C16 CPU Documentation

1 History (why it is as it is)

In early 2003, I had finished the design of an STM-1/STM-4 framer. The next step forward was extensive testing, but how? Since I was only using 30% of my FPGA (a Virtex 100E on an Avnet board), I thought a micro controller on the FPGA would be the easiest solution. The plan was simple enough: download a free CPU core, combine it with the STM framer, and that would be it. A weekend or two should suffice. Well, not exactly.

The first try was an open Z80 core. I chose Z80 since I was programming a lot of Z80 assembler back in the 70s (after the Z80, I fell in love with the 68000). After downloading the core, I figured that it did not fit into my FPGA. After analyzing the situation, I came to the conclusion, that a 8080 would probably be small enough. Since I couldn't a suitable core, I wrote one myself, which was finished some weekends later. At some point in time - all instructions were working, but I had not implemented interrupts yet - I thought it was time to look for a C compiler. I had a small loader that would read intel hex records over a serial interface into the FPGA memory. After searching on the web for some time, I learned that most C compilers were requiring a Z80 rather than a 8080, and the few 8080 compilers I found had some limitations that I didn't like. At least the assemblers I found were ok, so I decided to write my own C compiler.

A few weekends later, it was already mid 2003, The C compiler for the 8080 was ready. Although I exercised some care in generating compact code, even small C programs generated quite some code, and I had only 8kByte of internal FPGA memory left. I analyzed the generated code, and found that the 8080 was not really made for C. For example, ANDing two 16 bit numbers would create a lot of instructions, like:

```
LD      A, C
AND     A, E
LD      E, A
LD      A, B
AND     A, D
LD      D,A
```

Even though (or actually because) the 8080 had quite a few registers, the compiler had no choice but to move operands back and forth between these registers. Contrary to common wisdom I came to the conclusion that a good CPU does not have as many registers as possible, but instead as few registers as possible. The reasons for this is that (1) in FPGAs, internal memory is about as fast as registers, and (2) for preemptive multitasking (which I had in mind from the beginning), a small number of registers leads to faster context switches, since all registers need to be saved and restored.

The next step was then to design my own CPU. Since I was no longer bound by existing compilers or instruction sets, I could design the CPU in order to suit the compiler, rather than to write a compiler that suits a given CPU. The approach I took was to (1) take the 8080 backend of my compiler and to rewrite it towards a hypothetical CPU in such a way that most elementary back-

end operations would need a single 8 bit instruction and (2) to design that hypothetical CPU in the FPGA.

The first decision to make was the number of registers really required. Looking at C expressions, it turns out that in most nodes of the parsing tree generated by the compiler consists of expressions with a left and a right argument. Thus I gave the CPU two registers called LL and RR; LL holds the left argument of a binary operator, RR the right argument, and the result would be stored in back in RR. For function calls and local variables, a stack pointer, SP, would be required as well. This leads to only three registers LL, RR, and SP. and that is enough.

The next question is that of addressing modes required. Another common wisdom is that a good instruction set is orthogonal, and this turns out to be as wrong as the believe that many registers are good. In fact, what the compiler really needs is sufficient addressing modes for the leaves of the parse tree (which are always constants and variables). Thus the instruction set should be rich in immediate addressing (e.g. for ++, --, and frequently used binary operators), SP relative addressing including pre-decrement and post increment for local variables, and absolute addressing for global variables. Orthogonality is not required for these addressing modes, it is sufficient to have immediate addressing for the RR register only for most binary C operands, while absolute addressing helps also for LL register if a variable is a left operand.

Another thing to get rid of was a flag register. Considering that in C you can have constructs like

if (x > y) as well as

z = (x > y)

it makes more sense to have an opcode for a binary operator '>' rather than a compare opcode CMP, which sets a flag that needs to be checked later on. The good old 68000 had such a set of opcodes (Scc - set according to condition cc). Thus the decision was to provide a rich set of comparison operators and only a limited number of conditional branches (JMP RRZ and JMP RRNZ - jump in RR is zero resp. non-zero) instead of a single compare instruction and a rich set of jump instructions. As a consequence, there is no flag register in our CPU.

The CPU operates on 16 bit quantities only; conversion to and from **char** is made when the operands are moved into or out of the RR and LL registers (rather than having the same opcodes for different sizes as with the 68000), and **long** is not supported. The reason for not supporting **long** is essentially FPGA size. A byte operand move into a register is either zero extended or sign extended, as dictated by the opcode. In the assembler, we use the notation RU (R unsigned) for a byte operand that is zero extended, RS (R signed) for a byte operand that is sign extended and RR for a word operand. Likewise LU, LS, and LL for the left operand register.

Most immediate operands and SP offsets can be short (8 bit wide) or long (16 bit wide) as to reduce the program size.

2 Installation

The CPU comes with an assembler, a C compiler, a simulator, and a few simple utilities for generating VHDL files for the internal memory of the FPGA, communicating with serial ports on a PC, and so on. Everything has been tested on Windows XP, but should also work on other Win-

dows versions as well as Linux. I personally prefer Linux, but the fact that my Xilinx tools work under Windows has kind of forced me to do the entire development on Windows.

2.1 Prerequisites

For Windows XP, you can use the **.exe** files provided.

For other Windows versions, the tools provided may or may not work without recompilation.

For Linux you need to compile the tools.

When compilation is required, you should have **gmake**, **gcc**, **bison**, and **flex**. The following sites are useful for getting these tools for Windows:

- www.mingw.org (gcc)
- www.gnu.org (gmake, bison, flex)

Even if you don't compile, I would recommend **gmake** and **gcc** at least. Our compiler does little type checking, so you should syntax-check your own files with gcc before running the compiler provided.

2.2 Directory Structure

The entire package contains the following directories:

- **asm** source code for the assembler
- **compiler** source code for the C compiler
- **doc** contains this document
- **memory** utility to create **vhdl/mem_content.vhd** and **vhdl/board_cpu.ucf** (Xilinx and Avnet board specific)
- **sim** source code for the simulator
- **vhdl** vhdl code for the CPU

2.3 Makefiles and Building the Base System

There are 3 different targets for the top level Makefile.

- **loader** a small program that loads a subsequent memory image from the serial port of the FPGA into the CPU's memory.
- **test** a small monitor program for testing various I/O functions of the FPGA
- **rtos** the same monitor as for **test**, but using a preemptive multitasking operating system

The anticipated development process is as follows.

- Copy the CPU package on your machine
- Either install **gmake** (recommended) or else perform the actions in the top level **Makefile** manually later on.
- Build the utilities if required (see 2.1 regarding when this is needed).
- If you have a Virtex E evaluation kit from Avnet (**ADS-XLX-VE-EVL**, $150), then the **vhdl** files are ok already. Otherwise, you need to adapt the top level vhdl file **vhdl/board_cpu.vhd** and the UCF file **vhdl/board_cpu.ucf** (for the Xilinx design flow) to your actual hardware. Note that the utility **memory/makemem** will overwrite the UCF file, so when you use a different UCF file, then you should use a different name for it, so that it will not be overwritten,
- Do **make loader** in the top level directory. This compiles **loader.c** (generating **loader.asm**), assembles **loader.asm** (generating a binary file **loader.bin**, an intel hex file **loader.ihx** and a list file **loader.lst**), and creates **vhdl/mem_content.vhd** using the utility **makemem**).
- Compile the VHDL code and download to the FPGA.

At this point, you should have a working system on a chip. When you connect to the serial port of the FPGA (115,200 kBaud, 8 data bits, no parity, no flow control) and reset the FPGA, the system should print the following on the serial output:

LOAD >

This means the system is ready to load the desired application as a series of intel hex records. Every intel hex record loaded will be acknowledge by a dot printed on the serial output. Corrupted characters or records are indicated by the message **ERROR: not hex** (invalid character received, check baud rate etc.) or **CHECKSUM ERROR** (rather unlikely to happen).

2.4 Building Applications

After the base system containing the loader is working, you can develop your own applications. Two applications are provided with the CPU: **test** and **rtos**.

To build the application **test**, just do

make test

which creates (among others) **test.ihx**, which can be loaded into the FPGA via the loader. **test** is a small monitor that has functions for displaying and modifying memory, setting LEDs on the board, reading the DIP switches on the board, and reading the temperature sensor.

I was initially using the **HyperTerminal** program shipped with Windows XP, but copying (intel hex) files to the FPGA was very slow (even though the baud rate was 115,200). Therefore I wrote the tty.exe program supplied in the package which dumps files much faster on COM1. **tty** works from the DOS command line (program **cmd** in Windows XP) pretty much like HyperTerminal in a window. tty is started as:

tty [filename]

If no filename is provided, then **rtos.ihx** is assumed by default. **tty** prints characters received from **COM1:** on the **cmd** window in which **tty** was started and sends characters typed on the keyboard to **COM1:**. The special character **^L** causes **tty** to copy the file **filename (**or **rtos.ihx** if no filename is provided as a command line argument) to **COM1:**.

Note that there are two different ways to use applications:

- Do **make loader** first, compile the vhdl, and download to the FPGA (typically to a serial configuration PROM). After that, you leave the FPGA as it is and load applications over the serial port to the FPGA.
- Do **make test** (or rtos, or another application), compile the vhdl, and download to the FPGA (typically to a serial configuration PROM only when the application is finalized). This method does not use the loader and is somewhat simpler, but it also takes much more time for the vhdl compilation after every change of the application. This method is fine if you just want to play around with the code, without developing new applications.

The rtos application is a stripped down version of n real-time OS kernel which has been described earlier (for the 68000 processor). See **os_book.pdf** for details.

3 Software Description

3.1 C Compiler

Synopsis: **cc80 [-l] memtop infile [outfile]**

Example 1: **cc80 -l 0x2000 loader.c loader.asm**

Example 2: **cc80 0x2000 rtos.c rtos.asm**

Function: Compile the C source file **infile** and create the assembler file **outfile**. The -l option creates a slightly different startup code intended for a loader, which copies itself to the top of the memory. memtop is the top of the memory (for instance, 0x2000 = 8k for FPGA internal memory, or 0xA000 = 40k when an external SRAM is used).

Limitations: Not too well tested

No support for compound (i.e. **struct**) function arguments.

No **long** data type

Name should be cc16 (a left-over from the Z80 compiler)

3.2 Assembler

Synopsis: **assembler infile [binfile [listfile[symfile [ihxfile]]]]**

Example: **assembler rtos.asm rtos.bin**

Function: Assemble and link the input assembler file and create (1) a binary output file (used by the simulator and by the **makemem** utility), (2) a list file (useful for debugging), (3) a symbol file (used by the simulator to display source level symbols in a nice way), and (4) an intel hex file (used by the loader).

Limitations: Can not link several files.

3.3 Simulator

Synopsis: **simulate binfile symfile**

Example: **simulate test.bin test.sym**

Function: Simulate **binfile** at instruction level.

Limitations: Can not simulate interrupts.

Comment: The simulator is useful for debugging the compiler and assembler. If something does not work, check if it works in the simulator. If it works in the simulator, then the error is in the hardware (vhdl). If it does not work in the simulator, then the error is in the compiler.

3.4 Makemem

Synopsis: **makemem binfile**

Example: **makemem loader.bin**

Function: Create **../vhdl/mem_content.vhd** from **binfile**.

Limitations: Output file name should be a command line argument rather than a fixed file name

3.5 Tty

Synopsis: **tty ihxfile**

Example: **tty rtos.ihx**

Function: Terminal program for Windows. Typing ^L downloads **ihxfile**.

Limitations: Baudrate is fixed to 115, 200 Baud

Data format fixed to 8 data bits, 1 stop bit, no parity

3.6 Bin2array

Synopsis: **bin2array binfile**

Example: **bin2array loader.bin**

Function: Writes a C array representing binfile to stdout.

Comment: Useful for e.g. providing a loader in an application, so that one application can load another application. See array **loader[]** in rtos.c for an example.

4 Hardware Description

4.1 board_cpu.vhd

This is the top level design file.

Adaptations to other boards that the Avnet board should be made in this file.

Essentially instantiates **cpu16.vhd**.

4.2 cpu16.vhd

Breaks down the system on a chip into 3 parts:

- **cpu_engine.vhd**. This module is the CPU itself, plus 8kByte onchip RAM.
- **input_output.vhd** This module contains the I/O functions of the Avnet board. You need to rewrite this module (and possibly the applications) for other boards.
- **bin_to_7segment.vhd**. This module contains a driver that continuously displays the program counter of the CPU. Useful for debugging the system on a chip.

4.3 bin_to_7segment.vhd

This module samples the PC of the CPU at fixed intervals and shows the value on a pair of 7segment LEDs. This function is specific to the Avnet board providing the LEDs.

4.4 input_output.vhd

This module provides a number of I/O functions that can be accessed by the CPU through the assembler instructions

IN **(port), RU** and

OUT **R, (port)**

The ports implemented are:

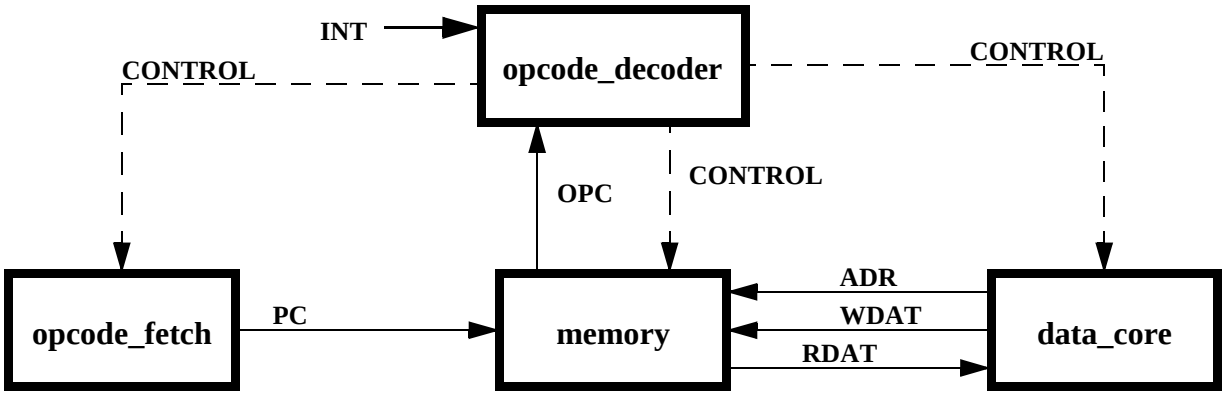
Port Function	IN	OUT
IN_RX_DATA: Data to be transmitted on serial output. In polled operation, you need to check **IN_STATUS** before sending data. Reading this port resets bits 4 and 0 in **IN_STATUS**.	0	

Port Function	IN	OUT
IN_STATUS: Status of serial I/O and timer Bit 7: not used (0) Bit 6: 1 iff timer interrupt enabled and timer interrupt has occured Bit 5:1 iff serial Tx interrupt enabled and serial Tx is ready to accept data Bit 4:1 iff serial Rx interrupt enabled and serial Rx has received valid data Bit 3: :not used (0) Bit 2:1 iff timer interrupt has occured Bit 1:iff serial Tx is ready to accept data Bit 0:1 iff serial Rx has received valid data	1	
IN_TEMPERAT: current value from temperature sensor (8 bit 2's complement in degrees Celsius) (Avnet board specific)	2	
IN_DIP_SWITCH: current setting of the DIP switch. (Avnet board specific)	3	
IN_CLK_CTR_LOW: current value of a 16 bit clock counter (low byte)	4	
IN_CLK_CTR_HIGH: current value of a 16 bit clock counter (high byte)	5	
OUT_TX_DATA: Data received on serial input. In polled operation, you need to check **IN_STATUS** before readiing data. Writing this port resets bits 5 and 1 in **IN_STATUS**.		0
not used		1
OUT_LEDS: Turns each of the 8 LEDs on or off. 1 turns LED on.(Avnet board specific)		2
OUT_INT_MASK: The interrupt maks for Rx, Tx and Timer interrupts. 1 means interrupt is enabled. Bit 7..3: not used Bit 2:Enable Timer interrupt (1 ms interval) Bit 1: Enable Rx Interrupt (receiver has received valid data) Bit 0: Enable Tx Interrupt (transmitter ready to accept data)		3
OUT_RESET_TIMER: Writing clears Timer interrupt		4
OUT_START_CLK_CTR: Start a 16 bit counter clocked at a rate of 20 MHz. Useful for measuring short intervals with high precision.		5
OUT_STOP_CLK_CTR: Stop the 16 bit counter		

The I/O ports are not in the focus of this document, so please refer to the VHDL files regarding their implementation. You may find the baudrate generator interesting due to its unlimited precision, and the Rx and Tx parts due to their very low size.

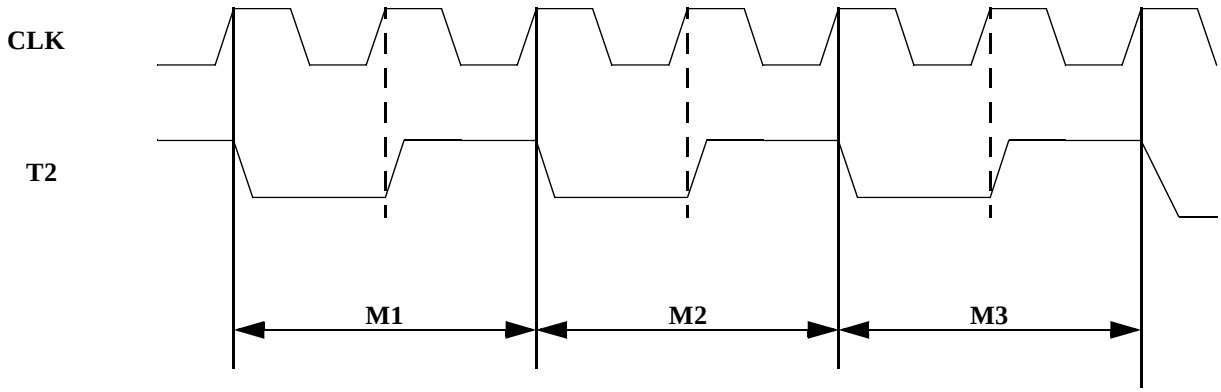
4.5 cpu_engine.vhd

This is the CPU itself. The structure of the **cpu_engine** is as follows:



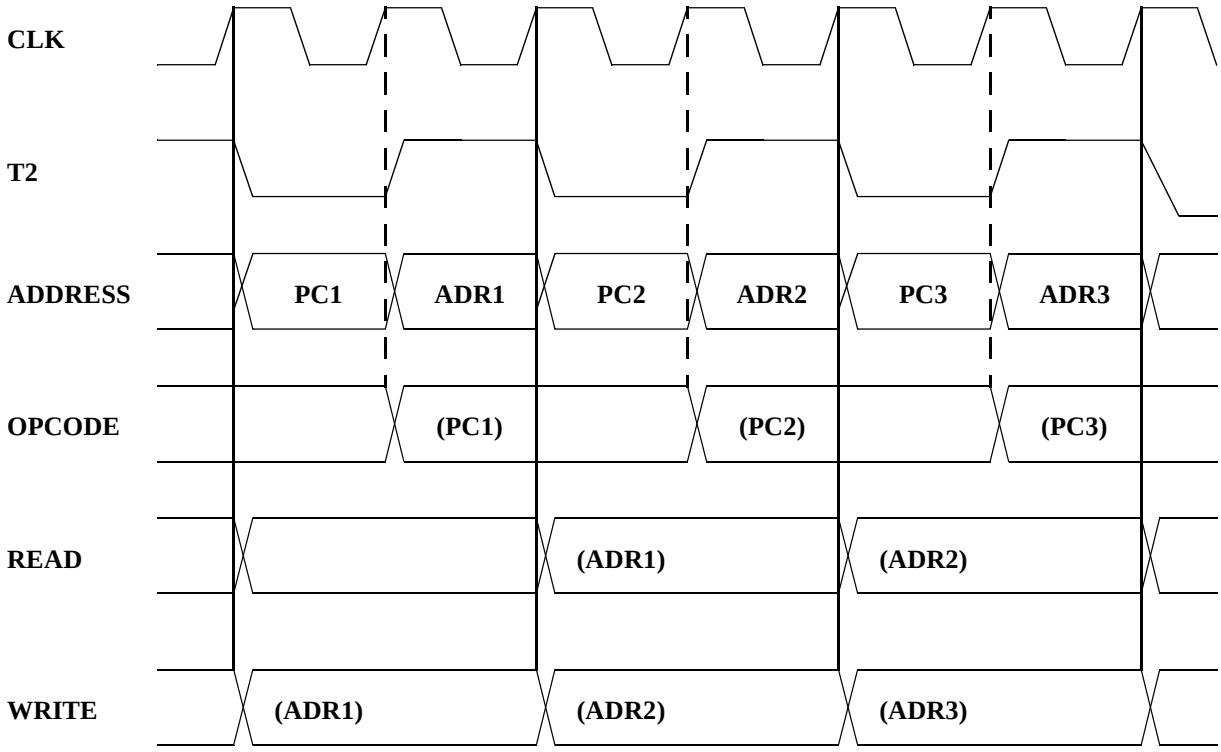
The memory signals are also extended to the outside of this module in order to connect to an optional external SRAM and to the **input_output.vhd** module.

The timing is as follows. All signals are clocked on the rising edge of the 40 MHz input clock. However, most signals are clocked on every second clock only. This is controlled by the T2 signal. The internal memory is dual-ported, but only to save address and data multiplexers. The first clock interval (say T1, or better 'not T2') is used for opcode reads, while the second phase (T2) is used for all other (that is, operand transfers; immediate operands are counted as opcode reads).

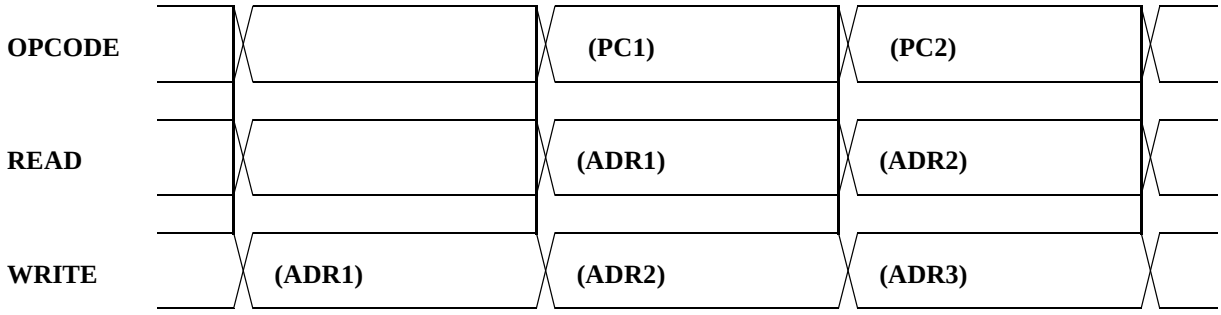


We call a full T2 cycle an M cycle; most opcodes use only a single M cycle (M1), opcodes with a short immediate operand require two M cycles (M1 and M2), and so on. The longest opcode is RET, reading the return address in M2 and M3, plus 2 M cycles delay from the PC to the execution unit. Opcodes without immediate operands (addresses or data) execute in a single M cycle. Some frequently used operations have a quick mode, where a 4 bit immediate operand is contained in the opcode; these opcodes have an immediate operand, but still execute in a single M cycle.

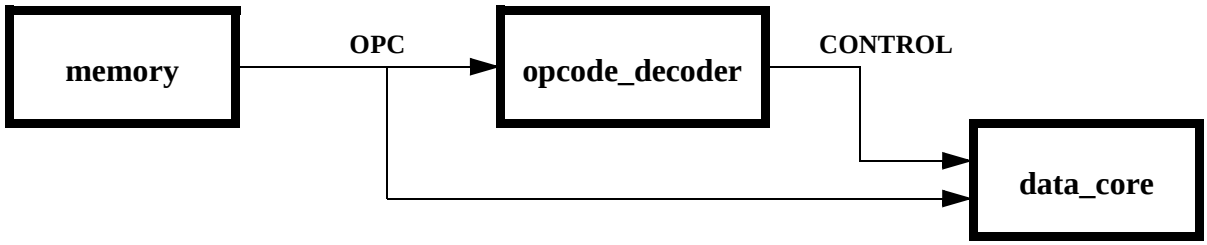
In the first half of a M cycle (T2 = 0), the program counter is placed on the memory address bus; this is always a read cycle. In the second half of a M cycle, the operand address is placed on the memory address bus; this could be a read cycle, a write cycle, or no cycle:



The opcode is clocked again at the end of an M cycle, so that, from the perspective of opcode_decoder and data_core, all signals change at the end of a M cycle, an the system looks like running at the rate of T2, rather than CLK: Therefore, with the exception of the signals above, all signals of the CPU are clocked at the end of T2 (they are still clocked with CLK, but only on every second cycle of CLK when T2 = 1).



The opcode_decoder requires one T2 cycle to decode an opcode. Therefore the result of opcode_decoder (the decoded opcode) and the first immediate operand (low byte of an immediate operand or address) arrive at data_core at the same time:



For instance, if opcode_decoder generates the control signals for M1 of the data core, it can assume that the low byte of an immediate operand or address is available already at the data_core. A read or write operation using short addresses (or SP offsets) can therefore start already in M1, while reads or writes using long addresses must wait until M2.

4.6 data_core.vhd

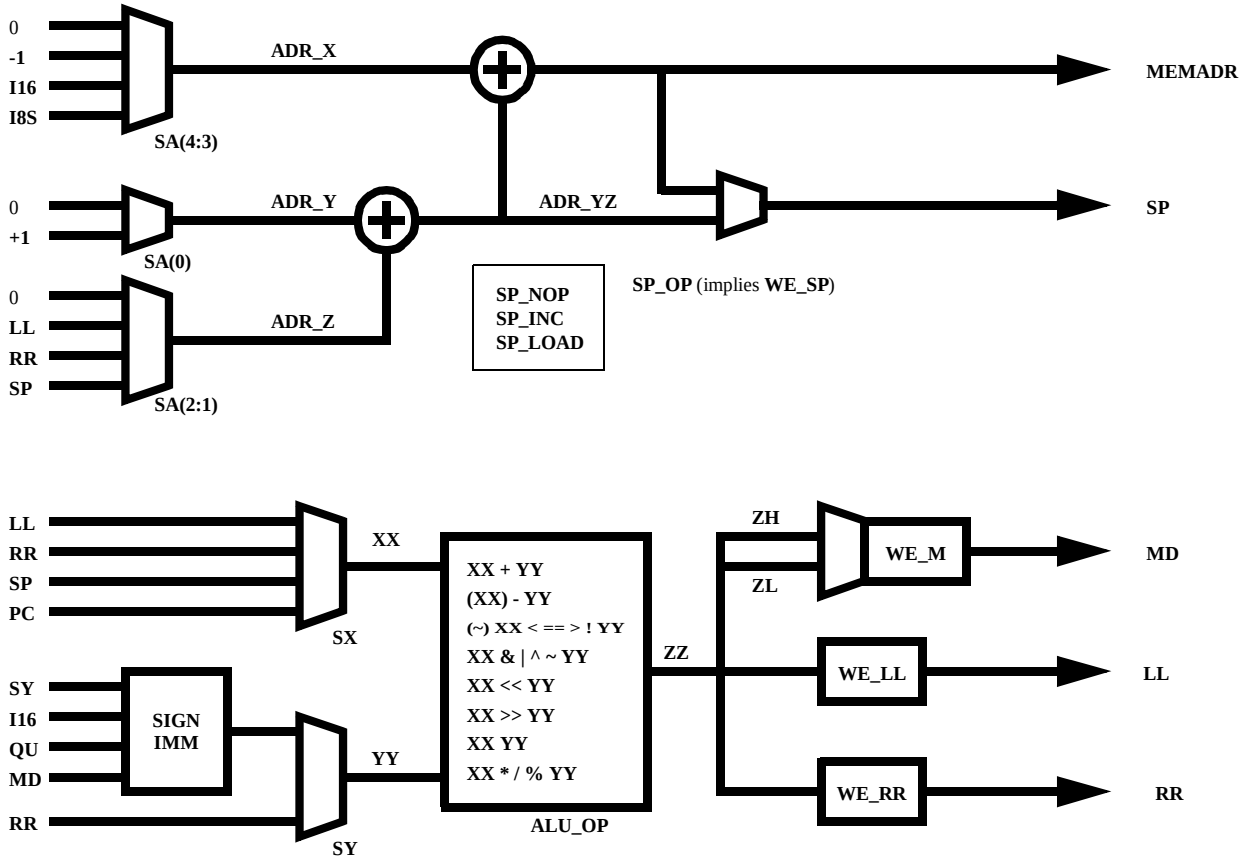
The data_core implements the data paths of the CPU. It consists of two separate parts: (1) the address calculation and (2) the ALU calculation.

The address calculation consists of two adders and a multiplexer. The inputs of the adders are multiplexed from different sources, so that a number of addressing modes are available. The input SA(4:0) controls the addressing mode. The signal SP_OP controls, whether the SP shall be updated with the computed address; this is used to implement -(SP) and (SP)+ addressing modes and also the MOVE x, SP and ADD x, SP opcodes.

Examples:

- **MOVE LL,(RR** SA selects (0) +(0) + (RR) = RR
- **CLRB -(SP)**: SA selects (-1) + (0) + (SP) = SP-1. In addition, SP_OP is set to SP_LOAD the that the SP-1 is loaded into SP. (**pre** decrement)
- **MOVE (SP)+,RU** SA selects (-1) + (+1) + (SP) = SP. In contrast to the previous example, SP_OP is set to SP_INC. Therefore the address is SP, but SP is loaded with SP + 1 (**post** increment)

Appropriate constants for SA are defined in **cpu_pack.vhd**. The middle multiplexer (0 / +1) is used for post increment addressing, but also for the second address (high byte, n + 1) of a word access to address n.



The second part of data_core contains the ALU. The ALU has two inputs, XX and YY. The operation of the ALU is controlled by ALU_OP and provides all operations required in C. Appropriate constants for **ALU_OP** are defined in **cpu_pack.vhd**.

The output ZZ of the ALU can be written to registers RR, register LL, or to the memory. A word write to memory is performed in two M cycles, multiplexing ZZ into its low byte ZL or high byte ZH. The ALU also contains a MIX operation where the (low) byte from a memory read in the previous cycle can be combined with the (high) byte of the present cycle.

The input to the ALU comes from two multiplexers that select the sources of the operand. The XX input is one of LL. RR. SP, or PC. The XX input can be ignored through the ALU operation **ALU_MOVE_Y**. Appropriate constants for SX are defined in **cpu_pack.vhd**. The YY input can be: SY_0 .. SY_3 (a fixed value that is independent from the opcode used for PC offsets), I16 (a collection of immediate operands with different sign-extensions), QU (quick: the lower 4 bits of the opcode) and MD (data read from memory or IO). Appropriate constants for SY are defined in **cpu_pack.vhd**. A number of sign extensions are provided through the SX signal, therefore the selection of YY has been moved into a separate module **select_yy.vhd**. The ALU operations are contained in **alu8.vhd**.

4.7 select_yy.vhd

See **data_core.vhd**.

4.8 alu8.vhd

Should actually be alu16.vhd. See **data_core.vhd**. In order to save slices, the multiplication, division, and modulo operations are implemented to do one bit at a time, thus requiring 16 instructions for a 16 bit multiplication, division, or modulo. The division and modulo results are accessible simultaneously, but C has no operator supporting this directly.

4.9 opcode_decoder.vhd

The **opcode_decoder** decodes an opcode into the signals required by **data_core**. It also contains a state machine for interrupt handling, which is controlled by an external interrupt signal INT, and by opcodes EI (enable interrupt), DI (disable interrupt), and HALT (halt operation until the next interrupt. HALTing the CPU with interrupts disabled will halt the CPU forever.

Note: The **EI** and **DI** opcodes do not directly enable or disable interrupts. Instead a counter is incremented resp, decremented. Reset clears the counter, disabling interrupts initially. The next **EI** enables interrupts. However, if a number of **DI** instructions are executed instead, then the according number of **EI** instructions are required before interrupts are enabled. This allows subroutines to disable interrupts independently of other subroutines, which simplifies software development. Executing too many (> 15 or so) **DI** instructions will cause to counter to overflow, which will crash your system.

4.10 opcode_fetch.vhd

This is the program counter of the CPU. It is controlled through PC_OP (generated by the opcode_decoder). The default PC_OP is PC_NEXT, which increments the PC. The other PC_OPs (which are defined in **cpu_pack.vhd**) set the PC according to immediate input data (PC_JMP), values popped from the stack on return from a subroutine (PC_RETH and PC_RETL), do nothing when an opcode requires more cycles that its length (PC_WAIT), the value of RR (PC_JPRR), or to a fixed address 0x0008 of the interrupt service routine

5 Future Plans

5.1 Wishbone Adaptation

This should be fairly straight-forward, but I am wondering if there is any interest in this CPU. Please feel free to comment if you think a Wishbone Adaptation would be worthwhile.