THE OPTIUM MICROPROCESSOR – AN FPGA-BASED IMPLEMENTATION

Radu Balaban
Computer Science student, Technical University of Cluj Napoca, Romania
horizon3d@yahoo.com
Horea Hopârtean
Computer Science student, Technical University of Cluj Napoca, Romania
just_a_passenger@yahoo.com
Octavian Creț
Lecturer, Technical University of Cluj Napoca, Romania
Octavian.Cret@cs.utcluj.ro
Zoltan Baruch
Ph.D., Technical University of Cluj Napoca, Romania
Zoltan.Baruch@cs.utcluj.ro

Abstract

We designed a microprocessor that features today's modern computing concepts while still using reasonably low resources. We decided to implement a simple eight-bit architecture and a small instruction set, on top of which to employ features such as burst memory access, branch prediction and pipelined execution. The microprocessor was simulated and implemented in a Xilinx XC4005E FPGA device, using the Foundation Series software and the "FPGA demoboard" manufactured by Xilinx Corporation.

1. Instruction Set Summary

The instruction set of the Optium processor is not very large, but it features all the basic operations found in a general microprocessor. The 8-bit word size of the processor is somewhat a limitation with respect to the extents of the instruction set, however there are several double byte instructions that are meant to overcome this limitation. In this section we shall describe the general instruction format, the main instruction types, and a list of *opcode*s for the supported instructions.

The general instruction format is built upon the two types of instructions that make up the Optium instruction set - the single-byte simple instructions and the double-byte instructions dealing with binary operations or addressing operands which are therefore more complex.

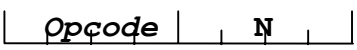

Figure 1: *Single byte instructions*

The basic format of a single-byte instruction is described in **Figure 1**, where *opcode* represents the operation code of the instruction (see **Table 1**), while *N* represents the number of the I/O port. The double-byte instruction format is presented in **Figure 2** and in this case *AM* and *JC*

stand for the addressing mode and for the jump condition, respectively. **Tables 2** and **3** give the possible values and their meaning for these symbols. Finally *m* represents either the immediate operand, or the address of the operand.

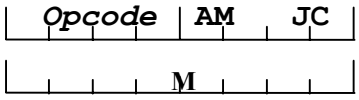

Figure 2: *Double byte instructions*

The instruction set is composed of two pairs of LOAD/STORE transfer instructions with the internal memory and an input or output port, respectively, one arithmetic instruction (ADD), three logic instructions (AND, CPL, RRC), and two jump instructions.

Instruction	Description	Flags	*Opcode*
LOAD A,m	A←m	Z	0000
STORE m,A	m←A		0001
ADD A,m	A←A+m	C,Z	0010
AND A,m	A←A∧s	Z	0011
JUMP m	PC←m		0100
JUMP t,m	PC←$\overline{t}$*(PC+1)∨t*m		0101
CPL A	A←$\overline{A}$	Z	1000
RRC A	A←A/2,C←A_0	C,Z	1001
LOAD A, $N	A←IN(N)	Z	1010
STORE $N,A	OUT(N)←A		1011

Table 1*: Operation codes and affected flags*

AM	Description
00	Immediate = m
01	Absolute = addr(m)
11	Indirect = addr(addr(m))

Table 2*: Addressing modes*

JC	Description
00	Carry Set
01	Carry Clear
10	Zero Set
11	Zero Clear

Table 3*: Jump conditions*

In **Table 1**, *A* is the only general-purpose 8-bit register of the processor, while *PC* is the program counter register. An operation takes one or two clock cycles to execute. The *opcode* of the double-byte instructions begins with the 0 binary digit, opposed to the *opcode* of the single-byte instructions, which begins with the 1 binary digit. Any other *opcode* or incorrect addressing mode is treated as a NOP (No Operation) instruction, which could be used for delaying or synchronizing the processor with an I/O operation. The NOP instruction does not alter the flags and it executes in one clock cycle.

2. The Optium Processor Architecture

Since the memory is embedded inside the chip, the processor interacts with the environment mainly by using one input and one output 8-bit port. The terminal of the bus could have various registers or I/O devices connected. The implementation has revealed that the Optium microprocessor together with its internal synchronized memory can work at a frequency of up to 20 MHz.

The block diagram of the processor is presented in **Figure 3**. The Optium microprocessor has three main components: the memory unit, the fetch/decode unit, and the execution unit. They are presented next.

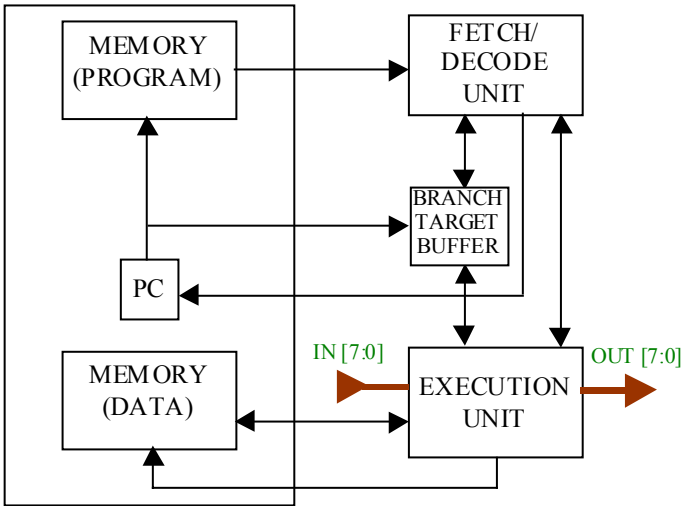

Figure 3: *The internal structure of the OPTIUM processor*

2.1 The Memory Unit

The Memory unit is a block of 256 bytes, equally divided between the lower (0-127 bytes) Program Memory which contains the code of the program to be executed, and the upper (128-255 bytes) Data Memory, containing the variables used by the program. The fetch/decode unit reads the lower area, while the upper area can be read or written by the Execution Unit. This ensures the independent access necessary for the two different stages of the pipelined execution. Locations in the memory are referred by 8-bit addresses. The addresses from the lower memory start with a 0 bit, while the addresses from the upper memory start with a 1 bit. In all cases, the first bit is disregarded since we separately refer to 128-bytes memories instead of referring to the entire 256-bytes memory. This prevents the user from mistaking data for program instructions due to accidentally jump addresses, or modifying the code during the execution, which both might result in errors that are hard to detect.

2.2 The Fetch/Decode Unit

The *Fetch/Decode Unit* has the task of taking the instructions from the lower memory. The control transfer instructions are dealt with right here, while the rest of the instructions are sent in a continuous flow towards the execution unit. Once the instructions are ready, this is signaled to the Execution Unit. Two important features accomplish this stage of the pipeline: burst memory access and branch prediction.

Burst memory access is achieved by buffering the bytes from the memory and sending them to the execution unit in “bursts”, such that both bytes of a double byte instruction are received simultaneously. In the worst case this buffering involves the same latency as a similar sequential process, but by placing a slow (2-clock interval) instruction before a double byte instruction, an important overall gain is achieved by the pipelined execution.

The Optium processor uses both static and dynamic *branch predictions*. The static branch prediction is used when encountering a conditional jump that has not been executed before, and the dynamic prediction is employed for an already “guessed” instruction. The purpose of the prediction is to avoid pipeline stalls, that is when a conditional jump is encoun-

tered, the Fetch/Decode Unit does not know whether it will be taken or not, since the instruction currently executed by the Execution Unit might change the state of the flags. Without the prediction, the Fetch/Decode Unit would have to wait for the Execution Unit to finish the job, then check the condition, and finally update the Program Counter to fetch the next instruction.

The static prediction uses an algorithm based upon the most probable behaviour: if the jump address is less than the address of the current instruction - such as in *loops* - the jump will be predicted to be taken, while in the other case it will be predicted not to be taken. Once a jump is encountered, it will be statically predicted and stored in the branch target buffer (BTB). When that jump is encountered again, it will be dynamically predicted using the prediction already stored in the BTB.

In the event of a misprediction signaled by the Execution Unit, the Fetch/Decode Unit has the task of recovering from the mistaken state, and switching to the correct context. The prediction is updated directly by the Execution Unit into the BTB, and will have an increased chance of being predicted accurately in a future occurrence.

The Program Counter is an 8-bit register controlled by the Fetch/Decode Unit. It is needed by the BTB for the comparison that decides the prediction. Its contents may not be read, but may be altered by means of the jump instructions.

2.3 The Execution Unit

Once the Fetch/Decode Unit has prepared the instructions, which is signaled by a validation bit, the Execution Unit has the task of actually carrying out the operations. This stage consists of a Control Unit, two 8-bit instruction registers, a general-purpose register, two flag registers, an arithmetic and logical unit - ALU, a unit that resolves the addressing mode - AMU, and is connected to the upper Data Memory and to the I/O ports of the Optium microprocessor. The execution of an instruction takes one, respectively two clock cycles for the instructions with an indirect addressing mode.

The *Control Unit* deals with the command variables of the entire Execution Unit. This unit consists mainly of two ROM chips: one is the Instruction Decoder which, based upon the *opcode* from the first instruction registers decides the output of the second, the State Decoder. This decoder stores the state of the command variables for the entire unit. For the second clock cycle of the instruction, the output of the State Decoder is commanded by its previous output (delayed with one clock cycle).

The *instruction registers* receive a complete (single or double byte) instruction from the Fetch/Decode Unit. The first register (containing the *opcode*) commands the Instruction Decoder of the Control Unit and is also used by the AMU. The contents of the second register (in the case of a double-byte instruction) are passed on to the ALU and to the AMU.

The *general-purpose register* works together with the ALU, as an 8-bit register that acts the typical role of an accumulator. It is involved in all arithmetic/logical operations as an operand and as the destination of the operation. The register is connected directly to the output port. The ALU receives data from the AMU, from the input port and from the general-purpose register itself.

The *flag registers* are two special 1-bit registers that reflect the state following the execution of an arithmetic/logical operation. The zero flag register will be set upon the execution of instructions that yield a zero result, and the carry flag register will usually signal arithmetic and shift overflows. These registers cannot be read; however, their status may be determined and reflected by the execution of the conditional jump instructions.

The *AMU* – addressing mode unit, interprets the two *AM* bits from the first instruction register and the operand *m* from the second instruction register and provides the proper value for the ALU by accessing the upper Data Memory. It can also write into this memory. The actions of this unit make the difference between the execution time of various operations. This is where one extra clock cycle is spent to retrieve a value or memory address from an indirect operand.

The Execution Unit also communicates with the BTB. It does that whenever it receives a conditional jump instruction in order to validate the prediction made by the Fetch/Decode Unit. If the contents of the flag registers invalidate the prediction, then the branch target buffer is updated and the Fetch/Decode Unit is notified of this event.

3. Software Tools

3.1 OptAsm

OptAsm is an assembler that generates specific code for the Optium processor. It has a very flexible syntax, which allows getting the most out of the instruction set available. It accepts text files which contain assembly language and outputs either memory configuration files (*.mem* files) to be loaded onto the processor, or binary memory files for emulation and testing purposes.

3.2 Syntax description

The basic instruction that OptAsm interprets has the following general format:

[*Label*:] [*Instruction*] [[*Operand1*] [,*Operand2*]]

where**:**

Label represents an identifier that receives the value of the address of the instruction; it is very useful for jump instructions and can also be used when defining variables; it must have at least three characters. *Instruction* may be any of the instruction presented in Section 1, plus an additional instruction - DB, which inserts the value specified in *Operand1* at the current address. *Operand1***,** *Operand2* are the operands that the instruction may receive; some instructions can require different number of operands. The syntax also permits the use of constants (hexadecimal two-digit values).

3.3 Addressing modes

The addressing modes, as they have been described in **Table 2**, are immediate, absolute and indirect. Specifying the addressing mode is done in OptAsm with the use of the parenthesis. In immediate addressing, the value is enclosed within normal parenthesis - (00); in absolute addressing, the value is enclosed within brackets - [00], and in indirect addressing the value is enclosed within braces - {00}.

Some instructions accept only some addressing modes. For example, only absolute addressing may be used with the jumps, and immediate addressing cannot be used when writing to memory. The addressing modes must be employed when using labels, simply enclosing the label within the right symbols. However, the results are somewhat different when using labels in immediate mode since the address of the label is obtained as an immediate value.

3.4 Jump conditions

The conditional jump instructions require jump conditions as their second operand. The syntax for describing jump conditions is the following: |*condition_code*|, where condition code may be one of the following: **CC** - Carry Clear, **CS** - Carry Set, **ZC** - Zero Clear, **ZS** - Zero Set.

3.6 OptEmu

OptEmu is an emulator of the Optium processor for the PC. It emulates the entire instruction set, the processor's behaviour, detects jump prediction and gives hints related to program optimizations. Besides all this, it has debugger functions, which allows to run the program step by step, load/save/edit/view memory contents, and unassemble portions of memory.

It also supports the altering of the processor internals such as the accumulator, the program counter and the flags, or the complete reset of the processor, which brings it into the fundamental status. The user may observe the internal status of the processor, so debugging programs will be very easy. In order to learn more about the commands that OptEmu supports, run the '?' command at the program prompt.

4. Conclusions

The Optium microprocessor project proves that modern computing techniques can be implemented even with the low resources (196 CLB) of a Xilinx XC4000 family FPGA board. The final design was specified using only schematic entry methods, as the available HDL compilers (ABEL) did not yield satisfactory results in terms of speed and number of gates used. The tests were successful, as the processor ran at a frequency of 20 MHz.

Several programs have been successfully implemented on the Optium despite its restrained memory size. The processor can be used as a data decoder/encoder with the possibility of modifying the encryption algorithm.

The instruction set is also subject to be extended or even changed to provide more speed for specific applications or algorithms. We are currently working on orienting the Optium towards a RISC architecture style, together with the use of reconfigurable computing techniques. Finding the balance between hard and soft is the keyword for the future of this project.

5. References

[1] S. Nedevschi, O. Creţ, Z. Baruch, "*Proiectarea sistemelor numerice folosind tehnologia FPGA*", Ed. Mediamira, Cluj-Napoca, 1999.

[2] Xilinx, Inc., "*The Programmable Logic Data Book*", 1999.