C74-6502 Microcode Pipeline

Post: July 3rd, 2016
http://forum.6502.org/viewtopic.php?f=4&t=3493&start=135#p46173

I was reviewing the critical-path analysis on the TTL CPU as part of getting ready to finalize the design when something jumped out at me. It turns out that of the 132ns on the critical-path, fully 51ns are spent fetching the next microcode instruction from ROM. Of course, this is to be expected, but nearly 40% of the total delay spent just waiting for microcode! Suddenly that seemed like way too much, so I decided to look for an alternative.

I had thought of replacing the ROMs with fast RAMs, but that requires cumbersome bootstrapping and in the end does not get at the root of the problem. Then it occurred to me that the datapath is completely idle while we wait for the microcode that drives it. And the reverse is also true - the ROMs are merely holding a value fixed when the datapath is active. If we can overlap these operations and pre-fetch the microcode, we would see a massive improvement in speed! Now this was too compelling to pass up, so despite being in a wrap-up mode, I found myself saving a version of the files and starting down the path to retro-fit a single-stage microcode pipeline into the design.

Dr Jefyll had in fact mentioned something akin to this early in this whole process, but frankly it went way over my head then. Somehow this now looked quite feasible, ignoring for the moment pcb density an layout considerations (and Dr Jefyll was once again on-hand to sort me out on a couple of issues!). After all, with the opcode in the IR, it seems simple enough to fetch the *next* micro-instruction and latch it into a Micro-Instruction Register (MIR) ready for use in the next cycle.

The first cycle after the opcode-fetch is clearly going to be a problem since we don't yet have the opcode for the pre-fetch. A quick scan of the 6502 and 65C02 instruction sets showed that nearly all opcodes perform exactly the same function in that initial cycle, namely to fetch the opcode operand with the address at PC and increment PC. That being the case, pre-loading the MIR with this "default" micro-instruction would satisfy most of the requirement, and leave only a few exceptions to handle. Also, re-working the microcode so that an "all-zeroes" control-word produced this function meant that a set of 74'273 registers could be cleared during the opcode-fetch to achieve the desired result.

As for the exceptions, there are only three:

1) We need to inhibit the incrementing of PC for any single-byte opcodes
2) We need to replace the current instruction with an opcode-fetch for 65C02 single-cycle NOPs
3) We need to generate an opcode-fetch in the next cycle of a branch if the branch is not taken

Let's look at each in turn:

First, single-byte opcodes have the form $x8 or $xA, meaning that a three input gate can catch them all. We don't have to worry about RTS($60) and RTI($40) since they replace the value of PC anyways. Similarly, the KIL($x2) illegal opcodes certainly need no protection from incrementing PC. This was looking very promising. It turns out the board already has logic for inhibiting +PC on the first cycle of an interrupt, so a couple of gates was enough to have it to do the same for single-byte instructions.

Next, single-cycle NOPs are $x3 and $xB opcodes on the 65C02, except that WAI($CB) and STP($DB) need to be excluded from this list (and included in the single-byte list above). A little more glue logic would be needed to trap these and, if detected, to replace the "default" micro-instruction with a fetch-opcode. That replacement requires only that we enable a couple of control bits - "IR.LD" to load the IR and "END" to reset the Q ROM-index to zero, all of which can be managed with just two OR gates on the outputs of the MIR.

Finally, interrogating the Branch Test Result (BTR) in the first cycle of a branch instruction was straight forward. All branches are of the form xxx1-0000 so once again fairly simple logic can detect them. If the branch test fails, a "FETCHOP" flip-flop can be set to generate a fetch-opcode micro-instruction in the next cycle (as above but for the next cycle this time).

With that done, I could then delete the first step of every opcode in the microcode, so that the micro-instruction for cycle 1 would then sit at index location 0 ready to be "pre-fetched". And with that, the basic engine was ready to go! The mechanism is surprisingly simple: clear the MIR on an opcode-fetch and latch the *next* micro-instruction at the end of every cycle. The micro-instruction is then ready to go at the start of the new cycle and the ROM delay is gone!

But, I needed also to deal with page boundaries during address calculation. This was accomplished before by incrementing the Q index by 2 if no page-boudary was crossed. Doing so had the effect of "skipping" the micro-instruction that adjusts the high-byte of the target address. Unfortunately, we don't discover the need to "skip" until very late in the cycle, at which point any progress in pre-fetching the microcode to that point is lost and we must begin again (which in effect puts the microcode fetch right back on the critical-path, this time at the end of cycles, rather than at the begining). Something had to be done ...

For branches, if adding the branch offset to PC does not cross a page, we can set the "FETCHOP" flip-flop for the next cycle and end the instruction early (just as we did above when a branch test failed). This can happen very late in the cycle with no adverse impact on the critical-patch. FETCHOP simply clears the MIR on the next cycle and then applies the required control bit transformations. There is no delay penalty incurred.

For indexed addressing modes, on the other hand, we need to insert an instruction into the stream to increment the high-byte, and then continue on to complete the memory read or write as the case may be. So, rather than "skipping" to avoid an operation, we introduce an additional cycle to execute it "on the fly" only if necessary. To do so, we can stall the Q counter so the same "low-byte increment" micro-instruction is re-fetched on the next cycle, but this time, it is transformed to a corresponding high-byte increment (DPL changed to DPH, etc.).

However, since these address operations go through the ALU, introducing gates at the outputs of MIR for those control signals would hit the critical-path. Instead the required micro-instruction is presented to the MIR *inputs* through 74'541 buffers with no delay penalty. The ROMs outputs which normally go to the MIR are simply disabled in that case so the MIR obligingly loads the inserted instruction at the right time and then goes back to regular programming thereafter. Once again, the required control-word ends up in the MIR ready to go at the beginning of cycle without delay, which is the objective.

And that, so far as I can tell, does the job. After some careful counting, I estimate the pipelined CPU will run with a cycle time of just 69ns, or at a 14.4MHz clock rate! That's nearly a 100% improvement over the roughy 7.5MHz the non-pipelined version could manage. So this certainly seems like a very worthwhile effort.

I've tested all of this on the Logisim model and all seems to be well. Frankly, I'm amazed at the substantial payback for what is, in the end, a fairly contained set of changes (which albeit do come at the cost of adding yet more ICs to already very dense boards). I'm hoping I have not made some as yet undiscovered error here as it does seem a little "too good to be true". Nevertheless, it sure is exciting to contemplate a TTL 6502 which runs as fast as its present day commercial counterparts!

Now I wonder how I will ever get this thing built

Post: August 1st, 2016
http://forum.6502.org/viewtopic.php?f=4&t=3493&start=135#p46616

Dieter calls it "the need for speed" and I guess I'm hooked.

Looking at it, the microcode architecture makes the ROM fetch-time the upper limit on speed. Using the microcode pipeline, the CPU is able to overlap the time required to fetch a microcode instruction with the processing time required for the rest of the circuitry. In perfect world, the two intervals match, resources are used optimally, and the CPU is going as fast it can.

Now, it takes 5ns to bump a counter to index into the ROMs, 45ns for the ROMs to respond and 2ns setup-time to latch the data into the Micro Instruction Register. That's 52ns, but just for fun, let's call it an even 50, and set 20MHz as the ceiling on this CPU design. The question is how close can we get to this theoretical maximum? The microcode pipeline delivered 69ns. Let's see where we can get to with a few more optimizations:

First up, CBT Parts: some time ago, Dr Jefyll suggested these in the context of bi-directional busses. I ended up not using them at the time, but I well-remembered their amazing propagation delay - 0.25ns tpd! Take the 74CBT3245 Octal FET Bus Switch, for example. The enable-time is roughly the same as the equivalent 74AC245 part. But once enabled, the CBT version is effectively a wire and lightning fast. The key (as Dr Jefyll explained) is to use it in situations where the "enable" signal is available a few nanoseconds before the data is ready to pass through the switch.

As it turns out, the outputs of the ALU are one such situation. Three distinct buffers are used to select between a shift, a binary or a BCD result. The control signals are ready early in the cycle, but the data goes through the ALU and arrives much later. So, the FET switch can be set up well in advance and the data flies through when it arrives. Jeff also suggested that a 74CBT3251 might replace a 74AC151 under similar conditions. Luckily, there is a selector on the board that's perfect, a '151 used for the V flag evaluation, and it's also on the critical path. The total savings from these two things: 12.5ns.

Now, by far the slowest path through the CPU is the BCD-adjust logic. However, the BCD operation can be spread over 2 cycles, so it's really the ALU binary path that's critical. As it happens, the two paths share the high-nibble adder, which saves a chip, but forces the binary carry through unnecessary logic. Inserting a dedicated BCD adder separates the paths, and removes a further 6.5ns from the critical path. Add that to our previous balance and the ALU is 19ns faster - right on target for our theoretical limit!

Now, as much fun as this is, a grain of salt is in order. The tolerances here are small, and tiny variances can throw the whole thing off - and by a wide margin. The savings above are significant, but the absolute numbers will only bear out once the CPU is built. For now, I'm very happy that the clock-rate might be pushing 18MHz. At the same time, 1-cycle NMOS BCD is also faster, and if enabled may get as high as 16MHz. Time will tell. 

Cheers for now,
Drass.