Design of a minimal processor on an FPGA

Andrey Akhmetov

October 1, 2015

Abstract

This document describes the development of a small processor using the Verilog programming language, a software-based simulator, and FPGA hardware. This processor was made as a learning exercise and is not necessarily efficient, reliable, or compliant with best-practices.

I would like to additionally thank the many helpful members of the ##fpga channel on the Freenode IRC network for their helpful tips and expertise.

1 Disclaimer

The goal of this project was to act as a learning exercise. The decisions here should *not* be considered best-practice. The implementation made here is not necessarily coherent, efficient, or useful for anything other than a small introductory learning exercise.

2 Introduction

Much of the development of this device is based around a rather vague goal of being able to execute machine code, and more specific constraints of the available hardware and its properties.

2.1 Available hardware

The project is targeted to run on a Xilinx Spartan-3E FPGA, namely the XC3S500E. The device contains 4656 logic slices, which are equivalent to 10476 logic cells, as well as roughly 360Kbit of block RAM spread over 20 RAM modules, each of which has 16384 bits of memory, with 2048 bits parity data.[2]

The development board being used contains 8 LEDs, 8 slide switches, four pushbuttons, a two-digit seven segment display, and 32 GPIO ports. There is an onboard clock crystal running at 50MHz, as well as a clock controlled by a physical dial for lower frequencies, from hertz to tens-of-kilohertz range.

3 Overall architecture

The overall architecture of the processor is an eight-bit, non-pipelined processor, with only in-order execution. While features such as out-of-order execution,

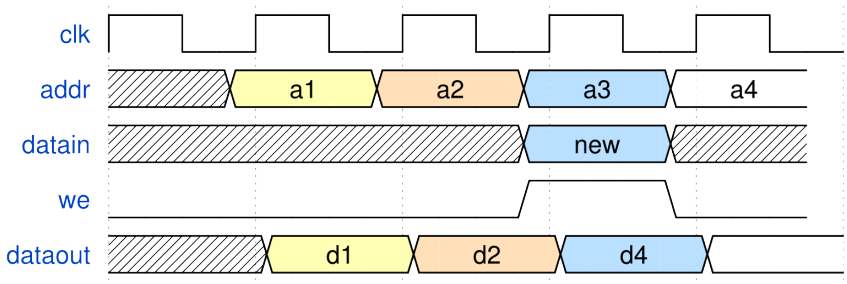

Figure 1: A display of memory accesses on one of the two ports of any given block RAM.

superscalar pipelining, and register renaming are useful for performance, they are considerably more difficult to understand and implement than the more basic design here, and consume more FPGA resources, such as circuitry to detect register access conflicts.

Due to instruction and addressing limitations (such as a bigger code memory than addressable data memory space) a Harvard architecture is used, meaning that code is stored separately from data. Instructions to write to program memory, that break this abstraction, are under consideration. However, it is more likely that there will be a process to upload new code using an external interface such as GPIOs.

4 Memory accesses

An eight-bit architecture was chosen due to the estimated number of implemented opcodes, and the available memory access patterns. While the actual memory width is nine bits, the fashion in which memory is pre-assigned in Verilog separates one of the bits from the rest and packs the ninth bits of different words into a single source code character, making programming by hand significantly more difficult. The memory access pattern of 8-bit data by 2048 different possible addresses allows for a reasonable 16K of program memory, more than sufficient for the small programs necessary for the learning purposes of this project.

The block RAMs on the FPGA are dual-ported, synchronous SRAMs, where the width of the two ports need not be the same. A write-enable signal can be used to write. Either a read or a write operation (selected by an input signal, write-enable) can occur on each clock cycle, as shown in 1. When the RAM is being used in dual-ported mode, two memory accesses (reads, writes, or one of each) can happen on each clock cycle, although write conflicts to the same address the two ports may present a danger of corruption, and thus avoided in the design.

The behavior of the read output when writing can be configured in software, to either pass through the new value to the output, return the stored value (get and set), or continue returning the result of the previous read operation. This is unimportant since there are no get-and-set scenarios in the current design.

4.1 Program memory

Due to the dual-ported nature of the program memory, it is possible to perform two 8-bit reads at the same time. This lends itself especially well to fetching an 8-bit instruction, and an 8-bit parameter (represented in Assembly languages similarly to INSTR *imm8*. This would not have been a guarantee for a single-ported 16-bit memory since instructions are aligned on 8 bits, so an instruction and its parameter may cross a 16-bit boundary.

The availability of a very efficient hardware adder that can be used on the address lines of the second port allows for one register, *PC* (short for Program Counter), to be used to determine the address of the next instruction to fetch, as well as the address of its parameter. The first port reads the opcode into a set of 8 FPGA registers, and the second port simultaneously reads the parameter into a second set of 8 FPGA registers, on the clock rising edge. If the parameter is not needed, later logic will ignore it.

4.2 Data memory

Due to the goal of fitting command parameters within 8 bits, only 256 bytes of memory can be directly addressed by a constant in an opcode. However, the full 2048 bytes (16384 bits) of a single block RAM element can be accessed by a paging mechanism with a dedicated opcode that sets the page to either an immediate parameter or a register value.

The memory is configured as an eight-bit width, with only a single port. Since the addresses on any given page are 8 bits, that means that only a single address will fit as a parameter. For this reason, a second port is not needed, as any operation will only access the memory once, as either a read or a write.

4.3 Registers

The processor contains 16 registers (the reason being discussed in the Opcodes section), named by hexadecimal 0 through F. Some of these registers are paired together for a specific feature, but may be used as general purpose registers. Registers 0 and 1 are paired to be the accumulator (0 being the low byte), 2 and 3 are the comparison operand, and 4 and 5 are an address (used for jumps and memory addressing). Due to addresses being 11 bits, only the 11 lowest bits of register 4-5 are considered when jumping or addressing. See Table 1.

The hardware implementation is currently a register file in a block RAM that is dual-ported, and 16 bits wide (18 bits including two unused parity bits). Up to two register fetches can occur on a single clock cycle, and a single register write can occur on the next cycle (the second bank's write enable is always off since there are no opcodes requiring more than one write, but this is flexible). The data width is 16 bits so that operations that need to fetch wide registers such as A, X, or AD, can do so using one of the ports, and fetch a parameter using the other. If an 8-bit register needs to be fetched, the 16-bit block containing it is fetched and connected to a multiplexer switching on the LSB of the register number. The half being used is connected combinatorially to the ALU's logic to calculate the result, and the other half is written back "as is" (if it is not, then the neighboring register will be zeroed on any write).

Table 1: Registers and their features

ID	Feature
0	Accumulator
1	
2	Product/comparison
3	
4	Address
5	
6	General-purpose only
7	General-purpose only
8	General-purpose only
9	General-purpose only
A	General-purpose only
B	General-purpose only
C	General-purpose only
D	General-purpose only
E	General-purpose only
F	Hardware write

To increase performance (at the expense of FPGA resources) the module can be easily replaced with one that stores values in distributed RAM (presumably within SLICEM resources on the FPGA), uses multiplexers for fetching and a 4-to-16 decoder connected to the write-enable of one or more of the registers. This will eliminate the need to wait for a clock edge to fetch inputs (as is needed with SRAM) but eliminates the possibility of writing with a second port without a set of multiplexers at the write circuitry, further increasing utilization of the FPGA resources.

4.4 PTR

Due to memory timing constraints and the limited number of read/write ports on the block RAMs, a separate 8-bit-long pointer (denoted PTR) can be used for addressing memory dynamically.

5 Opcodes

The choice of available operations and how they map to opcodes was based largely around the features that needed to be made available, and the widths of the opcodes and parameters. One clock cycle is required to fetch an opcode, which can be decoded during the clock period (as it is available at the start of the clock period). Illegal instructions cause the processor to halt, at which point the debug switch can be used to inspect the current opcode and program counter. This decision was made as it is not known whether a currently-unused opcode might take a parameter in a future version, and simply incrementing the program counter and continuing would cause drastic issues even if the corrupt or skipped instruction itself was not at fault.

Many of the choices surrounding assigning operations to opcodes were made

Table 2: LDC opcode

Instruction				**Destination**			
0	0	0	1	A	B	C	D

Table 3: Binary operation opcode

Instruction								**LHS**				**RHS**			
X	X	X	X	X	X	X	X	Y	Y	Y	Y	Z	Z	Z	Z

for efficiency. Where possible, single bits of the opcode are used to multiplex between various possible outputs or results, rather than having to use more expensive lookup table or matching logic.

5.1 Processor control

Opcodes from 0x00 to 0x0F are used for processor state control, such as clearing the carry flag, halting the processor, pausing for an adjustable number of clock cycles, and NOP (no-operation). NOP is mapped to 0x00, which is the default for memory contents (by virtue of the memory init parameters defaulting to 00).

5.2 Register load and store

Since registers are 8 bits, the load constant opcode (LDC)'s parameter is completely occupied by the value to load into a register, but which register to load to is not specified. For this reason, the actual register to be set to that value is specified in the low nibble of the opcode. The instruction decoder checks for LDC using a comparison of only the four high bits of the opcode. This means that the opcode for LDC is 0x1R where R is the hexadecimal ID for the register whose value is to be set. This is one of the primary reasons why 16 was chosen as the number of registers. Programming is easier since one of the hex digits of the opcode denotes the instruction LDC and the other denotes its destination, and only 16 opcodes are occupied by LDC (48 in total over all similar functions).

A similar situation occurs with LDM (load from memory), where the opcode is in the form 0x2R (R being the register to load to). The 8-bit parameter here specifies the memory location to read the value from (on the current page).

The same is the case for 0x3R (STR, store to memory), with the register and memory location in the same parts of the instruction as LDM, but the data transfer going the other way (register to memory).

5.3 Binary operations

Binary operations with registers are more economical in terms of the scheme used to encode opcodes and parameters. It is possible to encode two registers in the 8-bit parameter, since register IDs are 4 bits. However, this means that the destination register must be one of the two source registers.

For example, the instruction 49 A3 corresponds to XOR, such that Register A = Register A $\oplus$ Register 3. The full list of instructions is included in Table .

Some instructions that perform such operations reference the accumulator, a 16-bit representation of registers 0 and 1 (with register 0 being the low byte). For example, instruction ADDA (0x41) adds the value of an 8-bit register to the 16-bit accumulator. The register file memory is configured so that the old value is still maintained on the output during a write, so either of the accumulator's component registers can be referenced without risk of data corruption due to the value of the adder's combinatorial output changing on the clock edge.

5.4 Opcode list

The following table is a listing of all of the implemented opcodes.

Table 4: Opcode list

Opcode (hex)	**Parameter (hex)**	**Mnemonic**	**Action**
00	none	NOP	No operation.
01	none	CCF	Clear carry flag.
02	none	HLT	Halt until button BTN1 pressed. [1]
03	none	SLP	Sleep (no operation) for number of cycles equal to value in register AD.[2]
04-0F	none		Reserved.
10-1F	8-bit parameter	LDC	Load an 8-bit constant into register denoted by low nibble of opcode.
20-2F	8-bit parameter	LDM	Load an 8-bit memory value into register denoted by low nibble of opcode, from memory address in the current page given by constant.
30-3F	8-bit parameter	STR	Store value of register denoted by low nibble of opcode into current memory page at address given by constant.
40	flags,reg	ADDA	Adds value of register reg to accumulator. flags reserved for future expansion. Sets carry flag on overflow.
41	8-bit parameter	ADDCA	Adds value of param to accumulator. Sets carry flag on overflow.
42	flags,reg	SUBA	Subtracts value of register reg from accumulator. No effect on carry flag.
43	8-bit parameter	SUBCA	Subtracts value of param from accumulator. No effect on carry flag.
44-47	none		Reserved
48	lreg,rreg	INV	Sets lreg to the bitwise complement of rreg.
49	lreg,rreg	XOR	Performs lreg XOR rreg, writes result to lreg.
4A	lreg,rreg	AND	Performs lreg AND rreg, writes result to lreg.

Continued on next page

Table 4 – continued from previous page

Opcode (hex)	Parameter (hex)	Mnemonic	Action
4B	lreg,rreg	OR	Performs lreg OR rreg, writes result to lreg.
4C	lreg,rreg	MOV	Writes value of rreg to lreg.
4C-4F	none		Reserved
50	lreg,rreg	MUL	Sets register X to register lreg × register rreg.
51-5F	none		Reserved
60	none	JP	Jumps unconditionally.
61	none	JPN	Reserved, with no effect.
62	none	JPC	Jump if carry flag set.
63	none	JPNC	Jump if carry flag not set.
64	none	JPEQ	Jump if $A = X$.
65	none	JPNEQ	Jump if $A \neq X$.
66	none	JPZ	Jump if $A = 0$.
67	none	JPNZ	Jump if $A \neq 0$.
68-6F	none		Reserved
70	ignore,reg	PSR	Switch memory page to 3 lowest bits of register reg
71	8-bit parameter	PSC	Switch memory page to 3 lowest bits of param
72	8-bit parameter	PTRC	Set PTR to param
73	ignore,reg	PTRR	Set PTR to value of reg
74	ignore,reg	WRPTR	Write value of reg to RAM address at PTR
75	ignore,reg	RDPTR	Read from RAM address at PTR into reg
80	ignore,reg	INCR	Increment reg
81	ignore,reg	DECR	Decrement reg
82	none	INCP	Increment PTR
83	none	DECR	Decrement PTR
90	none	CALL	Call unconditionally.
91	none	CALLN	Reserved, with no effect.
92	none	CALLC	Call if carry flag set.
93	none	CALLNC	Call if carry flag not set.
94-95	none		Reserved[3]
96	none	CALLZ	Call if $A = 0$.
97	none	CALLNZ	Call if $A \neq 0$.
98-9F	none		Reserved
A0	none	RET	Return (call stack)
A1-FD	none		Reserved
FE	none	WRLED	No effect, illegal instruction
FF	none	WRSEG	Write register F to 7-segment display

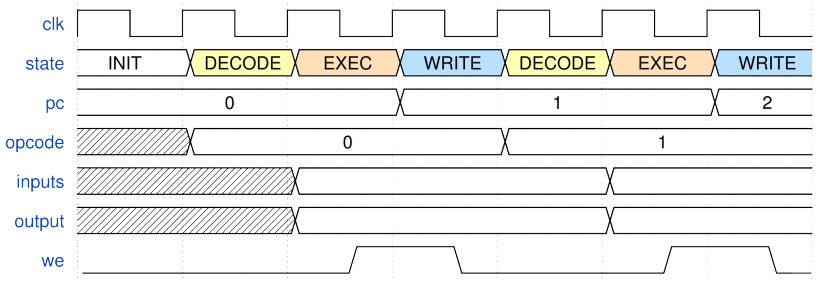

Figure 2: The timing diagram describing the operation of the FSM in the ALU.

6 ALU/CU

The key component of this project had been the ALU (arithmetic and logic unit), implemented as a Verilog module. Currently the program, data, and register memory modules are included inside the ALU at the Verilog source level, but they can be easily moved outside if I decide to somehow interface with external memory using a simple synchronous address/data bus. The ALU contains its own control circuitry, as a result of my misunderstanding the organization and naming of parts of modern processors when initially implementing it.

The timings of the memory accesses and computation were key in planning out the ALU, which is implemented as a state machine. Each operation is executed in three states (see 3). Each state corresponds to one clock cycle. The ALU begins with the program counter set to zero, and in an initialization state. On a rising clock edge, as the opcode is made available on the RAM read port, the ALU transitions into state DECODE, where the opcode is examined in a series of if-elseif statements. A switch is not used, since large block of opcodes (for example, LDC) are treated the same way and comparison of the upper 4 bits suffices for that check. The register IDs and RAM addresses that need to be fetched are written to the appropriate address lines on the block RAMs.

At the following transition, state EXEC begins. The program counter is updated and the register values are read and processed by a combinatorial circuit based on the opcode, and the output is connected to the write data lines on the block RAM. Write-enable is activated as appropriate, and on the clock cycle, the data is written and the WRITE state is entered. At this point, write-enable is deactivated, and the write is finished. For an unknown reason the synthesis tool does not respect setup times on the PC and PC+1 values set on the address lines of the program memory, so this clock cycle acts to respect this setup time as well. At the next clock edge, the ALU transitions back into the DECODE stage for the next instruction.

[1]This is implemented by not incrementing the PC value. A button used to resume execution acts as a stand-in for non-existent interrupt functionality.

[2]This register's name isn't completely correct. However, it's mostly used to store an address.

[3]For an optimization to better utilize both block RAM ports. Unexpected results (compares AD to A instead)

7 Call stack

The opcodes within the 9Xh block are used to call a subroutine, using a hardware stack. The primary deciding factor for its size was the register file and the block RAM that it needed to fit in. Because the registers take up 128 bits of space, and 16384 bits are available in a single block RAM without additional write-enable and multiplex logic, the call stack is 128 levels deep. In the case of a stack overflow, the processor will stop for debugging. In order to allow calls with only a few clock cycles, the old register values remain when calling to a specific stack depth. This is most evident in calling a routine that writes to a register, returning from it, and calling a routine that reads that same register without first writing anything to it.

This is not an issue as the RAM can be used to pass parameters and return values. Additionally, existing programming languages already assume that certain memory locations can have garbage if uninitialized, including high-level languages such as Java (in regard to the stack). Remarkably, CPU architectures such as Itanium have special values describing uninitialized or garbage values.[4]

8 Code upload

Code upload was done from a Raspberry Pi due to limited available hardware, and was implemented as a serial protocol. Currently, the protocol is a very simple bit-banged protocol with minimal error correction:

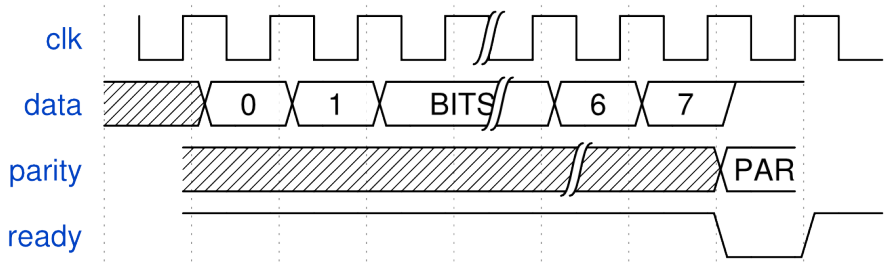

Figure 3: The timing diagram describing the upload protocol.

An acknowledgement is sent over the data line after each byte, and acts as a response to the parity sent by the FPGA. If the parity is incorrect or a framing issue occurred, a low value (NACK) is sent. If the parity and framing are correct, a high value (ACK) is sent instead.

The READY signal acts as a minimal form of framing, to allow a misalignment error to be handled by sending low signals until the READY signal goes low itself, followed by another low signal (NACK) to prevent the byte from being written.

The parity allows for basic checking of single-bit errors (for which case a negative acknowledgement and a retransmit are sent).

9 Practical implementation result

9.1 Resource utilization and statistics

The processor, when implemented, has requirements that are easily satisfied by the Spartan-3E FPGA, which is a fairly small and low-cost device. These figures do not include tangential functionality such as the code upload circuitry.

Table 5: Utilization of XC3S500E FPGA resources

Resource type	**Utilization**	**Percent utilization**
Flip-flops in slices	348	3%
4-input lookup tables	847	9%
Occupied slices	501	10%
Block RAM elements	3	15%
Clock multiplexers	1	4%

The Picoblaze microprocessor (available as a soft IP core from Xilinx) can be used as a comparison, as shown in Table 6 [1]

Table 6: Comparison to the Picoblaze microcontroller

	My implementation	**Picoblaze**
Architecture	8-bit	8-bit
Instruction store size	2048	1024
ALU width	Mixed (8-bit with 16-bit accumulator and comparison)	8-bit
Registers	16*8-bit	16*8-bit
IOs	8 status LEDs, byte display[4]	256 in, 256 out
Call stack	None[5]	31-position
Cycles per instruction	2 theoretical[6]	2
FPGA slices	501	96

9.2 Known issues

While the processor implementation was able to execute all implemented opcodes, there are currently some outstanding issues, each of which is discussed in greater detail later.

- **Timing:** It had been found that reads from the block RAM were failing under some circumstances.
- **Debuggability:** No provision so far had been implemented for a debugger interface other than LEDs for internal flags and 7-segment through opcode FF.
- **Missing opcodes and features:** Certain opcodes and features were missing in the first version of this learning exercise.

9.3 Timing

In implementing opcodes that read from both register file ports on the read stage, (specifically MOV) it was found that the FPGA implementation was failing to read the register file using port B, while port A would succeed. Furthermore, this issue could not be reproduced in behavioral and post-synthesis simulation. It appears from discussions with other FPGA programmers that this may be a bug in the ISE software used. For this reason, an extra clock cycle was added as a workaround to a few different operations.

The current clock frequency is 50MHz for testing, but timing constraints with frequencies as high as 100MHz were satisfied, although specialized profiles such as physical synthesis and lengthy place-and-route runs had to be used to reach this goal. The highest clock frequency reached was 101.033MHz.

9.4 Debuggability

Due to limited I/O resources, debugging is done by using 8 LEDs and a pair of 7-segment displays that are wired to display one byte. By using multiplexers and a few switches, the seven-segment display can be toggled between displaying program output (using opcode FF), the 8 lowest bits of the program counter, or the current opcode executing.

Additionally, a separate switch was used as a clock-enable to pause and resume execution. The LEDs display state such as operation, paused operation, jump flag, and carry flag.

In case of an invalid opcode, or stack overflow/underflow during call/return, respectively, the processor "locks up" by remaining in the same state with an unchanging program counter. This allows for easy debugging by using some of these features to probe the processor's state. Additionally, with a VGA controller implemented as a separate project, it is possible for an error to cause a jump into debugging code in a separate block RAM (using a multiplexer) or a separate location in the current block RAM, in which case the debugger can print the state of the processor including registers, type of error, and other information. However, extra instructions will need to be added to fully support debugging, including ones to read the program counter and possibly format a byte to ASCII hex in hardware.

9.5 Missing features

The original implementation was a learning exercise for a pure ALU. However, features such as interrupts are worth implementing as a further learning exercise.

Additionally, there were some opcodes that are not strictly required, but are more efficient than if they were emulated using the current instruction set, for example ADDC (add-with-carry-in) that would otherwise require a jump using JPC.

References

[1] *PicoBlaze 8-bit Embedded Microcontroller User Guide.* UG129. Rev 2.0. Xilinx Inc. June 2011.

[2] *Spartan-3E FPGA Family.* DS312. Rev. 4.1. Xilinx Inc. July 2013.

[3] *Spartan-3 Generation FPGA User Guide.* UG331. Rev. 1.8. Xilinx Inc. June 2011.

[4] *Uninitialized garbage on IA-64 can be deadly.* Microsoft Developer TechNet blog. Jan 2004.