

Microprocessor Course

Homework#3 – High Performance and Power-Efficient Application-Specific Microprocessor Design for On-Chip Interfaces

In several applications of digital system designs, an efficient hardware-software co-design is desirable. The hardware blocks must be improved in terms of area, performance and power consumption. On the other hand the skilled software engineers are also needed to provide suitable codes in terms of memory usage, code size and CPU run-time. It is also needed to reduce the number of arithmetic instructions in the software codes to improve power consumption.

There is also another major challenge in designing digital systems. General-purpose microprocessors like 8086 are widely used in several applications, but we may not need all the available instructions in a general purpose microprocessor. As software instructions mostly become the bottlenecks of digital system designs, it would be a great approach to optimize the microprocessor architectures for specific applications. As an example in designing a JPEG digital camera, which is a custom single-purpose processor, we have investigated that implementing the simplest microcontroller architecture, 8051, becomes two-times slower than the hardware blocks' critical path.

In this homework we are going to design a very simple microprocessor architecture. The power and performance issues of this microprocessor are really critical and we have to optimize them. Our desirable microprocessor has to cover the instructions in TABLE I.

TABLE I: The assembly instructions in our application-specific microprocessor

Assembly Command	Encoded Value (Machine Language)
MOV	000
ADD	001
MUL (2 clock cycles)	011
CALL	010
RETURN	110
JMP	100
JE (Jump if equal – Zero=1)	101
JNE (Jump if not equal – Zero=0)	111

Our custom microprocessor includes the following features:

- Three numbers of 16-bit registers: ax, bx, cx. These registers have specific 4-bit addresses. The addresses are 1101, 1110 and 1111, respectively.
- Twelve numbers of 16-bit registers as data memory. These registers are addressed by 0001, ..., 1100 using the above 4-bit register address (the address of *memory[0]* is 0001 and the address of *memory[11]* is 1100).
- The register address in the first operand in an instruction cannot be 0000, but if the second register address is 0000, then this operand is a constant value,

which will be determined by another 16-bit vector. Note that if the second register's address is not 0000, then the 16-bit constant value vector will be don't care.

- An 8-bit Program Counter (PC), which implies that we cannot use program source codes with more than 256 instructions.
- A zero flag (flip-flop), which refers to the equality/non-equality result after a comparison.
- A program memory to store the machine language source codes (instructions). This memory consists of 256 registers, in which each register refers to a specific instruction. In your source code, this memory should be programmed (loaded) by a *hex* file.
- All the instructions will be executed in one clock cycle, except for the multiplication, which will be executed in two clock cycles.
- Each instruction is 27-bits long ($inst[26:0]$). The first three bits ($inst[26:24]$) specify the instruction type based on TABLE I. The rest of the bits are defined as the following:

TABLE II: Instruction format ($inst[23:0]$)

Instruction Type	inst [23:20]	inst [19:16]	inst [15:0]
MOV	Reg#1 Address	Reg#2 Address	Constant value (This field is valid if Reg#2 address=0000)
ADD	Reg#1 Address	Reg#2 Address	Constant value (This field is valid if Reg#2 address=0000)
MUL	Reg#1 Address	Reg#2 Address	Constant value (This field is valid if Reg#2 address=0000)
CALL	Subroutine position (higher order 4-bits)	Subroutine position (lower order 4-bits)	Reserved
RETURN	Reserved	Reserved	Reserved
JMP, JE, JNE	PC next position (higher order 4-bits)	PC next position (lower order 4-bits)	Reserved

Note that in the MOV and ADD instructions, the result must be stored in Reg#1. However, in the 16-bit multiplication process, the 32-bit result will be stored in Reg#1 and Reg#2, where Reg#1/Reg#2 includes the higher/lower order 16-bits of the multiplication result. Another important issue in TABLE I is that if Reg#2-Address refers to cx (1111), then the second operand will be $memory[cx[3:0]]$ instead of cx. However, if the Reg#1-Address is 1111, then the first operand will be cx. Using this representation, we can realize the loops in our source codes. The exit instruction will also be determined by a 27-bit zero vector ($inst[26:0] = "000...0"$).

You are also required to design a power efficient processor. In order to fulfill this issue take the following techniques into consideration:

- *Operand Isolation*: A major percentage of switching power is dedicated to the arithmetic operations: addition and multiplication. In order to avoid such an issue you must design the arithmetic logic unit (ALU) so that the input vectors of the arithmetic blocks (adder and multiplier) become fixed, if the instruction is not addition or multiplication. This simple technique widely improves the

switching power consumption as it avoids the unnecessary transitions within the arithmetic blocks.

- *Clock-gating*: About 30% of the switching power is dedicated to the activity on the clock signal. In order to improve this problem the clock-gating technique will be applied to the registers, which do not load new data in specific clock cycles. You can fulfill this method by considering an input load signal for all of the registers.

Do your best to design a high performance area/power-efficient microprocessor. You should describe your system in Modelsim using Verilog language. It is highly recommended that you use structural and modular descriptions to achieve a more efficient implementation. Verify your system by loading the source code of the following arithmetic function:

$$f = \sum_{i=0}^3 memory[i] \times \sum_{i=4}^{11} memory[i] \quad (1)$$

In order to realize the above arithmetic function in your binary source codes, you should initially insert 12 sample values to the data memory by 12 numbers of MOV instructions. Then realize a loop for the sigma and benefit from cx to realize the variable i . You should put this loop in a subroutine, provide your binary source codes (instructions) in a hex file and then load it in your program memory. After verifying and simulating your design, synthesize it on Virtex4 using Xilinx tool.

Provide a complete report document and email it to omid_sarbishei@yahoo.com by 9:00am on Saturday 4th of Khordad. Your reports must cover the following issues:

- *Schematic of your design including the control unit's state machine and the datapath structure.*
- *Low-level assembly source codes for realizing Equation#1.*
- *Simulation results for the equivalent machine-level binary source codes, which will be loaded from a hex file. Use the Modelsim software for your simulations.*
- *Synthesis report on Virtex4 including area and maximum achievable frequency. What is the critical path in your design? How can you improve it?*
- *After estimating the maximum achievable frequency in your microprocessor, measure the CPU run-time to realize the arithmetic function in Equation#1 (Ignore the initial 12 numbers of MOV instructions, which initialize the data memory). Run the same assembly code on 8086 and compare your achieved performance with 8086's run-time.*

Regards,
O. Sarbishei