1 Tuning a Processor for FPGA Technology

This paper describes some methods to tune a soft-core processor, in this example a Java Optimized Processor (JOP), for the FPGA technology.

Table 1 shows the starting point for the enhancement with device utilization and maximum system speed of JOP in different FPGA families. JOP configuration consists of the core with an UART and the timer. As a reference NIOS, the RISC soft-core from Altera, is also included in the list. Version A is a minimum configuration without external memory. Version B adds an external memory interface, multiplication support and a timer.

Processor	Pipeline	FPGA	Resource [LC]	Memory [KB]	fmax [MHz]
JOP	4	EP1C6Q240C6	2024	3	94
JOP	4	EP1K50TC144-1	2413	3	47
JOP	4	XC2S300E-6	2634	3	32
NIOS A	5	EP1C6Q240C6	1828	6.2	120
NIOS B	5	EP1C6Q240C6	2923	5.5	119

Table 1: Starting Point

1.1 Find the Major Delays in the Design

Find the critical path through and insert registers. Doing this several times can give you a quick insight where your longest delays are and if the pipeline is balanced. In JOP it can be observed that the longest delays are in the translation from bytecodes to microcode addresses. An additional pipeline stage in this part increased the frequency by 13% to 106 MHz. The next hot spot is the 32-bit single cycle barrel shifter. Shift operation constitute about 0.26% in the dynamic instruction mix. Slowing down this operation to two cycles is not a big performance loss. At this point, we have gained a speedup of 18% from the original design with one additional pipeline stage. This design change increases only the CPI of the branch instructions from 4 to 5 resulting in 45 ns for a branch instead of 42.6 ns. Branch instructions constitute about 10.5% of the executed bytecodes (1.25% of them are unconditional). The next limiting design block is the calculation of the branch condition. Inserting a register changes the CPI for the conditional branches to 6, however the pipeline is still five stages long. Assuming all other instructions to be single cycle simplifies performance calculation. This is not the case in JOP, however all multicycle instructions gain direct from the increased clock frequency. The performance loss through the longer pipeline in the branch instructions gets less important when these instructions are correct counted. x shows that the solution with an added pipeline stage and a latency increase for the branch condition is 18% faster than the original design.

The next major delay is the microinstruction ROM. The address register is clocked with the inverted clock to keep the pipeline length short. If we change this to the normal clock, we end up with a maximum frequency of 145 MHz and an average instruction time of 11.2 ns. This is only a gain of 5% compared to the previous version. Further changes in the design with added registers show now the limiting components in the stack, the execution unit and the fetch unit again. Adding registers to these different stages result in frequencies of 147, 148, 154, 156 and 160 MHz. The version with 156 MHz shows now a performance degration with this simplified model. The 160 MHz version changes the execution time of local variables access to two cycles. However, these instructions constitute about 30.4% of all instructions. This results in an average execution time of 14 ns, longer than in the original design.

However, these changes are very radical and result in a too long pipeline. With this little increase of the clock frequency and the maximum delay spread over all pipeline stages we reached the limit of this technology. With the former solution, the pipeline is now well balanced.

$CPI_{4stages} = 0.105*4 + 0.895*1 = 1.315$
$CPU\ time_{4stages} = CPI * cycle\ time = 1.315 * 10.6\ ns = 13.9\ ns$
$CPI_{5stages} = 0.105*5 + 0.895*1 = 1.42$
$CPU\ time_{5stages} = CPI * cycle\ time = 1.42 * 9\ ns = 12.78\ ns$
$CPI_{regcond} = 0.0925*6 + 0.0125*5 + 0.895*1 = 1.513$
$CPU\ time_{regcond} = CPI * cycle\ time = 1.513 * 7.8\ ns = 11.8\ ns$
$CPI_{ROM} = 0.0925*7 + 0.0125*6 + 0.895*1 = 1.618$
$CPU\ time_{ROM} = CPI * cycle\ time = 1.618 * 6.9\ ns = 11.2\ ns$
$CPI_{156} = 0.0925*9 + 0.0125*8 + 0.895*1 = 1.83$
$CPU\ time_{156} = CPI * cycle\ time = 1.83 * 6.4\ ns = 11.7\ ns$
$CPI_{160} = 0.105*10 + 0.304*2 + 0.591*1 = 2.25$
$CPU\ time_{160} = CPI * cycle\ time = 2.25 * 6.25\ ns = 14.1\ ns$

Design Changes	Pipeline	fmax [MHz]
Original design	4	94
Register in the translation stage	5	106
Two cycle barrel shifter	5	111
Register branch condition	5/6	128
Use positive edge for ROM address register	6/7	145
Add a register to stack fill	7/8	147
Additional register in the decoding stage	8/9	148
Three cycle barrel shifter	8/9	156
Another register in the translation stage	9/10	154
Two cycle load of local variable	9/10	160

Table 2: Find the Major Delays in JOP

1.2 Changing the Fetch Logic

The inverted clock on the ROM address is used to get a microinstruction from the ROM in a single cycle. We can change the address register to the normal clock when feeding it with the pc multiplexer and not the pc output. However, the ROM address port is not connected to the reset signal. During reset, the first value written to the address register is pc+1, which is 1. Consequently, the fist microcode instruction is never executed. This change did not increase operating frequency (it is now 81 MHz), because the pc multiplexer now feeds two different registers adding interconnection delay. Now with the positive triggered address we can change the ROM to unregistered data output und use a discrete instruction register. The unregistered data output is now in the same pipeline level as the output from *bcfetbl*. The generation of the two signals *jfetch* and *jopdfetch*, which consumes 71 LCs and a significant delay of about 5 ns, can now be moved in the ROM. The ROM is now effective 10 bits wide. The multiplexer for the bytecode read address can be can be simpler than the multiplexer for *jpc*. A little delay can be saved by registering the address calculation for bytecode branches. However, to keep a pipeline length of 4, the lower 8 bits of the offset have to be read from the unregistered output of the bytecode RAM. Table 3 shows the results for different FPGA types after these optimizations. For the Xilinx version the generic ROM (rom.vhd) was used to be automatically placed in a block ram, instead of generating a Xilinx specific version (as in Table 1). This is probably the reason for the lower performance.

FPGA	Resource [LC]	Memory [KB]	fmax [MHz]
EP1C6Q240C6	2036	3.25	100
EP1K50TC144-1	2393	3.25	56
XC2S300E-6	2572	3.5	29

Table 3: Optimization of Instruction Fetch