Getting Rid of µCore's 2-phase Execution Cycle

Klaus Schleisiek - kschleisiek at freenet.de

µCore_1 did have a two phase instruction execution cycle, because the internal blockRAMs in FPGAs do have an internal address register that needs to be set first before data can be read. Each phase lasts for at least one clock cycle:

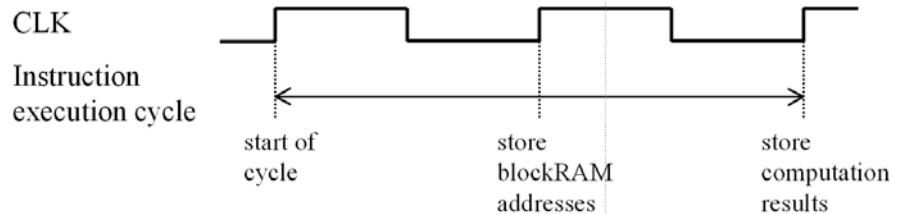
1st phase: All address computations are done for µCore's three memory areas:

1. Program memory
2. Data memory and return stack
3. Data stack

At the end of the 1st phase, these addresses may be registered in the respective memory areas depending on an instruction's need.

2nd phase: Now the memory areas may be read and used for computing results as well as reading the next instruction. At the end of the 2nd phase, results can be stored on the data stack, in the data memory, on the return stack, and in the instruction register.

This is a waste of computing time for the majority of instructions that do not need to read any memory except for the program memory.



I got rid of this unfortunate scheme by splitting up a random blockRAM memory access into two subsequent, indivisible instructions. Along the way, I invented a generic 'instruction chaining mechanism', which can readily be used for indivisible read-modify-write instructions like +!.

Program memory

The trick for the program memory is to get rid of the instruction register. At the end of every µCore_2 execution cycle the next instruction's address gets registered in the program memory's address register. In the next cycle the output data of the program memory constitutes the next instruction. The instruction register of µCore_1 is no longer needed. :-)

Data stack

The trick for the data stack memory is simple: The data stack pointer always points at the the last item pushed into the data stack memory. This implies that its internal address register is set to said item and therefore, it is ready to be read back in the next cycle already. No random access will ever happen, because the data stack memory solely serves as a stack that can only be pushed or poped.

Data memory and return stack

In µCore, both the data memory and the return stack reside in different regions of the same memory. Each memory read must be split up into two instructions: The 1st instruction sets the data memory's address register, and the 2nd instruction can then read the data and operate on it. These two instructions need to be indivisible or else an interrupt may overwrite the memory's address register before executing the 2nd instruction.

Getting Rid of µCore's 2-phase Execution Cycle

The 1st instruction (e.g. @) will be compiled by µCore's cross compiler. When it is executed, it will schedule the 2nd instruction for execution in the next cycle without advancing the program counter.

To this end I invented a general mechanism to chain up a series of indivisible instructions.

The following table lists µCore_2's two-cycle instructions:

instruction	2nd cycle
r>	store memory data into TOR
rdrop	store memory data into TOR
exit, iret	store memory data into TOR
?exit	only executed when TOS /= 0: store memory data into TOR
next	only executed when finishing a FOR ... NEXT loop (TOR = 0): store memory data into TOR
@	store memory data into TOS
+!	write (memory data + NOS) back into memory
I	store the sum of TOR and data memory (2nd return stack item) into TOS
IF	in the 1st cycle, the branch address is dropped, in the 2nd cycle the flag as well

These two cycle instructions constitute about 20 % of the instructions executed by a typical program. Therefore, total throughput of µCore_2 has increased by about 65 % compared to µCore_1.

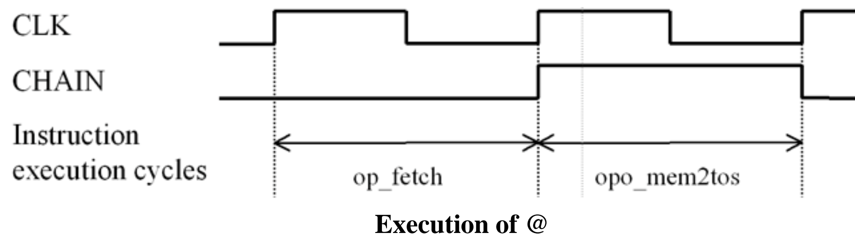
Instruction chaining mechanism

inst is an 8-bit register that delivers the next instruction when **chain** = **true** (see below).

chain is a flip-flop, which must be set when the next instruction should be read from **inst** instead of the program memory. When **chain** is being set, the program counter will not be advanced.

Interrupts will be suppressed as long as **chain** = **true**.

Given this mechanism, an arbitrary number of single instructions may be chained up for application specific opcodes.



Execution of @

Reset

After a reset, execution should start at address zero. Therefore, the processor must be set up to execute a noop while fetching the first instruction from address zero. This can be realised as follows:

```
IF  reset = true  THEN
   inst <= op_NOOP;
   chain <= true;
   program_memory_addr <= 0;
END IF;
```