INF2270, more detailed pipelining speed-up and a counter with synchronous load

.

March 19, 2012

Abstract

In this exercise we are having a closer look at pipelining speed-up refreshing and extending sequential circuits. The task is to extend the functionality of the synchronous counter that has been presented in the lecture.

Pipelined Processor Absolute Speed-Up

In the lecture that the speed-up of a pipelined vs. an equivalent serial processor for a specific program has been defined as the execution time of the serial- divided by the execution time of the pipelined architecture. We have made the simplyfing assumption that one stage in a n-stage pipeline consumes $\frac{1}{n}$ of the time that the serial processor needs to execute a complete instruction, i.e. that they both can run with the same clock speed. That is not the case in reality. If you look at the figure 8.11 in the compendium you may find it plausible that the storage of intermediate results (which is unnessecary in the srial implementation) comes at a time penalty. Thus, generally, the clock speed needs to be reduced somewhat in the pipelined architecture.

1. Thus if you consider a serial processor witha a clock frequency of 1GHz that needs 4 clock cycles per instruction (CPI) on average, what is its *throughput* (= instructions per second)?
2. If you consider an equivalent (i.e. same instruction set) pipelined architecture with 4 pipelining stages, which needs to reduce the clock speed because of the storing in three intermediate result registers by 10% and needs 1.2 CPI on average (not 1.0 CPI due to pipelining hazards), what will its troughput be?

Now, the main purpose of very long pipelines is to divide the operations into even smaller chunks, that actually require less execution time than a single clock cycle in a similar serial implementation. Thus, the clock speed can be increased. Thus ideally, a 16 stage pipelined implementation of a originally

present	in			next
S(2:0)	I(2:0)	CE	LD	S(2:0)
X	I	X	1	I
S	X	1	0	S+1
S	X	0	0	S

Table 1: State transition table for a counter with synchronous load and count enable

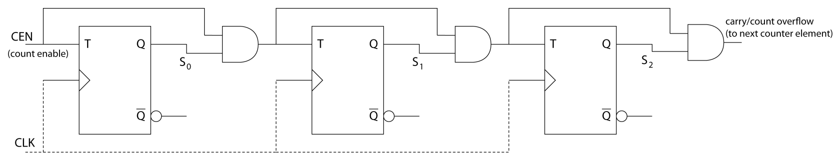

Figure 1: Synchronous counter from lecture with CE (but no LD)

serial processor that required 4CPI could actually run at 4 times the clock speed and achive a 16 time higher throughput. But again, there is a penalty per stage and there are pipelining hazards. Have a look at figures 4.32 and 4.33 a in the book. Let us assume that the time one needs to store an intermediate result and also the one to store the final result in a register is 10ps. The delay of one instruction in the original serial 1GHz processor is 4000ps, i.e. 3990ps to execute it in a combinational logic and 10ps to store the final result. The delay of a 16 stage implemention is thus about 4150ps and a single stage needs 3990ps/16+10ps=259ps.

1. So what's the 16-stage pipelined implementation's clock frequency and, assuming 1.0CPI, what is its throughput and how much faster is it than the original.
2. Assuming 1.2CPI, what are the numbers then?
3. Assuming storing in a (intermediate) register takes 20ps instead, what are the numbers (clock speed and throughput with 1.2CPI)?

Synchronous Counter with Synchronous Load

Consider the state transition table 1 which is an extension from the table of a 3bit counter presented in the lecture. Here, the states are given as unsigned integer variables instead of spelling out the binary numbers, but they are still binary numbers. Two control signals and a 3-bit input I have been added: count enable (CE) and synchronous load (LD). Remember that 'X' means 'don't care', i.e. the value of that variable does not matter for the outcome of the next state. The schematics of the synchronous counter in the lecture (figure 1) did already accommodate a control signal CE. Can you extend the circuit further to also implement the function of the control signal LD? Hint: use JK-flipflops instead of T-flipflops and use two 1-bit multiplexers and one inverter per JK-flipflop (in addition to the gates already used in the lecture script) to achieve one possible solution.