Versat Controller

Datasheet

January 29, 2018

Contents

List of Tables

List of Figures

1 Introduction

Versat controller is a low power programmable hardware fabric. The Controller is an Intellectual Property (IP) core. It is designed to be included in low power embedded systems that require simple algorithmic control, offering a programable solution.

The Controller is responsible for managing the hole system where it is integrated. It is programmable and features a minimal instruction set. The Controller is not designed for high performance computation, although, it is quite capable of perform algorithmic flow, which may involve some simple calculations.

Versat controller has a control interface and an instruction interface. The control interface is used to manage several peripherals in the system. The instruction interface is used to fetch instructions to execute. This instructions compose a program that when executed, perform one or more tasks.

2 Architecture

2.1 Controller

The Versat controller has a 8-bit minimal architecture to support simple calculations and managing some peripherals. Its accumulator architecture is shown in Fig. 1.

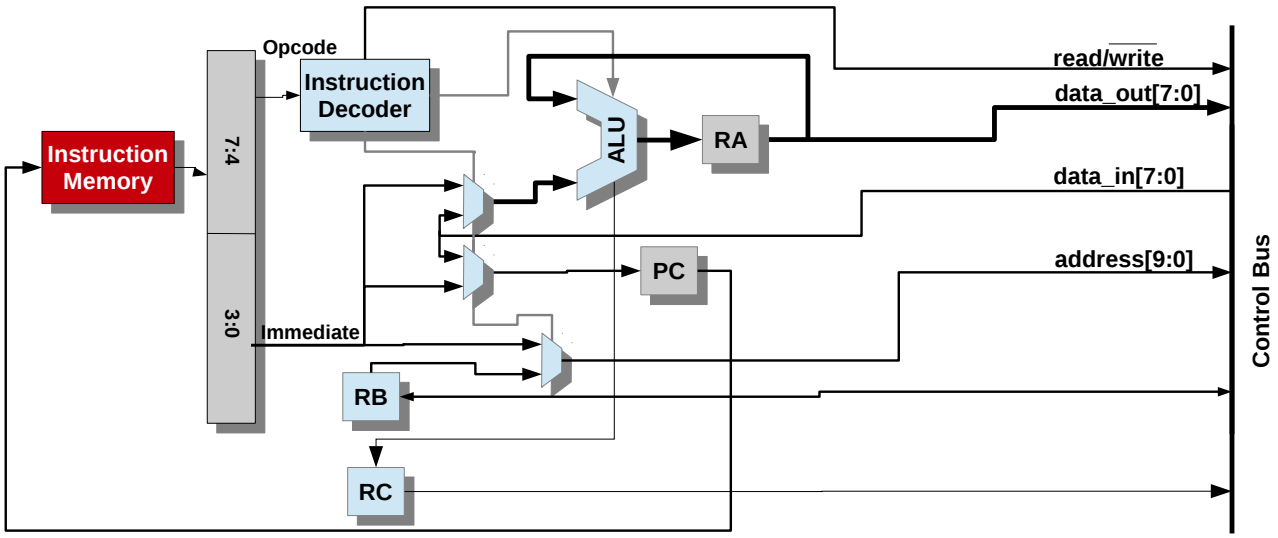

Figure 1: The Controller.

The controller executes machine code instructions. The instruction set is explained in section 4. The instructions must be stored in a Program Memory block. The fetched instruction is decoded; the operation code (opdocde) and immediate value are identified and the opcode is decoded to select the ALU operation, read or write modes.

The controller sees the other blocks in the system as memory mapped, and uses the RW bus to access its memory space.

The controller architecture contains 4 main registers: the accumulator (register A), the data pointer (register B), the carry save (register C) and the program counter (register PC).

2.1.1 Accumulator

Register A, the accumulator, is the main register in this architecture. It can be loaded with a value read from the instruction itself (immediate value) or a value read from the RW bus. It can perfom operations using as arguments its own contents and an immediate or addressed value. For that it is preceeded by an ALU block to compute its next value. Finally, its contents can be written to the RW bus.

2.1.2 Data Pointer

Register B, the data pointer, is used to implement indirect loads and stores to/from the accumulator, respectively. Its contents is the load/store address for the RW bus. Register B is in the memory map so it can be read

or written using the RW bus. This register has two times the data width, so it is addressable to the word level (most or less significant bits).

2.1.3 Carry Save

Register C, the carry save, is used to store the carry save bit. It is a read only register.

2.1.4 Program Counter

The PC contains the address of the next instruction to fetch from the Program Memory. The PC normally increments to fetch the next instruction or, to implement program jumps, it is loaded with an instruction immediate or with the register B value.

2.1.5 RW bus

The RW bus signals shown in Fig. 1 are described in Table 1.

Name	Direction	Description
req	OUT	Read or write request.
rnw	OUT	Characterizes the request as read if it is 1 or a write if it is 0.
address	OUT	Address to be read or written
data2read	IN	Data to be read from the RW bus
data2write	OUT	Data to be written to the RW bus

Table 1: RW bus signals as driven by the controller.

2.2 Interface Signals

The interface signals of the Versat controller core are described in Table 2.

Name	Direction	Description
clk	IN	Clock signal.
rst	IN	Reset signal.
Instruction Interface		
instruction[7:0]	IN	Instruction to execute.
pc[9:0]	OUT	Program Counter.
RW Bus Interface		
rw_req	OUT	Data request for read or write.
rw_rnw	OUT	Data read (1) or write (0) signal.
rw_addr[9:0]	OUT	Data address.
rw_data_to_rd[7:0]	IN	Data to read.
rw_data_to_wr[7:0]	OUT	Data to write.

Table 2: Interface signals.

2.2.1 Instruction Interface Timing Diagram

The timing diagram for pipelined instruction read using the Instruction Interface is shown in Figure 2.

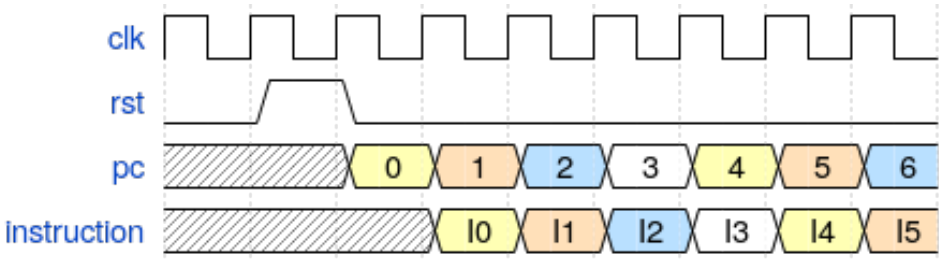

Figure 2: Instruction interface pipelined reads.

2.2.2 RW Bus Interface Timing Diagram

The timing diagrams for reads and writes using the RW Bus Interface are shown in Figure 3 and Figure 4, respectively. These operations may be consecutive or not, as illustrated.

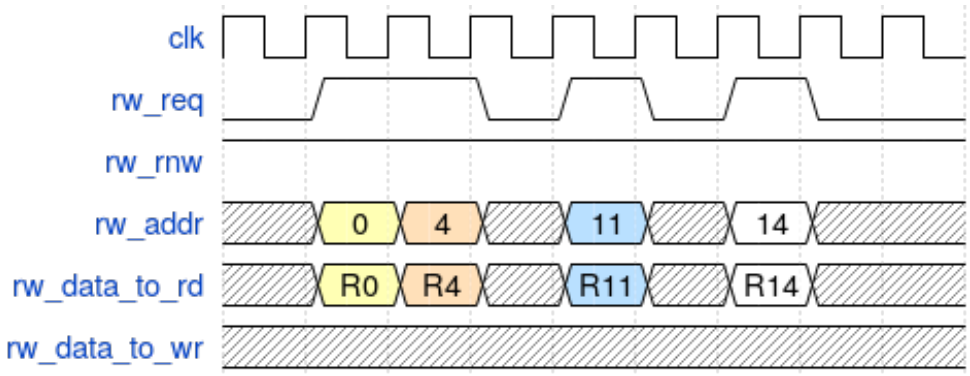

Figure 3: RW bus interface reads.

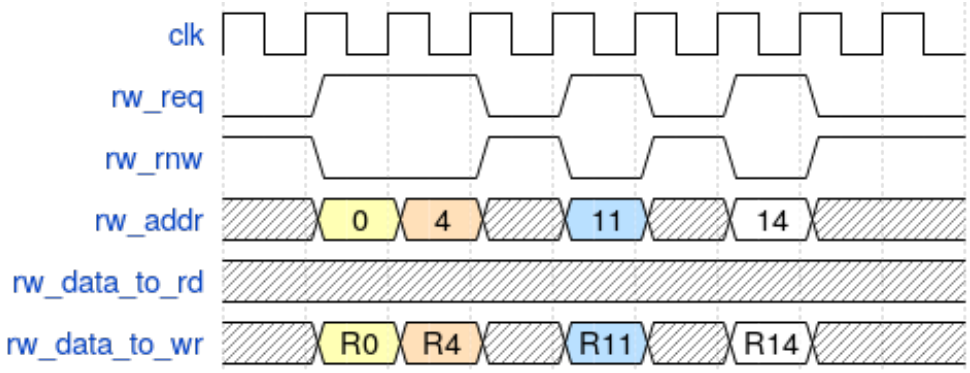

Figure 4: RW bus interface writes.

3 Memory Map

In this section the memory map of the system, where the controller is integrated, is presented as seen by its RW bus interface.

The RW bus interface, allows the programmers to access different parts of the system by using the addresses given in this section.

The base addresses given in Table 3 are reserved. Other addresses may be added to the memory map as they are needed to access new peripherals.

Mnemonic	Address	Read/Write permissions	Description
RB0	0	RW	Register B less significant bits
RB1	1	RW	Register B most significant bits
RC	2	R	Register C (carry save)
USER_MEM_BASE	3	RW	User memory base

Table 3: Memory map base addresses

4 Instruction Set

Versat controller features a very minimal set of instructions to control the execution hole system where it is integrated.

There is only one instruction type illustrated in Table 4.

Bits	Description
7-4	Operation code (opcode)
3-0	Immediate constant

Table 4: Instruction format.

The instruction set is given in Table 5. Brackets are used to represent memory pointers. For example, M[Imm] represents the contents of the memory position whose address is Imm.

Mnemonic	Opcode	Description
rdw	0x0	RA = M[Imm]; PC = PC+1
wrw	0x1	M[Imm] = RA; PC = PC+1
rdwb	0x2	RA = M[RB+Imm]; PC = PC+1
wrwb	0x3	M[RB+Imm] = RA; PC = PC+1
beqi	0x4	RA == 0? PC = Imm: PC += 1; RA = RA-1
beq	0x5	RA == 0? PC = M[Imm]: PC += 1; RA = RA-1
bneqi	0x6	RA != 0? PC = Imm: PC += 1; RA = RA-1
bneq	0x7	RA != 0? PC = M[Imm]: PC += 1; RA = RA-1
ldi	0x8	RA = Imm; PC=PC+1
ldih	0x9	RA[7:4] = Imm; PC=PC+1
shft	0xA	RA = (Imm $<$ 0)? RA $<<$ 1: RA $>>$ 1; PC=PC+1
add	0xB	RA = RA + M[Imm]; PC=PC+1
addi	0xC	RA = RA + Imm; PC=PC+1
sub	0xD	RA = RA - M[Imm]; PC=PC+1
and	0xE	RA = RA & M[Imm]; PC=PC+1
xor	0xF	RA = RA $\oplus$ M[Imm]; PC=PC+1

Table 5: Instruction Set.

4.1 Virtual Instructions

There is also a `nop` instruction, which do not have its own opcode. The assembler will translate this instruction as an `addi 0`. From the progammer point of view, it is an instruction available as any other.

4.2 Delayed Branches

The PC increments by one after non branch instructions. For branch or flow control instructions (`beqi, beq, bneqi, bneq`), the PC is assigned the branch target value, PC + Imm or RB, depending on whether it is a relative branch (`beqi, bneqi`) or an absolute branch (`beq, bneq`), respectively. Due to the controller pipeline circuits, the new value of the PC is delayed by 1 instruction. Hence, the 1 instructions immediately after the branch instruction is executed and constitute a delay slot. If one lacks a useful instruction for this slot, one should be filled with `nop` instruction.

5 Programming

The Versat controller can only be programmed in assembly language, which is not difficult due to its simple architecture. To better explain how the ecosystem works, a full example will be used in this section to show how to program and use the assembler, for an 8-bit controller.

5.1 The Assembler

The assembler is a Python script that reads the assembly code (in a <file>.va) and translates it into machine code. It is design to generate machine code for 2 different memories: the boot ROM memory and the instruction memory. (Depending on the system where it is inserted, the controller can have a program memory with a boot ROM and an instruction memories, or just the boot ROM.)

The assembler generates two files, the opcode.hex and the rom.v. The first one is just the machine code written as a text file (one instruction per line) with the program memory size, which are read with the *readmemh* (a verilog primitive). If the target memory has a 256 instruction capacity, it will generate 256 machine codes, the first instructions correspond to the ones given to the assembler and remain are filled with `nop` instructions (see section 4). The second one is a file that contains the same instructions in a verilog hard ROM format, so it is possible to use a ROM without the *readmemh*. This second file is crucial in case of ASIC implementation and is only generated for the boot ROM.

To use the assembler, there is the need to create a dictionary named xdict.txt with some parameters. This dictionary is a JSON file read by the assembler, so it can generates the machine codes correctly. The parameters needed are the following:

- All the memory addresses (memory map);
- Instruction parameters: immediate and instruction widths (IMM_W and INSTR_W, respectively). Note: the INSTR_W is equal to opcode size (4 bits) + IMM_W;
- Boot ROM address width (ROM_ADDR_W);
- Instruction memory address width (IADDR_W);
- Number of delay slots (DELAY_SLOTS).

Some of these parameters may not be necessary, it depends on the system where the controller is inserted.

For the boot ROM, the assembler should be invoked as follows:

```
./va -b < <assembly file>.va
```

For the instruction memory, the assembler should be invoked as in the boot ROM case, but without the boot ROM flag:

```
./va < <assembly file>.va
```

5.2 Assembly

5.2.1 Program

In figure 5 is an example of a full assembly routine for Versat controller.

```
# Read from SPI routine
#
# - reads the reading address from R3
# - writes the 32-bit data to R4-R7
# - returns to the address stored in R8-R9

rd_spi  ldi 31          # number of iterations (Nbits - 1)
        ldih 31
        wrw R1
        ldi R7
        ldih 0
        wrw R2
        ldi 0
        wrw R4
        wrw R5
        wrw R6
        wrw R7
        ldih 0x1        # constant for toggle the sclk signal
        wrw R0          # assert sclk signal
        rdw R3          # reads address for reading
        wrw SPI_CTRL_REG
        wrw R10
rdloop  rdw R10
        xor R0
        wrw SPI_CTRL_REG
        xor R0
        wrw SPI_CTRL_REG
        wrw R10
        rdw R2
        wrw RB
        ldi 0
        wrw RB,1
        rdwb
        shft -1
        add RTC_REG
        wrwb
        ldi 0x7
        and R1
        bneqi nincrd
        rdw R2
        addi -1
        wrw R2
nincrd  ldi rdloop
        ldih rdloop
        wrw RB
        ldi rdloop>>8
        wrw RB,1
        rdw R1
        bneq
        wrw R1
        rdw R8          # reads return address
        wrw RB
        rdw R9
        wrw RB,1
        ldi 0           # clears the SPI control register
        wrw SPI_CTRL_REG
        beq
        nop
```

Figure 5: Read from SPI routine.

Due to the given dictionary, all mnemonics are recognized by the assembler. For instance, the `SPI_CTRL_REG` is a known address from the assembler point of view.

This routine executes the SPI protocol for reading from external modules, by reading and writing to a peripheral present on the memory map. The comments are written after a "#" character is inserted, they can be used in the beginning of a line, or at the end of it.

The blue words are labels and the assembler use them to compute the jump addresses.

The immediate value has half the width of the data. To load a value bigger than that, the programmer should do as follows:

```
ldi 0x67
ldih 0x67
```

The assembler will first read the less significant bits from value, for the first instruction, and then will read the most significant bits, for the second one. The prefix `0x` is used to indicate to the assembler that it is a hexadecimal value.

When an immediate value is bigger than the IMM_W parameter, the programmer must use a shift as follows:

```
ldi 0x367>>8
```

This shift will be executed at compile time by the assembler. It performs the shift and then replaces the immediate in the instruction by the result (in this example, 3).

The same happens with labels, and because the programmer never knows what value they have, it is imperative to use this scheme.

```
ldi rdloop
ldih rdloop
wrw RB
ldi rdloop>>8
wrw RB,1
```

This last instruction is translated by the assembler as a write instruction to the sum of `RB` address plus one. To read or write any value to a memory mapped address bigger than immediate width (in this example, 4 bits), the instructions `rdwb` and `wrwb`, respectively, must be used.

For optimization, in a case of a `for loop`, the delay slot is filled with the write instruction to the control loop register (see 4). For example:

```
rdw R1
bneq
wrw R1
```

where the `R1` is the control register. Do not forget to load the `RB` with the label used in the `for loop` before the branch instruction.

5.2.2 Data

In figure 6 is an example of a data section in an assembly program for Versat controller.

```
# digit table
dtable .memset 0x3F # 0
       .memset 0x06 # 1
       .memset 0x5B # 2
       .memset 0x4F # 3
       .memset 0x66 # 4
       .memset 0x6D # 5
       .memset 0x7D # 6
       .memset 0x07 # 7
       .memset 0x7F # 8
       .memset 0x6F # 9
```

Figure 6: Data in the program memory.

Here, the data must be preceded from the directive `.memset`, in the instruction place. The use of a label is required to access the data.

References