Pipeline Analysis using Model Checking

By Andreas Engelbredt Dalsgaard, Mads Christian Olesen and Martin Toft
{andrease,mchro,mt}@cs.aau.dk

In a pipelined processor, the processor's data path is split into parts that are able to work in parallel. These parts are called pipeline stages and together they make up the processor's pipeline. As the circuitry of an individual stage is smaller than the circuitry of the entire, undivided data path, a stage is able to complete its work faster. This, combined with the fact that the stages work in parallel, enables the processor to do more work per time unit when supplied with a stream of instructions. Ultimately, the execution time for programs is reduced substantially, and the effects of pipelines are therefore important to consider in connection with worst-case execution time (WCET) analysis.

A pipeline is, however, difficult to model, as the jobs carried out in its stages are not completely independent. We will illustrate the behaviour using ARM assembly code and the specific five-stage pipeline found in the ARM9TDMI processor core. The pipeline is depicted in Figure 1.

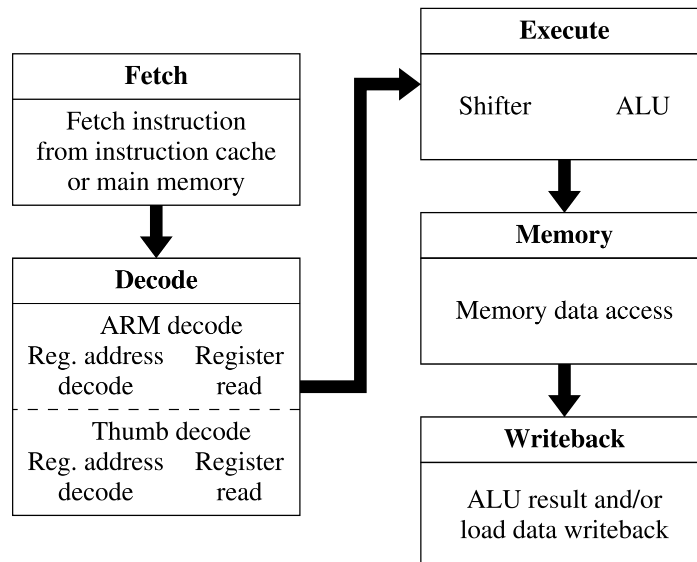

Figure 1: The five-stage pipeline in the ARM9TDMI processor core [1].

The following instruction sequence is executed under the assumption that each instruction can be fetched from an instruction cache with the speed of one instruction per clock cycle:

```
LDR R0, [R1]
ADD R2, R0, R1
```

The `LDR` (Load Register) instruction loads the value of the 32 bit memory cell pointed to by register `R1` into register `R0`. This value is then used by the `ADD` (Add) instruction that adds the values in `R0` and `R1` and stores the result in `R2`. Because arithmetic instructions are carried out in the pipeline's execute stage and load and store instructions are carried out in the memory stage, the instruction sequence causes a pipeline stall, which is illustrated in Figure 2. The `ADD` instruction must stay in the decode stage for an extra cycle, while it waits for `LDR` to be handled in the memory stage.

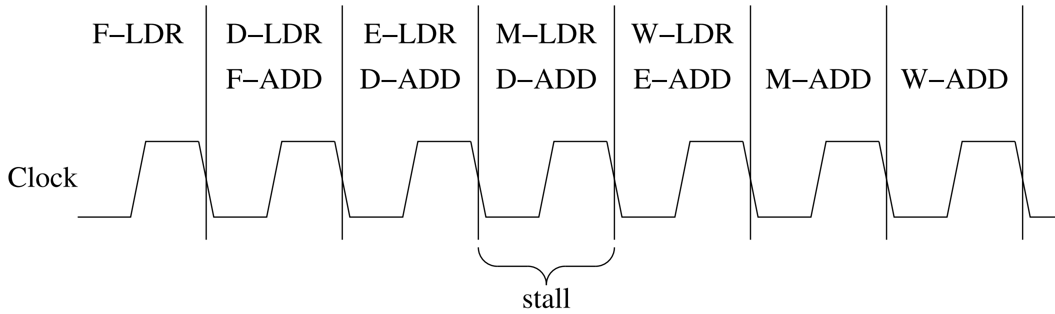

Figure 2: Timing diagram for a single load stall [2].

Another example is the following instruction sequence, where `LDRB` (Load Register Byte) loads `R0` with the 8 bit value at the memory address calculated by adding the constant value 1 to the memory address contained in `R1` [3]:

```
LDRB R0, [R1,#1]
ADD R2, R0, R1
```

As the requested byte is not at a 32 bit word boundary, it is necessary to shift the loaded word. Compared to the first example, the shift operation stalls the pipeline for an extra cycle, and `ADD` therefore spends three cycles in the decode stage (two of those due to stall).

The ARM instruction set also features the instruction `LDM` (Load Multiple) for loading multiple registers with values from memory. In the following instruction sequence, `LDM` loads `R1`, `R2` and `R3` with values from memory starting from the memory cell pointed to by `R12` [3]:

```
LDM R12, {R1-R3}
ADD R0, R1, R2
```

Each individual load done by `LDM` requires a trip through the circuitry of the memory stage and might stall subsequent instructions. The specified registers are loaded in sequence, from the lowest-numbered to the highest, and stalled instructions move on immediately when their dependencies are met. The `ADD` instruction uses `R1` and `R2`, and consequently it cannot be executed before they have been loaded. Figure 3 shows that `ADD` must stall for two cycles in the decode stage, while `R1` and `R2` are loaded by `LDM` in the memory stage.

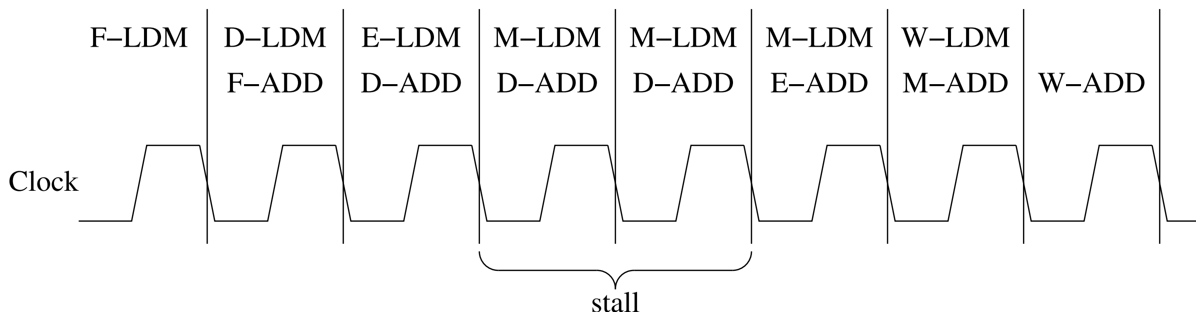

Figure 3: Timing diagram for a multiple load stall [2].

A variant of the multiple load example is the following instruction sequence:

```
LDM R12, {R1-R3}
ADD R0, R1, R3
```

Rather than adding `R1` and `R2`, `ADD` now adds `R1` and `R3`. The effect is that `ADD` must stall for three cycles in the decode stage, since `R3` is the last register loaded by `LDM` in the memory stage.

Using model checking, we are able to conduct a combined cache and pipeline analysis that takes the complex pipeline behaviour demonstrated above into account. By bringing together the OpenMoko ARM9 Toolchain [4], the objdump front-end Dissy [5], the model checker UPPAAL [6] and a compiler (written by us) that translates binary object files into UPPAAL models, we are able to determine safe and relatively sharp WCETs for programs.

References

[1] Simon Segars. The ARM9 Family - High Performance Microprocessors for Embedded Applications. In *ICCD '98: Proceedings of the International Conference on Computer Design*, page 230, Washington, DC, USA, 1998. IEEE Computer Society.

[2] ARM9TDMI Technical Reference Manual.
`http://infocenter.arm.com/help/topic/com.arm.doc.ddi0180a/DDI0180.pdf`

[3] Peter Knaggs and Stephen Welsh. ARM: Assembly Language Programming.
`http://www.arm.com/miscPDFs/9658.pdf`

[4] `http://wiki.openmoko.org/wiki/Toolchain`

[5] `http://code.google.com/p/dissy/`

[6] `http://uppaal.com/`