ARMv8 Simulator

Harshit Patel, Swarnadeep Saha

August 26, 2015

Introduction

ARMv8 simulator is developed to simulate limited instruction set of ARMv8 architecture in pipelined fashion. It is equipped with a debugger having a set of very useful functionalities.

Assumptions

1. There are five stages in processor pipeline, namely Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MA), Write Back (WB).
2. Instruction cache and data cache are saperate functional units.
3. All attempts to access data from instruction cache and data cache are hits.
4. All memory words are 0s unless something is stored into memory through program.
5. Functional unit to shift bits in operands of instruction is implemented as combinational circuit and has 0 latency.
6. Total 3 cycles are required to fetch an instruction from instruction cache and **2 extra cycles are not considered stall cycles.**
7. There are two separate register files:
   (a) Interger Register File (intRF) of 32 64-bit registers.
   (b) Float Register File (floatRF) of 32 128-bit registers.
8. Operands can be forwarded from EX/MA interstage register and MA/WB interstage register in operand forwarding mode.
9. Switching happens every cycle in a functional unit for the duration it is active.

Source Files

All source files are in [project_path]/ARMv8/ folder. Majority of source files have names of the format [stage_function]_[instruction_class].py where [stage_function] and [instruction_class] are as below:

- [stage_function]:
  - **opfetch**: These files contain code for decoding and operand fetching for instructions of [instruction_class] in ID stage.
  - **executor**: These files contain code for executing instructions of [instruction_class] in EX stage.
  - **memaccess**: These files contain code for memory read/write operations for instructions of [instruction_class] in MA stage.

  - **writeback**: These files contain code for writing results in destination registers for instructions of [instruction_class] in WB stage.

- [instruction_class]:
  - **ALU**: Implements CLS and CLZ instructions.
  - **FP_addSub**: Implements scalar and vector varients of FADD and FSUB instructions.
  - **FP_maxMin**: Implements scalar and vector varients of FMAX and FMIN instructions.
  - **adc**: Implements ADC instruction.
  - **addSub**: Implements ADD, ADDS, SUB and SUBS instructions.
  - **bitwise_shift**: Implements instructions.
  - **branch**: Implements B, BCOND, BL, BR, BLR, RET, CBZ and CBNZ instructions.
  - **conditional**: Implements CSET, CSINC, CNEG, CSNEG, CSINV and CSINV instructions.
  - **loadStore**: Implements LDR, LDRB, LDRSB, LDRH, LDRSH, LDRSW, LDP, STR and STP instructions.
  - **logical**: Implements AND and ANDS instructions.
  - **misc**: Implements ADR, ADRP and NOP instructions.
  - **mov**: Implements MOV, FMOV and FMOV general instructions.
  - **moveWide**: Implements MOVK, MOVN and MOVZ instructions.
  - **mulDiv**: Implements UMULL, UDIV and SDIV instructions.
  - **rotate**: Implements ROR instruction.
  - **shift**: Implements LSL, LSR and ASR instructions.

There are several other source files as follows:

- **armdebug.py**: Implements debugger.
- **utilfunc.py**: Implements utility functions for setting/getting registers from register file, storing/loading memory locations, integer and floating point addition/subtraction etc.
- **const.py**: Implements global constants and flags used for synchronization and decision making.
- **config.xml**: Contains hardware specifications of processor.
- **config.py**: Parses config.xml to read hardware specifications of processor into simulator.