow <= fontROM(1347);
```

  (note that 0x543 = 1347).
- It is highly advisable to access the character ROM via a synchronous interface. This is due to the fact that the FPGA synthesis tool then implements the corresponding ROM as a rather compact block RAM, otherwise this ROM occupies a rather big portion of the FPGA.

Part B: Test Environment

For all major system components, a test environment is provided:

- UKM910 Processor: simulation testbench with test programs for processor functionality
- PS2 interface:
  - PS2 input component: simulate the input waveforms given by a PS2 keyboard in VHDL simulation environment
  - PS2 scan code output: Component to output PS2 keyboard scan codes in hardware (required for milestone 2)
- VGA output component: check the timing of signals outputted to a VGA interface and output screen contents in simulation

Please note that using the PS2 input and the VGA output component, the complete calculator can be tested in simulation, which is usually a lot more effective than debugging in hardware due to the complete signal visibility.

In the following, the use of each test component is described shortly.

UKM910 Processor Testbench

The basic testbench provided for the UKM901 in the processor exercise is also used for the UKM910 developed here. To test the added functionality of the UKM910, various test programs are provided. To run the test, rename the corresponding memory file to memory.txt and use this file to replace the original memory.txt file provided with the UKM testbench.

- testProgram2.hex checks the complete functionality of the UKM910 instruction set. For details, check the commented assembler code provided as testProgram2.asm.

- memory.loadr_storer.txt checks the functionality of the register load/store instructions.
- memory.loadi_storei.txt checks the functionality of the load/store indirect instructions.
- memory.storeidec_loadiinc.txt checks the functionality store indirect with pre-decrement and the load indirect with pre-increment instructions.
- memory.storeiinc_loadidec.txt checks the functionality store indirect with post-increment and the load indirect with post-decrement instructions.
- memory.call_ret.txt checks the functionality of the subroutine call and return instructions.

PS2 Input Component

The PS2 input component is used to create waveforms following the PS2 keyboard protocol within the VHDL simulation environment. To use this component

- Compile the file PS2Tester.vhdl along with the other vhdl files.
- In your testbench, add the component

```
component PS2Tester
   port (
        PS2clk  : out std_logic;
        PS2data : out std_logic );
end component;
```

- If you want to get specific input scan codes, edit the file PS2Tester.vhdl. In the process „writeData“, you can add additional function calls in the form

```
writePS2Data(PS2clk, PS2data, X"7C");
```

  or edit the existing ones. The keystrokes simulated with the scan codes given in the file are documented there.

PS2 Scan Code Output

The PS2 scan code output component is provided to test your PS2 interface in hardware. It has an interface where your own PS2 interface component can be connected. The scan codes sent by a keyboard are then outputted on the LCD display which is mounted on the FPGA development board. To use this test system, proceed as follows:

- Edit the file PS2TestEnvironment.vhdl, replace the component „ps2interface“ (along with the corresponding instantiation) by your own PS2 component.
- Requirements for the PS2 component:
  - The availability of a new scan code byte has to be signalled by a rising edge on the signal „dataAvailable“.
  - The scan code byte needs to be written to the signal „data“. The scan code byte has to be valid when „dataAvailable“ rises.
- To implement the component on the FPGA board, include the files
  PS2TestEnvironment.vhdl
  LCDInterface.vhdl
  PS2TestEnvironment.ucf
  + your own PS2 interface design

  in a Xilinx ISE project and follow the usual synthesis procedure.

VGA Output Component

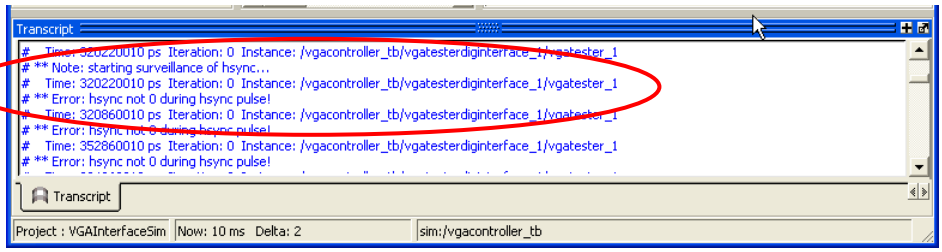
As it may be very time-consuming to debug the VGA signal timing directly in hardware, a test system checking the timing in simulation is provided. It also offers the functionality to output the current screen output to a bitmap file.

Use of the VGA Testbench

- Compile the file vgaTestPack.vhdl along with the other vhdl files.
- In your testbench, add the component

```
component vgaTesterDigInterface
   port (
        vsync : in std_logic;
        hsync : in std_logic;
        red   : in std_logic;
        green : in std_logic;
        blue  : in std_logic);
end component;
```

  along with a corresponding instance and connect the signals to the outputs of your VGA interface.
- During simulation, check the simulator console output („Transcript“). If timing errors occurs, corresponding error messages are outputted there.



- If no errors occurred, a bitmap file called „screenDump.bmp“ is outputted to your simulation base directory. It contains the screen data outputted during the last run. (It is created after all lines of a screen being outputted, thus the simulation needs to run at least 16.7ms).

Part C: Hardware Implementation

The following sections outline the implementation of the design on the FPGA hardware platform. First, the hardware is introduced shortly and a tutorial showing how a simple design is implemented in hardware is given. After that, it is explained how the memories present in the design (e.g. the processor memory) is implemented best. As the actual hardware components exhibit signal delays, it is important to consider signal timing for a hardware implementation. This is explained in detail in the last section.

Hardware Platform: FPGA board

Each group is provided a *Spartan-3E Starter Kit Board* offering a Xilinx Spartan 3E FPGA, a LCD display, VGA and PS2 interface and several other devices. A detailed description of the FPGA board and the individual components can be found in the file S3EStarter_ug230.pdf.

When implementing designs on the FPGA, it is important to define to which FPGA I/O pin the I/O signals in the VHDL description should be connected. This task is accomplished using "User-Constraint Files" (UCF). A template for the UCF necessary for this project is provided as *calculator.ucf*.

See the Synthesis Tutorial for an introduction into synthesizing a simple example design, and for using tools for on-chip debugging.

Implementing Synthesizable Memories with Predefined Contents

For the implementation of the microcontroller, it is necessary to init the memory with the program code, i.e. to instantiate a memory with predefined contents.
For simulation, this problem was solved using an entity reading a file with the desired memory contents at simulation start. A construct like this is not synthesizable, though. There are two basic concepts to instantiate a memory block with predefined contents on an FPGA, which are:

- Use of IP block: It is possible to use vendor-provided IPs which also allow for a direct instantiation of a so-called BlockRAM (RAM component on the FPGA), also allowing the input of initial values from a file. As the creation of IPs is rather slow in Xilinx ISE though and needs to be repeated each time the memory contents change, this approach is not recommended.
- Use of signal array with initial values: It is possible to assign initial values to a signal representing a memory in VHDL, e.g.

```
type memoryArray is array (0 to (2**addr'length-1)) of
                           std_logic_vector(15 downto 0);
signal data : memoryArray :=
       ( X"c008",
         X"c1d1",
         -- ...
         X"a200",
         X"a200",
         others => X"0000");
```

  If the corresponding memory interface is implemented synchronous, a component realized like this is also implemented as BlockRAM by the synthesis tool. For the

creation of the vector with the initial values, the script „build2vhdl“ provided with the assembler can be used.

- **Please note:** The **memory interface** should always be implemented **as "clean" as possible**, meaning that there should be a component implementing only the memory and no other control tasks or the like. All logic necessary to implement an advanced behaviour of the memory should be realized outside of this basic memory component. Otherwise it is possible that either the component is not recognized as memory component and therefore might not fit on the FPGA as it is implemented as distributed memory, or – even worse – it seems to be implemented correctly but exhibits functional errors.

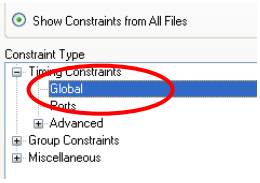
Timing Analysis in Xilinx ISE

When implementing a design, it is necessary to define constraints for a design. The first important class of constraints was already introduced in the „Synthesis Tutorial“: The definition which input/output of the implemented VHDL component should be connected to which package pin. The second important constraint class is the timing of the design and shall be explained in the following in more detail.

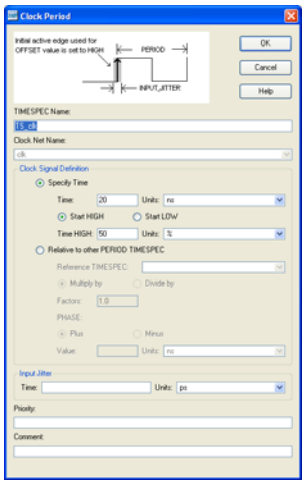
Applying Timing Constraints

To have a set of meaningful timing constraints, it is at least necessary to define the required clock period for all clocks in the design. To do so

- Select Top-Level entity of your design and start „User Contraints -> Create Timing Constraints“ in the „Processes“ section.
- Make sure that in the „Sources“ section, under „Constraint Type“, „Global“ is selected. Then all nets which were recognized as clock in are displayed in the main workspace. All nets which connect to a flipflop/latch clock input are automatically regarded as clocks, thus if you use e.g. edge-sensitive contstructs to detect interrupts, the corresponding nets may be regarded as clock nets, too.



- Double-click the „Period“ field for the input clock to define the corresponding parameters. As it connects to a 50 MHz clock source, it needs to be constrained to a period of 20ns with 50% high cycle.



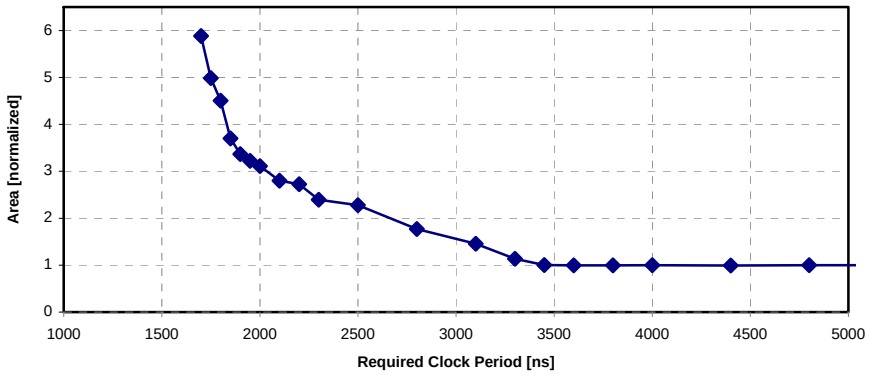
- Constrain the other clocks accordingly. You need to constrain clocks created by dividing the input clock, too, as the division factor is not recognized by the ISE tools automatically. You can use the section „Relative to other PERIOD TIMESPEC“ in the clock period definition dialog, though (please note that for a divided clock, you need to define a „Multiply by“ factor, as this definition relates to clock period and not clock frequency). Finally, all (real) clocks should be constrained.

Clock Net Name	Period	Pad to Setup	Clock to Pad
clk	20 ns. HIGH 50%		
calculator_i/PS2_Module_2/IRQ		N/A	N/A
calculator_i/vgaControlSystem_1/vgaCor	TS_clkDiv21 * 800 Phase 0 ns	N/A	N/A
clkDiv21	TS_clk * 2 Phase 0 ns.	N/A	N/A

- Selecting the Constraint Type „Ports“ (in „Sources“ section), it is additionally possible to constrain the timing at the in- and outputs. For inputs, the „Pad-to-Setup“ time can be defined, meaning how much time before the triggering clock edge the respective input signal is valid (this information tells the synthesis tool „how much logic can be placed before the flipflop latching this input“). For outputs, the „Clock-to-Pad“ time can be defined, meaning when after the clock edge the output signal at a FPGA pin needs to be valid (thus giving the information how much combinational logic can be placed in front of that output pin).
  As it is may be hard to define constraints here, these field can also be left blank and it can be checked after synthesis if the respective timing values lie within reasonable limits.

Why Applying Constraints?

One may one why it is necessary to apply timing constraints before the synthesis procedure instead of synthesizing the design and then checking for the maximum clock frequency possible.
The reason for this is that the timing information is an important hint for the synthesis tools on how to implement the design: There are usually many ways to implement logic with a certain functionality. If the synthesis/mapping/place&route tools recognize that certain blocks do not fulfil the timing requirement, these blocks may be optimized in terms of placement and logic design, possibly at the cost of an increased area. This is illustrated in the following figure where the required area vs. the required clock period is plotted for an ASIC design.

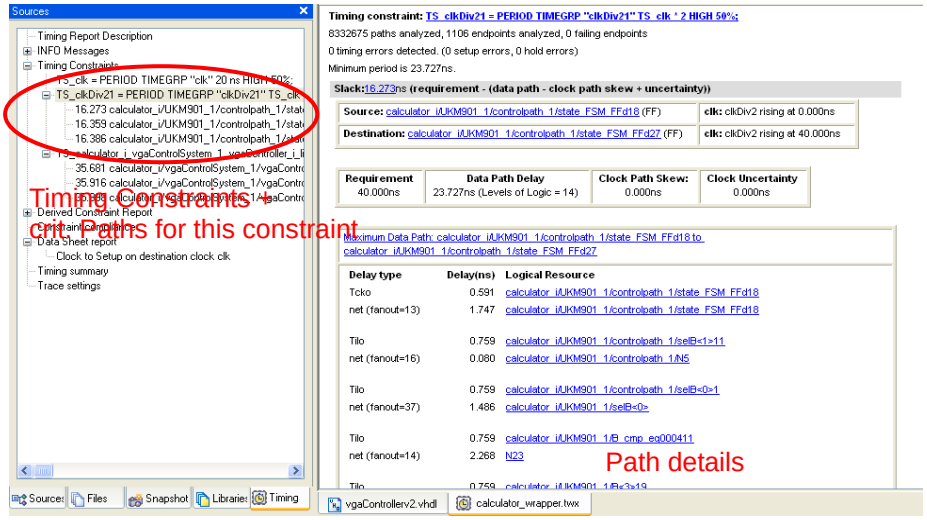


Timing Analysis

To identify the critical path in a design and to check whether the timing constraints could be fulfilled, timing analysis tools are used. There are multiple steps in the design process when the timing can be checked:

- After synthesis: Check the synthesis report (in ‘Processes’ region: ‘Synthesize – XST’ → ‘View Synthesis Report’). At the end of the report, the critical path is shown.
- After Mapping: Select ‘Implement Design’ → ‘Map’ → ‘Generate Post-Map Static Timing’ → ‘Analyze Post-Map Static Timing’.
- After Place & Route: Select ‘Implement Design’ → ‘Place & Route’ → ‘Generate Post- Place & Route Static Timing’ → ‘Analyze Post- Place & Route Static Timing’.

The later in the design process the timing is checked, the more accurate are the results (because the impact of the following design steps otherwise needs to be estimated using statistical models).
Within the timing analysis tool (started for Post-Map and Post-Place & Route Analysis) the defined constraints (in our case: the clock periods) are shown, along with the most critical path for this constraint and the corresponding 'path slack'. The path slack is the difference between the required time (here: clock period) and the actual path delay. Thus a positive slack means the design works with the desired clock frequency, whereas a negative slack indicates a failure (path delay is higher than the desired clock period).
By selecting a specific path in the 'Sources' section, it is possible to inspect which instances are on this path in the main section.



Timed VHDL Simulation of the Synthesized Design

It is often also of interest to be able to check the dynamic timing, i.e. check the timing behaviour of the synthesized model. This is done as follows:

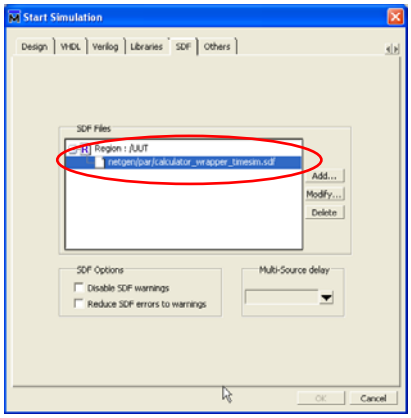
- Xilinx ISE outputs a VHDL netlist which contains all the basic cells used for the in the synthesized design (Look-Up-Tables, BlockRAMs, Multipliers, etc.) along with their interconnects.
- Xilinx ISE outputs a SDF (Standard Delay Format) file which contains information on the delay of the design elements and interconnects. This file is read by the simulation tool (Modelsim) and used during the simulation to apply the correct delays.

Just like the timing analysis, a timed simulation can also be conducted post-Synthesis, post-Map and post-Place & Route with increasing level of accuracy.

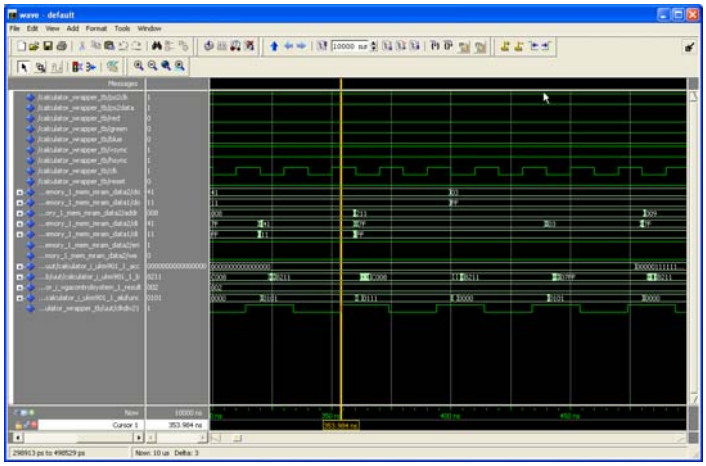
For a post-Place & Route timed simulation

- Include the testbench for the top-level entity into the ISE environment (with Add Source…). Please note: The instance name for the top-level entity within the testbench has to be 'UUT' (Unit under Test), otherwise the SDF delay annotation will fail.
- In the 'Processes' section, select 'Implement Design' → 'Place & Route' → 'Generate Post-Place & Route Simulation Model'.
- In the 'Sources' section, select 'Sources for: Post-Route Simulation'. The testbench with the VHDL netlist created by Xilinx ISE is shown. You can open the netlist to get an idea of the netlist structure created.
- Select the testbench in the 'Sources' section and in the 'Processes' section, select 'ModelSim Simulator' → 'Simulate Post-Place & Route Model'.

- Modelsim automatically starts with the correct model and the timing information. You can double-check that the SDF model really has been included by choosing ‘Simulate’ → ‘Start Simulation…’ → Tab ‘SDF’. Here a SDF file for your design should be shown.

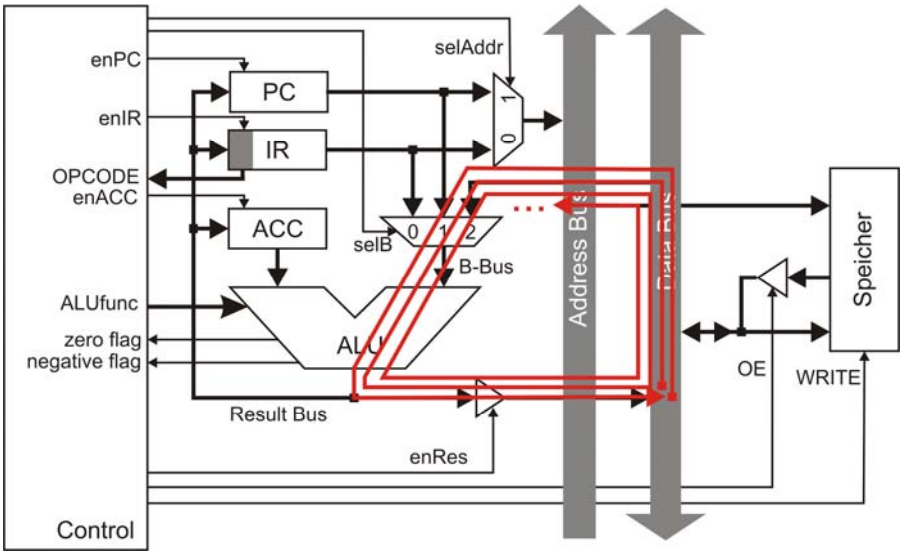


- You can simulate your model as usual. You will find that the signals don’t change right after the clock edge anymore but have considerable delays.
  The drawbacks of this simulation method are that due to the flattening of the design hierarchy during synthesis, the signal visibility is reduced, though, and simulation speed is considerably reduced due to the increased number of components.



Timing Issues in the UKM910

If you follow the UKM910 structure proposed in the lecture closely, it is well possible that the critical path detected for your design is considerably longer than the required clock period. Analyzing the path more closely, you may notice that it passes the result and B-Bus several times. The reason for this is the structure of the UKM910:



Regarding only the circuit topology, a signal may travel e.g. from the memory to the data bus, through the B bus and the ALU to the result bus and back to the data bus and so on. This is called a '**combinatorial loop**', a warning concerning this is issued during the synthesis step and the loop itself is automatically 'broken' for timing analysis. Still, a very long path remains from e.g. databus(0) via the ALU to databus(1) via the ALU … to databus(15).
Even though this path topologically exists, it is never used as the control ensures that one never reads from the result bus and writes from it at the same time, this cannot be detected by the timing analysis tool, though. Such paths are therefore called '**False Path**'.

There are means to tell the timing analysis tool that certain paths are false, so that these paths are ignored for timing analysis, unfortunately this does not work very well in Xilinx ISE, though. The best solution therefore is to topologically break the combinational loop in the design, e.g. implementing the write-path to the data bus such that it originates at the registers which may can written to the memory instead of using the result bus as origin.

Part D: Software Development

This document describes use of the assembler and debugger for the UKM910. The assembler/disassembler and linker tools are a port of the GNU Binutils package, the debugger is a port of the GNU Debugger. Thus further information on the general use of the programs described in this document can be found in the documentation for the corresponding GNU programs.
The package furthermore contains the editor „GNU Emacs“, which serves as user interface for the GNU Debugger port.

Code Blocks to be developed

Please note that within this project, it is not obligatory to develop the complete software implementing the desired calculator functionality, but the core program is provided in the file *calculator_body.asm* . This code though needs to be extended to interface your own hardware, details can be found within the file.

Assembler

Invoking Assembler and Linker

To simplify the creation of the translated output files, there are three scripts which automatically call assembler, linker and output file creator program depending on the desired target. To use the scripts, place your source files in the UKM910Dev\source directory. Then open a Command Prompt (Eingabeaufforderung), go to the „UKM910Dev“ directory and enter

- `build2elf programName`
  to get an output in ‘elf’ format, which is the right choice for debugging using GDB (the output resides in the \elf subdirectory)
- `build2hex programName`
  to get an output in ‘hex’ format, which can then be copied to the memory.txt file and used in simulation (the output resides in the \hex subdirectory)
- `build2array programName`
  to get an output in the format of a vhdl array, which can then be used for FPGA implementation (the output resides in the \array subdirectory)

`programName` in all cases is the name of the assembler file without the file extension (e.g. to compile the file myProgram.asm, you need to type `build2elf myProgram`).

Assembler Commands

Regular Commands

Mnemonic	OPCODE	Description	Flags
nop	0000	No operation	
Logic and Arithmetic Commands			
not	4000	Bitwise invert acculator register	N, Z
comp	5000	Complement accumulator register	OV, C, N, Z
shr	7000	Shift right accumulator register by 1 Bit	N, Z
rotr	6000	Rotate right accumulator register by 1 Bit	N, Z
add SADDR	1SSS	Add content of source address to accumulator, store result in accumulator	OV, C, N, Z
sub SADDR	2SSS	Substract content of source address from accumulator, store result in accumulator	OV, C, N, Z
and SADDR	3SSS	Logical AND of content of source address and accumulator, store result in accumulator	N, Z
LOAD/STORE Commands			
load SADDR	8SSS	Load content of source address to accumulator	N, Z
load REG	A00R	Load content of register to accumulator; REG may be: $sp, $ptr1, $ptr2, $ptr3, $psw, $ien, $iflag	N, Z
load (REG)	A01R	Load content of address stored in REG to accumulator; REG may be: $sp, $ptr1, $ptr2, $ptr3	N, Z
load dec(REG)	A02R	Before operation decrement REG by one, then load content of address stored in REG to accumulator; REG may be: $sp, $ptr1, $ptr2, $ptr3	N, Z
load (REG)inc	A03R	Load content of address stored in REG to accumulator, after operation increment REG by one; REG may be: $sp, $ptr1, $ptr2, $ptr3	N, Z
store TADDR	9TTT	Store accumulator to target address	
store REG	B00R	Store accumulator to register; REG may be: $sp, $ptr1, $ptr2, $ptr3, $psw, $ien, $iflag	
store (REG)	B01R	Store accumulator to address stored in register; REG may be: $sp, $ptr1, $ptr2, $ptr3	
store dec(REG)	B02R	Before operation decrement REG by one, then store accumulator to address stored in register; REG may be: $sp, $ptr1, $ptr2, $ptr3	
store (REG)inc	B03R	Store accumulator to address stored in register, after operation increment REG by one; REG may be: $sp, $ptr1, $ptr2, $ptr3	
Control Commands			
jump TADDR	CTTT	Continue program execution at target address	
bz TADDR	DTTT	Continue program execution at target address if acculmulator is zero	
bn TADDR	ETTT	Continue program execution at target address if acculmulator is negative	
call TADDR	FTTT	Call subprogram at target address (by decrementing stack pointer by one, storing current $pc value on stack, setting $pc to target address)	
ret	A100	Return from subprogram (by restoring $pc value and incrementing stack pointer by one)	
reti	A200	Return from interrupt (by restoring $pc value, incrementing stack pointer by one and setting global interrupt enable bit)	

Alias Commands

`Push` is equivalent to `store dec($sp)`
`Pop` is equivalent to `load ($sp)inc`

Pseudo-Instuctions

Mnemonic	OPCODE	Description	
`getaddr SADDR`	`0SSS`	No operation	

The pseudo-instruction ‘getaddr’ is meant to be used in combination with the load/store indirect commands: there it is necessary to load the address where a variable is stored to a pointer register, exactly this can be accomplished using getaddr. In the following example, the sum of three variables is calulated using the load indirect command:

```
...
      load  ARRAY1ADDR
      store $ptr1
      load  ($ptr1)inc
      store SUM
      load  ($ptr1)inc
      add   SUM
      store SUM
      load  ($ptr1)inc
      add   SUM
      store SUM
...
ARRAY1:     .word 0x001
ARRAY2:     .word 0x002
ARRAY3:     .word 0x003
ARRAY1ADDR: getaddr ARRAY1 # address of array1 is stored in this variable
...
SUM:        .word 0x000
```

Important Assembler Directives

Every symbol of the form

```
symbol: ....
```

indicates to the assembler a specific location in memory. This syntax is thus used to define marks to be used with branch and call commands as well as to define variables, which can then be used with the arithmetic/logic as well as the load/store commands.

The syntax of the compiler requires that the program start with the following directives:

```
.section .data

.section .text

_start: .global _start
```

whereby the special symbol _start indicates that the following instruction is put to address 0x000.

To define variables, the directives ‘.word’ and ‘.fill’ are essential. The first reserves 1 data word (16 bit) of memory and it is possible to assign an initial value to that memory location. ‘.fill’ reserves a larger memory blocks, e.g. for arrays.

```
Const1:  .word  0x001       # defines variable const1 with initial value 1
arrayX:  .fill 8, 2, 0x005  # reserves 8 words, each word 2 Byte wide,
                            # all filled with value 5
```

To define a constant (e.g. the address of an external component), it is possible to use the ‘.set’

directive. Please note that especially for addresses, it is necessary to express these as a byte-based address (as the assembler internally represents all addresses like this). Thus, to read from address 0x0801, write

```
.set EXTADDR, 0x0801 << 1  # shift desired address by 1 bit to get
                           # byte-based representation
....
load EXTADDR               # will be translated to opcode 0x8801
```

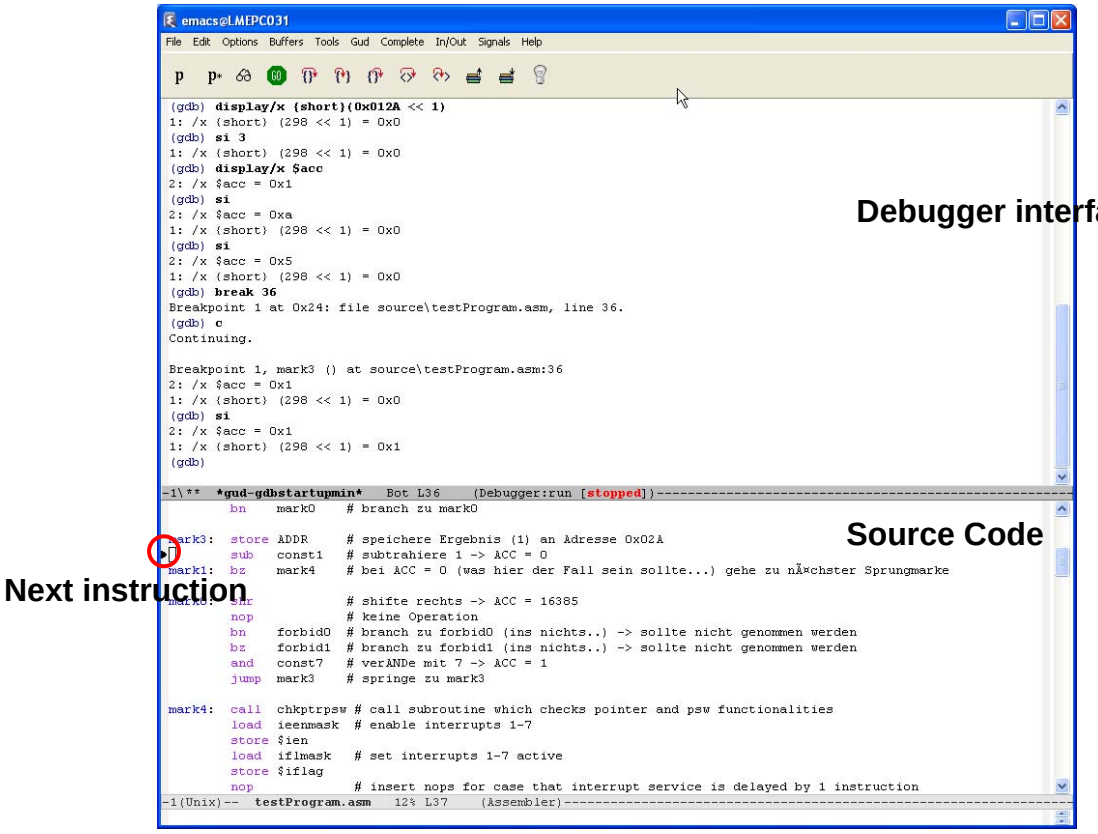
Debugger

To test the functionality of the developed UKM910 programs, the corresponding debugger may be used. To invoke the debugger, you first need to compile the program to be tested to elf format. Then, still in the UKM910Dev directory, type

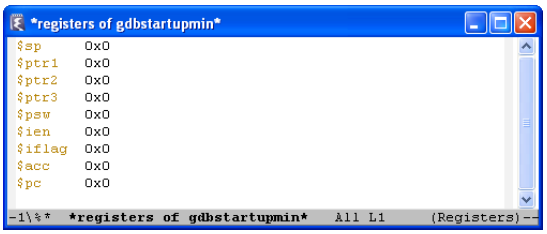
```
UKMgdb programName
```

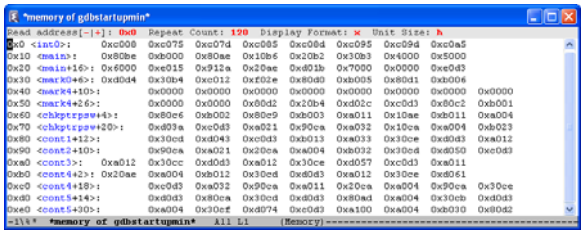
where `programName` again is the name of the assembler/elf file without file extension.

The debugger should start within Emacs.



The upper part of window can be used to enter debugging commands, whereas the lower part shows the source code of the current program, wherein the next instruction is marked with a small arrow.
Selecting Gud > GDB-Frames > Registers and Gud > GDB-Frames > Memory, it is also possible to display the current register and memory contents (please note that the addresses in the memory window are byte-based, not word-based, they are thus (address in program)*2).





Debugger Commands

The most important debugger commands are illustrated here by example:

Flow Control	
`si` `si 10`	Step Instruction: execute next instruction Step Instruction: execute next 10 instructions
`break 14` `break mark0`	Insert breakpoint at line 14 of sourcecode Insert breakpoint at mark0 (has to be a symbol defined in the sourcecode)
`delete 1`	Delete breakpoint nr. 1
`c`	Continue: execute program until next breakpoint (Please note: If you execute Continue without a breakpoint defined, the program will get into a infinite loop as it keeps on executing commands)
`jump mark1`	Jump to position mark1 (has to be a symbol defined in the sourcecode) and continue (Please note: a breakpoint needs to be defined „behind“ mark1)
Display Data	
`x/hx const1`	Display memory content at address const1 (as const1 contains the address of the variable const1, this command shows the content of this variable). The switches /hx mean: Display **h**alf-word (16 Bit) in he**x**adecimal format.

`p/x $acc`
`p/x {short}const2`
 	
`p/x {short}(0x0802<<1)`
	Print value of register $acc. The switch /x again mean he**x**adecimal format.
	Print value of memory at address const2. The brackets {} tell the debugger to interpret the following value as address, the term ‘short’ means that a short value (16 Bits) should be displayed.
	Print value of memory at address (0x0802<<1). As the value of the addresses is interpreted byte-based, one needs to shift addresses by one bit to display the intendedn memory cell.

`display/x $acc`	Display value of register $acc after each step/continue command. Any expression that can be used with the ‘print’ command can also be used here.
Modify Data	

`set $iflag = 2`
`set {short}const2 = 4`
`set {char}(0x100<<1) = 0x12`
`set {char}((0x100<<1)+1) = 0x34`
	Set value of register $iflag to 2.
	Set value of memory address const2 to 4.
	Write byte 0x12 to 1st byte of memory address 0x0100.
	Write byte 0x34 to 2nd byte of memory address 0x0100.

Session Control	
`quit`	Exit the debugger.

Because the Debugger internally works only with 8-bit bytes, and our memory is addressed as 16-bit words, writing a 16-bit value to a memory address takes two steps in the debugger.

Example:
To write the value 0x1234 to Address 0x100 (assuming it currently holds the value 0), we first write the 1st byte:

```
set {char} (0x100<<1) = 0x12
```

To check what was written, we can use the p/x command:

```
p/x {short} (0x100<<1)                    # prints: 0x1200
```

Then we write the 2nd byte, by incrementing the address by 1:

```
set {char} ((0x100<<1) + 1) = 0x34
p/x {short} (0x100<<1)                    # prints: 0x1234
```