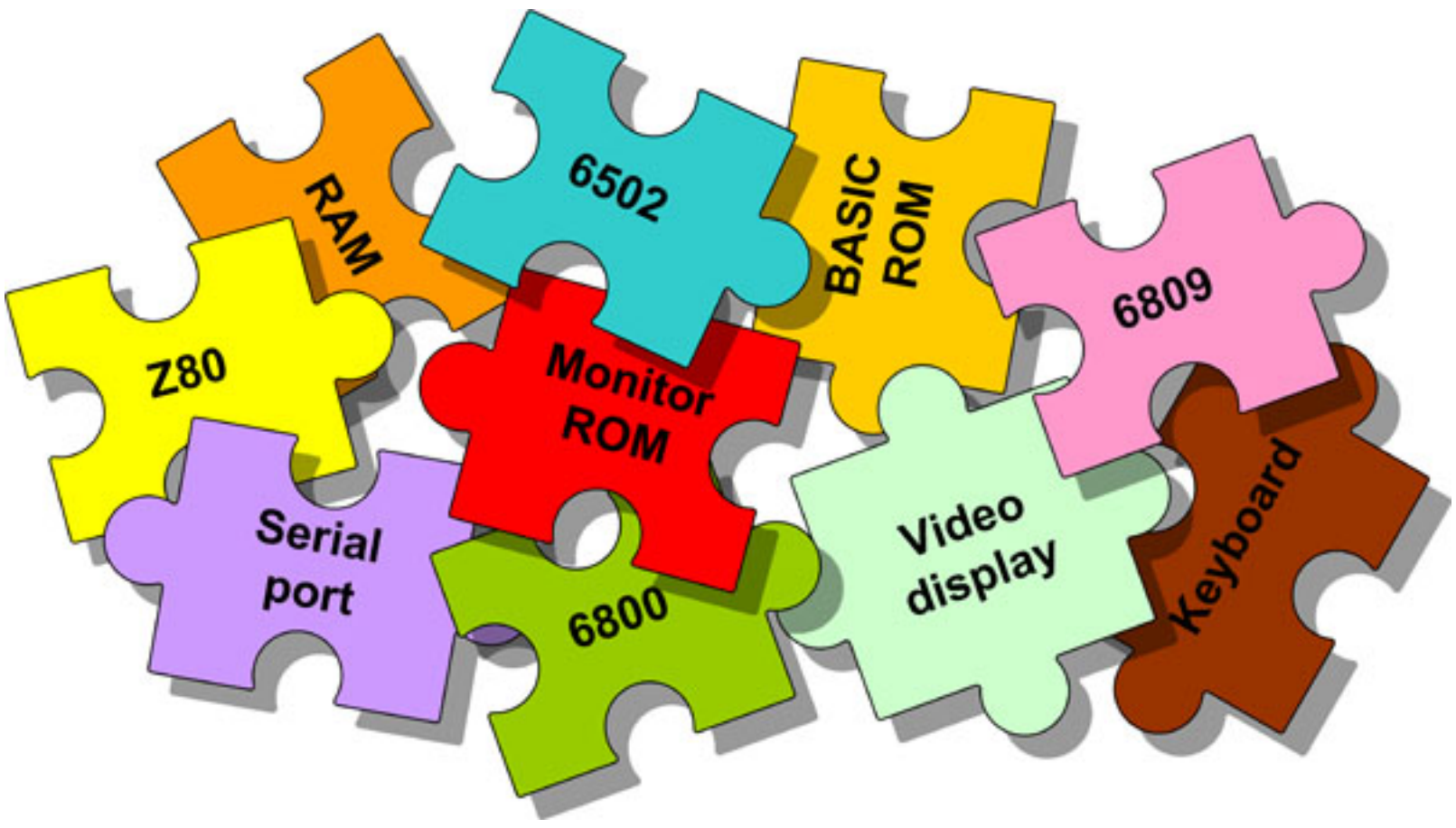


Pick-and-mix to create your own custom computer on a low-cost FPGA board

(BASIC, Z80, 6502, 6809, internal/external RAM, serial/keyboard+monitor, SD-card for CP/M or other storage)

...step by step design and implementation

PAGE STILL BEING UPDATED REGULARLY
Please come back again (and REFRESH the browser page) to see if there are updates.
IMPORTANT: Due to code updates, please make sure you have latest files before using this page.

VERIFIED WORKING
RE-BUILT FROM INFO PROVIDED HERE

by Grant Searle

For news and updates, follow me on Twitter:

Last update: 26th September 2014 9:30am BST

Major update history (minor updates not included here):
6th February 2014 - Initial release of this page.
7th February 2014 - ANSI cursor and screen colour codes and examples added below. Implemented BASIC keywords for the 6502 and 6809 BASIC below.
7th February 2014 PM - Change n_int to n_int1, added SD card interface VHDL, added SD card clock. Updated SD and top level VHDL in ZIP file. PLEASE GET LATEST ZIP TO USE THIS PAGE.
8th February 2014 AM - Z80 BASIC no longer needs interrupts (due to buffering done in my serial VHDL). Replaced Z80 BASIC ROM files in ZIP and changed "n_int" connection in the Z80 CPU VHDL below. PLEASE GET LATEST ZIP TO USE THIS PAGE.
9th February 2012 PM - Video out "SBCTextDisplayRGB.vhd" now ignores code $07 (bell). ZIP updated. No other change required.
10th February 2012 PM - De-glitcher added to serial input to prevent occasional invalid chars read - now 100% reliable. Needs a "clk" passed to the "bufferedUART". Serial interface VHDL below and ZIP updated. "SBCTextDisplayRGB.vhd" - ANSI top-left changed to (1,1) instead of (0,0).
11th February 2014 - Inverse and bold ANSI atts now supported. Added option to change default character attribute in display VHDL below.
12th February 2014 - ANSI "graphics rendition" now allows up to 8 parameters. ZIP updated, no VHDL change needed.
14th February 2014 - Various changes to reduce FPGA resources used in the serial and monitor/kb code. Buffer sizes reduced and ANSI "graphics rendition" reduced to 4 parameters. This has freed up approx 450 logic elements. KB clock filter changed to match my implementation in the serial port. KB clock write phase changed (thanks to David Bell for his analysis and feedback - appreciated!)
15th February 2014 - ANSI handling of delete line and insert line added. SBCTextDisplayRGB.vhd, DisplayRam1K and DisplayRam2K needed updates to handle this. Z80 BASIC keywords added to this page
17th February 2014 - CP/M implementation and all files provided here.
3rd March 2014 - Board schematic added.
26th September 2014 - 6809 CPU core updated using a new version provided by the author, John Kent. Many thanks to John for his support for this core. ***Please get the latest files here. Important: the 6809 include code is modified slightly to use the updated core - ensure you use the slightly VHDL updated text shown later on this page when using the new cpu09l.vhd file. If using an existing project then extract the new file and put it in the M6809 folder, then remove the previous 6809 file. You will then need to include the new file in your project within Quartus. The zip already contains all changes needed to the project file.***

Index

Introduction

On my other pages, I have information on how to construct simple 6502, 6809 or Z80 designs using the minimal amount of discrete components. This page allows you to take each of those designs and many other configurations and build them within an FPGA on a very low-cost board.

I will take you through the decisions you need to make and all the coding that you need to do to build the computer to your own specification. You choose the processor, whether you want a display/keyboard or just serial ports, how much RAM you want and what speed you want it to run at. All VHDL is provided so you can change the individual components to exactly match your requirements. Design will take the form of building on a VHDL breadboard (skeleton code and signals already provided for you) which will then have the component VHDL added within it to complete the design.

The smallest configurations will all fit within the one FPGA chip. Larger memory configurations only require a single SRAM chip to be added.

To connect a monitor/TV and keyboard very few components are needed - just a few resistors and connectors.

As a result, a complete and very fast computer can be built in a very small enclosure if needed.

The final cost of this is surprisingly low (**no need to buy the very expensive project boards**) - a very low cost board is easily available that has most of what is required - very few additional (low cost) parts are needed.

This page is not intended as an introduction to VHDL - there are many pages already explaining that, but this will take you through the complete design and implementation. You can, or course, just take the VHDL project that I created and use it immediately - but it's important to understand what I did so that you can do the same yourself.

Important note: FPGA I/O pins normally have 3.3V levels, NOT 5V, so be careful when interfacing to TTL and CMOS parts.

The board and keyboard takes less than 100mA so could be powered from batteries if needed.

The board that is used in this article measures only 72mm x 51mm (2 7/8" x 2") so a very small case can be used if needed.

The board is available from many sources, eBay suppliers probably the easiest and cheapest (search for EP2C5T144C8N mini board) .

Acknowledgements

Before I continue, I feel it is important to acknowledge all other's work that I have used.

Z80 CPU core - written by Daniel Wallner and is on the "opencores" website
6502 CPU core - written by Daniel Wallner and is on the "opencores" website, bugfixes by "Mike J" and "ehenciak"

6809 CPU core - written by John E. Kent and is on the "opencores" website
6800 CPU core (not used YET) - written by John E. Kent
SD card interface - my own adaptation from partial code originally published by Steven J Merrifield
BASIC interpreter for the Z80, 6502 and 6809 - Microsoft corporation (modifications and interface code all my own work)
CGA Font ROM - IBM
Quartus II software and **memory IP** components - Altera

ALL other diagrams, text, software and VHDL are MY OWN WORK

Personal acknowledgements for feedback and encouragement to Graeme and also to David Bell. Your support has been greatly appreciated.

Tools needed

FPGA development software - Altera Quartus II 13 sp1 (**Do <u>not</u> use newer versions** - Cyclone II FPGAs are not supported with newer versions)
FPGA USB programmer (USB-blaster or similar)
Low-cost Altera FPGA board (EP2C5T144C8N Cyclone II FPGA on a mini board although any larger board/FPGA can be used)

Additional components
(See the wiring diagram below for further details)

PS2 keyboard:
2x 10K resistors
PS/2 connector
PC keyboard with a PS/2 (round) connector

Monochrome video out:
RCA ("phono") connector for the
1K resistor
470R resistor

RGB video VGA monitor out:
3 x 680R resistors
3 x 470R resistors
15 pin VGA socket

RGB video SCART TV monitor out:
3 x 680R resistors
4 x 470R resistors
1 x 100R resistor
21 pin SCART socket

Serial interface:
3.3V USB to serial board (or serial voltage converter if connecting to a serial port on another terminal/pc)

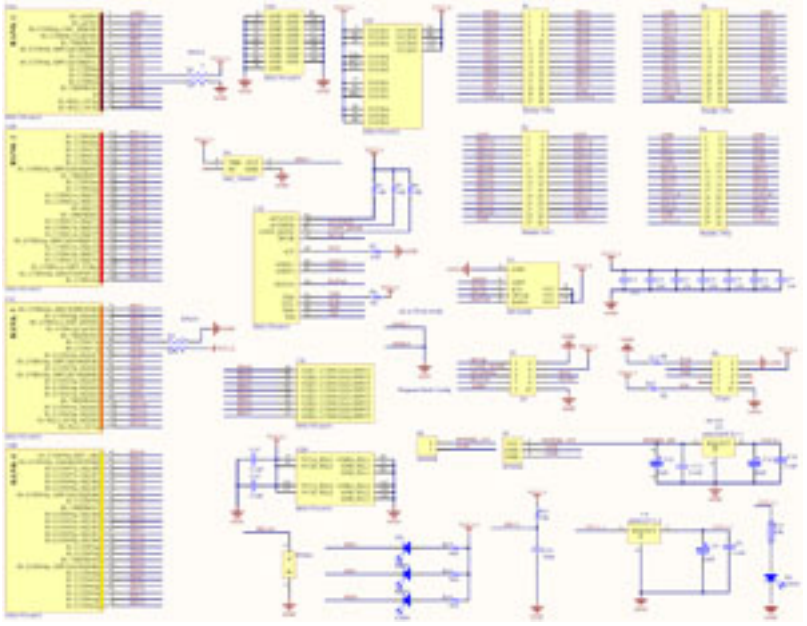
SD card interface:
Suitable SD or microSD socket
2GB SD Card (NOT SDHC at the moment)

5V Power supply with suitable connector for the board
Breadboard/PCB
Wires with ends that can plug onto the pins on the board

Important notes regarding the board being used

A schematic is available but was very spread out. I have compressed it to an image that can still be read when printed on A4.

This schematic of the board is available HERE (right-click and "Save as").



As you will see on the schematic, some FPGA pins on the board have connections made, so need to be accounted for when doing this or other designs on it...

To summarise the pins that are available on the connectors that need consideration...

Pin 3 - LED 1 (D2 on board) - low to light
Pin 7 - LED 2 (D4 on board) - low to light
Pin 9 - LED 3 (D5 on board) - low to light
Pin 144 - pushbutton to ground, no external pullup. Set internal pullup on FPGA configuration if used. I will use for UK101 "reset"
Pin 17 - 50MHz clock input
Pin 73 - 10uF capacitor to ground, 10K resistor to Vcc, for power up reset if needed?
Pin 26 - Connected to Vcc 1.2V - only needed for EPC28 though, so the "zero ohm" resistor could be removed and the pin used as normal.
Pin 27 - Connected to GND - only needed for EPC28 though, so the "zero ohm" resistor could be removed and the pin used as normal.
Pin 80 - Connected to GND - only needed for EPC28 though, so the "zero ohm" resistor could be removed and the pin used as normal.
Pin 81 - Connected to Vcc 1.2V - only needed for EPC28 though, so the "zero ohm" resistor could be removed and the pin used as normal.

Specifications

6502, 6809 or Z80 processor running at up to 25MHz

Microsoft BASIC in ROM (copyright of the original respectfully acknowledged)

Serial ports (can support several) up to 115200 baud

PC monitor or TV (PAL/NTSC) with fully configurable 40x25, 80x25, 68x30 (or many other formats) monochrome or 16 colour display. Character sizes also configurable. Standard ANSI interpreter built-in to control cursor and colours.

PS2 (standard PC keyboard with round plug) keyboard with LED support.

SD card interface for CP/M or other use (simple interface for use within BASIC or machine code). Not SDHC currently, but could be done.

Planning the computer

This computer is not based on any existing design, so the final configuration is entirely up to you (I will provide a couple of ready-made examples to get you started, though, if needed).

The following questions need to be answered before proceeding.

1. CPU type - options are:
6502
Z80
6809

2. Interface - pick-and-mix options are:
Mono ("black and white") video and a PS2 keyboard
RGB ("colour") video (can be TV RGB/SCART or PC "VGA") and a PS2 keyboard
Serial RS232 compatible

There is an option of a second interface which can be an additional serial port (actually able to add several serial ports if needed).

Note: The supplied BASIC will use the first interface ONLY when performing I/O. To access the second interface, you can use PEEK/POKE (6502 or 6809) or IN/OUT (Z80) to read or write to it. The display/keboard and serial ports use an identical interface fully software compatible with the 6850 ACIA. You can also change the ROM code as needed (source supplied). Alternatively, if using the PS2 keyboard interface, you can use one of the supplied "FN Key" output signals to toggle the active input/output between interface 1 and interface 2. That way, you can use keyboard/monitor as the first interface and a serial connection as the second interface then switch interfaces to load/save programs via the serial port. This option is straightforward and will be published at a later date.

3. Memory:
Internal RAM - 1K up to 4K (no external SRAM chip needed)
External RAM - up to full 64K address space (not all accessible due to ROM taking part of the address). Can use 2KB, 8KB, 32KB, 128KB(only 64KB used) external SRAM.

Once decided, you need to see if what you have chosen will fit into the chip.

The main constraints will be memory space on the chip. Using a TV display requires use of memory within the FPGA for the display RAM, colour attributes RAM (if colour per character chosen) and the character definitions ROM (128 or 256 characters).

The FPGA on this board holds 26 x 0.5Kbyte memory blocks (ie. 13K total).
The memory requirements of the component blocks are as follows:

BASIC ROM for each of the 6502, 6809 or Z80 - 8K

ROM/RAM for mono 80x25 display - 5K (2K RAM, 2K Character ROM)
ROM/RAM for mono 80x25 display, reduced character set - 3K (2K RAM, 1K Character ROM)
ROM/RAM for colour 80x25 display - 6K (4K RAM, 2K Character ROM) *
ROM/RAM for colour 80x25 display, reduced character set - 5K (4K RAM, 1K Character ROM)
ROM/RAM for colour 40x25 display - 4K (2K RAM, 2K Character ROM)
ROM/RAM for colour 40x25 display, reduced character set - 3K (2K RAM, 1K Character ROM)

ie.
Character RAM will be either 1K or 2K depending on the number of characters that you choose to display.
Colour attribute RAM will be the same size as the character RAM if you require colour specified for each character, otherwise it will not be used/required.
Character ROM will be either 1K or 2K depending on whether you need the standard (128) character set or extended DOS (256) character set.

So, a 6502 with mono 80x25 display, reduced character set would require 8+3=11K, leaving 2K RAM/ROM spare. Therefore you could also have 2K RAM internally.
* - The colour 80x25 display, full character set can't be used unless a smaller ROM is used (eg. tiny BASIC or a monitor/bootloader for CP/M etc).

Having checked what is required can actually fit, the build can proceed. If not sure, build a smaller configuration first then alter as needed.

Stuck on what to choose?

There are so many possibilities that you may not know what needs to be picked to start.

Here are some **suggestions** for the smaller configurations to get you started

No external SRAM chip

Monochrome video and PS2 keyboard, 80x25 text display, 2K internal RAM, 6502 processor

Serial connection only, 4K internal RAM, Z80 processor

Colour video, VGA monitor, 40x25 display, 2K internal RAM, 6809 processor

External SRAM chip

Colour video, VGA monitor, 80x25 display, 56K external RAM, Z80 processor

Pick your configuration then proceed through the rest of the page using that configuration. When you reach the end you should have a working computer.

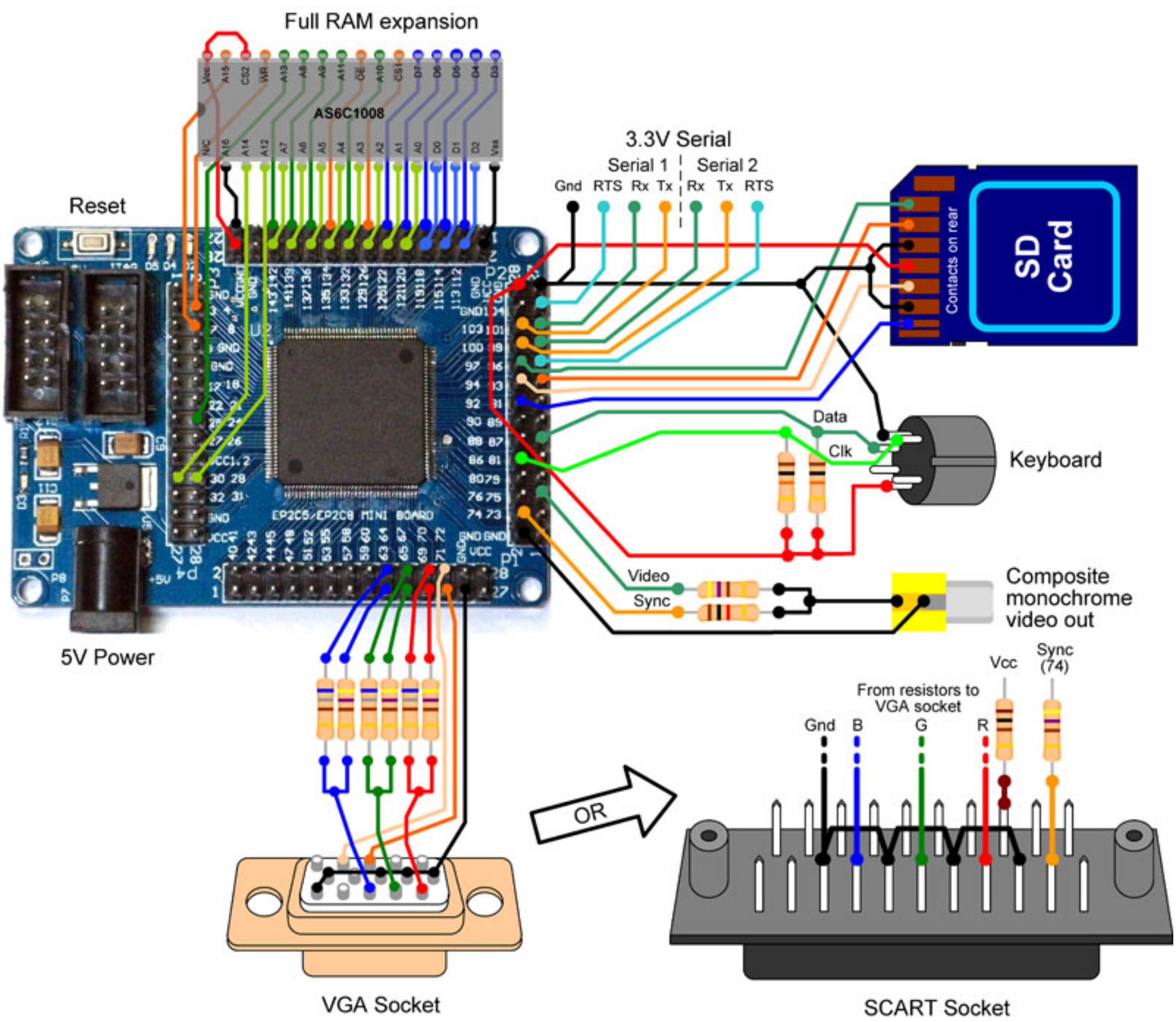
The build

Building this machine is partly electronic and partly software (VHDL).

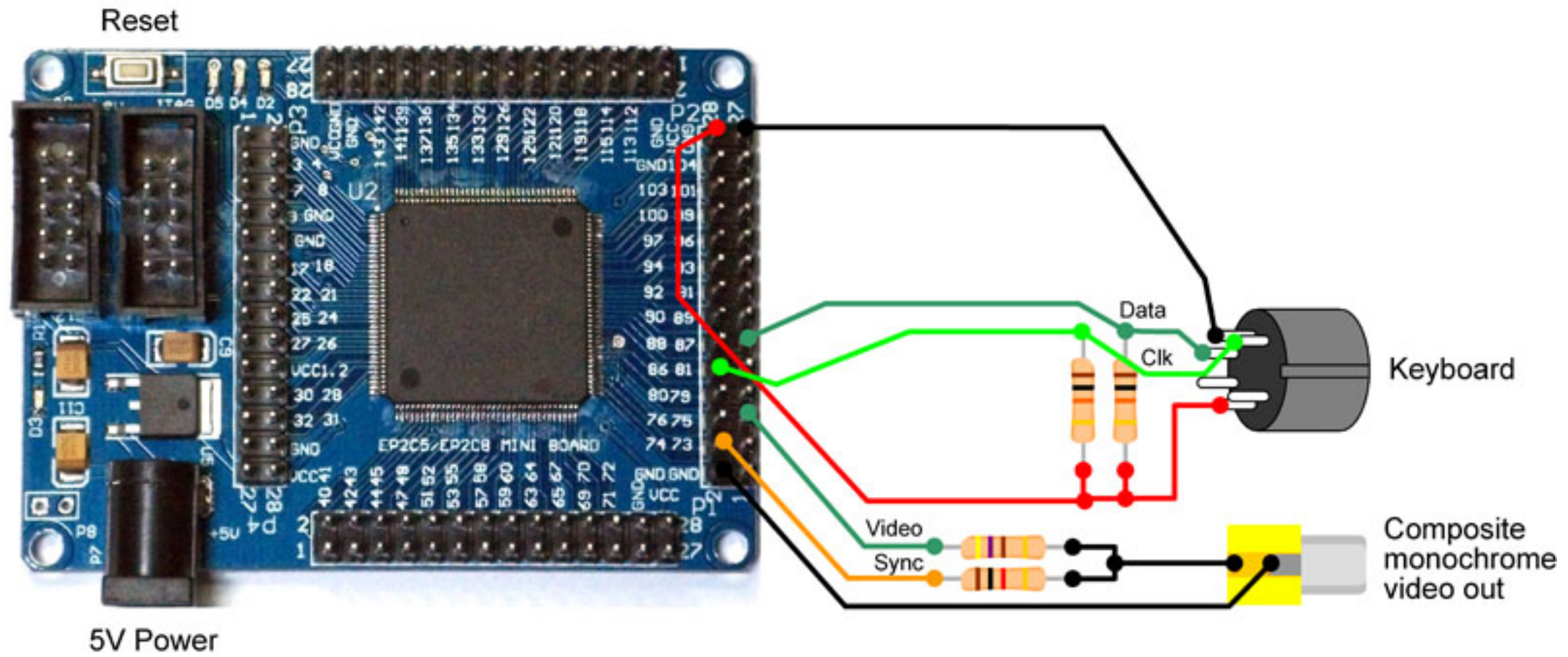
Hardware wiring

The electronic part depends on how much functionality you want to provide.

With ALL options installed and using the connections that I have defined in the files, the complete expanded build would be connected as follows:



However, if you want a simpler machine (eg. internal RAM, monochrome TV monitor output and a PS2 keyboard) then the connections would be reduced to...

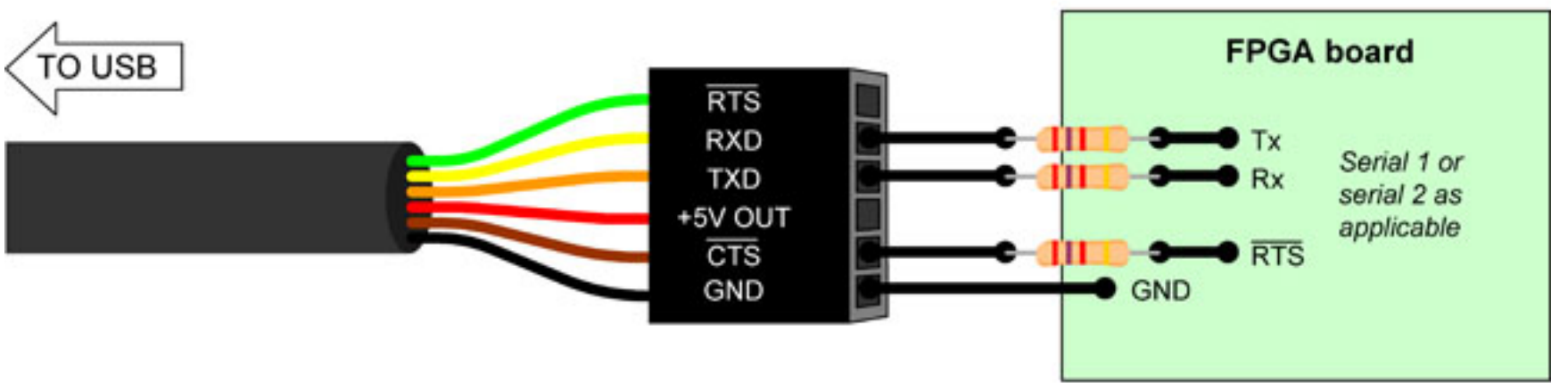


ALL configurations use the same pinout, so you could start with a simpler configuration then upgrade as needed.

Suggested serial connection

Serial 1 and Serial 2 can be connected to a USB to TTL (3.3V or 5V) serial cable as shown below.

Using a USB to 3.3/5V (TTL) serial cable instead of an RS232 interface



Resistors are present to safely limit current along the cable if the USB or computer is not powered.

By using a USB to serial cable the board can also be powered from the USB port down the same cable (a powered hub could be used if higher current needed), eliminating the need for a separate power supply for this board. Just connect the +5V out on the cable to the centre pin of the 5V input supply on the board.

The 2k7 resistors are present in the diagram because the board may be powered but not plugged in to the USB cable, or the USB cable could be plugged in and active without power to the board. These limit the current to avoid power being drawn through the interface pins. If always powering the board via the cable then no resistors are necessary.

VHDL "wiring"

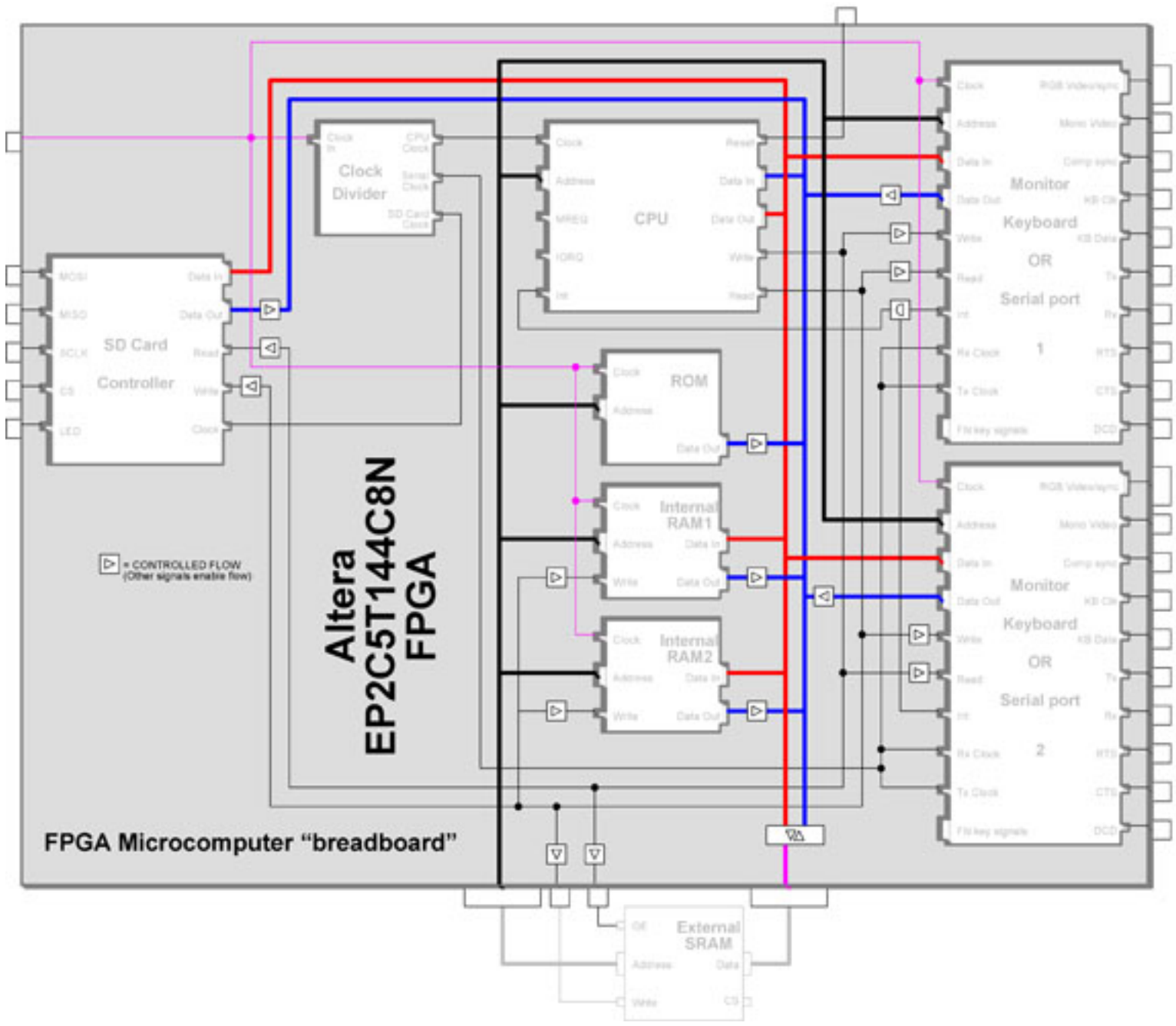
The only file that you are required to edit is "Microcomputer.vhd" which I have provided in the set of project files. Open the Microcomputer.qpf project file in Quartus II, then double-click the "Microcomputer" entity you see in the "project explorer" window. This will open the "Microcomputer.vhd" file in the editor window. You then follow my instructions below to complete the VHDL for your project.

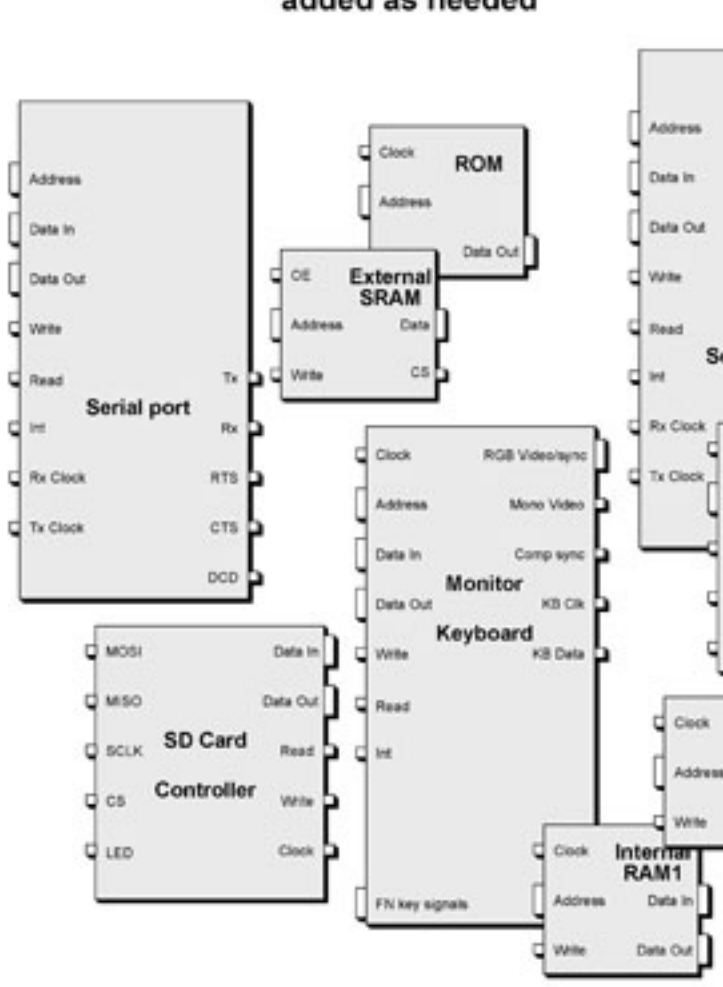
IMPORTANT: Ensure you follow the complete page as there are may be several entries needed to incorporate a component into the design. If you have a problem, please re-read the page carefully to ensure you haven't missed out any steps.

The bare breadboard VHDL is used and the relevant component VHDL added to it. All components that can be used have been added to the project file, so only changes to the top level VHDL is needed.

The bare VHDL (the "breadboard)

The following diagram illustrates the concept involved here.





The architecture of a VHDL projects consists of the "top level" code and components that can be added to it.

For this set of projects, all will use the same top level (effectively a bare circuit board with some connections already in place to plug in to).

The structure is as follows:
library definitions
entity Microcomputer is
port(
signals that are to be connected to actual pins on the chip
);
end Microcomputer;
architecture struct of Microcomputer is
signals ("wires") used within the system that are used to interconnect the components
begin
the VHDL definition of logic operations (glue etc) and connections to components
the components being used are also included in here and attached to the appropriate "wires"
end;

The VHDL used as the base (breadboard) is shown here.

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity Microcomputer is
port(

:
SIGNALS THAT CONNECT EXTERNALLY ARE ALREADY DEFINED HERE
:

);
end Microcomputer;

architecture struct of Microcomputer is

:
VARIOUS PREDEFINED BREADBOARD SIGNALS HERE
:

begin
-- ---------------------------------------------------------------------------
-- CPU CHOICE GOES HERE

-- ---------------------------------------------------------------------------
-- ROM GOES HERE
```

```
-- _____________________________________________________________________
-- RAM GOES HERE

-- _____________________________________________________________________
-- INPUT/OUTPUT DEVICES GO HERE

-- _____________________________________________________________________
-- MEMORY READ/WRITE LOGIC GOES HERE

-- _____________________________________________________________________
-- CHIP SELECTS GO HERE

-- _____________________________________________________________________
-- BUS ISOLATION GOES HERE

-- _____________________________________________________________________
-- SYSTEM CLOCKS GO HERE

end;
```

As you can see when you look at the file, the common wires used within a microcomputer system have been defined and are ready to be connected to. During construction, not all may be used. The unused ones can be removed if needed (don't have to) once the construction is complete.

Every microcomputer system has the following:
A clock to drive the system components (cpu and interface)
CPU - the main processor, such as a Z80, 6502, 6809 etc.
RAM - volatile memory for user programs and data
ROM - non-volatile holding I/O routines and a monitor and/or language code
Inputs - could be switches, pins, keyboard, serial etc.
Outputs - could be LEDs, pins, video (monitor), serial etc.

Looking at the breadboard VHDL, you will see there are sections to be completed during the construction to include the relevant components to make up the complete system.

I have already included ALL components in the VHDL project, so once open in Quartus II, you just add the relevant components into the above code. Once complete, compile and upload to the FPGA and you should (if all done correctly) have a working computer ready to use.

Adding the blocks

Each component is defined and attached in the following way.

1. Include the required component VHDL file within the project
2. Create a VHDL entry within the top-level code, defining the "instance" name of the component, the actual VHDL to use (as imported) and then map the signals to the I/O signals defined in the component.

```
instanceName : entity work.VHDLName
port map(
ComponentSignal1 => Signal1DefinedInTopLevelVHDL,
ComponentSignal2 => Signal2DefinedInTopLevelVHDL,
:
ComponentSignalN => SignalNDefinedInTopLevelVHDL);
```

Decide on the central processor (CPU) that you want to use and add it.

This needs to be decided first because it will decide which ROMs and interface connections are needed.

These are all existing "components" that are available on the net. No particular standard was used as they were all created independently so there is a difference to the naming of the signals that are internal to them and some have active high and some have active low connections. However, as I will show here, all can be connected to our existing breadboard connections by taking the appropriate component signal to the breadboard signal.

The CPUs on offer here are typical and share the following characteristics:
A 16-bit address bus. For full access to the address space, all 16 connections are made (A0...A15). The appropriate address signals are therefore connected to the address bus already set on the breadboard.
eg.

```
a => cpuAddress,
```

Similarly, all these CPUs have an 8-bit data bus. For the discrete components, the data bus is bidirectional. This could have been implemented in VHDL as bidirectional to match. However, the VHDL components (including RAM and ROM) tend to have separate incoming and outgoing data buses - this makes it easier to interface. Therefore the 8-bit data bus found on a CPU is represented as two separate data buses and are connected to the two data buses on the breadboard.
eg

```
di => cpuDataIn,
do => cpuDataOut
```

Additionally, CPUs need signals to identify to memory and peripherals whether read or writes are active (plus the Z80 has the ability to have memory access or IO access). These additional signals are again connected to the wires already set out on the breadboard VHDL.
eg.
n_WR - "not write" - "0" when a write is active (for CPUs other than the Z80, a "1" on here implies a read is active)
n_RD - "not read" (Z80 only) - "0" when a write is active
n_IORQ (Z80 only)
n_MEMRQ (Z80 only)
n_INT1 - this is an input to the CPU to inform it that a peripheral will request to interrupt the current process ("0" - interrupt request activated)
Other inputs to the processors that are not used are connected to the appropriate logic levels (=> '1' (high) or => '0' (low)) to allow them to work.

You have three choices. Pick one of these and copy and paste the relevant code into the above where it says "CPU CHOICE GOES HERE".

The choices are:

6502	Z80	6809 *(updated than this then pl*
cpu1 : entity work.T65 port map(Enable => '1', Mode => "00", Res_n => n_reset, Clk => cpuClock, Rdy => '1', Abort_n => '1', IRQ_n => '1', NMI_n => '1', SO_n => '1', R_W_n => n_WR, A(15 downto 0) => cpuAddress, DI => cpuDataIn, DO => cpuDataOut);	cpu1 : entity work.t80s generic map(mode => 1, t2write => 1, iowait => 0) port map(reset_n => n_reset, clk_n => cpuClock, wait_n => '1', int_n => '1', nmi_n => '1', busrq_n => '1', mreq_n => n_MREQ, iorq_n => n_IORQ, rd_n => n_RD, wr_n => n_WR, a => cpuAddress, di => cpuDataIn, do => cpuDataOut);	cpu1 : entity work port map(clk => not(cpuClo rst => not n_reset rw => n_WR, addr => cpuAddre data_in => cpuDa data_out => cpuD halt => '0', hold => '0', irq => '0', firq => '0', nmi => '0');

The copied text has defined an instance of a CPU component and assigned all the necessary signals to the connections on the breadboard VHDL. This has inserted the CPU into the design - easy enough!

ROMs

I have already prepared ROM code for the Z80, 6502 and 6809 designs and is virtually the same as you will see on my other "build your own computer" pages. The VHDL version directly replicates the discrete component version of the design (apart from moving the ACIA address) so the ROM code can run virtually unchanged (no changes apart from that ACIA address change) within this FPGA. The 6809 Extended BASIC ROM has been reduced to fit into an 8K ROM by removing the "PRINT USING" routine.

ROMs consist of the following signals:
address bus - to specify which location in the ROM will be accessed
q - data OUT from the ROM
Additionally, because the VHDL is a synchronous device, there is normally a clock input to allow the data to be registered to the output

Depending on which processor you have chosen will depend on which ROM is needed.

The ROMs can either use the built-in block memory in the FPGA or can be defined as an "array" which is then synthesized by the VHDL compiler into a complex combinational logic model. I am using the standard memory model ("M4K") available within the Altera suite. If using another manufacturer's FPGA then there would be an equivalent model. The HEX file that would be loaded into the memory block during compilation is included.

Copy and paste the appropriate ROMs into the breadboard VHDL where it says "ROM GOES HERE".

6502	Z80	6809
rom1 : entity work.M6502_BASIC_ROM -- 8KB BASIC port map(address => cpuAddress(12 downto 0), clock => clk, q => basRomData);	rom1 : entity work.Z80_BASIC_ROM -- 8KB BASIC port map(address => cpuAddress(12 downto 0), clock => clk, q => basRomData);	rom1 : entity work port map(address => cpuA clock => clk, q => basRomDat);

The above code will "attach" the ROM address to the bottom 13 bits (A0..A12) of the CPU address connections to allow the CPU to address all 8K (2^13 = 8192). The remainig 3 "top" bits of the address bus (A13..A15) must then be used in the "chip

select" (address decoder) logic to ensure the ROM will only appear in one location in the memory space. For the Z80, the ROM needs to appear in the bottom 8K, so A13..A15 will be "000". For the 6502 or 6809 the ROM needs to appear in the top 8K, so A13..A15 will be "111". You will see this address decoding in the "chip select" VHDL code later on this page.

The "q" data output is shown connected to the "basRomData" lines - this consists of 8 connections and will be connected to the CPU data input only when the ROM is being accessed (only one set of outputs can go to any input at the same time) ie. "connect the basRomData to the cpu data input if the rom is selected". That process is handled by the "bus isolation" lower on this page.

Program RAM

The choice is to either use internal RAM (1K to 4K available) or use an external RAM chip (56K available). If this is your first design, I recommend you start without connecting the external RAM chip and get everything working internally, so pick the internal RAM option.

If you require internal RAM

I have provided 1K, 2K and 4K memory entities. Choose the one that would fit. (it is also possible to have one 1K and one 2K to make 3K but that won't be covered here). Different size memories require a different amount of address lines, so the entity for each size RAM utilises the required number of address lines.

	6502, 6809 or Z80
1K *Note:6809 BASIC won't work with just 1K RAM*	ram1: entity work.InternalRam1K port map (address => cpuAddress(9 downto 0), clock => clk, data => cpuDataOut, wren => not(n_memWR or n_internalRam1CS), q => internalRam1DataOut);
2K	ram1: entity work.InternalRam2K port map (address => cpuAddress(10 downto 0), clock => clk, data => cpuDataOut, wren => not(n_memWR or n_internalRam1CS), q => internalRam1DataOut);
4K	ram1: entity work.InternalRam4K port map (address => cpuAddress(11 downto 0), clock => clk, data => cpuDataOut, wren => not(n_memWR or n_internalRam1CS), q => internalRam1DataOut);

Paste the appropriate code where it says "RAM GOES HERE"

If you require external SRAM

The external RAM can be 2K, 8K, 32K or 128K (only 64K is used). The appropriate RAM pins are connected to the matching connections as defined on my wiring diargram earlier. Unused addresses on the FPGA are left disconnected. Unused addresses on the RAM chip are connected to Gnd or Vcc.

The chosen size of RAM will affect the chip select code that is needed. Explanation and implementation details for the chip select is shown further down this page..

Instead of providing an internal component, what is needed is to assign the external signals (connected to the pins on the RAM chip and the FPGA) to the internal signals on the VHDL. This is done as follows.

Paste the following appropriate code where it says "RAM GOES HERE":

6502 or 6809	Z80

sramAddress(15 downto 0) <= cpuAddress(15 downto 0);
sramData <= cpuDataOut when n_WR='0' else (others => 'Z');
n_sRamWE <= n_memWR;
n_sRamOE <= n_memRD;
n_sRamCS <= n_externalRamCS;
	sramAddress(15 downto 0) <= cpuAddress(15 downto 0);
	sramData <= cpuDataOut when n_memWR='0' else (others => 'Z');
	n_sRamWE <= n_memWR or n_externalRamCS;
	n_sRamOE <= n_memRD or n_externalRamCS;
	n_sRamCS <= n_externalRamCS;

Note that the sramData bus is bidirectional and only outputs data when write is active (n_WR='0') otherwise it is set to an input so that the CPU can read from it (ie. (others => 'Z')).

Interfaces

I have deliberately designed the TV/monitor and keyboard interface to appear to the host software as if it was a 6850 serial port (write sends to the display, read gets the keyboard data, status request determines if the receive (key pressed) or send (display) is ready). As a result, these components are completely interchangeable which makes this design very flexible. The BASIC is already set up to use a 6850 ACIA compatible serial port so the monitor/keyboard can replace the serial port with NO SOFTWARE CHANGE.

Using the current configuration and pinout assignments you can have:

1 Serial port
2 Serial ports
1 TV/monitor and PS2 keyboard
1 TV/monitor and PS2 keyboard plus 1 serial port
other: there are several spare pins on the FPGA so several more serial ports are easily possible if needed

Decide on whether a serial port or monitor/keyboard is to be used and copy one of the following into the code where it says "INPUT/OUTPUT DEVICES GO HERE"

If you are to use two I/O devices (eg. two serial or one serial and one TV/keyboard) then repeat the below process, but for the second device, changing "io1" to "io2", "interface1DataOut" to "interface2DataOut", "rxd => rxd1" to "rxd => rxd2", "txd => txd1" to "txd => txd2" etc. NOTE: ONLY ONE MONITOR/KB INTERFACE CAN BE SUPPORTED WITH THIS BOARD. However, you can have several serial interfaces of one monitor/keyboard with additional serial interfaces.

Using a serial port (connected to a PC) as the output and input

The following definition needs to be copied and pasted (pick the one appropriate to the processor being used).

6502 or 6809

```
io1 : entity work.bufferedUART
port map(
clk => clk,
n_wr => n_interface1CS or cpuClock or n_WR,
n_rd => n_interface1CS or cpuClock or (not n_WR),
n_int => n_int1,
regSel => cpuAddress(0),
dataIn => cpuDataOut,
dataOut => interface1DataOut,
rxClock => serialClock,
txClock => serialClock,
rxd => rxd1,
txd => txd1,
n_cts => '0',
n_dcd => '0',
n_rts => rts1
);
```

Z80

```
io1 : entity work.bufferedUART
port map(
clk => clk,
n_wr => n_interface1CS or n_ioWR,
n_rd => n_interface1CS or n_ioRD,
n_int => n_int1,
regSel => cpuAddress(0),
dataIn => cpuDataOut,
dataOut => interface1DataOut,
rxClock => serialClock,
txClock => serialClock,
rxd => rxd1,
txd => txd1,
n_cts => '0',
n_dcd => '0',
n_rts => rts1
);
```

The "not read" and "not write" require slightly different connections when using the Z80, as can be seen above due to it appearing in "memory space" on the 6502 or 6809 architecture but appears in "IO space" in the Z80 architecture.

Data in connects to the CPU data out, same as for other memory/devices in this machine.

The data output is shown connected to the "interface1DataOut" (or "interface1DataOut" for the second instance if used) lines - this consists of 8 connections and will be connected to the CPU data input only when the interface is being accessed (only one set of outputs can go to any input at the same time) ie. "connect the interface1DataOut to the cpu data input if the interface1 is selected". That process is handled by the "bus isolation" lower on this page.

Using a TV/monitor and a PS2 keyboard as the output and input.

(This board running on a standard PAL television, full colour mode, 80x25 characters)

I have designed the TV display to be configurable to just about whatever you require of it. The "default" configuration is 80x25 chars on a standard VGA (640x480) display. This utilises 25 of the 30 lines in this mode, centred on the display. This, however, can be changed.

Video format options :

VGA timings (various modes) - 640x480, 60Hz default
PAL (50Hz) timings
NTSC (50Hz) timings

RGB 16 colour output (with single colour on composite video pin if needed)
Composite black and white output (with single colour also on RGB pins if needed)

To add a monitor/keyboard into the design, the following needs to be included:

Note the different connections for Z80 or 6502/6809. Comment/delete the 6502 lines and uncomment the Z80 lines if using a Z80.
If RGB or monochrome signals are not to be used then those lines not required are to be deleted.

The following definition needs to be copied and pasted.

6502 or 6809	Z80
io1 : entity work.SBCTextDisplayRGB port map (n_reset => n_reset, clk => clk, -- RGB video signals hSync => hSync, vSync => vSync, videoR0 => videoR0, videoR1 => videoR1, videoG0 => videoG0, videoG1 => videoG1, videoB0 => videoB0, videoB1 => videoB1, -- Monochrome video signals (when using TV timings only) sync => videoSync, video => video, n_wr => n_interface1CS or cpuClock or n_WR, n_rd => n_interface1CS or cpuClock or (not n_WR), n_int => n_int1, regSel => cpuAddress(0), dataIn => cpuDataOut, dataOut => interface1DataOut, ps2Clk => ps2Clk, ps2Data => ps2Data);	io1 : entity work.SBCTextDisplayRGB port map (n_reset => n_reset, clk => clk, -- RGB video signals hSync => hSync, vSync => vSync, videoR0 => videoR0, videoR1 => videoR1, videoG0 => videoG0, videoG1 => videoG1, videoB0 => videoB0, videoB1 => videoB1, -- Monochrome video signals (when using TV timings only) sync => videoSync, video => video, n_wr => n_interface1CS or n_ioWR, n_rd => n_interface1CS or n_ioRD, n_int => n_int1, regSel => cpuAddress(0), dataIn => cpuDataOut, dataOut => interface1DataOut, ps2Clk => ps2Clk, ps2Data => ps2Data);

This is for the default 80x25 chars on a standard VGA (640x480) display. The relevant options within the display/keyboard

component are available to be changed, however. These are done by "overriding" the values internally and are done as shown below. Here are some options that you are likely to use. These are, however, fully configurable and you can create a display with whatever format you want.

Here are some other options (additional code that needs to be added to the above code is shown in **bold red**). Note - NO semi-colon before or after the generic map block:

DESCRIPTION	ADDITIONAL CODE NEEDED
68 characters per line, 30 lines on a standard VGA display requires each character to be stretched horizontally (3 clocks per pixel instead of 2), the vertical characters increased, the characters per line changed and also the top scanline where the display is to start. To achieve this, include the "default" display component then override (change) some of the parameters to match what is now needed. This is done by adding the override values as shown in bold below (remaining code is same as above). This is what makes the display so configurable. *Internal memory used: 2K Display RAM, 2K Attribute RAM, 1K Character ROM = 5K total. The ROM takes a further 8K, so no remaining internal memory. Program memory must therefore use external SRAM.*	: io1 : entity work.SBCTextDisplayRGB **generic map(** **HORIZ_CHARS => 68,** **CLOCKS_PER_PIXEL => 2,** **VERT_CHARS => 30,** **DISPLAY_TOP_SCANLINE => 35** **)** port map (:
PAL colour display 80x25 chars centred on screen *Internal memory used: 2K Display RAM, 2K Attribute RAM, 1K Character ROM = 5K total. The ROM takes a further 8K, so no remaining internal memory. Program memory must therefore use external SRAM.*	: io1 : entity work.SBCTextDisplayRGB **generic map(** **CLOCKS_PER_PIXEL => 3,** **CLOCKS_PER_SCANLINE => 3200,** **DISPLAY_TOP_SCANLINE => 65,** **VERT_SCANLINES => 312,** **VERT_PIXEL_SCANLINES => 1,** **VSYNC_SCANLINES => 4,** **HSYNC_CLOCKS => 235,** **DISPLAY_LEFT_CLOCK => 850** **)** port map (n_reset => n_reset, :
PAL colour display 40x25 chars centred on screen *Internal memory used: 1K Display RAM, 1K Attribute RAM, 1K Character ROM = 3K total. The ROM takes a further 8K, so there is 2K spare internal memory. Therefore a full colour 40x25 computer with 2K program space is available within the FPGA without needing an external SRAM chip.*	: : io1 : entity work.SBCTextDisplayRGB **generic map(** **HORIZ_CHARS => 40,** **CLOCKS_PER_PIXEL => 6,** **CLOCKS_PER_SCANLINE => 3200,** **DISPLAY_TOP_SCANLINE => 65,** **VERT_SCANLINES => 312,** **VERT_PIXEL_SCANLINES => 1,** **VSYNC_SCANLINES => 4,** **HSYNC_CLOCKS => 235,** **DISPLAY_LEFT_CLOCK => 850** **)** port map (n_reset => n_reset, :
NTSC colour display 80x25 chars centred on screen *Internal memory used: 2K Display RAM, 2K Attribute RAM, 1K Character ROM = 5K total. The ROM takes a further 8K, so no remaining internal memory. Program memory must therefore use external SRAM.*	: : io1 : entity work.SBCTextDisplayRGB **generic map(** **CLOCKS_PER_PIXEL => 3,** **CLOCKS_PER_SCANLINE => 3200,** **DISPLAY_TOP_SCANLINE => 37,** **VERT_SCANLINES => 262,** **VERT_PIXEL_SCANLINES => 1,** **VSYNC_SCANLINES => 4,** **HSYNC_CLOCKS => 235,** **DISPLAY_LEFT_CLOCK => 850** **)** port map (n_reset => n_reset, :
NTSC colour display 40x25 chars centred on screen *Internal memory used: 1K Display RAM, 1K Attribute RAM, 1K Character ROM = 3K total. The ROM takes a further 8K, so there is 2K spare internal memory. Therefore a full colour 40x25 computer with 2K program space is available within the FPGA without needing an external SRAM chip.*	: : io1 : entity work.SBCTextDisplayRGB **generic map(** **HORIZ_CHARS => 40,** **CLOCKS_PER_PIXEL => 6,** **CLOCKS_PER_SCANLINE => 3200,** **DISPLAY_TOP_SCANLINE => 37,** **VERT_SCANLINES => 262,** **VERT_PIXEL_SCANLINES => 1,** **VSYNC_SCANLINES => 4,** **HSYNC_CLOCKS => 235,** **DISPLAY_LEFT_CLOCK => 850** **)** port map (n_reset => n_reset, :
VGA Monitor, 40x25 chars	: io1 : entity work.SBCTextDisplayRGB

Internal memory used: 1K Display RAM, 1K Attribute RAM, 1K Character ROM = 3K total. The ROM takes a further 8K, so there is 2K spare internal memory. Therefore a computer with full colour VGA monitor compatible 40x25 display and 2K program space is available within the FPGA without needing an external SRAM chip.	**generic map(** **HORIZ_CHARS => 40,** **CLOCKS_PER_PIXEL => 4** **)** port map (n_reset => n_reset, :
VGA Monitor, 80x25 chars running at 640x400 resolution *Same format as for the default VGA display but in a different VGA mode so that the complete screen is used (larger characters).* *Internal memory used: 2K Display RAM, 2K Attribute RAM, 1K Character ROM = 5K total. The ROM takes a further 8K, so no remaining internal memory. Program memory must therefore use external SRAM.*	: io1 : entity work.SBCTextDisplayRGB **generic map(** **DISPLAY_TOP_SCANLINE => 35,** **VERT_SCANLINES => 448,** **V_SYNC_ACTIVE => '1'** **)** port map (n_reset => n_reset, :

The complete set of parameters that can be altered are at the top of the display VHDL and shown here (values are for the default 640x480 VGA, 80x25 chars):

```
constant EXTENDED_CHARSET : integer := 0; -- 1 = 256 chars, 0 = 128 chars
constant COLOUR_ATTS_ENABLED : integer := 1; -- 1=Colour for each character, 0=Colour applied to whole display
-- VGA 640x480 Default values
constant VERT_CHARS : integer := 25;
constant HORIZ_CHARS : integer := 80;
constant CLOCKS_PER_SCANLINE : integer := 1600; -- NTSC/PAL = 3200
constant DISPLAY_TOP_SCANLINE : integer := 35+40;
constant DISPLAY_LEFT_CLOCK : integer := 288; -- NTSC/PAL = 600+
constant VERT_SCANLINES : integer := 525; -- NTSC=262, PAL=312
constant VSYNC_SCANLINES : integer := 2; -- NTSC/PAL = 4
constant HSYNC_CLOCKS : integer := 192; -- NTSC/PAL = 235
constant VERT_PIXEL_SCANLINES : integer := 2;
constant CLOCKS_PER_PIXEL : integer := 2; -- min = 2
constant H_SYNC_ACTIVE : std_logic := '0';
constant V_SYNC_ACTIVE : std_logic := '0';
constant DEFAULT_ATT : std_logic_vector(7 downto 0) := "00001111" -- background iBGR | foreground iBGR (i=intensity)
constant ANSI_DEFAULT_ATT : std_logic_vector(7 downto 0) := "00000111" -- background iBGR | foreground iBGR (i=intensity)
```

SD Card

Coming soon (all working, just not published yet)

To include the SD card interface, copy and paste the following to the same place as the serial or monitor/keyboard above.

6502 or 6809	Z80
sd1 : entity work.sd_controller port map(sdCS => sdCS, sdMOSI => sdMOSI, sdMISO => sdMISO, sdSCLK => sdSCLK, n_wr => n_sdCardCS or cpuClock or n_WR, n_rd => n_sdCardCS or cpuClock or (not n_WR), n_reset => n_reset, dataIn => cpuDataOut, dataOut => sdCardDataOut, regAddr => cpuAddress(2 downto 0), driveLED => driveLED, clk => sdClock -- twice the spi clk);	sd1 : entity work.sd_controller port map(sdCS => sdCS, sdMOSI => sdMOSI, sdMISO => sdMISO, sdSCLK => sdSCLK, n_wr => n_sdCardCS or n_ioWR, n_rd => n_sdCardCS or n_ioRD, n_reset => n_reset, dataIn => cpuDataOut, dataOut => sdCardDataOut, regAddr => cpuAddress(2 downto 0), driveLED => driveLED, clk => sdClock -- twice the spi clk);

Memory read/write signals

The memory read logic is the same for the 6502 or 6809. However, the Z80 uses different signals as it is able to address IO and memory space separately. As a result, the memory read and memory write signals are defined differently.

6502 or 6809	Z80

n_memRD <= not(cpuClock) nand n_WR;
n_memWR <= not(cpuClock) nand (not n_WR);
	n_ioWR <= n_WR or n_IORQ;
	n_memWR <= n_WR or n_MREQ;
	n_ioRD <= n_RD or n_IORQ;
	n_memRD <= n_RD or n_MREQ;

Paste the appropriate lines where it says "MEMORY READ/WRITE LOGIC GOES HERE"

Chip selects

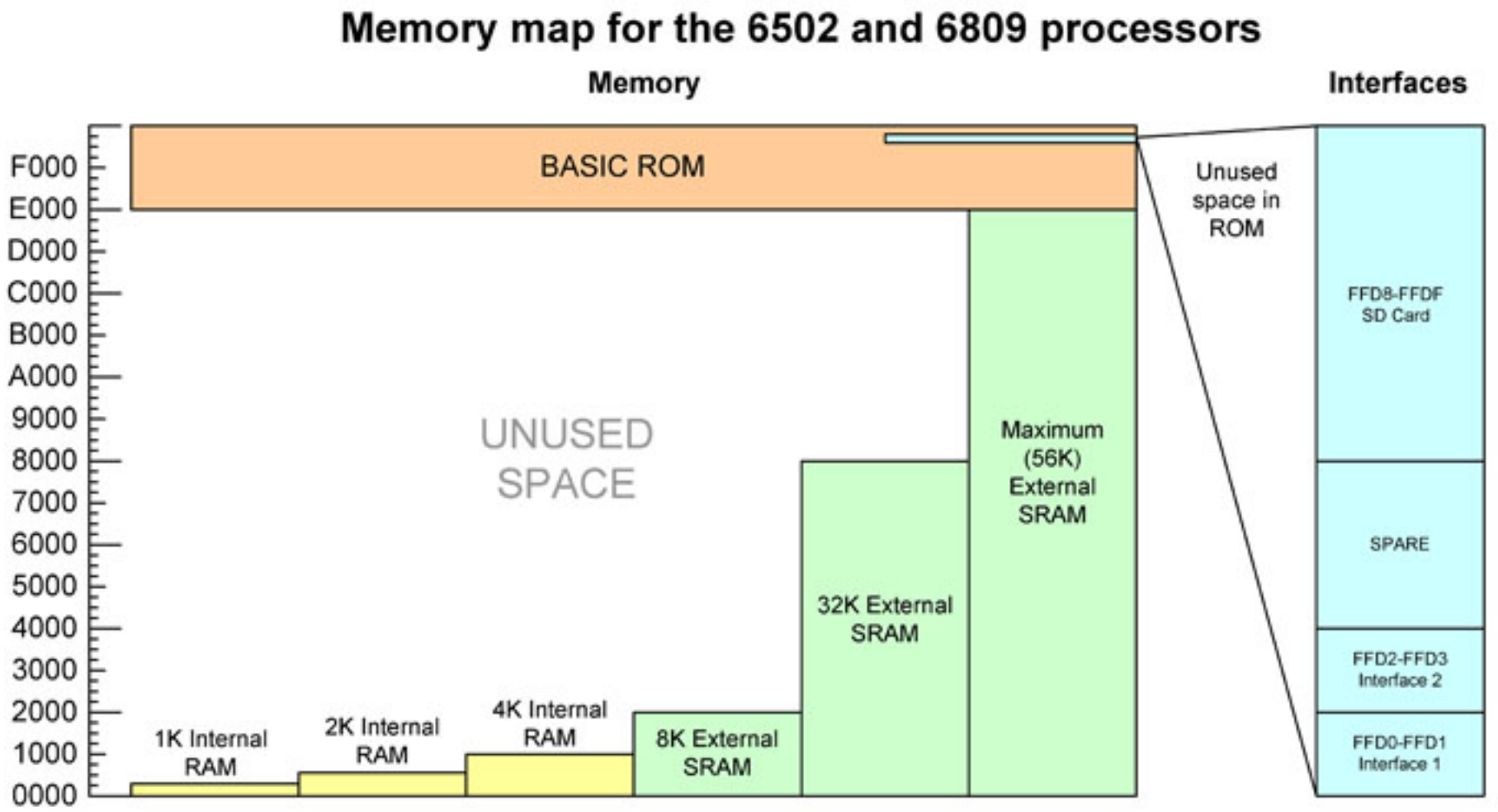
Each piece of memory or peripheral needs to be addressed at a fixed location within the CPU's 64KByte memory map.

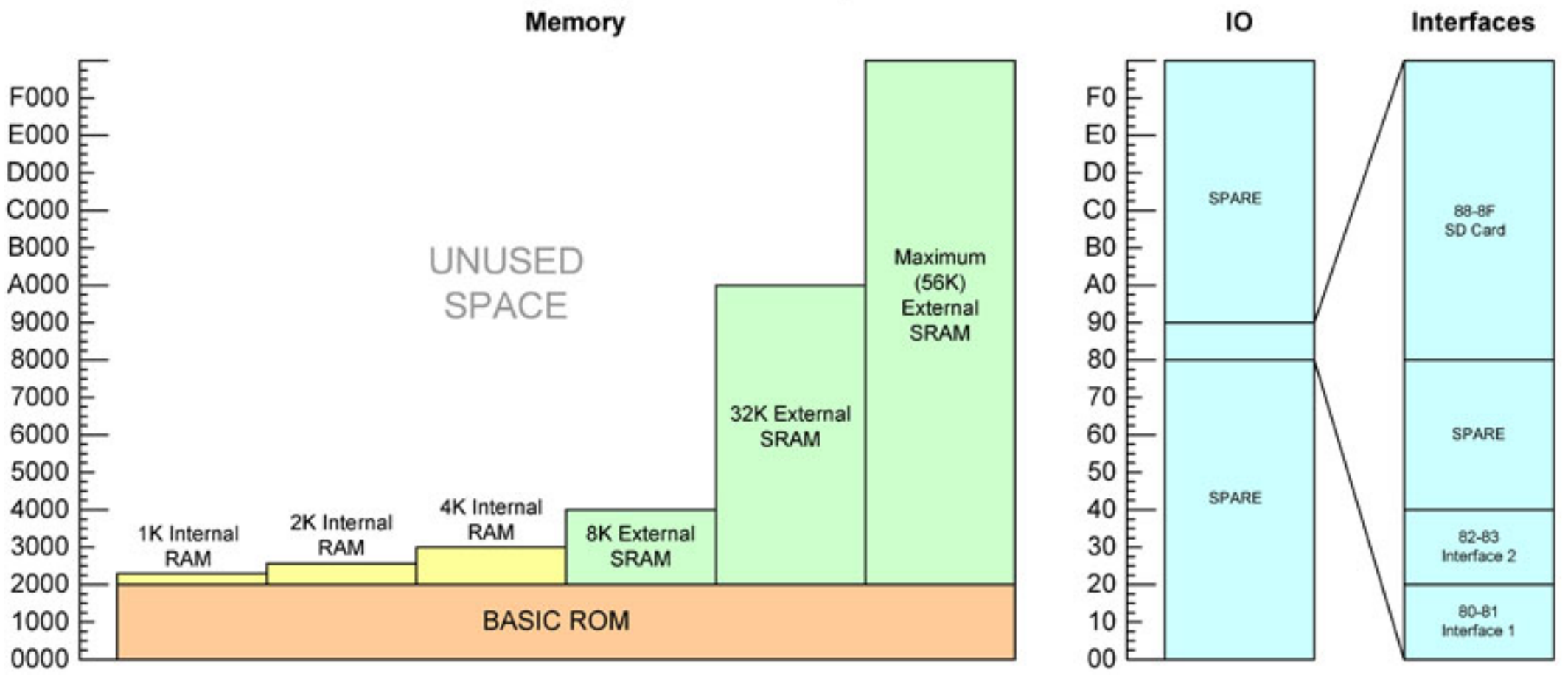
The 6809 and 6502 designs has the ROM appearing at the top of the memory space (because CPU execution reads the top few bytes of the memory to determine where to start, so ROM is needed at the top).

The Z80, on the other hand, always starts execution at location $0000, so for the Z80 system the ROM is placed at the bottom of memory.

The Z80 has separate addressing for I/O peripherals which doesn't interfere with memory addressing. The 6502 and 6809 don't have this, so peripherals need to appear within the memory space.

Here is the memory map that is used (you can change if needed by altering the BASIC ROM code)





The BASIC ROMs that have been provided have free space in the 6809 ROM between $FF85 and $FFE9. The 6502 ROM has unused space between $FF77 and $FFF9. So, to achieve as much commonality as possible between the systems, address $FFD0/$FFD1 is chosen to be the address where the first serial interface (or TV/Keyboard) will be addressed and $FFD2/$FFD3 will be the pairing chosen for the second (if used).

Within the Z80 architecture, IO address $80 and $81 is used for the first interface and $82 and $83 is used for the second, if fitted.

Address decoding for the RAM is described in the following table. Yellow cells are connected to the RAM. "x" cells are decoding "don't care" values.

For the 6502 and 6809 the RAM starts at zero so the address decoding is straightforward ie. all remaining bits not used by RAM must be zero to select it.

For the Z80 it is slightly more complex because of the 8K ($2000) offset. This is straightforward from RAM up to 8K as they all fit in the second 8K block in the memory map (ie. A15..A13 = "001"). The problem occurs with the 32K external RAM

option. To resolve this, the 32K memory is regarded as four 8K memory blocks, so each has a particular A15...A13 pattern. This is shown below. *Note, for the 32K memory for the Z80, A14 and A13 are used for both decoding and for internal addressing, which at first sight appears to break the rule where decode uses lines not used internally. However, this is correct and works properly.*

	Start addr	End addr	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4
6809 or 6502														
1K	0000	03FF	0	0	0	0	0	0	x	x	x	x	x	x
2K	0000	07FF	0	0	0	0	0	x	x	x	x	x	x	x
4K	0000	0FFF	0	0	0	0	x	x	x	x	x	x	x	x
8K	0000	1FFF	0	0	0	x	x	x	x	x	x	x	x	x
32K	0000	7FFF	0	x	x	x	x	x	x	x	x	x	x	x
Z80														
1K	2000	23FF	0	0	1	0	0	0	x	x	x	x	x	x
2K	2000	27FF	0	0	1	0	0	x	x	x	x	x	x	x
4K	2000	2FFF	0	0	1	0	x	x	x	x	x	x	x	x
8K	2000	3FFF	0	0	1	x	x	x	x	x	x	x	x	x
32K	2000	3FFF	0	0	1	x	x	x	x	x	x	x	x	x
	4000	5FFF	0	1	0	x	x	x	x	x	x	x	x	x
	6000	7FFF	0	1	1	x	x	x	x	x	x	x	x	x
	8000	9FFF	1	0	0	x	x	x	x	x	x	x	x	x

If MAXIMUM external RAM then the RAM will occupy all addresses not taken by the ROM, so the address decode is simply "not ROM".

For the 6502 or 6809 processors

Chip select code is

```
n_basRomCS <= '0' when cpuAddress(15 downto 13) = "111" else '1'; --8K at top of memory
n_interface1CS <= '0' when cpuAddress(15 downto 1) = "111111111101000" else '1'; -- 2 bytes FFD0-FFD1
n_interface2CS <= '0' when cpuAddress(15 downto 1) = "111111111101001" else '1'; -- 2 bytes FFD2-FFD3
n_sdCardCS <= '0' when cpuAddress(15 downto 3) = "1111111111011" else '1'; -- 8 bytes FFD8-FFDF
```

If the second interface is NOT being used then there is no need to include the line that starts "n_interface2CS <=". Similarly for the n_sdCardCS.

Then one of the following lines...

1K internal RAM	`n_internalRam1CS <= '0' when cpuAddress(15 downto 10) = "000000" else '1';`
2K internal RAM	`n_internalRam1CS <= '0' when cpuAddress(15 downto 11) = "00000" else '1';`
4K internal RAM	`n_internalRam1CS <= '0' when cpuAddress(15 downto 12) = "0000" else '1';`
8K external SRAM	`n_externalRamCS <= '0' when cpuAddress(15 downto 13) = "000" else '1';`
32K external SRAM	`n_externalRamCS <= '0' when cpuAddress(15) = '0' else '1';`
Full external SRAM	`n_externalRamCS<= not n_basRomCS;`

For the Z80 processor

Chip select code is

```
n_basRomCS <= '0' when cpuAddress(15 downto 13) = "000" else '1'; --8K at bottom of memory
n_interface1CS <= '0' when cpuAddress(7 downto 1) = "1000000" and (n_ioWR='0' or n_ioRD = '0') else '1'; -- 2 Bytes $80-$81
n_interface2CS <= '0' when cpuAddress(7 downto 1) = "1000001" and (n_ioWR='0' or n_ioRD = '0') else '1'; -- 2 Bytes $82-$83
n_sdCardCS <= '0' when cpuAddress(7 downto 3) = "10001" and (n_ioWR='0' or n_ioRD = '0') else '1'; -- 8 Bytes $88-$8F
```

If the second interface is NOT being used then there is no need to include the line that starts "n_interface2CS <=". Similarly for the n_sdCardCS.

Then one of the following lines...

1K internal RAM	`n_internalRam1CS <= '0' when cpuAddress(15 downto 10) = "001000" else '1';`
2K internal RAM	`n_internalRam1CS <= '0' when cpuAddress(15 downto 11) = "00100" else '1';`
4K internal RAM	`n_internalRam1CS <= '0' when cpuAddress(15 downto 12) = "0010" else '1';`
8K external SRAM	`n_externalRamCS <= '0' when cpuAddress(15 downto 13) = "001" else '1';`
32K external SRAM	`n_externalRamCS <= '0' when cpuAddress(15 downto 13) = "001" or cpuAddress(15 downto 13) = "010" or cpuAddress(15 downto 13) = "011" or cpuAddress(15 downto 13) = "100" else '1';`
Full external SRAM	`n_externalRamCS<= not n_basRomCS;`

Paste the appropriate code to where it says "CHIP SELECTS GO HERE"

If a particular peripheral or memory chip is to appear in a specific place in the memory map, it needs to be "fully decoded". It is very common in small computers for chips to only be "partially decoded", mainly due to the fact that full decoding needs more logic chips on the board. Using FPGA however,

full decoding can be done without any additional components.

So, for example, the SD card interface needs 3 address lines (therefore 8 internal addresses - 2^3=8) to access the internal features of the chip then these would normally be connected to the lowest 3 address lines of the address bus so that the 8 internal addresses would appear consecutively in the memory address space. Full memory decoding then needs ALL the remaining address lines not connected to the chip to go to the chip selection logic. In this example the address for this chip is
SSSS SSSS SSSS SCCC where S is a select address bit ant C is a chip internal address. All the "S" bits need to be assigned to a particular value.
*if we pick "1111 1111 1101 1" as the address chip select bits then that particular chip would appear at "FFD8" (1111 1111 1101 1**000**) to "FFDF" (1111 1111 1101 1**111**) in the memory map.*

Similarly, all other memory and peripherals are also fully decoded - the more internal addresses used in the chip, the fewer are used for the "chip selects", therefore different number of bits are used in the above equations.

Bus isolation

For the CPU incoming (read) data bus is shared by all peripherals, so VHDL glue logic is needed to ensure only one peripheral presents the data onto the bus at any time. Each peripheral has it's own data out bus, so depending on the chip selects, the appropriate data is connected to the CPU in bus.

Note: For the 6502 or 6809, because the interface memory map is "stealing" some locations from the BASIC ROM, the interface data lines must appear before (in preference to) the BASIC rom data line.

```
cpuDataIn <=
interface1DataOut when n_interface1CS = '0' else
interface2DataOut when n_interface2CS = '0' else
sdCardDataOut when n_sdCardCS = '0' else
basRomData when n_basRomCS = '0' else
internalRam1DataOut when n_internalRam1CS= '0' else
sramData when n_externalRamCS= '0' else
x"FF";
```

Important notes (changes needed to the above code):
1. If the second interface or SD card is not used then remove the "interface2DataOut" or "sdCardDataOut" lines of code as appropriate.
2. You would have chosen whether internal RAM or external RAM is being used, so remove the internalRam1DataOut line if using external RAM, or remove the sramData line if using internal RAM.

Paste the appropriate code to where it says "BUS ISOLATION GOES HERE".

The clocks

The CPU uses a simple divider to reduce the 50MHz incoming clock to the required CPU speed. Similarly, the 50MHz clock is divided by 50 to get the 1MHz SD card clock.

For the serial, a slightly more complex divider is used and is based on the DDS (Direct digital synthesis) method of producing an arbitrary frequency using a 16 bit counter. This is MUCH better for the serial speeds because a greater accuracy (well within 1%) is easily achieved. The default value (2416) within the code has the serial clock set for 115200 baud. See the comments within the code for the other values.

Basically, f = (increment x 50,000,000) / 65,536

Where f is the baud rate x 16, as required for the ACIA to run properly.

Paste the following code to where it says "SYSTEM CLOCKS GO HERE". The serial clock parts of the code can be removed if serial is not being used. Can remain, though.

```
-- SUB-CIRCUIT CLOCK SIGNALS
serialClock <= serialClkCount(15);
process (clk)
begin
if rising_edge(clk) then

if cpuClkCount < 4 then -- 4 = 10MHz, 3 = 12.5MHz, 2=16.6MHz, 1=25MHz
cpuClkCount <= cpuClkCount + 1;
else
cpuClkCount <= (others=>'0');
end if;
if cpuClkCount < 2 then -- 2 when 10MHz, 2 when 12.5MHz, 2 when 16.6MHz, 1 when 25MHz
cpuClock <= '0';
else
cpuClock <= '1';
end if;

if sdClkCount < 49 then -- 1MHz
sdClkCount <= sdClkCount + 1;
else
sdClkCount <= (others=>'0');
```

```
end if;
if sdClkCount < 25 then
sdClock <= '0';
else
sdClock <= '1';
end if;

-- Serial clock DDS
-- 50MHz master input clock:
-- Baud Increment
-- 115200 2416
-- 38400 805
-- 19200 403
-- 9600 201
-- 4800 101
-- 2400 50
serialClkCount <= serialClkCount + 2416;
end if;
end process;
```

This code above sets the CPU frequency at 10MHz, which is much faster than what you would get if you built a computer using discrete components. You can control the frequency of the CPU by changing the upper counter check shown in red, above. If you make the change to the line in red, the one in green (the "half way" count) also needs changing. Here is a table of some of the values you could use.

CPU frequency	Counter top	Counter half-way
1MHz	**if cpuClkCount < 49 then**	**if cpuClkCount < 25 then**
2MHz	**if cpuClkCount < 24 then**	**if cpuClkCount < 12 then**
5MHz	**if cpuClkCount < 9 then**	**if cpuClkCount < 4 then**
10MHz	**if cpuClkCount < 4 then**	**if cpuClkCount < 2 then**
12.5MHz	**if cpuClkCount < 3 then**	**if cpuClkCount < 2 then**
16.6MHz	**if cpuClkCount < 2 then**	**if cpuClkCount < 2 then**
25MHz	**if cpuClkCount < 1 then**	**if cpuClkCount < 1 then**

Note: high frequencies (in pink) work with internal RAM but are unlikely to work (particularly with the 6502 or 6809) with external SRAM unless the chip is fast and has short connections. If you try and the computer fails to respond properly then reduce the CPU frequency as shown above. Try it and see - no harm will come if you exceed the clock max.

CPU frequency = 50,000,000 / (counter top+1) Hz

Finished

The VHDL is now complete !

Assigning the internal connections to actual pins on the FPGA

If using the wiring that I show above, this has already been done for you.

Open the "Microcomputer.qsf" file and you will see the assignments.

The format of the assignment line is
set_location_assignment PIN_pinNumber **-to** vhdlSignalName

Most (not all) pins support weak pull-ups, so the built-in switch on the board (connected to pin 144) has the pull-up turned on.

```
set_instance_assignment -name WEAK_PULL_UP_RESISTOR ON -to n_reset
```

Build completion

If you have completed the above steps then that completes the build of the microcomputer. It can now be compiled and uploaded to the FPGA.

If you have wired up as shown in the diagram, once the upload is complete you will immediately (within a few tens of milliseconds) see the appropriate message appear on the monitor display or on the terminal connected to the board - whichever you chose to build.

The computer is then ready to use!

ANSI codes supported

The TV/monitor display supports the following "escape" sequences

<table>
<tr><th>SEQUENCE</th><th>NAME</th><th>DESCRIPTION</th><th>EXAMPLE</th></tr>
<tr><td>ESC[H</td><td>Cursor Home</td><td>Moves the display cursor to top-left</td><td>PRINT CHR$(27);"[H";</td></tr>
<tr><td>ESC[K</td><td>Erase EOL</td><td>All characters on current line from cursor position to end replaced with spaces</td><td>PRINT CHR$(27);"[K";</td></tr>
<tr><td>ESC[s</td><td>Save cursor pos</td><td>Current row and column saved</td><td>PRINT CHR$(27);"[s";</td></tr>
<tr><td>ESC[u</td><td>Restore cursor pos</td><td>Return cursor to previously stored row and column</td><td>PRINT CHR$(27);"[u";</td></tr>
<tr><td>ESC[2J</td><td>Clear screen</td><td>Same as PRINT CHR$(12) - clear screen and move cursor to top left</td><td>PRINT CHR$(27);"[2J";</td></tr>
<tr><td>ESC[0J
or
ESC[J</td><td>Clear to end of screen</td><td>All positions from the current cursor to end of screen are replaced with spaces</td><td>PRINT CHR$(27);"[0J";</td></tr>
<tr><td>ESC[{val}A</td><td>Cursor up</td><td>Move cursor up {val} places</td><td>PRINT CHR$(27);"[3A";</td></tr>
<tr><td>ESC[{val}B</td><td>Cursor down</td><td>Move cursor down {val} places</td><td>PRINT CHR$(27);"[2B";</td></tr>
<tr><td>ESC[{val}C</td><td>Cursor forward</td><td>Move cursor right {val} places</td><td>PRINT CHR$(27);"[15C";</td></tr>
<tr><td>ESC[{val}D</td><td>Cursor backward</td><td>Move cursor left {val} places</td><td>PRINT CHR$(27);"[1D";</td></tr>
<tr><td>ESC[{row};{col}H</td><td>Set cursor position</td><td>Move the cursor to row {row} and column {col} (1;1=top left)</td><td>PRINT CHR$(27);"[5;33H";</td></tr>
<tr><td>ESC[L
Only ONE line supported</td><td>Delete line</td><td>Delete the current cursor line and move the display underneath it up one line. Last line of the display then is blank.</td><td>PRINT CHR$(27);"[L"</td></tr>
<tr><td>ESC[M
Only ONE line supported</td><td>Insert line</td><td>Insert a blank line at the current cursor position and move the display below it down one line.</td><td>PRINT CHR$(27);"[M"</td></tr>
<tr><td>ESC[{val}m
ESC[{val1};...
{val8}m

(1 to 4 values)</td><td>Set graphics rendition

Note: Text and background setting reversed if inverse is active.</td><td>Set the active foreground or background text colour. If character attributes enabled, all characters that appear after this sequence will use the colour specified. If character attributes disabled then the complete screen will be displayed in the colours specified.

One or two parameters can be specified in the same command.
<table>
<tr><th>val</th><th>effect</th></tr>
<tr><td>0</td><td>Reset atts to default</td></tr>
<tr><td>1</td><td>Bold (bright) text on</td></tr>
<tr><td>7</td><td>Inverse colours (back<=>text)</td></tr>
<tr><td>22</td><td>Bold (bright) text off</td></tr>
<tr><td>27</td><td>Normal colours</td></tr>
<tr><td>30</td><td>Text BLACK</td></tr>
<tr><td>31</td><td>Text RED</td></tr>
<tr><td>32</td><td>Text GREEN</td></tr>
<tr><td>33</td><td>Text YELLOW</td></tr>
<tr><td>34</td><td>Text BLUE</td></tr>
<tr><td>35</td><td>Text MAGENTA</td></tr>
<tr><td>36</td><td>Text CYAN</td></tr>
<tr><td>37</td><td>Text WHITE</td></tr>
<tr><td>40</td><td>Background BLACK</td></tr>
<tr><td>41</td><td>Background RED</td></tr>
<tr><td>42</td><td>Background GREEN</td></tr>
<tr><td>43</td><td>Background YELLOW</td></tr>
<tr><td>44</td><td>Background BLUE</td></tr>
<tr><td>45</td><td>Background MAGENTA</td></tr>
<tr><td>46</td><td>Background CYAN</td></tr>
<tr><td>47</td><td>Background WHITE</td></tr>
<tr><td>90</td><td>Text GREY</td></tr>
<tr><td>91</td><td>Text BRIGHT RED</td></tr>
<tr><td>92</td><td>Text BRIGHT GREEN</td></tr>
<tr><td>93</td><td>Text BRIGHT YELLOW</td></tr>
<tr><td>94</td><td>Text BRIGHT BLUE</td></tr>
<tr><td>95</td><td>Text BRIGHT MAGENTA</td></tr>
<tr><td>96</td><td>Text BRIGHT CYAN</td></tr>
</table></td><td>PRINT CHR$(27);"[31;44mTHIS IS R
PRINT CHR$(27);"[93;42mTHIS IS B
YELLOW ON GREEN"
PRINT CHR$(27);"[30mTHIS IS BLA
PRINT CHR$(27);"[41mBACKGROU
RED"
will give the following...

THIS IS RED ON BLUE
THIS IS BRIGHT YELLOW ON GRE
THIS IS BLACK TEXT
BACKGROUND NOW RED</td></tr>
</table>

97	Text BRIGHT WHITE
100	Background GREY
101	Background BRIGHT RED
102	Background BRIGHT GREEN
103	Background BRIGHT YELLOW
104	Background BRIGHT BLUE
105	Background BRIGHT MAGENTA
106	Background BRIGHTCYAN
107	Background WHITE

ROM BASIC - implemented keywords and tokens

6809 (FROM EXTENDED COLOR BASIC)

FOR, GO, REM, ELSE, IF, DATA, PRINT, ON GOSUB, ON GOTO, INPUT, LINE INPUT, END, NEXT, DIM, READ, RUN, RESTORE, RETURN, STOP, POKE, CONT, LIST, CLEAR, NEW, EXEC, TAB, TO, SUB, THEN, NOT, STEP, +, -, *, /, ^, AND, OR, >, =, <, DEL, DEF, LET, RENUM, FN, &, &H, TRON, TROFF, EDIT, SGN, INT, ABS, USR, RND, SIN, PEEK, LEN, STR$, VAL, ASC, CHR$, LEFT$, RIGHT$, MID$, INKEY$, MEM, ATN, COS, TAN, EXP, FIX, LOG, SQR, HEX$, VARPTR, INSTR, STRING$, MID$ (MODIFICATION), POS

6502 (OSI BASIC)

END, FOR, NEXT, DATA, INPUT, DIM, READ, LET, GOTO, RUN, IF, RESTORE, GOSUB, RETURN, REM, STOP, ON, NULL, WAIT, DEF, POKE, PRINT, CONT, LIST, CLEAR, NEW, TAB(, TO, FN, SPC(, THEN, NOT, STEP, SGN, INT, ABS, USR, FRE, POS, SQR, RND, LOG, EXP, COS, SIN, TAN, ATN, PEEK, LEN, STR$, VAL, ASC, CHR$, LEFT$, RIGHT$, MID$, +, -, *, /, ^, AND, OR, >, +, <

Z80 (FROM NASCOM BASIC 4.7)

SGN, INT, ABS ,USR, FRE, INP, POS, SQR, RND ,LOG, EXP, COS, SIN, TAN, ATN, PEEK ,DEEK ,LEN, STR$, VAL ,ASC, CHR$,LEFT$, RIGHT$, MID$, END, FOR, NEXT, DATA, INPUT, DIM, READ, LET, GOTO, RUN, IF, RESTORE, GOSUB, RETURN, REM, STOP, OUT, ON, NULL, WAIT, DEF, POKE, DOKE, LINES, CLS, WIDTH, MONITOR, PRINT, CONT, LIST, CLEAR, NEW, TAB, TO, FN, SPC, THEN, NOT, STEP, +, -, *, /, ^, AND, OR, >, <, =

PLUS my additional implementations here (making it version 4.7b):

HEX$(nn) - convert a SIGNED integer (-32768 to +32767) to a string containing the hex value
BIN$(nn) - convert a SIGNED integer (-32768 to +32767) to a string containing the binary value
&Hnn - interpret the value after the &H as a HEX value (signed 16 bit)
&Bnn - interpret the value after the &B as a BINARY value (signed 16 bit)

Software and VHDL project download link

By downloading these files you must agree to the following:
The original copyright owners of ROM contents are respectfully acknowledged.
Use of the contents of any file within your own projects is permitted freely, but any publishing of material containing whole or part of any file distributed here, or derived from the work that I have done here will contain an acknowledgement back to myself, Grant Searle, and a link back to this page.
Any file published or distributed that contains all or part of any file from this page must be made available free of charge.

NOTE: updated 26th Sep 2014 to use new 6809 core - many thanks to John Kent for the update on his 6809 core file. If you are taking new version of this zip and you use the 6809 implementation then make sure you update your VHDL "cpu1" instance to match the code shown earlier on this page.

Everything you need to compile the complete project is in the zip file **HERE**.

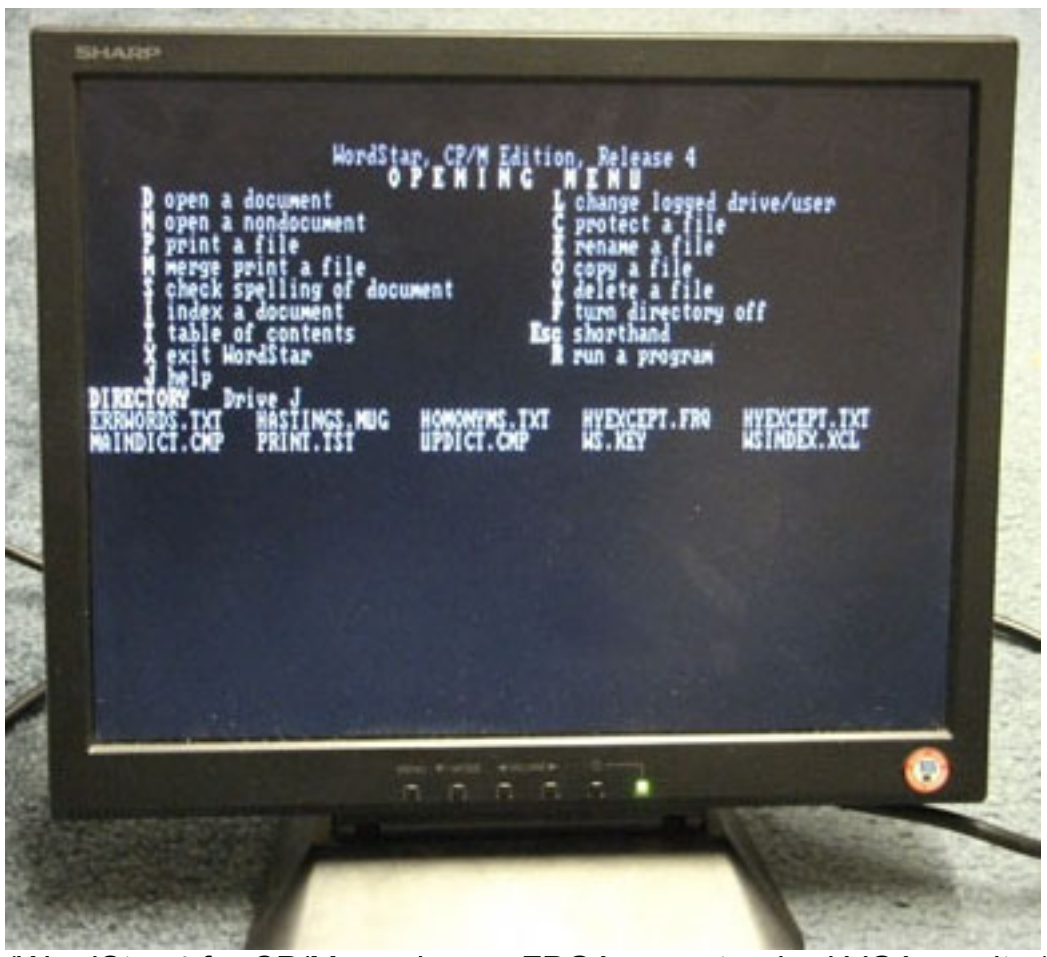
Extract to a folder (**ensure you keep the folder structure**) and open the "Microcomputer.qpf" project in Quartus II version 13sp1.

Included are the VHDL files, HEX files for the ROMs and the Quartus project file ready to open.

Worked examples - if you get stuck

Coming soon.
6502 Processor, Microsoft 8K BASIC, Monochrome monitor PAL timing, PS2 keyboard, Internal 2K RAM
6502 Processor, Microsoft 8K BASIC, VGA monitor (80x25 colour, 128 charset), PS2 keyboard, External full (56K) RAM
6502 Processor, Microsoft 8K BASIC, Serial Interface 9600 Baud, Internal 4K RAM
Z80 Processor, Microsoft 8K BASIC, Serial Interface 9600 Baud, Internal 4K RAM

Extension project 1: CP/M



(WordStar 4 for CP/M running on FPGA on a standard VGA monitor)

The full details for this are on their own web page. Please go HERE for details.

My other pages

CLICK HERE TO GO TO MY MAIN PAGE FOR MY OTHER PROJECTS

I hope this page has been useful.

Grant.

To contact me, my current eMail address can be found here. Please note that this address may change to avoid spam.