University of British Columbia  
Electrical and Computer Engineering  
CPEN 412 - **Microcomputer Systems Design**

**Assignment 1**

Copyright © 2020 PJ Davies



**Objectives**:

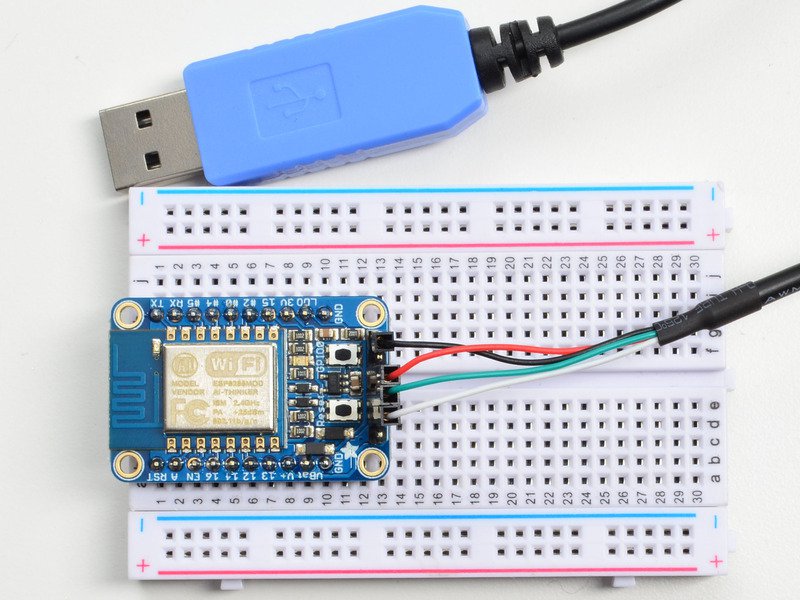
1. To design a 32k byte (16kword) Rom and 256kByte (128kWord) Ram using FPGA on-chip synchronous memory and interface to the 68000.
2. To compile and embed a 68000 debug monitor into the 32k on-Chip Rom.
3. To write ‘C’ program to test the memory in the system and add a memory test function to the debug monitor.
4. This assignment can be done in groups of 2

**Rational:** Microcomputer systems need a mixture of both RAM and ROM to function. The ROM holds firmware responsible for “booting” the system to life, while the RAM is needed for program variables, stack, heap etc. Both types of memory also need to be located in the CPUs memory map hence an address decoder is required. Firmware can be an OS, or in this case, a simple debug monitor program to allow you to interact with the system, download code, test and set memory etc.

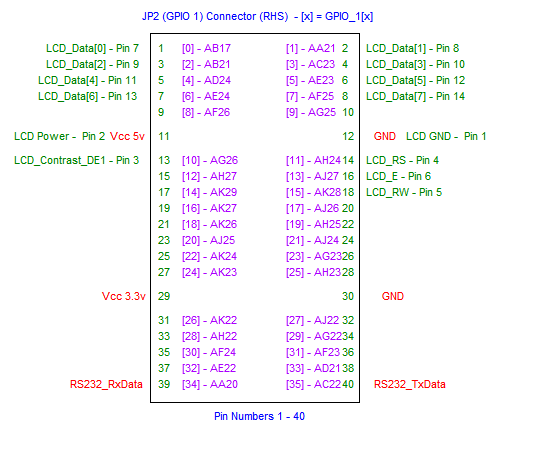
**Submission Date**: See Canvas for details

**Part A – Hardware Design - 50%**

* Make sure you have downloaded the 68k project for CPEN 412 students from the Canvas web site (*and put it in the correct folder on your C: drive – see Lab 1 details on Canvas*), along with other software and documentation such as the 68000 ‘C’ compiler/IDE and Hyper-terminal. Make sure this software in installed and that you have a USB-to-RS232 converter dongle such as the one shown below.
* This has **Black/Red** wires for power (LEAVE RED **UNCONNECTED**, as power comes via USB plug itself but connect the black to a ground on the board as shown below) - It also has **Green**/**White** wires for communication



These should be connected to the right most GPIO socket on the DE1 as indicated below



Green wire

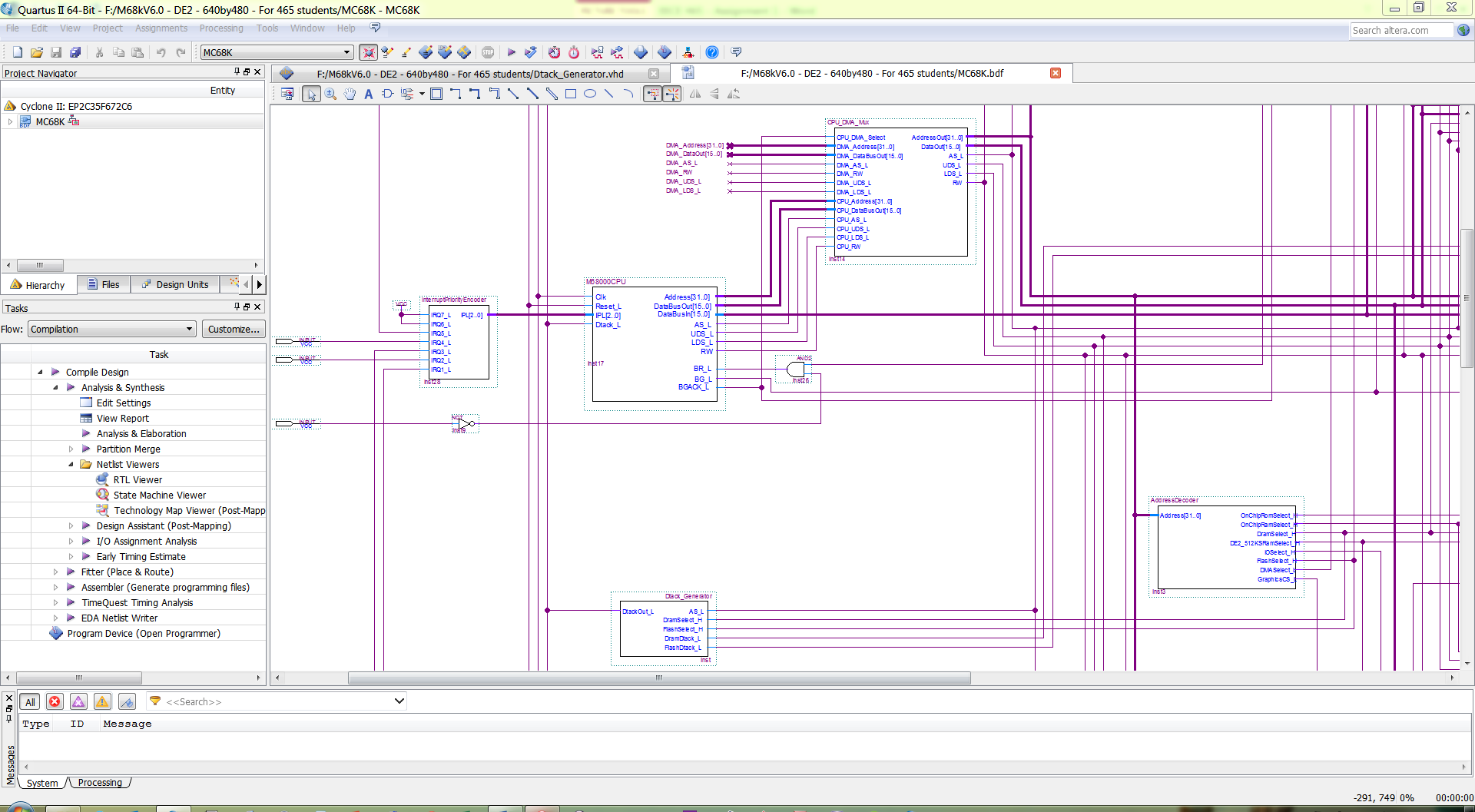
Black wire

White wire

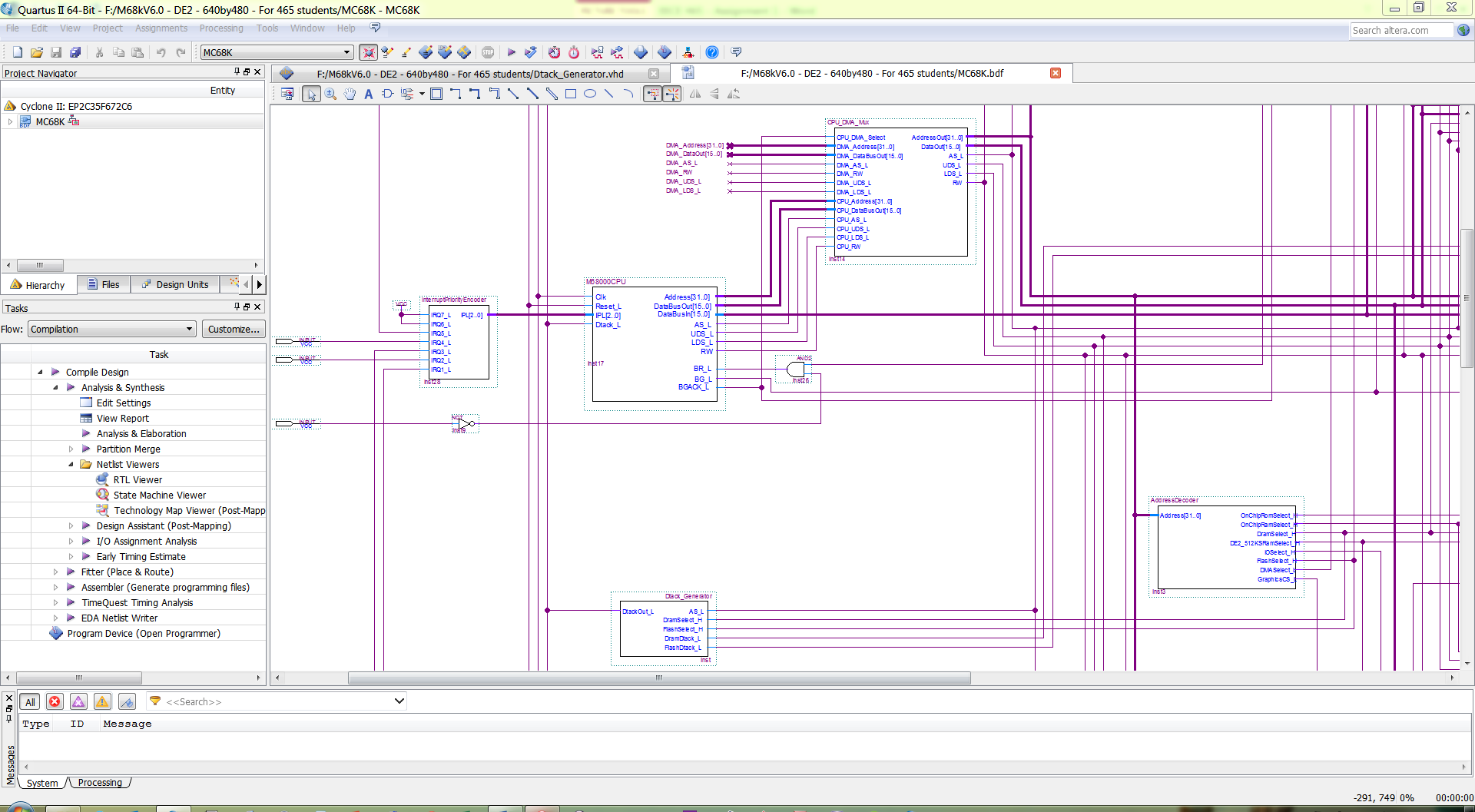
White wire

(*To test the board and the RS232 dongle, you can download the working Quartus ‘****MC68k.sof****’ file from the 412 Canvas site and program it into the DE1. You will get a debugger prompt in hyper-terminal window if it is all working).*

* ***IMPORTANT****: make sure the* ***2 left most slider******switches SW8 and 9*** *on the DE1 board are in the down position before pressing the* ***reset*** *button (Key0)*.
* Open the 68k project in Quartus. You will see the soft core 68k processor in the top level schematic as shown below

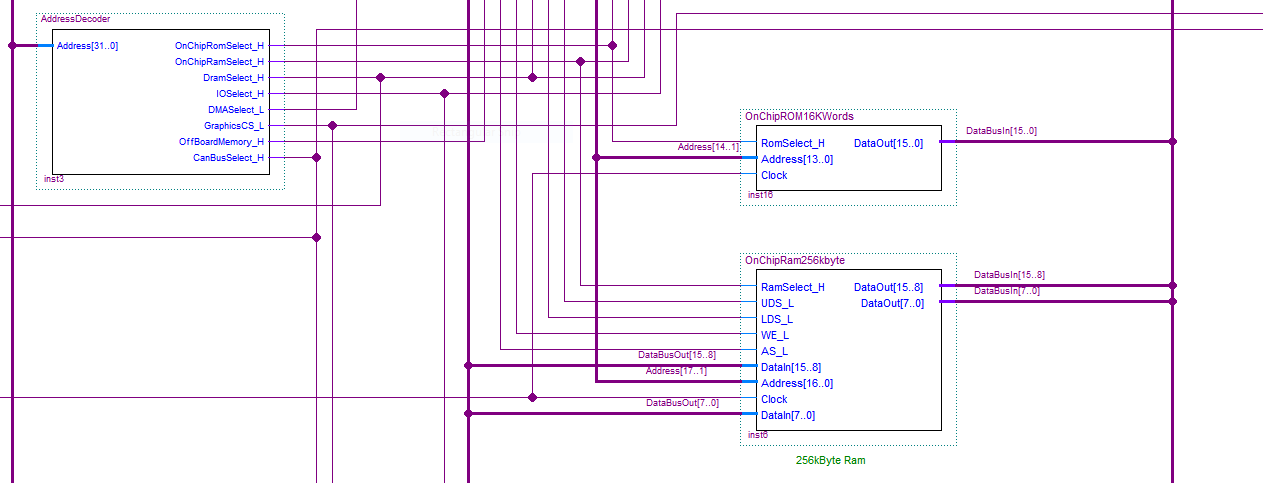


* There is also a simple **Dtack** **generator** and **Address decoder** **circuit** written in an Verilog.
* Note that the soft core version of the 68k (written in VHDL) differs slightly from the original Motorola 68000 in that
  + It has a 32 bit address bus (A31-A0) as it is not constrained to fit inside a 64 pin package like the original 68k.
  + It has separate 16 bit Data In and Data Out buses rather than one combined bi-directional data bus. This is normal for system-on-chip designs, as the two buses carry data in opposite directions. A bi-directional data bus only exists on ***real*** CPUs to cut down on the number of external pins, which is not an issue in a FPGA. Internally, real CPUs also use separate data buses, but have tri-state outputs to avoid conflict between data going in opposite directions when reading/writing via the same common pins.
  + There is no 68k **BERR** signal. This was connected to a memory management unit MMU in real systems and used to implement ‘**paging’** and **protection** when an OS was present like Unix, but is not relevant here.
  + Although the soft core 68k implements the same bus cycle timing and protocol as the original 68k (as discussed in lecture 8), the exact min/max timing relative to clock edges will vary due to different design layout and Quartus "**fitter**" choices.
* Note also that many of the 68000 signals pass through a large 2 way multiplexer (*mux*) labelled CPU\_DMA\_MUX before reaching other components on the board such as the address decoder, memory, IO etc. This mux allows a DMA (*direct memory access controller*) chip to take control of the 68000 buses to perform high speed transfers between memory (*see later lectures*). We may do an assignment on DMA later, but for the moment, the default mux signalling is set up to pass the CPU signals to the outside world, i.e. memory, IO etc.

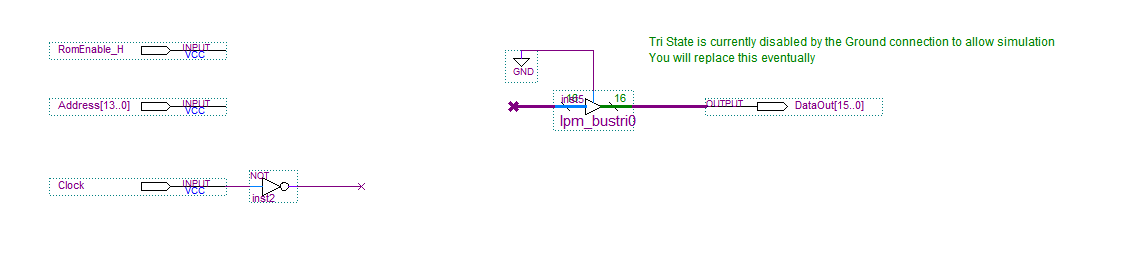


**Step 1 - Designing the 32kByte Rom to hold our Firmware**

* We need some Rom to hold a simple debug monitor so that we can talk to the 68000 system (via Hyperterminal) upon power-on/reset.
* Locate the Block for the 32kbyte (16K Word) Rom (see below), and open it by double clicking on the symbol.

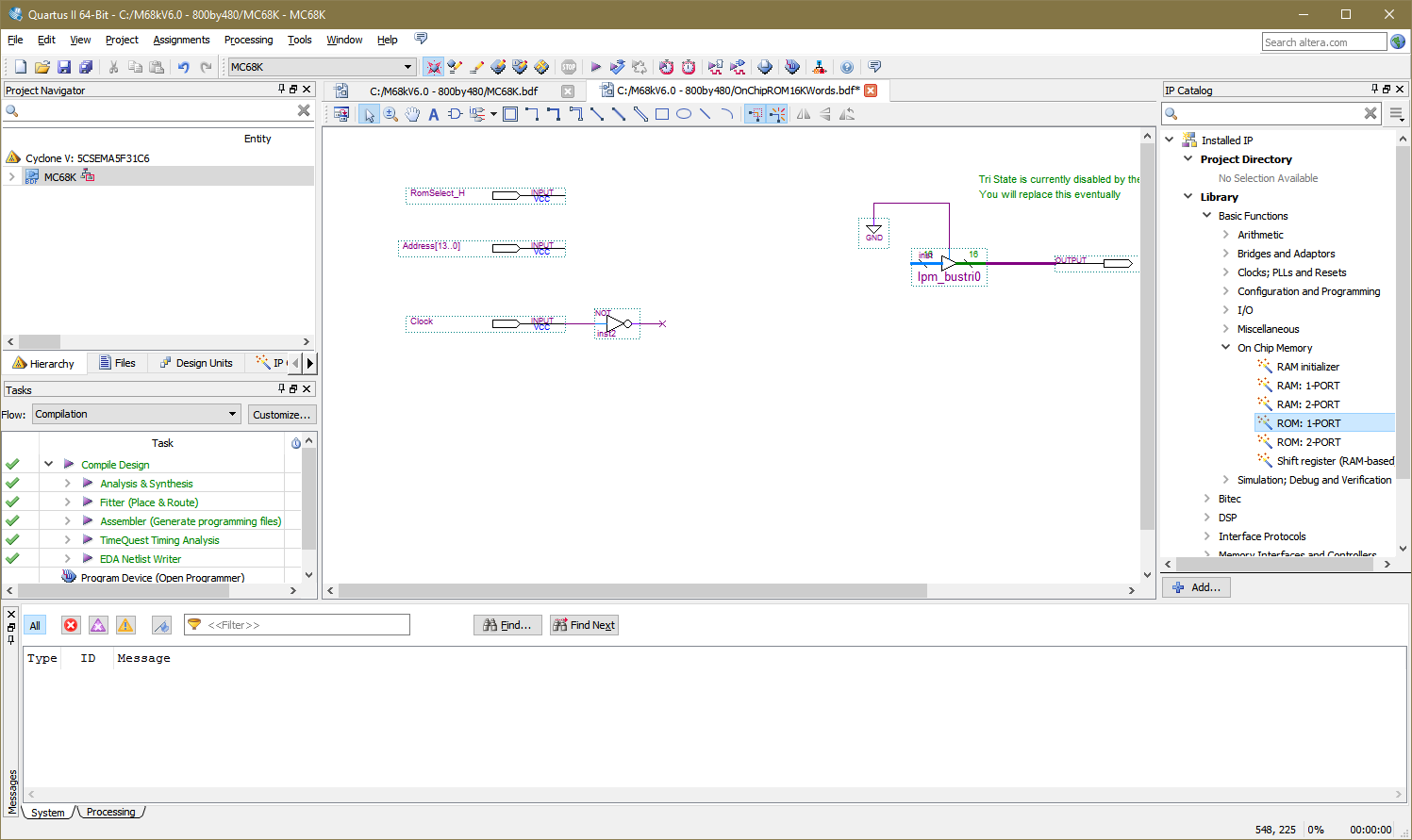


* You should see this (*incomplete*) circuit inside. RomEnable\_H comes from the main address decoder on the top level schematic, while the Address and **DataOut** buses connect to the 68000 Address and 68000 **DataIn** buses (Note the use of **In** here).
* Our job is to design 16k words of rom and connect to the these pins.

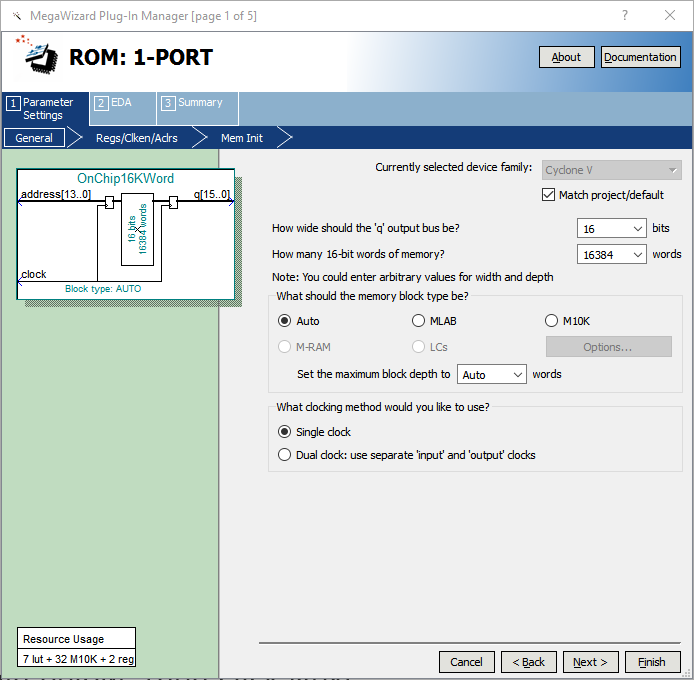


**Using the IP Catalog to create Synchronous On-Chip Rom**

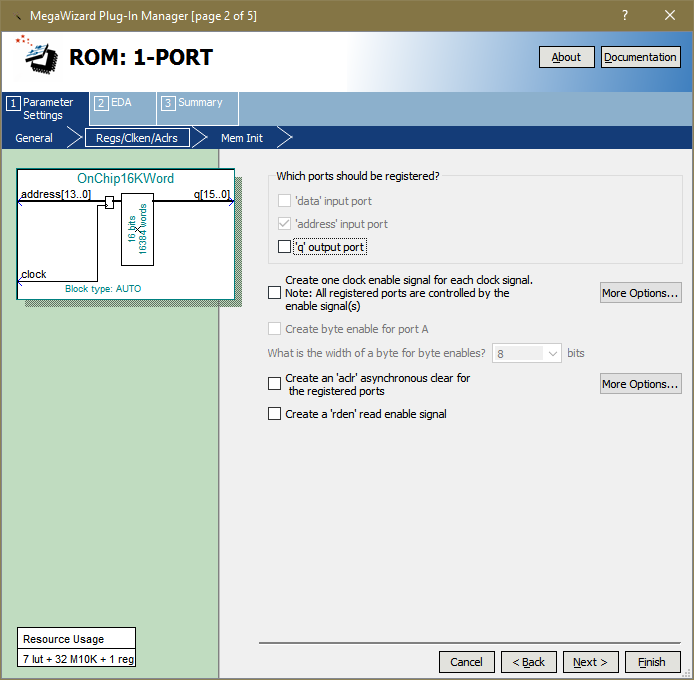
* Click on the menu “Tools->IP Catalog” and double click Rom: 1- Port from the on-chip memory choices in the left hand window.



* Name the new memory e.g."**OnChipRom16KWord**" for our new memory device
* This box will appear. Select the memory width to be 16 bits and the size to be 16,384 locations. This gives us 32kbytes (16kwords) of memory. **Click** **Next**.

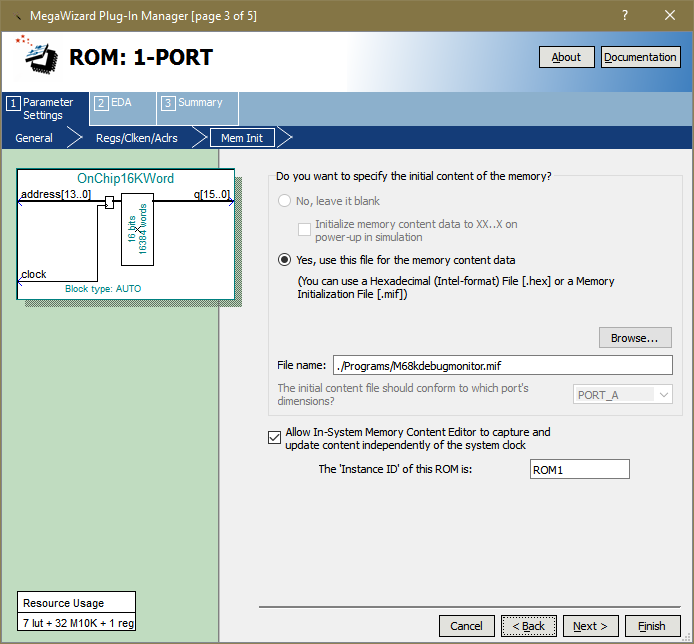


Disable the registered 'q' (data out) output as shown below, then **click Next**



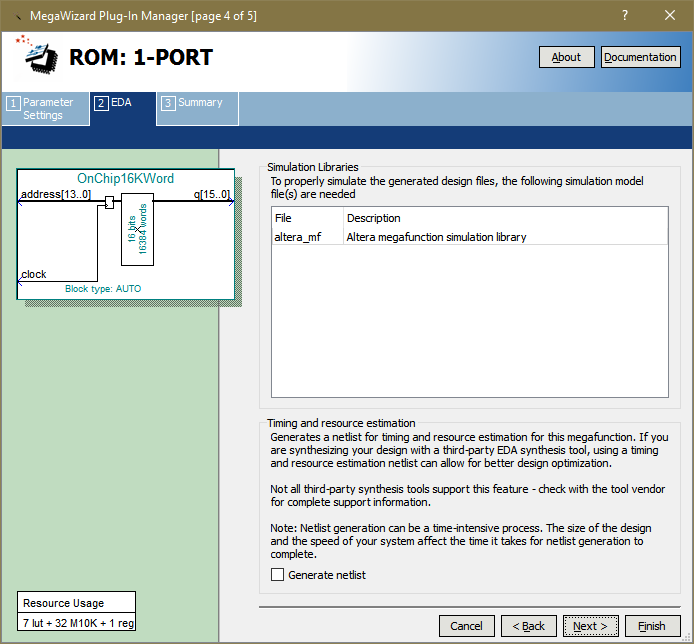
Now this dialog will appear.

Fill in the name of the file **M68kdebugmonitor.mif** that you should find by browsing a "**programs**" folder that is part of the project. A 'mif' (memory initialisation file), contains a prior-compiled version of the debug monitor which we are going to put into Rom (we'll explore this later and how to change it).

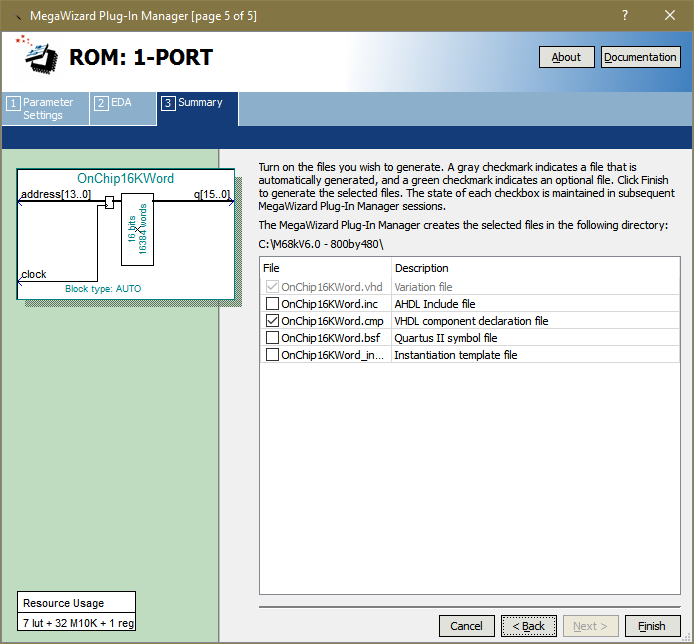


Check this **option** and fill in the name **ROM1**. This will allow Quartus to dynamically *capture* the contents of the ROM (and view it) and can be used to change the Rom contents on the DE1. Click **Next**

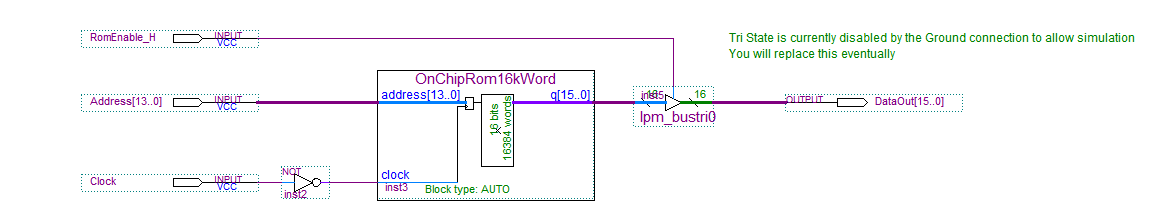
* This appears

****

Click **Next**



Now **click** **Finish**. This will have generated a logic symbol which we can now paste into our circuit as shown below and wire accordingly.

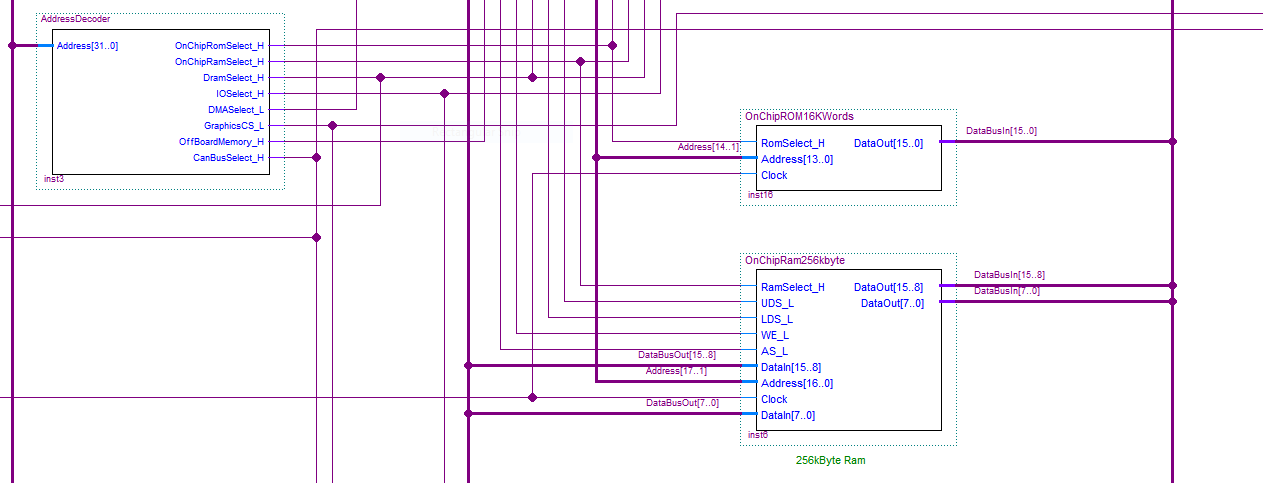


Ok let's summarise what we did here:-

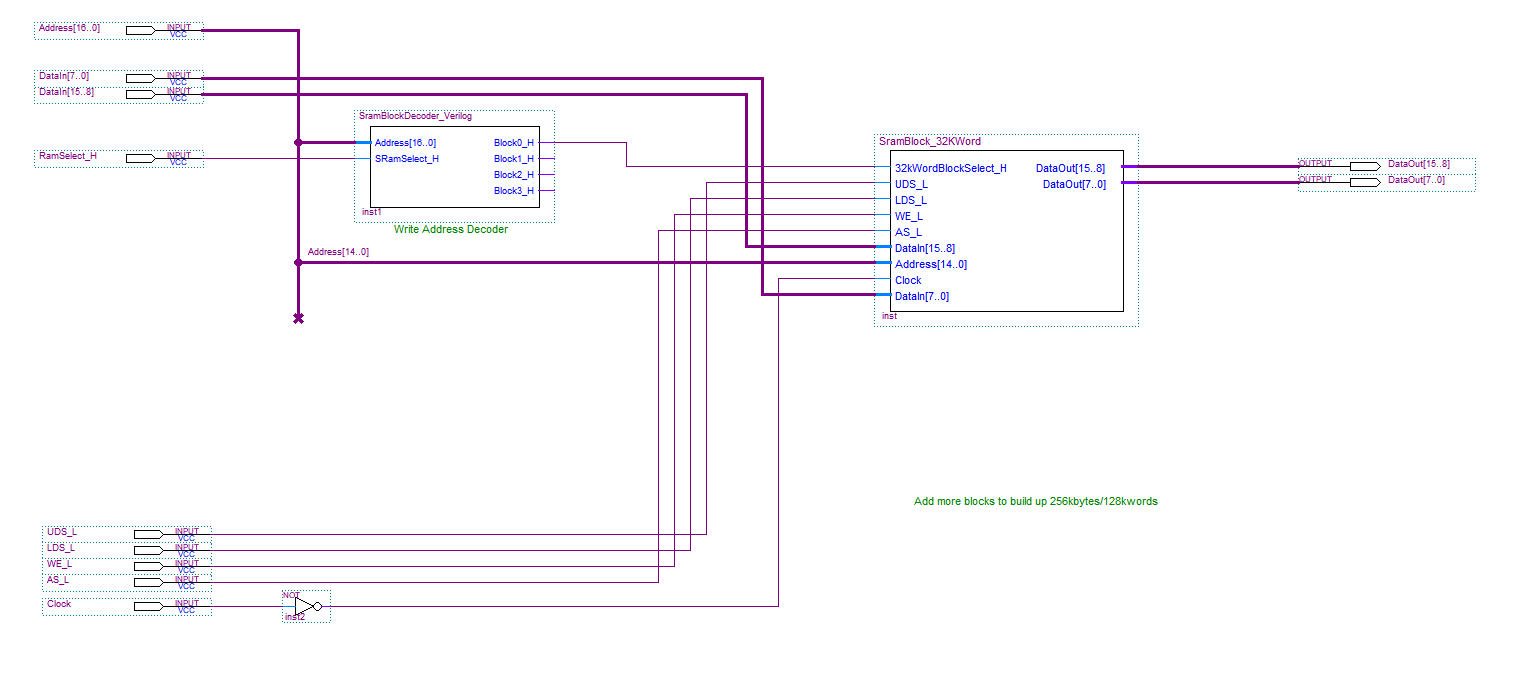
* We created a 16Kword Rom whose contents will be initialised by Quartus at project compile time with a **debug monitor program contained in a ‘.mif’ file**. Then we interfaced it to the 68000 buses as shown above.
* Note we didn't need the **R/W\*** signal since it Rom, so we cannot accidentally write to it.
* We also didn't need to use the **UDS\*/LDS\*** signals from the 68k since we are never writing to the Rom and whenever the 68k performs a read, the Rom will output all 16 bits of data leaving the 68k to pick up whatever half or both halves of the Rom data are being produced at that address.
* Finally, we only allow the output of the rom to *drive* the 68k's data bus (via the DataOut[] bus above) when the RomEnable\_H signal is activated. For all other situations it is tri-stated, allowing other memory and IO devices to drive this bus without conflict. This is typical of what we do with all devices connected to a CPU's data bus, they each have their own tri-state buffers that are only enabled when the device they are connected to is being read by the CPU - this includes the 68k processor we are using here.

**Step 2 - Creating 256k bytes of On-Chip Ram**

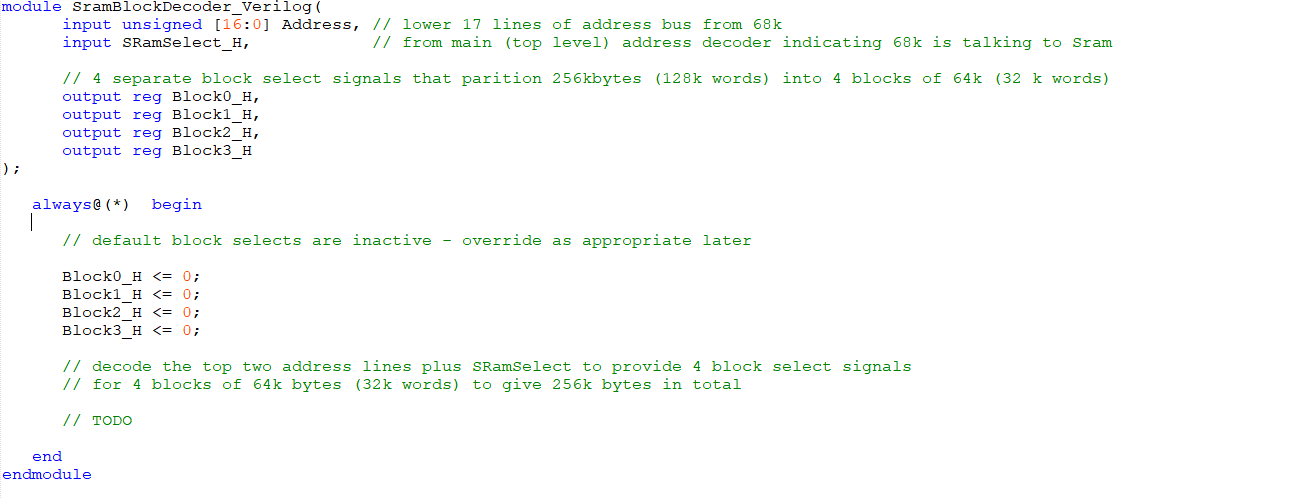
* Our debug monitor needs some Ram for variables and stack etc and so we can download programs to it. The DE1 has enough on-chip ram for 256k bytes.
* Going back to the top level schematic, we see the on-chip Ram block below.



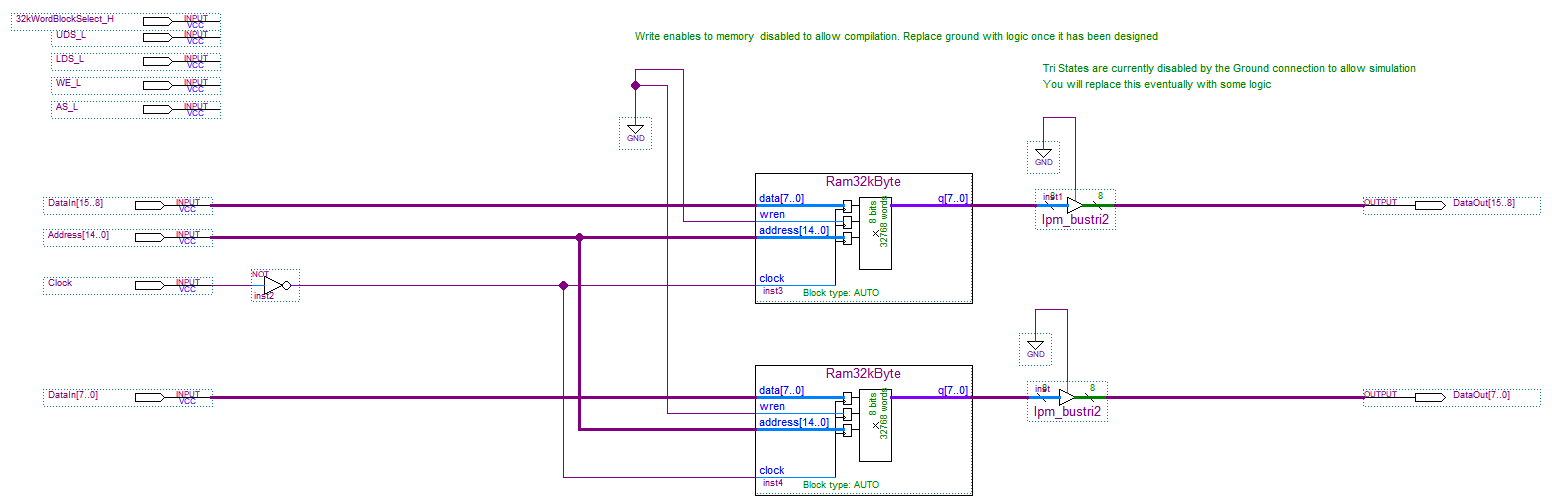
Double click the symbol above to open it and you should see this



This is the outline/template for 128k word block of memory composed out of 4 x 32k word blocks (*just for fun*). Add/Wire up the remaining 3 blocks to build up the 128k word and then modify the sram block decoder to provide select signals to the 3 extra blocks. The code is show below – You **MUST** make the alterations to the code to produce the **Block0\_H …. Block3\_H** signals (*to split the 128kword space into 4 blocks of 32kwords*) when the CPU outputs the correct address and **SRamSelect**\_H is activated by the main address decoder on the top level schematic.



If you double click on the SRamBlock\_32kWord symbol in the above schematic, you will see something like this.



This circuit contains **2 *single port* 32k BYTE** synchronous On-Chip RAM components (generated in exactly the same way as we generated the single port ROM in Step 1 using the IP Catalog). One device is connected to the lower half of the data bus, the other to the upper half. It has been done this way to enable the 68k to write a byte to one or the other memory device (or a word to both at the same time based on the value of UDS/LDS). *You can double click on them if you like, to open up the mega wizard to see how they were created.*

**Some Notes on Synchronous Memory devices**

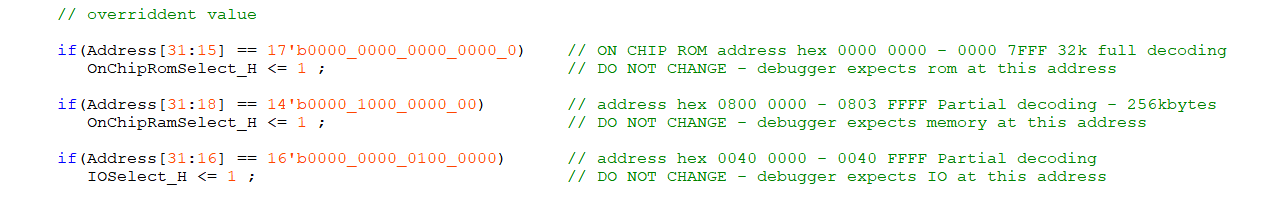
* Each device has a write enable signal (wren - active **high**).
* With every clock, the memory device carries out either a read or a write operation. It’s important therefore to ensure that wren is only set high when the 68k is issuing a write with a valid address directed at that half of the data bus. Reads are harmless – writes change data, so don’t get this wrong. Your Write signal should only be asserted when the address issued by the 68k is valid for this block of memory and the 68k’s r/w signal is indicating a write and the relevant UDS/LDS is asserted. Don’t make the mistake of enabling both lower and upper byte memory chips when the 68k is only writing a byte to only one of them.
* Each device has a 15 bit address bus (for 32k locations), plus an 8 bit data (in) and data out (q) bus, allowing each device to be accessed (for read or write) either independently of each other or together.
* Each device has it's output connected to a tri-state buffer so that it only drives the 68k data bus when it is being actively read. Think of this as being similar to a normal memory Output Enable (OE) when reading.
* We have to design the additional logic to make sure that each device and the tri-state buffers are enabled appropriately.
* Your task is to fill in the logic.
* Make sure your circuit copes with the 68k trying to write to upper (only) or lower (only) bytes or full words to the ram.
* Design the circuit to deal with this by asking yourself the following questions:

1) I want to be able to **read** (wren = 0) and turn on the output tri-state buffers when the signals above are ????

2) I want to be able to **write** (wren = 1) and the output buffers are off/tri-state when the the signals above are ????

**The Main Address Decoder**

Note the main address decoder for the 68k system on the top level schematic already includes Verilog code to produce the on chip rom and ram enable signals (see below)

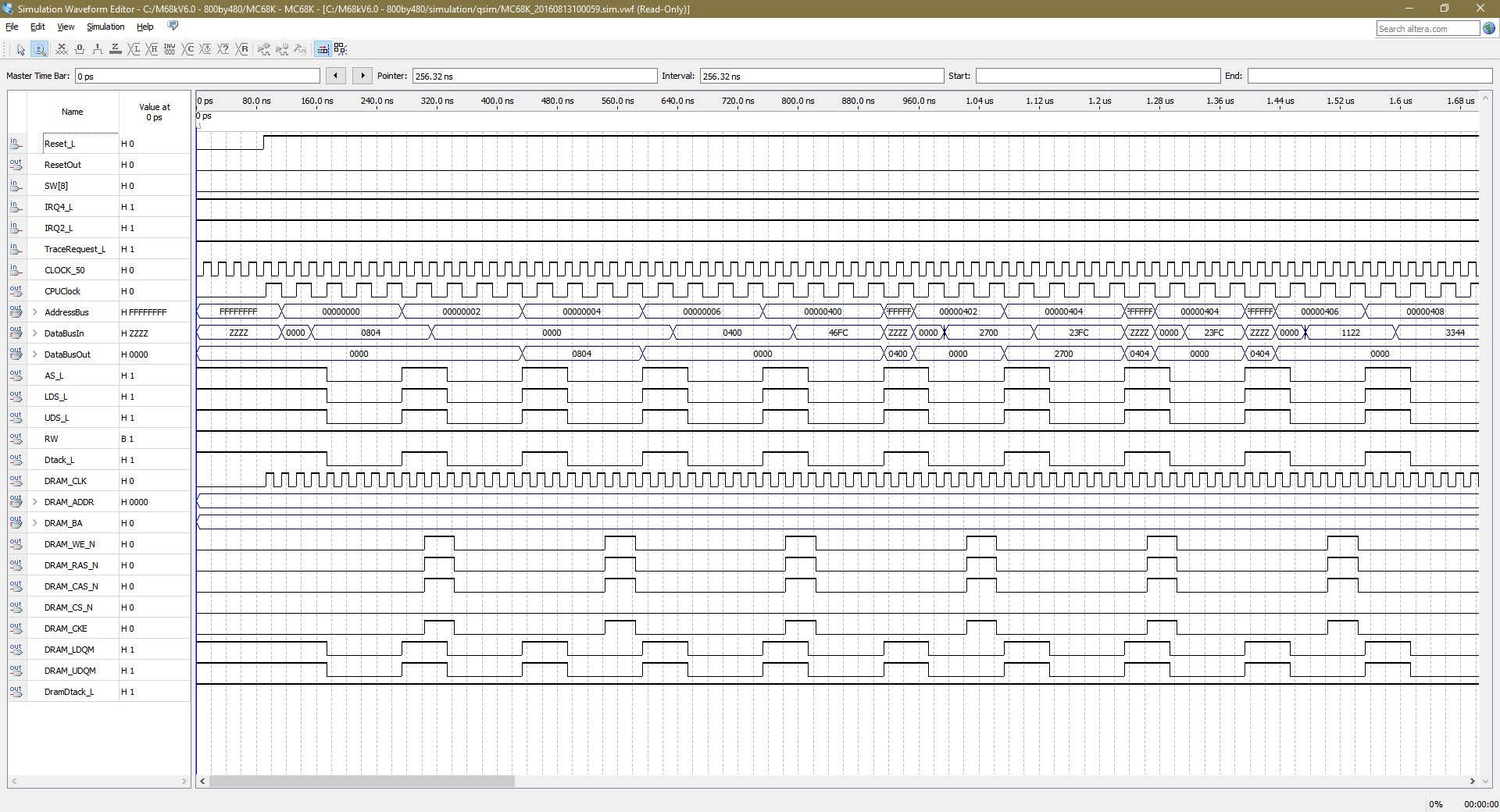
****

**Compile the Quartus project** and correct any errors.

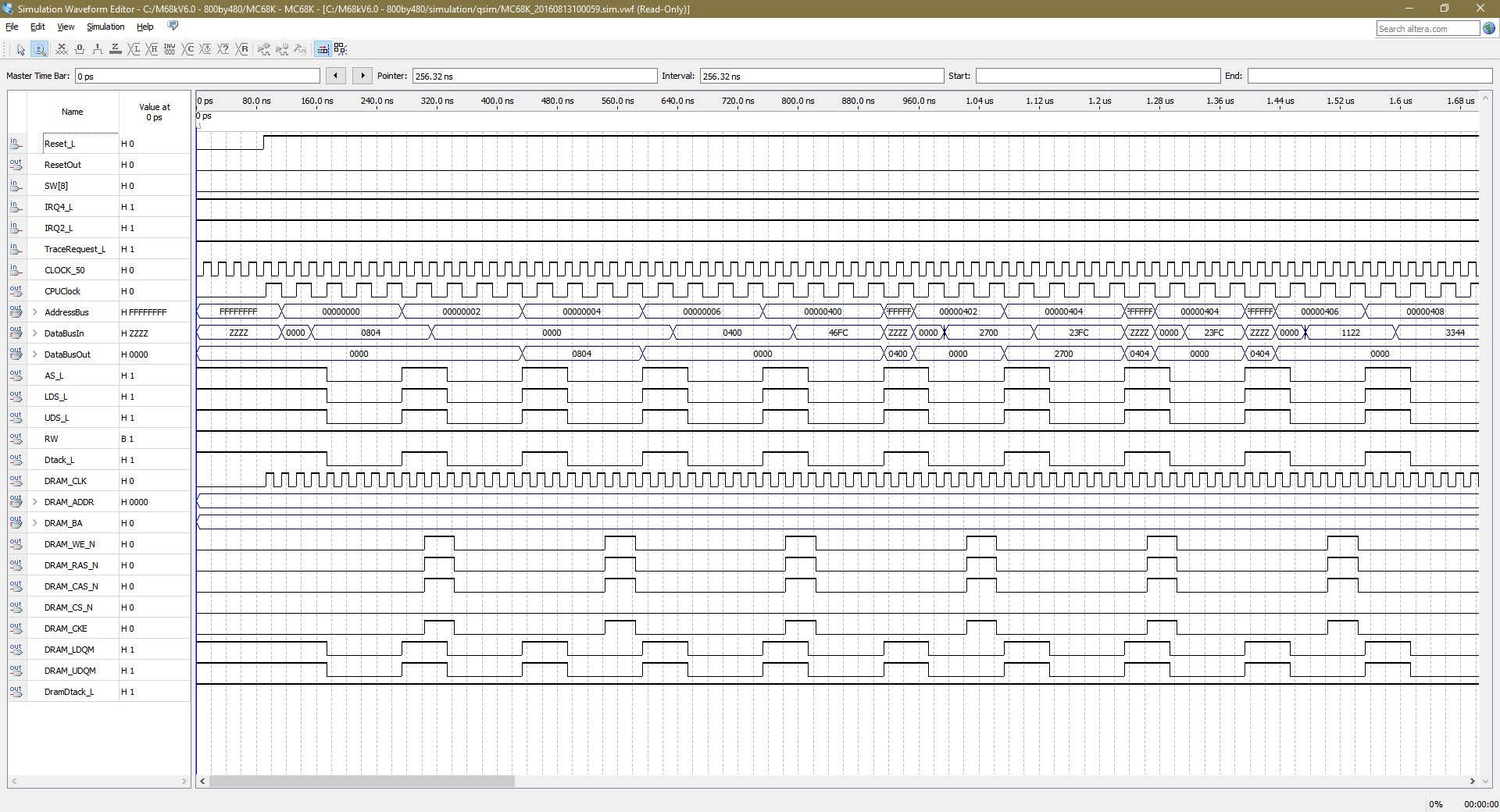
* Once you have completed your design, and it compiles without errors download the new .sof file to the DE1. If eveything is correct, when you turn on the power, you should see a **basic system spring to life and you should be able to communicate with the processor via hyperterminal on your PC via the RS-232 to USB converter given to you by the Instructor**.

**Simulating the design to see what happens after a reset**

* Open the simulation waveform file “Waveform68k-V15.0-DE1SoC\_Short7us.vwf” and run a Functional simulation using the **built in simulator**. This will take a couple of minutes. You should check (in V18.1 at least) that in the simulator settings the HDL is set to **Verilog** (not VHDL)
* You will see what happens in the immediate aftermath of a 68k **reset** going high – logic 1, i.e. when the reset key (push button 0) is removed - see below.



* Notice how the 68000 accesses and loads it’s stack pointer with the value 0804 0000 read from locations 0 and 2. The data comes from the on-chip ROM that contains the debug monitor, via the DataBusIn signal. The debug monitor has code to store 0804 0000 (the top of the on-chip-ram) into location 0, knowing the 68000 will fetch this value after a reset.
* Next the 68k accesses the reset vector at locations 4 and 6 above to fetch a 32 bit balue to load into its program counter (the value 0000 0400) and then begins to fetch/run program code (the debug monitor itself) starting at address 0000 0400. The first 68k instruction after a reset : hex 46FC is read from location 0000 0400.
* You can also see other signals such as the CPU CLOCK, AS\_L, LDS\_L, UDS\_L, RW and Dtack\_L signals from/to the CPU. You can add any additional pins/signals you like to the simulation to view that signal.
* You will also see that each memory access takes 4 CPU clock cycles (states S0-S7 as per the lecture 8 at 25Mhz). Notice also how Dtack should go **low** with each memory access to avoid wait states.



The Dtack\_L signal is generated by the Dtack generator on the top level schematic and can be *varied* to slow down the processor when talking to slower memory/peripherals. The default set up at the moment is that the Dtack is produced immediately the processor issues a valid address; the assumption being that the memory/peripheral the the CPU is talking to at the time is fast enough to run the CPU at full speed.

We would only need to change this if we interfaced slower memory etc to the CPU and we needed to introduce **wait state**s - it's unlikely the dtack generator will need changing at this stage but this will change in later assignments.

**Step 3 - Compiling and Embedding the Debug Monitor**

When we generated the on-chip Rom in Step 1 of Part A above, you were given a '.mif' file that contained a pre-compiled Debug Monitor that was placed into Rom by Quartus when we compiled the system. We are now going to see how that program was created.

Using the 68k C compiler, create a *new project* and add the files as shown in the image below (*most of these can be found in your Quartus project within a folder called Programs/DebugMonitorCode, one of them comes from the IDE lib folder for the C compiler*). Essentially there are 4 files that need to be added to the project:-

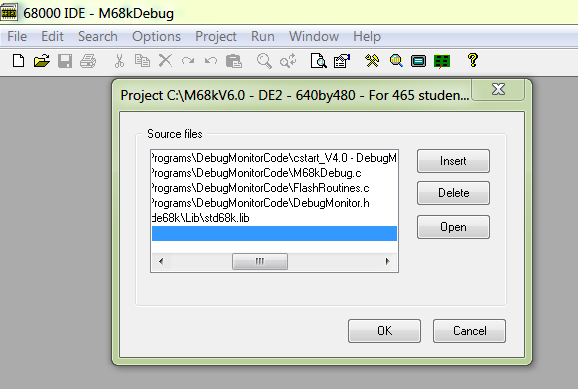
**M68kDebug.c** - the code for the debugger itself including a disassembler.  
  
**Note** a smaller debug source file called **M68kDebug (no disassembler).c** is also available which does NOT include a 68k disassembler but is otherwise identical. This compiles to much smaller object code which leaves more room in the ROM for your own code in later assignments (if needed).

**DebugMonitor.h** - a header file for the project

**Std68k.lib** - a C standard library file (*this will be in the C compiler LIB folder for the IDE68k compiler - wherever your installed that on your computer*)

**Importantly** – the following file, **MUST** be placed at the top of your project list (see image below), you can move it there afterwards if required. **Ignore** the **flashroutines.c** file shown below

**cstart\_V4.0-DebugMonitor.asm** - A 68k assembly language program that contains all the code that is called immediately after a reset, i.e. boot code, initialising stack pointer, program counter, interrupt vectors and code to call the debug monitor itself. You can see the effect of this code in the simulation waveform we looked at in step 2 when the reset was taken high. This file has to be **first** in the list as the code within it needs to reside at location 0 in Rom

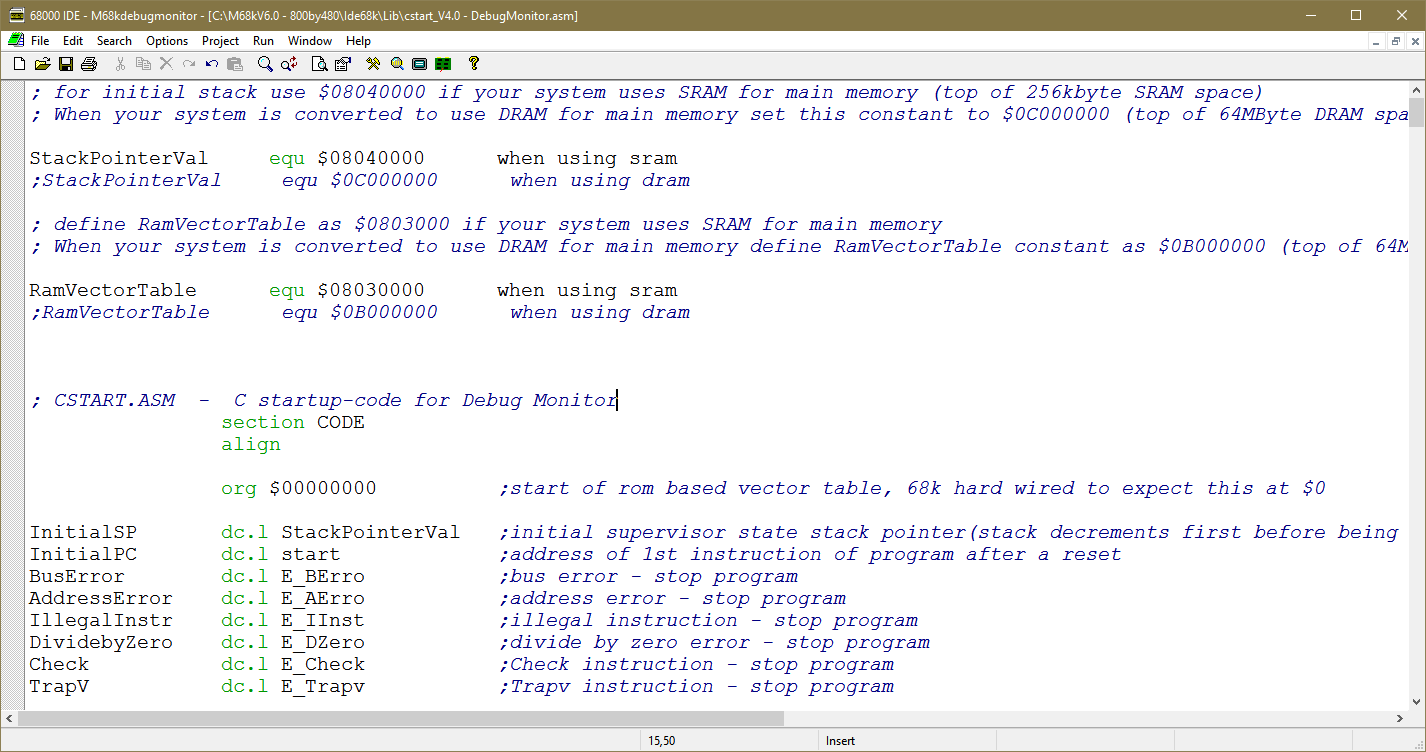


Save the new project using the name M68kDebugMonitor (it will create an M68kDebugMonitor.prj project file which saves all the settings). Now make sure this project **compiles** without errors.

If it is successful, a file called M68kDebugMonitor.hex will be created somewhere depending upon the file options you have set up for the compiler in Options>Directory->Default Directory

The cstartV4.0-\*\*\*\* file looks like this i.e. contains code starting at location 0 i.e. s dictated by the **org** or **origin** statement . You can see the code startes off with constants defined for the default stack pointer and default program counter at locations 0 and 4. There’s a whole bunch of other constants too, e.g.the address of a routine to handle bus errors, address errors, illegal instructions etc and, later on, constants that direct the CPU to the 7 interrupt request vectors.

You’ll notice that there are 2 **equ** statements at the top StackPointerVal and RamVectorTable these are set up to values inside the On-Chip Ram. Later when we switch to Dram, we’ll have to change these values to the “**commented out**” values immediately beneath (just a heads up)

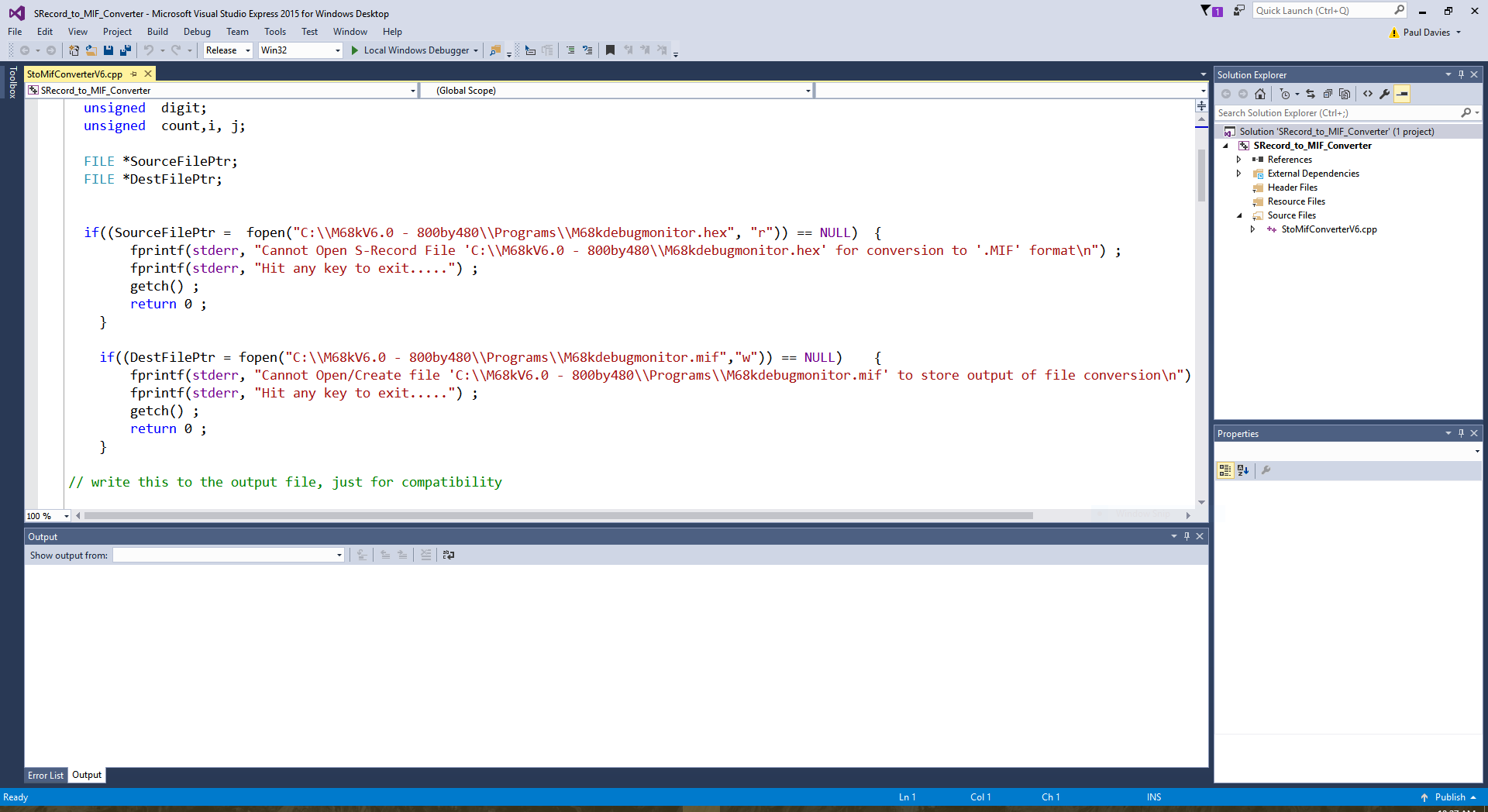


Once you have compiled the debug monitor, we need to turn the '.hex' file into a '.mif' file that Quartus can process and place into Rom.

**Using Visual Studio C++**

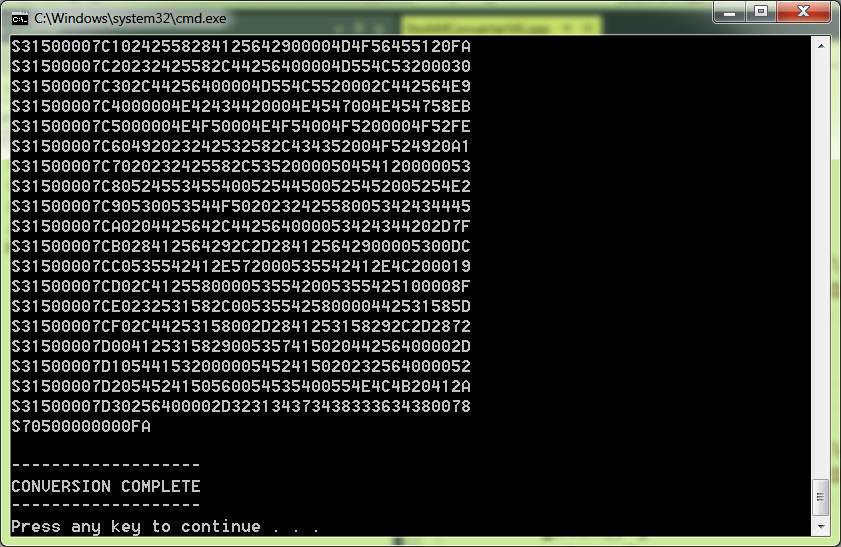
In your Quartus “Programs” folder, you will find a simple Visual Studio project Called SRecord\_to\_MIF\_Converter that will read a 'hex' file (produced by our ‘C’ compiler above) and convert it to a 'mif' file for use by Quartus.

Open that Project as shown below. If you can't open the project because you have an older version of visual studio, just go and create a general/empty project and add the StoMifConverterV6.cpp code to the project.



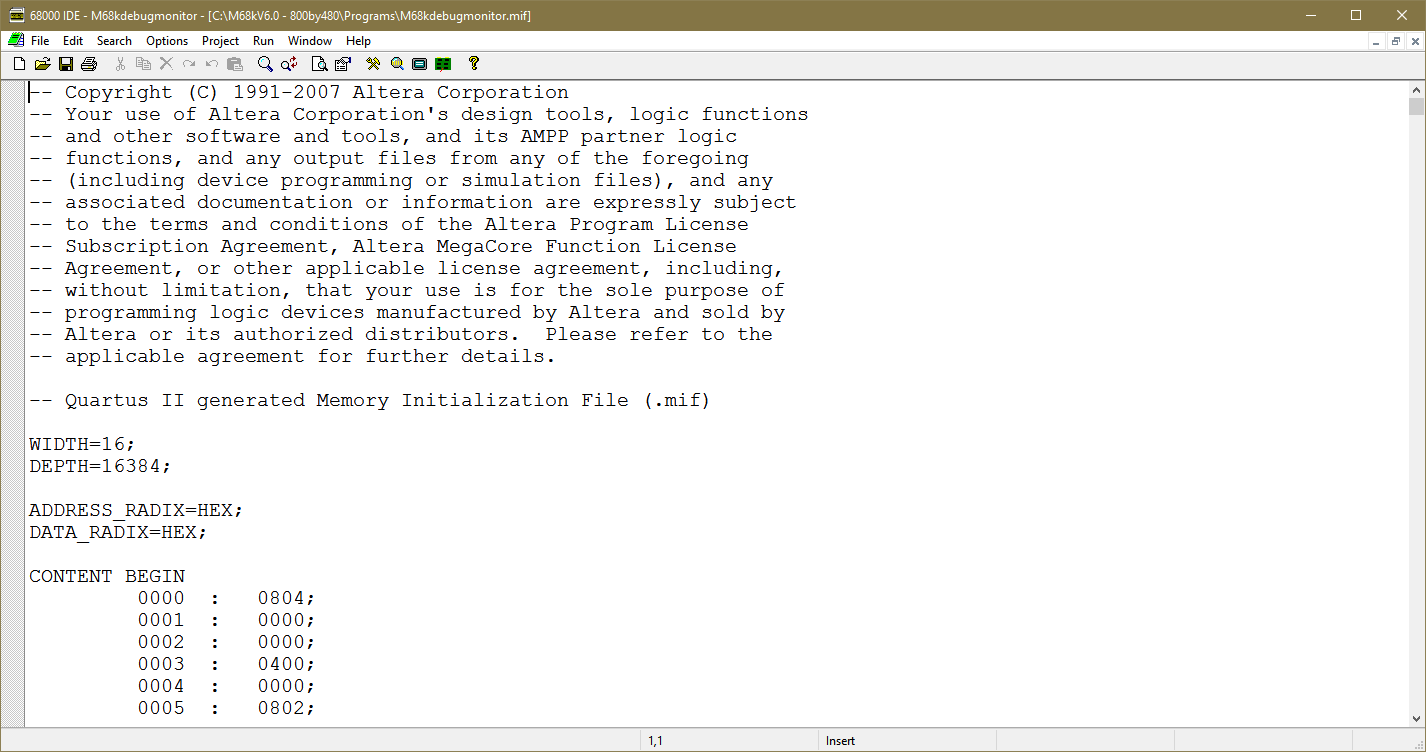
The program opens one (.hex) file, converts it and writes to another (.mif) file (see the **pathlists** contained in the program above) – you may need to change these to suit.

Rebuild, then run the Vis C++ project, you should see *something* like this produced on your screen (this one used the M68k **with** the disassembler) which shows the hex file (in the form of SRecords) being processed. This one shows **S3** records which contain a 4 byte address e.g. hex **0000 7C10** etc, but you may see an **S2** record which uses 3 byte addresses e.g. **00 7C10** or an **S1** record which uses 2 bytes e.g. **7C10.** (**S7** records are used to terminate an **S3** record etc)



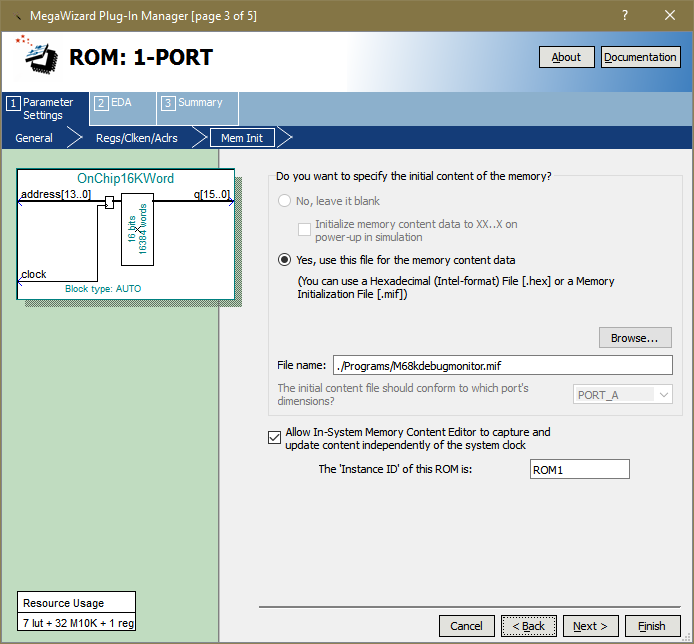
**REALLY IMPORTANT** **to check each time**: Remember the On-Chip Rom is only 32k bytes (16k words) in size, so make sure your debug monitor code will fit. If the address shown in the program output **above** ends up being greater than **00007FFF** (32k bytes) your program is **too big** and won't fit, so shorten it (you could use the debug source file **without** the disassembler which only half the size). This will ***only*** be an issue later when we modify the debug monitor to add the memory test code. In the illustration above the code fits because the last address is **00007D30 + 16** bytes i.e. **00007D3F** which is less than **00007FFF**.

The '.mif' output will be written to another file and will look like this (which is **BIG**). You can see in the example below the data starting at location **0000 = 0804**. In this case address 0 means relative to the start of the Rom which itself starts at location 0 in the 68k memory map. Make **sure** you see this data otherwise it won't appear in the ROM. You can see the initial SP values **0804000** at locations 0 and 1 and the initial PC values **00000400** at locations 2 and 3.



In essence, every time we change the debug monitor source code, we follow these steps:-

1. Recompile the code in the IDE68k compiler to create the Hex (SRecord) file.
2. Use Visual Studio to convert the hex to a mif file.
3. If necessary double click the 16kWord On-Chip Rom and adjust the pathlist to the new ‘.mif’ file we have created in 2) above.

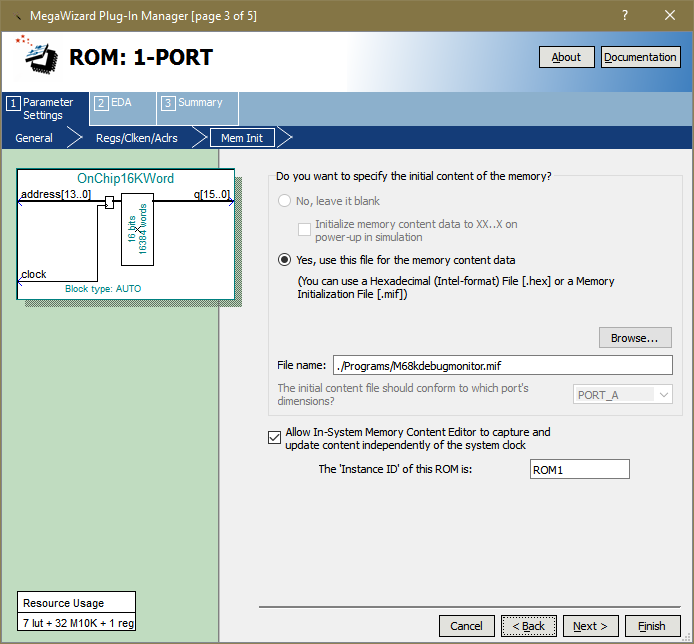


1. Recompile the Quartus Project (but also see next section below as a quicker solution).
2. Download the ‘.sof’ file to the DE1.

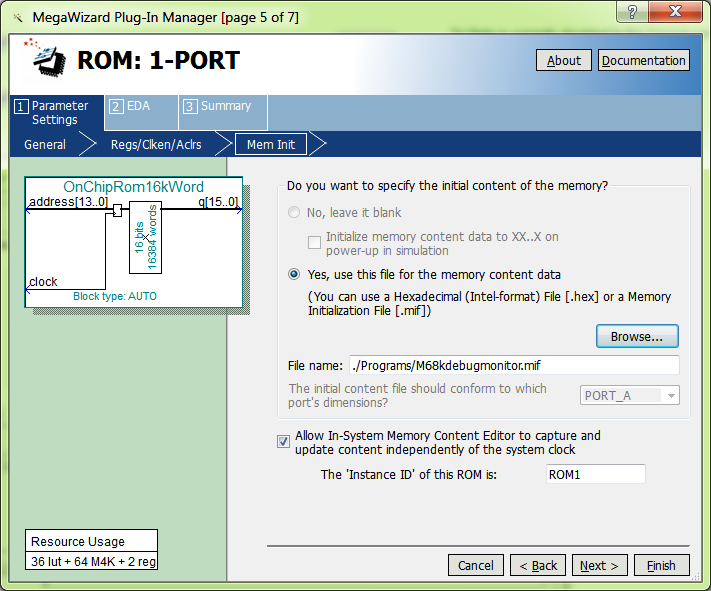
**How do we download the MIF file to the DE1?**

We can download the mif file to the Rom in one of two ways

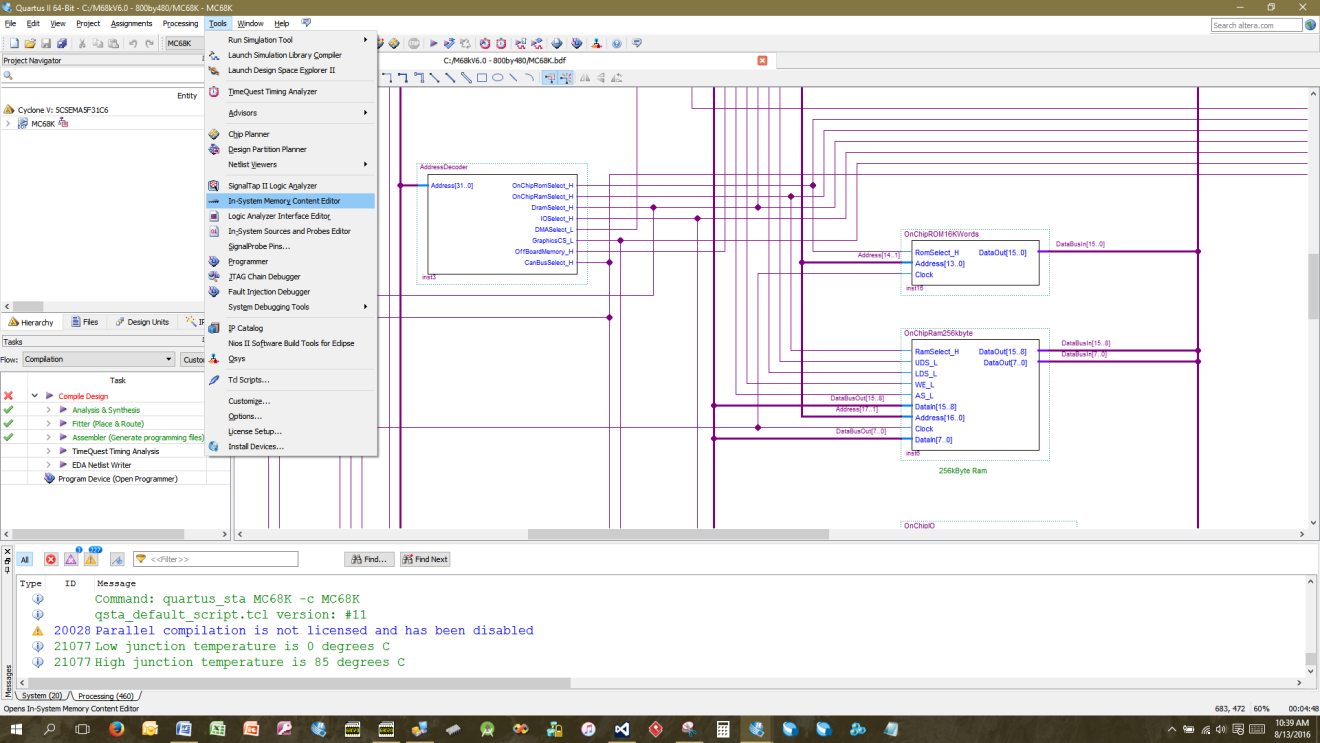
* The slow way is recompile the Quartus project so that the new compiled debug monitor code becomes part of the '.sof' file we download to the DE1 using the Quartus Programmer. However recompiling Quartus projects can take several minutes but it is the way to make the code permanent.



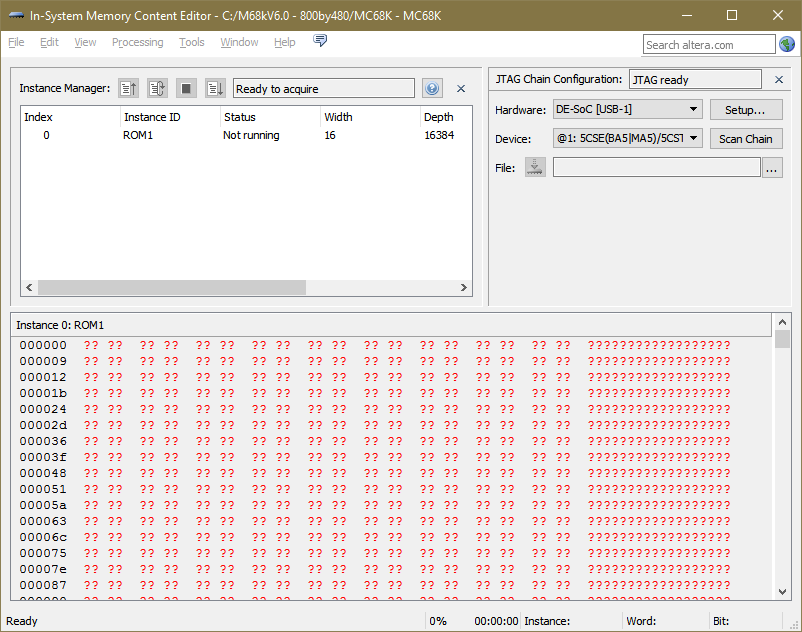
* A faster way, is to get Quartus to download the debug monitor code directly to the **rom** while we are connected to the DE1 board (via the Quartus Progammer).
* Remember in step 1 we enabled this feature and gave the rom the name **ROM1**.



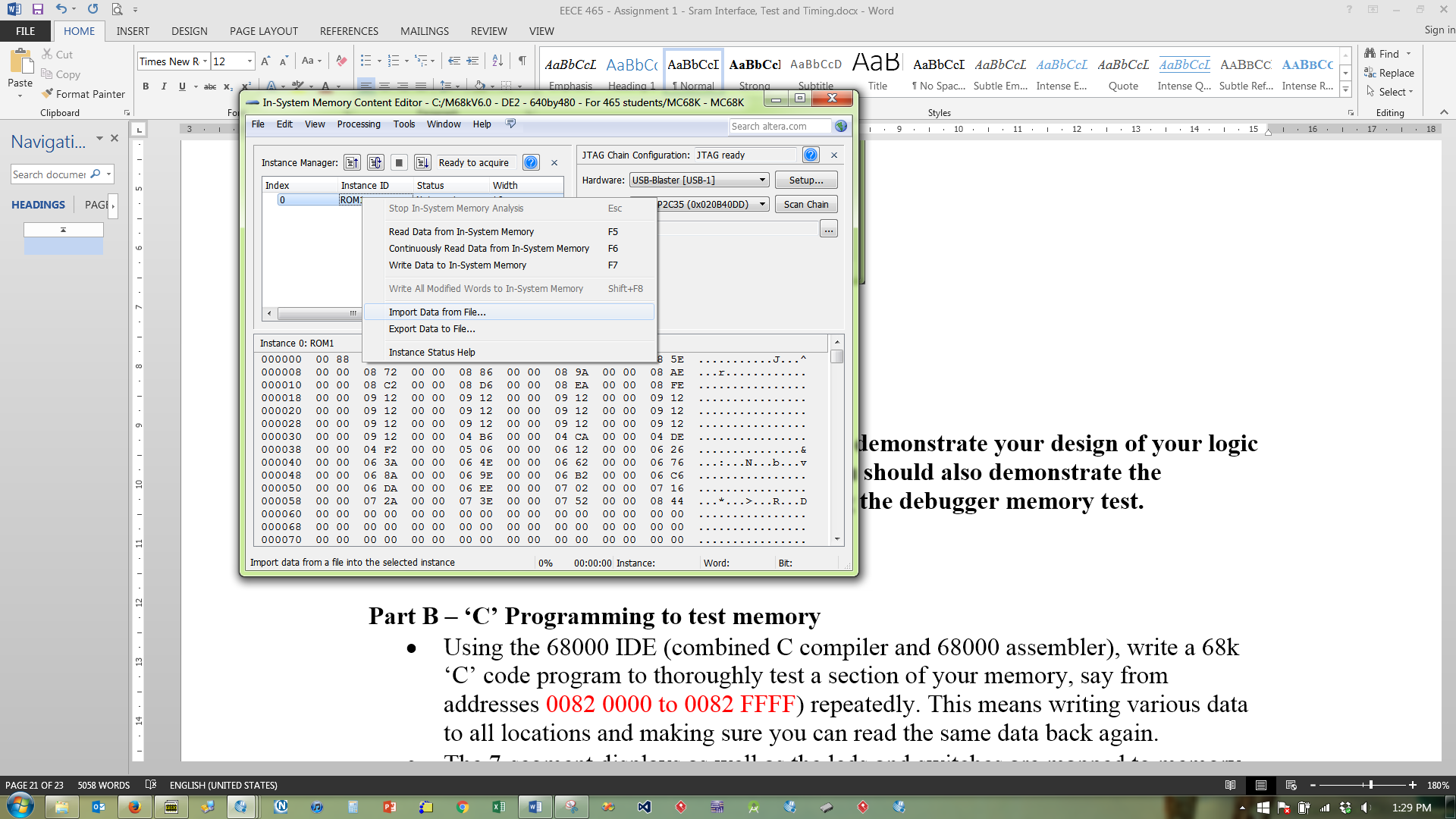
* With the Quartus programmer window open and the ‘.sof’ file **downloaded**, go to the main Quartus Window and click menu Tools->In-System Memory Content Editor



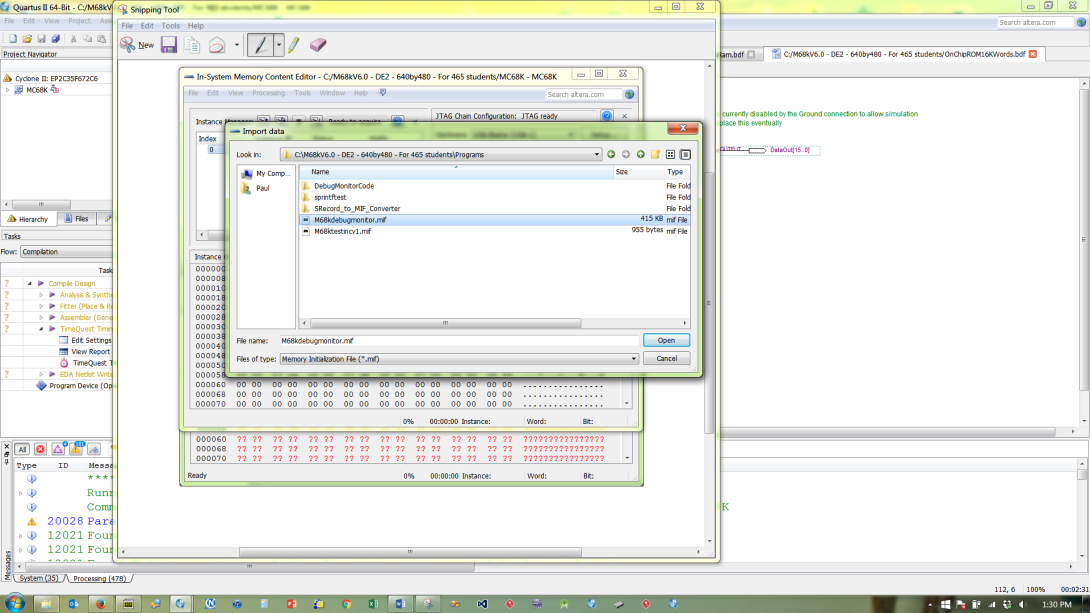
This window opens



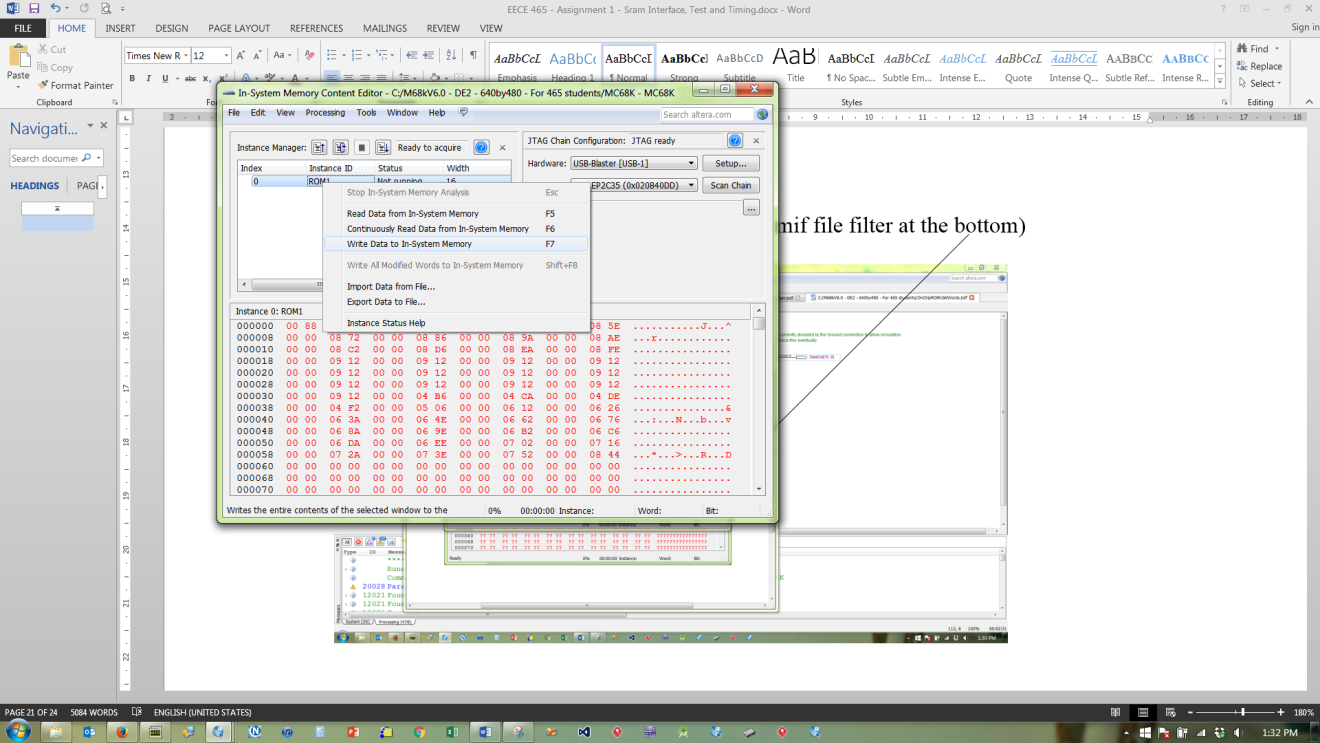
Right mouse click on the **ROM1** instance above and select "Import Data from File"



Select the ‘.mif’ file required (remember to set the mif filter at the bottom)



Once the file has been imported, right click on **rom1** again and select **Write data to the In-System Memory** and the contents of the rom will be updated inside the DE1 by Quartus (*remember to press reset on the DE1 after this, as you will be altering the debugger code while the 68k is still running it*).



This is a much faster way to check and download your 68k debug monitor code, but the change only lasts as long as there is power to the DE1.

To make the change **permanent**, (so it is part of the .sof file you download) you will have to recompile your Quartus project to use the new .mif file.

For Part A (Steps 1 - 4 above), you should demonstrate the design of your logic to interface the CPU to the *on chip rom* and on-chip *ram*, you should also demonstrate that you can *recompile the debugger* and use the *in-system memory content editor* to *download* a new debugger monitor to the system.

**Part B - 50% - Writing a Memory Test Program and adding it to the Debug Monitor**

Referring to the document on Canvas “Using IDE68k to Write C programs.docx” create a new user defined project in IDE68k with a **main()** (this project will use a different CStart.asm file as the code it produces runs *under* the debug monitor so doesn’t contain all the reset vectors etc and **doesn’t** produce code to run at address 0 (i.e ROM) but instead starts at 0x08000000 where out Sram is).

Inside the project write a C program to perform a memory test, by perhaps stepping through memory, writing values to successive memory locations in a particular range and reading them back to verify the memory works.

**What should my program do?**

1. **Your program should prompt the user for the option of carrying out a test using bytes, words or long words (i.e. 8, 16 or 32 bit read/writes)**
2. **It should prompt the user for a choice of 4 different patterns of test data to write, e.g. for bytes, the values 55, AA, FF, 00 etc. For words 5555, AAAA, FFFF, 0000 etc can be used and similarly for long words**
3. **It should also prompt for a start and an end address for the test (make sure that for word and long word memory test, the start and end addresses align to an even address, otherwise the 68k physically cannot handle it).**
4. **It should display progress e.g. show the addresses being tested and the pattern of data being read/written (perhaps for every 10k locations rather than every single location)**
5. **If an error occurs, it shold display the a message identifiying the location, the value written and the value read back. It should then stop.**
6. **It should display messages on the console window. Note there is a M68kUserProgram (DE1).c source file you can use that contains code to communicate with the user via the console window (i.e. basis input/output routines). You can copy/paste that code as required.**

Using the instructions in the above document, download the compiled memory test program into ram using the debugger ‘L’ command for load user program. Run the program with the ‘G’ command. Note the L and G command assume a download and program start point at location 0x0800000. Test that your SRam memory works.

**BE CAREFUL** running the test however, as the debugger uses some of our SRam memory for it’s own variables and stack space etc. (Remember the simulation earlier where the stack pointer was loaded with address 0804 0000) so only perform a check of memory between addresses in the range 0802 0000 – 0803 0000 which are not used by the debugger.

Once you have verified that the memory and your memory test code works, we can integrate this code into the debug monitor.

**Integrating the Memory Test into the Debug Monitor**

Re-open the IDE68k debug monitor project where you will find code to perform a memory test (search for the function MemoryTest() in the M68kDebug.c source file), however it is not yet complete.

Your job here is to **integrate the memory test code** you wrote above into the debug monitor so that it behaves just like our stand alone memory test program did.

**If your code becomes too big, you can always use the version of the debugger without the disassembler which frees up a lot of space.**

Of course you will have to generate the new ‘.mif’ file using visual studio again (as we did before) to be able to download or recompile to the **rom** on the DE1 board.

Once you have written your memory test inside the debugger, download it to the Rom on the DE1 (*note the important point on page 14 about making sure your debug code fits inside the 32k space*) as we did in step 4 of Part A and use it to check some of the Sram. There is also a command built into the debugger (which you don't have to write) to allow you to dump and/or examine and change data in memory locations.

Your debugger memory test should use the Console to display progress e.g. test number, pass/fail results.

**Finally – Last Step**

Now that you have tested the debug monitor with the Memory Test code you should do the following

* Change the Debug Monitor so that it announces your **name** and **student number** on the Console when you reset the DE1 board (by pressing Key 0)
* Recompile and make sure it works by downloading the new “.mif” file as above
* Once this checks out, recompile the Quartus project to create a new “.sof” file that you can download to the DE1SoC board and check that your memory test code and your student names/numbers appear after pressing the reset.
* If the last step checks out, convert the “**.sof**” file into a “**.jic**” file and program this into the **EPCS serial flash prom** on the DE1Soc board. This means that your hardware design and the debug monitor code will auto load when you power up your DE1 board, i.e. you have made a “turn-key” system with hardware and firmware.   
    
  Details of the “**.sof**” to “**.jic**” conversion and programming the serial flash chip can be found in the DE1SoC User Manual (Chapter 8).
* **Important**: It’s not obvious in the manual, but you may have to **ERASE** the chip first before programming it (*if it has been previously programmed*) otherwise programming will fail later, this is because flash technology chips cannot just be “programmed over” with new data, they have to be erased first.
* **Note**: It appears as if the serial flash chip that holds the “.jic” file has changed on some revisions of the DE1 board and thus the instructions in the DE1 User Manual may or may not be telling you to use the wrong chip setup when converting from “.sof” to “.jic” file format.   
    
  I believe older versions of the DE1 use an **EPCQ256** device, more recent ones use an **EPCS128**. If you select the wrong chip, the conversion to “jic” will work but programming the chip will fail later. In addition make sure the **MSEL** switches on the underside of the board are in the correct position otherwise programming will again fail– see DE1 user manual.
* The serial flash chip is a fairly large chip on the underside of the board between top side switches and underside MSEL switches. Perhaps you can read what part number is etched onto it or just try either of the above two until one of them works (remember also the comment about erasing the chip above).

When you have completed the Lab submit the following files as part of a **SINGLE ZIP** file to Canvas. **Only one student in your group (if you worked with a partner) needs to do this**

* **The .sof file produced by your compiled design so that we can test that it can be downloaded to a DE1 and will work. It should announce your name and student number as part of the boot prompt.**
* **The schematic (.bdf) file for the ROM design.**
* **The schematic (.bdf) file the RAM (and any sub-schematics associated with that).**
* **The Main and Sram address decoder (Verilog HDL) files.**
* **The C source code for the debug monitor that contains your memory test.**
* **The is NO need to submit the .jic file.**
* **The “Lab Student Contribution Sheet” from Canvas for this Lab**